García Blás, J., García Sánchez, J.D., Dolz, M., Carretero, J., Alemán, Y., Canales-Rodríguez, E., (2016). Migración portable y de altas prestaciones de aplicaciones Matlab a C++: deconvolución esférica de datos de resonancia magnética por difusión. En *Actas jornadas SARTECO 2016*, (105-110). Universidad de Salamanca.

Migración portable y de altas prestaciones de aplicaciones MATLAB a C++: deconvolución esférica de datos de resonancia magnética por difusión

Javier García Blas[1], J. Daniel García[1], Manuel F. Dolz[1], Jesús Carretero[1], Yasser Alemán[2] y Erick Jorge Canales-Rodríguez[3]

Resumen— **En muchos de los campos de la investigación científica, se ha establecido Matlab como herramienta *de facto* para el diseño de aplicaciones. Esta aproximación ofrece muchas ventajas como el rápido despliegue de prototipos, alto rendimiento en álgebra lineal, entre otros. Sin embargo, las aplicaciones desarrolladas son altamente dependientes del motor de ejecución de Matlab, limitando su despliegue en multitud de plataformas de altas prestaciones.**

En este trabajo presentamos un caso práctico de migración de una aplicación inicialmente basada en Matlab a una aplicación nativa en lenguaje C++. Para ello se presentará la metodología empleada para la migración y las herramientas que facilitan esta tarea. La evaluación llevada a cabo demuestra que la solución implementada ofrece un buen rendimiento sobre distintas plataformas y sistemas altamente heterogéneos.

Palabras clave— **Matlab, Imagen por Resonancia Magnética, Álgebra lineal.**

I. INTRODUCCIÓN

DESPUÉS de décadas de evolución en el campo de la Imagen por Resonancia Magnética (IRM), la implementación de una gran variedad de métodos avanzados y distintas modalidades de procesamiento de imagen ha servido para mostrar los complejos patrones de organización del cerebro, presentes tanto a escala micro como macroscópica. La organización estructural de la sustancia blanca del cerebro se puede caracterizar con gran detalle mediante los métodos de reconstrucción intravoxel basados en datos de resonancia magnética por difusión. Este tipo de técnica permite estudiar los patrones de conectividad estructural del cerebro de forma no invasiva e *in vivo*. El estudio de la conectividad ofrece múltiples ventajas en el diagnóstico de enfermedades mentales como la esquizofrenia y trastornos bipolares, entre otras.

Del conjunto de métodos y algoritmos propuestos hasta la fecha para el procesamiento de este subconjunto de imagen médica, podemos destacar la colección de rutinas de MATLAB HARDI-Tools[1], dedicada al análisis de datos de difusión de alta resolución angular (del inglés, High Angular Resolution Diffusion Imaging, HARDI). HARDI es un conjunto de código implementado y utilizado por investigadores de la Fundación de Investigación y Docencia FIDMAG (Barcelona, España), el Centro de Neurociencias de Cuba (La Habana, Cuba) y el Laboratorio de Imagen Médica del Hospital Gregorio Marañón (Madrid, España).

Por otro lado, desde décadas, el uso del lenguaje MATLAB se ha convertido en el estándar *de facto* para el diseño y el prototipado en una amplia variedad de aplicaciones de ciencia e ingeniería. Debido a su sencillez de programación, la comunidad científico-técnica ha optado por utilizar este lenguaje de alto nivel para el desarrollo de prototipos de aplicaciones que requieren el cómputo de problemas de álgebra lineal. Por contrapartida, muchos de estos prototipos han pasado a una etapa de producción sin haber sido convenientemente adaptados y optimizados para recibir grandes cargas de trabajo. Por ello, hoy en día se pueden encontrar múltiples ejemplos de aplicaciones MATLAB que se ejecutan en plataformas de producción de altas prestaciones pero que, debido a la naturaleza del propio lenguaje, no aprovechan correctamente los recursos proporcionados por éstas.

