Universidad de Murcia
Facultad de Informática

Grado en Ingeniería Informática
3 Curso

Arquitectura y Organización de Computadores

Práctica 1 - Optimización de una aplicación secuencial

Curso académico 2020-21

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

1. Introducción

Es frecuente entre los estudiantes de informática, e incluso entre los profesionales del sector, que se siga considerando una idea surgida al inicio de la década de los años 60, cuando surgieron los primeros compiladores. Dicha *leyenda urbana* la podríamos enunciar así: "Un programador con experiencia escribiendo una aplicación en lenguaje ensamblador puede superar cualquier código producido por un compilador". Afortunadamente, este ya no es el caso. Los compiladores actualmente son muy buenos, por lo que si proporciona código fuente de alta calidad, el resultado obtenido probablemente será mejor que cualquier resultado obtenido 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.

Pero para lograr ese mejor resultado, hay muchas cosas que el programador debe hacer para ayudar al compilador a realizar un mejor trabajo optimizando el código. O al menos hay muchas cosas que se pueden hacer para no obstaculizar los intentos del compilador de optimizar el código.

Esta práctica trata de ayudar a entender cómo eliminar los malos hábitos y hacer las cosas bien para facilitar la colaboración del programador y el compilador en la producción de código eficiente (rápido). Incluso muestra cuando se debería utilizar una biblioteca de alto rendimiento en lugar de código propio, pues una buena biblioteca es difícil de superar.

El proceso de construcción de una aplicación científica comienza con el desarrollo de un algoritmo, su implementación en un lenguaje dado, y termina con una pieza específica de software. En cada etapa del proceso, se pueden tomar decisiones o acciones que pueden influir fuertemente en el rendimiento final del software. Dicho rendimiento final 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, la cantidad de memoria que necesita para su ejecución, etc. En la Figura 1 ofrecemos una vista de dicho proceso.

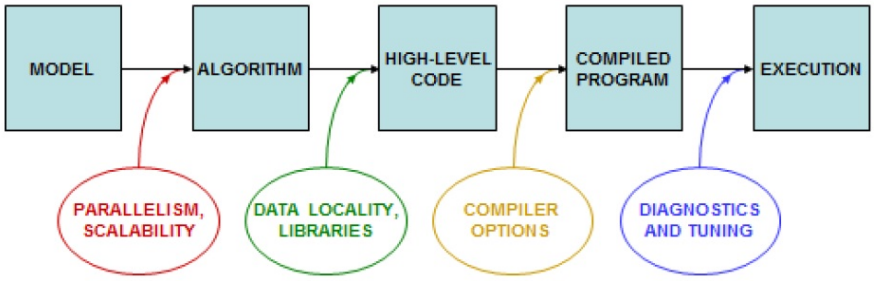

Figura 1: Estados en el desarrollo de software

La idea general de la optimización del código de una aplicación es instrumentar el código fuente por medio de llamadas a funciones para medir el tiempo para descubrir sus puntos calientes (*hot spots*) -esto es, dónde se gasta la mayor parte del tiempo de computación-, y luego dedicar especial atención a optimizar dichas zonas de código para que se ejecuten más rápido. Hay bastantes herramientas que se han creado para ayudar en este proceso, y pueden ser tan simples como `gprof`, o tan complejas como `Intel VTune`.

Alcanzar un buen rendimiento global depende de algo más que escribir un programa y asegurarse de que está dispuesto correctamente para la arquitectura destino. También depende de una serie de factores importantes a tener en cuenta, como son:

a. Uso de las opciones de compilación adecuadas

b. Evitar añadir en el código riesgos estructurales, de datos o de control

c. Emplear bibliotecas HPC[1] optimizadas siempre que sea posible en el programa

Estos principios están interrelacionados. Por ejemplo, un buen diseño de memoria y un patrón de acceso favorable pueden producir un código razonablemente rápido, pero para hacerlo lo más rápido

[1]Del inglés *High Performance Computing*, Computación de Altas Prestaciones.

posible, el compilador también puede tener que recibir los *flags*[2] correctos. Alternativamente, las mejores estructuras de datos pueden resultar ser las que son compatibles con una determinada biblioteca HPC, por lo que puede aprovechar la excelente organización de la memoria y los *flags* de compilador especializados que ya están incorporadas en el software.

