Programación Paralela y Distribuida
Clase 02 - *Background (2)*

John Sanabria - john.sanabria@correounivalle.edu.co

Agenda

- Reconociendo características del computador
  - Cache
  - Memoria virtual
- Hilos (en Python) adecuados para llevar a cabo tareas intensivas en E/S
- Taxonomía de Flynn

Modelos de máquina

- Para escribir programas eficientes es necesario conocer algunos detalles del hardware
- El hardware tiene sus particularidades pero en general ofrecen ciertas abstracciones como:
  - *cores*
  - unidades vectoriales
  - cache
  - *non-uniform memory*
- Hoy son comunes los *attached co-processors* (a.k.a. GPGPUs) que constituyen lo que se conoce como **computación heterogénea** (CPU + GPU)

Modelo de máquina

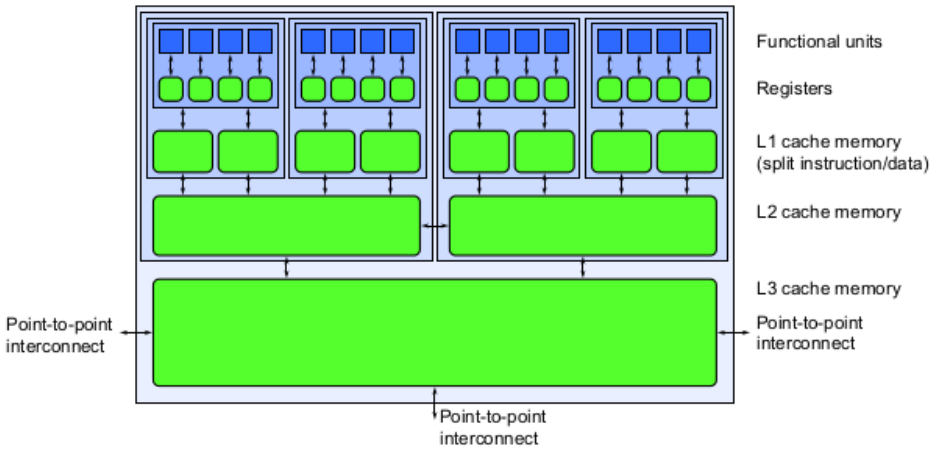


Figure 1: Procesador Multicore

- Procesador con múltiples núcleos
- Cada núcleo cuenta como varias unidades funcionales capaces de llevar a cabo operaciones aritméticas básicas
- Las unidades funcionales son elementos básicos de la computación más que el mismo núcleo en sí
- La cache permite mejorar el rendimiento de la memoria y la forma como se acceden a los datos que están en ella

Instruction Parallelism

- Un núcleo tiene múltiples unidades funcionales las cuales pueden llevar a cabo múltiples operaciones en paralelo
- Esta ejecución paralela se puede llevar a cabo de manera **implícita** o **explícita** a través de:
  - (I) **Ejecución superscalar de instrucciones**. El procesador identifica instrucciones que no son dependientes y las ejecuta en paralelo (ver código: `src/reorder_execution.py`)
  - (I) *Hardware Mulithreading*. Los *cores* son capaces de lanzar la ejecución de varios hilos en paralelo
    - **Multithreading simultáneo** en el cual se ejecutan múltiples flujos de instrucciones
    - ***Switch-on-event multithreading*** que permite el conmutar entre hilos cuando se identifica que estos hacen operaciones lentas (e.g. operaciones de E/S)
    - Ejemplo de código en Python de múltiples hilos
  - (E) **Instrucciones vectoriales** agrupan varias unidades funcionales para llevar a cabo operaciones sobre un conjunto pequeño de datos. (ver código: `simd-example.c`).

Jerarquía de memoria

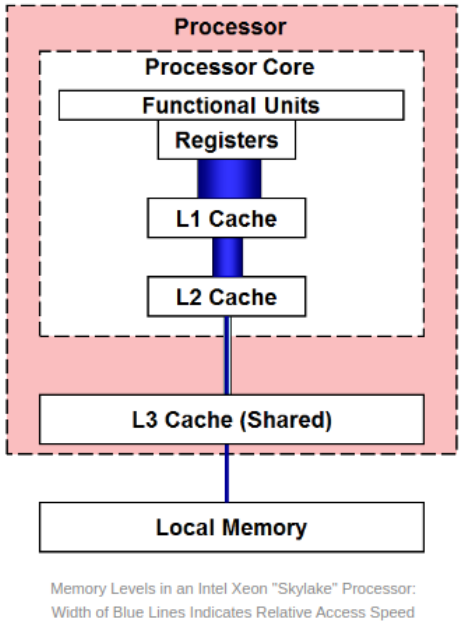


Figure 2: Localización Memoria Cache

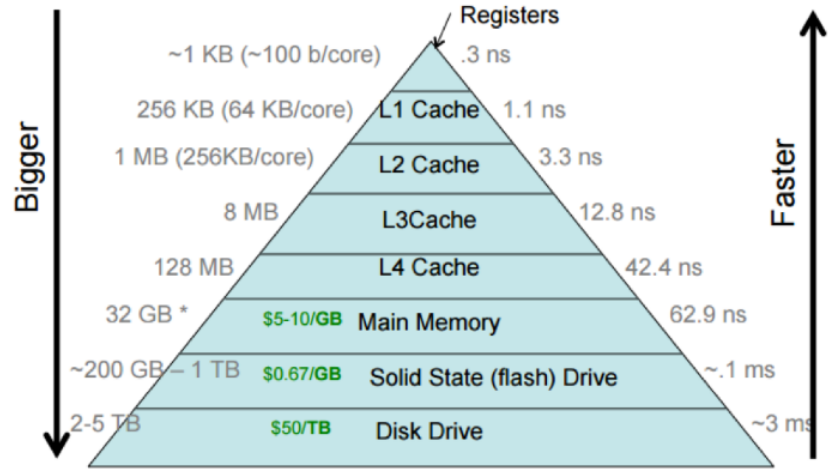


Figure 3: Velocidad Acceso a Memoria

Jerarquía de memoria

- Registros son las unidades de almacenamiento más rápidas pero más limitadas en capacidad
- La memoria cache es más rápida que la RAM y de mayor capacidad que los registros. Las memorias de cache se dividen en cache de datos y de instrucciones
  - Cuántos órdenes de magnitud diferencian unas memorias de las otras respecto a su capacidad de almacenamiento y velocidad?
- Es fácil encontrar memorias cache con capacidad de almacenamiento del orden de las decenas de megas

Cache y su almacenamiento

- Los bloques de almacenamiento del cache se conocen como *cache lines*. Su capacidad es del orden de 64 a 128 **bytes**
- Cuando se lee un dato de la RAM (e.g. una variable entera) este y sus zonas de memoria aledañas se almacenan en un *cache line* (aprovechar los accesos a RAM)

Operación de la cache, una representación visual (1)

