Serrano, E., García Blás, J., Carretero, J., Abella, M. (2016). Propuesta arquitectónica para la ejecución de tareas en Apache Spark para entornos heterogéneos. En *Actas jornadas SARTECO 2016* (299-304). Universidad de Salamanca

Propuesta arquitectónica para la ejecución de tareas en Apache Spark para entornos heterogéneos

Estefanía Serrano [1], Javier García Blas[1], Jesús Carretero[1] y Mónica Abella[2 3]

Resumen— **Las desventajas presentes en las plataformas de computación actuales y la fácil migración a la computación en la nube, han logrado que cada vez más aplicaciones científicas se adapten a los distintos *frameworks* de computación distribuida basadas en flujo de tareas. Sin embargo, muchas de ellas ya han sido optimizadas para su ejecución en aceleradores tales como GPUs. En este trabajo se presenta una arquitectura que facilita la ejecución de aplicaciones tradicionalmente basadas en entornos HPC al nuevo paradigma de computación Big Data. Además, se demuestra como gracias a una mayor capacidad de memoria, el reparto automático de tareas y a la mayor potencia de cálculo de los sistemas heterogéneos se puede converger a un nuevo modelo de ejecución altamente distribuido. En este trabajo se presenta un estudio de la viabilidad de esta propuesta mediante la utilización de GPUs dentro de la infraestructura de cómputo Spark. Esta arquitectura será evaluada a través de una aplicación de tratamiento de imagen médica. Los resultados demuestran que aunque nuestra arquitectura sobre un nodo no produce resultados absolutos mejores que la aplicación original, según se aumenta el número de GPUs y por lo tanto la ocupación de estas influye más, la aplicación basada en Spark se acerca al rendimiento del simulador original. Finalmente, realizamos un estudio de la ocupación de las GPUs empleadas para las distintas políticas propuestas, demostrando que al tener en cuenta las características dinámicas de la GPUs (número de tareas en ejecución) podemos tener una mayor ganancia de rendimiento.**

Palabras clave— **GPU, Spark, pyCUDA, Imagen médica.**

I. Introducción

La utilización cada vez más frecuente de *frameworks* de computación distribuida creados originalmente para problemas Big Data en aplicaciones científicas, está llevando a la adaptación de estas aplicaciones al paradigma MapReduce [1], el cual es, actualmente, el paradigma más usado por este tipo de *frameworks* en las plataformas cloud. La dificultad que a veces conlleva esta adaptación y, en ocasiones, la pérdida de eficiencia debido a las dependencias entre los datos, hace que muchas aplicaciones no sean migradas a este nuevo paradigma.

En el caso de aplicaciones científicas, como en el área de trabajo de la imagen médica, tradicionalmente se ha optado por la utilización de aceleradores de cómputo como las GPUs (de las siglas en inglés, Graphics Processing Unit). Estos aceleradores, debido a su arquitectura SIMD (de las siglas en inglés, Single Instruction Multiple Data) permiten la ejecución rápida de la mayoría de los algoritmos de reconstrucción y proyección, debido a que la generación de cada uno de los píxeles finales es totalmente independiente. El principal problema relacionado con el uso de este tipo de hardware es la falta de memoria dentro del dispositivo, debido a que la mayoría de tarjetas gráficas de bajo y mediano coste poseen alrededor de 3GB de memoria, típicamente GDDR5. Aunque es cierto que en estos momentos muchas tarjetas dedicadas al cómputo, como por ejemplo la gama Tesla de NVidia, pueden llegar a tener más de 10 GB de memoria, esto puede ser insuficiente para la futura generación de imágenes en alta resolución, la cual podría llegar a requerir 32 GB de almacenamiento en memoria principal. Una posible solución a este problema de limitación de memoria es el particionamiento del problema para su resolución independiente en diferentes GPUs. Sin embargo, la tarea de particionamiento no es sencilla, ya que en muchas ocasiones se requiere de modificaciones significativas en el código original.