El objetivo de este trabajo es presentar las tecnologías y herramientas actuales para la migración de aplicaciones basadas en MATLAB. En particular, en este trabajo estudiaremos el método de deconvolución esférica RUMBA-SD (del inglés, Robust and Unbiased Model-BAsed Spherical Deconvolution, RUMBA-SD [1]), debido a que esta técnica ha demostrado estar entre los métodos más avanzados hasta la fecha para inferir los cruces de fibras de la sustancia blanca, obteniendo el primer lugar en un concurso internacional de reconstrucción de imágenes[2].

Con el fin de mantener la compatibilidad y la solidez de esta aplicación, hemos implementado una versión equivalente C++ usando las bibliotecas Armadillo [2] y ArrayFire. Estas bibliotecas permiten representar las operaciones de álgebra lineal comunes como operadores aritméticos básicos de C++. Otra característica interesante es que este tipo de soluciones proporcionan compatibilidad con bibliotecas de álgebra lineal existentes, como OpenBlas, Intel MKL

[1]Dpto. de Informática, Universidad Carlos III de Madrid, 28911–Legan'es, e-mail: fjblas@inf.uc3m.es.

[2]BiiG, Hospital General Universitario Gregorio Marañón, 28007–Madrid

[3]Fundación para la Investigación y Docencia FIDMAG, CIBERSAM, 08035–Barcelona.

[1]http://neuroimagen.es/webs/hardi_tools/

[2]Para más información ver: http://hardi.epfl.ch/static/events/2013_ISBI/

o ATLAS. Por lo tanto, nuestra solución puede veneficiarse de múltiples arquitecturas multi-núcleo, simplemente mediante el despliegue de múltiples hilos paralelos.

Las contribuciones de este trabajo son las siguientes. Primero, presentamos la implementación de un de los métodos de reconstrucción incluidos en HARDI llamado *RUMBA-SD*. Segundo, indicamos los pasos necesarios y herramientas empleadas para este propósito. Finalmente, a través de una evaluación exaustiva, demostramos qué bibliotecas son más adecuadas para la migración de aplicaciones basadas en el motor de ejecución MATLAB.

El resto del trabajo está estructurado de la siguiente forma. En la Sección II se introducen las herramientas usadas en este trabajo. La Sección III contextualizamos el problema del procesamiento de en RUMBA-SD. A continuación, en la Sección IV detallamos la metodología aplicada para la paralelización de una de las herramientas incluidas en HARDI. En la Sección V presentamos resultados de la evaluación realizada. Trabajos relacionados con nuestra solución son presentados en la Sección VI. Finalmente, en la Sección VII discutimos sobre los resultados arrojados en este trabajo.

II. ÁLGEBRA LINEAL PARA ENTORNOS BASADOS EN C++

En esta sección resumiremos las principales herramientas que existen en la literatura para el procesamiento de álgebra lineal para el lenguaje de programación C++. Destacamos entre las opciones dos aproximaciones: Armadillo y ArrayFire.

Armadillo [2] es una biblioteca de álgebra lineal para el lenguaje C++, diseñada con el objetivo de alcanzar un buen equilibrio entre la velocidad y facilidad de uso. Dentro de las principales ventajas, destacamos la sintaxis de alto nivel (API) similar a MATLAB. Armarillo es útil para el desarrollo de algoritmos directamente en C++, o la conversión rápida de código para entornos de producción.

En el siguiente fragmento de código se muestra un ejemplo de como se implementa una multiplicación de matrices con Armadillo:

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(int argc, char** argv)
{
  mat A = randu<mat>(5000,5000);
  mat B = randu<mat>(5000,5000);
  mat C = A * B;
  return 0;
}
```

ArrayFire [3] es una biblioteca de funciones de código abierto con interfaces para C, C++, Java, R y Fortran. Se integra con cualquier aplicación CUDA, y contiene una API que facilita la programación. La bibloteca ArrayFire está diseñada para su uso en un amplio rango de sistemas, desde los sistemas compuestos por una sola GPU a grandes supercomputadores multi-GPU. En el siguiente ejemplo se muestra un caso simular alimplementado anteriormente en Armadillo:

