Universidad de Murcia
Facultad de Informática

Grado en Ingeniería Informática
3.er Curso

Arquitectura y Organización de Computadores

Práctica 1 - Introducción a la computación de alto rendimiento

Curso académico 2021–2022

Departamento de Ingeniería y Tecnología de Computadores
Área de Arquitectura y Tecnología de Computadores

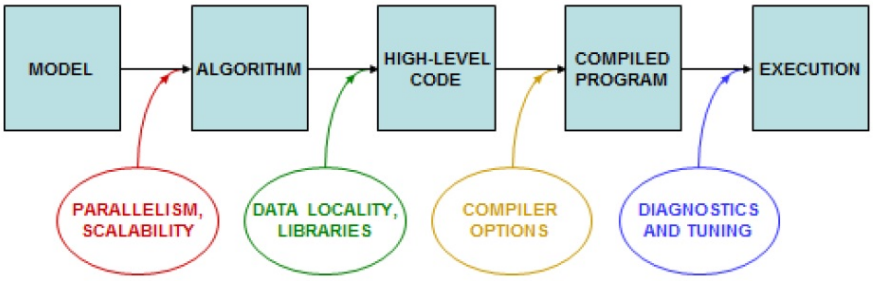


Figura 1: Estados en el desarrollo de software

1. Introducción

En esta asignatura estamos estudiando el diseño de los procesadores de alto rendimiento actuales, los cuales explotan el paralelismo tanto a nivel de hilo como de instrucción apoyándose en diversas técnicas que se explican a lo largo del curso, como los procesadores multicore, la ejecución fuera de orden, la especulación o la vectorización.

Los procesadores de alto rendimiento mencionados anteriormente ofrecen, gracias a estas técnicas y al avance de la tecnología VLSI, una capacidad de cómputo considerablemente mayor de la que podíamos disponer hace pocos años. Pero aprovechar esta capacidad de cálculo no siempre es sencillo. Es necesario proporcionarle al procesador programas que se hayan escrito de tal manera que aprovechen las características del procesador concreto. El objetivo principal de estas prácticas es introducir las técnicas básicas para obtener el mayor rendimiento de un programa al ejecutarlo en un procesador actual. Nos centraremos en la optimización de las aplicaciones científicas.

El proceso de construcción de una aplicación científica comienza con el diseño de un algoritmo, sigue con su implementación en un lenguaje de programación determinado, y termina con una pieza específica de software. En la Figura 1 se muestra una vista de dicho proceso. Las decisiones que se tomen a la hora de diseñar el algoritmo y programarlo afectarán al rendimiento del programa final. Para obtener el mejor resultado en cuanto al rendimiento, es necesario construir un programa *optimizado*. Para ello, se deben identificar las partes del programa que consumen más tiempo e invertir en ellas esfuerzo para mejorar su rendimiento mediante diferentes técnicas que incluyen la paralelización, la vectorización y el uso de librerías de funciones ya optimizadas para alto rendimiento. Además debemos utilizar adecuadamente las herramientas de desarrollo disponibles, especialmente los compiladores, que será necesario configurar mediante las opciones adecuadas para que generen código optimizado.

El rendimiento final de un programa se puede medir de formas diversas, siendo las más frecuentes la velocidad a la que se ejecuta la aplicación, la energía consumida por la misma o la cantidad de memoria que necesita para su ejecución. En estas prácticas nos fijaremos principalmente en el tiempo de ejecución, o equivalentemente en el caso de las aplicaciones científicas que consideramos: en la velocidad a la que se realizan operaciones de coma flotante.

La identificación de las partes del programa que más tiempo consumen y que, por tanto, es más útil optimizar, se puede realizar de varias formas. En esta práctica utilizaremos la más sencilla, que consiste en instrumentar el código fuente por medio de llamadas a funciones para medir el tiempo, aunque también existen múltiples herramientas para ayudarnos en este proceso y para obtener información más detallada (como qué accesos a memoria han producido fallos de caché, o qué saltos están siendo mal predichos) que nos permiten diagnosticar mejor los problemas de rendimiento. En esta categoría se pueden mencionar por su popularidad las herramientas `perf`[1], `Vtune`[2] o `gprof`[3].

Esta primera práctica trata de ayudar a conocer las características comunes de la microarquitectura de los procesadores multinúcleo actuales y a tener una idea de cómo el compilador trata de optimizar los programas para usar dichas características. Las prácticas segunda y tercera ampliarán estos conceptos

[1] `https://perf.wiki.kernel.org/index.php/Main_Page`
[2] `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html`
[3] https://en.wikipedia.org/wiki/Gprof

por medio de explotar las capacidades vectoriales y multinúcleo que tienen los procesadores actuales (práctica 2), y por medio de describir algunas optimizaciones avanzadas (práctica 3) que se pueden dar en las arquitecturas multinúcleo actuales.

2. Objetivos

El objetivo general de este primer boletín es aprender de manera práctica a obtener el mejor rendimiento del software en plataformas de computación modernas, centrándonos en los CMP.

Más concretamente, los objetivos de esta práctica son:

- Conocer cómo calcular y obtener el rendimiento y el ancho de banda pico de un procesador multinúcleo.
- Mostrar la relación entre el código escrito en lenguaje de alto nivel empleado y el código binario final generado por un compilador, y entender cómo se ve afectado por las diversas opciones de optimización ofrecidas por los compiladores actuales.
- Conocer el problema de los riesgos de datos en las variables de una aplicación, y cómo afecta al rendimiento obtenido por la aplicación.
- Ver un ejemplo de programación de una aplicación muy simple con necesidades de altas prestaciones a través de un algoritmo típico de cálculo numérico, optimizándolo mediante el uso de una librería de funciones matemáticas optimizadas para alto rendimiento.

3. Entorno de trabajo

Para la realización de estas prácticas vamos a utilizar compiladores de C++ modernos capaces de optimizar eficazmente programas para su ejecución en los procesadores actuales. Usaremos principalmente el GCC versión 11.0.1 y el ICC versión 2021.3.0, y mostraremos en algunos casos los resultados obtenidos también por Clang 12.0.0. GCC[4] es el compilador de GNU, que es capaz de generar código optimizado para prácticamente cualquier ISA y que está incluido en todas las distribuciones de Linux. ICC es el compilador de Intel, el cual se puede descargar de la página web de la compañía[5]. Clang es el compilador de C y C++ basado en LLVM[6] usado, entre otros, por Apple. Los tres compiladores soportan el estándar actual de C++ (C++20) y están disponibles en la mayoría de los sistemas operativos. Es importante utilizar las versiones indicadas de los compiladores a la hora de realizar las prácticas para obtener resultados correctos y evitar problemas de compatibilidad (normalmente se podrá utilizar versiones más nuevas, aunque los resultados obtenidos pueden ser inesperados).

Para facilitar el acceso a las versiones adecuadas de los compiladores y algunos otros programas que se utilizarán durante las prácticas, se ha preparado un *script* que genera una imagen Docker que se puede utilizar, si se desea, para realizar las prácticas. No se aconseja el uso de máquinas virtuales (como QEMU, KVM, VirtualBox o similares) para la realización de las prácticas, pues consumen recursos del sistema y no nos permiten obtener las máximas prestaciones del entorno que tengamos.

4. Características del computador utilizado

En primer lugar, necesitamos conocer adecuadamente las características básicas de la CPU y el sistema de memoria del ordenador que utilizaremos. La mayoría de la información la podremos obtener mirando en los ficheros `/proc/cpuinfo` y `/proc/meminfo`. Existe un gran número de utilidades que también nos pueden ayudar a determinar algunas características, como `lshw`, `lscpu`, `dmidecode`,

[4] `https://gcc.gnu.org/`

[5] `https://software.intel.com/content/www/us/en/develop/tools/oneapi/all-toolkits.html`. Usaremos la versión «clásica» de ICC, aunque esta versión está a punto de dejar de ser soportada por Intel, que va a basar las nuevas versiones de su compilador en Clang.

