Introducción a las arquitecturas paralelas

AdC 2022

Clasificación de Flynn

Flynn clasifica las arquitectura de computadoras en cuatro categorías:

- Single Instruction Single Data (SISD): Una instrucción opera sobre un único dato (ILP)

- Single Instruction Multiple Data (SIMD): Una misma instrucción opera sobre distintos datos (Procesadores vectoriales y matriciales) - SPMD

- Multiple Instruction Single Data (MISD): Distintas instrucciones operan sobre el mismo dato

- Multiple Instruction Multiple Data (MIMD): Distintas instrucciones operan sobre distintos datos (Multiprocessors, Multithreaded processors)

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Core i7

SIMD

Una misma operación (instrucción) se aplica a distintos datos.

La operación de distintas ALUs están sincronizadas y responden al mismo PC.

La motivación inicial es la de simplificar el control frente a una gran cantidad de ALUs.

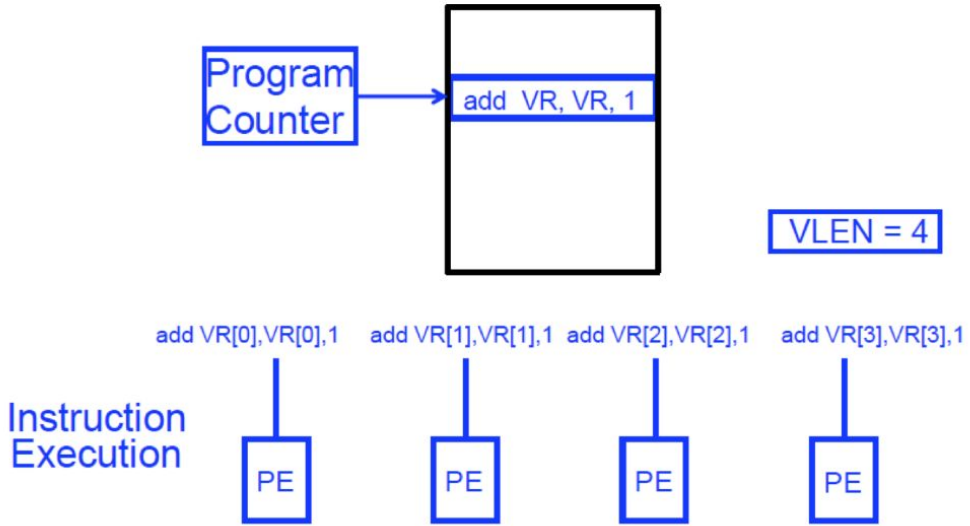
SIMD funciona muy bien para trabajar con bucles (for). Sin embargo, para esto debe existir una gran independencia entre los datos a procesar (data paralellism)

Facilita el trabajar con vectores (matrices) ya que la suma de cada coordenada es realizada por cada ALU.

Procesadores matriciales

La misma instrucción se aplica a distintos datos en paralelo.

Los elementos de proceso (PE) deben poder ejecutar cualquier instrucción



No confundir con VLIW donde cada elemento de proceso recibe una instrucción diferente

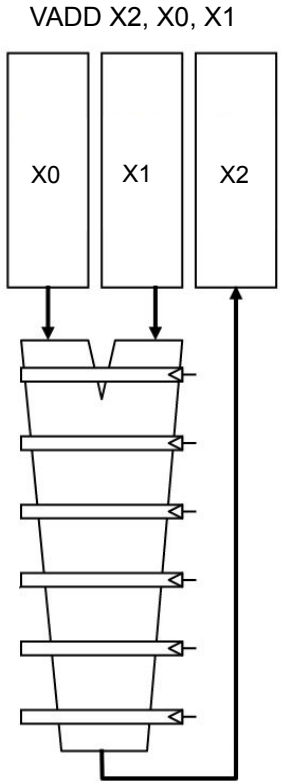
Procesadores vectoriales

Los elementos de procesos se segmentan (pipeline)

Algunos elementos de procesos están especializados en pocas tareas (multiplicar, sumar, de números enteros, de PF, etc.)

Existen registros vectoriales que contienen varios elementos del vector a procesar. Una arquitectura vectorial podría tener 32 registros de 64 elementos de 64 bits cada uno.

Los elementos de procesos se segmentan usando pipelines. Por ende una suma pasaba a realizarse en varios clocks. Esto permitia ir ejecutando por ej la suma de varios datos en varios elementos.



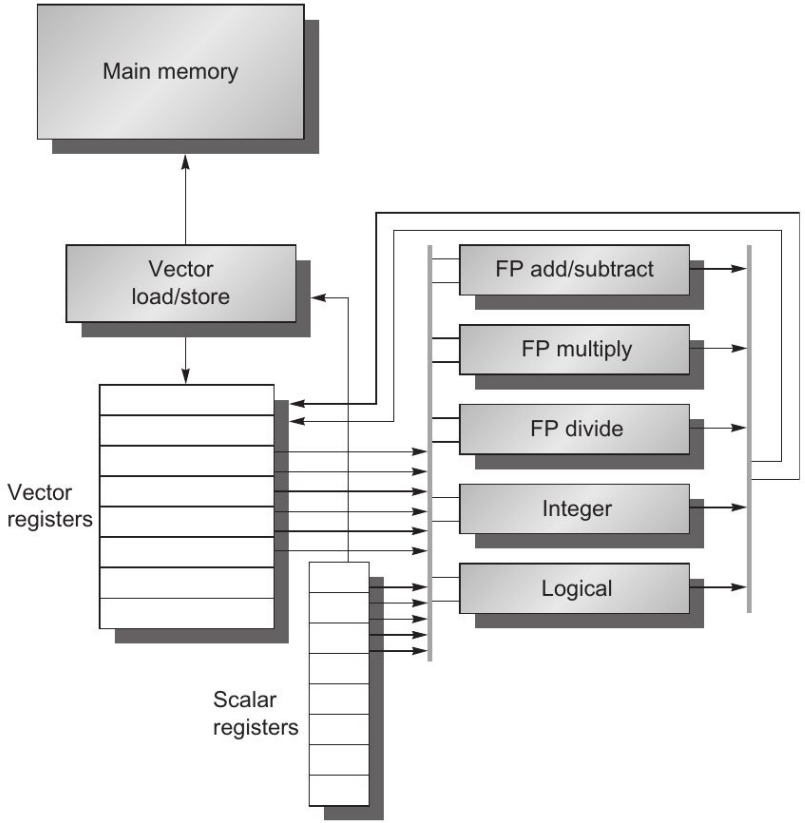
Procesadores vectoriales (RV64V)

Registros vectoriales: Cada registro contiene un vector de un determinado largo.

