UNIVERSIDAD AUTÓNOMA DE CHIHUAHUA
FACULTAD DE CIENCIAS QUÍMICAS
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

TESIS

IMPLEMENTACIÓN DEL ALGORITMO DE DIFERENCIAS FINITAS EN EL DOMINIO DEL TIEMPO PARA RESOLVER LA EVOLUCIÓN TEMPORAL DE FUNCIONES DE ONDA ELECTRÓNICAS

Que como requisito parcial para la obtención del grado de:
MAESTRA EN CIENCIAS EN QUÍMICA
QUE PRESENTA:

I. Q. Priscilla Ivette Escobedo

Director: Dr. José Manuel Nápoles Duarte
Co-directora: Dra. María Elena Fuentes Montero

Chihuahua, Chih., México.
Junio de 2020

UNIVERSIDAD AUTÓNOMA DE CHIHUAHUA

Chihuahua, Chih., a 26 de junio de 2020.
Oficio: 42/CA/SIP/20

Dr. Ildebrando Pérez Reyes
Secretario de Investigación y Posgrado
Facultad de Ciencias Químicas
Universidad Autónoma de Chihuahua
P r e s e n t e:

Los integrantes del comité, informamos a Usted que efectuamos la revisión de la tesis intitulada: **Implementación del algoritmo de Diferencias Finitas en el Dominio del Tiempo para resolver la evolución temporal de funciones de onda electrónicas** presentada por la **I.Q. Priscilla Ivette Escobedo** alumna del Programa de Maestría en Ciencias en Química.

Después de la revisión, indicamos a la tesista las correcciones que eran necesarias efectuar y habiéndolas realizado, manifestamos que la tesis, de la cual adjuntamos un ejemplar, ha cumplido con los objetivos señalados por el Comité de Tesis, por lo que puede ser considerada como adecuada para que se proceda con los trámites para la presentación de su Examen de Grado.

A t e n t a m e n t e
"Por la Ciencia para Bien del Hombre"

Dra. María Elena Fuentes Montero
Co-Directora de tesis

Dr. Juan Pedro Palomares Báez
Asesor de tesis

Dr. Marco Antonio Chávez Rojo
Asesor de tesis

Dr. José Manuel Nápoles Duarte
Director de tesis

FACULTAD DE CIENCIAS QUÍMICAS
SECRETARIA
INVESTIGACIÓN Y POSGRADO

Dr. Ildebrando Pérez Reyes
Secretario de Investigación y Posgrado

**El que suscribe certifica que las firmas que aparecen en esta acta, son auténticas, y las mismas que utilizan los C. Profesores mencionados.

FACULTAD DE CIENCIAS QUÍMICAS
Circuito Universitario
Campus Universitario #2 C.P. 31125
Tel. +52 (614) 236 6000
Chihuahua, Chihuahua, México
http://www.fcq.uach.mx

Agradecimientos

Debo agradecer primeramente a mi asesor, el doctor Nápoles, quien siempre ha sido paciente y comprensivo conmigo, buscando mi aprendizaje y superación: gracias por su confianza y apoyo en esta etapa. Agradezco también a la doctora Fuentes por su apoyo emocional y académico: gracias por sus consejos, incluyendo el “sólo se vive una vez”. A mi familia, quienes a su manera han estado para apoyarme en momentos difíciles. Tomando prestadas las palabras de Camilla Läckberg: “debo dar las gracias a mis amigos, por supuesto, que, haciendo gala de la mayor paciencia, siguen ahí, a pesar de los largos periodos en que ni siquiera los llamo”; ustedes están ahí aunque no los vea, en los buenos y malos momentos. A los que se fueron antes: gracias por lo vivido, nos vemos del otro lado.

He confirmado que hay mucha fe al estudiar ciencia: se persevera a pesar de los fracasos anteriores y se confía en una verdad aunque no se entienda por completo, y creo que Dios nos permite admirar su creación cuando estudiamos ciencia. También pienso que, de una forma u otra, lo que llamamos suerte es la manera en que recibimos las bendiciones de Dios, y considero que la suerte me ha sonreído muchas veces.

He aquí, Dios es el que me ayuda; el Señor está con los que sostienen mi vida.

Salmo 54:4

Jehová con sabiduría fundó la tierra; afirmó los cielos con inteligencia. Con su ciencia los abismos fueron divididos, y destilan rocío los cielos.

Proverbios 3:19-20

Índice general

Índice de tablas

Índice de figuras

Lista de términos y abreviaturas

ABC. Condiciones de Frontera Absorbentes, del inglés Absorbing Boundary Condition.

Block. Bloque o conjunto de threads en el software.

CUDA. Arquitectura Unificada de Dispositivos de Cómputo, del inglés Compute Unified Device Architecture. Los CUDA cores son unidades de cómputo o núcleos de una GPU.

Device. Referente a los elementos de la GPU.

FDTD. Diferencias-Finitas en el Dominio del Tiempo, del inglés Finite-Difference Time-Domain. FDTD-Q se refiere al FDTD cuántico, del inglés quantum.

FFT. Transformada Rápida de Fourier, del inglés Fast Fourier Transform.

GPU. Unidades de Procesamiento Gráfico, del inglés Graphics Processing Unit.

Grid. Malla o número de blocks en el kernel.

Host. Referente a los elementos del CPU.

Kernel. Función definida en el device.

Loop. Ciclo o bucle de programación.

PML. Capa Perfectamente Acoplada, del inglés Perfectly Matched Layer.

Scattering. Fenómeno donde se absorbe energía y se re-emite en diferentes direcciones con diferente intensidad.

SM. Multiprocesador de gráficos y computación, del inglés Streaming Multiprocessor.

Stretching coordinate. Coordenada de estiramiento. Factor complejo utilizado para implementar una PML.

Thread. Hilo o subtarea del kernel.

TM. Transmisión. Puesto que es una cuestión de probabilidad, su escala va de 0 a 1.

Unified Memory. Memoria Unificada de CUDA. Unifica las memorias del Host y el Device.

Warp. Conjunto de threads que corren simultáneamente en un multiprocesador.

Resumen

El avance tecnológico en computación ha permitido que los cálculos cuánticos mejoren y que se pueda ser más exigente con la precisión de los resultados buscados. El método de Diferencias-Finitas en el Dominio del Tiempo, abreviado FDTD por sus siglas en inglés, es una herramienta que se ha aplicado en la ecuación de Schrödinger, permitiendo realizar estudios electrónicos con un enfoque cuántico de manera sencilla. Aunque el método se ha optimizado al paralelizarse para el caso electromagnético, la versión cuántica no ha sido suficientemente explorada.

En el presente trabajo se implementó el método cuántico para FDTD en una y dos dimensiones, y se realizó su paralelización mediante CUDA (Compute Unified Device Architecture), ambiente ideal para la paralelización ya que las Unidades de Procesamiento Gráfico (GPU) de Nvidia tienen un gran número de núcleos de procesamiento, lo que acelera los códigos de manera muy eficiente. Para generar condiciones de frontera abiertas, se utilizó una PML (Perfectly Matched Layer) de acuerdo a la técnica de Zheng, debida a su flexibilidad y sencillez de implementación. Las paralelizaciones redujeron los tiempos de cómputo de forma muy significativa, con resultados muy similares entre diferentes tarjetas gráficas, generando una buena alternativa para realizar cómputo científico.

Las reducciones en tiempo dependen de la complejidad del sistema a analizar y del número de iteraciones de la simulación. Las limitaciones que puede tener esta implementación, se deben más por el hardware a utilizar que por el algoritmo, como lo son los requerimientos de memoria o sus límites de diseño. Estas limitantes se desvanecen con la evolución de las GPU's y mejoras en CUDA, lo que proporciona un gran potencial a este nuevo algoritmo además del que tiene en este momento.

Abstract

Technological advances in computation have allowed quantum calculations improved and they can be more demanding with the precision of the searched results. Finite-Difference Time-Domain (FDTD) is a method that has been applied to the Schrödinger equation, allowing to make electronic studies with a quantum focus in a simple way. Although the method has been optimized by parallelization for the electromagnetic case, the quantum version has not been sufficiently explored.

In this work, quantum FDTD method was implemented in one and two dimensions, and was parallelized though CUDA (Compute Unified Device Architecture), an ideal environment for parallelization due to the large number of processing cores in Graphics Processing Unit (GPU) of Nvidia, giving a code acceleration in a very efficient way. For generating open boundary conditions, a PML (Perfectly Matched Layer) was used according to the Zheng technique, due to its flexibility and easy implementation. Computation time was significantly reduced with parallelizations, with similar results between different graphic cards, giving a good alternative for scientific calculations.

Time reduction depends on the complexity of the analyzed system and the number of iterations in the simulation. Limitations in this implementation are more due to the hardware than to the algorithm, as are memory requirements or design limits. This limitations decrease with GPU evolution and improvements in CUDA, providing a great potential to this algorithm in addition to the one it has at this moment.

Capítulo 1

Introducción

1.1. Química computacional

Desde principios del siglo pasado, se hizo evidente la necesidad de estudiar sistemas microscópicos, y algunos macroscópicos como los cristales, láseres y superconductores (De la Peña, 2014), con teorías más completas que las que ofrecía la física clásica, dando lugar al desarrollo de la mecánica cuántica para estudiar partículas como electrones, núcleos y moléculas (Levine, 2014). El surgimiento de la teoría cuántica y el entendimiento de fenómenos hasta ese momento sin explicación, ha permitido la predicción de propiedades físicas y químicas a través de medios matemáticos y no sólo experimentales, gracias a los cálculos y al desarrollo de la química computacional.

Para que la química computacional pueda tener predicciones más acertadas a los fenómenos, las metodologías para realizar los cálculos computacionales requieren ser entendidas y mejoradas para obtener softwares más avanzados. Las técnicas que se utilizan para calcular energías y geometrías de los sistemas se pueden dividir en: métodos de mecánica molecular, métodos semiempíricos, métodos ab initio, métodos DFT (Density Functional Theory) y métodos de dinámica molecular. Entre estas técnicas, los semiempíricos, ab initio y DFT basan sus cálculos en la ecuación de Schrödinger y sus aproximaciones. Conforme estos métodos usan más parametrizaciones basadas en la ecuación de Schrödinger, y no en otra teoría o en datos experimentales, los cálculos se vuelven más costosos por las necesidades de cómputo (Lewars, 2011).

A pesar del costo computacional alto de los métodos que más utilizan la ecuación de

Schrödinger, estos cálculos son muy útiles para sistemas nuevos o desconocidos ya que no requieren tantos datos experimentales. Por ejemplo, la conductancia en transistores ha obligado a una descripción a nivel atómico de las propiedades electrónicas por la progresiva reducción de las dimensiones de éstos y obliga a tener en cuenta fenómenos como el tunelaje de electrones que no se habían estudiado antes (Hess y Iafrate, 1992).

1.2. Ecuación de Schrödinger

La ecuación de Schrödinger describe la función de onda ψ o función de estado en el sistema cuántico, siendo su forma más sencilla la estacionaria o independiente del tiempo:

$$E\psi = -\frac{\hbar^2}{2m}\nabla^2\psi + V\psi \tag{1.1}$$

donde $\hbar = \frac{h}{2\pi}$, siendo h la constante de Planck, V es el potencial, E es la energía, y m es la masa de la partícula.

La función de onda contiene toda la información que es posible conocer sobre el sistema, aunque por sí misma no tenga significado físico. Al tener números positivos, negativos y complejos, la función de onda no indica nada en cuestión de posición, pero la mecánica cuántica postula que $\psi^*\psi = |\psi|^2$ indica la densidad de probabilidad de la partícula. Para facilitar los cálculos, se suele normalizar la probabilidad al espacio de simulación, de manera que:

$$\int \psi^*\psi dr = 1 \tag{1.2}$$

La ecuación 1.1 es la base para la mayoría de los métodos utilizados para software cuántico desarrollado. Sin embargo, con algunas excepciones, casi todos los sistemas cuánticos a estudiar tienen una evolución en el tiempo, por lo que la ecuación de Schrödinger para estados estacionarios o independientes del tiempo se completa al intercambiar el hamiltoniano por $i\hbar\frac{\partial\psi}{\partial t}$, quedando de la siguiente manera (De la Peña, 2014).

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2\psi + V\psi \tag{1.3}$$

La ecuación de Schödinger completa o dependiente del tiempo, es necesaria para

estudiar la dinámica cuántica, aunque es menos estudiada y más complicada que la independiente del tiempo por contener coeficientes imaginarios (De la Peña, 2014). Esta ecuación tiene gran relevancia en aplicaciones como la interacción de una molécula con la luz, ya que los campos electromagnéticos varían con el tiempo, o en los cálculos DFT para obtener espectros del ultravioleta.

A pesar de que la ecuación de Schrödinger puede ser resuelta de manera analítica para sistemas sencillos, la mayoría de los casos prácticos requieren que se encuentre una solución por un método numérico. La forma más simple de utilizar un método numérico se basa en la discretización de la ecuación diferencial, como lo es el método de las diferencias finitas, que adquiere mayor precisión con la ecuación continua conforme las diferencias finitas se definan más pequeñas (Datta, 2005).

Puesto que el uso de métodos de diferencias finitas puede tener un costo computacional alto (Dixon, Feller, y Peterson, 2012), ya que requiere un gran número de iteraciones para que los datos sean suficientemente precisos, algunos estudios están buscando formas para reducir el costo computacional sin afectar la precisión de los cálculos. La búsqueda para evitar este conflicto muchas veces requiere recursos computacionales de alto nivel como el uso de clústers, dificultando el acceso a este tipo de simulaciones. Sin embargo, la paralelización por medio de computación heterogénea, como en el uso de tarjetas gráficas en el ambiente CUDA, puede optimizar las simulaciones en cuestión de tiempo y accesibilidad, ya que la presencia de tarjetas gráficas en computadoras de uso personal es cada vez mayor.

En el siguiente capítulo se revisarán algunos trabajos donde se implementa el método de las Diferencias Finitas en el Dominio del Tiempo para estudiar la ecuación de Schrödinger. El capítulo 3 describirá los objetivos de la tesis mientras que el 4 abordará de forma general la metodología para alcanzarlos. Los esfuerzos realizados para implementar el método FDTD se desarrollarán en el capítulo 5, comparando los lenguajes de programación y las formas seriales o paralelizadas de los códigos en una y dos dimensiones. Por último, el capítulo 6 resumirá las observaciones y resultados obtenidos a lo largo del desarrollo de la tesis, así como las recomendaciones pertinentes para profundizar en la temática con futuros trabajos.

Capítulo 2

Antecedentes

En 1966, Kane Yee desarrolló el método de Diferencias Finitas en el Dominio del Tiempo, más conocido como FDTD, como una solución numérica de las ecuaciones de Maxwell que describen los fenómenos electromagnéticos. Este método se basa en la discretización de dichas ecuaciones, realizando aproximaciones por diferencias finitas para las ecuaciones diferenciales, siendo más exacto conforme la diferencia finita sea más pequeña. Debido a que las ecuaciones discretizadas generan cierta independencia temporal entre el campo eléctrico y el magnético, el método permite visualizar los cambios a través del tiempo por medio de iteraciones computacionales (Taflove y Hagness, 2005). Posteriormente, la metodología del FDTD se aplicó a la ecuación de Schrödinger, obteniendo un método cuántico que además provee la visualización temporal de las funciones de onda (D. M. Sullivan, 2000).

El FDTD cuántico, o FDTD-Q, ha sido aplicado desde entonces para estudiar sistemas nanométricos como puntos cuánticos, puertas lógicas (Nagel, 2009), transporte cuántico, modelado de nanodispositivos, óptica cuántica (Xiong y Sha, 2014), interacción entre partículas (Moxley, Byrnes, Fujiwara, y Dai, 2012), condensados Bose-Einstein (Farrell y Leonhardt, 2005) o pozos cuánticos (D. M. Sullivan y Citrin, 2005). Estos últimos son muy útiles para estudiar semiconductores y, a través de condiciones de frontera cerradas, se pueden obtener sus eigenfunciones y eigenvalores, aunque estos estudios requieren al menos 2 simulaciones y por lo tanto, mayor tiempo de cómputo. Otro motivo que incrementa el tiempo de cómputo es la reducción en el tamaño de la celda espacial y temporal de la discretización, necesario para aumentar la precisión de ciertos cálculos

(Sudiarta y Geldart, 2007; D. Sullivan y Citrin, 2001).

En cuanto a simulaciones electrónicas, también es común estudiar el estado estacionario para evitar el uso de Condiciones de Frontera Absorbentes (ABC por sus siglas en inglés). Sin embargo, para proveer una mejor dirección de los experimentos en electrónica cuántica, controlados principalmente por la masa efectiva y la energía potencial, es necesario tomar en cuenta el *scattering* y los fenómenos cuánticos, lo que requiere el estudio del estado abierto. Debido a que los transistores y capacitores que cuenten con alguna dimensión cuántica se encuentran en estado formativo, las simulaciones cuánticas cobran gran relevancia para realizar predicciones que puedan ser reproducidas por los experimentos (Biegel, 1997).

A pesar que el FDTD-Q tiene menor costo computacional que la versión electromagnética, se debe evitar el incremento en el número de iteraciones de forma innecesaria por un paso temporal demasiado grande en relación al paso espacial elegido, pero el paso temporal debe ser lo suficientemente pequeño para que exista estabilidad en la simulación. Esta relación óptima ya fue analizada al definir el paso temporal crítico de modo que se cumpla con la siguiente ecuación,

$$\Delta t \leq \frac{\hbar}{\frac{\hbar^2}{m_e}\left[\frac{1}{\Delta x^2}+\frac{1}{\Delta y^2}+\frac{1}{\Delta z^2}\right]+V} \tag{2.1}$$

donde V es el potencial máximo a usar en la simulación. De esta manera se evita que exista una divergencia en el sistema después de cierto número de iteraciones (Soriano et al., 2004).

Cuando la simulación es unidimensional, la ecuación 2.1 se puede reducir a (Nagel, 2009):

$$\Delta t \leq \frac{\hbar}{\frac{\hbar^2}{m_e \Delta x^2}+\frac{V}{2}} \tag{2.2}$$

De forma más práctica, también se considera el criterio de estabilidad S_δ, que como indica la ecuación 2.3, establece la relación numérica que debe existir entre el paso temporal Δt y el paso espacial $\Delta\delta$ (δ es la dimensión x, y o z a utilizar), siendo un parámetro numérico que se logra implementar de manera más directa en la ecuación

de Schödinger (Xiong y Sha, 2014).

$$S_\delta = \frac{\Delta t}{\Delta \delta^2} \frac{\hbar}{2m_e} \tag{2.3}$$

Para evitar divergencia en las simulaciones, lo convencional es elegir pasos temporales y espaciales de manera que S_δ =0.125 (Dai, Li, Nassar, y Su, 2005), aunque en ocasiones se llega a tomar el valor de S_δ =0.1 (D. M. Sullivan, 2012). Ya que entre más pequeño se S_δ, se requiere un mayor número de iteraciones, se han desarrollado algoritmos con el propósito de relajar este criterio de estabilidad S_δ a números más grandes (Moxley III, Zhu, y Dai, 2012), aunque esto implica calcular otras variables adicionales, lo que también ha impulsado la búsqueda para paralelizar estos algoritmos mediante la programación con tarjetas gráficas (Wilson, 2019).

En virtud de que el método FDTD permite cierta independencia momentánea de las variables, tiene un gran potencial de ser paralelizado, como muestra la literatura para la versión electromagnética, principalmente en el ambiente CUDA (De Donno, Esposito, Tarricone, y Catarinucci, 2010; Livesey et al., 2012), que aprovecha los núcleos de las tarjetas gráficas. También se han hecho esfuerzos para paralelizar la versión cuántica del FDTD con el uso de un clúster, excluyendo las condiciones de frontera absorbentes por ser un cuello de botella para la paralelización (Strickland y Yager-Elorriaga, 2010). La versión dependiente del tiempo ha sido semiparalelizada en interfaz OpenMP para el estudio de guías de onda en una dimensión, recomendando el uso de más núcleos para reducir el tiempo de cómputo y la reescritura de los códigos en lenguajes como C++ o Python (Fang, 2019). Por otro lado, existen softwares basados en la ecuación de Schödinger que ya han sido acelerados mediante tarjetas gráficas como Gaussian, Abinit, Quantum Espresso y Octopus (NVIDIA, 2019), al igual que otros métodos cuánticos como Monte Carlo (Feldmann, Cummings, Kent IV, Muller, y Goddard III, 2008) y Chebyshev (Dziubak y Matulewski, 2012).

Capítulo 3

Objetivos

3.1. Objetivo general

El presente trabajo de tesis pretente implementar el algoritmo de Diferencias Finitas en el Dominio del Tiempo (FDTD) en la ecuación de Schrödinger dependiente del tiempo. Esta implementación mejorará los tiempos de cálculo de manera significativa por la programación mediante GPU's.

3.2. Objetivos específicos

- Implementar el algoritmo Q-FDTD en forma serial en lenguaje Python y C++ para una y dos dimensiones.
- Aplicar condiciones de frontera absorbentes del tipo PML.
- Paralelizar los algoritmos mediante el lenguaje de programación CUDA.
- Comparar los tiempos de cómputo en función del lenguaje utilizado y la tarjeta gráfica empleada en la paralelización.
- Validar el algoritmo FDTD en su versión cuántica reproduciendo sistemas reportados en la literatura.

Capítulo 4

Materiales y métodos

4.1. Materiales y equipo

Para las simulaciones en forma serial se utilizó el lenguaje Python 3.0 y C++, mientras que para las versiones paralelizadas solamente se utilizó CUDA C++. Para realizar las corridas en Python se contó con una computadora iMac con sistema operativo MacOS High Sierra Versión 10.13.6, procesador Intel Core I7 de 3.1 GHz y 16 GB de memoria. Para realizar corridas en C++ y en CUDA C++, se contaron con varias tarjetas gráficas de marca Nvidia, dos de ellas disponibles en el laboratorio de química computacional de la Facultad de Ciencias Químicas (GeForce GTX 1050 y Tesla K20c). También se realizaron corridas mediante la plataforma de Google Colaboratory (Google Colab), un entorno de Jupyter Notebook que se ejecuta en la nube de forma gratuita. Por configuración predeterminada, Colab emplea un lenguaje de código en Python 3.0, además de permitir el aceleramiento mediante GPU's, permitiendo incluso la instalación de un entorno para ejecutar CUDA C++ a través de ciertos comandos. El uso de esta plataforma permite el acceso a la tarjeta Tesla K80, una de la más modernas y rápidas del mercado al momento. Recientemente Colab implementó además las tarjetas Tesla T4, Tesla P4 y Tesla P100 (Lee, 2019), que permiten utilizar el modelo de Memoria Unificada en CUDA, utilizada en la última parte de la tesis. Actualmente Colab asigna cualquiera de las 4 GPU de forma aleatoria y en dependencia de su demanda, por lo que si se requiere utilizar *Unified Memory* se debe cuidar que no se asigne la Tesla K80, que no tiene soporte para esta herramienta (Robson, 2019). La tabla 4.1 muestra los componentes de las tarjetas

gráficas utilizadas para realizar comparaciones de tiempo de cómputo en este trabajo.

Tabla 4.1: Comparación de las tarjetas gráficas utilizadas para paralelizar los códigos

Tarjeta gráfica	GeForce GTX 1050	Tesla K20c	Tesla K80
Número de GPU's	1	1	2
Núcleos CUDA	640	2496	4992
Arquitectura	Pascal	Kepler	Kepler 2.0
Capacidad memoria	2 GB	5 GB	24 GB
Ancho de banda	112 GB/s	208.0 GB/s	240.6 X2 GB/s
Velocidad base	1354 MHz	706 MHz	562 MHz
Velocidad Boost	1455 MHz	-	824 MHz

4.2. Métodos

Para obtener las evoluciones temporales de las funciones de onda electrónicas, se aplicó el método FDTD a la ecuación de Schrödinger dependiente del tiempo. Se utilizaron condiciones de frontera aborbentes al implementar una PML descrita por Zheng para el caso cuántico en el 2007.

Tanto en los sistemas unidimensionales como en los bidimensionales, se implementó una simulación en forma serial en lenguaje Python, posteriormente en C++ y finalmente se implementó la paralelización en CUDA C++. A continuación se describirán de forma más profunda los métodos empleados en el proyecto.

4.2.1. Método de las Diferencias-Finitas en el Dominio del Tiempo

Aunque los problemas cuánticos se pueden resolver en forma matricial, el costo computacional de esta metodología aumenta de forma considerable al pasar de una a dos o hasta tres dimensiones, por lo que se suele recurrir al método FDTD para estudios donde se requieren más dimensiones para un modelamiento correcto del problema (Soriano et al., 2004). El método FDTD discretiza la ecuación de Schödinger y la resuelve en un proceso iterativo, además de permitir la implementación de potenciales arbitrarios.

El método considera que la función de onda es de la forma $\psi = \psi_{real} + i\psi_{imag}$, así que sustituyendo en la ecuación 1.3, se separa la parte real e imaginaria para obtener las siguientes dos ecuaciones. La masa del electrón m se cambia a una masa efectiva m_e que dependerá del material en que se encuentre interactuando, siendo $m_e = m_{electrón}$ cuando se encuentre en el vacío.

$$\frac{\partial \psi_{real}(r,t)}{\partial t} = -\frac{\hbar}{2m_e}\nabla^2 \psi_{imag}(r,t) + \frac{V}{\hbar}\psi_{imag}(r,t) \tag{4.1}$$

$$\frac{\partial \psi_{imag}(r,t)}{\partial t} = \frac{\hbar}{2m_e}\nabla^2 \psi_{real}(r,t) - \frac{V}{\hbar}\psi_{real}(r,t) \tag{4.2}$$

Luego se define una malla de puntos discretos que muestra la función de onda en el espacio y tiempo, de manera que $\psi(r,t) \approx \psi^m(i,j,k) = \psi(i\Delta x, j\Delta y, k\Delta z, m\Delta t)$. Las derivadas en el tiempo se discretizan con una diferencia centrada de segundo orden (Xiong y Sha, 2014), por lo que:

$$\frac{\partial \psi(r,t)}{\partial t} \approx \frac{\psi^{m+1}(i,j,k) - \psi^m(i,j,k)}{\Delta t} \tag{4.3}$$

donde m es el número de iteración del algoritmo. Mientras que el operador Laplaciano de segundo orden para una dimensión se aproxima de la siguiente forma.

$$\frac{\partial^2 \psi_{real}}{\partial x^2} \approx \frac{\psi^m_{real}(i+1) - 2\psi^m_{real}(i) + \psi^m_{real}(i-1)}{\Delta x^2} \tag{4.4}$$

$$\frac{\partial^2 \psi_{imag}}{\partial x^2} \approx \frac{\psi^{m+1/2}_{imag}(i+1) - 2\psi^{m+1/2}_{imag}(i) + \psi^{m+1/2}_{imag}(i-1)}{\Delta x^2} \tag{4.5}$$

La ecuación 4.5 indica el índice fraccionario $m+1/2$ para referirse a la media iteración de la parte imaginaria de ψ, ya que se realiza entre la iteración m y la $m+1$ de la parte real. Las discretizaciones anteriores se sustituyen en las ecuaciones 4.1 y 4.2, para poder dar las ecuaciones 4.6 y 4.7, que se implementan en un algoritmo iterativo para calcular la función de onda en un tiempo $t_0 + m\Delta t$.

$$
\begin{aligned}
\psi_{real}^{m+1}(i,j,k) =& \psi_{real}^{m} \\
&- \frac{\hbar\Delta t}{2m_e\Delta x^2}\left[\psi_{imag}^{m+1/2}(i+1,j,k) - 2\psi_{imag}^{m+1/2}(i,j,k) + \psi_{imag}^{m+1/2}(i-1,j,k)\right] \\
&- \frac{\hbar\Delta t}{2m_e\Delta y^2}\left[\psi_{imag}^{m+1/2}(i,j+1,k) - 2\psi_{imag}^{m+1/2}(i,j,k) + \psi_{imag}^{m+1/2}(i,j-1,k)\right] \\
&- \frac{\hbar\Delta t}{2m_e\Delta z^2}\left[\psi_{imag}^{m+1/2}(i,j,k+1) - 2\psi_{imag}^{m+1/2}(i,j,k) + \psi_{imag}^{m+1/2}(i,j,k-1)\right] \\
&+ \frac{\Delta t}{\hbar}V(i,j,k)\psi_{imag}^{m+1/2}(i,j,k)
\end{aligned}
\tag{4.6}
$$

$$
\begin{aligned}
\psi_{imag}^{m+1/2}(i,j,k) =& \psi_{real}(i,j,k)^{m-1/2} \\
&+ \frac{\hbar\Delta t}{2m_e\Delta x^2}\left[\psi_{real}^{m}(i+1,j,k) - 2\psi_{real}^{m}(i,j,k) + \psi_{real}^{m}(i-1,j,k)\right] \\
&+ \frac{\hbar\Delta t}{2m_e\Delta y^2}\left[\psi_{real}^{m}(i,j+1,k) - 2\psi_{real}^{m}(i,j,k) + \psi_{real}^{m}(i,j-1,k)\right] \\
&+ \frac{\hbar\Delta t}{2m_e\Delta z^2}\left[\psi_{real}^{m}(i,j,k+1) - 2\psi_{real}^{m}(i,j,k) + \psi_{real}^{m}(i,j,k-1)\right] \\
&- \frac{\Delta t}{\hbar}V(i,j,k)\psi_{real}^{m}(i,j,k)
\end{aligned}
\tag{4.7}
$$

Analizando las ecuaciones anteriores, se necesitan los valores anteriores de la celda a calcular y de las celdas vecinas, dando lugar a la celda del FDTD cuántico, representada en la figura 4.1.

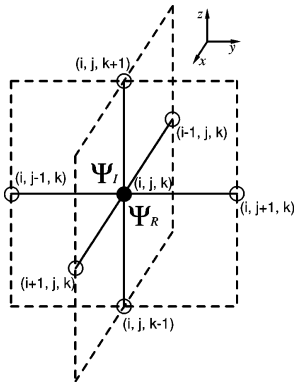


Figura 4.1: Celda del FDTD-Q. Tomado de Soriano et al. (2004)

Es evidente que entre más pequeños sean los pasos temporales y espaciales, los cálculos se asemejarán más a la función continua y se tendrán valores más realistas. Para obtener una buena discretización del problema, primero se selecciona el tamaño del paso espacial, que debe ser divisor de la longitud de onda más corta a estudiar, comúnmente

que pueda representar 10 puntos discretos (D. M. Sullivan, 2000). Después se selecciona el paso temporal crítico, que es el Δt máximo que evita una acumulación de error numérico en la simulación, dando estabilidad a la simulación. El error numérico se calcula comparando con problemas con solución analítica, para proporcionar convergencia al problema y determinar el tamaño requerido para la celda espacial (Soriano et al., 2004). Se ha demostrado que el error por discretización espacial es de segundo orden ($O(\Delta x^2)$), y que una buena resolución puede incrementar el tiempo de cómputo, ya que se reduce el tamaño de Δt para mantener la estabilidad, aumentando el número de iteraciones (Becerril, Guzmán, Rendón-Romero, y Valdez-Alvarado, 2008).

Cálculo de los observables

Aunque la función de onda ψ no tiene significado físico por sí misma, contiene toda la información física del sistema (D. M. Sullivan, 2012). Para encontrar dicha información se aplican operadores a la función, llamados observables si representan variables dinámicas. Los operadores elementales son los de posición, momento, energía, energía potencial, energía cinética y el hamiltoniano, cuyos valores medios o esperados son reales (De la Peña, 2014). Para este trabajo se utilizaron principalmente los observables de energía cinética y potencial.

El operador de la energía potencial es

$$\hat{V} = V(r) \tag{4.8}$$

por lo que su valor esperado será

$$\langle PE \rangle = \int_{-\infty}^{\infty} \psi^*(r) V(r) \psi(r) dr = \int_{-\infty}^{\infty} |\psi(r)|^2 V(r) dr \tag{4.9}$$

Ahora, para implementarla en el algoritmo FDTD, la ecuación 4.9 se discretiza de forma que

$$\langle PE \rangle = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} \sum_{k=1}^{N_z} [\psi_{real}^2(i,j,k) + \psi_{imag}^2(i,j,k)] V(i,j,k) \tag{4.10}$$

donde N_x, N_y y N_z corresponden al número de celdas espaciales en cada eje del espacio de simulación.

Por su parte, el operador para la energía cinética es

$$\hat{T} = \frac{-\hbar^2}{2m_e}\frac{\partial^2}{\partial r^2} \tag{4.11}$$

y su valor esperado está definido por

$$\langle KE \rangle = \int_{-\infty}^{\infty} \psi^*(r)\frac{-\hbar^2}{2m_e}\frac{\partial^2}{\partial r^2}\psi(r)dr = \frac{-\hbar^2}{2m_e}\int_{-\infty}^{\infty} \psi^*(r)\frac{\partial^2}{\partial r^2}\psi(r)dr \tag{4.12}$$

Para el caso unidimensional, la discretización de la parte real de $\langle KE \rangle$ queda de la siguiente manera

$$\langle KE \rangle = \frac{-\hbar^2}{2m_e\Delta x^2}\sum_{i=1}^{N_x}[\psi_{real}(i)\Delta_{real}(i) + \psi_{imag}(i)\Delta_{imag}(i)] \tag{4.13}$$

donde $\Delta_{real}(i)$ y $\Delta_{imag}(i)$ son los laplacianos, a saber

$$\Delta_{real}(i) = \psi^m_{real}(i+1) - 2\psi^m_{real}(i) + \psi^m_{real}(i-1) \tag{4.14}$$

$$\Delta_{imag}(i) = \psi^{m+1/2}_{imag}(i+1) - 2\psi^{m+1/2}_{imag}(i) + \psi^{m+1/2}_{imag}(i-1) \tag{4.15}$$

4.2.2. Condiciones de Frontera Absorbentes tipo PML

Para representar un sistema abierto, útil por ejemplo para estudiar estados resonantes o dinámica de reacciones, se necesitaría incrementar el espacio de simulación de modo que la propagación de ondas no tuviera interacción con reflexiones por los bordes de la simulación. Para evitar este gasto de recursos computacionales, o restricciones de memoria por el lenguaje a utilizar como Matlab (Nissen y Kreiss, 2011), se hacen ciertas aproximaciones para aplicar Condiciones de Frontera Absorbentes. Sin embargo, algunas metodologías son difíciles de implementar en la ecuación de Schrödinger por las relaciones de dispersión no lineales presentes, sobretodo en dos y tres dimensiones (Nagel, 2009).