La ventajas de la computación distribuida en términos de capacidad de memoria son evidentes. La memoria en estos entornos se multiplica, así como la capacidad computacional. Adicionalmente, en clústeres de computadores altamente heterogéneos es posible disponer de nodos con múltiples GPUs. Aunque existen entornos de computación distribuida como MPI, en los que ya se ha estudiado la viabilidad de ejecución de estos algoritmos [2], estos están principalmente orientados a aplicaciones científicas. Sin embargo, cuando hablamos de problemas científicos altamente paralelizables, la aplicación del paradigma MapReduce puede ser de gran ayuda. Este paradigma facilita el particionamiento automático de los datos, la ejecución basada en tareas y el aumento de la localidad de datos.

La principal contribución de este trabajo es el estudio de la viabilidad de la migración de aplicaciones inicialmente basadas en plataformas HPC a plataformas Big Data. Como caso de uso, se partirá de una aplicación de imagen médica, llamada Fux-Sim [3]. Las contribuciones de este trabajo se aplican no sólo a la posibilidad de ejecutar aplicaciones dentro del ámbito de la imagen médica en entornos Big Data, sino también a la generalización del proceso llevado a cabo aquí a cualquier campo de aplicación que haga un uso intensivo de GPUs. En resumen, las contri-

[1]Dpto. de Informática, Univ. Carlos III de Madrid, e-mail: esserran@inf.uc3m.es.
[2]Dpto. de Ingeniería Biomédica y Aerospacial, Univ. Carlos III de Madrid
[3]Instituto de Investigación Sanitaria Gregorio Maranón (IiSGM), Madrid

buciones de este trabajo son las siguientes: primero, se presenta una solución para proporcionar acceso a GPUs dentro de las tareas gereradas por Apache Spark; segundo, se detalla la metodología seguida para la migración de una aplicación inicialmente implementada en una plataforma HPC a un sistema Big Data; por último, se realiza una evaluación comparativa de la solución propuesta.

Este documento se organiza de la siguiente manera. En la Sección II se explicarán los fundamentos básicos de Apache Spark, se fundamentará la elección de las tecnologías utilizadas y se definirá el procedimiento llevado a cabo con nuestra aplicación de forma generalizada. La Sección IV detalla la funcionalidad del caso de estudio así como los algoritmos contenidos en sus kernels. A continuación, se llevará a cabo la evaluación de la propuesta en la Sección V. Finalmente, se proporcionará una relación de los trabajos relacionados y las conclusiones de este trabajo.

II. Conceptos previos

Esta sección introduce los conceptos y herramientas en las que se sustenta la propuesta. Para ello se describirá brevemente la arquitectura del *framework* Apache Spark y la interfaz de acceso a GPU, PyCuda.

A. Apache Spark

Apache Spark [4] es un *framework* de computación distribuida de próposito general. Utilizado en diversos ámbitos, su mayor éxito lo ha tenido en aplicaciones de análisis de datos en entornos Big Data. Posee APIs para programar en distintos lenguages como Scala, Java, Python y además cuenta con librerias enfocadas al aprendizaje automático.

Basado en el paradigma MapReduce y con una ejecución enfocada a tareas, Spark es compatible con varios gestores de recursos (por ejemplo Yarn) y tiene conectores para distintos sistemas de ficheros y bases de datos distribuidas. A diferencia de Hadoop, no sólo se reduce a implementar el paradigma MapReduce sino que ofrece muchas más posibilidades de manipulación de datos. Todas las operaciones disponibles se dividen en transformaciones y acciones sobre unas estructuras de datos llamadas *Resilient Distributed Datasets* (RDDs). Estas estructuras contienen los datos sobre los que se va a trabajar y están distribuidas sobre los diversos nodos favoreciendo la localidad de datos a la hora de computar, ya que pueden mantenerse dentro de la memoria principal del nodo.

La arquitectura de Spark incluye dos actores fundamentales: controlador (*driver*) y trabajador (*worker*). El *driver* es el encargado de lanzar la aplicación y de su manejo dentro del clúster de Spark. Para ello se comunicará con el gestor de recursos elegido. El *worker* es el proceso principal de cada nodo, encargado de lanzar los diversos ejecutores (*executors*), que serán los que lleven a cabo las distintas tareas.

