



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

GPGPU (General Purpose on Graphical Processor Units)

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



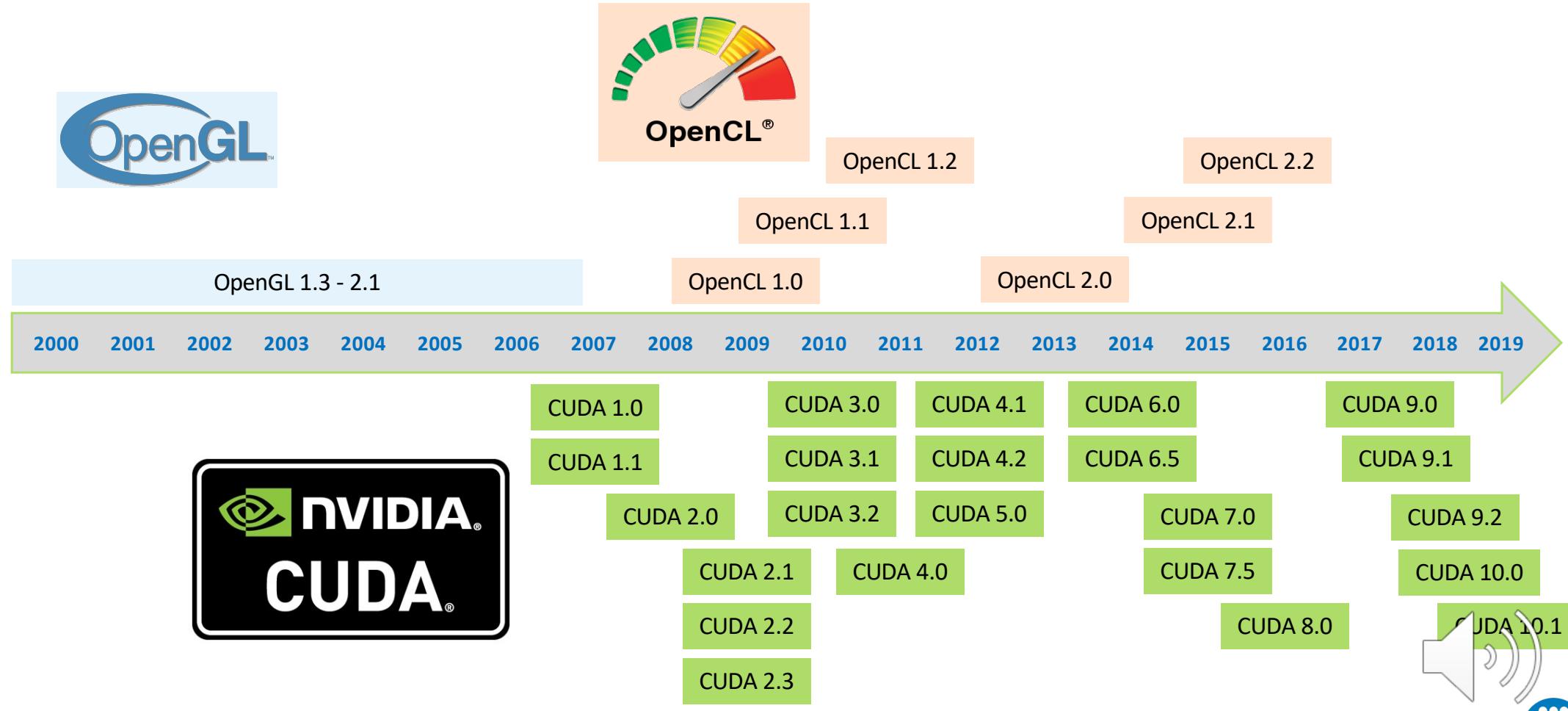
Antecedentes

- El objetivo de GPGPU es el uso de hardware gráfico (GPUs) para ejecutar aplicaciones que tradicionalmente corren en CPUs de propósito general.
- En la actualidad existen dos plataformas básicas para este propósito:
 - **OpenCL**, entorno abierto
 - **CUDA**, entorno propietario de nVIDIA
- Desde principios de 2000 existe una comunidad muy activa en este tema: www.gpgpu.org
- Cada pocos meses, en los congresos más importantes de Supercomputación y Gráficos, realizan cursos relacionados con el tema.
- Las primeras aproximaciones a GPGPU eran a través de las APIs gráficas (**OpenGL**).
- A partir de 2007 todos los cursos son con **CUDA** y desde 2009 aparece **OpenCL**.
- Para entender el avance que supone **CUDA** y **OpenCL** es muy interesante estudiar cómo se abordaba este tema en 2002-2006.

¡Todas las afirmaciones y datos del resto del tema han de situarse en esas fechas!



APIs para GPGPU Timeline



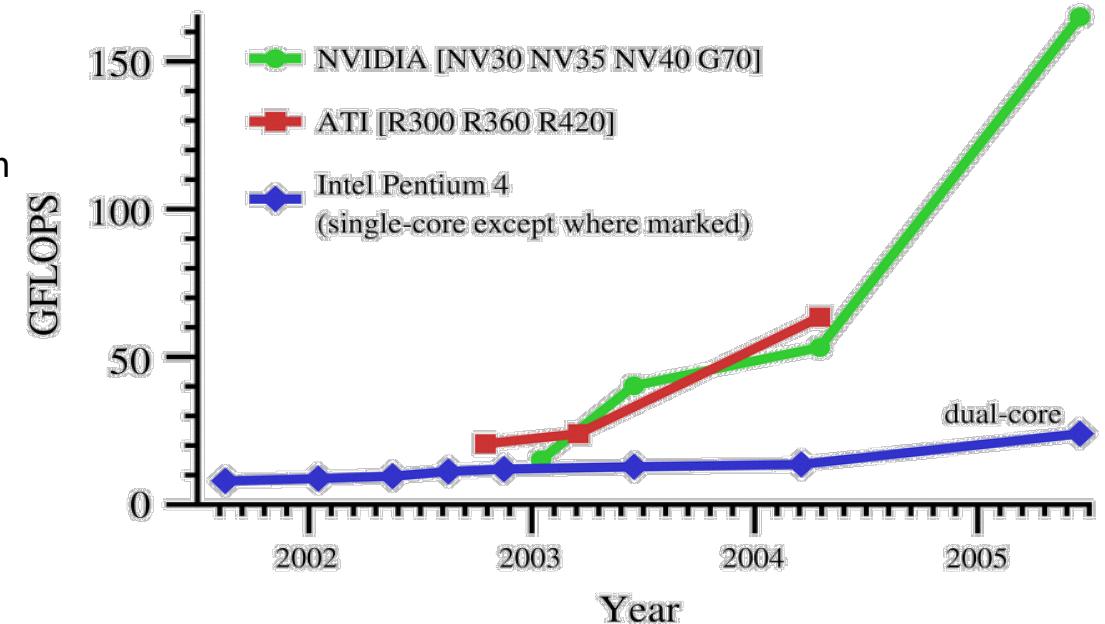
Motivación: Potencia de Cálculo

□ Las GPU son rápidas:

- 24.6 Gflops en una CPU Intel Pentium 4 Dual Core
- 165 Gflops en una GPU NVIDIA GeForce 7800
- 8.5 GBytes/s en un Intel Pentium Extreme Edition
- 37.8 GBytes/s en una ATI Radeon X850 XT Platinum

□ Y cada vez serán más rápidas:

- Factor 1.4x de crecimiento en CPU
 - ✓ 60x en la última década
- Factor 2x de crecimiento en GPUs
 - ✓ > 1000x en la última década



□ ¿Porqué son tan rápidas las GPUs?

- La especial naturaleza de las GPUs, facilita el uso de hardware adicional para cálculo.
- El multimillonario negocio de los videojuegos es una olla a presión que facilita la innovación en este área.



Motivación: Flexibilidad y Precisión

□ Las GPUs son totalmente programables

- Vertex Shader
- Fragment Shader
- Están apareciendo lenguajes de alto nivel para programar GPUs

□ Las GPUs pueden trabajar con números reales

- Coma flotante de 32 bits
 - ✓ Hasta ahora no disponían de coma flotante de 32 bits.
 - ✓ Formatos propios de coma flotante, p.e. de 24 bits.
 - ✓ No soportaban IEEE 754
- Suficiente para muchas aplicaciones [**iPERO NO TODAS!**]



Motivación: Potencial de GPGPU

- La potencia y flexibilidad de las GPUs hacen que sean una plataforma muy atractiva para la aplicaciones de propósito general con altas necesidades de cálculo y ancho de banda.
- Las aplicaciones van desde la simulación de la física de los juegos hasta aplicaciones científicas convencionales.
- El precio de las GPUs es muy bajo en comparación con las prestaciones que ofrece:
 - Workstation Sun Ultra 40 con dos CPUs (\approx 32 GFLOPS): 9.000 €
 - GeForce 7800 (165 GFLOPs): 350€
- Objetivo: hacer que la potencia disponible en las GPUs esté disponible para los desarrolladores de software.