[6] `https://llvm.org/`

Nombre	`compiladorintel.inf.um.es`	`hostname -f`
Sistema operativo	Ubuntu 14.04.6 LTS	`/etc/os-release`
CPU		
Modelo	Intel® Core™ i5-6400 CPU @ 2.70GHz	`lscpu`, `lshw`
Microarquitectura	Skylake	`ark.intel.com`, `cpufetch`
Número de cores	4	`lscpu`, `cpufetch`
SMT / HyperThreading	No	`ark.intel.com`
Frcuencia mínima	800 MHz	`lscpu`
Frecuencia máxima	3300 MHz	`lscpu`, `cpufetch`
Fecha de lanzamiento	Tercer trimestre de 2015	`ark.intel.com`
Litografía	14 nm	`ark.intel.com`, `cpufetch`
TDP	65 W	`ark.intel.com`
Caché L1 instrucciones	32 KiB (64 conjuntos, 8 vías, 64 bytes/bloque) por core (privada)	`lscpu`, `lshw -C memory`
Caché L1 datos	32 KiB por core (64 conjuntos, 8 vías, 64 bytes/bloque, privada)	`lscpu`, `lshw -C memory`
Caché L2	256 KiB por core (1024 conjuntos, 4 vías, 64 bytes/bloque, privada)	`lscpu`, `lshw -C memory`
Caché L3	6 MiB en total (8192 conjuntos, 12 vías, 64 bytes/bloque, compartida)	`lscpu`, `lshw -C memory`
Extensiones ISA soportadas	SSE4.1/4.2, AVX2 y otras	`ark.intel.com`, `lscpu`
Tamaño de registros vectoriales	256 bits (8 floats ó 4 doubles) `AVX2`	`lscpu`, `cpufetch`
Unidades FPU (vectoriales)	2 Add, 2 Mul	`wikichip.org`
Unidades de FMA	2	`wikichip.org`
Memoria		
Total instalada	8 GiB (1 DIMMs)	`lshw -C memory`
Controladores	1	`lshw -C memory`
Canales/controlador	2 (1 en uso solo)	`ark.intel.com`, `lshw -C memory`
Frecuencia	2133 MHz	`lshw -C memory`

Tabla 1: Características del servidor `compiladorintel.inf.um.es`. Se muestra a la derecha de cada dato las fuentes de las que se ha obtenido, donde `ark.intel.com` se refiere a la página `https://ark.intel.com/content/www/us/en/ark/products/88185/intel-core-i56400-processor-6m-cache-up-to-3-30-ghz.html` y `wikichip.org` se refiere a `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)`.

`cpufetch`[7] y muchas otras. Téngase en cuenta que para utilizar estas herramientas y obtener toda la información que pueden proporcionar es necesario tener permisos de `root`. Para algunos datos, una vez identificado el modelo exacto del procesador, será útil consultar la información provista por el fabricante (por ejemplo en `https://www.amd.com/en/products/specifications` para AMD o `https://ark.intel.com/products` para Intel) y otras fuentes como `https://en.wikichip.org/`.

En la tabla 1 se muestran las características de un servidor de la Facultad denominado `compiladorintel.inf.um` el cual se usará de ejemplo a lo largo del boletín (aviso: el nombre de dicha máquina no implica que se esté utilizando exclusivamente el compilador de Intel).

Es interesante conocer la versión del sistema operativo que se está empleando, aunque para las pruebas que vamos a realizar el resultado debería ser el mismo en cualquier caso. Debido a que los sistemas operativos que usaremos son multitarea y a que la velocidad de funcionamiento de los procesadores actuales no es constante[8], tendremos que considerar que el tiempo de ejecución de cualquier programa es una variable aleatoria. Por ello, siempre realizaremos varias medidas para las cuales calcularemos después la media y la variabilidad (desviación estándar). Además, siempre que realicemos alguna medi-

[7] Este programa está diponible en `https://github.com/Dr-Noob/cpufetch/`

[8] La frecuencia varía lo largo del tiempo en función de la carga y la temperatura, además de otras fuentes de variabilidad (e.g., el acceso a la DRAM).

da de tiempo será importante asegurarse de que el ordenador no esté haciendo otras tareas en segundo plano.

5. El papel del compilador y las opciones de compilación

Los programas que se ejecutan en los procesadores están casi siempre escritos en un lenguaje de alto nivel y necesitan ser traducidos a código máquina antes de ser efectivamente ejecutados. De esta traducción se encarga el compilador, y dependiendo de si hace un trabajo mejor o peor obtendremos un buen rendimiento o no.

Podemos pensar que obtendríamos un rendimiento mejor si la codificación de los programas la hiciéramos directamente en ensamblador, pero estaríamos equivocándonos en la mayoría de los casos. Si programáramos directamente en ensamblador, no solo sería muchísimo más difícil acometer el diseño de cualquier programa complejo, sino que el rendimiento que obtendríamos sería igual o peor en la mayoría de los casos. Los compiladores actualmente son programas muy complejos y muy avanzados, que si se les proporciona código fuente de buena calidad producirán un código ensamblador que será casi siempre mejor que el escrito por el programador directamente en ensamblador. Y es ciertamente más fácil escribir y mantener el código en un lenguaje de alto nivel que hacerlo en ensamblador.

Para que el compilador sea capaz de producir un ejecutable con el mejor rendimiento posible, tiene que recibir código fuente de calidad y estar correctamente configurado mediante las múltiples opciones (*flags*, *switches* o *knobs*) que soporte para realizar los análisis y transformaciones necesarios para la optimización.

Un código fuente de baja calidad puede complicar la labor de optimización del compilador. Hay muchas cosas que el programador puede hacer para ayudar al compilador a realizar un mejor trabajo, y también hay cosas que se deben hacer para al menos no obstaculizar los intentos del compilador de optimizar el código. Además, es muy importante no intentar nunca realizar optimizaciones manualmente a ciegas (es decir: sin medir cuidadosamente su efecto), ya que estas «optimizaciones» pueden complicar la labor del compilador y provocar que consigamos un rendimiento peor.

El rendimiento obtenido por una aplicación depende mucho del compilador utilizado y de las opciones de compilación que se han usado. Aunque en principio cada compilador tiene sus propias opciones y siempre tendremos que consultar la documentación del compilador utilizado para elegir las más adecuadas, al menos en el caso de los compiladores de C y C++ algunas opciones de compilación se han convertido en estándar de facto, por lo que funcionarán en la mayoría de los compiladores. Por ejemplo, la mayoría de los compiladores soportan una opción `-O` para especificar el «nivel de optimización» sin entrar en detalle de cada optimización concreta. Normalmente se soportan varios niveles:

- `-O0` para la compilación rápida sin optimización.
- `-O1` para una optimización limitada que no aumenta mucho el tamaño del código ni el tiempo de compilación.
- `-O2` para optimización moderada. Normalmente este es el nivel más adecuado para la mayoría de las aplicaciones, aunque en el caso de las aplicaciones científicas conviene asegurarse de que también se activa el soporte para vectorización.
- `-O3` para una optimización agresiva, que puede incrementar significativamente el tiempo de compilación y el tamaño del binario resultante. Por ejemplo, en el caso del compilador GCC, se activa la vectorización automática.

La mayoría de los compiladores también soportan la opción `-Wall`, que activa todos los avisos (*warnings*) del compilador. Esta opción es extremadamente útil a la hora de escribir código correcto y eficiente. Por razones históricas, C y C++ permiten construcciones que, aunque técnicamente sean válidas, son casi siempre un error de programación, y así debemos tratar siempre a los avisos que nos proporciona el compilador. Si estos lenguajes pudieran ser diseñados de nuevo sin preocuparnos de la compatibilidad con los programas ya escritos, muchas de estas construcciones no se aceptarían. Es

especialmente importante eliminar todos los avisos detectados en nuestro código si queremos que el compilador optimice al máximo y lo haga correctamente.

También hay que mencionar la opción `-g` soportada por la mayoría de los compiladores y que activa el soporte para depuración[9]. Es conveniente activarla siempre, salvo cuando queramos evitar un tamaño excesivo del binario generado.

Para poder aprovechar al máximo las características de un procesador concreto, suele ser útil agregar una opción para que el compilador sepa que también debe tratar de optimizar el código para ese modelo particular de procesador. De no hacerlo, el compilador tenderá a generar código compatible con otros procesadores y no usará información específica del que realmente queramos utilizar. Estas opciones se llaman `-march` y `-mtune` en el caso de GCC y Clang y `-xhost` en el caso de ICC. Normalmente se puede pedir que el compilador detecte automáticamente las características del procesador utilizado (e.g., `-march=native`) o especificar uno concreto (e.g., `-march=opteron`).

6. Rendimiento y ancho de banda pico

En primer lugar, para guiarnos a la hora de optimizar una aplicación, vamos a calcular el rendimiento pico teórico de nuestro computador, para saber hasta dónde es posible llegar. También mediremos experimentalmente el mismo valor, para comprobar si realmente podemos alcanzar el rendimiento predicho por nuestros cálculos.

El rendimiento de un computador depende mucho del tipo de aplicación que estemos ejecutando. **En** la mayoría de las **aplicaciones de cálculo científico**, podemos definir el rendimiento de un procesador como el número de operaciones en coma flotante por segundo que se realizan. Es importante recordar que este dato nos puede dar una idea de cómo de rápido es nuestro ordenador a la hora de realizar cálculos en coma flotante, pero no nos proporciona absolutamente ninguna información respecto a cómo de rápido es para otras tareas (como gestionar bases de datos o compilar programas).

Utilizaremos como unidad de medida los FLOPS[10], aunque lo más frecuente será usar un múltiplo de esta unidad, habitualmente MFLOPS (10^6) o GFLOPS (10^9). Además, esta medida es diferente si realizamos operaciones con números en simple o doble precisión.

En primer lugar, calcularemos el **rendimiento teórico pico** usando coma flotante de simple precisión para un solo core. Consideraremos que usamos todas sus unidades vectoriales. Supondremos, por simplicidad, que se realizan únicamente operaciones de multiplicación y suma combinda (Fused Multiply/Add, FMA), que son soportadas por la mayoría de los procesadores disponibles actualmente[11].

