Hacia una biblioteca BLAS realmente portable entre diferentes tipos de aceleradores

Eduardo Rodriguez-Gutiez,[1] Arturo Gonzalez-Escribano,[2] Diego R. Llanos [3]

Resumen— **Las rutinas de álgebra lineal BLAS son ampliamente utilizadas en aplicaciones científicas de todo tipo. Existen implementaciones específicamente optimizadas para diferentes tipos de plataformas de cómputo incluyendo aceleradores. Por ejemplo, la implementación contenida en la biblioteca Intel MKL, aparte de ejecutarse en CPUs, incluye versiones para Xeon Phi, mientras que la biblioteca cuBLAS está especialmente diseñada para GPUs de NVIDIA.**

Sin embargo, los mecanismos para gestionar la memoria utilizada por las estructuras de datos sobre las que se realiza el computo son diferentes en cada implementación, así como algunos mecanismos relacionados con las llamadas y el paso de parámetros.

En este artículo presentamos una interfaz única para BLAS, integrada en un modelo de programación heterogénea (Controllers) que soporta grupos de núcleos de CPU, aceleradores Xeon Phi o GPUs de NVIDIA de forma transparente para el programador.

Con esta propuesta es posible construir programas portables basados en rutinas BLAS, que se ejecutan en diferentes tipos de aceleradores cambiando simplemente un parámetro de inicialización. Nuestra propuesta explota internamente la biblioteca específica para cada tipo de dispositivo. Las diferencias en sus interfaces y en los mecanismos externos para gestionar la memoria de los dispositivos, minimizando transferencias, son transparentes para el programador. Los resultados experimentales muestran que nuestra abstracción no introduce pérdidas de rendimiento significativas.

Palabras clave— **BLAS, Programación heterogénea, Controllers, Aceleradores, GPU, Xeon Phi, MIC, CUDA.**

I. Introducción

La especificación BLAS, muy conocida en el ámbito de la computación científica, contiene la definición de tres conjuntos (denominados niveles) de rutinas matemáticas de bajo nivel que permiten realizar operaciones vector-vector, vector-matriz y matriz-matriz, respectivamente. Existen ciertas implementaciones de BLAS que permiten obtener un mejor rendimiento cuando se ejecutan sobre un hardware específico. Algunos ejemplos de éstas son cuBLAS[1], cuyas funciones están optimizadas para GPUs que soporten CUDA, y la implementación que se encuentra como parte de Intel MKL[2], que permite acelerar su ejecución bajo plataformas Intel, entre las que se incluyen los coprocesadores Xeon Phi.

Uno de los problemas asociados al uso de coprocesadores está en el elevado coste de mover los datos necesarios desde el host al dispositivo. Para una secuencia concreta de llamadas a rutinas BLAS, escoger los momentos en los que se sincronizan y mueven datos entre el host y el dispositivo es importante. Los mecanismos para realizar estas transferencias varían entre las diversas implementaciones de BLAS, e incluso dentro de cada implementación existen diferentes opciones con diferentes grados de transparencia y responsabilidad de cara al programador. Las bibliotecas MKL de Intel y cuBLAS/nvBLAS de NVIDIA proveen distintos mecanismos de *offloading*, ya sea totalmente automático, asistido por el compilador (*Compiler-Assisted Offloading*, CAO), o con transferencias programadas manualmente. A mayores, aunque la interfaz de las funciones BLAS pueda parecer uniforme entre las diferentes implementaciones, en la práctica existen pequeñas diferencias que conviene tener en cuenta. Por ejemplo, los mismos parámetros pueden aparecer por valor o por referencia en diferentes bibliotecas, diferentes nombres o constantes se usan para indicar el mismo tipo operaciones o datos, e incluso algunas bibliotecas pueden tener parámetros extra específicos de la propia implementación, tales como una estructura de datos que representa el contexto de la biblioteca en cuBLAS.

En trabajos previos presentamos el modelo denominado *Controller*[3], que permite la implementación de algoritmos concretos en unidades de cómputo o kernels. Cada kernel se identifica por un nombre e interfaz únicos, pero se pueden declarar diferentes implementaciones del mismo kernel. Desde una genérica que utiliza una abstracción que permite ejecutarlo en cualquier dispositivo, hasta una implementación específicamente optimizada para cada tipo de plataforma, o incluso para cada familia de arquitectura. Un controlador se asocia a un dispositivo en su creación. Permite asociar estructuras de datos a su contexto e invocar la ejecución de kernels por nombre. El controlador escoge la implementación del kernel invocado más adecuada para la plataforma o dispositivo al que está asociado. La primera propuesta soportaba como dispositivo tanto un grupo de núcleos de CPU, como GPUs de NVIDIA[3]. Posteriormente se añadió el soporte para aceleradores Intel MIC (Xeon Phi)[4]. Internamente los Controllers utilizan OpenMP y CUDA para la coordinación de tareas.

En este trabajo se presenta una capa software de abstracción entre diversas implementaciones de BLAS, utilizando como soporte la biblioteca de Controllers. Esta propuesta permite al usuario programar códigos con secuencias de llamadas a las rutinas de BLAS con una interfaz uniforme y seleccionar con un único parámetro el hardware destino en el que se ejecutará el código. Internamente, la biblioteca escogerá la mejor implementación disponible para el

[1]Dpto. de Informática, Univ. Valladolid, e-mail: eduardo@infor.uva.es

[2]Dpto. de Informática, Univ. Valladolid, e-mail: arturo@infor.uva.es

[3]Dpto. de Informática, Univ. Valladolid, e-mail: diego@infor.uva.es

hardware seleccionado, utilizando llamadas a biblioteca BLAS específicamente optimizadas para dicho hardware, provistas por el fabricante o por terceras partes. Nuestra propuesta combina portabilidad de código y de rendimiento.

