Universidad de Murcia
Facultad de Informática

Grado en Ingeniería Informática
3er Curso

Arquitectura y Organización de Computadores

Práctica 3 - Optimización del rendimiento de un procesador CMP

Curso académico 2021-22

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

1. Proceso de optimización de código

Antes de entrar propiamente en las técnicas avanzadas de vectorización y paralelización, vamos a recordar algunas de las transformaciones que se pueden realizar al código para optimizarlo. Muchas de las optimizaciones que se mencionan a continuación las realiza automáticamente cualquier buen compilador, aunque en ocasiones conviene hacerlas a mano.

Es importante escribir código "*compilador-friendly*", es decir, código que pueda ser procesado fácilmente por el compilador y optimizado eficientemente. El código *compilador-friendly* evita las partes del lenguaje (C++ en nuestro caso) que crean restricciones innecesarias o ambigüedades (comportamientos no totalmente definidos por el estándar del lenguaje). Por ejemplo, el uso de aritmética de punteros o de punteros no restringidos[1] dificulta enormemente la tarea de optimización automática debido a los posibles *alias* entre punteros que el compilador no siempre puede demostrar que no se producen (aunque a simple vista cualquier programador lo vea claro). En general, es buena idea minimizar el uso de punteros, aunque puedan ser convenientes, ya que los compiladores están a menudo obligados a hacer suposiciones conservadoras sobre los datos referenciados, lo que dificulta la aplicación de muchas optimizaciones.

Normalmente, el código *compilador-friendly* también es un código legible y claro para los seres humanos. En particular, hay que evitar la tentación de hacer manualmente optimizaciones sin medir rigurosamente el efecto en el rendimiento de cada cambio, ya que esto no solo puede comprometer la legibilidad del código para cualquier persona, sino que también puede dificultar la labor del compilador para aplicar otras optimizaciones. En ocasiones, es interesante ver el código ensamblador generado por el compilador para nuestro programa para comprobar cómo se han traducido aquellas partes que consumen más tiempo de ejecución (los "puntos calientes" del código).

Algunas optimizaciones importantes son las siguientes:

1. Mover invariantes fuera de bucles. Es importante evitar hacer cálculos repetidamente dentro de los bucles cuando el resultado sea igual para todas las iteraciones, ya que en ese caso es mucho mejor hacer el cálculo una sola vez antes del bucle. La idea es *mover* expresiones invariantes fuera de los bucles para evitar cálculos innecesarios. El compilador generalmente reconocerá estas situaciones y hará lo correcto. Sin embargo, en algunas ocasiones no podrá hacerlo (por no poder ver que la expresión es invariante). Un ejemplo típico es cuando dentro de un bucle se usan variables que han sido pasadas a una función por medio de punteros, por lo que el compilador no puede saber si dicho valor es invariante en el bucle y se ve obligado a omitir esta optimización (ya que puede existir otro puntero que apunte a la misma dirección y lo modifique).

2. Utilizar el *inlining*. La práctica de reemplazar la llamada a una función por el código de dicha función (el cuerpo de la función) se denomina *inlining*. Esta optimización no solo permite ahorrarnos la sobrecarga de las llamadas a las funciones pequeñas, sino que es clave para permitirle al compilador realizar otras transformaciones después, como la eliminación de código innecesario o la propagación de constantes.

3. Evitar las conversiones de tipos implícitas o explícitas. Las conversiones implican mover datos entre diferentes unidades de ejecución. Esta recomendación es especialmente importante dentro de un bucle.

4. Hacer aritmética inteligente, especialmente si el procesador cuenta con una operación combinada de multiplicar-sumar (FMA) en su conjunto de instrucciones (lo cual es cierto para la mayoría de los procesadores actuales que soportan vectorización). Del mismo modo, la multiplicación debe ser una operación favorecida sobre la división, ya que es una operación más barata.

5. Explotar la localidad de datos. Esta es una de las técnicas más importantes para mejorar el rendimiento por núcleo. Cualquier procesador tiene una jerarquía de memoria con varios niveles:

[1] Un puntero restringido garantiza al compilador que es el único puntero que apunta a una dirección de memoria concreta. En C se puede usar la palabra clave `restrict` para indicarlo, soportada también por casi todos los compiladores de C++.

caché L1 (el más rápido), luego L2, (luego L3), luego memoria principal. Durante un cálculo, el mejor resultado sería que nuestros datos se encuentren en la caché L1 cada vez que se necesiten. Para favorecer la localidad de datos es importante que el código acceda a direcciones de memoria contiguas (*stride-one*, paso uno), y que haya una gran re-utilización de los datos (que los datos permanezcan en caché el mayor tiempo posible, de modo que los niveles más profundos de la jerarquía de memoria se acceden con poca frecuencia). Esto es particularmente importante en los arrays con 2 o más dimensiones, pues hay que tener en cuenta cómo se almacenan los elementos del array en memoria para recorrerlos en un orden que facilite dicha localidad.

La localidad de datos se vuelve aún más importante para coprocesadores y aceleradores (por ejemplo, GPUs) que para las CPUs, simplemente porque una búsqueda desde la memoria principal a un coprocesador o caché de GPU suele ser más lenta de lo que sería para una CPU.

