MODERN ARCHITECTURES.

```
<p> Parallel Programming </p>
```

HELLO! We are...

- PERCA QUISPE, Joel Cristian
- VALDIVIA QUISPE, Eduardo Felipe
- VILCHEZ MOLINA, Misael Svante
- ESPINOZA PEÑALOZA, Edgar Alfonso
- BARRIOS CORNEJO, Selena

01 MEMORY HIERARCHY

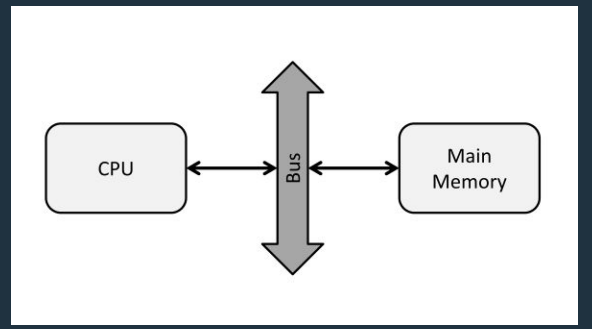
 Arquitectura Von Neumann

VON NEUMANN BOTTLENECK

 Velocidad de cómputo de la CPU

Discrepancia

 velocidad de la memoria principal (DRAM)



Ejemplo

CPU

- 8 cores
- 3GHz
- 1 Flop (16-double precisión) > 384 GFlop/s

DRAM

- 51.2 GB/s

```
double dotp = 0.0;
for (int i = 0; i < n; i++)
    dotp += u[i] * v[i];
```

N = 2^30 ⟶ 2N = 2^31Flop > 2GFlop

2^31 x 8B = 16GB

- Comp: t1 = 2GFlop/384GFlop/S = 5.2ms
- Data: t2 = 16GB/51.2GB/s = 312.5ms
- Exec: max(t1,t2) = 312.5ms

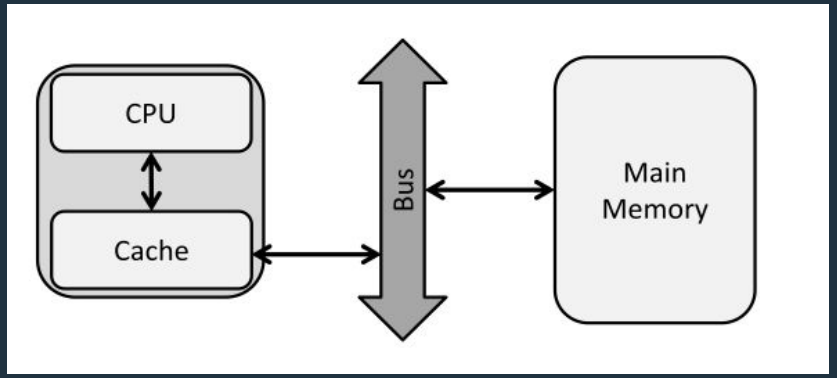
02 CACHE MEMORY

 Arquitectura Von Neumann

CACHE MEMORY

 El caché es una memoria rápida agregada entre la CPU y la memoria principal

 la capacidad es mucho menor



Ejemplo

CPU
- 8 cores
- 3GHz
- 1 Flop (16-double precisión) > 384 GFlop/s

DRAM
- 51.2 GB/s

CACHE
- 512 KB

W = U x V > N = 128

```
for (int i = 0; i < n ; i++)
    for (int j = 0; j < n; j++) {
        double dotp = 0.0;
        for (int k = 0; k < n; k++)
            dotp += U[i][k] * V[k][j];
        W[i][j] = dotp;
    }
```

128 x 128 x 8B = 128KB > 384KB

$2xN^3 = 2^{22}$

2^31 x 8B = 16GB

- Comp: t1 = 2^22Flop/384GFlop/S = 5.2us
- Data: t2 = 384KB/51.2GB/s = 7.5us
- Exec: t1 + t2 = 17.9us

02 CACHE ALGORITHMS

 Qué datos se almacenan en la memoria caché durante la ejecución de un programa

CACHE ALGORITHMS.

¿Qué datos cargamos de la memoria principal y en qué parte de la memoria caché los almacenamos?

Si el caché ya está lleno, ¿qué datos desalojamos?

económico

eficiente

Cache miss

Cache hit

Hit ratio

CACHE ALGORITHMS.

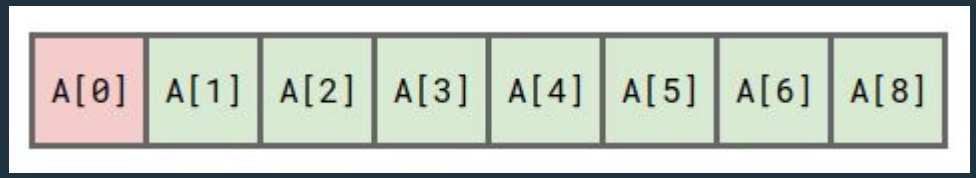
Localidad espacial

¿Qué datos cargamos de la memoria principal y en qué parte de la memoria caché los almacenamos?

ubicaciones de memoria contiguas

```
for (int i = 0; i < n; i++)
     maximum = max(a[i], maximum);
```

Cache line size = 64B



hit ratio= 87.5%

CACHE ALGORITHMS.

Localidad temporal

Si el caché ya está lleno, ¿qué datos desalojamos?

Desalojo del menos usado recientemente (LRU)

