Práctica 3: Optimización del problema del Binning

Jose Antonio Lorencio Abril

0. Introducción

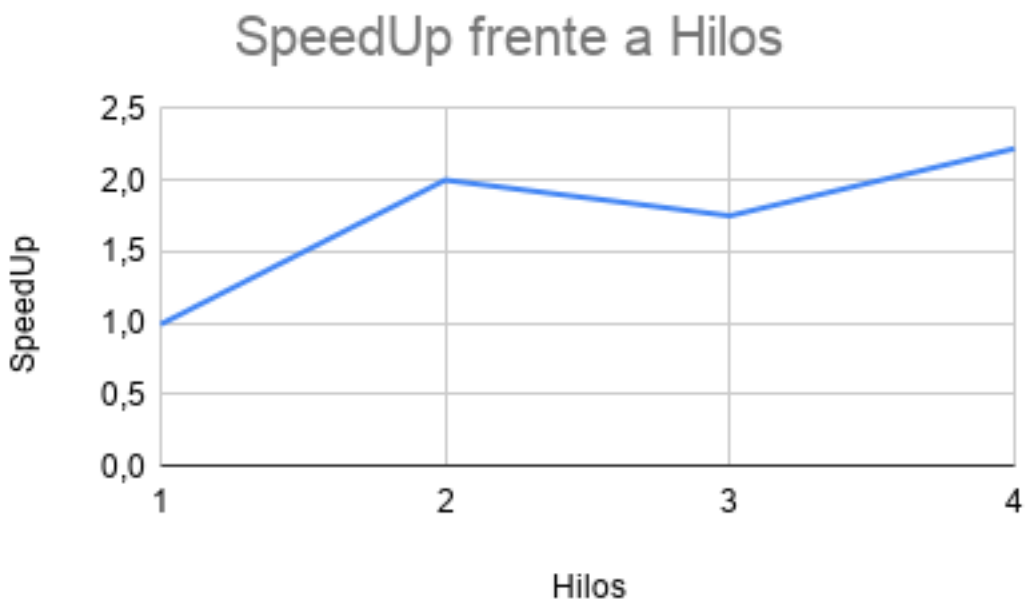
Para realizar esta práctica he usado el compilador de intel instalado en mi máquina. Mi procesador es un intel i5-7200U con arquitectura Kaby Lake y 2.5GHz. El compilador usado es el intel icpc versión 19.1.3.304.

