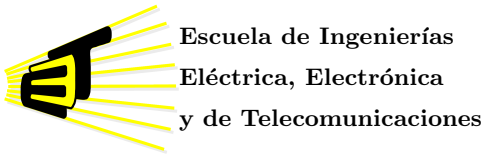
COMPARACIÓN DE LAS HERRAMIENTAS DE PROGRAMACIÓN EN PARALELO OpenCL Y CUDA PARA LA IMPLEMENTACIÓN DE LA PROPAGACIÓN DE LA ECUACIÓN DE ONDA ACÚSTICA 3D CON DENSIDAD CONSTANTE BASADA EN *STENCIL*.

Johan Sebastian Suarez Largo



Universidad Industrial de Santander

Facultad de Ingenierías Físico-Mecánicas

Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones

Bucaramanga

2016

COMPARACIÓN DE LAS HERRAMIENTAS DE PROGRAMACIÓN EN PARALELO OpenCL Y CUDA PARA LA IMPLEMENTACIÓN DE LA PROPAGACIÓN DE LA ECUACIÓN DE ONDA ACÚSTICA 3D CON DENSIDAD CONSTANTE BASADA EN *STENCIL*.

Johan Sebastian Suarez Largo

Trabajo de investigación presentado como requerimiento parcial para optar por el título de Ingeniero Electrónico

Director:

MS.Dorfell Leonardo Parra Prada

Co-Directores:

PhD(c). William Alexander Salamanca Becerra

PhD. Ana Beatriz Ramírez Silva

Escuela de Ingenierías
Eléctrica, Electrónica
y de Telecomunicaciones

Universidad Industrial de Santander
Facultad de Ingenierías Físico-Mecánicas
Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga

2016

Agradecimientos

Agradecimientos a Dios por darme una hermosa familia. A mis padres y a mis hermanos. A los profesores William, Dorfell y Ana por todo su tiempo, por escucharme, aconsejarme y guiarme en el desarrollo del trabajo. A los muchachos del "RTM Team". A los cursos de "MIT OpenCourseWare" y "Udacity". A la comunidad de desarrolladores de "Stack Overflow" y "NVidia Developer Zone". Y por supuesto a los muchachos del grupo CPS. A todos muchas gracias por que sin su ayuda no hubiera terminado este proyecto.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

Pag.

ÍNDICE DE TABLAS

Pag.

ÍNDICE DE ANEXOS

Pag.

RESUMEN

TÍTULO:

Comparación de las herramientas de programación en paralelo OpenCL y CUDA para la implementación de la propagación de la ecuación de onda acústica 3D con densidad constante basada en *stencil* *.

AUTOR:
JOHAN SEBASTIAN SUAREZ LARGO

PALABRAS CLAVES:

Inversión de onda completa, *occupancy*, métrica, CUDA, OpenCL, complejidad computacional, unidad de procesamiento gráfico.

El siguiente proyecto aporta a la apropiación tecnológica de plataformas heterogéneas, las cuales son utilizadas en aplicaciones que aprovechan las ventajas que ofrecen este tipo de plataformas respecto al enfoque de las plataformas de cómputo tradicionales. Al realizar aplicaciones en este tipo de plataformas resulta óptima la utilización de diferentes herramientas de programación como CUDA y OpenCL. En este trabajo de investigación se propone una implementación de una solución de la ecuación de onda acústica con densidad constante por medio del método *finite difference time domain* (FDTD) con las herramientas CUDA y OpenCL, con el fin de realizar un análisis comparativo entre ellas. Inicialmente se plantea un estudio de las herramientas de programación donde se muestran las ideas básicas. Luego se estudia la complejidad computacional del algoritmo a implementar y basado en ello se muestra la ganancia que se obtendrá al implementar de forma paralela dicho algoritmo. Luego se muestran los *kernels* que se crearon para realizar la solución de la ecuación y por último se realiza el análisis comparativo de métricas como tiempos de ejecución, *occupancy*, exactitud y forma de programación. De esto se obtiene que OpenCL tiene una solución 5,56873 % más exacta que CUDA, además se encuentra que OpenCL presenta un mejor desempeño que CUDA en ciertos *kernels* debido al nivel donde se programa, pero eso mismo hace que tenga un desempeño 4,82315 % más bajo en el tiempo global de la aplicación. La *occupancy* de los *kernels* que se encuentra es la misma debido que se intentan programar de la misma manera para que exista una comparación justa.

*Trabajo de grado.

**Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. MS. Directores Dorfell Leonardo Parra Prada, Ph.D(c). William Alexander Salamanca Becerra y Ph.D. Ana Beatriz Ramírez Silva.

ABSTRACT

TITLE:

Comparison of parallel programming frameworks OpenCL and CUDA to implement the spread of 3D acoustic wave equation with constant density based on stencil.

AUTHOR:

JOHAN SEBASTIAN SUAREZ LARGO**

KEYWORDS:

Analysis of algorithms, Acustic wave ecuation, OpenCL, metric, CUDA, OpenCL, graphics processing unit, occupancy.

The next project contributes to technological appropriation of heterogeneous platforms, which are used in applications that leverage the advantages of such platforms on the approach of traditional computing platforms. When designing applications in this kind of platforms the best option is to use different frameworks of parallel programming such as CUDA and OpenCL. In this research work, implementations of the acoustic wave equation solution with constant density is proposed. These implementation were made using the frameworks CUDA and OpenCL Them are based on the Finite Difference in Time-Domain (FDTD) method and the goal is to make a comparative analysis between both of them with the CPU serial implementation. First, a study about the programming frameworks is made showing the basic ideas. Then the computational complexity of the algorithm is computed and the improvements of using parallel computing of the algorithm are shown. Finally, the kernels are shown a solution of equation and finally make a comparative analysis of metrics like time of ejection, occupancy, accuracy and form of programming. This is obtained OpenCL is solvable 5,56873 % more accurate than CUDA, also is that OpenCL has a better performance than in certain CUDA kernels due to the level of frameworks, but that it does have a performance 4,82315 % lower in the overall time of the application. The occupancy of the kernels found is the same because that program in the same way becuase there is a fair comparison.

*Degree work

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Directed by MS. Directores Dorfell Leonardo Parra Prada, Ph.D(c). William Alexander Salamanca Becerra and Ph.D. Ana Beatriz Ramírez Silva.

INTRODUCCIÓN

La naturaleza del hombre siempre ha sido evolucionar. A lo largo de los tiempos se a buscado optimizar nuestro entorno para lograr hacer más con menos recursos, por esto se hace necesario el encontrar la mejor herramienta para cada aplicación [16].

Cuando se habla de procesar un gran volumen de datos, se hace necesario el estudio los temas de el procesamiento en paralelo ó programación heterogénea, esto justifica el estudio de las *General-purpose computing on graphics processing units* GPGPU como una unidad de cómputo de alto desempeño. Este estudio es un campo en crecimiento debido a sus aplicaciones en las grandes industrias, por ejemplo en el procesamiento de datos sísmicos [16][15].

Para el desarrollo de estas aplicaciones existen múltiples plataformas, de las cuales se destacan OpenCL y CUDA. OpenCL es una herramienta de programación que permite el desarrollo de aplicaciones en paralelo en múltiples plataformas [6]. CUDA es una herramienta de programación en paralelo creada por NVIDIA para el desarrollo de aplicaciones en sus GPUs [11].

El objetivo de este proyecto es implementar la solución de la ecuación de onda acústica 3D con densidad constante y hacer un análisis comparativo del desempeño que se obtiene de OpenCL y CUDA. Cabe resaltar que esta solución de la ecuación es utilizada en algoritmos como *Reverse Time Migration* (RTM) y *Full Waveform Inversion* (FWI).

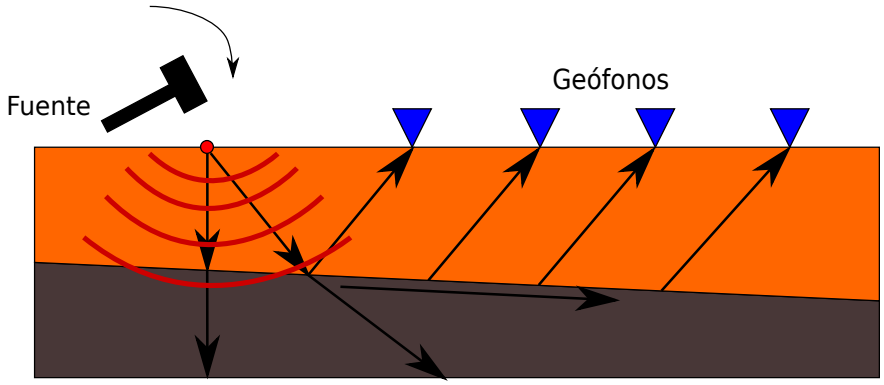
Capítulo 1

MARCO TEÓRICO

1.1 MODELADO SÍSMICO

La sismología es una rama de la geofísica, encargada del estudio acerca de la propagación de las ondas mecánicas en el interior de la tierra. La sismología se divide en varias ramas de las cuales se destaca la exploración sismológica, con la cual es posible encontrar rasgos en el subsuelo usando algunos métodos como la sismología de refracción y de reflexión. La exploración sismológica inicia con la aplicación de una perturbación en un punto específico del subsuelo, la cual puede ser un golpe seco con un martillo ó la detonación de un carga explosiva y luego captar las reflexiones de la energía con geófonos [10]. Esto se puede observar en la Figura 1.

Figura 1: Sismología de refracción y reflexión.



Con este tipo de estudios llegamos a la generación de imágenes de reflexión sísmica las cuales representan la estructura del subsuelo, los cuales son usados para estudiar el interior de la tierra. Permitiendo a la industria de los hidrocarburos la caracterización de yacimientos.

El modelado sísmico es una herramienta muy importante que nos permite sintetizar sismogramas e

imágenes del campo de onda en el espacio y tiempo. Estos cálculos son llevados a cabo con base a la ecuación de onda acústica, la cual nos permite modelar la propagación de las ondas en el subsuelo. Este tipo de fenómeno es representado por medio de EDPs (Ecuaciones Diferenciales Parciales), para las cuales hay dos tipos de soluciones: Una solución analítica y una solución numérica. Debido a que una solución analítica para este tipo de EDP es muy complicada, se ha optado por una solución numérica. Para ello se utiliza el método de diferencias finitas en el espacio tiempo (*finite difference time domain*) o FDTD, que consiste en discretizar las ecuaciones de forma que puedan ser implementadas en plataformas de cómputo [10].

1.2 ECUACIÓN DE ONDA ACÚSTICA Y CONDICIONES DE FRONTERA

1.2.1 Ecuación de Onda Acústica con Densidad Constante

La ecuación de onda acústica mostrada en (1.1a) es una ecuación diferencial parcial (EDP) lineal de segundo orden, de tipo diferencial hiperbólica, la cuál describe la propagación de perturbaciones ondulatorias de una manera simple. Al descomponer el segundo término de la ecuación (1.1a) para el caso de tres dimensiones espaciales obtenemos la ecuación (1.1b).

$$\frac{1}{v(x,y,z)^2}\frac{\partial^2 p(x,y,z,t)}{\partial t^2} = \nabla^2 p(x,y,z,t) \tag{1.1a}$$

$$\frac{1}{v(x,y,z)^2}\frac{\partial^2 p(x,y,z,t)}{\partial t^2} = \frac{\partial^2 p(x,y,z,t)}{\partial x^2} + \frac{\partial^2 p(x,y,z,t)}{\partial y^2} + \frac{\partial^2 p(x,y,z,t)}{\partial z^2} \tag{1.1b}$$

Donde (x,y,z) son las variables que recorren el espacio en coordenadas rectangulares, t es la variable del tiempo, p es el campo de presión de la onda y v es el campo de velocidades del medio por donde se propaga la onda acústica.

1.2.2 Condiciones de Frontera

Cuando hablamos de modelado de la ecuación de onda acústica por medio de métodos numéricos, se debe tener en cuenta que es un problema no acotado, lo que significa que se debe implementar un tipo de fronteras absorbentes que disipen la energía de la propagación, esto con el objetivo de evitar reflexiones no deseadas que se generan en el borde del modelo debido a que el tamaño de la memoria de la unidad no es infinita.

Existen diferentes estrategias entre las cuales estan: *Absorbing Boundary Condition* (ABC), *Perfectly Matched Layer* (PML) y *Convolutional Perfectly Matched Layer* (CPML) [2] [4] [5] [9]. En el método de ABC la energía es disipada por medio de la multiplicación del campo escalar por una función

exponencial decreciente [3]. El método PML consiste en introducir una capa absorbente de grosor finito alrededor del modelo, que minimice las reflexiones[1]. El método CPML es una variante de PML dónde se busca mejorar las reflexiones en baja frecuencia [4]. Para este trabajo se seleccionó CPML debido a la aceptación en el estado del arte de este método para implementaciones de RTM. En este método se puede resumir en agregar dos variables auxiliares (ϕ, ζ) que representan los campos que disipan la energía en cada dimensión espacial. La ecuación 1.2 representa la ecuación de la onda acústica con CPML [4] [5].

$$\begin{aligned}\frac{1}{v(x,y,z)^2}\frac{\partial^2 p(x,y,z,t)}{\partial t^2} = \nabla^2 p(x,y,z,t) + \frac{\partial \psi_x(x,y,z)}{\partial x} + \frac{\partial \psi_y(x,y,z)}{\partial y} + \frac{\partial \psi_z(x,y,z)}{\partial z} \\ +\zeta_x(x,y,z) + \zeta_y(x,y,z) + \zeta_z(x,y,z)\end{aligned} \tag{1.2}$$

La literatura [4] nos muestra que las variables auxiliares que minimizan las reflexiones en las fronteras vienen dadas por las ecuaciones (1.3) y (1.4).

$$\psi_i^n = b_i \psi_i^{n-1} + a_i \frac{\partial p(x,y,z,t)}{\partial i} \tag{1.3}$$

$$\zeta_i^n = b_i \zeta_i^{n-1} + a_i \left[\left(\frac{\partial^2 p(x,y,z,t)}{\partial i^2} \right)^n + \left(\frac{\partial \psi_i}{\partial i} \right)^n \right] \tag{1.4}$$

Donde i puede ser las dimensiones espaciales, n representa el paso de tiempo en que se encuentra, además se asume que para la primera iteración ζ_i^0 y ψ_i^0 son cero. Las constantes a_i y b_i pueden ser calculados usando las ecuaciones (1.5) y (1.6).

$$a_i = e^{-(\sigma_i + \alpha_i)\Delta t} \tag{1.5}$$

$$b_i = \frac{\sigma_i}{\sigma_i + \alpha_i}(a_i - 1) \tag{1.6}$$

Donde i puede ser las dimensiones espaciales, Δt es el valor del paso temporal y las constantes σ_i y α_i están dadas por las ecuaciones (1.7b) y (1.8) teniendo en en cuenta que $\sigma_i(0)$ esta dado por la ecuación (1.7a). Donde v_{max} es la velocidad máxima del modelo de velocidades, R es el coeficiente de reflectividad, L es la longitud de puntos de zona de atenuación, Δh es el valor de paso espacial y F es un vector que tiene valores $[0:1:L-1]$ [5].

$$\sigma_i(0) = \frac{-3}{2L}\ln(R) \tag{1.7a}$$

$$\sigma_i = \sigma_i(0) v_{max} \Delta h \left(\frac{F}{L} \right)^2 \tag{1.7b}$$

$$\alpha_i = \pi f_c \left(\frac{L - F}{L} \right) \tag{1.8}$$

Además se debe tener en cuenta que estos parámetros son 0 para regiones donde NO se aplique atenuación CPML, es decir $[L : N_i - L]$, lo que implica que ψ_i y ζ_i también son cero, y por lo tanto el campo escalar está dado solo por la solución de la ecuación (1.1b). Por otra parte, la ecuación de CPML (1.2), sólo será valida en las fronteras atenuantes, es decir solo entre $[0 : L]$ y $[N_i - L : N_i]$, donde N_i es el tamaño del modelo en una dirección determinada y L es el tamaño de la frontera absorbente.

1.3 SOLUCIÓN DE LA ECUACIÓN DE ONDA ACÚSTICA POR FDTD

La solución $p(x, y, z, t)$ de la ecuación diferencial (1.1b), representa la intensidad de presión de la onda en el espacio y el tiempo. El método de diferencias finitas en el espacio tiempo (*finite difference time domain* o en sus siglas FDTD) es uno de los más usados para discretizar la ecuación de la onda. Dicho método toma las derivadas y las aproxima por medio de las series de Taylor. A más términos en la serie mejor será la aproximación. Para discretizar la ecuación de onda es necesario conocer los tipos de esquemas en diferencias finitas que existen:

1. Esquema de diferencias finitas hacia atrás :

$$\left.\frac{\partial f}{\partial x}\right|_{x=x_0} = \frac{f(x_0) - f(x_0 - h)}{h} + O(h) \tag{1.9}$$

   Este esquema se usa para determinar un aproximación de la derivada en un punto específico x_0 teniendo en cuenta un punto anterior a una distancia h.

2. Esquema de diferencias finitas hacia adelante :

$$\left.\frac{\partial f}{\partial x}\right|_{x=x_0} = \frac{f(x_0 + h) - f(x_0)}{h} + O(h) \tag{1.10}$$

   Este esquema se usa para determinar un aproximación de la derivada en un punto específico x_0 teniendo en cuenta un punto siguiente a una distancia h.

3. Esquema de diferencias finitas centradas :

$$\left.\frac{\partial f}{\partial x}\right|_{x=x_0} = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + O(h) \tag{1.11}$$

   Este esquema se usa para determinar un aproximación de la derivada en un punto específico x_0 teniendo en cuenta un punto anterior y un punto adelante, ambos puntos a una distancia h del punto de interés.

Como se observa en la lista anterior los esquemas en diferencias finitas son para las primeras derivadas. Si se toman más términos en la serie de Taylor la precisión de la derivada mejora, a esto se le conoce como aumento del orden en la diferencia. El orden bien dado por el número de puntos que se usan para la aproximación y los ordenes de aproximación son pares, por ejemplo los esquemas que se mostraron anteriormente son de orden dos.

Para una segunda derivada la aproximación en diferencias finitas se muestra la ecuación (1.12), con orden dos de precisión en la aproximación.

$$\left.\frac{\partial^2 f}{\partial x^2}\right|_{x=x_0} \cong \frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} \tag{1.12}$$

Cabe resaltar que para nuestra aplicación se utilizaran los esquemas de diferencias finitas centradas con una aproximación de la primera y segunda derivada tanto de segundo orden como de octavo orden.

Basándose en la ecuación anterior (1.12), se puede deducir que la ecuación de la onda 3D (1.1b) discretizada para un orden dos de aproximación en el espacio y un orden dos en aproximación en el tiempo, esto se muestra en la ecuación (1.13).

$$\begin{gathered}\frac{1}{v^2}\frac{P^{n-1}_{x,y,z}-2P^{n}_{x,y,z}+P^{n+1}_{x,y,z}}{\Delta t^2}=\frac{C_{-1}P^{n}_{x-1,y,z}+C_0P^{n}_{x,y,z}+C_1P^{n}_{x+1,y,z}}{\Delta x^2}+\\ \frac{C_{-1}P^{n}_{x,y-1,z}+C_0P^{n}_{x,y,z}+C_1P^{n}_{x,y+1,z}}{\Delta y^2}+\frac{C_{-1}P^{n}_{x,y,z-1}+C_0P^{n}_{x,y,z}+C_1P^{n}_{x,y,z+1}}{\Delta z^2}+\\ f^{n}_{x_f,y_f,z_f}\end{gathered} \tag{1.13}$$

Donde los C_i representan los coeficientes de la aproximación [7] y $P^n_{x,y,z}$ es el campo escalar de la intensidad de presión para la el punto x,y,z y el tiempo n y $f^n_{x_0,y_0,z_0}$ es la fuente que perturba el medio.

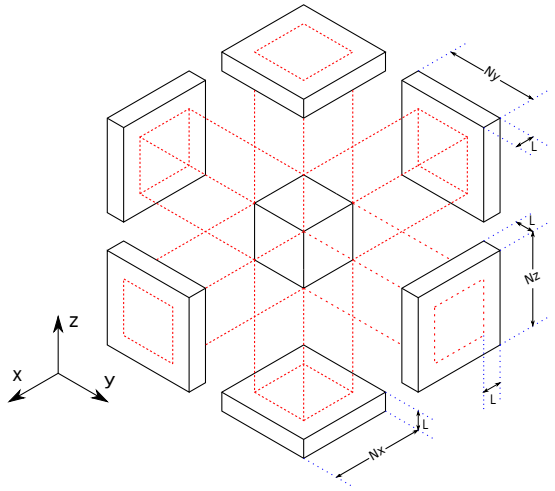
Despejando $P^{n+1}_{x,y,z}$ se obtiene:

$$P^{n+1}_{x,y,z}=-P^{n-1}_{x,y,z}+2P^{n}_{x,y,z}+\Delta t^2v^2\left(\begin{array}{c}\frac{C_{-1}P^{n}_{x-1,y,z}+C_0P^{n}_{x,y,z}+C_1P^{n}_{x+1,y,z}}{\Delta x^2}+\\ \frac{C_{-1}P^{n}_{x,y-1,z}+C_0P^{n}_{x,y,z}+C_1P^{n}_{x,y+1,z}}{\Delta y^2}+\\ \frac{C_{-1}P^{n}_{x,y,z-1}+C_0P^{n}_{x,y,z}+C_1P^{n}_{x,y,z+1}}{\Delta z^2}\end{array}\right)+\Delta t^2v^2f^{n}_{x_f,y_f,z_f} \tag{1.14}$$

La ecuación (1.14) puede ser implementada en plataformas de cómputo de una manera simple, pero se debe tener en cuenta que esta solución es donde las fronteras absorbentes no tienen efecto, por lo tanto

solo tienen validez entre $[L : N_i - L]$, donde N_i es el tamaño del modelo en una dirección determinada y L es el tamaño de la frontera absorbente. Lo anterior mencionado de puede observar en la Figura 2.

Figura 2: Estructura del campo 3D con las fronteras CPML y sus dimensiones.



Para la solución de $p(x, y, z, t)$ en las fronteras se debe discretizar la ecuación (1.2) como se mostró anteriormente, pero se debe tener en cuenta que existe un dependencia en las constantes ψ_i y ζ_i con respecto a $p(x, y, z, t)$.

A continuación se presenta a modo de ejemplo el cálculo de las constantes necesarias para las condiciones de frontera en la dirección x. Para el cálculo de las constates σ_x, α_x, a_x y b_x es necesario asumir las siguientes constantes:

Tabla 1: Tabla de datos Δt, Δh, L, R, V_{max} y f_c.

Constante	Descripción	Valor
Δt	Valor de paso temporal	0,004$[s]$
Δh	Valor de paso espacial	10$[m]$
L	Longitud de las fronteras absorbentes	10$[puntos]$
R	Coeficiente de reflexión	0,001
V_{max}	Velocidad máxima del modelo	2000$[m/s]$
f_c	Frecuencia de la fuente	10$[Hz]$

Ahora basándose en los valores anteriores y aplicando los en las ecuaciones (1.7a), (1.7b), (1.8), (1.5) y (1.6) se puede obtener que:

$$\sigma_i(0) = 1,03616329184732 \tag{1.15}$$

Tabla 2: Tabla de datos σ_x, α_x, a_x y b_x.

σ_x	α_x	a_x	b_x
0	$31,4159$	$0,8819$	$-0,1179$
$207,23265$	$28,2743$	$0,3898$	$-0,6096$
$828,93063$	$25,1327$	$0,0328$	$-0,9663$
$1865,0939$	$21,9911$	$0,0005$	$-0,9986$
$3315,7225$	$18,8495$	$1,6115e-06$	$-0,9991$
$5180,8164$	$15,7079$	$9,3910e-10$	$-0,9991$
$7460,3757$	$12,5663$	$1,0427e-13$	$-0,9991$
$10154,400$	$9,42477$	$2,2061e-18$	$-0,9991$
$13262,890$	$6,28318$	$8,8937e-24$	$-0,9991$
$16785,845$	$3,14159$	$6,8319e-30$	$-0,9991$

Cabe denotar que las constantes anteriormente mencionadas solo se deben calcular una única vez, para la aplicación de fronteras absorbentes a la ecuación de onda acústica.

Ahora se enunciaran los pasos para calcular los aportes de las fronteras absorbentes CPML a la ecuación de onda acústica solo en la dirección x debido a que es extrapolable para las otras dos.

1. El primer paso es calcular la primera derivada parcial del campo de precisión P en la dirección x la cual es necesaria para el cálculo de ψ_x^n, para ello se usa las diferencia finitas centradas de orden dos esto se muestra en la ecuación (1.16).

$$\frac{\partial p(x,y,z,t)}{\partial x} \cong \frac{C_{-1}P^n_{x-1,y,z} + C_1 P^n_{x+1,y,z}}{\Delta x} \tag{1.16}$$

2. El segundo paso es calcular ψ_x^n basados en la ecuación (1.3) de forma discreta la cual se muestra en la ecuación (1.17).

$$\psi_x^n = b_x \psi_x^{n-1} + a_x \frac{\partial p(x,y,z,t)}{\partial x} \tag{1.17}$$

3. El tercer paso es calcular la primera derivada parcial de ψ en la dirección x la cual es necesaria para el cálculo de ζ_x^n, para ello se usa las diferencias finitas centradas de orden dos, esto se muestra en la ecuación (1.18).

$$\frac{\partial \psi_x}{\partial x} \cong \frac{C_{-1}\psi^n_{x-1} + C_1 \psi^n_{x+1}}{\Delta x} \tag{1.18}$$

4. El cuarto paso es calcular la segunda derivada parcial del campo de precisión P en la dirección x la cual es necesaria para el cálculo de ζ_x^n, para ello se usa las diferencias finitas centradas de orden dos, esto se muestra en la ecuación (1.19).

$$\frac{\partial^2 p(x,y,z,t)}{\partial x^2} \cong \frac{C_{-1}P^n_{x-1,y,z} + C_0 P^n_{x,y,z} + C_1 P^n_{x+1,y,z}}{\Delta x^2} \tag{1.19}$$

5. El quinto paso es calcular ζ_x^n basados en la ecuación (1.4) de forma discreta la cual se muestra en la ecuación (1.20) y con los resultados de los pasos cuatro y tres.

$$\zeta_x^n = b_x \zeta_x^{n-1} + a_x \left[\left(\frac{\partial^2 p(x,y,z,t)}{\partial x^2} \right)^n + \left(\frac{\partial \psi_i}{\partial x} \right)^n \right] \tag{1.20}$$

6. El sexto paso es sumar el aporte basados en la ecuación (1.2).

El proceso anteriormente descrito puede ser implementada en plataformas de cómputo de una manera simple, pero se debe mantener el orden debido a que existe dependencia de datos.

1.4 GPU

La *Graphics Processing Unit* (GPU) es una unidad destinada para el procesamiento de gráficos, la cual se compone de un gran número de núcleos simples, debido a ello se puede realizar el procesamiento en paralelo. Las compañías que más producen GPUs son Intel, NVIDIA y AMD/ATI.

1.4.1 Arquitectura GPUs NVIDIA

La GPU internamente posee múltiples *Streaming Processors* (SP ó *Cores*). Los cuales son núcleos simplificados que se encargan de ejecutar instrucciones o *threads* de manera concurrente. Esta característica permite paralelar algoritmos con características determinadas. Los núcleos se agrupan en *Streaming Multiprocessor* (SM) como se muestra en la Figura 3. Unidades como registros, memoria *cache* L1, L2, SFU, *warp scheduler*, *Text Memory*, *Constant memory*, también se encuentran en el SM [14].

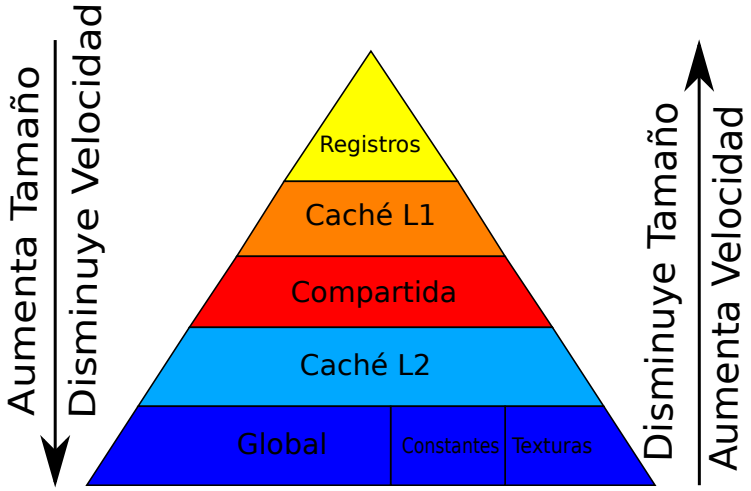
Figura 3: SM de la arquitectura *Kepler*.

Fuente: Adaptado de [13].

El paralelismo que utilizan las GPUs se conoce como *Single Instruction Multiple Thread* (SIMT), el cual consiste en asignarle una instrucción a un conjunto de *threads*, la cual va ser ejecutada por varios núcleos. Cada *thread* posee registros para el almacenamiento de datos. Los *threads* se agrupan en *warps*, en 32 *threads* por *warp*.

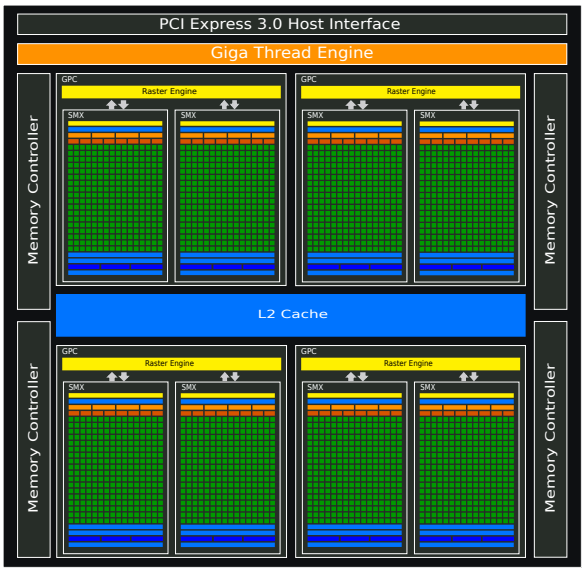
Los *warps* representan el número de *threads* que se ejecutan en un mismo instante tiempo. La administración de estos es realizada por los *warps Scheduler*, los cuales multiplexan en tiempo los *warps* y envian las solicitudes de acceso a memoria. Los SM tiene una limitación de ejecución de *warps* al mismo tiempo la cual está dada por el número máximo de *threads* por SM [14].

