ARQUITECTURA Y ORGANIZACIÓN DE COMPUTADORES

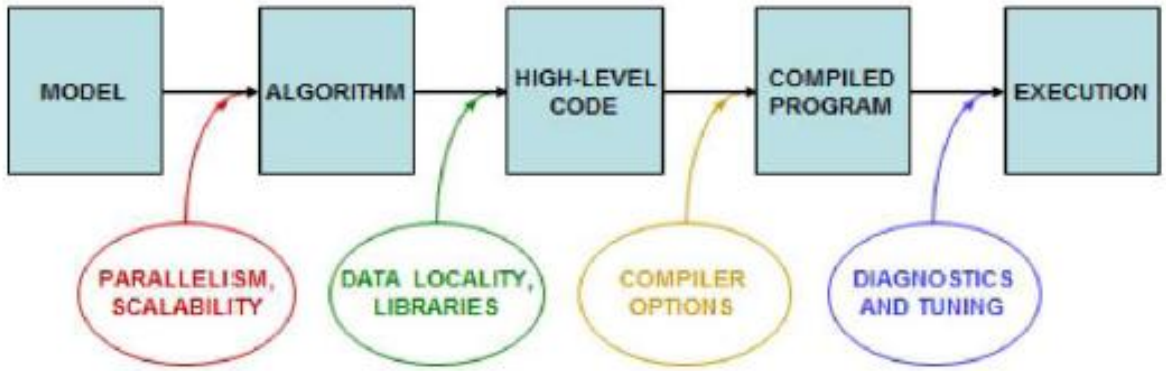
3er curso

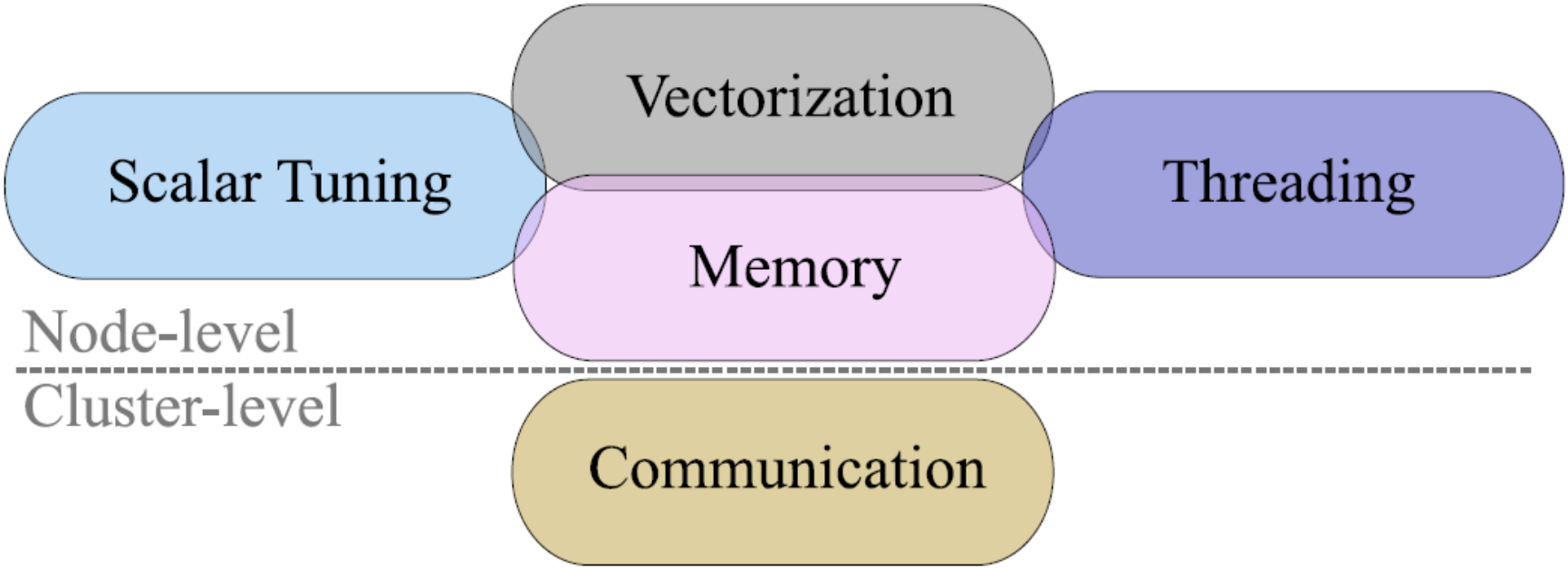
Práctica 3. Optimización del rendimiento de un CMP

