Análisis del rendimiento y consumo de programas orientados al cálculo

Grado en Ingeniería Informática

Ingeniería de Computadores

Proyecto de Fin de Grado

2016

Autor:

Marcos Ruiz Ramírez

Tutor:

Clemente Rodríguez Lafuente

Agradecimientos

Quisiera empezar a agradecer la realización de este proyecto a mi director, Clemente, por ayudarme a encontrar soluciones, encaminar el proyecto y sobre todo enseñarme cosas completamente nuevas que no había visto durante la carrera.

Gracias a mi familia, por confiar en mí, esforzarse para que tuviera la oportunidad de estudiar la carrera de Ingeniería Informática y apoyarme a lo largo de ella.

Gracias a mi cuadrilla, que aun estudiando diferentes carreras, muchas veces me han ayudado para encontrar la solución a algunos problemas que se me han planteado a lo largo de estos cuatro años.

Gracias a todos mis compañeros de clase, ahora amigos, que he tenido durante estos cuatro años, con los que he compartido buenos y malos momentos, pero que han hecho que la carrera sea mejor.

Y por último, agradecer a la persona que este leyendo esto, por consultar esta memoria y tener interés en este proyecto.

Resumen

En este proyecto se han implementado rutinas vectoriales y/o paralelas en un sistema *multicore* vectorial. Las rutinas pertenecen a un conjunto de funciones de algebra lineal llamada BLAS (*Basic Linear Algebra Subprograms*), únicamente un subconjunto de ellas (BLAS 1). El subconjunto fundamentalmente está acotado por memoria (*Memory Bound*). Este sistema *multicore* funciona bajo tecnologías Intel y se han ejecutado en 3 configuraciones diferentes.

- Intel Core i5 4200H 2.8 GHz (portátil)
    - 2 cores (HyperThreading)
    - AVX2, FMA y TurboBoost

- Intel Core i7 4810MQ 2.8 GHz (portátil)
    - 4 cores (HyperThreading)
    - AVX2, FMA y TurboBoost

- Intel Core 2 Quad CPU Q9550 2.83GHz (sobremesa)
    - 4 cores
    - SSE4.1

A partir de las rutinas vectoriales creadas, se han incluido todas ellas en una librería, estática y dinámica. Esta librería se caracteriza por conocer la arquitectura del ordenador con el que se trabaja, es decir, se trata de una librería consciente del hardware. Ejecuta la mejor opción de rendimiento dependiendo de la máquina y carga computacional.

Una vez implementados todos los códigos se ha realizado un estudio de rendimiento. Este estudio ha permitido caracterizar el rendimiento de las funciones BLAS 1, en función de la complejidad en memoria, cómputo y dependencias de control.

Por último, se ha realizado un estudio de consumo sobre un subconjunto de las funciones BLAS 1, permitiendo observar el impacto energético de las diferentes técnicas de computación. Este estudio ha sido posible mediante la herramienta de medición de consumo *Joulemeter*.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

1 CAPÍTULO: INTRODUCCIÓN

Este proyecto ha sido desarrollado con el objetivo de realizar un análisis de rendimiento y consumo de las funciones y subrutinas de cálculo pertenecientes al primer nivel del BLAS (*Basic Linear Algebra Subprograms*). Para ello, se han tomado como referencia los códigos secuenciales del repositorio Netlib, realizados en el lenguaje de programación Fortran entre los años 1978-1993. En este proyecto se ha construido una librería similar con todas las funciones y subrutinas en lenguaje C.

Se han añadido las técnicas de procesado vectorial y/o paralelo de los datos. Esto reduce el tiempo de espera por parte del usuario. El tipo de procesado es dependiente del tamaño de los datos que el programa utilice.

El proyecto en su totalidad está diseñado para procesadores de la marca estadounidense Intel.

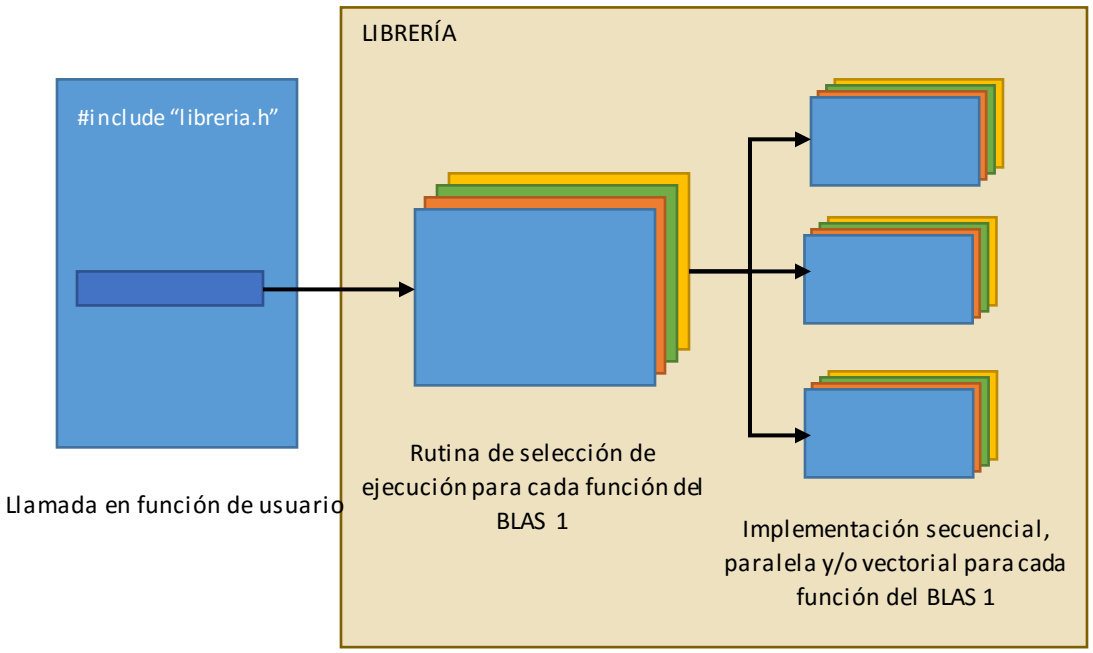


Ilustración 1-1 Representación del objetivo del proyecto

Tal y como se muestra en la ilustración 1-1, se pretende conseguir una librería que pueda ser utilizada por el usuario. Esta librería implementará todas las funciones del primer nivel de BLAS, seleccionado el tipo de ejecución que tiene que realizar dependiendo de la maquina: secuencial, vectorial y/o vectorial. La librería, en el caso del proyecto, es consciente del hardware con el que se trabaja.

1.1 MOTIVACIÓN

Como se explicará más adelante en el apartado 2.1, la vectorización y paralelismo de los datos son dos tecnologías que los procesadores de hoy en día tienen implementadas. Este tipo de técnicas es muy utilizada en diferentes tipos de centros orientados al procesado de datos, pero solo una parte muy reducida de los usuarios las utilizan o saben que existen. Es por ello que la utilización de este tipo de técnicas abriría una gran cantidad de posibilidades para reducir el tiempo de cómputo aprovechando al máximo el procesador con el que se ejecutan.

Además, en la actualidad se tiene muy en cuenta el consumo que genera esa ganancia de tiempo de computo, por lo que lo ideal sería encontrar un punto de equilibrio entre rendimiento y consumo de este tipo de técnicas de procesado, permitiendo así un rendimiento aceptable con un consumo limitado.

En conclusión, la utilización de este tipo de computación ofrece la posibilidad de realizar trabajos complejos y de gran envergadura, en un tiempo de cómputo más reducido. Este trabajo pretende crear una implementación de computación paralela/vectorial a las funciones y rutinas de cálculo del primer nivel del BLAS, teniendo en cuenta su impacto energético.

1.2 ESTRUCTURA DE LA MEMORIA

Este documento está estructurado en varios apartados y subapartados.

La primera parte de la documentación presenta brevemente una serie de conceptos básicos, la explicación del escenario donde se va a elaborar el proyecto y los recursos utilizados. Está formada por el segundo y tercer capítulo. Se presentan varios conceptos básicos sobre la computación paralela: la vectorización y el paralelismo como formas de computación. Una vez realizada esta introducción, se comentan las aplicaciones que hoy en día tiene n ambas técnicas, que como se podrá observar, serán bastantes. Con esta primera parte se pretende ubicar al lector sobre las posibilidades que ofrecen estas formas de cómputo y sobre el posterior análisis que se realizará sobre el primer nivel del BLAS, es decir, se trata de la pieza angular del proyecto, para que pueda entender correctamente el resto de la documentación.

El final de esta primera parte se centra en los recursos y el escenario donde se ha trabajo el proyecto, así como la instalación de todo lo necesario para llevarlo a cabo y sus requisitos.

En la segunda parte del documento, formada por el cuarto, quinto y sexto capítulo, se encuentra todo lo relacionado con el proceso de obtención de datos del rendimiento, pasando por la estructura, organización e implementación del código realizado. Primeramente se indican los modelos de datos que se han utilizado y las auto-limitaciones que se han impuesto. Seguidamente se explica el procedimiento seguido para obtener los datos y poder realizar comparaciones entre los diferentes tipos de cómputos. Por último, a partir de los datos obtenidos se explica el criterio de decisión generado y la elección de ejecución para un determinado tipo de procesador teniendo en cuenta los diferentes recursos de los que dispone.

En la cuarta parte, formada por el octavo capítulo, se analiza el consumo de los programas. En ella se muestra la obtención de datos relacionados con la energía, el análisis de los mismos y varios ejemplos gráficos con los resultados obtenidos.

En la quinta y última parte, formada por el noveno y décimo capítulo, se presentan algunos aspectos de la gestión del PFG, conclusiones del trabajo y varias propuestas de mejora.

Al final del documento puede encontrarse la bibliografía utilizada para obtener información sobre el proyecto realizado.

Además, adjunto a esta memoria, se puede encontrar un documento en el que se reflejan los datos de rendimiento obtenidos en el desarrollo del proyecto. Ese documento se compone de los resultados obtenidos por tres configuraciones diferentes. A parte de los datos, se indican las soluciones secuenciales creadas para cada función del primer nivel del BLAS y en algunos casos alguna solución vectorial o paralela. El documento prácticamente, está compuesto por tablas

con diferentes respuestas. A lo largo de esta memoria, se llamará a este documento por el nombre “Resultados de rendimiento”.

2 CAPÍTULO: CONCEPTOS BÁSICOS

2.1 COMPUTACIÓN PARALELA

2.1.1 Vectorización: Instrucciones SIMD

SIMD (*Single Instruction, Multiple Data*) es un tipo de técnica de computación basada en instrucciones. Ese conjunto de instrucciones son implementadas por algunos procesadores de hoy en día, los cuales internamente contienen diferentes unidades de procesamiento. Una sola instrucción es ejecutada por todas las unidades de procesamiento, dando lugar a paralelismo a nivel de datos.

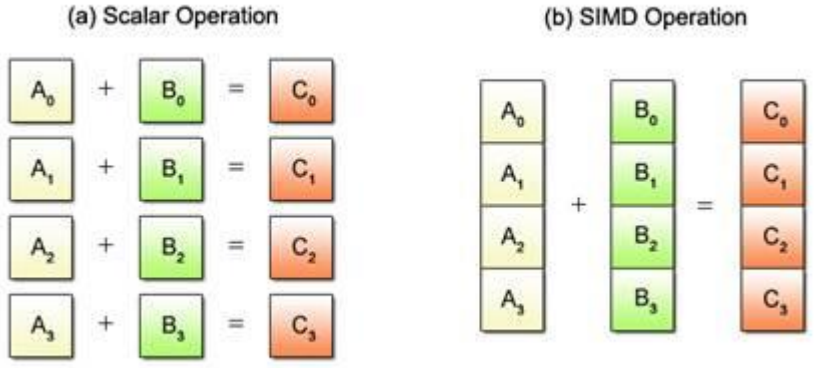


Ilustración 2-1 Ejemplo de comparación entre operaciones escalares y vectoriales (www.kernel.org)

Tal y como se muestra en la Ilustración 2-1, cuando se realiza una misma operación con dos vectores de datos, en el caso de la operación escalar (imagen izquierda), las operaciones se calculan en serie una detrás de otra. En cambio, en el caso vectorial (imagen derecha) la suma y el almacenamiento se realizan en bloques de elementos del vector.

Cabe destacar, que el tiempo en ciclos es diferente para cada situación. En este caso, las operaciones escalares tienen un coste de 4 ciclos, en cambio las operaciones vectoriales solo tienen 1 ciclo de coste.

Este tipo de instrucciones fueron introducidas por Intel. Existen diferentes tipos dependiendo del tipo de dato que se maneje, es decir, los datos que pueden procesar este tipo de instrucciones pueden ser los siguientes:

Tipo	Sufijo	Tamaño
Precisión simple (float)	s	32 bits
Precisión doble (double)	d	64 bits
Entero (int)	i8	8 bits
	i16	16 bits
	i32	32 bits
	i64	64 bits
	i128	128 bits

Tabla 2-1 Tipos de datos soportados por las instrucciones SIMD

Y al mismo tiempo también se pueden diferenciar por la versión:

MMX – SSE – SSE2 – **SSE3 – SSSE3 - SSE4.1 – SSE4.2 – AVX – AVX2 –** AVX3.1 – AVX3.2

Para este proyecto, se han utilizado las que están marcadas: SSE (*Streaming SIMD Extensions*) y AVX (*Advanced Vector Extensions*).

La estructura del conjunto de instrucciones de cada versión vectorial sigue un modelo similar y necesitan variables de diferentes tipos, tabla 2-2.

Conjunto instrucciones SIMD	Formato variable SIMD	Formato y número de elementos de vector	
SSE	_m128	float	4
	_m128d	double	2
	_m128i	int	4
AVX	_m256	float	8
	_m256d	double	4
	_m256i	int	8

Tabla 2-2 Tipos de variables SIMD

Existe un tipo de extensión relacionada con las versiones AVX que optimiza varias operaciones que pueden darse en el código donde se utilicen. Esta extensión tiene como siglas FMA (*Fused Multiply-Add*) y su función es sustituir las instrucciones que realicen una multiplicación seguido de una suma/resta en una única instrucción, siempre que los datos que se utilicen estén en coma flotante.

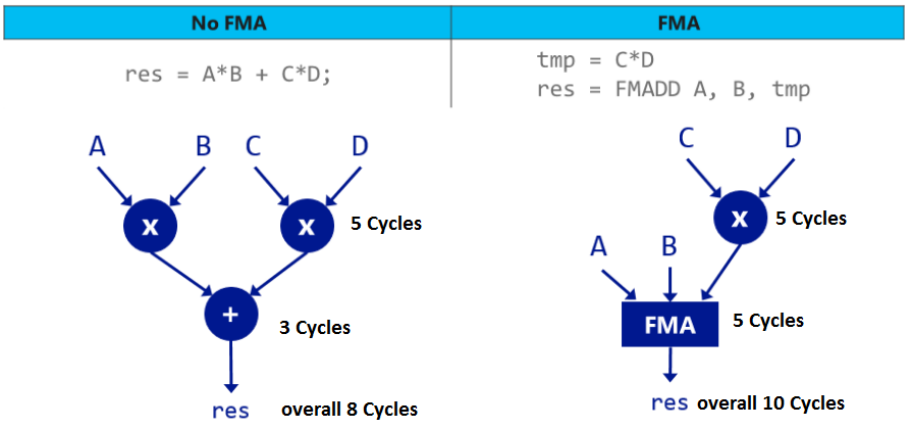


Ilustración 2-2 Representación del uso de la extensión FMA (www.stackoverflow.com)

Como puede observarse en la ilustración 2-2, cuando se utiliza FMA se realiza una operación en lugar de dos, realizando un producto y una suma (resta) en una única instrucción.

Para poder utilizar las instrucciones o intrínsecas que permiten realizar estas operaciones múltiples, es necesario que en el código donde se utilizan se incluya las correspondientes cabeceras:

SSE <xmmintrin.h>

SSE2<pmmintrin.h>

SSE3<pmmintrin.h>

SSE4.1<smmintrin.h>

AVX1/2<immintrin.h>

En estas cabeceras se encuentran todas las operaciones posibles que se puede realizar por medio de una instrucción, separada por cada versión SIMD, siempre que esta sea incluida en el condigo donde aparece. De hecho, también es necesario utilizar diferentes tipos de *flags* a la hora de compilar para que el compilador pueda reconocer el contenido del código.

gcc ***–mavx2 –msse4 –mfma*** *–o fichero.c*

Estas intrínsecas están disponibles en un portal web proporcionado por Intel:

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

En él, se pueden encontrar todas las instrucciones o intrínsecas para cualquier tipo de versión disponible hoy en día junto con una breve explicación y demostración de lo que realiza y necesita para poder utilizarse, ilustración 2-3.

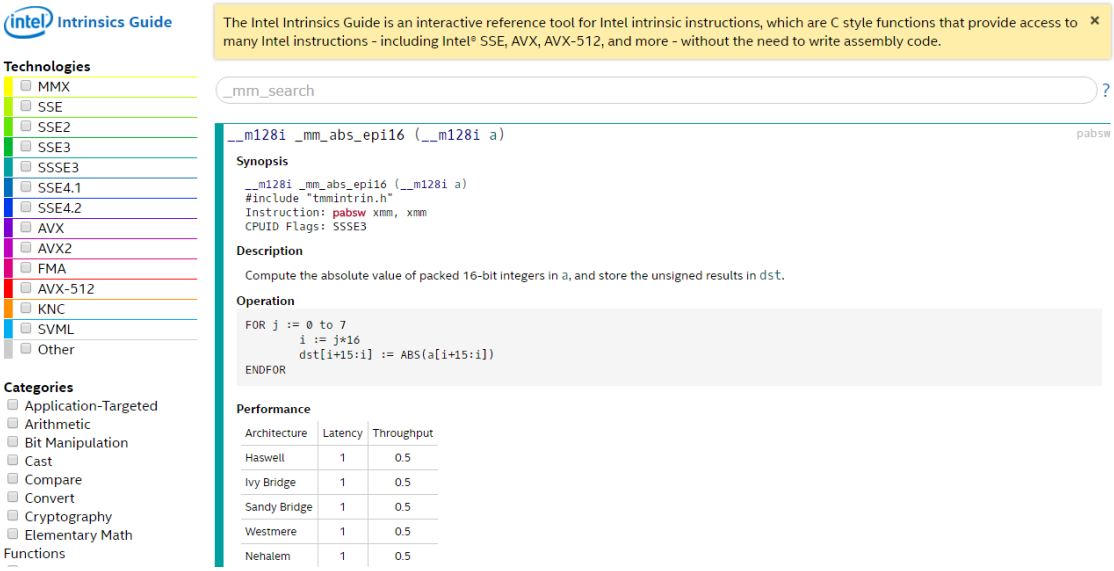


Ilustración 2-3 Guía de intrínsecas de Intel

2.1.1.1 Aplicaciones

Las aplicaciones del procesamiento vectorial en los procesadores Intel están enfocadas principalmente a que realicen cómputo vectorial y además en el desarrollo de terceras tecnologías. Aquí se muestran algunos de los ejemplos en los que más remarca Intel su utilización:

- Videojuegos.
- Procesamiento de video.
- Aplicaciones multimedia en terminales móviles.

Las rutinas BLAS son un caso ideal para su utilización.

2.1.2 Paralelismo y OpenMP

Se trata de una forma de computación que permite realizar cálculos simultáneamente, es decir, divide la carga de trabajo para poder solucionar el problema, siempre teniendo como objetivo obtener más rendimiento frente a la no utilización de paralelismo. Para utilizar este tipo de computación es necesario tener un *multicore* y/o multiprocesador.

En el proyecto, para realizar ese reparto de trabajo se ha utilizado una API (Application Programming Interface) de programación multiproceso de memoria compartida llamada OpenMP. Esta API es utilizada en los lenguajes C, C++ y Fortran. En nuestro caso será utilizada en C, ya que el proyecto ha sido realizado en C en su totalidad.

OpenMP permite añadir concurrencia al código, creando una copia de sí mismo y ejecutando ambas en paralelo por procesadores independientes (modelo *fork-join*). Para el proyecto solo es necesario utilizar este paralelismo en los bucles *for* que se presentan en las diferentes funciones y subrutinas, permitiendo así un reparto de carga entre los procesadores existentes.

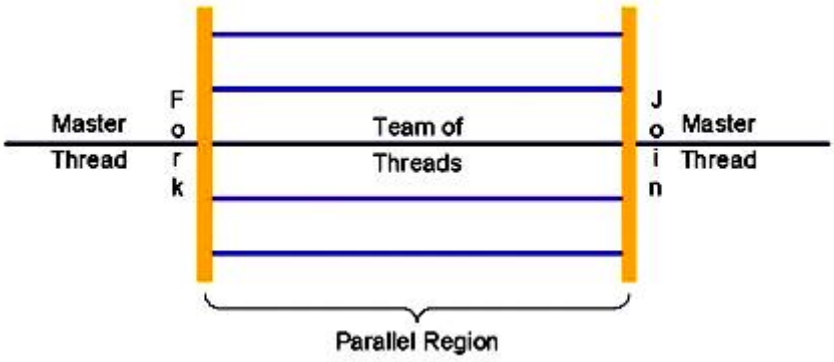


Ilustración 2-4 Representación de ejecución en paralelo

Al igual que en el apartado anterior 2.1.1, OpenMP también requiere de cabeceras y de *flags* de compilación para que pueda ser posible su utilización. En este caso, se simplifica drásticamente ya que solo existe una cabecera llamada *<omp.h>* donde se encuentran todas las operaciones posibles que pueden realizarse y el *flag "–fopenmp"* que permitiría que el compilador pudiese resolver las llamadas a operaciones realizadas en el código a compilar.

Existen varios tipos de operaciones, pero en nuestro caso se han utilizado varios fundamentalmente:

- ***#pragma omp parallel for***: es un tipo de directiva utilizada antes del bucle *for* que indica que lo que este en la siguiente línea, el *for*, tiene que ser dividido entre los diferentes procesadores, creando una copia para ambos e indicando con una sintaxis especifica los diferentes parámetros compartidos o no entre ellos, llamadas clausulas.
    - ***firstprivate():*** es una cláusula que permite crear una variable privada para cada thread inicializándola con el valor original.
    - ***private():*** es una cláusula que permite crear una variable privada para cada thread.
    - ***reduction():*** especifica que una o más variables que son privadas, al final de la región paralela, deberán de realizar una operación de reducción.
- ***omp_set_num_threads():*** es una función que establece el número de subprocesos permitidos.

La utilización del paralelismo no siempre es la mejor opción, ya que en muchos casos dependiendo del tamaño de los datos, es mejor realizar una ejecución secuencial. Esto se debe al tiempo de reparto de carga en las directivas.

Dentro de las arquitecturas de los procesadores se puede encontrar una tecnología llamada *HyperThreading*. Esta tecnología se basa en generar varios procesadores lógicos dentro de un procesador físico, permitiendo emular varios procesadores. El resultado es una mejora de velocidad y rendimiento por parte del procesador, obteniendo como mucho un 60% más de rendimiento, según defiende la propia empresa Intel.

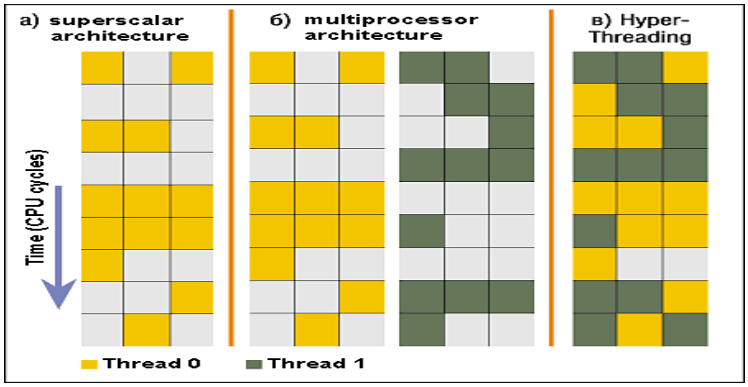


Ilustración 2-5 Comparativa entre procesadores: escalar, multiprocesador y procesador con HyperThreading (www.ixbtlabs.com)

En el capítulo 5 se explicará con más detalle el impacto de esta tecnología frente a los resultados obtenidos y las decisiones tomadas sobre su utilización.

2.1.2.1 Aplicaciones

Las aplicaciones del paralelismo son muchas y diversas. Siempre están relacionadas con cálculos complejos o de mucho coste computacional. Seguidamente se muestra una breve recopilación de ejemplos donde se utiliza este tipo de técnicas:

- Predicción de números atmosféricos.
- Oceanografía y astrofísica.
- Economía financiera.
- Análisis de elementos finitos.
- Inteligencia artificial y automatización.
- Ingeniería genética.
- Aplicaciones médicas.
- Visión por computador.
- Simulaciones matemáticas.

2.2 BLAS

"*Basic Linear Algebra Subprograms*" o BLAS, es una especificación que recoge un conjunto de rutinas de cálculo de bajo nivel. Estas rutinas producen una serie de operaciones, desde modificación de vectores, producto escalar, combinaciones lineales y multiplicación de matrices. Inicialmente fueron pensadas para ser recogidas en librerías algebraicas, que posteriormente

serían utilizadas por los lenguajes C y Fortran, tal y como se muestra en el capítulo 7 del documento.

Las rutinas BLAS en algunos casos suelen estar optimizadas para una determinada máquina, obteniendo de esa forma un rendimiento mayor. Este rendimiento se consigue a través del aprovechamiento del hardware de la maquina así como los registros de vectores y las instrucciones SIMD.

La primera librería creada, se remonta al año 1979, completamente desarrollada en Fortran y accesible públicamente gracias al portal web proporcionado por Netlib. Aunque fue la primera librería no se optimizó su rendimiento, por lo que todas sus rutinas están implementadas para una ejecución secuencial.

Esta librería ha sido incluida en muchas herramientas de software relacionadas con la computación de álgebra lineal, por ejemplo: Mathematica, MATLAB, GNU Octave, LAPACK, etc. En algunos de estos casos existen librerías con optimizaciones para el hardware.

Las rutinas BLAS se dividen en tres niveles de dificultad computacional, que también corresponden cronológicamente con el orden de creación y de publicación de las mismas.