6. Favorecer la vectorización. Cuando el rendimiento sea muy importante, se recomienda definir estructuras de arrays (SoA), no arrays de estructuras (AoS). El compilador tiene más dificultades para vectorizar un bucle en el que una secuencia de elementos debe extraerse de estructuras separadas antes de la computación si los elementos con los que se debe operar estén separados en memoria por un *stride* (paso) mayor que uno. Por otra parte, para obtener el mejor rendimiento y facilitar la vectorización, el acceso a los datos en memoria es importante que esté alineado al tamaño del vector.

7. Usar el *padding*. La técnica del *padding* (relleno de datos) es una técnica muy efectiva que favorece mucho tanto la vectorización del código de un bucle como su paralelización (gracias a evitar la compartición falsa de datos entre hilos).

2. Descripción y objetivos

Esta práctica trata de afianzar los conceptos de vectorización y paralelización vistos en la práctica anterior, mostrando algunas técnicas de optimización del rendimiento de un procesador CMP para mejorar la eficiencia de las aplicaciones paralelas.

Para explotar al máximo la capacidad multinúcleo de nuestro procesador, vamos a ver en esta práctica tercera 4 áreas importantes de optimización que tenemos que aplicar a cualquier aplicación. La primera área es la optimización escalar (también denominada a veces *reducción de fuerza*), la segunda se refiere a la correcta paralelización de la aplicación, la tercera a explotar eficientemente la capacidad vectorial que tienen los procesadores actuales, y la cuarta y última a la optimización en el acceso a memoria. Falta otra área de optimización adicional relacionada con la optimización de la comunicación de datos entre procesadores que queda fuera del alcance de esta asignatura.

Para ayudarnos en el proceso de optimización, vamos a usar la herramienta `perf`[2] que nos mostrará cómo las diversas optimizaciones realizadas mejoran los diversos contadores internos del procesador.

Más concretamente, los objetivos de esta práctica son:

- Comprender cómo mejorar el código aplicando la "reducción de fuerza" de operaciones complejas
- Explotar adecuadamente la paralelización de una aplicación
- Mejorar la vectorización cambiando el tipo de estructuras de datos que se utilizan en el programa
- Mejorar la vectorización por medio de alinear los datos utilizados
- Mejorar el rendimiento de la aplicación optimizando su acceso a memoria por medio de re-utilizar el acceso a los datos (técnica del *tiling* o enlosado)
- Usar la herramienta `perf` para obtener el IPC, los fallos de caché, etc, que nos permitan conocer mejor cómo han influido las optimizaciones realizadas

[2] `https://perf.wiki.kernel.org/index.php/Main_Page`

3. Entorno de trabajo

La realización de esta práctica considera que utilizamos los compiladores de C++ descritos en la práctica 1 (el GCC versión 11.0.1, o el ICC versión 2021.3.0, o el Clang 12.0.0), bien usando la imagen Docker facilitada en la asignatura, o bien teniéndolos instalados en el sistema que el alumno utilice.

4. Caso de estudio: Simulación de N cuerpos

Para explicar los conceptos de esta tercera práctica, vamos a tomar como ejemplo de aplicación un código que simula la interacción gravitacional o electrostática de N-cuerpos, que es una simulación de muchas partículas que interactúan unas con otras. Las simulaciones de N-cuerpos se utilizan en astrofísica para modelar la evolución de las galaxias, la colisión de galaxias, la distribución de la materia oscura en el universo, y los sistemas planetarios. También se utilizan en las simulaciones de estructuras moleculares.

En el código que vamos a usar, cuya complejidad es $O(n^2)$, se realiza un seguimiento de la posición y la velocidad de cada partícula mantenidas en una estructura de datos denominada *Particle* (se considera que la masa o la carga de cada partícula tiene valor 1). La simulación es discretizada a lo largo del tiempo en una cantidad prefijada de pasos de tiempo (*timesteps*). En cada paso de tiempo, el código lo primero que hace es calcular la fuerza sobre cada partícula por medio de un algoritmo básico todos-a-todos (con complejidad $O(n^2)$). A continuación, la velocidad de cada partícula se modifica y posteriormente las posiciones de las partículas son actualizadas (en ambos casos se utiliza el método de Euler). Las simulaciones reales astro-físicas de N-cuerpos, dirigidas a sistemas con miles de millones de partículas, utilizan simplificaciones para reducir la complejidad a $O(n \cdot logn)$. Sin embargo, el sencillo modelo utilizado en la práctica es la base sobre la que se construyen los modelos más complejos.

