Resolución de ecuaciones en GPU
Solving equations in GPU

Trabajo de Fin de Grado
Curso 2023–2024

Autor
Noelia Barranco Godoy

Director
Ana María Carpio Rodríguez

Doble grado en Matemáticas e Informática
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

Resolución de ecuaciones en GPU
Solving equations in GPU

Trabajo de Fin de Grado en Matemáticas

Autor
Noelia Barranco Godoy

Director
Ana María Carpio Rodríguez

Convocatoria: Septiembre 2024

Doble grado en Matemáticas e Informática
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

22 de julio de 2024

Resumen

Resolución de ecuaciones en GPU

En este trabajo pretendemos comparar las diferencias respecto a eficacia entre resolver ecuaciones diferenciales mediante algoritmos tradicionales y algoritmos implementados sobre GPU's.

En particular, implementaremos el código utilizando el lenguaje de programación Python, y la api de Nvidia CUDA

Palabras clave

ecuaciones, diferencias finitas, computación, gpu, ecuación del calor, ecuación de ondas, ecuación de Laplace

Abstract

Solving equations in GPU

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Keywords

10 keywords max., separated by commas.

Capítulo 1

Introducción

Resumen: En este capítulo pretendemos introducir los objetivos de este trabajo

1.1. Motivación

Desde el inicio de la computación, se han desarrollado métodos numéricos para aproximar soluciones de ecuaciones que no podemos resolver de manera analítica (o cuya solución exacta no se conoce). Con el auge de la computación en GPU, que permite computar los datos en paralelo, se pueden implementar estos mismos métodos de formas más eficientes para lograr mejores resultados.

1.2. Objetivos

En este trabajo pretendemos estudiar la implementación de métodos de diferencias finitas en la GPU y su mejora de eficiencia en las ecuaciones de *Laplace*, del calor y de ondas.

1.3. Nociones generales

Primero de todo necesitaremos hacer un estudio matemático sobre las ecuaciones diferenciales que trataremos. Todos los algoritmos que vamos a hacer están basados en el método de las diferencias finitas, que consiste en hacer una aproximación de las derivadas por un cociente incremental en puntos cercanos. Para hacer esto, necesitaremos definir sobre todos los problemas una malla discreta de puntos, y serán en estos donde hallemos soluciones aproximadas de las soluciones.

1.3.1. Notación

Las mallas son una discretización del dominio de las funciones, que en caso de las lineales será bidimensional y en el caso de las planas será tridimensional (ya que siempre hay que añadir el tiempo como dimensión). En todos los casos llamaremos Δt, Δx y Δy al espacio entre puntos de la malla en cada dimensión.

Los diferentes algoritmos que implementemos tomarán como entrada las condiciones iniciales del problema de valor inicial, así como cualquier parámetro que requiera la ecuación. Además también tomarán el intervalo sobre el que trabajamos (la forma de este variará en cada problema) y el número de puntos que queremos en cada dimensión de la malla.

Además, siguiendo con el convenio establecido por Isaacson y Keller (1966), denotaremos los cocientes incrementales (omitiendo x, y y t cuando no haya lugar a confusión)

$$\begin{cases} u_x \equiv \frac{u(x+\Delta x,y,t)-u(x,y,t)}{\Delta x} \\ u_{\overline{x}} \equiv \frac{u(x,y,t)-u(x-\Delta x,y,t)}{\Delta x} \\ u_{\hat{x}} \equiv \frac{1}{2}[u_x(x,y,t) - u_{\overline{x}}(x,y,t)] \end{cases}$$

también denotamos los puntos de la malla como $x_i = x_0 + i\Delta x$, $y_j = y_0 + j\Delta y$ y $t_k = t_0 + k\Delta t$, y el valor de la función en estos puntos como $u(x_i, t_k) \equiv u_{i,j}$ y $u(x_i, y_j, t_k) \equiv u^k_{i,j}$

1.4. Plan de trabajo

Para realizar el estudio, los lenguajes de programación que utilizaremos serán CUDA (una extensión de C++ que permite la ejecución de funciones -llamadas *kernels*- en la GPU) y Python, que tiene una librería llamada pycuda para ejecutar código CUDA. Además, todos los programas aquí mostrados y los resultados obtenidos serán ejecutados en la misma máquina, con las siguientes especificaciones, no obstante, los programas están pensados para poder ejecutarse en cualquier máquina[1]