- El primer nivel de BLAS o *"level 1"* contiene rutinas de orden O(n). Fue el primer nivel descrito el año 1979 y se compone de operaciones sobre vectores con desplazamiento (stride).
    - Producto escalar.
    - Norma vectorial o norma euclídea.
    - Modificación de valores.

```
                dim scalar vector   vector   scalars             5-element prefixes
                                                                 array

SUBROUTINE _ROTG (                                    A, B, C, S )          S, D
SUBROUTINE _ROTMG(                                D1, D2, A, B,         PARAM )  S, D
SUBROUTINE _ROT  ( N,         X, INCX, Y, INCY,                 C, S )          S, D
SUBROUTINE _ROTM ( N,         X, INCX, Y, INCY,                         PARAM )  S, D
SUBROUTINE _SWAP ( N,         X, INCX, Y, INCY )                                 S, D, C, Z
SUBROUTINE _SCAL ( N,  ALPHA, X, INCX )                                          S, D, C, Z, CS, ZD
SUBROUTINE _COPY ( N,         X, INCX, Y, INCY )                                 S, D, C, Z
SUBROUTINE _AXPY ( N,  ALPHA, X, INCX, Y, INCY )                                 S, D, C, Z
FUNCTION   _DOT  ( N,         X, INCX, Y, INCY )                                 S, D, DS
FUNCTION   _DOTU ( N,         X, INCX, Y, INCY )                                 C, Z
FUNCTION   _DOTC ( N,         X, INCX, Y, INCY )                                 C, Z
FUNCTION   __DOT ( N,  ALPHA, X, INCX, Y, INCY )                                 SDS
FUNCTION   _NRM2 ( N,         X, INCX )                                          S, D, SC, DZ
FUNCTION   _ASUM ( N,         X, INCX )                                          S, D, SC, DZ
FUNCTION   I_AMAX( N,         X, INCX )                                          S, D, C, Z
```

Ilustración 2-6 Rutinas de primer nivel de BLAS (Netlib)

- El segundo nivel de BLAS *o "level 2"* contiene rutinas de orden $O(n^2)$. Este nivel se desarrolló entre los años 1984-1988. Se compone principalmente de operaciones matriz-vector y aprovecha algunas de las operaciones descritas en primer nivel de BLAS.
    - Modificación de valores.
    - Multiplicación matriz-vector o vector-matriz.
    - Operación en matrices triangulares, simétricas, banda, dispersa, …

```
        options            dim   b-width scalar matrix  vector   scalar vector   prefixes

_GEMV (        TRANS,      M, N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) S, D, C, Z
_GBMV (        TRANS,      M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) S, D, C, Z
_HEMV ( UPLO,                 N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) C, Z
_HBMV ( UPLO,                 N, K,      ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) C, Z
_HPMV ( UPLO,                 N,         ALPHA, AP,     X, INCX, BETA,  Y, INCY ) C, Z
_SYMV ( UPLO,                 N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) S, D
_SBMV ( UPLO,                 N, K,      ALPHA, A, LDA, X, INCX, BETA,  Y, INCY ) S, D
_SPMV ( UPLO,                 N,         ALPHA, AP,     X, INCX, BETA,  Y, INCY ) S, D
_TRMV ( UPLO, TRANS, DIAG,    N,                A, LDA, X, INCX )                 S, D, C, Z
_TBMV ( UPLO, TRANS, DIAG,    N, K,             A, LDA, X, INCX )                 S, D, C, Z
_TPMV ( UPLO, TRANS, DIAG,    N,                AP,     X, INCX )                 S, D, C, Z
_TRSV ( UPLO, TRANS, DIAG,    N,                A, LDA, X, INCX )                 S, D, C, Z
_TBSV ( UPLO, TRANS, DIAG,    N, K,             A, LDA, X, INCX )                 S, D, C, Z
_TPSV ( UPLO, TRANS, DIAG,    N,                AP,     X, INCX )                 S, D, C, Z

        options            dim   scalar vector   vector   matrix  prefixes

_GER  (                    M, N, ALPHA, X, INCX, Y, INCY, A, LDA ) S, D
_GERU (                    M, N, ALPHA, X, INCX, Y, INCY, A, LDA ) C, Z
_GERC (                    M, N, ALPHA, X, INCX, Y, INCY, A, LDA ) C, Z
_HER  ( UPLO,                 N, ALPHA, X, INCX,          A, LDA ) C, Z
_HPR  ( UPLO,                 N, ALPHA, X, INCX,          AP )     C, Z
_HER2 ( UPLO,                 N, ALPHA, X, INCX, Y, INCY, A, LDA ) C, Z
_HPR2 ( UPLO,                 N, ALPHA, X, INCX, Y, INCY, AP )     C, Z
_SYR  ( UPLO,                 N, ALPHA, X, INCX,          A, LDA ) S, D
_SPR  ( UPLO,                 N, ALPHA, X, INCX,          AP )     S, D
_SYR2 ( UPLO,                 N, ALPHA, X, INCX, Y, INCY, A, LDA ) S, D
_SPR2 ( UPLO,                 N, ALPHA, X, INCX, Y, INCY, AP )     S, D
```

Ilustración 2-7 Rutinas de segundo nivel de BLAS (Netlib)

- El tercer nivel de BLAS o *"level 3"* contiene rutinas de orden $O(n^3)$. Este nivel se publicó en el año 1990. Se compone de operaciones matriz-matriz y aprovecha algunas de las funciones descritas en el segundo nivel de BLAS.
    - Multiplicación de matrices.
    - Transposición de matrices.

```
       options                        dim      scalar matrix  matrix  scalar matrix  prefixes

_GEMM (             TRANSA, TRANSB,      M, N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) S, D, C, Z
_SYMM ( SIDE, UPLO,                      M, N,    ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) S, D, C, Z
_HEMM ( SIDE, UPLO,                      M, N,    ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) C, Z
_SYRK (       UPLO, TRANS,                  N, K, ALPHA, A, LDA,         BETA,  C, LDC ) S, D, C, Z
_HERK (       UPLO, TRANS,                  N, K, ALPHA, A, LDA,         BETA,  C, LDC ) C, Z
_SYR2K(       UPLO, TRANS,                  N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) S, D, C, Z
_HER2K(       UPLO, TRANS,                  N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC ) C, Z
_TRMM ( SIDE, UPLO, TRANSA,        DIAG, M, N,    ALPHA, A, LDA, B, LDB )                S, D, C, Z
_TRSM ( SIDE, UPLO, TRANSA,        DIAG, M, N,    ALPHA, A, LDA, B, LDB )                S, D, C, Z
```

Ilustración 2-8 Rutinas de tercer nivel de BLAS (Netlib)

Este proyecto únicamente abarcará el primer nivel de BLAS sin utilizar desplazamientos entre elementos de vector, es decir, *"stride = 1"*.

Los diferentes niveles de BLAS se diferencian en relación a si están acotadas por memoria o por cómputo. El primer nivel de BLAS está acotado por memoria (*Memory Bound*), ya que se trata de un coste de O(kn) en memoria y no se reutilizan los datos. Por ello, la parte vectorial obtendrá mejor rendimiento en este nivel. A su vez, este nivel tiene un coste de computo que se encuentra entre O(0) y O(6n).

Los siguientes dos niveles de BLAS están acotados por computo (*CPU Bound*). Como se trata de matrices, se reutilizan más datos cuanto mayor es el nivel de BLAS, por lo que la parte paralela obtendría en algunos casos mejor rendimiento que la parte vectorial. Ambos niveles tienen un

coste de $O(n^2)$ en memoria, pero en cuanto al coste computacional el segundo nivel tiene $O(n^2)$ y el tercero $O(n^3)$.

Cada subrutina o función de cada nivel tiene una versión para diferentes tipos de datos de entrada y de salida. En la siguiente tabla se muestran los tipos de datos que pueden utilizar. No todas las rutinas se han implementado para todos los tipos de datos en este proyecto.

Tipo	Sigla	Descripción
float	S	Precisión simple
double	D	Precisión doble
complex	C	Número complejo
double complex	Z	Número complejo doble

Tabla 2-3 Tipos de datos implementados por la librería BLAS

En el proyecto no se han realizado implementaciones sobre números complejos debido a que las instrucciones SIMD no resuelven ese tipo de datos. En cambio, se ha añadido un nuevo tipo de dato que no se contempla en el primer nivel. Ese tipo es el entero, "int". Las instrucciones SIMD tienen operaciones que permiten operar con números enteros, por lo que se decidió añadirlo al conjunto de rutinas que permitían su utilización.

3 CAPÍTULO: ESCENARIO

Para la realización de este proyecto se ha necesitado de varias herramientas fundamentales para su elaboración.

De cara a obtener datos distintos, para poder realmente ver el comportamiento de los resultados, se ha decidido utilizar tres procesadores. Para poder encontrar diferencias notables se han utilizado las siguientes configuraciones:

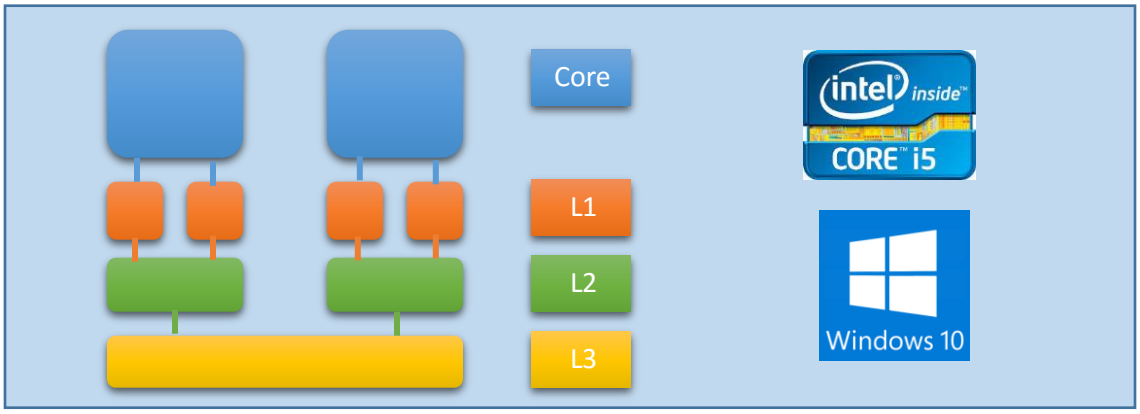


Ilustración 3-1 Configuración con Intel Core i5

Intel Core i5 4200H 2.8 GHz (portátil)

- 2 cores (HyperThreading)
- AVX2, FMA y TurboBoost
- Cache L1 = 2x32 KBytes
- Cache L2 = 2x256 KBytes
- Cache L3 = 3 MBytes
- GCC 4.8.1
- Windows 10

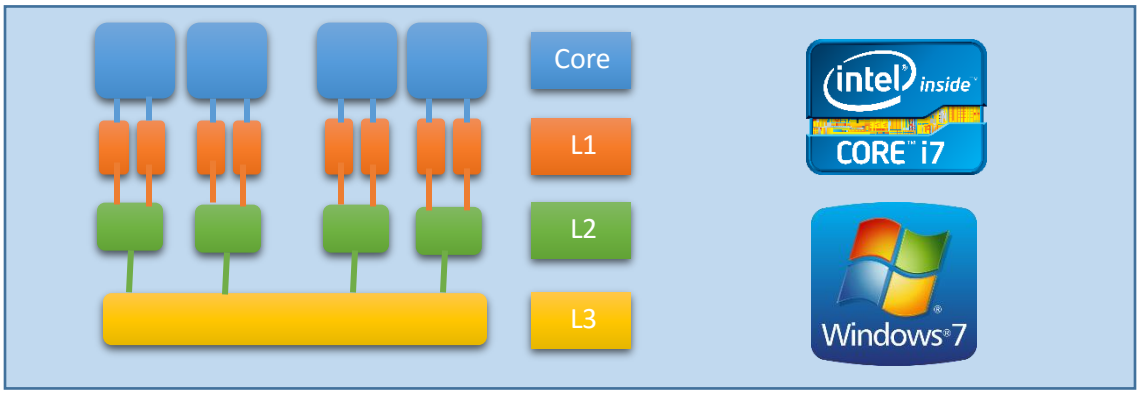


Ilustración 3-2 Configuración con Intel Core i7

Intel Core i7 4810MQ 2.8 GHz (portátil)

- 4 cores (HyperThreading)
- AVX2, FMA y TurboBoost
- Cache L1 = 4x32 KBytes
- Cache L2 = 4x256 KBytes
- Cache L3 = 6 MBytes
- GCC 4.8.1
- Windows 7

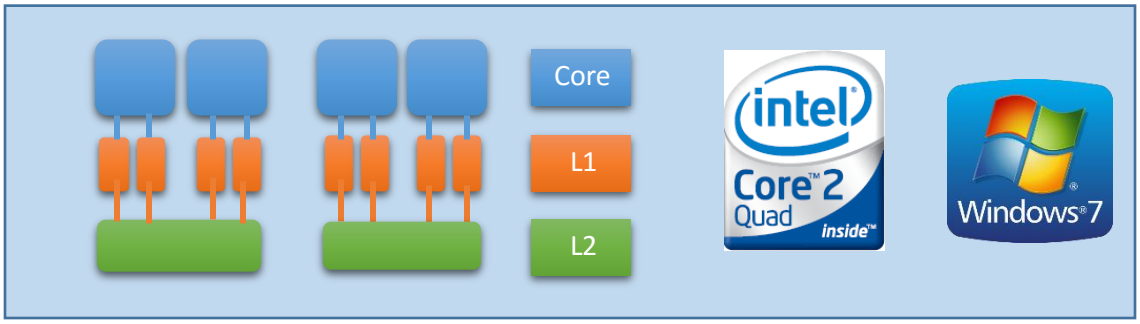


Ilustración 3-3 Configuración con Intel Core Quad

Intel Core 2 Quad CPU Q9550 2.83GHz (sobremesa)

- 4 cores
- SSE4.1
- Cache L1 = 8x32 KBytes
- Cache L2 = 2x6 MBytes
- GCC 4.4.0
- Windows 7

Las dos primeras configuraciones, que corresponden con la ilustración 3-1 y 3-2 tienen todas las tecnologías de procesado vectorial necesarias para la realización del proyecto en su máximo aprovechamiento. Cabe destacar que la segunda configuración tiene el doble de *cores* respecto a la primera. En cambio la configuración de la ilustración 3-3 es más antigua y carece de algunas de ellas, concretamente de las instrucciones AVX y AVX2. Por lo que de esta manera puede observar dos configuraciones parecidas, frente a otra diferente.

Los valores de almacenamiento disponible en cada nivel de caché de cada configuración son muy diferente en cada caso. La primera configuración es la que tiene menos memoria para poder almacenar elementos de vector de entrada, en cambio como tiene hasta tres niveles de caché su acceso es más rápido. La segunda configuración dispone de más memoria que la primera y también tiene tres niveles de caché. La última configuración es la que dispone de más memoria, pero solo tiene dos niveles de caché.

Las pruebas iniciales e implementación en paralelo y vectorial, se han realizado sobre la primera configuración. Se han verificado todas las funciones en los tres sistemas, véase el documento adjunto "Resultados de rendimiento".

3.1 INSTALACIÓN

Antes de comenzar con el proyecto, es necesario instalar las herramientas necesarias para poder llevarlo a cabo. La primera herramienta que se necesitará es el compilador GCC (*GNU Compiler Collection*) obtenible mediante el software *MinGW*.

GCC engloba un conjunto de compiladores desarrollados por GNU (*GNU's Not Unix*). Estos compiladores de software libre son predeterminados en los sistemas Linux y permiten la compilación de lenguajes como C/C++, Fortran y Java. *MinGW* es una implementación de los compiladores GCC para plataforma win32, es decir, que permite su uso en entornos Windows.

Como se trata de software libre, se puede proceder a su descargar desde el portal web habilitado para ello.

https://sourceforge.net/projects/mingw/files/

Una vez realizada la instalación es necesario introducir en el *Path* del sistema la ruta de acceso al compilador, para que cuando se realicen compilaciones pueda resolverlas. Para ello es necesario acceder a las propiedades de "Este equipo" o "Mi PC" dependiendo de la versión de Windows con la que se esté trabajando. Una vez abierta la ventana acceder a la configuración avanzada y pulsar en el menú de variables de entorno. Por último editar la variable *Path* que se muestre indicando el directorio donde se encuentre instalado *MinGW*. Un ejemplo sería el siguiente:

C:\Program Files\mingw\bin\

Una vez hecho, se ha de comprobar mediante la aplicación *MinGW Installation Manager* si se tienen todos los paquetes de GCC instalados, ilustración 3-4.

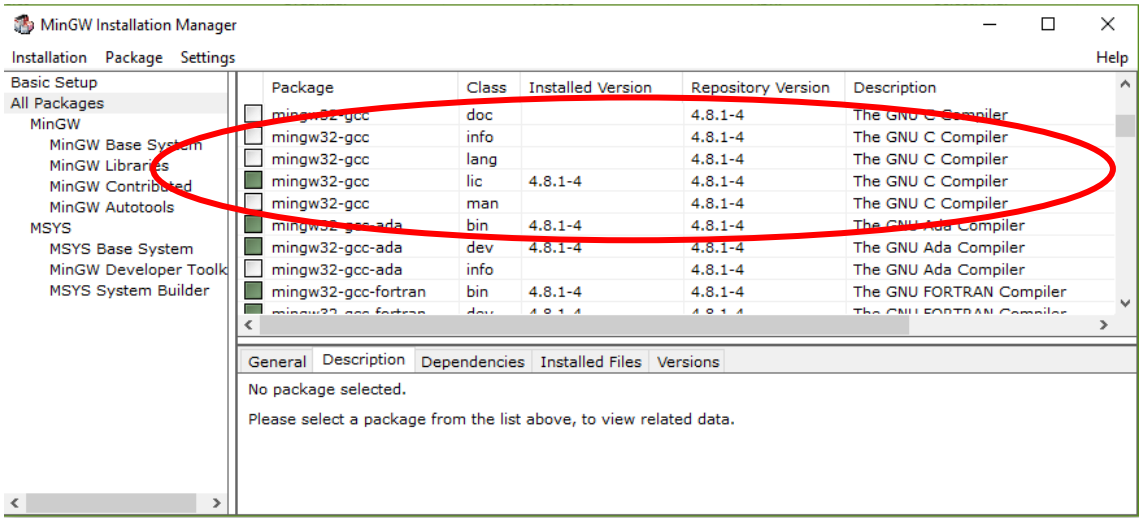


Ilustración 3-4 Software MinGW con los paquetes necesarios

En el caso de que no esté correctamente instalado, se ha de seleccionar los paquetes deseados e instalarlos mediante la pestaña de "*Installation*" y "*Apply Changes*". Cuando se complete esa comprobación, ya sería posible realizar completamente la segunda y tercera parte del proyecto formada por el cuarto, quinto, sexto y séptimo capítulo.

Ilustración 3-5 Icono del compilador GCC (gcc.gnu.org)

Por último para realizar la última de las partes relacionada con el consumo de las ejecuciones es necesario instalar un software que permite realizar las mediciones de consumo. Ese software se llama *Joulemeter*. Se trata de una herramienta creada por Microsoft que genera una tabla del consumo energético de una ejecución en un tiempo determinado manualmente. Mediante esta

herramienta se realizan las mediciones de los diferentes tipos de procesamientos de datos (paralelo, vectorial y secuencial).

Como también se trata de software libre, se puede proceder a su descarga desde el portal web disponible.

http://descargar.cnet.com/Microsoft-Joulemeter/3000-2086_4-75578519.html

Ilustración 3-6 Icono de Joulemeter

Una vez instalado el software, es necesario ponerlo en marcha y calibrarlo. Para ello si se está haciendo una calibración desde un ordenador portátil, es necesario tenerlo cargado (por encima del 50%) sin el cable de alimentación para realizarla. En el caso de que se esté trabajando sobre un ordenador sobremesa, es necesario un dispositivo externo para poder realizar la medición de consumo.

Pueden aparecer errores si se utiliza en el sistema Windows 10, para ello es necesario ejecutarlo con compatibilidad de Windows 7, acceder a las propiedades del ejecutable mediante el *click* derecho e indicar en la pestaña de "Compatibilidad" que se ejecutará en modo de compatibilidad para Windows 7, ilustración 3-7.

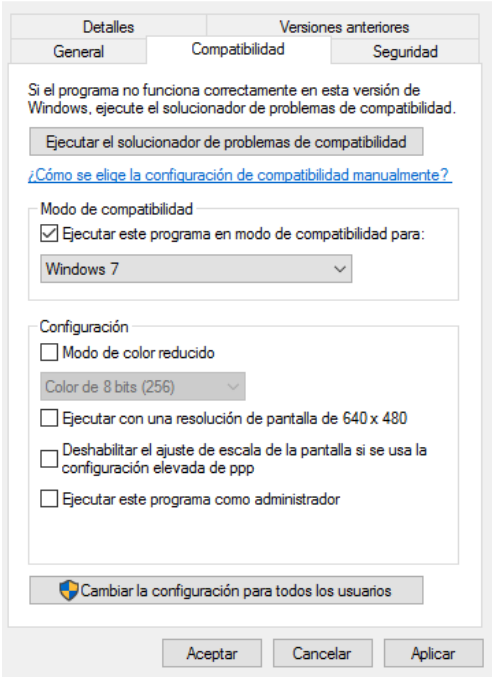


Ilustración 3-7 Compatibilidad Windows 7 en Joulemeter

En el capítulo 8, dedicado al consumo, se explicará detenidamente el funcionamiento de esta herramienta y los resultados obtenidos mediante su utilización.

4 CAPÍTULO: MODELOS DE DATOS

Antes de comenzar a explicar las estructuras o modelos de datos generados para replicar el comportamiento secuencial de los programas de cálculo, concretamente las funciones y subrutinas del primer nivel del BLAS, hay que añadir que todas las funciones y subrutinas utilizadas se han desarrollado a partir de las ya implementadas en la página web de Netlib.

http://www.netlib.org/blas/#_blas_routines

Estos programas estaban desarrollados en Fortran entre los años 1978-1993, por lo que en este proyecto se han migrado al lenguaje C y se ha añadido procesamiento en paralelo y vectorial en aquellos casos en los que era posible.

4.1 SECUENCIAL

La parte secuencial de las funciones y subrutinas aparentemente no se ha modificado, es decir, se ha reescrito en el lenguaje C y en algunos casos se ha aprovechado de funciones existentes en la librería <math.h> para simplificar el código.

Únicamente una de ellas ha sido modificada para poder aplicarle técnicas de programación paralela. Esta función es nrm2, norma vectorial de grado dos o norma euclídea. La solución que proponía la página web de Netlib, no era paralelizable ni vectorizable debido a dependencias entre variables, en cambio sí que era una optimización del propio algoritmo base, ilustración 4-1.

```
static double dnrm2_seq(int n, double *x)
{
   int i;
   double scale,ssq,absol,norm;

   if(n <= 1) norm = 0.0;
   else if(n == 1) norm = fabs(x[0]);
   else{
      scale = 0.0;
      ssq = 1.0;
      for(i=0;i<n;i++){
         if(x[i] != 0){
            absol = fabs(x[i]);
            if(scale < fabs(x[i])){
               ssq = 1+ssq*pow(scale/absol,2);
               scale = absol;
            }else{
               ssq = ssq + pow(absol/scale,2);
            }
         }
      }
      norm = scale*sqrt(ssq);
   }
   return norm;
}
```

No es paralelizable, ni vectorizable, debido a la variable *scale* y los cambios que realiza entre iteraciones, dependencias de datos. Por ello, no es posible realizar operaciones sin influir al resultado final.

Ilustración 4-1 Función nrm2 de Netlib en C

El algoritmo original de la norma euclídea explicado matemáticamente define que para un vector compuesto por $\vec{x} = (x_1, x_2, \ldots, x_n)$ su norma es $||\vec{x}||_2 = \sqrt{|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2}$.

En nuestro caso se ha realizado una leve modificación, debido a que si el vector contiene números grandes o pequeños, el cálculo del cuadrado de esos números producir *overflow* o *underflow* respectivamente en la variable que almacena el resultado, generando así resultados erróneos. Por ellos, se decidió obtener el máximo valor absoluto del vector y dividírselo a cada coeficiente de la raíz $||\vec{x}||_2 = \sqrt{|\frac{x_1}{x_{max}}|^2 + |\frac{x_2}{x_{max}}|^2 + \cdots + |\frac{x_n}{x_{max}}|^2}$. De esta forma no se genera *overflow*/*underflow* por parte de la multiplicación de los datos y hace posible su implementación en paralelo y en vectorial. Al final del proceso se multiplica por el máximo obtenido para que el resultado sea correcto.

```
static double dnrm2_seq2(int n, double *x)
{
   int ind, i;
   double norm = 0.0;
   double max = 0.0;
   double sum = 0.0;
   double mult;

   max = fabsf(x[0]);
   for(i=1;i<n;i++)
      if(fabs(x[i]) > max)
         max = fabs(x[i]);

   mult = (double)1/max;

   for(i=0;i<n;i++)
      sum = sum + (x[i]*mult)*(x[i]*mult);

   norm = sqrt((double)sum);
   return norm * max;
}
```

Ilustración 4-2 Función nrm2 propuesta

En la ilustración 4-2, se muestra el algoritmo y el código seguido para realizar las operaciones paralelas y vectoriales de la norma euclídea. El código que se muestra tiene un rendimiento en secuencial menor al propuesto por BLAS, pero va a poder ser paralelizable y vectorizable.

A pesar de tomar medidas, los datos requieren de precisión en la parte decimal y eso no se consigue. Esto tiene que ver con la aritmética finita. La ejecución del código en secuencial y en paralelo, mediante directivas OpenMP, obtienen el mismo resultado porque este dato se almacena en un solo acumulador. En cambio al utilizar la parte vectorial del procesador realizando la misma operación tiene un error de precisión en la parte decimal, debido a que utiliza tantos acumuladores como elementos tienen sus variables SIMD, tabla 2-2. Por ello se ha tomado en cuenta ese caso y se ha permitido un error menor que 0,001 respecto al secuencial.

```
...
if(fabsf(nrm2 - nrm2_avx)> (nrm2 * 0.001)){
      printf("Valor correcto %f - %f Valor incorrecto \n", nrm2, nrm2_avx);
      exito = 0;;
}
```

Ilustración 4-3 Error de precisión permitido

Este error de precisión de aritmética finita, también puede ocasionarse por la utilización de diferente unidad de coma flotante (FPU) del procesador, es decir, la parte vectorial realiza cálculos con una FPU diferente a la parte paralela, almacenando así, menor número de bits sobre la operación realizada y en consecuencia perdiendo precisión.

En conclusión, se han introducido dos tipos de procesamientos secuenciales, pero solo el algoritmo propuesto ha sido modificable en el momento de realizar códigos en paralelo y en vectorial.

4.2 VECTORIAL

Como se ha mencionado en el capítulo 2, en este proyecto se ha aprovechado de las instrucciones SIMD que los procesadores de hoy en día tienen incluidas. Esto permite realizar operaciones sobre vectores más rápido que secuencialmente.

El uso de las intrínsecas de Intel antes mencionadas son las que permiten que el procesador entienda el tipo de operaciones que se pretenden realizar, pero en muchas de ellas es necesario que los vectores con los que se trabaje, estén alineados en memoria para realizar los accesos más rápido. A continuación, se explicará el proceso seguido para conseguir esta alineación en el procesado de los datos vectorialmente.