Unidades funcionales vectorizadas: Cada unidad está segmentada (pipeline) y en cada ciclo de reloj ingresa una nueva operación.

Unidad de load/store vectorial: lee o escribe un vector de memoria a los registros vectoriales.

En cada clock, cada unidad funcional esta realizando una op distinta.

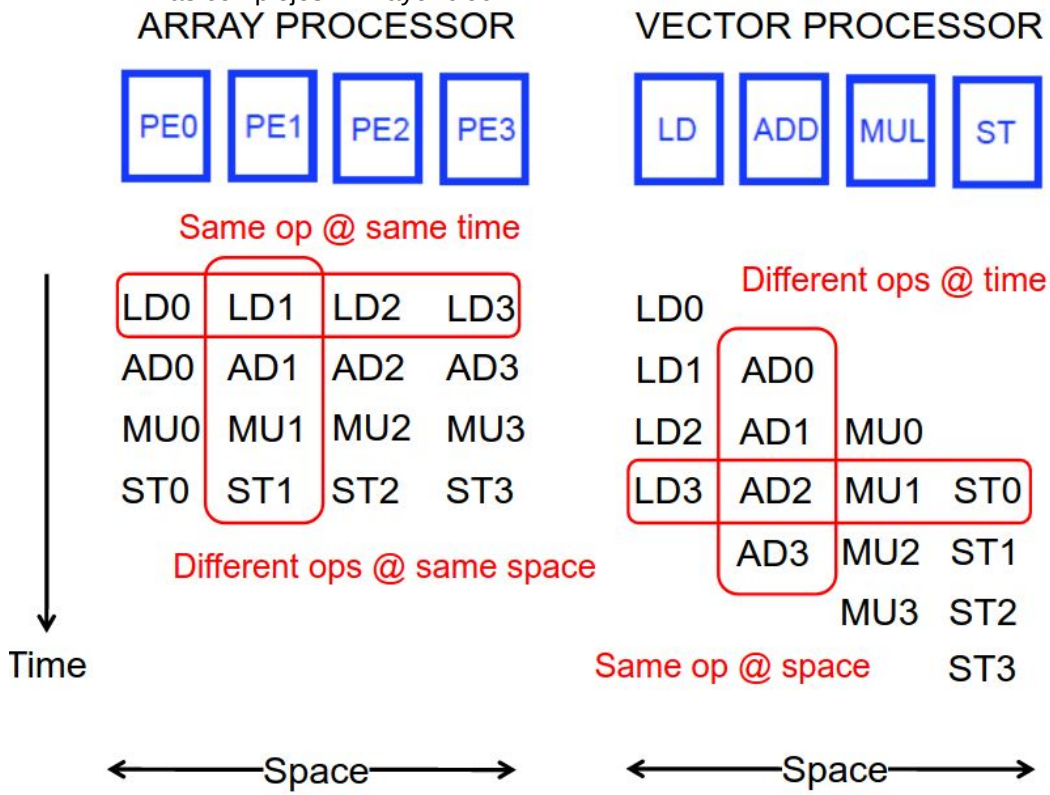


Procesadores matriciales vs vectoriales

```
LDURD D0, [X0]
VADDD D0, D0, 1
VMULD D0, D0, 2
STURD D0, [x0]
```

En estos, los elementos son mas complejos -> Mayor clock

En estos procesadores, cada elemento es menos complejo, por lo que sus clocks son mas cortos.



Ejemplo: DAXPY

Se pretende realizar la operación: $Y = a \times X + Y$ donde X e Y son vectores.

Este ejemplo es conocido como el DAXPY del benchmark Linpack. En LEGv8 quedaria:

```
        LDURD  D0,[X28,a]   //load scalar a
        ADDI   X0,X19,512   //upper bound of what to load
loop:   LDURD  D2,[X19,#0]  //load x(i)
        FMULD  D2,D2,D0     //a x x(i)
        LDURD  D4,[X20,#0]  //load y(i)
        FADDD  D4,D4,D2     //a x x(i) + y(i)
        STURD  D4,[X20,#0]  //store into y(i)
        ADDI   X19,X19,#8   //increment index to x
        ADDI   X20,X20,#8   //increment index to y
        CMPB   X0,X19       //compute bound
        B.NE   loop         //check if done
```

Ejemplo: DAXPY

Se pretende realizar la operación: $Y = a \times X + Y$ donde X e Y son vectores.

En instrucciones vectorizadas de LEGv8 quedaría:

```
LDURD   D0,[X28,a]  //load scalar a
LDURDV  V1,[X19,#0] //load vector x
FMULDVS V2,V1,D0    //vector-scalar multiply
LDURDV  V3,[X20,#0] //load vector y
FADDDV  V4,V2,V3    //add y to product
STURDV  V4,[X20,#0] //store the result
```

Trae un vector entero (conj de palabras) a un registro vectorizado.

Ejemplo: DAXPY

¿En que mejora esto el rendimiento?

- Se leen muchas menos instrucciones: 6 respecto a las casi 600 si el largo del vector es de 64 elementos.

