Procesadores gráficos

Programación utilizando CUDA

Jose A. Pascual Saiz

Despacho 214
joseantonio.pascual@ehu.eus
Ingeniería de Computadores
Facultad de Informática de San Sebastián (FISS)
Universidad del País Vasco (UPV-EHU)

Índice

Índice

Taxonomía de Flynn

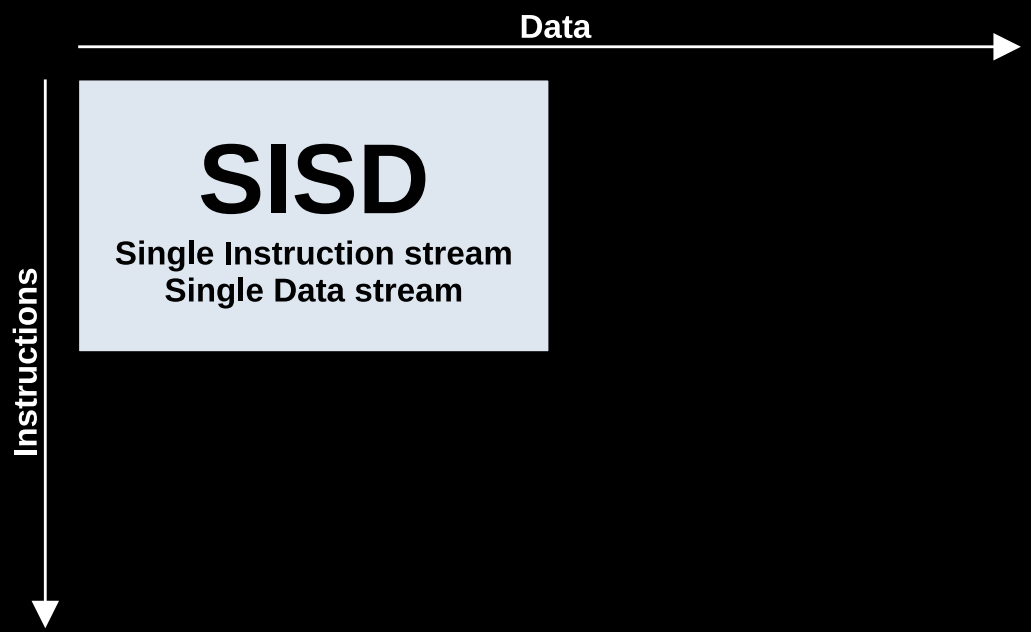
- Michael Flynn la propuso en 1966.
- Se utiliza el concepto de *Stream*.
  - Un *stream* es un flujo de elementos.
  - Pueden ser instrucciones o datos.
- Las arquitecturas de computadores se clasifican según el número de flujos de instrucciones y datos.

Taxonomía de Flynn

- Instrucciones y datos

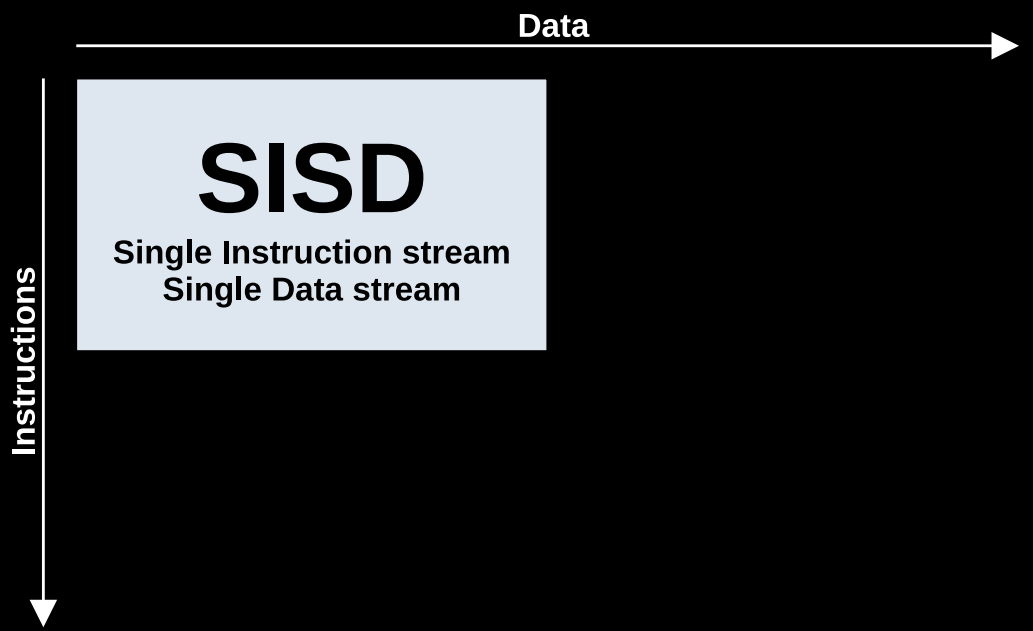
Taxonomía de Flynn (SISD)

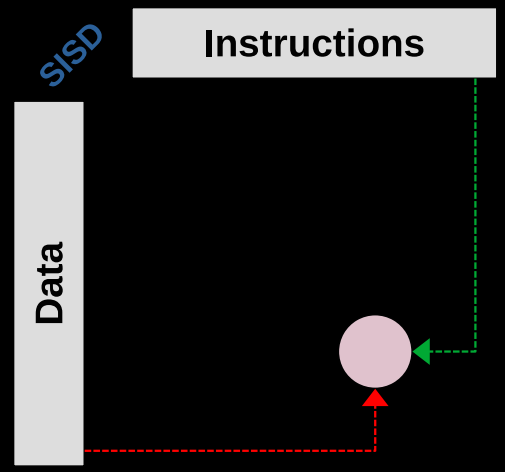
- Flujo de instrucciones único y flujo de datos único



Taxonomía de Flynn (SISD)

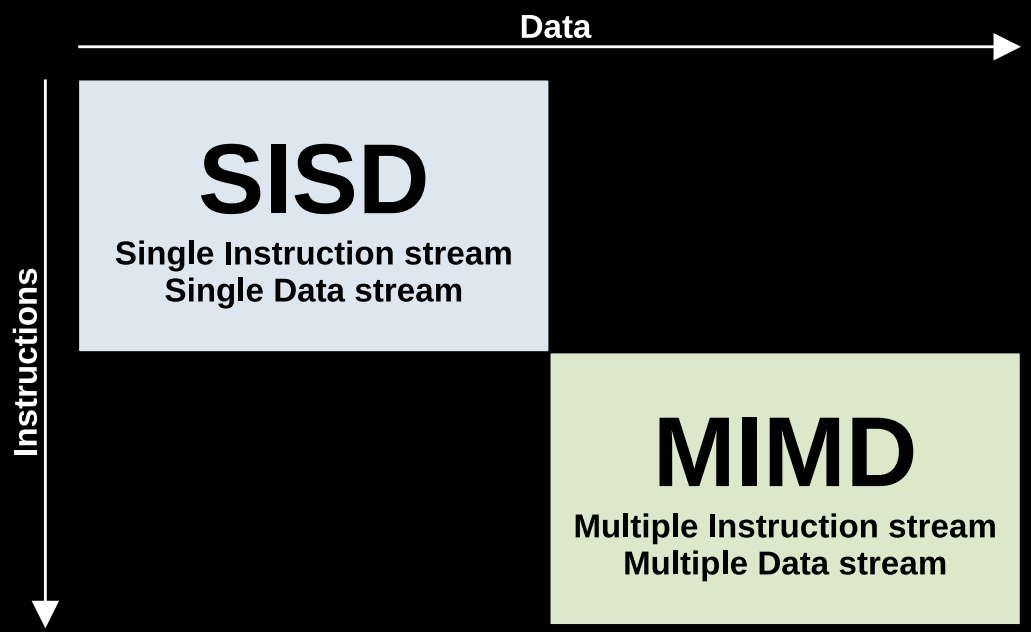
- Flujo de instrucciones único y flujo de datos único
  - Mono-procesadores (mono-núcleo) y *mainframes*





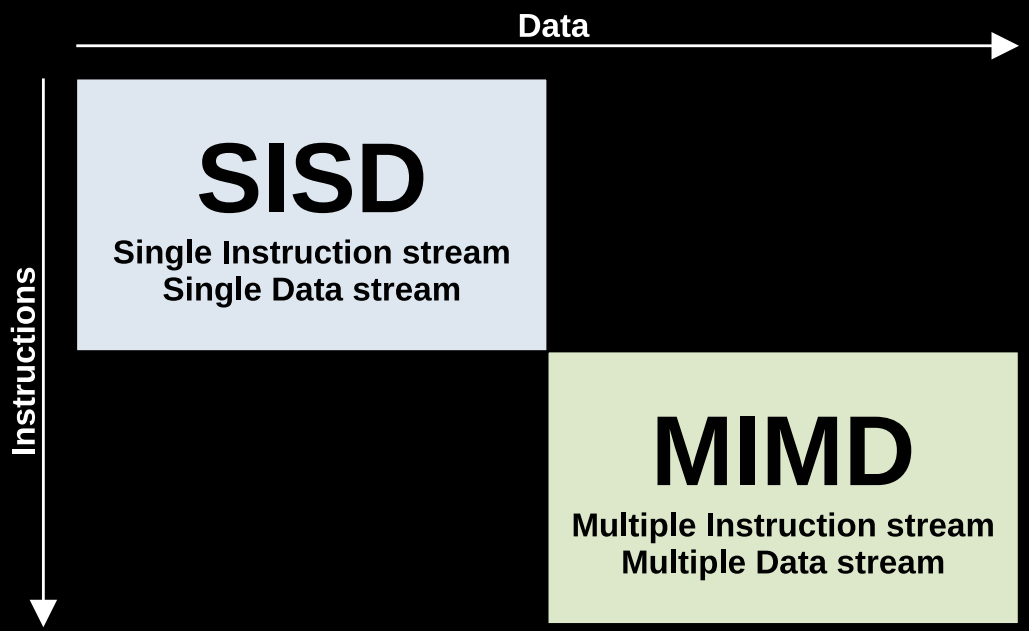
Taxonomía de Flynn (MIMD)

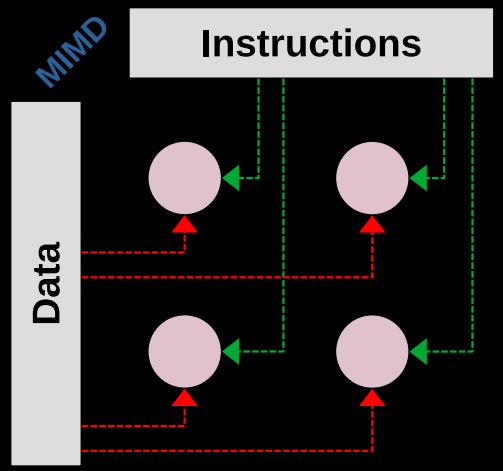
- Múltiple flujos de instrucciones y múltiples flujos de datos



Taxonomía de Flynn (MIMD)

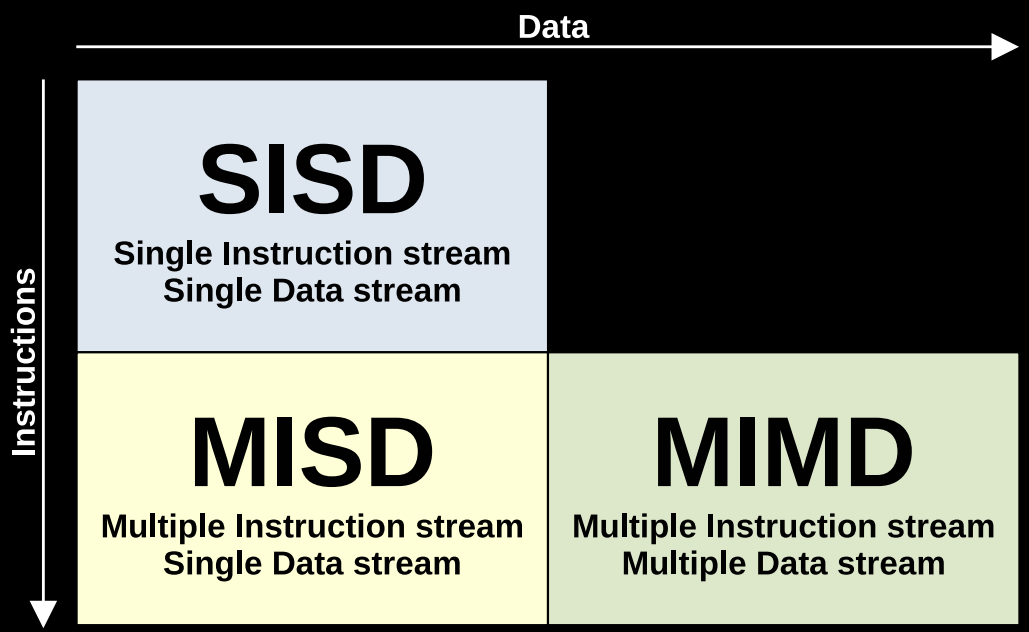
- Múltiple flujos de instrucciones y múltiples flujos de datos
  - Procesadores multi-núcleo y sistemas distribuidos





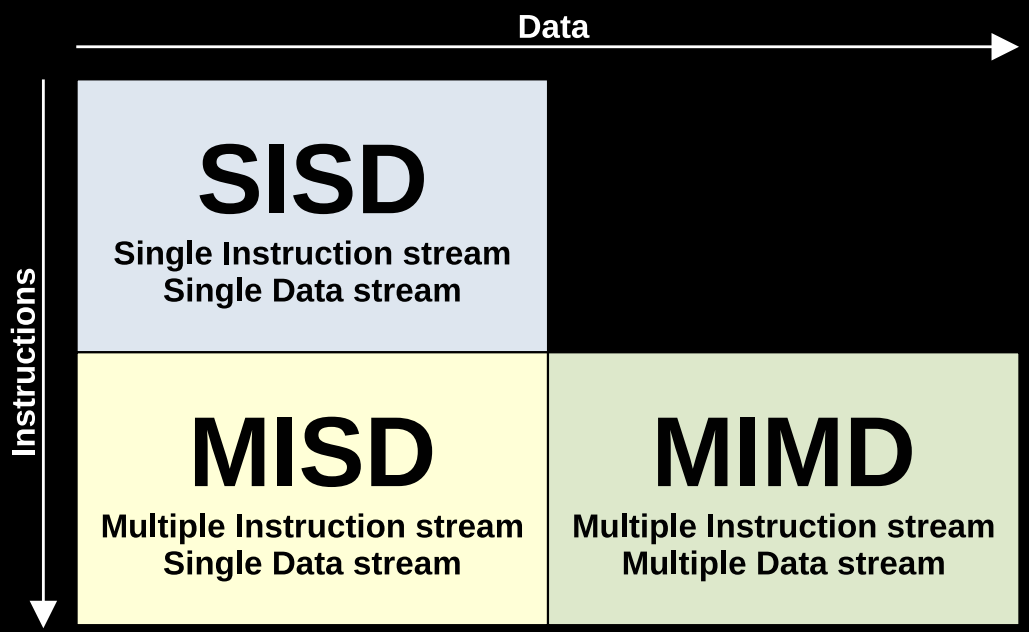
Taxonomía de Flynn (MISD)

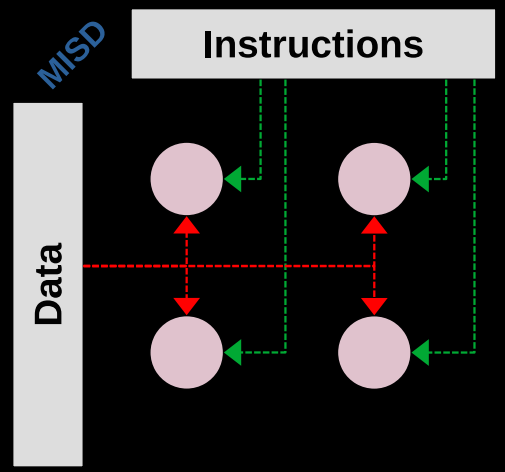
- Múltiples flujos de instrucciones y un único flujo de datos



Taxonomía de Flynn (MISD)

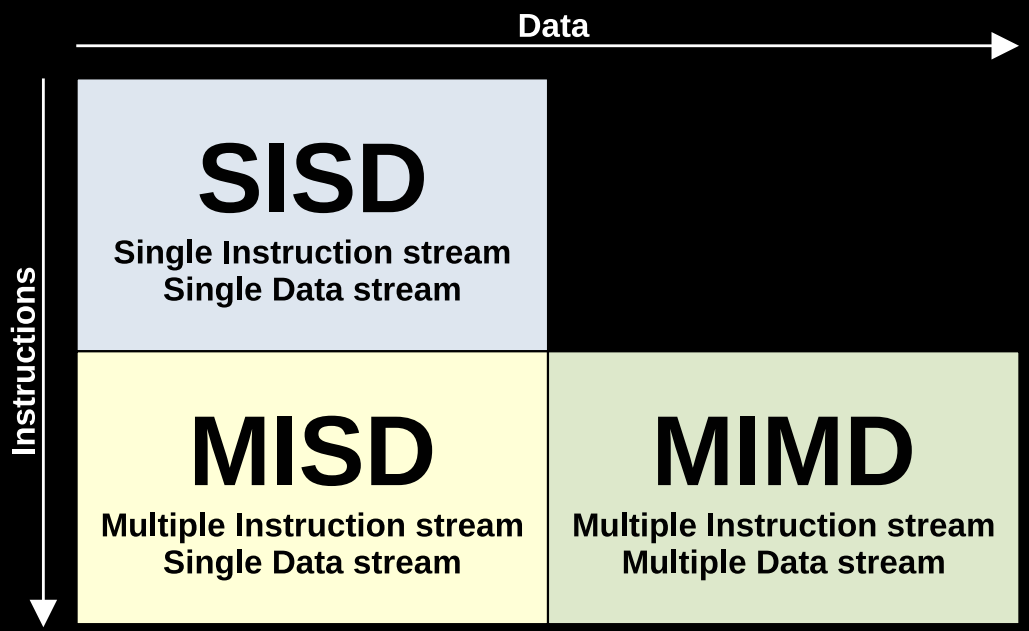
- Múltiples flujos de instrucciones y un único flujo de datos
  - Ejemplos?