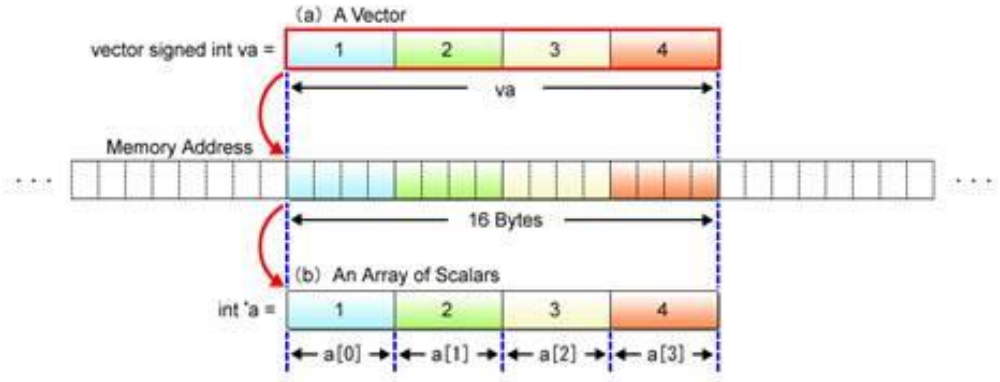


Ilustración 4-4 Recogida de datos alineada SIMD (www.kernel.org)

En todas las funciones o subrutinas en los que se ha podido realizar un replica utilizando las instrucciones SIMD existen dos ficheros fundamentales, uno en SSE y otro en AVX. Adicionalmente en los casos en los que se realice una multiplicación y una suma/resta consecutiva, existe otro en FMA.

4.2.1 Alineación de vector

Algunas instrucciones SIMD requieren que los vectores con los que trabajen estén alineados. Como no siempre es posible tomar el vector alineado, es necesario extraer la parte del vector que si lo está. Esta información alineada está contenida en bloques de memoria de 64 bytes. Por tanto, se ha creado una estructura que permite obtener que parte del vector se ha de tomar para que este alineado en memoria. La estructura es la siguiente:

```
int dir1 = (int) (&y[0]);
int nprol1 = 16 - ((dir1>>2)%16);
int nvec1 = (((int) floor((double)((n-nprol1)/4.0)))<<2)+ nprol1;

int dir2 = (int) (&x[0]);
int nprol2 = 16 - ((dir2>>2)%16);
```

```
if(nprol1 == 16) nprol1 = 0;
if(nprol2 == 16) nprol2 = 0;
```

Ilustración 4-5 Estructura de identificación de la parte alineada de un vector

Para comprender completamente la estructura es necesario describir que es lo que sucede en cada paso.

- Se obtiene la dirección del vector que se pretende alinear mediante la instrucción *int dir1 = (int) (&y[0]);* Se realiza el *cast* en número entero porque la dirección de memoria del vector tiene que almacenarse en enteros.

- Se calcula la variable *nprol* mediante la instrucción *int nprol1 = 16 - ((dir1>>2)%16);* Esa variable contiene el índice del primer elemento del vector alineado. Para ello dependiendo del tipo de dato que se esté utilizando realizará una determinada operación de desplazamiento en la dirección de memoria del vector.

  Puede observarse el comportamiento dependiendo del tipo de dato utilizado.

  - Precisión simple (float/int): El tamaño de los datos del vector en estos dos tipos es 4 bytes. Como se tiene un tamaño de bloque igual a 64 bytes se divide ese tamaño entre 4 y se obtiene un espacio de 16 posiciones de 4 bytes.

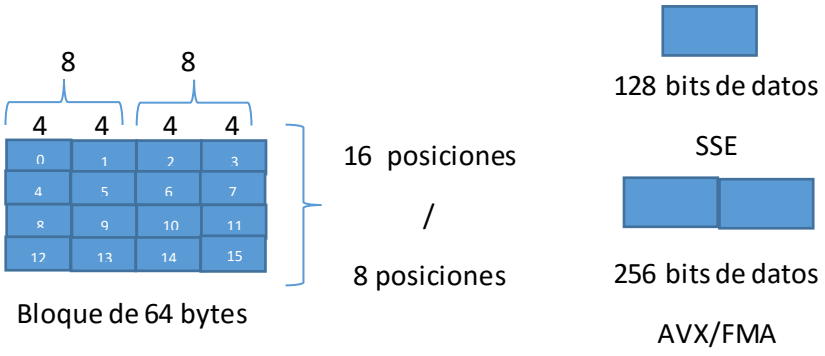


Ilustración 4-6 Representación división de bloque de memoria

Por lo tanto es necesario saber en qué parte de las 16 se ha de empezar para que el vector este alineado. Para realizar ese cálculo, primero se desplaza la dirección de memoria del vector dos veces a la derecha (igual que dividirla por 4), así se obtendría direccionamiento a palabra. A partir de ese dato se calcula su resto con 16, ya que es el tamaño de cada división obtenida, y se le resta de nuevo a 16, para saber en cuál de ellas se ha de empezar.

Cuando *nprol* sea igual a 16 equivaldría al valor 0 del siguiente bloque de 64 bytes, es por ello que se debe de añadir *if(nprol1 == 16) nprol1 = 0;*

  - Precisión doble (double): El tamaño de los datos del vector en este tipo es 8 bytes. Se realiza la misma operación del caso de precisión simple, sustituyendo el valor 16 por 8, ya que tiene un tamaño de bloque igual a 64 bytes se divide ese tamaño entre 8 y se obtiene un espacio de 8 posiciones de 8 bytes.

- Se calcula la variable nvec1 mediante *int nvec1 = (((int) floor((double)((n-nprol1)/4.0)))<<2)+ nprol1;* Esta variable determina el índice del último elemento del vector alineado. Para calcularlo es necesario restarle a todos los elementos del vector la variable *nprol* antes calculada y su resultado dividirlo por el número de elemento que pueda tomar cada versión vectorial. SSE toma 4 elementos float/int y 2 elementos double, en cambio AVX toma 8 elementos float/int y 4 elementos double. Por último a esa división se le aplica un desplazamiento hacia la izquierda (multiplicación), dependiendo del número de elementos tomados expresado en número de bits y se le suma la variable *nprol* antes calculada.

En el caso de que se esté trabajando con dos vectores, solamente uno de ellos puede ser alineado de la forma antes mencionada. En cambio sí se calcula cual es el inicio del segundo vector para que este esté alineado, calculando su nprol2, si coincide con el primer elemento del primer vector, también estará alineado. En caso contrario, este tendrá que ser alineado mediante las instrucciones SIMD de carga (load) y escritura de datos (store). Estas instrucciones contienen la letra "u" en su sintaxis, indicando que alineará el vector. La razón por la que se utiliza lo menos posible, es por su carga computacional.

Instrucciones de carga de datos:

- SSE precisión simple: _mm_loadu_ps()
- SSE precisión doble: _mm_loadu_pd()
- SSE entero: _mm_loadu_si128()
- AVX precisión simple: _mm256_loadu_ps()
- AVX precisión doble: _mm256_loadu_pd()
- AVX entero: _mm256_loadu_si256()

Instrucciones de escritura de datos:

- SSE precisión simple: _mm_storeu_ps()
- SSE precisión doble: _mm_storeu_pd()
- SSE entero: _mm_storeu_si128()
- AVX precisión simple: _mm256_storeu_ps()
- AVX precisión doble: _mm256_storeu_pd()
- AVX entero: _mm256_storeu_si256()

En el proyecto se han realizado dos funciones similares para cada caso, ilustración 4-7, teniendo como diferencia la alineación de los vectores, es decir, si ambos están alineados o solo uno lo está. Como los dos vectores tienen el mismo número de elementos, si los dos comienzan con el mismo elemento de vector alineado, ambos estarán alineados.

```
if(nprol1 == nprol2)
    function_con_vectores_alineados()
  else
    function_con_vectores_no_alineados()
}
```

Ilustración 4-7 Estructura de decisión de ejecución

4.2.2 Estructura de vectorización

Una vez que la función de alineado se completa, se llama a una de las dos funciones antes mencionadas, a la que está alineada o a la que no lo está (debido a uno de los vectores). Independientemente de que función sea, las estructuras son iguales y a continuación se explicarán cuales son.

El código de las funciones escritas con instrucciones SIMD sigue un patrón. Este patrón se basa en una división de código y a su vez en una división de ejecución, es decir, cada parte del código es ejecutada de un modo diferente, siempre que no se utilice código vectorial, realizando la misma operación que la ejecución secuencial del algoritmo general.

Para realizar esta división es necesario que la función tenga como parámetros los elementos antes mencionados: *nprol* y *nvec*; y la operación a realizar este dentro de un bucle. La división se puede observar en la ilustración 4-8.

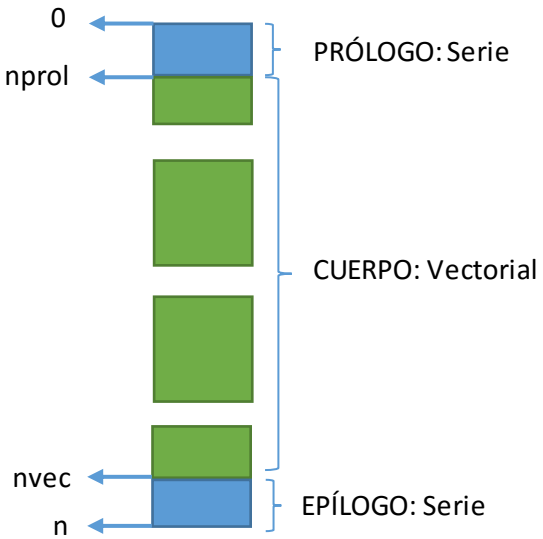


Ilustración 4-8 Estructura de código vectorial

Como se muestra en la imagen la división está compuesta de 3 partes: prologo, cuerpo y epilogo. Los cuadrados que se muestran son los bloques de memoria de 64 bytes y los colores permiten visualizar mejor que tipo de ejecución se realiza en cada parte, el color verde indica ejecución vectorial utilizando instrucciones SIMD y el color azul indica ejecución secuencial.

- **Prólogo:** Es la primera sección del código. Recorre el bucle secuencialmente simulando la operación en serie, entre el primer elemento del vector y el primer elemento alineado del vector (*nprol*).
- **Cuerpo:** Es la segunda sección del código y la parte más destacada de la función ya que es la que contiene las instrucciones SIMD. Su rango de elementos está determinado entre los valores *nprol* y *nvec* calculados en la alineación del vector. Debido a las instrucciones vectoriales el bucle avanza más rápido, ya que una instrucción vectorial trabaja con múltiples datos.
- **Epílogo:** Es la tercera y última sección del código. Recorre el bucle secuencialmente simulando la operación en serie, entre el último elemento alineado del vector (*nvec*) y el último elemento del vector (*n*).

4.2.3 Ejemplo visual

Para poder comprender mejor la estructura de operación antes mencionada, a continuación se mostrará la implementación de uno de las funciones a realizar del primer nivel de BLAS que tiene mayor complejidad a nivel de instrucciones. La función es si_amax() y está realizada con instrucciones SSE. Esta función, como resultado, devuelve el primer índice del máximo elemento en valor absoluto del vector, que se le pasa por parámetro.

```
static int alig_si_amax_sse(int n, float *x, int nprol, int nvec1)
{
   float max = 0.0;
   int i,j,iamax,ind;
   float __attribute__((aligned(64))) vindx[4];                    A
   __m128 v_x, v_neg, ind_ant, var_ant, var_max, ind_new, mask, indand, indnand;

   max = fabsf(x[0]);                                              B
   iamax = 0;

   //Prologo                                                       PRÓLOGO
   for(i=0;i<nprol;i++){
      if (fabsf(x[i]) > max){                                      C
         iamax = i;
         max = fabsf(x[i]);
      }
   }
   ind_ant = _mm_set1_ps((float)iamax);                            D
   var_ant = _mm_set1_ps((float)max);
   ind_new = _mm_setr_ps(nprol, nprol+1, nprol+2, nprol+3);
   //Cuerpo                                                        CUERPO
   for (i = nprol; i < nvec1; i+=4){
      v_x = _mm_load_ps(&x[i]);
      v_neg = _mm_sub_ps(_mm_setzero_ps(), v_x);
      v_x = _mm_max_ps(v_x, v_neg);
      var_max = _mm_max_ps(v_x,var_ant);

      mask = _mm_cmpeq_ps(var_max, var_ant);

      indand = _mm_and_ps(mask,ind_ant);
      indnand = _mm_andnot_ps(mask,ind_new);

      ind_ant = _mm_add_ps(indand,indnand);
      ind_new = _mm_add_ps(ind_new, _mm_set1_ps(4.0));
      var_ant = var_max;
   }
   _mm_store_ps(&vindx[0],ind_ant);

   for(i=0;i<4;i++){                                               Reducción
      ind = vindx[i];
      if(fabsf(x[ind])> max){
         max = fabsf(x[ind]);
         iamax = ind;
      }else if(fabsf(x[ind])== max && ind < iamax){
         max = fabsf(x[ind]);
         iamax = ind;
      }
   }
   //Epilogo
```

```
    for(i = nvec1; i < n ; i++){
      if (fabsf(x[i]) > max){
        iamax = i;
        max = fabsf(x[i]);
      }
    }
    return iamax;
}
```

EPÍLOGO

Ilustración 4-9 Función si_amax en SSE

Como puede comprobarse al inicio de la ilustración 4-9, se trata de la función alineada de la solución vectorial (SSE) al problema.

Primeramente es necesario declarar las variables necesarias para poder realizar las operaciones, tal como se muestra en las primeras líneas (A).

- ***max*:** es la variable que almacenará el máximo valor entre las tres secciones del código (prologo, cuerpo y epilogo).

- ***i***: es la variable utilizada para iterar en el bucle *for*.

- ***iamax*:** es la variable que almacena el índice del mayor elemento en valor absoluto. Es variable que se devolverá como salida de la función.

- ***ind* y *vindx*:** son variables necesarias para almacenar el valor después de utilizar las instrucciones vectoriales. En este caso *vindx* es un variable que contiene 4 valores alineados a bloques de 64 bytes. Su uso: *__attribute__((aligned(64)))*.

- Las demás variables declaradas son variables de tipo __m128, utilizadas por las instrucciones SIMD. Estas variables contienen 128 bits de información, es decir, 4 elementos de tipo *float*.

Una vez declaradas las variables se indica que el máximo está en el primer elemento del vector, también se indica su índice (B).

Seguidamente se encuentra el prólogo (C). Esta sección implementa el mismo código que la solución secuencial al problema, pero limitando el bucle *for* hasta el primer elemento alineado del vector (*nprol*).

Después de terminar de recorrer el prólogo, *max* e *iamax* pueden haber cambiado de valor. Antes de entrar en el bucle vectorial es necesario asignar los valores del máximo y el índice del máximo hasta el momento calculado en variables de tipo __m128, para que puedan utilizarse por las instrucciones SIMD. Como no existe una instrucción que almacene el índice del máximo valor, es necesario utilizar dos variables que guarden los índices anteriores y los nuevos índices (D).

Para asignar los valores se utilizan las instrucciones *_mm_set1_ps* y *_mm_setr_ps*. La instrucción *_mm_set1_ps* carga el valor pasado por parámetro en los cuatro valores float de los que se compone y la instrucción *_mm_setr_ps* carga los valores introducidos por parámetro en orden inverso en los cuatro valores float de los que se compone.

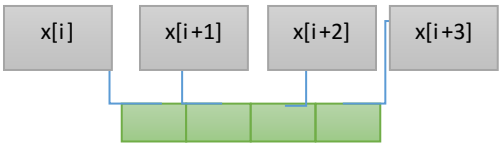
- ***ind_ant*:** va a contener los índices de los valores máximos del vector (en valor absoluto) leídos hasta el momento y va a ser actualizado en cada iteración con los nuevos índices.

- ***var_ant*:** va a contener los valores máximos de los índices almacenados en *ind_ant* en valor absoluto.

- ***ind_new*:** es una variable que se actualizará cada iteración con los índices de los siguientes valores. Inicialmente, se inicializa con los valores de los nuevos índices.

A continuación, se accede al bucle *for* y se realizan las siguientes operaciones con las instrucciones vectoriales.

- ***v_x = _mm_load_ps(&x[i]);***
  v_x es una variable que tendrá cargados 4 valores de tipo float, pertenecientes a la dirección del vector que se le pase por parámetro.



- Obtención del valor absoluto:
  - ***v_neg = _mm_sub_ps(_mm_setzero_ps(), v_x);***
    v_neg es una variable que almacenará los mismos datos que *v_x* con valor opuesto. Esta variable junto con *v_x* sirven para obtener el valor absoluto de los elemento del vector.

  - ***v_x = _mm_max_ps(v_x, v_neg);***
    v_x en este caso es reutilizada y almacenará los 4 valores con mayor valor entre las variables *v_x* y *v_neg*, es decir *v_x* tendrá el valor absoluto de sus datos iniciales.

  - ***var_max = _mm_max_ps(v_x, var_ant);***
    var_max es la variable que contiene los valores máximos de los elementos del vector entre iteraciones, en concreto almacena 4 valores en tipo float.

- ***mask = _mm_cmpeq_ps(var_max, var_ant);***
  mask es la máscara que se debe generar para que posteriormente se pueden almacenar los índices con mayor valor. Lo que realiza la instrucción es una comparación entre la variable *var_max* y *var_ant*, si alguno de los cuatro valores comparados son iguales almacenará un 1 es esa posición y un 0 en caso contrario.

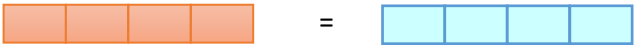
- ***indand = _mm_and_ps(mask,ind_ant);***
  indand es la variable que almacenará los índices anteriores que sigan siendo máximos, en caso contrario pondrá un 0. Aplica la operación AND sobre las dos variables pasadas por parámetro.

- ***indnand = _mm_andnot_ps(mask,ind_new);***
  indnand es la variable que almacenará los nuevos índices que serán máximos, en caso contrario pondrá un 0. Aplica la operación NAND sobre las dos variables pasadas por parámetro.

- ***ind_ant = _mm_add_ps(indand,indnand);***
  Se realiza la actualización de valores máximos, ya que en *indand* se tienen los índices máximos que se mantienen y en *indnand* los nuevos índices máximos.

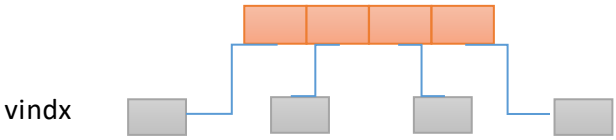
- ***ind_new = _mm_add_ps(ind_new, _mm_set1_ps(4.0));***
  Se actualizan los nuevos índices sumando 4 a cada valor ya existente por iteración.

- ***var_ant = var_max;***
  A *var_ant* se le asignarán los nuevos máximos recogidos en *var_max*.



Después de obtener cuatro valores máximos almacenados en la variable *var_ant* es necesario convertir esos valores de una variable __128 a una variable vector de tipo float, en este caso *vindx*. Para conseguir eso se utiliza la siguiente instrucción SIMD.

_mm_store_ps(&vindx[0],ind_ant);



Al almacenar los valores en la variable *vindx* es necesario procesarlos para obtener el máximo y el índice máximo. Para ello se calcula el índice máximo de los 4 valores almacenados. En el caso de que dos índices tengan el mismo valor, se escoge aquel que tenga el índice menor.

Por último, se realiza el epilogo con los datos proporcionados por el cuerpo de la misma forma que se implementó el prólogo y se devuelve el índice del máximo valor absoluto del vector.

4.2.3.1 Ejemplo iteración 16 elementos

Vector x	2	3	4	6	-2	-7	3	6	1	1	0	8	-9	5	8	2

Se presupone que directamente se accede al cuerpo de la estructura, es decir, el vector está alineado completamente y no se accede ni al prólogo, ni al epilogo.

Inicio del cuerpo				
ind_ant	0	0	0	0
var_ant	2	2	2	2
ind_new	0	1	2	3
Iteración 1				
v_x	2	3	4	6
v_neg	-2	-3	-4	-6
v_x	2	3	4	6
var_max	2	3	4	6
mask	1	0	0	0
indand	0	0	0	0
indnand	0	1	2	3
ind_ant	0	1	2	3
ind_new	4	5	6	7
var_ant	2	3	4	6
Iteración 2				
v_x	-2	-7	3	6
v_neg	2	7	-3	-6
v_x	2	7	3	6
var_max	2	7	3	6
mask	1	0	0	1
indand	0	0	0	3
indnand	0	5	6	0
ind_ant	0	5	6	3
ind_new	8	9	10	11
var_ant	2	7	3	6
Iteración 3				
v_x	1	1	0	8
v_neg	-1	-1	0	-8
v_x	1	1	0	8
var_max	2	7	3	8
mask	1	1	1	0
indand	0	5	6	0
indnand	0	0	0	8
ind_ant	0	5	6	8

ind_new	12	13	14	15
var_ant	2	7	3	8
Iteración 4				
v_x	-9	5	8	2
v_neg	9	-5	-8	-2
v_x	9	5	8	2
var_max	9	7	8	8
mask	0	1	0	1
indand	0	5	0	8
indnand	12	0	14	0
ind_ant	12	5	14	8
ind_new	16	17	18	19
var_ant	9	7	8	8
Fin del cuerpo				
vindx	12	5	14	8

Tabla 4-1 Desglose de iteraciones

En la tabla 4-1, a partir de del vector vindx se obtienen los índices de los elementos mayores en valor absoluto. Por lo que se comprueba entre ellos cual es el mayor. En caso de repetición de valores, el primer índice será el elegido.

```
for(i=0;i<4;i++){
      ind = vindx[i];
      if(fabsf(x[ind])> max){
         max = fabsf(x[ind]);
         iamax = ind;
      }else if(fabsf(x[ind])== max && ind < iamax){
         max = fabsf(x[ind]);
         iamax = ind;
      }
```

Ilustración 4-10 Obtención del índice mayor al final del cuerpo - Reducción

Se eliminan las dependencias de control por dependencias de datos. Esto hace que el predictor de saltos tenga una menor presión. Ocurre en las funciones i_amax y nrm2 del primer nivel de BLAS.

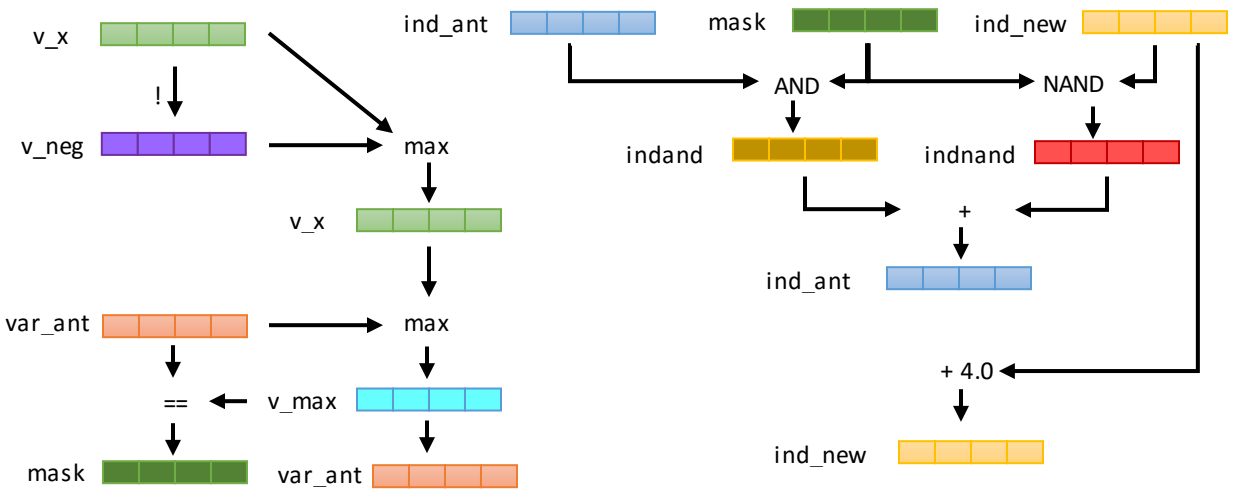


Ilustración 4-11 Diagrama de ejecución si_amax

4.3 PARALELO

La mayoría de los procesadores, hoy en día están compuestos por más de un *core* y en el caso de aquellos que sean de Intel pueden tener *HyperThreading* simulando más *cores* de los que tiene.

Para este proyecto se han utilizado varias configuraciones que se componían de varios *cores*. Por ello, todas las funciones y subrutinas del primer nivel del BLAS realizadas también implementan la posibilidad de ejecución en paralelo.

Adicionalmente, también se ha implementado una combinación entre las instrucciones SIMD y el paralelismo proporcionado por OpenMP.

4.3.1 Estructura de operación

La implementación del código en paralelo es una réplica del código secuencial de las funciones, diferenciándose únicamente en la integración de la directiva de OpenMP. En esa directiva se indican todos los parámetros de la operación paralela que se han de tener en cuenta, ilustración 4-12.

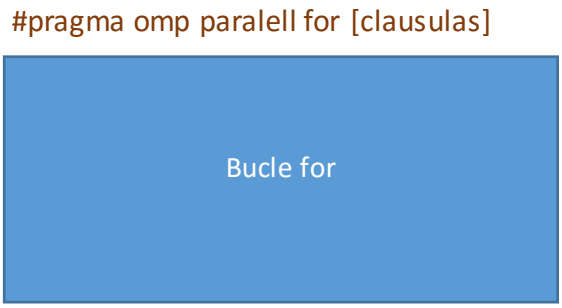


Ilustración 4-12 Representación del uso de directiva

En una de las funciones del nivel de BLAS, i_amax, ha sido necesaria la creación de una nueva función para realizar la ejecución en paralelo. Esta función es una réplica del código secuencial, pero no recorre por completo todos los elementos del vector, sino solo una parte de ellos dependiendo del número de *threads* con los que se ejecute. Se requiere de un reparto de carga para realizar este tipo de operaciones.

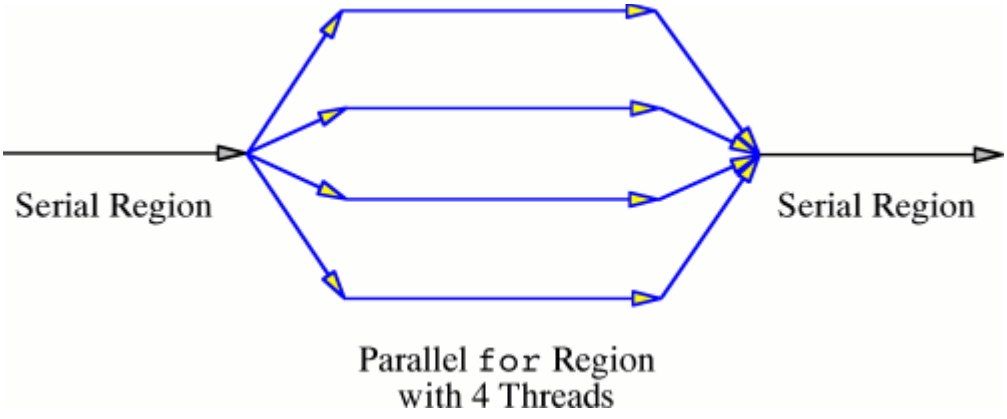


Ilustración 4-13 Representación del modelo fork-join

4.3.2 Reparto de carga

El reparto de carga es un tipo de acción que realizan las directivas OpenMP para repartir un bucle o región paralela. Una estructura *for* tiene determinadas el número de iteraciones que va a realizar. Por ello la directiva de OpenMP *"#pragma omp parallel for"* realiza una reparto de las iteraciones entre los *threads* disponibles. Existen varios tipos de reparto de carga, que están implementados por OpenMP, pero que requieren de una directiva para su activación. Los tipos son los siguientes:

- **Static**: reparto predeterminado y equitativo de iteraciones entre el número de *threads* disponibles. Si se especifica el *chunk* (número opcional de elementos), a cada *thread* se le asigna ese número fijo de iteraciones hasta que se reparta completamente. Menor coste y mejor localidad de datos. Por defecto realiza la operación "número de elementos/número de threads" consecutivos a cada thread.
- **Dynamic**: asignación dinámica de trozos de tamaño k. Cuando un *thread* este libre, se le asigna la iteración (o número de iteraciones igual al *chunk*). El tamaño por defecto es 1. Mayor coste y carga más equilibrada.
- **Guided**: planificación dinámica con trozos de tamaño decreciente. Se comporta como *dynamic*, pero reduciendo el *chunk* a lo largo de la ejecución.