Procesador Intel© Core™ i5-10400F CPU @ 2.90GHz × 6

RAM 15.5 GiB

GPU NVIDIA Corporation GA104 [GeForce RTX 3070]

SO Linux Mint 21.3 Cinnamon

En el capítulo 2 explicaremos cómo ejecutar programas en GPU usando la librería pycuda (para lo que primero debemos de entender como programar en CUDA), seguido de un par de programas para familiarizarnos con las librerías y ver que, en efecto, pueden acelerar los tiempos de ejecución.

{**TODO TODO TODO:** Completar con qué más hago en el TFG}

[1] Si se desea utilizar el script "generateMod.py.en un sistema basado en Windows o MAC, podría ser necesario hacer unos pocos cambios para adaptarse a las distintas formas de nombrar las rutas en estos sistemas.

Capítulo 2

Computación en GPU con CUDA y pycuda

Resumen: En este capítulo se pretenden introducir los conceptos básicos de la programación en GPU, en particular en el lenguaje CUDA C++ utilizado desde la librería de Python pycuda.

2.1. Bases de la programación en GPU

Las GPUs son componentes hardware del ordenador pensados para acelerar el procesamiento de los gráficos. Esto se hace aprovechando que la mayoría de los cómputos que se requieren para procesar estos se puede ejecutar de manera simultánea en paralelo.

Dada la naturaleza de las tarjetas gráficas, resulta natural el utilizarlas para implementar algoritmos que puedan beneficiarse de este paralelismo. Con ese fin, NVIDIA desarrolló el lenguage de programación CUDA[1], una extensión de C/C++ que permite la definición de *kernels*.

Los *kernels* son funciones de C/C++ diseñadas para ejecutarse en varios hilos al mismo tiempo en la GPU (a la que llamaremos *dispositivo* en este contexto). Estos kernels serán luego ejecutados desde la CPU, que en este contexto llamaremos *host*.

2.2. Programación en pycuda

Como adelantamos en la sección 1, utilizaremos pycuda para poder ejecutar código CUDA en la GPU desde un programa convencional en Python. Este será el proceso para hacer un programa:

1. Generar archivos con la extensión .cu que contengan la definición de las funciones que vayamos a ejecutar en la GPU, escritas en CUDA. Este será el único

[1]Puede consultarse el manual completo de CUDA en este *enlace*

código que escribamos en este lenguaje de programación, todo lo demás estará escrito en Python.

2. Crear un programa en Python, que será el que ejecutemos. Éste será el que se encargue de compilar y llamar (mediante las funciones de pycuda) a las funciones definidas en el apartado anterior. Para hacer esto, utilizaremos el programa generateMod.py (ver A.1). Además del programa como tal, tendremos que utilizar unas funciones de pycuda para reservar memoria y llevar los datos al dispositivo (para ver más sobre la gestión de memoria en dispositivo, ver 2.3.1).

2.3. Programación en CUDA

Aunque no vayamos a escribir mucho código en este lenguaje propiamente, necesitamos entender bien cómo funciona para evitar errores. La única diferencia que nos encontramos con C/C++ en la sintaxis, es que todos los *kernels* que definamos tienen que tener uno de los siguientes identificadores, que describe desde dónde se van a llamar y dónde se van a ejecutar:

device Funciones que van a ser llamadas y ejecutadas desde el dispositivo (la GPU)

global Funciones que van a se llamadas desde la CPU pero ejecutadas en el dispositivo. Cuando llamemos a estas funciones les pasaremos como meta-parámetro el número de veces que se van a ejecutar.

host Son funciones habituales de C++. Dado que vamos a utilizar CUDA a través de pycuda, no utilizaremos este tipo de funciones, ya que toda la programación en CPU la haremos en Python.

2.3.1. Gestión de memoria

Es importante comprender que el dispositivo y el host son unidades de computación diferentes. Esto significa que no comparten espacio de direcciones de memoria y, por tanto, tenemos que proceder de manera distinta dependiendo si estamos usando variables del dispositivo o del host.