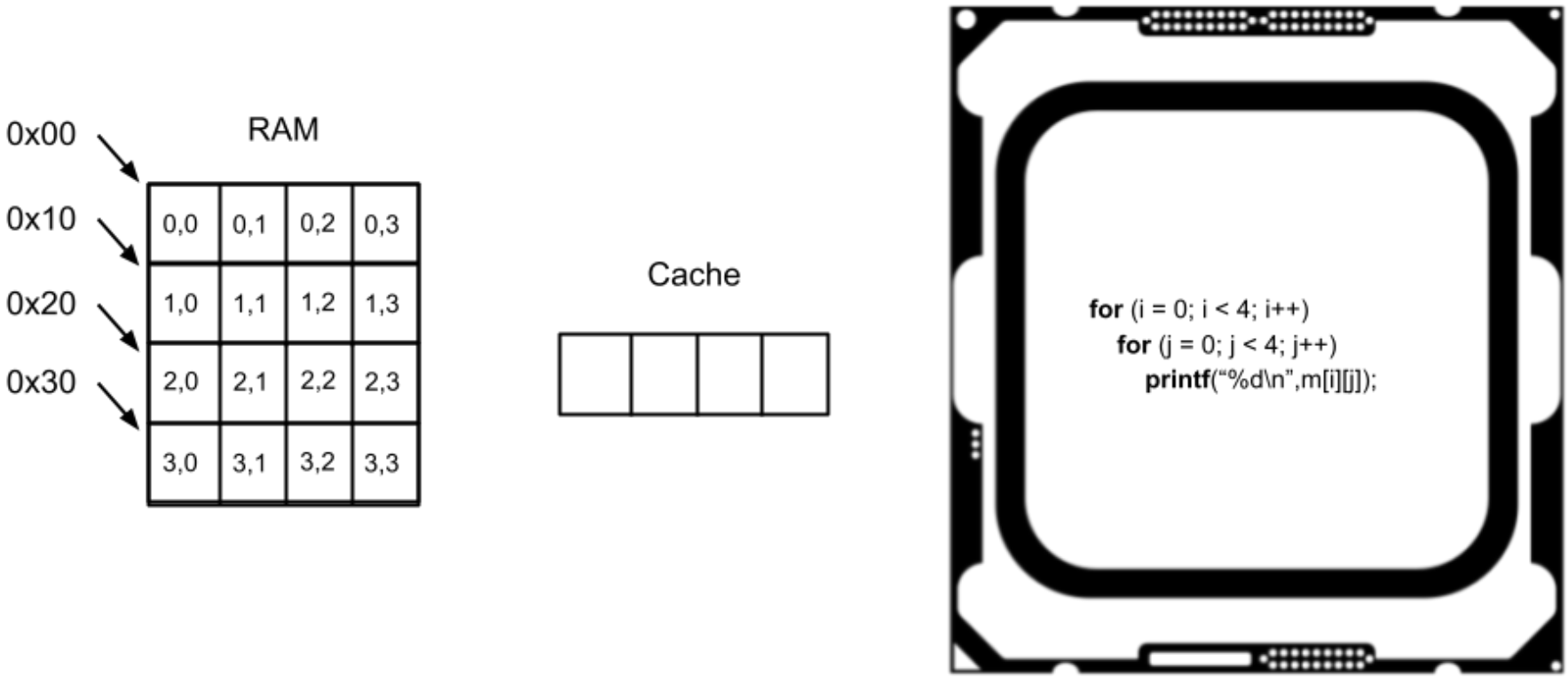


Figure 4: Esquema alto nivel: RAM, Cache, Instrucciones en CPU

Operación de la cache, una representación visual (2)

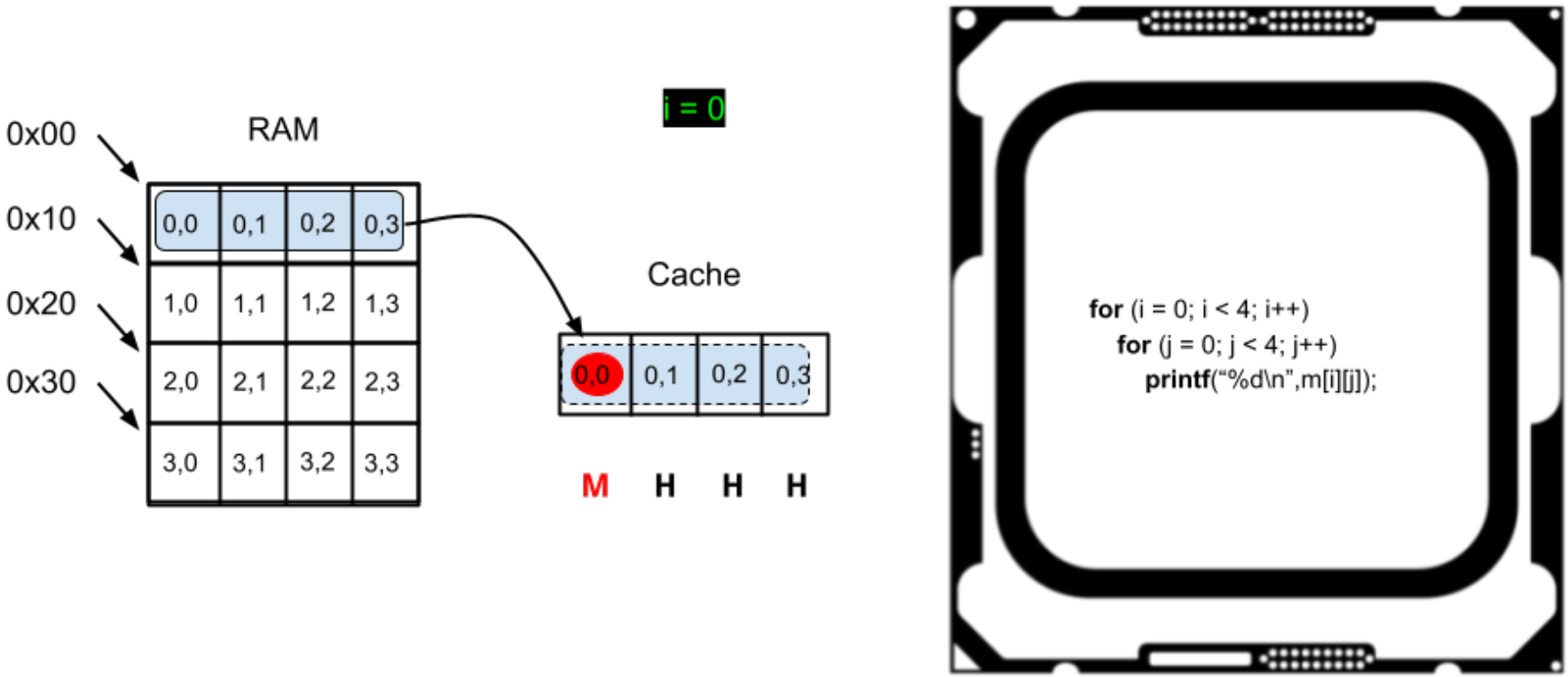


Figure 5: Accediendo a la fila **0**

Operación de la cache, una representación visual (3)

