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 2024-2025

Contenidos

- Introducción
- Arquitectura vectorial
  - Repertorio de instrucciones vectoriales.
  - Configuración de los registros vectoriales.
  - Procesamiento selectivo de elementos.
  - Tiempo de ejecución, medidas de rendimiento.
  - Ejemplos
  - Paralelismo a nivel de bucle: vectorización.
- Instrucciones SIMD para procesamiento multimedia
- Unidades para procesamiento gráfico (GPUs)
  - Modelo de programación a alto nivel: CUDA
  - Instrucciones PTX
  - Arquitectura del elemento de proceso
- Bibliografía
  - Cap 4 y Apéndice G de Hennessy & Patterson, 6th ed., 2019
  - "Introduction to the RISC-V Vector Extension", Roger Ferrer, ACM Summer School on HPC and AI, 2022.

(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`

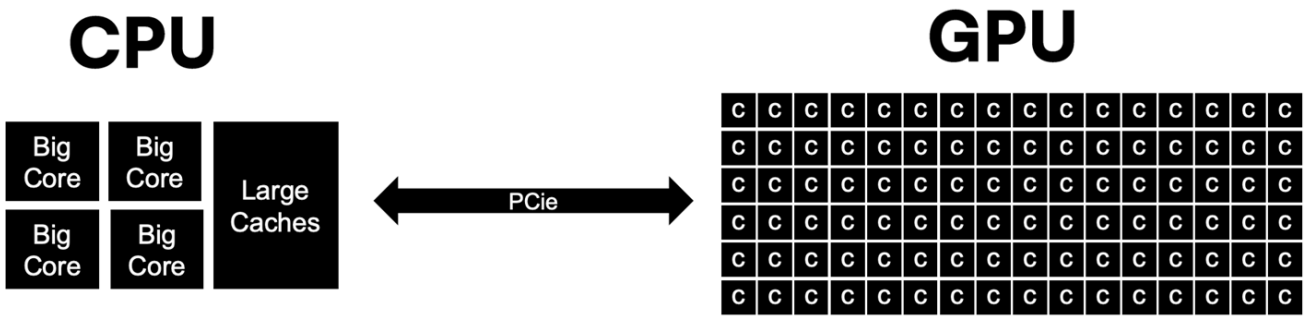
- Arquitectura SIMD: puede explotar una cantidad importante de paralelismo de datos en
  - Aplicaciones de ciencia/ingeniería con abundante cálculo matricial (ámbito tradicional)
  - Nuevas aplicaciones: gráficos, visión artificial, comprensión de voz, Deep Learning...

Introducción

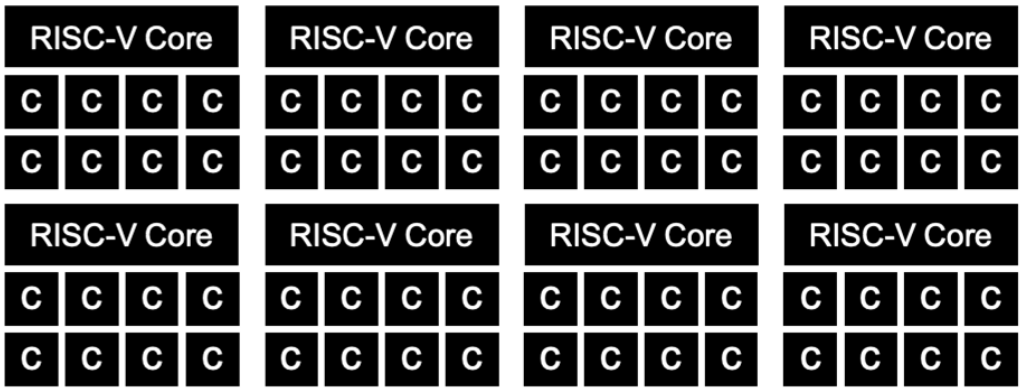
- Eficiencia energética de SIMD: puede ser ventajosa frente a MIMD
  - Sólo es preciso hacer un "fetch" para operar sobre varios datos.
  - Ahorro de energía atractivo en dispositivos portátiles
- En SIMD el programador sigue "pensando" básicamente en un flujo secuencial de instrucciones.
- Soporte arquitectónico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

Paralelismo a nivel de datos

La GPU es un acelerador:



Las arquitecturas vectoriales y multimedia integran la CPU + unidades vectoriales



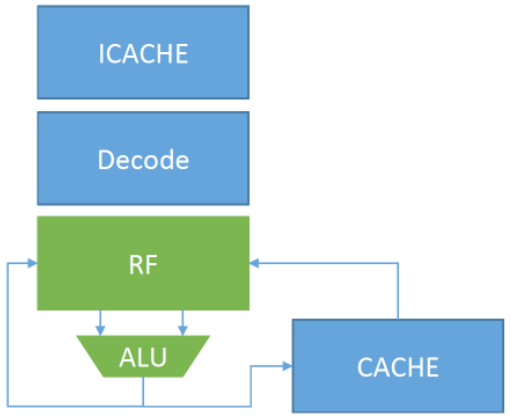
Arquitectura vectorial

- 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

Arquitectura vectorial

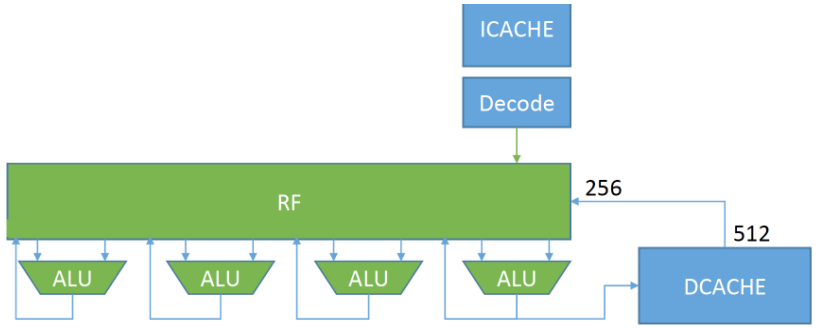
- Procesador (super)scalar

```
Load  (r1) -> r2
Load 4(r1) -> r3
Add r2, r3 -> r4
Store r4 -> (r1)
```

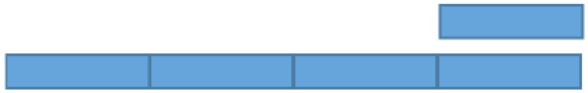


- Procesador Vectorial

```
vLoad  (r1) -> v2
vLoad 4(r1) -> v3
vAdd v2, v3 -> v4
vStore v4 -> (r1)
```



Scalar register 64 bits
Vector register 256 bits



Arquitectura vectorial RISC-V

El primero fue el Cray-1.

Actualmente hay dos arquitecturas vectoriales:

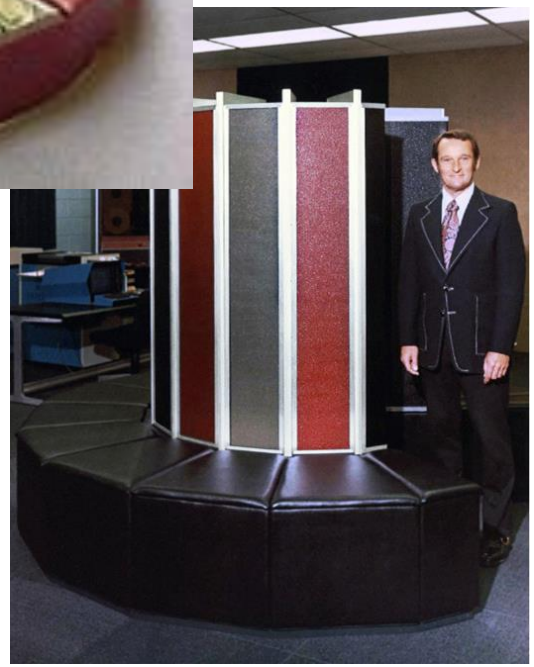
- ARM
- RISC-V V

Ventajas:

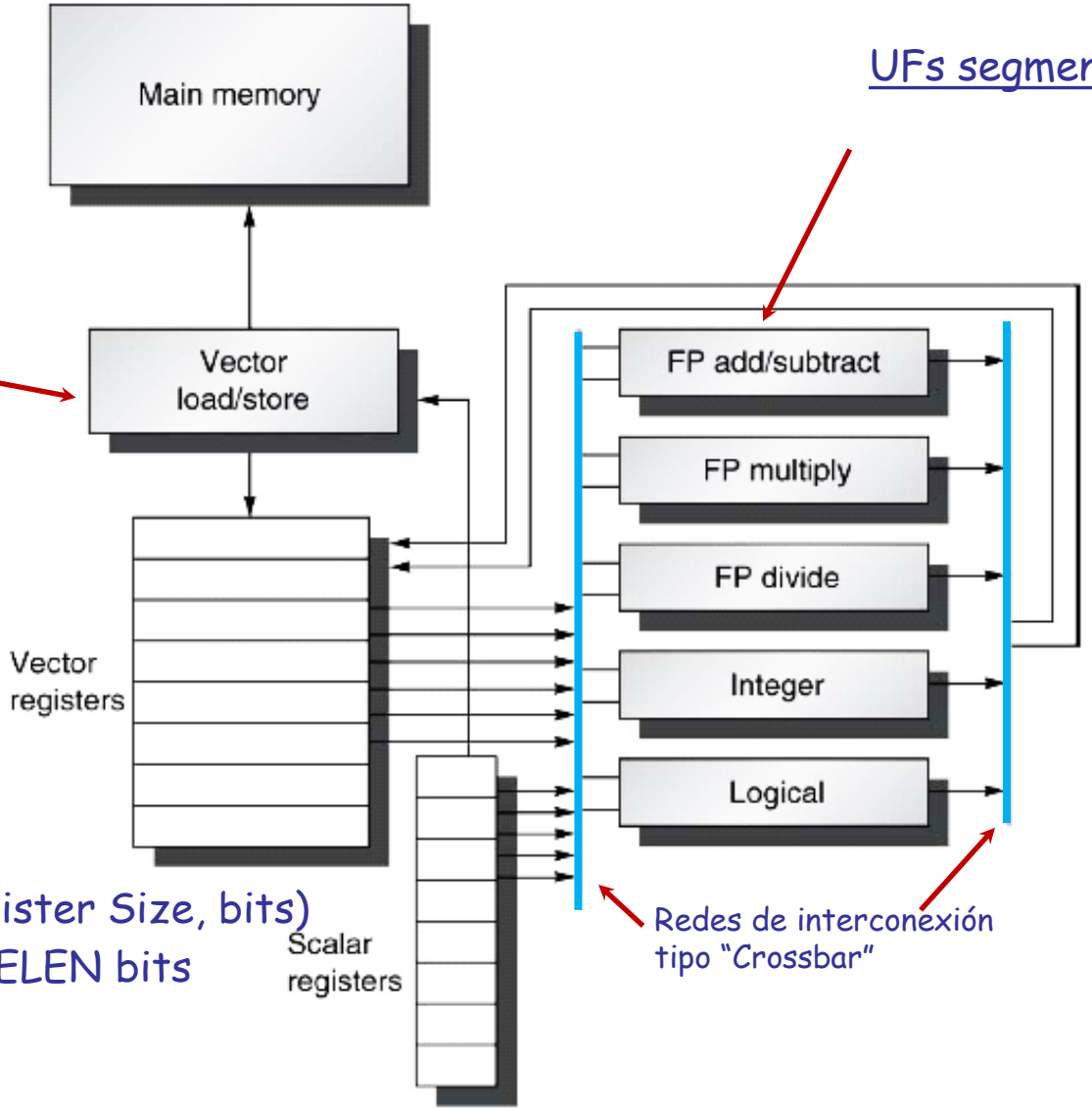
Vector ISA Goodness	RISC-V Vector Extension	Domains
• Reduced instruction bandwidth • Reduced memory bandwidth • Lower energy • Exposes DLP • Masked execution • Gather/Scatter • From small to large VPU	• Small • Natural memory ordering • Masks folded into vregs • Scalar, Vector & Matrix • Typed registers (extension)(*) • Reconfigurable • Mixed-type instructions • Common Vector/SIMD programming model • Fixed-point support • Easily Extensible • Best vector ISA ever ☺	• Machine Learning • Graphics • DSP • Crypto • Structural analysis • Climate modeling • Weather prediction • Drug design • And more…

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



❑ Estructura de un procesador vectorial



UFs segmentadas:

Funcionalmente equivalente a un pipe: un dato por ciclo, después de latencia inicial

32 registros vectoriales

tamaño VLEN (Vector Register Size, bits)

1 elemento = 8 bits hasta ELEN bits

8<=ELEN<=VLEN

Redes de interconexión tipo "Crossbar"

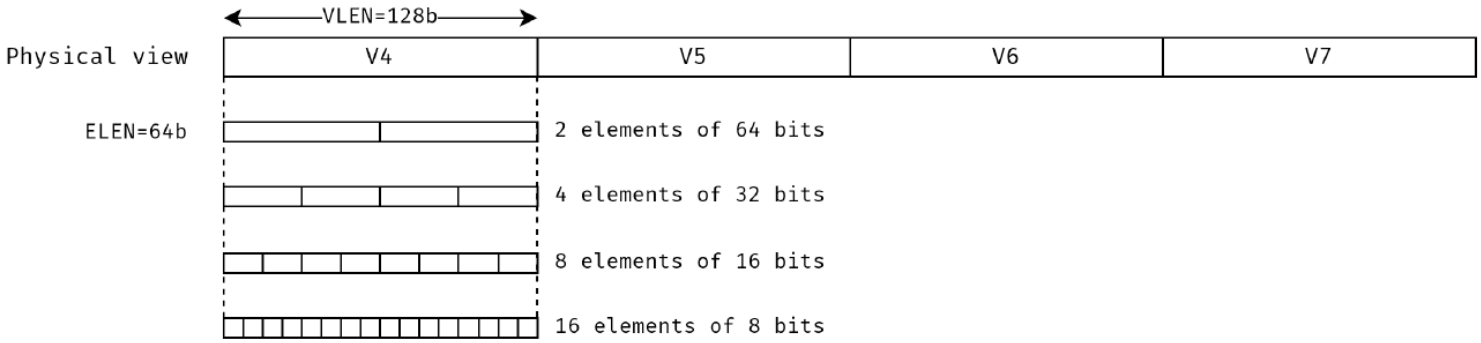
Repertorio RISC-V: terminología

Terminology

- ISA: Instruction Set Architecture
- GOPS: Giga Operations Per Second
- GFLOPS: Giga Floating-Point OPS
- **XRF**: Integer register file
- FRF: Floating-point register file
- **VRF**: Vector register file
- SIMD: Single Instruction Multiple Data
- MMX: Multi Media Extension
- SSE: Streaming SIMD Extension
- AVX: Advanced Vector Extension
- ACE: Andes Custom Extension
- CSR: Control and Status Register
- **SEW**: Element Width (8-64)
- ELEN: Largest Element Width (32 or 64)
- **XLEN**: Scalar register length in bits (64)
- FLEN: FP register length in bits (16-64)
- **VLEN**: Vector register length in bits (128-512)
- **LMUL**: Register grouping multiple (1/8-8)
- EMUL: Effective LMUL
- **VLMAX**/MVL: Vector Length Max
- AVL/**VL**: Application Vector Length
- **Configurable**: parameters are fixed at built time, i.e. cache size
- **Extensible**: added instructions to ISA includes custom instructions to be added by customer
- **Standard extension**: the reserved codes in the ISA for special purposes, i.e. FP, DSP, ...
- **Programmable**: parameters can be dynamically changed in the program