direct-mapped cache

RAM block

1

cache line

1

two-way set associative cache

RAM block

1

cache line

2

fully associative

RAM block

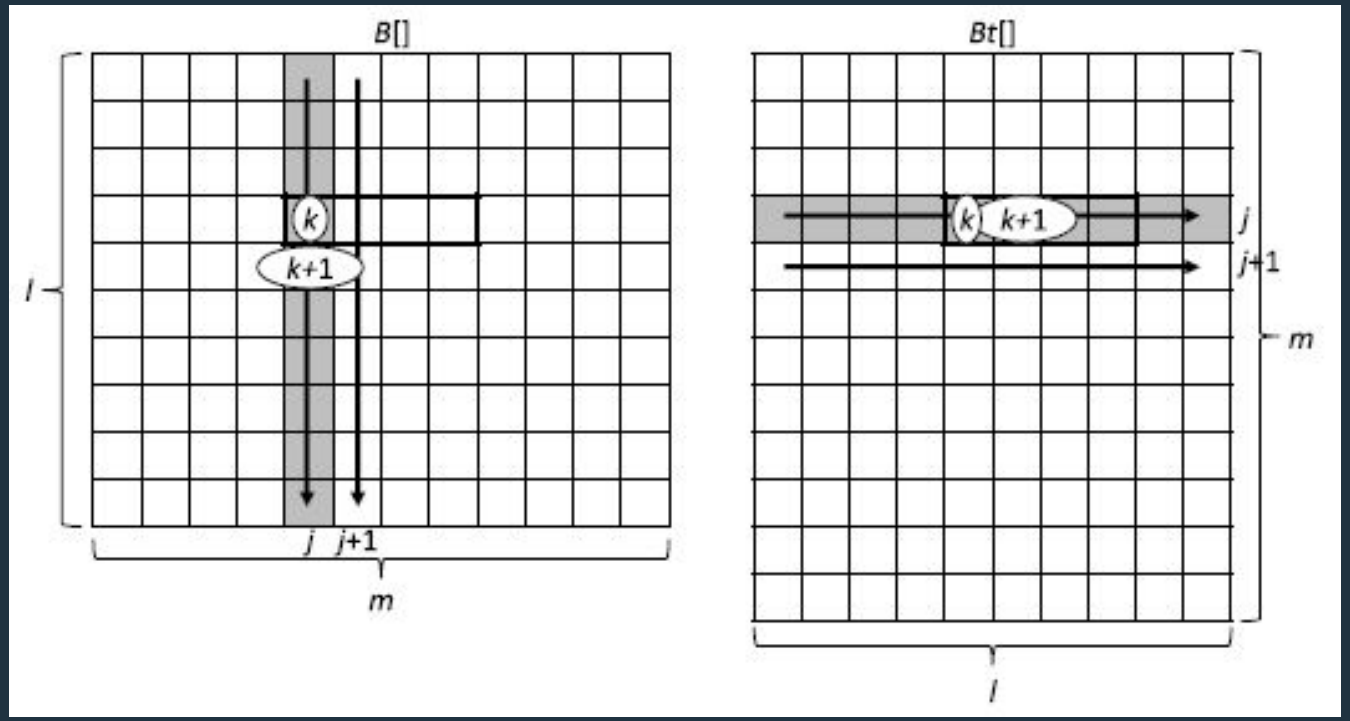
1

cache line

n

OPTIMIZING CACHE ACCESSES

```cpp
#include <iostream>
#include <cstdint>
#include <vector>
#include <chrono>
#include "include/hpc_helpers.hpp"


int main () {

    // matrix shapes
    const uint64_t m = 1 << 15;
    const uint64_t n = 1 << 15;
    const uint64_t l = 1 << 5;

    TIMERSTART(init)
    // sum_k A_ik * B_kj = sum_k A_ik * B^t_jk = C_ij
    std::vector<float> A (m*l, 0); // m x l
    std::vector<float> B (l*n, 0); // l x n
    std::vector<float> Bt(n*l, 0); // n x l
    std::vector<float> C (m*n, 0); // m x n
    TIMERSTOP(init)

    TIMERSTART(transpose_and_mult)
    TIMERSTART(transpose)
    #pragma omp parallel for collapse(2)
    for (uint64_t k = 0; k < l; k++)
        for (uint64_t j = 0; j < n; j++)
            Bt[j*l+k] = B[k*n+j];
    TIMERSTOP(transpose)

    TIMERSTART(transpose_mult)
    #pragma omp parallel for collapse(2)
    for (uint64_t i = 0; i < m; i++)
        for (uint64_t j = 0; j < n; j++) {
            float accum = 0;
            for (uint64_t k = 0; k < l; k++)
                accum += A[i*l+k]*Bt[j*l+k];
            C[i*n+j] = accum;
        }

    TIMERSTOP(transpose_mult)
    TIMERSTOP(transpose_and_mult)

    TIMERSTART(naive_mult)
    #pragma omp parallel for collapse(2)
    for (uint64_t i = 0; i < m; i++)
        for (uint64_t j = 0; j < n; j++) {
            float accum = 0;
            for (uint64_t k = 0; k < l; k++)
                accum += A[i*l+k]*B[k*n+j];
        C[i*n+j] = accum;
    }

    TIMERSTOP(naive_mult)

    return 0;
}
```



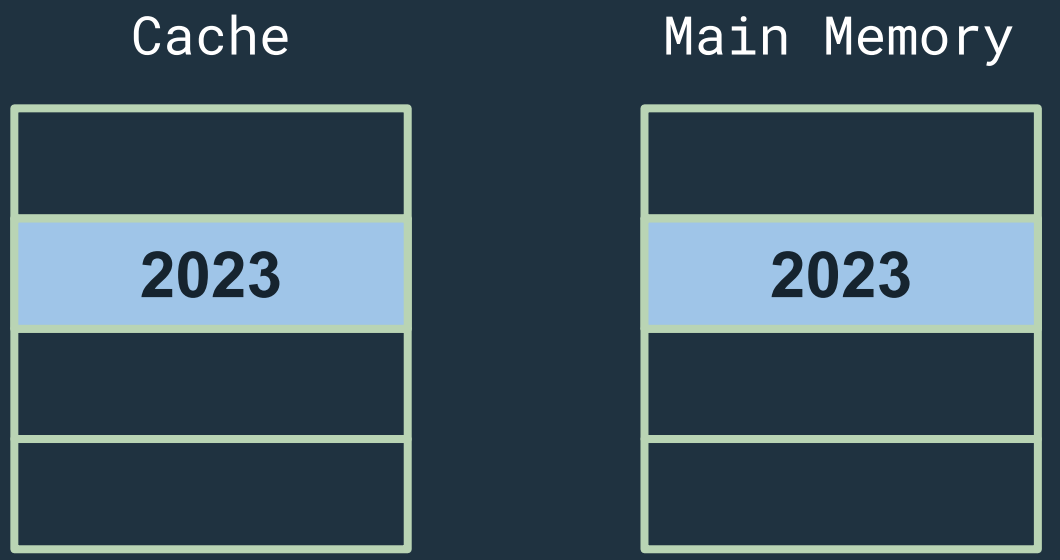

```
# elapsed time (naive_mult): 203.02s
```

```
# elapsed time (transpose_mult): 149.95s
```