1. Paralelización

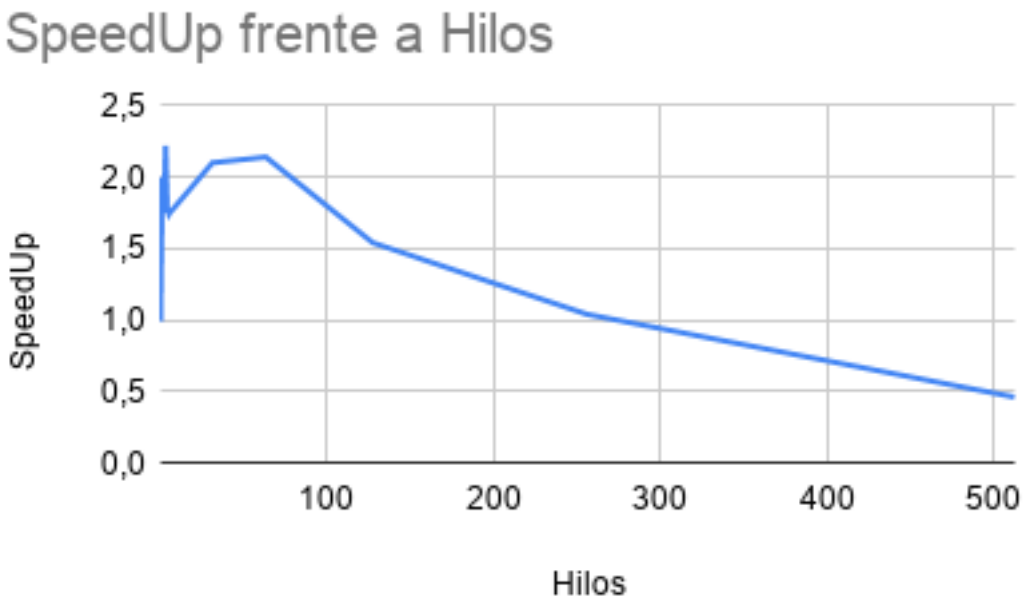
Para realizar la paralelización, debemos tener cuidado con los accesos a outPutBins, ya que los índices se calculan en cada iteración del array, lo que imposibilita que podamos hacer arreglos de antemano. Por ello, para paralelizar, debemos crear variables privadas de reducción, que combinaremos después (de forma atómica) para obtener el resultado correcto. En el siguiente fragmento de código se detallan los cambios realizados:

```
1  void BinParticles(const InputDataType  & inputData,
                     BinsType & outputBins)
   {
     #pragma omp parallel num_threads(4)
     {
6          //Private variable for reduction
       BinsType reductionOutputBins;
       for(int i=0; i<nBinsX;i++)
         for(int j=0; j<nBinsY;j++)
           reductionOutputBins[i][j] = 0;
11
       #pragma omp for
             for (int i = 0; i < inputData.numDataPoints; i++) {
               // Transforming from cylindrical to Cartesian coordinates:
               const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
16             const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);

               // Calculating the bin numbers for these coordinates:
               const int iX = int((x - xMin)*binsPerUnitX);
               const int iY = int((y - yMin)*binsPerUnitY);
21
               // Incrementing the appropriate bin in the counter
               ++reductionOutputBins[iX][iY];
             }

26     //Updating of outPutBins
       for(int i=0; i<nBinsX; i++)
         for(int j=0; j<nBinsY; j++)
           #pragma omp atomic
           outputBins[i][j] += reductionOutputBins[i][j];
31   }
   }
```

Y en la siguiente gráfica vemos la aceleración obtenida al variar el número de hilos:



Como mi ordenador tiene 4 núcleos, aumenté num_threads hasta llegar a 4. Pero, después quise comprobar el comportamiento al aumentar el número de hilos, pensando que iba a empeorar claramente. Sin embargo, los resultados obtenidos son los siguientes:



Y vemos como, aunque sí que parece que hay un empeoramiento en la eficiencia del programa, llega un punto, con 64 hilos, donde casi se iguala el speedup al obtenido con 4 hilos. Esto me hace pensar que este programa es muy susceptible a ser paralelizado, y que la eficiencia mejoraría considerablemente en un ordenador con muchos más núcleos. De hecho, no se ve un empeoramiento respecto la eficiencia base hasta superar los 256 núcleos, donde supongo que el trabajo de cada hilo es demasiado pequeño, y no compensa paralelizar. Según los datos, si tuviésemos 64 núcleos, probablemente obtendríamos la mayor aceleración con una cantidad de hilos rondando los 64 hilos

(eso no es tan obvio como parece, no siempre aumentar el número de hilos de procesamiento mejora el rendimiento).

2. Vectorización

Al vectorizar automáticamente, sin hacer cambio alguno en el programa, vemos que no puede vectorizar el bucle en el que se desarrolla el trabajo central del programa, porque no puede asegurar que no haya dependencias. Esto se debe a que los índices iX e iY se calculan en cada iteración y son impredecibles. Por tanto, esa parte del bucle no va a poder ser vectorizada. Así, lo que hacemos es separar el bucle en dos partes (loop-splitting), una será vectorizable y la otra no. Para ello, creamos dos arrays que guarden la información sobre los índices iX e iY, pero esto tiene un claro problema, y es que estamos necesitando 2 arrays adicionales (¡en cada hilo!) tan grandes como el array de datos, lo que significa un aumento significativo de uso de memoria. De hecho, el aumento es tal que, para el tamaño de nuestro problema, vamos a necesitar una excesiva cantidad adicional de memoria, así como aumentar el tráfico de datos.

