UNIVERSITAT D'ALACANT
Facultat de Ciències
Facultad de Ciencias

FACULTAD DE CIENCIAS

MÁSTER UNIVERSITARIO

EN CÁLCULO Y MODELIZACIÓN CIENTÍFICA

TRABAJO FIN DE MÁSTER

CURSO ACADÉMICO [2025-2026]

TÍTULO:

Computación de alto rendimiento mediante programación en GPUs

AUTOR:

Miguel Maciá Titos

Resumen

En este TFM hablaremos de las motivaciones detrás del desarrollo de la computación en GPUs, centrando nuestra atención en el lenguaje de programación CUDA de NVIDIA.

Proporcionaremos una guía detallada de como crear código paralelo en este lenguaje, ahondando en las diferentes herramientas que nos proporciona para optimizar nuestros programas, todo ello reforzado por ejemplos que ilustren de manera clara como utilizar estas características del lenguaje en la práctica. Implementaremos programas conocidos como la multiplicación de matrices o el Juego de la Vida de Conway en la GPU.

Finalizaremos con la implementación del filtro bilateral mediante CUDA, comparando el programa secuencial original con diversas alternativas en la GPU que incorporan las técnicas expuestas anteriormente.

Palabras Clave: Computación paralela, Programación en paralelo, Programación en C, Programación CUDA, GPU NVIDIA, Procesamiento de imágenes

Abstract

In this TFM we will discuss the motivations behind the development of GPU computing, focusing our attention on NVIDIA's CUDA programming language.

We will provide a detailed guide on how to create parallel code using this programming language, delving into the different tools it provides for our program's optimization, all of it reinforced through the use of examples that clearly illustrate how to use these features of the language in practice. We will implement well-known programs such as matrix multiplication or Conway's Game of Life.

We will conclude with the implementation of the bilateral filter through CUDA, comparing the original sequential program with several alternatives on the GPU which incorporate the techniques previously shown.

Keywords: Parallel computing, Parallel programming, C programming, CUDA programming, NVIDIA GPU, Image processing

Índice

1. Introducción

La programación en paralelo es una de las herramientas fundamentales del programador actual y son muchos los que se limitan a estudiar este procedimiento sobre la CPU, ya sea una única CPU multicore o varias conectadas entre si. Sin embargo, en los últimos años las GPUs se han asentado como una alternativa poderosa con la posibilidad de alcanzar un grado de paralelización mucho mayor, aun presentando sus propias dificultades y limitaciones. En este trabajo veremos la motivación detras del uso de este tipo de hardware y estudiaremos como podemos utilizar el software CUDA de forma eficiente para crear programas que operen sobre una GPU, introduciendo ejemplos para ilustrar sus uso. En primer lugar, veamos la historia que precede a este nuevo paradigma.

2. Historia

Es indiscutible que la historia del software siempre ha ido de la mano del hardware disponible, simultáneamente son muchas veces las necesidades del software las que guían la evolución del hardware, no hay campo que mejor ejemplifique esta relación que la historia y evolución de las GPUs.

2.1. Procesadores de un solo núcleo

Para entender la motivación del desarrollo de las CPUs es necesario conocer los dispositivos que las preceden y sus limitaciones, comenzando con el más fundamental, los procesadores de un solo núcleo. Desde las 5000 operaciones por segundo del ENIAC en 1946 [1] a los 5.7 billones de cada núcleo del Intel Core Ultra 9 285k [2] es la velocidad la que dictamina la eficiencia de estos procesadores, limitados a realizar las tareas en orden secuencial es esta incapacidad de repartir la carga de trabajo la que llevó al desarrollo de una nueva clase de procesador, los procesadores multinúcleo.

2.2. Procesadores multinúcleo

Entrando en el mercado por primera vez en 2001 [3], los procesadores multinúcleo representaron una revolución, ayudando a solventar los problemas de latencia que emergían de emplear diferentes procesadores en paralelo. Hoy en día estos procesadores son el estándar, siendo predominantes las CPUs de entre 4-8 núcleos entre los usuarios promedio

[4] y existiendo procesadores de hasta 192 procesadores [5] destinados a la computación en la nube. Aun solventando muchos de nuestros problemas, existen aplicaciones para las que esta clase de procesadores no es suficiente. Limitados en su grado de paralelización por el escalado de costes y las dificultades en el control de temperaturas, existen aplicaciones para las que una nueva clase de dispositivo es necesario, las GPUs.

2.3. Las GPUs

Popularizados inicialmente por la aparición de sistemas operativos gráficos como Microsoft Windows, no fue hasta mediados de los años 90 que la demanda entre los consumidores de procesadores gráficos explotó, causado por la proliferación de juegos con gráficos 3D como DOOM y QUAKE así como la salida al mercado de aceleradores gráficos asequibles gracias a compañías como NVIDIA, ATI Technologies y 3dfx Interactive [10].

Fue la tarjeta GeForce 256 de NVIDIA la primera que permitió realizar computaciones de transformación (proyección de un entorno 3D a una interfaz 2D) e iluminación de forma completamente independiente, suponiendo un salto en las capacidades de los dispositivos al alcance de los usuarios y marcando el camino al presente, donde las GPUs se encargan independientemente de la mayoría de procesos gráficos.

En nuestro caso, nos es de particular interés la salida al mercado de la serie GeForce 3 de NVIDIA, dado que fue la primera en implementar el estándar DirectX8 de Windows, el cual requería que el hardware incluyera vértices programables y una fase programable de sombreado de píxeles, representando la primera vez que los programadores tenían control sobre las operaciones que realizaba la GPU.

2.4. Los inicios de la programación en GPUs

En sus inicios la programación en GPUs era un proceso complejo y confuso, esto se debía principalmente a que la única manera de interactuar con los dispositivos era a través de API gráficas, las cuales como su nombre indican estaban muy restringidas en los métodos que ofrecían, teniendo que disfrazar las computaciones deseadas como tareas de renderizado.

Estos primeros programas en GPU definían los datos como texturas y aplicaban programas de sombreado y renderizado para obtener un output deseado, de nuevo en forma de una textura. Un ejemplo de esta técnica es la transformación rápida de Fourier (FFT) como se ilustra en [6].

Aunque esta forma de implementar programas era factible, no era particularmente sencilla o cómoda, particularmente para los programadores que no estaban acostumbrados a trabajar con código de procesado de gráficos.

Todas estas dificultades motivaron a la creación de Brook para GPUs [7], una extensión de C que permitía utilizar la GPU mediante streams consiguiendo resultados comparables al código para GPUs creado manualmente. En 2004 el jefe de desarrollo de Brooks, Ian Buck, se unió a NVIDIA [8] y lideró el lanzamiento de una solución general para la computación en GPUs, CUDA.

2.5. CUDA

Publicado por primera vez en 2006 junto a la tarjeta GForce 8800 CUDA (Compute Unified Device Architecture) permite a los usuarios crear programas que funcionan en las GPUs de NVIDIA con un lenguaje prácticamente idéntico a C aunque en versiones más modernas implementa también las características de C++ y puede utilizarse desde lenguajes como Fortran, MATLAB y Python [9].

La introducción de CUDA supuso una revolución simplificando la programación en GPUs y abriendo sus posibilidades a un público mucho mayor, encontrando aplicaciones en todo tipo de campos e industrias. Pese a la existencia de otras alternativas en la actualidad como OpenGL (de código abierto), CUDA continua dominando este área de la programación recibiendo actualizaciones periódicas junto a la salida de nuevas y mejores tarjetas gráficas.

3. CUDA

Ahora que hemos visto la motivación e historia detrás de este lenguaje, el siguiente paso es entender como utilizarlo:

3.1. Nociones básicas

Lo primero que debemos tener en cuenta es que todo lo que funciona en C funciona en CUDA C, es decir, un programa como el del Código 1 funcionaría sin problemas en CUDA C.

```
1    int add(int a, int b){
2    int c;
```

```
3     c = a+b;
4     return c;
5     }
```

Código 1: *Función suma de 2 enteros en C*

Veamos el comando fundamental de este lenguaje, que nos permitirá lanzar los procesos en paralelo en nuestra GPU:

funcion<<<bloques,hilos>>>(param1,...)

Desglosemos este comando:

- **funcion**: La función que queremos ejecutar en todos los hilos de forma paralela
- **bloques**: número de bloques paralelos que queremos lanzar, puede ser un número, un array 2D o un array 3D
- **hilos**: número de hilos paralelos que queremos lanzar dentro de cada bloque, puede ser un número, un array 2D o un array 3D
- **param1,...**: parámetros que queremos pasar a nuestra función

Más adelante veremos la diferencia entre bloques e hilos pero primero veamos un ejemplo de este comando en acción en el Código 2

```
1  #include <stdio.h>
2
3  __global__ void example(void){
4  }
5
6  int main(void){
7
8  example<<<2,1>>>();
9
10 return 0;
11 }
```

Código 2: *Lanzar 2 bloques en la GPU*

En este ejemplo lanzamos una función vacía en 2 bloques dentro de la GPU y dentro de cada uno de estos bloques lanzamos un único hilo. Podemos observar también que para poder lanzar una función desde nuestra GPU la debemos definir con la palabra clave **__global__**, alternativamente podemos hacerlo con la palabra clave **__device__** pero en este caso solo podremos llamarla dentro de la GPU.

Ahora que sabemos llamar funciones dentro de nuestra GPU el siguiente paso es poder utilizar variables dentro de ellas, además, nos interesa poder mandar información a la GPU desde la CPU y luego recibir los resultados. Para ello, utilizamos la siguientes funciones:

- **cudaMalloc((void**)&variable, tamañoVariable)**: Similar a la función malloc() de C nos permite reservar memoria en la GPU, teniendo que especificar el tamaño de la variable en bytes.
- **cudaMemcpy(variableReceptora, &variableEmisora, tamañoVariable, dirección)**: Permite mandar la información de una variable a otra, la dirección indica si este paso es de CPU a GPU (cudaMemcpyHostToDevice) o de GPU a CPU (cudaMemcpyDeviceToHost)
- **cudaFree(variableGPU)**: Similar a free() en C, libera la porción de memoria de la GPU asignada por cudaMalloc.

Veamos ahora un ejemplo en el Código 3.

```
#include <stdio.h>

//Definimos la funcion suma
__global__ void suma(int *a, int *b, int *c){

*c = *a + *b;
}

int main(){

int a, b, c;
int *gpu_a, *gpu_b, *gpu_c;

printf("Indica el valor de a: ");
scanf("%d", &a);
printf("Indica el valor de b: ");
scanf("%d", &b);

//Reservamos espacio para nuestras variables
cudaMalloc((void**)&gpu_a, sizeof(int));
cudaMalloc((void**)&gpu_b, sizeof(int));
cudaMalloc((void**)&gpu_c, sizeof(int));

//Enviamos el contenido de a y b a la GPU
```

```
25 cudaMemcpy(gpu_a, &a, sizeof(int), cudaMemcpyHostToDevice);
26 cudaMemcpy(gpu_b, &b, sizeof(int), cudaMemcpyHostToDevice);
27
28 //Llamamos a la funcion suma en la GPU
29 suma<<<1,1>>>(gpu_a, gpu_b, gpu_c);
30
31 //Extraemos el contenido de la variable gpu_c a una variable en la CPU
32 cudaMemcpy(&c, gpu_c, sizeof(int), cudaMemcpyDeviceToHost);
33 printf("%d + %d = %d \n", a, b, c);
34
35 //Liberamos las variables en la GPU
36 cudaFree(gpu_a);
37 cudaFree(gpu_b);
38 cudaFree(gpu_c);
39
40 return 0;
41 }
```

Código 3: *Función suma de 2 enteros en GPU*

Podemos observar en el ejemplo que debemos definir las variables de la GPU antes de asignarles la memoria, al igual que en C al usar malloc().

Finalmente, dado que el propósito de ejecutar funciones sobre la GPU es poder paralelizarlas, no nos interesa que todos los bloques/hilos realicen el mismo trabajo, por suerte, podemos utilizar los siguientes identificadores para indicar a cada uno lo que debe hacer:

- **blockIdx.**{x,y,z}
- **threadIdx.**{x,y,z}

Para ver estos identificadores en acción y cerrar esta sección con un ejemplo más elaborado, observemos como podemos utilizar la GPU para multiplicar matrices cuadradas de forma paralela en el Código 4, el código completo esta disponible en Github (enlace).

```
1
2 __global__ void sqr_matrix_mult(double *matrix_a, double *matrix_b,
     double *matrix_result, int size){
3
4 int i, j, iter, index;
5
6 //Recorremos las partes de la matriz asignadas a este bloque
7 for (i = blockIdx.x; i < size; i = i + gridDim.x){
```

```
  for (j = blockIdx.y; j < size; j = j + gridDim.y){

    index = i*size+j;
    matrix_result[index] = 0.0;

    //Calculamos el resultado de la multiplicacion para un indice
   especifico
    for (iter = 0; iter < size; iter++ ) {

    matrix_result[index] = matrix_result[index]
    + matrix_a[size*i+iter] * matrix_b[size*iter+j];

    }
  }
}
}
```

Código 4: *Multiplicación de 2 matrices cuadradas en GPU*

En esta función no utilizamos dobles vectores para definir la matriz sino que utilizamos el tamaño de la matriz para convertir sus índices a los de un único vector, hacemos esto para simplificar el proceso de mover la información entre la GPU y la CPU, similarmente en la CPU definimos las matrices con el Código 5.

```
struct Matrix{
int fil;
int col;
double* data;
};
```

Código 5: *Definición de un struct para matrices*

Para poder llamar la función sobre un conjunto bidimensional de bloques utilizamos la clase dim3, como ilustra el Código 6.

```
dim3 grid(1000,1000);
sqr_matrix_mult<<<grid,1>>>(gpu_A,gpu_B,gpu_C,N);
```

Código 6: *Definición de la grid y lanzamiento del kernel*

Si ahora ejecutamos este programa y lo comparamos con un programa alternativo paralelizado sobre la CPU con OpenMP (enlace) podemos observar que nuestro programa en CUDA es considerablemente más rápido, ¿como de rápido? Para poder medirlo introduzcamos el último elemento de esta sección, los eventos.