Para empezar, vamos a familiarizarnos con las características básicas de la aplicación y los primeros pasos a realizar en el proceso de optimización. Para ello, dentro de la carpeta *materiales* vete al subdirectorio *nbody* y encontrarás un fichero denominado *nbody.cpp*. Si estudias brevemente el código, observarás que por defecto tenemos 16.384 partículas, y que el paso de tiempo es de 0,01. Además, verás que cada partícula está almacenada en una estructura donde tienes su posición y velocidad:

```
struct ParticleType {
        float x, y, z;
        float vx, vy, vz;
};
```

En la función *main()* del programa se crea un array de `ParticleType` por medio de la instrucción

```
ParticleType* particle = new ParticleType[nParticles];
```

La parte *importante* del programa (la que hay que optimizar, pues es la que consume la mayoría del tiempo de ejecución) se encuentra en la función *MoveParticles()* donde se usa la ley de Newton para calcular las nuevas posiciones de cada partícula. Para cada *timestep* (paso de tiempo), el tiempo que ha tardado en realizar el cálculo de la nueva situación de las partículas está medido utilizando la función que nos proporciona OpenMP (*omp_get_wtime()*):

```
const double tStart = omp_get_wtime(); // Start timing
MoveParticles(nParticles, particle, dt);
const double tEnd = omp_get_wtime(); // End timing
```

A continuación compila el programa para obtener el fichero ejecutable por medio de la siguiente orden: `$ make all`. Como veras, se han generado 3 ficheros ejecutables (uno por cada compilador posible, el `icc`, el `gcc` y el `clang`). Si los ejecutas, obtendrás el rendimiento de referencia de la aplicación para cada uno de los compiladores usados[3].

Vamos a usar por primera vez la herramienta `perf`. Para una mejor comprensión de la misma, se recomienda leer el apartado de `perf` de la página web de Brendan Gregg[4]. En este caso, vamos a estar interesados en medir los siguientes eventos:

[3] Como verás, el rendimiento varía mucho de un compilador a otro (¡hasta casi 5 veces!), por lo que la elección del compilador es un parámetro importante cuando queremos optimizar el rendimiento de un código determinado.

[4] `https://www.brendangregg.com/perf.html`

- *cache-references*: Este evento representa el número de accesos a la caché de último nivel, pero esto puede variar según la CPU. Esto puede incluir los accesos por *prefetching* y los mensajes de coherencia; nuevamente, esto depende del diseño de la CPU.
- *cache-misses*: Este evento representa el número de accesos a la memoria que no pudieron ser atendidos por ninguna de las cachés. En conjunción con el evento anterior, se puede usar para obtener el *cache miss rate* de la CPU.
- *cycles*: Ciclos de reloj que ha tardado la ejecución del programa.
- *instructions*: Se refiere al número de instrucciones que se han *retirado* (han hecho *Commit*) del Buffer de Reordenación (ROB).
- *L1-dcache-loads*, *L1-dcache-load-misses*, *LLC-loads*, *LLC-load-misses*: Número de accesos por instrucciones de cargas, y fallos de cache a nivel de L1 y LLC.

Para obtener esta información, ejecuta, para cada uno de los 3 ejecutables, el comando `$ perf stat -d -e cache-references,cache-misses,cycles,instructions` *`Nombre_ejecutable`*. Veras que los compiladores `gcc` y `clang` han generado un código con muchas más instrucciones que el compilador `icc`, por lo que aunque su IPC es mayor, el tiempo de ejecución (que es la métrica que realmente importa) es peor en estos dos compiladores. Observa el número de accesos a cache L1 y LLC que luego nos fijaremos en eso conforme avance la práctica.

Por último, si pensamos un poco podemos darnos cuenta de que el rendimiento obtenido por cualquiera de los ejecutables es bastante pobre en comparación con el rendimiento pico que podíamos obtener en dicho ordenador.

4.1. Optimización del programa

En esta sección, vamos a realizar paso a paso una optimización del código de `nbody.cpp`, intentando mejorar su rndimiento. Ten en cuenta que cuando hablamos de mejora hay que especificar la métrica que queremos utilizar. En este caso la métrica va a ser el tiempo de ejecución (es decir, vamos a realizar mejoras que nos permitan reducir el tiempo de ejecución de la aplicación dada), pero podríamos tener otros objetivos[5].

Siempre que vayamos a optimizar un código, lo primero que hay que hacer es comprobar que el resultado sigue siendo válido (habitualmente el mismo), pues si no tenemos esa precaución podemos cambiar la semántica del código y que nos ofrezca resultados erróneos. Por ello, añadimos una nueva función denominada *MoveParticles_Orig()* que utiliza la función original sin optimizaciones para darnos el valor correcto. Cada vez que hagamos una optimización en la aplicación compararemos lo que obtenemos con esta salida y mostraremos si es o no correcto el proceso de optimización realizado.

El primer paso para empezar a optimizar es generar un fichero que nos de un informe de todo lo que ha hecho el compilador. Para ello, cámbiate al subdirectorio *solutions/01-messages* y vuelve a compilar con `$ make all`. En este nuevo *Makefile* hemos añadido unos *switches*[6] para que nos genere un informe del proceso de compilación. Si miras los ficheros de informe resultantes, comprobaras que hay mucha información. Nos vamos a centrar en el fichero `nbody-icc.optrpt` de `icc` que nos parece el más sencillo de entender. Aún así, iremos poco a poco explicando su contenido y viendo qué posibilidades de mejora podemos aplicar, así como si dichas mejoras ayudan en el rendimiento obtenido en la aplicación (medido en los GFLOP/s obtenidos en la ejecución).