El presente artículo está estructurado en varias partes. La sección 2 presenta algunos trabajos relacionados. En la sección 3 se repasan algunas características de las herramientas utilizadas. La sección 4 detalla las soluciones utilizadas en nuestra propuesta. En la sección 5 se describe el trabajo experimental para validar la propuesta. Finalmente, en la sección 6 se discuten las conclusiones y el trabajo futuro.

II. Trabajo relacionado

La idea de ejecutar las funciones de BLAS en un entorno heterogéneo no es nueva. Por ejemplo, existe una biblioteca BLAS pensada para ser ejecutada de forma paralela denominada PBLAS, presentada por Choi *et al.*[5] y que puede ser descargada desde[6]. Una variante de ésta, llamada HeteroPBLAS[7] y creada por Manumachu *et al.*, es capaz de ejecutar los algoritmos de BLAS explotando el paralelismo mediante memoria distribuida con HeteroMPI en clústeres heterogéneos, lo que añade la capacidad de balanceo de carga. Sin embargo, HeteroPBLAS no es capaz de explotar las capacidades de cómputo de los coprocesadores conectados a los nodos y únicamente utiliza los cores de CPU.

Una solución alternativa a la presentada aquí es el conjunto de bibliotecas MAGMA[8], que contiene versiones optimizadas de las funciones de BLAS y LAPACK, y está disponible tanto para CPUs, CUDA, OpenCL[9] y Xeon Phi[10]. La principal diferencia que presenta la solución descrita en este trabajo respecto a MAGMA es que no pretende generar una implementación eficiente por cada coprocesador existente, sino que utiliza en su lugar las bibliotecas proporcionadas por las propias compañías que desarrollan el hardware. De esta forma se evitan los problemas derivados de actualizar permanentemente los algoritmos de BLAS para cada nueva arquitectura[11], o de tener que incluir nuevos algoritmos que puedan aparecer para arquitecturas antiguas.

Finalmente, dado que es posible programar tanto para CPUs, GPUs y Xeon Phi utilizando OpenCL, también cabe citar la biblioteca clBLAS[12], que forma parte de clMathLibraries, que a su vez está siendo desarrollada por AMD.

III. BLAS y Controllers

A. Implementaciones especializadas de BLAS

La especificación BLAS, muy conocida en el ámbito de la computación científica, contiene la definición de tres conjuntos (denominados niveles) de rutinas matemáticas básicas que permiten realizar operaciones vector-vector, vector-matriz y matriz-matriz, respectivamente. Existen ciertas implementaciones de BLAS que permiten obtener un mejor rendimiento cuando se ejecutan sobre un hardware específico. Algunos ejemplos de éstas son cuBLAS[1], cuyas funciones están optimizadas para GPUs que soporten CUDA, y la implementación que se encuentra como parte de Intel MKL[2], que permite acelerar su ejecución bajo plataformas Intel, entre las que se incluyen los coprocesadores Intel MIC Xeon Phi.

Cada rutina tiene normalmente cuatro versiones para cuatro tipos de datos diferentes. El tipo de datos se refleja en el nombre de la función con una `s` para reales en punto flotante de precisión simple (p.ej. SGEMM); `d`, para reales en punto flotante de precisión doble (DGEMM); `c`, para números complejos en punto flotante de precisión simple (CGEMM) y finalmente `z`, para complejos en punto flotante de precisión doble (ZGEMM). Normalmente se suele emplear un asterisco `*` o algún otro símbolo en lugar de la letra correspondiente al tipo de datos cuando se cita el nombre de una rutina de forma genérica.

En cada nivel de BLAS se definen diferentes rutinas. En nivel 1 encontramos rutinas que hacen operaciones escalar-vector o vector-vector. Por ejemplo, la rutina *AXPY, que multiplica los elementos de un vector por un escalar, o I*AMAX, que para un vector devuelve el índice de la celda con el valor más alto del mismo. En nivel 2 encontramos rutinas con operaciones vector-matriz, como por ejemplo *GEMV, que devuelve el resultado de multiplicar una matriz por un vector. Finalmente, en nivel 3 hay operaciones matriz-matriz, como *GEMM que permite obtener el resultado de la multiplicación de dos matrices.

En cada implementación especializada para coprocesadores encontramos diversos mecanismos de *offloading*, que son las operaciones de movimiento de datos desde la jerarquía de memoria del host al dispositivo, o viceversa. La biblioteca MKL, por ejemplo, permite hacer *offloading automático*[13] de ciertas funciones a Xeon Phi. En este caso, una llamada a función de BLAS implica de forma transparente el movimiento de datos de entrada al coprocesador y la recuperación de los resultados moviéndolos al host. Este mecanismo, si bien debe ser habilitado por el usuario mediante la llamada a una función o la definición de una variable de entorno (`MKL_MIC_ENABLE=1`), su activación para una llamada a BLAS es decisión de la implementación. Sólo se realiza cuando ésta juzga que el tamaño de datos a procesar es lo suficientemente grande como para que el movimiento al coprocesador compense[14]. En caso contrario la rutina se ejecuta en el host. Aunque es una funcionalidad automática y transparente, sólo está disponible para para un conjunto muy reducido de funciones de BLAS (*GEMM, *TRSM, y *STRMM), las funciones de factorización LU (*GETRF, *GETRFNPI), QR (*POTRF) y Cholesky (*GEQRF) y la reducción a forma tridiagonal (SYRDB) de LAPACK[15], [16], [17]. El encadenamiento de secuencias de llamadas a estas funciones pueden producir múltiples movimientos de datos innecesarios, ya que aunque los resultados de una rutina se vayan a utilizar como entradas en otra posterior, el offloading automático siempre transfiere las entradas y salidas de cada rutina de forma independiente.