Figura 4: Jerarquía de Memoria GPU.



La GPU tiene una jerarquía de memoria como la que se muestra en la Figura 4. En la parte baja de la jerarquía las memorias tienen mayor capacidad y mayor latencia. En la cima las memorias de menor tamaño y menor latencia. Cabe resaltar que los registros, la *cache* L1, L2 y la memoria compartida (*Memory Shared*) se encuentran sobre el mismo sustrato de la GPU y son de tipo SRAM. Mientras que la memoria global, constantes y la memoria de textura se encuentran por fuera del chip, como se muestra en la Figura 5.

Figura 5: GPU de la arquitectura *Kepler*.

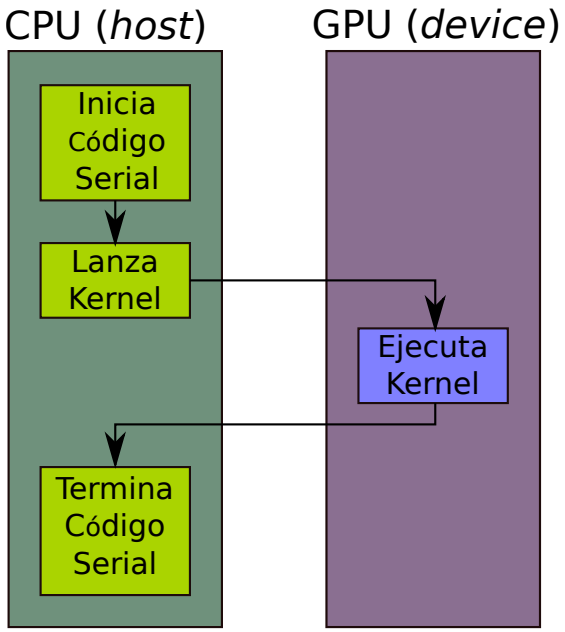


Fuente: Adaptado de [13].

1.4.2 Programación Heterogénea

En una GPU varios núcleos ejecutan instrucciones en paralelo. Por tal razón, la programación de la GPU no puede hacerse usando la programación en serie estándar, donde el principio es que las instrucciones se ejecutan de manera secuencial. Por tal razón la programación heterogénea permite aprovechar la capacidad que tiene la GPU de realizar cálculos en paralelo [11].

Figura 6: Programación Hetereogénea.



Fuente: Adaptado de [11].

La Figura 6 muestra la dinámica de la programación heterogénea. En la programación heterogénea la CPU es la encargada de ejecutar la parte secuencial del algoritmo. A la CPU se le denomina *host*, y está encargado de enviar el *kernel* para que la GPU lo ejecute, también define el tamaño del *kernel* y hace las transferencia de memoria entre ella y la GPU. La GPU se le suele llamar *device*, y su ejecución está descrita por el *kernel*.

1.5 MODELO DE PROGRAMACIÓN CUDA

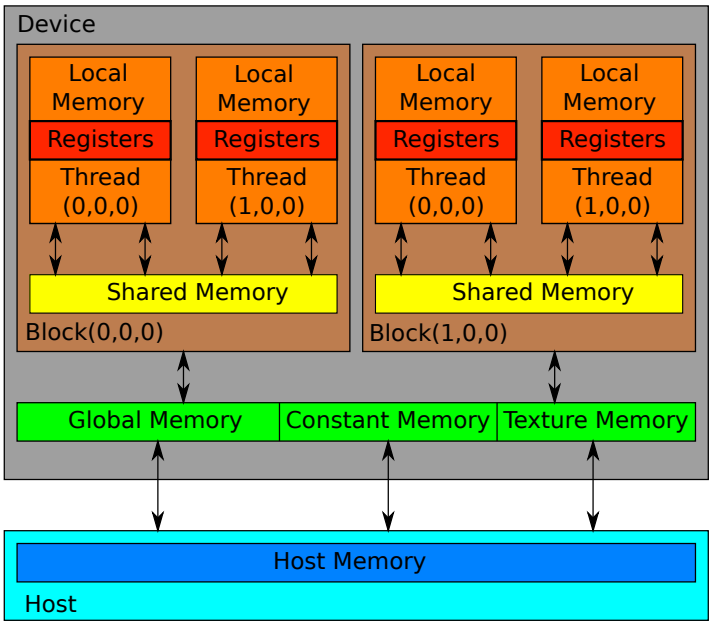
Fue creado en 2006 por NVIDIA con el fin de utilizar GPUs en propósitos generales, permitiendo aplicaciones específicas, en áreas como la dinámica de fluidos, procesamiento de datos sísmicos, etc. El modelo de programación CUDA es similar al estilo del modelo de software *Single Program Multiple-*

Data (SPMD)[1], con la diferencia que CUDA está basado en lanzar un *kernel*, en un coprocesador. Este lenguaje de programación es una extensión de C++ con funciones adicionales y con los principios de la programación heterogénea [14].

El modelo de programación CUDA consta de tres componentes principales, el primero es el *thread* o hilo el cual es encargado de ejecutar las instrucciones a un dato o conjuntos de datos en específico. Los *threads* se pueden ordenar como una grilla en 2D o 3D dependiendo de la aplicación o sea *threads* (Tx, Ty, Tz). El siguiente nivel es *Threads Blocks* (Bloques de Hilos) el cual es una agrupación de 1024 *threads* o menos sin importar el orden, lo que quiere decir que 2D ó 3D, es decir *threads Blocks* en direcciones (Dx, Dy, Dz). El último nivel es la *Grid* (Enmallado), la cual agrupa los *threads blocks* y define el tamaño del *kernel*. Estos tres componentes se pueden observar en la Figura 7, además se puede observar que los *threads* tienen acceso a la *Local Memory* (Memoria Local) y registros, los *threads blocks* tienen acceso a la *shared memory* (Memoria Compartida) y la *grid* tiene acceso a la *global memory*. Cabe resaltar que el programador debe especificar el número de *threads* por *threads blocks* y número de *threads blocks* por *grid* [11].

En el apéndice 5.2 se muestra una implementación de una suma de vectores en CUDA.

Figura 7: Modelo de memoria en CUDA.



Fuente: Adaptado de [24].

[1]Es una técnica empleada para conseguir paralelismo a nivel de datos, la cual consiste en un conjunto de datos que se ejecutan de forma separada en un único *core* por dato para obtener los resultados con mayor rapidez

1.6 MODELO DE PROGRAMACIÓN OPENCL

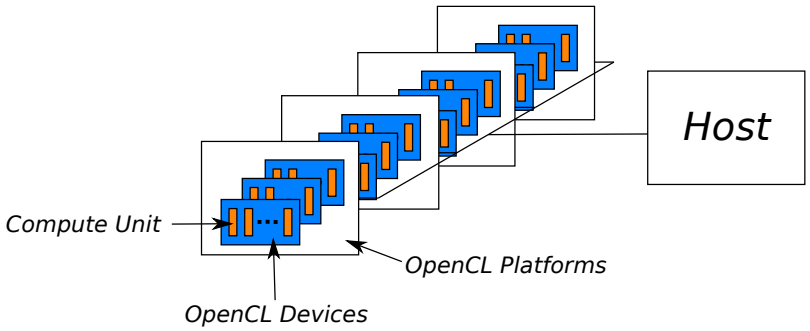
Es un estándar portátil abierto de programación heterogénea creado en 2008 inicialmente propuesto por Appel Inc. y el grupo *Khronos* con la idea que soportara las tarjetas gráficas más vendidas incluyendo NVIDIA, AMD, ARM, Intel, IBM, etc. Sin embargo el código escrito en OpenCL no puede ser fácilmente optimizado. Debido a esto, la optimización de los compiladores va de la mano con el hardware [15]. El problema del rendimiento es uno de los temas más interesantes en la computación paralela, por esta razón un excelente tema de investigación consiste en determinar si con OpenCL se puede lograr el rendimiento superior ó similar a otros desarrollos con plataformas que están diseñados para dispositivos específicos [15].Para describir las ideas básicas de OpenCL,las dividiremos en los siguientes modelos tomados de [6].

- Modelo de la plataforma.
- Modelo de ejecución.
- Modelo de memoria.
- Modelo de programación.

1.6.1 Modelo de la plataforma

El modelo de la plataforma de OpenCL se define como una representación del conjunto de plataformas heterogéneas usadas con OpenCL. Esto se puede observar en la Figura 8. Una plataforma OpenCL siempre esta compuesta por una unidad central (*host*) y varios dispositivos de procesamiento, los dispositivos de procesamiento se dividen en unidades de procesamiento y las unidades de procesamiento se dividen en uno o varios elementos de procesamiento. Donde las unidades de procesamiento pueden ser una o varias GPGPUs, CPUs, FPGAs u otros procesadores de los cuales OpenCL tenga soporte. Además se debe tener en cuenta que sobre estas unidades de procesamiento o *Device* es donde se ejecutan los *kernels*.

Figura 8: Modelo de la plataforma OpenCL.



Fuente: Tomado de [6].

1.6.2 Modelo de ejecución

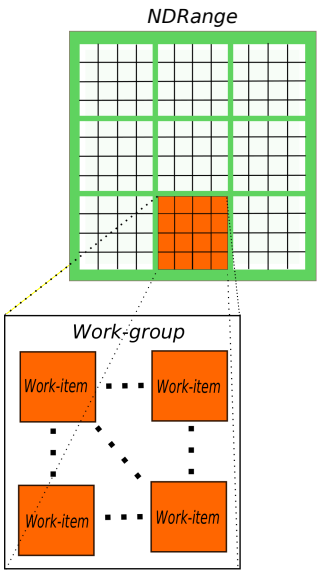
Una aplicación en OpenCL esta basado en el principio de la programación heterogénea, la cual nos indica que existe una parte del programa que se ejecuta *host* y unos *kernels* que se ejecutan en los *devices*.

Un *kernel* es una función simple que toma unos datos de entrada, los opera y arroja unos datos de salida [6]. OpenCL define dos tipos de *kernel*:

- **OpenCL Kernels:** Estos *kernels* son funciones escritas con el lenguaje C99 con un funciones de OpenCL[6].
- **Native Kernels:** Estos *kernels* son funciones que se crean fuera de OpenCL y se acceden dentro de OpenCL a través de una función puntero. Estas funciones pueden ser, por ejemplo, definidas en el código fuente de *host* o exportadas a partir de una librería especial. Tenga en cuenta que la capacidad de ejecutar *kernels* nativos es una funcionalidad opcional dentro OpenCL y la semántica de *kernels* nativos son definido por la implementación[6].

El *kernel* es definido y lanzado en *host*, cuando un kernel es lanzado se define un espacio entero indexado donde cada punto del espacio se conoce como *work-item*, dicho elemento se encarga de ejecutar la función que esta escrita en el *kernel* y se organizan en *work-groups*. Los *work-items* y los *work-groups* tienen un indexado global y un indexado local. Los *work-groups* se organizan en el espacio denominado NDRange. Un NDRange es un espacio de índice N-dimensional, donde N es uno, dos o tres [6]. Esto se puede observar en la Figura 9.

Figura 9: Distribución de indexado en OpenCL.



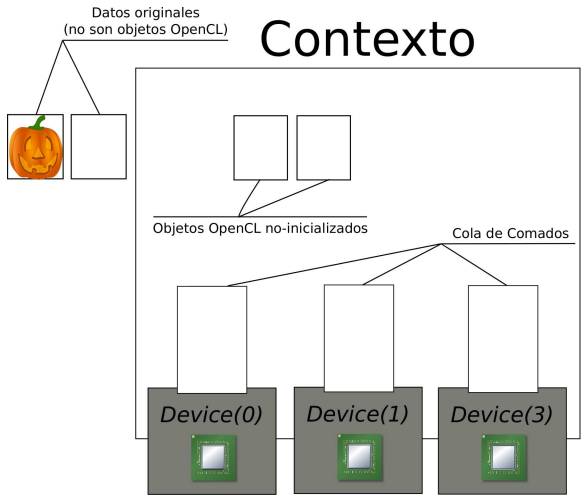
Fuente: Adaptado de [6].

Además el modelo de ejecución de OpenCL contiene los siguientes conceptos:

1. **Contexto:**

   El contexto en OpenCL se puede ver como el lugar donde van asociar todo los elementos necesarios para la ejecución de nuestra aplicación. Para la creación de un contexto siempre lleva asociado un o varios *devices* de una misma plataforma, este concepto se muestra en la Figura 10.

Figura 10: Contexto, cola de comandos y *Buffers* de memoria en OpenCL.



Fuente: Tomado de [6].

2. **Cola de Comandos:**

   La cola de comandos es un elemento donde se colocan todas las acciones que van ser ejecutadas para la aplicaciones, tales como creación, escritura y lectura de los objetos de memoria, lanzamiento de *kernels*, etc.
   La cola de comandos necesita un contexto ya creado. La idea de la cola de comandos dentro de contexto sería como la que se muestra en la Figura 10.

3. **Objetos de memoria:**

   Los objetos de memoria son la forma de manejar datos en aplicaciones de OpenCL, existen de dos tipos:

   - **Buffer Objects:** Son bloques de memoria contiguos que se encuentra en el o los *devices*, son creados en el *host* y dichos objetos pueden ser accedidos como vectores o punteros. Además se pueden leer y escribir en el mismo *kernel*.
   - **Image Objects:** Son objetos de memoria que solo pueden ser usado para guardar imágenes, son objetos 2D o 3D y solo pueden ser leídos o escritos en un mismo *kernel*.

   Para nuestra aplicación se usaron *Buffer Objects* debido a que se va a leer y escribir en el mismo *kernel*, para la manipulación de estos objetos de memoria existen diferentes comandos. Una extrapolación de este concepto se puede mostrar en la Figura 10.

4. **Elementos de compilación y ejecución de los *kernels*:**

   Estos elementos como su nombre lo dice son los encargados de la compilación y la ejecución de los *kernels*, pero se debe tener en cuenta que existen dos formas de compilación:

   - **Online:** Este tipo de compilación es en tiempo de ejecución, lo que significa que en el código de *host* se debe leer un archivo `.cl` con los *kernels*, utilizar comandos de OpenCL para compilar los *kernels* y crear los binarios que serán enviados al o los *devices*.
   - **Offline:** Este tipo de compilación requiere que se haga un pre-compilado antes de ejecutar la aplicación y en el código en *host* leer el binario de la pre-compilación y con ello crea los *kernel*.

Cabe resaltar que para esta aplicación se uso la compilación *Online* debido a que OpenCL no soporta compilación *Offline* en GPUs NVIDIA [6].

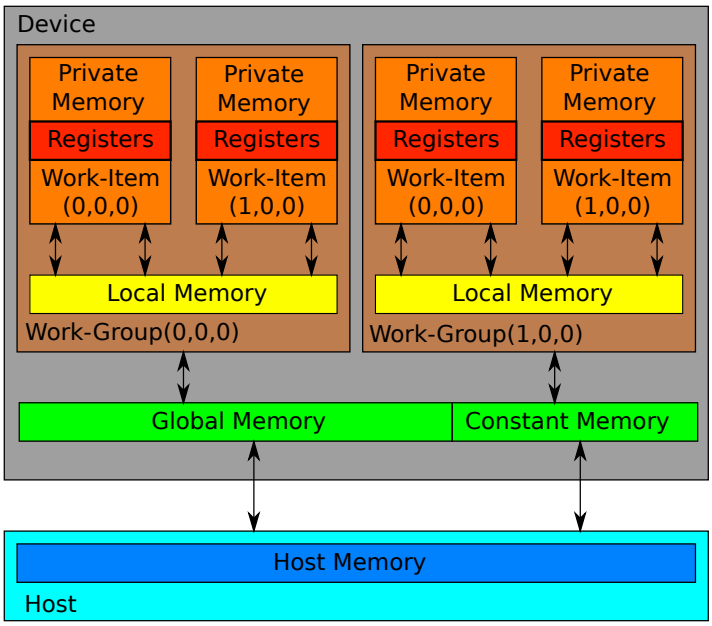
1.6.3 Modelo de memoria

El modelo de memoria de OpenCL define cinco diferentes regiones:

- **Memoria de *Host* o *Host Memory*:** Esta memoria es visible solo para el *host* y es donde se almacena los datos iniciales de nuestras aplicaciones, se manipula a través de los comandos 10, 11, 12, etc [6].
- **Memoria Global o *Global memory*:** Esta memoria permite de lectura y escritura por todos los *work-items* de todos *work-grups*. Además los *work-items* pueden leer o escribir una o varias direcciones de memoria, en esta memoria es donde se almacenan los *buffer* de memoria y normalmente esta por fuera de la GPU [6].
- **Memoria Constante o *Constant memory*:** Una región de memoria global que se mantiene constante durante la ejecución del kernel. El *host* asigna e inicializa los objetos de memoria, además los *work-items* solo la pueden leer [6].
- **Memoria Local o *Local memory*:** Esta memoria esta ubicada dentro de la GPU y es local a los *work-grups*, en esta memoria se pueden asignar variables que son compartidas por los *work-items* de un *work-grup* [6].
- **Memoria privada o *Private memory*:** Esta memoria es privada a los *work-items*. Las variables definidas en memoria privada de un *work-item* no son visibles a otros *work-items*, esta memoria en las GPUs NVIDIA es una parte de la memoria global [6].

Los conceptos anteriormente mencionados se pueden representar con la Figura 11.

Figura 11: Modelo de memoria en OpenCL.



Fuente: Adaptado de [24].

1.6.4 Modelo de programación

El modelo de programación de OpenCL es flexible debido a que depende de los algoritmos que quieran programar, OpenCL define dos diferentes modelos de programación en paralelo:

- **Paralelismo de Datos:** Se define como el cálculo en términos de una secuencia de instrucciones se aplica a varios elementos de un objeto de memoria [8]. En un modelo paralelo de datos, existe una asignación uno a uno entre el *work-item* y el elemento en un objeto de memoria sobre el cual un kernel puede ser ejecutado en paralelo, por ejemplo la suma de vectores que se presenta en la Figura 1 [6].
- **Paralelismo de Tareas:** Se define como un modelo donde un tarea es un *kernel* que se ejecuta en un único *work-item* independiente del *NDRange* y además pueden existir otros *kernels* ejecutándose concurrentemente. Es equivalente a decir que se asignará diferentes tareas a diferentes *work-items* de manera concurrente [6].

En el apéndice 5.2 se muestra una implementación de una suma de vectores en OpenCL.

Capítulo 2

ANÁLISIS DE LA COMPLEJIDAD COMPUTACIONAL

El análisis de los algoritmos es un campo de investigación importante en la ciencias de la computación, debido a que busca mejorar las implementaciones de los algoritmos en aspectos como simplicidad, robustez, facilidad de uso, funcionalidad, eficiencia, etc [18].

El objetivo de estudiar el análisis de algoritmos en este proyecto, es observar el comportamiento del tiempo de ejecución de la solución numérica por FDTD de la ecuación de onda acústica 3D con densidad constante, a medida que el tamaño del modelo cambia y la ganancia que se tiene al implementar dicho algoritmo de forma paralela. Para ello se hizo un análisis en complejidad computacional como lo muestra el libro "*Introduction to Algorithms*" [18].

El análisis en complejidad computacional estudia la eficiencia de los algoritmos estableciendo su efectividad de acuerdo al tiempo de ejecución y al espacio requerido para almacenar los datos, ayudando a evaluar la viabilidad de la implementación práctica en tiempo y costo. Además provee herramientas para clasificar la dificultad de un problema, de esta manera se puede conocer previamente si un algoritmo es eficiente para la solución de dicho problema [18]. El análisis en complejidad computacional se hace independiente de la unidad de cómputo que se piense usar para implementar el algoritmo.

El objetivo del análisis en complejidad computacional es estimar el comportamiento de un algoritmo cuando el tamaño del problema tiende al infinito, a esto se le conoce como análisis asintótico [18]. Por ejemplo se tiene que la ecuación (2.1) la cual define el tiempo de ejecución de un algoritmo en particular.

$$T(n) = a \cdot n^2 + b \cdot n + c \quad \text{para} \quad n \geq 1 \tag{2.1}$$

Donde a, b y c son constantes que depende de sistema donde se ejecute el algoritmo y n es la variable que al ser modificada afecta el tiempo de ejecución del algoritmo, ya sea el tamaño de algún vector ó matriz de entrada, número de iteraciones, etc. El objetivo de análisis asintótico es determinar que pasa con el tiempo de ejecución cuando $n \to \infty$, para ello se usa la notación asintótica (Θ) la cual nos dice que ignoremos la constantes del sistema y observemos la tasa o orden de crecimiento, para el ejemplo de la ecuación (2.1) se observa que la tasa de crecimiento viene dada por el término $a \cdot n^2$, entonces podríamos decir que:

$$T(n) = \Theta(n^2) \tag{2.2}$$

Esto quiere decir que el tiempo de ejecución crece de orden n^2 a medida que n crece. A continuación se presentará de manera formal la notación asintótica (Θ) [18]:

$\Theta(g(n))$	=	$\{f(n)$: Si existen las constantes positivas c_1, c_2 y n_0 de tal manera que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0\}$.

Si aplicamos esta definición al ejemplo anterior y asumimos que $a = 1/2$, $b = -3$ y $c = 0$ se obtiene que:

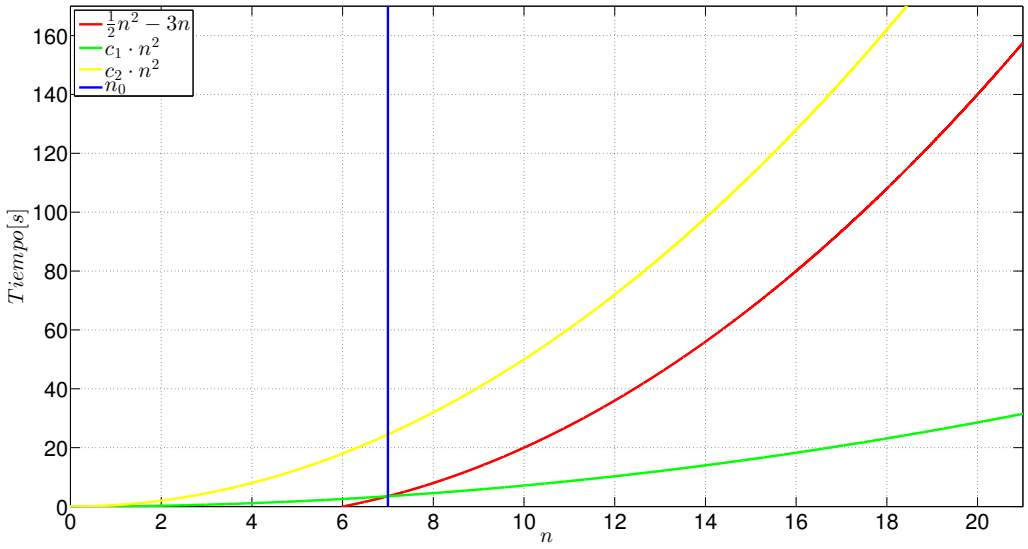
$$\begin{gathered} T(n) = \frac{1}{2}n^2 - 3n \\ c_1 \cdot n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 \cdot n^2 \end{gathered} \tag{2.3}$$

Dividiendo en ambas partes por n^2 se obtiene que:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \tag{2.4}$$

De lo anterior se puede deducir que si $n \geq n_0$ la ecuación (2.4) es decreciente a medida que n crece, entonces $c_2 \geq 1/2$. Si se asume que $n_0 = 7$ debido a que es el n donde $T(n)$ es positiva, se puede determinar con la ecuación (2.4) que $c_1 \leq 1/14$, este resultado se puede ver el Figura 12. Este ejemplo es tomado de [18].

Figura 12: $T(n)$, $c_1 \cdot n^2$, $c_2 \cdot n^2$ y n_0



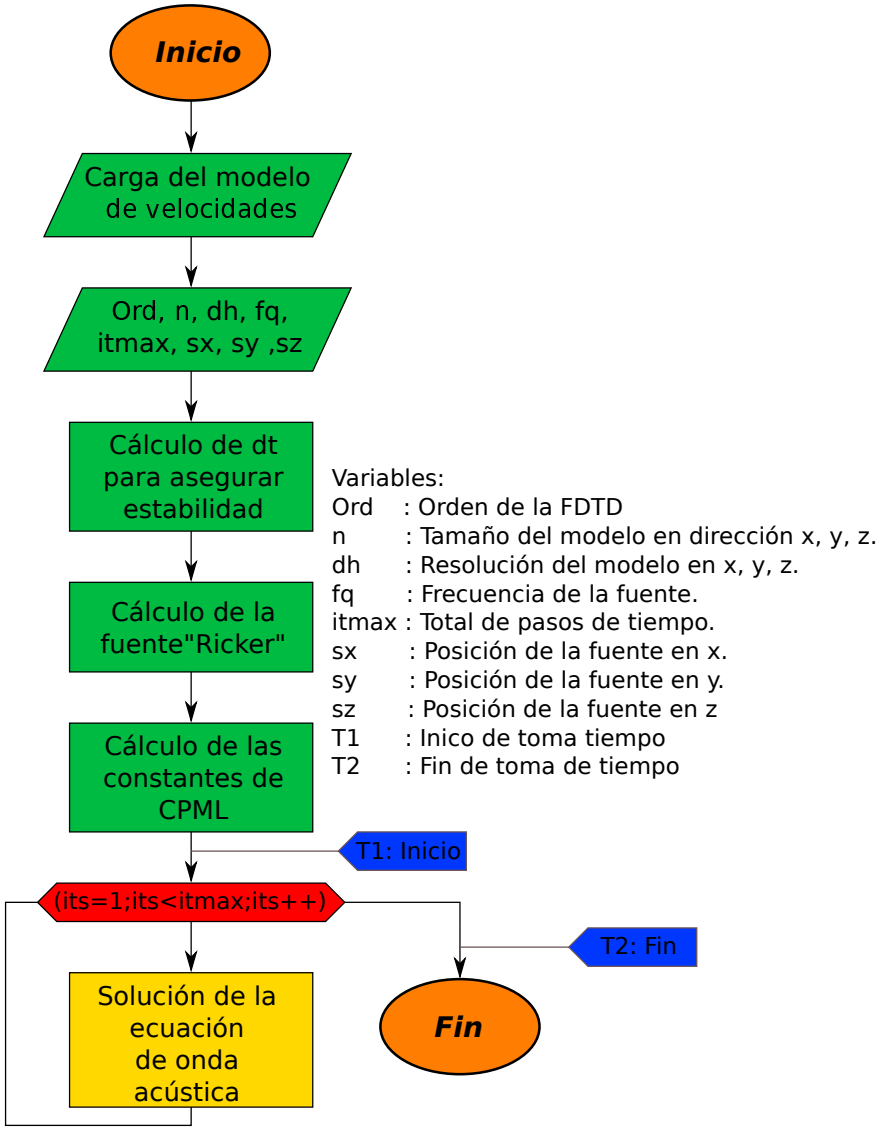
En general para todo polinomio dado por $p(n) = \sum_{i=0}^{d} a_i n^i$, donde a_i es una constante, d es el orden del polinomio y $a_d > 0$, se obtiene que $p(n) = \Theta(n^d)$ [18]. Cabe resaltar que existen más operadores además de Θ los cuales no usaremos para la estimación de la complejidad computacional del algoritmo que soluciona la ecuación de onda acústica 3D con densidad constante, para más información dirigirse a [18].

2.1 COMPLEJIDAD COMPUTACIONAL DEL ALGORITMO DE MODELADO EN SERIAL

En esta sección calcularemos la complejidad computacional para el algoritmo que soluciona la ecuación de onda acústica 3D con densidad constante, como primer paso presentaremos un pseudo-código de dicho algoritmo y luego presentaremos el cálculo de la complejidad.

El diagrama que se muestra en la Figura 13 es el que soluciona ecuación de onda acústica 3D con densidad constante, este diagrama muestra que el algoritmo necesita pasos extra para dar una correcta solución. Para el cálculo de la complejidad computacional solo se tendrá en cuenta la sección donde se soluciona la ecuación de onda por FDTD.

Figura 13: Diagrama de flujo que soluciona la ecuación de onda.



El modulo "solución de la ecuación de onda acústica" se amplia en el apéndice 5.2.

El primer paso para el cálculo de la complejidad computacional es determinar la función $T(n)$, esta se obtiene a través de contar el número instrucciones con las veces que se repiten, asumiendo que dichas instrucciones se demora un tiempo c_i y sumar dichas constantes. Si se toma el fragmento de algoritmo 1.

Algoritmo 1 Fragmento del algoritmo en serial

```
1: Cálculo del Campo  P^{its+1}_{x,y,z}
2: for (x = 0; x < n; x++) do
3:     for (y = 0; y < n; y++) do
4:         for (z = 0; z < n; z++) do
5:             P^{its+1}_{x,y,z} = -P^{its-1}_{x,y,z} + 2P^{its}_{x,y,z} + ... ; →  c_1
6:         end for
7:     end for
8: end for
9: ...
```

Se puede determinar que la función de $T(n)$ se define como sumar c_1 el numero de veces que se repite según lo determina los `for` donde se encuentra. Esto se muestra en la ecuación (2.5).