Por su relativa fácil implementación y buena precisión, el uso de PML's es muy común para estudiar sistemas abiertos, aunque el ajuste de los parámetros que la definen puede ser una tarea ardua (Antoine, Lorin, y Tang, 2017). La interacción de las ondas con una

PML no presenta reflexiones en teoría, aunque existe una pequeña reflexión debida a la discretización del problema (Farrell y Leonhardt, 2005). El tiempo de cómputo para calcularlas aumenta de manera lineal, a diferencia de otros métodos con resultados muy similares a través de los pasos temporales. Además, su implementación y modificaciones a través del tiempo se realizan de forma sencilla y flexible (Mennemann y Jüngel, 2014).

La base para insertar una PML es diseñar una capa en los bordes del espacio de simulación que absorba ondas sin presentar reflexión. Fue desarrollada primero en algoritmos de FDTD electromagnético para perfeccionar una técnica llamada de Capa Acoplada (Matched Layer o ML), evitando así reflexiones que exhibía para algunos casos (Berenger, 1994). La técnica de la PML se mejoró luego al considerar un factor complejo llamado *coordinate stretching*, que la vuelve susceptible a la paralelización, ya que algunas implementaciones resultan en un cuello de botella para los códigos (Chew y Weedon, 1994). Este principio se aplicó después para otros problemas diferentes al electromagnético, entre ellos el cuántico (Zheng, 2007). Aunque la técnica de Zheng se desarrolló para casos unidimensionales, su extensión a más dimensiones es de fácil desarrollo, lo que suele ser complicado para otros métodos porque las condiciones de frontera tienen una geometría en dos o tres dimensiones. De esta manera, la ecuación de Schrödinger con una PML del tipo stretching coordinate queda de la siguiente forma.

$$\frac{\partial \psi}{\partial t} = i\frac{\hbar}{2m_e}c\frac{\partial}{\partial r}\left(c\frac{\partial \psi(r,t)}{\partial r}\right) - \frac{i}{\hbar}V(r)\psi(r,t) \tag{4.16}$$

donde c es la ecuación de la PML definida por la ecuación 4.17, que contiene el número complejo R, definido por la ecuación 4.18, y la función de absorción $\sigma(r)$, definida por la ecuación 4.19.

$$c = \frac{1}{R\sigma(r) + 1} \tag{4.17}$$

$$R = e^{i\pi/4} = \frac{1+i}{\sqrt{2}} \tag{4.18}$$

$$\sigma(r) = \sigma_0(r - r_{pml})^2 \tag{4.19}$$

La resta $(r - r_{pml})$ representa la distancia que separa al vector de posición $\vec{r}$ con el inicio de la PML más cercana, es decir la capa de la malla de simulación que empieza a absorber las señales. Si $\vec{r}$ está fuera de la región definida para la PML, entonces $\sigma(r) = 0$,

por lo que $c = 1$.

Posteriormente, D. M. Sullivan y Wilson definieron el parámetro γ, de la siguiente forma

$$\gamma(r) = \left(\frac{1}{R\sigma(r) + 1} \right)^2 \tag{4.20}$$

por lo que la ecuación 4.16 se reescribe como la ecuación 4.21, donde $\gamma_{real} = 1$ y $\gamma_{imag} = 0$ si la región está fuera de la PML.

$$\frac{\partial \psi}{\partial t} = i \frac{\hbar}{2m_e} \left(\gamma(r) \frac{\partial^2 \psi(r,t)}{\partial r^2} \right) - \frac{i}{\hbar} V(r) \psi(r,t) \tag{4.21}$$

Aunque esta técnica no presenta reflexiones para potenciales constantes, existe una reflexión intrínseca para potenciales variables en el espacio y tiempo. A pesar de ello, las soluciones aproximadas del método tiene precisiones con errores menores al $2\,\%$, aumentando al principio de la simulación y manteniéndose constante en el tiempo, además de que el error se reduce al disminuir la discretización. La efectividad de la PML dependerá del tamaño de ésta y el valor de σ_0, que es un factor de fuerza de absorción. Una PML grande tendrá una mejor absorción que una pequeña, tendiendo a un máximo en precisión y con un impacto más efectivo en ésta que cambiar el valor de σ_0. Por ello, lo más común es elegir primero el tamaño de la PML, de acuerdo al problema a tratar, y después se varía el valor de σ_0 hasta obtener la precisión deseada (Bramble y Pasciak, 2013). Se ha demostrado que el error disminuye al aumentar el valor de σ_0 para potenciales constantes y al disminuirlo para potenciales variables (Zheng, 2007).

4.2.3. Programación en CUDA

Para escribir los códigos en paralelo de este trabajo, se utilizó la plataforma de computación en paralelo CUDA C++. CUDA es un modelo de computación heterogénea que facilita la paralelización de los algoritmos ya que el CPU trabaja en conjunto con la GPU que posee el equipo (Sanders y Kandrot, 2011). Este ambiente de programación fue creado por Nvidia, incluyendo herramientas y librerías de C/C++, y utiliza las tarjetas gráficas desarrolladas por dicha empresa.

La computación en paralelo basada en GPU's puede reducir el tiempo de cómputo de algunas tareas en varios órdenes de magnitud, debido a que una GPU contiene

cientos o miles de unidades de cómputo, llamados núcleos o CUDA cores, mientras que una CPU moderna contiene una o varias. Para lograr esto, el código debe dividirse en un gran número de subtareas independientes, que se ejecuten de forma simultánea y no de forma serial. De esta forma, el algoritmo posee un código en el *host* para las tareas que no son independientes, que se refiere al CPU y a su memoria, y otro código en el *device* para ejecutar las subtareas, que se refiere a la GPU y su memoria. La comunicación entre las memorias del host y del device se realiza con la función `cudaMemcpy()`, copiando las variables requeridas en ambas direcciones. Equivalente a la asignación y liberación de memoria en C++, el device utiliza las funciones `cudaMalloc()` y `cudaFree()`, respectivamente.

Para representar la arquitectura del device en el hardware, se define un tamaño de *grid* en el software. Esta grid se divide en *blocks* y éstos a su vez en *threads* o hilos, que dependen del número de subtareas en que se dividió el algoritmo. Cada thread tiene un índice de identificación que provee CUDA, ver figura 4.2, reemplazando a los índices que tuviera un *loop* serial, por lo que cada thread se calcula en un núcleo de la GPU. En las corridas, los threads se agrupan en grupos llamados *warps*, que corresponden al número de núcleos presentes en cada streaming multiprocessor (SM).

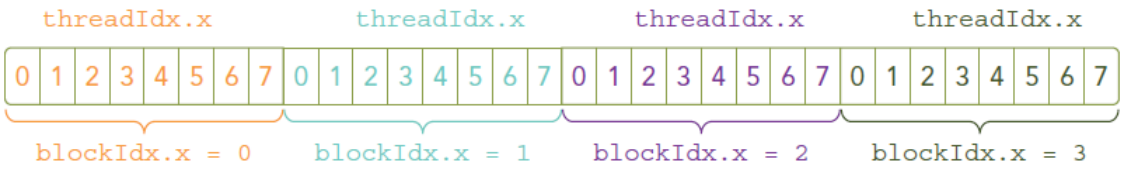


Figura 4.2: Identificación de índices en threads y blocks para una grid de 4 blocks y 8 threads por block. Tomado de Cheng et al. (2014).

Ya que el número de subtareas no tiene por qué corresponder al número de núcleos presentes en el device, se recomienda que el número de celdas sea un múltiplo de 16, incluso si algunos threads están vacíos, para optimizar el acceso de memoria por el tamaño de los warps (Chi, Liu, Weber, Li, y Crozier, 2011). La figura 4.3 ilustra esta definición de grid cuando el número total de elementos o subtareas es menor al número de threads definidos en la grid.

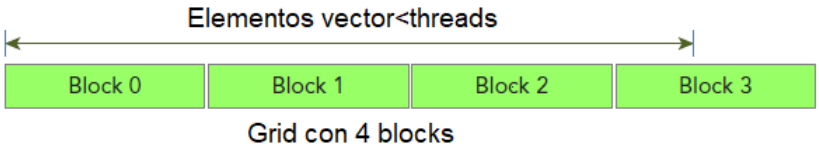


Figura 4.3: Diferencia entre el número de elementos que analizará el kernel y el número de threads de la grid. Adaptado de Cheng et al. (2014).

La figura 4.4 muestra la comparación entre elementos del software y del hardware de CUDA. La principal diferencia radica en que una GPU con más recursos de hardware, terminará más rápido las instrucciones del kernel porque los blocks y threads se distribuyen en los múltiples SM, proveyendo escalabilidad en los algoritmos, lo que se ilustra en la figura 4.5

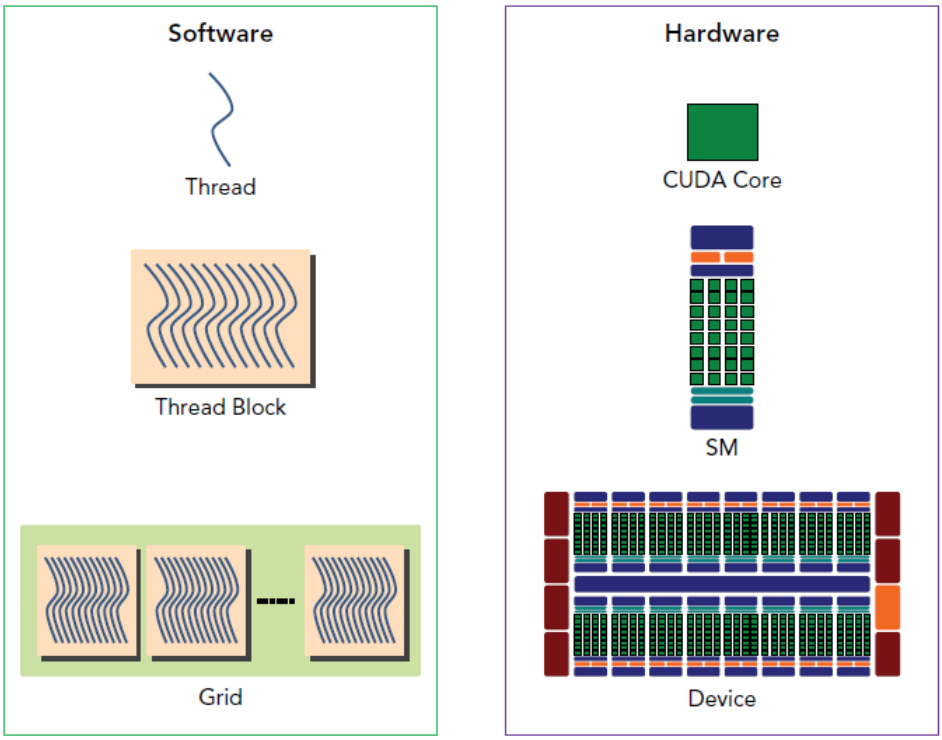


Figura 4.4: Comparación entre elementos del software y hardware en una GPU. Tomado de Cheng et al. (2014).

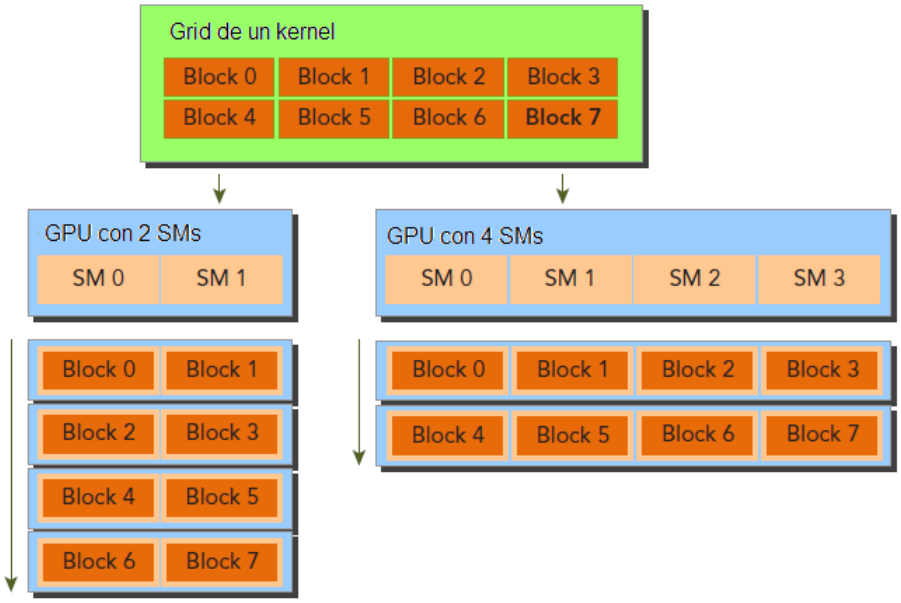


Figura 4.5: Ejemplo de escalabilidad en la arquitectura CUDA. Adaptado de Cheng et al. (2014).

La razón por la que una GPU puede optimizar las tareas mejor que un CPU es el manejo de latencia, que es el tiempo que espera un núcleo para recibir datos, que depende de la distancia entre éste y la localización de memoria donde se guarda el dato. Aunque el CPU está diseñado para minimizar la latencia, asignando mucho espacio para guardar datos que se pueden accesar rápidamente, el GPU esconde la latencia, así que asigna muchos núcleos a las tareas y, si los datos no están disponibles en un warp, el SM apaga ese warp y trabaja con otro que sí tenga datos disponibles. Para que los códigos puedan ser utilizados para casi cualquier GPU, algunas funciones no se utilizaron porque no están disponibles para todos los GPU's, tales como el uso de la doble precisión, impresiones en pantalla o mejoras en el manejo de memoria (Livesey et al., 2012; Storti y Yurtoglu, 2015).

Para crear las subtareas que correrán en paralelo, se define una función en el device llamada *kernel*. Las dimensiones de la grid y los block se definen al hacer la llamada del kernel y depende de cada problema. El siguiente es un ejemplo de cómo se manda llamar un kernel, incluyendo las variables que necesita para su ejecución.

```
nombre_kernel <<< gridDim, blockDim >>> (variable1, variable2, ...) ;
```

La definición del kernel tiene la siguiente estructura. El arreglo `*d_arreglo` debe tener forma unidimensional para explotar la linealidad de la memoria global del GPU. Para

simplificar la identificación de threads se realiza el cambio de variable de `x` por los `Idx` de los threads y blocks.

```
__global__ void nombre_kernel (tipo_de_variable_en_arreglo *d_arreglo,
tipo_de_variable variable){
  int x= blockIdx.x * blockDim.x + threadIdx.x;
  if (condicion_a_cumplir_threads){
  instruccion;
  }
}
```

El `__global__` indica que el tipo de memoria al que accede la GPU es memoria global. Aunque la memoria compartida (shared) y de registros son más fáciles de acceder, la memoria global es necesaria para grandes arreglos y está disponible para threads en diferentes blocks (De Donno et al., 2010), por lo que se recomienda dejar sólo las variables constantes en memoria constante para un acceso más eficiente. También se recomienda limitar la comunicación entre el CPU y el GPU para no consumir tiempo en la transferencia de datos (Chi et al., 2011). Las mediciones de tiempo de cómputo de los algoritmos se compararán en serie y en paralelo con la herramienta de `cudaEvent`.

En la última sección se utilizó la Memoria Unificada de CUDA, que optimiza los accesos de memoria del Host y del Device, y aunque sólo se utiliza para el último código, su traducción a memoria tradicional es bastante sencilla y sólo se dejó escrito en esta forma para ejemplificar las diferencias en escritura.

Capítulo 5

Resultados y discusión

5.1. Programación unidimensional

Para implementar una PML de acuerdo a la técnica de Zheng, la ecuación 4.21 en su forma unidimensional es la siguiente:

$$\frac{\partial \psi}{\partial t} = i\frac{\hbar}{2m_e}\left(\gamma(x)\frac{\partial^2 \psi(x,t)}{\partial x^2}\right) - \frac{i}{\hbar}V(x)\psi(x,t) \tag{5.1}$$

La ecuación 5.1 se separa en su parte real e imaginaria, recordando que tanto la función de onda ψ como el stretching coordinate γ son funciones complejas, dando lugar a las ecuaciones discretizadas 5.2 y 5.3.

$$\psi_{real}^{m+1}(i) = \psi_{real}^{m}(i) + \frac{\Delta t}{\hbar}V(i)\psi_{imag}^{m+1/2}(i) - \frac{\hbar}{2m_e}\frac{\Delta t}{(\Delta x)^2}[\gamma_{real}(i)\Delta_{imag(i)} + \gamma_{imag}(i)\Delta_{real(i)}] \tag{5.2}$$

$$\psi_{imag}^{m+3/2}(i) = \psi_{imag}^{m+1/2}(i) - \frac{\Delta t}{\hbar}V(i)\psi_{real}^{m+1}(i) + \frac{\hbar}{2m_e}\frac{\Delta t}{(\Delta x)^2}[\gamma_{real}(i)\Delta_{real(i)} - \gamma_{imag}(i)\Delta_{imag(i)}] \tag{5.3}$$

donde

$$\Delta_{real(i)} = \psi_{real}^{m}(i+1) - 2\psi_{real}^{m}(i) + \psi_{real}^{m}(i-1) \tag{5.4}$$

y

$$\Delta_{imag(i)} = \psi_{imag}^{m+1/2}(i+1) - 2\psi_{imag}^{m+1/2}(i) + \psi_{imag}^{m+1/2}(i-1) \tag{5.5}$$

Para obtener la parte real y la imaginaria del stretching coordinate γ en cualquier dimensión, se deben identificar los elementos que le dan su carácter complejo. Ya que σ_0 es el factor de fuerza de absorción, el cual es un parámetro numérico, entonces la función de absorción $\sigma(r)$ es un valor real y la separación de elementos reales e imaginarios de γ se puede escribir en función de ella.

Partiendo de la definición compleja de R, dada por la ecuación 4.18, la función compleja de γ dada por la ecuación 4.20 se puede escribir de la siguiente manera.

$$\gamma(r) = \left(\frac{1}{\frac{\sigma(r)}{\sqrt{2}} + 1 + \frac{\sigma(r)}{\sqrt{2}} i} \right)^2 \tag{5.6}$$

Desarrollando el cuadrado de la función, γ complejo sería de la forma

$$\gamma(r) = \frac{1}{\left(\left(\frac{\sigma(r)}{\sqrt{2}} + 1 \right)^2 - \frac{\sigma(r)^2}{2} + 2 \left(\frac{\sigma(r)}{\sqrt{2}} + 1 \right) \left(\frac{\sigma(r)}{\sqrt{2}} \right) i \right)^2} \tag{5.7}$$

Al separar la parte real e imaginaria de γ, se obtiene

$$\gamma_{real}(r) = \frac{\left(\frac{\sigma(r)}{\sqrt{2}} + 1 \right)^2 - \frac{\sigma(r)^2}{2}}{\left[\left(\frac{\sigma(r)}{\sqrt{2}} + 1 \right)^2 - \frac{\sigma(r)^2}{2} \right]^2 + 4\frac{\sigma(r)^2}{2} \left(\frac{\sigma(r)}{\sqrt{2}} + 1 \right)^2} \tag{5.8a}$$

$$\gamma_{imag}(r) = \frac{-2 \left(\frac{\sigma(r)}{\sqrt{2}} + 1 \right) \left(\frac{\sigma(r)}{\sqrt{2}} \right)}{\left[\left(\frac{\sigma(r)}{\sqrt{2}} + 1 \right)^2 - \frac{\sigma(r)^2}{2} \right]^2 + 4\frac{\sigma(r)^2}{2} \left(\frac{\sigma(r)}{\sqrt{2}} + 1 \right)^2} \tag{5.8b}$$

Debido a que las ecuaciones anteriores fueron escritas en función de $\sigma(r)$, y ésta a su vez depende de los parámetros que definen la PML, la elección de estos parámetros definirán la eficiencia de absorción que tendrá la PML.

5.1.1. Elección de los parámetros de la PML

Existen dos parámetros importantes en la definición de la PML: el valor numérico de σ_0 y el tamaño en celdas de la PML. Según el trabajo realizado por Zheng (2007), los parámetros llegan a un máximo en precisión cuando el número de celdas de la PML incrementa y cuando el valor de σ_0 disminuye, aunque esto último sólo si se trabaja con potenciales variables. También se ha observado que tiene un mayor impacto en la precisión el tamaño de la PML que un cambio en el valor de σ_0. Por ello, para realizar las simulaciones de este trabajo, se fijó primero el tamaño de la PML y luego se evaluó su eficiencia con distintos valores de σ_0. Para que el stretching coordinate γ sea adimensional, se ha de suponer que el valor de σ_0 tiene unidades de acuerdo a la discretización espacial, que aquí se obviarán, pero se espera un cambio en su valor óptimo si Δx cambia.

En una dimensión, la función de absorción dada por 4.19 se transforma en

$$\sigma(i) = \sigma_0(i - n_{pml})^2 \tag{5.9}$$

donde n_{pml} puede ser la PML izquierda o derecha, según la que esté más cerca del vector de posición (i), que denominaremos L_{pml} y R_{pml}, respectivamente.

Para analizar la eficiencia de la absorción de la PML, se implementó un algoritmo FDTD a un pulso gaussiano de $0{,}0512$ eV en un medio de GaAs. En nuestras simulaciones se centró el pulso a la mitad de un sistema de 120 nm, que se puede observar en la figura 5.1. Es relevante mencionar que para estos análisis se utilizó la versión paralelizada del apéndice I, ya que en las versiones en Python y C++ empieza a tener inconvenientes en el tiempo de cómputo en casos extremos. Las aceleraciones optenidas al utilizar la versión paralelizada se analizarán más adelante, pero todos los cálculos de esta sección tardan menos de un minuto en producirse en CUDA.

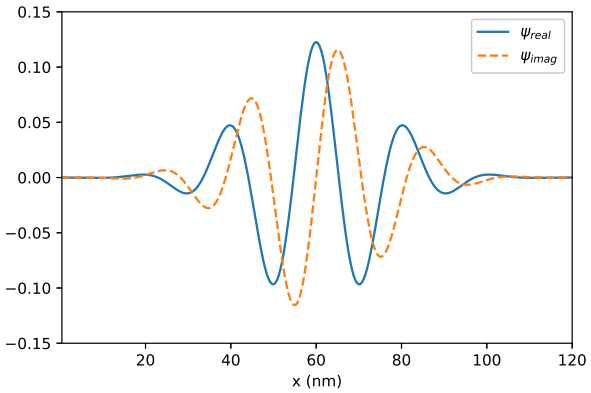


Figura 5.1: Inicialización de un pulso gaussiano con KE_0 =0.0512 eV en un medio de GaAs.

Para optener un valor óptimo de σ_0, se aplicaron distintos valores de éste a una PML de 20 nm y se comparó con la probabilidad en el espacio que existiría con un sistema de 6000 nm sin PML. Esta malla es lo suficientemente grande para no llegar a tener interacción con los bordes en un tiempo considerablemente alto, puesto que con un tiempo de 3 ps el pulso tiene el comportamiento de la figura 5.2. Usando ese sistema como referencia, se elimina el error producido por la discretización espacial y temporal, evaluando el efecto únicamente del comportamiento de la PML.

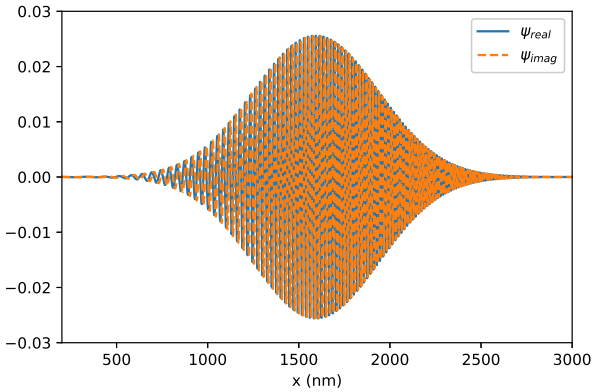


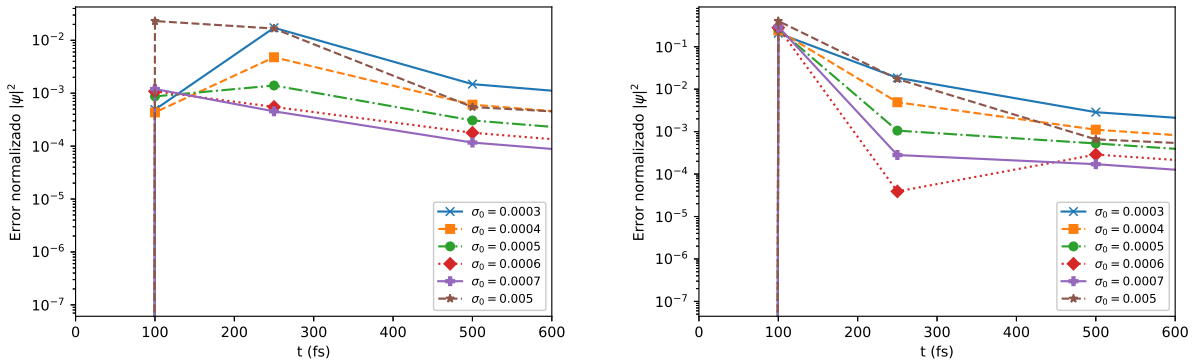
Figura 5.2: Comportamiento de una simulación sin PML que es suficientemente grande para no interaccionar con los bordes.

El error de la probabilidad fue medido de forma normalizada de acuerdo a la siguiente ecuación.

$$\text{Error normalizado}_t = \frac{||\psi_{pml}|_t^2 - |\psi_{ref}|_t^2|}{|\psi|_0^2} \tag{5.10}$$

donde $|\psi_{pml}|_t^2$ es la densidad de probabilidad obtenida al aplicar la PML, $|\psi_{ref}|_t^2$ es la obtenida con el sistema de referencia de 6000 nm y $|\psi|_0^2$ es la inicial del pulso.

La figura 5.3 muestra el error normalizado producido en $|\psi|^2$ a través del tiempo. Es evidente que el error aumenta cuando se empieza a interaccionar con la PML y luego disminuye cuando la onda es absorbida casi por completo, por lo que un valor ideal de σ_0 debe minimizar el error a tiempos pequeños. Esta minimización no se logra ni con el valor más alto ni el más pequeño de σ_0, pero logra un comportamiento más o menos constante con valores de 0.0005 a 0.0007, sobretodo si se analiza únicamente la probabilidad sin el espacio que interactúa con la PML.



(a) $|\psi|^2$ se calcula de L_{pml} a R_{pml}

(b) $|\psi|^2$ se calcula de 0 a 120 nm

Figura 5.3: Error normalizado producido por aplicar una PML de 20 nm con un Δx =0.4 nm.

Aunque puede existir un estudio más profundo para escoger los parámetros óptimos de la PML cuántica, al graficar los valores complejos de γ se puede inferir que el valor de σ_0, o al menos su orden de magnitud, debe escogerse para que el mínimo en γ_{imag} se obtenga cerca de la mitad del tamaño de la PML. Como muestra la figura 5.4, valores más altos de σ_0 tienden a mover el mínimo de γ_{imag} hacia n_{pml}, mientras que valores pequeños lo moverán hacia los bordes de la simulación.

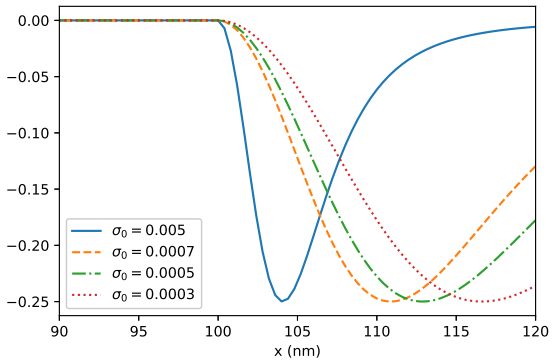
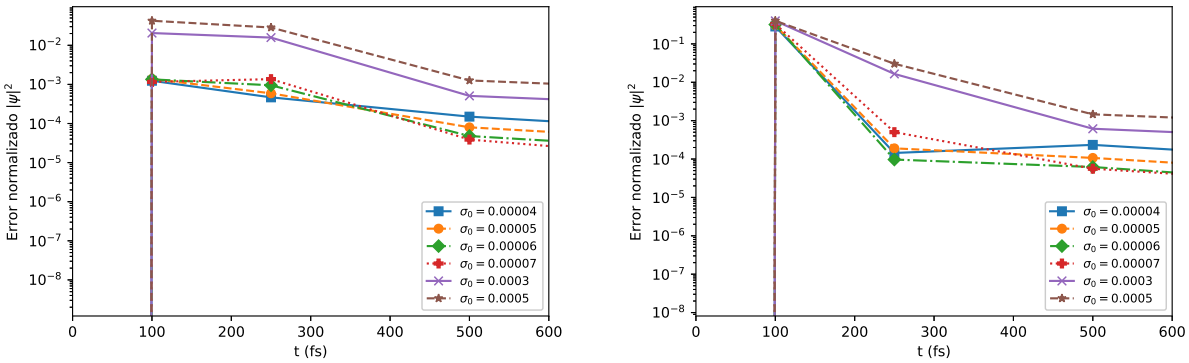


Figura 5.4: Localización del mínimo en γ_{imag} con Δx =0.4 nm y diferentes valores de σ_0.

Al analizar el mismo sistema, pero ahora con una discretización con Δx =0.1 nm, los valores óptimos para disminuir el error son totalmente diferentes, como indica la figura 5.5. En este caso los errores tienen un comportamiento muy similar cuando σ_0 es del mismo orden de magnitud, teniendo una mejor precisión con órdenes de 10^{-5} por lo que para obtener un valor medio se toma 0.00005 como el valor óptimo del sistema.

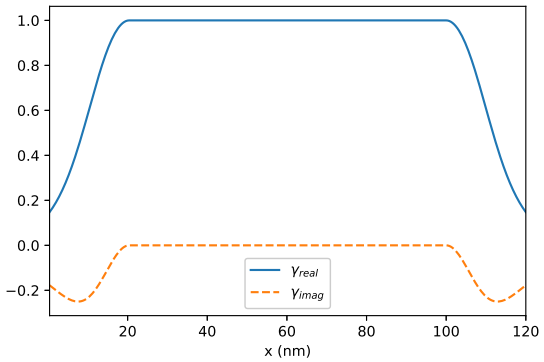


(a) $|\psi|^2$ se calcula de L_{pml} a R_{pml}

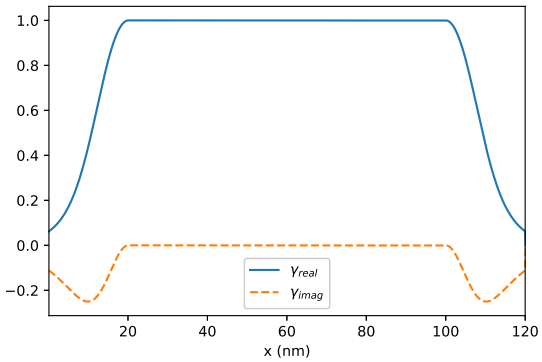
(b) $|\psi|^2$ se calcula de 0 a 120 nm

Figura 5.5: Error producido por aplicar una PML de 20 nm con un Δx =0.1 nm.

La figura 5.6 indica el comportamiento de γ complejo para diferentes discretizaciones espaciales. Se observa que, aunque tienen valores diferentes de σ_0, las diferencias en Δx hacen que tengan un comportamiento similar de forma global.



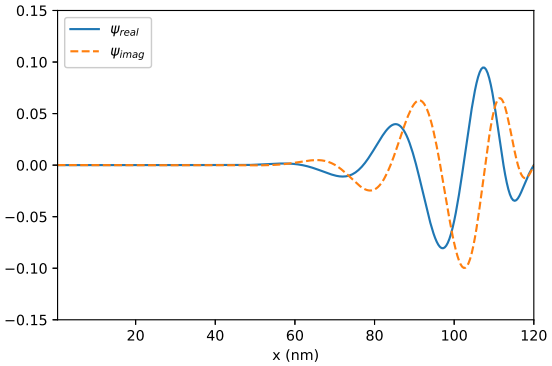
(a) σ_0 =0.0005 y Δx =0.4 nm



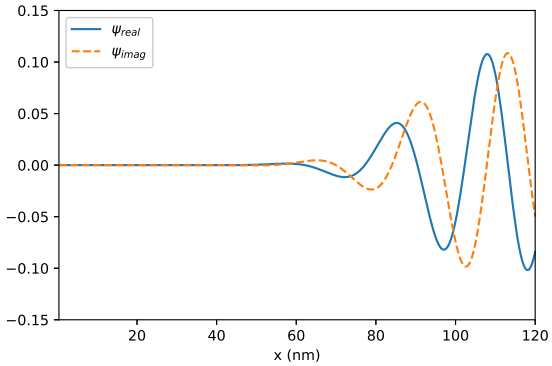
(b) σ_0 =0.00005 y Δx =0.1 nm

Figura 5.6: Comportamiento en una dimensión del stretching coordinate γ con 20 nm PML

Las figuras 5.7 y 5.8 muestran la comparación visual de la absorción del pulso gaussiano con la PML y el comportamiento del pulso en un sistema de 6000 nm, simulando una absorción perfecta.

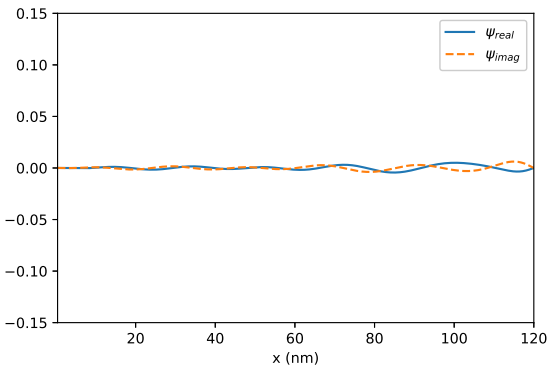


(a) PML=20 nm, σ_0 =0.0005 y Δx =0.4 nm

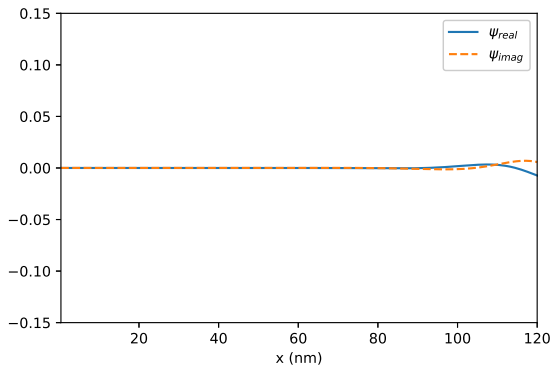


(b) Sistema de referencia con Δx =0.4 nm

Figura 5.7: Avance de la onda con KE_0 =0.0512 eV después de t =100 fs.



(a) PML=20 nm, σ_0 =0.0005 y Δx =0.4 nm



(b) Sistema de referencia con Δx =0.4 nm

Figura 5.8: Avance de la onda con KE_0 =0.0512 eV después de t =250 fs.