Otra opción consiste en el uso de Intel Language Extensions for Offload (LEO). Se trata de unas anotaciones del compilador de Intel (pragmas) que permiten realizar comunicaciones explícitas de datos entre host y dispositivo en los momentos deseados. Esto permite el mayor grado de control al programador, a costa de un mayor esfuerzo por su parte para determinar los movimientos óptimos y codificarlos, aparte de generar un código menos portable.

Como parte de las posibilidades que ofrece LEO está el modo de offloading asistido por el compilador (CAO), en el que el propio usuario es responsable de definir qué funciones o grupos de éstas se ejecutarán en el coprocesador y cuáles en el host, quedando los detalles del movimiento parcialmente ocultos. Es, por tanto, una solución más general que la anterior. Todas las funciones BLAS de MKL pueden ser ejecutadas mediante esta técnica y permite reutilizar datos entre llamadas a rutinas sin movimientos innecesarios Sin embargo, las decisiones son responsabilidad del programador, y los movimientos se sincronizan con los comienzos o finales de rutinas.

NVIDIA tiene una solución similar al *offloading automático* de MKL, mediante una biblioteca llamada nvBLAS, que se ejecuta sobre su implementación especializada de BLAS para sus GPUs, denominada cuBLAS. Al igual que en el caso anterior, verifica si la transferencia mediante el bus PCI compensa el tiempo estimado de ejecución. De nuevo al igual que en MKL, sólo está disponible para un conjunto reducido de funciones (*GEMM, *SYRK, (C,Z)HERK, *SYR2K, (C,Z)HER2K, *TRSM, *TRMM, *SYMM y (C,Z)HEMM)[18]. Si no se utiliza esta opción, las transferencias de memoria entre host y GPU deben ser gestionadas con las llamadas habituales a la biblioteca de CUDA. Esto es equivalente a la utilización de Intel LEO en cuanto a mayor control, mayor esfuerzo de desarrollo y menor portabilidad.

En cuanto a diferencias de interfaz podemos destacar que en la segunda versión del API de cuBLAS se utiliza un nuevo parámetro *handle* que es específico de esta implementación. Se trata de una estructura que contiene información sobre el contexto de la biblioteca, y que debe ser creada, pasada como primer parámetro en cada llamada a esta biblioteca, y posteriormente destruida. Otras diferencias menores aparecen por ejemplo en la forma de pasar ciertos parámetros. Por ejemplo, los escalares alfa, beta, etc., se pasan como puntero en todas las llamadas a cuBLAS, mientras que en Intel MKL se pasan por valor. Otro problema reside en los diferentes valores que cada implementación asigna a las constantes o tipos de dato que definen operaciones, tales como por ejemplo `cublasOperation_t` y `CBLAS_TRANSPOSE`, que especifican si se debe realizar o no la transpuesta de una de las matrices de entrada antes de realizar la operación. Relacionado con ésta, algunas bibliotecas como CBLAS permiten definir si la matriz de entrada está en *row-major order* (común en lenguajes como C) o en *column-major order* (empleado por programas en lenguaje Fortran).

B. Controllers

La biblioteca de Controllers[3], [4] introduce una entidad abstracta, el controlador, que se asocia en su creación a un dispositivo. Este puede ser una GPU, una acelerador Intel MIC Xeon Phi, o un grupo de núcleos de CPU que se manejan de forma transparente como un acelerador.

El controlador soporta funcionalidades para asociar/desasociar una estructura de datos a su contexto, crear/destruir estructuras de datos dentro de su contexto que no tienen reflejo en el host, y lanzar kernels (rutinas de trabajo) en el dispositivo.

Las llamadas a kernels reciben como parámetros valores, o referencias a estructuras de datos de su contexto (asociadas o creadas). La asociación de una estructura de datos al contexto de un controlador implica que antes de ser utilizada por un kernel los datos del host se copiaran a la imagen que se crea en el contexto del dispositivo. En el momento de desasociarla, si la estructura ha sido modificada en el dispositivo, los datos actuales se copian en la imagen original del host. Las copias hacia el dispositivo pueden ser *eager* o *lazy*, realizándose en el momento de la asociación o retrasándose hasta que son necesarias.

El controlador es responsable de gestionar la cola de peticiones de ejecución de kernels, y manejar las transferencias de memoria de forma transparente. Internamente, los controladores utilizan las bibliotecas o funcionalidades nativas que proveen los vendedores de las plataformas, o las bibliotecas de terceros más eficientes para cada plataforma. Por tanto proveen de una capa de abstracción para desarrollar programas portables, consiguiendo la máxima eficiencia en la gestión del dispositivo gracias a explotar los mecanismos nativos de cada uno.

