“PROGRAMACIÓN GENERAL SOBRE PROCESADORES GRÁFICOS NVIDIA”

GONZALO LUZARDO M.

JUNIO 2009

ÍNDICE

1. Introducción

El objetivo de el presente trabajo es de explicar el funcionamiento y los pasos necesarios que se necesitan para implementar en un GPU nVidia la operación **saxpy()** como se la conoce en las librerías BLAS. De igual forma, su finalidad es de mostrar de manera clara y sencilla, cómo desarrollar una aplicación sencilla que haga uso del poder de cómputo de un procesador gráfico de una NVidia, de tal forma que, en la medida de lo posible, pueda ser extendida a aplicaciones más complejas.

La función saxpy, se define como: *"Para dos vectores **x** y **y** de tamaño **N** y un valor escalar **alpha**, se calcula la siguientes sumatoria escalar de vectores:* $y = y + alpha\ x$*"*. La función **saxpy()**, por su sencillez de implementación resulta ser un buen ejemplo para ilustrar los conceptos de GPGPUs[1].

GPGPU o General-Purpose Computing on Graphics Processing Units, es un concepto reciente dentro del campo de la informática, el cual trata de estudiar y aprovechar las capacidades de cómputo de una GPU o procesador gráfico.

Una GPU es un procesador diseñado para los cómputos implicados en la generación de gráficos 3D en tiempo real. Algunas de sus características como la relación coste/potencia de cálculo, gran capacidad de paralelismo y optimización para cálculos de coma flotante; resultan atractivas para ser aprovechados en aplicaciones en otros campos diferentes a gráficos por ordenador, como en el ámbito del cómputo científico y de simulación por ordenador.

Desde sus inicios el desarrollo de aplicaciones GPGPU se había hecho bien en ensamblador, o en alguno de los lenguajes específicos para aplicaciones gráficas usando la GPU, como GLSL, Cg o HLSL. Recientemente han surgido herramientas para facilitar el desarrollo de aplicaciones GPGPU, las cuales se han hecho muy populares debido a que ayudan a abstraer muchos de los detalles relacionados con los gráficos, y a su vez presentar una interfaz de más alto nivel.

Una de las herramientas más populares para el desarrollo de aplicaciones GPGPU es BrookGPU, desarrollada en la Universidad de Stanford, que consiste en una extensión a ANSI C, la cual proporciona nuevos tipos de datos y operaciones automáticamente convertidos a una implementación que aprovecha la GPU sin intervención explícita por parte del programador.

Por otro lado existen esfuerzos de algunos fabricantes de procesadores gráficos para crear herramientas que faciliten el desarrollo de aplicaciones GPGPU. CUDA[2] es un compilador y conjunto

[1] Del acrónimo en inglés General-Purpose computation on GPUs (Cómputo de propósito general sobre GPUs).

de herramientas de desarrollo que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos para ejecutar en GPUs. CUDA es desarrollado por nVidia[3], lo que obliga a que para usar esta arquitectura se necesite GPUs y controladores nVidia. Todos los últimos controladores de NVidia contienen los componentes CUDA necesarios. CUDA funciona en todas las GPUs nVidia de la serie G8X en adelante, incluyendo GeForce, Quadro y la linea Tesla.

2. Modelo de programación GPGPU

Debido a las diferencias fundamentales entre las arquitecturas de la GPU y la CPU, no cualquier problema se puede beneficiar de una implementación en la GPU. En concreto, el acceso a memoria plantea las mayores dificultades. Las CPU están diseñadas para el acceso aleatorio a memoria, lo que favorece la creación de estructuras de datos complejas con punteros a posiciones arbitrarias en memoria. En cambio, en una GPU, el acceso a memoria es mucho más restringido.

La tarea del diseñador de algoritmos GPGPU consiste principalmente en adaptar los accesos a memoria y las estructuras de datos a las características de la GPU. Generalmente, la forma de almacenar datos es en un buffer 2D, en lugar de lo que normalmente sería una textura. El acceso a esas estructuras de datos es el equivalente a una lectura o escritura de una posición en la textura. Puesto que generalmente no se puede leer y escribir en la misma textura, si esta operación es imprescindible para el desarrollo del algoritmo, éste se debe dividir en varias pasadas.

Pese a que cualquier algoritmo que es implementable en una CPU lo es también en una GPU, esas implementaciones no serán igual de eficientes en las dos arquitecturas. En concreto, los algoritmos con un alto grado de paralelismo, sin necesidad de estructuras de datos complejas, y con una alta intensidad aritmética, son los que mayores beneficios obtienen de su implementación en la GPU.

3. Requerimientos de Hardware

Para poder ejecutar una aplicación para un GPU, es necesario disponer de una tarjeta de video que soporte GPGPU. Actualmente las tarjetas que proveen dicho soporte son las tarjetas NVIDIA a partir de los modelos GeForce FX y las tarjetas ATI a partir de los modelos RADEON 9500.

La tarjeta gráfica que se utilizará en la realización de este trabajo es una Nvidia GeForce 8400GM, una tarjeta gráfica de buenas prestaciones con una memoria compartida de hasta 1 GB,

[2] Del acrónimo en inglés Compute Unified Device Architecture
[3] Nvidia Corporation es un fabricante estadounidense de procesadores gráficos (GPUs), chipsets, tarjeta gráficas y dispositivos para consolas. Es junto con ATI Technologies e Intel Coporation, uno de los líderes del sector.

especialmente diseñada, por su bajo consumo de energía, para ser utilizada en ordenadores portátiles. Para más información acerca de esta tarjeta de vídeo véase el Anexo A1.

4. Requerimientos de Software

Toda la implementación será hecha sobre la distribución Debian de Linux. En primer lugar necesitamos tener las últimas actualizaciones de los drivers de la tarjeta de video que aloja la GPU, en nuestro caso utilizaremos el driver para Linux versión 185.18.14 para las tarjetas con GPU NVidia.

