ESTRUCTURA DE COMPUTADORES
PRÁCTICAS DE MEMORIAS

Técnicas de Optimización Software

Intercambio de bucles: Problema

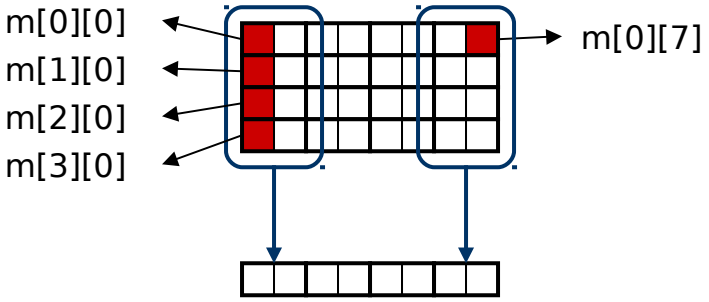
```
double m[4][8];

for(j=0; j < 8; j++)
  for(i=0; i < 4; i++)
    q += m[i][j];
```

C almacena por filas;
FORTRAN por columnas

	[0][0]	[0][1]	⋯	[0][7]	[1][0]	[1][1]	⋯	[1][7]	[2][0]	⋯

Caché de 64 bytes, líneas de 16 bytes, correspondencia directa



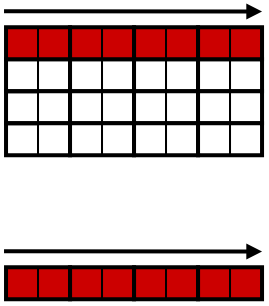
Intercambio de bucles: Solución

```
double m[4][8];

for(i=0; i < 4; i++)
  for(j=0; j < 8; j++)
    q += m[i][j];
```

	[0][0]	[0][1]	···	[0][7]	[1][0]	[1][1]	···	[1][7]	[2][0]	···

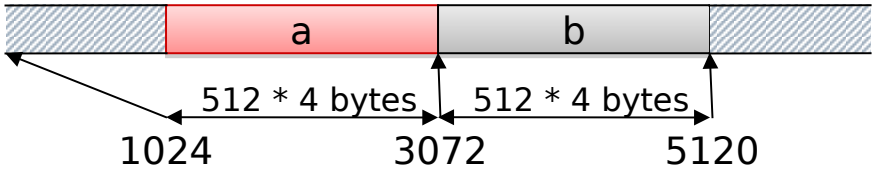
Caché de 64 bytes, líneas de 16 bytes, correspondencia directa



Fusión de arrays: Problema

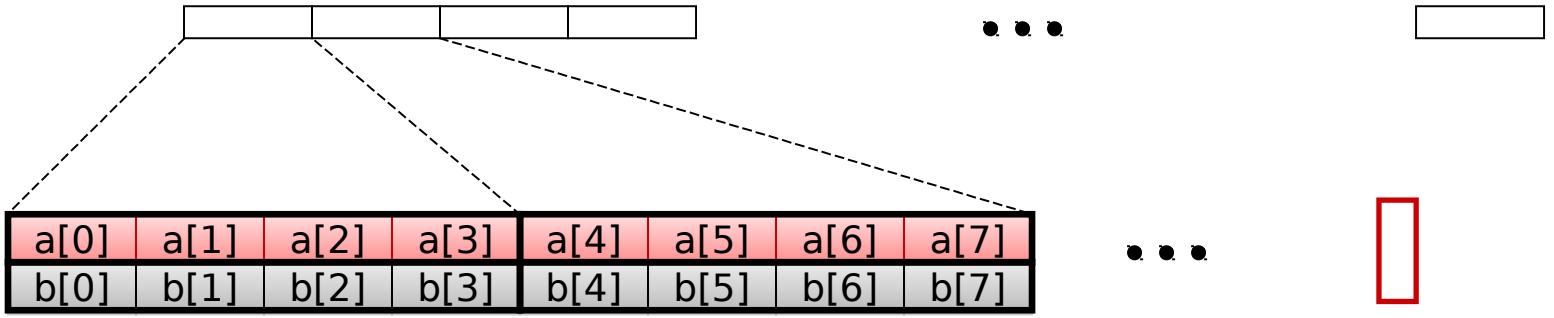
```
float a[512], b[512];

for(i=0; i<512; i++)
  a[i] += b[i];
```