Departamento de Ingeniería y Tecnología de Computadores

Universidad de Murcia

Presentación





Objetivos

- Comprender cómo mejorar el código aplicando la optimización escalar (a veces llamada "reducción de fuerza" de operaciones complejas)
- Explotar convenientemente la paralelización de una aplicación
- Mejorar la vectorización alineando los datos y cambiando el tipo de estructuras de datos que se utilizan en el programa
- Mejorar el rendimiento de la aplicación optimizando su acceso a memoria por medio de reutilizar 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

IMPORTANTE: Seguiremos usando la imagen de Docker de la práctica anterior

Caso de estudio: Simulación de N cuerpos (I)

Aplicaciones

1. Astrofísica:
   - Sistemas planetarios
   - Galaxias
   - Estructuras cosmológicas
2. Sistemas electrostáticos:
   - Moléculas
   - Cristales

El código realiza un seguimiento de la posición y velocidad de cada partícula. La simulación es discretizada a lo largo del tiempo en una cantidad prefijada de pasos de tiempo (*timesteps*).

Este código es un "modelo de juguete" con un algoritmo *O*(*n*^2) con interacciones *todos-para-todos*.

Simulaciones prácticas de N-cuerpos suelen usar algoritmos de árbol con complejidad *O*(*n*log*n*) .

Source: APOD, credit: DebraMeloy Elmegreen (Vassar College) et al., & the Hubble Heritage Team (AURA/ STScI/ NASA)

Caso de estudio: Simulación de N cuerpos (II)

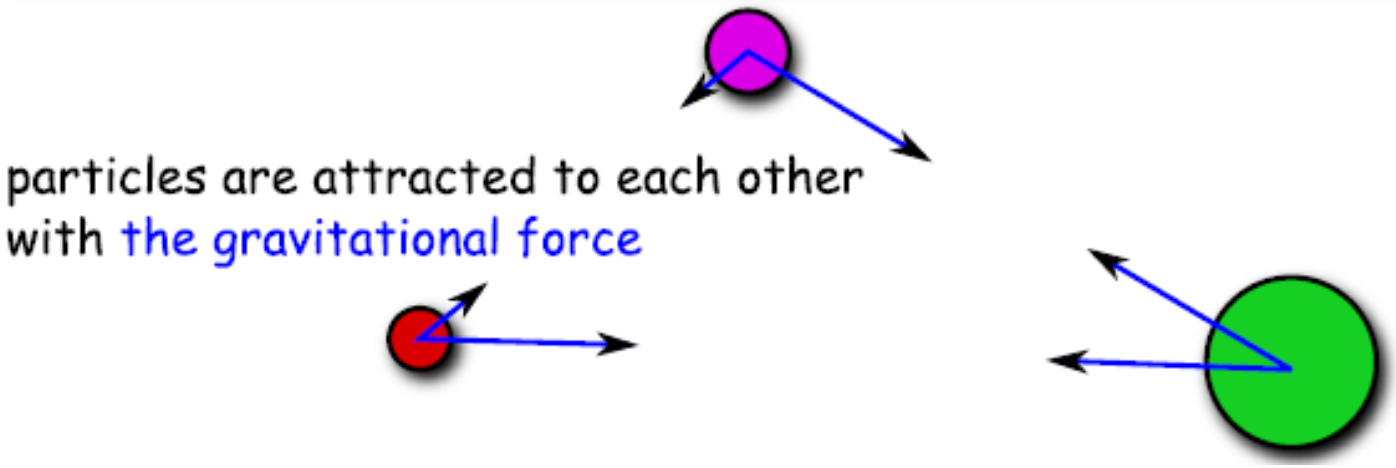
Dinámica gravitacional sobre N cuerpos

Newton's law of universal gravitation:

