¿Optimizar?

Estos dos códigos... ¿hacen lo mismo?

```
#define ROWS 10000000
#define COLS 100

int main()
{
    char* matrix = new char[ROWS*COLS];

    for (int j = 0; j < COLS; ++j)
    {
        for (int i = 0; i < ROWS; ++i)
        {
            matrix[i*COLS + j] = 0;
        }
    }

    delete [] matrix;

    return 0;
}
```

```
g++ -o example1 example1.cpp
time ./example1
```

```
#define ROWS 10000000
#define COLS 100

int main()
{
    char* matrix = new char[ROWS*COLS];

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLS; ++j)
        {
            matrix[i*COLS + j] = 0;
        }
    }

    delete [] matrix;

    return 0;
}
```

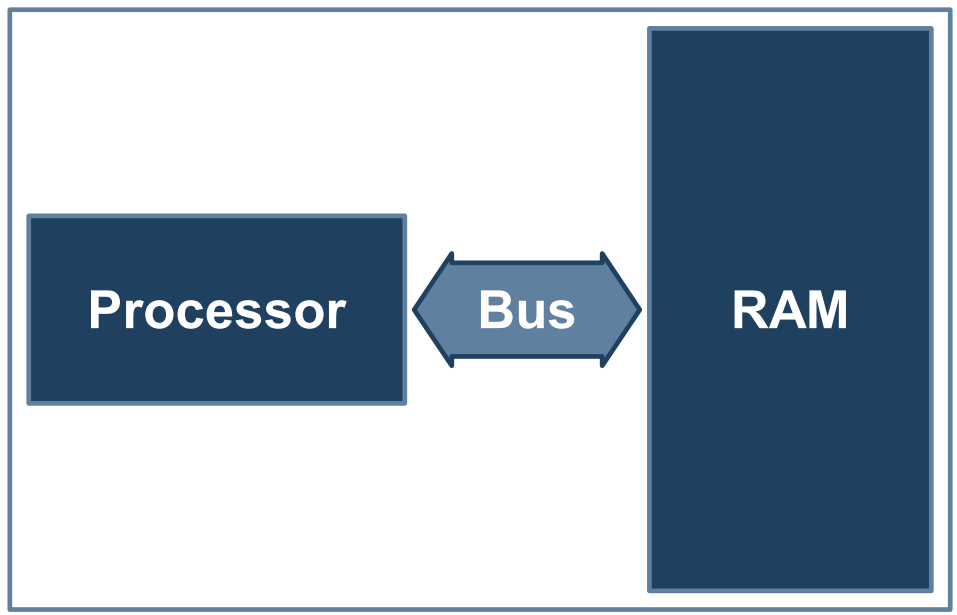
```
g++ -o example2 example2.cpp
time ./example2
```

Vamos a correrlos y ver...

Evolución del cómputo

Hace 30+ años

El siguiente es un modelo simplifacado de una computadora de principio de los 80's (Commodore 64, Tandy Radio-Shack TRS80, Apple II, etc).

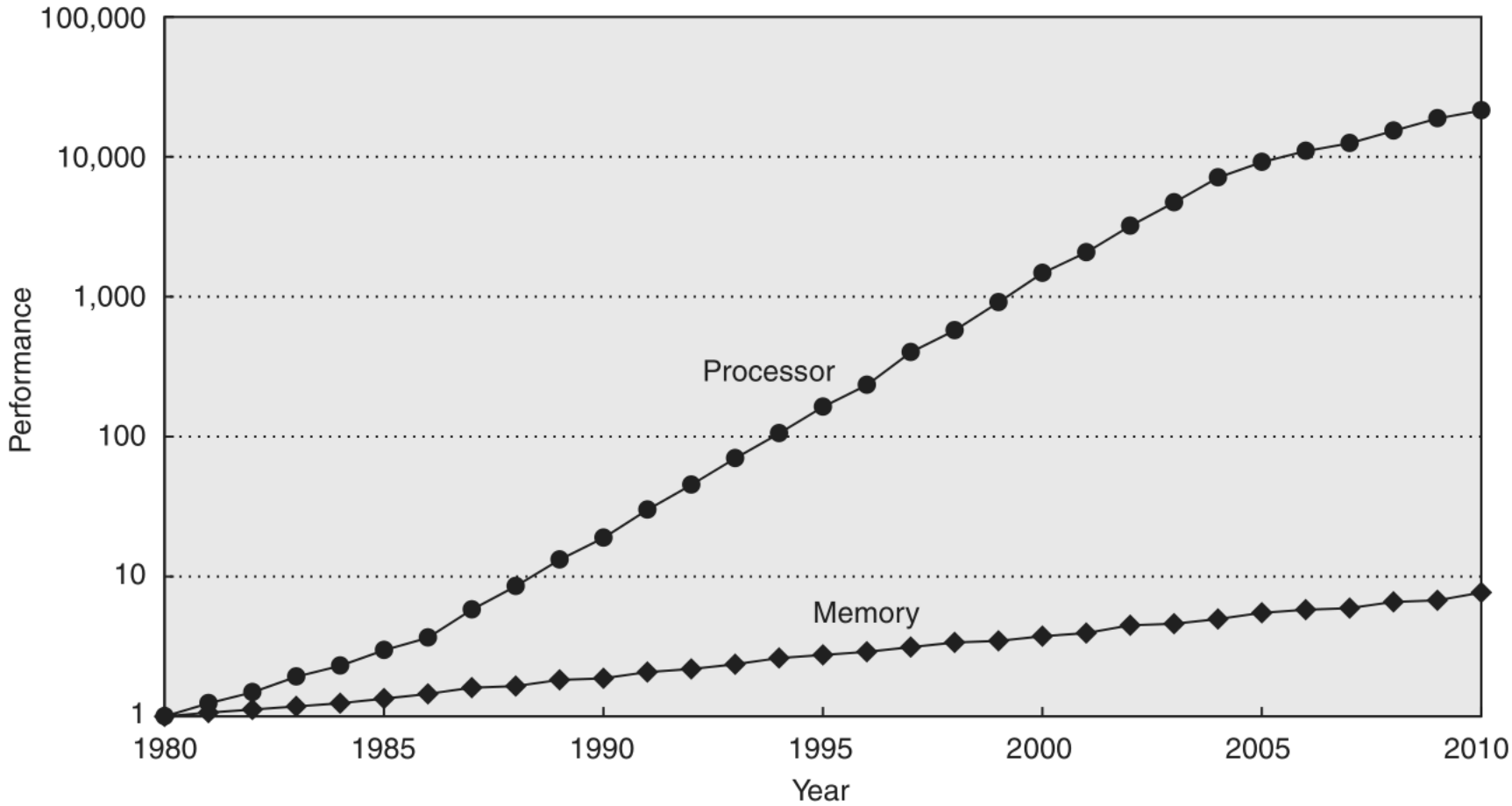


La computadora tiene un procesador que accesa directamente a la memoria RAM, a través de un bus de datos.

Una característica importante es que el procesador y la memoria trabajan a la misma velocidad.

Hace 20+ años

La velocidad del procesador y la memoria crecieron a diferente velocidad, los procesadores son más rápidos, mientas que la memoria RAM tiene que ser lenta para ser barata.

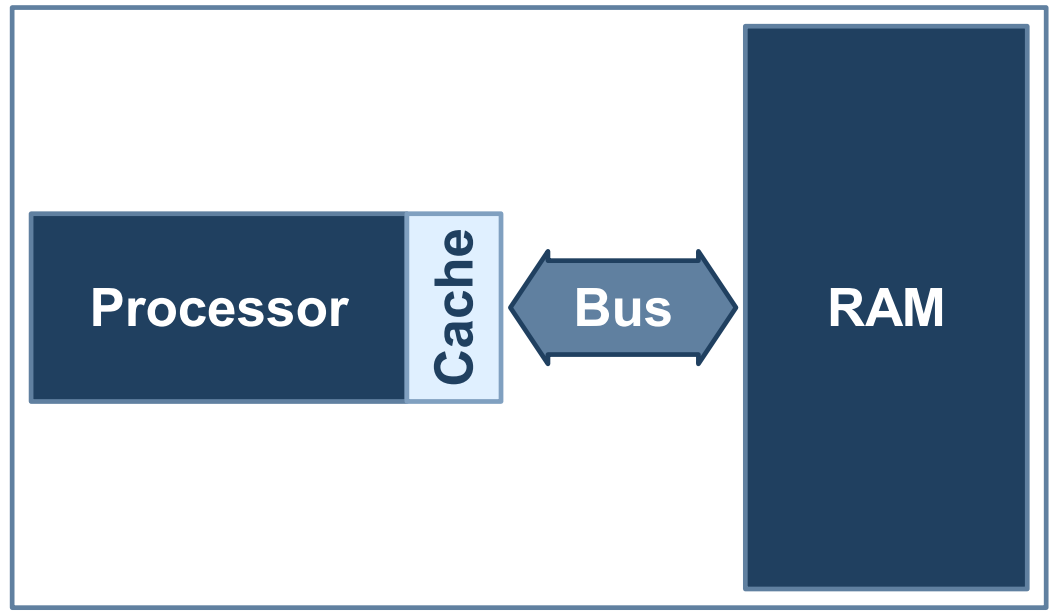


Starting with 1980 performance as a baseline, the gap in performance between memory and processors is plotted over time [Henn07].

El procesador tiene que esperar a la memoria, dado que esta no puede leerse y escribirse con suficiente velocidad [Wulf95].

El cache

Para evitar este problema, una **memoria rápida** es puesta entre el procesador y la RAM, llamada memoria *cache* [Mcke15].



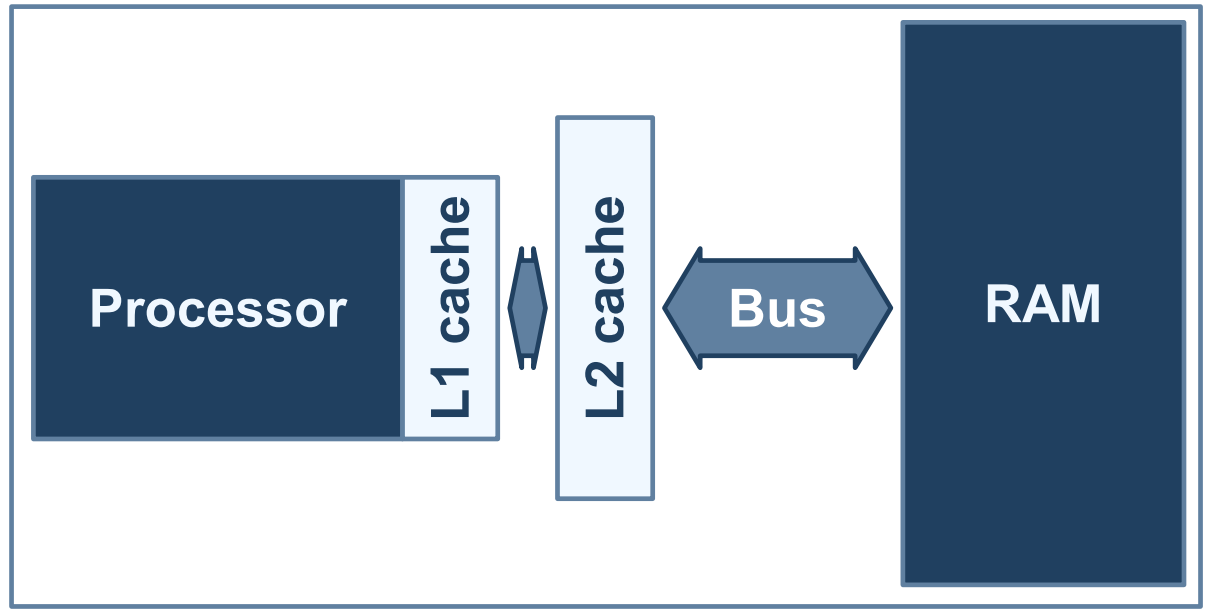
El cache es usado para mantener y procesar datos temporales.

Esta es una memoria de poca capacidad (aproximadamente 1/1000 del tamaño de la RAM).

Su uso suele ser transparente para el programador.

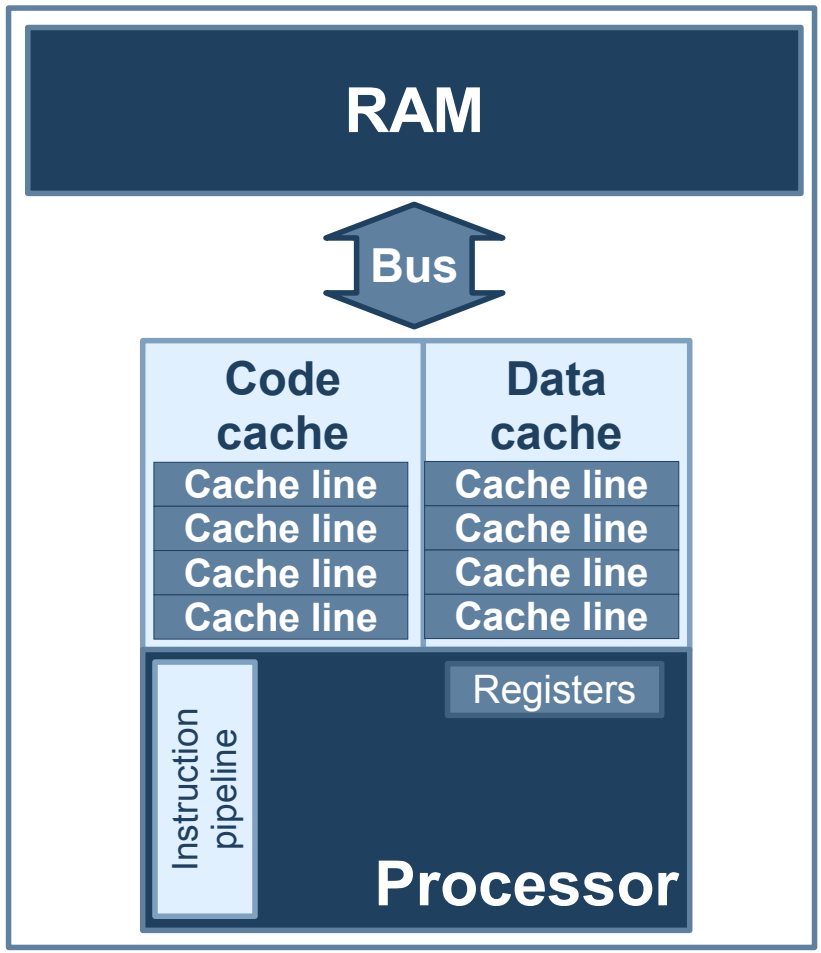
Todos los sistemas modernos usan memoria cache, incluso los procesadores gráficos (GPU) usan memoria cache.