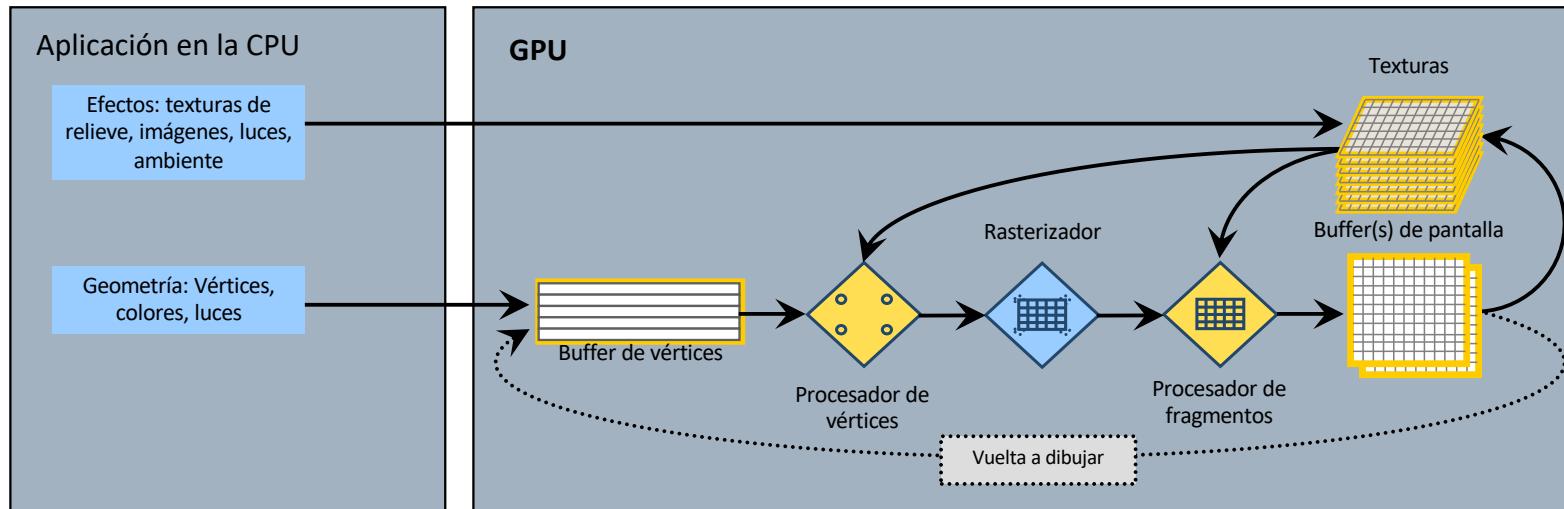


Problema: Dificultad de uso

- Las GPUs están diseñadas para ejecutar aplicaciones gráficas (videojuegos):
 - El modelo de programación es inusual y muy relacionado con los videojuegos
 - El entorno de programación es muy restringido.
- Las arquitecturas utilizadas:
 - Son altamente paralelas
 - Evolucionan muy rápidamente
 - ✓ Varían mucho entre generaciones.
 - ✓ Aplicaciones que funcionan perfectamente en una tarjeta gráfica, dejan de funcionar en la siguiente
 - La información disponible es muy escasa.
- No es “portar” el código a otra CPU. Hay que hacer mucho más.



Elementos programables



2 Elementos Programables:

- Procesador de Vértices
- Procesador de Fragmentos

Tienen características diferentes

Hay que seleccionar 1 de los 2 para correr las aplicaciones de propósito general.

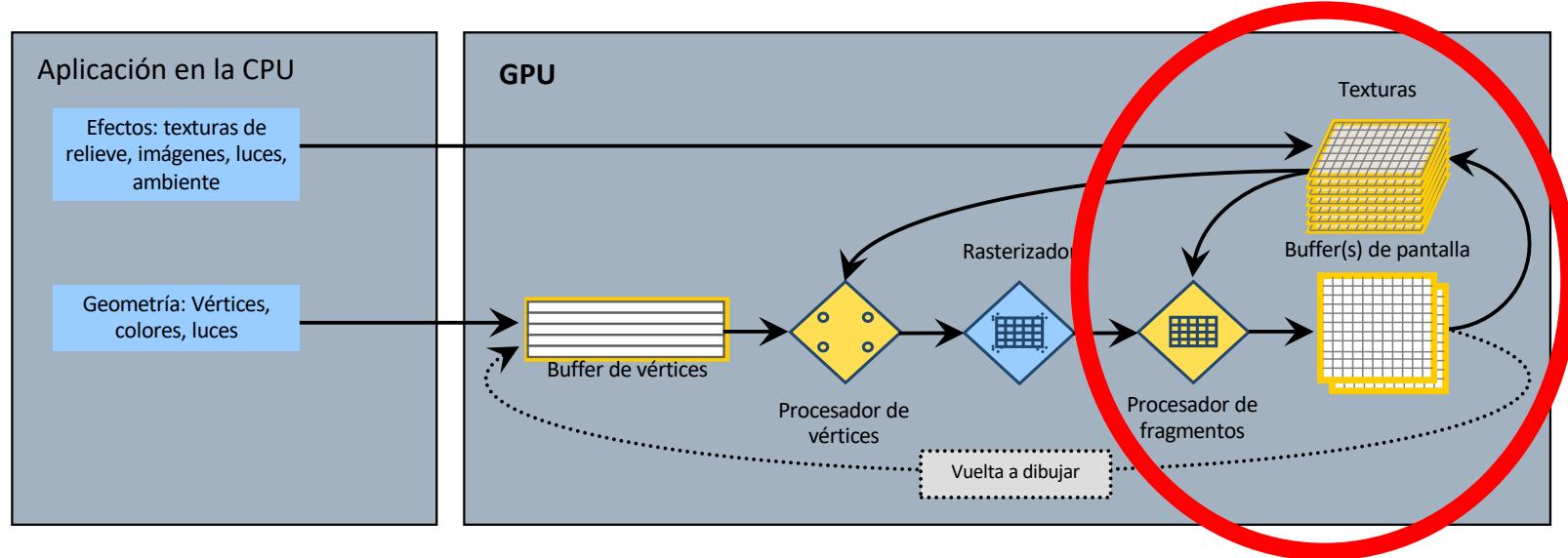
Elementos Programables

nVIDIA	Serie	Year	GPU	Memory MB	VS	FS	TU	ROP	Bandwidth (GB/s)
GeForce FX 5950 Ultra	GeForce FX	2003	NV38	256	3	4	8	4	30,4
GeForce 6800 GT	GeForce6	2004	NV45	256	6	16	16	16	32
GeForce 6800 Ultra	GeForce6	2005	NV45	512	6	16	16	16	33,6
GeForce 7900 GTX	GeForce7	2006	G71	512	8	24	24	16	51,2

ATI - AMD	Serie	Year	GPU	Memory MB	VS	FS	TU	ROP	Bandwidth (GB/s)
Radeon 9800 XT	Radeon R300	2003	R360	256	4	8	8	8	23,36
Radeon X850 XT	Radeon R400	2004	480	256	6	16	16	16	34,56
Radeon X1800 XT	Radeon R500	2005	R520	512	8	16	16	16	48
Radeon X1950 XT	Radeon R500	2006	R580+	512	8	48	16	16	57,6