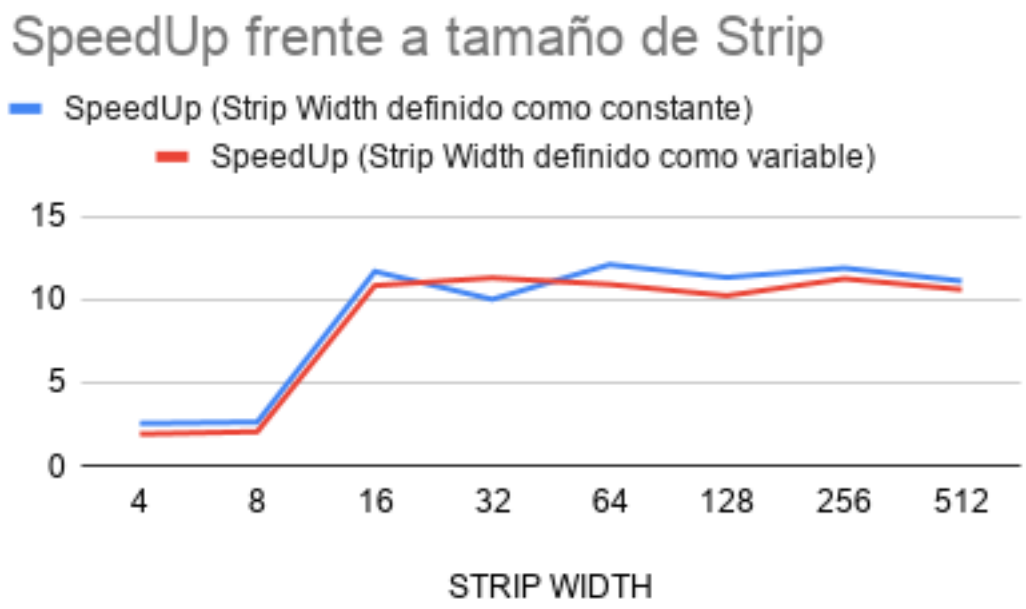
Para solucionar esto, podemos ahora usar strip-mining, o sea, dividir el bucle en bloques de igual tamaño, y dividir el trabajo en bucles a lo largo de cada bloque. De esta forma pertimitimos la vectorización automática, combinada con la paralalelización, y reducimos la carga de memoria, pues ahora podemos crear los arrays de índices de pequeño tamaño e ir haciendo los cálculos poco a poco, en cada bloque. Generalmente, debemos tener en cuenta que el tamaño de strip puede no ser divisor del tamaño de los datos, por lo que debemos asegurarnos de completar el bucle con los datos de cola que queden al final. En nuestro caso, vamos a escoger strips con tamaños potencia de 2, y el tamaño del problema es 2^{26}, por lo que no es necesario implementar estos cómputos de cola.

Por último, los arrays podemos definirnos para que queden alineados. Así, en el main cambiamos la definición de los arrays r y phi, usando _m_alloc y al final los liberamos con _m_free, y con tamaños 16, 32 y 64. Hacemos lo propio en el bucle donde se realiza el cuerpo del trabajo, para que los datos queden alineados, y usamos el #pragma vector aligned, para que el compilador se ahorre las comprobaciones de alineamiento.