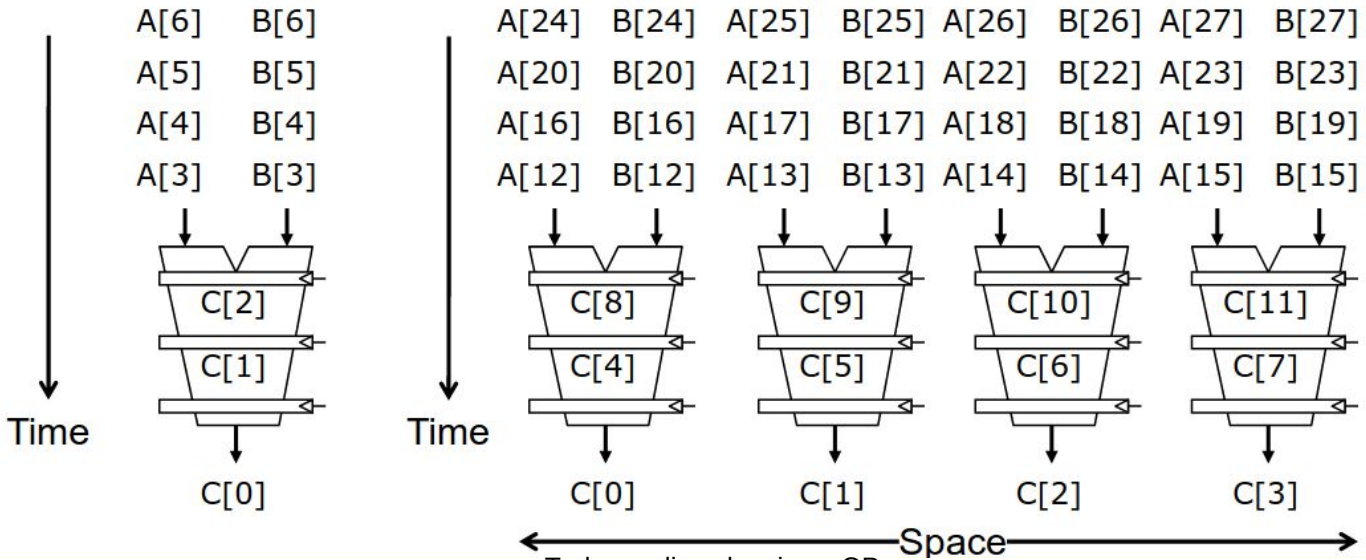
- Se reduce la cantidad de hazards: el procesador LEGv8 con pipeline, cada FADDD tiene que esperar al FMULD, cada STURD debe esperar al FADDD y cada FMULD debe esperar al LDURD.
En el procesador vectorial va a haber un stall en el primer elemento de cada vector, pero luego los datos van a fluir continuamente.

- Quitamos un salto condicional

- Al usar instrucciones vectorizadas, implicitamente se le está avisando al procesador que no va a haber dependencias de datos y por lo tanto, la lógica de verificación de hazards se simplifica.

Paralelización en procesadores vectoriales

Dado que las operaciones son independientes entre ellas, es posible agregar lanes que procesen estos datos en paralelo dentro de los PE

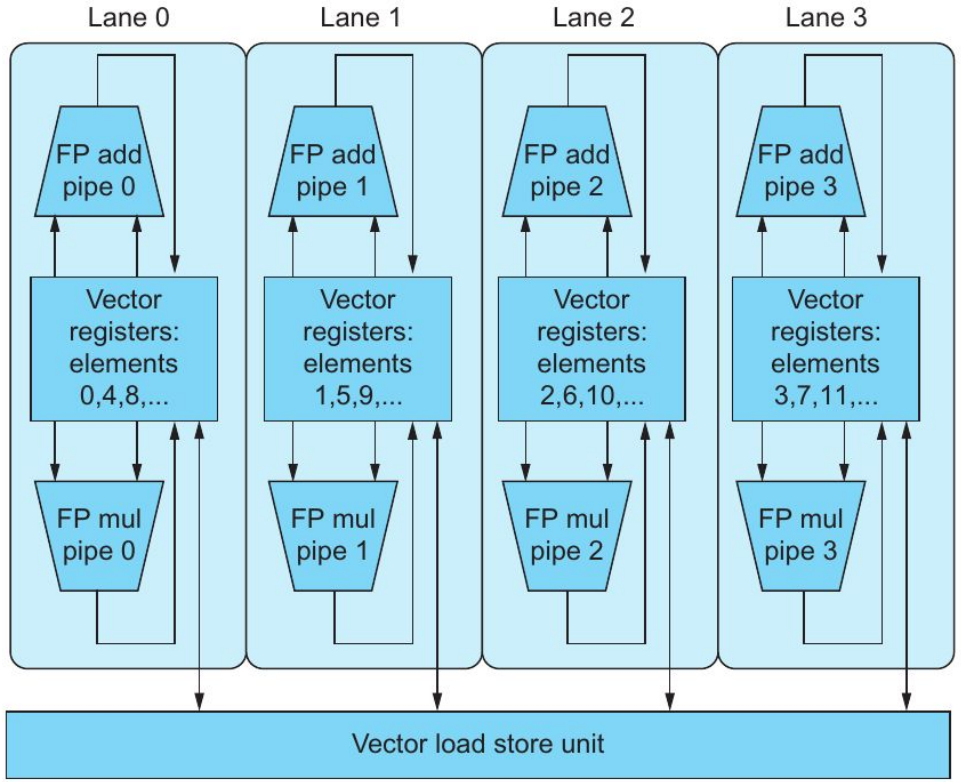


Todas realizan la misma OP.

