



# Universidad de Alcalá

Escuela Politécnica Superior

Universidad de Alcalá

## **Ampliación de Programación Avanzada**

### **Tema 1 - Cuda**

Grado en Ingeniería Informática – Curso 2018/2019

# APA – TEMA 1 – UNIDAD 1

## COMPUTACIÓN PARALELA

### DEFINICIONES

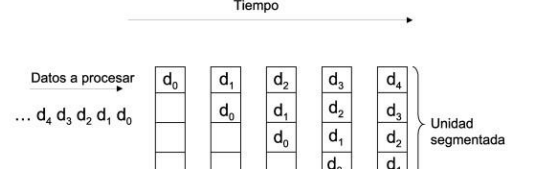
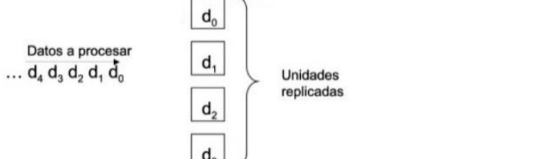
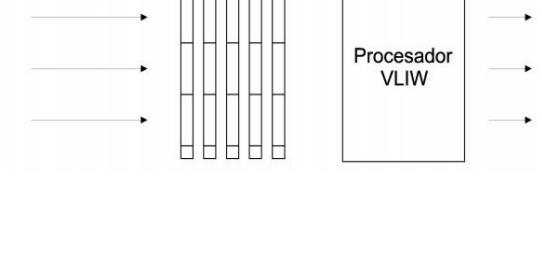
- Definición general: consiste en ejecutar muchas instrucciones en poco tiempo.
- Definición para monoprocesadores: se consigue ejecutando instrucciones solapadas simultáneamente.
- Definición para multiprocesadores: se consigue dividiendo el problema en partes y ejecutar cada parte en una CPU.

Se realiza para resolver problemas que no caben en una sola CPU o acelerar su resolución.

### PARALELISMO EN MONOPROCESADORES

#### TÉCNICAS

Sea una unidad funcional la unidad de procesamiento de información, el paralelismo se consigue mediante distintas técnicas.

<p><b>Segmentación:</b> se divide la unidad funcional en etapas y en cada ciclo se ejecuta cada una de esas partes, intercalando registro para datos intermedios.</p>	
<p><b>Replica:</b> se tienen varias unidades funcionales, que procesan instrucciones atómicamente.</p>	
<p><b>VLIW:</b> Se utilizan instrucciones con ancho de palabra superior al que gestiona la unidad funcional. Esta instrucción contiene instrucciones de ancho de palabra de unidad funcional. Utilizando varias UF, se consigue mayor procesamiento de instrucciones por segundo.</p>	

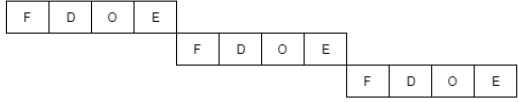
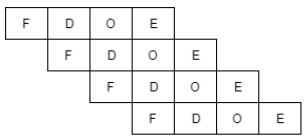
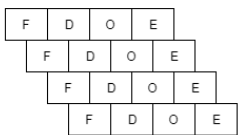
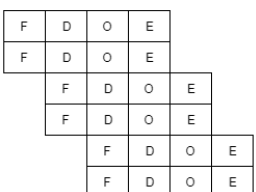
## IMPLEMENTACIONES

**F = Fetch**

**D = Decodificación**

**O = Operación**

**E = escritura**

<p><b>Procesador escalar</b></p> <p>1/4 instrucción / ciclo Implementación básica.</p>	<p>Tiempo →</p> <p>Instrucciones ↓</p> 
<p><b>Procesador segmentado</b></p> <p>1 instrucción / ciclo. Se segmenta el proceso de ejecución de instrucción. Tras etapa inicial de llenado se consigue una instrucción por ciclo.</p>	<p>Tiempo →</p> <p>Instrucciones ↓</p> 
<p><b>Procesador supersegmentado</b></p> <p>1 instrucción / ciclo acelerada. Se lanzan subetapas sin completar el ciclo de reloj.</p>	<p>Tiempo →</p> <p>Instrucciones ↓</p> 
<p><b>Procesador superescalar</b></p> <p>n instrucciones / ciclo. Se lanzan varias instrucciones de forma simultánea.</p>	<p>Tiempo →</p> <p>Instrucciones ↓</p> 

## PARALELISMO EN MULTIPROCESADORES

El paralelismo requiere y se logra en CPU y MEMORIA.

