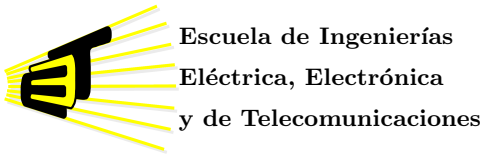
# Medición del desempeño de un algoritmo de inversión de onda completa (FWI) 2D acústica con densidad constante, implementado sobre una unidad de procesamiento gráfico (GPU)

**Juan David Sarmiento Peña**
**Cristian Camilo Garcia Alarcon**



Universidad Industrial de Santander
Facultad de Ingenierías Físico-Mecánicas
Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga
2016

# Medición del desempeño de un algoritmo de inversión de onda completa (FWI) 2D acústica con densidad constante, implementado sobre una unidad de procesamiento gráfico (GPU)

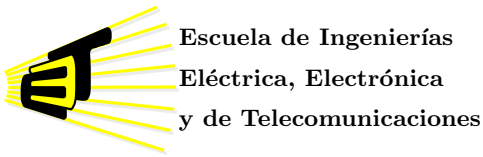
Trabajo de investigación presentado como requerimiento parcial para optar al título de:

Ingeniero Electrónico

**Juan David Sarmiento Peña**
**Cristian Camilo Garcia Alarcon**

Director:
PhD(c).Sergio Alberto Abreo Carrillo

Co-Director:
PhD. Ana Beatriz Ramírez Silva



Universidad Industrial de Santander
Facultad de Ingenierías Físico-Mecánicas
Escuela de Ingeniería Eléctrica, Electrónica y
de Telecomunicaciones
Bucaramanga
2016

# ÍNDICE GENERAL

# ÍNDICE DE FIGURAS

**Pag.**

# ÍNDICE DE TABLAS

**Pag.**

## RESUMEN

**TÍTULO**:

**Medición del desempeño de un algoritmo de inversión de onda completa (FWI) 2D acústica con densidad constante, implementado sobre una unidad de procesamiento gráfico (GPU) *.**

**AUTORES:**

CRISTIAN CAMILO GARCIA ALARCON**
JUAN DAVID SARMIENTO PEÑA**

**PALABRAS CLAVES:**

Inversión de onda completa, métrica, optimización, CUDA-C, unidad de procesamiento gráfico, ocupación, APOD.

El siguiente trabajo de investigación presenta una propuesta enfocada en realizar un análisis y mejora de desempeño de un algoritmo de inversión de onda completa (FWI, por sus siglas en inglés) 2D acústica con densidad constante, implementado sobre una unidad de procesamiento gráfico (GPU, por sus siglas en inglés), específicamente en una Nvidia GeForce GTX 970. Entiéndase como mejora una disminución del tiempo de ejecución, disminución de los requerimientos de memoria y una mejor utilización de los recursos de la GPU (ocupación). El lenguaje de programación usado es CUDA-C, una variación del lenguaje C que permite implementar funciones especiales llamadas kernels que son ejecutadas en la GPU mientras que el código principal es ejecutado en la unidad de procesamiento central (CPU, por sus siglas en inglés), en pocas palabras, la GPU actúa como un co-procesador del procesador central. En el análisis de desempeño se utilizan algunas herramientas de software y Hardware creadas por los desarrolladores de Nvidia, estas herramientas permiten obtener datos de desempeño del algoritmo (métricas), registrar actividades específicas, visualizar líneas de tiempo de las actividades de la CPU y GPU, entre otras. Todas, adquiridas durante tiempo de ejecución. El trabajo realizado se basó en un proceso de cuatro etapas, evaluación, paralelización, optimización e implementación (APOD, por sus siglas en ingles), se hicieron varias iteraciones en las cuales se hacia un pequeño análisis, una propuesta de implementación y una breve evaluación de los resultados obtenidos.


*Trabajo de grado.


**Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Directores Ph.D(c). Sergio Alberto Abreo Carrillo y Ph.D. Ana Beatriz Ramírez Silva.

## ABSTRACT

**TITLE**:

**Measuring the performance of an algorithm of full wave inversion (FWI) 2D acoustic with constant density, implemented on a graphics processing unit (GPU)*.**

**AUTHORS:**

CRISTIAN CAMILO GARCIA ALARCON**
JUAN DAVID SARMIENTO PEÑA**

**KEYWORDS:**

Full wave inversion, metric, optimization, CUDA-C, graphics processing unit, occupancy, APOD.