Para definir kernels, la biblioteca de controladores utiliza un sistema de declaración genérica de interfaz donde se detalla el nombre, número de parámetros, tipos, nombres y rol de entrada/salida. Un kernel con un nombre único puede ser declarado varias veces con un identificador de plataforma. Existe un identificador genérico que sirve para declarar kernels que se pueden ejecutar en cualquier plataforma. El controlador provee de un espacio de identificación de índices de thread similar al de CUDA u OpenCL, pero que asocia los threads en orden row-major, lo que permite portabilidad entre dispositivos. Se provee de un interfaz de acceso a los elementos de estructuras de datos único. Por tanto el programador puede definir kernels sencillos y portables. La especificación se hace en grano fino. El controlador es el responsable de agrupar los threads en bloques (para GPUs) o de generar tareas de grano grueso que recorren el espacio de índices con bucles (en OpenMP) para su ejecución eficiente.

Sin embargo, también se permite declarar kernels específicos para plataformas y arquitecturas, donde se pueden incluir estructuras y primitivas propias del modelo de programación interno. Por ejemplo, los kernels declarados para GPUs pueden utilizar sintaxis y primitivas de CUDA para declarar y usar

memoria compartida, sincronizar los threads del bloque, etc. Esto permite ir construyendo una biblioteca de kernels especializados que el controlador utilizará priorizándolos sobre los kernels genéricos cuando la arquitectura asociada sea la adecuada. Parámetros de lanzamiento tales como los tamaños de bloque y grid para GPUs, se seleccionan automáticamente en función de una declaración de propiedades del código suministrada por el programador y un modelo de predicción interno. En caso de no tener una caracterización específica se utilizan valores por defecto que aseguran máxima ocupación de los cores de la GPU.

Para conseguir una uniformidad en la gestión de las estructuras de datos y en el paso de parámetros, el modelo de controladores se apoya en la biblioteca Hitmap. Esta biblioteca permite definir estructuras de datos con dominios de índices, tales como arrays o grafos. Cada estructura se representa con una estructura denominada *HitTile* que contiene meta-datos y punteros a la zona de memoria con los datos. La biblioteca incluye un interfaz unificado para la declaración de espacios de índices, creación, destrucción, y acceso a los datos de un HitTile. Los parámetros de los kernels en el modelo de controladores son siempre tipos básicos pasados por valor, o HitTiles de cualquier tipo base.

IV. Propuesta de solución

En esta sección se detalla la solución propuesta para conseguir una biblioteca portable y eficiente de BLAS a través del modelo de controladores. La biblioteca de controladores nos facilita una forma de construir un interfaz único para kernels con diferentes implementaciones especializadas para diferentes plataformas. El controlador es el responsable de escoger la implementación adecuada para la arquitectura a la que se asocia en su construcción. En caso de no existir una apropiada, el controlador utilizará la implementación genérica no especializada.

Nuestra propuesta se basa en extender el mecanismo de los controladores para implementar kernels con código que se ejecuta en el host, para añadir tres nuevos tipos de implementaciones de kernels, uno para cada tipo de dispositivo soportado. Los tipos se denominan *libCPU*, *libXPhi* y *libGPU*.

Estos nuevos tipos de kernels se declaran como cualquier otro kernel, pero usando estos nombres como identificadores de arquitectura. En lugar de implementar un código especializado que se lanza y ejecuta en el dispositivo, implementan llamadas a funciones de bibliotecas externas. Por tanto su código se ejecuta en el host, pero la llamada a la biblioteca nativa dispara la ejecución de una rutina en el dispositivo. En el caso de grupos de CPUs (libCPU) o Intel MIC XeonPhi (libXPhi), las llamadas serán a la biblioteca MKL. En el caso de GPUs (libGPU) las llamadas serán a cuBlas.

Se ha desarrollado una colección de kernels. Uno por cada función de BLAS y tipo de datos, que actúan cómo wrappers sobre las bibliotecas originales. Hemos denominado a esta biblioteca de kernels *HitCtrlBLAS* (ver Fig. 1). En esta biblioteca, cada función de BLAS es un kernel del modelo de controladores, con tres implementaciones, una para cada tipo de dispositivo (CPUs, XeonPhi, GPU). El interfaz propuesto para cada función de BLAS es común y uniforme. Dentro de cada implementación se gestionan los detalles necesarios para hacer la llamada adecuada a la biblioteca correspondiente. Por ejemplo, las implementaciones libGPU contienen internamente el *handle* necesario para las llamadas a cuBlas, que se crea y destruye con el controlador cuando este está asociado a una GPU. Otras diferencias respecto de las API de las bibliotecas utilizadas como base se resuelven en la implementación de cada kernel.

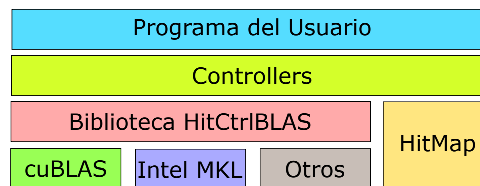

Fig. 1
Capas de abstracción y herramientas utilizadas en la solución propuesta.

Toda la gestión de transferencias de datos ya está resuelta en la biblioteca de controladores, que utiliza CUDA, o CAO según el tipo de dispositivo asociado al controlador. Las estructuras de tipo HitTile de la biblioteca Hitmap, utilizadas en los controladores, proveen también de un interfaz unificado para la gestión de las estructuras de datos.