Esta primera práctica trata de ayudar a conocer las características comunes de la microarquitectura de los procesadores multinúcleo actuales, y ayuda a tener una idea de cómo el compilador trata de optimizar las instrucciones para usar dichas características cuando se le da un determinado fragmento de código. La práctica se centra en aplicar aspectos básicos de la optimización de código que pueden tener un gran impacto, como el uso de algunos hábitos que evitan los principales problemas en el rendimiento, el uso de bibliotecas, el conocimiento y uso de los diversos *flags* del compilador, junto con algunos códigos de escritura.

Las prácticas segunda y tercera ampliarán estos conceptos por medio de explotar las capacidades vectoriales y multinúcleo que tienen los procesadores actuales (práctica 2), y por medio de describir las 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, incluidas las plataformas HPC. Se trata de poner de manifiesto las relaciones básicas existentes entre los lenguajes de alto nivel, los compiladores empleados, el repertorio de instrucciones máquina (ensamblador) de un procesador (ISA) y la arquitectura específica de dicho procesador, de cara a la ejecución de aplicaciones de altas prestaciones.

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 o CMP[3].
- Mostrar al alumno las implicaciones básicas entre el lenguaje de alto nivel empleado en la codificación de las aplicaciones, el compilador utilizado y los diversos *flags* del compilador en función de la arquitectura del procesador en el que se trabaja.
- Conocer el problema de los riesgos de datos en las variables de una aplicación, y cómo afecta al rendimiento obtenido por la aplicación.
- Afianzar la programación de aplicaciones con necesidades de altas prestaciones a través de un algoritmo típico de cálculo numérico. Uso de librerías optimizadas para HPC.

3. Entorno de evaluación

La realización de las prácticas por parte de los alumnos se puede hacer en el servidor proporcionado por el centro de cálculo `compiladorintel.inf.um.es`. Dicho servidor tiene instalado el compilador GNU GCC (version 4.8.4) y el compilador de INTEL ICC (version 16.0.0). Ten en cuenta que para poder ejecutar el compilador de Intel debes ejecutar primeramente el siguiente comando `source /opt/intel/bin/iccvars.sh intel64` (se recomienda que lo añadas a tu *bash* de inicio para no tener que ejecutarlo cada vez que empieces una sesión).

De forma alternativa, las prácticas las puedes realizar en tu propio ordenador. Para ello, se debe solicitar a Intel una licencia gratuita de su compilador (por ser estudiante), y eso hay que hacerlo en la siguiente dirección: `https://software.intel.com/en-us/qualify-for-free-software/student`.

En primer lugar, el alumno necesita conocer adecuadamente las características básicas de la CPU y el sistema de memoria del ordenador utilizado. Para ello, se puede emplear el comando `lshw` (si está instalado), o revisar los ficheros `cpuinfo` y `meminfo` que se encuentran habitualmente en `/proc`, o ejecutar el comando `lscpu`. En concreto, las características del servidor `compiladorintel.inf.um.es`,

[2] Son las diversas opciones de compilación disponibles.

[3] Del inglés *Chip MultiProcessor*, Multiprocesador en un chip.

que tiene una microarquitectura Skylake, son: Intel® Core™ i5 CPU 2.70GHz (hasta 3.3GHz en modo turbo), con 4 cores (sin HT), comenzado a vender en Q3'15, con litografía de 14 nm, y con 32KiB de caché L1, 256KiB de L2, y 6MiB de caché L3 compartida, con TDP de 65 W, y con ISA soportado de SSE4.1/4.2, AVX2.

Para consultar estas y otras características de los procesadores, puedes mirar en las siguientes páginas web: la de Intel (`https://ark.intel.com/products`, o bien la de wikichip (`https://en.wikichip.org/wiki/intel`). En nuestro caso, las páginas son las siguientes: `https://ark.intel.com/products/88185/Intel-Core-i5-6400-Processor-6M-Cache-up-to-3_30-GHz`, y: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)`.

Además, es importante conocer la versión del sistema operativo que se está empleando, para lo que se puede ejecutar el comando `hostnamectl` o `cat /etc/os-release`. El servidor de prácticas, por ejemplo, tiene instalado el SO Ubuntu release 14.04. Debido a que Linux es un SO multitarea y multiusuario, para obtener unos tiempos lo más precisos posibles, se recomienda ejecutar cualquier aplicación cuando el ordenador tenga la carga más baja posible (y sin otros usuarios ejecutando otras aplicaciones); si estás usando tu propio ordenador, te recomiendo que ejecutes la orden `systemctl isolate multi-user.target`. Además, lo habitual es ejecutar para cada caso el programa un número determinado de veces (5 veces o más), y obtener el tiempo de respuesta como la media del tiempo de ejecución de dichas pruebas.