La tabla 5.1 indica la disminución en la probabilidad conforme la PML absorbe el pulso gaussiano con un valor de σ_0 =0.0005, comparado con el sistema de referencia de 6000 nm. También se comparan los resultados obtenidos por D. M. Sullivan y Wilson (2012), quienes definen una PML similar con un valor de σ_0 =0.005 pero mismo tamaño de PML y Δx. A pesar de la falta de datos para reproducir totalmente los resultados de la literatura, es evidente que nuestros parámetros tienen una mejor absorción ya que a tiempos más grandes existe menor reflexión y los valores de probabilidad en el espacio se acercan más a nuestro sistema de referencia.

Tabla 5.1: Comparación de $|\psi|^2$ de un pulso gaussiano con KE_0 =0.05 eV.

Tiempo (fs)	Python, CUDA y C++	D. M. Sullivan y Wilson (2012)	Sistema de referencia
0	1.000000	1.00002	1.000000
100	0.488341	0.43189	0.752548
250	0.002635	0.00559	0.001551
3000	0.0000006	No especificado	0.0000003

Otro motivo importante que aumenta el error en las simulaciones, es que la longitud de onda para obtener un pulso inicial de 0.05 eV está definida para ocupar 53 Δx, pero la PML está definida en 50 celdas. Por tanto, pulsos de mayor energía serán absorbidos más eficientemente.

5.1.2. Obtención de los observables

Para calcular el valor esperado de la energía potencial en su forma discreta, se simplifica la ecuación 4.10 a su forma unidimensional con la siguiente sumatoria en el espacio.

$$\langle PE \rangle = \sum_{i=1}^{N_x} [\psi_{real}^2(i) + \psi_{imag}^2(i)] V(i) \tag{5.11}$$

La forma unidimensional para la energía cinética ya había sido expresada mediante la ecuación 4.13. Además, para calcular la probabilidad de encontrar a la partícula en el espacio de simulación, se aplica la siguiente expresión.

$$|\psi|^2 = \sum_{i=1}^{N_x} [\psi_{real}^2(i) + \psi_{imag}^2(i)] \tag{5.12}$$

En todas las simulaciones se aplicará el criterio de normalización, donde $|\psi|^2 = 1$ en el tiempo $t = 0$.

5.1.3. Paralelización del sistema unidimensional

De forma general, el algoritmo cuántico tendrá la estructura que indica la figura 5.9, tanto en forma serial como paralelizada. La parte del código que consume más tiempo de cómputo es el ciclo temporal, debido al gran número de iteraciones cuando el tiempo de simulación aumenta, por lo que es la parte que se busca acelerar mediante la programación con GPU's. Dependiendo de la complejidad del problema, el tratamiento de datos también puede consumir un tiempo de cómputo considerable, al igual que la impresión de datos; sin embargo, muchas de las operaciones que se llevan a cabo en estas partes son forzosamente seriales o su ganancia al paralelizar no es tan significativa como el ciclo temporal, aunque también se llevaron a cabo paralelizaciones para optimizar las corridas, como se puede analizar en el código del apéndice I.3.

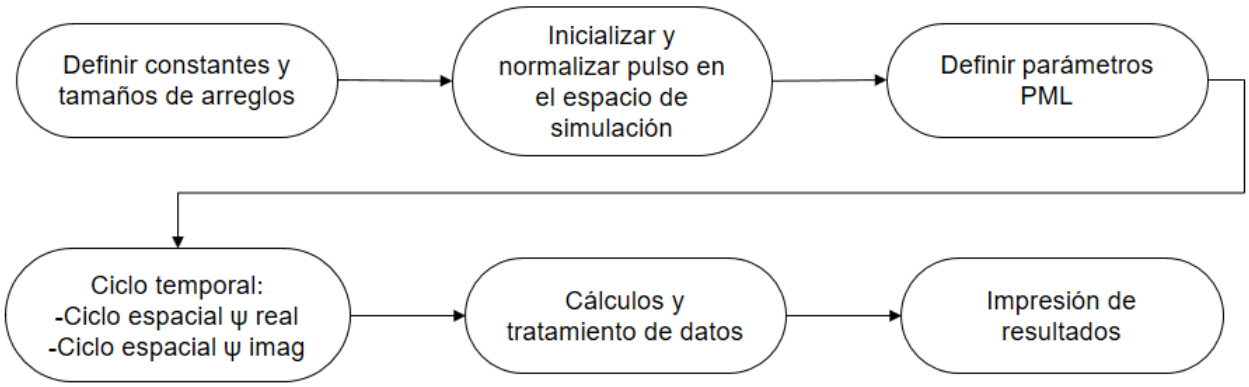


Figura 5.9: Diagrama de flujo del algoritmo Q-FDTD

Ya que cada iteración temporal conlleva un ciclo espacial en la parte real de la función de onda y otro ciclo para la parte imaginaria, se definió un kernel para cada ciclo espacial, siguiendo la analogía existente con la paralelización del caso electromagnético mediante CUDA. Esto se puede observar en el siguiente fragmento de código para un sistema unidimensional más simple, es decir, sin interacción con una PML y por tanto sin la adición de $\gamma(x)$ en las ecuaciones discretas.

```
int paso;
for(int m=1;m<=n_step;m++){
     paso=m;
     prl<<<grid,block>>>(d_prl, d_pim, Nx, dt, d_V, d_ra, paso);
     pim<<<grid,block>>>(d_prl, d_pim, Nx, dt, d_V, d_ra, paso);
}
```

El código es un ciclo temporal donde cada iteración tiene un kernel que sustituye a los ciclos espaciales para calcular la parte real e imaginaria de ψ, definidos como kernel “prl” e “imag”, respectivamente. Estos kernels son funciones paralelizadas de los ciclos espaciales, cuyas definiciones se muestran a continuación. La grid y el block deben ser lo suficientemente grandes para que su producto sea mayor al número de celdas espaciales del problema; tomando el caso anterior, una grid de 16 blocks de 32 threads cada uno es suficiente para calcular un sistema de 500 celdas o menos.

```
__global__ void prl (float *d_prl, float *d_pim, int Nx, float dt, float
                    *d_V, float*d_ra, int paso) {
```

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i< (Nx-1) && i>0) {
d_prl[i]=d_prl[i]-d_ra[i]*(d_pim[i-1]-2*d_pim[i]+d_pim[i+1])
         +(dt/hbar)*d_V[i]*d_pim[i];
}
}

__global__ void pim (float *d_prl, float *d_pim, int Nx, float dt, float
                     *d_V, float*d_ra, int paso) {
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i< (Nx-1) && i>0) {
d_pim[i]=d_pim[i]+d_ra[i]*(d_prl[i-1]-2*d_prl[i]+d_prl[i+1])
         -(dt/hbar)*d_V[i]*d_prl[i];
}
}
```

Este algoritmo es útil para estudiar sistemas cerrados, pero no para trabajar con PML's. Observando tanto la ecuación 5.2 como la 5.3, ambas contienen los laplacianos $\Delta_{real(i)}$ y $\Delta_{imag(i)}$, que cobran relevancia numérica en la región definida por la PML. Incluso en forma serial el algoritmo debe cuidar que, mientras se calcula un nuevo valor dentro de la PML, no se tome en cuenta el valor cambiado de la celda vecina $[i-1]$ o $[i+1]$, ya que éste afectaría el valor del laplaciano y a su vez el del nuevo valor ψ_{real}^{m+1} o $\psi_{imag}^{m+1/2}$, respectivamente. Aunque en forma serial esta discrepancia no produce grandes errores, la forma paralelizada comienza a ser afectada grandemente puesto que no se tiene control del orden en que la GPU calculará los nuevos valores. Para evitar esto, es necesario agregar dos *arrays* que mantengan los valores del paso temporal anterior hasta que se actualicen los nuevos; esta actualización se realiza con otros dos ciclos espaciales en ambas versiones. En la forma paralelizada los ciclos adicionales se sustituyen por dos kernels adicionales, de manera que el ciclo temporal se modifica como sigue.

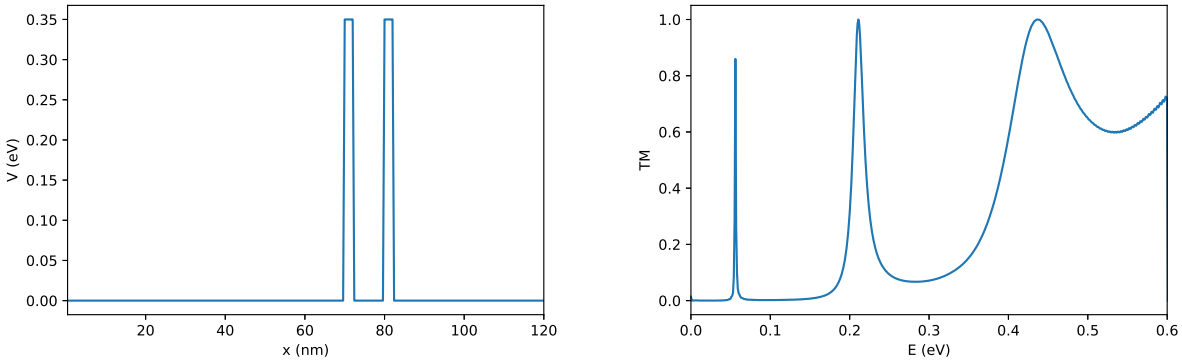
```
int paso;
for(int m=1;m<=n_step;m++){
paso=m;
```

```
prl_nueva<<<grid,block>>>(d_prl, d_prl_n, d_pim, Nx, dt, d_V, d_ra,
                          d_gamr, d_gami);
prl_reescribir<<<grid,block>>>(d_prl, d_prl_n, Nx, d_prl_in,
                               posicion_in, paso);
pim_nueva<<<grid,block>>>(d_prl, d_pim, d_pim_n, Nx, dt, d_V, d_ra,
                          d_gamr, d_gami);
pim_reescribir<<<grid,block>>>(d_pim, d_pim_n, Nx, d_pim_in,
                               posicion_in, paso);
}
```

Lógicamente, la adición de dos kernels o loops espaciales aumenta el tiempo de cómputo tanto en la versión serial como en la paralelizada. Sin embargo, se observó que con este procedimiento se obtienen mejores resultados en la absorción de ψ y en la resolución de picos para gráficas de transmisión, que se utilizan en secciones más adelante, además de que provee la estabilidad necesaria para la versión en CUDA. Es destacable mencionar que las reducciones en tiempo de cómputo en los ciclos temporales no fueron significativas contra la forma serial de C++. Incluso, en casos donde no hay un gran número de iteraciones, el tiempo de cómputo aumenta debido a la comunicación entre el host y el device. Este inconveniente se podría subsanar al realizar Paralelismo Dinámico en CUDA, una forma de paralelizar donde se corre un kernel “hijo” dentro de un kernel “padre”, minimizando la interacción con el CPU. Sin embargo, el Paralelismo Dinámico requiere una arquitectura Kepler (Cheng et al., 2014), que no poseen las tarjetas con usos más comerciales como las Geforce, además que el tiempo de cómputo necesario para la comunicación host-device es mínimo en comparación con la reducción de tiempo alcanzada con tiempos de simulación más grandes o mayor número de celdas espaciales. Por lo tanto, la estructura del código anterior es la base para paralelizar sistemas en cualquier dimensión.

5.1.4. Casos de estudio: Cálculo de transmisión a través de diferentes potenciales

A pesar de su simplicidad, el comportamiento de electrones en sistemas de una dimensión permite obtener las características esenciales de potenciales reales mediante modelos matemáticos más simples (De la Peña, 2014). El primer ejemplo de esto es el estudio de transmisión de un pozo finito o doble barrera de potencial, como se muestra en la figura 5.10b. La gráfica de transmisión indica que existe cierta probabilidad de transmitir ondas que estén en resonancia con la energías propias de la doble barrera, dando lugar al efecto de tunelamiento de energías menores a las barreras de potencial.

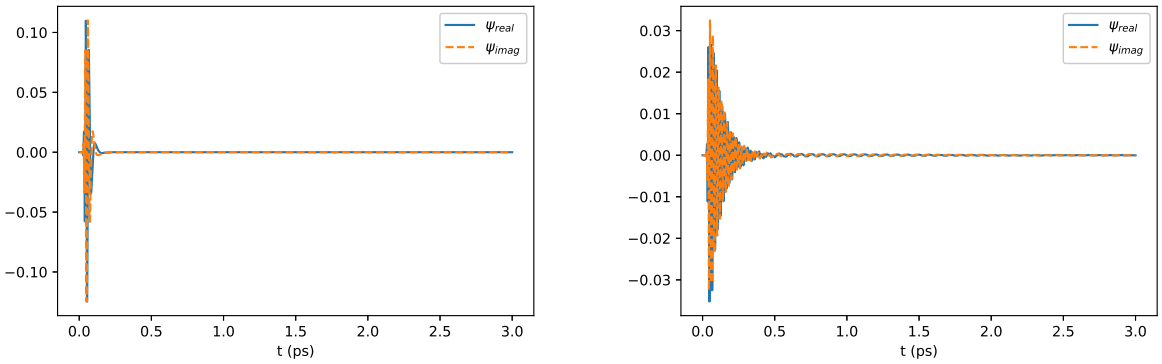


(a) Perfil de potencial de una doble barrera de 0.35 eV.

(b) Transmisión a través de un canal de potencial de 8 nm de ancho.

Figura 5.10: Problema unidimensional de un pozo finito o con doble barrera de potencial

Para obtener la gráfica de transmisión anterior, se realizó una simulación con una función onda de 0.204 eV de energía cinética inicial y se monitoreó a la derecha del potencial (95 nm) con $V = 0$ eV y con V =0.35 eV, para obtener una gráfica en el dominio del tiempo de la ψ, tanto a la entrada como a la salida del potencial (figura 5.11). Algunos algoritmos obtienen estos datos en una sola simulación, colocando dos monitores espaciales; sin embargo, en este algoritmo se optó por tener simulaciones separadas para evitar ruidos por rebote o requerir un análisis más profundo de la ubicación del monitor de entrada. Esto aumenta el tiempo de cómputo en las versiones paralelizada y serial, pero los datos son más limpios.



(a) Monitoreo de ψ de entrada.

(b) Monitoreo de ψ de salida.

Figura 5.11: Monitoreo a 95 nm.

Posteriormente, para obtener la transmisión del potencial se ejecuta una transformada de Fourier para obtener la información de ψ en el dominio de frecuencias. Esta operación se puede realizar con algoritmos propios de los lenguajes de programación con la llamada FFT (Fast Fourier Transform) o con un ciclo iterativo siguiendo la fórmula 5.13. En la versión de CUDA, se aplicó un ciclo computacional paralelizado que compite en rapidez con la FFT de Python, que puede además proporcionar mayor control en cuanto a las energías a estudiar o la obtención de más puntos que la FFT. La aplicación de ambas metodologías para la transformada de Fourier se encuentran en los apéndices.

$$\Psi(E) = \int_0^\infty \psi(t) e^{i(\frac{E}{\hbar})t} dt \tag{5.13}$$

Para obtener la sumatoria en el tiempo, hay que monitorear un tiempo lo suficientemente alto para que la función de onda no tenga cambios considerables, es decir, que se encuentre estabilizada, lo que se comprueba en las gráficas de la figura 5.11, al monitorear ψ durante una simulación de 3 ps. Para obtener un cálculo iterativo, la ecuación 5.13 se discretiza y se considera que tiene valores complejos, por lo que se obtienen las siguientes dos sumatorias.

$$\Psi_{real}[e] = \Sigma_0^\infty \psi_{real}[m] cos(\frac{e\Delta E}{\hbar}\Delta t) - \psi_{imag}(m) sen(\frac{e\Delta E}{\hbar}\Delta t) \tag{5.14a}$$

$$\Psi_{imag}[e] = \Sigma_0^\infty \psi_{real}[m] sen(\frac{e\Delta E}{\hbar}\Delta t) + \psi_{imag}(m) cos(\frac{e\Delta E}{\hbar}\Delta t) \tag{5.14b}$$

$$|\Psi[e]|^2 = (\Psi_{real}[e])^2 + (\Psi_{imag}[e])^2 \tag{5.14c}$$

Se observó que un ΔE =0.001 eV es suficiente para obtener igual o mejor resolución que la FFT. El índice (e) indica la iteración energética y (m) la iteración temporal, mientras que el infinito se considera la iteración temporal final.

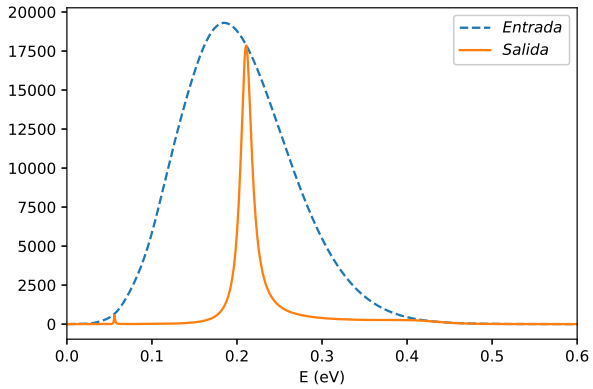
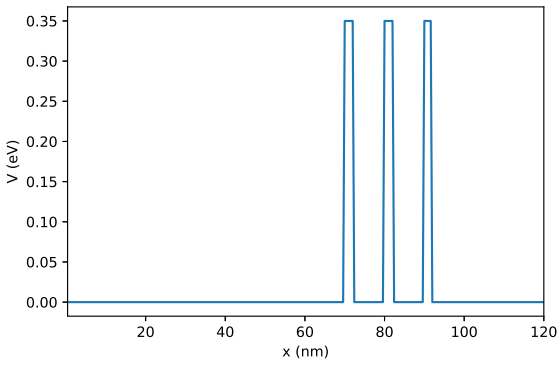


Figura 5.12: Dominio de energía de los monitores espaciales.

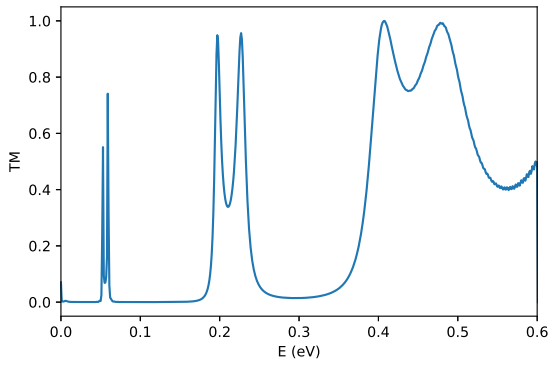
Una vez obtenida la información en el dominio de energías (figura 5.12), la transmisión (TM) se calcula con la ecuación 5.15.

$$TM[e] = \left| \frac{\Psi_{salida}[e]}{\Psi_{entrada}[e]} \right|^2 \tag{5.15}$$

La adición de otra barrera de potencial, como lo indica la figura 5.13b, provoca el desdoblamiento de las energías transmitidas del caso anterior (simulando el comportamiento que tienen los semiconductores en un efecto macroscópico, puesto que hay niveles de energía propios del material).



(a) Perfil de potencial de una triple barrera.

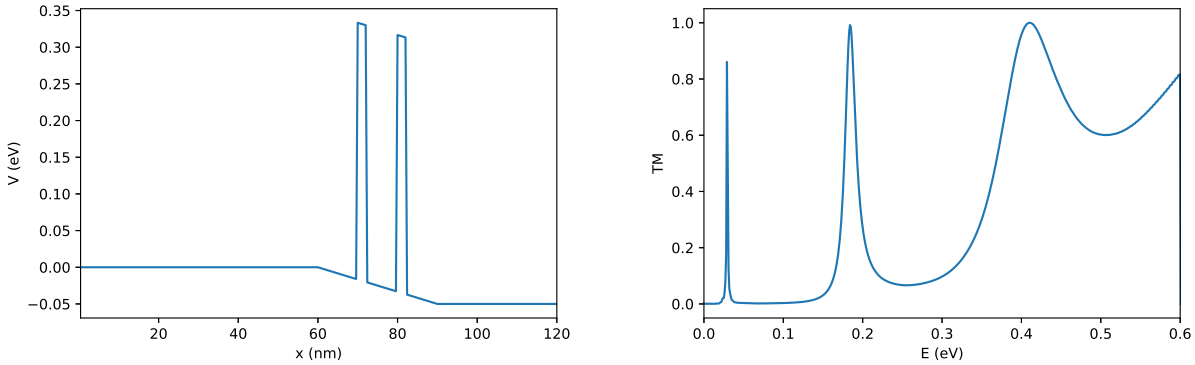


(b) Transmisión a través de una triple barrera.

Figura 5.13: Problema unidimensional de un doble pozo de potencial.

Otro efecto conocido es la adición de un campo eléctrico al sistema, simulado por un gradiente o rampa de potencial en cierta distancia, de acuerdo a la figura 5.14b. Esto indica que un campo eléctrico modificará las energías de transmisión, recorriendo los picos de transmitancia. Como el potencial negativo aumenta la energía cinética de la función de onda, se debe realizar un reescalamiento de la transmisión para que los picos resonantes del sistema tengan una correcta resolución (D. M. Sullivan y Wilson, 2012). Este reescalamiento se lleva a cabo por medio de la siguiente ecuación.

$$escala[e] = \sqrt{\frac{e\Delta E - V[monitor]}{e\Delta E}} \tag{5.16}$$



(a) Perfil de potencial de una doble barrera con la aplicación de un campo eléctrico.

(b) Transmisión a través de una doble barrera de potencial con un campo eléctrico.

Figura 5.14: Problema unidimensional que simula un pozo de potencial con un campo eléctrico.

La tabla 5.2 indica las frecuencias resonantes o picos en la gráfica de transmisión que se obtienen al definir los perfiles de potencial anteriores, indicando una gran similitud con la literatura. Las diferencias que existen están en el orden de las centésimas de eV y se desconoce si se deben, además, a alguna consideración que se omite en el escrito.

Tabla 5.2: Energías transmitidas (eV) de diferentes barreras de potencial

Perfil de potencial	CUDA/C++	D. M. Sullivan y Wilson (2012)
Pozo	0.06, 0.21, 0.44	0.05, 0.20, 0.42
Doble pozo (media de picos)	0.06, 0.21, 0.44	0.05, 0.20, 0.42
Pozo con campo eléctrico	0.03, 0.18, 0.41	0.03, 0.17, 0.40

Finalmente, el funcionamiento de un transistor se puede estudiar a partir de una analogía con un sistema unidimensional. En un transistor existe un canal con un material semiconductor entre dos contactos con materiales conductores (ver figura 5.15). Los contactos tienen una barrera energética que disminuye al encender el transistor, permitiendo un potencial de compuerta V_G y un potencial de drenado V_D. La interacción entre estos dos potenciales permite un flujo de electrones a través del canal, creando una corriente I y una conductancia G.

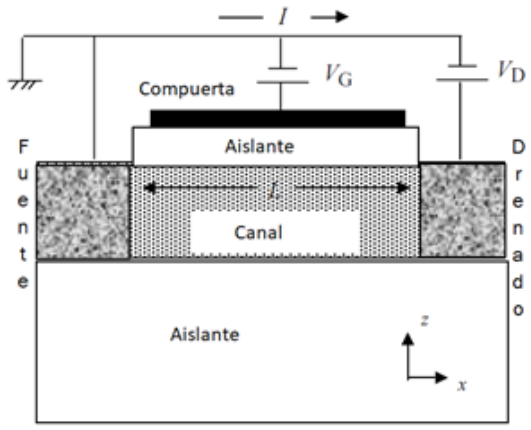
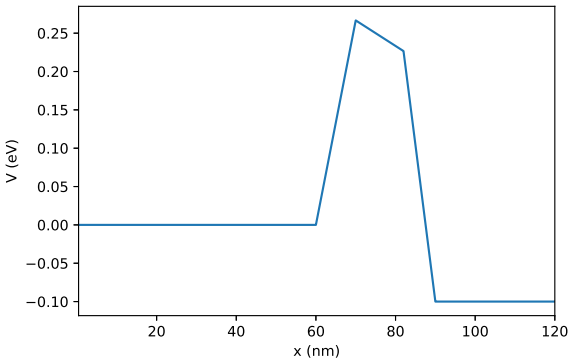
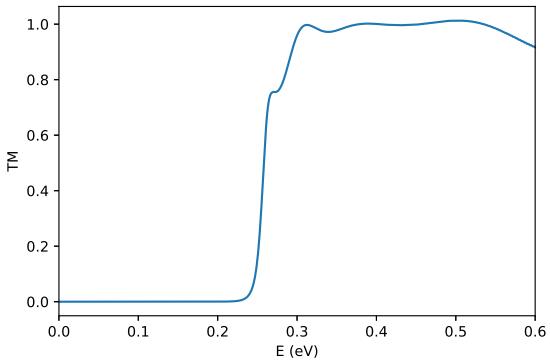


Figura 5.15: Funcionamiento de un transistor. Adaptado de Datta (2005).

Para simular el V_G se crea una barrera de potencial, o un pozo de potencial (Janik y Majkusiak, 1998); y para simular el V_D se crea un decaimiento de potencial o rampa ya que representa la brecha energética entre la fuente y el drenado. Para comparar con el transistor analizado por D. M. Sullivan y Wilson (2012), el material de la fuente y el drenado tiene una masa efectiva de 0.067, simulando GaAs, y el material del canal tiene una masa efectiva de 0.088, simulando $Al_{0,3}Ga_{0,7}As$, mientras que su longitud L será de 12 nm. Esta longitud es un buen ejemplo para analizar, puesto que ya existen transistores con canales que tienen al menos una longitud menor a los 10 nm (Lundstrom y Guo, 2006). La figura 5.16 muestra la representación más simple de estos potenciales en 1D, haciendo que el potencial V_G sea un potencial que cambie gradualmente mediante una rampa de potencial.



(a) Perfil de potencial de un transistor con V_G =0.3 eV y V_D =0.1 eV



(b) Transmisión a través de un transistor.

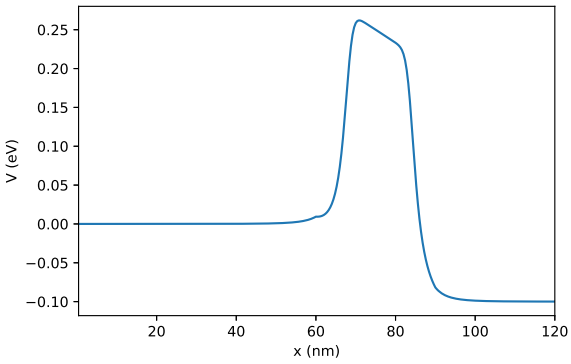
Figura 5.16: Representación simple de V_G en rampa y su TM.

Sin embargo, los potenciales reales en un transistor son más complejos que el mostrado, aunque para obtenerlos se requieren soluciones llamadas auto-consistentes, que son complejas y más costosas computacionalmente hablando, por lo que es común realizar ciertas aproximaciones (Janik y Majkusiak, 1998). En este trabajo se realizó un suavizamiento del potencial V_G aplicando la función de Butterworth, dada por la ecuación 5.17, que se utiliza en electrónica para atenuar una amplitud lo más plana posible entre el cero y la frecuencia de corte f_c, siendo n el orden el filtro (Pallás Areny, 2005).

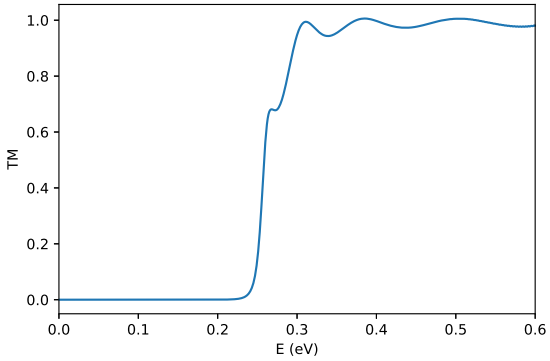
$$|H(f)|^2 = \frac{1}{1 + (f/f_c)^{2n}} \tag{5.17}$$

Haciendo un análogo del filtro con el potencial unidimensional, se encontró que la fórmula 5.18 proporciona buenos resultados visuales, comparando con los esperados en un transistor (Lundstrom y Guo, 2006). La fórmula 5.18 contiene una buena combinación de números para un canal de 12 nm y un campo eléctrico equivalente a 30 nm, donde V_c sería el centro de la barrera potencial. La figura 5.17 muestra la diferencia en el perfil de potencial y en la transmisión obtenida mediante el suavizamiento de V_G mediante el filtro de Butterworth.

$$V(i) = V_G \frac{1}{\sqrt{1 + (\frac{i - V_c}{20})^{10}}} \tag{5.18}$$



(a) Perfil de potencial con V_G =0.3 eV suavizado y V_D =0.1 eV



(b) Transmisión a través de un transistor

Figura 5.17: Representación con V_G suavizado y su TM.

Una vez obtenida la transmisión del potencial, se calculan las funciones de Fermi dadas por 5.19 y 5.20, donde E_F es la energía de Fermi, k_B es la constante de Boltzmann y T es la temperatura en °K, por lo que a 25°C, k_BT =0.0259 eV.

$$f_1(E) = \frac{1}{1 + e^{(E-\mu_1)/k_BT}} = \frac{1}{1 + e^{(E-E_F)/k_BT}} \tag{5.19}$$

$$f_2(E) = \frac{1}{1 + e^{(E-\mu_2)/k_BT}} = \frac{1}{1 + e^{(E-(E_F-eV_D))/k_BT}} \tag{5.20}$$

Para calcular la corriente del transistor, se aplica la ecuación 5.21, donde e es la carga del electrón en C.

$$I = \frac{e}{h} \int_0^\infty TM(E)(f_1(E) - f_2(E))dE \tag{5.21}$$

La figura 5.18 indica el cálculo de las funciones de Fermi y de la corriente con un potencial suavizado. Ya que la corriente I depende de las funciones de Fermi y éstas de la temperatura de trabajo, se puede comprobar numéricamente que a menor temperatura se obtendrán corrientes más altas, aunque tendrán mayor impacto en ella los potenciales V_G y V_D.

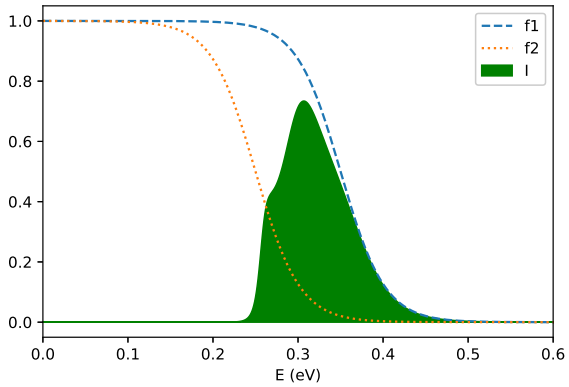


Figura 5.18: Funciones Fermi y cálculo de corriente en un transistor.

La conductancia G se calcula una vez obtenida la corriente I por medio de la siguiente fórmula (Datta, 2005).

$$G = \frac{I}{V_D} \tag{5.22}$$

Las mediciones de corriente y conductancia, obtenidas mediante las aproximaciones realizadas al potencial, se comparan en la tabla 5.3, con diferencias de menos del 2.3 %, logrando una mejor aproximación con la literatura cuando se utiliza el potencial suavizado. Por la forma del perfil de potencial utilizado por D. M. Sullivan y Wilson (2012), se cree que se usó un potencial autoconsistente (Nemnes, Ion, y Antohe, 2010; Saha, Sharma, Dabo, Datta, y Gupta, 2017).

Tabla 5.3: Cálculos de corriente (I) y conductancia (G) en una simulación 1D de un transistor.

Medición	Python V rampa	Python V suavizado	CUDA V rampa	CUDA V suavizado	Sullivan y Wilson (2012)
I (μ A)	2.87	2.79	2.87	2.78	2.81
G (μ S)	28.73	27.88	28.69	27.85	28.09

Se variaron además los valores de V_G y V_D para obtener gráficas de corriente-voltaje y de conductancia-voltaje (figuras 5.19 y 5.20), utilizando para ello el perfil de potencial suavizado. La gráfica 5.19 indica que hay un máximo de corriente que se puede alcanzar en el canal semiconductor analizado, en este caso 13.61μA, aumentando cuando V_D

aumenta (la diferencia de potencial entre la fuente y el drenado) y llegando a un máximo cuando V_G es pequeño. Este es un comportamiento esperado porque se desvanece la barrera de potencial y se llega al caso balístico de V_G =0 (D. M. Sullivan, 2012). Esto se observa también de forma experimental, aunque ahí se considera que V_G es el potencial aplicado para que empiecen a fluir los electrones, por lo que, a mayores V_G se obtiene mejor corriente hasta llegar a un máximo (Janik y Majkusiak, 1998); incluso hay que vencer un V_T (Voltage Threshold) para que se empiece a obtener tanto corriente como conductancia (Datta, 2005). Este voltaje umbral V_T está relacionado con el ancho del potencial, es decir, del canal semiconductor, y se sabe que su valor disminuye cuando se reduce la longitud del semiconductor (Sahay y Kumar, 2019), por lo que por efecto de tunelaje no aparecerá en todas las curvas de la gráfica 5.20.

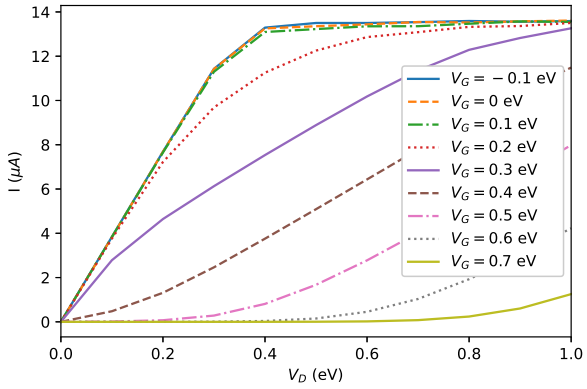


Figura 5.19: Curvas de corriente contra voltaje de drenado.

La gráfica 5.20 indica que también existe un máximo en cuanto a conductancia, lograda con barreras de potencial pequeñas pero no necesariamente con V_D grandes, como en el caso del valor máximo de la corriente I, lo que ya se ha observado en la literatura (Khemissi, 2012; Prakash, Prasad, y Jain, 2010). Se observa que se puede llegar cerca del máximo con V_D menores a 0.3 eV, siendo el mayor número 38.42μS, obtenido con V_D =0.1 eV y V_G =-0.1 eV, muy cerca del límite cuántico llamado conductancia cuántica G_0, cuyo valor teórico es $G_0 = \frac{e^2}{h}$ =38.7 μS), donde e es la carga del electrón (Datta, 2005; D. M. Sullivan, 2012).

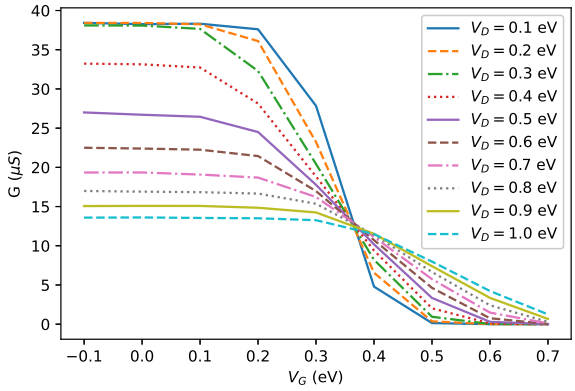


Figura 5.20: Curvas conductancia contra voltaje de compuerta.

También es interesante notar que, a V_D bajos, la conductancia tiene un aspecto escalonado parecido al encontrado experimentalmente por van Wees et al. (1988), mostrando un comportamiento cuantizado de la conductancia al mostrarse múltiples platos al variar el voltaje de compuerta.

5.2. Extensión a sistemas bidimensionales

Una vez obtenido y paralelizado el código cuántico en una dimensión, la extensión del algoritmo a dos dimensiones es más sencilla. Las variables vectoriales ahora deberán tener coordenadas (x,y) para estudiar sistemas con geometrías más complicadas. Lo más viable es obtener un sistema de estudio cuadrado, o al menos con el mismo tamaño de PML en ambos ejes, proporcionando cierta simetría en la absorción de las ondas, aunque se puede tratar de manera separada (Zheng, 2007). Aplicando la ecuación 4.21 a un sistema bidimensional, la ecuación de Schrödinger con PML es la siguiente.

$$\frac{\partial \psi}{\partial t} = i\frac{\hbar}{2m_e}\left(\gamma(x,y)\frac{\partial^2\psi(x,y,t)}{\partial(x,y)^2}\right) - \frac{i}{\hbar}V(x,y)\psi(x,y,t) \tag{5.23}$$

Separando las partes reales e imaginarias tanto de la función de onda ψ como de γ, la función de onda está definida por las siguientes expresiones.

$$\frac{\partial \psi_{real}}{\partial t} = \frac{V(x,y)}{\hbar}\psi_{imag}(x,y,t) - \frac{\hbar}{2m_e}\left(\gamma_{imag}(x,y)\frac{\partial^2\psi_{real}(x,y,t)}{\partial(x,y)^2} + \gamma_{real}(x,y)\frac{\partial^2\psi_{imag}(x,y,t)}{\partial(x,y)^2}\right) \tag{5.24a}$$

$$\frac{\partial \psi_{imag}}{\partial t} = -\frac{V(x,y)}{\hbar}\psi_{real}(x,y,t) + \frac{\hbar}{2m_e}\left(\gamma_{real}(x,y)\frac{\partial^2\psi_{real}(x,y,t)}{\partial(x,y)^2} - \gamma_{imag}(x,y)\frac{\partial^2\psi_{imag}(x,y,t)}{\partial(x,y)^2}\right) \tag{5.24b}$$

Puesto que

$$\frac{\partial^2\psi(x,y,t)}{\partial(x,y)^2} = \frac{\partial^2\psi(x,y,t)}{\partial x^2} + \frac{\partial^2\psi(x,y,t)}{\partial y^2} \tag{5.25}$$

y a su vez

$$\frac{\partial^2\psi(x,y,t)}{\partial x^2} \approx \frac{\psi(i+1,j) - 2\psi(i,j) + \psi(i-1,j)}{\Delta x^2} = \frac{\Delta_{(i)}}{\Delta x^2} \tag{5.26}$$

entonces las ecuaciones 5.24a y 5.24b se pueden discretizar como sigue

$$\psi^{m+1}_{real}(i,j) = \psi^{m}_{real}(i,j) + \frac{V(i,j)\Delta t}{\hbar}\psi^{m+1/2}_{imag}(i,j) - \frac{\hbar\Delta t}{2m_e\Delta x^2\Delta y^2}\left[\gamma_{imag}(i,j)(\Delta y^2\Delta_{real(i)} + \Delta x^2\Delta_{real(j)}) + \gamma_{real}(i,j)(\Delta y^2\Delta_{imag(i)} + \Delta x^2\Delta_{imag(j)})\right] \tag{5.27a}$$

$$\psi_{imag}^{m+3/2}(i,j) = \psi_{imag}^{m+1/2}(i,j) - \frac{V(i,j)\Delta t}{\hbar}\psi_{real}^{m+1}(i,j) + \frac{\hbar\Delta t}{2m_e\Delta x^2\Delta y^2}\left[\gamma_{real}(i,j)(\Delta y^2\Delta_{real(i)} + \Delta x^2\Delta_{real(j)}) - \gamma_{imag}(i,j)(\Delta y^2\Delta_{imag(i)} + \Delta x^2\Delta_{imag(j)})\right] \tag{5.27b}$$

Además, cuando $\Delta x = \Delta y$, las fórmulas discretizadas para calcular las siguientes iteraciones se reducen a

$$\psi_{real}^{m+1}(i,j) = \psi_{real}^{m}(i,j) + \frac{V(i,j)\Delta t}{\hbar}\psi_{imag}^{m+1/2}(i,j) - \frac{\hbar\Delta t}{2m_e\Delta x^2}\left[\gamma_{imag}(i,j)(\Delta_{real(i)} + \Delta_{real(j)}) + \gamma_{real}(i,j)(\Delta_{imag(i)} + \Delta_{imag(j)})\right] \tag{5.28a}$$

$$\psi_{imag}^{m+3/2}(i,j) = \psi_{imag}^{m+1/2}(i,j) - \frac{V(i,j)\Delta t}{\hbar}\psi_{real}^{m+1}(i,j) + \frac{\hbar\Delta t}{2m_e\Delta x^2}\left[\gamma_{real}(i,j)(\Delta_{real(i)} + \Delta_{real(j)}) - \gamma_{imag}(i,j)(\Delta_{imag(i)} + \Delta_{imag(j)})\right] \tag{5.28b}$$

Las ecuaciones 5.28a y 5.28b tienen similitudes con los desarrollos de las ecuaciones en dos y tres dimensiones, obtenidas sin la inclusión del parámetro γ que implementa las condiciones de frontera (D. M. Sullivan, Mossman, y Kuzyk, 2016), siendo iguales cuando los valores de γ corresponden a los del dominio de simulación, es decir, la región donde la función de onda no interactúa con la PML. Esta concordancia es un indicio positivo de que el escalamiento a tres dimensiones del algoritmo también es relativamente sencillo de implementar y de paralelizar, haciendo posible el estudio de sistemas más complejos sin un aumento tan drástico en los recursos computacionales.

Cálculo de la función de absorción

Para calcular los valores complejos del stretching coordinate, es decir los valores correspondientes a las ecuaciones 5.8a y 5.8b en forma bidimensional, se debe calcular el valor de la función de absorción $\sigma(x,y)$, ahora dado por la ecuación 5.29, donde n es

la coordenada discreta i o j, dependiendo del valor más cercano de n_{pml}.

$$\sigma(i,j) = \sigma_0(n - n_{pml})^2 \tag{5.29}$$

En el sistema bidimensional, existirán cuatro n_{pml}, uno superior U_{pml}, uno inferior D_{pml}, uno en la izquierda L_{pml} y otro en la derecha R_{pml}. El índice n a utilizar dependerá de la zona en que se encuentre la coordenada (i,j), expuesto en la figura 5.21. El tamaño de las PML será el mismo en las cuatro regiones para utilizar un mismo valor de σ_0.

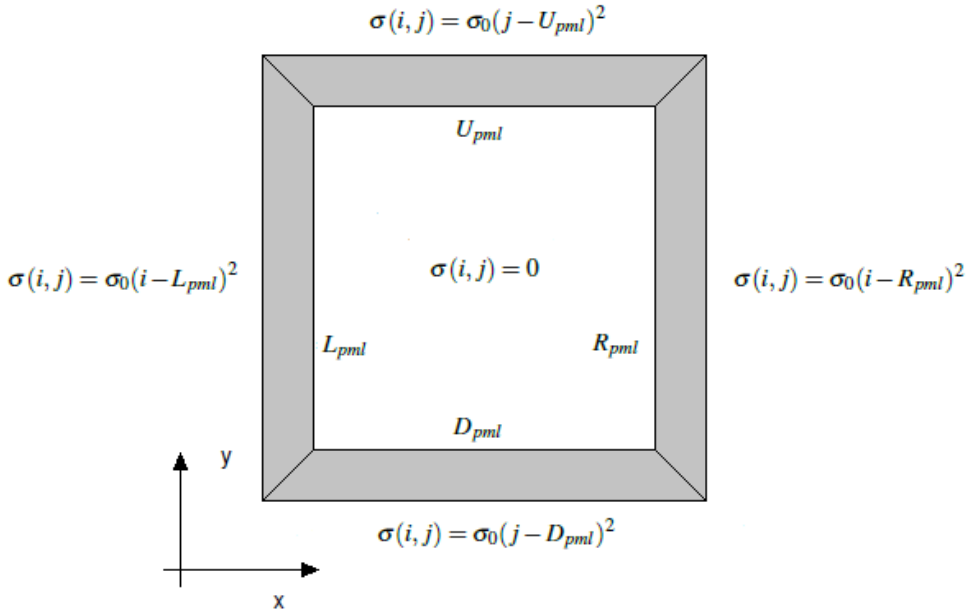
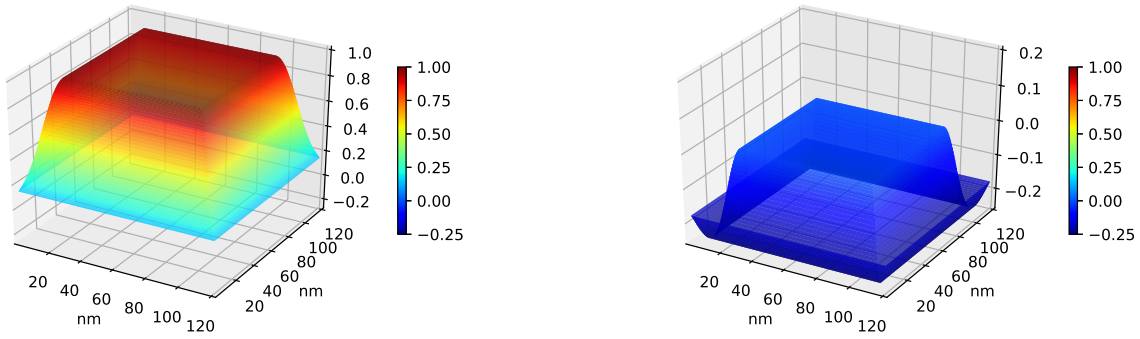


Figura 5.21: Elección del valor de $\sigma(i,j)$. El área sombreada es donde actúa la PML.

Partiendo del análisis de la sección 5.1.1 para elegir un buen valor de σ_0 con una PML de 20 nm, la figura 5.22 muestra los valores que tomaría γ en un espacio de simulación de 120X120 nm, con un σ_0 =0.0005 y un Δx =0.4 nm.



(a) γ_{real} (b) γ_{imag}

Figura 5.22: Forma de γ en dos dimensiones con una PML de 20 nm.

5.2.1. Obtención de los observables

Para el cálculo de la probabilidad en dos dimensiones, la ecuación 5.12 se modifica a

$$|\psi|^2 = \sum_{j=1}^{N_y} \sum_{i=1}^{N_x} [\psi_{real}^2(i,j) + \psi_{imag}^2(i,j)] \tag{5.30}$$

Por su parte, para calcular el valor esperado de la energía cinética KE, se aplica el operador $\hat{T}$ a la función de onda de la siguiente manera.

$$\langle KE \rangle = \int \psi^* \hat{T} \psi = \int (\psi_{real} - i\psi_{imag}) \left(-\frac{\hbar^2}{2m_e} \right) \nabla^2 (\psi_{real} + i\psi_{imag}) \tag{5.31}$$

Desarrollando el rotacional para el sistema en dos dimensiones:

$$\langle KE \rangle = \int -\frac{\hbar^2}{2m_e} (\psi_{real} - i\psi_{imag}) \left[\frac{\partial^2 \psi_{real}}{\partial x^2} + \frac{i\partial^2 \psi_{imag}}{\partial x^2} + \frac{\partial^2 \psi_{real}}{\partial y^2} + \frac{i\partial^2 \psi_{imag}}{\partial y^2} \right] \tag{5.32}$$

Ya que se sabe que KE es un valor real, se extrae la parte real de $\langle KE \rangle$, por lo que la ecuación 5.32 se reduce a

$$\langle KE \rangle = -\frac{\hbar^2}{2m_e} \int \psi_{real} \frac{\partial^2 \psi_{real}}{\partial x^2} + \psi_{real} \frac{\partial^2 \psi_{real}}{\partial y^2} + \psi_{imag} \frac{\partial^2 \psi_{imag}}{\partial x^2} + \psi_{imag} \frac{\partial^2 \psi_{imag}}{\partial y^2} \tag{5.33}$$

y al discretizar la ecuación

$$\langle KE\rangle = -\frac{\hbar^2}{2m_e\Delta x^2\Delta y^2}$$
$$\sum_{j=1}^{N_y}\sum_{i=1}^{N_x}\psi_{real}(i,j)(\Delta y^2\Delta_{real(i)} + \Delta x^2\Delta_{real(j)}) + \psi_{imag}(i,j)(\Delta y^2\Delta_{imag(i)} + \Delta x^2\Delta_{imag(j)}) \tag{5.34}$$

La ecuación anterior se puede simplificar si se considera que $\Delta x = \Delta y$, como es el caso de los sistemas a estudiar, por lo que se utilizará la siguiente fórmula para calcular la energía cinética en el espacio.

$$\langle KE\rangle = -\frac{\hbar^2}{2m_e\Delta x^2}\sum_{j=1}^{N_y}\sum_{i=1}^{N_x}\psi_{real}(i,j)(\Delta_{real(i)} + \Delta_{real(j)}) + \psi_{imag}(i,j)(\Delta_{imag(i)} + \Delta_{imag(j)}) \tag{5.35}$$

En cuanto a la energía potencial, aplicando la ecuación 4.10 a dos dimensiones, la fórmula discretizada es la siguiente.

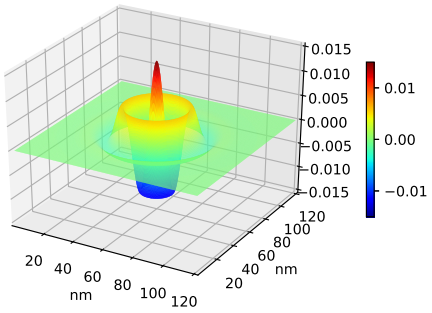
$$\langle PE\rangle = \sum_{i=1}^{N_x}\sum_{j=1}^{N_y}[\psi^2_{real}(i,j) + \psi^2_{imag}(i,j)]V(i,j) \tag{5.36}$$

5.2.2. Paralelización del sistema bidimensional

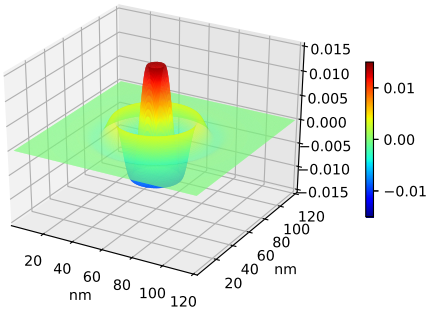
Para estudiar la aceleración del ciclo temporal en un sistema bidimensional, se inicializó un pulso gaussiano en el centro de un sistema de 120 nm X 120 nm, con una energía inicial de 0.0525 eV (ver figura 5.23). Al igual que en la paralelización unidimensional, se diseñaron 4 kernels que se mandan llamar dentro del ciclo temporal del algoritmo (ver apéndice II.3).

