Arquitecturas basadas en computación gráfica (GPU)

Francesc Guim
Ivan Rodero

PID_00184818

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este módulo didáctico, vamos a estudiar las arquitecturas basadas en computación gráfica (GPU), que es una de las tendencias en computación paralela con más crecimiento en los últimos años y que goza de gran popularidad, entre otras cuestiones, debido a la buena relación entre las prestaciones que ofrecen y su coste.

Primero repasaremos los fundamentos de la computación gráfica para entender los orígenes y las características básicas de las arquitecturas basadas en computación gráfica. Estudiaremos el *pipeline* gráfico y su evolución desde arquitecturas con elementos especializados hasta los *pipelines* programables y la arquitectura unificada, que permite la programación de aplicaciones de propósito general con GPU. Veremos las diferencias principales entre CPU y GPU y algunas de las arquitecturas GPU más representativas, poniendo énfasis en cómo pueden ser programados ciertos elementos.

Una vez presentadas las arquitecturas orientadas a computación gráfica, nos centraremos en las arquitecturas unificadas modernas que están focalizadas en la programación de aplicaciones de propósito general. Analizaremos las características de las principales arquitecturas relacionadas, como son las de Nvidia, AMD e Intel. Finalmente, estudiaremos CUDA y OpenCL, que son los principales modelos de programación para aplicaciones de propósito general sobre GPU.

Objetivos

Los materiales didácticos de este módulo contienen las herramientas necesarias para alcanzar los objetivos siguientes:

1. Conocer los fundamentos de la computación gráfica y las características básicas del *pipeline* gráfico.

2. Entender las diferencias y similitudes entre las arquitecturas CPU y GPU y conocer las características de las arquitecturas gráficas modernas.

3. Entender la creciente importancia de la programación masivamente paralela y las motivaciones de la computación de propósito general para GPU (GPGPU).

4. Entender las diferencias entre arquitecturas orientadas a gráficos y arquitecturas orientadas a la computación de propósito general.

5. Identificar la necesidad y fuente de paralelismo de datos de las aplicaciones para GPU.

6. Aprender los conceptos fundamentales para programar dispositivos GPU y los conceptos básicos de CUDA y OpenCL.

1. Introducción a la computación gráfica

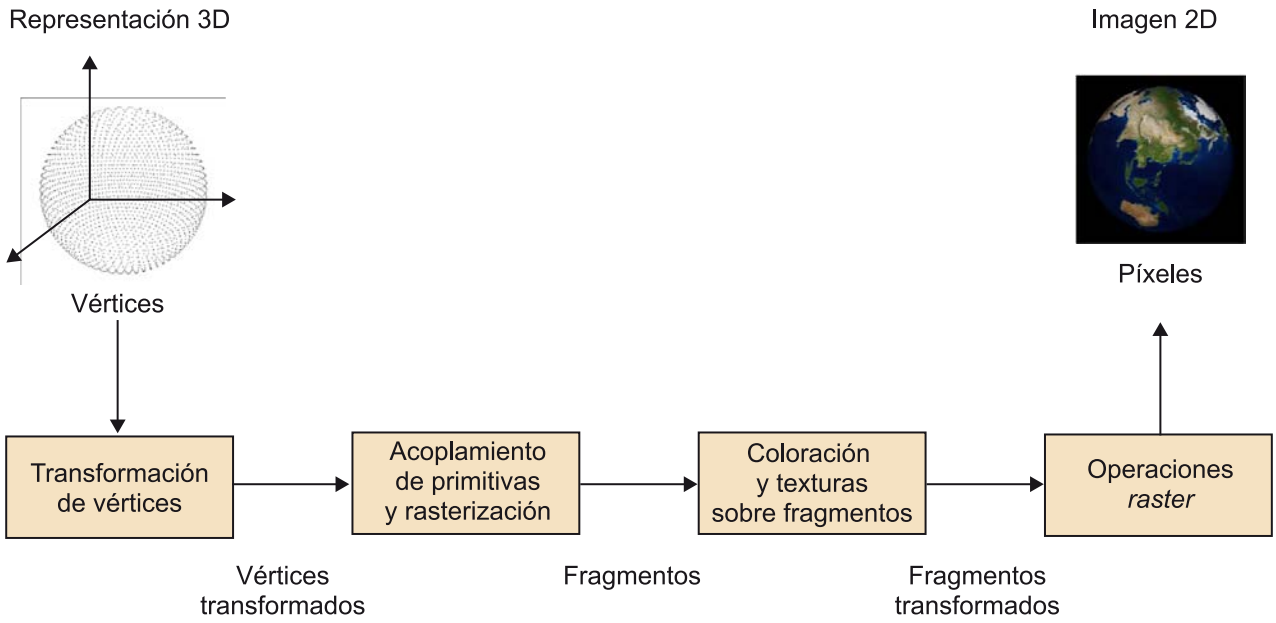
En este apartado, vamos a estudiar los fundamentos más básicos de la computación gráfica como paso previo para estudiar las características de las arquitecturas orientadas al procesamiento gráfico.

1.1. *Pipeline* básico

Tradicionalmente, los procesadores gráficos funcionan mediante un *pipeline* de procesamiento formado por etapas muy especializadas en las funciones que desarrollan y que se ejecutan en un orden preestablecido. Cada una de las etapas del *pipeline* recibe la salida de la etapa anterior y proporciona su salida a la etapa siguiente. Debido a la implementación mediante una estructura de *pipeline*, el procesador gráfico puede ejecutar varias operaciones en paralelo. Como este *pipeline* es específico para la gestión de gráficos, normalmente se denomina *pipeline* gráfico o *pipeline* de renderización. La operación de renderización consiste en proyectar una representación en tres dimensiones de una imagen en dos dimensiones (que es el objetivo de un procesador gráfico).

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, tal como muestra el *pipeline* simplificado de la figura 1.

Figura 1. *Pipeline* simplificado de un procesador gráfico



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. Algunas de estas operaciones son la

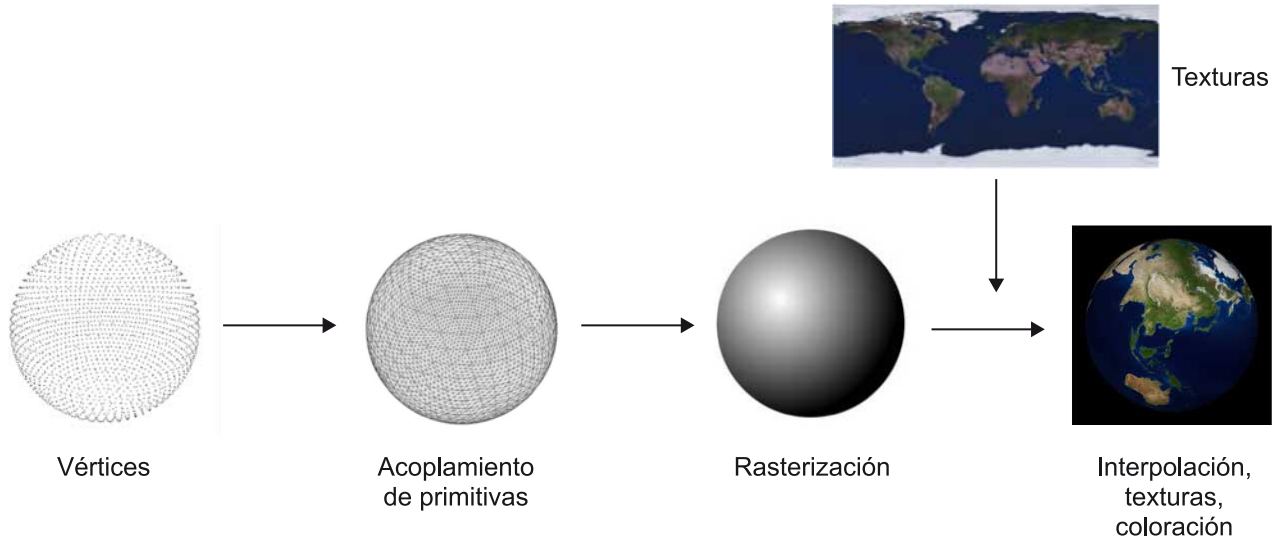
actualización de la posición o la rotación de los objetos representados, la generación de coordenadas para poder aplicar texturas o la asignación de color a los vértices. La salida de esta fase es un conjunto de vértices actualizados, uno para cada vértice de entrada.

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 vértices. Como resultado, se obtiene una secuencia de triángulos, líneas y puntos. Estos puntos son posteriormente procesados en una etapa llamada rasterización. La rasterización es un proceso que determina el conjunto de píxeles afectados por una primitiva determinada. Los resultados de la rasterización son conjuntos de localizaciones de píxeles y conjuntos de fragmentos. Un fragmento tiene asociada una localización y también información relativa a su color y brillo o bien a coordenadas de textura. Como norma general, podemos decir que, de un conjunto de tres o más vértices, se obtiene un fragmento.

En la etapa de **aplicación de texturas y coloreado**, el conjunto de fragmentos es procesado mediante operaciones de interpolación (predecir valores a partir de la información conocida), operaciones matemáticas, de aplicación de texturas y de determinación del color final de cada fragmento. Como resultado de esta etapa, se obtiene un fragmento actualizado (coloreado) para cada fragmento de entrada.

En las últimas etapas del *pipeline*, se ejecutan operaciones llamadas ***raster***, que se encargan de analizar los fragmentos mediante un conjunto de tests relacionados con aspectos gráficos. Estos tests determinan los valores finales que tomarán los píxeles. Si alguno de estos tests falla, se descarta el píxel correspondiente y, si todos son correctos, finalmente el píxel se escribe en la memoria (*framebuffer*). La figura 2 muestra las diferentes funcionalidades del *pipeline* gráfico

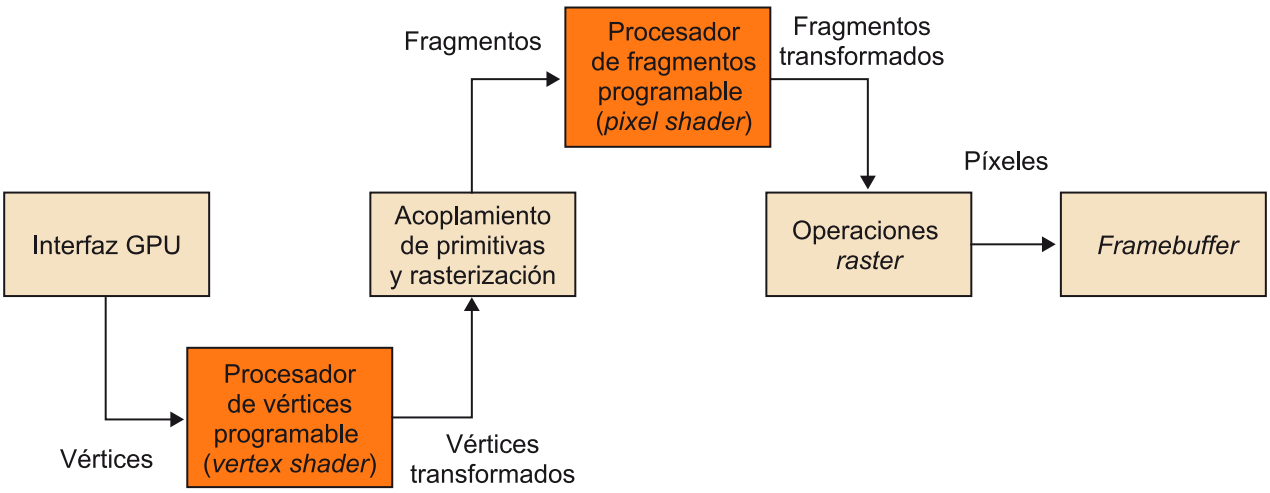
Figura 2. Resumen de las funcionalidades del *pipeline* gráfico



1.2. Etapas programables del *pipeline*

A pesar de que la tendencia es proporcionar un número más elevado de unidades programables en los procesadores gráficos, las fases que típicamente permiten la programación son las de transformación de vértices y transformación de fragmentos. Tal como muestra la figura 3, estas dos fases permiten programación mediante el procesador de vértices (*vertex shader*) y el procesador de fragmentos (*pixel shader*).

Figura 3. *Pipeline* gráfico programable



Las etapas de proceso de vértices y de fragmentos pueden ser programadas mediante los procesadores correspondientes.

El funcionamiento de un procesador de vértices programable es, en esencia, muy similar al de un procesador de fragmentos. 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 mismo. Hay tres tipos de registros:

- 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 los atributos se han cargado, el procesador ejecuta de manera secuencial cada una de las instrucciones que componen el programa. Estas instrucciones se encuentran en zonas reservadas de la memoria de vídeo.

Uno de los principales inconvenientes de estos tipos de procesadores programables es la limitación del conjunto de instrucciones que son capaces de ejecutar. Las operaciones que los procesadores de vértices tienen que poder hacer incluyen básicamente:

- Operaciones matemáticas en coma flotante sobre vectores (ADD, MULT, mínimo, máximo, entre otros).

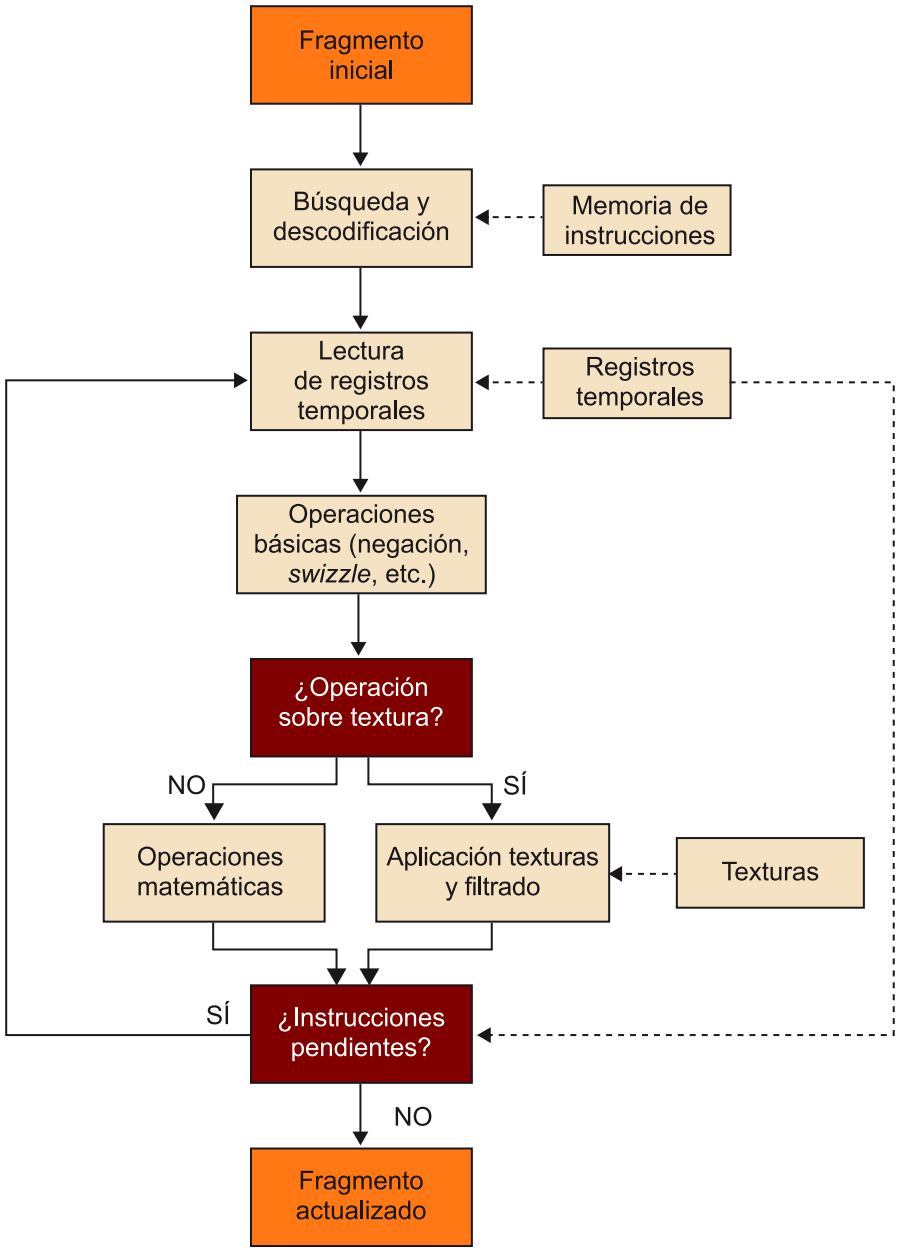
- Operaciones con hardware para la negación de vectores y *swizzling* (indexación de valores cuando se cargan de memoria).

- Exponenciales, logarítmicas y trigonométricas.

Los procesadores gráficos modernos soportan también operaciones de control de flujo que permiten la implementación de bucles y construcciones condicionales. Este tipo de procesadores gráficos dispone de procesadores de vértices totalmente programables que funcionan o bien en modalidad SIMD o bien en modalidad MIMD sobre los vértices de entrada.

Los procesadores de fragmentos programables requieren muchas de las operaciones matemáticas que exigen los procesadores de vértices, pero añaden operaciones sobre texturas. Este tipo de operaciones facilita el acceso a imágenes (texturas) mediante el uso de un conjunto de coordenadas para devolver a continuación la muestra leída tras un proceso de filtrado. Este tipo de procesadores solo funciona en modalidad SIMD sobre los elementos de entrada. Otra característica de estos procesadores es que pueden acceder en modalidad de lectura a otras posiciones de la textura (que corresponderán a un flujo con datos de entrada diferentes). La figura 4 muestra el funcionamiento esquemático de uno de estos procesadores.

Figura 4. Esquema del funcionamiento de un procesador de fragmentos



1.3. 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 para especificar los objetos y las operaciones necesarias para producir aplicaciones interactivas con gráficos en tres dimensiones.

OpenGL es una de las interfaces más populares. OpenGL está diseñada de manera completamente independiente al hardware para permitir implementaciones en varias plataformas. Esto hace que sea muy portátil, pero no incluye instrucciones para la gestión de ventanas o la gestión de acontecimien-

tos de usuario, entre otros. Las operaciones que se pueden llevar a cabo con OpenGL son principalmente las siguientes (y normalmente también en el orden siguiente):

- Modelar figuras a partir de primitivas básicas, que crean descripciones geométricas de los objetos (puntos, líneas, polígonos, fotografías y mapas de bits).

- Situar los objetos en el espacio tridimensional de la escena y seleccionar la perspectiva desde la que los queremos observar.

- Calcular el color de todos los objetos. El color se puede asignar explícitamente para cada píxel o bien se puede calcular a partir de las condiciones de iluminación o a partir de las texturas.

- Convertir la descripción matemática de los objetos y la información de color asociada a un conjunto de píxeles que se mostrarán por pantalla.

Aparte de estas operaciones básicas, OpenGL también desarrolla otras operaciones más complejas como, por ejemplo, la eliminación de partes de objetos que quedan ocultas tras otros objetos de la escena.

Dada la versatilidad de OpenGL, un programa en OpenGL puede llegar a ser bastante complejo. En términos generales, la estructura básica de un programa en OpenGL consta de las partes siguientes:

- Inicializar ciertos estados que controlan el proceso de renderización.

- Especificar qué objetos se tienen que visualizar mediante su geometría y sus propiedades externas.

El código 1.1 muestra un programa muy sencillo en OpenGL. En concreto, el código del ejemplo genera un cuadro blanco sobre un fondo negro. Como se trabaja en dos dimensiones, no se utiliza ninguna operación para situar la perspectiva del observador en el espacio 3D. La primera función abre una ventana en la pantalla. Esta función no pertenece realmente a OpenGL y, por lo tanto, la implementación depende del gestor de ventanas concreto que se utilice. Las funciones `glCleanColor` establecen el color actual y `glClear` borra la pantalla con el color indicado previamente con `glCleanColor`. La función `glColor3f` establece qué color se utilizará para dibujar objetos a partir de aquel momento. En el ejemplo, se trata del color blanco `(1.0, 1.0, 1.0)`. A continuación, mediante `glOrtho` se especifica el sistema de coordenadas 3D que se quiere utilizar para dibujar la escena y cómo se hace el mapeo en pantalla. Después de las funciones `glBegin` y `glEnd`, se define la geometría del objeto. En el ejemplo, se definen dos objetos (que aparecerán en la escena)

mediante `glVertex2f`. En este caso, se utilizan coordenadas en dos dimensiones, puesto que la figura que se quiere representar es un plano. Finalmente, la función `glFlush` asegura que las instrucciones anteriores se ejecuten.

```
#include <GL/gl.h>
#include <GL/glu.h>
void main ()
{
  OpenMainWindow ();
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f (1.0, 1.0, 1.0);
  glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  glBegin (GL_POLYGON);
    glVertice2f (-0.5, -0.5);
    glVertice2f (-0.5, 0.5);
    glVertice2f ( 0.5, 0.5);
    glVertice2f ( 0.5, -0.5);
  glEnd ();
  glFlush ();
}
```

Código 1.1. Ejemplo de código en OpenGL

La alternativa directa a OpenGL es Direct3D, que fue presentado en 1995 y finalmente se convirtió en el principal competidor de OpenGL. Direct3D ofrece un conjunto de servicios gráficos 3D en tiempo real que se encarga de toda la renderización basada en software-hardware de todo el *pipeline* gráfico (transformaciones, iluminación y rasterización) y del acceso transparente al dispositivo.

Direct3D es completamente escalable y permite que todo o una parte del *pipeline* gráfico se pueda acelerar por hardware. Direct3D también expone las capacidades más complejas de las GPU como por ejemplo *z-buffering*, *anti-aliasing*, *alfa blending*, *mipmapping*, efectos atmosféricos y aplicación de texturas mediante corrección de perspectiva. La integración con otras tecnologías de DirectX permite a Direct3D tener otras características, como las relacionadas con el vídeo, y proporcionar capacidades de gráficos 2D y 3D en aplicaciones multimedia.

Direct3D también tiene un nivel de complejidad bastante elevado. A modo de ejemplo, el código 1.2 muestra cómo podemos definir un cuadrado con Direct3D. Como Direct3D no dispone de ninguna primitiva para cuadrados, como en el caso de OpenGL, se tiene que definir con una secuencia de dos triángulos. Así pues, los tres primeros vértices forman un triángulo y después el resto de vértices añade otro triángulo formado por este vértice y los dos vértices anteriores. Por lo tanto, para dibujar un cuadrado, necesitamos definir cuatro vértices, que corresponden a dos triángulos.

```
Vertice vertices_cuadrado[] ={
  // x, y, z, rhw, color
  { 250.0f, 200.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,255,0) },
  { 250.0f, 50.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,0,255) },
  { 400.0f, 200.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(0,255,255) },
  { 400.0f, 50.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,255,255) }
};
```

Código 1.2. Ejemplo de código en Direct3D para dibujar un cuadrado

Otras alternativas de interfaces de programación del *pipeline* gráfico son SDL (*simple direct media layer*), Allegro y Render Ware.

