INFORME

Elena Pérez González-Tablas

SEMANA1

Casi de todas las llamadas que tenemos en el programa de la sesión de prácticas

```
 -> INLINE: (87,5) MoveParticles(int, ParticleType *, float) (isz = 149) (sz = 158)
```

Las variables que tienen un valor fijo el compilador las ha podido hacer desaparecer

```
   -> EXTERN: (31,31) pow(double, double)
```

Va a permitir que el bucle donde está sea vectorizable, si no como hay una llamada a una función no se podría vectorizar.

VECTORIZAR

```
for(int i = 0; i < nParticles; i++) {
    particle[i].x = rand()/RAND_MAX;
    particle[i].y = rand()/RAND_MAX;
    particle[i].z = rand()/RAND_MAX;
    particle[i].vx = rand()/RAND_MAX;
    particle[i].vy = rand()/RAND_MAX;
    particle[i].vz = rand()/RAND_MAX;
  }
LOOP BEGIN at nbody.cc(68,3)
   remark #15382: vectorization support: call to function rand() cannot be vectorized   [
nbody.cc(69,21) ]
   remark #15382: vectorization support: call to function rand() cannot be vectorized   [
nbody.cc(70,21) ]
   remark #15382: vectorization support: call to function rand() cannot be vectorized   [
nbody.cc(71,21) ]
   remark #15382: vectorization support: call to function rand() cannot be vectorized   [
nbody.cc(72,22) ]
   remark #15382: vectorization support: call to function rand() cannot be vectorized   [
nbody.cc(73,22) ]
   remark #15382: vectorization support: call to function rand() cannot be vectorized   [
nbody.cc(74,22) ]
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed OUTPUT dependence between call:rand() line 69 and
call:rand() line 74
   remark #15346: vector dependence: assumed OUTPUT dependence between  line 74 and  line 69
LOOP END
```

No lo ha vectorizado porque está la llamada a la función rand, pero no afecta al rendimiento porque solamente se mide el tiempo de ejecución del procedimiento MoveParticles.
El compilador no puede vectorizarlo porque hay dependencias entre iteraciones, puesto que los números son pseudoaleatorios, obtiene los resultados de forma cíclica y muy difícil de predecir. Se coge el resultado anterior o algún dato relacionado con este que se llama semilla, se realizan una serie de operaciones sobre esa semilla y esas operaciones dan lugar al número aleatorio y se actualiza la semilla que se utilizará en la siguiente llamada a la función rand.

```
// Loop over particles that experience force
  for (int i = 0; i < nParticles; i++) {

    // Components of the gravity force on particle i
    float Fx = 0, Fy = 0, Fz = 0;

    // Loop over particles that exert force: vectorization expected here
    for (int j = 0; j < nParticles; j++) {

      // Avoid singularity and interaction with self
      const float softening = 1e-20;

      // Newton's law of universal gravity
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
      const float drSquared = dx*dx + dy*dy + dz*dz + softening;
      const float drPower32 = pow(drSquared, 3.0/2.0);

      // Calculate the net force
      Fx += dx / drPower32;
      Fy += dy / drPower32;
      Fz += dz / drPower32;
    }

    // Accelerate particles in response to the gravitational force
    particle[i].vx += dt*Fx;
    particle[i].vy += dt*Fy;
    particle[i].vz += dt*Fz;
  }   LOOP BEGIN at nbody.cc(21,5) inlined into nbody.cc(87,5)
   <Remainder loop for vectorization>
   LOOP END
LOOP END

LOOP BEGIN at nbody.cc(15,3) inlined into nbody.cc(87,5)
   remark #25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due to other Compiler
Transformations)
   remark #25451: Advice: Loop Interchange, if possible, might help loopnest. Suggested Permutation : ( 1 2 ) --> ( 2 1 )
   remark #25236: Loop with pragma of trip count = 16384 ignored for large value
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at nbody.cc(21,5) inlined into nbody.cc(87,5)
        remark #15415: vectorization support: gather was generated for the variable particle_1785:  strided by 6   [ nbody.cc(27,24) ]
     remark #15415: vectorization support: gather was generated for the variable particle_1785:  strided by 6   [ nbody.cc(28,24) ]
     remark #15415: vectorization support: gather was generated for the variable particle_1785:  strided by 6   [ nbody.cc(29,24) ]
     remark #15305: vectorization support: vector length 8
     remark #15309: vectorization support: normalized vectorization overhead 0.300
     remark #15417: vectorization support: number of FP up converts: single precision to double precision 2   [ nbody.cc(31,29) ]
     remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   [ nbody.cc(31,29) ]
     remark #15300: LOOP WAS VECTORIZED
     remark #15460: masked strided loads: 3
     remark #15475: --- begin vector loop cost summary ---
     remark #15476: scalar loop cost: 191
     remark #15477: vector loop cost: 40.000
     remark #15478: estimated potential speedup: 4.510
     remark #15487: type converts: 3
     remark #15488: --- end vector loop cost summary ---
   LOOP END
```