De igual forma, el sistema que contiene la GPU donde se va a realizar la compilación de la aplicación que implementará la operación `saxpy`, debe tener instalado los siguientes paquetes junto con sus respectivas dependencias:

- Compilador GCC 3.4+, `gcc`
- Cross Plattform Make, `cmake`
- Librerías GLUT, `libglew_dev`
- Librerías GLEW, `libglut_dev`
- Librerías Boost para control de tiempo, `libboost-date-time-dev`

Como administrador, podemos realizar la instalación de todos estos paquetes escribiendo en la siguiente línea de comando en la consola:

```
$aptitude install gcc cmake libglew_dev libglut_dev libboost-date-time-dev
```

5. Configuración de OpenGL

5.1. Configuración de GLUT y GLEW

En general el **Conjunto de Utilidades para OpenGL (GLUT),** provee funciones para el manejo de eventos de las ventanas, crear menús, entre otras.

En nuestro caso será utilizado para configurar el ambiente de programación válido para OpenGL, permitiéndonos con unas pocas líneas de código accesar de manera directa al hardware gráfico. La inicialización es hecha a través de la función `initGLUT`, la cual recibe como parámetros la línea de comandos recibida desde la línea de comandos.

```
// Init glut
void initGLUT(int argc, char **argv) {
    glutInit ( &argc, argv );
    glutCreateWindow("SAXPY OVER GPU");
}
```

Muchas de las características que son requeridas para ejecutar operaciones con punto flotante sobre el GPU no son parte del núcleo OpenGL, sin embargo las **Extensiones OpenGL** (**GLEW)** proporcionan mecanismos para accesar a las características del hardware a través de la inclusión de extensiones para el API de OpenGL.

GLEW como una extensión de OpenGL será utilizada para cargar las librerías que encapsulan todo lo que necesitemos inicializar, como las referencias a las funciones de las extensiones OpenGL que utilizaremos.

```
// Init glew
void initGLEW(void) {
         int err = glewInit();
         if (GLEW_OK != err) {
                  std::cout << ((char*)glewGetErrorString(err));
                  exit(ERROR_GLEW);
         }
}
```

5.2. Configurar OpenGL para que realice un renderizado offscreen

En el pipeline de un GPU, el punto final de toda operación de renderizado es el frame buffer, el cual es un segmento de memoria para gráficos la cual es leída para generar las imágenes que serán mostradas en la pantalla. Un “renderizado offscreen” o renderizado en memoria es aquel renderizado que se ejecuta en el GPU sin que su resultado sea mostrado en pantalla.

Dependiendo de la configuración de pantalla, la máxima profundidad de color que podemos tener es de 32 bits, asignándose 8 bits a cada uno de los cuatro canales: rojo, verde, azul y el canal alpha, que nos sirve para almacenar la información de un color que luego será mostrado en pantalla. Lo que se resume a más de 16 millones de colores diferentes que podemos mostrar.

Dado que la operación **saxpy** trabaja con los valores de punto flotante, 8 bits nos resulta insuficiente con respecto a la precisión. Por otro lado los datos colocados en el frame buffer son **truncados** a valores en un rango de [0-1].

Por suerte existe un conjunto de extensiones de OpenGL que dan soporte para trabajar con valores de punto flotante de 32 bits en una GPU. La extensión para OpenGL llamada EXT_framebuffer_object, nos permite usar un **offscreen buffer** como el destino de las operaciones de renderizado, en nuestro caso los cálculos vectoriales que vamos a realizar. Esta extensión nos proporciona precisión total de los cálculos, y a su vez elimina el problema de truncado mencionado anteriormente. Este tipo de buffers se los denomina Frame Buffer Object (FBO).

La función `initFBO` no permite desactivar el buffer tradicional para realizar los cálculos y utilizar un FBO en los cálculos.

```
GLuint fb;
void initFBO(void) {
    glGenFramebuffersEXT(1, &fb);
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
}
```

6. Preparación de los datos a ser procesados

Acordémonos del cálculo que queremos ejecutar: **y** = **y** + *alpha* * **x** para un vector dado de tamaño *N*. Para esto necesitamos de tres arreglos que almacenen: los valores de punto flotante, una variable de tipo flotante, y el resultado de la operación.

Aunque el cómputo va a ser ejecutado en el GPU, necesitamos almacenar estos arreglos en el CPU e inicializarlos con datos de entrada.

```
float* dataY = (float*)malloc(N*sizeof(float));
float* dataX = (float*)malloc(N*sizeof(float));
float* result = (float*)malloc(N*sizeof(float));
float alpha;

for (int i=0; i<N*N; i++){
     dataX[i] = i+1.0;
     dataY[i] = i+2.0;
}
alpha=2.0;
```

7. Almacenamiento de los datos en texturas

Los arreglos unidimensionales son datos nativos de un CPU, de tal forma que un arreglo multidimensional no es otra cosa que arreglos unidimensionales accesados mediante saltos (offset) sobre arreglos unidimensionales. En cambio, en un GPU los arreglos bidimensionales son los datos nativos.

En los CPU, usualmente se habla acerca de **"índices de un arreglo"**, en cambio en un GPU necesitamos hablar de **"coordenadas de una textura"** para accesar a los valores almacenados en dicha. Normalmente los GPUs trabajan sobre la cuatro tuplas de datos de manera simultánea: existen cuatro canales de color llamados rojo, verde, azul y alfa (RGBA). Lo que una coordenada de textura debe direccionar es un **téxel** (algo muy parecido a un pixel).

7.1. Texturas de puntos flotantes en el GPU

En la CPU simplemente usaremos dos arreglos X y Y, los cuales almacenarán valores de punto flotante (`float`); y en el GPU usaremos texturas de punto flotante para almacenar los datos. Existen varios *tipos de texturas* a nuestra disposición, pero si consideramos sólo aquellas texturas bidimensionales soportadas por nuestro hardware tenemos: **`GL_TEXTURE_2D`** y **`ARB_texture_rectangle`**.

`GL_TEXTURE_2D` es el tipo para texturas bidimensionales en OpenGL, conocidas también como **`texture2D`**, muy utilizadas en OpenGL. Por otro lado **`GL_TEXTURE_RECTANGLE_ARB`** es una extensión OpenGL que provee los llamados **`texture rectangles`**, que tienen la ventaja de ser más fáciles de utilizar que los primeros.

La diferencia principal entre estos dos tipos de texturas radica en que el acceso a las **`texture2D`** es mediante coordenadas normalizadas en el rango de de [0,1] x [0,1], independientemente de la dimensión [0,M] x [0,N] de la textura, a diferencia de las **`GL_TEXTURE_RECTANGLE_ARB`** donde el acceso se lo realiza mediante coordenadas que no están normalizadas.

