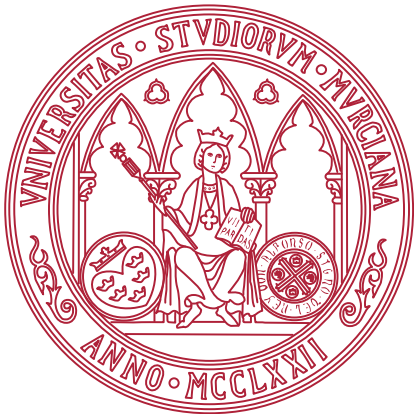
UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

TRABAJO FIN DE GRADO

TÉCNICAS DE AUTOOPTIMIZACIÓN DE RUTINAS BÁSICAS DE ÁLGEBRA LINEAL EN SISTEMAS MULTICORE+MULTIGPU

FRANCISCO JOSÉ HERRERA ZAPATA



Trabajo dirigido por

Antonio Javier Cuenca Muñoz
Domingo Giménez Cánovas

Codirector:
Luis Pedro García González, Universidad Politécnica de Cartagena

curso 2016-2017

Técnicas de Autooptimización de Rutinas Básicas de Álgebra Lineal en Sistemas Multicore+MultiGPU

Francisco José Herrera Zapata

Resumen

El propósito de este Trabajo Fin de Grado (TFG) es presentar a las rutinas de auto-optimización como herramientas que ayuden a mejorar el rendimiento de operaciones ya optimizadas con librerías BLAS, como puede ser la multiplicación matricial.

Para ello, se presenta la evolución de la multiplicación matricial hasta llegar a operar con ella utilizando rutinas optimizadas, como son las librerías BLAS, para cada tipo específico de hardware. En el caso del presente trabajo para CPU multicore y multiGPU, y desde un entorno homogéneo donde los dispositivos manycore, GPUs, serán todos iguales, hasta un entorno más heterogéneo donde las GPUs podrán ser distintas.

Una vez defina la operación, lo que se propone cuando se va a utilizar una técnica de auto-optimización es evaluar en términos de rendimiento cuál sería la mejor forma de repartir la carga de trabajo entre los dispositivos implicados. Se aplica la rutina que se pretende auto-optimizar sobre un conjunto de datos de entrada, que serán los tamaños de las matrices con las que trabajar, y se obtiene como resultado la configuración más óptima para cada uno de los tamaños.

Para conseguir estos resultados se estudiarán dos posibles técnicas de auto-optimización para la Multiplicación de Matrices. Por un lado la *búsqueda exhaustiva*, es decir, realizar una búsqueda completa por todo el espacio de soluciones obteniendo de esta forma la mejor configuración. Esta técnica, aunque se hayan realizado experimentos con ella para tamaños de problema pequeños, será inviable su uso en general ya que si se aplica sobre un conjunto real de datos los tiempos de ejecución serían desorbitados, no pudiendo realizarse la parte que denominamos instalación de la rutina dentro del proceso de auto-optimización.

La segunda técnica utiliza métodos heurísticos para reducir la búsqueda por el espacio de soluciones, intentando buscar sólo en aquellas zonas donde se crea que puede haber configuraciones óptimas o cercanas a ellas. El proceso que utiliza es el siguiente:

1. Dado un conjunto de datos de entrada denominado *conjunto instalación*, que contiene varios tamaños de matriz ordenados de menor a mayor, se toma el primer elemento (el menor) y se aplica el proceso de búsqueda exhaustivo, obteniendo y registrando como la mejor configuración posible para ese tamaño.

2. Esta mejor configuración se utilizará para comenzar la búsqueda para el siguiente tamaño, utilizando para su construcción técnicas heurísticas. Se evaluarán distintas opciones de generar la nueva configuración: proporcional, sobrecargar la unidad de cómputo que más carga tenga asignada al estimar que es la más rápida, distribuir entre todas las GPUs y la CPU la misma cantidad nueva de trabajo, o la media de las 3 opciones anteriores para cada unidad de cómputo.

3. Una vez definida y evaluada la nueva configuración se busca en su entorno la mejor opción.

4. La búsqueda finalizará una vez que uno de los elementos evaluados sea peor que el valor óptimo encontrado hasta el momento. Para no restringir tanto la búsqueda se aplicará un valor umbral, es decir, aunque la configuración evaluada sea peor que la óptima, si la diferencia está por debajo del umbral la búsqueda prosigue, y en caso contrario se detendría.

5. Una vez determinada la mejor configuración para una entrada, esta se almacena y se convierte en la semilla que generará la siguiente configuración para búsqueda en un nuevo valor del *conjunto instalación*.

Además del proceso de generar la semilla para el siguiente tamaño y el uso de un umbral para detener la búsqueda hay otro elemento que influye en el algoritmo, el estudio y evaluación de la vecindad. Se ha experimentado con dos posibilidades: vecindad aleatoria sin superar el tamaño de la vecindad al número de dispositivos de cómputo, o completa, donde el número de vecinos analizados aumentaría a $n * (n - 1)$ siendo n el número de elementos de cómputo, 1 CPU + $(n - 1)$ GPUs.

Desarrollados los algoritmos y probados en entornos homogéneos y heterogéneos, se llega a la conclusión de que las técnicas de auto-optimización sobre operaciones básicas de álgebra lineal (BLAS) ayudan a aumentar las prestaciones de los sistemas donde se instalan. El proceso de auto-optimización presentado se lleva a cabo de forma sencilla, dura en el peor de los casos unas pocas horas, y permite obtener configuraciones cercanas a la óptima de manera autónoma, con lo que es una herramienta útil para obtener buenas prestaciones independientemente de la experiencia en paralelismo del usuario. La auto-optimización es especialmente útil cuando el número de dispositivos y la heterogeneidad del sistema aumenta, lo que dificulta su explotación eficiente por los usuarios finales de la rutina.

Extended Abstract

The purpose of this project is to present auto-tuning routines as tools that help improve the performance of operations already optimized with BLAS libraries, such as matrix multiplication.

Since the creation of the first computers, these libraries have been used as support and to accelerate mathematical calculations in various branches of science and engineering. With their use, investigators have attempted to obtain more optimal results or at least closer to optimal results in a faster manner, or otherwise to use the same computation time to obtain results with greater precision. In fact, thanks to them, problems that previously were considered irresolvable have been able to be resolved. At present, the computers that are mostly used in these branches are called supercomputers. These computers have millions of computation cores and a heterogeneous nature. They are composed of several computational components, normally multicore CPUs and manycore coprocessors (GPUs or MICs).

To give an example, according to the current Top500.org ranking that dates from June 2017, the Chinese supercomputer Sunway TaihuLight, with its 10,649,600 cores, is the world's fastest computer, with a maximum theoretical Rmax performance of 93,014.6 TFlop/s. The second, Tianhe-2, is a cluster that combines Intel Xeon E5-2692v2 multiprocessors with Intel Xeon Phi 31S1P, with a total of 3,120,000 cores and a Rmax performance of 33,862.7 TFlop/s. In third place is the Swiss Piz Daint - Cray XC50, which uses a Xeon E5-2690v3 12C 2.6GHz processor along with NVIDIA Tesla P100, obtaining a maximum Rmax performance of 19,590 TFlops/s.

All this computing power needs the software to be able to correctly distribute all the work to all the computational components, so that they can function at full capacity. The appearance of Basic Algebra Linear routines, which are able to optimize the computational operations most used in numerical computation, together with the development of parallel programming, which implements these routines for parallel devices such as multicore CPUs or manycore devices (GPUs and MICs), have become the necessary tools to attain the maximum potential of these equipments.

One of these sets of routines is BLAS (Basic Linear Algebra Subprograms), a computational kernel that provides a series of basic routines for performing vector-vector operations at level 1, vector-matrix at level 2, and matrix-matrix at level 3. These computational kernels, due to their efficiency and portability, are widely available and used in the development of high quality linear algebra software. A particular implementation is MKL. Along with the development of multiprocessor or multicore libraries, such libraries have also been developed for manycores, for example the cuBLAS library for GPUs. Both libraries will be used in the development of this project.

With the help of the BLAS MKL and cuBLAS libraries, an optimized CPU + MultiGPU matrix multiplication algorithm will be developed, but it will require support in order

to work at full efficiency. This help is provided by **auto-tunning techniques for basic linear algebra operations**. Specifically, the matrix multiplication is used as case study. What this auto-tuning routine will do is to study the performance of each of the computing devices of a system working together on an operation (the multiplication of matrices) thus obtaining the workload each computing device must run according to the input size of the matrix. Thus, once the auto-tuning routine is installed for a set of matrices ordered by size from less to greater, an auto-tunning matrix multiplication operation will be obtained which, depending on the size of the input, will be configured optimally thus increasing the performance of the matrix multiplication. These auto-tuning techniques are the object of our study, and we want to show that their use helps to increase performance.

We consider a matrix multiplication in the form $C = \alpha AB + \beta C$, to be computed in a CPU + MutiGPU environment, with matrix B distributed among the n computing devices ($(n-1)$ GPUs plus one CPU). The multiplication of matrices is organized as $C = \big(\alpha AB_{gpu_1} + \beta C_{gpu_1} | \alpha AB_{gpu_2} + \beta C_{gpu_2} | \cdots | \alpha AB_{gpu_{n-1}} + \beta C_{gpu_{n-1}} | \alpha AB_{cpu} + \beta C_{cpu} | \big)$. Each block B_i represents to part of matrix B assigned to the computatinal component i. Although operations will be performed efficiently when using BLAS libraries, the same can not be said of performance due to the workload of each device. This is where the potential of auto-tuning routines is observed. They are able, once installed, to create an auto-tuning routine of matrix multiplication and an optimal configuration depending on the size of the input of the matrices to be operated.

The auto-tuning techniques for the matrix multiplication to be studied are **Exhaustive Search** and **Guided Search**. The purpose of these two techniques is the same. Installing the auto-tuning routine, taking as input an $installationset$ containing matrix sizes used in the system, ordered from lowest to highest, as a result you will get the auto-tuning operation of the matrix multiplication that will contain a configuration vector for each of the inputs. This configuration vector has the form $< N, gpu_1, gpu_2, ..., gpu_{n-1}, cpu, th >$, where N represents the size of the matrix, each gpu_i is the number of columns of matrix B assigned to the i-th GPU, cpu the corresponding number of columns assigned to the CPU, and th the number of threads to work on the CPU. When a developer uses the auto-tuning matrix multiplication, given the size n_E, the auto-tuning engine will use to solve the problem the configuration obtained in the installation process for the closest value to n_E in the $installation\ set$.

What the *auto-tuning technique with exhaustive search* does is to explore the whole solution tree of the problem until it finds the best solution. The distribution of the matrix B is a distribution by columns. Nevertheless, performing all the checks column by column would be very costly in terms of runtime. In addition, many checks would be redundant since the influence of a single column is null in the global count. For this reason, the distribution of the columns of the B matrix is done by blocks of columns of a given size, tb. This size is an estimation that is introduced as a parameter in the process of installing the auto-tuning routine, and its selection is responsibility of the manager or administrator who performs the process. The size should be large enough so that there are variations in

the exploration of the configurations, but small enough to detect variations in the performance by small movements of data.

Using the **backtracking technique**, the algorithm will perform an in depth backtracking search, until it obtains the best configuration for the indicated size. This process is repeated for each of the entries in the $installation\ set$. In spite of obtaining the most optimal configuration, this algorithm becomes practically unfeasible due to the exorbitant growth of the execution time when the number of devices, the size of the inputs or the number of elements in the $installation\ set$ increase.

The *auto-tuning guided search technique* uses heuristic techniques to drastically reduce the installation time, but the configurations will be close to the optimum, althought not necessarily the best. The mechanism used is as follows:

1. Given an input data set called *installation set*, which contains several array sizes ordered from least to greatest, the first element (the smallest) is taken and the exhaustive search process is applied for it, obtaining and registering the best configuration possible for that size.

2. This best configuration will be used to begin the search for the next size, using heuristic techniques for its construction. Different options for generating the new configuration will be evaluated: i) proportional method, ii) overloading the unit of computation that has the most load assigned to it, iii) distributing between all the GPUs and the CPU the same new amount of work, or iv) the average of the 3 previous options.

3. Once the new configuration is defined and evaluated, the best combination in its neighborhood is obtained.

4. The search will end once one of the elements evaluated is worse than the optimal value found so far. In order not to restrict the search so much, a threshold value will be applied. That is, even if the evaluated configuration is worse than the optimum, if the difference is below the threshold, the search continues. In other case, the search stops.

5. Once the best configuration for an input is determined, it is stored and converted into the seed that will generate the next configuration to search for a new value in the installation set.

Two ways of evaluating the neighborhood have been implemented. In the random approach, n elements are evaluated (n coincides with the number of computal devices), with the elements obtained redistributing tb columns of matrix B. Tabu techniques are uses to avoid reevaluation of configurations. In that way, the neighborhood has a total of $n*(n-1)$ elements. The $threshold$ determines when the search stops. For a configuration with a worse performance than the best at that moment, but with a difference lower than the $threshold$, the search continues. When the difference is greater, the search for that

entry size will end.

Experimental tests for the *exhaustive search installation* have been carried out for small configurations with which to observe the possible potential. However it has not been possible to be evaluated in a real environment. Experimental tests using *guided search installation* have been carried out in a homogeneous environment of CPU + 2 GPUs and in a heterogeneous environment of CPU + 6 GPUs.

The methodology used in both cases is the following:

1. Guided installation with the values of a $validation\ set$, in order to record configurations that obtain the highest performance for each input size.
2. Guided installation with the values of the $installation\ set$, in order to record configurations that obtain the highest performance for each installation size.
3. Using the information from the guided installation for the $installation\ set$, multiplication operations for matrices of the sizes in the $validationset$ are carried out with the auto-tuning technique using the information stored for the *installation set*.
4. The times obtained for the *validation set* when these sizes are used in the installation are compared with those obtained with the auto-tuning technique.

The analysis will be repeated for random neighborhood and complete neighborhood. In addition, for each one of them, the installation will be done for $threashold$ values 2 %, 5 % and 10 %. The installation time for *installation with the guided search* is between 4 and 10 min in a homogeneous environment, depending of the type of neighborhood and the $thredshold$. The highest workload is assigned to the GPUs, with assignations to the GPUs higher than 90 % when the input size increases. Regarding the distribution of operations in the GPUs, predictably, it is usually around 50 % of the total GPU work for each one. This was expected in a homogeneous system.

The validity of the auto-tuning methodology is tested by comparing the performances of the auto-tuning matrix multiplication for the sizes in the *validation set* with those obtained when the installation is carried out for these sizes. The performance is close in most cases,at the expense of low installation times when the *guided search* is used.

The methodology and the process used in the heterogeneous environment were the same but the results were even more conclusive. In an environment whose behaviour is difficult to predict due to the diversity and number of computing devices, the auto-tuning methodology proves to be an useful tool for efficiently exploiting all the computational resources in the system. In this case, the installation times range from 3 minutes 27 seconds, for a guided installation using a random neighborhood and $thredshold = 2\,\%$ to just over an hour for the search with complete neighborhood and a $thredshold = 10\,\%$. The results of these experiments have demonstrated that the performance of the auto-tuning routine is the same as that of a native installation.

Guided installation has been shown to be effective in homogeneous and heterogeneous environments. In fact, it is in heterogeneous environments where the auto-tuning techniques for BLAS routines have shown their strength. Near-optimal performance can be achieved with the auto-tuning techniques analysed, significantly improving the performance of operations without users intervention.

The **auto-tuning techniques of BLAS routines for CPU + multiGPU**, in particular, the one developed in this document for the matrix multiplication operation, has proved to be a more than valid option for this task. It facilitates the task of the developer and leaves the work of performing the desired operation efficiently in the hands of the auto-tuning routine.

Índice

1. Introducción

En este capítulo introductorio se comentará la gran capacidad de cómputo que alcanzan actualmente los ordenadores más potentes del planeta, los supercomputadores, y su repercusión en la investigación y la tecnología. Todo este potencial hardware, junto con el desarrollo de librerías matemáticas y estadísticas, ha ayudado a explotar al máximo el rendimiento que pueden llegar a alcanzar estos equipos. Pero esta tarea implica una dificultad añadida debido a la heterogeneidad en la composición hardware de los ordenadores, especialmente en los últimos años, con la aparición de CPUs multicore y sistemas manycore como GPUs o MICs, y la necesidad de que estas librerías evolucionen y también sean soportadas por estos nuevos componentes.

También se describirá el plan de trabajo que se va a seguir para la realización del Trabajo Fin de Grado (TFG) y el entorno donde se realizarán las pruebas pertinentes, finalizando el capítulo con la estructura que se va a seguir en este documento.

1.1. Introducción

Desde la creación de los primeros ordenadores estos se han utilizado como soporte, apoyo y aceleración de cálculos matemáticos en diversas ramas de la ciencia y la ingeniería. Con su uso, se ha pretendido obtener resultados más óptimos o más cercanos a ellos de una forma más rápida, o bien, empleando el mismo tiempo de cómputo, pero consiguiendo resultados con mayor precisión. De hecho, gracias a ellos, se han conseguido resolver problemas que anteriormente se consideraban irresolubles. En la actualidad, los ordenadores que mayormente se utilizan en estas ramas son los denominados supercomputadores. Estos equipos tienen millones de cores de cómputo y una naturaleza heterogénea al estar formados por distintos componentes, tanto CPUs multicore como manycore, como pueden ser GPUs o MICs.

