Tema 2: PARALELISMO DE DATOS
Parte 2: GPU's

Universidad de Cádiz

ÍNDICE

Pipeline Básico

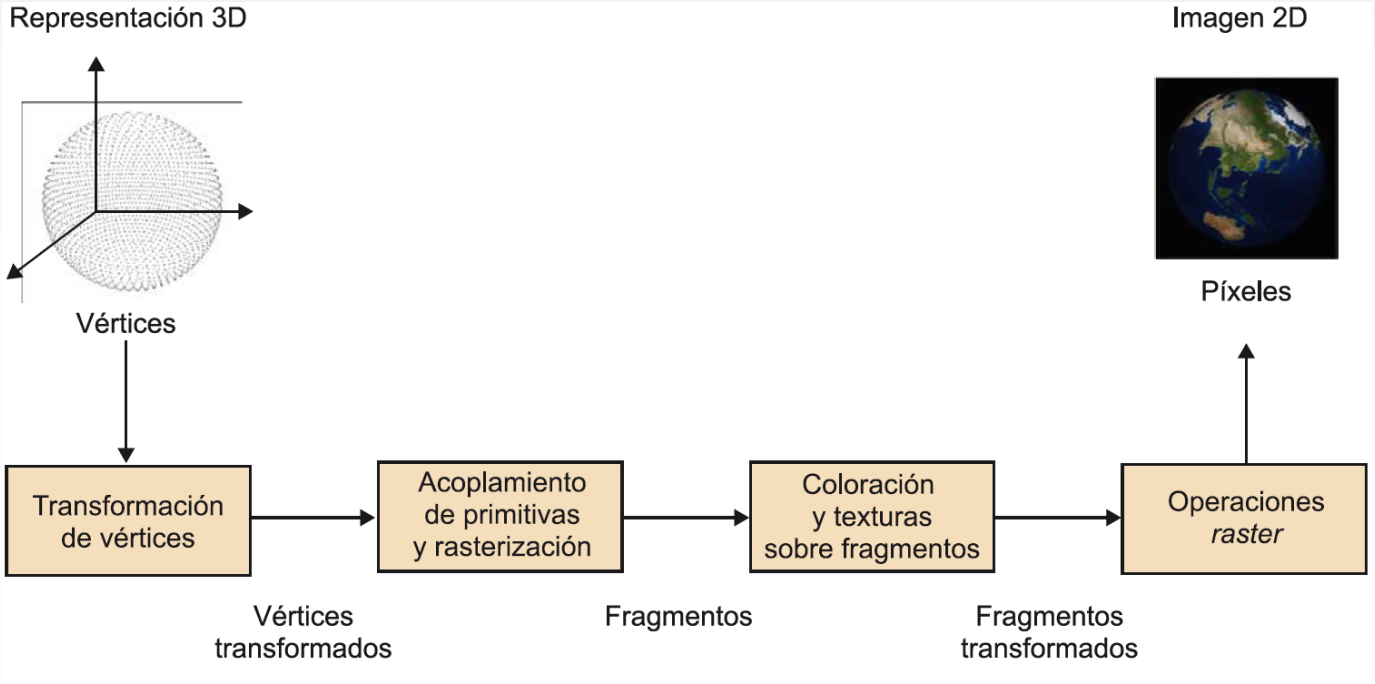
Los procesadores gráficos funcionan mediante un pipeline de procesamiento formado por etapas.

Cada una de las etapas del pipeline recibe la salida de la etapa anterior y proporciona su salida a la etapa siguiente.

Como este pipeline es específico para la gestión de gráficos, normalmente se denomina pipeline gráfico o pipeline de renderización

La entrada del procesador gráfico es una secuencia de vértices agrupados en primitivas geométricas (polígonos, líneas y puntos), que son tratadas secuencialmente por medio de cuatro etapas básicas.

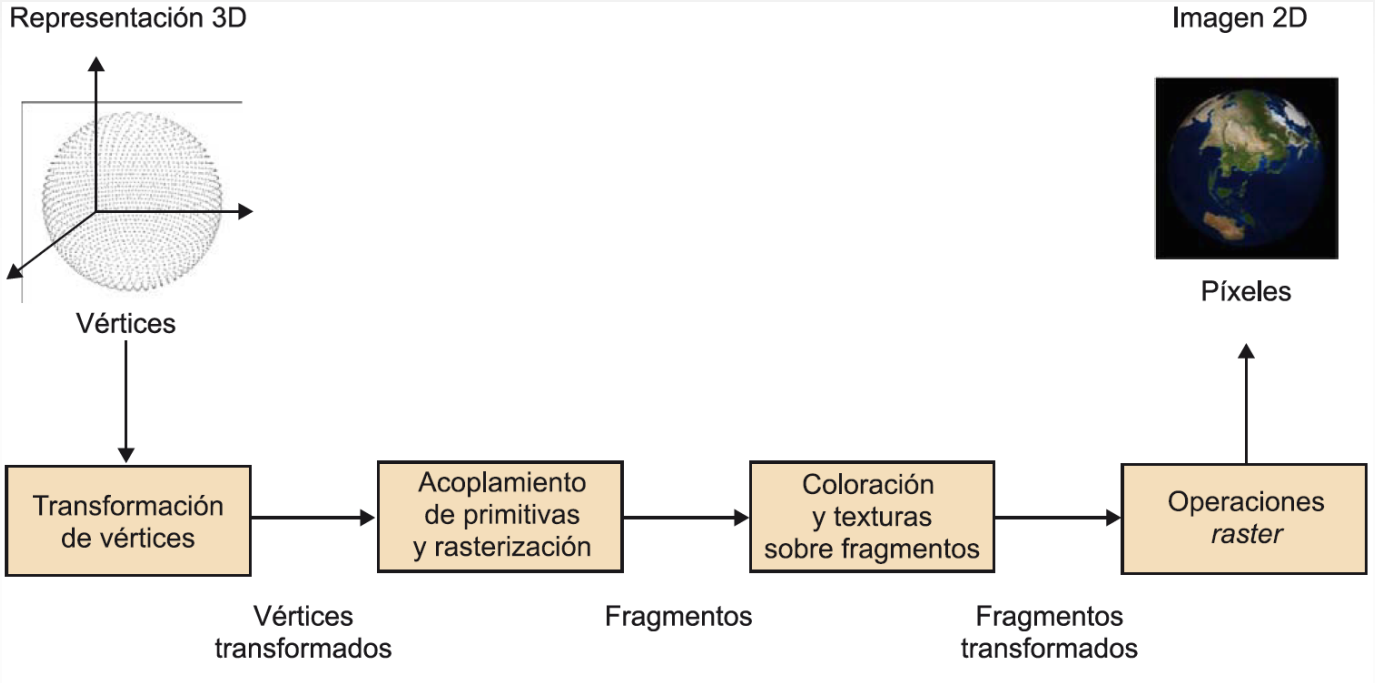
Pipeline Básico



Consta de 4 etapas:

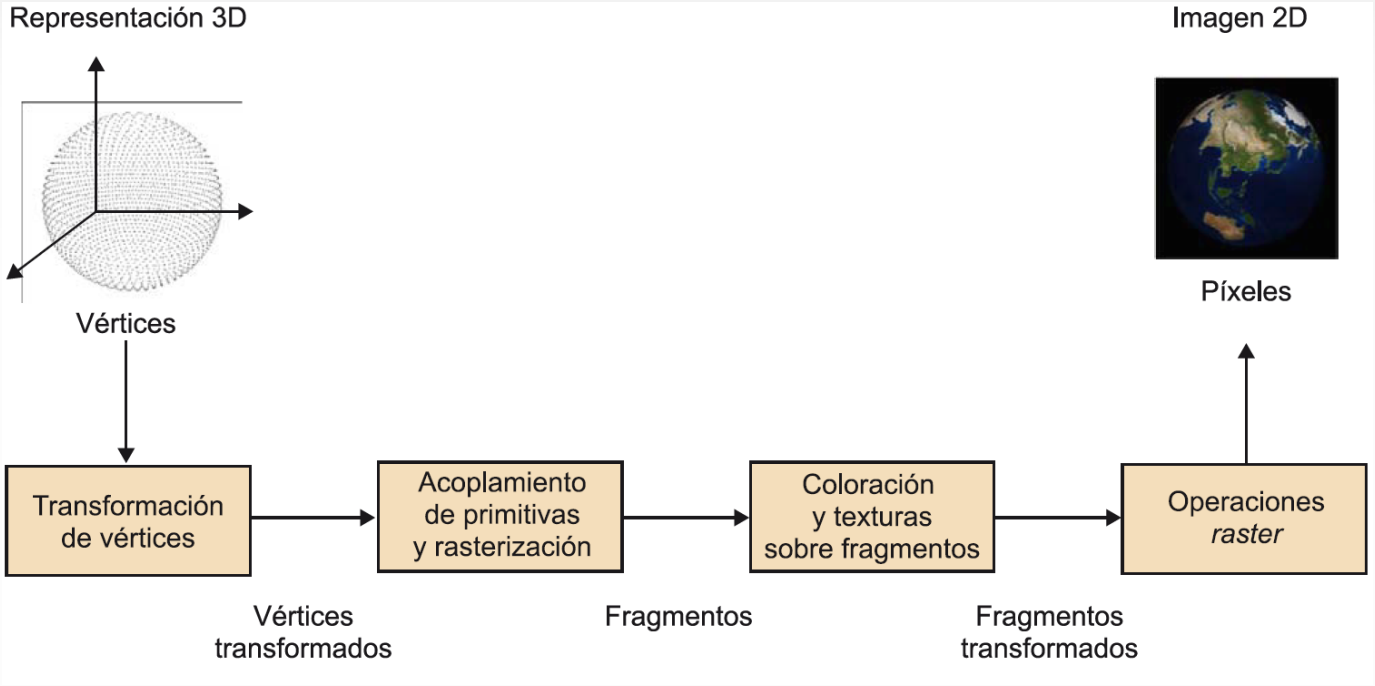
1ª etapa: La etapa de transformación de vértices consiste en la ejecución de una secuencia de operaciones matemáticas sobre los vértices de entrada basándose en la representación 3D proporcionada.

Pipeline Básico



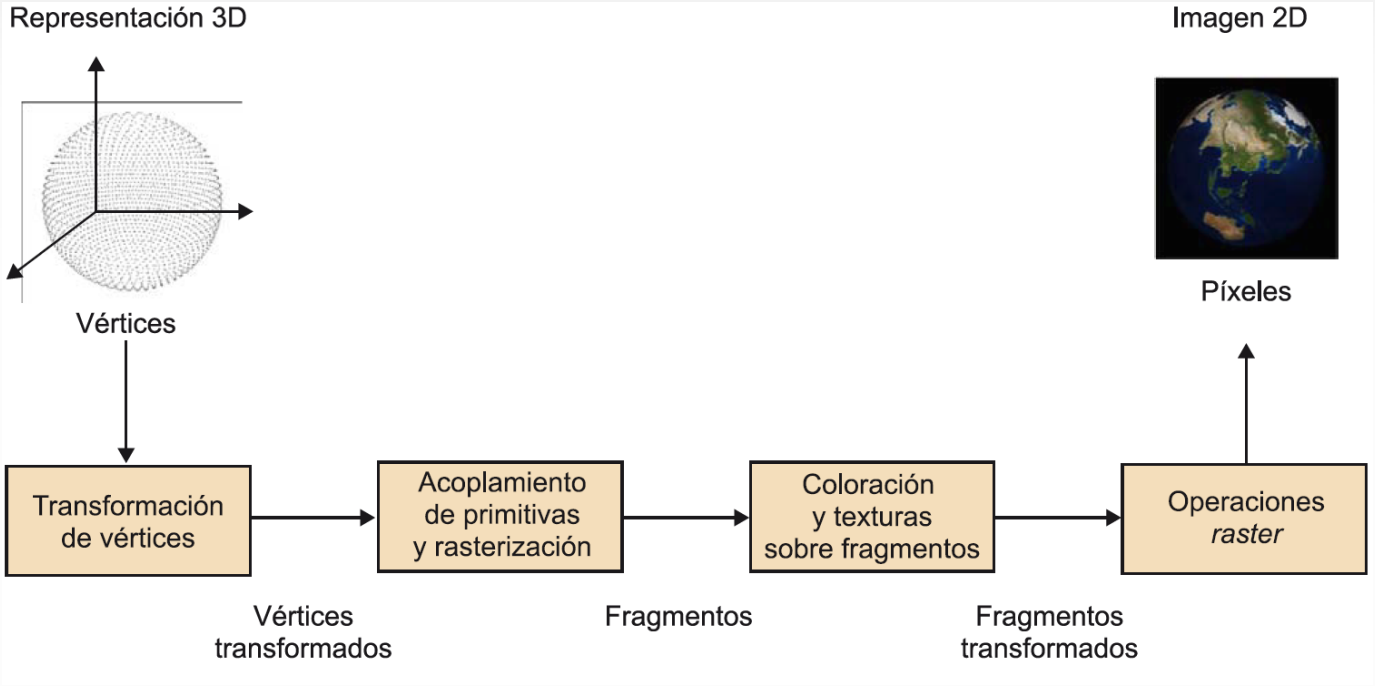
2ª etapa: En la etapa de acoplamiento de primitivas y rasterización, los vértices transformados se agrupan en primitivas geométricas basándose en la información recibida junto con la secuencia inicial de vértice. Como resultado, se obtiene una secuencia de triángulos, líneas y puntos. Los resultados de la rasterización son conjuntos de localizaciones de píxeles y conjuntos de fragmentos.