$$M_i \vec{R}_i''(t) = G \sum_j \frac{M_i M_j}{\left|\vec{R}_i - \vec{R}_j\right|^3} \left(\vec{R}_j - \vec{R}_i\right)$$

where:

$$\left|\vec{R}_i - \vec{R}_j\right| = \sqrt{(R_{i,\,x} - R_{j,\,x})^2 + (R_{i,\,y} - R_{j,\,y})^2 + (R_{i,\,z} - R_{j,\,z})^2}$$



Caso de estudio: Simulación de N cuerpos (III)

Cada partícula es almacena como una estructura:

```
1 struct ParticleType {
2   float x, y, z;
3   float vx, vy, vz;
4 };
```

`main()` crea un array de una estructura denominada `ParticleType`:

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

El tiempo tardado por el movimiento de las partículas es medido con OMP:

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

Caso de estudio: Simulación de N cuerpos (IV)

Código de partida de `MoveParticles`:

```
1  void MoveParticles(int nParticles, ParticleType* particle, float dt) {
2    for (int i = 0; i < nParticles; i++) {   // Particles that experience force
3      float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
4      for (int j = 0; j < nParticles; j++) { // Particles that exert force
5        // Newton's law of universal gravity
6        const float dx = particle[j].x - particle[i].x;
7        const float dy = particle[j].y - particle[i].y;
8        const float dz = particle[j].z - particle[i].z;
9        const float drSquared  = dx*dx + dy*dy + dz*dz + 1e-20;
10       const float drPower32  = pow(drSquared, 3.0/2.0);
11       // Calculate the net force
12       Fx += dx/drPower32;  Fy += dy/drPower32;  Fz += dz/drPower32;
13     }
14     // Accelerate particles in response to the gravitational force
15     particle[i].vx+=dt*Fx; particle[i].vy+=dt*Fy;  particle[i].vz+=dt*Fz;
16   }
17   ...
```

Uso de la herramienta perf

perf es una herramienta que permite analizar el rendimiento de una aplicación según diferentes *eventos.*

Para esta práctica, vamos a usar los siguientes *eventos*:

- *cache-references*: Número de accesos a a la caché de último nivel. Esto puede incluir accesos por *prefetching* y mensajes de coherencia.
- *cache-misses*: Número de accesos a memoria que no pudieron ser atendidos por ninguna de las caches. Junto al evento anterior, es posible calcular el *cache miss rate* de la CPU.
- *cycles*: Ciclos de reloj que ha tardado el programa.
- Instructions: Número de instrucciones que se han retirado (que han hecho *commit*) del ROB.
- *L1-dcache-loads*, *L1-dcache-load-misses*, *LLC-loads*, *LLC-load-misses*: Número de accesos y fallos de cache a nivel de L1 de datos y LLC.

$ perf stat -d -e cache-references,cache-misses,cycles,instructions *exe_name*

Optimización 1: Conversiones de tipos

- Si analizamos el informe del compilador *icc* (que lo tienes en en el directorio *solutions/01-messages*) veremos varios mensajes dentro del bucle principal. Uno de ellos es **type converts: 3**
- Esta conversión de tipos impacta negativamente en el rendimiento ya que se insertan instrucciones para cambiar entre los tipos de datos **float** y **double**
- Analizando el código vemos que esta conversión se hace en la llamada a la función **pow**
- En el directorio *solutions/02-conversions* tenemos la solución a este problema
- ¿Qué mejora obtenemos con esta modificación?