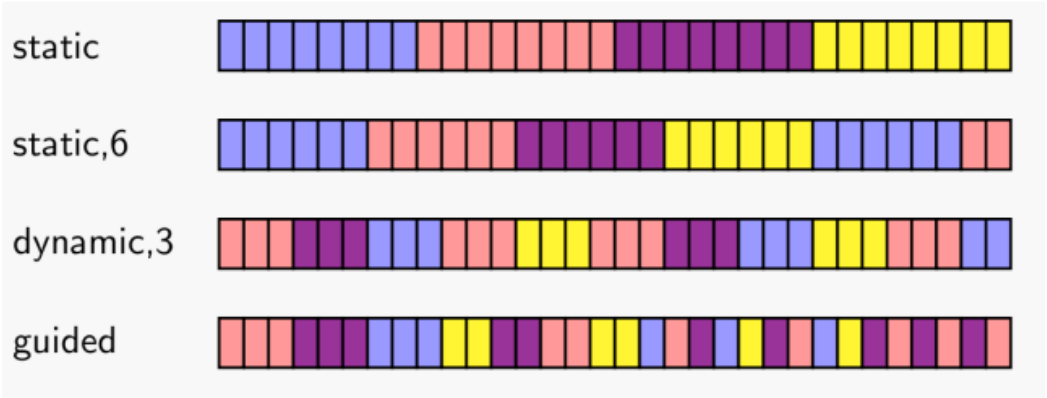


Ilustración 4-14 Tipos de planificación de reparto

De manera predeterminada el reparto utilizado en el proyecto es de tipo *static*, es decir, que las iteraciones se reparten equitativamente entre los *threads* disponibles.

Para indicar el número de *threads* se utiliza una función de OpenMP, que como parámetro se le pasa el número *threads* disponibles en el procesador. La función es *omp_set_num_threads().* Esta función se utiliza en la rutina de decisión de ejecución para determinar el número de *threads* que van a utilizarse. También es posible hacerlo mediante una clausula en el *pragma*.

Para el reparto manual de iteraciones, es decir, sin utilizar la directiva *pragma* sobre un bude *for*, se utiliza las regiones paralelas. Una región paralela es accedida por cada *thread* y todas realizan la misma operación, se utiliza en los casos en que se quiera añadir código vectorial.

A continuación se muestra un ejemplo de la estructura de reparto seguida, ilustración 4-15.

```
trozo=ceil((float)n/(float)NTHR);

#pragma omp parallel  private(i,ini,fin) firstprivate(n,trozo)
{
   i = omp_get_thread_num();
   ini=i*trozo;
   fin=min(trozo*(i+1),n);
   funcion_vectorial(fin-ini,&x[ini])
}
```

Ilustración 4-15 Reparto de carga manual

Tal y como se muestra en el código se realiza un reparto equitativo del vector entre los *threads*. Primero se divide el número total de elementos entre el número de *threads* que se van a utilizar para realizar el código en paralelo. Una vez obtenido el trozo (parte entera de la división) se accede a la sección paralela donde se indica que *i, ini y fin* son variables privadas para cada *thread*. Estas variables se utilizarán para obtener el rango de elementos del vector que cada *thread* tiene que recorrer. Este tipo de reparto se ha realizado para la ejecución combinada, paralela - vectorial, de las funciones del primer nivel del BLAS.

- La variable *i* contiene el número de *thread* que ha entrado en esa región. Numerados de 0 hasta n-1 *threads*.

- La variable *ini* contiene el índice del primer elemento del vector asignado a cada *thread*.

- La variable *fin* contiene el índice del último elemento del vector asignado a cada *thread*. En el caso de que la división entera no sea exacta se comprueba el mínimo con el número de elementos para asignar el final del trozo.

- Por último se llama a la función vectorial indicando los parámetros. Para cada *thread*, la operación *fin-ini* indica el número de elementos que recorre y *&x[ini]* indica el primer elemento del vector al que se tiene que acceder.

4.3.3 Ejemplo visual

Para poder tener una idea más visual del trabajo realizado a continuación se muestran ejemplos de los tipos de paralelismo creado.

```
static void alig_scopy_sse(int n, float *x, float *y, int nprol, int nvec1)
{
   int i;
   __m128 v_x, v_a;

   //Prologo
   for(i = 0; i < nprol ; i++)
      y[i] = x[i];

   //Cuerpo
   for (i = nprol; i < nvec1; i+=4){
      v_x = _mm_load_ps(&x[i]);;
      _mm_store_ps(&y[i],v_x);
   }

   //Epilogo
   for(i = nvec1; i < n ; i++)
      y[i] = x[i];
}

static void noalig_scopy_sse(int n, float *x, float *y, int nprol, int nvec1)
{
   int i;
   __m128 v_x, v_a;

   //Prologo
   for(i = 0; i < nprol ; i++)
      y[i] = x[i];

   //Cuerpo
   for (i = nprol; i < nvec1; i+=4){
      v_x = _mm_load_ps(&x[i]);;
      _mm_storeu_ps(&y[i],v_x);
   }

   //Epilogo
   for(i = nvec1; i < n ; i++)
      y[i] = x[i];
}

static void scopy_sse(int n, float *x, float *y)
{

   int dir1 = (int) (&x[0]);
   int nprol1 = 16 - ((dir1>>2)%16);
   int nvec1 = (((int) floor((double)((n-nprol1)/4.0)))<<2)+ nprol1;

   int dir2 = (int) (&y[0]);
   int nprol2 = 16 - ((dir2>>2)%16);

   if(nprol1 == 16) nprol1 = 0;
   if(nprol2 == 16) nprol2 = 0;
```

```
   if(nprol1 == nprol2)
      alig_scopy_sse(n,x,y,nprol1,nvec1);
   else
      noalig_scopy_sse(n,x,y,nprol1,nvec1);

}

static void scopy_omp_sse(int n, float *x, float *y){   A
   int i,trozo,ini,fin;
   trozo=ceil((float)n/(float)NTHR);

   #pragma omp parallel  private(i,ini,fin) firstprivate(n,trozo)
   {
      i = omp_get_thread_num();
      ini=i*trozo;
      fin=min(trozo*(i+1),n);
      scopy_sse(fin-ini,&x[ini],&y[ini]);   B
   }
}
```

Ilustración 4-16 Ejemplo de paralelismo indirecto de la función scopy (SSE)

La función de la ilustración 4-16 implementa la copia de elementos de un vector en otro. El código está dividido en dos funciones ya que a una de ellas se le llama desde la región paralela antes comentada. Por un lado se crea la estructura de reparto manual representada en la ilustración 4-15 (A) y en este caso, la llamada vectorial que se realiza es la que tiene instrucciones SSE (B).

Por otro lado se crean las dos funciones vectoriales necesarias para resolver el problema en paralelo. Desde la sección paralela se le llama a la función de alineamiento de vector (C) y después está a la que realiza todas las operaciones vectoriales (D).

```
static float sdot_omp(int n, float *x, float *y)
{
   int i;
   float dot = 0.0;
   #pragma omp parallel for reduction(+:dot)
   for (i = 0; i < n; ++i)
      dot = dot + x[i]*y[i];

   return dot;
}
```

Ilustración 4-17 Ejemplo de paralelismo en bucle - sdot

La función de la ilustración 4-17, se trata del producto escalar. Esta función realizada en paralelo, tiene un *pragma* que divide las iteraciones del bucle *for* entre los *threads* disponibles y una vez calculados los datos oportunos realiza una reducción de los datos para que recoja todos los valores calculados por los *threads* y, en este caso, los sume.

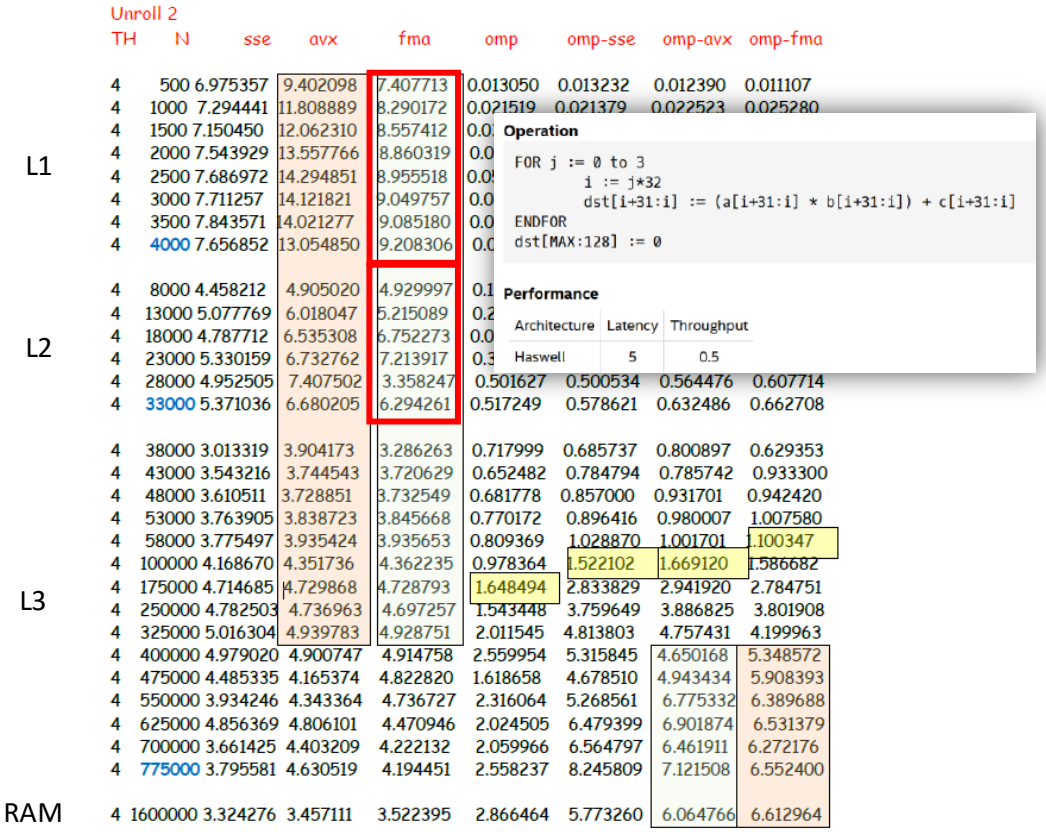
4.4 AUTO-LIMITACIONES

En la programación vectorial no están implementadas todas las instrucciones SIMD para cada versión, es decir, hay instrucciones para SSE que no se han implementado para AVX y viceversa. Esto en algunos casos obliga a utilizar varias instrucciones en lugar de una, dando lugar a una peor optimización. Un ejemplo de ello es la existencia de la instrucción de obtención de valor absoluto. El conjunto de instrucciones SSE Y AVX la tienen implementada solo para el tipo de dato entero, en el caso de los datos de precisión simple y doble hace falta calcularlo mediante más instrucciones.

Siguiendo con la programación vectorial, la tecnología FMA tiene dos inconvenientes. Por un lado, no existen instrucciones para realizar operaciones sobre números enteros, por lo que se limita a operaciones en coma flotante. Por otro lado, las instrucciones FMA tienen como coste 5 ciclos por instrucción. Ese coste está relacionado con la latencia de las instrucciones. En el caso de las instrucciones FMA tienen un *throughput* de 0.5 por lo que para obtener un mejor rendimiento sería necesario realizar un *unrolling 2* de las instrucciones FMA (dos instrucciones por ciclo) y de esta forma se ocultaría la latencia de la operación. En este proyecto no se ha realizado *unrolling 2*, por lo que en algunos casos el resultado obtenido podría mejorarse. En el documento adjunto de "Resultados de rendimiento" puede observarse la ganancia en uno de los casos.

	TH	N	sse	avx	fma	omp	omp-sse	omp-avx	omp-fma
	4	500	3.749826	6.854777	4.575680	0.011466	0.010820	0.013145	0.014639
	4	1000	3.874544	7.241990	4.659649	0.025473	0.026476	0.026008	0.026360
	4	1500	3.908666	7.582139	4.715474	0.030425	0.031625	0.014225	0.027749
L1	4	2000	3.890055	7.628386	4.690873	0.041609	0.042104	0.043427	0.043442
	4	2500	3.953109	7.738123	4.745863	0.050984	0.047375	0.052556	0.056215
	4	3000	3.984016	7.723069	4.789671	0.062125	0.062766	0.033990	0.062957
	4	3500	3.954867	7.758360	2.610089	0.071416	0.074950	0.076311	0.075392
	4	4000	3.964713	7.827048	4.747915	0.090152	0.099324	0.100659	0.096966
	4	8000	3.689465	4.735836	3.961448	0.138194	0.171371	0.168404	0.150570
	4	13000	3.806163	5.597061	4.237695	0.276800	0.265615	0.259499	0.242771
	4	18000	3.865862	6.147745	4.399558	0.319847	0.379504	0.333716	0.342740
L2	4	23000	3.898318	6.236363	4.460596	0.422268	0.378009	0.408934	0.445812
	4	28000	3.918687	6.704232	4.540413	0.563024	0.471970	0.511358	0.511857
	4	33000	3.987993	6.824754	4.644954	0.520939	0.610686	0.599110	0.686645
	4	38000	2.021901	3.967033	3.787984	0.703258	0.637601	0.647401	0.742857
	4	43000	3.344064	3.669530	3.542579	0.677391	0.720048	0.807476	0.812203
	4	48000	3.340824	3.656921	3.693579	0.722797	0.837597	1.019776	0.945117
	4	53000	3.401520	3.839493	3.807104	0.777415	0.839145	1.023318	1.129407
	4	58000	2.009828	3.707267	3.754838	0.786713	1.035213	0.971122	1.176605
	4	100000	3.704470	4.357507	4.204728	1.072276	1.495619	1.630760	1.547845
L3	4	175000	3.818545	4.525010	4.038125	1.220012	2.630402	2.819102	2.934816
	4	250000	3.856818	4.730270	4.539656	1.927240	3.432564	3.724284	3.443460
	4	325000	3.908076	4.905091	4.603240	1.487948	4.011391	4.405959	4.317255
	4	400000	3.706119	4.815726	4.271470	1.637939	4.084992	4.488303	4.381242
	4	475000	3.884584	4.846502	4.639358	2.662461	5.224535	4.991229	5.083981
	4	550000	3.814582	4.443401	4.703497	1.736618	5.239362	6.064794	5.033692
	4	625000	3.404177	4.685001	4.579550	1.798938	5.065277	5.030370	6.032582
	4	700000	3.359411	4.616135	4.563188	2.174080	5.459618	6.586344	6.301290
	4	775000	3.651821	4.034275	3.869359	2.016097	6.257515	5.888563	6.273144
RAM	4	1600000	3.255520	3.334554	3.397030	3.114893	5.773428	5.949997	5.756095

Ilustración 4-18 Resultado de rendimiento – 2ª configuración – sdot



Unroll 2

	TH	N	sse	avx	fma	omp	omp-sse	omp-avx	omp-fma
L1	4	500	6.975357	9.402098	7.407713	0.013050	0.013232	0.012390	0.011107
	4	1000	7.294441	11.808889	8.290172	0.021519	0.021379	0.022523	0.025280
	4	1500	7.150450	12.062310	8.557412	0.0			
	4	2000	7.543929	13.557766	8.860319	0.0			
	4	2500	7.686972	14.294851	8.955518	0.0			
	4	3000	7.711257	14.121821	9.049757	0.0			
	4	3500	7.843571	14.021277	9.085180	0.0			
	4	4000	7.656852	13.054850	9.208306	0.0			
L2	4	8000	4.458212	4.905020	4.929997	0.1			
	4	13000	5.077769	6.018047	5.215089	0.2			
	4	18000	4.787712	6.535308	6.752273	0.0			
	4	23000	5.330159	6.732762	7.213917	0.3			
	4	28000	4.952505	7.407502	3.358247	0.501627	0.500534	0.564476	0.607714
	4	33000	5.371036	6.680205	6.294261	0.517249	0.578621	0.632486	0.662708
L3	4	38000	3.013319	3.904173	3.286263	0.717999	0.685737	0.800897	0.629353
	4	43000	3.543216	3.744543	3.720629	0.652482	0.784794	0.785742	0.933300
	4	48000	3.610511	3.728851	3.732549	0.681778	0.857000	0.931701	0.942420
	4	53000	3.763905	3.838723	3.845668	0.770172	0.896416	0.980007	1.007580
	4	58000	3.775497	3.935424	3.935653	0.809369	1.028870	1.001701	1.100347
	4	100000	4.168670	4.351736	4.362235	0.978364	1.522102	1.669120	1.586682
	4	175000	4.714685	4.729868	4.728793	1.648494	2.833829	2.941920	2.784751
	4	250000	4.782503	4.736963	4.697257	1.543448	3.759649	3.886825	3.801908
	4	325000	5.016304	4.939783	4.928751	2.011545	4.813803	4.757431	4.199963
	4	400000	4.979020	4.900747	4.914758	2.559954	5.315845	4.650168	5.348572
	4	475000	4.485335	4.165374	4.822820	1.618658	4.678510	4.943434	5.908393
	4	550000	3.934246	4.343364	4.736727	2.316064	5.268561	6.775332	6.389688
	4	625000	4.856369	4.806101	4.470946	2.024505	6.479399	6.901874	6.531379
	4	700000	3.661425	4.403209	4.222132	2.059966	6.564797	6.461911	6.272176
	4	775000	3.795581	4.630519	4.194451	2.558237	8.245809	7.121508	6.552400
RAM	4	1600000	3.324276	3.457111	3.522395	2.866464	5.773260	6.064766	6.612964

Operation

```
FOR j := 0 to 3
        i := j*32
        dst[i+31:i] := (a[i+31:i] * b[i+31:i]) + c[i+31:i]
ENDFOR
dst[MAX:128] := 0
```

Performance

Architecture	Latency	Throughput
Haswell	5	0.5

Ilustración 4-19 Resultado de rendimiento - 2a configuración - sdot unrolling

Tal como se muestra en la ilustración 4-18 y 4-19, el impacto de aplicar *unrolling* en la parte FMA mejora en la caché de primer nivel, el segundo nivel tienen una mejora menor y el tercero tiene datos similares.

Debido a la existencia de una tecnología *Turbo Boost* que incorporan algunos procesadores Intel utilizados se han realizado modificaciones en el método de obtención de datos. La tecnología hace que los *cores* suban de voltaje y frecuencia ofreciendo más rendimiento a la ejecución en ese caso. Esto sucede cuando al ejecutarse una aplicación, uno o varios *cores* tienen mucha carga. Como solución, el procesador actúa habilitando esta tecnología. Por ello, se ha de mencionar que los *speedups* calculados son los mínimos que pueden obtenerse ya que pueden haberse obtenido cuando el *Turbo Boost* está activo.

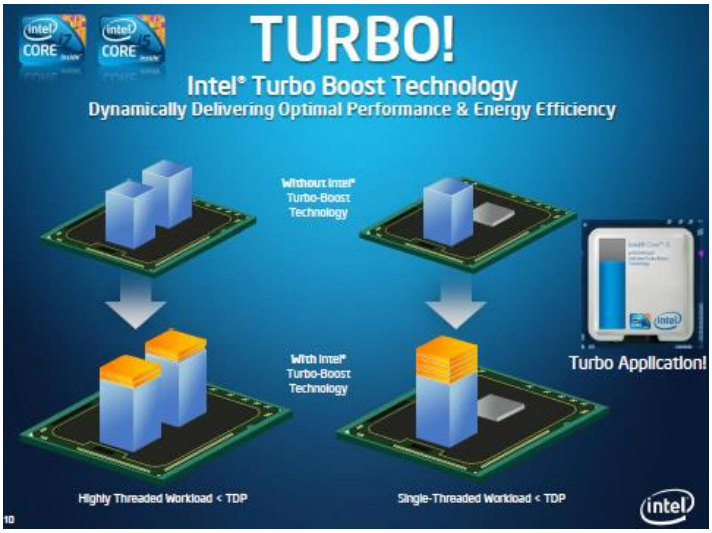


Ilustración 4-20 Representación de la tecnología Turbo Boost (Intel)

El *Turbo Boost* tiene una limitación que viene indicada por *Thermal Desing Power*, o TDP, del procesador. El TDP indica la máxima potencia que es capaz de usar un dispositivo (medida en vatios), sin que este comience a fallar. Cuando el procesador utiliza el *Turbo Boost* aumenta el consumo del procesador, pero siempre por debajo de ese valor. En el caso de que se llegue a ese valor crítico, el propio procesador reduce su frecuencia y tensión.

En este proyecto, todos los resultados recogidos en el documento "Resultados de rendimiento" han sido obtenidos bajo una ejecución en un sistema con opción energética "Equilibrado". Este tipo de opción se explicara en el capítulo 8 basado en el consumo.

5 CAPÍTULO: EXPERIMENTACIÓN

Después de realizar todas las implementaciones de las funciones de todos los tipos de programación posible: secuencial, paralela y/o vectorial, es necesario realizar una comprobación de los resultados para determinar su correcta implementación.

Como el código secuencial es una copia de los códigos originalmente realizados en Fortran, que han sido validados para la generación del estándar de la librería, se toman como referencia para comparar los resultados obtenidos frente a los demás tipos de funciones realizadas en paralelo, vectorial y/o paralelo. Para ello, se ha creado una estructura que permite localizar los errores cometidos.

5.1 OBTENCIÓN DE DATOS: TIEMPO Y VALIDACIÓN DE RESULTADOS

La obtención de datos se realiza mediante una estructura de comprobación por cada tipo de función realizada para cada rutina del primer nivel del BLAS.

Se crea un archivo *main.c* para cada rutina y se crea una estructura de datos para poder comprobar los datos. El orden es el siguiente:

- Se declaran todas las variables necesarias para poder comprobar y comparar entre los resultados de los diferentes tipos de procesamientos. Si las variables son vectores es necesario indicar mediante sintaxis de código ensamblador que debería ser alineada para obtener un máximo rendimiento.

```
static double __attribute__((aligned(64))) x[NUM_ELEM];
```

- Se declaran todas las variables relacionadas con la medición y obtención de resultados.

```
static unsigned long long ini,fin,dif,min,max,sum,pri,med,med_seq,wcalc,exito;
```

- Se crea una función de inicialización, que inicializa por igual todas las variables de tipo vector para cada tipo de programa. De esta manera, todas estarán bajo los mismos datos y deberán proporcionar el mismo resultado. Al igual que en la llamada a las funciones, todos los parámetros que no sean vectores tendrán que tener el mismo valor, ilustración 5-1.

```
void inicializar (double *x){
  int i;
  for (i=0; i<NUM_ELEM;  i++){ //Inicializar
    x[i]= (double)i/10;
  }
}
```

Ilustración 5-1 Ejemplo de inicialización de elementos de vector

- Seguidamente se crea la función principal (*main*). En ella se realizan comprobaciones para cada tipo de procesamiento disponible y se mide el tiempo de cómputo de la llamada a la función. Este tiempo se recoge 11 veces, por lo que esta medición está integrada en un bucle *for* que permite obtener medidas más estables de tiempo. Mientras que se realizan las llamadas a las funciones, se van obteniendo diferentes

datos de tiempo: media de ejecución, ejecución más larga, ejecución más corta y *speedup* respecto a la función que implementa el procesamiento secuencial, ilustración 5-2.

```
    for (min=UINT_MAX,  max= 0, sum=0, pri= 0, i=0; i<Nc; i++){
          inicializar(x,y_sse);
          ini = rdtsc();
          funcion(NUM_ELEM,  x);
          fin = rdtsc();
          if(i>0){
             dif = (fin - ini);
             sum += dif;
             max = (dif > max)? dif : max;
             min = (dif < min)? dif : min;
          }else{
             dif = (fin - ini);
             pri = dif;
          }
    }
med = sum/(Nc-1);
```

Ilustración 5-2 Ejemplo de estructura de obtención de datos

- Por último con todos los datos obtenidos, se calcula el éxito del vector de la variable resultado de la función, siempre comparándolo con el resultado secuencial , ilustración 5-3.

```
wcalc = NUM_ELEM;
for (i=0; i<NUM_ELEM;  i++){
  if(y[i]!=y_sse[i]){
     printf("Valor correcto %f - %f Valor incorrecto \n", y[i], y_sse[i]);
     wcalc --;
  }
}
exito = (wcalc/NUM_ELEM)*100;
```

Ilustración 5-3 Ejemplo de estructura de comprobación de datos

De esta forma se asegura que el valor de cada tipo de procesamiento es el mismo para cada función del primer nivel de BLAS. Esta comprobación se realizaba sobre un número fijo de elementos de vector y su salida contendrá toda la información necesaria.

Para realizar una medición del tiempo de CPU, de las 11 veces que se recoge el tiempo se desprecia la primera, debido a que tienen que cargarse todos los datos en memoria y se obtiene un tiempo con una desviación muy grande respecto a los 10 restantes, ilustración 5-4.

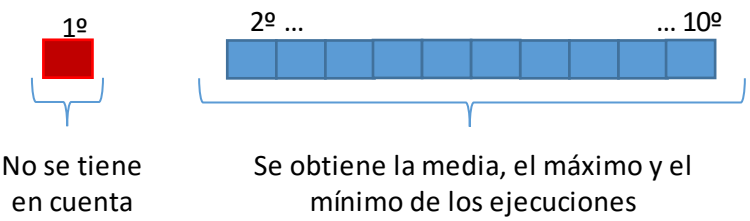


Ilustración 5-4 Elección de ejecuciones validas

La razón por la que se ha tomado esa decisión tiene que ver con el tipo de ejecución que se quiera realizar. Existe la ejecución en frio, en la que ningún dato está cargado en memoria y la ejecución en caliente en la que si lo están. En este proyecto se ha optado por una ejecución en caliente en la que se desprecia la primera ejecución en la que se cargan todos los datos.

5.2 MEDICIÓN DE TIEMPOS

La compilación de estos ficheros con las funciones se ha realizado mediante un fichero *.bat* que contiene los comandos necesarios para que el GCC previamente instalado en el terminal pudiera resolverlo.