Pipeline Básico



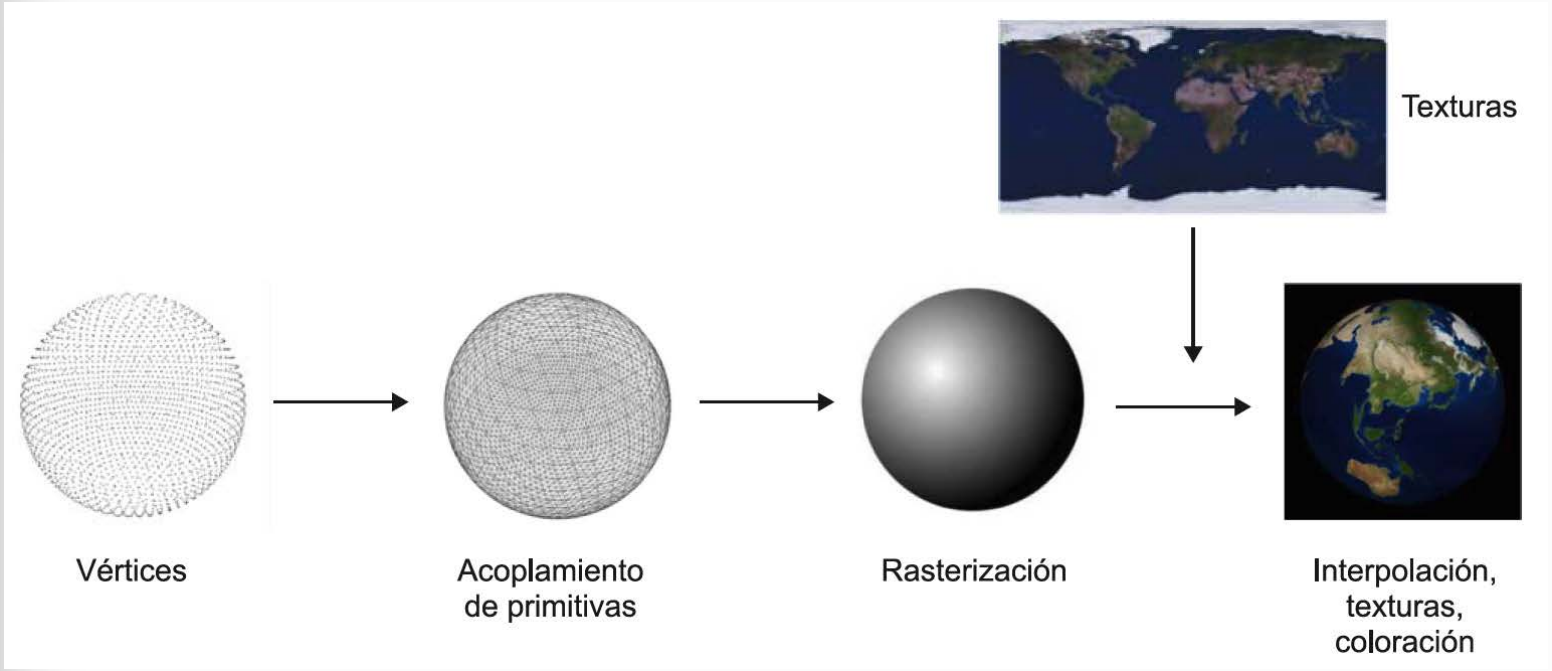
3ª etapa: En la etapa de aplicación de texturas y coloreado, el conjunto de fragmentos es procesado mediante operaciones matemáticas de aplicación de texturas y de determinación del color final de cada fragmento.

Pipeline Básico



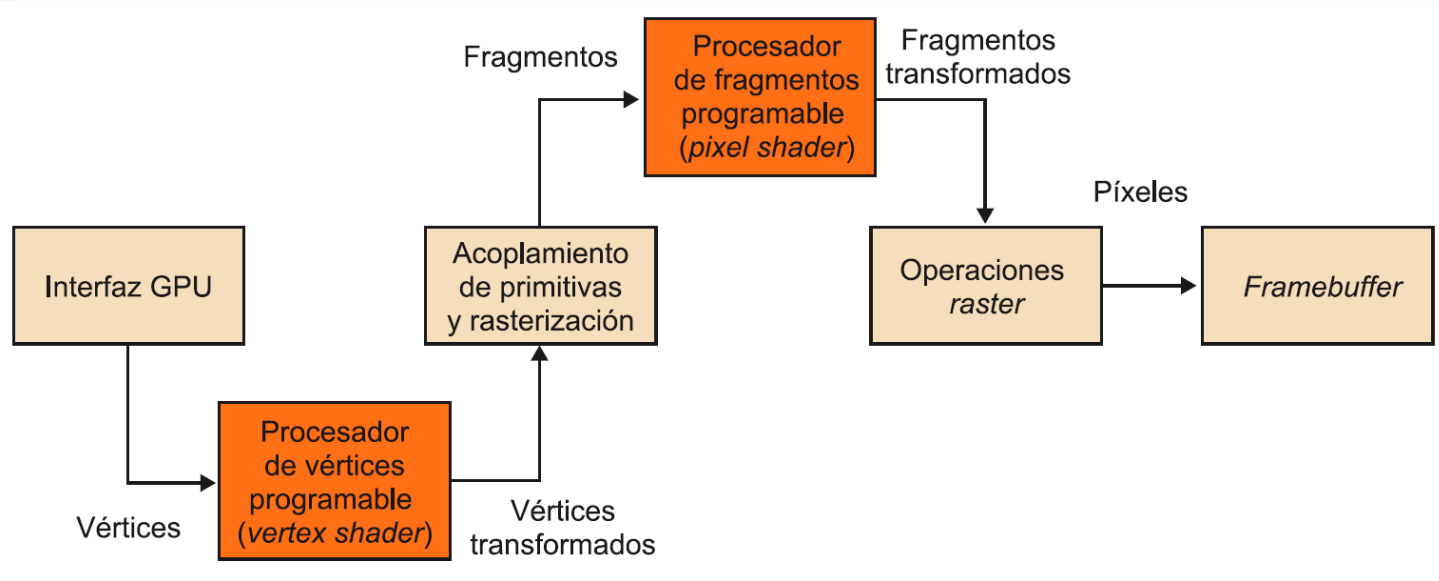
4ª etapa: Se ejecutan operaciones llamadas *raster*, que se encargan de analizar los fragmentos mediante un conjunto de tests. Estos tests determinan los valores finales que tomarán los píxeles.

Pipeline Básico



Etapas Programables del Pipeline

Las fases que permiten la programación, son las de transformación de vértices y transformación de fragmentos.



Estas dos fases permiten programación mediante el procesador de vértices (vertex shader) y el procesador de fragmentos (pixel shader).

Procesador de Vértices

El primer paso consiste en la carga de los atributos asociados a los vértices por analizar. Estos se pueden cargar en registros internos del procesador de vértices.

- Registros de atributos de vértices, solo de lectura, con información relativa a cada uno de los vértices.
- Registros temporales, de lectura/escritura, utilizados en cálculos provisionales.
- Registros de salida, donde se almacenan los nuevos atributos de los vértices transformados que, a continuación, pasarán al procesador de fragmentos.

Una vez cargado los atributos, el procesador ejecuta de manera secuencial cada una de las instrucciones.

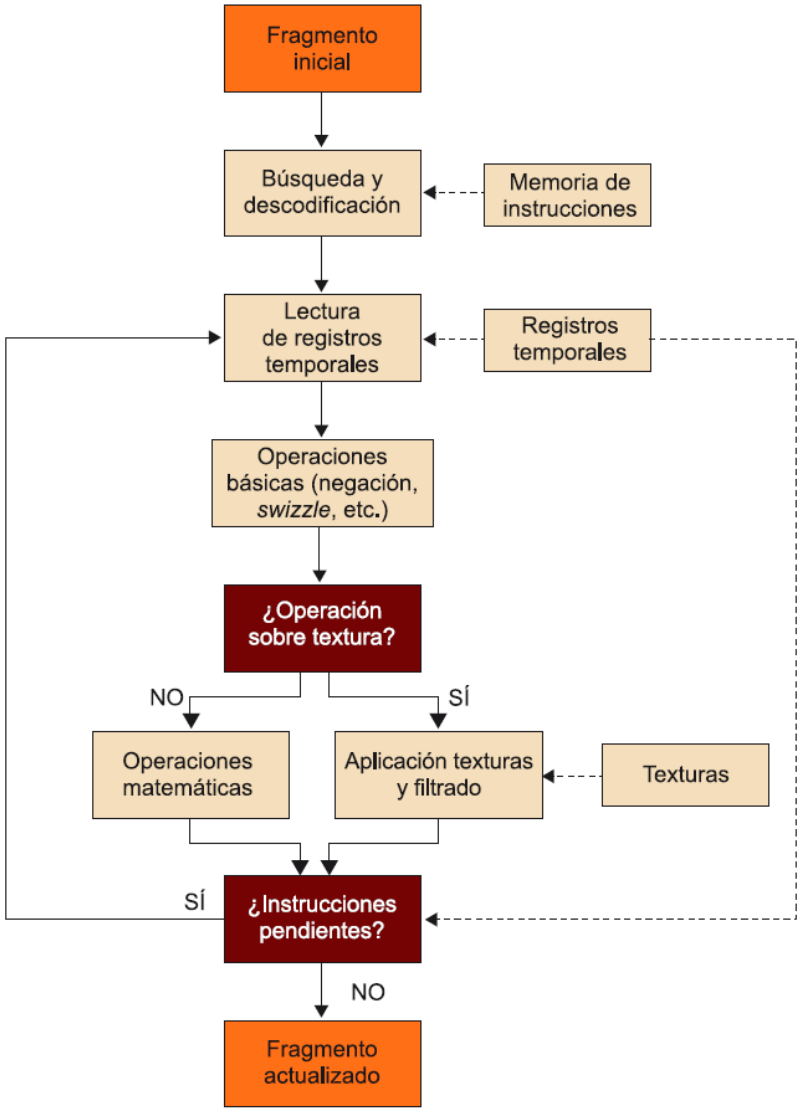
Procesador de Vértices

Las operaciones que los procesadores de vértices tienen que poder hacer son:

1. Operaciones matemáticas en coma flotante sobre vectores.
2. Navegación de vectores e indexación.
3. Exponenciales, logarítmicas y trigonométricas.

- Uno de los principales inconvenientes de estos tipos de procesadores es el limitado número de instrucciones que incorpora.
- Los procesadores gráficos modernos soportan también operaciones de control de flujo que permiten la implementación de bucles y construcciones condicionales.

Procesador de Fragmentos



- Los procesadores de fragmentos programables requieren muchas de las operaciones matemáticas que exigen los procesadores de vértice, pero incorporan también operaciones sobre texturas.
- Este tipo de procesadores solo funciona en modalidad SIMD sobre los elementos de entrada.

funcionamiento esquemático de uno de estos procesadores.

Interfaces de Programación del Pipeline Gráfico

Para programar el pipeline de un procesador gráfico de forma eficaz, los programadores necesitan bibliotecas gráficas que ofrezcan las funcionalidades básicas.

Existen muchas, pero en nuestro caso solo estudiaremos:

- OpenCL
- Direct3D

OpenCL

Características

- Está diseñada de manera completamente independiente al hardware.
- No incluye instrucciones para la gestión de ventanas
- Operaciones:
  - Modelar figuras a partir de primitivas básicas.
  - Situar los objetos en el espacio tridimensional.
  - Calcular el color de todos los objetos.
  - Convertir la descripción matemática de los objetos y la información de color asociada a un conjunto de píxeles

OpenCL

Características

- Operaciones complejas como:
  - Eliminación de partes de objetos que quedan ocultas tras otros objetos.

  - Como gran desventaja, un programa en OpenCL puede llegar a ser muy complejo.

Direct3D

- La alternativa a OpenCL es Direct3D
- Direct3D ofrece un conjunto de servicios gráficos 3D en tiempo real, que se encarga de:
  - ❖ Renderización.
  - ❖ Acceso transparente al dispositivo.

Direct3D

Características

- Es completamente escalable.
- Expone las capacidades mas complejas de la GPU.
  (efectos atmosféricos, etc)
- Integrado con DirectX.
- Nivel de complejidad elevado.
- Formas geométricas se definen como secuencias de triángulos.

Utilización del Pipeline Gráfico para Computación General

Podemos utilizar los procesadores gráficos para hacer computación general.

Para lograrlo la mejor opción es utilizar los procesadores de fragmentos por los motivos siguientes:

1. En un procesador gráfico hay más procesadores de fragmentos que procesadores de vértices.
2. Permite almacenar los resultados directamente en memoria. (en el caso de los procesadores de vértice, tiene que realizar varias etapas intermedias).

Utilización del Pipeline Gráfico para Computación General

La única forma que tienen los procesadores de fragmentos de acceder a la memoria es mediante las texturas. Cuando el procesador gráfico genera una imagen, puede haber dos opciones:

- Escribir la imagen en memoria, de forma que la imagen se muestre por pantalla.
- Escribir la imagen en la memoria de textura. Es la forma fácil de realimentación entre los datos de salida de un proceso con la entrada del proceso posterior sin pasar por la memoria.

- La desventaja de estos procesadores se encuentra en que solo pueden hacer una escritura al finalizar la ejecución.

Arquitecturas de Procesamiento Gráfico

Visión histórica

Características básicas de los sistemas basados en GPU

Arquitectura NVIDIA GeForce

Contexto y Motivación

Los métodos más relevantes para mejorar el rendimiento de los computadores ha sido el aumento de la velocidad del reloj del procesador. Sin embargo fabricantes se vieron obligados a buscar alternativa debido a:

- Limitaciones fundamentales en la fabricación de circuitos integrados.
- Restricciones de energía y calor.
- Limitaciones en el nivel de paralelismo a nivel de instrucción.

Contexto y Motivación

- La solución que la industria adoptó fue el desarrollo de procesadores con múltiples núcleos que se centran en el rendimiento de ejecución de aplicaciones paralelas.

- La demanda de gran potencia de cálculo para generar gráficos tridimensionales ha provocado una rápida evolución de las GPUs.

Visión Histórica

- En la década de 1980 se empezaron a popularizar los dispositivos **aceleradores** para **2D** orientados a computadoras personales. Estos aceleradores apoyaban al sistema operativo.

- Silicon Graphics popularizó el uso de tecnologías para 3D. En 1992 abrió la interfaz de programación por medio de la biblioteca **OpenCL** con el objetivo de que se convirtiera en el estándar para escribir aplicaciones gráficas **3D**.

- A mediado de los 90's empresas como **Nvidia**, ATI Technologies y 3dfx Interactive empezaron a comercializar aceleradores gráficos debido al aumento de la demanda de los usuarios.