Procesador de Fragmentos



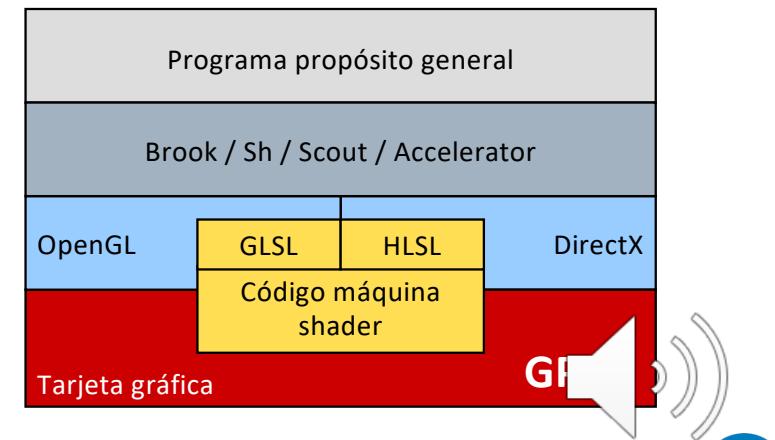
$$A = B + C$$
$$D = D \cdot A$$

FrameBuffer ← textura op textura



Herramientas de Programación

- Lenguajes de alto nivel que esconden la arquitectura
 - **Brook**: compilador de C con extensiones para ‘streams’ (última versión: Oct 2004)
 - **Sh**: librería C++ con un lenguaje de ‘meta-programación’ (última versión: Feb 2006)
 - **Scout**: un lenguaje, parecido al C*, para análisis de datos y visualización, con su compilador, GUI/IDE, (Sin noticias desde 2006)
 - **Accelerator**: librería Microsoft similar a Brook, pero usando arrays (versión de prueba publicada en abril del 2006).
- Lenguajes de nivel medio: definición del entorno gráfico + lenguaje de sombreado inteligible por el programador:
 - **OpenGL + GLSL**
 - **DirectX9 + HLSL** (equivale en un 99% al Cg de nVIDIA)
- Código máquina del shader:
 - Más Rendimiento



Herramientas de Programación

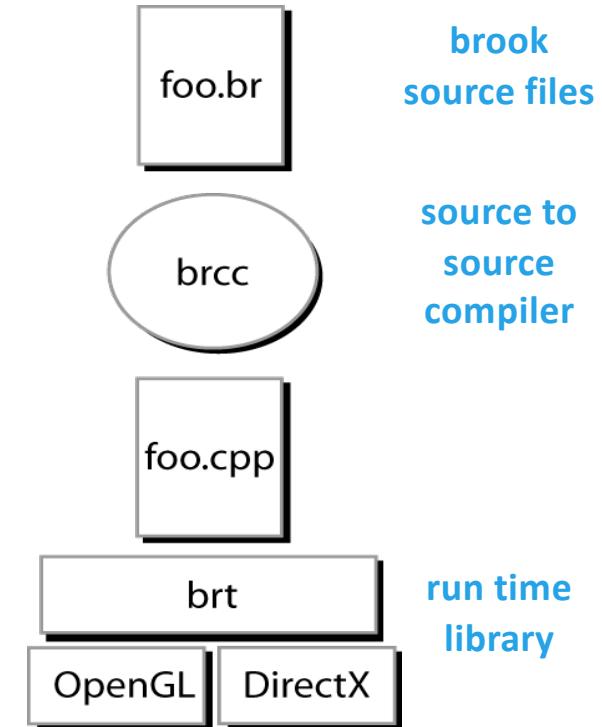
```
C
float a[1000];
float b[1000];
float c[1000];

for (i=0; i<1000; i++)
    c[i] = a[i] + b[i];
```

```
kernel void foo(float a<>,
                float b<>,
                out float c<>) {
    c = a + b;
}

float a<1000>;
float b<1000>;
float c<1000>;
foo(a, b, c);
```

Brook



Herramientas de Programación

```
float a[1000];          C
float r = 0;
for (i=0; i<1000; i++)
    r = r + a[i];
```

```
reduce void sum(float a<>,
                reduce float r<>)
{
    r = r + a;
}
```

```
float a<1000>;
float r;

sum(a, r);
```

Brook

REDUCCIONES

- Construcción muy importante.
- Sólo aplicable con Operadores Asociativos:
 $(a+b)+c = a+(b+c)$
- Operadores: suma, producto, max, min, OR, AND, XOR.



Herramientas de Programación

```
float a[1000];  
float b = 0.5;  
float c[1000];  
  
for (i=0; i<1000; i++)  
    c[i] = b * a[i];
```

C

```
ShAttrib1f b = 0.5;  
ShProgram program =  
SH_BEGIN_PROGRAM() {  
    ShInputAttrib1f input;  
    ShOutputAttrib1f output;  
    output = input * b;  
} SH_END;  
  
ShCompile(program);  
ShChannel<ShAttrib1f> a(1000);  
ShChannel<ShAttrib1f> c(1000);  
c = program << a;
```

Sh



Herramientas de Programación

```
void main( void) {  
    float vX[1000];  
    float vY[1000];  
    float res[1000];  
    float a = 0.5;  
  
    for (i=0; i <1000; i++)  
        res[i] = vY[i] + a* vX[i];  
}
```

C

```
sampler2D vY;  
sampler2D vX;  
float a;  
  
void main(in float2 idx:TEXCOORD0,  
          out res: COLOR0) {  
    float4 y = tex2D(vY, idx);  
    float4 x = tex2D(vX, idx);  
    res = y + a * x;  
}
```

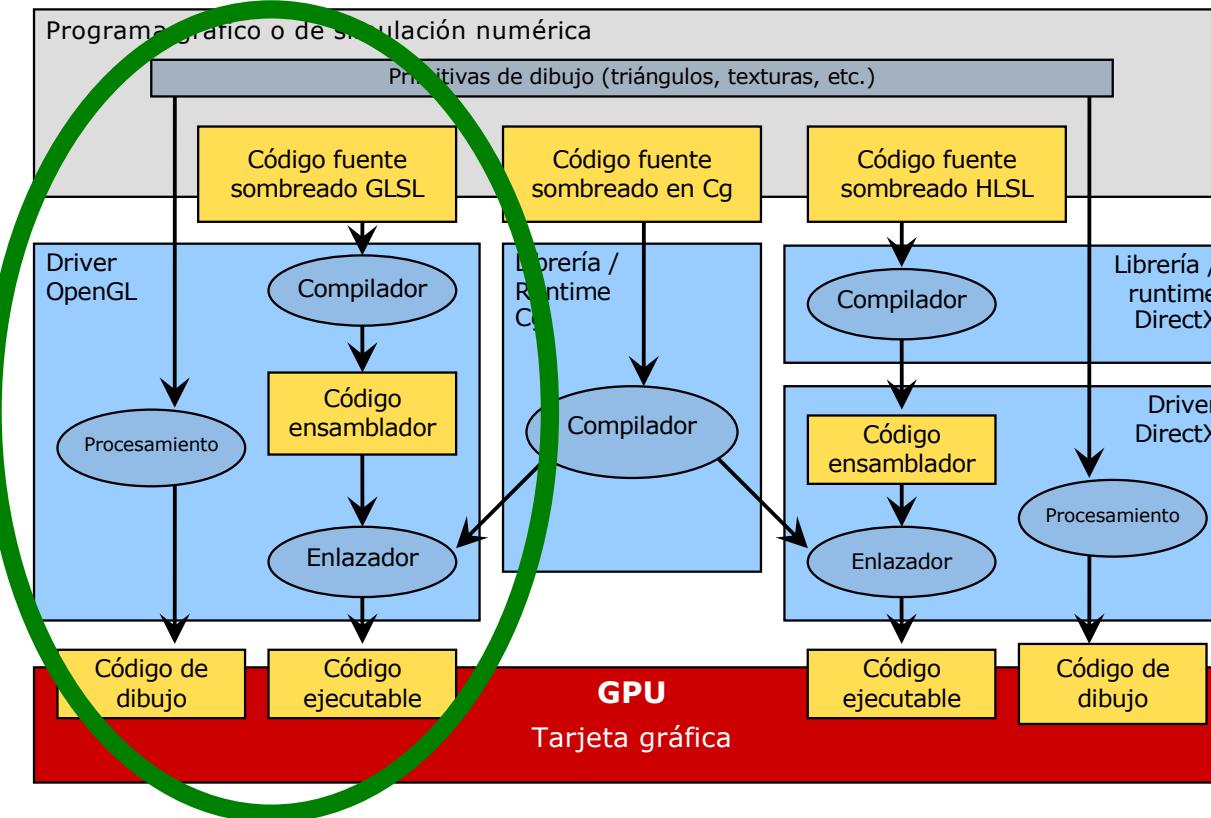
HLSL

```
uniform sampler2D vY;  
uniform sampler2D vX;  
uniform float a;  
  
void main(void) {  
    float2 idx = gl_TexCoord[0];  
    vec4 y = texture2D(vY, idx.st);  
    vec4 x = texture2D(vX, idx.st);  
    gl_FragColor = y + a * x;  
}
```