Para poder medir correctamente el tiempo de ejecución de nuestros programas sobre la GPU utilizamos una serie de funciones propias de CUDA:

- **cudaEvent_t evento**: definición de variable tipo evento.
- **cudaEventCreate(&evento)**: Crea el objeto evento.
- **cudaEventRecord (evento, stream)**: Captura el estado del stream en el momento de ejecución, veremos más adelante lo que significa stream, por ahora, sustituimos stream por 0.
- **cudaEventElapsedTime (tiempo, eventoInicial, eventoFinal)**: Devuelve el tiempo (en ms) que ha transcurrido entre los dos eventos capturados.
- **cudaEventDestroy (evento)**: Destruye el evento indicado.

Hay más funciones asociadas con los eventos pero para nuestros propósitos esto es suficiente, veamos ahora como utilizarlo para medir el tiempo de ejecución de nuestro programa en el Código 7.

```
1  //Creamos las variables evento
2  cudaEvent_t  start, stop;
3
4  //Creamos los eventos
5  cudaEventCreate (&start);
6  cudaEventCreate (&stop);
7
8  //Creamos una variable para almacenar los tiempos y otra para hacer
       media:
9  float timeTemp = 0;
10 float timeAvg = 0;
11
12 //Creamos un bucle
13 for(int i=0; i<1000; i++){
14
15 //Grabamos el evento inicial
16 cudaEventRecord(start,0);
17
18 //Enviamos nuestras matrices a la GPU
19 cudaMemcpy(gpu_A, A.data, N*N*sizeof(double), cudaMemcpyHostToDevice);
20 cudaMemcpy(gpu_B, B.data, N*N*sizeof(double), cudaMemcpyHostToDevice);
21
22 //Lanzamos nuestra funcion
```

```
23 sqr_matrix_mult<<<grid,1>>>(gpu_A,gpu_B,gpu_C,N);
24
25 //Extraemos nuestro resultado de la GPU
26 cudaMemcpy(C.data, gpu_C, N*N*sizeof(double), cudaMemcpyDeviceToHost);
27
28 //Grabamos el evento final
29 cudaEventRecord(stop,0);
30 cudaEventSynchronize(stop);
31
32 //Obtenemos el tiempo de ejecucion
33 cudaEventElapsedTime(&timeTemp, start, stop);
34 timeAvg = timeAvg + timeTemp;
35 }
36
37 //Destruimos los eventos
38 cudaEventDestroy(start);
39 cudaEventDestroy(stop);
40
41 //Imprimimos la media de tiempo de ejecucion
42 printf("Tiempo: %f\n", timeAvg/1000.0);
```

Código 7: *Medición del tiempo de ejecución mediante eventos*

Podemos ver aquí otra función, **cudaEventSynchronize(evento)**, esta función espera a que todos los procesos capturados en el evento terminen antes de continuar con el resto del código.

Compilando y ejecutando este código (disponible en Github) vemos que tarda una media de 180ms, por otro lado si ejecutamos el código equivalente en la CPU (disponible en Github) tenemos que tarda una media de 2,28 segundos con 10 hilos y ningún otro programa activo en el ordenador, es decir que en condiciones que podríamos llamar óptimas es 12,7 veces más lento que el equivalente en GPU, para el cual no hemos optimizado ninguna de esas condiciones.

3.2. Hilos

Para sacar el máximo partido a nuestra GPU no nos basta con repartir el trabajo en bloques, también es necesario dividirlo en hilos, y lo más esencial que debemos saber sobre los hilos es que no son independientes de los bloques, sino que cada bloque lanzará en su interior los hilos especificados, formando una estructura como la de la Figura 1.

En términos del lenguaje esto significa que si lanzamos un kernel con la siguiente

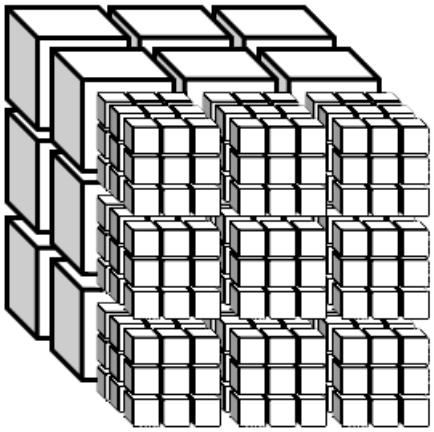

Figura 1: *Bloques e hilos*

instrucción <<<3,2>>>, se lanzarán 3 bloques y dentro de cada uno de estos bloques 2 hilos, y en caso de utilizar grids en vez de números ocurriría lo mismo pero en 2D-3D como indica el diagrama.

La primera pregunta que podemos plantear es, ¿por qué utilizar hilos? La respuesta proviene de la arquitectura de la GPU. Toda GPU contiene una cierto número de multiprocesadores (en nuestro caso 34), cada uno de los cuales puede ejecutar un bloque, por tanto, si solo utilizamos bloques como mucho podremos tener ese número de procesos en ejecución.

Por otro lado, cada multiprocesador puede tener a su vez varios hilos en paralelo en lo que se conoce como un warp, podemos ver la cantidad máxima de hilos en un warp con la variable predefinida **warpSize** (en nuestro caso 32), lo cual aumenta considerablemente la cantidad de paralelización que nos permite realizar la GPU.

Podemos ver esta mejora haciendo unos pequeños cambios a nuestro ejemplo anterior, ilustrados en el Código 8.

```
1  __global__ void sqr_matrix_mult(double *matrix_a, double *matrix_b,
       double *matrix_result, int size){
2
3  //Calculamos nuestro indice segun el bloque en el que estamos y nuestra
4  //posicion dentro de este
5  int i = threadIdx.x+blockDim.x + blockIdx.x;
6  int j = threadIdx.y+blockDim.y + blockIdx.y;
7
8
9  //Comprobamos que seguimos dentro de los limites de la matriz
10 if (i < size && j < size){
```

```
11   int index = i*size+j;
12   matrix_result[index] = 0.0;
13
14   for (int iter = 0; iter < size; iter++ ) {
15     matrix_result[index] = matrix_result[index]
16     + matrix_a[size*i+iter] * matrix_b[size*iter+j];
17     }
18   }
19 }
```

Código 8: *Ajuste de la función sqr_matrix_mult para utilizar hilos*

Y para lanzamos nuestro kernel con los parámetros ilustrados en el Código 9.

```
1 dim3 threadSize(64,64);
2 dim3 blockSize(N/threadSize.x + 1,N/threadSize.y + 1);
3
4 sqr_matrix_mult<<<blockSize,threadSize>>>(gpu_A,gpu_B,gpu_C,N);
```

Código 9: *Ajuste del kernel para utilizar hilos*

Hemos dividido la matriz en submatrices de 64x64 y asignamos cada una a un bloque según su indice relativo, creando los bloques justos para incluir toda la matriz. Si ahora ejecutamos nuestro código, tenemos que tarda 2,05 ms, 90 veces más rápido que nuestra versión con solo bloques.

Además de esta enorme mejora de eficiencia, los hilos presentan otras dos ventajas: la sincronización y la memoria compartida.

3.2.1. Sincronización

Hasta ahora hemos trabajado con algoritmos en paralelo que no requieren de ninguna cooperación interna, sin embargo existen muchos casos donde esto no es posible o práctico, es aquí donde el uso de hilos puede facilitar nuestro trabajo. Mientras que sincronizar diferentes bloques es más complicado y presenta ciertas limitaciones, sincronizar hilos es tan simple como utilizar el siguiente comando:

- **__syncthreads()**: Una vez un hilo llega a este comando se detiene la ejecución hasta que el resto de hilos de su bloque alcancen el mismo punto.

Es posible sincronizar los hilos de diversos bloques si lanzamos nuestro kernel con el comando cudaLaunchCooperativeKernel() [11], sin embargo esto presenta ciertas limitaciones:

Primero, el número de bloques totales que podemos lanzar esta limitado por el máximo de bloques por multiprocesador multiplicado por el número de multiprocesadores ($34 * 24 = 816$ en nuestro caso).

Segundo, nuestro dispositivo debe permitir este tipo de kernels, lo cual podemos comprobar mediante la propiedad **cudaDevAttrCooperativeLaunch**; si su valor es distinto de 0, el dispositivo lo soporta.

Tercero, este kernel no permite **paralelismo dinámico**, es decir, no podremos lanzar kernels desde dentro de un kernel, algo que CUDA permite desde la versión 5.0.

Adicionalmente, tener que sincronizar tantos hilos puede suponer un coste de eficiencia en caso de tener disparidades de tiempo de ejecución, por lo que no es muy recomendable. Generalmente en caso de necesitar sincronización entre bloques se lanza un segundo kernel desde la CPU al acabar el primero o se utilizan operaciones atómicas de las que hablaremos más adelante.

3.2.2. Memoria Compartida

El factor limitante por excelencia en la aplicaciones de GPU y por extensión CUDA es la latencia, cargar datos de la CPU a la GPU y luego desde la memoria de la GPU a cada hilo es costoso, es por ello que debemos hacer todo lo posible para facilitar el acceso a los datos que necesita el programa.

Hasta ahora nos hemos limitado a la memoria global, que pese a su versatilidad presenta la mayor latencia entre las memorias disponibles (a excepción de leer del disco), en el siguiente apartado discutiremos algunas de estas memorias, pero por ahora vamos a enfocar nuestra atención en la más relevante a los hilos, la memoria compartida.

La memoria compartida se diferencia de la global en varios aspectos, en primer lugar su nombre se debe a que se comparte unicamente entre los hilos de un mismo bloque, es decir, al iniciar un programa donde hemos reservado memoria compartida, cada bloque dispondrá de una copia aislada que podrá ser modificada exclusivamente por sus propios hilos. Debido a esto, la capacidad de esta memoria es considerablemente menor que la de la memoria global, en nuestro caso la memoria global dispone de 16 GB, mientras que la compartida está limitada a 49152 bytes por multiprocesador.

La principal ventaja de esta memoria es la velocidad de acceso, siendo mucho más veloz que la global y por tanto reduciendo considerablemente la latencia, esto se debe a que se guarda en el cache específico de cada multiprocesador en vez del general.

Además de mejorar la latencia existen muchas aplicaciones donde puede interesarnos

que cada bloque exprese su trabajo en una memoria separada y luego unir los resultados más adelante.

3.2.3. Producto escalar

Para observar tanto la memoria compartida como la sincronización de hilos en la práctica veamos una implementación del producto escalar en la GPU. Además, vamos a introducir otras dos funciones de CUDA: **cudaMemset** y **atomicAdd**.

- **cudaMemset(variableGPU, entero, tamañoVariableGPU)**: Inicializa o da a la variable el valor indicado, tratando los miembros como enteros y dando valor a tantos como indique tamañoVariableGPU.
- **atomicAdd(&variableGPU, sumando)**: Permite sumar a una misma variable desde diferentes hilos/bloques evitando conflictos de memoria.

Más adelantes hablaremos en mayor profundidad de la funciones atómicas, pero en este caso sencillamente lo utilizaremos para agrupar los resultados de cada bloque al final del programa.

La función cudaMemset, por otro lado, nos permite rellenar un vector de 0s, esto nos interesa debido a que cada bloque lanzará los hilos de 32 en 32, en caso de que un bloque tenga menos hilos ocupará el mismo espacio aun haciendo menos trabajo, es decir, no hay diferencia entre lanzar un bloque de 5 y 32 hilos, o de 64 y 35. Por tanto, para simplificar el programa redondearemos el tamaño de nuestros vectores al menor múltiplo del número de hilos mayor que su longitud. Dado que hacer esto en la CPU es un trabajo innecesario, definiremos los vectores con esa longitud en la CPU y rellenaremos el exceso con 0s, manteniendo el mismo producto escalar. Podemos observar este funcionamiento en el Código 10.

```
1  int padded_N;
2
3  if(N%threadSize==0)
4  padded_N = N;
5  else
6  padded_N = (N/threadSize+1)*threadSize;
7
8  //Reservamos espacio en la GPU para los vectores
9  cudaMalloc((void**) &gpu_vector_a, padded_N*sizeof(float));
10 cudaMalloc((void**) &gpu_vector_b, padded_N*sizeof(float));
11
```

```
//Rellenamos las variables en la GPU con 0s
cudaMemset(gpu_vector_a,0,padded_N*sizeof(float));
cudaMemset(gpu_vector_b,0,padded_N*sizeof(float));

//Enviamos los vectores a la GPU
cudaMemcpy(gpu_vector_a, vector_a, N*sizeof(float),
    cudaMemcpyHostToDevice);
cudaMemcpy(gpu_vector_b, vector_b, N*sizeof(float),
    cudaMemcpyHostToDevice);
```

Código 10: *Ejemplo de uso de cudaMemset*

Adicionalmente, dado que queremos guardar el resultado en la GPU antes de pasarlo a la CPU y será la suma de los valores de los bloques, creamos la variable y la inicializamos también a 0, como podemos observar en el Código 11.

```
//Reservamos espacio en la GPU para el resultado
cudaMalloc((void**) &gpu_result, sizeof(float));

//Inicializamos a 0
cudaMemset(gpu_result,0,sizeof(float));
```

Código 11: *Creación de la variable resultado en la GPU*

Dado que **cudaMemset** trata los valores como enteros podemos tener problemas si inicializamos variables con valores distintos de 0, por ejemplo, si ejecutamos el Código 12.

```
float *gpu_value;
float value;

cudaMalloc((void**) &gpu_value, sizeof(float));
cudaMemset(gpu_value, 100, sizeof(float));

cudaMemcpy(&value, &gpu_value[0], sizeof(float),cudaMemcpyDeviceToHost);

printf("value =%f\n",value);
```

Código 12: *Uso incorrecto de cudaMemset*

Tenemos que nuestro valor, que hemos intentado inicializar como 100, ronda 1.64e22, esto se debe a que para este comando gpu_value no es un float, un entero, rellenándolo con la representación binaria de 100. Una vez volvemos a tratarlo como un float C interpreta la expresion binaria de 100 como un float, resultando en este valor.