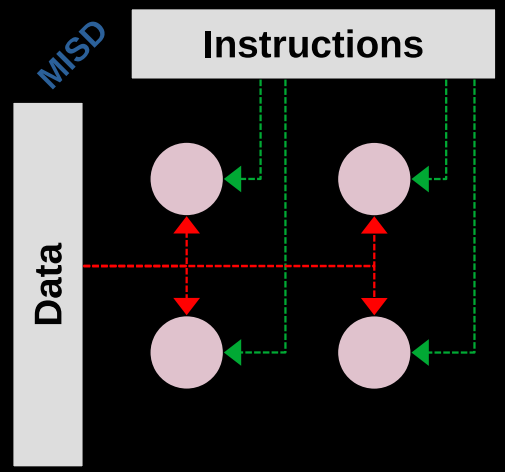




Taxonomía de Flynn (MISD)

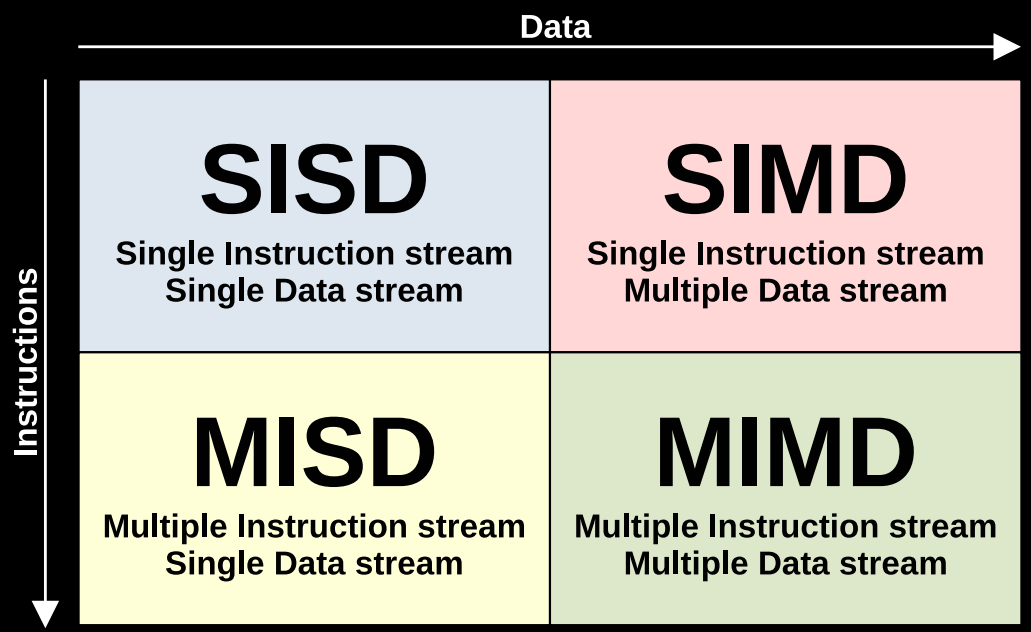
- Múltiples flujos de instrucciones y un único flujo de datos
  - *Space shuttle*, satelites, *SpaceX*...





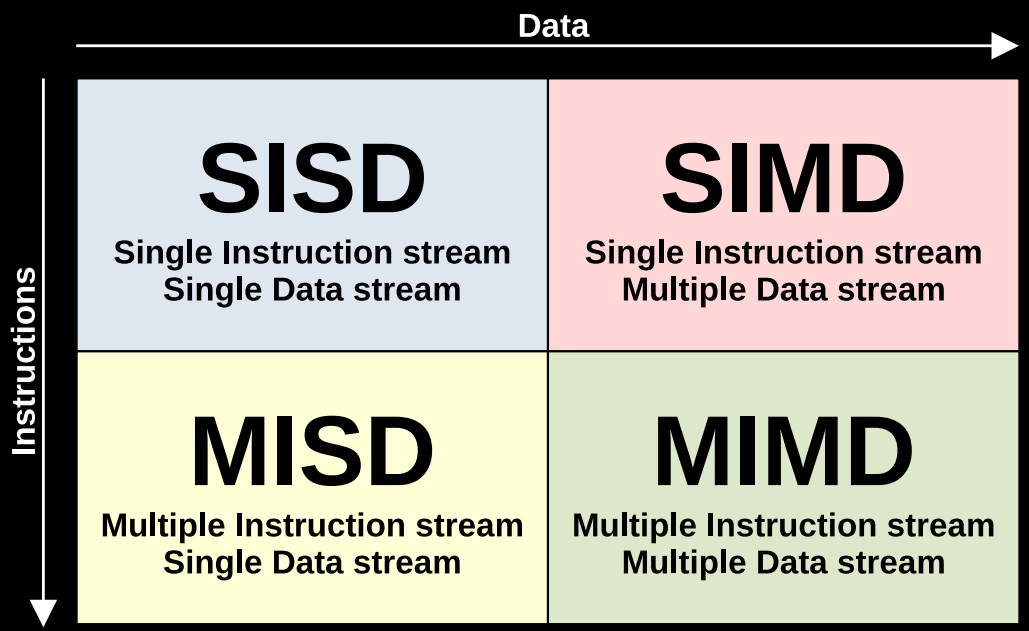
Taxonomía de Flynn (SIMD)

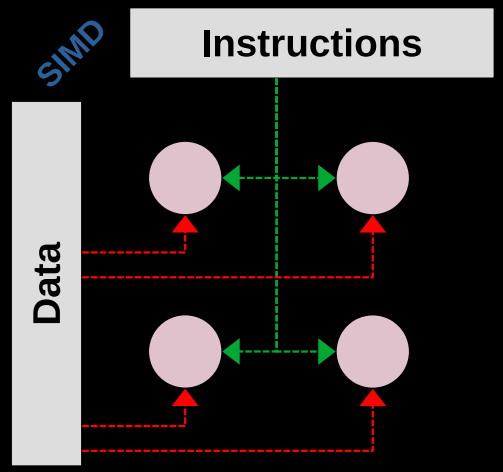
- Un único flujo de instrucciones y múltiples flujos de datos



Taxonomía de Flynn (SIMD)

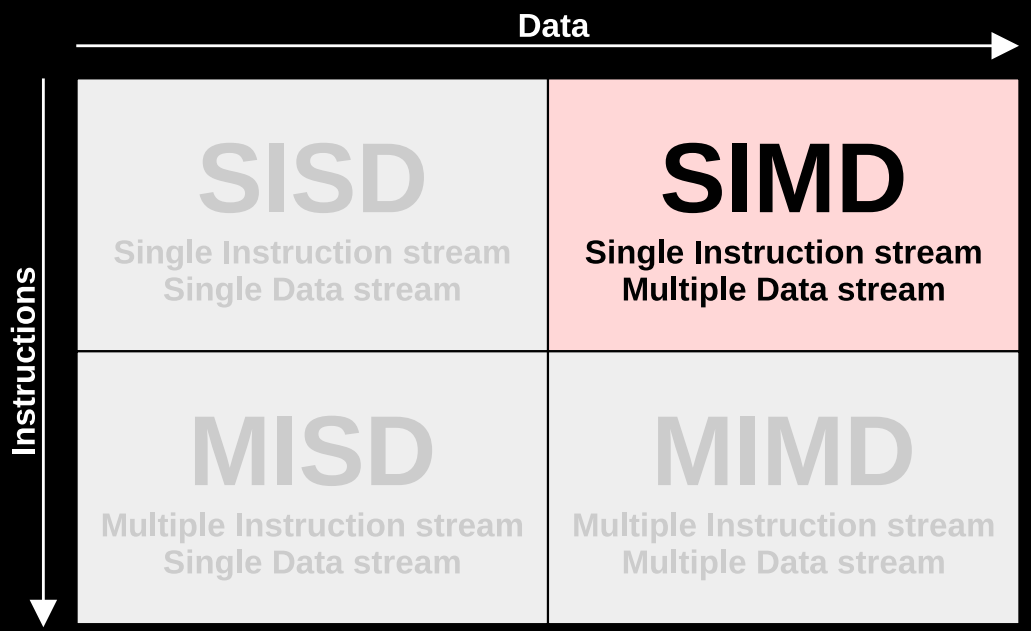
- Un único flujo de instrucciones y múltiples flujos de datos
  - Procesadores de tipo *array, pipelined eta associative*

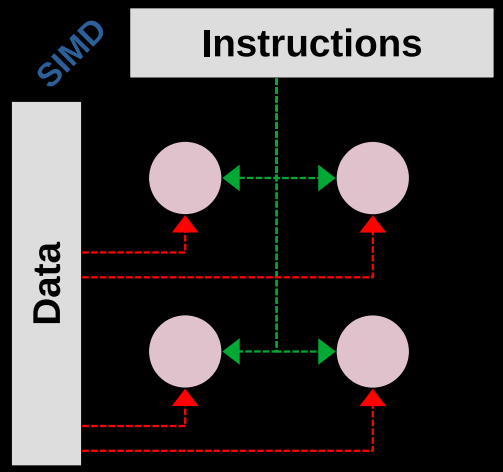




Arquitecturas de tipo SIMD

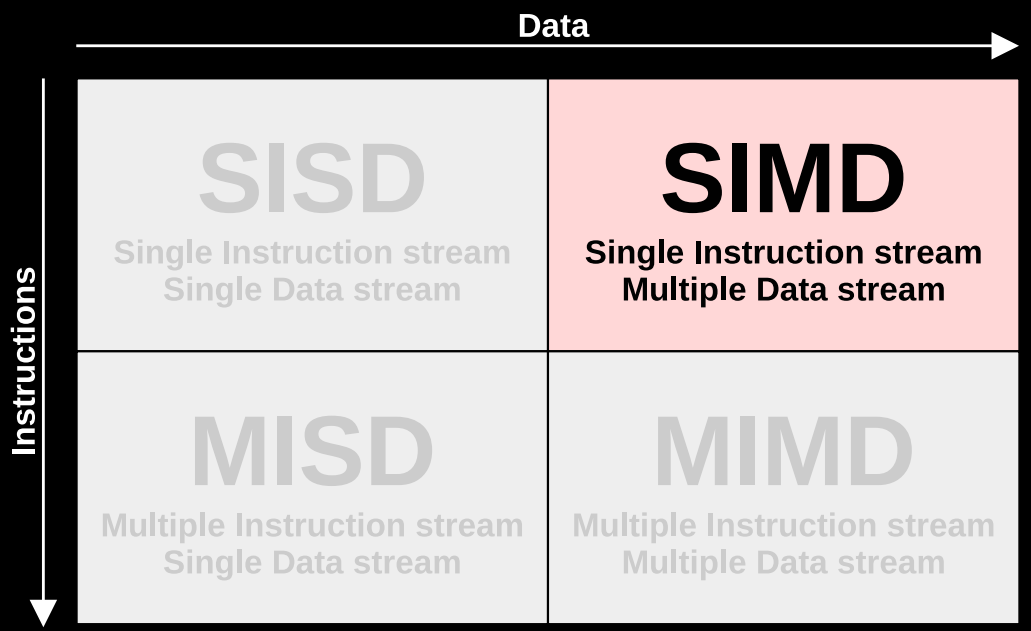
- Las unidades de proceso paralelas (UPP) reciben la misma instrucción pero . . .

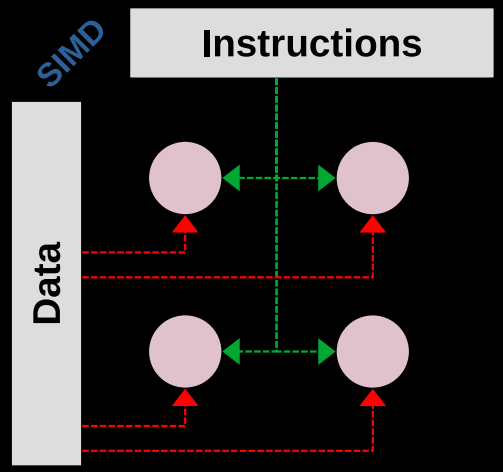




Arquitecturas de tipo SIMD

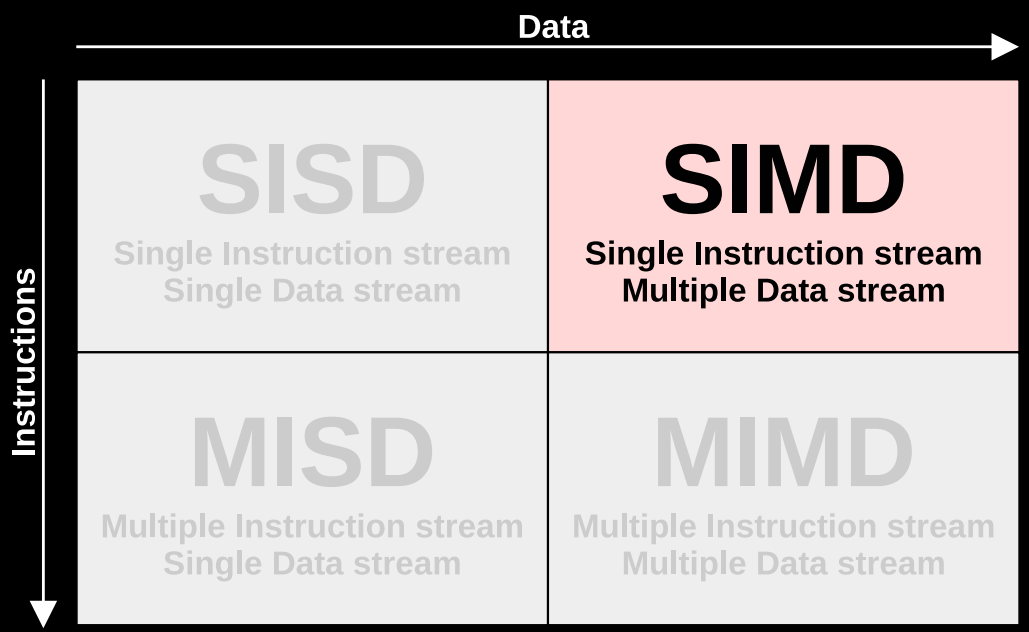
- Cada UPP memoria y registros propios (*array*)
- Leen de memoria y del banco de registros parte de los datos (*pipelined*)
- Toma la decisión de procesar o no los datos (*predicated*)

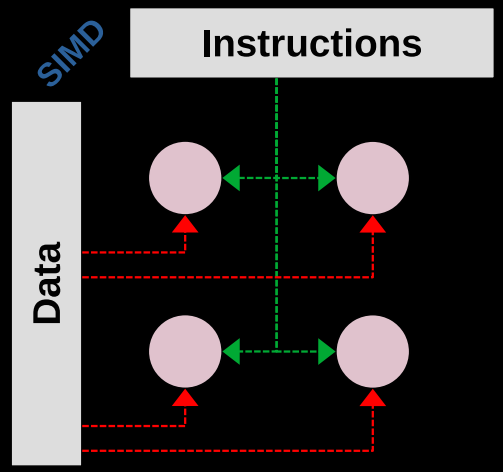




Arquitecturas de tipo SIMD

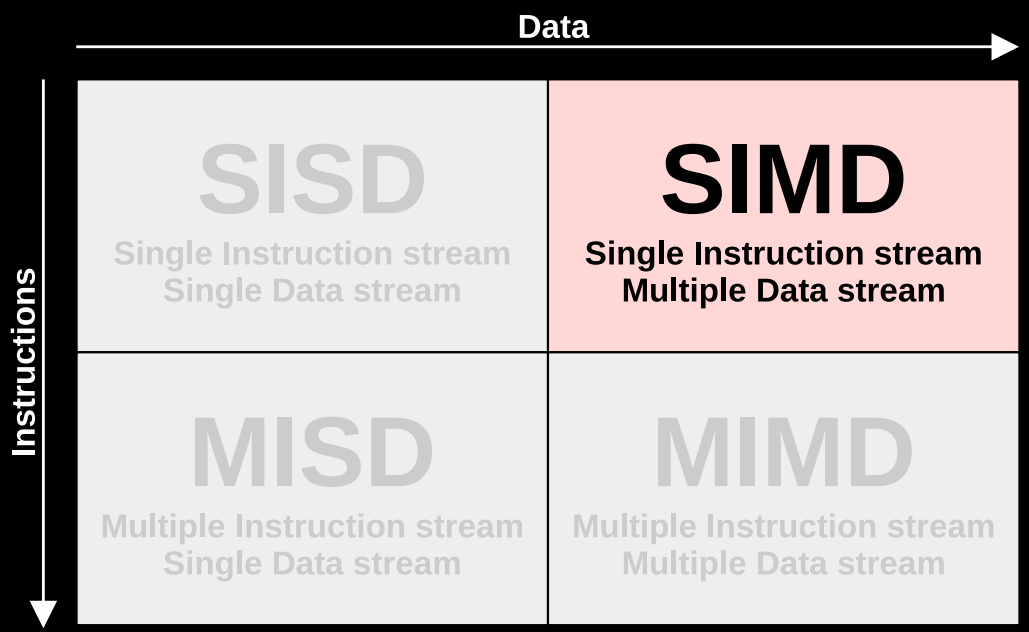
- Son las GPUs de tipo SIMD?