Dos bucles anidados, el bucle externo no se ha vectorizado porque es un bucle externo, pero el interno si. La idea de vectorizar, nos interesa buscar el paralelismo de grano más fino. Paralizar sería al contrario hasta cierto punto, por el balance de carga.
Ha intentado hacer intercambio de bucles pero no ha podido hacerlo porque los bucles no están perfectamente anidados porque hay algunas instrucciones que están fuera del bucle interno por lo cual no se puede aplicar esa transformación directamente.
// BUCLE INTERNO
remark #15300: LOOP WAS VECTORIZED
Si miramos el código ensamblador veremos que no hay ninguna llamada a la función pow si no que se ha hecho otra cosa para poder vectorizar. Sin embargo, la vectorización no es la mejor porque ha tenido que insertar instrucciones de gather.
remark #15415: vectorization support: gather was generated for the variable particle_1785: strided by 6 [nbody.cc(27,24)]
remark #15415: vectorization support: gather was generated for the variable particle_1785: strided by 6 [nbody.cc(28,24)]
remark #15415: vectorization support: gather was generated for the variable particle_1785: strided by 6 [nbody.cc(29,24)]
Lo veremos en la última de las transformaciones que haremos hoy pero lo que nos viene a decir es que para poder vectorizarlo necesita acceder a los campos x de las diferentes partículas.
const float dx = particle[j].x - particle[i].x;
Cuando se vectorial esta línea es hacer el trabajo de ocho iteraciones en una:
remark #15305: vectorization support: vector length 8
Sería tener una variable vectorial dx con ocho componentes:
const float dx0 = particle[j].x - particle[i].x;
const float dx1 = particle[j+1].x - particle[i+1].x;
const float dx2 = particle[j+2].x - particle[i+2].x;
const float dx3 = particle[j+3].x - particle[i+3].x;
const float dx4 = particle[j+4].x - particle[i+4].x;
const float dx5 = particle[j+5].x - particle[i+5].x;
const float dx6 = particle[j+6].x - particle[i+6].x;
const float dx7 = particle[j+7].x - particle[i+7].x;
Cada una calcula el resultado de una iteración, los ocho valores particle de j hasta j+7 van a un registro y los valores particle de i hasta i+7 van a otro. El resultado de la resta vectorial se guarda en otro registro vectorial. La operación es una resta de ocho elementos.
Las variables están almacenadas en memoria y como la estructura está definida así cuando se llama a new se crea un array en el que básicamente está para cada partícula:
x y z vx vy vz | x y z vx vy vz | x y z vx vy vz | x y z vx vy vz | x y z vx vy vz | x y z vx vy vz | x y z vx vy vz | x y z vx vy vz

Tenemos que leer las x e ir a esos registros, el problema que cada x de cada partícula están en una dirección de memoria no consecutivos, Hay que avanzar seis elementos del array “

strided by 6 ” 6*4 porque estamos trabajando con floats y por tanto necesita utilizar una instrucción de gather que peine acceder de forma arbitraria. Son más complejas y tardan más en ejecutarse.