A pesar de la necesidad de entender el funcionamiento de la memoria, no usaremos las funciones de gestión de memoria de CUDA, ya que la librería pycuda tiene sus propias funciones para hacer esto de manera un poco más cómoda.

Memoria en el host

Para trabajar con la memoria en la CPU se utilizarían las funciones habituales de C/C++ como *malloc* (para reservar memoria), pero como estamos trabajando sobre Python, no tenemos que preocuparnos por eso.

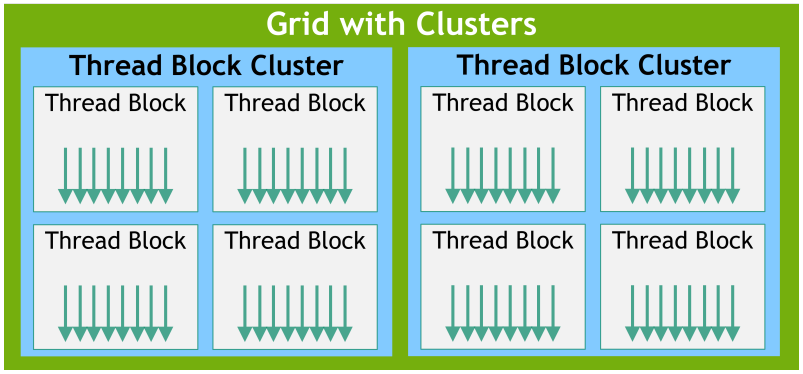

Figura 2.1: Cuadrícula de bloques de hilos

Memoria en el dispositivo

Al crear la memoria para el dispositivo desde el host, no podemos emplear las funciones habituales, pues estas no reservan memoria en la GPU. Existen funciones específicas para esto como *cudaMalloc*.

Compartir memoria entre dispositivo y host

Para esto, usamos unas funciones que nos proporciona pycuda, *memcpy_ htod* y *memcpy_ dtoh*, que nos permiten copiar memoria desde el host al dispositivo y en dirección contraria respectivamente.

2.3.2. Hilos y bloques

Al ejecutar un kernel, tenemos que especificar cuantos hilos ejecutarán esa función, los hilos se organizan en bloques, que a su vez se organizan en una cuadrícula (grid)[2] como podemos observar en la imagen 2.1

Un bloque es un conjunto de hilos (que se distribuyen en tres dimensiones) que, como máximo, puede tener 1024 hilos diferentes por motivos de implementación en memoria (podemos tener un bloque de 1024x1x1 o 256x2x2 por ejemplo, pero no de 256x4x2, ya que el límite se refiere a la cantidad total de hilos, no al máximo en cada dimensión).

Podemos tener tantos bloques como queramos, organizados a su vez en la cuadrícula (también tridimensional). En cada hilo podemos acceder al índice de su bloque y del hilo dentro del bloque con *blockIdx.dim* y *threadIdx.dim* respectivamente, siendo dim la dimensión, osea x, y o z.

Aunque a nivel de usuario pueda parecer enrevesada esta distribución, esto está estrechamente relacionado con la implementación y la memoria compartida como podemos observar en la imagen 2.2. Todos los hilos tienen su propia memoria privada y una memoria compartida entre los demás hilos de su bloque, pero para compartir memoria con otros bloques hay que utilizar otros mecanismos que pueden afectar

[2]En las últimas versiones también podemos tener Thread Block Clusters para agrupar bloques, pero esos no los trataremos en este trabajo.