Por suerte para nosotros la expresión binaria del 0 es todo 0s tanto en int como en float, por tanto, no nos importa que el comando lo trate como esta lista.

Ahora que hemos explicado este comando, pasemos a nuestro kernel en el Código 13.

```
__global__ void productoEscalar(float *vector_a, float *vector_b, float
    *res){

//Creamos la variable compartida, donde almacenamos la seccion de
    producto escalar asignada al bloque
__shared__ float cache[threadSize];

//Definimos un iterador y una variable que indica que elemento del
    vector corresponde al hilo
int iter = blockDim.x;
int index = threadIdx.x + blockIdx.x*blockDim.x;

//Multiplicamos los elementos de los vectores en el indice asociado al
    hilo y lo guardamos en la variable compartida
cache[threadIdx.x] = vector_a[index]*vector_b[index];

//Nos aseguramos de que todos los productos se han realizado antes de
    pasar a la suma
__syncthreads();

while(iter > 1){

//
if(threadIdx.x < iter/2){
cache[threadIdx.x] = cache[threadIdx.x] + cache[threadIdx.x + iter/2];
}
iter = iter/2;

//En cada paso aseguramos que hemos realizado todas las sumas antes de
    pasar a la siguiente iteracion
__syncthreads();
}

//Utilizamos una operacion atomica para sumar los resultados de cada
    bloque en una unica variable
if(threadIdx.x == 0)
atomicAdd(&res[0], cache[0]);
}
```

Código 13: *Función producto escalar sobre la GPU*

La mayor parte del código es fácil de comprender solo con los comentarios, así que

vamos a centrarnos en el bucle que suma los elementos de la variable compartida.

En esencia, sumamos la primera mitad del vector con la segunda y la almacenamos en la primera, hecho esto, tratamos la primera mitad como el vector completo y repetimos hasta quedar un único elemento. Para ver esto de forma más clara veamos el diagrama de la Figura 2.

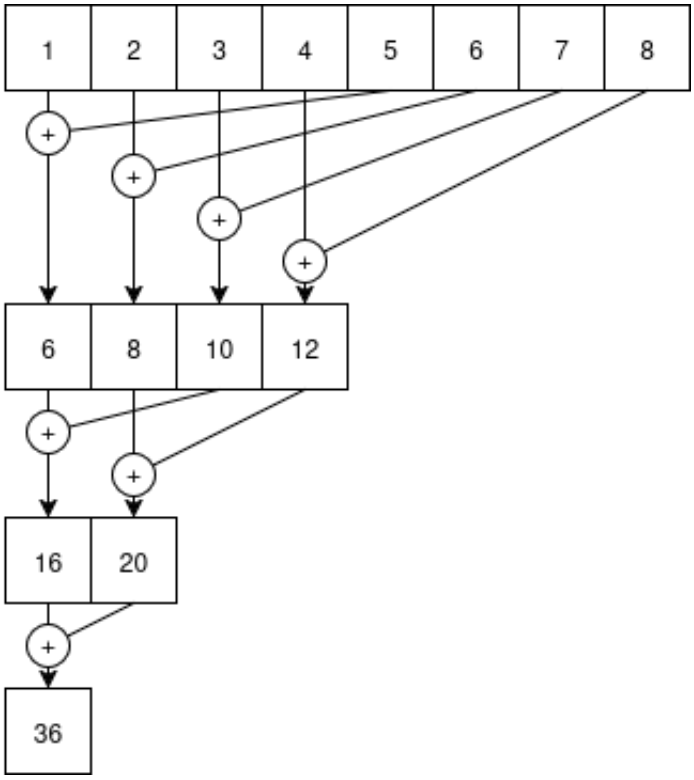

Figura 2: *Diagrama del bucle suma*

Como podemos ver en el código esta es la sección con más lecturas y escrituras de memoria, por lo cual se beneficia enormemente de trabajar en memoria compartida.

Finalmente para lanzar nuestro kernel simplemente incluimos el Código 14.

```
1 productoEscalar<<<padded_N/threadSize,threadSize>>>(gpu_vector_a,
    gpu_vector_b,gpu_result);
```

Código 14: *Lanzamiento del kernel productoEscalar*

El código completo está disponible en Github (enlace)

Una última observación fruto de la elaboración del código es que__syncthreads() nunca debe estar dentro de un condicional que puede no cumplirse, dado que si uno de los hilos no alcanza el comando, el resto se quedaran parados y nuestro programa dejará de

funcionar.

3.3. Memoria constante, registros y texturas

En la sección anterior hemos introducido la idea de diferentes tipos de memoria y hemos hablado de la memoria global y la memoria compartida, en este apartado hablaremos de los 3 tipos de memoria restantes, la memoria constante, los registros y las texturas.

3.3.1. Memoria constante

Como indica su nombre la memoria constante está dedicada a almacenar datos de solo lectura, presentando 2 propiedades especiales:

- Se guarda en caché, por lo que leer de la misma dirección varias veces no incrementa el trafico de memoria
- Se emite a todos los hilos de un warp simultáneamente, eliminando hasta 31 operaciones de memoria

Ambas propiedades tienen claras ventajas, sin embargo, es importante examinarlas en detalle. Lo primero que debemos saber es que si nuestro dispositivo tiene capacidad de computación 2.x o superior (algo muy probable), entonces los accesos a memoria global por bloque, es decir sin diferenciar por el id del hilo, también se guardan al caché del multiprocesador y dado que la memoria constante es mucho más limitada no es necesario usarla para este propósito. Si por otro lado queremos un acceso por hilo que se guarde en caché, si que nos interesa utilizar la memoria constante. En cuanto a la emisión de datos, esto solo es aplicable si todos los hilos de nuestro warp están accediendo al mismo elemento, como por ejemplo en el Código 15.

```
1 value = array[blockIdx.x];
```

Código 15: *Acceso a memoria por bloques*

Sin embargo, si nuestros hilos acceden a elementos separados la lectura de memoria se hará en serie por lo que puede ser más beneficioso utilizar la memoria global.

Para declarar la memoria como constante simplemente escribimos:

- **__constant__ clase nombre**

Podemos ver un ejemplo en el Código 16.

```
__constant__ int array[100];
```

Código 16: *Ejemplo de como reservar memoria constante*

Una vez la hemos declarado nos interesa darle un valor, algo que puede parecer contradictorio ya que en teoría es memoria **constante**, sin embargo, podemos reescribir su valor desde la CPU con el siguiente comando:

- **cudaMemcpyToSymbol(&variableGPU, &variableCPU, tamañoVariable)**

En caso de querer declarar un único valor como memoria constante es mejor utilizar #**define** y así no ocupamos ese espacio.

3.3.2. Registros

El concepto de registros en la GPU es similar a la CPU, a excepción de ciertos detalles. Primero cada multiprocesador de la GPU tiene miles de registros (65536 en nuestro caso) y en vez de asignarlos de forma dinámica, asigna registros reales a cada hilo [12](p.201). La ventaja que esto presenta es que el acceso a registros es mucho más eficiente, el problema es que limita la cantidad de registros que podemos tener sin afectar al rendimiento de nuestro programa.

Lo primero que tenemos que tener en cuenta para saber cuantos registros podemos utilizar es que si lanzamos N hilos entonces por cada variable (32 bits) declarada en un hilo necesitamos N registros. Teniendo esto en cuenta si queremos lanzar un bloque de 256 hilos en un multiprocesador con 64k registros podremos utilizar 65536/256=256 registros por hilo, alternativamente si necesitamos menos registros podremos llegar a lanzar más de un bloque por multiprocesador sin que los registros afecten a la eficiencia.

Alternativamente, podemos utilizar un programa como Nvidia Nsight ([13]) para calcular el numero de hilos/registros que debemos utilizar para maximizar el uso de nuestro dispositivo. Esta aplicación tiene multitud de usos relacionados con la programación en CUDA, pero nos limitaremos a esta utilidad.

Nvidia Nsight proporciona dos medidas de eficiencia, la primera es **occupancy** o ocupación, el ratio entre los warps activos y el máximo de warps permitidos en un micro-procesador, y la segunda es la cantidad de warps activos, siendo claramente equivalentes. Como norma general nos interesa que nuestro programa se acerque todo lo posible a un 100 % de ocupación por lo que al usar este programa podemos usar esa métrica para optimizar nuestra cantidad de registros y/o hilos.

Para ilustrar la diferencia de velocidad entre los registros y la memoria global vamos a comparar 2 kernels, uno que trabaja solo sobre memoria global y otro que trabaja principalmente con registros. Este kernel toma una lista de enteros aleatorios y suma a cada elemento otra lista de enteros aleatorios. Ambas listas contendrán 10000 elementos.

Veamos primero el kernel que trabaja sobre memoria global en el Código 17.

```
__global__ void kernel_gl_mem(int *list, int *add_list){
const int index = blockIdx.x + threadIdx.x*blockDim.x;

if(index<N){
  for(int i = 0; i<N; i++){
    list[index] = list[index] + add_list[i];
}}}
```

Código 17: *Suma de vectores sobre memoria global*

Podemos ver que cada paso del bucle leemos y escribimos a la variable global, en contraste, el Código 18 ilustra un kernel utilizando registros.

```
__global__ void kernel_reg_mem(int *list, int *add_list){
const int index = blockIdx.x + threadIdx.x*blockDim.x;
if(index<N){
  int value = list[index];
  for(int i = 0; i<N; i++){
    value = value + add_list[i];
}
list[index] = value;
}}
```

Código 18: *Suma de vectores sobre registros*

Podemos ver que solo realizamos una lectura desde memoria global para definir nuestra variable en un registro, después sumamos sobre esta variable y finalmente, escribimos el resultado de la suma a memoria global.

Ejecutamos ahora ambos kernels 100 veces para medir el tiempo de ejecución, código completo disponible en Github (enlace) ,obtenemos los siguientes resultados:

- Tiempo con memoria global: 4,5567 ms
- Tiempo con registros: 0,1059 ms

La diferencia entre los dos es obvia, el kernel con registros es 45 veces más rápido, en general nos interesa maximizar la cantidad de variables que almacenamos en registros,

teniendo que equilibrarlo con la cantidad de hilos que queremos lanzar, ya que con un nivel de paralelismo suficientemente alto podemos llegar a compensar las diferencias.

3.3.3. Texturas

El último tipo de memoria que vamos a explorar son las texturas, este tipo de memoria es de esperar debido al uso usual de las tarjetas gráficas y presenta ciertas ventajas para ciertos programas. Similar a la memoria constante las texturas se guardan en cache en el chip, ahorrando tiempo de transmisión de información, sin embargo, mientras que la memoria constante es muy pequeña y se accede por igual en todos los hilos, las texturas están optimizadas para un acceso espacial de la información, veamos que significa esto en más detalle:

Lo primero que debemos saber es que las texturas se utilizan principalmente para trabajar con datos organizados en 2 dimensiones, es decir en una cuadrícula como la ilustrada en la Figura 3.

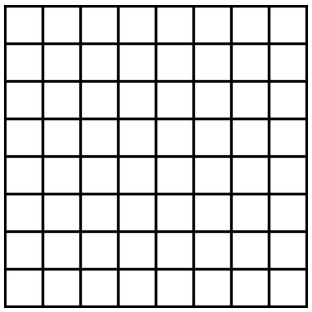

Figura 3: *Array 2d*

Ahora, cuando decimos un acceso espacial nos referimos a que los hilos consecutivos acceden a regiones de memoria cercanas entre si dentro de nuestra malla 2d, como por ejemplo en la Figura 4.

En caso de que nuestro programa acceda a la memoria de manera similar utilizar las texturas debería proporcionarnos una mejora en la eficiencia.

Ahora que hemos visto el contexto en el que se utilizan las texturas veamos como utilizarlas en la práctica. El primer paso como siempre es asignar esta memoria, que a diferencia de lo que hemos visto hasta ahora no es tan sencillo. Antes de poder construir nuestra textura debemos conocer una propiedad de nuestra tarjeta gráfica: el **texturePitchAlignment**. Para obtener este valor simplemente ejecutamos el Código 19.