Diseños posteriores de procesadores incluyen varios niveles de cache. El cache que está más cerca del procesador es llamado L1, otro cache L2 de mayor tamaño puede ser externo al procesador (por ejemplo el Intel 80486).



El tamaño del cache L2 es aproximadamente 1/100 del tamaño de la RAM. Modelos de procesadores más recientes pueden utilizar un nivel de cache L3.

La mayoría de los procesadores dividen el cache en dos partes, una para el código y otro para los datos.

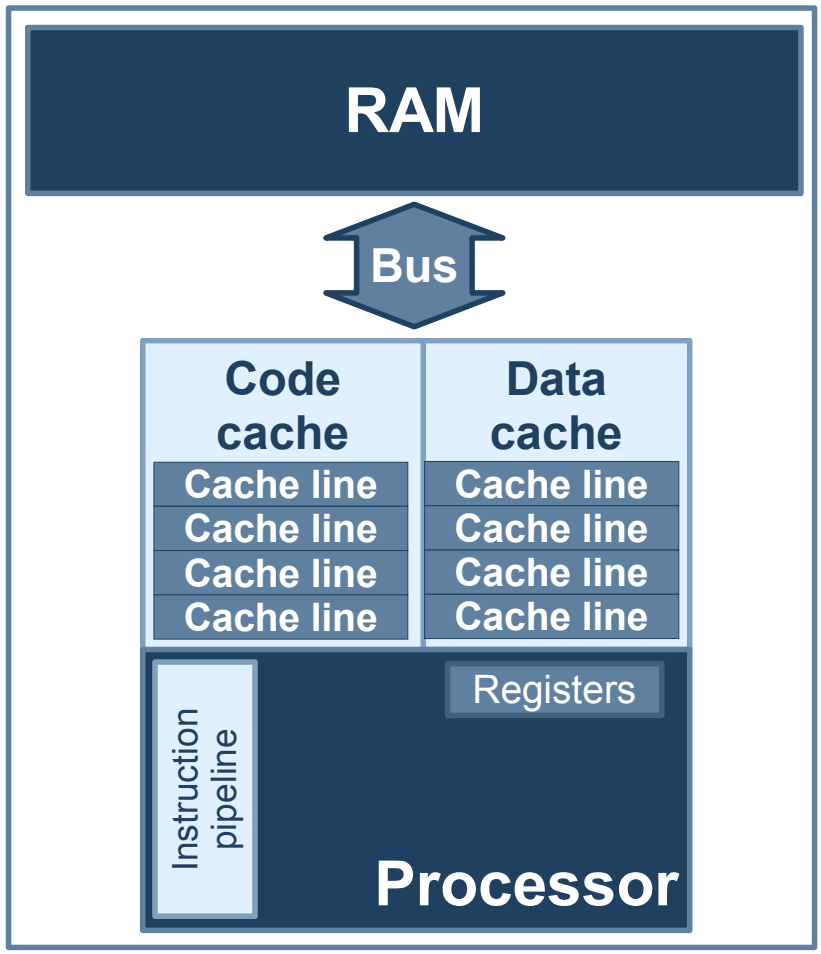



```
a = x1;
b = x2;
c = a + b;

if (c > 10)
{
     c = 10;
}

return;
```

```
[x1]
[x2]
```

```
[a]
[b]
[c]
```

La mayoría de los procesadores dividen el cache en dos partes, una para el código y otro para los datos.




```
mov ax, ptr [x1]
mov ptr [a], ax
mov ax, ptr [x2]
mov ptr [b], ax
mov cx, ptr [b]
mov ax, ptr [a]
add ax, cx
mov ptr [c], ax
cmp ptr [c], 10
jle +8Dh
mov ptr [c], 10
ret
```

```
[x1]
[x2]
```

```
[a]
[b]
[c]
```

¿Cuánto cache tiene mi computadora?

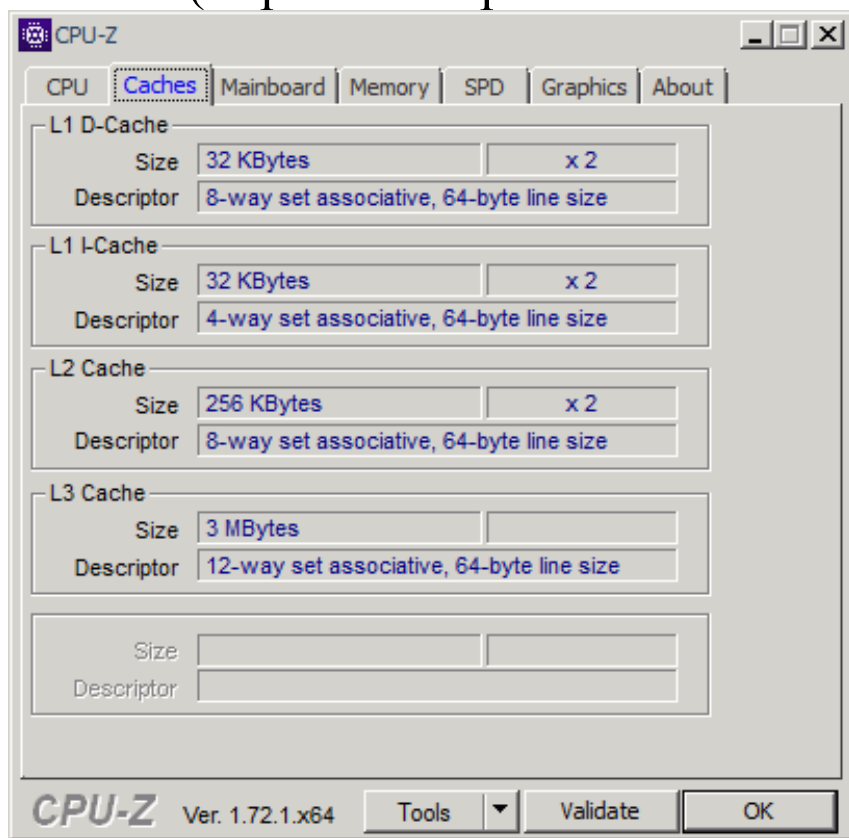
En Linux con el paquete hwloc

```
lstopo
```

o con el paquete dmidecode

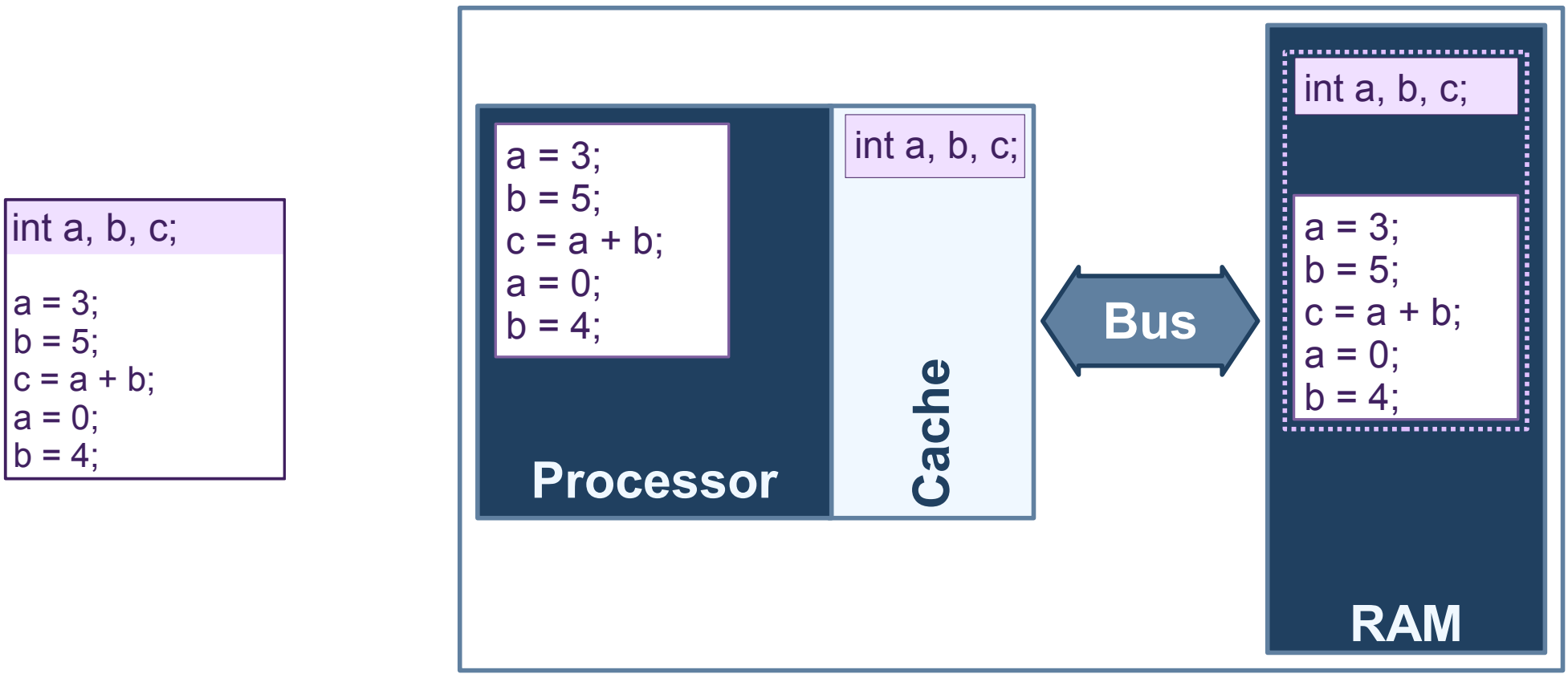
```
sudo dmidecode -t cache
```

En Windows con el programa CPU-Z (http://www.cpuid.com/softwares/cpu-z.html)



El cache

El cache opera haciendo copias de datos guardados en la RAM y permite que el procesador haga operaciones en ellos como si estuvieran en la RAM.



Después, todos los datos que fueron modificados son escritos de regreso a la RAM.

¿Qué tan rápido es el intercambio de datos entre el procesador, la RAM y los caches?

La siguiente tabla [Greg13] muestra el tiempo requerido para accesar un dato dependiendo de que tan cerca está del procesador.

Access	Time
1 CPU cycle	0.3 ns
Level 1 cache access	0.9 ns
Level 2 cache access	2.8 ns
Level 3 cache access	12.9 ns
Main memory access	120 ns
Solid-state disk I/O	50-150 µs
Rotational disk I/O	1-10 ms
Internet: SF to NYC	40 ms
Internet: SF to UK	81 ms
Internet: SF to Australia	183 ms

¿Qué tan rápido es el intercambio de datos entre el procesador, la RAM y los caches?

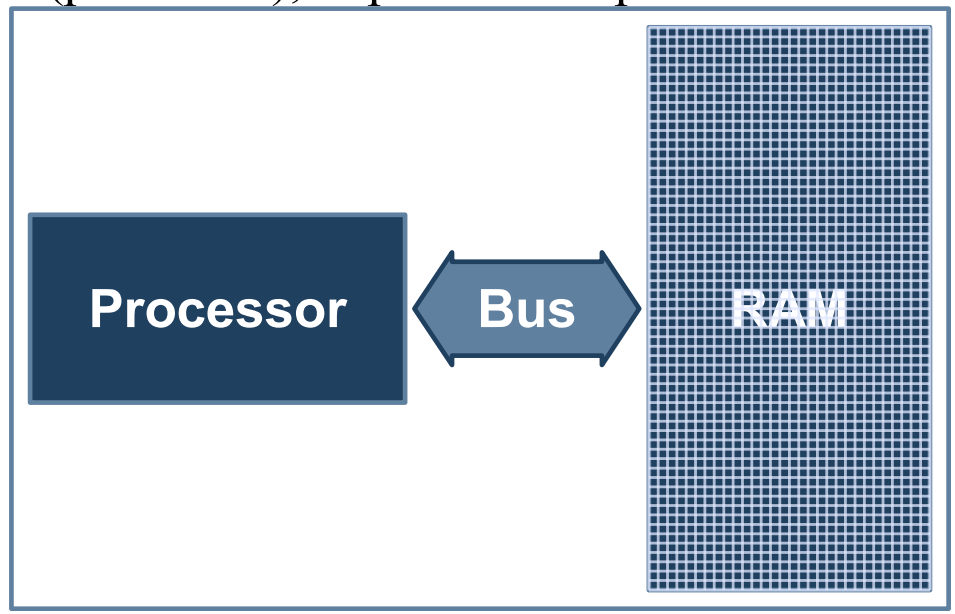
La siguiente tabla [Greg13] muestra el tiempo requerido para accesar un dato dependiendo de que tan cerca está del procesador.