La formula que nos da el rendimiento pico de un procesador es la siguiente:

$$\begin{aligned} Rendimiento\_pico_1(tipo) &= elementos/instrucción \times operaciones/instrucción \\ &\quad \times instrucciones/ciclo \times frecuencia_1 \end{aligned}$$

Cada término representa lo siguiente:

- *tipo*: especifica si utilizamos números en coma flotante de simple precisión (SP) o doble precisión (DP). Estos tipos equivaldrían a los tipos `float` y `double` de C, respectivamente.

[9] Para facilitar la depuración cuando se detecta algún problema, suele ser útil reducir el nivel de optimización e incluso usar la opción `-O0`, ya que algunas optimizaciones pueden dificultar el seguimiento del programa.

[10] **FLO**ating **P**oint **O**perations **P**er **S**econd, a veces tambiés escrito como FLOP/s o flop/s. No se debe confundir con los FLOP o FLOPs (FLOating Point OPerations), que se utiliza a veces para denotar el número total de operaciones de coma flotante realizadas. Ver `https://en.wikipedia.org/wiki/FLOPS`

[11] Los procesadores sin soporte para este tipo de instrucciones utilizarán instrucciones independientes de suma y multiplicación para realizar el mismo trabajo y consideraremos que medimos así su rendimiento pico. Aunque, en algún caso, podría alcanzarse un rendimiento pico mayor al calculado con la metodología de este boletín combinando el uso unidades FMA con otras unidades funcionales si el procesador lo permite, o balanceando de forma diferente el número de operaciones de suma y multiplicación.

- *elementos/instrucción*: es el número de operaciones que son ejecutadas simultáneamente por una instrucción. Si la instrucción es escalar este término vale 1, pero si la instrucción es vectorial es igual al ancho de la unidad vectorial.
- *operaciones/instrucción*: es el número de operaciones en coma flotante que cada instrucción puede realizar. Lo normal es que cada instrucción realice una única operación (suma, resta, multiplicación, etc), pero las instrucciones FMA son capaces de realizar una operación de suma y una operación de multiplicación en coma flotante a la vez (por lo que intentaremos usarlas, si están disponibles, a la hora de calcular el rendimiento pico).
- *instrucciones/ciclo*: es el número de instrucciones que pueden ser ejecutadas simultáneamente en un ciclo. Aunque es un parámetro complejo de conocer con exactitud, ya que está relacionado con el cauce de cada núcleo, el número de unidades funcionales que tiene y sus latencias, en la mayoría de los casos se puede aproximar por el número de unidades vectoriales que tiene cada núcleo (o de unidades vectoriales FMA si se utilizan estas instrucciones).
- $frecuencia_1$: es la frecuencia a la que opera el núcleo del procesador usado. Es de nuevo un parámetro complicado, porque la frecuencia de funcionamiento de cada núcleo en los procesadores actuales varía en función de la carga y la temperatura en cada momento. Los procesadores soportan un rango de frecuencias de forma que cada núcleo tiende a permanecer en la frecuencia mínima cuando no tiene trabajo y a utilizar la frecuencia máxima posible solo cuando haya instrucciones que ejecutar. Normalmente la frecuencia máxima solo se puede alcanzar cuando se utiliza uno o pocos núcleos a la vez, mientras que si están todos los núcleos funcionando a la vez se utilizarán frecuencias un 10–30 % más bajas. El tipo de instrucciones que se ejecuten también puede afectar a la frecuencia (por ejemplo: el uso de instrucciones vectoriales puede reducir la frecuencia). Indicaremos con un subíndice que nos referimos a la frecuencia que utiliza el procesador con un número de núcleos activos dado (1 en este caso).

En el caso del servidor de ejemplo, se trata de una arquitectura *skylake* que tiene instrucciones de FMA, con unidades vectoriales de 256 bits que **para números reales en simple precisión** (`float`, 32 bits) equivalen a **8** operaciones en coma flotante por cada instrucción. Cada núcleo tiene 2 unidades vectoriales FMA. La frecuencia máxima es de 3300 MHz. Por tanto:

$$\begin{aligned} Rendimiento\_pico_1(\text{sp}) &= 8\ (elementos/instrucción) \times 2\ (operaciones/instrucción) \\ &\quad \times 2\ (instrucciones/ciclo) \times 3300\ \text{MHz}\ (frecuencia_1) \\ &= 105{,}600\ \text{GFLOPS} \end{aligned}$$

Sin embargo, un procesador puede tener varios cores (4 en nuestro caso), y el servidor podría tener varios procesadores en diferentes sockets. Teniendo en cuenta esto, la formula que nos da el rendimiento pico de un procesador CMP actual es:

$$\begin{aligned} Rendimiento\_pico_{cores/procesador,procesadores}(tipo) &= elementos/instrucción \times operaciones/instrucción \\ &\quad \times instrucciones/ciclo \times frecuencia_{cores/procesador} \\ &\quad \times cores/procesador \times procesadores \end{aligned}$$

Donde:

- *cores/procesador*: es el número de núcleos físicos (reales) que tiene cada socket del procesador. Aunque un núcleo tenga capacidades SMT o HyperThreading (aumentando el número de núcleos virtuales del procesador), supondremos que no se utilizan, por lo que el rendimiento pico se obtiene de utilizar completamente las unidades funcionales de coma flotante con un solo hilo.
- *procesadores*: es el número de procesadores en sockets independientes que tiene el sistema.

En el caso del servidor de ejemplo, tenemos un solo procesador que tiene 4 núcleos (sin HyperThreading). La frecuencia que se utiliza cuando los 4 núcleos están activos a la vez realizando operaciones FMA vectoriales es de 3100 MHz[12]. Por tanto:

$$\begin{aligned} Rendimiento\_pico_{4,1}(\text{sp}) &= 8\ (elementos/instrucción) \times 2\ (operaciones/instrucción) \\ &\quad \times 2\ (instrucciones/ciclo) \times 3\,100\ \text{MHz}\ (frecuencia_4) \\ &\quad \times 4\ (cores/procesador) \times 1\ (procesadores) \\ &= 396{,}800\ \text{GFLOPS} \end{aligned}$$

Es decir, que el rendimiento pico del servidor de la Facultad llamado `compiladorintel.inf.um.es` es de:

$$Rendimiento\_pico_{compiladorintel}(\text{sp}) = Rendimiento\_pico_{i5-6400}(\text{sp}) = 396{,}800\ \text{GFLOPS}$$

Una vez obtido este importante parámetro, vamos a pasar a otro importante parámetro de un ordenador, que es el ancho de banda a memoria. Una de las razones por la que una aplicación de cálculo científico puede no alcanzar el rendimiento pico es porque la memoria no sea capaz de proporcionarle con suficiente rapidez al procesador datos con los que operar. Por tanto, un parámetro que nos interesa también conocer es el ancho de banda a memoria disponible en nuestro computador. Definimos el ancho de banda a memoria como el número de bytes por segundo que es posible transferir entre la memoria y el procesador. Lo habitual es usar un múltiplo de este valor, habitualmente MB/s (10^6) o GB/s (10^9), o bien MiB/s (2^{10}) o GiB/s (2^{20}).

Para calcular el **ancho de banda pico** a memoria que tiene nuestro sistema, considerando que usamos todos los controladores de memoria disponibles y todos los canales asociados a cada controlador, así como todo el ancho del bus de memoria, podemos utilizar la siguiente formula:

$$\begin{aligned} Ancho\_banda\_pico &= controladores\_memoria \times canales/controlador \\ &\quad \times bytes/transferencia \times frecuencia\_memoria \end{aligned}$$

Estos datos los podemos obtener mediante la herramienta `lshw` y de la documentación del procesador. Obsérvese que la frecuencia del bus de memoria no es la misma que la del procesador.

En el caso del servidor de ejemplo, este ordenador tiene un único controlador de memoria, que maneja 2 canales de acceso a memoria (cada uno con capacidad para poner 2 bancos de memoria DIMM), con un ancho de banda por canal de 64 bits (8 bytes), siendo la memoria instalada un único DIMM de 8 GiB de 2133 MHz. Como vemos, estamos utilizando un único canal de memoria (a pesar de que el procesador soporta 2), por lo que el ancho de banda pico teórico para esta configuración es el siguiente:

$$\begin{aligned} Ancho\_banda\_pico &= 1\ (controladores\_memoria) \times 1\ (canales/controlador) \\ &\quad \times 8\ (bytes/transferencia) \times 2133\ \text{MHz}\ (frecuencia\_memoria) \\ &= 17{,}064\ \text{GB/s} \end{aligned}$$

Con unos programas de prueba (*benchmarks*) adecuados, deberíamos ser capaces de aproximarnos a los valores de rendimiento pico teórico y de ancho de banda pico a memoria teórico. En la práctica, no siempre es fácil alcanzar esos valores incluso con programas diseñados únicamente con ese fin y que no realizan trabajo útil. Entre los materiales de este boletín, se incluyen los siguientes programas en el directorio `peak-performance`:

[12] Este valor se puede determinar utilizando el programa `peak-freq` incluido con los materiales de la práctica y el parámetro `--threads=4`.

peak-flops.cpp: Permite medir el rendimiento pico realizando operaciones FMA repetidamente tanto con un núcleo o con varios (utilizando el parámetro `--threads`). Para conseguir el rendimiento pico, el programa debe utilizar todos los núcleos disponibles, y ejecutar las operaciones FMA en un bucle que opera sobre un array cuyo tamaño óptimo depende del compilador y el procesador concreto que se utilice. El programa realiza varias pruebas variando este parámetro (`inner_iterations`) automáticamente. Dependiendo del procesador y del compilador, el mejor rendimiento se obtendrá normalmente con un valor de 32, 64 ó 128. Para permitir que el compilador pueda tener en cuenta el valor de este parámetro para optimizar, debe conocerse en tiempo de compilación. Por ello, el programa compilado incluye varias versiones del mismo procedimiento[13] generadas a partir del procedimiento plantilla `calculate`.

peak-mem.cpp: Mide el ancho de banda de memoria que se otiene al copiar datos entre dos arrays. Utiliza tanto un bucle manual como la función `strcpy`.

peak-freq.cpp: Permite medir la frecuencia utilizada por los núcleos del procesador mientras se realiza trabajo, ya sea con un núcleo o con varios (utilizando el parámetro `--threads`). Muestra la frecuencia media de todos los núcleos y la frecuencia del núcleo más rápido.