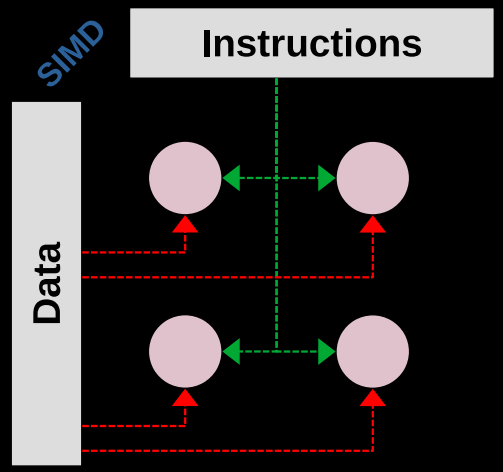




Arquitecturas de tipo SIMD

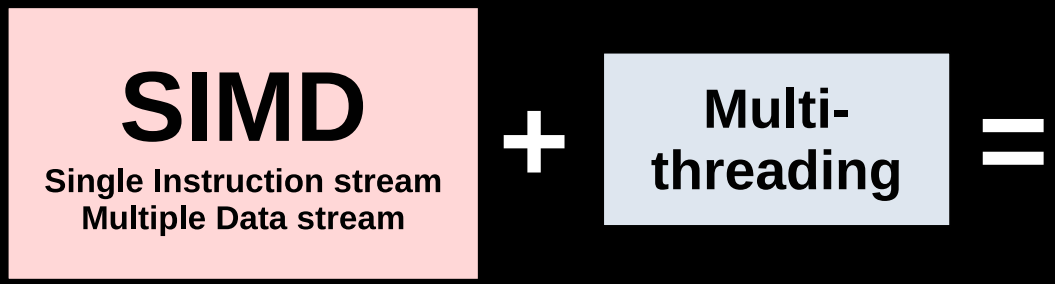
- Son las GPUs de tipo SIMD?
  - Si pero. . .





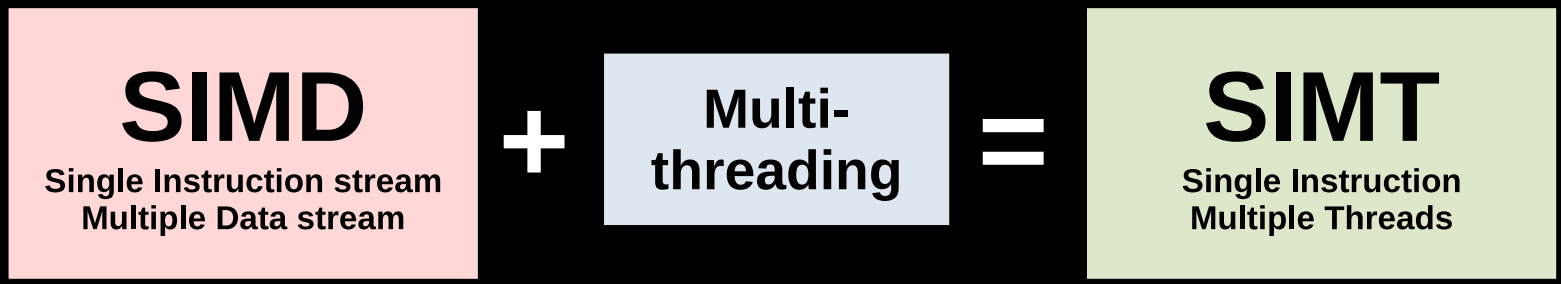
Arquitecturas de tipo SIMD

- A la arquitectura SIMD hay que añadirle...
  - Unidades de proceso multi-hilo



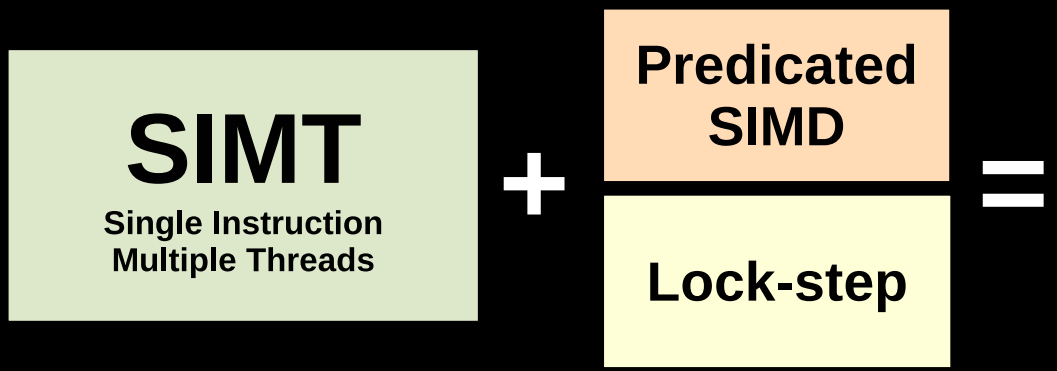
Arquitecturas de tipo SIMT

- A la arquitectura SIMD hay que añadirle...
  - Unidades de proceso multi-hilo



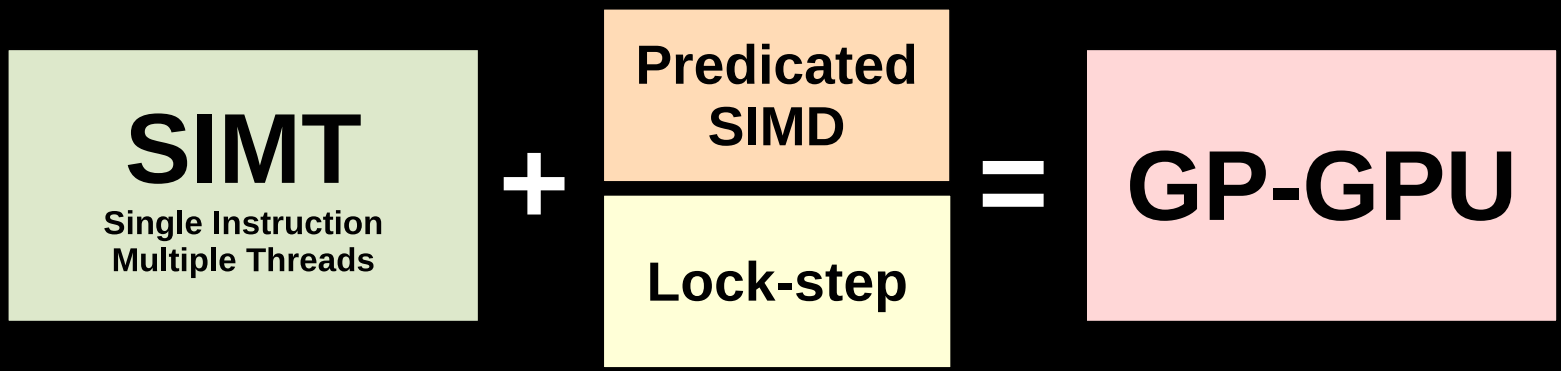
Arquitecturas de tipo SIMT

- A la arquitectura SIMT hay que añadirle...
  - Núcleos de tipo *predicated* SIMD
  - Modo de ejecución *lock-step*



General Purpose-Graphic Processing Units

- A la arquitectura SIMT hay que añadirle...
  - Núcleos de tipo *predicated* SIMD
  - Modo de ejecución *lock-step*



GP-GPU (Predicated SIMD)

- *Predicated* (*masked*) SIMD: Cada núcleo decide si ejecutar un código o no utilizando una máscara.
- Ejemplos: AVX2, AVX-512...

GP-GPU (Predicated SIMD)

- Tiene que ejecutar el siguiente código.

```
If condition
    code1;
else
    code2;
```

GP-GPU (Predicated SIMD)

- Tiene que ejecutar el siguiente código.
- Puede utilizar conditional branching.

```
If condition
    code1;
else
    code2;
```

```
If condition: label1
    code2;
goto label2
label1:
    code1;
label2:
...
```

GP-GPU (Predicated SIMD)

- Tiene que ejecutar el siguiente código.
- Puede utilizar conditional branching.
- Puede utilizar *predication* (*inline code*).

```
If condition
    code1;
else
    code2;
```

```
If condition: label1
    code2;
goto label2
label1:
    code1;
label2:
...
```

```
(condition):
    code1;
(not condition):
    code2;
```

GP-GPU (Ejecución Lock-step)

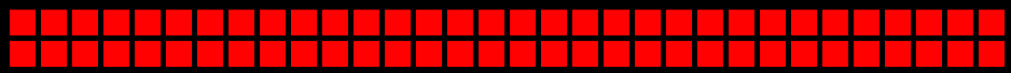
- En todos los núcleos se ejecuta el mismo código.
- Para ejecutar la siguiente instrucción, todas las anteriores (en todos los núcleos) tienen que haber acabado.
- Sino, se bloquean hasta que todos los núcleos hayan acabado.

GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la primera instrucción.

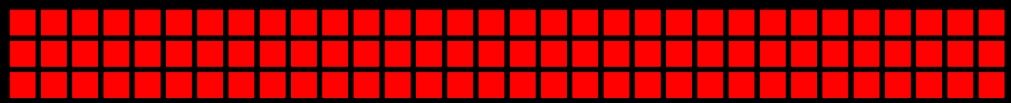
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la segunda instrucción.



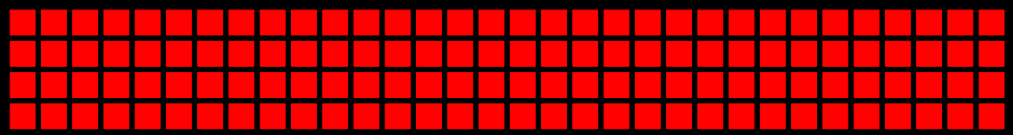
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la tercera instrucción.



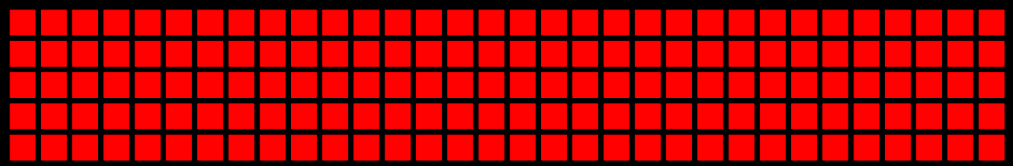
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la cuarta instrucción.



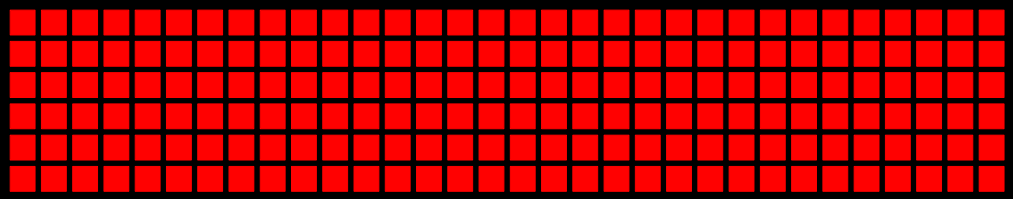
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la quinta instrucción.



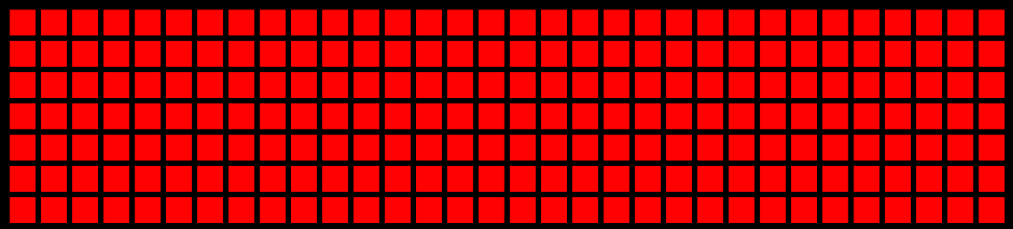
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la sexta instrucción.



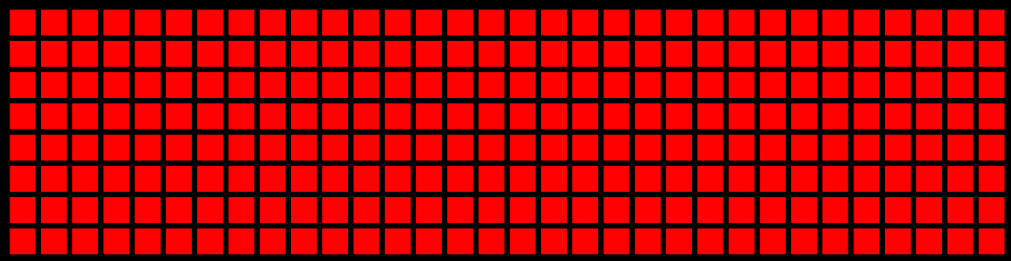
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la seṕtima instrucción.



GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la octava instrucción.

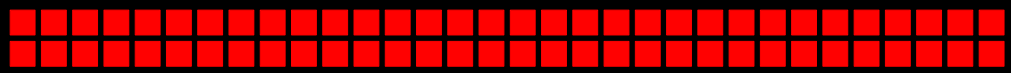


GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la primera instrucción.

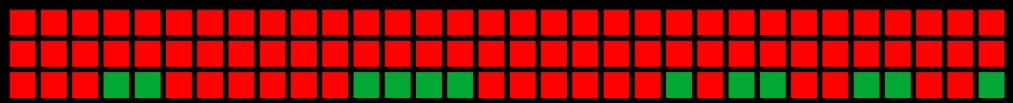
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la segunda instrucción.
  - La primera instrucción ha terminado.



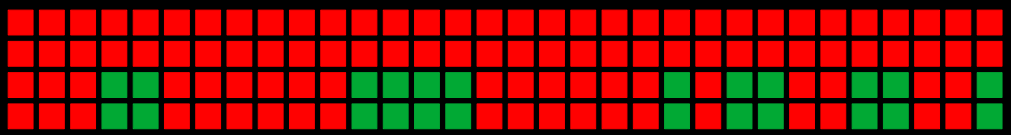
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la segunda instrucción.
  - La segunda instrucción no ha acabado en todos los núcleos.



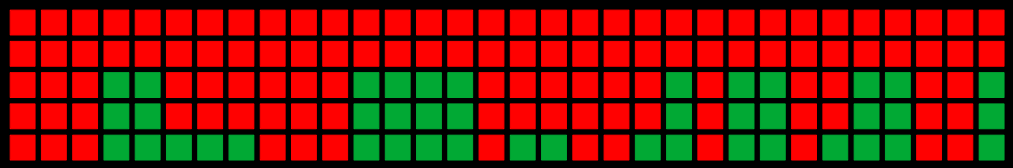
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la segunda instrucción.
  - La segunda instrucción no ha acabado en todos los núcleos.



