**UNIVERSIDAD JAUME I DE CASTELLÓN**
**E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES**


# Memoria de ejemplo para curso de LaTeX avanzado

Castellón, 3 de mayo de 2010

Trabajo presentado por: Francisco Daniel Igual Peña
Dirigido por: Director 1
Director 2

**UNIVERSIDAD JAUME I DE CASTELLÓN**
**E. S. DE TECNOLOGÍA Y CIENCIAS EXPERIMENTALES**


# Memoria de ejemplo para curso de LaTeX avanzado

Francisco Daniel Igual Peña

# Índice general

# Índice de figuras

# Índice de tablas

# Resumen

El presente documento presenta el proyecto realizado por el alumno Pepe Martínez (pepe@uji.es), dirigido y supervisado por el profesor Pedro Pérez (pedro@uji.es), miembro del Departamento de Ingeniería y Ciencia de los Computadores (*www.icc.uji.es*) de la Universitat Jaume I de Castellón de la Plana.

Este proyecto corresponde con el trabajo requerido para la asignatura Trabajo de Fin de Máster, perteneciente al Máster Oficial en Sistemas Inteligentes.

La creciente demanda del mercado durante la última década de aplicaciones gráficas de alta calidad ha implicado un gran aumento en la potencia de cálculo de los dispositivos *hardware* dedicados a tal fin. El uso de procesadores gráficos de alto rendimiento para tareas de carácter general es una tendencia en alza, por la excelente relación precio-prestaciones que éstos ofrecen.

El presente es un estudio de las arquitecturas más comunes en el campo de los procesadores gráficos existentes hoy en día en el mercado, así como de los algoritmos y aplicaciones que mejor se adaptan al paradigma de programación típico de este tipo de dispositivos.

## Palabras Clave

- Computación de Altas Prestaciones
- Procesadores Gráficos
- GPU

# Parte I

# La GPU como Unidad de Procesamiento Gráfico

Capítulo 1

# Introducción

*Mi carrera ha sido lenta como la del caracol, pero segura y sólida como su caparazón.*

Pepe

Durante la última década la demanda por parte de los usuarios de gran potencia de cálculo en el ámbito de la generación de gráficos tridimensionales, ha llevado a una rápida evolución del hardware dedicado a tal fin, dando lugar a la aparición de las Unidades de Procesamiento Gráfico o GPUs. A día de hoy, el poder computacional de una GPU a la hora de llevar a cabo su función supera con creces el rendimiento de las CPUs más avanzadas.

Una década son 10 años

Este hecho ha llevado a este tipo de procesadores a rebasar la frontera de la computación de gráficos tridimensionales, convirtiéndose en elementos de procesamiento ideales para la implementación de algoritmos de propósito general que hacen uso directamente del *hardware* gráfico. De hecho, algoritmos correctamente adaptados a las características específicas de ciertas GPUs han conseguido rendimientos muy superiores a sus correspondientes implementaciones sobre CPUs. En [1] se realiza un estudio en profundidad de los campos en los que se ha investigado en mayor medida la integración de las unidades de procesamiento gráfico en la ejecución de los algoritmos, así como su correcta adaptación a ellas. Entre ellos, podríamos destacar el álgebra lineal, procesamiento de imágenes, algoritmos de ordenación y búsqueda, procesamiento de consultas sobre bases de datos, etc.

Realmente, las GPU se han convertido en el primer sistema de computación paralela de altas prestaciones realmente extendido, debido, en gran medida, a su buena relación precio/prestaciones.

Sirvan dos ejemplos para ilustrar el creciente interés por la computación general sobre GPUs. *Folding@Home* es un proyecto de la Universidad de Stanford, similar al conocido *Seti@Home*, que, basándose en la computación distribuida, permite realizar simulaciones por ordenador de plegamiento de proteínas. La investigación actual dentro del proyecto (en colaboración con ATI) se centra en el aprovechamiento de las GPUs dentro de cada nodo de computación, para acelerar los cálculos realizados. Según los

estudios, sólo ciertas partes de las que forman la totalidad del cálculo a realizar son implementadas sobre GPUs, consiguiéndose en ellas rendimientos entre 20 y 40 veces mejores que sus correspondientes implementaciones sobre CPU.

NVIDIA presentó, en Junio de 2007, una nueva línea de *hardware* orientado a la computación general de altas prestaciones, de nombre *Tesla*, basado en sus productos gráficos de altas prestaciones. *Tesla* ofrece *hardware* de altas prestaciones (en forma, por ejemplo, de *clusters* de procesadores gráficos) sin ningún tipo de orientación a computación de aplicaciones gráficas. El interés de empresas tan importantes como AMD/ATI o NVIDIA así como el creciente número de estudios de la comunidad científica que demuestran las ventajas del uso de las GPUs sobre cierto tipo de aplicaciones, hacen de esta disciplina un campo de estudio relativamente nuevo y con un futuro prometedor.

El presente trabajo realiza un estudio de las posibilidades que ofrecen los procesadores gráficos actuales a nivel de computación de carácter general. Para ello, se ha optado por dividir el trabajo en tres partes diferenciadas:

- En una primera parte, se realizará un estudio del funcionamiento de las GPUs como unidades de procesamiento gráfico. Resulta esencial comprender cómo trabajan las GPUs con los datos para los cuales han sido diseñadas, para así de entender cómo se pueden adaptar los algoritmos de carácter general a este tipo de arquitecturas.
- En la segunda parte del trabajo se llevará a cabo una descripción detallada de las arquitecturas más comunes disponibles hoy en día a nivel de procesadores gráficos.
- La última parte del trabajo consiste en el desarrollo de rutinas para la evaluación de las prestaciones que puede llegar a ofrecer la GPU, así como de una biblioteca de rutinas desarrollada específicamente para facilitar la programación del procesador gráfico, uno de los mayores inconvenientes que presenta la programación con GPUs.

Capítulo 2

# El cauce de procesamiento gráfico

Pese a que nuestro trabajo se centrará en el uso de las GPU como unidades de procesamiento para computación de carácter general (GPGPU: *General Purpose Computing on GPU*), debido a la especificidad del *hardware* sobre el cual se trabajará, resulta totalmente necesario comprender el funcionamiento de las GPU desde el punto de vista de la computación de elementos gráficos, con el fin de, a continuación, realizar una analogía de cada uno de los mecanismos que se estudiarán con los que se aplicarán posteriormente en GPGPU.

Tradicionalmente, el funcionamiento de las GPU es sintetizado como un *pipeline* o cauce de procesamiento formado por etapas muy especializadas en la función que deben realizar, ejecutadas de forma paralela y en orden preestablecido. Cada una de estas etapas recibe su entrada de una etapa anterior y envía su salida a la siguiente etapa.

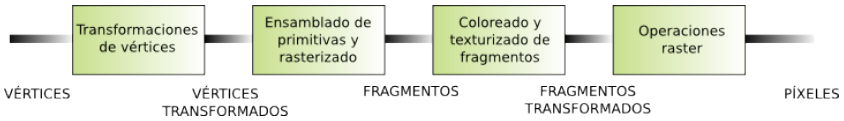



Figura 2.1: Pipeline gráfico de una GPU

La figura muestra el pipeline básico utilizado por las GPUs actuales (más adelante veremos las modificaciones que la serie G80 de Nvidia ha introducido en este pipeline clásico). La aplicación envía a la GPU una secuencia de vértices, agrupados en lo que se denominan primitivas geométricas: polígonos, líneas y puntos, que son tratadas secuencialmente a través de cuatro etapas diferenciadas:

## Primera etapa: Transformación de Vértices

La transformación de vértices es la primera etapa del pipeline de procesamiento gráfico. Básicamente, se lleva a cabo una secuencia de operaciones matemáticas sobre cada uno de los vértices suministrados (transformación de la posición del vértice en una posición en pantalla, generación de coordenadas para aplicación de texturas y asignación de color a cada vértice, ...) por la aplicación.

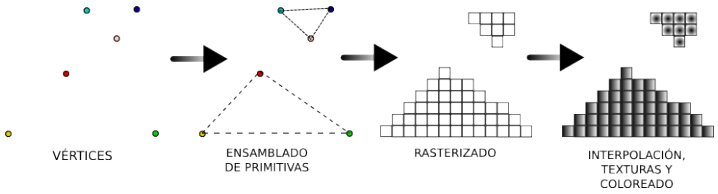



Figura 2.2: Representación de las funciones realizadas por el pipeline gráfico

## Segunda etapa: Ensamblado de Primitivas y Rasterización

Los vértices transformados recién generados pasan a una segunda etapa, en la que son agrupados en primitivas geométricas basándose en la información recibida junto con la secuencia inicial de vértices. Como resultado, se obtiene una secuencia de triángulos, líneas o puntos.

Dichos puntos son sometidos, a continuación, a una etapa llamada *rasterización*. La rasterización es el proceso por el cual se determina el conjunto de píxeles "cubiertos" por una primitiva determinada. Los resultados de la rasterización son conjuntos de localizaciones de píxeles y conjuntos de fragmentos.

Es importante definir correctamente el concepto de fragmento, por la importancia que cobrará cuando se trabaje en computación general. Un fragmento tiene asociada una localización de píxel, así como información relativa a su color, color especular y uno o más conjuntos de coordenadas de textura. Se puede pensar en un fragmento como en un "píxel en potencia": si el fragmento supera con éxito el resto de etapas del pipeline, se actualizará la información de píxel como resultado.

## Tercera etapa: Interpolación, Texturas y Colores

Una vez superada la etapa de rasterización, y con uno o varios fragmentos como resultado de la misma, cada uno de ellos es sometido a operaciones de interpolación, operaciones matemáticas y de textura (que resultarán la fase más interesante para nuestro trabajo) y determinación del color final de cada fragmento. Además de deteminar el color final que tomará el fragmento, en esta etapa es posible descartar un fragmento determinado para impedir que su valor sea actualizado en memoria; por tanto, esta etapa emite uno o ningún fragmentos actualizados para cada fragmento de entrada.

## Últimas etapas

Las últimas etapas del *pipeline*, que realizan operaciones llamadas *raster*, analizan cada fragmento, sometiéndolo a un conjunto de tests relacionados con aspectos gráficos del mismo. Estos tests determinan los valores que tomará el píxel que se generará en memoria a partir del fragmento original. Si cualquiera de estos tests falla, es en esta etapa cuando se descarta el píxel correspondiente, y por tanto no se realiza la escritura en memoria del mismo. En caso contrario, y como último paso, se realiza una escritura en memoria (llamada *framebuffer*) con el resultado final del proceso.

Capítulo 3

# Etapas programables del pipeline: procesadores de vértices y de fragmentos

Como se verá más adelante, la tendencia a la hora de diseñar GPUs marca un aumento del número de unidades programables dentro de la propia GPU. La figura 3.1 muestra, con mayor detalle, las etapas del pipeline gráfico de una GPU, añadiendo a la información mostrada en la figura 2.1 las unidades programables en las fases de transformación de vértices y transformación de fragmentos. Actualmente, estas dos fases son las que tienden a ofrecerse como programables, siendo el Procesador de Vértices (*Vertex Processor*) y el Procesador de Fragmentos (*Fragment Processor*) las unidades que se encargarán de llevar a cabo cada una de estas fases.

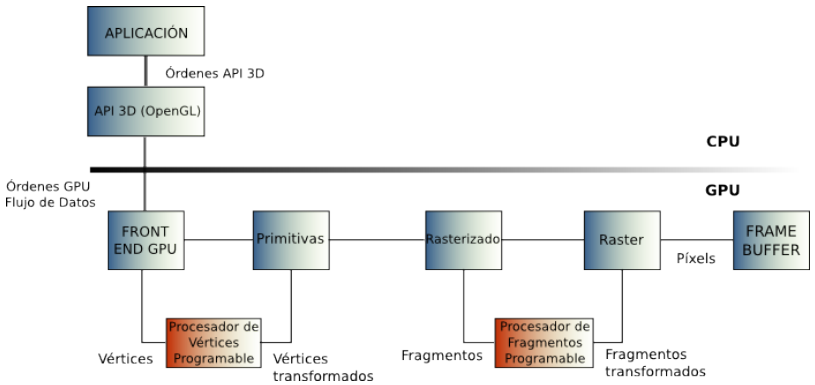



Figura 3.1: Pipeline gráfico programable. Las fases coloreadas en rojo se ofrecen como programables

## 3.1. El procesador de vértices programable

El funcionamiento de un procesador de vértices programable es muy similar al que se muestra en la figura 3.2 para un procesador de fragmentos programable. El primer

paso consiste en la carga de los atributos de cada uno de los vértices a analizar. El almacenamiento se suele realizar sobre registros internos del propio procesador de vértices. El procesador ejecuta de forma secuencial cada una de las instrucciones que componen el programa cargado, hasta que éste finaliza. Dichas instrucciones se localizan en zonas reservadas de la memoria de vídeo. Existen tres tipos de registros:

- Registros de atributos de vértices, de sólo lectura, con información relativa a cada uno de los vértices.
- Registros temporales, de lectura/escritura, utilizados en cálculos intermedios.
- Registros de salida, donde se almacenan los nuevos atributos de los vértices transformados que, a continuación (tras ciertas fases intermedias), pasarán al procesador de fragmentos.

Uno de los mayores contratiempos que surgen al trabajar con este tipo de procesadores programableses la limitación en el conjunto de operaciones que son capaces de ejecutar. De hecho, las operaciones que debe ser capaz de realizar todo procesador de vértices se pueden resumir en:

- Operaciones matemáticas en coma flotante sobre vectores de entre una y cuatro componentes (ADD, MULT, MAD, mínimo, máximo, ...).
- Operaciones vía hardware para negación de un vector y *swizzling*[1] (reordenación arbitraria de valores).
- Exponenciales, logarítmicas y trigonométricas.

Las GPUs de última generación soportan, igualmente, operaciones de control de flujo que permiten la implementación de bucles y construcciones condicionales. Este tipo de GPUs poseen procesadores de vértices totalmente programables, que operan bien en modo SIMD (*Simple Instruction, Multiple Data*) o bien en modo MIMD sobre los vértices de entrada. Al procesarse vértices representados por un vector $(x, y, z, w)$, el hardware que los implementa debe estar optimizado para procesar vectores de cuatro componentes, con el fin de producir resultados en pocos ciclos para vértices individuales.

Un aspecto importante es la capacidad de los procesadores de vértices de cambiar la posición de los vértices recibidos a la entrada. Por tanto, un procesador de vértices puede controlar la posición que ocupará en memoria un cierto dato escrito, o lo que es lo mismo, tiene capacidad para realizar operaciones de *scatter*. Por contra, un procesador de vértices no puede, de forma directa, acceder en modo lectura a información de vértices distintos al que se está procesando en un momento dado, por lo que su capacidad de realizar operaciones *gather* es nula. Esta es una de las diferencias principales con respecto a los procesadores de fragmentos, estudiados a continuación.

[1] El *swizzling*, o reordenación de valores, se estudiará con detalle más adelante. Su correcto uso es básico para aprovechar al máximo las capacidades de las GPUs actuales.

## 3.2. El procesador de fragmentos programable

Los procesadores programables de fragmentos requieren muchas de las operaciones matemáticas que exigen los procesadores de vértices, añadiendo además a éstas operaciones sobre texturas. Este tipo de operaciones facilitan el acceso a imágenes (texturas) mediante el uso de un conjunto de coordenadas, para a continuación, devolver la muestra leída tras un proceso de filtrado.

Las GPUs modernas poseen gran cantidad de procesadores de fragmentos, que al igual que los procesadores de vértices, son totalmente programables. Este tipo de procesadores únicamente operan en modo SIMD sobre los elementos de entrada, procesando vectores de cuatro elementos en paralelo. Además, un aspecto destacable es la capacidad de este tipo de procesadores de acceder en modo lectura a otras posiciones de la textura o flujo de datos de entrada distintas a la que en un momento determinado se está procesando. Por tanto, este tipo de unidades tienen la capacidad de realizar operaciones de *gathering*, hecho que no se daba en los procesadores de vértices. Sin embargo, el procesador no es capaz de cambiar la localización de salida de un píxel (es decir, la posición de memoria que ocupara un dato determinado una vez procesado). Por tanto, no son capaces, de forma nativa, de realizar operaciones de *scatter*. La figura 3.2 muestra el funcionamiento esquemático de uno de estos procesadores.

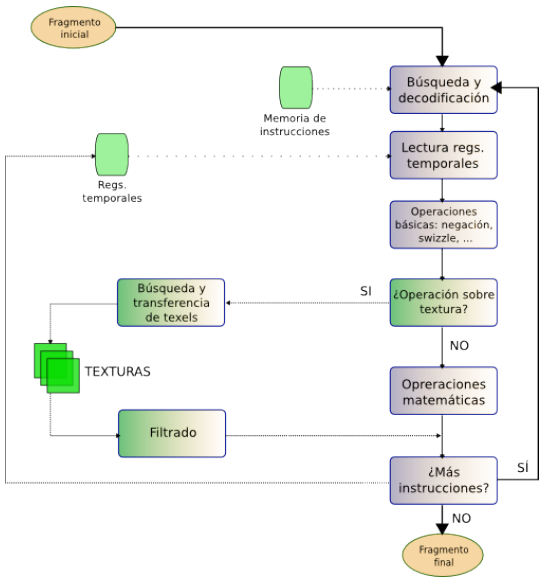



Figura 3.2: Funcionamiento esquemático de un procesador de fragmentos. El funcionamiento de un procesador de vértices sería muy similar, eliminando las operaciones relativas a texturas en caso de no estar soportadas por el mismo.

A la hora de adaptar las capacidades de las GPUs modernas a la computación general, los procesadores de fragmentos son la opción idónea por tres razones básicas:

- Habitualmente existen más procesadores de fragmentos que procesadores de vértices

en una GPU típica. Este hecho ha sido cierto hasta la aparición de las arquitecturas unificadas, de las que se hablará más adelante, y que fusionan ambos tipos de procesadores en un sólo modelo de procesador capaz de tratar tanto vértices como fragmentos.

- Los procesadores de fragmentos soportan operaciones de lectura de datos procedentes de texturas (aunque realmente los últimos procesadores gráficos también poseen esta capacidad en la etapa de procesamiento sobre procesadores de vértices). Las texturas, como se verá más adelante, juegan un papel determinante a la hora trabajar con conjuntos de datos (vectores o matrices) en GPU.

- El resultado de procesar un fragmento se deposita directamente en memoria, por lo que puede convertirse en un nuevo flujo de datos directamente para volver a ser procesado por el procesador de fragmentos. En cambio, en el caso de los procesadores de vértices, el resultado tras la computación debe pasar todavía por etapas de rasterizado y procesadores de fragmento antes de alcanzar la memoria, lo que hace más complicado su uso para computación de propósito general.

## 3.3. Otros aspectos importantes del pipeline

#### La unidad de texturas y la técnica Render-To-Texture

La única forma en que los procesadores de fragmentos pueden acceder a la memoria es en forma de texturas. La unidad de texturas, presente en cualquier implementación de GPUs, realiza el papel de interfaz de sólo lectura a memoria.

Cuando una imagen es generada por la GPU, existen dos opciones:

1. Escribir la imagen a memoria (*framebuffer*), de forma que sea mostrada en pantalla.

2. Escribir la imagen en memoria de textura, técnica llamada *render-to-buffer*. Esta técnica resulta imprescindible en GPGPU, ya que es el único mecanismo para implementar de forma sencilla una realimentación entre datos de salida de la GPU como dato de entrada para un próximo procesamiento sobre dichos datos, sin pasar por memoria principal del sistema, con el sobrecoste de transferencia de datos que esta acción conlleva. Podríamos realizar una analogía entre esta técnica y una interfaz de sólo escritura en memoria.

Vistas estas dos interfaces (lectura/escritura), podría parecer obvio el hecho de pensar en ellas como una única interfaz de lectura/escritura conjunta en memoria. Sin embargo, los procesadores de fragmentos pueden leer desde memoria un número de veces ilimitado dentro de un mismo programa, pero sólo es posible realizar una escritura en memoria, al finalizar el mismo. Por tanto, lecturas y escrituras en memoria son conceptos totalmente separados, y no es posible abstraer ambas interfaces como una sola.

### Tipos de datos

Aunque algunos de los lenguajes de programación para GPUs actuales disponen de tipos de datos booleanos o enteros, las GPUs actuales únicamente (de forma estricta) operan con números reales, bien con coma fija o flotante. Sin embargo, los procesadores gráficos más extendidos trabajan con precisiones de 16 bits (un bit de signo, 10 para la mantisa y 5 para el exponente), o de 32 bits (bit de signo, 23 bits de mantisa y 8 para el exponente, tal y como dicta el estándar IEEE-754) para formatos de coma flotante. Los productos de ATI soportan un formato de coma flotante e 24 bits, con un bit de signo, 16 bits para la mantisa y 7 bits para el exponente. Se espera soporte para precisión de 64 bits para la próxima generación de GPUs. Este hecho hará de las GPUs plataformas más atractivas todavía desde el punto de vista de las aplicaciones de carácter general.