Todos los programas anteriores se pueden compilar (utilizando el `Makefile` incluido, con el comando «`make all`»). Es importante observar algunas de las opciones de compilación utilizadas. Por ejemplo, en el caso de GCC[14]:

`-std=c++20` : Indicamos que utilizaremos el estándar de 2020 de C++. Esto nos permite especificar de forma más limpia algunos detalles importantes, como el alineamiento de los datos.

`-Wall` : Solicitamos al compilador que nos avise de cualquier código sospechoso aunque no sea un error.

`-march=native` : Le indicamos al compilador que puede generar código que utilice cualquier instrucción soportada por la máquina en la que se está compilando. Esto puede hacer que el binario no se ejecute en otros procesadores. También se podría especificar una microarquitectura concreta (por ejemplo, con `-march=pentium4`). La opción equivalente a esta en combinación con la siguiente (`-mtune`) en el caso del compilador ICC es `-xHost`.

`-mtune=native` : Le indicamos al compilador que optimice teniendo en cuenta las características del procesador de la máquina usada para compilar. Esta opción afecta a la hora de tomar decisiones a la hora de optimizar, pero no a la compatibilidad del código generado. También se podría especificar una microarquitectura concreta (por ejemplo, con `-mtune=skylake`).

`-O3` : Activamos todas las optimizaciones, incluyendo algunas potencialmente costosas como la vectorización[15].

`-ffast-math` : Permitimos al compilador que haga ciertas suposiciones sobre los números en coma flotante que no son técnicamente correctas de acuerdo con el estándar IEEE-754 pero que son válidas en la mayoría de los casos. En general, con esta opción el compilador asume que los números en coma flotante se comportan como si fueran números reales. Por ejemplo, permite tratar las operaciónes de suma como asociativas. Esto permite mucha mayor libertad a la hora de reordenar operaciones y es importante para muchas optimizaciones.

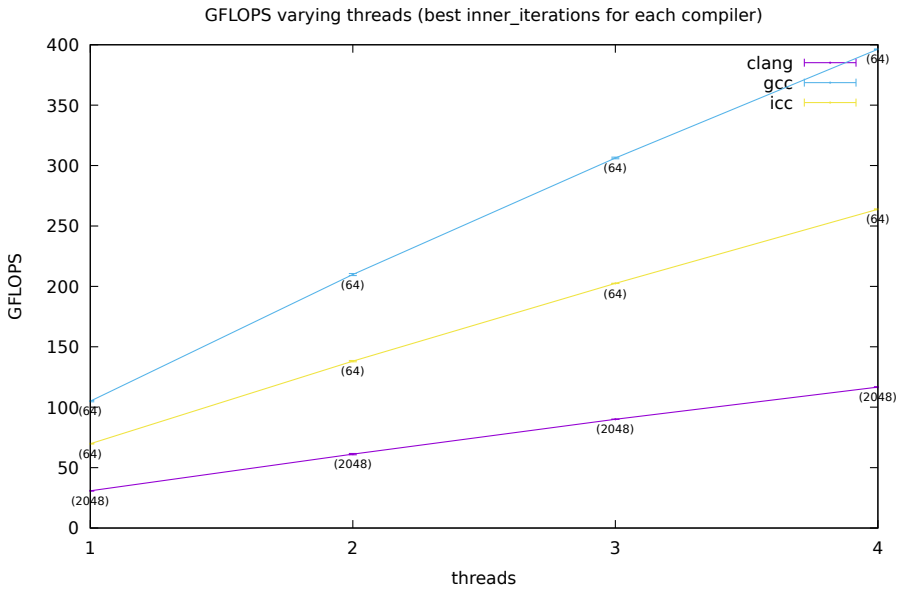
`-fopenmp` : Activamos el soporte para OpenMP, el cual se utilizará para ejecutar varios hilos en paralelo.

Obsérvese que, al utilizar las opciones `-march=native` y `-mtune=native`, el resultado de rendimiento obtenido por los programas dependerá del ordenador utilizado para compilar, además del ordenador en

[13] Podemos comprobarlo y analizar el código ensamblador generado con `objdump`. Si añadimos la opción `-g` al comando de compilación, podemos usar `objdump -Sl` para ver el código fuente correspondiente a cada parte del ensamblador.

[14] Clang utiliza las mismas opciones que GCC.

[15] ICC activa la vectorización automática con `-O2`.



	Hilos			
Compilador	1	2	3	4
gcc (64)	105.21±0.16	209.73±0.61	306.71±0.09	396.10±0.32
icc (64)	69.59±0.18	135.63±4.01	202.50±0.06	263.93±0.30
clang (2048)	30.92±0.03	61.64±0.12	89.79±0.34	116.89±0.00

Figura 2: Resultados de `peak-flops.cpp` en `compiladorintel.inf.um.es`. En la tabla se muestan los GFLOPS alcanzados por cada compilador para el mejor valor de `inner_iterations` probado, indicando entre paréntesis dicho valor.

el que se ejecuten (que esperamos que normalmente sea el mismo). Además, el binario generado no será portable.

Podemos compilar (con «`make all`») todos los programas anteriores en el ordenador que estemos evaluando y ver qué resultado obtenemos con el ejecutable generado por cada compilador. Probaremos a ejecutar cada una de los binarios generados (`peak-flops-gcc`, `peak-flops-icc` y `peak-flops-clang`) utilizando distinto número de hilos mediante la opción `--threads`.

En el caso del servidor de ejemplo, los resultados se muestran en la figura 2[16]. Podemos ver que el compilador GCC obtiene casi el 100 % del rendimiento pico predicho tanto en el caso de 1 núcleo como en el de 4 núcleos, mientras que ICC se queda en el 66 % aproximadamente en ambos casos y Clang no llega al 30 %. Aunque el programa que estamos compilando es aparentemente simple (se trata de un bucle que hace sumas y multiplicaciones en un bucle sin dependencias), vemos que el compilador usado para producir el binario tiene una gran influencia en el rendimiento final. A pesar de que los tres compiladores son programas muy avanzados y maduros que suelen obtener muy buenos resultados de rendimiento con programas reales, vemos que en este caso algunos se quedan lejos del rendimiento máximo potencial. Por otro lado, hay que tener en cuenta que este código no es muy representativo de ningún programa real, por lo que no debe interpretarse como un benchmark muy relevante para evaluar ninguno de los compiladores.

Como prueba de la importancia de las optimizaciones, podemos probar a compilar el mismo código sin las opciones de optimización. La línea mínima que permite compilar este programa es «`g++ -std=c++20 -fopenmp util.cpp peak-flops.cpp -o peak-flops-unoptimized`». Si probamos el rendimiento del binario generado, vemos que se queda muy lejos del objetivo, alcanzando solo 0,76 GFLOPS.

Como el código ensamblador óptimo para conseguir que un procesador ejecute operaciones en coma flotante a la mayor velocidad posible sin importarnos la utilidad de los resultados calculados no es

[16] Para lograr el rendimiento pico, se debe ejecutar estos benchmarks con el ordenador funcionando con una carga mínima (por ejemplo, en modo sin gráficos) y ejecutando `systemctl isolate multi-user.target` para aislar el servidor.