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} c_1 + ... \tag{2.5}$$

Cabe resaltar que los otros términos de la ecuación $T(n)$ se obtienen realizando el mismo proceso con el resto del algoritmo que se encuentra en el apéndice 5.2. La función $T(n)$ completa se puede observar la ecuación (2.6), además se debe tener en cuenta que el tamaño de la fronteras absorbentes L es constante. Por ello no se tendrá en cuenta para el cálculo de la complejidad computación.

$$\begin{aligned}
T(n) = &\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} c_1 + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{L-1}(c_2 + c_3) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{L-1}(c_4 + c_5 + c_6 + c_7) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{L-1}(c_8 + c_9) + \\
&\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{L-1}(c_{10} + c_{11} + c_{12} + c_{13}) + \sum_{i=0}^{L-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}(c_{14} + c_{15}) + \sum_{i=0}^{L-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}(c_{16} + c_{17} + c_{18} + c_{19}) + \\
&\sum_{i=0}^{L-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}(c_{20} + c_{21}) + \sum_{i=0}^{L-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}(c_{22} + c_{23} + c_{24} + c_{25}) + \sum_{i=0}^{n-1}\sum_{j=0}^{L-1}\sum_{k=0}^{n-1}(c_{26} + c_{27}) + \\
&\sum_{i=0}^{n-1}\sum_{j=0}^{L-1}\sum_{k=0}^{n-1}(c_{28} + c_{29} + c_{30} + c_{31}) + \sum_{i=0}^{n-1}\sum_{j=0}^{L-1}\sum_{k=0}^{n-1}(c_{32} + c_{33}) + \sum_{i=0}^{n-1}\sum_{j=0}^{L-1}\sum_{k=0}^{n-1}(c_{34} + c_{35} + c_{36} + c_{37}) + \\
&c_{38} + c_{39} + c_{40}
\end{aligned} \tag{2.6}$$

Donde n es el tamaño del modelo en dirección x, y, z y c_i contantes de tiempo por instrucción.

Si tenemos en cuenta la propiedad de las sumatorias que se muestre en la ecuación (2.7).

$$\sum_{i=1}^{n} c = c \cdot n \tag{2.7}$$

Aplicando la propiedad que se muestra en la ecuación (2.7) en la ecuación (2.6) se obtiene que :

$$
\begin{aligned}
T(n) = c_1 \cdot n^3 + (c_2 + c_3) \cdot n^2 \cdot L + (c_4 + c_5 + c_6 + c_7) \cdot n^2 \cdot L + (c_8 + c_9) \cdot n^2 \cdot L \\
+(c_{10} + c_{11} + c_{12} + c_{13}) \cdot n^2 \cdot L + (c_{14} + c_{15}) \cdot n^2 \cdot L + (c_{16} + c_{17} + c_{18} + c_{19}) \cdot n^2 \cdot L + \\
(c_{20} + c_{21}) \cdot n^2 \cdot L + (c_{22} + c_{23} + c_{24} + c_{25}) \cdot n^2 \cdot L + (c_{26} + c_{27}) \cdot n^2 \cdot L \\
+(c_{28} + c_{29} + c_{30} + c_{31}) \cdot n^2 \cdot L + (c_{32} + c_{33}) \cdot n^2 \cdot L + (c_{34} + c_{35} + c_{36} + c_{37}) \cdot n^2 \cdot L + \\
c_{38} + c_{39} + c_{40}
\end{aligned}
\tag{2.8}
$$

Basados en los resultados de la sección anterior se puede deducir que la complejidad computacional es de orden:

$$
T(n) = \Theta(n^3) \tag{2.9}
$$

Esto quiere decir que el tiempo de ejecución crece de forma cúbica a medida que el tamaño del modelo a solucionar crece, además se observa que la tendencia del tiempo de ejecución se ve mayormente afectado por el algoritmo que soluciona la ecuación de onda sin fronteras.

2.2 COMPLEJIDAD COMPUTACIONAL DEL ALGORITMO DE MODELADO EN PARALELO

En esta sección se estimará la complejidad computacional para el algoritmo que soluciona por la ecuación de onda acústica 3D con densidad constante en paralelo.

La dinámica de la ejecución en paralelo se conoce como "dinámica *multithreaded*" la cual se mostró en el capítulo 1, pero cabe resaltar que el objetivo de esta dinámica es "divide y conquistarás" esto significa es al subdividir nuestro problema se resolverá de manera más simple.

Para subdividir las tareas debe tener en cuenta que la dependencia de datos puede hacer que un algoritmo no se a paralelizable, en nuestro algoritmo tenemos ese problema con la variable del tiempo, por ello lo que es posible paralelar es los lazos que recorren el espacio. La subdivisión que se propone es que cada *thread* calcule la solución de un punto específico del espacio, quiere decir que el número de *threads* a lanzar serían $n*n*n$. Cabe resaltar que se asume que tenemos un *device* o sistema con infinitos *threads*.

Para calcular la complejidad computacional del algoritmo en paralelo se definirán unas palabras claves que nos ayudaran a entender el pseudo-código:

- **parallelism**(N): La instrucción se ejecutará en N *threads* al mismo tiempo.
- **sync**: Es colocará una barrera de sincronización para que todos los *threads* paren en un punto del programa y no ejecuten más instrucciones hasta que todos lleguen a dicho punto.

El pseudo-código del algoritmo que soluciona la ecuación de onda acústica 3D con densidad constante en paralelo se muestra en el apéndice 5.2.

El primer paso para calcular la complejidad computacional en paralelo es determinar $T_\infty(n)$, esto se hace contando el número de instrucciones del pseudo-código que se menciono anteriormente y con el peso en tiempo ejecución que se le asigno a cada instrucción, sumar dichas constante. Basado en lo anterior se obtiene que $T_\infty(n)$ es como muestra en la ecuación (2.2). Cabe aclara que el "∞" de la notación hace referencia a que el sistema tiene infinitos *threads*.

$$T_\infty(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_8 + ... \tag{2.10}$$

Como se asumió que el sistema posee infinitos *threads* se obtuvo que cada instrucción solo se ejecutará una única vez con se observa en la ecuación . Entonces se puede sumir que la complejidad computacional es:

$$T_\infty(n) = \Theta(1) \tag{2.11}$$

Basados en las ecuaciones 2.9 y 2.11 se puede encontrar que el nivel de paralelismo de nuestro algoritmo esta dado por la ecuación 2.12 [18].

$$\frac{T_1(n)}{T_\infty(n)} = \frac{\Theta(n^3)}{\Theta(1)} \tag{2.12}$$

Esto significa que si mi algoritmo procesará un modelo de tamaño $100 \times 100 \times 100$ entonces $n = 100$ y el nivel de paralelismo podría ser de 10^6 asumiendo que se tiene un sistema con núcleos infinitos. Este resulta nos muestra que realizar un implementación en paralelo se obtendrá una ganancia en tiempo de ejecución.

Capítulo 3

IMPLEMENTACIÓN DE LA FDTD EN CUDA Y OpenCL

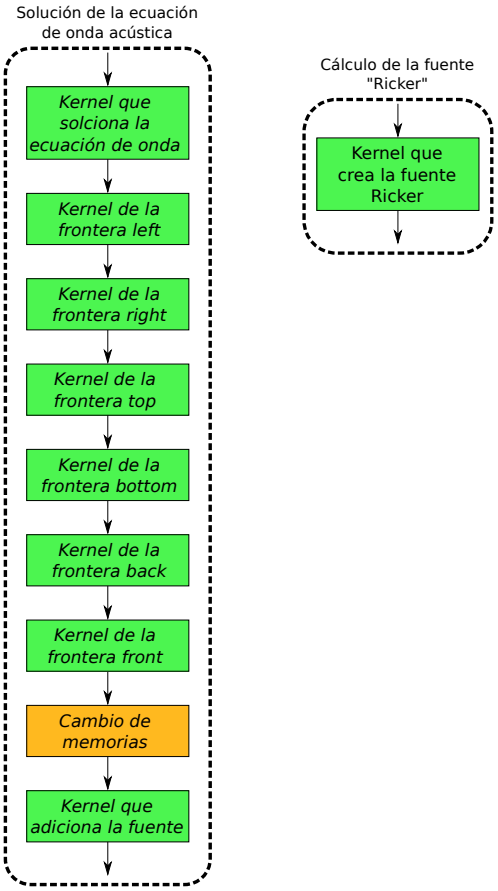
En este capítulo se mostrarán los *kernels* que se crearon para solucionar la ecuación de onda acústica 3D con densidad constante, implementados en tanto en CUDA como en OpenCL.

3.1 IMPLEMENTACIÓN EN CUDA

La implementación en CUDA se hizo en base con el diagrama de la Figura 13 y el algoritmo del apéndice 5.2, para la implementación de dicho algoritmo en CUDA se crearon nueve *kernels* que se muestran en la Figura 14.

Cabe resaltar que de las fronteras absorbentes solo se expondrá el de la frontera *left*, debido a que las otras tiene la misma filosofía pero cambia el sentido espacial de la solución. Además la Figura 14 muestra el orden de como se ejecutan dichos *kernels*, por ello se agrega el modulo de "cambio de memorias" debido a que este es necesario para el modelado.

Figura 14: Lista de *kernels*.



A continuación se mostrará los códigos de los *kernel* con una breve explicación.

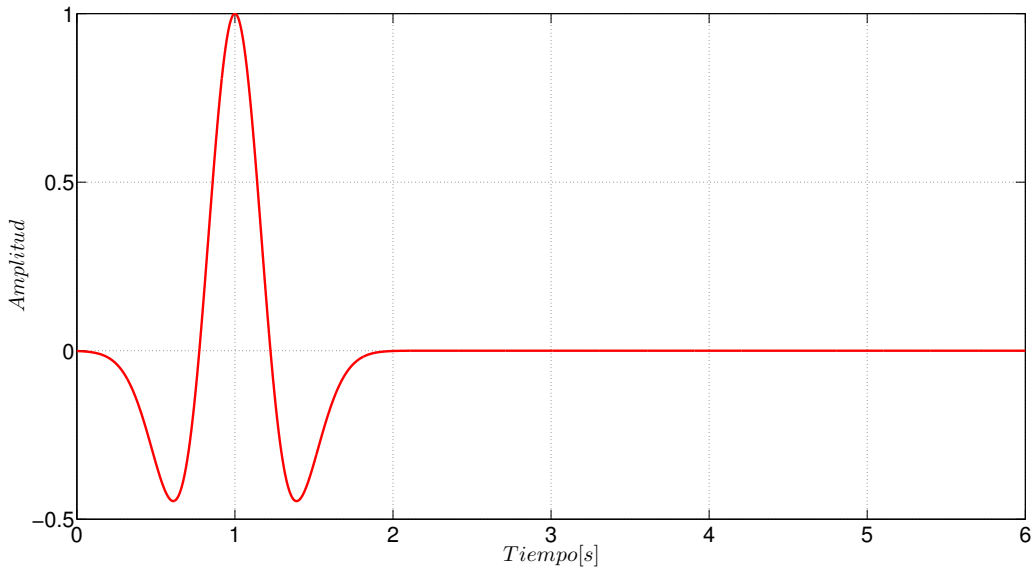
- ***Kernel* que crea la fuente "*Ricker*":**

  La fuente que se utilizó para simular la propagación de onda es la *Ricker* la cual esta definida por la ecuación (3.1) y sus constantes se encuentran especificadas en la Tabla 3.

$$f(t) = 1 - 2(\pi \cdot fq \cdot (t - t_0)))^2 (e^{-(\pi \cdot fq \cdot (t - t_0)))^2}) \tag{3.1}$$

  En la Figura 15 se puede observar la gráfica de la ecuación.

Figura 15: Función *Ricker*.



Cabe resaltar que la fuente se crea en GPU debido a que el aporte se debe adicionar en el campo $P^{its+1}_{x,y,z}$(futuro) y dicho campo esta almacenado allí. Además este *kernel* solo se tiene que lanzar una única vez.

A continuación se presenta la Tabla 3 donde definiremos las variable del *kernel*.

Tabla 3: Variables del *kernel* que crea la fuente.

Variable	**Descripción**
`t0`	Retardo inicial de *Ricker*
`fq`	Frecuencia de la *Ricker*
`dt`	Resolución en el paso de tiempo
`itmax`	Máximo número de pasos de tiempo
`source`	Estructura de la fuente que almacena: `sx`: Posición de la fuente en X `sy`: Posición de la fuente en Y `sz`: Posición de la fuente en Z `data`: Vector con los valores de la fuente

Como se puede observar el *kernel* que crea la *Ricker* se muestra en el cuadro de código 3.1 el cual calcula cada punto de tiempo con un *thread*, cabe destacar que el "`if`" se usa para asegura solo se calculen el número necesario de puntos.

Cuadro de Código 3.1: *Kernel* que crea la fuente "*Ricker*" en CUDA

```
__global__ void ricker_knl(float fq, float dt, int itmax,struct wavelet source) {
  /*Declaring variables*/
  float t, t0, arg;
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  /* Ricker shift (a period of fq)*/
  t0 = 1.0f/fq
  if (i < itmax){
      /* Computing the wavelet vector */
      t = ((float)i - 1.0f)*dt - t0;
      arg = ((float)M_PI)*((float)M_PI)*fq*fq*t*t;
      source.data[it]= (1.0f - 2.0f*arg)*(exp(-arg)); }
}
```

- ***Kernel* que suma el aporte de la fuente "*Ricker*":**

  Este *kernel* como su nombre lo dice es el encargado de sumar el aporte de la fuente al campo $P^{its}_{sx,sy,sz}$, cabe resaltar que "`its`" es la variable que contiene paso se tiempo actual y que "`P_2`" se encuentra enunciada en la Tabla 5. Además en este *kernel* solo se usa un *thread* para realizar dicha operación lo podemos observar en el cuadro de código 3.2. Cabe resaltar que "`Id(..`" es un macro que se encuentra en el cuadro de código 3.3, la idea de este macro es manejar volúmenes de datos de forma tridimensional pero almacenándolos de una forma lineal.

Cuadro de Código 3.2: *Kernel* que suma el aporte de la fuente "*Ricker*" en CUDA

```
__global__ void add_source_gpu(int its, float *P_2, struct wavelet source){aped.A_2
  /* Declaring variables*/
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  if (i < 1){
          Id(P_2,source.sx[i], source.sy[i], source.sz[i]) += source.data[its];}
}
```

- ***Kernel* que soluciona ecuación sin el aporte de las fronteras absorbentes:**

  Este *kernel* como su nombre lo dice es el encargado de soluciona ecuación sin el aporte de las fronteras absorbentes, básicamente lo que hace es calcular el campo $P^{its+1}_{idx,idy,idz}$ con base en la

ecuación (1.14).

A continuación se presenta la Tabla 5 donde definiremos las variable del *kernel*.

Tabla 4: Variables del *kernel* que soluciona ecuación sin el aporte de las fronteras absorbentes.

Variable	**Descripción**
`P_1`	Puntero con el campo de presión $P^{its-1}_{idx,idy,idz}$ (pasado).
`P_2`	Puntero con el campo de presión $P^{its}_{idx,idy,idz}$ (presente).
`c_in`	Puntero con el modelo de velocidades.
`dt`	Resolución en el paso de tiempo.
`Nx, Ny, Nz`	Tamaño del modelo en la direcciones. (x, y, z)
`dx, dy, dz`	Resolución en el paso de espacio en las direcciones (x, y, z).
`coef`	Puntero con los coeficientes de la segunda derivada en diferencia finitas de orden 8.

Para el desarrollo de este *kernel* se crearon cinco macros que ayudaran a la programación, estos se muestran en el cuadro de código 3.3. El primer macro calcula el mínimo entre dos números, el segundo se usa para indexar $P^{its+1}_{idx,idy,idz}$ (Futuro), $P^{its}_{idx,idy,idz}$ (Presente) y $P^{its-1}_{idx,idy,idz}$ (Pasado) en un tramo de memoria continua y los tres siguientes se usan para indexar ψ^{its}_i y ζ^{its}_i en un tramo de memoria continua. Cabe resaltar que se toma z como la variable más rápida debido a que en el modelado sísmico es muy común que los datos se distribuya de esta manera.

Cuadro de Código 3.3: Macros

```
1 #define MIN(x,y) ((x) < (y) ? (x) : (y)) // Calcula el minimo entre "X" y ↩
      "Y"
2 #define Id(A,i,j,k) (A)[(i)*Nz*Ny+(j)*Nz+(k)] // Macro del indexado para ↩
      el modelo completo
3 #define Idlz(A,i,j,k) (A)[Ny*L*(i)+L*(j)+(k)] // Macro del indexado para ↩
      las fronteras top, bottom
4 #define Idlx(A,i,j,k) (A)[Ny*Nx*(i)+Nz*(j)+(k)] // Macro del indexado para↩
       las fronteras left, right
5 #define Idly(A,i,j,k) (A)[Nz*L*(i)+Nz*(j)+(k)] // Macro del indexado para ↩
      las fronteras back, front
```

Para explicar el *kernel* que soluciona la ecuación sin el aporte de las fronteras absorbentes solo se presentaran las partes más relevantes del código y el *kernel* completo se puede observar en el apéndice 5.2.

Lo primero que hace es colocar las variable de entrada al *kernel*, el nombre de el y crea una grilla de *threads* en las tres direcciones "`idx, idy, idz`". Esto se muestra en el cuadro de código 3.4.

Cuadro de Código 3.4: Fragmento del *kernel* que soluciona ecuación de onda en CUDA

```
__global__ void stencil_eval_gpu(int Nx, int Ny, int Nz,/* Model size  */\
                float* P_1,             /* Past field         */\
                float* P_2,             /* Present field      */\
                float* c_in,            /* Velocity field     */\
                float  dt,              /* Time step          */\
                float  dx,              /* Model resolution   */\
                float  dy,              /*     (dx,dy,dz)     */\
                float  dz,              /*                    */\
                float* coef,            /* 2 deriv. FD coef   */\){
    /* Declaring variables */
    int idx  =  threadIdx.x + blockIdx.x*blockDim.x;
    int idy  =  threadIdx.y + blockIdx.y*blockDim.y;
    int idz  =  threadIdx.z + blockIdx.z*blockDim.z;
        {...}
```

Luego se restringe el número de *threads* con el "`if(idx < Nx && idy < Ny && idz < Nz)`", y se crean las variables que junto al macro "`MIN(..`" limitan las derivadas en los bordes del modelo. Esto se muestra en el cuadro de código 3.6.

Cuadro de Código 3.5: Fragmento del *kernel* que soluciona ecuación de onda en CUDA

```
13        {...}
14    /* Constraining threads */
15    if(idx < Nx &&  idy < Ny &&  idz < Nz){
16      int iOrd;
17            int Limxp = Nx - idx - 1, Limxn = idx;
18            int Limyp = Ny - idy - 1, Limyn = idy;
19            int Limzp = Nz - idz - 1, Limzn = idz;
20        {...}
```

Después se calculan las diferencias para cada dirección, esto se hace con un "`for(..`" donde el limite de iteraciones esta dado por el mínimo entre el orden de la diferencia y la Posición en que se encuentra. Esto se muestra solo para la dirección x en el cuadro de código 3.7.

Cuadro de Código 3.6: Fragmento del *kernel* que soluciona ecuación de onda en CUDA

```
20        {...}
21            d2x = 0;  d2y = 0;  d2z = 0;
22      /* Adding the points of the p-branch of X in the stencil */
23  for(iOrd = 0; iOrd<=MIN(4,Limxp); iOrd++){
24      d2x += coef[4-iOrd]*Id(P_2,idx+iOrd,idy,idz);}
25      /* Adding the points of the n-branch of X in the stencil */
26  for(iOrd = 1; iOrd<=MIN(4,Limxn); iOrd++){
27      d2x += coef[4-iOrd]*Id(P_2,idx-iOrd,idy,idz);}
28        {...}
```

Por último se suman todos los aportes para calcular $P^{its+1}_{sx,sy,sz}$ (Futuro). Pero cabe resaltar que el campo $P^{its+1}_{sx,sy,sz}$ (Futuro) se guarda en $P^{its-1}_{sx,sy,sz}$ (Pasado) esto para optimizar el uso de la memoria. Esto se muestra en el cuadro de código 3.7.

Cuadro de Código 3.7: Fragmento del *kernel* que soluciona ecuación de onda en CUDA

```
28          {...}
29       /* Computing the laplacian */
30      Lap = d2x/dx/dx + d2y/dy/dy + d2z/dz/dz;
31              /* Wave equation */
32            Id(P_1,idx,idy,idz)=-Id(P_1,idx,idy,idz)+2*Id(P_2,idx,idy,idz)+dt↩
                  *dt*Id(c_in,idx,idy,idz)*Id(c_in,idx,idy,idz)*lap;
33          }
34  }
```

- ***Kernel* que adiciona el aporte de la frontera *left*:**

  Este *kernel* como su nombre lo dice es el encargado de adicionar el aporte de las fronteras *left* a $P^{its+1}_{sx,sy,sz}$, esto lo hace con las ecuaciones (1.2), (1.16), (1.17), (1.18), (1.19) y (1.20). Además respetando el orden debido a la dependencia de datos.

A continuación se presenta la Tabla 5 donde definiremos las variable del *kernel*.

Tabla 5: Variables del *Kernel* que adiciona el aporte de la frontera *left*.

Variable	**Descripción**
`L`	Tamaño de las fronteras absorbentes.
`a_x, b_x`	Constantes de para el cálculo de ζ_x^{its}.
`a_x_h, b_x_h`	Constantes de para el cálculo de ψ_x^{its}.
`psi_left`	Puntero con el campo ψ_x^{its}.
`zeta_left`	Puntero con el campo ζ_x^{its}.
`coefd1PML`	Puntero con los coeficientes de la primera derivada en diferencia finitas de orden 8.

Para explicar el *kernel* que adiciona el aporte de la frontera *left*, solo se presentaran las partes más relevantes del código y el *kernel* completo se puede observar en el apéndice 5.30. Además se debe tener en cuenta los macros que se muestran en el cuadro de código 3.3.

Lo primero que hace es colocar las variable de entrada al *kernel*, el nombre de el y crea una grilla de *threads* en las tres direcciones "`idx, idy, idz`". Esto se muestra en el cuadro de código 3.8.

Cuadro de Código 3.8: Fragmento del *kernel* que adiciona el aporte de la frontera *left* en CUDA

```
__global__ void apply_cpml_left_gpu(int Nx, int Ny, int Nz,\
            float* P_2,           /* Field  P in t       */\
            float* P_1,           /* Field  P in t-d     */\
            float* c_in,          /* Velocity field      */\
            float  dt,            /* Time step           */\
            float  dx,            /* Spacial resolution  */\
            float  dy,            /*     (dx, dy, dz     */\
            float  dz,            /*                     */\,firstnumber=28
            float* coef,          /* 2 deriv. FD coeff   */\
            int    L,             /* CPML  layers        */\
            float*   a_x,         /* Auxiliar variable a */\
            float*   a_x_h,       /* Auxiliar variable a */\
            float*   b_x,         /* Auxiliar variable b */\
            float*   b_x_h,       /* Auxiliar variable b */\
            float* zeta_left,     /* Recursive var. zeta */\
            float* psi_left ,     /* Recursive var. psi  */\
            float* coefd1PML      /* 1 deriv. FD coeff   */){
    /* Declaring variables*/
    int idx  =  threadIdx.x + blockIdx.x*blockDim.x;
    int idy  =  threadIdx.y + blockIdx.y*blockDim.y;
    int idz  =  threadIdx.z + blockIdx.z*blockDim.z;
        {...}
```

Luego se restringe el número de *thread* con el "`if(idx < L && idy < Ny && idz < Nz)`" pero se debe tener en cuenta que el indexado no se necesita para todo el modelo, solo para el tamaño de la frontera y se crean las variables que junto al macro "`MIN(..`" limitan las derivadas en los bordes del modelo. Esto se muestra en el cuadro de código 3.9.

Cuadro de Código 3.9: Fragmento del *kernel* que adiciona el aporte de la frontera *left* en CUDA

```
        {...}
    int Limxn, Limxp; /* Limits of the branches of the stencil in x, y, z ↩
        */
    int iOrd;  /* Index for the each of the branch */
    float lap;  /* Laplacian */
    float dpsi;  /* Derivate of the field psi */
    float d1x;  /* First order derivates */
    float d2x;  /* Second  order derivates */
    /* Constraining threads */
```

```
30      if(idx < L && idy < Ny &&  idz < Nz){
31          {...}
```

Después se calculan las diferencias para la primera derivada de $P^{its}_{sx,sy,sz}$ en la dirección de x cabe destacar que esta derivada se hace en el punto intermedio para mejorar la exactitud del cálculo, esto se hace con un "`for(..`" donde el limite de iteraciones esta dado por el mínimo entre el orden de la diferencia y la posición en que se encuentra. Esto se muestra en el cuadro de código 3.10.

Cuadro de Código 3.10: Fragmento del *kernel* que adiciona el aporte de la frontera *left* en CUDA

```
31          {...}
32        /*** Psi field ***/
33          /* Updating the psi field */
34          Limxn = idx;   Limxp = Nx-idx-1;
35          /* Computing Initializing derivates */
36          d1x = 0;
37          /* Adding the points of the  p-branch of x stencil */
38          for(iOrd = 0; iOrd<=MIN(3,Limxp-1); iOrd++)
39              d1x += coefd1PML[iOrd]*Id(P_2,idx+iOrd+1,idy,idz);
40          /* Adding the points of the  n-branch of x stencil */
41          for(iOrd = 0; iOrd<=MIN(3,Limxn); iOrd++)
42              d1x -= coefd1PML[iOrd]*Id(P_2,idx-iOrd,idy,idz);
43          /* Computing the laplacian */
44          lap = d1x/dx;
45          {...}
```

Luego se calcula la variable ψ^{its}_x asumiendo $\psi^0_x = 0$, además cabe resaltar que se usa `a_x_h, b_x_h` por que se calcula un punto intermedio y se sincroniza debido a que es necesario que ψ^{its}_x este completamente calculado para poder calcular la derivada de el. Esto se muestra en el cuadro de código 3.11.

Cuadro de Código 3.11: Fragmento del *kernel* que adiciona el aporte de la frontera *left* en CUDA

```
45          {...}
46          /* Field psi left*/
47          Idlx(psi_left,idx,idy,idz) = b_x_h[L-idx-1]*lap + a_x_h[L-idx-1]*↩
                Idlx(psi_left,idx,idy,idz);
```

```
48          /* Synchronizing to computed dPsi */
49          __syncthreads();
50          {...}
```

Después se calcula la primera derivada de ψ_x^{its} en punto medio y la segunda derivada de $P_{sx,sy,sz}^{its}$. Esto se muestra en el cuadro de código 3.12.

Cuadro de Código 3.12: Fragmento del *kernel* que adiciona el aporte de la frontera *left* en CUDA

```
50      {...}
51          /*** dPsi field ***/
52          /* Updating the psi field */
53          Limxn = idx-1;    Limxp = L-idx-1+1;
54          /* Computing Initializing derivates */
55          d1x = 0;
56          /* Adding the points of the  p-branch of x stencil */
57          for(iOrd = 0; iOrd<=MIN(3,Limxp-1); iOrd++)
58              d1x += coefd1PML[iOrd]*Idlx(psi_left,idx-1+iOrd+1,idy,idz);
59          /* Adding the points of the  n-branch of x stencil */
60          for(iOrd = 0; iOrd<=MIN(3,Limxn); iOrd++)
61              d1x -= coefd1PML[iOrd]*Idlx(psi_left,idx-1-iOrd,idy,idz);
62          /* Computing the derivate */
63          dpsi = d1x/dx;
64          /*** Zeta field ***/
65          /* Updating the zeta field */
66          Limxn = idx;    Limxp = Nx-idx-1;
67          /* Computing Initializing derivates */
68          d2x = 0;
69          /* Adding the points of the  p-branch of x stencil */
70          for(iOrd = 0; iOrd<=MIN(4,Limxp); iOrd++)
71              d2x += coef[4-iOrd]*Id(P_2,idx+iOrd,idy,idz);
72          /* Adding the points of the  n-branch of x stencil */
73          for(iOrd = 1; iOrd<=MIN(4,Limxn); iOrd++)
74              d2x += coef[4-iOrd]*Id(P_2,idx-iOrd,idy,idz);
75          /* Computing the laplacian */
76          lap = d2x/dx/dx;
77      {...}
```

Por último se calcula ζ_x^{its} asumiendo $\zeta_x^0 = 0$ y se suma el aporte de esta frontera absorbentes

sobre $P^{its+1}_{sx,sy,sz}$ (futuro), pero cabe resaltar que el campo $P^{its+1}_{sx,sy,sz}$ (futuro) se guarda en $P^{its-1}_{sx,sy,sz}$ (pasado) esto para optimizar el uso de la memoria. Esto se muestra en el cuadro de código 3.13.

Cuadro de Código 3.13: Fragmento del *kernel* que adiciona el aporte de la frontera *left* en CUDA

```
77        {...}
78          /* Field Zeta left*/
79          Idlx(zeta_left,idx,idy,idz) = b_x[L-idx-1]*(lap+dpsi)+a_x[L-idx-1]*↩
                Idlx(zeta_left,idx,idy,idz);
80         /* Wave equation */
81         Id(P_1,idx,idy,idz)+=dt*dt*Id(c_in,idx,idy,idz)*Id(c_in,idx,idy,idz)↩
               *(Idlx(zeta_left,idx,idy,idz)+dpsi);
82      }
83  }
```

3.2 IMPLEMENTACIÓN EN OPENCL

La implementación en OpenCL se hizo en base con el diagrama de la Figura 13 y el algoritmo del apéndice 5.2, pero se debe tener en cuenta que OpenCL tiene unos pasos adicionales los cuales se amplían de forma detallada con el ejemplo de suma de vectores el cual se encuentra en el apéndice 5.2. Además los *kernels* se programaron lo más parecido posible a los de CUDA para realizar una comparación justa entre ellos.
Cabe resaltar que de las fronteras absorbentes solo se expondrá el de la frontera *left*, debido a que las otras tiene la misma filosofía pero cambia el sentido espacial de la solución. Además la Figura 14 muestra el orden de como se ejecutan dichos *kernels*, por ello se agrega el moduló de "cambio de memorias" debido a que este es necesario para el modelado.

A continuación se muestran los códigos de los *kernel* con una breve explicación.

- ***Kernel* que crea la fuente "*Ricker*":**

  En el cuadro de código 3.14 se encuentra el *kernel* que crea la fuente "*Ricker*", como se puede observar es similar al cuadro de código 3.1, con la diferencia que este se debe crea la estructura de la fuente dentro del *kernel* y que la estructura no puede tener la variable `data` debido a que OpenCL no permite el uso de punteros en estructuras [6]. Además los comandos con los que se el indexado es diferente, que se debe agregar la palabra `__global` a todas variables que estén en la memoria global y la palabra con la que arrancaban los *kernels* es `__kernel`.