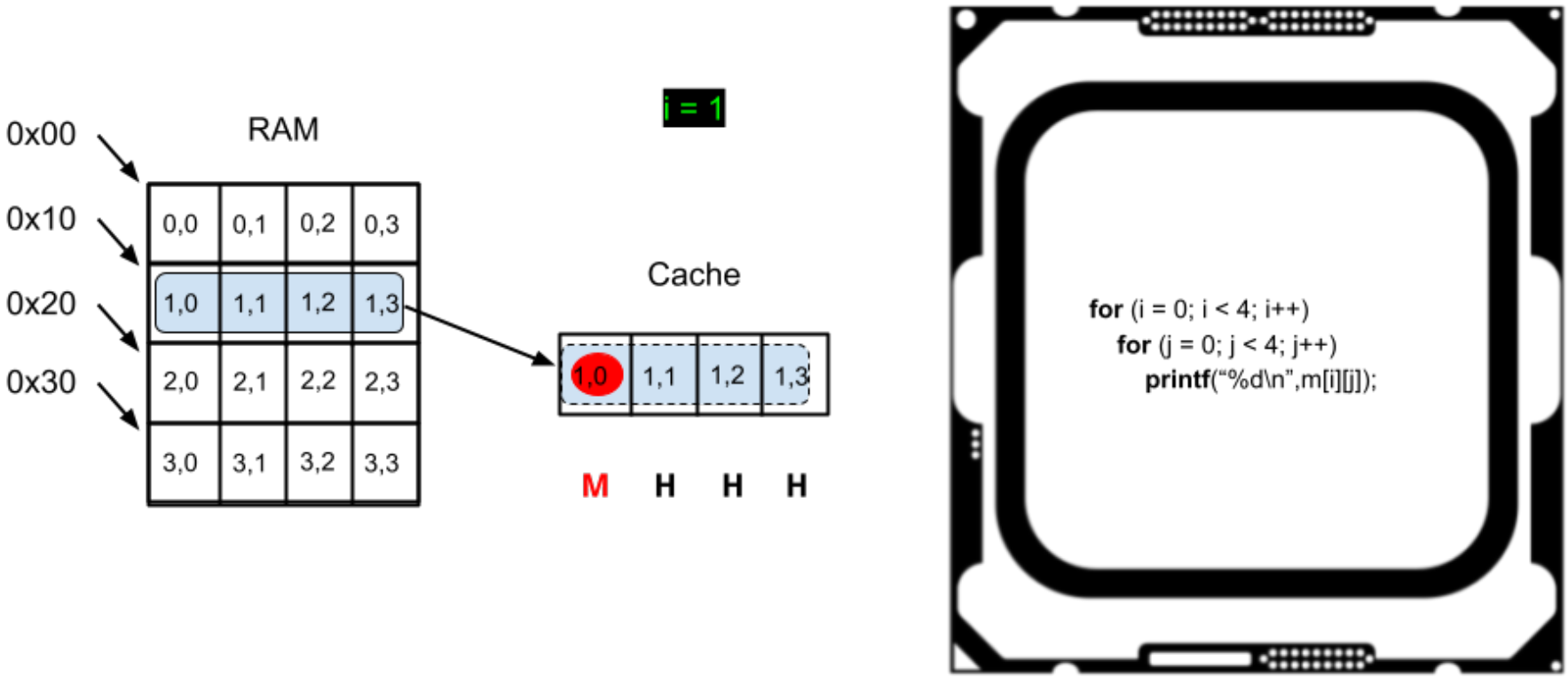


Figure 6: Accediendo a la fila **1**

Operación de la cache, una representación visual (4)

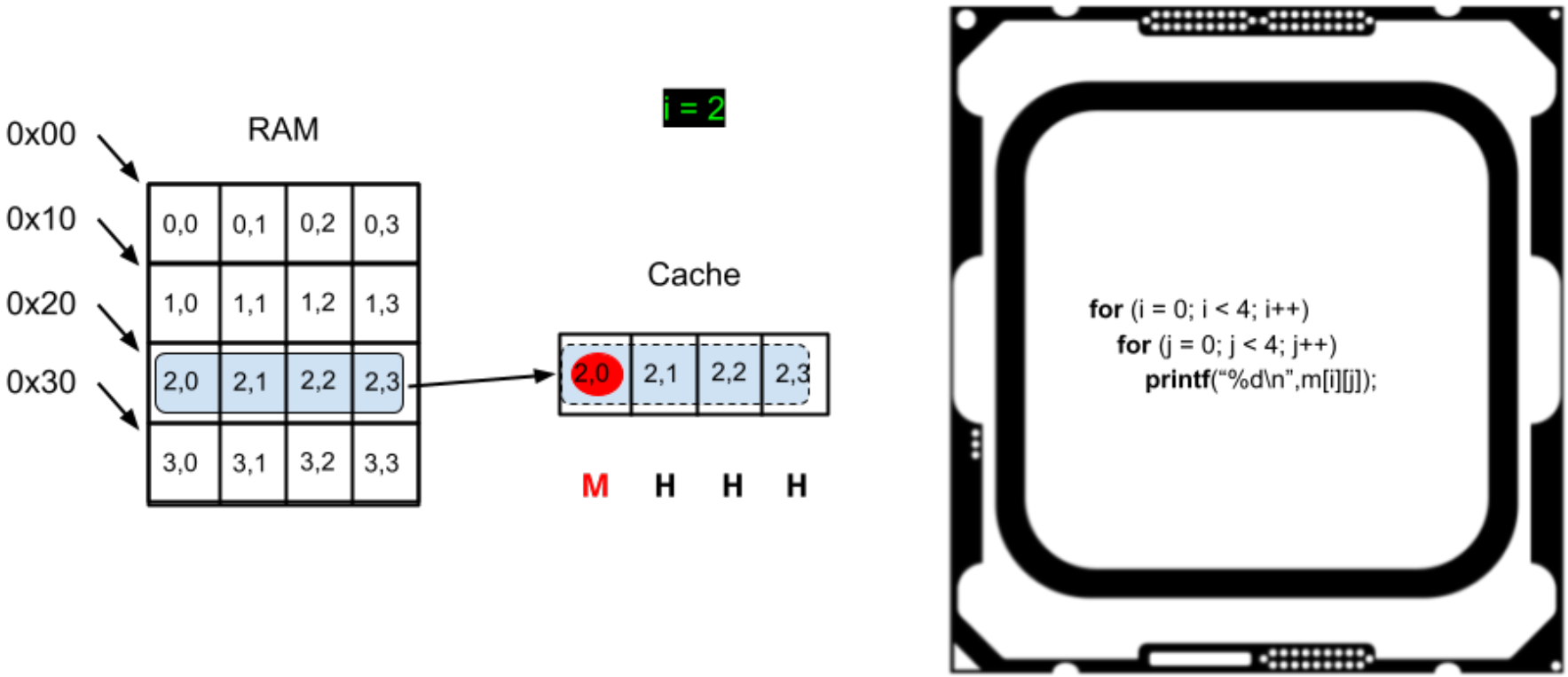
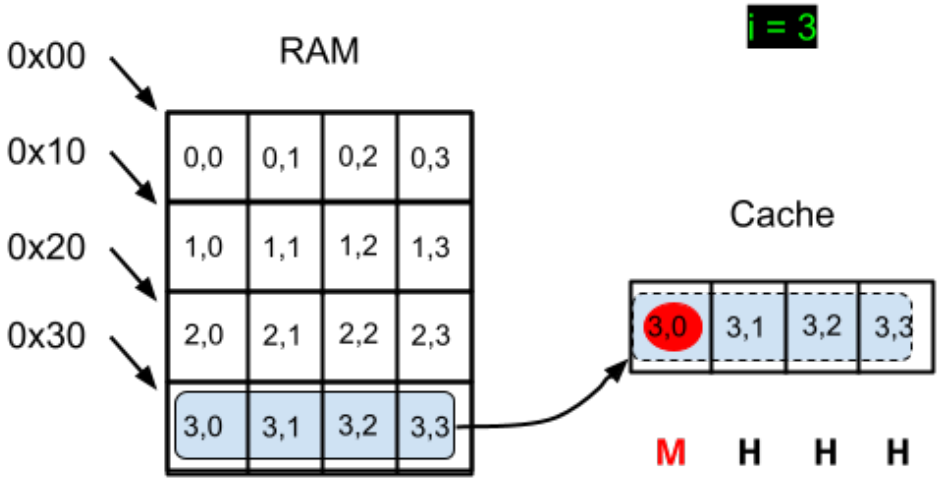


Figure 7: Accediendo a la fila **2**

Operación de la cache, una representación visual (5)



¿Cuántos *misses* y cuántos *hits* se contabilizan en este ejemplo?