Listado 1: Contenido de `skylake_256.cpp` de `peakperf`, que es el código utilizado para obtener el rendimiento pico en el procesador de `compiladorintel.inf.um.es`.

```
#include "skylake_256.hpp"
#define OP_PER_IT B_256_8_OP_IT

TYPE farr_skylake_256[MAX_NUMBER_THREADS][SIZE] __attribute__((aligned(64)));

void compute_skylake_256(TYPE *farr, TYPE mult, int index) {
  farr = farr_skylake_256[index];

  for(long i=0; i < BENCHMARK_CPU_ITERS; i++) {
    farr[0]  = _mm256_fmadd_ps(mult, farr[0], farr[1]);
    farr[2]  = _mm256_fmadd_ps(mult, farr[2], farr[3]);
    farr[4]  = _mm256_fmadd_ps(mult, farr[4], farr[5]);
    farr[6]  = _mm256_fmadd_ps(mult, farr[6], farr[7]);
    farr[8]  = _mm256_fmadd_ps(mult, farr[8], farr[9]);
    farr[10] = _mm256_fmadd_ps(mult, farr[10], farr[11]);
    farr[12] = _mm256_fmadd_ps(mult, farr[12], farr[13]);
    farr[14] = _mm256_fmadd_ps(mult, farr[14], farr[15]);
  }
}
```

necesariamente el código que va a generar un compilador diseñado para optimizar programas reales aunque intentemos que lo haga, puede ser una buena idea elegir manualmente las instrucciones y el orden en el que queremos que se ejecuten. Esto podemos hacerlo escribiendo el programa (o al menos la función que realiza el cálculo que queremos medir) en ensamblador o mediante el uso de *funciones intrínsecas*. Estas funciones son específicas para cada ISA (y en ocasiones para cada compilador) y se traducen directamente por instrucciones concretas de ensamblador. En el caso del ISA x86-64, las funciones intrínsecas definidas por Intel[17] están soportadas por la mayoría de los compiladores. El código que escribamos en este caso deberá ser diferente para cada procesador que queramos evaluar, y escribir dicho código requerirá de un conocimiento detallado de cada microarquitectura.

Afortunadamente, ya existe un programa para medir el rendimiento pico de los procesadores usando código especializado mediante intrínsecos para muchas microarquitecturas x86-64, llamado `peakperf`[18]. Es fácil localizar en el código fuente del programa el benchmark concreto que se usa para cada microarquitectura. En el listado 1 se muestra el utilizado para el servidor de ejemplo. Si ejecutamos este programa en el servidor de ejemplo, vemos que obtine un rendimiento de 104,92±0,39 GFLOPS con `--threads=1` y de 396,35±0.09 GFLOPS con `--threads=4`, y ambos valores son prácticamente el 100 % del valor teórico calculado.

Por último, podemos medir el ancho de banda de memoria máximo que podemos alcanzar con el programa `peak-mem.cpp`. A diferencia de los núcleos de procesamiento que tienen un comportamiento y unas características de rendimiento más predecibles, el rendimiento del subsistema de memoria se ve muy afectado por la implementación hardware, el ruido electromagnético en las conexiones, y una variedad de otros factores complejos del sistema. Por ello, es difícil que alcancemos el valor pico aunque utilicemos un *benchmark* adecuado.

En este caso, para el servidor de ejemplo, vemos que cuando la prueba la realizamos mediante una copia manual de arrays en C++, ICC obtiene un ancho de banda de 13,00±0.02 GB/s, mientras que GCC y Clang se quedan en 9,11±0.01 GB/s. Cuando se utiliza la función de librería `memcpy` (que está optimizada manualmente por el autor de la librería de C), ambas implementaciones obtienen casi el mismo rendimiento (12,98±0,01 GB/s con ICC y 12,84±0,02 GB/s con GCC). Vemos que en el mejor caso solo llegamos al 75 % del valor pico calculado.

[17] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

[18] Este programa está diponible en `https://github.com/Dr-Noob/peakperf/` y está instalado en la imagen Docker de las prácticas.

7. Riesgos de datos y riesgos estructurales

En esta sección vamos a ver cómo algunas de las razones que pueden limitar el rendimiento obtenido por una aplicación es la existencia de dependencias de datos entre las operaciones en coma flotante que realice, o la limitación en cuanto al número de unidades funcionales disponibles.

Para ello, utilizaremos un pequeño programa de prueba que nos permitirá comprobar cuál es es número de unidades funcionales disponibles en nuestro procesador para hacer operaciones de suma en coma flotante y medir la latencia de estas unidades. Por simplicidad, realizaremos este experimento utilizando un solo hilo y sin tener en cuenta la posibilidad de ejecutar varias sumas en paralelo en cada unidad funcional que nos ofrecen esas unidades (que, como ya hemos mencionado, son vectoriales en los procesadores actuales).

El programa de prueba que utilizaremos se limitará a realizar sumas repetidamente sobre una o varias variables de coma flotante de simple precisión, incrementando el valor de cada variable en 1,0 en cada iteración. Para poder saber exactamente qué instrucciones se van a ejecutar en nuestro programa de prueba, utilizaremos funciones intrínsecas para escribirlo. Además, utilizaremos *pragmas*[19] para indicarle al compilador cuándo queremos que desenrolle o no determinados bucles. De otra forma, las optimizaciones de los compiladores mejorarían mucho el rendimiento de este sencillo programa y harían muy difícil interpretar los resultados de este experimento.

En el directorio `data-hazards` de los materiales de este boletín, se incluyen los siguientes dos programas:

`data-hazards.cpp`: Programa de prueba descrito anteriormente, implementado mediante funciones intrínsecas y pragmas para generar, a partir de una función plantilla, 20 funciones similares que suman repetidamente 1,0 a un número de variables entre 1 y 20. En cada caso, se le suma 1,0 a cada variable el mismo número de veces (por tanto, la versión de dos variables realiza en total el doble de operaciones que la de una variable, y la versión de 20 variables realiza 20 veces más operaciones que la de una variable). El resultado total de la función es la suma de todas las variables. El programa llama a las 20 funciones y calcula el rendimiento obtenido.

`data-hazards-nointrinsics.cpp`: Programa equivalente al anterior implementado sin utilizar funciones intrínsecas ni pragmas, incluido para facilitar la comprensión del anterior y para comprobar que, aunque obtiene exactamente el mismo resultado, los compiladores son capaces de obtener un rendimiento mucho mejor para este programa si lo escribimos directamente sin usar intrínsecos ni pragmas.

Si compilamos y ejecutamos el programa `data-hazards`, éste medirá el rendimiento de 20 versiones de la función plantilla `calculate`, la cual está parametrizada por el entero `num_vars`. Cuando `num_vars` es 1, el trabajo de dicha función es equivalente al siguiente código:

```
for (size_t j = 0; j < total_iterations; ++j) {
  a1 = a1 + 1.0f;
}
```

Donde `total_iterations` es 640000000 por defecto y `a1` es una variable de tipo `float` almacenada en un registro `xmm`[20]. Cuando `num_vars` vale 2, el código equivalente sería:

[19] Las *pragmas* son construcciones que le indican al compilador cómo debe procesar un programa. Estrictamente hablando no forman parte de los lenguajes aunque se incluyen en la mayoría de los compiladores de C, C++ o Fortran. Al no ser parte estándar de los lenguajes, cada compilador soporta un conjunto diferente de pragmas (aunque algunas son comunes a varios compiladores). En C y C++ los pragmas se especifican mediante la directiva `#pragma` o la palabra clave `_Pragma`. En ocasiones, la funcionalidad de las pragmas también se puede conseguir mediante el uso de *atributos*. Los atributos son habitualmente también extensiones específicas de cada compilador, aunque C++ define una sintaxis estándar para los atributos desde la versión C++11 y un conjunto de atributos estándar que deben soportar todos los compiladores. Como veremos, las pragmas también se usan para implementar OpenMP.

[20] Se utilizan los registros vectoriales SSE de 128 bits, de los cuales hay 16 llamados `xmm0` a `xmm15`. Cada uno permite almacenar 4 floats de 32 bits. Sin embargo, se utilizan solo los primeros 32 bits de cada registro porque queremos operar con

Listado 2: Código ensamblador generado por GCC para `calculate<3>`, obtenido con `objdump -M intel -d data-hazards-gcc`. El bucle de interés va desde la etiqueta 401d28 a la 401d3a.

```
0000000000401d00 <_Z9calculateILi3EEvRfm>:
  401d00:       vxorps xmm4,xmm4,xmm4                   # inicializa a 0 xmm4
  401d04:       vinsertps xmm0,xmm4,xmm4,0xe            # inicializa xmm0 a 0 (a0)
  401d0a:       vmovaps xmm2,xmm0                       # inicializa xmm2 a 0 (a1)
  401d0e:       vmovaps xmm1,xmm0                       # inicializa xmm1 a 0 (a2)
  401d12:       test    rsi,rsi                         # rsi contiene total_iterations
  401d15:       je      401d3c <_Z9calculateILi3EEvRfm+0x3c>
  401d17:       xor     eax,eax                         # eax = 0 (j)
  401d19:       vmovss xmm3,DWORD PTR [rip+0x159ef]     # almacena 1.0f en xmm3
  401d20:                                               # nop
  401d21:       nop     DWORD PTR [rax+0x0]
  401d28:       inc     rax                             # j = j + 1
  401d2b:       vaddss xmm0,xmm0,xmm3                   # a0 = a0 + 1.0f
  401d2f:       vaddss xmm2,xmm2,xmm3                   # a1 = a1 + 1.0f
  401d33:       vaddss xmm1,xmm1,xmm3                   # a2 = a2 + 1.0f
  401d37:       cmp     rsi,rax                         # comprueba fin de bucle
  401d3a:       jne     401d28 <_Z9calculateILi3EEvRfm+0x28>
  401d3c:       vaddss xmm0,xmm0,xmm2                   # a0 = a0 + a1
  401d40:       vaddss xmm0,xmm0,xmm1                   # a0 = a0 + a2
  401d44:       vmovss DWORD PTR [rdi],xmm0             # result = a0
  401d48:       ret
  401d49:       nop     DWORD PTR [rax+0x0]
```

```
for (size_t j = 0; j < total_iterations; ++j) {
  a1 = a1 + 1.0f;
  a2 = a2 + 1.0f;
}
```