Cuadro de Código 3.14: *Kernel* que crea la fuente "*Ricker*" en OpenCL

```
typedef struct cl_wavelet {
  int x;       // x source/geophone position
  int y;       // y source/geophone position
  int z;       // z source/geophone position
}cl_wavelet;

__kernel void ricker_knl(float fq, float dt, int itmax, cl_wavelet cl_source,__global float* data) {
  /*Declaring variables*/
  float t, t0, arg;
  int it = get_global_id(0);
  /* Ricker shift (a period of fq)*/
  t0 = 1.0/fq;
    if (it < itmax){
      /* Computing the wavelet vector */
        t = ((float)it - 1.0)*dt - t0;
        arg = ((float)M_PI_F)*((float)M_PI_F)*fq*fq*t*t; //pi2*f2*t2
        data[it]= (1.0 - 2.0*arg)*(exp(-arg));

  }
}
```

- ***Kernel* que suma el aporte de la fuente "*Ricker*":**

  El código del *Kernel* que suma el aporte de la *Ricker* se puede observar en el cuadro de código 3.15, es similar al código 3.2 en CUDA, con la diferencia que la variable `data` esta aparte de la estructura.

Cuadro de Código 3.15: *Kernel* que suma el aporte de la fuente "*Ricker*" en OpenCL

```
__kernel void add_source_gpu(int its,__global float *P_2,cl_wavelet cl_source,__global float *data){
  /* Declaring variables*/
  int i =  get_global_id(0);
  if (i < 1){
          Id(P_2,cl_source.x, cl_source.y, cl_source.z) += data[its];
  }
}
```

- ***Kernel* que soluciona ecuación sin el aporte de las fronteras absorbentes:**

  El código del *kernel* que soluciona ecuación sin el aporte de las fronteras absorbentes se puede observar en 3.16, cabe resaltar es similar al código 5.2 en CUDA por ellos solo se muestra la parte donde es diferente. El código completo se puede observar en el apéndice 5.2.

Cuadro de Código 3.16: Fragmento del *kernel* que soluciona ecuación de onda en OpenCL

```
__kernel void stencil_eval_gpu(int Nx, int Ny, int Nz,/* Model size*/\
               __global float* P_1,    /* Past field         */\
               __global float* P_2,    /* Present field      */\
               __global float* c_in,   /* Velocity field     */\
               float dt,     /* Time step          */\
               float dx,     /* Model resolution   */\
               float dy,   /*     (dx,dy,dz)     */\
               float dz,       /*                    */\
               __global float* coef,   /* 2 deriv. FD coef   */\){
   /* Declaring variables */
   int idx  =  get_global_id(0);
   int idy  =  get_global_id(1);
   int idz  =  get_global_id(2);
      :
      :
```

- ***Kernel* que adiciona el aporte de la frontera *left*:**

  El código del *kernel* que adiciona el aporte de la frontera *left* en OpenCL se puede observar en 3.17, cabe resaltar es similar al código 5.30 en CUDA, por ellos solo se muestra la parte donde es diferente. El código completo se puede observar en el apéndice 5.2.

Cuadro de Código 3.17: Fragmento del *kernel* que adiciona el aporte de la frontera *left* en OpenCL

```
__kernel void apply_cpml_left_gpu(int Nx, int Ny, int Nz,\
           __global float* P_2,         /* Field  P in t       */\
           __global float* P_1,         /* Field  P in t-d     */\
           __global float* c_in,        /* Velocity field      */\
           float  dt,                   /* Time step           */\
```

```
6           float  dx,                    /* Spacial resolution  */\
7           float  dy,                    /*     (dx, dy, dz     */\
8           float  dz,                    /*                     */\
9           __global float* coef,         /* 2 deriv. FD coeff   */\
10          int    L,                     /* CPML  layers        */\
11          __global float*   a_x,        /* Auxiliar variable a */\
12          __global float*   a_x_h,      /* Auxiliar variable a */\
13          __global float*   b_x,        /* Auxiliar variable b */\
14          __global float*   b_x_h,      /* Auxiliar variable b */\
15          __global float*   zeta_left, /* Recursive var. zeta */\
16          __global float*   psi_left,  /* Recursive var. psi  */\
17          __global float* coefd1PML   /* 1 deriv. FD coeff   */){
18
19  /* Declaring variables*/
20  int idx  =  get_global_id(0);
21  int idy  =  get_global_id(1);
22  int idz  =  get_global_id(2);
23      :
24      :
```

Capítulo 4

ANÁLISIS COMPARATIVO EN CUDA Y OpenCL

En este capítulo se presenta el análisis de las diferentes comparaciones que se hicieron entre las implementaciones del algoritmo que soluciona la ecuación de onda por medio de la FDTD en CUDA y OpenCL.

4.1 COMPARACIÓN DE EXACTITUD ENTRE OPENCL Y CUDA CON RESPECTO A UNA SOLUCIÓN EN CPU

En esta sección se muestra cómo se comparó el error numérico entre CUDA y OpenCL con respecto a una solución más exacta. La forma como se hizo la comparación fue creando una solución de la ecuación de onda en C con precisión doble de la cual se guardó toda en binario y luego se guardó la solución para todo el tiempo de simulación que se obtuvo en CUDA y OpenCL, y por último se realizaron las siguientes operaciones:

$$Error_{C\ vs\ CUDA} = \underbrace{P^t_{x,y,z}}_{C} - \underbrace{P^t_{x,y,z}}_{CUDA} \tag{4.1}$$

$$Error_{C\ vs\ OpenCL} = \underbrace{P^t_{x,y,z}}_{C} - \underbrace{P^t_{x,y,z}}_{OpenCL} \tag{4.2}$$

Con los siguientes parámetros de modelado:

Tabla 6: Tabla de parámetros de modelado.

$n = 100$	$\longrightarrow$	*Tamaño del modelo en las direcciones X, Y y Z*
$itmax = 500$	$\longrightarrow$	*Número de pasos de tiempo*
$L = 8$	$\longrightarrow$	*Tamaño de las fronteras absorbentes*
$dx = dy = dz = 10[m]$	$\longrightarrow$	*Resolución del modelo espacial*
$dt = 0{,}002264[s]$	$\longrightarrow$	*Resolución del modelo temporal*
$v = 2000[m/s]$	$\longrightarrow$	*Velocidad del modelado de velocidades*

Se debe tener en cuenta que el modelo de velocidades es constante.

La GPU que se uso para las pruebas fue la GPU NVIDIA Tesla K40m y sus especificaciones se muestran en la Tabla 7.

Tabla 7: Tabla de características de la "Tesla K40m".

Característica	**Descripción**
Total de memoria global ó *global memory*:	11520 MBytes (12079136768 bytes)
(15) Multiprocessors, (192) CUDA Cores/MP:	2880 CUDA Cores
Velocidad del reloj:	745 MHz (0.75 GHz)
Velocidad del reloj de la memoria:	3004 Mhz
Ancho de banda del Bus:	384-bit
Tamaño total de la memoria *cache* L2:	1572864 bytes
Tamaño total de la memoria constate:	65536 bytes
Tamaño total de la memoria compartida por bloque:	49152 bytes
Tamaño total de la registros por bloque:	65536
Número de *threads* por *Warp*:	32
Máximo número de *threads* por *multiprocessor*:	2048
Máximo número de *threads* por bloque:	1024
Tamaño total de la memoria de texturas:	512 bytes

Fuente: Tomado de [27].

Cabe resaltar que este proceso se realizó en C donde se cargaron los tres binarios de las soluciones. Luego de obtener los resultados de las operaciones se procedió a encontrar la forma de la distribución de errores, las cuales se pueden observar en las Figuras 16 y 17.

Figura 16: Distribución de errores entre C y CUDA.

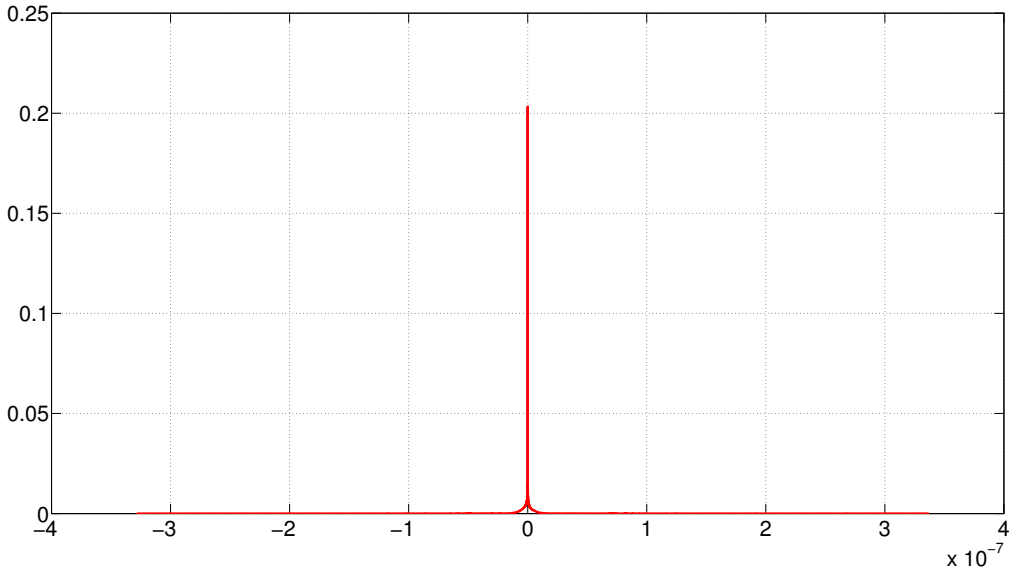
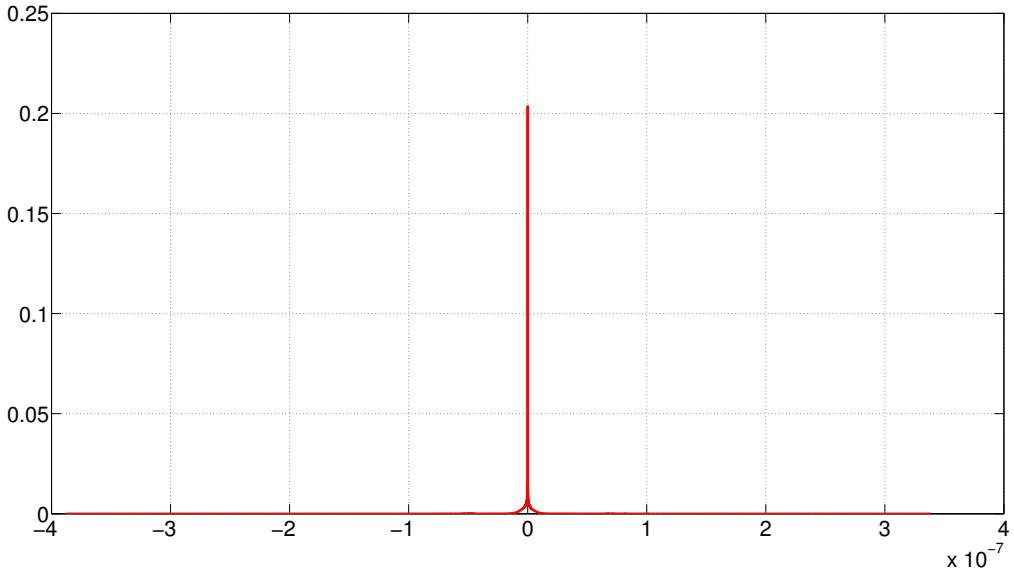


Figura 17: Distribución de errores entre C y OpenCL.



Donde el error máximo y mínimo entre C y CUDA es de:

Tabla 8: Errores máximo y mínimo entre C y CUDA.

Error	**Valor**
Mínimo	$-3{,}28822487 * 10^{-7}$
Máximo	$3{,}37426057 * 10^{-7}$

Y el error máximo y mínimo entre C y OpenCL es de :

Tabla 9: Errores máximo y mínimo entre C y OpenCL.

Error	**Valor**
Mínimo	$-3{,}884271306 * 10^{-7}$
Máximo	$3{,}38498899 * 10^{-7}$

Esto errores son debidos a que el truncamiento se hace diferente en CPU y en GPU según [19]. Una buena métrica de comparación es el error cuadrático medio normalizado (ECMN) el cual esta dado por la ecuación:

$$ECMN = \frac{\|x_i - \hat{x}_i\|_2^2}{\|x_i\|_2^2} \times 100 \tag{4.3}$$

Donde x_i es el valor real, $\hat{x}_i$ es el valor calculado. Los resultados obtenidos del ECMN se muestran en la Tabla 10.

8

Tabla 10: Errores cuadráticos medio normalizados entre C y CUDA, y C y OpenCL.

Error	**Valor [%]**
C y CUDA	$3{,}59665 * 10^{-9}$
C y OpenCL	$3{,}39636 * 10^{-9}$

Como se puede observar el error (ECMN) que se obtiene en OpenCL es 5,56873 % menor que el de CUDA, además cabe resaltar que la solución en CUDA y OpenCL son diferentes a pesar de que se ejecutan sobre la misma GPU, esto se debe a que la operación de división en CUDA y en OpenCL se hace de manera diferente [26].

4.2 COMPARACIÓN DE DESEMPEÑO ENTRE OPENCL Y CUDA

En esta sección se muestra cómo el comportamiento de el tiempo de ejecución a medida que el tamaño de modelo cambia en las implementaciones de en CUDA y OpenCL.

Para las pruebas que se realizaron con los códigos de CUDA y OpenCL se usaron los parámetros de la Tabla 11 y se ejecutaron sobre la GPU NVIDIA Tesla K40m.

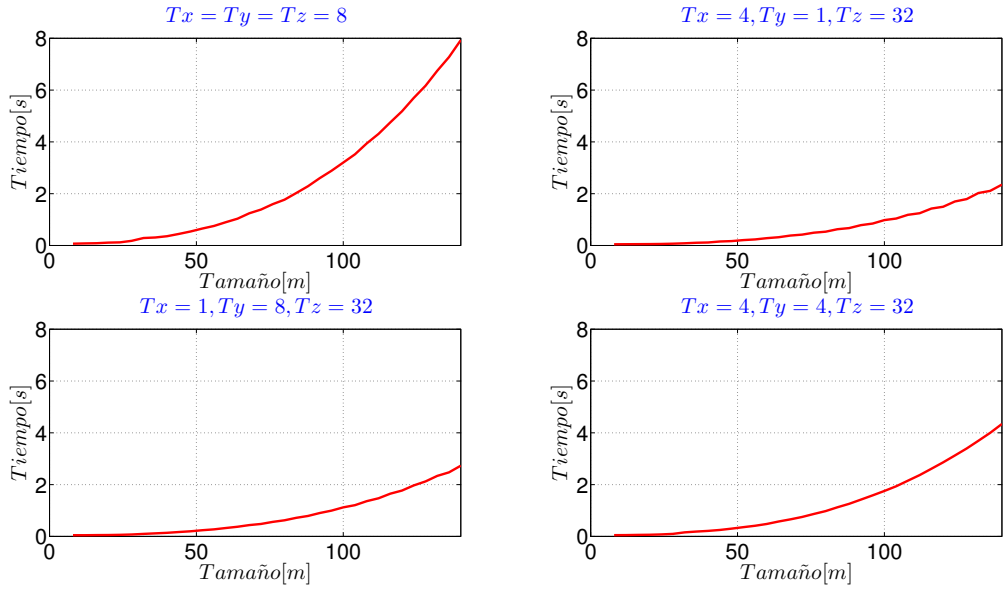
Tabla 11: Tabla de parámetros de modelado

$n = [8 : 4 : 140]$	$\longrightarrow$	*Tamaños de los modelo en las direcciones X, Y y Z*
$itmax = 500$	$\longrightarrow$	*Número de pasos de tiempo*
$L = 8$	$\longrightarrow$	*Tamaño de las fronteras absorbentes*
$dx = dy = dz = 10[m]$	$\longrightarrow$	*Resolución del modelo espacial*
$dt = 0{,}002264[s]$	$\longrightarrow$	*Resolución del modelo temporal*
$v = 2000[m/s]$	$\longrightarrow$	*Velocidad del modelado de velocidades*

Como se puede observar el parámetro n que es el que define el tamaño del modelo varia desde un modelo de $8 \times 8 \times 8$ hasta $140 \times 140 \times 140$ aumentando en pasos de 4 puntos en cada dirección. Además se usa un el modelo de velocidades constante.

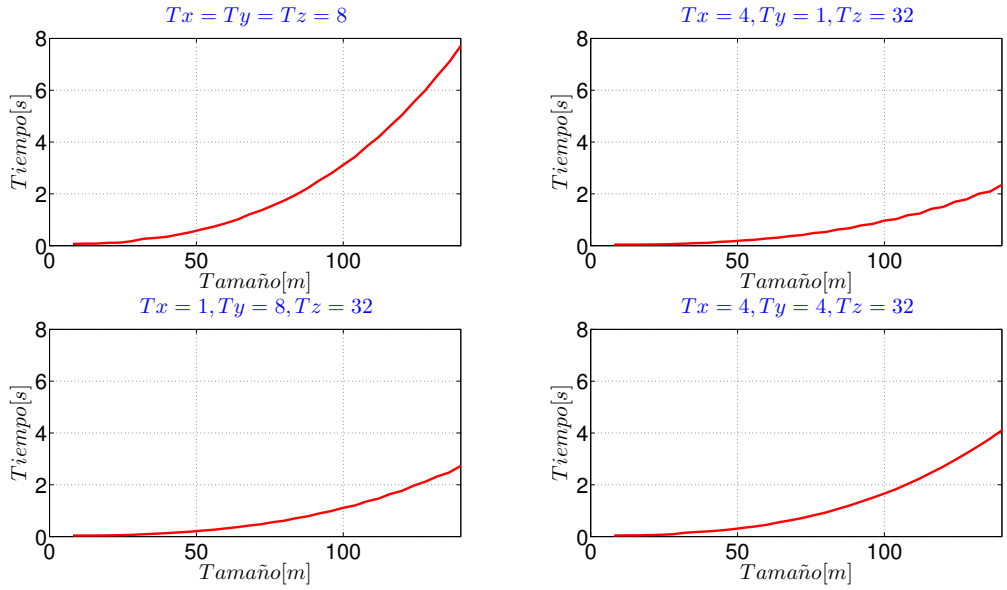
Se debe tener en cuenta que el tiempo de ejecución se ve afectado dependiendo de la configuración de bloques de *threads* que se envíe. En las Figuras 18 se observar el tiempo de ejecución para diferentes configuraciones de bloques y con la variación del tamaño de modelo para CUDA.

Figura 18: Tiempo de ejecución Vs Tamaño de modelo, para la implementación en CUDA.



En las Figuras 19 se observar las mismas pruebas con el código de OpenCL.

Figura 19: Tiempo de ejecución Vs Tamaño de modelo, para la implementación en OpenCL.



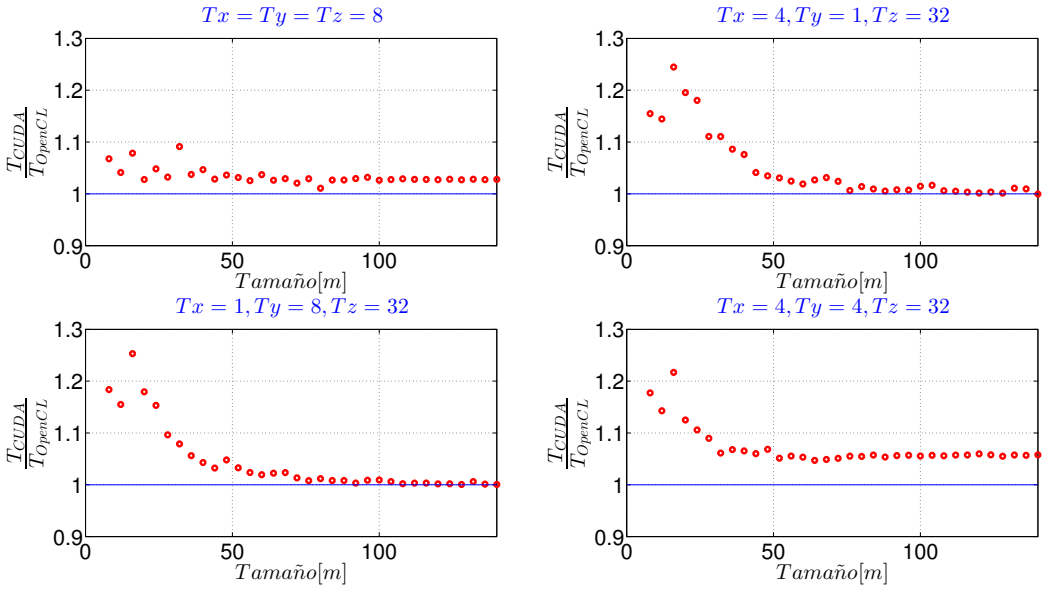
Estas variaciones de tiempo que se muestran en las Figuras 18 y 19 son debido a los accesos *coalesced* por ello para las configuraciones donde se asignan 32 *threads* en dirección z el tiempo es menor.

Si comparamos lo tiempos de ejecución aplicando la siguiente ecuación:

$$\frac{T_{CUDA}}{T_{OpenCL}} \tag{4.4}$$

Los resultados que se obtiene se puede observar en las Figuras 20

Figura 20: Tiempo de ejecución en CUDA sobre tiempo de ejecución en OpenCL Vs Tamaño de modelo.



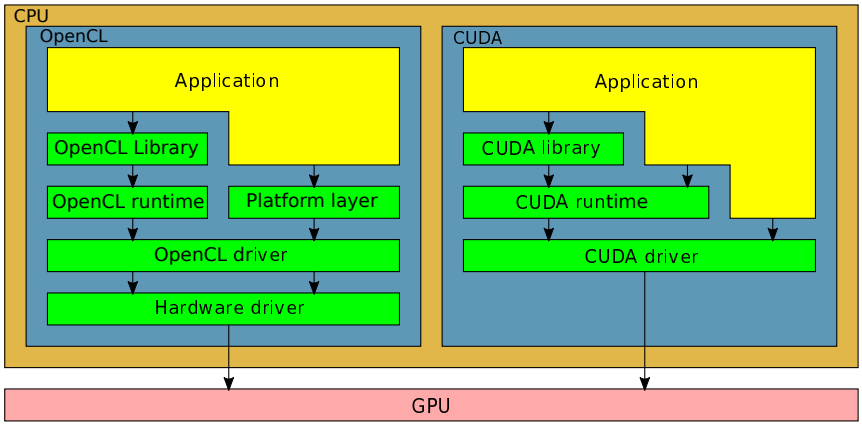
Como se puede observar de la gráfica anterior la implementación tiene un mejor desempeño en OpenCL que en CUDA pero esto es debido a que se programan en diferente niveles de *framework* [24]. Pero se debe tener en cuenta que la toma de tiempo se hace solo para la parte del algoritmo que se muestra en la Figura 13, lo que quiere decir que no tiene en cuenta los tiempos de creación, configuración y copia de memoria entre *host* y *device*.

Cabe aclarar que para cada medida de tiempo se hicieron 500 tomas de tiempo y se verificó su distribución para encontrar la medida más probable.

4.3 COMPARACIÓN DE LA FORMA DE PROGRAMACIÓN

En esta sección se muestra las diferencias que se encuentra entre los modelos de programación CUDA y OpenCL. Se comparan aspectos como los *frameworks* de cada uno, parentesco en la programación de los *kernel*, ventajas y desventajas de su forma de programar.

Figura 21: *Frameworks* de OpenCL y CUDA.



Fuente: Adaptado de [24].

En la Figura 21 se muestra los *frameworks* tanto de CUDA como de OpenCL, de esta imagen se evidencia que existen diferentes APIs para generar una implementación. La primer API es *library*, este consta de las funciones o algoritmos pre-programados por el fabricante. La segunda API es *runtime*, esta contiene todos los comandos necesarios para desarrollar y ejecutar una implementación en GPU, cabe resaltar que en OpenCL no deja crear un programa directamente con la API *runtime*, esto debido a que el necesita información sobre el *device* por lo que es un lenguaje que se puede usar en cualquier *device*. Esto representa una ventaja de CUDA sobre OpenCL por que se pueden generar implementación de una forma más simple. La tercer API es *driver* esta contiene los comandos y los *drivers* con los que se configura la GPU para la ejecución de un programa. Para OpenCL a esta API le agrega la *platform layer* la cual contiene los comandos con los que se reconocen las plataformas que soporta[24].

En nuestra implementación se creo desde el API *runtime* en CUDA y desde la API *device* en OpenCL. Debido a que OpenCL solo permite desarrollo desde este nivel y que en el grupo de investigación de viene trabajando en CUDA de esta forma.

Estos modelos de programación tiene parentesco en la terminología que manejan, se puede observar en Tabla 12 los términos equivalentes para cada uno de ellos [16].

Tabla 12: CUDA vs OpenCL.

Terminología CUDA	Terminología OpenCL
Global Memory	*Global Memory*
Constant Memory	*Constant Memory*
Shared Memory	*Local Memory*
Local Memory	*Private Memory*
Thread	*Work-item*
Thread-block	*Work-group*
Grid	*NDRange*
`__global__`	`__kernel`

Fuente: Adaptado de [16].

Además esta similitud también se puede observar en las Figuras 7 y 11 en el capitulo 1.

A pesar de las similitudes entre la herramientas de programación es más complejo el desarrollo de aplicaciones en OpenCL que en CUDA esto debido a que es multiplataforma. También cabe resaltar que NVIDIA no tiene el mismo número de herramientas de depuración para OpenCL y CUDA, esto representa una gran ventaja al desarrollar en CUDA.

4.4 COMPARACIÓN DE MÉTRICAS

En esta sección se enuncia las distintas herramientas que proporciona NVIDIA para observar el desempeño de nuestras códigos en sus GPUs y se comparar el desempeño del código en CUDA y OpenCL que nos arrojen la herramienta. Para CUDA existen varias herramientas como:

- *Visual Profiling* esta herramienta muestra la forma de ejecución de nuestra implementación, tiempos de ejecución de todos los *kernels* y diferentes métricas de ellos mismos, todo en una forma visual. Además muestra posibles factores que afectan el tiempo de ejecución.
- *Command Line Profiling* esta herramienta muestra la forma de ejecución de nuestra implementación, tiempos de ejecución de todos los *kernels* y diferentes métricas de ellos mismos, todo en una forma escrita. A diferencia con la herramienta anterior esta no muestra posibles factores que afectan el tiempo de ejecución.

Debido a que OpenCL es un lenguaje multiplataforma las herramientas para este son proporcionadas por lo fabricantes de dichas plataformas. Para OpenCL en GPUs NVIDIA sólo existe el *Command Line Profiling*.

Unas de las métricas que puede proporciona el *Command Line Profiling* se muestra en la Tabla 13.

Tabla 13: Tabla de métricas para el *Command Line Profiling*.

Métrica	Descripción
`method`	Esta métrica proporciona el nombre del *kernel* o comando de copia de memoria ejecutado.
`gputime`	Esta métrica proporciona el tiempo de ejecución del *kernel* o comando de copia de memoria. Ese valor se proporciona en precisión sencilla y en microsegundos.
`cputime`	Esta métrica proporciona el tiempo de lanzamiento de los *kernel* o comando de copia de memoria. Ese valor se proporciona en precisión sencilla y en microsegundos.
`occupancy`	Esta métrica es el número de *warps* activos sobre el máximo número de *warps* activos en un SM. Esto valor en el rango de 0 a 1

Fuente: Tomado de [25].

Se debe tener en cuenta que esta herramienta se arroja estos datos en una archivo `.log`. Para estas pruebas se usaron los parámetros de modelado que se muestran en la Tabla 6. Ahora presentaremos los datos que nos arrojaron el *Command Line Profiling*.
Para CUDA se obtuvieron los datos que se muestran en Tabla 14.

Tabla 14: Valores de las métricas obtenidas en la implementación de CUDA.

Nombre del *kernel*	**Tiempo en GPU en** $[us]$	**Tiempo en CPU en** $[us]$	***Occupancy***
`add_source_gpu`	12.37734	2.61321	0.25
`stencil_eval_gpu`	6047.500	3.75788	1
`apply_cpml_left_gpu`	466.6923	2.63779	1
`apply_cpml_right_gpu`	460.1208	2.60443	1
`apply_cpml_top_gpu`	364.2564	2.56698	1
`apply_cpml_bottom_gpu`	366.4738	2.52333	1
`apply_cpml_front_gpu`	456.0621	2.56643	1
`apply_cpml_back_gpu`	458.2655	2.80793	1

Para OpenCL se obtuvieron los datos que se muestran en Tabla 15.

Tabla 15: Valores de las métricas obtenidas en la implementación de OpenCL.

Nombre del *kernel*	**Tiempo en GPU en** $[us]$	**Tiempo en CPU en** $[us]$	***Occupancy***
`add_source_gpu`	3.425539	3.363709	0.25
`stencil_eval_gpu`	5857.304	3.214032	1
`apply_cpml_left_gpu`	466.7004	2.637392	1
`apply_cpml_right_gpu`	469.9575	3.383929	1
`apply_cpml_top_gpu`	419.4673	3.355338	1
`apply_cpml_bottom_gpu`	419.7600	3.374853	1
`apply_cpml_front_gpu`	469.2363	3.412160	1
`apply_cpml_back_gpu`	502.4641	3.727941	1

Si comparamos lo tiempos de ejecución aplicando la siguiente ecuación:

$$\frac{T_{CUDA}}{T_{OpenCL}} \tag{4.5}$$

Se obtiene que :

Tabla 16: Comparación de tiempos ente CUDA y OpenCL.