B. PyCUDA

En este trabajo, se ha considerado Python como lenguaje de programación por su utilización en computación científica para prototipado. Además se tiene en cuenta la gran cantidad de módulos científicos disponibles. Estos módulos poseen la mayoría de funciones matemáticas utilizadas en este tipo de aplicaciones, con el añadido de que están altamente optimizadas gracias a la utilización del lenguaje C. Aún así, el proceso descrito posteriormente podría aplicarse a cualquier otro lenguaje siempre que posea *bindings* para controladores de GPUs. En este caso, para la conexión con los aceleradores GPU, se ha utilizado PyCUDA, un módulo que posee diversas funciones que ponen a disposición del programador de Python la API del driver CUDA [5]. Su mayor ventaja es la posibilidad de reutilización de las funciones ya programadas para CUDA, de ahora en adelante *kernels*, previamente programados en C/C++.

III. Propuesta de arquitectura para un clúster heterogéneo de Spark

Como se puede ver en las Figuras 1 y 2, desde el punto de vista de la ejecución de la aplicación hay tres componentes principales: el *driver* de la aplicación, los *workers* en cada uno de los nodos del clúster y su correspondiente *GPU scheduler*.

El *driver* es el actor encargado de ejecutar el código principal de la aplicación incluyendo la gestión de los datos. En el caso de usar un sistema de ficheros distribuido como HDFS, cada uno de los nodos será el encargado de adquirir aquellos datos más cercanos, favoreciendo la localidad de datos en el proceso de ejecución. El *driver* es independiente de la ejecución de las tareas, por lo que no es necesaria la presencia de GPUs en el nodo desde el cual se ejecuta. Esto permite la ejecución de tareas que hacen un uso exclusivo de la GPU en todos aquellos nodos que la posean, pudiendo aprovechar nodos no heterógeneos para el lanzamiento del *driver*.

En el *executor* se lleva a cabo la ejecución de las tareas, que incluye la ejecución de los kernels de GPU. Para ello, primero se comprueba dentro de la tarea la compatibilidad del nodo al cual ha sido asignada. Si existe dicha compatibilidad, entonces procederá a la compilación y preparación de argumentos y datos para el kernel, así como a la conexión con el *scheduler* presente en el mismo nodo. De esta manera, primero ejecutará el kernel correspondiente en la GPU más adecuada y, después, deberá obtener los datos de la GPU para poder terminar la ejecución de la correspondiente tarea.

El *scheduler* es un servicio independiente encargado de la planificación y selección de los dispositivos en los cuales se ejecutarán los kernels. Debido a su característica de servicio independiente, es capaz de planificar las GPUs incluso entre aplicaciones independientes de Spark que se estén ejecutando dentro de un mismo nodo. Para la planificación se han elegido diversas políticas que tendrán en cuenta factores como los recursos disponibles en el dispositivo, el

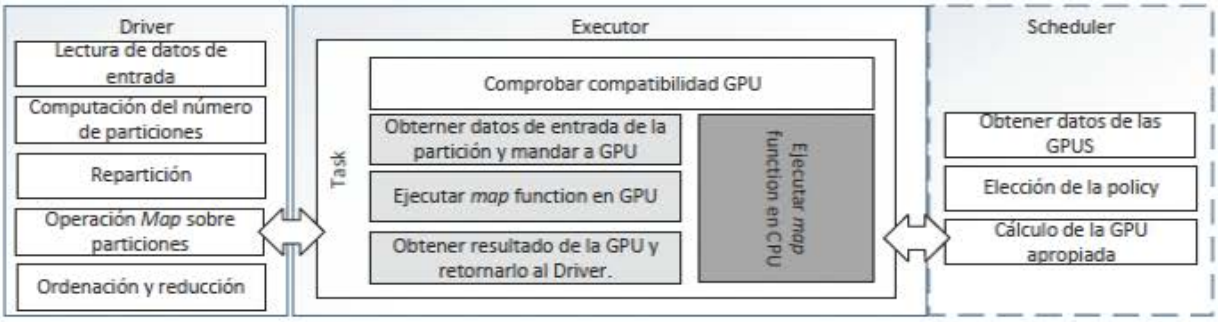


Fig. 1. Flujo general de una aplicación de GPU dentro de Spark.

rendimiento ofrecido, número de tareas en ejecución, etc. Esta aproximación permite aumentar el número de kernels concurrentes en la plataforma.