V. Estudio experimental

A continuación se presenta un trabajo experimental para comprobar que la implementación de la propuesta de la biblioteca de kernels no impone una penalización demasiado elevada con respecto al tiempo que tardaría una aplicación implementada directamente usando una de las bibliotecas originales.

Como caso de prueba se ha seleccionado un código que permite calcular el resultado de multiplicar una matriz A por otra B (ambas cuadradas) repetidas veces por la derecha. Dicho de otro modo, permite obtener el resultado de $C = A \times B^n$. En este caso se utiliza una implementación recursiva: $C_k = C_{k-1} * B$, para $k \in \{1, 2, ..., n\}$, donde $C_0 = A$. Algunas implementaciones como Intel MKL no permiten especificar el mismo argumento como parámetro de entrada y salida en la misma llamada, por lo que el resultado de multiplicar $C_k \times B$ no puede almacenarse de vuelta en C para multiplicar de nuevo por B por la derecha en la siguiente iteración. Esto obliga al uso de una tercera matriz reservada en la unidad de cómputo donde almacenar el resultado parcial. Utilizamos un segundo kernel para hacer la copia de la matriz intermedia a la que se usa como primer operando en cada iteración. Este segundo kernel no tiene carga computacional aparte de la simple transferencia de memoria, lo que permite ver mejor si las propias lla-

```
/* CAL_BLAS_XPHI.cu */
// Caracterización del kernel: Parámetros por defecto
CAL_KERNEL_GPU_CHAR_STATIC(scopy, 2, def, def, def);

// Declaración de la implementación para GPU
CAL_KERNEL_LIBGPU(scopy, 5,
                  IVAL, int, n,
                  IN, HitTile_float*, x,
                  IVAL, int, incx,
                  IO, HitTile_float*, y,
                  IVAL, int, incy)
{
        cublasScopy(comm->handleCUBLAS, n, x.kdata, incx, y.kdata, incy);
}

/* CAL_BLAS_XPHI.cpp */
// Declaración de la implementación para XeonPhi
CAL_KERNEL_XPHI(scopy, 5,
                IVAL, int, n,
                IN, HitTile_float*, x,
                IVAL, int, incx,
                IO, HitTile_float*, y,
                IVAL, int, incy)
{
        cblas_scopy(n, x.data, incx, y.data, incy);
}
```

Fig. 2

Ejemplo de declaración de dos implementaciones (para GPU-CUDA y para Intel MIC XeonPhi), del kernel *scopy* de BLAS, en la biblioteca HitCtrlBLAS.

madas a los kernels suponen un overhead apreciable. La implementación permite jugar con dos parámetros, el número de veces que se multiplica la matriz B y el tamaño de la matriz. Antes de comenzar la iteración principal, las matrices iniciales A y B se copian en la memoria de la unidad de cómputo.

La implementación del código para las pruebas se realizó en cuatro etapas y dio como resultado tres códigos diferentes además de la propia biblioteca de kernels. En una primera fase, la especificación de la aplicación se tradujo a dos variantes, una utilizando Intel MKL y añadiendo los pragmas necesarios para ejecutar las funciones de BLAS en una XeonPhi mediante Offloading asistido por compilador y optimizándola para reducir las transferencias de datos, y una segunda variante en la que se utiliza cuBLAS y el compilador de NVIDIA para realizar el mismo procesamiento en una GPU habilitada para CUDA. En ambos casos se ha verificado que las funciones de BLAS se ejecutan realmente en sus respectivos coprocesadores y que en ningún caso el compilador o el sistema de ejecución toman la decisión de ejecutar la función en el host. Por ejemplo en el caso de la Xeon Phi utilizamos variables de entorno, tales como `OFFLOAD_REPORT=3`, antes de la ejecución del programa[19]. Estas dos variantes (matrixABk_XeonPhi, matrixABk_CUDA, no utilizan *Controllers* y se emplean como referencias.

Posteriormente, se realizó la versión que utiliza controladores, empezando por la creación de la capa intermedia de kernels. Como ya se ha explicado, por cada función de BLAS se implementaron dos kernels con el mismo nombre, uno para libXPhi que utiliza su equivalente en Intel MKL y una segunda que utiliza la función correspondiente en cuBLAS. Todo esto se realiza cuatro veces por cada grupo de funciones (eg. *GEMM → SGEMM, DGEMM, CGEMM, ZGEMM), y los kernels de CUDA y los de Intel MKL se crean en dos ficheros separados para poder compilar cada uno de ellos con su compilador específico. A continuación se crearon sendos ficheros de cabecera, y finalmente el programa principal.

El aspecto del programa principal se muestra en la Fig. 3. En él se pueden ver las diferentes etapas de las que suele constar normalmente el desarrollo de programas usando Controllers. En primer lugar se inicializa el sistema de controladores y se declaran las variables, tales como matrices y vectores, que se vayan a emplear, utilizando HitTiles. Para las matrices A y B se declara y reserva espacio de memoria en el host en la misma llamada. Para la matriz C sólo se declaran sus características pero no se reserva memoria para los datos, ya que se utilizará sólo como variable temporal en el dispositivo.

El siguiente paso es declarar los controladores que deseemos con el tipo *CALCntrl*. Mediante la función de creación el controlador quedará asociado con una unidad de cómputo (como por ejemplo una GPU, una Xeon Phi específica o un conjunto de uno o más cores CPU). Dado que todas las funciones de controladores reciben como parámetro un *CALCntrl*, es posible seleccionar dinámicamente (en tiempo de ejecución) qué funciones serán ejecutadas por cada unidad de cómputo (controlador). Esto también permite que las mismas funciones puedan ser ejecutadas por diferentes controladores, o bien que cada controlador ejecute un conjunto concreto de funciones.