Arquitectura vectorial: repertorio RISC-V RVV (1)

- Operaciones vectoriales aritméticas sobre registros
- Instrucciones especiales de carga/almac de vectores (VLB,VSB)
- Modos de direccionamiento especiales para vectores no contiguos. Ejemplos
  - VLSB: Load Vector With Stride. Carga elementos equiespaciados a una cierta distancia
  - VLXB: Load Vector using Index. El contenido de un registro vectorial indica las posiciones de los elementos a cargar.
- Registros especiales de longitud vectorial (VLEN): V0 a V31



Extensión vectorial RISC-V 1.0
https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf

Arquitectura vectorial: repertorio RISC-V RVV (2)

- Instrucciones de Memoria

operation	instructions
vector load	vlb, vlbu, vlh, vlhu, vlw, vlwu, vld, vflh, vflw, vfld
vector load, strided	vlsb, vlsbu, vlsh, vlshu, vlsw, vlswu, vlsd, vflsh, vflsw, vflsd
vector load, indexed (gather)	vlxb, vlxbu, vlxh, vlxhu, vlxw, vlxwu, vlxd, vflxh, vflxw, vflxd
vector store	vsb, vsh, vsw, vsd
vector store, strided	vssb, vssh, vssw, vssd
vector store, indexed (scatter)	vsxb, vsxh, vsxw, vsxd
vector store, indexed, unordered	vsxub, vsxuh, vsxuw, vsxud

Arquitectura vectorial: repertorio RISC-V RVV (3)

- Instrucciones de enteros

Operandos
Vop.vv vector-vector
Vop.vx vector-escalar (registro x)
Vop.vi vector-inmediato

operation	instructions
add	vadd, vaddi, vaddw, vaddiw
subtract	vsub, vsubw
multiply	vmul, vmulh, vmulhsu, vmulhu
widening multiply	vmulwdn
divide	vdiv, vdivu, vrem, vremu
shift	vsll, vslli, vsra, vsrai, vsrl, vsrli
logical	vand, vandi, vor, vori, vxor, vxori
compare	vseq, vslt, vsltu
fixed point	vclipb, vclipbu, vcliph, vcliphu, vclipw, vclipwu

Arquitectura vectorial: repertorio RISC-V RVV (4)

- Instrucciones de Punto Flotante

Operandos
Vfop.vv vector-vector
Vfop.vf vector-escalar
(registro f)

operation	instructions
add	vfadd.h, vfadd.s, vfadd.d
subtract	vfsub.h, vfsub.s, vfsub.d
multiply	vfmul.h, vfmul.s, vfmul.d
divide	vfdiv.h, vfdiv.s, vfdiv.d
sign	vfsgn{j,jn,jx}.h, vfsgn{j,jn,jx}.s, vfsgn{j,jn,jx}.d
max	vfmax.h, vfmax.s, vfmax.d
min	vfmin.h, vfmin.s, vfmin.d
compare	vfeq.h, vfeq.s, vfeq.d, vltq.h, vflt.s, vflt.d, vfle.h, vfle.s, vfle.d
sqrt	vfsqrt.h, vfsqrt.s, vfsqrt.d
class	vfclass.h, vflcass.s, vflcass.d

Arquitectura vectorial: repertorio RISC-V RVV (5)

- PF Múltiples

operation	instructions
add	vfmadd.h, vfmadd.s, vfmadd.d
sub	vfmsub.h, vfmsub.s, vfmsub.d
widening add	vfmaddwdn.h, vfmaddwdn.s, vfmaddwdn.d
widening sub	vfmsubwdn.h, vfmsubwdn.s, vfmsubwdn.d

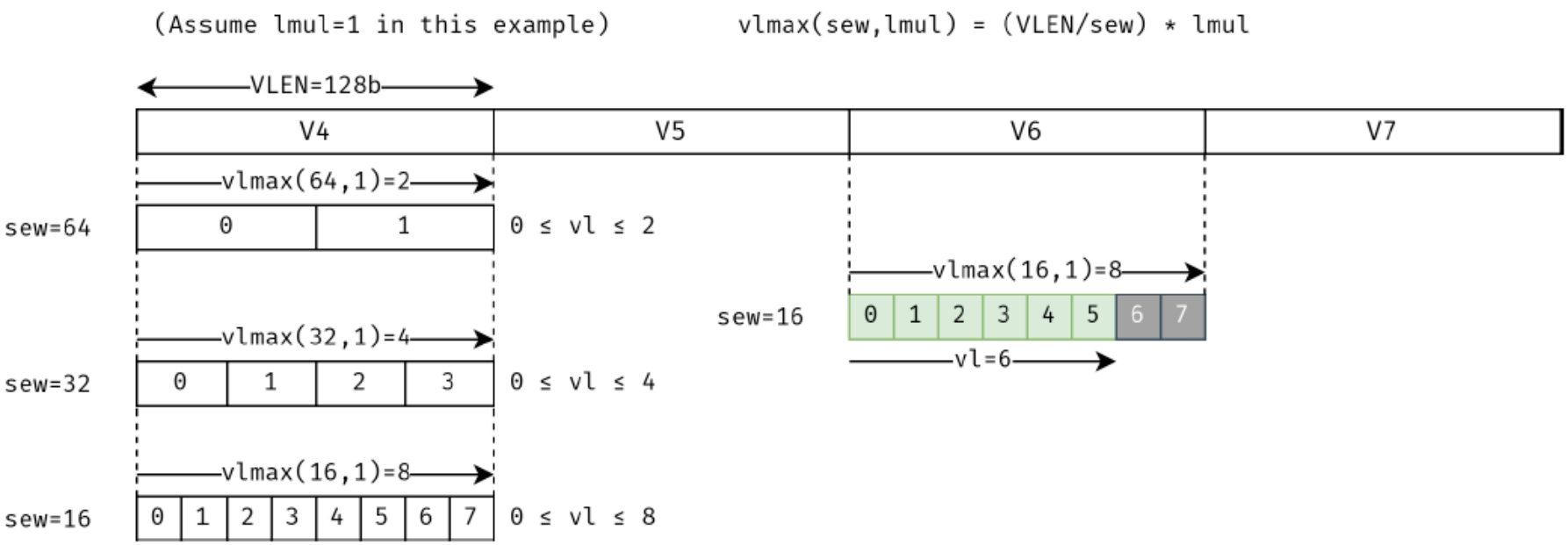
- Conversión de tipos

From Integer to Float	
To Half	vfcvt.h.i, vfcvt.h.u
To Single	vfcvt.s.i, vfcvt.s.u
To Double	vfcvt.d.i, vfcvt.d.u
From Float to Vemaxw Integer	
To Signed	vfcvt.i.h, vfcvt.i.s, vfcvt.i.d
To Unsigned	vfcvt.u.h, vfcvt.u.s, vfcvt.u.d
From Float to Float	
To Half	vfcvt.h.s, vfcvt.h.d
To Single	vfcvt.s.h, vfcvt.s.d
To Double	vfcvt.d.h, vfcvt.d.s

Configuración del tamaño: RVV Operational State

- Cuando operamos con vectores en RVV hay dos registros importantes:
  - VTYPE: tipo de Vector
  - VL: longitud del vector
    - No confundir con VLEN
- VTYPE describe el tipo de vector con el que vamos a operar e incluye:
  - SEW: Standard Element Width.
    - Tamaño en bits de los elementos que se operan
    - $8 \leq SEW \leq ELEN$
  - LMUL: Lenght Multiplier
    - Permite agrupar registros
    - lmul = 2^k where $-3 \leq k \leq 3$ (i.e., lmul $\in \{1/8, 1/4, 1/2, 1, 2, 4, 8\}$)
- VL describe cuántos elementos del vector (empezando por el elemento cero) vamos a operar
  - $0 \leq vl \leq vlmax\ (sew, lmul)$
  - vlmax (sew, lmul) = (VLEN / sew) x lmul

Ejemplo 1


(Assume lmul=1 in this example)
vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
vlmax(64,1)=2
sew=64
0
1
0 ≤ vl ≤ 2
vlmax(32,1)=4
sew=32
0
1
2
3
0 ≤ vl ≤ 4
vlmax(16,1)=8
sew=16
0
1
2
3
4
5
6
7
0 ≤ vl ≤ 8
vlmax(16,1)=8
sew=16
0
1
2
3
4
5
6
7
vl=6

Códigos válidos para elementos de distinto tamaño

- Los vectores con un tamaño de elemento más pequeño pueden contener un mayor número de elementos
  - Y al revés: un vector con elementos de tamaño ELEN bits puede contener el menor número de elementos
- Cuando operamos con vectores cuyos elementos son de diferente tamaño, tenemos diferente número de elementos
  - Esto causa problemas a los algoritmos, que quieren operar con el mismo número de elementos
- Podemos «armonizar» el número de elementos cuando el tamaño del elemento es diferente
  - No usar todo el registro vectorial para los tamaños de elemento pequeños. lmul < 1
  - Usar más de un registro vectorial para los tamaños de elemento grandes. lmul > 1

Multiplicador de Longitud

- RVV permite ampliar la longitud de los elementos a través del multiplicador de longitud (Lenght multiplier)
  - Cuando lmul = 1 podemos operar hasta todos los elementos de un registro vectorial
  - Cuando lmul < 1 podemos operar hasta una fracción de todos los elementos de un registro vectorial lmul ∈ {1/2, 1/4, 1/8}
  - Cuando lmul > 1 la operación utiliza un grupo vectorial de lmul registros vectoriales
  - Un grupo vectorial «agrupa» varios registros vectoriales.
  - El grupo vectorial se identifica por el registro vectorial con el número más pequeño del grupo.
    - 16 grupos vectoriales de lmul = 2
      - v0, v2, v4, v6, v8, v10, v12, v14, v16, ..., v28, v30
    - 8 grupos vectoriales de lmul = 4
      - v0, v4, v8, v12, v16, v20, v24, v28
    - 4 grupos vectoriales de lmul = 8
      - v0, v8, v16, v24

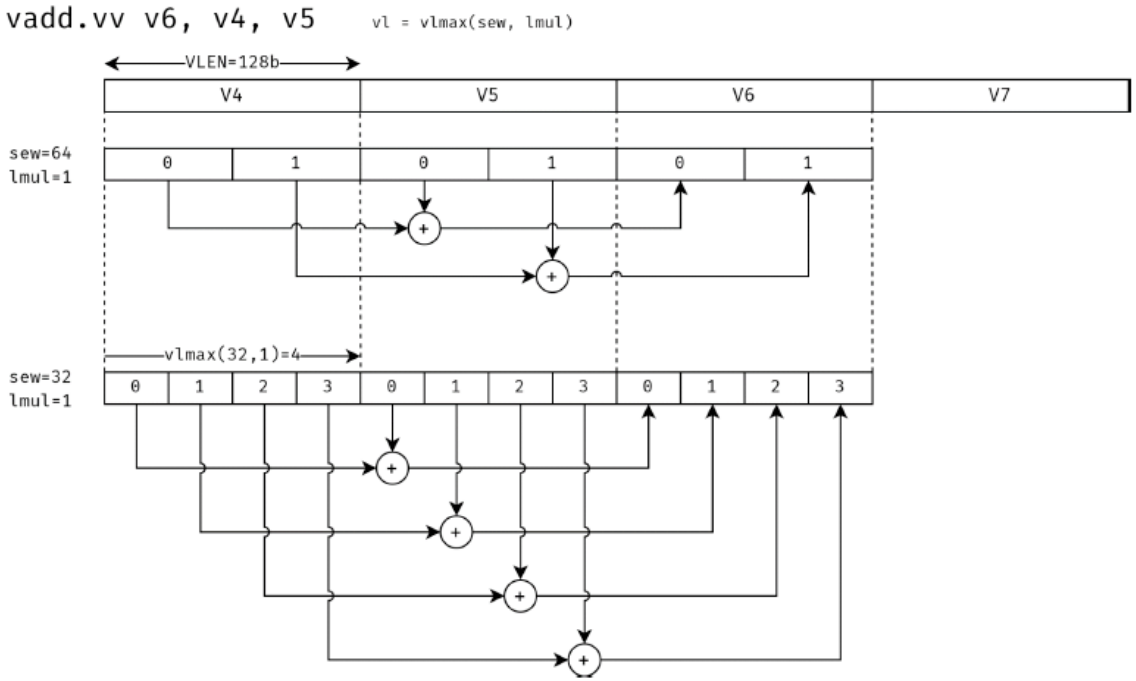
Ejemplo 2

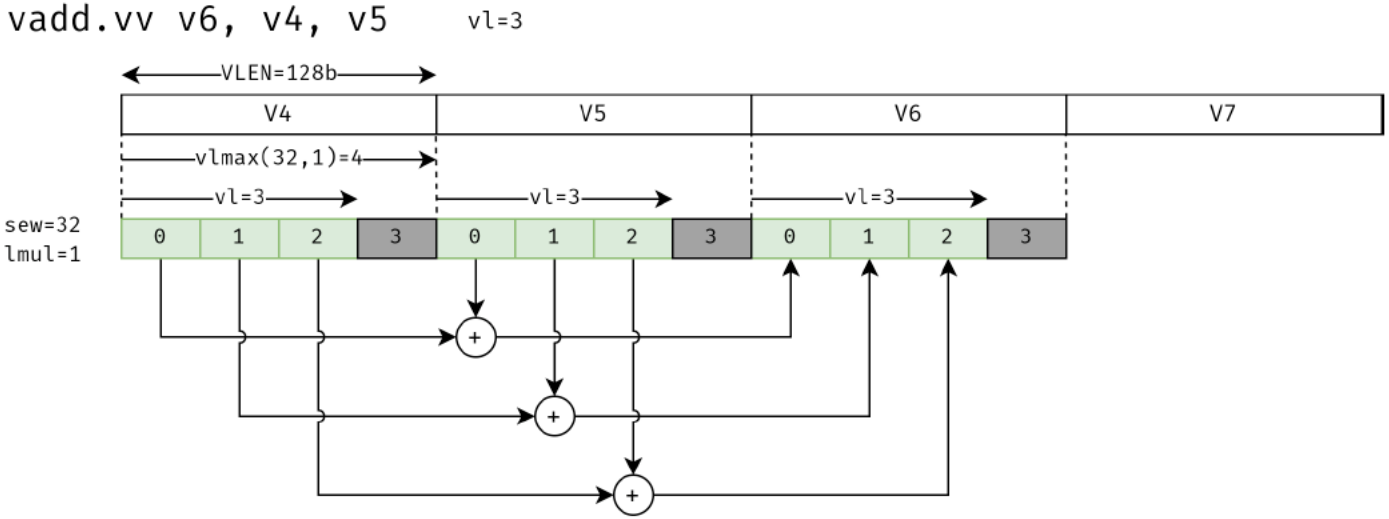

vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
vlmax(16,1)=8
sew=16
lmul=1
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 8
vl=6
vlmax(16,2)=16
sew=16
lmul=2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 ≤ vl ≤ 16
Same vlmax
vlmax(32,1)=4
sew=32
lmul=1
0 1 2 3
0 ≤ vl ≤ 4
vlmax(32,2)=8
sew=32
lmul=2
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 8
vl=6
vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
vlmax(16,1/2)=4
sew=16
lmul=1/2
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 4
vl=3
Same vlmax
vlmax(32,1)=4
sew=32
lmul=1
0 1 2 3
0 ≤ vl ≤ 4
vl=3