La falta de tipos de dato entero en GPUs resulta una limitación muy a tener en cuenta. De hecho, se suele solucionar parcialmente el problema mediante el uso en su lugar de números en coma flotante de 32 bits; aún así, estas representaciones, con 23 bits para la mantisa, no pueden representar de forma exacta el mismo rango de valores que puede ser representado mediante enteros de 32 bits. La tabla 3.1 muestra, de forma detallada, el formato y propiedades de cada formato de coma flotante utilizado a día de hoy en las GPUs.

| **Nombre** | **Signo** | **Exponente** | **Mantisa** | **Mayor Valor** | **Menor Valor** |
|---|---|---|---|---|---|
| NVIDIA 16-bits | 15 (1) | 14:10 (5) | 9:0 (10) | $\pm 65536$ | $\pm \sim 2^{-14} \cong 10^{-5}$ |
| ATI 16-bits | 15 (1) | 14:10 (5) | 9:0 (10) | $\pm 131008$ | $\pm \sim 2^{-15} \cong 10^{-5}$ |
| ATI 24-bits | 23 (1) | 22:16 (7) | 15:0 (16) | $\pm \sim 2^{64} \cong 10^{19}$ | $\pm \sim 2^{-62} \cong 10^{-19}$ |
| NVIDIA 32-bits | 31 (1) | 30:23 (8) | 22:0 (23) | $\pm \sim 2^{128} \cong 10^{38}$ | $\pm \sim 2^{-126} \cong 10^{-38}$ |

Tabla 3.1: Formatos de coma flotante soportados actualmente por las GPUs de ATI y NVIDIA

Capítulo 4

# Analogías entre GPU y CPU

Uno de los mayores inconvenientes a la hora de trabajar con GPUs radica en la dificultad que existe para el programador a la hora de realizar una transformación entre los programas a ejecutar en CPU y los que se ejecutarán en el procesador gráfico.

De hecho, una de las razones que han llevado a ciertas compañías como NVIDIA a desarrollar sus propias interfaces enfocadas a computación general sobre GPUs, ha sido la de abstraer al programador del hecho de estar desarrollando código para ser ejecutado sobre un procesador distinto a la CPU.

Pese a todo, y aunque las interfaces orientadas a computación general sobre GPUs están tomando una relevancia cada vez mayor, resulta conveniente realizar una analogía entre los conceptos básicos de programación en CPUs de propósito general y computación sobre GPUs, de forma que el paso de un tipo de programas a otro sea lo más claro y sencillo posible.

## Texturas = Vectores

Existen dos estructuras de datos fundamentales en las GPU para representar conjuntos de elementos del mismo tipo: las texturas y los arrays de vértices. Sin embargo, y ya que se ha comentado que los procesadores de fragmentos son la unidad programable elegida para la mayoría de aplicaciones, se realizará una analogía en la mayor parte de los casos entre vectores de datos en CPU y texturas en GPU.

La memoria de textura es la única accesible de forma aleatoria desde programas de fragmentos (y, en las últimas generaciones de GPUs, también desde programas de vértices). Cualquier vértice, fragmento o flujo que deba ser accedido de forma aleatoria, debe ser primero transformado en una textura. Las texturas pueden ser leídas o escritas tanto por la CPU como por la GPU; en este úlimo caso, la escritura se realiza llevando a cabo el proceso de renderizado directamente sobre una textura, o bien copiando los datos desde el *framebuffer* a memoria de textura directamente.

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, accediendo a cada uno de sus elementos mediante direcciones en una, dos o tres dimensiones, respectivamente. La forma más común de realizar la transformación entre vectores (o matrices) y texturas

es mediante la creación de texturas bidimensionales; aún así, existen numerosos estudios para adaptar cada problema concreto al tipo de textura más conveniente (véase [2] para más información).

## Kernels = Bucles internos

Cuando la programación se realiza sobre CPUs, es habitual iterar sobre los elementos de un flujo de datos (normalmente con la ayuda de un bucle, para iterar sobre los elementos de un vector, dos bucles, si se trata de una matriz, ...), procesando cada uno de forma secuencial. En este caso, se podría decir que las instrucciones pertenecientes al bucle representan el kernel de ejecución, aplicado a cada elemento del flujo.

En las GPUs, estas instrucciones son programadas dentro de un programa, llamado *fragment program*, y aplicadas a todos los elementos del flujo de entrada. La cantidad de paralelismo que se puede extraer en esta operación dependerá del número de procesadores que posea la GPU sobre la que se trabaje, pero también, a nivel de instrucción, de cómo se aproveche las facilidades de operación sobre vectores de cuatro elementos ofrecidas por el procesador gráfico.

## Render-to-Texture = Retroalimentación

En cualquier rama de la computación, todo problema a resolver puede ser dividido en una serie de etapas, las entradas de cada una de las cuales dependen de las salidas de etapas anteriores. Si hablamos de flujos de datos siendo tratados por una GPU, cada kernel debe procesar un flujo completo antes de que el siguiente kernel pueda comenzar a trabajar de nuevo con los datos resultantes de la anterior ejecución (cabe recordar que el conjunto completo de procesadores de vértices o fragmentos de una GPU es programado utilizando un único programa, de forma obligatoria).

La implementación de esta realimentación de datos entre etapas del proceso de computación es trivial en la CPU, debido a su modelo unificado de memoria, gracias al cual cualquier dirección de memoria puede ser leída o escrita en cualquier punto del programa.

La técnica Render-to-Texture, anteriormente descrita con detalle, es la que permitirá el uso de procedimientos similares en GPU, escribiendo los resultados de la ejecución de un programa en memoria para que puedan estar disponibles como entradas para futuras ejecuciones.

## Rasterización = Computación

Las rutinas habitualmente desarrolladas son programadas sobre el procesador de fragmentos. Por tanto, necesitan un flujo de fragmentos sobre el que operar. La invocación de la computación, pues, se reducirá a conseguir hacer llegar dicho flujo a las unidades funcionales de la GPU correspondientes.

A tenor de lo estudiado acerca del funcionamiento del pipeline gráfico, la invocación de la computación se reduce a la creación de un conjunto de vértices con los que proveer al procesador de vértices. La etapa de rasterización determinará qué píxeles del flujo de

datos se ven cubiertos por las primitivas generadas a partir de dichos vértices, generando un fragmento para cada uno de ellos.

Un ejemplo puede ilustrar mejor este mecanismo. Imaginemos que nuestro objetivo es el de operar sobre cada uno de los elementos de una matriz de N filas y M columnas. Por tanto, los procesadores de fragmentos deben realizar una (la misma) operación sobre cada uno de los NxM elementos que componen la matriz. La solución que suele adoptarse en GPGPU es la de enviar a la GPU información relativa a cuatro vértices, cada uno de los cuales se corresponderían con los cuatro vértices de un rectángulo, de forma que la GPU, automáticamente, sería capaz de generar un fragmento para cada uno de los elementos que componen el rectángulo (tratándolo a modo de cuadrícula), y pasando pues cada uno de ellos a la fase de procesamiento de fragmentos.

## Coordenadas de texturas = Dominio computacional

Cada rutina que es ejecutada en la GPU toma un conjunto de flujos de datos como entrada, y típicamente genera un único flujo de datos como salida (realmente, las GPUs de última generación son capaces de generar múltiples flujos de salida, hasta cuatro). Cualquier ejecución lleva asociados un dominio de entrada (conjunto de datos de entrada correspondientes a los flujos sobre los que se trabaja) y un rango de salida (subconjunto del flujo de salida sobre el que se depositará el resultado). Las dimensiones de ambos conjuntos no tienen necesariamente que coincidir, pudiendo darse casos de reducción (con el rango menor que el dominio) o ampliación (en los que el rango es mayor que el dominio) de los datos.

Las GPUs proporcionan un método sencillo para acceder a cada uno de los elementos que componen el flujo de entrada o dominio de ejecución de un programa; éste método se denomina *coordenadas de texturas*. Desde el punto de vista gráfico, los vértices son los únicos elementos con unas coordenadas asociadas que los ubican en el espacio. Como se verá más adelante, la computación es invocada mediante la construcción de un polígono (normalmente un rectángulo), a partir de los vértices que lo forman. Por tanto, mediante un método de interpolación lineal, será necesario calcular para cada fragmento sus coordenadas asociadas, de forma que sea posible referenciarlo dentro de cada programa. Dichas coordenadas pasan junto con el flujo como entrada del procesador de fragmentos.

Desde el punto de vista de la computación general, es posible pensar en las coordenadas de texturas como índices de vectores o matrices, ya que permiten acceder de forma sencilla a cualquier posición dentro de una textura en memoria (recordemos que el concepto de textura en GPU es asociado habitualmente al de matriz o vector en CPU).

## Coordenadas de vértices = Rango computacional

Como ya se ha visto, los fragmentos son generados durante la etapa de rasterizado, haciendo uso de los datos geométricos (vértices), proporcionados por el programador; dichos fragmentos se convertiran en píxeles una vez completada la etapa de procesamiento en el procesador de fragmentos.

Sin embargo, como ya se ha descrito, los procesadores de fragmentos no son capaces de realizar operaciones de *scatter*, por lo que no pueden escribir sus resultados en

cualquier posición de memoria. De hecho, son los vértices de entrada definidos (junto con un posible programa de vértices), los que determinan qué píxeles serán generados.

Para computación general, es habitual especificar cuatro vértices que formarán un rectángulo (también llamado *quad*), y no programar ningún procesador de vértices, de forma que no se realice ninguna transformación sobre ellos. Así, podemos decir que son las coordenadas de dichos vértices las que determinan el rango de salida de la computación.

## Ejemplo

Se estudiará a continuación un ejemplo completo que ilustra cada uno de los conceptos anteriormente explicados. La operación de suma de matrices simplemente toma como operandos de entrada dos matrices de valores reales, $A$ y $B$, para obtener una tercera matriz, $C$, cuyos elementos serán la suma de los correspondientes elementos de las matrices de entrada:

$$C_{ij} = A_{ij} + B_{ij} \tag{4.1}$$

Una implementación en CPU crea tres matrices en memoria, recorriendo cada uno de los elementos de las matrices de entrada, calculando para cada par su suma, y depositando el resultado en una tercera matriz, del siguiente modo:

```
1 for( i=0; i<M; i++ ){
    for( j=0; j<N; j++ ){
3     C[i, j] = A[i, j] + B[i, j];
    }
5 }
```

El procedimiento en GPU es sensiblemente más complejo:

- En primer lugar, es necesario definir tres texturas en memoria de vídeo, que actuarán del mismo modo en el que lo hacen las matrices definidas en memoria central para el caso de operar en CPUs.
- Como ya se ha indicado, cada *kernel* o programa a ejecutar dentro de la GPU corresponde a aquellas operaciones que se realizan para cada uno de los elementos que componen el flujo de entrada, es decir, aquellas operaciones que corresponden al bucle más interno de una implementación en CPU como la anteriormente vista. De este modo, un posible programa a ejecutar en cada uno de los procesadores de fragmentos podría tener simplemente la forma:

  ```
  1   texC[i, j] = texA[i, j] + texB[i, j]
  ```

  Sin embargo, ya se ha descrito la imposibilidad de los procesadores de fragmentos para realizar operaciones de *scatter*. Por tanto, la escritura de datos en la textura `texC` del modo anteriormente descrito no es viable: cada ejecución del anterior programa trabaja sobre un fragmento determinado, siendo capaz únicamente de variar su valor, y ningún otro. Por tanto, un programa correcto tendría la forma:

```
1   returnvalue = texA[i, j] + texB[i, j]
    return returnvalue
```

- Por tanto, ¿qué elemento de la computación en GPUs se correspondería con los dos bucles que recorren cada uno de los elementos de las matrices de entrada?. Recordemos que la computación en GPUs sigue un paradigma SIMD, en el que cada una de las operaciones programadas se ejecutan para cada uno de los elementos que componen un flujo de entrada al procesador. Así, todo lo necesario para una correcta ejecución se centra en crear un flujo de datos sobre los que la GPU pueda trabajar de forma correcta.

  Para ello, es necesario definir cuatro vértices, que se corresponderían con cada uno de los extremos de un hipotético rectángulo de dimensiones $NxM$, en cuyo interior se situaría la matriz $C$. Así, cada uno de sus elementos se convertiría en un fragmento, y podríamos escribir sobre él (y sobre ningún otro elemento en ese mismo instante) según el programa anteriormente definido.

La figura 4.1 muestra el proceso completo anteriormente descrito. Resulta necesario realizar una transferencia de datos inicial entre memoria central y memoria de texturas, para que los datos asociados a los operandos fuente se encuentren en dicho espacio de memoria en el momento de la ejecución del *shader*. Los resultados obtenidos una vez ejecutado el mismo son depositados directamente sobre la textura destino (mediante la técnica *render-to-texture*. Finalmente, los datos almacenados en dicha textura son transferidos de vuelta a memoria central, para que el programa que invocó la computación sobre el procesador gráfico continúe su ejecución.

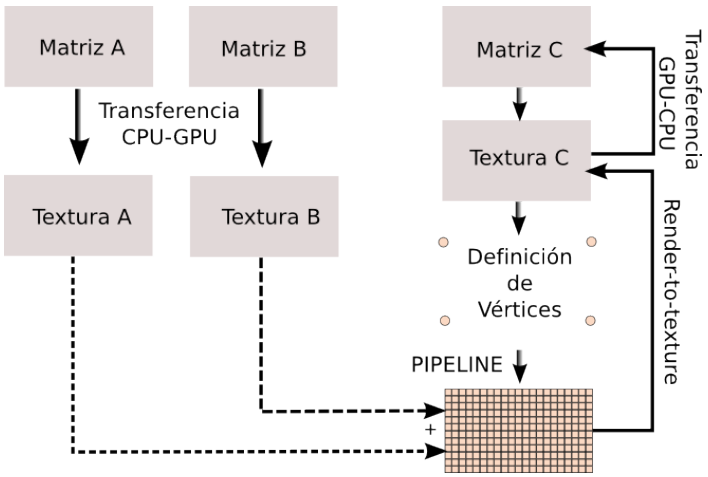



Figura 4.1: Ejemplo completo de ejecución de una operación sencilla en GPU

# Parte II

# Estudio de Arquitecturas GPU

Capítulo 5

# Estudio de la arquitectura GPU

La presente sección realiza un estudio detallado de las dos arquitecturas más comunes hoy en día para la implementación de GPUs: en primer lugar, se describe la arquitectura clásica implementada en GPUs correspondiente al cauce de ejecución descrito en capítulos anteriores; en segundo lugar, se estudia la arquitectura implementada en GPUs de última generación, y basada en un esquema unificado de las etapas que componen el cauce de procesamiento. Se intentará enfocar la explicación hacia aquellos aspectos que sean más relevantes a la hora de llevar a cabo computación de carácter general en el procesador gráfico.

## 5.1. La GPU dentro de la arquitectura global del sistema

En cualquier sistema, la comunicación entre CPU y GPU se realiza a través de un puerto dedicado; PCIExpress es a día de hoy el estándar a la hora de realizar la comunicación, aunque debido a la gran difusión del anterior sistema, AGP, es conveniente realizar una descripción del mismo.

AGP (Accelerated Graphics Port), es un puerto de comunicaciones desarrollado como respuesta a la creciente aumento de prestaciones de los procesadores gráficos durante la última etapa de la pasada década, con el consiguiente aumento en la necesidad de velocidad de transferencia de datos entre CPU y GPU. Se trata de un puerto paralelo de 32 bits, con acceso directo al NorthBridge del sistema, y por tanto, permitiendo emular memoria de vídeo sobre la memoria RAM del sistema. La frecuencia del bus en su primera generación era de 66 Mhz, aumentándose en sucesivas generaciones hasta alcanzar los 533 Mhz. Las velocidades de transmisión varían entre 264 MB/s (para AGP 1x) y 2 GB/s (para AGP 8x).

Sin embargo, estos ratios de transferencia no son suficientes para las tarjetas gráficas de última generación. Es por esto por lo que, en 2004, se publicó el estándar PCIExpress; se trata de un desarrollo del puerto PCI que, a diferencia de AGP, basa su comunicación en un enlace serie en lugar de paralelo. Gracias a PCIExpress, se pueden alcanzar ratios de transferencia teóricos de hasta 8 GB/s en sus últimos desarrollos.

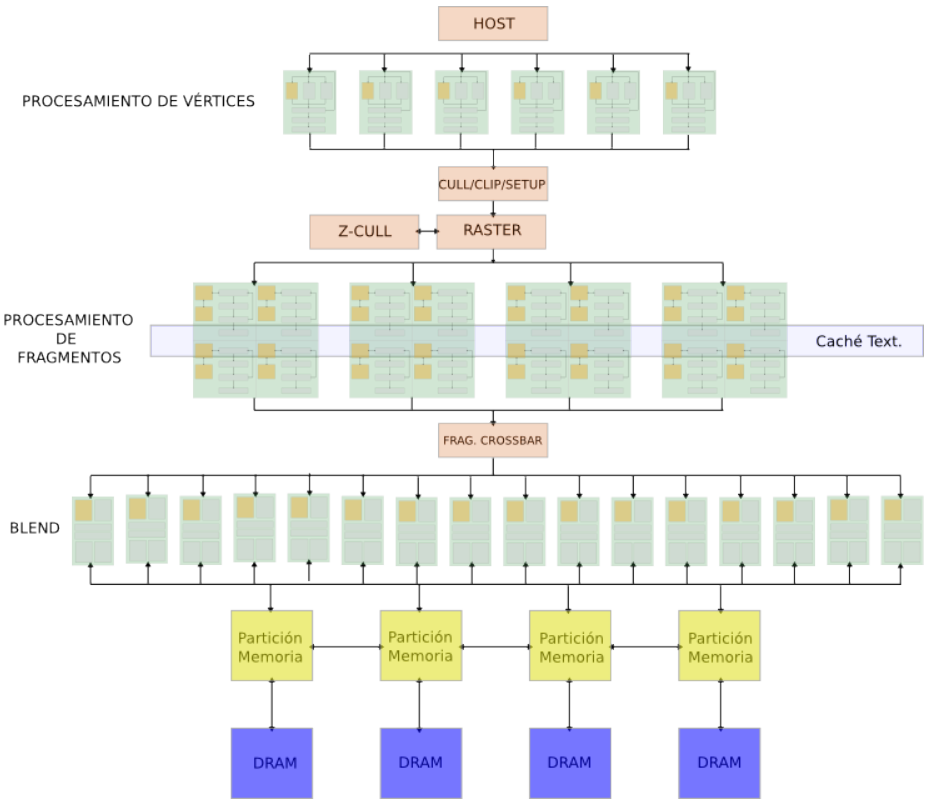



Figura 5.1: Diagrama de bloques de la arquitectura GeForce6 de NVIDIA

## 5.2. La serie Nvidia GeForce6

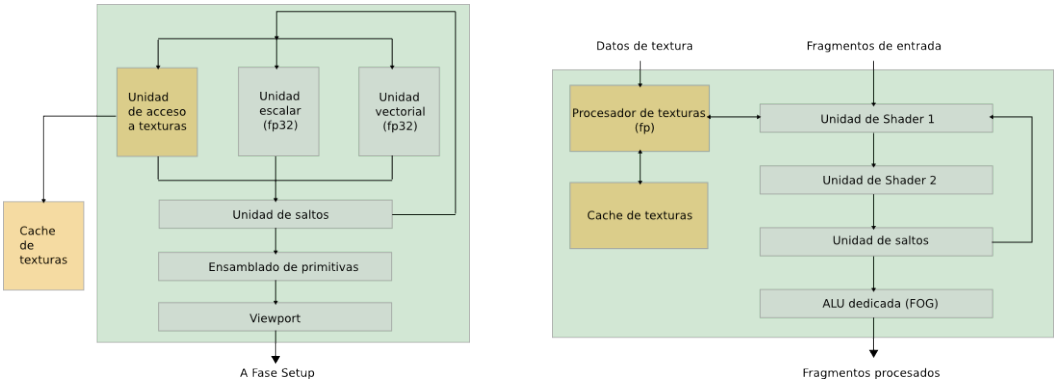
### Diagrama de bloques funcionales

La figura 5.1 muestra, de forma esquemática, los bloques principales que forman la arquitectura GeForce6 de NVIDIAṠe explicarán con más detalle las partes programables del pipeline ya descritas, y se mostrará el proceso que siguen los datos desde su llegada a la GPU desde memoria central hasta su escritura una vez finalizado su procesamiento.

La CPU envía a la unidad gráfica tres tipos de datos: comandos, texturas y vértices. Los procesadores de vértices, también llamados "vertex shaders" (ver figura 5.2), son los encargados de aplicar un programa específico sobre cada uno de los vértices recibidos desde la CPU, llevando a cabo operaciones de transformación sobre ellos. La serie GeForce 6 es la primera que permite que un programa ejecutado en el procesador de vértices sea capaz de consultar datos de textura. Todas las operaciones son realizadas con una precisión de 32 bits en coma flotante (fp32). El número de procesadores de vértices disponibles puede variar entre distintos modelos de procesador, aunque suele oscilar entre dos y dieciséis.

Al ser capaces de realizar lecturas de memoria de textura, cada procesador de vértices

tiene conexión con la cache de texturas; además, existe otra memoria cache, en este caso de vértices, que almacena datos relativos a vértices antes y después de haber pasado por el procesador de vértices.




(a) (b)

Figura 5.2: Procesadores de Vértices (a) y de Fragmentos (b) de la serie GeForce 6 de NVIDIA

Los vértices son seguidamente agrupados en primitivas (puntos, líneas o triángulos). El bloque etiquetado en la figura como Cull/Clip/Setup realiza operaciones específicas para cada primitiva, eliminándolas, transformándolas o preparándolas para la etapa de rasterización. El bloque funcional dedicado a la rasterización calcula qué píxeles son cubiertos por cada primitiva, haciendo uso del bloque Z-Cull[1] para descartar píxeles rápidamente. Ahora, cada fragmento puede ser visto como un candidato a píxel, y está listo para ser tratado por el procesador de fragmentos.

La figura 5.2 muestra la arquitectura de los procesadores de vértices y fragmentos típicos de la serie GeForce 6 de NVIDIA. Cada uno de los procesadores de fragmentos se divide en dos partes: la primera (unidad de texturas) está dedicada al trabajo con texturas, mientras que la segunda (unidad de procesamiento de fragmentos), opera, con ayuda de la unidad de texturas, sobre cada uno de los fragmentos recibidos por el procesador como entrada. Ambas unidades operan de forma simultánea para aplicar un mismo programa (shader) a cada uno de los fragmentos de forma independiente.