Por poner varios ejemplos, según el ranking actual Top500.org que data de junio de 2017, el chino Sunway TaihuLight, con 10,649,600 de cores, es el computador más rápido del mundo, con un rendimiento máximo teórico Rmax de 93,014.6 TFlop/s. El segundo, Tianhe-2, es un clúster que combina multiprocesadores Intel Xeon E5-2692v2 junto a Intel Xeon Phi 31S1P, con un total de 3,120,000 de cores y un rendimiento Rmax de 33,862.7 TFlop/s. En el último peldaño del pódium está el suizo Piz Daint - Cray XC50, que utiliza un procesador Xeon E5-2690v3 12C 2.6GHz junto con NVIDIA Tesla P100, obteniendo un rendimiento máximo Rmax de 19,590 TFlop/s, curiosamente sacando más rendimiento con menos cores en comparación con el cuarto, el estadounidense Titan con Cray XK7 Opteron 6274 16C 2.2GHz y GPUs NVIDIA K20x, que, pese a tener 560,640 cores, se queda por debajo con 17,590 TFlop/s.

Pero, toda esta potencia de cómputo necesita de un software capaz de distribuir correctamente todo el trabajo y que la haga funcionar a pleno rendimiento. Es aquí donde la programación paralela se hace vital, convirtiéndose en la herramienta necesaria para

poder sacar el máximo potencial de estos equipos.

La programación paralela ha sido ampliamente utilizada en la aceleración de algoritmos numéricos, especialmente en algoritmos matriciales, ya que en ellos se dan una serie de circunstancias que los hace propicios para su paralelización:

1. La resolución de numerosos problemas relacionados con la ciencia y la ingeniería utilizan estructuras de datos matriciales que se acoplan muy bien al procesamiento paralelo por la facilidad con la que se puede operar sobre una matriz [1].

2. Se puede trabajar con grandes volúmenes de datos pero con necesidad de respuesta inmediata, pudiendo ser incluso en determinados escenarios un requerimiento obtener una respuesta en tiempo real.

3. El uso extendido de matrices en la resolución de problemas de distinta índole, desembocó en la aparición de rutinas y librerías numéricas que optimizan y aumentan las prestaciones de operar con ellas: **speedup**, **eficiencia**, **escalabilidad**, etc. Estas rutinas, con la aparición de la programación paralela, también evolucionaron hasta su paralelización.

Uno de estos conjuntos de rutinas es BLAS (Basic Linear Algebra Subprograms) [11], un núcleo computacional que proporciona una serie de rutinas básicas para realizar operaciones vector-vector en el nivel 1, vector-matriz en el nivel 2 y matriz-matriz en el nivel 3. Estos núcleos computacionales, debido a su eficiencia y portabilidad, están ampliamente disponibles y se utilizan en el desarrollo de software de álgebra lineal de alta calidad, como puede ser MKL [17], librería que se va a utilizar en este trabajo. Junto al desarrollo de librerías para procesadores, ya sean multiprocesadores o multicores, también hay desarrolladas librerías para dispositivos manycores, como la librería cuBLAS [5] para GPUs.

Los supercomputadores y los grandes clústers, son los equipos ideales para desarrollar este tipo de cómputo, también denominado **supercomputación**, aunque para realizar supercomputación no es imprescindible el uso de un supercomputador propiamente dicho. En ocasiones, por cuestiones económicas, de recursos o de otra índole, no es necesario explotar todo el sistema. Es más, en la mayoría de ocasiones en que se utiliza un supercomputador sólo se utiliza una parte de él (uno o varios nodos) que, a su vez, pueden ser compartidos por otros usuarios. Estos nodos, suelen estar formados por procesadores multicore y uno o varios dispositivos manycore, tanto GPUs como MIC, dando como resultado el carácter heterogéneo comentado con anterioridad y siendo necesario el uso de librerías basadas en BLAS que estén optimizadas para este tipo de sistemas. Además, en numerosos centros de cálculo de universidades u organismos tanto públicos como privados, aunque no dispongan de un supercomputador sí que disponen de equipos y clústers, no tan potentes pero sí lo suficiente como para realizar tareas de supercomputación y que, por tanto, tienen el mismo tipo de necesidades al tratarse también de entornos heterogéneos.

Es precisamente en este punto donde este trabajo se inserta, para analizar y estudiar el comportamiento de librerías BLAS en sistemas heterogéneos desarrollando técnicas de auto-optimización experimentales con búsqueda guiada en sistemas **Multicore+multiGPU**. La versión MKL de la librería BLAS será la utilizada para la computación en la CPU multicore, mientras para GPU será cuBLAS. Se estudiará el comportamiento de los dispositivos, operando todos ellos sobre la misma función (en nuestro caso usamos como caso de prueba una multiplicación matricial) repetidamente, hasta encontrar el comportamiento más óptimo; es decir, encontrar la configuración de los datos de entrada (distribución de datos entre los distintos elementos de cómputo del sistema) con la que se obtenga el menor tiempo de ejecución. Una vez llevada a cabo esa experimentación, se trata de crear una rutina que, con la información resultante, sea capaz, a posteriori, de utilizar dicha información cada vez que se vaya a operar con la multiplicación de matrices y en función de la entrada de datos operar con configuraciones óptimas similares a las encontradas. Estas mejoras en el rendimiento de la operación matricial de la multiplicación no solo repercute sobre ella, sino directamente en la mejora del rendimiento de todas aquellas operaciones de nivel superior que la requieren, por ejemplo, la factorización LU, QR o Cholesky [14].

En el estudio práctico se hará uso de metaheurísticas para la realización de una búsqueda guiada a la hora de buscar las configuraciones más óptimas. Es decir, el comportamiento en un determinado paso anterior puede ser de utilidad en ese instante, pero también en el siguiente. Este tipo de procesos son útiles a la hora de hallar configuraciones óptimas o cercanas a las óptimas, siendo igual de válidos tanto para sistemas homogéneos como heterogéneos, así como independientemente del número de dispositivos implicados en la realización del trabajo.

1.2. Estado del Arte

La solución de forma eficiente de numerosos problemas científicos y de ingeniería pasa por el uso de librerías BLAS. Estas están construidas para actuar en dispositivos concretos del sistema o subsistema, como el procesador o un dispositivo manycore. En concreto, la operación de multiplicación matriz-matriz, que pertenece al nivel 3 de BLAS, es una de las operaciones a las que más esfuerzos se han dedicado para su optimización, especialmente cuando los sistemas crecen y se vuelven más complejos al aumentar en número de elementos computacionales y su heterogeneidad. Es aquí donde además de librerías algebraicas es necesario el uso de técnicas de auto-optimización [2, 4, 8, 15], que intentan adaptar las distintas librerías BLAS de cada dispositivo para la explotación eficiente de todos los componentes del sistema que están implicados en una misma operación, buscando así la configuración con la que más rendimiento podemos obtener.

El grupo de Computación Científica y Programación Paralela (CCPP) de la Universidad de Murcia [18] cuenta con un amplio bagaje en el desarrollo, optimización y auto-optimización de algoritmos paralelos y su aplicación dentro de la computación paralela en el ámbito científico. Con el aumento de la complejidad de los sistemas, la distribución

del trabajo entre todos sus componentes es una tarea complicada a la hora de obtener el resultado más optimo. Aunque cada componente de cómputo del sistema haga uso de una librería basada en BLAS optimizada para él, no se conoce de antemano cuál sería el mejor balanceo de trabajo a la hora de realizar la tarea y conseguir un resultado óptimo global para todo el sistema o subsistema que se esté utilizando en un mismo problema [8, 10].

En la industria, los fabricantes de tarjetas gráficas vieron cómo sus dispositivos, creados para una finalidad concreta, se convertían en coprocesadores de cómputo paralelo y, por tanto, en dispositivos programables. Para facilitar esta programación el fabricante NVIDIA contribuyó con su API CUDA [7], que facilita la programación de sus dispositivos gráficos. Además, crearon las correspondientes librerías basadas en BLAS, cuBLAS [5] para configuraciones tanto GPU como multiGPU. A través de la interfaz cuBLAS-XT [6], enrutan dinámicamente llamadas a BLAS a una o varias GPUs NVIDIA, así como a CPUs.

1.3. Plan de Trabajo

La supercomputación se ha convertido en una herramienta indispensable en distintos ámbitos de la ciencia y de la ingeniería. Por esta razón, resulta de gran importancia para estas comunidades que se les proporcione un software sencillo de instalar y fácil de utilizar para, así, poder sacarle el máximo provecho a los recursos hardware que tengan a su disposición. Este TFG pretende abordar este reto, marcándose para ello los siguientes objetivos que se irán desglosando a lo largo del documento:

- Crear una operación de multiplicación de matrices que sea capaz de utilizar un sistema compuesto por una CPU multicore y una o más GPUs, homogéneas o heterogéneas, que hagan uso de librerías tipo BLAS. Para la CPU se utilizará la librería MKL y para GPU la librería cuBLAS.

- Crear una rutina de instalación para la operación de multiplicación de matrices, anteriormente creada. El objetivo de esta rutina es encontrar el reparto óptimo entre los dispositivos utilizados realizando la operación de multiplicación en el menor tiempo y, consecuentemente, con el mayor rendimiento. Para ello se realizará una **búsqueda exhaustiva** entre todas las posibilidades de reparto. Además, antes de hacer uso de la rutina, analizar su comportamiento y calcular el tiempo de ejecución en función del tamaño de entrada de las matrices y del número de dispositivos implicados para analizar las prestaciones de la misma.

- Crear una rutina de instalación mediante **búsqueda guiada** haciendo uso de técnicas heurísticas que encuentren la configuración óptima del sistema según los distintos tamaños de las matrices con las que operar. Esta rutina será una evolución de la anterior, permitiendo encontrar configuraciones óptimas o cercanas a ellas con tiempos de instalación aceptables. El proceso será realizar una primera búsqueda exhaustiva para el menor tamaño, y con esta información continuar para los siguientes tamaños con búsqueda guiada.

- Analizar el comportamiento de la rutina, su tiempo de instalación, y comparar con distintas configuraciones para probar la bondad y validez de esta.
- Una vez realizada la instalación y obtenidos los valores óptimos para los tamaños de experimentación, se creará una librería que contendrá una nueva función de multiplicación de matrices que utilizará la información obtenida. De esta forma, un usuario que utilice esta función de multiplicación de matrices, al indicar el tamaño de entrada, automáticamente utilizará una configuración proporcional a la óptima más cercana.

Para alcanzar estos objetivos se propone seguir la siguiente metodología:

1. En un primer paso, implementar una rutina de multiplicación de matrices pensando en un esquema algorítmico de reparto del trabajo por bloques. Esta rutina se implantará sobre un sistema heterogéneo compuesto por CPU Multicore más dispositivos manycore, concretamente GPUs NVIDIA. Dados estos dispositivos CPU y GPU, se deberá repartir entre ellos todo el volumen de trabajo. Respecto a la implementación multicore se utilizará el paradigma de programación paralela en memoria compartida, concretamente OpenMP [3], además de las librerías, basadas en BLAS, MKL. Para la implementación GPU se utilizará la API CUDA [7] y la correspondiente librería BLAS, denominada cuBLAS. Todo ello sobre lenguaje C/C++.
2. Un segundo paso será crear una rutina que realice una búsqueda exhaustiva, obteniendo las configuraciones óptimas de carga de trabajo para la operación de multiplicación de matrices según un conjunto de tamaños dado como entrada, denominado **conjunto de instalación**. Este conjunto es una secuencia de números enteros y ordenados de menor a mayor que determinan el tamaño de las matrices; además, la diferencia entre un valor y su siguiente o anterior será siempre la misma. Para poder descubrir todas las configuraciones posibles se utilizará un esquema algorítmico por backtracking que realizará la búsqueda en función del tamaño de entrada y el número de dispositivos a utilizar. Posteriormente, se realizará un estudio del crecimiento del tiempo de ejecución según: número de dispositivos, tamaño de las matrices y tamaño del bloque de trabajo. Este tamaño del bloque de trabajo es el volumen de trabajo que se desplaza de un dispositivo a otro buscando el ajuste óptimo. Además, se comprobará su comportamiento con configuraciones pequeñas, con las que nos referimos a un número de dispositivos implicados reducido y a matrices que no sean de gran tamaño. Esta rutina será la denominada de **instalación completa**.
3. En este tercer paso se creará una rutina que hará uso de técnicas de búsqueda heurística con la posibilidad de fijar un umbral de búsqueda. Superado este umbral el algoritmo se detendrá. La forma de proceder será, dado un conjunto de instalación, se tomará el primer tamaño y se realizará una búsqueda exhaustiva obteniendo la configuración óptima. Esta configuración servirá como inicio en la búsqueda para el siguiente tamaño del conjunto de instalación. A partir de este segundo tamaño, las búsquedas dejarán de ser exhaustivas y se realizarán aplicando técnicas heurísticas,

que utilizarán el umbral para detenerse. El valor óptimo encontrado en cada paso servirá de inicio para el siguiente. Esta rutina será la denominada de **instalación guiada**.

4. Analizar y evaluar los experimentos obtenidos con la rutina de instalación guiada para distintos valores umbrales y exponer las principales conclusiones obtenidas.

En los siguientes capítulos, de forma más detallada, se explicará cómo se han realizado las tareas expuestas en cada uno de los puntos de la metodología.

1.4. Entorno de Trabajo

Disponer de un supercomputador en propiedad es algo bastante costoso, por lo que son recursos que se suelen compartir. Un ejemplo es el Barcelona Supercomputing Center - Centro Nacional de Supercomputación, con su **MareNostrum 4** de 148,176 cores y un rendimiento máximo Rmax de 6,227.2 TFlop/s. Este equipo es compartido por numerosos equipos de investigación tanto del ámbito académico como del mundo empresarial, donde todos ellos realizan trabajos de **High-Performance Computing (HPC)**, en español Computación de Alto Rendimiento.

Pero para realizar trabajos de HPC no es necesario tener un supercomputador. Concretamente, el Grupo de Computación Científica y Programación Paralela de la Facultad de Informática de la Universidad de Murcia (UMU) dispone en las instalaciones de ÁTICA[1] de un clúster heterogéneo denominado **Hetereosolar**1, formado por 5 nodos de cómputo más una entrada al cluster desde una máquina virtual. El sistema está interconectado por una red Gigabit Ethernet y distribuido de la siguiente forma:

- **Luna**: Máquina virtual que da acceso al sistema.
- **Mercurio**: 1 CPU AMD Phenom II X6 1075T (6 cores) at 3 GHz, con 16 GB de memoria RAM más 1 GPU Geforce GTX 590, con 1536 MBytes de Memoria Globla y 512 CUDA cores (15 Streaming Multiprocessors, con 32 Streaming Processors por Multiprocessor).
- **Marte**: Sistema idéntico a Mercurio.
- **Venus**: 1 CPU Xeon Haswell E5-2620 V3, a 3.4 GHz, 1 GPU Geforce GT 640, con 1024 MBytes de Memoria Globlal y 384 CUDA cores (2 Streaming Multiprocessors, con 192 Streaming Processors por Multiprocessor) y 2 MIC Intel Xeon Phi 3120a con 57 cores cada uno.
- **Saturno**: 1 CPU hexa-cores (24 cores) Intel Xeon E7530, 1.87 GHz, con 32 GB de memoria RAM, 1 GPU Tesla K20c (Kepler architecture), con 4800 MBytes de Memoria Global, con 2496 CUDA cores (13 Streaming Multiprocessors, con 192 Streaming Processors por Multiprocessor).

[1] Área de Tecnologías de la Información y las Comunicaciones Aplicadas de la Universidad de Murcia

- **Jupiter**: 1 CPU con 12 cores: 2 hexa-cores Intel Xeon E5-2620, a 2.00 GHz, con 32 GB de memoria RAM 6 GPUs, de las cuales hay 2 GPUs NVIDIA Fermi Tesla C2075 con 5375 Mbytes de Memoria Global, con 448 cores (14 Streaming Multiprocessors and 32 Streaming Processors por Multiprocessor) y 4 NVIDIA (agrupadas en dos tarjetas) GPU Geforce GTX 590, con 1536 MBytes de Memoria Globla y 512 CUDA cores (15 Streaming Multiprocessors, con 32 Streaming Processors por Multiprocessor).

