Arquitectura de Computadores

TEMA 4

Paralelismo a nivel de datos: arquitectura vectorial, instrucciones SIMD para multimedia, GPUs

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

Curso 2020-2021

Contenidos

- Introducción
- Arquitectura vectorial
  - Repertorio de instrucciones vectoriales
  - Tiempo de ejecución, medidas de rendimiento
  - Procesamiento selectivo de elementos
  - Ejemplos
- Instrucciones SIMD para procesamiento multimedia
- Unidades para procesamiento gráfico (GPUs)
  - Modelo de programación a alto nivel: CUDA
  - Instrucciones PTX
  - Arquitectura del elemento de proceso
- Paralelismo a nivel bucle: vectorización
  - Detección de dependencias
- Bibliografía
  - Cap 4 y Apéndice G de Hennessy & Patterson, 5th ed., 2012

(Nota.- En la elaboración de este material se han utilizado algunos contenidos del curso CS152 de la UC Berkeley)

Introducción

- SIMD (Single Instruction Multiple Data).
  - Una operación (codificada como una sola instrucción de LM) se ejecuta sobre un conjunto de datos (en contraposición a SISD).
  - Ejemplo:
    - En lenguaje matemático: $\vec{V3} = \vec{V2} + \vec{V1}$
    - En LAN:
      ```
      for (i = 0; i<N; i++)
          V3(i) = V2(i) + V1(i);
      ```
    - En LM: `ADDV  V3,V2,V1`

- Arquitectura SIMD: puede explotar una cantidad importante de paralelismo de datos en
  - Aplicaciones de ciencia/ingeniería con abundante cálculo matricial (ámbito tradicional)
  - Nuevas aplicaciones: gráficos, visión artificial, comprensión de voz, Deep Learning...

Introducción

- Eficiencia energética de SIMD: puede ser ventajosa frente a MIMD
  - Sólo es preciso hacer un "fetch" para operar sobre varios datos.
  - Ahorro de energía atractivo en dispositivos portátiles
- En SIMD el programador sigue "pensando" básicamente en un flujo secuencial de instrucciones.
- Soporte arquitectónico para explotar paralelismo SIMD
  - Arquitectura vectorial
  - Extensiones SIMD (extensiones multimedia)
  - Graphics Processor Units (GPUs)

Arquitectura vectorial

- Ideas básicas
  - Leer conjuntos de datos sobre "registros vectoriales"
  - Operar sobre el contenido de estos registros
  - Almacenar los resultados finales en memoria
    - Usar los registros vectoriales para ocultar la latencia de memoria
- Características de las operaciones vectoriales
  - Secuencias de cálculos independientes ➔ Ausencia de riesgos
  - Alto contenido semántico
    - Una instrucción ➔ Muchas operaciones
  - Patrón de accesos a memoria conocido
  - Explotación eficiente de memoria entrelazada
  - Disminución de instrucciones de salto (un bucle completo puede transformarse en una instrucción)
    - Reducción conflictos de control

En el principio... Seymour Cray – CDC 6600 (1963)

- No es una arquitectura vectorial, pero...
  - Muy segmentada, con palabra de 60 bits
  - Arquitectura Load/Store (RISC??)
  - Mp de 128 Kword con 32 bancos
  - 10 Fus (paralelas, no segmentadas)
    - PF: sumador, 2 mult, divisor
  - Control cableado
  - Planificación dinámica de instrucciones (scoreboard)
  - 10 procesadores de E/S
  - Clock 10 MHz
    - Muy rápido para la época
    - Suma FP en 4 ciclos
  - >400,000 transistores, 750 sq. ft. (~70 m^2), 5 tons, 150 KW, refrigeración freon
  - Máquina más rápida del mundo durante 5 años (hasta el 7600)
  - Vendidas >100 (a $7-10M c.u.)

Arquitectura vectorial

- En el principio... Seymour Cray – CDC 6600 (1963)

- Thomas Watson Jr., IBM CEO, August 1963:

  "Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer."

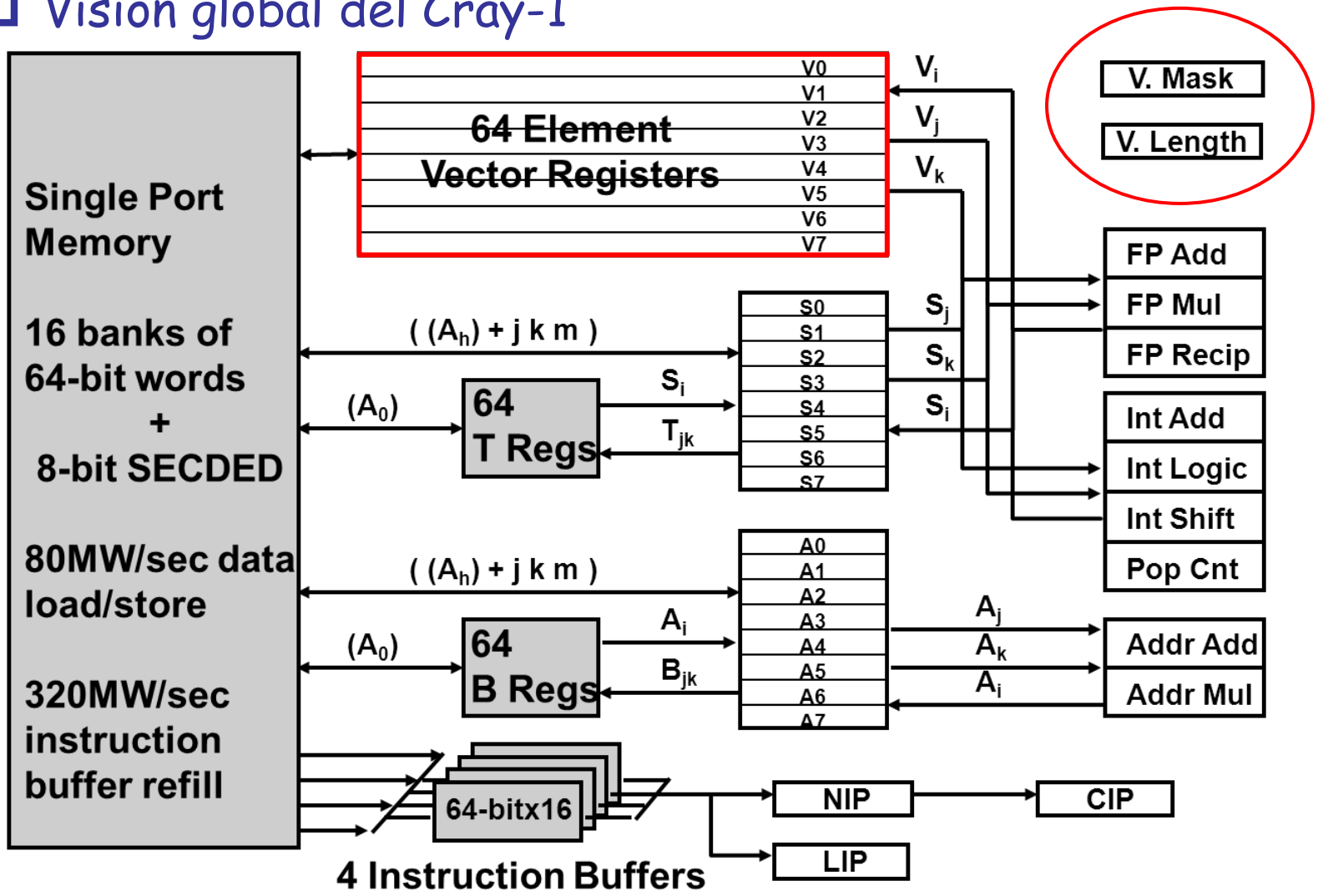
- To which Cray replied: "It seems like Mr. Watson has answered his own question."

- Despues Cyber 205, ETA10

El Cray-1 (1976)

- Unidad escalar
  - Arquitectura Load/Store
- Extensión Vectorial
  - Registros Vectoriales
  - Instrucciones Vectoriales
- Implementación
  - Control cableado
  - UF muy segmentadas
  - Memoria entrelazada
  - Sin cache de datos
  - Sin memoria virtual

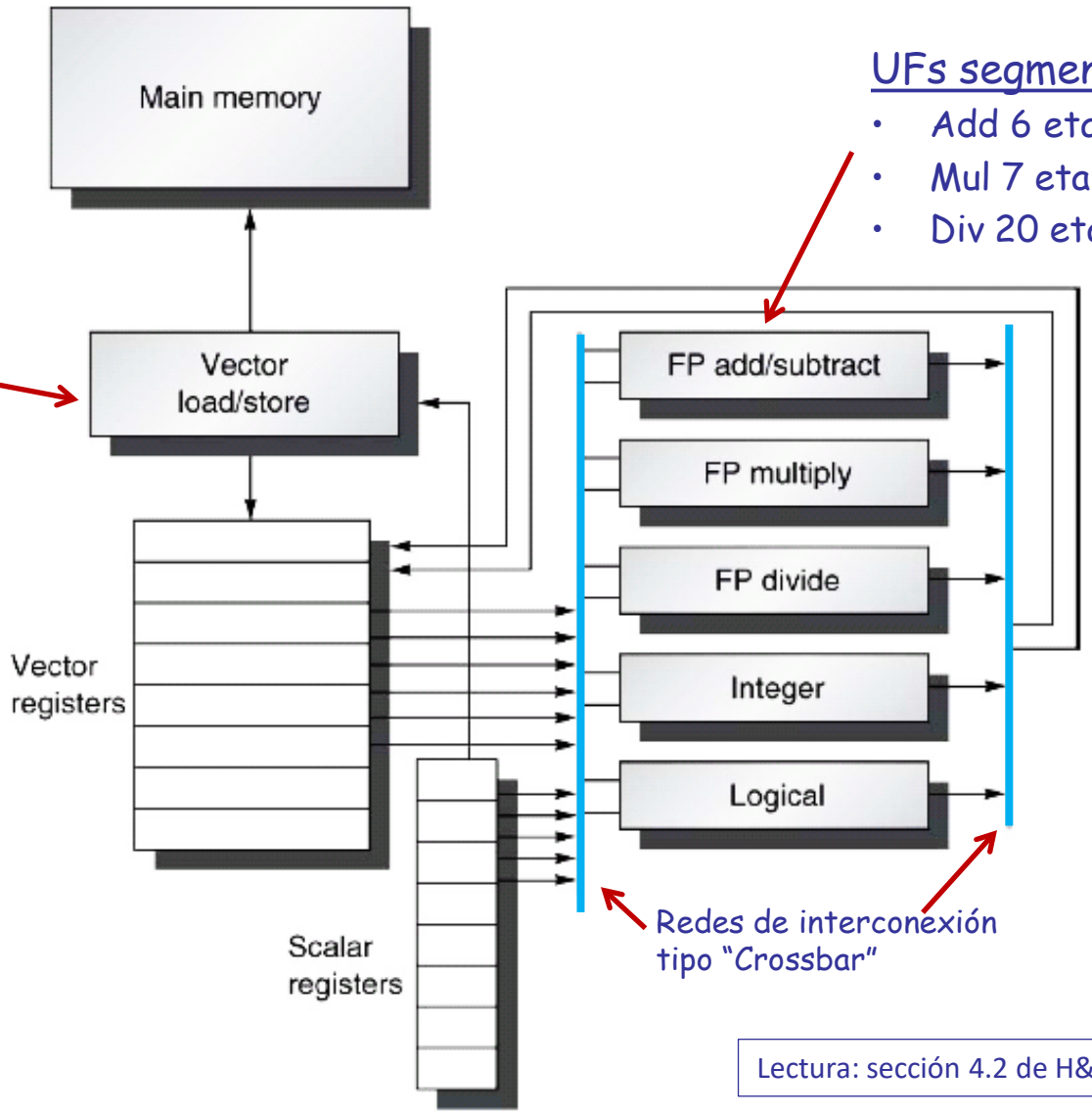
Visión global del Cray-1



T ciclo banco memoria: 50 ns, T ciclo procesador: 12,5 ns (80 Mhz)

❑ Estructura de un procesador vectorial: VMIPS

Funcionalmente equivalente a un pipe: un dato por ciclo, después de latencia inicial (12 ciclos).



UFs segmentadas:
- Add 6 etapas
- Mul 7 etapas
- Div 20 etapas

8 registros vectoriales
- 1 reg = 64 elementos
- 1 elemento = 64 bits
- 16 read, 8 write ports

Lectura: sección 4.2 de H&P 5th ed.

Arquitectura vectorial: repertorio VMIPS (1)

- Operaciones vectoriales aritméticas sobre registros
- Instrucciones especiales de carga/almac de vectores (LV,SV)
- Modos de direccionamiento especiales para vectores no contiguos. Ejemplos
  - LVWS: Load Vector With Stride. Carga elementos equiespaciados a una cierta distancia
  - LVI: Load Vector using Index. El contenido de un registro vectorial indica las posiciones de los elementos a cargar.
- Registros especiales de longitud vectorial (VLR) y máscara (VM)
  - Reg VLR: Indica la longitud de los vectores a procesar (≤ 64)
  - Reg VM: Registro de 64 bits. Para las posiciones con VM a cero, la operación no se ejecuta.
    - Ejecución selectiva de operaciones sobre componentes

Arquitectura vectorial: repertorio VMIPS (2)

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1, V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e .. R1 + i x R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e .. R1 + i x R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2) ,V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).

Arquitectura vectorial: repertorio VMIPS (3)

Instruction	Operands	Function
CVI	V1,R1	Create an index vector by storing the values 0, 1 x R1, 2 x R1, ... ,63 x R1 into V1.
S--VV.D S--VS.D	V1, V2 V1, F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector: otherwise put 0. Put resulting bit vector in vector mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to vector-length register VL. Move the contents of vector-length register VL to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to vector-mask register VM. Move contents of vector-mask register VM to F0.

Código escalar vs. Código vectorial

- Y = a*X + Y (AXPY, Sup: vectores de 64 elementos)
  - Versión escalar

```
        L.D     F0, a           ; load scalar a
        DADDIU  R4,Rx,#512      ; last address to load: (Rx)+8*64
Loop:   L.D     F2, 0(Rx)       ; Load X[i]
        MUL.D   F2, F2, F0      ; A x X[i]
        L.D     F4, 0(Ry)       ; Load Y[i]
        ADD.D   F4, F4, F2      ; A x X[i] + Y[i]
        S.D     F4, 0(Ry)       ; Store Y[i]
        DADDIU  Rx, Rx, #8      ; increment index X
        DADDIU  Ry, Ry, #8      ; increment index Y
        DSUBU   R20, R4, Rx     ; loop exhausted?
        BNEZ    R20, Loop
```

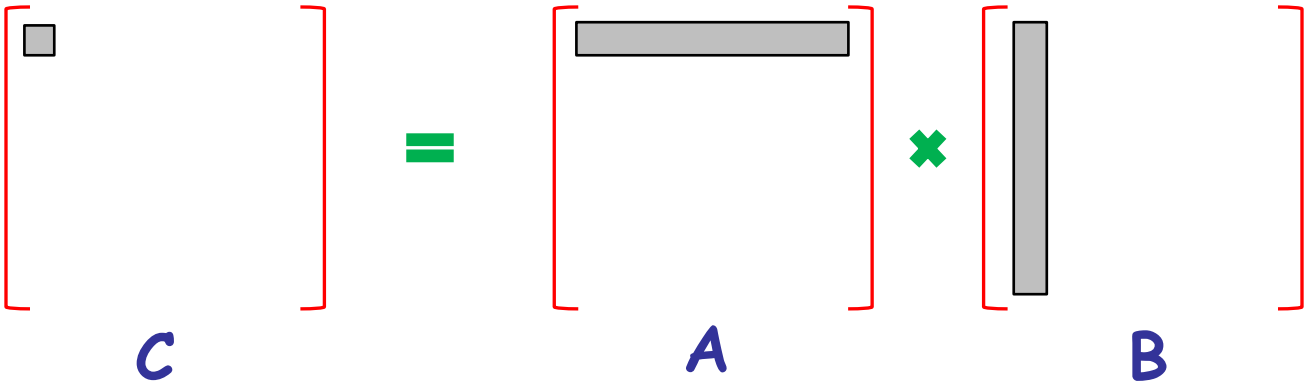
  - Versión vectorial

```
        L.D     F0, a           ; load scalar a
        LV      V1, Rx          ; Load vector X
        MULVS.D V2, V1, F0      ; Vector-scalar multiply
        LV      V3, Ry          ; Load vector Y
        ADDVV.D V4, V2, V3      ; Vector addition
        SV      V4, Ry          ; Store result Y
```

  - Instrucciones ejecutadas: ~ 580 vs. 6 !!