Operación vectorial de tamaño no máximo

- Las instrucciones vectoriales determinan completamente la operación vectorial que vamos a ejecutar utilizando los valores de vl y vtype
  - vl y vtype actúan como operandos implícitos de las instrucciones vectoriales
  - Cuando vl < vlmax entonces tenemos elementos que no son operados
    - Esos elementos se denominan elementos de cola
      - Tail elements
  - RVV ofrece dos políticas
    - tail undisturbed: cola inalterada. Los elementos de cola en el registro de destino no se modifican.
    - Tail agnostic: Puede comportarse como tail undisturbed o, alternativamente, todos los bits de los elementos tail del registro destino se ponen a 1

Ejemplo 3


vadd.vv v6, v4, v5
vl = vlmax(sew, lmul)
VLEN=128b
V4
V5
V6
V7
sew=64
lmul=1
0
1
0
1
0
1
vlmax(32,1)=4
sew=32
lmul=1
0
1
2
3
0
1
2
3
0
1
2
3


vadd.vv v6, v4, v5
vl=3
VLEN=128b
V4
V5
V6
V7
vlmax(32,1)=4
vl=3
vl=3
vl=3
sew=32
lmul=1
0
1
2
3
0
1
2
3
0
1
2
3

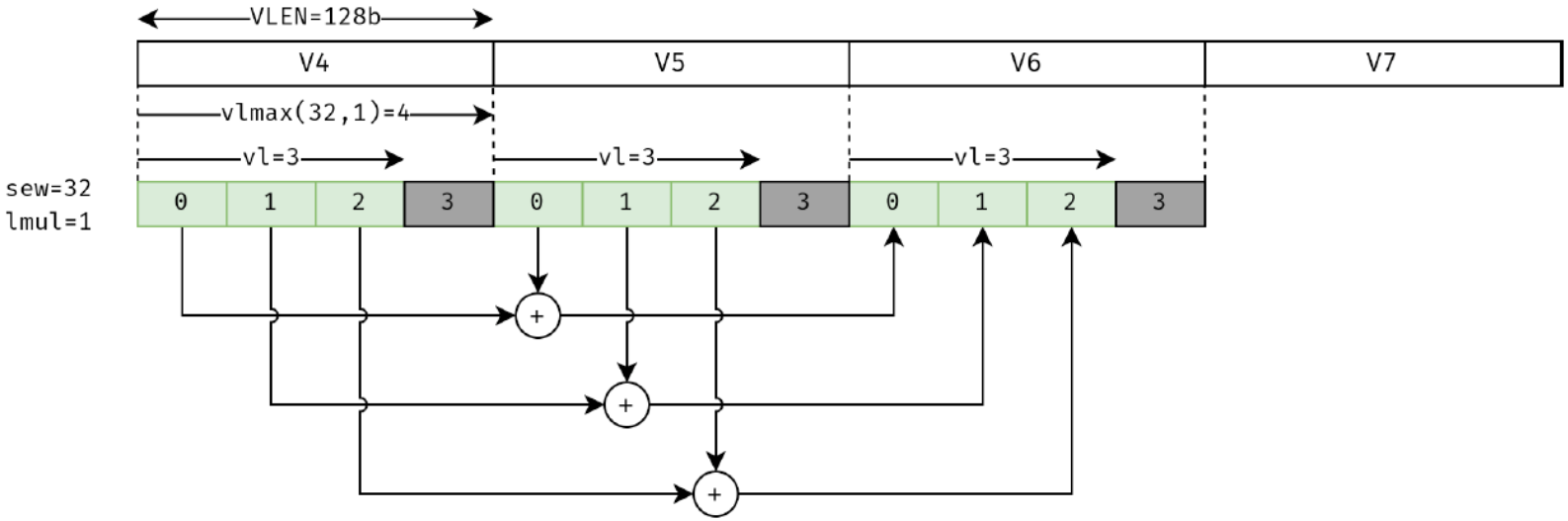
¿Cómo establecer la configuración? vset

- Las instrucciones RISCV genéricas no pueden establecer vl ni vtype
- Las instrucciones «establecer longitud de vector» permiten cambiar vl y/o vtype
  - instrucciones vsetvl/ vsetvli / vsetivli
  - instrucciones vle*ff
- Establecen vl y vtype
- La más común es vsetvli rd, rs, eN, mX, tP, mP
  - actualiza rd con la longitud de vector calculada a partir de rs, SEW (eN) y lmul (mX)
  - rs es un operando de registro de entrada que contiene la longitud de vector de aplicación (AVL) que representa la longitud de vector que el programa desea utilizar (por ejemplo, el n de for i=1 to n)
    - vsetivli sustituye este operando por un inmediato pequeño de 0 a 31.
  - N en eN es SEW (8, 16, 32, 64, etc)
  - X en mX es el lmul (expresado como fY para los casos 1/Y)
  - P es la política para tail (t) y mask (m):
    - u para undisturbed, a para agnostic

Configuración de los registros vectoriales: ejemplo

- Cada vez que vamos a ejecutar un código vectorial, debemos establecer la configuración

```
vsetivli x10, 3, e32,m1,ta,ma   # vl ← 3, sew ← 32, lmul ← 1
vadd.vv v6, v4, v5
```



- Normalmente se utiliza m1, ta y ma , en algunos ejemplos los obviaremos

Instrucciones vsetvli

- vsetvli rd , x0, eN, mX, tP, mP # rd != x0
  - Establece vl a vlmax (sew=N, lmul=X)
  - Establece vtype a sew=N, lmul a X
    - Si sólo se necesita el VLEN, existe un registro dedicado VLENB que devuelve VLEN en bytes (es decir. VLENB = VLEN/8)
- vsetvli x0, x0, eN, mX, tP, mP
  - Sólo cambia vtype
    - asume que el vector de aplicación es el vl actual
  - Sólo se usa cuando el nuevo vlmax no se modifica
  - vsetivli x0, x0, e32, m1,ta,ma
    - vlmax = (VLEN/32) * 1 = VLEN/32
      - operaciones vectoriales con sew=32, lmul = 1
  - vsetivli x0, x0, e16, mf2, ta, ma
    - vlmax = (VLEN/16) * 1/2 = VLEN/32
      - operaciones vectoriales con sew=16, lmul =1/2

Suma de vectores

A.1. Vector-vector add example

```
    # vector-vector add routine of 32-bit integers
    # void vvaddint32(size_t n, const int*x, const int*y, int*z)
    # { for (size_t i=0; i<n; i++) { z[i]=x[i]+y[i]; } }
    #
    # a0 = n, a1 = x, a2 = y, a3 = z
    # Non-vector instructions are indented
vvaddint32:
    vsetvli t0, a0, e32, ta, ma  # Set vector length based on 32-bit vectors
    vle32.v v0, (a1)             # Get first vector
      sub a0, a0, t0             # Decrement number done
      slli t0, t0, 2             # Multiply number done by 4 bytes
      add a1, a1, t0             # Bump pointer
    vle32.v v1, (a2)             # Get second vector
      add a2, a2, t0             # Bump pointer
    vadd.vv v2, v0, v1           # Sum vectors
    vse32.v v2, (a3)             # Store result
      add a3, a3, t0             # Bump pointer
      bnez a0, vvaddint32        # Loop back
      ret                        # Finished
```

vlmax (32,1)= 4
=> t0 = 4 en la primera pasada

En la última pasada t0 puede tomar un valor vl<vlmax (32,1)= 4, dependiendo del valor de n

Código vectorial a*X+Y

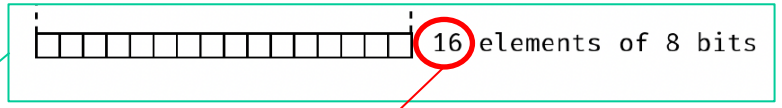
- Y = a*X + Y (*AXPY*)

```
# void
# saxpy(size_t n, const float a, const float *x, float *y)
# {
#   size_t i;
#   for (i=0; i<n; i++)
#     y[i] = a * x[i] + y[i];
# }
#
# register arguments:
#     a0      n
#     fa0     a
#     a1      x
#     a2      y

saxpy:
    vsetvli a4, a0, e32, m8, ta, ma
    vle32.v v0, (a1)
    sub a0, a0, a4
    slli a4, a4, 2
    add a1, a1, a4
    vle32.v v8, (a2)
    vfmacc.vf v8, fa0, v0
    vse32.v v8, (a2)
    add a2, a2, a4
    bnez a0, saxpy
    ret
```

Copia de memoria a memoria

```
  # void *memcpy(void* dest, const void* src, size_t n)
  # a0=dest, a1=src, a2=n
  #
memcpy:
    mv a3, a0 # Copy destination
loop:
  vsetvli t0, a2, e8, m8, ta, ma   # Vectors of 8b
  vle8.v v0, (a1)                  # Load bytes
    add a1, a1, t0                 # Bump pointer
    sub a2, a2, t0                 # Decrement count
  vse8.v v0, (a3)                  # Store bytes
    add a3, a3, t0                 # Bump pointer
    bnez a2, loop                  # Any more?
    ret                            # Return
```



t0 = 16 * 8 = 128 máximo

Operaciones condicionales: enmascaramiento (Predicación)

- El flujo de control puede ser problemático cuando se utilizan instrucciones vectoriales.
- Convertirlo en flujo de datos nos permite representar el flujo de control como un valor que puede ser retenido por un vector
- Un vector de máscara es un vector cuyos elementos son bits simples
  - No hay registros vectoriales distinguidos para vectores de máscara (se pueden usar v0 a v31)
- RVV define una disposición específica para vectores de máscara
  - los bits se empaquetan contiguamente en el registro vectorial, empezando por el bit LSB como el elemento 0 de la máscara.
- Las instrucciones pueden enmascararse utilizando el registro v0.
  - Aunque es posible operar vectores de máscara en todos los demás registros, sólo v0 puede utilizarse como operando de máscara al enmascarar una instrucción vectorial.

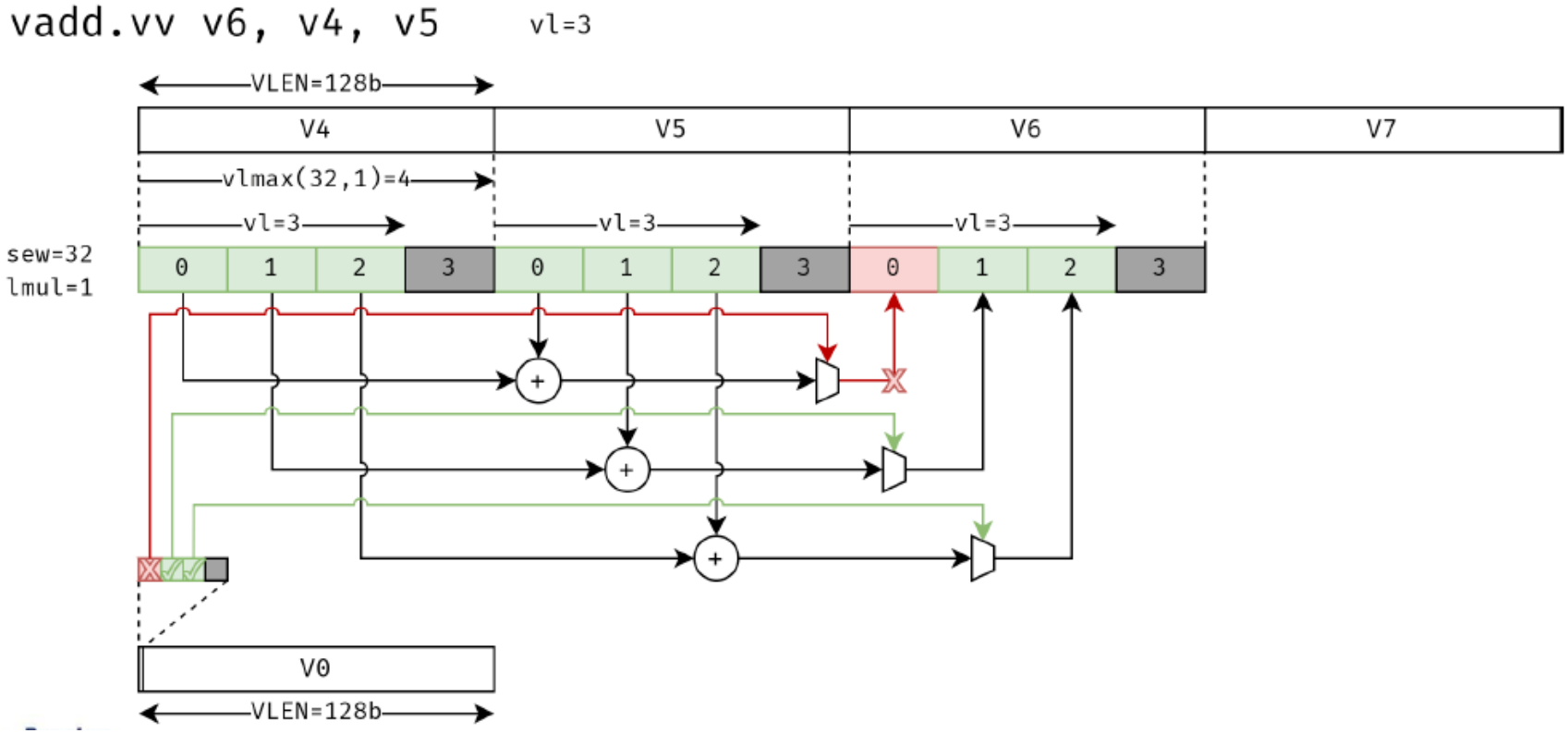
Operaciones condicionales: ejemplo

- Ejecutar

```
for (i = 0; i < 3; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] + Y[i];
```