GLSL



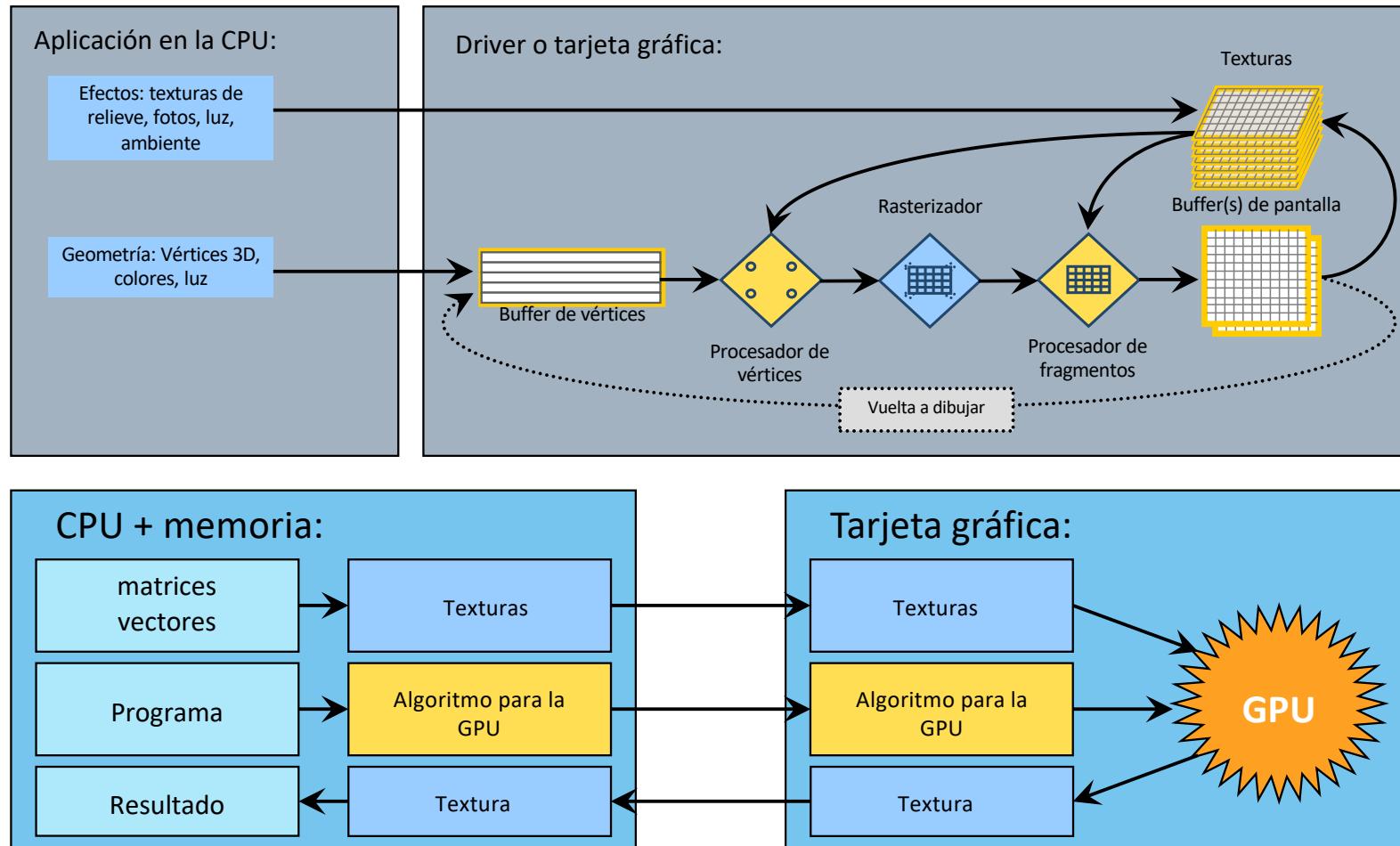
Herramientas de Programación



Alternativa más utilizada: OpenGL + GLSL



Funcionamiento



Caso de Estudio

- Vamos a utilizar un ejemplo sencillo para presentar los detalles

```
float vX[1000];  
float vY[1000];  
float a = 0.5;  
  
for (i=0; i <1000; i++)  
    vY[i] = vY[i] + a*vX[i];
```

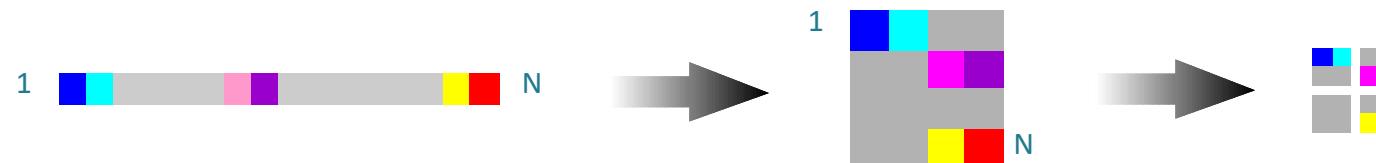
C

- El primer paso es empaquetar los vectores en texturas.
- En general podemos tener diferentes tipos de datos a empaquetar:
 - Vectores
 - Matrices Densas
 - Matrices en Banda
 - Matrices Dispersas



Vectores

- Para almacenar un vector se puede utilizar una textura 1D.
- Por motivos de tamaño y eficiencia es más adecuado utilizar una textura 2D

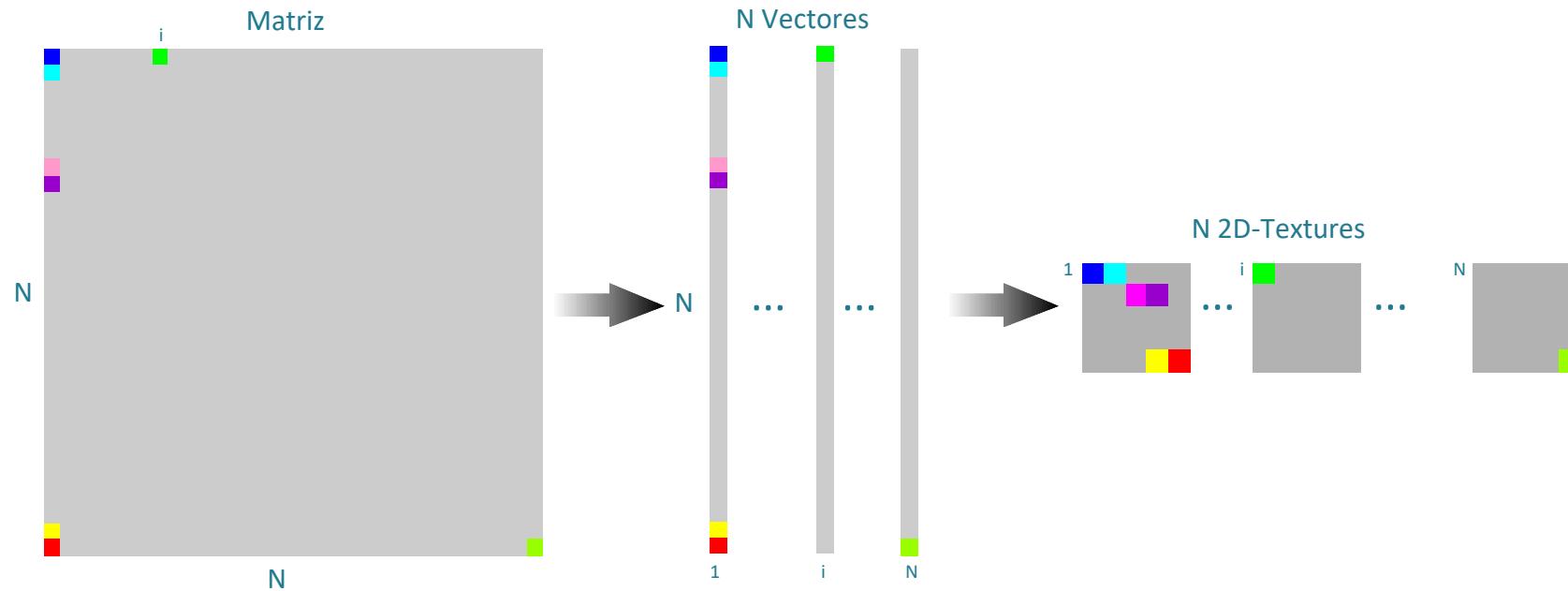


- ¡Atención a los restos!
- Tamaño y forma de las texturas



Matrices Densas

- Se tratan las matrices densas como conjuntos de vectores columna (o fila).
- De nuevo, cada vector se almacena como una textura 2D. [PROBLEMAS de TAMAÑO]