Código escalar vs. Código vectorial

- Ejemplo: producto de matrices nxn
  - Método clásico



  - Para cada elemento de C: 1 producto escalar (fila de A x col de B)
    - Un producto de 2 vectores: instrucción MULVV.D
    - Sumar todos los elementos del vector resultante: No es una operación vectorial ! ➔ Ejecución secuencial (escalar)
  - En total:
    - n^2 instrucciones MULVV.D
    - n^2 pasos de acumulación de todos los elementos de un vector: ~n^3 instrucciones de suma escalar

Código escalar vs. Código vectorial

- Ejemplo: producto de matrices nxn
  - Aplicación de AXPY. Se van calculando todos los elementos de una fila de C a la vez
    - Operaciones par el cálculo de la 1ª fila de C

$$
\begin{array}{lllllll}
c_{11} = & a_{11}b_{11} + & a_{12}b_{21} + & a_{13}b_{31} + & \ldots\ldots + & a_{1n}b_{n1} \\
c_{12} = & a_{11}b_{12} + & a_{12}b_{22} + & a_{13}b_{32} + & \ldots\ldots + & a_{1n}b_{n2} \\
c_{13} = & a_{11}b_{13} + & a_{12}b_{23} + & a_{13}b_{33} + & \ldots\ldots + & a_{1n}b_{n3} \\
\ldots\ldots & \ldots\ldots & \ldots\ldots & \ldots\ldots & \ldots\ldots & \ldots\ldots \\
c_{1n} = & a_{11}b_{1n} + & a_{12}b_{2n} + & a_{13}b_{3n} + & \ldots\ldots + & a_{1n}b_{nn}
\end{array}
$$

    - Método. Paso 0: Inicializar (c_{11}, c_{12}, c_{13},..., c_{1n}) a cero

Paso 1

$$
\begin{array}{lllll}
c_{11} = & a_{11} & b_{11} + & c_{11} \\
c_{12} = & a_{11} & b_{12} + & c_{12} \\
c_{13} = & a_{11} & b_{13} + & c_{13} \\
\ldots\ldots & \ldots\ldots & & \ldots\ldots \\
c_{1n} = & a_{11} & b_{1n} + & c_{1n}
\end{array}
$$

a X + Y

Paso 2

$$
\begin{array}{lllll}
c_{11} = & a_{12} & b_{21} + & c_{11} \\
c_{12} = & a_{12} & b_{22} + & c_{12} \\
c_{13} = & a_{12} & b_{23} + & c_{13} \\
\ldots\ldots & \ldots\ldots & & \ldots\ldots \\
c_{1n} = & a_{12} & b_{2n} + & c_{1n}
\end{array}
$$

• • •

Paso n

$$
\begin{array}{lllll}
c_{11} = & a_{1n} & b_{n1} + & c_{11} \\
c_{12} = & a_{1n} & b_{n2} + & c_{12} \\
c_{13} = & a_{1n} & b_{n3} + & c_{13} \\
\ldots\ldots & \ldots\ldots & & \ldots\ldots \\
c_{1n} = & a_{1n} & b_{nn} + & c_{1n}
\end{array}
$$

Código escalar vs. Código vectorial

- Ejemplo: producto de matrices nxn:
  - Aplicación de AXPY
  - n operaciones AXPY para cada fila
    - n instrucciones MULVS.D
    - n instrucciones ADDVV.D
  - En total:
    - n^2 MULVS.D
    - n^2 ADDVV.D
    - Todas las operaciones son vectoriales

Tiempo de ejecución

- Dependiente básicamente de tres factores
  - Longitud de los vectores operandos
  - Riesgos estructurales: las UF necesarias están ocupadas, no hay puertos del BR disponibles
  - Dependencias de datos
- Velocidad de procesamiento
  - Las FUs del VMIPS consumen (y producen) un elemento por ciclo de reloj
  - El tiempo de ejecución de una operación vectorial es *aproximadamente* igual a la longitud del vector
- Convoy
  - Se denomina así a un conjunto de (una o varias) instrucciones vectoriales que potencialmente pueden ejecutarse juntas (ausencia de riesgos estructurales). Pueden tener riesgos LDE.
- Paso (chime)
  - Unidad de tiempo para ejecutar un convoy
  - m convoyes se ejecutan en m pasos
  - Para vectores de longitud n, ejecutar m convoyes requiere (aprox.) mxn ciclos de reloj (Notación: $T_{chime}=m$)

Tiempo de ejecución

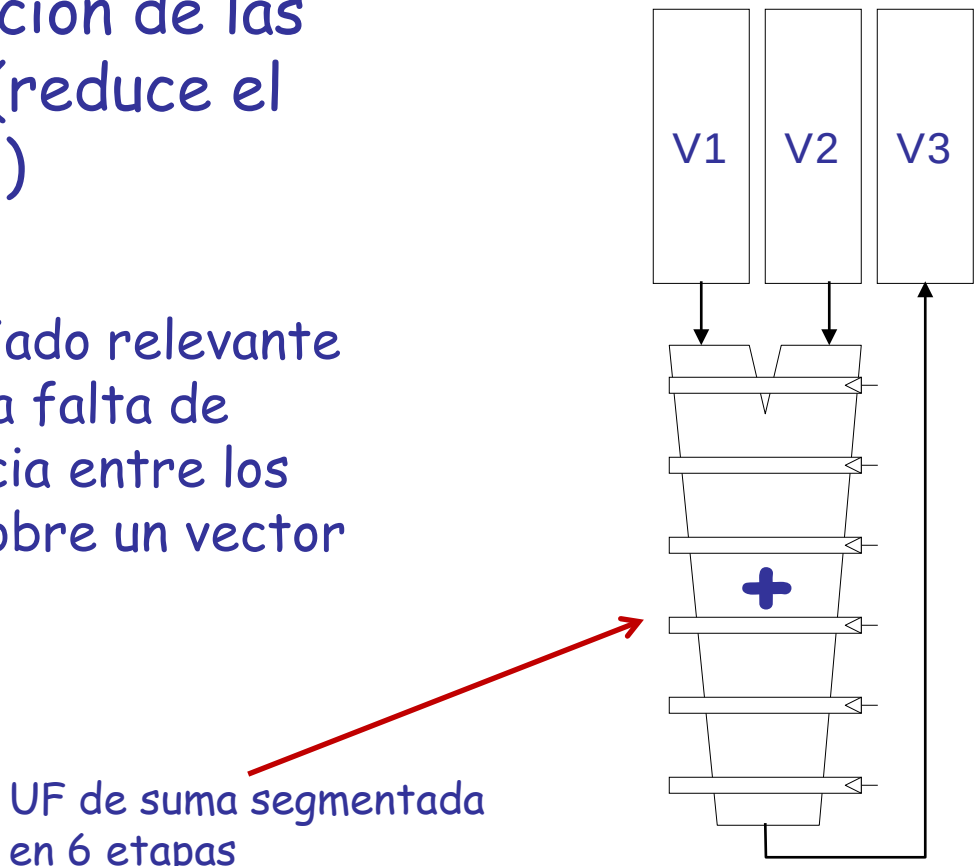
- Convoyes: ejemplo

```
1: LV       V1, Rx      ; load vector X
2: MULVS.D  V2, V1, F0  ; vector-scalar multiply
3: LV       V3, Ry      ; load vector Y
4: ADDVV.D  V4, V2, V3  ; Vector addition
5: SV       V4, Ry      ; Store result Y
```

  - Conflictos:
    - 1 y 2 no tienen conflictos estructurales
    - 3 tiene conflicto estructural con 1 (una sola unidad de Load/Store)
    - 4 no tiene conflictos estructurales con 3
    - 5 tiene conflicto estructural con 3
  - Convoyes resultantes
    - 1. Formado por LV y MULVS.D
    - 2. Formado por LV y ADDVV.D
    - 3. Formado por SV
  - Tiempo de cálculo aprox para 64 componentes: 3x64= 192 ciclos (Tchime=3)

Ejecución de operaciones aritméticas

- Uso de un pipe profundo para la ejecución de las operaciones (reduce el ciclo de reloj)
- Alta latencia
  - No demasiado relevante debido a la falta de dependencia entre los cálculos sobre un vector



V3 ← V1 + V2

Ejecución de operaciones aritméticas

- Operaciones independientes
  - Ejemplo

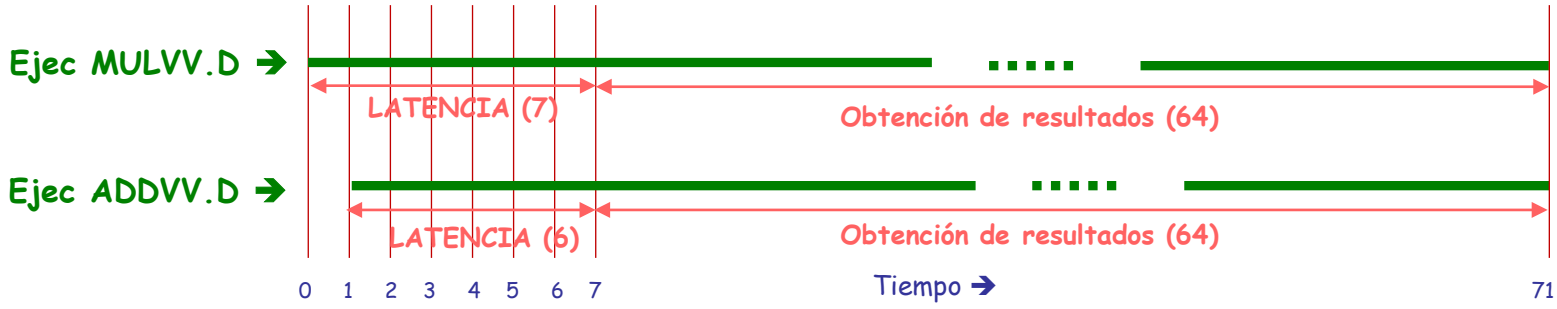
    MULVV.D V1, V2, V3

    ADDVV.D V4, V5, V6

    (1 convoy: la UF de * y la de + pueden actuar a la vez)
  - Comportamiento temporal (1 paso, Tchime=1)
    - Recordar latencias: MUL 7 ciclos, ADD 6 ciclos

Operación	Inicio	Fin
MULVV.D	0	7+64 = 71
ADDVV.D	1	1+6+64 = 71

  - En ausencia de conflictos lanza una instrucción por ciclo
  - Representación

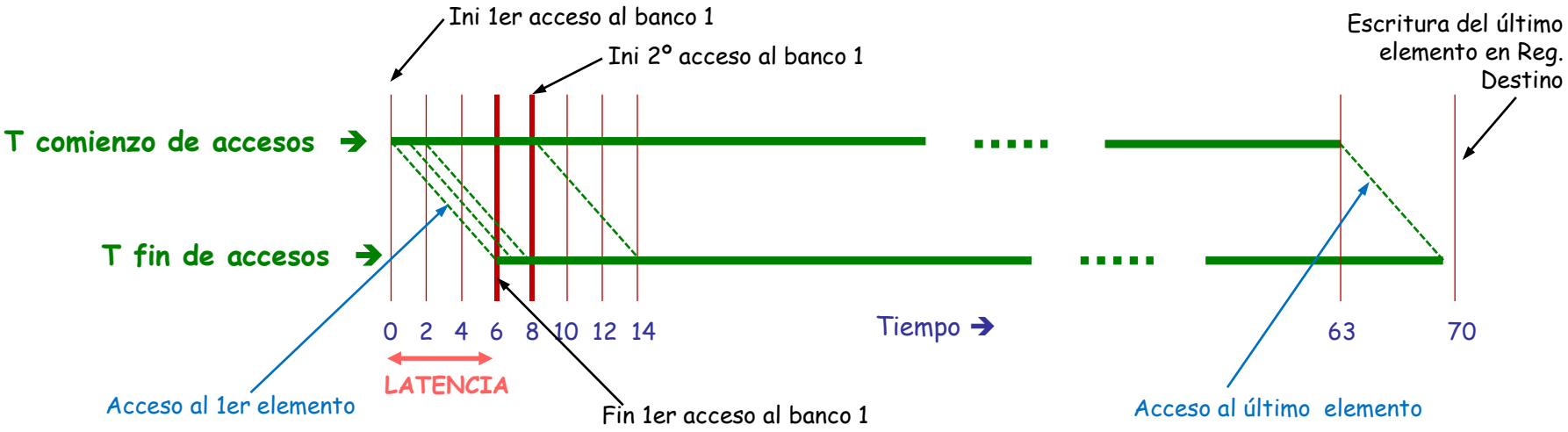


Ejecución de accesos a memoria

- Memoria entrelazada (equivalencia funcional con un pipe)
  - Ejemplo: Sup 8 bancos (ver tr. sigte.), T acceso a memoria: 6 ciclos.
    - Carga de un vector de 64 componentes que comienza en la dirección 136 (cada componente 8 bytes)
    - Formato de dirección: ...xxxx yyy 000 (siendo yyy = nº de banco)

Dir	136	144	152	160	168	176	184	192	200	208	...
Banco	1	2	3	4	5	6	7	0	1	2	...
Tini	0	1	2	3	4	5	6	7	8	9	...
Tfin	6	7	8	9	10	11	12	13	14	15	...

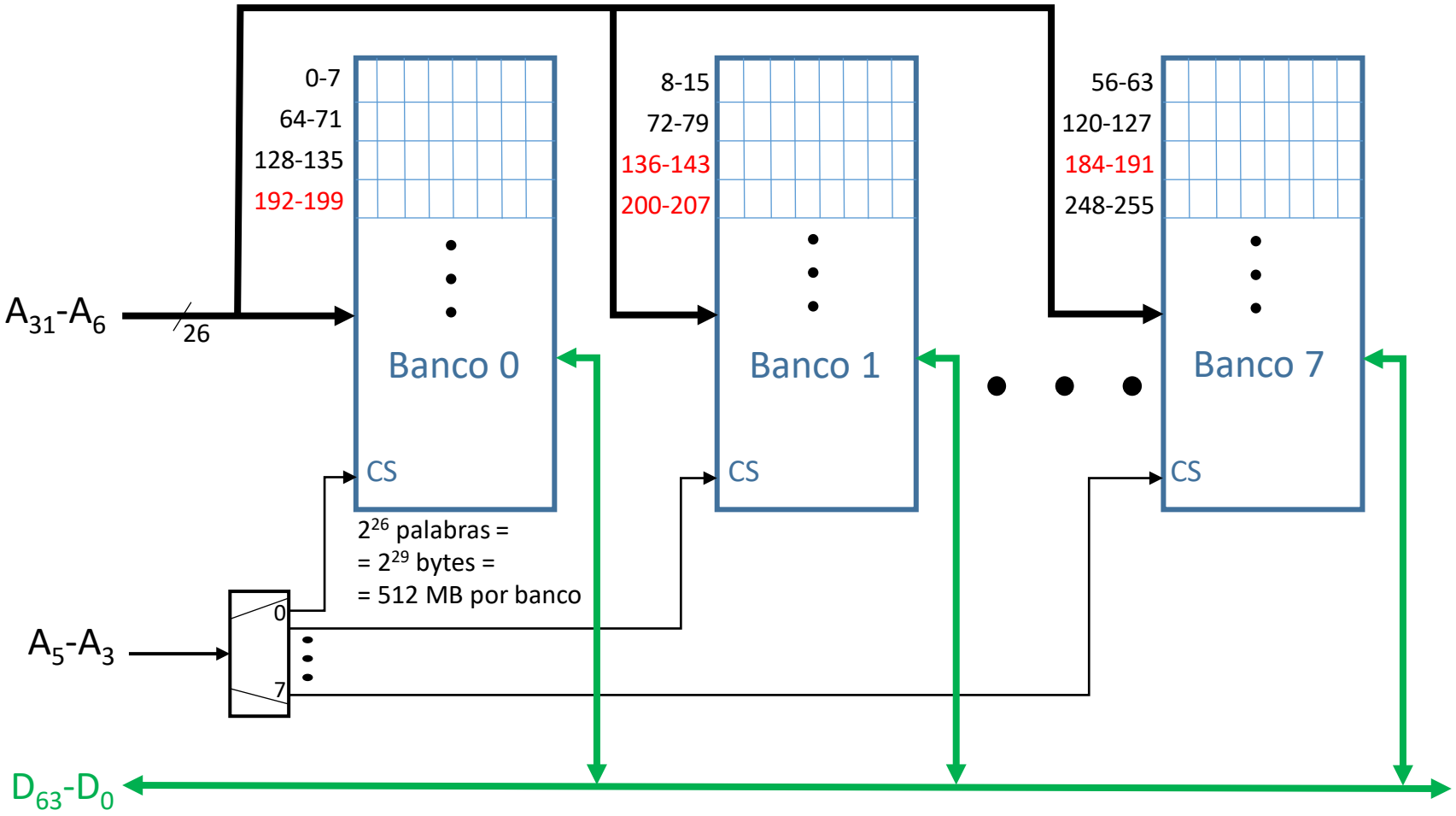
- Diagrama temporal



Memoria entrelazada

Memoria con entrelazamiento de orden bajo (a nivel de palabra) con 8 bancos

- Longitud de palabra: 8 bytes
- Palabras consecutivas están en bancos consecutivos.
- Cada 8 palabras se vuelve a usar el mismo banco.