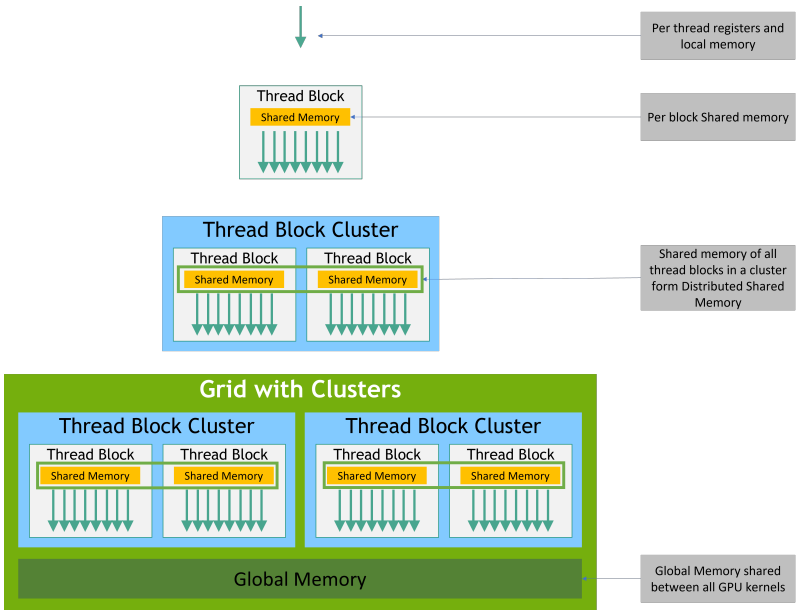

Figura 2.2: Memoria compartida entre elementos

al rendimiento, luego implementar los hilos de manera adecuada puede hacer más eficientes los programas.

2.4. Ejemplos

Procedemos ahora a hacer un par de ejemplos sencillos que nos permitan familiarizarnos con estos conceptos.

2.4.1. Hola Mundo

Como es costumbre a la hora de programar, lo primero es hacer un "hola mundo", un programa que escriba en la consola la frase "hola mundo". Como el objetivo es poner de manifiesto que las cosas se están ejecutando varias veces en la GPU, escribiremos el texto dos veces.

• En A.2 vemos un ejemplo de uso de *generateMod.py*, así como las funciones para reservar y copiar memoria en pycuda. Es importante destacar que a la hora de llamar al kernel desde la CPU, le tenemos que indicar qué forma va a tener cada uno de los bloques (osea, el número de hilos y cómo van a estar repartidos a lo largo de sus dimensiones).

• En A.3 podemos ver la implementación del hola mundo en CUDA, con el uso de la etiqueta **__global__** mencionada anteriormente.

• En A.4 vemos que la salida es la esperada.

2.4.2. Suma de vectores

Habiendo tenido ya nuestro primer contacto con pycuda, vamos ahora a poner de manifiesto la mejora de rendimiento que se puede conseguir. Vamos a realizar la suma de dos vectores de tamaños incrementalmente grandes, primero en CPU y luego en GPU, y vamos a comparar los tiempos que tarda en hacer ambas cosas.

• En A.5 podemos ver un ejemplo más complejo de código Python usando pycuda. La función *add_random_vects* genera dos vectores de números aleatorios y los suma, primero en la CPU, y luego en GPU, midiendo el tiempo de ambos. Hay que destacar un par de cosas importantes:

Por un lado, hay que hacer la gestión de memoria, osea reservar memoria en GPU y copiar los datos con las funciones de pycuda. Por otro lado hay que gestionar el tamaño de bloque. Recordemos que los bloques pueden tener a lo sumo 1024 hilos, mientras que las grids pueden ser arbitrariamente grandes[3], esto implica que podríamos simplemente ejecutar un hilo por bloque y tantas mallas como el tamaño del vector, pero si recordamos la imagen 2.2 podemos observar que los hilos de cada bloque comparten memoria local, por lo que si hacemos el máximo uso posible de los bloques (osea tratar con bloques de 1024 hilos), haremos un menor uso de la memoria y, por tanto, obtendremos resultados notablemente mejores.

Por último, comprobamos que las sumas coincidan y mostramos el incremento de eficiencia. Al ejecutar este script, simplemente llamamos a la función para valores de n entre 1 y 10^{10}

• En A.6 el código es bastante inmediato, calculamos el índice al que le corresponde nuestro hilo concreto y hacemos la suma. Solo hay una sutileza, y es que, en el último bloque que utilicemos, probablemente algunos hilos estén trabajando posiciones no válidas (porque todos los bloques tienen la misma cantidad de hilos, luego hay más hilos que posiciones en el vector). Para no acceder a posiciones de memoria posiblemente inválidas, simplemente añadimos a los datos de entrada el tamaño del vector y, si el hilo tiene un índice superior al tamaño del vector, no hacemos nada.