Gather -> recoger lw
Scather -> esparcir sw

remark #15417: vectorization support: number of FP up converts: single precision to double precision 2 [nbody.cc(31,29)]
remark #15418: vectorization support: number of FP down converts: double precision to single precision 1 [nbody.cc(31,29)]
remark #15487: type converts: 3

Conversiones de tipo que a lo mejor no se ven a simple vista. La función pow recibe dos argumentos de tipo double y devuelve un double pero nosotros le pasamos un argumento de tipo float, lo transforma y requiere una instrucción adicional. Hay instrucciones vectoriales para realizar las conversiones y “3.0/2.0” también hay que convertirlo. El resultado hay que transformarlo de double a float.
const float drPower32 = powf(drSquared, 3.0/2.0);
Añadiendo una f a pow solucionamos el problema porque ya está definida está función.
// Aplicación para comparar ficheros
```
$ meld nobody.optrpt.0 nobody.optrpt
```
Han desaparecido los avisos de las conversiones y el coste del bucle ha disminuido y el speedup es mayor
Pasamos de 15,6 GFLOPs a 16,7 GFLOPs que más o menos es un 6%, aumenta el rendimiento. Es el primer ejemplo de una optimización que el compilador no puede hacer.
La segunda cuestión de nuestra optimización escalar, en nuestro programa la mayoría del tiempo lo pasa en las instrucciones que calculan la expresión del incremento de Fx, Fy y Fz.
La expresión (1:18:56) la podemos sustituir por esta otra expresión equivalente y lo hacemos porque es más fácil de calcular (Pag 5).
El objetivo final es utilizar la instrucción vrsqrtps. Esta expresión solo realiza una división para calcular el recíproco de una raid cuadrada, antes se dividía 3 veces y el divisor solo se calculaba una vez. La división desaparece y ahora hay una multiplicación por el recÍproco
// Reducción de fuerza
Desaparece la función powf y las divisiones se transforman en multiplicaciones
El rendimiento aumenta de 16,7GFLOPs a 19,4GFLOPs
LA tercera ventaja que era utilizar la instrucción vrsqrtps el compilador ya la estaba haciendo, no tenía tan buen resultado como al hacer las modificaciones manualmente porque no parece que aproveche la ventaja para hacer una multiplicación por el recíproco
//PARALELIZACIÓN
No hay dependencias para paralizar el bucle i
Los campos que se leen y escriben de las partículas son distintos
La única variable (array) compartido al que acceden, el resto de variables son privadas
Añadiendo un pragma encima del bucle externo
#pragma amp parallel for
El rendimiento aumenta de 19,4 GFLOPs a 71,5 GFLOPs
Sppeud de 3,75 con 4 cores

// MAIN
En el main paralizamos la inicialización
De esta manera conseguimos poco porque en este código no es importante para este paso, en las siguientes secciones tendrá su importancia
// EL PROBLEMA DE LOS GATHER
No es una transformación sencilla,
Hay que cambiar el tipo de dato, en vez de una estructura representando una sola partícula lo convertimos a una estructura que defina un conjunto de partículas y en ese conjunto tenemos seis arrays uno para cada campo. El cambio es mecánico.
Reservamos ahora seis arrays en vez de uno pero con la función _mm_ malloc de intel para indicar que los queremos alineados en 32 bytes (múltiplo de 32 bytes), aunque sería mejor usar la función POSIX posix_memalign.
Evitamos así el looping y del resto de iteraciones del bucle
72,6 -> 201,7 (El triplete rendimiento)

SEMANA2

Este bucle no está siendo vectorizado porque hay dependencias entre las variables particle. Mismo problema que antes, leemos y escribimos de la variable particle. El compilador supone que hay dependencia de datos y la hay porque estamos leyendo para hacer las sumas, particle de x para sumarla con la solución de la multiplicación a esa misma variable. Aunque existen dependencias, no afectan a las distintas iteraciones. El bucle si se podría vectorizar, tendremos que añadir un pragma manualmente: “
#pragma simd
#pragma vector aligned”

```
// Move particles according to their velocities
  // O(N) work, so using a serial loop
  for (int i = 0 ; i < nParticles; i++) {
    particle[i].x  += particle[i].vx*dt;
    particle[i].y  += particle[i].vy*dt;
    particle[i].z  += particle[i].vz*dt;
  }
LOOP BEGIN at nbody.cc(49,3) inlined into nbody.cc(106,5)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed ANTI dependence between particle_1785
line 50 and particle_1785 line 52
   remark #15346: vector dependence: assumed FLOW dependence between particle_1785
line 52 and particle_1785 line 50
LOOP END
```

201,7 -> 211 (El cambio del rendimiento no es significativo porque el bucle que hemos paralizado es una parte muy pequeña del código).

