

▪ Proyecto Fin de Grado ▪

Ingeniería de Computadores

Programación heterogénea. Aplicación al cálculo matricial.

Autor

Ainhoa Havelka

Director

Clemente Rodríguez

Junio 2017

Agradecimientos

En primer lugar y especialmente a mi director de proyecto, Clemente, por su dedicación, por todo el camino recorrido, todo lo que me ha enseñado y ayudado y todo lo que me ha aportado en general, tanto a nivel académico como a nivel personal.

A mi amigo Igor S., por haberse ofrecido a leer este documento en un estado prematuro, habérselo leído detenida y atentamente y haberme dado su opinión sincera y minuciosa.

A mis compañeros de clase y amigos, Iñaki G., Patxi L., Jon Iker L. y Bienvenido A., por estar ahí, apoyarme y ayudarme (y aguantarme también un poco), no solo durante el proyecto, sino también a lo largo de toda la carrera.

A mis compañeros de Magna SIS, por haber enriquecido la experiencia en general, y haber hecho de la facultad un lugar más acogedor si cabe.

A mi familia, mis padres y especialmente mi pareja Santi, por su confianza y apoyo incondicional.

Muchas gracias a todos.



Ainhoa Havelka, 2017

Esta obra está bajo una licencia de *Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional*.

Para ver una copia de esta licencia visitar el siguiente enlace:

<https://creativecommons.org/licenses/by-nc-sa/4.0/>.

El reconocimiento se deberá hacer indicando los nombres y apellidos del autor.

Resumen

A lo largo de este proyecto se ha experimentado con diferentes técnicas y herramientas de optimización, con el objetivo de **estudiar el rendimiento que se puede obtener por programar siendo conscientes de la arquitectura.**

Para ello se ha trabajado tanto de forma **explícita** (optimizando manualmente los programas) como de forma **implícita** (utilizando la optimización y el *feedback* que ofrecen diversas herramientas y compiladores) para contrastar los resultados obtenidos y comprobar el potencial de las herramientas automatizadas:

- **Compiladores:** *GCC* e *Intel® C++ Compiler*
- **Programación:** *lenguaje de programación C* (en algún caso también la opción */C99*) y *NVIDIA CUDA* para la gestión de la GPU
- **Técnicas:** vectorización mediante *intrínsecas*, paralelización con *OpenMP*
- **Herramientas:** *Intel® Advisor*, *Intel® Roofline*, *Intel® VTune Amplifier*, *Intel® Roofline*
- **Librerías:** *Intel® Math Kernel Library (MKL)*

En el *capítulo 1* se proporciona el mínimo conocimiento necesario para entender el fondo de la memoria.

Se han realizado pruebas en varias **CPUs** diferentes, también en una **GPU**, con el objetivo de ver las diferencias entre unas y otras, y estudiar el comportamiento de los programas según el sistema de cómputo en el que se ejecutan. El *capítulo 4* se ha centrado en el análisis del rendimiento de la GPU.

Las rutinas en las que se ha basado la experimentación forman parte de la especificación **BLAS** (*Basic Linear Algebra Subprograms*), tanto de complejidad cuadrática (*BLAS2*) como de complejidad cúbica (*BLAS3*). Las operaciones matriciales se han realizado con matrices densas, diagonales, en banda y triangulares. El *capítulo 2* analiza el comportamiento de *BLAS2* y el *capítulo 3* de *BLAS3*.

Se han establecido los criterios para determinar las circunstancias en las que utilizar los diferentes sistemas de cómputo atendiendo al tamaño de la estructura de datos y el algoritmo utilizado. En el *capítulo 5* se expone una recopilación de los resultados obtenidos.

Índice

Resumen.....	iv
Índice	vi
Lista de Figuras y Tablas.....	viii
Introducción, visión general y nociones básicas.....	1
1.1 Visión general.....	1
1.2 BLAS.....	2
1.2.1 Qué son las especificaciones <i>BLAS</i>	2
1.2.2 Nomenclatura de las especificaciones <i>BLAS</i>	3
1.3 Conceptos básicos.....	5
1.3.1 Sistemas de Cómputo.....	5
1.3.2 Compiladores.....	12
1.3.3 Herramientas	13
1.3.3.4 <i>Intel® Advisor</i> – herramienta <i>Roofline</i>	17
1.4 Equipos utilizados para realizar las pruebas.....	19
Especificación BLAS 2.....	21
2.1 GNU Compiler Collection (<i>GCC</i>).....	21
2.1.1 Secuencial.....	22
2.1.2 Paralelismo explícito (con <i>OpenMP</i>).....	33
2.1.3 Vectorización explícita (con intrínsecas).....	36
2.1.4 Paralelización y vectorización.....	44
2.1.5 Experimentación.....	50
2.2 <i>Intel® C++ Compiler (ICC)</i>	61
2.2.1 Configuración del entorno.....	61
2.2.2 Autoparalelización y autovectorización e <i>Intel® Advisor</i>	64
2.2.4 Librerías <i>MKL</i>	70
2.2.5 <i>Intel® Advisor</i> – <i>Roofline</i>	75
2.3 Comparación de resultados obtenidos.....	77
Especificación BLAS 3.....	79
3.1 Algoritmos.....	79
3.1.1 Algoritmo secuencial.....	80
3.1.2 Algoritmo de intercambio de bucles ($i, j, k \rightarrow i, k, j$).....	80
3.1.3 Algoritmo de “blocking”	82
3.2 Auto-optimización.....	83

3.2.1 Reportes del compilador ICC.....	83
3.2.2 <i>Intel® VTune Amplifier XE – Hotspot analysis</i>	86
3.2.3 <i>Intel® Advisor</i>	86
3.2.4 <i>Intel® Roofline</i>	89
3.3 Librerías <i>MKL</i>	90
3.4 Resultados.....	90
GPU - NVIDIA CUDA.....	93
4.1 BLAS de nivel 2.....	94
4.2 BLAS de nivel 3.....	97
4.3 Resultados y comparativa general.....	100
Conclusiones y líneas abiertas.....	103
Bibliografía y referencias.....	108

Lista de Figuras y Tablas

FIGURAS

Figura 1.1 Visión general.....	2
Figura 1.2 Especificación BLAS de nivel 2.....	4
Figura 1.3 Especificación BLAS de nivel 3.....	5
Figura 1.4 Pipeline superescalar.....	6
Figura 1.5 Tecnología Hyperthreading.....	7
Figura 1.6 Intel® Intrinsic Guide.....	8
Figura 1.7 Operaciones vectoriales.....	9
Figura 1.8 Logotipo NVIDIA CUDA.....	10
Figura 1.9 Estructura de Threads (en Grid y Bloques) de CUDA.....	11
Figura 1.10 Esquema de ejecución CUDA.....	11
Figura 1.11 Logotipo GCC.....	12
Figura 1.12 Logotipo ICC.....	13
Figura 1.13 Logotipo MVS.....	13
Figura 1.14 Guía de creación de proyecto Visual Studio - 1.....	14
Figura 1.15 Guía de creación de proyecto Visual Studio - 2.....	14
Figura 1.16 Guía de creación de proyecto Visual Studio - 3.....	15
Figura 1.17 Ejemplo de uso de Intel® Vtune Amplifier.....	16
Figura 1.18 Ejemplo de uso de Intel® Advisor.....	17
Figura 1.19 Intel® Advisor - Roofline.....	18
Figura 2.1 Esquema general – algoritmos secuenciales.....	22
Figura 2.2 Matrices triangulares inferiores	24
Figura 2.3 Operaciones con las matrices triangulares inferiores.....	24
Figura 2.4 Matrices triangulares inferiores.....	25
Figura 2.5 Operaciones con las matrices triangulares superiores.....	25
Figura 2.6 Matriz triangular en banda inferior.....	27
Figura 2.7 Matriz triangular en banda inferior con restricción.....	27
Figura 2.8 Matriz triangular en banda superior.....	28
Figura 2.9 Matriz triangular en banda superior con restricción.....	28
Figura 2.10 Esquema general – función genérica.....	29
Figura 2.11 Matriz en banda general.....	30
Figura 2.12 Esquema general - paralelismo.....	33
Figura 2.13 Reparto estático (chunk = 10) de 32 elementos.....	33
Figura 2.14 Ejemplo de reparto estático de 100 filas entre 4 threads.....	34
Figura 2.15 Esquema general - vectorización.....	36
Figura 2.16 Ejemplo de alineamiento de un vector.....	38

Figura 2.17 Casos de alineamiento entre 'x' y la fila de la matriz.....	38
Figura 2.18 Nomenclatura de las variables de vectorización.....	39
Figura 2.19 Cuerpo del algoritmo vectorial.....	42
Figura 2.20 Suma de todos los elementos de un vector con "hadd" - SSE.....	43
Figura 2.21 Esquema general – paralelización y vectorización.....	44
Figura 2.22 Ejemplo de reparto de filas (paralelo-vectorial).....	46
Figura 2.23 Suma de todos los elementos de un vector con "hadd" - AVX.....	48
Figura 2.24 Visión general – i7 6700K.....	53
Figura 2.25 Comparativa general optimizaciones 1 – i7 6700K.....	54
Figura 2.26 Comparativa general optimizaciones 2 – i7 6700K.....	55
Figura 2.27 Discriminación de uso y speed-up 1 – i7 6700K.....	56
Figura 2.28 Relación de las configuraciones y la memoria caché.....	57
Figura 2.29 Explicación ejemplo con AVX y procesador i7 6700K.....	58
Figura 2.30 Discriminación de uso y speed-up 2 – i7 6700K.....	58
Figura 2.31 Discriminación de uso y speed-up 3 – i7 6700K.....	59
Figura 2.32 Configuración del entorno - 1.....	61
Figura 2.33 Configuración del entorno - 2.....	62
Figura 2.34 Configuración del entorno - 3.....	63
Figura 2.35 Autoparalelización / autovectorización – Bucle principal.....	65
Figura 2.36 Autoparalelización / autovectorización – Bucle interior.....	65
Figura 2.37 Fichero “seq.cpp” - sin optimizaciones.....	66
Figura 2.38 Análisis Intel® Advisor.....	66
Figura 2.39 Análisis paralelización Intel® Advisor – antes.....	68
Figura 2.40 Análisis paralelización Intel® Advisor – después.....	69
Figura 2.41 Intel® Advisor – Roofline; BLAS2.....	75
Figura 2.42 Intel® Roofline – Tiempos de ejecución.....	76
Figura 2.43 Comparativa general de resultados.....	77
Figura 3.1 Visualización del recorrido en índices de las matrices.....	81
Figura 3.2 Algoritmo MxM “blocking”.....	82
Figura 3.3 Algoritmo secuencial – reportes ICC.....	83
Figura 3.4 Algoritmo de intercambio de bucles – reportes ICC.....	84
Figura 3.5 Algoritmo de blocking – reportes ICC.....	85
Figura 3.6 Intel® VTune Amplifier XE – Algoritmos BLAS3.....	86
Figura 3.7 Intel® Advisor – Algoritmos BLAS3.....	86
Figura 3.8 Análisis paralelización intercambio de bucles - antes.....	87
Figura 3.9 Análisis paralelización intercambio de bucles - después.....	88
Figura 3.10 Análisis paralelización algoritmo “blocking” - antes.....	88
Figura 3.11 Análisis paralelización algoritmo “blocking” - después.....	88
Figura 3.12 Intel® Roofline - BLAS3.....	89
Figura 4.1 Esquema general – CUDA.....	94
Figura 4.2 Explicación función convencional CUDA BLAS2.....	95
Figura 4.3 Explicación función “blocking” CUDA BLAS2.....	97
Figura 4.4 Explicación función básica (convencional) CUDA BLAS3.....	98
Figura 4.5 Explicación función “blocking” CUDA BLAS3.....	100

Índice de tablas

Tabla 1.1 Taxonomía de Flynn.....	6
Tabla 1.2 Clasificación de intrínsecas 1.....	9
Tabla 1.3 Clasificación de intrínsecas 2.....	9
Tabla 1.4 Equipos utilizados para la realización de las pruebas.....	19
Tabla 2.1 Casuísticas de matrices según KL[inf] y KU[sup].....	30
Tabla 2.2 Ejemplo - reparto de filas para el algoritmo paralelo-vectorial..	45
Tabla 2.3 Casuísticas de matrices según inf y sup (%).....	50
Tabla 2.4 Resumen de los criterios de uso <i>BLAS2</i> (opt. explícita).....	60
Tabla 2.5 Comparativa general de resultados; datos concretos.....	77
Tabla 2.6 Resumen general de los criterios de uso (compiladores, <i>MKL</i>)..	78
Tabla 3.1 Resultados (ticks) de las pruebas realizadas – <i>BLAS3</i>	90
Tabla 3.2 Criterios de uso de los algoritmos <i>BLAS3</i>	91
Tabla 4.1 Datos <i>CUDA BLAS2</i>	100
Tabla 4.2 Datos <i>CUDA BLAS3</i>	101
Tabla 4.3 Comparación de ancho de banda y <i>GFLOPS</i> – <i>BLAS2</i> y 3.....	102
Tabla 4.4 Criterios de uso de la <i>GPU</i>	102
Tabla 5.1 Resumen de los criterios de uso <i>BLAS2</i> (opt. explícita).....	104
Tabla 5.2 Resumen general de los criterios de uso (compiladores, <i>MKL</i>).	105
Tabla 5.3 Criterios de uso de los algoritmos <i>BLAS3</i>	105
Tabla 5.4 Criterios de uso de la <i>GPU</i>	106
Tabla 5.5 Niveles de conocimientos propios.....	106

1

Introducción, visión general y nociones básicas

En este capítulo se expondrá una visión general del proyecto para comprender su desarrollo.

1.1 Visión general

El objetivo del proyecto es ver el rendimiento que se puede obtener por programar siendo conscientes de la arquitectura, o en su caso, siendo conscientes del sistema de cómputo en el que se está trabajando. Las funciones con las que se va a trabajar son funciones de cálculo de matriz por vector y matriz por matriz de las especificaciones de *BLAS de nivel 2* y *BLAS de nivel 3*.

Todas las funciones de *BLAS de nivel 2* han sido resumidas en una única función genérica. Ésta es capaz de dar a cada tipo de matriz el tratamiento adecuado según su tamaño en filas y columnas y su número de diagonales inferiores y superiores (su *banda*).

En función del compilador con el que estemos trabajando (*GNU Compiler Collection [GCC]* o *Intel® C++ Compiler [ICC]*) y en función de los parámetros nombrados se decide qué tipo de optimización aplicar: *secuencial*, *paralelo*, *vectorial [SSE/AVX]*, *paralelo-vectorial* o *GPU* (siempre, claro, que se dispongan de los medios físicos necesarios); *Figura 1.1*.

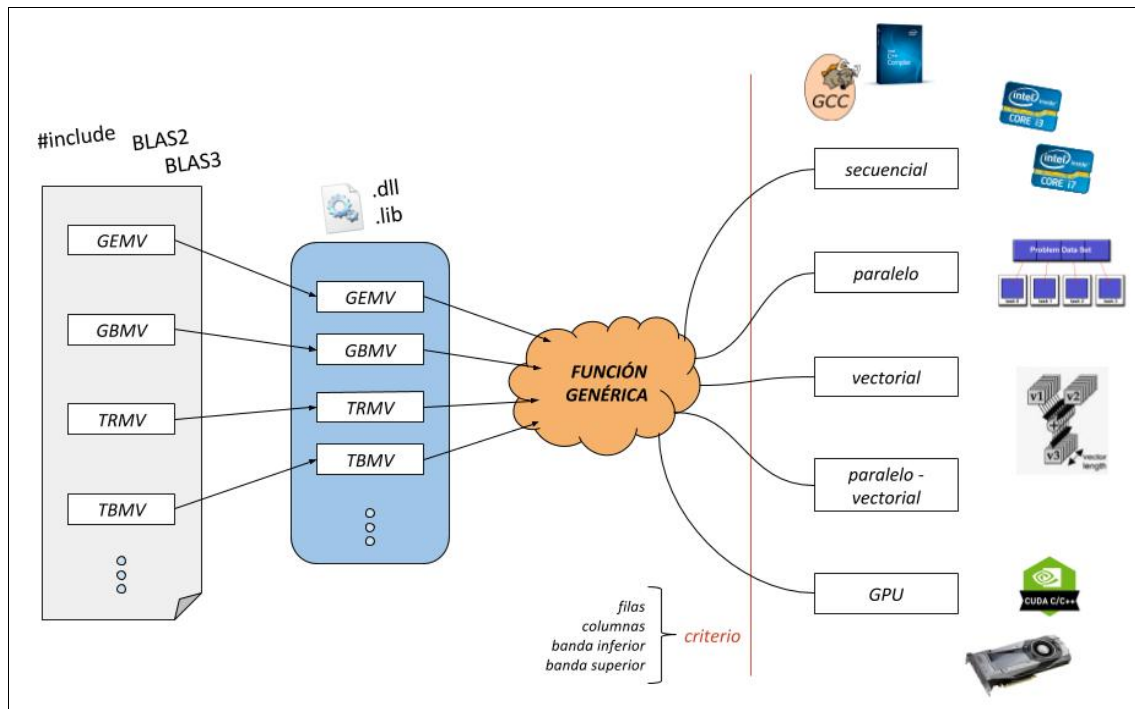


Figura 1.1 Visión general

1.2 BLAS

A continuación se darán unas nociones básicas sobre qué son las especificaciones *BLAS* y su nomenclatura.

1.2.1 Qué son las especificaciones BLAS

BLAS o *Basic Linear Algebra Subprograms*, son una serie de especificaciones que prescriben un conjunto de rutinas de bajo nivel para la realización de operaciones de álgebra lineal comunes.

Las rutinas están organizadas en tres **subcategorías** llamados **niveles** (nivel 1, nivel 2 y nivel 3). Dichos niveles corresponden tanto al orden cronológico de definición y publicación de las definiciones, como al grado de **complejidad polinomial** de las operaciones que describen. El nivel 1 corresponde a operaciones de vector por vector de complejidad lineal – $O(n)$. El nivel 2 corresponde a operaciones de matriz por vector de complejidad cuadrática – $O(n^2)$, y el nivel 3 corresponde a operaciones de matriz por matriz de complejidad cúbica – $O(n^3)$.

1.2.2 Nomenclatura de las especificaciones BLAS

La forma de nombrar a las diferentes funciones especificadas por *BLAS* está normalizada. A continuación se describirá la nomenclatura que ha sido utilizada en este proyecto.

1.2.2.1 Nomenclatura BLAS 2

En primer lugar la especificación *BLAS de nivel 2*.

Para empezar definiremos el **nombre** de las diferentes funciones. Su patrón general es $x\text{-}YY\text{-}MV^1$, donde:

- 'x' hace referencia al tipo de dato utilizado (*float, double, ...*)
- 'YY' hace referencia al tipo de matriz que será utilizada en la operación. Puede ser sustituido por uno de los siguientes:
 - **GE** – General (matriz densa)
 - **GB** – General en **B**anda (matriz general en banda, puede ser diagonal, banda inferior, banda superior, o banda inferior y superior)
 - **TR** – Triangular (solo matrices cuadradas)
 - **TB** – Triangular en **B**anda (ídem anterior, solo matrices cuadradas)

También se especifica para cada función los **parámetros** que admite. Algunos de los parámetros más significativos son:

- **A** – Es la matriz con la que se va a operar.
- **x, y** – Son los vectores con los que se va a operar.
- **UPLO** – En caso de que estemos en una función para matrices triangulares, este parámetro especifica si la matriz es triangular inferior o superior.

¹ Los guiones solo se han incluido como separadores visuales; en la especificación no aparecen

- *TRANx* – Especifica si se debe hacer el cálculo con la traspuesta de la matriz indicada o no.
- *DIAG* – Especifica si la matriz es una matriz con diagonal unidad o no.
- *M, N* – Especifican las dimensiones de la matriz (*filas* y *columnas* respectivamente). En algunas funciones, como las de matrices triangulares, solo es necesario indicar *N*, ya que se opera únicamente con matrices cuadradas.
- *KL, KU, K* – Especifican el número de diagonales inferiores (*KL*) y el número de diagonales superiores (*KU*). Si se trata de una matriz triangular en banda (*TBMV*), solo es necesario indicar una banda (dependiendo de si la matriz es triangular inferior en banda o triangular superior en banda, *K* será el número de diagonales inferiores o superiores).
- *alpha, beta* – Indican los escalares por los que se multiplicarán los vectores '*x*' e '*y*' (*alpha x; beta y*).

Finalmente, para cada función se especifica el **cálculo** que realiza.

- Para matrices densas (*GE*) y matrices en banda general (*GB*):

$$y \leftarrow A \alpha x + \beta y$$

- Para matrices triangulares (*TR*) y matrices triangulares en banda (*TB*):

$$x \leftarrow A x$$

En la *Figura 1.2* podemos ver el subconjunto de rutinas de *BLAS2* tratadas en el proyecto, extraídas directamente de la *Quick Reference Guide de la universidad de Tennessee*.

Level 2 BLAS								
	options	dim	b-width	scalar	matrix	vector	scalar vector	
xGEMV (TRANS,	M, N,		ALPHA, A,	LDA, X,	INCX, BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$
xGBMV (TRANS,	M, N, KL, KU,		ALPHA, A,	LDA, X,	INCX, BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$
xTRMV (UPLO, TRANS, DIAG,	N,		A,	LDA, X,	INCX)	$x \leftarrow Ax$	
xTBMV (UPLO, TRANS, DIAG,	N, K,		A,	LDA, X,	INCX)	$x \leftarrow Ax$	

Figura 1.2 Especificación BLAS de nivel 2

Se han mantenido las definiciones tal cual aparecen en la *Figura 1.2* con la salvedad de haber omitido el uso de los escalares *alpha / beta*, ya que se considera que no aportan mayor dificultad y no se ha visto necesario. Además, se han implementado solamente para el tipo de datos *float* (precisión simple). Esta precisión es la que permite obtener mayor rendimiento en la vectorización (ya que permite trabajar los vectores de cuatro en cuatro elementos en el caso de SSE y de ocho en ocho en el caso de AVX, respecto a la precisión doble que solo permite la mitad).

1.3.1.1 Paralelismo a nivel de instrucción – *procesamiento superescalar*

Los procesadores utilizados en el proyecto, al igual que la mayoría de los procesadores existentes hoy día, son procesadores **superescalares**.

Un procesador superescalar es una CPU que implementa una forma de paralelismo llamada *paralelismo a nivel de instrucción* (o en inglés: *instruction-level parallelism*). Esta forma de procesamiento permite obtener mayor *throughput* (número de instrucciones que pueden ser ejecutadas en una unidad de tiempo) ya que es capaz de **ejecutar más de una instrucción por ciclo de reloj**. Se mide en ciclos por instrucción (CPI) y su uso es habitual como medida de rendimiento (como veremos en las herramientas de *Intel®*, por ejemplo, en el *Intel® Advisor*). El término *superescalar* se emplea en contraposición a la *microarquitectura escalar*, que solo puede ejecutar una instrucción por cada ciclo de reloj.

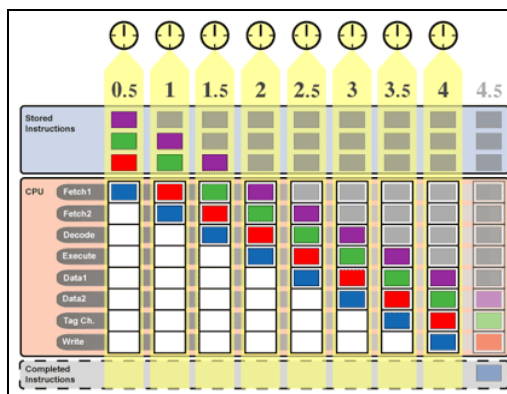


Figura 1.4 Pipeline superescalar

fuentes: <https://arstechnica.com/>

Un procesador *superescalar* es capaz de ejecutar más de una instrucción por ciclo de reloj ya que envía simultáneamente varias instrucciones diferentes a unidades funcionales diferentes. Cada una de estas unidades funcionales no es un procesador en sí mismo, sino un recurso computacional dentro del propio procesador (como puede ser por ejemplo una *unidad aritmético-lógica* o *ALU* o una unidad de lectura/escritura en memoria). Podemos ver un *pipeline* de una arquitectura *superescalar* en la *Figura 1.4*.

En la *taxonomía de Flynn* (*Tabla 1.1*) los procesadores *superescalares* mononúcleo están clasificados como *SISD* (del inglés 'Single Instruction stream, Single Data stream' – un flujo de instrucciones, un flujo de datos).

Muchos procesadores *superescalares* hoy día soportan paralelismo a nivel de datos (operaciones vectoriales), por lo que un procesador mononúcleo de esta índole podría clasificarse como *SIMD* (del inglés 'Single Instruction stream, Multiple Data stream' – un flujo de instrucciones, múltiples flujos de datos) – apartado 1.3.1.3 de la memoria.

Finalmente, si hablásemos de procesadores *superescalares* multinúcleo se clasificarían como *MIMD* (del inglés 'Multiple Instruction stream, Multiple Data stream' – múltiples flujos de instrucciones y múltiples flujos de datos).

	Un flujo de instrucciones	Múltiples flujos de instrucciones
Un flujo de datos	<i>SISD</i>	<i>MISD</i>
Múltiples flujos de datos	<i>SIMD</i>	<i>MIMD</i>

Tabla 1.1 Taxonomía de Flynn

1.3.1.2 Paralelismo a nivel de tareas - procesamiento *paralelo*

El paralelismo a nivel de tareas es un tipo de paralelismo en el que **la carga de trabajo se divide** en tareas más pequeñas que puedan ser llevadas a cabo simultáneamente. Estas tareas se reparten entre los núcleos y/o **threads** (hilos) del procesador.

Además, con el paralelismo a nivel de tareas podemos aprovechar una tecnología de los procesadores llamada *Simultaneous Multithreading* (SMT, en castellano, 'multihilos simultáneos'). Esta tecnología simula varios procesadores lógicos en un mismo procesador físico. No todos los procesadores incorporan esta tecnología. Además, el grado de *Multithreading* (cuántos hilos lógicos crea dentro de cada procesador físico) depende de cada modelo. En el caso de *Intel®* la tecnología recibe el nombre de *HyperThreading*, y la marca afirma que consigue una mejora en el rendimiento de aproximadamente un 60% (Figura 1.5).

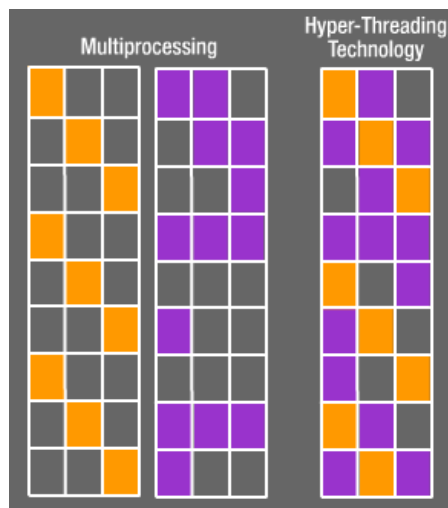


Figura 1.5 Tecnología Hyperthreading
fuente: <https://software.intel.com/>

En este proyecto todas las pruebas se han realizado con cuatro *threads*. En el caso de los procesadores *Intel® Core i7* corresponde al número de procesadores físicos de los que se dispone. En el caso del procesador *Intel® Core i3*, que solo dispone de dos núcleos físicos, éste utilizará la tecnología de *HyperThreading* de grado dos (dos procesadores lógicos dentro de cada procesador físico), para poder ejecutar las rutinas con cuatro *threads*.

Dado que el uso del paralelismo a nivel de tareas es “opcional” (es decir, no es implícito de la arquitectura, como sucede con el paralelismo a nivel de instrucciones), se deberá decidir en qué ocasiones conviene o no aprovechar su potencial. Habrá ocasiones en las que no merezca la pena: cuando la carga de trabajo a repartir sea lo suficientemente pequeña como para que el hecho de repartirla nos cueste más tiempo del que luego ahorraremos (**overhead**). Sin embargo, con cargas de trabajo suficientemente grandes el **overhead** empezará a ser despreciable respecto a la carga computacional, y es entonces cuando obtenemos mayor eficiencia con el procesamiento paralelo. Éste será uno de los objetivos de análisis: establecer los criterios de uso.

En el proyecto se han utilizado dos formas de implementar el paralelismo. Por un lado de forma explícita, es decir, programando manualmente el paralelismo utilizando la *API* (*Application Programming Interface*) **OpenMP v3.1**. Por otro lado de forma implícita, dejando que el compilador *Intel® C++ Compiler* paralelice las rutinas de forma automática. Cada método se explicará correspondientemente en los apartados 2.1.2 y 2.2.2 del capítulo 2 y el apartado 3.2 del capítulo 3.

1.3.1.3 Paralelismo a nivel de datos – *procesamiento vectorial*

El paralelismo a nivel de datos se centra en **distribuir los datos en diferentes nodos o unidades funcionales** que operan con los datos en paralelo. Se puede aplicar a estructuras de datos regulares como vectores y matrices trabajando cada elemento (o conjuntos de elementos) simultáneamente.

El conjunto de instrucciones utilizado son las **intrínsecas** de *Intel*[®], que permiten utilizar este potencial sin la necesidad de escribir código en ensamblador. Se pueden subdividir según la tecnología y según su fecha de publicación. También se pueden diferenciar según el tamaño de los vectores (en bits) con los que operan y las librerías (*includes*) necesarias para su uso (*Tabla 1.2*).

También se pueden clasificar por el tipo de datos que utilizan y su correspondiente nomenclatura para las instrucciones. Según el tipo de tecnología y el tipo de dato que se emplea, se podrán procesar más o menos elementos de vector a la vez. Esto queda reflejado en la *Tabla 1.3*. Esta clasificación solo se ha hecho para las tecnologías utilizadas en el proyecto (*SSE, AVX*).

Toda la información acerca de las funciones para utilizar las intrínsecas (*Figura 1.6*) está disponible en la página web de *Intel*[®] bajo el nombre de “*Intel*[®] *Intrinsics Guide*” o “*Guía de intrínsecas de Intel*[®]” (URL disponible en el apartado de *Bibliografía y referencias*).

La implementación realizada mediante el uso de las intrínsecas se explicará en su correspondiente apartado del *capítulo 2* de la memoria (*apartado 2.1.3*).

Al igual que sucedía con el paralelismo a nivel de tareas, también se ha implementado el paralelismo a nivel de datos de forma implícita dejando que el compilador *Intel*[®] *C++ Compiler* vectorice las rutinas de forma automática (*apartado 2.2.2 capítulo 2; apartado 3.2 capítulo 3*).

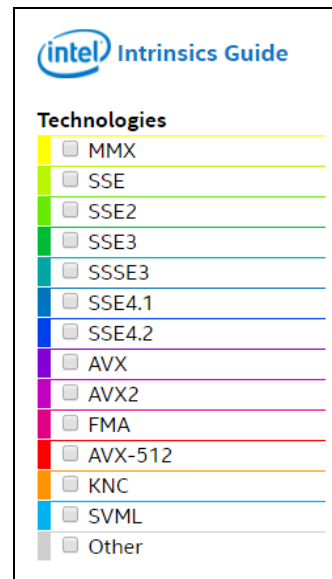


Figura 1.6 Intel[®] Intrinsic Guide

Tecnología	Publicación de la primera versión	Versiones	Includes	Tamaño del vector
MMX	1997	MMX	<mmintrin.h>	64 bits
SSE	1999	SSE SSE2 SSE3 SSE4.1 SSE4.2	<xmmintrin.h> <emmintrin.h> <pmmintrin.h> <smintrin.h> <nmmmintrin.h>	128 bits
AVX	2008	AVX AVX2	<immintrin.h>	256 bits
		AVX512		512 bits

Tabla 1.2 Clasificación de intrínsecas 1

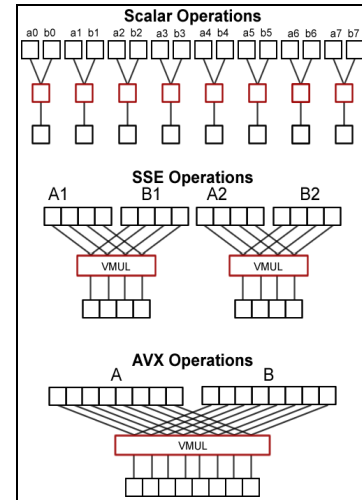


Figura 1.7 Operaciones vectoriales con intrínsecas SSE y AVX

Tecnología	Tamaño del vector	Tipo de dato	Tamaño del dato	Nomenclatura	Número de elementos del vector
SSE	128 bits	enteros	8 bits	i8	16 elementos
			16 bits	i16	8 elementos
			32 bits	i32	4 elementos
			64 bits	i64	2 elementos
			128 bits	i128	1 elemento
		precisión simple (float)	32 bits	ps	4 elementos
		precisión doble (double)	64 bits	pd	2 elementos
AVX	256 bits	enteros	8 bits	i8	32 elementos
			16 bits	i16	16 elementos
			32 bits	i32	8 elementos
			64 bits	i64	4 elementos
			128 bits	i128	2 elementos
			256 bits	i256	1 elemento
		precisión simple (float)	32 bits	ps	4 elementos
		precisión doble (double)	64 bits	pd	2 elementos

Tabla 1.3 Clasificación intrínsecas 2

1.3.1.4 GPU – NVIDIA CUDA

CUDA (*Compute Unified Device Architecture*; Figura 1.8) es una **arquitectura de cálculo paralelo** de NVIDIA que aprovecha la potencia de la **GPU** (unidad de procesamiento gráfico) para proporcionar un incremento del rendimiento. CUDA intenta aprovechar las ventajas de las GPU frente a las CPU de propósito general. Las GPU constan de un número de núcleos mucho más elevado que las CPU, que a su vez permite lanzar un número de hilos aún mayor. Para aplicaciones con **muchos hilos que realizan tareas independientes**, como por ejemplo el procesamiento gráfico, una GPU podrá ofrecer un gran rendimiento.

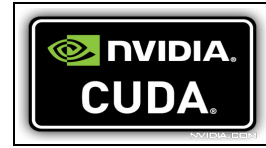


Figura 1.8 Logotipo NVIDIA
CUDA

Este modo de paralelismo también ha sido utilizado en el proyecto para implementar las rutinas de BLAS2 y 3. A continuación se expondrá la estructura general de un programa CUDA (Programa 1.1).

```
void miFuncion(float *host_data, float *param2, ...)  
{  
    static float* device_data;  
  
    // Reservar memoria para los datos del "device" (GPU)  
    CUDA_SAFE_CALL( cudaMalloc((void**)&device_data, size_data) );  
  
    // Copiar memoria del "host" (CPU) al "device" (GPU)  
    CUDA_SAFE_CALL( cudaMemcpy( device_data, host_data, size_data,  
                               cudaMemcpyHostToDevice) );  
  
    // Configurar parámetros para la ejecución  
    int threads = BLOCK_SIZE;  
    int grid = (nfilas+threads-1)/threads;  
  
    // Llamar al programa CUDA implementado  
    miFuncion_CUDA<<< grid, threads >>>(device_data);  
  
    // Copiar la memoria del "device" (GPU) al "host" (CPU)  
    CUDA_SAFE_CALL( cudaMemcpy( host_data, device_data, size_data,  
                               cudaMemcpyDeviceToHost) );  
  
    // Liberar la memoria del "device" (GPU)  
    CUDA_SAFE_CALL( cudaFree(device_data) );  
}
```

Programa 1.1 Estructura general de un programa CUDA

En resumen, se deben seguir seis pasos fundamentales:

1. Reservar memoria en la GPU.
2. Copiar los datos necesarios de la memoria de la CPU a la memoria de la GPU.

3. Configurar los parámetros para la ejecución. En una *GPU*, los *threads* se organizan por bloques. El conjunto de los bloques se denomina *grid* (Figura 1.9). En este caso las variables *threads* y *grid* hacen referencia a lo siguiente:

- *threads*: define el número de *threads* por bloque.
- *grid*: define el tamaño en bloques del *grid* (número de bloques).

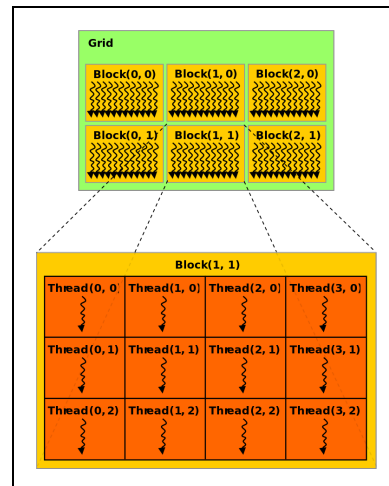


Figura 1.9 Estructura de Threads (en Grid y Bloques) de CUDA

4. Llamar al programa *CUDA* que hayamos implementado.
5. Copiar de vuelta los datos de la memoria de la *GPU* a la memoria de la *CPU* una vez se haya ejecutado nuestro programa.
6. Liberar la memoria de la *GPU*.

En la Figura 1.10 podemos ver el esquema de ejecución de programas *CUDA*.

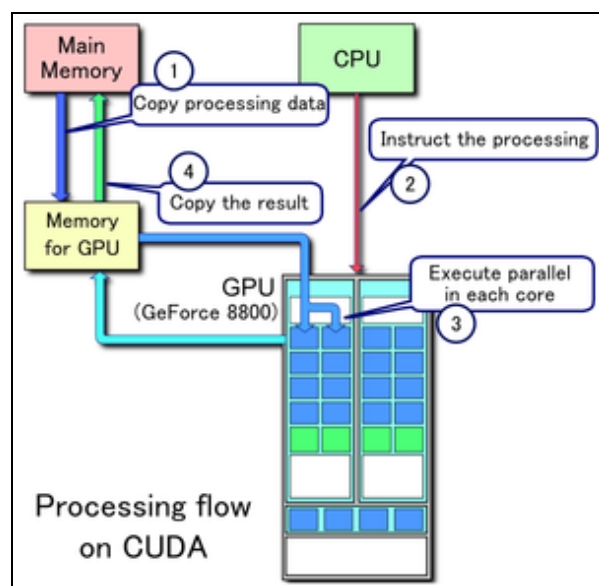


Figura 1.10 Esquema de ejecución *CUDA*

Se explicarán en mayor detalle los programas desarrollados en *CUDA* para *BLAS2* y *BLAS3* en el capítulo 4 de la memoria.

1.3.2 Compiladores

En este apartado se expondrán brevemente los dos compiladores utilizados en el proyecto.

1.3.2.1 GNU Compiler Collection – GCC

Uno de los compiladores utilizados ha sido el compilador de C perteneciente al conjunto de compiladores de *GNU Compiler Collection* (comúnmente “GCC”). La fundación de software libre (*FSF – Free Software Foundation*) distribuye GCC bajo la licencia conocida como *GNU General Public License (GNU GPL)*. GCC es el compilador estándar de la mayoría de Sistemas Operativos tipo Linux.

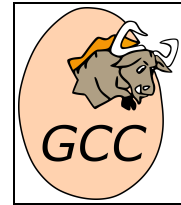


Figura 1.11 Logotipo GCC

Sin embargo, en este proyecto se ha utilizado *Windows* como Sistema Operativo para el desarrollo. Por ello, se ha instalado el *GCC* mediante la plataforma conocida como *MinGW (minimalist GNU for Windows)*.