### PARALELISMO EN CPU

#### TIPOS

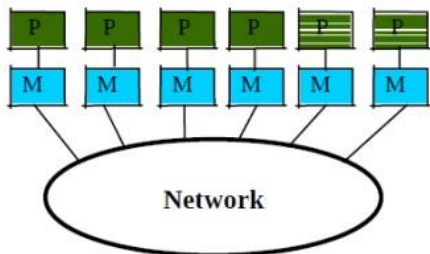
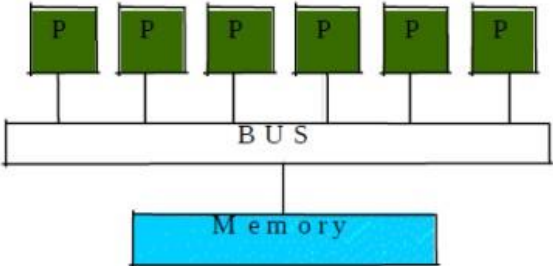
- Paralelismo interno: queda oculto al usuario.
- Paralelismo externo: queda visible al usuario.
- Paralelismo temporal: emplea segmentación.
- Paralelismo espacial: emplea réplica.

#### TAXONOMÍA DE FLYNN

Nombre	Descripción	#UC's	#ALU's
SISD	1 instrucción actúa sobre 1 dato	1	1
SIMD	1 instrucción actúa sobre n datos	1	n
MISD	n instrucciones actúan sobre 1 dato	n	1
MIMD	n instrucciones actúan sobre m datos	n	m

### PARALELISMO EN MEMORIA

Se puede lograr mediante 2 implementaciones:

MEMORIA DISTRIBUIDA	MEMORIA COMPARTIDA
Cada procesador tiene su memoria local. Intercambio de datos mediante paso de mensajes. El tiempo de acceso al dato está determinado por su ubicación. Se busca un compromiso entre velocidad y gasto de recursos en arquitectura de red. Topología de red en cubo, en círculo...	Se distinguen 2 tipos: <ul style="list-style-type: none"><li>• UMA: tiempo de acceso uniforme a memoria. Caché en cada procesador.</li><li>• NUMA: tiempo de acceso no uniforme, determinado por la ubicación del dato. Acceso local más rápido. Memoria compartida distribuida pero no global.</li></ul>
	

Todo esto apoyado por la jerarquía de memoria.

# APA – TEMA 1 – UNIDAD 2

## ARQUITECTURA CUDA

### CPU VS GPU

Existen marcadas diferencias entre el diseño de la CPU y la GPU.

CPU	GPU
Optimizadas para ejecución secuencial	Optimizadas para ejecución paralela
Instrucciones complejas	Instrucciones simples
Pocos núcleos	Muchos núcleos
Mucha caché	Poca caché
Mayor hardware de control	Mayor hardware de cálculo
Menor poder/precio	Mayor poder/precio
Programación más flexible	Programación más restrictiva

### COMPONENTES DE UNA GRÁFICA

- GPU: placa con núcleos.
- Memoria de vídeo.
- Conversor señal digital/analógica.
- Disipador y ventilador.
- Alimentación.
- Conexión con placa base.

### LENGUAJES Y APIS

- OpenGL: primeras implementaciones después de ensamblador.
- CUDA: desarrollado por Nvidia y basado en C++.
- OpenCL: framework con compilador basado en C99 y APIs.

### USOS

- Minería de datos y bitcoins
- Blockchain
- Predicción de mercados
- Análisis
- Juegos

## HISTORIA DE LAS GPU'S

Generación	Detalles
Primera	Poco rendimiento. Directx5. Años 80.
Segunda	Años 90. HW configurable pero no programable. Se populariza DirectX y OpenGL.
Tercera	Años 2000. Vertex shader y pixel shader programables. Multisampling y antialiasing. Soporte para transform y lightning.
Cuarta	Años 2003. Procesadores con DDR2. Vertex y Pixel Shader 2.0. Mejora de compresión en texturas. Desprivatización de instrucciones floating point de la fase de vertex shader y transform and lightning.
Quinta	Años 2005. Aumento de rendimiento considerable. Permite programación de Vertex y Pixel Shader. Xbox 360: procesador unificado que permite VS y PS en un mismo procesador. DirectX10: procesa todos los vértices en una primitiva.