EL siguiente aspecto que debemos tener el cuenta es el *formato de la textura*. Los GPUs permiten el procesamiento simultaneo de datos escalares, y tuplas de dimensión dos, tres o cuatro. En este ejemplo nos enfocaremos en la utilización de escalares y tuplas de cuatro solamente. El caso más sencillo podría ser asignar una textura que solo almacene el valor de un punto flotante por texel (`GL_LUMINANCE`), sin embargo para aprovechar las capacidades de la GPU podemos utilizar una

textura **GL_RGBA** que almacene cuatro valores de punto flotante por téxel, requiriendo 4x32=128 bits (16 bytes) por texel.

Hay tres extensiones de OpenGL que permiten utilizar valores de punto flotante de simple precisión como un formato interno para texturas: `NV_float_buffer`, `ATI_texture_float` y `ARB_texture_float`. Cada extensión define un conjunto de enumeradores (`GL_FLOAT_R32_NV` por ejemplo) y símbolos (0x8880 por ejemplo) que pueden ser utilizados para definir y asignar texturas.

El numerador para el formato de textura que nos interesa es **`GL_FLOAT_RGBA32_NV`**, el cual indica que queremos almacenar una 4-tupla de valores de punto flotante por cada texel. La extensión **`ARB_texture_float`** contiene el numerador **`GL_RGBA32F_ARB`** que nos permite definir este tipo de formato.

7.2. Almacenar la información de las matrices en la textura

El último problema que debemos abordar es la pregunta de cómo mapear un vector que está en el CPU en una textura de la GPU. La manera más sencilla sería la siguiente: Un vector de tamaño *N* es mapeado en una textura de formato RGBA de $\sqrt{N/4}$ x $\sqrt{N/4}$, asumiendo que el *N* es elegido de tal forma que el mapeo "encaje" perfectamente. Por ejemplo, $N = 1024^2$ produce una textura de tamaño de 512 x 512. Definimos el valor de *N* como el tamaño de la textura.

La función `setupTextureTarget`, nos permite configurar una textura de tipo flotante que utiliza los cuatro canales para el almacenamiento de la información referente a las matrices sobre las que vamos a operar. Recibe como parámetros el identificador de la textura que queremos configurar junto con el tamaño.

```
GLuint yTexID; // Y read only
GLuint xTexID; //  X

GLuint aTexID; // alpha

// Generate textures
glGenTextures (1, &yTexID); //Create y
glGenTextures (1, &resultTexID); //Create result
glGenTextures (1, &xTexID); //Create x
```

```
// Sets up a floating point texture with NEAREST filtering.
void setupTextureTarget(const GLuint texID, const int texSize) {
        // make active and bind
        glBindTexture(GL_TEXTURE_RECTANGLE_ARB,texID);
        // turn off filtering and wrap modes
        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);
        // define texture with floating point format
        glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,
                        0,
                        GL_RGBA32F_ARB,
                        texSize,
                        texSize,
                        0,
                        GL_RGBA,
                        GL_FLOAT,
                        0);
}
```

Esta función nos permite:

1. Definir el tipo de textura con la que vamos a trabajar.
2. Definir el formato y el tamaño de la textura.
3. Desactiva los filtros y truncado de datos o *wrap mode*, muy útiles si los datos que almacenaríamos fuera información referente a texturas reales.
4. Definir la estructura de tamaño `texSize`.

Para definir la textura utilizamos la función glTexImage2D, la cual recibe como primer parámetro el tipo de textura. El siguiente parámetro (definido como `0`) le dice a OpenGL que no utilice niveles de *mipmap*[4] para esta textura. Los dos siguientes serán el tamaño de la textura. El siguiente parámetro (definido como `0`) le indica al OpenGL que nuestra textura no contendrá bordes. El siguiente parámetro define el formato de la textura. El parámetro GL_FLOAT es relevante en el lado de la CPU, éste le dice a OpenGL que los datos que se le pasaran más tarde en las siguientes llamadas serán de punto flotante. El último parámetro (se pone a `0`) simplemente le dice a OpenGL que no queremos especificar ningún dato de la textura en este momento.

[4] Texturizado de Multi-Nivel. El cual establece utilizar diferentes niveles mipmap o tamaños de textura, dependiendo de cuánto un pixel de la textura se aprecia en la distancia.

Podría ser una buena analogía pensar en esta función como el equivalente de una asignación por `malloc ()` o algo muy similar.

7.3. Mapeo 1:1 de los índices del vector a las coordenadas de la textura

Los datos almacenados en las texturas serán actualizados mediante operaciones de renderizado. Para poder controlar exactamente qué elementos de la textura se procesan o acceden, necesitaremos seleccionar una proyección que mapee desde el mundo 3D a la pantalla 2D mediante un mapeo 1:1 entre los pixeles (lo que queremos renderizar) y los téxels (los datos a los que queremos acceder). La clave en este caso es seleccionar una proyección ortogonal[5] y un viewport[6] adecuado que nos permita el mapeo 1:1 entre las coordenadas de geometría (utilizadas en el renderizado) y las coordenadas de textura (utilizadas para la salida de datos). El mapeo se basa en el único valor del que disponemos, el tamaño (en cada dimensión) que nosotros hemos asignado a la textura. El siguiente código se puede añadir a la función `initFBO()` de tal forma que nos quede de la siguiente manera:

```
// Init of screen frame buffer
void initFBO(int texSize) {
          // Create FBO (off-screen framebuffer)
          glGenFramebuffersEXT(1, &fb);
          // Bind offscreen buffer
          glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
          // To map matrix data to texture
          // viewport for 1:1 pixel=texel=geometry mapping
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          gluOrtho2D(0.0, texSize, 0.0, texSize);
          glMatrixMode(GL_MODELVIEW);
          glLoadIdentity();
          glViewport(0, 0, texSize, texSize);
}
```

[5] Proyección ortogonal es aquella cuyas rectas proyectantes auxiliares son perpendiculares al plano de proyección, estableciéndose una relación entre todos los puntos del elemento proyectante con los proyectados.

[6] En gráficos 3D, el viewport se refiere al rectángulo 2D usando para proyectar la escena 3D en la posición de una cámara virtual ubicada en la escena.