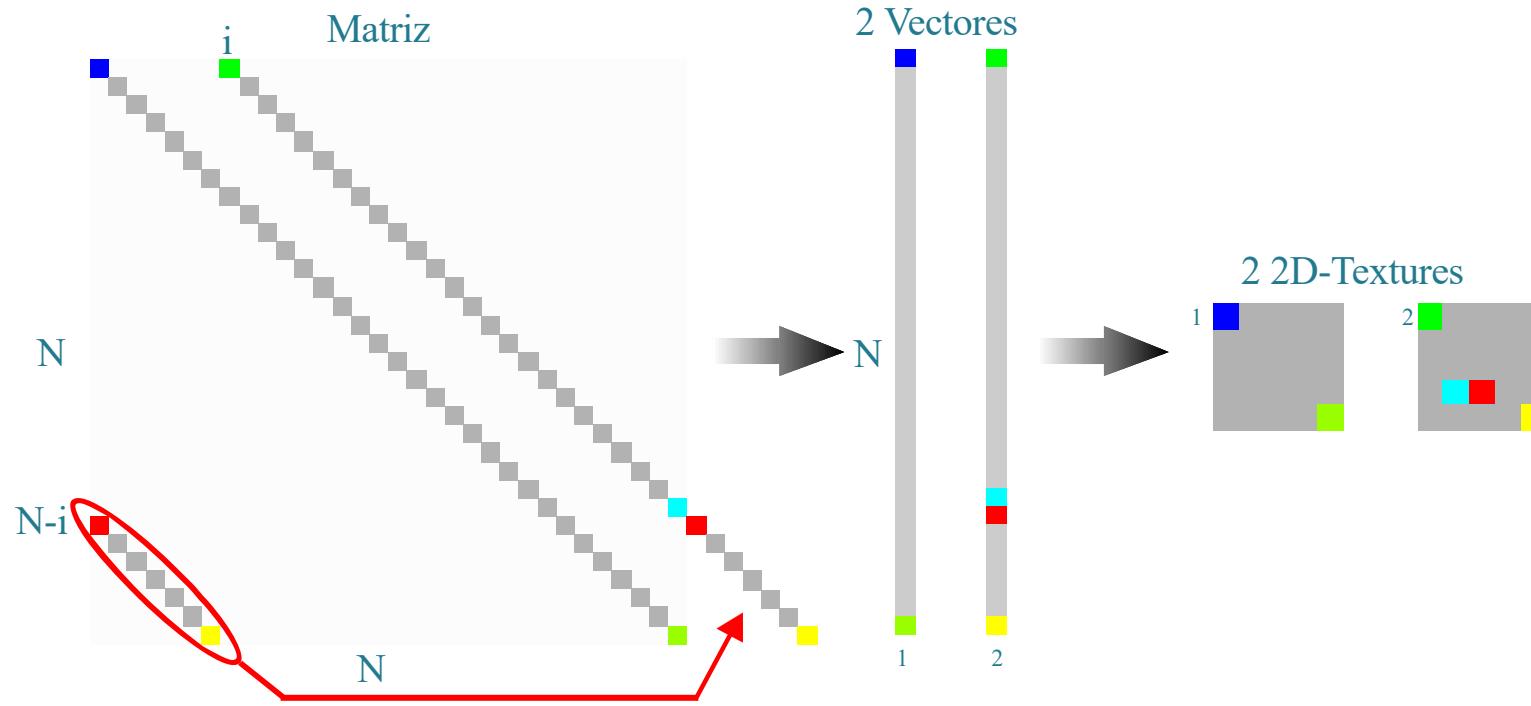


- Nada impide almacenar las matrices como conjuntos de vectores fila
- Muy adecuado para ciertos algoritmos [p.e. producto de matrices]



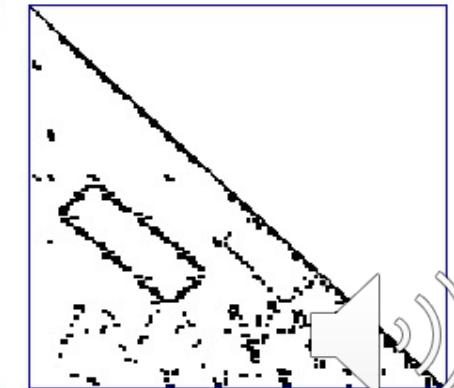
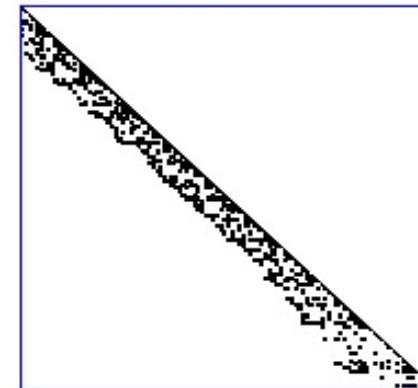
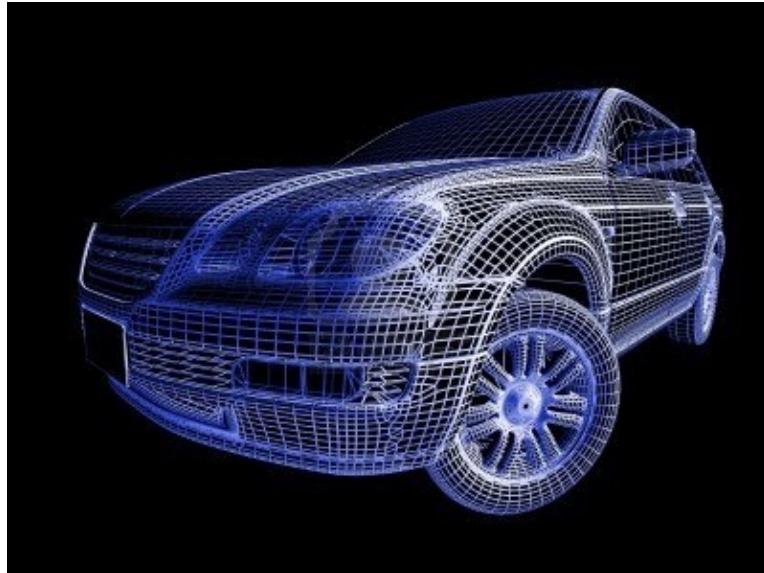
Matrices en Banda

- Matrices con la mayoría de sus elementos a cero, a excepción de algunas diagonales.
- Se almacenan las diagonales en texturas 2D.
- Se aprovechan los huecos para ahorrar espacio.



Matrices Dispersas y en Banda

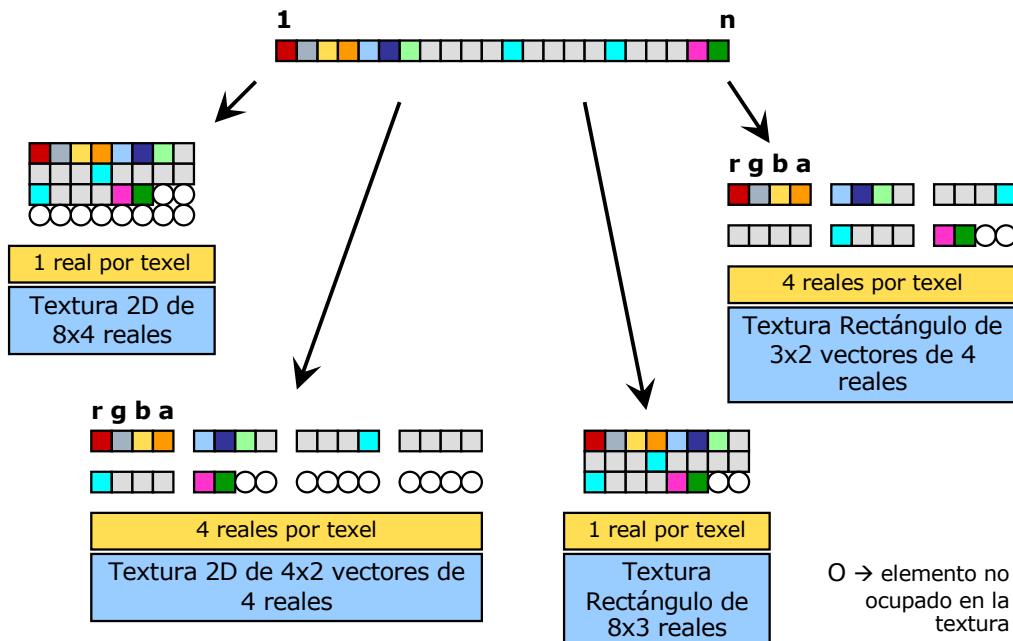
- Suelen ser sistemas de ecuaciones, por ejemplo, de aplicaciones con elementos finitos.



¿Qué tipo de textura escogemos?

Criterios:

- Minimizar la memoria perdida: textura rectangular
- maximizar las operaciones aritméticas por ciclo: RGBA
- escoger un formato que soporte reales de 32 bits.



