UNIVERSIDAD DE GRANADA

Facultad de Ciencias

GRADO EN FISICA

TRABAJO FIN DE GRADO

IMPLEMENTACIÓN DE MÉTODOS EN DIFERENCIAS FINITAS EN GPUs

Presentado por:
D. JUAN JOSÉ SALAZAR LÓPEZ

Curso Académico 2021/2022

Resumen

En el siguiente TFG se procede a presentar las ventajas de aplicar el método de diferencias finitas en el dominio del tiempo (FDTD) en GPU, a diferencia del modo de aplicación usual el cual se lleva a cabo usando la potencia de cálculo de las CPUs. La principal diferencia radica en el poder de paralelización de las GPUs, el cual nos proporciona al implementar el método en $\mathbb{R}^2$ una disminución en el tiempo de cálculo de las ecuaciones de Maxwell de aproximadamente un orden de magnitud.

Abstract

In the following TFG it is shown the advantages of implementing the finite difference time domain method (FDTD) in GPU, instead of using the calculus power of CPUs. The principal difference is the parallelization power of the GPUs which allows us to decrease the calculus time to solve the Maxwell equations in about one order of magnitude when we implement the method in $\mathbb{R}^2$.

Índice

1 Introducción

El método numérico de las diferencias finitas es usado en muchos campos para la resolución de ecuaciones diferenciales de primer orden. En particular, el método de la diferencia central finita cobra especial importancia en la electrodinámica computacional donde se emplea para resolver las ecuaciones de Maxwell en el dominio del tiempo, discretizando tiempo y espacio para poder implementar la solución en un algoritmo. Actualmente su uso está muy extendido ya que posibilita el estudio de la propagación de señales electromagnéticas por diversos medios. En particular puede usarse para determinar la radiación que actúa sobre una cabeza humana producida por un teléfono móvil o la propagación de una señal en el uso de radares marítimos entre otras aplicaciones.

Desde que se desarrolló el método y se generaron los primero algoritmos se ha hecho uso del poder de cómputo que nos proporcionan las CPUs, el cual ha ido creciendo con el paso de los años. No obstante, al aumentar el tamaño del modelo también aumenta el tiempo necesario para obtener la solución dado que se debe calcular punto a punto los valores de los campos al aplicar el método. Este problema puede solventarse haciendo uso del poder de paralelización que nos ofrecen las tarjetas gráficas (GPUs), pudiendo hacer simultaneamente, hasta cierto límite, todas las operaciones necesarias en cada coordenada de la dimensión espacial. Proceso que se consigue implementando los algoritmos FDTD en el lenguaje CUDA, con el cual podemos podemos determinar un conjunto de hilos para realizar los cálculos simultaneamente.

2 Metodología

2.1 El método de diferencias finitas en el dominio del tiempo

El método numérico de las diferencias finitas se emplea en la resolución de ecuaciones diferenciales de primer orden, en particular, el método de la diferencia central finita se basa en aproximar de forma sencilla la derivada de una función en un punto como la diferencia de dicha función evaluada en dos puntos equidistantes al mismo. Haciendo uso del desarrollo en serie de Taylor y aproximando a tercer orden obtenemos:

$$f(x+h) = y(x) + hy'(x) + \frac{h^3}{2}y''(x) + \frac{h^3}{6}y'''(x) \tag{2.1}$$

$$f(x-h) = y(x) - hy'(x) + \frac{h^3}{2}y''(x) - \frac{h^3}{6}y'''(x) \tag{2.2}$$

$$y\prime(x) = \frac{y(x+h) - y(x-h)}{2h} + O(h^2) \tag{2.3}$$

Donde hemos despejado el valor de la derivada de la función en un punto mediante su desarrollo en serie en $(x+h)$ y $(x-h)$.

Recordando la expresión de las ecuaciones de Maxwell para el campo electromagnético, haciendo uso del vector campo D.

$$\frac{\partial D}{\partial t} = \nabla \times H \tag{2.4}$$

$$\frac{\partial H}{\partial t} = -\frac{1}{\mu_0}\nabla \times E \tag{2.5}$$

$$D(\omega) = \varepsilon_0 \varepsilon_r^*(\omega) E(\omega) \tag{2.6}$$

Podemos observar que están expresadas en el dominio de la frecuencia para poder aplicarlas a medios cuya constante dieléctrica dependa de la misma. Resaltar que ε_0 y ε_r corresponden con las constantes dieléctricas del vacío y del medio respectivamente, y μ_0 es la permeabilidad magnética del vacío.

Para simplificar el formalismo es conveniente normalizar las ecuaciones haciendo uso de las constantes dieléctrica y permeabilidad magnética del vacío.

$$\tilde{E} = \sqrt{\frac{\varepsilon_0}{\mu_0}} E \tag{2.7}$$

$$\tilde{D} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}} D \tag{2.8}$$

De este modo podemos expresar las ecuaciones de Maxwell para el campo electromagnético como:

$$\frac{\partial \tilde{D}}{\partial t} = -\frac{1}{\sqrt{\varepsilon_0 \mu_0}}\nabla \times H \tag{2.9}$$

$$\frac{\partial H}{\partial t} = -\frac{1}{\sqrt{\varepsilon_0 \mu_0}}\nabla \times E \tag{2.10}$$

$$\tilde{D}(\omega) = \varepsilon_r^*(\omega) \tilde{E}(\omega) \tag{2.11}$$

En $\mathbb{R}^3$ tenemos 3 componentes para el campo eléctrico y 3 para el magnético $\tilde{E}_x$, $\tilde{E}_y$, $\tilde{E}_z$, H_x, H_y, H_z, por lo que, si queremos aplicar un modelo en dos dimensiones debemos elegir uno de los dos grupos de tres vectores, (i) el transversal magnético, formado por $\tilde{E}_z$, H_x, H_y, o (ii) el transversal eléctrico, compuesto por los vectores $\tilde{E}_x$, $\tilde{E}_y$ y H_z.