7.4. Almacenamiento de los resultados de las operaciones en texturas

Para aumentar el rendimiento de nuestra aplicación utilizaremos texturas no solo como búferes de entradas, sino también como búferes de salida. Utilizando la extensión `framebuffer_object` podemos *renderizar directamente a una textura*.

En nuestra implementación necesitaremos tres texturas: Dos texturas de sólo lectura para el vector **x**, otra para el vector **y**, y una tercera textura de solo escritura que va a contener el resultado del cálculo, de tal forma que la operación a ejecutar será:

$$result = y + alpha\ x$$

Para utilizar una textura como un buffer de renderizado nosotros debemos añadir la textura al FBO que ha sido cargado con anterioridad, mediante el siguiente código:

```
GLuint resultTexID; // Result write only
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT, // Write only
                          GL_TEXTURE_RECTANGLE_ARB,
                          resultTexID,
                          0);
```

El primer parámetro es obligatorio. El segundo parámetro define el punto de inclusión (hasta cuatro texturas diferentes pueden añadirse por FBO, dependiendo del hardware y puede ser consultado con `GL_MAX_COLOR_ATTACHMENTS_EXT`). El tercer y cuarto parámetro, definen el tipo de textura y su identificado, respectivamente. El último parámetro (definido como `0`) establece que no utilizaremos *mipmapping* para la textura donde almacenaremos los resultados.

7.5. Paso de la información desde los vectores de la CPU a las texturas de la GPU

En nuestro caso los vectores `dataX` y `dataY` que hacen referencia a matrices rectangulares contienen los datos a ser procesados que deben ser enviados a una textura, con el objetivo de que la GPU pueda procesarlos. Notamos claramente las siguientes limitaciones:

1. El tamaño mínimo de las matrices que podemos procesar es de 2x2.
2. El tamaño máximo de las matrices está definido mediante el tamaño máximo de una textura soportada por nuestro hardware gráfico.
3. El tamaño de la matriz en cada dimensión debe ser múltiplo de dos.

Para la transferencia de los datos a las texturas debemos planificar la transferencia de datos mediante una llamada de OpenGL. Como es de suponer, es esencial que el arreglo pasado a la función como parámetro esté correctamente dimensionado.

Como hemos utilizado `GL_FLOAT`, los datos tienen que ser un puntero a un arreglo de flotantes, tal como se definió previamente. Hay que tener en cuenta que no tenemos ningún control de cuando en realidad los datos se transfieren a la memoria gráfica, esto es completamente competencia del controlador de la tarjeta, sin embargo podemos estar seguros de la transferencia de información fue completada una vez que la llamada de OpenGL ha retornado, así mismo OpenGL asegura que los datos estarán disponibles cuando intentemos acceder a la información almacenada en la textura.

Al utilizar como hardware gráfico una tarjeta con chip NVidia, resulta conveniente utilizar el código definido en la función `transferDataToTexture` ya tiene la ventaja de ser acelerado por hardware. La función `transferDataToTexture` transfiere la información (`data`) a una textura de un tamaño específico (`texSize`) definida por su identificador (`texID`).

```
// Transfers data to texture.
void transferDataToTexture(float* data, GLuint texID, const int texSize) {
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB, texID);
    glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,
                            0,
                            0,
                            0,
                            texSize,
                            texSize,
                            GL_RGBA,
                            GL_FLOAT,
                            data);
}
```

Los tres ceros pasados como parámetros definen el *offset* y el nivel *mipmap*. Al no usar *mipmaps* y transferir los vectores de forma completa dentro de la textura con su tamaño correspondiente, debemos definirlos como cero a cada uno de ellos.

7.6. Paso de la información de las texturas de la GPU a los vectores de la CPU

Si la textura a ser leída se encuentra asignada a un FBO, que es nuestro caso, resulta muy sencillo leer dicha información mediante una redirección de punteros. La función

`transferTextureToData` transfiere la información de una textura de un tamaño específico (`texSize`) que reside en el FBO, a un arreglo que recibe como parámetro de entrada (`data`).

```
// Transfers data from currently texture, and stores it in given array.
void transferTextureToData(float* data, const int texSize) {
    glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    glReadPixels(0, 0, texSize, texSize,GL_RGBA,GL_FLOAT,data);
}
```

8. Kernel de procesamiento

Aquí discutiremos la diferencia fundamental entre el modelo de cálculo entre las GPUs y CPUs, prestando especial atención en aquellas referentes a la algoritmia y metodología que debemos aplicar en cada caso.

Para poder resolver nuestro problema de calcular: **y** = **y** + *alpha* * **x** en un CPU, utilizaríamos un bucle que recorra todos los elementos de los vectores, de la siguiente forma:

```
for (int i=0; i<N; i++)
    dataY[i] = dataY[i] + alpha * dataX[i];
```

Dos niveles de cálculo son activados al mismo tiempo: uno es el contador del bucle que tiene que ser incrementado y comparado con la longitud de los vectores y otro es el cálculo que deseamos realizar dentro del bucle, en nuestro caso una multiplicación y una suma más los accesos a los vectores. Notamos claramente como no existe dependencia de datos dentro de las operaciones que estamos realizando. Si dispusiéramos de un procesador vectorial que fuera capaz de hacer operaciones sobre la longitud total del vector N, no necesitaríamos de ningún bucle. A éste tipo de paradigma se le denomina SIMD[7].

La idea central del cálculo en la GPU para el problema que queremos abordar debe ser claro: separar el bucle exterior de los cálculos internos. Los cálculos que hacemos dentro del bucle se extraen en un *kernel computacional* de la forma: $resultado\ [i]\ =\ y\ [i]\ +\ alpha\ \ x\ [i]$.

Los índices de los datos de salida son los mismos de los datos de entrada con lo que hay una independencia total en cada componente de los vectores. Por ello cada una de las unidades

[7] Single instruction, multiple data

funcionales de la GPU podrá ser utilizada en paralelo, y a modo de analogía, como si fueran un procesador vectorial sobre cada una de las componentes de la textura. La parte programable del GPU que queremos usar con nuestros cálculos es la llamada *fragment pipeline* o *segmentos de programa*, la cual se compone de un conjunto de unidades de procesamiento paralelo. Sin embargo el hardware y el controlador, que sincroniza cada dato en los diferentes pipelines disponibles no son programables. Es por esto que desde el punto de vista conceptual, todo el trabajo que se realiza sobre los datos es ejecutado independientemente y sin influencia de algún fragmento libre de un pipeline.