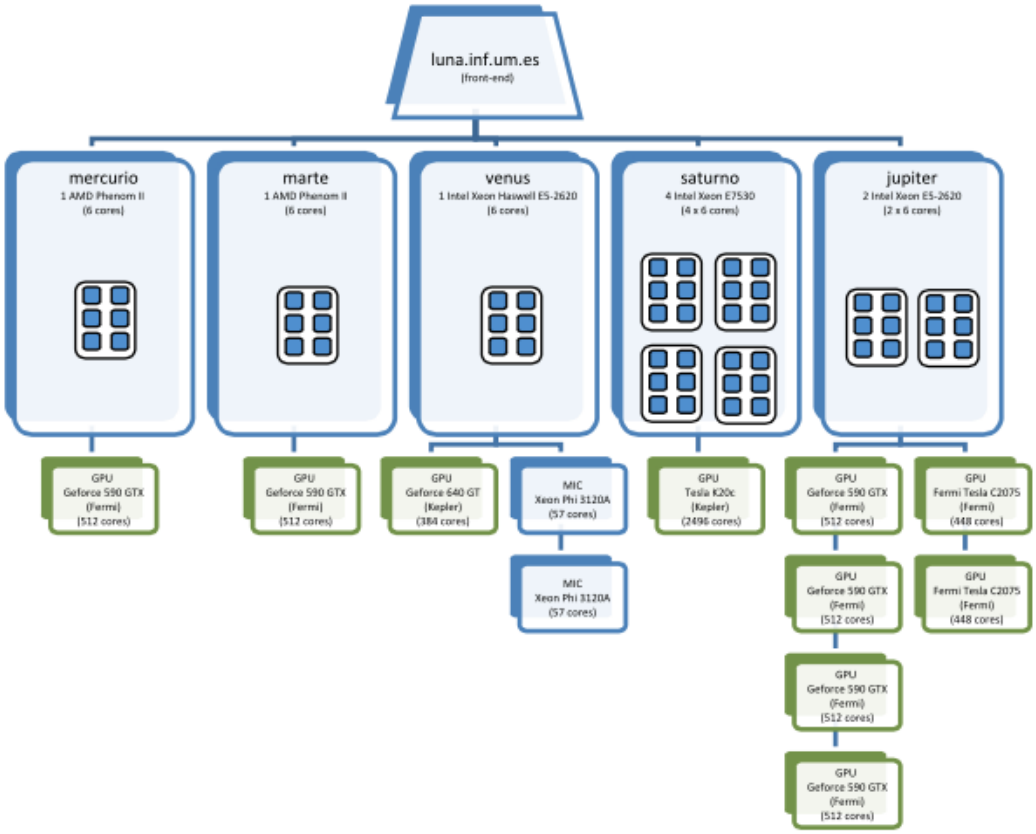


Figura 1: Esquema del clúster Hetereosolar.

El nodo denominado **Júpiter** será uno de los que se utilice en este trabajo, concretamente para los experimentos en un entorno heterogéneo. El compilador utilizado es el Intel C++ Compiler, también conocido como ICC, en su versión 12.0.2, Intel MKL v10.3.2 y la API CUDA 5.0 que incluye cuBlas v2.

Las pruebas en un entorno homogéneo se realizarán en el Sistema de Cálculo Científico del Servicio y Apoya al Investigación Tecnológica (SAIT) de la Universidad Poli-

técnica de Cartagena denominado **Prometeo**. Las características principales del sistema son:

- Un total de 15 nodos con CPU Intel Xeon, 16 cores por nodo y 64 Gbytes de RAM.
- Dos nodos con CPU Intel Xeon, 12 cores por nodo y 32 GBytes de RAM.
- 3 nodos de 16 cores llevan además 2 coprocesadores GPU NVIDIA Tesla K40m.
- 1 nodos de 12 cores lleva además 2 coprocesadores GPU NVIDIA Tesla K20c.
- 1 nodo de 16 cores lleva además 2 coprocesadores Intel Xeon Phi (61 cores).
- 1 nodo de 12 cores lleva además 2 coprocesadores Intel Xeon Phi (57 cores).
- Todo ello concectado con una red de interconexión Infiniband a 56 Gbits/s.

Las características el nodo que se utilizará son: 1 CPU Intel Xeon con 16 cores, 64GB de RAM y 2 GPU NVIDIA Tesla K40m. Las versiones del software utilizado serán el compilador ICC versión 16.0.2.181, Interl MKL versión 11.3.2 y la API CUDA 6.5 que incluye cuBlas v2.

1.5. Estructura del TFG

El presente trabajo pretende hacer ver al lector que en la computación de alto rendimiento, la potencia del hardware por sí sola no es suficiente, y necesita un software que le ayude a sacar el máximo rendimiento posible. Para ello, las librerías BLAS se hacen indispensables, pero, cuando los sistemas se vuelven más complejos tanto por número de elementos de computación como por heterogeneidad, éstas por sí solas no son suficientes. Es aquí donde se verá la importancia del uso de las herramientas de auto-optimización, fáciles de instalar y con tiempos de instalación más que aceptables, que ayudan de forma sustancial a aumentar el rendimiento de la operación de multiplicación matriz-matriz y, por tanto, de aquellas aplicaciones que hagan uso de ella en un nivel superior.

El capítulo 2 mostrará el algoritmo de la multiplicación de matrices partiendo de una versión secuencial, y su evolución hasta una versión paralelizada para CPU+MultiGPU heterogénea usando librerías tipo BLAS. Esta versión de multiplicación de matrices, junto con las configuraciones óptimas conseguidas en la instalación para auto-optimización, será la utilizada en la rutina final.

El capítulo 3 describirá en qué consisten las técnicas de auto-optimización y los dos modos de instalación utilizados. La instalación completa, empleando búsqueda exhaustiva, y la instalación guiada, haciendo uso de técnicas heurísticas que guían la búsqueda.

El capítulo 4 evaluará las distintas instalaciones de la rutina de auto-optimización, completa y guiada. La rutina de auto-optimización completa, por sus características, sólo

se realizará sobre un conjunto reducido de entradas debido a sus altos tiempos de ejecución. Por su parte, la rutina de auto-optimización guiada se realizará con tres valores distintos de *umbral* (2 %, 5 % y 10 %) y tomando como muestra dos conjuntos de tamaños distintos. En una búsqueda de valores óptimos, cuando se utiliza un valor *umbral* se hace para que el sistema siga ejecutándose y buscando nuevos valores aunque estos no mejoren el valor óptimo encontrado hasta ese momento. Siempre y cuando estos nuevos valores no sean peores que el valor óptimo encontrado en una cantidad determinada. Dicha cantidad es el valor *umbral* y al superarse el sistema sí detendría la búsqueda.

Por un lado, el *conjunto instalación*, con el que se procederá a una instalación de la *rutina con auto-optimización* obteniendo una configuración óptima para cada tamaño de matriz considerado y su tiempo de instalación. Una vez hecha la instalación, como usuarios del sistema, utilizaremos un *conjunto validación* que hará uso de la nueva multiplicación de matrices y utilizará las configuraciones óptimas encontradas en el proceso de instalación, tomando tiempos y rendimiento de estas nuevas operaciones. Para validar si la auto-optimización se comportada de forma satisfactoria, se procede a una nueva instalación de la *rutina con auto-optimización* utilizando en esta ocasión el *conjunto validación*. De esta forma, comparando los valores obtenidos en ambas situaciones podemos saber si la *rutina con auto-optimización* es una buena herramienta para mejorar el rendimiento de operaciones BLAS.

En el capítulo 5 se discutirán las conclusiones obtenidas, determinando el alcance del trabajo realizado y estableciendo una serie de líneas de trabajo futuras.

2. Multiplicación de Matrices

La multiplicación de matrices es una de las operaciones más estudiadas y utilizadas dentro de la computación numérica, utilizándose como base para la resolución de infinidad de problemas. En este capítulo, se describirá el algoritmo base de la multiplicación de matrices, así como su evolución en sistemas paralelos, mostrando algunas formas de paralelizar este algoritmo, desde una posible versión secuencial inicial hasta una versión paralela que utilice CPU+multiGPU.

2.1. Multiplicación de Matrices secuencial

Una matriz es una estructura rectangular donde se almacenan elementos que se organizan por filas y columnas. Concretamente y utilizando una notación formal, siendo $\mathbb{R}$ el conjunto de los números reales, se denota el espacio vectorial formado por todos los $m \times n$ como la matriz en $\mathbb{R}^{m \times n}$ [14]:

$$A \in \mathbb{R}^{m \times n} \Longleftrightarrow A = (a_{ij}) = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}, a_{ij} \in \mathbb{R}$$

Y la operación de multiplicación de matrices en el espacio $\mathbb{R}^{m \times p} \times \mathbb{R}^{p \times n} \rightarrow \mathbb{R}^{m \times n}$, se obtiene como:

$$C = AB \Rightarrow c_{i,j} = \sum_{k=1}^{p} a_{ik} b_{k,j}$$

Tal y como se puede observar en la definición de la operación mutiplicación matriz-matriz, para poder realizar la operación el número de columnas de la matriz A ha de ser igual al número de filas de la matriz B, por lo que no es una operación simétrica.

En computación numérica esta notación se amplía incluyendo la multiplicación de un escalar a la operación AB y la agregación del contenido de C multiplicado por otro escalar, quedando finalmente de la siguiente forma:

$$C = \alpha AB + \beta C \Rightarrow c_{ij} = \alpha \sum_{k=1}^{p} a_{ik} b_{k,j} + \beta c_{ij}$$

El algoritmo de la multiplicación de matrices será:

```
1 multiplicacion de matrices A, B -> C
2 {
3  para cada i fila de A
4   para cada j columna de B
5    para cada k elemento
6     C[i,j]=C[i,j]+A[i,k]*B[k,j];
7    finpara
8   finpara
```

```
 finpara
finmultiplicacion
}
```

El tiempo de ejecución de la multiplicación de matrices secuencial, teniendo en cuenta solamente conteo de operaciones con números reales, es decir, simplemente la multiplicación y la suma, sería:

$$mm\_sec(n) \quad = \quad 2n^3 \tag{2.1}$$

Donde *n* es la dimensión de las matrices. Por simplificar, consideramos matrices cuadradas de tamaño $n \times n$.

2.2. Paralelización de la Multiplicación de Matrices en Multicore

Una vez descrito el algoritmo secuencial, se estudiará su paralelización para multicore. Hay que tener en cuenta que la memoria es compartida entre todos los cores de un nodo. Esto supone una ventaja ya que todos los cores implicados en la computación comparten el mismo espacio de direcciones y, por tanto, pueden acceder a los datos sin que exista una latencia significativamente grande al no tener que trasladar los datos entre ellos.

Por tanto, existe una gran ventaja al estar toda la información disponible a todos los cores o threads del nodo, pero, al mismo tiempo, conlleva el inconveniente de tener que controlar por parte del programador el acceso a los datos cuando se va a escribir y evitar inconsistencias. Un thread "descontrolado" puede escribir sobre un dato antes de su lectura por parte de otro thread, cuando lo correcto debería ser primero la lectura y luego la escritura, por poner un ejemplo. Otro ejemplo más relacionado con nuestra operación es que dos threads realicen la operación de multiplicación sobre los mismos vectores fila y columna alterando el resultado final. Recordamos que, en la definición final, el contenido de la matriz C es acumulativo y por tanto este hecho alteraría el valor final.

Paralelizar una operación es repartir el trabajo entre los distintos actores que la realizan. Por tanto, una forma de paralelizar la multiplicación de matrices podría ser que todos los threads implicados en la operación compartan la matriz A, pero, en lugar de compartir toda la matriz B, cada thread disponga de un único bloque de trabajo asignado a él. Estos bloques serían las propias columnas de la matriz B. De esta forma cada columna es asignada a uno y solo un thread.

Una forma sencilla de asignar columnas a los threads para que estos sólo operen con los datos que les corresponden, es aprovechar la correspondencia directa entre el identificador numérico de un thread (*tid*) y el índice de una columna. Es decir, a la columna 0 se le asignará el identificador de thread *tid* 0, la columna 1 al *tid* 1 y así sucesivamente.

Pero sigue existiendo un problema, lo normal es que una matriz tenga miles de columnas, mientras que el número de threads es bastante limitado. Algunos de los procesadores más potentes disponen de 24 cores, lo que equivale a tener 48 threads si tienen habilitado el hyperthreading [16]. Con estos números, un thread realmente opera sobre un conjunto de columnas, y una forma de asignar estas columnas es por medio de aritmética modular, es decir, si $tid = columna \ mod \ numerodethreads$ la columna se le asigna al thread. El esquema de la multiplicación quedaría de esta forma:

```
obtenemos nuestro tid
multiplicacion de matrices A, B -> C
 para cada i fila de A
  para cada j columna de B
   si tid==(j mod num_threads) entonces
    para cada k elemento
     C[i,j]=C[i,j]+A[i,k]*B[k,j];
    finpara
   finsi
  finpara
 finpara
finmultiplicacion
```

Aunque es una forma de repartir el trabajo sencilla, otro tipo de repartos pueden resultar igualmente sencillos y favorecer que se pueda escalar, que es el objetivo final hasta llegar a multiGPU. La forma de repartir las columnas será por bloques de columnas consecutivas. De esta forma los datos sobre los que opera un thread están alineados. Aquí el reparto es directo, tantos bloques como de threads se dispongan, asumiendo que los bloques tendrán el mismo número de columnas. El esquema queda:

```
obtenemos nuestro tid
bloque = numero_columnas_B / num_threads
multiplicacion de matrices A, B -> C
 para cada i fila de A
  para cada j columna de B
   si (j >= tid*bloque) y (j < (tid+1)*bloque) entonces
    para cada k elemento
     C[i,j]=C[i,j]+A[i,k]*B[k,j];
    finpara
   finsi
  finpara
 finpara
finmultiplicacion
```

El objetivo ideal de la paralelización es reducir el tiempo de ejecución en el mismo factor que threads haya ejecutándose. Si despreciamos los tiempos de sincronización y puesta en marcha de los threads, el tiempo de ejecución ideal para la multiplicación de

matrices paralela en multicore con memoria compartida es:

$$mm\_paralela(n,p) = \frac{2n^3}{p} = \frac{mm\_sec(n)}{p} \tag{2.2}$$

siendo ahora el tiempo de ejecución en función de la dimensión de las matrices *n* y del número de threads implicados *p*, $t(n,p)$.

2.3. Paralelización de la Multiplicación de Matrices en multicore+-GPU

El esquema algorítmico de la Multiplicación de matrices paralelizada con GPU es similar al de multicore. Simplemente habría que añadir y controlar la carga de trabajo para la GPU. Aunque aquí, a la hora de calcular el tiempo de ejecución, sí que hay que tener en cuenta las comunicaciones, ya que la GPU no puede acceder directamente a los datos en memoria principal, hay que llevarlos al dispositivo. Estos costes, aunque no se estudien con detenimiento, sí que se van a contemplar como parte del tiempo invertido por la GPU en la operación de multiplicación.

Si la operación básica de la multiplicación de matrices se define como $C = \alpha AB + \beta C$, en este caso tendremos $C = \alpha A(B_1|B_2) + \beta(C_1|C_2)$, quedando finalmente como:

$$C = (\alpha AB_{gpu} + \beta C_{gpu}|\alpha AB_{cpu} + \beta C_{cpu}|) \tag{2.3}$$

donde el sub-índice 1 puede corresponder al trabajo de la GPU y el sub-índice 2 al trabajo de la CPU. Además, el trabajo asignado a la CPU se realizaría también paralelizándose siguiendo el esquema explicado en la sección anterior.

El algoritmo para la multiplicación de matrices CPU+GPU quedaría de la siguiente forma:

```
1  obtenemos nuestro tid
2  // El proceso padre, tid=0, es el encargado de enviar y recibir
      la informacion de la GPU
3  if (tid==0)
4   enviar A, B1, C1
5   mm_GPU(A,B1,C1)
6   recibir C1
7  finsi
8  bloque = numero_columnas_B2 / num_threads
9  multiplicacion de matrices A, B2 -> C2
10  para cada i fila de A
11   para cada j columna de B1
12    si (j >= tid*bloque) y (j < (tid+1)*bloque) entonces
13     para cada k elemento
14      C2[i,j]=C2[i,j]+A[i,k]*B2[k,j];
15     finpara
```

```
16     finsi
17    finpara
18   finpara
19 finmultiplicacion
```

Para simplificar el cálculo del tiempo de ejecución, el tiempo correspondiente a la GPU lo denominaremos *mm_gpu(n)*, siendo *n* el tamaño del bloque de columnas sobre el que se opera, e incluyendo el envío de datos, la operación de multiplicación y la vuelta del resultado. La parte asignada a la CPU también se paralelizaría. Por tanto, el tiempo de ejecución final quedaría como:

$$t(n, p, GPU) = max\{\frac{mm\_sec(|B_2|)}{p}, mm\_gpu(|B_1|)\} \tag{2.4}$$

Donde $|B_1|$ y $|B_2|$ son la cantidad de columnas de *B* asignadas tanto a la CPU como a la GPU, y donde $|B_1| + |B_2| = |B|$.

Es en este punto donde se empieza a ver la necesidad de auto-optimización en las operaciones de álgebra lineal. Se ha comentado que se divide la matriz B en dos bloques; con uno de ellos opera la CPU volviendo a repartir en partes iguales la correspondiente submatriz de B entre los threads, y el otro bloque se envía a GPU, pero, ¿qué tamaños tienen los bloques? Sabemos que $|B_1| + |B_2| = |B|$, pero desconocemos el tamaño de $|B_1|$ y $|B_2|$. Si este tamaño fuera arbitrario, por ejemplo, la mitad, ¿sería un reparto que minimizaría, y por tanto optimizaría, la función $t(n, p, GPU)$?

Desde este trabajo se pretende que esta decisión, que en este momento correría a cargo del programador, la tome una rutina de auto-optimización, asignando de forma óptima las cargas de trabajo, y realizando la asignación de forma transparente al programador a la hora de hacer uso de la operación de multiplicación de matrices.

2.4. Paralelización de la Multiplicación de Matrices en multicore+multiGPU

Aunque la evolución del algoritmo y los problemas en el reparto del trabajo que supone la paralelización de la multiplicación de matrices con multicore+multiGPU se puede intuir, vamos a ver el esquema algorítmico del problema.