GCC se ha utilizado para programar todo lo respectivo a la paralelización explícita con *OpenMP* y la vectorización explícita mediante *intrínsecas* (presentado en el apartado 2.1 del capítulo 2 de la memoria).

Se han empleado en total tres formas diferentes de compilar los ficheros *.c. Una para compilar los ficheros cuyo código debía ejecutarse estrictamente en secuencial. Otra para los que se debían ejecutar estrictamente en paralelo (con *OpenMP*). Finalmente una para los ficheros cuyo código tenía tanto partes paralelas (*OpenMP*) como partes vectoriales (*intrínsecas*), además de para crear el fichero ejecutable final.

1. Secuencial

```
>> gcc.exe -O3 -c fichero_seq.c
```

Esto genera el fichero '*fichero_seq.o*'.

2. Paralelo

```
>> gcc.exe -O3 -fopenmp -c fichero_omp.c
```

Esto genera el fichero '*fichero_omp.o*'.

3. Paralelo + Vectorial + fichero ejecutable

Ahora en la compilación se añaden los *flags* correspondientes al conjunto de instrucciones de *SSE (-msse4.1)* y *AVX (-mavx2)*. Además, se utiliza '-o' para enlazar los ficheros objeto anteriormente creados y añadir además a la compilación el fichero principal (*main.c*).

```
>> gcc.exe -O3 -fopenmp -msse4.1 -mavx2 -o ejecutable.exe main.c  
fichero_seq.o fichero_omp.o
```

1.3.2.2 Intel® C++ Compiler

El otro compilador utilizado ha sido el *Intel® C++ Compiler* (también *ICC*). Se trata de un compilador propietario de la empresa *Intel®*. Para poder descargarlo y utilizarlo ha sido necesario solicitar una licencia de estudiante del *Intel® Parallel Studio XE Cluster Edition* mediante el correo electrónico personal de la *UPV/EHU* (enlace en el apartado de Bibliografía y referencias). Se ha trabajado con él mediante su integración en la herramienta de *Microsoft Visual Studio 2012*.

Este compilador se ha utilizado principalmente para comprobar su potencial frente al compilador *GCC*, dejando que autoparalelice y autovectorice él mismo el código, sin hacerlo de forma explícita mediante *OpenMP* e *intrínsecas*. Además, se ha utilizado una de las librerías que proporciona *Intel®* llamada *MKL* (*Intel® Math Kernel Library*). Esta librería implementa todas las rutinas de *BLAS* optimizadas. Se han utilizado para contrastar los tiempos que se obtienen con ellas frente a los de nuestra implementación.



Figura 1.12 Logotipo ICC

1.3.3 Herramientas

En este apartado se expondrán las herramientas más significativas utilizadas en el proyecto.

1.3.3.1 Microsoft Visual Studio 2012

Microsoft Visual Studio (Figura 1.13) es una herramienta *IDE* (*Integrated Development Environment*) propietaria de *Microsoft*. Para poder descargarlo y utilizarlo ha sido necesario solicitar una licencia de estudiante mediante la plataforma que ofrece la *UPV/EHU* llamada *On The Hub*. En este proyecto se ha aprovechado la posibilidad de **integración del Intel® C++ Compiler** en el *Visual Studio*. Esto nos ha permitido trabajar cómodamente con este compilador y sus configuraciones.



Figura 1.13 Logotipo MVS

A continuación se expondrá una breve **guía de cómo crear un proyecto** en *Visual Studio*. Cuando iniciemos el programa, nos aparecerá la pantalla que vemos en la Figura 1.14. Para crear un proyecto nuevo seleccionamos “Nuevo proyecto...” (1) (*también accesible a través del menú superior: “Archivo” → “Nuevo” → “Proyecto”*). A continuación seleccionaremos la plantilla más apropiada para nuestro proyecto, en este caso “Visual C++” (2) → “Aplicación de consola Win32” (3). Esto nos generará los archivos correspondientes para una aplicación en el lenguaje *C++*. Hemos escogido “Aplicación de consola Win32” para poder lanzar las ejecuciones como con el compilador *GCC* (a través del *CMD – intérprete de comandos –* de Windows). En esta ventana también podemos cambiar, si lo deseamos, el nombre del proyecto y la ruta de trabajo. Tras seleccionar esta configuración, pulsamos aceptar. En la siguiente ventana (Figura 1.15) podemos pulsar finalizar directamente (la configuración por defecto es suficiente).

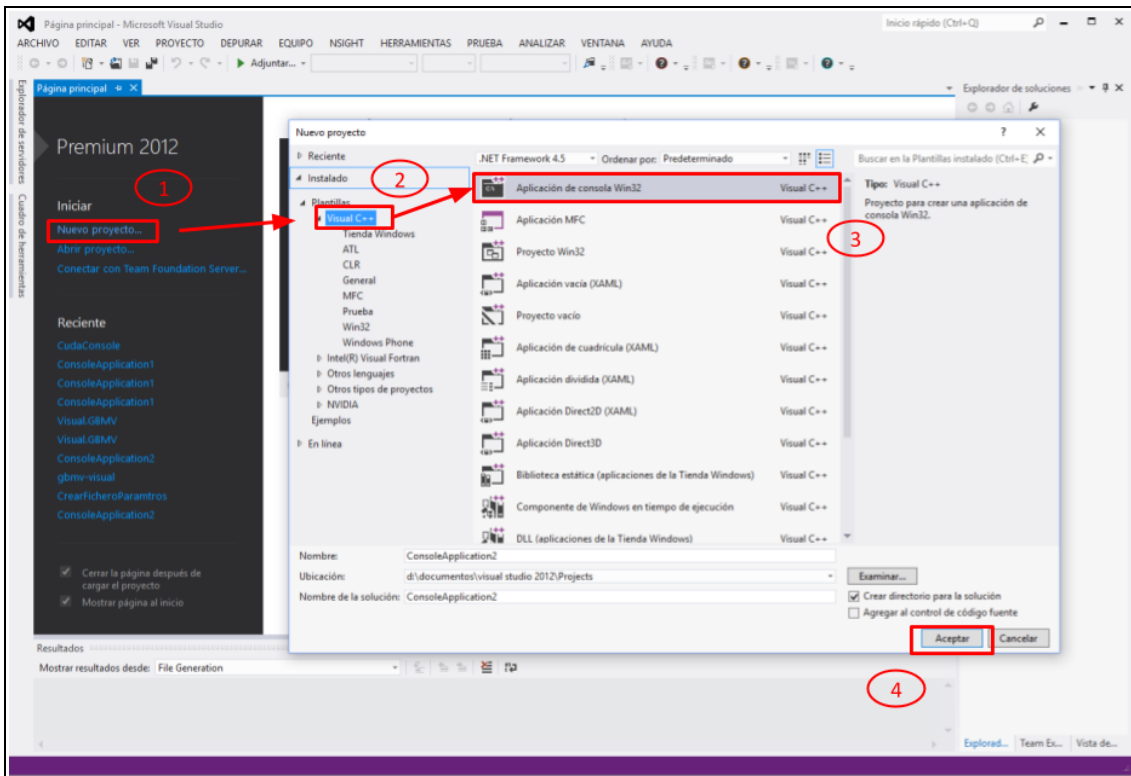


Figura 1.14 Guía de creación de proyecto Visual Studio - 1

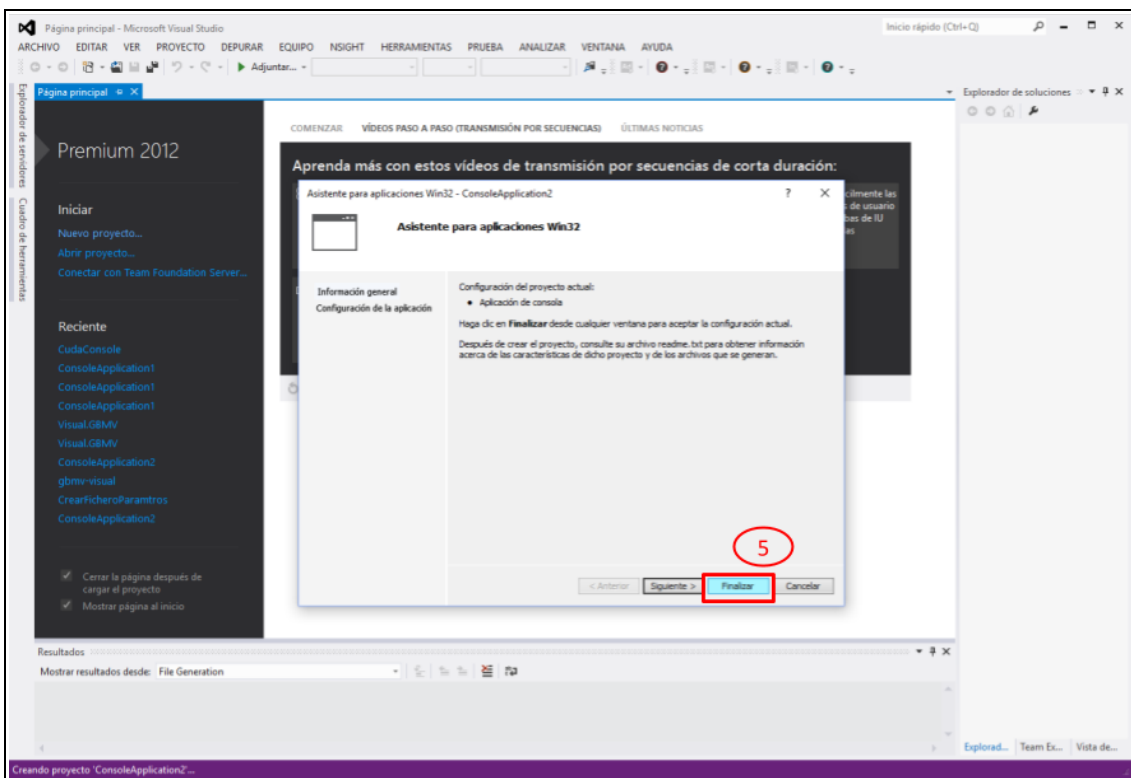


Figura 1.15 Guía de creación de proyecto Visual Studio - 2

Finalmente, para cambiar el compilador a utilizar (por defecto *Visual C++ Compiler*) deberemos acceder al menú superior “Proyecto” (6) → “Compilador” (7) → “Use Intel C++”. Además, deberemos tener en cuenta siempre en qué modo estamos trabajando (9), si en modo “Debug” o en modo “Release”, ya que los cambios en la configuración que hagamos en un modo no se verán reflejados en el otro (Figura 1.16).

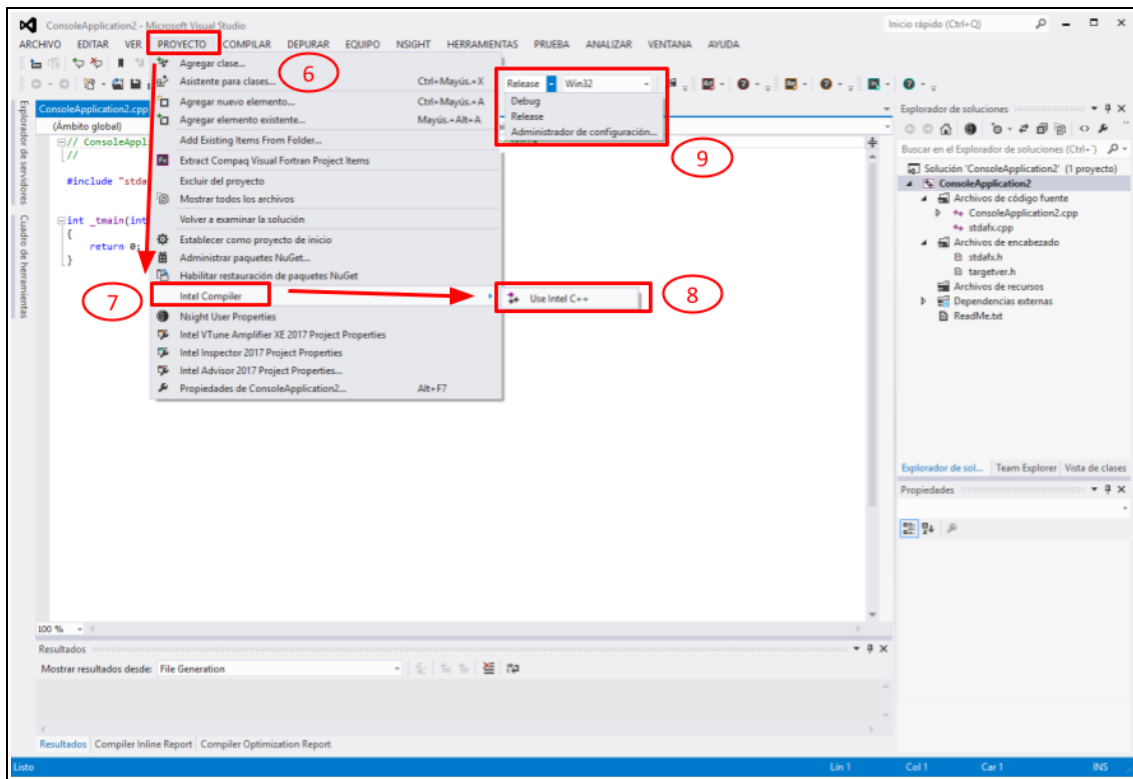


Figura 1.16 Guía de creación de proyecto Visual Studio - 3

En el *explorador de soluciones* (en este caso a la derecha de la Figura 1.13) tendremos todos los archivos de nuestro proyecto. En este caso el programa principal (`main.c` normalmente en GCC) se llama “`ConsoleApplication2.cpp`” (ya que no hemos cambiado el nombre por defecto). En el fichero ya estará definida la función `_tmain` con la nomenclatura adecuada para *Visual Studio*, además del fichero de cabecera “`#include stdafx.h`” que debe estar siempre al comienzo de los ficheros `.cpp` que haya en el proyecto (Programa 1.3).

```
// ConsoleApplication2.cpp: define el punto de entrada de la
// aplicación de consola.
//
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

Programa 1.2 Fichero por defecto `ConsoleApplication.cpp`

1.3.3.2 Intel® VTune Amplifier XE 2017

Intel® VTune Amplifier es una aplicación comercial para el **análisis de rendimiento de software** en máquinas de 32 y 64-bit basadas en la arquitectura x86. Tiene tanto *GUI* (interfaz gráfica de usuario) como línea de comandos y viene en versiones para sistemas operativos Linux o Microsoft Windows. Muchas de sus funcionalidades pueden utilizarse tanto en *hardware Intel* como en *AMD*, pero la obtención de información soportada en hardware avanzado requiere una *CPU* fabricada por *Intel®*.

En nuestro caso al instalar el *Intel® Parallel* sobre *Microsoft Visual Studio* también se ha instalado esta herramienta en el entorno de desarrollo. Podemos acceder a ella a través del menú superior de *Visual Studio* "Proyecto" → "Intel Vtune Amplifier XE 2017 Properties".

Se ha utilizado para hacer un análisis de los resultados obtenidos y contrastarlo con nuestras propias conclusiones. Podemos ver un ejemplo del uso de la herramienta en la *Figura 1.17*.

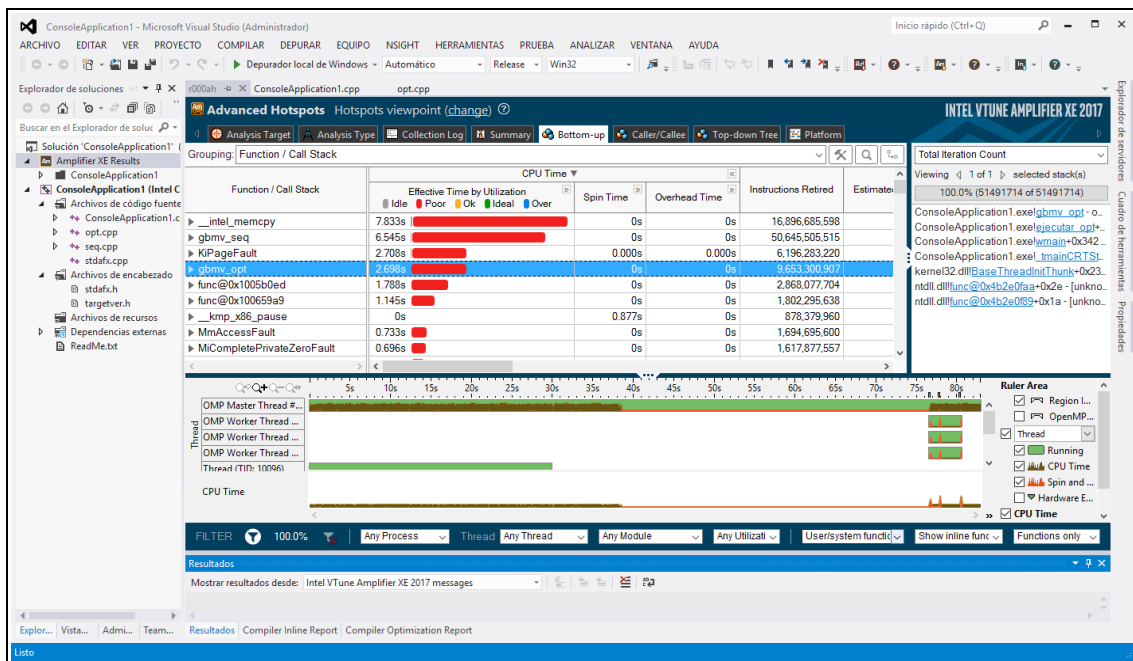


Figura 1.17 Ejemplo de uso de Intel® Vtune Amplifier

1.3.3.3 Intel® Advisor 2017

Intel® Advisor es un **asistente para la vectorización y la paralelización** de programas para los lenguajes de programación *C*, *C++*, *C#* y *Fortran* (ejemplo de uso en la *Figura 1.18*).

Al igual que el *Intel® VTune Amplifier*, esta herramienta se instala mediante el *Intel® Parallel Studio* junto con *Microsoft Visual Studio*.

Como en una parte del proyecto hemos dejado que el compilador *ICC* autovectorice y autoparalelice nuestro algoritmo, mediante esta herramienta hemos analizado la forma en la que lo ha hecho y la hemos contrastado con lo que hemos hecho explícitamente con *GCC*.

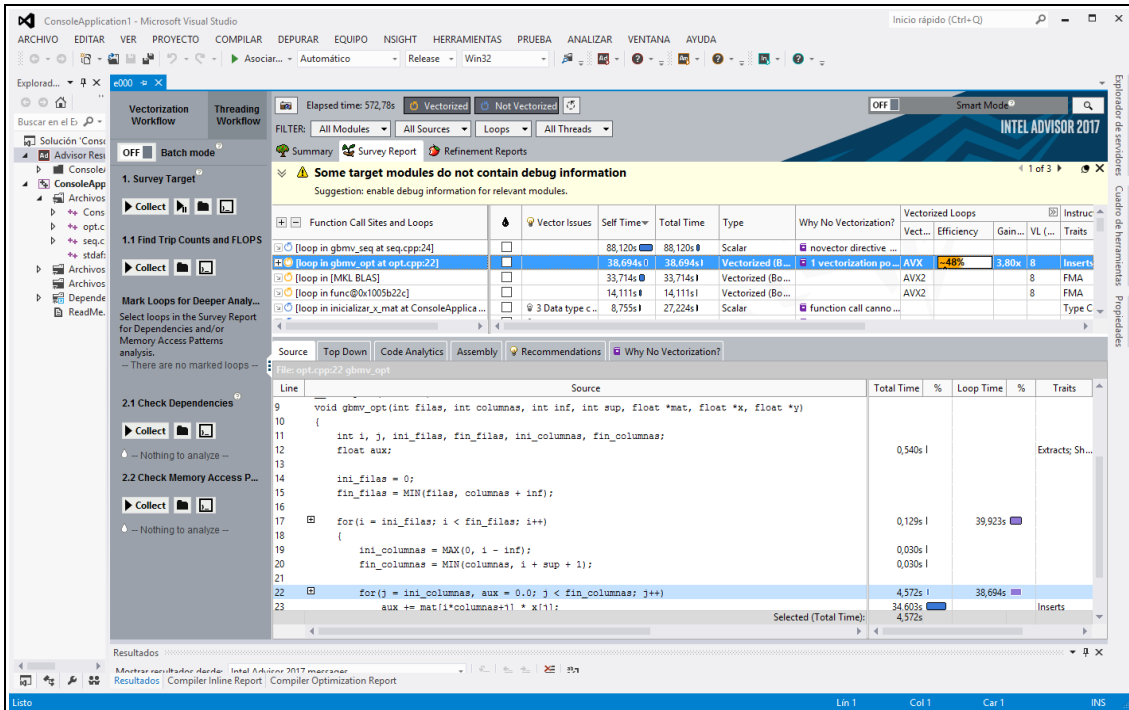


Figura 1.18 Ejemplo de uso de Intel® Advisor

1.3.3.4 Intel® Advisor – herramienta Roofline

La herramienta anteriormente explicada (Intel® Advisor), recientemente permite también crear un modelo *Roofline* del programa. Un gráfico *Roofline* es una representación visual del rendimiento de un programa en relación a las limitaciones del hardware, incluyendo el ancho de banda de la memoria y los picos computacionales. Este gráfico nos puede mostrar de un simple vistazo dónde están los cuellos de botella de nuestro programa, cuánto rendimiento podríamos obtener, qué cuellos de botella pueden ser solucionados y cuáles merece la pena solucionar, por qué aparecen estos cuellos de botella... Es una herramienta de diagnóstico útil.

Podemos ver un ejemplo de uso de esta herramienta en la Figura 1.19. En el eje de abscisas se representan *flops/Byte* (por cada Byte que traemos de memoria, cuántas operaciones podemos hacer), y en el eje de ordenadas *GFlops* (cuántos *GFlops* de rendimiento tiene nuestro programa). Además, las líneas diagonales nos indican cómo está acotado nuestro programa por la memoria de la que dispone nuestro sistema (*RAM*, *Caché L3*, *Caché L2...*) y las líneas horizontales nos indican las cotas del programa por las instrucciones de las que dispone nuestro procesador (*SSE*, *AVX*, *FMA...*).

También nos da información sobre el tiempo de ejecución de la función que tenemos seleccionada (en este caso *MxM_bloq*) 1, sobre el porcentaje de instrucciones de cada tipo (memoria, cómputo, *mixed/mezcla*) que tiene dicha función 2 y sobre los *GFlops* totales que

se han obtenido 3 . Se puede cambiar en cualquier momento la selección de la función haciendo 'clic' en otro de los círculos que se pueden ver en el gráfico (de esta forma podríamos analizar cada función por separado).

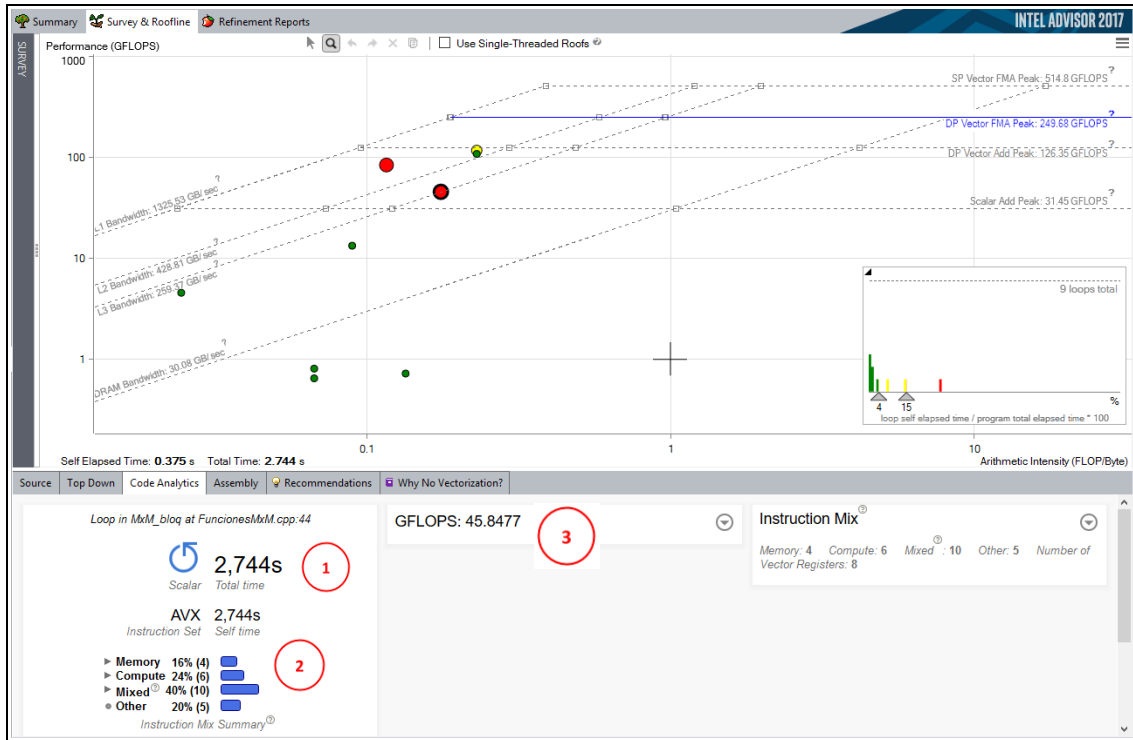


Figura 1.19 Intel® Advisor - Roofline

1.4 Equipos utilizados para realizar las pruebas

En este apartado se hará una breve presentación de los tres equipos diferentes utilizados para realizar las pruebas. Se distinguirán los tres por el procesador que incorporan (*Tabla 1.4*).

	Portátil – Intel® Core i3 3110M	Portátil – Intel® Core i7 4810MQ	Sobremesa – Intel® Core i7 6700K
Arquitectura	Ivy Bridge	Haswell	Skylake
Conjunto de instrucciones soportados	MMX, SSE, SSE2, SSE3, SSE4.1, SSE4.2, EM64T, VT-x, AVX	MMX, SSE, SSE2, SSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3	MMX, SSE, SSE2, SSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3, TSX
Número de núcleos	2	4	4
Número de threads	4 (<i>HypeThreading 2x</i>)	8 (<i>HypeThreading 2x</i>)	8 (<i>HypeThreading 2x</i>)
Frecuencia máxima de reloj	2.40 GHz	2.80 GHz	4.00 GHz
Tamaño caché L1 (Datos)	2 x 32 KB (64 KB)	4 x 32 KB (128 KB)	4 x 32 KB (128 KB)
Tamaño caché L1 (Instr.)	2 x 32 KB (64 KB)	4 x 32 KB (128 KB)	4 x 32 KB (128 KB)
Tamaño caché L2	2 x 256 KB (512 KB)	4 x 256 KB (1 MB)	4 x 256 KB (1 MB)
Tamaño caché L3	3 MB	6 MB	8 MB
GPU (modelo)	-	-	NVIDIA GTX 1080 Ti *

Tabla 1.4 Equipos utilizados para la realización de pruebas

* solamente se han realizado pruebas de CUDA con la GPU del sobremesa con procesador Intel® Core i7 6700K

Las pruebas se han realizado en cada una de estas tres máquinas (salvo las de *NVIDIA CUDA*). Se ha omitido la explicación de cada conjunto de pruebas en cada procesador por aligerar la lectura de este documento. Solamente se presentan los datos del ordenador de sobremesa (enmarcado en rojo en la *Tabla 1.4*).

2

Especificación BLAS 2

En este capítulo se explicará para cada tipo de compilador (*GNU Compiler Collection* e *Intel® C++ Compiler*), los algoritmos correspondientes a la especificación *BLAS de nivel 2* implementados y, en su caso, las optimizaciones realizadas presentando, al final de cada uno, los resultados obtenidos. Se concluirá el capítulo con un resumen de resultados general.

2.1 GNU Compiler Collection (GCC)

Empezando por el compilador *GNU Compiler Collection* se explicarán en primer lugar los algoritmos implementados en secuencial (resumidos finalmente en una función genérica a la que se le ha dado el nombre de *GBMV*). Se continuará el apartado explicando las optimizaciones realizadas a la función, en concreto su paralelización explícita mediante *OpenMP* y su vectorización explícita mediante las *intrínsecas de Intel®* y finalmente la combinación de ambos

métodos. Se terminará el apartado exponiendo los resultados de las pruebas realizadas para cada una de las optimizaciones.

En todos los algoritmos la matriz se trata por filas, ya que en el lenguaje de programación utilizado (C), los vectores se guardan por filas (es decir, elementos contiguos de los vectores fila están guardados de forma contigua en memoria). Esto permitirá que los accesos a los vectores se realice con *stride=1* (es decir, sin 'saltar' entre elementos no contiguos en memoria), lo que nos llevará a resultados más eficientes.

Los algoritmos especificados se explican con cierto nivel de detalle. Para programadores experimentados solo con ver el código sería suficiente, pudiéndose saltar las explicaciones.

2.1.1 Secuencial

En este subapartado se expondrán los algoritmos secuenciales implementados para cada una de las funciones de la especificación *BLAS de nivel 2*. En la *Figura 2.1* podemos ver la *sección* del esquema general que se va a tratar en este apartado.

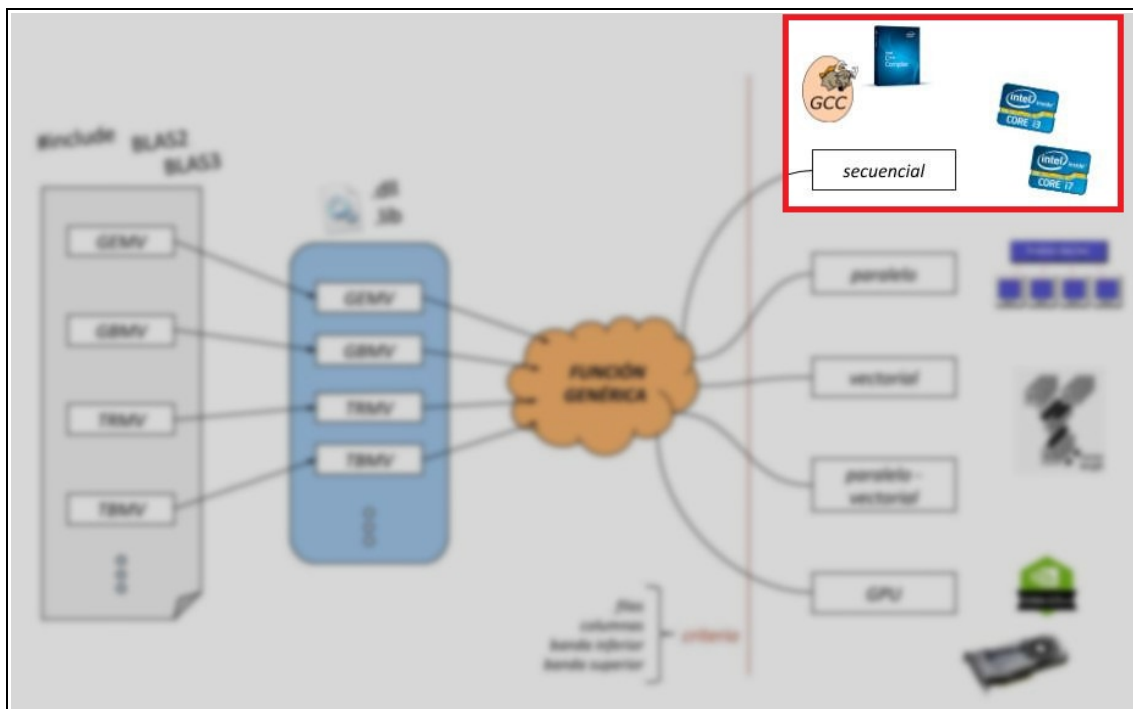


Figura 2.1 Esquema general – algoritmos secuenciales

2.1.1.1 GEMV – Matriz [densa] por vector

Según la especificación *BLAS*, **GEMV** debe atenerse a la siguiente **funcionalidad**:

$$y \leftarrow A x + \alpha y$$

El algoritmo secuencial diseñado para para ello es el que se muestra en el *Programa 2.1*.

```
void gemv_seq(int filas, int columnas, float *mat, float *x, float *y)
{
    int i,j;
    float aux;

    for (i = 0; i < filas; i++)
    {
        for (j = 0, aux = 0.0; j < columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
}
```

Programa 2.1 GEMV secuencial – Matriz [densa] por vector

También se han implementado los algoritmos secuenciales con **sintaxis C99, C11** como se muestra en el *Programa 2.2*. Para hacer un programa con esta sintaxis se deben pasar como primeros parámetros los tamaños de los vectores/matrices (en este caso *filas* y *columnas*). A continuación se declaran los vectores y/o matrices indicando su tamaño (con las variables anteriormente mencionadas).

Para recorrer un vector en su totalidad basta con escribir '`vector[:]`' (lo que equivale a '`vector[0:1:long.vector]`'), es decir, recorrer el vector desde el elemento 0 hasta su último elemento dando *saltos* de un elemento (o sea, de uno en uno, o *stride* = 1). Por lo que para multiplicar por pares los elementos de dos vectores (en este caso una fila de una matriz y un vector) y dejar el resultado en el vector '`y`' basta con escribir '`y[:] = mat[i][:] * x[:]`'. Finalmente en nuestro programa se incluye la reducción '`__sec_reduce_add`' para sumar todos los elementos resultantes del producto y dejar el resultado en la posición '`i`' del vector '`y`'. Esta operación (producto escalar entre dos vectores) se hace una vez por cada fila de la matriz para obtener la multiplicación del vector '`x`' por la matriz '`mat`' en el vector de resultados '`y`'.

```
void gemv_c99( int filas, int columnas, float mat[filas][columnas],
              float x[columnas], float y[filas] )
{
    int i;

    for (i = 0; i < filas; i++)
        y[i] = __sec_reduce_add(mat[i][:] * x[:]);
}
```

Programa 2.2 GEMV secuencial – sintaxis C99, C11

Esta sintaxis ayuda al compilador a *autovectorizar* y al programador a representar el algoritmo más algebraicamente (reduciendo bucles explícitos).

2.1.1.2 TRMV – Matriz [triangular] por vector

TRMV hace referencia a la multiplicación de una **matriz triangular** por un vector. Según la especificación *BLAS2* debe atenerse al siguiente funcionamiento:

$$x \leftarrow A x$$

Para esta función *BLAS* impone una restricción: que las matrices triangulares (inferiores o superiores indiferentemente) sean **cuadradas**. Sin embargo en este proyecto **se ha generalizado la operación para cualquier tipo de matriz triangular**. También se ha generalizado la operación que efectúa:

$$y \leftarrow A x + y$$

Para aumentar la eficiencia, se decidió descomponer *TRMV* en dos subprogramas: uno para matrices triangulares superiores y otro para matrices triangulares inferiores. Los motivos concretos se explicarán en cada subprograma.

TRMV [lower o inferior]

Los esquemas, en general, de las matrices triangulares inferiores tratadas son los que se pueden ver en la *Figura 2.2*. Cabe destacar que la parte sombreada es la parte no nula de cada matriz.

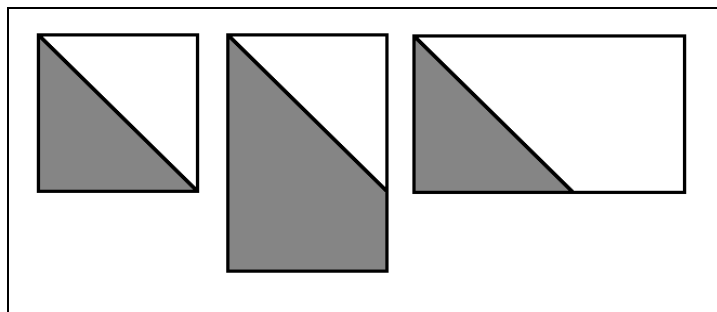


Figura 2.2 Esquemas de matrices triangulares inferiores

En la *Figura 2.3* podemos ver los esquemas concretos del cálculo que realiza *TRMV* para cada tipo de matriz triangular inferior. Se puede apreciar que siempre se trata el vector 'y' en su totalidad. La diferencia la marca *TRMV [superior]* como se verá en su correspondiente sección.

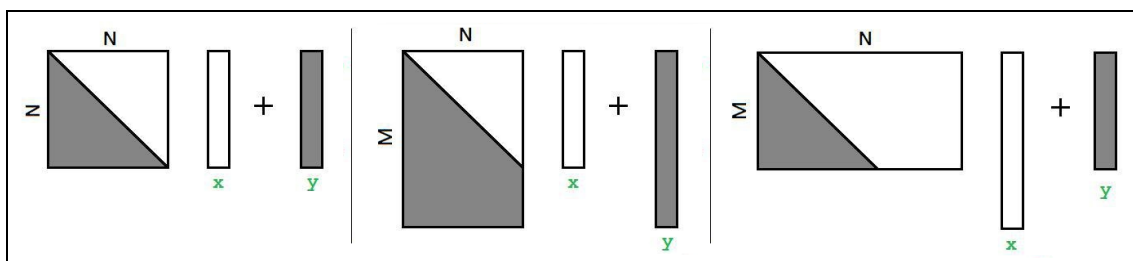


Figura 2.3 Esquema de operaciones con las matrices triangulares inferiores

El algoritmo secuencial diseñado para las matrices triangulares inferiores es el que se muestra en el *Programa 2.3*. En el código de este programa (y de los programas posteriores de este apartado) se han incluido comentarios donde se aclaran los límites de los accesos a la matriz.

```

void l_trmv_seq( int filas, int columnas, float *mat, float *x,
                float *y)
{
    int i,j;
    float aux;

    for (i = 0; i < filas; i++)
    {
        // El índice 'j' debe estar acotado. No debe sobrepasar ni la
        // diagonal principal (i+1) ni el número de columnas
        for (j = 0, aux = 0.0; j < MIN(i+1,columnas); j++)
            aux += mat[i*columnas+j] * x[j];
        y[i] += aux;
    }
}

```

Programa 2.3 L_TRMV secuencial – Matriz triangular inferior por vector

TRMV [upper o superior]

Los esquemas de las matrices triangulares superiores tratadas son los que se pueden ver en la *Figura 2.4*.

En la *Figura 2.5* podemos ver los esquemas concretos del cálculo que realiza TRMV para cada tipo de matriz triangular superior. Se puede apreciar que, a diferencia del caso anterior, no siempre se trata el vector 'y' en su totalidad por lo que podremos conseguir algo más de eficiencia en dichos casos. Estos casos son, en concreto, cuando la matriz es 'rectangular vertical'. Se tratan como máximo 'N' elementos de 'y'.

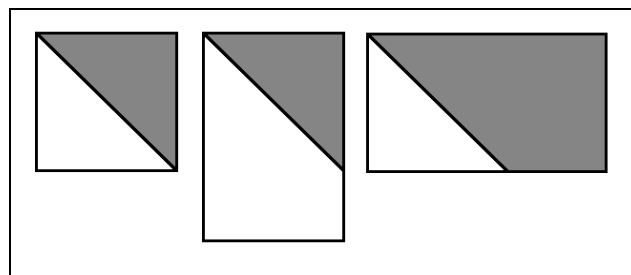


Figura 2.4 Esquema de matrices triangulares superiores

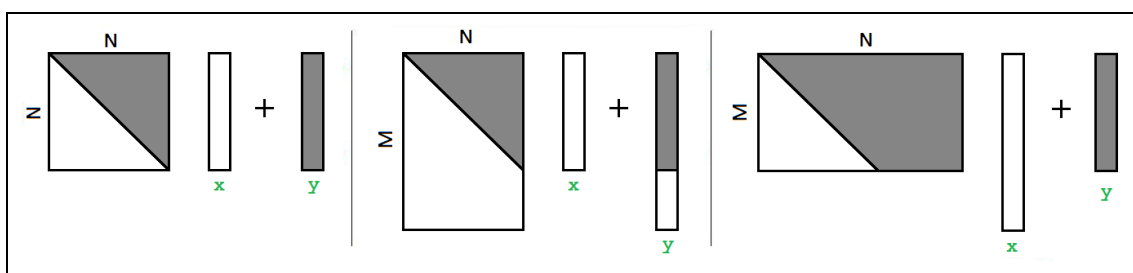


Figura 2.5 Esquema de operaciones con las matrices triangulares superiores

El algoritmo secuencial diseñado para las matrices triangulares superiores es el que se muestra en el *Programa 2.4*.