1.4. Utilización del *pipeline* gráfico para computación general

Más adelante, veremos que podemos utilizar los procesadores gráficos para hacer computación general. A la hora de adaptar las funcionalidades de los procesadores gráficos a la computación general, parece que la mejor opción es utilizar los procesadores de fragmentos por los tres motivos siguientes:

1) En un procesador gráfico, normalmente hay más procesadores de fragmentos que procesadores de vértices. Este hecho ha sido cierto hasta la aparición de las arquitecturas unificadas, que fusionan los dos tipos de procesadores en un único modelo de procesador capaz de tratar tanto vértices como fragmentos.

2) Los procesadores de fragmentos soportan operaciones de lectura de datos procedentes de texturas (a pesar de que los últimos procesadores gráficos también tienen esta capacidad en la etapa de procesamiento sobre procesadores de vértices). Las texturas tienen un papel determinante a la hora de trabajar con conjuntos de datos (vectores y matrices) en procesadores gráficos.

3) Cuando se procesa un fragmento, el resultado se almacena directamente en la memoria y, por lo tanto, se puede reaprovechar directamente para ser procesado otra vez como un nuevo flujo de datos. En cambio, en el caso de los procesadores de vértices, el resultado obtenido de la computación tiene que pasar todavía por etapas de rasterización y procesamiento de fragmento antes de llegar a la memoria, cosa que hace más complicado utilizarlo para computación de propósito general.

La única forma que tienen los procesadores de fragmentos de acceder a la memoria es mediante las texturas. La unidad de texturas que hay en cualquier procesador gráfico ejerce el papel de interfaz solo de lectura en memoria. Cuando el procesador gráfico genera una imagen, puede haber dos opciones:

- Escribir la imagen en memoria (*framebuffer*), de forma que la imagen se muestre por pantalla.

- Escribir la imagen en la memoria de textura. Esta técnica se denomina *render-to-buffer* y resulta imprescindible en computación general (tal como veremos más adelante), ya que es el único mecanismo para implementar de

forma sencilla una realimentación entre los datos de salida de un proceso con la entrada del proceso posterior sin pasar por la memoria principal del sistema (que implicaría una transferencia de los datos bastante costosa).

A pesar de disponer de interfaces tanto de lectura como de escritura en memoria, hay que tener en cuenta que los procesadores de fragmentos pueden leer sobre memoria un número de veces ilimitado durante la ejecución de un programa, pero solo pueden hacer una única escritura al finalizar la ejecución. Por lo tanto, será muy difícil utilizar el *pipeline* tradicional programable para ejecutar programas de propósito general de manera eficiente.

2. Arquitecturas orientadas al procesamiento gráfico

En este apartado, vamos a estudiar las motivaciones y los factores de éxito del desarrollo de arquitecturas basadas en computación gráfica, las características básicas de estas arquitecturas y el caso particular de la arquitectura Nvidia orientada a gráficos como caso de uso. Finalmente, vamos a analizar las posibilidades y limitaciones para poder ser utilizadas para computación de propósito general.

2.1. Contexto y motivación

Durante las últimas décadas, uno de los métodos más relevantes para mejorar el rendimiento de los computadores ha sido el aumento de la velocidad del reloj del procesador. Sin embargo, los fabricantes se vieron obligados a buscar alternativas a este método debido a varios factores como, por ejemplo:

- Limitaciones fundamentales en la fabricación de circuitos integrados como, por ejemplo, el límite físico de integración de transistores.

- Restricciones de energía y calor debidas, por ejemplo, a los límites de la tecnología CMOS y a la elevada densidad de potencia eléctrica.

- Limitaciones en el nivel de paralelismo a nivel de instrucción (*instruction level parallelism*). Esto implica que, haciendo el *pipeline* cada vez más grande, se puede acabar obteniendo peor rendimiento.

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 en contra de programas secuenciales. Así pues, en los últimos años se ha producido un cambio muy significativo en el sector de la computación paralela. En la actualidad, casi todos los ordenadores de consumo incorporan procesadores multinúcleo. Desde la incorporación de los procesadores multinúcleo en dispositivos cotidianos, desde procesadores duales para dispositivos móviles hasta procesadores con más de una docena de núcleos para servidores y estaciones de trabajo, la computación paralela ha dejado de ser exclusiva de supercomputadores y sistemas de altas prestaciones. Así, estos dispositivos proporcionan funcionalidades más sofisticadas que sus predecesores mediante computación paralela.

En paralelo, durante los últimos años, la demanda por parte de los usuarios de gran potencia de cálculo en el ámbito de la generación de gráficos tridimensionales ha provocado una rápida evolución del hardware dedicado a la computación gráfica o GPU (*graphics processing unit*).

2.2. Visión histórica

A finales de la década de 1980 y principios de la de 1990 hubo un aumento muy importante de la popularidad de los sistemas operativos con interfaces gráficas, como Microsoft Windows, que empezó a acaparar gran parte del mercado. A principios de la década de 1990, se empezaron a popularizar los dispositivos aceleradores para 2D orientados a computadoras personales. Estos aceleradores apoyaban al sistema operativo gráfico ejecutando operaciones sobre mapas de bits directamente con hardware.

A la vez que se producía esta evolución en la informática de masas, en el mundo de la computación profesional la empresa Silicon Graphics dedicó muchos esfuerzos durante la década de 1980 a desarrollar soluciones orientadas a gráficos tridimensionales. Silicon Graphics popularizó el uso de tecnologías para 3D en diferentes sectores como el gubernamental, de defensa y la visualización científica y técnica, además de proporcionar herramientas para crear efectos cinematográficos nunca vistos hasta aquel momento. En 1992, Silicon Graphics abrió la interfaz de programación de su hardware por medio de la biblioteca OpenGL con el objetivo de que OpenGL se convirtiera en el estándar para escribir aplicaciones gráficas 3D independientemente de la plataforma utilizada.

A mediados de la década de 1990, la demanda de gráficos 3D por parte de los usuarios aumentó vertiginosamente a partir de la aparición de juegos inmersivos en primera persona, como Doom, Duke Nukem 3D o Quake, que acercaban al sector de los videojuegos para ordenadores personales entornos 3D cada vez más realistas. Al mismo tiempo, empresas como Nvidia, ATI Technologies y 3dfx Interactive empezaron a comercializar aceleradores gráficos que eran suficientemente económicos para los mercados de gran consumo. Estos primeros desarrollos representaron el principio de una nueva era de gráficos 3D que ha llevado a una constante progresión en las prestaciones y capacidad computacional del procesamiento gráfico.

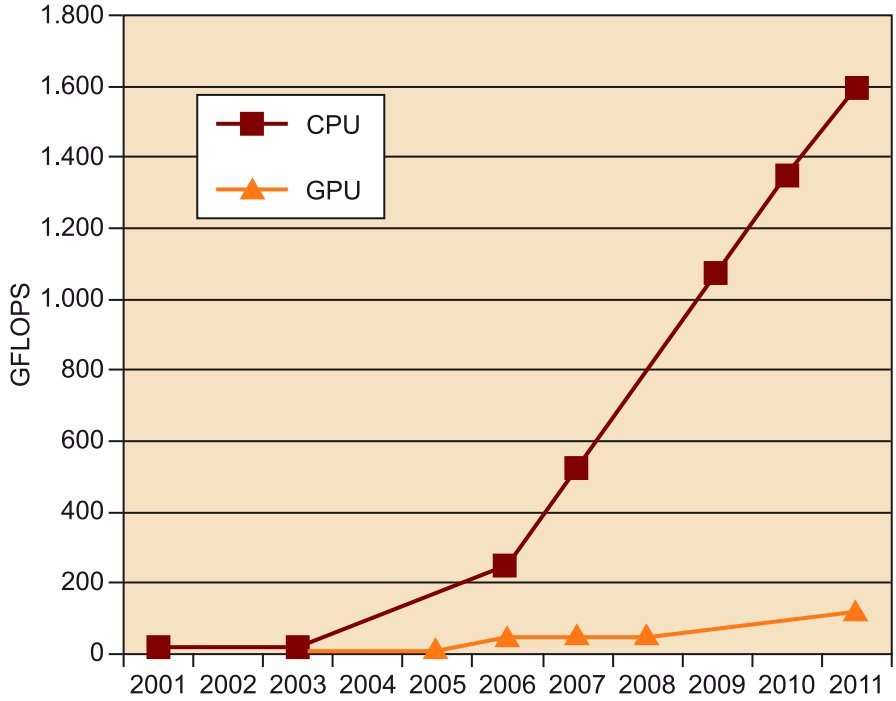
La aparición de la Nvidia GeForce 256 representó un impulso importante para abrir todavía más el mercado del hardware gráfico. Por primera vez, un procesador gráfico era capaz de entablar operaciones de transformación e iluminación directamente en el procesador gráfico y mejoraba así las posibilidades de desarrollar aplicaciones mucho más interesantes desde un punto de vista visual. Como las transformaciones y la iluminación ya eran parte del *pipeline* de OpenGL, la GeForce 256 marcó el comienzo de una progresión natural hacia dispositivos capaces de implementar cada vez más etapas del *pipeline* gráfico directamente en el procesador gráfico.

Desde el punto de vista de la computación paralela, la aparición de la GeForce 3 de Nvidia en el 2001 representó seguramente el cambio más significativo en la tecnología GPU hasta aquel momento. La serie GeForce 3 fue el primer chip del sector en implementar el que entonces era el nuevo estándar DirectX8.0. El

hardware compatible con este estándar disponía de etapas programables tanto para el procesamiento de vértices como para el procesamiento de fragmentos. Así pues, por primera vez, los desarrolladores tuvieron un cierto control sobre los cálculos que se podían desarrollar en las GPU.

Desde el punto de vista arquitectural, las primeras generaciones de GPU tenían una cantidad de núcleos bastante reducida, pero rápidamente se incrementó hasta hoy en día, cuando hablamos de dispositivos de tipo *many-core* con centenares de núcleos en un único chip. Este aumento de la cantidad de núcleos hizo que en el 2003 hubiera un salto importante de la capacidad de cálculo en coma flotante de las GPU respecto a las CPU, tal como muestra la figura 5. Esta figura muestra la evolución del rendimiento en coma flotante (pico teórico) de la tecnología basada en CPU (Intel) y GPU (Nvidia) durante la última década. Se puede apreciar claramente que las GPU van mucho más por delante que las CPU respecto a la mejora de rendimiento, en especial a partir del 2009, cuando la relación era aproximadamente de 10 a 1.

Figura 5. Comparativa de rendimiento (pico teórico) entre tecnologías CPU y GPU



Las diferencias tan grandes entre el rendimiento de CPU y GPU multinúcleo se deben principalmente a una cuestión de filosofía de diseño. Mientras que las GPU están pensadas para explotar el paralelismo a nivel de datos con el paralelismo masivo y una lógica bastante simple, el diseño de una CPU está optimizado para la ejecución eficiente de código secuencial. Las CPU utilizan lógica de control sofisticada que permite un paralelismo a nivel de instrucción y fuera de orden y utilizan memorias caché bastante grandes para reducir el tiempo de acceso a los datos en memoria. También hay otras cuestiones, como el consumo eléctrico o el ancho de banda de acceso a la memoria. Las GPU actuales tienen anchos de banda a memoria en torno a diez veces superiores

a los de las CPU, entre otras cosas porque las CPU deben satisfacer requisitos heredados de los sistemas operativos, de las aplicaciones o dispositivos de entrada/salida.

También ha habido una evolución muy rápida desde el punto de vista de la programación de las GPU que ha hecho cambiar el propósito de estos dispositivos. Las GPU de principios de la década del 2000 utilizaban unidades aritméticas programables (*shaders*) para devolver el color de cada píxel de la pantalla. Como las operaciones aritméticas que se aplicaban a los colores de entrada y texturas las podía controlar completamente el programador, los investigadores observaron que los colores de entrada podían ser cualquier tipo de dato. Así pues, si los datos de entrada eran datos numéricos que tenían algún significado más allá de un color, los programadores podían ejecutar cualquiera de los cálculos que necesitaran sobre esos datos mediante los *shaders*.

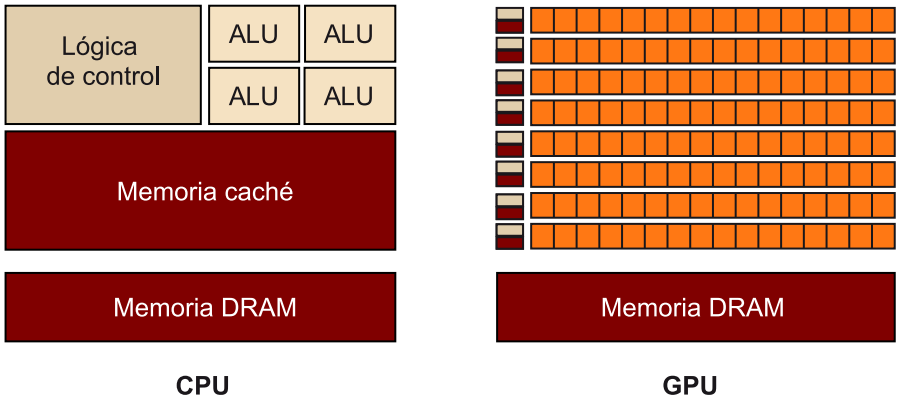
A pesar de las limitaciones que tenían los programadores para desarrollar aplicaciones sobre GPU (por ejemplo, escribir resultados en cualquier dirección de memoria), el alto rendimiento con operaciones aritméticas hizo que se dedicaran muchos esfuerzos a desarrollar interfaces y entornos de programación de aplicaciones de propósito general para GPU. Algunas de estas interfaces de programación han tenido mucha aceptación en varios sectores, a pesar de que su uso todavía requiere de una cierta especialización.

2.3. Características básicas de los sistemas basados en GPU

Tal como se ha indicado, la filosofía de diseño de las GPU está influida por la industria del videojuego, que ejerce una gran presión económica para mejorar la capacidad de realizar una gran cantidad de cálculos en coma flotante para procesamiento gráfico. Esta demanda hace que los fabricantes de GPU busquen formas de maximizar el área del chip y la cantidad de energía dedicada a los cálculos en coma flotante. Para optimizar el rendimiento de este tipo de cálculos, se ha optado por explotar un número masivo de flujos de ejecución. La 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.

Tal como se muestra en la figura 6, en este modelo se requiere menos lógica de control para cada flujo de ejecución. Al mismo tiempo, se dispone de una pequeña memoria caché que permite que flujos que comparten memoria tengan el ancho de banda suficiente para no tener que ir todos a la DRAM. En consecuencia, mucha más área del chip se dedica al procesamiento de datos en coma flotante.

Figura 6. Comparativa de la superficie dedicada típicamente a computación, memoria y lógica de control para CPU y GPU



Tal como veremos en detalle más adelante, además de disponer de muchos flujos de ejecución dispuestos en núcleos más sencillos que los de las CPU, otras características básicas de las arquitecturas GPU son las siguientes:

- Siguen el modelo SIMD (una instrucción con múltiples datos). 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.

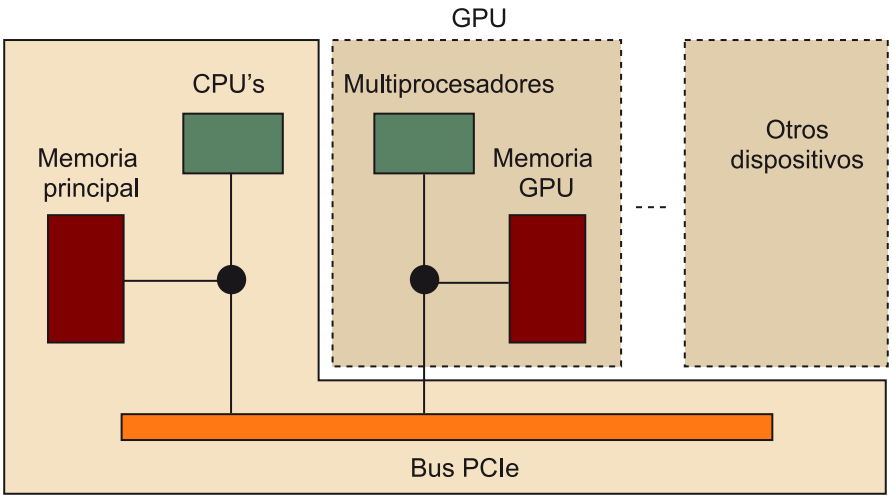
- La velocidad de ejecución se basa en la explotación de la localidad de los datos, tanto la localidad temporal (cuando accedemos a un dato, es probable que se vuelva a utilizar el mismo dato en un futuro cercano) como la localidad espacial (cuando accedemos a una fecha, es muy probable que se utilicen datos adyacentes a los ya utilizados en un futuro cercano y, por eso, se utilizan memorias caché que guardan varios datos en una línea del tamaño del bus).

- 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 (por ejemplo, solo lectura o lectura/escritura).

- El ancho de banda de la memoria es mayor.

En un sistema que dispone de una o de múltiples GPU, normalmente las GPU son vistas como dispositivos externos a la CPU (que puede ser multinúcleo o incluso un multiprocesador), que se encuentra en la placa base del computador, tal como muestra la figura 7. La comunicación entre CPU y GPU se lleva a cabo por medio de un puerto dedicado. En la actualidad, el PCI Express o PCIe (*peripheral component interconnect express*) es el estándar para ejecutar esta comunicación.

Figura 7. Interconexión entre CPU y GPU mediante PCIe



Otro puerto de comunicación muy extendido es el AGP (*accelerated graphics port*), que se desarrolló durante la última etapa de la pasada década en respuesta a la necesidad de velocidades de transferencia más elevadas entre CPU y GPU, debido a la mejora de las prestaciones de los procesadores gráficos. El AGP es un puerto paralelo de 32 bits con acceso directo al NorthBridge del sistema (que controla el funcionamiento del bus de interconexión de varios elementos cruciales como la CPU o la memoria) y, por lo tanto, permite utilizar parte de la memoria principal como memoria de vídeo. La velocidad de transferencia de datos varía entre los 264 MB/s y los 2 GB/s (para AGP 8x), en función de la generación de AGP.

A pesar del aumento en la velocidad de transferencia de datos en las subsiguientes generaciones de AGP, estas no son suficientes para dispositivos gráficos de última generación. Por este motivo, en el 2004 se publicó el estándar PCIe. El PCIe es un desarrollo del puerto PCI que, a diferencia del AGP, utiliza una comunicación en serie en lugar de ser en paralelo. Con el PCIe, se puede llegar a velocidades de transferencia de datos mucho más elevadas, que llegan a estar en torno a algunos GB/s. Por ejemplo, en la versión 3.0 del PCIe, el máximo teórico es de 16 GB/s direccionales y 32 GB/s bidireccionales.

Debido a la organización de las GPU respecto a la CPU, hay que tener en cuenta que 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** (en ambos sentidos). Como consecuencia, por ejemplo, no se puede usar `printf` en el código que se ejecuta en una GPU y, en general, el proceso de depuración en GPU suele ser bastante pesado.

2.4. Arquitectura Nvidia GeForce

Comercialmente, Nvidia ofrece diferentes productos, divididos en las tres familias principales siguientes:

- **GeForce**: orientada al gran mercado de consumo multimedia (videojuegos, edición de vídeo, fotografía digital, entre otros).

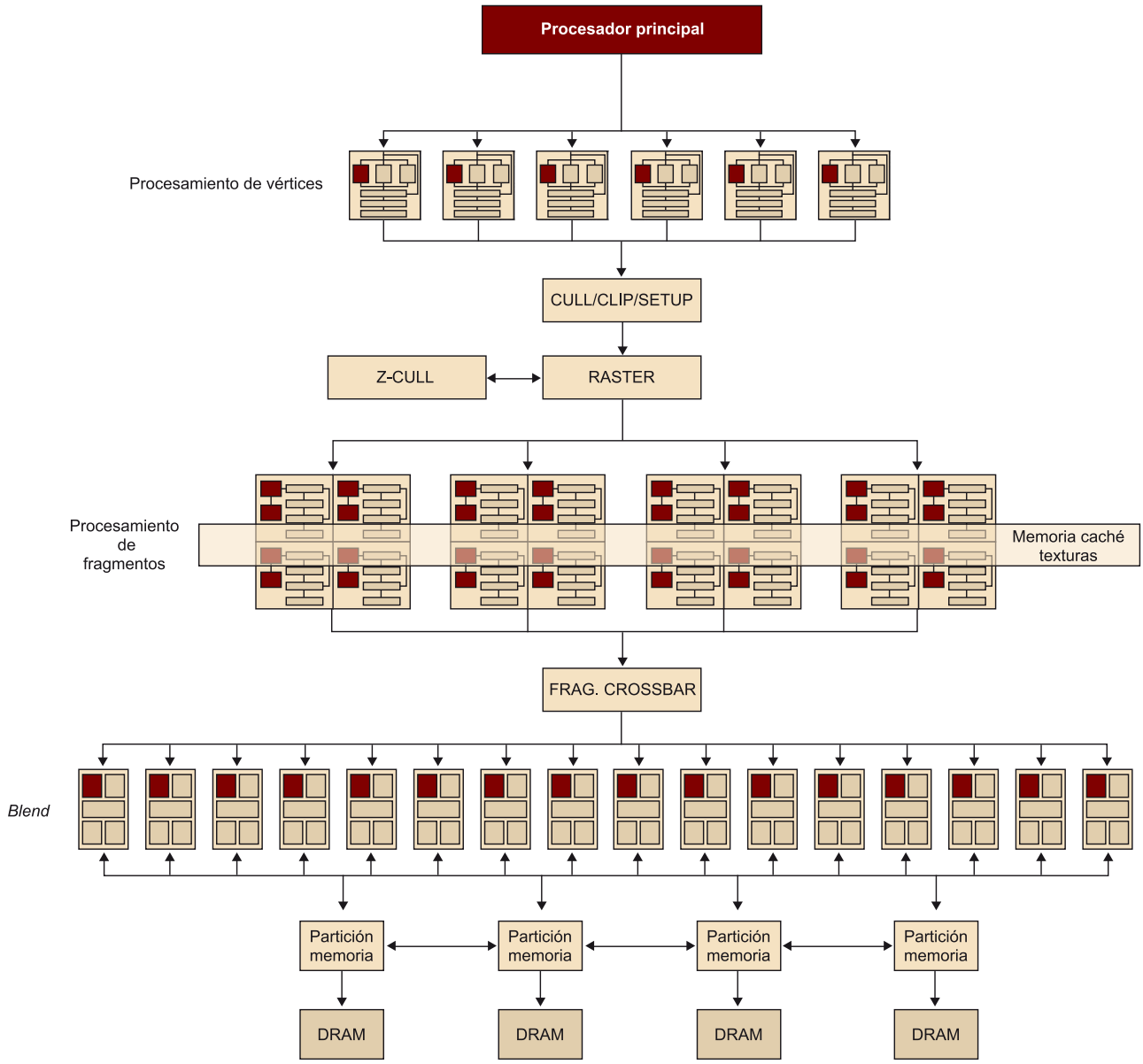
- **Cuadro**: orientada a soluciones profesionales que requieren modelos 3D, como los sectores de la ingeniería o la arquitectura.