En nuestro caso haremos que el *fragment pipeline* se comporte como un procesador vectorial de igual tamaño que nuestras texturas. Aunque internamente el cálculo se divide entre los pipelines de procesamiento disponibles, no podríamos controlar el orden en que los fragmentos son procesados, todo lo que sabemos es las coordenadas en la textura en la cual el resultado de la operación será colocado. De esta manera podemos asumir que todo el trabajo es hecho en paralelo sin ninguna interdependencia de datos.

Los kernels se traducen en Shaders en la GPU. Un shader, es un conjunto de instrucciones gráficas destinadas para el acelerador gráfico, estas instrucciones tienen el objetivo de dan el aspecto final de un objeto. Los shaders determinan materiales, efectos, color, luz, sombra, etc.

En nuestro caso utilizaremos shaders para definir las operaciones a ser ejecutadas sobre las texturas, de tal forma que debemos definir un shader apropiado e incluirlo en nuestra implementación. Para escribir el shader utilizaremos el lenguaje de shaders de OpenGL.

8.1. Utilización de Shaders con el lenguaje de Shaders OpenGL

OpenGL Shading Language (GLSL) el cual forma parte del núcleo de OpenGL permite definir shaders. Tres extensiones OpenGL (`ARB_shader_objects`, `ARB_vertex_shader` y `ARB_fragment_shader`) definen la API y sus especificaciones definen el lenguaje en sí.

Definimos una serie de variables globales para el programa, el shader y los manejadores (handles) para acceder a los datos desde el shader. Los dos primeros son simplemente contenedores gestionados por OpenGL; un programa puede consistir en un `vertex` y un `fragment shader`, ambos subtipos pueden constar de varios fuentes de shaders, un shader puede a su vez ser parte de varios programas, etc.

```
// GLSL Vars to make computation
GLuint glslProgram;
GLuint fragmentShader;
GLint yParam, xParam, alphaParam;
```

El shader para el Kernel de nuestro ejemplo será el siguiente:

```
// shader para datos luminance
// y texture rectangles
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect textureY;
uniform sampler2DRect textureX;
uniform float alpha;

void main(void) {
    float y = texture2DRect(
            textureY,
            gl_TexCoord[0].st).x;
    float x = texture2DRect(
            textureX,
            gl_TexCoord[0].st).x;
    gl_FragColor.x =  y + alpha*x;
}
```

La inicialización del GLSL la podemos encapsular en la función `initGLSL` de la siguiente manera:

```
// Sets up the GLSL runtime and creates shader.
void initGLSL(void) {
         // create program object
         glslProgram = glCreateProgram();
         // create shader object (fragment shader)
         fragmentShader = glCreateShader(GL_FRAGMENT_SHADER_ARB);
         // set source for shader
         //Shader source
         const GLchar* source = \
         "#extension GL_ARB_texture_rectangle : enable\n" \
         "uniform sampler2DRect textureY;" \
         "uniform sampler2DRect textureX;" \
         "uniform float alpha;" \
         "void main(void) { " \
         "  vec4 y = texture2DRect(textureY, gl_TexCoord[0].st);" \
         "  vec4 x = texture2DRect(textureX, gl_TexCoord[0].st);" \
```

```
        "  gl_FragColor = y + alpha*x;"\
        "}";
        glShaderSource(fragmentShader,1,&source,NULL);
        // compile shader
        glCompileShader(fragmentShader);
        // check for errors
        printShaderInfoLog(fragmentShader);
        // attach shader to program
        glAttachShader (glslProgram, fragmentShader);
        // link into full program, use fixed function vertex pipeline
        glLinkProgram(glslProgram);
        // Get location of the texture samplers for future use
        yParam = glGetUniformLocation(glslProgram, "textureY");
        xParam = glGetUniformLocation(glslProgram, "textureX");
        alphaParam = glGetUniformLocation(glslProgram, "alpha");
}
```

9. Cómputo

Para ejecutar el cálculo debemos realizar los siguientes pasos:

1. Preparar el Kernel computacional.
2. Los arreglos de entrada y salida deben ser asignados usando los shader.
3. Renderizar una geometría adecuada para realizar el cálculo.

9.1. Preparar el Kernel computacional

Para activar el Kernel utilizando GLSL lo único que necesitamos es instalar el programa como parte del pipeline de renderizado, lo cual requiere de sólo una línea de código:

```
glUseProgram(glslProgram);
```

9.2. Preparar los arreglos (texturas) de entrada

Lo que debemos hacer es enlazar nuestras texturas a diferentes unidades de texturas y pasar estas unidades a los parámetros de entrada del shader.

```
// enable texture y_old (read-only)
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,yTexID);
glUniform1i(yParam,0); // texunit 0

// enable texture x (read-only)
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,xTexID);
glUniform1i(xParam, 1); // texunit 1

// enable scalar alpha (same)
glUniform1f(alphaParam,alpha);
```

9.3. Preparar los arreglos (texturas) de salida

Definir el arreglo de salida es básicamente la misma operación que explicamos para transferir datos a una textura asignada a un FBO. Lo único que necesitamos es redirigir la salida, si no lo hemos hecho todavía, a una textura de nuestro FBO, y luego utilizar una llamada OpenGL para definirla como textura final en donde se realizará el renderizado. El código sería el siguiente:

```
// attach two textures to FBO
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT, // Write only
                          GL_TEXTURE_RECTANGLE_ARB,
                          resultTexID,
                          0);
// set render destination
glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);
```

9.4. Ejecutar el cálculo

Ejecutar el cálculo no es otra cosa que renderizar una geometría adecuada que asegure que nuestro *fragment shader* se ejecuta sobre toda la región que contiene los datos a ser procesados (textura), considerando el mapeo 1:1 definido anteriormente, entre los pixel objetivo, las coordenadas de la textura y la geometría que vamos a dibujar. Debemos asegurar que cada elemento es transformado en un único *fragmento*, de tal forma que un *fragment shader* se ejecute sobre cada región.