Usaremos el modo transversal magnético y utilizaremos directamente las letras E y D sin tildar para nombrar a las componentes normalizadas de los campos:

$$\frac{\partial D_z}{\partial t} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) \tag{2.12}$$

$$D(\omega) = \varepsilon_r^*(\omega) E(\omega) \tag{2.13}$$

$$\frac{\partial H_x}{\partial t} = -\frac{1}{\sqrt{\varepsilon_0 \mu_0}}\frac{\partial E_z}{\partial y} \tag{2.14}$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\sqrt{\varepsilon_0 \mu_0}}\frac{\partial E_z}{\partial x} \tag{2.15}$$

Estas son las ecuaciones de una onda plana moviendose en la dirección x e y con el campo eléctrico orientado en la dirección z y el magnético en las direcciones x e

y. Las componentes a calcular son ecuaciones diferenciales de primer orden, por lo que, aplicando el método de la diferencia central finita componente a componente obtenemos.

$$\begin{aligned}\frac{D_z^{n+1/2}(i,j) - D_z^{n-1/2}(i,j)}{\Delta t} = &\frac{1}{\sqrt{\varepsilon_0\mu_0}}\left[\frac{H_y^n(i+\frac{1}{2},j) - H_y^n(i-\frac{1}{2},j)}{\Delta x}\right] \\ &-\frac{1}{\sqrt{\varepsilon_0\mu_0}}\left[\frac{H_x^n(i,j+\frac{1}{2}) - H_x^n(i,j-\frac{1}{2})}{\Delta x}\right]\end{aligned} \tag{2.16}$$

$$\frac{H_x^{n+1}(i,j+\frac{1}{2}) - H_x^n(i,j+\frac{1}{2})}{\Delta t} = -\frac{1}{\sqrt{\varepsilon_0\mu_0}}\left[\frac{E_z^{n+1/2}(i,j+1) - E_z^{n+1/2}(i,j)}{\Delta x}\right] \tag{2.17}$$

$$\frac{H_y^{n+1}(i+\frac{1}{2},j) - H_y^n(i+\frac{1}{2},j)}{\Delta t} = \frac{1}{\sqrt{\varepsilon_0\mu_0}}\left[\frac{E_z^{n+1/2}(i+1,j) - E_z^{n+1/2}(i,j)}{\Delta x}\right] \tag{2.18}$$

En estas ecuaciones el tiempo se especifica con el superíndice, donde n es el paso temporal, de modo que $t = \Delta t \cdot n$. En los paréntesis utilizamos los valores de i, j para determinar la posición, sin olvidar que el campo eléctrico y el magnético están intercalados en tiempo y espacio, lo cual se especifica con la suma y resta de valores $\frac{1}{2}$ en los superíndices y paréntesis. En la figura 1 se muestra el intercalado de los campos en el espacio.

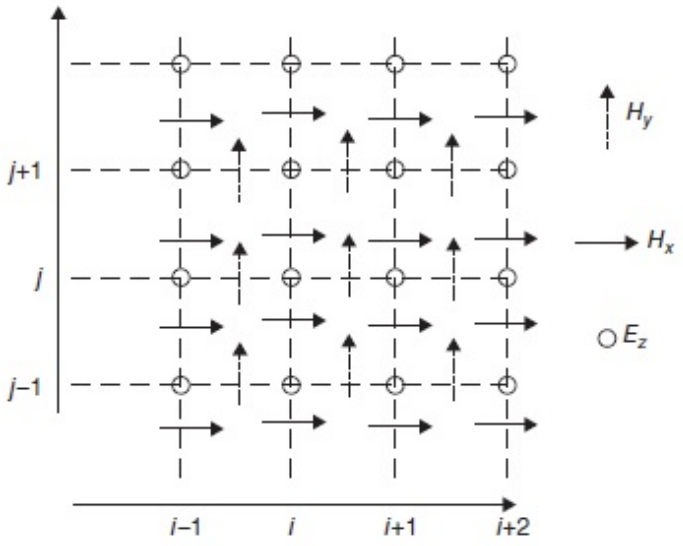

Figura 1: Campos Ez, Hx y Hy intercalados [1].

Utilizaremos $\Delta x = \Delta y$, por lo que al haber elegido el tamaño de celda ($\Delta x \cdot \Delta y$), podemos calcular el salto temporal como:

$$\Delta t = \frac{\Delta x}{2 \cdot c_0} \tag{2.19}$$

Donde c_0 es la velocidad de la luz en el vacío. Recordando que $\varepsilon_0\mu_0 = 1/(c_0)^2$ y teniendo en cuenta la normalización utilazada:

$$\frac{\Delta t}{\sqrt{\varepsilon_0\mu_0}\Delta x} = \frac{\Delta x}{2 \cdot c_0} \cdot \frac{1}{\sqrt{\varepsilon_0\mu_0}\Delta x} = \frac{1}{2} \tag{2.20}$$

Lo que nos permite, junto a las expresiones discretizadas en tiempo y espacio de las ecuaciones de Maxwell y la normalización empleada, generar un algoritmo de forma sencilla para calcular los campos:

$$\begin{aligned} dz[i,j] &= dz[i,j] + 0.5 * (hy[i,j] - hy[i-1,j] - hx[i,j] + hx[i,j-1]) \\ ez[i,j] &= gaz[i,j] * dz[i,j] \\ hx[i,j] &= hx[i,j] + 0.5 * (ez[i,j] - ez[i,j+1]) \\ hy[i,j] &= hy[i,j] + 0.5 * (ez[i+1,j] - ez[i,j]) \end{aligned} \tag{2.21}$$

El valor $gaz[i,j]$ depende de las características del medio, por lo que si suponemos vacío su valor es 1. De este modo las ecuaciones (2.21) corresponden a los vectores necesarios para determinar los campos en $\mathbb{R}^2$, donde solo es necesario conocer el valor de la fuente que genera la radiación, ya sea de cualquier tipo, sinusoidal o un pulso entre otros.