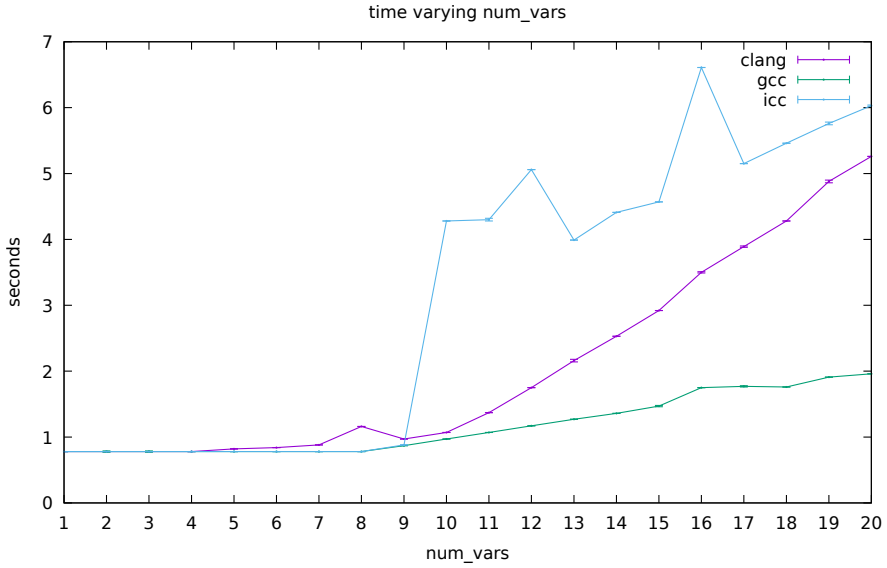
Donde `a2` es otra variables de tipo `float` almacenada en otro registro. El código generado para otros valores de `num_vars` es análogo. En el listado 2 se puede ver el código ensamblador generado para 3 variables.

Gracias a que el código se ejecuta en un procesador superescalar con ejecución fuera de orden y predicción de saltos, las instrucciones de control del bucle se ejecutarán en paralelo a las sumas y no incrementarán el tiempo de ejecución, como veremos en el tema 2. Por tanto, el tiempo de ejecución para 3 variables (por ejemplo) sería el mismo que si el programa fuera como sigue:

```
a1 = a1 + 1.0f;
a2 = a2 + 1.0f;
a3 = a3 + 1.0f;
a1 = a1 + 1.0f;
a2 = a2 + 1.0f;
a3 = a3 + 1.0f;
... (640000000 repeticiones en total)
a1 = a1 + 1.0f;
a2 = a2 + 1.0f;
a3 = a3 + 1.0f;
```

Podemos ver que existen dependencias RAW entre las instrucciones de suma de cada una de las variables, ya que el valor de entrada de cada iteración es el resultado de la anterior. Por tanto, estas

ellos de forma escalar. Esta es la forma habitual de trabajar con variables escalares de coma flotante en las arquitecturas x86 que soportan SSE. También existe un conjunto de instrucciones de coma flotante no vectoriales que usan una pila de registros de 80 bits cuyo origen es el coprocesador matemático 8087 que antiguamente complementaba al procesador 8086, pero estas instrucciones son obsoletas e Intel no recomienda su uso (los procesadores de Intel anteriores al 486DX no soportaban instrucciones de coma flotante si no estaban acompañados de un coprocesador).



vars	Tiempo (s)	GFLOPS
1	0.78±0.00	0.82±0.00
2	0.78±0.01	1.63±0.02
3	0.78±0.01	2.45±0.02
4	0.78±0.00	3.29±0.00
5	0.78±0.00	4.09±0.03
6	0.78±0.00	4.94±0.00
7	0.78±0.00	5.77±0.00
8	0.78±0.00	6.55±0.04
9	0.87±0.00	6.58±0.00
10	0.97±0.00	6.58±0.00
11	1.07±0.00	6.56±0.02
12	1.17±0.00	6.56±0.02
13	1.27±0.00	6.57±0.01
14	1.36±0.00	6.57±0.01
15	1.47±0.01	6.52±0.03
16	1.75±0.00	5.84±0.01
17	1.77±0.01	6.16±0.02
18	1.76±0.00	6.55±0.01
19	1.91±0.00	6.37±0.01
20	1.96±0.00	6.55±0.02

Figura 3: Resultados de `data-hazards.cpp` en `compiladorintel.inf.um.es`. En la tabla se muestan únicamente los resultados obtenidos con GCC por brevedad.

instrucciones deberán ejecutarse secuencialmente (es decir: hasta que no acabe la ejecución de la primera instrucción no puede empezar la ejecución de la segunda). Sin embargo, las instrucciones que operan sobre variables diferentes son totalmente independientes, por lo que podrían ejecutarse en paralelo.

Si el valor de `total_iterations` es suficientemente grande, podemos considerar que el tiempo necesario para ejecutar las instrucciones previas y posteriores al bucle es despreciable. Por tanto, para el caso de una sola variable, como las `total_iterations` sumas del bucle se ejecutarán secuencialmente debido a las dependencias RAW, si llamamos c al número de ciclos que tarda cada una de estas instrucciones, el tiempo de ejecución vendrá dado por la siguiente expresión:

$$T = \frac{total\_iterations \times c}{frecuencia_1}$$

Si ejecutamos el programa en el servidor de ejemplo, obtenemos los resultados que se muestran en la figura 3. Vemos que para el caso de 1 variable, el tiempo de ejecución es de 0,78±0.00 s. Substituyendo los valores conocidos en la ecuación anterior tenemos que:

$$0{,}78 \text{ s} = \frac{640000000 \times c}{3300 \text{ MHz}} \Rightarrow c = \frac{0{,}78 \text{ s} \times 3300 \times 10^6 \text{ Hz}}{640000000} = 4{,}02$$

Por lo que podemos deducir que cada operación de suma en coma flotante de simple precisión tiene una latencia de 4 ciclos en este procesador[21]. Obtenemos un valor ligeramente mayor a 4 debido a la sobrecarga introducida por el resto de instrucciones y por las propias instrucciones usadas para medir el tiempo de ejecución.

Estas unidades funcionales están segmentadas, por lo que cada una puede empezar a realizar una segunda suma en el ciclo siguiente a la primera, siempre y cuando no sean dependientes. Por tanto, una sola unidad funcional es capaz de estar ejecutando 4 sumas independientes a la vez, iniciando y terminando una cada ciclo. En efecto, podemos ver que el tiempo que tarda el programa es el mismo para 1, 2, 3 y 4 variables, a pesar de que el trabajo realizado es mayor en cada caso.

No solo podemos ejecutar 4 sumas en paralelo gracias a la segmentación de las unidades funcionales, sino que además podemos tener varias unidades funcionales. Podemos ver que en este procesador el tiempo de ejecución del programa es exactamente el mismo hasta 8 variables, y se incrementa cada vez más a partir de 9 variables (con los compiladores GCC e ICC). Por tanto, podemos ejecutar hasta

[21] Podemos comprobar que este es el valor correcto de acuerdo a la documentación del procesador, consultando por ejemplo la página 275 de `https://www.agner.org/optimize/instruction_tables.pdf`