Access	Time	One second scale
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

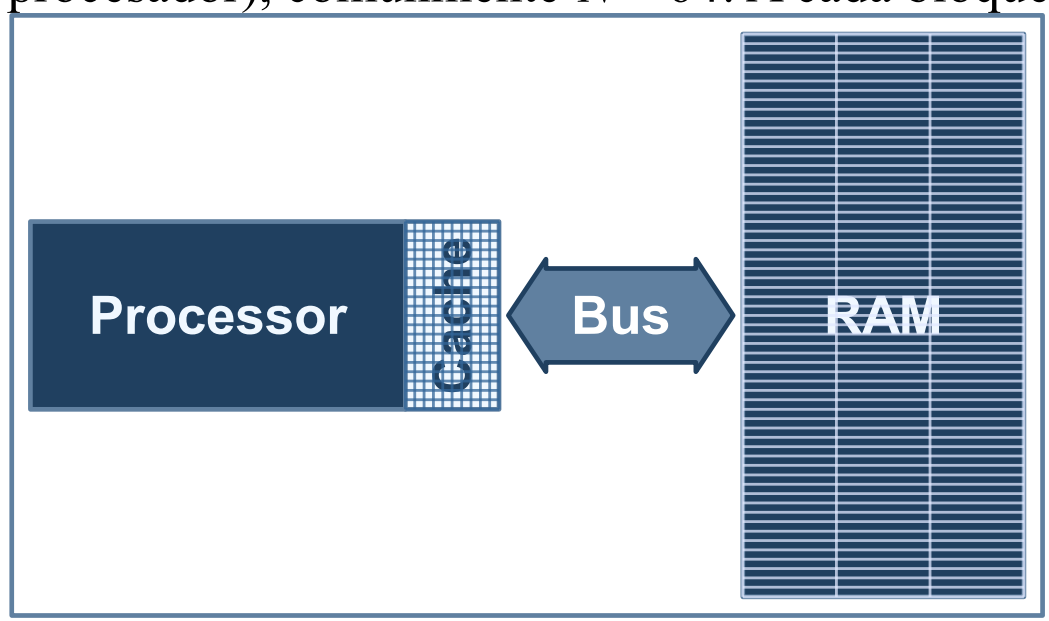
La diferencia de tiempo de acceso es demasiado grande como para no tomarla en cuenta.

Vamos a revisar cómo podemos diseñar algoritmos que aprovechen la velocidad del cache. Es decir, algoritmos que hagan más accesos al cache que a la RAM.

En las arquitecturas antiguas (pre-cache), el procesador puede leer/escribir la RAM byte por byte.

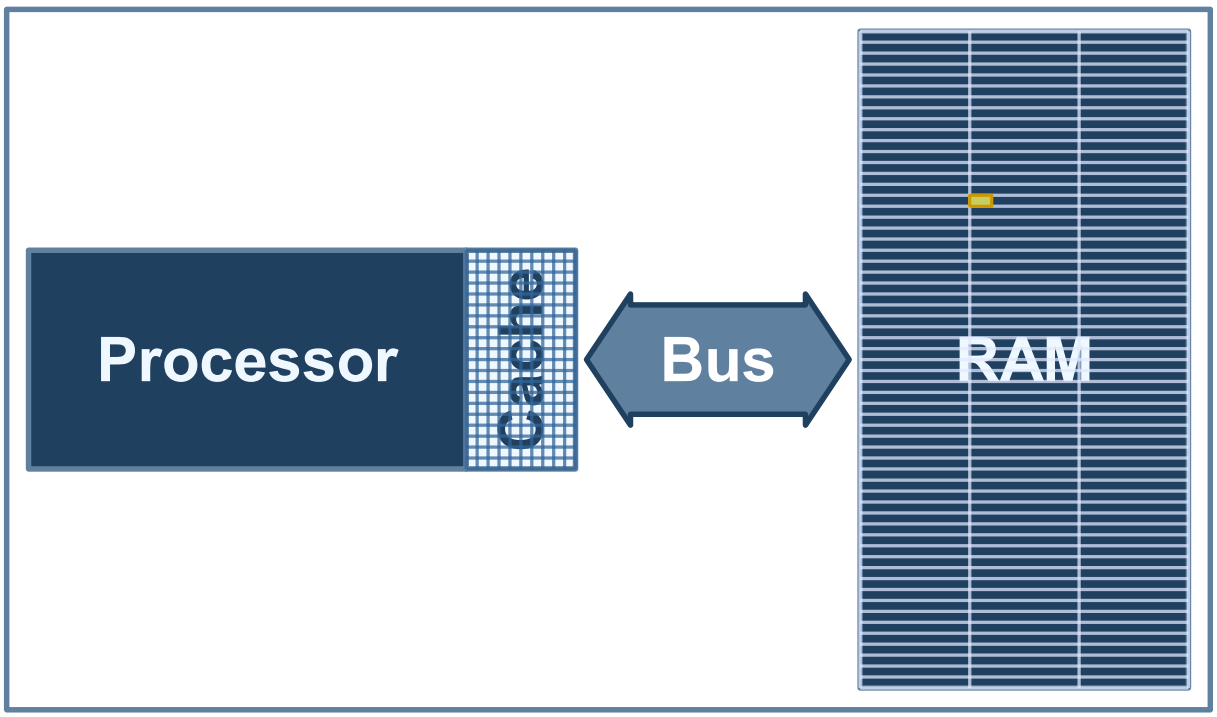


En cambio, el cache lee/escribe en la memoría RAM en **bloques** de N bytes, con $8 \leq N \leq 512$, (depende del modelo del procesador), comunmente $N = 64$. A cada bloque se le llama "cache line".



En cambio, el procesador lee/escribe el cache byte por byte.

Como se leen dos datos de la RAM



El procesador solicita d[0], éste está almacenado en una posición de la RAM

La línea del d[0] no está en el cache (cache-miss)

El cache se trae una copia de la línea

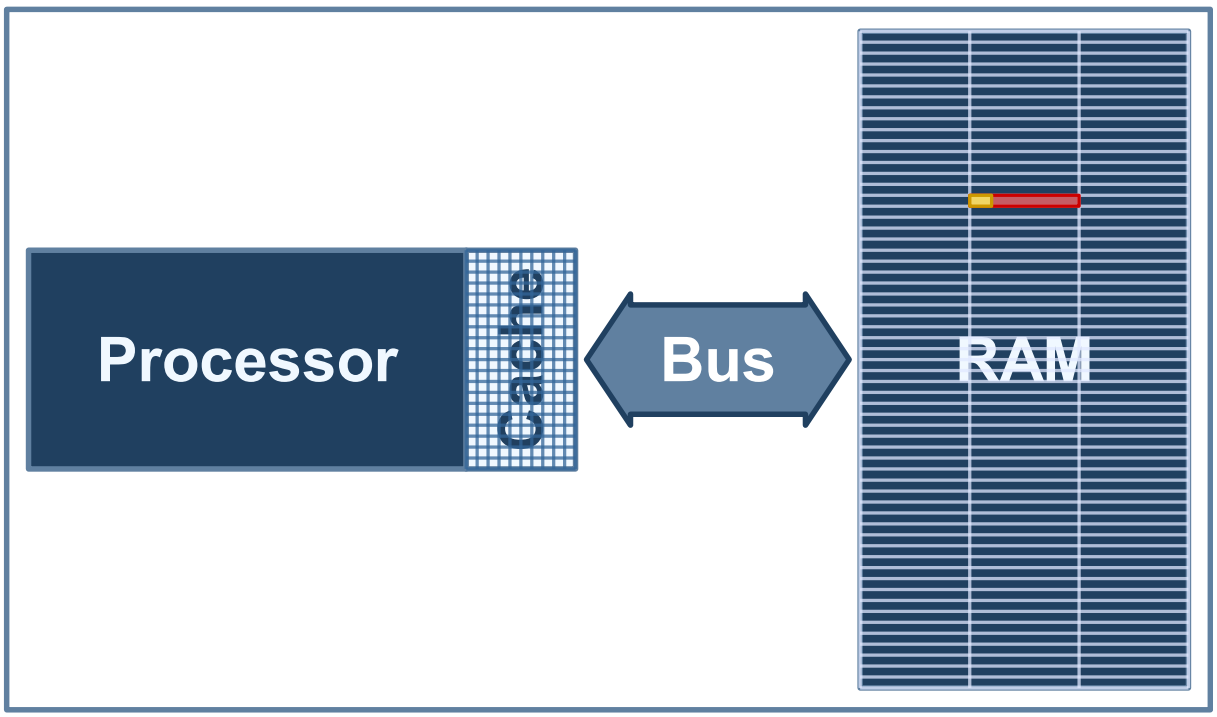
El procesador puede trabajar con d[0]

El procesador solicita d[1], éste está en el cache (cache-hit)

El procesador puede trabajar con d[1]

a = d[0];

Como se leen dos datos de la RAM



El procesador solicita d[0], éste está almacenado en una posición de la RAM

La línea del d[0] no está en el cache (**cache-miss**)

El cache se trae una copia de la línea

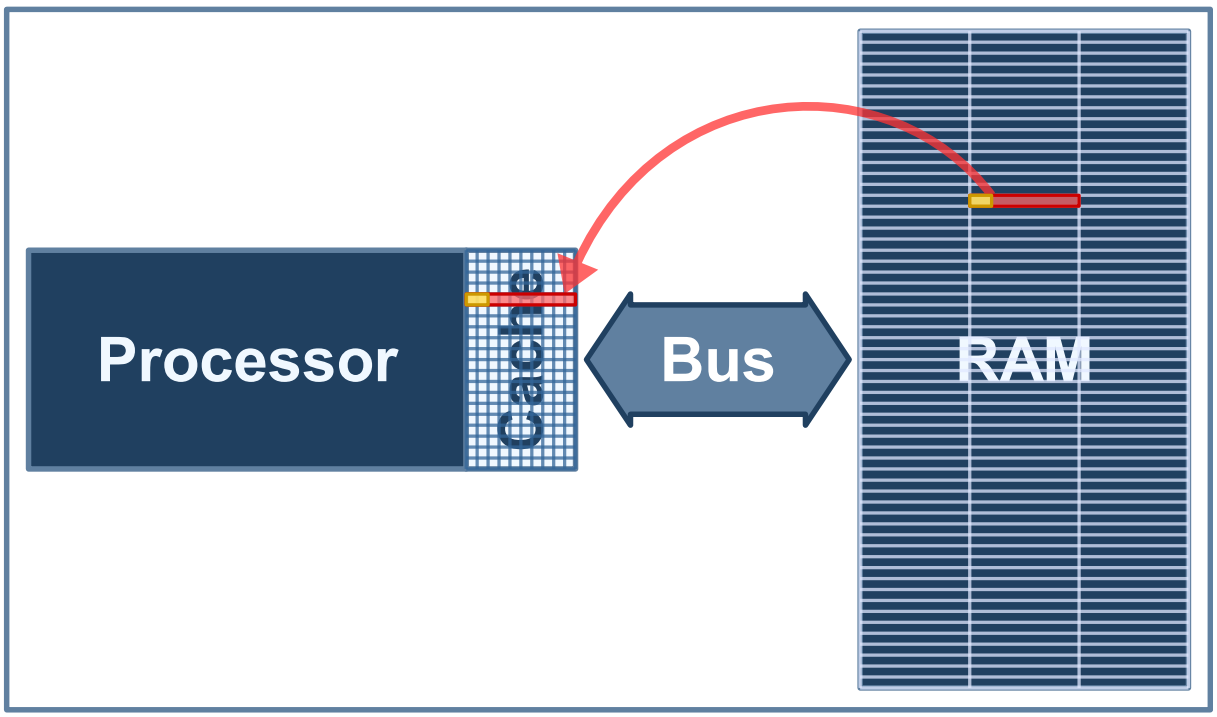
El procesador puede trabajar con d[0]

El procesador solicita d[1], éste está en el cache (cache-hit)

El procesador puede trabajar con d[1]

a = d[0];

Como se leen dos datos de la RAM



El procesador solicita d[0], éste está almacenado en una posición de la RAM

La línea del d[0] no está en el cache (cache-miss)

El cache se trae una copia de la línea

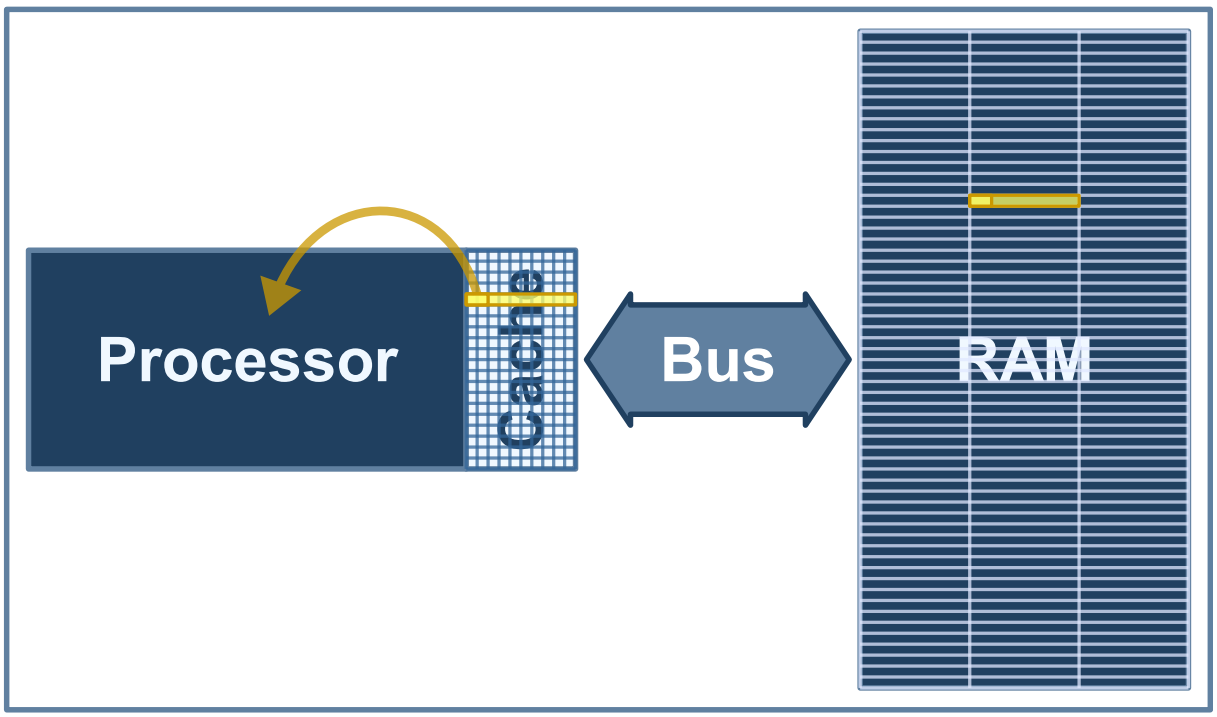
El procesador puede trabajar con d[0]