Caché de 1024 bytes, líneas de 16 bytes, correspondencia directa

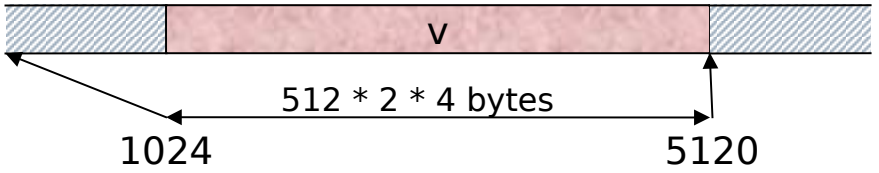
1024/16 = 64 líneas



Fusión de arrays: Solución

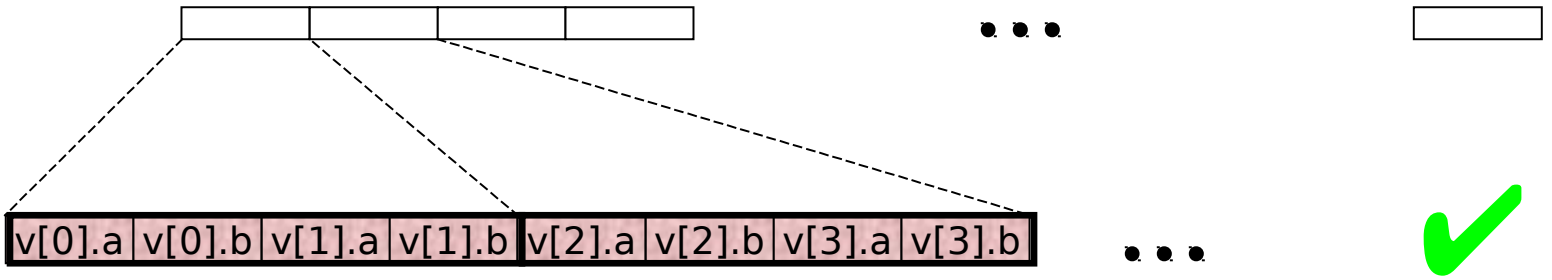
```
struct reg {
  float a, b;
};
struct reg v[512];

for(i=0; i<512; i++)
  v[i].a += v[i].b;
```



Caché de 1024 bytes, líneas de 16 bytes, correspondencia directa

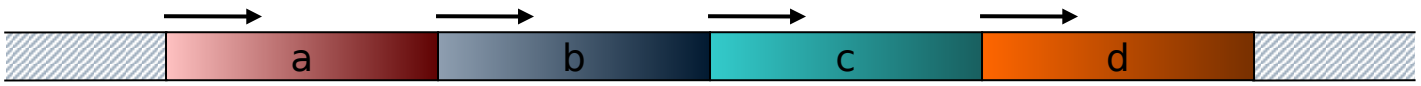
1024/16 = 64 líneas

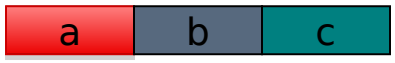


Fusión de bucles: Problema

```
for( i=0; i<N; i++)
  for( j=0; j<N; j++)
    a[i][j] = 1 / b[i][j] * c[i][j];

for( i=0; i<N; i++)
  for( j=0; j<N; j++)
    d[i][j] = a[i][j] + c[i][j];
```





La caché al finalizar el primer anidamiento de bucles tiene los últimos elementos accedidos de los arrays

Al empezar el segundo anidamiento se acceden los primeros elementos de los arrays, que ya no están

La caché al finalizar el primer anidamiento de bucles tiene los últimos elementos accedidos de los arrays

Fusión de bucles: Solución

```
for( i=0; i<N; i++)
  for( j=0; j<N; j++) {
    a[i][j] = 1 / b[i][j] * c[i][j];
    d[i][j] = a[i][j] + c[i][j];
}
```

- Los datos se tienen que cargar una única vez
- Además pueden reusarse en registros
  - En el ejemplo, a[i][j] y c[i][j]
- Ahorro en instrucciones de control de bucles