## PROCESO DE RENDERIZACIÓN

**Renderización:** proceso mediante el cual se interpreta una escena 3D y se genera una imagen 2D a partir de dicha escena.

Inicialmente, a la GPU le llega la información de la CPU en forma de vértices.

En la GPU se llevan a cabo los siguientes procesos:

1. Vertex shader: operaciones de píxeles.
  - Procesado de vértices: se divide el modelo en triángulos y se elabora una lista de todos los vértices. Estos vértices contienen arrays con información sobre luz, color, etc.
  - Agrupación en primitivas: los vértices se unen formando triángulos geométricos agrupados en primitivas.
2. Rasterización: operaciones de primitivas.
  - Rasterización: proceso mediante el cual se toma una imagen descrita en gráficos vectoriales (conjunto de primitivas) y se transforma en un conjunto de píxeles llamados fragmentos.
  - Clipping: tras transformar las primitivas a píxeles proyectándolos sobre una superficie bidimensional, parte de estos píxeles pueden quedar fuera de la ventana de visualización. El proceso de clipping "recorta" o descarta estos píxeles, de forma que no se procesen de cara a la imagen final.
  - Culling: al proyectar una escena 3D sobre una superficie 2D, hay píxeles que quedan "ocultos" al espectador, detrás de otros elementos o mirando a lado opuesto. El proceso de culling elimina píxeles que el espectador no puede ver.
3. Pixel Shader: operaciones de píxeles.

- Agrupación en fragmentos: proceso en el que los triángulos bidimensionales se rellenan de píxeles.
- Aplicación de texturas y luz.
- Coloreado de píxeles.
- Antialiasing: proceso en el que se dota de suavidad a los bordes de distintos colores y texturas a los píxeles.
  - i. Se unen los píxeles de color similar mediante triángulos.
  - ii. Se aplica un suavizado de color en las aristas de estos vértices.

# APA – TEMA 1 – UNIDAD 3

## ARQUITECTURA CUDA

### ARQUITECTURA DESDE EL PUNTO DE VISTA DEL PROGRAMADOR

Se utiliza un modelo de computación heterogéneo, utilizando la CPU como unidad de procesamiento de control y la GPU como unidad de cálculo.

Dispositivo	Referido como	Sintaxis de variables
CPU	Host	h_xx
GPU	Device	d_xx

### JERARQUÍA DE ESTRUCTURAS

Distinguimos entre hilos, bloques y kernels.

#### HILOS

Son la base de ejecución del código en la GPU. Contienen operaciones paralelas, y datos para el cálculo de esas operaciones.

Una vez en la GPU, a cada hilo se le asigna un identificador. Utilizamos este identificador para acceder a los datos con los que se vaya a trabajar.

#### BLOQUES

Conjunto de hilos concurrentes con código idéntico y datos similares, que pueden cooperar entre ellos a través de mecanismos de sincronización, y compartir accesos a un espacio de memoria de cada bloque.

Los bloques se pueden agrupar en mallas de una, dos o tres dimensiones.

### JERARQUÍA DE FUNCIONES

Se tienen 3 tipos de funciones distinguidas:

Nombre	Especificador	Llamada desde	Ejecutada en
Método llamador	__host__	Host	Host
Kernel	__global__	Host	Device
Función aux. GPU	__device__	Device	Device



## MÉTODO LLAMADOR

Esta función se ocupa de declarar/recibir los datos de CPU y llamar al kernel.

Pasos	Usando
1. Recibir / declarar datos.	Declaraciones con punteros.
2. Reservar espacio en memoria de device para los datos.	<code>cudaMalloc((void**) &amp;d_xx, bytes_d_xx)</code>
3. Transferir datos a device.	<code>cudaMemcpy(d_xx, h_xx, bytes_d_xx, cudaMemcpyHostToDevice)</code>
4. Llamada al kernel.	<code>kernel &lt;&lt;&lt; bloques, hilos &gt;&gt;&gt; (args)</code>
5. Transferir resultados de device a host.	<code>cudaMemcpy(d_xy, h_xy, bytes_d_xy, cudaMemcpyDeviceToHost)</code>

En la llamada al kernel, se le debe especificar el número de hilos por bloque, y el número de bloques, y la memoria compartida:

```
dim3 grid_bloques(100, 50); // Grid de bloques.
dim3 dimension_bloque(4, 8, 8); // Hilos por bloque.
size_t memoria_compartida = 64; // Bytes de memoria compartida.
nombre_kernel <<< grid_bloques, dimension_bloque, memoria_compartida
>>>(argumentos);
```

Los argumentos deben de ser pasados por referencia (punteros) a los datos en la memoria de device.