```
void u_trmv_seq(int filas, int columnas, float *mat, float *x, float
*y)
{
    int i,j,filas_aux;
    float aux;

    // El índice de filas (i) debe estar acotado. No debe ser nunca
    // mayor al número de columnas de la matriz. (calculado a priori)
    filas_aux=(filas > columnas)?columnas:filas;

    for (i = 0; i < filas_aux; i++)
    {
        for (j = i, aux = 0.0; j < columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
}
```

Programa 2.4 U_TRMV secuencial – Matriz triangular superior por vector

2.1.1.3 TBMV – Matriz [triangular en banda] por vector

TBMV hace referencia a la multiplicación de una **matriz triangular en banda** por un vector. Según la especificación *BLAS2* debe atenerse al siguiente funcionamiento:

$$x \leftarrow A x$$

Para esta función *BLAS* impone la misma restricción que para *TRMV*: que las matrices triangulares en banda (inferiores o superiores indiferentemente) sean **cuadradas**. En este caso **nos hemos atenido a la especificación** y hemos mantenido la restricción, aunque la operación sí se ha generalizado:

$$y \leftarrow A x + y$$

Para seguir con el mismo esquema que con *TRMV*, también se ha descompuesto *TBMV* en dos subprogramas: uno para matrices triangulares (en banda) superiores y otro para matrices triangulares (en banda) inferiores.

***TBMV* [lower o inferior]**

El esquema general de las matrices triangulares inferiores *en banda* tratadas es el que se puede ver en la *Figura 2.6*. En la *Figura 2.7* mostramos el mismo esquema resaltando la restricción que impone *BLAS*.

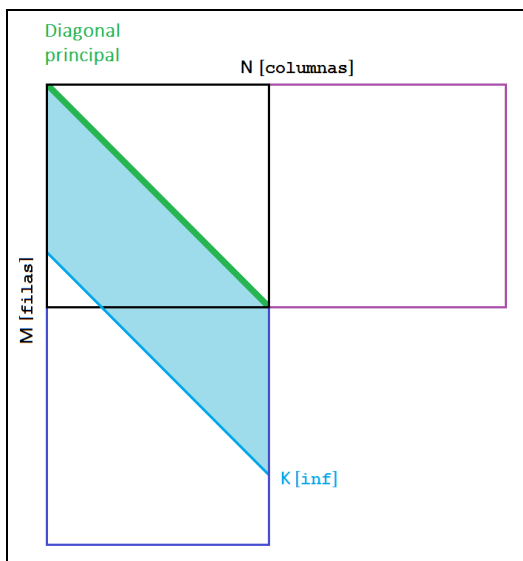


Figura 2.6 Esquema matriz triangular en banda inferior (general)

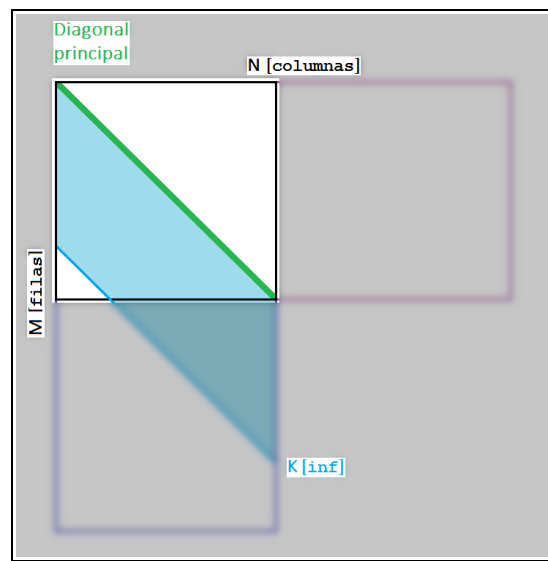


Figura 2.7 Esquema de matriz triangular en banda inferior con la restricción de BLAS

El algoritmo secuencial diseñado para las matrices triangulares inferiores *en banda* es el que se muestra en el Programa 2.5.

```

void l_tbmvsq ( int filas, int columnas, int inf, float *mat,
               float *x, float *y )
{
    int i, j, ini_columnas, fin_columnas;
    float aux;

    for(i = 0; i < filas; i++)
    {
        // El índice 'j' (recorrido en columnas) debe estar acotado.
        // Siempre comienza en 'i-inf' (partiendo de la diagonal
        // principal [i,i], 'inf' columnas a la izquierda) sin indexar
        // índices negativos. El recorrido termina en 'i+1' (diagonal
        // principal).

        ini_columnas = MAX(0, i - inf);
        fin_columnas = i+1;

        for(j = ini_columnas, aux = 0.0; j < fin_columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
}

```

Programa 2.5 L_TBMV secuencial – Matriz triangular inferior (en banda) por vector

TBMV [upper o superior]

El esquema general de las matrices triangulares superiores *en banda* tratadas es el que se puede ver en la *Figura 2.8* . En la *Figura 2.9* mostramos el mismo esquema resaltando la restricción que impone BLAS.

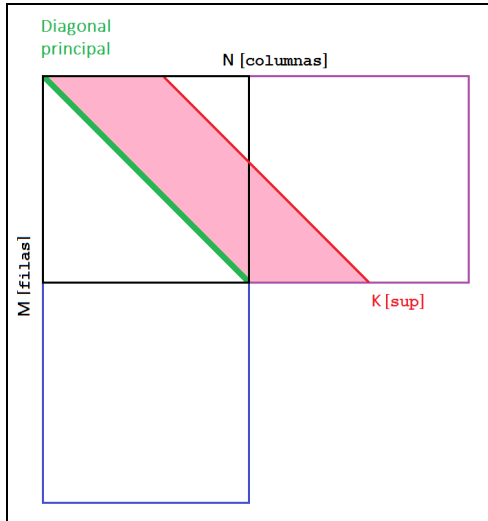


Figura 2.8 Esquema matriz triangular en banda superior (general)

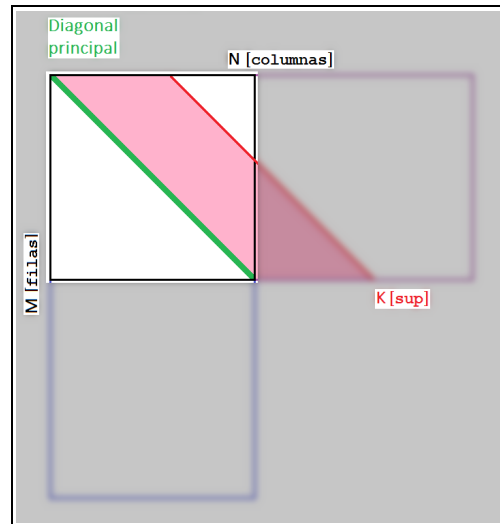


Figura 2.9 Esquema de matriz triangular en banda superior con la restricción de BLAS

El algoritmo secuencial diseñado para las matrices triangulares superiores *en banda* es el que se muestra en el *Programa 2.5*.

```
void u_tbmvs_seq ( int filas, int columnas, int sup, float *mat,
                  float *x, float *y )
{
    int i, j, ini_columnas, fin_columnas;
    float aux;

    for(i = 0; i < filas; i++)
    {
        // El índice 'j' (recorrido en columnas) debe estar acotado.
        // Siempre comienza en 'i' (diagonal principal).
        // Siempre llega hasta en 'i+sup+1' (partiendo de la diagonal
        // principal [i,i], 'sup' columnas a la derecha) sin indexar
        // un número mayor a 'columnas' (para no salirnos de rango).

        ini_columnas = i;
        fin_columnas = MIN(columnas, i + sup + 1);

        for(j = ini_columnas, aux = 0.0; j < fin_columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
}
```

Programa 2.6 U_TBMV secuencial – Matriz triangular superior (en banda) por vector

2.1.1.4 GBMV – Matriz [general en banda] por vector

Este apartado es el correspondiente a nuestra **función genérica** (Figura 2.10).

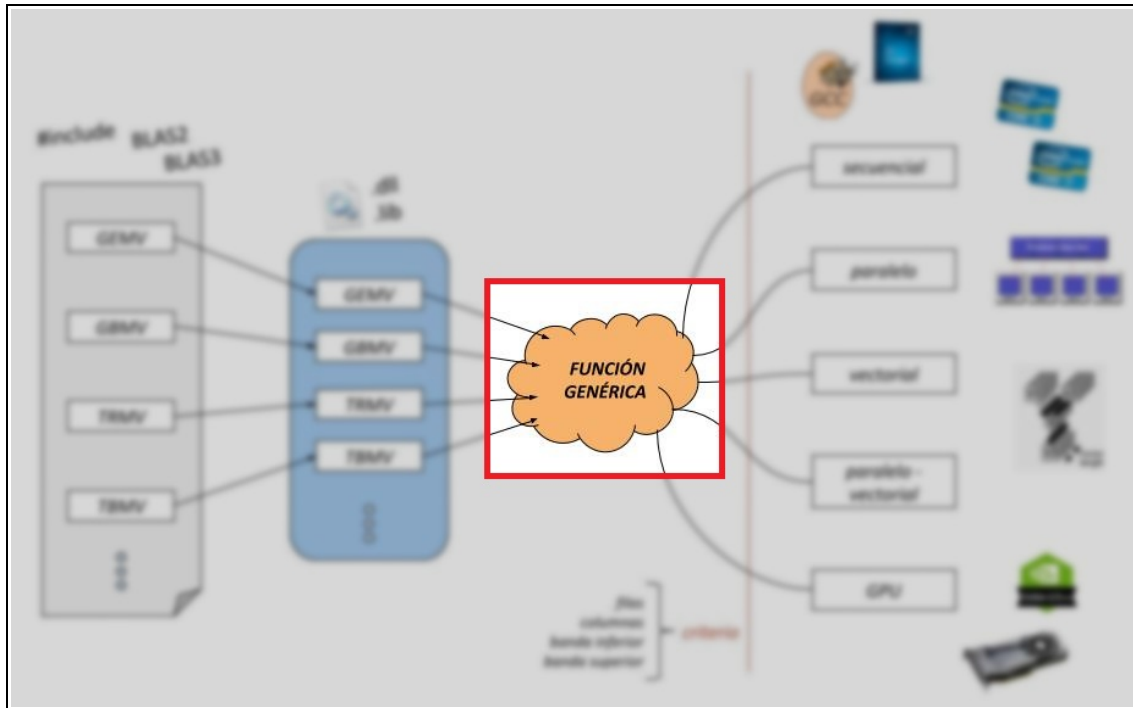


Figura 2.10 Esquema general – función genérica

Según la especificación BLAS, **GBMV** debe efectuar lo siguiente:

$$y \leftarrow A x + y$$

Para el desarrollo de este proyecto se decidió implementar esta función de forma que pudiese ejecutar **todas las casuísticas** de tipos de matrices posibles (según los parámetros de: número de filas, número de columnas, banda inferior y banda superior) y que así sirviera como función genérica de todo lo visto hasta el momento.

Para poder discernir todos los casos particulares nuestra función genérica recibe los parámetros nombrados anteriormente, al igual que la función *GBMV* según la especificación (entre corchetes el nombre que reciben en nuestra implementación):

- M [filas] - Número de filas de la matriz
- N [columnas] - Número de columnas de la matriz
- KL [inf] - Número de diagonales inferiores de la matriz
- KU [sup] - Número de diagonales superiores de la matriz

El **esquema general** de todas las matrices que tiene en cuenta la función es el que podemos ver en la *Figura 2.11*. La zona sombreada en azul corresponde al número de diagonales inferiores (KL [inf]). Así mismo, la zona sombreada en rojo corresponde al número de diagonales superiores (KU [sup]). El esquema pretende representar también las diferentes opciones de tamaños que puede tener la matriz. Por lo que, por un lado, la matriz podría ser cuadrada (solo el cuadrado negro), por otro lado podría ser *rectangular vertical* (teniendo en cuenta el rectángulo azul oscuro) y por último, podría ser *rectangular horizontal* (teniendo en cuenta el rectángulo morado).

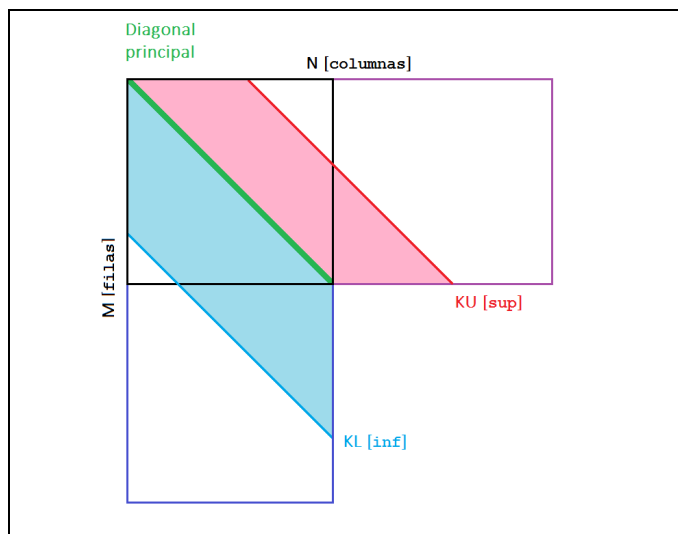


Figura 2.11 Esquema de matriz en banda general

Combinando diferentes valores de 'inf' y 'sup' podemos obtener todos los casos posibles. Todos estos casos están reflejados en la *Tabla 2.1*.

KL [inf]	KU[sup]	Tipo de matriz
filas	0	Triangular inferior
$0 < inf < filas$	0	Triangular inferior en banda
0	columnas	Triangular superior
0	$0 < sup < columnas$	Triangular superior en banda
0	0	Diagonal
filas	columnas	Densa
$0 < inf \leq filas$	$0 < sup < columnas$	Banda general
$0 < inf < filas$	$0 < sup \leq columnas$	Banda general

Tabla 2.1 Casuísticas de matrices según KL[inf] y KU[sup]

En el *Programa 2.7* podemos ver el código correspondiente al algoritmo secuencial de la función genérica.

Explicar el recorrido de la matriz en este caso:

- Respecto al índice 'i', viene limitado por las variables `ini_filas` y `fin_filas`, calculadas a priori.

El índice 'i' siempre comienza desde la fila 0 (desde el inicio) y este es el valor que se le asigna a `ini_filas`.

En las matrices cuadradas o las matrices *rectangulares horizontales* siempre se deben recorrer todas las filas. En cambio, en las matrices *rectangulares verticales* puede darse el caso de que no sea necesario recorrer todas las filas. En tal caso, el número de filas que es necesario recorrer es el número de columnas más el número de diagonales inferiores (`columnas + inf`). Por tanto, a `fin_filas` se le asigna el mínimo entre ambos valores.

- Respecto al índice 'j', viene limitado por las variables `ini_columnas` y `fin_columnas`. Estas dos variables no pueden ser calculadas a priori, ya que dependen del índice 'i'.

El índice 'j' en un principio comienza siempre desde '`i - inf`'. Esto es, partiendo de la diagonal principal `[i, i]`, '`inf`' columnas hacia la izquierda (dado que `inf` indica el número de diagonales inferiores que tiene la matriz). Sin embargo, si aplicásemos siempre este criterio podríamos salirnos del rango de la matriz (indexando un número de columna negativo). Por ello, se calcula `ini_columnas` como el máximo entre '0' e '`i - inf`'.

El final del índice 'j' en un principio siempre es '`i + sup + 1`'. Esto es, partiendo de la diagonal principal `[i, i]`, '`sup`' columnas hacia la derecha (dado que `sup` indica el número de diagonales superiores que tiene la matriz). Se le debe sumar 1 puesto que en C los índices comienzan desde 0. Sin embargo, siguiendo este criterio podríamos salirnos del rango de la matriz (indexando un número mayor al número de columnas). Por ello, se calcula `fin_columnas` como el mínimo entre '`i + sup + 1`' y '`columnas`'.

A partir de aquí todas las optimizaciones se realizarán sobre esta función, que denominamos como *Función Genérica*.

```

void gbmv_seq (int filas, int columnas, int inf, int sup, float *mat,
              float *x, float *y)
{
    int i, j, ini_filas, fin_filas, ini_columnas, fin_columnas;
    float aux;

    ini_filas = 0;
    fin_filas = MIN(filas, columnas + inf);

    for(i = ini_filas; i < fin_filas; i++)
    {
        ini_columnas = MAX(0, i - inf);
        fin_columnas = MIN(columnas, i + sup + 1);

        for(j = ini_columnas, aux = 0.0; j < fin_columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
}

```

Programa 2.7 GBMV secuencial – Matriz en banda general por vector (algoritmo genérico)

2.1.2 Paralelismo explícito (con *OpenMP*)

La primera optimización realizada ha sido la paralelización de la función (Figura 2.12).

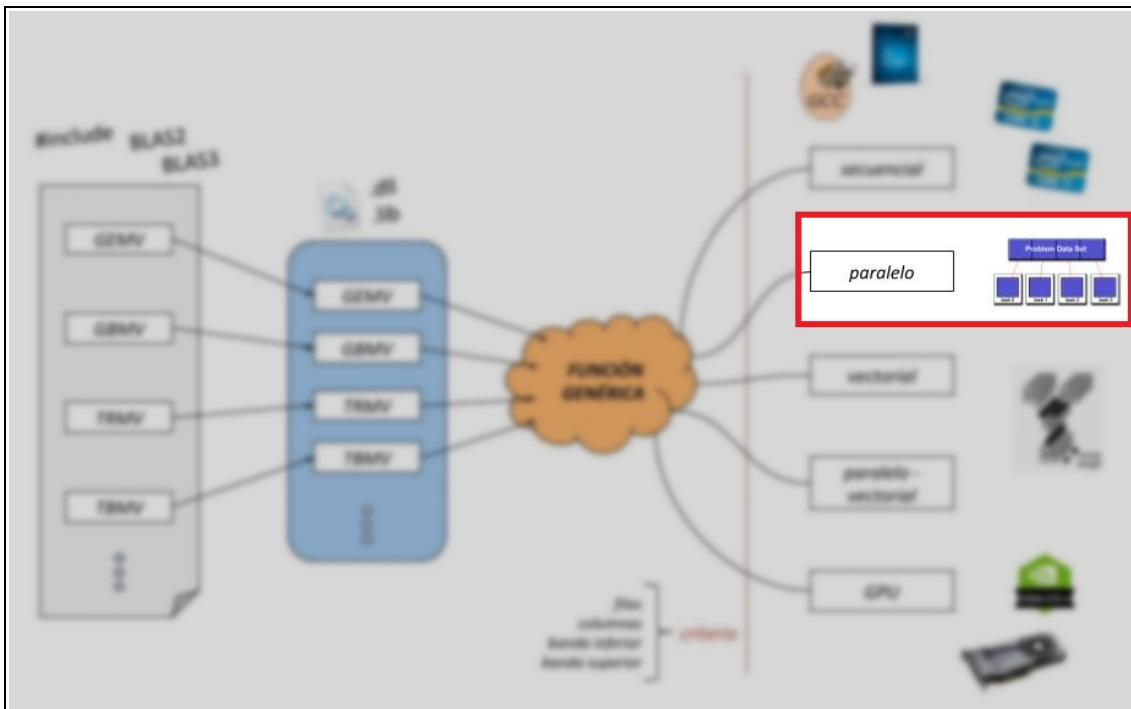


Figura 2.12 Esquema general - paralelismo

Como se ha nombrado con anterioridad, siempre se ha optado por un reparto de carga entre cuatro *threads* (luego $NTHR = 4$). Aparte, se ha tenido que decidir cómo se realiza el reparto de carga entre estos hilos. El tamaño del trozo que debe ejecutar cada uno recibe el nombre de **chunk**. En este caso *chunk* hace referencia al número de filas de la matriz que cada *thread* procesará. Se ha establecido *chunk* como el mínimo entre diez y $FILAS/NTHR$. El número mínimo de filas con el que se ha hecho pruebas es de 32. En ese caso cada *thread* de los cuatro procesará diez filas, salvo el último (*thread 3*), el cual solo procesará dos filas (Figura 2.13).

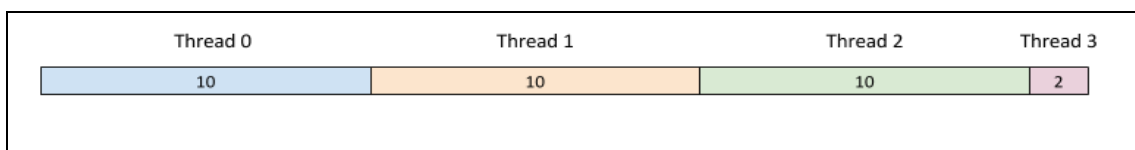


Figura 2.13 Reparto estático ($chunk = 10$) de 32 elementos

Para el caso general se ha definido *chunk* de forma que a cada *thread* le toquen ' $FILAS/NTHR$ ' filas contiguas de la matriz. Es decir, siempre se divide la matriz en cuatro submatrices rectangulares y cada *thread* procesa su propia submatriz (Figura 2.14). Esto se ha decidido así puesto que un tamaño de *chunk* menor incluiría sobrecarga innecesaria en el algoritmo (cada *thread* no tendría todos sus elementos contiguos en memoria).

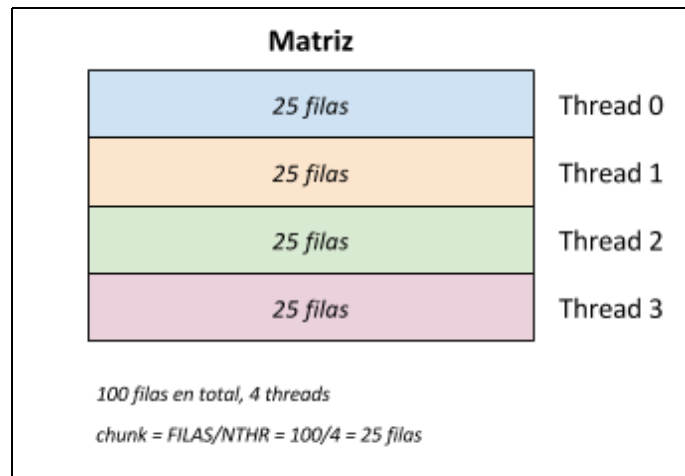


Figura 2.14 Ejemplo de reparto estático de 100 filas entre 4 threads

También se han realizado pruebas con reparto de carga dinámico. Los resultados han sido peores y se ha acabado por descartar esta opción.

En el *Programa 2.8* podemos ver cómo se establece en tiempo de ejecución el tipo de reparto de carga a utilizar (mediante la función de *OpenMP* `omp_set_schedule`) con su correspondiente *chunk*. Podemos alternar entre reparto estático y dinámico simplemente cambiando una variable definida (*#define*). Con la función `omp_set_num_threads` le indicamos el número de threads a utilizar (en este caso $NTHR=4$).

```

chunk = MIN(10, FILAS/NTHR);

omp_set_num_threads(NTHR);

#ifdef SCHED_TYPE_STATIC
    omp_set_schedule(omp_sched_static, chunk);
#elif defined SCHED_TYPE_DYNAMIC
    omp_set_schedule(omp_sched_dynamic, chunk);
#endif

```

Programa 2.8 Configuración del reparto de carga (schedule)

Tras esta decisión, lo único que se ha hecho para paralelizar la función es introducir la cláusula `#pragma omp parallel for` en el bucle exterior del algoritmo (para repartir solo las filas). A la cláusula se le indican las variables privadas con inicialización previa (`firstprivate`), las variables privadas sin inicialización previa (`private`) y el tipo de reparto (`schedule runtime`). El reparto `runtime` utiliza los datos establecidos con la función `omp_set_schedule` anteriormente nombrada (*Programa 2.9*).

```

void gbmv_omp( int filas, int columnas, int inf, int sup, float *mat,
              float *x, float *y)
{
    int i,j, ini_filas, fin_filas, ini_columnas, fin_columnas;
    float aux;

    ini_filas = 0;
    fin_filas = MIN(filas, columnas + inf);

    #pragma omp parallel for
    firstprivate (filas, columnas, inf, sup, ini_filas, fin_filas)
    private(i, j, aux, ini_columnas, fin_columnas)
    schedule(runtime)
    for(i = ini_filas; i < fin_filas; i++)
    {
        ini_columnas = MAX(0, i - inf);
        fin_columnas = MIN(columnas, i + sup + 1);

        for(j = ini_columnas, aux = 0.0; j < fin_columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
}

```

Programa 2.9 Función genérica paralelizada

2.1.3 Vectorización explícita (con *intrínsecas*)

La siguiente optimización realizada ha sido la vectorización de la función (Figura 2.15).

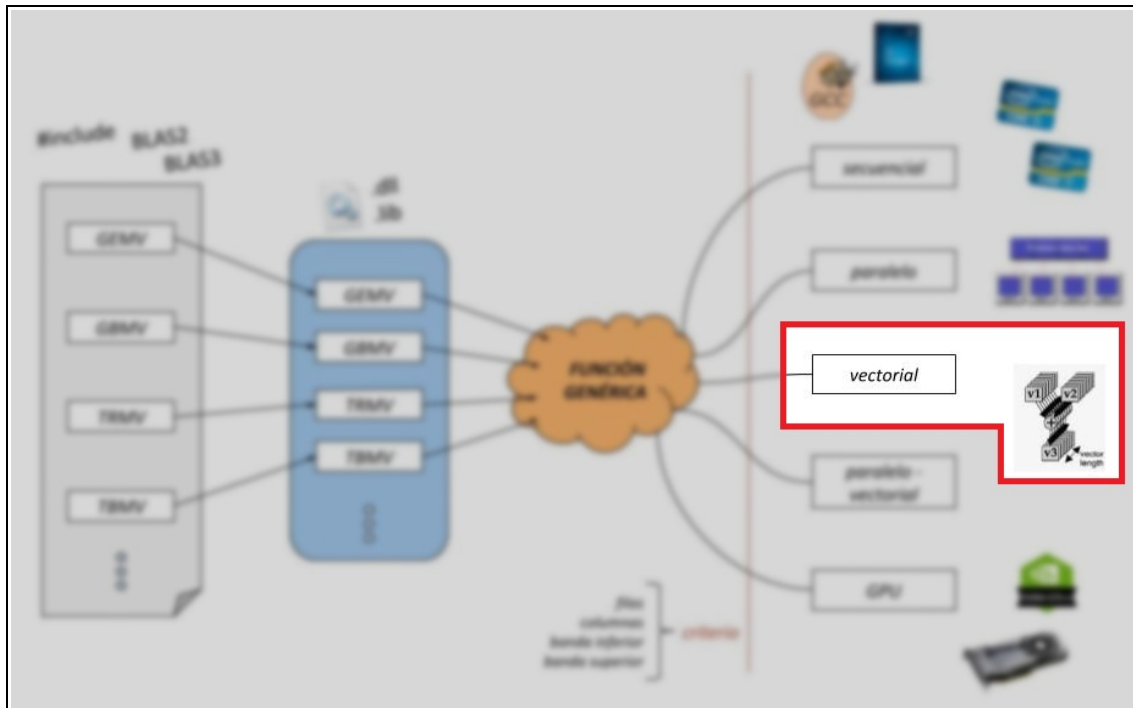


Figura 2.15 Esquema general - vectorización

Se ha implementado la vectorización tanto con *SSE* como con *AVX* (recordemos que *SSE* trabaja con vectores de 128 bits y *AVX* trabaja con vectores de 256 bits). En este apartado se explicará la vectorización con *SSE*, en el apartado siguiente, con *AVX* (ambos algoritmos, vectorial y paralelo-vectorial, se han vectorizado con ambas tecnologías).

Como la vectorización, como su nombre indica, utiliza vectores y sin perder de vista la operación que estamos realizando ($y \leftarrow A x + y$), nuestro objetivo será tratar la matriz por vectores filas y vectorizar cada una de las operaciones “fila de $A * x + y$ ”. La rutina principal que lo realiza se llama `gbmv_sse()`.

Una restricción impuesta a este algoritmo es que si los elementos no nulos de cada fila de la matriz no superan los 32 (es decir, si el número de diagonales inferiores sumado al número de diagonales superiores no supera los 32), todo el proceso se realiza en secuencial (Programa 2.10).

```
if (inf + sup < 32)
{
    gbmv_seq(filas, columnas, inf, sup, mat, x, y);
    return;
}
```

Programa 2.10 Restricción impuesta a la vectorización

Si esto no se cumple, se continua de forma normal con el algoritmo y se calculan las cotas del recorrido en filas de la matriz (*ini_filas* y *fin_filas*) al igual que en el algoritmo secuencial. Las cotas del recorrido en columnas (*ini_columnas* y *fin_columnas*) tampoco varían. Se calcula para cada fila el número total de columnas recorrido (*num_columnas*). Si el número de columnas a procesar de esa fila no es mayor que 32, esa fila se calcula en secuencial y se sigue con la siguiente iteración (*Programa 2.11*).

```

for (i = ini_filas; i < fin_filas; i++)
{
    ini_columnas = MAX(0, i - inf);
    fin_columnas = MIN(columnas, i + sup + 1);

    num_columnas = fin_columnas - ini_columnas;

    if (num_columnas < 32)
    {
        gbmv_seq_unaFila( ini_columnas, fin_columnas,
                        &mat[i*columnas], &x[0], &y[i]);

        continue;
    }

    ... (A)
}

```

Programa 2.11 Restricción impuesta a la vectorización para cada fila

Una parte de las intrínsecas de *Intel*[®] que no se ha explicado en el *apartado 1.3* de la memoria, es la necesidad de convertir los vectores escalares a *estructuras vectoriales*. El procedimiento que se sigue es el siguiente (especificado para la precisión simple, el caso que nos incumbe):

1. Declarar un vector del tipo `__m128` (es el tipo de dato correspondiente de *SSE* para vectores de precisión simple). Este vector podrá almacenar cuatro elementos de un vector habitual (un *float* ocupa 32 bits, si el vector es de 128 bits, puede almacenar cuatro elementos de tipo *float*).
2. Cargar cuatro elementos del vector de *float* correspondiente al vector `__m128` con una instrucción de carga (*load*). Hay dos tipos:
 - a) `var_tipo_m128 = _mm_load_ps(&x[i])`
Carga en `var_tipo_m128` cuatro elementos del vector '*x*' (*i*, *i*+1, *i*+2, *i*+3), **sabiendo que '*x*' está alineado** en memoria.
 - b) `var_tipo_m128 = _mm_loadu_ps(&x[i])`
Carga en `var_tipo_m128` cuatro elementos del vector *x* (*i*, *i*+1, *i*+2, *i*+3), **si '*x*' no está alineado** en memoria o no tenemos la certeza de ello.

¿Qué significa que 'x' esté alineado en memoria? La memoria de los procesadores Intel® hoy día está dividida en bloques de 64 Bytes (en memoria caché). Este tamaño de bloque permite el uso alineado de SSE, AVX y AVX512 (en AVX512 un vector trabajaría con el bloque de memoria completo; 512 bits = 64 Bytes). Un vector está alineado en memoria cuando el comienzo de alguna parte de dicho vector coincide con el comienzo de un bloque de memoria (Figura 2.16). Tal y como se ve en la figura, no es necesario que el vector 'x' esté alineado en su totalidad para poder tratarlo con operaciones del tipo (a). Se podría tratar la primera parte no alineada en secuencial (*prólogo*), la segunda parte de forma vectorial alineada (*cuerpo*), y finalmente los elementos “sobrantes” en secuencial (*epílogo*). Nosotros en la declaración de los vectores o, en su caso, de las matrices, utilizamos el alineamiento de 64 Bytes.

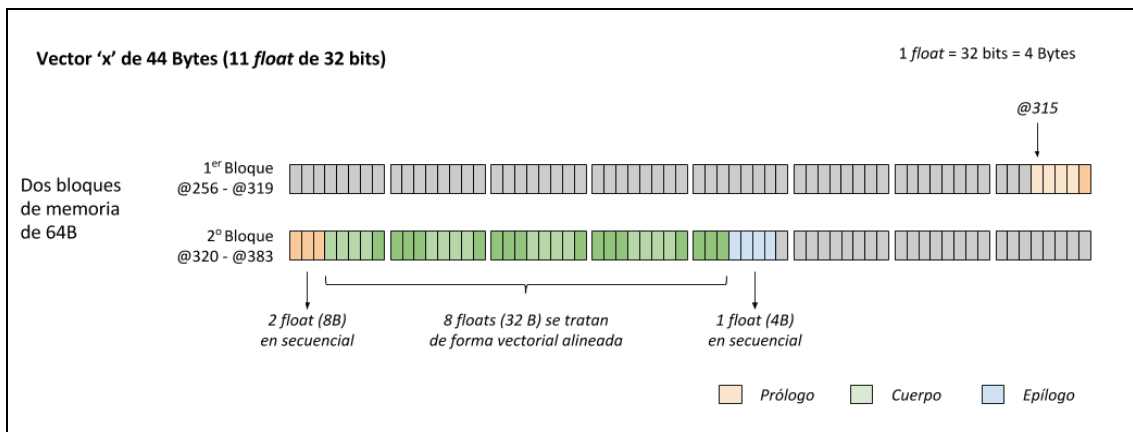


Figura 2.16 Ejemplo de alineamiento de un vector

Las operaciones tipo (a) (con vectores alineados) son mucho más eficientes que las operaciones tipo (b). Por ello, antes de proceder a la vectorización del cuerpo del bucle se ha comprobado si el vector 'x' y la fila de la matriz correspondiente están alineados o no (y por cuánto).

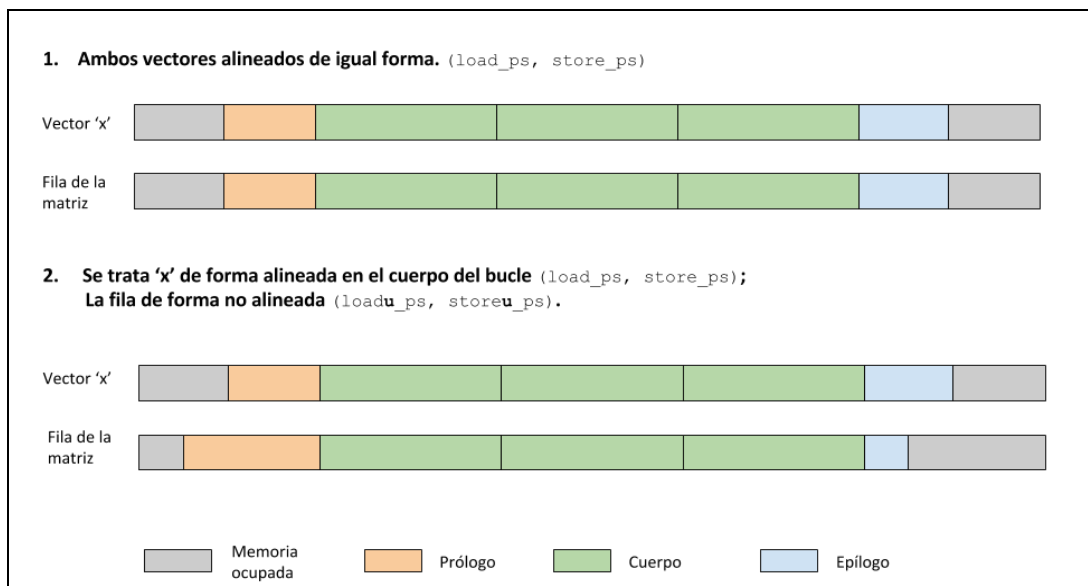


Figura 2.17 Casos de alineamiento entre 'x' y la fila de la matriz

En caso de que ambos estén alineados de la misma forma, es decir, tengan el mismo número de elementos en el prólogo, en el cuerpo y en el epílogo (Figura 2.17 – 1.), se llama a `gbmv_sse_alin`, en caso contrario (Figura 2.17 – 2.) se llama a `gbmv_sse_noalin`. En la primera, ambos vectores se tratan de forma alineada mientras que en la segunda se trabaja solo con 'x' de forma alineada.

Ahora bien, ¿cómo podemos determinar cuántos elementos del vector constituyen el prólogo, el cuerpo y el epílogo? Esto es lo que sigue al Programa 2.11 en la parte A. En primer lugar, explicar la nomenclatura del programa para designar las variables mediante la Figura 2.18.

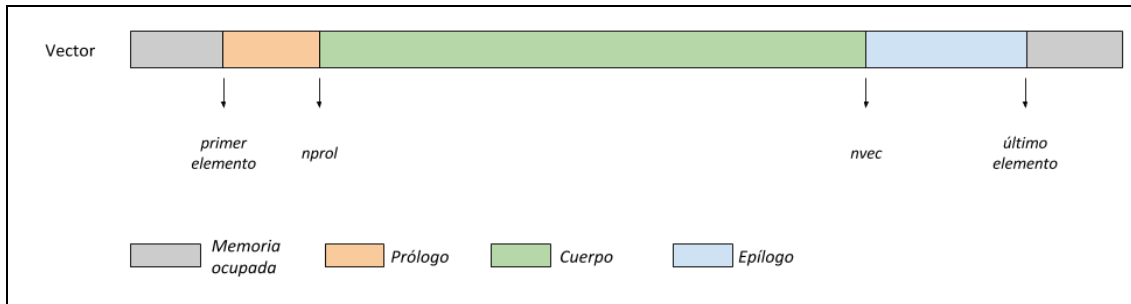


Figura 2.18 Nomenclatura de las variables de vectorización

Se procederá ahora a explicar la parte A del programa (Programa 2.12) utilizando como ejemplo los datos de la Figura 2.16. Destacar que para la vectorización solamente se tratan los elementos no nulos del vector y de la fila de la matriz (es decir, los elementos desde `ini_columnas` hasta `fin_columnas`).

En primer lugar se obtiene la dirección de comienzo del primer elemento no-nulo del vector 'x' - `&x[ini_columnas]` (tener en cuenta que el direccionamiento de la memoria es al Byte). Pongamos el supuesto de que la dirección almacenada en la variable `dir1` sea `@315` y que el número de elementos no-nulos del vector sea 15. Para saber cuántos elementos tiene el prólogo (`npról`) se efectúa el siguiente cálculo:

$$1. (dir1 \gg 2) = @315 / 4 = 78$$

Al dividir por cuatro con una operación de desplazamiento se eliminan los decimales.

$$2. (dir1 / 4) \% 16 = 78 \% 16 = 14$$

14 es el número de elementos que tiene este bloque de memoria antes de los elementos del vector 'x' (en un supuesto de que todos los elementos fuesen float de 4B).

$$3. 16 - (dir1 / 4) \% 16 = 16 - 14 = 2$$

2 es el número de elementos que tiene el prólogo.

Por tanto, en nuestro ejemplo la variable `npról` será igual a dos (al igual que se muestra en la figura de ejemplo).

A continuación, para saber cuántos elementos hay hasta el final de la parte que se ejecutará en vectorial de forma alineada (`nvec`) se siguen estos pasos:

1. `(double) (num_columnas - nprol_x) / 4 = (11 - 2) / 4 = 2.25`

Se divide el número de elementos no-nulos del vector entre cuatro, el resultado puede o no dar un número entero, por lo que se hace la conversión en tipo `double`.

2. `(int) (floor (2.25)) = 2`

Como en este caso el resultado da un número fraccionario, se redondea hacia abajo (`floor`) y se vuelve a convertir en entero.

3. `2 << 2 = 8`

Tras aplicar la operación `floor` y desplazar (multiplicar por cuatro) nos quedamos con un número de elementos múltiplo de cuatro (a ejecutar en vectorial de forma alineada). Esto es importante y era el objetivo perseguido ya que los vectores en SSE tienen capacidad para cuatro `float`.

4. `8 + nprol = 8 + 2 = 10`

Finalmente se vuelve a sumar `nprol` al resultado obtenido para que `nvec` sea el índice límite ('hasta donde') de la ejecución en vectorial.

Por tanto, en nuestro ejemplo la variable `nvec` será igual a 10 (también se puede comprobar en la figura de ejemplo; *Figura 2.16*). Finalmente, si `nprol` vale 16 (un bloque completo) equivale a que `nprol` valga 0 (el comienzo del vector coincidiría en este caso con el comienzo de un bloque).

Este mismo procedimiento se efectúa también para la fila de la matriz. Si el número de elementos del prólogo de la matriz coincide con el número de elementos del prólogo del vector '`x`', ambos vectores están alineados de igual forma (*Figura 2.17 – 1.*). Este caso sería el *ideal* y se llamaría a la subrutina `gbmv_sse_alin`. En caso contrario (*Figura 2.17 – 2.*) se llamaría a la subrutina `gbmv_sse_noalin`. Estas subrutinas lo que hacen es el producto escalar entre '`x`' y la fila de la matriz (utilizando las intrínsecas). La estructuración de las subrutinas `gbmv_sse_alin` y `gbmv_sse_noalin` es similar. Ambas se dividen en prólogo (secuencial, desde 0 hasta `nprol`), cuerpo (desde `nprol` hasta `nvec`) y epílogo (desde `nvec` hasta el final). Dada su similitud solo se explicará una de ellas (la versión no alineada, `gbmv_sse_noalin`).

Esta subdivisión de los vectores en *prólogo*, *cuerpo* y *epílogo* se denomina ***loop peeling*** en inglés. Este término aparecerá con el uso de las herramientas de Intel® como por ejemplo el Intel® Advisor.

Una vez identificada esta subdivisión procedemos a explicar la vectorización explícita con intrínsecas.