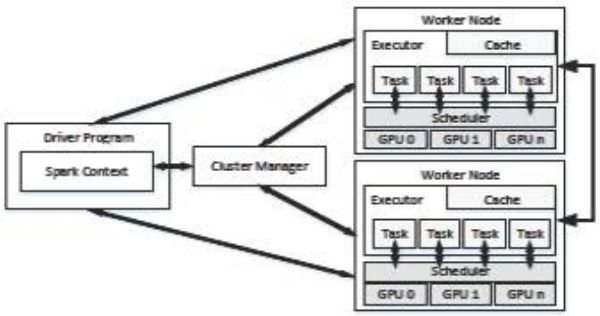


Fig. 2. Flujo de la retroproyección en Spark del simulador.

A. *Planificador de dispositivos GPU para tareas MapReduce*

Una de las principales limitaciones para la migración del paradigma MapReduce a entornos heretogéneos es la gestión eficiente de los aceleradores. Esto es especialmente relevante en despliegues con múltiples dispositivos, dado que, actualmente, las tareas generadas por el motor de flujos de trabajo son ajenas a los recursos disponibles. Para resolver este problema, nuestra solución incorpora un planificador que facilita dicha integración. Para ello, se ha diseñado e implementado una solución software que permite asignar GPUs a cada una de las tareas de forma eficiciente, basándonos en la invocación de procesos remotos (RPC).

El sistema está compuesto por una biblioteca invocada por las tareas y un proceso servidor para cada uno de los nodos de cómputo. El funcionamiento es el siguiente. En primer lugar, al inicio de cada tarea, se solicita al servicio de planificación qué GPU está disponible dados unos requisitos de capacidad de memoria dados. En segundo lugar, el planificador resuelve qué dispositivo está disponible, teniendo en cuenta el estado (por ejemplo, memoria disponible, procesos en ejecución, etc.) de cada dispositivo. Es importante destacar que en caso de no contar con los recursos de ejecución disponibles, está invocación quedará bloqueada. En tercer lugar, una vez obtenido el dispositivo disponible, en la tarea se selecciona el contexto de ejecución para la GPU dada. Finalmente, una vez concluida la ejecución de la tarea, se comunica al planificador que libere los recursos anteriormente reservados.

Para lograr un uso eficiente de los dispositivos, el planificador cuenta con distintas políticas de ejecución ya implementadas:

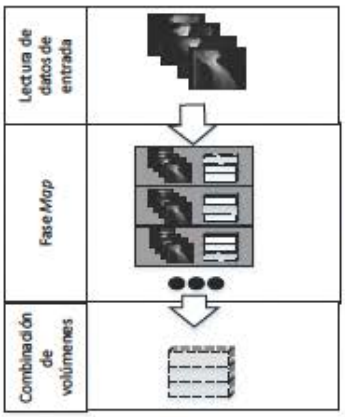
- Aleatoria (*random*): se elige una GPU de forma aleatoria de aquéllas disponibles.
- Dispositivo con menos procesos(*least processes*): se elige aquella GPU con menos tareas en ejecución.
- Round-robin: se asigna una GPUs de forma consecutiva a cada tarea que llega.

IV. Caso de uso en el campo de la imagen médica

Debido al aumento de la resolución en los dispositivos actuales y a las nuevas técnicas de análisis de imagen, el volumen de datos producidos por este tipo de aplicaciones se ha incrementado notablemente. La obtención de las imágenes finales pasa por, en muchos casos, la ejecución de algoritmos con una gran carga computacional. La mayoría de estos algoritmos han sido optimizados para adaptarlos a aceleradores de cómputo tales como Intel Xeon Phi o GPUs. Sin embargo, estos dispositivos no poseen la cantidad de memoria necesaria para generar imágenes en 3D en alta resolución, las cuales pueden llegar a ocupar más de 10 GB.

Un ejemplo de este tipo de problemas es la reconstrucción y los procesos de simulación en el campo de la imagen médica, la cual utiliza algoritmos cada vez computacionalmente más complejos. Es el caso del simulador de rayos-X Fux-Sim, que utiliza varios métodos de reconstrucción de imagen de Tomografía Axial Computerizada (TAC) y simula distintas geometrías de adquisición de datos.