// OPTIMIZANDO EL ACCESO A MEMORIA

Vamos a mejorar la localidad temporal. La intensidad aritmética mide cuantos cálculos hacemos con un dato cuando nos lo traemos de memoria. ¿Cual es la intensidad aritmética ideal para nuestro procesador? Para conseguir el rendimiento pico, hay que hacer muchos cálculos con cada uno de los datos que me traigo de memoria, 92 operaciones por cada operando.
En el bucle interno se hacen tres accesos a memoria (a caché), porque son los que se utilizan para acceder a los datos nuevos y aproximadamente 19 operaciones aritméticas. Por tanto, la IA es 6,3. No hemos contado los accesos a memoria de "i" porque aciertan en caché ya que estamos reutilizando continuamente la misma partícula. En el peor de los casos, los accesos a caché fallan siempre y solo en ese caso la IA es 6,3 pero casi siempre acertaran en caché, como mínimo 15 de cada 16 veces acierta por la localidad espacial en la caché.

```
const float dx = particle.x[j] - particle.x[i];
const float dy = particle.y[j] - particle.y[i];
const float dz = particle.z[j] - particle.z[i];
```

// DOS FORMAS DE APLICAR LA TÉCNICA TILING

1- UTILIZANDO COMO APOYO EL BUCLE EXTERIOR, PERO LA TÉCNICA SE APLICA A LOS DOS BUCLES A LA VEZ

Primero hacemos un "strip-mine" que es partir un bucle en bucles más pequeños de forma secuencial, es una transformación que realmente no tiene ningún efecto en cuanto al orden de las iteraciones pero nos sirve para después aplicar otra transformación. Convertimos el bucle j en dos bucles, el primero recorre desde 0 hasta n pero no de uno en uno si no de TILE en TILE, donde TILE es una constante y para cada una de las iteraciones del bucle se recorre los elementos individuales. Una vez hemos partido el bucle en dos, ahora es cuando viene la reordenación de iteraciones, cogemos la parte del bucle que se encarga de las tiras y se mueve a la parte más externa de la animación de bucles.
Ahora lo conveniente es paralelizar los dos bucles externos
"#pragma amp parallel for collapse(2)"

La variable TILE se le asigna el valor 4096 porque es lo que nos permite que los tres arrays que trabajamos principalmente nos quepan en la cachee primer nivel. 4096 elementos del array x son unos 16 k y como son tres arrays 16*3 = 48. La idea es buscar un tamaño TILE que nos minimice el número de fallos y ese tamaño va a ser lo que haga que la cantidad de datos que tenemos que leer de la memoria nos quepa en la caché de segundo nivel.

Ahora hay que calcular la aceleración varias veces porque el bucle interno lo hemos partido en tiras y al final de cada tira el resultado parcial hay que acumularlo en las variables, vx, vy, vz. Además, como los bucles exteriores los estamos ejecutando en paralelo, esas iteraciones se pueden producir en cualquier orden e incluso se pueden estar calculando dos iteraciones de la misma partícula en paralelo, se puede producir condición de carrera. Para asegurarnos el correcto funcionamiento, añadimos el pragma "#pragma omp atomic" tres veces ya que son secciones críticas y puede dar lugar a resultados incorrectos.

Se queda con casi el mismo rendimiento porque aunque el análisis indicaba que el programa lo estaba haciendo fatal en cuanto a la localidad temporal, en el fondo gracias a profeching y a la localidad espacial, no hay tantos fallos de caché como el boletín pretende hacer ver.

2- UTILIZANDO COMO APOYO EL BUCLE INTERIOR PERO LA TÉCNICA SE APLICA A LOS DOS BUCLES A LA VEZ

Primero hacemos un "strip-mine" sobre el bucle "i" y después cambiamos el orden de las iteraciones poniendo el segundo bucle generado en el interior de los bucles anidados. Con el pragma "#pragma simd" (peor porque quita la alineación) le decimos que se vectorial también el bucle de la j o "#pragma unroll(titleSize)". La transformación se llama "Register blocking" porque nos interesa que quepa todo e el banco de registros vectorial. El tileSize es de 8 porque queremos rellenar un registro entero con el bucle más interno.