En general, para conocer el valor del campo en todo el espacio es necesario calcular su componente punto a punto pero utilizando la paralelización de la GPU podemos calcularlos todos los valores de los vectores a la vez.

2.2 GPU

A lo largo de los años, debido a la necesidad de procesar gráficos de alta definición al instante, las tarjetas gráficas se han convertido en sistemas multinúcleo con la capacidad de multiproceso paralelizado. Las unidades de procesamiento de gráficos (GPUs) proporcionan más rendimiento y ancho de banda de memoria que las CPUs al mismo precio. Muchas aplicaciones aprovechan esta ventaja para ejercutarse más rapido en GPU que en CPU. La diferencia de potencia de las GPUs y las CPUs se basa en el propósito para el que fueron diseñadas las mismas. Las CPUs se crearon para que, con los pocos núcleos con que se construyen, ejecuten cálculos de forma secuencial en el menor tiempo posible. No obstante, las GPUs se crearon con miles de núcleos para que trabajen todos a la vez, realizando todos los cálculos paralelamente en vez de usar una metodología secuencial.

En la figura 2 se muestra la diferencia de estructura entre CPU y GPU, variando la cantidad de núcleos de la misma según el modelo de dispositivos utilizados. En general, dada la necesidad de ejecutar procesos tanto secuenciales como paralelos los sistemas están diseñados para poder mezclar los atributos de las CPUs y las GPUs maximizando así el rendimiento de los algoritmos [3].

Nvidia ha desarrollado CUDA para generar soluciones que utilicen la paralerización de las tarjetas gráficas, un lenguaje de programación basado en C, el cual comparte sintaxis en muchos aspectos con este último. En CUDA distinguimos dos tipos de procesos, los llevados a cabo en la CPU (host) y los ejecutados desde la GPU (Device o Kernel),

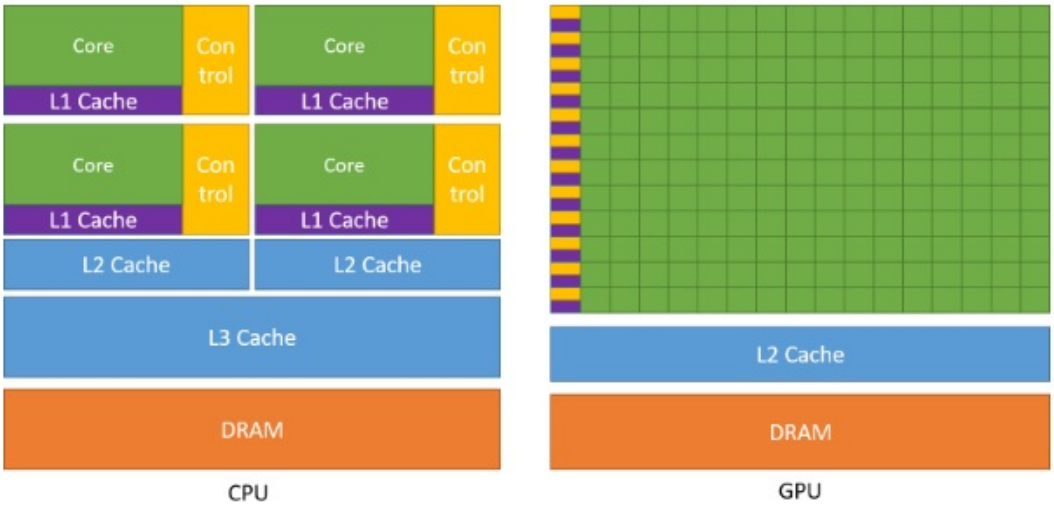

Figura 2: Diferencia de transistores entre CPU y GPU [3].

teniendo acceso ambos dispositivos a dos tipos de memoria, llamadas global y constante. En la figura 3 se muestra la estructura del device y cómo ambos comparten dichas memorias. Para identificar el lugar de ejecución de cada función en CUDA se especifica en el encabezado de la misma las palabras __host__ en las aplicaciones ejecutadas en CPU y __global__ en las ejecutadas en GPU.

Cada tarjeta gráfica tiene un número determinado de threads o hilos, los cuales pueden realizar un cálculo concreto simultaneamente cuando son llamados por una función global (desde el device). Para realizar esta llamada CUDA divide los núcleos en bloques y mallas como se puede ver en la figura 3, donde se muestra una malla de dos bloques y dos hilos en cada bloque, siendo 4 los hilos totales. Destacar que en general el número de hilos tiene que coincidir necesariamente con el número de núcleos ya que, como ocurre en las CPUs, puede haber varios hilos en un mismo núcleo. Esto significa que dentro de un mísmo núcleo pueden realizarse tantos cálculos simultáneos como hilos tenga dicho procesador, no obstante la diferencia de hilos entre un procesador y una gráfica es de varios órdenes de magnitud, pudiendo ser 6 en una CPU y miles en una GPU.

Al implementar una solución, desde el host se selecciona el número de hilos a utilizar teniendo en cuenta las limitaciones de hilos por bloques y bloques por mallas de la tarjeta gráfica que se esté utilizando y desde la función global (en el device) se determinan las operaciones a realizar por cada hilo. Es decir, antes de ejecutar la función __globla__ se debe conocer el número de hilos y bloques a utilizar por la GPU, pudiendo lanzar un número determinado de estos en las tres dimensiones del espacio (x,y,z).

Dadas las variables a utilizar, hilos, bloques y mallas, es necesario identificar el hilo que se quiere utilizar para ejecutar el cálculo. Conceptualmente podemos ver una malla como una matriz, dividida por bloques e hilos dentro de estos bloques como se muestra en la figura 4. En ella podemos ver una malla compuesta por 60 hilos, dividida por 4 bloques, dos en la dimensión x denotada por 'gridDim.x', dos en la dimensión y denotada por 'gridDim.y', estando cada bloque compuesto de 15 hilos.