```
int paso;
for (int m=1;m<=n_step;m++){
paso=m;
prl_nueva<<<grid,block>>>(d_prl, d_pim, d_prl_n, Nx, dt, d_V, d_ra,
                          d_gamr, d_gami, Ny);
prl_reescribir<<<grid,block>>>(d_prl, d_prl_n, Nx, Ny);
```

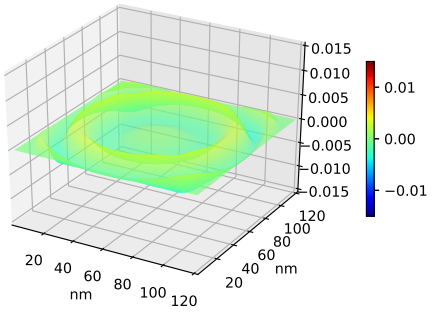
```
pim_nueva<<<grid,block>>>(d_prl, d_pim, d_pim_n, Nx, dt, d_V, d_ra,
                          d_gamr, d_gami, Ny);
pim_reescribir<<<grid,block>>>(d_pim, d_pim_n, Nx, Ny);
}
```



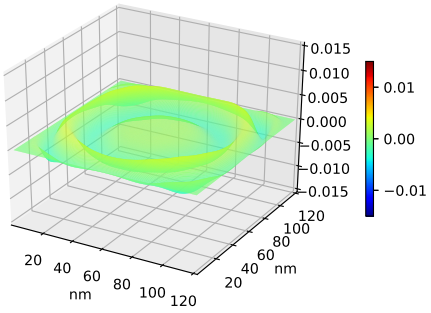
(a) ψ_{real} a t =0 fs



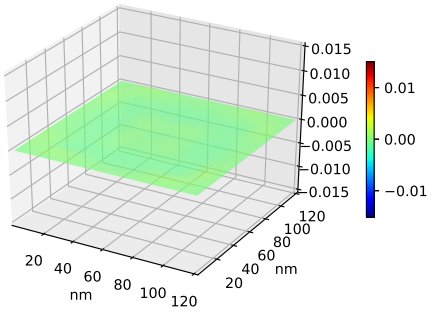
(b) ψ_{imag} a t =0 fs



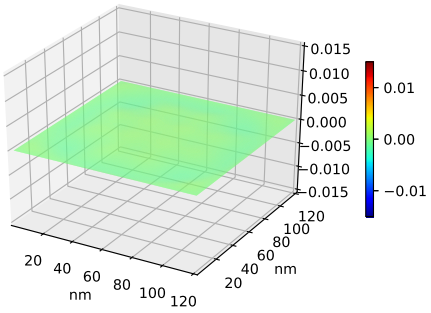
(c) ψ_{real} a t =100 fs



(d) ψ_{imag} a t =100 fs



(e) ψ_{real} a t =250 fs



(f) ψ_{imag} a t =250 fs

Figura 5.23: Absorción de un pulso con KE_0 =0.0525 eV y una PML de 20 nm.

Como se indica en la tabla 5.4, los tiempos de cómputo disminuyen conforme el

Tabla 5.4: Comparación de los tiempos de cómputo del ciclo temporal en diferentes entornos de programación

Tiempo simulación (fs)	Python (s)	C++ (s)	CUDA Geforce GTX 1050 (s)	CUDA Tesla K20c (s)	CUDA Tesla K80 (s)
0	0.00	0.00	0.00	0.00	0.00
100	5969.97	39.04	0.59	0.51	0.56
250	13878.92	96.81	1.41	1.27	1.30
3000	171325.49	1167.41	16.08	15.46	13.86

entorno de CUDA trabaja con mayor cantidad de núcleos en la tarjeta gráfica. Aunque existe una gran diferencia entre el tiempo de Python y el de C++, sobretodo con tiempos de simulación grandes, esa diferencia es de esperarse ya que Python es un lenguaje interpretado, mientras que C++ es un lenguaje compilado, lo que hace que las ejecuciones de éste último sean más rápidas. Por tal motivo, las comparaciones de incremento en rapidez de cálculo se analizan en función del lenguaje C++ y las diferentes tarjetas que utiliza CUDA para paralelizar al algoritmo.

La gráfica 5.24 muestra que con un gran número de pasos temporales en las simulaciones, el aceleramiento del algoritmo se relaciona con el número de núcleos de la tarjeta gráfica utilizada, siendo hasta 84 veces más rápida que la versión serial. Sin embargo, el sólo hecho de tener una paralelización con una tarjeta de menos núcleos aumenta la rapidez hasta 72 veces, por lo que no hay una gran ganancia de tiempo entre una GPU dedicada exclusivamente al cómputo científico, contra otra de uso comercial y más económica. También es importante notar que algunas veces la rapidez no sólo tiene que ver con el número de CUDA cores disponibles, sino con otros componentes propios de la tarjeta, aunque estas diferencias no son muy perceptibles. A pesar de que las comparaciones fueron realizadas en un sistema relativamente pequeño (300x300 celdas), esta comparación es un claro indicativo de la gran reducción que podría tener en el cómputo de sistemas más grandes y complejos.

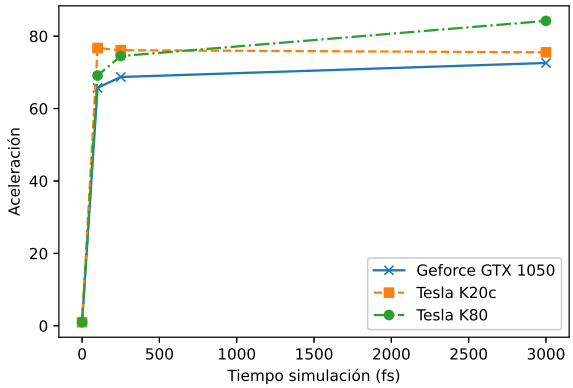


Figura 5.24: Aceleración del tiempo de cómputo del ciclo temporal de las tarjetas en relación a la versión serial en C++.

En cuanto a la absorción del pulso gaussiano de la figura 5.23, la tabla 5.5 indica la disminución de probabilidad conforme la función de onda interactúa con la PML cuadrada. Las diferencias entre los lenguajes se juzgan existentes por la precisión utilizada en cada uno. Aunque pareciera que la absorción de ψ no es suficientemente rápida y que podría deberse a un mal diseño de la PML, puesto que el pulso inicial tiene una longitud de onda equivalente a 53 Δx y la PML un tamaño de 50 Δx, un análisis del sistema verifica que longitudes de onda más pequeñas obtienen absorciones similares (tabla 5.6). La reducción en tiempo de cómputo, obtenida en el sistema paralelizado, es claramente necesaria para detectar rápidamente errores en el diseño de cualquier sistema cuántico, ya que cada corrida se obtiene en segundos, agilizando la detección de errores u optimizaciones de condiciones de frontera absorbentes.

Tabla 5.5: Comparación de $|\psi|^2$ del pulso gaussiano con KE_0 =0.0525.

Tiempo (fs)	Python	CUDA/C++
0	1.000000	1.000022
100	0.322697	0.322717
250	0.028343	0.028345
3000	0.002838	0.002838

Tabla 5.6: Comparación de la absorción de $|\psi|^2$ con diferentes KE_0.

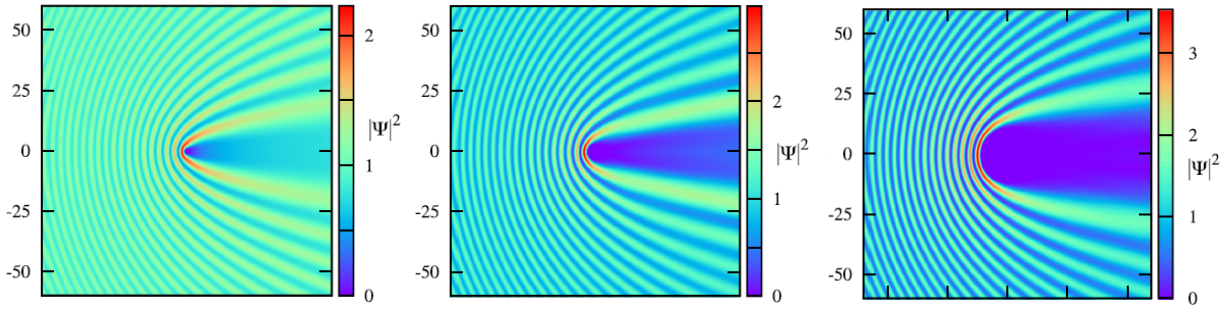
Tiempo (fs)	KE_0 =0.0525 λ =53 Δx	KE_0 =0.0545 λ =52 Δx	KE_0 =0.0567 λ =51 Δx	KE_0 =0590 λ =50 Δx	KE_0 =0.0614 λ =49 Δx
0	1.000022	1.000022	0.999998	0.999999	1.000000
100	0.322717	0.310138	0.297492	0.284817	0.272130
250	0.028345	0.026926	0.025552	0.024223	0.022939
3000	0.002838	0.002704	0.002570	0.002439	0.002317

5.2.3. Casos de estudio: Potencial circular

Un caso que puede resolverse de forma analítica en dos dimensiones es el scattering que se produce por la interacción de un potencial infinito con forma de círculo. Este fenómeno se suele estudiar desde un punto de vista electromagnético (Bohren y Huffman, 1998), por lo que es importante poder extraer información a través de simulaciones con enfoque cuántico y sin restriciones geométricas.

El scattering para potenciales circulares infinitos ya ha sido resuelto de forma analítica por McAlinden y Shertzer (2016) para el caso cuántico. Para ello utilizan la ecuación de Schrödinger en coordenadas cilíndricas (ρ,ϕ) en su forma independiente del tiempo. Se considera que la función de onda ψ tiene dos contribuyentes, uno de la onda plana incidente con forma $e^{ik\rho cos(\phi)}$ y otro por el scattering de forma $f_k(\phi)\frac{e^{ik\rho}}{\sqrt{\rho}}e^{i\pi/4}$, siendo $f_k(\phi)$ la amplitud compleja del scattering; de manera que $\psi = \psi_{inc} + \psi_{scat}$.

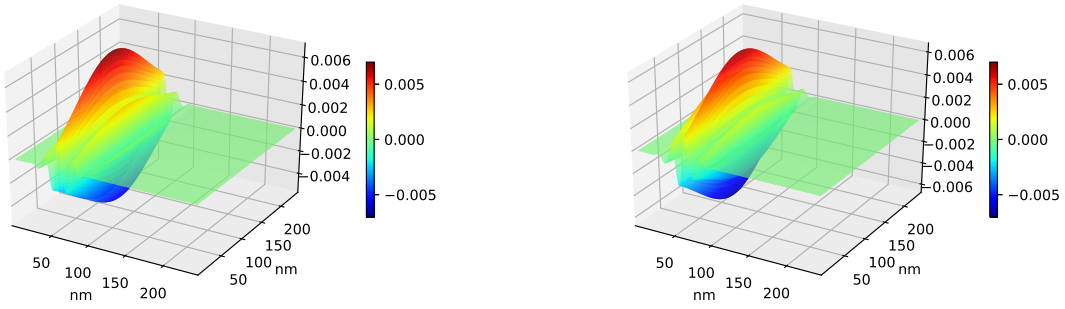
Para encontrar las soluciones se separa la zona de estudio en tres regiones: la primera, donde existe el potencial infinito y por lo tanto $\psi = 0$; la segunda, donde existen soluciones de acuerdo a funciones Hankel; y la tercera, donde la coordenada radial es suficientemente grande y ψ tiene soluciones con funciones Bessel J. En la figura 5.25 se puede observar la densidad de probabilidad $|\psi|^2$ de la solución estacionaria para tres diferentes tamaños de potencial, donde se considera la coordenada adimensional $\xi = k\rho$ y el radio de potencial adimensional $\alpha = rk = 2\pi r/\lambda$.



(a) $\alpha = \pi/5$ $(r = \lambda/10)$ (b) $\alpha = \pi$ $(r = \lambda/2)$ (c) $\alpha = 7\pi/2$ $(r = 7\lambda/4)$

Figura 5.25: Densidad de probabilidad para potenciales circulares infinitos con ejes adimensionales. Tomado de McAlinden y Shertzer (2016).

Para realizar la simulación con el algoritmo FDTD cuántico, se describió una onda de perfil plano para observar su interacción con un potencial circular (ver figura 5.26), que representa la interacción con un potencial cilíndrico en su forma transversal.



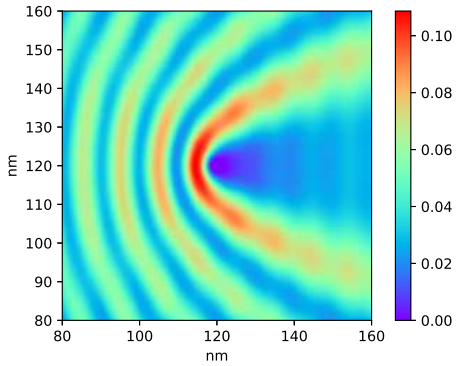
(a) ψ_{real} (b) ψ_{imag}

Figura 5.26: Inicialización de un pulso con perfil plano con una energía de 0.06 eV.

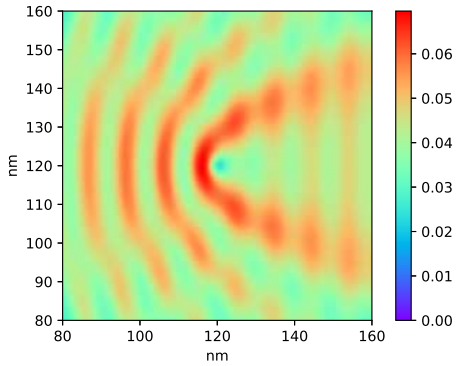
Para estas simulaciones sólo se utilizó la versión paralelizada, ya que se agrandó el espacio de simulación a 240 nm X 240 nm, requiriendo más recursos de cómputo, aunque todas tardan alrededor de dos minutos en CUDA. El código utilizado para estas simulaciones se encuentra en el apéndice II.4, que además muestra la implementación para el acceso a la Memoria Unificada en CUDA (Unified Memory), lo que reduce el tiempo de comunicación entre el Host y el Device al compartir los accesos de memoria (Storti y Yurtoglu, 2015).

La figura 5.27 muestra la probabilidad en el espacio una vez que se realiza la transformada de Fourier a la energía de 0.06 eV, indicando un comportamiento similar a

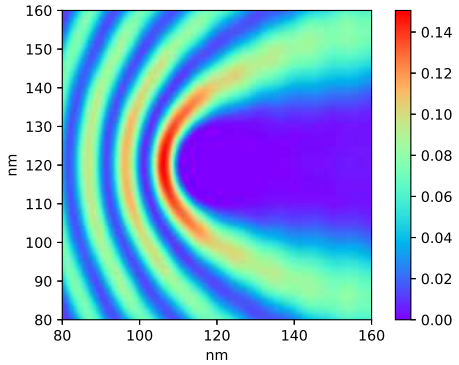
los calculados analíticamente. Se muestran además las comparaciones del scattering con un potencial de la misma forma pero con un valor de 0.1 eV, que puede emplearse para estudiar la penetración que puede existir en los potenciales circulares y las diferencias existentes por idealizar el valor del potencial a infinito.



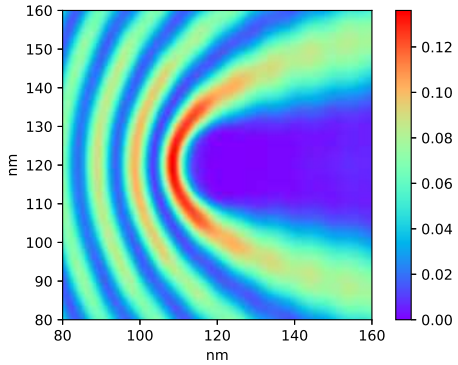
(a) Potencial infinito con radio de 1.4 nm



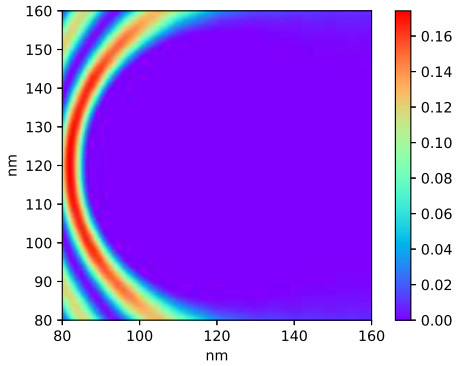
(b) Potencial de 0.1 eV con radio de 1.4 nm



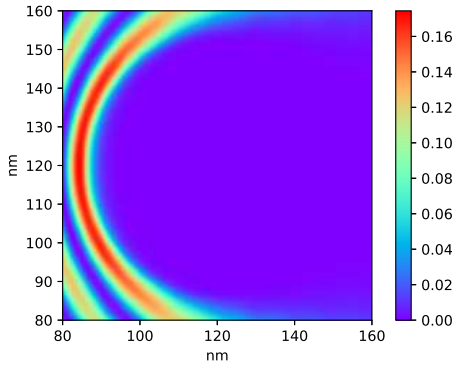
(c) Potencial infinito con radio de 9.6 nm



(d) Potencial de 0.1 eV con radio de 9.6 nm



(e) Potencial infinito con radio de 33.6 nm



(f) Potencial de 0.1 eV con radio de 33.6 nm

Figura 5.27: Scattering de una onda plana de 0.06 eV con potenciales circulares.

Las simulaciones de potenciales cilíndricos en dos dimensiones son útiles cuando

se quieren estudiar fenómenos de nanoalambres y su interacción con sus modos transversales. Estas simulaciones son posibles de mejorar al disminuir la discretización espacial o realizar otros ajustes, como el hacer promedios de los valores de potencial en celdas vecinas al perímetro (D. M. Sullivan et al., 2016). Este algoritmo tiene la versatilidad de modelarse con sistemas tan reales como sea posible programarse, lo que le confiere un gran potencial para simulaciones futuras en varios campos.

Se observó que al utilizar arreglos temporales y espaciales grandes, o al realizar transformadas de Fourier con muchas frecuencias a analizar, la memoria RAM incrementa durante la ejecución del código, llegando a interrumpirla. Este inconveniente, aunque se reduce de forma considerable al utilizar Unified Memory, no se presenta para ninguno de los códigos utilizados en este trabajo, aunque es evidente que es una problemática a resolver para estudios donde se requieran variables con arreglos muy grandes.

Capítulo 6

Conclusiones y recomendaciones

Es evidente que la paralelización de un algoritmo que emplea condiciones de frontera reflejantes es más simple que al utilizar condiciones absorbentes. En este último caso, las PML utilizadas requieren 2 kernels adicionales para evitar la sobreescritura de variables en un momento que no se desea. La implementación de condiciones periódicas sería un trabajo a paralelizar en futuros estudios.

Aunque la paralelización del caso unidimensional no arroja una gran aceleración en cuanto a la forma serial de C++, las ganancias son muy relevantes al extender el código al sistema bidimensional, con el que se pueden estudiar sistemas más complejos y con una mayor capacidad de cálculo. Esta paralelización también es la base para que se pueda extender el algoritmo, y por tanto su paralelización, a un sistema tridimensional, que además haría más notable y necesaria la utilización de tarjetas gráficas.

La plataforma CUDA cada vez tiene más herramientas para optimizar paralelizaciones, pero el mismo avance evita que todas las GPU's soporten dichas optimizaciones. Sin embargo, se pueden escribir códigos de manera general para que sea posible correrse en cualquier device, con una reducción de tiempo bastante significativa con cualquier GPU utilizada.

A pesar de que Python no posee la misma rapidez que un lenguaje como C++, el lenguaje sirve como guía para el desarrollo de códigos de forma sencilla. La plataforma de Google Colab también es una alternativa para desarrollar código tanto en Python como con CUDA C++, proporcionando tarjetas gráficas de manera gratuita, lo que facilita el alcance a los algoritmos desarrollados en esta tesis.

Apéndice I

Códigos unidimensionales

Los siguientes códigos se utilizaron para obtener las figuras y cálculos de la programación en una dimensión de la sección 5.1. El primer código está escrito en lenguaje Python de forma serial, el segundo código está escrito en forma serial para C++ y finalmente el tercer código está escrito en forma paralelizada para CUDA C++. Las figuras son realizadas con la herramienta de Matplotlib de Python especificada en el código, aunque son coherentes a las impresiones de datos de los códigos en C++ y CUDA.

I.1. Programa en forma serial en Python

Algunas operaciones se parten para que sean legibles en el ancho de la página, aunque esto se debe evitar porque el lenguaje lo interpreta como una nueva instrucción.