Nombre del *kernel*	**Comparación de tiempo en GPU**	**Comparación tiempo en CPU**
`add_source_gpu`	3.6132	0.7768
`stencil_eval_gpu`	1.0324	1.1679
`apply_cpml_left_gpu`	1.0065	0.7443
`apply_cpml_right_gpu`	0.9790	0.7696
`apply_cpml_top_gpu`	0.8683	0.7650
`apply_cpml_bottom_gpu`	0.8730	0.7476
`apply_cpml_front_gpu`	0.9719	0.7521
`apply_cpml_back_gpu`	0.9120	0.7532

En la Tabla 16 se puede observar que el tiempo de ejecución en le *kernel* `stencil_eval_gpu` en CUDA es 1,0324 más lento que OpenCL y en el *kernel* `add_source_gpu` es de 3,6132. Pero cabe resaltar que los *kernels* de las fronteras donde CUDA en general es más rápidos que OpenCL. La principal razón por la cual OpenCL tiene un mejor desempeño en ciertos *kernels* que CUDA es debido a que se programan en diferente niveles de *framework* [24].
Cabe aclarar que para cada medida de tiempo se hicieron 500 tomas de tiempo y se verificó su distribución para encontrar la medida más probable.

Otra prueba que se realizó fue a la tomo de tiempo de la implementación completa, en esta se tiene en cuenta todos los pasos que se muestra en la Figura 13 y los resultados obtenidos se muestran en la Tabla 17.

Tabla 17: Tiempos de la implementación completa en CUDA y OpenCL.

Herramienta de Programación	**Tiempo** $[s]$
Implementación en OpenCL	3,81688355
Implementación en CUDA	3,63278926

Y si aplicamos la ecuación (4.5) se encuentra que:

$$\frac{T_{OpenCL} - T_{CUDA}}{T_{OpenCL}} \times 100 = \frac{3{,}81688355 - 3{,}63278926}{3{,}81688355} \times 100 = 4{,}82315\,\% \tag{4.6}$$

Como se puede observar del resultado anterior la implementación tiene un mejor desempeño en CUDA que en OpenCL pero esto es debido a que OpenCL requiere pasos adicionales, como gestión de plataformas y *devices*, compilación de los *kernels*, etc.

Capítulo 5

DISCUSIÓN Y CONCLUSIONES

5.1 DISCUSIÓN

Una de las metas de este proyecto es realizar un análisis comparativo del desempeño de la solución de la ecuación de onda por medio de (FDTD) con las herramientas de programación CUDA y OpenCL. Para ello se realizaron las siguientes comparaciones con el fin de mostrar las ventajas de cada una.

La primer comparación fue la exactitud de la solución con respecto a una solución en C con mayor precisión, de esto se obtuvo que siempre se va obtener un error asociado a la lógica de ejecución según [19], además se obtuvo que OpenCL tiene un 5,56873 % menos de error que CUDA.

La segunda comparación de los tiempos de ejecución del algoritmo que soluciona la ecuación de onda, osea se tomaron como se observa en la Figura 13. Además se varió el tamaño de modelo y las configuraciones de los bloques de *threads* con el fin de mostrar como se afecta el tiempo de ejecución con estos cambios, en la Figura 20 se puede observar que OpenCL obtuvo un mejor desempeño que CUDA en casi todas las configuraciones y tamaños. Este resultado es bastante interesante porque esta función es la parte principal de algoritmos como *Full Waveform Inversion* (FWI) y *Reverse Time Migration* (RTM).

La tercer comparación fue basada en las métricas que nos pueden arrojar las herramientas de análisis de desempeño, para este caso se uso el *command line profiler* y se tomaron tres métricas, la primera es el tiempo de lanzamiento del *kernel* en CPU, la segunda es el tiempo de ejecución del *kernel* y la ultima fue la *ocupancy* de los *kernels*. La comparación de las dos primeras métricas se puede observar en la Tabla 16, se puede observar que CUDA tiene una mejor desempeño en los *kernels* que suman los aportes de las fronteras esto puede ser causado por las optimizaciones que hace el compilador y también se puede observar que OpenCL tiene un mejor desempeño en el *kernel* que soluciona la ecuación de onda sin los aportes de las fronteras esto se debe a que OpenCL se programa desde un nivel mas

bajo. De la *ocupancy* cabe resaltar que tenemos un 100 % en los *kernels* más grandes lo cual evidencia que es una buena forma de implementación ya que tiene el mayor número posible de *warps* activos.

La cuarta y última comparación fue el tiempo total de la aplicación, el objetivo de esta era observa si los pasos extra que contiene OpenCL afectan el tiempo de ejecución, los resultados se muestra en la Tabla 17 y se evidencia que sí lo afecta de una manera considerable.

Un trabajo futuro puede ser adicionar los algoritmos creados en este trabajo en la FWI y RTM, implementar el algoritmo de la solución de ecuación de onda con FDTD en por medio de la CUDA *driver* API y realizan un análisis en complejidad computacional de los algoritmos de FWI y RTM.

Se recomienda para trabajos futuros utilizar el análisis de complejidad para escoger tipo de programación más óptimo de nuestra aplicación y que se tenga en cuenta si se desea mejorar la eficacia global del sistema ó reducir el tiempo de desarrollo a la hora de escoger herramienta de programación.

5.2 CONCLUSIONES

- El flujo de implementación en CUDA es mucho más sencillo que OpenCL, esto es debido a que OpenCL es un lenguaje multiplataforma y requiere pasos adicionales para la ejecución de aplicaciones, pero la programación de los *kernels* es muy similar. Esto permite a los investigadores y desarrolladores elegir un tipo de modelo de programación para controlar la GPU en función de su necesidad, ya sea tanto mejorar la eficacia global del sistema ó reducir el tiempo de desarrollo.
- El análisis de algoritmos por medio de la complejidad computacional nos muestra la ganancia en tiempo de ejecución, que se pudiese obtener al cambiar la implementación de nuestro algoritmo de una forma serial a una forma paralela, a medida que cambia el tamaño del modelo.
- Se encontró que a diferencia que en error cuadrático medio normalizado que se encuentra entra la solución de CPU y GPU es de $3{,}59 * 10^{-11} \sim 3{,}39 * 10^{-11}$. Esto se debe a que el orden en el cual se hacen las operaciones afecta la exactitud.
- La implementación de la solución de la ecuación de onda acústica en OpenCL tiene un 5,56873 % menos de error que la de CUDA, esto se debe a que la operación de división se hace de manera diferente [26].
- La implementación de la solución de la ecuación onda acústica en CUDA obtuvo un 4,82315 % mejor desempeño que la de OpenCL en el tiempo de ejecución completo, esto se debe a que OpenCL tiene varios pasos extra como la consultas de plataforma que soporten OpenCL, la compilación mientras se ejecuta la aplicación, etc.

REFERENCIAS

[1] Ruiz, Alfredo Ghisays, Carlos Alberto Vargas, and Luis Alfredo Montes Vides. "MÉTODO HÍBRIDO DE FRONTERAS NO REFLECTIVAS EN LÍMITES DE MODELOS SÍSMICOS." Revista de la Academia colombiana de ciencias exactes, físicas y naturales 30.115 (2006): 209.

[2] Yu, Yaxin, and Jamesina J. Simpson. "A magnetic field-independent absorbing boundary condition for magnetized cold plasma." IEEE Antennas and Wireless Propagation Letters 10 (2011): 294-297.

[3] Nan, Wang, and Tan Wei. "Application of a new isotropic absorbing boundary condition (ABC) with PML in optimal chiral absorbing condition research." 2008 8th International Symposium on Antennas, Propagation and EM Theory. 2008.

[4] Pasalic, Damir, and Ray McGarry. "Convolutional perfectly matched layer for isotropic and nisotropic acoustic wave equations." 2010 SEG Annual Meeting. Society of Exploration Geophysicists, 2010.

[5] Pérez Solano, C. A. "Two-dimensional near-surface seismic imaging with surface waves: alternative methodology for waveform inversion." PhD, Ecole Nationale Superieure des Mines de Paris, France (2013).

[6] Munshi, Aaftab, et al. "OpenCL programming guide." Pearson Education, 2011.

[7] Fornberg, Bengt. "Generation of finite difference formulas on arbitrarily spaced grids." Mathematics of computation 51.184 (1988): 699-706.

[8] José Manuel Rodríguez Alves, "OpenCL frente a CUDA para análisis de imágenes hiperespectrales en GPU.", UNIVERSIDAD DE EXTREMADURA, Escuela Politécnica Ingeniería en Informática, diciembre, 2011.

[9] Clayton, Robert W., and Björn Engquist. "Absorbing boundary conditions for wave-equation migration." Geophysics 45.5 (1980): 895-904.

[10] Silva Alberto. “Modelado de la Propagación de Onda Acústica Teniendo en Cuenta Condiciones de Frontera Finitas Basado en Esquemas De Diferencias Finitas Dominio Temporales Paralelizados.”, Universidad Industrial de Santander, 2012.

[11] NVIDIA, “CUDA C Programming GUIDE v5.5.”[en línea] 2013. Disponible en: http://www.nvidia.com. [fecha de consulta: Febrero 2016]

[12] NVIDIA, “CUDA C Best Practices GUIDE v5.5.”[en línea] 2013. Disponible en: http://www.nvidia.com. [fecha de consulta: Febrero 2016]

[13] NVIDIA, “NVIDIA’s Next Generation CUDA Compute Architecture Kepler GK110.” [en línea] 2013 . Disponible en: http://www.nvidia.com. [fecha de consulta: Febrero 2016]

[14] E. Lindholm, J. Nickolls, S.Oberman and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, 28(2):39–55, March-April 2008.

[15] Sun, Peiyuan, and Xiaohua Shi. “An OpenCL Approach of Prestack Kirchhoff Time Migration Algorithm on General Purpose GPU.” 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies. IEEE, 2012.

[16] Jianbin Fang, Ana Lucia Varbanescu and Henk Sip “A Comprehensive Performance Comparison of CUDA and OpenCL.”. Parallel and Distributed Systems Group, Delft University of Technology Delft, the Netherlands, 2011.

[17] Sim Jaewoong, Dasgupta Aniruddha, Kim Hyesson, Vuduc Richard, “A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications.”. Georgia Institute of Technology, Power and Performance Optimization Labs, AMD. New Orleans, Louisiana, USA, ACM 2012.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms, Second Edition.” The MIT Press Cambridge , Massachusetts London, England McGraw-Hill Book Company, 2001 by The Massachusetts Institute of Technology.

[19] Whitehead, Nathan, and Alex Fit-Florea. “Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs.” *rn(A+B)* 21 (2011): 1-1874919424.

[20] Savage, John E. “Models of computation.” Exploring the Power of Computing (1998).

[21] Software.intel.com. (2016). Code Samples for Intel®OpenCL™Support — Intel®Software.[en línea] Disponible en: “https://software.intel.com/en-us/intel-opencl-support/code-samples” [fecha de consulta: Octubre 2015].

[22] Khronos.org. (2016). OpenCL 1.1 Reference Pages. [en línea] Disponible en: https://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/ [fecha de consulta: Octubre 2015].

[23] NVIDIA, "CUDA RUNTIME API v8.0'[en línea].Febrero 2016. Disponible en: http://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html [fecha de consulta: Mayo 2016]

[24] Su, Ching-Lung, et al. "Overview and comparison of OpenCL and CUDA technology for GPGPU." Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on. IEEE, 2012.

[25] NVIDIA "COMPUTE COMMAND LINE PROFILER DU-05982-001_v03." [en línea] Noviembre 2011. Disponible en: https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/C/doc/Compute_Command_Line_Profiler_User_Guide.pdf [fecha de consulta: Mayo 2016]

[26] CUDATM, N. "OpenCL Programming Guide for the CUDA Architecture.". NVIDIA Corporation (2009).

[27] NVIDIA "TESLA K40 GPU ACCELERATOR BD-06902-001_v05." [en línea] Noviembre 2013. Disponible en: http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf. [fecha de consulta: Mayo 2016]

BIBLIOGRAFÍA

Clayton, Robert W., and Björn Engquist. "Absorbing boundary conditions for wave-equation migration." Geophysics 45.5 (1980): 895-904.

CUDATM, N. "OpenCL Programming Guide for the CUDA Architecture." NVIDIA Corporation (2009).

E. Lindholm, J. Nickolls, S.Oberman and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, 28(2):39–55, March-April 2008.

Fornberg, Bengt. "Generation of finite difference formulas on arbitrarily spaced grids." Mathematics of computation 51.184 (1988): 699-706.

Jianbin Fang, Ana Lucia Varbanescu and Henk Sip "A Comprehensive Performance Comparison of CUDA and OpenCL.". Parallel and Distributed Systems Group, Delft University of Technology Delft, the Netherlands, 2011.

José Manuel Rodríguez Alves, "OpenCL frente a CUDA para análisis de imágenes hiperespectrales en GPU.", UNIVERSIDAD DE EXTREMADURA, Escuela Politécnica Ingeniería en Informática, diciembre, 2011.

Khronos.org. (2016). OpenCL 1.1 Reference Pages. [en línea] Disponible en: https://www.khronos.org /registry/cl/sdk/1.1/docs/man/xhtml/ [fecha de consulta: Octubre 2015].

Munshi, Aaftab, et al. "OpenCL programming guide.". Pearson Education, 2011.

Nan, Wang, and Tan Wei. "Application of a new isotropic absorbing boundary condition (ABC) with PML in optimal chiral absorbing condition research." 2008 8th International Symposium on Antennas, Propagation and EM Theory. 2008.

NVIDIA "COMPUTE COMMAND LINE PROFILER DU-05982-001_v03." [en línea] Noviembre 2011. Disponible en: https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/C/doc/Compute_Command _Line_Profiler_User_Guide.pdf [fecha de consulta: Mayo 2016]

NVIDIA, "CUDA C Best Practices GUIDE v5.5."[en línea] 2013. Disponible en: http://www.nvidia.com. [fecha de consulta: Febrero 2016]

NVIDIA, "CUDA C Programming GUIDE v5.5."[en línea] 2013. Disponible en: http://www.nvidia.com. [fecha de consulta: Febrero 2016]

NVIDIA, "CUDA RUNTIME API v8.0'[en línea].Febrero 2016. Disponible en: http://docs.nvidia.com /cuda/cuda-runtime-api/api-sync-behavior.html [fecha de consulta: Mayo 2016]

NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture Kepler GK110." [en línea] 2013 . Disponible en: http://www.nvidia.com. [fecha de consulta: Febrero 2016]

NVIDIA "TESLA K40 GPU ACCELERATOR BD-06902-001_v05." [en línea] Noviembre 2013. Disponible en: http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf. [fecha de consulta: Mayo 2016]

Pasalic, Damir, and Ray McGarry. "Convolutional perfectly matched layer for isotropic and nisotropic acoustic wave equations." 2010 SEG Annual Meeting. Society of Exploration Geophysicists, 2010.

Pérez Solano, C. A. "Two-dimensional near-surface seismic imaging with surface waves: alternative methodology for waveform inversion." PhD, Ecole Nationale Superieure des Mines de Paris, France (2013).

Ruiz, Alfredo Ghisays, Carlos Alberto Vargas, and Luis Alfredo Montes Vides. "MÉTODO HÍBRIDO DE FRONTERAS NO REFLECTIVAS EN LÍMITES DE MODELOS SÍSMICOS." Revista de la Academia colombiana de ciencias exactes, físicas y naturales 30.115 (2006): 209. Savage, John E. "Models of computation." Exploring the Power of Computing (1998).

Silva Alberto. "Modelado de la Propagación de Onda Acústica Teniendo en Cuenta Condiciones de Frontera Finitas Basado en Esquemas De Diferencias Finitas Dominio Temporales Paralelizados.", Universidad Industrial de Santander, 2012.

Sim Jaewoong, Dasgupta Aniruddha, Kim Hyesson, Vuduc Richard, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications.". Georgia Institute of Technology, Power and Performance Optimization Labs, AMD. New Orleans, Louisiana, USA, ACM 2012.

Software.intel.com. (2016). Code Samples for Intel®OpenCL™Support — Intel®Software.[en línea] Disponible en: "https://software.intel.com/en-us/intel-opencl-support/code-samples" [fecha de consulta: Octubre 2015].

Su, Ching-Lung, et al. "Overview and comparison of OpenCL and CUDA technology for GPGPU." Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on. IEEE, 2012.

Sun, Peiyuan, and Xiaohua Shi. "An OpenCL Approach of Prestack Kirchhoff Time Migration Algorithm on General Purpose GPU." 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies. IEEE, 2012.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms, Second Edition." The MIT Press Cambridge , Massachusetts London, England McGraw-Hill Book Company, 2001 by The Massachusetts Institute of Technology.

Whitehead, Nathan, and Alex Fit-Florea. "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs." *rn(A+B)* 21 (2011): 1-1874919424.

Yu, Yaxin, and Jamesina J. Simpson. "A magnetic field-independent absorbing boundary condition for magnetized cold plasma." IEEE Antennas and Wireless Propagation Letters 10 (2011): 294-297.

ANEXO A: IMPLEMENTACIÓN SUMA DE VECTORES EN CUDA

A continuación se presentará a modo de ejemplo el código de la suma de dos vectores "*VectorAdd*" `A` y `B` de 2048 elementos y guardando su resultados en un vector `C`,implementado en CUDA, cabe resaltar que este código fue tomado de *CUDA Programming Guide* [11]. Cuando se piensa en programar en paralelo un buen comienzo es considerar la solución serial del problema la cual se muestra en el cuadro de código 5.1.

Cuadro de Código 5.1: Suma de vectores serial

```
....
  for(i=0 ; i<N ;i++){
    C[i] = A[i] + B[i];
  }
....
```

Ahora bien para la implementación de un código en CUDA se recomienda que se deben seguir los siguientes pasos:

Paso 1: Inclusión de librerías.

Paso 2: Definición del *kernel* ó los *kernels*.

Paso 3: Definición de variables del *Host* y *Device*.

Paso 4: Copia de variables a *Device*.

Paso 5: Lanzamiento de *kernels* desde *Host*.

Paso 6: Copia de variables del *Device* al *Host*.

Paso 7: Liberación de memoria.

Paso 1: Inclusión de librerías:

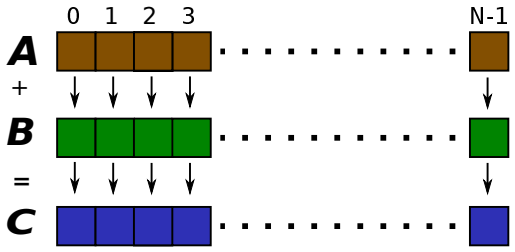
Las librerías que recomendadas para un desarrollo en CUDA son la que se muestra en el cuadro de código 5.2.

Cuadro de Código 5.2: Librerías Recomendadas para Código en CUDA

```
1    #include<stdlib.h>
2    #include<stdio.h>
3    #include<string.h>
4    #include<math.h>
5    #include<time.h>
6    #include<cuda_runtime.h>
```

Paso 2: Definición del *kernel* ó los *kernels*: Para la creación del *kernel* se debe tener en cuenta la solución serial del problema, para nuestro ejemplo se muestra en el cuadro de código 5.1, para paralelizar este problema se propone que cada *thread* calcule la suma de un único elemento como se muestra en la Figura 1.

Figura 1: Suma de vectores.



Fuente: Adaptado de [11].

Basándose en la información anterior se crea el *kernel* que realice lo propuesto, el cual se muestra en el cuadro de código 5.3, se debe tener en cuenta que las funciones para el indexado de los *threads* se muestran en la Tabla 1.

Cuadro de Código 5.3: *Kenrel* de suma de vectores CUDA

```
7    __global__ void VecAdd(float* A, float* B, float* C, int N){
```

```
8     int i = blockDim.x * blockIdx.x + threadIdx.x;
9     if (i < N){
10      C[i] = A[i] + B[i];
11      }
12    }
```

Tabla 1: Tabla de comandos para indexado.

Comando	Descripción
`threadIdx.x`	Genera los índices de los *threads* en la dimensión x
`blockIdx.x`	Genera los índices de los bloques en la dimensión x
`blockDim.x`	Genera el número de bloques que especificados en la dimensión x

Fuente: Tomado de [12].

Paso 3: Definición de variables del *Host* y *Device*:

Para este paso lo primero que se debe tener en cuenta es que la memoria del *Host* y el *Device* son físicamente diferentes, por ello se hace necesario el uso de punteros tanto para *Host* como *Device*. Además se debe tener en cuenta que para reservar la memoria del *Device* se deben usar la función de CUDA la cual se muestra en la Tabla 2.

Tabla 2: Lista de comandos para reservas y copia de memoria en GPU.

Comando	
`cudaMalloc( void** devPtr, size_t size )`	
Argumento	**Descripción**
`devPtr`	Puntero en el cual se reservar memoria.
`size`	Número de *Bytes* de la reserva.

Fuente: Tomado de [12].

Basado en lo anterior se puede encontrar que la forma de inicializar variables se escribe como se muestra en el cuadro de código 5.4.

Cuadro de Código 5.4: Inicializar Variables

```
14  int i ;
15  int N = 2048;
16  size_t size = N * sizeof(float);
17 // Allocate input vectors h_A and h_B in host memory
18  float* h_A = (float*)malloc(size);
19  float* h_B = (float*)malloc(size);
```

```
// Initialize input vectors ...
  for(i = 0; i<N ; i ++){
      h_A[i] = 1.0;
      h_B[i] = 2.0;
      }
// Allocate vectors in device memory
  float* d_A;
  cudaMalloc((void **)&d_A, size);
  float* d_B;
  cudaMalloc((void **)&d_B, size);
  float* d_C;
  cudaMalloc((void **)&d_C, size);
```

Paso 4: Copia de variables a *Device*:

En este paso se deben tener en cuenta la función `cudaMemcpy` la cual se explica en la Tabla 3 y se usan como se muestra en el cuadro de código 5.5.

Tabla 3: Comando para copia de memoria entre GPU y CPU.

Comando	
`cudaMemcpy( void* dst, const void* src, size_t count, cudaMemcpyKind kind)`	
Argumento	**Descripción**
`dts`	Es el puntero de memoria del destino.
`src`	Es la dirección de memoria del fuente de los datos que se quieran copiar.
`count`	Es el número de *Bytes* de memoria que se quiere copiar.
`kind`	Es el tipo de transferencia y exsiten: 1. *cudaMemcpyHostToHost* 2. *cudaMemcpyHostToDevice* 3. *cudaMemcpyDeviceToHost* 4. *cudaMemcpyDeviceToDevice*

Fuente: Tomado de [12].

Cuadro de Código 5.5: Copia de memoria de *Host* a *Device*

```
// Copy vectors from host memory to device memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Paso 5: Lanzamiento de *kernels* desde *Host*: Para este paso se debe tener en cuenta las dimensión y la forma de ejecución del *Kernel*, debido a que se debe especifica el número de *thread per*

block (hilos por bloque) y el número de *block per Grid* (bloques por grilla), pero dependiendo del problema se pueden hacer muchas configuraciones, por ejemplo para el problema que se planteo pueden hacerse estas configuraciones:

Ejemplo	`threadsPerBlock`	`blocksPerGrid`
1	1024	2
2	512	4
3	256	8
4	128	16

Basado en lo anterior y con el comando para lanzar *kernels* que se muestra en la Tabla 4, entonces que el comando se escribe como se muestra en el cuadro de código 5.6.

Tabla 4: Comando para lanzar un *kernel* en GPU.

Comando	
`Kernel_name<<<threadsPerBlock,blocksPerGrid>>>(Argument);`	
Argumento	**Descripción**
`Kernel_name`	Nombre del kernel.
`threadsPerBlock`	Número de hilos por bloque.
`blocksPerGrid`	Número de bloques por grilla.
`Argument`	Punteros, variables o constantes de entrada al *kernel*.

Fuente: Tomado de [12].

Cuadro de Código 5.6: Lanzamiento del kernel

```
35 // Invoke kernel
36   int threadsPerBlock = 256;
37   int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
38   VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

Paso 6: Copia de variables del *Device* al *Host*:

Este paso es similar al 4 pero la copia se hace de *Device* a *Host* para visualizar el resultados y se escribe como se muestra en el cuadro de código 5.7.

Cuadro de Código 5.7: Copia de memoria *Device* a *Host*

```
39 // Copy result from device memory to host memory
40 // h_C contains the result in host memory
```

```
41   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
42   for(i = 0; i<10 ; i++){
43       printf(" h_C[%d]=%f ",i,h_C[i]);
44       }
```

Paso 7: Liberación de memoria:

Este paso es necesario para liberar a memoria de la GPU y CPU y pueda ser usada por otros procesos, los comandos para liberar la memoria en la GPU se muestra en el cuadro de código 5.24.

Cuadro de Código 5.8: Liberación de memoria

```
45 // Free device memory
46   cudaFree(d_A);
47   cudaFree(d_B);
48   cudaFree(d_C);
49 // Free host memory ...
```

En el código completo se encuentra en el apéndice 5.2.

Se debe tener en cuenta que existen otras formas para el manejo de memoria pero la que se mostrara en este ejemplo es la mas simple.

ANEXO B: IMPLEMENTACIÓN SUMA DE VECTORES EN OpenCL

A continuación se presentará a modo de ejemplo el código de la suma de dos vectores "*VectorAdd*" `A` y `B` de 2048 elementos y guardando su resultados en un vector `C`,implementado en OpenCL, cabe resaltar que este código fue creado por cuenta propia pero se tomaron varias fuentes de [21].

Para desarrollar este ejemplo se seguirán los siguientes pasos:

Paso 1: Selección de plataforma y *device*.

Paso 2: Definición del *kernel* ó los *kernels*.

Paso 3: Creación de contexto.

Paso 4: Creación de cola de comandos.

Paso 5: Creación, escritura y lectura de objetos de memoria.

Paso 6: Elementos de compilación y ejecución de los *kernels*.

Paso 7: Liberación de memoria.

Paso 1: Selección de plataforma y *device*:

Dentro del modelo de plataformas se encuentra los comandos que proporciona OpenCL para preguntarle sistema por las plataformas y los *devices* soportados, dichos comandos se muestra en las Tablas 5 y 6.

Antes de enunciar los comando y su uso es necesario definir las variables que requieren dichos comandos, las cuales se muestran en el siguiente cuadro de código 5.9. De dichas variables se puede observar que arrancan con `cl` y que tiene tipo de variables específicas para sus comandos.

Cuadro de Código 5.9: Variables de OpenCL.

```
1    cl_platform_id * platform_id;
2    cl_device_id * device_id;
3    cl_uint ret_num_devices;
4    cl_uint ret_num_platforms;
5    cl_int errNum;
6    cl_int id_plat=1;
7    cl_int id_device=0;
```

Tabla 5: Comando para solicitar el número de plataformas y sus direcciones.

Comando	
`cl_int clGetPlatformIDs (cl_uint num_entries, cl_platform_id * platforms, cl_uint * num_platforms)`	
Argumento	**Descripción**
`num_entries`	Variable que limita número de plataformas que puede retornar el comando. (si se usa "0" entregará todas las que existan).
`platforms`	Puntero donde se guardan las direcciones de las plataformas que soporten OpenCL.(si este argumento es "NULL" el comando retornar de plataformas que soporten OpenCL).
`num_platforms`	Variable donde se almacena el número de plataformas que soporten OpenCL.

Fuente: Tomado de [6].

El comando `clGetPlatformIDs` además de retornar el número de plataformas y sus respectivas direcciones retorna un entero donde se puede evaluar si el comando se ejecuto correctamente o ocurrió algún error. Una de las formas de usar el comando es como se presenta en el código 5.10.

Se debe tener en cuenta que la función `checkErr` fue tomada de [21] y se muestra en el apéndice 5.2, dicha función los que hace es comparar el valor de `errNum` con los posibles errores que puedan ocurrir, esta variable de errores la poseen una gran mayoría de la funciones de OpenCL esto se evidenciará cuando se muestres los demás comandos. Además se usa dos veces el comando para verificar el número de plataformas y con ello crear una reserva de memoria de dicho tamaño, en la cual se podrán almacenar las direcciones de todas las plataformas que soporten OpenCL. Además se usa la función `checkMalloc` la cual fue creada para verificar si la reserva de memoria es correcta y dicha función esta en el apéndice 5.2.

Cuadro de Código 5.10: Código para solictar el número de plataformas y sus direcciones.

```
1    // First, query the total number of platforms
2      errNum = clGetPlatformIDs(0, NULL, &ret_num_platforms);
3             checkErr(errNum,"clGetPlatformIDs --> Number of platforms ")↩
                  ;
4    // Next, allocate memory for the installed platforms, and query
5       platform_id = (cl_platform_id *)malloc(sizeof(cl_platform_id)*↩
            ret_num_platforms);
6       checkMalloc(platform_id,"platform_id");
7    // to get the list.
8       errNum = clGetPlatformIDs(ret_num_platforms, platform_id, NULL);
9                 checkErr(errNum,"clGetPlatformIDs --> platforms ID ");
```

Se hace la aclaración de que se debe escoger a priori un plataforma para la ejecución de el o los *kernel* para ello se desarrollo una función llamada `info_platforms_and_devices_open_cl` y que se encuentra en el apéndice 5.2, además para que una plataforma soporte OpenCL se le debe instalar el respectivo SDK, por ejemplo para este trabajo se utilizo el SDK de OpenCL 1.1 el cual soporta la GPU Tesla K40 de NVIDIA .

Tabla 6: Comando para solicitar el número de *devices* y sus direcciones.

Comando	
`cl_int clGetDeviceIDs (cl_platform_id platform, cl_device_type device_type, cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices)`	
Argumento	**Descripción**
`platforms`	Este argumento es la dirección de la plataforma que se desea usar.
`device_type`	Este argumento especifica el tipo de *device* que se busca y posibles argumentos son: 1. `CL_DEVICE_TYPE_CPU`. 2. `CL_DEVICE_TYPE_GPU`. 3. `CL_DEVICE_TYPE_ACCELERATOR`. 4. `CL_DEVICE_TYPE_DEFAULT`. 5. `CL_DEVICE_TYPE_ALL`.
`num_entries`	Este argumento es una variable que limita número de *devices* que puede retornar el comando. (si se usa "0" entregará todas las que existan).
`devices`	Este argumento es un puntero donde se guardan las direcciones de los *devices* que soporten OpenCL ya sea CPUs, GPUs, etc.(si este argumento es "NULL" el comando retornar el número de *Devices* que soporten OpenCL de dicha plataforma).
`num_devices`	Este argumento es una variable donde se almacena el número de *devices* que soporten OpenCL de dicha plataforma).