No obstante, en CUDA esta división es lineal como se muestra en la figura 5. En ella

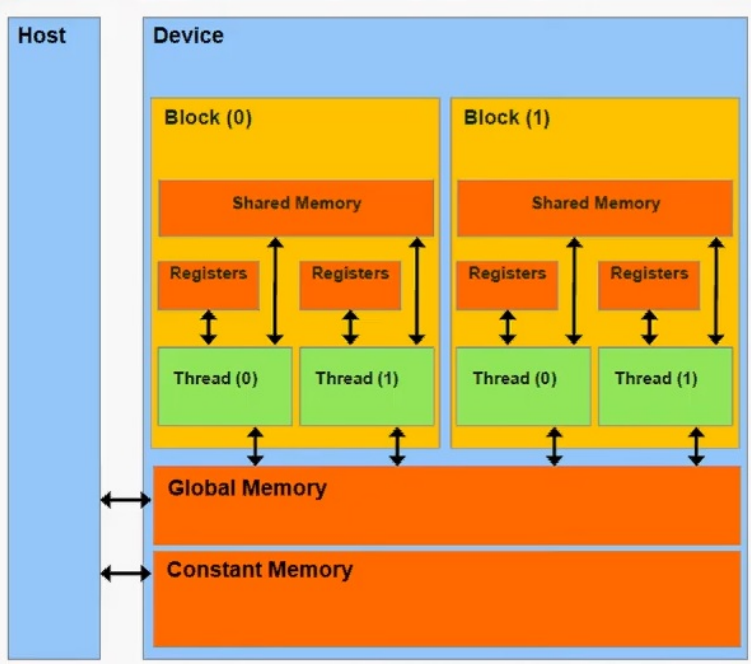

Figura 3: Estructura del Host ([3]).

podemos ver separados por franjas claras y oscuras el paso de un bloque a otro, estando estos divididos por el número de hilos que hay dentro de cada bloque. De modo que, para poder identificar un hilo podemos concebir la malla como una matriz donde las filas determinan el bloque y las columnas los hilos. Así, se distingue unívocamente cada hilo multiplicando las columnas totales por el número del bloque donde está el hilo y sumandole la posición que ocupa en el.

En nuesto caso, debido a las necesidades del problema necesitamos identificar los hilos de una malla lanzada con hilos y bloques en dos dimensiones. Por lo que necesitamos identificar qué son las filas y las columnas en nuestra función __globla__. Esta identificación la conseguimos usando las palabras reservadas que nos proporciona CUDA:

```
int fil = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

Es necesario resaltar la necesidad de seleccionar la cantidad de hilos y bloques acorde a la capacidad de la gráfica y las necesidades del problema. En específico, para la resolución de las ecuaciones de Maxwell en dos dimensiones para un mallado de un millón de puntos, dividido en 1000 posiciones en la dimensión x y 1000 posiciones en la dimensión y, con la gráfica utilizada en este TFG (la cual perrmite 1024 hilos por bloque), es necesario utilizar un mínimo de 977 bloques de 1024 hilos cada uno.

Finalmente, para medir el tiempo de cálculo en el device CUDA nos proporciona la función 'cudaEvent', con la cual se establecen dos marcas de tiempo en el algoritmo y se calcula el tiempo como la resta de estas mismas, siendo la diferencia el tiempo transcurrido en el device al realizar las operaciones. Para calcular el tiempo en el host se ha utilizado la función 'clock_t' y de la misma forma se han creado dos marcas temporales y se ha calculado la diferencia. En ambos casos se han dividido los resultados entre las

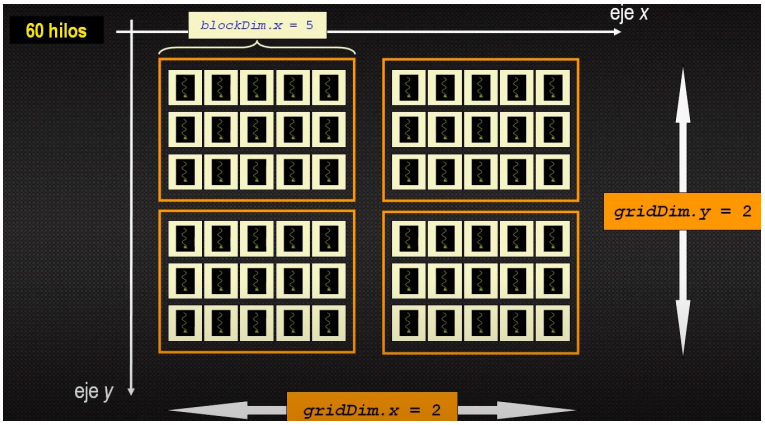

Figura 4: Topología dentro de hilos, bloques y mallas [3]

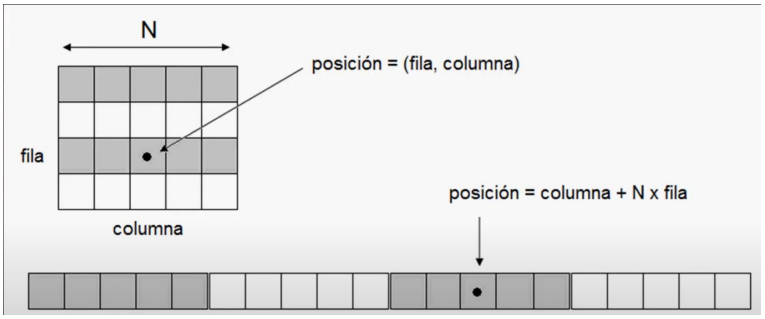

Figura 5: Identificación de hilo dentro de una malla.

cuentas por segundo dentro del procesador y la tarjeta gráfica para obtener el tiempo en segundos con una precisión de ± 0.001 s.

3 Algoritmo

En este TFG el autor ha desarrollado varios códigos en CUDA de libre uso para la resolución de las ecuaciones de Maxwell aplicando el método FDTD en dos dimensiones. Los códigos generados pueden encontrarse en el siguiente repositorio de github:

https://github.com/juanjo1213/TFG-FDTD

En el mismo están los algoritmos utilizados para la solución en 2D implementada tanto en la cpu y en la gpu. La implementación de la solución en el device está nombrada como 'fd2d_simp.cu' y la ejecutada en el host como 'HOST_FD2D_SIMP.cu'.

Para la ejecución de los algoritmos se ha utilizado un procesador i7-6700K procesador de 4 núcleos y 8 hilos con una frecuencia máxima de 4.2 GHz por núcleo y una tarjeta gráfica GTX 1070 con 8GB de memoria RAM y un total de 1920 núcleos CUDA.

A continuación se muestra la función __host__ generada para el cálculo de las ecuaciones 2.21, esta es llamada desde el host y recibe como parámetro un 'struct' que contiene los vectores campo a calcular. En ella se ejecutan varios bucles para el cálculo secuencial de los campos punto a punto y para mayor similitud, se han almacenado las matrices como vectores de dimensión $dimx \cdot dimy$ y se ha determinado cada componente a calcular como se expuso en la sección 2.2.