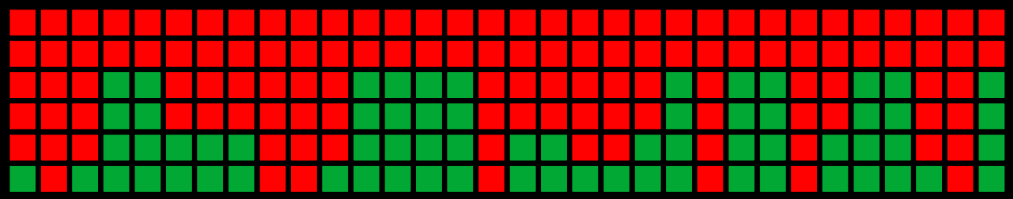
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la segunda instrucción.
  - La segunda instrucción no ha acabado en todos los núcleos.



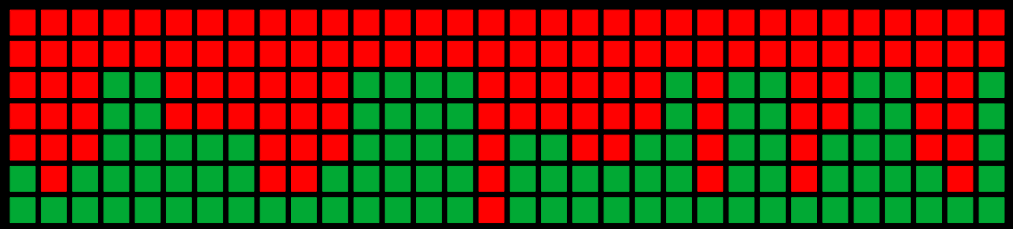
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la segunda instrucción.
  - La segunda instrucción no ha acabado en todos los núcleos.



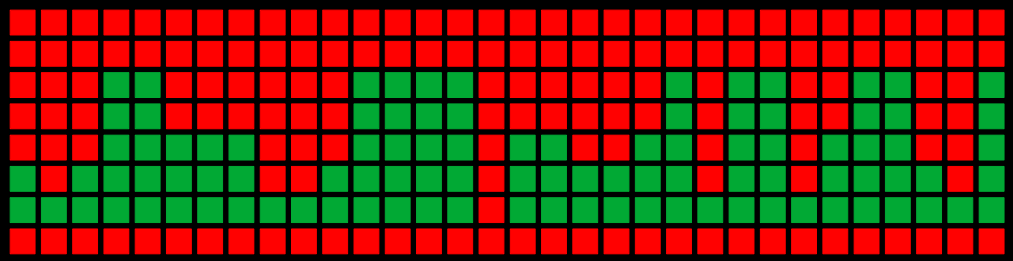
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la segunda instrucción.
  - La segunda instrucción no ha acabado en todos los núcleos.



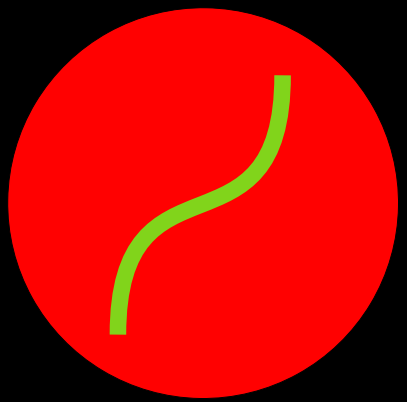
GP-GPU (Ejecución Lock-step)

- Todos los núcleos ejecutan la tercera instrucción.
  - La segunda instrucción ha acabado en todos los núcleos.