• En A.7 se pone de manifiesto la mejora en la eficiencia, y es que cuando el tamaño del vector es suficientemente grande, el algoritmo es unas 10 veces más rápido.

[3]En realidad hay un límite de tamaño dependiente del hardware, pero es tan grande que puede desestimarse

Capítulo 3

Ecuación del calor

Resumen: En este capítulos desarrollaremos dos métodos numéricos, basado en las diferencias finitas, para aproximar la solución a la ecuación del calor en una y dos dimensiones.

3.1. Introducción

En *Théorie Analytique de la Chaleur*, Fourier define la ley de la conducción térmica

$$\mathbf{q} = -k\nabla T$$

3.2. Caso lineal

En el caso unidimensional, mediante un cambio de variable podemos omitir la constante k (Cannon, 1984, ver), por lo que la ecuación podemos reducirla a

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \tag{3.1}$$

Pueden pedirse diferentes condiciones iniciales para asegurar la existencia y unicidad de 3.1, pero nosotros utilizaremos en concreto las condiciones

$$u(x,0) = f(x) \qquad -\infty \leq x \leq \infty \tag{3.2}$$

El problema de valor inicial que hemos definido tiene como dominio un rectángulo con uno de sus lados abierto hacia el infinito. Si fijamos $T > 0$ tenemos ahora un rectángulo con el que trabajaremos. Definimos pues B_T como la frontera del rectángulo y D_T el interior de este, y por último definimos $D = D_T \cup B_T$.

3.2.1. Existencia y unicidad

Antes de estudiar la existencia y unicidad de la solución, necesitamos definir un concepto.

Definicion 3.2.1 (Función continua a trozos). *Una función es continua a trozos si es continua en todos sus puntos salvo en una cantidad finita de puntos.*

Ahora enunciamos unas condiciones para la unicidad de las soluciones

Teorema 3.2.1 (Unicidad). *Sean u y v soluciones de la ecuación 3.1 en D_T continuas en D, si $u = v$ en B_T entonces $u = v$ enD*

Teorema 3.2.2 (Unicidad extendida). *Sean u y v soluciones de la ecuación 3.1 en D_T continuas a trozos en D con una cantidad finita de discontinuidades acotadas, si $u = v$ en B_T (excepto los puntos de discontinuidad) entonces $u = v$ en D*

Estos teoremas nos dicen que bastan con comprobar que todas las soluciones coinciden en los límites del rectángulo para ver que la solución es única.

Demostración. Ver Cannon (1984), p.22 □

Teorema 3.2.3 (Existencia y unicidad). *Sean f, α y β funciones continuas a trozos, la función*

$$u(x,t) = \int_a^b \theta(x-\xi,t) - \theta(x+\xi,t) f(\xi) d\xi - 2\int_0^t \frac{\partial \theta}{\partial x}(x, t-\tau)\alpha(\tau)d\tau + 2\int_0^t \frac{\partial \theta}{\partial x}(x-1, t-\tau)\beta(\tau)d(\tau) \quad (3.3)$$

donde $\theta(x,t)$ y $K(x,t)$ se definen como

$$\theta(x,t) = \sum_{m=-\infty}^{\infty} K(x+2m,t) \qquad t > 0$$

$$K(x,t) = \frac{e^{\frac{-x^2}{4t}}}{\sqrt{4}} \qquad t > 0$$

Es la única solución acotada del problema del valor inicial

$$\begin{cases} \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} & a < x < b, \quad 0 < t, \\ u(x,0) = f(x), & a < x < b, \\ u(a,t) = \alpha(t), & 0 < t, \\ u(b,t) = \beta(t), & 0 < t. \end{cases} \quad (3.4)$$

Demostración. Puede verse en Cannon (1984) que en efecto 3.3 es solución de la ecuación del calor, por lo que solo tenemos que preocuparnos por la unicidad. Esto es inmediato por el teorema 3.2.3, ya que las condiciones iniciales fijan el valor de cualquier solución en B_T. □

Ahora podemos asegurar que 3.4 tiene una única solución, por lo que podemos proceder a aproximarla con un método de diferencias finitas.

3.2.2. Aproximación de la solución

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

Miguel de Cervantes Saavedra

Cannon, J. R. *The One-dimensional heat equation*. Reading, Massachusetts [etc] : Addison-Wesley, 1984. ISBN 0201135221.