Viendo el funcionamiento del anterior algoritmo con una GPU, se intuye que se deberá trocear la matriz B entre tantos bloques como CPUs y GPUs haya. Además, habrá que llevar la matriz A y el correspondiente bloque de B a todas y cada una de las GPUs que se utilicen. Si implicamos un thread en esta tarea, aunque la operación de multiplicación en GPU sea potencialmente más rápida que en CPU, al final se tendría un comportamiento secuencial por GPU:

```
1 ...
```

```
Para i: 1..Numero de GPUs
 enviar A, B_gpu_i, C_gpu_i a la  gpu_i
 lanzar mm_GPU(A,B_gpu_i, C_gpu_i)
 recibir C_gpu_i
finPara
...
```

Por tanto, el primer paso es asignar a cada GPU un thread que se encargue de las tres operaciones: enviar datos, lanzar proceso en GPU y recibir datos. Seguidamente, los threads libres, realizarán la operación de multiplicación paralela en multicore con su bloque asignado. Tenemos así el esquema:

```
obtenemos nuestro tid
obtener gpu_id // El identificador de cada GPU
if (gpu_id==tid)
 enviar A, B_gpu_id, C_gpu_tid a la  gpu_id
 lanzar mm_GPU(A,B_gpu_id, C_gpu_id)
 recibir C_gpu_id
finsi
bloque = numero_columnas_Bn / num_threads
multiplicacion de matrices A, Bn -> Cn
 para cada i fila de A
  para cada j columna de B1
   si (j >= tid*bloque) y (j < (tid+1)*bloque) entonces
    para cada k elemento
     C1[i,j]=C1[i,j]+A[i,k]*B1[k,j];
    finpara
   finsi
  finpara
 finpara
fin multiplicacion
```

En realidad, el algoritmo final difiere del mostrado al utilizar librerías BLAS provistas de la operación de multiplicación de matrices paralelizada, tanto para CPU multicore como GPU, quedando de la siguiente forma:

```
obtenemos nuestro tid
obtener gpu_id
if (tid==0)
 // Siendo p el numero de threads disponibles
 mm_BLAS_paralela_cpu(A, B1, C1, p)
finsi
if (tid>0)
 enviar A, B_tid+1, C_tid+1 a la gpu_id=tid-1
 lanzar mm_BLAS_paralela_gpu(GPU_ID,A,B_tid+1, C_tid+1)
 recibir C_tid+1
finsi
```

Donde el bloque B_1 se asignará a la CPU y los bloques B_2 a B_n se asignarán a las GPUs con id 1 hasta $n-1$ respectivamente, y habiendo un total de $n-1$ GPUs.

La duda que se plantea aquí es la misma que en el punto anterior, aunque aumentando su dificultad al aumentar también el número de elementos de cómputo. Si tenemos varias GPUs iguales (homogéneas), aunque no se tenga claro cuánto trabajo mandar a las GPUs y cuánto a la CPU, el programador puede pensar que una buena opción es repartir a partes iguales el trabajo de las GPUs entre cada una de ellas, pero, ¿está seguro de que esa es la mejor opción? Con los costes de enviar los datos y su posterior recepción, tal vez sea preferible no utilizar todas las GPUs. En el caso de tener un sistema heterogéneo, con varias GPUs distintas, el programador podría pensar en enviar la información sólo a las más rápidas, pero de esta forma tal vez esté desaprovechando tener las otras GPUs paradas, aunque sean más lentas, y si las hubiera puesto a funcionar se realizaría antes el trabajo.

Al final, es la misma cuestión abordada en el punto anterior. Toda la decisión queda en manos del programador que podrá, por su experiencia, intuir comportamientos aceptables sin saber si realmente lo son. La propuesta que desde este trabajo se hace, como se ha dicho, es el uso de una rutina de auto-optimización, y que sea esta, con ayuda de búsqueda exhaustiva o heurística, la que vaya tomando las decisiones. En el siguiente capítulo se explicará en qué consisten estas técnicas de auto-optimización y cómo funcionan, tanto en entornos homogéneos como heterogéneos.

3. Técnicas de auto-optimización en la Multiplicación de Matrices

En este capítulo se explicará en qué consisten las técnicas de auto-optimización utilizando la multiplicación de matrices, operación de nivel 3 de BLAS. Tal y como se ha indicado a lo largo del documento, la multiplicación de matrices es una operación que tiene un alto coste computacional con un orden $O(n^3)$, pero muy usada al ser la base de numerosas funciones numéricas de nivel superior como son las factorizaciones LU, QR o Cholesky, que hacen un uso recurrente de ella en sus cálculos. Por ello, cada vez que hay un avance hardware o aparece un nuevo dispositivo de cómputo, rápidamente, aparecen implementaciones de la multiplicación de matrices para estos nuevos sistemas, evolucionando así a la par que ellos.

La operación de multiplicación de matrices, $C = \alpha AB + \beta C$, tal y como se definió en el capítulo 2, cuando se quiere acelerar por medio de la paralelización se puede organizar en la forma $C = \alpha A(B_1|B_2) + \beta(C_1|C_2)$, y presenta, principalmente, una dificultad a la hora de la elección de la carga de trabajo para cada elemento de cómputo. En la paralelización en CPU, el problema de la asignación de trabajo pasa a ser trivial, asignando a todos los threads la misma cantidad de trabajo. Si seguimos a CPU+GPU, el problema deja de ser trivial a la hora de encontrar un reparto óptimo. Dependiendo del tamaño de las matrices a operar un reparto óptimo podría ser enviar todo el trabajo a GPU, aunque, tal vez, lo mejor sea hacer lo contrario y que realice todo el trabajo solo la CPU. La cuestión es que, a priori, no se es capaz de realizar un reparto óptimo de la carga de trabajo. Si este problema se extiende al caso general de CPU+multiGPU, la dificultad aumenta de tal forma que encontrar un reparto óptimo, o cercano, se hace bastante difícil, aun siendo un programador experto.

Las técnicas de auto-optimización ayudan a resolver este problema, aplicándose a una rutina con el objetivo de dotarla de cierta capacidad de ajustarse automáticamente y por sí misma a las condiciones de ejecución, de cara a maximizar sus prestaciones. En el caso de la rutina de multiplicación matricial, anteriormente explicada, se buscará que esta tenga capacidad para decidir, dado un problema concreto, cuál sería el reparto óptimo de trabajo entre la CPU y las $n - 1$ GPUs de que disponga el sistema. Con este objetivo, en la fase de instalación de la rutina se realizará la búsqueda del mejor reparto de trabajo para cada tamaño de entre un conjunto de tamaños de problema o $Conjunto\_de\_instalacion$. Posteriormente, cuando se pretenda resolver un problema concreto de tamaño n_E, se utilizará la información obtenida en la instalación para decidir qué reparto de trabajo se aplicará a este problema, por ejemplo, aplicando para el tamaño n_E el mismo reparto que se aplicó al valor más cercano n_I del $Conjunto\_de\_instalacion$.

Al final del capítulo, de forma breve, se explicará cómo utilizar la rutina de auto-optimización.

3.1. Búsqueda exhaustiva

Una forma de resolver el problema de la carga de trabajo en la multiplicación matricial CPU+multiGPU es realizar en la rutina de auto-optimización una *búsqueda completa* de la mejor solución. Esto es, analizar todos los repartos de trabajo posibles entre la CPU y las GPUs. Considerando el ejemplo sencillo de una multiplicación matricial sobre una CPU y una GPU, la operación consistiría en medir el tiempo de ejecución de la multiplicación de matrices con toda la carga de trabajo en la CPU y nada en la GPU, volver a medir el tiempo de ejecución poniendo en este caso toda la carga de trabajo menos el de una columna en la CPU y el trabajo de una columna en la GPU, seguidamente, ir repitiendo el proceso midiendo los tiempos de ejecución y pasando columna a columna el trabajo de la CPU a la GPU hasta que sea la GPU quien realice todo el trabajo. Llegados a este punto la combinación que menor tiempo de ejecución tenga es la más óptima. Realizar este proceso columna a columna es ralentizarlo innecesariamente; lo normal es tomar un tamaño arbitrario, lo suficientemente grande para que produzca un cambio en el rendimiento pero lo suficientemente pequeño para que permita trazar esta evolución sin que haya saltos significativos entre un estado y el siguiente, evitando de esta forma que se puedan quedar sin explorar estados que potencialmente puedan ser óptimos. Este valor lo denotamos como tb.

Por tanto, volviendo al ejemplo sencillo CPU+GPU, si el tamaño de nuestro problema es m y la información la disponemos en un vector en la forma $< m, trabajo\_gpu, trabajo\_cpu >$ donde $trabajo\_gpu + trabajo\_cpu = m$, tendremos:

- Paso 1. $< m, m, 0 >$
- Paso 2. $< m, m - tb, tb >$
- Paso 3. $< m, m - 2 * tb, 2 * tb >$
- Paso i-ésimo. $< m, m - (i - 1) * tb, (i - 1) * tb >$
- Último paso. $< m, 0, m >$

Para el caso general con $n - 1$ GPUs y una CPU, $< m, gpu_1, gpu_2, ..., gpu_{n-1}, cpu >$, partiríamos de una configuración $< m, m, 0, ..., 0, 0 >$ hasta llegar a $< m, 0, 0, ..., 0, m >$ pasando por todas las combinaciones posibles.

Los esquemas algorítmicos que se ajustan bien a este tipo de problemas son los que se resuelven por búsquedas en un árbol de soluciones, hallando así la solución más optima. En concreto, la técnica a utilizar será la de **backtracking** [13]. En el backtracking, la búsqueda por el árbol de soluciones lógico se realiza en profundidad y retrocediendo, de ahí su nombre.

El algoritmo para saber cuál es la mejor configuración utiliza un **vector de configuración** similar igual al descrito anteriormente. Su definición formal es:

$< N, gpu_1, gpu_2, ..., gpu_{n-1}, cpu, th >$,
donde:

- N es el tamaño de entrada del problema.
- gpu_i el trabajo asignado a la GPU i-ésima. En total habrá $n-1$ GPUs trabajando.
- cpu el trabajo asignado a la CPU.
- th el número de threads que utilizará la CPU. $th \leq Maximo\_de\_threads\_asignados$.
- $gpu_1 + gpu_2 + ... + gpu_{n-i} + cpu = N$

Con los valores de estos vectores configuración, el algoritmo irá comprobando cada uno de ellos e irá guardando el mejor. La adaptación del esquema general de backtracking al problema objeto de estudio es la siguiente:

```
Instalacion Completa con Busqueda Exhaustiva
{
FOR (i=0, i < numero elementos cj_install)  {
 N=cj_install[i];
 //INICIALIZAR VECTOR CONFIGURACION Y VECTOR SOLUCION
 v_conf <- <n,-tb,-tb,...,-tb,-tb>
 v_conf_s <- <n,-tb,-tb,...,-tb,-tb>
 nivel=1;
 S=0;
 do {
  //GENERAR
  v_conf[nivel]+=tb;
  // SOLUCION
  if (nivel==num_gpus+1){
   //TRATAR
   S = suma (v_conf)
   //COMPROBAMOS QUE LA SUMA ES N
   if (S==N){
    FLOPS_ANT = 0
    FLOPS = mm_mgpu_th(v_conf);
    if (FLOPS > FLOPS_ANT)
     v_conf_s <- v_conf
    }   //FIN DE TRATAR
  }//FIN SOLUCION
  //CRITERIO
   S = suma (v_conf)
   if ((S<N) && (nivel<num_gpus+1)){
    nivel++;
   } else {
    while ((nivel>0) && (v_conf[nivel]>=N)){
```

```
31          //RETROCEDER
32          v_conf[nivel]=(-1)*tb;
33          nivel--;
34         }
35        }
36      } while (nivel);
37     }
38    }
```

Como se trata de una búsqueda completa, el método debe recorrer todo el árbol de soluciones. Si calculamos el número total de soluciones podremos ver la magnitud del problema. La cantidad de soluciones va en función de:

- m tamaño de la matriz.
- tb cantidad de columnas que se intercambian en la búsqueda.
- Para facilitar los cálculos llamaremos k al número de bloques de tamaño tb que tiene una matriz, $k = m/tb$.
- n al número de dispositivos de cómputo. Se dispondrá de $n-1$ GPUs y una CPU.

Dadas k particiones hay que decidir cómo se agrupan los bloques consecutivos para asignar a los n elementos de cómputo. Es posible que a algún elemento de cómputo no se le asigne computación, con lo que la misma zona de corte de la asignación se puede seleccionar dos veces, y la primera zona puede ser la cero y la última ha de ser forzosamente la k. Por tanto se trata de contar el número de combinaciones con repetición de $k+1$ elementos (del 0 al k) tomados de $n-1$ en $n-1$ (el último no se selecciona pues es forzosamente k):

$$busqueda\_completa(m, tb, k, n) = \binom{n+k-1}{n} = \frac{(n+k-1)!}{(n-1)!\quad k!} = \frac{(n+\frac{m}{tb}-1)!}{(n-1)!\quad \frac{m}{tb}!}$$

Obtenemos el número de posibles soluciones para un ejemplo:

- Tamaño de una matriz que por cantidad de computo pueda requerir aceleración por GPU, $m = 9216$.
- Cantidad de columnas que se intercambian entre los dispositivos, $tb = 192$.
- Cantidad de bloques a repartir entre los dispositivos, $k = \frac{9216}{192} = 48$.
- 6 GPUs y una CPU, $n = 7$.

$\binom{7+48-1}{7} = \frac{(7+48-1)!}{6!\quad 48!} = \frac{54!}{6!\quad 48!} \approx 26$ Millones de comprobaciones.

Esta estimación que se ha mostrado como ejemplo, aún deja elementos sin evaluar; principalmente dos:

- En la evaluación de cada solución, el algoritmo, si la CPU tiene carga de trabajo, evalúa la operación con un thread, con dos, con tres, y consecutivamente hasta el máximo de threads disponibles. Realmente, como teóricamente a mayor número de threads menor tiempo de ejecución, para disminuir el factor de este proceso se hace a la inversa, comenzando por el mayor número de threads disponible y disminuyendo en uno. En el instante en que una solución sea peor que la anterior, en esta fase de threads de CPU, el algoritmo dejaría de comprobar las configuraciones menores y continuaría con el proceso general. Este procedimiento, aunque ayuda a encontrar la mejor configuración del sistema, multiplica como mínimo por un factor de 2, hasta un máximo que corresponde al $numero\ de\ threads$ dedicados, en los casos casos en que la CPU dispone de carga de trabajo.

- Como se ha especificado en apartados anteriores, la operación de auto-optimización, en su proceso de instalación, va evaluando diferentes configuraciones para conseguir aquellas que optimizan la operación objetivo. Para ello, se provee al motor de auto-optimización de un conjunto de valores, denominado $conjunto\_instalacion$, que contiene uno valores de tamaño del problema ordenados de menor a mayor. En el caso de la multiplicación matricial contiene el tamaño de las matrices. Por tanto, para cada elemento del $conjunto\_instalacion$ hay que aplicar una búsqueda exhaustiva para encontrar la configuración que optimiza la operación. Un $conjunto\_instalacion = \{768, 1536, 2304, 3072, 3840, 4608, 5376, 6144, 6912, 7680, 8448, 9216, 9984, 10752, 11520\}$ hace que sea prácticamente inviable su instalación en unos tiempos razonables.

El problema de la técnica de auto-optimización utilizando búsqueda exhaustiva, lejos de solucionarse, puede incluso empeorar si es aplicado en sistemas con mayor capacidad de cómputo. Mayor capacidad de cómputo implicaría más elementos y de tamaños mayores en el $conjunto\_instalacion$, al poder abordar tamaños de problema mayores, aumentando de forma significativa el número de combinaciones y comprobaciones. Lo mismo ocurriría incrementando el número de GPUs a evaluar.

En conclusión, esta técnica de auto-optimización, pese a ser capaz de encontrar la configuración óptima, no es operativa. Aplicarla implica tener el sistema dedicado por completo durante todo el proceso de instalación, que, como se ha visto, sería una tarea que no se realizaría en un tiempo razonable.

3.2. Búsqueda guiada

La *búsqueda exhaustiva* es un proceso lento al buscar por todo el espacio de soluciones, pero, precisamente por esto, muy efectivo al encontrar siempre la solución óptima. El proceso de búsqueda guiada consiste en aplicar técnicas heurísticas que nos permitan encontrar, si no la solución óptima, sí una solución cercana a la óptima en tiempos mucho más razonables que los de la búsqueda exhaustiva.

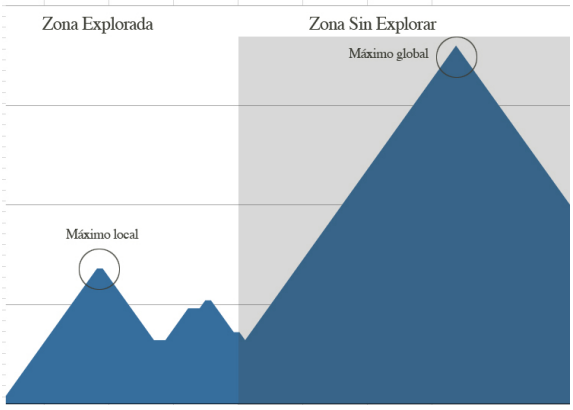


Figura 2: Ejemplo gráfico de la Búsqueda Local por ascensión de colinas.

El proceso es sencillo. Al igual que la búsqueda exhaustiva, la búsqueda guiada dispone de un $conjunto\_instalacion$ como entrada, y se buscará la configuración óptima para cada uno de sus elementos. El proceso comienza con el primer elemento del $conjunto\_instalacion$, donde se aplicará **búsqueda exhaustiva**. Precisamente por ser el elemento de menor tamaño, es el elemento que menos combinaciones y soluciones distintas propone, con lo que es el tamaño del conjunto para el que menos se tarda en buscar la solución óptima. Una vez que para el primer elemento se ha obtenido y registrado la configuración óptima, esta información se utiliza para generar una posible solución para el segundo elemento del $conjunto\_instalacion$. Se realiza una búsqueda local en torno a una combinación obtenida a partir de la óptima para el tamaño anterior. Se evalúan todas las configuraciones, la generada con la información del paso anterior y las encontradas en el proceso de búsqueda local en su entorno, seleccionando la mejor solución y convirtiéndose esta en la solución óptima para este elemento, además de ser la información que se utilizará en el siguiente paso, repitiendo así el proceso.