## FUNCIÓN KERNEL

Esta función contiene el código del hilo. Debe de devolver siempre void.

Pasos	Usando
1. Acceso a operandos.	<code>threadIdx.x</code> , <code>blockId</code> .
2. Operaciones.	Multiplicaciones, etc.
3. Sincronización si es necesaria.	<code>__synchronize()</code>
4. Escritura de resultados por referencia.	

```
__global__ void kernel(int* d_v1, int* d_v2, int* d_vr)
{
    // Lectura operandos
    int op1 = v1[threadIdx.x];
    int op2 = v2[threadIdx.x];
    // Operacion
    int res = op1 + op2;
    // Sincronización:
    __syncthreads();
    // Escritura de resultados:
    d_vr[threadIdx.x] = res;
}
```

### FUNCIÓN AUXILIAR

Se ejecuta en device. Sirve como una función CPU, para agrupar código y jerarquizar este.

### JERARQUÍA DE MEMORIA

Existe una jerarquía de memoria asociada a la GPU.

Memoria	Unida	Acceso	Descripción	Latencia
Local	Hilo	Privada	Lenta por ser off-chip. Es una sección privada de la memoria global.	Alta
Registros	Hilo	Privada	Rápidos, para operaciones.	Muy baja
Compartida	Bloque	Hilos del bloque	Rápida y compartida por los hilos del bloque.	Baja
Global	Host y Device	Hilos y CPU	Lenta, usada para pasar datos entre CPU y GPU.	Muy alta
Constante	Host y Device	Hilos y CPU en lectura y declaración.	Más rápida que la global, usada para datos de sólo lectura. Cacheada a nivel de bloque.	Media
Texturas	Host y Device	Hilos y CPU en lectura y declaración.	Similar a la constante.	Media

## ARQUITECTURA DESDE EL PUNTO DE VISTA DEL HARDWARE

### TERMINOLOGÍA

Abrev.	Elemento	Asociada a	Memoria	Contiene
<b>SP</b>	Streaming processor	Hilos	Local y registros	ALU
<b>SM</b>	Streaming multiprocessor	Bloques	Compartida	SP's
<b>TPC</b>	Texture processor cluster	Jerarquía SM	Texturas	SM's
<b>SPA</b>	SP array	Jerarquía GPU	Memoria global	TPC's

### PROCESO DE EJECUCIÓN

1. Se invoca el kernel, donde se especifican el número de hilos por bloque y el número de bloques que conforman la malla.
2. Una vez en la GPU, los hilos reciben un único identificador dentro de su bloque.
3. Los bloques reciben un único identificador dentro de su malla.
4. La unidad de distribución distribuye los bloques en los SM's disponibles de la GPU.
5. Los SM's mapean cada hilo del bloque sobre un núcleo SP, y cada hilo recibe su contador de programa y registros de estado y registros asignados independientemente.
6. Los SM's dividen el bloque en unidades de 32 hilos llamados WARPS.
7. Se selecciona un WARP que esté listo para ejecutarse, y sus instrucciones se ejecutan una a una concurrentemente y atómicamente en los SP's. Si se llega a una instrucción que no puede ser aún ejecutada (falta de datos en memoria local), se saca, se marca como en espera y busca otro WARP marcado como listo para ejecutarlo.

### PLANIFICADOR DE WARPS

Los bloques se asignan a SM's. Una vez llega un bloque a un SM, se divide en WARPS. Los WARPS son agrupaciones de 32 hilos, cuyas instrucciones (NO LOS HILOS) se ejecutan concurrentemente y atómicamente en los SP's del SM.

El SM contiene un Planificador de Warps sin sobrecarga.

Los WARP's cuya siguiente instrucción y operandos estén listos son marcados como listos, y los que no, como espera.

Los WARP's listos son encolados en una lista de prioridad, de acuerdo a ciertas políticas (Round-robin, edad, scoreboard ...).