Reducción de fuerza (ejemplos)

Common Subexpression Elimination.

```
1 for (int i = 0; i < n; i++)  {
2   A[i] /= B;
3 }
```

```
1 const float Br = 1.0f/B;
2 for (int i = 0; i < n; i++)
3   A[i] *= Br;
```

Replace division with multiplication.

```
1 for (int i = 0; i < n; i++)  {
2   P[i] = (Q[i]/R[i])/S[i];
3 }
```

```
1 for (int i = 0; i < n; i++) {
2   P[i] = Q[i]/(R[i]*S[i]);
3 }
```

Use functions with Hardware support.

```
1 double r = pow(r2, -0.5);
2 double v = exp(x);
3 double y = y0*exp(log(x/x0)*
4            log(y1/y0)/log(x1/x0));
```

```
1 double r = 1.0/sqrt(r2);
2 double v = exp2(x*1.44269504089);
3 double y = y0*exp2(log2(x/x0)*
4            log2(y1/y0)/log2(x1/x0));
```

Optimización 2: Reducción de fuerza

Los cálculos que se hacen en el bucle interno pueden representarse con la siguiente expresión:

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

Esta expresión podemos simplificarla y obtener la siguiente expresión equivalente:

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

En esta nueva expresión desaparece la división y la potencia, ambas operaciones costosas. Aunque ambas sigan presentes en la fórmula, pueden cambiarse por operaciones equivalentes más eficientes.

Optimización 3: Paralelismo de datos

Before:

```
1  for (int i = 0; i < nParticles; i++) {   // Particles that experience force
2    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
3    for (int j = 0; j < nParticles; j++) { // Particles that exert force
4      // Newton's law of universal gravity
5      ...
```

After:

```
1  #pragma omp parallel for
2  for (int i = 0; i < nParticles; i++) {   // Particles that experience force
3    float Fx = 0, Fy = 0, Fz = 0; // Gravity force on particle i
4    for (int j = 0; j < nParticles; j++) { // Particles that exert force
5      // Newton's law of universal gravity
6      ...
```

¿Cómo se refleja esto en los resultados obtenidos por **perf**?
¿Qué mejora obtenemos?

Optimización 4: Vectorización (I)

El acceso a memoria de paso unitario (unit-stride) es óptimo:

```
1 for (int i = 0; i < n; i++)
2 A[i] += B[i];
```

El acceso con paso fuera de la unidad es más lento:

```
1 for (int i = 0; i < n; i++)
2 A[i*stride] += B[i];
```

El acceso estocástico puede ser vectorizado (pero no ser eficiente)