4.1.1. Conversiones de tipo innecesarias

Vamos a empezar fijándonos en los mensajes *type converts: 3*. Estos mensajes se producen en la zona del código del cálculo de la fuerza que actúa sobre cada partícula (el bucle más interno de la función

[5] Como por ejemplo, que el código del fichero ejecutable sea lo más reducido posible o que la ejecución consuma la menor energía, factores ambos importantes en aplicaciones a ser usadas con dispositivos móviles.

[6] (Para `icc` el `-qopt-report=5 -qopt-report-file=nbody-icc.optrpt`, para `gcc` el `-fopt-info-all=nbody-gcc.oprpt`, y para `clang` el `-foptimization-record-file=nbody-clang.optrpt`)

MoveParticles() que es el bucle *j*) debido a conversiones implícitas de tipos. En particular, el problema lo causa la llamada a `pow`, debido a que el tipo de esta función es `double pow(double,double)` mientras que el resultado se está asignando a `drPower32` (cuyo tipo es `float`) y como primer argumento está recibiendo `drSquared` (cuyo tipo también es `float`). Para eliminar estas conversiones de tipos, podemos utilizar la función `powf` en lugar de `pow`, cuyo tipo es `float powf(float,float)`.

Si vas al subdirectorio *solutions/02-conversions* encontraras una versión modificada *nbody.cpp*. Compila de nuevo y genera el fichero de informe `nbody-icc.optrpt` y comprobarás que han desaparecido los mensajes acerca de la conversión de tipo. Si ahora ejecutas la aplicación, se debe observar una mejora de rendimiento (en la máquina `compiladorintel.inf.um.es` se obtiene sobre un 20 % de mejora sobre la versión anterior para el ejecutable obtenido con ICC, un 30 % en el caso de GCC y se dobla el rendimiento en el caso de Clang).

4.1.2. Reducción de fuerza

Una vez realizado esto, vamos a fijarnos mejor en cómo se están haciendo los cálculos en el bucle más interno y vamos a tratar de reescribirlos para obtener el mismo resultado de manera más eficiente. Vamos a reescribir las expresiones que calculan cuánto hay que incrementar cada una de las componentes del vector de fuerza (F_α donde α es x, y o z). Pasaremos de:

$$\Delta F_\alpha = \frac{d_\alpha}{(d_x^2 + d_y^2 + d_z^2 + softening)^{\frac{3}{2}}}$$

A la expresión equivalente:

$$\Delta F_\alpha = d_\alpha \times \left(\frac{1}{\sqrt{d_x^2 + d_y^2 + d_z^2 + softening}} \right)^3$$

Esto tiene las siguientes ventajas:

- No utiliza divisiones, salvo para calcular el recíproco de la raíz cuadrada. Las división presente en la fórmula original se ha transformado en una multiplicación por el recíproco del divisor.
- No requiere usar la función `powf` para elevar a 3/2. En su lugar, este cálculo se realiza mediante una raíz cuadrada (para elevar a 1/2) y dos multiplicaciones (para elevar al cubo).
- Permite utilizar la instrucción `vrsqrtps`, la cual permite calcular de forma eficiente el recíproco de la raíz cuadrada elemento a elemento de un vector. Obsérvese que usando esta instrucción en lugar de `rsqrtps` (que calcula raíces cuadradas) nos ahorramos también la única división que aparece en la fórmula modificada.

Si comprobamos el código ensamblador generado para la versión anterior, podemos ver que el `icc` ya estaba usando la instrucción `vrsqrtps` y multiplicaciones para evitar la llamada a `powf` y poder vectorizar el bucle (el compilador realiza esta transformación después de hacer el *inlining* de powf). Sin embargo, la transformación que realizaba automáticamente no obtenía tan buen rendimiento como la que hacemos manualmente.

Si ahora te mueves al subdirectorio *solutions/03-strength-reduction* encontraras el nuevo fichero *nbody.cpp*. Compila de nuevo y genera el fichero de informe `nbody-icc.optrpt` y comprobarás que han desaparecido los mensajes acerca de la conversión de tipo. Si ahora ejecutas la aplicación, comprobarás que el rendimiento ha subido un poco más (en `compiladorintel.inf.um.es`, se obtiene sobre un 24 % de mejora sobre la versión anterior para el ejecutable obtenido con ICC, un 4 % en el caso de GCC y no se mejora el rendimiento en el caso de Clang).

Al igual que antes, vamos a volver a usar la herramienta `perf`. Para ello, ejecuta de nuevo el comando `$ perf stat -d -e cache-references,cache-misses,cycles,instructions` *`Nombre_ejecutable`* para cada uno de los 3 ejecutables. Veras que en todos los casos el número de instrucciones generado por cada compilador es similar a la versión inicial (etiquetada con 00), pero que el número de ciclos para ejecutar dichas instrucciones ha bajado por lo que ha subido el IPC. El resto de parámetros se mantiene mas o menos estable.

4.1.3. Paralelización

Por último, vamos a paralelizar la función principal de la aplicación `MoveParticles` utilizando OpenMP. Recuerda que hay dos bucles que podrían ser paralelizados, pero sólo vamos a paralelizar el bucle más exterior[7]. Además, vamos a modificar la instrucción de impresión para que imprima el número real de hilos utilizado.