El procesador solicita d[1], éste está en el cache (cache-hit)

El procesador puede trabajar con d[1]

```
a = d[0];
```

Como se leen dos datos de la RAM



El procesador solicita d[0], éste está almacenado en una posición de la RAM

La línea del d[0] no está en el cache (cache-miss)

El cache se trae una copia de la línea

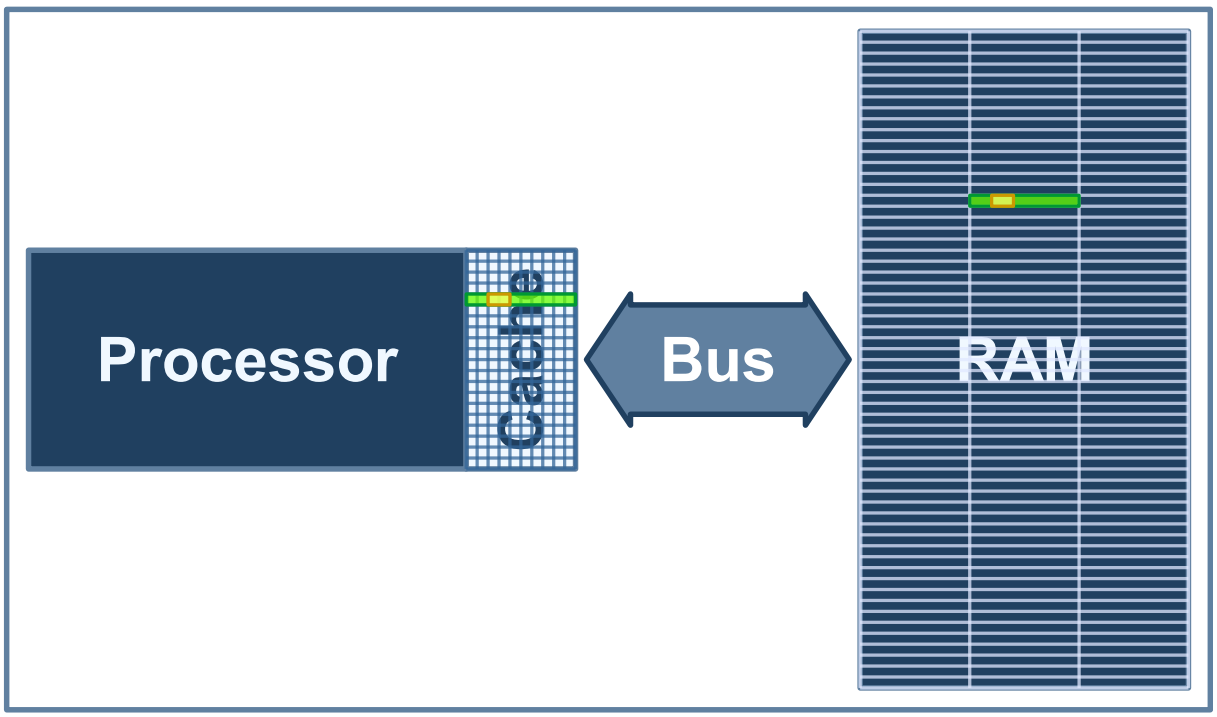
El procesador puede trabajar con d[0]

El procesador solicita d[1], éste está en el cache (cache-hit)

El procesador puede trabajar con d[1]

```
a = d[0];
```

Como se leen dos datos de la RAM



El procesador solicita d[0], éste está almacenado en una posición de la RAM

La línea del d[0] no está en el cache (cache-miss)

El cache se trae una copia de la línea

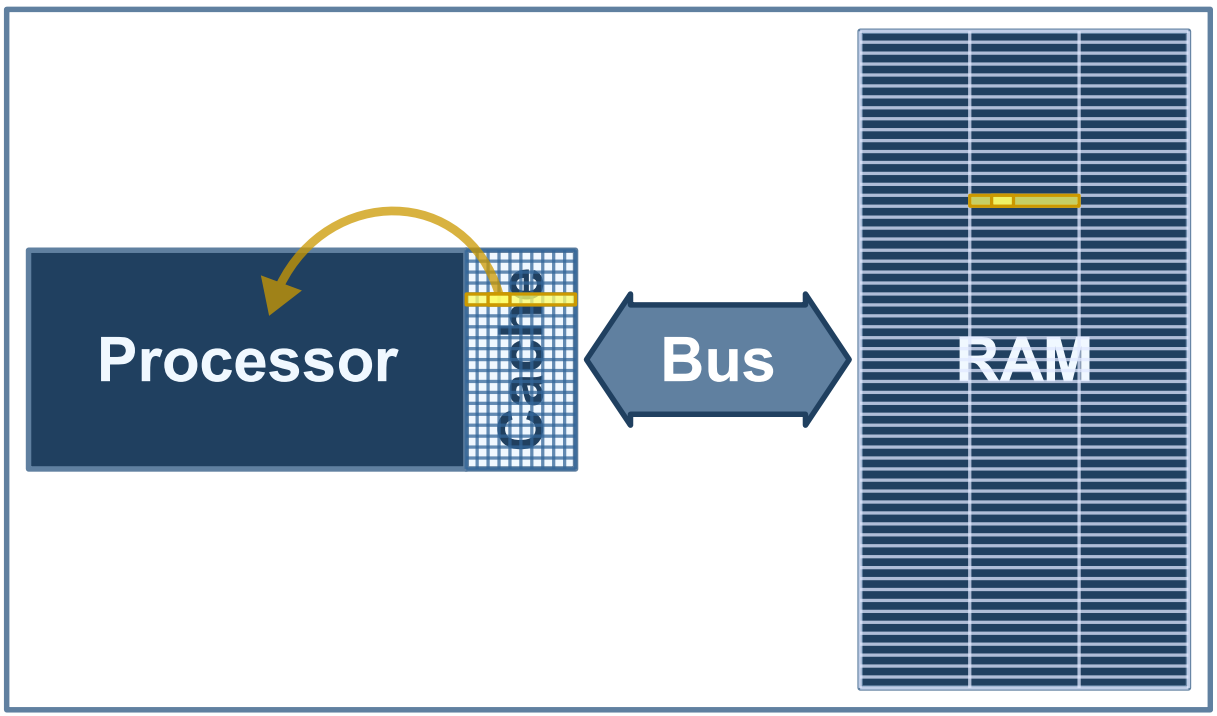
El procesador puede trabajar con d[0]

El procesador solicita d[1], éste está en el cache (**cache-hit**)

El procesador puede trabajar con d[1]

```
a = d[0];
a += d[1];
```

Como se leen dos datos de la RAM



El procesador solicita d[0], éste está almacenado en una posición de la RAM

La línea del d[0] no está en el cache (cache-miss)

El cache se trae una copia de la línea

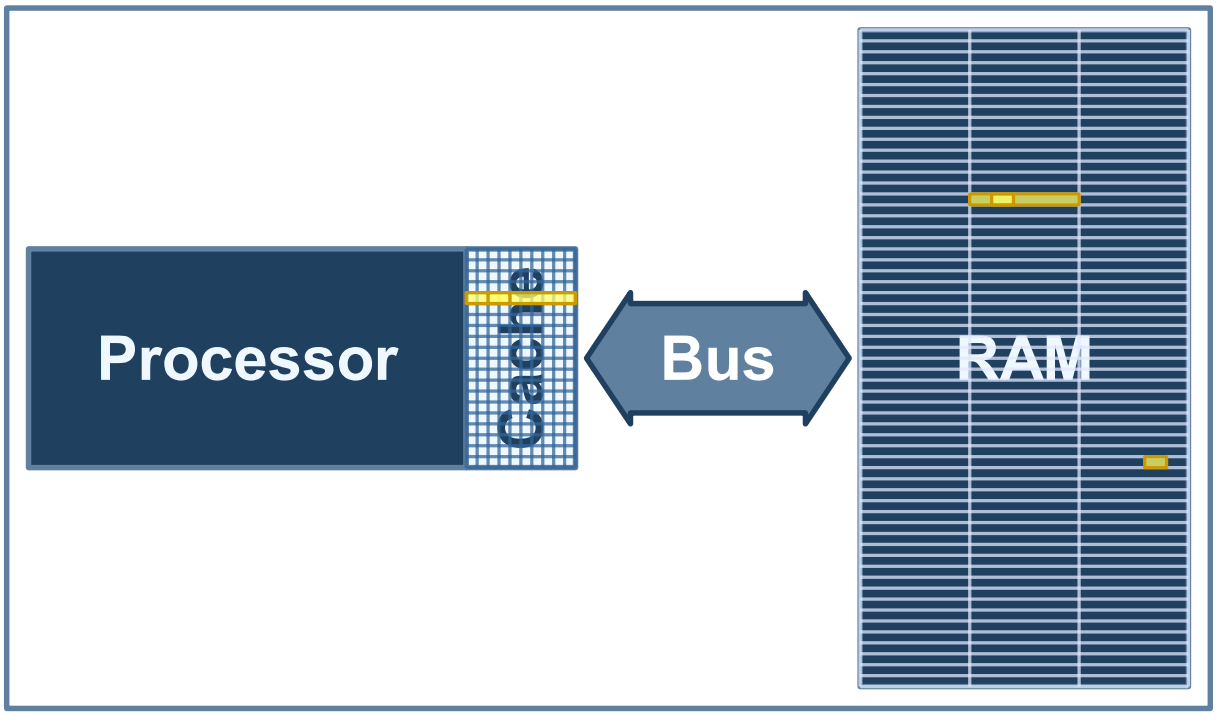
El procesador puede trabajar con d[0]

El procesador solicita d[1], éste está en el cache (cache-hit)

El procesador puede trabajar con d[1]

```
a = d[0];
a += d[1];
```

Leer otro dato



El procesador solicita d[200], éste está almacenado en una posición de la RAM

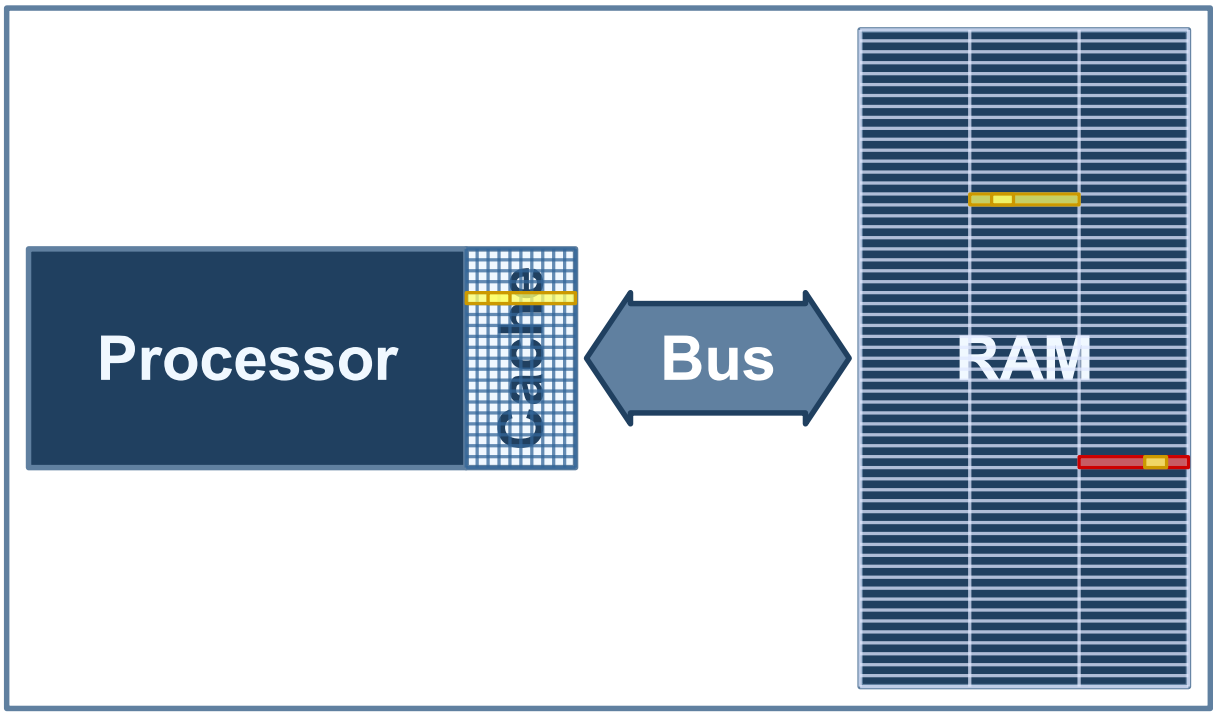
La línea del d[200] no está en el cache (cache-miss)