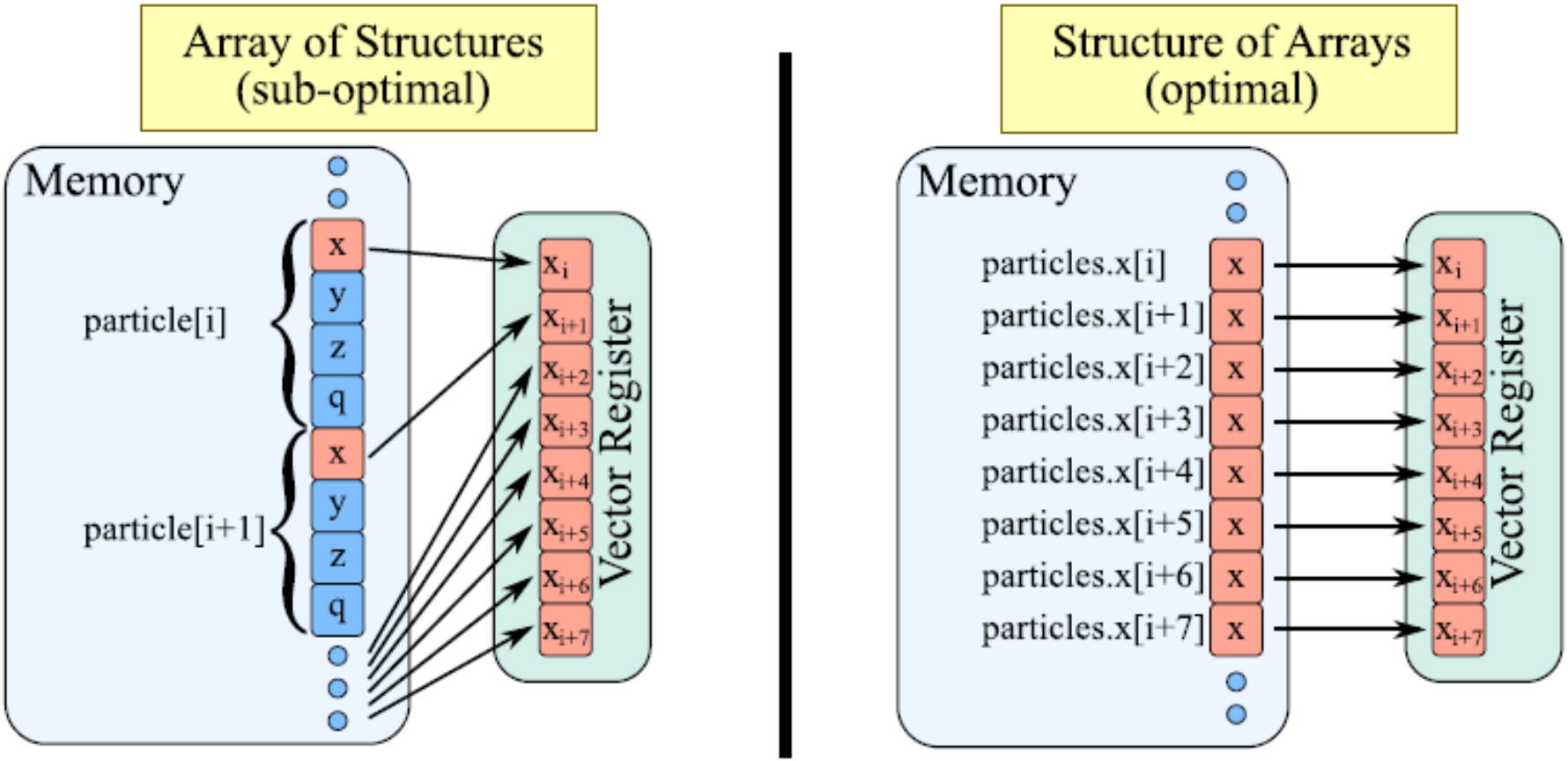
```
1 for (int i = 0; i < n; i++)
2 A[offset[i]] += B[i];
```

Puede ser cuestión de cambiar el orden de anidación de bucles, pero a veces es necesario modificar estructuras de datos

Optimización 4: Vectorización (II)

Necesitamos un acceso a memoria de paso unidad:

Cambio de AoS a SoA



Optimización 4: Vectorización (III)

Before:

```
1 struct ParticleType {
2   float x, y, z, vx, vy, vz;
3 };  // ...
4       const float dx = particle[j].x - particle[i].x;
5       const float dy = particle[j].y - particle[i].y;
6       const float dz = particle[j].z - particle[i].z;
```

After:

```
1 struct ParticleSet {
2   float *x, *y, *z, *vx, *vy, *vz;
3 }; // ...
4       const float dx = particle.x[j] - particle.x[i];
5       const float dy = particle.y[j] - particle.y[i];
6       const float dz = particle.z[j] - particle.z[i];
```

¡Muy importante también tener en cuenta el **alineamiento**!

¿Cómo se refleja esto en los resultados obtenidos por **perf**?
¿Qué mejora obtenemos?

Optimización 5: El acceso a memoria (I)

Existen 3 técnicas que se suelen usar para optimizar el acceso a memoria:

- El acceso a posiciones de memoria con paso unidad (*Unit-stride access*)
- El alineamiento de memoria (a veces usando *padding*).
- Mejorar el rendimiento de la aplicación optimizando su acceso a memoria por medio de reutilizar el acceso a los datos (técnica del *tiling*)

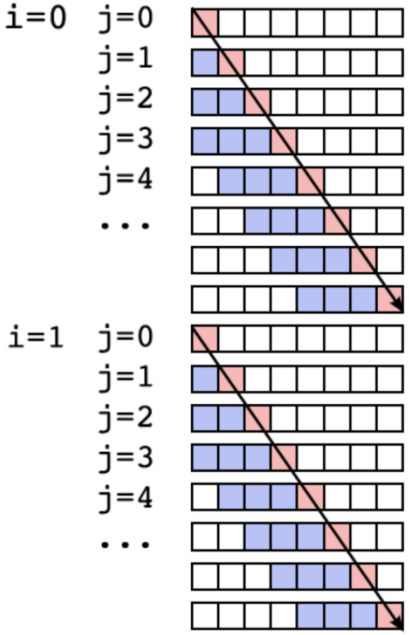
Nos centraremos en la última (**loop tiling**), ya que las otras dos ya las hemos aplicado.