En el subdirectorio *solutions/04-parallel* encontraras el nuevo fichero *nbody.cpp*. Compila de nuevo y ejecuta la aplicación, y comprobarás que el rendimiento ha subido enormemente, lo que no es ninguna sorpresa pues hemos pasado de utilizar 1 núcleo a usar todos los núcleos que tiene nuestro procesador (por lo hemos obtenido una mejora cercana al número de núcleos físicos). Como puedes también comprobar, el resultado sigue siendo correcto (y así nos lo dice la aplicación), pues simplemente hemos aplicado una estrategia de paralelismo de datos, en donde hemos dividido el número de partículas que tiene la aplicación por el número de hilos disponibles.

Vamos a ver ahora que resultados obtenemos con la herramienta `perf` y compararlos con los que obteníamos para la versión anterior (los valores siguientes se refieren a `compiladorintel.inf.um.es`). Vemos que el número de instrucciones aumenta muy ligeramente (4 %), y que el número de ciclos aumenta algo más (18 %). Por tanto, el IPC del procesador ha bajado ligeramente (12 % menos). Sin embargo, el tiempo de ejecución de la aplicación se ha reducido (casi a la mitad en `compiladorintel.inf.um.es`). Esto es porque para casi todas las estadísticas mostradas por `perf`, los valores se refieren a la suma de los de todos los procesadores[8]. Es decir: aunque se hace un poco más de trabajo debido al sobrecosto de la paralelización (más instrucciones), el trabajo se reparte entre varios procesadores, lo cual les permite acabarlo antes (menor tiempo de ejecución), incluso aunque cada procesador trabaje más despacio (menos IPC, además de menor frecuencia muy probablemente). El número de instrucciones ejecutadas ha crecido ligeramente debido a las instrucciones que introduce `OpenMP` para manejar y sincronizar los hilos.

4.1.4. Arrays de estructuras vs estructuras de arrays y alineamiento

En esta sección vamos a ver cómo podemos optimizar la vectorización por parte del compilador, gracias a usar estructuras de datos que nos permitan mejorar el proceso de vectorización de una aplicación por parte del compilador, y a aplicar el alineamiento a las estructuras de datos que usemos.

En la implementación actual, los datos de cada partícula se almacenan en una estructura de datos tipo Array de Estructuras (AoS), es decir, un array de una estructura de `ParticleTypes`. Aunque esto es muy bueno para la legibilidad y la abstracción del programa, es sub-óptimo para el rendimiento, ya que las coordenadas de las partículas consecutivas no son adyacentes. Así, cuando se accede en el bucle a las posiciones y a las velocidades y se intenta vectorizar, los datos tienen un *stride* (paso) distinto de la unidad, lo que dificulta el rendimiento.

Para entender mejor lo que vamos a hacer, vuelve a mirar el fichero de informe de las optimizaciones realizadas por el compilador que encontrarás en el subdirectorio *solutions/01-messages*. Fíjate especialmente en los mensajes que ha generado el compilador para el bucle más interno de la función *MoveParticles()*.

```
LOOP BEGIN at nbody.cpp(21,5) inlined into nbody.cpp(87,5)
remark #15415: vectorization support: non-unit strided load was generated for
  the variable <particle->x[j]>, stride is 6    [ nbody.cpp(27,24) ]
(...)
remark #15300: LOOP WAS VECTORIZED
remark #15452: unmasked strided loads: 3
(...)
LOOP END
        }
```

[7] Este es el procedimiento habitual, pues el bucle interior es el candidato ideal para ser vectorizado.

[8] De igual forma, podemos ver que el tiempo total consumido por los procesadores (reportado por `perf` como «`seconds user`» más «`seconds sys`») es mucho mayor que el tiempo que realmente ha tardado el programa (reportado como «seconds time elapsed»).

Como ves, aparece el mensaje de que el bucle ha sido vectorizado, pero con un paso diferente a uno (fíjate que aparece la palabra *strided*, junto a *masked* o *gather*).

Para solucionar esto, en este apartado de la práctica vamos a cambiar la estructura de datos AoS por una SoA (Estructura de Arrays), mediante la sustitución de `ParticleType` por `ParticleSet`. `ParticleSet` debería tener 6 arrays de tamaño `n`, uno por cada dimensión de las coordenadas de la posición (x, y, z) y de la velocidad (vx, vy, vz). El elemento i-ésimo de cada array es la coordenada o velocidad de la partícula i-ésima. Cambiar los datos de AoS a SoA implica que hay que modificar también la inicialización de los mismos en `main()`, y modificar el acceso a los arrays dentro de la función `MoveParticles()`.

Adicionalmente, para obtener el mejor rendimiento de la vectorización hay que procurar que los accesos a memoria estén alineados. El valor de la alineación depende del tamaño de la unidad vectorial: si ésta es de 256 bits sería a múltiplos de 32 bytes, si fuera de 512 bits a múltiplos de 64 bytes, etc. Para ello, es necesario reservar la memoria utilizando una función que nos permita especificar el alineamiento deseado, como `std::aligned_alloc` (en C++, cabecera `<memory>`) o `posix_memalign` (en C, cabecera `<stdlib.h>`). En ambos casos, posteriormente se debe liberar la memoria con `free()`. Además es necesario informar al compilador en el momento que necesita acceder a través de punteros a esa memoria que los punteros utilizados apuntan a direcciones alineadas, lo cual se puede hacer mediante la función `std::assume_aligned<a>(p)` en C++, que devuelve un puntero igual a p asumiendo que la dirección es múltiplo de a. En el caso del compilador ICC, podemos usar el pragma `#pragma vector aligned` antes del bucle que esperamos vectorizar para decirle al compilador que asuma que los datos están debidamente alineados.