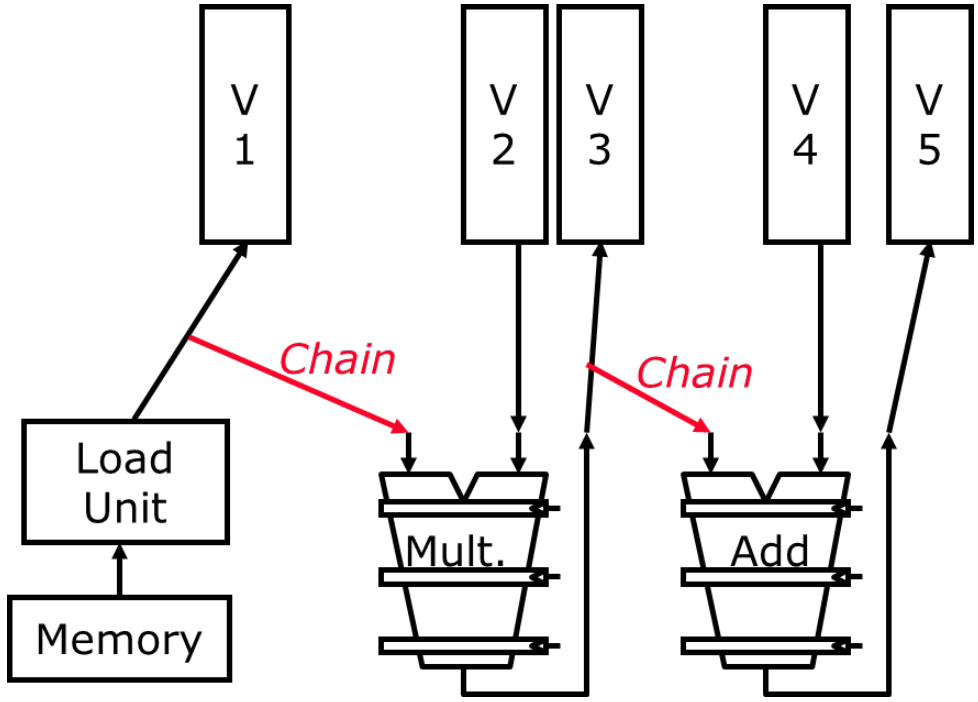
Encadenamiento

- Operaciones dependientes: Tratamiento de dependencias RAW

- Problema

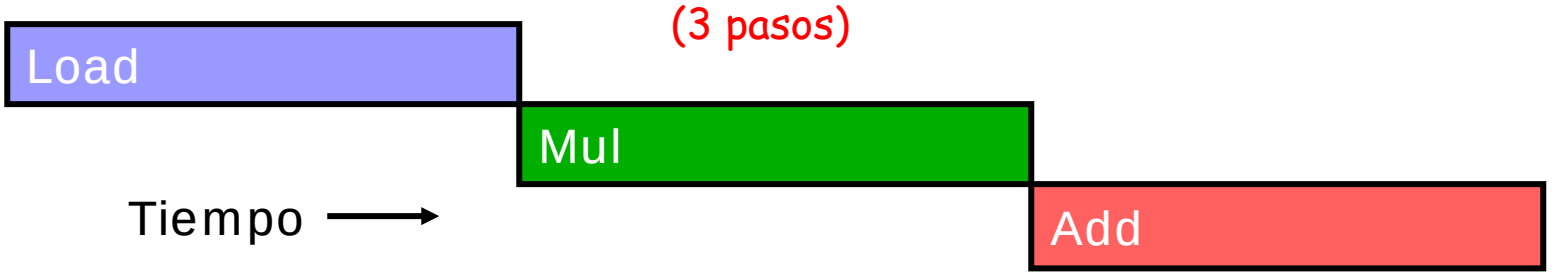
LV	V1
MULVV.D	V3,V1,V2
ADDVV.D	V5,V3,V4

- Solución:
  - Cray-1. Extensión del concepto de anticipación de operandos ⇒ Encadenamiento de operaciones (chaining)
  - 3 operaciones, pero 1 paso
    - Tchime = 1
  - En proc modernos: "encadenamiento flexible"

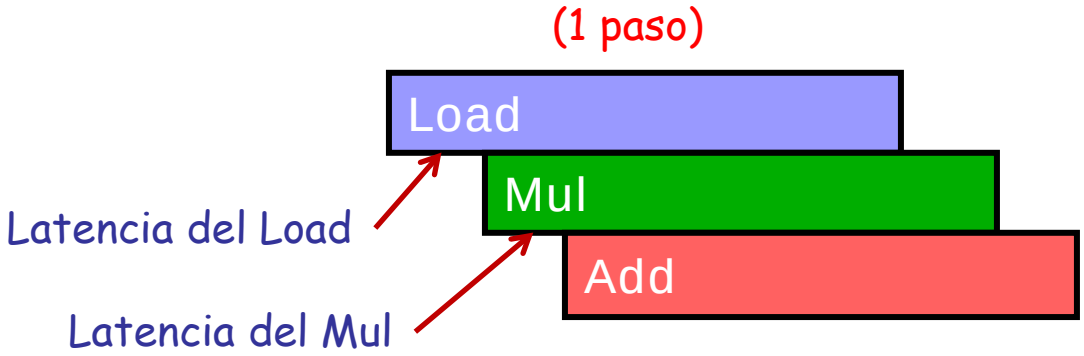


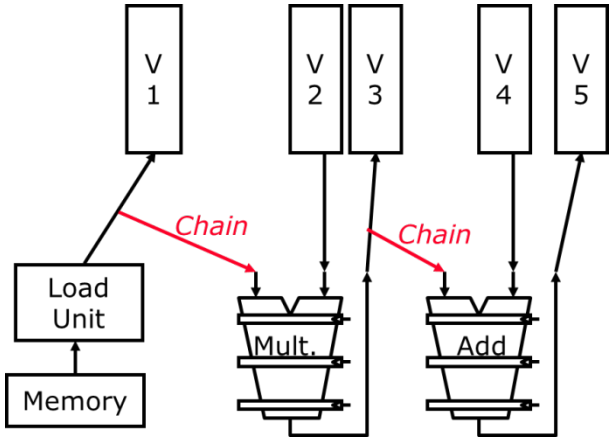
Encadenamiento

- Sin encadenamiento: esperar hasta que se haya calculado el último elemento de la operación anterior



- Con encadenamiento: Una instrucción puede comenzar cuando está disponible el primer elemento de la operación de la que depende

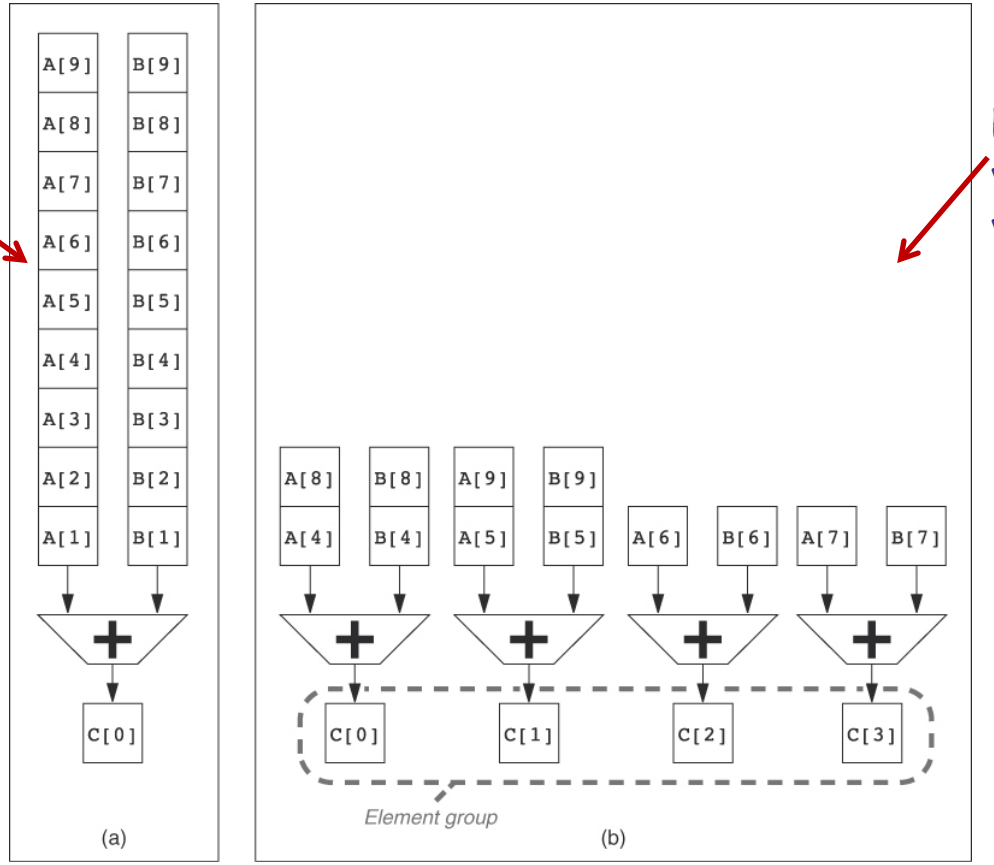




Procesamiento con vías múltiples

- Aceleración de los cálculos poniendo varias UF segmentadas del mismo tipo
  - Solamente una parte de las componentes es procesada por cada UF.
  - Esquema

Ejecución vectorial convencional



Ejecución vectorial con varias vías

Vectores de longitud arbitraria

- Registro de longitud vectorial (VLR)
  - El valor cargado en VLR determina el nº de componentes sobre los que actúa la op. vectorial lanzada.
- Vectores cortos: n ≤ long. registros vectoriales (MVL)
  - Cargar registro VLR
  - Ejecutar operación vectorial
- Vectores largos: n > MVL ➔ Proceso por bloques (strip mining)
  - Descomponer operación en varias suboperaciones
  - n/MVL operaciones vectoriales de longitud MVL
  - 1 operación vectorial de longitud (n mod MVL)

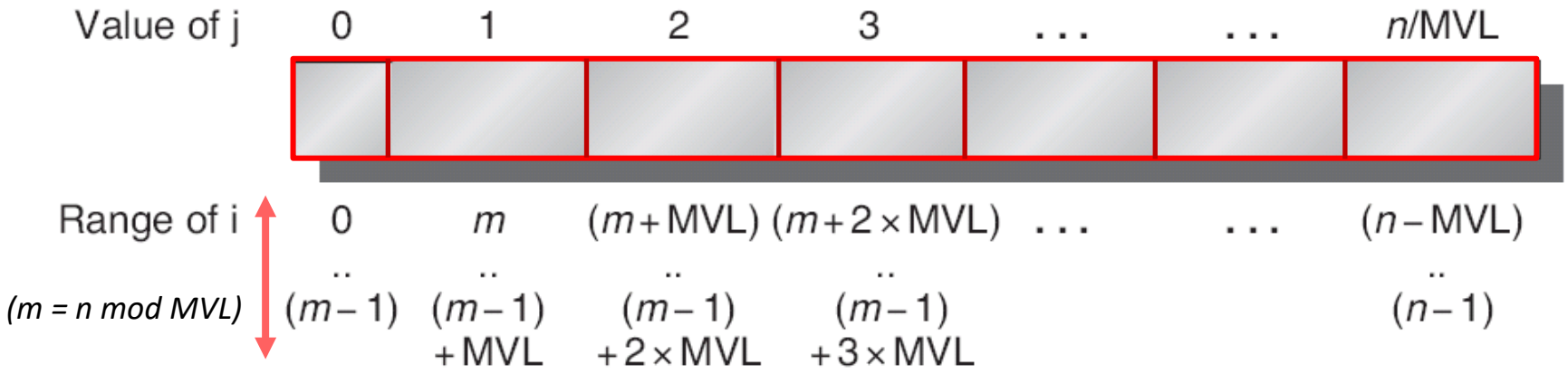
Lectura: sección G.2 de H&P 5th ed.

Vectores de longitud arbitraria (strip mining)

- Ejemplo: Bucles anidados para ejecución de DAXPY

```
low = 0;
VL = (n % MVL);                          /*find odd-size piece using modulo */
for (j = 0; j <= (n/MVL); j=j+1) {       /*outer loop*/
  for (i = low; i < (low+VL); i=i+1)     /*runs for length VL*/
        Y[i] = a * X[i] + Y[i] ;         /*main operation*/
  low = low + VL;                        /*start of next vector*/
  VL = MVL;                              /*reset the length to MVL*/
}
```

- Diagrama de ejecución de DAXPY.
  - Nº de iteraciones del bucle externo: $\lceil n/MVL \rceil$
  - Cada iteración del bucle interno se implementa con instr. vectoriales



Vectores de longitud arbitraria (strip mining)

- Modelo de rendimiento para operaciones por bloques
  - Inicialización de operaciones (T_{base}): Cálculo de dir iniciales, op escalares de preparación del bucle (1 sola vez)
    - Simplificación: despreciable
  - Penalización por inicialización y control del bucle (1 vez por cada itearación)
    - T_{start}: nº ciclos para el llenado de pipes. Depende de las operaciones vectoriales incluidas en el bucle. Si las instr son independientes, el llenado de pipes se solapa.
    - T_{loop}: actualización de punteros, detección de fin. Simplifiación: 15 ciclos
  - Nº de convoyes en el bucle (T_{chime})
  - Tiempo total de cálculo para vectores de n elementos
    - $T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$

  - Rendimiento
    - $R\ (FLOP/ciclo) = \frac{Nº\ de\ Operaciones\ en\ PF}{Tiempo\ cálculo\ (ciclos)} = \frac{Nº\ de\ Operaciones\ en\ PF}{T_n}$

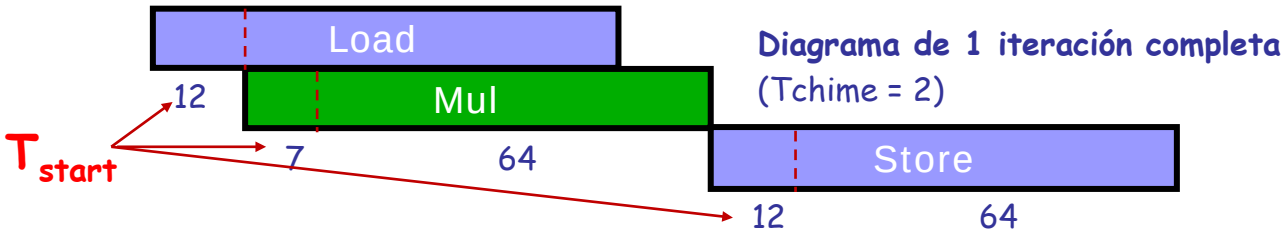
Vectores de longitud arbitraria (strip mining)

- Ejemplo: A= B x s, para vectores de 200 componentes

```
        DADDUI   R2,R0,#1600   ;total # bytes in vector
        DADDU    R2,R2,Ra      ;address of the end of A vector
        DADDUI   R1,R0,#8      ;loads length of 1st segment
        MTC1     VLR,R1        ;load vector length in VLR
        DADDUI   R1,R0,#64     ;length in bytes of 1st segment (8 elements)
        DADDUI   R3,R0,#64     ;vector length of other segments (64 elements)
Loop:   LV       V1,Rb         ;load B
        MULVS.D  V2,V1,Fs      ;vector * scalar
        SV       Ra,V2         ;store A (structural hazard)
        DADDU    Ra,Ra,R1      ;address of next segment of A
        DADDU    Rb,Rb,R1      ;address of next segment of B
        DADDUI   R1,R0,#512    ;load byte offset next segment
        MTC1     VLR,R3        ;set length to 64 elements
        DSUBU    R4,R2,Ra      ;at the end of A?
        BNEZ     R4,Loop       ;if not, go back
```

- $T_n = \left\lceil \frac{n}{MVL} \right\rceil \times \left(T_{loop} + T_{start}\right) + n \times T_{chime} =$

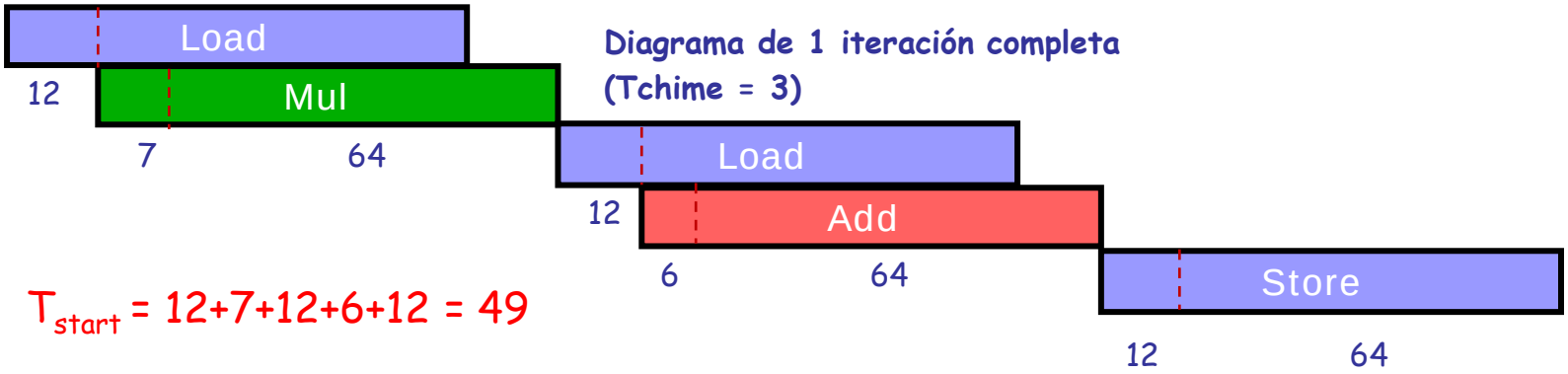
$= \left\lceil \frac{200}{64} \right\rceil \times \left(15 + (12 + 7 + 12)\right) + 200 \times 2 = 4 \times 46 + 400 = 584\ ciclos$



Medidas de rendimiento

- Rendimiento asintótico (R_∞): Rendimiento obtenido para para supuestos vectores de longitud infinita
  - Consideremos la op DAXPY sin limitaciones debidas a la longitud de registros vectoriales (tr. 19, 3 convoyes)
    - 2n FLOP en 3n ciclos => R_∞ = 2/3 FLOP/ciclo.
    - En MFLOPS: Si sup f=500 MhZ => R_∞ = 2/3 x 500 x 10^6 = 333 MFLOPS

  - Efecto de strip mining: Suponemos MVL = 64



T_{start} = 12+7+12+6+12 = 49

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime} = \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + n \times 3 \approx 4 \times n$$

Para n grande: $\left\lceil \frac{n}{64} \right\rceil \approx \frac{n}{64}$

$$R_\infty = \lim_{n\to\infty} R = \lim_{n\to\infty} \frac{2n}{T_n} = \lim_{n\to\infty} \frac{2n}{4n} = \frac{1}{2}\frac{FLOP}{ciclo}$$ = 250 MFLOPS !!