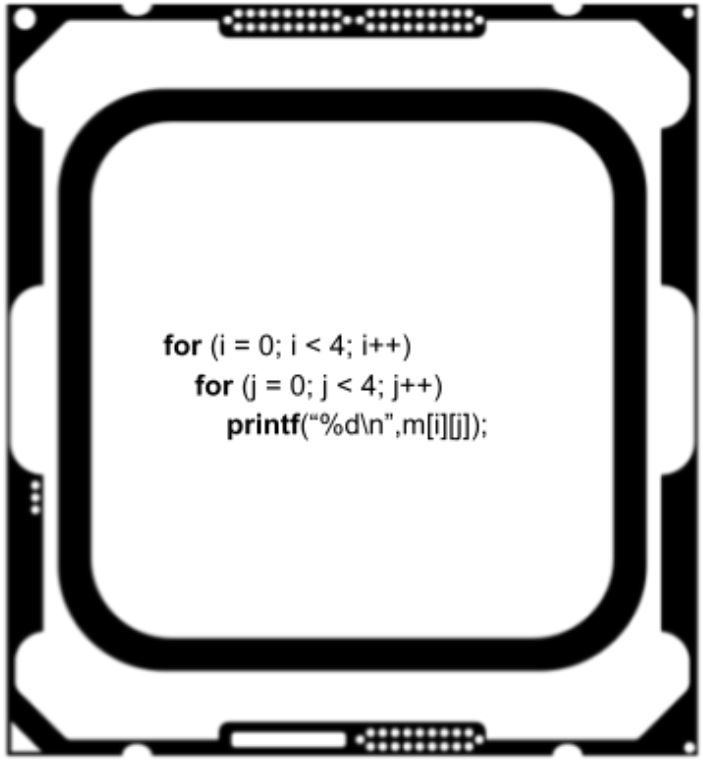
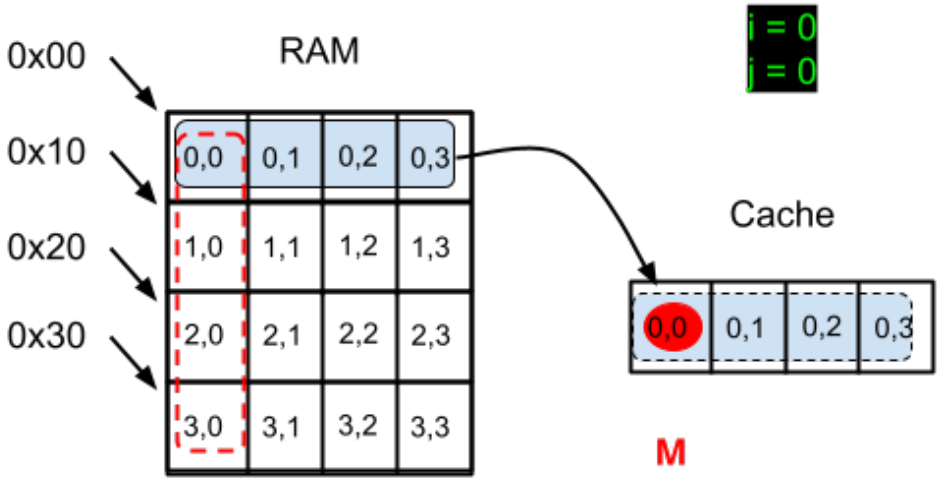


Figure 8: Accediendo a la fila **3**

Operación de la cache, una representación visual (6)



¿Qué tan útil será la información de la caché cuando j = 1? ¿Qué información contendrá la caché una vez j = 1? ¿Cuántos *misses* veremos en esta ejecución?

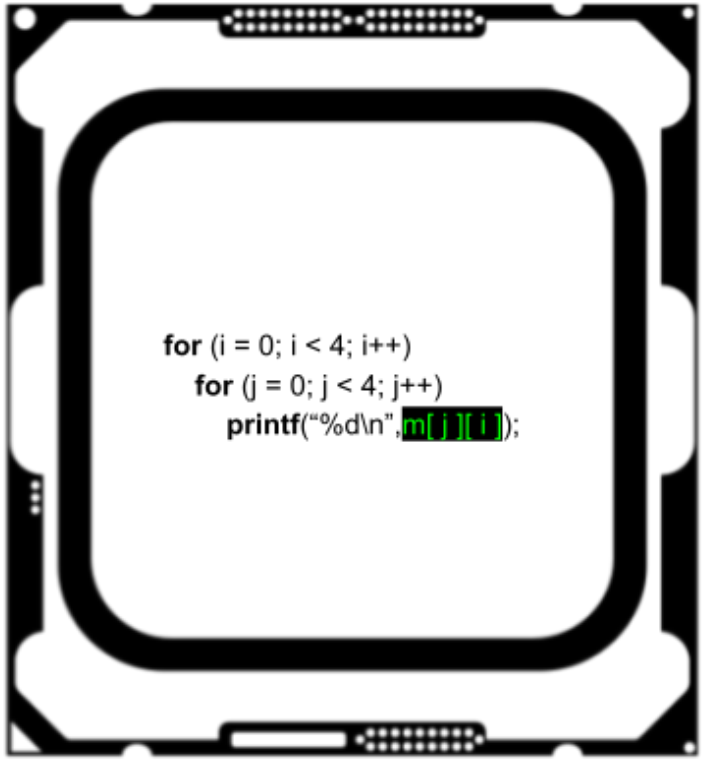


Figure 9: Accediendo por columnas primero

Consideraciones respecto a la Caché

- Accesos **aleatorios** (no organizados) a la RAM **anulan** los beneficios de la cache
- Las operaciones de escritura de datos son más costosas que las de lectura: *read data → write data (in cache) → write data (in RAM)*

Cache y velocidad

- Hay dos elementos que afectan la velocidad de acceso a la cache:
  - *Bandwidth* es la capacidad del canal que comunica la RAM con la cache para transmitir datos
  - *Latency* es el tiempo que toma recuperar el dato que se debe acceder. Incluso **las demoras de acceso a datos** en RAM y/o cache pueden provocar el ***switching*** de un *thread* solicitando acceder a un dato por otro *thread*

Cache, RAM ←→ cache

- Los datos de cache son una copia de datos que están en la RAM. La forma como se localizan los datos que se necesitan de la RAM en la cache puede ser:
  - *Direct-mapped*
  - *Set-associate*
  - *Fully associative*
- Cache placement policies

Cache, *Direct-mapped*

- *Direct-mapped caches* divide la caché en n conjuntos (o *sets*) y cada conjunto tiene un **solo bloque de memoria**
- El *set* donde se ubica un bloque de memoria **se deriva de la dirección de memoria** de ese bloque. Si el *set* está ocupado se reemplaza
- Suponga una memoria de 16 KB (2^{14}), una caché de 256 bytes (2^8) y con bloque (o línea de caché) de 4 bytes (2^2). Esta caché tendrá 64 *sets* ($256/4 = 2^6$)
- La dirección de memoria (14 bits) se divide en:
  - *Offset* (2 bits) para indicar el byte en la línea de caché
  - *Index* (6 bits) para indicar uno de los 64 *sets* de la caché
  - *Tag* (6 bits) representa un *tag* que asocia la dirección con la solicitud en la caché