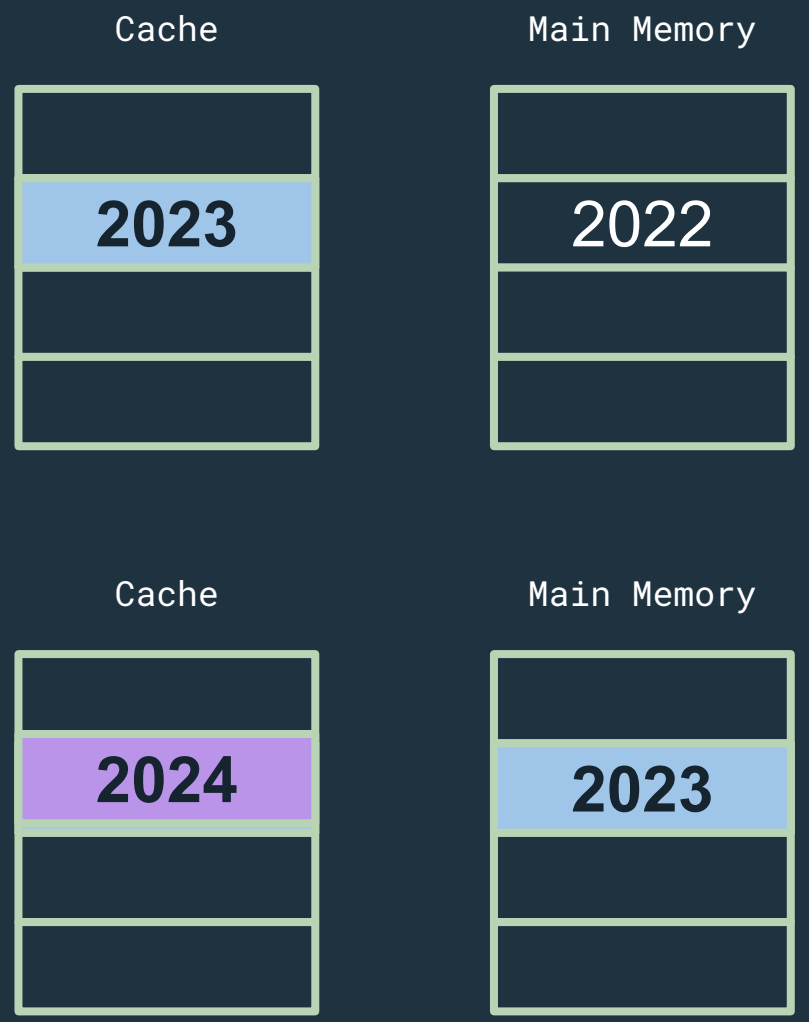
CACHE COHERENCE

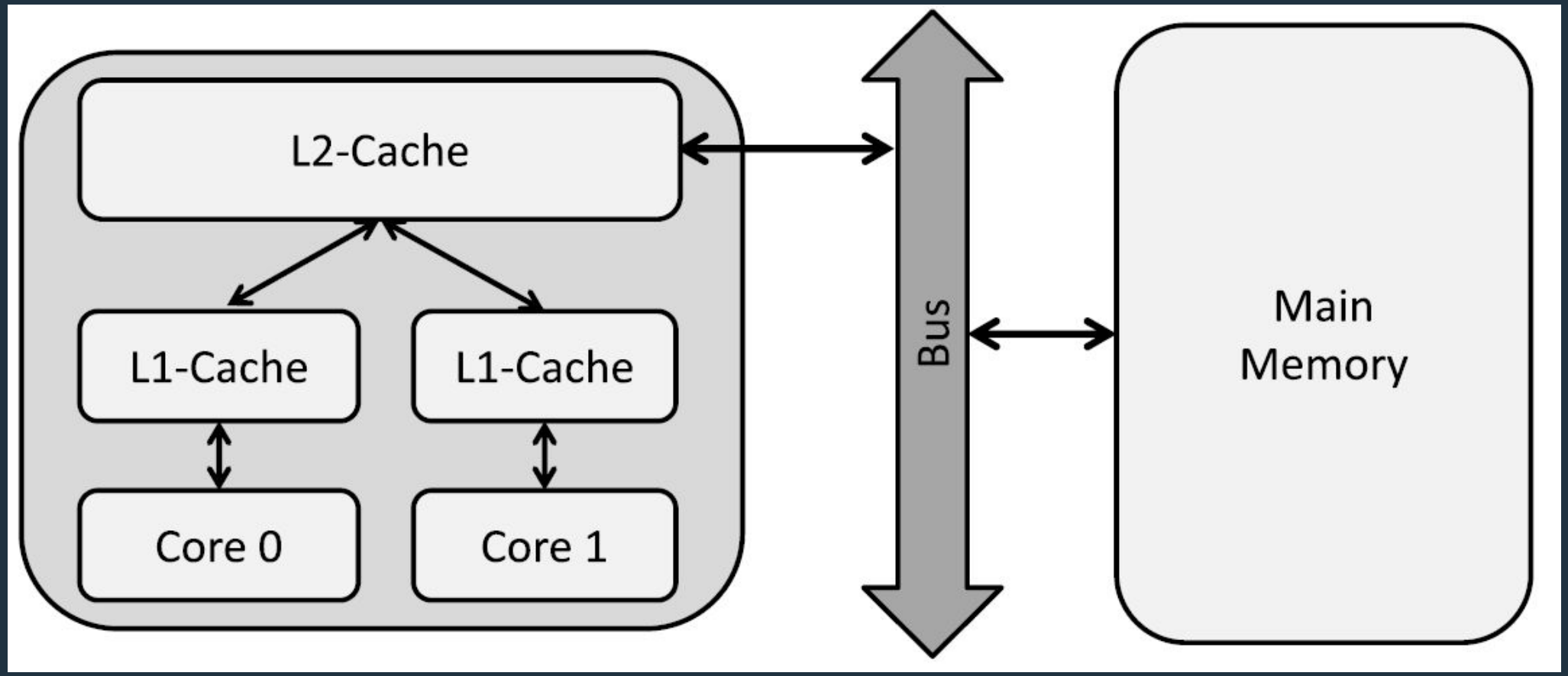
Si solo modificamos el valor almacenado en la caché, se produciría una **incoherencia** entre la copia **almacenada en la caché** y el valor **original en la memoria principal**