En este fichero, para realizar pruebas sobre varios tipos de tamaños de vector pertenecientes a los diferentes tamaños de caché del procesador, se ha generado y modificado continuamente un fichero .h que contiene el número de elementos que debe tener el vector, indicado por la variable NUM_ELEM y los *threads* que se debe utilizar, NTHR, Ilustración 5-5 (A).

- **Nc**: es el número de ejecuciones que se van a realizar para tomar los datos. Sobre esas 11 ejecuciones, tal como se muestra en el apartado 5.1.1, se obtienen las medias de tiempo despreciando el primero.
- **NTHR**: indica el número de *threads* con los que se va a trabajar. En este caso se indican manualmente, pero sería necesario comprobar la información del procesador para indicarlo correctamente.
- **NUM_ELEM**: indica el número de elementos de los vectores que se van a utilizar. Este dato se modifica para poder obtener todos los datos pertenecientes a los *speedups* en cada nivel de caché.

```
del saxpy.h
echo #define Nc 11 >> saxpy.h
echo #define NTHR 2 >> saxpy.h
echo #define NUM_ELEM  500 >> saxpy.h

"C:\MinGW\bin\gcc.exe"  -O3          -c saxpy_seq.c
"C:\MinGW\bin\gcc.exe"  -O3   -fopenmp   -c saxpy_omp.c
"C:\MinGW\bin\gcc.exe"  -O3   -fopenmp -mavx2 –msse4 -mfma   -o saxpy_stats.exe main.c
saxpy_seq.o saxpy_omp.o

saxpy_stats.exe >> ticks.txt

del saxpy.h
echo #define Nc 11 >> saxpy.h
echo #define NTHR 2 >> saxpy.h
echo #define NUM_ELEM  1000 >> saxpy.h

"C:\MinGW\bin\gcc.exe"  -O3          -c saxpy_seq.c
"C:\MinGW\bin\gcc.exe"  -O3   -fopenmp   -c saxpy_omp.c
"C:\MinGW\bin\gcc.exe"  -O3   -fopenmp -mavx2 –msse4 -mfma   -o saxpy_stats.exe main.c
saxpy_seq.o saxpy_omp.o

saxpy_stats.exe >> ticks.txt
…
```

A B C D

Ilustración 5-5 Ejemplo de compilación y recogida de datos de subrutina saxpy

Mediante este archivo *.bat,* ilustración 5-5, se redirige la salida de información a un fichero *ticks.txt* donde se recogen los datos pertenecientes a los *speedup* de todos los tipos de procesamiento de las funciones respecto a su procesamiento secuencial (C).

Antes de ello es necesario realizar la compilación de los datos (B). Esta compilación se realiza separando el código secuencial, paralelo y vectorial. En el caso de que se compile en conjunto con los mismos *flags*, el compilador ve posible una optimización vectorial sobre código secuencial y paralelo y la realiza, obteniendo de esta manera datos con un rendimiento menor. Esta compilación se realiza para cada tamaño de vector (D). Este contenido es controlado por el menú principal antes mencionado en el apartado 5.1.

Este archivo solo sirve para la recogida de datos y debe de ejecutarse por cada función del primer nivel de BLAS. Una vez obtenidos, se creará el fichero *ticks.txt* antes mencionado con los datos de los *speedups* organizados por número de elementos de vector y por tecnología paralela.

### 2 threads ###						
2	500	3,170929	3,236264	0,006558	0,007286	0,007323
2	1000	3,472381	3,555339	0,013179	0,017095	0,017846
2	1500	3,814464	3,796460	0,010947	0,013152	0,012930
2	2000	4,966557	4,492544	0,016454	0,027813	0,030972
2	2500	3,617104	3,759596	0,046964	0,034867	0,042779
2	3000	3,450619	3,183333	0,036145	0,032806	0,047544
2	3500	3,874456	3,724631	0,039712	0,042154	0,045990
2	4000	3,168621	3,363960	0,035071	0,039079	0,051308
2	8000	1,911544	1,924526	0,084436	0,056686	0,076888
2	13000	2,542896	2,534707	0,166597	0,202814	0,127510
2	18000	2,124932	2,110889	0,186961	0,215129	0,129145
2	23000	2,237994	1,949952	0,228131	0,287645	0,289931
2	28000	1,952013	1,730564	0,230148	0,308019	0,285531
2	33000	2,402191	2,186102	0,362633	0,462818	0,259251
2	38000	1,587540	1,451468	0,297207	0,394157	0,362420
2	43000	1,530781	1,567709	0,368821	0,486027	0,490784
2	48000	2,093034	2,452750	0,561456	0,703775	0,711169
2	53000	1,959480	1,985200	0,504974	0,625849	0,638028
2	58000	2,243772	2,239115	0,675974	0,892505	0,875095
2	100000	1,731284	1,998180	0,735864	0,851581	0,746933
2	175000	2,261285	2,240408	1,083375	1,363663	1,553539
2	250000	2,207516	2,135077	1,295233	1,650833	1,935149
2	325000	1,916871	1,902564	1,318820	1,724247	1,749642
2	400000	1,534802	1,594194	1,369892	1,734846	1,643743
2	475000	1,361007	1,619016	1,299292	1,631152	1,658817
2	550000	2,148061	2,044756	1,106924	1,895393	1,234151
2	625000	1,391836	1,452137	1,315637	1,389929	1,438449
2	700000	1,382206	1,367695	1,293929	1,434509	1,469238
2	775000	1,769168	1,764572	1,860003	2,058640	1,683302
HYPERTHREADING						
4	100000	1,577836	1,853299	0,578321	0,723271	0,805998
4	175000	2,376492	2,299423	1,107256	1,624433	1,753820
4	250000	1,952191	2,074723	0,950944	1,573093	1,595221
4	325000	1,749904	1,669536	1,017240	1,827909	1,831662
4	400000	1,566122	1,466250	1,233237	1,813515	1,795477
4	475000	1,417551	1,471758	1,507163	1,876042	1,807037
4	550000	1,862169	1,884381	2,001720	1,970697	2,481077
4	625000	1,377995	1,445781	1,481438	1,570155	1,637799
4	700000	1,965608	2,086574	2,279337	2,330487	2,291496
4	775000	1,353648	1,428165	1,481366	1,559550	1,532064

Ilustración 5-6 Ejemplo de recogida de speedup

Cada fila es un experimento diferente con un número de elementos del vector diferente. El número de elementos lo indica el segundo elemento de cada fila y el número de *cores* utilizados el primer elemento. Cada columna almacena los *speedup* de cada tipo de procesamiento a partir de la segunda columna, en este caso, el orden de columnas es el siguiente: SSE – AVX - Paralelo – SSE con paralelo – AVX con paralelo.

Todos los datos obtenidos en los que se realizan operaciones secuenciales son resultado de operaciones con datos alineados en memoria, es decir, los datos de rendimiento en estos casos son los mejores posibles. En el caso de que los datos no estuviesen alineados inicialmente, los

speedups obtenidos tendrían valores menores, ya que se deberían de alinear. En algunos casos la ejecución secuencial sería la mejor opción, debido a que el *speedup* sería menor que 1.

A pesar de trabajar con un procesador con tecnología *HyperThreading* se decidió no utilizar esa tecnología debido al ruido que generaba, es decir, se obtenía una mejora mínima respecto a la utilización de los *cores* físicos. Por ello, todas las pruebas se han realizado mediante los *cores* físicos del procesador, obviando los lógicos y reduciendo así el número de datos obtenidos. En el documento "Resultados de rendimiento" pueden observarse los comportamientos del *HyperThreading*, a su vez también se muestra un ejemplo en la ilustración 5-6.

5.2.1 Función rdtsc

Esta función permite contar el tiempo de CPU consumido entre dos llamadas. En el proyecto se ha utilizado para calcular el tiempo de CPU que consume cada tipo de procesamiento y así obtener el *speedup* respecto al código secuencial de cada una. Cada vez que es llamado guarda un tiempo de CPU, por lo que la resta de la segunda llamada a la función con la primera devuelve el tiempo de CPU transcurrido.

Esta función está realizada en código máquina y es una adquisición del material de la asignatura PAR (Procesadores de alto rendimiento) del curso.

```
static __inline__ unsigned long long rdtsc(void)
{
  unsigned hi, lo;
  __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
  return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}
```

Ilustración 5-7 Función rdtsc

5.3 ANÁLISIS DE INFORMACIÓN RECOGIDA

A partir de los datos obtenidos se realiza un estudio independiente para cada función o subrutina del primer nivel de BLAS. A continuación se muestra un ejemplo de la representación de los mismos.

			SAXPY						
	th	n	sse	avx	fma	paralelo	par-sse	par-avx	par-fma
L1	2	500	3,593933	5,858054	6,225424	0,008947	0,010855	0,009585	0,009678
	2	1000	3,47093	7,341532	6,476331	0,032287	0,034028	0,023531	0,04088
	2	1500	3,808596	7,086323	7,174801	0,039613	0,042528	0,056496	0,056731
	2	2000	3,831235	4,835024	4,384427	0,030184	0,038571	0,043469	0,043573
	2	2500	3,853029	7,007255	7,392729	0,019769	0,080661	0,049034	0,086731
	2	3000	3,910162	7,22449	7,681736	0,053091	0,049505	0,064671	0,057685
	2	3500	3,838203	6,643501	7,149905	0,098587	0,09832	0,071342	0,131485
	2	4000	3,907405	7,466543	7,510242	0,075549	0,07026	0,06578	0,072204
L2	2	8000	3,446677	5,01316	4,884599	0,132473	0,204912	0,208489	0,292735
	2	13000	2,462128	3,043349	2,867716	0,162015	0,183285	0,18608	0,186237
	2	18000	3,32462	4,768234	4,805398	0,23927	0,404397	0,428968	0,4249
	2	23000	3,230987	4,688673	4,431687	0,469081	0,697216	0,723816	0,719696
	2	28000	3,371136	4,663166	4,951251	0,406478	0,967859	1,001255	1,00353
L3	2	33000	2,752683	2,582217	2,217298	0,394488	0,458605	0,469613	0,471983
	2	38000	3,39415	4,435064	5,099894	0,823818	0,764491	1,060792	1,259391
	2	43000	3,14492	4,650795	4,43652	0,798567	1,159943	0,909051	0,785528
	2	48000	3,022529	3,738793	3,951523	0,610108	1,239345	1,283126	1,290586
	2	53000	1,761166	2,153891	2,693956	0,507582	0,760796	0,781711	0,742341
	2	58000	4,03318	4,557199	4,563554	0,671318	0,381733	0,953065	1,164106
	2	100000	3,020203	3,895312	3,970166	1,066293	1,977895	1,662813	1,641814
	2	175000	1,853115	1,96533	1,924354	0,720329	1,464895	1,428867	1,579351
	2	250000	3,117274	3,87064	4,410517	1,802666	2,793951	2,82899	2,835047
	2	325000	3,441095	3,775535	3,864875	2,192616	3,685566	3,507222	2,799502
RAM	2	400000	2,342462	2,556777	2,497869	1,512689	2,25555	2,0243	2,505159
	2	475000	3,017322	3,768753	3,785754	2,334286	3,900793	4,026145	3,824336
	2	550000	3,15128	3,163932	3,096608	1,796305	2,937407	3,073688	3,074921
	2	625000	2,51408	2,839382	2,770727	1,906015	2,912808	3,03343	2,943326
	2	700000	1,982868	2,004225	1,996135	1,312273	2,014373	2,100548	2,197954
	2	775000	2,467659	2,433326	2,483966	1,387114	2,418637	2,25633	2,011255

Ilustración 5-8 Representación de datos obtenidos – 1º configuración - saxpy

Tal y como se muestra en la ilustración 5-8, esta sería la representación de los datos antes mencionados. En filas, el número de elementos con el que se trabaja (siempre el mismo para cada experimento) y el rango de datos de cada tipo de caché. En columnas, todos los tipos de procesamiento utilizados para obtener la misma solución que el procesamiento secuencial.

Una vez que se tienen los datos ordenados y claramente visibles, es necesario realizar un análisis de los mismos para seleccionar situaciones donde es más viable no realizar procesamiento en serie. Este tipo de situaciones en la imagen de ejemplo se representan con colores diferentes.

El color amarillo más oscuro determina la mejor elección en cuanto a *speedup* se refiere y el degradado indica el orden de las siguientes opciones en el caso de que no se precise de esa tecnología.

El color morado, indica la escala donde se comienza a conseguir rendimiento utilizando procesamiento entre más de un *core*, es decir, a partir de qué punto el tiempo de carga entre *threads* empieza a ocultarse.

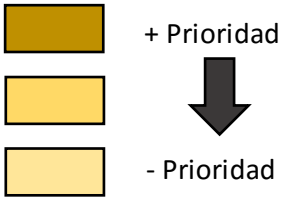


Ilustración 5-9 Prioridad de colores

Este tipo de selección servirá para realizar una rutina de decisión que se explicará detenidamente en el próximo capítulo.

Tal y como se muestra en el documento "Resultados de rendimiento", también se han realizado tablas sobre los otros tipos de configuraciones. A continuación se muestran los resultados obtenidos de las configuraciones restantes sobre la misma función *saxpy* del primer nivel de BLAS.

TH	N	sse	avx	fma	omp	omp-sse	omp-avx	omp-fma
4	500	3.576288	6.062603	6.312178	0.007498	0.007416	0.010777	0.010820
4	1000	3.411905	5.950997	6.335102	0.017586	0.019039	0.018058	0.017209
4	1500	6.653014	13.178760	13.266268	0.041226	0.040276	0.040206	0.038363
4	2000	3.851035	7.407543	7.426996	0.030173	0.006711	0.027805	0.019076
4	2500	3.921508	7.384551	7.366097	0.037845	0.039953	0.038594	0.039351
4	3000	3.942490	7.584318	7.622526	0.047542	0.045545	0.051349	0.047353
4	3500	3.902547	7.585486	7.682342	0.053143	0.056997	0.056653	0.057327
4	4000	3.860215	7.471286	2.512936	0.048061	0.056233	0.039780	0.049940
4	8000	2.831294	3.072308	3.068234	0.109340	0.119421	0.125626	0.110809
4	13000	3.000649	3.406002	3.380520	0.183139	0.186623	0.168737	0.178057
4	18000	3.053318	3.513367	3.502847	0.277241	0.230705	0.238921	0.225120
4	23000	2.891756	3.335553	3.363028	0.313727	0.289317	0.317980	0.404954
4	28000	2.724041	3.133128	3.117115	0.294782	0.342314	0.388711	0.393368
4	33000	2.798868	3.010794	3.179151	0.327548	0.443266	0.438944	0.448975
4	38000	2.388118	2.575393	2.591576	0.423548	0.535917	0.457156	0.434625
4	43000	2.789482	3.130293	3.114701	0.547469	0.588258	0.618263	0.615084
4	48000	2.554068	2.721332	2.712594	0.557556	0.521307	0.425034	0.686159
4	53000	2.574475	2.689611	2.750101	0.591329	0.561987	0.630998	0.686191
4	58000	2.975657	3.276720	3.208216	0.782420	0.914592	0.772575	0.743620
4	100000	2.164882	2.394022	2.413550	0.845713	1.081701	1.047792	1.067620
4	175000	2.793776	2.902671	2.883505	1.308967	2.121300	2.223630	2.171225
4	250000	2.435407	2.569648	2.554842	1.118140	1.947859	2.053535	0.395483
4	325000	2.319982	2.417456	2.400034	1.376175	2.459367	2.447678	2.444579
4	400000	2.343044	1.749163	2.361075	1.461517	2.593670	2.896228	2.355990
4	475000	2.443886	2.526135	2.505522	1.774469	3.175569	3.067886	2.910725
4	550000	3.130081	2.993391	2.873331	1.916471	3.492293	2.659368	4.190927
4	625000	2.392140	2.697657	2.829993	1.803816	3.846623	3.665556	3.682741
4	700000	1.785939	1.626708	1.964800	1.708977	3.038862	2.872965	2.773879
4	775000	1.605662	1.759241	2.107361	1.821027	3.064414	3.679651	4.067634
4	1600000	1.928259	1.843286	1.904007	2.531070	3.347168	3.738216	3.556301

Ilustración 5-10 Representación de datos obtenidos – 2ª configuración - saxpy

	th	n	sse	paralelo	par-sse
L1	4	500	2,021705	0,005509	0,005437
	4	1000	2,474368	0,011574	0,013317
	4	1500	2,650934	0,022727	0,023692
	4	2000	2,730359	0,02358	0,024372
	4	2500	2,813543	0,029135	0,032966
	4	3000	2,861037	0,029653	0,030442
	4	3500	2,907459	0,040141	0,042175
	4	4000	2,816601	0,043977	0,047946
L2	4	8000	2,356601	0,041957	0,048546
	4	13000	2,197173	0,129728	0,122965
	4	18000	2,225746	0,179019	0,184942
	4	23000	2,239602	0,247673	0,2588
	4	28000	2,15358	0,25713	0,268972
	4	33000	2,259089	0,317847	0,325635
	4	38000	2,260986	0,358024	0,46543
	4	43000	2,784079	0,459927	0,580361
	4	48000	2,220798	0,343682	0,362723
	4	53000	2,272941	0,371926	0,413094
	4	58000	2,265926	0,439801	0,606873
	4	100000	2,288774	0,516001	0,571217
	4	175000	2,85961	0,573964	0,970687
	4	250000	2,256898	0,640448	0,797279
	4	325000	1,741555	0,731726	0,770394
	4	400000	1,797613	0,598472	0,76374
	4	475000	2,110709	0,557419	0,714175
	4	550000	2,576886	0,632327	0,834199
	4	625000	1,012434	0,765525	0,937354
	4	700000	1,445857	0,748619	0,780304
RAM	4	775000	1,193545	1,289943	1,546743

Ilustración 5-11 Representación de datos obtenidos – 3ª configuración - saxpy

6 CAPÍTULO: CRITERIO DE DECISIÓN

Después de obtener todos los datos y escoger cuales son las mejores elecciones de ejecución para cada función y subrutina del primer nivel de BLAS, es necesario implementar una estructura de decisión que compruebe los recursos que dispone el procesador y ejecute la mejor opción dependiendo de ellos. Básicamente se transformaría el análisis de la tabla de cada función en una rutina de decisión.

6.1 OCULTACIÓN

Antes de comenzar con la creación de la rutina de decisión, es necesario modificar todas las funciones de procesamiento de dato generadas para cada función y subrutina del primer nivel de BLAS.

Para hacerlo, se debe añadir a cada función la palabra "*static*". De esta manera, esas funciones solo pueden ser accedidas por funciones que se encuentren en el mismo fichero. Por ello en el momento que se tiene la necesidad de llamar a una función con *static*, se debe de incluir todo el fichero donde se encuentra mediante *#include <fichero_con_funciones_static.c>*. Si se induye de esa manera el código es copiado al fichero aumentando el peso, pero se mantiene oculta su implementación.

Las funciones quedarían de esta forma:

*static void saxpy_sse(int n, float a, float *x, float *y){…}*

Esto no se hace en las funciones secuenciales y paralelas, debido a que no deben de compilarse junto con los *flags* vectoriales para no ser modificadas por el compilador y así obtener datos fiables.

6.2 CPU –ID

En las comprobaciones de datos, se tenía información previa de las instrucciones SIMD del procesador donde se realizaron las pruebas y en el fichero de compilación se indicaban los *threads* a utilizar, pero en realidad esa información es dependiente para cada procesador y es necesario obtenerla de algún modo en tiempo de ejecución.

Para la obtención de esos datos, se ha utilizado una rutina creada por el tutor del proyecto, llamada "*cpuid*" que recoge la información hardware de la que dispone el sistema. Esta rutina permite obtener los datos necesarios para realizar una elección de ejecución dependiendo de los recursos que se dispongan, ilustración 6-1.

La rutina ha sido implementada en lenguaje C, utilizando instrucciones en ensamblador proporcionadas por Intel.

Como la rutina devolvía demasiados valores que eran inservibles para el proyecto, se creó una función que tenía como objetivo rellenar una serie de variables para obtener la información sobre los *threads* y las instrucciones SIMD que disponía el procesador.

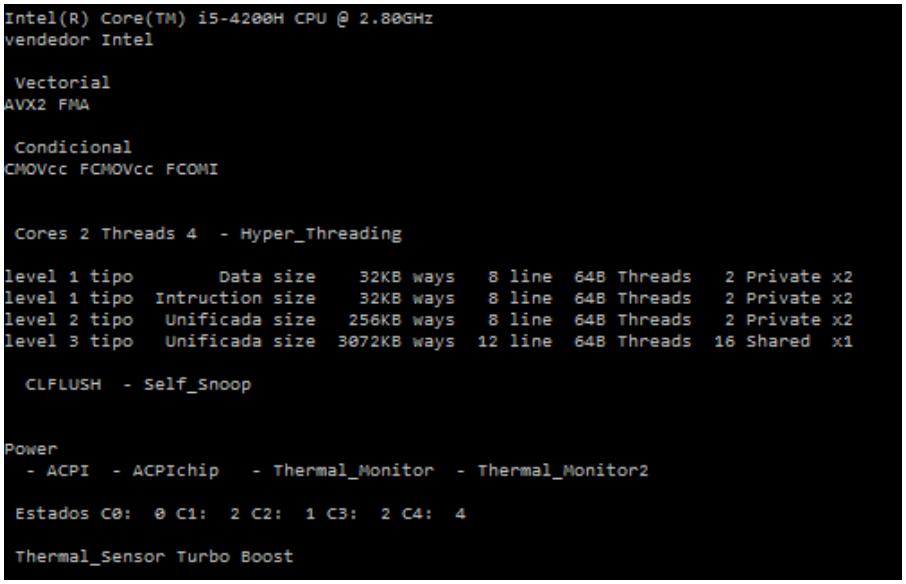


Ilustración 6-1 Datos devueltos por la función cpuid

Los datos utilizados por la función de la ilustración 6-2 están recogidos en un fichero .h, al cual, todas las funciones tienen acceso a él.

```
#include"cpuid.h"
static void set_cpu_info(){
   Tipo_Cpuid t;
   cpuid_address_size(&t);
   threads = t.cores;
   fmap = t.fma;
   if(t.vectorial == 7) avx = 1;
   if(t.vectorial == 8) avx = 2;
}
```

Ilustración 6-2 Función de obtención de datos necesarios cpuid

Para no recoger la misma información por cada llamada a una función del primer nivel de BLAS, se ha creado una variable de control que dependiendo de su estado indica si es necesario llamar a la rutina "*cpuid*" o no, la variable es *first*. Por ello, la variable de control debe ser compartida por todas las funciones y subrutinas del BLAS. La información necesaria para realizar preguntas en la rutina de decisión se almacenan en las variables *avx*, *fma* y *threads*. El código que se muestra en la ilustración 6-3 se encuentra dentro de una de las funciones que selecciona el tipo de ejecución a realizar, con el objetivo de declarar e inicializar las variables.

```
int NTHR;
int first = 0;
int threads = 1;
int fma = 0;
int avx = 0; //1=avx, 2=avx2
```

Ilustración 6-3 Declaración e inicialización de variables - fichero dasum.c

Para que el resto de funciones de selección puedan modificar los datos de las variables mencionadas en la ilustración 6-4, es necesario incluir esas variables como externas en las demás, ilustración 6-3.

```
extern int NTHR;
extern int first;
extern int threads;
extern int fmap;
extern int avx; //1=avx, 2=avx2
```

Ilustración 6-4 Declaración externa de variables compartidas – var.h

La estructura utilizada para controlar si se han obtenido los datos del procesador o no se muestra en la ilustración 6-5.

```
if(first == 0){
    set_cpu_info();
    first = 1;
  }
```

Ilustración 6-5 Control de recogida de datos del procesador

6.3 CONDICIONES

Las condiciones que se han creado para representar la estructura de decisión se basa en la elección de ejecución dependiendo de los recursos de los que disponga la máquina y el número de elementos que se hayan introducido en la llamada. La selección de ejecución se realiza mediante la tabla de datos mencionada en el capítulo anterior en la sección 5.3.

```
#include "var.h"
#include <omp.h>
#include "cpu_info.c"

extern void saxpy_seq(int n, float a, float *x, float *y);
extern void saxpy_omp(int n, float a, float *x, float *y);
#include "..\_axpy\float\saxpy_omp_vec.c"

void saxpy(int n, float a, float *x, float *y)
{
  if(first == 0){
    set_cpu_info();
    first = 1;
  }

  if (n <= 0 || a == 0) return;

  if (n <= 4000){
    if (fmap) saxpy_fma(n,a,x,y);
    else if (avx) saxpy_avx(n,a,x,y);
    else saxpy_sse(n,a,x,y);

  }else if (n <= 33000){
    if (avx) saxpy_avx(n,a,x,y);
    else saxpy_sse(n,a,x,y);

  }else if (n <= 325000){
    if (fmap) saxpy_fma(n,a,x,y);
    else if (avx) saxpy_avx(n,a,x,y);
    else saxpy_sse(n,a,x,y);
```

```
  }else{
    NTHR = my_cpu.threads;
    omp_set_num_threads(NTHR);
    if (fmap) saxpy_omp_fma(n,a,x,y);
    else if (avx) saxpy_omp_avx(n,a,x,y);
    else saxpy_omp_sse(n,a,x,y);
  }
}
```

Ilustración 6-6 Estructura de decisión para la función saxpy

Primero, se comprueba el número de datos con el que se va a realizar la operación y a continuación algunos datos referentes a las tecnologías que implementa el procesador que se van a usar.

Por último, se realiza la ejecución de la función dependiendo de la estructura de decisión seguida.

En este proyecto, las estructuras de decisión han supuesto la existencia de instrucciones vectoriales, es decir, al menos existen instrucciones SSE. Esto no sería correcto si se realizará un análisis más profundo con más configuraciones que careciesen de algunas tecnologías mencionadas, pero en este caso para reducir la complejidad de la estructura se ha seguido esa suposición.

6.4 VERIFICACIÓN DE LA LIBRERÍA

Después de realizar todas las estructuras de decisión para cada función del primer nivel de BLAS, era necesario verificar su correcto funcionamiento. Por ello se ha decidido crear un pequeño programa que utilice todas las rutinas de decisión generadas y compruebe su *speedup* respecto al secuencial, tal y como se obtuvo inicialmente en el apartado de medición.