De forma similar a lo que ocurría con los procesadores de vértices, los datos de textura pueden almacenarse en memorias cache en el propio chip con el fin reducir el ancho de banda en el enlace memoria de vídeo-procesador requerido para mantener al procesador con la máxima utilización, y aumentar así el rendimiento del sistema.

Las dos unidades de procesamiento (unidad de texturas y unidad de procesamiento de fragmentos), operan sobre conjuntos de cuatro píxeles (llamados *quads*) de forma simultánea. Cada procesador de fragmentos opera sobre grupos mayores de píxeles en modo SIMD , con cada procesador de fragmentos trabajando sobre un fragmento de forma concurrente con el resto.

[1]Es posible utilizar técnicas de control de flujo que se sirven de la unidad Z-Cull para su funcionamiento

El procesador de fragmentos utiliza la unidad de texturas para cargar datos desde memoria (y, opcionalmente, filtrarlos antes de ser recibidos por el propio procesador de fragmentos). La unidad de texturas soporta gran cantidad de formatos de datos y de tipos de filtrado, aunque todos los datos son devueltos al procesador de fragmentos en formato fp32 o fp16.

Cada procesador de fragmentos posee dos unidades de procesamiento que operan con una precisión de 32 bits (*shader units*); los fragmentos circulan por ambas unidades y por la unidad de saltos antes de ser encaminados de nuevo a las unidades de procesamiento para ejecutar el siguiente conjunto de 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 caso de que se produzca una lectura de datos de textura en la primera unidad de procesamiento.

La memoria del sistema se divide en cuatro particiones independientes, cada una de ellas construida a partir de memorias dinámicas (DRAM), con el fin de reducir costes de fabricación. Todas los datos procesados por el pipeline gráfico son almacenados en memoria DRAM, mientras que las texturas y los datos de entrada (vértices), pueden almacenarse tanto en memoria DRAM como en la memoria principal del sistema. Estas cuatro particiones de memoria proporcionan un subsistema de memoria ancho (256 bits) y flexible, consiguiendo 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 2 transferencias por ciclo).

Por tanto, comparando la implementación realizada por esta serie de procesadores (muy similar a otras series de la misma generación), es posible identificar qué unidades funcionales corresponden con cada una de las etapas del *pipeline* gráfico analizado en capítulos anteriores. Este tipo de implementaciones han sido las más extendidas, hasta la aparición de la última generación de GPUs, con una arquitectura unificada, sin diferenciación entre las distintas etapas del cauce de procesamiento a nivel de *hardware*, y que se describirá más adelante.

#### Diagrama de bloques funcionales para operaciones convencionales

Resulta interesante realizar una analogía a nivel de bloques funcionales entre la estructura del procesador gráfico en términos de procesamiento de información gráfica, y su estructura cuando hablamos de computación de datos que no tienen relación con los gráficos. La figura 5.3 muestra vistas simplificadas de la arquitectura de la serie GeForce 6 de NVIDIA.

La primera imagen representa la arquitectura cuando es utilizada como pipeline gráfico. Contiene un conjunto de procesadores programables de vértices, y otro de procesadores programables de fragmentos, así como otras unidades de funcionalidad no programable (unidad de carga/filtrado de datos de textura, unidad de escritura en memoria, ...).

La segunda imagen muestra la arquitectura desde otro punto de vista: al utilizar la GPU como procesador de datos no gráficos, ésta puede ser considerada como dos unidades funcionales básicas operando de forma secuencial: el procesador de vértices y el procesador de fragmentos. Ambos utilizan la unidad de texturas como unidad de

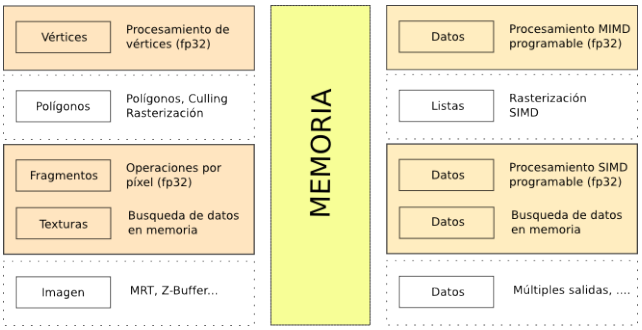



Figura 5.3: Arquitectura de las serie NVIDIA GeForce6 vista como pipeline gráfico (izquierda) o enfocada hacia aplicaciones no gráficas (derecha)

acceso aleatorio a memoria, realizando dichos accesos a velocidades realmente elevadas. Además, como se verá más adelante, cada uno de estos procesadores posee una gran capacidad de proceso por sí solo.

El procesador de vértices opera sobre los datos, transfiriéndolos a su salida directamente al procesador de fragmentos (o bien, realizando entre ambas unidades una operación de rasterizado). Antes de que un fragmento alcance la entrada del procesador, una unidad llamada Z-Cull (ver figura 5.1) compara la profundidad del píxel con los valores de profundidad que ya existen en un buffer especial llamado *depth buffer*. Si la profundidad es mayor, el píxel no será visible, por lo que no tiene sentido tratarlo en el procesador de fragmentos, y se descarta (esta optimización se lleva a cabo siempre y cuando sea claro que el programa cargado en el procesador de fragmentos no va a modificar el valor de profundidad del fragmento). Desde el punto de vista de la computación general, esta característica será útil a la hora de seleccionar sobre qué conjunto de fragmentos no se va aplicar un determinado programa cargado en el procesador de fragmentos. Este tipo de métodos de control de flujo son específicos del hardware gráfico, pero permiten aprovechar unidades funcionales cuyo cometido es específico del procesamiento de gráficos para adaptarlo a la computación general.

Una vez que el procesador de fragmentos opera sobre cada fragmento, éste debe someterse a un conjunto de tests antes de continuar a través del *pipeline*. Realmente, puede haber más de un fragmento obtenido como resultado tras la ejecución en el procesador de fragmentos: este caso se da si se utiliza la técnica llamada MRT (Multiple Render Targets). Es posible utilizar hasta cuatro MRTs para escribir grandes cantidades de datos en memoria.

### Características de la GPU

Es posible realizar una división de las características de la GPU atendiendo a si éstas son fijas o programables. Debido al enfoque del documento hacia la computación general en procesadores gráficos, nos centraremos en las características programables de la serie GeForce 6 de NVIDIA. Con la adopción por parte de Nvidia del modelo de programación Shader Model 3.0 para sus procesadores programables, los modelos de programación para procesadores de vértices y de fragmentos han aproximado sus características: ambos

soportan ahora precisión de 32 bits, consulta de datos de textura y el mismo juego de instrucciones. A continuación se estudiarán las características concretas de cada uno de los procesadores.

### Procesador de vértices

- Número de instrucciones por *shader*: el número de instrucciones por programa es de 512 instrucciones estáticas y hasta 64K instrucciones dinámicas. Las primeras representan el número de instrucciones de un programa tal y como es compilado. Las segundas representan el número de instrucciones realmente ejecutadas (mayor debido a bucles o llamadas a función).
- Registros temporales: cada procesador de vértices puede acceder hasta a 32 registros temporales de cuatro elementos cada uno de ellos.
- Control de flujo dinámico: tanto bucles como estructuras condicionales forman parte del modelo de programación. Ambas construcciones tienen un sobrecoste de dos ciclos. Además, cada vértice puede tomar un camino u otro en función de los saltos, sin penalización de rendimiento, como sí ocurre en el caso de los procesadores de fragmentos.
- Acceso a texturas desde el procesador de vértices.

Cada procesador de vértices es capaz de realizar de forma simultánea una operación MAD (*multiply + add*) sobre vectores de cuatro elementos, así como una función escalar por ciclo de reloj de entre las siguientes:

- Funciones trigonométricas.
- Funciones exponenciales.
- Funciones inversas.

Desde este punto de vista, los procesadores de vértices (y también los de fragmentos) podrían ser comparados con procesadores que se ciñen a arquitecturas VLIW, extrayendo paralelismo a nivel de instrucción.

### Procesador de fragmentos

- Número de instrucciones por *shader*: el número de instrucciones, tanto estáticas como dinámicas, es de 64K. Aún así, existen limitaciones temporales, básicamente a nivel de sistema operativo, sobre el tiempo de ejecución de cada programa. De cualquier modo, el número de instrucciones de un *shader* típico suele ser reducido.
- MRT (*Multiple Render Targets*): el procesador de fragmentos es capaz de proporcionar resultados hasta a cuatro buffers de color independientes (más un quinto buffer de profundidad). Esto significa que, en un misma pasada de renderizado, el procesador es capaz de proporcionar hasta cuatro resultados a partir de un mismo fragmento. Dichos resultados deberán tener el mismo formato de representación y

el mismo tamaño. Esta técnica resulta muy útil al programar sobre datos escalares, puesto que, en una sola ejecución del procesador de fragmentos, es posible obtener hasta 16 valores escalares disponibles para ser escritos en memoria.

- Soporte para precisiones internas de 16 y 32 bits: los programas cargados soportan precisión de 32 o 16 bits tanto a nivel de operaciones como de almacenamiento intermedio.

- Lanzamiento de múltiples instrucciones simultáneas.

  - *3:1 y 2:2 co-issue*: cada unidad de procesamiento, capaz de operar de forma simultánea sobre las cuatro componentes de un mismo vector, es capaz además de realizar dos instrucciones independientes en paralelo sobre dicho vector, de dos formas, como se muestra en la figura 5.4: bien realizando una operación sobre tres elementos (RGB) del mismo y una sobre el canal Alpha, o bien realizando una operación sobre dos de los elementos (canales RG) y una segunda sobre el resto (canales BA). Con esta técnica, el compilador tiene más oportunidades para agrupar datos escalares en vectores, agrupando así también las operaciones sobre ellos.


Figura 5.4: Funcionamiento de la técnica co-issue: dos operaciones independientes pueden ejecutarse de forma concurrente en diferentes partes de un registro de cuatro elementos.

  - *Dual issue*: esta técnica es similar a la anterior, pero con la particularidad de que las dos instrucciones independientes pueden ser ejecutadas en partes diferentes del pipeline del procesador de fragmentos, facilitando así su planificación (véase figura 5.5).

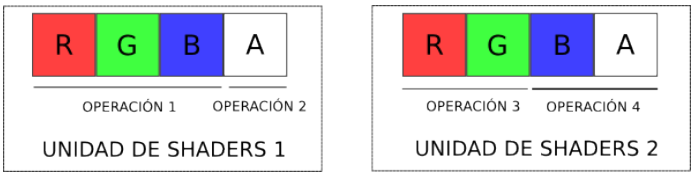



Figura 5.5: Funcionamiento de la técnica dual-issue: instrucciones independientes pueden ser ejecutadas en unidades independientes del pipeline gráfico

### Rendimiento del procesador de fragmentos

La arquitectura del procesador programable de fragmentos de la serie GeForce6 de Nvidia posee las siguientes características desde el punto de vista del rendimiento:

| Instrucción | Coste (Ciclos) |
|---|---|
| If/endif | 4 |
| If/else/endif | 6 |
| Call | 2 |
| Ret | 2 |
| Loop/endloop | 4 |

Tabla 5.1: Sobrecoste asociado a la ejecución de operaciones de control de flujo en programas de fragmentos

- Cada procesador es capaz de realizar, por ciclo de reloj:
  - Una operación MAD, con posibilidad de *co-issue*, sobre vectores de cuatro elementos, o bien un producto escalar (DP4) sobre los cuatro elementos de un vector.
  - Una operación de producto sobre vectores de cuatro elementos, con posibilidad tanto de co-issue como de dual issue.

  Estas operaciones se ejecutan a igual velocidad tanto con precisión de 32 bits como con precisión de 16 bits. Sin embargo, factores externos como la limitación en ancho de banda o en capacidad de almacenamiento pueden hacer que el rendimiento de las aplicaciones trabajando con precisión de 16 bits supere al rendimiento en el caso de trabajar con precisión de 32 bits.

- Independientemente, y gracias al hardware dedicado, es posible realizar la normalización de un vector con precisión de 16 bits en paralelo con las operaciones anteriormente descritas.

- Además, también es posible la ejecución en paralelo con las anteriores de una operación de cálculo de la inversa de cada componente de un vector vía hardware.

- En cuanto al sobrecoste en las instrucciones de salto, éste viene dado por la tabla 5.1. Es necesario estudiar la problemática en el control de flujo en arquitecturas SIMD; el hecho de que todas las unidades de procesamiento deban ejecutar las mismas instrucciones hace que aquellos de ejecución ejecutándose para fragmentos que tomen un salto determinado deban esperar a los flujos que dentro de un mismo programa no lo tomen, y por tanto aumenten su tiempo de finalización, por la pérdida de tiempo que supone la sincronización entre ambos. Como resultado, muchos fragmentos tomarán como tiempo de ejecución el de las dos ramas del salto, además del sobrecoste añadido mostrado en la tabla por el hecho de evaluar la condición.

Por último, y como ejemplo representativo de la serie GeForce6 de Nvidia, se listan las características principales de la GPU GeForce 6800 Ultra (chip Nvidia NV45):

- Reloj de procesador a 425 Mhz.

- Reloj de memoria a 550 Mhz.
- Capacidad de proceso de 600 millones de vértices por segundo, 6.400 millones de *texels* (elementos de textura) por segundo y 12.800 millones de píxeles por segundo.
- 6 procesadores de vértices, lo que conlleva una capacidad de realización de 6 operaciones en coma flotante de 32 bits de precisión sobre vectores de cuatro elementos de tipo MAD por ciclo de reloj en dichos procesadores, además de una operación escalar de forma simultánea (por ejemplo, operación matemática de tipo trigonométrico).
- 16 procesadores de fragmentos disponibles, por lo que es capaz de llevar a cabo 16 operaciones en coma flotante de 32 bits de tipo MAD sobre vectores de cuatro elementos en un ciclo de reloj, de forma simultánea con 16 operaciones de multiplicación sobre datos del mismo tipo.
- Capacidad de llevar a cabo operaciones Z-Cull sobre 64 píxeles por ciclo de reloj.

Resulta significativa la gran diferencia existente entre la velocidad de reloj de este tipo de procesadores (425 Mhz en el caso del procesador estudiado, inferior en el resto de la familia) frente a la velocidad de reloj de los procesadores de propósito general actuales (que superan los 3 Ghz en muchos casos). Sin embargo, hay tres aspectos diferenciadores entre ambos tipos de procesadores:

- Especialización en el trabajo con datos de coma flotante por parte de las GPUs actuales.
- Unidades funcionales de carácter vectorial, capaces de trabajar sobre vectores de cuatro elementos en un mismo ciclo de reloj.
- Replicación de unidades funcionales (procesadores de vértices y de fragmentos).

Estos tres aspectos hacen de las GPUs soluciones a tener muy en cuenta frente a aplicaciones que cumplan una serie de requisitos básicos, llegando a superar, en muchos casos, el rendimiento de los procesadores de propósito general.

## 5.3. La arquitectura G80 de Nvidia

### El concepto de arquitectura unificada

La arquitectura de la serie GeForce 6 de Nvidia anteriormente estudiada podría definirse como una arquitectura dividida a nivel de *shaders* o procesadores programables: existe en ella hardware especializado para ejecutar programas que operan sobre vértices, y otro dedicado exclusivamente a su ejecución sobre fragmentos. La serie GeForce 7 de Nvidia (y correspondientes a la misma generación en productos de otras compañías), se basó en una arquitectura similar a anteriores series, aumentando su potencia en base a un aumento de la superficie de silicio del chip.

Pese a que el hardware dedicado puede adaptarse en mayor medida a su función, existen ciertos inconvenientes que hacen que se haya optado por arquitecturas totalmente

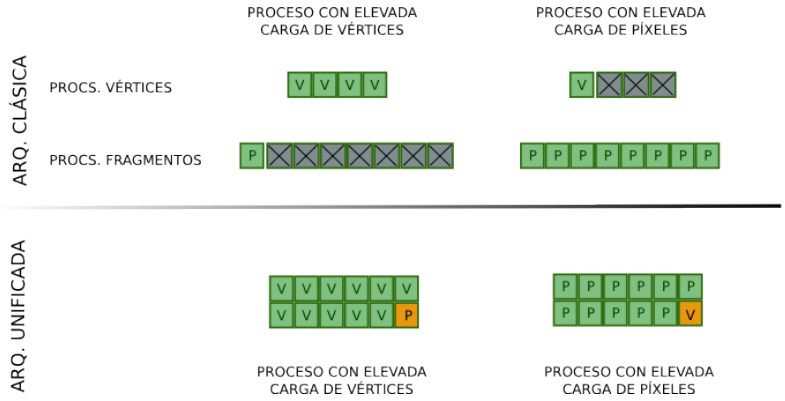



Figura 5.6: Ventajas de la arquitectura unificada: en la parte superior, distribución de la carga entre los distintos tipos de procesadores que forman la GPU, para una arquitectura no unificada, ante dos distribuciones de carga distintas; abajo, distribución de la carga para una arquitectura unificada, ante los mismos tipos de aplicación.

diferentes a la hora de desarrollar una nueva generación de procesadores gráficos, basados en una arquitectura unificada. El problema principal de las arquitecturas anteriores surgía en aplicaciones (gráficas), cuya carga de trabajo a nivel de geometría (vértices) y procesado de píxeles (fragmentos) no estaba equilibrada. Así, aplicaciones con gran carga de trabajo geométrico implicaban una total ocupación de los procesadores de vértices, que habitualmente aparecen en un número reducido en las GPUs, y un desaprovechamiento de capacidad de cálculo a nivel de fragmento, ya que muchas de las unidades procesadoras de fragmentos permanecían ociosas (véase figura 5.6). El mismo problema se da en procesos con alta carga a nivel de fragmentos y bajo coste geométrico.

La solución que se ha adoptado en productos de última generación (G80 de Nvidia o serie R600 de ATI) es la de crear arquitecturas unificadas a nivel de *shaders*. En este tipo de arquitecturas, no existe ya división a nivel de hardware entre procesadores de vértices y procesadores de fragmentos. Cualquier unidad de procesamiento que las forma (llamadas también *Stream Processors*), es capaz de trabajar tanto a nivel de vértice como a nivel de fragmento, sin estar especializado en el procesamiento de ninguno de los dos en concreto. Así, este cambio de arquitectura también conlleva un cambio en el pipeline gráfico: con la arquitectura unificada, ya no existen partes específicas del chip asociadas a una etapa concreta del pipeline, sino que una única unidad central de alto rendimiento será la encargada de realizar todas las operaciones, sea cual sea su naturaleza.

Una de las ventajas que este tipo de arquitecturas presenta es el autoequilibrado de la carga computacional. El conjunto de procesadores pueden ahora asignarse a una tarea u otra dependiendo de la carga que el programa exija a nivel de un determinado tipo de procesamiento. Así, a cambio de una mayor complejidad en cada uno de los procesadores que componen la GPU y de una mayor genericidad en su capacidad de procesamiento, se consigue reducir el problema del equilibrado de carga y de la asignación de unidades de procesamiento a cada etapa del pipeline gráfico.

A continuación se presentará una implementación concreta de la arquitectura unifi-

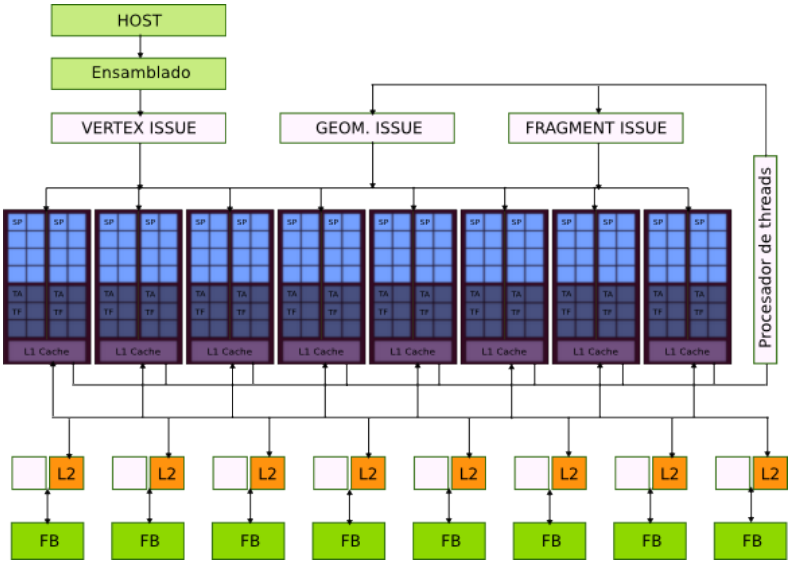



Figura 5.7: Implementación que la serie G80 de Nvidia hace del concepto de arquitectura unificada. Nótese la realimentación entre las salidas de los distintos clusters y sus entradas, hecho que da el carácter cíclico a este tipo de arquitecturas.

cada anteriormente descrita: la arquitectura G80 de Nvidia.

### Descripción de la arquitectura

La arquitectura G80 de Nvidia se define como una arquitectura totalmente unificada, sin diferenciación a nivel hardware entre las distintas etapas que forman en *pipeline* gráfico, totalmente orientada a la ejecución de *threads*, y con autoequilibrado de carga. Opera de forma integral con una precisión de 32 bits para datos flotantes, ajustándose al estándar IEEE 754. La figura 5.7 muestra un esquema completo de la arquitectura G80; en las siguientes secciones se detallará cada una de las partes que lo componen.