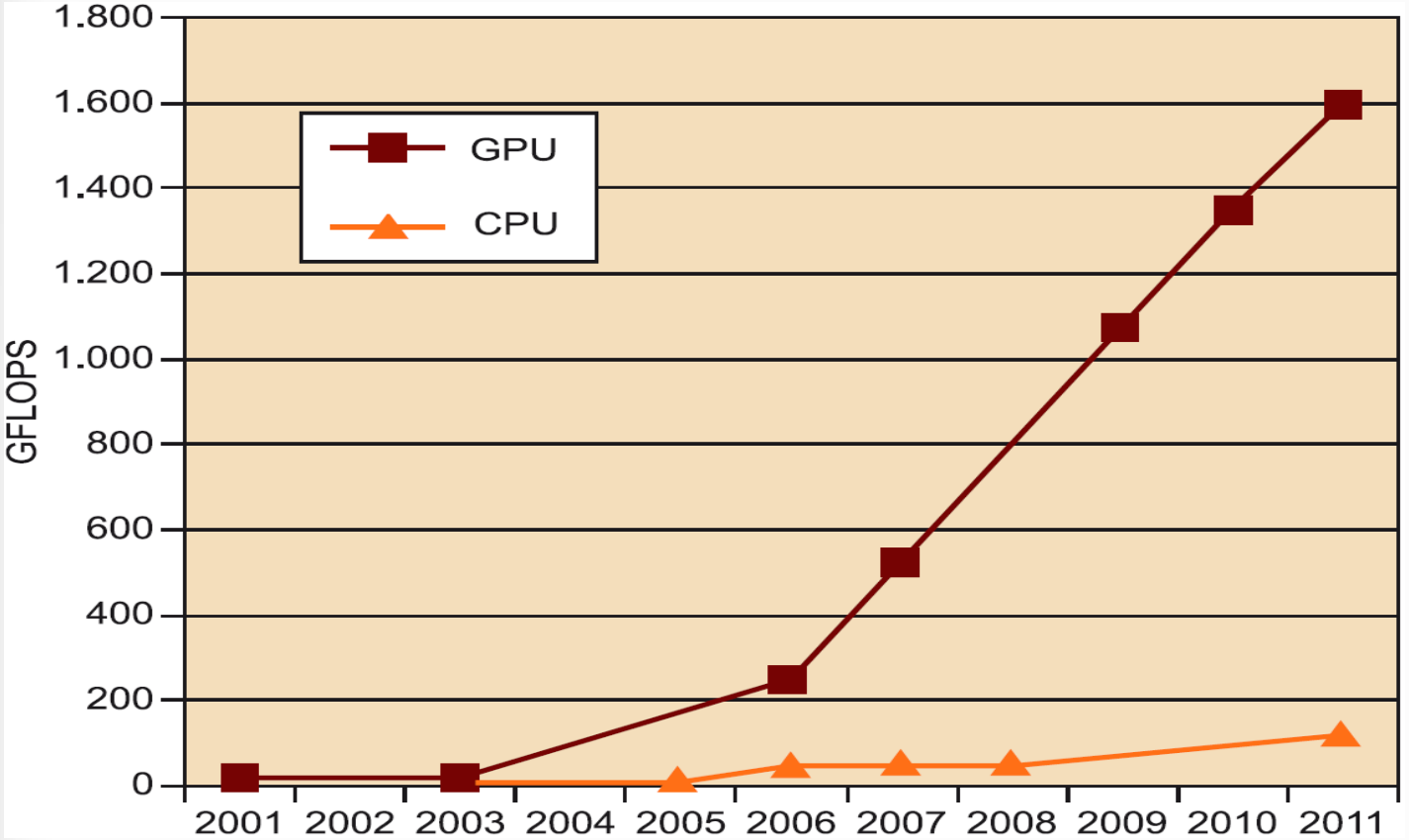
Visión Histórica

- La aparición de la **Nvidia GeForce 256** (en 1999) representó un impulso importante para abrir todavía más el mercado del hardware gráfico. Marcó el comienzo de implementar cada vez más etapas del pipeline gráfico en procesadores gráficos.

- En 2001 con la aparición de **la GeForce 3** de Nvidia (representó el cambio más significativo en la computación paralela). Disponía etapas programables para el procesamiento de vértices como para el procesamiento de fragmentos, danto al programador control sobre los cálculos a realizar sobre las GPU.

Visión Histórica

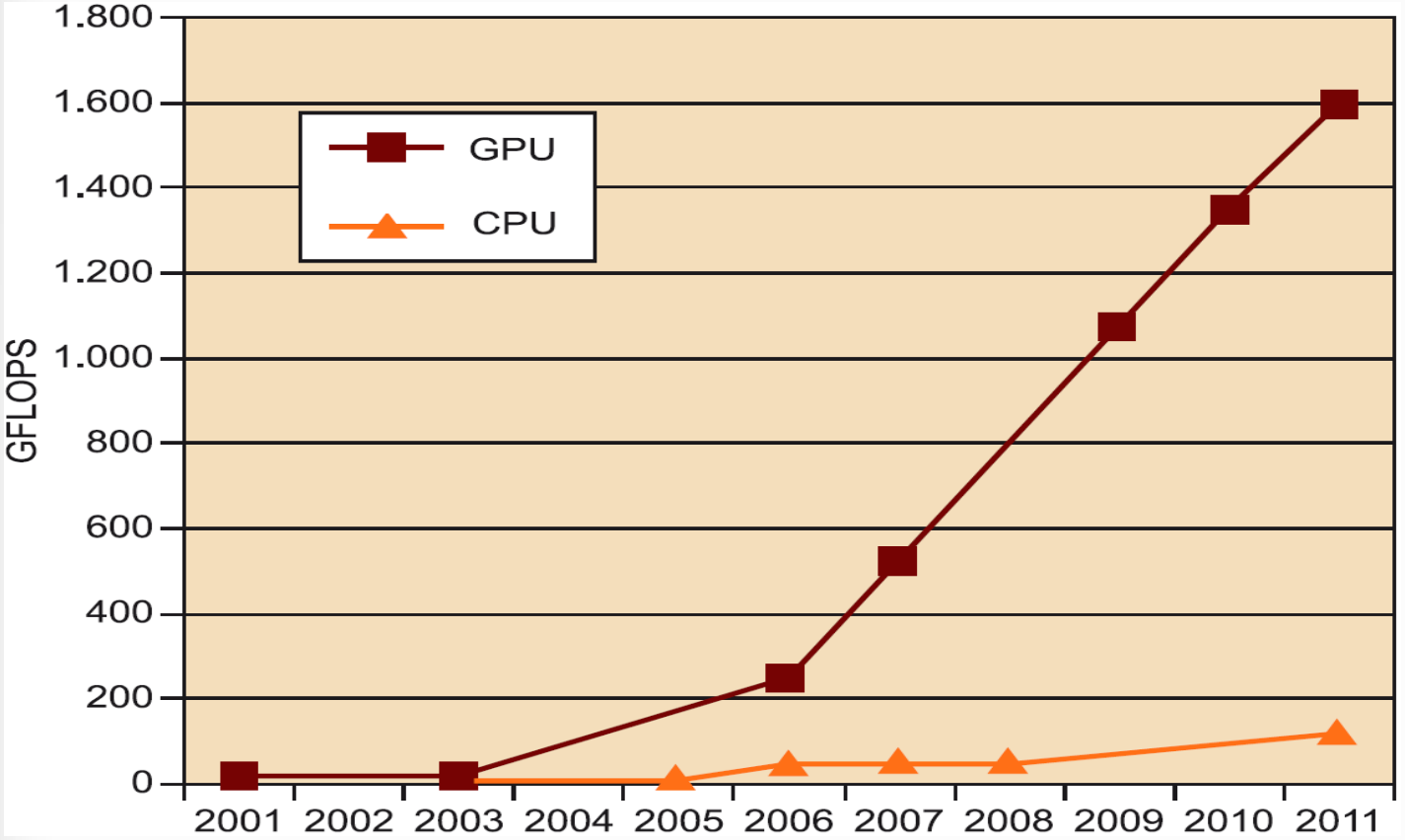
Las primeras generaciones de GPU tenían una cantidad de núcleos bastante reducida.



Diferencia del aumento de núcleos de las GPU y las CPU. Las GPU están pensadas para explotar el paralelismo a nivel de datos con el paralelismo masivo y una lógica bastante simple

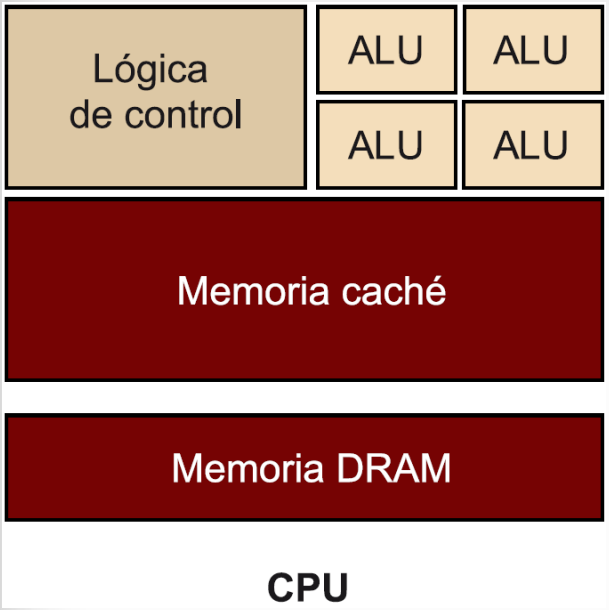
Visión Histórica

Las primeras generaciones de GPU tenían una cantidad de núcleos bastante reducida.



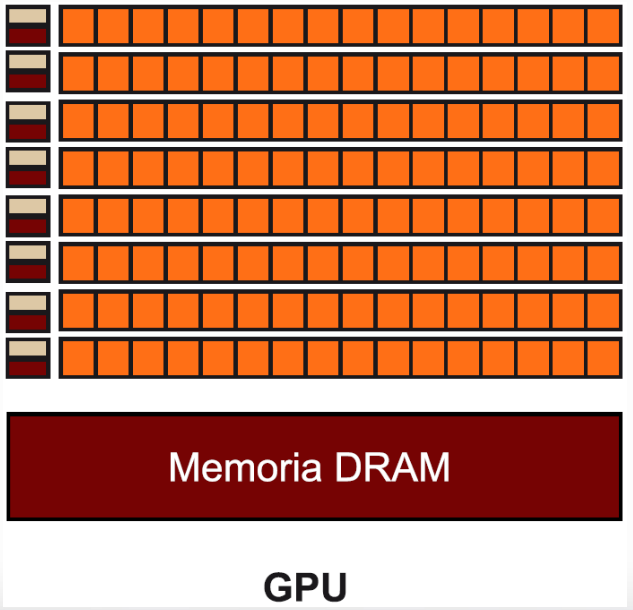
¡¡ La GeForce GTX TITAN X tiene 3072 núcleos !!

Características Básicas de los Sistemas Basados en GPU



- Para optimizar el rendimiento del cálculo en coma flotante, se ha optado por explotar un número masivo de flujos de ejecución.

- Esta estrategia consiste en explotar estos flujos de tal manera que, mientras que unos están en espera para el acceso a memoria, el resto pueda seguir ejecutando una tarea pendiente.



Características Básicas de los Sistemas Basados en GPU

Características:

- Siguen el modelo SIMD: Todos los núcleos ejecutan a la vez una misma instrucción; por lo tanto, solo se necesita descodificar la instrucción una única vez para todos los núcleos.
- La velocidad de ejecución se basa en la explotación de la localidad de los datos, tanto la localidad temporal como la localidad espacial.
- La memoria de una GPU se organiza en varios tipos de memoria (local, global, constante y textura), que tienen diferentes tamaños, tiempos de acceso y modos de acceso.

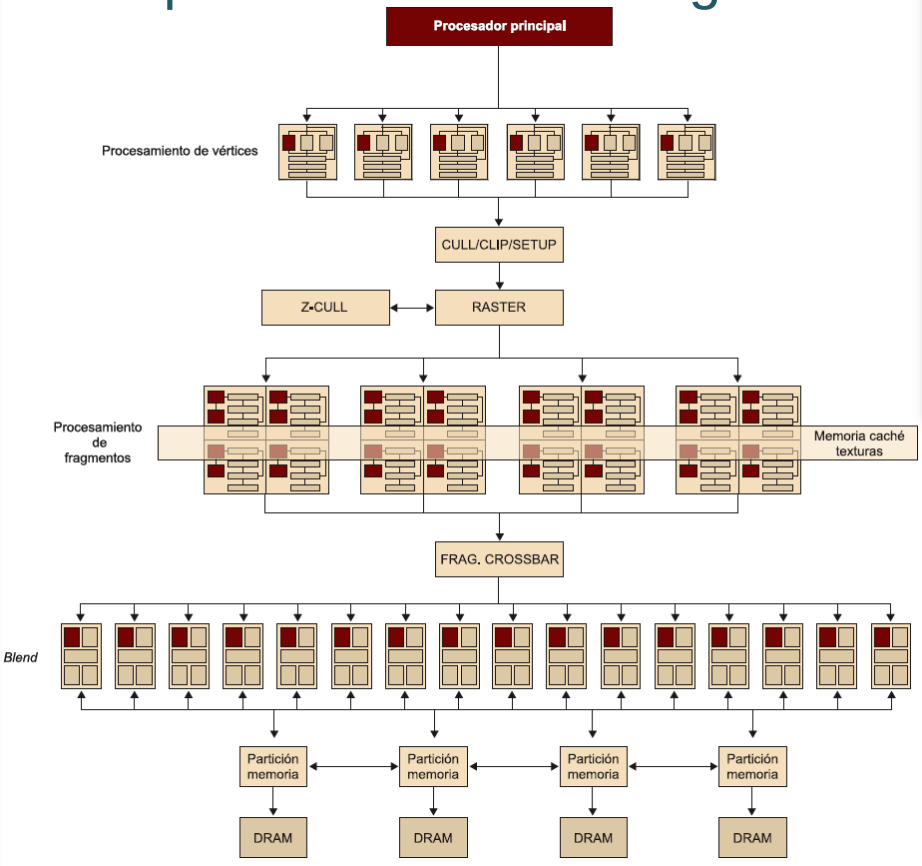
Características Básicas de los Sistemas Basados en GPU

Características:

- El ancho de banda de la memoria es mayor.
  - La GPU son sistemas externos a las CPU. La comunicación entre ellos se lleva a cabo por medio de un puerto dedicado. En la actualidad, el PCI Express o el AGP.
  - Una GPU no puede acceder directamente a la memoria principal y que una CPU no puede acceder directamente a la memoria de una GPU. Por lo tanto, habrá que copiar los datos entre CPU y GPU de manera explícita.