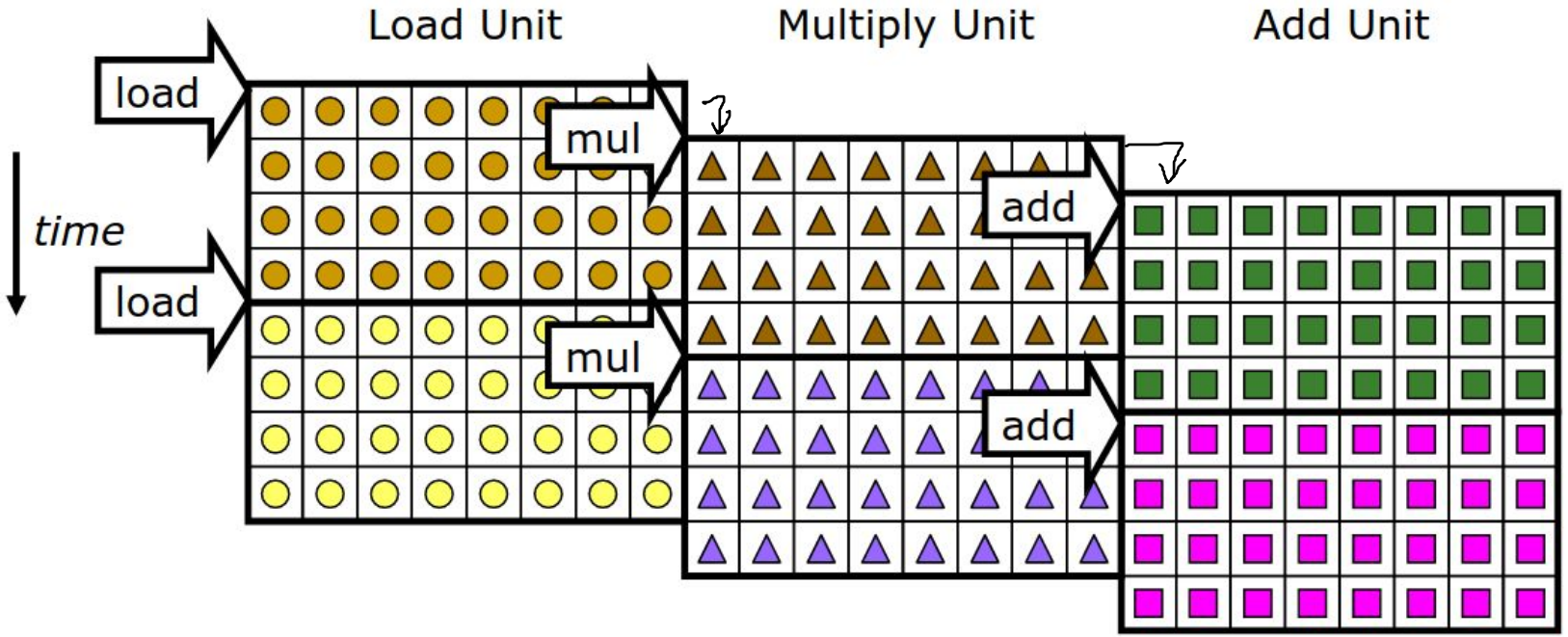
Paralelización en procesadores vectoriales

Que es una lane?
Una lane contiene elementos de operacion.
Y registros de vectores asociados a esos elementos de procesos.
Asociados a esa LANE.

Cada lane esta pipelineada internamente.



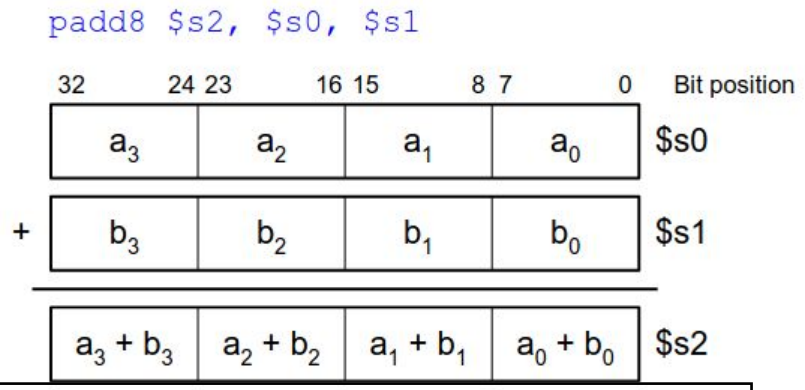
Paralelización en procesadores vectoriales



SIMD en instrucciones para gráficos

Las instrucciones SIMD son muy frecuentemente utilizadas para el procesamiento gráfico: Muchos datos independientes entre ellos.

Haciendo una pequeña modificación a la ALU es posible convertir operaciones comunes en operaciones vectoriales de enteros de pocos bits (8 o 16 bits). Solamente hay que anular el acarreo cada una cierta cantidad de bits.



Esto permite utilizar registros "comunes" para realizar operacionesvectoriales

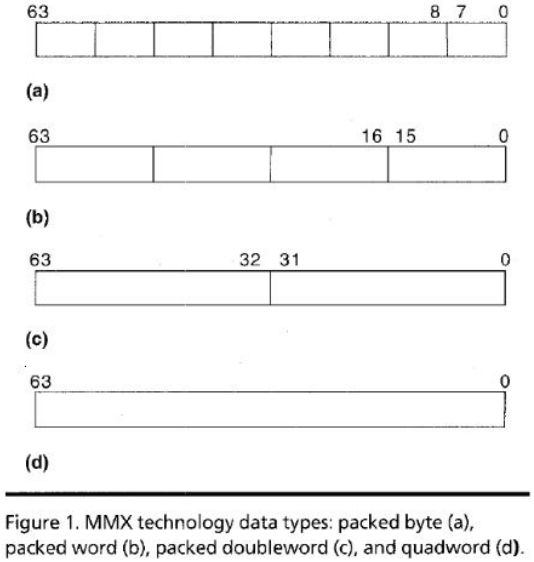


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

Multithreading

Multithreading

Thread: Una pequeña secuencia de código independiente que es gestionada por un planificador de tareas (generalmente el sistema operativo)

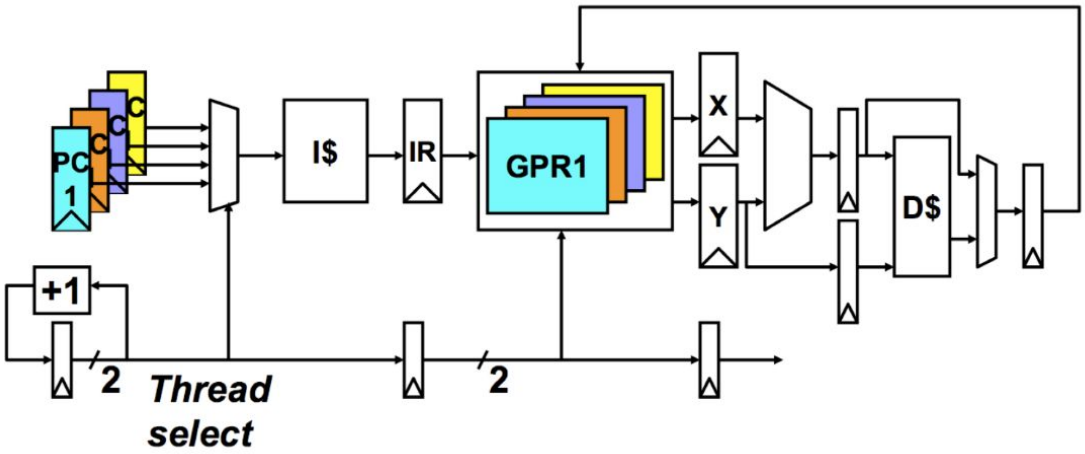
Un conjunto de instrucciones encapsulado.