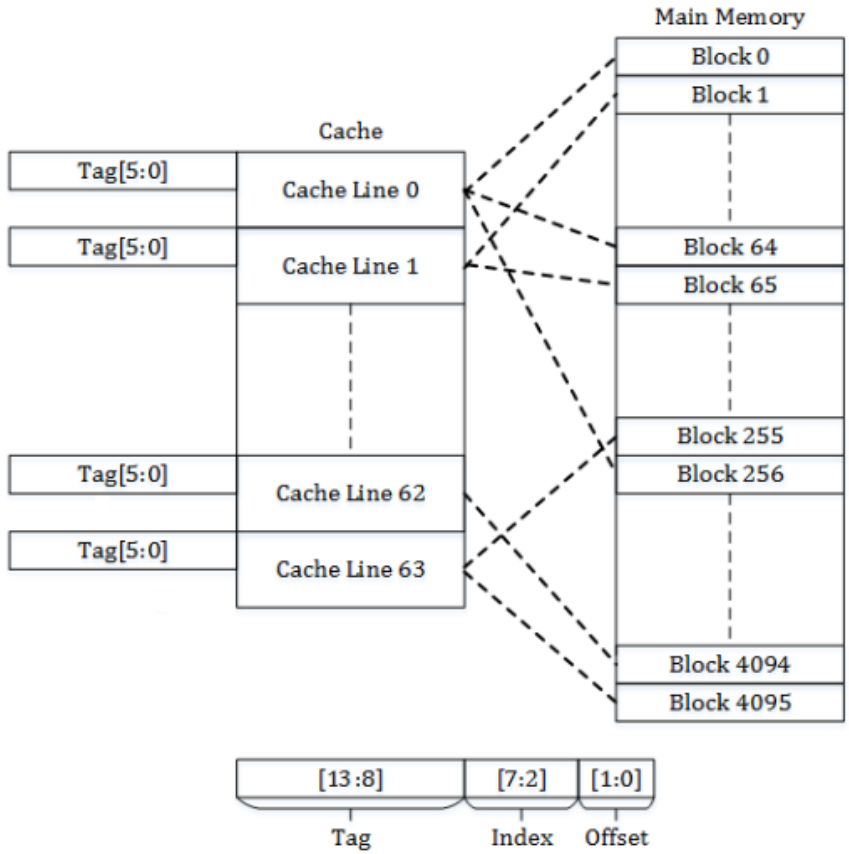


Figure 10: Direct mapping

Ejemplos

- 0x0000 = 0b00 00000000 0000
  - *tag* = 0b00 0000 , *index* = 0b0000 00 , *offset* = 0b00
  - Corresponde al bloque 0 de memoria (0b00 0000 0000 00) y lo asocia al conjunto 0 de caché (0b0000 00)
- 0x00FF = 0b00 00001111 1111
  - *tag* = 0b00 0000 , *index* = 0b1111 11 , *offset* = 0b11
  - Corresponde al bloque 63 de memoria (0b00 0000 1111 11) y lo asocia al conjunto 63 de caché (0b1111 11)
- 0x0100 = 0b00 00010000 0000
  - *tag* = 0b00 0001 , *index* = 0b0000 00 , *offset* = 0b00
  - Corresponde al bloque 64 de memoria (0b00 0001 0000 00) y lo asocia al conjunto 0 de caché (0b0000 00)

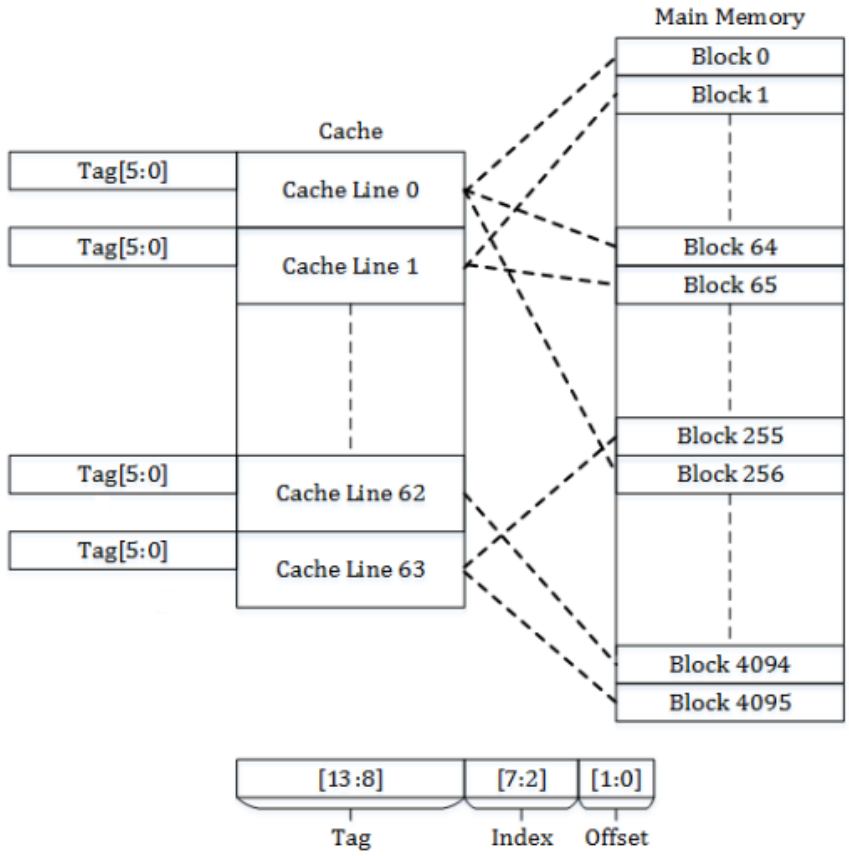


Figure 11: Direct mapping

Cache, *Fully associative*

- La caché es organizada como un único conjunto de caché con **múltiples líneas de caché**
- Un bloque de memoria puede **ocupar cualquiera** de las líneas de caché
- Este esquema **tiene la ventaja** de:
  - Ubicar un bloque de memoria en cualquier línea de caché
  - Ofrecer una amplia variedad de algoritmos de reemplazo
- Este esquema **tiene la desventaja** de ser lento pues toma tiempo iterar sobre todas las líneas de la caché
- Considere:
  - Una memoria de 16 KB (2^{14}), una caché *fully associative* de 256 bytes (2^8) y un tamaño de bloque de 4 bytes.
  - Un único conjunto con $2^8/2^2 = 64$ líneas de caché de tamaño 4 bytes
  - Una dirección de memoria se divide en 2 bits para el *offset* y 12 bits para el *tag*