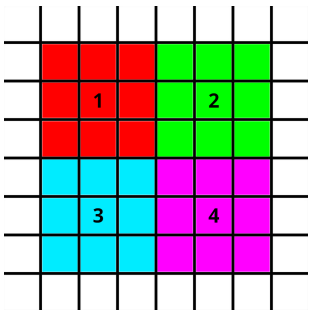

Figura 4: *Acceso de memoria por hilos*

```
cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0);
printf("texturePitchAlignment: %lu\n", prop.texturePitchAlignment);
```

Código 19: *Obtención de la propiedad texturePitchAlignment de nuestra GPU*

La razón por la que necesitamos conocer este valor es que el número de columnas de nuestra textura deber ser un múltiplo suyo, es decir, si es 32 nuestra textura deberá tener exactamente $32x$ columnas, $x \in \{1, 2, 3, ...\}$. Podemos ver las dimensiones máximas de nuestra textura ejecutando el Código 20.

```
printf( "Max 2D texture dimensions: (%d, %d)\n",
prop.maxTexture2DLinear[0], prop.maxTexture2DLinear[1]);
```

Código 20: *Obtención de las dimensiones máximas de una textura sobre nuestra GPU*

Si ejecutamos el código podemos ver que esta memoria tiene considerablemente más capacidad que las otras memorias especiales que hemos visto anteriormente, de nuevo algo predecible teniendo en cuenta la función usual de estas tarjetas.

Ahora que sabemos el valor de esta propiedad lo siguiente que debemos definir es el tipo de variable que contiene la textura, en particular, debe ser un tipo de entero (ej. **uint8_t**). Una vez elegimos el tipo de nuestra textura, creamos un puntero a una variable de ese tipo y le alocamos memoria del tamaño adecuado a la textura con el Código 21.

```
uint8_t* dataTexture;
cudaMalloc((void**)&dataDev, num_col*num_row*sizeof(uint8_t));
```

Código 21: *Alocación de memoria a la textura*

Esta variable contendrá los valores de nuestra textura y pese a ser unidimensional una vez creemos la textura tomará la forma adecuada.

Ahora tenemos que hablar de un tipo de variable exclusivo a CUDA necesario para crear una textura, no entraremos en detalle ya que se utiliza para otros propósitos, pero

en resumen se trata de una descripción que le indica a CUDA que tipo de recurso estamos creado y cuales son sus propiedades, se trata de **struct cudaResourceDesc** y vamos a explicar las partes que nos interesan dentro del Código 22.

```
//Creamos una serie de variables auxiliares
const int num_rows = 128;
const int num_cols = prop.texturePitchAlignment*4;
const int ts = num_cols*num_rows;
const int ds = ts*sizeof(uint8_t);

//Creamos la variable en cuestion
struct cudaResourceDesc resDesc;

//Damos valor 0 a todos los componente para evitar posibles problemas
   con los apartados que ignoramos
memset(&resDesc, 0, sizeof(resDesc));

//Declaramos el tipo de recurso a crear, en este caso una objeto 2D de
   tipo pitch
resDesc.resType = cudaResourceTypePitch2D;

//Declaramos el puntero que contiene los datos
resDesc.res.pitch2D.devPtr = dataDev;

//Declaramos las dimensiones del objeto
resDesc.res.pitch2D.width = num_cols;
resDesc.res.pitch2D.height = num_rows;

//Creamos un descriptor de canal del tipo adecuado a nuestra textura
resDesc.res.pitch2D.desc = cudaCreateChannelDesc<uint8_t>();

//Indicamos el numero de bytes de cada fila
resDesc.res.pitch2D.pitchInBytes = num_cols*sizeof(uint8_t);
```

Código 22: *Creación de un cudaResourceDesc*

Con estos valores definidos el siguiente paso involucra declarar dos nuevas variables de tipos exclusivos de CUDA, el primer tipo es **cudaTextureObject_t** que determina donde se establecerá nuestra textura y el segundo es **struct cudaTextureDesc** que declararemos como 0 pero es necesario para declarar nuestra textura. Vemos este procedimiento en el Código 23.

```
cudaTextureObject_t tex;
struct cudaTextureDesc texDesc;
```

```
3
4 memset(&texDesc, 0, sizeof(texDesc));
```

Código 23: *Creación de cudaTextureDesc*

Con todo esto en mano podemos finalmente crear nuestra textura con el comando **cudaCreateTextureObject()**, como ilustra el Código 24.

```
1 cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
```

Código 24: *Creación de la Textura*

Ahora que finalmente hemos creado nuestra textura el siguiente paso es acceder a su contenido, afortunadamente, esto es mucho más sencillo, simplemente utilizamos el siguiente comando:

- **tex2D<claseVariable>(textura, x, y)**: Devuelve el valor de clase *claseVariable* en la posicion (x,y) de la textura *textura*

Antes de hacer un ejemplo es importante mencionar que la forma en la que las texturas funcionan actualmente es relativamente reciente, en versiones anteriores las texturas se trabajaban mediante referencias, lo cual cambiaba completamente tanto su creación como el acceso a su memoria. La mayoría de fuentes online hacen referencia a este tipo de texturas que actualmente se encuentran deprecadas. Es gracias a [15], complementado con [11] que he conseguido recabar suficiente información para completar esta sección.

3.3.4. El juego de la vida de Conway

Para demostrar una aplicación de las texturas vamos a programar una de las simulaciones más conocidas en el mundo de la computación, el juego de la vida de Conway. El juego de la vida fue diseñado por el matemático John Horton Conway en 1970 en consiste en una sucesión de pasos discretos regulados por una serie de normas muy simples.

En esencia nosotros tenemos un tablero con dos tipos de celulas: vivas(blancas) y muertas(negras). Ilustramos esto en la Figura 5.

En el juego base las normas son las siguiente:

- Si una célula viva no está rodeada por 2 o 3 células vivas muere
- Si una célula muerta esta rodeada por exactamente 3 células vivas se convierte en una célula viva

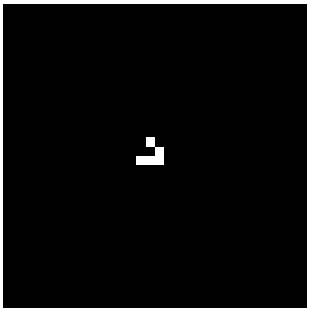

Figura 5: *Conway: ejemplo*

Estas normas se pueden representar con la nomenclatura 23/3, siendo la primera parte el número de células vivas necesarias para sobrevivir y la segunda las células vivas necesarias para sobrevivir. Podemos utilizar otras normas que en consecuencia llevan a otro tipo de resultados, en nuestra implementación daremos la opción de elegir cualquier set de normas.

Tenemos también que elegir como tratar los bordes dado que nuestro tablero será finito, para simplificar el código y por preferencia personal haremos que sean periódicos.

Ahora que hemos descrito el funcionamiento del programa en abstracto, veamos como implementarlo. Lo primero es preguntarnos si las texturas nos benefician en algo, para ello debemos ver como acceda cada hilo a los datos. Nuestra textura será el tablero del paso anterior, mientras que cada hilo simulará una celda y accederá únicamente a las celdas que lo rodean, como ilustra el siguiente gráfico de la Figura 6.

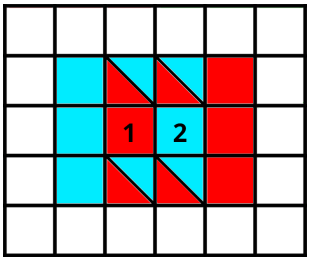

Figura 6: *Conway: acceso a memoria*

Podemos ver que el acceso es prácticamente el caso ideal para el uso de texturas y por tanto es un uso apropiado.

Ahora que hemos decidido utilizar una textura el siguiente paso es crearla, siguiendo la guía del apartado anterior tenemos el Código 25.

```
1 cudaDeviceProp prop;
2 cudaGetDeviceProperties(&prop, 0);
```

```
cudaTextureObject_t tex;

//Podemos modificar las dimensiones alterando estos numeros
const int num_rows = 320;
const int num_cols = prop.texturePitchAlignment*10;
const int ts = num_cols*num_rows;
const int ds = ts*sizeof(sm);
sm* dataDev;
cudaMalloc((void**)&dataDev, ds);

//Creamos la textura
struct cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypePitch2D;
resDesc.res.pitch2D.devPtr = dataDev;
resDesc.res.pitch2D.width = num_cols;
resDesc.res.pitch2D.height = num_rows;
resDesc.res.pitch2D.desc = cudaCreateChannelDesc<sm>();
resDesc.res.pitch2D.pitchInBytes = num_cols*sizeof(sm);
struct cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
```

Código 25: *Creación de la textura/tablero*

En el preludio del programa definimos:

- **typedef uint8_t sm;**

Ahora definimos los datos iniciales de la textura, comenzando con un tablero vacío y seleccionando manualmente las células vivas (en el patrón del ejemplo anterior), adicionalmente creamos un tablero de salida para el programa, que inicializamos con los mismos datos. Ilustramos todo esto en el Código 26.

```
sm *dataIn;
dataIn =(sm*) malloc(ts);

//Inicializamos un tablero vacio y un tablero de output
for (int i = 0; i < ts; i++) dataIn[i] = 0;

sm* dataOut;
cudaMalloc((void**)&dataOut, ds);

//Declaramos las celulas iniciales
```

```
dataIn[num_cols*16+14] = 1;
dataIn[num_cols*16+15] = 1;
dataIn[num_cols*16+16] = 1;
dataIn[num_cols*15+16] = 1;
dataIn[15+num_cols*14] = 1;

//Pasamos los datos iniciales a la textura y al output
cudaMemcpy(dataDev, dataIn, ds, cudaMemcpyHostToDevice);
cudaMemcpy(dataOut, dataIn, ds, cudaMemcpyHostToDevice);
```

Código 26: *Inicialización de los tableros de entrada y salida*

Los últimos datos que nos quedan por definir antes de pasar a la función como tal son las normas, las implementaremos como 2 vectores de 9 componentes que podrán ser 1 o 0. Dado que no modificaremos estos valores a lo largo del programa y es muy posible que un bloque de hilos lea la misma lista, lo definimos en la memoria constante en el Código 27.

```
//Definiremos las normas en memoria constante
__constant__ sm liveArray[9] = {0,0,1,1,0,0,0,0,0}; //1 si la celula
    vive, 0 si muere
__constant__ sm breedArray[9] = {0,0,0,1,0,0,0,0,0}; //1 si la celula
    nace, 0 si no
```

Código 27: *Definición de las normas en memoria constante*

Ahora que hemos visto con que tipo de datos estamos trabajando el siguiente paso es implementarlo. Dividiremos esta implementación en 2 partes, una primera que contará las células que rodean a nuestra celda y una segunda que tomará este número y aplicará las normas para decidir el estado de la celda al terminar el paso.

Para la mayoría de celdas la implementación de este conteo es relativamente simple, es una suma de los valores entorno a los índices de la celda, calculando el resto de los índices a la derecha y debajo entre el tamaño total para satisfacer la periodicidad en estos bordes, como ilustra el Código 28.

```
__device__ int conwaySum(int *sum, int x, int y, int mode,
    cudaTextureObject_t tex){
int rows = blockDim.y*gridDim.y;
int cols = blockDim.x*gridDim.x;
*sum =
  tex2D<sm>(tex, (x - 1), y) +
  tex2D<sm>(tex, (x + 1) % cols , y) +
  tex2D<sm>(tex, x, (y - 1)) +
  tex2D<sm>(tex, (x - 1), (y - 1)) +
```

```
  tex2D<sm>(tex, (x + 1) % cols , (y - 1)) +
  tex2D<sm>(tex, x, (y + 1) % rows) +
  tex2D<sm>(tex, (x - 1), (y + 1) % rows) +
  tex2D<sm>(tex, (x + 1) % cols , (y + 1) % rows);
```

Código 28: *Suma de valores de las celdas del entorno*

Aunque el código parezca complicado simplemente estamos sumando los valores de la textura en las 8 celdas que rodean a la nuestra. El problema aparece cuando estamos trabajando con los bordes izquierdo y superior, ya que al estar restando nuestra división entera no da un resultado apropiado, es por esto que los tratamos por separado mediante un switch, determinando el borde en el que están con el Código 29 en la función principal.

```
int edge = 0;
//En caso de estar en estos bordes no basta con la division para
   conseguir todos los valores, asi que los tratamos por separado
if(x == 0) edge = edge + 1;
if(y == 0) edge = edge + 2;
```

Código 29: *Cálculo del borde*

Y completamos la función de conteo en el Código 30.

```
__device__ int conwaySum(int *sum, int x, int y, int mode,
   cudaTextureObject_t tex){
int rows = blockDim.y*gridDim.y;
int cols = blockDim.x*gridDim.x;

//Calculamos segun el lugar donde se encuentra la celula
switch(mode){
case 0:
  //...
break;
case 1:
  //...
break;
case 2:
  //...
break;
case 3:
  //...
break;
}
```

Código 30: *Función de conteo*

Siendo el caso 0 el que ya hemos visto, el caso 1 si se encuentra en el borde izquierdo, el caso 2 si se encuentra en el borde superior y el caso 3 si se encuentra en la esquina superior izquierda. Definimos la función como **__device__** en vez de **__global__** ya que solo accederemos a ella desde otra función ejecutada en la GPU.