```
#Programa para simular la propagaci'on de una part'icula en un sistema 1D
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline

#Definici'on de unidades fijas
L=120e-9 #Longitud de la malla en metros
del_x=.4e-9  #Tama'no de la celda en metros
Nx=round(L/del_x)  #N'umero de puntos en la partici'on
```

```
hbar=1.054E-34
m0=9.1E-31  #Masa del electr'on en un espacio libre
meff=0.067  #Masa efectiva del GaAs
meffpozo=0.088 #Masa efectiva del Al.3Ga.7As
melec=m0*meff  #Masa del electr'on
ecoul=1.6e-19  #Carga del electr'on en Coulomb
eV2J=1.6e-19  #Factor de conversi'on de energ'ia en eV a Joules
J2eV=1/eV2J #Factor de conversi'on de energ'ia de Joules a eV
dt=2*melec*del_x**2/(8*hbar)    #Tama'no del paso temporal
ra=np.zeros(Nx+1) #ra es el criterio de estabilidad.
                  #Cambia si se cambia el material
V=np.zeros(Nx+1) #Potencial inicializado en 0
Dx=del_x*1e9  #Tama'no de la celda en nm

#Definir ra para monitoreo de entrada.
#Cambia si se cambia el material
ra.fill((.5*hbar/melec)*(dt/del_x**2))

#C'alculo para definir longitud de onda del pulso inicial
#La energ'ia inicial es 0.204 eV
lamb=4.135e-15*eV2J/np.sqrt(0.204*eV2J*2*melec)
#Cambio de longitud de onda de metros a unidades de partici'on
lambd=round(lamb/del_x)
sigma=lambd
nc=Nx*40/120 #Posici'on inicial del pulso en 40 nm
prl=np.zeros(Nx+1) #Parte real del estado variable
pim=np.zeros(Nx+1) #Parte imaginaria del estado variable

#Ciclo para inicializar pulso gaussiano
ptot=0
for k in range(1,Nx+1):
```

```
    prl[k]=np.exp (-1*((k-nc)/sigma)**2)*np.cos(2*np.pi*(k-nc)/lambd)
    pim[k]=np.exp (-1*((k-nc)/sigma)**2)*np.sin(2*np.pi*(k-nc)/lambd)
    ptot+=prl[k]**2+pim[k]**2

pnorm=np.sqrt(ptot) #Constante para normalizar

#Ciclo para normalizar el pulso gaussiano
ptot=0
for k in range(1,Nx+1): #Va de 1 a Nx
    prl[k]=prl[k]/pnorm
    pim[k]=pim[k]/pnorm
    ptot+=prl[k]**2+pim[k]**2

#Comprobar que la integral en el espacio es 1
print("ptot={}" .format(ptot))

#Definir par'ametros de la PML
npml=50 #N'umero de particiones de la pml, equivale a 20 nm
sigma0=0.0005
gamr=np.zeros(Nx+1)
gami=np.zeros(Nx+1)
R=(1.0+1.0j)/np.sqrt(2.0) #El j indica que R es un n'umero complejo
for n in range(1,Nx+1): #Va de 1 a Nx
    if n>Nx-npml: #Zona de Rpml a Nx
      sigmapml=sigma0*(n-(Nx-npml))**2
      gamma=(1.0/(1.0+R*sigmapml))**2
      gammar=gamma.real
      gammai=gamma.imag
    elif n<=npml: #Zona de 1 a Lpml
      sigmapml=sigma0*(n-1-npml)**2
      gamma=(1.0/(1.0+R*sigmapml))**2
```

```
      gammar=gamma.real
      gammai=gamma.imag
    else: #Zona de Lpml a Rpml
      gammar=1.0
      gammai=0.0
    #Terminaci'on del if y definici'on de valores de gamma
    gamr[n]=gammar
    gami[n]=gammai

#Graficar valores de gamma complejo
gam_xgraf=np.arange(0,300,1)
gamr_ygraf=gamr[gam_xgraf]
gami_ygraf=gami[gam_xgraf]
plt.plot(gam_xgraf*Dx+Dx,gamr_ygraf,label ='$\gamma_{real}$')
plt.plot(gam_xgraf*Dx+Dx,gami_ygraf,
        label ='$\gamma_{imag}$',linestyle='dashed')
plt.legend()
plt.xlim(1, 120)
plt.xlabel('nm')
plt.savefig('gamma1D.eps', format='eps')

#Definir variables para monitoreo
picoseg=3 #tiempo en ps de la simulaci'on
n_step=int(picoseg/(dt*1e12)) #Pasos temporales
TD_in=95 #nm donde se realiza el monitoreo de entrada
posicion_in=int(TD_in/(del_x*1e9)) #Celda para monitoreo de entrada
posicion_out=posicion_in #Celda para monitoreo salida
#Arreglos para los monitoreos reales e imaginarios de entrada y salida
#Van de 1 a n_step
prl_in=np.zeros(n_step)
prl_out=np.zeros(n_step)
```

```
pim_in=np.zeros(n_step)
pim_out=np.zeros(n_step)
#Definir arreglos para mantener valores de psi en el paso temporal
prl_n=np.zeros(Nx+1)
pim_n=np.zeros(Nx+1)

#Ciclo temporal de entrada (sin cambios de masa efectiva ni potenciales)
for m in range(0,n_step): #Pasos 1 a n_step
#Ciclo para calcular la parte real de psi
  for n in range(2,Nx): #Va de 2 a Nx-1
    deltar=prl[n-1]-2*prl[n]+prl[n+1]
    deltai=pim[n-1]-2*pim[n]+pim[n+1]
    prl_n[n]=prl[n]-ra[n]*(gamr[n]*deltai+gami[n]*deltar)
        +(dt/hbar)*V[n]*pim[n]
#Ciclo para actualizar la parte real de psi
  for n in range(2,Nx):
    prl[n]=prl_n[n]
  #Ciclo para calcular la parte imaginaria de psi
  for n in range(2,Nx):  #Va de 2 a Nx-1
    deltar=prl[n-1]-2*prl[n]+prl[n+1]
    deltai=pim[n-1]-2*pim[n]+pim[n+1]
    pim_n[n]=pim[n]+ra[n]*(gamr[n]*deltar-gami[n]*deltai)
        -(dt/hbar)*V[n]*prl[n]
#Ciclo para actualizar la parte imaginaria de psi
  for n in range(2,Nx):
    pim[n]=pim_n[n]
  #Guardar datos para el monitoreo
  prl_in[m]=prl[posicion_in]
  pim_in[m]=pim[posicion_in]

#Graficar monitoreo de entrada
```

```
xgraf=np.arange(0,n_step,1)
yreal=prl_in[xgraf]
yimag=pim_in[xgraf]
plt.plot(xgraf*dt*1e12,yreal,label ='$\psi_{real}$')
plt.plot(xgraf*dt*1e12,yimag,'--',label='$\psi_{imag}$')
plt.ylabel('')
plt.xlabel('ps')
plt.xlim(0,(n_step*dt*1e12))
plt.legend(loc=1)
plt.savefig('psi_in.eps', format='eps')

#C'alculo de la energ'ia de entrada con FFT
#S'olo se toma la parte real de la funci'on de onda
from scipy.fftpack import fft
data_in=prl_in
fft_in = fft(data_in)
lenx=fft_in.shape[0]
magnitud_in = [np.sqrt(i.real**2 + i.imag**2)/len(fft_in) for i in fft_in]

#Definir forma del potencial
iniciopotencial=int(70e-9/del_x) #Inicio de la barrera de potencial en 70 nm
iniciocampo=round(60e-9/del_x) #Inicio del campo el'ectrico
fincampo=round(90e-9/del_x)
VG=0.3 #Tama'no del potencial en eV
VD=0.1 #Decaimiento del potencial o potencial de drenado en eV o V/C

#Descomentar si es una barrera simple de 12 nm
for k in range(iniciopotencial,iniciopotencial+int(12e-9/del_x)+1):
    V[k]=VG*eV2J
    ra[k]=(.5*hbar/(m0*meffpozo))*(dt/del_x**2)
```

```
#Descomentar si es una doble barrera
#for k in range(iniciopotencial+int(10e-9/del_x),
#               iniciopotencial+int(12e-9/del_x)+1):
 #   V[k]=VG*eV2J
  #  ra[k]=(.5*hbar/(m0*meffpozo))*(dt/del_x**2)

#Descomentar si es una triple barrera
#for k in range(iniciopotencial+int(20e-9/del_x),
#               iniciopotencial+int(22e-9/del_x)+1):
 #   V[k]=VG*eV2J
  #  ra[k]=(.5*hbar/(m0*meffpozo))*(dt/del_x**2)

#Descomentar si el potencial del transistor se definir'a como rampa
#for k in range(iniciocampo,iniciopotencial):
 #   V[k]=VG*eV2J*(k-iniciocampo)/(iniciopotencial-iniciocampo)
#for k in range(iniciopotencial+int(12e-9/del_x),fincampo+1):
 #   V[k]=VG*eV2J*(k-fincampo)/(iniciopotencial+int(12e-9/del_x)-fincampo)

#Descomentar si el potencial del transistor se suavizar'a
for k in range(1,Nx+1):
    V[k]=VG*eV2J*(1/(1+((k-int(76e-9/del_x))/20.0)**10)**.5)

#Definici'on del campo el'ectrico o decaimiento potencial
for k in range(iniciocampo,fincampo):
  V[k]=V[k]-(VD/(fincampo-iniciocampo))*(k-iniciocampo)*eV2J
for k in range(fincampo,Nx+1):
  V[k]=V[k]-VD*eV2J

#Graficar perfil de potencial V
V_xgraf=np.arange(1,Nx,1)
V_ygraf=V[V_xgraf]
```

```
plt.plot(V_xgraf*Dx,V_ygraf*J2eV,label ='')
plt.ylabel('eV')
plt.xlabel('nm')
plt.savefig('V1D.eps', format='eps')

#Reiniciar y normalizar valores de psi
ptot=0
for k in range(1,Nx+1):
    prl[k]=np.exp (-1*((k-nc)/sigma)**2)*np.cos(2*np.pi*(k-nc)/lambd)
    pim[k]=np.exp (-1*((k-nc)/sigma)**2)*np.sin(2*np.pi*(k-nc)/lambd)
for k in range(1,Nx+1):
    prl[k]=prl[k]/pnorm
    pim[k]=pim[k]/pnorm

#Ciclo temporal de salida con cambios de V y ra
for m in range(0,n_step): #Pasos 1 a n_step
#Ciclo para calcular la parte real de psi
  for n in range(2,Nx): #Va de 2 a Nx-1
    deltar=prl[n-1]-2*prl[n]+prl[n+1]
    deltai=pim[n-1]-2*pim[n]+pim[n+1]
    prl_n[n]=prl[n]-ra[n]*(gamr[n]*deltai+gami[n]*deltar)
         +(dt/hbar)*V[n]*pim[n]
#Ciclo para actualizar la parte real de psi
  for n in range(2,Nx):
    prl[n]=prl_n[n]
  #Ciclo para calcular la parte imaginaria de psi
  for n in range(2,Nx):  #Va de 2 a Nx-1
    deltar=prl[n-1]-2*prl[n]+prl[n+1]
    deltai=pim[n-1]-2*pim[n]+pim[n+1]
    pim_n[n]=pim[n]+ra[n]*(gamr[n]*deltar-gami[n]*deltai)
         -(dt/hbar)*V[n]*prl[n]
```

```
#Ciclo para actualizar la parte imaginaria de psi
  for n in range(2,Nx):
    pim[n]=pim_n[n]
#Guardar datos para el monitoreo
  prl_out[m]=prl[posicion_out]
  pim_out[m]=pim[posicion_out]

#Graficar monitoreo de salida
xgraf=np.arange(0,n_step,1)
yreal=prl_out[xgraf]
yimag=pim_out[xgraf]
plt.plot(xgraf*dt*1e12,yreal,label ='$\psi_{real}$')
plt.plot(xgraf*dt*1e12,yimag,'--',label='$\psi_{imag}$')
plt.ylabel('')
plt.xlabel('ps')
plt.xlim(0,(n_step*dt*1e12))
plt.legend(loc=1)
plt.savefig('psi_out.eps', format='eps')

#C'alculo de la energ'ia de salida con FFT
data_out=prl_out
fft_out = fft(data_out)
lenx=fft_out.shape[0]
magnitud_out= [np.sqrt(i.real**2 + i.imag**2)/len(fft_out) for i in fft_out]

#Modificar escala de energ'ia de salida por voltaje de drenado
escala_e=np.zeros(n_step+1)
energia=np.zeros(n_step+1)
V_out=V[posicion_out]
for i in np.arange(1,n_step):
  energia[i]=i*hbar*2*np.pi/(dt*n_step)
```

```
  escala_e[i]=np.sqrt((energia[i]-V_out)/energia[i])
escala_e[0]=1

#Definir energ'ias y transmisi'on debidas al reescalamiento
E_in=[magnitud_in[i] for i in np.arange(0,n_step)]
E_out=[magnitud_out[i]*np.sqrt(escala_e[i]) for i in np.arange(0,n_step)]
trans=[(magnitud_out[i]/magnitud_in[i])**2*escala_e[i]
       for i in np.arange(0,len(magnitud_in))]

#Graficar monitoreos de entrada y salida en el dominio de frecuencias
Egraf=np.arange(0,n_step)
plt.plot(Egraf*J2eV*hbar*2*np.pi/(dt*n_step),E_in,'--',label='Entrada')
plt.plot(Egraf*J2eV*hbar*2*np.pi/(dt*n_step),E_out,label='Salida')
plt.xlim(0,.6)
plt.legend(loc=1)
plt.ylabel('Unidades arbitrarias')
plt.xlabel('E (eV)')
plt.savefig('transformada.eps', format='eps')

#Graficar transmisi'on
plt.plot(Egraf*J2eV*hbar*2*np.pi/(dt*n_step),trans,label='trans')
plt.xlim(0,0.6)
plt.ylim(.0,1.01)
plt.xlabel('eV')
plt.savefig('TM.eps', format='eps')

#Graficar avance de la onda en el espacio
xgrafica=np.arange(1,Nx,1)
yreal=prl[xgrafica]
yimag=pim[xgrafica]
plt.plot(xgrafica*Dx,yreal,label ='$\psi_{real}$')
```

```
plt.plot(xgrafica*Dx,yimag,'--',label='$\psi_{imag}$')
plt.ylabel('')
plt.xlabel('nm')
plt.ylim(-0.15,0.15)
plt.xlim(0,(L/1e-9))
plt.legend(loc=1)
plt.savefig('1D.eps', format='eps')

#Calcular la corriente I y conductancia G
energia[0]=0
E_f=0.35*eV2J #Energ'ia de Fermi
#Funci'on de Fermi 1
f1=[1/(1+np.exp((energia[i]-E_f)/(0.0259*eV2J)))
    for i in np.arange(0,n_step)]
#Funci'on de Fermi 1
f2=[1/(1+np.exp((energia[i]-(E_f-ecoul*VD))/(0.0259*eV2J)))
    for i in np.arange(0,n_step)]
#Funci'on para calcular I
I_funcion=[trans[i]*(f1[i]-f2[i]) for i in np.arange(0,n_step)]
from scipy import integrate
x=np.arange(0,n_step)
I=(ecoul/(dt*n_step))*integrate.simps(I_funcion,x)
print('I={} \u03BC A si se integra con comando'.format(I*1e6))
Corriente=0
for i in range (0,n_step):
  Corriente+=I_funcion[i]
Corriente=Corriente*ecoul/(dt*n_step)
print('I={} \u03BC A si se integra con ciclo'.format(Corriente*1e6))
G=I/VD
print('G={} \u03BC S con I1'.format(G*1e6))
G=Corriente/VD
```

```
print('G={} \u03BC S con I2'.format(G*1e6))

#Graficar funciones de Fermi y corriente
plt.plot(Egraf*J2eV*hbar*2*np.pi/(dt*n_step),f1,label='f1')
plt.plot(Egraf*J2eV*hbar*2*np.pi/(dt*n_step),f2,'--',label='f2')
plt.fill_between(Egraf*J2eV*hbar*2*np.pi/(dt*n_step),0,I_funcion,
                 color='g',label='I')
plt.xlim(0,0.6)
plt.ylim(.0,1.01)
plt.legend(loc=1)
plt.savefig('Fermi.eps', format='eps')

#Calcular valores esperados
#C'alculo de energ'ia cin'etica
ke=0
for k in range(2,Nx): #Va de 2 a Nx-1
    ke+=prl[k]*(prl[k+1]-2*prl[k]+prl[k-1])
    +pim[k]*(pim[k+1]-2*pim[k]+pim[k-1])
KE=-J2eV*((hbar/del_x)**2/(2*melec))*ke
print('KE={} eV'.format(KE))
pe=0
proba=0
for k in range(1,Nx+1): #Va de 1 a Nx
    pe+=(prl[k]**2+pim[k]**2)*V[k]
    proba+=prl[k]**2+pim[k]**2 #Calcula la probabilidad
print('Probabilidad={}'.format(proba))
PE=pe*J2eV
print('PE={} eV'.format(PE))
Etot=KE+PE
print('Energia total={} eV'.format(Etot))
```

```
#C'alculo de la Transformada Fourier iterada
E_eV=0.6 #Energ'ia final en eV para el ciclo
E=E_eV*eV2J
dE=0.001*eV2J #Paso de energ'ia
E_step=int(E/dE) #N'umero de pasos energ'eticos
trans=np.zeros(E_step+1)
E_in=np.zeros(E_step+1)
E_out=np.zeros(E_step+1)
for e in range (0,E_step+1):
    Sum_in_r=0
    Sum_in_i=0
    Sum_out_r=0
    Sum_out_i=0
    for k in range(0,n_step): #Sumatorias en el tiempo
        Sum_in_r+=prl_in[k]*np.cos(e*dE*(k+1)*dt/hbar)
          -pim_in[k]*np.sin(e*dE*(k+1)*dt/hbar)
        Sum_in_i+=prl_in[k]*np.sin(e*dE*(k+1)*dt/hbar)
          +pim_in[k]*np.cos(e*dE*(k+1)*dt/hbar)
        Sum_out_r+=prl_out[k]*np.cos(e*dE*(k+1)*dt/hbar)
           -pim_out[k]*np.sin(e*dE*(k+1)*dt/hbar)
        Sum_out_i+=prl_out[k]*np.sin(e*dE*(k+1)*dt/hbar)
           +pim_out[k]*np.cos(e*dE*(k+1)*dt/hbar)

    E_in[e]=Sum_in_r**2+Sum_in_i**2
    E_out[e]=Sum_out_r**2+Sum_out_i**2

#Reescalamiento de E_out iterada
escala=np.zeros(E_step+1)
energia=np.zeros(E_step+1)
for e in range (1,E_step+1):
  energia[e]=e*dE
```

```
  escala[e]=np.sqrt((energia[e]-V[posicion_out])/energia[e])
  trans[e]=E_out[e]*escala[e]/E_in[e]

escala[0]=1
trans[0]=E_out[0]/E_in[0]

#Graficar TM con FT iterada
xgraf_e=np.arange(0,E_step+1,1)
y_trans=trans[xgraf_e]
plt.plot(xgraf_e*dE*J2eV,y_trans,label='trans')
plt.ylim(0,1.03)
plt.ylabel('TM')
plt.xlabel('E (eV)')
plt.savefig('TM2.eps', format='eps')

#Calcular la corriente I y conductancia G con FT iterada
E_f=0.35*eV2J
f1=[1/(1+np.exp((energia[i]-E_f)/(0.0259*eV2J)))
   for i in np.arange(0,E_step+1)]
f2=[1/(1+np.exp((energia[i]-(E_f-ecoul*VD))/(0.0259*eV2J)))
    for i in np.arange(0,E_step+1)]
I_funcion=[trans[i]*(f1[i]-f2[i]) for i in np.arange(0,E_step+1)]
from scipy import integrate
x=np.arange(0,E_step+1)
I=(ecoul*dE/(hbar*2*np.pi))*integrate.simps(I_funcion,x)
print('I3={} \u03BC A si se integra con comando'.format(I*1e6))
Corriente=0
for i in range (0,E_step+1):
  Corriente+=I_funcion[i]
Corriente=Corriente*ecoul*dE/(hbar*2*np.pi)
print('I4={} \u03BC A si se integra con ciclo'.format(Corriente*1e6))
```

```
G=I/VD
print('G={} \u03BC S con I3'.format(G*1e6))
G=Corriente/VD
print('G={} \u03BC S con I4'.format(G*1e6))
```

I.2. Programa en forma serial en C++

```
//Programa serial unidimensional en C++
#include "../common/book.h"
#include <math.h> //Funciones cos y sin en kernel
#include <cuda_runtime.h> //Para los cuda Events

//----Definir unidades fijas-----------------------
#define L 120e-9
#define del_x 0.4e-9
#define m0 9.1e-31
#define meff 0.067 //Masa efectiva del GaAs
#define meffpozo 0.088 //Masa efectiva del Al.3Ga.7As
#define ecoul 1.6e-19 //Carga del electr'on en Coulomb
#define epsz 8.85e-19 //Dielectric del espacio libre
#define eV2J 1.6e-19 //Factor de conversi'on de energ'ia
#define hbar 1.054e-34
#define E_inicial 0.204 //Energ'ia aproximada del pulso inicial en eV.
#define VG 0.3 //Voltaje de compuerta en eV
#define VD 0.1 //Voltaje de drenado en eV o V/C
#define ener_fermi 0.35 //Energ'ia de Fermi en eV
#define picoseg 3

  int main( void ) {
     float pi;
     pi=atan(1)*4;
```

```
int Nx, nc, lambd;
Nx=round(L/del_x);
printf("Nx=%d \n", Nx);
nc=int(Nx*40/120);
float melec,J2eV, sigma, dt, KE, PE, E_total, proba;
melec=m0*meff;
J2eV=1/eV2J;
lambd=round(hbar*2*pi/(pow(E_inicial*eV2J*2*melec,0.5)*del_x));
sigma=float(lambd);
dt=2*melec*pow(del_x,2)/(8*hbar);

int n_step;
n_step=int(picoseg/(dt*1e12));
printf("n_step=%d \n", n_step);
float *prl=new float[Nx*1];
float *pim=new float[Nx*1];
float *prl_n=new float[Nx*1];
float *pim_n=new float[Nx*1];
float *V=new float[Nx*1];
float *ra=new float[Nx*1];
float *gamr=new float[Nx*1],*gami=new float[Nx*1];
float ptot,ptot2, factor_norm;
float *d_prl,*d_pim,*d_prl_n,*d_pim_n,*d_V, *d_ra;//*d_ptot;
for (int i=0; i<Nx; i++) {
    V[i] =0.0;
    ra[i] =0.125;
}

int iniciopotencial, iniciocampo, fincampo;
iniciopotencial=round(70e-9/del_x); //Inicio potencial en 70 nm
iniciocampo=round(60e-9/del_x);
```

```
    fincampo=round(90e-9/del_x);

//----------Inicializar pulso-----------
 //Se a'nade a la f'ormula i+1 en vez de i para que recorra el 'indice
    ptot=0.0;
    for (int i=0; i<(Nx); i++)  {
          prl[i] = exp(-pow((i-nc+1)/sigma,2))*cos(2*pi*(i-nc+1)/lambd);
          pim[i] = exp(-pow((i-nc+1)/sigma,2))*sin(2*pi*(i-nc+1)/lambd);
          ptot+=pow(prl[i],2)+pow(pim[i],2);
      }
     factor_norm=pow(ptot,.5);
 //----------Fin Inicio pulso-----------

 //--------Normalizaci'on del pulso---------------------
    ptot2=0.0;
    for (int i=0; i<Nx; i++) {
            prl[i]=prl[i]/factor_norm;
            pim[i]=pim[i]/factor_norm;
            ptot2+=pow(prl[i],2)+pow(pim[i],2);
      }
    printf("Psi^2 inicial=%f \n", ptot2);
 //----------Fin normalizaci'on del pulso------------

 //--------------Determinaci'on stretching coordinate--------------------
 //Los 'indices se recorren 1 unidad de partici'on
 //Valores de sigma0 e inicio pml
    int npml;
    float sigma0,sigmapml;
    sigmapml=0.0;
    npml=int(20e-9/del_x); //Equivalente a 20 nm y 50 particiones.
    sigma0=0.0005;
```

```
  for (int i=0; i<Nx; i++) {
    if (i< npml){ //Celdas de 0 a 49
      sigmapml=sigma0*powf(i-npml,2);
      gamr[i]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))/
  (pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
  +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
      gami[i]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))/
  (pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
  +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
    }
    else if (i<Nx && i>(Nx-npml-1)) {
      sigmapml=sigma0*powf(i+1-(Nx-npml),2);
      gamr[i]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))/
  (pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
      gami[i]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))/
  (pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
    }
    else  {
      gamr[i]=1.0;
      gami[i]=0.0;
    }
  }
//--------------Fin Determinaci'on stretching coordinate-------------

//-------------Determinaci'on celdas de monitoreo-------------------
  int posicion_in, posicion_out;
  posicion_in=int(95e-9/del_x);
  posicion_out=int(95e-9/del_x);
```

```
  float *prl_in=new float[n_step],*pim_in=new float[n_step],
      *prl_out=new float[n_step], *pim_out=new float[n_step];
//------------Fin Determinaci'on celdas de monitoreo-----------------

//---------Ciclo temporal sin barreras-------------------------------

  //---------Empezar a contar el tiempo de c'omputo--------------
  cudaEvent_t inicio_ciclo, fin_ciclo;
  HANDLE_ERROR( cudaEventCreate( &inicio_ciclo ) );
  HANDLE_ERROR( cudaEventCreate( &fin_ciclo ) );
  HANDLE_ERROR( cudaEventRecord( inicio_ciclo, 0 ) );
  //---------Fin de crear eventos para tiempo c'omputo-----------
  int paso;
 for (int t=0;t<n_step;t++){ //Pasos de 1 a n_step

   for (int i=1; i<(Nx-1); i++) {
      prl_n[i]=prl[i]-ra[i]*(gamr[i]*(pim[i-1]-2*pim[i]+pim[i+1])
  +gami[i]*(prl[i-1]-2*prl[i]+prl[i+1]))+(dt/hbar)*V[i]*pim[i];
   }

   for (int i=1; i<(Nx-1); i++) {
      prl[i]=prl_n[i];
   }
   prl_in[t]=prl[posicion_in];

   for (int i=1; i<(Nx-1); i++) {
      pim_n[i]=pim[i]+ra[i]*(gamr[i]*(prl[i-1]-2*prl[i]+prl[i+1])
-gami[i]*(pim[i-1]-2*pim[i]+pim[i+1]))-(dt/hbar)*V[i]*prl[i];
   }

   for (int i=1; i<(Nx-1); i++) {
```

```
      pim[i]=pim_n[i];
   }
   pim_in[t]=pim[posicion_in];
}
     //----------Impresi'on tiempo de c'omputo------------------
    HANDLE_ERROR( cudaEventRecord( fin_ciclo, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( fin_ciclo ) );
    float elapsedTime;
    HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime,inicio_ciclo,fin_ciclo));
    printf( "Tiempo de c'omputo ciclo_in: %3.2f ms\n", elapsedTime );
    HANDLE_ERROR( cudaEventDestroy( inicio_ciclo ) );
    HANDLE_ERROR( cudaEventDestroy( fin_ciclo ) );
     //--------Fin impresi'on tiempo de c'omputo-----------------

 //------Fin ciclo temporal sin barreras--------------------------------

//----------Definir zona del potencial-----------------------------------

     //-------Barrera simple----------//Va de 70 a 82 nm.
 // for(int i=iniciopotencial-1;i<(iniciopotencial+round(12e-9/del_x));i++){
    //     V[i] =VG*eV2J;
      //   ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
    // }

     //-------Doble barrera----------
//   for (int i=(iniciopotencial-1+round(10e-9/del_x));
//          i<(iniciopotencial+round(12e-9/del_x));i++){
  //       V[i] =VG*eV2J;
    //     ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
     //}
```

```
     //----Triple barrera------------
  //   for (int i=(iniciopotencial-1+int(20e-9/del_x));
//         i<(iniciopotencial+int(22e-9/del_x)); i++) {
    //     V[i] =VG*eV2J;
      //   ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
    // }

     //----Potencial en rampa-------
 //    for (int i=(iniciocampo-1); i<iniciopotencial; i++) {
   //      V[i]=VG*eV2J*(i+1-iniciocampo)/(iniciopotencial-iniciocampo);
     //}
    //for(int i=iniciopotencial-1;i<(iniciopotencial+round(12e-9/del_x));i++){
       //  V[i] =VG*eV2J;
         //ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
     //}
    //for (int i=(iniciopotencial-1+round(12e-9/del_x)); i<fincampo; i++) {
    //V[i]=VG*eV2J*(i+1-fincampo)/(iniciopotencial+round(12e-9/del_x)-fincampo);
    // }

     //----Potencial suavizado------
     for(int i=iniciopotencial-1;i<(iniciopotencial+round(12e-9/del_x));i++){
         ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
     }
     for (int i=0; i<Nx; i++) {
         V[i]=VG*eV2J*(1/pow(1+pow((i+1-round(76e-9/del_x))/20.0,10),0.5));
     }

     //----Campo el'ectrico----------
     for (int i=(iniciocampo-1); i<(fincampo-1); i++) {
         V[i] =V[i]- (VD/(fincampo-iniciocampo))*(i+1-iniciocampo)*eV2J;
     }
```

```
    for (int i=(fincampo-1); i<Nx; i++) {
        V[i] =V[i]- VD*eV2J;
    }

 //----------Fin Definici'on zona del potencial----------------------------

//--------Segunda inicializaci'on y normalizaci'on pulso-------------------
  for (int i=0; i<(Nx); i++)  {
  prl[i] = exp(-pow((i-nc+1)/sigma,2))*cos(2*pi*(i-nc+1)/lambd)/factor_norm;
  pim[i] = exp(-pow((i-nc+1)/sigma,2))*sin(2*pi*(i-nc+1)/lambd)/factor_norm;
  }
//----------Fin de la segunda inicializaci'on----------------------------

//--------Ciclo temporal con barreras de potencial-----------------------
for (int t=0;t<n_step;t++){ //Pasos de 1 a n_step
  for (int i=1; i<(Nx-1); i++) {
     prl_n[i]=prl[i]-ra[i]*(gamr[i]*(pim[i-1]-2*pim[i]+pim[i+1])
      +gami[i]*(prl[i-1]-2*prl[i]+prl[i+1]))+(dt/hbar)*V[i]*pim[i];
  }

  for (int i=1; i<(Nx-1); i++) {
     prl[i]=prl_n[i];
  }
     prl_out[t]=prl[posicion_out];

  for (int i=1; i<(Nx-1); i++) {
     pim_n[i]=pim[i]+ra[i]*(gamr[i]*(prl[i-1]-2*prl[i]+prl[i+1])
       -gami[i]*(pim[i-1]-2*pim[i]+pim[i+1]))-(dt/hbar)*V[i]*prl[i];
  }

  for (int i=1; i<(Nx-1); i++) {
```

```
     pim[i]=pim_n[i];
  }
  pim_out[t]=pim[posicion_out];

}
//----------Fin ciclo temporal con barreras de potencial---------------

//--------Transformada de Fourier de los monitores espaciales----------
    float Ef_step_eV, Ef_step_J, dE;
    int E_step;
    Ef_step_eV=0.6; //Energ'ia final de la FT
    Ef_step_J=Ef_step_eV*eV2J;
    //dE=0.0001*Ef_step_J; //Paso energ'etico
    E_step=int(600);
    //E_step=int(Ef_step_J/dE);
    dE=Ef_step_J/E_step;
    printf("pasos energeticos=%d\n", E_step);

    float *E_in=new float[E_step+1], *E_out=new float[E_step+1],
*trans=new float[E_step+1];
    float sum_in_r, sum_in_i, sum_out_r, sum_out_i;
    sum_in_r=0;
    sum_in_i=0;
    sum_out_r=0;
    sum_out_i=0;

if (n_step>0){ //Si no hay n_step no se calcula FT
  for (int e=0; e<(E_step+1); e++){
    sum_in_r=0;
    sum_in_i=0;
    sum_out_r=0;
```

```
    sum_out_i=0;

    for (int k=0; k<n_step+1; k++) {
      sum_in_r+=prl_in[k]*cos(e*dE*(k+1)*dt/hbar)
          -pim_in[k]*sin(e*dE*(k+1)*dt/hbar);
      sum_in_i+=prl_in[k]*sin(e*dE*(k+1)*dt/hbar)
          +pim_in[k]*cos(e*dE*(k+1)*dt/hbar);
      sum_out_r+=prl_out[k]*cos(e*dE*(k+1)*dt/hbar)
           -pim_out[k]*sin(e*dE*(k+1)*dt/hbar);
      sum_out_i+=prl_out[k]*sin(e*dE*(k+1)*dt/hbar)
           +pim_out[k]*cos(e*dE*(k+1)*dt/hbar);
    }
    E_in[e]=pow(sum_in_r,2)+pow(sum_in_i,2);
    E_out[e]=pow(sum_out_r,2)+pow(sum_out_i,2);
    }
}
//------------Fin Transformada de Fourier de los monitores espaciales----

//-------C'alculo de Transmisi'on y reescalamiento por E_out-----------
    //Empieza de e=1 para la escala
    trans[0]=E_out[0]/E_in[0];
    for (int e=1; e<E_step+1; e++) {
        trans[e]=(E_out[e]/E_in[e])*pow((e*dE-V[posicion_out])/(e*dE),0.5);
    }
//--------------------Fin c'alculo de Transmisi'on----------------------