```

for (i = ini_filas; i < fin_filas; i++)
{
    ... // (Programa 2.11)

    dir1 = (int) (&x[ini_columnas]);

    nprol_x = 16 - ((dir1>>2)%16);
    nvec_x = (((int) floor((double)((num_columnas - nprol_x)/4.0)))<<2) + nprol_x;
    if(nprol_x == 16) nprol_x = 0;

    dir2 = (int) (&mat[i*columnas + ini_columnas]);

    nprol_mat = 16 - ((dir2>>2)%16);
    if(nprol_mat == 16) nprol_mat = 0;

    if(nprol_x == nprol_mat)
        y[i] += gbmv_float_sse_alin( num_columnas, &mat[i*columnas + ini_columnas],
                                   &x[ini_columnas], nprol_x, nvec_x);
    else
        y[i] += gbmv_float_sse_noalin( num_columnas, &mat[i*columnas + ini_columnas],
                                       &x[ini_columnas], nprol_x, nvec_x);
}

```

Programa 2.12 Alineación de los vectores



```

float gbmv_sse_noalin(int columnas, float *fila_mat, float *x, int nprol_x, int nvec_x)
{
    float __attribute__((aligned(64))) suma[4];
    float resultado = 0;
    int j;

    __m128 v_sum = _mm_setzero_ps();
    __m128 v_fila_mat, v_x, v_aux;

    // Prólogo
    for (j = 0; j < nprol_x; j++)
        resultado += fila_mat[j] * x[j];

    // Cuerpo
    for (j = nprol_x; j < nvec_x; j += 4)
    {
        v_fila_mat = _mm_loadu_ps(&fila_mat[j]);
        v_x = _mm_load_ps(&x[j]);

        v_aux = _mm_mul_ps(v_fila_mat, v_x);

        v_sum = _mm_add_ps(v_aux, v_sum);
    }

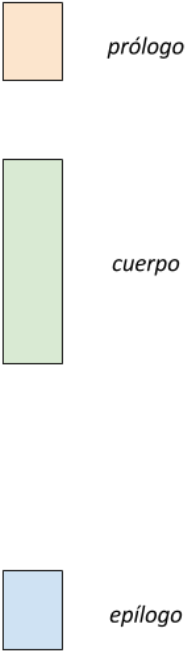
    // Acumular resultado
    v_sum = _mm_hadd_ps(v_sum, v_sum);
    v_sum = _mm_hadd_ps(v_sum, v_sum);
    _mm_store_ps(&suma[0], v_sum);

    resultado += suma[0];

    // Epílogo
    for (j = nvec_x; j < columnas; j++)
        resultado += fila_mat[j] * x[j];

    return resultado;
}

```



Programa 2.13 Subrutina vectorial SSE – producto escalar entre 'x' y una fila de la matriz

En el *Programa 2.13* vemos la programación vectorial explícita mediante intrínsecas. Su funcionalidad es la de calcular el producto escalar entre el vector 'x' y la fila de la matriz, retornando el valor que se deberá sumar a la posición del vector 'y' correspondiente (*Programa 2.12*).

En primer lugar se declaran cuatro variables del tipo `__m128` (tipo de dato que equivale a un vector de cuatro elementos en SSE). Uno de ellos, `v_sum`, se inicializa con ceros mediante la función `_mm_setzero_ps()`.

A continuación se efectúa el prólogo en secuencial (los `nprol` primeros elementos no alineados calculados con anterioridad).

Para el cuerpo, se desarrolla un bucle con índice 'j' (denotado así, ya que equivale al índice de columnas) que va desde `nprol` hasta `nvec` de cuatro en cuatro (ya que procesaremos de forma vectorial cuatro elementos simultáneamente). En el bucle se cargan los vectores SSE `v_fila_mat` y `v_x` mediante las instrucciones `_mm_loadu_ps` y `_mm_load_ps` respectivamente (la instrucción de carga de la fila de la matriz es no alineada, mientras que la del vector 'x' es alineada, por lo que se ha explicado con anterioridad). Tras haber cargado los cuatro valores correspondientes de la fila de la matriz en `v_fila_mat(j, j+1, j+2, j+3)` y los cuatro valores correspondientes de 'x' en `v_x(j, j+1, j+2, j+3)`, se multiplican ambos vectores con la función `_mm_mul_ps()` y se deja su resultado en `v_aux`. Finalmente se suma el resultado obtenido en el vector `v_sum` mediante la función `_mm_add_ps()` (*Figura 2.19*).

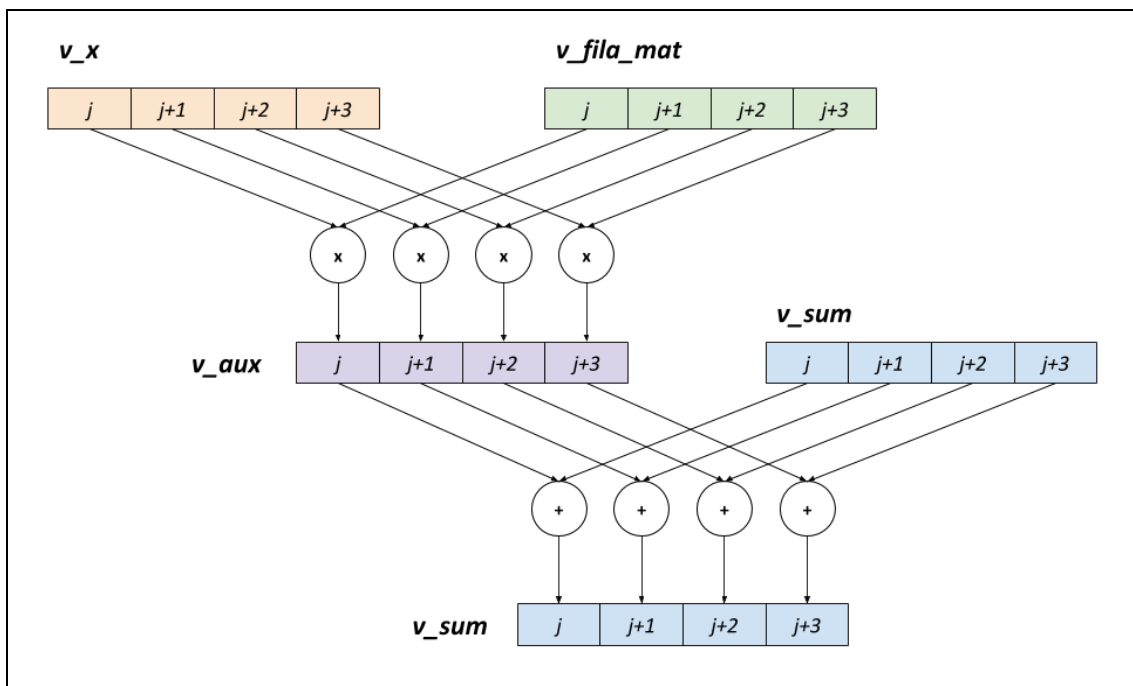


Figura 2.19 Cuerpo del algoritmo vectorial

Tras la ejecución del bucle tendremos en `v_sum` las sumas parciales del producto escalar (exceptuando las sumas que faltan por calcular en el epílogo). Para calcular la suma total de la

ejecución vectorial y dejarla en una variable escalar (*suma*) se ha utilizado la función `_mm_hadd_ps()`. Tras obtener el resultado de la suma total como se muestra en la *Figura 2.20* (en cada posición de *v_sum* obtenemos la suma total), se ha guardado el vector *v_sum* en el vector (escalar) *suma* mediante la función `_mm_store_ps()`. Se ha podido utilizar una función de guardado alineado debido a que la variable *suma* ha sido declarada de forma alineada (`__attribute__((aligned(64))) suma[4];`). Finalmente se suma a la variable *resultado* (donde se acumula la suma total tanto del prólogo, como del cuerpo, como del epílogo) la suma total del cuerpo, es decir, `suma[0]`.

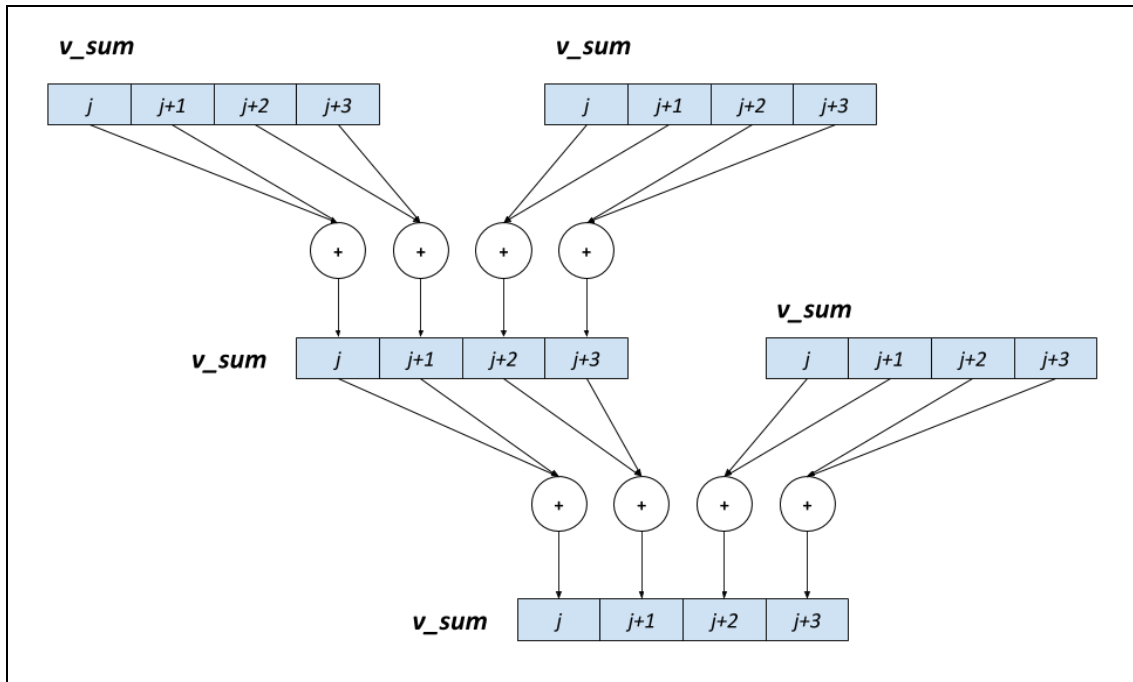


Figura 2.20 Suma de todos los elementos de un vector con `_mm_hadd_ps()` - SSE

Para finalizar, se efectúa el epílogo (desde *nvec* hasta el final, *num_columnas*) y se retorna la variable *resultado* que ahora ya contiene el producto escalar de 'x' por la fila de la matriz en su totalidad. Este resultado se sumará a la posición correspondiente del vector 'y' para cumplir con la funcionalidad deseada ($y \leftarrow A x + y$).

La única diferencia entre esta subrutina (`gbmv_sse_noalin()`; *Programa 2.13*) y la subrutina `gbmv_sse_alin()` es que en la segunda las operaciones de carga (`_mm_loadu_ps`) y guardado (`_mm_storeu_ps`) también se realizan alineadas (`_mm_load_ps`; `_mm_storeu_ps`; como las operaciones de carga y guardado que podemos ver sobre 'x').

Todo este procedimiento se efectúa para el rango establecido de filas de la matriz (`ini_filas - fin_filas`).

2.1.4 Paralelización y vectorización

La intención ahora es combinar los dos métodos anteriormente expuestos (*apartados 2.1.2 y 2.1.3* de la memoria; *Figura 2.21*).

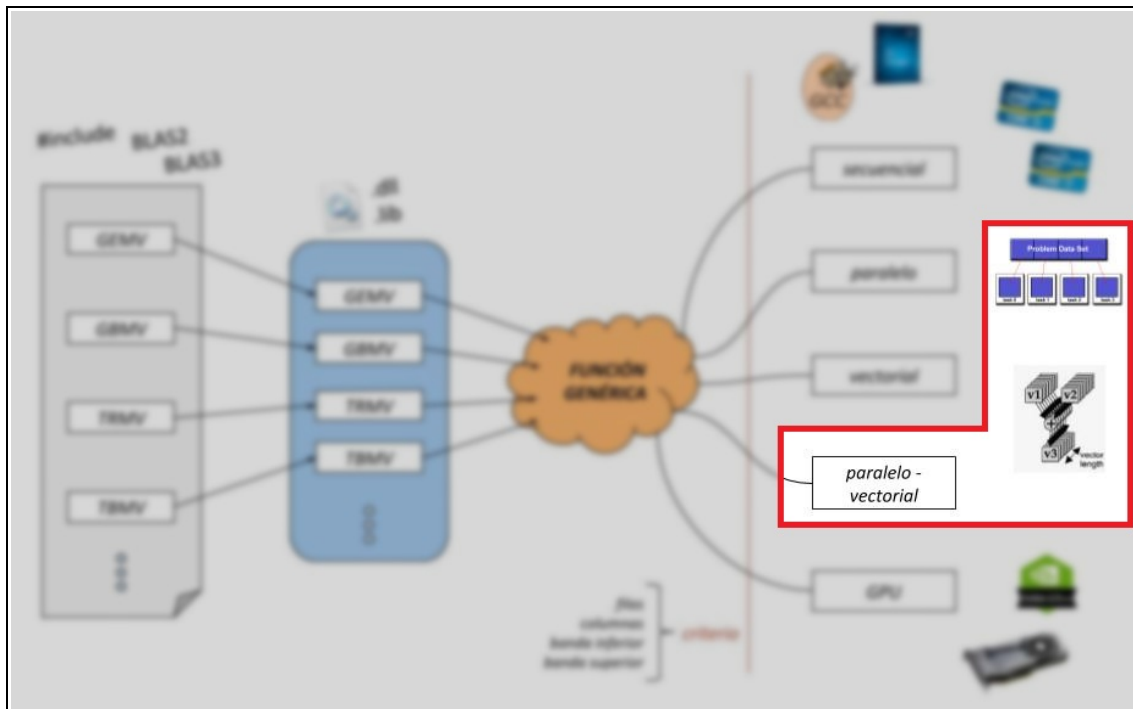


Figura 2.21 Esquema general – paralelización y vectorización

Como el proceso principalmente es el mismo, se hará hincapié sobre todo en las diferencias respecto a lo que se ha explicado hasta el momento, intentando no repetir de nuevo todas las aclaraciones. En este apartado se expondrá la vectorización con *AVX*, y se nombrarán las diferencias más significativas con respecto a la vectorización con *SSE*.

El objetivo del algoritmo paralelo-vectorial es el siguiente: por un lado, la división de la matriz en cuatro submatrices rectangulares, una para cada uno de los cuatro *threads* (en este caso el reparto de filas solo se realiza con las filas no nulas de la matriz, es decir, desde 0 hasta *fin_filas*). Por otro lado cada *thread* utilizará la vectorización para calcular los productos escalares entre 'x' y cada fila de la submatriz que les corresponde. La rutina principal para el algoritmo es la que se muestra en el *Programa 2.14*. Una restricción total impuesta, al igual que en el algoritmo de vectorización, es que si ninguna fila tiene más de 32 elementos no nulos, se ejecuta todo en secuencial.

```

static void gbmv_omp_avx( int filas, int columnas, int inf, int sup,
                        float *mat, float *x, float *y)
{
    int thread_num, fin_filas, trozo, ini_local, fin_local;

    fin_filas = MIN(filas, columnas + inf);

    trozo = ceil((float) (fin_filas)/(float)NTHR);

    if (inf + sup < 32)
    {
        gbmv_float_seq(filas, columnas, inf, sup, &mat[0], x, y);
        return;
    }

    #pragma omp parallel private(thread_num, ini_local, fin_local)
    firstprivate(filas, columnas, trozo, fin_filas)
    {
        thread_num = omp_get_thread_num();
        ini_local = trozo * thread_num;
        fin_local = MIN(trozo*(thread_num + 1), fin_filas);
        gbmv_avx_parallel( fin_local - ini_local, columnas, inf, sup,
                          &mat[0], &x[0], &y[0], ini_local);
    }
}

```

Programa 2.14 Rutina principal paralelización – vectorización con AVX

En primer lugar se calcula el rango de filas no nulas (al igual que se hacía en el algoritmo secuencial). La última fila no nula se almacena en la variable `fin_filas`. Cada `trozo` por tanto será de tamaño `fin_filas / NTHR` (en total cuatro trozos). Tras decidir esto se procede al reparto estático (manual) de las filas entre los `threads`. Se inicia una zona paralela mediante la cláusula de *OpenMP* `#pragma omp parallel`. A la cláusula se le indican las variables privadas con inicialización previa (`firstprivate`) y las variables privadas sin inicialización previa (`private`). A continuación cada `thread` almacenará en la variable `thread_num` su identificador (0, 1, 2 ó 3 correspondientemente). El inicio de filas de cada `thread` (`ini_local`) será su identificador por el tamaño del trozo. El fin de filas de cada `thread` (`fin_local`) será su identificador más uno por el tamaño del trozo. El mínimo entre '`thread_num + 1`' y '`fin_filas`' se ha establecido para el caso del último `thread`, para que no sobrepase el límite de filas no nulas. Podemos ver un ejemplo en la *Tabla 2.2* acompañado de su representación visual en la *Figura 2.22*.

Filas	100	thread_num	ini_local	fin_local	filas_local = fin_local – ini_local
Columnas	50	0	0	19	19
inf	25	1	19	38	19
sup	20	2	38	57	19
fin_filas	75	3	57	75	18
trozo	19				

Tabla 2.2 Ejemplo de reparto de filas para el algoritmo paralelo-vectorial

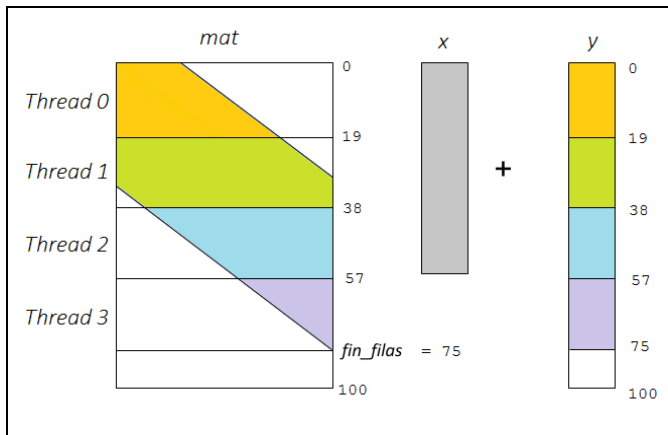


Figura 2.22 Ejemplo de reparto de filas para el algoritmo paralelo-vectorial

Tras ajustar todos los parámetros necesarios para el reparto, cada *thread* llama a la rutina `gbmv_avx_parallel()` con sus propios parámetros, indicando el número de filas que procesa cada uno (`fin_local - ini_local`) y el inicio de su trozo (`ini_local`).

La rutina estructuralmente no difiere mucho de la antes explicada `gbmv_sse()` o, en su caso, de su equivalente `gbmv_avx()` (Programa 2.15). La única diferencia significativa es que ahora cada *thread* no procesa desde `ini_filas` hasta `fin_filas` como era habitual hasta el momento, sino que `ini_filas` se obtiene por parámetro en la función, y `fin_filas` se calcula como `ini_filas` más `filas_local` (el número de filas que debe ejecutar ese *thread*; equivale a `fin_local - ini_local` de la función anterior). Otra sutil diferencia respecto a `gbmv_sse()` es el proceso de alineamiento de los vectores. Como AVX trabaja con vectores de 256 bits (a diferencia de SSE que utiliza vectores de 128 bits) la manera de proceder es algo diferente.

Para el alineamiento:

1. `dir = (int) (&x[ini_columnas]);`

La obtención de la dirección de memoria de 'x' (de la fila correspondiente al primer elemento no nulo del vector) es similar.

2. `nprol = 16 - ((dir1>>2)%16);`

La forma de obtener el número de elementos del prólogo tampoco varía. Esto se debe a que seguimos trabajando con el mismo tipo de dato (*float* – 4B), y que buscamos la posición del primer elemento del vector dentro de un bloque de memoria. Si ahora trabajásemos con, por ejemplo, un tipo de dato *double*, cada bloque de memoria (de 64B) tendría 8 elementos (de 8B) en lugar de 16 elementos (de 4B) en cuyo caso la operación sería: `[8 - ((dir1>>3)%8)]`.

3. `nvec_x = ((int) floor((double) ((num_columnas - nprol_x) / 8.0))) << 3 + nprol_x;`

Este paso varía respecto a lo explicado en el apartado 2.1.3. Como los vectores ahora son de ocho elementos de longitud, es necesario dividir por ocho, redondear (*floor*), y multiplicar por ocho (*shift*), en lugar de por cuatro.


```

void gbmv_avx_parallel( int filas_local, int columnas, int inf,
                       int sup, float *mat, float *x, float *y,
                       int ini_filas)
{
    int num_columnas, fin_filas, ini_columnas, fin_columnas;
    int dir1, dir2, nprol_x, nprol_mat, nvec_x;
    int i, j;

    fin_filas = ini_filas + filas_local;

    for (i = ini_filas; i < fin_filas; i++)
    {
        ini_columnas = MAX(0, i - inf);
        fin_columnas = MIN(columnas, i + sup + 1);

        num_columnas = fin_columnas - ini_columnas;

        if (num_columnas < 32)
        {
            gbmv_seq_unaFila( ini_columnas, fin_columnas,
                             &mat[i*columnas], &x[0], &y[i]);
            continue;
        }

        dir1 = (int) (&x[ini_columnas]);

        nprol_x = 16 - ((dir1>>2)%16);
        nvec_x = (((int)floor(((double)((num_columnas-nprol_x)/8.0)))<<3)
                 + nprol_x);
        if(nprol_x == 16) nprol_x = 0;

        dir2 = (int) (&mat[i*columnas + ini_columnas]);
        nprol_mat = 16 - ((dir2>>2)%16);
        if(nprol_mat == 16) nprol_mat = 0;

        if(nprol_x == nprol_mat)
            y[i] += gbmv_avx_alin( num_columnas,
                                   &mat[i*columnas + ini_columnas],
                                   &x[ini_columnas], nprol_x, nvec_x);
        else
            y[i] += gbmv_avx_noalin(num_columnas,
                                    &mat[i*columnas + ini_columnas],
                                    &x[ini_columnas], nprol_x,
                                    nvec_x);
    }
}

```

Programa 2.15 Algoritmo vectorial dentro de cada thread (AVX)

El resto del proceso es similar. Si el número de elementos del prólogo de la matriz coincide con el número de elementos del prólogo del vector 'x', ambos vectores están alineados de igual forma y se llamaría a la subrutina `gbmv_avx_alin`. En caso contrario se llamaría a la subrutina `gbmv_avx_noalin`. En este caso se mostrará la subrutina `gbmv_avx_alin` (Programa 2.16) como ejemplo (en contraposición al apartado anterior, en la que se mostró la rutina correspondiente a 'noalin').

La estructura de estas dos subrutinas también es similar a la explicada en el apartado anterior. La principal diferencia es que las variables vectoriales de AVX se declaran como `__mm256` en lugar de `__mm128` (para el caso de SSE). La nomenclatura de las funciones varía de igual forma (en lugar de ser simplemente `_mm_función` son `_mm256_función`). El índice del bucle principal (cuerpo de la función) debe aumentar de ocho en ocho en cada iteración (en lugar de en cuatro), ya que ahora caben el doble de elementos en un vector y podremos realizar las operaciones el doble de rápido. La suma de lo calculado en el bucle principal (recordar que cada elemento de `v_sum` es una suma parcial del resultado total del bucle) ahora debe ser obtenida de una forma algo diferente debido a la estructura de la función `_mm256_hadd_ps()` (Figura 2.23). Una vez realizada la operación `_mm256_hadd_ps()` dos veces sobre el vector (vectorial) `v_sum`, se carga el vector en el vector escalar con `_mm256_store_ps` (alineado, de nuevo, porque el vector `suma` ha sido declarado como `__attribute__((aligned(64))) suma[8];`), y finalmente se suman `suma[0]` y `suma[4]` al resultado final (se deben sumar las posiciones 0 y 4 por lo que se muestra en la Figura 2.23, en la cual se sigue el resultado final con el color rojo, aunque habría más variantes).

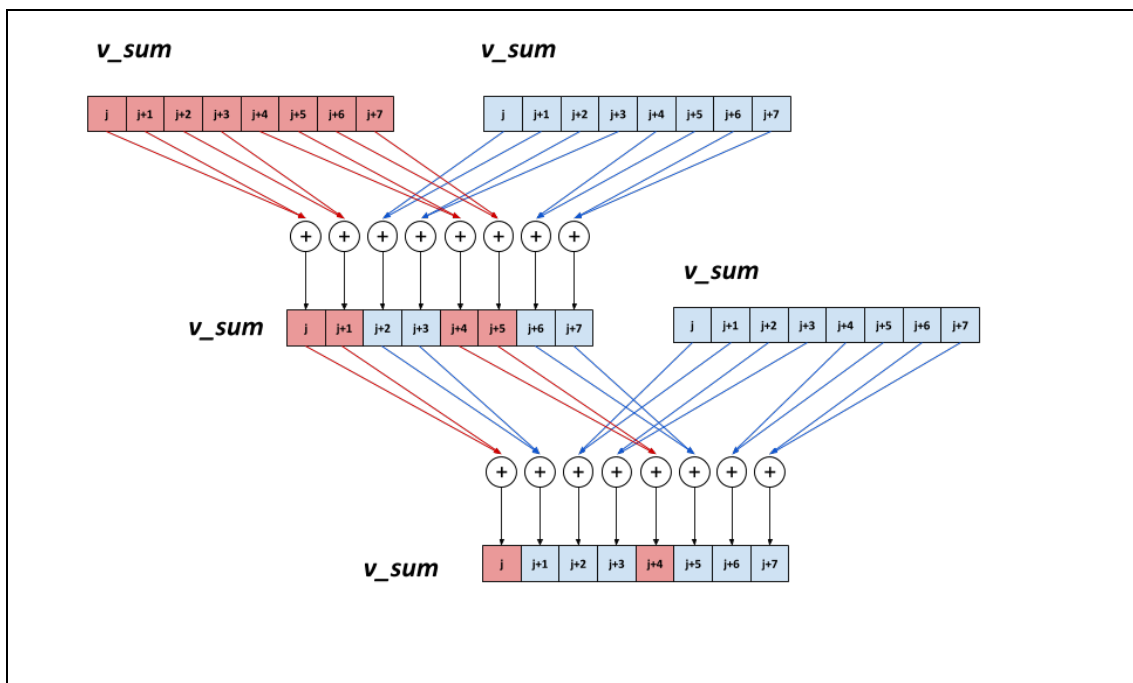


Figura 2.23 Suma de todos los elementos de un vector con `_mm256_hadd_ps()` - AVX

El resto del algoritmo vectorial es similar a lo explicado en el *apartado 2.1.3*.

Con todo esto, cada *thread* habrá calculado su propia parte del vector de resultado 'y'. Tal y como está planteado el algoritmo no es necesario en ningún momento que dos *threads* accedan nunca a la misma posición de memoria (índice) de ninguno de los vectores ni de la matriz.

```
float gbmv_avx_alin( int columnas, float *fila_mat, float *x,
                    int nprol_x, int nvec_x)
{
    float __attribute__((aligned(64))) suma[8];
    float resultado = 0.0;
    int j;

    __m256 v_sum = _mm256_setzero_ps();
    __m256 v_fila_mat, v_x, v_aux;

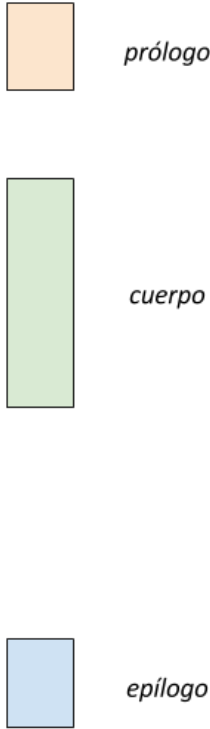
    // Prólogo
    for (j = 0; j < nprol_x; j++)
        resultado += fila_mat[j] * x[j];

    // Cuerpo
    for (j = nprol_x; j < nvec_x; j += 8)
    {
        v_fila_mat = _mm256_load_ps(&fila_mat[j]);
        v_x = _mm256_load_ps(&x[j]);
        v_aux = _mm256_mul_ps(v_fila_mat, v_x);
        v_sum = _mm256_add_ps(v_aux, v_sum);
    }

    // Acumular resultado
    v_sum = _mm256_hadd_ps(v_sum, v_sum);
    v_sum = _mm256_hadd_ps(v_sum, v_sum);
    _mm256_store_ps(&suma[0], v_sum);
    resultado += (suma[0] + suma[4]);

    // Epílogo
    for (j = nvec_x; j < columnas; j++)
        resultado += fila_mat[j] * x[j];

    return resultado;
}
```



Programa 2.16 Subrutina vectorial AVX – producto escalar entre 'x' y una fila de la matriz

2.1.5 Experimentación

El propósito principal de la experimentación es la obtención de un criterio de decisión automatizado según los resultados obtenidos en las pruebas, de forma que el programa se ejecute de la forma más eficientemente posible (secuencial, paralelo solo, vectorial solo, paralelo-vectorial) en función de los parámetros principales (número de filas, número de columnas, número de diagonales inferiores, número de diagonales superiores).

Antes de exponer los resultados obtenidos se explicarán todas las pruebas realizadas. Cada conjunto de pruebas que se explicará ha sido ejecutado en las tres máquinas presentadas en el apartado 1.4 de la memoria, aunque solo se muestran los del ordenador de sobremesa con procesador Intel® Core i7 6700K.

2.1.5.1 Casuística completa de pruebas

Las pruebas han sido realizadas con diferentes parámetros.

Para el tamaño de la matriz:

- *Número de filas:* 32, 100, 500, 1000, 2000 y 4000
- *Número de columnas:* 32, 100, 500, 1000, 2000 y 4000

Para la densidad de los datos (elementos no nulos):

- *Número de diagonales inferiores (% filas):* 0, 12, 25, 50, 75, 87 y 100
- *Número de diagonales superiores (% columnas):* 0, 12, 25, 50, 75, 87 y 100

Recordar que variando el número de diagonales inferiores y diagonales superiores podemos reconstruir todos los tipos diferentes de matrices especificados por BLAS:

Tipo de matriz	subconjunto	Diagonales inferiores (%)	Diagonales superiores (%)	Número de pruebas
Triangular	inferior	100	0	6*
	superior	0	100	6*
Triangular en banda	inferior	$0 < \text{inf} < 100$	0	30*
	superior	0	$0 < \text{sup} < 100$	30*
Banda general	-	$0 < \text{inf} \leq 100$	$0 < \text{sup} < 100$	810
	-	$0 < \text{inf} < 100$	$0 < \text{sup} \leq 100$	810
Densa	-	100	100	36
Diagonal	-	0	0	36

Tabla 2.3 Casuísticas de matrices según inf y sup (%)

*El número de pruebas se ha calculado teniendo en cuenta que las matrices triangulares (tanto normales como en banda) son cuadradas.

Los diferentes números de filas y columnas combinados nos proporcionan **36 matrices** de distintos tamaños. Además, las diferentes bandas nos proporcionan matrices con **49 densidades** distintas. Todas estas pruebas combinadas crean un banco de experimentación de $36 \times 49 =$ **1764 pruebas** a hacer en total en cada una de las **tres máquinas** ($1749 \times 3 = 5247$ pruebas).

Se ha utilizado el registro *TSC (Time Stamp Counter)* con su correspondiente instrucción *rdtsc* para medir el tiempo de cada ejecución en ciclos de *CPU*.

Además, para cada caso particular se han tomado **101 mediciones de tiempo** (ciclos) distintas (101, para despreciar la primera por inicializaciones de variables y *overhead* del SO etc. y quedarnos con las 100 restantes). Se ha calculado el tiempo mínimo, el tiempo máximo y el tiempo medio de cada caso, aunque finalmente solo se ha tenido en cuenta el tiempo medio.

Para cada caso, se ha comprobado el resultado final obtenido (el vector '*y*') con el resultado de ese mismo caso en secuencial. Además, es el tiempo de esta ejecución secuencial el que se ha tomado como referencia para calcular los *speed-up* ($speed-up = tiempo\ en\ secuencial / tiempo\ optimizado$) obtenidos para cada optimización.

Para la ejecución de todas las pruebas en el programa principal (main.h, desde el que se lanza cada caso de prueba) se han incluido dos ficheros *.h. Por un lado *gbmv.def.h*, que define el número de ejecuciones por caso (*Nc*), el número de threads a utilizar (*NTHR*) y el número de diagonales inferiores (*INF*) y superiores (*SUP*) y por otro lado *gbmv.dim.h*, que define el número de filas (*FILAS*) y el número de columnas (*COLUMNAS*) de la matriz, además del parámetro "primeravez" que vale '1' solamente para el primero de los casos (para escribir la cabecera del fichero de pruebas).

gbmv.def.h

```
#define Nc 101
#define NTHR 4
#define SUP 100
#define INF 100
```

gbmv.dim.h

```
#define FILAS 4000
#define COLUMNAS 4000
#define PRIMERAVEZ 0
```

El ejemplo anterior corresponde a una matriz densa (100% de diagonales inferiores y diagonales superiores) de tamaño 4000x4000. Para la ejecución se utilizarán 4 *threads*. Además, esta prueba se ejecutará 101 veces (y se despreciará la primera prueba).

Se ha decidido separar los parámetros en dos ficheros distintos ya que para cada combinación de banda [*INF*, *SUP*] se ejecutan los 36 casos posibles de tamaños de matrices. Por tanto, el fichero *gbmv.dim.h* se tiene que modificar para cada prueba, mientras que el fichero *gbmv.def.h* solo se modificará cada 36 pruebas.

Cada una de las pruebas se compila y lanza por separado mediante un fichero *.bat* tal y como se muestra en el *Programa 2.17*. Se escriben las variables necesarias en los ficheros **.h* correspondientes, se compilan los programas secuencial y paralelo por separado con sus correspondientes *flags*. Se borra el fichero ejecutable (si existe), se hace un *ping* antes de volver a crear el fichero ejecutable (porque el SO a veces no identifica el cambio y lanza el ejecutable anterior) y finalmente se crea el nuevo ejecutable. Una vez creado se inicia la ejecución y se escriben los parámetros en el fichero *gbmv.ticks.txt*.