Ejemplo: X[]= {0, 32, -5}

1. Comparar X con 0 y almacenar el resultado en V0
2. Sumar usando el registro de máscara.



Operaciones condicionales: registro de máscara

```
# C code
# set mask
for(i=0;i<8;i++)
        mask[i] = i % 2;
for(i=0; i<8; i++){
        if(mask[i])
                y[i]=a*x[i]+y[i];
         }
```

```
# Scalar Code
        li      a0, 8
loop:
        lw      a4, 0(a2)
        lw      a5, 0(a3)
        lw      t1,  0(a4)
        beqz t1,  skip #if mask[i]=0
        mul  a4, a4, a1
        add  a4, a4 ,a5
        sw    a4, 0(a3)
skip:
        addi a0,  a0, -1
        addi a3,  a3,  4
        addi a2,  a2,  4
        bnez a0, loop
```

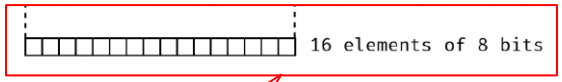
```
# Vector Code
vsetvli  t0, zero, e32, m2, ta, ma   # t0 = 8
vl2re32.v  v8,   (a2)
vl2re32.v  v10, (a3)
vl2re32.v  v12,   (a4)
vmsne.vx  v0, v12, zero
# Set the v0, enabling the mask if mask[i] is not zero
vmacc.vx  v10, a1,  v8, v0.t
vs2r.v        v10,  (a3)

# (a2)    x
# (a3)    y
# (a4)    mask
# a1       a
```

Operación vectorial enmascarada

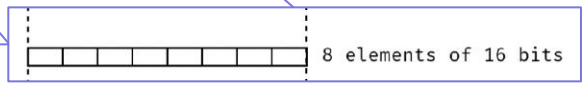
- Cuando se ejecuta una operación vectorial, el registro v0 (interpretado como una disposición vectorial enmascarada) determina si un elemento sin cola está activo o inactivo
  - Los elementos activos funcionan como de costumbre
  - Los elementos inactivos no se operan
- Los elementos inactivos también tienen una política sin máscara.
  - Mask undisturbed: El elemento correspondiente del registro de destino no se modifica. (mu)
  - Mask Agnostic: El elemento correspondiente del registro de destino pone todos sus bits a 1. (ma)

Ejemplo de procesamiento selectivo




```
# (int16) z[i] = ((int8) x[i] < 5) ? (int16) a[i] : (int16) b[i];
#

loop:
    vsetvli t0, a0, e8, m1, ta, ma # Use 8b elements.
    vle8.v v0, (a1)          # Get x[i]
      sub a0, a0, t0         # Decrement element count
      add a1, a1, t0         # x[i] Bump pointer
    vmslt.vi v0, v0, 5       # Set mask in v0
    vsetvli t0, a0, e16, m2, ta, mu  # Use 16b elements.
      slli t0, t0, 1         # Multiply by 2 bytes
    vle16.v v2, (a2), v0.t   # z[i] = a[i] case
    vmnot.m v0, v0           # Invert v0
      add a2, a2, t0         # a[i] bump pointer
    vle16.v v2, (a3), v0.t   # z[i] = b[i] case
      add a3, a3, t0         # b[i] bump pointer
    vse16.v v2, (a4)         # Store z
      add a4, a4, t0         # z[i] bump pointer
      bnez a0, loop
```



Uso de máscara

A.2. Example with mixed-width mask and compute.

```
# Code using one width for predicate and different width for masked
# compute.
#   int8_t a[]; int32_t b[], c[];
#   for (i=0;  i<n; i++) { b[i] =  (a[i] < 5) ? c[i] : 1; }
#
# Mixed-width code that keeps SEW/LMUL=8
  loop:
    vsetvli a4, a0, e8, m1, ta, ma   # Byte vector for predicate calc
    vle8.v v1, (a1)                  # Load a[i]
      add a1, a1, a4                 # Bump pointer.
    vmslt.vi v0, v1, 5               # a[i] < 5?

    vsetvli x0, a0, e32, m4, ta, mu  # Vector of 32-bit values.
      sub a0, a0, a4                 # Decrement count
    vmv.v.i v4, 1                    # Splat immediate to destination
    vle32.v v4, (a3), v0.t           # Load requested elements of C, others undisturbed
      sll t1, a4, 2
      add a3, a3, t1                 # Bump pointer.
    vse32.v v4, (a2)                 # Store b[i].
      add a2, a2, t1                 # Bump pointer.
      bnez a0, loop                  # Any more?
```

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 consumen (y producen) un elemento por ciclo
  - 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.) m x n ciclos de reloj (Notación: $T_{chime}=m$)

Tiempo de ejecución

- Convoyes: ejemplo

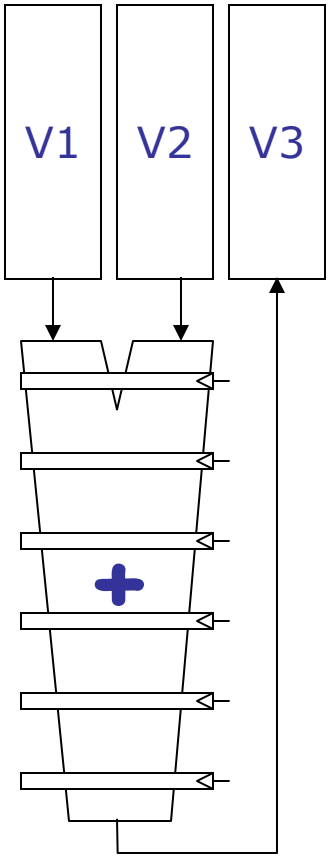
```
VSETVLI T0,X0,E32,M1,TA,MA

1: VLE32.V     V1, (Rx)      ; load vector X
2: VFMULV.VF   V2, V1, F0    ; vector-scalar multiply
3: VLE32.V     V3, (Ry)      ; load vector Y
4: VFADD.VV    V4, V2, V3    ; Vector addition
5: VSE32.V     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 load y multiplicación
    - 2. Formado por load y suma
    - 3. Formado por store
  - 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



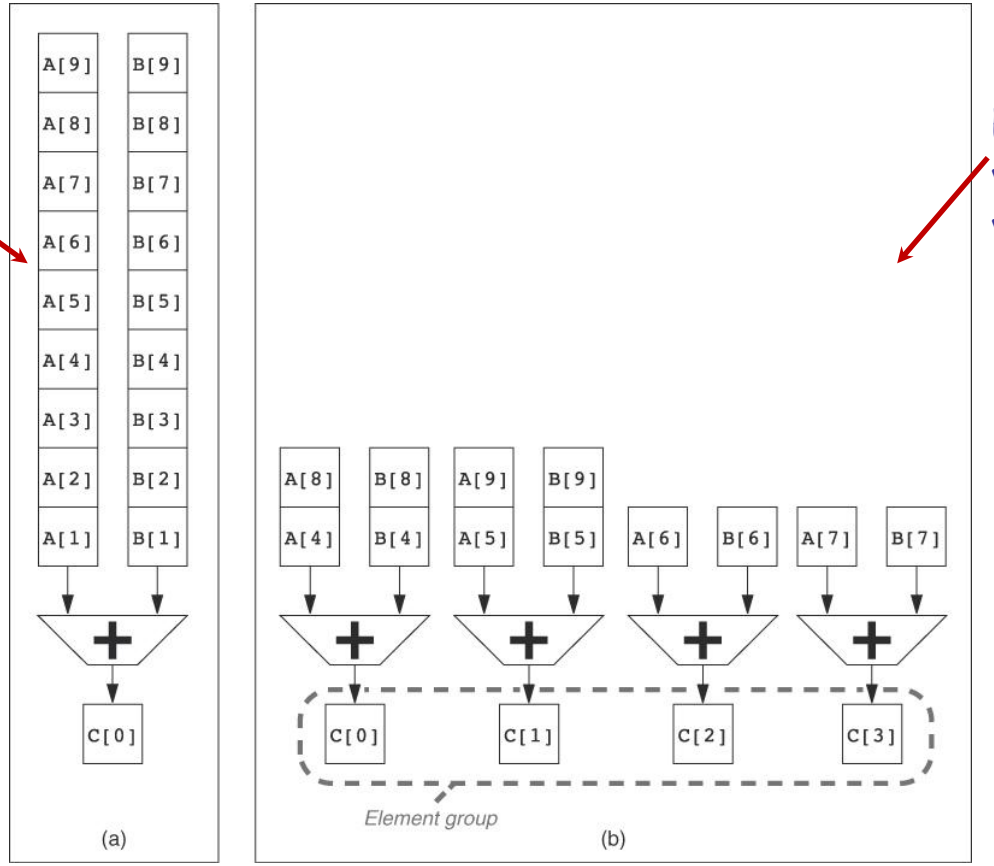
UF de suma segmentada en 6 etapas

V3 = V1 + V2

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

Ejecución de operaciones aritméticas

- Operaciones independientes
  - Ejemplo

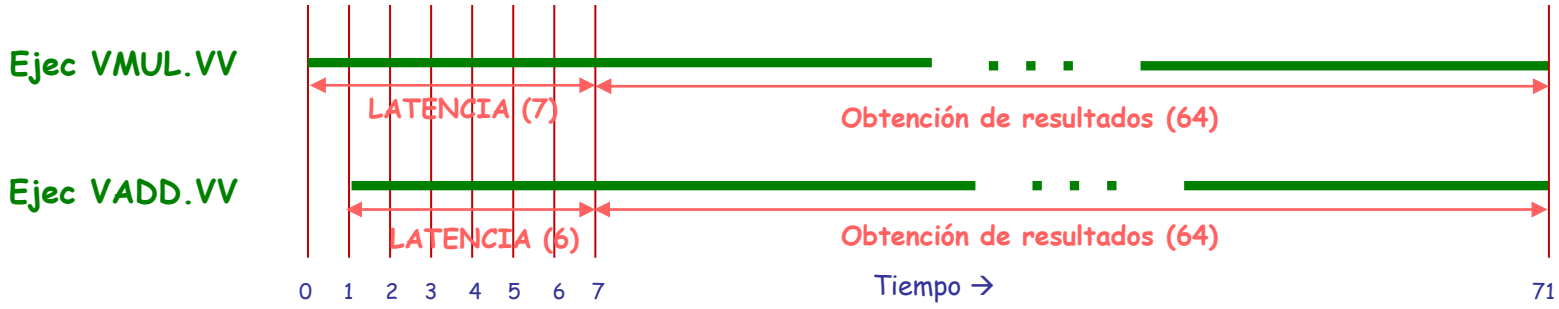
    VMUL.VV V1, V2, V3

    VADD.VV V4, V5, V6

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

Operación	Inicio	Fin
VMUL.VV	0	7+64 = 71
VADD.VV	1	1+6+64 = 71

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

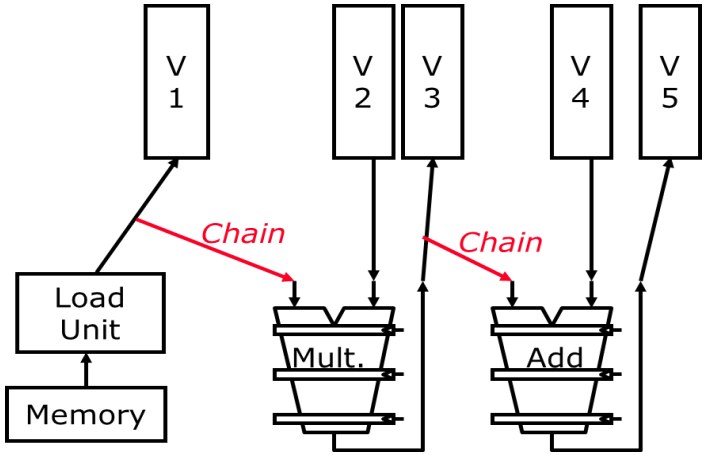


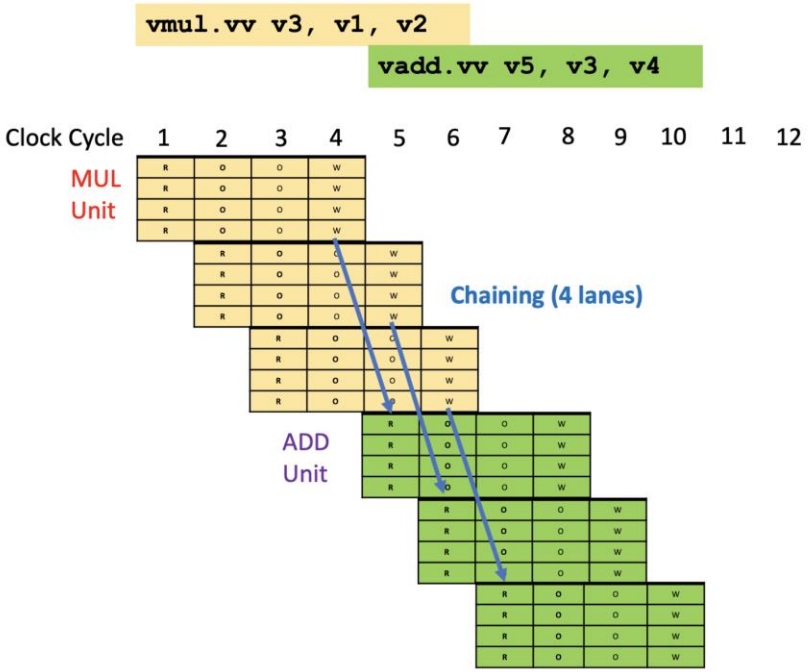
Encadenamiento

- Operaciones dependientes: Tratamiento de dependencias LDE (RAW)
- Problema