El cache se trae una copia de la línea

El procesador puede trabajar con d[200]

```
a = d[0];
a += d[1];
b = d[200];
```

Leer otro dato



El procesador solicita d[200], éste está almacenado en una posición de la RAM

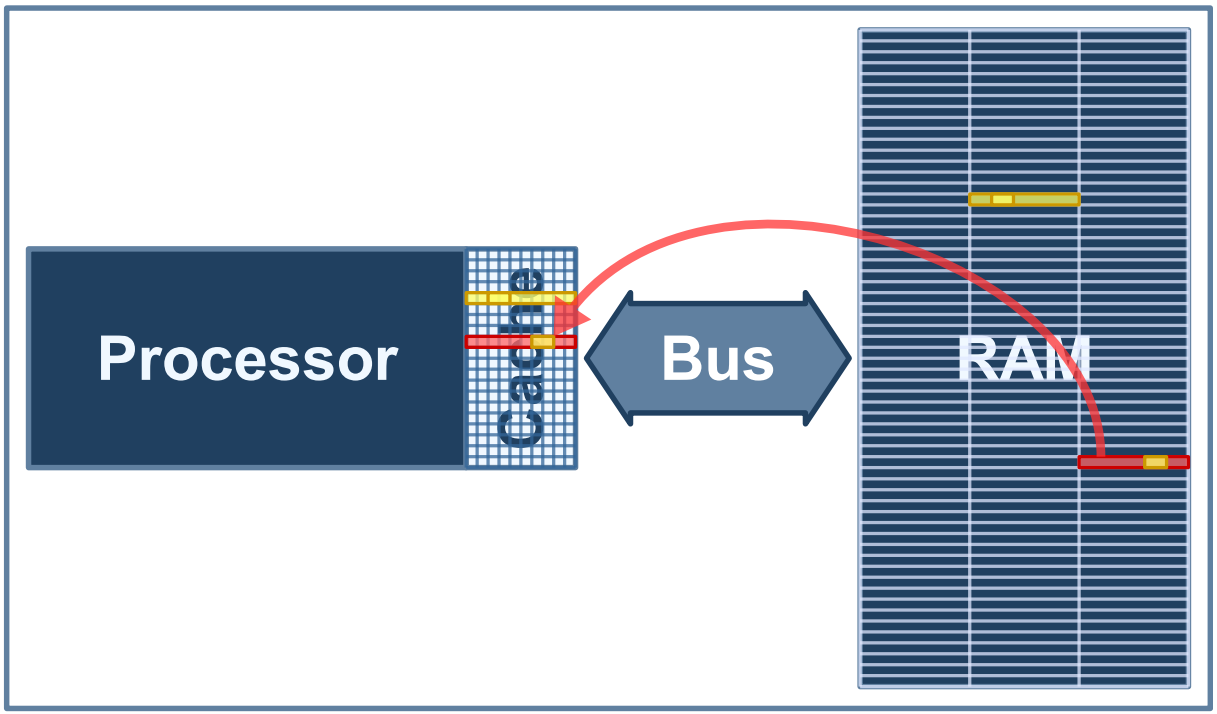
La línea del d[200] no está en el cache (**cache-miss**)

El cache se trae una copia de la línea

El procesador puede trabajar con d[200]

```
a = d[0];
a += d[1];
b = d[200];
```

Leer otro dato



El procesador solicita d[200], éste está almacenado en una posición de la RAM

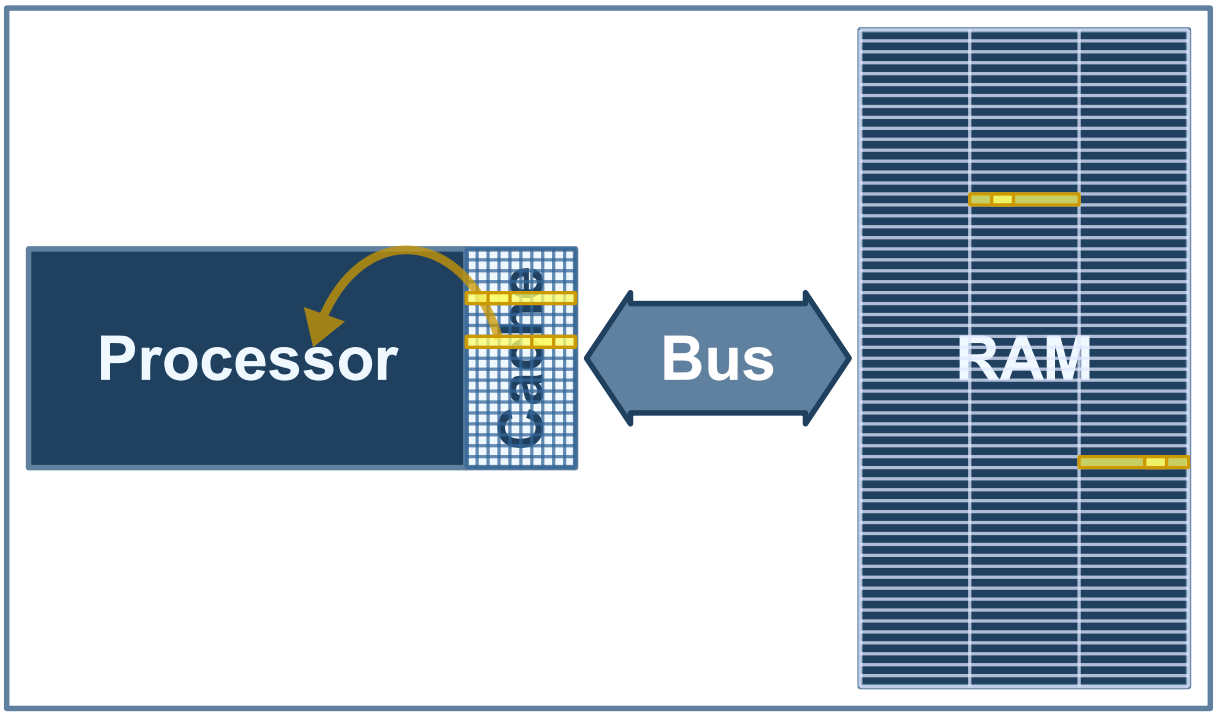
La línea del d[200] no está en el cache (cache-miss)

El cache se trae una copia de la línea

El procesador puede trabajar con d[200]

```
a = d[0];
a += d[1];
b = d[200];
```

Leer otro dato



El procesador solicita d[200], éste está almacenado en una posición de la RAM

La línea del d[200] no está en el cache (cache-miss)

El cache se trae una copia de la línea

El procesador puede trabajar con d[200]

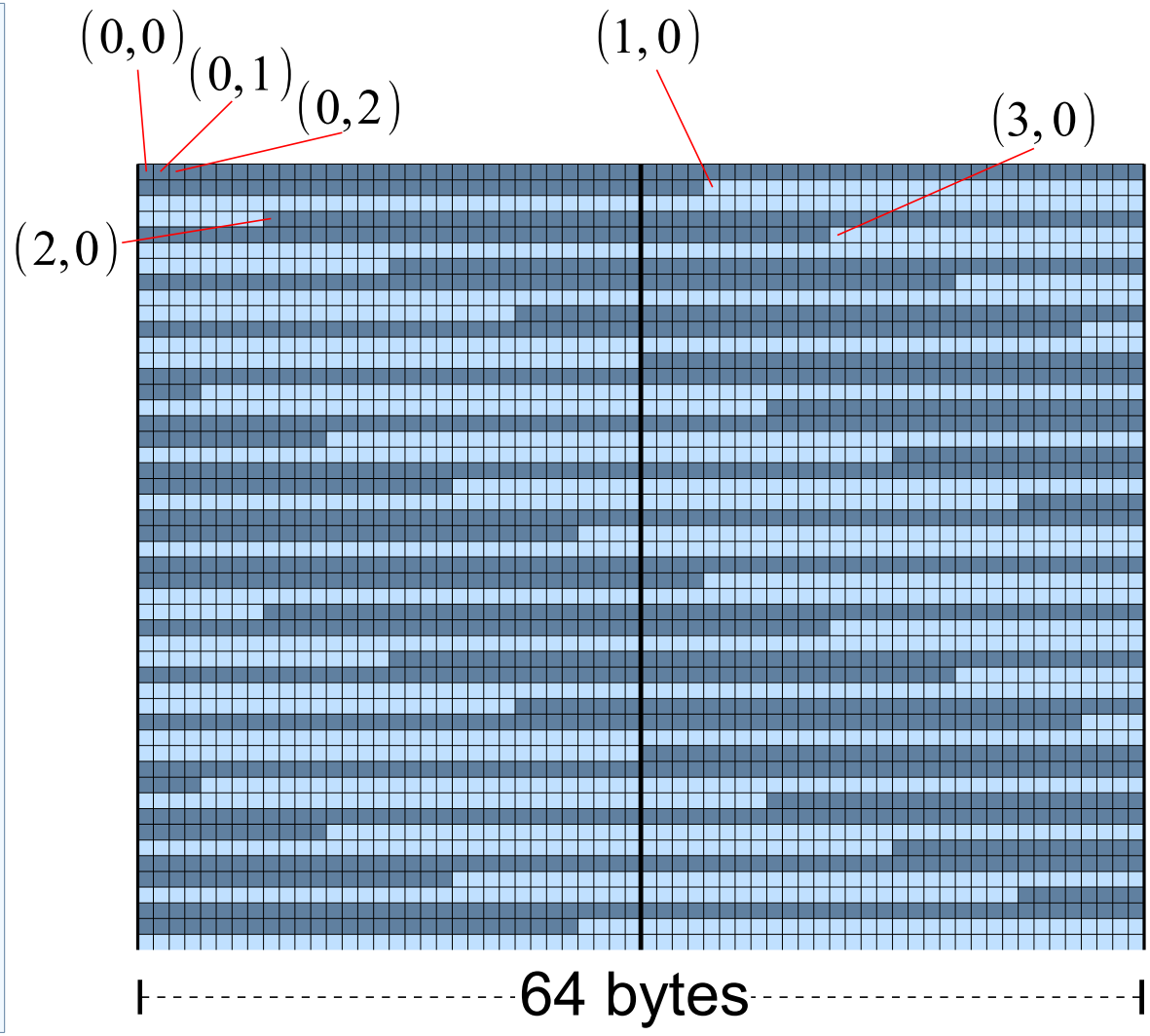
```
a = d[0];
a += d[1];
b = d[200];
```

Cuando se va a escribir en la memoria, el procesador de todas formas tiene que leer la línea de cache primero [Drep07 p14].

Vamos a suponer que el tamaño de las líneas de caché es de 32 bytes.

¿Cómo funciona el primer código?

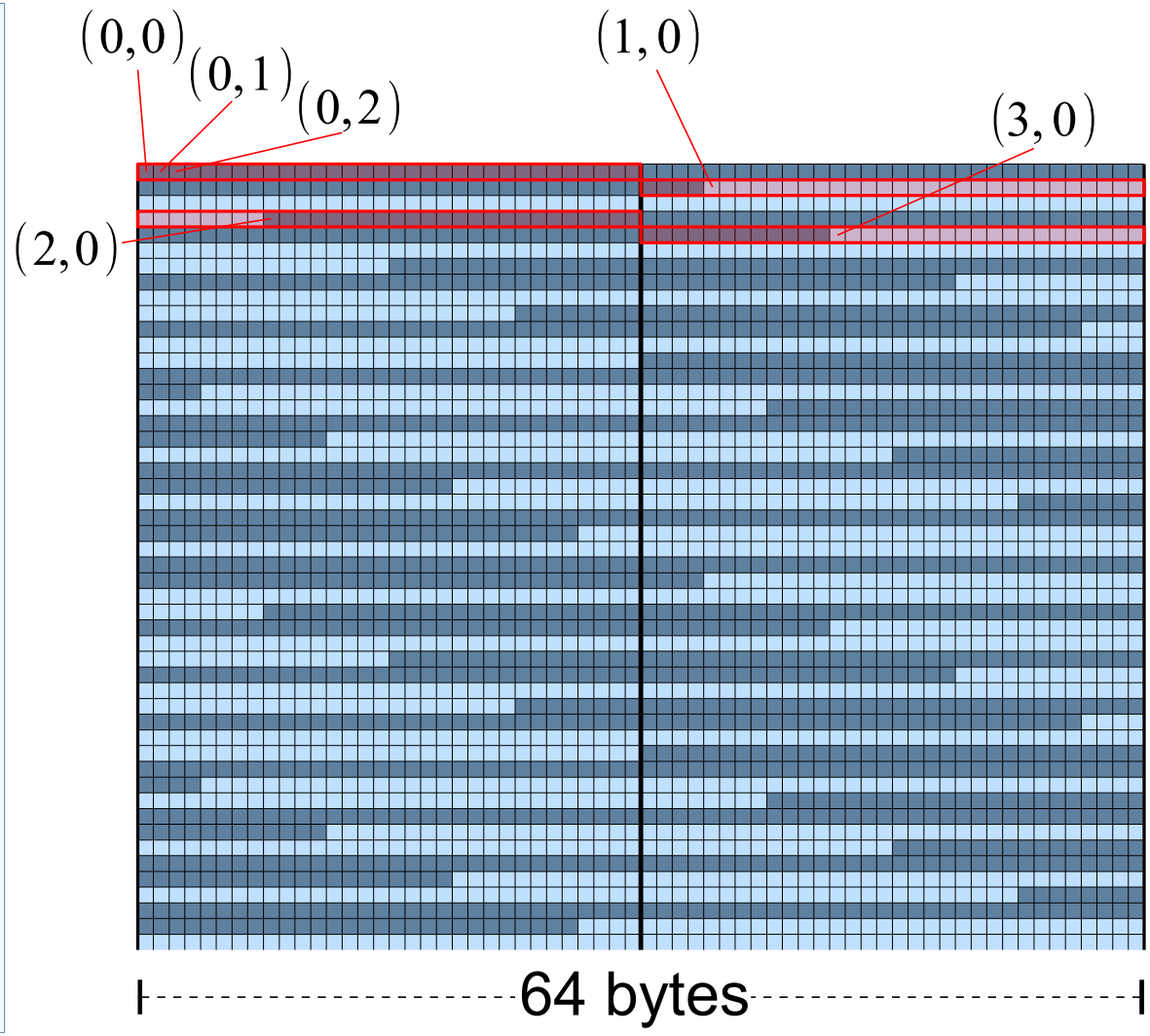
```
#define ROWS 10000000
#define COLS 100

int main()
{
    char* matrix = new char[ROWS*COLS];

    for (int j = 0; j < COLS; ++j)
    {
        for (int i = 0; i < ROWS; ++i)
        {
            matrix[i*COLS + j] = 0;
        }
    }

    delete [] matrix;

    return 0;
}
```



Vamos a suponer que el tamaño de las líneas de caché es de 32 bytes.