Medidas de rendimiento

- Longitud del rendimiento mitad del asintótico ($N_{1/2}$)
  - Longitud de los vectores operandos para la que se obtiene la mitad del rendimiento asintótico.
- Ejemplo: Sup que se obtiene con n<MVL ➔ 1 iteración

  ➔ $\left\lceil \frac{n}{MVL} \right\rceil = 1$

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime} = 1 \times (15 + 49) + n \times 3 =$$

$$= 64 + 3n$$

$$En\ el\ ejemplo\ \ \frac{R_\infty}{2} = \frac{1}{4}\frac{FLOP}{Ciclo}$$

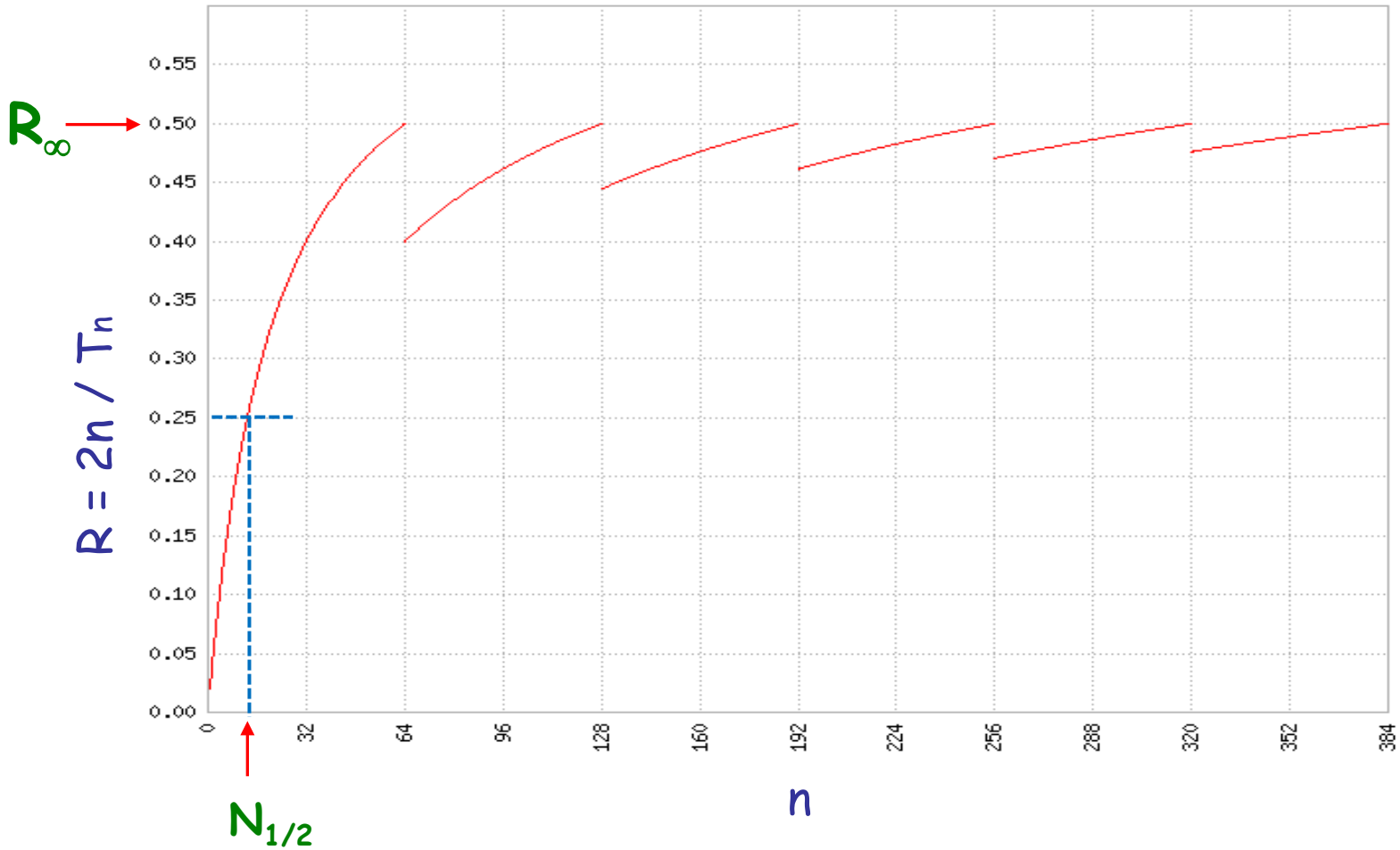
$$Por\ def\ R = \frac{2n}{T_n}; Sustituyendo: \frac{1}{4} = \frac{2n}{64 + 3n};\ \ n = 12.8;\ \ N_{1/2} = 13$$

Es decir, con vectores de sólo 13 componentes ya se obtiene un rendimiento que es la mitad del rendimiento asintótico

Medidas de rendimiento

Ejemplo: Rendimiento obtenido (FLOP/ciclo) en función de n

$$R = \frac{2n}{T_n} = \frac{2n}{\left\lceil \frac{n}{64} \right\rceil \times 64 + 3 \times n}$$



Operaciones condicionales: registro de máscara

- Ejecutar

```
for (i = 0; i < 64; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
```

- El registro de máscara vectorial permite omitir las operaciones sobre los elementos que no cumplen la condición

```
LV        V1,Rx      ;load vector X into V1
LV        V2,Ry      ;load vector Y
L.D       F0,#0      ;load FP zero into F0
SNEVS.D   V1,F0      ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D   V1,V1,V2   ;subtract under vector mask
SV        Rx,V1      ;store the result in X
```

- Obviamente, el rendimiento se reduce
  - Se consume el mismo tiempo
  - Se ejecutan menos operaciones útiles

Bancos de memoria

- El sistema de memoria debe soportar un elevado ancho de banda en la carga y almacenamiento de vectores
- Idea fundamental: Dispersar los accesos entre múltiples bancos
  - Control independiente de las direcciones en cada banco
  - Load y store de palabras no secuenciales
  - Soporte para que múltiples procesadores vectoriales puedan compartir la misma memoria

- Ejemplo:
  - 32 procesadores, cada uno generando 4 loads y 2 stores por ciclo
  - Tiempo de ciclo del procesador: 2.167 ns, tiempo de ciclo de la SRAM: 15 ns
  - ¿Cuántos bancos de memoria serían necesarios?

H&P 5th ed., p. 277

Vectores no consecutivos en memoria

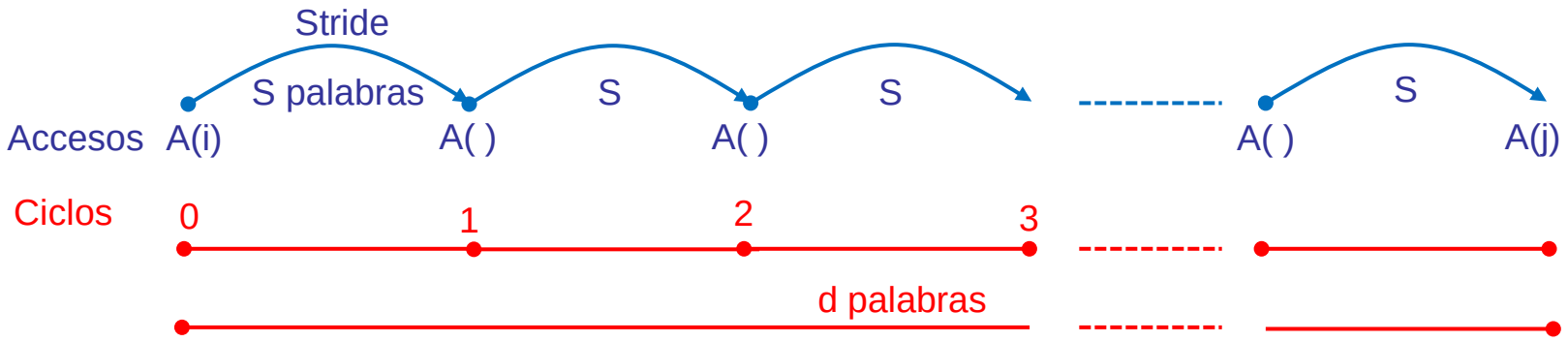
- Ejemplo: Producto de matrices

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

- Las matrices están ordenadas por filas
  - Para vectorizar el producto de filas de B por columnas de D
    - Acceso a elementos consecutivos de B
    - Acceso a elementos de D separados por 100 palabras de distancia
- Soporte: Instrucciones de load/store con "stride" (espaciamiento)
  - Problema: Repetición de acceso al mismo banco de memoria antes del fin de la op anterior
  - Acceso libre de conflicto si:
    - mcm (espaciamiento, nº bancos) / espaciamiento ≥ T acceso a banco

Vectores no consecutivos en memoria

- Acceso a array A con espaciamiento S palabras. Objetivo: un acceso por ciclo sin conflictos



Conflicto entre los accesos a A(i) y A(j) si

- A(i) y A(j) están en el mismo banco, o sea, d es múltiplo del Nº bancos y
- cuando comienza el acceso a A(j) todavía no ha acabado el acceso a A(i)

Además, por construcción d es múltiplo de S

¿Cuándo ocurre por primera vez que d es múltiplo de Nº bancos y de S ?
Cuando d = mcm (S, Nº bancos)

¿Cuántos ciclos de reloj han pasado? Han pasado: mcm (S, Nº bancos) / S ciclos

Si para ese momento el acceso a A(i) ya ha acabado, el acceso a A(j) podrá realizarse sin problema ➔ Acceso sin conflicto si: **mcm (S, Nº bancos) / S ≥ Tacceso a banco**

Caso particular. Si S y Nº bancos son primos entre si, entonces para que no haya conflictos de acceso basta que: Nº bancos ≥ Tacceso a banco

Vectores dispersos

- Los elementos no equidistan: utilización de vectores de índices
- **Carga.** Registro vectorial ← elementos dispersos de un array (compresión - *gather*): LVI V1, (R1+V2)
  - V2 indica las direcciones de los elementos a cargar en V1 como desplazamientos respecto de una dir inicial indicada por R1
- **Almacenamiento.** Llevar contenido de un registro vectorial a posiciones dispersas de un array (expansión - *scatter*):
  SVI (R1+V2), V1
- Ejemplo
  - Código fuente: sólo se accede a algunos elementos de A y C

```
for (i = 0; i < n; i=i+1)
        A[K[i]] = A[K[i]] + C[M[i]];
```

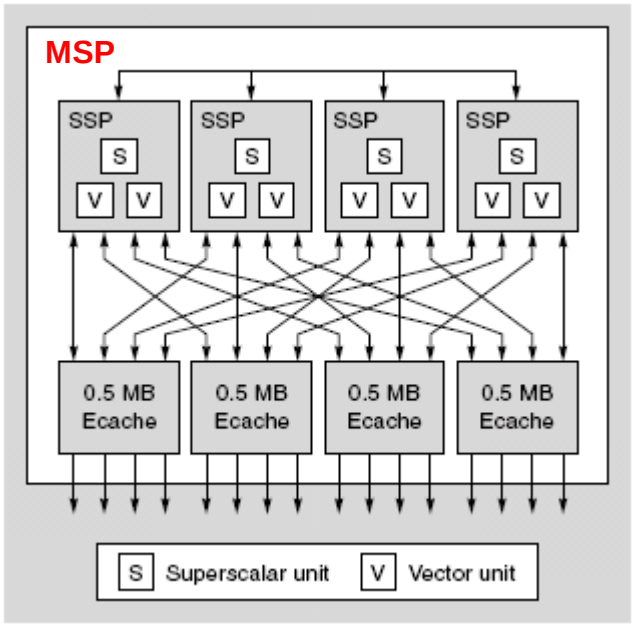
  - Código máquina: Vk y Vm usados como índices

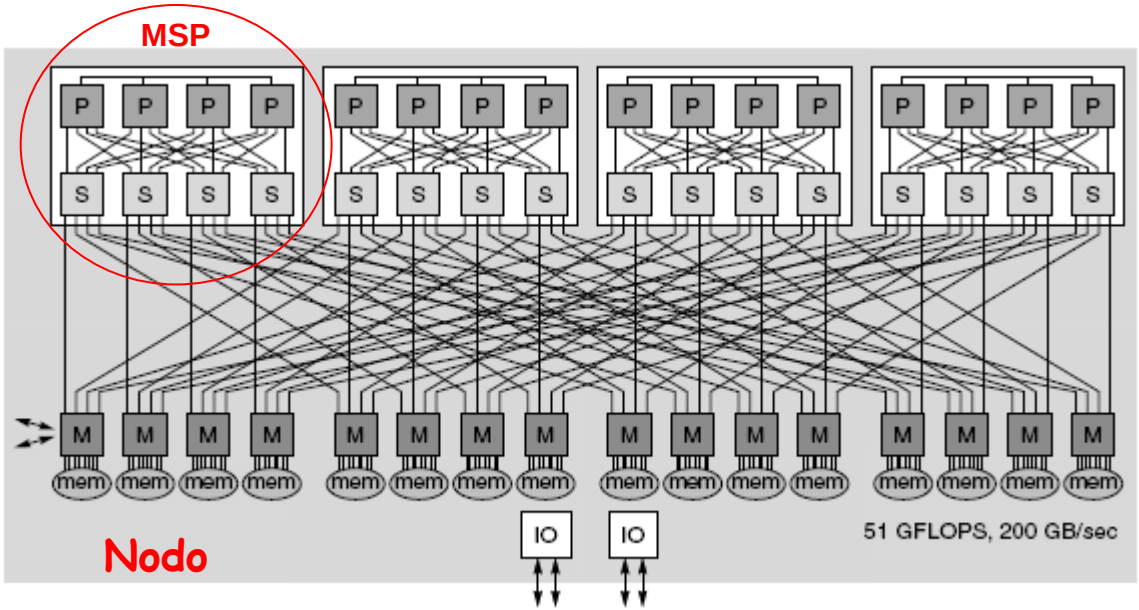
```
LV        Vk, Rk        ;load K
LVI       Va, (Ra+Vk)   ;load A[K[ ]]
LV        Vm, Rm        ;load M
LVI       Vc, (Rc+Vm)   ;load C[M[ ]]
ADDVV.D   Va, Va, Vc    ;add them
SVI       (Ra+Vk), Va   ;store A[K[ ]]
```

Ejemplo de Supercomputador Vectorial: Cray X1

Lectura: sección G.7 de H&P 5th ed.