El código, entonces, queda:

```
void BinParticles(const InputDataType  & inputData,
                        BinsType & outputBins) {
  #pragma omp parallel num_threads(4)
  {
        //Private variable for reduction
    BinsType reductionOutputBins;
    #pragma simd
    for(int i=0; i<nBinsX;i++)
      for(int j=0; j<nBinsY;j++)
        reductionOutputBins[i][j] = 0;

    #pragma omp for
    for(int ii=0; ii<inputData.numDataPoints; ii+=STRIP_WIDTH){
      int iX[STRIP_WIDTH] __attribute__((aligned(64)));
      int iY[STRIP_WIDTH] __attribute__((aligned(64)));

      const FTYPE* r = &(inputData.r[ii]);
      const FTYPE* phi = &(inputData.phi[ii]);

      #pragma vector aligned
      for(int c=0; c<STRIP_WIDTH; c++){
        const FTYPE x = r[c]*COS(phi[c]);
              const FTYPE y = r[c]*SIN(phi[c]);
        iX[c] = int((x-xMin)*binsPerUnitX);
        iY[c] = int((y-yMin)*binsPerUnitY);
      }

      for(int c=0; c<STRIP_WIDTH; c++){
        reductionOutputBins[iX[c]][iY[c]]++;
      }
    }
    //Updating of outPutBins
    for(int i=0; i<nBinsX; i++)
      for(int j=0; j<nBinsY; j++)
        #pragma omp atomic
        outputBins[i][j] += reductionOutputBins[i][j];
  }
}
```

Vamos a ver los resultados obtenidos. Antes de introducir el alineamiento de datos, debemos determinar el tamaño del strip óptimo. Además, comparo los resultados al definir este tamaño como una constante en el .h, o como una variable entera declarada justo antes de su uso:



Vemos que es, generalmente, mejor definir el tamaño del strip como una constante. Además, el tamaño óptimo parece que será bien 16 o bien 64, con aceleraciones cercanas a 12X.

Al introducir el alineamiento de datos con alineamiento de 64 bytes, obtenemos una aceleración de 10X para un tamaño de strip 16, y de 11X para un tamaño de strip 64. Para los alineamientos menores obtenemos peores resultados.

Por tanto, parece que nuestra mejor opción es no usar alineamiento de datos, y que es ligeramente mejor utilizar un tamaño de strip de 64. La explicación del empeoramiento al introducir manualmente el alineamiento de datos, debe deberse a que los datos están mayoritariamente alineados. Esto es una característica circunstancial de este problema, no algo que vaya a suceder de forma general.

3. Optimización del acceso a memoria

Para optimizar el acceso a memoria, debemos encontrar patrones de acceso. Vamos a ver si podemos utilizar alguna de las técnicas estudiadas:

- **Cache blocking:** para usar caché blocking, debemos detectar datos que entran y salen de caché varias veces, y reordenar el código para que esto suceda lo menos posible. Ahora bien, estamos recorriendo dos arrays lineales, en orden, por lo que esta técnica no es aplicable, ya que cuando saquemos un bloque de memoria, es porque ya habremos recorrido todos los elementos anteriores en el array, y no volverán a entrar (cada elemento es accedido una única vez).
- **Register blocking:** no es aplicable por la misma razón, al recorrer dos arrays lineales de forma lineal una única vez.

Así, parece ser que no vamos a poder mejorar el rendimiento en cuanto a accesos a memoria. Esto tiene mucho sentido, porque al final lo que tenemos es un array de datos no relacionados, y que no interactúan entre sí. Los cálculos que hacemos son individuales para cada punto y solo se hacen una vez.

Al reducir el tamaño de nBinsX y nBinsY a la mitad, no parecen apreciarse diferencias en cuanto al speedup obtenido. Pero sí se ve cómo aumentan los GP/s, esto se debe a que estamos escogiendo una menor cantidad de conjuntos en los que encasillamos las partículas, por lo que es predecible que se hará el trabajo ligeramente más rápido.

Al aumentar el tamaño del problema, se obtiene un evidente aumento del tiempo de ejecución, y, además, se aumenta el speedup obtenido con las optimizaciones, es decir, al aumentar el tamaño del problema el efecto de las optimizaciones aumenta. Esto no termino de comprenderlo, ya que el porcentaje del programa paralelizable es el mismo, y la vectorización se produce por igual, pues en todos los casos hay una cantidad de datos múltiplo del tamaño del vector. A partir de tamaño 2^{30} no puedo ejecutarlo, ya que mi ordenador tiene 8GB de RAM, y debería hacer cambios en el programa para poder tener más datos que memoria. Los speedups medios obtenidos han sido:

N	SpeedUp
2^{26}	11.8
2^{27}	12
2^{28}	12.2
2^{29}	12.3

La mejora es pequeña, pero hay mejora.