```
#include <iostream>
#include <arrayfire.h>

using namespace std;
using namespace af;

int main(int argc, char** argv)
{
  array A, B;
  af_randn(&A,5000,5000,fp32);
  af_randn(&A,5000,5000,fp32);
  array C = matmul(A, B);
  return 0;
}
```

En este caso, tanto las matrices A y B son entidades abstractas cuya localización vendrá dada por el *back-end* seleccionado para su ejecución. En caso de escoger aceleradores GPU, ambas matrices estarán almacenadas integramente en la memoria del dispositivo GPU.

III. INTRAVOXEL FIBER RECONSTRUCTION (RUMBA-SD)

En esta sección introduciremos los conceptos básicos de RUMBA-SD para su contextualización y para mostrar el tamaño del problema.

En tejidos biológicos, la difusión del agua no es igual en todas las direcciones debido a la presencia de obstáculos o barreras que limitan el movimiento molecular. Para caracterizar cuantitatívamente este fenómeno se usa el tensor de difusión (DTI, del inglés, Diffusion Tensor Imaging). Este tensor se define como una simétrica 3x3 y la matriz positiva definida (mostrada en la Ecuación 1).

$$D = \begin{pmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{yx} & D_{yy} & D_{yz} \\ D_{zx} & D_{zy} & D_{zz} \end{pmatrix} \quad (1)$$

El tensor de difusión puede ser visualizado como un elipsoide, donde las orientaciones de sus tres ejes principales se definen por sus vectores propios o autovectores y la longitud de los ejes por la magnitud de sus valores propios o difusividades (Figura 1).

El tensor de difusión resulta ser un modelo adecuado en el caso de que cada voxel del cerebro (pixel tridimensional) contenga un solo grupo de fibras paralelas (ver el panel (a) de la Figura 1). En este caso el eje principal del elipsoide asociado al tensor de difusión coincide con la orientación del grupo de fibras. De igual manera, el tensor de difusión es efectivo para caracterizar el proceso de difusión en regiones del cerebro donde la difusión ocurre sin interactuar con los tejidos y por tanto tiene una geometría esférica (ver el panel (c) de la Figura 1). Sin embargo, en regiones del cerebro donde varias fibras se cruzan e interceptan, el modelo del tensor de difusión no permite inferir las orientaciones de los haces de fibras (ver el panel b de la Figura 1). De hecho, diferentes cruces de fibras podrían dar lugar a tensores de difusión similares (ver Figura 2).

Tales limitaciones en el enfoque DTI han impulsado el reciente desarrollo de numerosos protocolos de

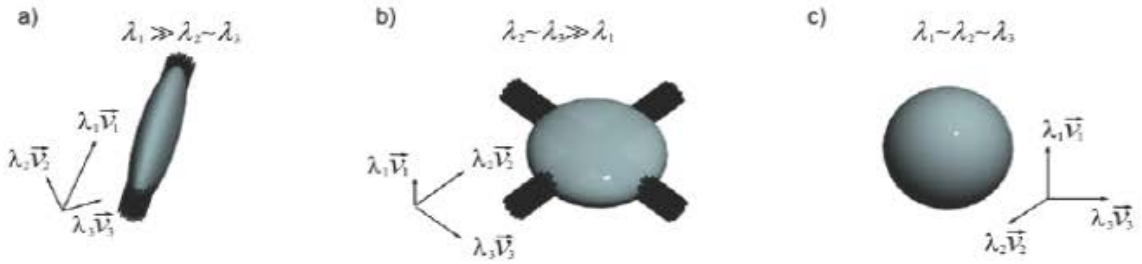

Fig. 1. La organización microestructural puede ser revelada por el elipsoide de tensor de difusión.

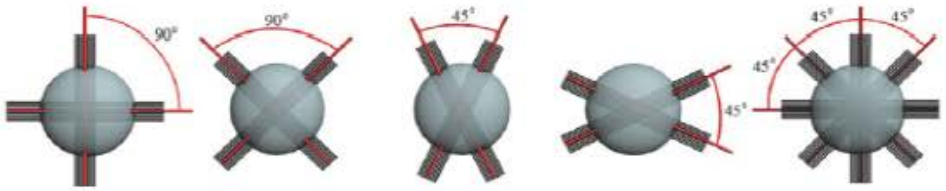

Fig. 2. Diferentes cruces de fibra pueden producir el mismo elipsoide de difusión.

muestreo, modelos de difusión y técnicas de reconstrucción. Entre estos métodos, la técnica RUMBA-SD se basa en asumir que cada voxel está compuesto por un número elevado de tensores de difusión y que cada tensor corresponde a un compartimento que contiene un solo grupo de fibras paralelas con una orientación predefinida. El objetivo del método es estimar la fracción de volumen de cada uno de estos compartimentos para obtener, en cada voxel, la función de distribución de orientación de las fibras (ODF, del inglés, Orientational Distribution Function). La Figura 3 muestra un ejemplo de ODF que corresponde al cruce de tres grupos de fibras perpendiculares. Las orientaciones donde la ODF alcanza sus valores máximos se corresponden con las orientaciones de las fibras nerviosas.