Fourier, J. B. J. *Théorie Analytique de la Chaleur*. Cambridge University Press, 1822. ISBN 9780511693229.

Isaacson, E. y Keller, H. B. *Analysis of Numerical Methods*. John Wiley and Sons, 1966. ISBN 9780471428657.

Apéndice A

Código

```
import pycuda.autoinit
from pycuda.compiler import SourceModule
import os, sys

def search_dir(path, files):
    files += [f for f in os.listdir(path) if os.path.isfile(path + os.sep + f) and f[-3:]==".cu"]
    for d in [path + os.sep + d for d in os.listdir(path) if os.path.isdir(path+os.sep+d) and d != "__pycache__"]:
        search_dir(d, files)


def init(files=None):
    """
    Generates SourceModule with the code and autoinits pycuda.

    :param [str] files: The files from which the code is generated (if none, all the files in the directory (recursive) are considered)
    :return: The SourceModule
    """
    if files == None:
        files = []
        search_dir(os.getcwd(), files)
    code = ""
    for file in files:
        code += open(os.getcwd() + os.sep + "CUDA" + os.sep + file, 'r').read() + '\n'
    return SourceModule(code)



if __name__ == "__main__":
    files = []
    search_dir(os.getcwd(), files)
    print(files)
```

Listing A.1: Módulo para generar SourceModule a partir de los archivos

```python
import generateMod as generateMod
import pycuda.driver as cuda
import numpy as np
import sys


mod = generateMod.init(["HelloWorld.cu"])
hello_world = mod.get_function("hello_world")
a = [1.0, 2.0]
a = np.asarray(a).astype("float32") #Esta es la memoria que gestionaremos en host, arrays de numpy
a_gpu = cuda.mem_alloc(a.nbytes) #Reservamos la memoria en GPU
cuda.memcpy_htod(a_gpu, a)
hello_world(a_gpu, block = (2,1,1))
```

Listing A.2: Hola Mundo (Código Python)

```
__global__ void hello_world(float* a){
    int id = threadIdx.x;
    printf("Hello World, my id is %i and my number is \"%f\".\n", id, a[id]);
}
```

Listing A.3: Hola Mundo (Código CUDA)

```
Hello World, my id is 0 and my number is "1.000000".
Hello World, my id is 1 and my number is "2.000000".
```

Listing A.4: Hola Mundo (Salida)

```python
import numpy as np
import generateMod as generateMod
import pycuda.driver as cuda
from timeit import default_timer as timer

mod = generateMod.init(["vectorAdd.cu"])
vectorAdd = mod.get_function("vectorAdd")
def add_random_vects(n, out):
    #Tiempo en CPU (Contando reservar memoria)
    start = timer()
    a = np.random.randn(n).astype(np.float32)
    b = np.random.rand(n).astype(np.float32)
    c2 = np.zeros(n).astype(np.float32)
    np.add(a,b,out=c2)
    CPUt = timer()-start
    #Tiempo en GPU (Contando reservar y copiar los datos al dispositivo)
    start = timer()
    c1 = np.zeros(n).astype(np.float32)
    a_gpu = cuda.mem_alloc(a.nbytes)
    b_gpu = cuda.mem_alloc(b.nbytes)
    c_gpu = cuda.mem_alloc(c1.nbytes)
    n_gpu = cuda.mem_alloc(np.int32(n).nbytes)
    cuda.memcpy_htod(a_gpu, a)
    cuda.memcpy_htod(b_gpu, b)
```

```
    b = n if n < 1024 else 1024 #el bloque va a ser de 1024 (siempre que n > 1024)
    g = n // 1024 + 1 #Tantos grids como haga falta
    vectorAdd(a_gpu, b_gpu, c_gpu, np.int32(n), grid=(g,1,1), block=(b,1,1))
    cuda.memcpy_dtoh(c1, c_gpu)
    GPUt = timer() - start

    if (c1 == c2).all():
        text = f"""------------------N = {n}------------------
        Los resultados coinciden:
        Tiempo en GPU {GPUt}
        Tiempo en CPU: {CPUt}
        La GPU es un {round(CPUt/GPUt*100,1)}% mas rapida"""
        print(text, file=out)
        print(text)
    else:
        print("Ha habido un error en el computo en GPU.\n", file=out)
        print("Ha habido un error en el computo en GPU.\n")
```