- **Tesla**: orientada a la computación de altas prestaciones, como el procesamiento de información sísmica, simulaciones de bioquímica, modelos meteorológicos y de cambio climático, computación financiera o análisis de datos.

En este subapartado, vamos a utilizar la serie Nvidia GeForce 6 como caso de uso de GPU pensada para tratamiento de gráficos. A pesar de no ser la arquitectura más actual, nos servirá para estudiar mejor las diferencias con las arquitecturas orientadas a computación de propósito general y a entender mejor la evolución de las arquitecturas de GPU. También podemos encontrar arquitecturas parecidas de otros fabricantes, como la ATI (actualmente AMD).

La figura 8 muestra de modo esquemático los bloques principales que forman la arquitectura GeForce6 de Nvidia.

Figura 8. Esquema de la arquitectura de la GPU GeForce 6 de Nvidia

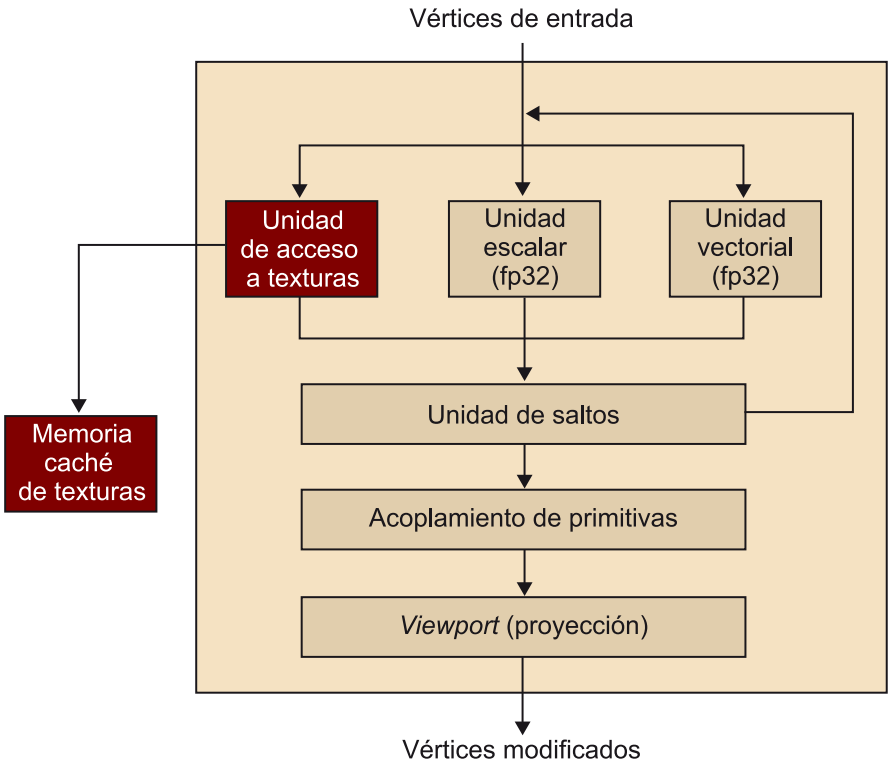


La CPU (*host* en la figura) envía a la unidad gráfica tres tipos de datos: instrucciones, texturas y vértices. Los procesadores de vértices son los encargados de aplicar un programa específico que se dedica a ejecutar las transformaciones sobre cada uno de los vértices de entrada. La serie GeForce 6 fue la primera que permitía que un programa ejecutado en el procesador de vértices fuera capaz de consultar datos de textura. Todas las operaciones se llevan a cabo con una precisión de 32 bits en coma flotante (fp32). El número de procesadores de vértices disponibles es variable en función del modelo de procesador, a pesar de que suele estar entre dos y dieciséis.

Como los procesadores de vértices son capaces de realizar lecturas de la memoria de texturas, cada uno tiene conexión a la memoria caché de texturas, tal como muestra la figura 9. Además, hay otra memoria caché (de vértices) que almacena datos relativos a vértices antes y después de haber sido procesados por el procesador de vértices.

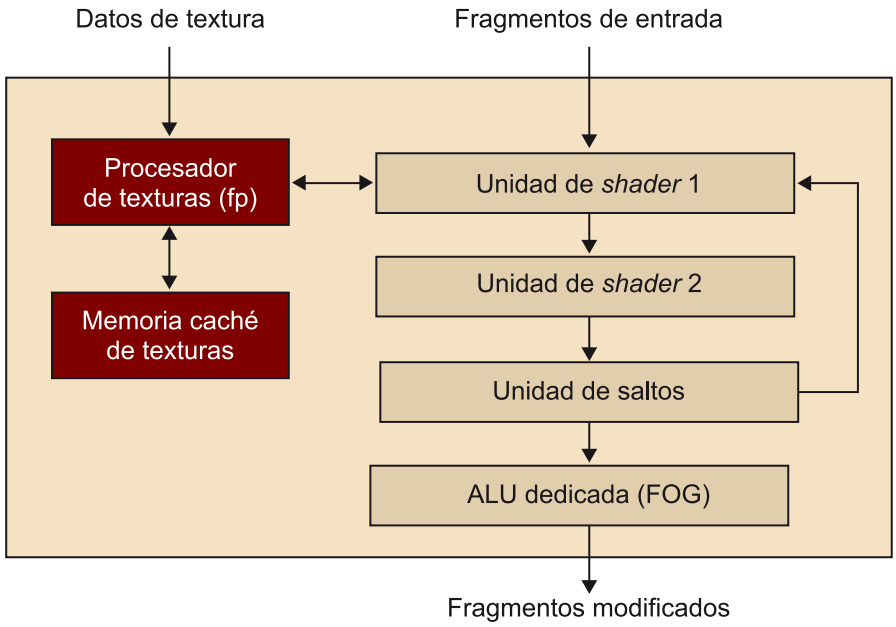
Los vértices se agrupan seguidamente en primitivas (puntos, líneas o triángulos). El bloque etiquetado en la figura como CULL/CLIP/SETUP ejecuta operaciones específicas para cada primitiva, que elimina, transforma o prepara para la etapa de rasterización. El bloque funcional dedicado a la rasterización calcula qué píxeles son afectados por cada primitiva y hace uso del bloque Z-CULL para descartar píxeles. Una vez ejecutadas estas operaciones, los fragmentos pueden ser vistos como candidatos a píxeles y pueden ser transformados por el procesador de fragmentos.

Figura 9. Esquema del procesador de vértices de la serie GeForce 6 de Nvidia



La figura 10 muestra la arquitectura de los procesadores de fragmentos típicos de la serie GeForce 6 de Nvidia. Los procesadores de fragmentos se dividen en dos partes: la unidad de textura, que está dedicada al trabajo con texturas, y la unidad de procesamiento de fragmentos, que opera, con ayuda de la unidad de texturas, sobre cada uno de los fragmentos de entrada. Las dos unidades operan de forma simultánea para aplicar un mismo programa (*shader*) a cada uno de los fragmentos de manera independiente.

Figura 10. Esquema del procesador de fragmentos de la serie GeForce 6 de Nvidia



De modo similar a lo que pasaría con los procesadores de vértices, los datos de textura se pueden almacenar en memorias caché en el chip mismo con el fin de reducir el ancho de banda en la memoria y aumentar así el rendimiento del sistema.

El procesador de fragmentos utiliza la unidad de texturas para cargar datos desde la memoria (y, opcionalmente, filtrar los fragmentos antes de ser recibidos por el procesador de fragmentos mismo). La unidad de texturas soporta gran cantidad de formatos de datos y de tipos de filtrado, a pesar de que todos los datos son devueltos al procesador de fragmentos en formato fp32 o fp16. Los procesadores de fragmentos poseen dos unidades de procesamiento que operan con una precisión de 32 bits (unidades de *shader* en la figura). Los fragmentos circulan por las dos unidades de *shader* y por la unidad de saltos antes de ser encaminados de nuevo hacia las unidades de *shader* para seguir ejecutando las operaciones. Este reencaminamiento sucede una vez por cada ciclo de reloj. En general, es posible llevar a cabo un mínimo de ocho operaciones matemáticas en el procesador de fragmentos por ciclo de reloj o cuatro en el supuesto de que se produzca una lectura de datos de textura en la primera unidad de *shader*.

Para reducir costes de fabricación, la memoria del sistema se divide en cuatro particiones independientes, cada una construida a partir de memorias dinámicas (DRAM). Todos los datos procesados por el *pipeline* gráfico se almacenan en memoria DRAM, mientras que las texturas y los datos de entrada (vértices) se pueden almacenar tanto en la memoria DRAM como en la memoria principal del sistema. Estas cuatro particiones de memoria proporcionan un subsistema de memoria de bastante anchura (256 bits) y flexible, que logra velocidades de transferencia cercanas a los 35 GB/s (para memorias DDR con velocidad de reloj de 550 Mhz, 256 bits por ciclo de reloj y dos transferencias por ciclo).

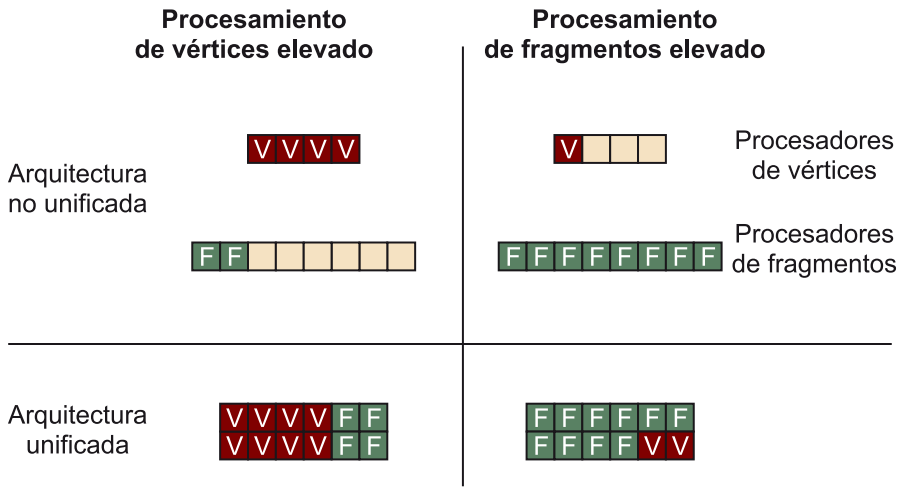
Por lo tanto, comparando la implementación hecha por esta serie de procesadores (muy similar a otras series de la misma generación), es posible identificar qué unidades funcionales se corresponden con cada una de las etapas del *pipeline* gráfico. Este tipo de implementaciones han sido las más extendidas hasta la aparición de la última generación de GPU, que se basan en una arquitectura unificada y no establecen ninguna diferenciación entre las diferentes etapas del flujo de procesamiento a nivel de hardware.

2.5. Concepto de arquitectura unificada

La arquitectura de la serie GeForce 6 estudiada con anterioridad se podría definir como una arquitectura dividida a nivel de *shaders* o procesadores programables. Esto quiere decir que dispone de un hardware especializado para ejecutar programas que operan sobre vértices y otro dedicado exclusivamente a la ejecución sobre fragmentos. A pesar de que el hardware dedicado se puede adaptar bastante bien a su función, hay ciertos inconvenientes que hacen que se haya optado por arquitecturas totalmente diferentes a la hora de desarrollar una nueva generación de procesadores gráficos, basados en una arquitectura unificada.

Las arquitecturas anteriores (no unificadas) tenían problemas importantes cuando la carga de trabajo de procesamiento de vértices y de procesamiento de fragmentos de las aplicaciones gráficas que ejecutaban no estaba balanceada. Como normalmente las GPU tienen menos unidad de procesamiento de vértices que de fragmentos, el problema se agravaba cuando la cantidad de trabajo sobre vértices era predominante, puesto que en ese caso las unidades de vértices quedaban totalmente ocupadas, mientras que muchas unidades de fragmentos podían quedar desaprovechadas. Del mismo modo, cuando la carga es principalmente sobre fragmentos, también se puede producir un desperdicio de recursos. La figura 11 muestra la ejecución de dos aplicaciones no balanceadas y cómo la arquitectura unificada puede ofrecer una solución más eficiente.

Figura 11. Comparativa de la asignación de procesadores de una GPU en el procesamiento de vértices y de fragmentos en arquitecturas unificadas y no unificadas, para diferentes tipos de aplicaciones



La arquitectura unificada permite ejecutar más computación simultánea y mejorar la utilización de los recursos.

La solución que se desarrolló a partir de la serie G80 de Nvidia o la R600 de ATI (actualmente AMD) fue crear arquitecturas unificadas a nivel de *shaders*. En este tipo de arquitecturas, no existe la división a nivel de hardware entre procesadores de vértices y procesadores de fragmentos. Cualquier unidad de procesamiento que las forma (denominadas también ***stream processors***) es capaz de trabajar tanto a nivel de vértice como a nivel de fragmento, sin estar especializada en un tipo en concreto.

Este cambio en la arquitectura también comporta un cambio importante en el *pipeline* gráfico. Con arquitecturas unificadas, no hay partes específicas del chip asociadas a una etapa concreta del *pipeline*, sino que un único tipo de unidad es el encargado de ejecutar todas las operaciones, sea cual sea su naturaleza. Una de las ventajas de este tipo de arquitecturas es el balanceo implícito de la carga computacional. El conjunto de procesadores se puede asignar a una tarea o a otra, dependiendo de la carga que el programa exija a nivel de un determinado tipo de procesamiento. Así pues, la arquitectura unificada puede solucionar el problema de balanceo de la carga y asignación de unidades de procesamiento en cada etapa del *pipeline* gráfico pero, como contrapartida, los procesadores que forman la GPU son más complejos, puesto que son más genéricos.

Este tipo de arquitecturas ofrece un potencial mucho mayor para hacer computación de propósito general. En el apartado siguiente, vamos a ver cómo podemos programar aplicaciones de propósito general en este tipo de dispositivos y también vamos a estudiar varias arquitecturas GPU orientadas a la computación de propósito general.

3. Arquitecturas orientadas a computación de propósito general sobre GPU (GPGPU)

En este apartado, vamos a estudiar cómo los procesadores gráficos pueden ser utilizados para ejecutar aplicaciones que tradicionalmente son ejecutadas en CPU, vamos a ver cuáles son los tipos de aplicaciones adecuados para computación gráfica y, finalmente, vamos a estudiar algunas de las principales arquitecturas GPU orientadas a computación de propósito general.

Tal como ya hemos visto, la capacidad de cálculo de una GPU actual es más elevada que la de las CPU más avanzadas para ejecutar ciertas tareas. Esto ha hecho que este tipo de dispositivos se estén volviendo muy populares para el cómputo de algoritmos de propósito general y no solamente para la generación de gráficos. La computación de propósito general sobre GPU se conoce popularmente como GPGPU (*general-purpose computing on graphics processing unit*).

Además del nivel de paralelismo masivo y del alto rendimiento que proporcionan las plataformas GPU, hay que destacar que estos dispositivos ofrecen una muy buena relación entre el precio y las prestaciones, factores esenciales para que los fabricantes hayan apostado fuertemente por esta tecnología. Aun así, las GPU proporcionan rendimientos muy elevados solo para ciertas aplicaciones debido a sus características arquitecturales y de funcionamiento. De modo general, podemos decir que 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.

Existen varios dominios en los que la introducción de la GPGPU ha proporcionado una gran mejora en términos de *speedup* de la ejecución de las aplicaciones asociadas. Entre las aplicaciones que se pueden adaptar eficientemente a la GPU, podemos destacar el álgebra lineal, el procesamiento de imágenes, algoritmos de ordenación y búsqueda, procesamiento de consultas sobre bases de datos, análisis de finanzas, mecánica de fluidos computacional o predicción meteorológica.

Uno de los principales inconvenientes a la hora de trabajar con GPU es la 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. Por este motivo, se han desarrollado modelos de progra-

mación, ya sean de propiedad (CUDA) o abiertos (OpenCL), que proporcionan al programador un nivel de abstracción más cercano a la programación para CPU, que le simplifican considerablemente su tarea.

A pesar de que los programadores pueden ver reducida la complejidad de la programación de GPGPU mediante estas abstracciones, vamos a analizar los principales fundamentos de la programación GPGPU antes de ver las arquitecturas sobre las que se ejecutarán los programas desarrollados. El principal argumento del modelo GPGPU es la utilización del procesador de fragmentos (o *pixel shader*) como unidad de cómputo. También hay que tener en cuenta que la entrada/salida es limitada: se pueden hacer lecturas arbitrariamente, pero hay restricciones para las escrituras (por ejemplo, en las texturas).

Para comprender cómo las aplicaciones de propósito general se pueden ejecutar en una GPU, podemos hacer una serie de analogías entre GPU y CPU. Entre estas, podemos destacar las que estudiaremos a continuación.

Hay dos estructuras de datos fundamentales en las GPU para representar conjuntos de elementos del mismo tipo: las texturas y los vectores de vértices. Como los procesadores de fragmentos son los más utilizados, podemos establecer un símil entre los vectores de datos en CPU y las texturas en GPU. La memoria de texturas es la única memoria accesible de manera aleatoria desde programas de fragmentos o de vértices. Cualquier vértice, fragmento o flujo al que se tenga que acceder de forma aleatoria se tiene que transformar primero en textura. Las texturas pueden ser leídas o escritas tanto por la CPU como por la GPU. Podemos establecer un símil entre la lectura de memoria en CPU y la lectura de texturas en GPU. En el caso de la GPU, la escritura se hace mediante el proceso de renderización sobre una textura o bien copiando los datos del *framebuffer* a la memoria de textura. Desde el punto de vista de las estructuras de datos, las texturas son declaradas como conjuntos de datos organizados en una, dos o tres dimensiones, y se accede a cada uno de sus elementos mediante direcciones de una, dos o tres dimensiones, respectivamente. La manera más habitual de ejecutar la transformación entre vectores (o matrices) y texturas es mediante la creación de texturas bidimensionales.

En la mayoría de aplicaciones, en especial en las científicas, el problema se suele dividir en diferentes etapas, cuyas entradas dependen de las salidas de etapas anteriores. Estas también se pueden ver como las diferentes iteraciones de los bucles. Si hablamos de flujos de datos que son tratados por una GPU, cada núcleo tiene que procesar un flujo completo antes de que el núcleo siguiente se pueda empezar a ejecutar con los datos resultantes de la ejecución anterior. La implementación de esta retroalimentación de datos entre etapas del programa es trivial en la CPU, ya que cualquier dirección de memoria puede ser leída o escrita en cualquier punto del programa. La técnica *render-to-texture* es la que permite el uso de procedimientos similares en GPU, al escribir resultados de la

ejecución de un programa en la memoria para que puedan estar disponibles como entradas para etapas posteriores. Así pues, podemos establecer un símil entre la escritura en la memoria en CPU y el *render-to-texture* en GPU.

En GPU, la computación se hace normalmente por medio de un flujo de procesador de fragmentos, que se tendrá que ejecutar en las unidades funcionales de la GPU correspondientes. Así pues, podemos establecer un símil entre el programa en CPU y la rasterización (o programa del *shader*) en GPU. Para empezar la computación, se crea un conjunto de vértices con los que podemos alimentar los procesadores de vértices. La etapa de rasterización determinará qué píxeles del flujo de datos se ven afectados por las primitivas generadas a partir de estos vértices, y se genera un fragmento para cada uno. Por ejemplo, imaginemos que tenemos como objetivo operar sobre cada uno de los elementos de una matriz de N filas y M columnas. En este caso, los procesadores de fragmentos tendrán que ejecutar una (la misma) operación sobre cada uno de los $N \times M$ elementos que conforman la matriz.

A continuación, utilizaremos la suma de matrices para ejemplificar las similitudes entre GPU y CPU a la hora de programar una aplicación de propósito general. En este ejemplo, se toman como operandos de entrada dos matrices de valores reales, A y B, para obtener una tercera matriz C. Los elementos de la matriz C serán la suma de los elementos correspondientes de las matrices de entrada. Una implementación en CPU crea tres matrices en la memoria al recurrir a cada uno de los elementos de las matrices de entrada, calcular para cada pareja la suma y poner el resultado en una tercera matriz, tal como muestra el código 3.1.

```
float *A, B, C;
    int i, j;
    for (i=0; i<M; i++){
        for (j=0; j<N; j++){
            C[i][j] = A[i][j] + B[i][j];
        }
    }
```

Código 3.1. Implementación de la suma de matrices para CPU

El procedimiento en GPU es algo más complejo. Primero, es necesario definir tres texturas en memoria de vídeo, que actuarán del mismo modo que las matrices definidas en la memoria principal en el caso de las CPU. Cada núcleo o programa para ejecutar en la GPU corresponde a aquellas operaciones que se ejecutan en el bucle más interno de la implementación para CPU. Sin embargo, hay que tener en cuenta algunas limitaciones adicionales de las GPU, como el hecho de que no podremos escribir el resultado de la operación de la suma directamente en la textura correspondiente a la matriz C. Así pues, será necesario devolver el resultado de la suma y crear un flujo adicional que recogerá este resultado y lo enviará a la textura de destino (mediante la técnica *render-to-texture*). Finalmente, los datos almacenados en la textura correspon-

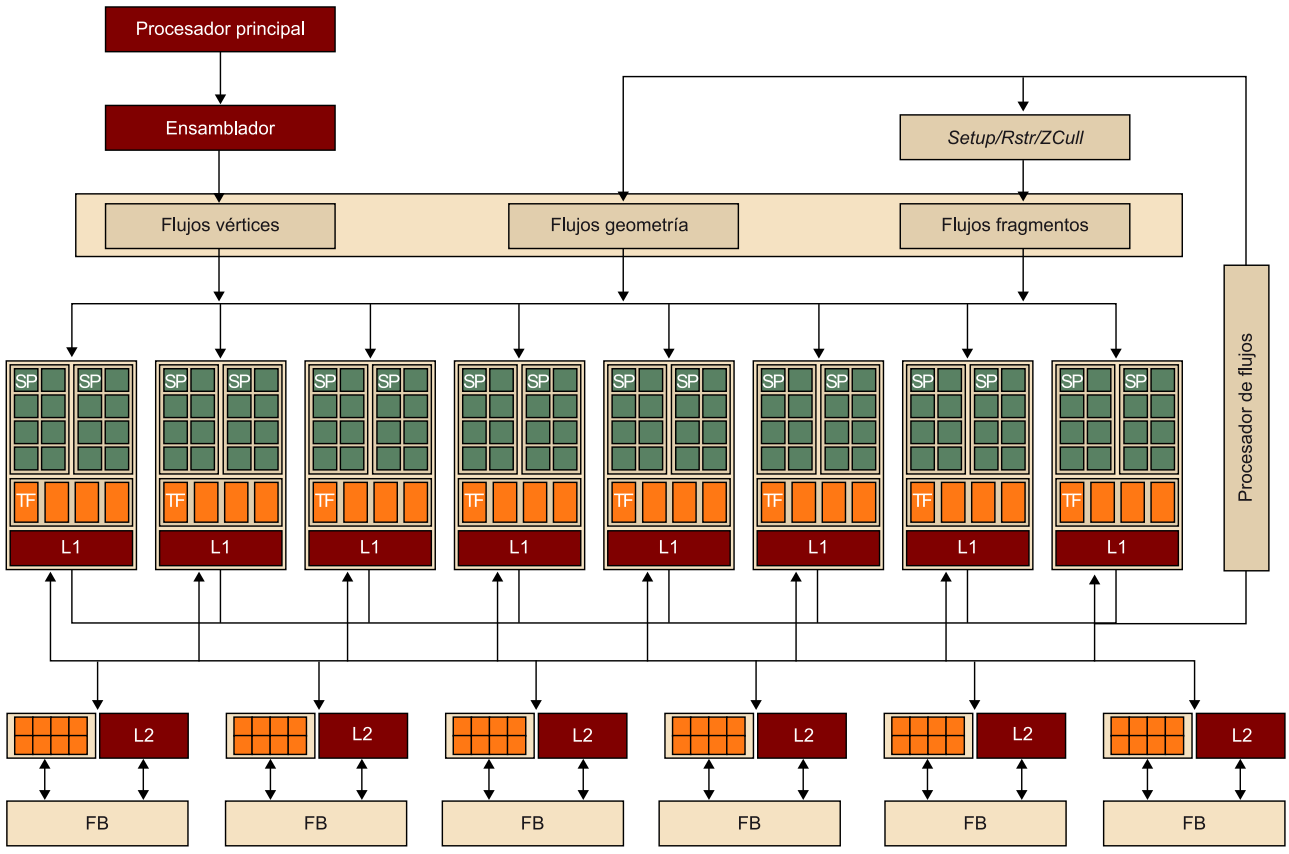
diente a la matriz *C* se transfieren otra vez hacia la memoria central para que el programa que invocó la ejecución del núcleo a la GPU pueda continuar con la ejecución.