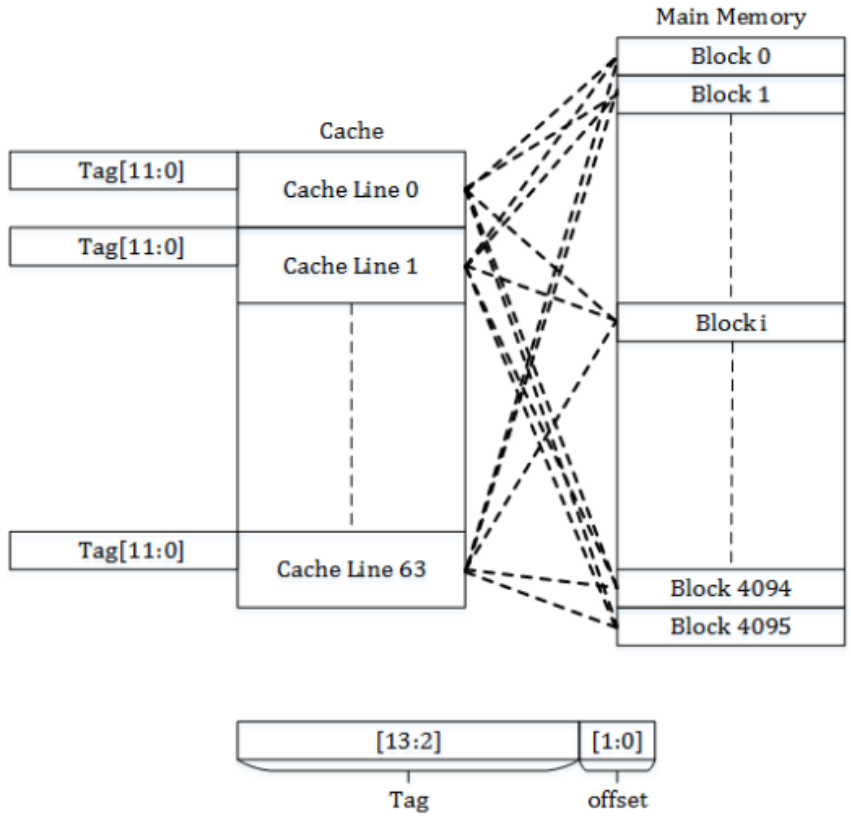


Figure 12: Fully-associative

Cache, *Set associative* (1)

- Es una mezcla del *direct-mapped* y del *fully associative*. La caché se divide en n conjuntos y cada conjunto tiene m líneas de caché
- Un bloque de memoria primero **se asocia a un conjunto** y **se ubica en cualquier línea de caché de ese conjunto**
- Considere:
  - Una memoria de 16 KB (2^{14}), un tamaño de bloque de 4 bytes, y un caché *2-way set-associative* de 256 bytes (2^8). Se usan 14 bits para representar una dirección de memoria
  - Como es un caché ***2-way*** *set-associative* el número total de conjuntos en el caché es $\frac{256}{4 \times 2} = 32$
- Una dirección de memoria (de 14 bits) está dividida en el *offset* (2 bits), el índice (5 bits) y el *tag* (7 bits)

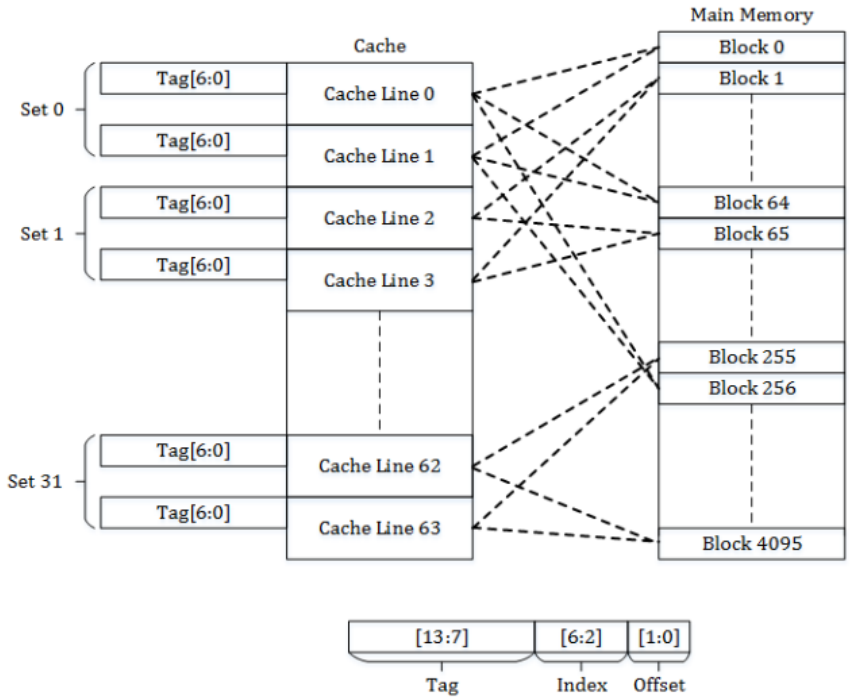


Figure 13: Set-associative

Cache, *Set associative* (2)

Ejemplos

- 0x0000 = 0b00 0000 0000 0000
  - *tag* = 0b00 0000 0 , *index* = 0b000 00 , *offset* = 0b00
  - Corresponde al bloque 0 de memoria (0b00 0000 0000 00) y lo asocia al conjunto 0 de caché (0b000 00)
- 0x00FF = 0b00 0000 1111 1111
  - *tag* = 0b00 0000 1 , *index* = 0b111 11 , *offset* = 0b11
  - Corresponde al bloque 63 de memoria (0b00 0000 1111 11) y lo asocia al conjunto 31 de caché (0b111 11)
- 0x0100 = 0b00 0001 0000 0000
  - *tag* = 0b00 0001 0, *index* = 0b000 00 , *offset* = 0b00
  - Corresponde al bloque 64 de memoria (0b00 0001 0000 00) y lo asocia al conjunto 0 de caché (0b000 00)

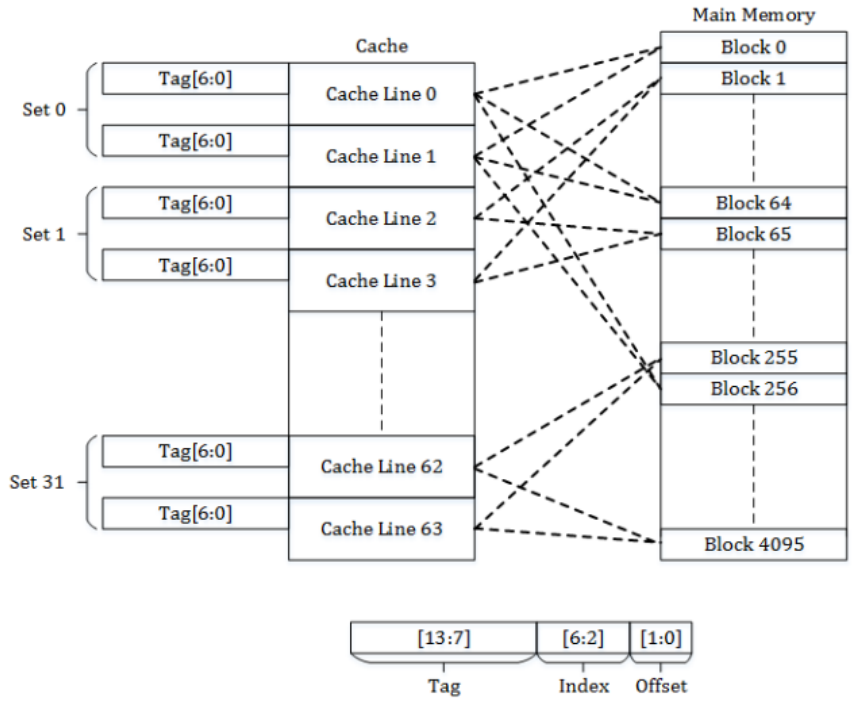


Figure 14: Set-associative

Cache - palabras finales

- Arquitecturas modernas de procesadores tienen diferentes niveles de cache, algunos cache son **de uso exclusivo** de un *core* pero otros caches son compartidos entre diferentes *cores*
- Los *cores* se comunican **compartiendo memoria** a través de mecanismos de **coherencia de cache**
- Cuando varios *cores* comparten una línea de cache se pueden presentar sobre-escrituras de esta línea, **false sharing**