```

echo #define Nc 101 > gbmh.def.h
echo #define NTHR 4 >> gbmh.def.h
echo #define SUP 0 >> gbmh.def.h
echo #define INF 0 >> gbmh.def.h

echo #define FILAS 32 > gbmh.dim.h
echo #define COLUMNAS 32 >> gbmh.dim.h
echo #define PRIMERA VEZ 1 >> gbmh.dim.h

"gcc.exe" -O3 -c gbmh_seq.c
"gcc.exe" -O3 -fopenmp -c gbmh_omp.c

IF EXIST gbmh.float.exe del gbmh.exe
ping -n 2 0.0.0.0 > nul

"gcc.exe" -O3 -fopenmp -msse4.1 -mavx2
-o gbmh.exe main.c gbmh_seq.o gbmh_omp.o

gbmh.exe > gbmh.ticks.txt

echo #define FILAS 32 > gbmh.dim.h
echo #define COLUMNAS 100 >> gbmh.dim.h
echo #define PRIMERA VEZ 0 >> gbmh.dim.h

"gcc.exe" -O3 -c gbmh_seq.c
"gcc.exe" -O3 -fopenmp -c gbmh_omp.c

del gbmh.exe
ping -n 2 0.0.0.0 > nul

"gcc.exe" -O3 -fopenmp -msse4.1 -mavx2
-o gbmh.exe main.c gbmh_seq.o gbmh_omp.o

gbmh.exe > gbmh.ticks.txt

```

primera
ejecución
matriz
diagonal
32 filas
32 columnas

segunda
ejecución
matriz
diagonal
32 filas
100 columnas

Programa 2.17 Fichero de ejecución de pruebas ".bat"

Todas las pruebas se han realizado con la configuración energética *Alto Rendimiento* de los sistemas operativos Windows. En ningún caso se ha realizado *prefetch* (traer un bloque de datos de la memoria principal antes de que se necesite realmente, evita esperas, ya que la latencia de la memoria principal es alta) ni *loop-unrolling* (escribir 'n' veces el cuerpo del bucle de forma explícita para disminuir el número de instrucciones de salto; aumenta la eficiencia pero también aumenta el tamaño en memoria del programa en sí).

2.1.5.2 Resultados

Tras explicar todas las casuísticas de pruebas se procederá a la exposición y análisis de los resultados. En este documento solamente se explicarán los resultados obtenidos para las pruebas con el procesador *i7 6700K*. Se han realizado y analizado pruebas con todos los sistemas de cómputo presentados en el primer capítulo, pero no han sido incluidos para aligerar la lectura del documento. Asimismo no se hará referencia en ningún caso a datos concretos obtenidos (tablas de datos), todo será explicado mediante gráficas (las gráficas se han realizado mediante el *programa estadístico R*).

Visión general de las pruebas

En la *Figura 2.24* tenemos una visión general de todas las ejecuciones. Esta figura (*boxplot*) representa los ciclos de reloj (en escala logarítmica decimal) que ha tardado en ejecutarse nuestro algoritmo en función de cada combinación de banda inferior y banda superior (en porcentaje). Destacar que para esta gráfica se ha utilizado el mejor valor de *speed-up* obtenido para cada prueba (independientemente de si el mejor valor de *speed-up* ha sido la ejecución en secuencial, la paralela, la vectorial o la paralela-vectorial). Además, en la figura han sido remarcadas algunas ejecuciones más características. En concreto, la matriz diagonal (elementos no nulos solamente en la diagonal principal), la matriz triangular inferior, la matriz triangular superior y la matriz densa.

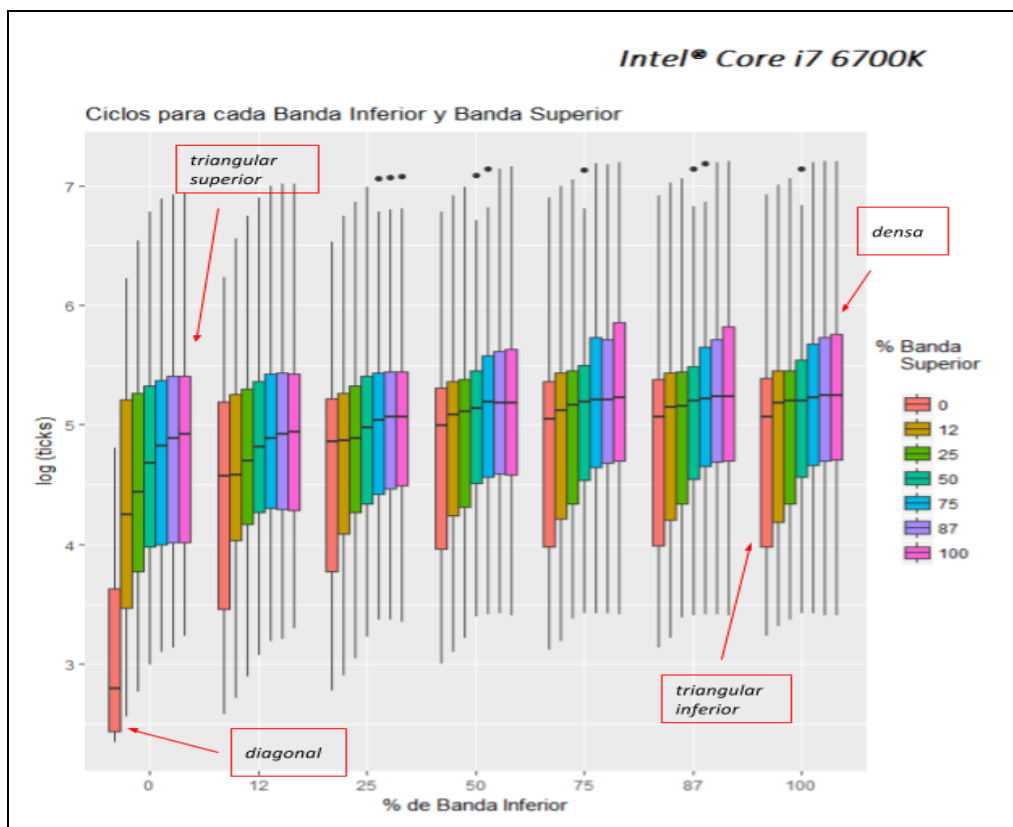


Figura 2.24 Visión general – i7 6700K

Analizando la figura podemos ver que nuestro algoritmo tiene el rendimiento esperado. El comportamiento en general de los *speed-up* es logarítmico y acaba saturando. Conforme aumenta el número de elementos no nulos de la matriz (por incremento del porcentaje de las bandas) las ejecuciones tardan más. Las matrices diagonales son las que con diferencia menos tardan, lo cual era de esperar, ya que son las que precisan de menos cálculos. Las varianzas de las ejecuciones con bandas pequeñas son mucho mayores que las de ejecuciones con bandas grandes. Así pues, las ejecuciones matrices densas son las que menos varianza tienen.

A continuación compararemos las diferentes optimizaciones (*Figuras 2.25 y 2.26*).

Las gráficas de estas figuras comparan el *speed-up* obtenido (ejes 'x' e 'y') entre dos optimizaciones para un mismo número de elementos no nulos de la matriz. Colores azules hacen referencia a menor número de elementos no nulos, colores rojos a mayor número de elementos no nulos (cada gráfica incluye la escala de colores utilizada, aunque es la misma para todas ellas).

Respecto a la comparativa **AVX / SSE** podemos ver que, independientemente del número de elementos no nulos de la ejecución, siempre es mejor (más eficiente) ejecutar con AVX (si nuestro procesador dispone de este conjunto de instrucciones). (*Figura 2.25 primera gráfica*)

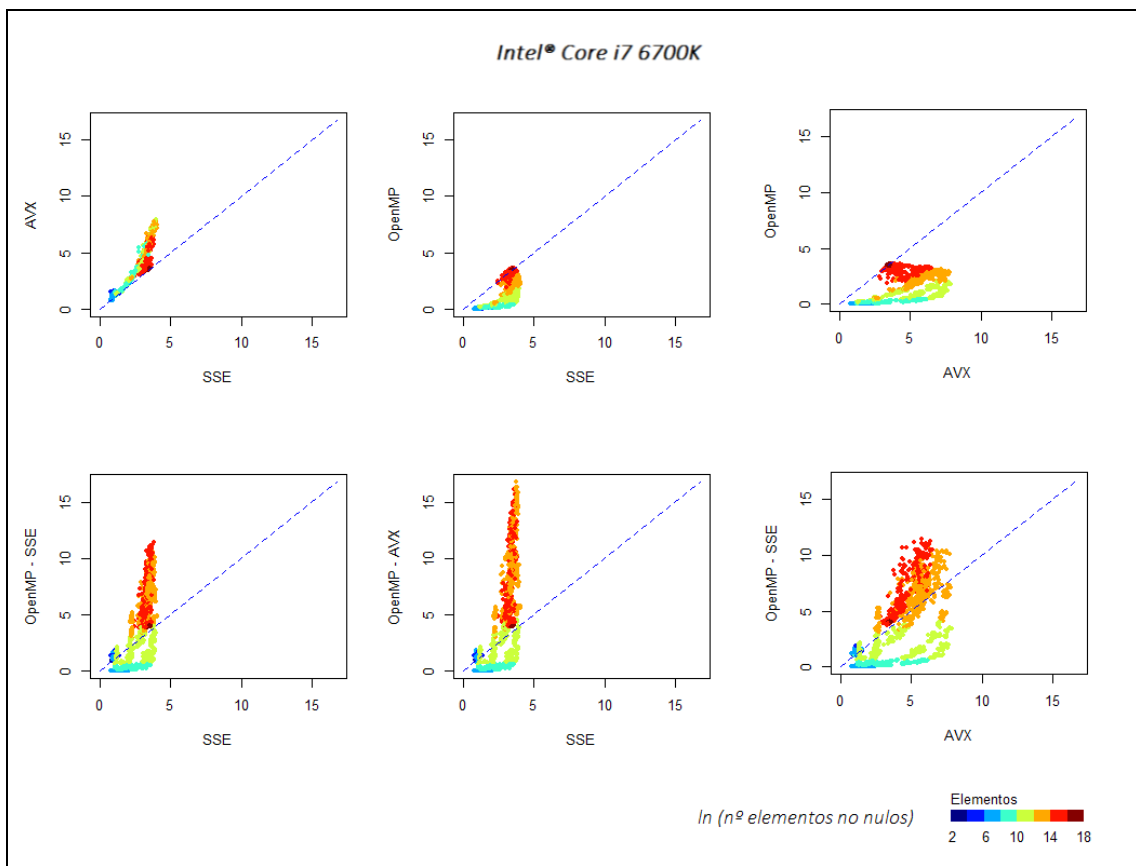


Figura 2.25 Comparativa general optimizaciones 1 – i7 6700K

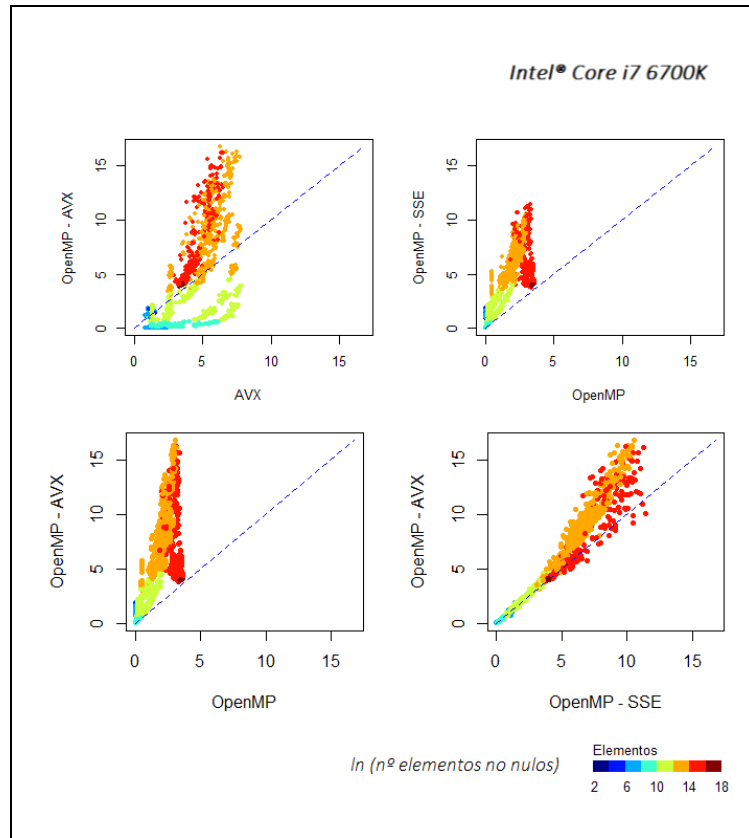


Figura 2.26 Comparativa general optimizaciones 2 – i7 6700K

Respecto a la comparativa **OpenMP / SSE**, tanto para un número de elementos pequeño (tonos azules) como para un gran número de elementos (tonos naranjas-rojos) es mejor utilizar la vectorización. Establecemos por tanto que es mejor ejecutar vectorialmente que paralelizando solo. Lo mismo sucede con la comparativa **OpenMP / AVX**. (Figura 2.25 segunda y tercera gráfica)

Respecto a las comparativas **OpenMP-SSE / SSE** y **OpenMP-AVX / SSE** (paralelo-vectorial / vectorial SSE) en general podemos decir que a partir de los 160 mil elementos no nulos (e^{22} elementos – colores a partir de naranja) es mejor combinar paralelización y vectorización (cualquiera de los dos tipos) en lugar de usar solo la vectorización con SSE. A la paralelización se le puede sacar mejor rendimiento cuantos más elementos tengamos para repartir (ya que es entonces cuando se puede despreciar el *overhead* frente al coste del cálculo en sí). Hemos visto que la paralelización por sí sola no aporta demasiado, pero combinando ambos métodos obtenemos *speed-up* de incluso más de 15. (Figura 2.25 cuarta y quinta gráfica)

Lo mismo que sucede para las comparativas anteriores, sucede también para las comparativas **OpenMP-SSE / AVX** y **OpenMP-AVX / AVX** (paralelo-vectorial / vectorial AVX). La única diferencia significativa es que, en general, el *speed-up* obtenido mediante AVX (en el intervalo de menor número de elementos no nulos) es mejor que el *speed-up* obtenido con SSE (podemos ver que es aproximadamente el doble, como era de esperar). (Figura 2.25 sexta gráfica Figura 2.26 primera gráfica)

En las comparativas *OpenMP-SSE / OpenMP* y *OpenMP-AVX / OpenMP* (paralelo-vectorial / paralelo solo) vemos que en *ningún caso* es mejor utilizar solo la paralelización. (Figura 2.26 segunda y tercera gráfica)

Finalmente, cuando comparamos *OpenMP-SSE / OpenMP-AVX* podemos ver que la tendencia es que la paralelización combinada con las instrucciones AVX es algo mejor, aunque las diferencias son bastante pequeñas. (Figura 2.26 cuarta gráfica)

Por último, para reafirmar las conclusiones obtenidas hasta el momento y refinarlas se ha realizado otra serie de gráficas que ilustran mejor el criterio de uso de las tecnologías. En este caso se ilustran en función del número de filas y en función de la banda superior.

Las gráficas de **discriminación general (izquierda)** muestran, en función del número de filas y el número de diagonales superiores de la matriz (ambos representados utilizando una escala logarítmica), en qué casos la optimización mencionada es mejor que la ejecución en secuencial (verde, si el *speed-up* obtenido para esa combinación es mayor que uno, rojo si es menor).

Cuando entramos en las gráficas **en detalle (derecha)** se ilustran en rojo los *speed-up* entre cero y dos, en azul los *speed-up* entre dos y cuatro y en verde los *speed-up* mayores que cuatro para las respectivas configuraciones de $\log_2(\text{filas})$ y la banda superior.

Se han elegido como criterio el número de filas y el número de elementos de la banda superior ya que son los criterios principales que afectan a la paralelización y a la vectorización respectivamente. Además, estos son parámetros que el programador especifica en la llamada a la función.

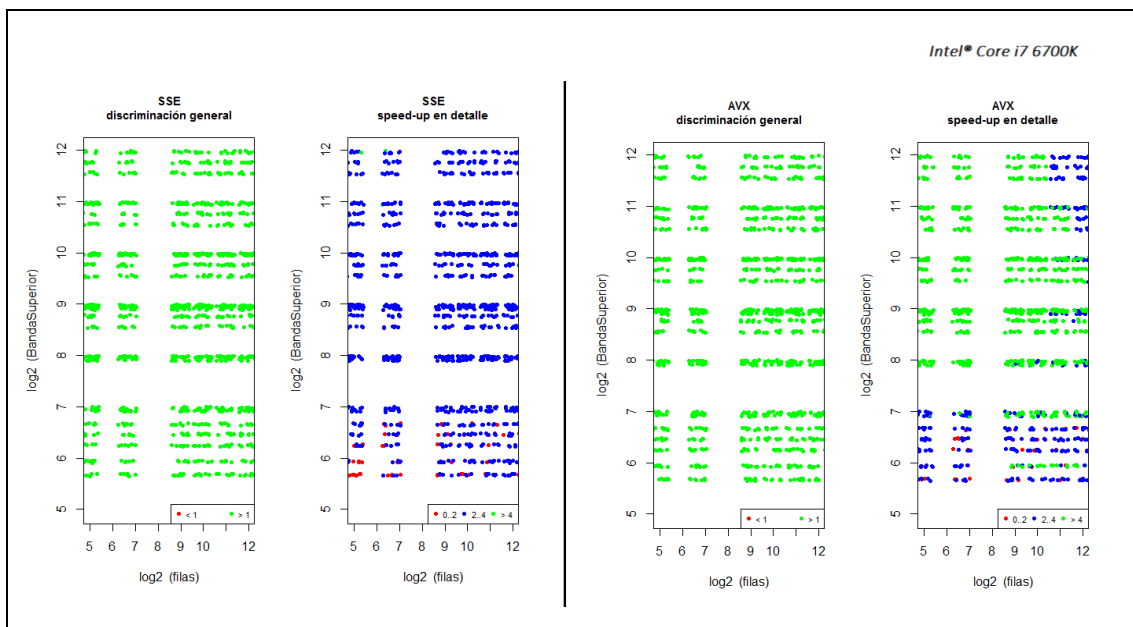


Figura 2.27 Discriminación de uso y speed-up en detalle 1 – i7 6700K

Así pues, **comenzando por analizar las ejecuciones vectoriales** (Figura 2.27), vemos que la ejecución SSE siempre es mejor que la ejecución secuencial (salvo algunas excepciones puntuales que no son representativas del comportamiento general). Para este procesador en concreto todas las ejecuciones consiguen entre dos y cuatro de *speed-up* (nunca más de cuatro, ya que el *speed-up* máximo ideal utilizando SSE es de cuatro, por el tamaño del registro vectorial).

La ejecución en AVX también es siempre mejor que la ejecución en secuencial. Sin embargo, en la gráfica detallada observamos un comportamiento algo diferente al de la anterior (SSE). Para bandas pequeñas (< 128 [2⁷] aproximadamente) el *speed-up* obtenido es menor (entre dos y cuatro). En el resto de los casos obtenemos más de cuatro de *speed-up*. Sin embargo, aparecen otros casos puntuales (aparte de los de banda pequeña) en los que el *speed-up* obtenido no es mayor a cuatro.

Para poder explicar esto hemos relacionado las configuraciones con el tamaño de la memoria caché del procesador. Podemos ver esta relación en la Figura 2.28. En ella tenemos por bloques (con las escalas de las gráficas anteriores) el tamaño de la matriz en memoria (es aproximado ya que estamos trabajando solamente con el número de elementos de la banda superior y no con el número de columnas en sí). Se ilustra en verde el conjunto de matrices que sí caben enteras en caché, en naranja las configuraciones de matrices que puede ser que no quepan y finalmente en rojo el conjunto de configuraciones de matrices que no caben en memoria caché.

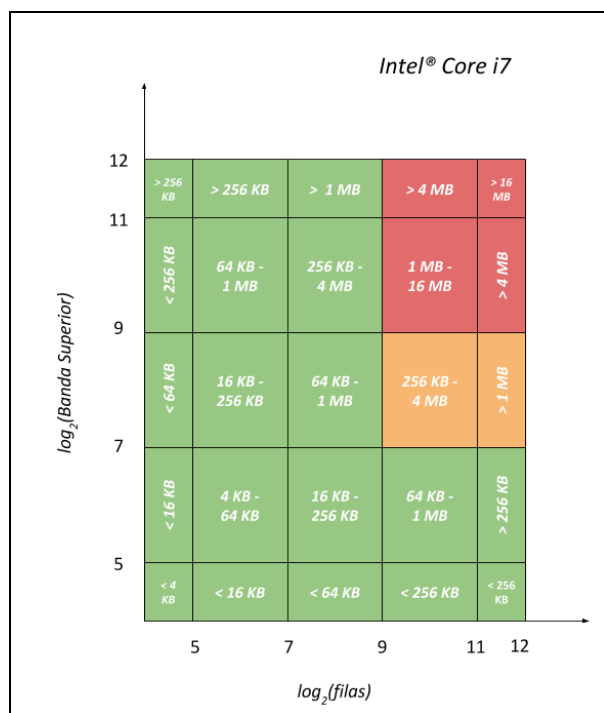


Figura 2.28 Relación de las configuraciones y la memoria caché

En la Figura 2.29 podemos ver un ejemplo de cómo la memoria caché influye en el rendimiento de las diferentes configuraciones de matrices.

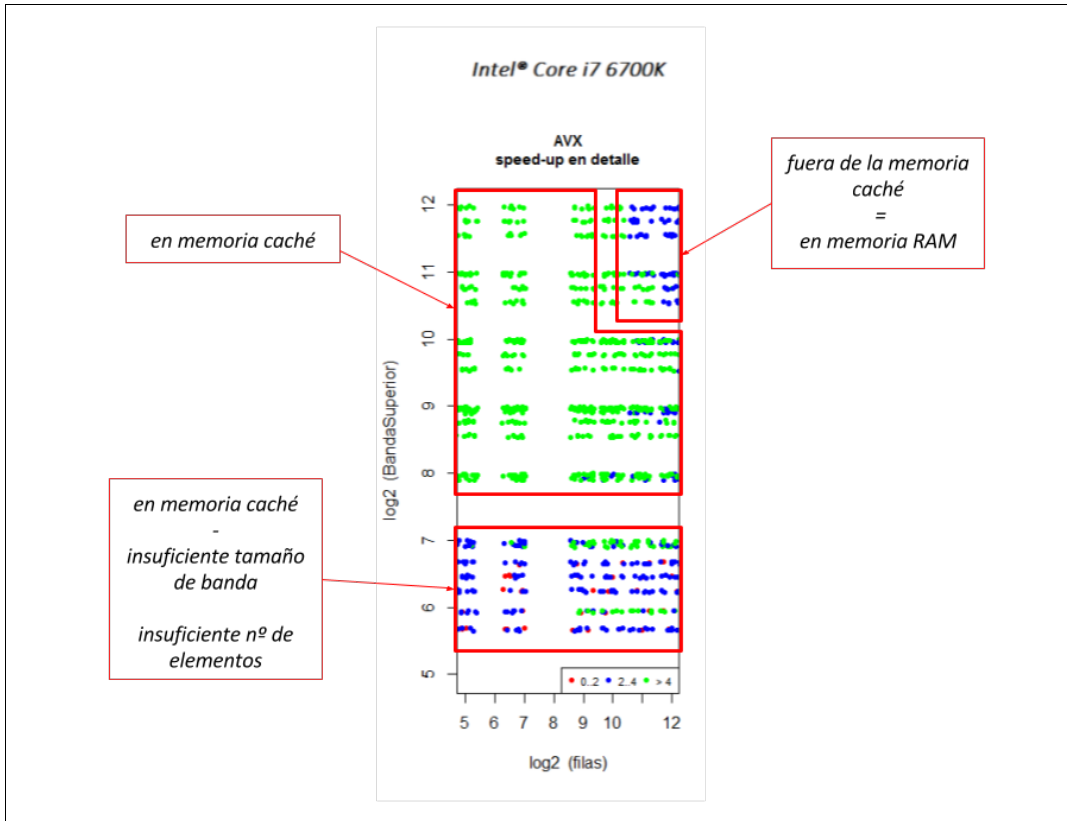


Figura 2.29 Explicación ejemplo con AVX y procesador i7 6700K

Continuamos el análisis con las ejecuciones paralelas-vectoriales (*OpenMP-SSE* y *OpenMP-AVX* respectivamente). (Figura 2.30)

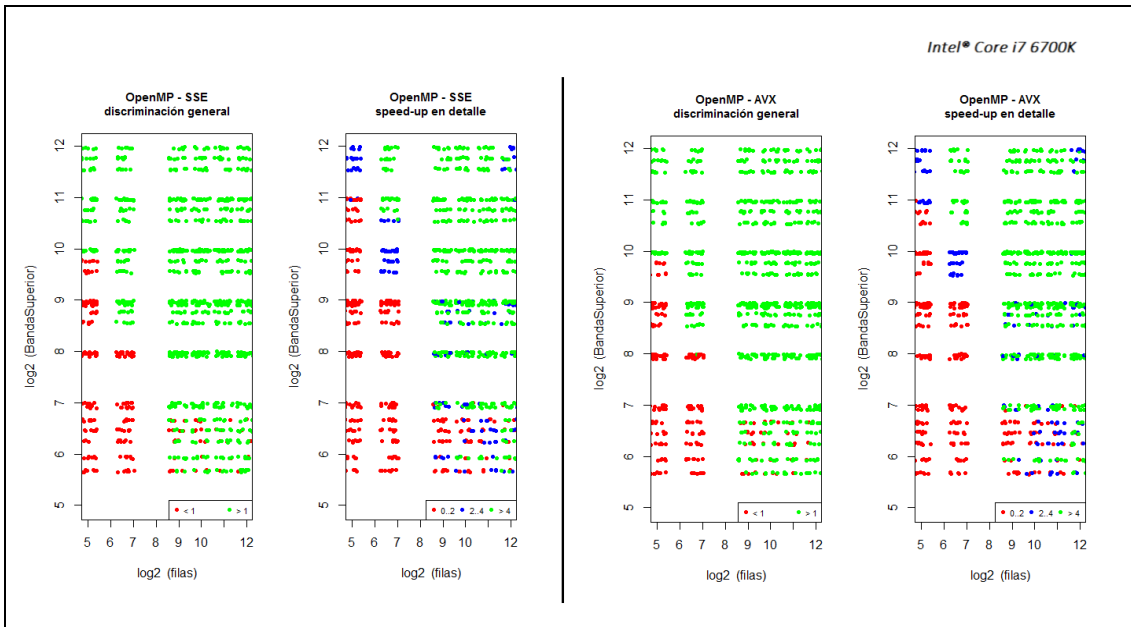


Figura 2.30 Discriminación de uso y speed-up en detalle 2 – i7 6700K

En cuanto a la discriminación general, en ambos tipos de ejecución no obtenemos *speed-up* mayores que uno para pocos elementos. Esto se debe a que la carga de trabajo a repartir no es lo suficientemente grande como para que el *overhead* de la paralelización sea despreciable respecto al tiempo de los cálculos en sí. Es aquí donde se puede apreciar la importancia de tener suficiente carga computacional (en este caso, suficientes filas con suficientes elementos) para obtener mejor rendimiento mediante la paralelización.

Finalmente tenemos una gráfica con las **ejecuciones paralelas** y una con la **comparativa general** del uso de las diferentes optimizaciones (Figuras 2.31).

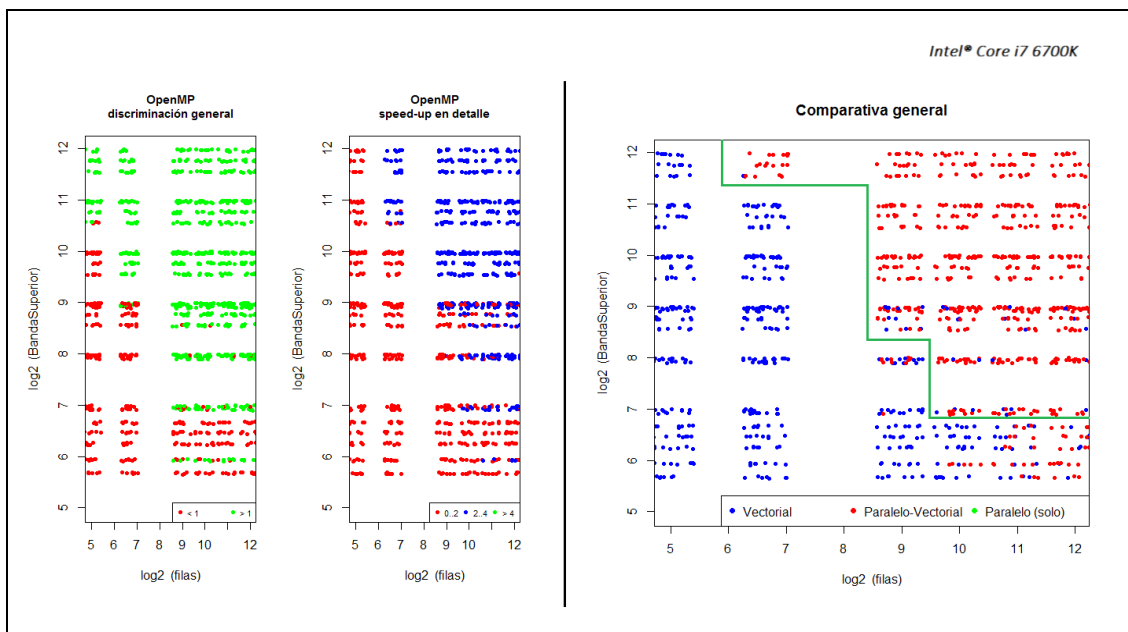


Figura 2.31 Discriminación de uso y *speed-up* en detalle 3 – i7 6700K

En cuanto a las gráficas de *OpenMP*, al igual que sucedía en el caso anterior, hasta que no tenemos suficientes elementos a procesar no obtenemos *speed-up*.

Todos los datos expuestos serán resumidos a continuación en la *Tabla 2.4*. En ella se establecen los criterios de uso de todas las técnicas (y sus combinaciones). Las técnicas de una fila se deberán utilizar frente a las técnicas de las columnas siempre y cuando se cumpla la condición establecida en la casilla correspondiente de la tabla. En caso de mostrar un 'tic', la condición será "siempre se usa la técnica de la fila respecto a la de la columna", en caso de mostrar una 'cruz' la condición será "nunca se usa la técnica de la fila respecto a la de la columna".

En las dos últimas columnas se muestra el *speed-up* obtenido por la técnica de la correspondiente fila. En algunos casos es necesario especificar con qué configuraciones se obtiene entre dos y cuatro de *speed-up* y con qué configuraciones se obtiene más de cuatro de *speed-up*.

	Secuencial	OpenMP	SSE	AVX	Open-SSE	Open-AVX	2 < s.u. < 4	s.u. > 4
Secuencial	-	*C1	✗	✗	*C2	*C2	-	-
OpenMP	complementario *C1	-	✗	✗	✗	✗	✓	-
SSE	✓	✓	-	✗	*C3	*C3	✓	-
AVX	✓	✓	✓	-	*C4	*C4	*SU1	complementario *SU1
Open-SSE	complementario *C2	✓	complementario *C3	complementario *C4	-	✗	*SU2	*SU2
Open-AVX	complementario *C2	✓	complementario *C3	complementario *C4	✓	-	*SU3	*SU3

Tabla 2.4 Resumen de los criterios de uso BLAS2 (optimización explícita)

Criterios:

*C1 filas < $2^6 \wedge BS < 2^{10}$
filas < $2^7 \wedge BS < 2^9$
filas > $2^7 \wedge BS < 2^7$

*C2 filas < $2^6 \wedge BS \leq 2^{10}$
filas < $2^7 \wedge BS < 2^8$

*C3 filas < 2^6
filas < $2^7 \wedge BS < 2^{10}$

*C4 $BS < 2^{11}$
filas < $2^{11} \wedge BS > 2^{11}$

Speed-up:

*SU1 $BS \leq 2^7$
filas > $2^{11} \wedge BS > 2^{11}$

*SU2 filas < $2^6 \wedge BS > 2^{11}$
filas < $2^7 \wedge 2^9 < BS < 2^{10}$

*SU3 filas > $2^8 \wedge BS \geq 2^7$
filas > $2^6 \wedge BS > 2^{10}$

Cabe destacar que el máximo de *speed-up* obtenido por SSE cuatro y el máximo *speed-up* obtenido por AVX es ocho por el tamaño de los *registros vectoriales*. El máximo *speed-up* obtenido por OpenMP es de cuatro por disponer solamente de cuatro *threads*. Con Open-SSE y Open-AVX conseguimos *speed-up* mayores por la consiguiente combinación de los métodos.

2.2 Intel® C++ Compiler (ICC)

Con respecto al *Intel® C++ Compiler* se explicarán en primer lugar los ficheros con los que se ha trabajado y sus respectivas configuraciones para el compilador. A continuación se hará un análisis de la autoparalelización y autovectorización de la función genérica por parte del compilador mediante la herramienta *Intel® Advisor* – presentada en el apartado 1.3.3.3 de la memoria. Tras esto se introducirá al uso de la librería *MKL* de *Intel® (Math Kernel Library)*. Finalmente se analizarán todos los resultados obtenidos (algoritmo en secuencial, algoritmo autoparalelizado y autovectorizado y su contraste con el uso de las correspondientes funciones de *MKL*).

2.2.1 Configuración del entorno

Tras la creación del proyecto como se ha explicado en el apartado 1.3.3.1 de la memoria, se han creado otros dos ficheros en el proyecto. Por un lado “seq.cpp” y por otro lado “opt.cpp”. Cada uno de estos ficheros contiene la función genérica tal y como se presentó al comienzo. Se han incluido en dos ficheros separados para poder configurar el compilador de forma que el fichero “seq.cpp” no se optimice al compilar, pero el fichero “opt.cpp” se optimice al completo (autoparalelización y autovectorización).

Para configurar esto, los pasos que se han seguido son los siguientes:

1. Teniendo seleccionado nuestro proyecto completo en el *explorador de soluciones* (para aplicar los próximos cambios al proyecto en su totalidad) seleccionamos en el menú principal “Proyecto” → “Propiedades” (Figura 2.32). Se nos abrirá otra ventana nueva.

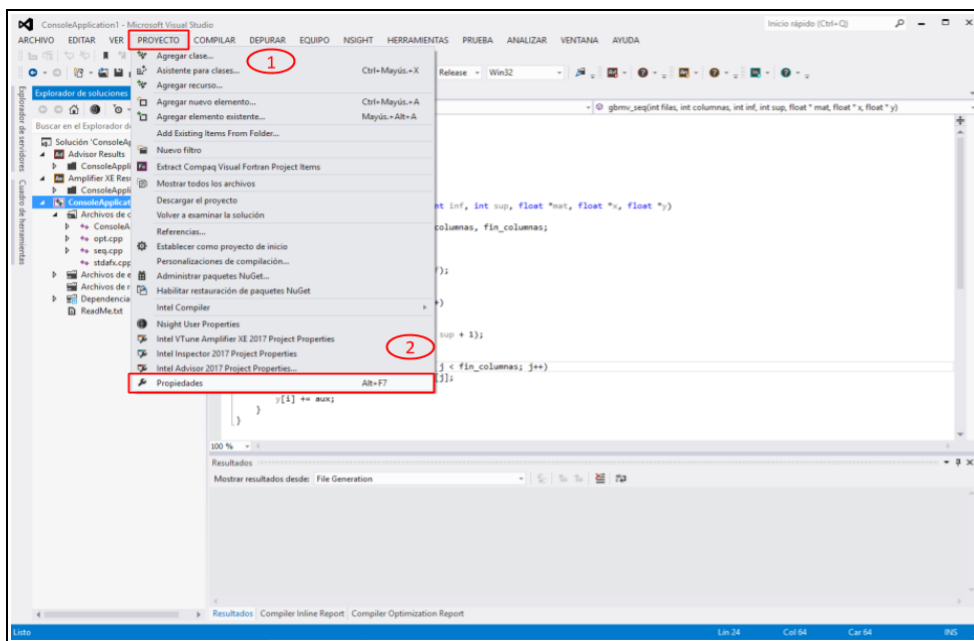


Figura 2.32 Configuración del entorno - 1

- En esta nueva ventana, en primer lugar seleccionaremos “Propiedades de configuración” → “Intel performance libraries” y seleccionaremos “parallel” en la opción “Use Intel® MKL” dentro del desplegable “Intel® Math Kernel library” (Figura 2.33).

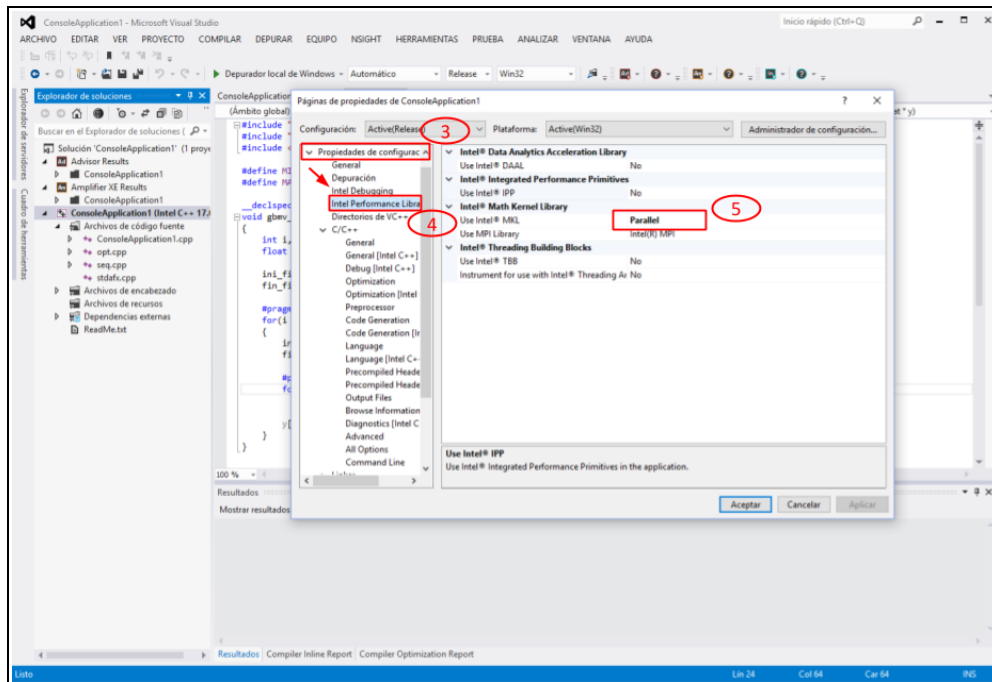


Figura 2.33 Configuración del entorno - 2

- Finalmente, estando en las “Propiedades de configuración”, desplegamos la opción “C/C++” y nos situamos en “All options”(Figura 2.34). La configuración que se modificará a continuación en realidad lo que hace es modificar los diferentes *flags* del compilador de forma automatizada en lugar de tener que introducirlos manualmente como lo hemos hecho con GCC (también se pueden introducir de forma manual en el menú “Command Line”).

La configuración que modificaremos es la siguiente:

- **Additional Include Directories:** *ruta del Intel Advisor*
- **Enable C99 Support:** **Sí (/Qstd=c99)** - *(para habilitar el uso de C99)*
- **Enable Enhanced Instruction Set:** **Intel(R) Advanced Vector Extension (/arch: AVX)** – *(seleccionar las instrucciones de las que dispone nuestro procesador)*
- **Enable Function-Level Linking:** **Sí (/Gy)** – *(para la optimización de las funciones por parte del compilador; permite empaquetar funciones similares)*
- **Enable Intrinsic Functions:** **Sí (/Oi)** – *(para el uso de intrínsecas; autovectorización)*

- **Inline Function Expansion: Only `__inline (/Ob1)`** – (para que el compilador sustituya las llamadas a las funciones con las funciones directamente; mejor rendimiento)
- **OpenMP Support: Generate Parallel Code (`/Qopenmp`)** – (para poder utilizar OpenMP con este compilador; equivalente a `-fopenmp` en GCC)
- **Optimization: Highest Optimizations (`/O3`)** – (para las optimizaciones del compilador; nivel más alto)
- **Parallelization: Sí (`/Qparallel`)** – (para la autoparalelización)
- **Precompiled Header: Use (`/Yu`)** – (para la inclusión de ficheros de cabecera `*.h`)

Además, para realizar pruebas existen se han tocado otros dos parámetros:

- **Guided Auto Parallelism Analysis: Extreme (`/Qguide=4`)** – (para el análisis de la autoparalelización)
- **Optimization Diagnostic Level: Level 5 (`/Qopt-report:5`)** – (para el análisis de las optimizaciones en general; genera un reporte de optimización)

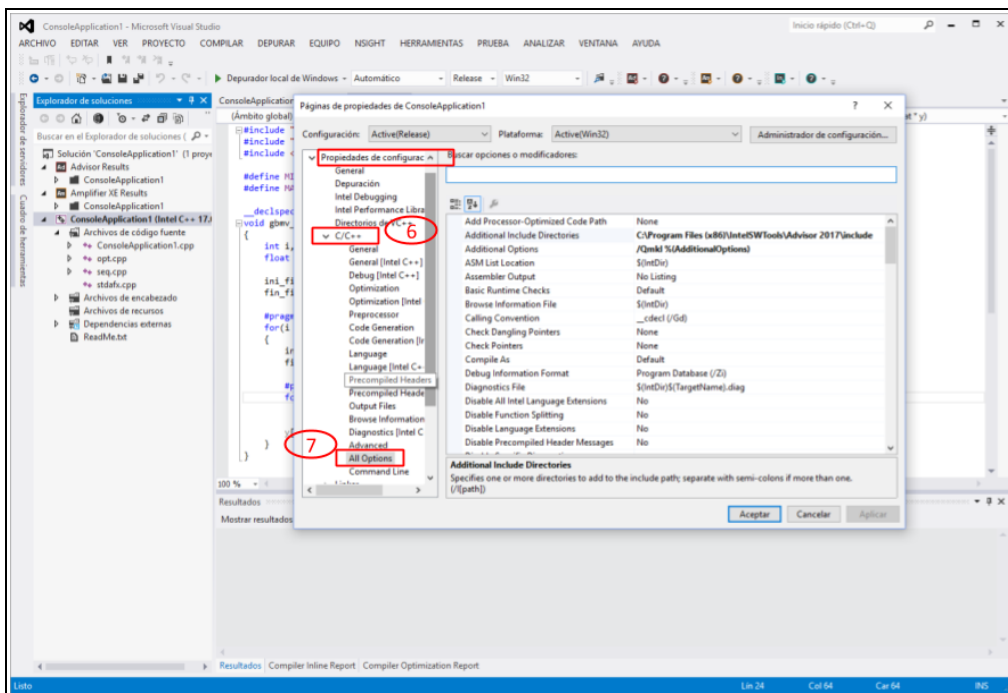


Figura 2.34 Configuración del entorno - 3

Como esta configuración se aplica a todo el proyecto pero no queremos que el fichero “seq.cpp” sea optimizado lo seleccionamos en el *explorador de soluciones* y repetimos el proceso. Seleccionamos “Proyecto” → “Propiedades” → “Propiedades de configuración” → “C/C++” → “All options” y desactivamos las opciones siguientes:

- Enable Enhanced Instruction Set: Not set
- Enable Intrinsics Functions: No
- OpenMP Support: No
- Parallelization: No
- *Guided Auto Parallelism Analysis: Disable*
- *Optimization Diagnostic Level: Disable*

Estas dos últimas opciones dependiendo de la situación pueden variar (por ejemplo, cuando queramos hacer pruebas).

El fichero principal, el equivalente a “`main.c`” en *GCC*, desde el cual se lanzan todas las pruebas, se comprueban los resultados y se guardan los *speed-up* en un fichero, no ha sido configurado de una manera especial, se ha dejado tal y como hemos configurado el proyecto en general. Con esto ya tenemos configurado nuestro entorno de desarrollo de la forma adecuada.

2.2.2 Autoparalelización y autovectorización e Intel® Advisor

Comenzaremos mostrando cómo ha autoparalelizado y autovectorizado el compilador nuestra función genérica. Tras ello haremos un análisis algo más exhaustivo mediante la herramienta Intel® Advisor.

2.2.2.1 Autoparalelización y autovectorización

Tras haber configurado el proyecto como se menciona en el apartado 2.2.1 el compilador ya autoparaleliza y autovectoriza nuestra función genérica (que ahora se encuentra en el fichero “`opt.cpp`”). Para poder ver exactamente qué es lo que ha hecho el compilador, se han activado los *flags* “*Guided Autoparallelism Analysis: Extreme (/Qguide=4)*” y “*Optimization Diagnostic Level: Level 5 (/Qopt-report:5)*” para activar los reportes del compilador. Por tanto, una vez compilado el proyecto con estos parámetros podemos ver junto al código anotaciones de las optimizaciones realizadas por el compilador.

En la *Figura 2.35* podemos ver que el compilador ha autoparalelizado el bucle principal (el recorrido en filas) al igual que hacíamos nosotros de forma explícita en *GCC*. Además, el compilador nos muestra las variables declaradas como *firsprivate*, *lastprivate*, etc. También nos dice que el bucle principal no ha sido vectorizado, ya que lo ha sido el interior (el recorrido en columnas), de nuevo, tal y como hacíamos de forma explícita con *GCC*.