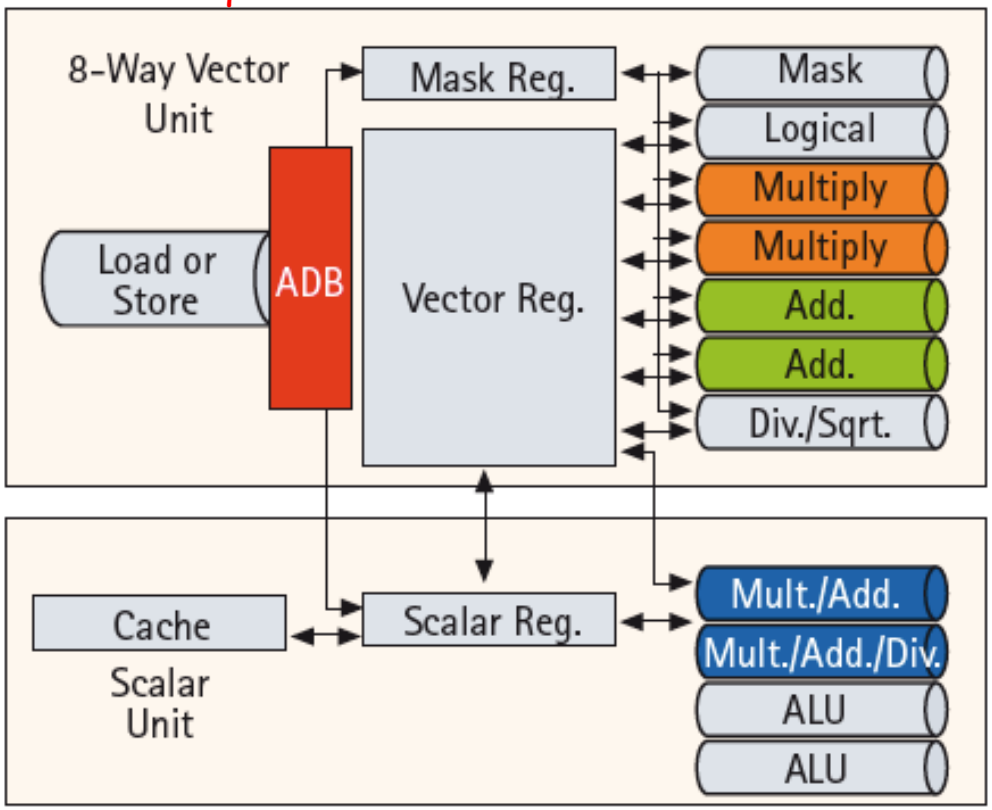
- Introducido en 2002
- Elemento básico
  - Single-Streaming Processor (SSP): μP vectorial con 2 vías + O-o-O procesador superescalar. Cada vía: 1 suma + 1 prod FP por ciclo.
- MSP: formado por 4 SSP. Rend pico: 12.8 GFLOPS
- Ecache: AB 1 palabra por vía por ciclo a 800 MHZ (sobre 50 GB/s)
- Nodo: formado por 4 MSP, 16 controladores de memoria, DRAM (AB 200 GB/s)
- Sistema: hasta 1024 nodos. Red global de alta velocidad. Un único espacio de direcciones. (4096MSPs, 16K SSPs, 52,2 TFLOPs)





Ejemplo de Supercomputador Vectorial: NEC SX-9

Esquema de 1 CPU

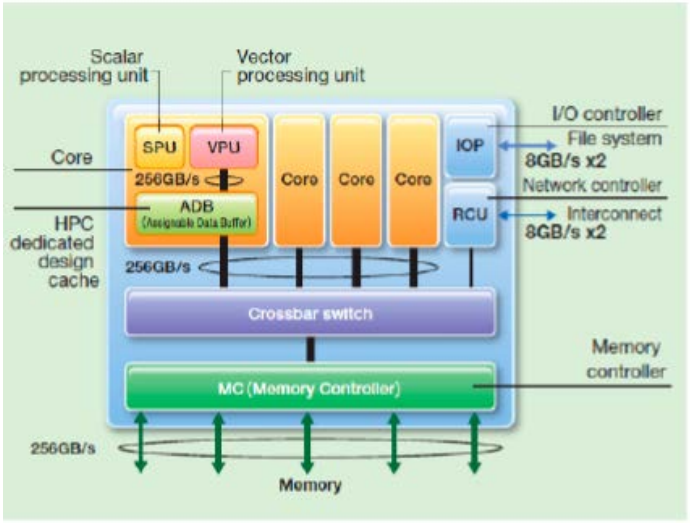


- Introducido en 2008
- Tecnología 65nm CMOS
- Unidad Vectorial (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elementos/VReg)
  - UFs de 64-bits: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 8 vías (32+ FLOPS/cycle, 100+ GFLOPS pico por CPU)
  - 1 load or store unit (8 x 8-byte accesos/ciclo)
- Unidad escalar (1.6 GHz)
  - Superescalar 4 vías O-o-O
  - 64KB I-cache, 64KB data cache
- AB Memoria: 256GB/s por CPU
  - Hasta 16 CPUs y hasta 1TB de DRAM forman un **nodo** con mem compartida
  - AB total: 4TB/s a la DRAM compartida
- Hasta 512 nodos conectados mediante enlaces de 128 GB/s (Paso de mensajes entre nodos) 800 TFLOPs

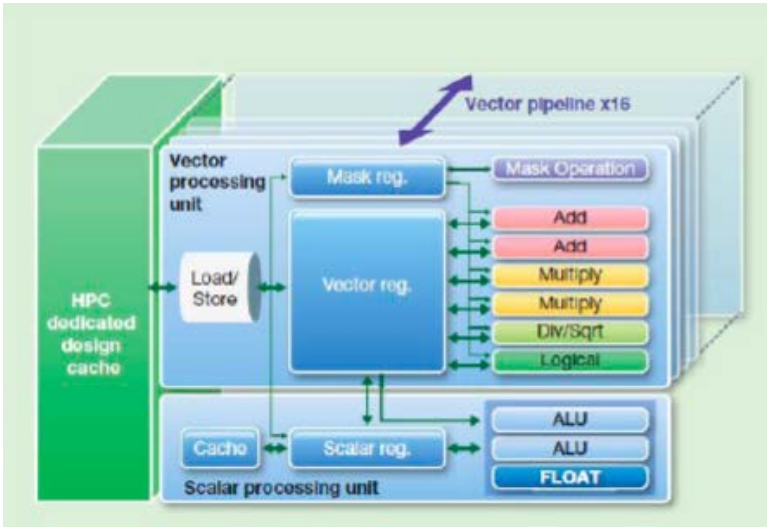
Ejemplo de Supercomputador Vectorial: NEC SX-ACE

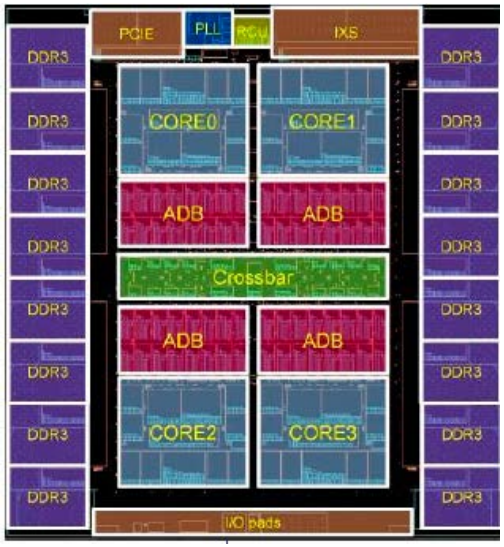
- Introducido en 2014, Tecnología 28nm, 1Ghz, 570mm^2, 2000 MTrans

Chip

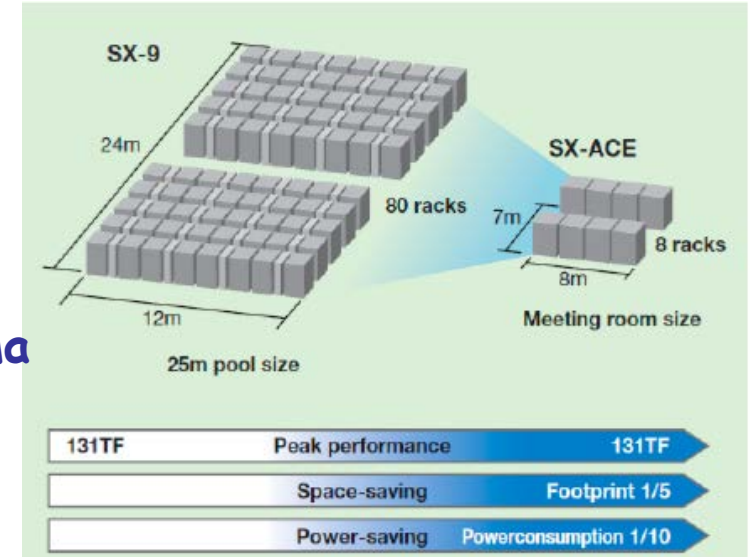


Core





Sistema



Ejemplo de Supercomputador Vectorial: NEC Aurora Vector

Vector Engine Processor Overview

SX-Aurora TSUBASA

Components

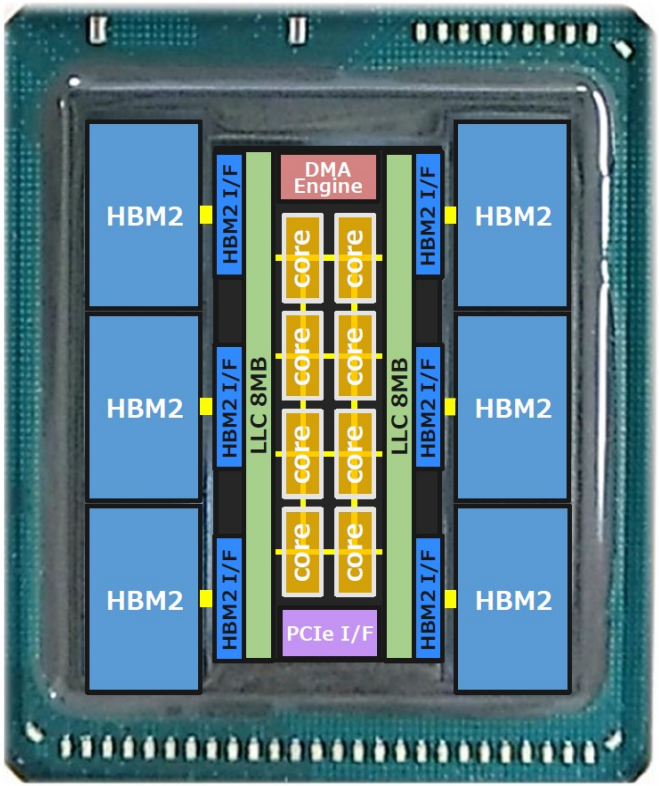
- 8 vector cores
- 16MB LLC
- 2D mesh network on chip
- DMA engine
- 6 HBM2 controllers and interfaces
- PCI Express Gen3 x16 interface

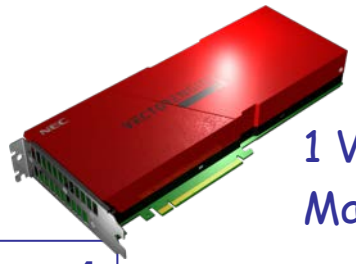
Specs

Core frequency	1.6GHz
Core performance	307GF(DP) 614GF(SP)
CPU performance	2.45TF(DP) 4.91TF(SP)
Memory bandwidth	1.2TB/s
Memory capacity	24/48GB

Technology

- 16nm FinFET process



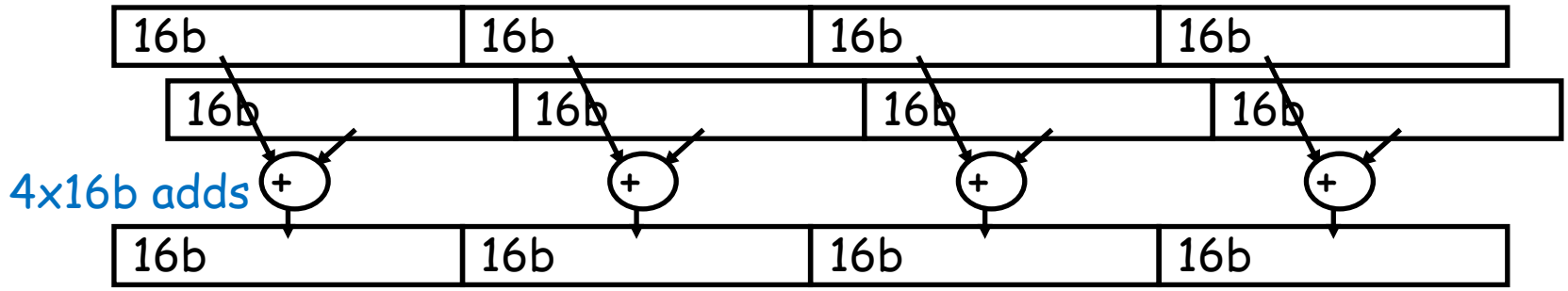


1 Vector Engine
Max 2.45 TF (DP)

64 Vector Engines
Max 157 TF (DP)

Extensiones SIMD

- También conocidas como "extensiones multimedia"
- Observación: las aplicaciones multimedia suelen operar sobre datos de menor anchura que las UFs y los registros disponibles
- Idea: Realizar varias operaciones a la vez
  - Por ejemplo: desconectar la cadena de propagación de carries
  - Una sola instrucción de suma opera sobre varios elementos almacenados en un registro ➔ equivale a una op vectarial, aunque con un vector de pocos elementos

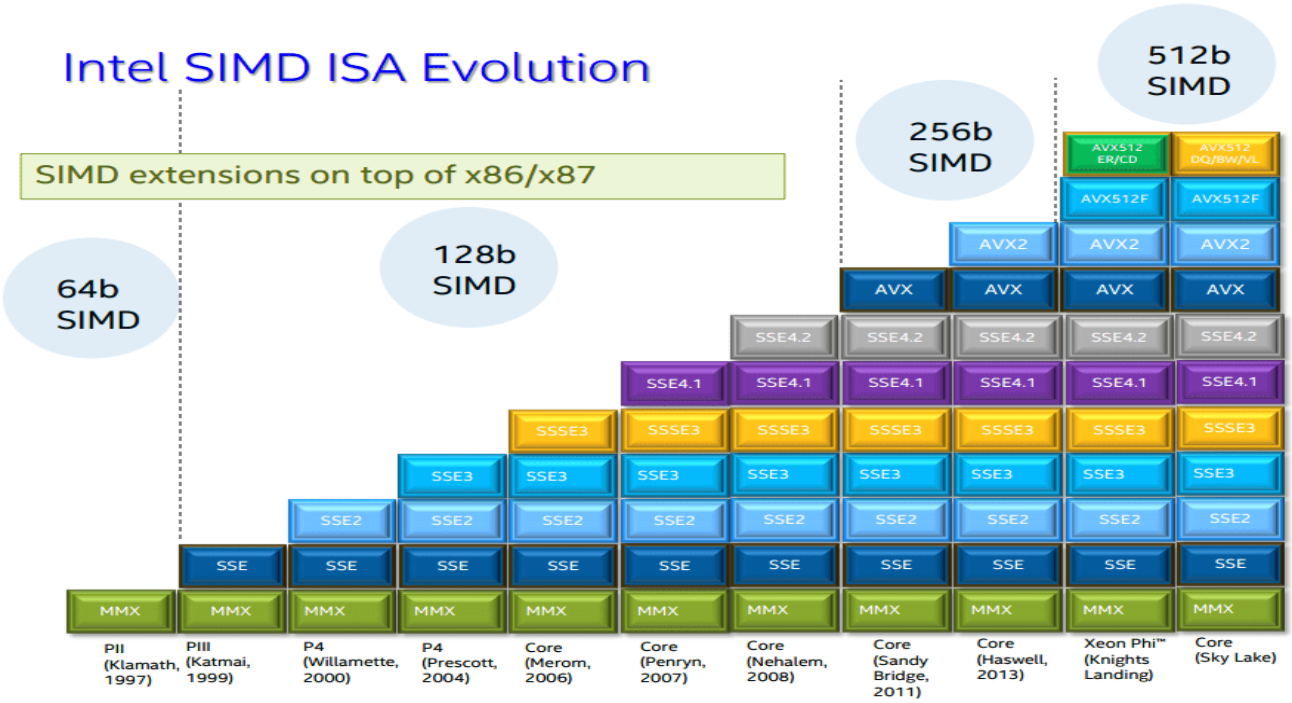


- Limitaciones, comparado con op vectoriales:
  - La longitud de los vectores se codifica en el Cod_op
  - No existe direccionamiento sofisticado (stride, gather,...)
  - No hay reg de máscara

Lectura: sección 4.3 de H&P 5th ed.