3.1. Arquitectura Nvidia

Nvidia presentó a finales del 2006 una nueva línea de hardware orientado a la computación general de otras prestaciones llamada Tesla, que se empezó a desarrollar a mediados del 2002. Tesla ofrece un hardware de altas prestaciones (en forma, por ejemplo, de bloques de procesadores gráficos) sin ningún tipo de orientación a aplicaciones gráficas. A pesar de que no es la implementación de Tesla más actual, en este apartado nos vamos a centrar en la arquitectura G80, ya que representó un salto tecnológico importante por el hecho de implementar una arquitectura unificada y, tal como vamos a ver en el siguiente apartado, vino con el modelo de programación CUDA.

La arquitectura G80 de Nvidia se define como una arquitectura totalmente unificada, sin diferenciación a nivel de hardware entre las diferentes etapas que forman el *pipeline* gráfico, totalmente orientada a la ejecución masiva de flujos y que se ajusta al estándar IEEE 754. La figura 12 muestra el esquema completo de la arquitectura G80.

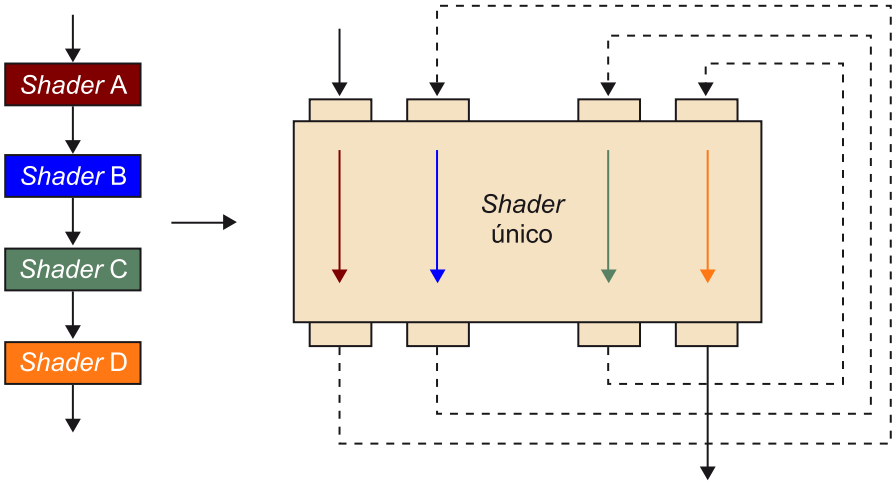
Figura 12. Arquitectura (unificada) de la serie G80 de Nvidia



La aparición de la arquitectura G80 representó una mejora de la capacidad de procesamiento gráfico y un incremento de las prestaciones respecto a la generación anterior de GPU, pero la clave en orden al ámbito de la computación general fue la mejora de la capacidad de cálculo en coma flotante. También se añadieron al *pipeline* para cumplir las características definidas por Microsoft en DirectX 10.

Gracias a la arquitectura unificada, el número de etapas del *pipeline* se reduce de manera significativa y pasa de un modelo secuencial a un modelo cíclico, tal como muestra la figura 13. El *pipeline* clásico utiliza diferentes tipos de *shaders* por medio de los cuales los datos se procesan secuencialmente. En cambio, en la arquitectura unificada solo hay una única unidad de *shaders* no especializados que procesan los datos de entrada (en forma de vértices) por pasos. La salida de un paso retroalimenta los *shaders* que pueden ejecutar un conjunto diferente de instrucciones, de este modo se emula el *pipeline* clásico, hasta que los datos han pasado por todas las etapas del *pipeline* y se encaminan hacia la salida de la unidad de procesamiento.

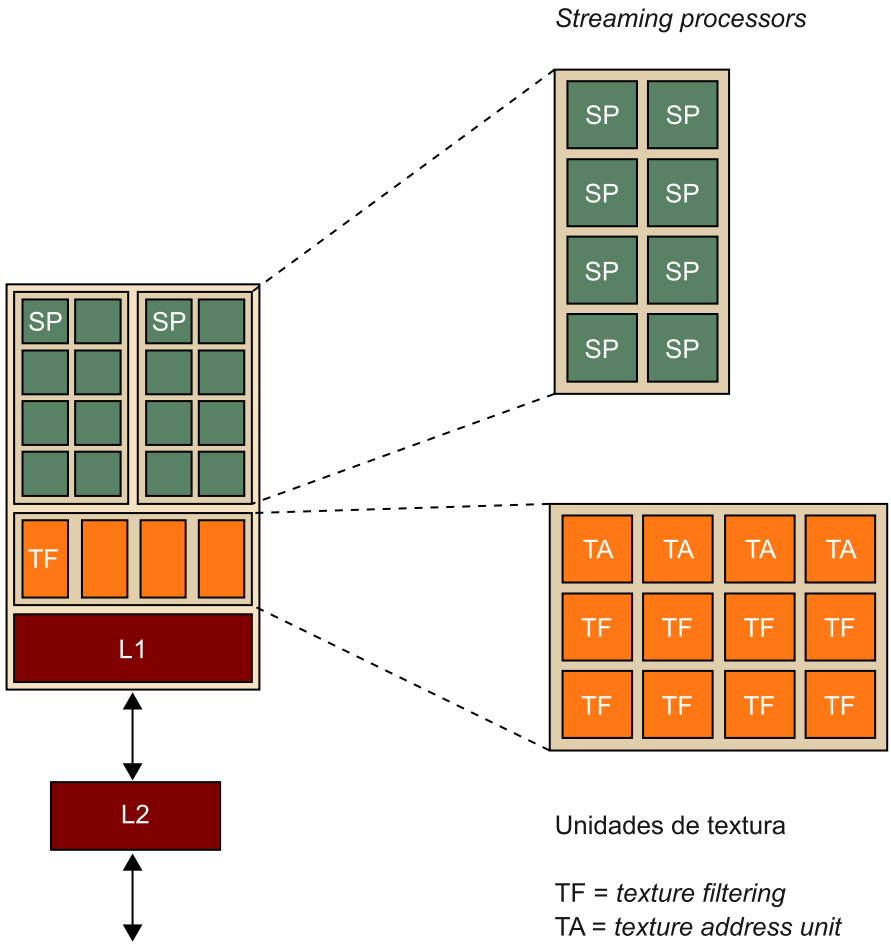
Figura 13. Comparación entre el *pipeline* secuencial (izquierda) y el cíclico de la arquitectura unificada (derecha)



El funcionamiento básico en la arquitectura G80 para ejecutar un programa consiste en dos etapas diferenciadas. En la primera etapa, los datos de entrada se procesan mediante hardware especializado. Este se encarga de distribuir los datos de tal manera que se puedan utilizar el máximo número de unidades funcionales para obtener la máxima capacidad de cálculo durante la ejecución del programa. En la segunda etapa, un controlador global de flujos se encarga de controlar la ejecución de los flujos que ejecutan los cálculos de manera coordinada. También determina en cada momento qué flujos y de qué tipo (de vértices, de fragmentos o de geometría) serán enviados a cada unidad de procesamiento que compone la GPU. Las unidades de procesamiento tienen un planificador de flujos que se encarga de decidir la gestión interna de los flujos y de los datos.

El núcleo de procesamiento de los *shaders* está formado por ocho bloques de procesamiento. Cada uno de estos bloques está formado por dieciséis unidades de procesamiento principal, denominadas *streaming processors* (SP). Tal como se ha indicado, cada bloque tiene un planificador de flujos, pero también memoria caché de nivel 1 (caché L1) y unidades de acceso y filtrado de texturas propias. Así pues, cada grupo de 16 SP agrupados dentro de un mismo bloque comparte unidades de acceso a texturas y memoria caché L1. La figura 14 muestra de modo esquemático la estructura de los bloques que forman la arquitectura G80 de Nvidia.

Figura 14. *Streaming processors* y unidades de textura que forman un bloque en la arquitectura G80 de Nvidia



Cada SP está diseñado para llevar a cabo operaciones matemáticas o bien direccionamiento de datos en memoria y la transferencia posterior. Cada procesador es una ALU que opera sobre datos escalares de 32 bits de precisión (estándar IEEE 754). Esto contrasta con el apoyo vectorial que ofrecían las arquitecturas anteriores a los procesadores de fragmentos.

Internamente, cada bloque está organizado en dos grupos de ocho SP. El planificador interno planifica una misma instrucción sobre los dos subconjuntos de SP que forman el bloque y cada uno toma un cierto número de ciclos de reloj, que estará definido por el tipo de flujo que se esté ejecutando en la unidad en aquel instante. Además de tener acceso a su conjunto de registros pro-

pio, los bloques también pueden acceder tanto al conjunto de registros global como a dos zonas de memoria adicionales, solo de lectura, llamadas memoria caché global de constantes y memoria caché global de texturas.

Los bloques pueden procesar flujos de tres tipos: de vértices, de geometría y de fragmentos, de forma independiente y en un mismo ciclo de reloj. También tienen unidades que permiten la ejecución de flujos dedicados exclusivamente a la transferencia de datos en memoria: unidades TF (*texture filtering*) y TA (*texture addressing*), con el objetivo de solapar el máximo posible las transferencias de datos a memoria con la computación.

Además de poder acceder a su conjunto propio de registros dedicado, en el conjunto de registros global, en la memoria caché de constantes y en la memoria caché de texturas, los bloques pueden compartir información con el resto de bloques por medio del segundo nivel de memoria caché (L2), a pesar de que únicamente en modo lectura. Para compartir datos en modo lectura/escritura es necesario el uso de la memoria DRAM de vídeo, con la penalización consecuente que representa en el tiempo de ejecución. En concreto, los bloques tienen los tipos de memoria siguientes:

- Un conjunto de registros de 32 bits locales en cada bloque.
- Una memoria caché de datos paralela o memoria compartida, común para todos los bloques y que implementa el espacio de memoria compartida.
- Una memoria solo de lectura llamada memoria caché de constantes, compartida por todos los bloques, que acelera las lecturas en el espacio de memoria de constantes.
- Una memoria solo de lectura llamada memoria caché de texturas, compartida por todos los bloques, que acelera las lecturas en el espacio de memoria de texturas.

La memoria principal se divide en seis particiones, cada una de las cuales proporciona una interfaz de 64 bits, y así se consigue una interfaz combinada de 384 bits. El tipo de memoria más utilizado es el GDDR. Con esta configuración, se logran velocidades de transferencia de datos entre memoria y procesador muy elevadas. Esta es una de las claves en la computación con GPU, ya que el acceso a la memoria es uno de los principales cuellos de botella.

El ancho de banda de comunicación con la CPU es de 8 GB/s: una aplicación CUDA puede transferir datos desde la memoria del sistema a 4GB/s a la vez que puede enviar datos hacia la memoria del sistema también a 4GB/s. Este ancho de banda es mucho más reducido que el ancho de banda en la memoria, lo que puede parecer una limitación, pero no lo es tanto, puesto que el ancho de banda del bus PCI Express es comparable al de la CPU en la memoria del sistema.

Además de las características antes expuestas, hay que tener en cuenta otras mejoras respecto a arquitecturas anteriores, como las siguientes:

- Conjunto unificado de instrucciones.
- Más registros y constantes utilizables desde un mismo *shader*.
- Número ilimitado de instrucciones para los *shaders*.
- Menos cambios de estado, con menos intervención de la CPU.
- Posibilidad de recirculación de los datos entre diferentes niveles del *pipeline*.
- Control del flujo dinámico tanto a nivel de *shaders* como de vértices y de píxeles.
- Introducción de operaciones específicas sobre enteros.

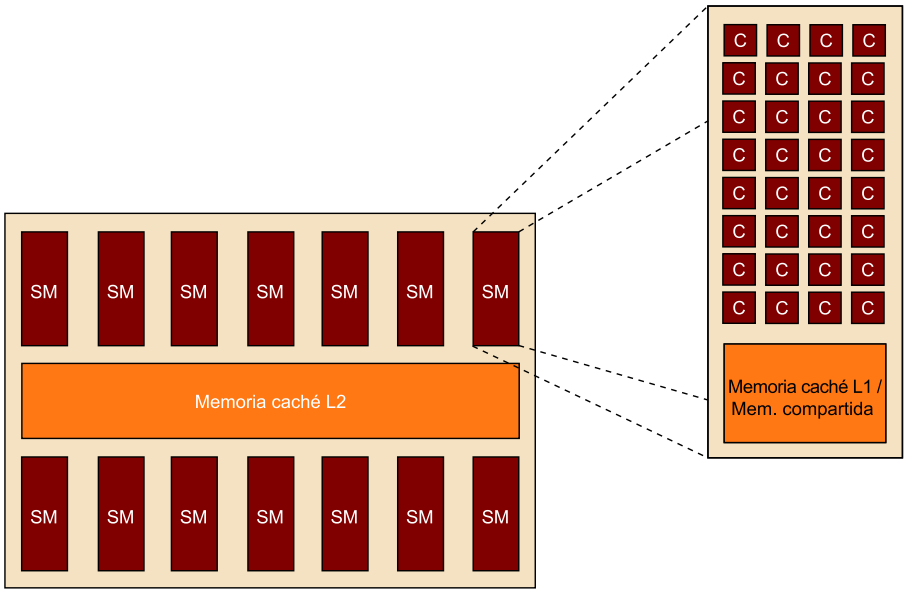
Implementaciones de Tesla posteriores a la serie G80 ofrecen todavía más prestaciones, como un número más elevado de SP, más memoria o ancho de banda. La tabla 1 muestra las principales características de algunas de las implementaciones dentro de esta familia de Nvidia.

Tabla 1. Comparativa de varios modelos de la familia Tesla de Nvidia

Modelo	Arquitectura	Reloj (MHz)	Núcleos		Memoria				Rendimiento - pico teórico (GFLOP)
			Número de SP	Reloj (MHz)	Tipo	Tamaño (MB)	Reloj (MHz)	Ancho de banda (GB/s)	
C870	**G80**	600	128	1.350	GDDR3	1.536	1.600	76	518
C1060	**GT200**	602	240	1.300	GDDR3	4.096	1.600	102	933
C2070	**GF100**	575	448	1.150	GDDR5	6.144	3.000	144	1.288
M2090	**GF110**	650	512	1.300	GDDR5	6.144	3.700	177	1.664

Las últimas generaciones de Nvidia implementan la arquitectura Fermi, que proporciona soluciones cada vez más masivas a nivel de paralelismo, tanto desde el punto de vista del número de SP como de flujos que se pueden ejecutar en paralelo. La figura 15 muestra un esquema simplificado de la arquitectura Fermi. En esta, vemos que el dispositivo está formado por un conjunto de *streaming multiprocessors* (SM) que comparten memoria caché L2. Cada uno de los SM está compuesto por 32 núcleos y cada uno de ellos puede ejecutar una instrucción entera o en punto flotante por ciclo de reloj. Aparte de la memoria caché, también incluye una interfaz con el procesador principal, un planificador de flujos y múltiples interfaces de DRAM.

Figura 15. Esquema simplificado de la arquitectura Fermi



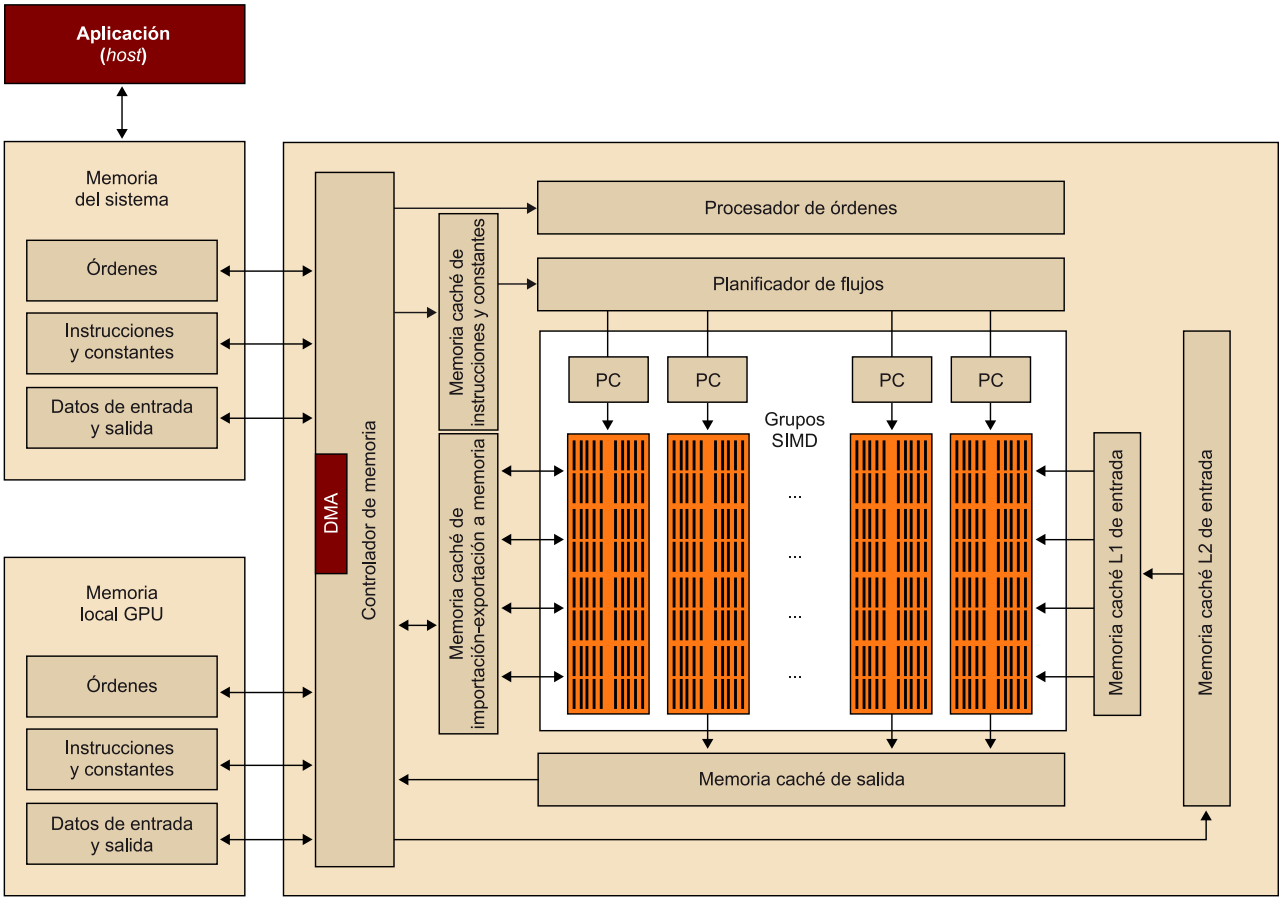
Fermi proporciona un nivel de abstracción en el que la GPU está compuesta por un conjunto uniforme de unidades computacionales con solo unos pocos elementos que le apoyan. Así pues, el objetivo principal de este diseño es dedicar la mayor parte de la superficie del chip y de la corriente eléctrica a la aplicación en cuestión y maximizar así el rendimiento en coma flotante.

3.2. Arquitectura AMD (ATI)

De forma similar a la tecnología de Nvidia, la tecnología GPU desarrollada por AMD/ATI ha evolucionado muy rápidamente en la última década hacia la computación GPGPU. En este subapartado, vamos a estudiar las principales características de la arquitectura de la serie R600 de AMD (que es el equivalente a la Nvidia G80) y la arquitectura CU (implementada por la familia Evergreen de AMD), que es un desarrollo nuevo y está asociada al modelo de programación OpenCL.

La serie R600 de AMD implementa una arquitectura unificada como en el caso de la Nvidia G80. También incluye un *shader* de geometría que permite ejecutar operaciones de creación de geometría en la GPU y así no tener que sobrecargar la CPU. La serie R600 incorpora 320 SP, que son muchos más que los 128 de la G80. Sin embargo, esto no quiere decir que sea mucho más potente, ya que los SP de las dos arquitecturas funcionan de forma bastante diferente. La figura 16 muestra el esquema de la arquitectura de la serie R600 de AMD.