Como se muestra en la Figura 3, para facilitar la visualización, se asigna el código de color de la superficie de acuerdo con la dirección de difusión $([x, y, z] - [r, b, g]$, donde r = rojo representa la probabilidad a lo largo de la orientación de izquierda a derecha, b = azul representa superior-inferior orientación, y g = verde representa la orientación antero posterior).

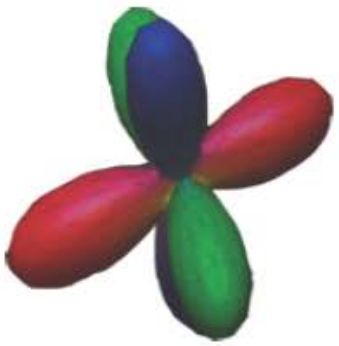

Fig. 3. Función de distribución de orientación.

IV. Parallel HARDI

Dentro del conjunto de utilidades de HARDI, en este trabajo nos hemos centrado en la paralelización de RUMBA-SD [3]. Sin embargo, la incorporación de nuevos métodos resulta sencillo debido al diseño modular de la solución.

Como se observa en la Figura 4, pHARDI posee un diseño basado en capas, el cual permite el uso de distintos aceleradores de álgebra lineal para una amplia gama de dispositivos. Estos dispositivos pueden ser multi-core, dispositivos GPU (tanto para CUDA como OpenCL) o incluso aceleradores basados en tecnología x86 como Intel Xeon Phi. En el caso de no contar con este tipo de dispositivos, nuestra solución soporta el acceso de múltiples bibliotecas optimizadas de cómputo de álgebra lineal. Para ello se han desarrollado dos versiones distintas: una puramente basada en Armadillo y otra con soporte a ArrayFire. Para esta aproximación, únicamente se ha utilizado ArrayFire para aquellos fragmentos de código computacionalmente más caros, también llamados núcleos de ejecución o *kernels*. Es importante destacar que ambas soluciones usan la misma disposición lógica de las matrices o volúmenes, tanto en CPU como en GPUs (*row-major*). Por lo tanto, no son necesarias transformaciones adicionales en el código fuente original.

Fig. 4. Arquitectura software de pHARDI.

pHARDI admite dos modelos de acceso a las imágenes para el procesamiento de datos de entra-

[3] https://bitbucket.org/fjblas/phardi

da. El primer patrón procesa por separado cada rodaja a partir de los volúmenes de datos de entrada ($x \times y \times z$), lo que resulta en un total de z rodajas, cada una compuesta por $x \times y \times t$ voxels, siendo t el número de orientaciones. El segundo diseño procesa todos los píxeles de los volúmenes en una sola matriz. Esta matriz tiene una dimensión de $n \times m$ elementos, donde n corresponde con la cantidad de orientaciones evaluadas (por ejemplo 100) y m con la cantidad de voxels en cada volumen (por ejemplo 125 megapíxeles).

Respecto a la gestión de datos, se ha optado por el uso de la biblioteca ITK [4]. Esta biblioteca facilita la gestión de lectura y escritura de datos en formatos de datos comunes en el ámbito de la imagen médica, como pueden ser DICOM y NIFTI. Otra de las ventajas del uso de ITK es el soporte de compresión automática de ficheros, reduciendo significativamente el espacio necesario en el almacenamiento, tanto para la entrada como para la salida de datos.

Listing 1
NÚCLEO DE EJECUCIÓN DE RUMBA-SD IMPLEMENTADO EN ARMADILLO.