La arquitectura G80 de Nvidia comenzó su desarrollo a mediados de 2002, publicando su versión definitiva en los últimos meses de 2006. El objetivo básico de mejora de las capacidades de procesamiento gráfico conllevó:

- Incremento significativo de las prestaciones con respecto a la última generación de GPUs.
- Aumento de la capacidad de cálculo en coma flotante por parte de la GPU, con la vista puesta en la introducción definitiva de este tipo de procesadores en el ámbito de la computación general.
- Adición de nuevos elementos al *pipeline* clásico, para cumplir las características definidas por Microsoft en DirectX 10.

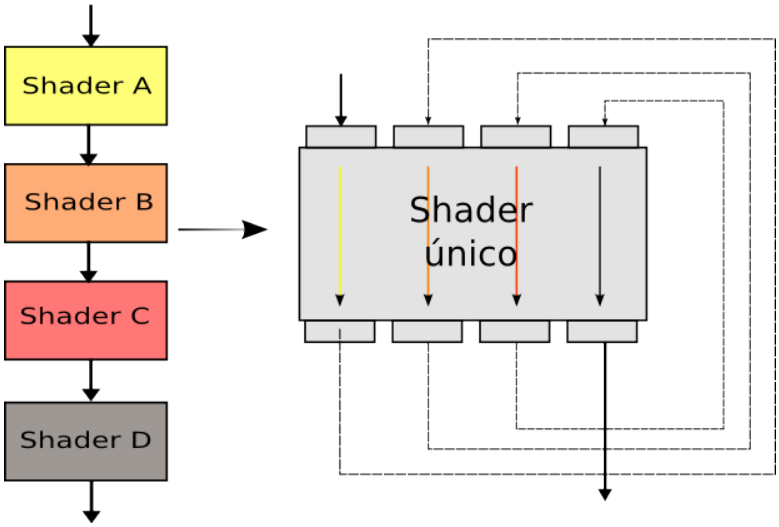



Figura 5.8: Transformación entre el *pipeline* clásico (izquierda) y el modelo unificado (derecha)

Gracias a la arquitectura unificada, el número de etapas del *pipeline* se reduce de forma significativa, pasando de un modelo secuencial a un modelo cíclico, como se muestra en la figura 5.8. El *pipeline* clásico utiliza tipos de shaders distintos, diferenciables en la imagen mediante diferentes colores, a través de los cuales los datos fluyen de forma secuencial. En la arquitectura unificada, con una única unidad de shaders no especializada, los datos que llegan a la misma (en forma de vértices) en primera instancia son procesados por la unidad, enviándose los resultados de vuelta a la entrada para ser procesados de nuevo, realizando operaciones distintas esta vez, y emulando de ese modo el *pipeline* clásico visto en apartados anteriores, hasta que los datos han pasado por todas las etapas del mismo y son encaminados hacia la salida de la unidad de procesamiento.

Realmente, el uso de una arquitectura unificada no implica en sí mismo unas mejores prestaciones. Sin embargo, el estudio del reparto de carga entre procesadores de vértices y procesadores de fragmentos frente a una aplicación determinada, lleva a la conclusión de que, normalmente, existe un desequilibrio entre las unidades de procesamiento de vértices y las unidades de procesamiento de fragmentos utilizadas [2]. Un ejemplo que ilustra este hecho se extrae de gráficas similares a la mostrada en la figura 5.9: se combinan ciclos de trabajo caracterizados por un bajo uso de las unidades de vértices y elevado uso de las de fragmentos con ciclos con características totalmente opuestas.

### Funcionamiento de la GPU

En una primera instancia, los datos recibidos desde la aplicación son preprocesados en hardware específico, con el objetivo de realizar una primera organización de los mismos que aproveche al máximo la capacidad de cálculo del sistema, intentando evitar la existencia de unidades funcionales ociosas durante la posterior ejecución.

[2] Normalmente, las aplicaciones gráficas combinan ciclos de trabajo con cálculos geométricos (de vértices) intensivos, con otros ciclos que hacen un uso intensivo de los procesadores de fragmentos

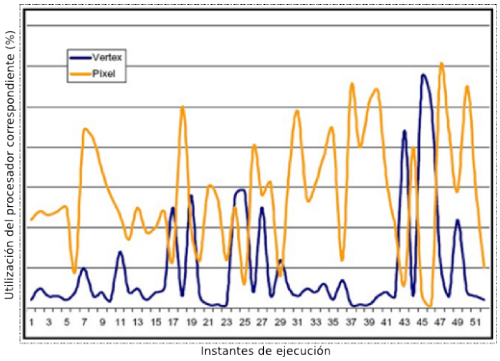



Figura 5.9: Carga de trabajo geométrico y de fragmentos a través del tiempo. Fuente: [8]

A partir de ese momento, el control de la ejecución de cada uno de los *threads* que llevarán a cabo de forma conjunta el cálculo se transfiere a un controlador global de *threads*, que será la unidad encargada de determinar, en cada momento, qué hilos, y de qué tipo (de vértices, de fragmentos o de geometría[3]) serán enviados a cada unidad de procesamiento de las que componen el procesador gráfico.

Cada una de las unidades de procesamiento poseerá, a su vez, un planificador de *threads* propio que decidirá cómo se llevará a cabo la gestión interna de *threads* y datos dentro de la misma.

### Unidades escalares de procesamiento: SPs

El núcleo de procesamiento de *shaders* está formado por ocho *clusters* de procesamiento [4]. Cada uno de dichos clusters estará formado, a su vez, por 16 unidades de procesamiento principal, llamados *Streaming Processors* (SP). Existe un planificador de threads asociado a cada cluster, así como memoria cache de nivel 1 y unidades de acceso y filtrado de texturas propias. Por tanto, cada grupo de 16 SP agrupados dentro de un mismo cluster comparten tanto unidades de acceso a texturas como memoria cache de nivel 1. La figura 5.10 muestra, de forma esquemática, la estructura de cada uno de los clusters que forman la arquitectura G80 de Nvidia.

Cada SP está diseñado para llevar a cabo operaciones matemáticas, o bien direccionamiento de datos en memoria y posterior transferencia de los mismos. Cada procesador es una ALU operando sobre datos escalares (a diferencia del soporte vectorial ofrecido por anteriores arquitecturas en sus procesadores de fragmentos) de 32 bits de precisión (IEEE 754). Es capaz de lanzar dos operaciones (*dual-issue*) por ciclo de reloj de tipo MADD y MUL. Cada uno de los *Stream Processor* trabaja a 1.35 Ghz (nótese la diferencia en cuanto a frecuencia de reloj con respecto a anteriores generaciones de

[3]Las unidades de geometría, novedad en las últimas generaciones de procesadores gráficos, son unidades programables capaces de trabajar a nivel de primitivas (triángulos, por ejemplo) tomando primitivas completas como entrada y generando primitivas completas como salida.

[4]Al igual que en la arquitectura de la serie GeForce 6, el número de procesadores variará en función de la versión del procesador que se esté utilizando

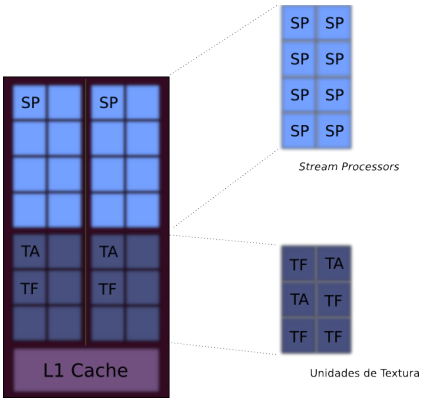



Figura 5.10: *Streaming Processors* y Unidades de Textura que componen un *cluster* en la arquitectura G80

GPUs).

La utilización de unidades de cálculo escalar frente a las unidades de las que se hacía uso hasta esta generación de GPUs, con capacidad vectorial, supone un punto de inflexión en la concepción de las unidades de procesamiento gráfico. La razón por la cual la arquitectura vectorial se había convertido en un estándar en los procesadores gráficos radica en el tipo de operaciones a ejecutar sobre ellos: una gran parte de las mismas se llevan a cabo sobre vectores de datos (por ejemplo, sobre vectores de cuatro elementos, RGBA, en el caso de los procesadores de fragmentos). Sin embargo, no resulta despreciable el número de operaciones escalares que se realizan durante la ejecución de un programa sobre GPU, más aún cuanto más complejos son los cálculos a realizar sobre ellos. Como consecuencia, resulta complicado aprovechar al máximo todas las unidades de procesamiento vectorial de las que dispone la GPU si un elevado número de operaciones a ejecutar son escalares, aunque se haga uso de características como lanzamientos *dual-issue* como los vistos para la serie GeForce 6.

Internamente, cada cluster esta organizado en dos grupos de 8 SPs; el planificador interno lanzará a ejecución una misma instrucción sobre cada uno de los dos subconjuntos de SP que forman el cluster, tomando cada una un cierto número de ciclos de reloj, que vendrá definido por el tipo de thread que se esté ejecutando en la unidad en ese instante.

Dado que existen 8 agrupaciones de 16 Streaming Processors, sería posible abstraer la arquitectura, representándola como una configuración MIMD de 8 nodos de computación, cada uno de ellos formado por 16 unidades o clusters de procesamiento SIMD (los SP).

Cada cluster, además de tener acceso a su propio conjunto de registros, accederá también al fichero de registros global, así como a dos zonas de memoria adicionales, de solo lectura, llamadas *cache global de constantes* y *cache global de texturas*.

Cada uno de los clusters puede procesar threads de tres tipos: vértices, geometría y píxeles (o fragmentos), de forma independiente y en un mismo ciclo de reloj. Así, uniendo el planificador interno de cada cluster con la capacidad de proceso de los tres

tipos de thread disponibles, se sustenta la arquitectura unificada propuesta por Nvidia, caracterizada pues por la autoplanificación interna de cada cluster, el equilibrado global de la carga entre los clusters disponibles y la capacidad de cada unidad de procesamiento de ejecutar cualquier tipo de operación en un determinado ciclo de reloj.

La naturaleza orientada a hilos del procesador también se da a nivel de lectura de datos en cada cluster; cada uno de ellos posee unidades que permiten la ejecución de hilos dedicados exclusivamente a la transferencia de datos con memoria (unidades TF (*Texture Filtering*) y TA (*Texture Addressing*), con el objetivo de solapar en la medida de lo posible los accesos a memoria con el cálculo , y mantener una elevada eficiencia en cada instante.

### Jerarquía de memoria

Además de acceso a su fichero de registros dedicado, al fichero de registros global, a la cache de constantes y a la cache de texturas, cada cluster puede compartir información con el resto de clusters a través del segundo nivel de cache (L2), aunque únicamente en modo lectura. Para compartir datos en modo lectura/escritura, es necesario el uso de la memoria DRAM de vídeo, con la consiguiente penalización temporal que este hecho supone.

Detallando más la configuración de memoria de la arquitectura G80 de Nvidia, cada uno de los clusters de SP posee los siguientes tipos de memoria:

- Un conjunto de registros de 32 bits locales a cada cluster.
- Una caché de datos paralela o *memoria compartida*, común para todos los clusters, y que implementa el espacio de memoria compartido.
- Una memoria de sólo lectura llamada cache de constantes, compartida por todos los clusters, que acelera las lecturas desde el espacio de memoria de constantes.
- Una memoria de sólo lectura llamada cache de texturas, compartida por todos los clusters, que acelera las lecturas desde 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, consiguiendo así una interfaz combinada de 384 bits. El tipo de memoria más utilizado es GDDR3, funcionando a 900 Mhz; con esta configuración, se consiguen tasas de transferencia entre memoria y procesador de hasta 86.4 GBps.

Uno de los mayores cuellos de botella en la computación sobre GPUs se centra en la velocidad de acceso a memoria. Las operaciones asociadas suelen ser las de direccionamiento de datos en texturas, y transferencia y filtrado de los mismos. Mientras que en la serie GeForce 6, la operación de transferencia de un dato desde memoria de vídeo (o memoria central) hacía que la unidad de procesamiento quedase bloqueada a la espera del mismo, la serie G80 permite que mientras un *thread* está ejecutando una operación de consulta sobre una textura, la ejecución pase a otro *thread*, asegurando así un máximo aprovechamiento la unidad de procesamiento correspondiente.

### Modificaciones sobre el pipeline clásico

La serie 8 de Nvidia, así como la mayoría de GPUs de su misma generación, introduce una serie de mejoras sobre el *pipeline* estudiado para arquitecturas de generaciones anteriores. La mayoría de estas modificaciones son consecuencia de la introducción de las recomendaciones del *Shader Model 4* dictado por DirectX.

Con respecto a modelos anteriores (la serie GeForce 6 se ceñía a *Shader Model 3*), se introducen gran cantidad de mejoras, entre las que destacan:

- Conjunto unificado de instrucciones.
- Aumento del número de registros y constantes utilizables desde un mismo *shader*.
- Número de instrucciones ilimitado para los *shaders*.
- Menor número de cambios de estado, con menor intervención de la CPU.
- Soporte para MRT con 8 destinos distintos, en lugar de 4.
- Posibilidad de recirculación de los datos entre los distintos niveles del *pipeline*.
- Control de flujo dinámico tanto a nivel de shaders de vértices y de píxeles.
- Introducción de operaciones específicas sobre enteros.
- Aumento del tamaño máximo de textura 2D. Una de las limitaciones existentes en anteriores generaciones era la limitación a $4096 \times 4096$ elementos de las texturas 2D utilizadas. La nueva generación de GPUs soporta texturas de hasta $8192 \times 8192$ (con posibilidad de almacenar 4 valores por elemento), lo que resulta una ventaja a tener en cuenta.

Otra de las nuevas características de la serie G80, llamada *Stream Output*, permite que los datos generados desde las unidades dedicadas a geometría (o vértices, en el caso de no utilizar *shaders* de geometría) sean enviados a buffers de memoria y a continuación reenviados de nuevo al inicio del *pipeline* gráfico para ser procesados de nuevo. Se trata de una técnica similar a la anteriormente estudiada *render-to-buffer*, pero más generalizada para llevarse a cabo independientemente de la etapa del *pipeline* que se esté ejecutando.

### CUDA

La razón principal que se oculta tras la evolución del poder computacional de las nuevas GPU es su especialización para realizar cálculo intensivo y computación masivamente paralela y por tanto, su diseño orientado a dedicar más superficie de silicio al procesamiento a costa de otros aspectos importantes en otras arquitecturas como caches de datos o control de flujo.

Así, las GPU están orientadas a problemas con alto nivel de paralelismo de datos y gran cantidad de operaciones aritméticas sobre ellos. Como cada programa es ejecutado por cada elemento de datos, se requieren métodos de control de flujo menos sofisticados, y debido la alta densidad de cálculo ejecutado por elemento, la latencia de acceso a

memoria queda en un segundo plano, por lo que no son estrictamente necesarias caches de datos de elevado rendimiento.

Sin embargo, el acceso a todo el poder computacional que hoy en día ofrecen las GPU, se veía limitado por tres razones principales:

- La GPU sólo podía ser programada a través de un API gráfico, caracterizado en la mayoría de casos por su complejidad y por el sobrecoste que introducía al no estar orientado a aplicaciones de carácter general, sino únicamente a gráficos.
- Como se ha explicado con anterioridad, la DRAM de las GPU puede ser leída de forma aleatoria (operación *gather*); sin embargo, los programas ejecutados en GPU no pueden realizar operaciones de scatter sobre cualquier dirección de memoria, restando flexibilidad a los programas desarrollados para procesadores gráficos pero en el ámbito de la computación general.
- Algunas aplicaciones se veían limitadas por el cuello de botella que supone el ancho de banda limitado de la memoria gráfica.

Una de las soluciones que se ha propuesto para solucionar esta serie de problemas es CUDA. CUDA (*Compute Unified Device Architecture*) es una arquitectura hardware y software (modelo de programación) que permite realizar cálculos en la GPU, tratándola como un dispositivo de proceso paralelo de datos, sin necesidad de utilizar para ello APIs gráficos.

Desde el punto de vista del software, CUDA presenta una estructura por capas: un driver hardware, un API para programación y bibliotecas de más alto nivel (en la actualidad, Nvidia provee bibliotecas para BLAS y FFT). Además, el API de CUDA se presenta como una extensión al lenguaje C para mayor facilidad de uso.

Desde el punto de vista del acceso a memoria, CUDA proporciona al programador acceso general a la DRAM gráfica, sin limitaciones en cuanto a direccionamiento de lectura o de escritura. Así, tanto la operación de *gather* como la de *scatter* están soportadas, y por tanto, es posible realizar lecturas y escrituras en cualquier dirección de memoria, como ocurre en CPUs.

Por último, CUDA introduce un nuevo nivel en la jerarquía de memoria, en forma de cache de datos paralela (memoria compartida en el propio chip por todas las unidades de procesamiento), de mayor velocidad tanto a nivel de lectura como de escritura, útil para que los distintos *threads* en ejecución compartan datos. Así, se reduce el cuello de botella causado por las limitaciones del ancho de banda disponible para el acceso a memoria gráfica. CUDA es todavía un concepto en desarrollo; para más información, véase [9].

#### Implementación hardware y modelo de ejecución

CUDA define un modelo abstracto de dispositivo que debe cumplir ciertas características tanto a nivel de hardware como a nivel de modelo de ejecución. Este dispositivo se implementará como un conjunto de multiprocesadores, cada uno de ellos con una arquitectura SIMD. En cada ciclo de reloj, cada procesador perteneciente a un mismo multiprocesador ejecutará una misma instrucción, pero operando sobre distintos datos.

Un conjunto de *threads* que ejecuta un mismo *shader* o *kernel* se organiza de la siguiente forma:

- Un bloque de *threads* es un conjunto de *threads* cooperantes que comparten datos a través de un espacio de memoria compartido y de alta velocidad. Además, los *threads* dentro de un mismo bloque pueden sincronizar su ejecución mediante puntos de sincronización insertados en el propio *kernel*. Cada uno lleva asociado su identificador, que corresponde al número de *thread* dentro de un bloque.
- Existe una limitación en el número de *threads* que pueden formar parte de un bloque; sin embargo, es posible unir bloques que ejecuten el mismo *kernel* en una nueva estructura llamada *grid* de bloques. Pese a suponer una ventaja, por elevar el número de *threads* por conjunto, los miembros de un mismo *grid* pero de distintos bloques no pueden sincronizarse ni comunicarse con otros miembros.

Además, cada multiprocesador posee necesariamente memoria en el propio chip. Los tipos de memoria exigidos por CUDA son los implementados por la serie G80 de Nvidia, ya estudiados en su momento, y que pueden abstraerse como:

- Registros de lectura-escritura a nivel de *thread*.
- Memoria local de lectura-escritura a nivel de *thread*.
- Memoria compartida de lectura-escritura a nivel de *bloque*.
- Memoria global de lectura-escritura a nivel de *grid*.
- Memoria de constantes, de sólo lectura, a nivel de *grid*.
- Memoria de texturas, de sólo lectura, a nivel de *grid*.

La figura 5.11 muestra de forma resumida la anterior distribución de los espacios de memoria en dispositivos compatibles con CUDA.

Desde el punto de vista del modelo de ejecución, un *grid* es ejecutado en GPU distribuyendo uno o más bloques de *threads* entre cada multiprocesador. Cada uno de los bloques se divide en grupos de *threads* SIMD llamados *warps*, con un número constante de hilos de ejecución, y es ejecutado por el multiprocesador en modo SIMD: el planificador de *threads*, de forma periódica, elige el *warp* a ejecutar para maximizar el uso de los recursos disponibles.

#### Especificaciones técnicas de la serie G80

A modo de ejemplo, veremos a continuación las especificaciones técnicas de la serie 8800 de Nvidia, por ser representativa de la arquitectura G80. Cada unidad de la serie 8800 cumple con las siguientes especificaciones:

- El número máximo de *threads* por bloque es de 512.
- El tamaño máximo de cada *grid* de bloques es 64K.

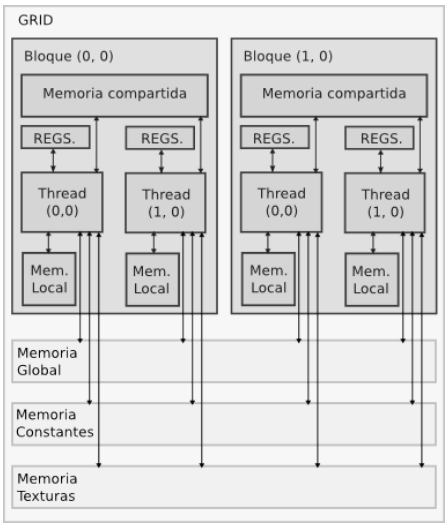



Figura 5.11: Modelo de memoria propuesto por CUDA

- El tamaño de *warp* es de 32 threads.
- El número de registros por cluster es de 8K.
- La cantidad de memoria compartida disponible por cluster es de 16 Kbytes, dividida en 16 bancos de memoria, esta memoria puede ser accedida por todos los *threads* de un mismo *warp*, a velocidad similar a la de los registros. Cada banco de memoria compartida puede ser accedido de forma simultánea a otros bancos. Los datos se distribuyen de forma que palabras sucesivas de 32 bits son asignadas a bancos sucesivos; el acceso sobre una palabra de 32 bits conlleva dos ciclos de reloj.
- La cantidad de memoria de constantes disponible es de 64 Kbytes, con caches de 8 Kbytes por cluster (la cache de texturas tiene el mismo tamaño por cluster).
- La cantidad de memoria caché de texturas por multiprocesador es de 8 Kbytes por multiprocesador, optimizada además para explotar la localidad espacial en 2 dimensiones.
- El máximo número de bloques que pueden ejecutarse concurrentemente en un multiprocesador es de 8.
- El máximo número de *warps* que pueden ejecutarse concurrentemente en un multiprocesador es de 24.
- El máximo número de *threads* que pueden ejecutarse concurrentemente en un multiprocesador es de 768.