### DIRECCIONAMIENTO DE BANCOS

Solo 1 hilo debería poder acceder a 1 banco de memoria, de forma que evitamos las condiciones de carrera.

#B = nº de bancos	S = entero coprimo con #B	T_id = Id del Hilo
-------------------	---------------------------	--------------------

Banco asignado al hilo =  $[(S * T\_id) \% \#B]$

## EJEMPLO: ARQUITECTURA DE LA G80

La G80 contiene:

Características:	Limitaciones:
• 8 TPC	• 8 Bloques / SM
• 2 SM / TPC = 16 SM's en total	• 512 Hilos / Bloque
• 8 SP / SM = 128 SP's en total	• 768 Hilos / SM
• 8192 Registros / SM.	• 16 KB de memoria compartida / SM
• 16 Bancos de registro / SM	• 8KB de memoria constante y de texturas / TPC

## EJERCICIOS RÁPIDOS CUDEIROS

### EJERCICIOS DE HILOS Y REGISTROS

1. Introducimos 1 bloque de hilos de 16x16 hilos en un SM. Cada hilo consume 10 registros.

a) ¿Cuántos registros utiliza el bloque?

$$\text{num.registros} = \text{num.hilos} \times \left( \frac{\text{registros}}{\text{hilo}} \right) = (16 \times 16) \text{ hilos} \times (10) \text{ registros} = 2560$$

b) ¿Cuántos bloques de este tipo cabrían en un SM?

i. Límite por hilos:

$$\text{num.bloques} = \frac{(\text{max.hilos/SM})}{(\text{hilos/bloque})} = \frac{768}{256} = 3$$

ii.

iii. Límite por registros:

$$\text{num.registros} = \frac{(\text{max.registros/SM})}{(\text{registros/bloque})} = \frac{8192}{2560} = 3$$

Vemos que en ambos casos son 3, por lo tanto, caben 3 bloques. Si no fuesen iguales, habría que quedarse con el menor.

c) Repetir si cada hilo consume 11 registros.

$$\text{num.registros} = \text{num.hilos} \times \left( \frac{\text{registros}}{\text{hilo}} \right) = (16 \times 16) \text{ hilos} \times (11) \text{ registros} = 2816$$

i. Límite por hilos:

$$\text{programas por limite de hilos} = \frac{(\text{max.hilos/SM})}{(\text{hilos/bloque})} = \frac{768}{256} = 3$$

ii. Límite por registros:

$$\text{programas por limite de registros} = \frac{(\text{max.registros/SM})}{(\text{registros/bloque})} = \frac{8192}{2816} = 2.9$$

Vemos que ya no caben 3 programas debido a que nos quedamos cortos en nº de registros.

2. ¿Cuáles de estos bloques de hilos caben en un SM de la G80?

Bloque	¿cabe?	¿Cuántas veces?
8x8 hilos = 64 hilos	Si: 64 < 512 hilos/bloque	Como sólo caben 8 bloques por SM, cabría 8 veces como máximo.
16x16 hilos = 256 hilos	Si: 256 < 512 hilos/bloque	Cabría el mismo programa (768 hilos máximos / SM ) / (256 hilos / bloque) = 3 veces.
4 x 4 hilos = 16 hilos	Si: 16 hilos < 512 hilos/bloque	Cabría 8 veces por el mismo motivo que el de 8x8.
32 x 32 hilos = 1024 hilos	No: 1024 hilos > 512 hilos/bloque	Ninguna.

3. Supongamos que ejecutamos 3 bloques de 16 x 16 hilos en un SM. Cada bloque consume 5 KB de memoria compartida.

a) ¿Entraría en un SM? ¿Correría correctamente?

Para ver si cabría, comprobamos las limitaciones por hilos y memoria compartida.

i. Límite por hilos:

$$\text{programas por limite de hilos} = \frac{(\text{max. hilos/SM})}{(\text{hilos/bloque})} = \frac{768}{256} = 3$$

ii. Límite por memoria compartida:

$$\text{programas por limite de memoria} = \frac{(\text{max. memoria compartida/SM})}{(\text{memoria compartida / bloque})} = \frac{16\text{KB}}{5\text{KB}} = 3 \text{ bloques}$$

b) ¿Cuántos bloques de 16 x 16 hilos cabrían en un SM si cada bloque consume 8 KB de memoria compartida?

i. Límite por hilos:

$$\text{programas por limite de hilos} = \frac{(\text{max. hilos/SM})}{(\text{hilos/bloque})} = \frac{768}{256} = 3 \text{ bloques}$$

ii. Límite por memoria compartida:

$$\text{programas por limite de memoria} = \frac{(\text{max. memoria compartida/SM})}{(\text{memoria compartida / bloque})} = \frac{16\text{KB}}{8\text{KB}} = 2 \text{ bloques}$$

Vemos que estamos limitados por el límite de memoria compartida por bloque, por tanto sólo podremos lanzar 2 bloques que consuman 8 KB de memoria compartida cada uno.