The following research work presents a proposal focused on perform an analysis and an improvement of performance of an algorithm of full wave inversion (FWI, for its acronym) 2D acoustic with constant density, implemented on a graphics processing unit (GPU, for its acronym), specifically on a Nvidia GeForce GTX 970. To be understood as improvement a decrease in runtime, decrease of memory requirements and a better use of GPU resources (occupancy). The programming language used is CUDA-C, a variation of language C that allows to implement special functions called kernels, which are executed on the GPU while the principal code is executed in the central processing unit (CPU, for its acronym), in other words, the GPU acts as a co-processor of the central processor. In the analysis of performance some software and hardware tools are used created by the Nvidia developers, those tools allow to obtain data of the algorithm performance (metrics), register specific activities, and view time lines of activities of the CPU and GPU, among other. All of them, acquired during execution time. The work done was based on a process of four stages: assess, parallelize, optimize and implement (APOD, for its acronym), several iterations were made in which a small analysis was done, an implementation proposal and a brief evaluation of the results obtained.


*Degree work


** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Directed by Ph.D(c). Sergio Alberto Abreo Carrillo and Ph.D. Ana Beatriz Ramírez Silva.

# INTRODUCCIÓN

En la actualidad las grandes empresas manufactureras de chips hacen más evidente que nunca, que los futuros diseños de microprocesadores y computadores de alto desempeño serán híbridos por naturaleza y se basarán en la integración de dos componentes principales, las unidades de procesamiento central (CPUs) con varios núcleos (Multicore) y el Hardware de propósito especial[22].

Un ejemplo de este Hardware son las unidades de procesamiento gráfico (GPUs) las cuales permiten en conjunto con la CPU acelerar cálculos u operaciones altamente paralelizables. Este hecho en conjunto con nuevas herramientas de programación como CUDA, OpenCL y otras ha permitido encontrar nuevas aplicaciones para las GPUs más allá de su uso en el procesamiento de datos gráficos. Con él ha surgido el concepto de GPGPU o también llamada GPU de propósito general[14].

El concepto GPGPU se basa en pasar de realizar "procesamiento central" en la CPU a un "coprocesamiento" repartido entre la CPU y la GPU, teniendo como resultado un incremento en el rendimiento del sistema.

Este proyecto presenta una propuesta enfocada en el análisis y mejora del desempeño de un algoritmo de inversión de onda completa acústica con densidad constante 2D (FWI) implementado sobre una arquitectura de cómputo paralelo (GPGPU-Nvidia GeForce GTX 970) que usa como lenguaje de programación CUDA-C. Esto se llevará a cabo tomando medidas de diferentes parámetros de desempeño (Métricas) durante la ejecución del algoritmo.

Capítulo 1

# OBJETIVOS

## 1.1. OBJETIVO GENERAL

Mejorar el desempeño del algoritmo FWI 2D durante su ejecución sobre una arquitectura de cómputo en paralelo GPGPU.

## 1.2. OBJETIVOS ESPECÍFICOS

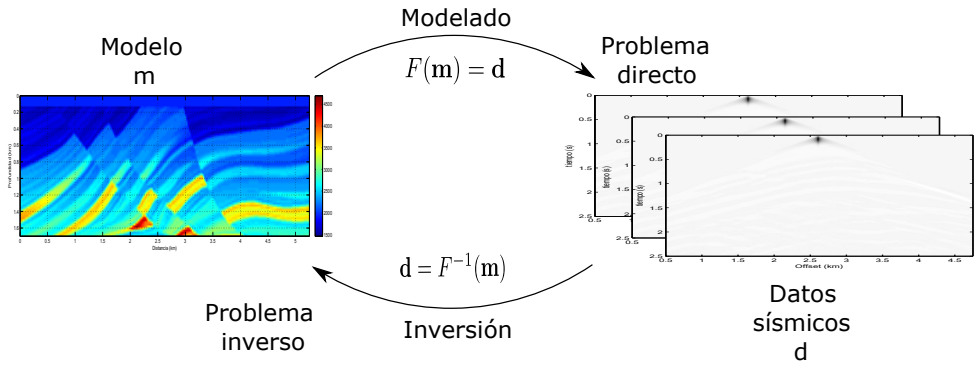
- Seleccionar las métricas que se utilizarán para mejorar el desempeño del algoritmo FWI 2D durante su ejecución sobre una arquitectura de cómputo GPGPU.
- Aplicar las métricas seleccionadas para identificar las partes del código que se deben mejorar.
- Comparar el desempeño de la versión del código mejorada respecto a su versión original.