```
VSETVLI T0,X0,E8,M1,TA,MA
VLE32.V     V1, (a0)
VMUL.VV     V3, V1, V2
VADD.VV     V5, V3, V4
```

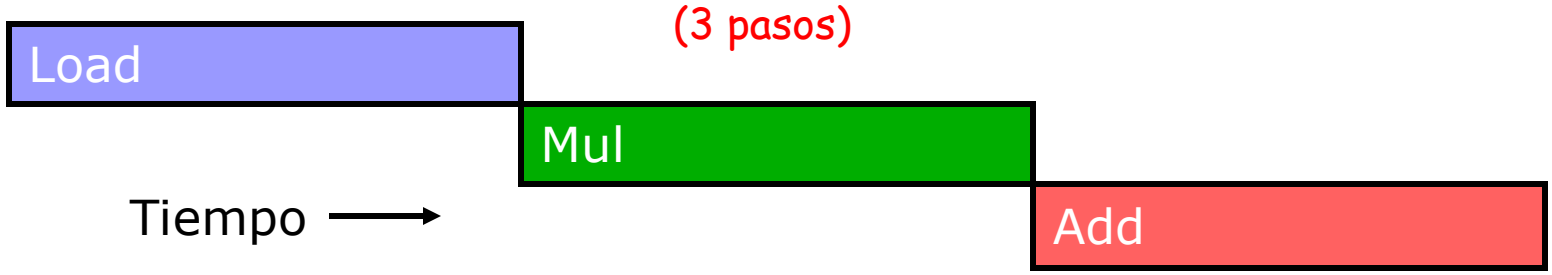
- Solución:
  - Cray-1. Extensión del concepto de anticipación de operandos ⇒ Encadenamiento de operaciones (chaining)
  - 3 operaciones, pero 1 paso
    - Tchime = 1
  - En proc modernos: "encadenamiento flexible"



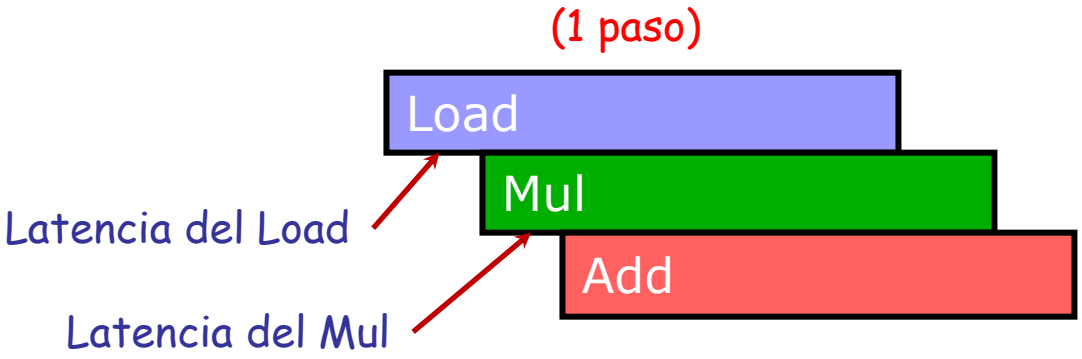


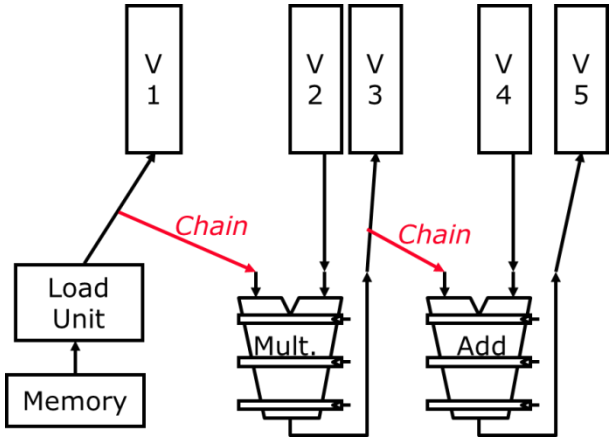
Encadenamiento

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



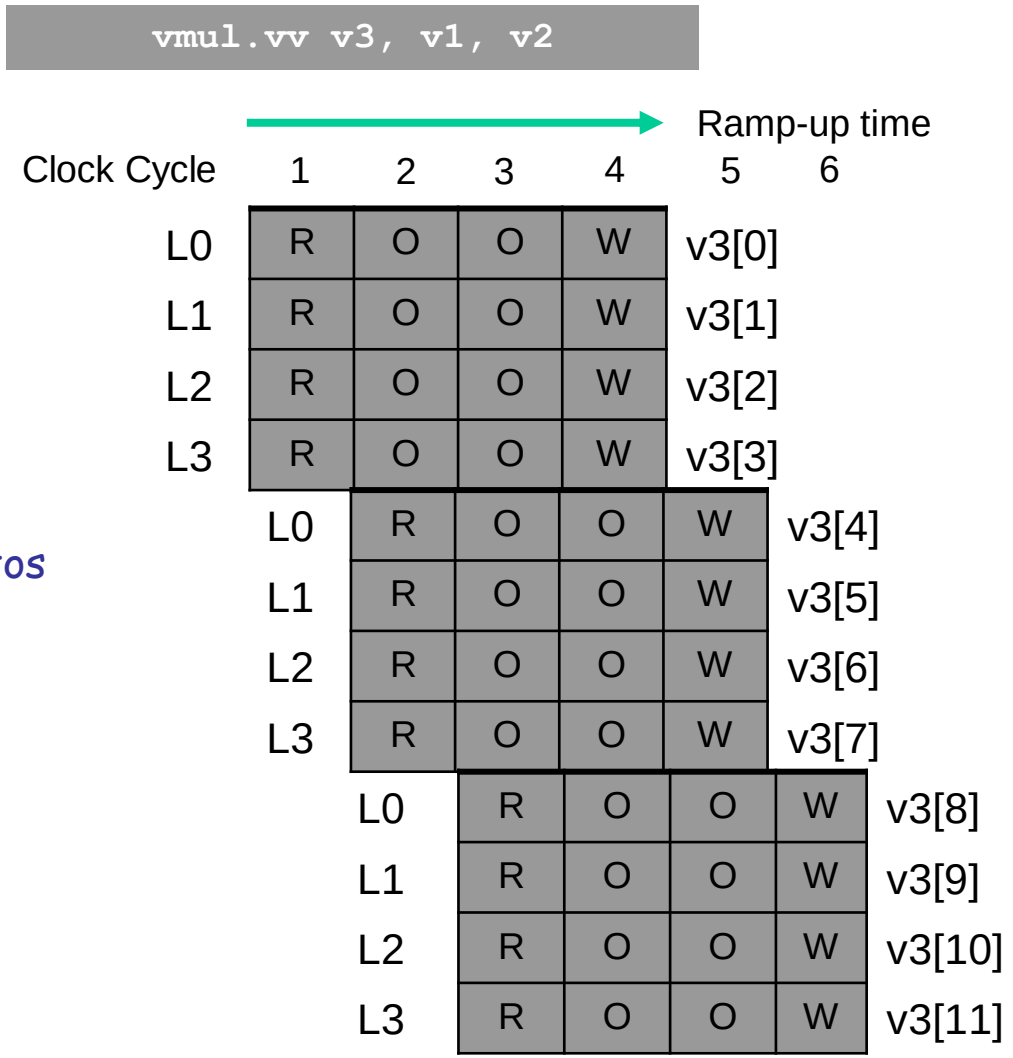
- Con encadenamiento: Una instrucción puede comenzar cuando está disponible el primer elemento de la operación de la que depende





Tiempo con varias vías y encadenamiento

- 4 Lineas de ejecución
- Fus 4 etapas
- Tamaño de vector de 12
- No existen dependencias entre datos
- Resultado disponible en 6 ciclos
- 4 ciclos de llenado del pipeline

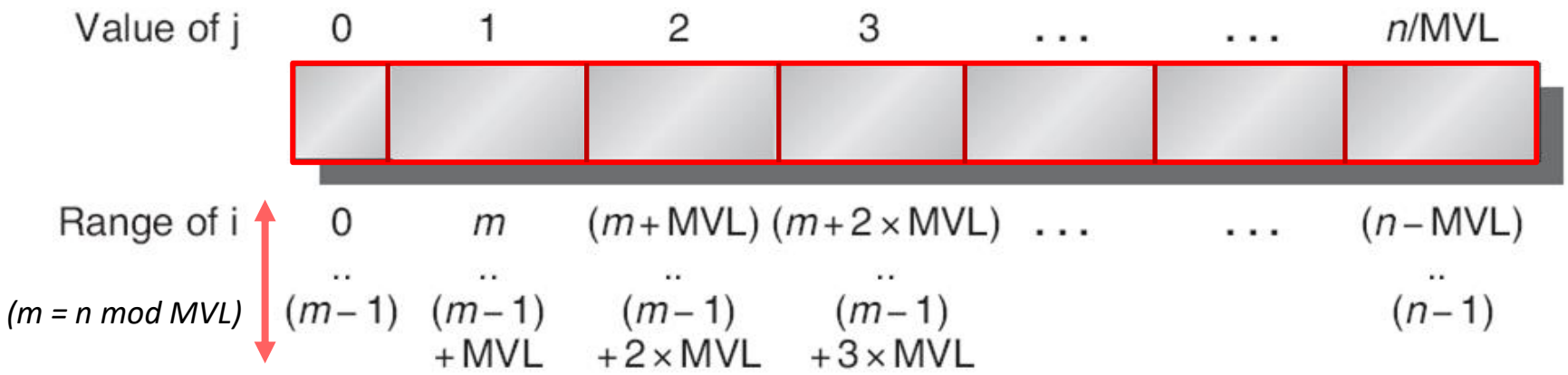


Vectores de longitud arbitraria (strip mining)

- 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, operaciones 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 instrucciones son independientes, el llenado de pipes se solapa.
    - T_{loop}: actualización de punteros, detección de fin. Simplificación: 15 ciclos
  - Nº de convoyes en el bucle (T_{chime})
  - Tiempo total de cálculo para vectores de n elementos (1 vía)
    - $T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$

Nº de iteraciones

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

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
  - 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

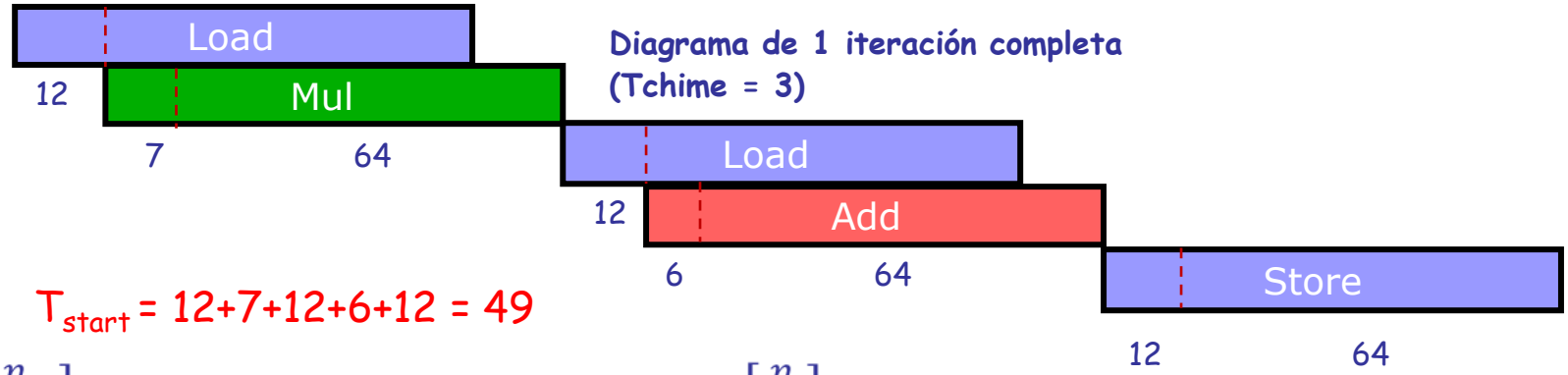


Diagrama de 1 iteración completa (Tchime = 3)

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

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

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: Suponemos que se obtiene con n<MVL

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

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

$$= 64 + 3n$$

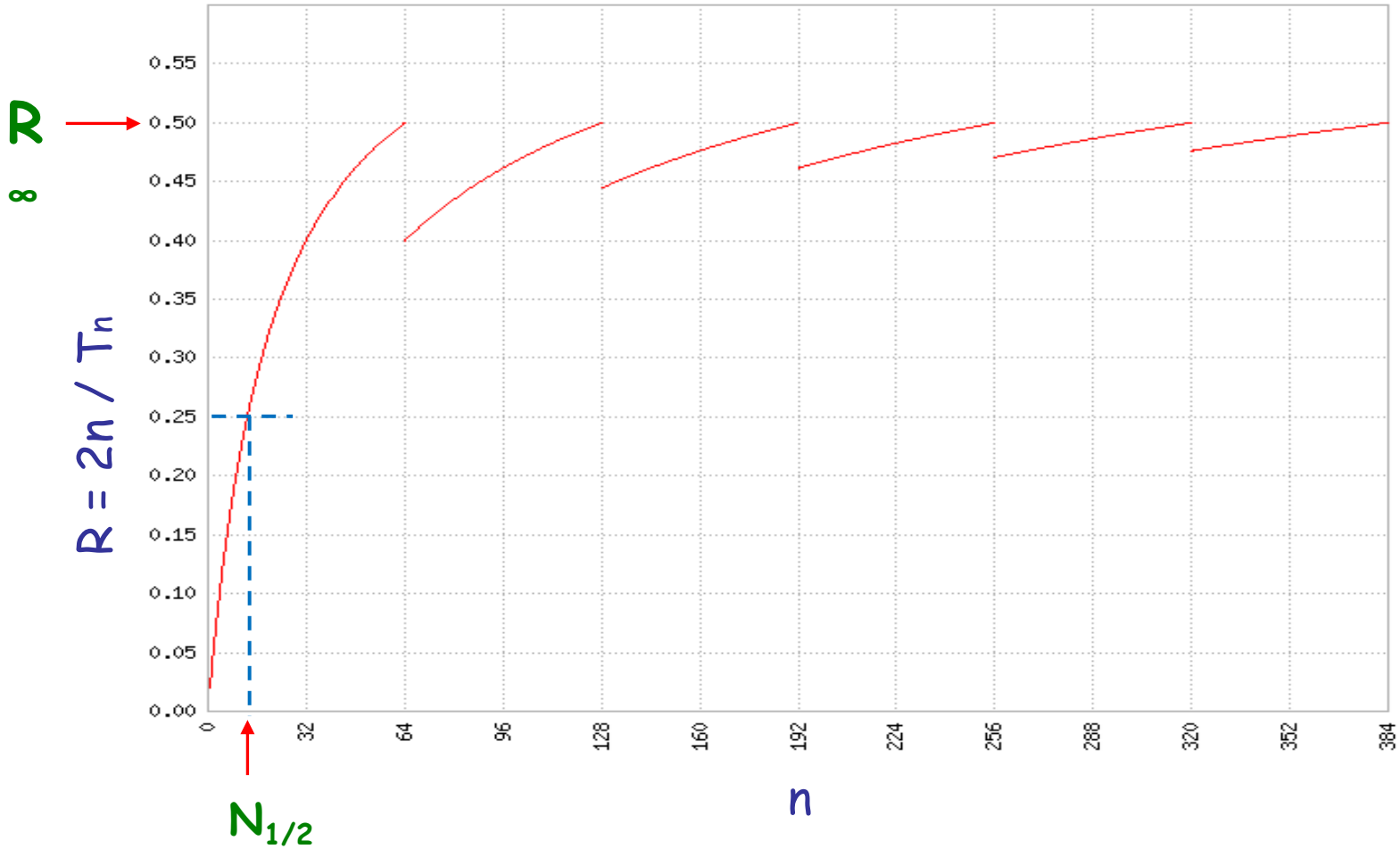
$$En\ el\ ejemplo\ \ \frac{R_\infty}{2} = \frac{1}{4}\frac{FLOP}{Ciclo}$$

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

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

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

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



Ejemplo de Supercomputador Vectorial: NEC SX-Aurora

Vector Engine Processor Overview

SX-Aurora TSUBASA

Components

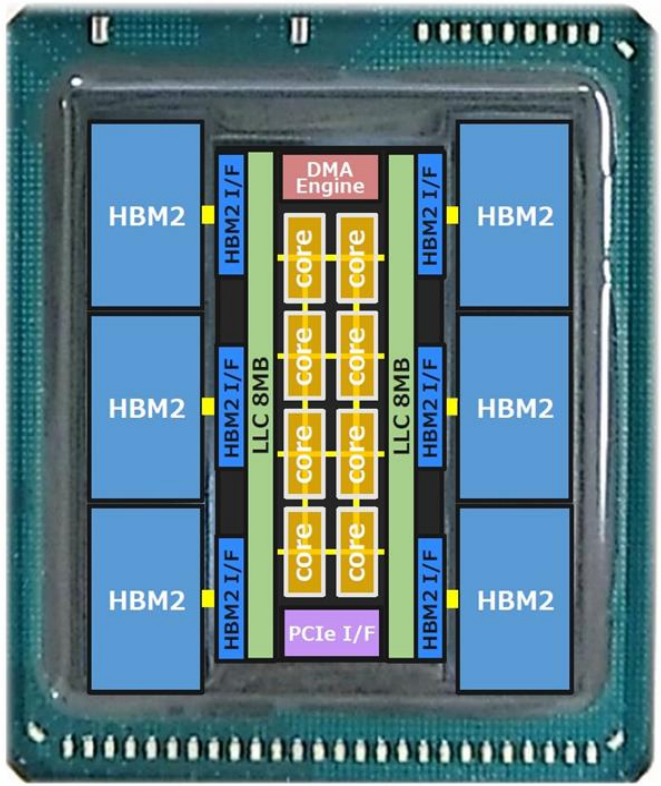
- 8 vector cores
- 16MB LLC
- 2D mesh network on chip
- DMA engine
- 6 HBM2 controllers and interfaces
- PCI Express Gen3 x16 interface

Specs

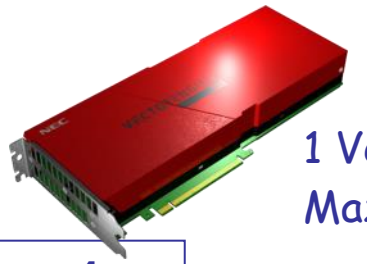
Core frequency	1.6GHz
Core performance	307GF(DP) 614GF(SP)
CPU performance	2.45TF(DP) 4.91TF(SP)
Memory bandwidth	1.2TB/s
Memory capacity	24/48GB

Technology

- 16nm FinFET process



© NEC Corporation 2018
\Orchestrating a brighter world NEC



1 Vector Engine
Max 2.45 TF (DP)

64 Vector Engines
Max 157 TF (DP)

Ejemplo de Supercomputador Vectorial: NEC SX-Aurora

- Vector core:
  - Cada registro tiene 256 componentes
  - Cada FMA tiene 32 vías

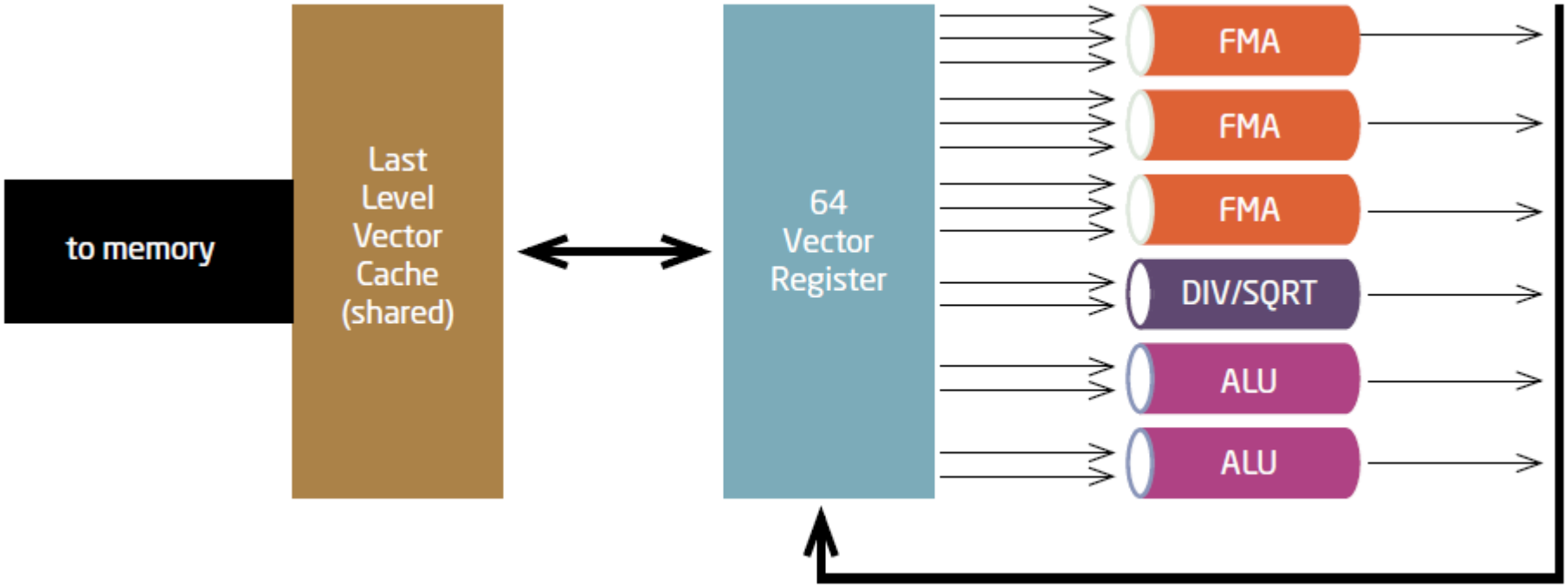


Imagen tomada de https://www.nec.com/en/global/solutions/hpc/sx/architecture.html?

Paralelismo a nivel de bucle: vectorización

- Dependencia directa: dependencia RAW entre sentencias de una misma iteración
  - No impide la vectorización (mediante encadenamiento de pipes)
  - Obliga a ejecutar en el orden dado
  - Ejemplo

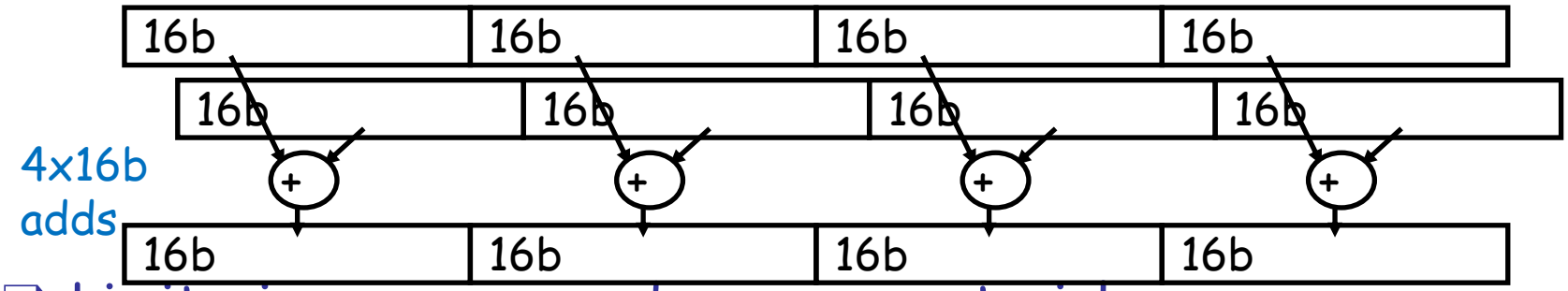
```
for (i=999; i>=0; i=i-1){
        x[i] = x[i] + s;        /* S1 */    Orden: 1º S1 y luego S2
        z[i] = z[i] + x[i];     /* S2 */
}
```

- Dependencia en el espacio de iteraciones (loop-carried): los datos de una iteración son dependientes de los resultados generados en iteraciones previas
  - Puede impedir la vectorización en algunos casos o pueden existir reordenaciones de sentencias válidas en otros.
  - Ejemplo

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

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 vectorial, aunque con un vector de pocos elementos

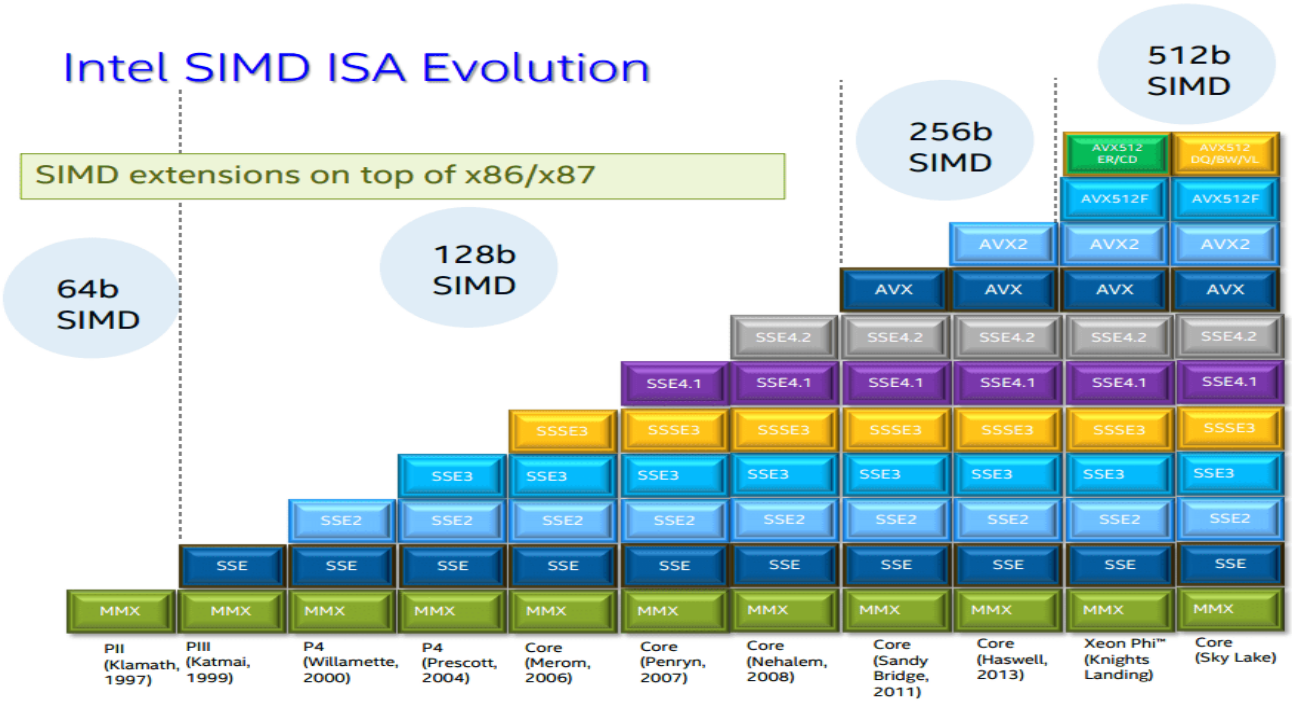


- Limitaciones, comparado con op vectoriales:
  - La longitud de los vectores se codifica en el Cod_op
  - No existe direccionamiento sofisticado (stride, gather,...)
  - No hay reg de máscara

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



Extensiones SIMD

- Ejemplo: Instrucciones AVX para la arquitectura x86

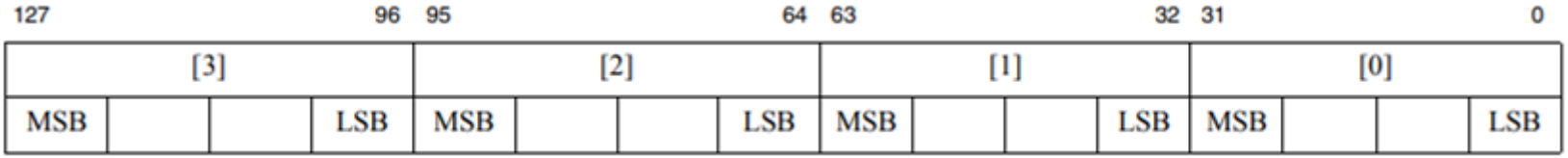
4 operandos de 64 bits

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.

Extensiones multimedia: MIPS SIMD (1)

- MIPS64 SIMD Architecture module (2016)
  - Más de 150 instrucciones
  - 32 registros vectoriales de 128 bits (w0 a w31)
  - Operandos en punto flotante de 32 y 64 bits y enteros de 8, 16, 32 y 64 bits
  - Ejemplo: almacenamiento de 4 elementos de tamaño palabra (32 bits) en un registro vectorial MSA



- Sintaxis de las instrucciones: func.df operandos
  - Funct: operación (ADDV, ADDVI, MULV,…)
  - Df: Formato de datos del destino de la operación (B, H, W, D)
  - wd, ws, wt: registros vectoriales (ej: $w3)
  - rd, rs: registros escalares de propósito general (ej: $8)
  - ws[n]: componente n-ésima del registro vectorial ws
  - m: valor inmediato

Material tomado de https://www.mips.com/products/architectures/ase/simd/

Extensiones multimedia: MIPS SIMD (2)

Instrucción	Operandos	Función
ADDV.df	wd,ws,wt	Add elements of ws and wt, then put each result in wd.
ADDVI.df	wd,ws,m	Add constant m to each element of ws, then put each result in wd.
SUBV.df	wd,ws,wt	Subtract elements of ws from wt, then put each result in wd.
SUBVI.df	wd,ws,m	Subtract constant m from elements of ws, then put each result in wd.
MULV.df	wd,ws,wt	Multiply elements of ws and wt, then put the least significant half of each result in wd.
MADDV.df	wd,ws,wt	Multiply elements of ws and wt, then add to wd (result as MULV)
DIV_S.df	wd,ws,wt	Signed divide elements of ws by wt, then put each result in wd.
DIV_U.df	wd,ws,wt	Unsigned divide elements of ws by wt, then put each result in wd.
FADD.W FADD.D	wd,ws,wt	IEEE FP (single or double) add elements of ws and wt, then put each result in wd. (FSUB, FMUL, FMADD, FDIV also available)
LD.df	wd,m(rs)	Load wd from address at rs + m elements.
ST.df	wd,m(rs)	Store wd in memory from address at rs + m elements.
FILL.df	wd,rs	The value in register rs is replicated to all elements in wd.
LDI.df	wd,m	Constant m is replicated to all elements in wd.
SPLAT.df	wd,ws[rt]	Replicate element of ws with index given by rt to all elements in wd.
MOVE.V	wd,ws	Copy all bits in ws to wd.
AND.V	wd,ws,wt	Logical AND of each bit of ws and wt stored in wd. (OR, NOR, XOR)

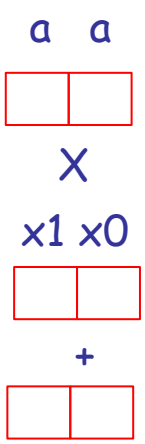
Extensiones multimedia: MIPS SIMD (3)

Instrucción	Operandos	Función
BZ.V	wt,m	Branch if all elements in wt are zero (displacement m is added to PC).
BZ.df	wt,m	Branch if at least 1 element of wt is zero
C--.df	wd,ws,wt	Compare the elements (EQ, GT_S, GT_U, LT_S, LT_U, GE_S, GE_U, LE_S, LE_U) in ws and wt. If condition is true, put all 1s in the corresponding element of wd: otherwise put 0s. The instruction C--I_-.df performs the same compare but using a scalar value as one operand.
C--I_-.df	wd,ws,m	
BSEL.V	wd,ws,wt	Selectively copy bits from ws and wt into wd based on the corresponding bit in wd: if 0 copies the bit from ws, if 1 copies the bit from wt
BMZ.V	wd,ws,wt	Copy to wd all bits from ws for which the corresponding bits from wt are 0 and leave unchanged all remaining destination bits. BMNZ copies bits that are 1 in wt.
BMNZ.V		

Ejemplo: código con extensiones SIMD en MIPS

- Código para DAXPY: Vectores 64 componentes