En el presente trabajo, se ha migrado una parte del simulador al *framework* de computación distribuida Apache Spark: la retroproyección. La retroproyección es la parte esencial de la reconstrucción, ya que se encarga de calcular los valores de cada uno de los vóxeles que forman la imagen final en 3D. En nuestro caso, el algoritmo de reconstrucción elegido realiza la integral de cada una de las líneas de proyección y tiene su origen en un algoritmo tradicionalmente




```
read_projections = read_binary_image()

read_projections_sc = sc.broadcast(read_projections)



size_volume_rdd = sc.parallelize([1..1..z_size_volume])
volume_result = size_volume_rdd.mapPartitions(lambda iterator:
backproject(iterator, read_projections_sc))



volume_iterator = volume_result.toLocalIterator()
for i in volume_iterator:
     write_binary_image(i)
```

Fig. 3. Flujo de la retroproyección en Spark del simulador.

utilizado, el FDK [6]. Estas líneas de proyección comienzan en la fuente de Rayos-X y pasan por cada uno de los vóxeles a calcular para terminar en las imágenes de entrada, donde se produce una interpolación lineal.

El cómputo de cada una de las líneas de proyección es completamente independiente, lo que permite la paralelización del algoritmo. Debido a la gran cantidad de valores a calcular y a la facilidad de utilización de los algoritmos de interpolación, tradicionalmente este tipo de algoritmos han sido implementados y optimizados para aceleradores de tipo SIMD como las GPUs. Utilizando la arquitectura definida en la sección anterior se puede reutilizar los kernels existentes junto con la distribución de tareas que ofrece Spark para proporcionar una ejecución acelerada gracias a la arquitectura GPU. Las operaciones llevadas a cabo así como un pseudocódigo pueden verse en la Figura 3. Los datos de entrada son leídos en el *driver* y retransmitidos a cada uno de los *executors* para su procesamiento. Durante la fase *map*, cada partición tiene asignadas un determinado número de rodajas del volumen a reconstruir, éstas serán las que se devuelvan posteriormente. El RDD resultante será el que finalmente se escriba, bien en un sistema de ficheros local en el *driver* o de forma distribuida a través de HDFS. Esta última característica permite el tratamiento de grandes estudios.

V. Evaluación

La evaluación se ha realizado en un nodo de cómputo compuesto por dos procesadores Intel(R) Xeon(R) CPU E5-2620 0 a 2.00GHz y 3 GPUs, 1 Tesla K40c (12 GB de memoria GDDR5) y 2 Geforce GTX Titan (6GB de memoria GDDR5). En las pruebas en las que así se especifique se utilizarán estas GPUs con la memoria limitada a 2GB.

El sistema está supervisado mediante Cloudera 5.7. La versión de Spark sobre la que se ha trabajado es 1.6 en modo de ejecución *stand-alone*, los ficheros de entrada se encuentran en un SSD local y se escriben los ficheros resultantes en un directorio almacenado en HDFS, también sobre SSDs y una red Ethernet 10 Gbps. La versión de python utilizada es 2.7 y de pyCUDA 1.3. Los datos utilizados en la evaluación consisten en 360 imágenes (radiografías) de 1024x1024 píxeles. El volumen total reconstruido es de 1024x1024x1024 vóxeles, formando un total de 4 GB de datos de salida.

A. Evaluación del tiempo de ejecución en GPUs

Debido a la sobrecarga de la ejecución del runtime asociado a Apache Spark los tiempos de ejecución para vólumenes estándar de $1024x1024x1024$ vóxeles (4 GB) de la aplicación de retroproyección distribuida en un nodo no son totalmente competitivos con la ejecución del simulador en Fux-Sim para el caso del mismo nodo y número de GPUs. En la Figura V izquierda, mostramos los tiempos de ejecución para el simulador y la aplicación en Spark para distintas GPUs. La configuración de Spark consistía en 3 threads y 3 particiones, lo más parecido a la configuración del simulador en modo multiGPU. En todos los casos Spark requiere más tiempo para completar la ejecución sin embargo se puede apreciar que según aumenta el número de GPUs utilizadas esta diferencia se reduce. Esto es debido al mejor aprovechamiento de los recursos por nuestra arquitectura gracias al planificador, como veremos posteriormente.