El procesador cuenta con varios PC y registros, uno para cada thread.

Busca eliminar los stalls de las dep de datos.

Existen tres tipos de multithreading: Fine-grained, coarsed-grained y SMT

Cuando en un thread ocurre una DdD.Utiliza otro thread para aprovechar los ciclos que se pierden por stall. (Fine-grained multithread)



Multithreading

Fine-grained: En cada ciclo de clock se cambia de thread, es necesario que el procesador tenga la capacidad de cambiar de thread muy rápidamente.
El objetivo es evitar los stalls cortos. La desventaja es que se demora la ejecución de un thread único.

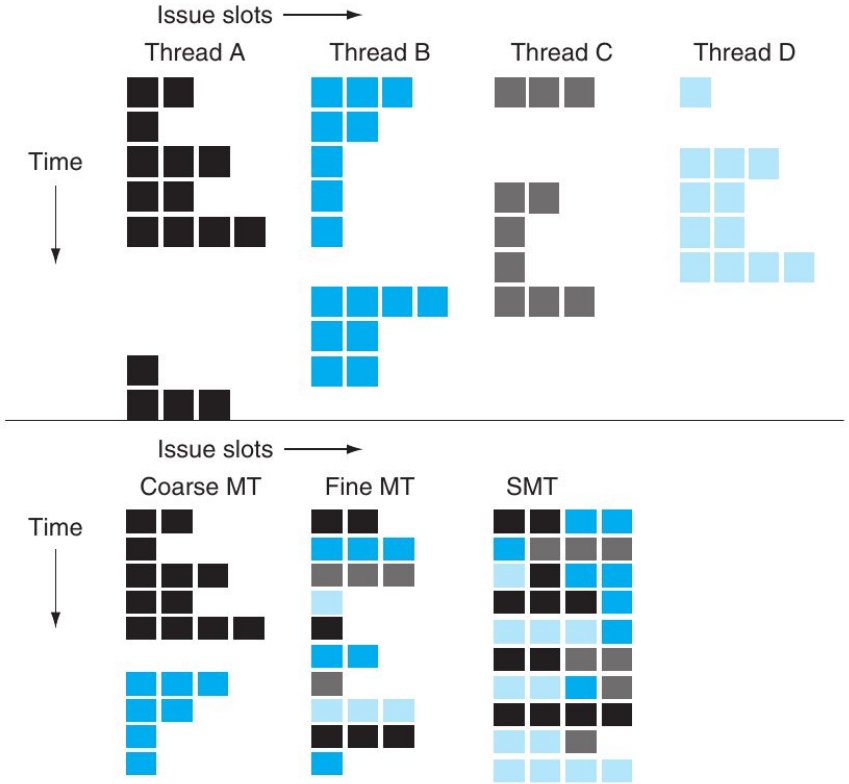
Coarsed-grained: Se cambia de thread cuando hay stalls largos, como fallos de caché.

Simultaneous multithreading (SMT): Utiliza los recursos de los procesadores dynamic multiple-issue para ejecutar en forma paralela instrucciones de distintos threads.
Por lo tanto, no se cambia de thread en ningun momento, constantemente se estan ejecutando instruciones de distintos threads.

Multithreading

Esta hecho como multiple issue.

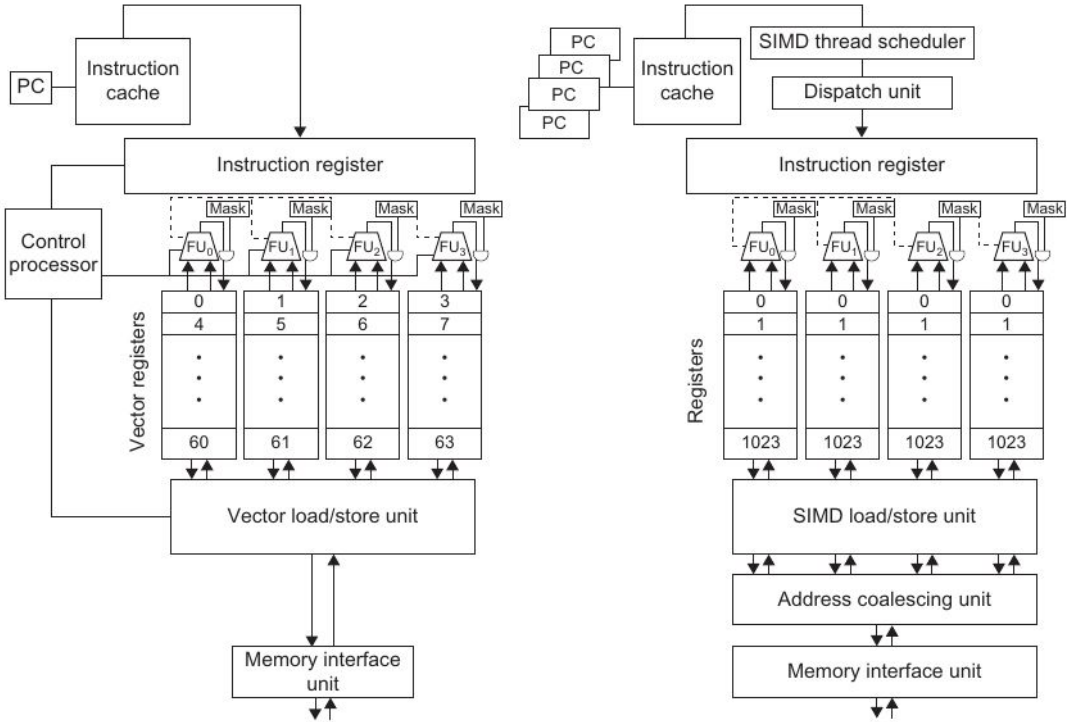
Cada cuadrado representa una operacion, y el vacio es una NOP.
En el grafico de arriba representa las instrucciones de cada Thread, y debajo como lo procesa cada tipo de procesador multithread.