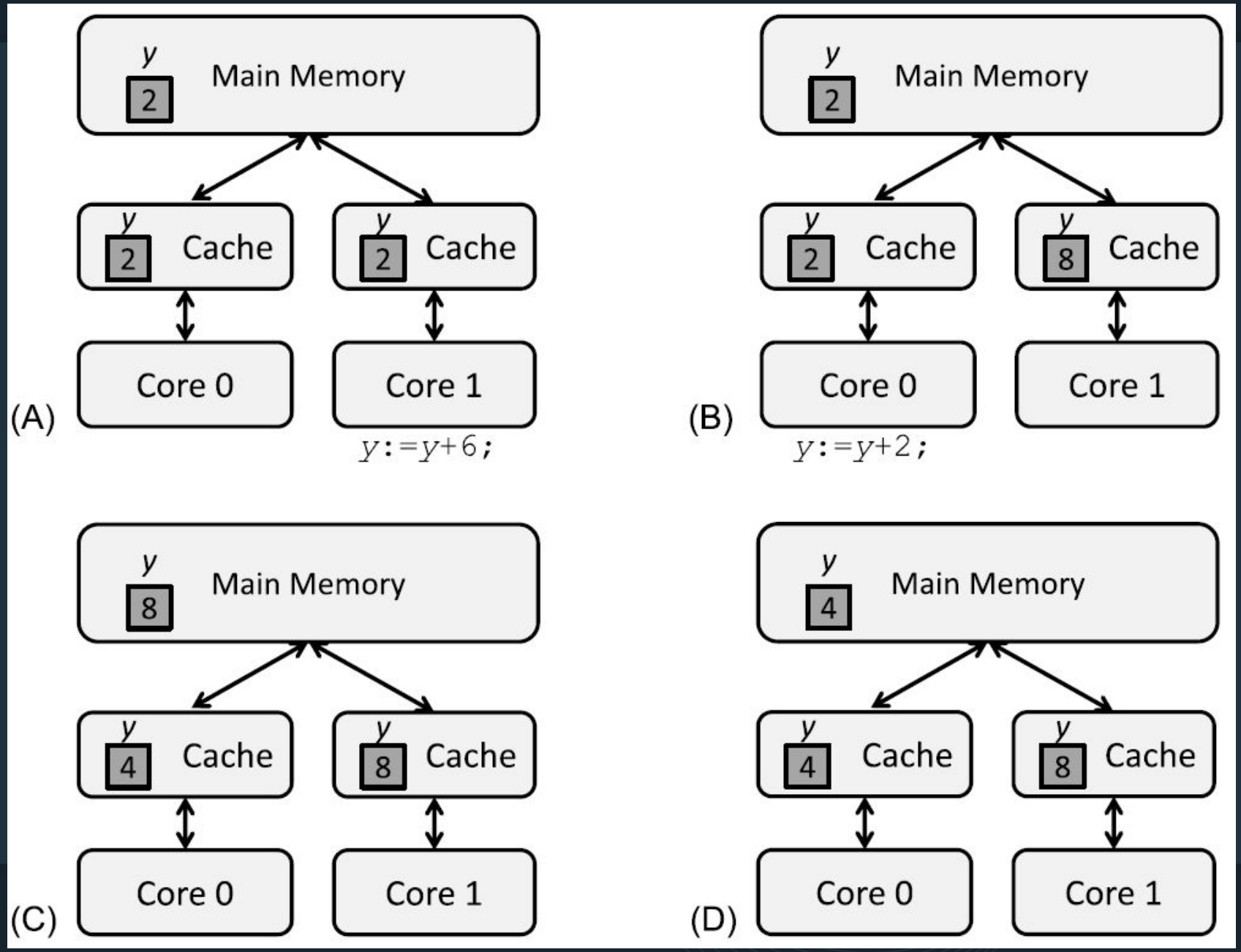
En el write-through, los datos se actualizan simultáneamente en la caché y en la memoria.



La dirección de la memoria principal no se modifica inmediatamente. En su lugar, sólo se modifica la línea de caché asociada y se marca como dirty

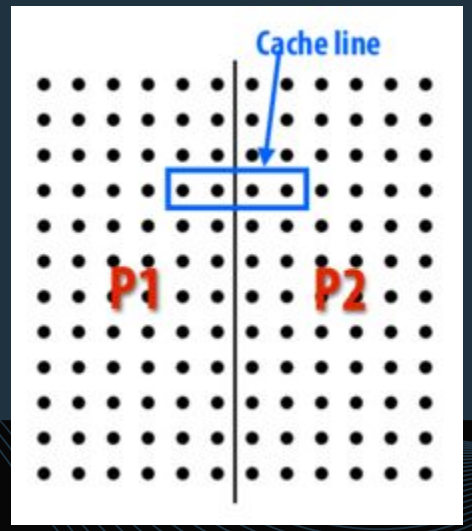




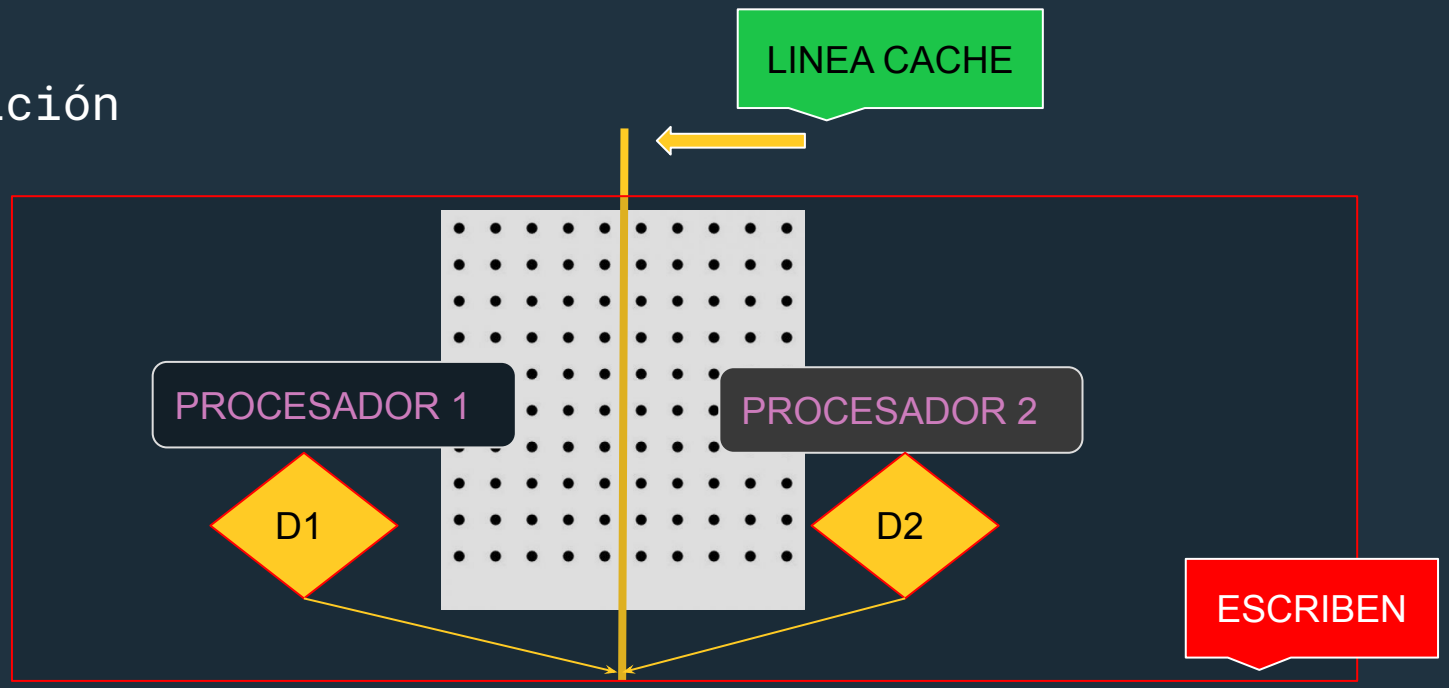


05 FALSE SHARING

 ¿Cuándo ocurre?



1. Condición



2. La línea de caché hace ping-pong entre cachés de procesos de escritura, generando cantidades significativas de comunicación debido al **protocolo de coherencia**.