El rendimiento es un poco mayor de lo obtenido anteriormente.

TAREA

Kernel muy típico, la función binParticles es con la que tenemos que trabajar. Algoritmo de clasificación, se reciéntenlo una serie de datos, se realiza una serie de cálculos y en función del resultado de los datos se clasifican en una serie de categorías y se cuentan cuantos elementos hay de cada categoría. En nuestro caso es la transformación de coordenadas polares a cartesianas. Se reciben una cantidad en formato cilíndrico (magnitud y ángulo) y aplicando trigonometría se convierten en coordenadas polares. En cada uno de los bits se toma la parte entera de las coordenadas y se apuntan cuantas partículas hay en cada coordenada entera.

// Paralelización

Sí se puede paralelizar pero no es suficiente con poner esto porque el resultado no es correcto, porque en función de los cálculos se decide que posición de la variable de salida hay que actualizar y se incrementa esa posición de la variable de salida, el problema es que la variable de salida solo hay una, si todos los hilos están modificando de forma concurrente esa variable de salida, ahí tenemos una sesión crítica y una carrera, hay que proteger el acceso a esa variable compartida.

Patrón de reducción que las hacía automáticamente el omp en cuanto se lo indicábamos, pero solo sabe hacerlo para variables escalares y esta es un array de dos dimensiones pero la misma transformación que vimos que se podía hacer pero ahora de forma manual. El rendimiento será bastante bueno.
#pragma amp parallel for
// Vectorización
Automáticamente y ver el informe. No se vectorial bien y habrá que hacer algo. 5 instrucciones vectoriales pero al final cuando incrementamos, el acceso a memoria es totalmente aleatorio
El bucle partirlo en dos trozos para que la primera parte si se pueda vectorizar y la segunda no. Crear una estructura de datos intermedia para guardar los resultados del primero trozo de código (dos arrays, uno de iX y otro de iY) y luego hacer la parte que no se puede vectorizar de forma secuencial.

// Acceso a memoria

El código resultante con el unroll-and-jam es más fácil de vectorizar
Análisis de sensibilidad (prueba y error)
Gráficas por lo del tamaño,
En un unroll-and-jam el valor tiene que ser pequeño y parecido en todos, en torno a 4 y 32 iteraciones
Y el otro uno mucho más grande

CUESTIONES

Instrucciones

En el archivo adjunto a esta tarea encontrarás un micro-kernel que resuelve el problema del *binning*:
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.

```
//NORMAL
void BinParticles(const InputDataType  & inputData, BinsType & outputBins) {
  for (int i = 0; i < inputData.numDataPoints; i++) {
    // Transforming from cylindrical to Cartesian coordinates:
    const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
    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);
```

```
    // Incrementing the appropriate bin in the counter
    ++outputBins[iX][iY];
  }
}
```

```
Baseline performance:   9.10e-02 GP/s

Benchmarking...

  Trial    Time, s    Speedup       GP/s *
      1  7.474e-01       0.99   8.98e-02 **
      2  7.467e-01       0.99   8.99e-02 **
      3  7.438e-01       0.99   9.02e-02
      4  7.468e-01       0.99   8.99e-02
      5  7.489e-01       0.98   8.96e-02
      6  7.481e-01       0.99   8.97e-02
      7  7.446e-01       0.99   9.01e-02
      8  7.461e-01       0.99   8.99e-02
      9  7.467e-01       0.99   8.99e-02
     10  7.417e-01       0.99   9.05e-02
-----------------------------------------------------------
Optimized performance:   0.99   9.00e-02 +- 2.66e-04 GP/s
```

Se te pide que realices los tres ejercicios siguientes:

Ejercicio 1: Paralelización.