Usando **texture rectangles** podremos conseguir lo previamente descrito con las siguientes líneas de código:

```
glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0);
        glVertex2f(0.0, 0.0);
        glTexCoord2f(texSize, 0.0);
        glVertex2f(texSize, 0.0);
        glTexCoord2f(texSize, texSize);
        glVertex2f(texSize, texSize);
        glTexCoord2f(0.0, texSize);
        glVertex2f(0.0, texSize);
glEnd();
```

10. CONCLUSIONES

Una vez culminado este trabajo podemos obtener las siguientes conclusiones:

- La programación de un operador simple en una GPU no es una tarea trivial. Aunque la dificultad se ve recompensada por grandes potencias de cálculo a bajo coste.
- No todos los algoritmos pueden ser implementados para que puedan hacer uso del poder de cómputo de en una GPU. Esto depende mucho de la naturaleza del problema, así como de la independencia de datos que posea.
- Problemas de tipo procesamiento vectorial, encajan muy bien en la filosofía de diseño de las GPU, altamente paralelizadas y optimizadas para el procesamiento vectorial.
- La portabilidad del código es un serio problema, aunque existen esfuerzos de estandarizar la programación en GPU mediante la creación de librerías, es muy probable que una aplicación funcione solamente para un hardware determinado, como lo es en nuestro caso.

A. Anexos

A1. Descripción técnica de la tarjetas NVIDIA Serie 8M

NVIDIA® Unified Architecture

- Unified shader architecture
- GigaThread™ technology
- Full support for Microsoft® DirectX® 10
  - Geometry shaders
  - Geometry instancing
  - Streamed output
  - Shader Model 4.0
- Full 128-bit floating point precision through the entire rendering pipeline

NVIDIA Lumenex™ Engine

- 16x full screen anti-aliasing
- Transparent multisampling and transparent supersampling
- 16x angle independent anisotropic filtering
- 128-bit floating point high dynamic-range (HDR) lighting with anti-aliasing
- 32-bit per component floating point texture filtering and blending
- Advanced lossless compression algorithms for color, texture, and z-data
- Support for normal map compression
- Z-cull
- Early-Z

NVIDIA SLI® Technology[1]

Patented hardware and software technology allows two GeForce-based graphics cards to run in parallel to scale performance and enhance image quality on today's top titles.

NVIDIA PureVideo® HD Technology [2]

- Dedicated on-chip video processor
- High-definition H.264, VC-1, MPEG2 and WMV9 decode acceleration
- Advanced spatial-temporal de-interlacing
- HDCP capable [3]

- Noise Reduction
- Edge Enhancement
- Bad Edit Correction
- Inverse telecine (2:2 and 3:2 pull-down correction)
- High-quality scaling
- Video color correction
- Microsoft® Video Mixing Renderer (VMR) support

Advanced Display Functionality

- Dual-link DVI outputs for digital flat panel display resolutions up to 2560x1600
- Dual integrated 400MHz RAMDACs for analog display resolutions up to and including 2048x1536 at 85Hz
- Integrated HDTV encoder provides analog TV-output (Component/Composite/S-Video) up to 1080i/1080p resolution
- NVIDIA nView® multi-display technology capability
- 10-bit display processing

Built for Microsoft® Windows Vista™

- Full DirectX 10 support (see above similar comment)
- Dedicated graphics processor powers the new Windows Vista Aero 3D user interface
- VMR-based video architecture

High Speed Interfaces

- Designed for PCI Express® x16
- Designed for high-speed GDDR3 & cost-competitive DDR2 memory

Operating Systems

- Built for Microsoft Windows Vista
- Windows XP/Windows XP 64
- Linux

API Support

Complete DirectX support, including Microsoft DirectX 10 Shader Model 4.0

Full OpenGL® support, including OpenGL 2.1

1 - SLI technology available only on select GeForce 8600M notebook GPUs

2 - Feature requires supported video hardware and software. Features may vary by product.

3- Requires other compatible components that are also HDCP capable.