# APA – TEMA 1 – UNIDAD 4

## CONCEPTOS CLAVE

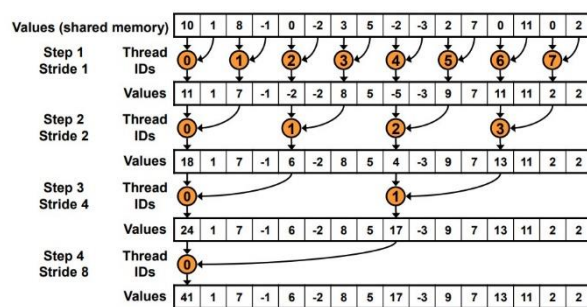
### CONCEPTOS CLAVE Y OPTIMIZACIÓN DE PARALELISMO

Revisaremos aquí algunos conceptos clave en relación a la programación paralela.

### ALGORITMOS DE REDUCCIÓN

Se tienen 2 principales algoritmos de reducción paralelizables:

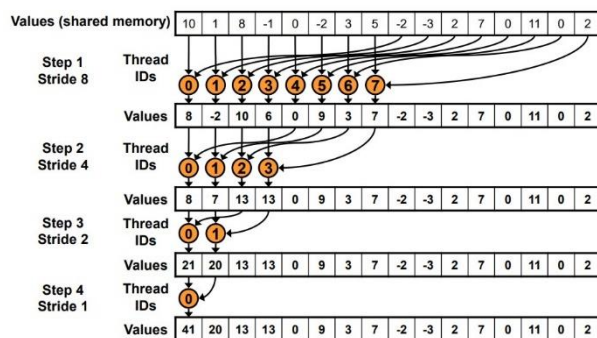
#### REDUCCIÓN PARALELA INTERCALADA



Consiste en operar cada elemento con el elemento de al lado, de forma que se va reduciendo en 2 por cada paso.

Tiene como desventajas que no es un acceso coalescente, por tanto, será menos eficiente que si lo fuese, y que genera conflictos en el acceso a bancos.

#### REDUCCIÓN PARALELA SECUENCIAL



Consiste en que en cada paso se divide el vector de elementos en 2 y operar como si de dos vectores se tratase.

No tiene problemas de bancos y su acceso es coalescente.

### ILP VS TLP

El ILP (Instruction Level Parallelism) es la capacidad de un procesador de ejecutar más de una instrucción de un mismo flujo de instrucciones simultáneamente. Esto se consigue teniendo dos o más ALU's y 2 o más UC's. Que un procesador soporte el ILP no implica que todas las instrucciones se ejecuten en paralelo.

El TLP (Thread Level Parallelism) es capacidad de un procesador de ejecutar distintos flujos de instrucciones de la misma aplicación simultáneamente.

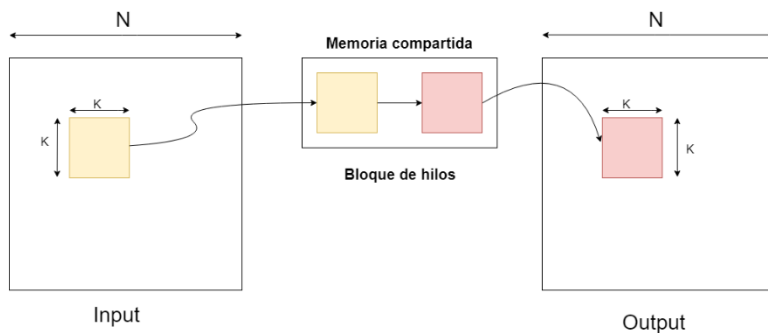
## ALGORITMOS TESELADOS

Consiste en dividir un problema de tamaño  $N$  en  $m$  subproblemas de tamaño  $(N/k)$ .