```

#define MIN(x,y) (((x)<(y)) ? (x) : (y))
#define MAX(x,y) (((x)>(y)) ? (x) : (y))

__declspec (noinline)
void gbmv_opt(int filas, int columnas, int inf, int sup, float *mat, float *x, float *y)
{
    int i, j, ini_filas, fin_filas, ini_columnas, fin_columnas;
    float aux;

    ini_filas = 0;
    fin_filas = MIN(filas, columnas + inf);

    for(i = ini_filas; i < fin_filas; i++)
    {
        ini_columnas = MAX(0, i - inf);
        fin_columnas = MIN(columnas, i + sup + 1);

        for(j = ini_columnas, aux = 0.0; j < fin_columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
}

```

Figura 2.35 Autoparalelización / autovectorización – Bucle principal

En la Figura 2.36 podemos ver que el bucle interior (recorrido en columnas) ha sido vectorizado pero **no paralelizado**. Además, para la vectorización del bucle se ha hecho **loop peeling**, es decir, separar el bucle en varios trozos para conseguir el alineamiento correcto (similar a lo efectuado con la vectorización explícita: prólogo, cuerpo, epílogo). Además nos dice el **speed-up potencial estimado** con esta optimización: aproximadamente **3.79**.

```

__declspec (noinline)
void gbmv_opt(int filas, int columnas, int inf, int sup, float *mat, float *x, float *y)
{
    int i, j, ini_filas, fin_filas, ini_columnas, fin_columnas;
    float aux;

    ini_filas = 0;
    fin_filas = MIN(filas, columnas + inf);

    for(i = ini_filas; i < fin_filas; i++)
    {
        ini_columnas = MAX(0, i - inf);
        fin_columnas = MIN(columnas, i + sup + 1);

        for(j = ini_columnas, aux = 0.0; j < fin_columnas; j++)
            y[i] += mat[i*columnas+j] * x[j];
    }
}

```

Figura 2.36 Autoparalelización / autovectorización – Bucle interior

Finalmente, para asegurarnos de que el compilador no optimice la función que queremos ejecutar en secuencial (en el fichero “seq.cpp”), se han añadido las cláusulas #pragma novector delante de cada uno de los bucles (tanto del principal como del interior). En la Figura 2.37 podemos ver que el compilador, efectivamente, no ha realizado ninguna optimización en el fichero “seq.cpp” en general.

```

#define MIN(x,y) (((x)<(y)) ? (x) : (y))
#define MAX(x,y) (((x)>(y)) ? (x) : (y))

__declspec (noinline)
void gbmv_seq(int filas, int columnas, int inf, int sup, float *mat, float *x, float *y)
{
    int i, j, ini_filas, fin_filas, ini_columnas, fin_columnas;
    float aux;

    ini_filas = 0;
    fin_filas = MIN(filas, columnas + inf);

    #pragma novector
    for(i = ini_filas; i < fin_filas; i++)
    {
        ini_columnas = MAX(0, i - inf);
        fin_columnas = MIN(columnas, i + sup + 1);

        #pragma novector
        for(j = ini_columnas, aux = 0.0; j < fin_columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
}

```

Figura 2.37 Fichero “seq.cpp” - sin optimizaciones

Se ha comprobado todo ello también mediante la herramienta Intel® Advisor. En la Figura 2.38 podemos ver un análisis más exhaustivo de la vectorización que ha efectuado el compilador.

Function Call Sites and Loops	Total Time	Type	Why No Vectorization?	Vectorized Loops	Instruction Set Analysis	Advanced	Location
[loop in gbmv_seq at seq.cp...	85,198s	Scalar	novector dire...				seq.cpp:24
[loop in gbmv_opt at opt.cp...	36,901s	Vectorized (Body; Peelt; Remainder)	1 vectorizatio...	AVX	Inserts	Unrolled by 2	opt.cpp:22
[loop in gbmv_opt at opt.cp...	35,808s	Vectorized (Body)		AVX	Inserts	Unrolled by 2	opt.cpp:22
[loop in gbmv_opt at opt.cp...	0,641s	Peelt					opt.cpp:22
[loop in gbmv_opt at opt.cp...	0,453s	Remainder	vectorization ...				opt.cpp:22
[loop in (MKL BLAS)	29,281s	Vectorized (Body)		AVX2	FMA		
[loop in func@0x1005b12e]	13,223s	Vectorized (Body)		AVX2	FMA		
[loop in inicializar_x_mat at C...	26,711s	Scalar	function call ...		Type Conversions		ConsoleApplic...

Figura 2.38 Análisis Intel® Advisor

En primer lugar nos muestra el porcentaje de tiempo de la ejecución total que ha estado en ejecución nuestro bucle principal (*loop in gbm_opt at opt.cpp* - 19.9%) y el tiempo en segundos (36.9 s) **1**. Nos muestra que ha realizado *loop peeling* (descomposición en tres) **2**: *body, peeled, remainder*. Nos dice que la eficiencia es de aproximadamente un 48% y que ganamos aproximadamente 3.8 de *speed-up* **3**. La longitud de los vectores que ha utilizado es de ocho elementos (ya que se ha vectorizado con AVX: 256 bits ~ 8 float). Los tipos de datos utilizados son float de 32 bits y además muestra que se ha realizado *loop unrolling* de dos (transformar el bucle en un bucle equivalente que realice dos iteraciones en una, reduciendo así el número de saltos a la mitad) **4**. También podemos ver que MKL se ejecuta mediante el conjunto de instrucciones FMA (*Fused Multiply-Add*) **5**, un conjunto de instrucciones vectorial similar a los explicados (SSE, AVX) con la salvedad de que permite multiplicar y sumar dos vectores en un mismo ciclo de reloj. Además, aunque no sea el caso de nuestra función *auto-optimizada*, hay que tener en cuenta especialmente las anotaciones de tipo *Type Conversion*, dado que suelen ser operaciones problemáticas respecto a pérdida de eficiencia **6**.

Para comprobar la **eficiencia de la paralelización** también se ha utilizado esta herramienta, pero ha sido necesario incluir anotaciones especiales de dicha herramienta en el código:

- ANNOTATE_SITE_BEGIN() - antes del bucle principal (región paralela)
- ANNOTATE_ITERATION_TASK() - antes del bucle interior (tarefas)
- ANNOTATE_SITE_END() - finalizar la región paralela

Podemos ver estas anotaciones en el Programa 2.18 (en negrita).

```
void gbm_opt( int filas, int columnas, int inf, int sup, float *mat,
             float *x, float *y)
{
    int i, j, ini_filas, fin_filas, ini_columnas, fin_columnas;
    float aux;

    ini_filas = 0;
    fin_filas = MIN(filas, columnas + inf);

    ANNOTATE_SITE_BEGIN( MySite1 );
    for(i = ini_filas; i < fin_filas; i++)
    {
        ini_columnas = MAX(0, i - inf);
        fin_columnas = MIN(columnas, i + sup + 1);

        ANNOTATE_ITERATION_TASK( MyTask1 );
        for(j = ini_columnas, aux = 0.0; j < fin_columnas; j++)
            aux += mat[i*columnas+j] * x[j];

        y[i] += aux;
    }
    ANNOTATE_SITE_END();
}

```

Programa 2.18 Anotaciones Intel® Advisor

En la *Figura 2.39* a la izquierda se muestra una gráfica en la que podemos ver el *speed-up* que obtendría nuestro programa tal y como lo hemos hecho. A la derecha podemos ver todas las optimizaciones que puede realizar el compilador.

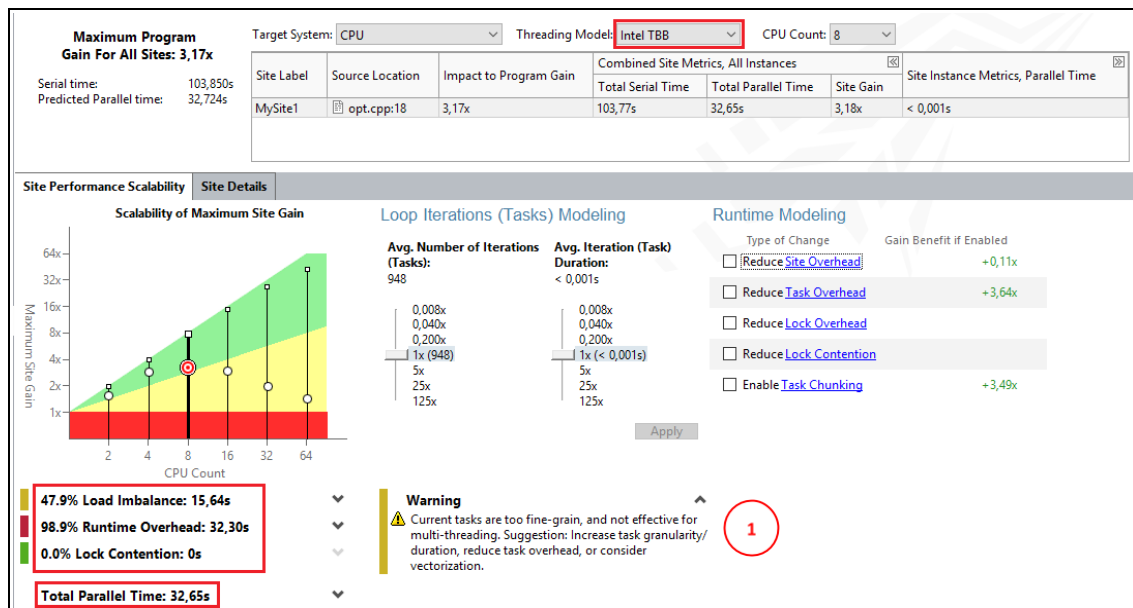


Figura 2.39 Análisis paralelización Intel® Advisor – antes

Las optimizaciones más destacables que nos propone son las siguientes:

- **Enable Task Chunking:** *Habilitar el reparto de tareas (load balance) con un tamaño de 'chunk' óptimo.*
- **Reduce Task Overhead:** *Reducir el tiempo perdido al crear una tarea y asignársela a un thread (en nuestro caso tiene poco interés, ya que hemos comprobado que la mejor ejecución se obtiene con reparto de carga estático; no se crean tareas de forma dinámica)*
- **Reduce Site Overhead:** *Reducir el tiempo perdido al crear una sección paralela y cerrar dicha sección paralela.*

El compilador además nos advierte de que la **granularidad** de nuestras tareas es demasiado fina (cada tarea requiere de demasiado poco tiempo de ejecución). Nos aconseja aumentar la granularidad de la tarea para poder así también aumentar la eficiencia del algoritmo (1).

En estos ejemplos estamos situados en **Thread Modeling: Intel TBB** (remarcado en la *Figura 2.39*), lo cual quiere decir que estamos utilizando los procedimientos de Intel® para crear *threads*. Podríamos cambiar esta opción y analizar, por ejemplo, *Thread Modeling: OpenMP*.

Tras aplicar, hipotéticamente, estas optimizaciones el programa dice que el tiempo total en la sección paralela se ha reducido a 24,76 s de 156,12 s. El *speed-up* que se obtendría sería cercano a ocho si disponemos de ocho *threads* (*Figura 2.40*).

Cabe destacar que todos los análisis efectuados en este apartado han sido realizados con el

equipo de sobremesa (*Core i7 6700K*). Los datos podrían variar de un procesador a otro. Por ejemplo, en la *Figura 2.40* con el procesador *i3* tendríamos que situarnos en *CPU Count = 4* (ya que solamente dispone de cuatro *threads*).

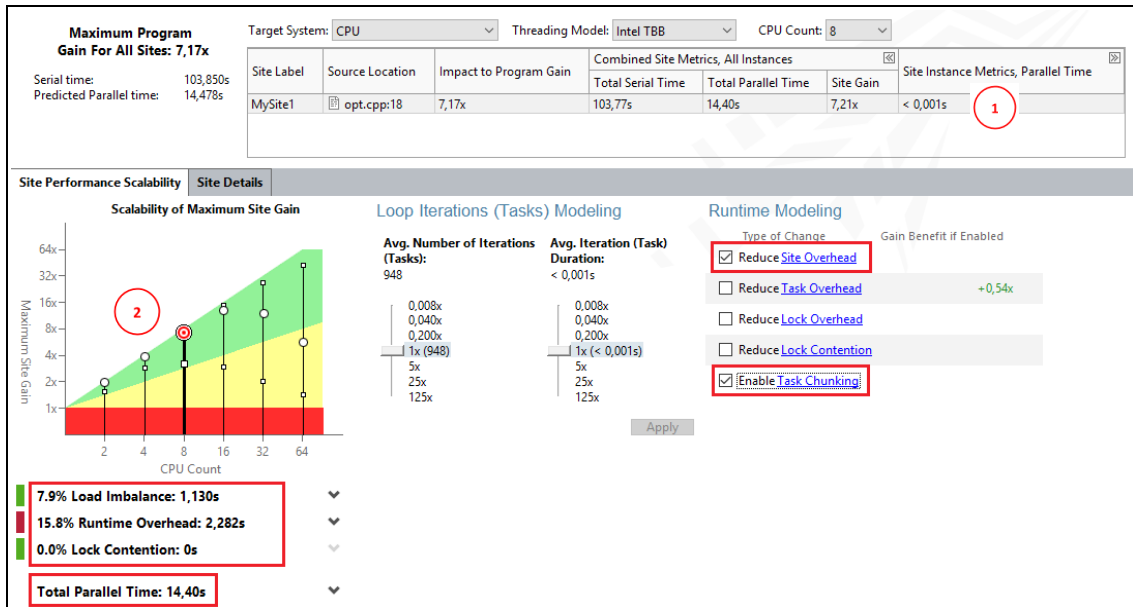


Figura 2.40 Análisis paralelización Intel® Advisor – después

Para obtener esta gráfica Intel® Advisor tiene en cuenta la **Ley de Amdahl**:

$$speed-up = \frac{1}{[(1 - F) + F/P] + K}$$

Siendo 'F' el porcentaje del programa paralelizable, 'P' el número de threads y 'K' el overhead de la paralelización.

Habiendo obtenido el tiempo secuencial [1-F] y el tiempo paralelo del programa [F] (1 en la Figura 2.40), el programa extrapola el problema a diferentes números de threads (2 en la Figura 2.40). Podemos apreciar que la aproximación consigue el *speed-up* máximo con 16 threads (*CPU Count = 16*), pero que a partir de entonces satura y decae. Esto se debe al *overhead* 'K' de la creación de las tareas.

2.2.4 Librerías MKL

A continuación explicaremos el uso que se le ha dado a la librería *MKL* de Intel®.

El propósito de su uso es contrastar los *speed-up* obtenidos al usar funciones de esta librería con los *speed-up* obtenidos al autoparalelizar y autovectorizar nuestro algoritmo.

Como esta librería implementa las rutinas de la especificación *BLAS* tal y como aparecen en la propia especificación y nosotros disponemos de un banco de pruebas muy amplio, se ha tenido que obtener un criterio para saber cuándo utilizar una función de *MKL* u otra (*Programa 2.19*):

- Si la matriz es una matriz triangular
 - Si es cuadrada: se llamará a la función *MKL* `cblas_strmv()`
 - Si no es cuadrada: se llamará a la función *MKL* `cblas_sgbmv()`
- Si la matriz es triangular en banda
 - Si es cuadrada: se llamará a la función *MKL* `cblas_stbmv()`
 - Si no es cuadrada: se llamará a la función *MKL* `cblas_sgbmv()`
- Si la matriz es densa: se llamará a la función *MKL* `cblas_sgemv()`
- Si la matriz es diagonal: se llamará a la función *MKL* `cblas_sgbmv()`
- Si no cumple ninguno de los anteriores: se llamará a la función *MKL* `cblas_sgbmv()`

```
if ( (inf == 0 && sup == columnas) || (sup == 0 && inf == filas) )
    if (filas == columnas)
        caso = TR_C; //strmv
    else
        caso = TR_R; //sgbmv
else if ((inf == 0 && sup > 0 && sup < columnas ) ||
        (sup == 0 && inf > 0 && inf < filas ))
    if (filas == columnas)
        caso = TB_C; //stbmv
    else
        caso = TB_R; //sgbmv
else if ( inf == filas && sup == columnas )
    caso = GE; //sgemv
else if (inf == 0 && sup == 0)
    caso = GB_D; //sgbmv
else
    caso = GB; //sgbmv
```

Programa 2.19 Criterio de uso de funciones *MKL*

A continuación se explicará la forma de utilizar las diferentes funciones de *MKL* nombradas.

2.2.4.1 *MKL* – *cblas_strmv()*

Esta función *MKL* es la correspondiente a *STRMV* de la especificación *BLAS*. Es para matrices triangulares (inferiores o superiores) cuadradas. Efectúa el cálculo siguiente: $x \leftarrow A x$.

En el *Programa 2.20* podemos ver cómo se ha utilizado la función (*referencia*: <https://software.intel.com/en-us/node/520772>). Los parámetros son los siguientes:²

- *CblasRowMajor*: si estamos programando en C, este es el parámetro que se le debe pasar a la función. Si estuviésemos trabajando en Fortran sería *CblasColMajor*.
- *CblasLower* / *CblasUpper*: indica si la matriz es triangular inferior o triangular superior. En nuestro caso se comprueba previamente y se llama a la función de una forma u otra.
- *CblasNoTrans*: para indicar que no estamos trabajando con matrices traspuestas.
- *CblasNonUnit*: para indicar que nuestra diagonal principal no es unidad.
- *filas*: el número de filas de la matriz (en este caso *filas* == *columnas*, por ser cuadrada)
- *mat*: la matriz con la que se va a operar.
- *lda*: debe ser el máximo entre '1' y '*columnas*', por lo que *lda* == *filas* == *columnas*.
- *x*: el vector con el que se va a operar.
- *incx* = '1': indica el *stride* entre elementos en '*x*'. Se trabaja con elementos consecutivos por lo que se le pasa el valor '1'.

```
if (sup == 0)
    cblas_strmv(CblasRowMajor, CblasLower, CblasNoTrans, CblasNonUnit,
               filas, mat, lda, x, 1);
else
    cblas_strmv(CblasRowMajor, CblasUpper, CblasNoTrans, CblasNonUnit,
               filas, mat, lda, x, 1);

y[0:filas] += x[0:columnas];
```

Programa 2.20 Uso de *cblas_strmv()* de *MKL*

Tras ejecutar la función debemos sumarle el vector '*x*' (donde *cblas_strmv* nos deja el resultado) al vector '*y*' para poder compararlo con nuestra ejecución habitual ($y \leftarrow A x + y$). Se ha hecho la suma de los vectores con sintaxis C99.

² En las siguientes funciones utilizadas de *MKL* únicamente se dará la referencia de Intel® *BLAS* Routines correspondiente. No se explicarán todos los parámetros explícitamente.

2.2.4.2 MKL – cblas_stbmv()

Esta función MKL es la correspondiente a STBMV de la especificación BLAS. Es para matrices triangulares (inferiores o superiores) cuadradas en banda. Efectúa el cálculo siguiente: $x \leftarrow A x$. En el Programa 2.21 podemos ver cómo se ha utilizado la función (referencia: <https://software.intel.com/en-us/node/520768>).

En este caso es necesario pasarle a la función la matriz modificada de una forma específica (la forma de modificarla viene en la página de documentación de Intel® BLAS Routines, el enlace a la página se encuentra en el apartado de Bibliografía y referencias).

El valor de lda ahora depende de si estamos trabajando con una matriz triangular inferior en banda (en cuyo caso $lda = inf + 1$) o con una matriz triangular superior en banda (en cuyo caso $lda = sup + 1$).

```
if (sup == 0)
{
    lda = inf + 1;
    wmat = (float*) aligned_malloc(lda*columnas*sizeof(float),64);

    for (int i = 0; i < columnas; i++)
    {
        m = inf - i;
        for (int j = max(0, i - inf); j <= i; j++)
            wmat[(m+j) + i*lda] = mat[j + i * columnas];
    }

    cblas_stbmv(CblasRowMajor, CblasLower, CblasNoTrans,
               CblasNonUnit, filas, inf, wmat, lda, x, 1);
}
else
{
    lda = sup + 1;
    wmat = (float*) _aligned_malloc(lda*columnas*sizeof(float),64);

    for (int i = 0; i < columnas; i++)
    {
        m = -i;
        for (int j = i; j < MIN(columnas, i + sup + 1); j++)
            wmat[(m + j) + i * lda] = mat[j + i * columnas];
    }

    cblas_stbmv(CblasRowMajor, CblasUpper, CblasNoTrans,
               CblasNonUnit, filas, sup, wmat, lda, x, 1);
}

y[0:filas] += x[0:columnas];
aligned_free(wmat);
```

Transformación a la matriz

Programa 2.21 Uso de cblas_stbmv() de MKL

Para poder ejecutar la función se reserva memoria de forma dinámica para una nueva matriz del tamaño necesario (especificado también en la documentación de Intel®). El espacio que ocupa esta matriz será liberado poco después de la ejecución de la función.

Se sigue el siguiente orden de ejecución:

- Calculamos el valor de *lda* correspondiente
- Reservamos memoria para la matriz
- Efectuamos la transformación a la matriz
- Ejecutamos la rutina `cblas_stbmv()`
- Sumamos el vector '*x*' al vector '*y*' (al igual que en el caso anterior)

Como la transformación de la matriz tiene un coste computacional significativo, a la hora de realizar las pruebas se tomarán dos datos distintos: uno imputando el coste de la transformación a la función *MKL* y otro sin tenerlo en cuenta.

2.2.4.3 MKL – `cblas_sgemv()`

Esta función *MKL* es la correspondiente a *SGEMV* de la especificación *BLAS*. Es para matrices densas (cuadradas o rectangulares indistintamente). Efectúa el cálculo siguiente: $y \leftarrow A \alpha x + \beta y$. En el Programa 2.22 podemos ver cómo se ha utilizado la función (referencia: <https://software.intel.com/en-us/node/520750>).

```
cblas_sgemv(CblasRowMajor, CblasNoTrans, filas, columnas, 1.0, mat,
           lda, x, 1, 1.0, y, 1);
```

Programa 2.22 Uso de `cblas_sgemv()` de *MKL*

2.2.4.4 MKL – `cblas_sgbmv()`

Esta función *MKL* es la correspondiente a *SGBMV* de la especificación *BLAS*. Es para matrices en banda general. Efectúa el cálculo siguiente: $y \leftarrow A \alpha x + \beta y$. En el Programa 2.23 podemos ver cómo se ha utilizado la función (referencia: <https://software.intel.com/en-us/node/520749>).

En este caso, al igual que sucedía con `cblas_stbmv()`, es necesario pasarle a la función la matriz modificada de una forma específica (la forma de modificarla para esta función también viene en la página de documentación de Intel® *BLAS Routines*).

Para poder ejecutar la función se reserva memoria de forma dinámica para una nueva matriz del tamaño necesario (especificado también en la documentación de Intel®). El espacio que ocupa esta matriz será liberado poco después de la ejecución de la función.

Se sigue el siguiente orden de ejecución:

- Reservamos memoria para la matriz
- Efectuamos la transformación a la matriz
- Ejecutamos la rutina `cblas_sgbmv()`

Como la transformación de la matriz tiene un coste computacional significativo, a la hora de realizar las pruebas se tomarán dos datos distintos: uno imputando el coste de la transformación a la función *MKL* y otro sin tenerlo en cuenta.

```
// Reservar espacio para la matriz (alineado).
wmat = (float*) _aligned_malloc(lda*filas*sizeof(float),64);

// Aplicar la transformación a la matriz
for(int i = 0; i < filas; i++)
{
    k = inf - i;
    for(int j = MAX(0, i-inf); j < MIN(columnas, i + sup + 1); j++)
        wmat[(j + k) + i * lda] = mat[j + i * columnas];
}

cblas_sgbmv(CblasRowMajor, CblasNoTrans, filas, columnas, inf, sup,
            1.0, wmat, lda, x, 1, 1.0, y, 1);

// Liberar espacio (alineado).
_aligned_free(wmat);
```

Programa 2.23 Uso de `cblas_sgbmv()` de *MKL*

2.2.5 Intel® Advisor – Roofline

A continuación analizaremos los programas compilados con el compilador de Intel® (secuencial, auto-optimizado y MKL) mediante la herramienta Intel® Advisor Roofline (*partiendo de lo explicado en el apartado 1.3.3.4 de la memoria*). En la *Figura 2.41* podemos ver el gráfico referente a estos tres programas.

En el gráfico el programa secuencial se resalta con un punto rojo. Significa que en él recaerá el mayor tiempo de ejecución, como es normal. Este programa está acotado por la velocidad de la memoria principal (memoria RAM) del ordenador (líneas diagonales, *DRAM Bandwidth 30.31 GB/s*) además de por el propio procesamiento sin utilización de instrucciones adicionales (líneas horizontales, *Scalar Add Peak 31.24 GFLOPS*). Esto quiere decir, que teóricamente el programa nunca podrá tener mayor ancho de banda que el que ofrece la propia memoria RAM, ni podrá tener más GFlops que los que se indican en la cota.

Por otro lado, el programa auto-optimizado y el programa MKL están ambos acotados por caché de nivel 3 (*L3 Bandwidth 258.8 GB/s*), la cual ofrece un ancho de banda muchísimo mayor del que ofrece la memoria principal. Sin embargo, según el gráfico, estos dos programas, al igual que el programa secuencial, están acotados por el procesamiento escalar (*Scalar Add Peak 31.24 GFLOPS*).

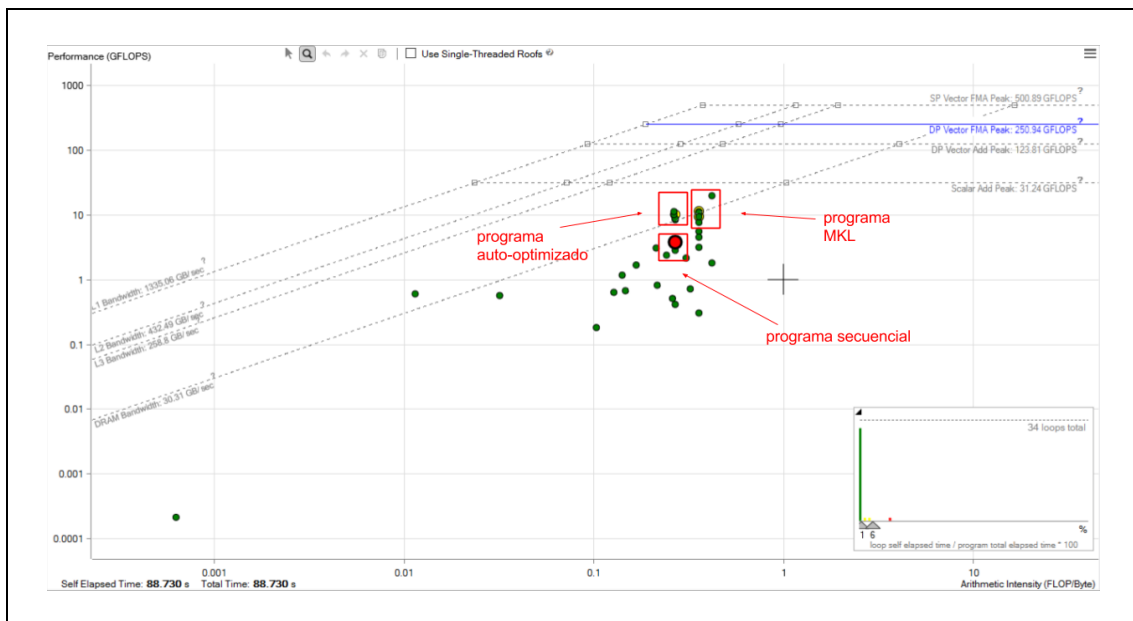


Figura 2.41 Intel® Advisor – Roofline; BLAS2

En la *Figura 2.42* hemos recopilado la información que da *Intel® Roofline* sobre los tiempos de ejecución de cada una de las funciones (*secuencial, optimizada y MKL; de izquierda a derecha en la figura*).

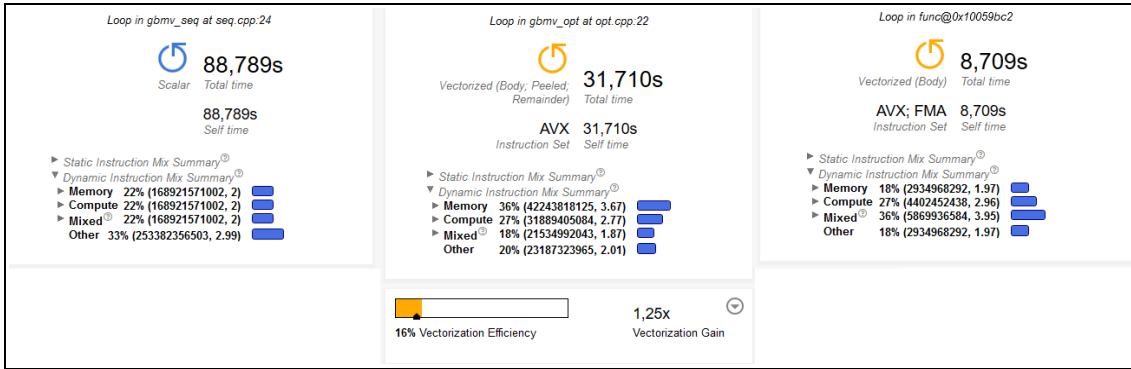


Figura 2.42 Intel® Roofline – Tiempos de ejecución

Como podemos ver, el programa secuencial está estimado en *88.78 segundos*, mientras que nuestro programa *auto-optimizado* tiene un tiempo de ejecución estimado de *31 segundos*. La función *MKL* (denotada como *func@0x...*) únicamente está estimada en *8.7 segundos* de ejecución. También nos muestra el porcentaje de instrucciones de cada tipo en cada una de las funciones (*memoria*, *cómputo*, *'mezcla'* y *otros*). Finalmente, podemos apreciar que la función auto-optimizada ha sido vectorizada con *AVX* (con una eficiencia estimada del 16%), mientras que *MKL* utiliza la combinación de las intrínsecas *AVX* y *FMA*, uno de los motivos por los cuales resulta más eficiente.

2.3 Comparación de resultados obtenidos

Finalmente concluiremos el capítulo con la comparación de todos los resultados obtenidos hasta el momento. Se recuerda que la experimentación ha constado principalmente de las siguientes partes:

- Compilador GCC (caso mejor)
- Compilador Intel® auto-optimización
- Compilador Intel® librerías MKL (métodos *TRMV*, *TBMV*, *GEMV*, *GBMV*)

En la *Figura 2.43* podemos ver un gráfico de comparación de todos los *speed-up* obtenidos. También se muestran los *speed-up* obtenidos de unos métodos respecto de otros (*GCC/AUTO*, *GCC/MKL* y *AUTO/MKL*)

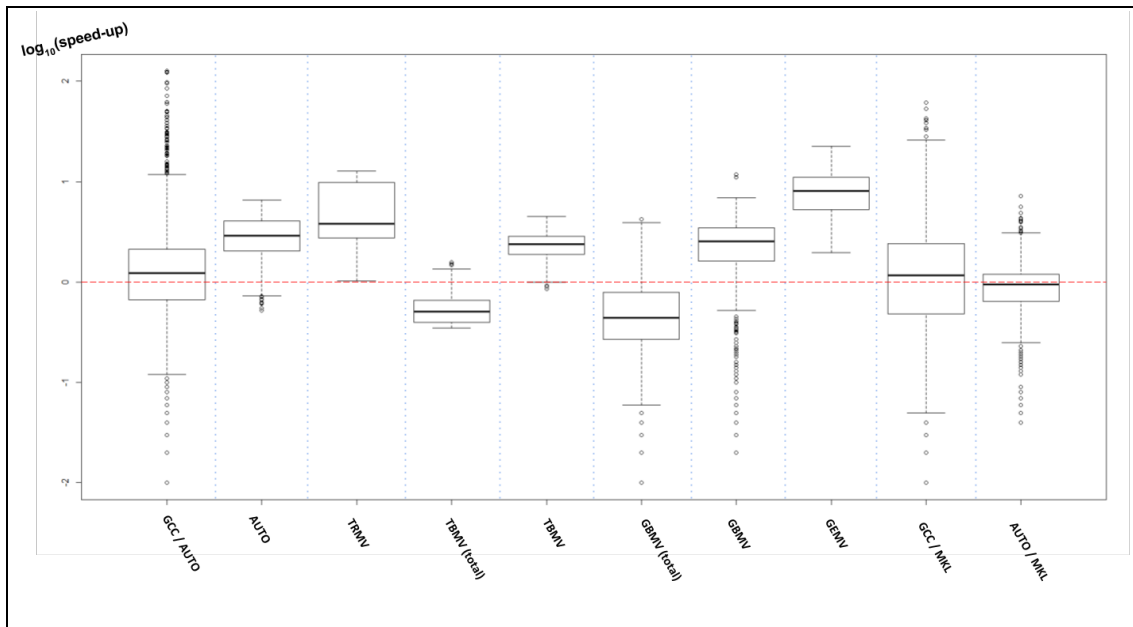


Figura 2.43 Comparativa general de resultados

En la *Tabla 2.5* podemos ver los datos correspondientes a los *boxplot* de la figura anterior.

	GCC / AUTO	AUTO	TRMV	TBMV (total)	TBMV	GBMV (total)	GBMV	GEMV	GCC / MKL	AUTO / MKL
Mínimo	0,01	0,52	1,03	0,35	0,86	0,01	0,02	1,97	0,01	0,04
Percentil 25	0,67	2,04	2,76	0,4	1,89	0,27	1,62	5,27	0,48	0,64
Mediana	1,23	2,89	3,82	0,51	2,4	0,44	2,55	8,1	1,17	0,95
Percentil 75	2,13	4,07	9,83	0,66	2,88	0,79	3,5	11,06	2,41	1,2
Máximo	126,54	6,58	12,68	1,58	4,51	4,24	11,77	22,66	61,33	7,19

Tabla 2.5 Comparativa general de resultados; datos concretos

Para usar las funciones *MKL TBMV* y *GBMV* era necesario transformar la matriz en primer lugar (ver apartado 2.2.4). Las columnas de la *Tabla 2.4 TBMV (total)* y *GBMV (total)* son teniendo en cuenta el tiempo (sobrecoste) que añade transformar la matriz previamente. Las columnas *TBMV* y *GBMV* a secas se han calculado teniendo en cuenta únicamente el cómputo en sí.

Tras analizar los datos podemos establecer los criterios de uso reflejados en la *Tabla 2.6*. Esta tabla ha sido diseñada de la misma forma que la *Tabla 2.4*. Las técnicas de una fila se deberán utilizar frente a las técnicas de las columnas siempre y cuando se cumpla la condición establecida en la casilla correspondiente de la tabla.

	GCC	Auto	MKL
GCC	-	Cuando GCC ejecuta en serie	Cuando GCC ejecuta en serie
Auto	Cuando GCC no ejecuta en serie	-	Cuando no son matrices densas (MKL no utiliza las funciones <i>TRMV / GEMV</i>)
MKL	Cuando GCC no ejecuta en serie	Cuando son matrices densas (MKL utiliza las funciones <i>TRMV / GEMV</i>)	-

Tabla 2.6 Resumen general de los criterios de uso (compiladores y MKL)

Recuérdese que al programar manualmente en GCC se impuso una restricción para hacer todo el algoritmo en secuencial: si el número de elementos no nulos de una fila era menor que 32. Gracias a este criterio somos capaces de superar en *speed-up* a MKL en algunos casos.

3

Especificación BLAS 3

En este capítulo se expondrán los algoritmos correspondientes a la especificación *BLAS de nivel 3* implementados. Cabe destacar que solamente se han trabajado con el compilador de *Intel*[®], por lo que todas las optimizaciones y resultados se expondrán en función de las herramientas correspondientes (*Intel*[®] VTune Amplifier, *Intel*[®] Advisor e *Intel*[®] Roofline principalmente).

3.1 Algoritmos

En primer lugar se expondrán los algoritmos realizados para la rutina *BLAS de nivel 3*, *GEMM*, cuya especificación es la siguiente:

$$C \leftarrow A B + C$$

En todos los algoritmos que se expondrán en este apartado los tamaños de las matrices se denominan de la siguiente forma:

- La matriz C de dimensiones [filas][columnas]
- La matriz A de dimensiones [filas][kcolumnas]
- La matriz B de dimensiones [kcolumnas][columnas]

Esta rutina de *BLAS de nivel 3* es la más significativa en cuanto al rendimiento, y por ello, la única estudiada en este apartado. Además, todas las pruebas se han realizado con matrices densas cuadradas, dado que también ayuda a comprobar de forma más sencilla los rendimientos obtenidos mediante los distintos algoritmos.

3.1.1 Algoritmo secuencial

El algoritmo secuencial que se ha utilizado para contrastar los demás en cuanto a rendimiento es el que podemos ver en el *Programa 3.1*.