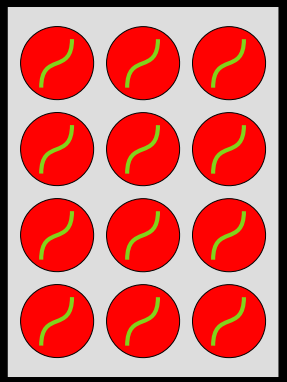
Componentes de una GP-GPU

- Un núcleo mono-hilo



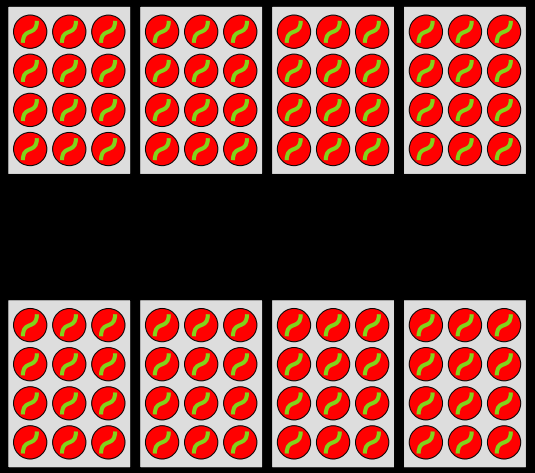
Componentes de una GP-GPU

- Streaming Multiprocessor (SM) multi-núcleo



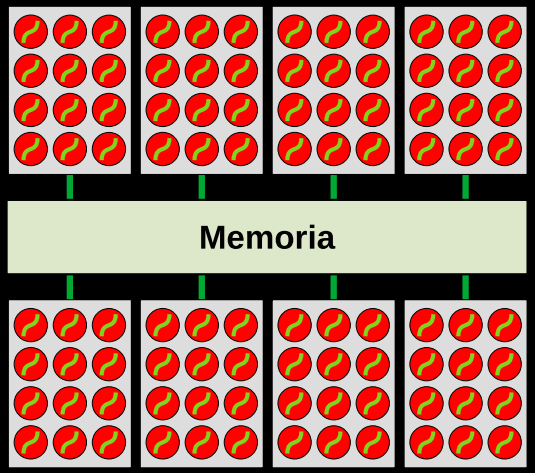
Componentes de una GP-GPU

- Múltiples Streaming Multiprocessor



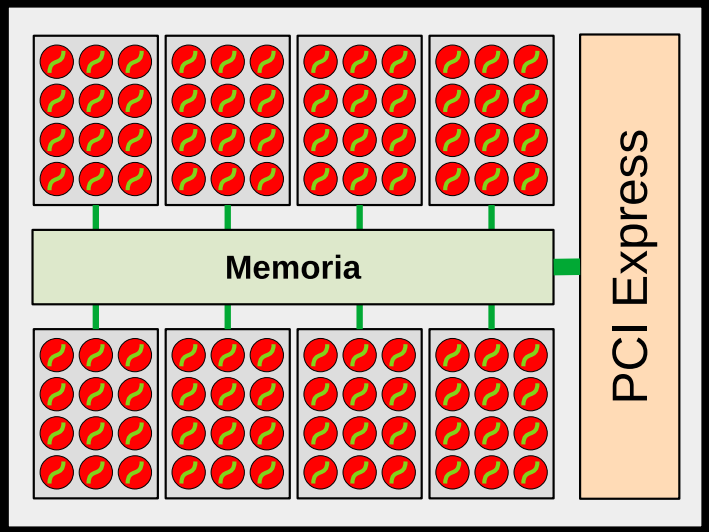
Componentes de una GP-GPU

- Memoria principal



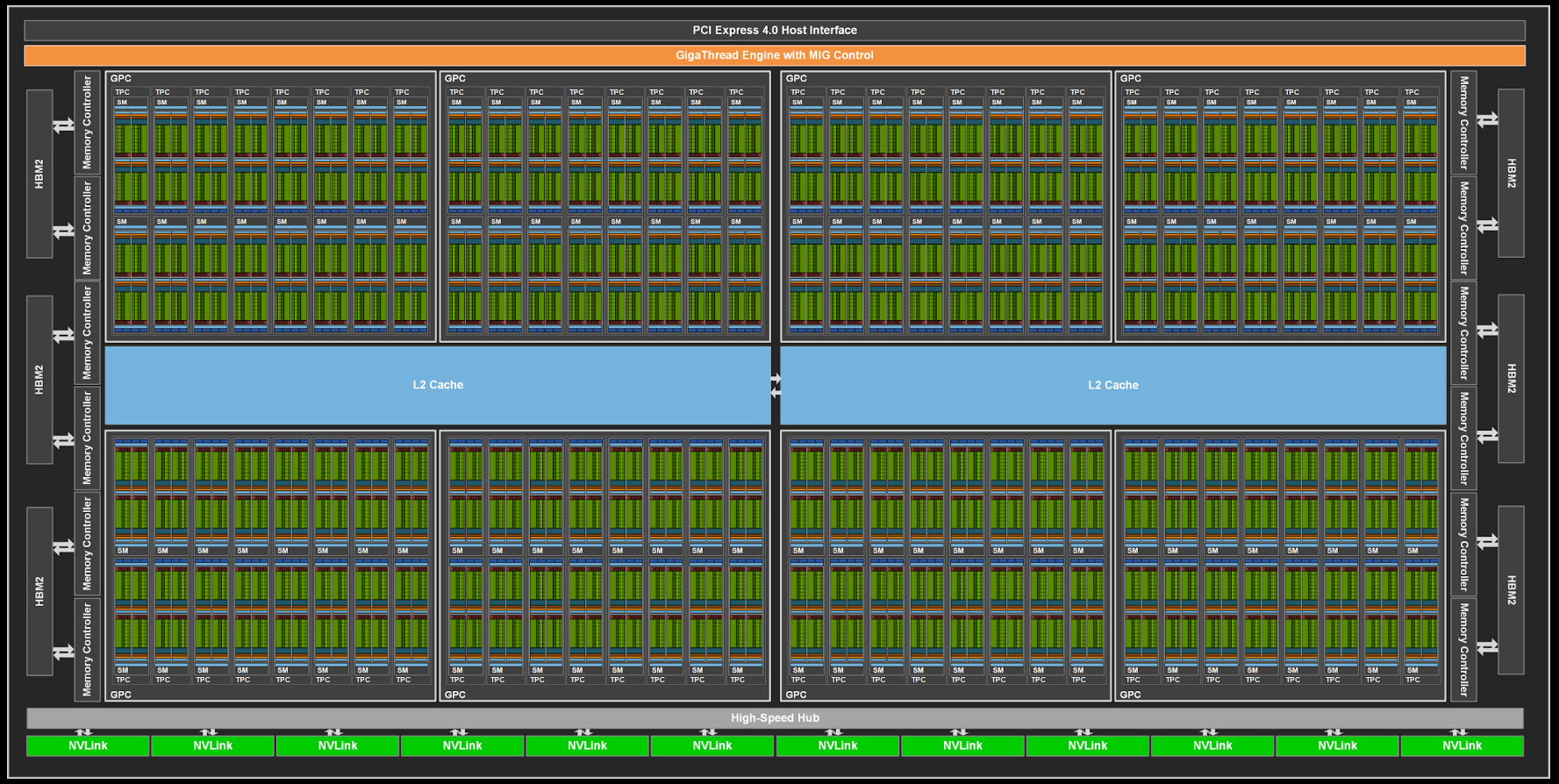
Componentes de una GP-GPU

- Interfaz PCI Express

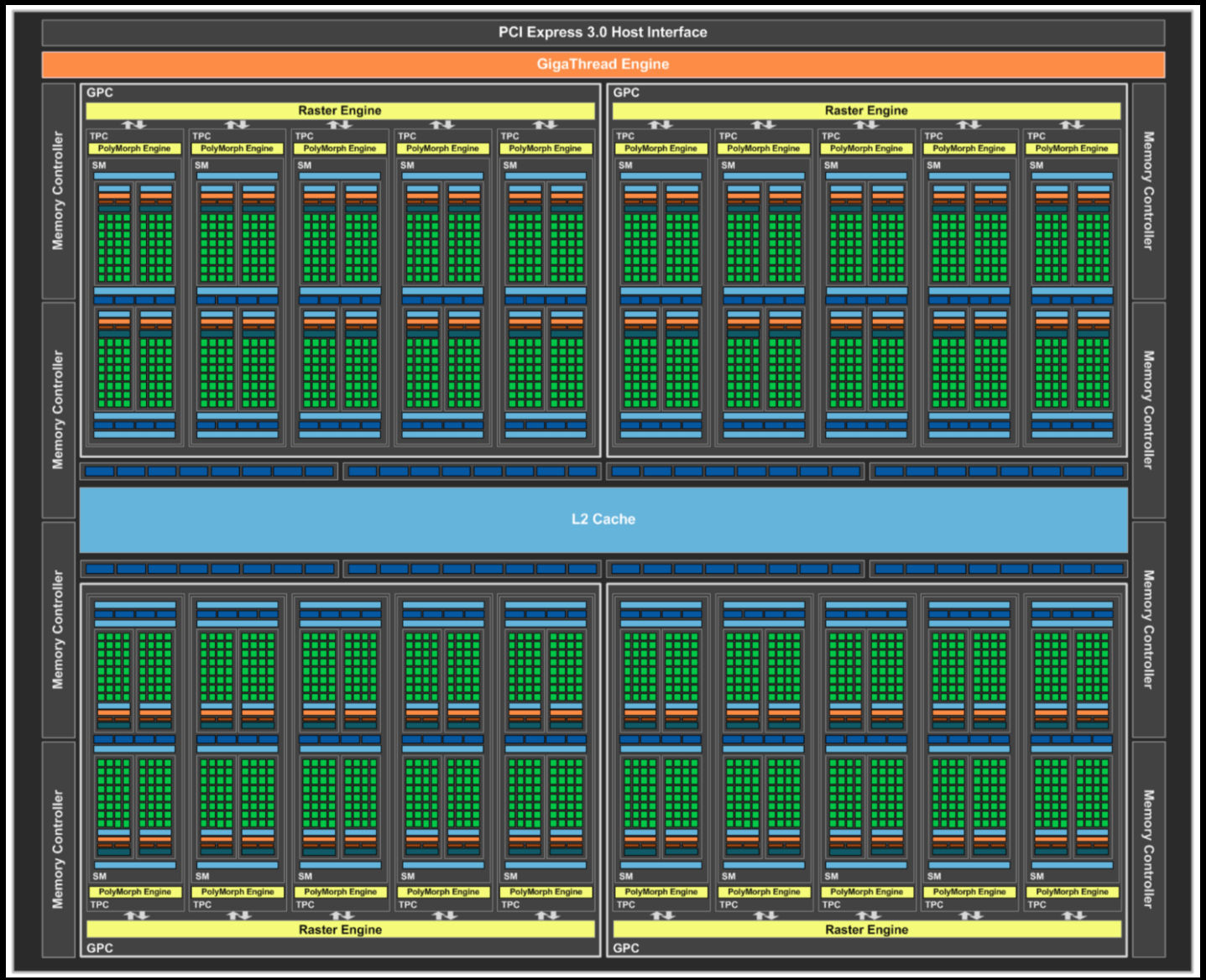


GPU NVidia H-100

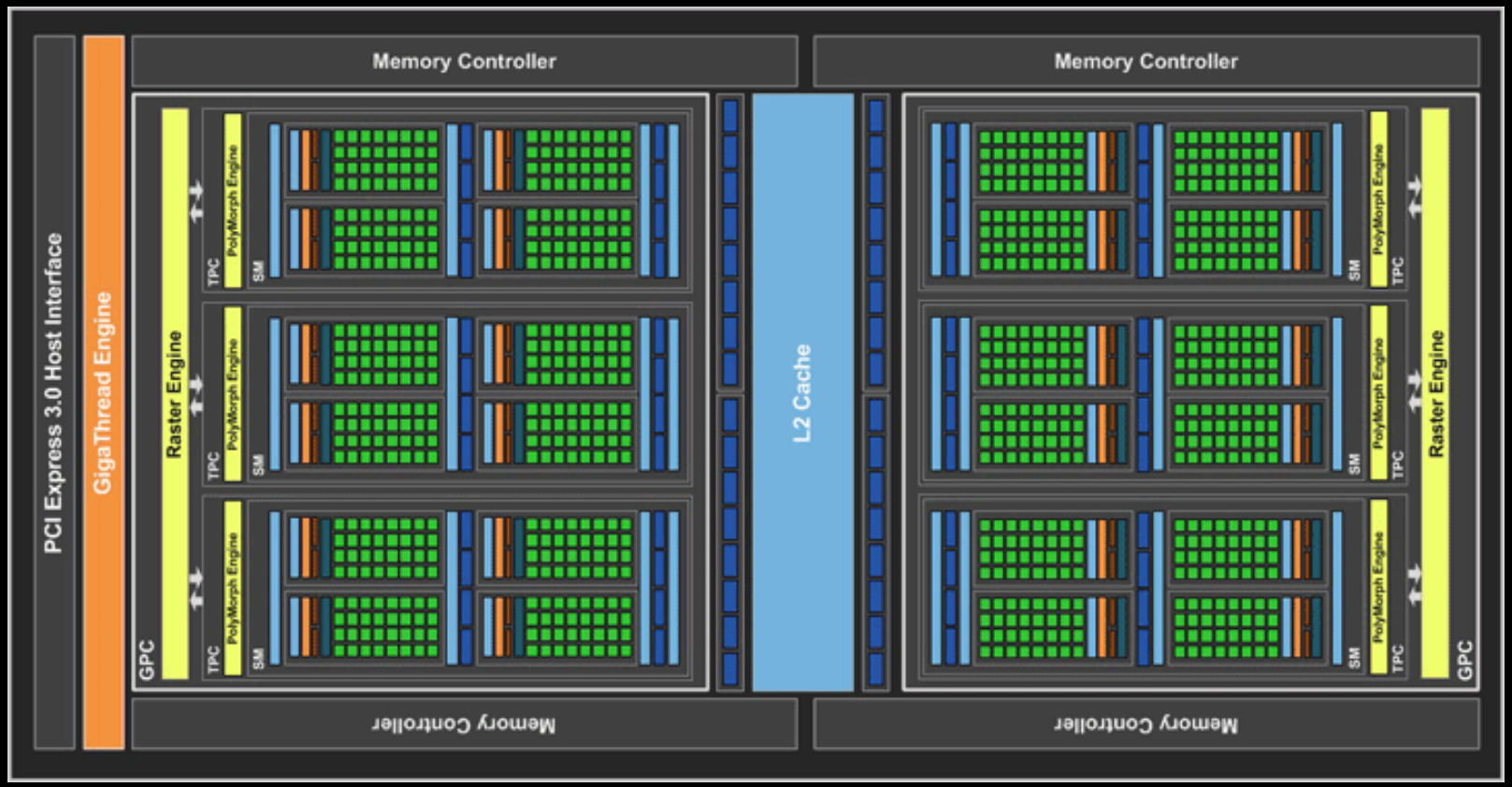
GPU NVidia A-100



GPU NVidia Tesla-P4000



GPU NVidia GTX-1050



Programación de GPUs NVidia

- Se programan utilizando la plataforma CUDA
  - La parte secuencial se ejecuta en la CPU y la paralela en la GPU.
  - Se pueden utilizar los lenguajes C, C++, Matlab, Python y Fortran.
  - El paralelismo se representa usando *keywords* muy sencillas.
  - El concepto más importante es el de *kernel*: un código pequeño que se ejecuta en todos los núcleos a la vez.

Primer código CUDA

- Consigue y explica las características de la GPU.

```
1   void printDevProp(cudaDeviceProp devProp){
2       /* TODO */
3   }
4   int main(){
5       int i, devCount;
6       cudaGetDeviceCount(&devCount);
7       printf("%d CUDA devices.\n", devCount);
8       for (i = 0; i < devCount; ++i){
9           printf("CUDA Device #%d\n", i);
10          /* Get device properties */
11          cudaDeviceProp devProp;
12          cudaGetDeviceProperties(&devProp, i);
13          printDevProp(devProp);
14      }
15      return(0);
16  }
```

Compilación de códigos CUDA

- Hay que utilizar el compilador **nvcc**.
- Para compilar códigos pequeños, pero...
- para compilar códigos grandes necesitaremos un Makefile.
- Como ejemplo, utilizaremos la siguiente estructura.

```
|--> Project/
     |--> Makefile
     |--> src/ (source files)
     |--> include/ (header files)
     |--> bin/
```

Segundo código CUDA

- Ejecuta un código híbrido *hello_world* en la GPU.

```
1
2     int main(){
3         hellow_world(); /* C wrapper */
4         return(0);
5     }
```

Índice

Qué es una GP-GPU? Definición.

- *Scalable array of multithreaded Streaming Multiprocessors*
  - Está compuesto por múltiples Streaming Multi-processor.
  - Cada SM puede ejecutar cientos de hilos.
  - Los SM están compuestos por múltiples unidades funcionales para ejecutar:
    - Operaciones con enteros o de coma flotante,
    - operaciones matriciales (FMA),
    - en distintas precisiones (representaciones).
  - Utilizan la representación *little-endian*.

CPU vs GPU (1)

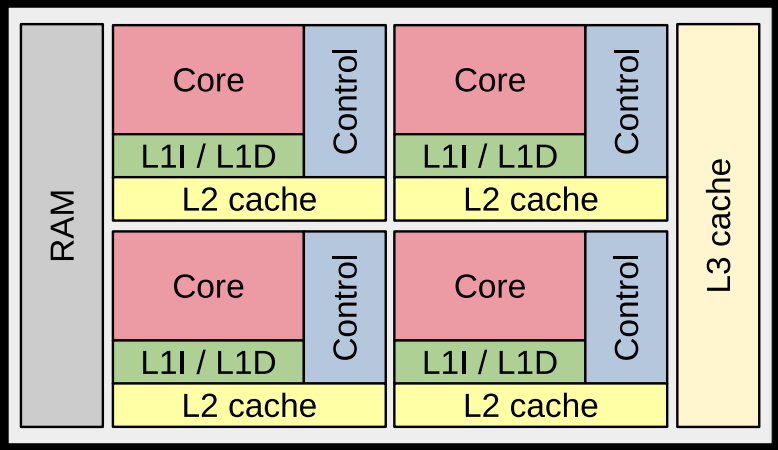
- El objetivo del diseño de las CPU es
  - ejecutar un conjunto de hilos pequeño y
  - hacerlo lo más rápido posible.
  - Los núcleos son de alto rendimiento. . .
  - **Problema**: La cantidad de núcleos limita el paralelismo.

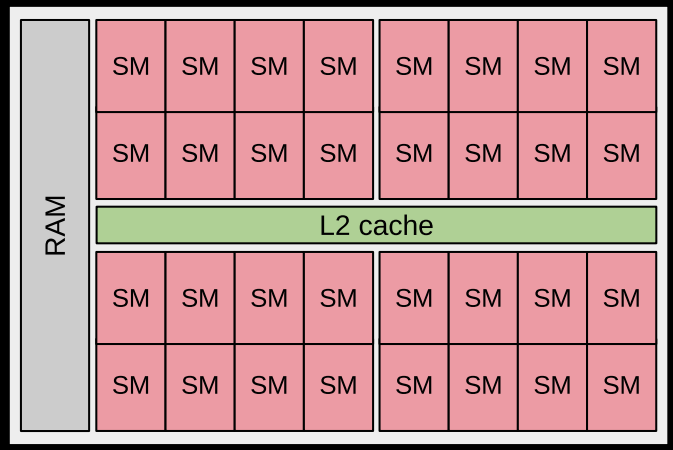
CPU vs GPU (2)

- Por el contrario, el objetivo del diseño de las GPU es
  - ejecutar miles de hilos de forma concurrente.
  - De esta manera, se balancea la perdida de rendimiento que surge por usar núcleos lentos.

CPU vs GPU (3)

- En las CPU se utilizan más transistores para hacer el control de flujo y cachear datos.
- En las GPU se utilizan más para procesar datos.





Ventajas de las GPU

- Utilizar más hardware para los programas muy paralelos es beneficioso.
- Haciendo cálculos se esconde la latencia por acceder a memoria.
- En las CPU para reducir la latencia de la memoria hay que utilizar mucho hardware.
  - Memoria caché y control de flujo complejo.
- En general, las aplicaciones tienen partes secuenciales y partes paralelas
  - utilizarán tanto la CPU como la GPU.

Desventajas de las GPU

- El código se ejecuta en la máquina *host* y
  - el código paralelo y los datos hay que enviarlos a la GPU.
- Para aprovechar bien la GPU hay que llenarla lo más posible pero . . .
  - los recursos de la GPU son limitados.

Organización de la ejecucion

- La programación mediante CUDA requiere conocer como se organizan los hilos en el hardware de la GPU.
  - El programador es el responsable de organizar la estructura de los hilos.
  - Puede tener un impacto muy alto en el rendimiento.
  - Es muy importante conocer el hardware.

Organización: hilos y bloques

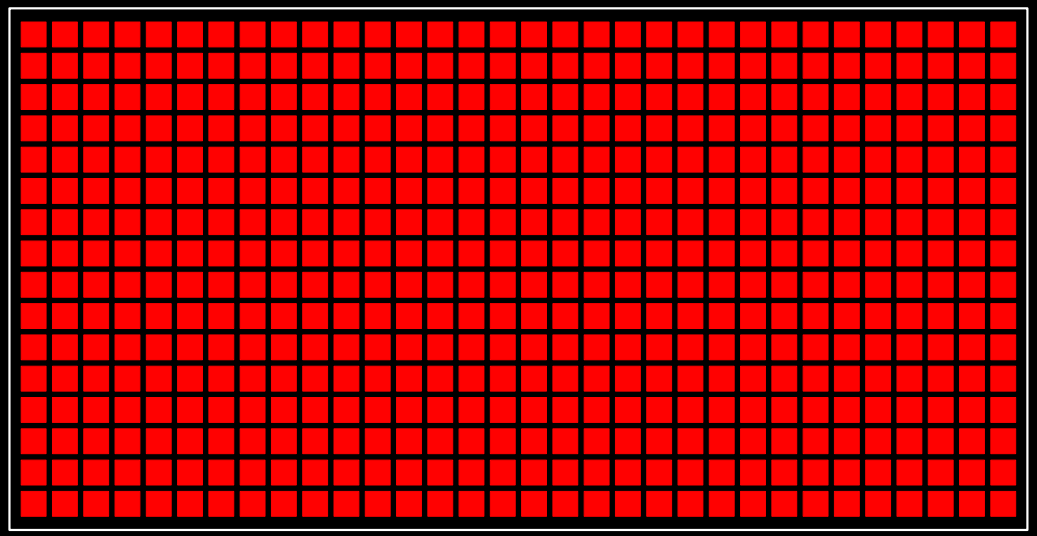
- Los hilos son organizados en bloques.
- Un bloque es ejecutado por un SM.
- Los hilos de un bloque son indexados usando:
  - índices de 1D: (x)
  - índices de 2D: (x,y)
  - índices de 3D: (x,y,z)
- Hay un **máximo** de hilos por bloque.

Oraganización: bloques y grid

- Conjunto de bloques lanzados en un kernel.
- Los bloques están agrupados en una *grid*.

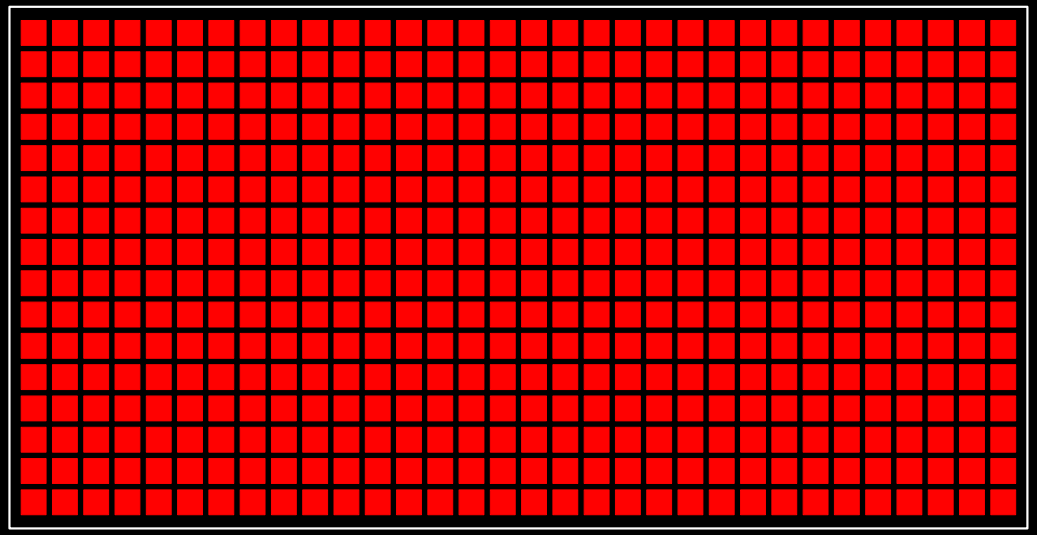
Ejemplo: Grid

- Conjunto de hilos a ejecutar (512).



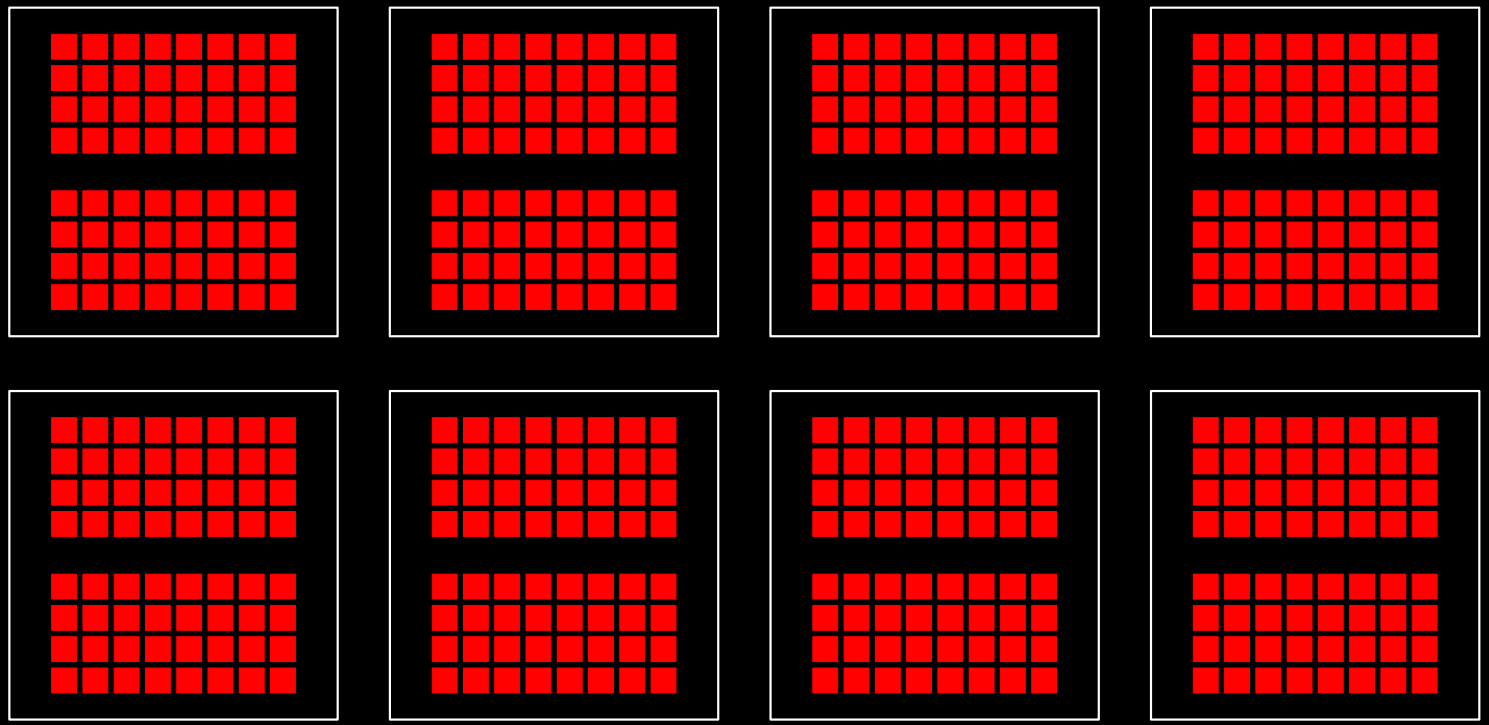
Ejemplo: Grid

- Conjunto de hilos a ejecutar (512).
  - Hay que organizar los hilos en bloques.



Ejemplo: Bloques

- Organización en bloques (8).



Organización en tiempo de ejecucion: Warp

- Cada bloque se asigna a un SM, pero...
  - solo puede haber 32 hilos concurrentes (?).
- Los bloques se planifican de 32 en 32 hilos (**warp**)
  - y cada **warp** se ejecuta en modo **lock-step**.
- La razón es que hay recursos (registros, UF, etc.) limitados.

Organización en tiempo de ejecucion: Warp

- A cada SM se le asigna un bloque y ejecuta un máximo de 32 hilos en paralelo (*warp*).