Adicionalmente, para comprobar el acceso a cada condicional empleado, se ha replicado la rutina de decisión inicial, mostrando por pantalla un texto de lo que se ejecutaría en cada caso. A lo largo de esta función son modificadas varias variables antes mencionada. En ella se va limitando progresivamente las tecnologías de las que se dispone para poder mostrar el comportamiento en las decisiones tomadas.

```
void dasum2(int n)
{
  set_cpu_info();

  printf("normal \n",n);

  if (n <= 0) printf("no entra \n");

  if (n <= 250000){
    if (my_cpu.avx) printf("%d entra en avx \n",n);
    else printf("%d entra en sse \n",n);

  }else{
    if (avx) printf("%d entra en avx \n",n);
    else if (threads % 2 == 0) printf("%d entra en sse omp 2 \n",n);
    else printf("%d entra en sse \n",n);
  }
```

```
    set_cpu_info();
    printf("\n");
    printf("sin avx \n",n);
    avx = 0;

    if (n <= 0) printf("no entra \n");
    if (n <= 250000){
      if (avx) printf("%d entra en avx \n",n);
      else printf("%d entra en sse \n",n);

    }else{
      if (avx) printf("%d entra en avx \n",n);
      else if (threads % 2 == 0) printf("%d entra en sse omp 2 \n",n);
      else printf("%d entra en sse \n",n);
    }

}
```

Ilustración 6-7 Función de comprobación para la función de selección dasum

Como puede observarse, en la ilustración 6-7, no se realiza ningún tipo de llamada a las funciones anteriormente implementadas. Esta función indica en que parte de la rutina de decisión se entra, limitando los recursos SIMD.

El programa de verificación se aprovecha de esta función para poder mostrar las diferentes opciones que se han implementado. Principalmente, este programa lo que hace es, ejecutar secuencialmente una función, después ejecutarla mediante la estructura de decisión, devolver el *speedup* conseguido y por último llamar a la función mostrada en la ilustración 6-6 para poder saber que ha hecho y así poder ser verificada correctamente.

La compilación secuencial y paralela tiene que hacerse independientemente de los *flags* vectoriales, tal como se explica en la sección 5.2 del documento, en caso contrario el propio compilador vectorizaría el código y se obtendrían *speedups* erróneos.

Para la creación de la librería sería necesario incluir todos los ficheros .o de salida, para su correcto funcionamiento, ilustración 6-8.

```
"C:\MinGW\bin\gcc.exe"    -O3       -c ..\_asum\double\dasum_seq.c
"C:\MinGW\bin\gcc.exe"    -O3 -fopenmp     -c ..\_asum\double\dasum_omp.c
"C:\MinGW\bin\gcc.exe"    -O3       -fopenmp -msse4 -mavx2 -c fdasum.c
"C:\MinGW\bin\gcc.exe"    -O3       -c ..\_axpy\double\daxpy_seq.c
"C:\MinGW\bin\gcc.exe"    -O3 -fopenmp     -c ..\_axpy\double\daxpy_omp.c
"C:\MinGW\bin\gcc.exe"    -O3       -fopenmp -msse4 -mavx2 -mfma -c   fdaxpy.c
"C:\MinGW\bin\gcc.exe"    -O3       -c ..\_copy\double\dcopy_seq.c
"C:\MinGW\bin\gcc.exe"    -O3 -fopenmp     -c ..\_copy\double\dcopy_omp.c
"C:\MinGW\bin\gcc.exe"    -O3       -fopenmp -msse4 -mavx2 -c fdcopy.c
"C:\MinGW\bin\gcc.exe"    -O3       -c ..\_dot\double\ddot_seq.c
"C:\MinGW\bin\gcc.exe"    -O3 -fopenmp     -c ..\_dot\double\ddot_omp.c
"C:\MinGW\bin\gcc.exe"    -O3       -fopenmp -msse4 -mavx2 –mfma -c   fddot.c
"C:\MinGW\bin\gcc.exe"    -O3       -c ..\_dot\double-float\dsdot_seq.c
"C:\MinGW\bin\gcc.exe"    -O3 -fopenmp     -c ..\_dot\double-float\dsdot_omp.c
"C:\MinGW\bin\gcc.exe"    -O3       -fopenmp -msse4 -mavx2 –mfma -c   fdsdot.c
"C:\MinGW\bin\gcc.exe"    -O3       -c ..\_i_amax\double\di_amax_seq.c
"C:\MinGW\bin\gcc.exe"    -O3 -fopenmp     -c ..\_i_amax\double\di_amax_omp.c
"C:\MinGW\bin\gcc.exe"    -O3       -fopenmp -msse4 -mavx2 -c fdi_amax.c
…
```

Ilustración 6-8 Extracto de compilación de funciones de selección de ejecución

Después de incluir todos los ficheros, se crean las funciones de inicialización de vector necesarias para cada tipo de dato, número de vectores o en algunos casos como "rotm", personalizadas, ilustración 6-9.

```
void inicializarxyf (int n, float *x, float *y){
   int i;
   for (i=0; i<n; i++){ //Inicializar
      x[i]= (float)i/10;
      y[i]= (float)i/10;
   }
}
…
void inicializarxd (int n, double *x){
   int i;
   for (i=0; i<n; i++){ //Inicializar
      x[i]= (double)i/10;
   }
}
…
void inicializarxyi_rotm (int n, int *x, int *y, int * param){
   int i;
   for (i=0; i<n; i++){ //Inicializar
      x[i]= (int)i/10;
      y[i]= (int)i/10;
   }
   param[0] = 4;
   param[1] = 2;
   param[2] = 1;
   param[3] = 3;
   param[4] = 1;
}
```

Ilustración 6-9 Inicializaciones de función de verificación

Seguidamente se crea el *main* del programa y se declaran las variables necesarias, ilustración 6-10.

```
void main(int argc, char * argv[]){
   int opcion, n, i;
   static unsigned long long fin, ini, dif, sum, dif_seq;   Variables de medición
   float auxf;
   double auxd;                                             Variables para funciones con salida
   int auxi;

   float * xf;
   float * yf;
   double * xd;                                             Variables dinámicas para los vectores
   double * yd;
   int * xi;
   int * yi;

   float paramf[5];
   double paramd[5];                                        Variables para la función rotm
   int parami[5];
      …
```

Ilustración 6-10 Declaración de variables de función de verificación

A continuación, se crea el menú de opciones con cada tipo de función del primer nivel del BLAS. Para ello, después de mostrarlo, se pide por teclado la opción escogida y el número de elementos con los que se quiere trabajar (con su correspondiente control), ilustración 6-11.

```
1.saxpy         2.daxpy         3.iaxpy
4.scopy         5.dcopy         6.icopy
7.srot          8.drot          9.irot
10.srotm        11.drotm        12.irotm
13.sscal        14.dscal        15.iscal
16.sswap        17.dswap        18.iswap
19.sasum        20.dasum        21.iasum
22.sdot         23.ddot         24.dsdot        25.sdsdot       26.idot
27.snrm2        28.dnrm2
29.si_amax      30.di_amax      31.zi_amax      32.EXIT
Introducir funcion a realizar:
0
 *** La opcion seleccionada no existe ***
Introducir funcion a realizar:
33
 *** La opcion seleccionada no existe ***
Introducir funcion a realizar:
1
Numero de elementos de vector (>0):
0
 *** El numero de datos es incorrecto ***
Numero de elementos de vector (>0):
```

Ilustración 6-11 Impresión y petición de información al usuario

Por último, se completa cada tipo de opción con la estructura antes mencionada, ilustración 6-12.

- Inicialización de los datos y ejecución secuencial de la función. Ejecutada once veces, despreciando la primera de ellas, y realizando una media para obtener el tiempo de CPU obtenido.
- Inicialización de los datos y ejecución escogida por la rutina de decisión de la función. Ejecutada once veces, despreciando la primera, y realizando una media para obtener el tiempo de CPU obtenido.
- Llamada a la función de la ilustración 6-6, donde solo se muestra aquello en lo que se entra con determinadas condiciones.
- Impresión del *speedup* obtenido mediante el tiempo de CPU de la ejecución secuencial y la seleccionada por la rutina de decisión.

```
switch(opcion){
        case 1:
          for (sum=0, i=0; i<11; i++){
            inicializarxyf(n,xf,yf);
            ini = rdtsc();
            saxpy_seq(n,2.5,xf,yf);
            fin = rdtsc();
            if(i>0){
              dif_seq = (fin - ini);
              sum += dif_seq;
            }
          }
          dif_seq = sum/10.0;

          for (sum=0, i=0; i<11; i++){
```

```
            inicializarxyf(n,xf,yf);
            ini = rdtsc();
            saxpy(n,2.5,xf,yf);
            fin = rdtsc();
            if(i>0){
              dif = (fin - ini);
              sum += dif;
            }
          }
          dif = sum/10.0;

          saxpy2(n);
          printf("Speedup = %f \n",(float)dif_seq/(float)dif);
          break;
        case 2:
          ...
```

Ilustración 6-12 Estructura de selección de opción del programa de verificación

A continuación, se muestra una respuesta del programa, ilustración 6-13, junto con el resultado obtenido en la tabla de medición para poder verificar su funcionamiento. Los datos pueden no ser los mismos.

	th	n	sse	avx	fma
L1	2	500	3,593933	5,858054	6,225424
	2	1000	3,47093	7,341532	6,476331
	2	1500	3,808596	7,086323	7,174801
	2	2000	3,831235	4,835024	4,384427
	2	2500	3,853029	7,007255	7,392729
	2	3000	3,910162	7,22449	7,681736
	2	3500	3,838203	6,643501	7,149305
	2	4000	3,907405	7,466543	7,510242
L2	2	8000	3,446677	5,01316	4,884599
	2	13000	2,462128	3,043349	2,867716
	2	18000	3,32462	4,768234	4,805398
	2	23000	3,230987	4,688673	4,431687
	2	28000	3,371136	4,663166	4,951251
L3	2	33000	2,752689	2,582217	2,217298
	2	38000	3,39415	4,435064	5,099894
	2	43000	3,14492	4,650795	4,43652
	2	48000	3,022529	3,738793	3,951523
	2	53000	1,761166	2,153891	2,693956
	2	58000	4,03318	4,557199	4,563554
	2	100000	3,020203	3,895312	3,970166
	2	175000	1,853115	1,96533	1,924354
	2	250000	3,117274	3,87064	4,410517
	2	325000	3,441095	3,775535	3,864875

Ilustración 6-13 Ejemplo de ejecución de programa de verificación

El primer resultado obtenido por el programa de verificación es similar al obtenido en las tablas, pero una vez que se realizan más verificaciones se puede observar que el *speedup* obtenido no se parece al de las tablas. Esto se debe a que inicialmente los datos se encuentran alineados y se obtiene un resultado parecido al ya calculado. En cambio, las siguientes verificaciones pueden no tener alineado los datos, por lo que requiere de mayor cómputo alinearlos con motivo de realizar la operación, obteniendo así un *speedup* menor y diferente al registrado, ilustración 6-14.

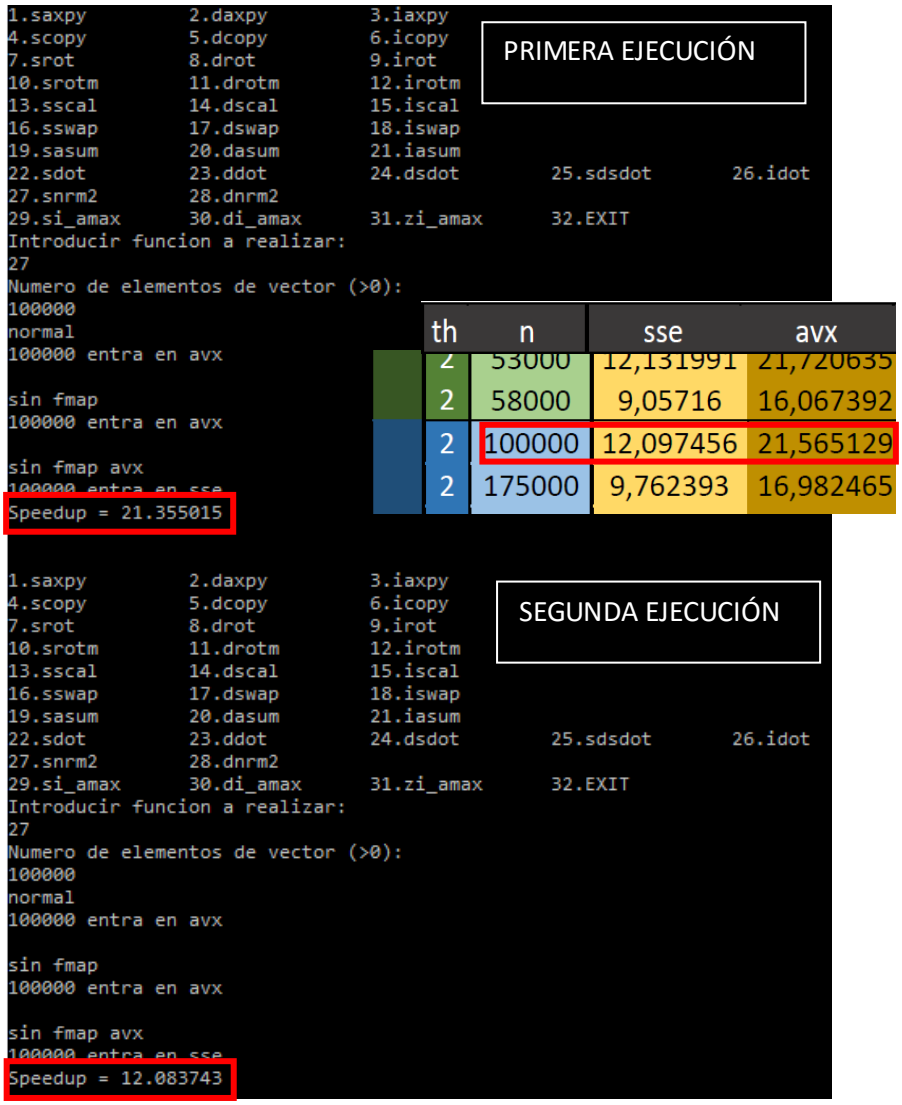


Ilustración 6-14 Ejecución ininterrumpida – snrm2 -100000 elementos

7 CAPÍTULO: CREACIÓN DE LIBRERÍA

Este tipo de rutinas que se han implementado, originariamente se recogían a modo de librería para ciertos lenguajes. En este caso, se ha decidido hacer lo mismo, concretamente para el lenguaje C, donde las funciones de las que se compondrá serán el conjunto de funciones que recojan la decisión de ejecución tal y como se ha visto en el capítulo anterior.

Las librerías son archivos que pueden incluirse o importarse en nuestro programa. Estos archivos se componen de especificaciones ya construidas y utilizables en cualquier programa donde se incluyan. Estas especificaciones son archivos en código objeto o archivos terminados en extensión ".o".

El uso de librerías permite mejorar la modularidad y reutilización de los códigos, eliminando la necesidad de generar funciones de todo tipo, en el propio fichero, para su posterior uso.

La declaración de cabeceras de funciones incluidas en las librerías, tanto en C como en C++ se realiza con la siguiente sintaxis: *#include <nombre_de_librería.h> o #include "nombre_de_librería.h"*. Es posible declarar tantas librerías como se quiera, pero no tiene sentido si no va a utilizarse. En el caso de tratar con librerías externas es necesario compilar el ejecutable con ellas, añadiendo las cabeceras del programa que recogen las funciones de la librería para poder ser llamadas.

```
#include "blas1.h"
#include <stdio.h>
```

Existen dos tipos de librerías que pueden utilizarse, estática y dinámica. En este proyecto se han realizado ambos tipos de librería e incluso un ejecutable sin librería. Ahora se describirán y se mostrará su creación.

7.1 ESTÁTICA

Las librerías estáticas son aquellas que "se copian" al programa ejecutable cuando se compila. Una vez obtenido el ejecutable, la librería ya no tiene influencia sobre él, por lo que solo serviría para proyectos independientes. Si se borrara después de compilar, el programa seguiría funcionando.

La creación de una librería estática es básicamente un conjunto de archivos objeto que se copiarán en un solo fichero. Ese fichero independiente es la librería estática y se genera con el comando (*ar*) que permite archivar.

El siguiente comando genera una librería estática llamada libejemplo.a (es necesario que empiece por *lib* y su extensión sea *.a*) e incluye tres archivos objeto ficticios llamados test1.o, test2.o y test3.o en la librería. Si la librería ya existe, estos tres objetos se incluyen a la librería existente. Si la librería ya existe y contiene alguno de los objetos, los objetos repetidos se sobrescriben por los nuevos introducidos.

```
ar rcs libejemplo.a test1.o test2.o test3.o
```

- **r:** añadir el archivo objeto en la librería, r indica la reescritura del archivo en caso de que exista teniendo el mismo nombre.
- **c:** crea la librería si no existe.

- **S:** mantiene la tabla de nombres de símbolos mapeada a los nombres de fichero de los archivos objeto.

Una vez creada la librería mediante el comando: *ar –t libejemplo.a* es posible listar todos los archivos objeto de los que se compone para poder comprobar que están todos.

Para poder compilar, se pueden encontrar dos situaciones. La primera situación y más habitual después de crear la librería, es que la ubicación de la librería sea el mismo directorio o un directorio cercano a los archivos objetos de los que se compone. Si esto es así para poder compilar un fichero es necesario utilizar el *flag* –L seguido de la ubicación de la librería estática e introducir el nombre de la librería sustituyendo el comienzo *"lib"* por un *"l"*. Este puede ser un ejemplo de compilación de la librería antes creada.

gcc main.c –L[direccion_de_libreria] –lejemplo –o libreria_estatica

La segunda situación es que la ubicación de la librería se encuentre junto a las demás librerías del sistema, donde el propio sistema operativo al realizar la compilación busca su existencia. Debido a ello, el comando para compilar seria el mismo sin la necesidad del *flag* –L.

7.2 DINÁMICA

Las librerías dinámicas son aquellas que su contenido no se copia al programa ejecutable al compilarlo. Cuando el código del programa necesite algo de la librería, ira a búscalo a esta. Si se borra la librería, nuestro programa indicara con un error que no puede localizarla.

La creación de una librería dinámica o compartida, al igual que la estática, es básicamente un conjunto de archivos objeto. En este caso no se copiarán, si no que se asignarán a diferentes ficheros, para que cuando se realice una llamada este cargue la función del propio fichero. El fichero de salida después de la creación será la librería dinámica. En este caso, los archivos objeto pueden necesitar el *flag –fPIC* en su creación debido a que para crear librerías dinámicas es necesario que todos los archivos objeto generados tengan posiciones independientes de código.

gcc –c –fPIC main.c

Al compilar puede mostrar un mensaje de advertencia como este:

```
cc1: warning: -fPIC ignored for target (all code is position independent)
```

Este mensaje advierte que no es necesario utilizar el *flag –fPIC,* porque la familia de microprocesadores x86 ya lo solucionan. De todas formas, el manual dice que es necesario ponerlo.

El siguiente comando genera una librería dinámica llamada libejemplo.dll, como en el ejemplo anterior. En este caso también es necesario que empiece por *lib* y su extensión sea *.so o .dll*. Incluye tres archivos objeto ficticios llamados test1.o, test2.o y test3.o en la librería.

gcc -shared -o libejemplo.dll test1.o test2.o test3.o

Para poder compilar es necesario incluir en el comando la librería con su extensión.

gcc main.c –o ejecutable libejemplo.dll

7.3 DIFERENCIAS

Existen ciertas ventajas entre los dos tipos de librería existentes. En la siguiente tabla se podrán observar sus diferencias:

Librería estática	Librería dinámica
El programa resultante una vez compilado es más grande, ya que hace copia de todo lo que necesita.	El peso del programa resultante es mucho menor, debido a que no hace copia de todo lo que necesita. Ahorra espacio en disco.
El programa resultante puede ser portado a otro ordenador sin necesidad de mover las librerías.	Más de un proceso puede aprovechar un mismo archivo de librería dinámica.
La ejecución es más rápida, ya que la función de librería la tiene en su código y no tienen necesidad de cargarlo	Permite la creación de versiones y a su vez cargas alternativas entre lenguajes.
Si se modifica una librería estática a los ejecutables no les afecta.	Si se modifica una librería dinámica los ejecutables pueden ser modificados, para por ejemplo corregir un error.

Tabla 7-1 Diferencias entre los tipos de librerías

La elección del tipo de librería depende de cada persona y sus necesidades. Las librerías estáticas se utilizan con ejecutables de poco peso o ejecutables que no van a ser modificados pero que pueden moverse entre terminales. En cambio, para ejecutables grandes está pensada las librerías dinámicas, ya que reduce gran parte del peso.

7.4 INFORMACIÓN A TENER EN CUENTA

Para la utilización de librerías, tanto dinámicas como estáticas, GCC puede ser un problema. Esto se debe a la versión del compilador con la que se ha creado la librería. Una librería creada bajo una versión GCC puede ser utilizada por otra máquina que tenga la misma versión, en caso contrario falla y es necesario volver a compilar la librería.

La creación de librerías propuesta en el proyecto compila los ficheros vectoriales con la solución AVX (y en algunos casos FMA) a las funciones del primer nivel de BLAS. Debido a esta compilación, aquellas maquinas que carecen de esta tecnología no pueden utilizar la librería, ya que cuando se compila junto a otros ficheros el compilador muestra errores indicando que no puede solucionar instrucciones AVX (o FMA). Para ello, es necesario crear una nueva librería sin esos ficheros vectoriales.

8 CAPÍTULO: CONSUMO

Debido a la importancia que hoy en día tiene el consumo en todo tipo de máquinas y de computación, en este proyecto se ha decidido elaborar un análisis de algunos comportamientos que pueden observarse a partir de los programas estudiados. Los resultados de consumo están en principio en consonancia con el *speedup* obtenido en la parte de medición. Cuanto más *speedup* se consiga mayor ahorro energético se debería de conseguir, hablando teóricamente. A continuación, se analiza la relación entre eficiencia y consumo utilizando la primera configuración mencionada en el capítulo 3.

8.1 SELECCIÓN DE FUNCIONES

Para la realización de este apartado, no se han utilizado todas las funciones del primer nivel del BLAS, debido a que nos ha parecido que no era necesario realizar este estudio en funciones que tienen un comportamiento similar en rendimiento. Se han seleccionado tres de entre todas las funciones disponibles, con sus respectivas variantes de tipo de dato (int, float y double). Estas funciones son las siguientes:

- **_axpy**: debido a que es la función más característica del primer nivel del BLAS y tiene datos que obtienen una ganancia mínima en algunos casos y aceptable en otros.

- **_i_amax y _nrm2**: debido a que estas funciones son las que obtienen un mayor rendimiento respecto al cálculo secuencial respectivamente.

8.2 JOULEMETER

Como ya se ha mencionado anteriormente, el software que se ha utilizado para desarrollar la parte de consumo del proyecto es *Joulemeter*. Se trata de una herramienta creada por Microsoft que permite realizar las mediciones sobre el gasto energético de una aplicación y almacenarlo en una tabla para su posterior análisis. En este proyecto se ha utilizado para ver el comportamiento energético en los diferentes tipos de procesamientos de datos (paralelo, vectorial y secuencial).

Ilustración 8-1 Icono de Joulemeter (Microsoft Research)

8.2.1 Funcionamiento

Antes de comenzar a explicar el proceso de obtención de datos y su análisis es necesario mostrar el funcionamiento de esta herramienta. Una vez instalado y ejecutado el programa de libre distribución, se muestra una ventana con tres pestañas en las que pone "*Calibration*", "*Power Usage*" y "*About*". Primero se debe acceder a la pestaña de calibrado para obtener los datos de consumo del ordenador con el que se pretende trabajar.

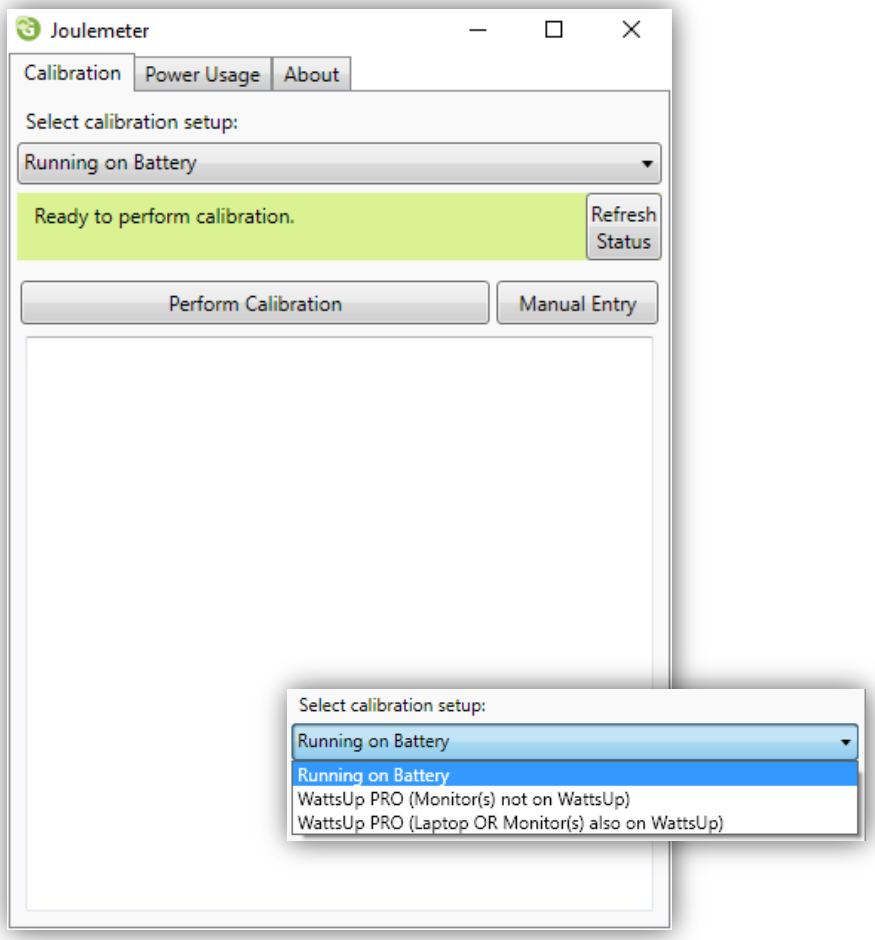


Ilustración 8-2 Calibración Joulemeter

Para ello, tal y como se muestra en la ilustración 8-2, es necesario indicar en qué condiciones energéticas se está trabajando. Las opciones se muestran en la parte inferior de la imagen y dependiendo si se está trabajando con batería o con alimentación externa se ha de seleccionar una de las opciones.

En este proyecto solo se ha trabajado con batería, ya que la maquina con la que se ha trabajado es un ordenador portátil. La medición en ordenadores de sobremesa requiere un dispositivo externo que permite obtener los datos necesarios (*WattsUp*).

Para realizar la calibración por batería, esta tiene que estar al menos al 50%. Cuando este todo preparado, se pulsa el botón "*Refresh Status*" por precaución, y si el mensaje que muestra por pantalla es "*Ready to perform calibration*", ya podría pulsarse el botón "*Perform Calibration*" para comenzar con la calibración. Es muy importante que no se utilice el ordenador hasta que termine. Muestra un tiempo estimado de finalización para el usuario.

Una vez calibrado, se accede a la pestaña de "*Power Usage*" y se observa el contenido de la parte superior, donde indica el consumo en vatios de cada componente, ilustración 8-3.