GPU

GPU

En las GPUs se combinan los conceptos de SIMD y Multithreading para conformar los llamados *multithreaded SIMD processor*.



GPU

En las GPUs se combinan los conceptos de SIMD y Multithreading para conformar los llamados *multithreaded SIMD processor*.

Estos son similares a los procesadores vectoriales pero con muchas más unidades funcionales más simples (menos pipeline)

Por lo general, las GPU contienen varios procesadores SIMD multithreading, por lo que son MIMD.

Los threads están compuestos exclusivamente por instrucciones vectoriales. Por lo tanto, el procesador debería tener varias unidades funcionales para ejecutar esto en paralelo, las cuales se llaman *lanes*

Por ejemplo, si se tienen 16 lanes se necesitan 2 clocks para ejecutar una instrucción vectorial.

Una GPU puede tener un único *multithreaded SIMD processor* o muchos.

Arquitectura nvidia

Se le llama *grid* al código paralelizable que se va a correr en la GPU y que consiste en un conjunto de threads (*threads block*).

Ejemplo:

```
for(i=0; i< 8192; i++) {
      A[i] = B[i] * C[i]
}
```

Los 8k elementos no entran en un core, por lo que se subdividen en threads independientes. Ej 8192/4 elementos por thread. Ese conjunto de threads es la grid

En este caso se le llama grid al código que computa la multiplicación de estos dos vectores de 8192 elementos. Estos se subdividen en threads blocks de 512 elementos cada uno.

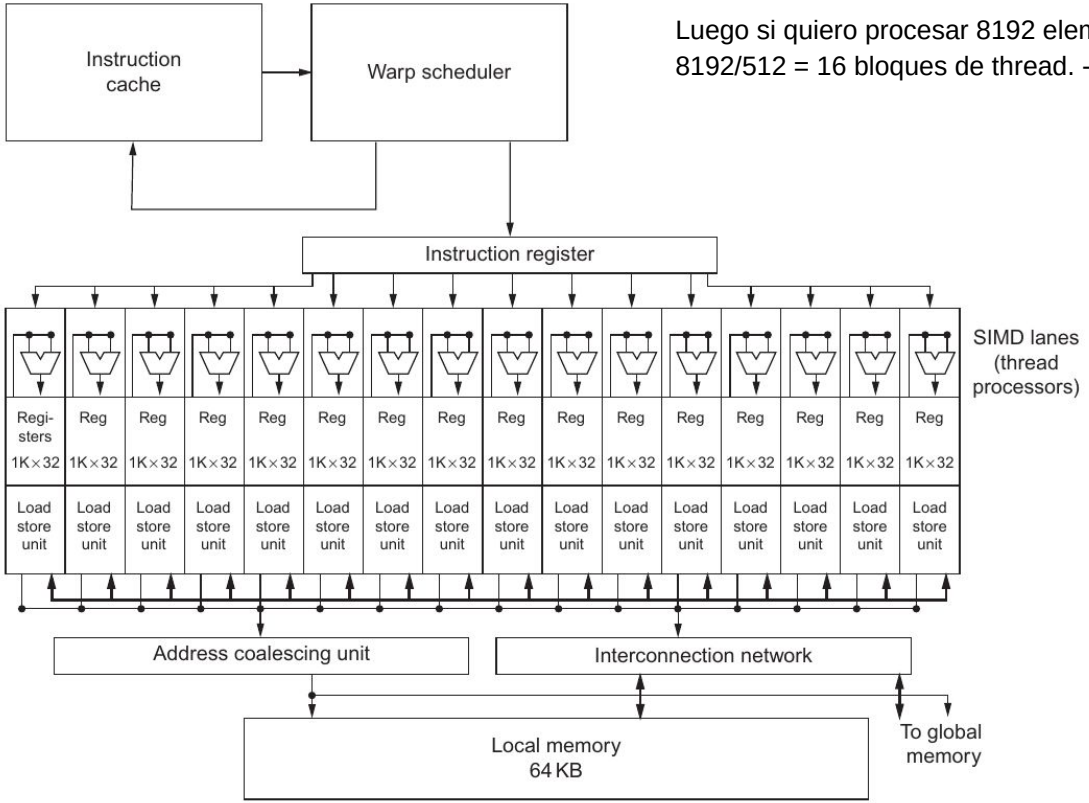
Cada instrucción SIMD computa 32 elementos en paralelo. Por lo tanto, se necesitan 16 threads blocks para este ejemplo.

Cada thread block es asignado a un *multithreaded SIMD processor* mediante el *Thread Block Scheduler*. Luego hay otro scheduler interno que elige entre estos threads para el procesado (warp scheduler o SIMD Thread Scheduler)