Optimización 5: El acceso a memoria (II)

Necesitamos un acceso a memoria que explote la localidad de datos:
Loop Tiling (Cache blocking)

Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```



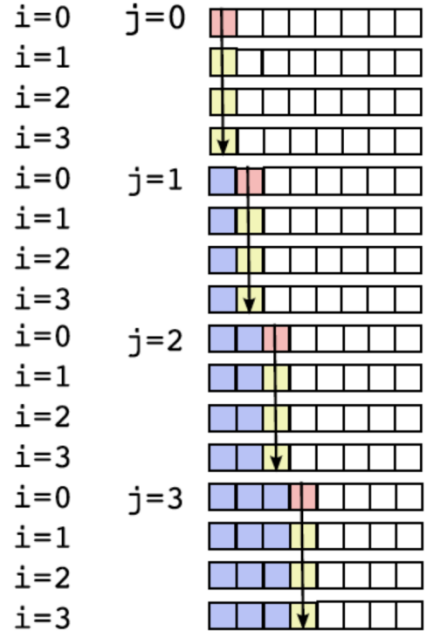
- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

Cache size: 4
TILE=4
(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

Tiled:

```
for (ii=0; ii<m; ii+=TILE)
  for (j=0; j<n; j++)
    for (i=ii; i<ii+TILE; i++)
      ...=...*b[j];
```



Optimización 5: El acceso a memoria (III)

```
1  for (int i = 0; i < m; i++)  // Original code:
2    for (int j = 0; j < n; j++)
3      compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
1  // Step 1: strip-mine inner loop
2  for (int i = 0; i < m; i++)
3    for (int jj = 0; jj < n; jj += TILE)
4      for (int j = jj; j < jj + TILE; j++)
5        compute(a[i], b[j]); // Same order of operation as original
```

```
1  // Step 2: permute
2  for (int jj = 0; jj < n; jj += TILE)
3    for (int i = 0; i < m; i++)
4      for (int j = jj; j < jj + TILE; j++)
5        compute(a[i], b[j]); // Re-use to j=jj sooner
```