A continuación, en el código, se define una configuración que incluye el número de hilos (lógicos o virtuales) que se utilizarán en la unidad de cómputo, y se copian las matrices desde la CPU a la misma usando *CAL_CntrlAttach*. Como se expuso anteriormente,

```
int main (int argc, char* argv[])
{
        CAL_CntrlInit(1);

        // Declaración de dominio de indices de cada vector/matriz
        HitTile_float matrixA, matrixB, matrixC;
        hit_tileDomain(&matrixC, float, 2, rows, columns);

        // Reserva de espacio (solo para vectores/matrices en CPU)
        hit_tileDomainAlloc(&matrixA, float, 2, rows, columns);
        hit_tileDomainAlloc(&matrixB, float, 2, rows, columns);

        // Inicializar datos en host
        ...

        // Crear controlador
        CALCntrl commGPU;
        CAL_CntrlCreate(&commGPU, CAL_CNTRL_LIBGPU, 0);

        // Número de threads (reales en GPU, virtuales en CPU)
        CALThread threads;
        CALThreadInit(threads, 2, rows, columns);

        // Asociar matrices que residirán tanto en host como en acelerador
        CAL_CntrlAttach(&commGPU, &matrixA);
        CAL_CntrlAttach(&commGPU, &matrixB);

        // Crear matrices que residirán sólo en el acelerador
        CAL_CntrlInternal(&commGPU, &matrixC);
        ...

        // Lanzamiento de kernels
        int times;
        for (times = 0; times < final_power; times++)
                {
                CAL_CntrlLaunch(commGPU, sgemm, threads, 13, &tile_op_notra,
                                &tile_op_notra, rows, columns, columns,
                                1.0, &matrixA, columns, &matrixB, columns,
                                0.0, &matrixC, rows);
                CAL_CntrlLaunch(commGPU, scopy, threads, 5, rows*columns,
                                &matrixC, 1, &matrixA, 1);
                }
        ...

        // Liberar estructuras
        CAL_CntrlDetach(&commGPU, &matrixA);
        CAL_CntrlDetach(&commGPU, &matrixB);
        CAL_CntrlDestroyInternal(&commGPU, &matrixC);

        CAL_CntrlDestroy(&commGPU);
        hit_tileFree(matrixA);
        hit_tileFree(matrixB);
}
```

Fig. 3

Extracto del programa principal del ejemplo $C = A \times B^k$ utilizando la biblioteca de controladores y HitCtrlBLAS. En este caso el controlador utilizado está asociado a la primera GPU del sistema.

puede ser necesario reservar espacio dentro del dispositivo para una estructura temporal, sin memoria reservada en en la CPU. En el ejemplo se utiliza la función *CAL_CntrlInternal* para realizar esta reserva para la matriz C. Seguidamente, el código contiene un bucle que lanza los kernels correspondientes a SGEMM, pasando los parámetros necesarios. En función del tipo de dispositivo al que esté asociado el *CALCntrl* que se pasa como parámetro, el framework de controladores elegirá entre la implementación de cuBLAS o la de Intel MKL para Xeon Phi. Durante las iteraciones no se produce movimiento de datos entre el coprocesador y el host, lo que aumenta la eficiencia. Finalmente, al realizar la operación de *detach* de las matrices A y B, los datos de la matriz resultado almacenada en A se copian de vuelta a la CPU. En el caso de B, el controlador detecta que no ha sido modificada y no realiza transferencia de memoria a la CPU. La matriz C reservada sólo en el dispositivo se destruye. Se destruye el controlador en uso y se libera la memoria usada en el host.

Las pruebas con este código fueron ejecutadas en una máquina denominada 'Chimera', que contiene las características especificadas en la Tabla I. El motivo para su selección, como se puede ver en dicha tabla, es que la misma dispone tanto de un coprocesador Xeon Phi como de una GPU Titan Black, lo que permite hacer pruebas utilizando cualquiera de los dos dispositivos, asegurando que el resto del hardware de la plataforma es exactamente el mismo. Cada experimento se repite al menos cinco veces, observándose una alta estabilidad en las medidas y re-

producibilidad de los resultados.

Las Fig. 4 y 5 muestran la relación entre los tiempos de ejecución del programa que obtiene la multiplicación de una matriz por otra repetidas veces ($C = A \times B^k$), para un $k = 3$, en función del tamaño de la matriz que se le pasa como parámetro.

La primera figura, Fig. 4, muestra la diferencia entre la versión que hace llamadas a cuBLAS directamente (representada mediante puntos cuadrados) y la versión que utiliza llamadas a las funciones del controlador que actúan como wrappers de cada función de cuBLAS. En esta gráfica se puede ver que el uso de *Controllers* (y por tanto de la biblioteca que se describe en este trabajo). La diferencia de rendimiento para la versión con *Controllers* comienza con una perdida de hasta un 79 % para matrices pequeñas de 1000×1000, que se reduce rápidamente al aumentar el tamaño de la matriz, hasta algo menos de un 4 % de penalización para matrices de 20000×20000.