SIMD multithreading processor

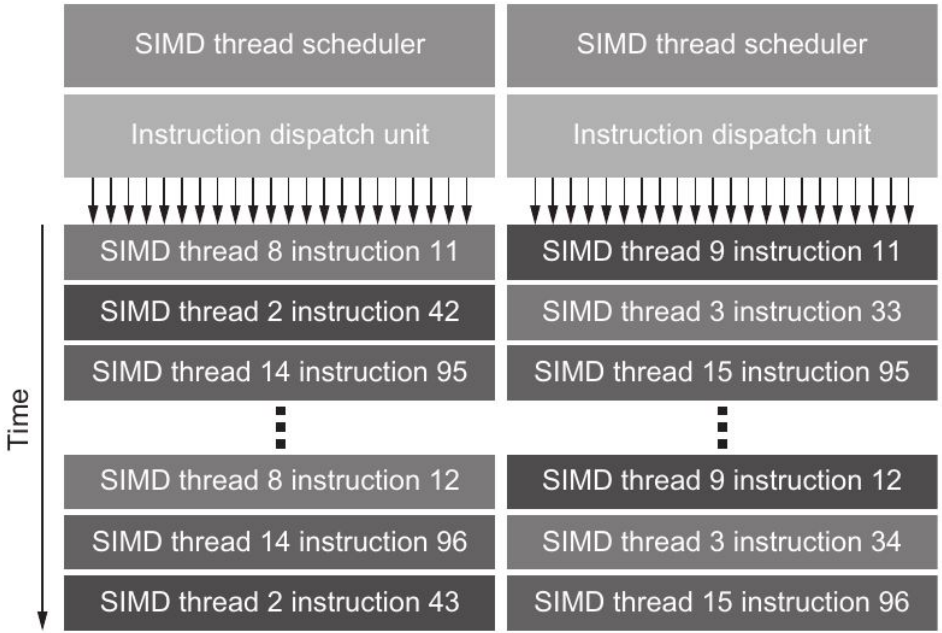
Cada SIMD tiene 16 lanes y que procesa 32 elementos cu.
Por lo que cada bloque puede procesar 512 elementos.

Luego si quiero procesar 8192 elementos. necesito 8192/512 = 16 bloques de thread. -> la grid.



Pascal P100

El *multithreaded SIMD processor* de la Pascal P100 tiene dos SIMD thread schedulers con la capacidad de trabajar con varios threads.



Pascal P100

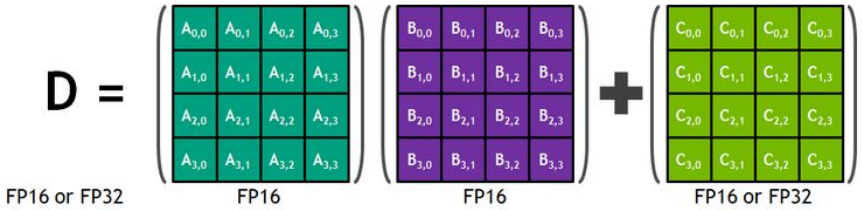

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink

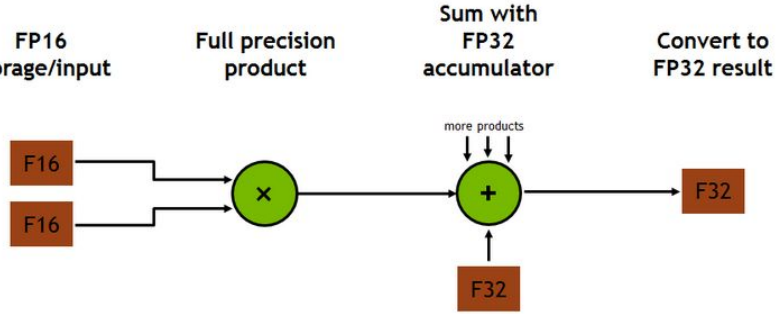
Volta GV100


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

Volta GV100


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
LD/ST
TENSOR CORE
SFU
128KB L1 Data Cache / Shared Memory
Tex


D =
FP16 or FP32
FP16
FP16
FP16 or FP32


FP16 storage/input
Full precision product
Sum with FP32 accumulator
Convert to FP32 result
more products
F16
F16
F32
F32

Paralelismo

¿Por qué es necesario el paralelismo?

Aunque se continúa mejorando la arquitectura de los procesadores de un núcleo, las mejoras ya no son tan sustanciales.

Se alcanzó el límite de la potencia que es posible disipar, esto limita la frecuencia máxima de trabajo del procesador. Incluso con nuevas tecnologías de fabricación el aumento es bajo

Sin embargo, es posible seguir aumentando la densidad de transistores por área de silicio al reducirse continuamente el tamaño de los transistores.

La estrategia que se viene usando los últimos años es: en lugar de hacer más complejos los procesadores, poner varios en paralelo.

Paralelismo

Existen dos grandes tipos de paralelismo:

- Paralelismo de datos: la misma tarea se ejecuta sobre distintos datos
- Paralelismo de tarea: diferentes tareas de un problema se aplican al mismo tiempo

Ningún dispositivo es bueno para ambos tipos de paralelismo:

- Los procesadores multicores superescalares con dynamic scheduling son la mejor opción para paralelismo de tarea

Las GPU para el paralelismo de datos

Ejemplo

Se desea paralelizar el siguiente código que calcula un valor para cada índice y luego acumula los resultados.

```
sum = 0;
for (i = 0; i < n; i++) {
    x = computeValue(i);
    sum += x;
}
```

¿Cómo conviene paralelizar si tenemos p cores?

Ejemplo

Si p << i, cada core ejecuta:

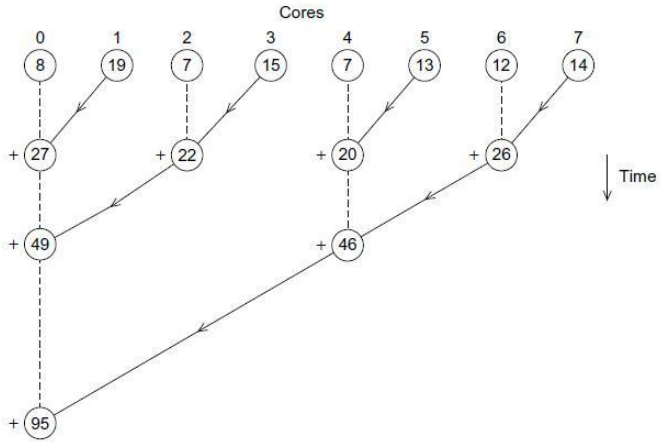
```
my_sum = 0; // local variable in p-core
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
      my_x = computeValue(my_i);
      mysum += my_x;
}
```

Luego, un "master core" acumula los resultados parciales

```
if (I’m the master core) {
      sum = my x;
      for each core other than myself {
            receive value from core;
            sum += value;
} else {
      send my x to the master;
}
```

Ejemplo

Si *i* es cercano a *p*, el último proceso se vuelve un cuello de botella.
Es posible optimizar esto con un árbol de suma.
Con 8 cores y p=i, con el primer método se hacen 7 sumas y con el segundo 3.
Con 1000 cores, de 999 se mejora a 10, es decir, 100 veces mejor.