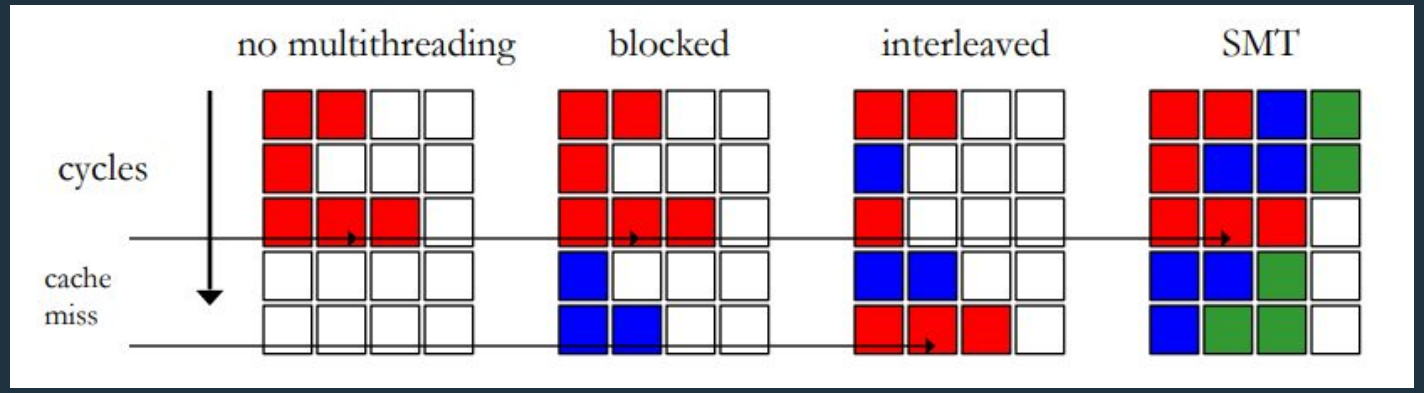
Si ocurre con frecuencia:

1. Tráfico de interconexión **aumenta**
2. El rendimiento y la escalabilidad de una aplicación OpenMP **sufren** significativamente.

```
struct pack_t {
    uint64_t ying;
    uint64_t yang;
    pack_t() : ying(0), yang(0) {}
};
```

06 SIMULTANEOUS MULTITHREADING AND PREFETCHING

 SMT, la idea básica sería un procesador superescalar obtiene y emite instrucciones de diferentes hilos/procesos simultáneamente



Ventajas:

+ Puede manejar no solo latencias prolongadas y burbujas de flujo de trabajo, sino también espacios de problemas no utilizados
+ Rendimiento completo en modo de un solo hilo
- El hardware más complejo de todos los esquemas de subprocesos múltiples

PREFETCHING

Idea: Obtener los datos antes de que el programa los necesite

¿Por qué?

1. La latencia de la memoria es alta. Si podemos realizar una captación previa con precisión y con la antelación suficiente, podemos reducir/eliminar esa latencia.
2. Puede eliminar errores de caché obligatorios

¿Puede eliminar todos los errores de caché? ¿Capacidad, conflicto?

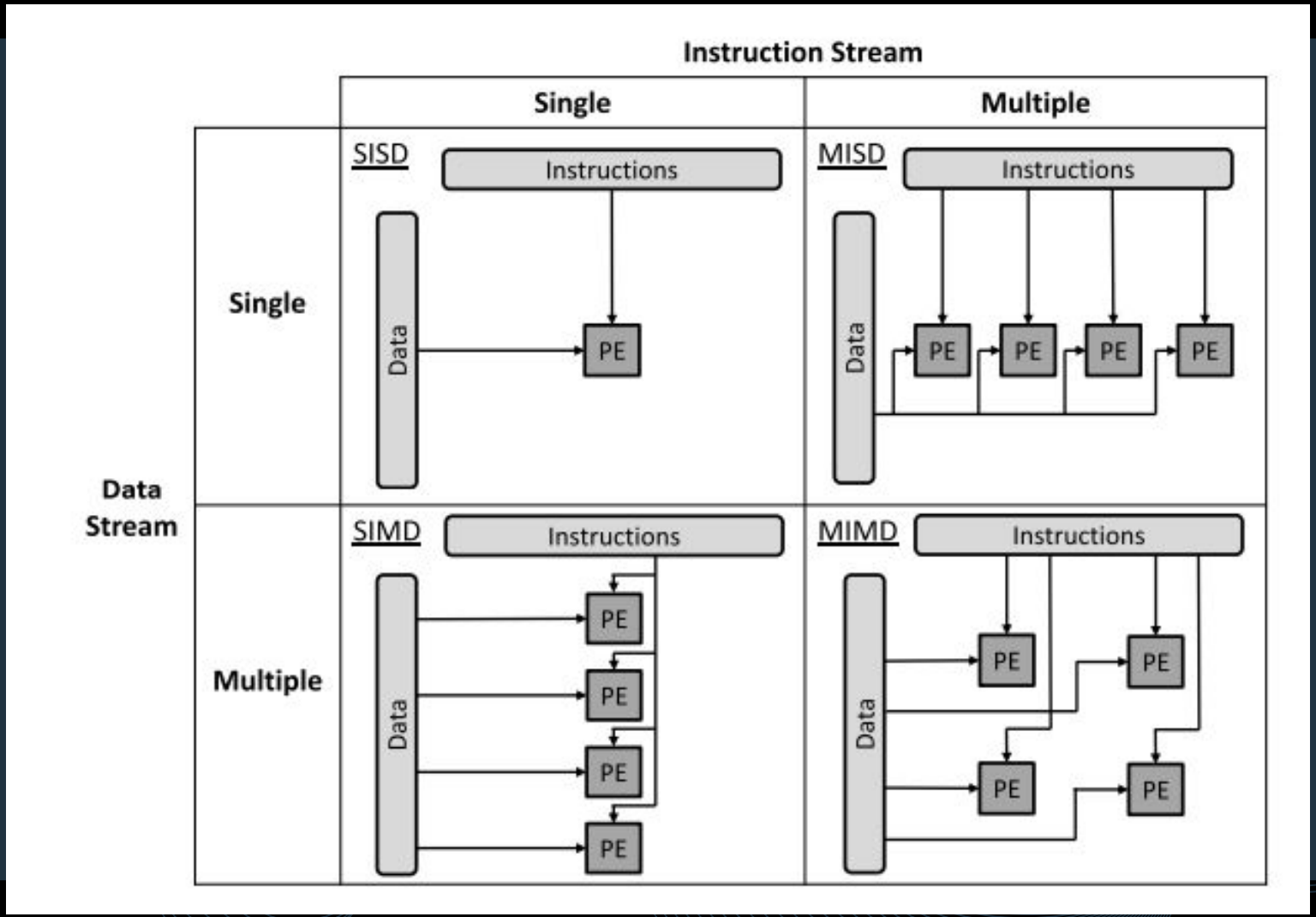
1. Implica predecir qué dirección será necesaria en el futuro
2. Funciona si los programas tienen patrones de pérdida de dirección predecibles