La segunda gráfica, Fig. 5 contiene la misma comparación para Xeon Phi. Una versión utiliza llamadas directas a la implementación CBLAS de Intel MKL, y la otra utiliza el mismo código de controlador utilizado en el caso de la GPU, pero esta vez asociando el controlador a la unidad Xeon Phi en el momento de su creación. Se puede ver que en este caso los tiempos de ejecución de la variante con controlador, con matrices grandes a partir de 6500×6500, la desventaja ronda siempre entre el 10 % y el 19 %. Sin embargo, con tamaños pequeños de hasta 6000×6000, el comportamiento depende mucho del tamaño en concreto. En ciertos casos se obtienen penalizaciones de hasta el 19 %. En otros, sin embargo, la versión con controlador obtiene ventajas de rendimiento de hasta un 14 %, o incluso en un caso concreto con una ventaja de un 46 %. Puesto que estos resultados son consistentes a lo largo de varias ejecuciones, actualmente se está trabajando en realizar un profiling detallado para averiguar las razones de estas variaciones según el tamaño de las matrices,

Otra ventaja que se ha podido demostrar mediante la implementación de este software es que el uso de las nuevas bibliotecas de kernels permiten que la aplicación pueda cambiar entre las funciones de CBLAS MKL y las de cuBLAS tan sólo cambiando un único parámetro al crear el *Controller*, consiguiendo la portabilidad deseada.

VI. Conclusiones y trabajo futuro

En este trabajo se presenta una implementación de una biblioteca de funciones BLAS heterogénea, de tal forma que es posible definir algoritmos genéricos independientes de la plataforma, utilizando de forma transparente tanto implementaciones propias como las funciones BLAS que proveen las bibliotecas de los propios fabricantes de los diferentes aceleradores y coprocesadores. Esta solución permite portar fácilmente códigos que previamente realizaran llamadas a bibliotecas específicas BLAS de forma muy sencilla, a la vez que habilita el uso de múltiples coprocesadores de tipos diferentes sin obligar al programador a resolver las diferencias entre diferentes interfaces o duplicar el código.

Sistema operativo	CentOS 7.2.1511
Procesador	Intel Xeon E5-2620 v3
Velocidad de reloj	2.40GHz
Memoria principal	32 GB DDR3 1333 MHz
Memoria caché	15 MB
Coprocesador Intel	Xeon Phi (KNC) A3120
Coprocesador NVIDIA	GTX Titan Black GK110B
Entorno Intel	Parallel St. XE 2017.0.035
Entorno NVIDIA	CUDA (nvcc) 8.0.44

TABLA I
Características hardware y software del entorno de pruebas.

A mayores, gracias al sistema de controladores, esta solución evita el movimiento innecesario de los datos que residen en los diferentes tipos de unidades de cómputo hasta que los datos resultado son explícitamente sincronizados con el host por el programador. Al utilizar este sistema, el overhead introducido con respecto a una implementación directa del programa principal utilizando las bibliotecas proporcionadas por los fabricantes es pequeño.

Uno de los objetivos a corto plazo es la reducción del overhead que existe en las ejecuciones para la versión *libXPhi* con tamaños de matrices grandes, como se ha podido apreciar mediante el caso de estudio utilizado. Además, para poder tener una base experimental más amplia, se están implementando y probando nuevos ejemplos, incluyendo casos que permitan probar diversas optimizaciones o solapar cálculo y comunicación. Se plantea también como trabajo futuro comparar con otras librerías de BLAS orientadas a la portabilidad en sistemas heterogéneos.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Ministerio de Ciencia e Innovación (MICINN) y por el programa ERDF de la Unión Europea: Proyecto HomProg-HetSys (TIN2014-58876-P), CAPAP-H6 (TIN2016-81840-REDT), y COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

Referencias

[1] NVIDIA Corporation, "cuBLAS Library: User Guide," `http://docs.nvidia.com/pdf/CUBLAS_Library.pdf`, Jan. 2017.

[2] Intel Corporation, "Intel® Math Kernel Library (Intel® MKL)", `http://software.intel.com/en-us/intel-mkl`.

[3] Ana Moreton-Fernandez, Hector Ortega-Arranz, and Arturo Gonzalez-Escribano, "Controllers: An abstraction to ease the use of hardware accelerators," *The International Journal of High Performance Computing Applications*, p. 109434201770296, May 2017.

[4] Ana Moreton-Fernandez, Eduardo Rodriguez-Gutiez, Arturo Gonzalez-Escribano, and Diego R. Llanos, "Supporting the Xeon Phi coprocessor in a Heterogeneous Programming Model," in *Euro-Par 2017: Parallel Processing*, Santiago de Compostela, Galicia, Spain., Aug. 2017, Springer, Cham, In press.

[5] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R. Clinton Whaley, "A

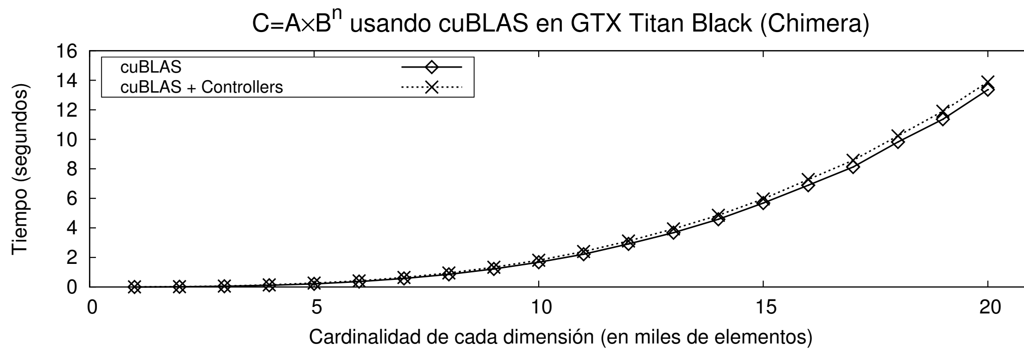

Fig. 4

Resultados experimentales para el programa de multiplicación $C = A \times B^n$ ejecutado en una GPU NVIDIA, tanto la versión que realiza llamadas directamente a cuBLAS, como la versión con *Controllers*.