```
                                ;suppose w1 preloaded with {a,a}
        DADDIU R4,Rx,#512       ;last address to load
Loop:   LD.D w2,0(Rx)           ;load X[i], X[i+1]
        LD.D w3,0(Ry)           ;load Y[i], Y[i+1]
        FMADD.D w3,w2,wa        ;a×X[i]+Y[i], a×X[i+1]+Y[i+1]
        ST.D w3,0(Ry)           ;store into Y[i], Y[i+1]
        DADDIU Rx,Rx,#16        ;increment index to X
        DADDIU Ry,Ry,#16        ;increment index to Y
        DSUBU R20,R4,Rx         ;compute bound
        BNEZ R20,Loop           ;check if done
```



Para FPD sólo se reducen a la mitad las iteraciones.

Para números enteros de tamaño pequeño (8, 16 bits) la mejora es mayor.

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"

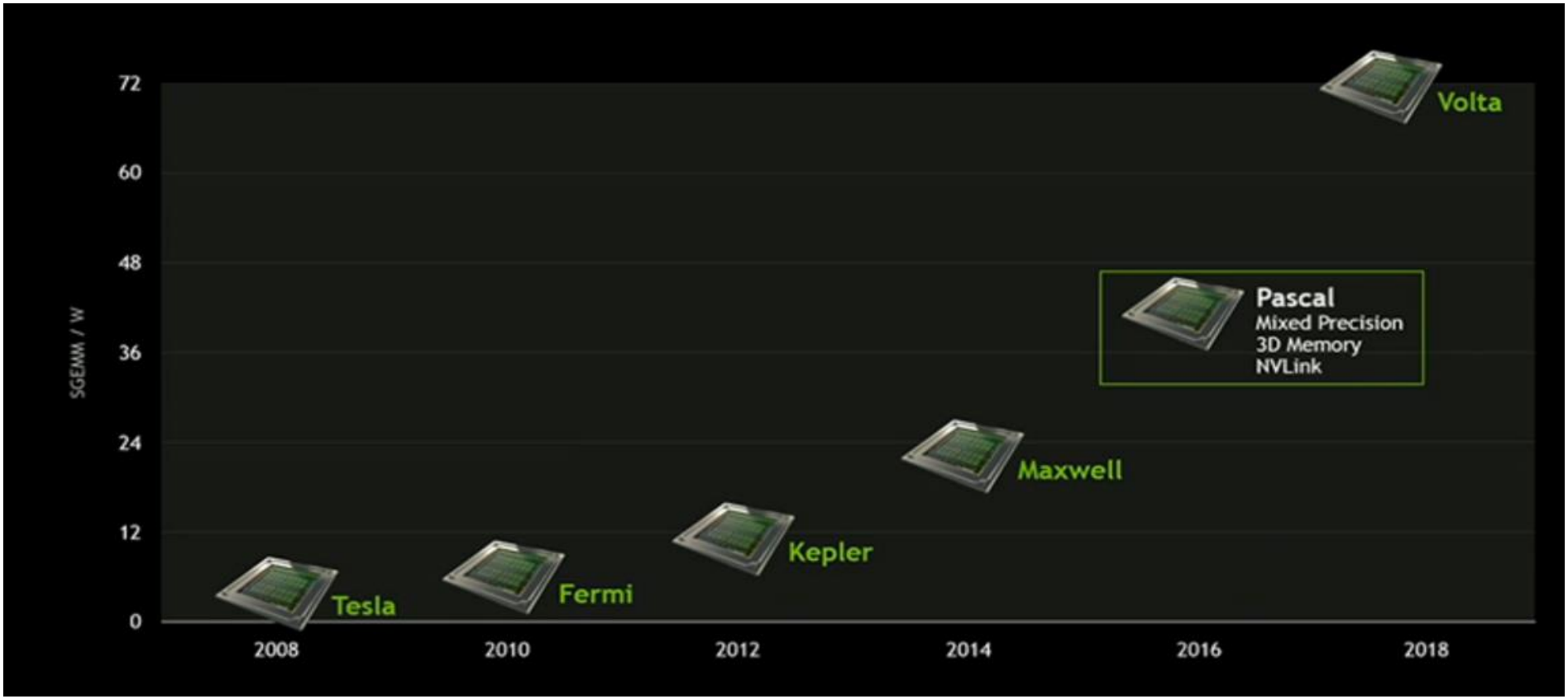
Lectura: sección 4.4 de H&P 5th ed.

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
- Actualmente diseñan GPUs tanto AMD (Instinct MI200) como Intel (Ponte Vechio). Se programan usando OpenCL.

Arquitectura de NVIDIA GPU

- GPU NVIDIA
  - 2018 Volta (7,5Tflops DP, 15Tflops SP)
  - 2020 Ampere (9,8Tflops DP, 54 Btrans, 400w, 7nn)
  - 2022 Hopper (30 Tflops DP, 80 Btrans, 700w, 4N, 814mm2). Datos estimados



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 (grid) 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

CPU