Por lo tanto:

1. Funciona bien para patrones regulares de acceso a la memoria
2. La obtención previa de patrones de acceso irregulares es difícil, imprecisa y requiere mucho hardware

07 Flynn's Taxonomy

 Clasificación de los diferentes tipos de paralelismo



Características que explotan el paralelismo

Multiples Núcleos

Uso popular de MIMD

Unidades Vectoriales

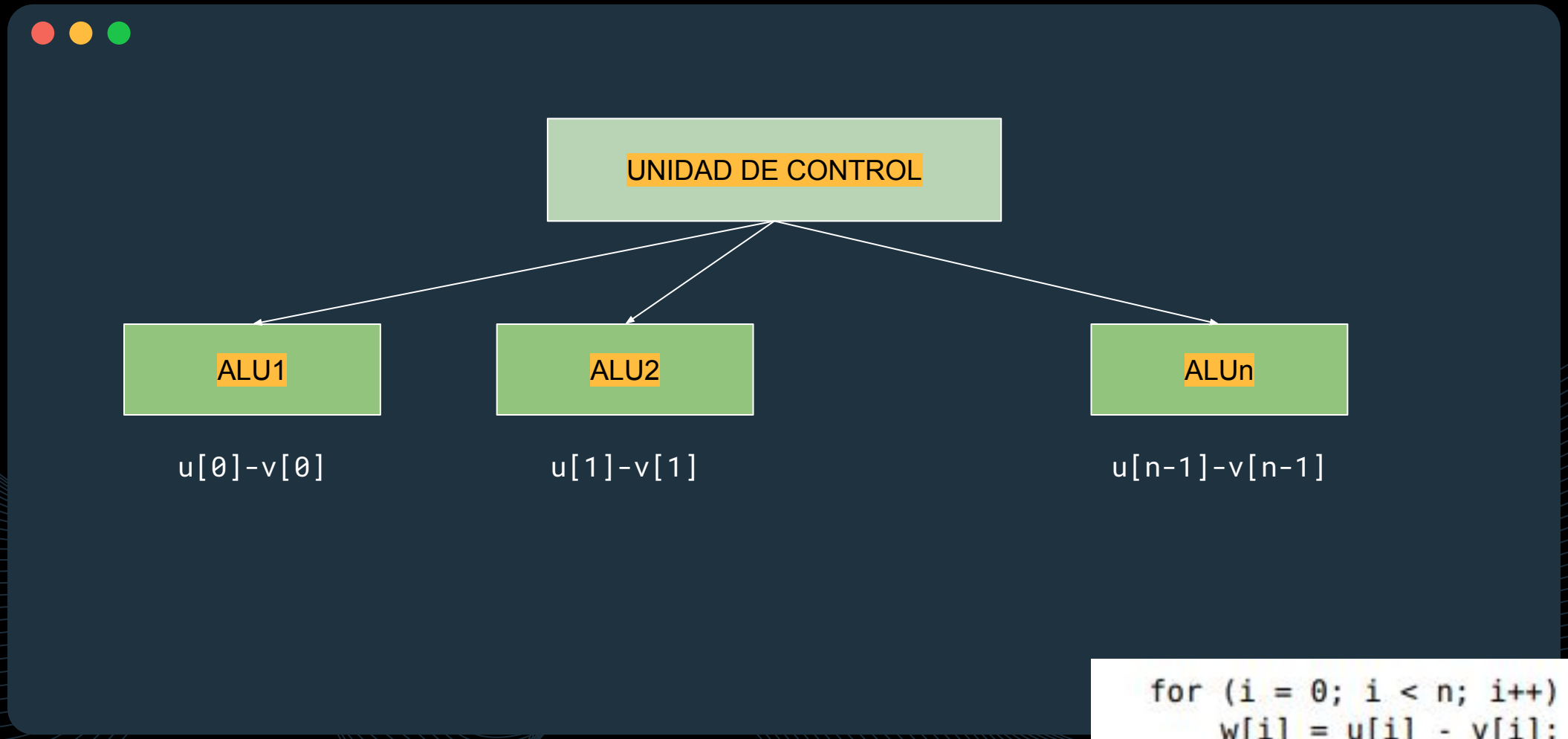
Explotación del paralelismo a nivel de datos

Paralelismo a nivel de instrucciones

Explotación de ILP mediante pipelining y la ejecución superescalar de instrucciones