Podemos considerar una implementación con una metaheurística de búsqueda local por ascensión de colinas y posiciones tabú. De esta forma, reducimos el número de comprobaciones que realiza la búsqueda exhaustiva y otras técnicas metaheurísticas basadas en poblaciones que también requieren un alto número de comprobaciones. Un problema que puede encontrarse en los procesos de búsqueda local es tomar como valores óptimos valores que realmente son óptimos locales, finalizando la búsqueda y no encontrando valores que realmente puedan ser óptimos al problema por el hecho de estar en otra parte del espacio de búsqueda y, por tanto, no evaluándose como posible opción (figura 2). Para intentar mitigar este problema se plantean dos opciones, por un lado, seguir buscando aunque las configuraciones que se evalúen sean peores, y, por otro, cuando se toma la información del vector óptimo del paso anterior, generar a partir de él de distintas formas el nuevo vector desde el que se lanza la búsqueda.

Respecto a la primera opción, seguir buscando configuraciones aunque los resultados empeoren a los evaluados, es una decisión que no se puede mantener por tiempo indefinido. La solución es seguir la búsqueda mientras los valores que se evalúen, aunque sean

peores que los encontrados hasta ese momento, se mantengan por debajo de un $umbral$ cuyo valor se determina al inicio de la instalación. Es decir, si la configuración a evaluar es peor que el mejor valor hasta ese momento más el valor del umbral, la búsqueda continúa; si, por el contrario, es mayor, la búsqueda se detendría. Los valores $umbral$ utilizados en este estudio han sido del 2 %, 5 % y 10 %.

Respecto a la segunda opción, cuando llega la información del vector óptimo para el elemento $i-1$ del $conjunto\_instalacion$ y el proceso de búsqueda se relanza para el elemento i, la información contenida en el vector hay que adaptarla al siguiente tamaño. Una manera de intentar paliar el problema de los máximos locales es probar de distintas maneras de crear este vector de inicio en lugar de hacerlo siempre de la misma forma. Generar el nuevo vector de modo $proporcional$ al óptimo anterior es una buena forma de crear el nuevo vector, pero, hacerlo siempre de esta forma implica que las búsquedas serán en torno a un trozo del espacio de soluciones no explorando el resto. Por tanto, se plantea generar este vector de distintas formas, evaluarlas, y relanzar la búsqueda con la mejor configuración. Los modos que aquí se evaluarán serán: proporcional, sobrecargar la unidad de cómputo que más carga tenga asignada al estimar que es la más rápida, distribuir entre todas las GPUs y la CPU la misma cantidad nueva de trabajo, o la media de las 3 opciones anteriores para cada unidad de cómputo.

Una vez que se reinicia la búsqueda, el algoritmo irá evaluando a los **vecinos** del estado actual para así saber a cuál saltar. Hay que tener en cuente que el uso de posiciones tabú significa que no volverán a ser evaluados elementos ya estudiados previamente. El concepto $vecino$ se define como el vector que está a tb elementos de distancia del actual. Un ejemplo, recordando la definición del vector configuración $< m, gpu_1, gpu_2, ..., gpu_{n-1}, cpu >$, si en un instante determinado el vector actual tiene la configuración $< m, h, j, k, l >$ donde $h+j+k+l=m$, $\{h, j, k\}$ son las cargas de trabajo para cada GPU y l es la carga de trabajo para la CPU, $< m, h-tb, j+tb, k, l >$ y $< m, h, j, k+tb, l-tb >$ son dos vecinos de $< m, h, j, k, l >$.

El algoritmo de auto-optimización con búsqueda guiada queda de la siguiente forma:

```
1  Instalacion Guidada
2  {
3   v_conf;
4   v_best;
5   v_conf_vecino;
6   FLOPS, BEST_FLOPS, FLOPS_VECINO =0;
7   N=cj_install[1];
8
9   //BUSQUEDA COMPLETA PARA EL PRIMER ELEMENTO DEL CONJUNTO
       INSTALACION
10  v_best <- instalacionCompleta(N);
11  v_conf<-v_best;
12
```

```
13 //BUSQUEDA GUIADA PARA EL RESTO PARTIENDO DE UN VECTOR
      CONFIGURACION
14  for (i=1, i < numero elementos cj_install)  {
15  N=cj_install[i];
16  visitados=0
17
18  v_conf = iniciarBusqueda(v_conf_ant);
19  add_Visitado(visitados, v_conf);
20  BEST_FLOPS=mm_mgpu_th(v_conf);
21  v_best <- v_conf;
22
23  stop=0;
24  do {
25   FLOPS_VECINO=mejorVecino(v_conf, v_conf_vecino, visitados);
26   if (FLOPS_VECINO>BEST_FLOPS){
27    v_best <- v_conf_vecino
28   }
29
30   if (FLOPS_VECINO - BEST_FLOPS < umbral){
31    stop=1;
32   } else {
33    v_conf <- v_conf_vecino;
34   }
35  } while (!stop);
36  v_conf_ant <- v_best;
37  }
38 }
```

El funcionamiento, como se puede observar, es bastante sencillo, se realiza una *búsqueda completa* para el primer elemento del *conjunto de instalación*, obteniendo así el mejor vector configuración para este tamaño, que a su vez, servirá para construir el vector de inicio de la segunda entrada del *conjunto instalación*, aplicando ya sobre él la *búsqueda guiada*. Lo que primero hace el algoritmo dentro del bloque de la búsqueda guiada es, junto al vector de la mejor configuración del paso anterior, *v_conf_ant*, crear un buen vector inicio, v_conf, lo que se realiza en *iniciarBusqueda*.

Entre una entrada del *conjunto instalación* y la siguiente, existe una diferencia. Es justo esta diferencia la que debe repartirse entre los elementos de cómputo. En *iniciarBusqueda* se comprueba cuál sería la mejor forma de repartir esta diferencia, ya sea: de forma proporcional, sobrecargando la unidad de cómputo que más carga tenga asignada, distribuir entre todas las GPUs y CPU la misma cantidad de trabajo, o realizar una media aritmética de las tres opciones anteriores. Evaluando cada una de estas configuraciones con la operación de la multiplicación matricial, se obtiene el vector de inicio. Como el algoritmo incluye que la búsqueda se realice con *elementos tabú*, este nuevo vector v_conf se añade a *visitados* y se convierte en el vector con la mejor configuración hasta el momento.

A continuación comienza una búsqueda a través del espacio de soluciones en torno a v_conf, procediendo de la siguiente forma: evaluando a los vecinos del vector configuración actual, v_conf, y tomando de ellos el mejor v_conf_vecino. Esta tarea se realiza dentro de la función $mejorVecino(v\_conf, v\_conf\_vecino, visitados)$. Realmente hay dos formas de evaluar al mejor vecino. Por un lado, tomar dos valores aleatorios comprendidos entre $1, \ldots, n$, es decir, de forma aleatoria seleccionar entre las GPUs y la CPU dos elementos de cómputo; al primero se le restará tb y al segundo se le sumará tb. Si el vector es válido (han de sumar N y ningún valor será negativo) y no ha sido visitado, se toma como candidato a mejor vecino. La cantidad de población a evaluar vendrá determinada por el número de elementos de cómputo que tenga el sistema, es decir, n. Otra forma de elegir el mejor vecino es evaluar todas los posibles vecinos, pero este número puede ser muy algo si el número de elementos de cómputo lo es, y puede ser preferible evaluar un número preestablecido de vecinos.

El análisis de la vecindad con la búsqueda aleatoria analizando en cada caso n vecinos tiene un orden $O(n)$ al tomar tantos como elementos de cómputo. Por otro lado, analizar todos los posibles vecinos es de orden $O(n^2)$ al combinar $n(n-1)$ elementos. La ventaja de analizar todos los posibles vecinos es que obtendremos el mejor vecino, con a un coste que puede ser asumible si el número de elementos de cómputo no es excesivo, lo que suele ocurrir en la práctica.

Una vez que se han evaluado los vecinos y se obtiene el mejor vecino, v_conf_vecino, si este mejora a la mejor configuración hallada hasta el momento, v_best, se convierte en el mejor vector configuración. Por contra, si no mejora, pueden ocurrir dos situaciones. Si no mejora pero está por debajo del *umbral* establecido al inicio de la búsqueda, el algoritmo continúa por el vector v_conf_vecino convirtiéndose así en el vector actual v_conf y continuando por él el proceso de búsqueda. En caso contrario, si se sobrepasa el $umbral$ fijado, la búsqueda finalizaría para esta entrada del *conjunto instalación*, el vector *v_best* sería el vector con la mejor configuración para este tamaño convirtiéndose, además, en la configuración anterior v_conf_ant, para el siguiente paso si lo hubiera, es decir, para continuar por el siguiente elemento del *conjunto instalación*.

Con este algoritmo de búsqueda guiada que hace uso de metaheurística en la toma de decisiones, se pretende conseguir configuraciones óptimas, o muy cercanas a ellas, reduciendo significativamente el tiempo del proceso de instalación. Este vendrá determinado principalmente por el valor $umbral$. A mayor $umbral$ más se expande la búsqueda y más tiempo tarda la búsqueda, pero se conseguirán mejores valores. Por contra, a menor $umbral$, se reduce el espacio de búsqueda y el tiempo del proceso de instalación, pero los valores obtenidos pueden ser de menor calidad que los de mayor $umbral$.

3.3. Utilización de la rutina de auto-optimización

El administrador (o *manager*) del sistema realizará el proceso de instalación de la rutina de auto-optimización, dando por hecho que el equipo ya está operativo y funcionando con todas las APIs, librerías y tecnologías necesarias. En el proceso de instalación se determina, según el uso del equipo, el tamaño menor del conjunto instalación, el incremento entre tamaños, el tamaño del elemento mayor del conjunto instalación y el valor tb. Estos valores vendrán determinados según el tipo de uso que se haga del equipo.

La rutina, además, mostrará un listado de las GPUs instaladas y ofrecerá la opción de determinar cuántas y cuáles serán elegidas para el proceso de instalación. Después de esta elección, solicitará el tipo de instalación, *búsqueda exhaustiva* o *búsqueda guiada*, y en el caso de la búsqueda guiada el valor $umbral$ que se utilizará. Una vez introducidos todos estos valores comenzará el proceso de instalación de la rutina de auto-optimización para la operación de multiplicación de matrices utilizando librerías BLAS.

El proceso de instalación, al terminar, genera un fichero fuente que contiene la nueva operación de multiplicación de matrices disponible para su utilización por parte de los usuarios. Esta nueva función es totalmente transparente al usuario que desee utilizarla, ya que los únicos parámetros que deberá introducir para su uso serán el de las matrices de entrada A y B junto a sus dimensiones, y los correspondientes para la matriz de salida C. De esta forma quedará definida la operación de multiplicación. Sólo, en el caso de que el usuario lo desee, éste podrá además indicar la cantidad de *threads* con la que va trabajar.

Una vez que se lanza, en función de los tamaños de entrada de las matrices, la propia rutina buscará la configuración que mejor se ajuste a estos valores y automáticamente mandará los correspondientes procesos de multiplicación de matrices haciendo uso de librerías BLAS con las matrices y submatrices correspondientes. La rutina enviará A, B_{cpu} y C_{cpu} a la CPU y esta realizará la multiplicación con la librería BLAS de MKL. De la misma forma hará con las GPUs, que operarán con cuBLAS.

De una forma sencilla, el usuario, un programador o un científico, realizará la operación de multiplicación de matrices de manera fácil y cercana a la óptima. Las operaciones de multiplicación estarán totalmente optimizadas al usar librerías BLAS para CPU y GPU. Y, en conjunto, la operación de multiplicación de matrices también está optimizada en cuanto al rendimiento gracias a un reparto que lo favorece, información suministrada por la rutina de auto-optimización.

4. Experimentos Multicore+multiGPU

En la siguiente sección se mostrarán los experimentos llevados a cabo en la instalación de la rutina de auto-optimización para la multiplicación de matrices con librerías BLAS en multicore+multiGPU y utilizando búsqueda guiada. Estas pruebas de instalación se realizarán un sistema homogéneo y en otro heterogéneo pudiendo ver así su comportamiento de forma global. También, a título ilustrativo, se comentará el resultado de una instalación completa.

4.1. Experimentos con instalación completa

La principal dificultad de la instalación completa radica en el gran número de combinaciones distintas que hay que probar para recorrer completamente el árbol. Este número viene determinado por tres variables: el número de elementos del $conjunto\ instalacion$, el tamaño de las matrices y el tamaño del bloque de desplazamiento, tb. El proceso real de instalación de esta rutina implicaría días, por no decir semanas, de dedicación exclusiva.

Por esta cuestión, y solamente con interés ilustrativo, se mostrará una instalación denominada "*de juguete*" en un entorno homogéneo con el siguiente $conjuntoinstalacion = \{1000, 2000, 4000, 8000\}$, un valor tb no absoluto y sí porcentual de $tb = 10\,\%$ el tamaño de problema (tb tomará los valores $tb = \{100, 200, 400, 800\}$). Además, para hacerlo más sencillo aún, el tamaño de los threads dispuestos para el cómputo se fija en 10, $th = 10$. Las características del nodo utilizado en el sistema *Prometeo* son:

- 1 CPU Intel Xeon con 16 cores.
- 64GB de RAM.
- 2 GPU NVIDIA Tesla K40m

Pese a todo, con esta configuración aparentemente tan sencilla, el algoritmo registró más de 60 comprobaciones en cada uno de los tamaños. Hay que tener en cuenta que al utilizar un valor porcentual para tb el número de búsquedas y comprobaciones será el mismo en todos los tamaños del $conjunto\ instalacion$.

Si analizamos la tabla 1, se aprecia que responde a lo esperado y fácil de intuir al haber pocos elementos de cómputo disponibles y ser las GPUs homogéneas. En valores pequeños, $N = \{1000, 2000\}$, la CPU mantiene una alta carga de trabajo dejando las GPUs un tanto descompensadas. Conforme aumenta el tamaño, $N = \{4000, 8000\}$, se incrementa significativamente el trabajo de las GPUs, y, además, al tratarse de componentes homogéneos, se iguala el trabajo asignado a cada una. Por su parte, el trabajo de la CPU incluso llega a ser nulo con $N = 4000$ y apenas con un bloque de trabajo, $tb = 800$ con $N = 8000$.

Como se aprecia, el tiempo de la instalación completa para esta configuración tan pequeña es significativamente bajo, poco más de minuto y medio, pero, como se ha explicado en distintos puntos del documento, es una situación irreal aunque ilustrativa.

Tabla 1: Instalación completa en GPUs homogéneas

N	GPU_1	GPU_2	CPU	th	GFLOPS
1000	400	200	400	10	644.138
2000	1000	600	400	10	1045.13
4000	2000	2000	0	10	1456.53
8000	4000	3200	800	10	1928.71

Instalación completa: 106.529 segundos.

El proceso de búsqueda completa sobre un conjunto de instalación con la configuración:

- $conjunto\ instalacion = \{768, 1536, 2304, 3072, 3840, 4608, 5376, 6144, 6912, 7680\}$
- $tb = 192$
- 6 GPUs y una CPU multicore.

y con parámetros similares a los utilizados en los procesos de búsqueda guiada tanto homogénea como heterogénea, implicaría tiempos de instalación de días con un uso exclusivo.

4.2. Experimentos con instalaciones guiadas homogéneas

Los experimentos para la instalación guiada homogénea se realizarán en el mismo nodo de cómputo que el caso de la instalación completa. Dentro del sistema Prometo, se usa un nodo con 1 CPU Intel Xeon con 16 cores, 64GB de RAM y 2 GPU NVIDIA Tesla K40m. La metodología que se seguirá es la siguiente:

1. Instalación guiada con los valores del *conjunto validación*, registrando configuraciones que obtengan el mayor rendimiento para cada tamaño de entrada.
2. Instalación guiada con los valores del *conjunto instalación* quedando registradas las mejores configuraciones para cada tamaño.
3. Utilizando la información de la instalación guiada del conjunto instalación, realizar la operación de multiplicación de matrices con los tamaños de entrada del *conjunto validación*.
4. Comparativa de tiempos y rendimiento de la instalación del *conjunto validación* y los obtenidos en el paso 3 por este mismo *conjunto validación* sobre la instalación del paso 2.

El proceso se repetirá para instalaciones guiadas con búsqueda en vecindad aleatoria e instalaciones guiadas con búsqueda en vecindad completa. Además, para cada una de

ellas se realizará la instalación para un valor $umbral$ al 2 %, 5 % y 10 %.

Los parámetros de entrada que se utilizarán en la instalación serán, para el $conjunto\ instalacion = \{768, 1536, 2304, 3072, 3840, 4608, 5376, 6144, 6912, 7680, 8448, 9216, 9984, 10752\}$, el tamaño del bloque de desplazamiento $tb = 192$, dos GPUs con las mismas características y un valor $max\_num\_threads = 16$ de threads disponibles para computación en CPU. Para la instalación del conjunto validación lo único que variará será el propio conjunto validación con los valores $conjunto\ validacion = \{384, 1152, 1920, 2688, 3456, 4224, 4992, 5760, 6528, 7296, 8064, 8832, 9600, 10368, 11136\}$

Los tiempos de instalación para estas configuraciones (tabla 2), rondan los 4 minutos aproximadamente en el caso de instalación random o aleatoria y los 10 minutos en la instalación completa. Estos tiempos son asumibles en cualquier entorno de alta computación o supercomputación.