Estos subproblemas se pasan a bloques de hilos que emplean la memoria compartida para hacer operaciones.

Por ejemplo, en la transposición de matrices, se divide la matriz en submatrices. Cada bloque de hilos transpone una de estas submatrices, utilizando la memoria compartida.

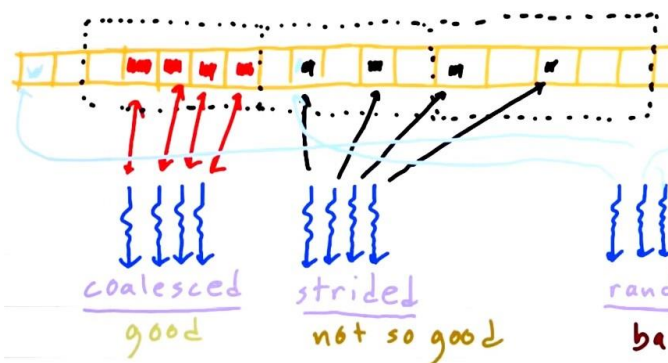
Tiene como ventaja que la lectura y escritura es coalescente.



## DESENROLLAMIENTO

La técnica de desenrollamiento o unwrapping consiste en eliminar los bucles y sustituirlo por el código secuencial. Esto salva ciclos innecesarios, asignaciones de variables temporales e instrucciones extra sin trabajo real.

## COALESCENCIA



Es el acto de la fusión de dos bloques libres adyacentes en memoria.

Al acceder a memoria global, el rendimiento máximo se alcanza cuando todos los hilos de medio warp acceden a direcciones contiguas.

Esto es debido a que los bloques de memoria se acceden de 16 en 16, es decir, que si todos los elementos están juntos, sólo se tiene que realizar un acceso a memoria, mientras que si los elementos están dispersos, se tendrán que hacer sucesivos accesos a memoria para traer todos los datos necesarios para la ejecución del medio warp.

## CONCEPTOS CLAVE DE CUDA

Estos conceptos están relacionados, además de con la paralelización, con la arquitectura CUDA.

### SPMD

Las GPU's de Nvidia no encajan en la taxonomía de Flynn. Se pueden interpretar como un dispositivo Single Program Multiple Data Stream, es decir, un solo programa que ejecuta múltiples hilos que acceden a múltiples flujos de datos.

### COMPILADOR DE PROGRAMAS CUDA

Cuda utiliza el driver de compilación NVCC. Este driver trabaja invocando todas las herramientas y compiladores necesarios: cudacc, g++, cl, mingw, gcc, etc.

NVCC genera, respecto al host:

- Código C para el host, que debe ser compilado por otra herramienta dentro del NVCC.

NVCC genera, respecto al device, una de las siguientes dos opciones:

- Código en PTX para el device. Este código es código objeto que debe ser compilado para una GPU concreta.
- Código en fuente PTX para el device, que es interpretado en tiempo de ejecución.

El NVCC sigue estos pasos:

1. Se utiliza CUDAFE, un preprocesador de código que divida la aplicación en la GPU (archivo.s) y CPU (archivo.cpp).
2. El código de CPU es compilado por un compilador gcc, mingw, etc.
3. El código de GPU es convertido por NVOPENCC generando un código ensamblador PTX.
4. El PTX se pasa al OGC, que asigna registros y la lista de instrucciones de acuerdo a la GPU que se utilice y optimice. De esta forma se genera un código objeto para la GPU, que más tarde da lugar al ejecutable.

Para realizar el enlazado de PTX, se necesitan las librerías dinámicas de cuda runtime library y cuda core library.

Estas API's son una extensión del lenguaje C, que contiene funciones para ejecutar código en el device, funciones matemáticas optimizadas para cuda, operaciones enteras en la GPU, funciones de manejo de memoria y sincronización,

El NVCC dispone de un modo de emulación disponible hasta la versión 3.0. Esta emulación no requiere un dispositivo CUDA, se ejecuta en el host. Los hilos, sin embargo, se ejecutan secuencialmente. La referencia a memoria de host puede causar problemas. Los resultados en coma flotante pueden ser diferentes.



## COMPUTE CAPABILITIES

Nvidia utiliza este formato estandarizado para especificar las características de cada tarjeta gráfica.

Esta categorización incluye dos números:

- El primero indica cambios de generación.
- El segundo indica revisiones en una generación.