A nivel de instrucción, cada multiprocesador o cluster toma los siguientes ciclos de reloj para llevar a cabo el lanzamiento de instrucciones:

- Instrucciones aritméticas: 4 ciclos de reloj para suma, producto y MAD sobre operandos en coma flotante, suma y operaciones a nivel de bit en operandos enteros, y otras operaciones como máximo, mínimo o conversión de tipo. Las operaciones logarítmicas o de cálculo de inversa conllevan 16 ciclos de reloj. La alta especialización de este tipo de procesadores en operaciones sobre coma flotante queda demostrada por el coste de una operación de multiplicación sobre datos enteros de 32 bits: 16 ciclos de reloj.

- Instrucciones de acceso a memoria: los accesos a memoria tienen un sobrecoste de 4 ciclos de reloj, tanto a nivel de memoria global como de memoria compartida; además, el acceso a memoria global suele acarrear entre 400 y 600 ciclos de reloj debido a la latencia propia del acceso a niveles inferiores en la jerarquía de memoria.

- Instrucciones de sincronización: una instrucción de sincronización a nivel de *threads* supone un sobrecoste de 4 ciclos de reloj, más el tiempo de espera propio de la sincronización de todos los hilos de ejecución.

# Parte III

# Evaluación del rendimiento

Capítulo 6

# *Software* desarrollado

## 6.1. Objetivos

Uno de los mayores inconvenientes del desarrollo de programas cuyo destino de ejecución es la GPU es la dificultad de programación de las mismas, más aún cuando el programador no dispone de una base de conocimientos relativos a la programación de aplicaciones gráficas.

Uno de los objetivos del presente proyecto es la obtención de una interfaz de programación que facilite, en la medida de lo posible, el desarrollo de aplicaciones de carácter general sobre procesadores gráficos. Dicha interfaz servirá como base para el desarrollo de bibliotecas que permitan evaluar el rendimiento de la GPU ante determinado tipo de operaciones (básicamente operaciones de álgebra lineal, aunque se implementarán también operaciones de tratamiento de imágenes).

En última instancia, se realizará una evaluación del rendimiento de las rutinas implementadas, comparándolas con rutinas optimizadas para el caso de ejecución en CPU. Para completar el estudio, se han desarrollado una serie de programas de evaluación del rendimiento; se trata de microbenchmarks de carácter específico, cuya finalidad es la de medir las prestaciones de puntos concretos de la GPU.

Los resultados obtenidos servirán para extraer conclusiones acerca de qué tipo de operaciones de carácter general son adecuadas para ser implementadas de forma eficiente sobre GPUs.

## 6.2. Interfaz de programación de GPUs: AGPlib

### Conceptos generales

AGPlib (*Algebra for Graphics Processors library*) es una biblioteca de funciones desarrollada en lenguaje C, cuyo objetivo principal es el de actuar como una interfaz de programación de GPUs sencilla para programadores no habituados a la programación sobre este tipo de hardware.

Existen otros sistemas similares para el desarrollo de programas sobre GPUs. Brook[1]

[1] http://graphics.stanford.edu/projects/brookgpu/

es un compilador y entorno de ejecución que permite el desarrollo, a través de un lenguaje propio, de algoritmos orientados a flujos sobre GPUs; Sh[2] es una biblioteca de funciones escrita en C++ que permite programar GPUs tanto para aplicaciones gráficas como para aplicaciones de carácter general de forma sencilla. Sin embargo, uno de los objetivos del proyecto consistía en la familiarización con todos aquellos aspectos de bajo nivel relativos a programación de GPUs que pueden resultar complicados para programadores con poca experiencia a la hora de desarrollar programas sobre GPUs. Por tanto, se consideró que el desarrollo de una interfaz de estas características resultaría realmente útil como primera toma de contacto con la programación de GPUs, y a la vez resultaría posible la adaptación de ciertas características de la propia interfaz al tipo concreto de aplicaciones a evaluar.

La infraestructura de una aplicación ejecutada sobre GPU resulta realmente compleja. Todas las aplicaciones se fundamentan en el uso de interfaces de programación 3D, como OpenGL o Direct3D. Por su flexibilidad, extensibilidad y capacidad de ejecución en múltiples plataformas, AGPlib se ha construido sobre OpenGL.

Sin embargo, OpenGL no ofrece por sí mismo al programador capacidad para llevar a cabo las tareas de programación de los procesadores de vértices y de fragmentos estudiados. De hecho, el objetivo principal de AGPlib es ofrecer capacidades de programación de estas unidades funcionales del procesador gráfico.

Existe una familia de lenguajes dedicados exclusivamente a la programación de los procesadores de vértices y fragmentos. Entre ellos, destacan GLSL, creado por el consorcio desarrollador de OpenGL, y Cg, desarrollado por Nvidia y Microsoft. Ambos lenguajes ofrecen la posibilidad de llevar a cabo la programación de las dos etapas programables del pipeline gráfico sin necesidad de recurrir a lenguaje ensamblador, proporcionando tipos de datos adaptados a los utilizados internamente por el procesador, así como construcciones típicas de los lenguajes de programación de alto nivel (bucles, estructuras condicionales, ...).

Para el desarrollo de AGPlib se ha escogido Cg como como lenguaje de programación de la GPU; sin embargo, su traducción a otros lenguajes como GLSL sería sencillo y se presenta como una de las posibles extensiones de la interfaz de cara al futuro. Una buena fuente de información sobre el lenguaje Cg es [4].

### Principales componentes de AGPlib

Cada uno de los conceptos teóricos estudiados en el capítulo 4 se ha tenido en cuenta en el desarrollo de AGPlib. De hecho, como se verá más adelante, cada uno de los conceptos estudiados tiene su implementación asociada en AGPlib. Se estudiarán a continuación los detalles de implementación de cada uno de ellos:

**Estructura de datos AGPBuffer:** la estructura de datos AGPBuffer es una abstracción de las estructuras de datos matriz utilizadas en CPUs. Como ya se ha estudiado, la estructura de datos asociada a las matrices cuando la computación se lleva a cabo en GPUs es la textura. La diferencia más significativa a la hora de trabajar con texturas es la forma de acceder a cada uno de sus elementos: mientras que en

[2] http://libsh.org

la CPU el direccionamiento se realiza mediante índices, al trabajar con texturas éste se lleva a cabo mediante coordenadas de textura.

Además AGPlib ofrece rutinas para la creación (`AGPCreateBuffer`), transferencia de datos entre memoria central y AGPBuffer (`AGPTransferToBuffer`) y transferencia inversa, entre AGPBuffer y memoria central (`AGPTransferFromBuffer`).

Un ejemplo sencillo de aplicación que hace uso de rutinas de AGPlib para el trabajo con texturas podría ser la transferencia a memoria de vídeo de los datos contenidos en una matriz en memoria central, y su posterior escritura de vuelta en memoria central. Una posible solución resultaría:

Transferencia de datos CPU-GPU y viceversa

```
#include <AGPlib.h>

int main( void ){

  AGPBuffer GPUBuffer;
  int n, m;
  float * CPUMatrix;

  CPUMatrix = ( float * )malloc( n * m * sizeof( float ) );

  /* Inicializacion de la matriz en CPU */
  ...

  n = m = 16;

  /* Inicializacion del entorno */
  AGPInit( AGP_MODE_FOUR );
  AGPSetEnvironment( argc, argv );

  /* Creacion de los buffers en GPU */
  AGPCreateBuffer( &GPUBuffer, n/2, m/2, AGP_READ );

  /* Transferencia de datos a GPU */
  AGPTransferToBuffer( &GPUBuffer, CPUMatrix );

  /* Transferencia de vuelta a CPU */
  AGPTransferFromBuffer( &GPUBuffer, CPUMatrix );

}
```

A la vista del anterior código, cabe destacar la existencia de un conjunto de rutinas para la inicialización del entorno sobre el que se ejecutará la aplicación. Uno de los factores clave a la hora de diseñar la aplicación es la elección del formato de almacenamiento de los datos dentro de las texturas; AGPlib soporta tanto el modo de almacenamiento consistente en un elemento por cada *texel*[3], como de cuatro elementos por *texel*. Debido a que, en el ejemplo anterior, se ha optado

[3]Denominaremos *texel* a cada uno de los elementos de una textura

por este último formato de almacenamiento, el número de elementos de la textura será cuatro veces inferior al número de elementos existentes en la matriz original, de ahí las dimensiones especificadas a la hora de crear la estructura de datos `AGPBuffer` en memoria de vídeo.

**Definición de *shaders* y parámetros para los mismos:** Cada *kernel* (también llamado *shader*) programado será ejecutado sobre cada uno de los elementos que componen el flujo de datos. AGPlib proporciona rutinas tanto para definir nuevos *shaders* como para asociar a los mismos sus parámetros correspondientes. El siguiente código sirve para ilustrar el funcionamiento de esta familia de rutinas:

Programa para la implementación del escalado de matrices

```
int main( void ){

  AGPBuffer GPUBuffer;
  AGPShader GPUShader;

  int n, m;
  float * CPUMatrix;

  float alpha = 2;

  CPUMatrix = ( float * )malloc( n * m * sizeof( float ) );

  /* Inicializacion de la matriz en CPU */
  ...

  n = m = 16;

  /* Inicializacion del entorno */
  AGPInit( AGP_MODE_FOUR );
  AGPSetEnvironment( argc, argv );

  /* Creacion de los buffers en GPU */
  AGPCreateBuffer( &GPUBuffer, n/2, m/2, AGP_READ );

  /* Transferencia de datos a GPU */
  AGPTransferToBuffer( &GPUBuffer, CPUMatrix );

  /* Programacion de la GPU */
  AGPLoadShader( &GPUShader, "./sum.cg", "func_sum" );
  AGPBindBufferToShader( &GPUShader, "flujoA", &GPUBuffer );
  AGPBindFloatToShader( &GPUShader, "alpha", alpha );

  AGPRunShader( &GPUShader, &GPURange,  &GPUBuffer );

  /* Transferencia de vuelta a CPU */
  AGPTransferFromBuffer( &GPUBuffer, CPUMatrix );

}
```

Su función es la de, dado un vector de números reales de entrada, multiplicar en GPU cada uno de sus elementos por 2, y transferir de vuelta el resultado a memoria central.

Las rutinas añadidas en este programa con respecto al anterior son rutinas propias de AGPlib, y relacionadas con la definición del *shader* a ejecutar:

- `AGPLoadShader`: rutina de definición del *shader* o programa a utilizar. Recibe como parámetros de entrada el fichero en el que se encuentra el código fuente del shader, y la función utilizada como punto de entrada en el momento de la ejecución. Realiza la compilación del mismo de forma dinámica (el shader es compilado en el momento de la ejecución del programa), así como la carga del mismo en la GPU.
- `AGPBindBufferToShader`: asocia un *buffer* (una textura) a un determinado parámetro de entrada de un shader. Recibe como parámetros el *shader* al cual se asociará el buffer, el nombre del parámetro y la estructura de datos `AGPBuffer` que se asociará al mismo.
- `AGPBindFloatToShader`: asocia un valor escalar a uno de los parámetros del *shader* definido. Toma como parámetros el *shader* sobre el cual se trabaja, el nombre el parámetro asociado y el dato escalar a asociar. AGPlib ofrece también rutinas para asociar valores vectoriales (de dos, tres y cuatro elementos) a parámetros de los *shaders*.
- `AGPRunShader`: lleva a cabo la invocación del *shader* sobre cada uno de los datos de entrada que formarán el flujo definido. Recibe como parámetros el *shader* a ejecutar, el rango de ejecución (más adelante se aportarán detalles sobre la implementación de este concepto), y el *buffer* sobre el cual se depositarán los resultados tras la ejecución.

El código del *shader* se especificará en el fichero `sum.cg` (tal y como se ha definido en la llamada a la rutina `AGPLoadShader`), presentando el siguiente aspecto:

Shader de ejemplo para escalado de matrices

```
float4 func_sum( float2 coordenadas: TEXCOORD0,
                 uniform samplerRECT flujoA,
                 uniform float alpha) : COLOR{

  return alpha * texRECT( flujoA, coordenadas );
}
```

A la vista del anterior programa, cabe destacar ciertos aspectos importantes:

- El programa definido será ejecutado sobre cada uno de los elementos que conforman el flujo de datos creado en el programa principal (en este caso, sobre cada uno de los elementos que forman el vector `CPUMatrix`).
- En el programa principal anteriormente definido, se escogió una representación interna de los datos en textura que almacenaba cuatro elementos por *texel*. Por tanto, cuando el anterior *shader* sea ejecutado sobre un elemento concreto

de la textura de entrada, estará realmente operando sobre cuatro elementos del vector original de forma simultánea. Es por esto que su tipo de retorno debe ser un vector de cuatro elementos flotantes (en Cg, el tipo reservado para esta estructura de datos es `float4`).

Por tanto, la operación de multiplicación es realizada también sobre los cuatro canales (RGBA) del *texel* correspondiente de forma simultánea.

- Como se ha comentado anteriormente, el acceso a cada elemento de una textura se realiza en GPU mediante coordenadas de textura. Cg reserva la función `texRECT`, que permite el acceso (en modo lectura) a elementos aleatorios de texturas a partir de sus coordenadas de textura.

  Como se describió en el capítulo 4, los *shaders* corresponden a los núcleos de computación asociados a los bucles internos de los algoritmos a ejecutar. De hecho, y a la vista de los programas de ejemplo mostrados, en ningún momento se especifican los índices de los bucles externos que recorren el vector de entrada. Estos índices (o coordenadas) son asociados de forma implícita a cada fragmento tras la etapa de rasterización previa al procesamiento de fragmentos, sin que el programador deba realizar ninguna acción adicional.

  En Cg, se reserva la *semántica* [4] `TEXCOORDO`, asociada a una determinada variable, para asignar a dicha variable las coordenadas del fragmento correspondiente calculadas en la etapa anterior del *pipeline* gráfico.

  Así, en el shader de ejemplo, la variable de entrada de nombre `coordenadas`, recibiría, de forma transparente al programador, las coordenadas asociadas al fragmento sobre el cual está trabajando en cada momento.

  Nótese además el uso de la palabra reservada `uniform` asociada a ciertos parámetros de entrada. Una variable declarada como `uniform` indica que su valor inicial proviene de un entorno externo al programa Cg sobre el que se define.

**Creación de dominios y rangos computacionales:** Resulta interesante ofrecer la posibilidad de definir qué datos concretos pasarán a formar parte de cada uno de los flujos de datos que se lanzarán sobre la GPU para que ésta opere sobre ellos. Para ello, AGPlib soporta la definición de dominios y rangos de ejecución. Gracias a los primeros, es posible definir qué elementos de los datos de entrada pasarán a formar parte de los flujos sobre los cuales operará la GPU. La definición del rango de ejecución permite especificar sobre qué elementos del *buffer* de destino se escribirán los resultados obtenidos tras la ejecución del *shader*.

A modo de ejemplo, el siguiente código operaría sobre los cuatro últimos elementos del flujo de datos de entrada, almacenando sus resultados en los cuatro primeros elementos del flujo de salida:

Programa ilustrativo del uso de dominios y rangos de ejecución

[4] Las *semánticas* (del inglés *semantics*) representan el nexo de unión entre un programa Cg y el resto del *pipeline* gráfico. Por ejemplo, las semánticas `POSITION` o `COLOR`, en las fases de procesamiento de vértices o de fragmentos, respectivamente, asociadas a un dato de retorno de un programa, indican el recurso *hardware* al que se le proporciona el dato obtenido tras la ejecución del *shader*.

```
int main( void ){

  AGPBuffer GPUBuffer;
  AGPShader GPUShader;

  AGPDomain GPUDomain;
  AGPRange GPURange;

  int n, m;
  float * CPUMatrix;

  /* Inicializacion de la matriz en CPU */
  ...

  n = m = 16;

  /* Inicializacion del entorno */
  AGPInit( AGP_MODE_FOUR );
  AGPSetEnvironment( argc, argv );

  /* Creacion de los buffers en GPU */
  AGPCreateBuffer( &GPUBuffer, n/2, m/2, AGP_READ );

  /* Transferencia de datos a GPU */
  AGPTransferToBuffer( &GPUBuffer, CPUMatrix );

  /* Definicion de dominios y rango de ejecucion */
  AGPSetDomain( &GPUDomain, &GPUBuffer, n-1, n, m-1, m );
  AGPAddDomain( &GPUShader, &GPUDomain );

  /* Rango: primeros cuatro elementos del flujo */
  AGPSetRange( &GPURange, 0, 1, 0, 1 );

  /* Programacion de la GPU */
  AGPLoadShader( &GPUShader, "./sum.cg", "func_sum" );
  AGPBindBufferToShader( &GPUShader, "flujoA", &GPUBuffer );
  AGPBindFloatToShader( &GPUShader, "alpha", alpha );

  AGPRunShader( &GPUShader, &GPURange, &GPUBuffer );

  /* Transferencia de vuelta a CPU */
  AGPTransferFromBuffer( &GPUBuffer, CPUMatrix );

}
```

AGPlib dispone del siguiente conjunto de funciones para la definición de dominios y rango de ejecución:

- `AGPSetDomain`: permite la definición de un nuevo dominio de ejecución sobre un *buffer* concreto. Toma como parámetros las coordenadas $(x_1, x_2, y_1, y_2)$ del dominio deseado.

- `AGPAddDomain`: asocia el dominio anteriormente creado a un *shader* determinado.
- `AGPSetRange`: permite la definición del rango de ejecución sobre un *shader*. Toma como parámetros las coordenadas $(x_1, x_2, y_1, y_2)$ del rango deseado.

**Implementación de la técnica *render-to-texture*:** como se estudió en el capítulo 4, resulta imprescindible disponer de un mecanismo que permita almacenar en forma de textura los resultados obtenidos tras la ejecución de un *shader*. La técnica *render-to-texture* permite que los resultados de una ejecución completa del *pipeline* gráfico estén disponibles, sin necesidad de pasar por la memoria del sistema, para siguientes pasos del algoritmo.

Existen diferentes métodos para llevar a cabo la implementación de la técnica *render-to-texture*. En el Capítulo 31 de [2] se propone la técnica *PBuffer*, del mismo modo que se implementa en *Brook*. Hasta poco tiempo atrás, sin embargo, era la única opción disponible para la implementación de este tipo de técnicas. *PBuffer*, además, presenta problemas de rendimiento, como se explica en [6], por lo que resulta conveniente el estudio de nuevas técnicas de implementación.

La extensión a OpenGL llamada *Framebuffer Object* (FBO) permite el uso de texturas como destinos finales de la renderización. Además, ofrece mejoras en el rendimiento con respecto a *PBuffer*, así como mayor flexibilidad.

Un FBO es una colección lógica de objetos (por ejemplo, de texturas), llamados *attachments*, que pueden servir como fuente o destino para la ejecución de un determinado *shader*. Existen varias alternativas a la hora de implementar la técnica *render-to-texture* utilizando FBO, dependiendo del número de FBO utilizados. Basada en la clasificación de Green ([6]) se propone una clasificación en función del rendimiento obtenido:

1. Utilización de un FBO distinto por cada textura a generar. Resulta una opción sencilla de implementar, aunque comparable en rendimiento con la técnica *PBuffer*.
2. Implementación de un FBO por *buffer*. Cada *buffer* dispondrá de un único FBO, aunque el número de *attachments* (texturas asociadas al buffer), podrá variar en función del tipo de *buffer* sobre el que se esté trabajando. Así, se definirán *buffers* de sólo lectura, con una textura asociada, o de lectura-escritura, en cuyo caso cada FBO tendrá dos texturas asociadas, una disponible para recibir los resultados y otra para proporcionar datos. De este modo, operaciones muy comunes del tipo $y = \alpha \cdot y$ podrán ser llevadas a cabo sobre un sólo *buffer* y un único FBO.

   Pese a la flexibilidad proporcionada, también existe una penalización en rendimiento por el uso de más de un FBO durante la ejecución del programa.
3. Utilización de un único FBO común, distribuyendo sus *attachments* o texturas asociadas entre los *buffers* existentes. Pese a ser la opción más rápida (ya que simplemente se utiliza un FBO, sin ningún cambio de contexto asociado al uso de varios FBO), presenta dos limitaciones principales:

- En primer lugar, todas las texturas asociadas a un mismo FBO deben presentar el mismo formato y dimensiones, lo que resta flexibilidad al sistema desarrollado.
- Por otra parte, el número de *attachments* asociados a un mismo FBO está limitado.

Valorando las ventajas e inconvenientes de cada implementación, se ha optado por el uso de un FBO distinto por cada *buffer* utilizado; pese a ser una opción más complicada a la hora de la implementación, la búsqueda del mayor rendimiento por parte de la GPU hace que no sea conveniente perder prestaciones en aspectos no relacionados con el cálculo por parte del procesador, como es el caso.

Así, se han definido dos tipos de *buffers*: *buffers* de sólo lectura o sólo escritura, con una única textura asociada sobre la cual leer o escribir, y *buffers* de lectura/escritura, con dos texturas asociadas, una de las cuales actuará como fuente de datos y otra como destino de la renderización. Además, se proporciona la rutina `AGPSwapBuffers`, que permite intercambiar la funcionalidad de cada una de las texturas que pertenecen a un determinado *buffer*.

**Etapa de rasterización:** Por último, la última etapa de todo programa basado en AGPlib consiste en la ejecución de las operaciones programadas en pasos anteriores. AGPlib proporciona la rutina `AGPRunShader`, que toma como argumentos el rango de ejecución, el *shader* a ejecutar y el *buffer* que recibirá los resultados de la ejecución. Internamente, el funcionamiento se basa en la creación de un *quad* (rectángulo), que cubra, tomando el rango definido por el usuario como los cuatro vértices que definen el *quad*, toda la matriz de entrada, convirtiéndola en un flujo de datos sobre los que se ejecutará el *shader* especificado.