Extensiones SIMD

- Implementaciones:
  - Intel MMX (1996) 64bits
    - Ocho ops enteras de 8-bit o cuatro ops enteras de 16-bit
    - Usa los registros de PF
  - Streaming SIMD Extensions (SSE- SSE4) (1999-2007) 128bits
    - Ocho ops enteras de 16-bit
    - Cuatro ops enteras/FP de 32-bit o dos ops enteras/FP de 64-bit
    - Añade registro propios de 128bits
  - Advanced Vector Extensions (AVX)(2010) 256bits
    - Cuatro ops enteras/FP de 64-bit
  - AVX-512 (2016-17) XeonPhi , Skylake y Xeon Scalable 512bits
    - Ocho ops enteras/FP de 64-bit
  - Los operandos deben ser consecutivos y en posiciones de memoria alineadas



❑ Ejemplo: Instrucciones AVX para la arquitectura x86

4 operandos de 64 bits

AVX Instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD	Multiply and subtract four packed double-precision operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ..
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one double-precision operand to four locations in a 256-bit register

Como la longitud de los operandos va indicada en el Cod_op, puede dar la impresión de que el nº de instr de las "extensiones multimedia" es mayor de lo que en realidad es.

Ejemplo: código con extensiones SIMD en MIPS

- Sup: Añadimos instrucciones multimedia SIMD de 256 bits al MIPS (".4D" ➔ op sobre 4 operandos de 64 bits a la vez)

- Código para DAXPY:

```
        L.D F0,a                ;load scalar a
        MOV F1, F0              ;copy a into F1 for SIMD MUL
        MOV F2, F0              ;copy a into F2 for SIMD MUL
        MOV F3, F0              ;copy a into F3 for SIMD MUL
        DADDIU R4,Rx,#512       ;last address to load
Loop:   L.4D F4,0(Rx)           ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D F4,F4,F0         ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D F8,0(Ry)           ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D F8,F8,F4         ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D F8,0(Ry)           ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU Rx,Rx,#32        ;increment index to X
        DADDIU Ry,Ry,#32        ;increment index to Y
        DSUBU R20,R4,Rx         ;compute bound
        BNEZ R20,Loop           ;check if done
```

Unidades para Procesamiento Gráfico (GPUs)

- Las GPUs son económicas, accesibles y contienen una gran cantidad de elementos de cómputo.
- Se han concebido con el objeto de realizar los procesamientos característicos de las aplicaciones gráficas
- ¿Cómo poder utilizar la gran potencia de los procesadores gráficos en un espectro de aplicaciones más amplio?
- Idea básica
  - Modelo de ejecución heterogéneo (CPU+GPU)
  - Desarrollar un lenguaje de programación tipo C que permita programar la GPU
  - Unificar todo el paralelismo de la GPU bajo la abstracción denominada "CUDA Thread"
  - Modelo de Programación: "Single Instruction (SIMD) Multiple Thread"

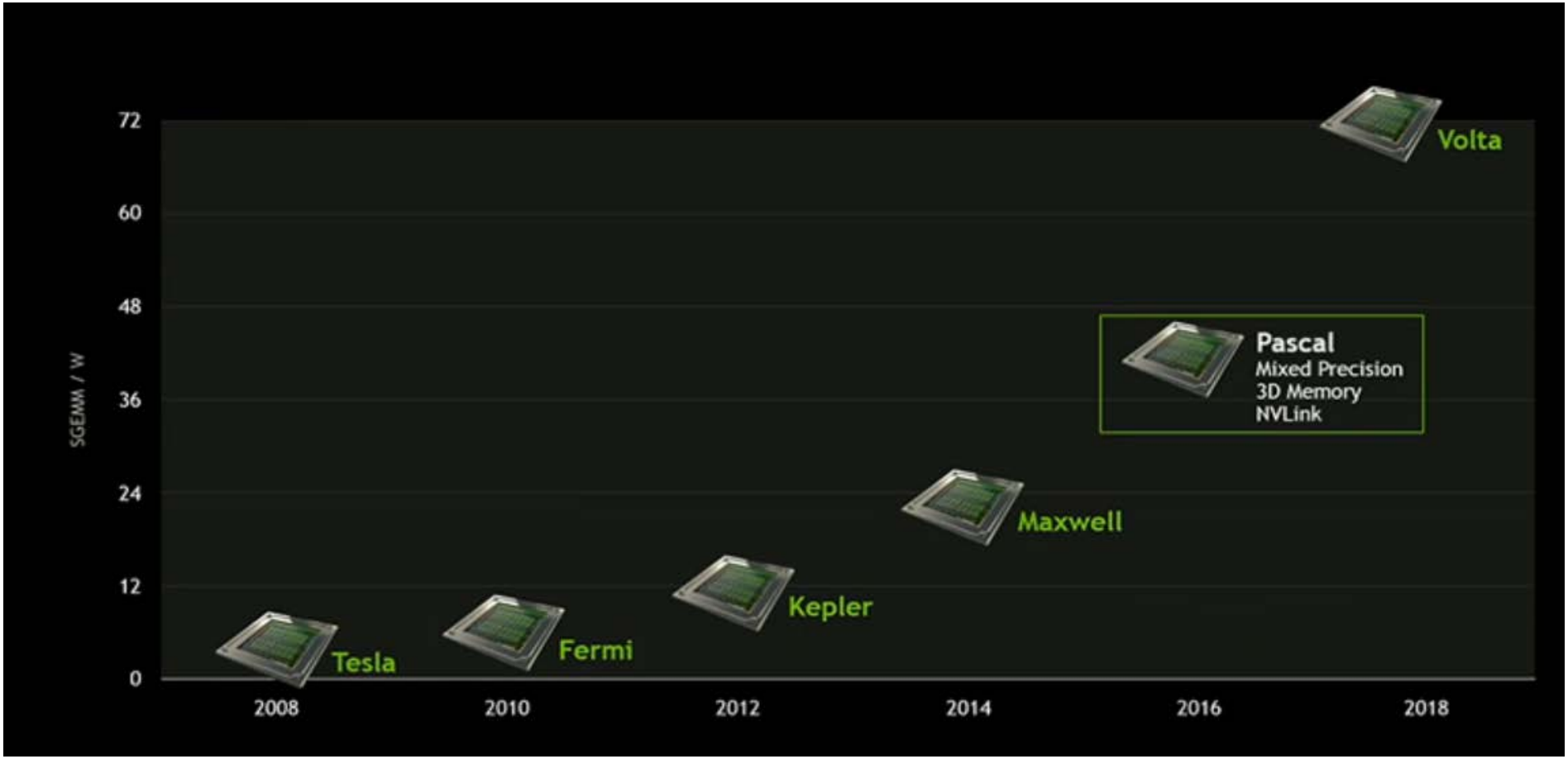
Lectura: sección 4.4 de H&P 5th ed.

Arquitectura de NVIDIA GPU

- GPU NVIDIA
  - Multiprocesador compuesto por un conjunto de procesadores SIMD MT
- NVIDIA vs procesadores vectoriales
  - Similaridades
    - Funciona bien en problemas con paralelismo de datos
    - Transferencias con memoria tipo dispersar/reunir (scatter/gather)
    - Registros de máscara
    - Existencia de grandes ficheros de registros
  - Diferencias
    - No hay un procesador escalar
    - Utilización de multithreading para ocultar la latencia de memoria
    - Existencia de gran cantidad de UFs.
      - Contrasta con la reducida cantidad de UFs muy segmentadas, que es típica de los procesadores vectoriales

Arquitectura de NVIDIA GPU

- GPU NVIDIA
  - Pascal 10x Maxwell (4 Tflops DP, 8 Tflops SP)
  - 2018 Volta (7,5Tflops DP, 15Tflops SP)



CUDA(Compute Unified Device Architecture)

- CUDA is an elegant solution to the problem of representing parallelism in algorithms, not all algorithms, but enough to matter. It seems to resonate in some way with the way we think and code, allowing an easier, more natural expression of parallelism beyond the task level.

  Vincent Natoli
  "Kudos for CUDA", *HPC Wire* (July 2010)
  http://www.hpcwire.com/hpcwire/2010-07-06/kudos_for_cuda.html

- CUDA produce código C/C++ para host y dialecto de C y C++ para la GPU

  - Idea básica: crear un thread (hilo) separado para cada elemento de los vectores a procesar
    - Objetivo: generar un gran nº de hilos de cómputo independientes

  - Los threads se agrupan en bloques de threads
    - El número de threads por bloque puede definirlo el programador
    - Cada bloque es ejecutado por un procesador SIMD MT de la GPU
    - Varios bloques pueden ejecutarse en paralelo sobre varios procesadores

  - El conjunto de bloques que implementan un cálculo vectorial sobre la GPU se denomina Grid (malla). La ejecución del cálculo se produce con una llamada similar a una función en C:

    - **nombre_función** <<<dimGrid,dimBlock>>> **(... lista de parámetros ...)**
      - dimGrid: nº de bloques en el Grid
      - dimBlock: nº de threads por bloque
      - Los bloques y las mallas pueden tener hasta 3 dimensiones, que se identifican con .x , .y, .z.

CUDA vs C. Ejemplo DAXPY

- Versión C

```
// Invocar DAXPY
daxpy(n, 2.0, x, y);

// DAXPY en C (bucle escalar: una iteración por elemento)
void daxpy(int n, double a, double *x, double *y)
{
        for (int i = 0; i < n; i++)
                y[i] = a*x[i] + y[i];
}
```

CUDA vs C. Ejemplo DAXPY

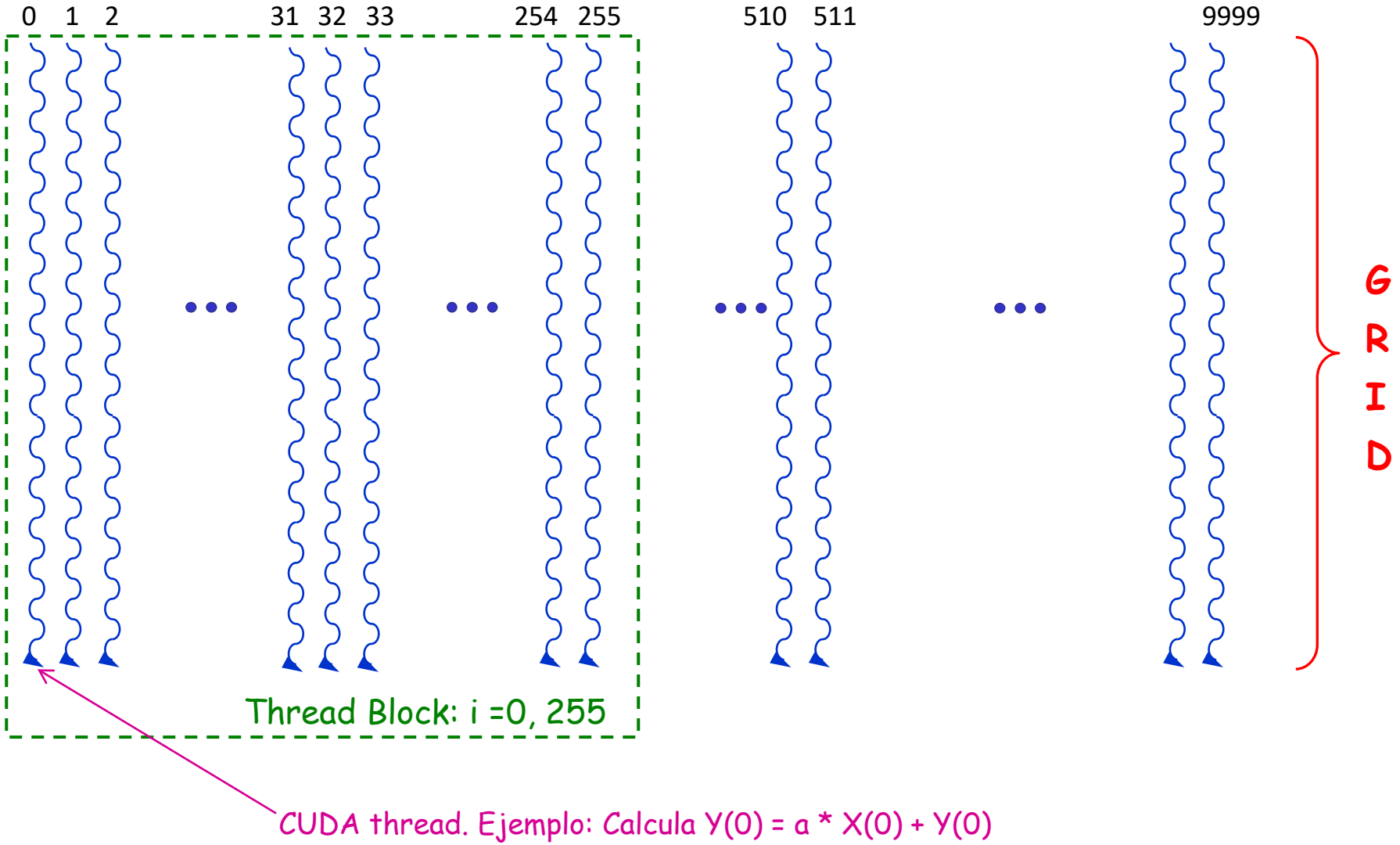
Versión CUDA

```
// Invocar DAXPY con 256 threads por Bloque (dimBlock)
__host__                              /* código para la CPU */
int nblocks = (n+ 255) / 256;    /*  cálculo del nº total de bloques en el Grid (dimGrid) */
daxpy <<<nblocks, 256>>> (n, 2.0, x, y);

// DAXPY en CUDA (representa el cálculo ejecutado para un elemento)
__device__                            /* código para los procesadores de la GPU */
void daxpy(int n, double a, double *x, double *y)
{
// ¿Qué thread soy? Calcular i = nº elemento del vector a procesar (= nº de thread), siendo
// nº elemento = (nº de bloque x tamaño de bloque) + (nº de thread dentro del bloque)
int i = blockIdx.x*blockDim.x + threadIdx.x;

// Si el nº elemento obtenido es mayor que el tamaño del vector, ignorar operación
if (i < n) y[i] = a*x[i] + y[i];
}
```

CPU

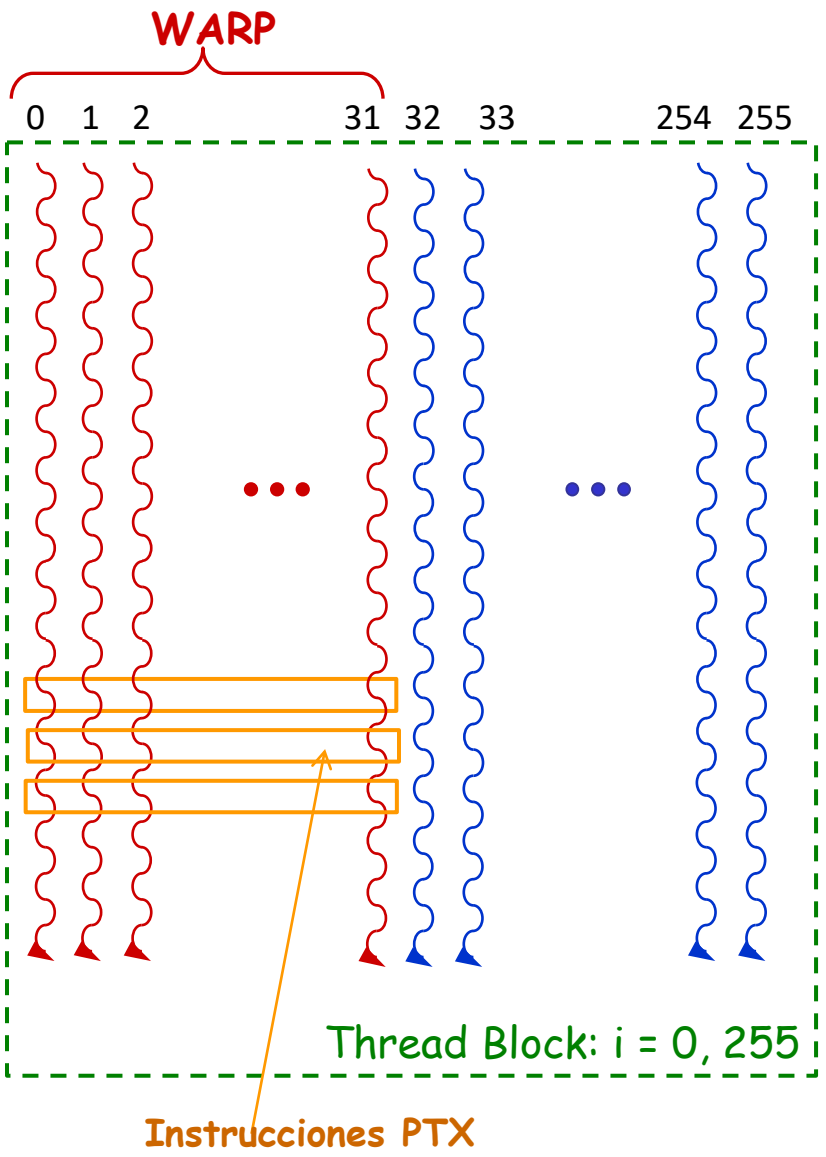
GPU

- Ejemplo: vectores de 10,000 componentes



Thread blocks e Instrucciones PTX

- Cada Thread Block se ejecuta en un procesador SIMD MT de la GPU.
- Varios procesadores SIMD MT de la GPU pueden procesar diferentes Thread Blocks en paralelo.
- **Instrucción PTX**: Ejecuta un mismo cálculo sobre varios (e.g. 32) datos (Instr SIMD).
  - Resultados afectados por registro de máscara.
- **WARP**: Secuencia (thread) de instr PTX. El procesador ejecuta los WARP de un Thread Block en modo MT.
  - En el ejemplo hay 256/32 = 8 WARPs .
  - Cambios de thread ocultan latencias de acceso a memoria