```
for (size_t i = 0; i < Niter; ++i) {
    Ratio = mBessel_ratio<T>(n_order,
        Reblurred_S);

    #pragma omp parallel for simd
    for (size_t k = 0; k < SR.n_cols; ++k )
        for (size_t j = 0; j < SR.n_rows; ++j)
            SR(j,k) = Signal(j,k) * Ratio(j,k);
    KTSR = KernelT * SR;
    KTRB = KernelT * Reblurred;

    #pragma omp parallel for simd
    for (size_t k = 0; k < fODF.n_cols; ++k)
        for (size_t j = 0; j < fODF.n_rows; ++j
            )
            fODF(j,k) = fODF(j,k) * KTSR(j,k) /
                (KTRB(j,k) + std::
                numeric_limits<double>::epsilon
                ());

    Reblurred = Kernel * fODF;

    #pragma omp parallel for simd
    for (size_t k = 0; k < Signal.n_cols; ++k)
        for (size_t j = 0; j < Signal.n_rows;
            ++j)
            SUM(j,k) = (pow(Signal(j,k),2) +
                pow(Reblurred(j,k),2))/2 - (
                sigma2(j,k) * (Signal(j,k) *
                Reblurred(j,k)) / sigma2(j,k))
                * Ratio(j,k);

    sigma2_i = (1.0/N) * sum( SUM , 0) /
        n_order;

    #pragma omp parallel for
    for (size_t k = 0; k < sigma2_i.n_elem; ++k
        )
        sigma2_i(k) = std::min<T>(std::pow<T
            >(1.0/10.0,2),std::max<T>(sigma2_i(
            k), std::pow<T>(1.0/50.0,2)));