4. Comentarios sobre la práctica

En esta práctica hemos conseguido mejorar un programa con una aceleración de 12X. Vemos como las técnicas vistas en clase, efectivamente son útiles, y sí que mejoran el uso de recursos de nuestros programas.

Hemos visto también la importancia que tiene la elección de las zonas a paralelizar, así como la influencia que tiene la forma en la que realizamos los cálculos para facilitarle al compilador la vectorización. Esto son aspectos que hasta ahora no habíamos tenido en cuenta y que, ciertamente, tienen una gran influencia en la eficiencia de nuestros programas. También vemos en esta práctica como vectorización y paralelización son técnicas que se llevan bien, ya que la vectorización se hace a nivel de núcleo, esto es un detalle que puede mejorar enormemente el rendimiento de nuestros programas (si solo usamos paralelización, obtenemos una mejora de ~2.3X, y si solo usamos vectorización, la mejora es de ~5.25X, pero si usamos ambas técnicas juntas obtenemos una mejora de ~12X).

Además, me parece que por lo general, la aplicación de todas estas técnicas y conceptos, entra en conflicto con la legibilidad del código, por lo que la correcta documentación se va a hacer especialmente importante cuando las usemos.

Por otro lado, el ejercicio 3 me ha parecido que tiene poco sentido, ya que las técnicas no son aplicables por la naturaleza del problema. Quizás estaría bien cambiar el problema de la práctica y utilizar uno en el que sea aplicable alguna de las técnicas de mejora del acceso a memoria, además de las técnicas de paralelización y vectorización, con las que en realidad ya hemos tratado en las prácticas anteriores.

5. Indagación adicional

Muchas veces he escuchado la enorme cantidad de datos que se produce en el CERN en cada segundo, y la dificultad que tiene su procesamiento, así que he estado buscando y he encontrado un

documento de este centro de investigación en el que tratan precisamente las técnicas estudiadas para un problema real de física. El documento puede encontrarse en
`https://cds.cern.ch/record/282614/files/AT00000115.pdf`

En la página 31 encontramos el problema de La formulación en retículo de la cromodinámica cuántica, que, como explican, fue uno de los problemas que motivaba el desarrollo de las técnicas que hemos estudiado en esta asignatura. A lo largo de este caso de estudio, vemos cómo se usan técnicas de paralelización para varios propósitos, como la generación aleatoria de las partículas en un retículo de 4 dimensiones para estudiar su desarrollo con el tiempo, o los cálculos de las interacciones entre las diferentes partículas del retículo. Aunque el artículo no entra mucho en temas de vectorización, sí que se menciona su importancia en varias ocaciones a lo largo del documento, y, para nosotros, tras el trabajo realizado, debe ser evidente. Por ejemplo, los gluones se representan como matrices 3x3, y para calcular las paridades deben hacerse sumas, que son susceptibles de ser vectorizadas. Además, la actualización de los gluones en cada paso también se puede paralelizar.

Vemos así como queda patente que lo que hemos visto en la asignatura no son ideas esotéricas que nunca vamos a poner en práctica, sino que, al revés, vienen motivadas por problemas reales y son un activo campo de investigación en sí mismo, y de aplicación en diversos campos en los que es importante la eficiencia de cómputo y de acceso a memoria, como es la trata de grandes conjuntos de datos o simulaciones costosas (por ejemplo, la simulación de un sistema cuántico escala con un factor de n^m, donde n es el número de partículas cuánticas y m el número de estados de cada partícula, como podemos imaginarnos, estas simulaciones son increiblemente difíciles de realizar).