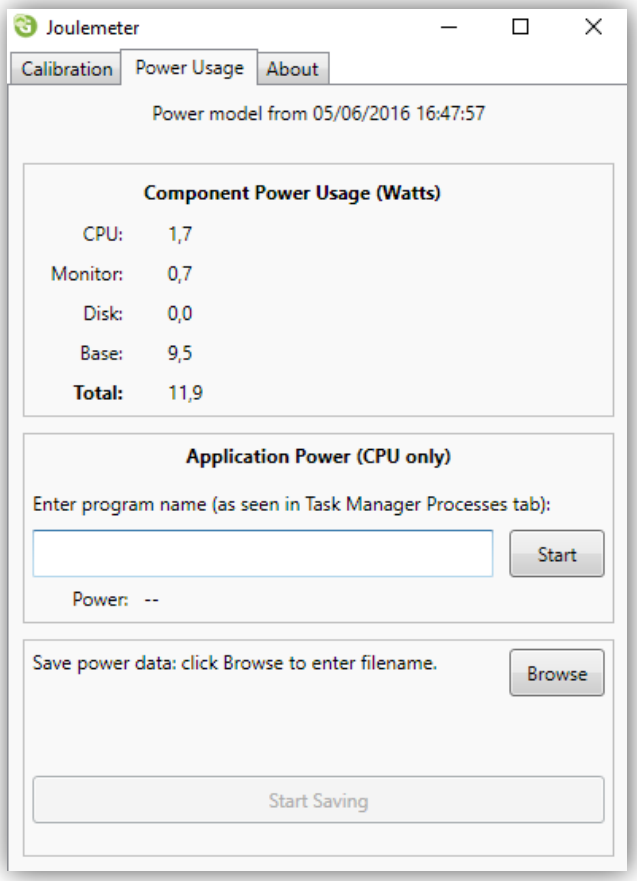


Ilustración 8-3 Resultado de calibración Joulemeter

Una vez que se llega a este punto, se puede comenzar con la obtención de los datos.

8.2.2 Obtención de datos

Para obtener los datos energéticos hay que realizar varios pasos previos.

Primero es necesario rellenar los datos necesarios en la herramienta *Joulemeter*, para indicar que aplicación tiene que medir.

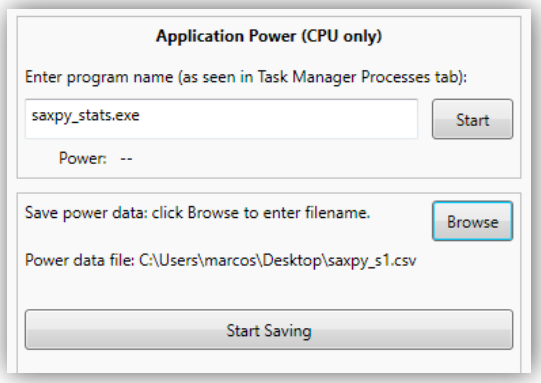


Ilustración 8-4 Ejemplo de Joulemeter

Como se muestra en la ilustración 8-4 hay que rellenar varios campos para comenzar a medir. En el primer campo se indica la aplicación o el nombre de proceso que *Joulemeter* tiene que medir y en el segundo se indica la ruta del sistema donde se van a guardar las mediciones obtenidas.

Para iniciar la captura de datos, primero hay que pulsar el botón "*Start Saving*", después el botón "*Start*" donde se muestra el nombre de proceso de la aplicación y por último ejecutar la aplicación para poder ser medida. Al finalizar la ejecución es necesario volver a pulsar los dos botones para que el programa deje de almacenar datos.

Es recomendable eliminar procesos que se encuentren en segundo plano no necesarios para el funcionamiento del sistema con el objetivo de obtener unos datos más fiables, ya que todos los procesos en segundo plano también consumen energía por pequeña que pueda ser

Para no obtener muchos datos similares, la obtención de datos se limita a los límites del nivel de caché del procesador, es decir, un número limitado de elementos de vector. La siguiente figura es una representación de los datos seleccionados para cada nivel de caché (color verde), ilustración 8-5.

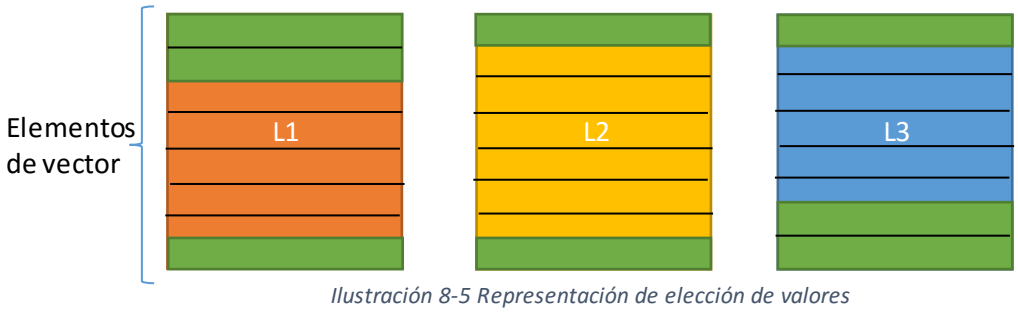


Ilustración 8-5 Representación de elección de valores

Para que *Joulemeter* tuviese más carga computacional y así poder observar unos resultados más estables sobre su comportamiento, se aumentó el peso computacional de todas las llamadas realizadas, equilibrando la carga en cada número de elementos de vector, para obtener las mismas condiciones para todos. Esto se debe a que *Joulemeter*, cuando mide el consumo, el muestreo lo hace por segundo. Por ello, era necesario aumentar el tiempo de cómputo para poder visualizar comportamientos.

Se generó un nuevo archivo *.bat* que realizase cada compilación y llamada introduciendo un tiempo de espera entre ellas para poder diferenciarlas en el resultado de *Joulemeter*.

```
...
del snrm2.h
echo #define Nc 10000001 >> snrm2.h
echo #define NTHR 2 >> snrm2.h
echo #define NUM_ELEM  500 >> snrm2.h
echo #define OPCION 3 >> snrm2.h

echo 3-500 > sal1.txt

"C:\MinGW\bin\gcc.exe"  -O3            -c snrm2_seq.c
"C:\MinGW\bin\gcc.exe"  -O3   -fopenmp   -c snrm2_omp.c
"C:\MinGW\bin\gcc.exe"  -O3   -fopenmp -mavx2 -msse3 -mfma   -o snrm2_stats.exe main.c
snrm2_seq.o snrm2_omp.o
```

A

```
ping -n 2 0.0.0.0 >nul       < - Espera de dos segundos
snrm2_stats.exe

del snrm2.h
echo #define Nc 5000001 >> snrm2.h
echo #define NTHR 2 >> snrm2.h
echo #define NUM_ELEM  1000 >> snrm2.h
echo #define OPCION 1 >> snrm2.h

echo 1-1000 > sal1.txt

"C:\MinGW\bin\gcc.exe"  -O3            -c  snrm2_seq.c
"C:\MinGW\bin\gcc.exe"  -O3   -fopenmp   -c  snrm2_omp.c
"C:\MinGW\bin\gcc.exe"  -O3   -fopenmp  -mavx2 -msse3 -mfma   -o  snrm2_stats.exe main.c
snrm2_seq.o snrm2_omp.o

ping -n 2 0.0.0.0 >nul
snrm2_stats.exe
...
```

Ilustración 8-6 Compilación para obtener consumo

Como puede observarse en la ilustración 8-6, al igual que la medición de datos de los capítulos anteriores también se crea un fichero .h que contiene las variables que el *main* va a manejar.

A pesar de que el número de elementos de vector se diferente en cada caso, la multiplicación junto con la variable *Nc* da como resultado el mismo número o uno muy aproximado en todos los casos. De esta manera se consiguen las mismas condiciones para cada llamada (A).

También, puede observarse la existencia de una variable llamada OPCION. Esta variable se modifica a lo largo del archivo *.bat* y junto con el *main* del programa realiza llamadas a los diferentes tipos de procesamiento para obtener los resultados de consumo independientes: secuencial, vectorial y/o paralelo.

Adicionalmente, se añade un comando que guarda en un fichero el progreso de la ejecución.

```
echo 1-1000 > sal1.txt
```

En esta parte del proyecto, el *main* es alterado de tal forma que la variable opción antes mencionada pueda realizar el procesamiento de datos indicado. Es prácticamente igual que el *main* utilizado para medir *speedups* en la sección 5.2, pero con la diferencia de que en este caso no se muestra nada por pantalla, lo que importa es el tiempo de llamada y su consumo en consecuencia.

En el siguiente código se muestra la estructura seguida para generar el *main,* ilustración 8-7.

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

  switch (OPCION){

    case 1:
      for (min=UINT_MAX,  max= 0, sum=0, pri= 0, i=0; i<Nc; i++){
        inicializar(x);
        ini = rdtsc();
```