    sigma2 = repmat(sigma2_i, N, 1);
}
```

El Listado 1 muestra el núcleo de ejecución de RUMBA-SD. Este código se ejecuta un determinado número de iteraciones ($Niter$). Cada iteración se compone de tres multiplicaciones de matrices y varias multiplicaciones elemento a elemento. Hasta donde sabemos, Armadillo carece de una implementación paralela de las llamadas de multiplicación elemento a elemento, por lo que estas llamadas se han paralelizado mediante OpenMP, preservando al mismo tiempo las primitivas de acceso a datos de Armadillo. Los bucles también están paralelizados aplicando la directiva OpenMP *pragmasimd*, con el objetivo de vectorizar de los contenidos de cada bucle.

V. EVALUACIÓN EXPERIMENTAL

La evaluación del rendimiento se ha llevado a cabo en un equipo con Ubuntu 14.04 x64. El hardware del sistema consiste en un Intel(R) Xeon(R) CPU E5-2630 v3 a 2.40GHz y 128GiB de RAM. Los compiladores utilizados son GCC 4.8 e Intel 15.1. La tarjeta gráfica empleada consiste en una NVidia Titan Black (perfil alto) y una GTX 680 (perfil bajo). La versión de CUDA usada en el proceso de compilador es 7.5. En todos los casos se ha compilado el código fuente con las opciones `-O3` y `-DNDEBUG`. Todos los resultados de la evaluación fueron adquiridos como el promedio de cinco ejecuciones consecutivas.

En este trabajo se han evaluado las siguientes configuraciones de pHARDI: BLAS, OpenBLAS, NVBLAS, Intel MKL y ArrayFire. Para ello se ha experimentado con tres casos de prueba distintos, en lo que varia el tamaño del conjunto de datos.

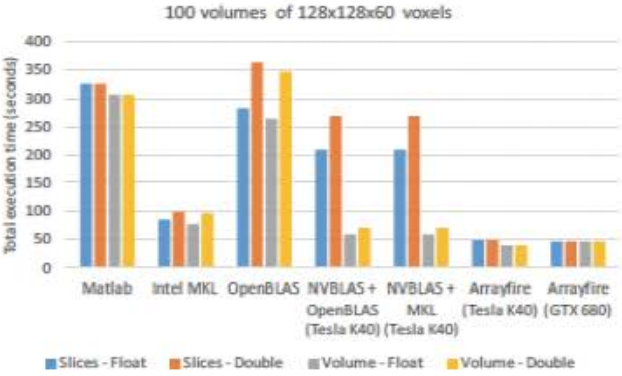

Fig. 5.Comparación de tiempo de ejecución de pHARDI sobre distintas soluciones de aceleración de álgebra lineal.

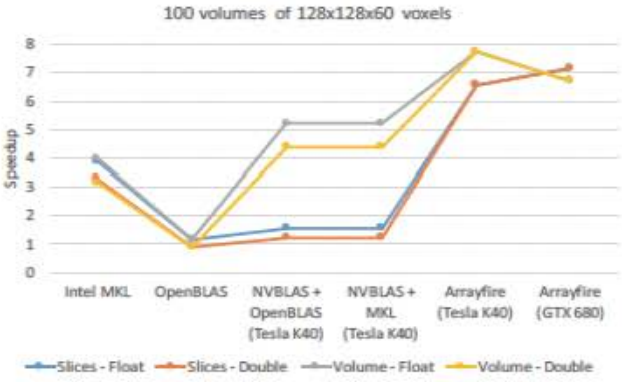

Fig. 6.Aceleración alcazada sobre distintas soluciones de aceleración de álgebra lineal. Se toman los tiempos de ejecución de MATLAB como valores de referencia.

El caso de entrada evaluado en este trabajo consis-

te en un estudio real compuesto por 101 volúmenes de $128x128x60$ píxeles cada uno. En la Figura 5 y 6 se muestran los tiempos de ejecución (en segundos) y la aceleración alcanzada de dos tipos de ejecución, respectivamente. Se ha procedido a ejecutar pHARDI sobre dos disposiciones de datos diferentes. En la primera, se procesa cada una de las rodajas de los volúmenes, resultado en 60 iteraciones ("slices"). En la segunda, todos los píxeles son procesados en una única matrix. Para ambos casos el proceso de reconstrucción se obtiene trás la ejecución de 300 iteraciones ("volume"). En la figura se incluye el tiempo total de ejecución, incluyendo los tiempos de entrada y salida.

A simple vista podemos observar que la solución que ofrece mayor rendimiento son las versiones implementadas con Arrayfire. Obviamente la ventaja viene dada por el aprovechamiento de la capacidad de cómputo en GPU. Sin embargo, apreciamos que no hay una gran diferencia entre los dos modelos de GPU analizados. En el futuro se pretende realizar un análisis mas detallado sobre este aspecto. Adicionalmente observamos que la implementación compilada sobre la biblioteca Intel MKL es la que más beneficia el tiempo de ejecución de la aplicación. Hay que señalar que Matlab utiliza esta misma tecnología para la paralelización de álgebra lineal. Sin embargo, nuestra implementación hace un uso más eficiente en la carga y almacenamiento de datos.

Es importante destacar que NVBLAS únicamente es capaz de descargar operaciones relacionadas con BLAS3, por lo que limita sustancialmente el rango de memojora. Además, en cada una de las iteraciones, es necesario realizar copias de memoria entre el anfitrión y el dispositivo. Sin embargo, este proceso no es necesario con Arrayfire, ya que gracias al API de desarrollo es posible especificar que variables se mantienen en la memoria del dispositivo. Esta característica es especialmente benificiosa en el caso de aplicaciones con un proceso iterativo.