Tabla 2: Tiempos de instalación en entorno homogéneo

Instalación Homogénea (2 GPUs + Multicore)		
	Random	**Completa**
2 %	4'17"	5'39"
5 %	3'45"	9'33"
10 %	4'14"	9'18"

Si se analiza el comportamiento de la *instalación guiada con vecindad aleatoria*, tabla 3 y figura 3, se observa que las tres instalaciones siguen un proceso de búsqueda similar, y que aumentar el ratio de búsqueda aumentando el $umbral$ no quiere decir que se obtengan mejores valores.

En sistemas homogéneos, a priori, es normal pensar que conforme aumenten los tamaños de entrada el reparto del trabajo en CPU sea casi residual, mientras que en GPU sea equilibrado entre ellas haciéndose cargo además las GPUs de la mayor parte del cómputo. Como se puede comprobar, ocurre lo predecible. El cómputo en las GPUs tiende a igualarse quedando una carga de trabajo significativamente menor para la CPU, aunque, eso sí, empleando un gran número de los threads disponibles, $max\_num\_threads = 16$.

La tabla 4 y la figura 4 comparan, de forma similar al caso random, las prestaciones obtenidas cuando se realiza la instalación con vecindad completa. Este método conlleva mayor tiempo de instalación al explorar una vecindad más amplia en la mejora de cada elemento, y consecuentemente se deben obtener mayores prestaciones, aunque as observa que la mejora no es considerable.

Para evaluar esta técnica se procede con la metodología descrita. Hacer dos instalaciones de la rutina de auto-optimización, una para el *conjunto validación* y otra para

Tabla 3: Instalaciones guiadas usando vecindad aleatoria en un sistema homogéneo.

Random 2 %					
n	gpu1	gpu2	cpu	th	GFLOPS
768	192	192	384	15	278,239
1536	960	192	384	13	446,7
2304	1056	672	576	12	747,366
3072	1056	1440	576	15	852,428
3840	1220	1762	858	16	**944,885**
4608	1663	1804	1141	16	983,625
5376	2431	1804	1141	16	1140,2
6144	2420	2611	1113	13	1376
6912	2612	3187	1113	13	1300,04
7680	3380	3187	1113	12	1570,31
8448	3956	3763	729	13	1481,79
9216	4532	4147	537	16	1535,73
9984	5300	4147	537	16	1525,2
10752	5108	4915	729	13	**1643,35**
Random 5 %					
n	gpu1	gpu2	cpu	th	GFLOPS
768	0	192	576	12	**383,907**
1536	512	512	512	12	**726,595**
2304	960	960	384	16	**1029,17**
3072	1280	1344	448	12	**1173,23**
3840	1580	1626	634	16	888,318
4608	2348	1626	634	13	**1569,3**
5376	3116	1626	634	15	**1209,71**
6144	3116	2202	826	13	1427,71
6912	3168	2583	1161	13	**1451,6**
7680	3168	3351	1161	13	**1803,09**
8448	3168	4119	1161	15	**1886,5**
9216	3744	4119	1353	15	**1627,5**
9984	3744	4887	1353	15	**1672,8**
10752	4512	5079	1161	15	1453,56
Random 10 %					
n	gpu1	gpu2	cpu	th	GFLOPS
768	0	192	576	13	281,872
1536	428	362	746	13	478,624
2304	768	768	768	15	571,896
3072	1388	1130	554	12	947,247
3840	1280	1088	1472	14	817,136
4608	1708	1309	1591	14	955,478
5376	1832	1798	1746	16	1032,81
6144	2984	2182	978	14	**1587,38**
6912	2984	3142	786	16	1268,56
7680	2954	3064	1662	14	1651,64
8448	3530	3064	1854	13	1612,45
9216	3485	3159	2572	15	1350,36
9984	3485	3927	2572	15	1396,56
10752	3864	3913	2975	16	1197,47

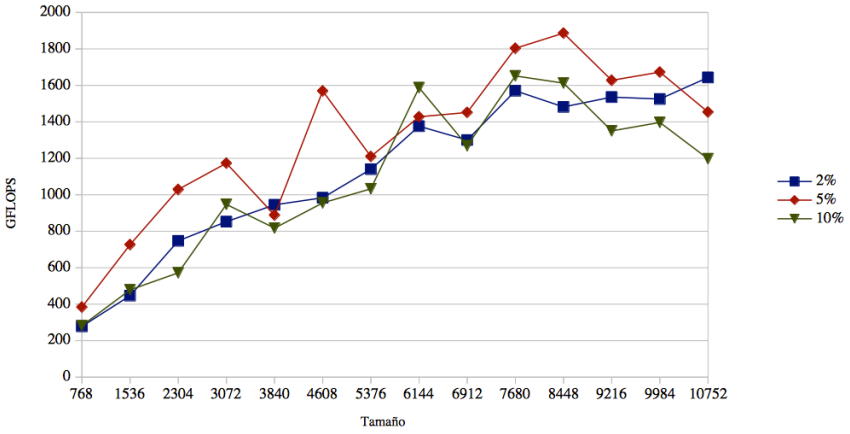


Figura 3: Instalación vecindad aleatoria en entorno homogéneo.

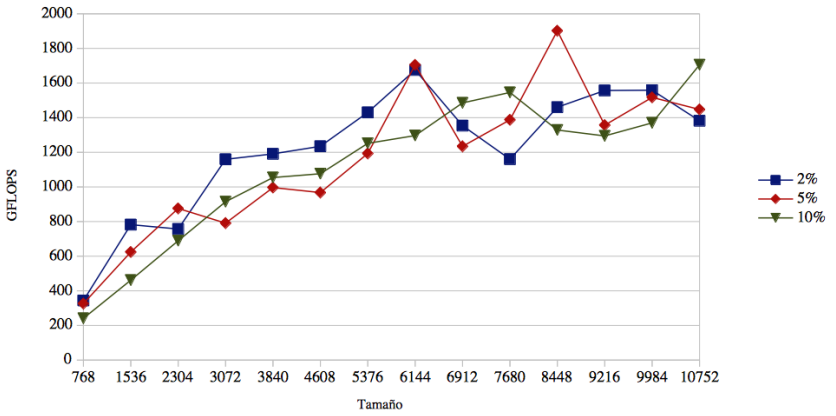


Figura 4: Instalación vecindad completa en entorno homogéneo.

Tabla 4: Instalaciones guiadas usando vecindad completa en un sistema homogéneo.

Vecindad 2 %					
n	gpu1	gpu2	cpu	th	GFLOPS
768	0	0	768	14	**344,195**
1536	512	512	512	14	**781,935**
2304	768	768	768	16	756,888
3072	1344	1152	576	16	**1159,41**
3840	1344	1920	576	15	**1191,01**
4608	1690	1984	934	12	**1234,7**
5376	2458	1984	934	16	**1430,32**
6144	2650	2560	934	12	1675,33
6912	2650	3328	934	13	1353,87
7680	2975	3195	1510	15	1160,19
8448	3022	3175	2251	15	1460,33
9216	3022	4135	2059	16	**1557**
9984	3022	4903	2059	12	**1558,83**
10752	3022	4903	2827	16	1382,01
Vecindad 5 %					
n	gpu1	gpu2	cpu	th	GFLOPS
768	0	0	768	12	325,893
1536	512	512	512	11	623,947
2304	940	682	682	14	**875,743**
3072	940	1450	682	13	790,776
3840	1280	1280	1280	13	996,271
4608	1708	1450	1450	16	967,303
5376	2176	1600	1600	14	1193,08
6144	3136	2176	832	13	**1704,77**
6912	3136	2944	832	16	1234,27
7680	3136	3712	832	16	1387,36
8448	3582	3537	1329	12	**1901,8**
9216	3909	3858	1449	15	1356,61
9984	4677	3858	1449	15	1517,18
10752	5230	4346	1176	13	1447,29
Vecindad 10 %					
n	gpu1	gpu2	cpu	th	GFLOPS
768	576	0	192	16	240,882
1536	1004	170	362	15	461,61
2304	768	960	576	13	689,088
3072	939	1344	789	16	914,385
3840	1324	1498	1018	14	1054,32
4608	1484	1866	1258	11	1076,29
5376	2252	1866	1258	14	1251,67
6144	2444	2634	1066	13	1296,64
6912	2444	3402	1066	13	**1485,27**
7680	2830	3247	1603	11	**1545,69**
8448	2830	4015	1603	15	1328,71
9216	2830	4207	2179	13	1294,63
9984	3268	4286	2430	14	1369,82
10752	4036	4478	2238	13	**1705,39**

el *conjunto instalación*. Comparar la instalación del *conjunto validación*, denominándolo *instalación nativa*, con los valores obtenidos por este mismo *conjunto validación* ya utilizando configuraciones proporcionadas por la rutina de auto-optimización del *conjunto instalación*. Repitiendo este proceso un valor *umbral* al 2 %, 5 % y 10 %. Como se puede apreciar en la figura 5, el rendimiento de una instalación nativa con un valor $umbral = 2\,\%$, es muy similar a los obtenidos en la rutina de auto-optimización, pero, conforme aumenta el valor umbral, $umbral = 5\,\%$ y $umbral = 10\,\%$, también aumenta espacio de búsqueda y por tanto el rendimiento de una instalación con valores nativos (mismos valores utilizados en una rutina ya instalada) aumenta. Tabla 5 y figuras 6 y 7.

Comparativa entre instalación aleatoria 2 % del conjunto validación y este mismo conjunto utilizando la rutina de auto-optimización con una instalación previa del conjunto instalación.

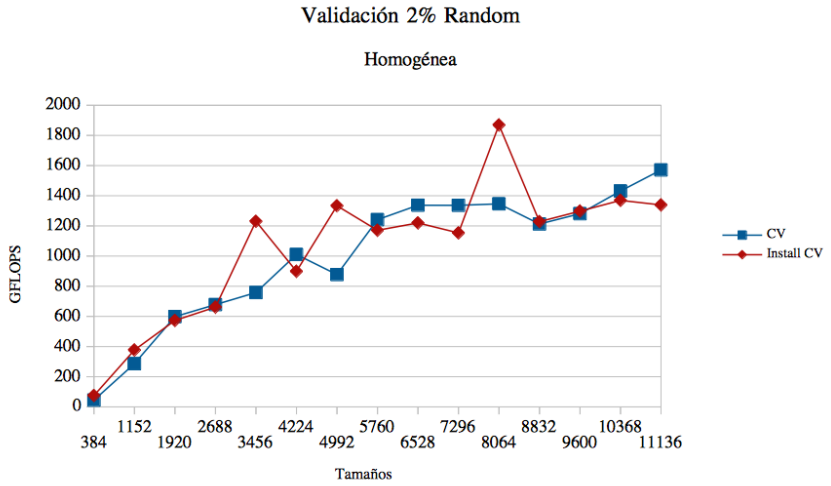


Figura 5: Instalación Homogénea. Método de búsqueda con vecindad aleatoria y umbral al 2 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación con esos valores (Install CV).

En el caso de la *instalación búsqueda guiada con vecindad completa* (tabla 6 y figuras 8, 9 y 10), precisamente al examinar todos los posibles vecinos definidos, aumenta el rendimiento con las configuraciones seleccionadas en comparación con el de las seleccionadas con vecindad random. La distancia entre el uso de una instalación guiada frente a la óptima disminuye. Este comportamiento viene motivado por la ampliación del espacio de búsqueda al evaluar a todos los vecinos.

Según los experimentos realizados sobre el *conjunto de validación* (tablas 5 y 6), aplicar la búsqueda utilizando un tipo de vecindad aleatoria o completa no mejora significativamente uno frente a otro. Curiosamente, aunque la búsqueda aleatoria ha obtenido el mayor número de valores máximos en todo el conjunto de entradas, la configuración de mayor rendimiento se ha dado utilizando *vecindad completa* con *umbral=10 %*. Este era el resultado esperado al ser el algoritmo que teóricamente más configuraciones evalúa.

Tabla 5: Comparativas instalaciones aleatorias del conjunto validación y este mismo conjunto utilizando la rutina de auto-optimización con una instalación previa del conjunto instalación.

N	2 % Random		5 % Random		10 % Random	
	CV	Install CV	CV	Install CV	CV	Install CV
384	44,4788	75,1922	55,7564	66,2652	56,4791	62,1876
1152	286,4	376,975	327,403	320,441	294,428	530,187
1920	598,325	572,872	518,334	645,85	407,699	649,289
2688	678,087	661,332	555,098	806,952	546,943	815,046
3456	758,164	1230,39	561,097	988,309	234,357	1211,27
4224	1010,61	898,855	677,305	1292,03	635,582	1051,28
4992	877,314	1333,42	700,515	1157,43	679,115	1086,04
5760	1241,46	1170,73	844,954	1432,63	842,734	1076,32
6528	1336,97	1219,74	988,897	1223,43	1034,45	1369,23
7296	1336,41	1153,6	1137,38	1162,94	996,669	1190,69
8064	1346,08	1869,35	1252,42	1476,14	1038,8	1581,72
8832	1212,46	1228,35	981,626	1606,1	898,937	1451,36
9600	1280,68	1297,79	1000,78	1549,88	1117,64	1396
10368	1431,25	1369,4	1240,71	1553,56	1110,65	1291,89
11136	1570,92	1338,83	1650,36	1367,84	1368,16	1700,89

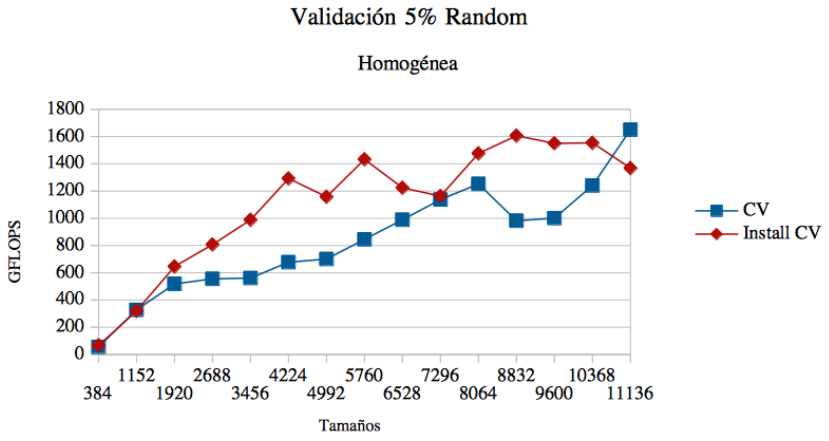


Figura 6: Instalación Homogénea. Método de búsqueda con vecindad aleatoria y umbral al 5 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

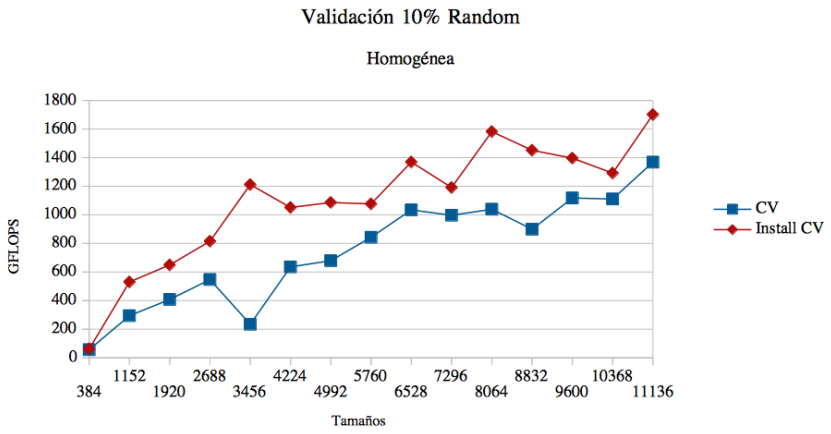


Figura 7: Instalación Homogénea. Método de búsqueda con vecindad aleatoria y umbral al 10 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

Tabla 6: Instalación Homogénea. Método de búsqueda con vecindad completa. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

N	2 % Vecindad Completa		5 % Vecindad Completa		10 % Vecindad Completa	
	CV	Install CV	CV	Install CV	CV	Install CV
1152	129,561	169,435	113,419	166,069	132,005	172,289
1920	326,931	297,515	310,407	310,658	278,29	327,711
2688	312,086	387,458	449,629	453,117	321,488	478,952
3456	550,325	445,543	500,555	495,549	392,769	536,199
4224	526,337	578	602,099	575,185	483,146	603,945
4992	623,681	541,488	657,359	679,982	503,576	607,404
5760	569,16	611,636	674,758	689,57	643,354	627,684
6528	692,501	664,373	680,877	754,39	593,092	747,2
7296	769,966	673,753	712,14	766,502	659,042	723,707
8064	698,722	669,235	742,738	742,989	663,379	704,962