Código generado por compiladores de NVIDIA

- Instrucciones PTX (Parallel Thread Execution)
  - Abstracción del repertorio de instrucciones hw
  - Formato: opcode.type dest, src1, scr2, src3
  - Instrucción que ejecuta una operación elemental sobre múltiples datos (SIMD) utilizando todas la vías del procesador
  - Ejemplo: un conjunto de instrucciones PTX representativas

Instruction	Example	Meaning	Comments
arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
add.type	add.f32 d, a, b	d = a + b;	
sub.type	sub.f32 d, a, b	d = a – b;	
mul.type	mul.f32 d, a, b	d = a * b;	
mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
mov.type	mov.b32 d, a	d = a;	move
selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space

Código generado por compiladores de NVIDIA

- Ejemplo: secuencia de instrucciones PTX para una iteración del blucle DAXPY
  - Usa reg virtuales: Ri (32 bits), RDi (64 bits)
  - Asigna reg físicos en el momento de la carga del programa

```
shl.u32      R8, blockIdx, 8      ; Thread Block ID * Block size (256 or 2^8)
add.u32      R8, R8, threadIdx    ; R8 = i = my CUDA Thread ID
shl.u32      R8, R8, 3            ; byte offset
ld.global.f64  RD0, [X+R8]        ; RD0 = X[i]
ld.global.f64  RD2, [Y+R8]        ; RD2 = Y[i]
mul.f64      RD0, RD0, RD4        ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64      RD0, RD0, RD2        ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64  [Y+R8], RD0        ; Y[i] = sum (X[i]*a + Y[i])
```

Ojo! Recordar que cada instrucción PTX procesa 32 elementos (1 WARP de 32 threads)

Resumen de terminología: arquitectura vectorial vs. GPUs

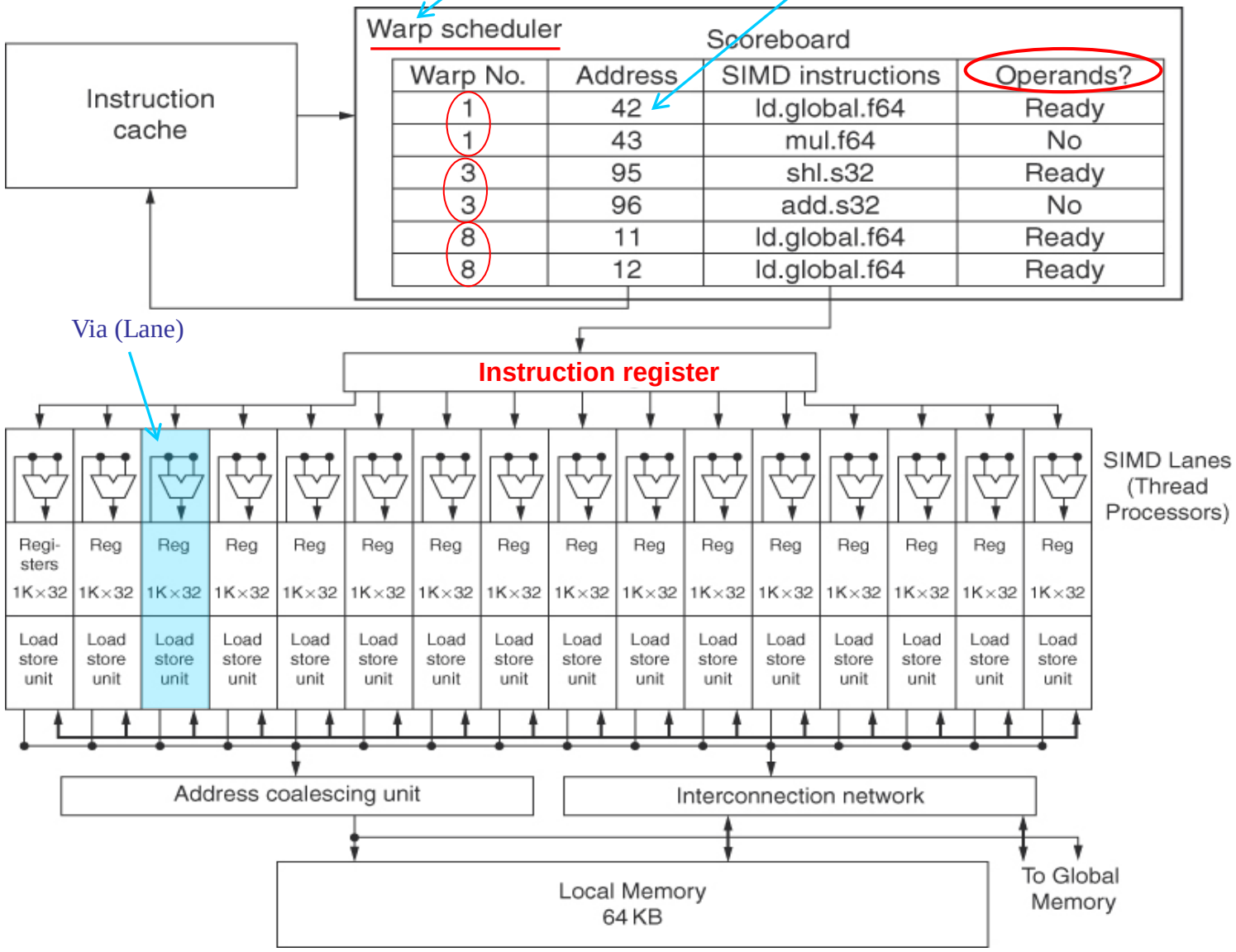
Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid (Malla) Ej. i = 0..9999	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block Ej. i = 0..255	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread Ej. i = 12	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp (Trama)	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction Ej. i = 0..31	A single SIMD instruction executed across SIMD Lanes.

Procesadores de una GPU

❑ Procesador SIMD MT

(aka SIMD Thread scheduler)

Cada Warp (thread de instrucciones SIMD) tiene su PC

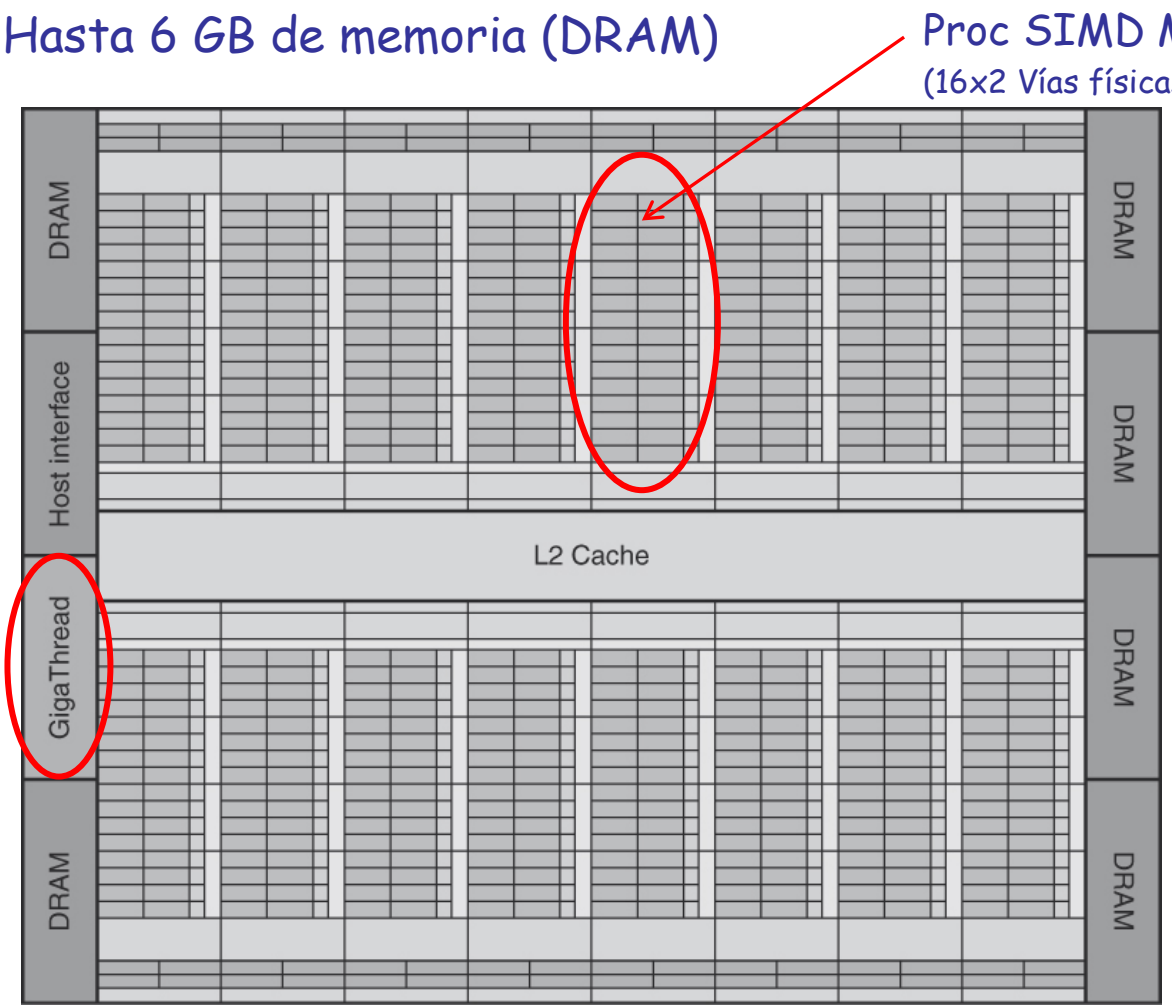


- Todas las vías ejecutan la misma instrucción
- Según la máscara, unas guardan el resultado y otras no

16 Lanes

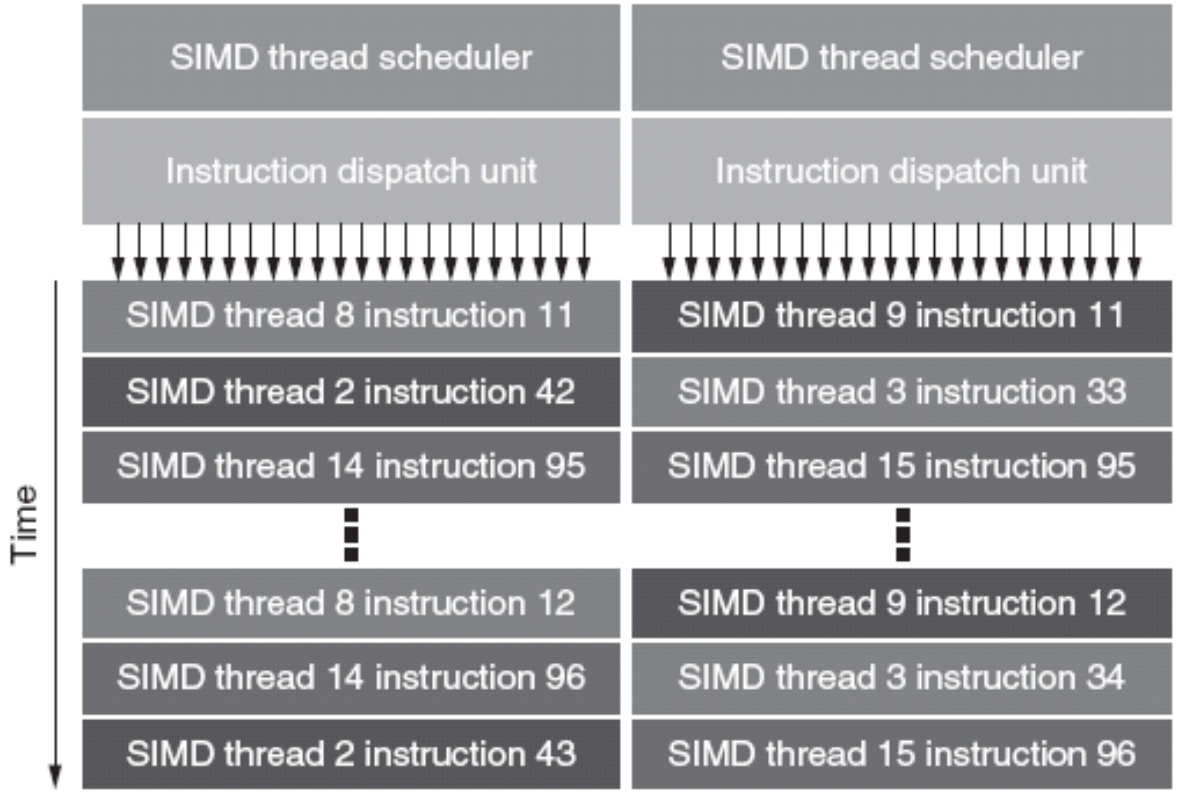
Procesadores de una GPU

- Ejemplo: Fermi GTX 480 (16 procesadores SIMD MT)
  - GigaThread (aka Thread Block scheduler): Distribuye bloques a procesadores
  - Hasta 6 GB de memoria (DRAM)



Procesadores de una GPU

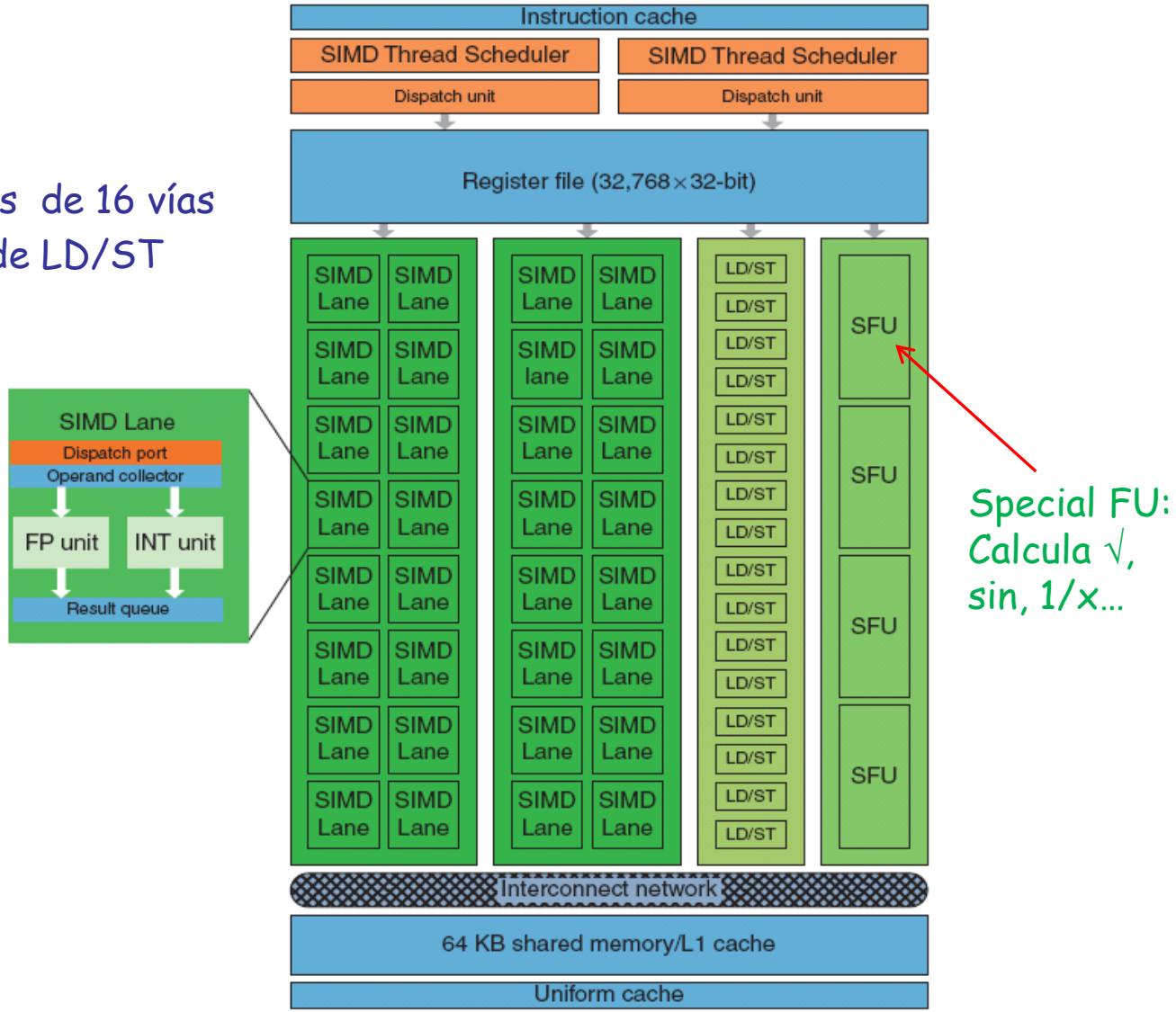
- Fermi: Thread Scheduler
  - Cada procesador: 2 thread schedulers en paralelo (en Kepler 4)
  - Selecciona una instrucción de cada thread y las envía a dos conjuntos de 16 vías físicas
  - Cada instrucción SIMD procesa 32 elementos (y se ejecuta sobre 16 vías físicas) ➔ 2 ciclos por instrucción