4. Rendimiento y ancho de banda pico

Para optimizar adecuadamente una aplicación, tenemos que manejar los conceptos de rendimiento del procesador y de su ancho de banda a memoria. En aplicaciones de cálculo científico, definimos el rendimiento de un procesador como el número de operaciones en coma flotante por segundo que realiza. Como esperamos que sean muchas, lo habitual es usar un múltiplo de este valor, habitualmente MFLOP/s (10^6) o GFLOP/s (10^9). De cara a hacernos una idea de si el resultado obtenido es bueno o no, es conveniente calcular el rendimiento teórico pico para dicho procesador. Para ello, consideramos que usamos todos los núcleos posibles del procesador haciendo uso de sus unidades vectoriales.

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

$$\begin{aligned} Rendimiento_{pico}(GFLOP/s) &= (Flop/operation) \times (\#\ operations/instruction) \times (\#\ Instructions/cycle) \\ &\quad \times Freq.(GHz) \times (\#\ cores/socket) \times (\#\ sockets/node) \end{aligned}$$

Cada término representa lo siguiente:

- Flop/operation: 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 hay la instrucción `FMA` (*Fused Multiply Add*) es capaz de realizar en un ciclo de reloj una operación de suma y una operación de multiplicación en coma flotante a la vez;[4]

- # operations/instruction: 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 depende del ancho de la unidad vectorial.

- # Instructions/cycle: es el número de instrucciones que puede ser ejecutado simultáneamente en un ciclo. Aunque es un párametro complejo de conocer (está relacionado con el cauce de cada núcleo y el número de unidades funcionales que tiene), en la actualidad se puede aproximar por el número de unidades vectoriales que tiene cada núcleo.

- Freq. (GHz): representa la frecuencia a la que operan cada uno de los núcleos del procesador. Es de nuevo un parámetro complicado, pues el fabricante suele darnos la frecuencia base (o TDP) y

[4] La mayoría de procesadores con repertorio de instrucciones a partir de AVX2 implementan esta instrucción.

la frecuencia turbo[5], pero ambos valores asumen que el procesador está ejecutando instrucciones escalares. En el caso de estar ejecutando instrucciones vectoriales, hay otros valores *base AVX* y *turbo AVX* que suelen tener una pérdida de entre el 20-30 % de los valores escalares.

- # cores/socket: es el número de núcleos físicos (reales) que tiene cada socket del procesador. Hay que tener en cuenta que aunque un núcleo tenga la capacidad de HT (*hyperthreading*) esto no influye en este parámetro.

- # sockets/node: es el número de sockets que tiene el procesador.

En el caso del servidor del laboratorio, se trata de una arquitectura *skylake* que tiene la operación de FMA, con unidades vectoriales de 256 bits que para números reales en simple precisión equivalen a **8** operaciones en coma flotante de Simple Precision (tamaño 32 bits) por cada instrucción, donde cada núcleo tiene dos unidades vectoriales, tiene una frecuencia turbo AVX de 3,1 Ghz, tiene 4 núcleos y un único socket, por lo que obtenemos el siguiente rendimiento pico:

$$\begin{aligned} Rendimiento_{pico}(SP) &= 2\ (FMA\ operations) \times 8\ (SP\ SIMD\ Lane) 2\ (\#VectorLanes(Add, Mul)) \times \\ &\quad \times 3{,}1\ (GHz) \times 4\ (\#\ cores) \\ &\sim 396{,}8\ GFLOP/sec \end{aligned}$$

Por otra parte, hay otro tipo de aplicaciones muy utilizadas que no se pueden evaluar por su rendimiento pico en operaciones de coma flotante, pues están limitadas por el ancho de banda a memoria disponible que tiene nuestra plataforma hardware. Para ese tipo de aplicaciones, es muy útil manejar el concepto de ancho de banda a memoria, definido como el número de bytes por segundo que es capaz de transferir entre la memoria y el procesador. Como esperamos que sea un valor grande, lo habitual es usar un múltiplo de este valor, habitualmente MBytes/s (10^6) o GBytes/s (10^9). De cara a hacernos idea de si el resultado obtenido es bueno o no, es conveniente 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 el ancho del bus de memoria y su frecuencia. La formula que empleamos es la siguiente:

$$Bandwidth_{pico} GB/sec = (\#\ Controlador\ Memoria) \times (\#\ Canales/Control.) \times (\#\ Bytes) \times Freq.-Bus-Mem\ (GT/s)$$