```
void FD2D(FDTDData* field, int xdim, int ydim, int nsteps)
{
 int ic = int(xdim / 2),
  jc = int(ydim / 2);

 //Medium conditions
 double* gaz;

 gaz = (double*)malloc(xdim * ydim * sizeof(double));

 for (int j = 0;j < ydim;j++) {
  for (int i = 0;i < xdim;i++) {
   gaz[j * xdim + i] = 1.0;
  }
 }


 double ddx = 0.01,         //cell size
     dt = ddx / 6e8,     //Time step size
     epsz = 8.854e-12;   //Dielectric profile



 //Pulse parameters
 double t0 = 20.0,
  spread = 6.0;

 //Main loop
 for (int time_step = 1;time_step < nsteps + 1;time_step++)
 {
  //Calculate Dz
  for (int j = 1;j < ydim;j++) {
   for (int i = 1;i < xdim;i++) {
    field[j*xdim+i].dz= field[j * xdim + i].dz+
                        0.5*(field[j*xdim+i].hy- field[j * xdim + i-1].hy-
                        field[j*xdim+i].hx+ field[(j-1) * xdim +i].hx);
   }
  }

  //Gaussian pulse in the middle
  double pulse = exp(-0.5 *
           ((t0 - time_step) / spread) * ((t0 - time_step) / spread));
  field[jc * xdim + ic].dz = pulse;

  //Calculate the Ez field from Dz
  for (int j = 1;j < ydim;j++) {
   for (int i = 1;i < xdim;i++) {
```

```
    field[j * xdim + i].ez = gaz[j * xdim + i] * field[j * xdim + i].dz;
   }
  }

  //Calculate the Hx field
  for (int j = 0;j < ydim-1;j++) {
   for (int i = 0;i < xdim-1;i++) {
    field[j * xdim + i].hx = field[j * xdim + i].hx +
            0.5 * (field[j * xdim + i].ez - field[(j + 1) * xdim + i].ez);
   }
  }

  //Calculate the Hy field
   //Calculate the Hx field
  for (int j = 0;j < ydim - 1;j++) {
   for (int i = 0;i < xdim - 1;i++) {
    field[j * xdim + i].hy = field[j * xdim + i].hy +
          0.5 * (field[j * xdim + i + 1].ez - field[j * xdim + i].ez);
   }
  }

 }




}
```

En el siguiente algoritmo se muestra la función __global__ generada para realizar los cálculos de forma paralela. En ella podemos ver como se determinan las filas y las columnas utilizadas para determinar los hilos a utilizar como se explicó en la sección 2.2. La función, al igual que la función anteior, recibe como parámetro un struct que contiene los vectores a clacular, las dimensiones x e y del problema y el paso temporal que se está ejecutando. Al existir la posibilidad de reservar más bloques e hilos de los necesarios para realizar los cálculos, se usa el condicional 'if' para especificar el número de hilos a utilizar.

```
    __global__ void FD2D(FDTDData* field, int xdim, int ydim, int ts)
{
 int fil = blockIdx.y * blockDim.y + threadIdx.y;
 int col = blockIdx.x * blockDim.x + threadIdx.x;


 int ic = int(xdim / 2),
  jc = int(ydim / 2);

 double ddx = 0.01,
  dt = ddx / 6e8,
```

```
 epsz = 8.854e-12;

//pulse parameters
double t0 = 20.0,
 spread = 6.0;

//Calculate Dz
if (0 < fil && fil < ydim && 0 < col && col < xdim) {
 field[fil * xdim + col].dz += 0.5 *
     (field[fil * xdim + col].hy - field[fil * xdim + col - 1].hy -
  field[fil * xdim + col].hx + field[(fil - 1) * xdim + col].hx);
}
__syncthreads();

//Gaussian pulse in the middle
double pulse = exp(-0.5 * ((t0 - ts) / spread) * ((t0 - ts) / spread));
field[jc * xdim + ic].dz = pulse;
__syncthreads();

//Calculate the Ez field from Dz
if (0 < fil && fil < ydim && 0 < col && col < xdim) {
 field[fil * xdim + col].ez = field[fil * xdim + col].gaz *
                              field[fil * xdim + col].dz;
}
__syncthreads();

//Calculate the Hx field
if (fil < ydim - 1 && col < xdim - 1) {
 field[fil * xdim + col].hx += 0.5 * (field[fil * xdim + col].ez -
                                      field[(fil + 1) * xdim + col].ez);
}
__syncthreads();

//Calculate the Hy field
if (fil < ydim - 1 && col < xdim - 1) {
 field[fil * xdim + col].hy += 0.5 *
       (field[fil * xdim + col + 1].ez - field[fil * xdim + col].ez);
}

}
```

En ambas funciones se emplea un bucle ya sea dentro o fuera de la función para especificar el número de pasos temporales a realizar, además de utilizar como fuente de los campos un pulso gaussiano, denotado en ambas funciones como 'pulse' y generado en el centro del espacio.

4 Resultados

Al resolver las ecuaciones 2.21 obtenemos una onda electromagnética propagandose por el espacio, en nuestro caso, siendo generada por un pulso gaussiano como fuente. La figura 6 muestra la propagación del campo eléctrico (Ez) en un espacio de 3600 puntos y 20, 30, 40, 50 pasos temporales. Como podemos ver, la onda generada en el centro del espacio se propaga en las direcciones x e y.

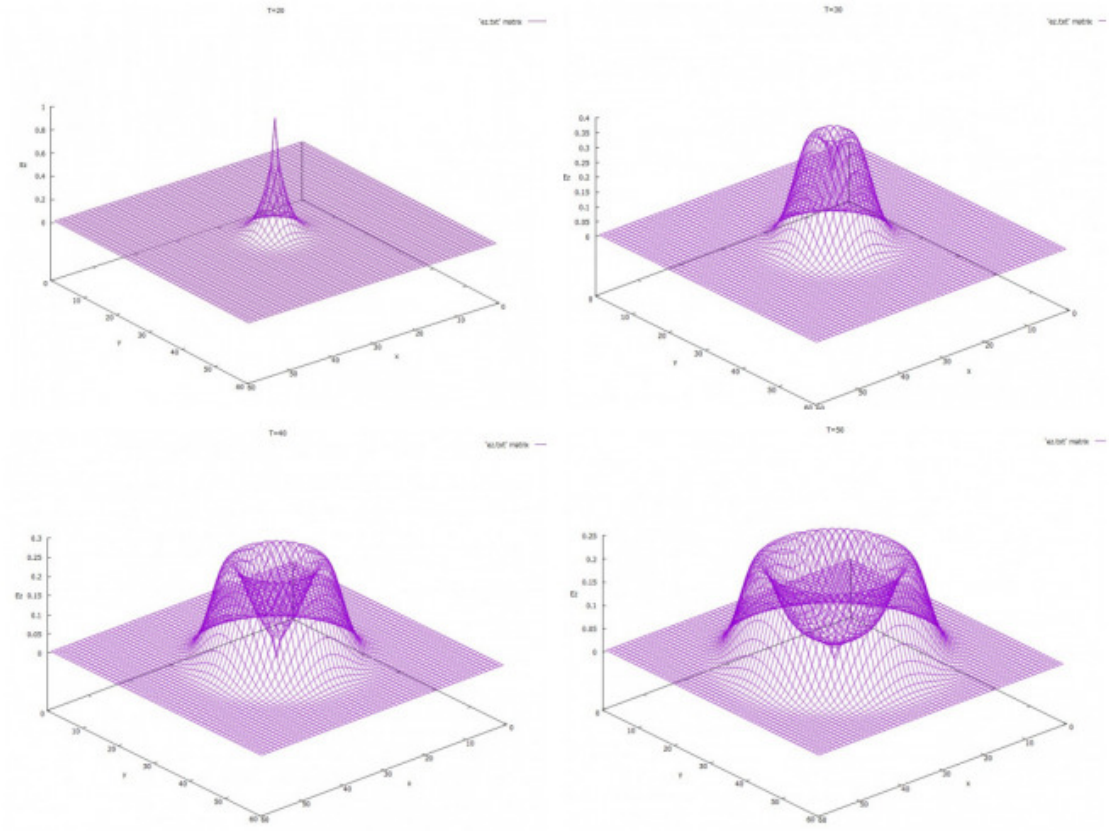

Figura 6: Propagación del campo eléctrico en $\mathbb{R}^2$ en un espacio de 3600 puntos en 20 (Superior-Izquierda), 30 (Superior-Derecha), 40 (Inferior-izquierda) y 50 (Inferior-Derecha) pasos temporales.

Para determinar el aumento de velocidad en tiempo de cálculo al implementar este algoritmo en CUDA se han realizado dos experimenteos donde se determina su dependencia con el número de pasos temporales y la densidad de puntos. En ambos se han realizado una serie de ejecuciones para cada tiempo a determinar, de las cuales se han calculado el porcentaje de dispersión, la media y se ha elejido el error del resultado como el máximo entre la sensibilidad de la medida y la desviación de la media.

4.1 Dependencia del tiempo de cálculo con el número de pasos temporales

Para determinar la dependencia del tiempo de cálculo con el número de pasos temporales se han llevado a cabo varias ejecuciones en host y device. En estas se ha utilizado una densidad constante de puntos de 25 millones y se han variado los pasos temporales

desde 10 hasta 10 000 realizando un mínimo de tres medidas para cada valor de pasos temporales en cada algoritmo. En la tabla 1 se enumeran los datos obtenidos en las ejecuciones realizas, en ella se ha nombrado al tiempo de ejecución en CPU como t_{CPU} y el tiempo de ejecución en el device como t_{GPU}. En la última columna se ha calculado el llamado aumento de velocidad (speed up) para determinar la mejora en tiempo de ejecución de una tecnología a otra.

Como podemos ver en la tabla, haciendo la media de los valores calculados, obtenemos una mejora en la velocidad de ejecución (speed up) de aproximádamente 12.31x. Esto nos indica que el álgoritmo CUDA nos ofrece una solución en 12.31 menos tiempo que la solución en CPU.

Pasos Temporales	t_{CPU} (s)	t_{GPU} (s)	$(t_{\text{GPU}}/t_{\text{CPU}})$
10	1.971 ± 0.008	0.165 ± 0.001	11.91 ± 0.09
100	19.32 ± 0.05	1.633 ± 0.003	11.83 ± 0.04
1000	209.1 ± 1.4	16.397 ± 0.023	12.75 ± 0.09
10000	2150 ± 13	168.6 ± 0.9	12.75 ± 0.10

Tabla 1: Datos obtenidos tras medir la dependencia del tiempo de cálculo con el número de pasos temporales en CPU y GPU además de calcular el aumento en velocidad al usar CUDA.

Al escoger un aumento lineal del número de pasos temporales cabe esperar que el aumento del tiempo de cálculo también sea lineal, ya que en ambos se ha empleado un bucle para realizar las funciones un número determinado de veces. No obstante, tambien es de esperar que en el device el tiempo de cálculo sea mucho menor debido a la paralelización aplicada, reduciendo el tiempo que se tarda en pasar de una sección del bucle a otra. Podemos comprobar estos resultados en la figura 7, donde vemos un gráfico de los resultados obtenidos en la tabla 1, representando en color azul los correspondientes al algoritmo ejecutado en el host y en color naranja los del device. En el gráfico se han representado las regresiónes lineales obtenidas para las dos series de datos, siguiendo ambas una tendencia completamente lineal ya que en las dos se ha obtenido un valor del coeficiente R^2 de 0.99.

Es de interes destacar la diferencia en las pendientes de ambas regresiones, siendo en la del host de 0.2152 ± 0.0003 s y la del device de $0.01687 \pm 3{\cdot}10^{-5}$ s, lo que nos indica, al ser la pendiente del host un orden de magnitud mayor que la del device, un aumento mucho menor en el tiempo de cálculo al aumentar el número de pasos temporales utilizando CUDA que aplicando el algoritmo en la CPU.

4.2 Variación del tiempo de cálculo al variar la densidad de puntos

Para determinar la dependencia del tiempo de cálculo de los algoritmos con la densidad de puntos utilizada también se han realizado varias ejecuciones tanto en host como en device. En todas ellas se ha mantenido constante el número de pasos temporales, 1000 y se ha variado la densidad de puntos desde 10^6 hasta $25 \cdot 10^6$. No se han escogido valores equiespaciados de la densidad de puntos para la recolección de datos por la condición mencionada en la sección 2.1, donde escogimos $\Delta x = \Delta y$ para así facilitar el desarrollo de las expresiones. En la tabla 2 se recogen los datos obtenidos divididos en columnas, al

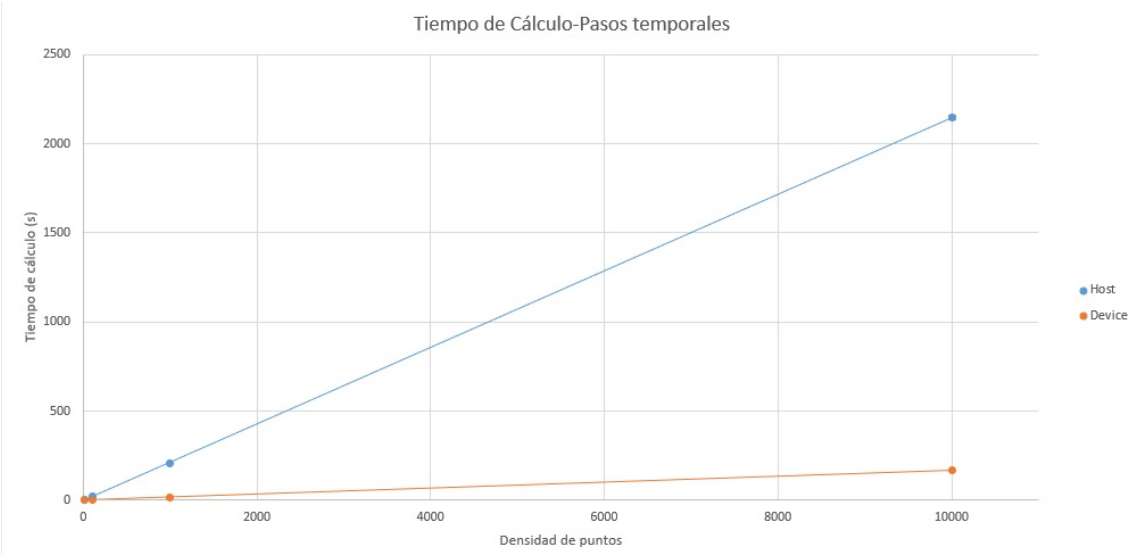

Figura 7: Relación entre el tiempo de cálculo y el número de pasos temporales para una densidad de puntos de 25 millones. La línea naranja representa el tiempo de ejecución en el device y la azul el tiempo de ejecución en el host. En ambas series de datos se ha representado la recta obtenida tras hacer una regresión lineal. Los valores de la pendiente obtenidos son de 0.2152 ± 0.0003 s para el host y de $0.01687 \pm 3{\cdot}10^{-5}$ s para el device.

igual que en el apartado anterior, se han utilizado las letras t_{CPU} y t_{GPU} para nombrar a los tiempos de cálculo de host y device respectivamente.

En la última columna también se recoge el aumento en velocidad de cálculo (speed up), siendo en este caso la media de los valores de aproximádamente 13x, lo que nos indica que obtenemos una solución en CUDA 13 veces más rápido que en CPU.

Densidad de puntos	t_{CPU} (s)	t_{GPU} (s)	$(t_{\text{GPU}}/t_{\text{CPU}})$
10^6	7.67 ± 0.04	0.640 ± 0.002	11.98 ± 0.07
$4 \cdot 10^6$	35.36 ± 0.23	2.447 ± 0.016	14.44 ± 0.13
$9 \cdot 10^6$	72.3 ± 0.3	5.582 ± 0.021	12.96 ± 0.07
$16 \cdot 10^6$	126.3 ± 0.5	9.81 ± 0.03	12.88 ± 0.06
$25 \cdot 10^6$	209.1 ± 1.4	16.397 ± 0.23	12.75 ± 0.09

Tabla 2: Datos obtenidos tras medir la dependencia del tiempo de cálculo con la densidad de puntos en CPU y GPU además de calcular el aumento de velocidad al usar CUDA.

Al igual que el apartdo anterior, al aumentar la densidad de punto haciendo el problema un número x de veces más grande cabe esperar algún tipo de tendencia en el tiempo de cálculo para ambos algoritmos, siendo esta tendencia menos pronunciada en el algoritmo CUDA. Para determinar este comportamiento, en la figura 8 se han representado los datos obtenidos en la tabla 2. En ella se ha realizado una regresión lineal para las dos series de valores (CPU y GPU), deduciendo que los valores se ajustan a una recta al obtener un valor de R^2 de 0.99 para ambas.

Como es de esperar se obtiene un valor de pendiente mayor para el algoritmo implementado en la CPU, $8.2 \pm 0.2 \cdot 10^{-6}$, que el calculado en el algoritmo implementado en GPU, $6.52 \pm 0.16 \cdot 10^{-7}$. No obstante, esta diferencia no es tan acusada como la deter-

minada en la sección 4.1, donde la calculada para GPU era de un orden de diferencia menor.

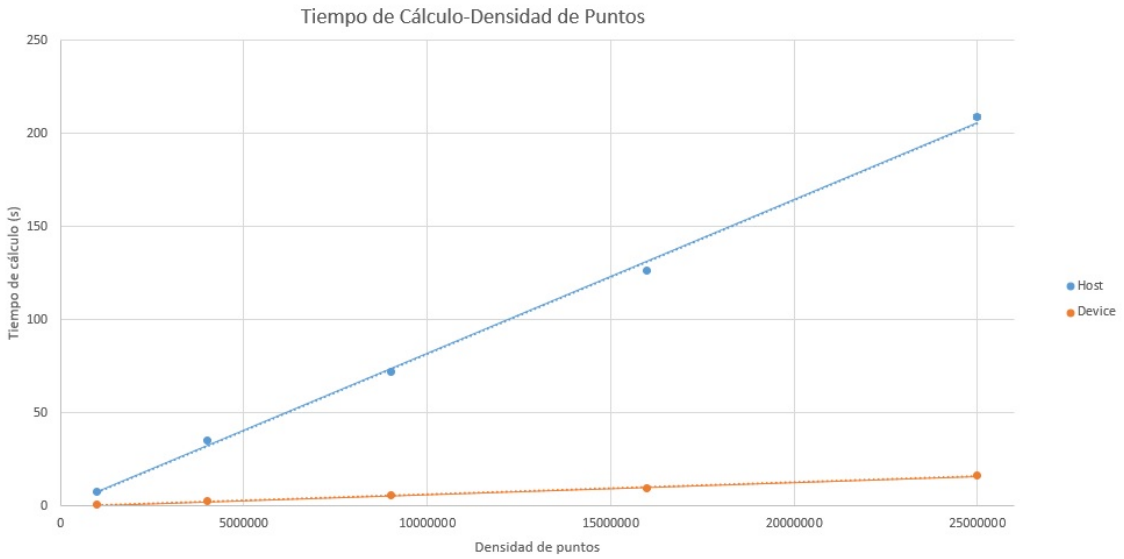

Figura 8: Relación entre el tiempo de cálculo y la densidad de puntos para 1000 pasos temporales. La línea naranja representa el tiempo de ejecución en el device y la azul el tiempo de ejecución en el host. En ambas series de datos se ha representado la recta obtenida tras hacer una regresión lineal. Los valores de las pendientes obtenidos son 8.2 $\pm 0.2 \cdot 10^{-6}$ para la CPU y 6.52 $\pm 0.16 \cdot 10^{-7}$ para la GPU.

5 Conclusiones

A Datos y errores

A.1 Errores

Para la propagación de errores se ha utilizado la expresión:

$$\Delta f(a_i) = \sqrt{\sum_{i=1}^{n} (\frac{\delta f(a_i)}{\delta a_i} \cdot \Delta a_i)^2} \tag{A.1}$$

A.2 Datos

CPU			
Pasos Temporales	t_{CPU} (± 0.001 s)	t_{CPU} (± 0.001 s)	t_{CPU} (± 0.001 s)
10	1.964	1.965	1.983
100	19.270	19.300	19.400
1000	209.483	210.762	206.966
10000	2169.530	2139.410	2139.820

Tabla 3: Ejecución en CPU con una densidad de puntos de 25 millones

GPU			
Pasos Temporales	t_{GPU} (± 0.001 s)	t_{GPU} (± 0.001 s)	t_{GPU} (± 0.001 s)
10	0.165	0.167	0.164
100	1.638	1.632	1.630
1000	16.388	16.431	16.371
10000	168.697	167.173	169.841

Tabla 4: Ejecución en GPU con una densidad de puntos de 25 millones

CPU			
Densidad de puntos	t_{GPU} (± 0.001 s)	t_{GPU} (± 0.001 s)	t_{GPU} (± 0.001 s)
10^6	7.729	7.612	7.675
$4 \cdot 10^6$	35.557	35.019	35.500
$9 \cdot 10^6$	71.978	72.406	72.770
$16 \cdot 10^6$	126.000	126.994	125.917
$25 \cdot 10^6$	209.483	210.762	206.996

Tabla 5: Ejecución en CPU para un número de pasos de 1000

GPU			
Densidad de puntos	t_{GPU} (± 0.001 s)	t_{GPU} (± 0.001 s)	t_{GPU} (± 0.001 s)
10^6	0.638	0.643	0.639
$4 \cdot 10^6$	2.438	2.471	2.432
$9 \cdot 10^6$	5.573	5.560	5.614
$16 \cdot 10^6$	9.853	9.779	9.785
$25 \cdot 10^6$	16.388	16.431	16.371

Tabla 6: Ejecución en GPU para un número de pasos de 1000

Referencias

[1] Jennifer E Houle and Dennis M Sullivan,
Electromagnetic simulation using the FDTD method with Python,
IEEE PRESS WILEY, 2020.

[2] Allen Taflove and Susan C Hagness,
Computational electrodynamics. The Finite-Difference Time-Domain Method,
ARTECH HOUSE, 2005.

[3] Nvidia documentation,
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html