VI. Trabajos relacionados

Para hacer frente a estos problemas de portabilidad de MATLAB, existen diferentes alternativas: *ii*) usar de transcompiladores (*source-to-source compilers*) para migrar códigos MATLAB a lenguajes compilados y orientados a computación de altas prestaciones, e.g., C/C++ o Fortran; *ii*) optimizar el código MATLAB mediante el uso de bibliotecas y modelos de programación paralela y *iii*) traducir, de forma manual, la aplicación MATLAB a un lenguaje HPC para finalmente optimizarla mediante paradigmas de programación paralelos.

Como bien es sabido, MATLAB es un lenguaje sencillo pero requiere de un intérprete para ejecutarse, convirtiéndolo en algunos casos en una solución más lenta frente a lenguajes compilados. Por ello, el compilador comercial MCC [5] permite traducir de forma automática un programa MATLAB a C/C++ o Fortran e incluso compilarlo un ejecutable único MEX. No obstante, también pueden encontrarse soluciones libres: el compilador Falcon [6] traduce aplicaciones MATLAB a códigos Fotran 90; otros compiladores como Conlab [7] y Otter [8] están orientados a generar código para plataformas de memoria distribuida y compartida. Sin embargo, transformar una aplicación MATLAB no sólo significa traducir instrucción por instrucción a un lenguaje orientado a HPC, sino también aplicar optimizaciones. Por ejemplo, el compilador Falcon reemplaza el código cuando encuentra coincidencias con patrones sintácticos por rutinas ya optimizadas. Otros, como el prepreprocesador Kapf [9] centra sus esfuerzos en detectar automáticamente porciones de código que encajan con operaciones BLAS y sustituirlas por su correspondiente llamada.

Cuando la traducción realizada por un transcompilador no genera las mejoras del rendimiento esperadas, se pueden aplicar mejoras directamente en el código MATLAB. Por ejemplo, se pueden utilizar versiones de bibliotecas propietarias de BLAS optimizadas para arquitecturas de propósito general (e.g. ACML [10] (AMD), MKL [11] (Intel) y ESSL [12] (IBM)) o específico (cuBLAS [13] para GPUs NVIDIA). También existen implementaciones de terceros de BLAS, como por ejemplo GotoBLAS [14], ATLAS [15] o GSL [16]. Otros *frameworks* permiten incluso el uso de modelos de programación paralela OpenMP o MPI y aceleradores [17], [18].

En caso de que sea posible traducir el código MATLAB de forma manual, se pueden utilizar algunas bibliotecas existentes en la literatura que, debido a su interfaz de alto nivel, permiten aliviar la tarea de traducción. Bibliotecas como Eigen [19] o ViennaCL [20] en C++ proporcionan una interfaz a BLAS sencilla y utilizan, al mismo tiempo, modelos de programación OperMP, OpenCL y/o CUDA. Otras bibliotecas en C++, como Armadillo [2], tienen el objetivo de alcanzar un buen equilibrio entre la velocidad y facilidad de uso. Dentro de las principales ventajas, destaca su sintaxis de alto nivel, muy similar a la de MATLAB. Armardillo es útil para el desarrollo de algoritmos directamente en C++, o la conversión rápida de código para entornos de producción. Otra alternativa es la biblioteca ArrayFire [3], que soporta interfaces para C, C++, Java, R y Fortran y es integrable con códigos basados en CUDA.

VII. Conclusiones

En este trabajo se ha presentado un caso práctico de migración de una aplicación desarrollada inicialmente en el lenguaje MATLAB, a una nueva versión basada totalmente en el lenguaje C++. La principal ventaja de la solución propuesta es la portabilidad ofrecida, siendo compatible con un gran número de plataformas. Además, esta aproximación proporciona una solución flexible, permitiendo la integración de distintas bibliotecas o aceleradores de álgebra lineal.