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

```
//PARALELIZACIÓN
#define DEFAULT_SIZE 32
void BinParticles(const InputDataType  & inputData, BinsType & outputBins) {
    static int *iX = (int*)_mm_malloc(inputData.numDataPoints*sizeof(int), DEFAULT_SIZE);
    static int *iY = (int*)_mm_malloc(inputData.numDataPoints*sizeof(int), DEFAULT_SIZE);

#pragma omp parallel for reduction(+:outputBins)
    for (int i = 0; i < inputData.numDataPoints; i++) {
        const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
        iX[i] = int((x - xMin)*binsPerUnitX);
        const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);
        iY[i] = int((y - yMin)*binsPerUnitY);
        ++outputBins[iX[i]][iY[i]];
    }
}
```

```
Baseline performance:   8.35e-02 GP/s

Benchmarking...

  Trial    Time, s    Speedup       GP/s *
      1  4.422e-01       1.82   1.52e-01 **
```

```
      2  3.260e-01       2.46   2.06e-01 **
      3  3.330e-01       2.41   2.02e-01
      4  3.435e-01       2.34   1.95e-01
      5  3.305e-01       2.43   2.03e-01
      6  3.269e-01       2.46   2.05e-01
      7  3.381e-01       2.38   1.99e-01
      8  3.268e-01       2.46   2.05e-01
      9  3.347e-01       2.40   2.01e-01
     10  3.498e-01       2.30   1.92e-01
-----------------------------------------------------
Optimized performance:   2.40   2.00e-01 +- 4.46e-03 GP/s
-----------------------------------------------------
```

La paralelización es la técnica que ha obtenido mayor rendimiento al utilizarse aisladamente, el compilador de gcc permite el parámetro outputBins en reduction que es una matriz, en el compilador de intel se hace manualmente.

Ejercicio 2: 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. Evalúa la acelaración que has obtenido en función del tamaño de la unidad vectorial y del número de unidades vectoriales que tengas, y comenta los resultados obtenidos.

```
// VECTORIZADO
#define DEFAULT_SIZE 32
void BinParticles(const InputDataType  & inputData, BinsType & outputBins) {
    static int *iX =
(int*)__builtin_assume_aligned((int*)_mm_malloc(inputData.numDataPoints*sizeof(int),
DEFAULT_SIZE), DEFAULT_SIZE);
    static int *iY =
(int*)__builtin_assume_aligned((int*)_mm_malloc(inputData.numDataPoints*sizeof(int),
DEFAULT_SIZE), DEFAULT_SIZE);
#pragma omp simd
    for (int i = 0; i < inputData.numDataPoints; i++) {
        const FTYPE x = inputData.r[i]*COS(inputData.phi[i]);
        iX[i] = int((x - xMin)*binsPerUnitX);
        const FTYPE y = inputData.r[i]*SIN(inputData.phi[i]);
        iY[i] = int((y - yMin)*binsPerUnitY);
        ++outputBins[iX[i]][iY[i]];
    }
}
```

```
Baseline performance:   7.85e-02 GP/s

Benchmarking...

  Trial    Time, s    Speedup       GP/s *
      1  1.089e+00       0.79   6.16e-02 **
      2  9.023e-01       0.95   7.44e-02 **
      3  8.709e-01       0.98   7.71e-02
      4  8.934e-01       0.96   7.51e-02
      5  8.742e-01       0.98   7.68e-02
      6  8.668e-01       0.99   7.74e-02
      7  8.566e-01       1.00   7.83e-02
      8  8.435e-01       1.01   7.96e-02
      9  8.340e-01       1.03   8.05e-02
     10  8.512e-01       1.00   7.88e-02
-----------------------------------------------------
```

```
Optimized performance:    0.99    7.79e-02 +- 1.60e-03 GP/s
```

La vectorización, que consiste en indicar que los datos están alineados en la caché no mejora el código. Incluso podría decir que lo ha empeorado. Los pragmas de gcc nos distintos a los de icc, por tanto, he tenido que buscar información de como se hacía por internet, en la bibliografía están las referencias.
El informe generado es ilegible en gcc, aunque tanto en icc como en gcc indican que no se el compilador no sabía vectorizarlo directamente.

Ejercicio 3: Optimización del acceso a memoria.