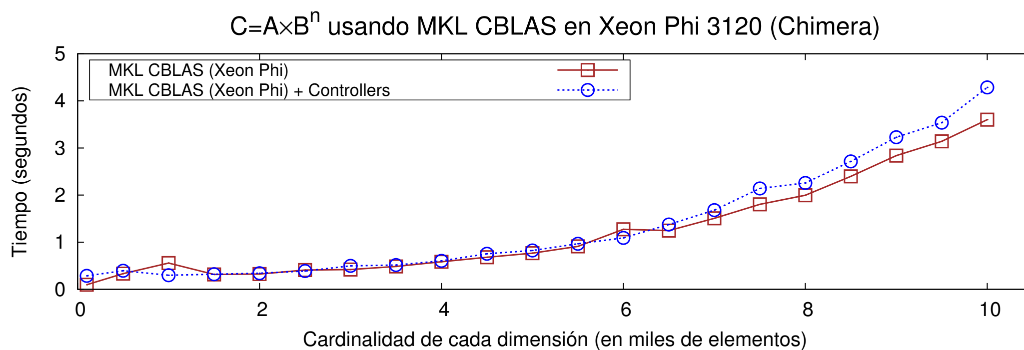

Fig. 5

Resultados experimentales para el programa de multiplicación $C = A \times B^n$, ejecutado en un coprocesador Xeon Phi, tanto la versión que realiza llamadas directamente a CBLAS de MKL. como la versión con *Controllers*.

proposal for a set of parallel basic linear algebra subprograms," in *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*. Aug. 1995, pp. 107–114, Springer, Berlin, Heidelberg.

[6] "PBLAS Home Page," `http://www.netlib.org/scalapack/pblas_qref.html`.

[7] Ravi Reddy Manumachu, Alexey Lastovetsky, and Pedro Alonso, "Heterogeneous PBLAS: Optimization of PBLAS for Heterogeneous Computational Clusters," in *2008 International Symposium on Parallel and Distributed Computing*, July 2008, pp. 73–80.

[8] Stanimire Tomov, Jack Dongarra, and Marc Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5, pp. 232–240, June 2010.

[9] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, Aug. 2012.

[10] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov, "HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi," *Scientific Programming*, vol. 2015, pp. e502593, Apr. 2015.

[11] Mark Gates, "MAGMA Forum: Performance issue," `http://icl.cs.utk.edu/magma/forum/viewtopic.php?f=2&t=1475`, Dec. 2016.

[12] Kent Knox, Jian Liu, David Tanner, Pavan Yalamanchili, Christian Kellner, Hugh Perkins, Tingxing Tim Dong, Gaëtan Lehmann, Cedric Nugteren, and Benjamin Coquelle, "clBLAS: a software library containing BLAS functions written in OpenCL," `https://github.com/clMathLibraries/clBLAS`, May 2017, original-date: 2013-08-13T15:05:53Z.

[13] Intel Corporation, "Using Intel® MKL Automatic Offload on Intel® Xeon Phi Coprocessors," `https://goo.gl/1kq8GB`, 2011.

[14] Intel Corporation, "Intel® MKL Automatic Offload enabled functions for Intel Xeon Phi coprocessors," `https://goo.gl/9jV7PY`, Feb. 2013.

[15] Kent Milfeld, "Intel Xeon Phi MIC Offload Programming Models," `https://goo.gl/fKxFgx`, Nov. 2014.

[16] Intel Corporation, "Intel® Math Kernel Library (Intel® MKL) 11.3 Release Notes," `https://goo.gl/oPU5ay`, Feb. 2016.

[17] Intel Corporation, "Intel® Math Kernel Library (Intel® MKL) 2017 Release Notes," `https://goo.gl/82fMkh`, Sept. 2016.

[18] NVIDIA Corporation, "NVBLAS," `http://docs.nvidia.com/cuda/nvblas/index.html`.

[19] James Jeffers, James Reinders, and Avinash Sodani, *Intel Xeon Phi Processor High Performance Programming*, Morgan Kaufmann, Cambridge, MA, edición: 2 edition, June 2016.

[20] L. Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R. Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, and Andrew Lumsdaine, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, June 2002.

[21] Gary W. Howell, James W. Demmel, Charles T. Fulton, Sven Hammarling, and Karen Marmol, "Cache efficient bidiagonalization using BLAS 2.5 operators," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, pp. 1–33, May 2008.

[22] Jeremy G. Siek, Ian Karlin, and E. R. Jessup, "Build to order linear algebra kernels," Apr. 2008, pp. 1–8, IEEE.

[23] Gary W. Howell and Charles T. Fulton, "Cache Efficient Householder Bidiagonalization," The College of William and Mary, Williamsburg, Virginia, June 2003.