Uno de los problemas que hemos observado en ArrayFire es la limitada cobertura para el desarrollo de aplicaciones basadas en múltiples dispositivos

GPU. Por ello, planeamos desarrollar bibliotecas que faciliten esta aproximación. Otra línea de trabajo que arroja este trabajo es la detección y aplicación de patrones de paralelismo al código implementado, con el objetivo de aprovechar todos los dispositivos disponibles en el computador, y no solo limitar su aplicación a GPUs.

Agradecimientos

Este trabajo ha sido financiado por el Proyecto Europeo ICT 644235 "RePhrase: REfactoring Parallel Heterogeneous Resource-Aware Applications" y el Ministerio de Economía y Competitividad, bajo el proyecto TIN2013-41350-P "Scalable Data Management Techniques for High-End Computing Systems".

Referencias

[1] Erick J Canales-Rodríguez, Alessandro Daducci, Stamatios N Sotiropoulos, Emmanuel Caruyer, Santiago Aja-Fernández, Joaquim Radua, Jesús M Yurramendi Mendizabal, Yasser Iturria-Medina, Lester Melie-García, Yasser Alemán-Gómez, et al., "Spherical deconvolution of multichannel diffusion MRI data with non-Gaussian noise models and spatial regularization," *PloS one*, vol. 10, no. 10, pp. e0138910, 2015.

[2] Conrad Sanderson, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," 2010.

[3] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos, "ArrayFire: a GPU acceleration platform," in *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2012, pp. 84030A–84030A.

[4] Hans J Johnson, Matthew M McCormick, and Luis Ibanez, *The ITK Software Guide Book 1: Introduction and Development Guidelines-Volume 1*, Kitware, Inc., 2015.

[5] The Mathworks, Inc., "C/C++ Compiler Suite," `http://www.mathworks.com`, Online; accedido el 18 de Mayo de 2016.

[6] Luiz De Rose and David Padua, "Techniques for the translation of matlab programs into fortran 90," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 2, pp. 286–323, Mar. 1999.

[7] Peter Drakenberg, Peter Jacobson, and Bc Kågström, "A conlab compiler for a distributed memory multicomputer.," in *PPSC*, 1993, pp. 814–821.

[8] "Results from a parallel matlab compiler," in *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, Washington, DC, USA, 1998, IPPS '98, pp. 81–, IEEE Computer Society.

[9] Kuck and Associates, Inc., "KAP for IBM Fortran and C," `http://www.kai.com/product/ibminf.html`, Online; accedido el 18 de Mayo de 2016.

[10] AMD Core Math Library (ACML), "`http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/`," .

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

[12] IBM, "Engineering Scientific Subroutine Library," .

[13] nVidia, "cuBLAS Library User Guide ," `https://developer.nvidia.com/cubla`, 2012, Online; accedido el 18 de Mayo de 2016.

[14] John Markoff, "Writing the fastest code, by hand, for fun: A human computer keeps speeding up chips," *The New York Times*, 2005.

[15] R. Clint Whaley and Jack J. Dongarra, "Automatically tuned linear algebra software," in *International Conference on Supercomputing (ICS)*, 1998.

[16] Brian Gough, *GNU Scientific Library Reference Manual - Third Edition*, Network Theory Ltd., 3rd edition, 2009.

[17] The Mathworks, Inc., "Multicore-Capable Code Generation Using OpenMP," `http://es.mathworks.com/products/matlab-coder/features.html#multicore-capable-code-generation-using-openmp`, Online; accedido el 18 de Mayo de 2016.

[18] The Mathworks, Inc., "Parallel Computing Toolbox," `http://es.mathworks.com/products/parallel-computing/`, Online; accedido el 18 de Mayo de 2016.

[19] Gaël Guennebaud, Benoît Jacob, et al., "Eigen v3," http://eigen.tuxfamily.org, 2010.

[20] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.