¿Cómo funciona el primer código?

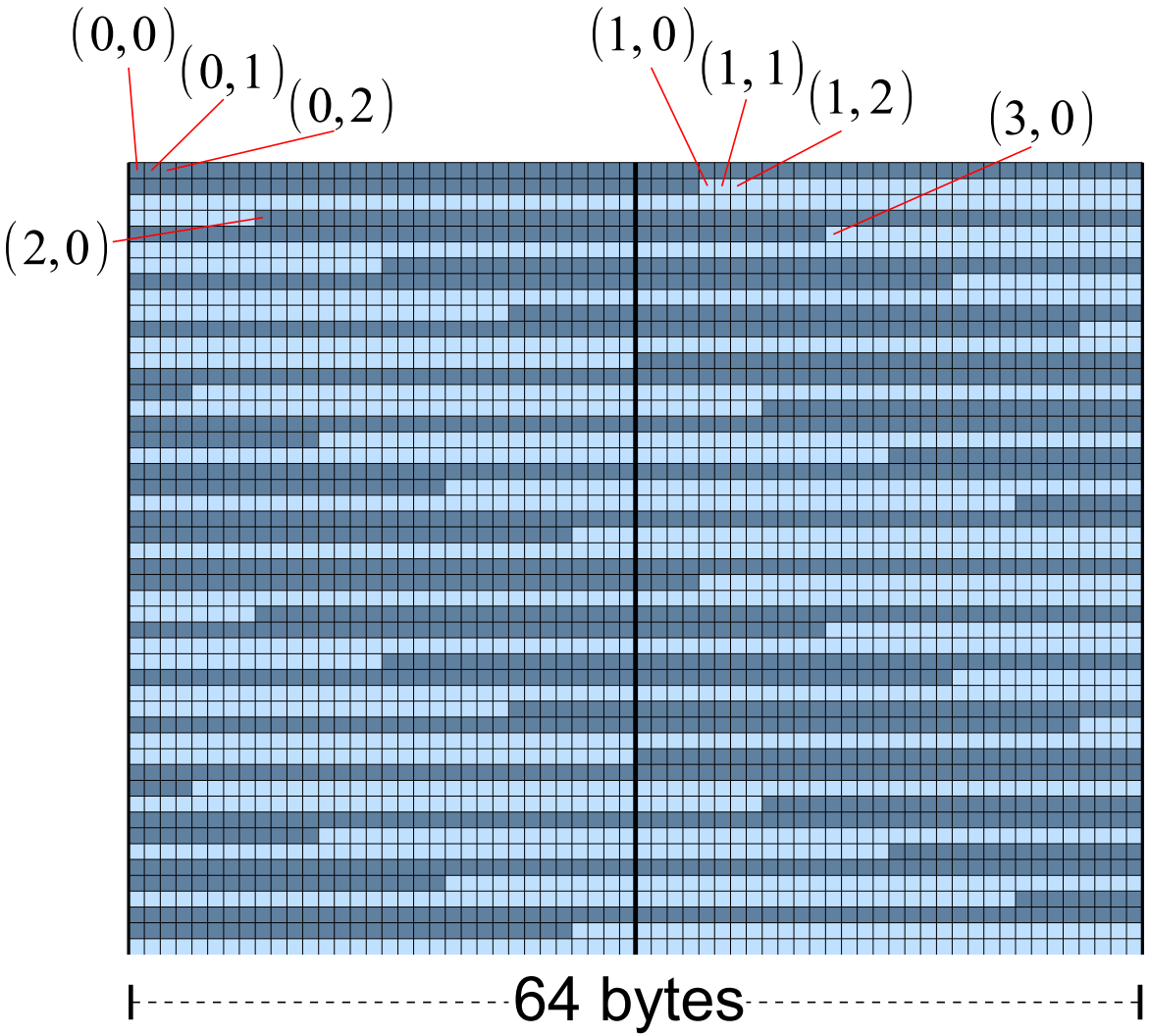
```
#define ROWS 10000000
#define COLS 100

int main()
{
    char* matrix = new char[ROWS*COLS];

    for (int j = 0; j < COLS; ++j)
    {
        for (int i = 0; i < ROWS; ++i)
        {
            matrix[i*COLS + j] = 0;
        }
    }

    delete [] matrix;

    return 0;
}
```



Al accesar $(0,0)$ se produce un **cache-miss**, $(1,0)$ un **cache-miss**, $(2,0)$ un **cache-miss**, $(3,0)$ **un cache-miss**, etc...

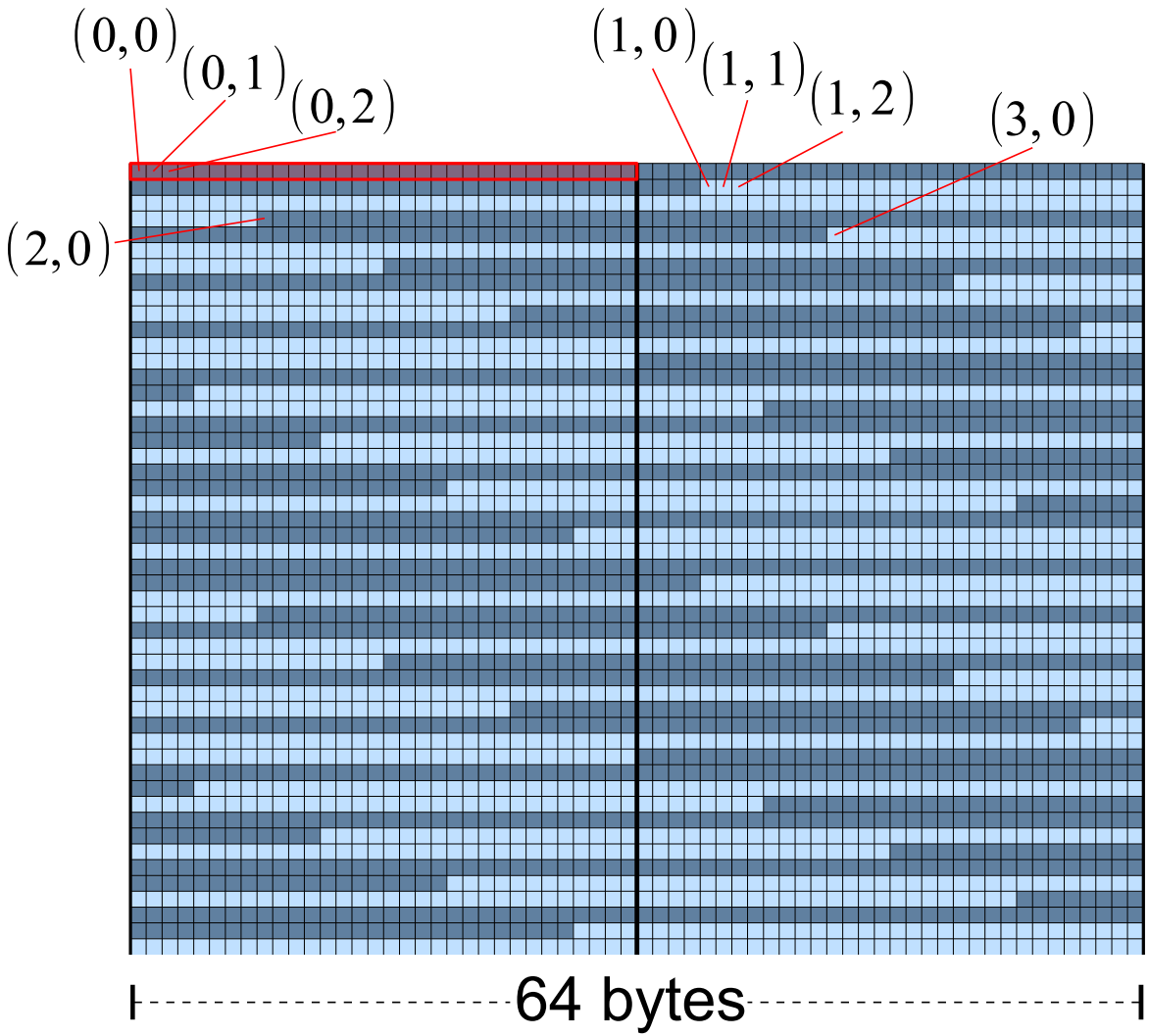
Por cada **cache-miss** se carga una línea de cache 32 bytes desde la RAM.

En cambio, en el otro código...




```
#define ROWS 10000000
#define COLS 100

int main()
{
    char* matrix = new char[ROWS*COLS];

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLS; ++j)
        {
            matrix[i*COLS + j] = 0;
        }
    }

    delete [] matrix;

    return 0;
}
```

En cambio, en el otro código...




```
#define ROWS 10000000
#define COLS 100

int main()
{
    char* matrix = new char[ROWS*COLS];

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLS; ++j)
        {
            matrix[i*COLS + j] = 0;
        }
    }

    delete [] matrix;

    return 0;
}
```

Al accesar $(0,0)$ se produce un **cache-miss**, $(0,1)$ un **cache-hit**, $(0,2)$ un **cache-hit,** $(0,3)$ **un cache-hit** etc...

Despúes, $(0,31)$ **cache-hit**, $(0,32)$ un **cache-miss**, $(0,33)$ un **cache-hit**, $(0,34)$ un **cache-hit**, etc...

Cada **cache-hit** implica accesar el cache y **no** la RAM (un órden de magnitud más rápido).

¿Cómo está organizada la memoria de un programa?

Cuando un programa es ejecutado, el sistema operativo toma el archivo con el código binario (archivos .exe en Windows o archivos ELF en Unix-like). Después el sistema operativo reserva espacio en memoria para los datos de trabajo del programa.

La memoria utilizada por el programa está dividida en cinco partes:

Heap: Almacena los datos reservados dinámicamente (malloc o new).

Stack: Acumula las variables declaradas dentro de las funciones. Almacena el punto de retorno a la función que llamó a la función actual (permite hacer recursividad).

Data: Contiene las variables globales y estáticas (static) que se definen con un valor inicial.

BSS: Contiene las variables globales y estáticas (static) que se definen sin valor inicial.

Text/Code: Contiene código máquina del programa ejecutable.

¿Cuál es cual?

En qué segmento de la memoria será almacenada cada variable del siguiente código:

```
#define SIZE 10
int f = 20;
int* vector;

int Factorial(int n)
{
    static int call_count = 0;
    ++call_count;

    if (n <= 1)
        return 1;
    return n*Factorial(n - 1);
}

int main()
{
    int r = Factorial(f);
    vector = new int[SIZE];
    for (int i = 0; i < SIZE; ++i)
        vector[i] = i*r*f;
    delete [] vector;
    return 0;
}
```

¿Cuál es cual?