Para obtener estos valores, lo primero que tenemos que hacer es conocer la cantidad de memoria que tiene instalado nuestro ordenador. Esto lo podemos obtener ejecutando el comando `more /proc/meminfo` (o también el comando `free -m`). En nuestro caso, tenemos 8 GiB de memoria RAM instalada. Para conocer su disposición y calcular el ancho de banda, necesitamos ejecutar el comando `$ sudo lshw -class memory` o el comando `$ sudo dmidecode -type 17`[6]. En el caso del servidor del laboratorio, 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 de 64 bits (8 bytes), siendo la memoria instalada un único DIMM de 8 GiB de 2133 MHz. A partir de la información mostrada, vemos que estamos utilizando un único canal de memoria, por lo que el ancho de banda pico para esta configuración es la siguiente:

$$Bandwidth_{pico} = 1\ (Controlador\ Memoria) \times 1\ (Canales/Control.) \times 8\ (Bytes) \times 2{,}133\ (GT/s) \sim 17{,}1\ GB/sec$$

A diferencia de los núcleos de procesamiento que tienen un comportamiento y unas características de rendimiento bastante predecibles, el rendimiento del subsistema de memoria se ve afectado por la

[5] la cual se alcanza únicamente cuando uno o pocos núcleos están en funcionamiento; si están todos los núcleos funcionando puede haber una pérdida entre un 10-20 %.

[6] Estos comandos sólo los podemos ejecutar en máquinas donde tengamos permisos de administrador.

implementación hardware, el ruido electromagnético en los "cables", y una variedad de otros factores complejos del sistema. Por ello, aunque utilicemos un *benchmark* adecuado, sólo podemos esperar alcanzar un ancho de banda pico efectivo en las aplicaciones reales del orden del 50 al 70 por ciento de la velocidad de transferencia máxima especificada.

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. A continuación, en este apartado de la práctica vamos a usar sencillos (*benchmarks*) para probarlo. Utilizaremos los siguientes *benchmarks* que encontrarás en el subdirectorio *peak-performance*:

helloflops1 Ejemplo de uso: y = mx + b; escala el 1er array y lo suma en un 2º array (SP).

helloflops2 Modificación de helloflops1 que usa OpenMP para ejecutarlo en todos los núcleos (SP).

hellomem Mide el ancho de banda de memoria en la copia de dos arrays (DP).

copy Similar al anterior, una versión diferente y algo más rápida (DP).

Vamos a empezar utilizando el *benchmark helloflops1*. Ejecuta la orden `$ icc -Wall -O2 -xCORE-AVX2 -o helloflops1 helloflops1.c` para compilar el programa. Si ejecutas el programa compilado, observaras que obtenemos un rendimiento entre 50-60 GFLOP/s, que es aproximadamente un 60 % del máximo que podríamos obtener de una aplicación ejecutándose en un único núcleo.

Para obtener el rendimiento máximo del procesador (lo más cercano al rendimiento pico), vamos a utilizar el *benchmark helloflops2* que está preparado para escalar a todos los núcleos. Ejecuta la orden `$ icc -Wall -O2 -qopenmp -xCORE-AVX2 -o helloflops2 helloflops2.c` para compilar el programa. La aplicación está preparada para aceptar como parámetro el número de hilos que deseamos usar (por defecto 4 si no se pone ningún parámetro). Como observaras, para 4 hilos hemos obtenido un rendimiento pico en torno a los 370 GFLOP/s (alrededor del 90 % del rendimiento máximo). Prueba a ejecutar la aplicación con distinto número de hilos.

Para medir el ancho de banda de memoria real, vamos a hacer uso de los otros dos *benchmarks* que tenemos, *hellomem* y *copy*. Si compilamos con la orden `icc -Wall -O2 -qopenmp -xCORE-AVX2 -o hellomem hellomem.c` y `icpc -Wall -O2 -qopenmp -xCORE-AVX2 -o copy copy.cc`, y ejecutamos los ficheros binarios[7], obtenemos un valor en torno a los 12 GB/s. Como vemos, estamos en el entorno del 60-70 % del valor teórico que habíamos calculado anteriormente.