Arquitectura Nvidia GeForce

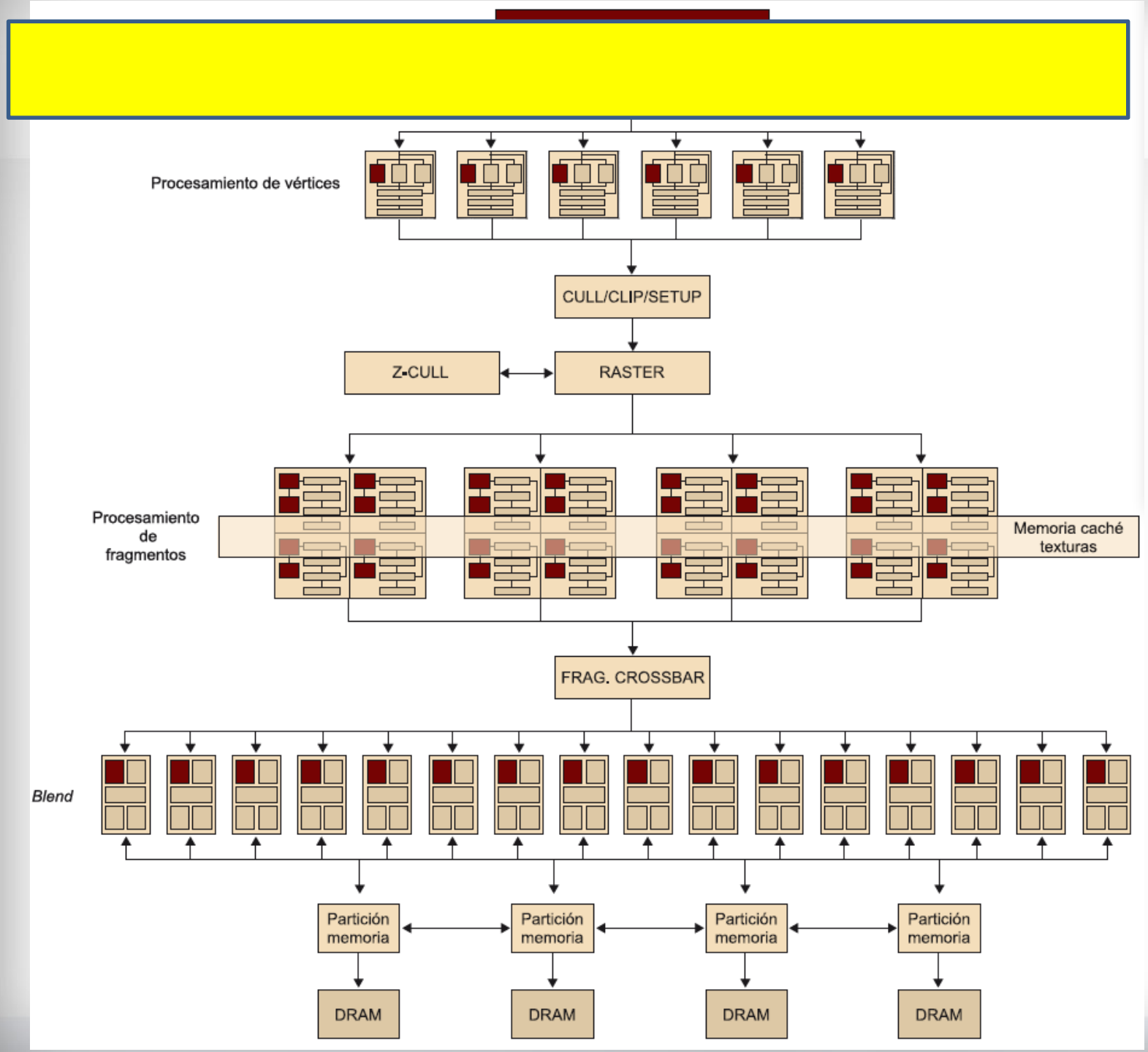
Vamos a utilizar la Nvidia GeForce 6 como caso de uso de GPU pensada para tratamiento de gráficos.



Modo esquemático los bloques principales que forman la arquitectura GeForce6 de Nvidia.

Arquitecturas de Procesamiento Gráfico

Arquitectura Nvidia GeForce



Arquitectura Nvidia GeForce

- La CPU envía a la unidad gráfica tres tipos de datos:
  - Instrucciones.
  - Texturas.
  - Vértice.
- Los procesadores de vértice se encargan de las transformaciones sobre cada uno de los vértices de entrada.
- La GeForce 6 permitía que un programa ejecutado en el procesador de vértices fuera capaz de consultar datos de textura.

Arquitectura Nvidia GeForce

- El número de procesadores de vértices disponibles es variable en función del modelo de procesador.

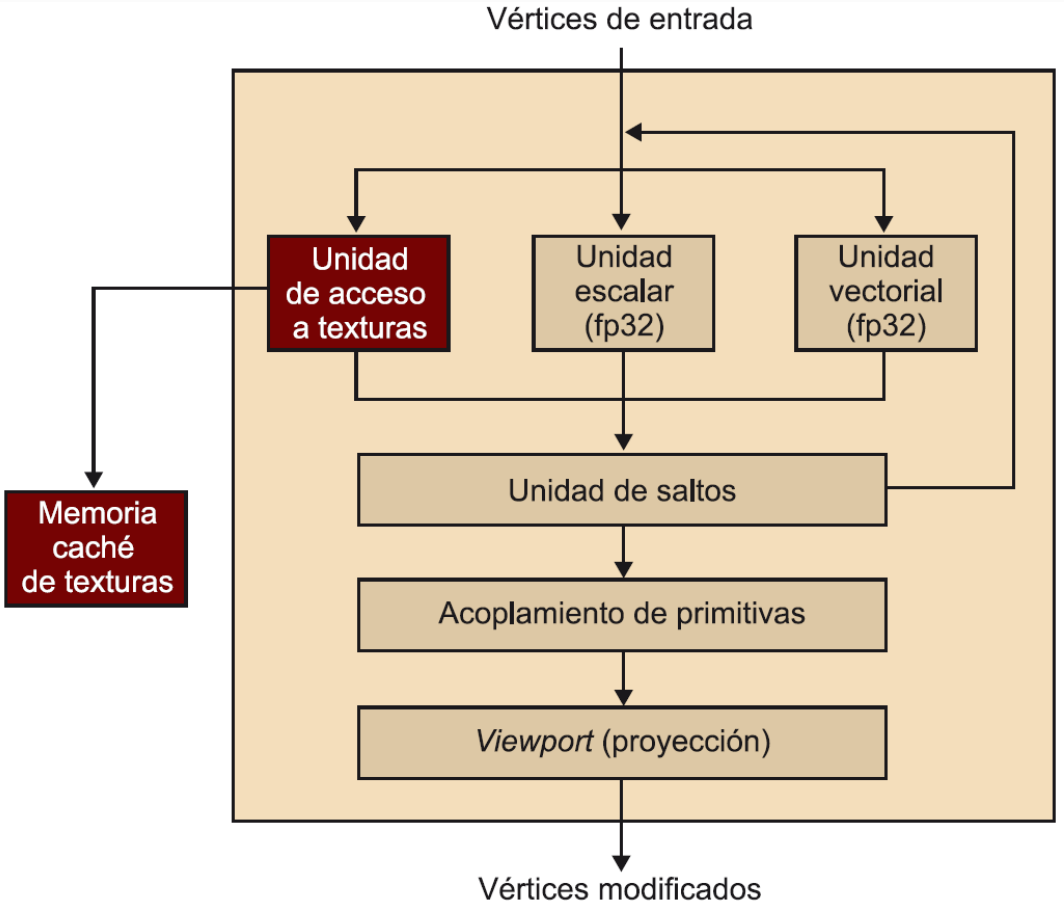
  (entre 2 y 16)

- Posee dos memorias caché:
  - Memoria cache de texturas
  - Memoria de vértices

    (almacena datos relativos a vértices antes y después de haber sido procesados).

Arquitectura Nvidia GeForce

- Todas las operaciones se llevan a cabo con una precisión de 32 bits en coma flotante



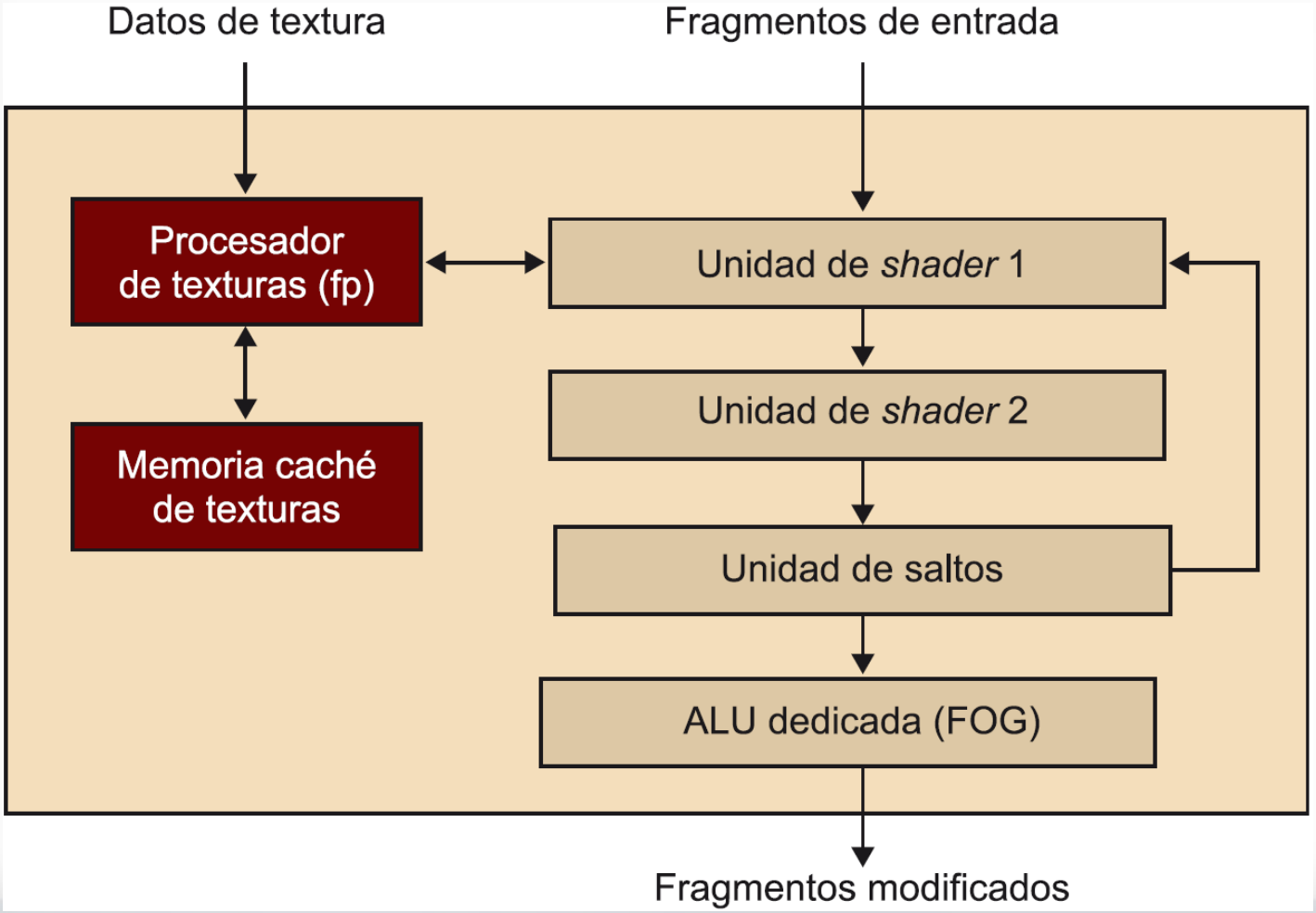
- Los procesadores de vértices, tienen conexión a la memoria caché de texturas

Arquitectura Nvidia GeForce

- Los procesadores fragmentados se divide en 2:
  - La unidad de textura
    Está dedicada al trabajo con texturas.
  - La unidad de procesamiento de fragmentos
    Opera con ayuda de la unidad de texturas, sobre cada uno de los fragmentos de entrada.
    - Las dos unidades operan de forma simultánea.

Arquitectura Nvidia GeForce

Arquitectura de los procesadores de fragmentos típicos de la serie GeForce 6 de Nvidia.



Arquitectura Nvidia GeForce

Características:

- Los datos de textura se pueden almacenar en memorias caché para aumentar el rendimiento.
- El procesador de fragmentos utiliza la unidad de texturas para cargar datos desde la memoria.
- Poseen dos unidades de procesamiento que operan con una precisión de 32 bits.

Arquitectura Nvidia GeForce

Características:

- La unidad de texturas soporta gran cantidad de formatos de datos, pero son devueltos al procesador de fragmentos en formato fp32 o fp16.
- La memoria del sistema se divide en cuatro particiones independientes.
- Todos los datos procesados por el pipeline gráfico se almacenan en memoria DRAM,
- Las texturas y los datos de entrada (vértices) se pueden almacenar tanto en la memoria DRAM como en la memoria principal del sistema.