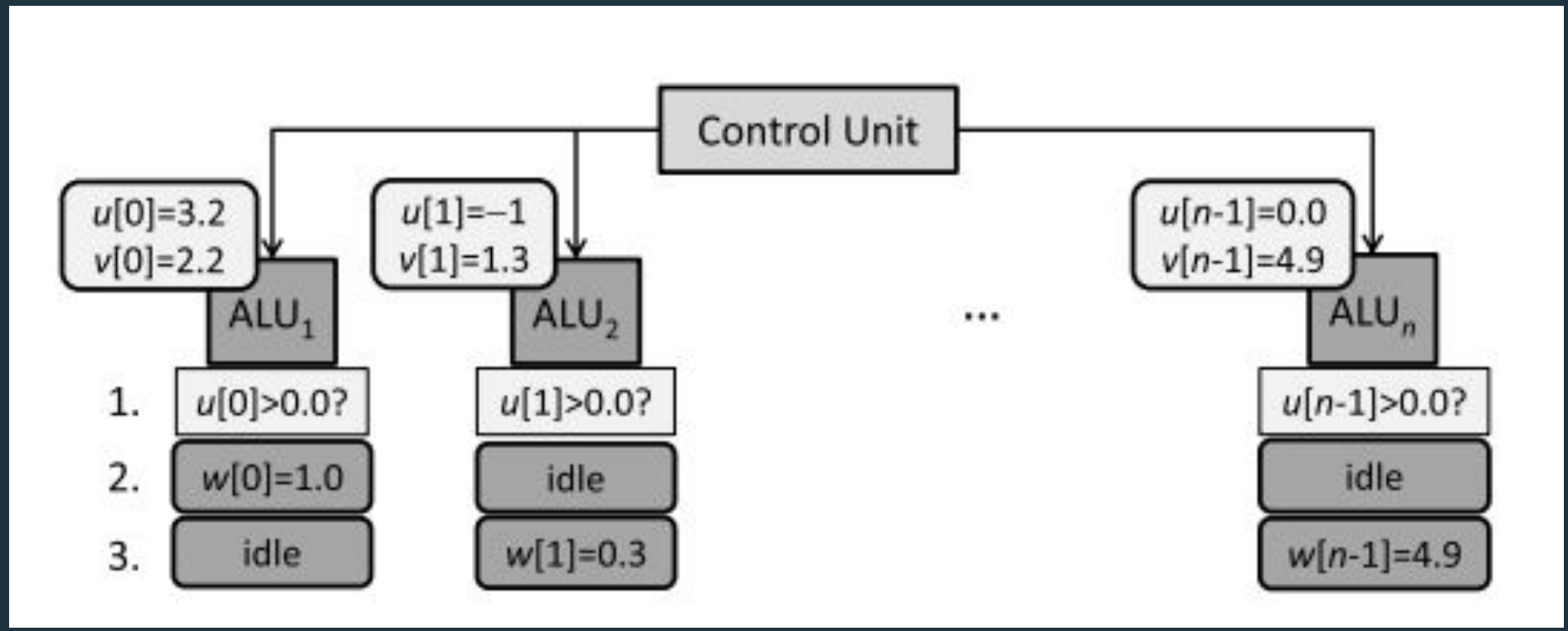
08 SIMD Concept

Single Instruction Multiple Data




```
for (i = 0; i < n; i++)
    w[i] = u[i] - v[i];
```

```
for (i = 0; i < n; i++)
    if (u[i] > 0)
        w[i] = u[i] - v[i];
    else
        w[i] = u[i] + v[i];
```



10 VECTORIZATION ON COMMON MICROPROCESSORS

SOPORTE DE OPERACIONES SIMD.

1997

MMX (Multi Media Extension - Intel) y 3DNow! - AMD

Registros de 64 bits que pueden realizar aritmética con: 2 enteros de 32 bits, 4 enteros de 16 bits o 4 enteros de 8 bits.

1999

SSE (Streaming SIMD Extensions)

Registros de 128 bits que pueden realizar aritmética tanto con enteros como con números punto flotantes.

2011

AVX (Advanced Vector Extensions): Registros de 256 bits.

2015

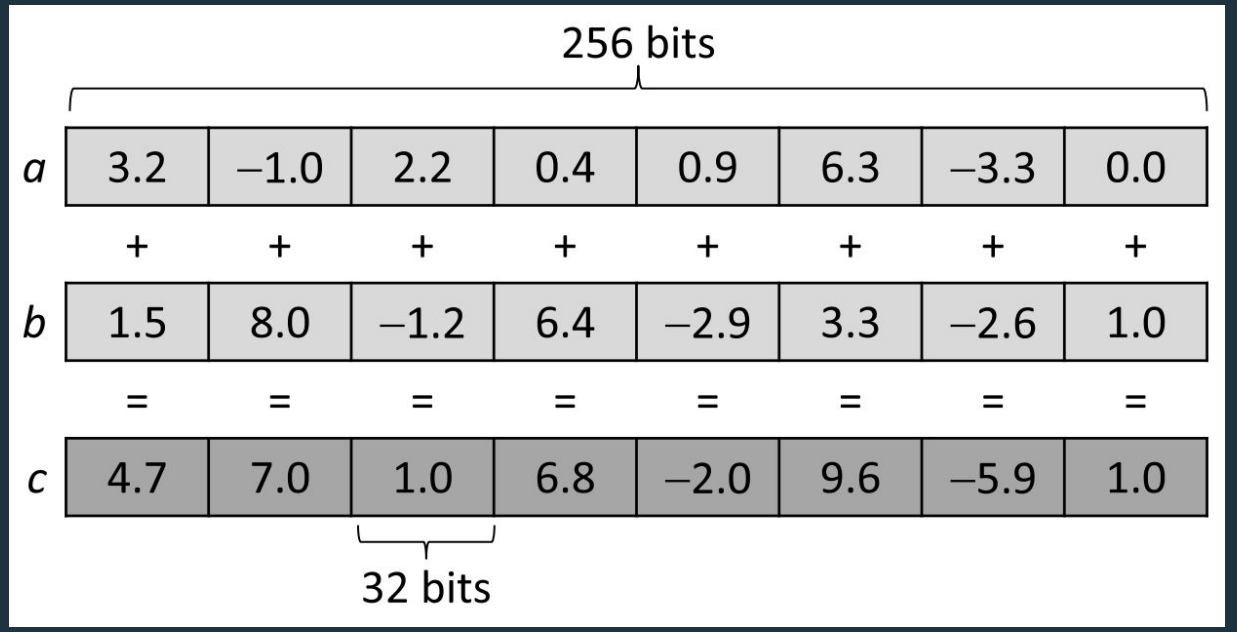
AVX-512: Registros de 512 bits.

INTRINSICS.

Intrinsics

Definiciones de tipos de datos y funciones implementadas en lenguaje ensamblador.

```
__m256 a,b,c;
...
c = _mm256_add_ps(a,b);
```



INTRINSICS: PRODUCTO MATRICIAL

```
for (uint64_t i = 0; i < M; i++)
    for (uint64_t j = 0; j < N; j++) {
        float accum = float(0);
        for (uint64_t k = 0; k < L; k++)
            accum += A[i*L+k]*B[j*L+k];
        C[i*N+j] = accum;
   }
```

Clásico

```
for (uint64_t i = 0; i < M; i++)
    for (uint64_t j = 0; j < N; j++) {

        __m256 X = _mm256_setzero_ps();
        for (uint64_t k = 0; k < L; k += 8) {
            const __m256 AV = _mm256_load_ps(A+i*L+k);
            const __m256 BV = _mm256_load_ps(B+j*L+k);
            X = _mm256_fmadd_ps(AV,BV,X);
        }

        C[i*N+j] = hsum_avx(X);
   }
```

Vectorizado

INTRINSICS: PRODUCTO MATRICIAL

AV	*A*[*i***L*]	*A*[*i***L*+1]	*A*[*i***L*+2]	*A*[*i***L*+3]	*A*[*i***L*+4]	*A*[*i***L*+5]	*A*[*i***L*+6]	*A*[*i***L*+7]
	*	*	*	*	*	*	*	*
BV	*B*[*j***L*]	*B*[*j***L*+1]	*B*[*j***L*+2]	*B*[*j***L*+3]	*B*[*j***L*+4]	*B*[*j***L*+5]	*B*[*j***L*+6]	*B*[*j***L*+7]
	+	+	+	+	+	+	+	+
X	*X*[0]	*X*[1]	*X*[2]	*X*[3]	*X*[4]	*X*[5]	*X*[6]	*X*[7]
	=	=	=	=	=	=	=	=
X	*X*[0]	*X*[1]	*X*[2]	*X*[3]	*X*[4]	*X*[5]	*X*[6]	*X*[7]