//-------------------Calcular funciones de Fermi----------------------
     float E_f, sumafuncionI, corriente, G;
     float *f1=new float[E_step+1], *f2=new float[E_step+1],
       *I_funcion=new float[E_step+1];
     E_f=ener_fermi*eV2J;
```

```
    sumafuncionI=0;
     for (int e=0; e<E_step+1; e++) {
        f1[e]=1/(1+exp((e*dE-E_f)/(0.0259*eV2J)));
        f2[e]=1/(1+exp((e*dE-(E_f-ecoul*VD))/(0.0259*eV2J)));
        I_funcion[e]=trans[e]*(f1[e]-f2[e]);
        sumafuncionI+=I_funcion[e];
     }

     //-----Calcular 'area bajo la curva, I y G------
     corriente=sumafuncionI*dE*ecoul/(hbar*2*pi);
     printf("I=%f microA\n", corriente*1e6);

     G=corriente/VD;
     printf("G=%f microS\n", G*1e6);
     //----Fin cÃ¡lculo 'area bajo la curva, I y G----
//----------------Fin c'alculo de funciones de Fermi-------------------

//--------C'alculo de valores esperados--------------------------------
  KE=0;
  for (int i=1; i<(Nx-1); i++) {
    KE+=prl[i]*(prl[i+1]-2*prl[i]+prl[i-1])
    +pim[i]*(pim[i+1]-2*pim[i]+pim[i-1]);
  }
  KE=-KE*J2eV*pow(hbar/del_x,2)/(2*melec);

  PE=0.0;
  proba=0.0;
  E_total=0.0;
  for (int i=0; i<Nx; i++) {
    PE+=(pow(prl[i],2)+pow(pim[i],2))*V[i];
```

```
    proba+=pow(prl[i],2)+pow(pim[i],2);
  }
  PE=PE*J2eV;
  E_total=KE+PE;
 //--------Fin de C'alculo de valores esperados----------------------

 //--------Imprimir archivos-----------------------

//--Monitoreo de entrada--
  FILE *archivo1 = fopen("psi_in.dat","w");
    for (int i=0; i<n_step; i++) {
      fprintf(archivo1, "%f\n", prl_in[i] );
    }
    fprintf(archivo1, "\n" );
for (int i=0; i<n_step; i++) {
      fprintf(archivo1, "%f\n", pim_in[i] );
    }
     fclose ( archivo1 );

 //--Monitoreo de salida--
   FILE *archivo2 = fopen("psi_out.dat","w");
     for (int i=0; i<n_step; i++) {
   fprintf(archivo2, "%f\n", prl_out[i] );
     }
     fprintf(archivo2, "\n" );
     for (int i=0; i<n_step; i++) {
       fprintf(archivo2, "%f\n", pim_out[i] );
     }
     fclose ( archivo2 );

 //--Malla--
```

```
 FILE *archivo3 = fopen("qpsi.dat","w");
   for (int i=0; i<Nx; i++) {
     fprintf(archivo3, "%f", prl[i]);
     fprintf(archivo3, "\n" );
   }
   fprintf(archivo3, "\n" );
   for (int i=0; i<Nx; i++) {
     fprintf(archivo3, "%f", pim[i] );
     fprintf(archivo3, "\n" );
   }
   fclose ( archivo3 );

//-------Imprimir en pantalla------------

 printf("KE=%f eV\n", KE*1);
 printf("PE=%f eV\n", PE*1);
 printf("E total=%f eV\n", E_total*1);
 printf("Psi^2=%f \n", proba*1);

 delete[] prl;
 delete[] pim;
 delete[] V;
 delete[] ra;
 delete[] gamr;
 delete[] gami;
 delete[] prl_in;
 delete[] pim_in;
 delete[] prl_out;
 delete[] pim_out;
 cudaFree( d_prl );
 cudaFree( d_pim );
```

```
  cudaFree( d_prl_in );
  cudaFree( d_prl_out );
  cudaFree( d_pim_in );
  cudaFree( d_pim_out );
  cudaFree( d_V );
  cudaFree( d_ra );
  cudaFree( d_gamr );
  cudaFree( d_gami );
  cudaFree( d_E_in );
  cudaFree( d_E_out );
  return 0;
}
```

I.3. Programa paralelizado en CUDA

```
//Programa unidimensional paralelizado en CUDA C++
#include "../common/book.h"
#include <math.h>
#include <cuda_runtime.h>

//----Definir unidades fijas-----------------------
#define L 120e-9
#define del_x 0.4e-9
#define m0 9.1e-31
#define meff 0.067 //Masa efectiva del GaAs
#define meffpozo 0.088 //Masa efectiva del Al.3Ga.7As
#define ecoul 1.6e-19 //Carga del electr'on en Coulomb
#define epsz 8.85e-19 //Dielectric del espacio libre
#define eV2J 1.6e-19 //Factor de conversi'on de energ'ia
#define hbar 1.054e-34
#define E_inicial 0.204 //Energ'ia aproximada del pulso inicial en eV.
```

```
#define VG 0.3 //Voltaje de compuerta en eV
#define VD 0.1 //Voltaje de drenado en eV o V/C
#define ener_fermi 0.35 //Energ'ia de Fermi en eV
#define picoseg 3

//------------Definiciones de los kernels----------------------
 __global__ void pulso_inicial (float *d_prl, float *d_pim,float sigma,
            float pi, int lambd, float ptot, int nc, int Nx){
   int x = blockIdx.x * blockDim.x + threadIdx.x;

   if (x< Nx) {
     d_prl[x] = exp(-pow((x-nc+1)/sigma,2))*cos(2*pi*(x-nc+1)/lambd);
     d_pim[x] = exp(-pow((x-nc+1)/sigma,2))*sin(2*pi*(x-nc+1)/lambd);
   }
 }

__global__ void pulso_normal (float *d_prl, float *d_pim,
            float factor_norm,int Nx){
  int x = blockIdx.x * blockDim.x + threadIdx.x;

  if (x< Nx) {
    d_prl[x]=d_prl[x]/factor_norm;
    d_pim[x]=d_pim[x]/factor_norm;
  }
}

__global__ void calculo_gamma (float *d_gamr, float *d_gami, int Nx,
           float sigma0, int npml) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;

  float sigmapml;
```

```
  if (x< npml){ //Celdas de 0 a 49
    sigmapml=sigma0*powf(x-npml,2);
    d_gamr[x]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))/
   (pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
   +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    d_gami[x]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))/
   (pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
 +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

  if (x<Nx && x>(Nx-npml-1)) {
    sigmapml=sigma0*powf(x+1-(Nx-npml),2);
    d_gamr[x]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))/
  (pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    d_gami[x]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))/
  (pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
       }

   if (x<= (Nx-npml-1) && x>=npml) {
     d_gamr[x]=1.0;
     d_gami[x]=0.0;
   }
}

__global__ void prl_nueva (float *d_prl, float *d_prl_n, float *d_pim,
    int Nx, float dt,float *d_V, float*d_ra, float *d_gamr, float *d_gami){
  int x = blockIdx.x * blockDim.x + threadIdx.x;

  if (x< (Nx-1) && x>0) {
```

```
    d_prl_n[x]=d_prl[x]-d_ra[x]*(d_gamr[x]*(d_pim[x-1]-2*d_pim[x]+d_pim[x+1])
+d_gami[x]*(d_prl[x-1]-2*d_prl[x]+d_prl[x+1]))+(dt/hbar)*d_V[x]*d_pim[x];
  }
}

__global__ void prl_reescribir (float *d_prl, float *d_prl_n, int Nx,
           float *d_prl_in, int posicion_in, int paso) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;

  if (x< (Nx-1) && x>0) {
    d_prl[x]=d_prl_n[x];
  }
  d_prl_in[paso]=d_prl_n[posicion_in];
}

__global__ void prl_reescribir_out (float *d_prl, float *d_prl_n, int Nx,
           float *d_prl_out, int posicion_out, int paso) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;

  if (x< (Nx-1) && x>0) {
    d_prl[x]=d_prl_n[x];
  }
  d_prl_out[paso]=d_prl_n[posicion_out];
}

__global__ void pim_nueva (float *d_prl, float *d_pim, float *d_pim_n,int Nx,
         float dt,float *d_V, float*d_ra, float *d_gamr, float *d_gami) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;

  if (x< (Nx-1) && x>0) {
    d_pim_n[x]=d_pim[x]+d_ra[x]*(d_gamr[x]*(d_prl[x-1]-2*d_prl[x]+d_prl[x+1])
```

```
 -d_gami[x]*(d_pim[x-1]-2*d_pim[x]+d_pim[x+1]))-(dt/hbar)*d_V[x]*d_prl[x];
  }
}

__global__ void pim_reescribir (float *d_pim,float *d_pim_n,int Nx,
           float *d_pim_in, int posicion_in, int paso) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;

  if (x< (Nx-1) && x>0) {
    d_pim[x]=d_pim_n[x];
  }
  d_pim_in[paso]=d_pim_n[posicion_in];
}

__global__ void pim_reescribir_out (float *d_pim,float *d_pim_n,int Nx,
           float *d_pim_out, int posicion_out, int paso) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;

  if (x< (Nx-1) && x>0) {
    d_pim[x]=d_pim_n[x];
  }
  d_pim_out[paso]=d_pim_n[posicion_out];
}

__global__ void transformada_fourier (float *d_E_in, float *d_E_out,
           int n_step, float dt, float *d_prl_out, float *d_pim_out,
           float *d_prl_in, float *d_pim_in, float dE, int E_step) {
  int e = blockIdx.x * blockDim.x + threadIdx.x;

  float sum_in_r, sum_in_i, sum_out_r, sum_out_i;
  if (e< (E_step+1)) {
```

```
    sum_in_r=0;
    sum_in_i=0;
    sum_out_r=0;
    sum_out_i=0;

    for (int k=0; k<n_step+1; k++) {
      sum_in_r+=d_prl_in[k]*cos(e*dE*(k+1)*dt/hbar)
          -d_pim_in[k]*sin(e*dE*(k+1)*dt/hbar);
      sum_in_i+=d_prl_in[k]*sin(e*dE*(k+1)*dt/hbar)
          +d_pim_in[k]*cos(e*dE*(k+1)*dt/hbar);
      sum_out_r+=d_prl_out[k]*cos(e*dE*(k+1)*dt/hbar)
          -d_pim_out[k]*sin(e*dE*(k+1)*dt/hbar);
      sum_out_i+=d_prl_out[k]*sin(e*dE*(k+1)*dt/hbar)
          +d_pim_out[k]*cos(e*dE*(k+1)*dt/hbar);
    }
    d_E_in[e]=pow(sum_in_r,2)+pow(sum_in_i,2);
    d_E_out[e]=pow(sum_out_r,2)+pow(sum_out_i,2);
    }
}
//--------Fin de definiciones de los kernels--------------

//----------Inicio del programa--------------
int main( void ) {
  float pi;
  pi=atan(1)*4;
  int Nx, nc, lambd;
  Nx=round(L/del_x);
  printf("Nx=%d \n", Nx);
  nc=int(Nx*40/120);
  float melec,J2eV, sigma, dt, KE, PE, E_total, proba;
  melec=m0*meff;
```

```
  J2eV=1/eV2J;
  lambd=round(hbar*2*pi/(pow(E_inicial*eV2J*2*melec,0.5)*del_x));
  sigma=float(lambd);
  dt=2*melec*pow(del_x,2)/(8*hbar);

  int n_step;
  n_step=int(picoseg/(dt*1e12));
  printf("n_step=%d \n", n_step);
  float *prl=new float[Nx*1];
  float *pim=new float[Nx*1];
  float *V=new float[Nx*1];
  float *ra=new float[Nx*1];
  float *gamr=new float[Nx*1],*gami=new float[Nx*1];
  float ptot,ptot2, factor_norm;
  float *d_prl,*d_pim,*d_prl_n,*d_pim_n,*d_V, *d_ra;//*d_ptot;
  for (int i=0; i<Nx; i++) {
    V[i] =0.0;
    ra[i] =0.125;
  }

  int iniciopotencial, iniciocampo, fincampo;
  iniciopotencial=round(70e-9/del_x);
  iniciocampo=round(60e-9/del_x);
  fincampo=round(90e-9/del_x);

//----------Inicializar pulso-----------

//Definir tama'no de la malla
  dim3 grid(512,1,1);
  dim3 block(128,1,1);
//Fin definici'on tama'no de la malla
```

```
 cudaMalloc( (void**)&d_prl, Nx * sizeof(float)  );
 cudaMalloc( (void**)&d_pim, Nx * sizeof(float)  );

 pulso_inicial<<<grid,block>>>(d_prl,d_pim, sigma, pi, lambd, ptot, nc, Nx);
 HANDLE_ERROR(cudaMemcpy(prl,d_prl,Nx*sizeof(float),cudaMemcpyDeviceToHost));
 HANDLE_ERROR(cudaMemcpy(pim,d_pim,Nx*sizeof(float),cudaMemcpyDeviceToHost));

 ptot=0.0;
 for (int i=0; i<(Nx); i++) {
   ptot+=pow(prl[i],2)+pow(pim[i],2);
 }

 factor_norm=pow(ptot,.5);
//----------Fin Inicio pulso-----------

//--------Normalizaci'on del pulso---------------------
 cudaMalloc( (void**)&d_V, Nx * sizeof(float)  );
 cudaMalloc( (void**)&d_ra, Nx * sizeof(float)  );

 pulso_normal<<<grid,block>>>(d_prl,d_pim,factor_norm,Nx);

 HANDLE_ERROR(cudaMemcpy(prl,d_prl,Nx*sizeof(float),cudaMemcpyDeviceToHost));
 HANDLE_ERROR(cudaMemcpy(pim,d_pim,Nx*sizeof(float),cudaMemcpyDeviceToHost));
    //Verificar normalizaci'on
 ptot2=0.0;
 for (int i=0; i<Nx; i++) {
   ptot2+=pow(prl[i],2)+pow(pim[i],2);
 }
 printf("ptot=%f \n", ptot2);
//----------Fin normalizaci'on del pulso------------
```

```
 //--------------Determinaci'on par'ametros PML--------------------
  int npml;
  float sigma0;
  float *d_gamr,*d_gami;
  npml=int(20e-9/del_x); //Equivalente a 20 nm y 50 particiones.
  sigma0=0.0005;

  cudaMalloc( (void**)&d_gamr, Nx * sizeof(float)  );
  cudaMalloc( (void**)&d_gami, Nx * sizeof(float)  );

  calculo_gamma<<<grid,block>>>(d_gamr, d_gami, Nx, sigma0, npml);

  HANDLE_ERROR(cudaMemcpy(gamr,d_gamr,Nx*sizeof(float),cudaMemcpyDeviceToHost));
  HANDLE_ERROR(cudaMemcpy(gami,d_gami,Nx*sizeof(float),cudaMemcpyDeviceToHost));
//--------------Fin Determinaci'on par'ametros PML-------------

 //-------------Determinaci'on celdas de monitoreo-------------------
  int posicion_in, posicion_out;
  posicion_in=int(95e-9/del_x);
  posicion_out=int(95e-9/del_x);

  float *prl_in=new float[n_step],*pim_in=new float[n_step],
    *prl_out=new float[n_step], *pim_out=new float[n_step];
  float *d_prl_in, *d_pim_in, *d_prl_out, *d_pim_out;

  cudaMalloc( (void**)&d_prl_in, n_step * sizeof(float)  );
  cudaMalloc( (void**)&d_pim_in, n_step * sizeof(float)  );
  cudaMalloc( (void**)&d_prl_out, n_step * sizeof(float)  );
  cudaMalloc( (void**)&d_pim_out, n_step * sizeof(float)  );
 //------------Fin Determinaci'on celdas de monitoreo----------------
```

```
//---------Ciclo temporal sin barreras----------------------------------------
 HANDLE_ERROR(cudaMemcpy(d_V,V,Nx*sizeof(float),cudaMemcpyHostToDevice));
 HANDLE_ERROR(cudaMemcpy(d_ra,ra,Nx*sizeof(float),cudaMemcpyHostToDevice));
 HANDLE_ERROR(cudaMemcpy(d_gamr,gamr,Nx*sizeof(float),cudaMemcpyHostToDevice));
 HANDLE_ERROR(cudaMemcpy(d_gami,gami,Nx*sizeof(float),cudaMemcpyHostToDevice));

 cudaMalloc( (void**)&d_prl_n, Nx * sizeof(float)  );
 cudaMalloc( (void**)&d_pim_n, Nx * sizeof(float)  );
 HANDLE_ERROR(cudaMemcpy(d_prl,prl,Nx*sizeof(float),cudaMemcpyHostToDevice));
 HANDLE_ERROR(cudaMemcpy(d_pim,pim,Nx*sizeof(float),cudaMemcpyHostToDevice));

    //---------Empezar a contar el tiempo de c'omputo--------------
 cudaEvent_t inicio_ciclo, fin_ciclo;

 HANDLE_ERROR( cudaEventCreate( &inicio_ciclo ) );
 HANDLE_ERROR( cudaEventCreate( &fin_ciclo ) );
 HANDLE_ERROR( cudaEventRecord( inicio_ciclo, 0 ) );
    //---------Fin de crear eventos para tiempo c'omputo-----------
 int paso;
for (int t=0;t<n_step;t++){ //Pasos de 1 a n_step
  paso=t;

  prl_nueva<<<grid,block>>>(d_prl, d_prl_n, d_pim, Nx, dt,
                            d_V, d_ra, d_gamr, d_gami);
  prl_reescribir<<<grid,block>>>(d_prl, d_prl_n, Nx, d_prl_in,
                            posicion_in, paso);
  pim_nueva<<<grid,block>>>(d_prl, d_pim, d_pim_n, Nx, dt,
                            d_V, d_ra, d_gamr, d_gami);
  pim_reescribir<<<grid,block>>>(d_pim, d_pim_n, Nx, d_pim_in,
                            posicion_in, paso);
```

```
}

     //----------Impresi'on tiempo de c'omputo------------------
   HANDLE_ERROR( cudaEventRecord( fin_ciclo, 0 ) );
   HANDLE_ERROR( cudaEventSynchronize( fin_ciclo ) );

   float elapsedTime;

 HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime,inicio_ciclo,fin_ciclo));

   printf("Tiempo de c'omputo ciclo_in: %3.2f ms\n", elapsedTime );

   HANDLE_ERROR( cudaEventDestroy( inicio_ciclo ) );
   HANDLE_ERROR( cudaEventDestroy( fin_ciclo ) );
   //--------Fin impresi'on tiempo de c'omputo-----------------

   HANDLE_ERROR(cudaMemcpy(prl,d_prl,Nx*sizeof(float),cudaMemcpyDeviceToHost));
   HANDLE_ERROR(cudaMemcpy(pim,d_pim,Nx*sizeof(float),cudaMemcpyDeviceToHost));

   HANDLE_ERROR(cudaMemcpy(prl_in,d_prl_in,n_step*sizeof(float),
              cudaMemcpyDeviceToHost));
   HANDLE_ERROR(cudaMemcpy(pim_in,d_pim_in,n_step*sizeof(float),
              cudaMemcpyDeviceToHost));
 //------Fin ciclo temporal sin barreras-----------------------------------

//----------Definir zona del potencial-------------------------------------

     //-------Barrera simple----------//Va de 70 a 82 nm.
  //for(int i=iniciopotencial-1;i<(iniciopotencial+round(12e-9/del_x));i++){
    //V[i] =VG*eV2J;
    //ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
```

```
  //}

     //-------Doble barrera----------
//  for (int i=(iniciopotencial-1+round(10e-9/del_x));
    //     i<(iniciopotencial+round(12e-9/del_x));i++){
  //  V[i] =VG*eV2J;
    //ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
  //}

     //----Triple barrera------------
  // for (int i=(iniciopotencial-1+int(20e-9/del_x));
  //      i<(iniciopotencial+int(22e-9/del_x));i++){
    // V[i] =VG*eV2J;
     //ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
  // }

     //----Potencial en rampa-------
// for (int i=(iniciocampo-1); i<iniciopotencial; i++) {
 //  V[i]=VG*eV2J*(i+1-iniciocampo)/(iniciopotencial-iniciocampo);
 //}
 //for (int i=iniciopotencial-1; i<(iniciopotencial+round(12e-9/del_x)); i++){
  //  V[i] =VG*eV2J;
   //ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
   //}
  //for (int i=(iniciopotencial-1+round(12e-9/del_x)); i<fincampo; i++) {
 //   V[i]=VG*eV2J*(i+1-fincampo)/(iniciopotencial+round(12e-9/del_x)-fincampo);
 // }

     //----Potencial suavizado------
     //Punto medio debe ser flotante por si se encuentra entre celdas
  for (int i=iniciopotencial-1; i<(iniciopotencial+round(12e-9/del_x));i++){
```

```
    ra[i]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
  }
  for (int i=0; i<Nx; i++) {
    V[i]=VG*eV2J*(1/pow(1+pow((i+1-round(76e-9/del_x))/20.0,10),0.5));
  }

  //----Campo el'ectrico----------
  for (int i=(iniciocampo-1); i<(fincampo-1); i++) {
    V[i] =V[i]- (VD/(fincampo-iniciocampo))*(i+1-iniciocampo)*eV2J;
  }
  for (int i=(fincampo-1); i<Nx; i++) {
    V[i] =V[i]- VD*eV2J;
  }

 //----------Fin Definici'on zona del potencial----------------------------

//----------Segunda inicializaci'on pulso----------------------------------
  pulso_inicial<<<grid,block>>>(d_prl,d_pim, sigma, pi, lambd, ptot, nc, Nx);

  pulso_normal<<<grid,block>>>(d_prl,d_pim,factor_norm,Nx);

//----------Fin de la segunda inicializaci'on----------------------------

//-----------------Ciclo temporal con barreras de potencial---------------
  HANDLE_ERROR(cudaMemcpy(d_V,V,Nx*sizeof(float),cudaMemcpyHostToDevice));
  HANDLE_ERROR(cudaMemcpy(d_ra,ra,Nx*sizeof(float),cudaMemcpyHostToDevice));

for (int t=0;t<n_step;t++){ //Pasos de 1 a n_step
  paso=t;

 prl_nueva<<<grid,block>>>(d_prl, d_prl_n, d_pim, Nx, dt, d_V,
```

```
                            d_ra, d_gamr, d_gami);
 prl_reescribir_out<<<grid,block>>>(d_prl, d_prl_n, Nx, d_prl_out,
                            posicion_out, paso);
 pim_nueva<<<grid,block>>>(d_prl, d_pim, d_pim_n, Nx, dt, d_V,
                            d_ra, d_gamr, d_gami);
 pim_reescribir_out<<<grid,block>>>(d_pim, d_pim_n, Nx, d_pim_out,
                            posicion_out, paso);
}

 HANDLE_ERROR(cudaMemcpy(prl_out,d_prl_out,n_step*sizeof(float),
              cudaMemcpyDeviceToHost));
 HANDLE_ERROR(cudaMemcpy(pim_out,d_pim_out,n_step*sizeof(float),
              cudaMemcpyDeviceToHost));
//-----------Fin ciclo temporal con barreras de potencial---------

//---------Transformada de Fourier de los monitores espaciales-------
    float Ef_step_eV, Ef_step_J, dE;
    int E_step;
    Ef_step_eV=0.6; //Energ'ia final de la FT
    Ef_step_J=Ef_step_eV*eV2J;
    E_step=int(600); //N'umero de pasos energ'eticos
    dE=Ef_step_J/E_step;
    printf("pasos energeticos=%d\n", E_step);

    float *E_in=new float[E_step+1], *E_out=new float[E_step+1],
      *trans=new float[E_step+1];
    float *d_E_in, *d_E_out;

    cudaMalloc( (void**)&d_E_in, (E_step+1) * sizeof(float)  );
    cudaMalloc( (void**)&d_E_out, (E_step+1) * sizeof(float)  );
```

```
    //----Definir grid y block por c'alculo FT----------
    dim3 malla(32,1,1);
    dim3 bloque(320,1,1);
    //----Fin definici'on nueva grid y block por FT-----

  HANDLE_ERROR(cudaMemcpy(d_prl_in,prl_in,n_step*sizeof(float),
              cudaMemcpyHostToDevice));
  HANDLE_ERROR(cudaMemcpy(d_pim_in,pim_in,n_step*sizeof(float),
              cudaMemcpyHostToDevice));
  HANDLE_ERROR(cudaMemcpy(d_prl_out,prl_out,n_step*sizeof(float),
              cudaMemcpyHostToDevice));
  HANDLE_ERROR(cudaMemcpy(d_pim_out,pim_out,n_step*sizeof(float),
              cudaMemcpyHostToDevice));

  if (n_step>0){
    transformada_fourier<<<malla,bloque>>>(d_E_in, d_E_out, n_step, dt,
               d_prl_out, d_pim_out, d_prl_in, d_pim_in, dE, E_step);
  }

  HANDLE_ERROR(cudaMemcpy(E_in,d_E_in,(E_step+1)*sizeof(float),
              cudaMemcpyDeviceToHost));
  HANDLE_ERROR(cudaMemcpy(E_out,d_E_out,(E_step+1)*sizeof(float),
               cudaMemcpyDeviceToHost));

//----------Fin Transformada de Fourier de los monitores espaciales---------

//--------C'alculo de Transmisi'on y reescalamiento de E_out---------------
    //Empieza de e=1 para la escala
    trans[0]=E_out[0]/E_in[0];
    for (int e=1; e<E_step+1; e++) {
       trans[e]=(E_out[e]/E_in[e])*pow((e*dE-V[posicion_out])/(e*dE),0.5);
```

```
    }
//-----------------------Fin c'alculo de Transmisi'on----------------------

//-----------------------Calcular funciones de Fermi-----------------------
  float E_f, sumafuncionI, corriente, G;
  float *f1=new float[E_step+1], *f2=new float[E_step+1],
        *I_funcion=new float[E_step+1];
  E_f=ener_fermi*eV2J;

  sumafuncionI=0;
  for (int e=0; e<E_step+1; e++) {
    f1[e]=1/(1+exp((e*dE-E_f)/(0.0259*eV2J)));
    f2[e]=1/(1+exp((e*dE-(E_f-ecoul*VD))/(0.0259*eV2J)));
    I_funcion[e]=trans[e]*(f1[e]-f2[e]);
sumafuncionI+=I_funcion[e];
  }

  //-----Calcular 'area bajo la curva, I y G------
  corriente=sumafuncionI*dE*ecoul/(hbar*2*pi);
  printf("I=%f microA\n", corriente*1e6);

  G=corriente/VD;
  printf("G=%f microS\n", G*1e6);
  //----Fin c'alculo 'area bajo la curva, I y G----

//-------------Fin c'alculo de funciones de Fermi----------------------

  HANDLE_ERROR(cudaMemcpy(prl,d_prl,Nx*sizeof(float),cudaMemcpyDeviceToHost));
  HANDLE_ERROR(cudaMemcpy(pim,d_pim,Nx*sizeof(float),cudaMemcpyDeviceToHost));

//--------C'alculo de valores esperados-------------------------------
```

```
  KE=0;
  for (int i=1; i<(Nx-1); i++) {
    KE+=prl[i]*(prl[i+1]-2*prl[i]+prl[i-1])
    +pim[i]*(pim[i+1]-2*pim[i]+pim[i-1]);
  }
  KE=-KE*J2eV*pow(hbar/del_x,2)/(2*melec);

  PE=0.0;
  proba=0.0;
  E_total=0.0;
  for (int i=0; i<Nx; i++) {
    PE+=(pow(prl[i],2)+pow(pim[i],2))*V[i];
    proba+=pow(prl[i],2)+pow(pim[i],2);
  }
  PE=PE*J2eV;
  E_total=KE+PE;
 //--------Fin de C'alculo de valores esperados----------------------

 //--------Imprimir archivos------------------------
//--Monitoreo de entrada--
  FILE *archivo1 = fopen("psi_in.dat","w");
    for (int i=0; i<n_step; i++) {
      fprintf(archivo1, "%f\n", prl_in[i] );
    }
    fprintf(archivo1, "\n" );
for (int i=0; i<n_step; i++) {
      fprintf(archivo1, "%f\n", pim_in[i] );
    }
     fclose ( archivo1 );

 //--Monitoreo de salida--
```

```
  FILE *archivo2 = fopen("psi_out.dat","w");
    for (int i=0; i<n_step; i++) {
  fprintf(archivo2, "%f\n", prl_out[i] );
    }
    fprintf(archivo2, "\n" );
    for (int i=0; i<n_step; i++) {
      fprintf(archivo2, "%f\n", pim_out[i] );
    }
    fclose ( archivo2 );

//--Malla--
  FILE *archivo3 = fopen("qpsi.dat","w");
    for (int i=0; i<Nx; i++) {
      fprintf(archivo3, "%f", prl[i]);
      fprintf(archivo3, "\n" );
    }
    fprintf(archivo3, "\n" );
    for (int i=0; i<Nx; i++) {
      fprintf(archivo3, "%f", pim[i] );
      fprintf(archivo3, "\n" );
    }
    fclose ( archivo3 );

//-------Imprimir en pantalla------------

 printf("KE=%f eV\n", KE*1);
 printf("PE=%f eV\n", PE*1);
 printf("E total=%f eV\n", E_total*1);
 printf("Psi^2=%f \n", proba*1);

 delete[] prl;
```

```
  delete[] pim;
  delete[] V;
  delete[] ra;
  delete[] gamr;
  delete[] gami;
  delete[] prl_in;
  delete[] pim_in;
  delete[] prl_out;
  delete[] pim_out;
  cudaFree( d_prl );
  cudaFree( d_pim );
  cudaFree( d_prl_in );
  cudaFree( d_prl_out );
  cudaFree( d_pim_in );
  cudaFree( d_pim_out );
  cudaFree( d_V );
  cudaFree( d_ra );
  cudaFree( d_gamr );
  cudaFree( d_gami );
  cudaFree( d_E_in );
  cudaFree( d_E_out );
  return 0;
}
```

Apéndice II

Códigos bidimensionales

Los siguientes códigos se utilizaron para obtener las figuras y cálculos de la programación en una dimensión de la sección 5.1. El primer código está escrito en lenguaje Python de forma serial, el segundo código está escrito en forma serial para C++ y finalmente el tercer código está escrito en forma paralelizada para CUDA C++. Las figuras son realizadas con la herramienta de Matplotlib de Python especificada en el código, aunque son coherentes a las impresiones de datos de los códigos en C++ y CUDA. Para obtener visualizaciones en el tiempo se imprimieron archivos tipo vtk para realizar videos mediante ParaView.

II.1. Programa 2D en forma serial en Python

Al igual que en el código en una dimensión, algunas operaciones se parten para que sean legibles en el ancho de la página, aunque esto se debe evitar porque el lenguaje lo interpreta como una nueva instrucción.