A2. Código fuente

```
// Defines
#define SIZE_PARAM 's'
// Error codes
#define ERROR_CG  -1
#define ERROR_GLEW  -2
#define ERROR_TEXTURE  -3
#define ERROR_BINDFBO  -4
#define ERROR_FBOTEXTURE  -5
#define ERROR_PARAMS  -6
#include <stdio.h>
#include <sstream>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <iostream>
#include <unistd.h>
#include <boost/lexical_cast.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include "config.hpp"

#include <GL/glew.h>
#include <GL/glut.h>

// Data to store matrix information
float* dataY;
float* dataX;
float* result;
float alpha;

// Global Variables
int N; //dimention of cuadratic matrix

GLuint fb; // FOB Identifier
// Texture identifiers
// Result = Y + alpha*X SAXPY Sum AX Plus Y
GLuint yTexID; // Y read only
GLuint resultTexID; // Result write only
GLuint xTexID; //  X
GLuint aTexID; // alpha

// GLSL Vars to make computation
GLuint glslProgram;
GLuint fragmentShader;
GLint yParam, xParam, alphaParam;

// Checks for OpenGL errors.
void checkGLErrors (const char *label) {
    GLenum errCode;
    const GLubyte *errStr;

    if ((errCode = glGetError()) != GL_NO_ERROR) {
        errStr = gluErrorString(errCode);
        std::cerr << "OpenGL ERROR:" << (char*)errStr << "(Label: " << label << ")" << std::endl;
    }
}


// Error checking for GLSL
void printProgramInfoLog(GLuint obj) {
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;
    glGetProgramiv(obj, GL_INFO_LOG_LENGTH, &infologLength);
    if (infologLength > 1) {
        infoLog = (char *)malloc(infologLength);
        glGetProgramInfoLog(obj, infologLength, &charsWritten, infoLog);
        std::cout << infoLog << std::endl;
        free(infoLog);
    }
}
```

```
// Init glew
void initGLEW(void) {
        int err = glewInit();

        if (GLEW_OK != err) {
                std::cout << ((char*)glewGetErrorString(err));
                exit(ERROR_GLEW);
        }
}


// Checks framebuffer status.
bool checkFramebufferStatus() {
    GLenum status;
    status = (GLenum) glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
    switch(status) {
        case GL_FRAMEBUFFER_COMPLETE_EXT:
            return true;
        case GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT_EXT:
            printf("Framebuffer incomplete, incomplete attachment\n");
            return false;
        case GL_FRAMEBUFFER_UNSUPPORTED_EXT:
            printf("Unsupported framebuffer format\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT_EXT:
            printf("Framebuffer incomplete, missing attachment\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT:
            printf("Framebuffer incomplete, attached images must have same dimensions\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_FORMATS_EXT:
            printf("Framebuffer incomplete, attached images must have same format\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT:
            printf("Framebuffer incomplete, missing draw buffer\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT:
            printf("Framebuffer incomplete, missing read buffer\n");
            return false;
    }
    return false;
}

// Init of screen frame buffer
void initFBO(int texSize) {
        // Create FBO (off-screen framebuffer)
        glGenFramebuffersEXT(1, &fb);
        // Bind offscreen buffer
        glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
        // To map matrix data to texture
        // viewport for 1:1 pixel=texel=geometry mapping
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, texSize, 0.0, texSize);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glViewport(0, 0, texSize, texSize);
}

// Sets up a floating point texture with NEAREST filtering.
void setupTextureTarget(const GLuint texID, const int texSize) {
        // make active and bind
        glBindTexture(GL_TEXTURE_RECTANGLE_ARB,texID);
        // turn off filtering and wrap modes
        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);
        // define texture with floating point format
        glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,
                        0,
                        GL_RGBA32F_ARB,
                        texSize,
                        texSize,
                        0,
                        GL_RGBA,
                        GL_FLOAT,
                        0);
        // check if that worked
        if (glGetError() != GL_NO_ERROR) {
                std::cerr << "glTexImage2D():\t\t\t [FAIL]" << std::endl;
                exit (ERROR_TEXTURE);
        }
}
```

```
// Init glut
void initGLUT(int argc, char **argv) {
    glutInit ( &argc, argv );
    glutCreateWindow("SAXPY OVER GPU");
}

// Transfers data to texture.
// Valid only on nvidia hardware
void transferDataToTexture(float* data, GLuint texID, const int texSize) {
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB, texID);
    glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,
                            0,
                            0,
                            0,
                            texSize,
                            texSize,
                            GL_RGBA,
                            GL_FLOAT,
                            data);
}

// Print shader info log
void printShaderInfoLog(GLuint obj) {
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;
    glGetShaderiv(obj, GL_INFO_LOG_LENGTH, &infologLength);

    if (infologLength > 1) {
        infoLog = (char *)malloc(infologLength);
        glGetShaderInfoLog(obj, infologLength, &charsWritten, infoLog);
        std::cout << infoLog << std::endl;
        free(infoLog);
    }
}

// Sets up the GLSL runtime and creates shader.
void initGLSL(void) {
         // create program object
         glslProgram = glCreateProgram();
         // create shader object (fragment shader)
         fragmentShader = glCreateShader(GL_FRAGMENT_SHADER_ARB);
         // set source for shader

         //Shader source
         const GLchar* source = \
         "#extension GL_ARB_texture_rectangle : enable\n" \
         "uniform sampler2DRect textureY;" \
         "uniform sampler2DRect textureX;" \
         "uniform float alpha;" \
         "void main(void) { " \
         "  vec4 y = texture2DRect(textureY, gl_TexCoord[0].st);" \
         "  vec4 x = texture2DRect(textureX, gl_TexCoord[0].st);" \
         "  gl_FragColor = y + alpha*x;"\
         "}";

         glShaderSource(fragmentShader,1,&source,NULL);
         // compile shader
         glCompileShader(fragmentShader);
         // check for errors
         printShaderInfoLog(fragmentShader);
         // attach shader to program
         glAttachShader (glslProgram, fragmentShader);
         // link into full program, use fixed function vertex pipeline
         glLinkProgram(glslProgram);
         // check for errors
         printProgramInfoLog(glslProgram);
         checkGLErrors("render(2)");
         // Get location of the texture samplers for future use
         yParam = glGetUniformLocation(glslProgram, "textureY");
         xParam = glGetUniformLocation(glslProgram, "textureX");
         alphaParam = glGetUniformLocation(glslProgram, "alpha");
}


void showVector (const float *p, const int N) {
         for (int i=0; i<N; i++)
                  std::cout << p[i] << " ";
         std::cout << std::endl;
}
```

```
// Performs the actual calculation.
void performGPUComputation(int texSize) {
        //For time calculations
        boost::posix_time::ptime start_time;
        boost::posix_time::ptime end_time;
        boost::posix_time::time_duration used_time;

        std::cout << "1. Setting program for calculation..." << std::endl;
        // enable GLSL program
        glUseProgram(glslProgram);

        std::cout << "2. Setting X and Y input arrays read only (textures)..." << std::endl;
        // enable texture y_old (read-only)
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_RECTANGLE_ARB,yTexID);
        glUniform1i(yParam,0); // texunit 0

        // enable texture x (read-only)
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_RECTANGLE_ARB,xTexID);
        glUniform1i(xParam, 1); // texunit 1

        // enable scalar alpha (same)
        glUniform1f(alphaParam,alpha);


        std::cout << "3. Setting output Result array  (textures)" << std::endl;
        // attach two textures to FBO