```
// 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);
```

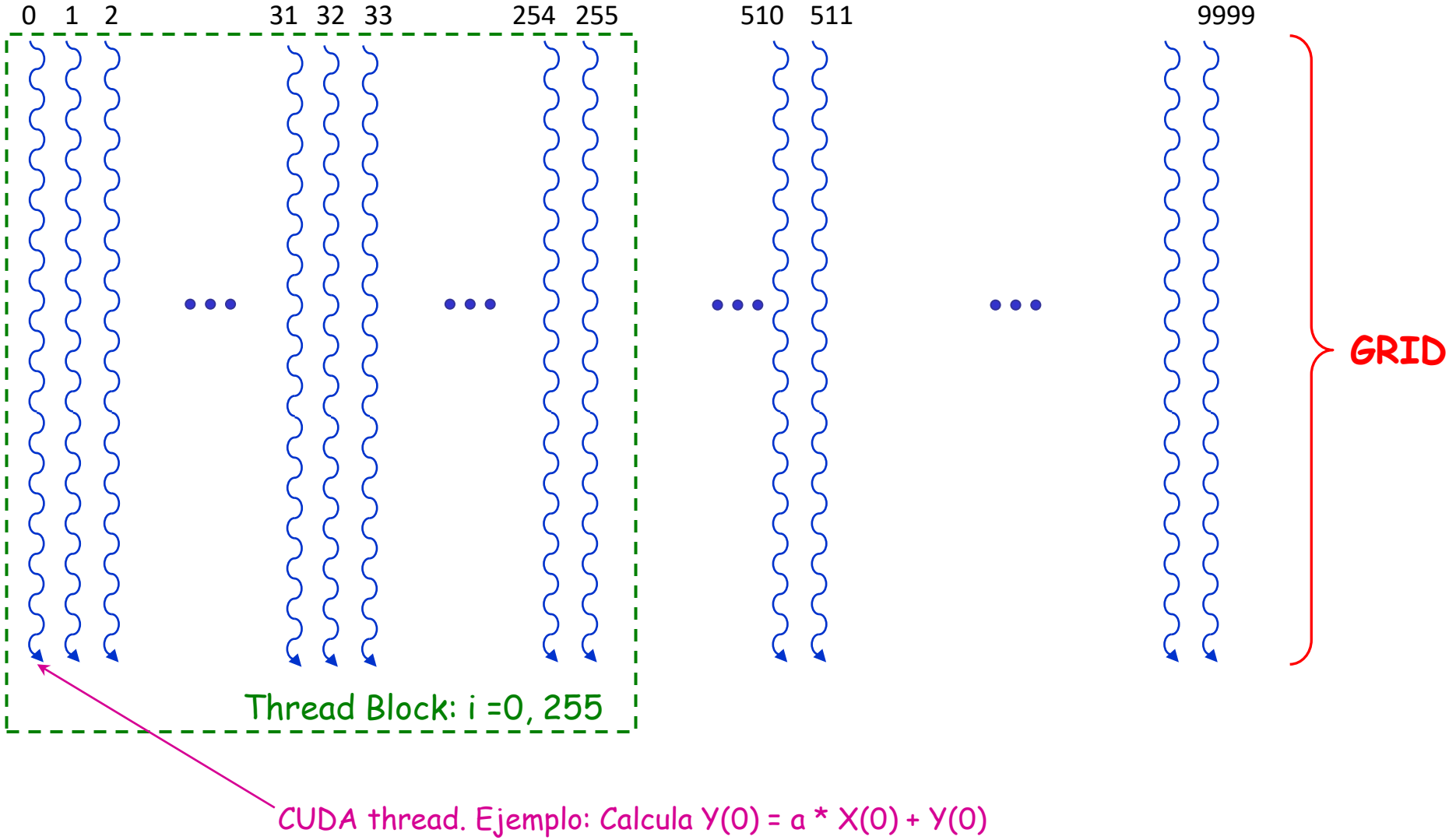
GPU