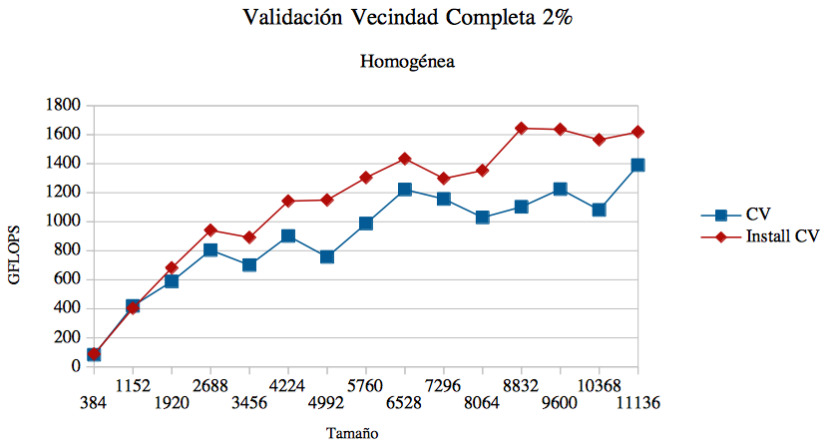


Figura 8: Instalación Homogénea. Método de búsqueda con vecindad completa y umbral al 2 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

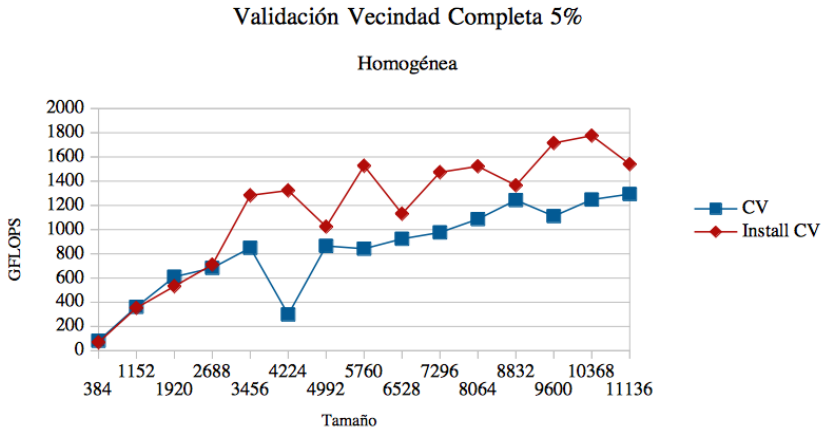


Figura 9: Instalación Homogénea. Método de búsqueda con vecindad completa y umbral al 5 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

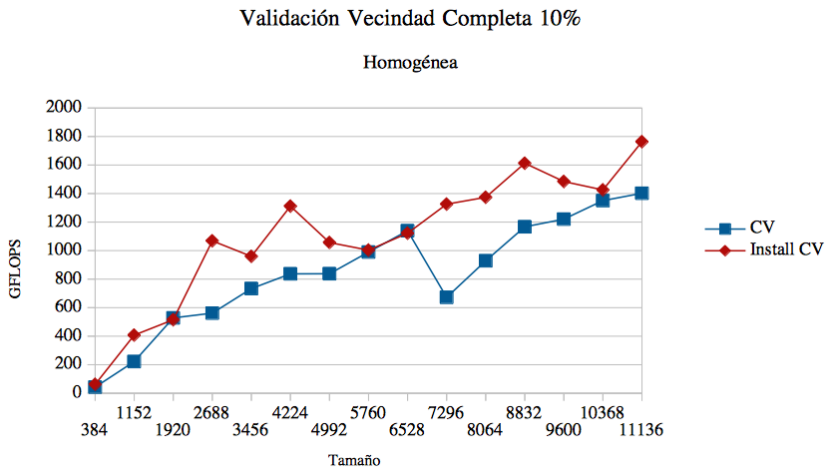


Figura 10: Instalación Homogénea. Método de búsqueda con vecindad completa y umbral al 10 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

Respecto al $umbral$, en el caso del sistema homogéneo estudiado, no se observa una mejora significativa obteniendo un comportamiento similar. Aunque, curiosamente, los máximos valores se han encontrado en la columna del $umbral = 2\,\%$ con vecindad aleatoria 3. Este comportamiento puede deberse a que la búsqueda guiada pretende que el algoritmo llegue a aquellas zonas del espacio de búsqueda donde pueda haber configuraciones óptimas. Y aumentar el $umbral$, al evaluar un subconjunto aleatorio de la totalidad de vecinos definida, nos lleve a configuraciones peores.

Utilizando el ejemplo de la ascensión de colinas, los umbrales $umbral = 2\,\%$ y $umbral = 5\,\%$, permiten evaluar valores óptimos que se encuentran en torno a la cima. Cuando se evalúan los vecinos para producir un desplazamiento, como el vecino a evaluar es una elección aleatoria, puede darse que el salto hacia el mejor vecino evaluado en lugar de ser en la zona superior sea en la zona inferior. El sistema aquí pronto se detendrá al tener un $umbral$ bajo. Pero, en esta misma situación con un umbral mayor, $umbral = 10\,\%$, el sistema seguirá evaluando vecinos y al tratarse de un análisis de la vecindad aleatorio podría descender la colina, evaluar configuraciones peores y por tanto innecesarias hasta, en algún momento, detener la búsqueda al sobrepasar el $umbral$, aumentando así el número de comprobaciones pero no mejorando el resultado. Ver tabla 6 y figuras 8, 9 y 10.

En un entorno homogéneo es sencillo intuir un rendimiento si no óptimo sí aceptable, repartiendo a partes iguales la carga de trabajo entre las GPUs e incluso despreciando el cómputo de la CPU. Esto ocurre especialmente conforme aumentan los tamaños de las matrices. Pero el coste de una instalación guiada es tan bajo (el peor caso de este estudio está por debajo de los **10 minutos** para la instalación guiada con vecindad completa y un $umbral = 10\,\%$) que merece la pena la instalación de la rutina de auto-optimización y mejorar así el rendimiento de la operación de la multiplicación matricial.

4.3. Experimentos con instalación guiada heterogénea

La misma técnica y la misma metodología empleada en un entorno homogéneo será el que se utilice en un entorno heterogéneo, pudiendo comprobar cómo la rutina de auto-optimización se adapta a él. Los experimentos se realizaron en el clúster Heterosolar del grupo de investigación de Computación Científica y Programación Paralela de la Universidad de Murcia, en el nodo denominado Júpiter, con las siguientes características:

- Dos hexa-cores (12 cores en total) Intel Xeon E5-2620 a 2.00GHz.
- 32 GB de memoria RAM.
- 2 GPUs NVIDIA Fermi Tesla C2075 con 5375 MBytes de Global Memory y 448 cores con 14 Streaming Multiprocessors y 32 Streaming Processors.
- 4 GPUs agrupadas en dos tarjetas, cada una con dos dispositivos NVIDIA GeForce GTX 590 con 1536 MBytes de Global Memory y 512 CUDA cores con 16 Streaming Multiprocessors y 32 Streaming Processors y un total de 1024 cores por tarjeta.

Los parámetros de entrada que se utilizarán en la instalación serán muy similares a los del proceso homogéneo. Por tanto, para el $conjunto\ instalacion = \{768, 1536, 2304, 3072, 3840, 4608, 5376, 6144, 6912, 7680\}$, el tamaño del bloque de desplazamiento $tb = 192$, seis GPUs heterogéneas con las características anteriormente descritas y un valor $max_num_threads = 12$ de threads disponibles para computación en CPU. El conjunto de validación tendrá los tamaños de entrada $conjunto\ validacion = \{1152, 1920, 2688, 3456, 4224, 4992, 5760, 6528, 7296, 8064\}$.

El tiempo de la instalación utilizando búsqueda guiada con vecindad aleatoria (tabla 7) es bastante similar al de la instalación homogénea pese a disponer de 6 GPUs frente a las 2 del sistema homogéneo. Sin embargo, los tiempos del proceso de instalación utilizando una vecindad completa son significativamente mayores, llegando a sobrepasar la hora de instalación en el caso de la instalación utilizando búsqueda guiada con vecindad completa y un $umbral = 10\,\%$. El motivo principal de esta diferencia viene dado por el número de comprobaciones que se realizan según el tipo de vecindad, marcadas por el número de dispositivos. En una búsqueda con vecindad aleatoria, el grupo de vecinos a evaluar se reduce a 3 elementos en el caso homogéneo, 2 GPUs + 1 CPU, mientras que se eleva a 7 elementos en el caso práctico heterogéneo, 6 GPUs + 1 CPU. En el caso de la vecindad completa al evaluar todos los vecinos definidos el techo es $n(n-1)$, mientras en el caso homogéneo apenas son *6 comprobaciones* en el caso heterogéneo, por el incremento del número de dispositivos, el valor sube a *42 comprobaciones*.

Pese a todo, dentro de un sistema de alta computación, un tiempo de instalación como el indicado, que incluso supera la hora de cómputo, puede resultar aceptable. Especialmente, si como beneficio se obtiene la operación de la multiplicación matricial optimizada con librerías BLAS y mejorada, en términos de rendimiento, al utilizarla sobre la rutina

Tabla 7: Tiempos de instalación sistema heterogéneo

Instalación Heterogénea (6 GPUs + Multicore)		
	Random	**Completa**
2 %	3'27"	17'01"
5 %	5'30	15'01"
10 %	6'29"	1h 05 '04"

de auto-optimización creada.

Analizando las *instalaciones aleatorias* (tabla 8 y figura 11), el rendimiento de las 3 instalaciones es muy similar, aunque la instalación usando vecindad aleatoria con un $umbral = 5\,\%$ es la que destaca por encima de las otras al presentar el mayor número de mejores valores por tamaño de entrada. Esto ocurre por la misma situación descrita en el apartado anterior de la instalación homogénea. En el proceso de búsqueda guiada, cuando comienza la búsqueda para un nuevo tamaño, el proceso $iniciarBusqueda$ evalúa el mejor vector configuración para reiniciar el proceso. Esto nos lleva a evaluar configuraciones óptimas. Ampliar el $umbral$, como ya estamos en torno a estas configuraciones óptimas, puede suponer evaluar estados peores. Con un $umbral = 5\,\%$ el sistema alcanza a evaluar valores óptimos cercanos que mejoran los explorados, mejorando así los resultados con $umbral = 2\,\%$

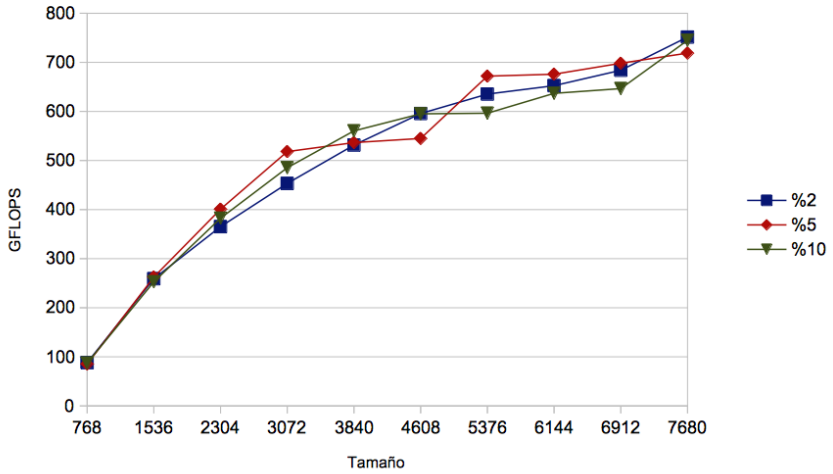


Figura 11: Instalación vecindad aleatoria en entorno heterogéneo.

En el caso de la *instalación con vecindad completa* (tabla 9 y figura 12), al analizar todo el espectro de vecinos eliminando el factor aleatoriedad, todas las configuraciones (buenas y malas) se evaluarán. Además, el aumentar el umbral ampliará aún más la exploración del espacio de búsqueda, consiguiendo mejores configuraciones.

Tabla 8: Instalaciones guiadas usando vecindad aleatoria en un sistema heterogéneo.

Random 2 %									
n	gpu1	gpu2	gpu3	gpu4	gpu5	gpu6	cpu	th	GFLOPS
768	384	0	0	0	0	192	192	12	**87,9935**
1536	222	219	219	219	219	219	219	9	259,117
2304	296	292	292	292	292	548	292	12	365,229
3072	379	629	373	373	373	572	373	11	453,24
3840	471	654	462	462	462	867	462	9	531,41
4608	567	698	558	558	558	1111	558	7	**595,722**
5376	662	814	651	459	651	1488	651	11	635,364
6144	961	937	757	620	757	1355	757	11	652,475
6912	1013	1248	865	768	865	1288	865	12	683,902
7680	1013	2016	865	768	865	1288	865	12	751,485
Random 5 %									
n	gpu1	gpu2	gpu3	gpu4	gpu5	gpu6	cpu	th	GFLOPS
768	0	0	384	0	192	0	192	10	85,2514
1536	222	219	219	219	219	219	219	9	**262,427**
2304	296	356	292	292	292	484	292	10	**400,899**
3072	380	678	373	373	192	714	362	10	**518,159**
3840	471	947	654	462	326	526	454	7	536,453
4608	568	913	698	558	458	861	552	12	544,94
5376	760	1105	698	750	458	861	744	11	**671,935**
6144	871	1262	797	857	331	1176	850	11	**675,852**
6912	949	1222	893	936	563	1418	931	11	**698,19**
7680	845	1417	802	1024	953	1619	1020	12	718,836
Random 10 %									
n	gpu1	gpu2	gpu3	gpu4	gpu5	gpu6	cpu	th	GFLOPS
768	0	0	0	0	192	0	576	12	87,1699
1536	222	219	219	219	219	219	219	10	252,792
2304	296	292	292	292	292	548	292	11	382,457
3072	379	629	373	373	373	572	373	12	485,34
3840	475	786	466	466	466	715	466	8	**560,4**
4608	570	795	561	561	561	999	561	11	595,108
5376	668	927	654	654	654	1165	654	12	596,308
6144	766	1059	747	747	747	1331	747	12	636,822
6912	864	1191	840	840	840	1497	840	10	646,484
7680	864	1959	840	840	840	1497	840	12	744,838

Tabla 9: Instalaciones guiadas usando vecindad completa en un sistema heterogéneo.

Vecindad 2 %									
n	gpu1	gpu2	gpu3	gpu4	gpu5	gpu6	cpu	th	GFLOPS
768	384	0	0	0	0	0	384	9	84,0651
1536	222	219	219	219	219	219	219	9	231,78
2304	296	548	292	292	292	292	292	9	337,595
3072	379	572	373	373	373	629	373	11	452,06
3840	471	867	462	462	462	654	462	11	509,253
4608	567	855	558	558	558	954	558	11	574,038
5376	662	997	651	651	651	1113	651	12	545,706
6144	757	1139	744	744	744	1272	744	9	617,107
6912	757	1907	744	744	744	1272	744	12	680,596
7680	901	1707	889	889	889	1516	889	12	721,402
Vecindad 5 %									
n	gpu1	gpu2	gpu3	gpu4	gpu5	gpu6	cpu	th	GFLOPS
768	192	0	0	0	0	0	576	10	85,2993
1536	222	411	219	219	219	219	27	9	242,254
2304	296	452	292	292	292	548	132	11	360,614
3072	381	778	373	373	389	730	0	12	449,25
3840	538	972	466	466	486	912	0	12	530,159
4608	647	1166	559	559	583	1094	0	12	610,275
5376	756	1360	652	652	680	1276	0	12	622,237
6144	756	1360	652	652	680	1276	768	11	**706,277**
6912	852	1530	733	733	765	1435	864	11	703,895
7680	948	1700	814	814	850	1594	960	12	731,788
Vecindad 10 %									
n	gpu1	gpu2	gpu3	gpu4	gpu5	gpu6	cpu	th	GFLOPS
768	0	0	0	192	192	0	384	12	**90,2185**
1536	222	219	219	219	219	219	219	12	**273,594**
2304	296	548	292	292	292	292	292	11	**393,096**
3072	397	922	389	389	389	389	197	7	**513,125**
3840	484	874	474	474	474	730	330	11	**626,226**
4608	579	860	566	566	566	1010	461	9	**675,462**
5376	678	1133	664	664	664	985	588	8	**696,191**
6144	678	1133	664	856	664	1753	396	8	692,598
6912	813	1579	799	935	607	1570	609	11	**723,066**
7680	942	1860	735	1023	856	1470	794	11	**782,469**

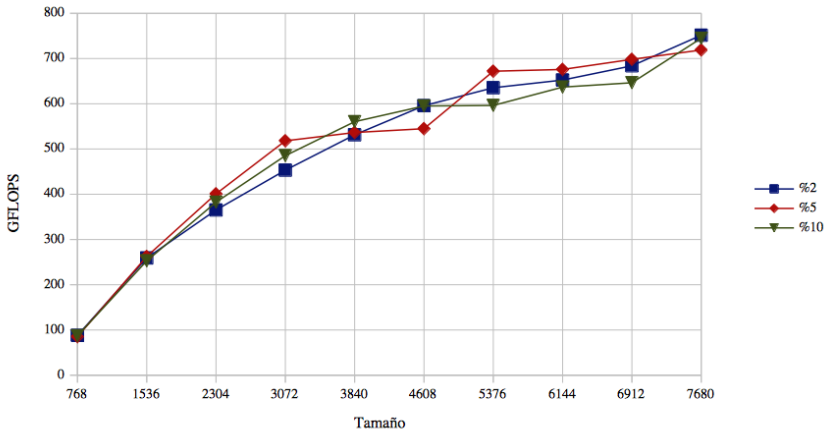


Figura 12: Instalación vecindad completa en entorno heterogéneo.

Comparamos los resultados con una *instalación del conjunto de validación* con los obtenidos con el uso de la rutina de auto-optimizaicón de la operación multiplicación matricial para el mismo conjunto de validación.