Nota: Para medir el tiempo de ejecución, hemos utilizado la función *dtime()* dada en el programa, aunque también lo podríamos haber hecho a través del comando *time* (el tiempo (*user*) obtenido sería el que tomaríamos para el cálculo de los GFLOPs).

5. Compiladores y opciones de compilación

Como se mencionó anteriormente, el código generado por el compilador generalmente supera al código ensamblador escrito a mano. Sin embargo, los compiladores no son (todavía) tan inteligentes que no necesiten ninguna guía del programador humano. El rendimiento obtenido por una aplicación depende mucho del compilador utilizado y las opciones de compilación que se han usado. Una forma de guiar al compilador es a través de las diversas opciones de compilación que están disponibles (denominadas *flags*[8]). Habitualmente se introducen en la línea de comandos, y hay que tener en cuenta que como algunas de ellas están destinadas a ser específicas de la arquitectura destino, es muy recomendable echar un vistazo a los *makefiles* que se distribuyen con el software, pues a menudo hay por defecto *flags* del compilador que están en conflicto con una nueva arquitectura.

Algunas opciones de compilación se han convertido en estándar, por lo que funcionarán en la mayoría de los compiladores[9]. Esto es particularmente cierto en el indispensable *flag* `"-O"`(O mayúscula), el cual

[7] Nota que *hellomem* es bastante más lento para obtener el mismo resultado.

[8] También denominadas a veces *switches* o *knobs*.

[9] La opción -Wall es muy interesante porque nos ofrece los *warnings* del compilador. Es especialmente importante si queremos que el compilador optimice al máximo y lo haga correctamente.

permite sin entrar en datalles el esfuerzo de optimización que realizará el compilador. Casi todos los compiladores soporatan al menos 4 niveles (`-O0`, `-O1`, `-O2`, `-O3`), aunque su significado concreto depende de cada compilador. En el caso del compildor de Intel son:

`-O0` para la compilación rápida sin optimización.

`-O1` para una optimización limitada que no aumenta el tamaño del código (se excluye el *inlining*).

`-O2` para optimización moderada incluyendo vectorización. El soporte para depuración se conserva si se especifica -g.

`-O3` para la optimización agresiva, necesitando un tiempo más largo de compilación, y busca mejorar la velocidad del código ejecutable a costa de su tamaño. Esto puede cambiar la semántica del código y, ocasionalmente, los resultados. El nivel de optimización más agresivo `-O3` hace más *prefetching* y más transformaciones de bucles. También permite las optimizaciones presentes en `-O2`.

El nivel frecuentemente preferido es `-O2`. Este nivel de optimización permite cosas como:

- Planificación de instrucciones: reordenando las instrucciones para evitar los riesgos en el cauce del procesador.
- Propagación de copias: sustitución de variables en expresiones por sus valores reales.
- Software *pipelining*: ejecución de varias etapas de un bucle al mismo tiempo.
- Eliminación común de subexpresiones: encontrar expresiones idénticas para calcularlas sólo una vez.
- *Prefetching* - solicitando explícitamente datos antes de que sea necesario, para que los datos estén listos sin provocar detenciones en el cauce del procesador.
- Transformaciones de bucle - fisión, fusión, intercambio, inversión, embaldosado (*tiling*), desenrollado, movimiento de código constante fuera de un bucle exterior, etc.

Suele ser muy útil agregar un `flag` para que el compilador sepa que también debe tratar de optimizar el código para un modelo particular de procesador. Las opciones estándar recomendadas para el compilador de Intel `icc` son: el `flag -xhost`, para que el código se genere específicamente para el procesador donde se está compilando el programa. Si se quiere hacer compilación cruzada, lo normal es especificar el tipo de procesador, como por ejemplo -xCORE-AVX2 (para procesadores Intel con microarquitectura *Core* e instrucciones AVX2)[10].