En la Figura V derecha mostramos los tiempos de ejecución de la aplicación en Spark para 3 hilos, distinto número de particiones y las 3 políticas implementadas. Estos experimentos se hicieron estableciendo un límite de memoria, con el objetivo de emular dispositivos de bajo coste (familia GTX). A partir de los resultados, se puede observar que para el caso de un sólo nodo el aumento del número de particiones influye negativamente en los tiempos de ejecución, debido principalmente al aumento del uso de la memoria. De las tres políticas la única que se diferencia claramente del resto negativamente es la política aleatoria mientras que Round Robin y menos procesos se mantienen igualadas, aunque como veremos posteriormente, con un mayor número de hilos paralelos y un studio de la ocupación sí se aprecian diferencias significativas.

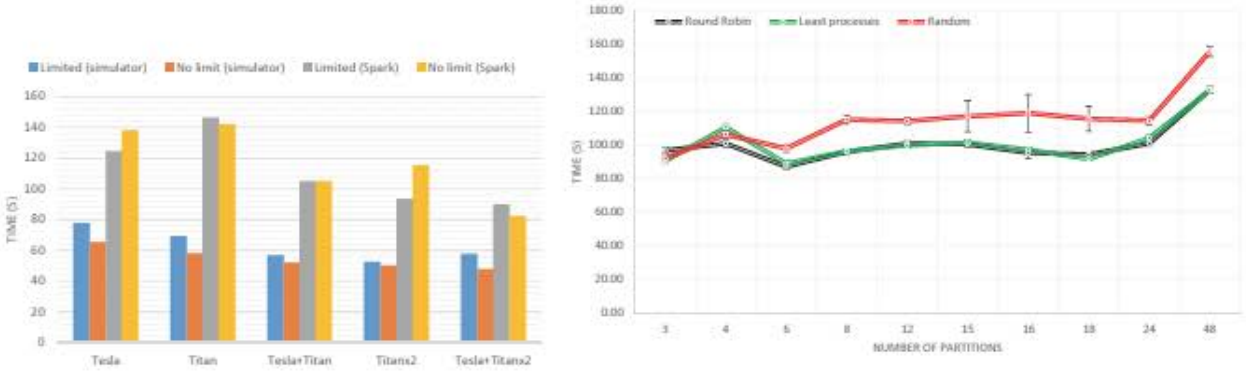


Fig. 4.A la izquierda: tiempos de ejecución para las distintas combinaciones de GPUs en un nodo, con limitación de memoria, sin limitación, en el simulador original y en su versión en Spark. A la derecha: resultados de la evaluación de la aplicación con 3 hilos y diferente número de particiones para cada una de las políticas y su correspondiente error estándar.