El código completo se puede observar en Github (enlace)

Teniendo esta función el resto de la implementación es relativamente simple, como podemos ver en el Código 31.

```
__global__ void conway(sm *output, cudaTextureObject_t tex){
int sum;
int x = threadIdx.x+blockDim.x*blockIdx.x;
int y = threadIdx.y+blockDim.y*blockIdx.y;
int edge = 0;

//En caso de estar en estos bordes no basta con la division para conseguir todos los valores, asi que los tratamos por separado
if(x == 0) edge = edge + 1;
if(y == 0) edge = edge + 2;

//Llamamos a la funcion suma para ver el numero de celulas vivas a su alrededor
conwaySum(&sum, x, y, edge, tex);

//Decidimos el estado de la celula segun las normas que hemos indicado
if (tex2D<sm>(tex, x, y) == 0 && breedArray[sum] == 1){
output[x + y * blockDim.x * gridDim.x] = 1;
}
else if (tex2D<sm>(tex, x, y) == 1 && liveArray[sum] == 0){
output[x + y * blockDim.x * gridDim.x] = 0;
}
}
```

Código 31: *Implementación del Juego de la Vida de Conway*

Como podemos ver en la función miramos un array u otro según el estado de la celda y solo lo cambiamos si el valor de ese array en la posición suma es el adecuado, en caso contrario la celda es estable y por tanto se mantiene con su valor anterior.

Ahora que tenemos una implementación, nos interesa poder visualizar los resultados, para ellos utilizamos un paquete de C que nos permite codificar GIFs manualmente, gifenc [16].

Para incorporar este código en C a nuestra aplicación en CUDA añadimos lo siguiente

al inicio de nuestro código:

- **#include "./gifenc.h"**
- **#include <cstdint>**

Y lo compilamos de la siguiente manera:

- nvcc example8.cu gifenc.c -o example8

Los detalles del paquete pueden verse en su repositorio pero para nuestros propósitos solo vamos a ver como utilizarlo al nivel más simple. Lo primero que debemos hacer para crear un GIF es definir sus dimensiones y una paleta de colores, como ilustra el Código 32.

```
int w = num_cols, h = num_rows;
uint8_t palette[] = {
    0x00, 0x00, 0x00, // 0 -> black
    0xFF, 0xFF, 0xFF, // 1 -> white
    };
```

Código 32: *Definición de la paleta de colores*

La principal ventaja de este paquete frente a otros equivalentes es esta paleta de colores. Mientras que muchos paquetes codifican los frames con 3 arrays (uno para cada elemento de RGB), este nos permite trabajar con un único array de elementos mucho más pequeños siempre que limitemos nuestros colores, en nuestro caso es perfecto dado que trabajamos con 2.

Ahora para crear el GIF tomamos estos valores y un nombre para el archivo y ejecutamos el Código 33.

```
ge_GIF *gif;

gif = ge_new_gif(
        "conway_23_3.gif",  /* file name */
        w, h,           /* canvas size */
        palette,
        1,              /* palette depth == log2(# of colors) */
        -1,             /* no transparency */
        0               /* infinite loop */
  );
```

Código 33: *Creación del GIF*

Ahora, cada vez que queremos añadir un frame a nuestro GIF cambiamos los datos del array **gif->frame** y ejecutamos el comando **ge_add_frame(gif, 5)**, siendo el 5 la duración del frame en nuestra animación en centésimas de segundo. Por tanto declaramos el frame inicial con el Código 34.

```
gif->frame =  dataIn;
ge_add_frame(gif, 5);
```

Código 34: *Declaración del frame inicial*

Ahora, para los siguientes frames exportamos directamente los datos de la GPU a nuestro array y actualizamos los datos de la textura usando cudaMalloc entre elementos de la GPU, como podemos ver en el Código 35.

```
for (int i = 0; i < frames; i++) {
  printf("frame = %d\n",i);

    //Procesamos el siguiente frame
    conway<<<blockSize, threadSize>>>(dataOut, tex);
    cudaDeviceSynchronize();

    //Damos los datos del frame nuevo a la textura
    cudaMemcpy(dataDev, dataOut, ds, cudaMemcpyDeviceToDevice);

  //Lo extraemos y insertamos al gif
  cudaMemcpy(gif->frame, dataDev, ds, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    ge_add_frame(gif, 5);
}
```

Código 35: *Bucle de generación del GIF*

Ahora con todo esto basta con ejecutar el código, en el caso base obtenemos el siguiente GIF, ilustramos alguno de sus frames en la Figura 7.

Podemos ver el resultado esperado, nuestras células recorriendo el tablero diagonalmente.

Por otro lado, si cambiamos la normas a 23/23 (incrementando la supervivencia) obtenemos el siguiente GIF, ilustramos alguno de sus frames en la Figura 8.

Podemos ver que al facilitar la supervivencia las células vivas proliferan mucho más, podemos observar además que la ejecución es más lenta, sin entrar en un análisis muy profundo esto puede deberse a que en la primera simulación la mayoría de celdas estaban muertas por lo que todas leían la misma lista de memoria constante aprovechando así sus

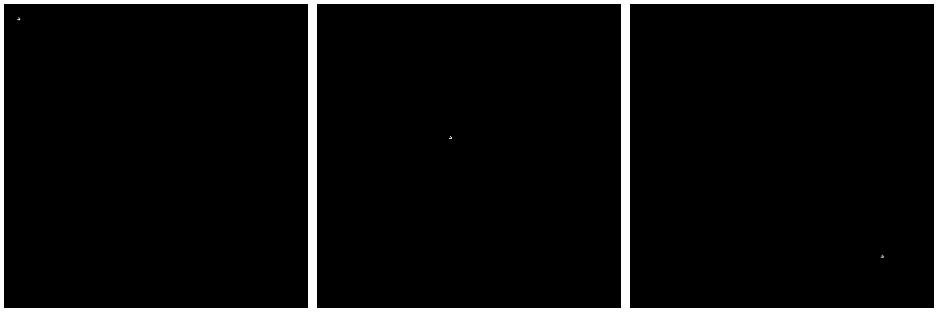

Figura 7: *Conway 23/3 (frame 0-500-1000)*

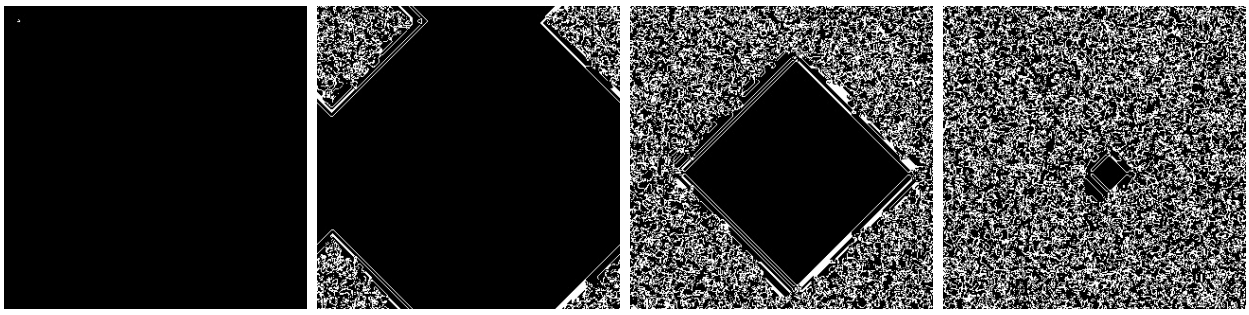

Figura 8: *Conway 23/23 (frame 0-100-200-300)*

propiedades, mientras que la segunda animación es mucho más caótica requiriendo que cada hilo reciba una lista distinta y por tanto trabajando en serie.

Este ejemplo no solo muestra que las texturas son relativamente útiles sino que son muy sencillas de utilizar, sin embargo, en situaciones donde requerimos una memoria mucho más grande seguimos necesitando recurrir a la memoria global o dividir nuestro problema en partes separadas.

3.4. Funciones atómicas

Aunque ya hemos introducido el concepto de función atómica en el apartado 3.2.3, es importante explorarlo en profundidad.

En esencia, la funciones atómicas permiten a diferentes hilos de CUDA interactuar con una misma variable evitando conflictos de acceso. Esto simplifica considerablemente nuestro programa ya que nos ahorra tener que hacer esta sincronización manualmente.

Existe una amplia selección de funciones atómicas en CUDA, como podemos ver en [14], la mayoría siguen un esquema similar a **AtomicAdd()**, modificando en memoria la variable cuya dirección indicamos en el primer argumento según la variable dada en

el segundo. Además esta funciones devuelven el valor original, algo particularmente útil en el caso de funciones como **AtomicExch()** que guarda el segundo argumento en la posición del segundo y nos devuelve el valor original guardado en dicha posición, algo que nos permite crear el Código 36.

```
__global__ void hiloFinal(int *hilo){
int nonsense = 0;
for(long int i = 0; i<255-threadIdx.x;i++){
nonsense++;
}
atomicExch(&(hilo[0]),nonsense);
}
```

Código 36: *Kernel utilizando atomicExch()*

Cuando ejecutamos esta función el valor final de la variable *hilo* será el último hilo en terminar de compilar, algo que puede sernos útil si queremos diagnosticar problemas o simplemente ver como organiza nuestra GPU la ejecución de hilos.

Otras funciones atómicas que pueden resultar muy útiles son:

- **AtomicMin(),AtomicMax()**: Leen una variable en memoria, la comparan con uno nuevo que le pase el hilo y almacenan el menor/mayor de los dos en la primera variable.
- **AtomicAnd(),AtomicOr(),AtomicXor()**: Realizan estas operaciones lógicas entre una variable en memoria y un valor dado y guardan el resultado en la primera variable.

Para ver que ocurre si no utilizamos las funciones atómicas y por qué no debemos abusar de ellas vamos a comparar tres implementaciones del producto escalar, la primera de ellas está ilustrada en el apartado 3.2.3 y las otras dos se muestran en los Códigos 37 y 38.

```
__global__ void productoEscalarNoAtomic(float *vector_a, float *vector_b
    , float *res){

__shared__ float cache[threadSize];
int iter = blockDim.x;
const int index = threadIdx.x + blockIdx.x*blockDim.x;

cache[threadIdx.x] = vector_a[index]*vector_b[index];
__syncthreads();
```

```
9
10 while(iter > 1){
11
12 if(threadIdx.x < iter/2){
13 cache[threadIdx.x] = cache[threadIdx.x] + cache[threadIdx.x + iter/2];
14 }
15
16 iter = iter/2;
17 __syncthreads();
18 }
19 if(threadIdx.x == 0)
20 res[0]= res[0]+cache[0];
21 }
```

Código 37: *Implementación del producto escalar sin funciones atómicas*

Esta segunda implementación ignora las funciones atómicas y simplemente suma directamente a memoria.

```
1 __global__ void productoEscalarAtomicAll(float *vector_a, float *
     vector_b, float *res){
2 const int index = threadIdx.x + blockIdx.x*blockDim.x;
3 atomicAdd(&res[0], vector_a[index]*vector_b[index]);
4 }
```

Código 38: *Implementación del producto escalar solo con funciones atómicas*

Esta tercera implementación ignora la memoria compartida y cada hilo suma directamente usando **AtomicAdd()**.

Ahora ejecutamos el siguiente código (disponible en Github) para obtener una serie de comparaciones. Lo primero a saber es que en todos los casos estamos multiplicando dos vectores de un millón de floats cuyos elementos son todos $1,0$, sabiendo esto podemos comparar los resultados de nuestra función original con nuestra función no atómica:

- Resultado atómico $= 1000000,0$
- Resultado no atómico $= 20416,0$

Como era de esperar, sin las funciones atómicas nuestro programa sucumbe a las condiciones de carrera entre los diferentes bloques y nos da un resultado completamente incorrecto.

Por otro lado es importante no abusar de las funciones atómicas, ya que al acceder todas a la misma ubicación en memoria en serie pueden causar enormes problemas de

eficiencia, podemos ver esto claramente si comparamos los tiempos de ejecución entre la implementación original y la implementación sumando directamente con la función atómica:

- Tiempo original: $0,027515$
- Tiempo modificado: $1,441171$

Un aumento del tiempo de ejecución del 7000 %, ilustrando que aunque son las funciones atómicas son de gran es mejor limitar su uso a lo estrictamente necesario si queremos hacer un programa rápido y eficiente que realmente aproveche el paralelismo de la GPU.

3.5. Streams

Como hemos visto en el apartado anterior, una de las principales limitaciones de la eficiencia de nuestros programas en la GPU es la latencia de la memoria, y entre todos los tipos de movimientos de datos el que incurre una mayor latencia es el paso de datos entre el disco y la GPU. Pese a que esta latencia es inevitable, podemos utilizar los **streams** de CUDA para reducir su impacto. Un **stream** es una cola de operaciones sobre la GPU, por ejemplo, todo lo que hemos lanzado hasta ahora sobre la GPU ha sido dentro del **stream predeterminado**.

La razón por la que nos interesa utilizar estos **streams** es el hecho de que se ejecutan en paralelo. Si únicamente nos preocupara la capacidad de cálculo de la GPU esto podría parecer un uso relativamente de nicho, dado que a menos que queramos ejecutar varios programas diferentes que solo consumen porciones de la capacidad de la GPU, los programas que aprovechan toda su potencia seguirán ejecutándose en serie conforme se liberen los procesadores. Sin embargo, como hemos indicado nuestra principal preocupación es la latencia, y esta ejecución en paralelo nos permite transmitir datos, al mismo tiempo que se procesan otros, como ilustra este diagrama de la Figura 9.