Tipo de textura	NVIDIA
Textura 2D: extensión requerida para reales	ARB_texture_float
Textura rectangular: extensión requerida	NV_float_buffer
Textura 2D: 1 componente (luminancia)	GL_FLOAT_R32_NV
Textura 2D : 4 componentes (rgba)	GL_RGBA32F_ARB
Textura rectangular : 1 componente (luminancia)	GL_FLOATR32_NV
Textura rectangular : 4 componentes (rgba)	GL_FLOAT_RGBA32_NV

¿Qué pasa si los datos no caben en 1 textura?



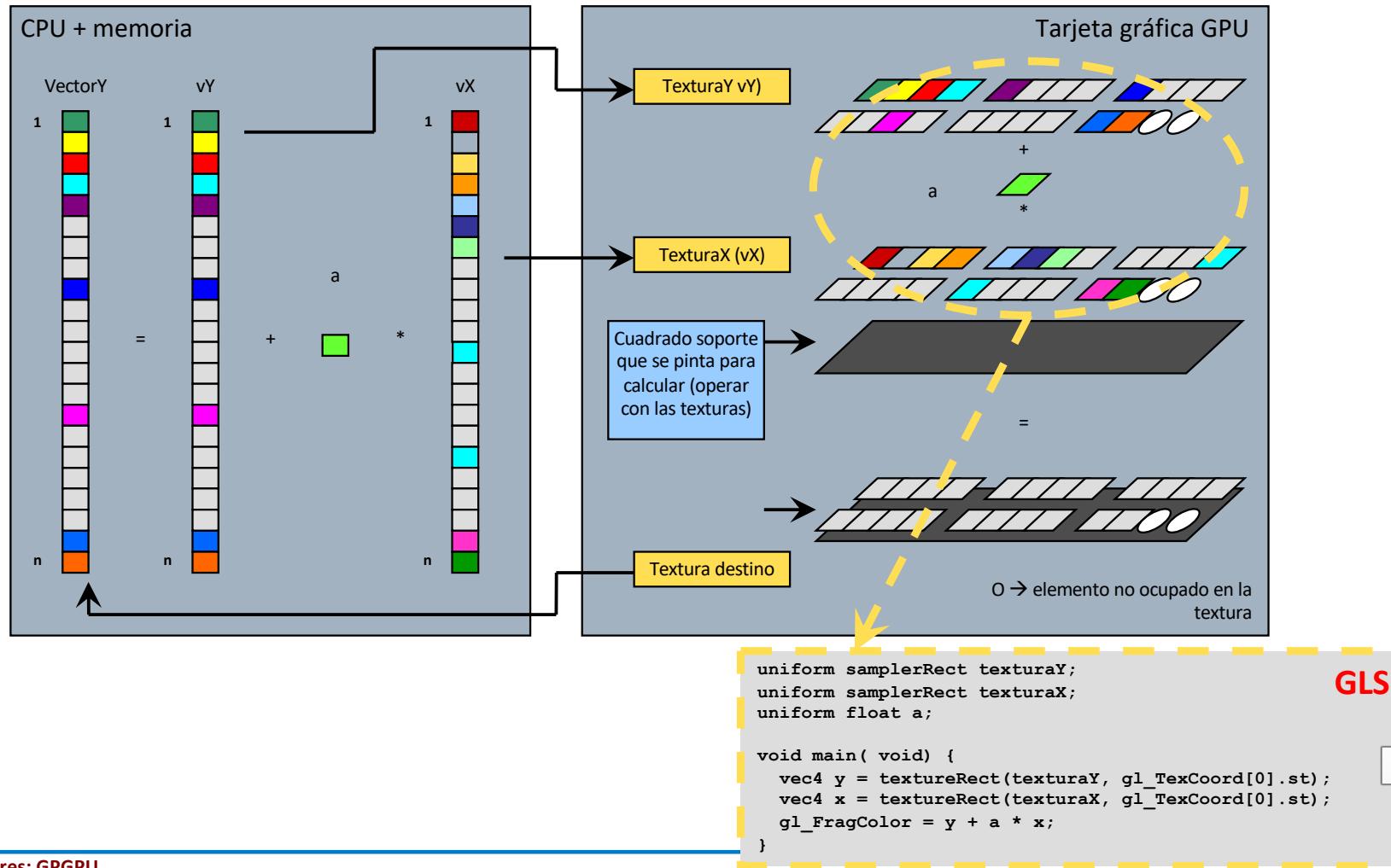
¿Cómo cargamos la textura?

- A una textura se puede acceder utilizando diferentes tipos de filtro: bilinear, trilinear, anisotrópico, etc. Nos hemos de asegurar de **no utilizar ningún tipo de filtrado**.

```
float vX[1000]; // nuestro vector de reales  
  
// Hay que calcular el tamaño de la textura y reajustar vX con los 0's necesarios  
GLuint id_texVectorX; // identificador de la textura  
  
glEnable(GL_TEXTURE_RECTANGLE_ARB);  
glGenTextures(1, &idx_texVectorX);  
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, id_texVectorX);  
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_S, GL_CLAMP);  
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T, GL_CLAMP);  
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, GL_FLOAT_RGBA32_NV,  
            ancho_tex, alto_tex, 0, GL_RGBA, GL_FLOAT, NULL);  
  
//Transferimos los datos a la tarjeta:  
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, ancho_tex, alto_tex,  
                GL_RGBA, GL_FLOAT, vX);
```



Funcionamiento



Funcionamiento: detalle del Kernel

```
uniform samplerRect texturaY; GLSL
uniform samplerRect texturaX;
uniform float a;

void main( void ) {
    vec4 y = textureRect(texturaY, gl_TexCoord[0].st);
    vec4 x = textureRect(texturaX, gl_TexCoord[0].st);
    gl_FragColor = y + a * x;
}
```

Vectores de 4 elementos

= ¿quién soy yo?

Sólo está el código de 1 "fragmento".
Este código se aplica a todos los "fragmentos" de la imagen a dibujar.

¿Dónde se hace el dibujo? y ¿qué es exactamente?



¿Dónde dejar el resultado?

- Finalmente hemos de dejar el resultado en una textura.
- Por definición, en un kernel, no se puede utilizar una textura en lectura / modificación / escritura:
 - $tx = tx + a * ty$ [NO ES POSIBLE]
 - $tx2 = tx + a * ty$ [HAY QUE USAR TEXTURA AUXILIAR]
- Lo que hay que hacer es un “dibujo fuera de pantalla” (*offscreen*)
- Se utiliza un **FrameBuffer Object** (FBO)
 - Se crean en el mismo contexto gráfico
 - El formato del FBO viene determinado por el formato de la textura
 - Las texturas pueden ser compartidas por diferentes FBO
 - Los FBO están disponibles desde OpenGL 2.0



¿Dónde dejar el resultado?

```
// definición de la textura destino [falta tamaño, tipo, ...]
GLuint id_texDestino;
 glGenTextures(1, &id_texDestino);

// definición del buffer off-screen
GLuint id_fbo;
 glGenFramebuffersEXT(1, &id_fbo);
 glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, id_fbo);

// ligamos la textura al FBO
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                           GL_TEXTURE_RECTANGLE_ARB, id_texDestino, 0);

// activamos el buffer de destino para 'calcular' en él
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
( . . . )

// ahora recogemos el resultado
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0, 0, ancho_tex, alto_tex, GL_RGBA, GL_FLOAT, vY);
```



El kernel se ha de compilar

```
// creamos los objetos de programa y sombreado
GLhandleARB ObjPrograma = glCreateProgramObjectARB();
GLhandleARB ObjSombreado = glCreateShaderObject(GL_FRAGMENT_SHADER_ARB);

// definimos y compilamos el código
GLcharARB codigo = \
    "uniform samplerRect texturaY;" \
    "uniform samplerRect texturaX;" \
    "uniform float a;" \
    "void main( void) { " \
        "    vec4 y = textureRect(texturaY, gl_TexCoord[0].st);" \
        "    vec4 x = textureRect(texturaX, gl_TexCoord[0].st);" \
        "    gl_FragColor = y + a * x;" \
    "}";
glShaderSource(ObjSombreado, 1, &codigo, NULL);
glCompilerShaderARB(ObjSombreado);
glLinkProgramARB(ObjPrograma);
GLint exito;
glGetObjectParameterivARB( ObjPrograma, GL_OBJECT_LINK_STATUS_ARB, &exito);
if ( !exito) printf( "Error compilando programa\n");

// ahora cogemos los identificadores de las variables que necesitamos
Glint id_VariableTexturaX = glGetUniformLocationARB(ObjPrograma, "texturaX");
Glint id_VariableTexturaY = glGetUniformLocationARB(ObjPrograma, "texturaY");
Glint id_VariableTexturaAlpha = glGetUniformLocationARB(ObjPrograma, "a");
```



Y ahora, a “dibujar” [¿o calcular?]