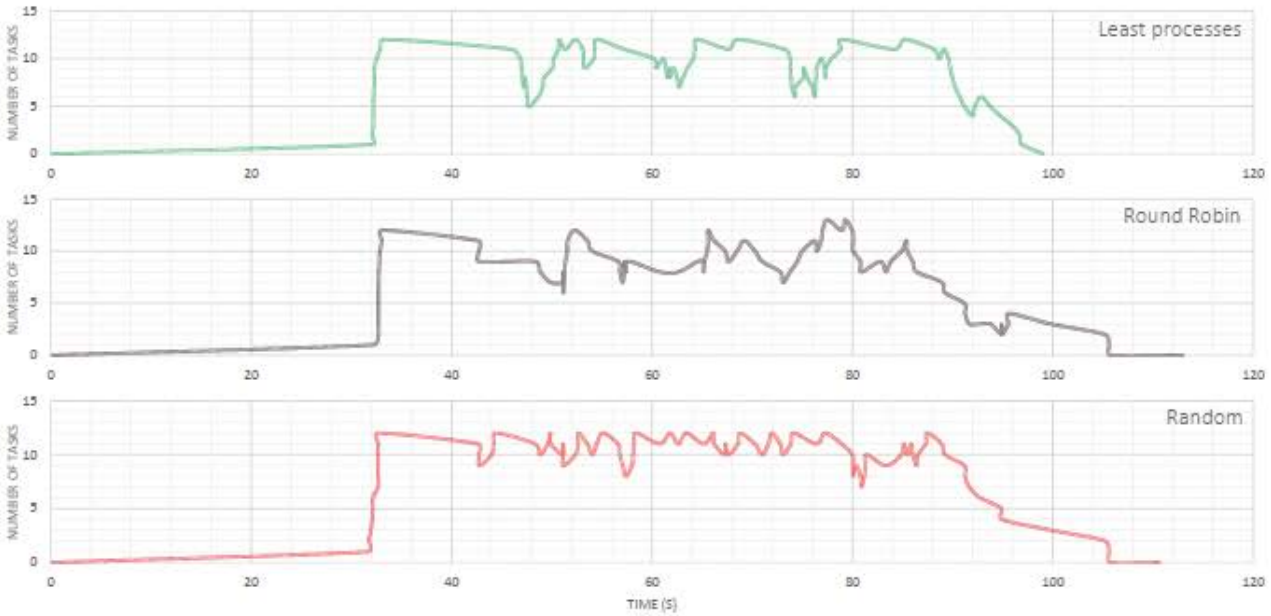


Fig. 5.Resultados de la evaluación de la aplicación con 12 hilos y 48 particiones para cada una de las políticas. Se muestra el número de tareas ejecutadas en cada momento, en segundos, por las GPUs.

B. Ocupación de las GPUs en las distintas políticas

El objetivo de la capa *scheduler* es el aprovechamiento de las GPUs de los nodos en los cuáles se está ejecutando la aplicación. En la Figura 5 mostramos la ejecución de la aplicación sobre 12 threads y 48 particiones para cada una de las políticas implementadas. Para emular un sistema homogéneo, la memoria total de las GPUs ha sido limitada a 2GB. Como se puede ver, la política que planifica en función del número de procesos resulta en un mejor tiempo de ejecución, tal y como habíamos confirmado anteriormente. Una de las razones para este menor tiempo de ejecución es el aprovechamiento regular de los recursos disponibles en el nodo. Tanto Round Robin como la política aleatoria poseen numerosos picos de ocupación, mientras que con la primera política se mantiene estable para la ejecución de 12 tareas. Este valor resulta ser el número ideal ya que es el número máximo de threads que ejecutan en paralelo. Aún así, se aprecian tres caídas regulares de la ocupación, las cuales corresponden con la finalización de los paquetes de tareas formados por 12 cada uno. Por ello, en un sistema homogéneo, la política que favorece los dispositivos con menos procesos es la que lleva a un mejor reparto de tareas.

VI. Trabajos relacionados

La unificación de arquitecturas heterogéneas con frameworks de computación distribuida (por ejemplo Apache Hadoop) ha sido tratado en diversos trabajos desde distintos puntos de vista. En Hadoop existen varios ejemplos de esta unificación, bien centrados en modelos de programación concretos basados en GPU como HadoopCL [7] o centrados en la programación de los aceleradores para ofrecer soporte tanto a OpenCL como CUDA, como es el caso de Hadoop+ [8]. En cuanto a Apache Spark, está actualmente en desarrollo una mejora del framework llamada HeteroSpark [9] basada en Java RMI, sin embargo no se encuentra disponible actualmente. La mayoría de estos frameworks se basan, al igual que en nuestro trabajo, en la compilación de kernels programados, aunque la planificación de los trabajos mandados a

la GPU no está incluida, reduciendo la capacida de mejora.

Sin embargo, otros trabajos relacionados han optado por no modificar el framework distribuido y añadir GPUs a nivel de aplicación. Ejemplos de este procedimiento son, por citar algunos, el presentado por Zheng et al. [10] que utiliza GPUs dentro de Hadoop para acelerar el algoritmo *kmeans* o [11] que hace uso de Spark un servidor separado para las GPUs para el análisis de imágenes de resonancia magnética. Nosotros hemos optado por una solución similar, pero dentro del framework de Apache Spark.

Más allá del uso de aceleradores GPUs, en estos entornos distribuidos algunos trabajos en el campo de la imagen médica se han centrado en su evaluación en entornos sin presencia de aceleradores como por ejemplo [12], que implementa el algoritmo de tomografía computerizada FDK en su versión 4D utilizando el paradigma MapReduce en Hadoop. Otra solución [13], implementa una deconvolución en Spark y lo compara con otras alternativas como el uso de GPUs o multicore CPUs.