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                                  GL_COLOR_ATTACHMENT0_EXT, // Write only
                                  GL_TEXTURE_RECTANGLE_ARB,
                                  resultTexID,
                                  0);
        // check if that worked
        if (!checkFramebufferStatus()) {
                std::cerr << "Error ataching textures" << std::endl;
                exit (ERROR_FBOTEXTURE);
        }

        // set render destination
        glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);

        std::cout << "4. Perform computation" << std::endl;
        glPolygonMode(GL_FRONT,GL_FILL);

        start_time = boost::posix_time::microsec_clock::universal_time();

        // render with unnormalized texcoords
        glBegin(GL_QUADS);
                glTexCoord2f(0.0, 0.0);
                glVertex2f(0.0, 0.0);
                glTexCoord2f(texSize, 0.0);
                glVertex2f(texSize, 0.0);
                glTexCoord2f(texSize, texSize);
                glVertex2f(texSize, texSize);
                glTexCoord2f(0.0, texSize);
                glVertex2f(0.0, texSize);
        glEnd();

        end_time = boost::posix_time::microsec_clock::universal_time();
        glFinish();


        //Get used time for compute
        used_time = end_time - start_time;
        // Calc mflops
        std::cout << "Time used with GPU compute = " << used_time.total_milliseconds() << " milliseconds"
<<std::endl;

        checkFramebufferStatus();
        checkGLErrors("render()");
}
```

```
// Creates textures
void createTextures(const int texSize) {
        // create textures
        // y gets two textures, alternatingly read-only and write-only,
        // x is just read-only
        glGenTextures (1, &yTexID); //Create y, write
        glGenTextures (1, &resultTexID); //Create result, read
        glGenTextures (1, &xTexID); //Create x texture

        // Set up textures and write data
        setupTextureTarget(yTexID,texSize);
        transferDataToTexture(dataY,yTexID,texSize);

        setupTextureTarget(resultTexID,texSize);
        transferDataToTexture(dataY,resultTexID,texSize);

        setupTextureTarget(xTexID,texSize);
        transferDataToTexture(dataX,xTexID,texSize);

        // set texenv mode from modulate (the default) to replace
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        // check if something went completely wrong
        checkGLErrors ("createFBOandTextures()");
        std::cout << "Textures for X, Y were sucefully created and data stored in each one" << std::endl;
}

// Transfers data from currently texture, and stores it in given array.
void transferTextureToData(float* data, const int texSize) {
    glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    glReadPixels(0, 0, texSize, texSize,GL_RGBA,GL_FLOAT,data);
}


// Analyze results
void analyzeResults(const int texture_size){
        //For time calculations
        boost::posix_time::ptime start_time;
        boost::posix_time::ptime end_time;
        boost::posix_time::time_duration used_time;

        //Geting result from GPU
        std::cout << "Getting results from GPU" << std::endl;
        transferTextureToData(result,texture_size);

        //Result matrix
        float* resultCPU = (float*)malloc((N*N)*sizeof(float));

        //Perform matrix computation using CPU
        std::cout << "Performing sapxy operations on CPU" << std::endl;
        start_time = boost::posix_time::microsec_clock::universal_time();
        for(int i=0;i<N*N;i++){
                resultCPU[i]=dataY[i]+ alpha*dataX[i];
        }
        end_time = boost::posix_time::microsec_clock::universal_time();
        used_time = end_time - start_time;
        std::cout << "Time used with CPU compute = " << used_time.total_milliseconds() << " milliseconds"
<<std::endl;

        //Compare results
        std::cout << std::endl << "Comparing results.." << std::endl;
        double avgError;
        double maxError=0.0;
        float equals=true;
        for (int i=0; i<N*N; i++) {
                double diff = fabs(result[i]-resultCPU[i]);
                if(diff > maxError)
                        maxError = diff;
                if(result[i] != resultCPU[i])
                        equals = false;
                avgError += diff;
        }
        avgError /= (double)N*N;
        if(equals)
                std::cout << "Both matrices are equals!!" << std::endl;
        else
                std::cerr << "Error, Matrices are not equals !!!" << std::endl;
        std::cout << "Error average = " << avgError << std::endl;
        std::cout << "Error max = " << maxError << std::endl;
        free(resultCPU);
}
```

```
// Main function
int main(int argc,char** argv )
{
        int maxtexsize; // Max size of texture buffer in GPU
        int texture_size=0; // Texture size

        // Getting the size of the matrix from command line
        std::stringstream optstring;
        optstring << SIZE_PARAM << "," ;
        int option_char;
        bool par_setted=false;
        while ((option_char = getopt(argc,argv,optstring.str().c_str())) != EOF){
                switch (option_char)
                {
                        case SIZE_PARAM:
                                par_setted = true;
                                N=boost::lexical_cast<int>(argv[optind]);
                        break;
                }
        }
        //Verify if path is setted
        if (!par_setted){
                std::cerr<<"mult -s [MATRIX SIZE]" << std::endl;
                exit(ERROR_PARAMS);
        }
        // Verify if N is 2 divisible
        else{
                if(N<2 || N%2!=0){
                        std::cout << "Min matrix size is 2, and should be divisible for 2" << std::endl;
                        exit(ERROR_CG);
                }
        }

        std::cout << "Matrix size readed from commnad line N: " << N << std::endl;

        // Matrix size to make saxpy operation
        std::cout << "Creating X,Y and result matrices (" << N << "x" << N << ") for saxpy operation " <<
std::endl;
        // Create matrix float (32 bits) NxN dimensions
        dataY = (float*)malloc((N*N)*sizeof(float));
        dataX = (float*)malloc((N*N)*sizeof(float));
        result = (float*)malloc((N*N)*sizeof(float));

        // Fill matrix X and Y
        std::cout << "Poblating X and Y matrix" << std::endl;
       for (int i=0; i<N*N; i++){
                dataX[i] = i+1.0;
                dataY[i] = i+2.0;
        }
        alpha=2.0;

        std::cout << "Starting openGL" << std::endl;
        initGLUT(argc,argv);
        std::cout << "Starting GLEW" << std::endl;
        initGLEW();

        // Getting max texture size
        glGetIntegerv(GL_MAX_TEXTURE_SIZE,&maxtexsize);
        std::cout  <<  "One  of  three  dimension  of  texture:  GL_MAX_TEXTURE_SIZE:  "  <<  maxtexsize  <<
std::endl;

        // Define texture size based in matrix size
        // Will to maximize the texture size using 4-tuple in each texture position
        // NxN = 4 x texture_size x texture_size, so texture_size = N/2
        texture_size = N/2;
        std::cout << "Using texture size of " << texture_size << std::endl;

        // Init off screen frame buffer
        std::cout << "InitFBO (Off screen frame buffer) " << std::endl;
        initFBO(texture_size);

        // Definig texture target, texture format and internal format
        std::cout << "Creating textures (" << texture_size << "x" << texture_size << "x4)" << " and set
turn off filtering and proper wrap mode" << std::endl;
        //Creating x texture
        std::cout << "Creating X, Y textures" << std::endl;
        createTextures(texture_size);

        // Init shader runtime
        std::cout << "Init shader runtime" << std::endl;
        initGLSL();
        std::cout << "Starting GPU computation..." << std::endl;
        performGPUComputation(texture_size);

        // Analyze results
        std::cout << "Starting with results analyzing..." << std::endl;
        analyzeResults(texture_size);
```

```
        // Free memory
        free(dataX);
        free(dataY);
        free(result);
        glDeleteFramebuffersEXT (1,&fb);
        glDeleteTextures (1,&xTexID);
        glDeleteTextures (1,&yTexID);
        glDeleteTextures (1,&resultTexID);

        return 0;
}
```