Listado 3: Algoritmo básico de multiplicación de matrices 2D

```
void multiply_matrix(Matrix& C, const Matrix& A, const Matrix& B) {
  assert(A.height == C.height && B.width == C.width && A.width == B.height);
  for (size_t i = 0; i < A.height; ++i) {
    for (size_t j = 0; j < B.width; ++j) {
      C[i][j] = 0;
      for (size_t k = 0; k < A.width; ++k) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

8 sumas en paralelo, lo que nos indica que disponemos de 2 unidades funcionales de suma en coma flotante de simple precisión[22].

Con este programa podríamos analizar la latencia y el número de unidades funcionales de suma de cualquier procesador siempre que necesitemos como máximo 15 variables, ya que disponemos solo de 16 registros `xmm`, de los cuales uno se utiliza para almacenar la constante 1,0.

8. Bibliotecas HPC

Para finalizar esta primera práctica, vamos a mostrar la utilidad de usar bibliotecas de cálculo científico especialmente optimizadas para computación de altas prestaciones (HPC), mostrando cómo de esta forma se puede obtener un muy buen rendimiento de nuestro hardware invirtiendo muy poco esfuerzo, siempre que el algortimo que estemos utilizando se pueda aprovechar de alguna implementación disponible en una de estas bibliotecas.

Como ejemplo de aplicación, vamos a utilizar una operación muy común dentro del campo del álgebra lineal como es la multiplicación de matrices de 2 dimensiones. Recordemos que dos matrices A y B de dimensiones $m \times k$ y $k \times n$ respectivamente se pueden multiplicar dando como resultado una tercera matriz C de dimensiones $m \times n$, donde cada elemento $C_{i,j}$ de C se calcula como el producto escalar del vector fila i de A por el vector columna j de B. El número de columnas de A, por tanto, debe ser igual al número de filas de B.

El algoritmo estándar para la multiplicación de matrices es bien conocido y consiste en un triple bucle anidado, como se puede ver en el listado 3. Su complejidad algorítmica es de $O(m \times n \times k)$ (es decir: $O(n^3)$ en el caso de matrices cuadradas). Por tanto, cuando se usan matrices grandes (a partir de 1 millón de elementos de simple o doble precisión) es interesante optimizar el programa que implemente el algoritmo. Existen algoritmos con mejor complejidad asintótica, pero no se usan habitualmente debido a que en la práctica solo son más rápidos que el algoritmo básico para matrices muchísimo más grandes que las que se utilizan en cualquier aplicación.

Sin embargo, el algoritmo básico se puede optimizar (sin cambiar su complejidad algorítmica). Por ejemplo, cambiar el orden de los bucles anidados dará un comportamiento diferente con respecto a los patrones de acceso a la memoria al leer los datos de las matrices A y B y al escribir los resultados en los elementos de la matriz C. Es posible además mejorar la localidad de los accesos a memoria aplicando técnicas de *tiling*, lo que resulta fundamental si las matrices no caben en la caché. En esta práctica no vamos a describir en detalle estas optimizaciones, sino que nos limitaremos a usar una función de biblioteca ya optimizada.

Existen varias bibliotecas de algoritmos numéricos para HPC. Estas bibliotecas han sido optimizadas por especialistas a lo largo de mucho tiempo y utilizando varias plataformas, por lo que, normalmente, el rendimiento que proporcionen será mucho mejor que el que podamos obtener escribiendo nosotros mismos el código. Adicionalmente, si usamos estas librerías el código de nuestro programa será más sencillo y más portable. Las funciones incluidas en estas bibliotecas aceleran tareas tan simples como multiplicar

[22] Podemos comprobar que este resultado concuerda con lo indicado en `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)`

un vector por un escalar o tan complejas como encontrar mínimos en funciones multidimensionales o realizar una descomposición de Cholesky.

En el ámbito numérico, son especialmente importantes las implementaciones de BLAS, que es una librería cuyas primeras versiones se publicaron en 1979 en Fortran. BLAS se convirtió en un estándar de facto y su interfaz se fue ampliando hasta 1990 con funciones de álgebra lineal que operan con vectores y matrices. Otras librerías (como LAPACK) se basan en BLAS para ofrecer procedimientos de más alto nivel. A día de hoy existen múltiples implementaciones de BLAS optimizadas de diversas formas, como la implementación de referencia[23], OpenBLAS[24] o la incluida en MKL de Intel o en la librería GSL[25].

Por supuesto, existen muchas otras librerías de diferentes tipos de funciones diferentes a las incluidas en BLAS, como la ya mencionada MKL de Intel, la GSL, Blitz++, librerías para calcular transformadas discretas de Fourier (como FFTW). También existen bibliotecas optimizadas para algoritmos que no son estrictamente numéricos (como por ejemplo AMBAR, NAMD o GROMACS que son del ámbito bioinformático). La mayoría de las librerías ofrecen interfaces para C o C++, aunque algunas (como BLAS) definieron su interfaz originalmente en Fortran. Existen también librerías que ofrecen interfaces para otros lenguajes, como NumPy[26] que es muy popular en Python (aunque los algoritmos numéricos propiamente dichos de NumPy están implementados en C).

En esta práctica vamos a utilizar la librería BLAS. Concretamente, usaremos la función de nivel 3 `SGEMM` que permite multiplicar dos matrices entre sí[27]. Utilizaremos la implementación proporcionada por OpenBLAS, aunque el código sería el mismo que si usáramos otra implementación. Por ejemplo, si quisiéramos utilizar la implementación ofrecida por Intel en MKL deberíamos cambiar solo un *include* en el código fuente y la línea de compilación del `Makefile` (pero solo podríamos utilizar ICC, ya que MKL no es portable).

Para esta sección, vamos a utilizar el programa de multiplicación de matrices de números en coma flotante de simple precisión que se encuentra en el subdirectorio `HPC-library`. Este programa incluye cinco implementaciones de la multiplicación de matrices que se pueden ver en el fichero `multiply_matrix.cpp`. El resto de ficheros se encargan de crear las matrices, llamar al procedimiento de multiplicación que se indique mediante la opción `--implementation`, medir el tiempo y calcular el rendimiento. Como el objetivo del programa es únicamente medir el rendimiento de las diferentes implementaciones, las matrices se inicializan siempre de forma aleatoria. Las tres implementaciones son las siguientes:

`basic`: Versión básica, equivalente al listado 3. Alternativamente, podríamos llamar a esta versión *ijk* si nos fijamos en el orden de los bucles anidados.

`ikj`: Versión en la que se han intercambiado los dos bucles interiores de la versión `basic` para mejorar la localidad de los accesos a memoria.

`basic_omp`: Como la versión `basic`, pero paralelizada utilizando OpenMP.

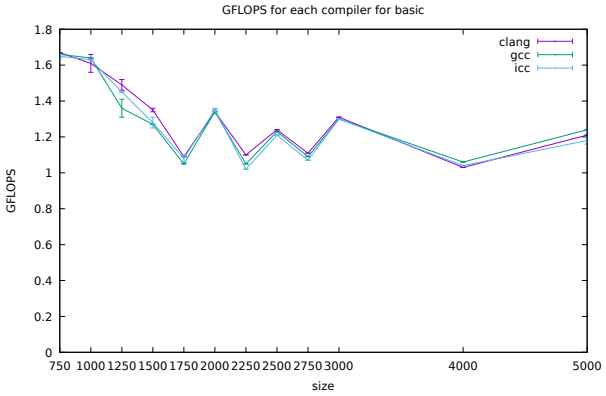
`basic_ikj`: Como la versión `ikj`, pero paralelizada utilizando OpenMP.

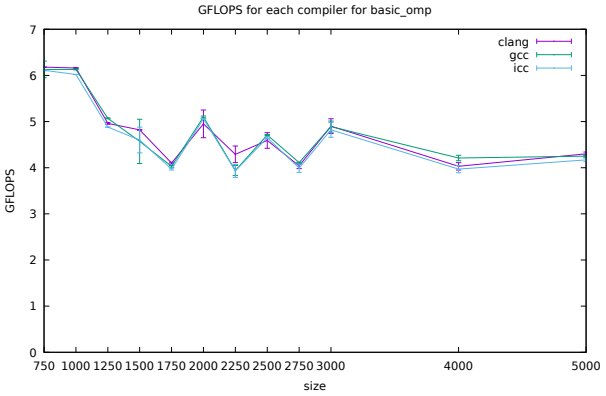
`blas`: Versión utilizando la función `cblas_sgemm` de la librería BLAS.

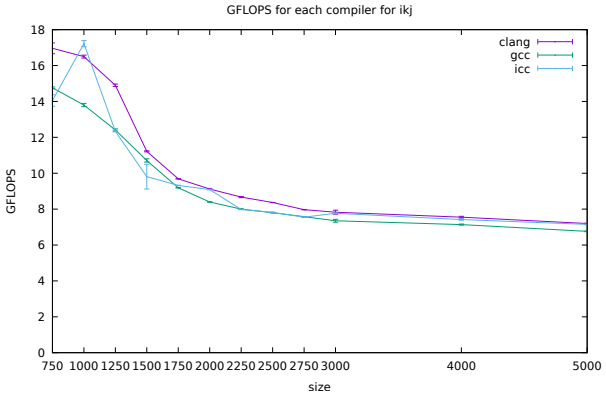
Igual que en otras secciones de la práctica, el programa se puede compilar con los tres compiladores GCC, ICC y Clang mediante el `Makefile` incluido. Al contrario de lo que ocurría en la primera sección, veremos que para este programa la diferencia de rendimiento entre los tres compiladores es mucho más pequeña.

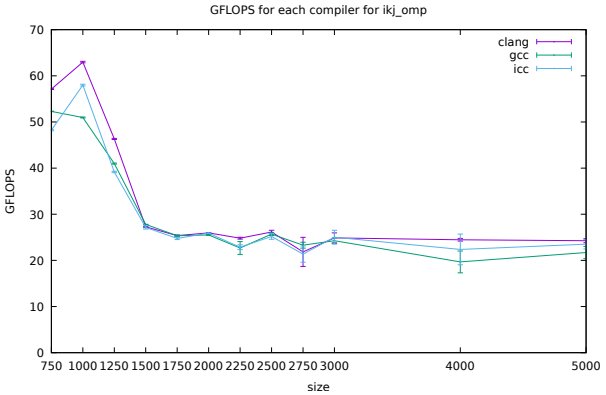
Una vez compilado el programa (con «`make all`») podemos probar el rendimiento que se obtiene al multiplicar matrices de diferentes tamaños con la opción «`--square-size=`*n*» y usando diferentes implementaciones con la opción «`--implementation=`*i*». Por ejemplo, usando «`./matrix-gcc`