VII. Conclusiones

Este trabajo ha presentado una prueba de concepto del uso de GPUs dentro del *framework* Apache Spark a través de la adaptación de una aplicación de reconstrucción de imagen médica. Se han detallado las características de la solución y de las herramientas necesarias para llevarlo a cabo esta tarea. El propotipo se basa en el conector PyCUDA, aunque se puede generalizar a otros modelos de programación en aceleradores de cómputo, así como otros lenguajes de programación que tengan disponibles estos conectores. Además se ha provisto de un planificador adicional intranodo en la plataforma Spark para el mejor aprovechamiento de los recursos disponibles. Este planificador posee diversas políticas que han sido evaluadas sobre un sólo nodo, demostrando su utilidad cuando comparamos con la ejecución de la alicación original. Cuando se aumenta el número de GPU a utilizar, nuestra arquitectura aprovecha de mejor manera los recursos de cómputo disponibles. Esto último se aprecia especialmente cuando se hace uso de una política que tiene en cuenta parámetros dinámicos de la GPU, como el número de procesos en ejecución.

El siguiente paso de este trabajo consistirá en la evaluación sobre varios nodos, así como en ampliar el planificador de GPU a nivel de clúster para un mejor reparto de los trabajos. Además se prevé utilizar el mismo procedimiento en otras aplicaciones de procesamiento de imagen médica, como algoritmos iterativos de reconstrucción, para su posterior combinación con flujos de análisis de las imágenes resultantes.

Agradecimientos

Este trabajo ha sido parcialmente subvencionado por el Ministerio de Economía y Competitividad, bajo el proyecto TIN2013-41350-P, *Scalable Data Management Techniques for High-End Computing Systems*. También queremos agradecer a Nvidia por proporcionar la tarjeta Tesla K40 con la que se han realizado los experimentos.

Referencias

[1] Felix Garcia-Carballeira Jesus Carretero Perez Silvina Caino-Lores, Alberto Garcia Fernandez, "A cloudification methodology for multidimensional analysis: Implementation and application to a railway power simulator," vol. 55, pp. 46–62, June 2015.

[2] Javier Garcia Blas, Monica Abella, Florin Isaila, Jesus Carretero, and Manuel Desco, "Surfing the optimization space of a multiple-gpu parallel implementation of a x-ray tomography reconstruction algorithm," *Journal of Systems and Software*, vol. 95, pp. 166 – 175, 2014.

[3] Claudia Molina Ines Garcia Jesus Carretero Manuel Desco Estefania Serrano, Javier Garcia Blas and Monica Abella, "Design and evaluation of a parallel and multiplatform cone-beam x-ray simulation framework," July 2016.

[4] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[5] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[6] LA Feldkamp, LC Davis, and JW Kress, "Practical cone-beam algorithm," *JOSA A*, vol. 1, no. 6, pp. 612–619, 1984.

[7] Max Grossman, Mauricio Breternitz, and Vivek Sarkar, "Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1918–1927.

[8] Wenting He, Huimin Cui, Binbin Lu, Jiacheng Zhao, Shengmei Li, Gong Ruan, Jingling Xue, Xiaobing Feng, Wensen Yang, and Youliang Yan, "Hadoop+: Modeling and evaluating the heterogeneity for mapreduce applications in heterogeneous clusters," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 143–153.

[9] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao, "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms," in *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 347–348.

[10] HuanXin Zheng and JunMin Wu, "Accelerate k-means algorithm by using gpu in the hadoop framework," in *Web-Age Information Management*, pp. 177–186. Springer, 2014.

[11] Roland N Boubela, Klaudius Kalcher, Wolfgang Huf, Christian Našel, and Ewald Moser, "Big data approaches for the analysis of large-scale fmri data using apache spark and gpu processing: A demonstration on resting-state fmri data from the human connectome project," *Frontiers in neuroscience*, vol. 9, 2015.

[12] Bowen Meng, Guillem Pratx, and Lei Xing, "Ultrafast and scalable cone-beam ct reconstruction using mapreduce in a cloud computing environment," *Medical physics*, vol. 38, no. 12, pp. 6603–6609, 2011.

[13] Lianyu Cao, Penghui Juan, and Yinghua Zhang, "Real-time deconvolution with gpu and spark for big imaging data analysis," in *Algorithms and Architectures for Parallel Processing*, pp. 240–250. Springer, 2015.