```

void MxM_seq ( int filas, int columnas, int kcolumnas,
              float C[filas][columnas],
              float A[filas][kcolumnas],
              float B[kcolumnas][columnas])
{
    // Para que el compilador presuponga que las matrices ya están
    // alineadas en memoria (dado que han sido declaradas así)
    __assume_aligned (C,64);
    __assume_aligned (A,64);
    __assume_aligned (B,64);

    int i,j,k;

    // "#pragma novector" para que el compilador no vectorice los
    // bucles
    #pragma novector
    for (i=0;i<filas;i++)
    {
        #pragma novector
        for (j=0;j<columnas;j++)
        {
            #pragma novector
            for (k=0;k<kcolumnas;k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    }
}

```

sintaxis /C99

alineamiento en memoria

Programa 3.1 Algoritmo MxM Secuencial

3.1.2 Algoritmo de intercambio de bucles (i, j, k → i, k, j)

Un algoritmo comúnmente utilizado para la multiplicación de matrices es el del intercambio de los dos bucles interiores. Tal y como estamos acostumbrados a multiplicar matrices

normalmente, estaríamos recorriendo la matriz B por columnas, es decir, con $stride = \text{columnas}$, saltando entre elementos no consecutivos de memoria constantemente. Esto ralentiza en gran medida estos accesos a memoria ya que propicia los fallos en los accesos a memoria en los niveles más bajos de memoria caché. Por tanto, en lugar de recorrer las matrices en orden " i, j, k " se intercambian los dos bucles interiores (i, k, j) para permitir los accesos a memoria con $stride = 1$. La *Figura 3.1* puede resultar de ayuda para la visualización del algoritmo.

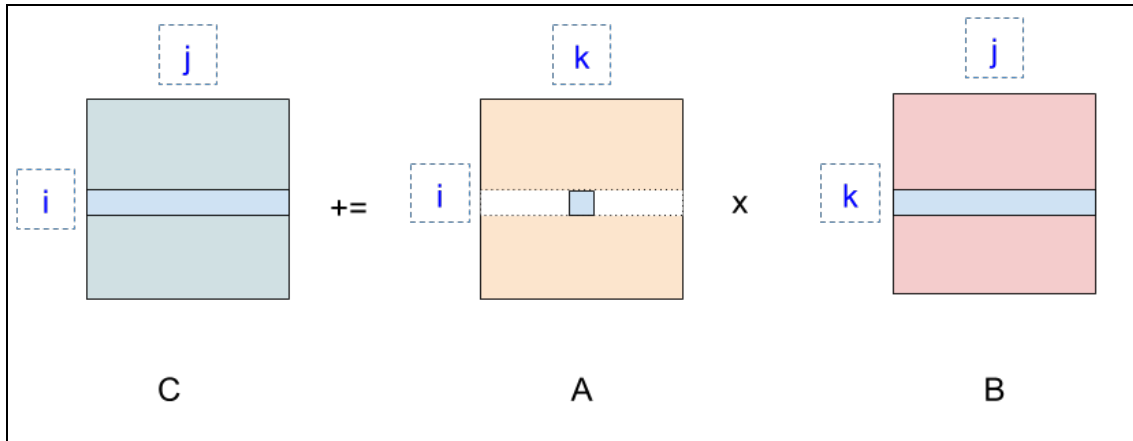


Figura 3.1 Visualización del recorrido en índices de las matrices

En el *Programa 3.2* podemos ver el algoritmo en sí.

```
void MxM ( int filas, int columnas, int kcolumnas,
          float C[filas][columnas],
          float A[filas][kcolumnas],
          float B[kcolumnas][columnas])
{
    __assume_aligned (C,64);
    __assume_aligned (A,64);
    __assume_aligned (B,64);

    int i,j,k;

    for (i=0;i<filas;i++)
    {
        for (k=0;k<kcolumnas;k++)
        {
            for (j=0;j<columnas;j++)
                C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

Programa 3.2 Algoritmo MxM de intercambio de bucles

3.1.3 Algoritmo de “blocking”

El algoritmo de “blocking” consiste en subdividir las matrices que se quieren multiplicar en submatrices más pequeñas, de forma que se acceda repetidamente a dichas submatrices aumentando la localidad. En otras palabras, dichas submatrices acabarán en caché, por lo que los accesos a ellas en memoria serán mucho más rápidos. Las submatrices se tratan como en una multiplicación de matrices usual.

En la *Figura 3.2* se pretende ilustrar el algoritmo.

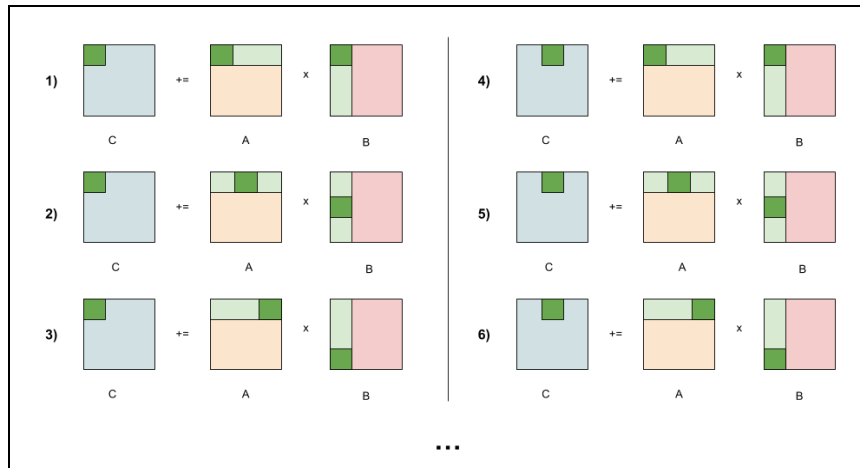


Figura 3.2 Algoritmo $M \times M$ “blocking”

En el *Programa 3.3* se muestra el algoritmo en sí.

```
void MxM_bloq ( int filas, int columnas, int kcolumnas,
               float C[filas][columnas],
               float A[filas][kcolumnas],
               float B[kcolumnas][columnas])
{
    __assume_aligned (C, 64);
    __assume_aligned (A, 64);
    __assume_aligned (B, 64);
    int wi, wj, wk;
    int i, j, k;
    float temp;

    for (wi=0; wi<filas; wi+=32) {
        for (wj=0; wj<columnas; wj+=32) {
            for (wk=0; wk<kcolumnas; wk+=32) {
                for (i=wi; i<wi+32; i++) {
                    for (k=wk; k<wk+32; k++) {
                        temp = A[i][k];
                        for (j=wj; j<wj+32; j++)
                            C[i][j] += temp * B[k][j];
                    }
                }
            }
        }
    }
}
```

manejo de submatrices

multiplicación con intercambio (i,k,j)

Programa 3.3 Algoritmo $M \times M$ “blocking”

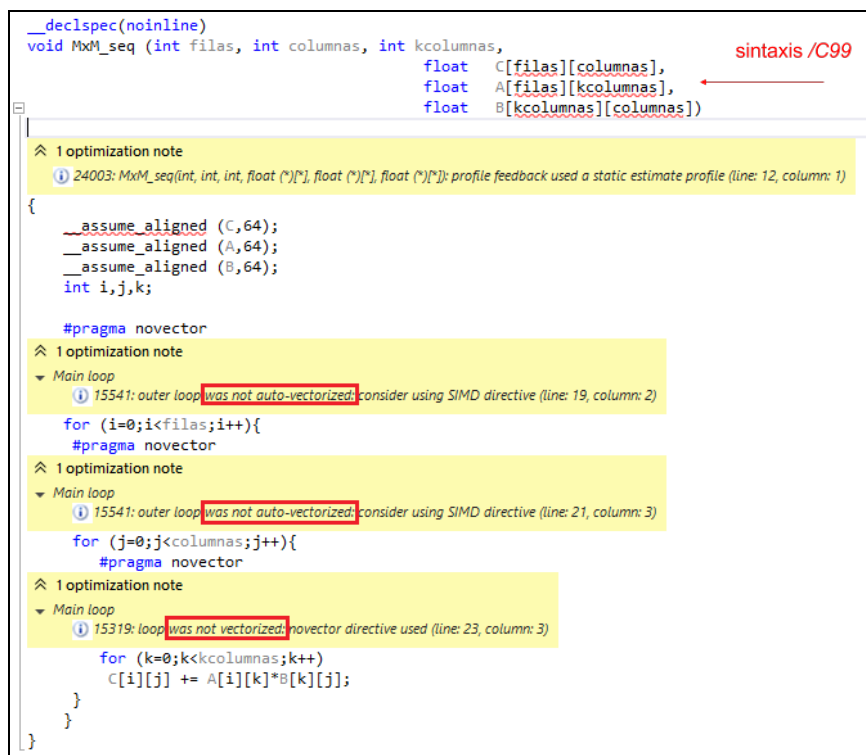
3.2 Auto-optimización

Una vez presentados los algoritmos y sabiendo cómo se configura el entorno *Microsoft Visual Studio*, pasamos a analizar las optimizaciones que realiza el compilador de Intel®.

3.2.1 Reportes del compilador ICC

En primer lugar queremos verificar a grandes rasgos qué optimizaciones ha realizado el compilador con nuestros algoritmos. Para ello hemos utilizado los *flags* de compilación “*Optimization Diagnostic Level: Level 5*” y “*Guided Auto Parallelism Analysis: Extreme*” (recuérdese: *Propiedades del proyecto* → *Propiedades de configuración*).

En primer lugar hemos verificado que el algoritmo secuencial no haya recibido ninguna optimización. Los reportes correspondientes al programa en secuencial son los que podemos ver en la *Figura 3.3*.



```
__declspec(noinline)
void MxM_seq (int filas, int columnas, int kcolumnas,
             float C[filas][columnas],
             float A[filas][kcolumnas],
             float B[kcolumnas][columnas])
{
    __assume_aligned (C,64);
    __assume_aligned (A,64);
    __assume_aligned (B,64);
    int i,j,k;

    #pragma novector
    for (i=0;i<filas;i++){
        #pragma novector
        for (j=0;j<columnas;j++){
            #pragma novector
            for (k=0;k<kcolumnas;k++){
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

sintaxis /C99

1 optimization note
24003: MxM_seq(int, int, int, float (*)[*], float (*)[*], float (*)[*]); profile feedback used a static estimate profile (line: 12, column: 1)

1 optimization note
Main loop
15541: outer loop was not auto-vectorized; consider using SIMD directive (line: 19, column: 2)

1 optimization note
Main loop
15541: outer loop was not auto-vectorized; consider using SIMD directive (line: 21, column: 3)

1 optimization note
Main loop
15319: loop was not vectorized; novector directive used (line: 23, column: 3)

Figura 3.3 Algoritmo secuencial – reportes ICC

Como podemos ver, ninguno de los bucles ha sido vectorizado gracias a la cláusula utilizada. Además, el compilador no da información sobre la paralelización, por lo que no ha sido efectuada (también, porque no se ha utilizado el *flag* “*Parallelization*” del compilador para este fichero).

Pasamos a analizar los cambios que ha realizado el compilador en nuestro *algoritmo de intercambio de bucles* ($i, j, k \rightarrow i, k, j$). Los reportes correspondientes a este algoritmo los podemos ver en la *Figura 3.4*.

```

int i,j,k;

^ 2 optimization notes
- Main loop
  17109: LOOP WAS AUTO-PARALLELIZED (line: 18, column: 2)
  17101: parallel loop shared={} private={} firstprivate={k j B A C i} lastprivate={} firstlastprivate={} reduction={} (line: 18, column: 2)
  15542: loop was not vectorized: inner loop was already vectorized (line: 18, column: 2)
  25015: Estimate of max trip count of loop=4096 (line: 18, column: 2)
- Main loop
  15542: loop was not vectorized: inner loop was already vectorized (line: 18, column: 2)
  25015: Estimate of max trip count of loop=4096 (line: 18, column: 2)
for (i=0;i<filas;i++){
^ 2 optimization notes
- Main loop
  15542: loop was not vectorized: inner loop was already vectorized (line: 19, column: 3)
  25015: Estimate of max trip count of loop=4096 (line: 19, column: 3)
- Main loop
  17104: loop was not parallelized: existence of parallel dependence (line: 19, column: 3)
  17106: parallel dependence: assumed OUTPUT dependence between C[i][j] (2:1:5) and C[i][j] (2:1:5) (line: 19, column: 3)
  17106: parallel dependence: assumed OUTPUT dependence between C[i][j] (2:1:5) and C[i][j] (2:1:5) (line: 19, column: 3)
  15542: loop was not vectorized: inner loop was already vectorized (line: 19, column: 3)
  25015: Estimate of max trip count of loop=4096 (line: 19, column: 3)
for (k=0;k<kcolumnas;k++){
^ 2 optimization notes
- Main loop
  15475: --- begin vector cost summary --- (line: 20, column: 4)
  15476: scalar cost: 9 (line: 20, column: 4)
  15477: vector cost: 0.870 (line: 20, column: 4)
  15478: estimated potential speedup: 10.270 (line: 20, column: 4)
  15488: --- end vector cost summary --- (line: 20, column: 4)
  25015: Estimate of max trip count of loop=128 (line: 20, column: 4)
- Main loop
  17107: loop was not parallelized: inner loop (line: 20, column: 4)
  15475: --- begin vector cost summary --- (line: 20, column: 4)
  15476: scalar cost: 9 (line: 20, column: 4)
for (j=0;j<columnas;j++)
  C[i][j] += A[i][k]*B[k][j];
}
}
  
```

Figura 3.4 Algoritmo de intercambio de bucles – reportes ICC

El compilador nos indica las siguientes optimizaciones:

- 1 El bucle exterior ha sido *autoparalelizado*.
- 2 El bucle principal (interior) ha sido *autovectorizado*.
- 3 El speed-up estimado de la *autovectorización* es de 10.27.

También nos da indicaciones de por qué no ha paralelizado los demás bucles, como por ejemplo problemas de dependencias.

Finalmente analizaremos los cambios que ha realizado el compilador en nuestro *algoritmo de "blocking"*. Los reportes correspondientes a este algoritmo los podemos ver en la *Figura 3.5*.

```

int wi,wj,wk;
int i,j,k;
float temp;

1 optimization note
Main loop
25101: Loop Interchange not done due to: Original Order seems proper (line: 40, column: 2)
25452: Original Order found to be proper, but by a close margin (line: 40, column: 2)
17104: loop was not parallelized: existence of parallel dependence (line: 40, column: 2)
17106: parallel dependence: assumed OUTPUT dependence between C[i][j] (47:11) and C[i][j] (47:11) (line: 40, column: 2)
17106: parallel dependence: assumed OUTPUT dependence between C[i][j] (47:11) and C[i][j] (47:11) (line: 40, column: 2)
15542: loop was not vectorized: inner loop was already vectorized (line: 40, column: 2)
for (wi=0;wi<filas;wi+=32){

2 optimization notes
Main loop
17109: LOOP WAS AUTO-PARALLELIZED (line: 41, column: 4)
17101: parallel loop shared={ } private={ } firstprivate={ wk i k temp j A B C wj wi } lastprivate={ } firstlastprivate={ } reduction={ } (line: 41, column: 4)
15542: loop was not vectorized: inner loop was already vectorized (line: 41, column: 4)
25015: Estimate of max trip count of loop=128 (line: 41, column: 4)
Main loop
for (wj=0;wj<columnas;wj+=32){

2 optimization notes
Main loop
15542: loop was not vectorized: inner loop was already vectorized (line: 42, column: 6)
25015: Estimate of max trip count of loop=128 (line: 42, column: 6)
Main loop
for (wk=0;wk<kcolumnas;wk+=32){

2 optimization notes
Main loop
25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due to other Compiler Transformations) (line: 43, column: 7)
25452: Original Order found to be proper, but by a close margin (line: 43, column: 7)
15542: loop was not vectorized: inner loop was already vectorized (line: 43, column: 7)
25015: Estimate of max trip count of loop=32 (line: 43, column: 7)
Main loop
for (i=wi;i<wi+32;i++){

2 optimization notes
Main loop
15542: loop was not vectorized: inner loop was already vectorized (line: 44, column: 9)
25015: Estimate of max trip count of loop=32 (line: 44, column: 9)
Main loop
for (k=wk;k<wk+32;k++){
temp = A[i][k];

2 optimization notes
Main loop
15475: --- begin vector cost summary --- (line: 46, column: 8)
15476: scalar cost: 8 (line: 46, column: 8)
15477: vector cost: 0.870 (line: 46, column: 8)
15478: estimated potential speedup: 8.000 (line: 46, column: 8)
15488: --- end vector cost summary --- (line: 46, column: 8)
Main loop
for (j=wj;j<wj+32;j++){
C[i][j] += temp*B[k][j];
}

```

Figura 3.5 Algoritmo de blocking – reportes ICC

El compilador nos indica las siguientes optimizaciones:

- 1 No ha realizado intercambio de bucles ya que el orden tal y como se ha diseñado en un principio es correcto.
- 2 El segundo bucle ha sido *autoparalelizado*.
- 3 El bucle principal (interior) ha sido *autovectorizado*.
- 4 El speed-up potencial de la *autovectorización* es de 8.0.

3.2.2 Intel® VTune Amplifier XE – Hotspot analysis

A continuación hemos utilizado la herramienta *Intel® VTune Amplifier XE* para ver los “hotspot” (dónde se concentra el mayor tiempo de la ejecución) de nuestro programa.

En la *Figura 3.6* podemos ver el resultado obtenido.

Function / Call Stack	Effective Time by Utilization				Spin Time				Overhead Time			
	Idle	Poor	Ok	Over	Imbalance or Serial Spinnin	Lock Contention	Other	Creation	Schedulino	Reduction	Atomics	
MxM_seq	34.826s				0s	0s	0s	0s	0s	0s	0s	
MxM	2.819s				0s	0s	0s	0s	0s	0s	0s	
MxM_bloq	2.444s				0s	0s	0s	0s	0s	0s	0s	
func@0x1012af0d	0.496s				0s	0s	0s	0s	0s	0s	0s	

Figura 3.6 Intel® VTune Amplifier XE – Algoritmos BLAS3

Podemos ver que el mayor tiempo de nuestro programa se concentra en el algoritmo secuencial, como era de esperar. A continuación están el *algoritmo de intercambio de bucles*, el *algoritmo de “blocking”* y la rutina MKL (*func@0x1012af0d*, el compilador no muestra el nombre explícito de la rutina MKL dado que se trata de una rutina propietaria de *Intel®*).

A partir de los desgloses del tiempo de cada una de las funciones (haciendo 'clic' en cada función y viendo la información que nos proporciona la herramienta; tiempos *secuencial*, *paralelo* y *vectorial*) podríamos volver a aplicar la *Ley de Amdahl* tal y como hemos explicado en el apartado 2.2.2.1 de la memoria).

3.2.3 Intel® Advisor

La herramienta *Intel® Advisor* nos da información más detallada de las optimizaciones realizadas. En la *Figura 3.7* podemos ver la información que proporciona.

Function Call Sites and Loops	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops	Instruction Set Analysis	Advanced	Location
MxM	0.000s	6.109s	Function			Inserts	Float32 8	Funciones.cpp:12
MxM_MKL	0.000s	1.076s	Function				0	Funciones.cpp:60
MxM_bloq	0.000s	5.152s	Function			Inserts; Shuffles	Float32 8	Funciones.cpp:32
MxM_seq	0.000s	40.276s	Function				5	Funcion_seq.cpp:12
[MKL BLAS]@avx2_sgemv_copybn	0.000s	0.010s	Function				0	
[MKL BLAS]@avx2_sgemm_par	0.000s	0.971s	Function				2	
[MKL BLAS]@get_kernel_api_version	0.000s	0.184s	Function				0	
[loop in MxM at Funciones.cpp:18]	0.000s	3.050s	Scalar Versions	1 inner loop was alr...			8	Funciones.cpp:18
[loop in MxM at Funciones.cpp:19]	0.039s	3.050s	Scalar	inner loop was alre...		Inserts	Float32 8	Funciones.cpp:19
[loop in MxM at Funciones.cpp:20]	3.010s	3.010s	Vectorized (Body)		AVX	100%	10,24x 8	Float32 8
[loop in MxM_bloq at Funciones.cpp:40]	0.000s	2.606s	Scalar	inner loop was alre...			8	Funciones.cpp:40
[loop in MxM_bloq at Funciones.cpp:41]	0.000s	2.546s	Scalar	inner loop was alre...			8	Funciones.cpp:41
[loop in MxM_bloq at Funciones.cpp:42]	0.000s	2.546s	Scalar	inner loop was alre...			8	Funciones.cpp:42
[loop in MxM_bloq at Funciones.cpp:43]	0.236s	2.546s	Scalar	inner loop was alre...			8	Funciones.cpp:43
[loop in MxM_bloq at Funciones.cpp:44]	2.310s	2.310s	Scalar	inner loop was alre...		Inserts	Float32 8	Funciones.cpp:44
[loop in MxM_bloq at Funciones.cpp:45]	n/a	n/a	Vectorized (Bod...		100%	8,00x 8	0	Contains Com... Funciones.cpp:45
[loop in MxM_seq at Funcion_seq.cpp:19]	0.000s	40.276s	Scalar	outer loop was not ...			5	Funcion_seq.cpp:19
[loop in MxM_seq at Funcion_seq.cpp:21]	0.018s	40.276s	Scalar	outer loop was not ...			5	Funcion_seq.cpp:21
[loop in MxM_seq at Funcion_seq.cpp:23]	40.258s	40.258s	Scalar	movector directive ...			5	Funcion_seq.cpp:23

Figura 3.7 Intel® Advisor – Algoritmos BLAS3

En primer lugar 1 podemos ver el **tiempo total de ejecución** de cada una de las rutinas. El tiempo que se muestra aquí concuerda proporcionalmente con el tiempo que se nos ha mostrado con la anterior herramienta. Cabe destacar que aquí sí podemos ver la rutina MKL (*MxM_MKL*), dado que se trata de una subrutina nuestra, que únicamente llama a la subrutina de *MKL*, no se trata de la rutina de *MKL* en sí.

Después **2** podemos ver en detalle todas las optimizaciones que ha realizado el compilador con nuestro **algoritmo de intercambio de bucles (MxM)**. Se ha realizado vectorización con el conjunto de instrucciones AVX, con una eficiencia estimada del 100% y un *speed-up* de 10.24. La longitud del vector es de ocho (8 *float* de 32 bits en vectores de 256 bits). Además ha efectuado *loop-unrolling* de cuatro.

Finalmente **3** podemos ver en detalle las optimizaciones realizadas sobre el **algoritmo de "blocking" (MxM_bloq)**. Podemos suponer que ha efectuado la vectorización con el conjunto de instrucciones AVX dado que la *eficiencia estimada* es del 100% y el *speed-up* de ocho. Además, el programa (aunque esté cortado en la figura) nos dice que ha desenrollado completamente el bucle interior ("*Contains Completely Unrolled Loop by 4, Vectorized (Body) Completely Unrolled*"). Dado que hemos hecho submatrices de 32 elementos, al desenrollar el bucle cuatro veces ($32/4 = 8$) y vectorizar con AVX (vector de 8 elementos), solo es necesaria una iteración, por eso el programa destaca lo de "desenrollado completamente".

También se ha utilizado nuevamente la herramienta de Intel® Advisor para recibir información detallada de la **paralelización de los algoritmos**. La manera de obtener esta información ya fue explicada en las pruebas del capítulo segundo.

En primer lugar analizamos el **algoritmo de intercambio de bucles**. Antes de realizar ninguna optimización el compilador nos muestra la información que podemos ver en la *Figura 3.8*.

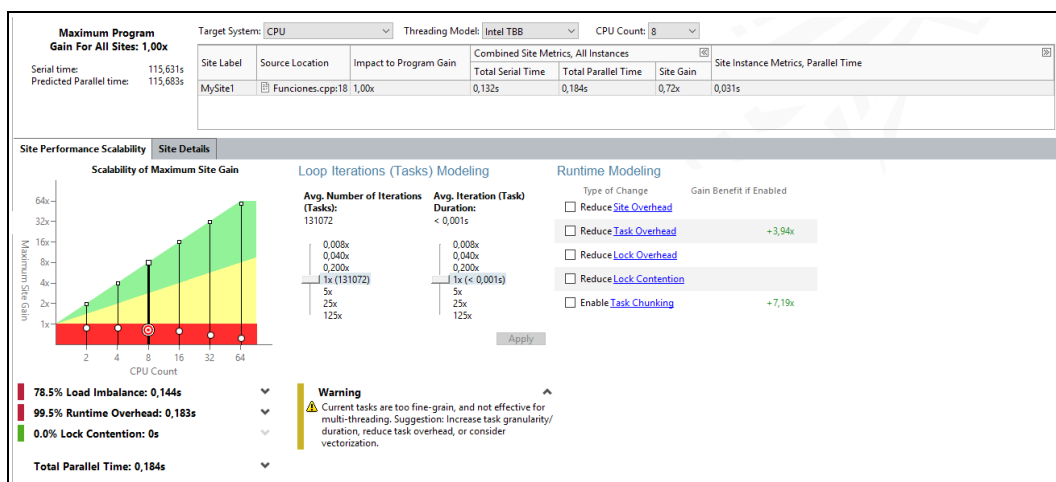


Figura 3.8 Análisis paralelización algoritmo intercambio de bucles - antes

Tras reducir el *overhead* de crear la región paralela (*site overhead*) y tras habilitar el *chunk* de las tareas (*task chunking*, método de reparto de tareas) el compilador nos muestra la información que podemos ver en la *Figura 3.9*.

El tiempo total de la región paralela disminuye de 0.184s a 0.017s. Además los *overhead* se han visto reducidos significativamente y el balanceo de carga es óptimo. Según el gráfico con estas optimizaciones podemos acercarnos a los 8 de *speed-up*. Cabe destacar que se ha seleccionado como número de CPU's ocho, dado que nuestro procesador Intel® Core i7 6700K cuenta con 4 núcleos físicos con hyperthreading de 2. Además, las pruebas han sido realizadas para el modelo de paralelismo de Intel® (Intel® TBB).

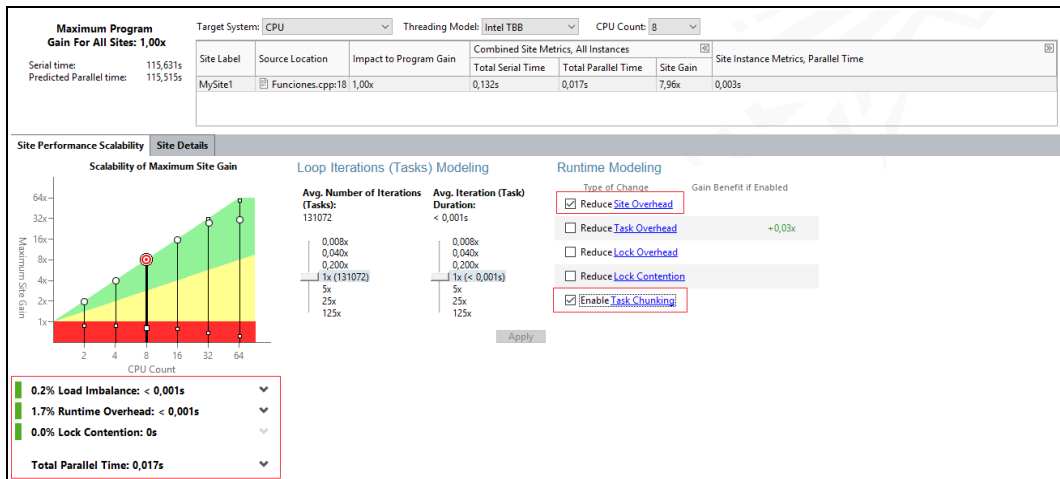


Figura 3.9 Análisis paralelización algoritmo intercambio de bucles - después

Hemos realizado el mismo análisis con el algoritmo de “blocking”. En la Figura 3.10 podemos ver la información del compilador sin realizar optimizaciones. En la Figura 3.11 podemos ver lo que alcanzaríamos hipotéticamente aplicando las mismas optimizaciones: “Reduce Site Overhead” y “Enable Task Chunking”.

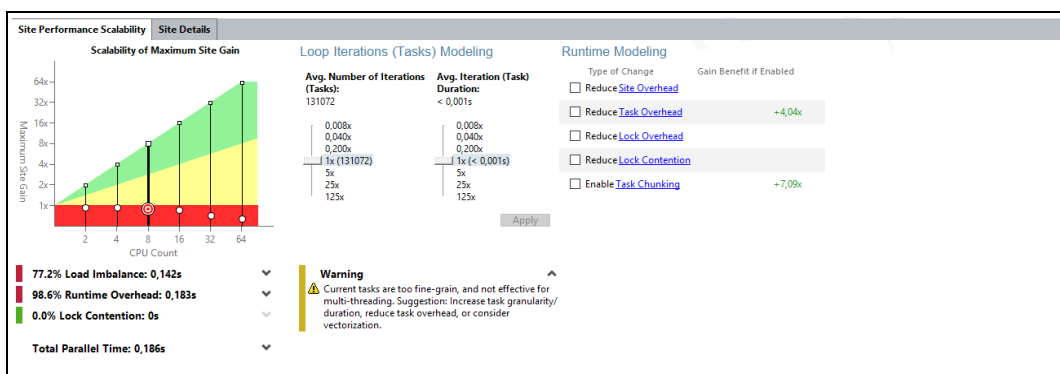


Figura 3.10 Análisis paralelización algoritmo “blocking” - antes

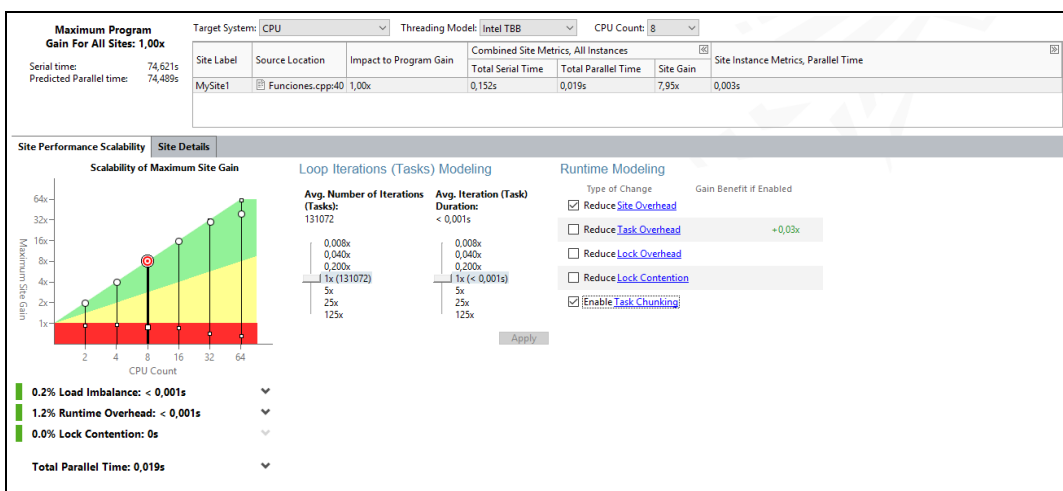


Figura 3.11 Análisis paralelización algoritmo “blocking” - después

3.2.4 Intel® Roofline

El último análisis realizado ha sido mediante la herramienta *Intel® Roofline*. Podemos ver el gráfico *Roofline* correspondiente a nuestros algoritmos de *BLAS de nivel 3* en la *Figura 3.12*.

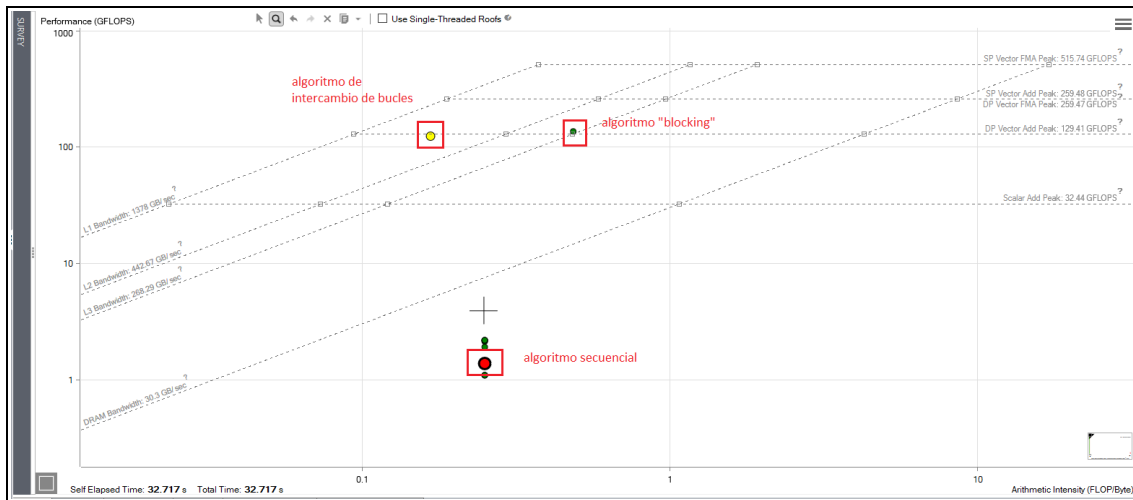


Figura 3.12 Intel® Roofline - BLAS3

Podemos ver que el *algoritmo de intercambio de bucles* está acotado por caché de nivel 1, con un ancho de banda de 1378 GB/s (*L1 Bandwidth 1378 GB/s*). Es el mejor de los casos. Por otro lado, está acotado por las instrucciones vectoriales de doble precisión (*DP Vector Add Peak 129.41 GFLOPS*) que teóricamente no ofrecen más de 129.41 GFLOPS.

El *algoritmo de "blocking"* está acotado por caché de nivel 2, con un ancho de banda de 442.67 GB/s (*L2 Bandwidth 442.67 GB/s*). En cuanto a GFLOPS, está acotado por las instrucciones vectoriales FMA (*fused multiply-add*) de precisión doble (*DP Vector FMA Peak 259.47 GFLOPS*). Estas instrucciones vectoriales permiten multiplicar y sumar en un mismo ciclo.

Las diferencias entre el algoritmo de *intercambio de bucles* y el algoritmo de *"blocking"* se deben a que el compilador realiza *loop unrolling* y *prefetch* en el primero, mientras que nuestro algoritmo de *"blocking"* no lo permite.

3.3 Librerías MKL

También se ha implementado la rutina de BLAS de nivel 3 ($C \leftarrow A B + c$) mediante la función MKL correspondiente, `cblas_sgemv` (referencia: <https://software.intel.com/en-us/mkl-developer-reference-c-cblas-gemm>).

Lo único que ha sido necesario es invocar a la función con los parámetros que se especifican (Programa 3.4).