Figura 16. Diagrama de bloques de la arquitectura de la serie R600 de AMD



Los SP en la arquitectura R600 están organizados en grupos de cinco (grupos SIMD), de los que solo uno puede ejecutar algunas operaciones más complejas (por ejemplo, operaciones en coma flotante de 32 bits), mientras que los otros cuatro se dedican solo a ejecutar operaciones simples con enteros. Esta arquitectura SIMD híbrida también se conoce como VLIW-5D. Como los SP están organizados de este modo, la arquitectura R600 solo puede ejecutar 64 flujos, cuando podríamos pensar que podría soportar hasta 320. Desde una perspectiva optimista, estos 64 flujos podrían ejecutar cinco instrucciones en cada ciclo de reloj, pero hay que tener en cuenta que cada una de estas instrucciones tiene que ser completamente independiente de las demás. Así pues, la arquitectura R600 deja gran responsabilidad al planificador de flujos que, además, tiene que gestionar una cantidad muy grande de flujos. El planificador de flujos se encarga de seleccionar dónde hay que ejecutar cada flujo mediante una serie de árbitros, dos para cada matriz de dieciséis grupos SIMD. Otra tarea que desarrolla el planificador de flujos es determinar la urgencia de la ejecución de un flujo. Esto quiere decir que el planificador de flujos puede asignar a un flujo un grupo SIMD ocupado, mantener los datos que este grupo SIMD estaba utilizando y, una vez el flujo prioritario ha finalizado, seguir ejecutando el flujo original con toda normalidad.

Las diferencias entre los SP de las arquitecturas Nvidia y AMD también incluyen la frecuencia de reloj a la que trabajan. Mientras que los SP de la implementación de Nvidia utilizan un dominio específico de reloj que funciona a

una velocidad más elevada que el resto de los elementos, en la implementación de AMD los SP utilizan la misma frecuencia de reloj que el resto de elementos y no tienen el concepto de dominio de reloj.

La tabla 2 muestra las características principales de las arquitecturas AMD orientadas a GPGPU. Notad que el número de SP es mucho más elevado que en las Nvidia, pero también existen diferencias importantes respecto a la frecuencia de reloj y ancho de banda en la memoria.

Tabla 2. Comparativa de varios modelos de GPU AMD

Modelo	Arquitectura (familia)	Núcleos		Memoria				Rendimiento - pico teórico (GFLOP)
		Número de SP	Reloj (MHz)	Tipo	Tamaño (MB)	Reloj (MHz)	Ancho de banda (GB/s)	
R580	**X1000**	48	600	GDDR3	1.024	650	83	375
RV670	**R600**	320	800	GDDR3	2.048	800	51	512
RV770	**R700**	800	750	GDDR5	2.048	850	108	1.200
Cypress (RV870)	**Evergreen**	1.600	825	GDDR5	4.096	1.150	147	2.640

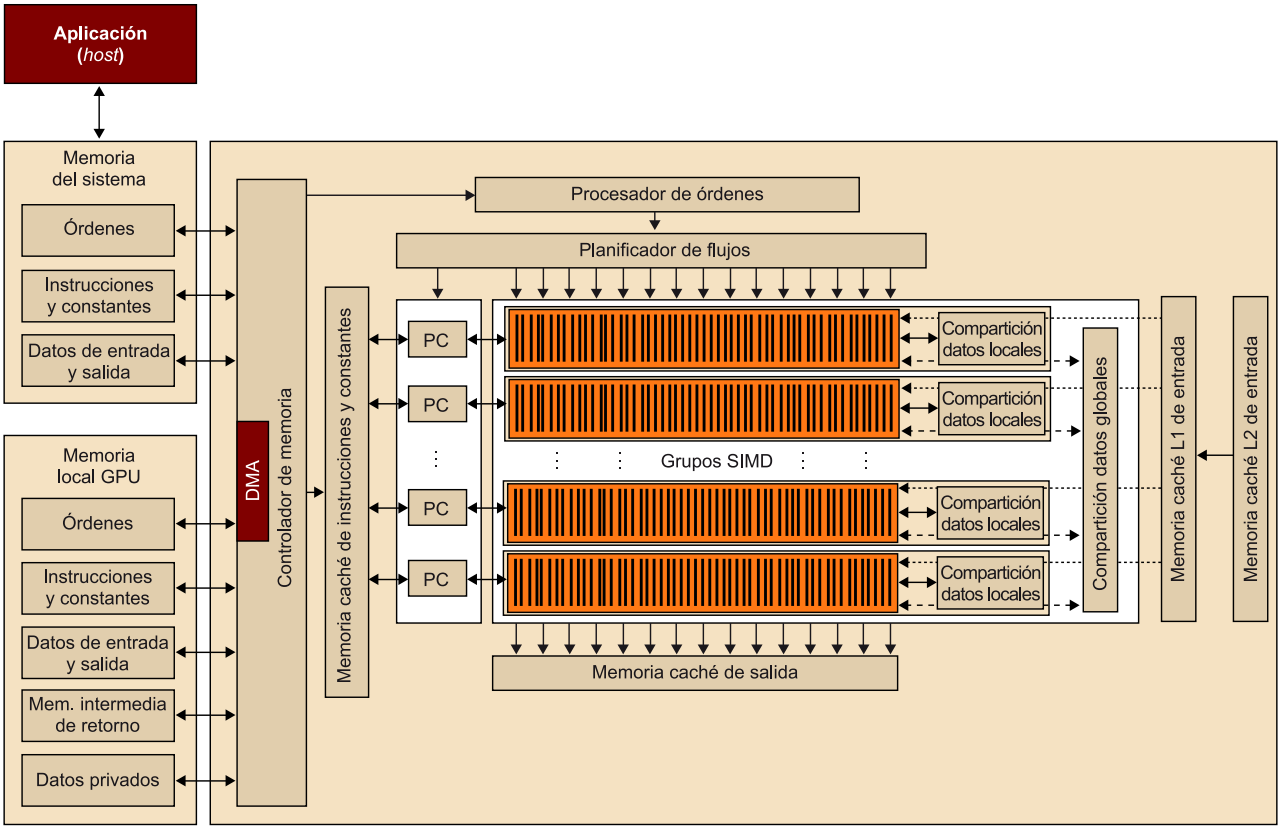
3.2.1. Arquitectura AMD CU

La adquisición de ATI por parte de AMD en el 2006 (que implicó la supresión de la marca ATI de sus productos) fue una pieza importante dentro de la estrategia de AMD para desarrollar una generación de procesadores que integraran capacidades para computación de propósito general y de funciones gráficas en un mismo chip. Así pues, AMD hizo grandes esfuerzos para desarrollar una nueva arquitectura que sustituyera la basada en VLIW-5D por otra más homogénea. Esta arquitectura se denominó CU (*Compute Unit*) y tiene como objetivo simplificar el modelo de programación para alentar a los programadores a la utilización de GPGPU. Tal como veremos en el próximo apartado, esta arquitectura viene con modelo de programación OpenCL.

La arquitectura CU es una arquitectura orientada a la computación multiflujo intensivo pero sin dejar de ofrecer gran rendimiento para la computación gráfica. Está basada en un diseño escalar (similar al Nvidia Fermi) con administración de recursos fuera de orden y enfocados a la ejecución simultánea de diferentes tipos de instrucciones independientes (gráficas, texturas, vídeo y cómputo general). Así pues, ofrece más potencia, latencias más pequeñas y, sobre todo, mucha más flexibilidad a los programadores. El nuevo núcleo de *shaders* unificado abandona el diseño vectorial VLIW y, en su lugar, se utiliza el nuevo CU, que está formado por cuatro unidades SIMD escalares.

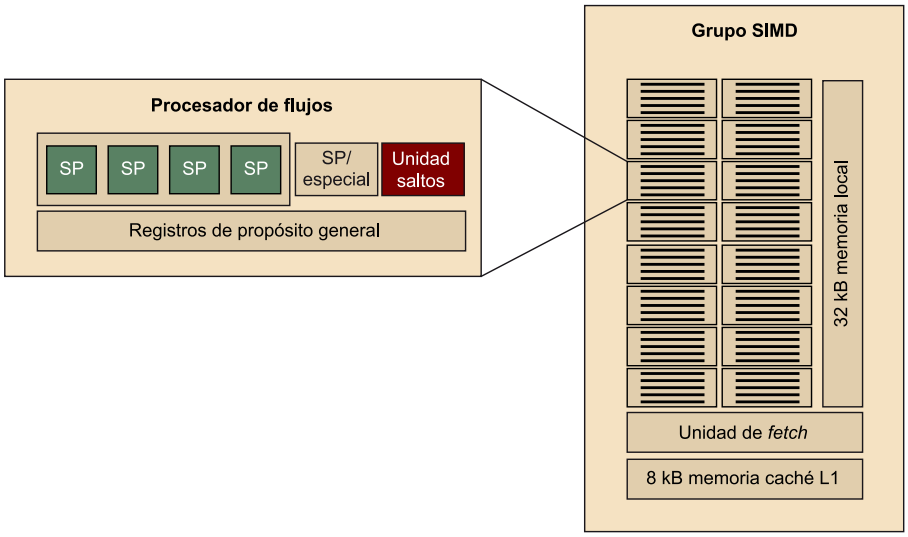
Una implementación de este tipo de arquitectura se encuentra en la familia Evergreen. La figura 17 muestra un diagrama de bloques de la arquitectura Evergreen. Esta está compuesta por un conjunto de grupos SIMD que, al mismo tiempo, están formados por dieciséis unidades de flujos, cada una con cinco núcleos, tal como muestra el detalle de la figura 18.

Figura 17. Diagrama de bloques de la arquitectura Evergreen de AMD



Teniendo en cuenta el ejemplo de arquitectura Evergreen de la tabla 2, podemos contar veinte grupos SIMD, cada uno con dieciséis unidades de flujos, que están formados por cinco núcleos. Todo ello suma un total de 16.000 núcleos, cuya gestión eficaz es un reto importante tanto desde el punto de vista del hardware como a la hora de programar las aplicaciones (que deben proporcionar el nivel de paralelismo suficiente).

Figura 18. Diagrama de bloques de los grupos SIMD de la arquitectura Evergreen de AMD



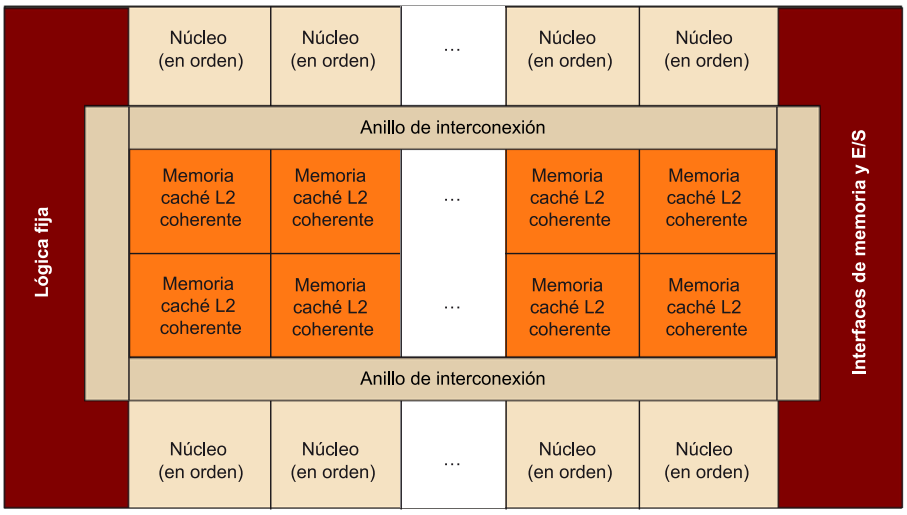
Si consideramos el caso anterior, con un grupo de cuatro núcleos podríamos ejecutar, por ejemplo, en un único ciclo de reloj:

- cuatro operaciones MUL+ADD (fusionadas) de 32 bits en coma flotante;
- dos operaciones MUL o ADD de 64 bits de doble precisión;
- una operación MUL+ADD (fusionadas) de 64 bits de doble precisión;
- cuatro operaciones MUL o ADD+ enteras de 24 bits.

3.3. Arquitectura Intel Larrabee

Larrabee es una arquitectura altamente paralela basada en núcleos que implementan una versión extendida de x86 con operaciones vectoriales (SIMD) y algunas instrucciones escalares especializadas. La figura 19 muestra un esquema de esta arquitectura. Una de las principales características es que la memoria caché de nivel 2 (L2) es coherente entre todos los núcleos.

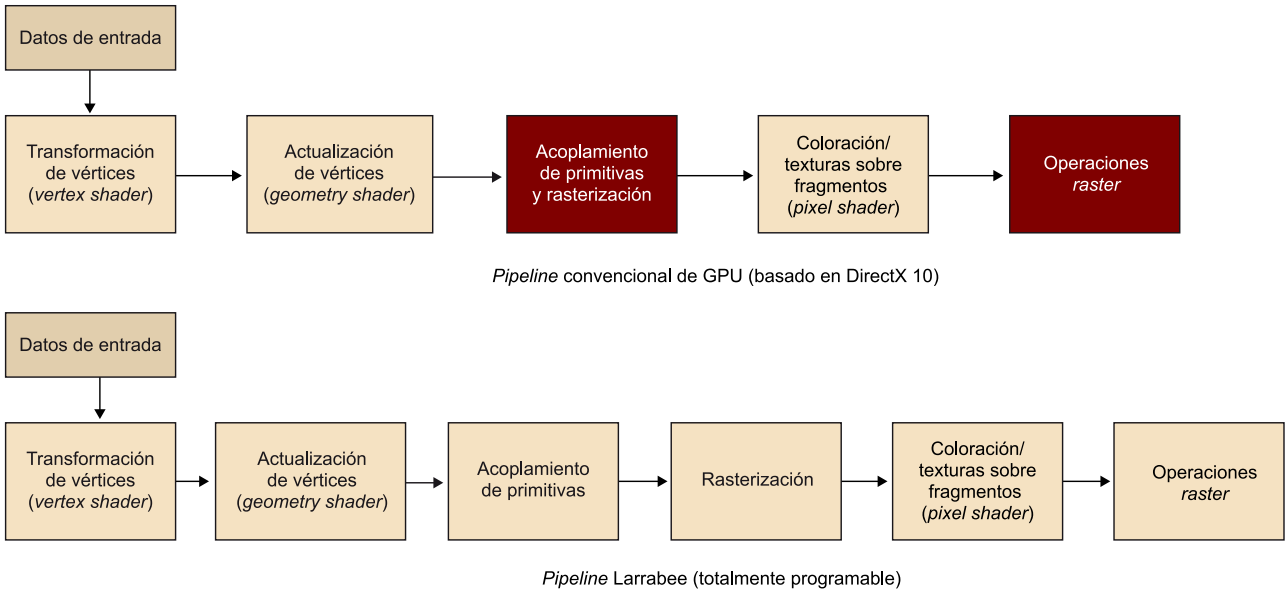
Figura 19. Esquema de la arquitectura Larrabee



La cantidad de núcleos y la cantidad y tipo de coprocesadores y bloques de E/S son dependientes de la implementación.

El hecho de disponer de coherencia de memoria caché y de una arquitectura tipo x86 hace que esta arquitectura sea más flexible que otras arquitecturas basadas en GPU. Entre otras cuestiones, el *pipeline* gráfico de la arquitectura Larrabee es totalmente programable, tal como se muestra en la figura 20.

Figura 20. Comparativa entre el *pipeline* convencional de GPU y el de la arquitectura Larrabee

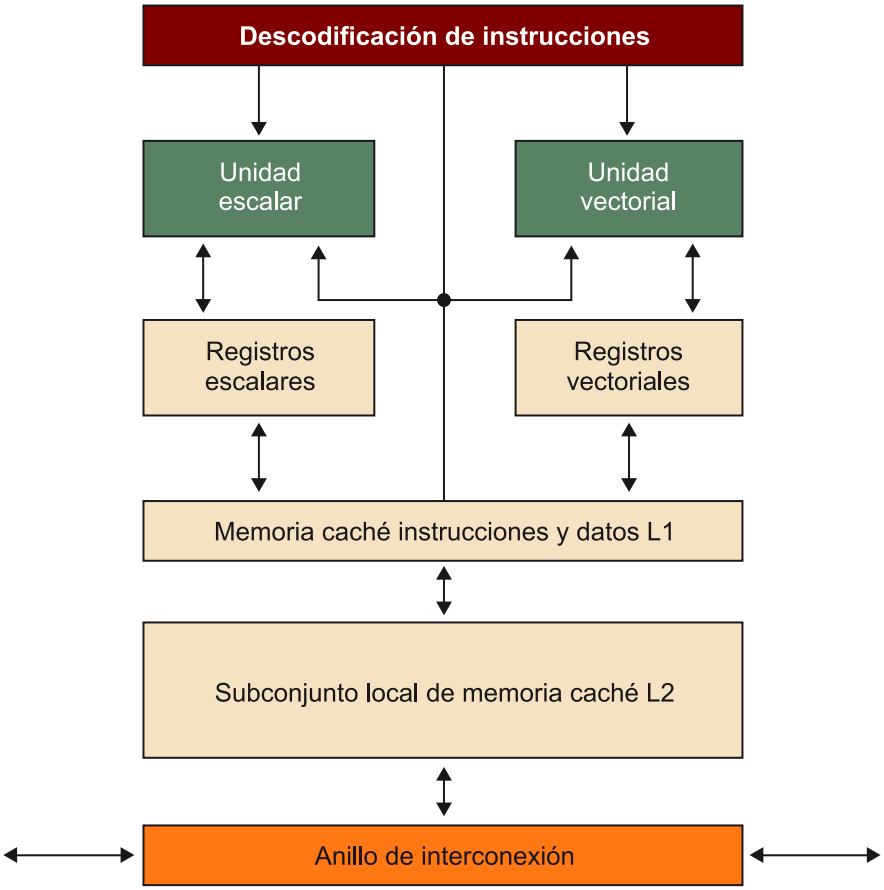


Los elementos no ensombrecidos de la figura son las unidades programables.

Los diferentes núcleos se comunican mediante una red de interconexión de un alto ancho de banda con lógica fija, interfaces de entrada/salida (E/S) y de alguna otra lógica de E/S según la implementación concreta (por ejemplo, una implementación como GPU incorporaría soporte para bus PCIe).

La figura 21 muestra un esquema de un núcleo con la interconexión de red del chip mismo asociada y su subconjunto de memoria caché L2. El descodificador de instrucciones soporta el conjunto de instrucciones x86 del procesador Pentium estándar y algunas otras instrucciones nuevas. Para simplificar el diseño, las unidades escalar y vectorial utilizan diferentes conjuntos de registros. Los datos que se transfieren entre estas unidades se escriben en la memoria y después se leen otra vez de la memoria caché L1. La memoria caché L1 permite accesos muy rápidos desde las unidades escalar y vectorial; por lo tanto, de alguna manera la memoria caché L1 se puede ver como una especie de conjunto de registros.

Figura 21. Diagrama de bloque de los núcleos de la arquitectura Larrabee



La memoria caché L2 se divide en diferentes subconjuntos independientes, uno para cada núcleo. Cada núcleo tiene un camino de acceso rápido y directo a su subconjunto local de la memoria caché L2. Los datos que se leen en un núcleo se almacenan en su subconjunto de memoria caché L2 y se puede acceder a ella a la vez que otros núcleos acceden a su subconjunto de memoria caché L2. Los datos que escribe un núcleo se almacenan en la memoria caché L2 y, si es necesario, también se actualizan otros subconjuntos de memoria caché L2 para mantener su consistencia. La red en forma de anillo garantiza la coherencia para datos compartidos. El tamaño de la memoria caché L2 es bastante considerable (por ejemplo, en torno a los 256 kB), lo que hace que el procesamiento gráfico se pueda aplicar a un conjunto de datos bastante grande. En orden al procesamiento de propósito general, esta característica es especialmente positiva, ya que facilita la implementación eficiente de ciertas aplicaciones.

El *pipeline* escalar de la arquitectura Larrabee se deriva del procesador Pentium P54C, que dispone de un *pipeline* de ejecución bastante sencillo. Aun así, la arquitectura Larrabee tiene algunas características más avanzadas que el P54C, como por ejemplo el apoyo para multiflujo, extensiones de 64 bits y un *prefetching* más sofisticado. Los núcleos también disponen de algunas instrucciones

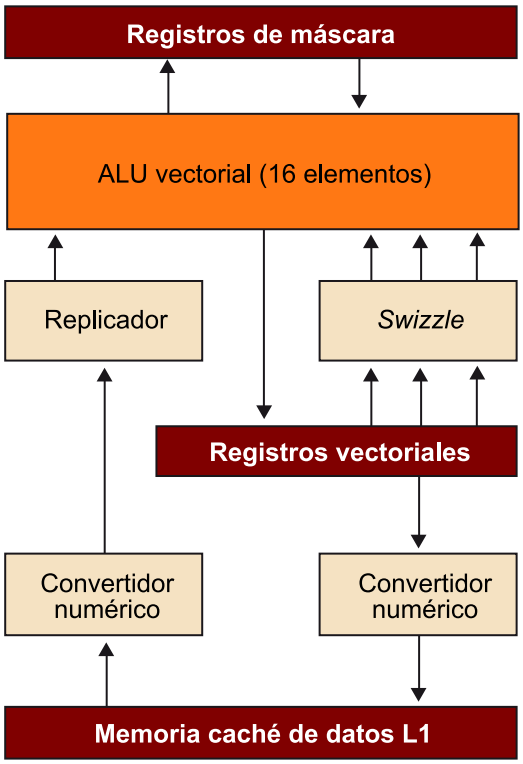
escalares añadidas, pero son compatibles con todo el conjunto de instrucciones del Pentium x86; por lo tanto, pueden ejecutar códigos existentes, como núcleos de sistemas operativos y aplicaciones.

Asimismo, se añaden algunas instrucciones y modalidades nuevas para permitir el control de la memoria caché de manera explícita. Por ejemplo, incluye instrucciones específicas para hacer *prefetching* de datos en la memoria caché L1 o L2 y también modalidades para reducir la prioridad de una línea de memoria caché.

Sincronizar el acceso de varios flujos de un mismo núcleo a la memoria compartida no es demasiado costoso. Los flujos de un mismo núcleo comparten la misma memoria caché local L1; por lo tanto, basta con hacer una simple lectura en un semáforo atómico para implementar la sincronización. Sincronizar el acceso entre diferentes núcleos es algo más costoso, ya que requiere *locks* entre núcleos (lo que es un problema típico en el diseño de multiprocesadores). Hay que tener en cuenta que Larrabee soporta cuatro flujos de ejecución con conjuntos de registros independientes para cada flujo.

La arquitectura Larrabee dispone de una unidad vectorial (VPU, *vector processing unit*) de elementos que ejecuta instrucciones tanto con enteros como en coma flotante de precisión simple y doble. Esta unidad vectorial, junto con sus registros, ocupa aproximadamente un tercio de la superficie del procesador, pero proporciona mejor rendimiento con enteros y coma flotante. La figura 22 muestra el diagrama de bloques de la VPU con la memoria caché L1.

Figura 22. Diagrama de bloque de la unidad vectorial de la arquitectura Larrabee



Las instrucciones vectoriales permiten hasta tres operandos de origen, de los que uno puede venir directamente de la memoria caché L1. Tal como hemos visto antes, si los datos ya están en la memoria caché, entonces la memoria caché L1 es como un conjunto de registros adicionales.