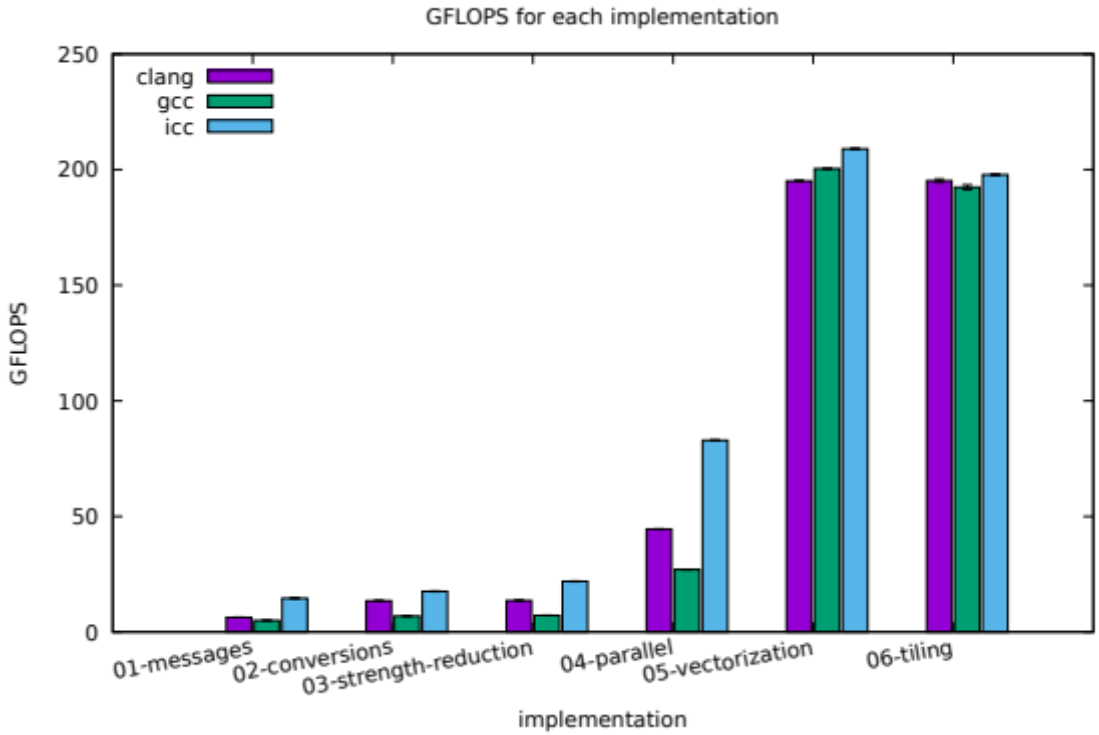
Normalmente

Cache blocking = Strip-mining + loop interchange + collapse

Optimización 5: El acceso a memoria (III)

- **Ojo:** Nos aparecen condiciones de carrera al usar *strip-mining*
- También reducimos significativamente el número de iteraciones del bucle más externo, el que estamos paralelizando. Por ello es importante usar el modificador **collapse(2)**
- ¿Qué tamaño usamos para *TILE*? Debe de ser siempre múltiplo de 64
- ¿Cómo se refleja esta modificación en los resultados?
- ¿Qué mejora obtenemos?