Fuente: Tomado de [6].

El comando de `clGetDeviceIDs` es similar al anteriormente mencionado con la diferencia de poder escoger el tipo de *device* y que es necesario escoger la plataforma, esto se puede observar el cuadro de código 5.11, en este se puede observar que maneja la misma filosofía de la comando [6].

Cuadro de Código 5.11: Código para solictar el número de *device* y sus direcciones.

```
1  errNum = clGetDeviceIDs( platform_id[id_plat],CL_DEVICE_TYPE_GPU, 0,↩
       NULL, &ret_num_devices);
2        checkErr(errNum,"clGetDeviceIDs --> Number of device ");
3  // Next, allocate memory for the device platforms, and query
4  device_id = (cl_device_id*)malloc(sizeof(cl_device_id)*ret_num_devices)↩
       ;
5        checkMalloc(device_id,"device_id");
```

```
6   errNum = clGetDeviceIDs(platform_id[id_plat],CL_DEVICE_TYPE_GPU,↩
        ret_num_devices ,device_id, &ret_num_devices);
7     checkErr(errNum,"clGetDeviceIDs --> device ID ");
```

Además se reitera la aclaración de que se debe escoger a priori un *device* para la ejecución de el o los *kernel*.

Paso 2: Definición del *kernel* ó los *kernels*:

Para la creación del *kernel* de nuestro ejemplo *VecAdd* se debe tener en cuenta la solución serial del problema, para nuestro ejemplo es la que se muestra en el cuadro de código 5.1. Una forma de paralelizar este problema se propone que cada *work-item* calcule la suma de un único elemento como se muestra en la Figura 1.
Basándose en la información anterior se crea el *kernel* que realice lo propuesto, el cual se muestra en el código 5.12, se debe tener en cuenta que las funciones para el indexado de los *work-items* se muestra en la Tabla 7.

Cuadro de Código 5.12: *Kernel* de suma de vectores en OpenCL.

```
1  __kernel void vector_add(__global const float *A, __global const float *B↩
       , __global float *C){
2  // Get the index of the current element to be processed
3   int i = get_global_id(0);
4   C[i] = A[i] + B[i];
5  }
```

Tabla 7: Tabla de comandos para indexado.

Comando	Descripción
`get_global_id(0)`	Genera los índices de los *work-items* en la dimensión x.
`get_group_id(0)`	Genera los índices de los *work-grups* en la dimensión x.
`get_num_groups(0)`	Genera el número de *work-grups* en la dimensión x.

Fuente: Tomado de [6].

Paso 3: Creación de Contexto:

Para la creación del contexto se usa el comando de la Tabla 8.

Tabla 8: Comando para la creación de contextos en OpenCL.

Comando	
`cl_context clCreateContext(const cl_context_properties *properties, cl_uint num_devices, const cl_device_id *devices, void (CL_CALLBACK *pfn_notify)(const char *errinfo, const void *private_info,size_t cb,void *user_data), void *user_data,cl_int *errcode_ret);`	
Argumento	**Descripción**
`properties`	Este argumento es una variable que se usa para configurar las propiedades de nuestro contexto como por ejemplo escoger la plataforma donde están el o los *devices* que se de sean usar, si se coloca "0" usará la primer plataforma que contenga los *devices*. Existen otras propiedades que se pueden especificar para más información consultar [6]).
`num_devices`	Variable con el número de *devices* que tendrá el contexto.
`devices`	Puntero con las direcciones de los *devices* que tendrá el contexto.
`(CL_CALLBACK *pfn_notify) (const char *errinfo, const void *private_info, size_t cb,void *user_data)`	Este argumento es una función usad para registrar errores del contexto, se puede consultar de forma asíncrona y sus argumentos son: 1. `errinfo`: Es una cadena de caracteres con el error 2. `private_info` y `cb`: Es un puntero binario con información adicional del error que puede ayudar en la depuración. 3. `user_data`: Es un puntero a los datos suministrados por el usuario. (Si se coloca "NULL" no se almacenaran dicha información)
`user_data`	Este argumento pasa el `user_data` de `pfn_notif` cuando dicha función es llamada.
`errcode_ret`	Este argumento es una variable donde se retorna errores en la creación del contexto, además esta variable se puede comparar con constantes del la librería de OpenCL para conocer más información del error.

Fuente: Tomado de [6].

Se debe tener encenta que las propiedades que se le pueden dar al contexto depende de nuestro *devices* y del SKD de OpenCL que se este manejado. En el cuadro de código 5.13 se muestra como se usa dicho comando para nuestra aplicación *VecAdd*. Además se hace la aclaración de que se debe escoger a priori el o los *device* que tendrá el contexto, debido a ello se crean las variables `id_plat` y `id_device` donde se asigna la plataforma y el *device* ya escogido.

Cuadro de Código 5.13: Código para la creación del contexto.

```
1   cl_context_properties properties[]={CL_CONTEXT_PLATFORM,(↩
        cl_context_properties)platform_id[id_plat],0};
2     cl_context context = clCreateContext(properties,ret_num_devices↩
          ,&device_id[id_device],NULL, NULL, &errNum);
3     checkErr(errNum,"clCreateContext");
```

Como se observa en el cuadro de código anterior se le asigna una única propiedad la cual especifica la plataforma que se usará. Cabe resaltar que se pueden usar más propiedades la cuales se especifican en [22].

Paso 4: Creación cola de comandos:

Para la creación de la cola de comandos se usa el comando de la Tabla 9.

Tabla 9: Comando para la creación de la cola de comandos.

Comando	
`cl_command_queue clCreateCommandQueue (cl_context context, cl_device_id device, cl_command_queue_properties properties, cl_int *errcode_ret)`	
Argumento	**Descripción**
`context`	Este argumento es la variable con el contexto ya creado.
`device`	Este argumento un puntero con las direcciones de los *devices* que tendrá la cola de comandos.
`properties`	Este argumento es una variable se usa para configurar nuestra cola de comandos, como por ejemplo escoger el orden de ejecución en los *devices*, existen otras propiedades que se pueden especificar para más información consultar [6].(si se coloca "0" se crear con la propiedades por defecto).
`errcode_ret`	Este argumento es una variable donde se retorna errores en la creación de la cola de comandos, además esta variable se puede comparar con constantes del librería de OpenCL para conocer más información del error.

Fuente: Tomado de [6].

La utilización del comando anteriormente mencionado para aplicación de *VecAdd*, esto se muestra en el cuadro de código 5.14, como se evidencia se usan las propiedades por defecto del SDK.

Cuadro de Código 5.14: Código para la creación de la cola de comandos.

```
1 cl_command_queue command_queue = clCreateCommandQueue(context, device_id[↩
      id_device], 0, &errNum);
2     checkErr(errNum,"clCreateCommadQueue");
```

Paso 5: Creación, escritura y lectura de objetos de memoria:

Para la manipulación los objetos de memoria existen diferentes comandos, de los cuales se presentaran los tres comúnmente usados y aplicados a nuestra aplicación "*VecAdd*".

El primer comando para analizar es el que se usa para crea los *buffers* el cual se muestra en la Tabla 10, para este comando se debe tener en cuenta que los *buffers* de OpenCL son de tipo `cl_mem` el cual es un tipo de variable propia de la API.

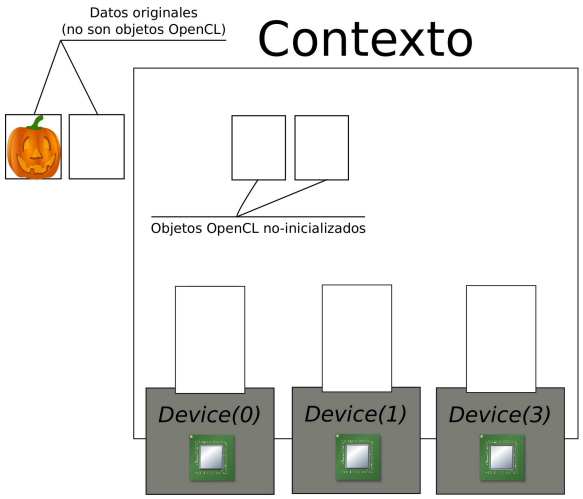
Tabla 10: Comando para la creación de *buffers*.

Comando	
`cl_mem clCreateBuffer (cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)`	
Argumento	**Descripción**
`context`	Este argumento es la variable con el contexto ya creado.
`flags`	Este argumento especifica el tipo de *buffer* que se pueden crear, los posibles argumentos son: 1. `CL_MEM_READ_WRITE`. 2. `CL_MEM_WRITE_ONLY`. 3. `CL_MEM_READ_ONLY`. 4. `CL_MEM_USE_HOST_PTR`. 5. `CL_MEM_ALLOC_HOST_PTR`. 6. `CL_MEM_COPY_HOST_PTR`.
`size`	Este argumento es el tamaño en *Bytes* del *buffer* que se desea reservar.
`host_ptr`	Este argumento es un puntero donde se localizan las direcciones los datos del *buffer* en *host*(Este puntero se usa con *buffers* de tipo: `CL_MEM_USE_HOST_PTR, CL_MEM_ALLOC_HOST_PTR, CL_MEM_COPY_HOST_PTR`).
`errcode_ret`	Este argumento es una variable donde se retorna errores en la creación del *buffer* en el *device*, además esta variable se puede comparar con constantes del librería de OpenCL para conocer más información del error.

Fuente: Tomado de [6].

De este comando se destaca que existen diferentes tipos de *buffers*, para nuestra aplicación se utilizo `CL_MEM_READ_WRITE`, además que dichos *buffers* de memoria están asociados a un contexto y no contiene ningún valor inicial, una extrapolación de este concepto se puede mostrar en la Figura 2.

Figura 2: *Buffers* de memoria en OpenCL.



Fuente: Tomado de [6].

La utilización de dicho comando para nuestra aplicación se muestra en el cuadro de código 5.15, cabe destacar que el `prt_host` se pone `NULL` debido al tipo de *buffer* que se uso.

Cuadro de Código 5.15: Código para la creación de los *buffers* de memoria.

```
1 // Create memory buffers on the device for each vector
2 cl_mem A_d = clCreateBuffer(context, CL_MEM_READ_WRITE,N*sizeof(float), ↩
      NULL, &errNum);
3         checkErr(errNum,"clCreateBuffer --> A_d ");
4 cl_mem B_d = clCreateBuffer(context, CL_MEM_READ_WRITE,N*sizeof(float), ↩
      NULL, &errNum);
5         checkErr(errNum,"clCreateBuffer --> B_d ");
6 cl_mem C_d = clCreateBuffer(context, CL_MEM_READ_WRITE,N*sizeof(float), ↩
      NULL, &errNum);
7         checkErr(errNum,"clCreateBuffer --> C_d ");
```

El segundo comando es con el cual se escriben los *buffers* con los datos necesarios para la aplicación, dicho comando se muestra en la Tabla 11.

Tabla 11: Comando para la escritura del *buffer*.

Comando	
`cl_int clEnqueueWriteBuffer(cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_write, size_t offset, size_t cb, void * ptr, cl_uint num_events_in_wait_list, const cl_event * event_wait_list, cl_event *event)`	
Argumento	**Descripción**
`command_queue`	Este argumento es la variable con la cola de comandos ya creada.
`buffer`	Este argumento es el *buffer* de *device* donde se va escribir la información.
`blocking_write`	Este argumento es un variable la cual especifica si la escritura es *blocking* o *non-blocking*. Si `blocking_write` es `CL_TRUE` la escritura es *blocking* lo cual significa que el comando no retornar hasta que haya finalizado la escritura. Si `blocking_write` es `CL_FALSE` la escritura es *non-blocking* lo que significa que el `command_queue` lea el comando y retorne sin que necesariamente hubiese finalizado la escritura.
`offset`	Este argumento es número de *bytes* donde se quiere que comience a escribir en la memoria.(si se usa "0" comenzara en el inicio).
`cb`	Este argumento es el tamaño en *Bytes* del *buffer* que se desea escribir.
`ptr`	Este argumento es el puntero de *host* donde se encuentran los datos que se desean escribir en el *buffer* de *device*.
`num_events_in_wait_list`	Este argumento es una variable donde se almacena el número de eventos que deben terminar para realizar la escritura del *buffer*.
`event_wait_list`	Este argumento es un puntero con los eventos que deben terminar para realizar la escritura del *buffer*.
`event`	Este argumento es una variable con el evento de salida la cual se activa cuando a finalizado la escritura del *buffer*.

Fuente: Tomado de [6].

Para este comando se debe tener en cuenta que los datos de *host* deben estar asociados a un puntero, además el tamaño de dichos datos debe ser igual o menor al tamaño del *buffer*. La utilización de dicho comando para nuestra aplicación de *VecAdd* se muestra en el cuadro de código 5.16, además se muestra como se inicializan los datos en *host* para nuestra aplicación.

Cuadro de Código 5.16: Código para la escritura de los *buffers* de memoria.

```
// Create the two input vectors
int i;
const int N = 2048;
float *A = (float*)malloc(sizeof(float)*N);
  checkMalloc(A,"A");
float *B = (float*)malloc(sizeof(float)*N);
  checkMalloc(B,"B");
    for(i = 0; i < N; i++){
  A[i]= 1;
  B[i]= 2;
    }
// Copy the lists A and B to their respective memory buffers
errNum = clEnqueueWriteBuffer(command_queue, A_d, CL_TRUE, 0,N*sizeof(↩
    float), A, 0, NULL, NULL);
       checkErr(errNum,"clEnqueueWriteBuffer --> A to A_d ");
errNum = clEnqueueWriteBuffer(command_queue, B_d, CL_TRUE, 0,N*sizeof(↩
    float), B, 0, NULL, NULL);
       checkErr(errNum,"clEnqueueWriteBuffer --> B to B_d ");
```

El tercer y último comando es con el que se leen los *buffers* con los resultados de los *kernels*, dicho comando se muestra en la Tabla 12, este comando es bastante similar al `clEnqueueWriteBuffer`.

Tabla 12: Comando para la lectura de los *buffer* de memoria.

Comando	
`cl_int clEnqueueReadBuffer(cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_read, size_t offset, size_t cb, void * ptr, cl_uint num_events_in_wait_list, const cl_event * event_wait_list, cl_event *event)`	
Argumento	**Descripción**
`command_queue`	Este argumento es la variable con la cola de comandos ya creada.
`buffer`	Este argumento es el *buffer* de *device* donde se va leer la información.
`blocking_write`	Este argumento es un variable la cual especifica si la lectura es *blocking* o *non-blocking*. Si `blocking_write` es `CL_TRUE` la lectura es *blocking* lo cual significa que el comando no retornar hasta que haya finalizado la lectura. Si `blocking_write` es `CL_FALSE` la lectura es *non-blocking* lo que significa que el `command_queue` lea el comando y retorne sin que necesariamente hubiese finalizado la lectura.
`offset`	Este argumento es número de *bytes* donde se quiere que comience a lectura en la memoria.(si se usa "0" comenzara en el inicio).
`cb`	Este argumento es el tamaño en *Bytes* del *buffer* que se desea lectura.
`ptr`	Este argumento es el puntero de *host* donde se encuentran los datos que se desean lectura en el *buffer* de *device*.
`num_events_in_wait_list`	Este argumento es una variable donde se almacena el número de eventos que deben terminar para realizar la lectura del *buffer*.
`event_wait_list`	Este argumento es un puntero con los eventos que deben terminar para realizar la lectura del *buffer*.
`event`	Este argumento es una variable con el evento de salida la cual se activa cuando a finalizado la lectura del *buffer*.

Fuente: Tomado de [6].

Para este comando se debe tener en cuenta que se tener un putero en *host* asociado con un tamaño igual o mayor al tamaño del *buffer* en *device*. La utilización de dicho comando para nuestra aplicación de *VecAdd* se muestra en el cuadro de código 5.17, además se muestra como se inicializa el puntero donde se almacenaran los datos del *device*.

Cuadro de Código 5.17: Código para la lectura de los *buffers* de memoria

```
float *C = (float*)malloc(sizeof(float)*N);
        checkMalloc(C,"C");
errNum = clEnqueueReadBuffer(command_queue, C_d, CL_TRUE, 0,N*sizeof(↩
    float), C, 0, NULL, NULL);
        checkErr(errNum,"clEnqueueReadBuffer --> C_d to C ");
```

Paso 6: Elementos de compilación y ejecución de los *kernels*:

En nuestra aplicación de "*VecAdd*" el *kernel* esta escrito en un archivo diferente al del código de *host*, el cual tiene un nombre "vector_add_kernel.cl" esto debido a que así lo recomienda en *OpenCL Programming Guide* [6], entonces se hace necesario leer dicho archivo en el programa de *host* lo cual se muestra en el cuadro de código 5.22.

Cuadro de Código 5.18: Código para la lectura del archivo del *kernel*

```
// Load the kernel source code into the array source_str
FILE *fp;
char *source_str;
size_t source_size;
    fp = fopen("vector_add_kernel.cl", "r");
    if (!fp) {
        fprintf(stderr, "Failed to load kernel.\n");
        return 0;
    }
    source_str = (char*)malloc(MAX_SOURCE_SIZE);
    source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
    fclose( fp );
```

Luego de leer el archivo y guardar como una cadena de caracteres se procede a cargar el archivo al contexto en forma de programa, para ello se usa el comando que se muestra en la Tabla 13. Con este comando se debe tener en cuenta que se pueden cargar diferentes cadenas de caracteres (diferentes archivos ".cl") con diferentes *kernels* o una cadena de caracteres con diferentes *kernels* al mismo tiempo.

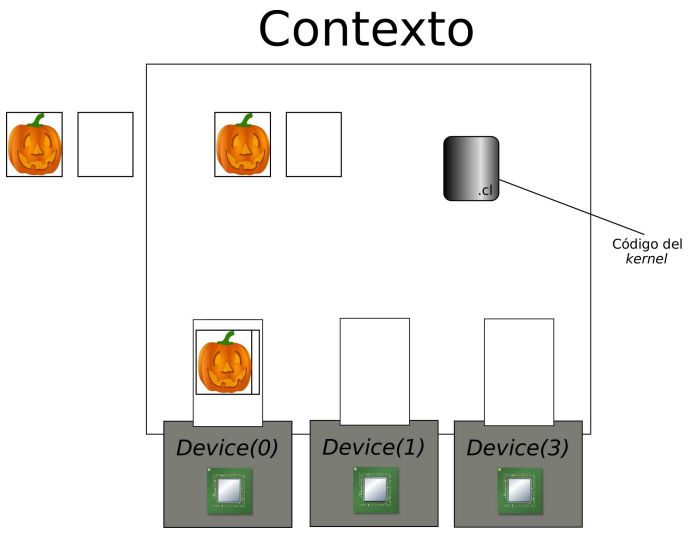
Tabla 13: Creación de programa.

Comando	
`cl_program clCreateProgramWithSource(cl_context context, cl_uint count, const char **strings, const size_t *lengths, cl_int *errcode_ret)`	
Argumento	**Descripción**
`context`	Este argumento es la variable con el contexto ya creado.
`count`	Este argumento el el número de arreglos que se quieran leer.
`strings`	Este argumento es un arreglo de la o las cadenas de caracteres donde esta escrito el o los *kernels*.
`lengths`	Este argumento un arreglo de los tamaños de las cadenas de caracteres que contiene el o los *kernels*.
`errcode_ret`	Este argumento es una variable donde se retorna errores en la lectura *kernels*, además esta variable se puede comparar con constantes del librería de OpenCL para conocer más información del error.

Fuente: Tomado de [6].

Una forma de ver este concepto gráficamente se muestra en la Figura 3.

Figura 3: Carga de archivo ".cl" al contexto.



Fuente: Tomado de [6].

La forma de uso del comando para nuestra aplicación *VecAdd* se muestra en el cuadro de código 5.19.

Cuadro de Código 5.19: Código para carga el archivo ".cl.ªl contexto

```
1 // Create a program from the kernel source
2 cl_program program = clCreateProgramWithSource(context,1,(const char **)&↩
      source_str, (const size_t *)&source_size, &errNum);
3             checkErr(errNum,"clCreateProgram");
```

El segundo paso es la compilación de los *kernels* y creación del binario, en este paso se usa el comando que se muestra en la Tabla 14, como se observa en dicha tabla es necesario el o los *devices* donde se van ejecutar los *kernels* , el archivo cargado y además se deben especificar las opciones de compilación y macros usen *kernels*.

Tabla 14: Comando para la compilación de los *kernels*.

Comando	
`cl_int clBuildProgram(cl_program program, cl_uint num_devices, const cl_device_id *device_list, const char *options, void (CL_CALLBACK *pfn_notify) (cl_program program, void *user_data), void *user_data)`	
Argumento	**Descripción**
`program`	Este argumento es la variable que retorna el comando `clCreateProgramWithSource`.
`num_devices`	Este argumento es una variable con el número de *devices* que ejecutaran el o los *kernels*.
`device_list`	Este argumento es un puntero con las direcciones de el o los *devices* asociados al programa que ejecutaran el o los *kernels*.(si se usa "NULL" ejecutaran con los *devices* asociados al programa).
`options`	Este argumento es una cadena de caracteres que almacena las opciones de compilación, macros y directivas para el o los *kernels*.
`(CL_CALLBACK *pfn_notify) (const char *errinfo, const void *private_info, size_t cb,void *user_data)`	Este argumento es una función usad para registrar errores de la compilación del *kernel*, se puede consultar de forma asíncrona y sus argumentos son: 1.`errinfo`: Es una cadena de caracteres con el error 2.`private_info` y `cb`: Es un puntero binario con información adicional del error que puede ayudar en la depuración. 3.`user_data`: Es un puntero a los datos suministrados por el usuario. (Si se coloca "NULL" no se almacenaran dicha información).
`user_data`	Este argumento pasa el `user_data` de `pfn_notif` cuando dicha función es llamada.

Fuente: Tomado de [6].

La forma de uso del comando para nuestra aplicación *VecAdd* se muestra en el cuadro de código 5.20, este comando se complementa con el uso de `clGetProgramBuildInfo` esto para que en caso de que la compilación no se pueda complementar el programa nos arroje los problemas de la compilación.

Cuadro de Código 5.20: Código para la compilación del programa

```
1    // Build the program
2      errNum = clBuildProgram(program,1,&device_id[id_device], NULL, NULL↩
           , NULL);
3    if ( errNum != CL_SUCCESS) {
4      printf("error build program\n");
5      // Determine the reason for the error
6      char buildLog[16384];
7      clGetProgramBuildInfo(program, device_id[id_device], ↩
           CL_PROGRAM_BUILD_LOG,sizeof(buildLog), buildLog, NULL);
8      printf("ERROR:\t\t%s \n",buildLog);
9      clReleaseProgram(program);
10   }
```

El tercer paso es la creación de cada kernel a partir del programa ya compilado, esto se hace a través del comando que se muestra en la Tabla 15.

Tabla 15: Comando para crear los *kernels*.

Comando	

`cl_kernel clCreateKernel(cl_program program,`
`const char *kernel_name,`
`cl_int *errcode_ret)`	

Argumento	**Descripción**
`program`	Este argumento es la variable que retorna el comando `clCreateProgramWithSource`.
`kernel_name`	Este argumento es una cadena de caracteres con el nombre del *kernel*. Dicho nombre es el que esta inmediatamente después de `__kernel`.
`errcode_ret`	Este argumento es una variable donde se retorna errores en la creación de los *kernels* , además esta variable se puede comparar con constantes del librería de OpenCL para conocer más información del error.

Fuente: Tomado de [6].

La forma de uso del comando para nuestra aplicación *VecAdd* se muestra en el cuadro de código 5.21.

Cuadro de Código 5.21: Código para la creación del *kernel*

```
1 // Create the OpenCL kernel
2 cl_kernel kernel_add_vector = clCreateKernel(program, "vector_add", &↩
    errNum);
3       checkErr(errNum,"clCreateKernel --> kerne_add_vector ");
```

Luego de la creación de los *kernels* el siguiente paso es el lanzamiento del o los *kernels* esto se hace son los dos siguientes pasos.

El primer paso es la asignación de argumentos al o los *kernel*, esto se realiza a través del comando de la Tabla 16, para este comando se debe tener en cuenta el número, orden y tipo de variable de cada argumento.

Tabla 16: Comando para asignación de argumentos al o los *kernel*.

Comando	
`cl_int clSetKernelArg (cl_kernel kernel, cl_uint arg_index, size_t arg_size, const void *arg_value)`	
Argumento	**Descripción**
`kernel`	Este argumento es el *kernel* que se crea con el comando `clCreateKernel`
`arg_index`	Este argumento es un variable con el índice de los argumentos del *kernel*, esta variable va desde "0" hasta "$n-1$" donde "n" es el número de argumentos del *kernel*.
`arg_size`	Este argumento es el tamaño en *bytes* del tipo de variable que es argumento del *kernel*.
`arg_value`	Este argumento es el argumento del *kernel*.

Fuente: Tomado de [6].

La forma de uso del comando para nuestra aplicación *VecAdd* se muestra en el cuadro de código 5.21.

Cuadro de Código 5.22: Código para la lectura del archivo del *kernel*

```
1     // Set the arguments of the kernel
2       errNum = clSetKernelArg(kernel_add_vector, 0, sizeof(cl_mem), (void↩
            *)&A_d);
3     checkErr(errNum,"clSetKernelArg --> A_d  ");
4       errNum = clSetKernelArg(kernel_add_vector, 1, sizeof(cl_mem), (void↩
            *)&B_d);
```

```
5     checkErr(errNum,"clSetKernelArg --> B_d ");
6       errNum = clSetKernelArg(kernel_add_vector, 2, sizeof(cl_mem), (void↩
            *)&C_d);
7     checkErr(errNum,"clSetKernelArg --> C_d ");
```

El segundo paso es la puesta en cola del *kernel* que se quiera ejecutar, esto se hace con el comando que se muestra en la Tabla 17, se debe tener en cuenta que el nombre de *kernel* es el que se puso en el comando `clCreateKernel`.

Tabla 17: Comando para lanzamiento del *kernel*.

Comando	
`cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue, cl_kernel kernel, cl_uint work_dim, const size_t *global_work_offset, const size_t *global_work_size, const size_t *local_work_size, cl_uint num_events_in_wait_list, const cl_event * event_wait_list, cl_event *event)`	
Argumento	**Descripción**
`command_queue`	Este argumento es la variable con la cola de comandos ya creada.
`kernel`	Este argumento es el *kernel* que se crea con el comando `clCreateKernel`.
`work_dim`	Este argumento indica al *kernel* le número de dimensiones que manejara, este número es mayor que "0" y menor que `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS`.
`global_work_offset`	Este argumento es un número donde se indica el comienzo del indexado de los *work-items* en el *kernel*.(si se usa "0" comenzara con el)
`global_work_size`	Este argumento en un vector con el tamaño del *NDRange* donde cada elemento del vector es el tamaño por dimensión.
`local_work_size`	Este argumento en un vector con el tamaño de los *Work-grups* donde cada elemento del vector es el tamaño por dimensión.
`num_events_in_wait_list`	Este argumento es una variable donde se almacena el número de eventos que deben terminar para realizar la ejecución del *kernel*.
`event_wait_list`	Este argumento es un puntero con los eventos que deben terminar para realizar la ejecución del *kernel*.
`event`	Este argumento es una variable con el evento de salida la cual se activa cuando a finalizado la ejecución del *kernel*.

Fuente: Tomado de [6].

La forma de uso del comando para nuestra aplicación *VecAdd* se muestra en el cuadro de código 5.23, cabe destacar que es para este comando se debe especificar el número de *work-*

itens por *work-grup* y el número total de *work-itens* en la *NDRange*, esto puede ser variable pero no afecta la solución pero si el tiempo de ejecución. A continuación se muestra variaos ejemplos de configuraciones :

Ejemplo	`global_item_size`	`local_item_size`
1	2048	2
2	2048	4
3	2048	8
4	2048	16

Como se observa `global_item_size` siempre mantiene el mismo valor esto debido a que el tamaño de los vectores es igual.

Cuadro de Código 5.23: Código para lanzamiento del *kernel*

```
1    // Execute the OpenCL kernel on the list
2    size_t global_item_size = N; // Process the entire lists
3    size_t local_item_size = Work_grup_dim; // Divide work items into ↩
         groups of
4      errNum = clEnqueueNDRangeKernel(command_queue, kernel_add_vector,1,↩
           NULL,&global_item_size, &local_item_size, 0, NULL, NULL);
5    checkErr(errNum,"cl_run_kernel");
```

Paso 7: Liberación de memoria:

Este paso es necesario para liberar a memoria de la GPU y CPU y pueda ser usada por otros procesos, los comandos para liberar la memoria en la GPU se muestra en el cuadro de código 5.24.

Cuadro de Código 5.24: Liberación de memoria

```
1    errNum = clFlush(command_queue);
2    errNum = clFinish(command_queue);
3    errNum = clReleaseKernel(kernel_add_vector);
4    errNum = clReleaseProgram(program);
5    errNum = clReleaseMemObject(A_d);
6    errNum = clReleaseMemObject(B_d);
7    errNum = clReleaseMemObject(C_d);
8    errNum = clReleaseCommandQueue(command_queue);
```

```
9     errNum = clReleaseContext(context);
```

Para finalizar con el ejemplo *VecAdd* se presenta en código de *host* completo en 5.2 y con su *kernel* en 5.12, además en el anexo 5.2 se presenta como compilar y ejecutar.