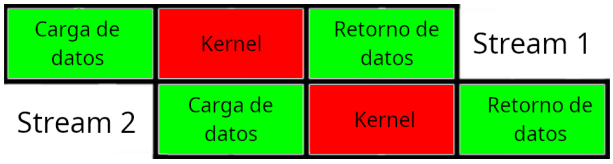

Figura 9: *Diagrama Streams*

En un caso ideal esto nos permitiría dividir a la mitad la latencia de envió y retorno de datos desde el disco, pero hasta en casos menos idílicos podemos ver mejoras de eficiencia,

siempre y cuando nuestro algoritmo pueda dividirse en secciones independientes, tanto en términos de datos como operaciones.

Para crear un **stream** el primer paso es declararlo de la siguiente manera:

- **cudaStream_t** stream

Una vez declarado, podemos crearlo con el siguiente comando:

- **cudaStreamCreate**(&stream)

Y cuando acabemos de trabajar con el podemos eliminarlo con:

- **cudaStreamDestroy**(&stream)

Ahora, para lanzar un kernel en un stream específico debemos añadir 2 parámetros a nuestra instrucción:

funcion<<<bloques,hilos, **bytes_compartidos**, **stream**>>>(param1,...)

- **bytes_compartidos**: Tamaño en bytes de la memoria compartida alocada dinámicamente por cada bloque
- **stream**: Stream en el que ejecutamos nuestro kernel

El primer parámetro (**bytes_compartidos**) no nos interesa particularmente y generalmente simplemente lo declararemos como 0, por otro lado el segundo parámetro (**stream**) es lo que permite el paralelismo que buscamos.

En cuanto al movimiento de memoria, no basta con utilizar CudaMemcpy(), dado que se ejecuta siempre en serie dentro del stream predeterminado, debemos utilizar un nuevo comando muy similar:

cudaMemcpyAsync (variableReceptora, &variableEmisora, tamañoVariable, dirección, stream)

Los parámetros y funcionamiento de esta función son idénticos a CudaMemcpy() a excepción del último parámetro (**stream**) que dice a CUDA en que cola debe colocar esta transmisión de memoria, permitiendo que una cola transmita mientras otra procesa, de forma asíncrona, como indica el nombre de la función.

Finalmente, del mismo modo que utilizamos cudaDeviceSynchronize () para asegurarnos de que todas las operaciones que hemos lanzado en nuestra GPU hayan acabado, si queremos asegurar que las instrucciones de un stream se hayan ejecutado utilizamos el comando:

- **cudaStreamSynchronize**(stream): Similar a cudaDeviceSynchronize() pero exclusivo para los comandos ejecutados sobre *stream*

Para ver el funcionamiento de los streams en la práctica veamos un ejemplo muy sencillo, la suma de 2 vectores, pero vamos a trabajar con dichos vectores como si se trataran de objetos demasiado grandes como para tener guardados en memoria, es decir, lo pasaremos a la GPU por bloques, sumaremos las secciones y devolveremos las partes del resultado después de procesarlas. Primero definimos la función que realizará la suma con el Código 39.

```
__global__ void sumaVectorial(int *gpu_a, int *gpu_b, int *gpu_result,
    int size) {
  int index = threadIdx.x + blockDim.x*blockIdx.x;
  if(index < size) gpu_result[index] = gpu_a[index] + gpu_b[index];
}
```

Código 39: *Función suma vectorial*

Ahora, veamos primero una implementación sin utilizar streams ($N = 1e9, partial\_N = 1e4$) en el Código 40

```
//Definimos los vectores
int *gpu_a, *gpu_b, *gpu_res;
int size = partial_N *sizeof(int);

//Alocamos la memoria
cudaMalloc((void**)&gpu_a, size);
cudaMalloc((void**)&gpu_b, size);
cudaMalloc((void**)&gpu_res, size);


//Comenzamos a sumar por partes
int i;
for(i = 0; i < N; i = i + partial_N){
cudaMemcpy(gpu_a, &vect_a[i], size, cudaMemcpyHostToDevice);
cudaMemcpy(gpu_b, &vect_b[i], size, cudaMemcpyHostToDevice);
sumaVectorial<<<partial_N/64+1, 64>>>(gpu_a, gpu_b, gpu_res, partial_N);
cudaMemcpy( &vect_res[i], gpu_res, size, cudaMemcpyDeviceToHost);
}

//Sumamos los elementos restantes
if((N-partial_N) > 0){
cudaMemcpy(gpu_a, &vect_a[i], (N-partial_N)*sizeof(int),
    cudaMemcpyHostToDevice);
```

```
cudaMemcpy(gpu_b, &vect_b[i], (N-partial_N)*sizeof(int),
    cudaMemcpyHostToDevice);
sumaVectorial<<<partial_N/64+1, 64>>>(gpu_a, gpu_b, gpu_res, (N-
    partial_N)*sizeof(int));
cudaMemcpy( &vect_res[i], gpu_res, (N-partial_N)*sizeof(int),
    cudaMemcpyDeviceToHost);
}


cudaDeviceSynchronize();
```

Código 40: *Implementación sin streams*

Simplemente recorremos los vectores por secciones, cargando en memoria solo lo que necesitamos en ese momento.

Ahora, para realizar el mismo algoritmo utilizando streams, creamos 2 streams y alternamos las instrucciones entre uno y otro, como podemos ver en el Código 41.

```
cudaStream_t s_1, s_2;
cudaStreamCreate(&s_1);
cudaStreamCreate(&s_2);

//Definimos los vectores
int *gpu_a_1, *gpu_b_1, *gpu_res_1;
int *gpu_a_2, *gpu_b_2, *gpu_res_2;
int size = partial_N *sizeof(int);

//Alocamos la memoria
cudaMalloc((void**)&gpu_a_1, size);
cudaMalloc((void**)&gpu_b_1, size);
cudaMalloc((void**)&gpu_res_1, size);
cudaMalloc((void**)&gpu_a_2, size);
cudaMalloc((void**)&gpu_b_2, size);
cudaMalloc((void**)&gpu_res_2, size);

//Comenzamos a sumar por partes
int i;
int j = 1;
for(i = 0; i < N; i = i + partial_N){
//Alternamos los streams
j = -j;
if(j == 1){
cudaMemcpyAsync(gpu_a_1, &vect_a[i], size, cudaMemcpyHostToDevice, s_1);
cudaMemcpyAsync(gpu_b_1, &vect_b[i], size, cudaMemcpyHostToDevice, s_1);
```

```
27 sumaVectorial<<<partial_N/64+1, 64, 0, , s_1>>>(gpu_a_1, gpu_b_1,
      gpu_res_1, partial_N);
28 cudaMemcpyAsync( &vect_res[i], gpu_res_1, size, cudaMemcpyDeviceToHost,
      s_1);
29 }
30 else{
31 cudaMemcpyAsync(gpu_a_2, &vect_a[i], size, cudaMemcpyHostToDevice, s_2);
32 cudaMemcpyAsync(gpu_b_2, &vect_b[i], size, cudaMemcpyHostToDevice, s_2);
33 sumaVectorial<<<partial_N/64+1, 64, 0, , s_2>>>(gpu_a_2, gpu_b_2,
      gpu_res_2, partial_N);
34 cudaMemcpyAsync( &vect_res[i], gpu_res_2, size, cudaMemcpyDeviceToHost,
      s_2);
35 }
36 }
37
38 //Sumamos los elementos restantes
39 if((N-partial_N) > 0){
40 if(j == 1){
41 ...
42 }
43 else{
44 ...
45 }
46 }
47
48 cudaStreamSynchronize(s_1);
49 cudaStreamSynchronize(s_2);
```

Código 41: *Implementación con streams*

Como podemos ver la implementación es prácticamente idéntica, únicamente alternando stream en nuestro bucle. Si ahora medimos los tiempos de ejecución de ambas implementaciones obtenemos:

- Tiempo sin streams: 316,24 ms
- Tiempo con streams: 257,58 ms

Efectivamente la implementación con streams es más eficiente. El código completo esta disponible en Github (enlace).

4. Filtro Bilateral

Ahora que tenemos una visión general de como crear código para la GPU mediante CUDA, veamos una aplicación real, el filtro bilateral. El cómputo en GPUs es especialmente útil en algoritmos de filtrado de imágenes, como el filtro bilateral, que requieren realizar numerosos cálculos locales por píxel y, por tanto, presentan un alto coste computacional. La aceleración proporcionada por CUDA permite abordar estos filtros en alta resolución y con parámetros más exigentes, haciendo posible su integración en aplicaciones que demandan procesamiento en tiempo real o análisis de grandes volúmenes de datos.

4.1. ¿Qué es el filtro bilateral?

Introducido por Tomasi y Manduchi en 1998 [17] el filtro bilateral es un filtro no lineal que busca suavizar una imagen conservando los bordes, además, en las imágenes a color no solo no presenta colores fantasma cerca de los bordes sino que en caso de existir, los reduce.

El filtro consiste en aplicar la siguiente fórmula a cada píxel:

$$I^{filt}(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) f_r(||I(x_i) - I(x)||) g_s(||x_i - x||)$$

$$W_p = \sum_{x_i \in \Omega} f_r(||I(x_i) - I(x)||) g_s(||x_i - x||)$$

Donde

- I^{filt}: Imagen filtrada
- I: Imagen original
- x: Coordenadas de un píxel
- Ω: Un entorno del píxel a filtrar
- W_p: El término de normalización
- f_r: Función que suaviza las diferencias en intensidad
- g_s: Función que suaviza las diferencias en coordenadas

En nuestro caso vamos a tomar Ω como los píxeles a distancia menor a *KERNEL_RADIUS*, tomando la distancia del taxista. Además, trabajaremos cada canal de la imagen por separado y definiremos:

$$f_r(||I(x_i) - I(x)||) = e^{-\frac{(I(x_i) - I(x))^2}{2\sigma_r(x)^2}}, \quad \sigma_r(x) = 2\sigma_{\Omega_x}.$$

Siendo $\sigma^2_{\Omega_x}$ la varianza local en el entorno de x.

Finalmente, definiremos:

$$g_s(||x_i - x||) = e^{-\frac{||x_i - x||^2}{2\sigma_s^2}}.$$

Siendo σ_s un parámetro predefinido en nuestro programa.

Resumiendo, para cada píxel de nuestra imagen necesitaremos calcular el valor de estas funciones en todos los puntos de su entorno, para lo cual deberemos haber calculado previamente la varianza local, y en consecuencia, la media local.

4.2. Implementación sobre CPU

Vamos a partir de un código ya creado en C para realizar nuestra adaptación a CUDA, por tanto el primer paso debe ser entender este código (disponible en Github).

Primero, tanto para leer como para escribir las imágenes utiliza el paquete stb [18]. Para leer la imagen, sus dimensiones y el número de canales presentes en ella utiliza el siguiente comando:

- **unsigned char *img = stbi_load(argv[1], &width, &height, &channels, 0);**

Donde **argv[1]** es el nombre de la imagen que queremos leer y **img** es una lista de valores entre 0 y 255. Los valores corresponden a cada píxel leyendo de izquierda a derecha y de arriba a bajo. Los valores de cada píxel en cada canal se colocan seguidos, es decir, tendremos el valor del píxel 1 en el canal 1 seguido del valor del píxel 1 en el canal 2, etc. hasta recorrer todos los canales y pasar al píxel 2.

Y para escribir la imagen, utilizamos el comando:

- **stbi_write_png(argv[2], width, height, channels, output, width * channels);**

Donde **argv[2]** es el nombre de la imagen a escribir y **output** es una lista en el mismo formato que **img**.

Entrando ahora dentro de la implementación del filtro como tal, lo primero que observamos es la normalización de los valores del input en el Código 42.

```
float *inF = (float*)malloc(width * height * channels * sizeof(float));
    for (i = 0; i < width * height * channels; i++)
        inF[i] = input[i] / 255.0f;
```

Código 42: *Bucle de normalización*

Una vez creado un vector de floats normalizados, lo siguiente que encontramos es el conjunto de bucles que recorren todos los píxeles de la imagen en todos sus canales en el Código 43.

```
for (y = 0; y < height; y++) {
        for (x = 0; x < width; x++) {
            for (c = 0; c < channels; c++) {
            ...
            }}}
```

Código 43: *Bucle de recorrido del entorno*

Es importante mencionar esta parte dado que será el elemento principal de la paralelización. Dentro de estos bucles, el cálculo de la media y la varianza se realiza de forma muy similar, por ejemplo, la media se calcula con el Código 44.

```
for (j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++) {
    int yy = y + j;
    if (yy < 0 || yy >= height) continue;
        for (i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; i++) {
            int xx = x + i;
            if (xx < 0 || xx >= width) continue;
                float val = inF[(yy * width + xx) * channels + c];
                local_mean += val;
                 count++;
            }
        }
        local_mean /= count;
```

Código 44: *Cálculo de la media del entorno*

Nosotros simplificaremos ligeramente este código pero en esencia se asegura de que los elementos que utiliza para calcular estos valores se encuentren dentro de la imagen

Teniendo la varianza local, calcula $\sigma_r(x)$ con el Código 45.

```
float sigma_r_adapt = 2.0f * sqrtf(local_var + 1e-6f);
```

Código 45: *Cálculo de $\sigma_r(x)$*

El factor **1e-6f** es necesario para evitar dividir entre 0 en caso de que todos los píxeles del entorno sean idénticos.