En el subdirectorio *solutions/05-vectorization* encontraras el nuevo fichero *nbody.cpp*. Compila el programa permitiendo que genere informes, y vuelve a mirar el fichero de informe de las optimizaciones realizadas por el compilador, para comprobar que ahora tenemos un acceso a los datos con un *stride* de 1 y que el acceso a los datos ahora está alineado. Si ahora ejecutamos el programa, ¡hemos obtenido otra importante mejora con respecto al apartado anterior! Como verás, el beneficio está relacionado con el tamaño de las unidades vectoriales, el número de unidades vectoriales que tiene el procesador, y el compilador empleado. En este caso, puedes ver que los 3 compiladores ofrecen rendimientos cercanos, y en este caso se nota menos el impacto de usar un compilador u otro.

También ahora nos fijamos en los resultados que obtenemos con la herramienta `perf`. Gracias a la vectorización, ha bajado el número total de instrucciones, pero sobre todo se ha reducido considerablemente el número de ciclos, lo que se traduce en una reducción directa del tiempo total de ejecución.

Por lo tanto, a menudo es beneficioso cambiar la estructura de datos que contiene a las partículas, y aplicar en su lugar una Estructura de Arrays (SoA), donde una única estructura tiene los arrays de coordenadas. Además, de cara a lograr una adecuada vectorización, necesitamos alinear los datos a múltiplos de 32 o 64 bytes.

4.1.5. Optimizando el acceso a memoria

En esta última sección de la práctica vamos a mejorar el acceso a memoria para intentar aumentar aún más el rendimiento de nuestra aplicación. Para que tenga sentido aplicar este tipo de optimización, debemos tener un problema (un algoritmo) que tenga una ejecución del tipo "limitado por memoria" (*memory-bound*), ya que si el problema es del tipo "limitado por cómputo" (*compute-bound*) este tipo de optimización no tiene sentido aplicarlo.

Hay tres técnicas que se suelen utilizar para optimizar el acceso a la memoria: a) El acceso a posiciones de memoria con paso unidad (*Unit-stride access*), b) el alineamiento de los datos a posiciones de memoria múltiplos de 32 o 64 bytes, que incluye muchas veces usar la técnica del *padding* (o *restregado*) para mantener dicho alineamiento entre diversos accesos, y c) el reuso de datos en caché, por medio de usar la técnica conocida como *tiling* (usar azulejos o enlosado). Las 2 primeras ya les hemos visto en el apartado de vectorización, por lo que nos vamos a centrar en la última de ellas.

Loop tiling es una técnica para la optimización del tráfico de memoria en algoritmos que implican bucles anidados, matrices multi-dimensionales, y que tienen patrones de acceso a memoria regulares. Con esta técnica pretendemos específicamente mejorar la utilización de la memoria caché. Esta técnica también se conoce como *strip-mine and permut*, debido a que para recorrer un bucle usando azulejos,

el programador utiliza la técnica denominada *strip-mine* (hacer "tiras") en el bucle interno o externo, y luego permuta algunos de los bucles en el código resultante.

A continuación vamos a aplicar la técnica del *tiling* sobre el bucle más interno *j*. A esta variante del *tiling* se le denomina *cache blocking*. La forma de hacerlo es descomponer el bucle (*strip-mine*) en 2 bucles usando una nueva constante llamada `TILE` y luego hacemos una permuta de bucles, quedando de la siguiente forma:

```
#pragma omp parallel for collapse(2)
for (int jj = 0; jj < nParticles; jj+=TILE)
  for (int i = 0; i < nParticles; i++) {
    for (int j = jj; j < jj+TILE; j++) {
}
}
```

De cara a tener una paralelización efectiva, es habitual añadir el modificador `collapse(2)` en la paralelización del bucle para que nos una (colapse) los dos primeros bloques[9]. De esta forma, tenemos siempre bloqueado una parte de tamaño `TILE` cuando estamos recorriendo el bucle en *i*, que esperamos pueda alojarse en la cache del procesador (habitualmente la L2). El valor de TILE es un parámetro a ajustar, manteniéndolo siempre en un múltiplo de 64.

Esta solución tiene un problema, y es que tiene condiciones de carrera cuando se actualiza el valor de la velocidad de cada partícula, por lo que tenemos que garantizar que dicha operación la hacemos de forma atómica por medio de escribir el siguiente código:

```
#pragma omp atomic
particle.vx[i] += dt*Fx;
#pragma omp atomic
particle.vy[i] += dt*Fy;
#pragma omp atomic
particle.vz[i] += dt*Fz;
```

En el subdirectorio *solutions/06-tiling* encontraras el nuevo fichero *nbody.cpp* con un tamaño de `TILE` de 4096. Puedes probar con otros valores de TILE. Como verás, la mejora es muy sensible a este valor. En el caso de la máquina `compiladorintel.inf.um.es` dicha mejora no es importante (incluso empeora levemente el rendimiento), debido a que tenemos un tamaño de problema muy pequeño (tan sólo 16384 partículas), y a la introducción de las operaciones atómicas en el código para evitar las condiciones de carrera. En otros ordenadores, podrás tener una mejora adicional con esta técnica entre el 5 % y el 40 %.