ANEXO C:
CÓDIGOS Y ALGORITMOS

ALGORITMO QUE SOLUCIONA LA ECUACIÓN DE ONDA DE FORMA SERIAL

1: *Cálculo del Campo* $P_{x,y,z}^{its+1}$
2: **for** $(x = 0;\ x < n;\ x++)$ **do**
3: **for** $(y = 0;\ y < n;\ y++)$ **do**
4: **for** $(z = 0;\ z < n;\ z++)$ **do**
5: $P_{x,y,z}^{its+1} = -P_{x,y,z}^{its-1} + 2P_{x,y,z}^{its} + ... ; \rightarrow\ c_1$
6: **end for**
7: **end for**
8: **end for**
9: *Cálculo de las fronteras absorbentes para en campo* $P_{x,y,z}^{its+1}$
10: *Cálculo de la fronteras TOP*
11: **for** $(x = 0;\ x < n;\ x++)$ **do**
12: **for** $(y = 0;\ y < n;\ y++)$ **do**
13: **for** $(z = 0;\ z < L;\ z++)$ **do**
14: $\frac{\partial p(x,y,z,t)}{\partial z} = ... ; \rightarrow\ c_2$
15: $\psi_z^{its} = ...; \rightarrow\ c_3$
16: **end for**
17: **end for**
18: **end for**
19: **for** $(x = 0;\ x < n;\ x++)$ **do**
20: **for** $(y = 0;\ y < n;\ y++)$ **do**
21: **for** $(z = 0;\ z < L;\ z++)$ **do**
22: $\frac{\partial \psi_z}{\partial z} = ...; \rightarrow\ c_4$
23: $\frac{\partial^2 p(x,y,z,t)}{\partial z^2} = ...; \rightarrow\ c_5$

24: $\zeta_z^{its} = ...; \rightarrow\ c_6$

25: $P_{x,y,z}^{its+1} += ...; \rightarrow\ c_7$

26: **end for**

27: **end for**

28: **end for**

29: *Cálculo de la fronteras BOTTOM*

30: **for** $(x\ =\ 0;\ x\ <\ n;\ x++)$ **do**

31: **for** $(y\ =\ 0;\ y\ <\ n;\ y++)$ **do**

32: **for** $(z\ =\ n-L-1;\ z\ <\ n;\ z++)$ **do**

33: $\frac{\partial p(x,y,z,t)}{\partial z} = ... ; \rightarrow\ c_8$

34: $\psi_z^{its} = ...; \rightarrow\ c_9$

35: **end for**

36: **end for**

37: **end for**

38: **for** $(x\ =\ 0;\ x\ <\ n;\ x++)$ **do**

39: **for** $(y\ =\ 0;\ y\ <\ n;\ y++)$ **do**

40: **for** $(z\ =\ n-L-1;\ z\ <\ n;\ z++)$ **do**

41: $\frac{\partial \psi_z}{\partial z} = ...; \rightarrow\ c_{10}$

42: $\frac{\partial^2 p(x,y,z,t)}{\partial z^2} = ...; \rightarrow\ c_{11}$

43: $\zeta_z^{its} = ...; \rightarrow\ c_{12}$

44: $P_{x,y,z}^{its+1} += ...; \rightarrow\ c_{13}$

45: **end for**

46: **end for**

47: **end for**

48: *Cálculo de la fronteras LEFT*

49: **for** $(x\ =\ 0;\ x\ <\ L;\ x++)$ **do**

50: **for** $(y\ =\ 0;\ y\ <\ n;\ y++)$ **do**

51: **for** $(z\ =\ 0;\ z\ <\ n;\ z++)$ **do**

52: $\frac{\partial p(x,y,z,t)}{\partial x} = ... ; \rightarrow\ c_{14}$

53: $\psi_x^{its} = ...; \rightarrow\ c_{15}$

54: **end for**

55: **end for**

56: **end for**

57: **for** $(x\ =\ 0;\ x\ <\ L;\ x++)$ **do**

58: **for** $(y\ =\ 0;\ y\ <\ n;\ y++)$ **do**

59: **for** $(z\ =\ 0;\ z\ <\ n;\ z++)$ **do**

60: $\frac{\partial \psi_x}{\partial x} = ...; \rightarrow\ c_{16}$

61: $\frac{\partial^2 p(x,y,z,t)}{\partial x^2} = ...; \rightarrow\ c_{17}$

62: $\zeta_x^{its} = ...; \rightarrow\ c_{18}$

63: $P_{x,y,z}^{its+1} += ...; \rightarrow\ c_{19}$

64: **end for**

65: **end for**

66: **end for**

67: *Cálculo de la fronteras RIGHT*

68: **for** $(x\ =\ n-L-1;\ x\ <\ n;\ x++)$ **do**

69: **for** $(y\ =\ 0;\ y\ <\ n;\ y++)$ **do**

70: **for** $(z\ =\ 0;\ z\ <\ n;\ z++)$ **do**

71: $\frac{\partial p(x,y,z,t)}{\partial x} = ... ; \rightarrow\ c_{20}$

72: $\psi_x^{its} = ...; \rightarrow\ c_{21}$

73: **end for**

74: **end for**

75: **end for**

76: **for** $(x\ =\ n-L-1;\ x\ <\ n;\ x++)$ **do**

77: **for** $(y\ =\ 0;\ y\ <\ n;\ y++)$ **do**

78: **for** $(z\ =\ 0;\ z\ <\ n;\ z++)$ **do**

79: $\frac{\partial \psi_x}{\partial x} = ...; \rightarrow\ c_{22}$

80: $\frac{\partial^2 p(x,y,z,t)}{\partial x^2} = ...; \rightarrow\ c_{23}$

81: $\zeta_x^{its} = ...; \rightarrow\ c_{24}$

82: $P_{x,y,z}^{its+1} += ...; \rightarrow\ c_{25}$

83: **end for**

84: **end for**

85: **end for**

86: *Cálculo de la fronteras BACK*

87: **for** $(x\ =\ 0;\ x\ <\ n;\ x++)$ **do**

88: **for** $(y\ =\ 0;\ y\ <\ L;\ y++)$ **do**

89: **for** $(z\ =\ 0;\ z\ <\ n;\ z++)$ **do**

90: $\frac{\partial p(x,y,z,t)}{\partial y} = ... ; \rightarrow\ c_{26}$

91: $\psi_y^{its} = ...; \rightarrow\ c_{27}$

92: **end for**

93: **end for**

94: **end for**

95: **for** $(x\ =\ 0;\ x\ <\ n;\ x++)$ **do**

96: **for** $(y\ =\ 0;\ y\ <\ L;\ y++)$ **do**

97: **for** $(z\ =\ 0;\ z\ <\ n;\ z++)$ **do**

98: $\frac{\partial \psi_y}{\partial y} = ...; \rightarrow \ c_{28}$

99: $\frac{\partial^2 p(x,y,z,t)}{\partial y^2} = ...; \rightarrow \ c_{29}$

100: $\zeta_y^{its} = ...; \rightarrow \ c_{30}$

101: $P_{x,y,z}^{its+1} += ...; \rightarrow \ c_{31}$

102: **end for**

103: **end for**

104: **end for**

105: *Cálculo de la fronteras FRONT*

106: **for** $(x \ = \ 0; \ x \ < \ n; \ x++)$ **do**

107: **for** $(y \ = \ n - L - 1; \ y \ < \ n; \ y++)$ **do**

108: **for** $(z \ = \ 0; \ z \ < \ n; \ z++)$ **do**

109: $\frac{\partial p(x,y,z,t)}{\partial y} = ... \ ; \rightarrow \ c_{32}$

110: $\psi_y^{its} = ...; \rightarrow \ c_{33}$

111: **end for**

112: **end for**

113: **end for**

114: **for** $(x \ = \ 0; \ x \ < \ n; \ x++)$ **do**

115: **for** $(y \ = \ n - L - 1; \ y \ < \ n; \ y++)$ **do**

116: **for** $(z \ = \ 0; \ z \ < \ n; \ z++)$ **do**

117: $\frac{\partial \psi_y}{\partial y} = ...; \rightarrow \ c_{34}$

118: $\frac{\partial^2 p(x,y,z,t)}{\partial y^2} = ...; \rightarrow \ c_{35}$

119: $\zeta_y^{its} = ...; \rightarrow \ c_{36}$

120: $P_{x,y,z}^{its+1} += ...; \rightarrow \ c_{37}$

121: **end for**

122: **end for**

123: **end for**

124: *Actualización de los Campos*

125: $P_{x,y,z}^{its-1} = P_{x,y,z}^{its}; \rightarrow \ c_{38}$

126: $P_{x,y,z}^{its} = P_{x,y,z}^{its+1}; \rightarrow \ c_{39}$

127: *Adición de la Fuente*

128: $P_{x,y,z}^{n} += f_{x_0,y_0,z_0}^{n}; \rightarrow \ c_{40}$

ALGORITMO QUE SOLUCIONA LA ECUACIÓN DE ONDA DE FORMA PARALELA

1: *Cálculo del Campo* $P_{x,y,z}^{its+1}$

2: **parallelism**$(n * n * n)$ $P_{x,y,z}^{its+1} = -P_{x,y,z}^{its-1} + 2P_{x,y,z}^{its} + ... \ ; \rightarrow \ c_1$

3: *Cálculo de las fronteras absorbentes para en campo* $P_{x,y,z}^{its+1}$

4: *Cálculo de la fronteras TOP*

5: **parallelism**$(n*n*L)$**:** $\frac{\partial p(x,y,z,t)}{\partial z} = ... \; ; \rightarrow \; c_2$

6: **parallelism**$(n*n*L)$**:** $\psi_z^{its} = ...; \rightarrow \; c_3$

7: **sync**;

8: **parallelism**$(n*n*L)$**:** $\frac{\partial \psi_z}{\partial z} = ...; \rightarrow \; c_4$

9: **parallelism**$(n*n*L)$**:** $\frac{\partial^2 p(x,y,z,t)}{\partial z^2} = ...; \rightarrow \; c_5$

10: **parallelism**$(n*n*L)$**:** $\zeta_z^{its} = ...; \rightarrow \; c_6$

11: **parallelism**$(n*n*L)$**:** $P_{x,y,z}^{its+1} += ...; \rightarrow \; c_7$

12: *Cálculo de la fronteras BOTTOM*

13: **parallelism**$(n*n*L)$**:** $\frac{\partial p(x,y,z,t)}{\partial z} = ... \; ; \rightarrow \; c_8$

14: **parallelism**$(n*n*L)$**:** $\psi_z^{its} = ...; \rightarrow \; c_9$

15: **sync**;

16: **parallelism**$(n*n*L)$**:** $\frac{\partial \psi_z}{\partial z} = ...; \rightarrow \; c_{10}$

17: **parallelism**$(n*n*L)$**:** $\frac{\partial^2 p(x,y,z,t)}{\partial z^2} = ...; \rightarrow \; c_{11}$

18: **parallelism**$(n*n*L)$**:** $\zeta_z^{its} = ...; \rightarrow \; c_{12}$

19: **parallelism**$(n*n*L)$**:** $P_{x,y,z}^{its+1} += ...; \rightarrow \; c_{13}$

20: *Cálculo de la fronteras LEFT*

21: **parallelism**$(n*n*L)$**:** $\frac{\partial p(x,y,z,t)}{\partial x} = ... \; ; \rightarrow \; c_{14}$

22: **parallelism**$(n*n*L)$**:** $\psi_x^{its} = ...; \rightarrow \; c_{15}$

23: **sync**;

24: **parallelism**$(n*n*L)$**:** $\frac{\partial \psi_x}{\partial x} = ...; \rightarrow \; c_{16}$

25: **parallelism**$(n*n*L)$**:** $\frac{\partial^2 p(x,y,z,t)}{\partial x^2} = ...; \rightarrow \; c_{17}$

26: **parallelism**$(n*n*L)$**:** $\zeta_x^{its} = ...; \rightarrow \; c_{18}$

27: **parallelism**$(n*n*L)$**:** $P_{x,y,z}^{its+1} += ...; \rightarrow \; c_{19}$

28: *Cálculo de la fronteras RIGHT*

29: **parallelism**$(n*n*L)$**:** $\frac{\partial p(x,y,z,t)}{\partial x} = ... \; ; \rightarrow \; c_{20}$

30: **parallelism**$(n*n*L)$**:** $\psi_x^{its} = ...; \rightarrow \; c_{21}$

31: **sync**;

32: **parallelism**$(n*n*L)$**:** $\frac{\partial \psi_x}{\partial x} = ...; \rightarrow \; c_{22}$

33: **parallelism**$(n*n*L)$**:** $\frac{\partial^2 p(x,y,z,t)}{\partial x^2} = ...; \rightarrow \; c_{23}$

34: **parallelism**$(n*n*L)$**:** $\zeta_x^{its} = ...; \rightarrow \; c_{24}$

35: **parallelism**$(n*n*L)$**:** $P_{x,y,z}^{its+1} += ...; \rightarrow \; c_{25}$

36: *Cálculo de la fronteras BACK*

37: **parallelism**$(n*n*L)$**:** $\frac{\partial p(x,y,z,t)}{\partial y} = ... \; ; \rightarrow \; c_{26}$

38: **parallelism**$(n*n*L)$**:** $\psi_y^{its} = ...; \rightarrow \; c_{27}$

39: **sync**;

40: **parallelism**$(n*n*L)$**:** $\frac{\partial \psi_y}{\partial y} = ...; \rightarrow\ c_{28}$

41: **parallelism**$(n*n*L)$**:** $\frac{\partial^2 p(x,y,z,t)}{\partial y^2} = ...; \rightarrow\ c_{29}$

42: **parallelism**$(n*n*L)$**:** $\zeta_y^{its} = ...; \rightarrow\ c_{30}$

43: **parallelism**$(n*n*L)$**:** $P_{x,y,z}^{its+1} += ...; \rightarrow\ c_{31}$

44: *Cálculo de la fronteras FRONT*

45: **parallelism**$(n*n*L)$**:** $\frac{\partial p(x,y,z,t)}{\partial y} = ... ; \rightarrow\ c_{32}$

46: **parallelism**$(n*n*L)$**:** $\psi_y^{its} = ...; \rightarrow\ c_{33}$

47: **sync**;

48: **parallelism**$(n*n*L)$**:** $\frac{\partial \psi_y}{\partial y} = ...; \rightarrow\ c_{34}$

49: **parallelism**$(n*n*L)$**:** $\frac{\partial^2 p(x,y,z,t)}{\partial y^2} = ...; \rightarrow\ c_{35}$

50: **parallelism**$(n*n*L)$**:** $\zeta_y^{its} = ...; \rightarrow\ c_{36}$

51: **parallelism**$(n*n*L)$**:** $P_{x,y,z}^{its+1} += ...; \rightarrow\ c_{37}$

52: *Actualización de los Campos*

53: $P_{x,y,z}^{its-1} = P_{x,y,z}^{its}; \rightarrow\ c_{38}$

54: $P_{x,y,z}^{its} = P_{x,y,z}^{its+1}; \rightarrow\ c_{39}$

55: *Adición de la Fuente*

56: $P_{x,y,z}^{its} += f_{s_x,s_y,s_z}^{its}; \rightarrow\ c_{40}$

CÓDIGO PARA SUMAR VECTORES EN CUDA

Cuadro de Código 5.25: Código para sumar vectores en CUDA

```
1     #include<stdlib.h>
2     #include<stdio.h>
3     #include<string.h>
4     #include<math.h>
5     #include<time.h>
6     #include<cuda_runtime.h>
7
8  __global__ void VecAdd(float* A, float* B, float* C, int N){
9     int i = blockDim.x * blockIdx.x + threadIdx.x;
10    if (i < N){
11      C[i] = A[i] + B[i];
12      }
13    }
14
15 main ( ){
```

```
  int N = 2048;
  int i;
  size_t size = N * sizeof(float);
// Allocate input vectors h_A and h_B in host memory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  float* h_C = (float*)malloc(size);
// Initialize input vectors
  for(i = 0; i<N ; i ++){
      h_A[i] = 1.0;
      h_B[i] = 2.0;
      }
// Allocate vectors in device memory
  float* d_A;
  cudaMalloc((void **)&d_A, size);
  float* d_B;
  cudaMalloc((void **)&d_B, size);
  float* d_C;
  cudaMalloc((void **)&d_C, size);
// Copy vectors from host memory to device memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
// Invoke kernel
  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
// Copy result from device memory to host memory
// h_C contains the result in host memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  for(i = 0; i<10 ; i++){
      printf(" h_C[%d]=%f ",i,h_C[i]);
      }
// Free device memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
// Free host memory ...
}
```

CÓDIGO PARA SUMAR VECTORES EN OPENCL

Cuadro de Código 5.26: Código para la suma de vectores en OpenCL

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MAX_SOURCE_SIZE (0x100000)
#define Work_grup_dim 32

void checkErr(cl_int error, char *name);
char * GetErrorString(cl_int error);
void checkMalloc(void * var,char *name);
void info_platforms_and_devices_open_cl(void);
 int main(int argc, const char *argv[]){
    // Create the two input vectors
    int i;
    const int N = 2048;
    float *A = (float*)malloc(sizeof(float)*N);
    check\cite{Algorithm}Malloc(A,"A");
    float *B = (float*)malloc(sizeof(float)*N);
    checkMalloc(B,"B");
      for(i = 0; i < N; i++) {
    A[i]= 1;
          B[i]= 2;
      }
    // Load the kernel source code into the array source_str
    FILE *fp;
    char *source_str;
    size_t source_size;
```

```
35      fp = fopen("vector_add_kernel.cl", "r");
36      if (!fp) {
37          fprintf(stderr, "Failed to load kernel.\n");
38          return 0;
39      }
40      source_str = (char*)malloc(MAX_SOURCE_SIZE);
41      source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
42      fclose( fp );
43
44    // Get platform and device information
45    cl_platform_id *platform_id;
46    cl_device_id* device_id;
47    cl_uint ret_num_devices;
48    cl_uint ret_num_platforms;
49    cl_int errNum;
50    cl_int id_plat=1;
51    cl_int id_device=0;
52    // First, query the total number of platforms
53  info_platforms_and_devices_open_cl();
54  errNum = clGetPlatformIDs(0, NULL, &ret_num_platforms);
55    checkErr(errNum,"clGetPlatformIDs --> Number of platforms ");
56  // Next, allocate memory for the installed platforms, and query
57  platform_id = (cl_platform_id *)malloc(sizeof(cl_platform_id)*↩
        ret_num_platforms);
58    checkMalloc(platform_id,"platform_id");
59  // to get the list.
60  errNum = clGetPlatformIDs(ret_num_platforms, platform_id, NULL);
61    checkErr(errNum,"clGetPlatformIDs --> platforms ID ");
62    // Second, query the total number of device in platform chose
63  errNum = clGetDeviceIDs( platform_id[id_plat],CL_DEVICE_TYPE_GPU, 0,NULL, &↩
        ret_num_devices);
64    checkErr(errNum,"clGetDeviceIDs --> Number of device ");
65  // Next, allocate memory for the device platforms, and query
66  device_id = (cl_device_id*)malloc(sizeof(cl_device_id)*ret_num_devices);
67    checkMalloc(device_id,"device_id");
68  errNum = clGetDeviceIDs(platform_id[id_plat],CL_DEVICE_TYPE_GPU,↩
        ret_num_devices ,device_id, &ret_num_devices);
69    checkErr(errNum,"clGetDeviceIDs --> device ID ");
70        // get platform info for each platform
71    // Create an OpenCL context
```

```
72  cl_context_properties properties[3]={CL_CONTEXT_PLATFORM,(↩
        cl_context_properties)(platform_id[id_plat]),0};
73  cl_device_id dev = device_id[id_device];
74  cl_context context = clCreateContext(NULL,ret_num_devices,&dev,NULL, NULL, &↩
        errNum);
75        checkErr(errNum,"clCreateCotext --> context ");
76    // Create a command queue
77      cl_command_queue command_queue = clCreateCommandQueue(context, device_id[↩
          id_device], 0, &errNum);
78    checkErr(errNum,"clCreateCommadQueue --> Queue ");
79    // Create memory buffers on the device for each vector
80      cl_mem A_d = clCreateBuffer(context, CL_MEM_READ_WRITE,N*sizeof(float), ↩
          NULL, &errNum);
81    checkErr(errNum,"clCreateBuffer --> A_d ");
82      cl_mem B_d = clCreateBuffer(context, CL_MEM_READ_WRITE,N*sizeof(float), ↩
          NULL, &errNum);
83    checkErr(errNum,"clCreateBuffer --> B_d ");
84      cl_mem C_d = clCreateBuffer(context, CL_MEM_READ_WRITE,N*sizeof(float), ↩
          NULL, &errNum);
85    checkErr(errNum,"clCreateBuffer --> C_d ");
86    // Copy the lists A and B to their respective memory buffers
87      errNum = clEnqueueWriteBuffer(command_queue, A_d, CL_TRUE, 0,N*sizeof(↩
          float), A, 0, NULL, NULL);
88    checkErr(errNum,"clEnqueueWriteBuffer --> A to A_d ");
89      errNum = clEnqueueWriteBuffer(command_queue, B_d, CL_TRUE, 0,N*sizeof(↩
          float), B, 0, NULL, NULL);
90    checkErr(errNum,"clEnqueueWriteBuffer --> B to B_d ");
91    // Create a program from the kernel source
92    cl_program program = clCreateProgramWithSource(context,1,(const char **)&↩
        source_str, (const size_t *)&source_size, &errNum);
93    checkErr(errNum,"clCreateProgram --> program ");
94    // Build the program
95      errNum = clBuildProgram(program,1,&device_id[id_device], NULL, NULL, NULL)↩
          ;
96    if ( errNum != CL_SUCCESS) {
97      printf("error build program\n");
98      // Determine the reason for the error
99      char buildLog[16384];
100     clGetProgramBuildInfo(program, device_id[id_device], CL_PROGRAM_BUILD_LOG,↩
          sizeof(buildLog), buildLog, NULL);
```

```
101       printf("ERROR:\t\t%s \n",buildLog);
102       clReleaseProgram(program);
103     }
104     // Create the OpenCL kernel
105   cl_kernel kernel_add_vector = clCreateKernel(program, "vector_add", &errNum);
106     checkErr(errNum,"clCreateKernel --> kerne_add_vector ");
107     // Set the arguments of the kernel
108       errNum = clSetKernelArg(kernel_add_vector, 0, sizeof(cl_mem), (void *)&A_d↩
            );
109     checkErr(errNum,"clSetKernelArg --> A_d  ");
110       errNum = clSetKernelArg(kernel_add_vector, 1, sizeof(cl_mem), (void *)&B_d↩
            );
111     checkErr(errNum,"clSetKernelArg --> B_d ");
112       errNum = clSetKernelArg(kernel_add_vector, 2, sizeof(cl_mem), (void *)&C_d↩
            );
113     checkErr(errNum,"clSetKernelArg --> C_d ");
114     // Execute the OpenCL kernel on the list
115     size_t global_item_size = N; // Process the entire lists
116     size_t local_item_size = Work_grup_dim; // Divide work items into groups of ↩
          64
117       errNum = clEnqueueNDRangeKernel(command_queue, kernel_add_vector,1,NULL,&↩
            global_item_size, &local_item_size, 0, NULL, NULL);
118     checkErr(errNum,"cl_run_kernel");
119     // Read the memory buffer C on the device to the local variable C
120     float *C = (float*)malloc(sizeof(float)*N);
121     checkMalloc(C,"C");
122       errNum = clEnqueueReadBuffer(command_queue, C_d, CL_TRUE, 0,N*sizeof(float↩
            ), C, 0, NULL, NULL);
123     checkErr(errNum,"clEnqueueReadBuffer --> C to C_d ");
124     // Display the result to the screen
125       for(i = 0; i < N; i++){
126     printf("%f + %f = %f\n", A[i], B[i], C[i]);
127   }
128     // Clean up
129     errNum = clFlush(command_queue);
130     errNum = clFinish(command_queue);
131     errNum = clReleaseKernel(kernel_add_vector);
132     errNum = clReleaseProgram(program);
133     errNum = clReleaseMemObject(A_d);
134     errNum = clReleaseMemObject(B_d);
```

```
135     errNum = clReleaseMemObject(C_d);
136     errNum = clReleaseCommandQueue(command_queue);
137     errNum = clReleaseContext(context);
138     free(A);
139     free(B);
140     free(C);
141     return 0;
142 }
143 
144 void checkErr(cl_int error, char *name){
145 
146   if(error != CL_SUCCESS){
147     char *temp;
148     temp=(char*)GetErrorString(error);
149     printf("state = %s | type = %s\n",temp,name);
150   }else{
151     printf(" good = %s\n",name);
152   }
153 }
154 
155 char * GetErrorString(cl_int error){
156 
157     switch (error){
158 
159     case CL_SUCCESS:
160         return "CL_SUCCESS";
161     case CL_DEVICE_NOT_FOUND:
162         return "CL_DEVICE_NOT_FOUND";
163     case CL_DEVICE_NOT_AVAILABLE:
164         return "CL_DEVICE_NOT_AVAILABLE";
165     case CL_COMPILER_NOT_AVAILABLE:
166         return "CL_COMPILER_NOT_AVAILABLE";
167     case CL_MEM_OBJECT_ALLOCATION_FAILURE:
168         return "CL_MEM_OBJECT_ALLOCATION_FAILURE";
169     case CL_OUT_OF_RESOURCES:
170         return "CL_OUT_OF_RESOURCES";
171     case CL_OUT_OF_HOST_MEMORY:
172         return "CL_OUT_OF_HOST_MEMORY";
173     case CL_PROFILING_INFO_NOT_AVAILABLE:
174         return "CL_PROFILING_INFO_NOT_AVAILABLE";
```

```
175     case CL_MEM_COPY_OVERLAP:
176         return "CL_MEM_COPY_OVERLAP";
177     case CL_IMAGE_FORMAT_MISMATCH:
178         return "CL_IMAGE_FORMAT_MISMATCH";
179     case CL_IMAGE_FORMAT_NOT_SUPPORTED:
180         return "CL_IMAGE_FORMAT_NOT_SUPPORTED";
181     case CL_BUILD_PROGRAM_FAILURE:
182         return "CL_BUILD_PROGRAM_FAILURE";
183     case CL_MAP_FAILURE:
184         return "CL_MAP_FAILURE";
185     case CL_INVALID_VALUE:
186         return "CL_INVALID_VALUE";
187     case CL_INVALID_DEVICE_TYPE:
188         return "CL_INVALID_DEVICE_TYPE";
189     case CL_INVALID_PLATFORM:
190         return "CL_INVALID_PLATFORM";
191     case CL_INVALID_DEVICE:
192         return "CL_INVALID_DEVICE";
193     case CL_INVALID_CONTEXT:
194         return "CL_INVALID_CONTEXT";
195     case CL_INVALID_QUEUE_PROPERTIES:
196         return "CL_INVALID_QUEUE_PROPERTIES";
197     case CL_INVALID_COMMAND_QUEUE:
198         return "CL_INVALID_COMMAND_QUEUE";
199     case CL_INVALID_HOST_PTR:
200         return "CL_INVALID_HOST_PTR";
201     case CL_INVALID_MEM_OBJECT:
202         return "CL_INVALID_MEM_OBJECT";
203     case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
204         return "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR";
205     case CL_INVALID_IMAGE_SIZE:
206         return "CL_INVALID_IMAGE_SIZE";
207     case CL_INVALID_SAMPLER:
208         return "CL_INVALID_SAMPLER";
209     case CL_INVALID_BINARY:
210         return "CL_INVALID_BINARY";
211     case CL_INVALID_BUILD_OPTIONS:
212         return "CL_INVALID_BUILD_OPTIONS";
213     case CL_INVALID_PROGRAM:
214         return "CL_INVALID_PROGRAM";
```

```
215     case CL_INVALID_PROGRAM_EXECUTABLE:
216         return "CL_INVALID_PROGRAM_EXECUTABLE";
217     case CL_INVALID_KERNEL_NAME:
218         return "CL_INVALID_KERNEL_NAME";
219     case CL_INVALID_KERNEL_DEFINITION:
220         return "CL_INVALID_KERNEL_DEFINITION";
221     case CL_INVALID_KERNEL:
222         return "CL_INVALID_KERNEL";
223     case CL_INVALID_ARG_INDEX:
224         return "CL_INVALID_ARG_INDEX";
225     case CL_INVALID_ARG_VALUE:
226         return "CL_INVALID_ARG_VALUE";
227     case CL_INVALID_ARG_SIZE:
228         return "CL_INVALID_ARG_SIZE";
229     case CL_INVALID_KERNEL_ARGS:
230         return "CL_INVALID_KERNEL_ARGS";
231     case CL_INVALID_WORK_DIMENSION:
232         return "CL_INVALID_WORK_DIMENSION";
233     case CL_INVALID_WORK_GROUP_SIZE:
234         return "CL_INVALID_WORK_GROUP_SIZE";
235     case CL_INVALID_WORK_ITEM_SIZE:
236         return "CL_INVALID_WORK_ITEM_SIZE";
237     case CL_INVALID_EVENT_WAIT_LIST:
238         return "CL_INVALID_EVENT_WAIT_LIST";
239     case CL_INVALID_EVENT:
240         return "CL_INVALID_EVENT";
241     case CL_INVALID_OPERATION:
242         return "CL_INVALID_OPERATION";
243     case CL_INVALID_GL_OBJECT:
244         return "CL_INVALID_GL_OBJECT";
245     case CL_INVALID_BUFFER_SIZE:
246         return "CL_INVALID_BUFFER_SIZE";
247     case CL_INVALID_MIP_LEVEL:
248         return "CL_INVALID_MIP_LEVEL";
249     case CL_INVALID_GLOBAL_WORK_SIZE:
250         return "CL_INVALID_GLOBAL_WORK_SIZE";
251     //case CL_PLATFORM_NOT_FOUND_KHR:
252     //    return "CL_PLATFORM_NOT_FOUND_KHR";
253   // unknown
254     default:
```

```
255         return "unknown error code";
256     }
257
258 }
259
260 void checkMalloc(void * var,char *name){
261   if( var == NULL){
262           printf("It's not possible to allocate  %s. \n",name);}
263       else {
264           printf(" Memory has already been allocated %s.\n",name);}
265 }
266
267 void info_platforms_and_devices_open_cl(void)
268 {
269     cl_platform_id *platform_id;
270     cl_uint ret_num_devices;
271     cl_uint ret_num_platforms;
272     cl_int errNum;
273     cl_uint id_p;
274     cl_uint id_d;
275     char cBuffer[16000];
276     char cBuffer_d[16000];
277     // First, query the total number of platforms
278   errNum = clGetPlatformIDs(0, NULL, &ret_num_platforms);
279     checkErr(errNum,"clGetPlatformIDs --> Number of platforms ");
280   printf("OpenCL Platforms found : %d \n", ret_num_platforms);
281   // Next, allocate memory for the installed platforms, and query
282   platform_id = (cl_platform_id *)malloc(sizeof(cl_platform_id)*↩
        ret_num_platforms);
283   // to get the list.
284   errNum = clGetPlatformIDs(ret_num_platforms, platform_id, NULL);
285     checkErr(errNum,"clGetPlatformIDs --> platforms ID ");
286   for(id_p = 0; id_p < ret_num_platforms; ++id_p) {
287     errNum = clGetPlatformInfo (platform_id[id_p], CL_PLATFORM_NAME, sizeof(char↩
          )*16000, &cBuffer, NULL);
288     if(errNum == CL_SUCCESS) {
289           printf("CL_PLATFORM_NAME: \t %s | id = %d \n", cBuffer,id_p);
290       errNum = clGetDeviceIDs( platform_id[id_p],CL_DEVICE_TYPE_ALL, 0,NULL, &↩
            ret_num_devices);
291       checkErr(errNum,"clGetDeviceIDs --> Number of device ");
```