□ Últimos pasos:

- Activar el programa
- Asociar las texturas con las unidades de textura y estas con los identificadores del programa

```
glUseProgramARB(ObjPrograma);  
  
// asociamos las texturas con sus unidades de texturas e identificadores  
// el vector vY  
glActiveTexture( GL_TEXTURE1);  
glBindTexture( GL_TEXTURE_RECTANGLE_EXT, id_texVectorY);  
glUniform1iARB( id_VariableTexturaY, 1); // unidad de textura 1  
  
// el parámetro a  
glUniform1fARB( id_VariableAlpha, a); // directamente el parámetro  
  
// el vector vX  
glActiveTexture(GL_TEXTURE0);  
glBindTexture(GL_TEXTURE_RECTANGLE_EXT, id_texVectorX);  
glUniform1iARB(id_VariableTexturaX, 0); // unidad de textura 0
```



Y ahora, a “dibujar” [¿o calcular?]

- Para hacer los cálculos, hay que “**dibujar**” alguna cosa.
 - Hay que escoger una proyección que tenga una correspondencia 1 a 1 entre los píxeles de destino y las texturas de entrada.
 - Proyección ortogonal.
 - Coordenadas de las texturas para obtener datos de entrada
 - Coordenadas de los píxeles para los datos de salida

```
glViewport( 0, 0, ancho_tex, alto_tex);  
glMatrixMode(GL_PROJECTION);  
gluOrtho2D(0.0, ancho_tex, 0.0, alto_tex);  
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glDisable(GL_DEPTH_TEST);
```



Y ahora, a “dibujar” [¿o calcular?]

- Lo último, dibujamos un rectángulo

```
// activamos el buffer de destino:  
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);  
  
glBegin(GL_QUADS);  
glTexCoord2f(0.0, 0.0);  
glVertex2f(0.0, 0.0);  
glTexCoord2f(ancho_tex, 0.0);  
glVertex2f(ancho_tex, 0.0);  
glTexCoord2f(ancho_tex, alto_tex);  
glVertex2f(ancho_tex, alto_tex);  
glTexCoord2f(0.0, alto_tex);  
glVertex2f(0.0, alto_tex);  
glEnd();
```

- Sólo un detalle más. Este código funciona con texturas GL_TEXTURE_RECTANGLE, que se indexan como una matriz C. Si la textura es GL_TEXTURE_2D se indexa con valores entre 0 y 1 y hay que cambiar las coordenadas del dibujo.



Resumiendo

Todos los pasos

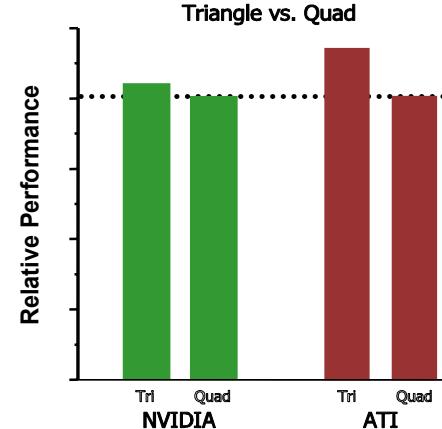
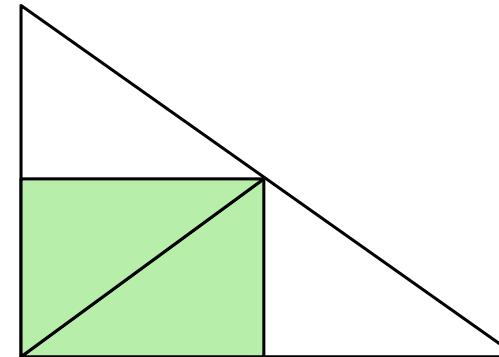
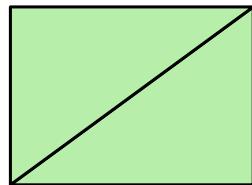
1. Creación del contexto gráfico de la aplicación (ventana)
2. Almacenar vectores en texturas
3. Creación de la textura y del buffer *offscreen*
4. Creación, compilación y enlazado del programa
5. Inicialización de entorno gráfico *offscreen*
6. Mapeo de variables del programa a texturas
7. Envío de las texturas y del programa a la tarjeta gráfica
8. Cálculo (dibujamos un rectángulo)
9. Recogemos la textura del resultado
10. Borramos texturas y destruimos contextos gráficos

¡Fácil!, ¿no?



Trucos / Curiosidades

- En algún momento hay que “dibujar algo”.
- Dos opciones



```
glBegin(GL_QUADS);  
glVertex2f(0, 0);  
glVertex2f(width, 0);  
glVertex2f(width, height);  
glVertex2f(0, height);  
glEnd();
```

```
glViewport(0, 0, width, height)  
glBegin(GL_TRIANGLE);  
glVertex2f(0, 0);  
glVertex2f(width*2, 0);  
glVertex2f(0, height*2);  
glEnd();
```

¡Esta opción es la más rápida!

¿Cuál es más rápido?

¿Porqué hay diferencia?



Trucos / Curiosidades

- A pesar de que el hardware soporta múltiples outputs ...

#Outputs	Ancho banda efectivo
1	9,25 GB/s
2	2,91 GB/s
3	3,97 GB/s
4	2,76 GB/s

ATI 9800XT

¿Cuál es la razón?

¡No debería ser así!



Reducción

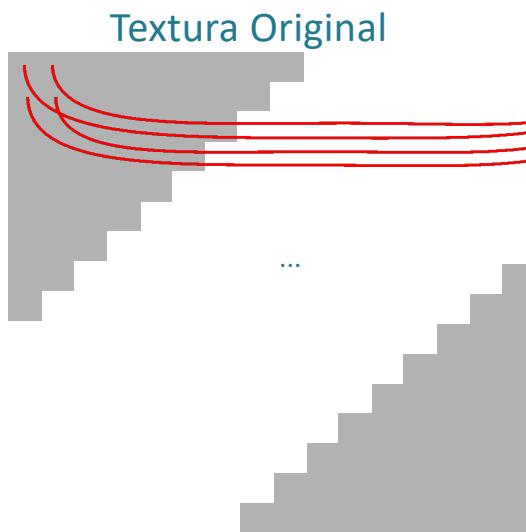
- Algoritmo fundamental que aparece en múltiples ocasiones.

```
float a[1000];  
float sum = 0;  
  
for (i=0; i<1000; i++)  
    sum = sum + a[i];
```

C

```
float a[1000];  
float m = a[0];  
  
for (i=1; i<1000; i++)  
    m = max(m, a[i]);
```

C

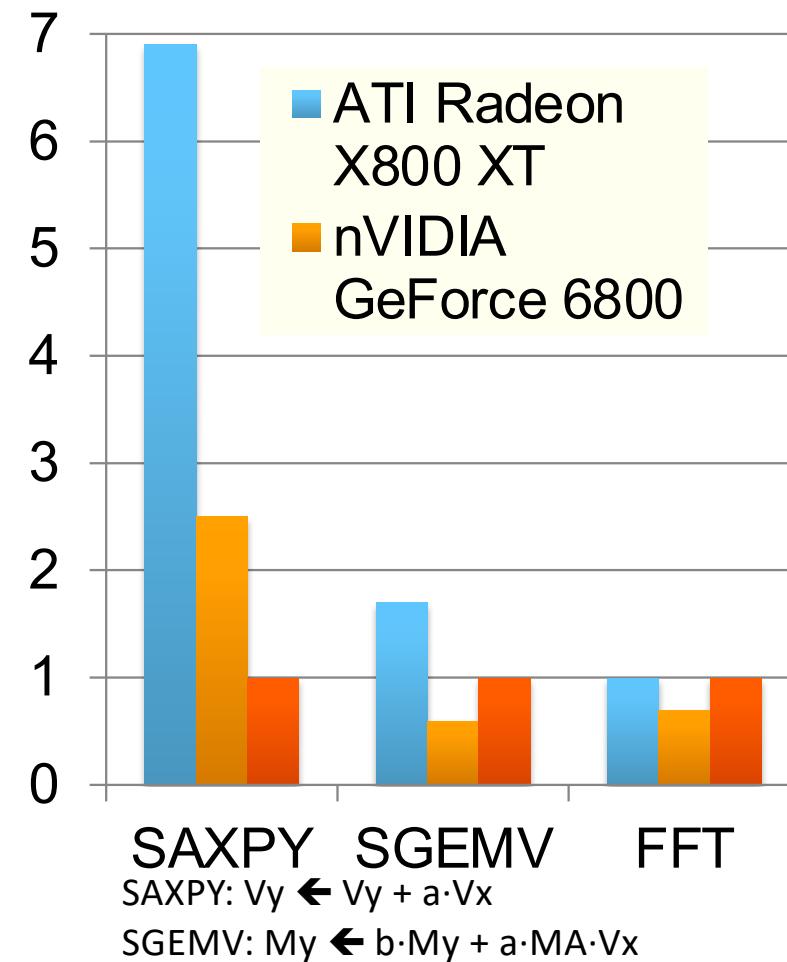
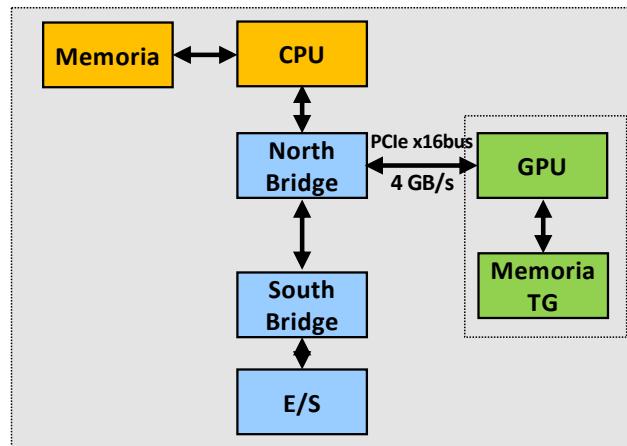