Una vez descrito el indispensable *flag* "`-O`"del compilador, vamos a ver cómo influye en el compilador icc de Intel. En primer lugar, comentar que el compilador icc de Intel por defecto añade el *flag* `-O2`. Si compilamos `$ icc -Wall -O0 -o helloflops1-O0 helloflops1.c` y ejecutamos el fichero obtenido, obtenemos un rendimiento inferior a 1 GFLOP/s. Si ahora compilamos con `$ icc -Wall -O2 -o helloflops1-O2-noavx2 helloflops1.c` y ejecutamos el fichero obtenido, obtenemos un rendimiento en torno a 14 GFLOP/s. La razón de la diferencia respecto al rendimiento de la sección anterior está en que el compilador icc con el *flag* `-O2` también vectoriza el código, pero en este caso no le hemos dicho que pueda utilizar las instrucciones AVX2. Para lograr de nuevo el rendimiento inicial (en torno a los 60 GFLOP/s), le tenemos que decir al icc que el procesador destino soporta las instrucciones AVX2 (añadiéndole el `flag` -xCORE-AVX2). Para acabar de cerrar el círculo, podemos ver qué pasa si le decimos al compilador icc que no vectorice, compilando de la siguiente forma: `icc -Wall -O2 -xCORE-AVX2 -no-vec -o helloflops1-O2-novec helloflops1.c` Como podemos observar, obtenemos un rendimiento en torno a 4 GFLOP/s, por lo que podemos darnos cuenta de la importancia que tiene usar las instrucciones vectoriales en los procesadores actuales.

[10]Ten en cuenta que estas opciones pueden variar de un compilador a otro. Por ejemplo, en el caso del compilador GNU gcc, lo normal es poner -march = native para el caso de usar el mismo procesador, y -march=core-avx2 para el caso de compilación cruzada

6. Programación: Evitar los riesgos de datos

En esta sección, vamos a ver como una aplicación mal programada puede obtener un pobre resultado en un procesador actual. Las pruebas las vamos a realizar con código secuencial ejecutándolo en un sólo núcleo, pero lo dicho es aplicable a una aplicación que se ejecutara en paralelo usando varios núcleos. Para medirlo vamos a utilizar los siguientes *benchmarks* que puedes encontrar en el subdirectorio *data-hazards*:

sumaflops1 Versión secuencial con *una* variable de acumulación (reducción) de Simple Precision (SP).

sumaflops2 Versión con *dos* variables de acumulación de Simple Precision (SP).

...

sumaflops16 Versión con *dieciséis* variables de acumulación de Simple Precision (SP).

Como habíamos visto en la sección anterior, para un núcleo obteníamos un rendimiento en torno a los 60 GFLOP/s. Vamos a empezar con el *benchmark sumaflops1*. Compilamos (`$ icc -Wall -O2 -xCORE-AVX2 -o sflops1 sumaflops1.c`) y ejecutamos, obteniendo un rendimiento de 0,8 GFLOP/s.

¿A qué es debido este pobre resultado? Para interpretarlo correctamente, tenemos que suponer que el código obtenido no está vectorizado (ambos compiladores ofrecen el mismo resultado, y sería el mismo que obtendríamos si compilaramos con la opción `-no-vec`), por lo que el compilador `icc` debería ofrecer la octava parte del rendimiento para un núcleo (es decir, algo más de 3 GFLOP/s).

Pero como hemos visto, el programa ejecutable solo obtiene 0,8 GFLOP/s. ¿A qué es debido? La razón de este comportamiento la podemos entender si ponemos seguidas dos iteraciones del bucle:

```
suma1 = suma1 + 1.0;
suma1 = suma1 + 1.0;
```

Como podemos ver, la variable de entrada de la segunda instrucción (`suma1`) es la misma que la variable de salida de la primera instrucción (`suma1`). lo que provoca una dependencia de datos real (tipo RAW) en la variable `suma1`. Dicha dependencia de datos hace que hasta que no acabe la ejecución de la primera instrucción no puede empezar la ejecución de la segunda, y ello hace que me encuentre con la latencia de la unidad funcional que estoy usando (en este caso, la de suma/multiplicación de números en coma flotante). Como hemos obtenido una cuarta parte del rendimiento esperado, podríamos suponer que el servidor que estamos usando tiene una UF con una latencia de 4 ciclos de reloj.

Para comprobarlo, y dado que lo habitual es que dicha UF esté segmentada, vamos a utilizar dos variables distintas de reducción, por lo que esperamos obtener el doble de rendimiento. Para ver si estamos en lo cierto, vamos a usar el *benchmark sumaflops2*. Compilamos con `$ icc -Wall -O2 -xCORE-AVX2 -o sflops2 sumaflops2.c` y ejecutamos, obteniendo un rendimiento de 1,6 GFLOP/s. Si hacemos esto mismo con los *benchmarks* de 3 y 4 variables, vemos que vamos progresando a 2,4 y 3,2 GFLOP/s respectivamente.