Identificación de hilos y bloques

- Los hilos se pueden identificar de diferentes formas estructurando el grid y los bloques al lanzar un kernel:
  - kernel_func <<< **gridShape**, **blockShape** >>>();
- Para especificar la estructura se utiliza el tipo **dim3** y si no se especifica un campo se asume *1*.

```
1   /*
2    * dim3 gridShape = dim3(MaxXGridDim, MaxYGridDim, MaxZGridDim);
3    * dim3 blockShape = dim3(MaxXBlkDim, MaxYBlkDim, MaxZBlkDim);
4    /*
5
6    /* Grid de 3 x 2 y Bloques de 2 x 2 */
7    dim3 gridShape = dim3(3, 2); // Equivalente a dim3(3, 2 , 1)
8    dim3 blockShape = dim3(2, 2); // Equivalente a dim3(2, 2 , 1)
9
10  /* Grid de 8 x 1 y Bloques de 8 x 1 */
11   dim3 gridShape = dim3(8);    // Equivalente a dim3(8, 1 , 1)
12                                // y a int gridShape = 8;
13   dim3 blockShape = dim3(8);   // Equivalente a dim3(8, 1 , 1)
14                                // y a int blockShape = 8;
```

Identificación de hilos y bloques

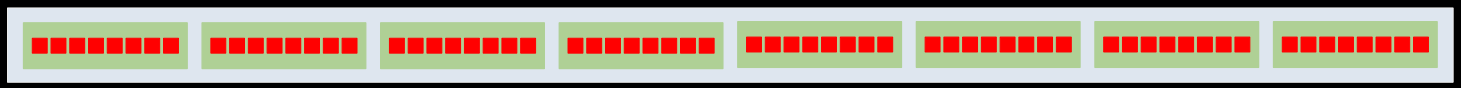
- Número de bloques en un grid
  - gridDim.x: en la dim *x* de un grid
  - gridDim.y: en la dim *y* de un grid
  - gridDim.z: en la dim *z* de un grid
- Número de hilos en un bloque
  - blockDim.x: en la dim *x* de un bloque
  - blockDim.y: en la dim *y* de un bloque
  - blockDim.z: en la dim *z* de un bloque

Identificación de hilos y bloques

- Índice de un bloque en un grid
  - blockIdx.x: en la dim *x*
  - blockIdx.y: en la dim *y*
  - blockIdx.z: en la dim *z*
- Índice de un hilo en un bloque
  - threadIdx.x: en la dim *x*
  - threadIdx.y: en la dim *y*
  - threadIdx.z: en la dim *z*

Identificación de hilos y bloques

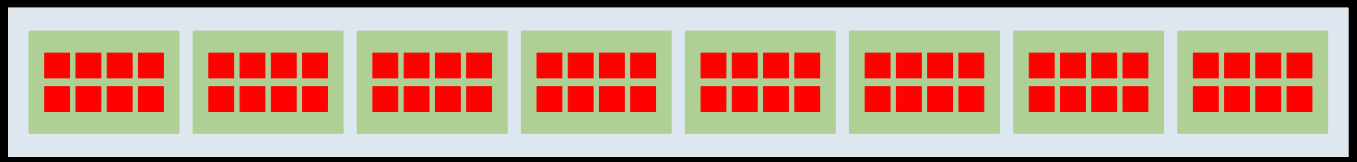
- Grid 1D y Bloques 1D
  - 64 hilos: 8 bloques de 8 hilos
  - **dim3** gridShape = **dim3**(8);
  - **dim3** blockShape = **dim3**(8)
  - Lanzamiento: <<< gridShape, blockShape >>>



```
1    __device__ int getGlobalIdx_1D_1D() {
2        return blockIdx.x *blockDim.x + threadIdx.x;
3    }
```

Identificación de hilos y bloques

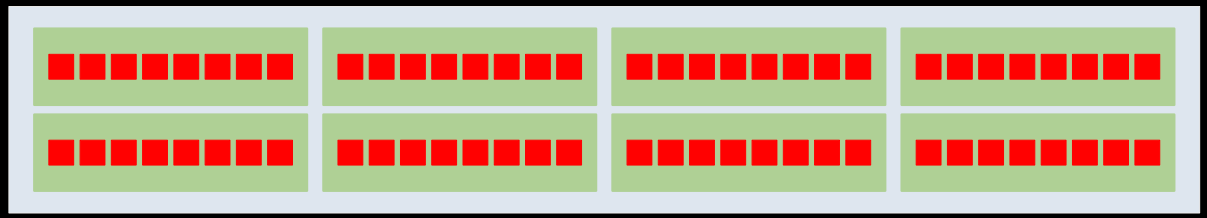
- Grid 1D y Bloques 2D
  - 64 hilos: 8 bloques de (2x4) hilos
  - **dim3** gridShape = **dim3**(8);
  - **dim3** blockShape = **dim3**(2, 4)
  - Lanzamiento: <<< gridShape, blockShape >>>



```
1    __device__ int getGlobalIdx_1D_2D() {
2          return blockIdx.x  * blockDim.x * blockDim.y
3             + threadIdx.y * blockDim.x + threadIdx.x;
4    }
```

Identificación de hilos y bloques

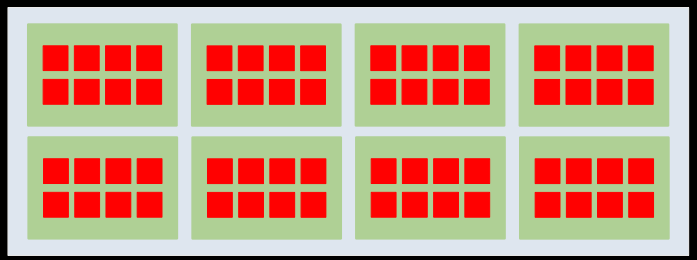
- Grid 2D y Bloques 1D
  - 64 hilos: (2x4) bloques de 8 hilos
  - **dim3** gridShape = **dim3**(2,4);
  - **dim3** blockShape = **dim3**(8)
  - Lanzamiento: <<< gridShape, blockShape >>>



```
1    __device__ int getGlobalIdx_2D_1D() {
2          int blockId   = blockIdx.y * gridDim.x + blockIdx.x;
3          int threadId = blockId * blockDim.x + threadIdx.x;
4          return threadId;
5    }
```

Identificación de hilos y bloques

- Grid 2D y Bloques 2D
  - 64 hilos: (2x4) bloques de (2x4) hilos
  - **dim3** gridShape = **dim3**(2,4);
  - **dim3** blockShape = **dim3**(2,4)
  - Lanzamiento: <<< gridShape, blockShape >>>



```
1    __device__ int getGlobalIdx_2D_2D() {
2          int blockId = blockIdx.x + blockIdx.y * gridDim.x;
3          int threadId = blockId * (blockDim.x * blockDim.y)
4                + (threadIdx.y * blockDim.x) + threadIdx.x;
5          return threadId;
6    }
```

Tercer código CUDA

- Añade el código necesario al programa “Hello_world” para lanzar e identificar los hilos de las 9 posibles formas.

```
1    /* File id.cu which contains 9 functions to identify the threads.
2     * Launch the Hellow_world kernel using 9 different structures.
3     * Each thread should identify itself with the global id and the
4     * coordinates inside the block.
5     */
```

Compute capabilities

- Especificaciones y características de una arquitectura

Compute capability	Microarquitectura	Año
1.x	Tesla	2006
2.x	Fermi	2010
3.x	Kepler	2012
5.x	Maxwell	2014
6.x	Pascal	2016
7.0-7.2	Volta	2017
7.5	Turing	2018
8.x	Ampere	2020
9.x	Hopper	2022

Compute capabilities

Data center GPU	Tesla P100	Tesla V100	A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	Pascal	Volta	Ampere
Compute Capability	6.0	7.0	8.0
Threads / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks / SM	32	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
FP32 Cores / SM	64	64	64
Ratio SM Regs / FP32 Cores	1024	1024	1024
Shared Memory Size / SM	64 KB	96 KB	164 KB

Compute capabilities

Data center GPU	Tesla P100	Tesla V100	A100
GPU Codename	GP100	GV100	GA100
GPU Board Form Factor	SXM	SXM2	SXM4
SMs	56	80	108
TPCs	28	40	54
FP32 Cores / SM	64	64	64
FP32 Cores / GPU	3584	5120	6912
FP64 Cores / SM	32	32	32
FP64 Cores / GPU	1792	2560	3456
INT32 Cores / SM	NA	64	64
INT32 Cores / GPU	NA	5120	6912
Tensor Cores / SM	NA	8	42
Tensor Cores / GPU	NA	640	432
GPU Boost Clock	1480 MHz	1530 MHz	1410 MHz
Texture Units	224	320	432

Compute capabilities

Data center GPU	Tesla P100	Tesla V100	A100
GPU Codename	GP100	GV100	GA100
Memory Interface	4096-bit HBM2	4096-bit HBM2	5120-bit HBM2
Memory Size	16 GB	32 GB / 16 GB	40 GB
Memory Data Rate	703 MHz DDR	877.5 MHz DDR	1215 MHz DDR
Memory Bandwidth	720 GB/sec	900 GB/sec	1555 GB/sec
L2 Cache Size	4096 KB	6144 KB	40960 KB
Shared Memory Size / SM	64 KB	96 KB	164 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	14336 KB	20480 KB	27648 KB

Compute capabilities

Data center GPU	Tesla P100	Tesla V100	A100
GPU Codename	GP100	GV100	GA100
TDP	300 Watts	300 Watts	400 Watts
Transistors	15.3 billion	21.1 billion	54.2 billion
GPU Die Size	610 mm^2	815 mm^2	826 mm2
TSMC Manufacturing Process	16 nm FinFET+	12 nm FFN	7 nm N7

Cuarto código CUDA

- Implementa el siguiente programa en CUDA y mide el tiempo de ejecución.

```
1    /* Add two randomly generated vectors A and B into a third vector C
2     * where the size of the vectors is N = 1024 * 1024
3     * Use the followings functions:
4     * cudaMalloc(...), cudaMemcpy(...), cudaFree(...)
5     * cudaEvent_t, cudaEventCreate(...), cudaEventRecord(...),
6     * cudaEventSynchronize(...), cudaEventElapsedTime(...)
7     */
```

Quinto código CUDA

- Implementa de nuevo el cuarto código sin utilizar *cudaMalloc*, *cudaMemcpy* y *cudaFree*.

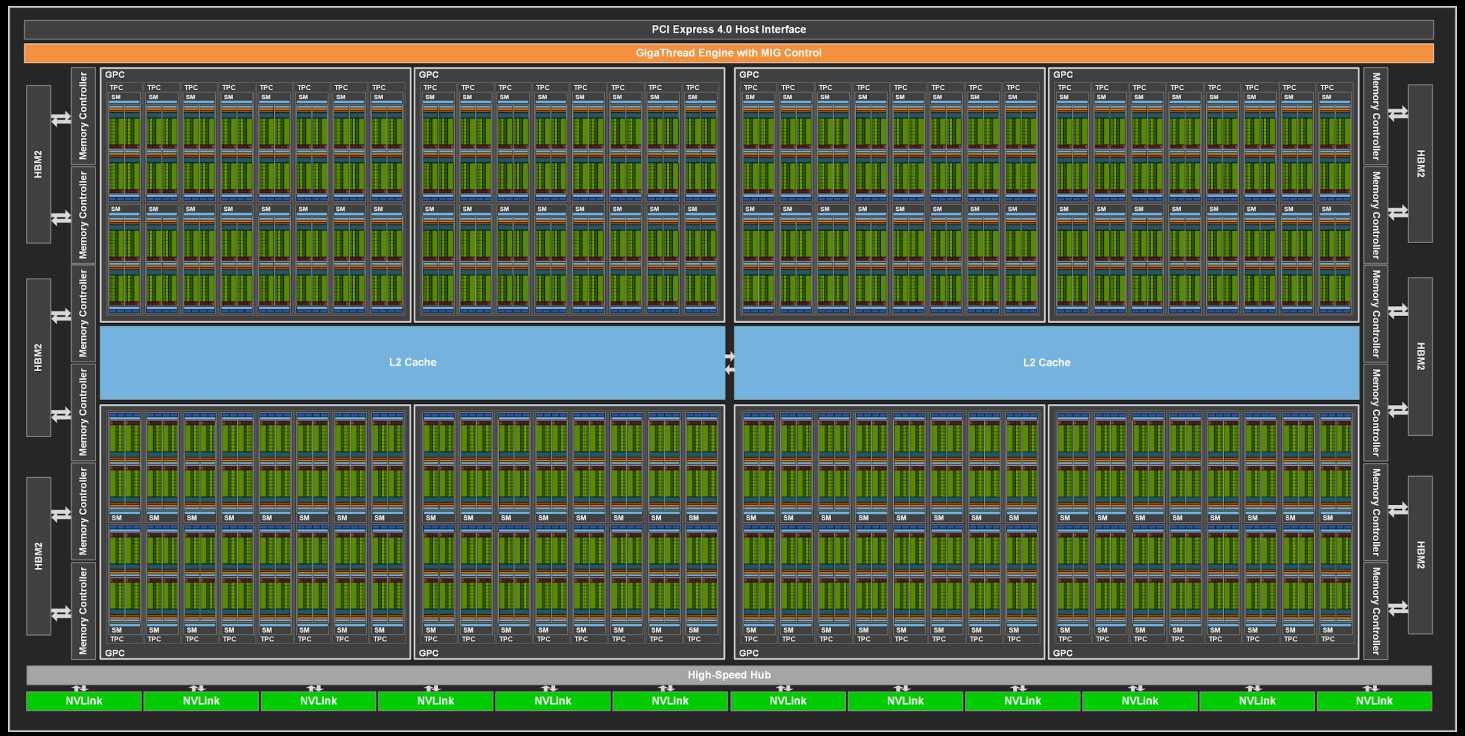
```
1    /* Add two randomly generated vectors A and B into a third vector C
2     * where the size of the vectors is N = 1024 * 1024
3     * Use the followings functions:
4     * cudaEvent_t, cudaEventCreate(...), cudaEventRecord(...),
5     * cudaEventSynchronize(...), cudaEventElapsedTime(...)
6     */
7
```

Arquitectura Ampere (GA100)

- La arquitectura **Ampere** está compuesta por:
  - GPU processing clusters (GPC)
  - Texture processing clusters (TPC)
  - Streaming multiprocessors (SM)
  - High Bandwidth Memory 2 (HBM2)
- **GA100** es la arquitectura completa y **A100** es una implementación parcial.

Arquitectura Ampere (GA100)

- 8 GPC8 TPC/GPC, 2 SM/TPC, 16 SM/GPC, 128 SM
- 64 FP32 CUDA Cores/SM, 8192 FP32 CUDA Core/GPU
- 4 Tensor Cores/SM, 512 Tensor Cores/full GPU
- 6 HBM2 stacks, 12 512-bit memory controllers



Arquitectura Ampere (A100)

- 7 GPC, 8 TPC/GPC, 2 SM/TPC, 16 SM/GPC, 108 SM
- 64 FP32 CUDA Cores/SM, 6912 FP32 CUDA Cores/GPU
- 4 Tensor Cores/SM, 432 Tensor Cores/GPU
- 5 HBM2 stacks, 10 512-bit memory controllers

Arquitectura Ampere(SM)

- Memoria caché L0(I)
- Warp Scheduler
- Dispatch Unit
- 16 INT32
- 16 FP32
- 8 FP64
- 1 TENSOR CORE
- Register file (16384)
- 8 LD / ST
- 4 SFU

Arquitectura Ampere (SM)

- x4

Arquitectura Ampere (SM)

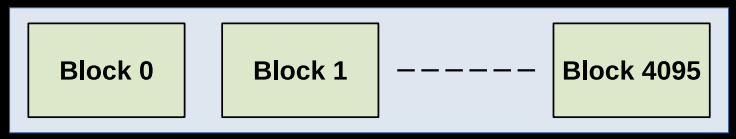
- 192 KB-ko L1 (D) / mem
- Memoria caché L1 (I)
- 4 Texture processors

Modelo de ejecución

1. Lanzar aplicación de 4096 bloques y 256 hilos/bloque
2. Cada bloque se asigna a un SM
3. Los bloques se dividen en warps
4. Los warp schedulers planifican los warps
5. Cada instrucción se asigna a una unidad funcional
6. La ejecución de los warps en desorden
   - pero dentro de cada warp en orden
7. El contexto de cada warp se mantiene (PC, registros, etc)
   - La memoria y la caché se particionan entre bloques
8. Número de bloques y warps limitado por la GPU