```
#Programa para simular la absorci'on de un pulso gaussiano en 2D
import numpy as np
from matplotlib import pyplot as plt
% matplotlib inline


#Definici'on de unidades fijas
L=120e-9 #Longitud de la malla en metros
```

```
del_x=.4e-9  #Tama'no de la celda
Nx=int(L/del_x)+1  #N'umero de puntos en la partici'on
Ny=Nx #Sistema cuadrado
hbar=1.054E-34
m0=9.1E-31  #Masa del electr'on en un espacio libre
meff=0.067  #Masa efectiva del GaAs
melec=m0*meff  #Masa del electr'on
ecoul=1.6e-19  #Carga del electr'on
epsz=8.85e-9  #Dielectric del espacio libre
eV2J=1.6e-19  #Factor de conversi'on de energ'ia
J2eV=1/eV2J
dt=2*melec*del_x**2/(8*hbar) #Paso tiempo considerando que del_x=del_y
ra=np.zeros((Nx+1,Ny+1))
Dx=del_x*1e9  #Tama'no de la celda en nm
ra.fill((.5*hbar/melec)*(dt/del_x**2)) #ra=0,125
V=np.zeros((Nx+1,Ny+1))

#Calcular longitud de onda para pulso gaussiano
lamb=4.135e-15*eV2J/np.sqrt(0.05*eV2J*2*melec) #m
lambd=round(lamb/del_x)
sigma=float(lambd)
nc_x=int(Nx/2) #Centro del pulso inicial en x
nc_y=int(Ny/2) #Centro del pulso inicial en y
prl=np.zeros((Nx+1,Ny+1)) #Parte real del estado variable
pim=np.zeros((Nx+1,Ny+1)) #Parte real del estado variable

#Ciclo para inicializar onda sinoidal.
ptot=0.0
for k in range(1,Ny+1): #Va de 1 a Ny
  for j in range(1,Nx+1):
    prl[k][j]=np.exp(-(((k-nc_y)**2+(j-nc_x)**2)**.5/sigma)**2)
```

```
       *np.cos(2*np.pi*((k-nc_y)**2+(j-nc_x)**2)**.5/lambd)
    pim[k][j]=np.exp(-(((k-nc_y)**2+(j-nc_x)**2)**.5/sigma)**2)
       *np.sin(2*np.pi*((k-nc_y)**2+(j-nc_x)**2)**.5/lambd)
    ptot+=prl[k][j]**2+pim[k][j]**2

pnorm=np.sqrt(ptot)

#Ciclo para normalizar.
ptot=0.0
for k in range(1,Ny+1): #Va de 1 a Ny
  for j in range(1,Nx+1):
    prl[k][j]=prl[k][j]/pnorm
    pim[k][j]=pim[k][j]/pnorm
    ptot+=prl[k][j]**2+pim[k][j]**2

print("Psi^2 inicial={} eV".format(ptot))

#Modificar tiempo
femtoseg=0 #0, 100, 250 fs
n_step=int(femtoseg/(dt*1e15))

#Indicar par'ametros para formar la PML
npml=50 #Equivale a 20 nm
sigma0=0.0005
gamr=np.zeros((Nx+1,Ny+1))
gami=np.zeros((Nx+1,Ny+1))
R=(1.0+1.0j)/np.sqrt(2.0) #El j indica n'umero complejo
for k in range(1,Ny+1): #Va de 1 a Ny
  for j in range(1,Nx+1):
    if k<npml+1 and k<=j and k<=(Nx-j):
      sigmapml=sigma0*(k-1-npml)**2
```

```
      gamma=(1.0/(1.0+R*sigmapml))**2
      gammar=gamma.real
      gammai=gamma.imag
    elif k>Ny-npml and k>=j and k>=(Ny-j):
      sigmapml=sigma0*(k-(Ny-npml))**2
      gamma=(1.0/(1.0+R*sigmapml))**2
      gammar=gamma.real
      gammai=gamma.imag
    elif j>Nx-npml and k<=j and k>=(Ny-j):
      sigmapml=sigma0*(j-(Nx-npml))**2
      gamma=(1.0/(1.0+R*sigmapml))**2
      gammar=gamma.real
      gammai=gamma.imag
    elif j<npml+1 and k>=j and k<=(Ny-j):
      sigmapml=sigma0*(j-1-npml)**2
      gamma=(1.0/(1.0+R*sigmapml))**2
      gammar=gamma.real
      gammai=gamma.imag
    else:
      gammar=1.0
      gammai=0.0
    gamr[k][j]=gammar
    gami[k][j]=gammai

#Graficar valores de gamma compleja en 3D
from mpl_toolkits.mplot3d import Axes3D
x = np.linspace(0,120,301)
y = np.linspace(0,120,301)
X,Y = np.meshgrid(x,y)
fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
superficie=ax.plot_surface(X, Y, gamr ,rstride=1, cstride=1, cmap=plt.cm.jet,
                           linewidth=0, vmin=-0.25, vmax=1.0)
ax.set_xlim(0.4, 120)
ax.set_ylim(0.4, 120)
ax.set_zlim(-0.25, 1.0)
ax.set_xlabel('nm')
ax.set_ylabel('nm')
fig.colorbar(superficie, shrink=0.5)
plt.savefig('gamr2D.eps', format='eps')
fig = plt.figure()
ax = fig.gca(projection='3d')
superficie=ax.plot_surface(X, Y, gami ,rstride=1, cstride=1, cmap=plt.cm.jet,
                           linewidth=0, vmin=-0.25, vmax=1.0)
ax.set_xlim(0.4, 120)
ax.set_ylim(0.4, 120)
ax.set_zlim(-0.25, 0.2)
ax.set_xlabel('nm')
ax.set_ylabel('nm')
fig.colorbar(superficie, shrink=0.5)
plt.savefig('gami2D.eps', format='eps')

#Ciclo de tiempo. N'ucleo del FDTD.
prl_n=np.zeros((Nx+1,Ny+1))
pim_n=np.zeros((Nx+1,Ny+1))
import time
starting_point = time.time()
for m in range(0,n_step): #Pasos 1 a n_step. Ciclo temporal
  #Ciclo de parte real
  for k in range(2,Ny): #Ciclo en eje "y"
    for j in range(2,Nx): #Ciclo en eje "x"
      deltar_x=prl[k][j-1]-2*prl[k][j]+prl[k][j+1]
```

```
      deltar_y=prl[k-1][j]-2*prl[k][j]+prl[k+1][j]
      deltai_x=pim[k][j-1]-2*pim[k][j]+pim[k][j+1]
      deltai_y=pim[k-1][j]-2*pim[k][j]+pim[k+1][j]
      prl_n[k][j]=prl[k][j]-ra[k][j]*(gamr[k][j]*(deltai_x+deltai_y)
     +gami[k][j]*(deltar_x+deltar_y))+(dt/hbar)*V[k][j]*pim[k][j]

  #Reescribir parte real
  for k in range(2,Ny):
    for j in range(2,Nx):
      prl[k][j]=prl_n[k][j]

  #Ciclo de parte imaginaria
  for k in range(2,Ny): #Ciclo en eje "y"
    for j in range(2,Nx): #Ciclo en eje "x"
      deltar_x=prl[k][j-1]-2*prl[k][j]+prl[k][j+1]
      deltar_y=prl[k-1][j]-2*prl[k][j]+prl[k+1][j]
      deltai_x=pim[k][j-1]-2*pim[k][j]+pim[k][j+1]
      deltai_y=pim[k-1][j]-2*pim[k][j]+pim[k+1][j]
      pim_n[k][j]=pim[k][j]+ra[k][j]*(gamr[k][j]*(deltar_x+deltar_y)
     -gami[k][j]*(deltai_x+deltai_y))-(dt/hbar)*V[k][j]*prl[k][j]

  #Reescribir parte imaginaria
  for k in range(2,Ny):
    for j in range(2,Nx):
      pim[k][j]=pim_n[k][j]

elapsed_time = time.time () - starting_point
print("Tiempo de c'omputo del ciclo temporal={} ms".format(elapsed_time*1000))

#Graficar valores psi
fig = plt.figure()
```

```
ax = fig.gca(projection='3d')
superficie=ax.plot_surface(X, Y, prl ,rstride=1, cstride=1, cmap=plt.cm.jet,
                           linewidth=0, vmin=-0.015, vmax=0.015)
ax.set_xlim(0.4, 120)
ax.set_ylim(0.4, 120)
ax.set_zlim(-0.015, 0.015)
ax.set_xlabel('nm')
ax.set_ylabel('nm')
fig.colorbar(superficie, shrink=0.5)
plt.savefig('prl2D.eps', format='eps')
fig = plt.figure()
ax = fig.gca(projection='3d')
superficie=ax.plot_surface(X, Y, pim ,rstride=1, cstride=1, cmap=plt.cm.jet,
                           linewidth=0, vmin=-0.015, vmax=0.015)
ax.set_xlim(0.4, 120)
ax.set_ylim(0.4, 120)
ax.set_zlim(-0.015, 0.015)
ax.set_xlabel('nm')
ax.set_ylabel('nm')
fig.colorbar(superficie, shrink=0.5)
plt.savefig('pim2D.eps', format='eps')

#C'alculo de energ'ia cin'etica
ke=0
for k in range(2,Ny): #Va de 2 a Ny-1
  for j in range(2,Nx):
    deltar_x=prl[k][j-1]-2*prl[k][j]+prl[k][j+1]
    deltar_y=prl[k-1][j]-2*prl[k][j]+prl[k+1][j]
    deltai_x=pim[k][j-1]-2*pim[k][j]+pim[k][j+1]
    deltai_y=pim[k-1][j]-2*pim[k][j]+pim[k+1][j]
    ke+=prl[k][j]*(deltar_x+deltar_y)+pim[k][j]*(deltai_x+deltai_y)
```

```
KE=-J2eV*((hbar/del_x)**2/(2*melec))*ke

#C'alculo de energ'ia potencial y probabilidad
pe=0
proba=0
for k in range(1,Ny+1): #Va de 1 a Ny
  for j in range(1,Nx+1):
    pe+=(prl[k][j]**2+pim[k][j]**2)*V[k][j]
    proba+=prl[k][j]**2+pim[k][j]**2

PE=pe*J2eV
Etot=KE+PE
print("KE={}".format(KE))
print("PE={} eV".format(PE))
print("Psi^2 final={}".format(proba))
print("Etot={} eV".format(Etot))
```

II.2. Programa 2D en forma serial en C++

```
//Programa para para simular la absorci'on de un pulso en C++ serial
#include "../common/book.h"
#include <math.h>

//----Definir unidades fijas-----------------------
#define L 120e-9
#define del_x 0.4e-9
#define m0 9.1e-31
#define meff 0.067 //Masa efectiva del GaAs
#define meffpozo 0.088 //Masa efectiva del Al.3Ga.7As
#define ecoul 1.6e-19 //Carga del electr'on en Coulomb
```

```
#define epsz 8.85e-19 //Dielectric del espacio libre
#define eV2J 1.6e-19 //Factor de conversi'on de energ'ia
#define hbar 1.054e-34
#define E_inicial 0.05 //Energ'ia aproximada del pulso inicial en eV
#define picoseg 0

int main( void ) {
   float pi;
   pi=atan(1)*4;
   int Nx, Ny, nc_x, nc_y, lambd;
   Nx=round(L/del_x);
   Ny=Nx;
   nc_x=int(Nx/2);
   nc_y=int(Ny/2);
   float melec,J2eV, sigma, dt, KE, PE, E_total, proba;
   melec=m0*meff;
   J2eV=1/eV2J;
   lambd=round(hbar*2*pi/(pow(E_inicial*eV2J*2*melec,0.5)*del_x));
   sigma=float(lambd);
   dt=2*melec*pow(del_x,2)/(8*hbar);

   int n_step;
   n_step=int(picoseg/(dt*1e12));
   float *prl=new float[Nx*Ny], *prl_n=new float[Nx*Ny];
   float *pim=new float[Nx*Ny], *pim_n=new float[Nx*Ny];
   float *V=new float[Nx*Ny];
   float *ra=new float[Nx*Ny];
   float *gamr=new float[Nx*Ny],*gami=new float[Nx*Ny];
   float ptot, ptot2, factor_norm;
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
```

```
        V[i+Nx*j] =0.0;
        ra[i+Nx*j] =0.125;
      }
    }

    int iniciopotencial;
    iniciopotencial=int(70e-9/del_x); //Inicio potencial en 70 nm

//----------Inicializar pulso-----------
    ptot=0.0;
    for (int i=0; i<Nx; i++) {
      for (int j=0; j<Ny; j++) {
        prl[i+Nx*j]= exp(-pow(sqrt(pow(i+1-nc_x,2)+pow(j+1-nc_y,2))/sigma,2))
               *cos(2*pi*sqrt(pow(i+1-nc_x,2)+pow(j+1-nc_y,2))/lambd);
        pim[i+Nx*j]= exp(-pow(sqrt(pow(i+1-nc_x,2)+pow(j+1-nc_y,2))/sigma,2))
               *sin(2*pi*sqrt(pow(i+1-nc_x,2)+pow(j+1-nc_y,2))/lambd);
        ptot+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
      }
    }
    factor_norm=sqrt(ptot);
//----------Fin Inicio pulso-----------

//--------Normalizaci'on del pulso--------------------
  ptot2=0.0;
  for (int i=0; i<Nx; i++) { //Va de 1 a Nx.
    for (int j=0; j<Ny; j++) {
      prl[i+Nx*j]=prl[i+Nx*j]/factor_norm;
      pim[i+Nx*j]=pim[i+Nx*j]/factor_norm;
      ptot2+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
    }
  }
```

```
  printf("Psi^2 inicial=%f \n", ptot2);
 //----------Fin normalizaci'on del pulso------------

//--------------Determinaci'on PML--------------------
  int npml;
  float sigma0,sigmapml;
  npml=round(20e-9/del_x); //Equivalente a 20 nm y 50 particiones.
  sigma0=0.0005;

  for (int i=0; i<Nx; i++) {
    for (int j=0; j<Ny; j++) {
      if (i<npml && j>=i && j<=(Ny-1-i)){
        sigmapml=sigma0*powf(i-npml,2);
        gamr[i+Nx*j]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
/(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
       +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
        gami[i+Nx*j]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
/(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
        +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
      }

      else if (i<Nx && i>(Nx-npml-1) && j<=i && j>=(Ny-1-i) && j<Ny) {
        sigmapml=sigma0*powf(i+1-(Nx-npml),2);
        gamr[i+Nx*j]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
/(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
         +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
        gami[i+Nx*j]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
/(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
         +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
      }
```

```
      else if (j>(Ny-1-npml) && j>=i && j>=(Ny-1-i) && i<Nx && j<Ny) {
        sigmapml=sigma0*powf(j+1-(Ny-npml),2);
        gamr[i+Nx*j]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
/(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
        +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
        gami[i+Nx*j]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
/(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
         +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
      }

      else if (j<npml && j<=i && j<=(Nx-1-i) && i<Nx) {
        sigmapml=sigma0*powf(j-npml,2);
        gamr[i+Nx*j]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
/(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
        +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
        gami[i+Nx*j]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
/(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
         +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
      }

      else {
        gamr[i+Nx*j]=1.0;
        gami[i+Nx*j]=0.0;
      }
    }
  }
 //--------------Fin Determinaci'on PML-------------

//-------------Determinar celdas de monitoreo-------------------
  int posicion_in, posicion_out;
  posicion_in=int(62e-9/del_x)+1+Nx*(int(62e-9/del_x)+1);
```

```
  posicion_out=posicion_in;

  float *prl_in=new float[n_step],*pim_in=new float[n_step],
        *prl_out=new float[n_step], *pim_out=new float[n_step];

 //------------Fin Determinaci'on celdas de monitoreo------------

 //--------------------Ciclo temporal--------------------------

     //---------Empezar a contar el tiempo de c'omputo-----------
   cudaEvent_t inicio_ciclo, fin_ciclo;

  HANDLE_ERROR( cudaEventCreate( &inicio_ciclo ) );
  HANDLE_ERROR( cudaEventCreate( &fin_ciclo ) );
  HANDLE_ERROR( cudaEventRecord( inicio_ciclo, 0 ) );
   //---------Fin de crear eventos para tiempo c'omputo--------------
  int paso;
  for (int t=0;t<n_step;t++){ //Pasos de 1 a n_step
    paso=t;

    for (int x=1;x<(Nx-1);x++){
      for (int y=1;y<(Ny-1);y++) {
        prl_n[x+Nx*y]=prl[x+Nx*y]-ra[x+Nx*y]*(gamr[x+Nx*y]*
((pim[x-1+Nx*y]-2*pim[x+Nx*y]+pim[x+1+Nx*y])
+(pim[x+Nx*(y-1)]-2*pim[x+Nx*y]+pim[x+Nx*(y+1)]))
+gami[x+Nx*y]*((prl[x-1+Nx*y]-2*prl[x+Nx*y]+prl[x+1+Nx*y])
+(prl[x+Nx*(y-1)]-2*prl[x+Nx*y]+prl[x+Nx*(y+1)])))
+(dt/hbar)*V[x+Nx*y]*pim[x+Nx*y];
    }}

    for (int x=1;x<(Nx-1);x++){
```

```
      for (int y=1;y<(Ny-1);y++) {
        prl[x+Nx*y]=prl_n[x+Nx*y];
      }}
   prl_in[paso]=prl[posicion_in];

   for (int x=1;x<(Nx-1);x++){
     for (int y=1;y<(Ny-1);y++) {
       pim_n[x+Nx*y]=pim[x+Nx*y]+ra[x+Nx*y]*(gamr[x+Nx*y]*
((prl[x-1+Nx*y]-2*prl[x+Nx*y]+prl[x+1+Nx*y])
+(prl[x+Nx*(y-1)]-2*prl[x+Nx*y]+prl[x+Nx*(y+1)]))
-gami[x+Nx*y]*((pim[x-1+Nx*y]-2*pim[x+Nx*y]+pim[x+1+Nx*y])
+(pim[x+Nx*(y-1)]-2*pim[x+Nx*y]+pim[x+Nx*(y+1)])))
-(dt/hbar)*V[x+Nx*y]*prl[x+Nx*y];
   }}

   for (int x=1;x<(Nx-1);x++){
     for (int y=1;y<(Ny-1);y++) {
       pim[x+Nx*y]=pim_n[x+Nx*y];
   }}
   pim_in[paso]=pim[posicion_in];
 }
  //----------Impresi'on tiempo de c'omputo-------------------
  HANDLE_ERROR( cudaEventRecord( fin_ciclo, 0 ) );
  HANDLE_ERROR( cudaEventSynchronize( fin_ciclo ) );
  float elapsedTime;

  HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime,inicio_ciclo,fin_ciclo));

  printf( "Tiempo de c'omputo ciclo: %3.2f ms\n", elapsedTime );

  HANDLE_ERROR( cudaEventDestroy( inicio_ciclo ) );
```

```
  HANDLE_ERROR( cudaEventDestroy( fin_ciclo ) );

  //--------Fin impresi'on tiempo de c'omputo-------------------------

//--------------------------Fin ciclo temporal-----------------------

//-------C'alculo de valores esperados----------------------------
  KE=0;
  for (int i=1; i<(Nx-1); i++) {
    for (int j=1; j<(Ny-1); j++) {
      KE+=prl[i+Nx*j]*((prl[i-1+Nx*j]-2*prl[i+Nx*j]+prl[i+1+Nx*j])
  +(prl[i+Nx*(j-1)]-2*prl[i+Nx*j]+prl[i+Nx*(j+1)]))+pim[i+Nx*j]
*((pim[i-1+Nx*j]-2*pim[i+Nx*j]+pim[i+1+Nx*j])+(pim[i+Nx*(j-1)]
-2*pim[i+Nx*j]+pim[i+Nx*(j+1)]));
     }
   }
   KE=-KE*J2eV*pow(hbar/del_x,2)/(2*melec);

   PE=0.0;
   proba=0.0;
   E_total=0.0;
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
       PE+=(pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2))*V[i+Nx*j];
       proba+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
     }
   }
   PE=PE*J2eV;
   E_total=KE+PE;
 //---Fin de C'alculo de valores esperados-------------------
```

```
//-------Imprimir archivos-----------------------------
   FILE *archivo1 = fopen("gamma_2D.dat","w");
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
     fprintf(archivo1, "%d %d %f \n", i+1, j+1, gamr[i+Nx*j] );
   }}
   fprintf(archivo1, "\n" );
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
      fprintf(archivo1, "%d %d %f \n", i+1, j+1, gami[i+Nx*j] );
   }
   fclose ( archivo1 );

   FILE *archivo2 = fopen("psi_2D.dat","w");
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
     fprintf(archivo2, "%d %d %f \n", i+1, j+1, prl[i+Nx*j] );
   }}
   fprintf(archivo2, "\n" );
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
      fprintf(archivo2, "%d %d %f \n", i+1, j+1, pim[i+Nx*j] );
   }
     fclose ( archivo2 );
//-------Imprimir archivo para animacion-----------------
   char format[] = "anim%06d.vtk";
   char filename[sizeof format+100];
   sprintf(filename,format,n_step);
   FILE *archivo = fopen(filename,"w");

   fprintf(archivo, "# vtk DataFile Version 2.0 \n" );
```

```
      fprintf(archivo, "Volume example \n" );
      fprintf(archivo, "ASCII \n" );
      fprintf(archivo, "DATASET STRUCTURED_POINTS \n" );
      fprintf(archivo, "DIMENSIONS  300  300 1 \n" );
      fprintf(archivo, "ASPECT_RATIO 1 1 0 \n" );
      fprintf(archivo, "ORIGIN 0 0 0  \n" );
      fprintf(archivo, "POINT_DATA   90000 \n" );
      fprintf(archivo, "SCALARS volume_scalars float 1 \n" );
      fprintf(archivo, "LOOKUP_TABLE default \n" );
        for (int i=0; i<Nx*Ny; i++) {
          fprintf(archivo, "%f \n", prl[i] );
        }
        fprintf(archivo, "\n" );
   fclose ( archivo );

//----------Imprimir en pantalla----------------
  printf("KE=%f eV\n", KE);
  printf("PE=%f eV\n", PE);
  printf("E total=%f eV\n", E_total);
  printf("Psi^2 final=%f \n", proba);

  delete [] prl;
  delete [] pim;
  delete [] prl_n;
  delete [] pim_n;
  delete [] prl_in;
  delete [] prl_out;
  delete [] pim_in;
  delete [] pim_out;
  delete [] V;
  delete [] ra;
```

```
  delete [] gamr;
  delete [] gami;
  return 0;
}
```

II.3. Programa 2D paralelizado en CUDA

```
//Programa para simular la absorci'on de un pulso en CUDA
#include "../common/book.h"
#include <math.h>

//----Definir unidades fijas-----------------------
#define L 120e-9
#define del_x 0.4e-9
#define m0 9.1e-31
#define meff 0.067 //Masa efectiva del GaAs
#define meffpozo 0.088 //Masa efectiva del Al.3Ga.7As
#define ecoul 1.6e-19 //Carga del electr'on en Coulomb
#define epsz 8.85e-19 //Dielectric del espacio libre
#define eV2J 1.6e-19 //Factor de conversi'on de energ'ia
#define hbar 1.054e-34
#define E_inicial 0.05 //Energ'ia aproximada del pulso inicial en eV
#define picoseg 0

//------------Definiciones de los kernels----------------------
__global__ void pulso_inicial (float *d_prl, float *d_pim,float sigma,
           float pi, int lambd, int nc_x, int nc_y, int Nx, int Ny) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x<Nx && y<Ny) {
```

```
    d_prl[x+Nx*y] = exp(-powf(sqrt(powf((x-nc_x+1),2)
  +powf((y-nc_y+1),2))/sigma,2))*cos(2*pi*sqrt(powf((x-nc_x+1),2)
  +powf((y-nc_y+1),2))/lambd);
    d_pim[x+Nx*y] = exp(-powf(sqrt(powf((x-nc_x+1),2)
  +powf((y-nc_y+1),2))/sigma,2))*sin(2*pi*sqrt(powf((x-nc_x+1),2)
+powf((y-nc_y+1),2))/lambd);
  }
}

__global__ void pulso_normal (float *d_prl, float *d_pim,
           float factor_norm, int Nx, int Ny){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x<Nx && y<Ny) {
    d_prl[x+Nx*y]=d_prl[x+Nx*y]/factor_norm;
    d_pim[x+Nx*y]=d_pim[x+Nx*y]/factor_norm;
  }
}

__global__ void calculo_gamma (float *d_gamr, float *d_gami, int Nx,
           float sigma0, int npml, int Ny){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  float sigmapml;

  if (x<npml && y>=x && y<=(Ny-1-x)){
    sigmapml=sigma0*powf(x-npml,2);
    d_gamr[x+Nx*y]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
```

```
    d_gami[x+Nx*y]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

if (x<Nx && x>(Nx-npml-1) && y<=x && y>=(Ny-1-x) && y<Ny) {
    sigmapml=sigma0*powf(x+1-(Nx-npml),2);
    d_gamr[x+Nx*y]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    d_gami[x+Nx*y]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

  if (y>(Ny-1-npml) && y>=x && y>=(Ny-1-x) && x<Nx && y<Ny) {
    sigmapml=sigma0*powf(y+1-(Ny-npml),2);
    d_gamr[x+Nx*y]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    d_gami[x+Nx*y]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
  +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

  if (y<npml && y<=x && y<=(Nx-1-x) && x<Nx) {
    sigmapml=sigma0*powf(y-npml,2);
    d_gamr[x+Nx*y]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    d_gami[x+Nx*y]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
```

```
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

  if (x<= (Nx-npml-1) && x>=npml && y<= (Ny-npml-1) && y>=npml) {
    d_gamr[x+Nx*y]=1.0;
    d_gami[x+Nx*y]=0.0;
  }
}

__global__ void prl_nueva (float *d_prl, float *d_pim, float *d_prl_n,
float *d_pim_n, int Nx, float dt,float *d_V, float*d_ra, float *d_gamr,
float *d_gami, float *d_prl_in, float *d_pim_in, int posicion_in, int Ny){
 int x = blockIdx.x * blockDim.x + threadIdx.x;
 int y = blockIdx.y * blockDim.y + threadIdx.y;

 if (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
   d_prl_n[x+Nx*y]=d_prl[x+Nx*y]-d_ra[x+Nx*y]*(d_gamr[x+Nx*y]
 *((d_pim[x-1+Nx*y]-2*d_pim[x+Nx*y]+d_pim[x+1+Nx*y])+(d_pim[x+Nx*(y-1)]
 -2*d_pim[x+Nx*y]+d_pim[x+Nx*(y+1)]))+d_gami[x+Nx*y]*((d_prl[x-1+Nx*y]
 -2*d_prl[x+Nx*y]+d_prl[x+1+Nx*y])+(d_prl[x+Nx*(y-1)]-2*d_prl[x+Nx*y]
 +d_prl[x+Nx*(y+1)])))+(dt/hbar)*d_V[x+Nx*y]*d_pim[x+Nx*y];
         }
  }

__global__ void prl_reescribir(float *d_prl, float *d_prl_n, int Nx, int Ny){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
    d_prl[x+Nx*y]=d_prl_n[x+Nx*y];
```

```
  }
}

__global__ void pim_nueva (float *d_prl, float *d_pim, float *d_prl_n,
  float *d_pim_n, int Nx, float dt,float *d_V, float*d_ra, float *d_gamr,
float *d_gami, float *d_prl_in, float *d_pim_in, int posicion_in, int Ny){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if  (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
    d_pim_n[x+Nx*y]=d_pim[x+Nx*y]+d_ra[x+Nx*y]*(d_gamr[x+Nx*y]
*((d_prl[x-1+Nx*y]-2*d_prl[x+Nx*y]+d_prl[x+1+Nx*y])+(d_prl[x+Nx*(y-1)]
-2*d_prl[x+Nx*y]+d_prl[x+Nx*(y+1)]))-d_gami[x+Nx*y]*((d_pim[x-1+Nx*y]
-2*d_pim[x+Nx*y]+d_pim[x+1+Nx*y])+(d_pim[x+Nx*(y-1)]-2*d_pim[x+Nx*y]
+d_pim[x+Nx*(y+1)]))))-(dt/hbar)*d_V[x+Nx*y]*d_prl[x+Nx*y];
  }
 }

 __global__ void pim_reescribir (float *d_pim, float *d_pim_n, int Nx, int Ny) {
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;

   if (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
     d_pim[x+Nx*y]=d_pim_n[x+Nx*y];
   }
}

__global__ void monitores (float *d_prl, float *d_pim, float *d_prl_in,
            float *d_pim_in, int posicion_in, int paso) {
  d_prl_in[paso]=d_prl[posicion_in];
  d_pim_in[paso]=d_pim[posicion_in];
```

```
}

//--------Fin de definiciones de los kernels--------------

//----------Inicio del programa-------------------------
int main( void ) {
   float pi;
   pi=atan(1)*4;
   int Nx, Ny, nc_x, nc_y, lambd;
   Nx=round(L/del_x);
   Ny=Nx;
   nc_x=int(Nx/2);
   nc_y=int(Ny/2);
   float melec,J2eV, sigma, dt, KE, PE, E_total, proba;
   melec=m0*meff;
   J2eV=1/eV2J;
   lambd=round(hbar*2*pi/(pow(E_inicial*eV2J*2*melec,0.5)*del_x));
   sigma=float(lambd);
   dt=2*melec*pow(del_x,2)/(8*hbar);

 int n_step;
   n_step=int(picoseg/(dt*1e12));
   float *prl=new float[Nx*Ny], *prl_n=new float[Nx*Ny];
   float *pim=new float[Nx*Ny], *pim_n=new float[Nx*Ny];
   float *V=new float[Nx*Ny];
   float *ra=new float[Nx*Ny];
   float *gamr=new float[Nx*Ny],*gami=new float[Nx*Ny];
   float ptot, ptot2, factor_norm;
   float *d_prl,*d_pim,*d_V, *d_ra, *d_prl_n, *d_pim_n;
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
```

```
        V[i+Nx*j] =0.0;
        ra[i+Nx*j] =0.125;
      }
    }

    int iniciopotencial;
    iniciopotencial=int(70e-9/del_x); //Inicio potencial en 70 nm

 //Definir tama'no de la malla.
    dim3 grid(16,16,1);
    dim3 block(32,32,1);
 //Fin definici'on tama'no de la malla

//----------Inicializar pulso-----------
   cudaMalloc( (void**)&d_prl, Nx*Ny * sizeof(float)  );
   cudaMalloc( (void**)&d_pim, Nx*Ny * sizeof(float)  );

   pulso_inicial<<<grid,block>>>(d_prl,d_pim,sigma,pi,lambd,nc_x,nc_y,Nx,Ny);

   HANDLE_ERROR(cudaMemcpy(prl,d_prl,Nx*Ny*sizeof(float),
              cudaMemcpyDeviceToHost));
   HANDLE_ERROR(cudaMemcpy(pim,d_pim,Nx*Ny*sizeof(float),
              cudaMemcpyDeviceToHost));

ptot=0.0;
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
       ptot+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
     }
   }
factor_norm=sqrt(ptot);
```

```
//----------Fin Inicio pulso-----------
//--------Normalizaci'on del pulso---------------------
  cudaMalloc( (void**)&d_V, Nx*Ny * sizeof(float)  );
  cudaMalloc( (void**)&d_ra, Nx*Ny * sizeof(float)  );

  pulso_normal<<<grid,block>>>(d_prl,d_pim,factor_norm, Nx, Ny);

  HANDLE_ERROR(cudaMemcpy(prl,d_prl,Nx*Ny*sizeof(float),
              cudaMemcpyDeviceToHost));
  HANDLE_ERROR(cudaMemcpy(pim,d_pim,Nx*Ny*sizeof(float),
              cudaMemcpyDeviceToHost));

  //Verificar normalizaci'on
  ptot2=0.0;
  for (int i=0; i<Nx; i++) {
    for (int j=0; j<Ny; j++) {
      ptot2+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
    }
  }
  printf("Psi^2 inicial=%f \n", ptot2);
 //----------Fin normalizaci'on del pulso------------

//--------------Determinaci'on PML--------------------
  int npml;
  float sigma0;
  float *d_gamr,*d_gami;
  npml=round(20e-9/del_x); //Equivalente a 20 nm y 50 particiones
  sigma0=0.0005;

  cudaMalloc( (void**)&d_gamr, Nx*Ny * sizeof(float)  );
  cudaMalloc( (void**)&d_gami, Nx*Ny * sizeof(float)  );
```

```
  calculo_gamma<<<grid,block>>>(d_gamr, d_gami, Nx, sigma0, npml, Ny);

  HANDLE_ERROR(cudaMemcpy(gamr,d_gamr,Nx*Ny*sizeof(float),
              cudaMemcpyDeviceToHost));
  HANDLE_ERROR(cudaMemcpy(gami,d_gami,Nx*Ny*sizeof(float),
              cudaMemcpyDeviceToHost));

//--------------Fin Determinaci'on PML-------------

//-------------Determinaci'on celdas de monitoreo-------------------
  int posicion_in, posicion_out;
  posicion_in=int(62e-9/del_x)+1+Nx*(int(62e-9/del_x)+1);
  posicion_out=posicion_in; //Coordenada(62 nm,62 nm)

  float *prl_in=new float[n_step],*pim_in=new float[n_step],
      *prl_out=new float[n_step], *pim_out=new float[n_step];
  float *d_prl_in, *d_pim_in, *d_prl_out, *d_pim_out;

  cudaMalloc( (void**)&d_prl_in, n_step * sizeof(float)  );
  cudaMalloc( (void**)&d_pim_in, n_step * sizeof(float)  );
  cudaMalloc( (void**)&d_prl_out, n_step * sizeof(float)  );
  cudaMalloc( (void**)&d_pim_out, n_step * sizeof(float)  );
//------------Fin Determinaci'on celdas de monitoreo----------------

//-------------------------Ciclo temporal----------------------
  cudaMalloc( (void**)&d_prl_n, Nx*Ny * sizeof(float)  );
  cudaMalloc( (void**)&d_pim_n, Nx*Ny * sizeof(float)  );

  HANDLE_ERROR(cudaMemcpy(d_V,V,Nx*Ny*sizeof(float),
              cudaMemcpyHostToDevice));
```

```
 HANDLE_ERROR(cudaMemcpy(d_ra,ra,Nx*Ny*sizeof(float),
            cudaMemcpyHostToDevice));
 HANDLE_ERROR(cudaMemcpy(d_gamr,gamr,Nx*Ny*sizeof(float),
            cudaMemcpyHostToDevice));
 HANDLE_ERROR(cudaMemcpy(d_gami,gami,Nx*Ny*sizeof(float),
            cudaMemcpyHostToDevice));

 HANDLE_ERROR(cudaMemcpy(d_prl,prl,Nx*Ny*sizeof(float),
            cudaMemcpyHostToDevice));
 HANDLE_ERROR(cudaMemcpy(d_pim,pim,Nx*Ny*sizeof(float),
            cudaMemcpyHostToDevice));

 //---------Empezar a contar el tiempo de c'omputo----------------
 cudaEvent_t inicio_ciclo, fin_ciclo;

 HANDLE_ERROR( cudaEventCreate( &inicio_ciclo ) );
 HANDLE_ERROR( cudaEventCreate( &fin_ciclo ) );
 HANDLE_ERROR( cudaEventRecord( inicio_ciclo, 0 ) );

//---------Fin de crear eventos para tiempo c'omputo--------------
 int paso;
 for (int t=0;t<n_step;t++){
   paso=t;

   prl_nueva<<<grid,block>>>(d_prl, d_pim, d_prl_n, d_pim_n, Nx, dt, d_V,
          d_ra, d_gamr, d_gami, d_prl_in, d_pim_in, posicion_in, Ny);
   prl_reescribir<<<grid,block>>>(d_prl, d_prl_n, Nx, Ny);

   pim_nueva<<<grid,block>>>(d_prl, d_pim, d_prl_n, d_pim_n, Nx, dt, d_V,
          d_ra, d_gamr, d_gami, d_prl_in, d_pim_in, posicion_in, Ny);
   pim_reescribir<<<grid,block>>>(d_pim, d_pim_n, Nx, Ny);
```

```
    monitores<<<grid,block>>>(d_prl, d_pim, d_prl_in, d_pim_in, posicion_in,
            paso);
   }
  //----------Impresi'on tiempo de c'omputo------------------
  HANDLE_ERROR( cudaEventRecord( fin_ciclo, 0 ) );
  HANDLE_ERROR( cudaEventSynchronize( fin_ciclo ) );

  float elapsedTime;

  HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime,inicio_ciclo,fin_ciclo));

  printf( "Tiempo de c'omputo ciclo: %3.2f ms\n", elapsedTime );

  HANDLE_ERROR( cudaEventDestroy( inicio_ciclo ) );
  HANDLE_ERROR( cudaEventDestroy( fin_ciclo ) );

 //--------Fin impresi'on tiempo de c'omputo-----------------------

  HANDLE_ERROR(cudaMemcpy(prl,d_prl,Nx*Ny*sizeof(float),
            cudaMemcpyDeviceToHost));
  HANDLE_ERROR(cudaMemcpy(pim,d_pim,Nx*Ny*sizeof(float),
            cudaMemcpyDeviceToHost));

  HANDLE_ERROR(cudaMemcpy(prl_in,d_prl_in,n_step*sizeof(float),
            cudaMemcpyDeviceToHost));
  HANDLE_ERROR(cudaMemcpy(pim_in,d_pim_in,n_step*sizeof(float),
            cudaMemcpyDeviceToHost));
//--------------------Fin ciclo temporal--------------------------

//-------C'alculo de valores esperados---------------------------
```

```
  KE=0;
  for (int i=1; i<(Nx-1); i++) {
    for (int j=1; j<(Ny-1); j++) {
      KE+=prl[i+Nx*j]*((prl[i-1+Nx*j]-2*prl[i+Nx*j]+prl[i+1+Nx*j])
    +(prl[i+Nx*(j-1)]-2*prl[i+Nx*j]+prl[i+Nx*(j+1)]))+pim[i+Nx*j]
  *((pim[i-1+Nx*j]-2*pim[i+Nx*j]+pim[i+1+Nx*j])+(pim[i+Nx*(j-1)]
  -2*pim[i+Nx*j]+pim[i+Nx*(j+1)]));
     }
   }
   KE=-KE*J2eV*pow(hbar/del_x,2)/(2*melec);

   PE=0.0;
   proba=0.0;
   E_total=0.0;
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
       PE+=(pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2))*V[i+Nx*j];
       proba+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
     }
   }
   PE=PE*J2eV;
   E_total=KE+PE;
 //---Fin de C'alculo de valores esperados-------------------

//-------Imprimir archivos----------------------------

   FILE *archivo1 = fopen("gamma_2D.dat","w");
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
     fprintf(archivo1, "%d %d %f \n", i+1, j+1, gamr[i+Nx*j] );
   }}
```

```
   fprintf(archivo1, "\n" );
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
      fprintf(archivo1, "%d %d %f \n", i+1, j+1, gami[i+Nx*j] );
   }
   fclose ( archivo1 );

   FILE *archivo2 = fopen("psi_2D.dat","w");
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
     fprintf(archivo2, "%d %d %f \n", i+1, j+1, prl[i+Nx*j] );
   }}
   fprintf(archivo2, "\n" );
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
      fprintf(archivo2, "%d %d %f \n", i+1, j+1, pim[i+Nx*j] );
   }
     fclose ( archivo2 );
//-------Imprimir archivo para animaci'on------------------
   char format[] = "anim%06d.vtk";
   char filename[sizeof format+100];
   sprintf(filename,format,n_step);
   FILE *archivo = fopen(filename,"w");

   fprintf(archivo, "# vtk DataFile Version 2.0 \n" );
     fprintf(archivo, "Volume example \n" );
     fprintf(archivo, "ASCII \n" );
     fprintf(archivo, "DATASET STRUCTURED_POINTS \n" );
     fprintf(archivo, "DIMENSIONS  300  300 1 \n" );
     fprintf(archivo, "ASPECT_RATIO 1 1 0 \n" );
     fprintf(archivo, "ORIGIN 0 0 0  \n" );
```

```
      fprintf(archivo, "POINT_DATA   90000 \n" );
      fprintf(archivo, "SCALARS volume_scalars float 1 \n" );
      fprintf(archivo, "LOOKUP_TABLE default \n" );
        for (int i=0; i<Nx*Ny; i++) {
          fprintf(archivo, "%f \n", prl[i] );
        }
        fprintf(archivo, "\n" );
    fclose ( archivo );

//----------Imprimir en pantalla----------------
  printf("KE=%f eV\n", KE);
  printf("PE=%f eV\n", PE);
  printf("E total=%f eV\n", E_total);
  printf("Psi^2 final=%f \n", proba);

  delete [] prl;
  delete [] pim;
  delete [] prl_n;
  delete [] pim_n;
  delete [] prl_in;
  delete [] prl_out;
  delete [] pim_in;
  delete [] pim_out;
  delete [] V;
  delete [] ra;
  delete [] gamr;
  delete [] gami;
  cudaFree( d_prl );
  cudaFree( d_pim );
  cudaFree( d_prl_n );
  cudaFree( d_pim_n );
```

```
  cudaFree( d_prl_in );
  cudaFree( d_prl_out );
  cudaFree( d_pim_in );
  cudaFree( d_pim_out );
  cudaFree( d_V );
  cudaFree( d_ra );
  cudaFree( d_gamr );
  cudaFree( d_gami );
  return 0;
}
```

II.4. Programa paralelizado en CUDA para un potencial circular

Este código utiliza Unified Memory por lo que requiere correr con GPU que soporten este modelo.