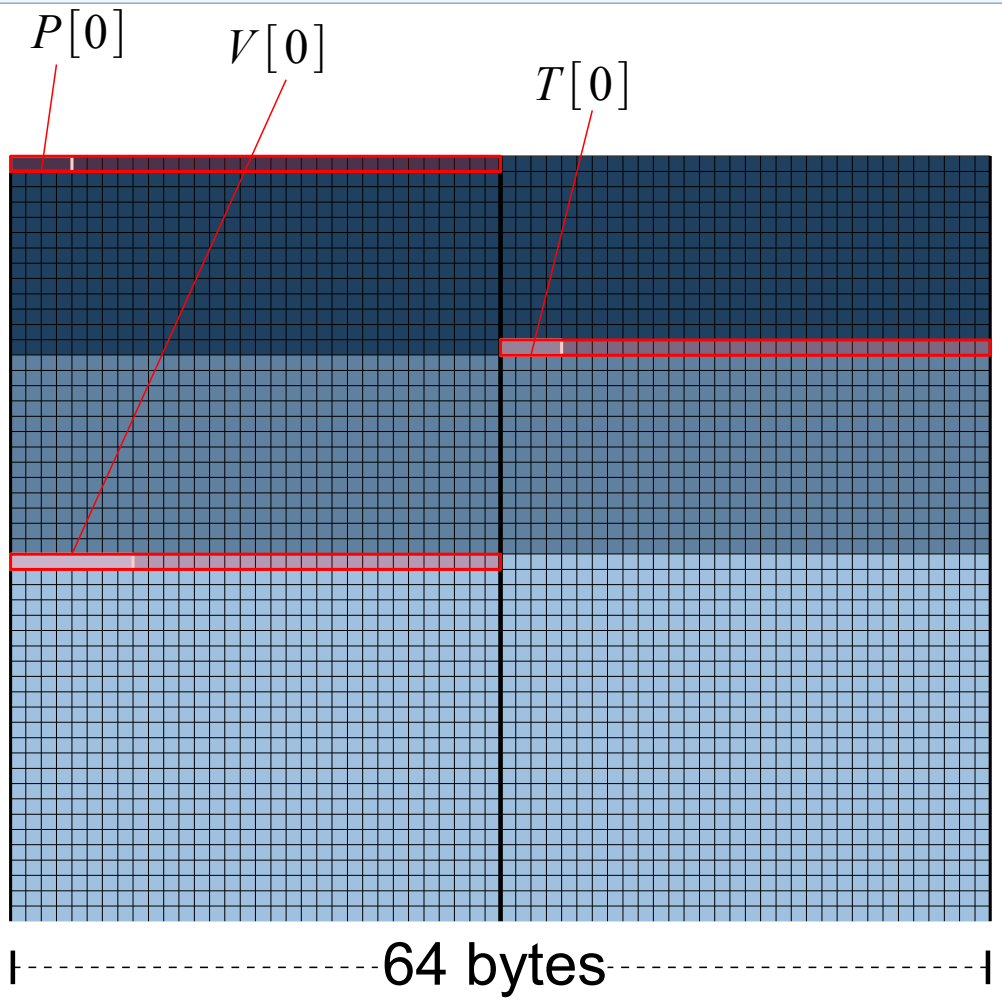
En qué segmento de la memoria será almacenada cada variable del siguiente código:

```
#define SIZE 10
int f = 20;
int* vector;

int Factorial(int n)
{
    static int call_count = 0;
    ++call_count;

    if (n <= 1)
        return 1;
    return n*Factorial(n - 1);
}

int main()
{
    int r = Factorial(f);
    vector = new int[SIZE];
    for (int i = 0; i < SIZE; ++i)
        vector[i] = i*r*f;
    delete [] vector;
    return 0;
}
```

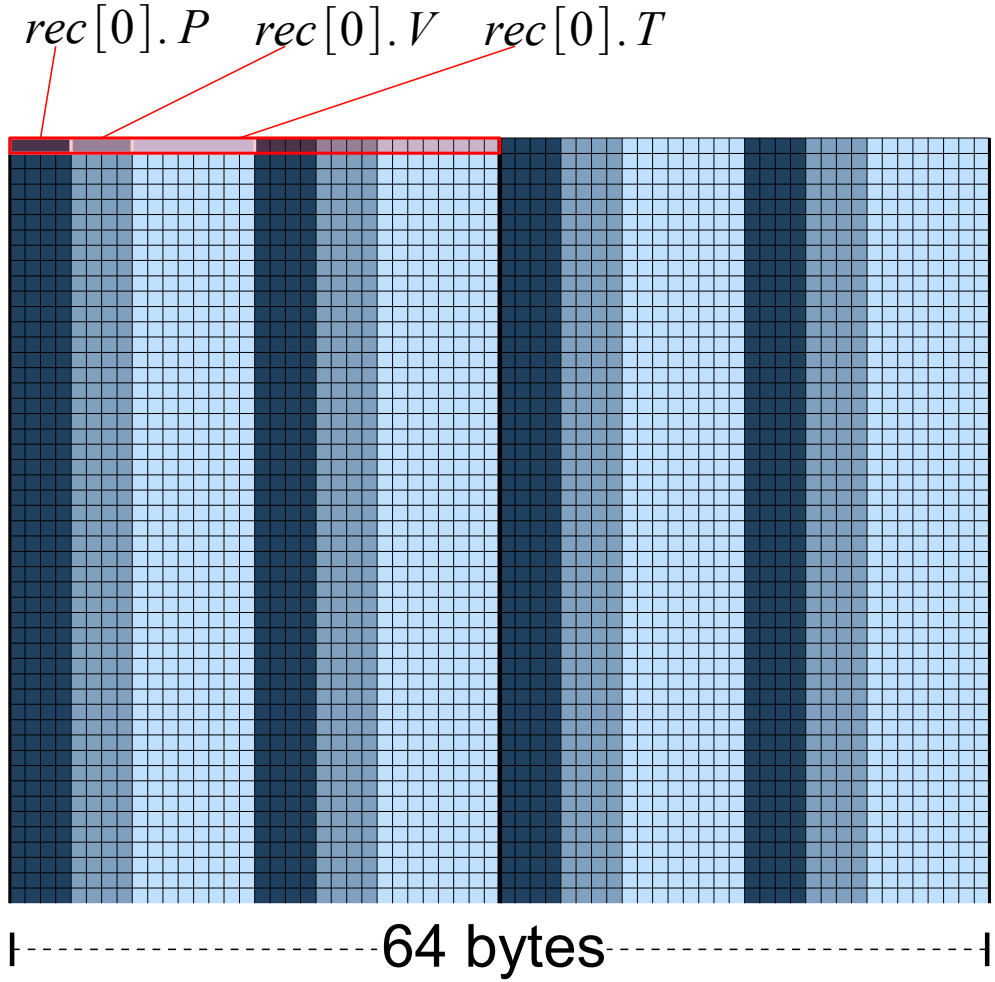
Estratégias de programación para aprovechar el cache

Accesar (leer/escribir) localidades de memoria contiguas.

```
vector<int> P(200);

vector<int> V(200);

vector<double> T(200);
```




```
struct Record {
    int P, V;
    double T;
};
vector<Record> rec(200);
```



Otras sugerencias

- Las variables deben ser declaradas dentro de la función en la cual serán utilizados, así estas estarán almacenados en el stack, lo cual facilitará que caigan en el cache L1 [Fog14].

- Una línea de cache se accesa más rápido si se la lee de forma secuencial hacia adelante (i++). Leerlo en forma inversa es un poco más lento (i--).

Optimizar el código compilado

Los compiladores tiene estrategías para mejorar el código objeto generado, como:

- Desenrollar ciclos
- Reutilizar valores
- Poner valores temporales en registros
- Mezclar varias líneas de código
- Simplificar expresiones matemáticas
- Eliminar código no utilizado

Estas optimizaciones pueden evitar que el código se pueda depurar (el debugger no puede identificar donde inicia o termina una línea de código).

Con GCC ésto se haría con

g++ **-O3** -o <executable> <archivo.cpp>

Otros compiladores tienen opciones similares.

Al utilizar esta directiva el tiempo de compilación puede aumentar considerablemente.

¿Que tan notorio es?

```
#define ROWS 10000000
#define COLS 100

int main()
{
    char* matrix = new char[ROWS*COLS];

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLS; ++j)
        {
            matrix[i*COLS + j] = 0;
        }
    }

    delete [] matrix;

    return 0;
}
```

```
g++ -o example2 example2.cpp
time ./example2
```

```
#define ROWS 10000000
#define COLS 100

int main()
{
    char* matrix = new char[ROWS*COLS];

    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLS; ++j)
        {
            matrix[i*COLS + j] = 0;
        }
    }

    delete [] matrix;

    return 0;
}
```

```
g++ -O3 -o example2 example2.cpp
time ./example2
```

Multiplicación de matrices

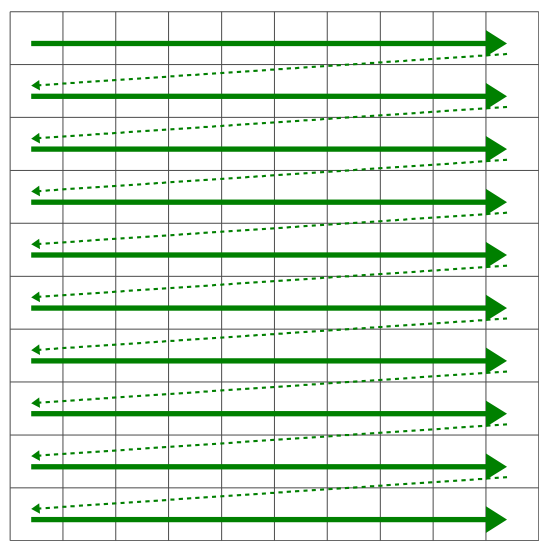
Sean **A**, **B**, **C** matrices de tamaño $n \times n$. La multiplicación $\mathbf{C} = \mathbf{A}\,\mathbf{B}$, es

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

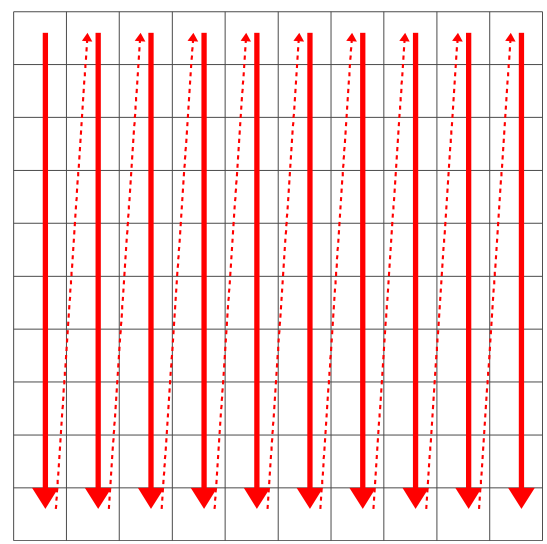
El código “ingenuo” para hacer esta multiplicación sería:

```
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        float sum = 0;
        for (int k = 0; k < N; ++k)
        {
            sum += A[i*N + k]*B[k*N + j];
        }
        C[i*N + j] = sum;
    }
}
```

```
g++ -O3 -o mult1 mult1.cpp
./mult1
```



a_{ik}



b_{kj}

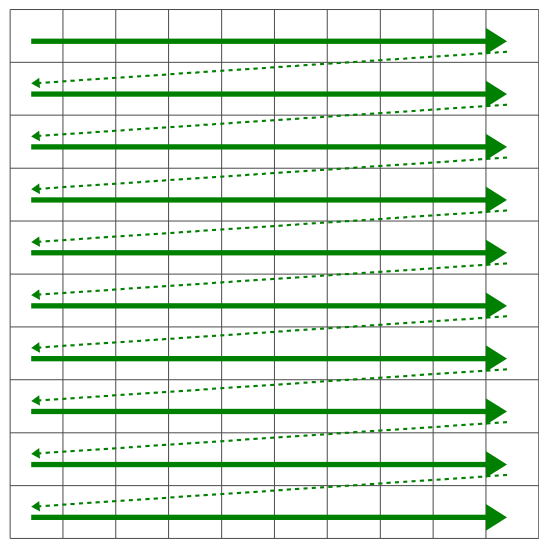
Una forma de mejorarlo sería utilizar $\mathbf{B}^{\mathrm{T}}$ en vez de $\mathbf{B}$, la multiplicación $\mathbf{C}=\mathbf{A}\left(\mathbf{B}^{\mathrm{T}}\right)^{\mathrm{T}}$, con

$$c_{ij}=\sum_{k=1}^{n} a_{ik} b^{\mathrm{T}}_{jk}.$$

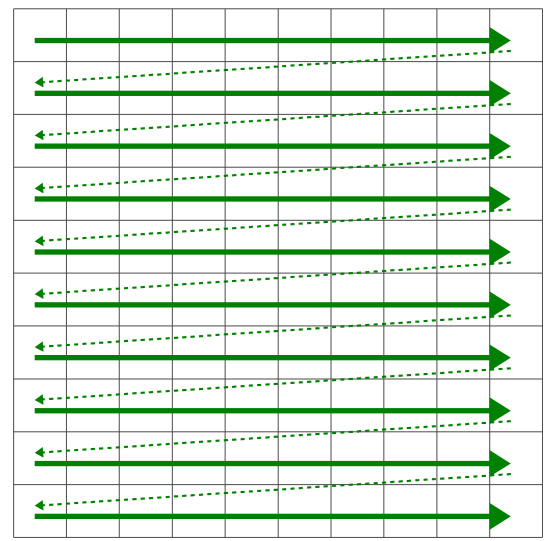
Utilizamos más memoria al almacenar $\mathbf{B}^{\mathrm{T}}$. El código en sí no cambia mucho:

```
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        float sum = 0;
        for (int k = 0; k < N; ++k)
        {
            sum += A[i*N + k]*Bt[j*N + k];
        }
        C[i*N + j] = sum;
    }
}
```

```
g++ -O3 -o mult2 mult2.cpp
./mult2
```



a_{ik}



b^{T}_{jk}

¿Cuánto se reduce el tiempo de ejecución?

¿Cómo hacer la multiplicación eficiente sin utilizar $\mathbf{B}^{\mathrm{T}}$?

La idea es hacer la multiplicación por bloques [Prok99]. Con bloques que quepan en el cache L1.