```
            nrm2 = snrm2_seq(NUM_ELEM,x);
            fin = rdtsc();
            if(i>0){
               dif = (fin - ini);
               sum += dif;
               max = (dif > max)? dif : max;
               min = (dif < min)? dif : min;
            }else{
               dif = (fin - ini);
               pri = dif;
            }
         }
         med_seq  = sum/(Nc-1);
         break;
         case 2:
         ...
```

Ilustración 8-7 Elección de ejecución según valor de OPCION

Una vez que se han modificado todos los ficheros y se han rellenado los campos indicados de la herramienta *Joulemeter*, se dispone a obtener los datos en las tablas de salida del programa, ilustración 8-8.

TimeStamp (ms)	Total Power (W)	CPU (W)	Monitor (W)	Disk (W)	Base (W)	Application (W)
6,3601E+13	10,4	0,2	0,7	0	9,5	--
6,3601E+13	10,2	0,1	0,7	0	9,5	Waiting for [snrm2_stats] application data
6,3601E+13	10,2	0,1	0,7	0	9,5	Waiting for [snrm2_stats] application data
6,3601E+13	14,3	4,2	0,7	0	9,5	Waiting for [snrm2_stats] application data
6,3601E+13	12,4	2,3	0,7	0	9,5	Waiting for [snrm2_stats] application data
6,3601E+13	14,9	4,8	0,7	0	9,5	0
6,3601E+13	14,9	4,8	0,7	0	9,5	4,7
6,3601E+13	15,2	5	0,7	0	9,5	4,7
6,3601E+13	15,1	4,9	0,7	0	9,5	4,7
6,3601E+13	15	4,9	0,7	0	9,5	4,7
6,3601E+13	14,9	4,7	0,7	0	9,5	4,7
6,3601E+13	14,9	4,8	0,7	0	9,5	4,7
6,3601E+13	14,9	4,8	0,7	0	9,5	4,7
6,3601E+13	15	4,9	0,7	0	9,5	Se seleccionan estos 4,7
6,3601E+13	14,9	4,8	0,7	0	9,5	4,7
6,3601E+13	15	4,9	0,7	0	9,5	4,7
6,3601E+13	14,9	4,7	0,7	0	9,5	4,7
6,3601E+13	14,9	4,8	0,7	0	9,5	4,7
6,3601E+13	15	4,9	0,7	0	9,5	4,7
6,3601E+13	14,9	4,7	0,7	0	9,5	4,7
6,3601E+13	14,9	4,7	0,7	0	9,5	4,7

Ilustración 8-8 Tabla de salida Joulemeter

8.3 ANÁLISIS

Después de obtener todas las tablas de cada función del primer nivel del BLAS seleccionadas, es necesario hacer un análisis para poder observar el comportamiento de cada tipo de procesamiento.

Para ello, a partir de todos los datos que ofrece *Joulemeter*, es necesario limitarlos y seleccionar solo dos de ellos. *Joulemeter* ofrece:

- Tiempo en milisegundos (equivalente a un segundo)
- Consumo total
- Consumo de CPU
- Consumo del monitor
- Consumo del disco
- Consumo de la placa base
- Consumo de la aplicación que se quiere medir

Como este tipo de programas está pensado para ejecutarse en centros de cálculo donde se puede prescindir momentáneamente de la pantalla, se decidió eliminar ese consumo. El consumo del disco es 0 en las ejecuciones. El consumo de la aplicación también se elimina, debido a que ese consumo está dentro del propio consumo de CPU. El consumo total no interesa, debido a que se han eliminado varios casos y se vería afectado el valor. Por lo tanto, los datos que se han de seleccionar para el análisis son: tiempo, CPU y placa base.

A partir de esos datos observando la columna de consumo de aplicación se puede indicar el tiempo transcurrido por número de filas y realizar el cálculo del consumo mediante el consumo de la placa y de CPU por fila.

Esta suma se realiza para todos los casos secuenciales. Cuando se pretende obtener el consumo de los demás tipos de procesamiento, por lo general suelen tener menor número de filas, tarda menos, pero como debe de estar en las mismas condiciones que el secuencial para poder ser comparado, es necesario añadir el consumo cuando la maquina este en estado *iddle* o sin hacer nada.

Para realizar este cálculo correctamente se realiza el cálculo de consumo de la parte en serie y se anota el número de filas que tiene. Después en cada tipo de procesamiento diferente se realiza el mismo cálculo, pero al tener menor número de filas se le suma el consumo en estado *iddle* de las restantes. En el proyecto al estado *iddle* se le ha asignado un valor fijo de 9,6, consumo de la placa 9,5 y más la CPU en *iddle* 0,1 (en media).

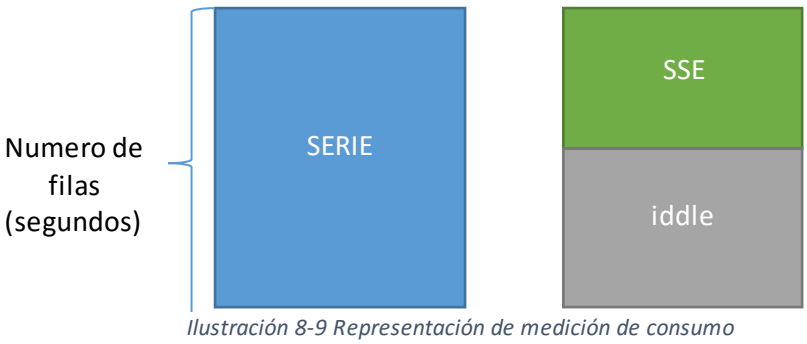


Ilustración 8-9 Representación de medición de consumo

8.3.1 Tipos de consumo

Existen varias opciones de consumo configurables por el sistema. Estas opciones pueden ser habilitadas de manera manual por el usuario en el menú de energía del sistema. En él se mostrarán tres tipos de opciones de energía:

- **Bajo consumo**: el sistema intenta reducir el consumo total, impidiendo que los *cores* puedan aumentar de frecuencia y voltaje.
- **Alto rendimiento**: el sistema aumenta la frecuencia y voltaje de los *cores* del procesador de manera que se pueda realizar mucho más rápido una operación. Este aumento está limitado por el TDP del procesador. El consumo es mayor.
- **Equilibrado**: es la opción por defecto del sistema y alterna entre las dos opciones anteriores dependiendo de las situaciones en las que se encuentre el sistema.

En este proyecto solo se han realizado prueba bajo las opciones de "Alto rendimiento" y "Equilibrado".

8.3.2 Representaciones graficas

En la muestra de datos, se ha contemplado una diferenciación de resultados por parte de ambos tipos de opciones de energía utilizados.

Cuando la opción era "Equilibrado" se ha generado una tabla representando los *speedups* conseguidos, junto con el consumo de la obtención de los mismos por parte de los diferentes tipos de procesamiento respecto al secuencial. Remarcando en negrita aquellos datos que tienen mejor *speedup* y consumo para cada número de elementos de vector, ilustraciones 8-(10-17).

		SAXPY													
	n	sse		avx		fma		paralelo		par-sse		par-avx		par-fma	
		Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo
L1	500	3,593933	0,956769	5,858054	**0,95165**	**6,22542**	0,9562	0,008947	x	0,010855	x	0,009585	x	0,009678	x
	1000	3,47093	0,956989	**7,341532**	**0,9451**	6,476331	0,9485	0,032287	x	0,034028	x	0,023531	x	0,04088	x
	4000	3,907405	0,964263	7,466543	0,9759584	**7,51024**	**0,95712**	0,075549	x	0,07026	x	0,06578	x	0,072204	x
L2	8000	3,446677	0,963661	**5,01316**	**0,96136**	4,884599	0,969641	0,132473	x	0,204912	x	0,208489	x	0,292735	x
	28000	3,371136	0,978406	4,663166	**0,97532**	**4,95125**	0,990746	0,406478	x	0,967859	x	1,001255	x	1,00353	x
L3	33000	**2,75269**	0,97026	2,582217	0,972119	2,217298	**0,9698**	0,394488	x	0,458605	x	0,469613	x	0,471983	x
	250000	3,117274	0,962803	3,87064	0,9653973	**4,41052**	**0,95329**	1,802666	0,991782	2,793951	0,961938	2,82699	0,9636678	2,835047	0,965398
	325000	3,441095	0,980062	3,775535	**0,97785**	**3,86488**	0,981391	2,192616	1,019938	3,685566	0,993354	3,507222	0,9880372	2,799502	0,988037

Ilustración 8-10 Estudio de consumo y rendimiento para la función saxpy

		DAXPY													
	n	sse		avx		fma		paralelo		par-sse		par-avx		par-fma	
		Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo
L1	500	2,211938	0,931896	3,751991	0,9230769	**4,25823**	**0,92308**	0,017163	x	0,014727	x	0,014893	x	0,014801	x
	1000	1,873367	0,980412	3,608763	0,9721649	**3,62114**	**0,96907**	0,031697	x	0,034059	x	0,020838	x	0,023329	x
	2000	1,913369	0,988017	3,629691	0,9738562	**3,27368**	**0,97113**	0,049504	x	0,08383	x	0,084057	x	0,059024	x
L2	2500	1,260233	0,988772	1,285395	0,9894966	**1,31608**	**0,98877**	0,048693	x	0,04904	x	0,055257	x	0,055199	x
	13000	1,794204	0,984451	2,271806	**0,94644**	**2,91402**	0,979959	0,325048	x	0,365327	x	0,376342	x	0,37797	x
L3	18000	**1,27547**	**0,99174**	1,22359	0,9938838	1,263225	0,99633	0,274362	x	0,316764	x	0,325285	x	0,328765	x
	100000	1,116385	0,994792	1,523951	0,9981618	**1,56447**	**0,99479**	0,766656	1,345895	0,964989	1,106618	1,01565	1,0735294	1,011536	1,075061
	175000	1,229663	0,990805	1,174573	0,9902299	**1,32667**	**0,99023**	0,91775	1,013218	1,098237	1,00977	1,133269	1,008046	1,082047	1,009195

Ilustración 8-11 Estudio de consumo y rendimiento para la función daxpy

		IAXPY									
	n	sse		avx		paralelo		par-sse		par-avx	
		Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo
L1	500	3,069514	0,937882	**5,536254**	**0,92753**	0,006072	x	0,007595	x	0,007664	x
	1000	3,059746	0,960945	**5,063815**	**0,95034**	0,013072	x	0,016968	x	0,02155	x
	4000	4,410341	0,974756	**4,421302**	**0,96672**	0,09359	x	0,109173	x	0,096403	x
L2	8000	2,605338	0,964549	**2,950523**	**0,96041**	0,069089	x	0,150778	x	0,167316	x
	28000	2,610138	**0,97399**	**3,112863**	1,00000	0,268344	x	0,363132	x	0,485533	x
L3	33000	2,561112	0,950181	**3,050935**	**0,94339**	0,388528	x	0,520714	x	0,404439	x
	250000	2,890947	**0,98006**	**4,438194**	0,9911915	1,722776	1,013908	2,793054	0,988873	2,848558	0,989801
	325000	2,208004	0,98181	**2,350315**	**0,97959**	1,076091	1,015528	1,638598	0,988465	1,8189	0,999556

Ilustración 8-12 Estudio de consumo y rendimiento para la función iaxpy

		SLAMAX									
	n	sse		avx		paralelo		par-sse		par-avx	
		Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo
L1	500	6,066989	0,952565	**9,782851**	**0,949367089**	0,028201	x	0,035877	x	0,035886	x
	1000	6,3973	0,956603	**11,597222**	**0,952209798**	0,057602	x	0,071254	x	0,071372	x
	8000	5,861943	0,95148	**11,152837**	**0,947425328**	0,44067	x	0,424253	x	0,573915	x
L2	13000	6,099778	0,951505	**12,718924**	**0,947375433**	0,844775	x	0,750472	x	0,901139	x
	58000	6,41703	0,949412	**13,013738**	**0,944763467**	2,115114	x	2,774279	x	3,768312	x
L3	100000	6,714844	0,952324	**13,07196**	**0,94911859**	2,420039	x	3,959895	x	5,486334	x
	700000	6,583278	0,943566	**13,319073**	0,941152815	6,015717	0,95308311	9,02914	0,946380697	13,01227	**0,941018767**
	775000	6,677285	0,956365	**13,404923**	0,949426208	6,126535	0,962636776	**6,943**	0,959701094	11,13194	0,950226848

Ilustración 8-13 Estudio de consumo y rendimiento para la función si_amax

		DLAMAX									
	n	sse		avx		paralelo		par-sse		par-avx	
		Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo
L1	500	2,885057	0,965765	**6,182266**	**0,961759082**	0,044149	x	0,038966	x	0,027405	x
	1000	2,833057	0,968296	**6,072684**	**0,958575581**	0,033799	x	0,03759	x	0,08159	x
	4000	2,587883	0,972127	**5,562261**	**0,96508936**	0,32122	x	0,262303	x	0,297939	x
L2	8000	3,487583	0,970798	**6,820197**	**0,969699744**	0,572631	x	0,535236	x	0,262161	x
	28000	2,77037	0,98154	**5,596384**	**0,970389805**	0,842048	x	1,630428	x	1,847251	x
L3	33000	2,992613	0,97184	**6,634754**	**0,967895799**	0,851536	x	1,337236	x	1,830426	x
	250000	3,338623	0,968707	**6,667243**	**0,964829687**	1,689589	1,001292347	4,093172	0,972399151	6,018759	0,96529124
	325000	3,290604	0,977085	**6,556827**	**0,950441029**	1,70331	0,969719014	4,408972	1,001909612	6,555787	0,974720378

Ilustración 8-14 Estudio de consumo y rendimiento para la función di_amax

		ZLAMAX									
	n	sse		avx		paralelo		par-sse		par-avx	
		Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo
L1	500	1,856131	0,980458	**3,015494**	**0,968808719**	0,019745	x	0,020556	x	0,020612	x
	1000	1,792753	0,984075	**3,083174**	**0,962721679**	0,019556	x	0,03019	x	0,03391	x
	8000	1,816276	0,977494	**3,216983**	**0,968867217**	0,160345	x	0,18556	x	0,299312	x
L2	13000	2,652838	0,982746	**4,699871**	**0,972243061**	0,455458	x	0,395415	x	0,562062	x
	58000	1,991461	0,978857	**3,729746**	**0,979248238**	0,610909	x	1,101904	x	1,524653	x
L3	100000	2,013353	0,975769	**3,635439**	**0,960216998**	0,882467	x	1,332086	x	1,990515	x
	700000	1,805222	0,979321	3,225599	0,973858759	0,986236	1,012485369	2,749612	0,977760437	**4,0631**	**0,967225907**
	775000	2,082357	0,974396	3,712469	**0,961053011**	1,708944	1,016588532	3,153674	0,974756581	**4,7626**	0,973674721

Ilustración 8-15 Estudio de consumo y rendimiento para la función zi_amax

		SNRM2														
	n	sse		avx		fma		paralelo		par-sse		par-avx		par-fma		serie2
		Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	
L1	500	10,31464	**0,94501992**	**15,857143**	0,942549801	13,072314	x	0,032605	x	0,047863	x	0,047733	x	0,047707	x	0,073094
	1000	11,21578	0,945331016	**19,663351**	**0,941064939**	15,646055	x	0,057319	x	0,065245	x	0,082149	x	0,102871	x	0,078943
	8000	11,933892	0,946161275	**23,446411**	**0,942539836**	17,739964	x	0,401801	x	0,687361	x	0,668841	x	0,659623	x	0,079542
L2	13000	12,133345	0,929907264	**23,392294**	**0,929057187**	17,545451	x	0,737148	x	1,178909	x	0,882846	x	1,051425	x	0,080533
	58000	9,05716	0,930606742	**16,067392**	**0,925303371**	15,319436	x	1,387016	x	3,285948	x	3,606198	x	4,092337	x	0,081217
L3	100000	12,097456	0,934521503	**21,565129**	**0,932119334**	16,490495	x	1,715961	x	4,658036	x	5,985838	x	5,495531	x	0,075595
	700000	11,433659	0,896141998	**22,028137**	**0,894381693**	17,573788	0,894895115	2,54768	0,931054716	14,959604	0,897388881	18,072726	0,895701922	15,467456	0,896141998	0,07848
	775000	11,549549	0,965476379	**20,436337**	**0,963501343**	17,083047	0,964291357	2,54558	1,003160057	12,68028	0,967846421	15,559916	0,964686364	15,798439	0,964844367	0,078535

Ilustración 8-16 Estudio de consumo y rendimiento para la función snrm2

		DNRM2														
	n	sse		avx		fma		paralelo		par-sse		par-avx		par-fma		serie2
		Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	
L1	500	4,91711	0,957976897	**9,222402**	**0,954979707**	8,131604	x	0,033784	x	0,034777	x	0,042607	x	0,054944	x	0,084374
	1000	5,783998	0,946045371	**11,09306**	**0,941079093**	8,513349	x	0,060545	x	0,090179	x	0,088318	x	0,089119	x	0,078391
	4000	6,312392	0,963923768	**12,488277**	**0,958378482**	9,415904	x	0,229298	x	0,261891	x	0,271579	x	0,336345	x	0,082887
L2	8000	5,96859	0,973219337	**11,579106**	**0,960771987**	8,796804	x	0,391851	x	0,615322	x	0,687864	x	0,659791	x	0,07914
	28000	6,039146	0,962838693	**11,91164**	**0,959863257**	9,010379	x	1,176009	x	1,739765	x	1,877998	x	2,151865	x	0,073544
L3	33000	4,909438	0,961789844	**8,365127**	**0,959401709**	6,921675	x	1,204377	x	2,107986	x	2,568744	x	2,615713	x	0,082313
	250000	2,018954	0,960307945	**10,984069**	**0,955322774**	8,882054	0,958225532	2,19236	0,986495867	5,561289	0,96390484	7,058728	0,958288635	7,087426	0,960181738	0,081686
	325000	5,697404	0,958232333	7,769305	**0,954876002**	8,621063	0,957672944	2,398172	0,987258375	7,084123	0,960656349	**9,673696**	0,955373236	8,887001	0,955994779	0,076198

Ilustración 8-17 Estudio de consumo y rendimiento para la función dnrm2

Aquellos casos en los que el *speedup* era menor que el caso secuencial no se han calculado, por tanto se representan con un color más blanco y con el consumo con "x".

Cuando la opción era "Alto rendimiento" se han generado varias graficas por número de elementos de vector (únicamente en el límite del nivel L3 de caché) en la que puede observarse una ganancia considerable de consumo y rendimiento respecto al procesamiento secuencial.

Este análisis solo se ha realizado con las funciones snrm2 y dnrm2 del BLAS, debido a que eran las que más rendimiento, ilustraciones 8-(18-19).

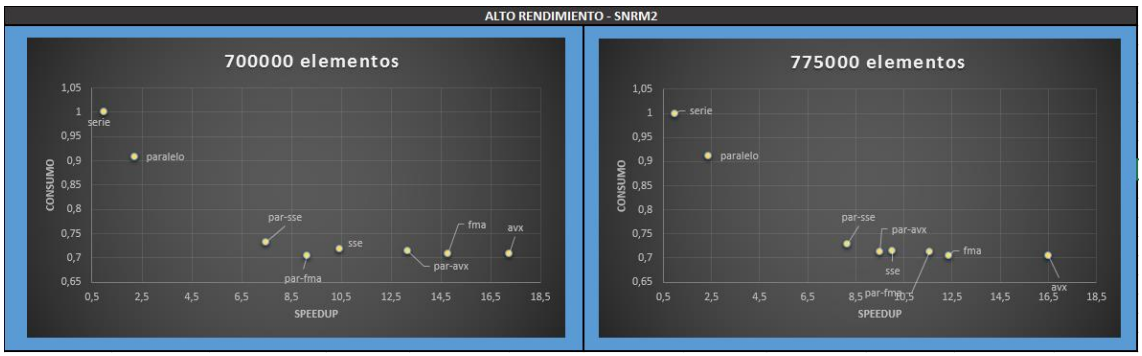


Ilustración 8-18 Representación gráfica por elementos de vector de función snrm2

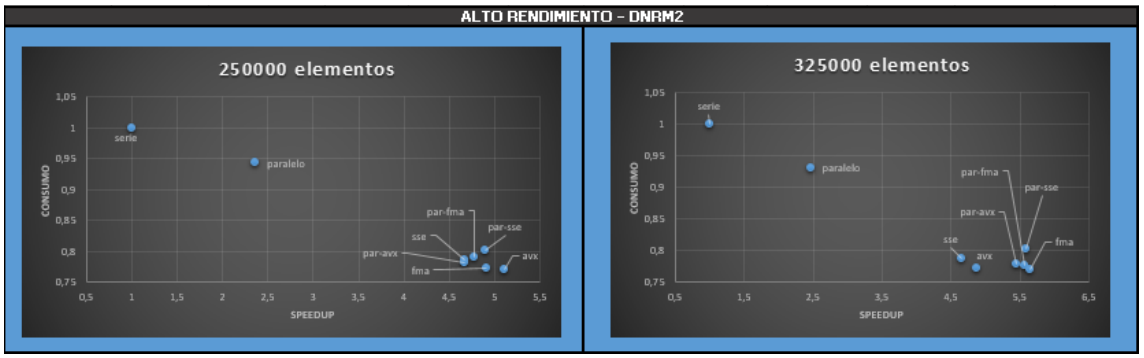


Ilustración 8-19 Representación gráfica por elementos de vector de función dnrm2

8.4 PERFMONITOR 2

Como herramienta secundaria para la verificación de los datos sobre el consumo, se ha utilizado un software que monitoriza diferentes componentes de la máquina y del sistema. En el caso del proyecto, se ha utilizado para observar el consumo de energía en las diferentes opciones de consumo utilizadas.

Primeramente se descarga el programa del portal web habilitado para ello.

http://www.cpuid.com/softwares/perfmonitor-2.html

Una vez instalado y arrancado, el propio programa comienza a mostrar información del sistema y sus componentes. Pero en este caso, se le da importancia a la monitorización del consumo, por lo que se accede la pestaña de "*Selection*" y se selecciona la opción "*GT Power*".

Aquí se muestran varios ejemplos de la monitorización realizada:

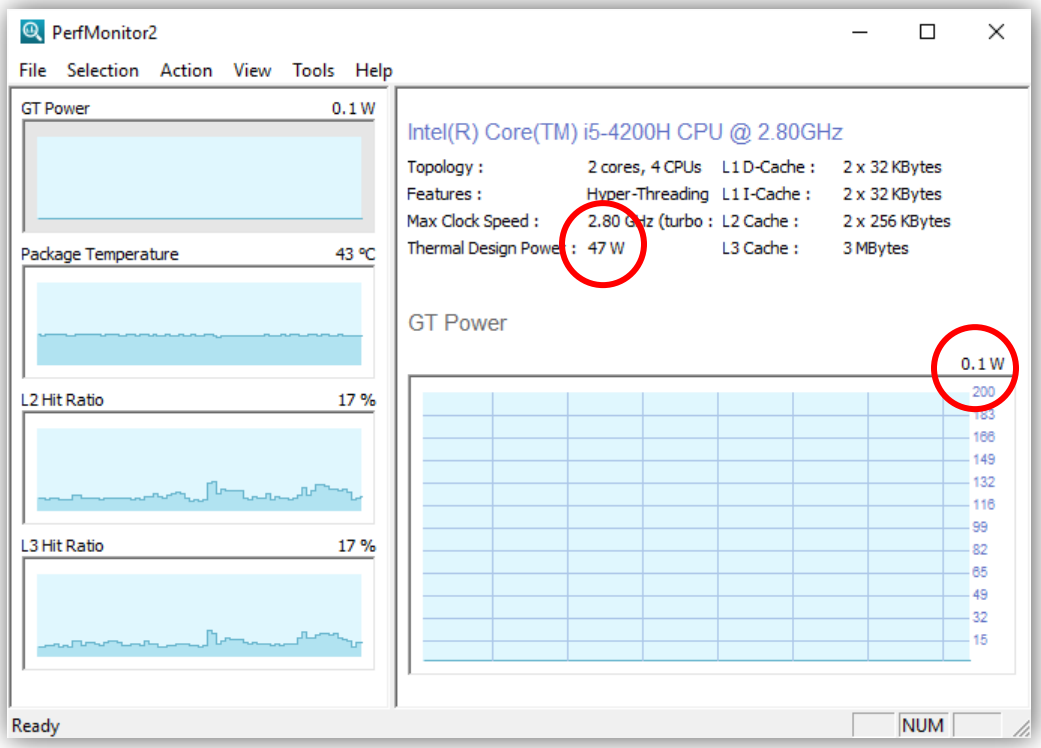


Ilustración 8-20 Monitorización en modo iddle

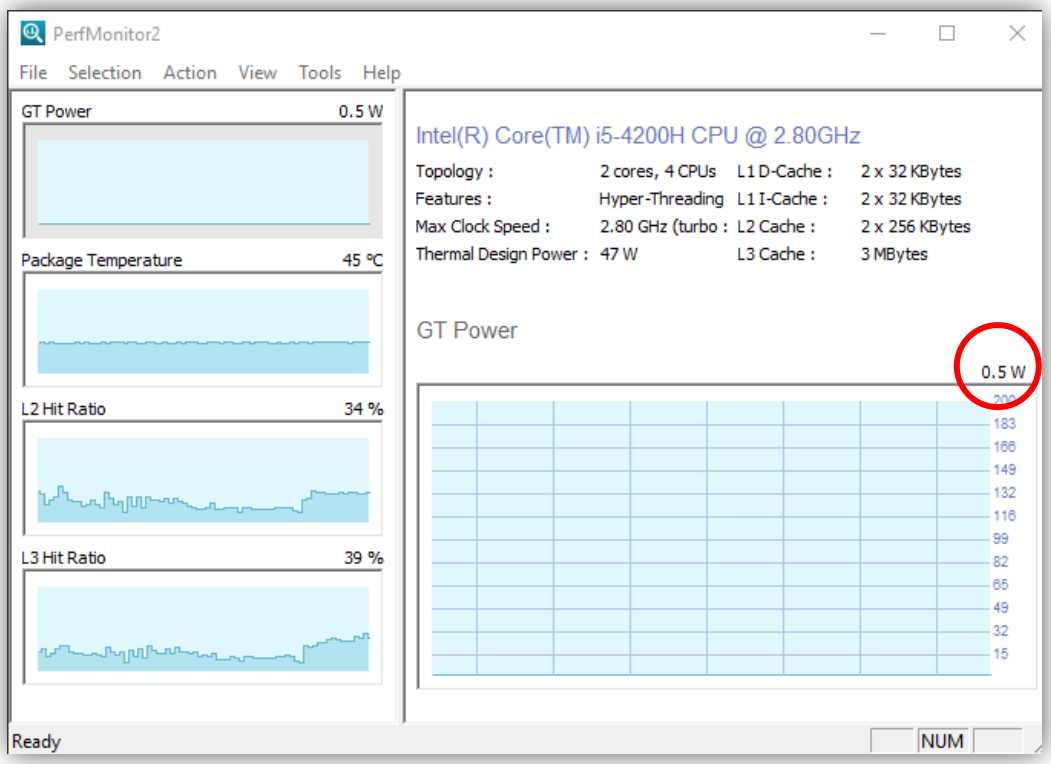


Ilustración 8-21 Monitorización en modo Equilibrado

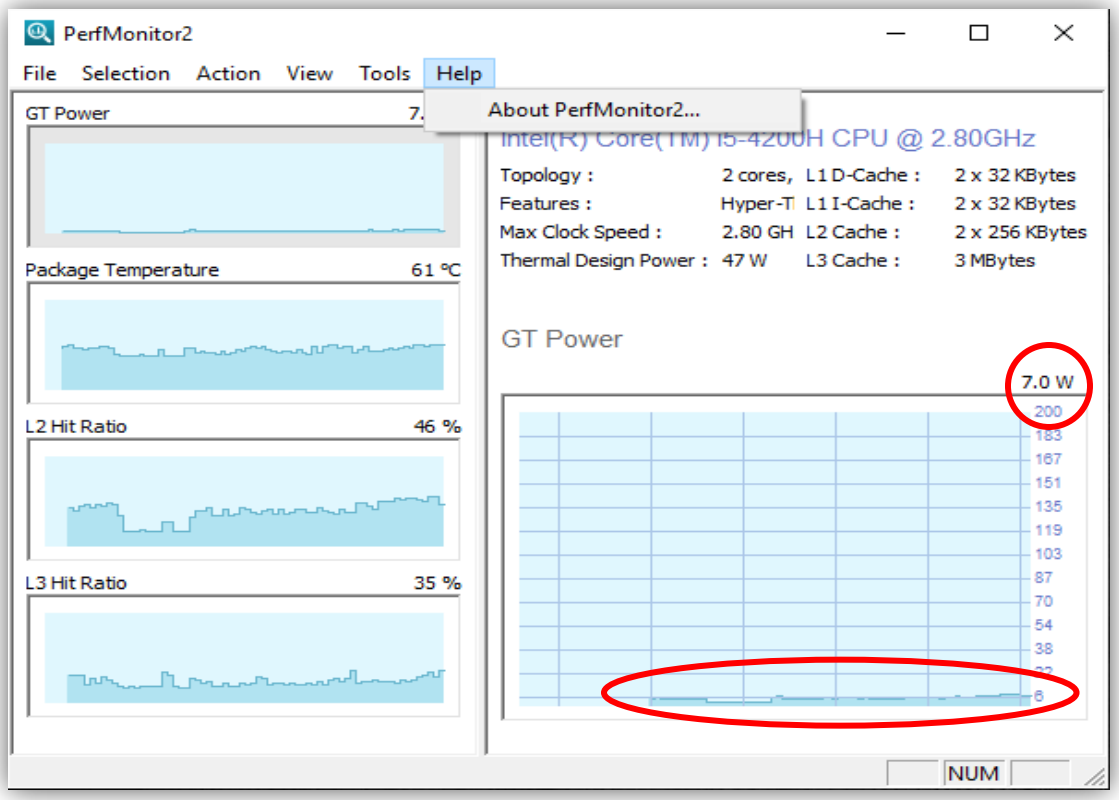


Ilustración 8-22 Monitorización en modo Alto rendimiento

No se puede observar correctamente, debido a que no puede modificarse la escala de potencia de la gráfica, pero hay más datos que son interesantes.

Por un lado se muestra la información del procesador del que se dispone, indicando el valor del TDP que puede soportar, en este caso 47W.

Por otro lado se puede observar el consumo capturado en cada instante. En modo *iddle* el consumo de CPU es 0,1 W. Cuando el sistema está con la opción "Equilibrado" y comienza a ejecutar algo aumenta a 0,5W. En cambio, cuando la opción es "Alto rendimiento" su consumo aumenta hasta 7W.

9 CAPÍTULO: GESTIÓN DEL PROYECTO

9.1 EDT

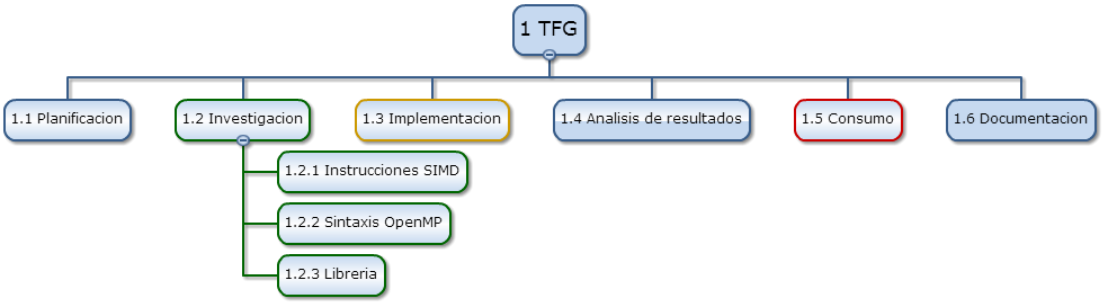


Ilustración 9-1 EDT

9.2 ESTIMACIÓN DEL TIEMPO

Se ha estimado el tiempo de cada fase en horas. De esta manera se calcula el tiempo total estimado para el proyecto.

Fases	Tiempo (h)
Planificación	12
Investigación	33
Implementación	200
Análisis de resultados	22
Consumo	15
Documentación	70
Presentación	
Total	352

Tabla 9-1 Dedicación del proyecto

9.3 REUNIONES CON EL TUTOR

La comunicación con el tutor se ha realizado principalmente mediante reuniones o mediante correo electrónico.

- Reuniones: No han sido planificadas y no se han realizado con una periodicidad concreta, sino que se han realizado cuando ha sido necesario. Además de estas reuniones, ha existido la posibilidad de ir al despacho del tutor para solucionar dudas para con el proyecto.
- Email: El correo electrónico ha permitido planificar las reuniones, intercambiar trabajo realizado y en casos concretos resolver dudas.

10 CAPÍTULO: CONCLUSIONES

En este capítulo se presentan las conclusiones del proyecto.

10.1 CONCLUSIONES GENERALES DEL PROYECTO

En el proyecto aquí presentado, se ha realizado una implementación en lenguaje C del primer nivel del BLAS. De esas implementaciones que permiten aprovechar la maquina con la que se está trabajando, se han obtenido diferentes resultados de rendimiento recogidos en el documento adjunto "Resultados de rendimiento". A partir de esos datos se ha generado la siguiente tabla para caracterizar el comportamiento del primer nivel de BLAS sobre una de las configuraciones utilizadas, concretamente la segunda, de manera que se pueda observar un patrón dependiendo de varios factores.

Sin IF – Sin computo			Sin IF – Con computo			Con IF		
Función + Cómputo O(kn)	Último Speedup vectorial seleccionado (float)	Último Speedup parvec seleccionado (float)	Función + Cómputo O(kn)	Último Speedup vectorial seleccionado (float)	Último Speedup parvec seleccionado (float)	Función + Cómputo O(kn)	Último Speedup vectorial seleccionado (float)	Último Speedup parvec seleccionado (float)
copy (0)	1.57	2.4	scal (1)	3.1	3.69	_i_amax (0)	13.46	27.08
swap (0)	2.13	3.62	dot (2)	4.81	5.94	_nrm2 (4)	22.62	32.59
			asum (1)	7.72	11.69			
			axpy (2)	2.55	3.55			
			rot (6)	4.59	6.51			
			rotm (4-6)	2.93	4.22			
Rendimiento menor			Rendimiento medio			Rendimiento mayor		

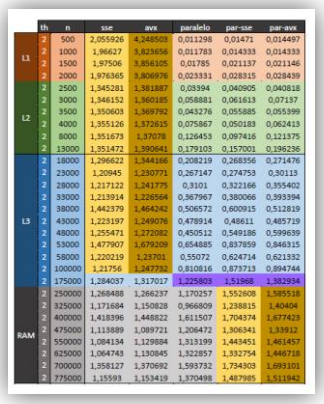
Tabla 10-1 Caracterización del BLAS 1 sobre el proyecto

Tal y como se muestra en la tabla 10-1, se han representado los diferentes comportamientos de las funciones del primer nivel de BLAS, de tipo float, respecto a los resultados de rendimiento obtenidos. Se puede encontrar tres subgrupos:

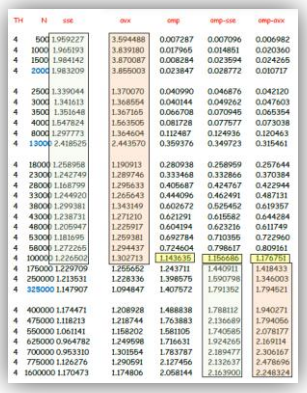
- Funciones que carecen de operaciones aritméticas (computo O(0)) y no tienen dependencias de control. Estas funciones tienen el menor rendimiento.

- Funciones que tienen operaciones aritméticas, entre 1-6 operaciones (Computo O(n) - O(6n)), y no tienen dependencias de control. Debido a la existencia de operaciones aritméticas el rendimiento paralelo-vectorial crece respecto al primer subgrupo.

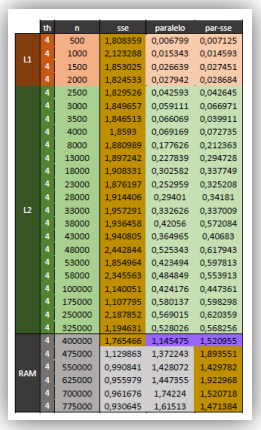
- Funciones que originalmente tienen dependencias de control, pero que en su implementación han sido eliminadas. Este motivo acarrea mayor rendimiento, independientemente del número de operaciones a realizar.



1ª configuración



2ª configuración



3ª configuración

Ilustración 10-1 Tendencia de elección de ejecución en las tres configuraciones - dswap

Generalizando, cuando el rango de datos con el que se trabaja se encuentra dentro de los límites de la memoria caché, la elección principal es la ejecución en vectorial. Cuando los datos superan el último nivel de caché del que se dispone, en memoria RAM, la elección principal es la ejecución en paralelo-vectorial.

Este comportamiento se debe a que el primer nivel del BLAS está acotado por memoria (*Memory Bound*). Para ocultar la latencia se ejecuta en vectorial. Cuando se supera el último nivel de cache y se entra en RAM, se oculta la latencia ejecutando en paralelo-vectorial.

Una vez obtenidos los datos de rendimiento, se han seleccionado los mayores mediante un criterio de decisión, reflejado en una función para cada tipo de función o subrutina del primer nivel de BLAS, y se ha generado una librería (dinámica o estática). Esta librería dependiendo del tipo que sea puede ser actualizada simplemente modificando sus códigos implementados, en el caso de que sea dinámica, y puede ser portable, en el caso de que sea estática.

Para finalizar el proyecto, se ha realizado un estudio de consumo sobre varias funciones del primer nivel de BLAS con diferentes resultados de rendimiento. Este estudio ha sido posible mediante la herramienta *Joulemeter*. A continuación se observa la diferencia de resultados respecto a las opciones de energías analizadas, "Equilibrado" y "Alto Rendimiento", para la función dnrm2 con 750000 elementos.

	SNRM2														
	sse		avx		fma		paralelo		par-sse		par-avx		par-fma		serie2
	n	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	Speedup	Consumo	
L1	500	10,31464	**0,94501992**	**15,857143**	0,942549801	13,072314	x	0,032605	x	0,047863	x	0,047733	x	0,047707	x	0,073094
	1000	11,21578	0,945331016	**19,663351**	**0,941064939**	15,646055	x	0,057319	x	0,065245	x	0,082149	x	0,102871	x	0,078943
	8000	11,933892	0,946161275	**23,446411**	**0,942539836**	17,739964	x	0,401801	x	0,687361	x	0,668841	x	0,659623	x	0,079542
L2	13000	12,133345	0,929907264	**23,392294**	**0,929057187**	17,545451	x	0,737148	x	1,178909	x	0,882846	x	1,051425	x	0,080533
	58000	9,05716	0,930606742	**16,067392**	**0,925303371**	15,319436	x	1,387016	x	3,285948	x	3,606198	x	4,092337	x	0,081217
	100000	12,097456	0,934521503	**21,565129**	**0,932119334**	16,490495	x	1,715961	x	4,658036	x	5,985838	x	5,495531	x	0,075595
L3	700000	11,433659	0,935141998	**22,028137**	**0,894381693**	17,573788	0,894895115	2,54768	0,931054716	14,959604	0,897388881	18,072726	0,895701922	15,467456	0,896141998	0,07848
	775000	11,549543	0,965476379	**20,436337**	**0,963501343**	17,083047	0,964291357	2,54558	1,003160057	12,68028	0,967846421	15,559916	0,964886364	15,798439	0,964844367	0,078535

Ilustración 10-2 Ejemplo tabla consumo - speedup de función snrm2

Tal y como se observa en la ilustración 10-2, opción "Equilibrado", el mejor dato de *speedup* y consumo se encuentran en la parte vectorial. Sin embargo la ejecución vectorial reduce levemente el consumo de la ejecución secuencial, en este caso a un 96%.

En la ilustración 10-3, opción “Alto Rendimiento”, se puede observar una ganancia de rendimiento considerable respecto a la ejecución secuencial por parte de las ejecuciones vectoriales y/o paralelas, pero menor que utilizando la opción “Equilibrado”. En cambio, se observa el consumo reducido al 70-75%.

En vista a los resultados, se puede afirmar que la ejecución vectorial y/ paralela con la opción energética “Alto Rendimiento”, obtiene una mayor reducción de consumo que la opción “Equilibrado” respecto a sus respectivas ejecuciones secuenciales, pero con una menor ganancia de rendimiento.

10.2 LÍNEAS FUTURAS

A continuación se presentan algunas ideas futuras que pueden incluirse en este proyecto:

- Realizar el mismo estudio sobre el segundo y tercer nivel de BLAS. Este estudio tendría variaciones respecto al realizado, ya que este estaría acotado por computo (*CPU bound*) y no por memoria. Además, como esos dos niveles trabajan sobre matrices, sería una buena idea incluir aparte del procesamiento vectorial y/o paralelo, el procesamiento mediante instrucciones CUDA (*Compute Unified Device Architecture*), aprovechando así la potencia de las tarjetas gráficas.
- Afinar el criterio de decisión realizado, teniendo en cuenta los datos que la función cpuid puede ofrecernos sobre los niveles de caché, es decir, obtener los datos de cada nivel de caché y a partir de las conclusiones mencionadas, junto con el número de elementos de vector realizar una implementación sobre las rutinas de decisión más generalizada.
- Realizar un criterio de decisión que tenga en cuenta el consumo de las ejecuciones secuenciales, vectoriales y/o paralelas.

BIBLIOGRAFÍA

Kernel.org. (2016). *Basics of SIMD Programming*. [online] Disponible en: https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html [Accedido el 12 Jun. 2016].

C++, B. and González, J. (2016). *Bibliotecas o librerías de C++. Uso del include en C++*. [online] Programarya.com. Disponible en: https://www.programarya.com/Cursos/C++/Bibliotecas-O-Librerias [Accedido el 12 Jun. 2016].

Wikipedia. (2016). *FMA instruction set*. [online] Disponible en: https://en.wikipedia.org/wiki/FMA_instruction_set [Accedido el 12 Jun. 2016].

Software.intel.com. (2016). *Intel Intrinsics Guide*. [online] Disponible en: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ [Accedido el 12 Jun. 2016].

Labs, i. (2016). *Intel Hyper-Threading Technology Review*. [online] iXBT Labs. Disponible en: http://ixbtlabs.com/articles/pentium4xeonhyperthreading/ [Accedido el 12 Jun. 2016].

Chuidiang.com. (2016). *Librerías estáticas y dinámicas*. [online] Disponible en: http://www.chuidiang.com/clinux/herramientas/librerias.php [Accedido el 12 Jun. 2016].

performance?, H. (2016). *How to chain multiple fma operations together for performance?*. [online] Stackoverflow.com. Disponible en: http://stackoverflow.com/questions/23710356/how-to-chain-multiple-fma-operations-together-for-performance [Accedido el 12 Jun. 2016].

Netlib.org. (2016). *Quick Reference Guide to the BLAS*. [online] Disponible en: http://www.netlib.org/lapack/lug/node145.html [Accedido el 12 Jun. 2016].

Scarpino, M. (2016). *Crunching Numbers with AVX and AVX2 - CodeProject*. [online] Codeproject.com. Disponible en: http://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX [Accedido el 12 Jun. 2016].

Univ-orleans.fr. (2016). *SIMD: Main Page*. [online] Disponible en: http://www.univ-orleans.fr/lifo/Members/Sylvain.Jubertie/doc/SIMD/html/index.html [Accedido el 12 Jun. 2016].

Msdn.microsoft.com. (2016). *Streaming SIMD Extensions (SSE)*. [online] Disponible en: https://msdn.microsoft.com/en-us/library/t467de55(v=vs.90).aspx [Accedido el 12 Jun. 2016].

Intel. (2016). *Tecnología Intel® Turbo Boost 2.0*. [online] Disponible en: http://www.intel.la/content/www/xl/es/architecture-and-technology/turbo-boost/turbo-boost-technology.html [Accedido el 12 Jun. 2016].

Tommesani, S. (2016). *SSE Intrinsics - Stefano Tommesani*. [online] Tommesani.com. Disponible en: http://www.tommesani.com/index.php/simd/73-sse-intrinsics.html [Accedido el 12 Jun. 2016].

Es.wikipedia.org. (2016). *Unidad de coma flotante*. [online] Disponible en: https://es.wikipedia.org/wiki/Unidad_de_coma_flotante [Accedido el 12 Jun. 2016].

Msdn.microsoft.com. (2016). *Ventajas de utilizar archivos DLL*. [online] Disponible en: https://msdn.microsoft.com/es-es/library/dtba4t8b.aspx [Accedido el 12 Jun. 2016].

Wikipedia. (2016). *OpenMP*. [online] Disponible en: https://en.wikipedia.org/wiki/OpenMP [Accedido el 12 Jun. 2016].

Basic Linear Algebra Subrprogramas (BLAS1) (2015). [pdf]

Summary of OpenMP 3.0 C/C++ Syntax, (2016). [online] Disponible en: http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf [Accedido el 12 Jun. 2016].

Finis, J. (2016). [online] Disponible en: https://db.in.tum.de/~finis/x86-intrin-cheatsheet-v2.1.pdf [Accedido el 12 Jun. 2016].

Phillips, C. (2014). [online] Disponible en: http://www.ira.inaf.it/difx-2014/presentations/DIFX_2014-SIMD.pdf [Accedido el 12 Jun. 2016].

Icl.utk.edu. (2016). *Documentation*. [online] Disponible en: http://www.icl.utk.edu/~mgates3/docs/ [Accedido el 12 Jun. 2016].