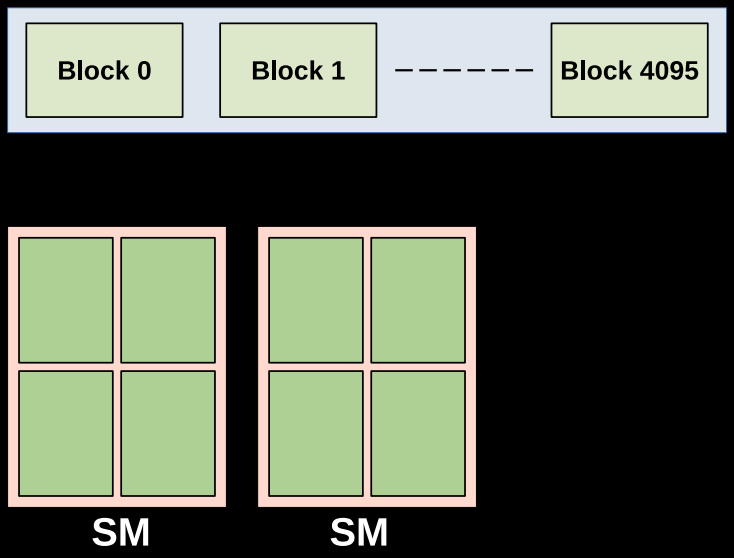
Modelo de ejecución

- Aplicación de 4096 bloques y 256 hilos



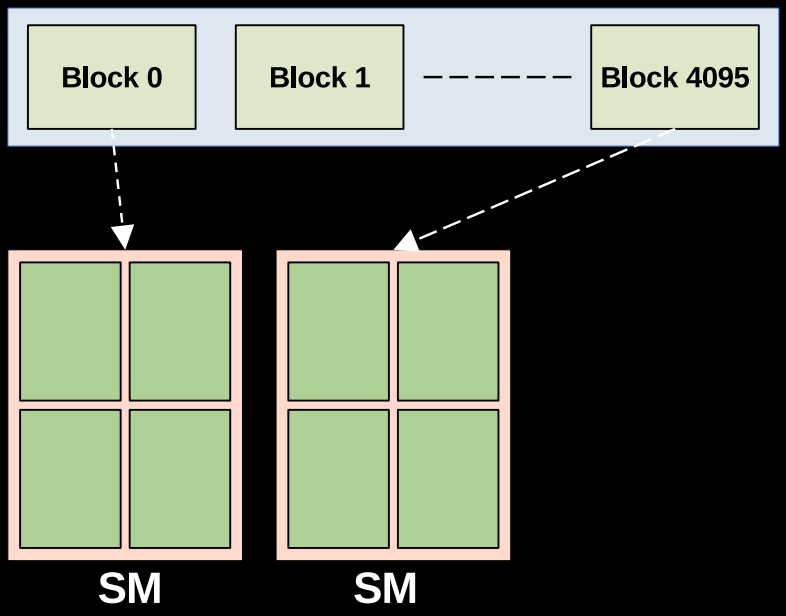
Modelo de ejecución

- Los SM tienen 4 particiones (4 warp schedulers)



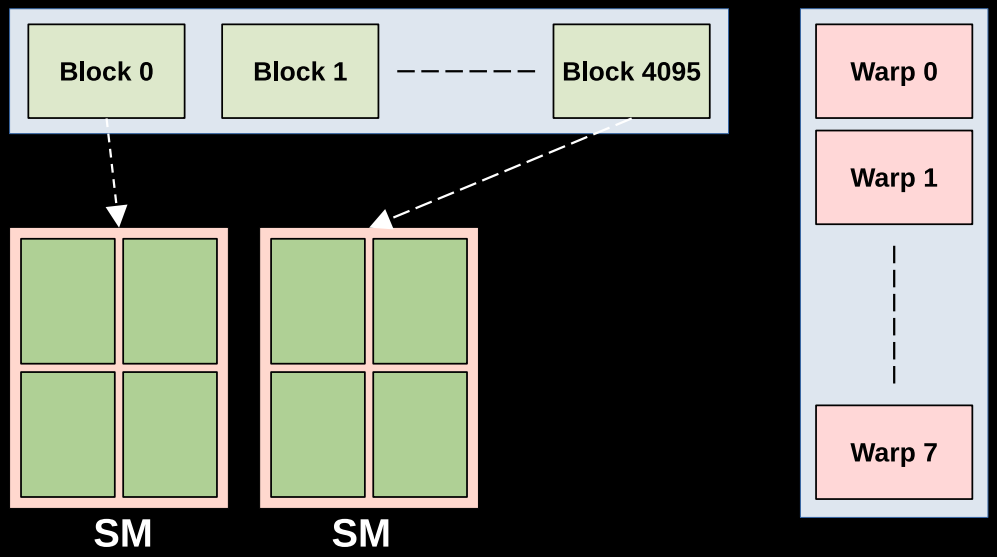
Modelo de ejecución

- Cada bloque se asigna a un SM



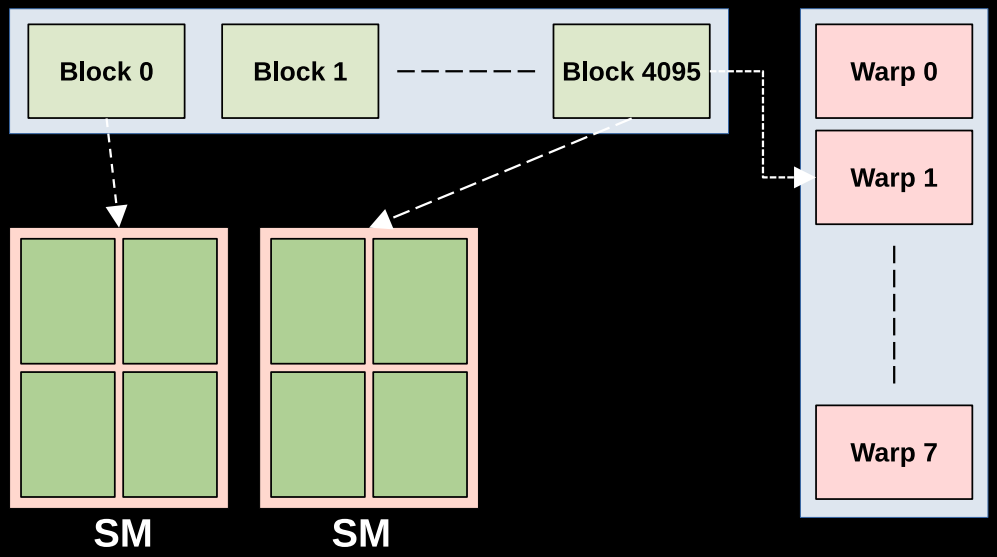
Modelo de ejecución

- Cada bloque se ejecuta en conjuntos de 32 hilos



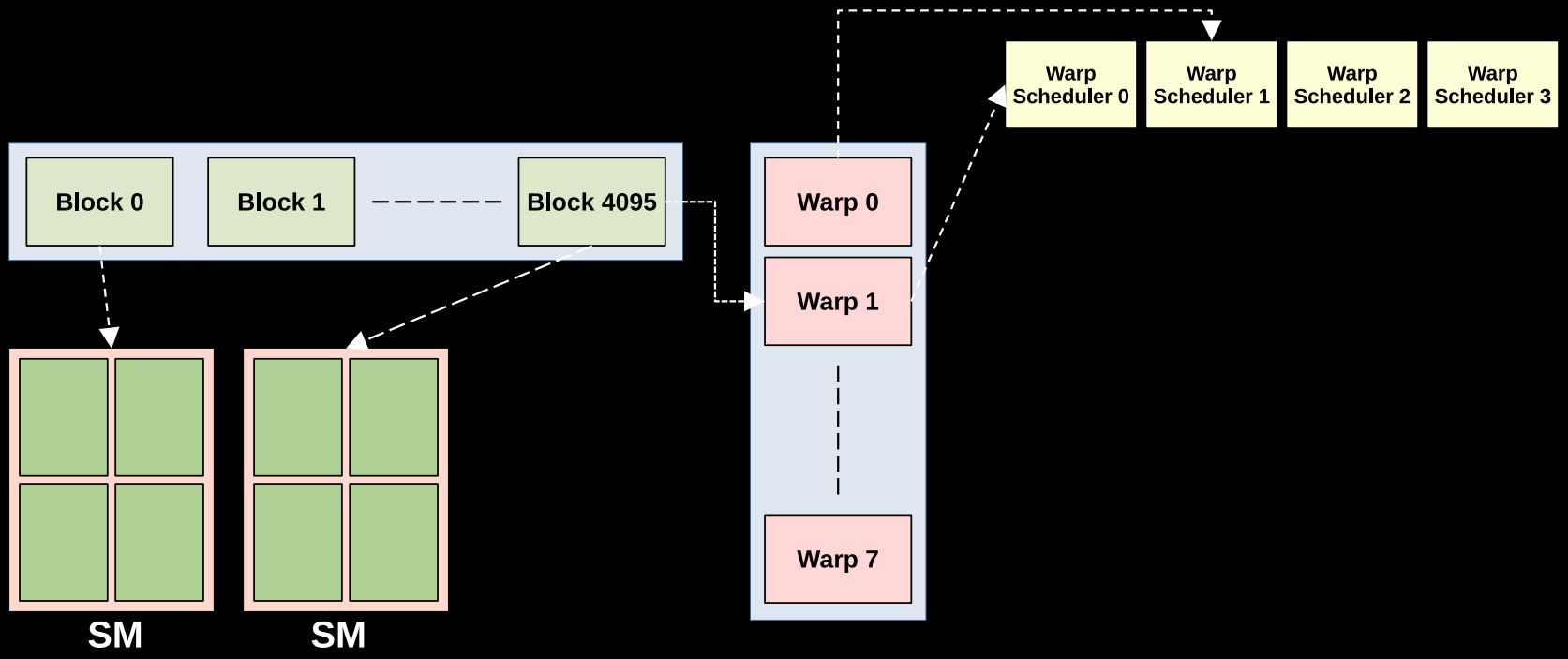
Modelo de ejecución

- Cada bloque se divide en warps (32 hilos)



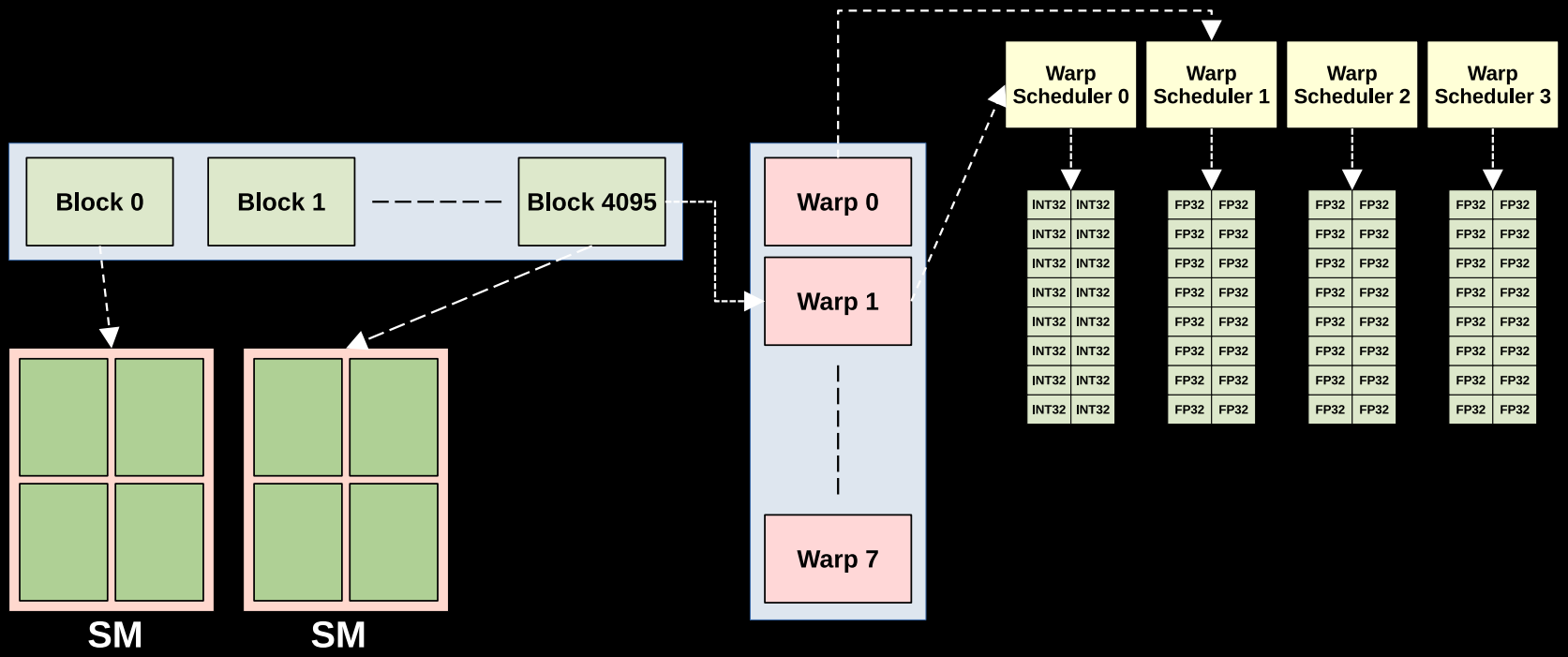
Modelo de ejecución

- Cada warp scheduler planifica uno o varios warps



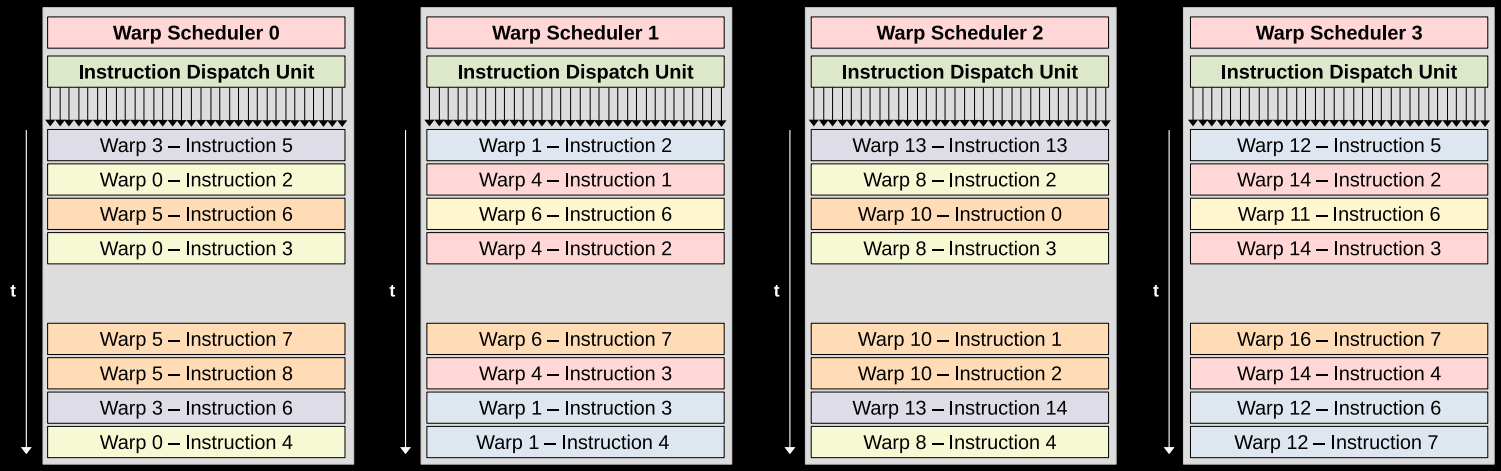
Modelo de ejecución

- Los cuatro schedulers lanzan instrucciones del mismo warp en orden o de diferentes en desorden