Viendo las matrices por bloques, tenemos $\mathbf{C}=\mathbf{A}\,\mathbf{B}$ como

$$\begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} & \mathbf{C}_{13} & \cdots \\ \mathbf{C}_{21} & \mathbf{C}_{22} & \mathbf{C}_{23} & \cdots \\ \mathbf{C}_{31} & \mathbf{C}_{32} & \mathbf{C}_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{A}_{13} & \cdots \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} & \cdots \\ \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} & \mathbf{B}_{13} & \cdots \\ \mathbf{B}_{21} & \mathbf{B}_{22} & \mathbf{B}_{23} & \cdots \\ \mathbf{B}_{31} & \mathbf{B}_{32} & \mathbf{B}_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix},$$

con $\mathbf{A}_{ij}$, $\mathbf{B}_{ij}$, $\mathbf{C}_{ij}$ de tamaño $s\times s$, con

$$i=1,2,\ldots,\frac{n}{s},$$

$$j=1,2,\ldots,\frac{n}{s},$$

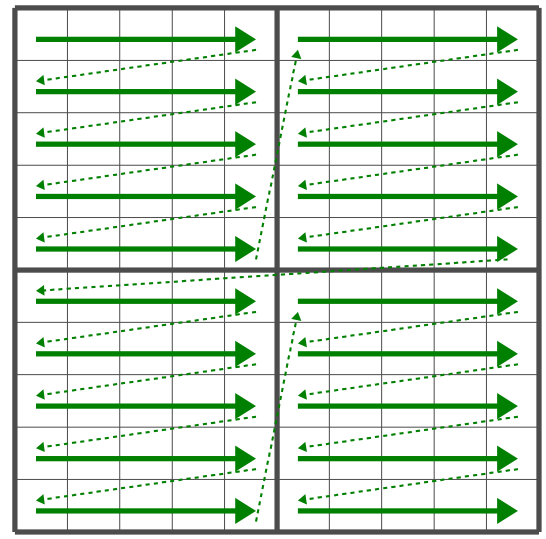
(aquí usaremos la suposición de que $\frac{n}{s}$ es entero).

El algoritmo sería

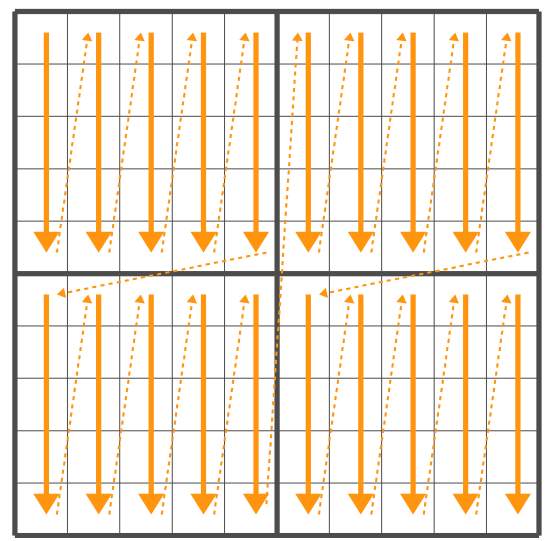
```
for i ← 1, 2, ..., n/s
    for j ← 1, 2, ..., n/s
        for k ← 1, 2, ..., n/s
            C_ij ← C_ij + A_ik B_kj
```

El truco es elegir un s de tal forma que el producto dentro del bloque quepa en el cache.

```
for (int I = 0; I < N/S; ++I)
{
    for (int J = 0; J < N/S; ++J)
    {
        for (int K = 0; K < N/S; ++K)
        {
            for (int i = 0; i < S; ++i)
            {
                for (int j = 0; j < S; ++j)
                {
                    float sum = 0;
                    for (int k = 0; k < S; ++k)
                    {
                        sum += A[(I*S+i)*N + (K*S+k)]*B[(K*S+k)*N + (J*S+j)];
                    }
                    C[(I*S + i)*N + (J*S + j)] += sum;
                }
            }
        }
    }
}
```



a_{ik}



b_{kj}

```
g++ -O3 -o mult3 mult3.cpp
./mult3
```

Versiones más sofisticadas de esta estratégia se pueden encontrar en [Prok99].

Cuál es la lección

Es importante conocer que existen niveles de memoria (RAM, cache L2, cache L1, registros) para diseñar código que sea eficiente:

El acceso a la memoria es el mayor cuello de botella.

Puede ser muy conveniente utilizar más memoria para poder eficientar el uso del caché. Sin embargo, lo mejor es:

Buscar algoritmos que sean “Cache-Oblivious” [Prok99].

Trabajar con vectores

Elijamos $n \leftarrow 10000000$. Sea $\mathbf{a} \in \mathbb{Z}^n$, con $\left(\mathbf{a}\right)_i = i$.

```
#include <vector>

int main()
{
    int size = 10000000;
    std::vector<int> a;

    for (int t = 0; t < 120; ++t)
    {
        a.clear();
        for (int i = 0; i < size; ++i)
        {
            a.push_back(i);
        }
    }

    return a.back();
}
```

```
g++ vector_push_back.cpp -o vector_push_back
time ./vector_push_back
```

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
20.229 s						

Trabajar con vectores

Elijamos $n \leftarrow 10000000$. Sea $\mathbf{a} \in \mathbb{Z}^n$, con $\left(\mathbf{a}\right)_i = i$.

```
#include <vector>

int main()
{
    int size = 10000000;
    std::vector<int> a(size);

    for (int t = 0; t < 120; ++t)
    {
        int i = 0;
        for (std::vector<int>::iterator it = a.begin(); it != a.end(); ++it, ++i)
        {
            *it = i;
        }
    }

    return a.back();
}
```

```
g++ vector_iterator.cpp -o vector_iterator
time ./vector_iterator
```

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
20.229 s	18.027 s					

Trabajar con vectores

Elijamos $n \leftarrow 10000000$. Sea $\mathbf{a} \in \mathbb{Z}^n$, con $\left(\mathbf{a}\right)_i = i$.

```
#include <vector>

int main()
{
    int size = 10000000;
    std::vector<int> a(size);

    for (int t = 0; t < 120; ++t)
    {
        for (int i = 0; i < a.size(); ++i)
        {
            a[i] = i;
        }
    }

    return a[size - 1];
}
```

```
g++ vector_call_size.cpp -o vector_call_size
time ./vector_call_size
```

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
20.229 s	18.027 s	8.748 s				

Trabajar con vectores

Elijamos $n \leftarrow 10000000$. Sea $\mathbf{a} \in \mathbb{Z}^n$, con $\left(\mathbf{a}\right)_i = i$.

```
#include <vector>

int main()
{
    int size = 10000000;
    std::vector<int> a(size);

    for (int t = 0; t < 120; ++t)
    {
        for (int i = 0; i < size; ++i)
        {
            a[i] = i;
        }
    }

    return a[size - 1];
}
```

```
g++ vector_simple.cpp -o vector_simple
time ./vector_simple
```

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
20.229 s	18.027 s	8.748 s	6.790 s			

Trabajar con vectores

Elijamos $n \leftarrow 10000000$. Sea $\mathbf{a} \in \mathbb{Z}^n$, con $(\mathbf{a})_i = i$.

```
#include <malloc.h>

int main(void)
{
    int size;
    char* a;
    int t, i;

    size = 10000000;
    a = (int*)malloc(size*sizeof(int));

    for (t = 0; t < 120; ++t)
    {
        for (i = 0; i < size; ++i)
        {
            a[i] = i;
        }
    }

    return a[size - 1];
}
```

```
gcc pointer.c -o pointer
time ./pointer
```

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
20.229 s	18.027 s	8.748 s	6.790 s	4.913 s		

Trabajar con vectores

Elijamos $n \leftarrow 10000000$. Sea $\mathbf{a} \in \mathbb{Z}^n$, con $(\mathbf{a})_i = i$.

```
template <typename T>
struct vector
{
    T* entry;
    int size;

    vector(int new_size = 0)
    :   entry(new T[new_size]),
        size(new_size)
    {
    }

    ~vector() throw()
    {
        delete [] entry;
    }

    inline T& operator [] (int i) const throw()
    {
        return entry[i];
    }
};
```

```
int main()
{
    int size = 10000000;
    vector<int> a(size);

    for (int t = 0; t < 120; ++t)
    {
        for (int i = 0; i < a.size; ++i)
        {
            a[i] = i;
        }
    }

    return 0;
}
```

```
g++ other_operator.cpp -o other_operator
time ./other_operator
```

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
20.229 s	18.027 s	8.748 s	6.790 s	4.913 s	6.260 s	

Trabajar con vectores

Elijamos $n \leftarrow 10000000$. Sea $\mathbf{a} \in \mathbb{Z}^n$, con $(\mathbf{a})_i = i$.

```
template <typename T>
struct vector
{
    T* entry;
    int size;

    vector(int new_size = 0)
    :   entry(new T[new_size]),
        size(new_size)
    {
    }

    ~vector() throw()
    {
        delete [] entry;
    }
};
```

```
int main()
{
    int size = 10000000;
    vector<int> a(size);

    for (int t = 0; t < 120; ++t)
    {
        for (int i = 0; i < a.size; ++i)
        {
            a.entry[i] = i;
        }
    }

    return 0;
}
```

```
g++ other_simple.cpp -o other_simple
time ./other_simple
```

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
20.229 s	18.027 s	8.748 s	6.790 s	4.913 s	6.260 s	4.878 s

El compilador puede optimizar hasta cierto punto los contenedores e iteradores de la STL, utilizando la optimización con -O2 y -O3.

Tiempos originales

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
20.229 s	18.027 s	8.748 s	6.790 s	4.913 s	6.260 s	4.878 s

Tiempos compilando con -O3

vector_push_back.cpp	vector_iterator.cpp	vector_call_size.cpp	vector_simple.cpp	pointer.c	other_operator.cpp	other_simple.cpp
2.034 s	1.541 s	1.603 s	1.555 s	1.406 s	1.407 s	1.388 s

Con algoritmos más complejos puede suceder que el compilador no pueda optimizar contenedores o iteradores complejos tan fácilmente.

¿Por qué este código...

```
// order1.cpp

#define LOOPS 1000000
#define SIZE 1000

int a[SIZE] = {17, 4, 75, 73, 81, 92, 95, 5, 70, 96, 98, 67, 38, 40, 75, 35, 47, ... , 25};

int main()
{
    int c = 0;
    for (int l = 0; l < LOOPS; ++l)
    {
        for (int i = 0; i < SIZE; ++i)
        {
            if (a[i] < 50)
            {
                ++c;
            }
        }
    }
    return 0;
}
```

```
g++ -o order1.exe order1.cpp
time ./order1.exe
```

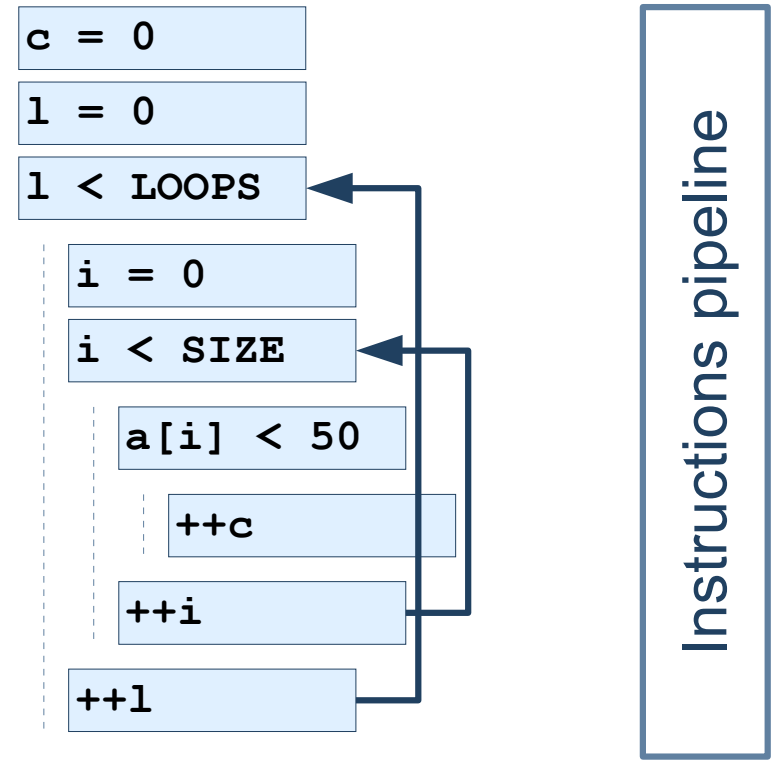
... es más lento que este otro?

```
// order2.cpp

#define LOOPS 1000000
#define SIZE 1000

int a[SIZE] = {0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, ... , 100};

int main()
{
    int c = 0;
    for (int l = 0; l < LOOPS; ++l)
    {
        for (int i = 0; i < SIZE; ++i)
        {
            if (a[i] < 50)
            {
                ++c;
            }
        }
    }
    return 0;
}
```

```
g++ -o order2.exe order2.cpp
time ./order2.exe
```

Branch prediction

```
int c = 0;
for (int l = 0; l < LOOPS; ++l)
{
    for (int i = 0; i < SIZE; ++i)
    {
        if (a[i] < 50)
        {
            ++c;
        }
    }
}
```



El procesador utiliza resultados anteriores para predecir que instrucción es la que sigue. Entonces carga la instrucción en el **pipeline de instrucciones** para tenerla lista para ejecutar. Lo que hará que el código se ejecute más rápido [Mcke15].

Si los datos están desordenados el procesador no puede predecir si entra o no al **if**. Tiene que actualizar el pipeline de instrucciónes muchas veces.

Al trabajar con los datos ordenados el procesador trabaja con el mismo pipeline casi todo el tiempo.

¿Preguntas?

miguelvargas@cimat.mx

Referencias

[Drep07] U. Drepper. What Every Programmer Should Know About Memory. Red Hat. 2007.

[Fog14] A. Fog. Optimizing software in C++. An optimization guide for Windows, Linux and Mac platforms. Copenhagen University College of Engineering. 2011.

[Greg13] B. Gregg. Systems Performance: Enterprise and the Cloud. Prentice Hall. 2013.

[Henn07 J. L. Hennessy, D. A. Patterson. Computer architecture. A quantitative approach. Fourth Edition. Elsevier. 2007.

[Mcke15] P. E. McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It? Linux Technology Center. IBM Beaverton. 2015.

[Prok99] H. Prokop. Cache-Oblivious Algorithms. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. 1999.

[Wulf95] W. A. Wulf , S. A. Mckee. Hitting the Memory Wall: Implications of the Obvious. Computer Architecture News, 23(1):20-24, March 1995.