Clásico: 12.2992s

Vectorizado: 2.133s

11 AoS AND SoA

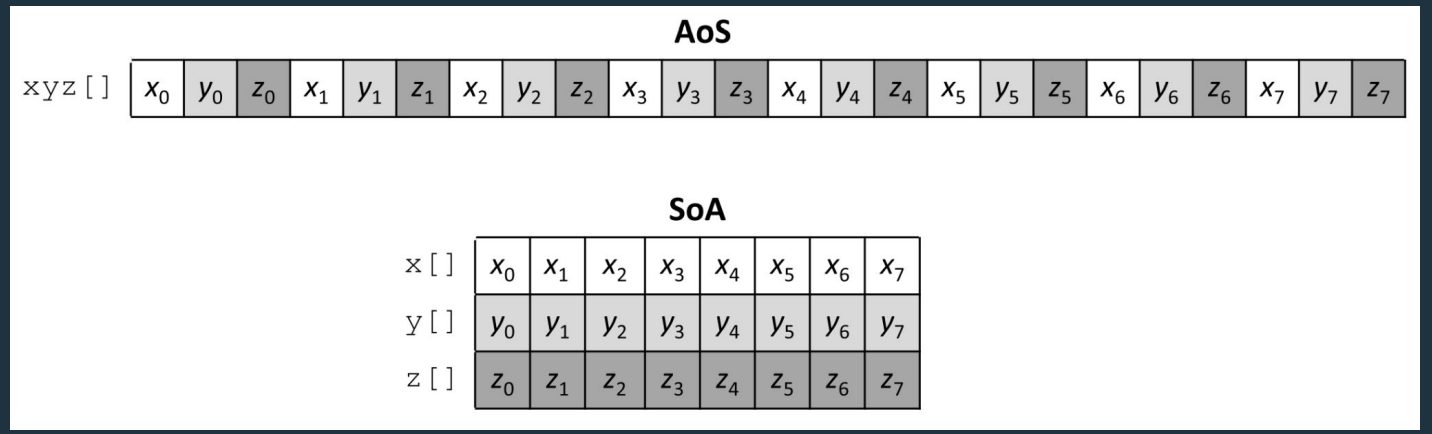
AoS y SoA.

AoS (Array of Structures)

Los registros se almacenan de forma consecutiva en un solo array.

SoA (Structure of Arrays)

Utiliza un array por dimensión.



AoS y SoA: NORMALIZACIÓN

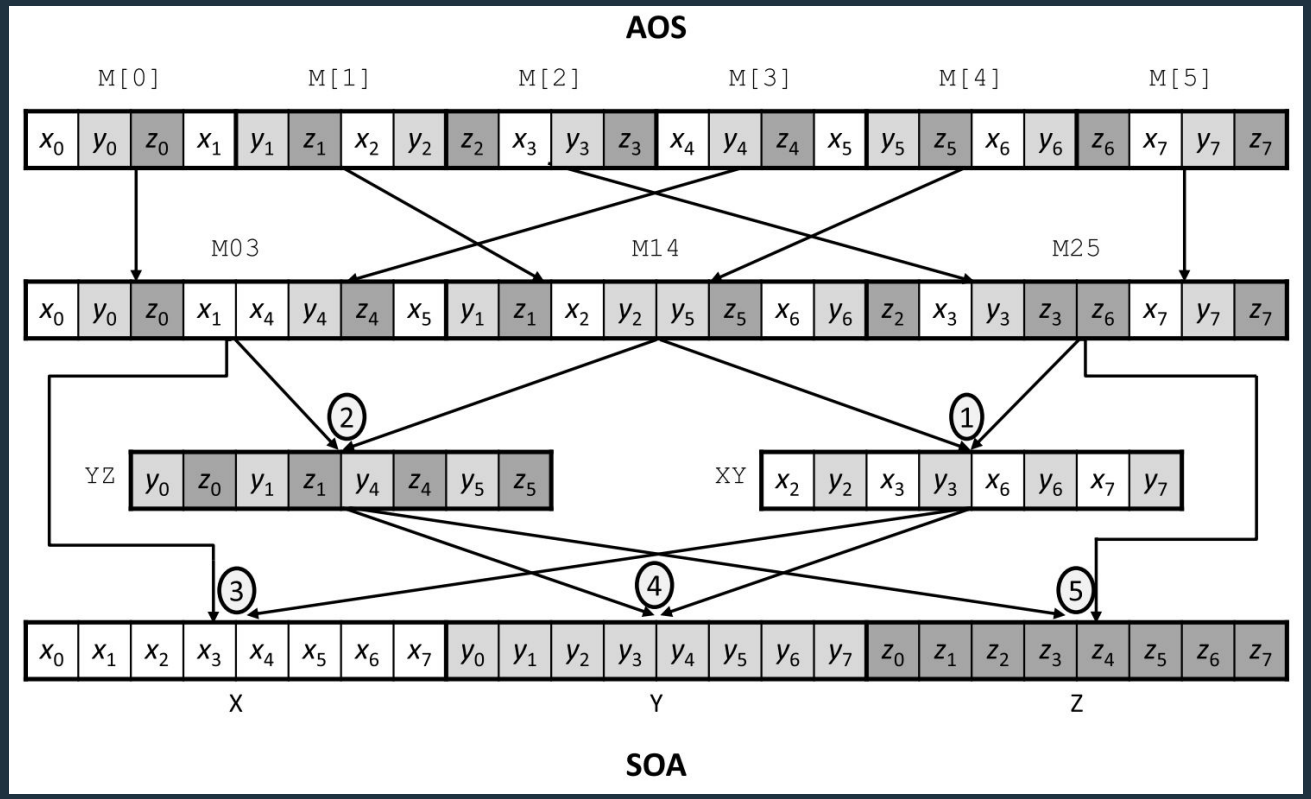
```
void plain_aos_norm(float * xyz, uint64_t length) {

    for (uint64_t i = 0; i < 3*length; i += 3) {
        const float x = xyz[i+0];
        const float y = xyz[i+1];
        const float z = xyz[i+2];

        float irho = 1.0f/std::sqrt(x*x+y*y+z*z);

        xyz[i+0] *= irho;
        xyz[i+1] *= irho;
        xyz[i+2] *= irho;
    }
}
```

AoS

```
void avx_soa_norm(float * x, float * y, float * z,
                  uint64_t length) {

    for (uint64_t i = 0; i < length; i += 8) {

        // aligned loads
        __m256 X = _mm256_load_ps(x+i);
        __m256 Y = _mm256_load_ps(y+i);
        __m256 Z = _mm256_load_ps(z+i);

        // R <- X*X+Y*Y+Z*Z
         __m256 R = _mm256_fmadd_ps(X, X,
                    _mm256_fmadd_ps(Y, Y,
                    _mm256_mul_ps  (Z, Z)));

        // R <- 1/sqrt(R)
        R = _mm256_rsqrt_ps(R);

        // aligned stores
        _mm256_store_ps(x+i, _mm256_mul_ps(X, R));
        _mm256_store_ps(y+i, _mm256_mul_ps(Y, R));
        _mm256_store_ps(z+i, _mm256_mul_ps(Z, R));
    }
}
```

SoA

AoS y SoA: TRANSPOSICIÓN



THANK YOU!

Do you have any questions?