```
// 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];
}
```

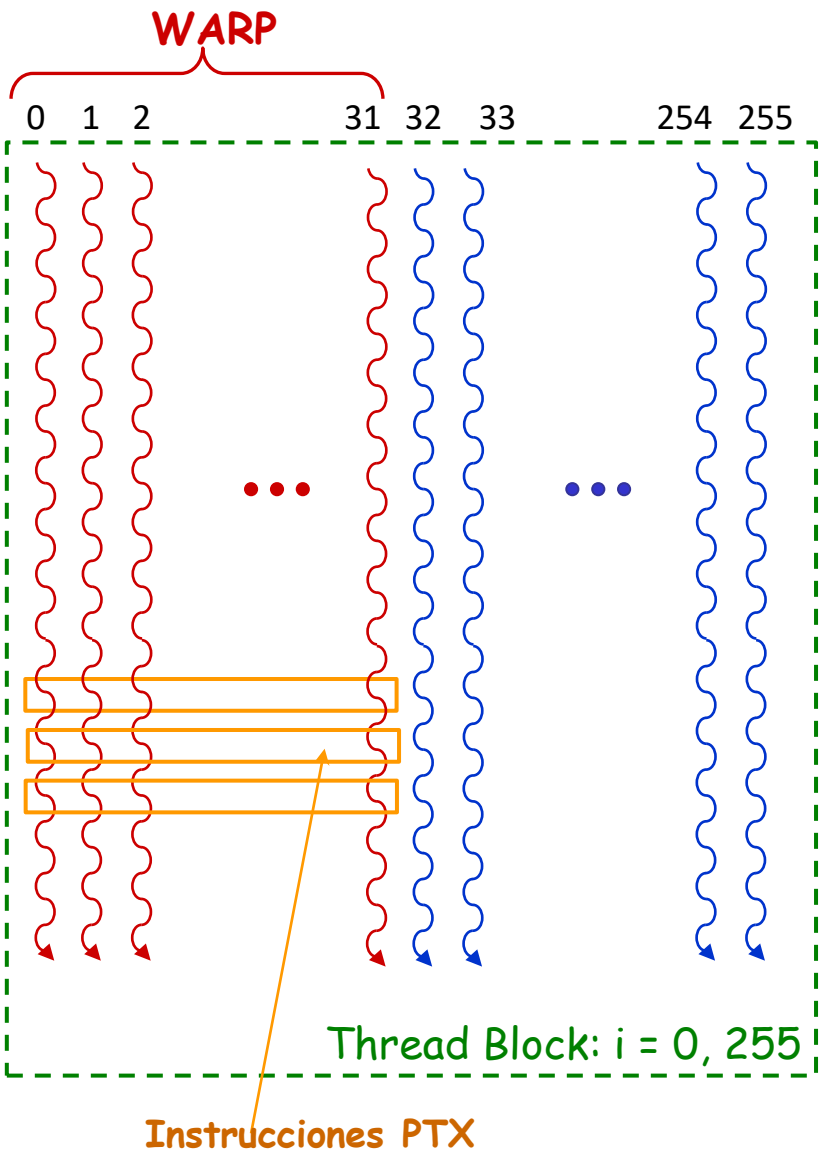
CUDA(Compute Unified Device Architecture)

- 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 bucle 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

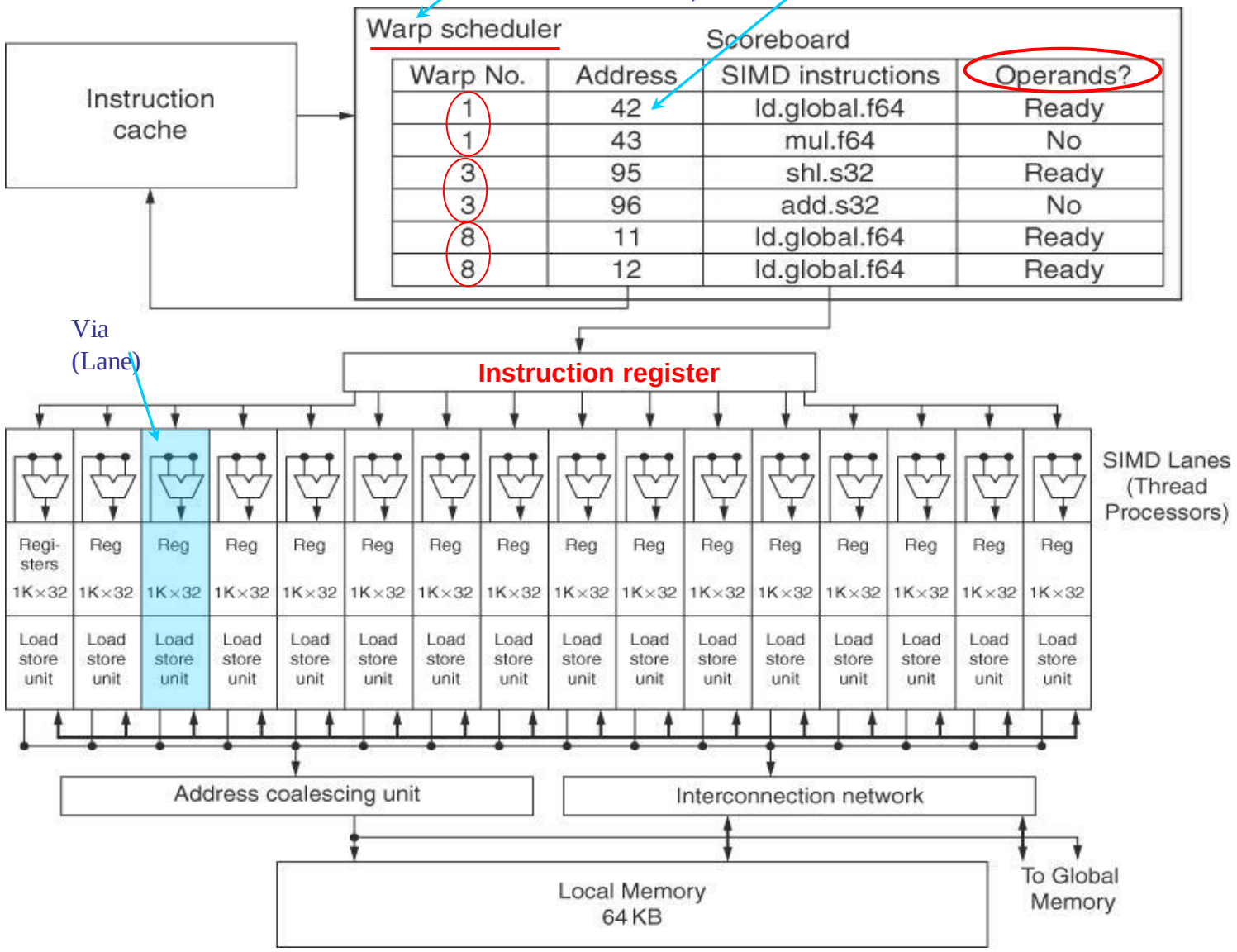
Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid (Malla) Ej. i = 0..9999	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that 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

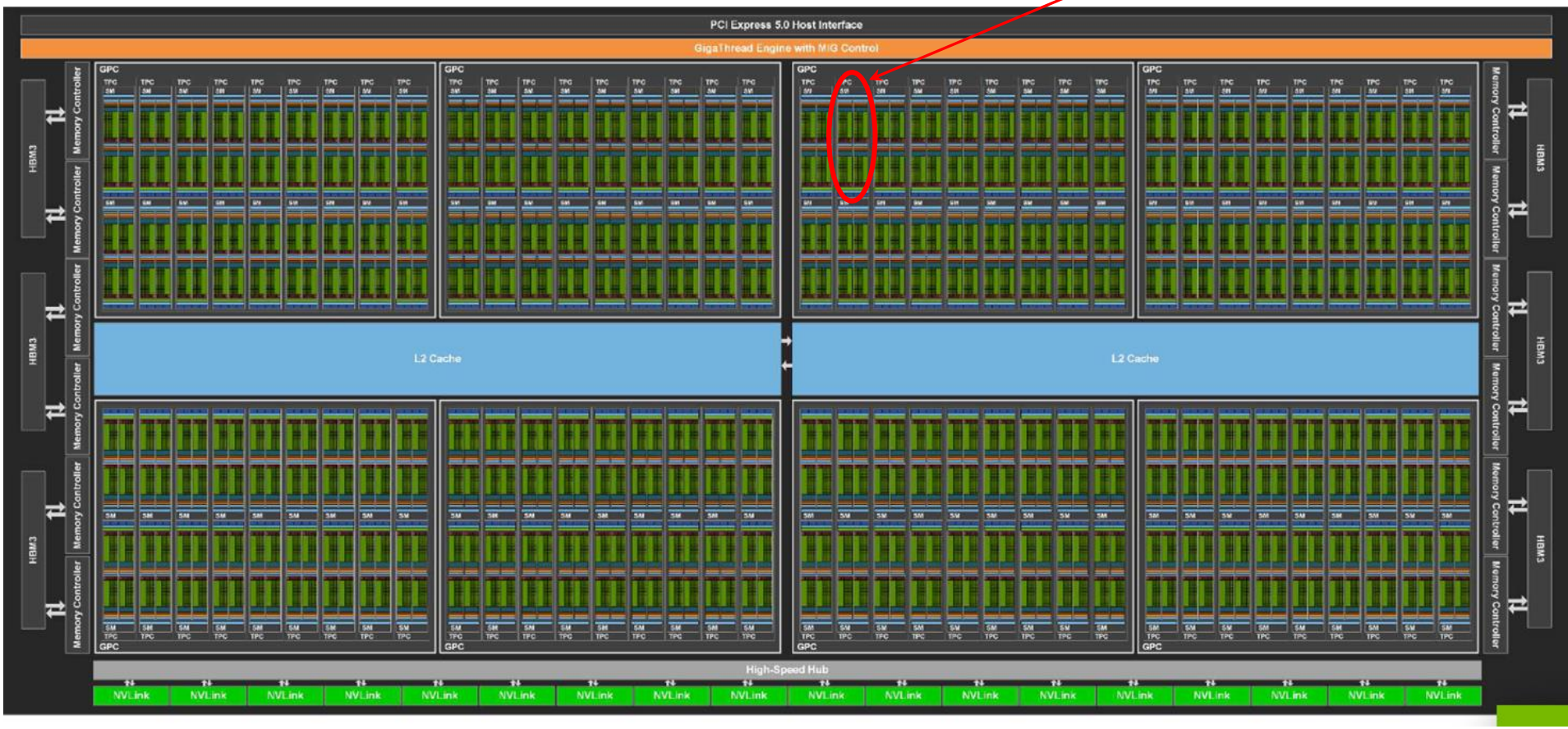


- Todas las vías ejecutan la misma instrucción
-Según la máscara, unas guardan el resultado y otras no

16 Lanes

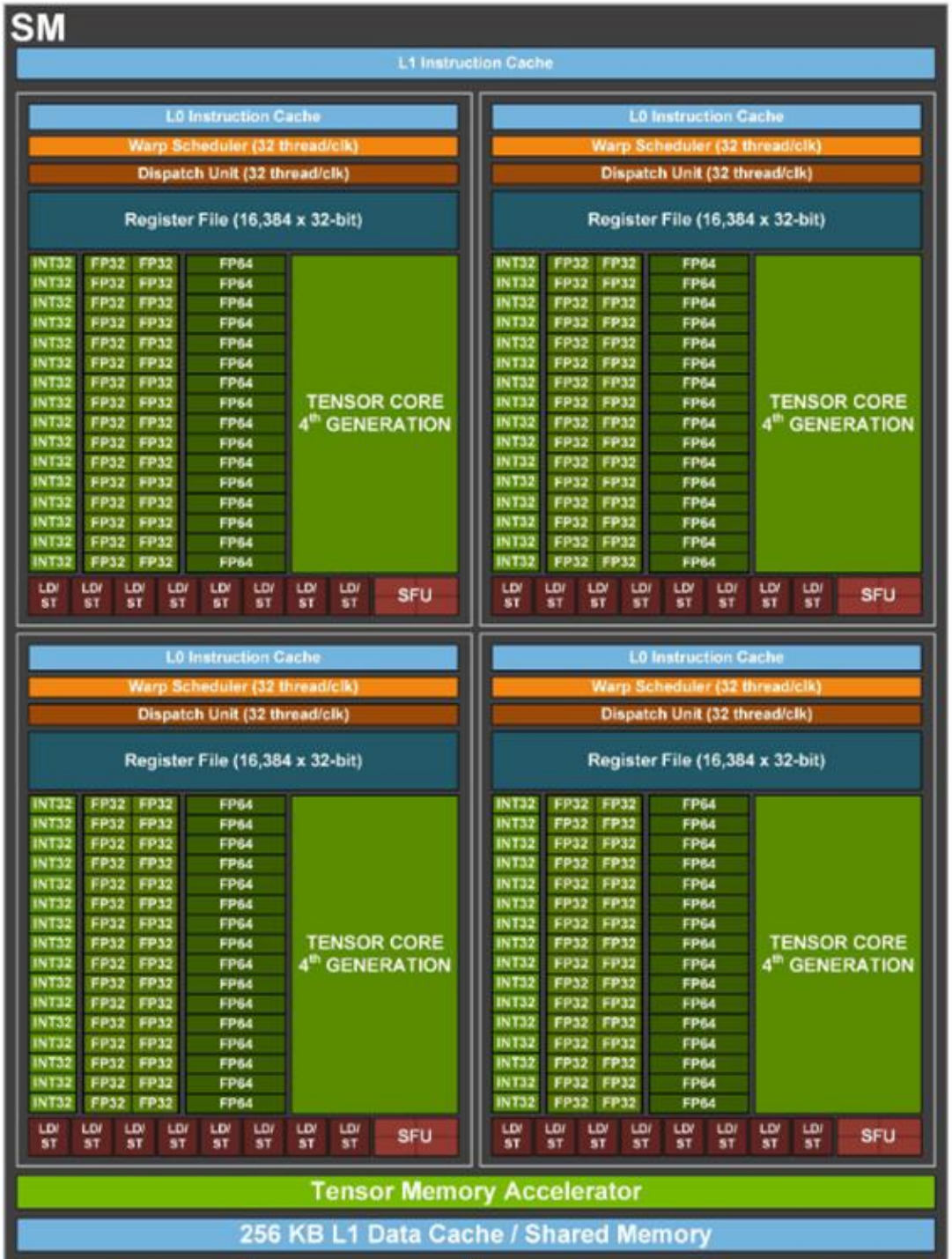
Ejemplo: GPU NVIDIA H100 (Arq. Hopper)

- Imagen tomada de https://resources.nvidia.com/en-us-tensor-core
- 144 procesadores SIMD MT (SM), 80 GB de memoria HBM3, 50 MB de cache L2



Hopper Streaming Multiprocessor (SM)

- https://www.nvidia.com/es-es/technologies/hopper-architecture/



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];
```

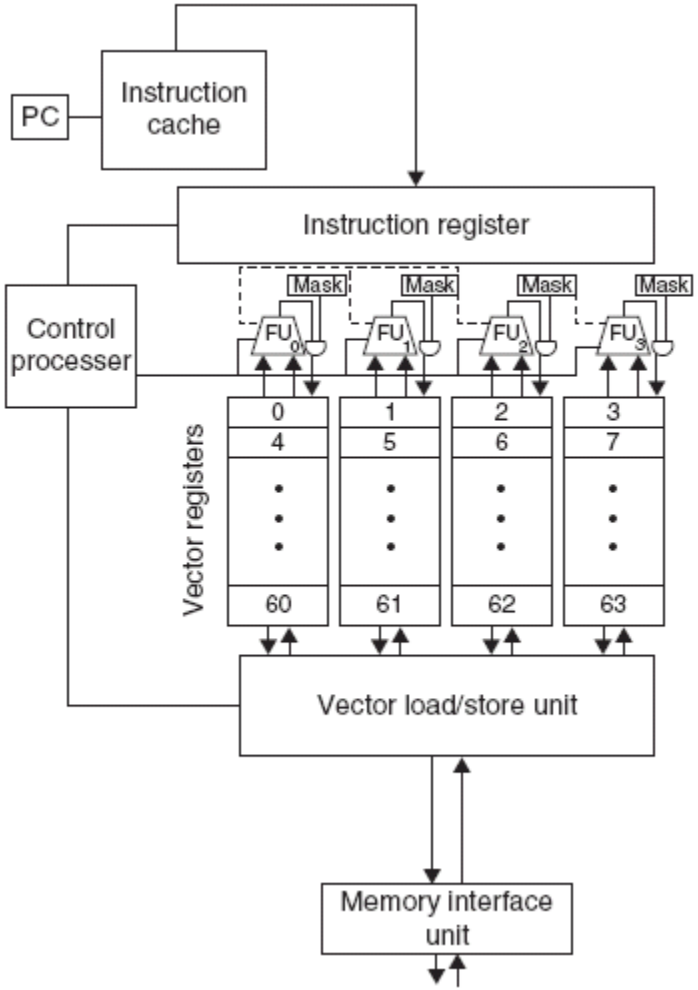
  - Código PTX

```
                                                ;  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
```

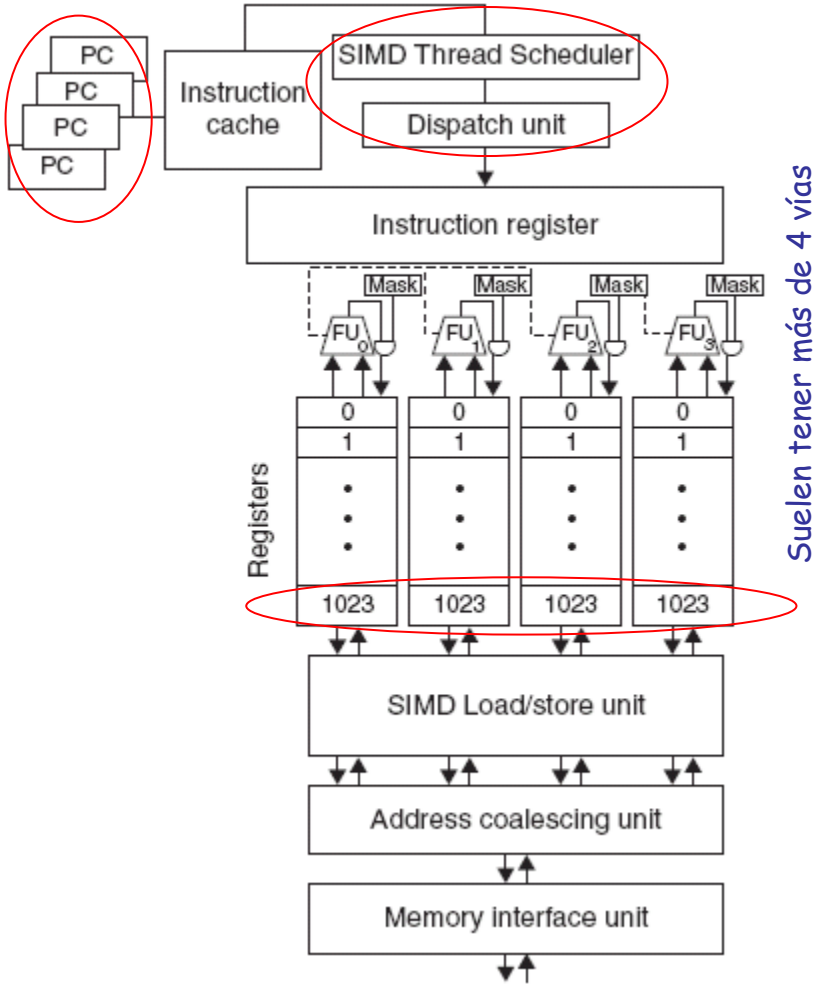
Ojo! Recordar que cada instrucción PTX procesa 32 elementos (1 WARP de 32 threads)

Comparación Procesador Vectorial - GPU

Procesador vectorial con cuatro vías



Procesador SIMD MT (4 PCs) con cuatro vías



Suelen tener más de 4 vías