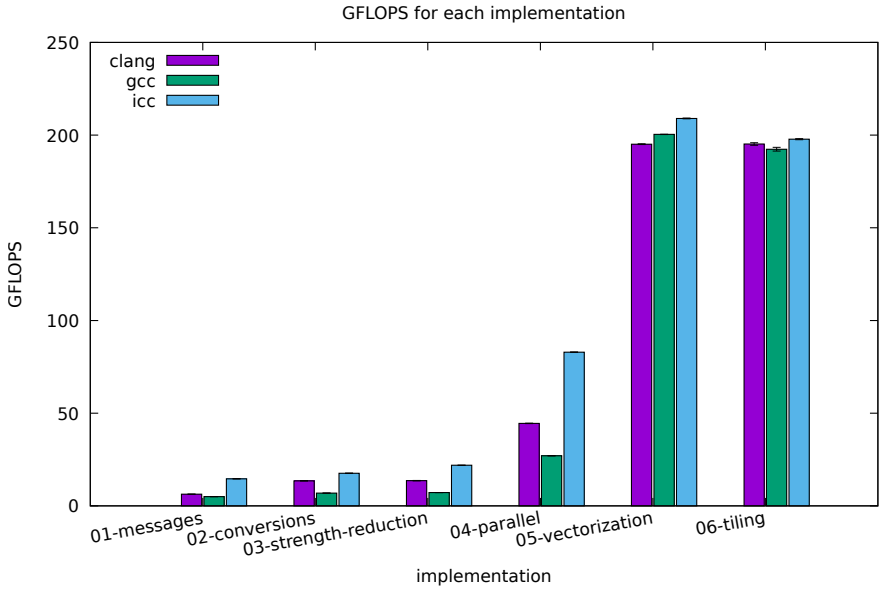
Como último punto, vamos también a ver qué resultados obtenemos al aplicar la herramienta `perf` a este código optimizado. Como podemos ver, hemos reducido significativamente el número de accesos a la cache L1 (de datos), lo que se ha traducido en una reducción en el número de ciclos de reloj y por tanto en el tiempo de ejecución. Es también muy interesante en este momento examinar cómo han variado los valores de los diversos eventos relacionados con la caché. Por ejemplo, en el caso de los accesos a la cache de último nivel (evento *cache-references*), estos accesos se han reducido por un factor en torno a 10X de la versión inicial (versión 00) a esta última versión (versión 05) donde estamos aplicando la técnica del *tiling*.

En ocasiones, es mejor aplicar la técnica del *tiling* al bucle *i*. A esta variante del *tiling* se le denomina *register blocking* o *unroll-and-jam*. Para ello, tenemos que modificar el código para intercambiar los bucles y vectorizar en *i*. Además de esto, a veces ayuda desenrollar el bucle en *j*. Esta alternativa no se desarrolla más pues está fuera del ámbito de esta práctica.

4.2. Conclusiones

Como conclusión de este caso práctico, hemos visto cómo se ha obtenido una mejora muy importante en el rendimiento de una aplicación intensiva en cómputo, gracias al conocimiento de la arquitectura del procesador y de aplicar unas *sencillas* optimizaciones que se desprenden de dicho co-

[9] Si no lo hacemos, es habitual que tengamos pocos hilos y, por tanto, nos descienda el rendimiento paralelo de la aplicación por utilizar un número suboptimo de hilos.



	Compilador		
Versión	GCC	Clang	ICC
01-messages	5.00±0.00	6.30±0.00	14.60±0.10
02-conversions	6.90±0.00	13.50±0.10	17.60±0.10
03-strength-reduction	7.20±0.00	13.60±0.10	21.90±0.10
04-parallel	27.00±0.00	44.50±0.10	82.90±0.10
05-vectorization	200.40±0.10	195.10±0.20	209.00±0.20
06-tiling	192.40±1.00	195.20±0.70	197.80±0.30

Figura 1: Resultados de rendimiento de las diferentes versiones del programa en `compiladorintel.inf.um.es`. En la tabla se muestan los GFLOPS alcanzados por cada compilador.

nocimiento. En la figura 1 se muestran los resultados de rendimiento obtenidos durante la práctica en `compiladorintel.inf.um.es`.

5. Ejercicios pedidos

A continuación se te pide que realices un proceso similar al mostrado en el caso de uso para otro ejemplo habitual. En el archivo adjunto a esta tarea encontrarás un micro-kernel que resuelve el problema del *binning*.

Vamos a explicar brevemente en qué consiste este problema. Supongamos que tenemos datos provenientes de una simulación o de un experimento sobre partículas en movimiento en un detector de partículas cilíndrico. Las posiciones de las partículas se obtienen en coordenadas polares, y queremos agrupar las partículas en grupos/contenedores (*bins*) definidos en coordenadas cartesianas.