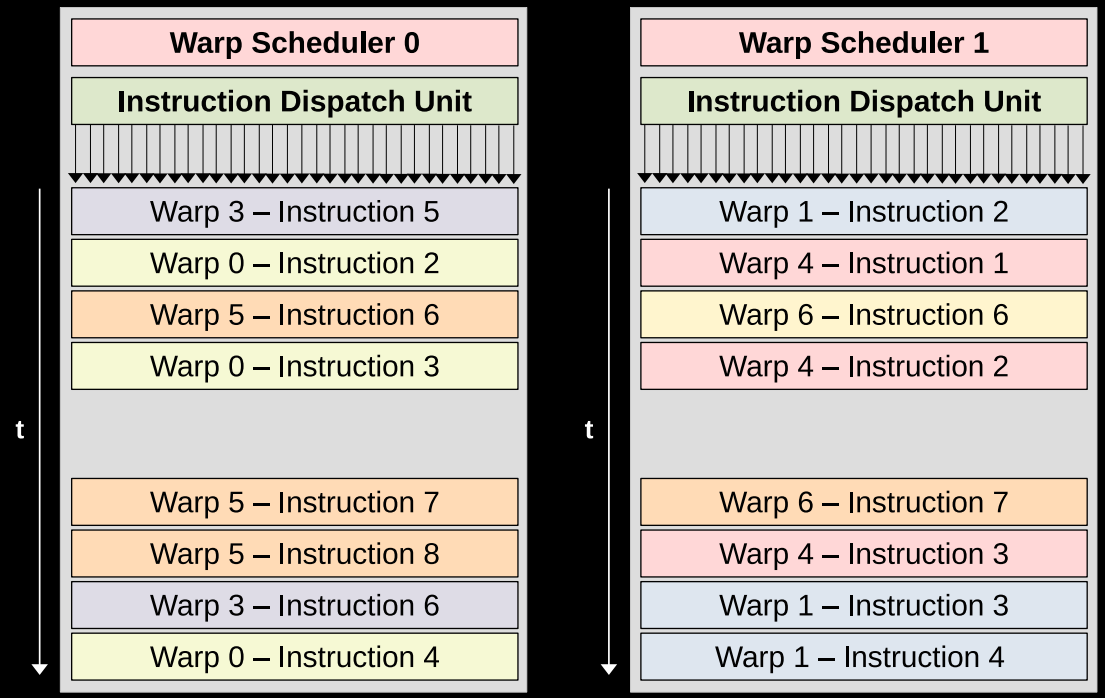
Warp schedulers

- Ejemplo de ejecución de instrucciones en 4 schedulers



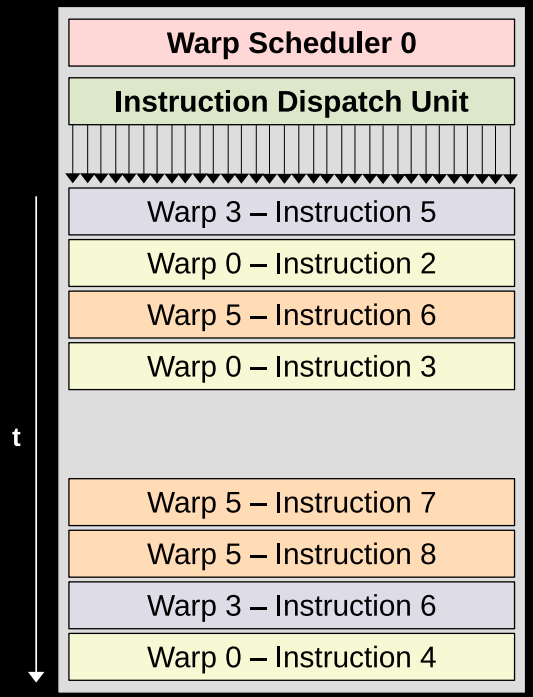
Warp schedulers

- Ejemplo de ejecución de instrucciones en 2 schedulers



Warp schedulers

- Ejemplo de ejecución de instrucciones en 1 scheduler



Arquitectura Multi-Instance GPU (MIG)

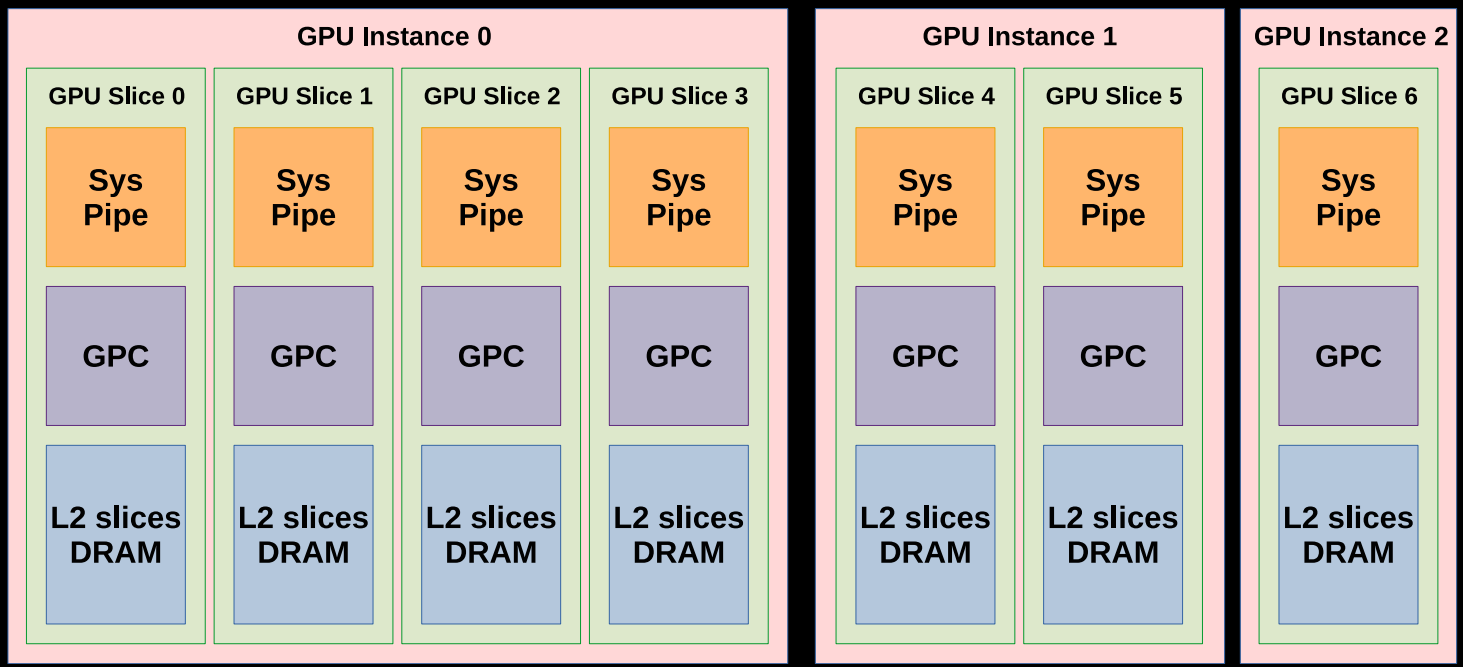
- Volta *Multi-process service* permite ejecutar *kernels* y operaciones de memoria de procesos diferentes de manera concurrente.
  - Sin embargo, las memorias caché y el ancho de banda son compartidos,
  - lo cual tiene un impacto negativo en el rendimiento.
- La solución en *Ampere* es virtualizar la GPU
  - creando hasta 7 GPU virtuales (instancias),
  - cada una de ellas con recursos dedicados.

Arquitectura Multi-Instance GPU (MIG)

- Cada instancia tiene SM dedicados o caminos independientes en el sistema de memoria:
  - *Crossbars* para el control.
  - Particiones de la caché L2.
  - Controladores de memoria.
  - Memoria y buses de acceso.
- Proporciona *QoS* y tolerancia a fallos.

Arquitectura Multi-Instance GPU (MIG)

- Ejemplo de 3 instancias de GPU.
  - Los trabajos pueden migrar entre instancias.



Sistema de memoria

- 40 GB de memoria HBM2 a 1555 GB/s
  - DRAM organizada en 5 *stacks* en 3D.
  - Mayor ancho de banda y menor consumo que GDDR6.
  - Utiliza códigos de corrección de errores.

Memoria caché L2

- 40 MB de memoria caché L2
- La caché L2 está dividida en dos particiones:
  - Mayor ancho de banda y menor latencia
- La coherencia hardware mantiene la consistencia de los datos de una aplicación en toda la GPU.
- Cada partición de la caché se divide en 40 sub-particiones de 512 KB.

Copia asíncrona de memoria global a compartida

- Nueva instrucción de copia asíncrona (CUDA 11)
- Permite copiar datos de memoria global directamente a memoria compartida
  - sin pasar por la memoria caché L1
  - y sin utilizar registros intermedios.

Índice

CUDA

- Compute Unified Device Architecture (CUDA)
  - Plataforma para la programación de GPU de NVidia
  - Muy ligada a la arquitectura del hardware
  - Versiones de CUDA ligadas a las CC

Estructura de un programa CUDA

- Flujo de un programa CUDA
  1. Copiar los datos de la memoria del host a la GPU
  2. La CPU lanza el *kernel* con la estructura adecuada
  3. La GPU ejecuta el kernel en paralelo
  4. Se copia el resultado de la memoria de la GPU a la del host
  5. Se libera la memoria en la GPU y en el host

Variables y constantes

- __**shared**__: La variable se almacena en la memoria compartida (SMEM) del SM y es compartida por los hilos que pertenecen al mismo bloque. **Acceso rápido.**
- __**device**__: La variable se almacena en la memoria global y puede ser usada por todos los hilos del un kernel. **Acceso lento**.
- __**constant**__: Variable de **solo lectura.** Se encuentra mapeada en una área de la memoria caché L2. Es útil para almacenar contantes a las que acceden todos los hilos de un *warp*.

Variables *built-in*

- **gridDim**: Tamaño del grid.
- **blockDim**: Tamaño del bloque.
- **blockIdx**: Identificador del bloque en el grid.
- **theadIdx**: Identificador del hilo en el bloque.

Tipos de funciones

- __**host**__: Función que llama y ejecuta el host. No es necesaria.
- __**device**__: Función que llama y ejecuta la GPU.
- __**global**__: Función (kernel) que llama el host y ejecuta el dispositivo (GPU). Es de tipo *void* y solo puede acceder a memoria de la GPU.

Lanzamiento de kernels

- Los hilos se pueden identificar de diferentes formas estructurando el grid y los bloques al lanzar un kernel:
  - kernel_func <<< **gridShape**, **blockShape** >>>();
- Para especificar la **estructura** del **grid** y de los **bloques** se utiliza el tipo de dato **dim3**. Ver la Sección sobre **Identificación de hilos**.

Reserva y transferencia de memoria

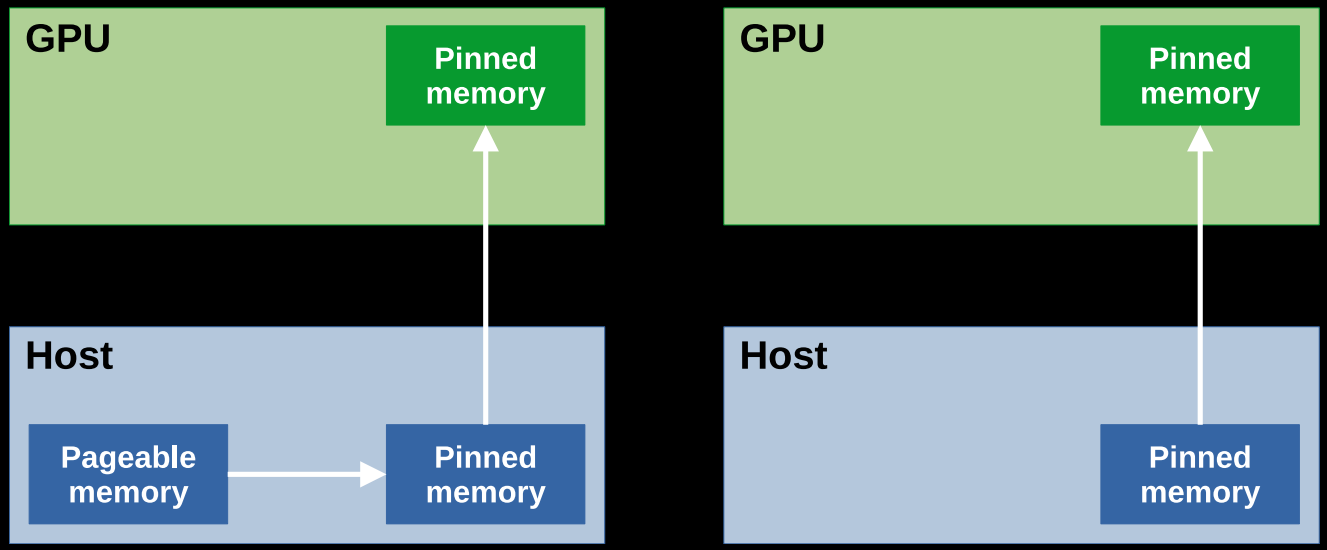
- Reserva y transferencia síncrona

```
cudaError_t cudaMalloc (void** devPtr, size_t size);
cudaError_t cudaFree(void* devPtr);
cudaError_t cudaMemset(void* devPtr, int  value, size_t count);
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
            cudaMemcpyKind kind);
```

Memoria *pinned*

- La reserva “normal” de memoria se hace sobre memoria estándar que puede ser paginada.
- Por ello, CUDA lo copia sobre una zona especial de la memoria (temporal)
  - que no puede ser paginada, pero que
  - implica una copia extra de los datos.
- Para evitarlo, se puede utilizar reservar *pinned memory* directamente y ahorrar una copia.
- Peligroso, porque el SO y otros programas se pueden quedar sin memoria.

Memoria *pinned*

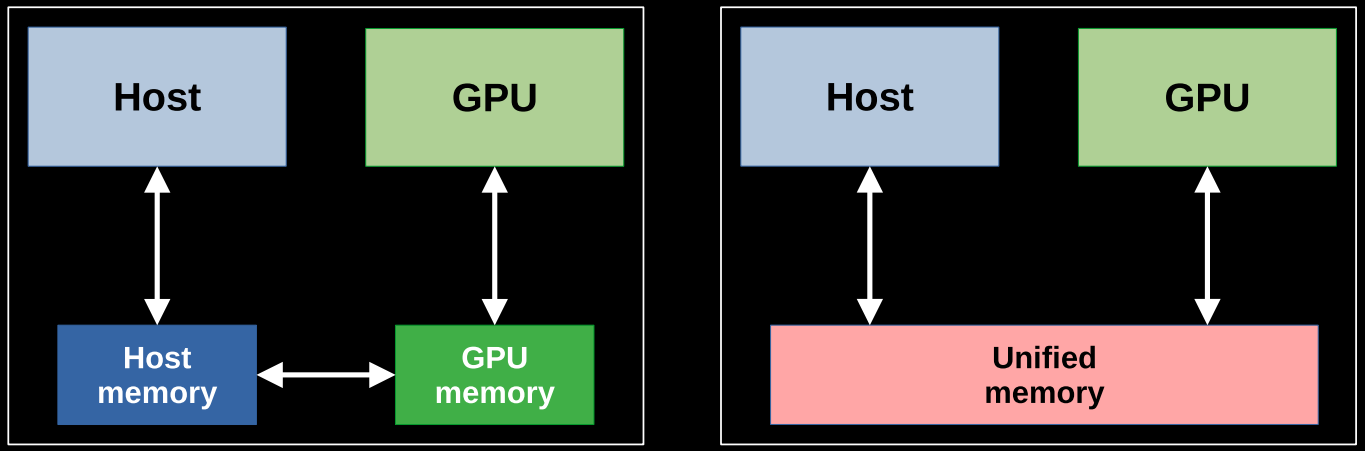



```
cudaError_t cudaMallocHost((void**)devPtr, size_t size);
```

Memoria *unified*

- Memoria *pinned* que es visible tanto en el host como en la GPU.
- No hay transferencias explícitas, CUDA las hace internamente.
- Se reduce la complejidad del código.

Memoria *unified*




```
cudaError_t cudaMallocManaged(void** devPtr, size_t size, unsigned int
      flags = cudaMemAttachGlobal)
```

Sincronización explícita de streams

- cudaDeviceSynchronize();
- cudaStreamSynchronize();
- cudaStreamWaitEvent();
- cudaStreamQuery();

Sincronización de hilos de un kernel

- void __syncthreads();
- int __syncthreads_count(int predicate);
- int __syncthreads_and(int predicate);
- void __syncwarp(unsigned mask=0xffffffff);

Operaciones atómicas

- Operaciones atómicas sobre variables compartidas
  - atomicAdd ()
  - atomicSub ()
  - atomicMax()
  - atomicMin()

Medición de tiempos

- Tipo: cudaEvent_t
- Funciones
  - cudaEventCreate (&t0);
  - cudaEventRecord (t0);
  - cudaEventSynchronize
  - cudaEventElapsedTime
  - cudaEventDestroy

Medición de tiempos

```
1   /*
2    * dim3 gridShape = dim3(MaxXGridDim, MaxYGridDim, MaxZGridDim);
3    * dim3 blockShape = dim3(MaxXBlkDim, MaxYBlkDim, MaxZBlkDim);
4    */
5
6    float         t;
7    cudaEvent_t   t0, t1;
8
9    cudaEventCreate (&t0);    // sortu objektua
10   cudaEventCreate (&t1);
11
12   cudaEventRecord (t0);    // t0-n uneko denbora
13
14   /* Launch kernel */
15
16   cudaEventRecord (t1);    // t1-n uneko denbora
17
18   cudaEventSynchronize (t1); // itxaron t1 prest egon arte
19   cudaEventElapsedTime (&Tex, t0, t1); //t0 eta t1-en arteko
20   printf (" Tex (kernela): %f\n", Tex);
21   cudaEventDestroy (t0);    // ezabatu objektua
22   cudaEventDestroy (t1);
23
```