Arquitectura Unificada

- Estas arquitecturas han evolucionado a las arquitecturas GPU's Unificadas.
- Optimizan la cargas de trabajo no balanceados cuando no hay un reparto proporcionado en la carga de trabajo en los proc. de vértices y de fragmentos.
- En estas arquitecturas hay una unión entre los procesadores de vértices y fragmentos, de forma que cualquier procesador que los forma (***Steam Processor)*** podría trabajar a nivel de vértice como de contexto
- Implica un cambio importante en el concepto del pipeline grafico. Permite asignar las tareas a cada procesador.
- Este tipo de arquitecturas ofrece un potencial mucho mayor para hacer computación de propósito genera

Arquitectura Orientadas a Computación de Propósito General sobre GPU **(GPGPU)**

GPGPU

Modelos de programación para GPGPU

- **CUDA**
  - Arquitectura compatible con CUDA
  - Entorno de programación
  - Modelo de memoria
  - Definición de kernel
  - Organización de flujos
- **OpenCL**
  - Modelo de paralelismo a nivel de datos
  - Arquitectura conceptual
  - Modelo de memoria
  - Gestión de kernels y dispositivos

Arquitectura Orientadas a Computación de Propósito General sobre GPU **(GPGPU)**

Los procesadores gráficos pueden ser utilizados para ejecutar aplicaciones que tradicionalmente que son ejecutadas en CPU, ya que las GPU poseen:

- Nivel de paralelismo masivo.
- Alto rendimiento que proporcionan las plataformas GPU

Las aplicaciones que pueden aprovechar mejor las capacidades de las GPU son aquellas que cumplen las dos condiciones siguientes:

- Trabajan sobre vectores de datos grandes
- Tienen un paralelismo de grano fino tipo SIMD.

Ejemplos: álgebra lineal, algoritmos de ordenación, etc.

GPGPU

Inconveniente a la hora de trabajar con GPU:

Dificultad para el programador a la hora de transformar programas diseñados para CPU tradicionales en programas que puedan ser ejecutados de manera eficiente en una GPU.

Se han desarrollado modelos de programas que proporcionan al programador un nivel de abstracción más cercano a la programación.

Podemos destacar entre ellos:

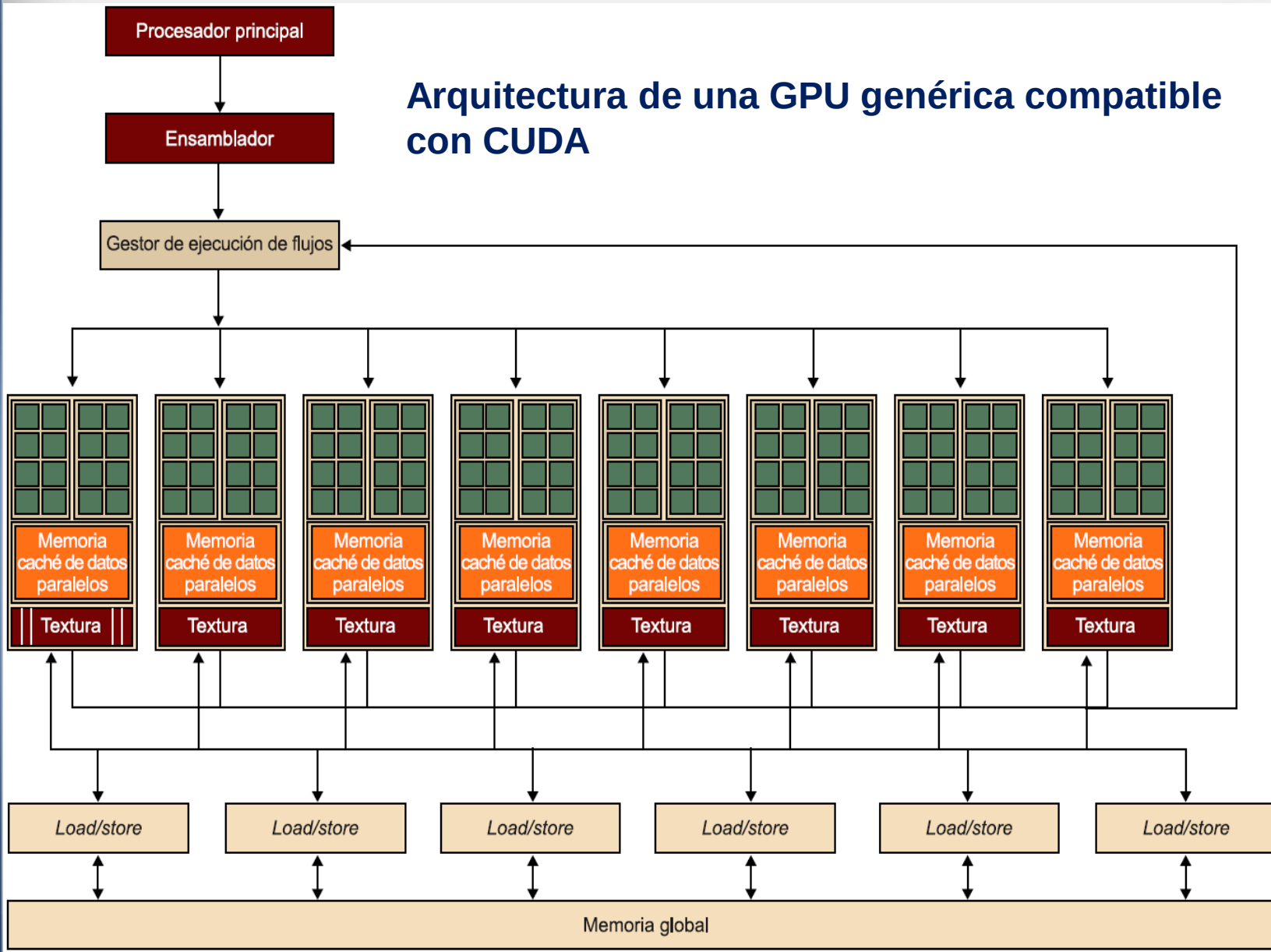
- **CUDA**
- **OpenCL**

GPGPU

- Principales fundamentos de la programación GPGPU:
  - Utilización del procesador de fragmentos como unidad de cómputo.
  - La entrada/salida es limitada: lecturas abiertas, restricciones en las escrituras.

CUDA – Arquitecturas Compatibles

Arquitectura de una GPU genérica compatible con CUDA



CUDA – Arquitecturas Compatibles

Descripción de la arquitectura:

- Está organizada en base a multiprocesadores ***streaming multiprocessors (SM).***
- Dos SM forman un bloque.
- Cada SM tiene un número de ***streaming processors (SP)*** que comparten lógica de control y una memoria caché de instrucciones.
- Poseen memorias DRAM de tipo GDDR *(graphics double data rate).*
- Los SP son masivamente paralelos, por lo que es muy importante intentar aprovechar este nivel de paralelismo cuando se desarrollen aplicaciones para GPU.

CUDA – Entorno de Programación

- CUDA pertenece al modelo SIMD, pensado para el paralelismo de datos. Permite que un conjunto de operaciones aritméticas se puedan ejecutar sobre un conjunto de datos de manera simultánea.

- Un programa en CUDA esta formado por fases/partes
  - Las que se ejecutan el la CPU
    (partes con poco paralelismo)
  - Las que se ejecutan en la GPU
    (partes del código con mayor paralelismo a nivel de datos)

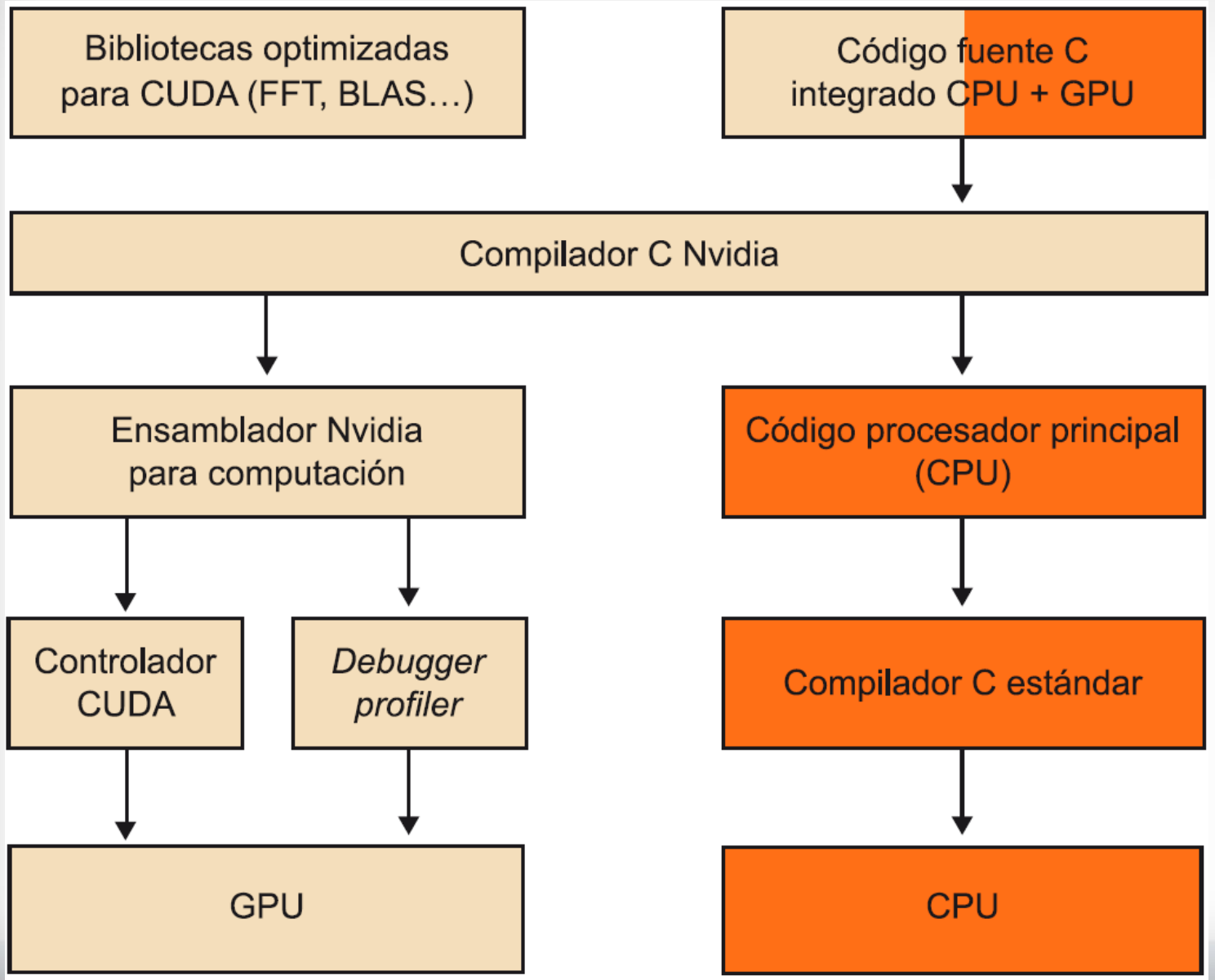
CUDA – Entorno de Programación

- El compilador de NVIDIA (nvcc) se encarga de proporcionar la parte del código correspondiente a la CPU y al dispositivo durante el proceso de compilación.
- Tanto el código del procesador como el código del dispositivo es código ANSI C.

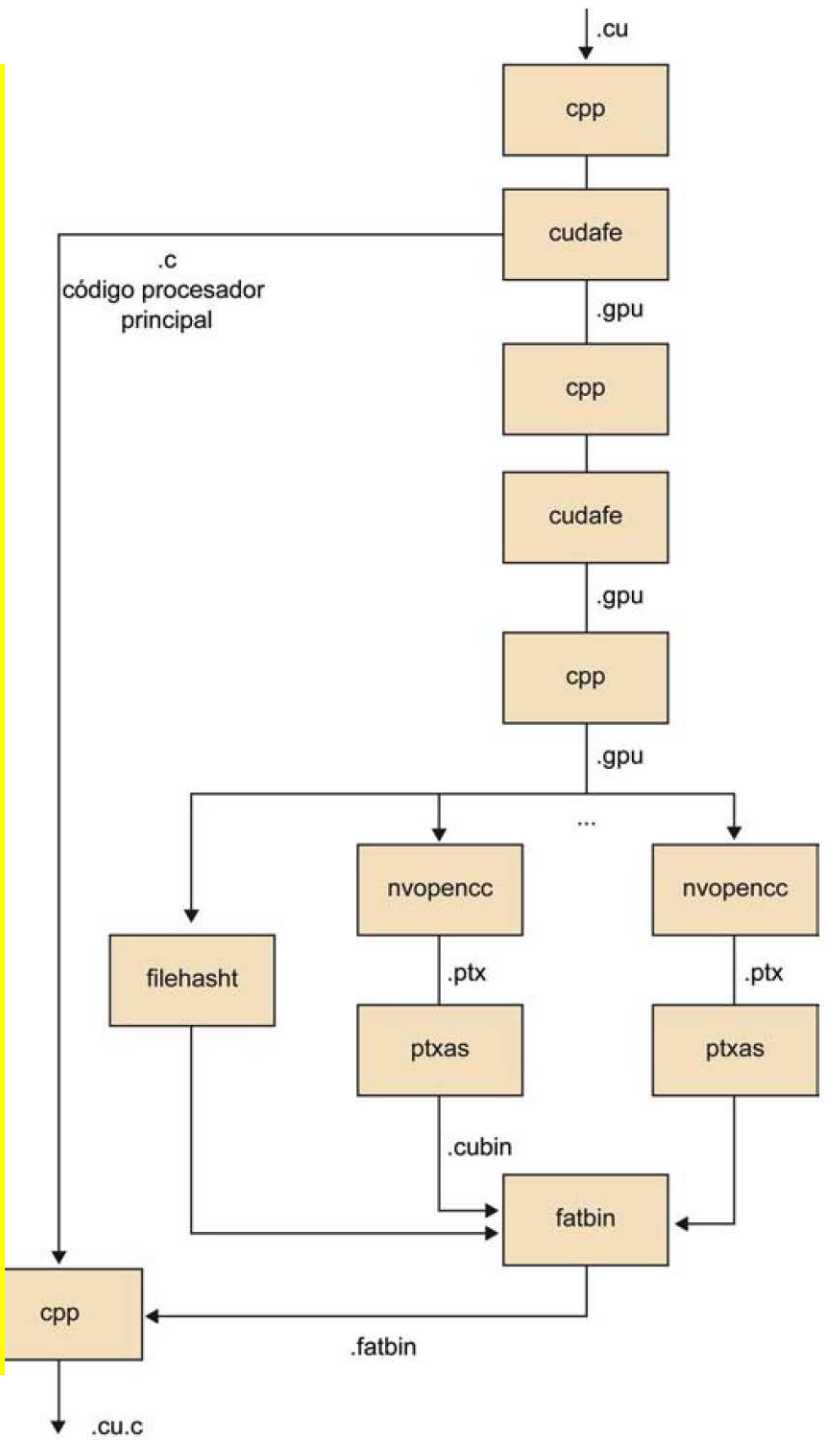
  Para hacer uso de la potencia del paralelismo se han implementado extensiones KERNEL que habrá que incluir en el código para poder definir las funciones que tratan los datos en paralelo.

- También se pueden ejecutar los kernels en una CPU convencional mediante herramientas de emulación que incorpora CUDA

CUDA – Entorno de Programación



PASOS EN LA COMPILACIÓN DE UN CODIGO CUDA



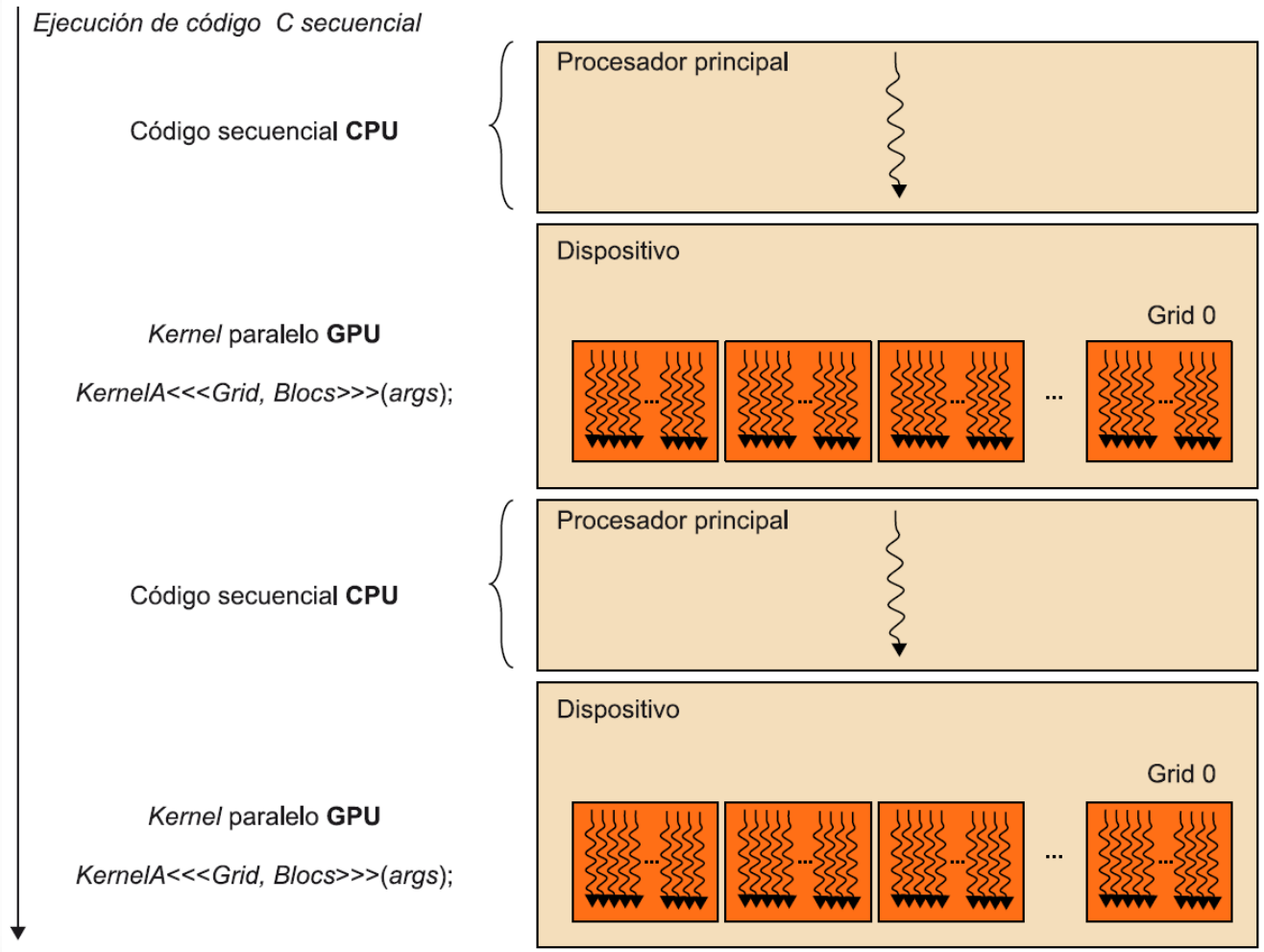
CUDA – Entorno de Programación

- Para conseguir explotar el paralelismo de datos, los kernels tienen que generar una cantidad de flujos de ejecución.
- Los flujos de CUDA son mucho más ligeros que los flujos de CPU, de hecho para generar dichos flujos necesitamos muchos menos ciclos de reloj.

1. Comienza la ejecución en la CPU.
2. Cuando se invoca un kernel, la GPU continua con ejecución.
3. En la GPU se genera un conjunto elevado de flujos llamado grid.
4. Finaliza la ejecución cuando todos los kernel finalizan.
5. La ejecución del programa continua en la CPU hasta que se invoque un nuevo kernel.

CUDA – Entorno de Programación

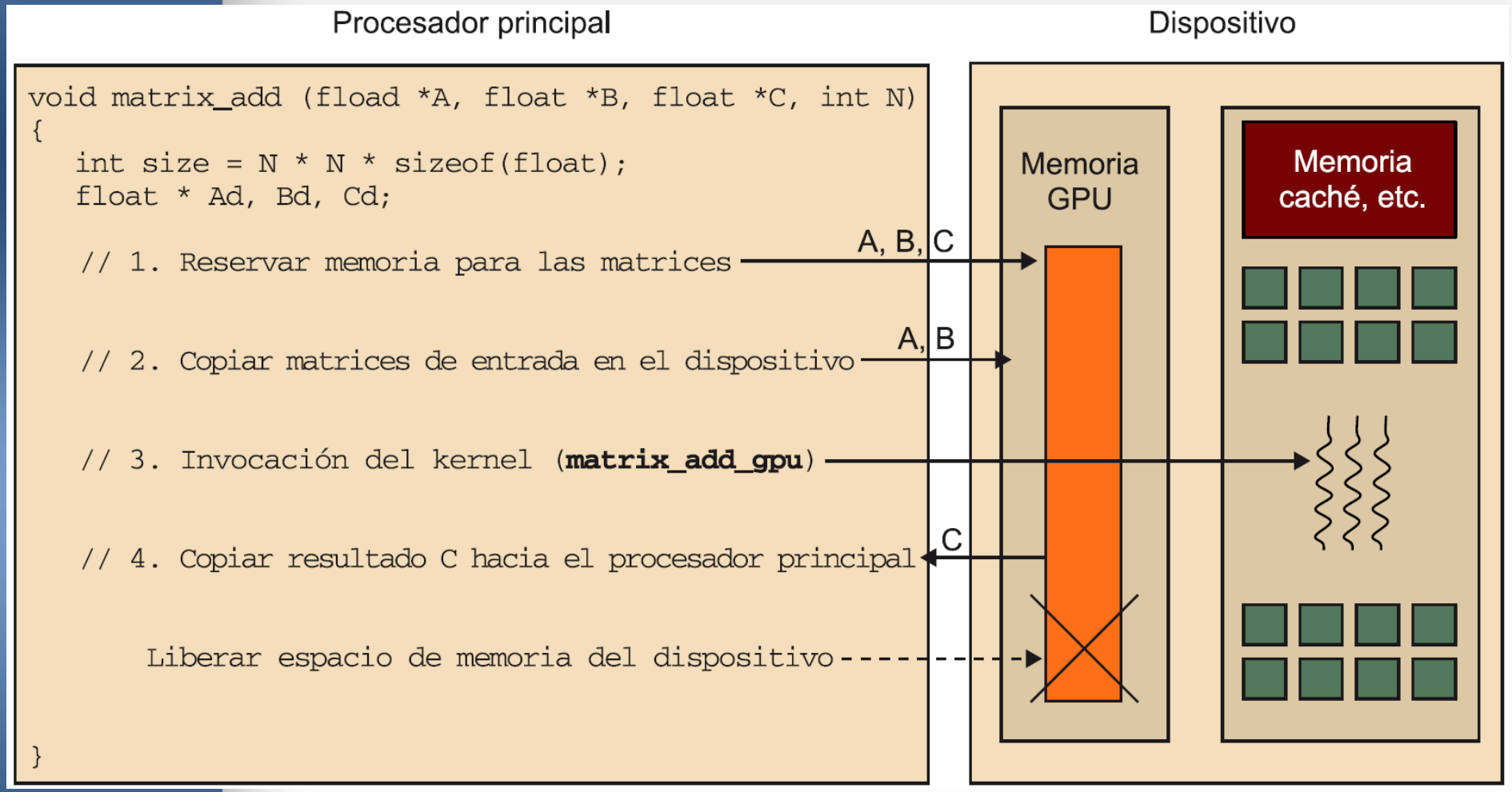
Etapas en la ejecución de un programa típico CUDA.



CUDA – Modelo de Memoria

- La memoria de la CPU y GPU están en espacios de memoria separados.
- Pasos para ejecutar un kernel en una GPU:
  - Paso 1: Reservar memoria en el dispositivo.
  - Paso 2: Transferir los datos desde la memoria principal hasta la memoria de la GPU.
  - Paso 3: Invocar la ejecución del kernel.
  - Paso 4: Transferir los datos resultantes desde la memoria de la GPU hasta la memoria principal.

CUDA – Modelo de Memoria

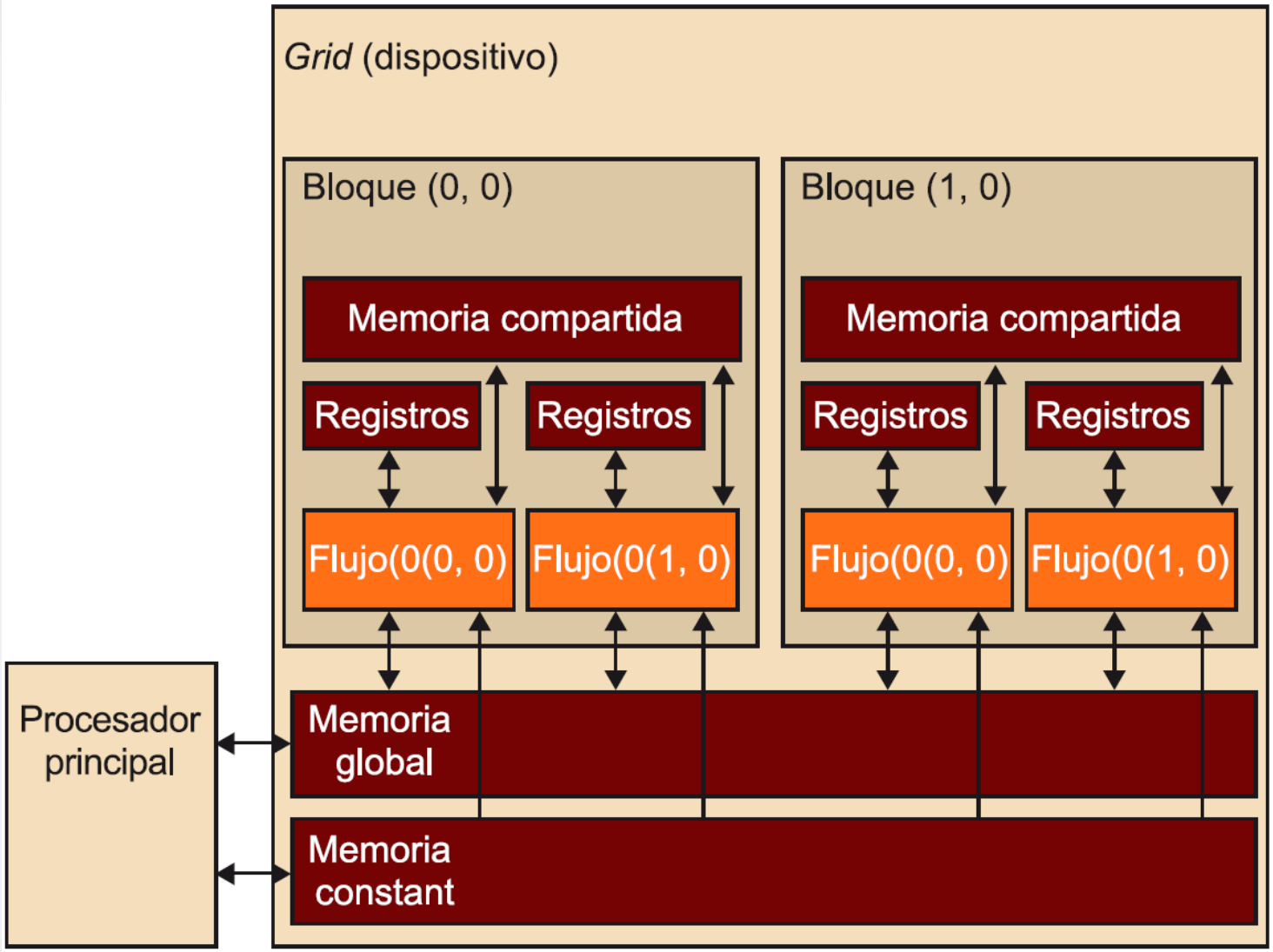


Esquema de los pasos para la ejecución de un kernel en una GPU

CUDA – Modelo de Memoria

- Por parte de la GPU, se pueden acceder a estas memorias con los siguientes modos de acceso:
  - De lectura/escritura a la memoria global, por grid.
  - Solo de lectura a la memoria constante, por grid.
  - De lectura/escritura a la memoria local, por flujo.
  - De lectura/escritura a los registros, por flujo.
  - De lectura/escritura a la memoria compartida, por bloque.

CUDA – Modelo de Memoria



CUDA – Modelo de Memoria

- **cudaMalloc**: Asigna un espacio de memoria global para un objeto. Puede ser llamada desde el código de la CPU. Posee dos parámetros:
  - **1er param**: Dirección del puntero hacia el objeto.
  - **2do param**: Tamaño del objeto en bytes.

- **cudaFree**: Libera el espacio de memoria del objeto.
  - **1er param**: Objeto del que se quiere liberar memoria.

Ejemplo:

```
float *Matriz;

    int medida = ANCHURA * LONGITUD * sizeof(float);
    cudaMalloc((void **) &Matriz, medida);
    ...
    cudaFree(Matriz);
```

CUDA – Modelo de Memoria

- **cudaMemcpy**: Permite transmitir los datos necesarios desde la CPU hasta la GPU y viceversa.
  - **1er param**: Puntero o dirección destino de la copia.
  - **2do param**: Puntero o dirección origen de la copia.
  - **3er param** : Nº de bytes que se quieren copiar.
  - **4to param** : Tipo de memoria involucrado. Puede ser:

  - **cudaMemcpyHostToHost**: De la memoria de la CPU hacia la memoria de la CPU.
  - **cudaMemcpyHostToDevice**: De la memoria de la CPU hacia la memoria de la GPU.
  - **cudaMemcpyDeviceToHost**: De la memoria de la GPU hacia la memoria de la CPU.
  - **cudaMemcpyDeviceToDevice**: De la memoria de la GPU hacia la memoria de la GPU.

CUDA – Modelo de Memoria

Ejemplo:

```
void matrix_add (fload *A, float *B, float *C, int N)
{
    int size = N * N * sizeof(float);
    float * Ad, Bd, Cd;

    // 1. Reservar memoria para las matrices
    cudaMalugar((void **) &Ad, size);
    cudaMalugar((void **) &Bd, size);
    cudaMalugar((void **) &Cd, size);

    // 2. Copiar matrices de entrada al dispositivo
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    ...
    // 4. Copiar resultado C hacia el procesador principal
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    ...
}
```

CUDA – Modelo de Memoria

- En CUDA se pueden declarar las variables de diversas formas, en función del tipo de memoria, ámbito o ciclo de vida:

Declaración de variables	Tipo de memoria	Ámbito	Ciclo de vida
Por defecto (diferentes vectores)	Registro	Flujo	*Kernel*
Vectores por defecto	Local	Flujo	*Kernel*
`__device__, __shared__, int SharedVar;`	Compartida	Bloque	*Kernel*
`__device__, int GlobalVar;`	Global	*Grid*	Aplicación
`__device__, __constant__, int ConstVar;`	Constante	*Grid*	Aplicación

CUDA – Definición de Kernel

El kernel es el código que se ejecuta en el dispositivo y es la función que ejecutan los diferentes flujos durante la fase paralela

CUDA sigue el modelo SPMD

(Single-Program Multiple-Data).

- SPMD es una técnica empleada para lograr paralelismo considerada como una subcategoría de MIMD

CUDA – Definición de Kernel

```
__global__ matrix_add_gpu (fload *A, float *B, float *C, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;

    if (i<N && j<N){
    C[index] = A[index] + B[index];
    }
}

int main(){
    dim3 dimBlock(blocksize, blocksize);
    dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
    matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

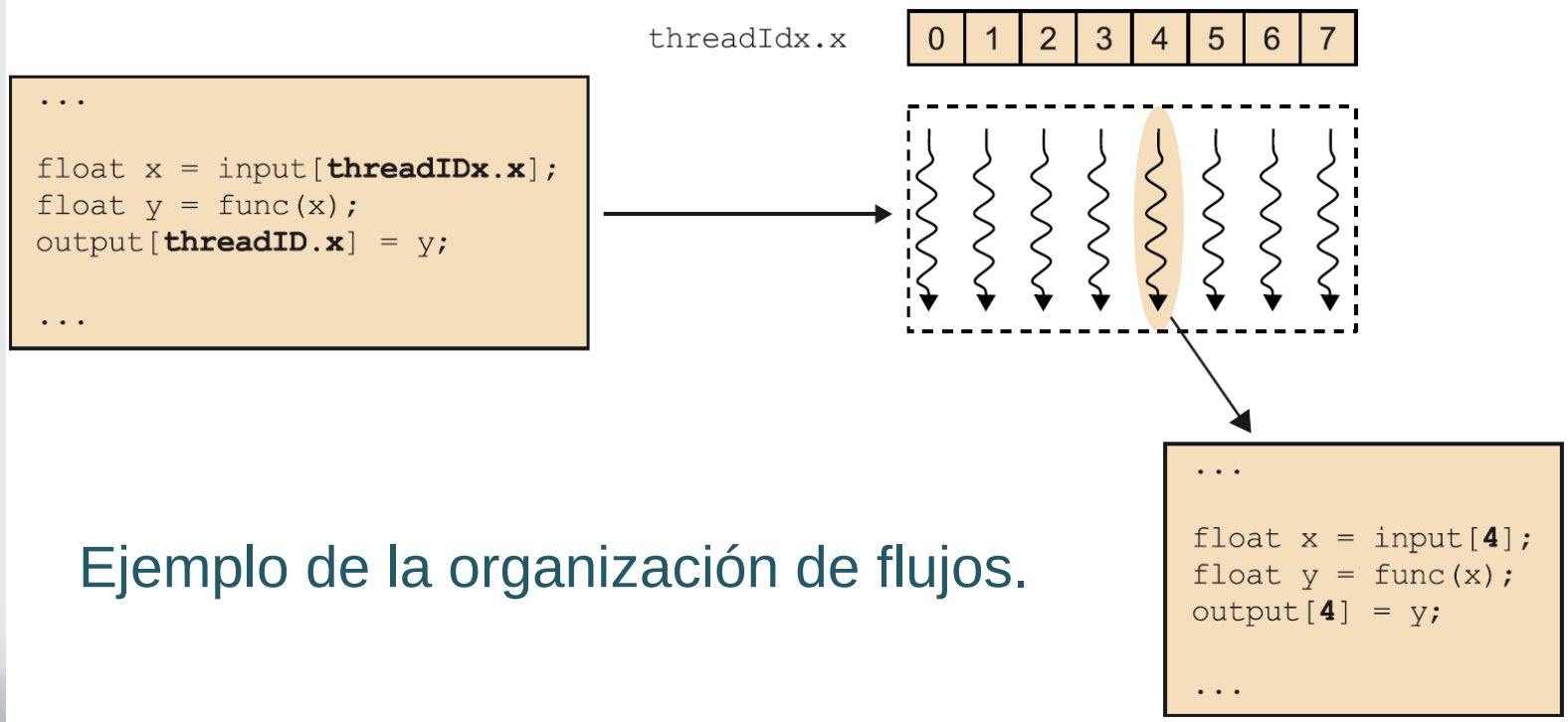
CUDA – Definición de Kernels

- La palabra clave ***__global__*** indica que esa función es un kernel. Existen dos palabras mas que se pueden utilizar antes de *__global__*:
  - ***__device__*** indica que la función es una función de la GPU. Esta función se ejecuta únicamente en un dispositivo CUDA y solo pueden ser llamadas desde un kernel. No pueden tener llamadas recursivas.
  - ***__host__*** indica que la función es una función de la CPU, se ejecuta en la CPU.

  - Si no se indica, por defecto las funciones en un proceso CUDA son funciones del procesador principal.
  - Ambas palabras pueden ser utilizadas en una misma función, lo que provocaría que el compilador generara dos versiones.

CUDA – Organización de Flujos

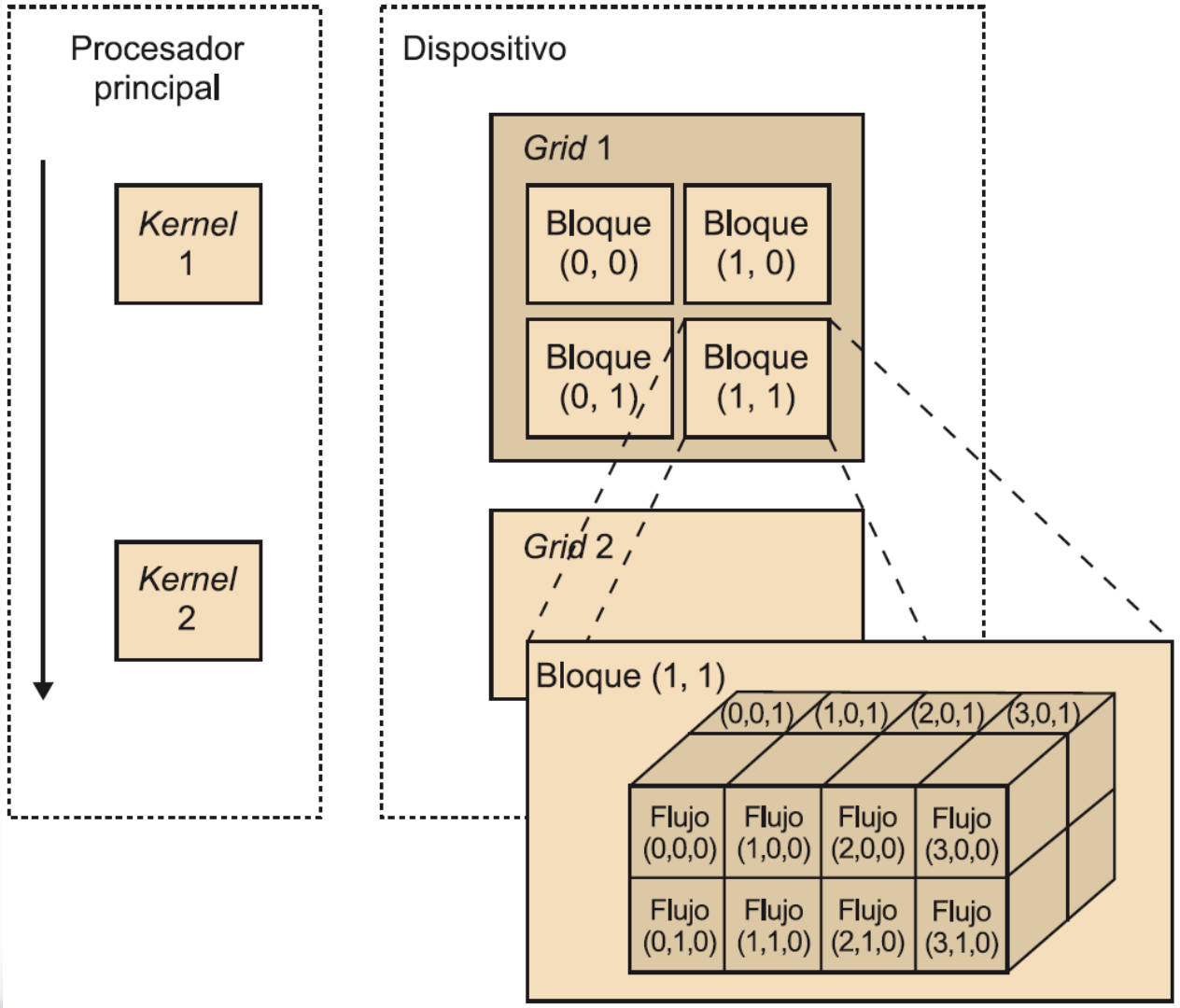
- El kernel se ejecuta mediante una serie de flujos, como todos los flujos ejecutan el mismo kernel se necesitan mecanismos para diferenciarlos y asignarles a cada flujo un conjunto de datos.
- Para lograr esto CUDA incorpora palabras claves para hacer referencia al índice(threadIdx.x, threadIDx.y ...).



Ejemplo de la organización de flujos.

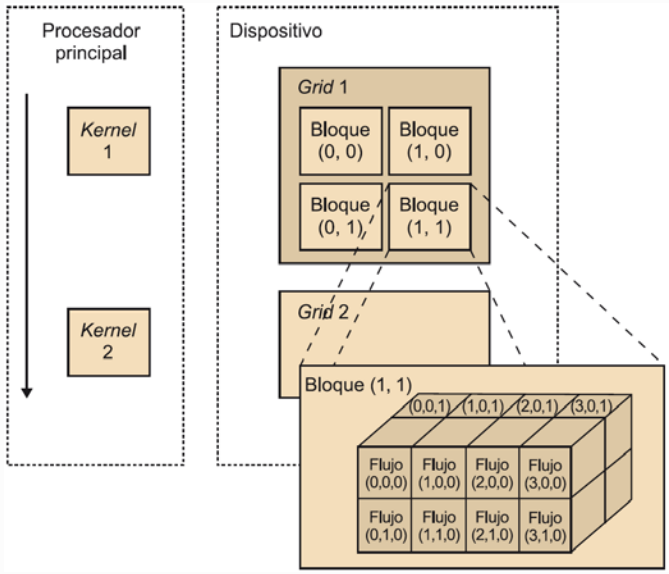
CUDA – Organización de Flujos

Niveles de jerarquía de los grids



CUDA – Organización de Flujos

- Cada grid tiene un número de bloques, a su vez cada bloque contiene un conjunto de flujos de ejecución.
- Nivel superior: consiste en uno o más bloques de flujos. Todos los bloques tienen un mismo número de flujos y deben estar organizados del mismo modo.
- Cada bloque de un grid tiene una coordenada única mediante las palabras clave blockIdx.x y blockIdx.y. Los bloques se organizan en un espacio de 3 dimensiones con las palabras claves threadIdx.x, threadIdx.y y threadIdx.z.



CUDA – Organización de Flujos

Cuando un proceso quiere ejecutar un kernel debe especificar el tamaño del grid y de los bloques de flujo.

Para especificar estos tamaños se usan parámetros de tipo dim3 que reciben como argumentos las dimensiones del grid en termino de nº de bloques y como segundo parámetro la dimensión de cada bloque en términos de nº de flujos.

Ejemplo:

```
// Configuración de las dimensiones de grid y bloques
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);

// Invocación del kernel (suma de matrices)
matrix_add_gpu<<<dimGrid, dimBlock>>>(a, b, c, N);
```

CUDA también incorpora un mecanismo de sincronización de flujos mediante barrier __syncthreads().

OpenCL

- OpenCL es un interfaz que define una API multiplataforma para la programación paralela. Consta de 3 partes:
  - La especificación de un lenguaje multiplataforma.
  - Una interfaz de entorno de computación.
  - Una interfaz para coordinar la computación paralela.

- Tiene muchas similitudes con CUDA. Las diferencias principales son:
  - OpenCL, tiene un modelo de gestión de recursos más complejo.
  - Portabilidad entre diferentes fabricantes.
  - Soporta niveles de paralelismo de datos y de tareas.

OpenCL – Modelo de DPL

Igual que CUDA un programa en OpenCL esta formado por dos partes: los kernels y un programa que se ejecuta en la CPU.

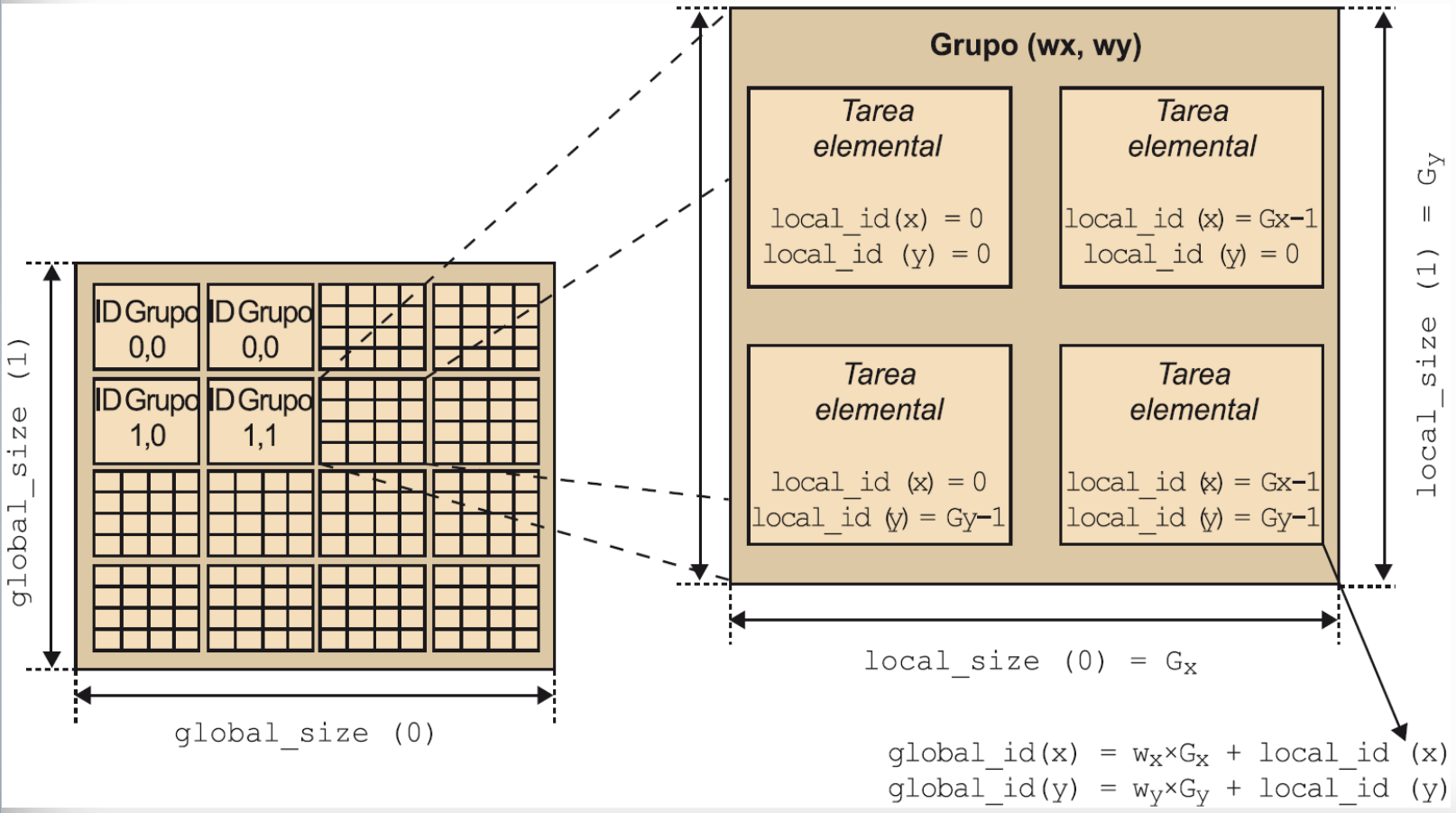
Correspondencias CUDA y OpenCL

OpenCL	CUDA
Kernel	*Kernel*
Programa procesador principal	Programa procesador principal
NDRange (rango de dimensión *N*)	*Grid*
Tarea elemental (*work item*)	Flujo
Grupo de tareas (*work group*)	Bloque
get_global_id(0);	blockIdx.x * blockDim.x + threadIdx.x
get_local_id(0);	threadIdx.x
get_global_size(0);	gridDim.x*blockDim.x
get_local_size(0);	blockDim.x

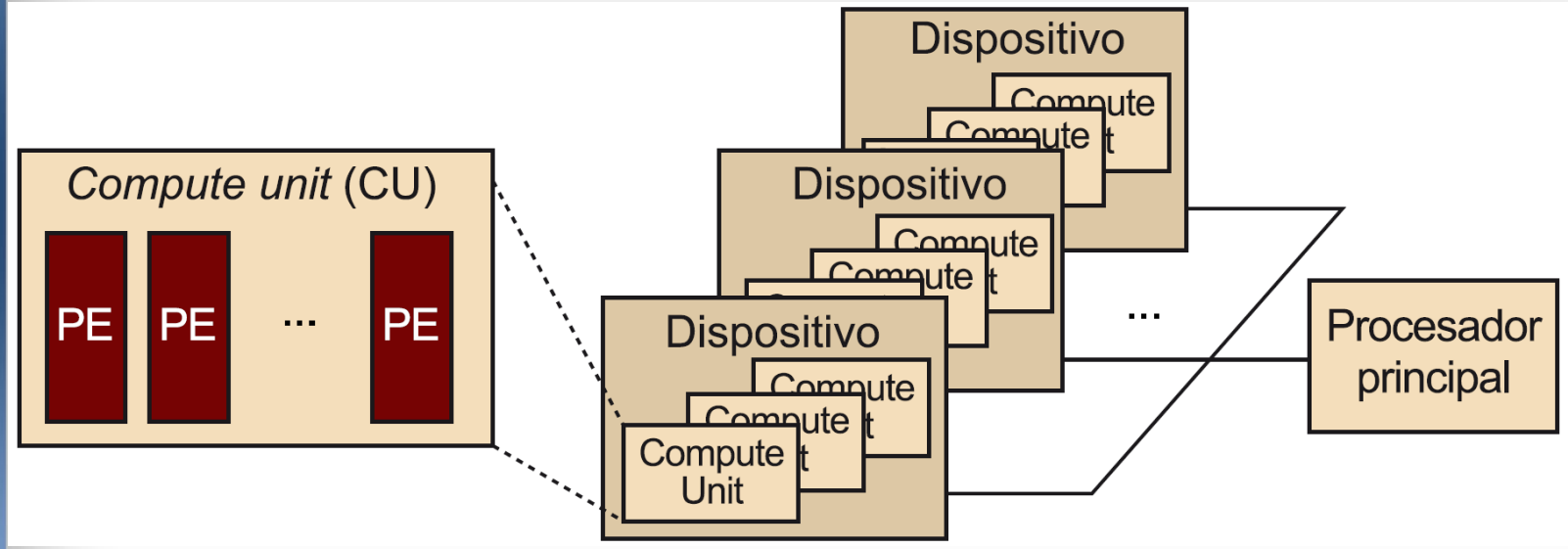
OpenCL – Modelo de DPL

Paralelismo a nivel de datos

get_global_id(), get_local_id()
get_global_size(), get_local_size()



OpenCL - Arquitectura Conceptual



- Posee un procesador principal conectado a uno o más dispositivos OpenCL.
- Estos dispositivos OpenCL están compuestos por unidades de cómputo (CU) (similares a las SM de CUDA)
- A su vez las CU están compuestas elementos de procesamiento (PE) (similares a los SP de CUDA)

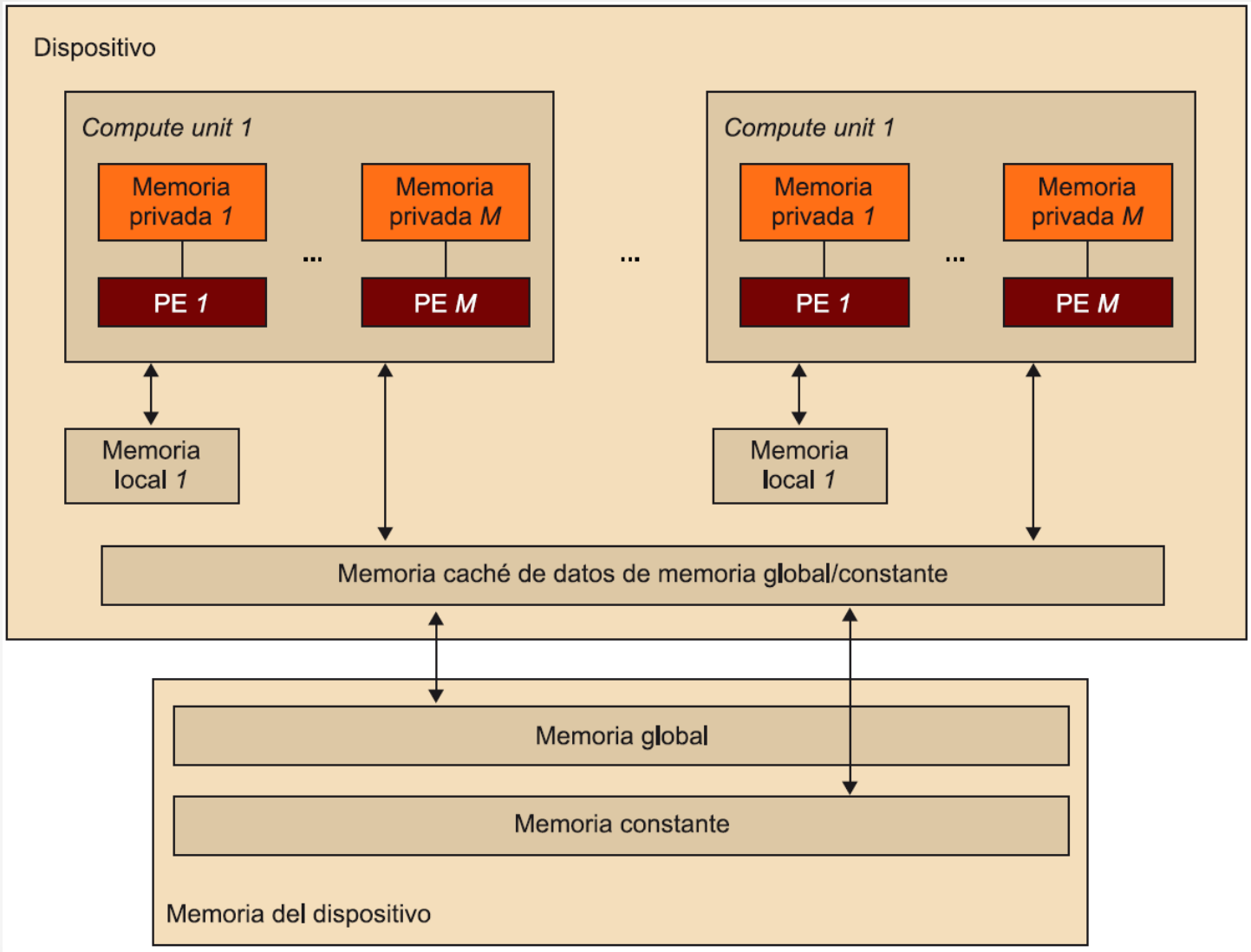
OpenCL - Modelo de Memoria

Jerarquía de la memoria de OpenCL

Hay disponibles varios tipos de memoria:

- **Memoria global:** la pueden utilizar todas las unidades de cálculo del dispositivo.
- **Memoria constante:** se utiliza para almacenar datos constantes para acceso solo de lectura de todas las unidades de cálculo.
- **Memoria local:** se suele utilizar para las tareas elementales de un grupo.
- **Memoria privada:** es la que puede utilizar únicamente una unidad de cálculo única.

OpenCL - Modelo de Memoria



OpenCL - Modelo de Memoria

- La iteración entre el espacio de memoria principal y de los dispositivos(GPU) es completamente independiente y puede ser de dos tipos:
  - Copiando datos explícitamente.
    1. Bloqueante: Se pueden acceder a los datos de manera segura desde la memoria principal.
    2. No bloqueante: La llamada a la función OpenCL finaliza inmediatamente, sacándose de la cola.
  - Mapeando/desmapeando regiones de un objeto OpenCL en memoria.

(Las instrucciones también pueden ser bloqueantes y no bloqueantes)

OpenCL - Gestión de Kernels y Dispositivos

Los kernels en OpenCL se ejecutan de la misma manera que en CUDA, las funciones se declaran utilizando la palabra clave __kernel:

Ejemplo:

```
__kernel void matrix_add_opencl
    (   __global const float *A,
        __global const float *B,
        __global float *C,
        int N)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    int index = i + j*N;

    if (i<N && j<N){
    C[index] = A[index] + B[index];
    }
}
```

OpenCL - Gestión de Kernels y Dispositivos

- El modelo de gestión es mas sofisticado que el de CUDA, este modelo permite a los dispositivos gestionarse mediante contextos.

  Para gestionar a los dispositivos hay que:

  1. Crear un contexto que contenga los dispositivos (clCreateContext() o clCreateContextFromType()).
  2. Indicar a las funciones la cantidad y el tipo de dispositivos del sistema mediante la función clGetDeviceIDs().
  3. Crear una cola de instrucciones para el dispositivo mediante la función clCreateCommandQueue().

  - Una vez hecho esto el procesador ya puede ejecutar el programa y enviar los datos del kernel a la GPU.

OpenCL - Gestión de Kernels y Dispositivos

Ejemplo de gestión de 2 dispositivos