## 6.3. Bibliotecas adicionales sobre GPUs: AGPBLAS y AGPImage

Uno de los objetivos principales del presente estudio es el de estudiar posibles algoritmos cuyas características permitan adaptarlos fácilmente y con buenos resultados a la ejecución sobre GPUs. Para ello, se ha optado por implementar bibliotecas de funciones que implementen operaciones sobre GPUs, haciendo uso de la biblioteca AGPlib anteriormente descrita. En las siguientes secciones se explica con detalle la implementación de algunas de estas funciones, más concretamente se describen rutinas de álgebra lineal (AGPBLAS) y rutinas para el tratamiento de imágenes (AGPImage).

### AGPBLAS

Las rutinas BLAS (Basic Linear Algebra Subprograms) son rutinas que implementan operaciones básicas sobre matrices y vectores. Debido a su eficiencia, portabilidad y disponibilidad, es muy común su uso en el desarrollo de *software* orientado al álgebra lineal, como por ejemplo LAPACK, siendo el rendimiento de algunas rutinas BLAS clave para el rendimiento final de este tipo de bibliotecas.

Es por ello que resulta interesante el estudio detallado del rendimiento que este tipo de operaciones pueden ofrecer al ejecutarse sobre todo tipo de arquitecturas, y más concretamente al ejecutarse sobre GPUs.

AGPBLAS es una implementación de algunas de las rutinas básicas de BLAS sobre GPU, cuya finalidad es estudiar su rendimiento sobre estas arquitecturas. A continuación se estudiarán en detalle las implementaciones de las operaciones de producto de matrices (rutina xGEMM en BLAS) y producto matriz-vector (rutina xGEMV en BLAS).

### Producto de matrices en GPU

El producto de matrices se presenta como un algoritmo con las características necesarias para su correcta adaptación a la ejecución sobre Unidades de Procesamiento Gráfico. Su patrón regular de accesos a memoria y el abundante paralelismo ofrecido por el algoritmo hacen suponer un buen rendimiento cuando su ejecución se realice sobre GPUs.

Las siguientes secciones muestran sucesivos refinamientos sobre un algoritmo de implementación simple, que sucesivamente irán adaptando las implementaciones a las características concretas de las GPUs, para extraer todo el rendimiento que éstas puedan llegar a ofrecer.

### Implementación simple

Se estudia en primer lugar una transformación directa del algoritmo básico para el producto de matrices en CPU, en base a las analogías entre ambos tipos de procesadores estudiadas en el capítulo 4. Por tanto, es posible partir del siguiente algoritmo, que, dadas dos matrices de entrada $A$ y $B$ de dimensión $N \times N$ calcula su producto $C = AB$:

Producto básico de matrices en CPU

```
for ( i=0; i<N; i++ )
  for ( j=0; j<N; j++ ){
    c[i,j] = 0;
    for ( k=0; k<N; k++ )
      c[i,j] += A[i,j] * B[i,j];

  }
```

Los pasos necesarios para, haciendo uso de AGPlib, transformar este algoritmo en un algoritmo apto para ser ejecutado sobre GPU serán:

1. Inicialización del entorno, utilizando el modo de almacenamiento de un valor por elemento de la textura[5].

2. Creación de tres *buffers*, de forma que puedan almacenar tanto los operandos de entrada como el resultado de la ejecución. El tamaño de las texturas vendrá definido por el tamaño de las matrices de entrada.

[5]Realmente, es posible utilizar texturas que almacenen elementos en sus cuatro canales (RGBA). De cualquier forma, y debido a que este algoritmo simple utilizará únicamente uno de ellos, resulta más sencillo el uso directo de texturas de este tipo.

3. Transferencia de los datos de cada matriz de entrada a sus correspondientes *buffers* en memoria de vídeo.

4. Definición del rangos de computación: puesto que deseamos que cada flujo de datos contenga todos los elementos de las matrices de entrada, no es necesario definir dominios de ejecución. Pese a todo, en caso de definirlos, éstos vendrían definidos por las coordenadas $(0, N, 0, N)$, de forma que se cubriese toda la matriz con el rectángulo definido por ellas. Esta definición conlleva, de forma implícita, la ejecución de los dos bucles externos del algoritmo básico de cálculo en CPU presentado anteriormente, ya que hará que el *kernel* de ejecución sea aplicado sobre cada uno de los elementos del flujo creado.

5. Carga del *shader*, asociación de parámetros y ejecución del proceso de computación. Dicho *shader* contendrá, como se estudió en el capítulo 4, el código ejecutado para cada iteración del bucle más interno del algoritmo original. En el caso del producto de matrices, el shader contendrá un código correspondiente al siguiente:

Código a ejecutar sobre cada elemento del flujo de entrada

```
1 c[i,j] = 0;
  for ( k=0; k<N; k++ )
3   c[i,j] += A[i,j] * B[i,j];
```

6. Transferencia de los resultados desde el *buffer* correspondiente a memoria central.

La rutina `sgemm` implementada sobre AGPlib se reescribiría como:

Producto simple de matrices en GPU (programa principal)

```
1 void sgemm( float * a, float * b, float * c, int n, int m ){

3   AGPBuffer A, B, C;
    AGPRange  range;
5   AGPDomain domain1, domain2;
    AGPShader shader;
7
    /* Paso 1: Inicializacion del entorno */
9   AGPInit( AGP_MODE_ONE );
    AGPSetEnvironment( argc, argv );
11
    /* Paso 2: Creacion de los buffers en GPU */
13  AGPCreateBuffer( &A, n, m, AGP_READ );
    AGPCreateBuffer( &B, n, m, AGP_READ );
15  AGPCreateBuffer( &C, n, m, AGP_WRITE );

17  /* Paso 3: Transferencia de datos */
    AGPTransferToBuffer( &A, a );
19  AGPTransferToBuffer( &B, b );

21  /* Paso 4: Definicion de rango */
    AGPSetRange( &range, 0, n, 0, m );
23
```

```
    /* Paso 5: Carga del shader y ejecucion */
25  AGPLoadShader( &shader, "sgemm.cg", "sgemm" );
    AGPBindFloatToShader( &shader, "n", n );
27  AGPBindBufferToShader( &shader, "a", &A );
    AGPBindBufferToShader( &shader, "b", &B );
29
    AGPRunShader( &shader, &range,  &C );
31
    /* Paso 6: Transferencia de resultados */
33  AGPTransferFromBuffer( &C, c );
}
```

El *shader* a ejecutar en GPU resultaría:

Producto simple de matrices en GPU (*shader*)

```
float sgemm( float2 coords: TEXCOORD0,
2            uniform samplerRECT a,
             uniform samplerRECT b,
4            uniform float n) :
              COLOR {
6
  float retval = 0.0;
8 float2 new_coordsa, new_coordsb;

10 for( i = 0.0; i < n; i++ ){
     new_coordsa = float2( i, coords.y );
12   new_coordsb = float2( coords.x. i );

14   retval += texRECT( a, new_coordsa ) * texRECT( b, new_coordsb );
   }
16
  return retval;
18
}
```

### Implementación optimizada para unidades vectoriales

Sin embargo, en el estudio de la arquitectura de la GPU realizado en el capítulo 5 se destacó la capacidad de trabajo vectorial de las unidades funcionales que la componen. En el ejemplo anteriormente desarrollado, sólo se trabaja sobre uno de los cuatro canales sobre los que puede operar cada unidad de coma flotante en un mismo ciclo de reloj.

La rutina que se desarrollará a continuación se basa en el uso de *buffers* que almacenan cuatro elementos en cada uno de los canales $(x, y, z, w)$[6] que los forman. Tanto las operaciones en el propio *shader* como las operaciones de preparación de los datos son más complejas que las que se daban en las implementaciones anteriores; por contra, se obtendrá un mayor rendimiento por parte de la GPU, al estar aprovechando en mayor grado su capacidad de cálculo vectorial.

[6] También denotados mediante *RGBA*.

El primer paso previo a llevar a cabo es la correcta reorganización de los datos en memoria con el fin de que la transformación de éstos en textura sea correcta. La razón de esta necesidad radica en el hecho de que la transferencia de datos entre memoria central y memoria de vídeo se realiza por filas: los cuatro primeros elementos de la primera fila pasarán a constituir los cuatro canales del primer elemento de la textura, y así sucesivamente. Sin embargo, es necesario para este tipo de algoritmos que los elementos se distribuyan en la textura manteniendo la ordenación original en sus elementos, tal y como muestra la figura 6.1. En la figura, se observan dos posibles transformaciones de una misma matriz sobre una textura en memoria de vídeo. La transformación mostrada en la parte superior de la imagen no realiza ningún tratamiento previo sobre la matriz original antes de ser mapeada como textura; como se puede observar, la distribución final de la misma en memoria de vídeo no corresponde con la distribución original de la matriz en memoria. En cambio, si se realiza una transformación previa sobre la matriz original, como se muestra en la parte inferior de la imagen, la textura resultante sí mantiene la misma distribución que la matriz original en memoria central.

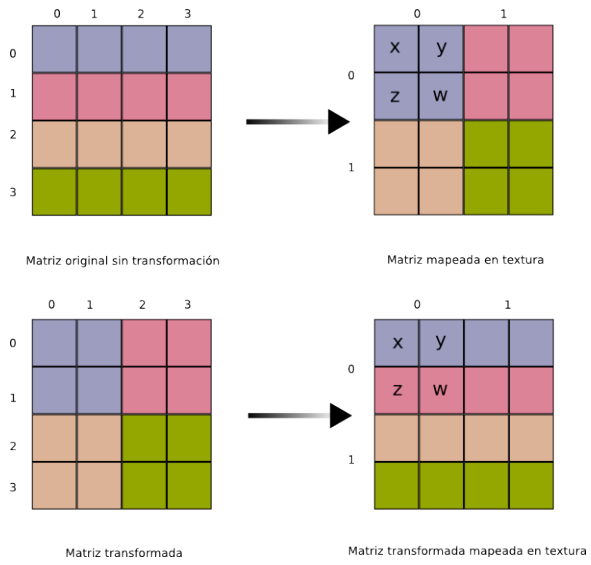



Figura 6.1: Dos posibles transformaciones matriz-textura: sin transformación previa (arriba) y con ella (abajo)

Este tipo de transformaciones resulta muy común como paso previo a algunas de las operaciones implementadas en AGPBLAS. Como es lógico, la transformación inversa es necesaria una vez la computación en la GPU ha finalizado y los datos han sido transferidos de vuelta a memoria central.

Con esta organización, se almacenan bloques de $2 \times 2$ elementos de la matriz original en cada *texel* formado por cuatro componentes. Esta distribución permite que cada *texel* transferido desde memoria de vídeo sea utilizado en dos ocasiones en cada ejecución del *shader* que se muestra a continuación:

*Shader* para la ejecución del producto de matrices en GPU

```
for( k=0; k<N/2; k++ ){
  C[i, j].xyzw += A[i,k].xxzz * B[k,j].xyxy +
                  A[i,k].yyww * B[k,j].zwzw;
}
```

Para facilitar la comprensión, se ha utilizado pseudocódigo para mostrar el anterior programa, y se han obviado detalles de implementación menos importantes. Sin embargo, es necesario destacar ciertos aspectos importantes:

- Las variables `i` y `j` son proporcionadas al *shader* de forma implícita (asociadas a cada fragmento con el que trabaja).
- Así como en los anteriores programas se iteraba sobre $N$ elementos (siendo $N$ la dimensión de la matriz), al utilizar ahora los cuatro canales disponibles en cada *texel*, la textura únicamente tiene $N/2 \times N/2$ elementos, y por tanto, únicamente se iterará sobre este número de elementos.
- Se introduce en este punto una nueva notación no existente en otros lenguajes, y que implementa una técnica muy común en la programación de *shaders*: el *swizzling*. Mediante esta técnica, es posible crear nuevos vectores reordenando los componentes de un vector original:

```
// Vector original
float4 vector1 = float4( 1.0, 3.0, 2.0, 42.0);
// Vector con elementos en orden inverso
float4 vector2 = vector1.wzyx;
// Vector con las componentes y y z del vector
// original como componentes x e y
float2 vector.xy = vector2.yz;
...
```

  Los sufijos utilizados, $(x, y, w, z)$ indican sobre qué componentes del vector se está trabajando. Además, esta notación permite definir operaciones sobre vectores de cuatro elementos, tal y como se ha implementado en el anterior *shader*. Así, por ejemplo, dada la siguiente expresión:

```
C[i,j].xyzw = A[i,k].xxzz * B[k,j].xyxy;
```

  Se realizarían las operaciones:

$$C(i,j).x = A(i,k).x * B(k,j).x$$
$$C(i,j).y = A(i,k).x * B(k,j).y$$
$$C(i,j).z = A(i,k).z * B(k,j).x$$
$$C(i,j).w = A(i,k).z * B(k,j).y$$

El *shader* presentado realiza la lectura de dos filas de la matriz $A$ y dos columnas de la matriz $B$ para calcular cuatro elementos de la matriz resultado. Realmente, cada

ejecución del *shader* realiza cuatro de las iteraciones del bucle interno del algoritmo original expuesto anteriormente para su ejecución sobre CPUs.

A continuación se muestra una traza de ejecución del algoritmo implementado sobre GPU, para matrices cuadradas de dimensión 4. Únicamente se muestran los cálculos que se llevarían a cabo para el fragmento con coordenadas asociadas $(0, 0)$ (correspondiente pues a los cuatro primeros elementos de la matriz resultado). Sin embargo, mientras esta ejecución se está llevando a cabo en uno de los procesadores de fragmentos de los que dispone la GPU, el resto de procesadores trabajarán sobre otros fragmentos de forma paralela, ejecutando cada uno de ellos las mismas instrucciones sobre cada dato de entrada.

- $k = 0$

  **Textura:** $C(0,0).x+ = A(0,0).x * B(0,0).x + A(0,0).y * B(0,0).z$

  **Matriz:** $C_{00}+ = A_{00} * B_{00} + A_{01} * B_{10}$

  **Textura:** $C(0,0).y+ = A(0,0).x * B(0,0).y + A(0,0).y * B(0,0).w$

  **Matriz:** $C_{01}+ = A_{00} * B_{01} + A_{01} * B_{11}$

  **Textura:** $C(0,0).z+ = A(0,0).z * B(0,0).x + A(0,0).w * B(0,0).z$

  **Matriz:** $C_{10}+ = A_{10} * B_{00} + A_{11} * B_{10}$

  **Textura:** $C(0,0).w+ = A(0,0).z * B(0,0).y + A(0,0).w * B(0,0).w$

  **Matriz:** $C_{11}+ = A_{10} * B_{01} + A_{11} * B_{11}$

- $k = 1$

  **Textura:** $C(0,0).x+ = A(0,1).x * B(1,0).x + A(0,1).y * B(1,0).z$

  **Matriz:** $C_{00}+ = A_{02} * B_{20} + A_{03} * B_{30}$

  **Textura:** $C(0,0).y+ = A(0,1).x * B(1,0).y + A(0,1).y * B(1,0).w$

  **Matriz:** $C_{01}+ = A_{02} * B_{21} + A_{03} * B_{31}$

  **Textura:** $C(0,0).z+ = A(0,1).z * B(1,0).x + A(0,1).w * B(1,0).z$

  **Matriz:** $C_{10}+ = A_{12} * B_{20} + A_{13} * B_{30}$

  **Textura:** $C(0,0).w+ = A(0,1).z * B(1,0).y + A(0,1).w * B(1,0).w$

  **Matriz:** $C_{11}+ = A_{12} * B_{21} + A_{13} * B_{31}$

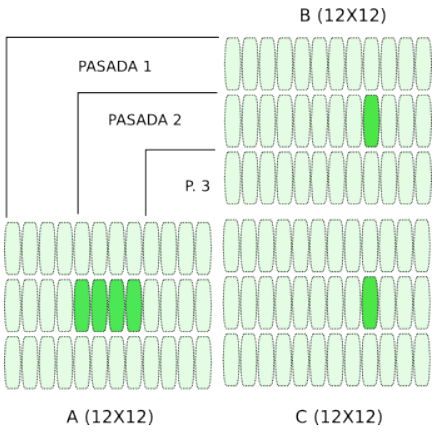



Figura 6.2: Representación esquemática del producto de matrices multipasada. Cada elemento de la figura corresponde con un texel, y almacena cuatro elementos consecutivos de cada columna de la matriz original.

### Implementación multipasada

Al igual que en los algoritmos desarrollados en CPU se suelen aplicar técnicas de programación por bloques para aprovechar el principio de localidad de referencias (y por tanto, la alta velocidad de las memorias cache), en GPU también se pueden aplicar técnicas similares.

Para conseguir unos resultados similares, el procedimiento consiste en transformar el producto de dos matrices en un conjunto de productos matriz-vector de dimensiones reducidas. Para ello, en primer lugar, deberemos agrupar en un mismo *texel* cuatro elementos consecutivos de una columna de la matriz original. Así, una matriz $M \times N$ pasará a mapearse en memoria de vídeo como una textura de dimensiones $M/4 \times N$, tal como indica la figura 6.2.

El procedimiento a partir de ese punto, consiste en realizar operaciones producto entre matrices de tamaño $4 \times 4$ y vectores $4 \times 1$. Un *shader*, expresado en pseudocódigo, que podría implementar estas operaciones tomaría la forma:

```
C[i,j].xyzw = accum[i,j].xyzw +
            + A[i,k].xyzw * B[k/4, j].xxxx +
            + A[i,k+1].xyzw * B[k/4, j].yyyy +
            + A[i,k+2].xyzw * B[k/4, j].zzzz +
            + A[i,k+3].xyzw * B[k/4, j].wwww +
```

En este caso, realizaríamos $k/4$ invocaciones de renderización sobre la matriz original; en cada una de ellas, $k$ se vería incrementada en 4 unidades. Los índices $i, j$ pasarían implícitamente al *shader* como coordenadas de textura, mientras que sería necesario el uso de una textura intermedia (llamada `accum` en el anterior *shader*), que almacenaría los resultados parciales necesarios en cada iteración.

Este método reutiliza cuatro veces por cada ejecución los datos asociados a cada *texel* leído perteneciente a la matriz $B$; sin embargo, únicamente utiliza una vez los

datos obtenidos de los *texels* pertenecientes a la matriz $A$. Para la realización de cuatro operaciones MAD, pues, son necesarias 6 consultas de datos, por lo que, en total, este algoritmo necesitará el doble de consulta de datos procedentes de texturas que el algoritmo de una sola pasada estudiado anteriormente.

Además, el coste adicional asociado a la lectura y escritura de datos en memoria de vídeo tras cada renderización es elevado, penalizando así también el rendimiento obtenido. Aún así, resulta interesante estudiar este tipo de algoritmos multipasada, para evaluar los pros y contras de su implementación, a la vista de los resultados obtenidos.

En este tipo de algoritmos, es posible aplicar otro tipo de técnicas comunes en computación general, como por ejemplo el desenrollado de bucles: es posible disminuir el número de ejecuciones en GPU aumentando la carga computacional que realiza cada una de ellas, siempre teniendo en cuenta la limitación en cuanto a número de instrucciones por *shader* que imponen la mayoría de GPUs actuales.

### Producto matriz-vector en GPU. Implementación de otras rutinas BLAS

Al igual que se ha conseguido implementar y encontrar distintos algoritmos para el producto de matrices, también se ha implementado el producto matriz-vector, con el fin de evaluar su rendimiento.

Al igual que en la anterior rutina, el estudio comienza con una implementación simple del producto matriz-vector en CPU, utilizando las analogías estudiadas en el capítulo 4. Así, una implementación para el producto de una matriz de dimensiones $N \times N$ por un vector $N \times 1$ sencilla seguiría los siguientes pasos principales:

1. Definición de tres *buffers* en memoria de vídeo, uno para cada uno de los operandos de la operación:
   - El primer *buffer* tendrá un tamaño en memoria de vídeo de $N \times N$ elementos (teniendo en cuenta que en esta primera implementación no utilizaremos la capacidad de cálculo vectorial de la GPU).
   - Los *buffers* relativos al segundo operando y al resultado tendrán $N \times 1$ elementos.
2. Definición del rango de ejecución; es importante observar como el rango de ejecución, en este caso, no coincide con los tamaños de todos los operandos de entrada: debido a que el flujo de datos a crear comprenderá únicamente los $N \times 1$ elementos del vector de salida, este será el rango que se deberá definir con ayuda de las funciones correspondientes de AGPlib.
3. Un posible *shader* a ejecutar para cada uno de los elementos del flujo tomaría la forma:

```
1 float sgemv( float2 coords: TEXCOORD0, uniform
              samplerRECT a,
3             uniform samplerRECT b,
              uniform float size) : COLOR{
5
  float i, nueva_i;
```

```
7  float2 coords_tmp_a, coords_tmp_b;

9  float tmp_retval = 0.0;

11 for( i = 0; i < size; i++ ){
     coords_tmp_a = float2( i, coords.y );
13   coords_tmp_b = float2( coords.x, i );

15   tmp_retval += texRECT(a, coords_tmp_a) * texRECT(b, coords_tmp_b);
   }
17 return tmp_retval;
}
```

El funcionamiento es sencillo: para cada uno de los elementos del flujo de datos, se determinará su valor final tomando cada elemento del vector (en este caso asociado a una textura) que actúa como segundo operando (`b` en el ejemplo), multiplicando su valor por cada uno de los elementos de la fila de la matriz de entrada (`a`) correspondiente a las coordenadas del elemento final. El resultado se almacenará el en elemento correspondiente de la textura resultado.

Por otra parte, se han implementado también algoritmos que hacen uso de las capacidades de cálculo vectorial de la GPU, al igual que en el caso del producto de matrices, con el fin de evaluar las mejoras en el rendimiento que esta técnica pueda suponer.