Rellenado de arrays: Problema

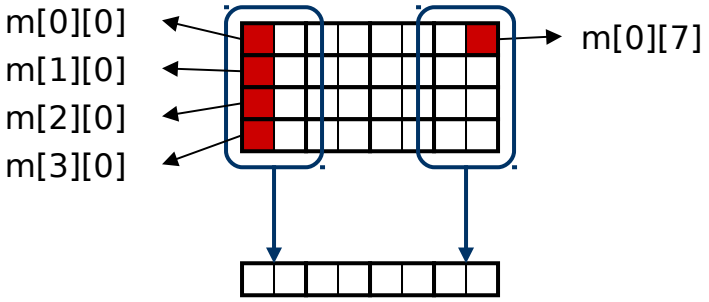
```
double m[4][8];

for(j=0; j < 8; j++)
  for(i=0; i < 4; i++)
    q = q * 1.2 + m[i][j];
```

C almacena por filas;
FORTRAN por columnas

	[0][0]	[0][1]	⋯	[0][7]	[1][0]	[1][1]	⋯	[1][7]	[2][0]	⋯

Caché de 64 bytes, líneas de 16 bytes, correspondencia directa

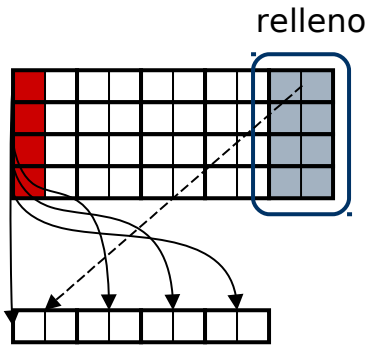


Rellenado de arrays: Solución

```
double m[4][10];

for(j=0; j < 8; j++)
  for(i=0; i < 4; i++)
    q = q * 1.2 + m[i][j];
```

Caché de 64 bytes, líneas de 16 bytes, correspondencia directa



Partición en bloques: Problema

```
float a[10], v[1000];
for( i=0; i<10; i++)
  for( j=0; j<1000; j++)
    a[i] += i * v[j];
```

Caché de 256 bytes, líneas de 16 bytes, asociativa de dos vías

i=0 ⇒ Se carga a[0] y v[0:999], total 4016 bytes, ~251 fallos

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	...	v[28]	v[29]	v[30]	v[31]
v[32]	v[33]	v[34]	v[35]	v[36]	v[37]	v[38]	v[39]		v[60]	v[61]	v[62]	v[63]

i=1 ⇒ Se carga a[1] y v[0:999], total 4016 bytes, ~251 fallos

...

En total hay aproximadamente ~2510 fallos

Partición en bloques:Solución

```
float a[10], v[1000];
for( jj=0; jj<1000; jj+=64)
  for( i=0; i<10; i++)
    for( j=jj; j<min(jj+64,1000); j++)
      a[i] += i * v[j];
```

Caché de 256 bytes, líneas de 16 bytes, asociativa de dos vías

jj=0 ⇒ i=0 ⇒ Se carga a[0] y v[0:63], ~1+16 fallos
i=1 ⇒ Se (re)carga a[1] y v[0:63], ~1+2 fallos
…
i=9 ⇒ Se (re)carga a[9] y v[0:63], ~1+2 fallos

} ~44 fallos

jj=1 ⇒ i=0 ⇒ Se carga a[0] y v[64:127], ~1+16 fallos
i=1 ⇒ Se (re)carga a[1] y v[64:127], ~1+2 fallos
…
i=9 ⇒ Se (re)carga a[9] y v[64:127], ~1+2 fallos

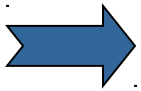
} ~44 fallos

…

En total hay aproximadamente ~700 fallos

Partición en bloques:Uso de CPU

```
float a[10], v[1000];
for( i=0; i<10; i++)
  for( j=0; j<1000; j++)
    a[i] += i * v[j];
```



```
float a[10], v[1000];
for( jj=0; jj<1000; jj+=64)
  for( i=0; i<10; i++)
    for( j=jj; j<min(jj+64,1000); j++)
      a[i] += i * v[j];
```

- 10 iteraciones del bucle i
- 10000 iteraciones del bucle j
- 11 predicciones erróneas de salto

- 16 iteraciones del bucle jj
- 160 iteraciones del bucle i
- 10000 iteraciones del bucle j
- 177 predicciones erróneas de salto
- Cálculos del límite del bucle j