En el caso de la *instalación guiada con vecindad aleatoria* (tabla 10 y figuras 13, 14 y15), conforme aumenta el valor $umbral$, más cercanos están los resultados obtenidos con auto-optimización a los valores óptimos de la instalación. Estos valores llegan incluso a igualarse en el caso de las instalaciones con un $umbral = 10\,\%$.

Tabla 10: Instalación Heterogénea. Método de búsqueda con vecindad aleatoria. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

N	2 % Random		5 % Random		10 % Random	
	CV	**Install CV**	**CV**	**Install CV**	**CV**	**Install CV**
1152	110,684	185,168	130,724	175,666	107,762	156,802
1920	268,784	301,29	313,403	318,151	302,635	307,047
2688	406,815	447,892	299,824	420,466	310,608	430,479
3456	314,289	535,751	363,888	504,865	496,864	515,254
4224	514,048	535,316	521,607	576,75	495,323	541,637
4992	563,353	641,196	539,966	641,57	603,715	591,901
5760	503,147	714,899	574,811	683,079	615,046	615,622
6528	605,853	701,123	627,874	698,459	660,695	663,377
7296	636,502	709,402	622,177	783,269	632,486	655,691
8064	624,098	751,179	658,319	804,647	688,118	689,044

En el caso de la *instalación guiada con vecindad completa* (tabla 11 y figuras 16, 17 y18), sucede un efecto similar al de la vecindad aleatoria, obteniendo rendimientos muy

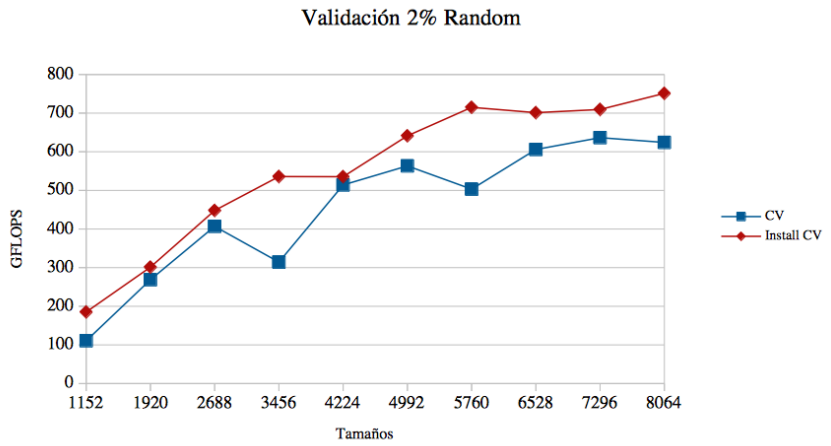


Figura 13: Instalación Heterogénea. Método de búsqueda con vecindad aleatoria y umbral al 2 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

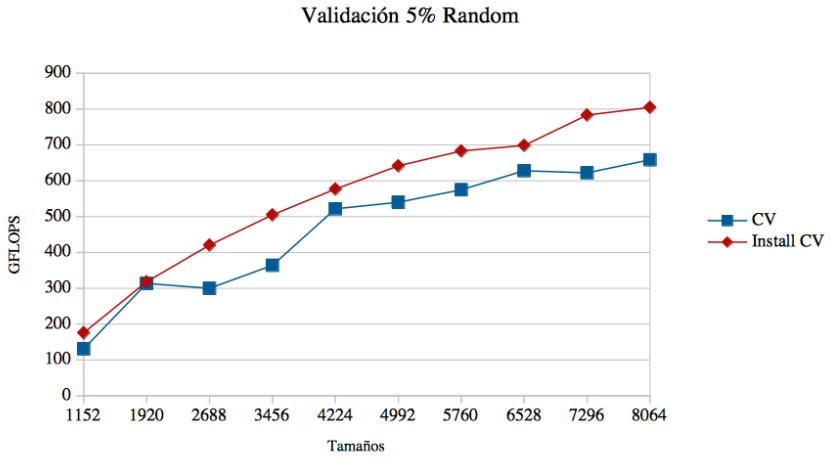


Figura 14: Instalación Heterogénea. Método de búsqueda con vecindad aleatoria y umbral al 5 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

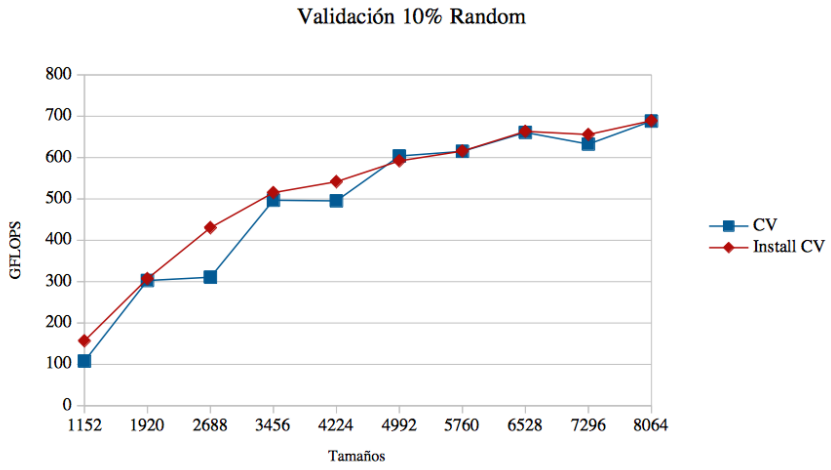


Figura 15: Instalación Heterogénea. Método de búsqueda con vecindad aleatoria y umbral al 10 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

similares.

Tabla 11: Instalación Heterogénea. Método de búsqueda con vecindad completa. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

N	2 % Vecindad Completa		5 % Vecindad Completa		10 % Vecindad Completa	
	CV	Install CV	CV	Install CV	CV	Install CV
1152	129,561	169,435	113,419	166,069	132,005	172,289
1920	326,931	297,515	310,407	310,658	278,29	327,711
2688	312,086	387,458	449,629	453,117	321,488	478,952
3456	550,325	445,543	500,555	495,549	392,769	536,199
4224	526,337	578	602,099	575,185	483,146	603,945
4992	623,681	541,488	657,359	679,982	503,576	607,404
5760	569,16	611,636	674,758	689,57	643,354	627,684
6528	692,501	664,373	680,877	754,39	593,092	747,2
7296	769,966	673,753	712,14	766,502	659,042	723,707
8064	698,722	669,235	742,738	742,989	663,379	704,962

Analizando los resultados, en un sistema heterogéneo ya no es tan fácil intuir una configuración aceptable, mucho menos si además se dispone de 6 GPUs + Multicore, como el caso del objeto de estudio. Como se ha podido observar en los experimentos, especialmente en las comparativas de validación, la rutina de auto-optimización con búsqueda guiada en sus dos opciones, aleatoria y completa, resulta ser una herramienta de gran utilidad a la hora de aumentar el rendimiento de sistemas heterogéneos que utilicen librerías BLAS. Esto se consigue además con un coste de cómputo global bajo. El mayor tiempo de instalación invertido en las pruebas ha sido el de la búsqueda guiada con vecindad

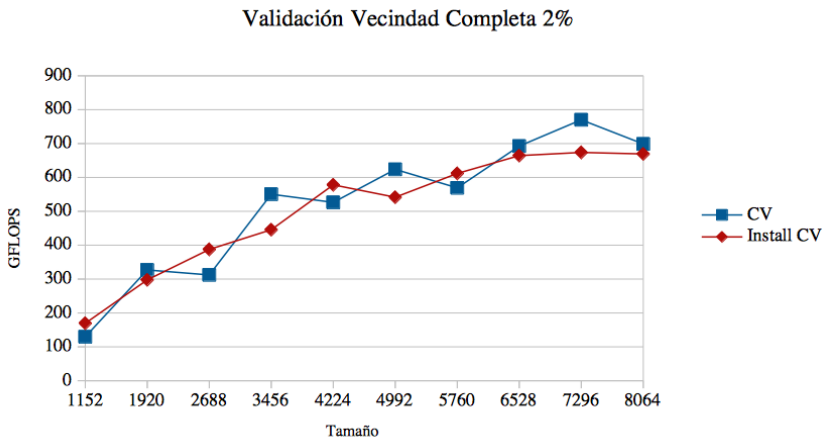


Figura 16: Instalación Heterogénea. Método de búsqueda con vecindad completa y umbral al 2 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

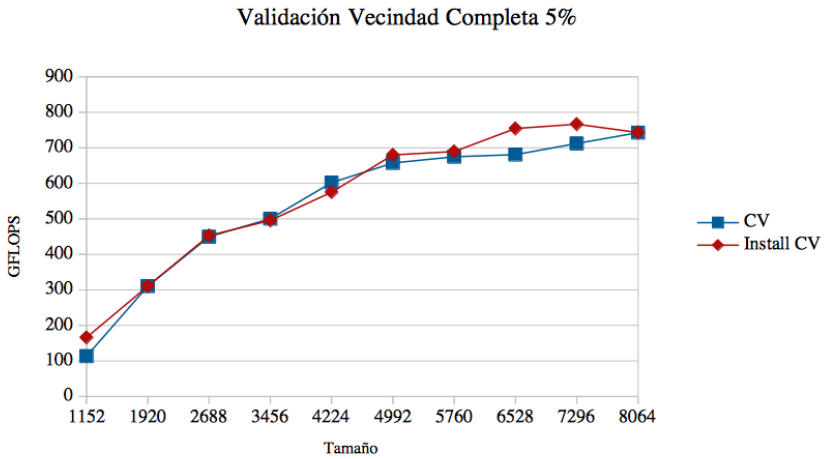


Figura 17: Instalación Heterogénea. Método de búsqueda con vecindad completa y umbral al 5 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

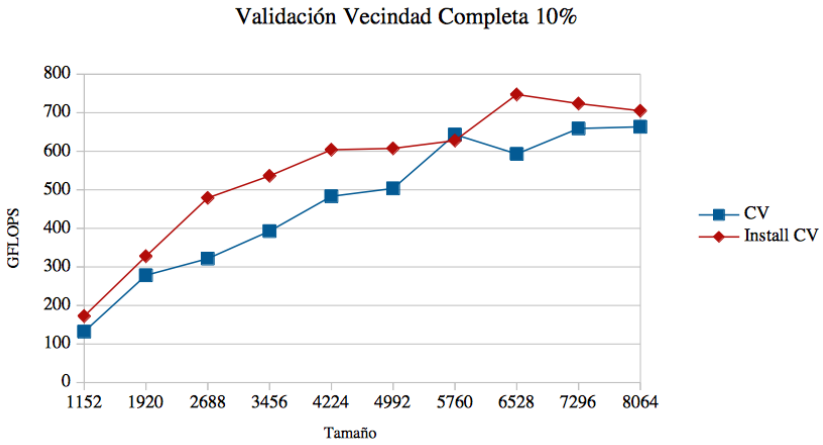


Figura 18: Instalación Heterogénea. Método de búsqueda con vecindad completa y umbral al 10 %. Comparativa del CV haciendo uso de la rutina de auto-optimización frente a una instalación nativa (Install CV).

completa y $umbral = 10\,\%$, con 1 hora 5 minutos y 4 segundos.

5. Conclusiones y Trabajo Futuro

5.1. Conclusiones

El presente Trabajo Fin de Grado comenzó mostrando la evolución y situación actual de los equipos destinados a la Computación de Alto Rendimiento, y cómo la ayuda de las librerías numéricas como BLAS, que están optimizadas para obtener el mayor rendimiento posible de estos equipos, han contribuido a ello.

En este TFG se estudia la conveniencia de utilizar técnicas de auto-optimización para rutinas de álgebra lineal en sistemas heterogéneos multicore+multiGPU. Los resultados iniciales se presentaron en [12]. Se ha podido comprobar que la optimización de estas librerías (en concreto MKL para multicore y cuBLAS para GPU) no es suficiente para una explotación lo más eficiente posible de los nodos computacionales actuales, que tienen una configuración heterogénea combinando multicores con una o varias GPUs. Cuando los sistemas crecen en número de dispositivos de cómputo y estos presentan lógicas de funcionamiento dispares (dispositivos multicore y manycore) pretender obtener el mismo rendimiento global que sumando el de las distintas partes del sistema es una tarea difícil.

Es aquí donde las técnicas de auto-optimización de rutinas BLAS han mostrado su fortaleza y han demostrado que, con un aprendizaje en el proceso de instalación a un coste en tiempo de cómputo muy bajo (recordar que el peor caso consumía apenas una hora), se pueden conseguir rendimientos cercanos a los óptimos, mejorando significativamente el rendimiento de las operaciones, especialmente si los equipos disponen de varios dispositivos de cómputo y además son heterogéneos.

Las técnicas de auto-optimización de rutinas BLAS para CPU+multiGPU (en concreto la desarrollada en este documento para la operación multiplicación matricial) han demostrado ser una opción más que válida para este menester, facilitando la tarea del desarrollador y dejando en manos de la rutina de auto-optimización la labor de realizar la operación deseada de forma eficiente.

Además, al operar directamente sobre la operación BLAS, su uso es totalmente transparente para el desarrollador, pudiendo utilizarla en funciones y operaciones de nivel superior que requieran operar con la multiplicación matricial.

5.2. Trabajo futuro

Los resultados experimentales obtenidos, aunque han demostrado que el uso de la auto-optimización en rutinas BLAS es una opción más que aconsejable en grandes sistemas de cómputo, también han sacado a la luz el margen de mejora que aún tiene.

En concreto, que los tiempos de instalación de la rutina sean en términos humanos aceptables, e incluso bajos, hace pensar que se puede emplear tiempo de cómputo de la

instalación en abordar y estudiar el ajuste de otros parámetros que puedan ayudar a mejorar más y mejor el rendimiento de la rutina. Uno de estos parámetros es el del tamaño del bloque de desplazamiento tb, fijado siguiendo el criterio de encontrar un cambio que mejore el rendimiento pero manteniéndose fijo sin evolucionar con el algoritmo, ya sea en las búsquedas de un tamaño de entrada o variando conforme va aumentando el tamaño de las entradas de las matrices en el proceso de instalación.

Pese a esta y otras mejoras que se puedan realizar, la rutina está en una situación de uso en sistemas Multicore+multiGPU, y, por tanto, donde poder evolucionar y crecer de la siguiente forma:

- Utilizar esta multiplicación de matrices ya optimizada sobre rutinas de un nivel superior y estudiar el aumento de rendimiento.
- Utilizar técnicas de auto-optimización sobre otro tipo de rutinas, tal vez de mayor nivel.
- Utilizar esta técnica de auto-optimización de la operación multiplicación matricial en un sistema totalmente heterogéneo compuesto por multicore+multiGPU+multiMIC.
- Y por último, adaptar las técnicas a clusters con múltiples nodos multicore+multiGPU+ multiMIC.

En [9] se dan algunas indicaciones de cómo podrían realizarse algunas de estas extensiones, en particular a multicore+multicoprocesadores y a clusters de nodos de este tipo.

Bibliografía

[1] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio M. Vidal. *Introducción a la programación paralela*. Paraninfo Cengage Learning, 2008.

[2] Pedro Alonso, Ravi Reddy, and Alexey L. Lastovetsky. Experimental study of six different implementations of parallel matrix multiplication on heterogeneous computational clusters of multicore processors. In *PDP*, pages 263–270, 2010.

[3] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kauffman, 2001.

[4] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self Adapting Software for Numerical Linear Algebra and LAPACK for Clusters. *Parallel Computing*, 29:1723–1743, 2003.

[5] CUBLAS. `http://docs.nvidia.com/cuda/cublas/`.

[6] CUBLAS-XT. `https://developer.nvidia.com/cublasxt`.

[7] CUDA Zone. `http://www.nvidia.com/cuda`.

[8] J. Cuenca, L. P. García, D. Giménez, and J. Dongarra. Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters. In *Proc. IEEE Int. Conf. on Cluster Computing*. IEEE Computer Society, September 2005.

[9] Javier Cuenca, Luis-Pedro García, Domingo Giménez, and Francisco-José Herrera. Guided installation of basic linear algebra routines in a cluster with manycore components. *Concurrency and Computation: Practice and Experience*, 29(15), 2017.

[10] Javier Cuenca, Domingo Giménez, José González, Jack Dongarra, and Kenneth Roche. Automatic optimisation of parallel linear algebra routines in systems with variable load. In *PDP*, pages 409–416, 2003.

[11] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subroutines. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[12] Luis-Pedro García, Javier Cuenca, Francisco-José Herrera, and Domingo Giménez. On guided installation of basic linear algebra routines in nodes with manycore components. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 114–122, 2016.

[13] Domingo Giménez, Ginés García, Joaquín Cervera, and Norberto Marín. *Algoritmos y estructuras de datos. Volumen II: Algoritmos*. Texto guía Universidad de Murcia, Diego Marín, 2003.

[14] G. Golub and C. F. Van Loan. *Matrix Computations*. The John Hopkins University Press, fourth edition, 2013.

[15] S. Hunold and T. Rauber. Automatic tuning of PDGEMM towards optimal performance. In *11th International Euro-Par Conference, Lecture Notes in Computer Science*, volume 3648, pages 837–846, 2005.

[16] Intel. `https://www.intel.es/content/www/es/es/processors/xeon/xeon-e7-8800-4800-v4-product-families-brief.html`.

[17] Intel MKL web page. `http://software.intel.com/en-us/intel-mkl/`.

[18] Web page of the Scientific Computing and Parallel Programming Group at the University of Murcia. `http://luna.inf.um.es/grupo_investigacion/`.