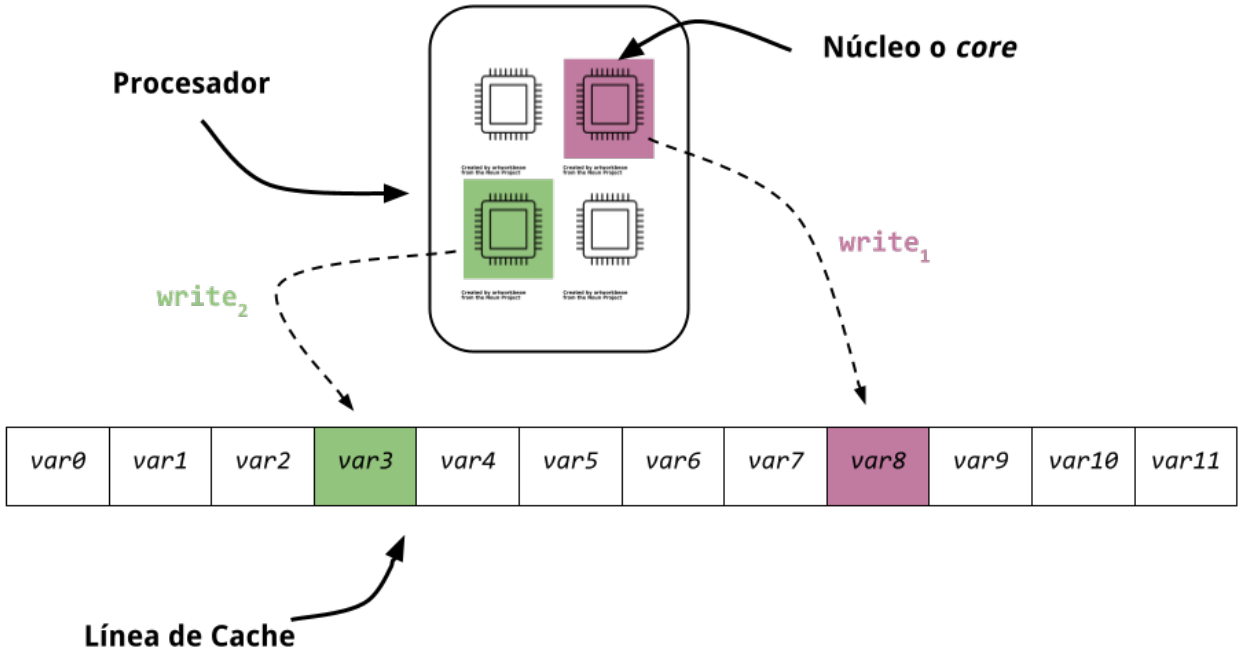


Figure 15: *Cores* compartiendo línea de cache

Memoria virtual

- La **memoria virtual**
  - Permite que cada procesador tenga su propio **espacio de direcciones** de memoria
  - Permite que un proceso *sienta* que tiene un espacio **gigante** de memoria (alrededor de 18,500,000 de Terabytes)
  - La **memoria de intercambio** (o *swap*) se encuentra en disco y guarda aquellas **páginas** de memoria que no caben en memoria física → la RAM es un cache de los datos que están en disco
- Los procesos conocen de direcciones virtuales y es responsabilidad del **MMU** convertir direcciones de memoria virtual a física
- Acceder a una **dirección de memoria** que está en *swap* genera un **fallo de página** el cual es **costoso en tiempo**
- Muchos fallos de página **decrementan el rendimiento** de un programa

Memoria virtual (2)

- La **localidad de los datos** hace referencia a un **conjunto limitado de direcciones de memoria** que en un momento dado son accedidos por un proceso
- Este conjunto de direcciones, **agrupadas en páginas**, se le conoce como el ***working set*** y se refiere entonces a la cantidad de memoria que en un espacio “pequeño” de tiempo un programa accede
- Esta agrupación de páginas es pequeño y logra ser ubicado en el espacio de memoria asignado al proceso
- El **TLB** es un **cache de direcciones** que permite asociar rápidamente una **dirección de memoria virtual** con uno de **memoria física**
- La cantidad de direcciones que puede contener un TLB es limitado (8 a 128) y cuando un proceso accede a un ***working set*** que sobrepasa la capacidad del TLB genera lo que se conoce como *TLB trashing* (e.g. acceder a una matriz muy grande)

Sistemas multiprocesador

- Sistemas de varios procesadores y cada uno de ellos con su propio ***banco*** de memoria
- Cada procesador accede a su propio banco más rápidamente que al banco conectado a otro procesador, **NUMA** *non-uniform memory access*
- El problema del ***false sharing*** se incrementa en este escenario
- Hacer uso eficiente del cache es un **debe ser** en programación serial pero en un **tiene que ser** en programación paralela

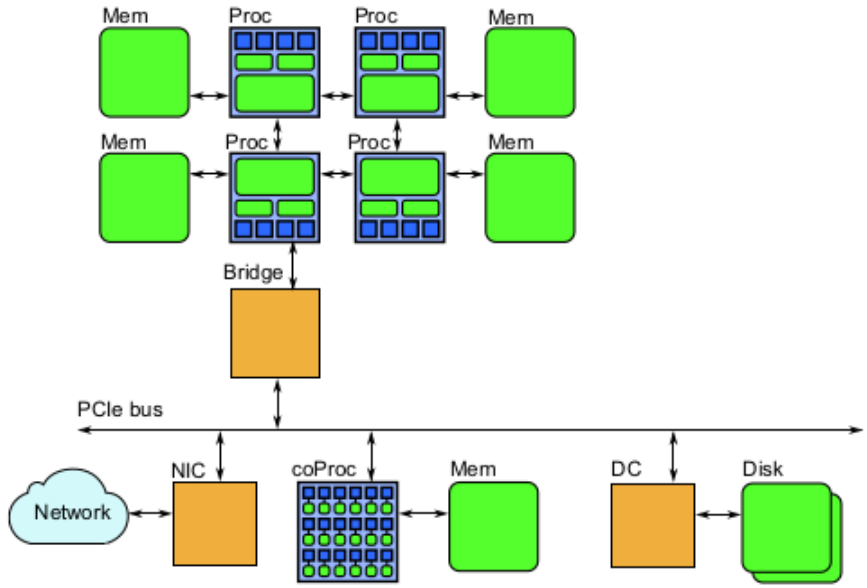


Figure 16: Esquema de sistema multiprocesador

Sistemas multiprocesador (2)

- El PCI bus permite la rápida interconexión entre dispositivos tales como: NIC, MIC (Intel Many Integrated Core), GPU, Disco, entre otros
- El MIC constituido de alrededor de 50 *cores* es adecuado para operaciones vectoriales
  - Procesadores adecuados para operaciones escalares
  - MIC adecuado para operaciones vectoriales. Puede ejecutar programas que usualmente se corren en la CPU, multiplicación de matrices.
  - GPU adecuado para tareas masivamente paralelas (cientos o miles de operaciones en paralelo). Las GPUs requieren instrucciones especiales para ser utilizadas