Se parece mucho a la construcción de los mipmaps.



¿Y el rendimiento?

- Los resultados son buenos en algunos casos.
- Desesperanzadores en otros.
- Uno de los elementos que más perjudica el rendimiento de estas aplicaciones es la comunicación con la Tarjeta gráfica.



GPGPU sobre OpenGL

Conceptos Básicos:

- Arrays = Texturas
- Kernels = Shaders
- Computing = Drawing
- Feedback (reusar cálculos previos)



Dificultades

- Nueva sintaxis a aprender.
- Nueva filosofía de programación.
- “Reprogramación” de algoritmos.
- Sincronización de datos.
- Interacción con la librería gráfica en algunos casos.
- Coma Flotante de 32 bits
- Acceso a datos a través de texturas
- Comunicación a través del bus PCIe
- Los datos no caben en memoria.



Deseos de “futuro”

- Más elementos de cálculo (y unificados,)
- IEEE 754 ()
- Coma Flotante de 64 bits ()
- Acceso directo a datos con instrucciones load/store ()
- Evitar la comunicación a través del bus PCIe ()
- Aumentar la capacidad de memoria ()
- No usar la librería gráfica ()



Recordando CUDA: el kernel

```
void saxpyS (int N, float a, float *x, float *y) {  
    for (int i=0; i<N; i++)  
        y[i] = a * x[i] + y[i];  
}  
  
// Invocación  
saxpyS(N, 3.5, x, y);
```

Código Secuencial

Describe **TODO** lo que hay que hacer.

```
_global_  
void saxpyP (int N, float a, float *x, float *y) {  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if (i<N) y[i] = a * x[i] + y[i];  
}
```

Kernel CUDA

Indica **quién** soy.

Describe cómo se calcula **1 elemento** del vector: $y[i]$. Es lo que se ejecuta en un thread.

Recordando CUDA: el kernel vs código HLSL

```
uniform samplerRect texturaY;                                GLSL
uniform samplerRect texturaX;
uniform float a;

void main(void) {
    vec4 y = textureRect(texturaY, gl_TexCoord[0].st);
    vec4 x = textureRect(texturaX, gl_TexCoord[0].st);
    gl_FragColor = y + a * x;
}
```

```
__global__
void saxpyP (int N, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<N) y[i] = a * x[i] + y[i];
}
```

Kernel CUDA

Indica **quién soy**.

Describe cómo se calcula **1 elemento** del vector: $y[i]$.
Es lo que se ejecuta en un thread.

Recordando CUDA: el código del HOST

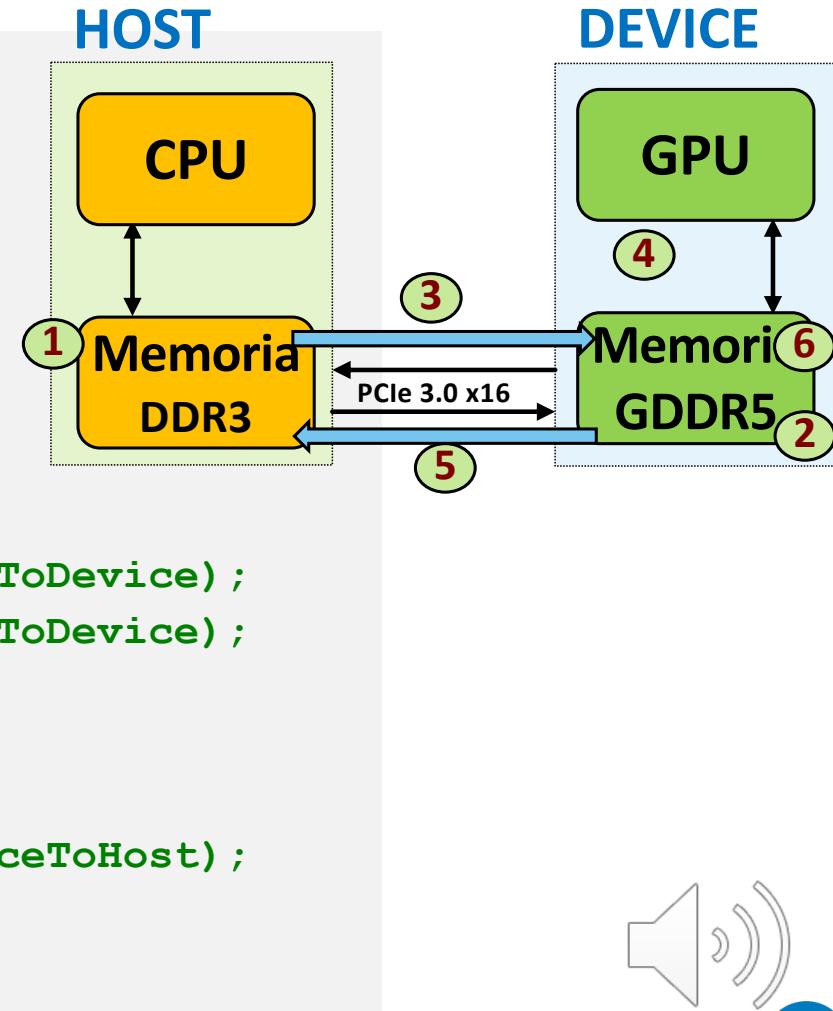
```
1 // Obtener Memoria en el host  
2 // Obtener Memoria en el device  
3 // Copiar datos desde el host en el device  
4 // Ejecutar el kernel  
5 // Obtener el resultado desde el host  
6 // Liberar Memoria del device
```

unsigned int numBytes = N * sizeof(float);
float* h_x = (float*) malloc(numBytes);
float* h_y = (float*) malloc(numBytes);

float* d_x, d_y;
cudaMalloc((void**)&d_x, numBytes);
cudaMalloc((void**)&d_y, numBytes);

cudaMemcpy(d_x, h_x, numBytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, numBytes, cudaMemcpyHostToDevice);

saxpyP<<<(N+63)/64, 64>>>(N, 3.5, d_x, d_y);
cudaMemcpy(h_y, d_y, numBytes, cudaMemcpyDeviceToHost);
cudaFree(d_x); cudaFree(d_y);



CUDA vs GPGPU (OpenGL)

- No es necesario tener ninguna experiencia con GPUs o una API gráfica
- No es necesario utilizar ninguna API gráfica. CUDA se encarga de gestionar la GPU.
- CUDA es C con extensiones:
 - Los datos se definen y acceden de forma natural.
 - El programador describe el comportamiento de un thread, y el código se instancia automáticamente para miles de threads.
 - El programador ha de realizar de forma explícita la transferencia de datos entre CPU y GPU.
 - El programador ha de gestionar la memoria compartida de forma explícita.
 - Una API muy simple que permite manipular dispositivos, memoria, transferencias, ...
- Las aplicaciones CUDA son escalables
- No hay que manipular el kernel (compilar, asociar parámetros, ...)
- En pocos días se pueden aprender los conceptos básicos de CUDA.



Lectura Complementaria

- Chris J. Thompson, Sahngyun Hahn and Mark Oskin
“Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis”
35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), 2002.
- John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone and James C. Phillips (UIUC)
“GPU Computing”
Proceedings of the IEEE, Vol. 96(5), pp. 879-899, May 2008





UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

GPGPU (General Purpose on Graphical Processor Units)

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