La etapa siguiente consiste en alinear los datos de los registros y memoria con las líneas correspondientes de la VPU (*swizzle*). Esta es una operación típica tanto en el proceso de datos gráficos como no gráficos para aumentar la eficiencia de la memoria caché. También implementa conversión numérica mediante los módulos correspondientes. La VPU dispone de un amplio abanico de instrucciones tanto enteras como en coma flotante. Un ejemplo es la operación aritmética estándar conjunta de multiplicación y suma (MUL+ADD).

La arquitectura Larrabee utiliza un anillo bidireccional que permite comunicar elementos como los núcleos, la memoria caché L2 y otros bloques lógicos entre ellos dentro de un mismo chip. Cuando se utilizan más de dieciséis núcleos, entonces se usan varios anillos más pequeños. El ancho de datos del anillo es de 512 bits por sentido.

La memoria caché L2 está diseñada para proporcionar a cada núcleo un ancho de banda muy grande a direcciones de memoria que otros núcleos no pueden escribir, mediante el subconjunto local de memoria caché L2 del núcleo. Cada núcleo puede acceder a su subconjunto de la memoria caché L2 en paralelo sin comunicarse con otros núcleos. Aun así, antes de asignar una nueva línea a la memoria caché L2, se utiliza el anillo para comprobar que no haya compartición de datos para mantener la coherencia de estos. El anillo de interconexión también proporciona a la memoria caché L2 el acceso a memoria.

4. Modelos de programación para GPGPU

En apartados anteriores, hemos visto que hay interfaces de usuario para la programación gráfica, como OpenGL o Direct3D, y que las GPU se pueden usar para computación de propósito general, a pesar de que con ciertas limitaciones. Dadas las características y la evolución de las GPU, los programadores cada vez necesitan hacer frente a una variedad de plataformas de computación masivamente paralelas y la utilización de modelos de programación estándar es crucial.

En este apartado, vamos a estudiar los principales modelos de programación para GPU orientados a aplicaciones de propósito general, específicamente CUDA y OpenCL, que son los estándares actuales para GPGPU.

4.1. CUDA

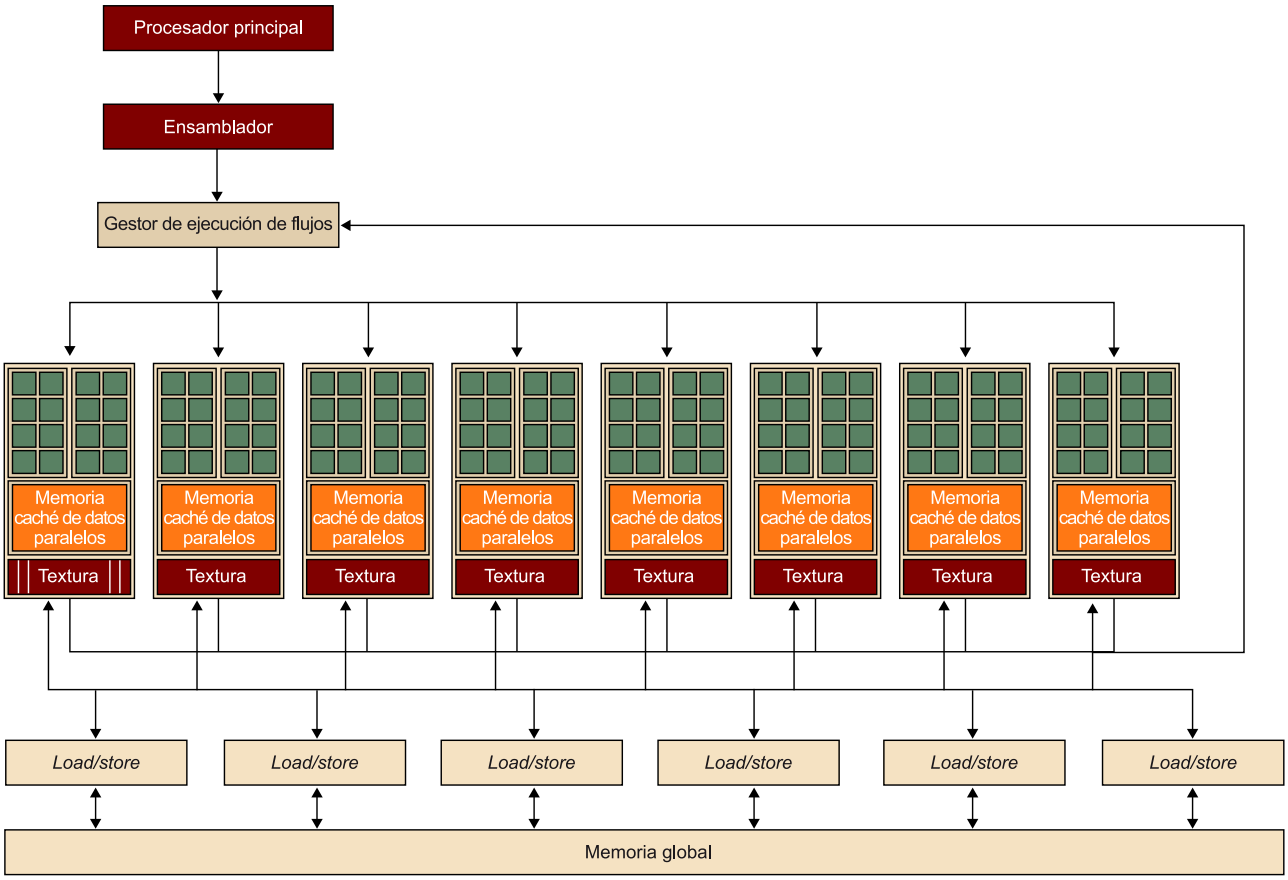
CUDA (*compute unified device architecture*) es una especificación inicialmente de propiedad desarrollada por Nvidia como plataforma para sus productos GPU. CUDA incluye las especificaciones de la arquitectura y un modelo de programación asociado. En este apartado, vamos a estudiar el modelo de programación CUDA; sin embargo, hablaremos de dispositivos compatibles con CUDA para el caso de aquellas GPU que implementan la arquitectura y especificaciones definidas en CUDA.

4.1.1. Arquitectura compatible con CUDA

La figura 23 muestra la arquitectura de una GPU genérica compatible con CUDA. Esta arquitectura está organizada en una serie de multiprocesadores o *streaming multiprocessors* (SM), que tienen una cantidad elevada de flujos de ejecución. En la figura, dos SM forman un bloque, a pesar de que el número de SM por bloque depende de la implementación concreta del dispositivo. Además, cada SM de la figura tiene un número de *streaming processors* (SP) que comparten la lógica de control y la memoria caché de instrucciones.

La GPU tiene una memoria DRAM de tipo GDDR (*graphics double data rate*), que está indicada en la figura 23 como memoria global. Esta memoria GDDR se diferencia de la memoria DRAM de la placa base del computador en el hecho de que en esencia se utiliza para gráficos (*framebuffer*). Para aplicaciones gráficas, esta mantiene las imágenes de vídeo y la información de las texturas. En cambio, para cálculos de propósito general esta funciona como memoria externa con mucho ancho de banda, pero con una latencia algo más elevada que la memoria típica del sistema. Aun así, para aplicaciones masivamente paralelas el ancho de banda más grande compensa la latencia más elevada.

Figura 23. Esquema de la arquitectura de una GPU genérica compatible con CUDA



Notad que esta arquitectura genérica es muy similar a la G80 descrita antes, ya que la G80 la implementa. La arquitectura G80 soporta hasta 768 flujos por SM, lo que suma un total de 12.000 flujos en un único chip.

La arquitectura GT200, posterior a la G80, tiene 240 SP y supera el TFlop de pico teórico de rendimiento. Como los SP son masivamente paralelos, se pueden llegar a utilizar todavía más flujos por aplicación que la G80. La GT200 soporta 1.024 flujos por SM y, en total, suma en torno a 30.000 flujos por chip. Por lo tanto, la tendencia muestra claramente que el nivel de paralelismo soportado por las GPU está aumentando rápidamente. Así pues, será muy importante intentar explotar este nivel tan elevado de paralelismo cuando se desarrollen aplicaciones de propósito general para GPU.

4.1.2. Entorno de programación

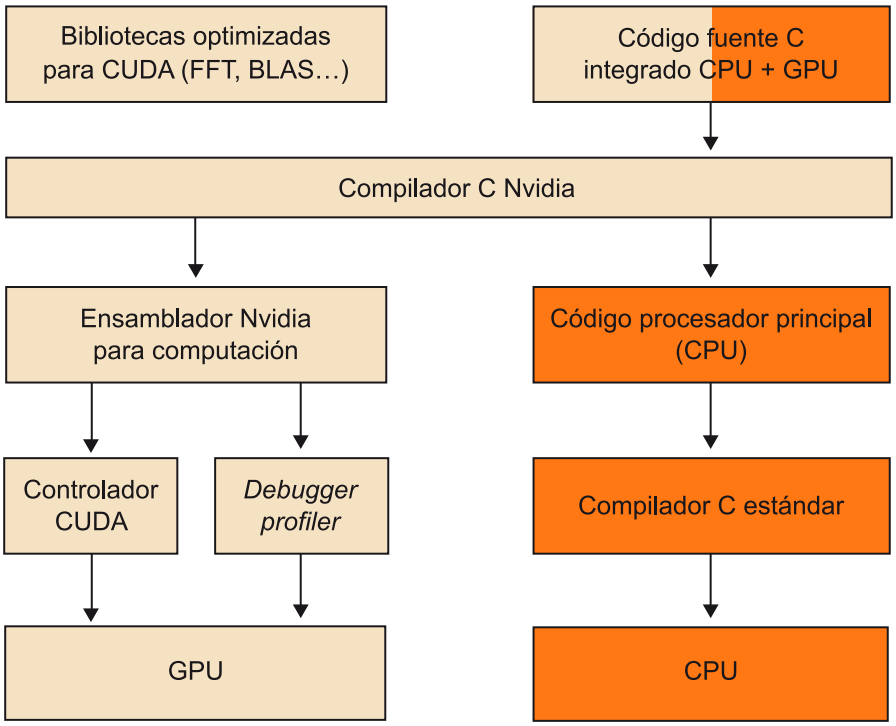
CUDA se desarrolló para aumentar la productividad en el desarrollo de aplicaciones de propósito general para GPU. Desde el punto de vista del programador, el sistema está compuesto por un procesador principal (*host*), que es una CPU tradicional, como por ejemplo un procesador de arquitectura Intel, y por uno o más dispositivos (*devices*), que son GPU.

CUDA pertenece al modelo SIMD; por lo tanto, está pensado para explotar el paralelismo a nivel de datos. Esto quiere decir que un conjunto de operaciones aritméticas se puede ejecutar sobre un conjunto de datos de manera simultánea. Afortunadamente, muchas aplicaciones tienen partes con un nivel muy elevado de paralelismo a nivel de datos.

Un programa en CUDA consiste en una o más fases que pueden ser ejecutadas o bien en el procesador principal (CPU) o bien en el dispositivo GPU. Las fases en las que hay muy poco o ningún paralelismo a nivel de datos se implementan en el código que se ejecutará en el procesador principal y las fases con un nivel de paralelismo a nivel de datos elevado se implementan en el código que se ejecutará en el dispositivo.

Tal como muestra la figura 24, el compilador de NVIDIA (`nvcc`) se encarga de proporcionar la parte del código correspondiente al procesador principal y al dispositivo durante el proceso de compilación. El código correspondiente al procesador principal es simplemente código ANSI C, que se compila mediante el compilador de C estándar del procesador principal, como si fuera un programa para CPU convencional. El código correspondiente al dispositivo también es ANSI C, pero con extensiones que incluyen palabras clave para poder definir funciones que tratan los datos en paralelo. Estas funciones se denominan ***kernels***.

Figura 24. Esquema de bloques del entorno de compilación de CUDA



El código del dispositivo se vuelve a compilar con `nvcc` y entonces ya se puede ejecutar en el dispositivo GPU. En situaciones en las que no hay ningún dispositivo disponible o bien el *kernel* es más apropiado para una CPU, también se pueden ejecutar los *kernels* en una CPU convencional mediante herramien-

tas de emulación que proporciona la plataforma CUDA. La figura 25 muestra todas las etapas del proceso de compilación y la tabla 3 describe cómo el compilador `nvcc` interpreta los diferentes tipos de archivos de entrada.

Figura 25. Pasos en la compilación de código CUDA, desde *.cu* hasta *.cu.c*

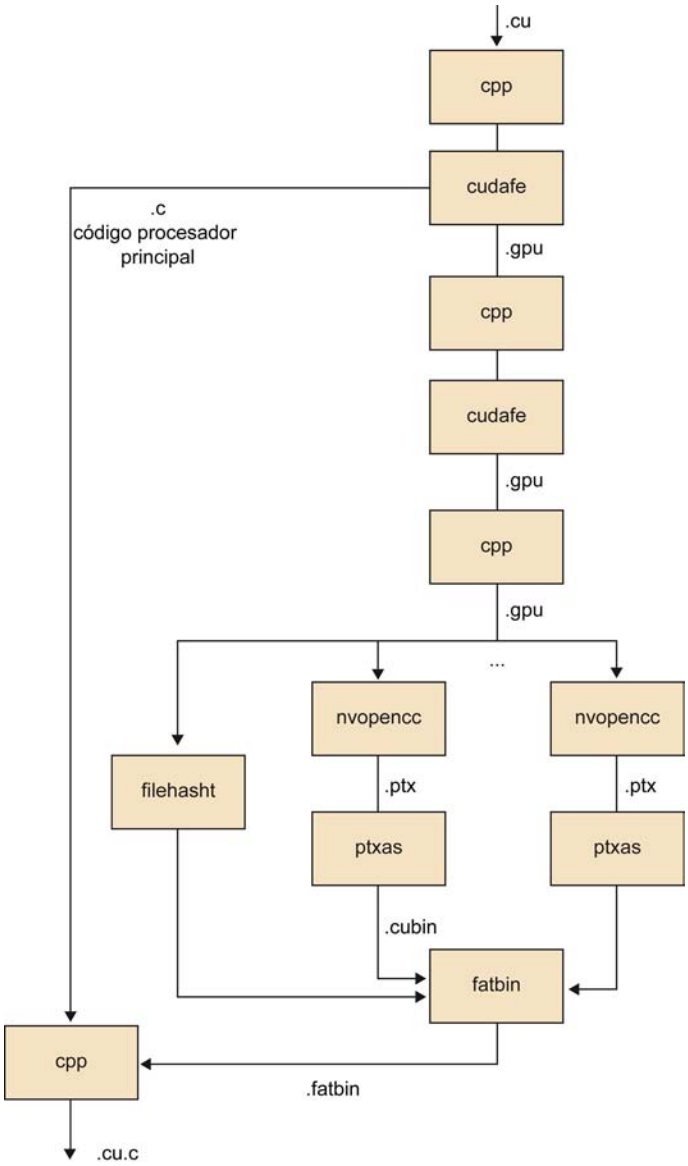


Tabla 3. Interpretación de `nvcc` de los archivos de entrada

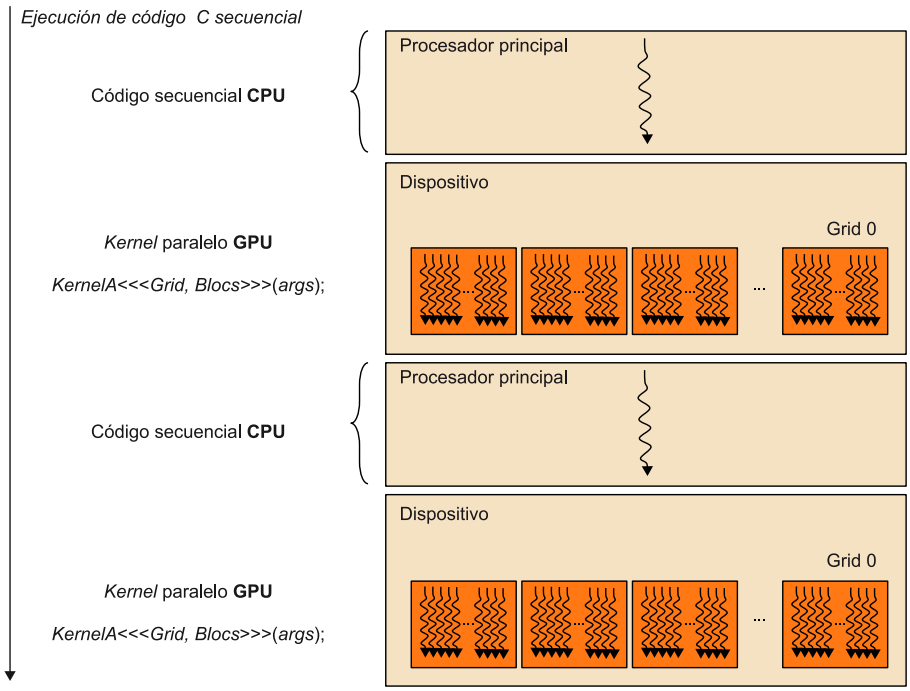
.cu	Código fuente CUDA que contiene tanto el código del procesador principal como las funciones del dispositivo
.cup	Código fuente CUDA preprocesado que contiene tanto el código del procesador principal como las funciones del dispositivo
.c	Archivo de código fuente C
.cc, .cxx, .cpp	Archivo de código fuente C++
.gpu	Archivo intermedio *gpu*
.ptx	Archivo ensamblador intermedio *ptx*
.o, .obj	Archivo de objeto

.a, .lib	Archivo de biblioteca
.res	Archivo de recurso
.so	Archivo de objeto compartido

Para explotar el paralelismo a nivel de datos, los *kernels* tienen que generar una cantidad de flujos de ejecución bastante elevada (por ejemplo, en torno a decenas o centenares de miles de flujos). Debemos tener en cuenta que **los flujos de CUDA son mucho más ligeros que los flujos de CPU**. De hecho, podremos asumir que, para generar y planificar estos flujos, solo necesitaremos unos pocos ciclos de reloj debido al soporte de hardware, en contraste con los flujos convencionales para CPU, que normalmente requieren miles de ciclos.

La ejecución de un programa típico CUDA se muestra en la figura 26. La ejecución empieza con la ejecución en el procesador principal (CPU). Cuando se invoca un *kernel*, la ejecución se mueve hacia el dispositivo (GPU), donde se genera un número muy elevado de flujos. El conjunto de todos estos flujos que se generan cuando se invoca un *kernel* se denomina *grid*. En la figura 26, se muestran dos *grids* de flujos. La definición y organización de estos *grids* las estudiaremos más adelante. Un *kernel* finaliza cuando todos sus flujos finalizan la ejecución en el *grid* correspondiente. Una vez finalizado el *kernel*, la ejecución del programa continúa en el procesador principal hasta que se invoca otro *kernel*.

Figura 26. Etapas en la ejecución de un programa típico CUDA



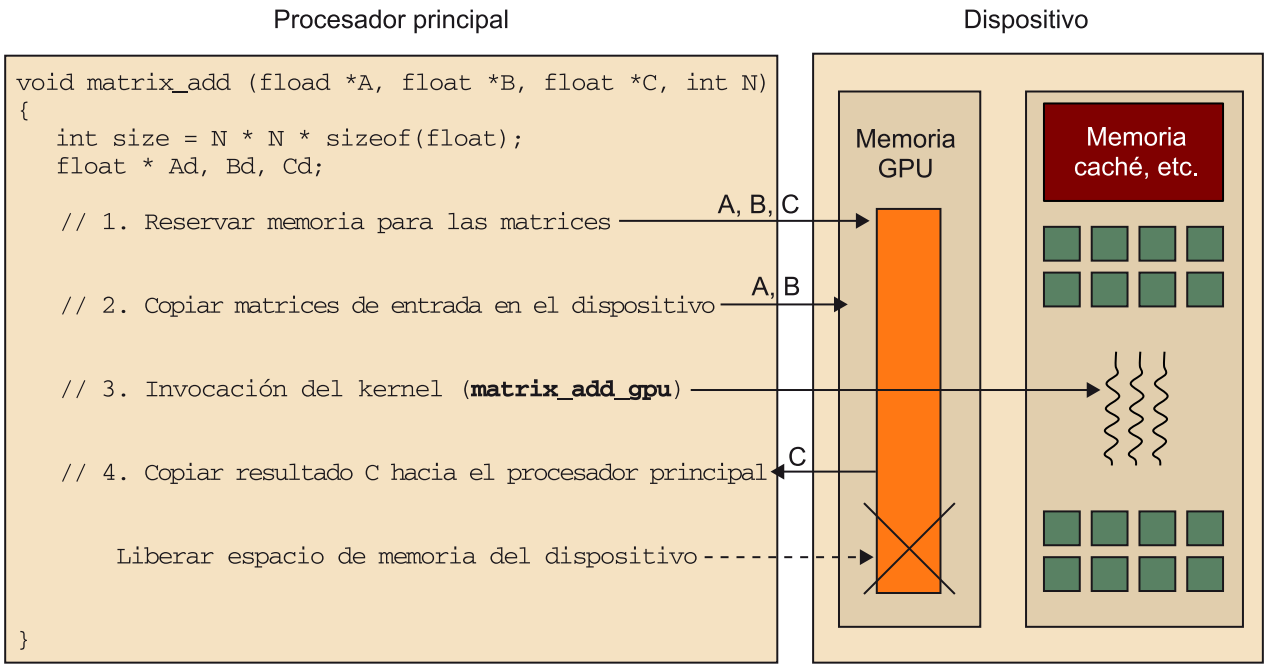
4.1.3. Modelo de memoria

Es importante destacar que **la memoria del procesador principal y del dispositivo son espacios de memoria completamente separados**. Esto refleja la realidad por la que los dispositivos son típicamente tarjetas que tienen su propia memoria DRAM. Para ejecutar un kernel en el dispositivo GPU, normalmente hay que seguir los pasos siguientes:

- Reservar memoria en el dispositivo (paso 1 de la figura 27).
- Transferir los datos necesarios desde el procesador principal hasta el espacio de memoria asignado al dispositivo (paso 2 de la figura 27).
- Invocar la ejecución del *kernel* en cuestión (paso 3 de la figura 27).
- Transferir los datos con los resultados desde el dispositivo hacia el procesador principal y liberar la memoria del dispositivo (si ya no es necesaria), una vez finalizada la ejecución del *kernel* (paso 4 de la figura 27).

El entorno CUDA proporciona una interfaz de programación que simplifica estas tareas al programador. Por ejemplo, basta con especificar qué datos hay que transferir desde el procesador principal hasta el dispositivo y viceversa.

Figura 27. Esquema de los pasos para la ejecución de un *kernel* en una GPU



Durante todo este apartado, vamos a utilizar el mismo programa de ejemplo que realiza la suma de dos matrices. El código 4.1 muestra el código correspondiente a la implementación secuencial en C estándar de la suma de matrices para una arquitectura de tipo CPU. Notad que esta implementación es ligeramente diferente a la del código 3.1.