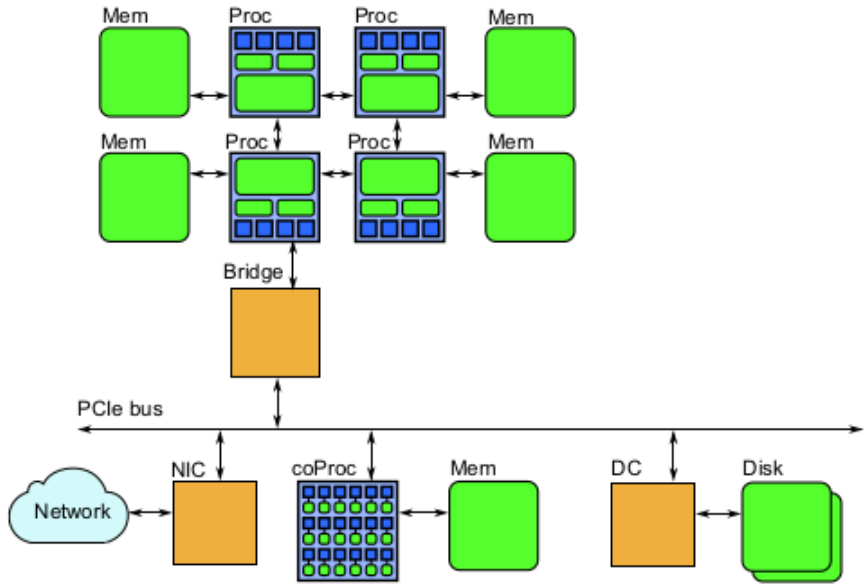


Figure 17: Esquema de sistema multiprocesador

Características claves del rendimiento

- Dos elementos que se deben considerar a la hora de maximizar el paralelismo son: ***data locality*** y ***availability of parallel operation***
- Aunque algún afinamiento específico del hardware puede llegar a ser necesario, si estos dos elementos no se tienen en cuenta a la hora de programar entonces **no hay afinamiento que valga**

Data locality Localidad de los datos

- A la hora de programar se debe considerar:
  - Dividir el procesamiento de datos en tamaños que quepan en el cache
  - Pensar en la reutilización de datos e instruciones
  - Reducir el número de ***TLB misses***
  - Alinear los datos para reducir el ***false sharing***
- Estas consideraciones se deben tener en cuenta incluso a la hora de programar secuencial
- Estos ajustes dependen en gran manera de las características del hardware (tamaño del cache, tamaño del TLB) entonces desarrollar el código de modo que este esté parametrizado y que se pueda adecuadar a las características del hardware (***autotuning***)
- Considerar la proporción de operaciones en CPU vs las operaciones en RAM, ***intensidad aritmética***. Es decir, maximizar operaciones en la CPU en lugar de operaciones que requieran el uso de datos en RAM

Parallel Slack

- *Parallel slack* hace referencia a especificar un paralelismo potencial (maximizar el paralelismo en el código) el cual es mayor que paralelismo real (capacidad del hardware)
- Un ejemplo típico de *parallel slack* es lanzar la ejecución de varios hilos sobre un mismo núcleo. Sin embargo, **sobrecargar** un núcleo puede tirar al traste la velocidad de procesamiento
- Suponga un escenario donde usted lanza varios hilos sobre un mismo núcleo pero dentro del código del hilo **este invoca una función g()** que fue desarrollada para ser ejecutada en paralelo. Esto causará una sobrecarga importante en el número de hilos por *core*

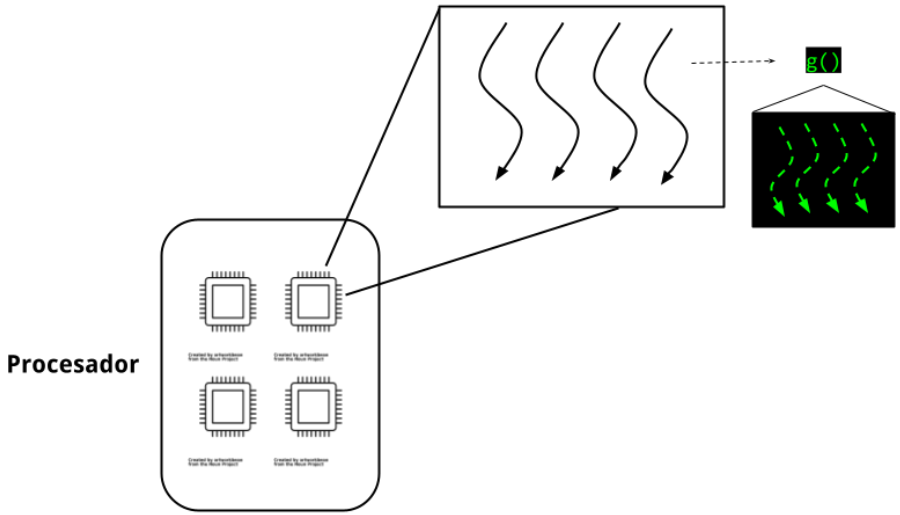


Figure 18: Over subscription

Taxonomía de Flynn

- La caracterización de Flynn se define alrededor de dos aspectos de la computación: **flujo de control** y **gestión de datos**
  - *Single Instruction, Single Data (SISD)* → procesador escalar
  - *Single Instruction, Multiple Data (SIMD) array processor* capaz de aplicar **una operación** sobre **muchos datos**
  - *Multiple Instruction, Multiple Data (MIMD)* se evidencia más naturalmente en *cluster* de computadores

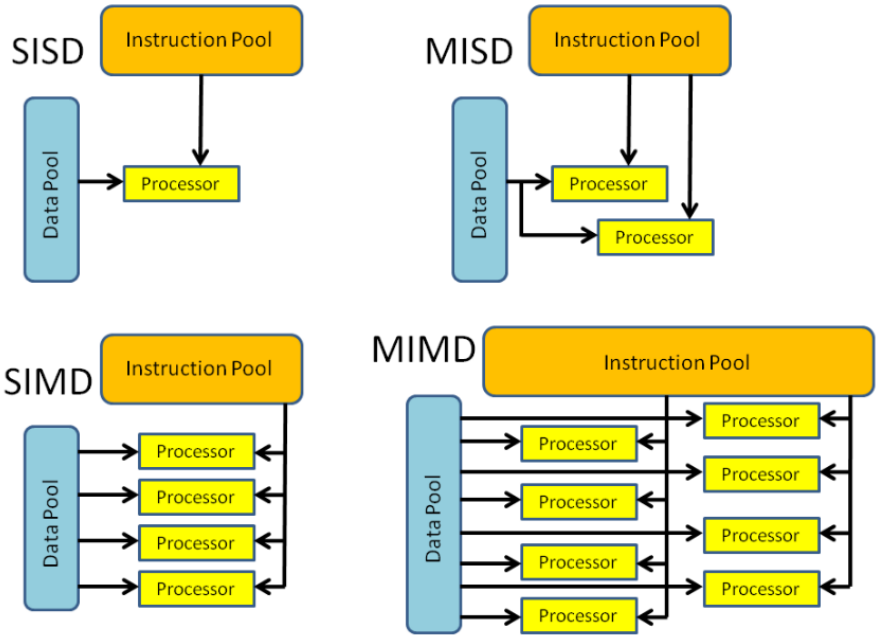


Figure 19: Taxonomía de Flynn

Otros tipos de entornos computacionales

- Entornos de memoria compartida. Equipos generalmente distribuidos pero que comparten una región de memoria. La memoria es distribuida pero por software se oculta la distribución
- SIMT (*Single Instruction Multiple Threads*) es un mosaico de SIMDs. En este contexto existe un procesador de control que gestiona todos los SIMDs del entorno computacional.
  - Los *threads* en un SIMD son llamadas *fibers*
  - Se espera que las *fibers* en un SIMD accedan a una misma línea de cache por eficiencia
  - *fibers* en diferentes *SIMDs* (o *cores*) deberían acceder a diferentes líneas de cache para no causar ***divergent memory accesses***