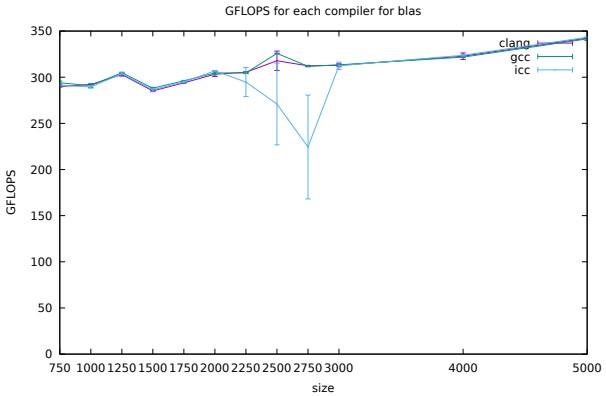
[23] `http://www.netlib.org/blas/`
[24] `https://www.openblas.net/`
[25] `https://www.gnu.org/software/gsl/`
[26] `https://numpy.org/`
[27] La documentación de esta función se puede consultar en `http://www.netlib.org/blas/#_level_3`.











	basic	ikj	basic_omp	ikj_omp	blas
750	1.66±0.00	14.76±0.05	6.13±0.18	52.29±0.05	294.02±1.83
1000	1.64±0.00	13.80±0.09	6.13±0.01	50.97±0.10	290.84±1.95
1250	1.36±0.05	12.41±0.06	5.07±0.01	40.98±0.11	304.90±0.56
1500	1.27±0.00	10.71±0.09	4.57±0.48	27.80±0.01	288.12±0.92
1750	1.05±0.00	9.19±0.01	4.03±0.02	25.34±0.12	296.09±0.44
2000	1.34±0.01	8.40±0.01	5.10±0.02	25.52±0.09	304.61±1.93
2250	1.05±0.00	8.00±0.02	3.94±0.11	22.69±1.39	304.70±0.64
2500	1.23±0.00	7.80±0.01	4.71±0.01	25.70±0.31	325.79±0.23
2750	1.09±0.00	7.58±0.01	4.11±0.01	23.30±0.64	311.77±0.45
3000	1.30±0.00	7.35±0.08	4.89±0.12	24.27±0.60	313.43±0.48
4000	1.06±0.00	7.14±0.03	4.21±0.06	19.67±2.36	321.79±0.32
5000	1.24±0.00	6.76±0.02	4.25±0.03	21.70±1.32	341.49±1.71

Figura 4: Resultados de las diferentes implementaciones de `multiply_matrix.cpp` en `compiladorintel.inf.um.es` para diferentes tamaños. En la tabla se muestran únicamente los resultados de GCC por brevedad.

`--square-size=600 --implementation=basic_omp`» en el servidor de ejemplo podemos comprobar que se obtiene un rendimiento de 6,21±0,03 GFLOPS. Para mejorar este rendimiento, se pueden utilizar diversas técnicas entre las que destacan la paralelización, la vectorización y la optimización de los accesos a memoria, que se comentarán en las siguientes prácticas. En esta práctica, mostramos a modo de ejemplo el resultado de la aplicación rápida de la paralelización con OpenMP y una optimización sencilla de los accesos a memoria para compararlo con la versión optimizada ofrecida por BLAS. En la figura 4 se muestran los resultados obtenidos por las diferentes implementaciones para varios tamaños.

Salta a la vista que el rendimiento obtenido por la implementación `basic` se queda muy alejado del rendimiento pico que podemos obtener de un núcleo del servidor. La implementación `ikj` obtiene resultados algo mejores gracias al incremento de la localidad y la consecuente reducción en el número de fallos de caché. Las versiones paralelas de estas dos implementaciones casi multiplican el rendimiento por 4 (que es el número de núcleos de nuestro procesador), mostrando que, para algoritmos tan sencillos como este, la paralelización con OpenMP permite aprovechar fácilmente el paralelismo a nivel de hilo (obsérvese que la paralelización ha consistido únicamente en añadir una línea al programa en cada caso). Sin embargo, el rendimiento de las implementaciones mencionadas anteriormente no se acerca al obtenido por la función `sgemm`, que es de 341.49±1.71 en el caso de matrices de 5000×5000. Aún así, nos quedamos en el 86 % del rendimiento pico, lo cual no es sorprendente ya que dicho rendimiento pico es una cota superior difícilmente alcanzable fuera de aplicaciones *de juguete* especialmente programadas para ello.

9. Ejercicios pedidos

Elige un ordenador diferente de `compiladorintel.inf.um.es` y contesta a las siguientes preguntas:

1. Detalla las características relevantes del ordenador elegido (microarquitectura de CPU y memoria).
2. Calcula el rendimiento pico teórico para operaciones en coma flotante de simple precisión que puede alcanzar tanto usando un solo núcleo como usando todos los disponibles (no se debe tener en cuenta el uso de SMT/HyperThreading si el procesador lo soporta).
3. Calcula ancho de banda pico que puede alcanzar el sistema de memoria.
4. Mide el rendimiento pico en coma flotante de simple precisión del ordenador elegido utilizando los benchmarks `peak-flops.cpp` y `peakperf` tanto para el caso de 1 hilo como usando tantos hilos como núcleos disponibles. Explica los resultados obtenidos.
5. Mide el ancho de banda pico de memoria del ordenador elegido utilizando el benchmark `peak-mem.cpp`. Explica los resultados obtenidos.
6. Calcula la latencia de las sumas en coma flotante de simple precisión y trata de determinar de cuántas unidades funcionales dispone cada núcleo para esta operación.
7. Explica los resultados numéricos obtenidos en la ejecución de `data-hazards.cpp` (el valor mostrado en cada caso como «`result = ...`», no el tiempo medido).

9.1. Entrega

Tras finalizar los ejercicios anteriores, se pide entregar lo siguiente:

- Un documento en formato PDF que será el documento principal utilizado a la hora de evaluar la práctica. Dicho fichero debe incluir:
  - Información completa y fehaciente sobre la autoría de la práctica.
  - Las respuestas a los ejercicios solicitados.

  - Un breve informe comentando los aspectos positivos de la práctica, así como cualquier aspecto negativo y cosas que has echado en falta en la misma. La longitud del informe no debe exceder las 2000 palabras

- El código fuente de cualquier programa o script desarrollado (o modificado) para la realización de la práctica. Estos ficheros se mirarán opcionalmente para comprobar cualquier aspecto que no esté claro en el documento PDF.
- Un fichero de texto llamado `README` identificando todos los ficheros fuente incluidos con instrucciones claras para su compilación.

Los ficheros solicitados se deben entregar en un único archivo comprimido en formato `.tar.gz` o `.zip`. No se admitirán otros formatos distintos a los indicados.

Para el envío, se tiene que utilizar la herramienta de *Tareas* del Aula Virtual y envíar el trabajo antes de la fecha prevista. Se permite reenviar dicho fichero dentro del plazo establecido, por si después de haberlo enviado se detectan errores.

9.2. Recursos

En la zona de Recursos del Aula Virtual, dentro de la carpeta denominada *Practicas*/*Practica*1 puedes encontrar todo lo necesario para realizar esta práctica.

10. Criterios de evaluación

Esta práctica se evaluará teniendo en cuenta los siguientes criterios de evaluación (sobre 10 puntos):

- Realización y corrección de los apartados pedidos (*6 puntos*) .
- Presentación y claridad de las explicaciones (*1 punto*).
- Aportación de ideas originales en las explicaciones a los apartados realizados (*1 punto*).
- Realización de alguna actividad *extra* relacionada con los ejercicios de la práctica demostrando curiosidad (*1 punto*).
- Comentarios sobre la práctica, incluyendo aspectos negativos y positivos (*1 punto*).

Créditos y recursos adicionales

- Para la sección 6 de cálculo del rendimiento pico, es muy útil revisar el artículo « *Theoretical peak FLOPS per instruction set: a tutorial*», Dolbeau, R., Journal of Supercomputing (2018) 74: 1341. `https://doi.org/10.1007/s11227-017-2177-5`.
- Para entender las funciones intrínsecas usadas por `peakperf` se puede consultar la guía de Intel al respecto en `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, junto con alguna referncia del ISA x86-64 como `https://www.amd.com/system/files/TechDocs/40332.pdf` o `https://www.felixcloutier.com/x86/`.
- Asimismo, en la página web de Agner Fog (`http://www.agner.org/optimize/`), Catedrático de la *Technical University of Denmark, Department of Information Technology, Copenhague* (Dinamarca), hay muchos documentos acerca de la microarquitectura de los procesadores x86.
- En `https://support.pawsey.org.au/documentation/download/attachments/2162899/Optimising%20Serial%20Code.pdf?api=v2` hay disponible una presentación útil como introducción a la optimización de aplicaciones secuenciales.
- El profesor de prácticas de la asignatura *Pablo Antonio Martinez Sanchez <pabloantonio.martinezs@um.es>* es el autor de `peakperf` y `cpufetch`.