Finalmente calcula los valores de g_s y f_r con el Código 46.

```
float gs = gaussian(sqrtf(i*i + j*j), SIGMA_S);
float gr = gaussian(val - center_val, sigma_r_adapt);
```

Código 46: *Cálculo de g_s y f_r*

Llamando a la función **gaussian** definida previamente como ilustra el Código 47.

```
float gaussian(float x, float sigma) {
    return expf(-(x * x) / (2.0f * sigma * sigma));
}
```

Código 47: *Función gaussian*

Hecho esto calcula la suma (sum) y el término de normalización (wsum) y escribe el valor al output, asegurándose de que se encuentre entre 0 y 255, como podemos ver en el Código 48.

```
output[(y * width + x) * channels + c] = (unsigned char)(fminf(fmaxf(
   sum / wsum, 0.0f), 1.0f) * 255.0f);
```

Código 48: *Cálculo del output*

Aplicando este programa a la clásica imagen de Lenna tras aplicarle previamente un ruido Gaussiano, obtenemos el resultado ilustrado en la Figura 10.

Figura 10: *Original (ruido) - Procesada*

4.3. Implementación CUDA

Adaptar este código a CUDA ilustra una de sus mejores características, en la mayoría de casos es extremadamente simple.

Lo primero que hacemos es externalizar la normalización de los datos, de esta manera podemos tratar el problema como la transformación de una lista en otra con acceso secuencial, optimizando así el acceso a la memoria por parte de nuestra función. Vemos esta implementación en el Código 49.

```
__global__ void normalize(unsigned char *input, float *output){
  int ind = threadIdx.x +blockIdx.x*blockDim.x;
  int len = widthGPU*heightGPU*channelsGPU;
  if(ind < len){
  output[ind] = input[ind]/ 255.0f;
  }
}

...

normalize<<<(width * height * channels)/64+1, 64>>>(inputGPU, normGPU);
```

Código 49: *Función normalize sobre GPU*

En esta función hemos utilizado los parámetros **widthGPU**, **heightGPU** y **channelsGPU** sin introducirlos en los argumentos, esto se debe a que los hemos definido en memoria constante, como podemos observar en el Código 50.

```
__constant__ int widthGPU, heightGPU, channelsGPU;

...

cudaMemcpyToSymbol(widthGPU, &width, sizeof(int));
cudaMemcpyToSymbol(heightGPU, &height, sizeof(int));
cudaMemcpyToSymbol(channelsGPU, &channels, sizeof(int));
```

Código 50: *Definición y envio de memoria constante en la GPU*

Dado que todos los hilos de nuestro filtro deberán tener acceder a estas variables por igual, es el caso ideal para este tipo de memoria, tanto en esta función de normalización como en el filtro.

Dentro de la función filtro como tal simplemente alteramos ciertos parámetros de entrada y dividimos el bucle entre los hilos de la forma que ilustra el Código 51.

```
__global__ void bilateral_adaptive(float *input, unsigned char *output)
    {
     int x = threadIdx.x +blockIdx.x*blockDim.x;
     int y = threadIdx.y +blockIdx.y*blockDim.y;
     int c = blockIdx.z;
     ...
     }
```

Código 51: *Coordenadas del pixel asociado a un hilo*

En este caso utilizaremos bloques bidimensionales sobre un grid tridimensional, definido en el Código 52

```
dim3 threadSize(16,16);
dim3 blockSize(width/threadSize.x + 1,height/threadSize.y + 1, channels)
    ;
bilateral_adaptive<<<blockSize, threadSize>>>(normGPU, outputGPU);
```

Código 52: *Definición del grid y lanzamiento del kernel*

Finalmente, definimos la función gaussian para que opere sobre la GPU con un simple cambio, como vemos en el Código 53.

```
__device__ float gaussian(float x, float sigma) {
    return expf(-(x * x) / (2.0f * sigma * sigma));
}
```

Código 53: *Función gaussian para GPU*

El movimiento y alocado de memoria es idéntico al resto de nuestros códigos y puede verse en el código completo disponible en Github (enlace)

Una última consideración es la cantidad de registros que utiliza esta función, como hemos visto en el apartado correspondiente, si tenemos demasiados registros, puede limitar cuantos hilos ejecuta cada warp sobre un procesador. Para comprobar si esto es un problema que debemos solucionar, contamos el total de registros en la función (18) e introducimos los datos en la aplicación NVIDIA Nsight, viendo que en con 256 (16*16) hilos por bloque no tenemos ninguna limitación con 18 registros por hilo.

Tras este trabajo relativamente rápido si ejecutamos el programa obtenemos un resultado idéntico a la implementación lineal como cabe esperar, sin embargo, el tiempo de ejecución es sin duda muy diferente. Mientras que el código lineal tarda **140ms** en procesar la imagen, el código en CUDA tarda **0,2ms**, dramáticamente más rápido (ignorando la lectura y escritura de la imagen).

Para ver como escala ese aumento de velocidad con diferentes tamaños de imagen vamos a tomar la imagen de la Figura 11 [19] (cuya licencia permite su uso en este contexto) y reescalarla desde un 100 % de su tamaño original (3300x2400) hasta un 10 % en decrementos del 10 %, midiendo el tiempo de procesado en cada paso.

Figura 11: *Imagen de prueba (resolución original)*

Si ahora medimos el tiempo que tarda cada respectivo programa en procesar estas imágenes y lo ploteamos respecto al total de píxeles de cada imagen observamos que la diferencia de velocidad es enorme, del orden de 1000 veces más rápido observando la gráfica de speedup, que se calcula como:

$$S = \frac{T_{old}}{T_{new}}$$

Siendo T_{old} el tiempo de ejecución secuencial y T_{new} el tiempo de ejecución paralelo. Las gráficas se muestran en la Figura 12.

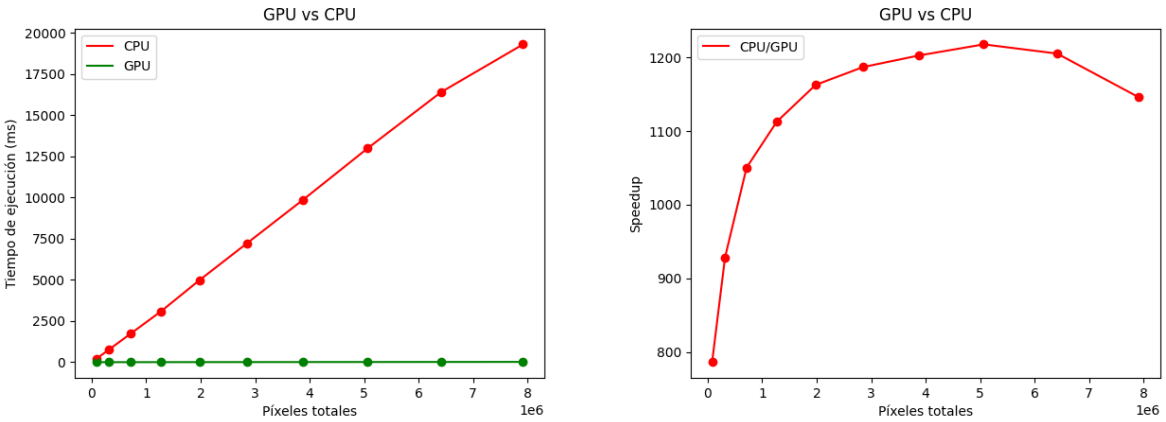

Figura 12: *Diferencia entre CPU y GPU*

Esto no es una simple cuestión de tardar menos en procesar una imagen, es algo crucial para muchas aplicaciones. En muchos caso el procesado de imágenes no es algo puntual sino

un proceso continuo, por ejemplo, no es inusual preprocesar imágenes antes de introducirla a un sistema de inteligencia artificial como puede ser un sistema de conducción automática. Para poner en perspectiva la diferencia entre procesar imágenes en la CPU y la GPU vamos a hablar en terminos de FPS, frames por segundo, en el caso de las televisiones el video se enseña a 24FPS, mientras que el estándar en el mundo de los videojuegos es entre 60-120FPS.

Digamos por ejemplo que queremos aplicar este filtro a un canal de televisión estandar, esto significa que tenemos 41,7ms para procesar cada una de estas imágenes, lo cual observando los gráficos significa que la resolución más grande que acepta nuestra CPU para poder realizar este proceso es menor a 330x240, lo cual para el usuario medio sería inaceptable. Por otro lado, nuestra GPU puede procesar imágenes de calidad 4k en menos de la mitad de este tiempo, pudiendo alcanzar hasta los 59FPS bajo las mismas condiciones.

En conclusión, si queremos procesar imágenes en tiempo real, la única posibilidad realista para la mayoría de aplicaciones es recurrir a la GPU, especialmente dado que como hemos visto el código no es particularmente complejo.

4.4. Implementaciones alternativas

Ahora que ya tenemos una implementación básica de nuestro programa, puede interesarnos intentar aplicar algunos de los métodos aprendidos para mejorar su eficiencia, sin embargo, como veremos a continuación aun siendo superiores en la teoría, muchas de estas implementaciones sufren de necesitar un preprocesado costoso o de no ofrecer mejoras sustanciales en la práctica.

4.4.1. Texturas

La primera implementación alternativa que veremos utilizará las texturas, en teoría este tipo de memoria es ideal para la aplicación, estando el acceso a memoria de cada hilo limitado a un entorno 2D centrado en su pixel asociado. Idealmente dividiríamos la imagen en tantas texturas 2D como canales tenga y trabajaríamos desde ahí. Esta división presenta el primer problema, dado que la forma en que el formato original alterna canales antes que píxeles es nefasto para el acceso de memoria de CUDA, estando la implementación de la división en el Código 54.

```
1 __global__ void normalizeAndSplit(unsigned char *input, float **output){
2   int x = threadIdx.x+blockDim.x*blockIdx.x;
3   int y = threadIdx.y+blockDim.y*blockIdx.y;
```

```
4   int c = blockIdx.z;
5
6   if (y < heightGPU) {
7           if (x < widthGPU) {
8       output[c][x+y*textWidth] = input[(x+y*widthGPU)*channelsGPU+c]/
     255.0f;
9   }}
10 }
```

Código 54: *Función de division de la imagen en texturas*

Hecho esto, las únicas otras diferencias son la creación de las texturas y el acceso a memoria de la función, sustituyendo

float val = input[(yy * widthGPU + xx) * channelsGPU + c];

por

float val = tex2D<float>(input[c], xx, yy);

El código completo esta disponible en Github (enlace)

Midiendo ahora los tiempos de ejecución y comparando con los originales vemos que la mejora del uso de texturas se ve completamente mitigado por este preprocesado, obteniendo una eficiencia menor como ilustra la Figura 13.

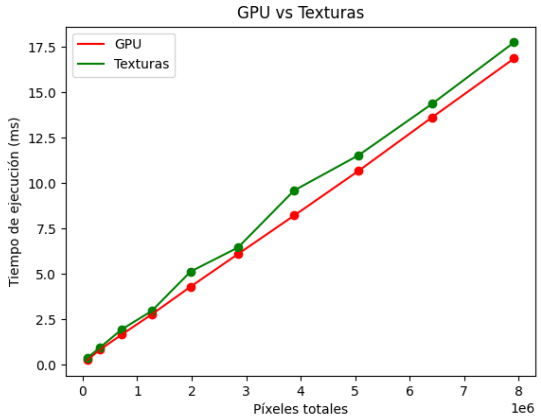

Figura 13: *Diferencia entre GPU básico y GPU con texturas*

Este resultado también es consecuencia de el esfuerzo de Nvidia por optimizar la memoria global en CUDA, reduciendo en gran parte la utilidad de herramientas como las texturas.

4.4.2. Memoria Compartida

La segunda y última implementación alternativa de este filtro utilizará la memoria compartida, guardando previamente los datos en una variable compartida de la que extraerá la información en los siguientes pasos de la función.

Aunque la siguiente implementación resulta en muchos accesos de memoria redundantes, diferentes pruebas con métodos alternativos proporcionas eficiencias equivalentes o peores, por lo que sencillamente elegimos la más simple. Podemos ver esta implementación en el Código 55

```
const int dimX = BLOCK_SIDE+2*KERNEL_RADIUS;
    const int dimY = BLOCK_SIDE+2*KERNEL_RADIUS;
    __shared__ float cache[dimX][dimY];
    if (y < heightGPU) {
        if (x < widthGPU) {
          for (j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++) {
                    int yy = y + j;
                    int yyCach = threadIdx.y+KERNEL_RADIUS + j;
                    if (yy < 0 || yy >= heightGPU) continue;
                    for (i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; i++) {
                        int xx = x + i;
                        int xxCach = threadIdx.x+KERNEL_RADIUS + i;
                        if (xx < 0 || xx >= widthGPU) continue;
                        cache[xxCach][yyCach] = input[(yy * widthGPU +
   xx) * channelsGPU + c];
    }}}}
    __syncthreads();
```

Código 55: *Guardado en memoria compartida*

En esta implementación también debemos definir las dimensiones de los bloques con #**define** para crear la variable compartida.