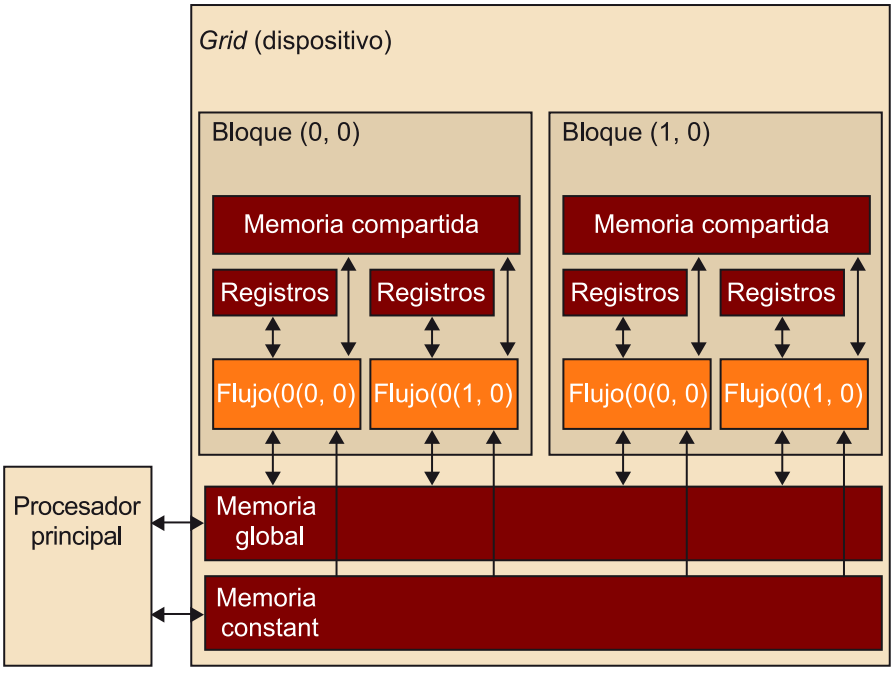
```
void matrix_add_cpu (fload *A, float *B, float *C, int N)
{
    int i, j, index;
    for (i=0; i<N; i++){
            for (j=0; j<N; j++){
                index = i+j*N;
                C[index] = C[index] + B[index];
            }
        }
    }

int main(){
    matrix_add_cpu(a, b, c, N);
}
```

Código 4.1. Implementación secuencial en C estándar de la suma de matrices para una arquitectura de tipo CPU

La figura 28 muestra el modelo de memoria de CUDA expuesto al programador en términos de asignación, transferencia y utilización de los diferentes tipos de memoria del dispositivo. En la parte inferior de la figura, se encuentran las memorias de tipo global y constante. Estas memorias son aquellas a las que el procesador principal puede transferir datos de manera bidireccional, por *grid*. Desde el punto de vista del dispositivo, se puede acceder a diferentes tipos de memoria con los modos siguientes:

- acceso de lectura/escritura a la memoria global, por *grid*;
- acceso solo de lectura a la memoria constante, por *grid*;
- acceso de lectura/escritura a los registros, por flujo;
- acceso de lectura/escritura a la memoria local, por flujo;
- acceso de lectura/escritura a la memoria compartida, por bloque;
- acceso de lectura/escritura.

Figura 28. Modelo de memoria de CUDA



La interfaz de programación para asignar y liberar memoria global del dispositivo consiste en dos funciones básicas: `cudaMalugar()` y `cudaFree()`. La función `cudaMalugar()` se puede llamar desde el código del procesador principal para asignar un espacio de memoria global para un objeto. Como habréis observado, esta función es muy similar a la función `malugar()` de la biblioteca de C estándar, ya que CUDA es una extensión del lenguaje C y pretende mantener las interfaces cuanto más similares a las originales mejor.

La función `cudaMalugar()` tiene dos parámetros. El primer parámetro es la dirección del puntero hacia el objeto una vez se haya asignado el espacio de memoria. Este puntero es genérico y no depende de ningún tipo de objeto; por lo tanto, se tendrá que hacer *cast* en tipo `(void **)`. El segundo parámetro es el tamaño del objeto que se quiere asignar en bytes. La función `cudaFree()` libera el espacio de memoria del objeto indicado como parámetro de la memoria global del dispositivo. El código 4.2 muestra un ejemplo de cómo podemos utilizar estas dos funciones. Tras hacer el `cudaMalugar(), Matriz` apunta a una región de la memoria global del dispositivo que se le ha asignado.

```
float *Matriz;

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

Código 4.2. Ejemplo de utilización de las llamadas de asignación y liberación de memoria del dispositivo en CUDA

Una vez que un programa ha asignado la memoria global del dispositivo para los objetos o estructuras de datos del programa, se pueden transferir los datos que serán necesarios para la computación desde el procesador principal hacia el dispositivo. Esto se hace mediante la función `cudaMemcpy()`, que permite transferir datos entre memorias. Hay que tener en cuenta que la transferencia es asíncrona.

La función `cudaMemcpy()` tiene cuatro parámetros. El primero es un puntero en la dirección de destino donde se tienen que copiar los datos. El segundo parámetro apunta a los datos que se tienen que copiar. El tercer parámetro especifica el número de bytes que se tienen que copiar. Finalmente, el cuarto parámetro indica el tipo de memoria involucrado en la copia, que puede ser uno de los siguientes:

- `cudaMemcpyHostToHost`: de la memoria del procesador principal hacia la memoria del mismo procesador principal.
- `cudaMemcpyHostToDevice`: de la memoria del procesador principal hacia la memoria del dispositivo.
- `cudaMemcpyDeviceToHost`: de la memoria del dispositivo hacia la memoria del procesador principal.

- `cudaMemcpyDeviceToDevice`: de la memoria del dispositivo hacia la memoria del dispositivo.

Hay que tener en cuenta que esta función se puede utilizar para copiar datos de la memoria de un mismo dispositivo, pero no entre diferentes dispositivos. El código 4.3 muestra un ejemplo de transferencia de datos entre procesador principal y dispositivo basado en el ejemplo de la figura 27.

```
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);
    ...
```

Código 4.3. Ejemplo de transferencia de datos entre procesador principal y dispositivo

Además de los mecanismos de asignación de memoria y transferencia de datos, CUDA también soporta diferentes tipos de variables. Los diferentes tipos de variables que utilizan los diversos tipos de memoria son utilizados en varios ámbitos y tendrán ciclos de vida diferentes, tal como resume la tabla 4.

Tabla 4. Diferentes tipos de variables en CUDA

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

4.1.4. Definición de *kernels*

El código que se ejecuta en el dispositivo (*kernel*) es la función que ejecutan los diferentes flujos durante la fase paralela, cada uno en el rango de datos que le corresponde. Hay que recordar que CUDA sigue el modelo SPMD (*single-program multiple-data*) y, por lo tanto, todos los flujos ejecutan el mismo código. El código 4.4 muestra la función o *kernel* de la suma de matrices y su llamada.

```
__global__ matrix_add_gpu (float *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);
}
```

Código 4.4. Implementación de la suma de matrices en CUDA

Podemos observar la utilización de la palabra clave específica de CUDA `__global__` ante la declaración de `matrix_add_gpu()`. Esta palabra clave indica que esta función es un *kernel* y que hay que llamarlo desde el procesador principal para generar el *grid* de flujos que ejecutará el *kernel* en el dispositivo. Además de `__global__`, hay dos palabras clave más que se pueden utilizar ante la declaración de una función:

1) La palabra clave `__device__` indica que la función declarada es una función CUDA de dispositivo. Una función de dispositivo se ejecuta únicamente en un dispositivo CUDA y solo se puede llamar desde un *kernel* o desde otra función de dispositivo. Estas funciones no pueden tener ni llamadas recursivas ni llamadas indirectas a funciones mediante punteros.

2) La palabra clave `__host__` indica que la función es una función de procesador principal, es decir, una función simple de C que se ejecuta en el procesador principal y, por lo tanto, que puede ser llamada desde cualquier función de procesador principal. Por defecto, todas las funciones en un programa CUDA son funciones de procesador principal, si es que no se especifica ninguna palabra clave en la definición de la función.

Las palabras clave `__host__` y `__device__` se pueden utilizar simultáneamente en la declaración de una función. Esta combinación hace que el compilador genere dos versiones de la misma función, una que se ejecuta en el procesador principal y que solo se puede llamar desde una función de procesador principal y otra que se ejecuta en el dispositivo y que solo se puede llamar desde el dispositivo o función de *kernel*.

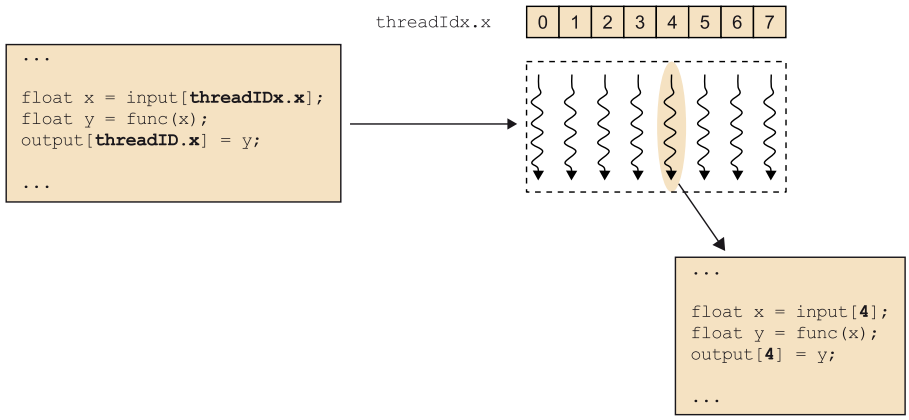
4.1.5. Organización de flujos

En CUDA, un *kernel* se ejecuta mediante un conjunto de flujos (por ejemplo, un vector o una matriz de flujos). Como todos los flujos ejecutan el mismo *kernel* (modelo SIMT, *single instruction multiple threads*), se necesita un mecanismo que permita diferenciarlos y así poder asignar la parte correspondiente de

los datos a cada flujo de ejecución. CUDA incorpora palabras clave para hacer referencia al índice de un flujo (por ejemplo, `threadIdx.x` y `threadIDx.y` si tenemos en cuenta dos dimensiones).

La figura 29 muestra que el *kernel* hace referencia al identificador de flujo y que, durante la ejecución en cada uno de los flujos, el identificador se sustituye por el valor que le corresponde. Por lo tanto, las variables `threadIdx.x` y `threadIDx.y` tendrán diferentes valores para cada uno de los flujos de ejecución. Notad que las coordenadas reflejan la organización multidimensional de los flujos de ejecución, a pesar de que el ejemplo de la figura 29 solo hace referencia a una dimensión (`threadIdx.x`).

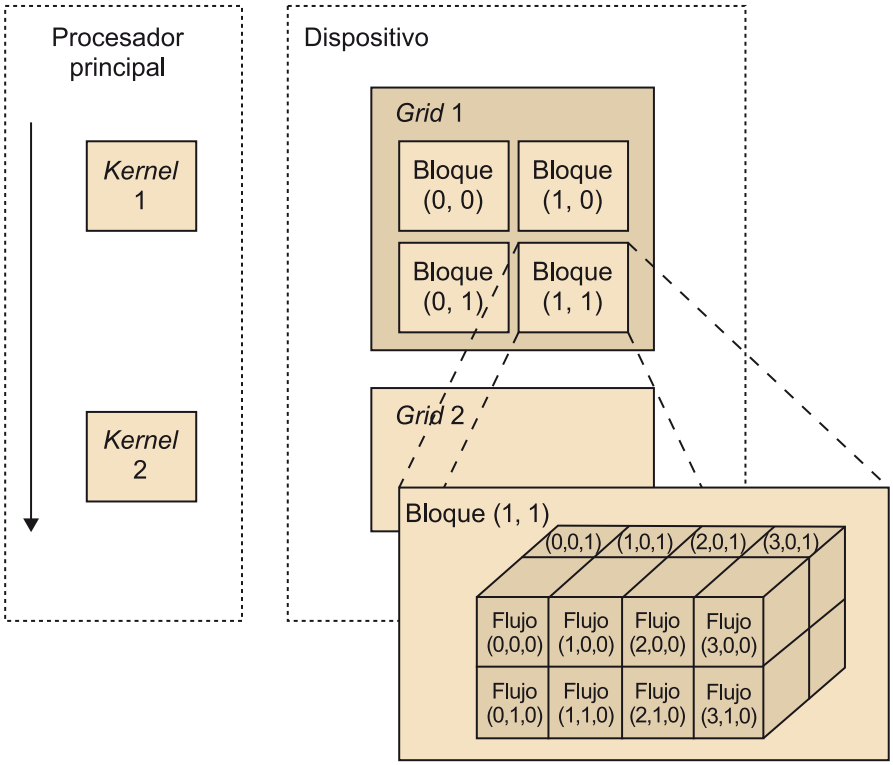
Figura 29. Ejecución de un *kernel* CUDA en un vector de flujos



Normalmente, los *grids* que se utilizan en CUDA están formados por muchos flujos (en torno a miles o incluso millones de flujos). Los flujos de un *grid* están organizados en una jerarquía de dos niveles, tal como se puede ver en la figura 30. También se puede observar que el `kernel 1` crea el `Grid 1` para la ejecución. El nivel superior de un *grid* consiste en uno o más bloques de flujos. Todos los bloques de un *grid* tienen el mismo número de flujos y deben estar organizados del mismo modo. En la figura 30, el `Grid 1` está compuesto por cuatro bloques y está organizado como una matriz de 2 × 2 bloques.

Cada bloque de un *grid* tiene una coordenada única en un espacio de dos dimensiones mediante las palabras clave `blockIdx.x` y `blockIdx.y`. Los bloques se organizan en un espacio de tres dimensiones con un máximo de 512 flujos de ejecución. Las coordenadas de los flujos de ejecución en un bloque se identifican con tres índices (`threadIdx.x`, `threadIdx.y` y `threadIdx.z`), a pesar de que no todas las aplicaciones necesitan utilizar las tres dimensiones de cada bloque de flujos. En la figura 30, cada bloque está organizado en un espacio de 4 × 2 × 2 flujos de ejecución.

Figura 30. Ejemplo de organización de los flujos de un *grid* en CUDA



Cuando el procesador principal hace una llamada a un *kernel*, se tienen que especificar las dimensiones del *grid* y de los bloques de flujos mediante parámetros de configuración. El primer parámetro especifica las dimensiones del *grid* en términos de número de bloques y el segundo especifica las dimensiones de cada bloque en términos de número de flujos. Ambos parámetros son de tipo `dim3`, que en esencia es una estructura de C con tres campos (`x, y, z`) de tipo entero sin signo. Como los *grids* son grupos de bloques en dos dimensiones, el tercer campo de parámetros de configuración del *grid* se ignora (cualquier valor será válido). El código 4.5 muestra un ejemplo en el que dos variables de estructuras de tipo `dim3` definen el *grid* y los bloques del ejemplo de la figura 30.

```
// 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);
```

Código 4.5. Ejemplo de definición de un *grid* y bloques

CUDA también ofrece un mecanismo para sincronizar los flujos de un mismo bloque mediante la función de tipo *barrier* `__syncthreads()`. Cuando se llama a la función `__syncthreads()`, el flujo que la ejecuta quedará bloqueado hasta que todos los flujos de su bloque lleguen a ese mismo punto. Esto sirve para asegurar que todos los flujos de un bloque han completado una fase antes de pasar a la siguiente.

4.2. OpenCL

OpenCL es una interfaz estándar, abierta, libre y multiplataforma para la programación paralela. La principal motivación para el desarrollo de OpenCL fue la necesidad de simplificar la tarea de programación portátil y eficiente de la creciente cantidad de plataformas heterogéneas, como CPU multinúcleo, GPU o incluso sistemas incrustados. OpenCL fue concebida por Apple, a pesar de que la acabó desarrollando el grupo Khronos, que es el mismo que impulsó OpenGL y su responsable.

OpenCL consiste en tres partes: la especificación de un lenguaje multiplataforma, una interfaz a nivel de entorno de computación y una interfaz para coordinar la computación paralela entre procesadores heterogéneos. OpenCL utiliza un subconjunto de C99 con extensiones para el paralelismo y utiliza el estándar de representación numérica IEEE 754 para garantizar la interoperabilidad entre plataformas.

Existen muchas similitudes entre OpenCL y CUDA, aunque OpenCL tiene un modelo de gestión de recursos más complejo, ya que soporta múltiples plataformas y portabilidad entre diferentes fabricantes. OpenCL soporta modelos de paralelismo tanto a nivel de datos como a nivel de tareas. En este subapartado, nos vamos a centrar en el modelo de paralelismo a nivel de datos, que es equivalente al de CUDA.

4.2.1. Modelo de paralelismo a nivel de datos

Del mismo modo que en CUDA, un programa en OpenCL está formado por dos partes: los *kernels* que se ejecutan en uno o más dispositivos y un programa en el procesador principal que invoca y controla la ejecución de los *kernels*. Cuando se hace la invocación de un *kernel*, el código se ejecuta en tareas elementales (*work items*) que corresponden a los flujos de CUDA. Las tareas elementales y los datos asociados a cada tarea elemental se definen a partir del rango de un espacio de índices de dimensión *N* (NDRanges). Las tareas elementales forman grupos de tareas (*work groups*), que corresponden a los bloques de CUDA. Las tareas elementales tienen un identificador global que es único. Además, los grupos de tareas elementales se identifican dentro del rango de dimensión *N* y, para cada grupo, cada una de las tareas elementales tiene un identificador local, que irá desde 0 hasta el tamaño del grupo 1. Por lo tanto, la combinación del identificador del grupo y del identificador local dentro del grupo también identifica de manera única una tarea elemental. La tabla 5 resume algunas de las equivalencias entre OpenCL y CUDA.

Tabla 5. Algunas correspondencias entre OpenCL y CUDA

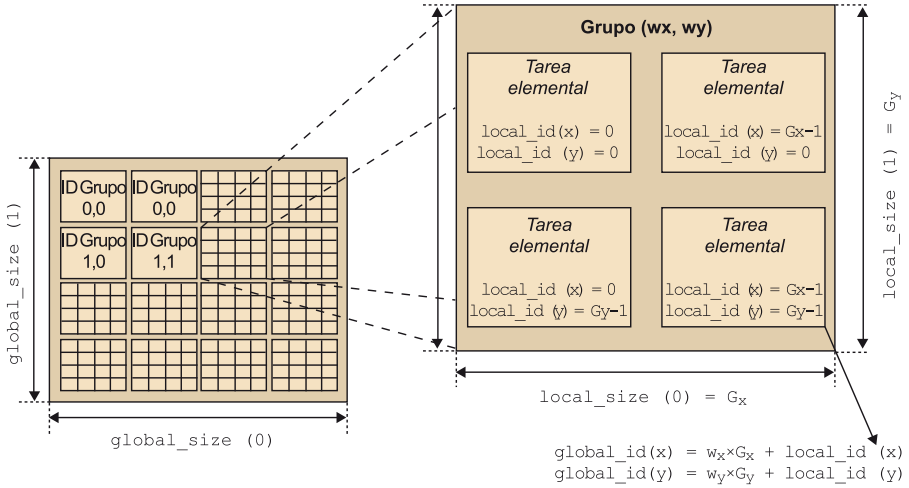
OpenCL	CUDA
Kernel	*Kernel*
Programa procesador principal	Programa procesador principal

OpenCL	CUDA
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

La figura 31 muestra el modelo de paralelismo a nivel de datos de OpenCL. El rango de dimensión *N* (equivalente al *grid* en CUDA) contiene las tareas elementales. En el ejemplo de la figura, el *kernel* utiliza un rango de dos dimensiones, mientras que en CUDA cada flujo tiene un valor de `blockIdx` y `threadIdx` que se combinan para obtener e identificar el flujo, en OpenCL disponemos de interfaces para identificar las tareas elementales de las dos maneras que hemos visto. Por un lado, la función `get_global_id()`, dada una dimensión, devuelve el identificador único de tarea elemental a la dimensión especificada. En el ejemplo de la figura, las llamadas `get_global_id(0)` y `get_global_id(1)` devuelven el índice de las tareas elementales a las dimensiones *X* e *Y*, respectivamente. Por el otro lado, la función `get_local_id()`, dada una dimensión, devuelve el identificador de la tarea elemental dentro de su grupo a la dimensión especificada. Por ejemplo, `get_local_id(0)` es equivalente a `threadIdx.x` en CUDA.

OpenCL también dispone de las funciones `get_global_size()` y `get_local_size()` que, dada una dimensión, devuelven la cantidad total de tareas elementales y la cantidad de tareas elementales dentro de un grupo a la dimensión especificada, respectivamente. Por ejemplo, `get_global_size(0)` devuelve la cantidad de tareas elementales, que es equivalente a `gridDim.x*blockDim.x` en CUDA.

Figura 31. Ejemplo de rango de dimensión *N* en el que se pueden observar las tareas elementales, los grupos que forman e identificadores asociados.

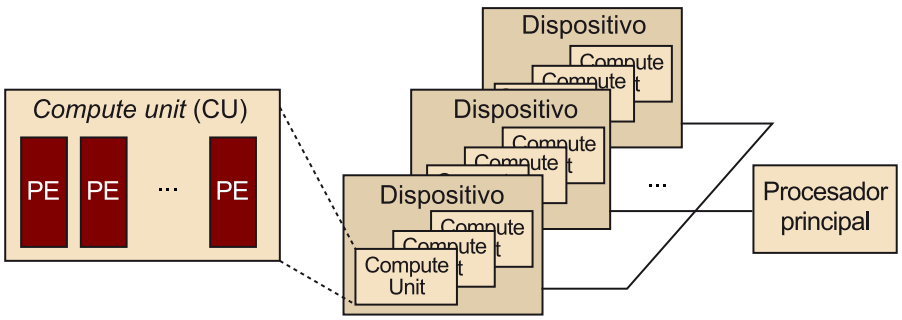


Las tareas elementales dentro del mismo grupo se pueden sincronizar entre ellas utilizando *barriers*, que son equivalentes a `__syncthreads()` de CUDA. En cambio, las tareas elementales de diferentes grupos no se pueden sincronizar entre ellas, excepto si es por la terminación del *kernel* o la invocación de uno nuevo.

4.2.2. Arquitectura conceptual

La figura 32 muestra la arquitectura conceptual de OpenCL, que está formada por un procesador principal (típicamente una CPU que ejecuta el programa principal) conectado a uno o más dispositivos OpenCL. Un dispositivo OpenCL está compuesto por una o más unidades de cómputo o *compute units* (CU), que corresponden a los SM de CUDA. Finalmente, una CU está formada por uno o más elementos de procesamiento o *processing elements* (PE), que corresponden a los SP de CUDA. La ejecución del programa se acabará haciendo en los PE.

Figura 32. Arquitectura conceptual de OpenCL



4.2.3. Modelo de memoria

En un dispositivo OpenCL, hay disponible una jerarquía de memoria que incluye varios tipos de memoria diferentes: global, constante, local y privada.

- La memoria global es la que pueden utilizar todas las unidades de cálculo de un dispositivo.

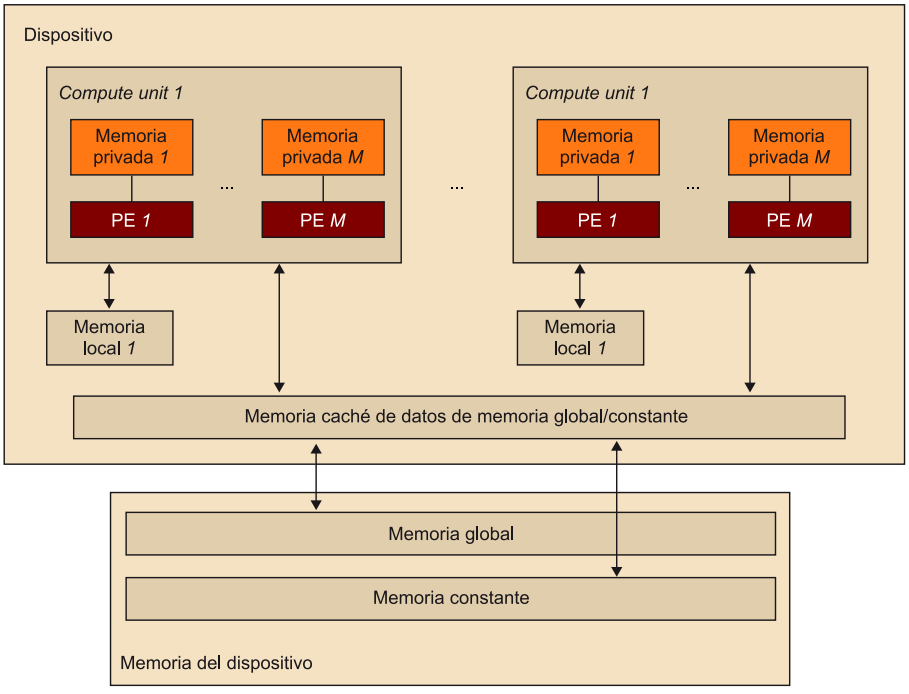
- La memoria constante es la memoria que se puede utilizar para almacenar datos constantes para acceso solo de lectura de todas las unidades de cálculo de un dispositivo durante la ejecución de un *kernel*. El procesador principal es responsable de asignar e iniciar los objetos de memoria que residen en el espacio de memoria.

- La memoria local es la memoria que se puede utilizar para las tareas elementales de un grupo.

- La memoria privada es la que puede utilizar únicamente una unidad de cálculo única. Esto es similar a los registros en una única unidad de cálculo o un único núcleo de una CPU.

Así pues, tanto la memoria global como la constante corresponden a los tipos de memoria con el mismo nombre de CUDA, la memoria local corresponde a la memoria compartida de CUDA y la memoria privada corresponde a la memoria local de CUDA.

Figura 33. Arquitectura y jerarquía de memoria de un dispositivo OpenCL



Para crear y componer objetos en la memoria de un dispositivo, la aplicación que se ejecuta en el procesador principal utiliza la interfaz de OpenCL, ya que los espacios de memoria del procesador principal y de los dispositivos son principalmente independientes el uno del otro. La interacción entre el espacio de memoria del procesador principal y de los dispositivos puede ser de dos tipos: copiando datos explícitamente o bien mapeando/desmapeando regiones de un objeto OpenCL en la memoria. Para copiar datos explícitamente, el

procesador principal envía las instrucciones para transferir datos entre la memoria del objeto y la memoria del procesador principal. Estas instrucciones de transferencia pueden ser o bien bloqueantes o bien no bloqueantes. Cuando se utiliza una llamada bloqueante, se puede acceder a los datos desde el procesador principal con seguridad, una vez la llamada ha finalizado. En cambio, para hacer una transferencia no bloqueante, la llamada a la función de OpenCL finaliza inmediatamente y se saca de la cola, a pesar de que la memoria desde el procesador principal no es segura para ser utilizada. La interacción mapeante/desmapeante de objetos en memoria permite que el procesador principal pueda tener en su espacio de memoria una región correspondiente a objetos OpenCL. Las instrucciones de mapeado/desmapeado también pueden ser bloqueantes o no bloqueantes.

4.2.4. Gestión de *kernels* y dispositivos

Los *kernels* de OpenCL tienen la misma estructura que los de CUDA. Por lo tanto, son funciones que se declaran empezando con la palabra clave `__kernel`, que es equivalente al `__global` de CUDA. El código 4.6 muestra la implementación de nuestro ejemplo de suma de matrices en OpenCL. Fijaos en que los argumentos del *kernel* correspondientes a las matrices están declarados como `__global`, ya que se encuentran en la memoria global y las dos matrices de entrada están también declaradas como `const`, puesto que solo habrá que hacer accesos en modalidad de lectura. En la implementación del cuerpo del *kernel* se utiliza `get_global_id()` en las dos dimensiones de la matriz para definir el índice asociado. Este índice se utiliza para seleccionar los datos que corresponden a cada una de las tareas elementales que instancie el *kernel*.

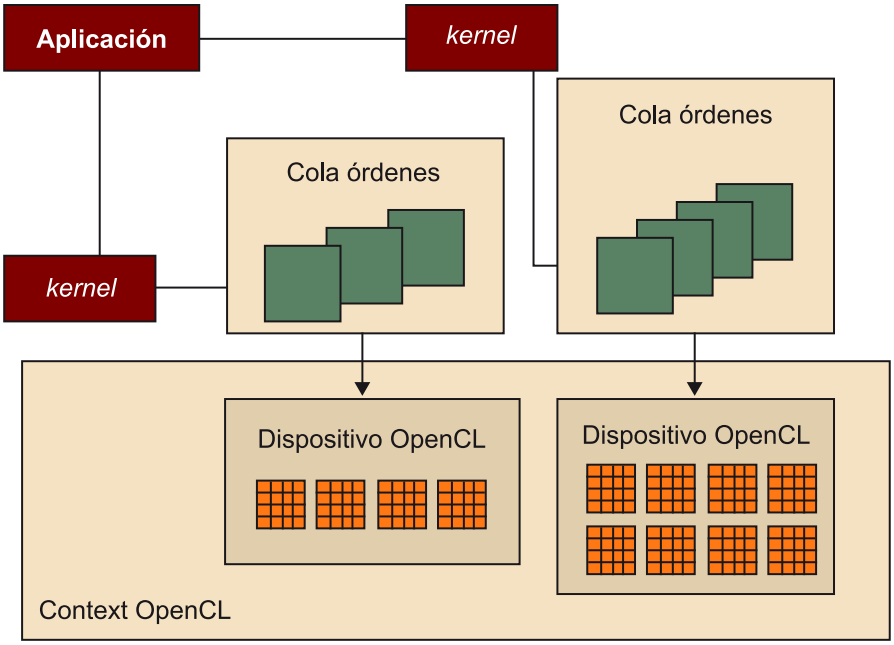
```
__kernel void matrix_add_opencl ( __global const float *A,
                                  __global const float *B,
                                  __global float *C,
    int N) {
    int i = get_global_id(0);
    int j = get_global_id(1);
    int index = i + j*N;
    if (i<N && j<N){
      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