Capítulo 2

# INVERSIÓN DE ONDA COMPLETA

El objetivo final de la exploración sísmica es hallar modelos del subsuelo (**m**) a partir de datos sísmicos medidos (**d**). Esto requiere la resolución de un problema inverso regido por un operador directo **F**. En la industria de la exploración sísmica se han desarrollado varias metodologías para construir modelos de velocidades de la tierra , las tres categorías importantes son: tomografías de tiempo de viaje (*traveltime tomography*), análisis de la velocidad de migración MVA (*migration velocity analysis*) y inversión de onda completa FWI (*full waveform inversion*) [5].

Figura 1: Relación entre el espacio del modelo **m** y el espacio de datos **d**.




## 2.1. INVERSIÓN DE ONDA COMPLETA

Con el incremento de la capacidad computacional, la inversión de onda completa (FWI) se ha convertido en una herramienta cada vez más práctica para estimar parámetros del subsuelo confiables y de alta resolución, lo cual es el principal objetivo de la exploración sísmica. La FWI actualiza iterativamente un modelo estimado del subsuelo y calcula los correspondientes datos sintéticos **F(m)** con el fin de reducir la diferencia entre los datos medidos **d** y los datos sintéticos (ecuación 2.1)[5].

## 2.2. OBSTÁCULOS DE LA FWI

Aunque la FWI presenta beneficios,también tiene dos grandes obstáculos que impiden su aplicación generalizada en la exploración sísmica.

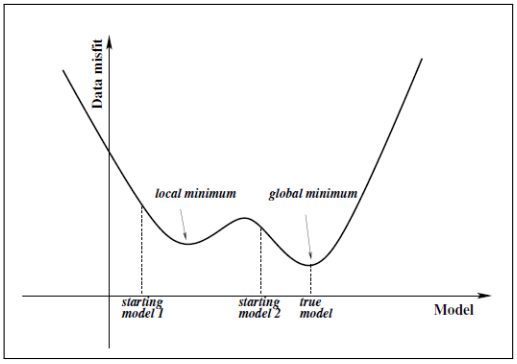
### 2.2.1. Alto costo computacional

La FWI requiere gran cantidad de simulaciones para reconstruir campos de ondas sísmicas, el costo computacional es proporcional al número de fuentes o el número de disparos. Para el caso de experimentos sísmicos de 3D este costo puede ser más elevado. Además la FWI requiere múltiples iteraciones para minimizar el desajuste entre los datos sintéticos y observados haciendo que el costo computacional también sea proporcional al número de iteraciones requeridas[5].

### 2.2.2. No unicidad

El problema inverso planteado por la FWI no tiene solución única. En primer lugar porque la FWI es un típico problema indeterminado, diferentes modelos pueden entrar en una región de modelos igualmente validos con un margen de tolerancia razonable. Este problema de no unicidad es causado principalmente por la presencia de mínimos locales en la función de desajuste **E(m)**. La presencia de mínimos locales es debido a que el operador **F** es generalmente una función no lineal del modelo **m** lo cual genera mínimos locales en la función de desajuste (ecuación 2.1) como se ve en la figura 2. La función de desajuste puede converger en presencia de mínimos locales, al mínimo global si el modelo inicial esta cerca del modelo original. Si el modelo inicial está demasiado lejos del original puede que las iteraciones lleguen a un mínimo local [5][4].

Figura 2: Problema de mínimos locales que presenta la FWI.




Fuente: Full waveform inversion with image-guided gradient [5].