Ejemplos de aplicación de algoritmos similares a este se pueden encontrar en la física de partículas (por ejemplo, para detectar huellas de partículas), en simulaciones de Montecarlo y también en problemas estadísticos que traten con la transformación y el agrupamiento de datos.

Dentro de la carpeta *ejercicios* vete al subdirectorio *binning*. El directorio «`00-reference`» contiene la versión inicial del programa (versión de referencia), la cual no se debe modificar. Dicha versión inicial contiene el fichero denominado *binning.cpp*. que es el fichero a optimizar, junto a otros ficheros de utilidades y el fichero principal *main.cpp*. Adicionalmente, se te adjuntan 2 scripts, el script «`benchmark-binning`» que te puede ayudar a comprobar la corrección y medir el rendimiento de las soluciones, y el script «`benchmark-binning-plots`» para dibujar las gráficas de comparación de las soluciones.

Se te pide que realices los ejercicios siguientes:

1. En primer lugar, empieza por familiarizarte con las características básicas de la aplicación. Compila con el `Makefile` adjunto y obtén el rendimiento base en el ordenador en el que vayas a hacer esta práctica. Con respecto al compilador a usar, sólo es necesario que utilices el `icc`, aunque también puedes usar algún otro compilador. A continuación, vamos a pasar al proceso de optimización.

2. Paralelización. Optimiza el rendimiento de este micro-kernel usando OpenMP para paralelizar su ejecución. Ten en cuenta que pueden aparecer condiciones de carrera. Evalúa la aceleración que has obtenido en función del número de núcleos que utilizas y comenta los resultados obtenidos.

3. Vectorización. Optimiza el rendimiento de este micro-kernel usando el compilador para vectorizar automáticamente su ejecución. Observa el informe del compilador para encontrar los problemas que pueden aparecer para realizar dicha vectorización y estudia si es adecuado cambiar de los datos de AoS a SoA. Evalúa la aceleración que has obtenido en función del tamaño de la unidad vectorial y del número de unidades vectoriales que tengas, y comenta los resultados obtenidos.

4. Optimización del acceso a memoria. Estudia si el rendimiento de este micro-kernel se puede mejorar optimizando el acceso a memoria que hace. ¿Se podría hacer *loop tiling* para mejorar el rendimiento? Comenta los resultados obtenidos.

5. ¿Afecta el tamaño del problema a la solución obtenida?¿Qué pasaría si aumentamos el tamaño del número de *Bins* en los que clasificamos los datos de entrada? Comenta tu opinión al respecto.

Desarrolla tus soluciones en subdirectorios del directorio «`solutions`», usando un directorio con un nombre adecuado para cada paso (encontrarás un subdirectorio de ejemplo con una versión casi idéntica a la versión de referencia). Al final debe haber al menos 3 nuevos subdirectorios de «`solutions`», uno por cada uno de los 3 ejercicios de optimización.

Para la explicación de los apartados anteriores, haz uso de la herramienta `perf` para justificar tus respuestas. Además, añade al final una gráfica similar a la figura 1 para mostrar una comparativa de los resultados de rendimiento obtenidos en las diferentes optimizaciones.

Para la realización de esta práctica, se puede utilizar uno de los ordenadores del laboratorio o cualquier otro ordenador disponible (en especial si tiene 4 núcleos o más).

5.1. Entrega

Tras finalizar los apartados anteriores, en esta práctica se te pide lo siguiente:

1. Documento en PDF donde ofrezcas las respuestas a los ejercicios solicitados.
2. Además, elaborar un informe donde comentes brevemente los aspectos positivos de la práctica, así como aquellos negativos o cosas que has echado en falta en la misma. La longitud del informe no debe exceder una cara de folio. Entrega el informe en un fichero PDF.
3. El código fuente de cualquier programa o script desarrollado (o modificado) para la realización de la práctica.
4. 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.

Utilizando la herramienta de *Tareas* del Aula Virtual envía 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.

5.2. Recursos

En la zona de Recursos del Aula Virtual, dentro de la carpeta denominada *Practicas/Practica*3 puedes encontrar todo lo necesario para realizar esta práctica.

6. Criterios de valoración

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

- Realización y corrección de los apartados pedidos (*6 puntos*)
- Presentación y claridad de las explicaciones (*1 punto*)
- Aportar ideas originales en las explicaciones a los apartados realizados (*1 punto*)
- Realizar 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

- Esta práctica toma ideas del capítulo 4 del libro «*Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors*». 2nd Edition. Colfax International, 2015.
- La página web de *The National Center for Supercomputing Applications (NCSA)* `http://www.ncsa.illinois.edu` tiene unos tutoriales muy interesantes (`https://www.citutor.org/browse.php`) relacionados con la vectorización, paralelización, evaluación de rendimiento, etc
- Asimismo, en la página web de *The Cornell University Center for Advanced Computing (CAC)* `https://cvw.cac.cornell.edu/topics` hay un material relacionado con la vectorización y paralelización de aplicaciones muy adecuado.
- El tutorial «*Optimising Serial Code* de Pawsey Supercomputing Center, y el resto de Webinars sobre programación eficiente son muy interesantes y los puedes encontrar en: `https://pawseysc.github.io/code.html`.