```
void MxM_MKL ( int filas, int columnas, int kcolumnas,
               float C[filas][columnas],
               float A[filas][kcolumnas],
               float B[kcolumnas][columnas])
{
    __assume_aligned (C,64);
    __assume_aligned (A,64);
    __assume_aligned (B,64);

    cblas_sgemv(CblasRowMajor, CblasNoTrans, CblasNoTrans,
               filas, columnas, kcolumnas, 1.0, &A[0][0],
               kcolumnas, &B[0][0], columnas, 1.0, &C[0][0], columnas);
}
```

Programa 3.4 Función MKL GEMM - BLAS3

3.4 Resultados

Finalmente se expondrá una tabla de resultados para las pruebas que han sido realizadas. Se han realizado las pruebas con matrices cuadradas de tamaños 128, 256, 512, ..., 4096.

Tamaño	Ticks - secuencial	Speed-up - intercambio de bucles	Speed-up - "blocking"	Speed-up - función MKL
128	8,272.50	56.58	15.95	56.95
256	77,576.10	90.47	15.70	91.63
512	834,940.13	91.86	19.70	114.22
1024	7,437,126.50	109.69	111.41	127.74
2048	165,264,704.00	153.49	154.49	364.81
4096	1,746,901,760.00	147.22	231.28	651.25

Tabla 3.1 Resultados (ticks) de las pruebas realizadas - BLAS3

Podemos apreciar que en tamaños de matrices pequeños (menores o iguales que 512) el algoritmo de intercambio de bucles y la función MKL son muy parejos. Conforme aumenta el

tamaño de las matrices el *algoritmo de "blocking"* mejora su rendimiento frente al *algoritmo de intercambio de bucles*, que incluso con matrices de tamaño de 1024 se acerca al *speed-up* de la rutina de *MKL*. Sin embargo, con tamaños de matrices muy grandes, *MKL* obtiene un rendimiento extraordinario frente a cualquiera de los otros algoritmos.

En la *Tabla 3.2* se presentan los criterios de uso como acostumbramos a hacer al final de cada análisis de resultados. Los algoritmos de una fila se deberán utilizar frente a los algoritmos de la columna siempre y cuando se cumpla la condición establecida en la casilla correspondiente de la tabla. En caso de mostrar un 'tic', la condición será "siempre se usa el algoritmo de la fila respecto a la de la columna", en caso de mostrar una 'cruz' la condición será "nunca se usa el algoritmo de la fila respecto a la de la columna".

	Secuencial	Intercambio de bucles	"Blocking"	MKL
Secuencial	-	✗	✗	✗
Intercambio de bucles	✓	-	Cuando el tamaño de la matriz es inferior o igual a 1024 x 1024	✗
"Blocking"	✓	Cuando el tamaño de la matriz es superior o igual a 1024 x 1024	-	✗
MKL	✓	✓	✓	-

Tabla 3.2 Criterios de uso de los algoritmos BLAS3

4

GPU - NVIDIA CUDA

En este capítulo se explicará la programación en *NVIDIA CUDA* de los dos algoritmos principales tratados a lo largo del proyecto. Estos son: la *función genérica* de *BLAS2* y la función de *GEMM* de *BLAS3*. Cada una de estas funciones ha sido desarrollada de dos formas distintas: una de forma *convencional* y otra aprovechando la *localidad*, de forma similar al algoritmo de “blocking” explicado en el capítulo anterior. En cada apartado (*BLAS de nivel 2* y *BLAS de nivel 3*) se explicarán más detalladamente cada uno de los desarrollos. Se concluirá el capítulo con los resultados obtenidos y una comparativa entre ambas rutinas en *CUDA* (*Figura 4.1*).

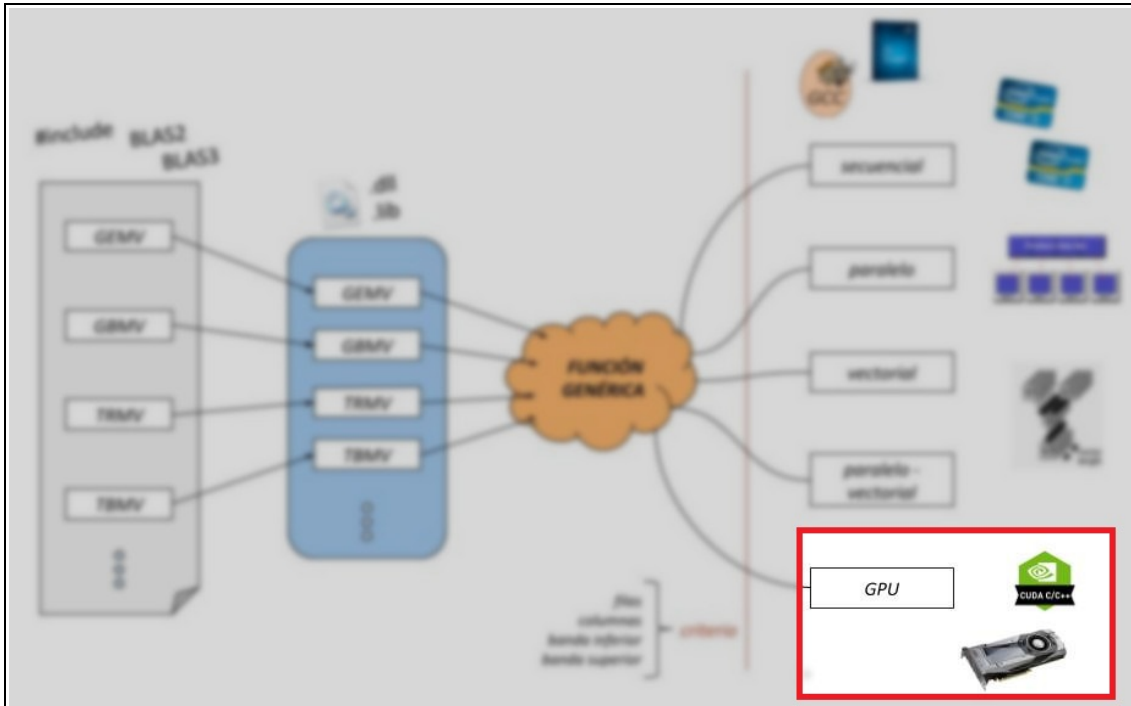


Figura 4.1 Esquema general – CUDA

4.1 BLAS de nivel 2

Recordemos lo implementado para la especificación *BLAS de nivel 2*:

$$y \leftarrow A \times x + y$$

Dado que la *GPU* dispone de muchos *threads* de los que podemos hacer uso, aunque las matrices no sean densas hemos optado por realizar la multiplicación completa, puesto que las diferencias en rendimiento serían inapreciables.

Comenzaremos explicando la forma *convencional* en la que hemos implementado el algoritmo de *BLAS de nivel 2* mediante *CUDA*.

Tal y como se explica en el primer capítulo de la memoria (*apartado 1.3.1.4*), se reserva memoria para las variables en la *GPU*, se transfieren de la memoria de la *CPU* a la memoria de la *GPU* los datos necesarios y se preparan los parámetros de ejecución (en este caso el tamaño en *threads* de los *bloques* es de 1024 y el tamaño del *grid* es de $(1024 + \text{filas} - 1) / 1024$; habrá al menos un *thread* por fila de la matriz, puesto que cada *thread* se encargará de calcular un elemento del vector de resultados '*y*', es decir, cada *thread* efectuará un producto escalar). Tras definir estos parámetros se llamará al *kernel* de *CUDA*, es decir, a nuestra función de *CUDA* (*Programa 4.1*).


```

__global__ void mAxy_1( float * y, float * a, float * x, int Width, int
nfilas)
{
    int fil = blockIdx.x * blockDim.x + threadIdx.x; 1
    if (fil < nfilas) 2
    {
        float sum = 0.0f;
        for (int k = 0; k < Width; k++)
        { sum = sum + a[fil*Width+k] * x[k]; }
        y[fil] += sum;
    }
}

```

Programa 4.1 Función básica (convencional) CUDA BLAS2

Para poder explicar mejor el funcionamiento de este algoritmo se presenta un ejemplo explicativo en la Figura 4.2.

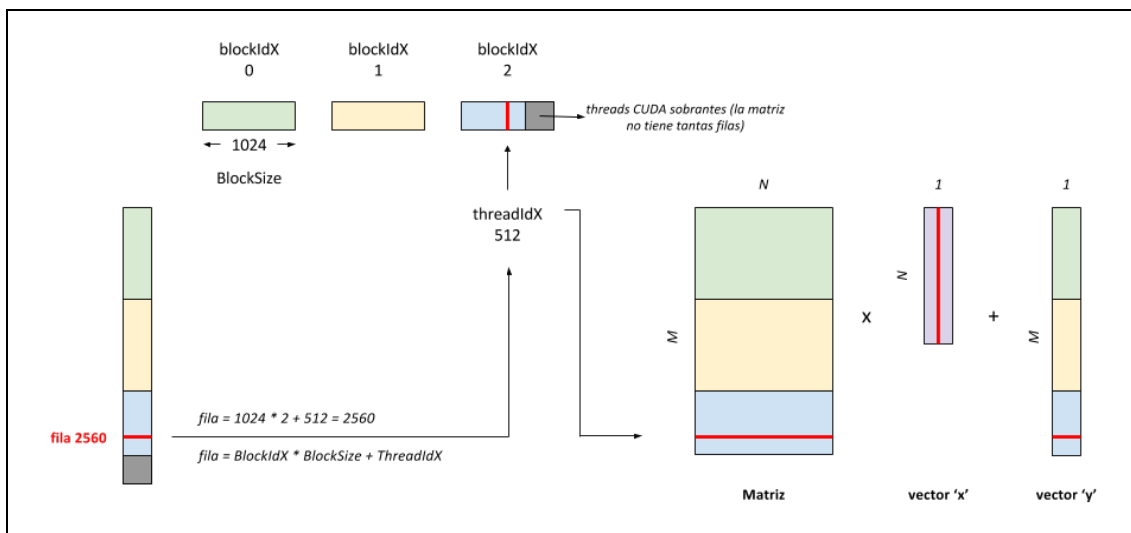


Figura 4.2 Explicación función convencional CUDA BLAS2

Como podemos ver en el ejemplo, disponemos de una matriz de tamaño $M \times N$, siendo M 2730. Por tanto, nuestra estructura en *CUDA* dispone de tres bloques de 1024 *threads*, un total de 3072 *threads* (más de los que filas hay, por tanto habrá $3072 - 2730 = 342$ *threads* “ociosos”; gris en la figura). Cada *thread* en *CUDA* dispone de varios identificadores, en este caso los más relevantes son: *BlockIdx* (en qué bloque me encuentro) y *ThreadIdx* (dentro de ese bloque, qué *thread* soy). Por tanto, supongamos que somos el *thread* situado en el bloque 2 con identificador 512. La fila que nos tocaría procesar sería la $1024 * 2 + 512 = 2560$ ($BlockSize * BlockIdx + ThreadIdx$); en el programa la variable *fil* **1**. Si somos, por ejemplo, el *thread* 3000 no se nos asigna ninguna fila; por ello la condición “if (*fil* < *nfilas*)” **2** del programa. Finalmente, cada *thread* calcula su producto escalar y lo suma a la posición ‘y’ correspondiente.

A continuación expondremos el método de “*blocking*” (o *block tiled*) implementado mediante *CUDA* para *BLAS* de nivel 2.

De nuevo, los primeros pasos son los de reserva de memoria para las variables en la GPU, transferir de la memoria de la CPU a la memoria de la GPU los datos necesarios y preparar los parámetros de ejecución (estos parámetros son los mismos que los anteriores). Tras definirlos se llamará al *kernel* de *CUDA* (Programa 4.2).

```

__global__ void mAxy_2( float * vY, float * MA, float * vX, int Width,
                      int nfilas)
{
    // Thread index
    int fil = blockIdx.x * blockDim.x + threadIdx.x;

    int xBegin = Width * BLOCK_SIZE * blockIdx.x;
    int xEnd   = xBegin + Width - 1;
    int xStep  = BLOCK_SIZE;

    float Pvalue = 0;

    for (int x = xBegin; x <= xEnd; x += xStep)
    {
        __shared__ float vXs[BLOCK_SIZE]; 2
        if (x+threadIdx.x < Width)
        {
            vXs[threadIdx.x]=vX[x+threadIdx.x]; 1
        }
        __syncthreads(); 3
        for (int k = 0; k < BLOCK_SIZE; k++)
        {
            float Melemen=MA[fil*Width+x+k]; 4
            float Nelemen=vXs[k];
            Pvalue += ( Melemen * Nelemen );
        }
        __syncthreads(); 5
    }
    vY[fil] = Pvalue; 6
}

```

Programa 4.2 Función “*blocking*” *CUDA* *BLAS*2

Presentaremos el funcionamiento del algoritmo mediante una figura (Figura 4.3). En este caso la matriz es de tamaño $M \times N$, siendo M aproximadamente 1900. Si suponemos que somos el *thread* 617 dentro del bloque 1, nuestra fila asignada sería la 1641 (calculada de la misma forma que en el ejemplo anterior). Ahora la idea es que cada bloque de *threads* lea tantos elementos de ‘x’ como *threads* hay en el bloque. En este caso serían 1900, que además coincide con la longitud del vector de resultados ‘y’ (en concreto, el bloque 0 leería los primeros 1024 elementos de ‘x’ y el bloque 1 leería los 876 elementos siguientes) 1. Estas lecturas se guardan en memoria compartida (memoria caché) de cada bloque de *threads* 2. Todos los *threads* se sincronizan para asegurar que la memoria *shared* se ha rellenado 3.

A continuación, cada *thread* hace su producto escalar (en esta ocasión solamente con un fragmento de 'x', no con el vector entero) **4**. Tras hacer el primer producto escalar, se avanza al siguiente bloque y se repite el proceso hasta haber obtenido el producto escalar de su fila con el vector 'x' completo. Para asegurar que se ha terminado el producto interno, todos los *threads* vuelven a sincronizarse **5**. El resultado se almacena en la fila del vector de resultado 'y' correspondiente **6**.

Al avanzar por bloques y aprovechar la lectura y la memoria compartida de todos los *threads* de cada bloque, se hace uso de la localidad y los accesos a memoria son mucho más eficientes.

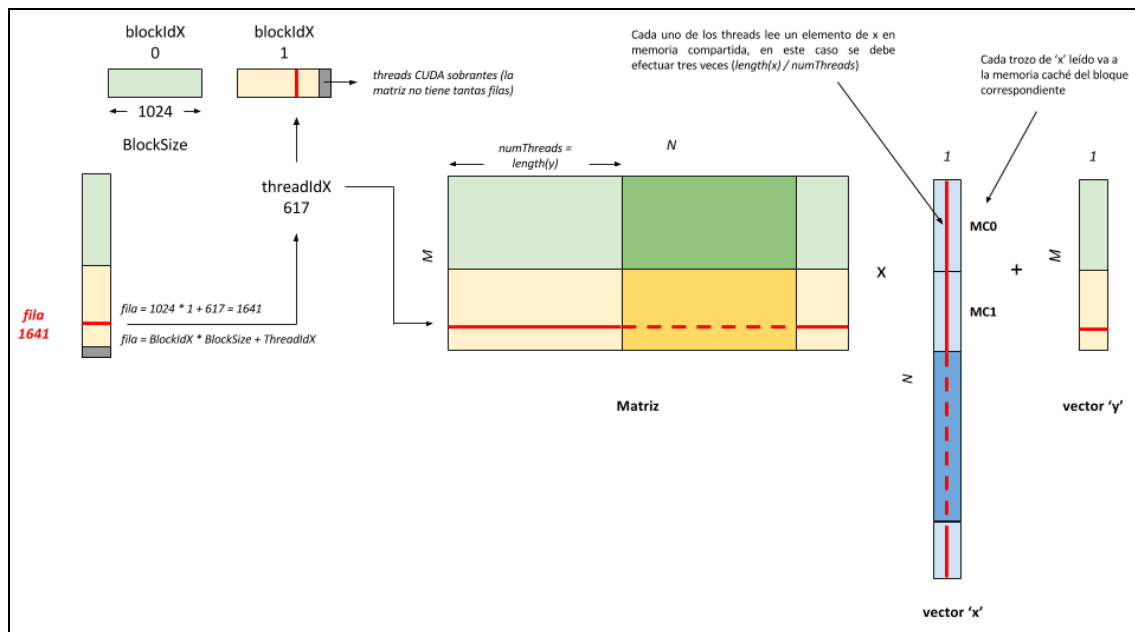


Figura 4.3 Explicación función "blocking" CUDA BLAS2

4.2 BLAS de nivel 3

Recordemos lo implementado para la especificación *BLAS de nivel 3*:

$$C \leftarrow A B + C$$

Comenzaremos explicando la forma *convencional* en la que hemos implementado el algoritmo de *BLAS de nivel 3* mediante *CUDA*.

En primer lugar se reserva memoria para las variables en la GPU, se transfieren de la memoria de la CPU a la memoria de la GPU los datos necesarios y se preparan los parámetros de ejecución (en este caso el tamaño en *threads* de los *bloques* es de 16 – bidimensional 4x4 – y

el tamaño del $grid(x,y)$ es de $(filas/4, columnas/4)$; cada $thread(i,j)$ será el encargado de calcular el producto escalar entre la fila 'i' de la matriz A ¹ y la columna 'j' de la matriz B ². Tras definir estos parámetros se llamará al *kernel* de CUDA (Programa 4.3).

```

__global__ void mAxAT_1( float *c, float *a, float *b, int Width)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int k = 0; k < Width; k++)
        { sum = sum + a[row*Width+k] * b[k*Width+col]; }
    c[row*Width+col] += sum;
}

```

Programa 4.3 Función básica (convencional) CUDA BLAS3

Explicaremos este programa haciendo uso de la Figura 4.4.

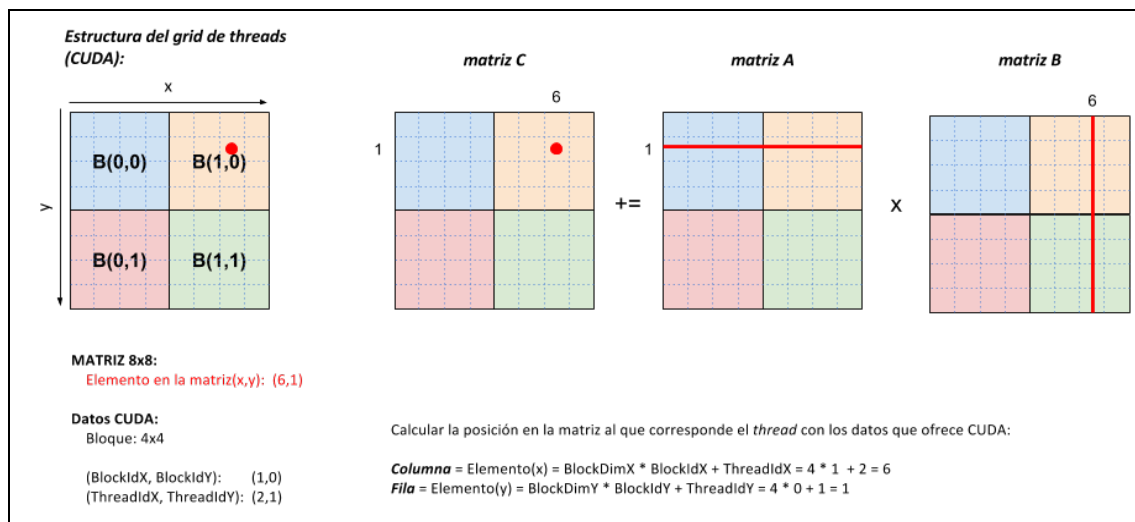


Figura 4.4 Explicación función básica (convencional) CUDA BLAS3

En la imagen, en la parte izquierda, podemos ver la estructura en *grid* de los *threads* de CUDA (es un ejemplo simplificado con *bloques* de 4x4 *threads*, en lugar de 16x16). Supongamos que somos el *thread* (2,1) del *bloque* (1,0). Nuestro elemento de la matriz C a calcular sería el (6,1). En la figura también podemos ver cómo obtener este resultado. Por tanto, seremos el *thread* encargado de calcular el elemento $(i,j) = (1,6)$ de la matriz C. Nuestra labor será efectuar el producto escalar entre la fila 1 de A y la columna 6 de B, y sumar el resultado a la matriz C en la posición $(i,j) = (1,6)$.

A continuación explicaremos el método de “blocking” para este algoritmo de BLAS de nivel 3 mediante CUDA.

Los primeros pasos son similares a los anteriores, reservar memoria, transferir datos, definir parámetros de ejecución (con los mismos datos que en el caso anterior) y llamar al *kernel* (Programa 4.4).

```

__global__ void mAxAT_2( float *C, float *A, float *B, int Width)
{
    // Thread index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int aBegin = Width * BLOCK_SIZE * by;
    int aEnd   = aBegin + Width - 1;
    int aStep  = BLOCK_SIZE;

    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE*Width;

    float Pvalue = 0;

    for (int a = aBegin, b = bBegin; a<=aEnd;a += aStep, b+= bStep) 5
    {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE]; 2
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        As[ty][tx]=A[a+Width*ty+tx]; 1
        Bs[ty][tx]=B[b+Width*ty+tx];

        __syncthreads(); 3
        for (int k = 0; k < BLOCK_SIZE; k++)
        {
            float Melemen=As[ty][k]; 4
            float Nelemen=Bs[k][tx];
            Pvalue += ( Melemen * Nelemen );
        }
        __syncthreads();
    }
    int c=Width*BLOCK_SIZE*by+BLOCK_SIZE*bx; 6
    C[c+ ty * Width + tx] += Pvalue;
}

```

Programa 4.4 Función "blocking" CUDA BLAS3

Para comprender más fácilmente el programa utilizaremos el ejemplo de la *Figura 4.5*. De nuevo, supongamos que somos el *thread* del ejemplo anterior y que se nos haya asignado hacer el producto escalar entre la fila 1 de la matriz A y la columna 6 de la matriz B. Esta vez subdividiremos el problema en bloques de submatrices. El primer paso sería que todos los *threads* del *bloque* (1,0) leyesen conjuntamente los bloques correspondientes (resaltados en la figura) **1**. Cada *thread* leerá un elemento de la matriz A y un elemento de la matriz B y dejará el dato en memoria compartida del bloque al que pertenece (en este caso el *bloque* (1,0)) **2**. Una vez sincronizados, **3** tras haber efectuado todas las lecturas, cada *thread* efectuará el producto escalar que le corresponde, pero solo de las dos submatrices que han sido leídas **4**. Tras guardar el resultado intermedio, se procede a leer el siguiente conjunto de bloques (*Paso 2* en la figura), **5** y efectuar el producto escalar de los siguientes dos trozos de vectores correspondientes. Este proceso se repetirá hasta haber abarcado el producto escalar completo. El resultado se suma a la posición de C correspondiente **6**. Cada *thread* realiza exactamente la misma acción para su fila y su columna asignadas.

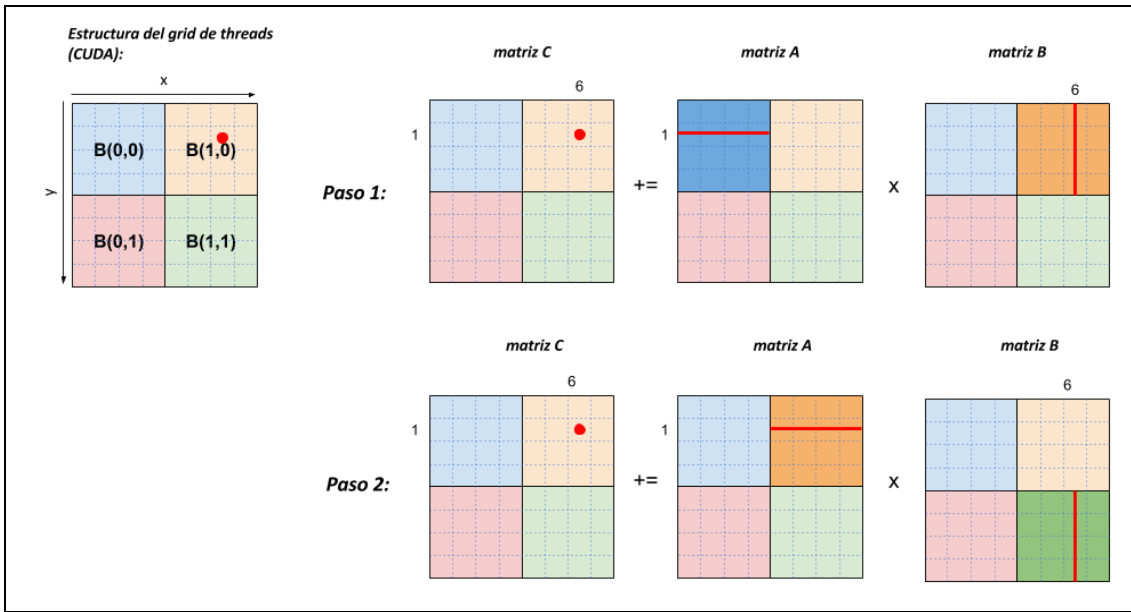


Figura 4.5 Explicación función "blocking" CUDA BLAS3

Este método aprovecha las lecturas simultáneas de todas las submatrices que son necesarias, además de poder acceder posteriormente a los datos a través de memoria caché, y no de memoria principal. Al tratarse de lecturas de dos matrices densas (no solo de un vector como en el caso de *BLAS 2*) en este caso podremos apreciar una mejora en el rendimiento.

4.3 Resultados y comparativa general

Finalmente, para ratificar lo explicado en los apartados anteriores sobre CUDA, se expondrán datos de las pruebas realizadas. Todas las pruebas realizadas han sido con matrices cuadradas.

Empezando por los datos de *CUDA* de *BLAS 2* (Tabla 4.1).

Tamaño	Ticks ICC	Speed-up CUDA convencional	Speed-up Kernel convencional	Speed-up CUDA "blocking"	Speed-up Kernel "blocking"
128	54.54	0.01	2.61	0.01	2.61
256	233.51	0.04	10.97	0.04	11.06
512	977.53	0.14	45.96	0.14	45.94
1024	3,991.16	0.38	180.35	0.38	180.51
2048	16,272.20	0.81	667.71	0.81	670.47
4096	65,091.57	1.10	2024.62	1.10	2034.11

Tabla 4.1 Datos CUDA BLAS2

En cuanto a los *speed-up* totales, incluso con un tamaño elevado (4096) nuestro algoritmo en *CUDA* apenas logra acercarse al programa usual en la *CPU* (i7 6700K). Esto se debe a que el cálculo en sí es mucho más rápido en la *GPU* que en la *CPU* (*columna speed-up kernel en la tabla*) pero la transferencia de los datos (copiar las matrices y los vectores de la memoria de la *CPU* a la *GPU* y copiar el resultado de vuelta) añade demasiado sobrecoste al tiempo de cálculo. Tal es así, que en tamaños pequeños perdemos muchísimo *speed-up* en *CUDA* frente a la *CPU*.

Si, en cambio, el algoritmo reusara la matriz si que merecería la pena utilizar *CUDA*, ya que entonces solamente se tendría en cuenta el tiempo del *kernel* (las transferencia acaban siendo despreciables).

Podemos contrastar estos datos con los obtenidos en *CUDA* de *BLAS3* (*Tabla 4.2*).

Tamaño	Ticks ICC	Speed-up CUDA convencional	Speed-up Kernel convencional	Speed-up CUDA "blocking"	Speed-up Kernel "blocking"
128	16,106.00	12.99	201.33	14.38	402.65
256	138,721.00	65.43	315.28	78.82	1156.01
512	1,176,614.00	193.52	498.57	217.89	1634.19
1024	23,870,274.00	773.00	1269.70	1302.96	4115.56
2048	231,450,112.00	1218.41	1574.92	2552.38	4794.89
4096	1,910,654,336.00	1730.35	2028.12	3985.84	5993.27

Tabla 4.2 Datos *CUDA* *BLAS3*

Los *speed-up* obtenidos en la *GPU* ahora son mucho mayores respecto a los obtenidos mediante la *CPU*. En este caso la diferencia sí es muy grande, dado que el peso del programa reside en la complejidad cúbica de la multiplicación matricial, y el tiempo de transferencia de datos *CPU* → *GPU* y viceversa acaba siendo despreciable.

Además, aquí también podemos ver que nuestro algoritmo de "blocking" obtiene mejores resultados que el algoritmo *convencional*. En *BLAS2* apenas se notaba diferencia entre un método y otro. Al residir el mayor tiempo de ejecución (de *BLAS2*) en la transferencia de datos y no en el cómputo en sí, y ser además el cómputo más liviano que el de matriz por matriz (*BLAS3*), las diferencias entre uno y otro son poco apreciables.

Vamos a concluir el capítulo con una comparativa entre *BLAS2* y *BLAS3* en cuanto a *GFLOPS* (operaciones en coma flotante por segundo) y *ancho de banda* (transferencia de bytes de la *GPU* a la *CPU* y viceversa); *Tabla 4.3*.

Podemos ver que la velocidad de las transferencias es prácticamente la misma (sin demasiadas diferencias) tanto en *BLAS2* como en *BLAS3*. Esto se debe a que el tiempo de transferencia depende únicamente del ancho de banda del *PCI Express* (*Peripheral Component Interconnect Express*) y el tamaño en Bytes de la transferencia. Las de *BLAS3* son algo superiores dado que también hay una mayor cantidad de datos a transferir. Sin embargo, podemos ver que, efectivamente entre ambas complejidades, cuadrática frente a cúbica, donde reside la mayor

diferencia es en los *GFLOPS*. *BLAS3* consigue mucho mayor rendimiento que *BLAS2* en operaciones por segundo, con la misma proporción de tasa de transferencia. Es por ello que en operaciones de este estilo (multiplicación matricial), o en operaciones de mayor orden de complejidad, *CUDA* obtiene mucho mejor rendimiento que las *CPU's* convencionales. La cantidad de datos a procesar es lo suficientemente grande como para poder despreciar el tiempo de transferencia. Si además, como sucede a menudo en procesamiento gráfico, se tiene que operar muchas veces con los mismos datos y se puede ahorrar la transferencia *CPU* → *GPU* y viceversa, la efectividad de las placas gráficas se dispara.

Tamaño	BLAS2 convencional – GFLOPS	BLAS2 convencional – ancho de banda	BLAS2 “blocking” – GFLOPS	BLAS2 “blocking” – ancho de banda	BLAS3 convencional – GFLOPS	BLAS3 convencional – ancho de banda	BLAS3 “blocking” – GFLOPS	BLAS3 “blocking” – ancho de banda
128	9.20 GF	0.05 GB/s	9.22 GF	0.04 GB/s	273.56 GF	0.89 GB/s	449.09 GF	0.95 GB/s
256	36.14 GF	0.18 GB/s	36.43 GF	0.17 GB/s	301.47 GF	2.50 GB/s	1026.00 GF	2.60 GB/s
512	144.71 GF	0.61 GB/s	144.66 GF	0.56 GB/s	457.62 GF	4.52 GB/s	1474.16 GF	3.60 GB/s
1024	556.43 GF	1.59 GB/s	557.00 GF	1.57 GB/s	456.82 GF	5.56 GB/s	1481.17 GF	5.37 GB/s
2048	2020.71 GF	3.29 GB/s	2029.46 GF	3.32 GB/s	467.67 GF	6.24 GB/s	1425.36 GF	6.32 GB/s
4096	6127.20 GF	4.44 GB/s	6155.61 GF	4.50 GB/s	583.63 GF	6.62 GB/s	1724.75 GF	6.69 GB/s

Tabla 4.3 Comparación de ancho de banda y GFLOPS – BLAS2 y 3

Tras lo analizado se han establecido los criterios de uso que podemos ver en la *Tabla 4.4*.

<i>Si disponemos de GPU</i>		
En <i>BLAS2</i>	Si no reutilizamos la estructura de datos	<i>CPU</i>
	Si reutilizamos la estructura de datos	Si el tamaño de la matriz es > 512x512 → <i>GPU</i>
En <i>BLAS3</i>	Si no reutilizamos la estructura de datos	Si el tamaño de la matriz es ≥ 512x512 → <i>GPU</i>
	Si reutilizamos la estructura de datos	<i>GPU</i>

Tabla 4.4 Criterios de uso de la GPU

En caso de que no ejecutáramos en la *GPU*, tendríamos que recurrir al *capítulo 2* de la memoria para el caso de *BLAS2*. Para el caso de *BLAS3* bastaría con utilizar *MKL*.

5

Conclusiones y líneas abiertas

Para definir un criterio de uso consistente ha sido necesario tener en cuenta una gran cantidad de factores. Entre los más importantes se encuentran los **sistemas de cómputo** disponibles, para saber si realmente se pueden acceder a los diferentes recursos, como por ejemplo: la paralelización, la vectorización (*SSE, AVX, ...*), *GPU, ...*

A la hora de diseñar el **algoritmo** también es importante tener en cuenta las **estructuras de datos** con las que se trabaja (*filas, columnas, bandas...*), la **complejidad de los algoritmos** [$O(n^2)$, $O(n^3)$...], la **jerarquía de memorias** (*para aprovechar al máximo el reuso de los datos y minimizar las latencias de acceso*), ...

Además, todo puede resultar más sencillo con las herramientas adecuadas, luego es importante saber con cuáles se puede contar. Por ejemplo, **el compilador**, para saber si tiene la posibilidad de *auto-optimizar* (*autoparalelizar, autovectorizar...*) y si realmente obtiene un rendimiento aceptable. Asimismo, disponer de las **herramientas de análisis** adecuadas (*Intel® VTune, Intel® Advisor, Intel® Roofline...*) también puede facilitar el proceso.

Otro factor a tener en cuenta sería, ¿existe ya una solución a mi problema? ¿Hay alguna librería especializada para lo que quiero desarrollar? En este caso la respuesta ha sido *MKL*.

Todo ello ha sido analizado a lo largo del proyecto para decidir **criterios de uso** consistentes tanto para la **función genérica de BLAS2** desarrollada, como para la función de **multiplicación matricial** (de matrices cuadradas densas) de **BLAS3**.

A continuación se expone un resumen de resultados obtenidos. En primer lugar, el criterio de uso en **BLAS2 (CPU) mediante el compilador GCC (programación explícita)**; *Tabla 5.1*.

	Secuencial	OpenMP	SSE	AVX	Open-SSE	Open-AVX	2 < s.u. < 4	s.u. > 4
Secuencial	-	*C1	✗	✗	*C2	*C2	-	-
OpenMP	complementario *C1	-	✗	✗	✗	✗	✓ máx = threads = 4	-
SSE	✓	✓	-	✗	*C3	*C3	✓ tamaño de rep. vectorial	-
AVX	✓	✓	✓	-	*C4	*C4	*SU1	complementario *SU1
Open-SSE	complementario *C2	✓	complementario *C3	complementario *C4	-	✗	*SU2	*SU2
Open-AVX	complementario *C2	✓	complementario *C3	complementario *C4	✓	-	*SU3	*SU3

Tabla 5.1 Resumen de los criterios de uso BLAS2 (optimización explícita)

Criterios:

*C1 filas < $2^6 \wedge BS < 2^{10}$
filas < $2^7 \wedge BS < 2^9$
filas > $2^7 \wedge BS < 2^7$

*C3 filas < 2^6
filas < $2^7 \wedge BS < 2^{10}$

*C2 filas < $2^6 \wedge BS \leq 2^{10}$
filas < $2^7 \wedge BS < 2^8$

*C4 $BS < 2^{11}$
filas < $2^{11} \wedge BS > 2^{11}$

Speed-up:

***SU1** $BS \leq 2^7$
filas $> 2^{11} \wedge BS > 2^{11}$

***SU2** filas $< 2^6 \wedge BS > 2^{11}$
filas $< 2^7 \wedge 2^9 < BS < 2^{10}$

***SU3** filas $> 2^8 \wedge BS \geq 2^7$
filas $> 2^6 \wedge BS > 2^{10}$

Seguindo por el criterio de uso en **BLAS2** entre la *optimización explícita (GCC)*, la *auto-optimización (Auto)* y las librerías **MKL**; *Tabla 5.1*.

	GCC	Auto	MKL
GCC	-	Cuando <i>GCC</i> ejecuta en serie	Cuando <i>GCC</i> ejecuta en serie
Auto	Cuando <i>GCC</i> no ejecuta en serie	-	Cuando no son matrices densas (<i>MKL</i> no utiliza las funciones <i>TRMV / GEMV</i>)
MKL	Cuando <i>GCC</i> no ejecuta en serie	Cuando son matrices densas (<i>MKL</i> utiliza las funciones <i>TRMV / GEMV</i>)	-

Tabla 5.2 Resumen general de los criterios de uso (compiladores y MKL)

El criterio de uso en **BLAS3** entre los diferentes algoritmos desarrollados, además de utilizar también *MKL*; *Tabla 5.3*.

	Secuencial	Intercambio de bucles	"Blocking"	MKL
Secuencial	-	✗	✗	✗
Intercambio de bucles	✓	-	Cuando el tamaño de la matriz es inferior o igual a 1024 x 1024	✗
"Blocking"	✓	Cuando el tamaño de la matriz es superior o igual a 1024 x 1024	-	✗
MKL	✓	✓	✓	-

Tabla 5.3 Criterios de uso de los algoritmos **BLAS3**

Y finalmente el criterio de uso de la **GPU** – NVIDIA CUDA – tanto de **BLAS2** como de **BLAS3**;

Tabla 5.4.

Si disponemos de GPU		
En BLAS2	Si no reutilizamos la estructura de datos	CPU
	Si reutilizamos la estructura de datos	Si el tamaño de la matriz es > 512x512 → GPU
En BLAS3	Si no reutilizamos la estructura de datos	Si el tamaño de la matriz es ≥ 512x512 → GPU
	Si reutilizamos la estructura de datos	GPU

Tabla 5.4 Criterios de uso de la GPU

Queda como **línea abierta** extender el estudio de este proyecto a un *cluster* (memoria distribuida) utilizando, por ejemplo, *MPI* (Message Passing Interface).

Las **instalaciones necesarias** para el proyecto han sido las de los programas: *GCC* (MinGW), *ICC*, *Microsoft Visual Studio*, programa estadístico 'R' y *CUDA v6.0*.

Los **conocimientos previos** a la realización de este proyecto y el **progreso alcanzado** con las diferentes herramientas, lenguajes de programación y técnicas utilizadas se muestran en la *Tabla 5.5*.

Herramienta / Lenguaje / Compilador	Nivel de conocimiento		
	Nulo	Introductorio	Avanzado
C (lenguaje de prog.)			✓ →
Paralelizar		✓ →	
Vectorizar		✓ →	
Loop peeling	✓ →		
NVIDIA CUDA	✓ →		
GCC (compilador)			✓ →
ICC (compilador)	✓ →		
MKL (librería)	✓ →		
Intel® Advisor	✓ →		
Intel® VTune	✓ →		
Intel® Roofline	✓ →		
Microsoft Visual Studio	✓ →		
Álgebra matricial (bandas)		✓ →	
R (programa estadístico)	✓ →		

Tabla 5.5 Niveles de conocimiento propios

Tras la realización de este proyecto soy mucho más consciente de la complejidad de implementar programas que se preocupen de la máquina en la que se van a ejecutar. Es complicado implementar cada una de las metodologías (*secuencial, paralelo, vectorial, paralelo-vectorial, GPU, ...*) y encontrar un criterio de decisión para utilizar unos métodos u otros, y más sabiendo que cada *CPU* y cada *GPU* (en general, cada sistema de cómputo completo) presenta una casuística propia expresa. Requiere muchísimas pruebas, muchos sistemas de prueba diferentes y muchísimo tiempo de análisis implementar un programa que sea lo más eficiente posible independientemente de la arquitectura.

Es bueno saber que podemos contar con herramientas como el *Intel® Parallel Studio* que hacen estas tareas por nosotros de forma automatizada, incluso mejor que nosotros, sin apenas esfuerzo, con un *feedback* tremendo y con infinidad de posibilidades. Hasta ahora nunca había probado una herramienta parecida pero se ve que tienen un potencial muy grande y que apenas he llegado a la superficie de lo que ofrecen.

En general, se puede decir que he podido profundizar mucho en cosas que ya sabía parcialmente, y que he podido aprender cosas nuevas. Pero cada vez soy más consciente, como decía Sócrates, de que *solo sé que no sé nada*, y de que me queda mucho que aprender.

Bibliografía y referencias

- i. Wikipedia – *Superscalar processor*. [Internet; inglés; Marzo 2017]
URL: https://en.wikipedia.org/wiki/Superscalar_processor
- ii. Wikipedia – *Superescalar*. [Internet; castellano; Marzo 2017]
URL: <https://es.wikipedia.org/wiki/Superescalar>
- iii. Wikipedia – *Data parallelism*. [Internet; inglés; Marzo 2017]
URL: https://en.wikipedia.org/wiki/Data_parallelism
- iv. Wikipedia – *Vector processor*. [Internet; inglés; Marzo de 2017]
URL: https://en.wikipedia.org/wiki/Vector_processor
- v. Wikipedia – *GNU Compiler Collection*. [Internet; inglés; Marzo de 2017]
URL: https://en.wikipedia.org/wiki/GNU_Compiler_Collection
- vi. Intel – *Guía de intrínsecas*. [Internet; inglés; periódicamente: Septiembre 2016 – Junio 2017]
URL: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>
- vii. Intel – *BLAS Routines*. [Internet; inglés, periódicamente: Febrero 2017 – Abril 2017]
URL: <https://software.intel.com/en-us/node/520725>
- viii. Intel, artículo – *Introducción a la tecnología HyperThreading*. [Internet; inglés; Marzo 2017]
URL: <https://software.intel.com/en-us/articles/introduction-to-hyper-threading-technology>
- ix. Intel, Software – *Free Software tools*. [Internet; inglés; Diciembre 2016]
URL: <https://software.intel.com/en-us/qualify-for-free-software/student>
- x. NVIDIA – *Procesamiento paralelo CUDA*. [Internet; castellano; Mayo 2017]
URL: <http://www.nvidia.es/object/cuda-parallel-computing-es.html>
- xi. NVIDIA – *Using Shared Memory in CUDA C/C++*. [Internet, inglés; Mayo 2016]
URL: <https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/>
- xii. MinGW – *Minimalis GNU for Windows*. [Internet; inglés; Septiembre 2016]
URL: <http://www.mingw.org/>
- xiii. On The Hub – *Plataforma UPV/EHU – FISS*. [Internet; castellano; Diciembre 2016]
URL: <https://e5.onthehub.com/WebStore/ProductsByMajorVersionList.aspx?ws=0060a29a-689b-e011-969d-0030487d8897>
- xiv. R-project – *Software matemático*. [Internet; inglés, Diciembre 2016]
URL: <https://www.r-project.org/>
- xv. C Reference Card (ANSI). [formato físico; inglés; 1999]
Autor: Joseph H. Silverman
- xvi. ars Technica – *Pipelining: An Overview*. [Internet; inglés; Marzo 2017]
URL: <https://arstechnica.com/features/2004/09/pipelining-2/>