Evaluación del rendimiento

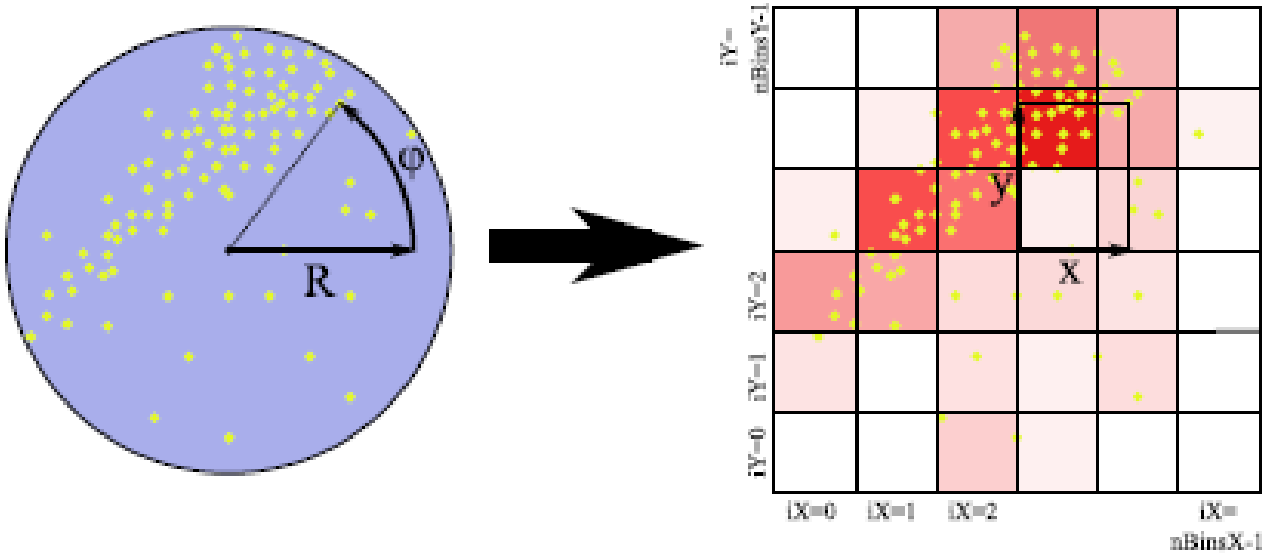


	Compilador		
Versión	GCC	Clang	ICC
01-messages	5.00±0.00	6.30±0.00	14.60±0.10
02-conversions	6.90±0.00	13.50±0.10	17.60±0.10
03-strength-reduction	7.20±0.00	13.60±0.10	21.90±0.10
04-parallel	27.00±0.00	44.50±0.10	82.90±0.10
05-vectorization	200.40±0.10	195.10±0.20	209.00±0.20
06-tiling	192.40±1.00	195.20±0.70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

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.

Ejercicios pedidos (I)

Algoritmo de *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.

Ejercicios pedidos (II)

El directorio *00-reference* contiene la versión inicial del programa, la cual no se debe modificar. Además del código, se adjuntan 2 scripts, que pueden ayudar a comprobar la corrección y medir el rendimiento, y para dibujar las gráficas de comparación de las soluciones. Sólo hay que usar el **icc** para esta práctica. Se pide:

- **Paralelización**: Utiliza OpenMP para paralelizar el programa. Evalúa la aceleración obtenida y comenta los resultados.
- **Vectorización**: Usa el compilador para vectorizar automáticamente el programa. Observa el informe del compilador para encontrar posibles problemas en la vectorización. Considera el uso de SoA. Evalúa la aceleración obtenida y comenta los resultados.
- **Optimización de acceso a memoria**: ¿Se podría hacer *loop tiling*? Comenta los resultados.
- ¿Afecta el tamaño del problema a la solución obtenida? ¿Qué pasaría si aumentásemos el número de *Bins* en los que clasificamos los datos de entrada?

Ejercicios pedidos (III)

Aspectos a tener en cuenta:

- Guarda todas las soluciones en el directorio **solutions**; dentro de él, crea un directorio para cada uno de las mejoras que desarrolles, usando un nombre adecuado para cada paso. Al final debe de haber al menos 3 directorios dentro de **solutions**, uno para cada uno de los ejercicios de optimización.
- Para justificar las respuestas en la explicación de los ejercicios, se debe usar la herramienta **perf** y/o el informe del compilador.
- Se pide añadir una **gráfica** donde se muestre el resultado obtenido por cada una de las mejoras respecto al programa inicial (como la gráfica de la diapositiva 20).
- Al igual que en la práctica 2, se recomienda el uso de un ordenador que tenga **4 o más cores.**

Entrega

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

- Un documento en formato PDF que será el documento principal utilizado a la hora de evaluar la práctica. Dicho fichero debe incluir:
  - Información sobre la autoría de la práctica.
  - Las respuestas a los ejercicios solicitados.
  - Un breve informe comentando los aspectos positivos de la práctica, así como cualquier aspecto negativo y cosas que has echado en falta en la misma. La longitud del informe no debe exceder las 2000 palabras
- El código fuente de cualquier programa o script desarrollado (o modificado) para la realización de la práctica. Estos ficheros se mirarán opcionalmente para comprobar cualquier aspecto que no esté claro en el documento PDF.
- Un fichero de texto llamado README identificando todos los ficheros fuente incluidos con instrucciones claras para su compilación.

Criterios de evaluación

- Esta práctica se evaluará teniendo en cuenta los siguientes criterios de evaluación (sobre 10 puntos):
  - Realización y corrección de los apartados pedidos (6 puntos)
  - Presentación y claridad de las explicaciones (1 punto)
  - Aportación de ideas originales en las explicaciones a los apartados realizados (1 punto)
  - Realización de alguna actividad extra relacionada con los ejercicios de la práctica demostrando curiosidad (1 punto)
  - Comentarios sobre la práctica, incluyendo aspectos negativos y positivos (1 punto)

ARQUITECTURA Y ORGANIZACIÓN DE COMPUTADORES

3er curso

Práctica 3. Optimización del rendimiento de un CMP

Departamento de Ingeniería y Tecnología de Computadores

Universidad de Murcia