## 2.3. CÁLCULO FWI

La FWI usa los datos sísmicos registrados **d** para estimar parámetros de un modelo del subsuelo **m** con el fin de minimizar la diferencia entre los datos registrados y los datos sintéticos (trazas sísmicas) obtenidos al aplicar el operador **F** sobre los parámetros del modelo **m**. Como la mayoría de los problemas geofísicos de inversión sísmica **F** es una función no lineal de los parámetros del modelo **m**, como por ejemplo la velocidad de las ondas sísmicas. Desafortunadamente el operador **F** no tiene inversa $\mathrm{F}^{-1}$ por ello no se puede simplemente hallar **m** usando $\mathrm{m} = \mathrm{F}^{-1}(\mathrm{d})$, entonces la FWI se formula como un problema de optimización de mínimos cuadrados en la que se busca calcular un modelo **m** que minimice la función de desajuste **E(m)** (ecuación 2.1)

$$E(m) = \frac{1}{2}\|F(m) - d\|_2^2 \tag{2.1}$$

Donde $\|.\|_2$ indica la norma L2. Toda la información en los datos registrados de ondas sísmicas deberían en principio ser tomadas en cuenta dentro de la función de desajuste. Por lo tanto la FWI compara toda la información alrededor de los datos registrados y sintéticos. Este enfoque distingue a la FWI frente a otros métodos como *traveltime tomography* que trabaja solo con los tiempos de llegada de la onda P (primarias o longitudinales).

La expresión que se utiliza para actualizar el modelo del subsuelo es obtenida mediante el método de Newton [5].

$$m_{i+1} = m_i - \alpha_i * {H_i}^{-1} g_i \tag{2.2}$$

Donde $\alpha_i$ es un valor constante que representa el tamaño del paso, $\mathrm{g_i} = \mathrm{g(m_i)} = \frac{\partial \mathrm{E(m_i)}}{\partial \mathrm{m}}$ es el gradiente de la función de desajuste definido por la ecuación 2.3 y $\mathrm{H_i} = \frac{\partial^2 E(m_i)}{\partial^2 m}$ es la matriz Hessiana para el modelo $\mathrm{m_i}$. Sin embargo, en experimentos de gran tamaño la matriz Hessiana $\mathrm{H_i}$, depende de un gran número de parámetros en el modelo lo cual limita la aplicación del método de Newton. El gradiente $\mathrm{g(m_i)}$ se calcula como se propone en [3] donde se define el gradiente de la función de desajuste como

$$g(m_i) = -\sum_s \int_0^T q_s(\mathbf{x}, T-t)\frac{\partial^2 p_s(\mathbf{x}, t)}{\partial t^2} dt \tag{2.3}$$

donde $\mathrm{q_s}$ es llamado propagación hacia atrás del campo de onda usando como fuente los datos residuales, $\mathrm{p_s}$ es la propagación hacia adelante usando las fuentes reales, $T$ es el tiempo de observación, $dt$ es el ancho de los pasos de tiempo, $t$ es la variable de tiempo y $\mathbf{x}$ representa las coordenadas espaciales $(x, y, z)$ dependiendo de las dimensiones usadas en el modelado sísmico.

## 2.4. ECUACIÓN DE ONDA ACÚSTICA ISOTRÓPICA 2D

La ecuación de onda acústica con densidad constante en un medio isotrópico está definida mediante la expresión matemática

$$(c(x,z))^2\nabla^2 P(x,z,t) = \frac{\partial^2 P(x,z,t)}{\partial t^2} \tag{2.4}$$

donde $c$ representa la velocidad acústica del medio, $\nabla^2$ representa el operador laplaciano, $P$ es el campo escalar de presión en función de las coordenadas espaciales $x$, $z$ y $t$ es la variable de tiempo.

### 2.4.1. Planteamiento del problema

A la ecuación de onda acústica con densidad constante en un medio isotrópico dada por la ecuación 2.4, se le debe añadir una función fuente S(t) que perturba el sistema y tiene la siguiente forma

$$S(X,t) = s(t)\delta(X - X_r) \tag{2.5}$$

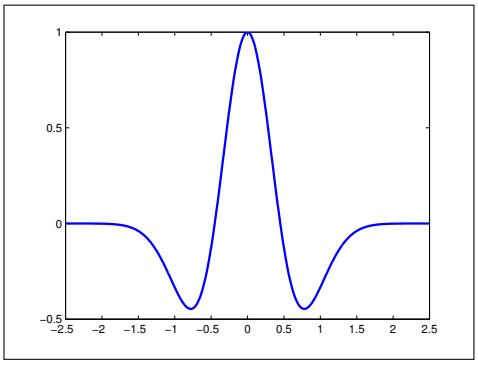
donde $\delta$ es el delta de Dirac, $X_r$ son las coordenadas $(x,z)$ espaciales donde se ubica la fuente, una de las expresiones matemáticas más usada para simular la fuente es la *Ricker wavelet* [10] (ecuación 2.6)