Procesadores de una GPU

❑ Fermi: Esquema de un procesador SIMD MT

- Dos conjuntos de 16 vías
- 16 unidades de LD/ST



Saltos condicionales en GPUs

- Gestión de registros de máscara (predicado) similar a los procesadores vectoriales.
  - Para componentes enmascaradas, el resultado no se guarda en el registro destino
  - Permite implementar construcciones IF...THEN mediante una instrucción PTX "compare and set predicate" (setp)
  - Construcciones IF... THEN...ELSE: mecanismo similar, pero para la parte ELSE el registro de máscara se complementa.
  - Impacto en rendimiento
  - Ejemplo:

```
if (i < n)
        j = j + 1;
```

Se puede implementar mediante:

```
        setp.lt.s32 p, i, n;        // p = (i < n)
@p      add.s32     j, j, 1;        // if i < n, add 1 to j
```

Denota ejecución bajo control de un registro de predicado

Identifica el registro de predicado. En la instrucción add solo se guarda el resultado para los threads en los que el bit p(i) es "cierto"

Saltos condicionales en GPUs

- Además existen instrucciones PTX de salto: Permiten implementar construcciones condicionales mediante saltos verdaderos en el código.
  - Formato: @p branch target
    - Para los threads tales que el bit p(i) es "falso" el flujo de cálculos continua en secuencia.
    - Para los threads tales que el bit p(i) es "cierto" se produce una salto a la instrucción de etiqueta "target". Estos threads no realizan trabajo hasta que se finalizan los primeros. Entonces todos los threads se vuelven a sincronizar.
    - Ejemplo:

```
if (i < n)
    j = j + 1;
```

se puede implementar también como:

```
            setp.lt.s32   p, i, n;   // compare i to n
@!p         bra           L1;        // if False, branch over
            add.s32       j, j, 1;
L1: …
```

  - Además para preservar la máscara existente antes de entrar en un IF…THEN…ELSE, existe un stack de máscaras.
    - Apilar máscara (push) antes de entrar al IF…THEN…ELSE, desapilar (pop) al salir. Complementar (comp) máscara actual al entrar en parte ELSE
    - Marcadores de sincronización: *push, *pop, *comp
  - El flujo de programa externo al IF…THEN…ELSE no continua hasta que todas los threads han finalizado

Saltos condicionales en GPUs

- Ejemplo

```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];
```

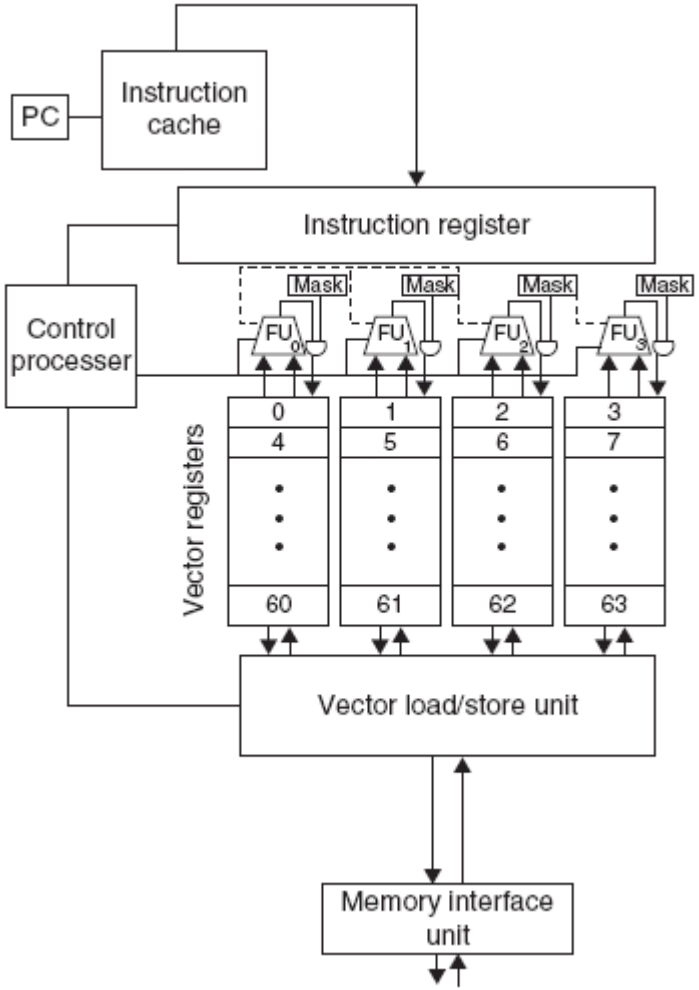
  - Código PTX

```
                                        ; R8 actualizado de acuerdo con Thread Id
        ld.global.f64   RD0, [X+R8]     ; RD0 = X[i]
        setp.neq.s32    P1, RD0, #0     ; P1 is predicate register 1
        @!P1  bra       ELSE1, *Push    ; Push old mask, set new mask bits
                                        ; if P1 false, go to ELSE1
        ld.global.f64   RD2, [Y+R8]     ; RD2 = Y[i]
        sub.f64         RD0, RD0, RD2   ; Difference in RD0
        st.global.f64   [X+R8], RD0     ; X[i] = RD0
        @P1  bra        ENDIF1, *Comp   ; complement mask bits
                                        ; if P1 true, go to ENDIF1
ELSE1:  ld.global.f64   RD0, [Z+R8]     ; RD0 = Z[i]
        st.global.f64   [X+R8], RD0     ; X[i] = RD0
ENDIF1: <next instruction>, *Pop        ; pop to restore old mask
```

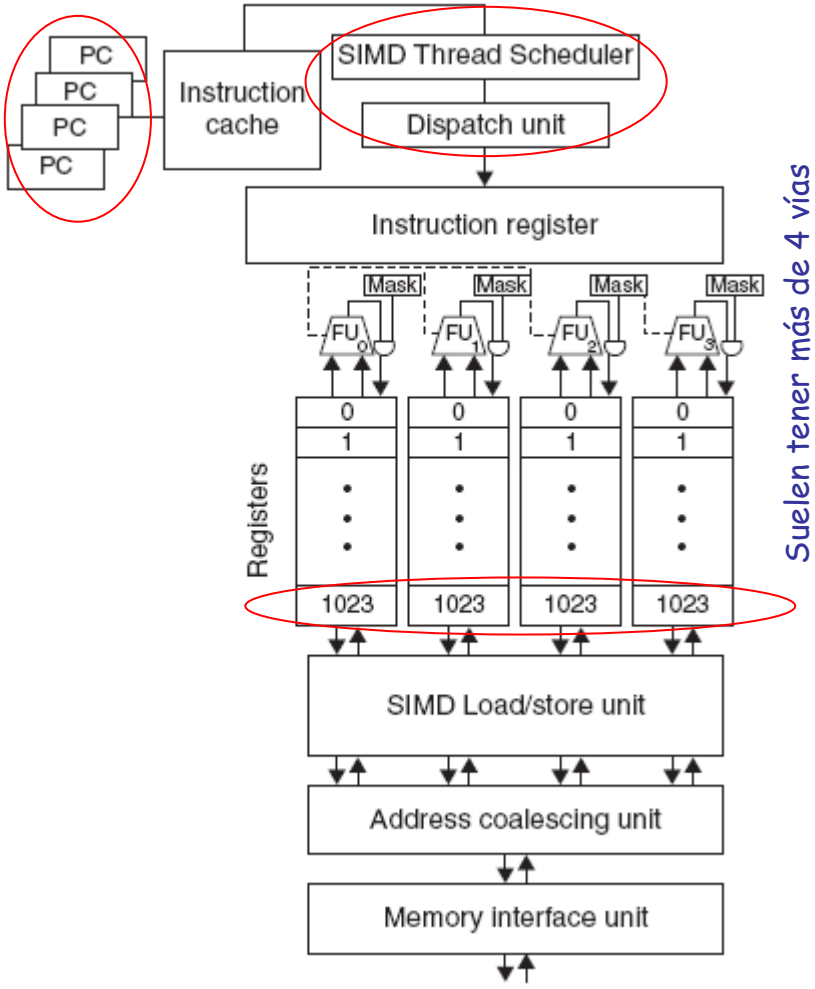
Ojo! Recordar que cada instrucción PTX procesa 32 elementos (1 WARP de 32 threads)

Comparación Procesador Vectorial - GPU

Procesador vectorial con cuatro vías



Procesador SIMD MT (4 PCs) con cuatro vías



Suelen tener más de 4 vías

Paralelismo a nivel de bucle: vectorización

- Las dependencias RAW entre sentencias de una misma iteración no impiden la vectorización eficiente (mediante encadenamiento de pipes)
  - Dependencia directa: obliga a ejecutar en el orden dado
- ¿Qué ocurre si los datos de una iteración son dependientes de los resultados generados en iteraciones previas?
  - Dependencia en el espacio de iteraciones (loop-carried)
  - Puede impedir la vectorización
  - Pueden existir reordenaciones de sentencias válidas
- Ejemplo 1

```
for (i=999; i>=0; i=i-1){
        x[i] = x[i] + s;        /* S1 */
        z[i] = z[i] + x[i];     /* S2 */
}
```

Dep directa: no impide la vectorización. Orden: 1º S1, luego S2

```
x[0:999] = x[0:999] + s;
z[0:999] = z[0:999] + x[0:999]
```

Paralelismo a nivel de bucle: vectorización

- Ejemplo 2:

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i];        /* S1 */
        B[i+1] = B[i] + A[i+1];      /* S2 */
}
```

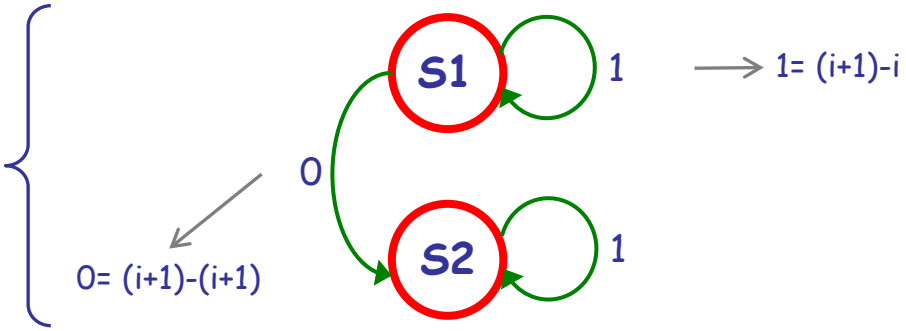
- S1 y S2 usan valores calculados por ellas mismas en la iteración previa ➔ ejecución en serie
- S2 usa resultados de S1 en la misma iteración.
  - Si ésta fuera la única dependencia (no loop-carried), el bucle sería vectorizable
  - En todo caso, el orden de ejecución de S1 y S2 debe mantenerse. Si se cambian de orden se altera la semántica del programa

Grafo de dependencias

Nodos: Sentencias

Arcos: Dependencias entre sentencias

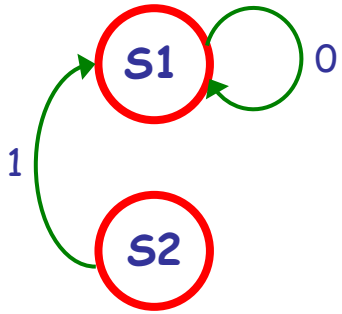
Etiq: Nº de iteraciones que separan dos operaciones dependientes



Paralelismo a nivel de bucle: vectorización

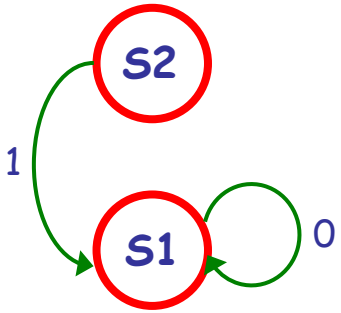
- Ejemplo 3

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i];        /* S1 */
        B[i+1] = C[i] + D[i];      /* S2 */
 }
```



- S1 usa un valor calculado por S2 en la iteración previa, pero la dependencia no es circular ➔ Vectorizable…Cómo?
- Única flecha hacia arriba tiene d>0 ➔ Transformable a (orden de sentencias inverso):

```
for (i=0; i<100; i=i+1) {
        B[i+1] = C[i] + D[i];      /* S2 */
        A[i] = A[i] + B[i];        /* S1 */
}
```



Caso particular de estudios matemáticos (Hans Zima, 1991). Si:
- todos los arcos entre distintas sentencias apuntan hacia abajo, y
- los arcos entre una sentencia consigo misma tienen etiq=0

entonces el bucle es vectorizable.

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Alternativa en H&P 5th ed.

Detección de dependencias

- Sup: los índices de los bucles toman valores de acuerdo con una función afín
  - a*i + b (siendo i el índice)
- Sup:
  - Almacenar en la posición a*i + b de un vector. Después:
  - Cargar desde la posición c*i + d del mismo vector
  - i toma valores desde m hasta n
- Existe una dependencia si:
  - Dados j, k tales que $m \le j \le n$, $m \le k \le n$
  - Almacenar en a*j + b, cargar desde a * k + d, y

$$a * j + b = c * k + d$$

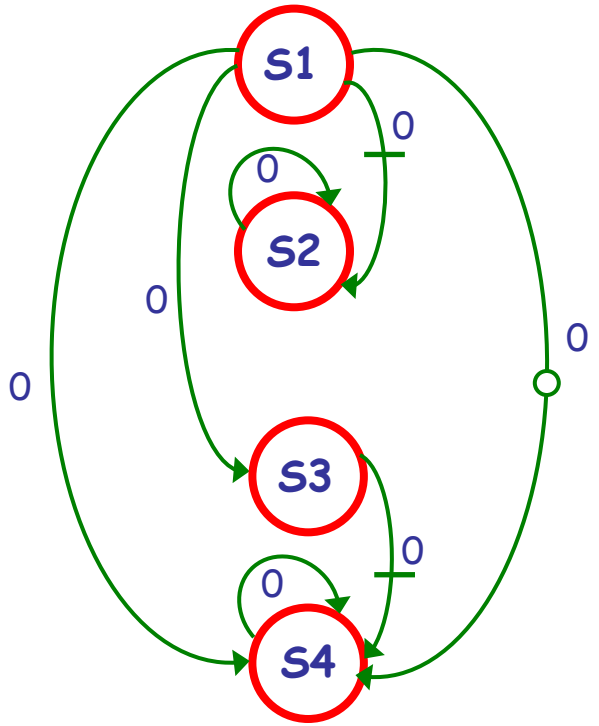
- Test del MCD
  - Si existe dep entonces MCD(c,a) es un divisor de (d-b)
- Ejemplo 4:

```
for (i=0; i<100; i=i+1) {
        X[2*i+3] = X[2*i] * 5.0;
}
```

  - a=2, c=2, b=3, d=0. MCD(2,2)=2. d-b=-3 ➔ No dependencia

Antidependencias y dependencias de salida

- Las dependencias de nombre pueden evitarse renombrando variables
- Ejemplo 5




```
for (i=0; i<100; i=i+1) {
   Y[i] = X[i] / c;        /* S1 */
   X[i] = X[i] + c;        /* S2 */
   Z[i] = Y[i] + c;        /* S3 */
   Y[i] = c - Y[i];        /* S4 */
}
```

- Transformar a:

```
for (i=0; i<100; i=i+1 {
   T[i] = X[i] / c;     /* Y renamed to T to remove output dependence */
   X1[i] = X[i] + c; /* X renamed to X1 to remove antidependence */
   Z[i] = T[i] + c;    /* Y renamed to preserve true dependence*/
   Y[i] = c - T[i];
}
```

Reducciones

- Ejemplo de operación de reducción :

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];   /* no vectorizable */
```

- Transformar a…

```
for (i=9999; i>=0; i=i-1)
        sum [i] = x[i] * y[i]; /* vectorizable */

for (i=9999; i>=0; i=i-1)
        finalsum = finalsum + sum[i]; /* no vectorizable */
```

- Suma final se puede acelerar. Si tenemos 10 procesadores (p=0..9), procesar 1000 elementos en cada uno:

```
for (i=999; i>=0; i=i-1)
        finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- Se está asumiendo que la operación es asociativa