Descomposición

Cuando se piensa en paralelizar un problema, se utiliza el concepto de descomposición:

- Descomposición de tareas: El algoritmo se subdivide en tareas sin analizar los datos.

- Descomposición de datos: Los datos se subdividen en grupos a los que se le aplica tareas en paralelo.

A su vez, la descomposición de datos puede hacerse respecto a la entrada o la salida:

- Respecto a la salida: A partir de los resultados que necesito realizar, se subdividen los datos de entrada para obtener un elemento de salida. Ej: multiplicación de vectores.

- Respecto a la entrada: Simplemente se subdividen los datos en grupos iguales. Ej: buscar el número de ocurrencias en un string.

Ejemplo

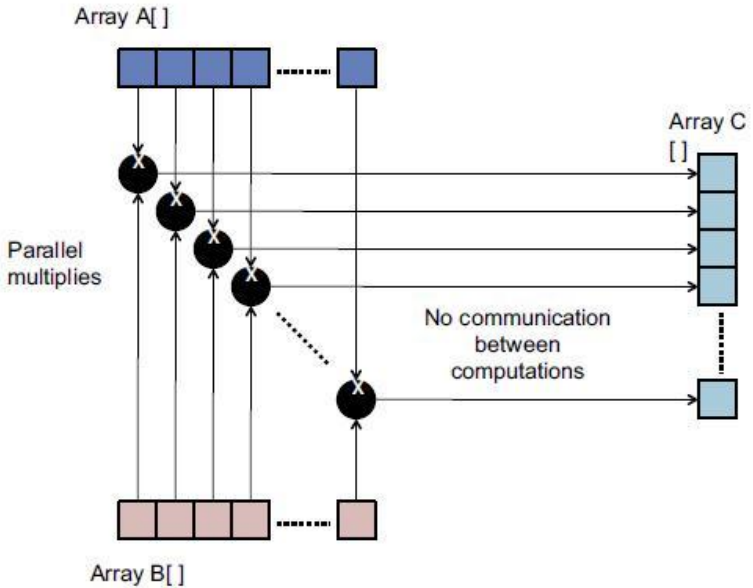


FIGURE 1.2

Multiplying two arrays: This example provides for parallel computation without any need for communication.

Coordinación entre cores

Cuando las tareas entre los cores no son completamente independientes, es necesario coordinar la ejecución.

Las comunicación entre cores se da mediante la memoria:

- Memoria distribuida: La comunicación se da mediante "Message passing"
- Memoria compartida: Se sincroniza mediante operaciones de lectura y escritura.

Es importante balancear la carga para que todos ejecuten cantidades similares de trabajo y se minimicen las sincronizaciones/comunicaciones entre cores.

Loop stripe mining

Es una técnica de transformación de bucles que permite que varias iteraciones:

- Se ejecuten en paralelo (SIMD)

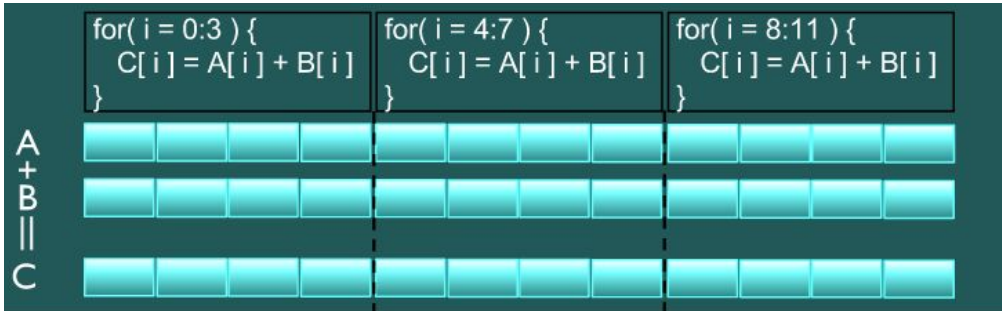
- Se separen en distintas unidades de proceso (CPU multi core)

- O ambas (GPU)

Loop stripe mining

```
for(i=0; i<N; i++) {
    C[i] = A[i] * B[i];
    }
```


```
n=N/p
for(p=0;p<core;p++) {
    for(i=p*n; i<(p*n)+n; i++) {
        C[i] = A[i] * B[i]; }
    }
```



SPMD

Single Program Multiple Data (SPMD): Se ejecutan muchas instancias del mismo programa independientemente donde cada instancia trabaja sobre distintos datos

La diferencia con SIMD es que en SPMD cada unidad de proceso puede estar en distintos puntos del programa. Al mapear un programa en un thread, se convierte en SIMT.

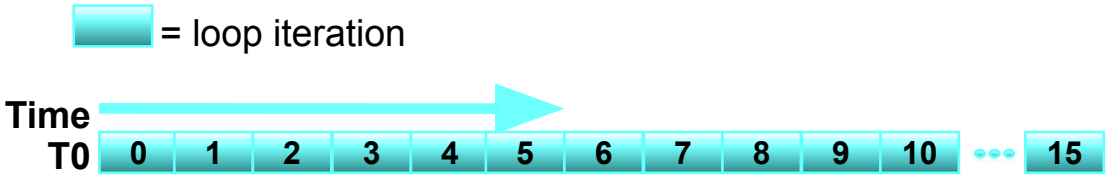
La combinación con loop stripe mining es una técnica de paralelización muy utilizada.

En CPUs hay relativamente pocos threads que se pueden crear porque crear/cambiar entre threads es costoso. En las GPU el costo es muy bajo, por lo que es posible crear un thread por cada iteración del loop.

SPMT

Single-threaded (CPU)

```
// there are N elements
for(i = 0; i < N; i++)
  C[i] = A[i] + B[i]
```



Multi-threaded (CPU)

```
// tid is the thread id
// P is the number of cores
for(i = 0; i < tid*N/P; i++)
  C[i] = A[i] + B[i]
```

Massively Multi-threaded (GPU)

```
// tid is the thread id
C[tid] = A[tid] + B[tid]
```