La sorpresa viene ahora porque si seguimos añadiendo variables, de 5 a 8, seguimos obteniendo un incremento lineal, hasta alcanzar los 6,4 GFLOP/s. ¿A qué es debida esta mejora? Pues en este caso, y como ya habíamos comentado al inicio de la práctica, a que cada núcleo del procesador tiene 2 UF que pueden hacer operaciones en coma flotante, por lo que en el caso de 8 variables estamos usando las 2 UF a su plena capacidad. Para verificar esto, podemos compilar y ejecutar los *benchmarks* de *sumaflops9* a *sumaflops16*, donde comprobamos que ya no obtenemos ninguna mejora.

7. Programación: Usar bibliotecas HPC

Para finalizar esta primera práctica, vamos a mostrar la utilidad de usar bibliotecas de cálculo científico especialmente optimizadas para HPC como un medio muy adecuado para obtener el máximo rendimiento de un procesador que esté ejecutando un algoritmo que pueda hacer uso de dicha biblioteca.

En este apartado, vamos a seguir usando el compilador icc y su biblioteca de funciones matemáticas MKL para HPC.

Para esta última sección, vamos a utilizar una aplicación muy común dentro del campo del álgebra lineal como es la multiplicación de matrices de 2 dimensiones (esta sección tiene sobre todo sentido cuando se usan matrices de tamaños grandes, a partir de un millón de elementos por matriz con números en simple o doble precisión).

A modo de recordatorio, dos matrices A y B se pueden multiplicar conjuntamente dando como resultado una tercera matriz C, donde un elemento de C se calcula ejecutando una multiplicación de los elementos de una fila de A por los elementos correspondientes de una columna de B y luego sumando todos los productos. Esto implica que el número de columnas en A debe ser igual al número de filas de B. El tamaño de la matriz de resultados C tendrá el mismo número de filas que A y el mismo número de columnas que B. Por ejemplo, si A es una matriz `m × p` y B es una matriz `p × n`, entonces C será una matriz `m × n`.

El algoritmo estándar de triple bucle anidado para la multiplicación de matrices es bien conocido, con una complejidad $O(n^3)$ (para matrices cuadradas `n × n`). El código para este algoritmo se muestra en el Listado 1. 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. En el cuerpo del código del bucle más interior del Listado 1, se accede a los elementos de C y A por filas (*row-major order*), mientras que se accede a los elementos de B por columnas (*column-major order*).

Listado 1: Algoritmo estándar de multiplicación de matrices 2D

```
void matrix_multiply (int m, int n, int p, float **A, float **B, float **C)
{
        int i, j, k;

        for (i = 0; i < m; i++)
        for (j = 0; j < n; j++) {
            float tmp = 0.0;
            for (k = 0; k < p; k++)
            tmp += A[i][k] * B[k][j];
        C[i][j] = tmp;
        }
}
```

Para esta sección, vamos a utilizar las siguientes aplicaciones que se encuentran en el subdirectorio *HPC-library*::

MMultSeq2D.c Versión con matrices con memoria dinámica (puntero de puntero)

MMult2Seq2D.c Versión con matrices con memoria dinámica lineal, con 2 recorridos: ijk e ikj

MMultmkl2D.c Versión utilizando la función `cblas_sgemm` de la librería MKL de Intel para multiplicación de matrices (en SP).

Para empezar, vamos a compilar las dos versiones secuenciales que nos dan. Para ello, usamos los siguientes comandos: `icpc -Wall -O2 -xCORE-AVX2 -o MMultSeq2D MMultSeq2D.cc` y `icc -Wall -O2 -xCORE-AVX2 -o MMult2Seq2D MMult2Seq2D.c`. Para ejecutar los programas con tamaños de matrices de 2048 elementos, en el primer caso debemos dar los nombres de los ficheros donde tenemos las matrices de entrada (`./mmulseq mat_a_2048 mat_b_2048`[11]), mientras que en el segundo caso tan sólo es necesario decir el tamaño de las matrices y la aplicación las crea con valores aleatorios (`./mmul2seq 2048`). En el primer caso obtenemos un valor de 9,2 GFLOP/s, mientras que la segunda versión nos ofrece un rendimiento ligeramente superior, de 9,4 GFLOP/s.

El primer comentario es que el rendimiento obtenido en una aplicación de algebra lineal es bastante pobre, pues es una sexta parte del rendimiento que nos ofrece uno de los núcleos del servidor, y alrededor de 25 veces menos de lo máximo que podríamos obtener con dicho procesador.

[11] En la práctica tienes el fichero *gen.c* para generar otras matrices de entrada de diferentes tamaños.