El coste depende mucho del compilador y flags de optimización empleados

Partición en bloques: Ayudando a un mal compilador

```
float a[10], v[1000];
for( jj=0; jj<1000; jj+=64)
  for( i=0; i<10; i++)
    for( j=jj; j<min(jj+64,1000); j++)
      a[i] += i * v[j];
```

```
float a[10], v[1000];
for( jj=0; jj<1000; jj+=64) {
  const int limj = min(jj+64, 10000);
  for( i=0; i<10; i++)
    for( j=jj; j<limj; j++)
      a[i] += i * v[j];
}
```

Distribución de datos en estructuras: Problema

```
struct datos {
  float coef, b, c;
  char id[20];
  float posicion[3];
  double v;
};
struct datos ar[N];

for(i=0; i<M; i++) {
  r += ar[d[i]].coef * ar[d[i]].v;
}
```

- Los componentes de una struct se disponen en memoria en el orden en el que se los define
- Si campos que se usan conjuntamente están lejos, estarán probablemente en líneas diferentes

Distribución de datos en estructuras: Solución

```
struct datos {
  double v;
  float coef, b, c;
  char id[20];
  float posicion[3];
};
struct datos ar[N];

for(i=0; i<M; i++) {
  r += ar[d[i]].coef * ar[d[i]].v;
}
```

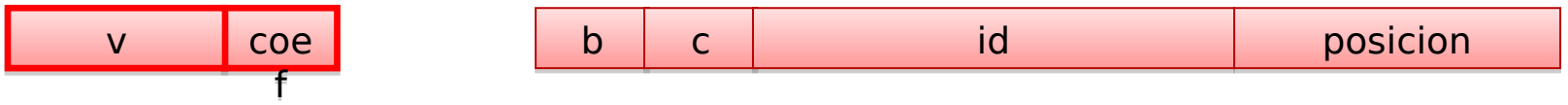
- Colocar cerca los campos que se usan conjuntamente

Distribución de datos en estructuras: otra Solución

```
struct datos1 {
  double v;
  float coef;
};
struct datos2 {
  float b, c;
  char id[20];
  float posicion[3];
};
struct datos1 ar[N];
struct datos2 ar2[N];

for(i=0; i<M; i++) {
  r += ar[d[i]].coef *
ar[d[i]].v;
}
```

- Subdividir la struct en varias
- Lo más sencillo:
  - Una struct con los campos más usados
  - Otra con el resto



Prebúsqueda

- Idea: requerir que los datos se traigan a caché antes del momento en el que son necesarios.
- Permite solapar su acercamiento al procesador con computaciones precedentes
- Dos tipos:
  - Hardware: El programador no puede controlarla, aunque sabiendo cómo funciona, puede escribir sus códigos para que intenten explotarla
  - Software: Automática por el compilador o controlada por el programador

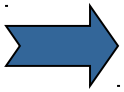
Prebúsqueda: limitaciones

- El exceso de prebúsquedas degrada el rendimiento
  - Supone ejecutar más instrucciones
    - Ej.: No hacer prebúsquedas consecutivas a la misma línea
  - Puede interferir con accesos más prioritarios para mantener alimentado el procesador
- Suelen despreciarse si fallan en la TLB
- Problema de temporización
  - Si se hace demasiado pronto el dato prebuscado podría ser expulsado antes de ser usado
  - Si se hace demasiado tarde el dato no llega a tiempo para su uso y hay que esperar
  - Se controla normalmente por la distancia de prebúsqueda, medida en iteraciones de bucle

Prebúsqueda: sintaxis

- No está estandarizado
  - Depende del lenguaje y del compilador
  - Puede ser una llamada a una función o una directiva al compilador
- Siempre se da la dirección que debe traerse a la caché
  - La dirección puede ser no válida
  - Pero la lectura de la dirección sí debe ser válida
- Ejemplo gcc:

```
for( i= 0; i < 2000; i++)
  t += x[i][i] + sqrt(y[i][i]);
```



```
for( i= 0; i < 2000; i++) {
 __builtin_prefetch(&x[i+8][i+8]);
 __builtin_prefetch(&y[i+8][i+8]);
 t += x[i][i] + sqrt(y[i][i]);
}
```