Código 4.6. Implementación del *kernel* de suma de matrices en OpenCL

El modelo de gestión de dispositivos de OpenCL es mucho más sofisticado que el de CUDA, ya que OpenCL permite abstraer diferentes plataformas de hardware. Los dispositivos se gestionan mediante contextos. Tal como muestra la figura 34, para gestionar uno o más dispositivos (dos en el ejemplo de la figura), primero hay que crear un contexto que contenga los dispositivos mediante o bien la función `clCreateContext()` o bien la función `clCreateContextFromType()`. Normalmente, hay que indicar a las funciones `CreateContext` la cantidad y el tipo de dispositivos del sistema mediante la función `clGetDeviceIDs()`. Para ejecutar cualquier tarea en un dispositivo, primero hay que crear una cola de instrucciones para el dispositivo mediante la

función `clCreateCommandQueue()`. Una vez se ha creado una cola para el dispositivo, el código que se ejecuta en el procesador principal puede insertar un *kernel* y los parámetros de configuración asociados. Una vez el dispositivo esté disponible para ejecutar el *kernel* siguiente de la cola, este *kernel* se elimina de la cola y pasa a ser ejecutado.

Figura 34. Gestión de dispositivos en OpenCL mediante contextos



El código 4.7 muestra el código correspondiente al procesador principal para ejecutar la suma de matrices mediante el *kernel* del código 4.6. Supongamos que la definición e inicialización de variables se ha hecho correctamente y que la función del *kernel* `matrix_add_opencl()` está disponible. En el primer paso, se crea un contexto y, una vez se han obtenido los dispositivos disponibles, se crea una cola de instrucciones que utilizará el primer dispositivo disponible de los presentes en el sistema. En el segundo paso, se definen los objetos que necesitaremos en la memoria (las tres matrices *A* , *B* y C). Las matrices *A* y *B* se definen de lectura, la matriz *C* se define de escritura. Notad que en la definición de las matrices *A* y *B* se utiliza la opción `CL_MEM_COPY_HOST_PTR`, que indica que los datos de las matrices *A* y *B* se copiarán de los punteros especificados (`srcA` y `srcB`). En el tercer paso, se define el *kernel* que se ejecutará con posterioridad mediante `clCreateKernel` y se especifican también los argumentos de la función `matrix_add_opencl`. A continuación, se envía el *kernel* a la cola de instrucciones previamente definida y, por último, se leen los resultados del espacio de memoria correspondiente a la matriz *C* por medio del puntero `dstC`. En este ejemplo, no hemos entrado en detalle en muchos de los parámetros de las diferentes funciones de la interfaz de OpenCL; por lo tanto, se recomienda repasar su especificación, que está disponible en la página web http://www.khronos.org/opencl/.

```
main(){
    // Inicialización de variables, etc.
    (...)
```

```
// 1. Creación del contexto y cola en el dispositivo
cl_context context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);
// Para obtener la lista de dispositivos GPU asociados al contexto
size_t cb;
clGetContextInfo( context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malugar(cb);
clGetContextInfo( context, CL_CONTEXT_DEVICES, cb, devices, NULL);
cl_cmd_queue cmd_queue = clCreateCommandQueue(context, devices[0], 0 , NULL);
// 2. Definición de los objetos en memoria (matrices A, B y C)
cl_mem memobjs[3];
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_float)*n, NULL, NULL);

// 3. Definición del kernel y argumentos
cl_program program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);
cl_int err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "matrix_add_opencl", NULL);
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&memobjs[2]);
err |= clSetKernelArg(kernel, 3, sizeof(int), (void *)&N);

// 4. Invocación del kernel
size_t global_work_size[1] = n;
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size,
NULL, 0, NULL, NULL);

// 5. Lectura de los resultados (matriz C)
err = clEnqueueReadBuffer(context, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float),
dstC, 0, NULL, NULL);
(...)
```

Código 4.7. Código del procesador principal para la ejecución de la suma de matrices en OpenCL

Resumen

En este módulo, hemos estudiado cómo los dispositivos gráficos han evolucionado desde dispositivos especializados en su tarea para la generación de gráficos hasta dispositivos computacionales masivamente paralelos aptos para la computación de otras prestaciones de propósito general.

Hemos visto que las arquitecturas gráficas están formadas por un *pipeline* gráfico, compuesto por varias etapas, que ejecutan diferentes tipos de operaciones sobre los datos de entrada. Hemos destacado que la clave en la evolución de las arquitecturas gráficas está en el hecho de poder disponer de *shaders* programables que permiten al programador definir su funcionalidad.

Después de ver las características de algunas arquitecturas orientadas a computación gráfica, hemos estudiado las arquitecturas unificadas, que fusionan diferentes funcionalidades en un único modelo de procesador capaz de tratar tanto vértices como fragmentos.

Una vez observadas las limitaciones de la programación sobre GPU, hemos identificado el tipo de aplicaciones que pueden sacar provecho de la utilización de GPU. También hemos estudiado algunas arquitecturas GPU unificadas orientadas a la computación de propósito general. Hemos visto que la principal característica de este tipo de arquitectura es disponer de elementos computacionales o *stream processors* (SP), que pueden implementar cualquier etapa del *pipeline* y proporcionan la flexibilidad suficiente para implementar aplicaciones de propósito general, a pesar de que con ciertas limitaciones.

Finalmente, hemos estudiado los dos modelos de programación principales para computación de propósito general sobre GPU (GPGPU): CUDA, que es una extensión de C inicialmente concebida para arquitecturas de propiedad (Nvidia), y OpenCL, que es una interfaz estándar, abierta, libre y multiplataforma para la programación paralela.

Actividades

En este módulo, proponemos las actividades siguientes para profundizar en las arquitecturas basadas en computación gráfica y para ayudar a complementar y a profundizar en los conceptos introducidos.

1. Durante este módulo, hemos utilizado la suma de matrices como ejemplo principal. Pedimos que penséis e implementéis la multiplicación de matrices por vuestra parte utilizando CUDA u OpenCL. Una vez tengáis vuestra versión propia, comparadla con alguna de las numerosas soluciones que se pueden encontrar haciendo una búsqueda en Internet o bien en la bibliografía.

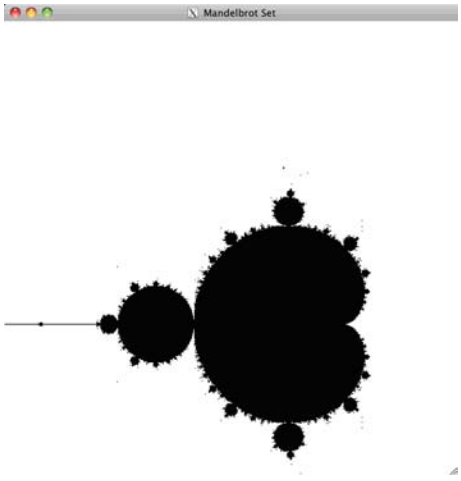
2. Explorad el modelo de programación Microsoft Direct Compute, que es una alternativa a CUDA/OpenCL. ¿Cuáles son las principales ventajas e inconvenientes respecto a CUDA/OpenCL?

3. La mayoría de los PC actuales disponen de una GPU. Escribid si en vuestro sistema personal tenéis alguna y cuáles son las características. Si tenéis alguna GPU en vuestro sistema, tened en cuenta las características en orden a las actividades siguientes.

4. Explorad cómo se pueden implementar aplicaciones que utilicen más de una GPU en el mismo sistema. Buscad la manera de consultar las características del sistema compatible con CUDA (por ejemplo, el número de dispositivos disponibles en el sistema). ¿Qué modelo seguiríais en CUDA/OpenCL para utilizar varias GPU distribuidas en diferentes nodos? (Notad que, al tratarse de múltiples nodos, vais a necesitar mensajes.)

5. Una de las características positivas de las GPU es que pueden llegar a ofrecer un rendimiento en coma flotante muy elevado a un coste energético bastante reducido. Buscad en Internet la corriente eléctrica que requieren varias arquitecturas GPU y elaborad un estudio comparativo respecto a las CPU desde un punto de vista de rendimiento y consumo eléctrico. ¿Qué condiciones se tienen que cumplir para que salga a cuenta la utilización de GPU?

6. Os proponemos programar una versión del conjunto Mandelbrot para CUDA u OpenCL. El conjunto Mandelbrot resulta en una figura geométrica de complejidad infinita (de naturaleza fractal) obtenida a partir de una fórmula matemática y un pequeño algoritmo recursivo, tal como muestra la figura siguiente.



El código siguiente en C es una implementación secuencial del conjunto Mandelbrot que resulta en la figura mostrada. Tomad esta implementación como referencia y tened en cuenta que, para compilarla, habrá que incluir las bibliotecas `libm` y `libX11` y, por lo tanto, una posible manera de compilación sería:

```
gcc -I/usr/include/X11 -omandel mandel.c -L/usr/lib -lX11 -lm
```

(Notad que los directorios pueden depender de cada sistema en particular.)

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

#define         X_RESN    800     /* resolución eje de las X */
#define         Y_RESN    800     /* resolución eje de las Y */

typedef struct complextype {
        float real, imag;
} Compl;

int main ()
{
        Window          win;                    /* inicialización de una ventana */
        unsigned int    width, height,          /* medida de ventana */
                        x, y,                   /* posición ventana */
                        border_width,           /* grueso marco en píxeles */
                        display_width, display_height,  /* medida pantalla */
                        screen;
        char            *window_name = "Mandelbrot Set", *display_name = NULL;
        GC                      gc;
        unsigned long           valuemask = 0;
        XGCValues               values;
        Display                 *display;
        XSizeHints              size_hints;
        Pixmap                  bitmap;
        XPoint                  points[800];
        FILE                    *fp, *fopen ();
        char                    str[100];
        XSetWindowAttributes    attr[1];

/* variables Mandlebrot */
   int                                  i, j, k;
   Compl                                z, c;
   float                                lengthsq, temp;

/* conexión al Xserver */
        if (  (display = XOpenDisplay (display_name)) == NULL ) {
           fprintf (stderr, "drawon: cannot connect to X server %s\n", XDisplayName
           (display_name) );
           exit (-1);
        }

/* obtener tamaño de la pantalla */
        screen = DefaultScreen (display);
        display_width = DisplayWidth (display, screen);
        display_height = DisplayHeight (display, screen);

/* establecer tamaño de la ventana */
        width = X_RESN;
        height = Y_RESN;

/* establecer posición de la ventana */
        x = 0;
        y = 0;

/* crear una ventana opaca */
        border_width = 4;
        win = XCreateSimpleWindow ( display, RootWindow (display, screen), x, y,
                width, height, border_width, BlackPixel (display, screen),
                WhitePixel(display, screen));
        size_hints.flags = USPosition|USSize;
        size_hints.x = x;
        size_hints.y = y;
        size_hints.width = width;
        size_hints.height = height;
        size_hints.min_width = 300;
        size_hints.min_height = 300;
        XSetNormalHints (display, win, &size_hints);
        XStoreName(display, win, window_name);

/* crear un contexto para los gráficos */
        gc = XCreateGC (display, win, valuemask, &values);
        XSetBackground (display, gc, WhitePixel (display, screen));
        XSetForeground (display, gc, BlackPixel (display, screen));
        XSetLineAttributes (display, gc, 1, LineSolid, CapRound, JoinRound);
        attr[0].backing_store = Always;
        attr[0].backing_planes = 1;
        attr[0].backing_pixel = BlackPixel(display, screen);
        XChangeWindowAttributes(display, win, CWBackingStore | CWBackingPlanes |
        CWBackingPixel, attr);
        XMapWindow (display, win);
        XSync(display, 0);

 /* Calcular y dibujar los puntos */
     for(i=0; i < X_RESN; i++)
     for(j=0; j < Y_RESN; j++) {

          z.real = z.imag = 0.0;
          c.real = ((float) j - 400.0)/200.0;  /* factores de escalado para ventana de
          800x800 */
          c.imag = ((float) i - 400.0)/200.0;
          k = 0;

           do  {                             /* iterar para definir el color por el píxel */
             temp = z.real*z.real - z.imag*z.imag + c.real;
             z.imag = 2.0*z.real*z.imag + c.imag;
             z.real = temp;
             lengthsq = z.real*z.real+z.imag*z.imag;
             k++;
           } while (lengthsq < 4.0 && k < 100);

          if (k == 100) XDrawPoint (display, win, gc, j, i);

          }

          XFlush (display);
          sleep (30);      /* Esperamos unos cuantos segundos para poder ver el resultado */
}
```

Una vez hayáis hecho la implementación, podéis elaborar un pequeño estudio de rendimiento utilizando diferentes configuraciones (por ejemplo, diferentes tamaños de bloque o distribución de flujos). Notad que, para escribir los resultados, no podréis utilizar el mismo me-

canismo, ya que desde los dispositivos GPU no se puede acceder directamente al espacio de memoria de la ventana que se mostrará por pantalla.

Una vez tengáis el estudio de rendimiento, lo más probable es que no observéis escalabilidad respecto al número de núcleos utilizados (*speedup*), en especial si consideráis el tiempo de ejecución secuencial de referencia como el de la ejecución en CPU. Esto es porque el problema es muy pequeño. Además, los dispositivos GPU requieren de un cierto tiempo de inicialización, que podría dominar la ejecución del programa. Pensad cómo haríais el problema más grande para poder observar una cierta escalabilidad y tener tamaños significativos.

La implementación proporcionada resulta en una imagen fractal en blanco y negro. Podéis hacer una pequeña búsqueda en Internet y añadir colores. También podéis proponer otras extensiones, como la implementación con múltiples GPU, ya sean en un mismo sistema o distribuidas, en varios nodos.

Bibliografía

AMD (2008). *R600-Family Instruction Set Architecture. User Guide*.

AMD (2010). *ATI Stream Computing OpenCL. Programming Guide*.

AMD (2011). *Evergreen Family Instruction Set Architecture Instructions and Microcode. Reference Guide*.

Gaster, B.; Howes, L.; Kaeli, D. R.; Mistry, P.; Schaa, D. (2011). *Heterogeneous Computing with OpenCL*. Morgan Kaufmann.

Glaskowsky, P. (2009). *NVIDIA's Fermi: The First Complete GPU Computing Architecture*.

Green, S. (2005). "The OpenGL Framebuffer Object Extension". En: *Game Developers Conference (GDC)*.

Kirk, D. B.; Hwu, W. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.

Munshi, A.; Gaster, B.; Mattson, T. G.; Fung, J.; Ginsburg, D. (2011). *OpenCL Programming Guide*. Addison-Wesley Professional.

Munshi. A. (2009). *The OpenCL Specification*. Khronos OpenCL Working Group.

Nvidia (2007). *Nvidia CUDA Compute Unified Device Architecture. Technical Report*.

Nvidia (2007). *The CUDA Compiler Driver NVCC*.

Nvidia (2008). *NVIDIA GeForce GTX 200 GPU Architectural Overview. Second-Generation Unified GPU Architecture for Visual Computing. Technical Brief*.

Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A.; Purcell, T. J. (2007). "A Survey of General-Purpose Computation on Graphics Hardware". *Computer Graphics Forum*.

Pharr, M.; Randima, F. (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.

Randima, F.; Kilgard, M. J. (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley.

Sanders, J.; Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional.

Seiler, L.; Carmean, D.; Sprangle, E.; Forsyth, T.; Abrash, M.; Dubey, P.; Junkins, S.; Lake, A.; Sugerman, J.; Cavin, R.; Espasa, R.; Grochowski, E.; Juan, T.; Hanrahan, P. (2008). "Larrabee: A Many–Core x86 Architecture for Visual Computing". *ACM Trans. Graph.* (núm. 27, vol. 3, art. 18).