```
//Programa para calcular el scattering en un potencial circular.
#include "../common/book.h"
#include <math.h>

//----Definir unidades fijas-----------------------
#define L 240e-9//120e-9 //Para un sistema de 600X600 celdas
#define del_x 0.4e-9
#define m0 9.1e-31
#define meff 0.067 //Masa efectiva del GaAs
#define meffpozo 0.088 //Masa efectiva del Al.3Ga.7As
#define ecoul 1.6e-19 //Carga del electr'on en Coulomb
#define epsz 8.85e-19 //Dielectric del espacio libre
#define eV2J 1.6e-19 //Factor de conversi'on de energ'ia
#define hbar 1.054e-34
```

```
#define E_inicial 0.06 //Energ'ia aproximada del pulso inicial en eV
#define picoseg 1

//------------Definiciones de los kernels----------------------
__global__ void pulso_inicial (float *prl, float *pim, float sigma,
           float pi, int lambd, int nc_x, int nc_y, int Nx, int Ny) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x<Nx && y<Ny) {
    prl[x+Nx*y] = exp(-powf(sqrt(powf((x-nc_x+1),2)
  +powf((y-nc_y+1),2))/sigma,2))*cos(2*pi*sqrt(powf((x-nc_x+1),2)
  +powf((y-nc_y+1),2))/lambd);
    pim[x+Nx*y] = exp(-powf(sqrt(powf((x-nc_x+1),2)
  +powf((y-nc_y+1),2))/sigma,2))*sin(2*pi*sqrt(powf((x-nc_x+1),2)
+powf((y-nc_y+1),2))/lambd);
  }
}

__global__ void pulso_normal (float *prl, float *pim,
           float factor_norm, int Nx, int Ny){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x<Nx && y<Ny) {
    prl[x+Nx*y]=prl[x+Nx*y]/factor_norm;
    pim[x+Nx*y]=pim[x+Nx*y]/factor_norm;
  }
}

__global__ void calculo_gamma (float *gamr, float *gami, int Nx,
```

```
          float sigma0, int npml, int Ny){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  float sigmapml;

  if (x<npml && y>=x && y<=(Ny-1-x)){
    sigmapml=sigma0*powf(x-npml,2);

    gamr[x+Nx*y]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    gami[x+Nx*y]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

if (x<Nx && x>(Nx-npml-1) && y<=x && y>=(Ny-1-x) && y<Ny) {
    sigmapml=sigma0*powf(x+1-(Nx-npml),2);

    gamr[x+Nx*y]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    gami[x+Nx*y]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

  if (y>(Ny-1-npml) && y>=x && y>=(Ny-1-x) && x<Nx && y<Ny) {
    sigmapml=sigma0*powf(y+1-(Ny-npml),2);

    gamr[x+Nx*y]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
```

```
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    gami[x+Nx*y]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
  +pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

  if (y<npml && y<=x && y<=(Nx-1-x) && x<Nx) {
    sigmapml=sigma0*powf(y-npml,2);

    gamr[x+Nx*y]=(pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)), 2));
    gami[x+Nx*y]=-(2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5))
  /(pow((pow(sigmapml/powf(2,0.5)+1,2)-pow(sigmapml/powf(2,0.5),2)),2)
+pow((2*(sigmapml/powf(2,0.5)+1)*sigmapml/powf(2,0.5)),2));
  }

  if (x<= (Nx-npml-1) && x>=npml && y<= (Ny-npml-1) && y>=npml) {
    gamr[x+Nx*y]=1.0;
    gami[x+Nx*y]=0.0;
  }
}

//Descomentar if en kernels si es potencial infinito
__global__ void prl_nueva (float *prl, float *pim, float *prl_n, int Nx,
float dt, float *V, float*ra, float *gamr, float *gami, int Ny,
float radiopotencial, int iniciopotencial, int nc_y) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
```

```
 // if (x< (Nx-1) && x>0 && y<(Ny-1) && y>0 &&
//powf(powf(x-(iniciopotencial-Nx*nc_y),2)+powf(y-nc_y,2),0.5)>radiopotencial){
  if  (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
   prl_n[x+Nx*y]=d_prl[x+Nx*y]-ra[x+Nx*y]*(gamr[x+Nx*y]
 *((pim[x-1+Nx*y]-2*pim[x+Nx*y]+pim[x+1+Nx*y])+(pim[x+Nx*(y-1)]
 -2*pim[x+Nx*y]+pim[x+Nx*(y+1)]))+gami[x+Nx*y]*((prl[x-1+Nx*y]
 -2*prl[x+Nx*y]+prl[x+1+Nx*y])+(prl[x+Nx*(y-1)]-2*prl[x+Nx*y]
 +prl[x+Nx*(y+1)]))))+(dt/hbar)*V[x+Nx*y]*pim[x+Nx*y];
   }
 }

__global__ void prl_reescribir (float *prl, float *pim, float *prl_n, int Nx,
float dt, int Ny, float radiopotencial, int iniciopotencial, int nc_y) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

 // if (x< (Nx-1) && x>0 && y<(Ny-1) && y>0 &&
//  powf(powf(x-(iniciopotencial-Nx*nc_y),2)+powf(y-nc_y,2),0.5)>radiopotencial){
  if  (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
    prl[x+Nx*y]=prl_n[x+Nx*y];
  }
}

__global__ void pim_nueva (float *prl, float *pim, float *pim_n, int Nx,
float dt,float *V, float*ra, float *gamr, float *gami, int Ny,
float radiopotencial, int iniciopotencial, int nc_y) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

//  if  (x< (Nx-1) && x>0 && y<(Ny-1) && y>0 &&
//  powf(powf(x-(iniciopotencial-Nx*nc_y),2)+powf(y-nc_y,2),0.5)>radiopotencial){
```

```
  if  (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
    pim_n[x+Nx*y]=pim[x+Nx*y]+ra[x+Nx*y]*(gamr[x+Nx*y]*((prl[x-1+Nx*y]-
  2*prl[x+Nx*y]+prl[x+1+Nx*y])+(prl[x+Nx*(y-1)]-2*prl[x+Nx*y]
  +prl[x+Nx*(y+1)]))-gami[x+Nx*y]*((pim[x-1+Nx*y]-2*pim[x+Nx*y]
+pim[x+1+Nx*y])+(pim[x+Nx*(y-1)]-2*pim[x+Nx*y]+pim[x+Nx*(y+1)])))
-(dt/hbar)*V[x+Nx*y]*prl[x+Nx*y];
  }
}

__global__ void pim_reescribir (float *prl, float *pim, float *pim_n, int Nx,
float dt, int Ny, float radiopotencial, int iniciopotencial, int nc_y) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

//  if (x< (Nx-1) && x>0 && y<(Ny-1) && y>0 &&
//  powf(powf(x-(iniciopotencial-Nx*nc_y),2)+powf(y-nc_y,2),0.5)>radiopotencial){
  if  (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
    pim[x+Nx*y]=pim_n[x+Nx*y];
  }
 }

__global__ void definir_potencial (float *V, float *ra, float pi, float dt,
float radiopotencial,int Nx,int Ny,int iniciopotencial,int nc_y,
float *prl,float *pim){
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (powf(powf(x-(iniciopotencial-Nx*nc_y),2)+powf(y-nc_y,2),0.5)
                                          <radiopotencial) {
    V[x+Nx*y]=0.1*eV2J;
    ra[x+Nx*y]=(0.5*hbar*dt)/(m0*meffpozo*pow(del_x,2.0));
```

```
  }
}

__global__ void monitoreo_out (float *prl, float *pim, int Nx,
float *sum_out_r, float *sum_out_i, int iniciopotencial, int paso, int Ny,
int n_step, float pi, int lambd, int nc_y, float dt) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if  (x< (Nx-1) && x>0 && y<(Ny-1) && y>0) {
    sum_out_r[x+Nx*y]+=prl[x+Nx*y]*cos((E_inicial*eV2J)*(paso+1)*dt/hbar)
  -pim[x+Nx*y]*sin((E_inicial*eV2J)*(paso+1)*dt/hbar);
    sum_out_i[x+Nx*y]+=prl[x+Nx*y]*sin((E_inicial*eV2J)*(paso+1)*dt/hbar)
  +pim[x+Nx*y]*cos((E_inicial*eV2J)*(paso+1)*dt/hbar);
  }
}

__global__ void transformada_fourier_out (float *E_out, int n_step,
float dt, float *sum_out_r, float *sum_out_i,  int Nx, int Ny) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  if (x< (Nx-1) && x>0 && y<(Ny-1) && y>0){
    E_out[x+Nx*y]=pow(sum_out_r[x+Nx*y],2)+pow(sum_out_i[x+Nx*y],2);
  }
}
//--------Fin de definiciones de los kernels--------------
//----------Inicio del programa--------------
  int main( void ) {
  float pi;
  pi=atan(1)*4;
```

```
int Nx, Ny, nc_x, nc_y, lambd;
Nx=round(L/del_x);
Ny=Nx;
nc_x=int(300/2);
nc_y=int(Ny/2);
float melec,J2eV, sigma, dt, KE, PE, E_total, proba, radiopotencial;
melec=m0*meff;
J2eV=1/eV2J;
lambd=round(hbar*2*pi/(pow(E_inicial*eV2J*2*melec,0.5)*del_x));
sigma=float(lambd);
dt=2*melec*pow(del_x,2)/(8*hbar);
radiopotencial=1*lambd/10;//lambd/10, lambd/2

int n_step;
n_step=int(picoseg/(dt*1e12));
float *prl, *pim;
float *prl_n, *pim_n;
float *V, *ra;
float *gamr,*gami;
float ptot, ptot2, factor_norm;
cudaMallocManaged(&V, Nx*Ny * sizeof(float));
cudaMallocManaged(&ra, Nx*Ny * sizeof(float));
cudaMallocManaged(&gamr, Nx*Ny * sizeof(float));
cudaMallocManaged(&gami, Nx*Ny * sizeof(float));
for (int i=0; i<Nx; i++) {
  for (int j=0; j<Ny; j++) {
    V[i+Nx*j] =0.0;
    ra[i+Nx*j] =0.125;
    gamr[i+Nx*j]=1.0;
    gami[i+Nx*j]=0.0;
   }
```

```
  }

  int iniciopotencial;
  iniciopotencial=int(120e-9/del_x)+Nx*nc_y;
 //Definir tama'no de la malla.
   dim3 grid(256,256,1);
   dim3 block(32,32,1);
 //Fin definici'on tama'no de la malla

//----------Inicializar pulso-----------
  cudaMallocManaged(&prl, Nx*Ny * sizeof(float));
  cudaMallocManaged(&pim, Nx*Ny * sizeof(float));
  pulso_inicial<<<grid,block>>>(prl,pim,sigma,pi,lambd,nc_x,nc_y,Nx,Ny);
  cudaDeviceSynchronize();

  ptot=0.0;
  for (int i=0; i<Nx; i++) {
    for (int j=0; j<Ny; j++) {
      ptot+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
    }
  }
  factor_norm=pow(ptot,.5);
//----------Fin Inicio pulso-----------

//--------Normalizaci'on del pulso---------------------

  pulso_normal<<<grid,block>>>(prl,pim,factor_norm, Nx, Ny);
  cudaDeviceSynchronize();

//Verificar normalizaci'on
  ptot2=0.0;
```

```
  for (int i=0; i<Nx; i++) {
    for (int j=0; j<Ny; j++) {
      ptot2+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
    }
  }
  printf("Psi^2 inicial=%f \n", ptot2);
//----------Fin normalizaci'on del pulso------------

//--------------Determinaci'on stretching coordinate--------------------
  int npml;
  float sigma0;
  npml=round(20e-9/del_x); //Equivalente a 20 nm y 50 particiones.
  sigma0=0.0005;

  calculo_gamma<<<grid,block>>>(gamr, gami, Nx, sigma0, npml, Ny);
  cudaDeviceSynchronize();

//--------------Fin Determinaci'on stretching coordinate-------------
  int paso;

  float *sum_out_r, *sum_out_i;

  cudaMallocManaged(&prl_n, Nx*Ny * sizeof(float));
  cudaMallocManaged(&pim_n, Nx*Ny * sizeof(float));

//----------Definir zona del potencial----------

  definir_potencial<<<grid,block>>>(V, ra, pi, dt, radiopotencial, Nx,
                                    Ny, iniciopotencial, nc_y, prl, pim);
  cudaDeviceSynchronize();
```

```
//--------Fin Definici'on zonas del potencial----

  cudaMallocManaged(&sum_out_r, Nx*Ny * sizeof(float));
  cudaMallocManaged(&sum_out_i, Nx*Ny * sizeof(float));
//---------Ciclo temporal-----------
  for (int t=0;t<n_step;t++){
    paso=t;

    prl_nueva<<<grid,block>>>(prl, pim, prl_n, Nx, dt, V, ra, gamr, gami, Ny,
                        radiopotencial, iniciopotencial, nc_y);
    prl_reescribir<<<grid,block>>>(prl, pim, prl_n, Nx, dt, Ny,radiopotencial,
                        iniciopotencial, nc_y);
    pim_nueva<<<grid,block>>>(prl, pim, pim_n, Nx, dt, V, ra, gamr, gami, Ny,
                        radiopotencial, iniciopotencial, nc_y);
    pim_reescribir<<<grid,block>>>(prl, pim, pim_n, Nx, dt, Ny,radiopotencial,
                        iniciopotencial, nc_y);
    monitoreo_out<<<grid,block>>>(prl, pim, Nx, sum_out_r, sum_out_i,
        iniciopotencial, paso, Ny, n_step, pi, radiopotencial, nc_y, dt);

  //Descomentar secci'on para escribir archivos para animaci'on en Paraview
    // if ( t % 100 == 0 ) {
    //   cudaDeviceSynchronize();

    //   char format[] = "anim%06d.vtk";
    //   char filename[sizeof format+100];
    //   sprintf(filename,format,t);
    //   FILE *archivo = fopen(filename,"w");
    //   fprintf(archivo, "# vtk DataFile Version 2.0 \n" );
    //   fprintf(archivo, "Volume example \n" );
    //   fprintf(archivo, "ASCII \n" );
    //   fprintf(archivo, "DATASET STRUCTURED_POINTS \n" );
```

```
    //   fprintf(archivo, "DIMENSIONS  %d  %d 1 \n", Nx, Ny);
    //   fprintf(archivo, "ASPECT_RATIO 1 1 0 \n" );
    //   fprintf(archivo, "ORIGIN 0 0 0  \n" );
    //   fprintf(archivo, "POINT_DATA   %d \n", Nx*Ny);
    //   fprintf(archivo, "SCALARS volume_scalars float 1 \n" );
    //   fprintf(archivo, "LOOKUP_TABLE default \n" );
    //   for (int i=0; i<(Nx*Ny); i++) {
    //     fprintf(archivo, "%f \n", prl[i] );
    //   }
    //   fprintf(archivo, "\n" );
    //   fclose ( archivo );
    //   }
  }
  cudaDeviceSynchronize();
 //-----------------Fin ciclo temporal------

//-----------Calcular FT para datos salida------------------
  float *E_out;

  cudaMallocManaged(&E_out, Nx*Ny * sizeof(float));

  if (n_step>0){
    transformada_fourier_out<<<grid,block>>>(E_out, n_step, dt, sum_out_r,
                                        sum_out_i, Nx, Ny);
  }
  cudaDeviceSynchronize();
//---------Fin c'alculo FT para datos salida-----------------

//----------Obtener integral de E_out----------------------
  float maxaltura;
```

```
  FILE *archivoimagen = fopen("energias.dat","w");

  maxaltura=0;
  for (int x=0; x<Nx; x++) {
    for(int y=0; y<Ny; y++){
       maxaltura+=E_out[x+Nx*y];
    }
   }

  for (int x=0; x<Nx; x++) {
    for(int y=0; y<Ny; y++){
      fprintf(archivoimagen, "%d %d %f\n", x, y,
         E_out[x+Nx*y]/sqrt(maxaltura));
     }
  }
  fclose ( archivoimagen );
//----------Fin obtenci'on m'aximo-------------------------------

//-------C'alculo de valores esperados------------
  KE=0;
  for (int i=1; i<(Nx-1); i++) {
    for (int j=1; j<(Ny-1); j++) {
      KE+=prl[i+Nx*j]*((prl[i-1+Nx*j]-2*prl[i+Nx*j]
+prl[i+1+Nx*j])+(prl[i+Nx*(j-1)]-2*prl[i+Nx*j]
+prl[i+Nx*(j+1)]))+pim[i+Nx*j]*((pim[i-1+Nx*j]
-2*pim[i+Nx*j]+pim[i+1+Nx*j])+(pim[i+Nx*(j-1)]
-2*pim[i+Nx*j]+pim[i+Nx*(j+1)]));
     }
  }
  KE=-KE*J2eV*pow(hbar/del_x,2)/(2*melec);
```

```
  PE=0.0;
  proba=0.0;
  E_total=0.0;
  for (int i=0; i<Nx; i++) {
    for (int j=0; j<Ny; j++) {
      PE+=(pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2))*V[i+Nx*j];
      proba+=pow(prl[i+Nx*j],2)+pow(pim[i+Nx*j],2);
    }
  }
  PE=PE*J2eV;
  E_total=KE+PE;
//---Fin de C'alculo de valores esperados

   FILE *archivo2 = fopen("psi_2D.dat","w");
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
     fprintf(archivo2, "%d %d %f \n", i+1, j+1, prl[i+Nx*j] );
   }}
   fprintf(archivo2, "\n" );
   for (int i=0; i<Nx; i++) {
     for (int j=0; j<Ny; j++) {
      fprintf(archivo2, "%d %d %f \n", i+1, j+1, pim[i+Nx*j] );
   }
     fclose ( archivo2 );

 //Impresi'on en pantalla
   printf("KE=%f eV\n", KE);
   printf("PE=%f eV\n", PE);
   printf("E total=%f eV\n", E_total);
   printf("Psi^2 final=%f \n", proba);
```

```
cudaFree( prl );
  cudaFree( pim );
  cudaFree( prl_n );
  cudaFree( pim_n );
  cudaFree( V );
  cudaFree( ra );
  cudaFree( gamr );
  cudaFree( gami );
  cudaFree( sum_out_r );
  cudaFree( sum_out_i );
  cudaFree( E_out );
  return 0;
}
```

Bibliografía

Antoine, X., Lorin, E., y Tang, Q. (2017). A friendly review of absorbing boundary conditions and perfectly matched layers for classical and relativistic quantum waves equations. *Molecular Physics*, *115*(15-16), 1861–1879. doi: 10.1080/00268976.2017.1290834

Becerril, R., Guzmán, F. S., Rendón-Romero, A., y Valdez-Alvarado, S. (2008). Solving the time-dependent Schrödinger equation using finite difference methods. *Revista Mexicana de Fisica E*, *54*(2), 120–132.

Berenger, J.-P. (1994). A Perfectly Matched Layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, *114*(2), 185–200.

Biegel, B. A. (1997). *Quantum electronic device simulation* (Tesis Doctoral no publicada). Stanford University. (Department of electrical engineering)

Bohren, C. F., y Huffman, D. R. (1998). *Absorption and scattering of light by small particles*. John Wiley & Sons.

Bramble, J. H., y Pasciak, J. E. (2013). Analysis of a Cartesian PML approximation to acoustic scattering problems in R2 and R3. *Journal of Computational and Applied Mathematics*, *247*(1), 209–230. doi: 10.1016/j.cam.2012.12.022

Cheng, J., Grossman, M., y McKercher, T. (2014). *Professional CUDA C Programming*. John Wiley & Sons, Inc. doi: 10.1017/CBO9781107415324.004

Chew, W. C., y Weedon, W. H. (1994). A 3D perfectly matched medium from modified Maxwell's equations with stretched coordinates. *Microwave and Optical Technology Letters*, *7*(13), 599–604. doi: 10.1002/mop.4650071304

Chi, J., Liu, F., Weber, E., Li, Y., y Crozier, S. (2011). GPU-accelerated FDTD modeling of radio-frequency field-tissue interactions in high-field MRI. *IEEE Transactions on Biomedical Engineering*, *58*(6), 1789–1796. doi: 10.1109/TBME.2011.2116020

Dai, W., Li, G., Nassar, R., y Su, S. (2005). On the stability of the FDTD method for solving a time-dependent Schrödinger equation. *Numerical Methods for Partial Differential Equations*, *21*(6), 1140–1154. doi: 10.1002/num.20082

Datta, S. (2005). *Quantum transport: Atom to transistor*. Cambridge University Press. doi: 10.1017/CBO9781139164313

De Donno, D., Esposito, A., Tarricone, L., y Catarinucci, L. (2010). Introduction to GPU computing and CUDA programming: A case study on FDTD. *IEEE Antennas and Propagation Magazine*, *52*(3), 116–122. doi: 10.1109/MAP.2010.5586593

De la Peña, L. (2014). *Introducción a la mecánica cuántica*. Fondo de Cultura Económica. (Ediciones científicas universitarias)

Dixon, D. A., Feller, D., y Peterson, K. A. (2012). Chapter one - a practical guide to reliable first principles computational thermochemistry predictions across the periodic table. En R. A. Wheeler (Ed.), *Annual reports in computational chemistry* (Vol. 8, p. 1 - 28). Elsevier. doi: https://doi.org/10.1016/B978-0-444-59440-2.00001-6

Dziubak, T., y Matulewski, J. (2012). An object-oriented implementation of a solver of the time-dependent Schrödinger equation using the CUDA technology. *Computer Physics Communications*, *183*(3), 800–812. doi: 10.1016/j.cpc.2011.11.026

Fang, Y. L. L. (2019). FDTD: Solving 1+1D delay PDE in parallel. *Computer Physics Communications*, *235*, 422–432. doi: 10.1016/j.cpc.2018.08.018

Farrell, C., y Leonhardt, U. (2005, 1). The perfectly matched layer in numerical simulations of nonlinear and matter waves. *Journal of Optics B: Quantum and Semiclassical Optics*, *7*(1), 1–4. doi: 10.1088/1464-4266/7/1/001

Feldmann, M. T., Cummings, J. C., Kent IV, D. R., Muller, R. P., y Goddard III, W. A. (2008). Manager-worker-based model for the parallelization of quantum Monte Carlo on heterogeneous and homogeneous networks. *Journal of Computational Chemistry*, *29*(1), 8-16. doi: 10.1002/jcc.20836

Hess, K., y Iafrate, G. J. (1992, July). Approaching the quantum limit. *IEEE Spectrum*, *29*(7), 44-49. doi: 10.1109/6.144511

Janik, T., y Majkusiak, B. (1998, 07). Analysis of the MOS transistor based on the self-consistent solution to the Schrodinger and Poisson equations and on the local mobility model. *Electron Devices, IEEE Transactions on*, *45*, 1263 - 1271. doi:

10.1109/16.678531

Khemissi, S. (2012, 09). Analytical Model and Numerical Simulation for the Transconductance and Drain Conductance of GaAs MESFETs. En M. Andriychuk (Ed.), *Numerical simulation- from theory to industry* (p. 259-274). IntechOpen. doi: 10.5772/47741

Lee, C. (2019, 21 de 06). *Mixed Precision Training on Tesla T4 and P100.* https://medium.com/the-artificial-impostor/mixed-precision-training-on-tesla-t4-and-p100-d82e5d3b987d.

Levine, I. N. (2014). *Quantum chemistry* (7.ª ed.). Pearson.

Lewars, E. G. (2011). *Computational chemistry* (2.ª ed.). Springer.

Livesey, M., Stack, J. F., Costen, F., Nanri, T., Nakashima, N., y Fujino, S. (2012). Development of a CUDA implementation of the 3D FDTD method. *IEEE Antennas and Propagation Magazine*, *54*(5), 186–195. doi: 10.1109/MAP.2012.6348145

Lundstrom, M. S., y Guo, J. (2006). *Nanoscale transistors: Device physics, modeling and simulation*. Springer.

McAlinden, S., y Shertzer, J. (2016). Quantum scattering from cylindrical barriers. *American Journal of Physics*, *84*(10), 764–769. doi: 10.1119/1.4960021

Mennemann, J. F., y Jüngel, A. (2014). Perfectly Matched Layers versus discrete transparent boundary conditions in quantum device simulations. *Journal of Computational Physics*, *275*, 1–24. doi: 10.1016/j.jcp.2014.06.049

Moxley, F. I., Byrnes, T., Fujiwara, F., y Dai, W. (2012). A generalized finite-difference time-domain quantum method for the N-body interacting Hamiltonian. *Computer Physics Communications*, *183*(11), 2434–2440. doi: 10.1016/j.cpc.2012.06.012

Moxley III, F. I., Zhu, F., y Dai, W. (2012). A Generalized FDTD Method with Absorbing Boundary Condition for Solving a Time-Dependent Linear Schrödinger Equation. *American Journal of Computational Mathematics*, *2*(3), 163-172. doi: 10.4236/ajcm.2012.23022

Nagel, J. R. (2009). A review and application of the Finite-Difference Time-Domain algorithm applied to the Schrödinger equation. *Applied Computational Electromagnetics Society Journal*, *24*(1), 1–8.

Nemnes, G., Ion, L., y Antohe, S. (2010, 01). Self-consistent potentials and linear regime

conductance of cylindrical nanowire transistors in the R-matrix formalism. *Journal of Applied Physics*, *106*, 113714-1 – 113714-7. doi: 10.1063/1.3269704

Nissen, A., y Kreiss, G. (2011). An optimized perfectly matched layer for the Schrödinger equation. *Communications in Computational Physics*, *9*(1), 147–179. doi: 10.4208/cicp.010909.010410a

NVIDIA. (2019, Noviembre). *GPU applications catalog.* `https://www.nvidia.com/en-us/gpu-accelerated-applications`.

Pallás Areny, R. (2005). *Adquisición y distribución de señales*. Marcombo.

Prakash, B., Prasad, V., y Jain, S. (2010, 02). Nano scale soi mosfet structures and study of performance factors. *International Journal of Computer Applications*, *1*(28), 42-47. doi: 10.5120/513-830

Robson, W. (2019, 31 de 07). *How to Install RAPIDS in Google Colab.* `https://medium.com/dropout-analytics/installing-rapids-ai-in-google-colab-87c247f2c468`.

Saha, A. K., Sharma, P., Dabo, I., Datta, S., y Gupta, S. K. (2017). Ferroelectric transistor model based on self-consistent solution of 2D Poisson's, non-equilibrium Green's function and multi-domain Landau Khalatnikov equations. En *2017 IEEE International Electron Devices Meeting (IEDM)* (p. 13.5.1-13.5.4).

Sahay, S., y Kumar, M. J. (2019). *Junctionless field-effect transistors*. John Wiley & Sons, Inc. doi: 10.1002/9781119523543.ch8

Sanders, J., y Kandrot, E. (2011). *CUDA by Example: An Introduction to General Purpose Graphical Processing Unit*. Addison-Wesley.

Soriano, A., Navarro, E. A., Portí, J. A., y Such, V. (2004). Analysis of the finite difference time domain technique to solve the Schrödinger equation for quantum devices. *Journal of Applied Physics*, *95*(12), 8011–8018. doi: 10.1063/1.1753661

Storti, D., y Yurtoglu, M. (2015). *CUDA for Engineers: An Introduction to High-Performance Parallel Computing*. Addison-Wesley.

Strickland, M., y Yager-Elorriaga, D. (2010, 08). A parallel algorithm for solving the 3D Schrödinger equation. *Journal of Computational Physics*, *229*, 6015-6026. doi: 10.1016/j.jcp.2010.04.032

Sudiarta, I. W., y Geldart, D. J. W. (2007, feb). Solving the Schrödinger equation using

the finite difference time domain method. *Journal of Physics A: Mathematical and Theoretical*, *40*(8), 1885–1896. doi: 10.1088/1751-8113/40/8/013

Sullivan, D., y Citrin, D. (2001, 04). Time-domain simulation of two electrons in a quantum dot. *Journal of Applied Physics*, *89*(7), 3841-3846. doi: 10.1063/1.1352559

Sullivan, D. M. (2000). *Electromagnetic simulation using the fdtd method*. IEEE Microwave Theory Techniques Society. doi: 10.1002/9781118646700.ch1

Sullivan, D. M. (2012). *Quantum mechanics for electrical engineers*. Hoboken, New Jersey: John Wiley & Sons, Inc. doi: 10.1002/9781118169780

Sullivan, D. M., y Citrin, D. S. (2005). Determining quantum eigenfunctions in three-dimensional nanoscale structures. *Journal of Applied Physics*, *97*(10), 10–16. doi: 10.1063/1.1896437

Sullivan, D. M., Mossman, S., y Kuzyk, M. G. (2016, 04). Time-domain simulation of three dimensional quantum wires. *PLOS ONE*, *11*(4), 1-11. doi: 10.1371/journal.pone .0153802

Sullivan, D. M., y Wilson, P. M. (2012). Time-domain determination of transmission in quantum nanostructures. *Journal of Applied Physics*, *112*(6), 064325-1–064325-7. doi: 10.1063/1.4754812

Taflove, A., y Hagness, S. C. (2005). *Computational Electrodynamics The Finite-Difference Time-Domain Method* (3.ª ed.). Artech House.

van Wees, B. J., van Houten, H., Beenakker, C. W. J., Williamson, J. G., Kouwenhoven, L. P., van der Marel, D., y Foxon, C. T. (1988, Feb). Quantized conductance of point contacts in a two-dimensional electron gas. *Physical Review Letters*, *60*, 848–850. doi: 10.1103/PhysRevLett.60.848

Wilson, J. P. (2019). Generalized Finite-Difference Time-Domain method with Absorbing Boundary Conditions for solving the nonlinear Schrödinger equation on a GPU. *Computer Physics Communications*, *235*, 279–292. doi: 10.1016/j.cpc.2018.02.013

Xiong, X. Y., y Sha, W. E. (2014). The FDTD Method: Essences, Evolutions, and Applications to Nano-Optics and Quantum Physics. En S. M. Musa (Ed.), *Computational nanotechnology using finite difference time domain* (p. 37-82). Boca Raton: CRC Press.

Zheng, C. (2007). A Perfectly Matched Layer approach to the nonlinear Schrödinger

wave equations. *Journal of Computational Physics*, *227*(1), 537–556. doi: 10.1016/j.jcp.2007.08.004

UNIVERSIDAD AUTÓNOMA DE CHIHUAHUA

Chihuahua, Chih., a 26 de junio de 2020.
Oficio: 42/CA/SIP/20

Dr. Ildebrando Pérez Reyes
Secretario de Investigación y Posgrado
Facultad de Ciencias Químicas
Universidad Autónoma de Chihuahua
P r e s e n t e:

Los integrantes del comité, informamos a Usted que efectuamos la revisión de la tesis intitulada: **Implementación del algoritmo de Diferencias Finitas en el Dominio del Tiempo para resolver la evolución temporal de funciones de onda electrónicas** presentada por la **I.Q. Priscilla Ivette Escobedo** alumna del Programa de Maestría en Ciencias en Química.

Después de la revisión, indicamos a la tesista las correcciones que eran necesarias efectuar y habiéndolas realizado, manifestamos que la tesis, de la cual adjuntamos un ejemplar, ha cumplido con los objetivos señalados por el Comité de Tesis, por lo que puede ser considerada como adecuada para que se proceda con los trámites para la presentación de su Examen de Grado.

A t e n t a m e n t e
"Por la Ciencia para Bien del Hombre"

Dra. María Elena Fuentes Montero
Co-Directora de tesis

Dr. Juan Pedro Palomares Báez
Asesor de tesis

Dr. Marco Antonio Chávez Rojo
Asesor de tesis

Dr. José Manuel Nápoles Duarte
Director de tesis

Dr. Ildebrando Pérez Reyes
Secretario de Investigación y Posgrado

**El que suscribe certifica que las firmas que aparecen en esta acta, son auténticas, y las mismas que utilizan los C. Profesores mencionados.

FACULTAD DE CIENCIAS QUÍMICAS
Circuito Universitario
Campus Universitario #2 C.P. 31125
Tel. +52 (614) 236 6000
Chihuahua, Chihuahua, México
http://www.fcq.uach.mx