Listing A.5: Suma de vectores (Código Python)

```
__global__ void vectorAdd(float* a, float* b, float* c, int n){
    int i = blockIdx.x * 1024 + threadIdx.x; //Calculamos el indice teniendo en cuenta que cada bloque tiene 1024 elementos
    if(i<n){
        c[i] = a[i] + b[i];
    }
}
```

Listing A.6: Suma de vectores (Código CUDA)

```
------------------N = 1------------------
        Los resultados coinciden:
        Tiempo en GPU 0.00034869599949161056
        Tiempo en CPU: 6.595699960598722e-05
        La GPU es un 18.9% mas rapida
------------------N = 100------------------
        Los resultados coinciden:
        Tiempo en GPU 0.00011117699978058226
        Tiempo en CPU: 2.217499968537595e-05
        La GPU es un 19.9% mas rapida
------------------N = 10000------------------
        Los resultados coinciden:
        Tiempo en GPU 0.00012266700105101336
        Tiempo en CPU: 0.00028031600049871486
        La GPU es un 228.5% mas rapida
------------------N = 1000000------------------
        Los resultados coinciden:
        Tiempo en GPU 0.0037263039994286373
        Tiempo en CPU: 0.03368916900035401
        La GPU es un 904.1% mas rapida
------------------N = 100000000------------------
        Los resultados coinciden:
        Tiempo en GPU 0.2606051149996347
```

```
        Tiempo en CPU: 3.1032862800002476
        La GPU es un 1190.8% mas rapida
```

Listing A.7: Suma de vectores (Salida)

```python
import numpy as np
import generateMod as generateMod
import pycuda.driver as cuda
from timeit import default_timer as timer

mod = generateMod.init(["vectorAdd.cu"])
vectorAdd = mod.get_function("vectorAdd")
def add_random_vects(n, out):
    #Tiempo en CPU (Contando reservar memoria)
    start = timer()
    a = np.random.randn(n).astype(np.float32)
    b = np.random.rand(n).astype(np.float32)
    c2 = np.zeros(n).astype(np.float32)
    np.add(a,b,out=c2)
    CPUt = timer()-start
    #Tiempo en GPU (Contando reservar y copiar los datos al dispositivo)
    start = timer()
    c1 = np.zeros(n).astype(np.float32)
    a_gpu = cuda.mem_alloc(a.nbytes)
    b_gpu = cuda.mem_alloc(b.nbytes)
    c_gpu = cuda.mem_alloc(c1.nbytes)
    n_gpu = cuda.mem_alloc(np.int32(n).nbytes)
    cuda.memcpy_htod(a_gpu, a)
    cuda.memcpy_htod(b_gpu, b)
    b = n if n < 1024 else 1024 #el bloque va a ser de 1024 (siempre que n > 1024)
    g = n // 1024 + 1 #Tantos grids como haga falta
    vectorAdd(a_gpu, b_gpu, c_gpu, np.int32(n), grid=(g,1,1), block = (b,1,1))
    cuda.memcpy_dtoh(c1, c_gpu)
    GPUt = timer() - start

    if (c1 == c2).all():
        text = f"""------------------N = {n}------------------
        Los resultados coinciden:
        Tiempo en GPU {GPUt}
        Tiempo en CPU: {CPUt}
        La GPU es un {round(CPUt/GPUt*100,1)}% mas rapida"""
        print(text, file=out)
        print(text)
    else:
        print("Ha habido un error en el computo en GPU.\n", file=out)
        print("Ha habido un error en el computo en GPU.\n")

if __name__ == "__main__":
    with open("./out/vectorAdd.out", "w") as out:
        for n in [10**(2*e) for e in range(5)]:
            add_random_vects(n, out)
```

Listing A.8: Método numérico para la ecuación del calor en 1D (Código Python)

Este texto se puede encontrar en el fichero Cascaras/fin.tex. Si deseas eliminarlo, basta con comentar la línea correspondiente al final del fichero TFGTeXiS.tex.

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.

Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes