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

El método numérico de las diferencias finitas dependientes del tiempo (FDTD) se usa como herramienta para la resolución de ecuaciones diferenciales de primer orden. Este cobra especial importancia en la electrodinámica computacional dado que nos permite resolver las ecuaciones de Maxwell realizando algoritmos con los que calcular punto a punto los valores del campo electromagnético. Estos algoritmos se implementan generalmente usando la potencia de cálculo que nos proporcionan las CPUs y dada la necesidad de discretizar el espacio, el timepo empleado en el proceso de cálculo aumenta con las dimensiones del problema. En este TFG se han generado varios algoritmos de uso libre en CUDA para exponer las ventajas que se obtienen al implementar este método usando el poder de paralelización que nos proporcionan las GPUs, lo que disminuye el tiempo empleado en el cálculo de los campos.

Abstract

The numerical method of the finite differences time domain (FDTD) is used to solve once order differential equations. This method is specially useful in the computational electrodynamics since we can solve Maxwell equations using algorithms to calculate the electromagnetic field point to point. These algorithms are usually implemented using the calculus power of CPUs and, due to the need to discretyze space, the calculate time increase with the problem size. In this TFG it has been developed some CUDA open source algorithms to show the parallelization power given by the GPUs, which allows us to decrease the time needed to calculate the fields.

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.

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 pueden encontrarse los algoritmos utilizados para la solución en 2D implementada en la cpu y en la gpu, siendo todo este repositorio de libre uso.

2 Metodología

2.1 Método

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 segundo 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.

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

$$D(\omega) = \varepsilon_0 \varepsilon_r^*(\omega) E(\omega) \tag{2.5}$$

$$\frac{\partial H}{\partial t} = -\frac{1}{\mu_0}\nabla \times E \tag{2.6}$$

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 a 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:

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

$$\tilde{D}(\omega) = \varepsilon_r^*(\omega)\tilde{E}(\omega) \tag{2.10}$$

$$\frac{\partial H}{\partial t} = -\frac{1}{\sqrt{\varepsilon_0\mu_0}}\nabla \times E \tag{2.11}$$

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 especificamos 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. Asumiremos $\Delta x = \Delta y$.

Una vez se ha 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:

$$dz[i,j] = dz[i,j] + 0.5 * (hy[i,j] - hy[i-1,j] - hx[i,j] + hx[i,j-1]) \tag{2.21}$$

$$ez[i,j] = gaz[i,j] * dz[i,j] \tag{2.22}$$

$$hx[i,j] = hx[i,j] + 0.5 * (ez[i,j] - ez[i,j+1]) \tag{2.23}$$

$$hy[i,j] = hy[i,j] + 0.5 * (ez[i+1,j] - ez[i,j]) \tag{2.24}$$

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 (2.21-2.24) 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 cualquier tipo de fuente, 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 CPU que en GPU. 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 hilos con que se construyen, ejecuten cálculos de forma secuencial en el menor tiempo posible. No obstante, las GPUs se crearon con miles de hilos para que trabajen todos a la vez, realizando todos los cálculos paralelamente en vez de usar una metodología secuencial.

En la figura (1) se muestra la diferencia de la 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 la CPU y la GPU maximizando así el rendimiento de los algoritmos [3].

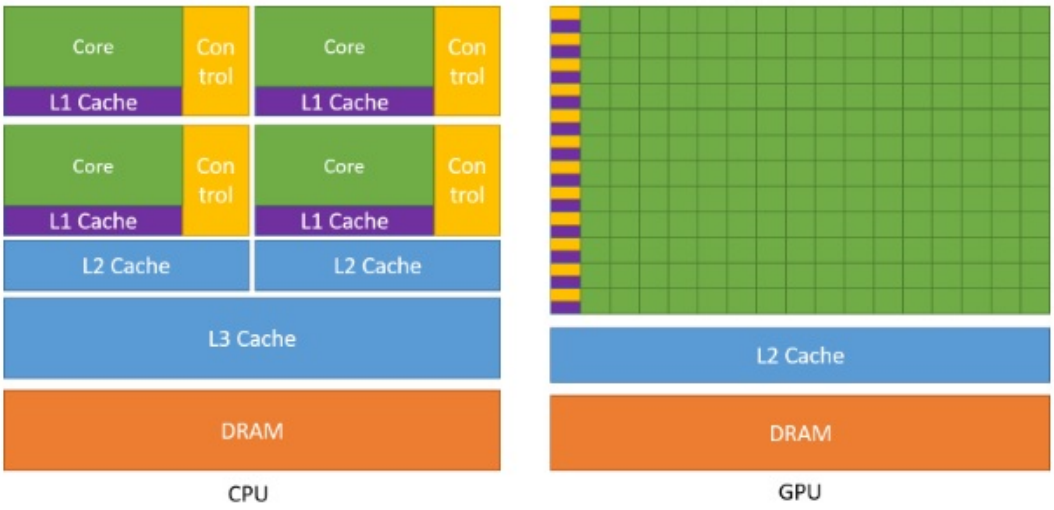

Figura 1: Diferencia de transistores entre CPU y GPU (Documentación de Nvidida [3]).

Para generar soluciones que utilicen la paralerización de las tarjetas gráfics Nvidia ha desarrollado CUDA, un lenguaje de programación basado en C, el cual comparte sintaxis en muchos aspectos con este último lenguaje. En CUDA distinguimos dos tipos de procesos, los llevados a cabo en la CPU o 'host' y los ejecutados desde la GPU o 'device', teniendo acceso ambos dispositivos a dos tipos de memoria, global y constante. En la figura (2) se muestra al estructura del device y cómo ambos comparten memoria. Para identificar el lugar de ejecución de cada función cuda ha nombrado como __host__ a las aplicaciones ejecutadas en CPU y __global__ a las ejecutadas en GPU.

Cada tarjeta gráfica tiene un número determinado de hilos (Threads) los cuales pueden realizar un cálculo concreto, simultaneamente, cuando son llamados por una función global (desde el device). Para realizar esta llamada CUDA ha dividido los hilos en blo-

ques y mallas como se puede ver en la figura (1), donde se muestra una malla de dos bloques y dos hilos e cada bloque, siendo 4 los hilos totales.

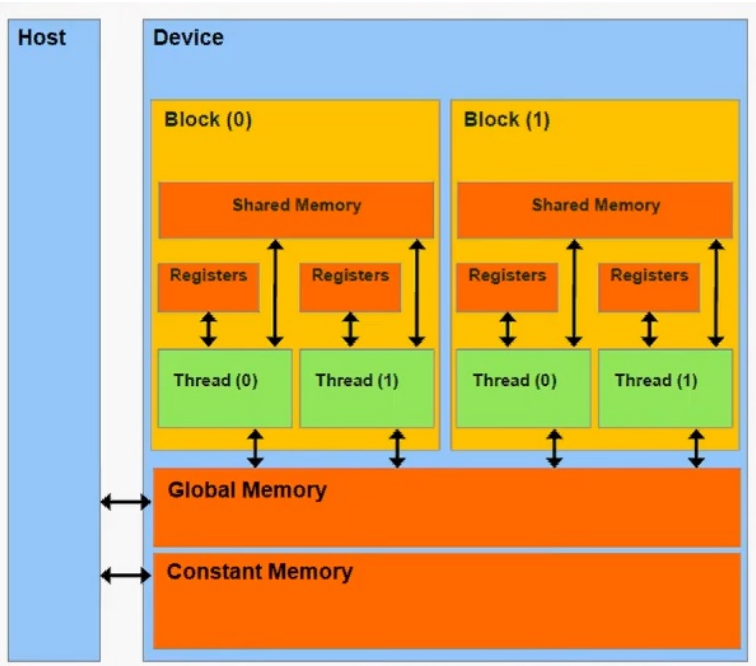

Figura 2: Estructura del Host (Documentación de Nvidia [3]).

En el algoritmo, desde el host se seleccionan 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.

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 (3). En ella se muestra una malla compuesta por 60 hilos, dividida por 4 bloques de 15 hilos cada uno.

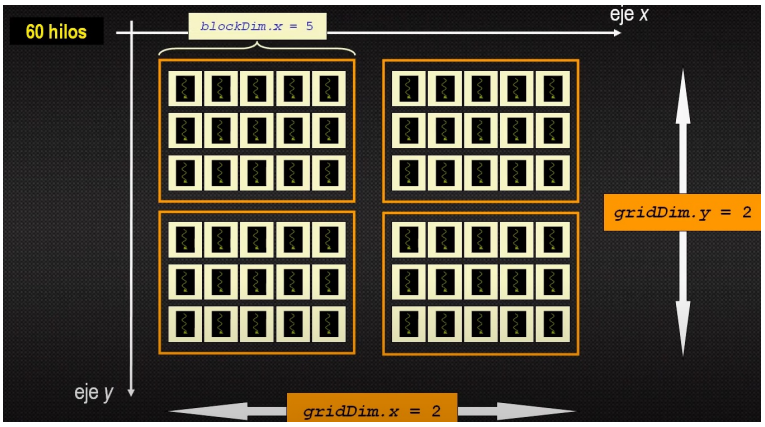

Figura 3: Topología dentro de hilos, bloques y mallas (Documentación de Nvidia [3])

No obstante, la división que hace CUDA de los hilos es lineal como se muestra en

la figura (4). En ella podemos ver separados por franjas claras y oscuras el paso de un bloque a otro. 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.

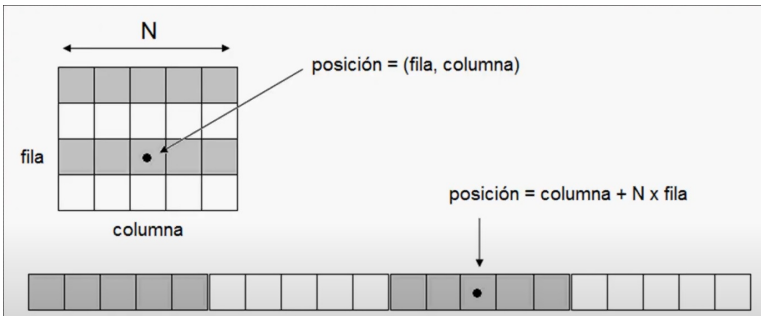

Figura 4: Identificación de hilo dentro de una malla.

Esta identificación se lleva a cabo en los algoritmos utilizados en este TFG, denotando las filas y las columnas como 'fil' y 'col' respectivamente, utilizando las palabras reservadas que proporciona CUDA para identificar cada hilo, bloque y malla.

Es necesario resaltar la necesidad de seleccionar la cantidad de hilos y bloques acorde a la capidad 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.

3 Algoritmo

A continuación se expone la implementación de las ecuaciones (2.21-2.24) usando CUDA, donde se han desarrollado dos algoritmos, uno usando el poder secuencial de la CPU y otro utilizando la capacidad de paralelización de la GPU.

En la figura (5) se muestra la implementación en el host, donde se hace uso de varios bucles para realizar el calculo secuencial posición a posición. Dada la reserva de memoria de los punteros en el host, los cuales son vectores aunque se quiera trabajar con matrices, para ver con mas claridad la similitud, se identifica cada posición como se ha expuso en el apartado (1.2).

```
66      //Main loop
67      for (int time_step = 1;time_step < nsteps + 1;time_step++)
68      {
69          //Calculate Dz
70          for (int j = 1;j < ydim;j++) {
71              for (int i = 1;i < xdim;i++) {
72                  field[j*xdim+i].dz= field[j * xdim + i].dz+0.5*(field[j*xdim+i].hy- field[j * xdim + i-1].hy-
73                                                                 field[j*xdim+i].hx+ field[(j-1) * xdim +i].hx);
74              }
75          }
76
77          //Gaussian pulse in the middle
78          double pulse = exp(-0.5 * ((t0 - time_step) / spread) * ((t0 - time_step) / spread));
79          field[jc * xdim + ic].dz = pulse;
80
81          //Calculate the Ez field from Dz
82          for (int j = 1;j < ydim;j++) {
83              for (int i = 1;i < xdim;i++) {
84                  field[j * xdim + i].ez = gaz[j * xdim + i] * field[j * xdim + i].dz;
85              }
86          }
87
88          //Calculate the Hx field
89          for (int j = 0;j < ydim-1;j++) {
90              for (int i = 0;i < xdim-1;i++) {
91                  field[j * xdim + i].hx = field[j * xdim + i].hx + 0.5 * (field[j * xdim + i].ez - field[(j + 1) * xdim + i].ez);
92              }
93          }
94
95          //Calculate the Hy field
96              //Calculate the Hx field
97          for (int j = 0;j < ydim - 1;j++) {
98              for (int i = 0;i < xdim - 1;i++) {
99                  field[j * xdim + i].hy = field[j * xdim + i].hy + 0.5 * (field[j * xdim + i + 1].ez - field[j * xdim + i].ez);
100             }
101         }
102
103     }
```

Figura 5: Implementación de las ecuaciones (2.21-2.24) para el cálculo de los campos en la CPU

A diferencia del algoritmo anterior, en la figura (6) se muestra la implementación de las ecuaciones para su cálculo de forma paralela en el device, donde se especifica en los vectores que realicen los cálculos hilo a hilo.

```
73      //Calculate Dz
74      if (0 < fil && fil < ydim && 0 < col && col < xdim) {
75          field[fil * xdim + col].dz += 0.5 * (field[fil * xdim + col].hy - field[fil * xdim + col - 1].hy -
76              field[fil * xdim + col].hx + field[(fil - 1) * xdim + col].hx);
77      }
78      __syncthreads();
79
80      //Gaussian pulse in the middle
81      double pulse = exp(-0.5 * ((t0 - ts) / spread) * ((t0 - ts) / spread));
82      field[jc * xdim + ic].dz = pulse;
83      __syncthreads();
84
85      //Calculate the Ez field from Dz
86      if (0 < fil && fil < ydim && 0 < col && col < xdim) {
87          field[fil * xdim + col].ez = field[fil * xdim + col].gaz * field[fil * xdim + col].dz;
88      }
89      __syncthreads();
90
91      //Calculate the Hx field
92      if (fil < ydim - 1 && col < xdim - 1) {
93          field[fil * xdim + col].hx += 0.5 * (field[fil * xdim + col].ez - field[(fil + 1) * xdim + col].ez);
94      }
95      __syncthreads();
96
97      //Calculate the Hy field
98      if (fil < ydim - 1 && col < xdim - 1) {
99          field[fil * xdim + col].hy += 0.5 * (field[fil * xdim + col + 1].ez - field[fil * xdim + col].ez);
100     }
```

Figura 6: Implementación de las ecuaciones (2.21-2.24) para el cálculo de los campos en la GPU

4 Resultados

5 Conclusiones

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