Las rutinas anteriormente descritas pertenecen a los niveles 2 (en el caso del producto matriz-vector) y 3 (para el producto de matrices) de BLAS. Los operandos de entrada de este tipo de rutinas son matrices y vectores (en el primer caso), o únicamente matrices (para BLAS-3). Existen otras rutinas, cuyos operandos de entrada son exclusivamente vectores; estas rutinas se engloban dentro del nivel 1 de BLAS.

Se han implementado dos ejemplos de rutinas de BLAS de nivel 1:

**`SAXPY`:** dados dos vectores de entrada, $x$ e $y$, y un escalar $\alpha$, la rutina realiza la siguiente operación:

$$y = \alpha x + y \tag{6.1}$$

**`SSCAL`:** dado un vector de entrada $x$, y un escalar $\alpha$, la rutina realiza la siguiente operación:

$$x = \alpha x \tag{6.2}$$

Aunque no se trata de rutinas numéricamente intensivas, la baja reutilización de cada elemento de entrada las convierte en candidatas a conseguir buenos resultados a la hora de ser implementadas sobre procesadores gráficos. Su implementación resulta sencilla: en el caso de `saxpy`, por ejemplo, basta con definir dos *buffers* (uno de ellos, el asociado al operando $y$, podría ser de lectura-escritura), y crear un flujo que abarque todos los datos del vector resultado. El *shader* tomaría la forma:

```
float4 saxpy (
    float2 coords : TEXCOORD0,
    uniform samplerRECT Y,
    uniform samplerRECT X,
    uniform float alpha ) : COLOR {

    float4 result;
    float4 y = texRECT(Y, coords);
    float4 x = texRECT(X, coords);
    result = y + alpha*x;

    return result;
}
```

Este *shader*, como se puede comprobar, hace uso de la capacidad de cálculo vectorial de las unidades funcionales de la GPU. En el caso de la rutina `sscal`, el programa a ejecutar sería todavía más sencillo, tomando cada elemento de la textura de entrada y realizando el producto con el parámetro $\alpha$ proporcionado.

### AGPImage

Otro de los campos de interés en el que se plantea el uso de GPUs con el objetivo de mejorar el rendimiento es el procesamiento de imágenes. En [1] se realiza un recorrido por las distintas disciplinas en las que la computación sobre GPUs ha sido estudiada, recibiendo especial interés el procesamiento de imágenes y señales. Se han llevado a cabo multitud de estudios sobre segmentación, registrado de imágenes, procesado de imagen médica, visión por computador, etc. (ver [1] para más información), siendo, en muchos casos, implementados con éxito sobre GPUs.

A modo de ejemplo, se explica a continuación el procedimiento seguido para la implementación de una rutina que realiza la operación de convolución entre un filtro y una imagen, asociada, como se verá más adelante, a una textura. Se explicará en primer lugar el procedimiento seguido para una implementación simple, y a continuación un refinamiento para explotar todas las capacidades de cálculo de la GPU.

Los dos métodos son muy similares en su implementación; para el primero de ellos, definiremos dos *buffers* en memoria de vídeo, de forma que uno de ellos (de dimensiones $N \times N$ para una imagen $N \times N$) contendrá la imagen a procesar, mientras que el segundo contendrá el filtro a aplicar a la imagen.

Tras la definición del flujo de datos (con la ayuda de las funciones de AGPlib, se define un rango que cubra toda la matriz (imagen) de entrada), sobre cada uno de los elementos de dicho flujo se aplicará el siguiente *shader*:

```
float conv( float2 coordsa: TEXCOORD0,
            uniform samplerRECT a,
            float2 coordsb:TEXCOORD1,
            uniform samplerRECT b,
            uniform float size) : COLOR{

  int i, j;
  float2 new_pos, new_pos_kernel;
```

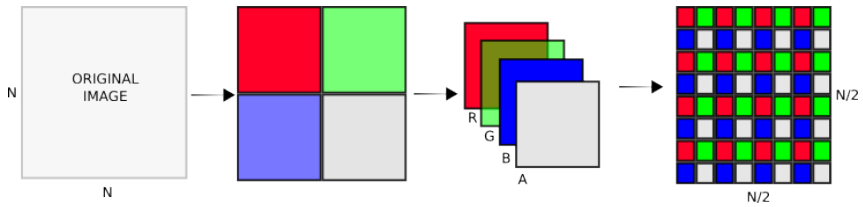



Figura 6.3: Transformaciones previas a la computación de la convolución. La imagen original de tamaño $N \times N$ se transforma en una textura de dimensiones $N/2 \times N/2$, almacenando los elementos correctos en cada canal de cada uno de sus elementos.

```
9   float tmp_retval = 0;

11  for( i=-size/2; i<=size/2; i++ ){
      for( j=-size/2; j<=size/2; j++ ){
13      new_pos.x = coordsa.x + j;
        new_pos.y = coordsa.y + i;
15      new_pos_kernel.x = size/2 - i;
        new_pos_kernel.y = size/2 - j;
17      tmp_retval += texRECT( a, new_pos ) *
                    * texRECT( b, new_pos_kernel );
19    }
    }
21
    return tmp_retval;
23  }
```

El anterior programa funcionará correctamente utilizando el modo de ejecución de AGPlib en el que cada *texel* almacena un sólo elemento de la imagen original. Sin embargo, este método no consigue extraer todo el rendimiento de posible de la GPU. El segundo método implementado requiere una redistribución inicial de los datos que forman la imagen, de forma que se pueda aprovechar la capacidad de las GPUs para almacenar cuatro elementos por cada uno de los *texels* cada textura.

La figura 6.3 muestra, esquemáticamente, la reordenación de los datos llevada a cabo antes del comienzo de la computación. El procedimiento consiste en la división de la imagen original en cuatro cuadrantes de igual dimensión, entrelazando a continuación cada uno de sus elementos sobre cada uno de los cuatro canales de cada de los *texel* que formará la textura sobre la que se trabajará.

Así, con un *shader* idéntico al anterior (con un valor de retorno de tipo `float4`, es decir, vector de cuatro enteros), se conseguirá estar trabajando de forma simultánea sobre cuatro elementos de la imagen de salida, y a la vez reducir el tamaño de la textura a procesar, que pasará de tener unas dimensiones $N \times N$ a otras más reducidas $N/2 \times N/2$.

## 6.4. Microbenchmarks para evaluación del rendimiento de GPUs

De forma paralela al desarrollo de las bibliotecas anteriormente descritas, se ha implementado también un conjunto de rutinas que permitirán construir *microbenchmarks*

para medir aspectos concretos del rendimiento de la GPU sobre la cual se ejecuten.

Aunque existen muchos parámetros a evaluar en que afectan al rendimiento final de los programas implementados sobre GPUs, se ha optado por medir los siguientes:

- Rendimiento pico del procesador.
- Ancho de banda en las comunicaciones procesador-cache.
- Ancho de banda en lectura/escritura secuencial de datos de textura.
- Rendimiento del bus AGP/PCIExpress.
- Sobrecoste de las operaciones condicionales en *shaders*.
- Sobrecoste debido a la preparación del entorno para la ejecución de *shaders*.

A continuación, se explica con mayor nivel de detalle la implementación de cada uno de los programas anteriores.

#### Rendimiento pico del procesador

Con el fin de evaluar el rendimiento máximo que puede ofrecer la GPU, resulta necesario conocer qué tipo de operaciones y en qué cantidad es capaz de ejecutar la GPU por ciclo de reloj.

El test ha sido desarrollado para GPUs de la serie 6 de Nvidia[7], cuyas características se estudiaron en la sección 5.2. Según el estudio, la tarjeta es capaz de llevar a cabo una operación MAD sobre un vector de cuatro elementos por ciclo de reloj.

Por tanto, medir el rendimiento máximo que puede ofrecer una GPU se reduce a conseguir crear un *shader* compuesto únicamente por instrucciones MAD sobre vectores de cuatro elementos. Es necesario tener en cuenta las posibles optimizaciones de código que el compilador de Cg suele aplicar, y que podrían hacer que el *shader* real programado en GPU no sea el deseado[8].

#### Rendimiento del bus de comunicaciones

Otro de los típicos cuellos de botella en las aplicaciones gráficas (y por tanto también en las aplicaciones de propósito general ejecutadas en GPU) se centra en el rendimiento del bus de comunicaciones. Por tanto, es conveniente conocer con exactitud qué prestaciones se pueden extraer del mismo.

La evaluación se ha llevado a cabo realizando sucesivas operaciones de escritura-lectura de texturas de tamaño relativamente elevado (limitado únicamente por la arquitectura), transfiriendo los datos entre memoria central y memoria de vídeo, y viceversa. Por tanto, en este caso no es necesaria la programación de ningún *shader* en GPU, ya que los datos no deben fluir a través del *pipeline* gráfico.

[7] Con la misma arquitectura, también es válido para GPUs de las series 5 y 7

[8] Puede servir de ayuda la capacidad de AGPlib de mostrar el código ensamblador del *shader* programado para asegurar su correcto funcionamiento.

### Rendimiento del sistema de memoria de la GPU

El objetivo de este test es la evaluación del rendimiento de la comunicación entre GPU y el nivel más alto de la jerarquía de memoria propia de los procesadores gráficos.

Para evaluar el rendimiento de la memoria cache, se ha creado un *shader* cuya función es la realización de lecturas sucesivas sobre un mismo elemento (idénticas coordenadas de textura) de una textura determinada. Con esto, se pretende conseguir acceder a datos recientemente leídos, y por tanto, residentes en memoria cache. A partir del número de datos leídos y del tiempo utilizado, es posible medir con exactitud el rendimiento del enlace procesador-cache.

En segundo lugar, se intenta analizar también el rendimiento de las comunicaciones entre memoria de vídeo y procesador gráfico, sin tener en cuenta el nivel intermedio de cache. Para ello, se implementa un *shader* que, de forma secuencial, realice lecturas de datos de todos los elementos de una textura de entrada, mapeándolos sobre la textura resultado. De este modo, se deja de aprovechar la localidad de referencias y se elimina, por tanto, el impacto de la memoria cache sobre el rendimiento final.

### Impacto de las instrucciones condicionales y coste adicional introducido por la invocación de la computación

Por último, se han desarrollado tests adicionales para observar el coste introducido por el uso de instrucciones de control de flujo y el sobrecoste que conlleva el proceso de invocación de la computación sobre el procesador gráfico.

Para el primero de ellos, se han desarrollado programas que realicen una función fija, utilizando estructuras de control (bucles y condicionales), y sin hacer uso de ellas, comparando los resultados obtenidos.

En el segundo caso, de modo similar al comentado en el caso del desenrollado de bucles para el producto de matrices multipasada, se ha reducido el número de invocaciones a un *shader* común aumentando la carga de trabajo del mismo, con el fin de observar cómo afecta este hecho al rendimiento final de la aplicación.

Capítulo 7

# Resultados experimentales

## Descripción de los experimentos

Los experimentos realizados se dividen en dos partes: en una primera parte se caracterizará el *hardware* gráfico utilizado, haciendo uso de los *microbenchmarks* anteriormente descritos. En una segunda parte, se estudiará el rendimiento de las rutinas utilizadas, haciendo uso de los resultados obtenidos durante la fase de caracterización del *hardware* para realizar una justificación adecuada de los mismos.

Todo el estudio se ha realizado sobre una GPU de la serie GeForce 6 de Nvidia, estudiada en detalle en el capítulo 5. Más concretamente, se trata del modelo GeForce 6200, cuyas principales características se resumen en la tabla 7.1.

| **Nvidia GeForce 6200** | |
|---|---|
| Código del chip | NV44a |
| Año de fabricación | 2004 |
| Velocidad del procesador | 350 Mhz |
| Cantidad de memoria | 128 Mb |
| Velocidad de memoria | 533 Mhz |
| Número de procesadores de vértices | 3 |
| Número de procesadores de fragmentos | 4 |
| Anchura del bus | 64 bits |
| Bus de conexión | AGP 4x |

Tabla 7.1: Características básicas de la GPU utilizada para llevar a cabo los experimentos

Realmente, no se trata de un procesador gráfico de última generación; sin embargo, el propósito del estudio también se centra en observar como *hardware* específico y de bajo coste es capaz de ofrecer unos rendimientos considerables, en comparación con *hardware* de propósito general, de mayor coste, y de una generación similar.

Por ello, para realizar las comparativas entre algoritmos similares implementados en GPU e implementados en CPU, se ha escogido un procesador de aproximadamente la misma generación que la GPU elegida; se trata de un AMD Athlon XP 2400+, fun-

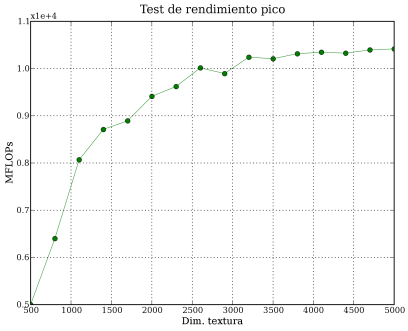



Figura 7.1: Resultados experimentales para el test de rendimiento máximo de la GPU sobre la que se realizarán el resto de experimentos

cionando a una velocidad de reloj de 2 Ghz (nótese la diferencia de velocidad de reloj entre este procesador y el procesador gráfico anteriormente descrito), $64Kb$ de cache L1 de datos (la misma cantidad para cache de instrucciones), $256Kb$ de cache L2 y *Front Side Bus* a 133 Mhz. En cuanto al *software*, resulta interesante la comparación entre rutinas implementadas en GPU y rutinas optimizadas para CPU; por ello, se ha escogido la implementación de BLAS realizada por Kazushige Goto[1]; GotoBLAS es a día de hoy la implementación más rápida de la biblioteca BLAS para una gran cantidad de plataformas. Existen estudios comparativos con bibliotecas menos optimizadas, como [3] y [7], pero ninguno realiza una comparación real entre bibliotecas realmente optimizadas, como GotoBLAS y las correspondientes implementaciones en GPU.

Estudiaremos a continuación cual es el rendimiento máximo que puede ofrecer la GPU sobre la que se realizará la evaluación de las rutinas. Para ello, haremos uso del *microbenchmark* específico descrito en la sección anterior; la figura 7.1 muestra los resultados obtenidos para la ejecución del test con tamaños de flujo de entre 500 y 5000 elementos. Se ha obtenido un rendimiento máximo de 10416,66 MFlops/s (millones de operaciones en coma flotante por segundo) para el mayor tamaño de flujo de datos con el que se ha experimentado.

El rendimiento teórico máximo que la GPU podría ofrecer, a partir las características presentadas por el fabricante, y ante un programa compuesto exclusivamente por instrucciones MAD, sería:

- Cada instrucción MAD sobre operandos vectoriales de 4 elementos supone realmente pues 8 operaciones en coma flotante o FLOPS.
- Dado que la GPU posee cuatro procesadores de fragmentos programables, es capaz de realizar 32 operaciones en coma flotante por segundo en total.

[1]Disponible en http://www.tacc.utexas.edu/resources/software/#blas

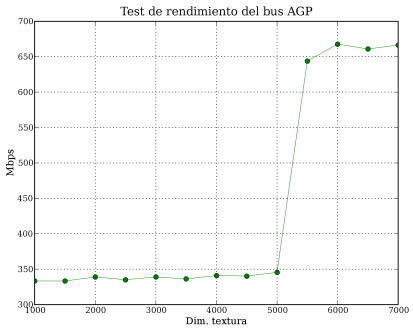



Figura 7.2: Resultados obtenidos para la evaluación del rendimiento del bus de comunicaciones entre el sistema y la GPU

- A partir de su frecuencia de reloj ($350Mhz$), es posible calcular un máximo de 11200 Mflops para este procesador.

Como se puede apreciar, los resultados teóricos y experimentales son muy similares; la pequeña diferencia (de alrededor de un 7 %) puede deberse al sobrecoste introducido por el *driver* de la propia tarjeta gráfica o las transferencias entre memoria central y memoria de vídeo.

De hecho, es posible también estudiar el rendimiento del bus AGP de comunicaciones entre la GPU y el resto del sistema. Para ello, se evaluará el test implementado con tal fin y descrito en la anterior sección, obteniéndose los resultados que se muestran en la figura 7.2.

Realmente, los resultados obtenidos son muy inferiores a los que teóricamente puede llegar a ofrecer el bus AGP; los flujos utilizados no tienen el suficiente tamaño como para explotar al máximo su capacidad, aunque con flujos suficientemente grandes (a partir de texturas con 5500 elementos de tipo flotante, aproximadamente), el salto en las prestaciones es evidente. La limitación en el tamaño de los *buffers* que pueden ser creados en GPU, sin embargo, hace que la reducida velocidad obtenida a través del bus pueda convertirse en un problema, al menos en este tipo de buses en los que la velocidad no es tan elevada como en los buses PCIExpress de última generación. Por tanto, aunque el bus de comunicaciones aumente su rendimiento, como lo hacen las últimas generaciones de PCIExpress, en este tipo de computación, con estructuras de datos de tamaño relativamente reducido, nunca llegarán a alcanzar el máximo de sus posibilidades en cuanto a velocidad de transferencia de datos.

Otro de los aspectos que resultarán útiles a la hora de justificar el rendimiento del sistema es la velocidad ofrecida en el acceso a memoria cache del mismo; también se evaluará el rendimiento de acceso secuencial a memoria, tal y como se describió en su momento. Los resultados obtenidos se muestran en la figura 7.3; resulta evidente como

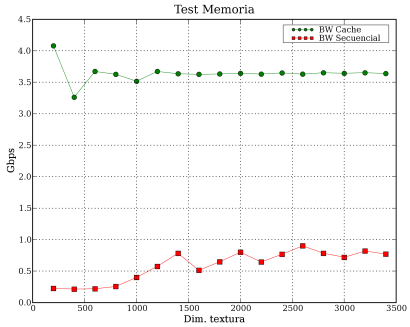



Figura 7.3: Rendimiento del sistema ante accesos a memoria cache y accesos secuenciales a datos de textura

el acceso a un mismo *texel* de forma repetitiva, haciendo por tanto uso de la memoria cache, mejora en gran medida el rendimiento con respecto al acceso secuencial a datos de textura, donde las caches juegan un papel mucho menos importante.

Los resultados obtenidos, sin embargo, son muy inferiores a los obtenidos para procesadores de similar generación pero características superiores, tanto a nivel de rendimiento pico, como a nivel de acceso al sistema de memoria de vídeo. Como ejemplo, una GPU representativa de la serie GeForce 6, más concretamente el modelo 6800 Ultra, ofrece un rendimiento pico de 43,9 GFlops (cuatro veces superior a la GPU estudiada), 18,3 GBytes/s en acceso a memoria cache y $3,8$ Gbytes/s en acceso secuencial (datos extraídos de [3]). Sin embargo, se trata de una GPU funcionando a 400 Mhz, sobre un bus PCIExpress, ancho de palabra de 256 bits, reloj de memoria de 110 0 Mhz y, por encima de todo, con 16 procesadores de fragmentos (y de vértices) programables, lo que le confiere una potencia de cálculo aproximada cuatro veces superior al procesador sobre el que se realizarán los experimentos.

## AGPBLAS

### Rendimiento del producto de matrices

Se estudia a continuación el rendimiento obtenido para cada uno de los algoritmos que implementan el producto de matrices estudiado en secciones anteriores. La comparación se realizará con respecto a la rutina `sgemm` de BLAS, que realiza el producto de matrices de datos flotantes en simple precisión; el hecho de que las GPUs trabajen con precisión simple de 32 bits hace que se haya descartado el uso de la rutina `dgemm`, que realiza también un producto de matrices, pero sobre operandos representados en coma flotante de doble precisión.

Se parte de tres algoritmos diferentes para su implementación en GPU:

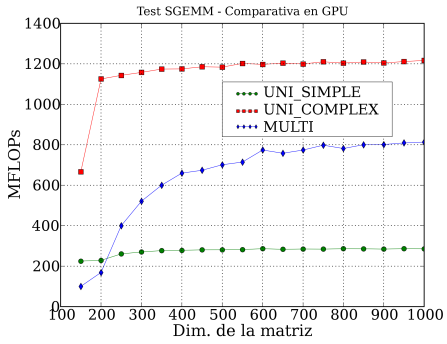



Figura 7.4: Comparación de los distintos algoritmos utilizados para la evaluación del producto de matrices en GPU

- Algoritmo de una sola pasada, sin aprovechar las capacidades vectoriales de las unidades funcionales del procesador (algoritmo `UNI_SIMPLE`).
- Algoritmo de una sola pasada, con aprovechamiento de las capacidades de cálculo vectorial del procesador (algoritmo `UNI_COMPLEX`).
- Algoritmo multipasada, con cálculo sobre datos de tipo vectorial (algoritmo `MULTI`).

La gráfica de la figura 7.4 muestra una comparativa del rendimiento obtenido, en MFLOPs, para cada uno de los tres algoritmos anteriores. A la vista de la figura, resulta destacable el aumento de rendimiento obtenido al aprovechar las capacidades de cálculo sobre vectores del procesador de fragmentos (implementado en el algoritmo `UNI_COMPLEX`) frente al rendimiento que se obtiene tras la ejecución del algoritmo `UNI_SIMPLE`. De hecho, se puede observar como el rendimiento obtenido por el primero es, aproximadamente, cuatro veces superior al obtenido por el segundo; este tipo de comportamiento es muy común en los algoritmos implementados sobre GPUs. Además de la aceleración producida por el hecho de aprovechar la capacidad de cálculo vectorial del procesador, las texturas utilizadas son de dimensión menor a las matrices originales en memoria, lo que conlleva:

1. Un menor número de iteraciones en el bucle programado en el *shader*; así, la penalización producida por instrucciones de control de flujo se ve reducida.
2. Un menor número de operaciones de consulta (*lookup*) de datos de textura, con la consiguiente reducción en el tiempo de espera por parte del procesador.

Estos tres hechos explican el mejor rendimiento el algoritmo `UNI_COMPLEX` sobre el algoritmo sin capacidad vectorial.

En cambio, resulta destacable el escaso rendimiento que se consigue al implementar un algoritmo multipasada para el producto de matrices. El objetivo final era el de

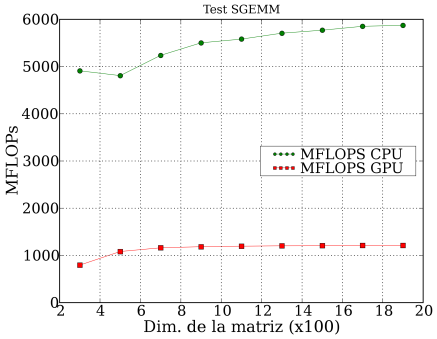



Figura 7.5: Comparativa entre el producto de matrices en GPU y en CPU (GotoBLAS)

aumentar la cantidad de aciertos de caché producidos, con el sobrecoste adicional que supone la escritura en memoria de los resultados intermedios tras cada pasada del algoritmo. Sin embargo, esta penalización resulta, para esta GPU, demasiado alta para que el mayor número de aciertos de cache la contrarreste. De hecho, el resultado obtenido, a la vista de la figura 7.4, resulta aproximadamente la mitad que el obtenido con el mejor de los algoritmos implementados.

Sin duda, el escaso rendimiento que la GPU estudiada obtiene a nivel de acceso a memoria (véase figura 7.3), en comparación con otros procesadores de gama ligeramente superior (como se puede comprobar en [3]), hace que este tipo de algoritmos sean convenientes únicamente para procesadores gráficos acompañados de un subsistema de memoria de elevado rendimiento. Es posible, adicionalmente, aplicar técnicas comunes en computación de altas prestaciones, como el desenrollado de bucles, a este tipo de algoritmos multipasada, de forma que aumente la cantidad de trabajo llevada a cabo en cada ejecución del *shader*; por tanto, el bucle a desenrollar sería aquel que realiza la invocación de la renderización, con el consiguiente aumento en el número de instrucciones que formarían el *shader* a ejecutar. En el caso de las GPUs, es posible conseguir, de este modo, una reducción en el sobrecoste asociado a la invocación de cada proceso de renderizado, ya que su número disminuye de forma proporcional al grado de desenrollado del bucle asociado.

Tomando pues como referencia el mejor de los algoritmos implementados para GPU, es decir, el algoritmo de una sola pasada con aprovechamiento de las capacidades de cálculo vectorial de la GPU, la figura 7.5 muestra una comparativa con respecto al rendimiento obtenido, en ejecución sobre CPU, de la rutina `SGEMM` implementada por GotoBLAS.

El rendimiento obtenido tras la ejecución en GPU no es comparable con el obtenido para una implementación totalmente optimizada sobre CPU. La gran diferencia de prestaciones hace pensar que el algoritmo elegido para su implementación en GPU podría no ser óptimo desde el punto de vista de los accesos a memoria. Para comprobar la veracidad de esta afirmación, se ha implementado un experimento consistente en la

ejecución de un *shader* similar al utilizado para el producto de matrices, aunque sin utilización de operaciones matemáticas (únicamente con operaciones de consulta de datos desde texturas). Con este tipo de *shaders*, es posible, a partir del número de bytes leídos desde memoria de vídeo, obtener un valor para el ancho de banda total utilizado por el programa. En este caso, este valor se sitúa alrededor de 3,04 Gbps. Este valor, a la vista de los resultados pico obtenidos experimentalmente y mostrados en la gráfica 7.3, supone un 83 % del ancho de banda máximo que puede extraerse, y demuestra que, aun no siendo óptimo, el patrón de accesos a memoria resulta cercano al óptimo.

Por tanto, los resultados indican que el procesador toma datos desde el sistema de memoria, para el caso del producto de matrices, a una tasa únicamente un 17 % menor de la máxima tasa que es posible obtener. Sin embargo, los resultados finales a nivel de potencia de cálculo, comparando el rendimiento obtenido con el rendimiento máximo que experimentalmente se obtuvo (ver figura 7.1), muestran una eficiencia a nivel de potencia de cálculo de sólo el 11.6 %[2] elementos.

Así, y pese a que se está suministrando datos a la GPU a una tasa realmente elevada, cercana a la tasa máxima de transferencia entre memoria y procesador, no es posible, con el *shader* programado, conseguir que el procesador se mantenga ocupado una mayor parte del tiempo. De hecho, el rendimiento del sistema de caches de las GPUs resulta uno de los mayores impedimentos a la hora de conseguir buenos resultados de rendimiento en programas ejecutados sobre GPU.

#### Rendimiento del producto matriz-vector y otras operaciones de álgebra lineal

De modo más breve, se estudia a continuación el rendimiento de las rutinas correspondientes al producto matriz-vector y otras operaciones de álgebra lineal.

La figura 7.6 muestra el rendimiento máximo obtenido por la implementación en GPU del producto matriz-vector; se realiza la comparación con el rendimiento obtenido con la rutina `sgemv` de la biblioteca BLAS de Goto. Como se puede observar, los resultados siguen favoreciendo a la implementación en CPU; sin embargo, comparándolos con los obtenidos para el producto de matrices (ver figura 7.5),los resultados para GPU obtenidos en este caso están mucho más cerca de igualar los resultados para CPU de lo que lo estaban para la rutina `sgemm`.

Así, parece que operaciones con menor reutilización de los datos, como es el caso de la rutina de producto matriz-vector, se comportan mejor al ser implementadas en GPU que rutinas con alto grado de reutilización de los datos, como es el caso del producto de matrices, donde cada uno de los datos es utilizado $O(n)$ veces durante el cálculo. La reutilización de datos, realmente, resulta incompatible con la orientación a flujos de los algoritmos.

Por tanto, una primera característica deseable para que un algoritmo se adapte bien al paradigma de programación orientado a flujos radica en el hecho de que presente una baja reutilización de los datos de entrada. De hecho, existen rutinas BLAS sin ningún tipo de reutilización de los datos de entrada. Veremos los resultados para dos de ellas: producto de vectores (rutina `saxpy`) y escalado de vectores (rutina `sscal`). Se trata de

[2]Tomando como rendimiento máximo de la GPU 10416 MFLOPs y como mejor rendimiento para el producto de matrices, 1211 MFLOPs, obtenidos para un tamaño de matriz de $1900 \times 1900$

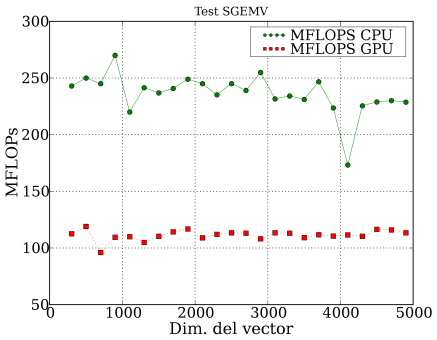



Figura 7.6: Resultados obtenidos para la rutina `sgemv`, tanto en GPU como en CPU (GotoBLAS)

rutinas totalmente orientadas a flujo, sin ningún tipo de reutilización de los datos de entrada durante el proceso de cálculo.

La figura 7.7 muestra los resultados obtenidos en comparación con los resultados que se obtienen con las bibliotecas optimizadas sobre CPU; los resultados obtenidos muestran como, de nuevo, esta última implementación obtiene mucho mejor rendimiento que la ejecutada sobre el procesador gráfico.

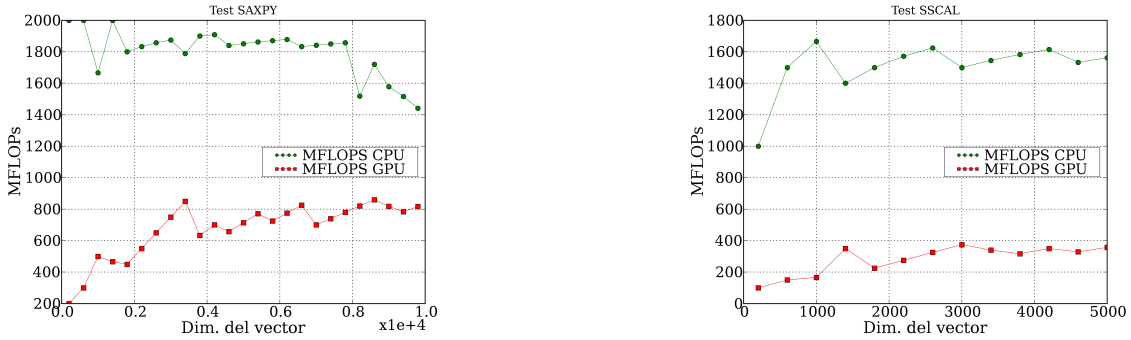
El problema en este caso se centra en la escasa carga computacional que implican las operaciones llevadas a cabo sobre cada elemento del flujo de entrada. De hecho, los resultados obtenidos son cuantitativamente mejores para la rutina `saxpy`, con mayor carga computacional que la rutina `sscal`, cuya única función es la de realizar un producto entre dos escalares para cada elemento del flujo; la rutina `saxpy` realiza una operación más de adición para cada elemento.

Por tanto, pese a ser algoritmos totalmente adaptables a la programación orientada a flujos, sin ningún tipo de reutilización de los datos de entrada, la baja carga computacional que conllevan las operaciones llevadas a cabo para cada elemento del flujo hace que no extraigan de la GPU todo el poder computacional que ésta puede llegar a ofrecer.

## AGPImage. Filtros de convolución

A la vista de los anteriores resultados, es necesario encontrar algoritmos que cumplan ciertas características a la hora de ser adaptados correctamente a la programación orientada a flujos de datos típica de las GPUs. Básicamente, los siguientes son requisitos deseables:

- Patrón de acceso regular a memoria.
- Baja (o nula) reutilización de los datos de entrada.




(a) (b)

Figura 7.7: Resultados experimentales obtenidos para las rutinas `saxpy` (a) y `sscal` (b). Comparación con las implementaciones en CPU proporcionadas por GotoBLAS

- Alta carga computacional por cada elemento del flujo de entrada.
- Por supuesto, alto grado de paralelismo de datos.

Un algoritmo que cumple las anteriores premisas, y por tanto, se presenta como buen candidato para una correcta adaptación a la programación orientada a flujos es la aplicación de filtros de convolución sobre imágenes (o señales). En principio, se trata de un algoritmo que reúne todas las características anteriormente descritas:

- Presenta tanto un patrón de acceso a memoria regular como un alto grado de paralelismo de datos, lo que permitirá explotar los bajos recursos disponibles a nivel de memoria cache y la replicación de unidades funcionales (procesadores de fragmentos) en la GPU, respectivamente.
- Existe reutilización de los datos, aunque ésta es muy baja.
- La carga computacional por cada unidad del flujo es relativamente elevada (y basada en operaciones de producto y acumulación (MAD), para las cuales las GPUs están muy bien adaptadas, como se vio en el capítulo 5).

La figura 7.8 muestra los resultados obtenidos para cada uno de los dos algoritmos implementados en GPU para el proceso de convolución, así como una comparativa con respecto a los resultados obtenidos tras la ejecución de la misma operación sobre CPU, para una imagen de $512 \times 512$ elementos, variando la dimensión del filtro de convolución entre 3 y 15 elementos (se aplican filtros media).

Como se puede observar, en este caso los resultados obtenidos en el caso de ejecutar el programa sobre GPU sí mejoran de forma significativa los conseguidos tras la ejecución en CPU. De hecho, ya el algoritmo simple (sin almacenamiento en los cuatro canales

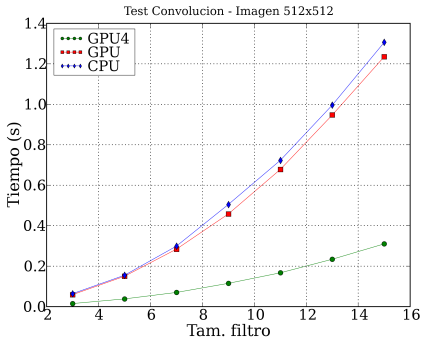



Figura 7.8: Resultados obtenidos para la operación de convolución, con tamaños de filtro variables, para una imagen de 512x512 píxeles

de cada *texel*) estudiado consigue mejorar el tiempo de ejecución del algoritmo básico implementado en CPU.

También cabe destacar la mejora que consigue el refinamiento introducido sobre el primer algoritmo ejecutado en GPU, logrando tiempos de ejecución que mejoran en cuatro veces, aproximadamente, los tiempos de ejecución del primer algoritmo implementado.

Aunque también existe reutilización de cada dato leído desde memoria de texturas, parece, a la vista de los resultados obtenidos, que para los tamaños de filtro estudiados los datos tanto del propio filtro como de las proximidades del punto a actualizar sí pueden ser almacenados en memoria cache, y por tanto no existe penalización en los resultados obtenidos. Además, pese a que las ni las especificaciones ni el funcionamiento concreto del sistema de memoria de las GPUs son publicados por sus fabricantes, existen estudios ([5]) que concluyen (al igual que parece extraerse de los resultados obtenidos en nuestro estudio), con que la arquitectura de memoria cache en las GPUs está optimizada para explotar la localidad de referencias en 2D, en lugar del carácter unidimensional de las memorias cache habituales.

Los resultados obtenidos confirman las conclusiones extraídas a partir de los resultados para operaciones de álgebra lineal; no todos los algoritmos se adaptan de forma correcta a la operación en GPUs, por lo que no se trata de un *hardware* útil para cualquier tipo de rutina desde el punto de vista del rendimiento ofrecido. Es necesario pues, estudiar con detenimiento las características del algoritmo a implementar, y valorar si realmente es viable la adaptación del mismo para su ejecución sobre el procesador gráfico, escogiendo en su caso la organización de datos en memoria, operaciones a llevar a cabo y sobre qué partes concretas del algoritmo es conveniente realizar la computación sobre el procesador gráfico.

Capítulo 8

# Conclusiones y trabajo futuro

La realización del presente trabajo ha servido como introducción a la computación general sobre GPUs, tanto desde el punto de vista del estudio de las arquitecturas típicas de los procesadores gráficos actuales, como desde la perspectiva del desarrollo de algoritmos sobre este tipo de plataformas.

El planteamiento inicial del trabajo contemplaba una fase inicial de estudio tanto del modo de funcionamiento de los procesadores gráficos, con sus diferencias más importantes con respecto a los procesadores de carácter general, como de las arquitecturas más extendidas (para lo cual se realizó un estudio de la serie Geforce 6 de Nvidia), como de las arquitecturas de última generación, enfocadas claramente a abrir las puertas de este tipo de procesadores a la computación no orientada a gráficos. Uno de los problemas principales que supone el estudio de este tipo de arquitecturas es el carácter cerrado de las mismas: pese a que sus fabricantes ofrecen nociones generales sobre su funcionamiento interno, es complicado (incluso en ocasiones imposible), acceder con mayor nivel de detalle a las especificaciones de sus productos.

Una vez asimilados los anteriores conceptos, el objetivo ha sido el de evaluar qué tipo de algoritmos se adaptan de forma correcta desde el punto de vista del rendimiento a la ejecución sobre procesadores gráficos.

Realmente, una de las mayores dificultades que plantea la programación de GPUs, al menos hasta su última generación, se centra en el propio proceso de desarrollo de *software*. La utilización de APIs diseñados exclusivamente para desarrollo de aplicaciones gráficas, y en ningún caso para otro tipo de algoritmos, hace que el desarrollo sea lento y a menudo complicado. La aparición de conceptos como CUDA, que ofrece al programador la posibilidad de obviar el hecho de estar desarrollando algoritmos sobre procesadores gráficos, con extensiones para lenguajes de propósito general, abre las puertas al desarrollo rápido de aplicaciones, con menor (o nula) necesidad de conocimientos sobre APIs gráficas, lo que supone el desarrollo de programas de forma más rápida, y también más eficiente.

Con el fin de facilitar el desarrollo de *software* durante el transcurso del proyecto, se ha desarrollado una biblioteca de funciones, AGPlib, que permite ocultar los aspectos orientados a gráficos propios de este tipo de aplicaciones. Pese a que existen interfaces desarrolladas que permiten realizar tareas similares, resulta muy útil como método de

introducción y familiarización con este tipo de programación, el desarrollo de una biblioteca de este tipo, con dos razones principales:

- En primer lugar, ha servido como método de estudio de cada una de las fases que conforman una aplicación ejecutada sobre GPU. Detalles que, de no haber sido desarrollado una interfaz de este tipo, hubieran sido pasados por alto, han sido estudiados e implementados en su totalidad.

- Además, y debido a que uno de los objetivos principales del proyecto era el de extraer el máximo de rendimiento de los procesadores gráficos, ha sido posible adaptar la interfaz (por ejemplo, mediante la introducción de la extensión FBO de OpenGL, no implementada en otro tipo de productos similares) para ofrecer el máximo rendimiento posible.

Además del estudio del funcionamiento de los procesadores gráficos y el desarrollo de AGPlib, el proceso de evaluación de distintas rutinas ha permitido caracterizar los algoritmos que mejor se adaptan al modelo de ejecución que se da en GPUs. A nivel de arquitectura, el modelo SIMD y el bajo rendimiento de las caches (en la generación de GPUs sobre la que se han ejecutado las pruebas), limitan en gran medida el número de algoritmos potencialmente idóneos para ser ejecutados sobre procesadores gráficos. Algoritmos con patrones de acceso regular a memoria, baja reutilización de los datos de entrada (es decir, fácilmente adaptables a la computación orientada a flujos), elevado paralelismo de datos, y con alta carga computacional por cada uno de los elementos sobre los que operan, se presentan como firmes candidatos a conseguir buen rendimiento al ser ejecutados en este tipo de plataformas.

La aparición de una nueva generación de procesadores (de los que se ha estudiado la serie G80 de Nvidia como ejemplo representativo), con una nueva arquitectura unificada, sistemas de memoria optimizados (sobre todo a nivel de cache), frecuencias de reloj muy superiores a las que se dan en generaciones anteriores y buses de comunicaciones de alto rendimiento, junto con las nuevas interfaces de programación no orientadas a aplicaciones gráficas, hacen de este tipo de arquitecturas sistemas muy interesantes desde el punto de vista de la computación de altas prestaciones.

De hecho, las tendencias actuales apuntan no sólo a la aparición de *hardware* gráfico orientado exclusivamente a computación general, sino a la integración de la GPU como un coprocesador más, sirviendo de soporte a la CPU para cierto tipo de aplicaciones, en las que el aumento de prestaciones es elevado.

Por tanto, resultará interesante realizar un estudio similar al presentado en esta memoria, sobre productos de última generación, tanto a nivel de *software* como de *hardware*, observando en qué medida pueden llegar a mejorar los resultados aquí obtenidos.

Desde el punto de vista de la comunidad científica, la programación sobre procesadores gráficos ha creado en los últimos tiempos una gran expectación. La posibilidad de conseguir rendimientos muy elevados, a cambio de un bajo coste de adquisición, hace que se esté estudiando el modo de adaptar multitud de algoritmos conocidos a las características concretas que requiere la computación sobre GPUs. Disciplinas como la minería de datos, tratamiento de imágenes y señales, imágen médica, álgebra lineal, simulaciones físicas o tratamiento de consultas sobre bases de datos son sólo algunas

de las ramas que han mostrado gran interés en este tipo de computación, obteniendo además resultados positivos.

Los procesadores gráficos correspondientes a las últimas generaciones desarrolladas se han convertido en el primer sistema realmente extendido de computación paralela a nivel de datos (junto con el procesador Cell, desarrollado por IBM, Sony y Toshiba, curiosamente muy orientado también a la programación sobre flujos de datos). Su rápido proceso de desarrollo en comparación con los procesadores de carácter general, y su relativamente bajo coste, hacen de ellos una opción muy interesante desde el punto de vista del rendimiento, así como también un campo de estudio relativamente joven y con perspectivas de futuro.

Sin embargo, es necesario también reconocer el hecho de que este crecimiento tendrá un límite, en lo que respecta a su adaptación a la computación general: pese a que cada vez el rendimiento ofrecido por este tipo de procesadores aumenta, conviene recordar que se está hablando de procesadores de carácter específico, totalmente ligado al procesamiento de gráficos de alto rendimiento, desarrollados por empresas y utilizado por usuarios que no desean sacrificar en ningún caso potencia de cálculo en aplicaciones gráficas para favorecer a otro tipo de aplicaciones. Por tanto, el diseño de las próximas generaciones de procesadores gráficos supondrá un reto a la hora de combinar elevado rendimiento gráfico con la suficiente generalidad a nivel de cálculo como para hacer factible la ejecución de aplicaciones de carácter general sobre ellos, y un campo de estudio todavía mayor para extraer de ellos todo el rendimiento para este tipo de aplicaciones.

# Bibliografía


[1] John D. Owens et al. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 2007.

[2] Randima Fernando et al. *GPUGems 2. Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[3] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. *Graphics Hardware*, 2004.

[4] Randima Fernando. *The Cg Tutorial. The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[5] Todd Gamblin. Caching Architectures and Graphics Processing. 2006.

[6] Simon Green. The Opengl Framebuffer Object Extension. *Game Developers Conference (GDC) 05, San Francisco*, 2005.

[7] E. Scott Larsen and David McAllister. Fast Matrix Multiplies using Graphics Hardware. *SC2001*, 2001.

[8] NVIDIA. NVIDIA Geforce 8800 GPU Architecture Overview. Technical report, NVIDIA, Corp., 2006.

[9] NVIDIA. Nvidia CUDA Compute Unified Device Architecture. Technical report, NVIDIA, Corp., 2007.

# Índice alfabético