Hecho esto simplemente alteramos el acceso para cuadrar los valores, estando el código completo disponible en Github (enlace) y los cambios clave en el Código 56.

```
int xShar = threadIdx.x+KERNEL_RADIUS;
int yShar = threadIdx.y+KERNEL_RADIUS;
for (j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++) {
    int yy = y + j;
    if (yy < 0 || yy >= heightGPU) continue;
    int yyCach = yShar + j;
    for (i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; i++) {
        int xx = x + i;
```

```
9           if (xx < 0 || xx >= widthGPU) continue;
10          int xxCach = xShar + i;
11          float val = cache[xxCach][yyCach];
12          ...
13  }}
```

Código 56: *Bucle en memoria compartida*

Midiendo los tiempos de ejecución (con mayor número de iteraciones que los otros) este caso si observamos una mejora de velocidad, aunque muy pequeña como ilustran las gráficas de la Figura 14:

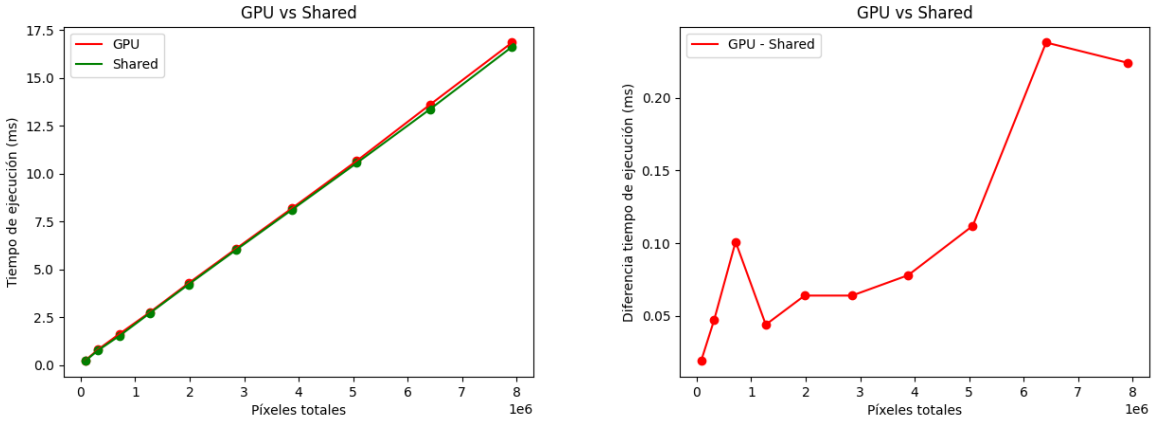

Figura 14: *Diferencia entre GPU básico y GPU con memoria compartida*

Si lo ilustramos en términos del speedup, podemos ver que la mejora comienza entorno al 9 % y eventualmente oscila entorno al 1 %, no presentando por tanto una gran diferencia como ilustra la Figura 15.

5. Conclusiones

En este trabajo hemos analizado el uso de CUDA para la computación de altas prestaciones mediante la implementación de diversos códigos que demuestran las capacidades de esta herramienta. Dado que el cálculo sobre GPUs es especialmente útil en el procesamiento de imágenes, se han implementado varias versiones del filtro bilateral, aprovechando distintos tipos de memoria disponibles en las GPUs. Estas implementaciones logran tiempos de ejecución significativamente menores en comparación con la versión secuencial, alcanzando un speedup notable.

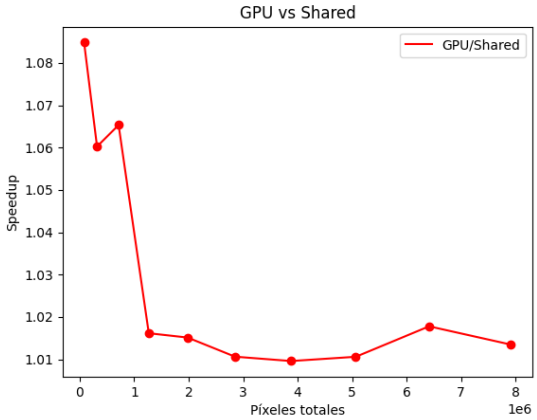

Figura 15: *Speedup entre GPU básico y GPU con memoria compartida*

A lo largo de este proceso hemos podido observar como mediante CUDA podemos crear código para GPU de forma cómoda, eficiente y limpia, presentando una manera extremadamente simple de paralelizar código sin entrar en detalles pero permitiendo al usuario experimentado personalizar aspectos más técnicos del proceso. Este código en GPU no solo es más eficiente gracias a estas posibilidades, sino que además deja libre la CPU para realizar otros procesos de forma simultánea.

Aun así, no es una herramienta libre de problemas. La documentación oficial del lenguaje es obtusa y requiere un conocimiento previo alto para poder extraer la información esencial, mientras que muchas de las fuentes de información de terceros están desactualizadas, utilizando comandos y procedimientos deprecados o proporcionando información obsoleta. Adicionalmente, pese a existir herramientas para aminorar su efecto el movimiento de memoria entre disco y GPU es un factor limitante a tener en cuenta, junto a las partes del proceso que nos vemos limitados a realizar desde la CPU.

En conclusión, pese a sus limitaciones CUDA es sin duda una herramienta esencial para todo programador que quiera profundizar en el mundo de la paralelización y, considerando su importancia en el mundo actual, es de esperar que continúe siéndolo en el futuro.

Referencias

[1] Powell, P. The history of Central Processing Unit (CPU). IBM. https://www.ibm.com/think/topics/central-processing-unit-history (Consultado el 17 de Junio 2025)

[2] Procesador Intel® CoreTM ultra 9 285K (caché de 36m, hasta 5,70ghz) - especificaciones de productos. Intel. https://www.intel.la/content/www/xl/es/products/sku/241060/intel-core-ultra-9-processor-285k-36m-cache-up-to-5-70-ghz/specifications.html (Consultado el 17 de Junio 2025)

[3] IBM POWER4. IBM. https://www.ibm.com/history/power (Consultado el 17 de Junio 2025)

[4] May 2025. Steam Hardware and Software Survey. https://store.steampowered.com/hwsurvey/cpus/?sort=pct (Consultado el 17 de Junio 2025)

[5] AmpereOne product brief. Ampere Computing®. https://amperecomputing.com/briefs/ampereone-family-product-brief

[6] Rreusser. RREUSSER/GLSL-FFT: GLSL setup for a fast fourier transform of two complex matrices. GitHub. https://github.com/rreusser/glsl-fft

[7] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for gpus. ACM SIGGRAPH 2004 Papers, 777–786. https://doi.org/10.1145/1186562.1015800

[8] Ian Buck. NVIDIA Blog. https://blogs.nvidia.com/blog/author/ian-buck/

[9] Oh, F. (2012, September 10). What is Cuda?. NVIDIA Blog. https://blogs.nvidia.com/blog/what-is-cuda-2/

[10] Sanders, J., and Kandrot, E., Cuda by example an introduction to general-purpose GPU programming. - includes index. Addison-Wesley. (2010).

[11] NVIDIA Corporation. (2025). Index. CUDA Runtime API::CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[12] Cook, S. (2013). Cuda Programming: A developer's guide to parallel computing with gpus. Morgan Kaufmann, an imprint of Elsevier.

[13] NVIDIA. Nvidia Nsight Systems. NVIDIA Developer. https://developer.nvidia.com/nsight-systems

[14] NVIDIA Corporation. (2025, October 2). CUDA C programming guide. Atomic Functions. CUDA C++ Programming Guide - CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

[15] Crovella, R. (2019, January 9). Cuda: How to create 2D texture object. Stack Overflow. https://stackoverflow.com/a/54100165

[16] Lecram. Lecram/gifenc: Small C GIF encoder. GitHub. https://github.com/lecram/gifenc

[17] Tomasi, C., and Manduchi, R. (1998). Bilateral filtering for gray and color images. Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271), 839–846. https://doi.org/10.1109/iccv.1998.710815

[18] Nothings. Nothings/STB: STB single-file public domain libraries for C/C++. GitHub. https://github.com/nothings/stb

[19] Garageband. (2017). Imagen de un cuadro con múltiples colores. pixabay. Retrieved November 15, 2025, from https://pixabay.com/es/photos/vistoso-cuadro-antecedentes-resumen-2468874/.

A. Especificaciones GPU

En el desarrollo de este trabajo los códigos se han ejecutado sobre una tarjeta gráfica NVIDIA GeForce RTX 4060 Ti con 16GB de VRAM. En cuanto a los detalles de las especificaciones, el Código 57 nos proporciona la información que hemos utilizado a lo largo del trabajo.

```
#include <stdio.h>
__global__ void kernel( void ) {
}
int main( void ) {
cudaDeviceProp prop;
int count;
cudaGetDeviceCount( &count );
for (int i=0; i< count; i++) {
cudaGetDeviceProperties( &prop, i );

printf( "--- General Information for device %d ---\n", i );
printf( "Name:%s\n", prop.name );
printf( "Compute capability: %d.%d\n", prop.major, prop.minor );
printf( "Clock rate:%d\n", prop.clockRate );
printf( "Device copy overlap:" );
if (prop.deviceOverlap)
printf( "Enabled\n" );
else
printf( "Disabled\n" );
printf( "Kernel execition timeout :" );
if (prop.kernelExecTimeoutEnabled)
printf( "Enabled\n" );
else
printf( "Disabled\n" );

printf("Concurrent kernels:%d\n",prop.concurrentKernels);

printf( "--- Memory Information for device %d ---\n", i );
printf( "Total global mem: %ld\n", prop.totalGlobalMem );
printf( "Total constant Mem: %ld\n", prop.totalConstMem );
printf( "Max mem pitch: %ld\n", prop.memPitch );
printf( "Texture Alignment: %ld\n", prop.textureAlignment );
printf( "--- MP Information for device %d ---\n", i );
printf( "Multiprocessor count:%d\n",prop.multiProcessorCount );
printf( "Max blocks per multiprocessor:%d\n",prop.
   maxBlocksPerMultiProcessor);
```

```
36 printf( "Shared mem per mp:%ld\n", prop.sharedMemPerBlock );
37 printf( "Registers per mp:%d\n", prop.regsPerBlock );
38 printf( "Threads in warp:%d\n", prop.warpSize );
39 printf( "Max threads per multiprocessor:%d\n",prop.
       maxThreadsPerMultiProcessor );
40 printf( "Max threads per block:%d\n",prop.maxThreadsPerBlock );
41 printf( "Max thread dimensions: (%d, %d, %d)\n", prop.maxThreadsDim[0],
       prop.maxThreadsDim[1], prop.maxThreadsDim[2] );
42 printf( "Max grid dimensions: (%d, %d, %d)\n", prop.maxGridSize[0], prop
       .maxGridSize[1], prop.maxGridSize[2] );
43 printf( "Max 2D texture dimensions: (%d, %d, %d)\n",  prop.
       maxTexture2DLinear[0], prop.maxTexture2DLinear[1], prop.
       maxTexture2DLinear[2] );
44
45 printf( "\n" );
46 }
47 return 0;
48 }
```

Código 57: *Propiedades de la GPU*

Obteniendo al compilar y ejecutar la siguiente información:

— General Information for device 0 —

Name:NVIDIA GeForce RTX 4060 Ti

Compute capability: 8.9

Clock rate:2595000

Device copy overlap:Enabled

Kernel execition timeout :Enabled

Concurrent kernels:1

— Memory Information for device 0 —

Total global mem: 16713449472

Total constant Mem: 65536

Max mem pitch: 2147483647

Texture Alignment: 512

— MP Information for device 0 —

Multiprocessor count:34

Max blocks per multiprocessor:24

Shared mem per mp:49152

Registers per mp:65536

Threads in warp:32

Max threads per multiprocessor:1536

Max threads per block:1024

Max thread dimensions: (1024, 1024, 64)

Max grid dimensions: (2147483647, 65535, 65535)

Max 2D texture dimensions: (131072, 65000, 2097120)

Además, para compilar el código hemos utilizado:

- **gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0**: Para el codigo de C.
- **Cuda compilation tools, release 12.0, V12.0.140**: Para el código de CUDA.

B. Tabla de tiempos de ejecución del filtro bilateral

En los apartados 4.3 y 4.4 hemos medido los tiempos de ejecución de la diferentes implementaciones del filtro bilateral. Los archivos de código e imágenes de prueba que hemos utilizado están disponibles en Github(enlace). En la siguiente table recogemos los datos obtenidos durante estas mediciones (el tiempo se ha medido en ms):

%	**pixeles totales**	**CPU**	**GPU Base**	**GPU Text**	**GPU Comp**
100	7920000	19311,76	16,86	17,74	16,63
90	6415200	16409,88	13,62	14,36	13,38
80	5068800	13003,53	10,68	11,54	10,57
70	3880800	9870,17	8,21	9,59	8,13
60	2851200	7227,97	6,09	6,45	6,03
50	1980000	4988,98	4,29	5,12	4,23
40	1267200	3078,90	2,77	2,96	2,72
30	712800	1730,18	1,65	1,93	1,55
20	316800	767,98	0,83	0,94	0,78
10	79200	191,10	0,24	0,36	0,23

Tabla 1: *Tiempos de ejecución según número de píxeles*

GPU Text: GPU utilizando texturas

GPU Comp: GPU utilizando memoria compartida

C. Detalles del desarrollo del trabajo

Todo el código elaborado para el trabajo se encuentra disponible en el siguiente repositorio de Github: https://github.com/mmaciatitos-gif/TFM_CUDA_C/tree/main

Tiempo dedicado a la realización del trabajo:

Tarea	Tiempo (horas)
Recopilación de materiales	20
Estudio de Bibliografía	20
Elaboración del Código	70
Redacción de la Memoria	30
Total	150

Asignatura	Páginas	Descripción
Computación en Paralelo para el Cálculo Científico	General	Conceptos generales de computación en paralelo y programación en C

Tabla 2: *Asignaturas relacionadas con el trabajo*