Para mejorar este rendimiento, deberíamos intentar mejorarlo por medio de la vectorización y paralelización (técnicas que se verán en las práctica 2, y se ampliarán en la práctica 3). En esta primera práctica, queremos mostrar una forma sencilla de mejorar el rendimiento de una aplicación por medio de usar una biblioteca de HPC (si dicha aplicación lo permite).

Las bibliotecas numéricas suelen ser mucho más rápidas que cualquier código que se pueda escribir. Aceleran tareas tan simples como escalar un *array*, o tan complejas como encontrar mínimos en funciones multidimensionales. Hay bibliotecas para aplicaciones completas (como AMBAR, NAMD o GROMACS que son del ámbito bioinformático), para acelerar funciones matemáticas (como Intel MKL o la GNU GSL), para mejorar la entrada/salida (como HDF5, Parallel-NetCDF o Globus), o para evaluar algunas características de la aplicación (como Tau, PAPI o DDT). Entre las bibliotecas numéricas optimizadas, las más utilizadas son aquellas que soportan el álgebra lineal, especialmente las implementaciones de BLAS así como las transformadas discretas de Fourier (DFTs).

En esta práctica nos vamos a centrar en una biblioteca optimizada en particular, la Intel MKL. ¿Qué es MKL? La biblioteca de funciones matemáticas de Intel que ofrece interfaces para los lenguajes de programación Fortran y C, e incluye funciones para los niveles BLAS 1-3, LAPACK, FFT, la Biblioteca de Matemáticas Vectoriales (VML) y otros. MKL está optimizado por Intel para todas las arquitecturas actuales de Intel, e incorpora paralelismo de memoria compartida a través de OpenMP. A pesar de que nos centramos en C y Fortran, otros lenguajes de programación a menudo también tienen bibliotecas optimizadas[12].

Otro beneficio adicional obtenido de usar funciones optimizadas en bibliotecas es que muy probablemente dicha función habrá sido optimizada para una variedad de plataformas, por lo que tendremos un código más portable, con muchas menos líneas de código, y que tiene en cuenta la localidad de los datos.

En nuestro caso, vamos a usar la función *cblas_sgemm* de la biblioteca MKL que nos hace la multiplicación de matrices para números reales en simple precisión. Para poderla usar, en la cabecera del programa hay que incluir dicha biblioteca *#include <mkl.h>*, y asimismo hay que incluirla con un *switch* en el proceso de compilación. El programa *MMultmkl2D.c* ya está preparado para usar dicha biblioteca. Por lo tanto si compilamos con `icpc -Wall -O2 -qopenmp -xCORE-AVX2 -mkl -o MMultmkl2D MMultmkl2D.c` y corremos el fichero ejecutable, obtenemos un rendimiento de 206 GFLOP/s para matrices cuadradas de 4096 elementos.

¿Qué conclusiones podemos extraer? Por una parte, lo fácil que ha sido multiplicar por algo más de 20 el rendimiento de una aplicación en el caso de usar una función que se encuentre en la bilioteca MKL. Por otra parte, podemos ver que a pesar de ello, no hemos obtenido el rendimiento pico del procesador, quedándonos en torno al 60 %. Esto es lógico, pues dicho rendimiento pico es una cota superior difícilmente alcanzable fuera de aplicaciones *de juguete* especialmente programadas para ello.

Créditos

- Esta práctica toma ideas de las secciones 1.1 y 5.1 del libro «*Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors*». 2nd Edition. Colfax International, 2015.

- Para la sección 4 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`

- Asimismo, en la página web de Agner Fog (`http://www.agner.org/optimize/?e=0#0`), Catedrático de la *Technical University of Denmark, Department of Information Technology, Copenhague* (Dinamarca), hay muchos documentos acerca de la microarquitectura de los procesadores x86.

- Finalmente, agradecer al alumno interno de la asignatura *Pablo Antonio Martinez Sanchez <pabloantonio.martinezs@um.es>* la revisión que ha realizado de la práctica y la corrección de algunos errores. Además, Pablo ha desarrollado un micro-benchmark que puedes encontrar aquí: `https:`

[12] NumPy es un ejemplo preeminente de esto para el lenguaje Python.

//github.com/Dr-Noob/FLOPS[13].

[13]Para entenderlo correctamente, te puede venir bien consultar esta dirección: https://software.intel.com/sites/landingpage/IntrinsicsGuide/