```
292       if(ret_num_devices>1){
293         cl_device_id * device_id_2 = (cl_device_id*)malloc(sizeof(cl_device_id)*↩
              ret_num_devices);
294         errNum = clGetDeviceIDs(platform_id[id_p],CL_DEVICE_TYPE_GPU,↩
              ret_num_devices ,device_id_2, &ret_num_devices);
295         checkErr(errNum,"clGetDeviceIDs --> device ID ");
296               for(id_d = 0 ;id_d < ret_num_devices; id_d++){
297           errNum = clGetDeviceInfo(device_id_2[id_d],CL_DEVICE_NAME,sizeof(char)↩
                *16000,&cBuffer_d,NULL);
298           if(errNum == CL_SUCCESS) {
299             printf("CL_DEVICE_NAME: \t\t %s | id = %d \n",cBuffer_d,id_d);
300           } else{
301             checkErr(errNum,"clGetDeviceInfo --> Call ");
302           }
303         }
304             } else{
305         id_d=0;
306         cl_device_id device_id_1;
307         errNum=clGetDeviceIDs(platform_id[id_p],CL_DEVICE_TYPE_ALL,↩
              ret_num_devices,&device_id_1, &ret_num_devices);
308         checkErr(errNum,"clGetDeviceIDs --> device_1 ID ");
309         errNum = clGetDeviceInfo(device_id_1,CL_DEVICE_NAME,sizeof(char)*16000,&↩
              cBuffer_d,NULL);
310         if(errNum == CL_SUCCESS) {
311           printf("CL_DEVICE_NAME: \t\t %s | id = %d \n",cBuffer_d,id_d);
312         } else{
313           checkErr(errNum,"clGetDeviceInfo --> Call ");
314         }
315       }
316           } else {
317       checkErr(errNum,"clGetPlatformInfo --> Call ");
318       }
319   }
320 }
```

CÓDIGO PARA VISUALIZAR TODAS LAS PLATAFORMAS Y LOS *DEVICES* QUE SOPORTAN OPENCL

```
void info_platforms_and_devices_open_cl(void)
{
    cl_platform_id *platform_id;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_int errNum;
    cl_uint id_p;
    cl_uint id_d;
    char cBuffer[16000];
    char cBuffer_d[16000];
    // First, query the total number of platforms
        errNum = clGetPlatformIDs(0, NULL, &ret_num_platforms);
                checkErr(errNum,"clGetPlatformIDs --> Number of platforms ");
        printf("OpenCL Platforms found : %d \n", ret_num_platforms);
        // Next, allocate memory for the installed platforms, and query
        platform_id = (cl_platform_id *)malloc(sizeof(cl_platform_id)*ret_num_platforms);
        // to get the list.
        errNum = clGetPlatformIDs(ret_num_platforms, platform_id, NULL);
                checkErr(errNum,"clGetPlatformIDs --> platforms ID ");
        for(id_p = 0; id_p < ret_num_platforms; ++id_p) {
                errNum = clGetPlatformInfo (platform_id[id_p], CL_PLATFORM_NAME,sizeof(char)*16000, &cBuffer, NULL);
                if(errNum == CL_SUCCESS) {
                        printf("CL_PLATFORM_NAME: \t%s | id = %d \n", cBuffer,id_p);
                        errNum = clGetDeviceIDs( platform_id[id_p],CL_DEVICE_TYPE_ALL, 0,NULL, &ret_num_devices);
                        checkErr(errNum,"clGetDeviceIDs --> Number of device ");
                        cl_device_id * device_id_2 = (cl_device_id*)malloc(sizeof(cl_device_id)*ret_num_devices);
                        if(ret_num_devices>1){
                                errNum = clGetDeviceIDs(platform_id[id_p],CL_DEVICE_TYPE_GPU,ret_num_devices ,device_id_2, &ret_num_devices);
                                checkErr(errNum,"clGetDeviceIDs --> device ID ");
                                for(id_d = 0 ;id_d < ret_num_devices; id_d++){
                                        errNum = clGetDeviceInfo(device_id_2[id_d],CL_DEVICE_NAME,sizeof(char)
```

```
                                        *16000,&cBuffer_d,NULL);
32                                      if(errNum == CL_SUCCESS) {
33                                              printf("CL_DEVICE_NAME: \t\t%s |↩
                                                    id = %d \n",cBuffer_d,id_d)↩
                                                ;
34                                      } else{
35                                              checkErr(errNum,"clGetDeviceInfo↩
                                                    --> Call ");
36                                      }
37                              }
38                      } else{
39                              id_d=0;
40                              errNum =clGetDeviceIDs(platform_id[id_p],↩
                                    CL_DEVICE_TYPE_ALL,ret_num_devices ,↩
                                    device_id_2, &ret_num_devices);
41                              checkErr(errNum,"clGetDeviceIDs --> device_1 ID ↩
                                    ");
42                              errNum = clGetDeviceInfo(device_id_2[id_d],↩
                                    CL_DEVICE_NAME,sizeof(char)*16000,&cBuffer_d↩
                                    ,NULL);
43                              if(errNum == CL_SUCCESS) {
44                                      printf("CL_DEVICE_NAME: \t\t%s | id = %d↩
                                            \n",cBuffer_d,id_d);
45                              } else{
46                                      checkErr(errNum,"clGetDeviceInfo --> ↩
                                            Call ");
47                              }
48                      }
49              } else {
50                      checkErr(errNum,"clGetPlatformInfo --> Call ");
51              }
52      }
53  }
```

CÓDIGO PARA VISUALIZAR LOS ERRORES QUE PUEDE RETORNAR LOS COMANDOS DE OPENCL

```
1  void checkErr(cl_int error, char *name){
```

```
2
3   if(error != CL_SUCCESS){
4     char *temp;
5     temp=(char*)GetErrorString(error);
6     printf("state = %s | type = %s\n",temp,name);
7   }else{
8     printf(" good = %s\n",name);
9   }
10 }
11
12 char * GetErrorString(cl_int error){
13
14     switch (error){
15
16     case CL_SUCCESS:
17         return "CL_SUCCESS";
18     case CL_DEVICE_NOT_FOUND:
19         return "CL_DEVICE_NOT_FOUND";
20     case CL_DEVICE_NOT_AVAILABLE:
21         return "CL_DEVICE_NOT_AVAILABLE";
22     case CL_COMPILER_NOT_AVAILABLE:
23         return "CL_COMPILER_NOT_AVAILABLE";
24     case CL_MEM_OBJECT_ALLOCATION_FAILURE:
25         return "CL_MEM_OBJECT_ALLOCATION_FAILURE";
26     case CL_OUT_OF_RESOURCES:
27         return "CL_OUT_OF_RESOURCES";
28     case CL_OUT_OF_HOST_MEMORY:
29         return "CL_OUT_OF_HOST_MEMORY";
30     case CL_PROFILING_INFO_NOT_AVAILABLE:
31         return "CL_PROFILING_INFO_NOT_AVAILABLE";
32     case CL_MEM_COPY_OVERLAP:
33         return "CL_MEM_COPY_OVERLAP";
34     case CL_IMAGE_FORMAT_MISMATCH:
35         return "CL_IMAGE_FORMAT_MISMATCH";
36     case CL_IMAGE_FORMAT_NOT_SUPPORTED:
37         return "CL_IMAGE_FORMAT_NOT_SUPPORTED";
38     case CL_BUILD_PROGRAM_FAILURE:
39         return "CL_BUILD_PROGRAM_FAILURE";
40     case CL_MAP_FAILURE:
41         return "CL_MAP_FAILURE";
```

```
42      case CL_INVALID_VALUE:
43          return "CL_INVALID_VALUE";
44      case CL_INVALID_DEVICE_TYPE:
45          return "CL_INVALID_DEVICE_TYPE";
46      case CL_INVALID_PLATFORM:
47          return "CL_INVALID_PLATFORM";
48      case CL_INVALID_DEVICE:
49          return "CL_INVALID_DEVICE";
50      case CL_INVALID_CONTEXT:
51          return "CL_INVALID_CONTEXT";
52      case CL_INVALID_QUEUE_PROPERTIES:
53          return "CL_INVALID_QUEUE_PROPERTIES";
54      case CL_INVALID_COMMAND_QUEUE:
55          return "CL_INVALID_COMMAND_QUEUE";
56      case CL_INVALID_HOST_PTR:
57          return "CL_INVALID_HOST_PTR";
58      case CL_INVALID_MEM_OBJECT:
59          return "CL_INVALID_MEM_OBJECT";
60      case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
61          return "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR";
62      case CL_INVALID_IMAGE_SIZE:
63          return "CL_INVALID_IMAGE_SIZE";
64      case CL_INVALID_SAMPLER:
65          return "CL_INVALID_SAMPLER";
66      case CL_INVALID_BINARY:
67          return "CL_INVALID_BINARY";
68      case CL_INVALID_BUILD_OPTIONS:
69          return "CL_INVALID_BUILD_OPTIONS";
70      case CL_INVALID_PROGRAM:
71          return "CL_INVALID_PROGRAM";
72      case CL_INVALID_PROGRAM_EXECUTABLE:
73          return "CL_INVALID_PROGRAM_EXECUTABLE";
74      case CL_INVALID_KERNEL_NAME:
75          return "CL_INVALID_KERNEL_NAME";
76      case CL_INVALID_KERNEL_DEFINITION:
77          return "CL_INVALID_KERNEL_DEFINITION";
78      case CL_INVALID_KERNEL:
79          return "CL_INVALID_KERNEL";
80      case CL_INVALID_ARG_INDEX:
81          return "CL_INVALID_ARG_INDEX";
```

```
82      case CL_INVALID_ARG_VALUE:ape_checkmalloc
83          return "CL_INVALID_ARG_VALUE";
84      case CL_INVALID_ARG_SIZE:
85          return "CL_INVALID_ARG_SIZE";
86      case CL_INVALID_KERNEL_ARGS:
87          return "CL_INVALID_KERNEL_ARGS";
88      case CL_INVALID_WORK_DIMENSION:
89          return "CL_INVALID_WORK_DIMENSION";
90      case CL_INVALID_WORK_GROUP_SIZE:
91          return "CL_INVALID_WORK_GROUP_SIZE";
92      case CL_INVALID_WORK_ITEM_SIZE:
93          return "CL_INVALID_WORK_ITEM_SIZE";
94      case CL_INVALID_EVENT_WAIT_LIST:
95          return "CL_INVALID_EVENT_WAIT_LIST";
96      case CL_INVALID_EVENT:
97          return "CL_INVALID_EVENT";
98      case CL_INVALID_OPERATION:
99          return "CL_INVALID_OPERATION";
100     case CL_INVALID_GL_OBJECT:
101         return "CL_INVALID_GL_OBJECT";
102     case CL_INVALID_BUFFER_SIZE:
103         return "CL_INVALID_BUFFER_SIZE";
104     case CL_INVALID_MIP_LEVEL:
105         return "CL_INVALID_MIP_LEVEL";
106     case CL_INVALID_GLOBAL_WORK_SIZE:
107         return "CL_INVALID_GLOBAL_WORK_SIZE";
108     //case CL_PLATFORM_NOT_FOUND_KHR:
109     //    return "CL_PLATFORM_NOT_FOUND_KHR";
110   // unknown
111     default:
112         return "unknown error code";
113     }
114
115 }
```

CÓDIGO PARA VERIFICAR LAS RESERVA DE MEMORIA

```
1 void checkMalloc(void * var,char *name){
```

```
2   if( var == NULL){
3           printf("It's not possible to allocate  %s. \n",name);}
4       else {
5           printf(" Memory has already been allocated %s.\n",name);}
6   }
```

ANEXO D:
COMPILACIÓN

COMPILACIÓN EN CUDA

Para el desarrollo de nuestras aplicaciones en CUDA NVIDIA nos presenta su propio compilador "`nvcc`" el cual es basado en "`LLMV`" un compilador que es software abierto. Este puede compilar códigos escritos en C++ y C [11].

En el cuadro de código 5.27 se muestra como compilar nuestro ejemplo *VecAdd* en *linux* desde la terminal de comandos.

Cuadro de Código 5.27: Compilación de Códigos CUDA

```
1 nvcc vector_add.cu -o vecadd
```

Cabe resaltar que los códigos escritos en CUDA se deben guardar en un archivo con extensión "`.cu`". Además es necesario pre-instalar el SDK de CUDA que soporten nuestro *hardware*.

COMPILACIÓN EN OPENCL

Para el desarrollo de nuestras aplicaciones en OpenCL Kronos nos permite compilar con la bandera "`-lOpenCL`" la cual se puede usar en compiladores de C, C++, python, etc. Para mas información consultar [6].

En el cuadro de código 5.28 se muestra como compilar nuestro ejemplo *VecAdd* en *linux* desde la terminal de comandos.

Cuadro de Código 5.28: Compilación de Códigos OpenCL

```
gcc -lm -lOpenCL -Wall add_vector.c -o vector_add
```

Cabe resaltar que es necesario pre-instalar el SDK de OpenCL que soporten nuestro *hardware*.

ANEXO E:
KERNELS

KERNEL QUE SOLUCIONA ECUACIÓN SIN EL APORTE DE LAS FRONTERAS ABSORBENTES EN CUDA

Cuadro de Código 5.29: *Kernel* que soluciona la ecuación sin el aporte de las fronteras en CUDA

```
__global__ void stencil_eval_gpu(int Nx, int Ny, int Nz,/* Model size  */\
                float* P_1,             /* Past field         */\
                float* P_2,             /* Present field      */\
                float* c_in,            /* Velocity field     */\
                float  dt,              /* Time step          */\
                float  dx,              /* Model resolution   */\
                float  dy,              /*      (dx,dy,dz)    */\
                float  dz,              /*                    */\
                float* coef,            /* 2 deriv. FD coef   */\){
    /* Declaring variables */
    int idx  =  threadIdx.x + blockIdx.x*blockDim.x;
    int idy  =  threadIdx.y + blockIdx.y*blockDim.y;
    int idz  =  threadIdx.z + blockIdx.z*blockDim.z;

    /* Constraining threads */
    if(idx < Nx &&  idy < Ny &&  idz < Nz){
      int iOrd;
            int Limxp = Nx - idx - 1, Limxn = idx;
            int Limyp = Ny - idy - 1, Limyn = idy;
            int Limzp = Nz - idz - 1, Limzn = idz;
            float lap;
```

```
22              float d2x, d2y, d2z;
23              /* Computing 2 derivates (center of the stencil) */
24              d2x = 0;  d2y = 0;  d2z = 0;
25          /* Adding the points of the p-branch of X in the stencil */
26    for(iOrd = 0; iOrd<=MIN(4,Limxp); iOrd++){
27              d2x += coef[4-iOrd]*Id(P_2,idx+iOrd,idy,idz);}
28          /* Adding the points of the n-branch of X in the stencil */
29          for(iOrd = 1; iOrd<=MIN(4,Limxn); iOrd++){
30              d2x += coef[4-iOrd]*Id(P_2,idx-iOrd,idy,idz);}
31          /* Adding the points of the p-branch of Y in the stencil */
32      for(iOrd = 0; iOrd<=MIN(4,Limyp); iOrd++){
33              d2y += coef[4-iOrd]*Id(P_2,idx,idy+iOrd,idz);}
34          /* Adding the points of the n-branch of Y in the stencil */
35    for(iOrd = 1; iOrd<=MIN(4,Limyn); iOrd++){
36              d2y += coef[4-iOrd]*Id(P_2,idx,idy-iOrd,idz);}
37              /* Adding the points of the p-branch of Z in the stencil */
38      for(iOrd = 0; iOrd<=MIN(4,Limzp); iOrd++){
39              d2z += coef[4-iOrd]*Id(P_2,idx,idy,idz+iOrd);}
40          /* Adding the points of the n-branch of Z in the stencil */
41          for(iOrd = 1; iOrd<=MIN(4,Limzn); iOrd++){
42              d2z += coef[4-iOrd]*Id(P_2,idx,idy,idz-iOrd);}
43              /* Computing the laplacian */
44            lap = d2x/dx/dx + d2y/dy/dy + d2z/dz/dz;
45                /* Wave equation */
46              Id(P_1,idx,idy,idz)=-Id(P_1,idx,idy,idz)+2*Id(P_2,idx,idy,idz)+dt*dt*↩
                    Id(c_in,idx,idy,idz)*Id(c_in,idx,idy,idz)*lap;
47            }
48  }
```

KERNEL QUE ADICIONA EL APORTE DE LA FRONTERA *LEFT* EN CUDA

Cuadro de Código 5.30: *Kernel* que adiciona el aporte de la frontera *left* en CUDA

```
1   __global__ void apply_cpml_left_gpu(int Nx, int Ny, int Nz,\
2               float* P_2,           /* Field  P in t       */\
3               float* P_1,           /* Field  P in t-d     */\
```

```
            float* c_in,          /* Velocity field      */\
            float  dt,            /* Time step           */\
            float  dx,            /* Spacial resolution  */\
            float  dy,            /*     (dx, dy, dz     */\
            float  dz,            /*                     */\
            float* coef,          /* 2 deriv. FD coeff   */\
            int    L,             /* CPML  layers        */\
            float*   a_x,         /* Auxiliar variable a */\
            float*   a_x_h,       /* Auxiliar variable a */\
            float*   b_x,         /* Auxiliar variable b */\
            float*   b_x_h,       /* Auxiliar variable b */\
            float* zeta_left,     /* Recursive var. zeta */\
            float* psi_left ,     /* Recursive var. psi  */\
            float* coefd1PML      /* 1 deriv. FD coeff   */){
    /* Declaring variables*/
    int idx  =  threadIdx.x + blockIdx.x*blockDim.x;
    int idy  =  threadIdx.y + blockIdx.y*blockDim.y;
    int idz  =  threadIdx.z + blockIdx.z*blockDim.z;

    int Limxn, Limxp; /* Limits of the branches of the stencil in x, y, z */
    int iOrd;  /* Index for the each of the branch */
    float lap;  /* Laplacian */
    float dpsi;  /* Derivate of the field psi */
    float d1x;  /* First order derivates */
    float d2x;  /* Second  order derivates */
    /* Constraining threads */
    if(idx < L && idy < Ny &&  idz < Nz){
        /*** Psi field ***/
        /* Updating the psi field */
        Limxn = idx;   Limxp = Nx-idx-1;
        /* Computing Initializing derivates */
        d1x = 0;
        /* Adding the points of the  p-branch of x stencil */
        for(iOrd = 0; iOrd<=MIN(3,Limxp-1); iOrd++)
            d1x += coefd1PML[iOrd]*Id(P_2,idx+iOrd+1,idy,idz);
        /* Adding the points of the  n-branch of x stencil */
        for(iOrd = 0; iOrd<=MIN(3,Limxn); iOrd++)
            d1x -= coefd1PML[iOrd]*Id(P_2,idx-iOrd,idy,idz);
        /* Computing the laplacian */
        lap = d1x/dx;
```

```
44          /* Field psi left*/
45          Idlx(psi_left,idx,idy,idz) = b_x_h[L-idx-1]*lap + a_x_h[L-idx-1]*Idlx(↩
                psi_left,idx,idy,idz);
46          /* Synchronizing to computed dPsi */
47          __syncthreads();
48          /*** dPsi field ***/
49          /* Updating the psi field */
50          Limxn = idx-1;   Limxp = L-idx-1+1;
51          /* Computing Initializing derivates */
52          d1x = 0;
53          /* Adding the points of the  p-branch of x stencil */
54          for(iOrd = 0; iOrd<=MIN(3,Limxp-1); iOrd++)
55              d1x += coefd1PML[iOrd]*Idlx(psi_left,idx-1+iOrd+1,idy,idz);
56          /* Adding the points of the  n-branch of x stencil */
57          for(iOrd = 0; iOrd<=MIN(3,Limxn); iOrd++)
58              d1x -= coefd1PML[iOrd]*Idlx(psi_left,idx-1-iOrd,idy,idz);
59          /* Computing the derivate */
60          dpsi = d1x/dx;
61          /*** Zeta field ***/
62          /* Updating the zeta field */
63          Limxn = idx;   Limxp = Nx-idx-1;
64          /* Computing Initializing derivates */
65          d2x = 0;
66          /* Adding the points of the  p-branch of x stencil */
67          for(iOrd = 0; iOrd<=MIN(4,Limxp); iOrd++)
68              d2x += coef[4-iOrd]*Id(P_2,idx+iOrd,idy,idz);
69          /* Adding the points of the  n-branch of x stencil */
70          for(iOrd = 1; iOrd<=MIN(4,Limxn); iOrd++)
71              d2x += coef[4-iOrd]*Id(P_2,idx-iOrd,idy,idz);
72          /* Computing the laplacian */
73          lap = d2x/dx/dx;
74          /* Field Zeta left*/
75          Idlx(zeta_left,idx,idy,idz) = b_x[L-idx-1]*(lap+dpsi)+a_x[L-idx-1]*Idlx(↩
                zeta_left,idx,idy,idz);
76        /* Wave equation */
77        Id(P_1,idx,idy,idz)+=dt*dt*Id(c_in,idx,idy,idz)*Id(c_in,idx,idy,idz)*(↩
              Idlx(zeta_left,idx,idy,idz)+dpsi);
78      }
79  }
```

KERNEL QUE SOLUCIONA ECUACIÓN SIN EL APORTE DE LAS FRONTERAS ABSORBENTES EN OPENCL

Cuadro de Código 5.31: *Kernel* que soluciona la ecuación sin el aporte de las fronteras en OpenCL

```
__kernel void stencil_eval_gpu(int Nx, int Ny, int Nz,/* Model size*/\
                __global float* P_1,     /* Past field         */\
                __global float* P_2,     /* Present field      */\
                __global float* c_in,    /* Velocity field     */\
                float dt,      /* Time step          */\
                float dx,      /* Model resolution   */\
                float dy,    /*      (dx,dy,dz)      */\
                float dz,       /*                     */\
                __global float* coef,    /* 2 deriv. FD coef   */\){
    /* Declaring variables */
    int idx  =  get_global_id(0);
    int idy  =  get_global_id(1);
    int idz  =  get_global_id(2);
    /* Constraining threads */
    if(idx < Nx &&  idy < Ny &&  idz < Nz){
      uint iOrd;
      int Limxp = Nx - idx - 1;
      int Limxn = idx;
      int Limyp = Ny - idy - 1;
      int Limyn = idy;
      int Limzp = Nz - idz - 1;
      int Limzn = idz;
      float lap;
      float d2x;
      float d2y;
      float d2z;
            /* Computing 2 derivates (center of the stencil) */
      d2x = 0;
      d2y = 0;
      d2z = 0;
            /* Adding the points of the p-branch of X in the stencil */
            for(iOrd = 0; iOrd<=MIN(4,Limxp); iOrd++)
              d2x += coef[4-iOrd]*Id(P_2,idx+iOrd,idy,idz);
```

```
34              /* Adding the points of the n-branch of X in the stencil */
35              for(iOrd = 1; iOrd<=MIN(4,Limxn); iOrd++)
36                d2x += coef[4-iOrd]*Id(P_2,idx-iOrd,idy,idz);
37              /* Adding the points of the p-branch of Y in the stencil */
38              for(iOrd = 0; iOrd<=MIN(4,Limyp); iOrd++)
39                d2y += coef[4-iOrd]*Id(P_2,idx,idy+iOrd,idz);
40              /* Adding the points of the n-branch of Y in the stencil */
41              for(iOrd = 1; iOrd<=MIN(4,Limyn); iOrd++)
42                d2y += coef[4-iOrd]*Id(P_2,idx,idy-iOrd,idz);
43        /* Adding the points of the p-branch of Z in the stencil */
44              for(iOrd = 0; iOrd<=MIN(4,Limzp); iOrd++)
45                d2z += coef[4-iOrd]*Id(P_2,idx,idy,idz+iOrd);
46              /* Adding the points of the n-branch of Z in the stencil */
47              for(iOrd = 1; iOrd<=MIN(4,Limzn); iOrd++)
48                d2z += coef[4-iOrd]*Id(P_2,idx,idy,idz-iOrd);
49        /* Computating the Laplacian */
50              lap = d2x/(dx*dx) + d2y/(dy*dy) + d2z/(dz*dz);
51              /* Wave equation  */
52              Id(P_1,idx,idy,idz)=-Id(P_1,idx,idy,idz)+2*Id(P_2,idx,idy,idz)+dt*dt↩
                    *Id(c_in,idx,idy,idz)*Id(c_in,idx,idy,idz)*lap;
53      }
54  }
```

KERNEL QUE ADICIONA EL APORTE DE LA FRONTERA *LEFT* EN OPENCL

Cuadro de Código 5.32: *Kernel* que adiciona el aporte de la frontera *left* en OpenCL

```
1   __kernel void apply_cpml_left_gpu(int Nx, int Ny, int Nz,\
2               __global float* P_2,          /* Field  P in t       */\
3               __global float* P_1,          /* Field  P in t-d     */\
4               __global float* c_in,         /* Velocity field      */\
5               float  dt,                    /* Time step           */\
6               float  dx,                    /* Spacial resolution  */\
7               float  dy,                    /*     (dx, dy, dz     */\
8               float  dz,                    /*                     */\
9               __global float* coef,         /* 2 deriv. FD coeff   */\
```

```
10              int    L,                    /* CPML  layers        */\
11              __global float*   a_x,       /* Auxiliar variable a */\
12              __global float*   a_x_h,     /* Auxiliar variable a */\
13              __global float*   b_x,       /* Auxiliar variable b */\
14              __global float*   b_x_h,     /* Auxiliar variable b */\
15              __global float*   zeta_left, /* Recursive var. zeta */\
16              __global float*   psi_left,  /* Recursive var. psi  */\
17              __global float* coefd1PML   /* 1 deriv. FD coeff   */){
18
19      /* Declaring variables*/
20      int idx  =  get_global_id(0);
21      int idy  =  get_global_id(1);
22      int idz  =  get_global_id(2);
23      int Limxn, Limxp; /* Limits of the branches of the stencil in x, y, z */
24      int iOrd; /* Index for the each of the branch */
25      float lap; /* Laplacian */
26      float dpsi; /* Derivate of the field psi */
27      float d1x; /* First order derivates */
28      float d2x; /* Second  order derivates */
29      /* Constraining threads */
30      if(idx < L && idy < Ny &&  idz < Nz){
31          /*** Psi field ***/
32          /* Updating the psi field */
33          Limxn = idx;   Limxp = Nx-idx-1;
34          /* Computing Initializing derivates */
35          d1x = 0;
36          /* Adding the points of the  p-branch of x stencil */
37          for(iOrd = 0; iOrd<=MIN(3,Limxp-1); iOrd++)
38              d1x += coefd1PML[iOrd]*Id(P_2,idx+iOrd+1,idy,idz);
39          /* Adding the points of the  n-branch of x stencil */
40          for(iOrd = 0; iOrd<=MIN(3,Limxn); iOrd++)
41              d1x -= coefd1PML[iOrd]*Id(P_2,idx-iOrd,idy,idz);
42          /* Computing the laplacian */
43          lap = d1x/dx;
44          /* Field psi left*/
45          Idlx(psi_left,idx,idy,idz) = b_x_h[L-idx-1]*lap + a_x_h[L-idx-1]*Idlx(↩
                psi_left,idx,idy,idz);
46          /* Synchronizing to computed dPsi */
47          barrier(CLK_GLOBAL_MEM_FENCE);
48          /*** dPsi field ***/
```

```
49          /* Updating the psi field */
50          Limxn = idx-1;    Limxp = L-idx-1+1;
51          /* Computing Initializing derivates */
52          d1x = 0;
53          /* Adding the points of the  p-branch of x stencil */
54          for(iOrd = 0; iOrd<=MIN(3,Limxp-1); iOrd++)
55              d1x += coefd1PML[iOrd]*Idlx(psi_left,idx-1+iOrd+1,idy,idz);
56          /* Adding the points of the  n-branch of x stencil */
57          for(iOrd = 0; iOrd<=MIN(3,Limxn); iOrd++)
58              d1x -= coefd1PML[iOrd]*Idlx(psi_left,idx-1-iOrd,idy,idz);
59          /* Computing the derivate */
60          dpsi = d1x/dx;
61          /*** Zeta field ***/
62          /* Updating the zeta field */
63          Limxn = idx;    Limxp = Nx-idx-1;
64          /* Computing Initializing derivates */
65          d2x = 0;
66          /* Adding the points of the  p-branch of x stencil */
67          for(iOrd = 0; iOrd<=MIN(4,Limxp); iOrd++)
68              d2x += coef[4-iOrd]*Id(P_2,idx+iOrd,idy,idz);
69          /* Adding the points of the  n-branch of x stencil */
70          for(iOrd = 1; iOrd<=MIN(4,Limxn); iOrd++)
71              d2x += coef[4-iOrd]*Id(P_2,idx-iOrd,idy,idz);
72          /* Computing the laplacian */
73          lap = d2x/dx/dx;
74          /* Field psi left*/
75          Idlx(zeta_left,idx,idy,idz) = b_x[L-idx-1]*(lap+dpsi)+a_x[L-idx-1]*Idlx(↩
                zeta_left,idx,idy,idz);
76         /* Wave equation */
77         Id(P_1,idx,idy,idz)+=dt*dt*Id(c_in,idx,idy,idz)*Id(c_in,idx,idy,idz)*(↩
               Idlx(zeta_left,idx,idy,idz)+dpsi);
78
79      }
80  }
```