Optimiza el rendimiento de este micro-kernel optimizando el acceso a memoria que hace. Prueba a hacer uso de la técnica de *loop tiling* en sus dos variantes (*cache blocking* y *unroll-and-jam*) para mejorar el rendimiento. Evalúa la acelaración que has obtenido en función del tamaño de la memoria necesaria. Para ello, junto al caso base de usar para *n* un valor de 2^27, prueba también con 2^30 y 2^33. Asimismo, modifica el tamaño base de los contenedores (nBinsX = nBinsY = 10) a la mitad. Comenta los resultados obtenidos.

```
//COMPLETO

#define DEFAULT_SIZE 32

void BinParticles(const InputDataType  & inputData, BinsType & outputBins) {

    static int *iX =
(int*)__builtin_assume_aligned((int*)_mm_malloc(inputData.numDataPoints*sizeof(int),
DEFAULT_SIZE), DEFAULT_SIZE);

    static int *iY =
(int*)__builtin_assume_aligned((int*)_mm_malloc(inputData.numDataPoints*sizeof(int),
DEFAULT_SIZE), DEFAULT_SIZE);



#pragma omp parallel for reduction(+:outputBins)

    for (int ii = 0; ii < inputData.numDataPoints; ii+=DEFAULT_SIZE) {

#pragma omp simd

        for(int i= 0; i< DEFAULT_SIZE; i++)

        {

            if(inputData.numDataPoints > i+ii)

            {

                const FTYPE x = inputData.r[i+ii]*COS(inputData.phi[i+ii]);

                iX[i+ii] = int((x - xMin)*binsPerUnitX);
```

```
                const FTYPE y = inputData.r[i+ii]*SIN(inputData.phi[i+ii]);

                iY[i+ii] = int((y - yMin)*binsPerUnitY);

            }

        }

        for(int i= 0; i < DEFAULT_SIZE; i++){

            if(inputData.numDataPoints > i+ii)

                ++outputBins[iX[i+ii]][iY[i+ii]];

        }

    }

}
```

```
Baseline performance:   8.36e-02 GP/s

Benchmarking...

  Trial    Time, s    Speedup       GP/s *

      1  4.568e-01       1.76   1.47e-01 **

      2  3.462e-01       2.32   1.94e-01 **

      3  3.472e-01       2.31   1.93e-01

      4  3.637e-01       2.21   1.85e-01

      5  3.533e-01       2.27   1.90e-01

      6  3.496e-01       2.30   1.92e-01

      7  3.652e-01       2.20   1.84e-01

      8  3.472e-01       2.31   1.93e-01

      9  3.507e-01       2.29   1.91e-01

     10  3.610e-01       2.22   1.86e-01

-----------------------------------------------------------

Optimized performance:   2.26   1.89e-01 +- 3.69e-03 GP/s

-----------------------------------------------------------
```

El rendimiento obtenido de la fusión de los tres métodos de optimización no es el esperado.

COMENTARIOS SOBRE LA PRÁCTICA

No me ha gustado esta práctica porque tengo la sensación de que he aprendido bastante pero no lo suficiente como para saber resolver esta tarea correctamente. Los videos de las clases han servido pero son muy generales como para saber resolver este problema de forma óptima y poder plantear cualquier problema, no los mostrados en clase que son muy concretos. En ellos solo se planteaban algunos casos específicos y al intentar abordar otros diferentes como los de esta tarea me ha costado bastante. En la vectorización ha sido imposible comprender el informe porque se genera en gcc distinto a icc. Además, en clases de prácticas hemos visto como interpretar la información del fichero pero no como mejorar manualmente un bucle para su vectorización, entonces no se si es porque realmente no se puede o porque no se ha planteado durante las sesiones. En la parte de comprobación ha sido la más difícil porque probando en tres máquinas diferentes me salían distintos resultados y los pragmas de icc vistos en clase no son compatibles en gcc. Las diapositivas de ayuda en inglés solo me han servido para liarme más con algunos conceptos y he echado en falta un ejemplo en clase que fusionara todos los mecanismos diferentes que se pueden integrar para paralelizar. Me gustaría que dejaran la solución correcta en el aula virtual porque no sé si algunos razonamientos o implementaciones son correctos o si puede haber una mejor versión. El servidor estaba saturado y he tenido que usar mi ordenador para todo.

BIBLIOGRAFÍA

https://w3.ual.es/~jjfdez/IC/Practicas/optim.html

http://oa.upm.es/40000/1/TFG_BUSTOS_MARTIN_OSCAR.pdf

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0886r0.pdf

https://stackoverflow.com/questions/35416001/typed-variant-of-builtin-assume-aligned-in-gnu-gcc