.

$$s(t) = (1 - 2\Pi^2 f^2 t^2)e^{-\Pi^2 f^2 t^2} \tag{2.6}$$

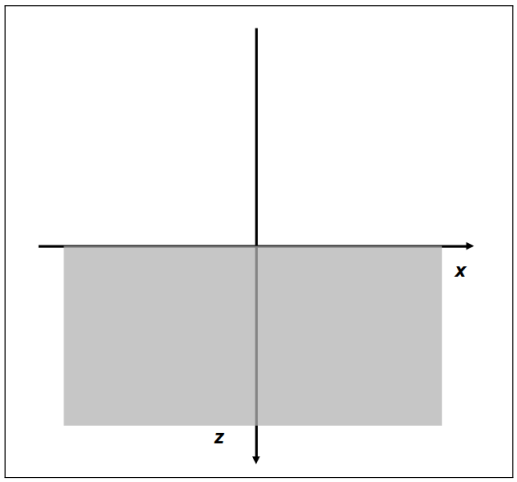
donde $f$ es el pico de frecuencia y $t$ es el tiempo. En la figura 3 se puede visualizar.

Figura 3: Gráfica de la función s(t).




Teniendo en cuenta que el medio de propagación es la Tierra y dado que los levantamientos sísmicos de reflexión son de escala local [8] para esta aplicación se asume como plana la tierra de esta manera se puede establecer un plano cartesiano como sistema de referencia figura 4.
En este sistema de referencia se hace coincidir uno de sus ejes con la superficie terrestre el otro se escoge ortogonal al terreno y su sentido es positivo hacia el interior de la Tierra.

Figura 4: sistema de referencia que se considera para solución 2D.




Ahora adicionando la fuente a la ecuación 2.4 y teniendo en cuenta las dimensiones del modelado $(x, z)$ y el sistema de referencia tenemos que solucionar

$$\begin{aligned} \frac{1}{(c(x,z))^2}\frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial z^2} + S(x,z,t), -\infty \leq x \leq \infty, 0 \leq z \\ P(x,z,0) = 0 \\ \frac{\partial P(x,z,t)}{\partial t}\Big|_{t=0} = 0 \\ P(x,z,t) = 0, z < 0, t \in [0,T], \end{aligned} \tag{2.7}$$

las condiciones iniciales igual a cero indican que el sistema está en reposo.

### 2.4.2. Discretización del problema

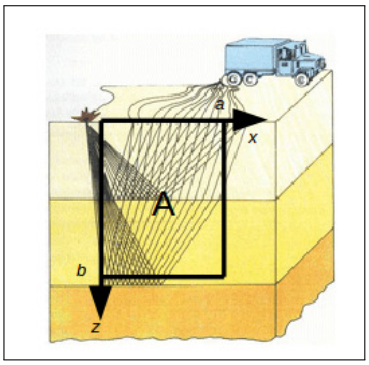
Primero se debe tener en cuenta que el problema (2.7) debe ser solucionado numéricamente para un dominio espacial finito debido a que la memoria computacional es limitada, respecto a lo anterior lo anterior se considera la región rectangular $A = [0, a]\text{x}[0, b]$, ver figura 5. La región es cubierta con una malla uniforme de puntos, espaciados $\triangle x$ en la dirección $x$ y $\triangle z$ en la dirección $z$ (figura 6), estos espaciados se pueden calcular

$$\triangle x = \frac{a}{N_x}, \quad \triangle z = \frac{b}{N_z}, \quad N_x, N_z \in \mathbb{Z} \tag{2.8}$$

de forma similar como la ecuación de onda acústica depende del tiempo $T$ (tiempo de observación) este se divide entre $N_t$ partes.

$$\triangle t = \frac{T}{N_t} \tag{2.9}$$

Figura 5: Dominio espacial considerado en el problema.



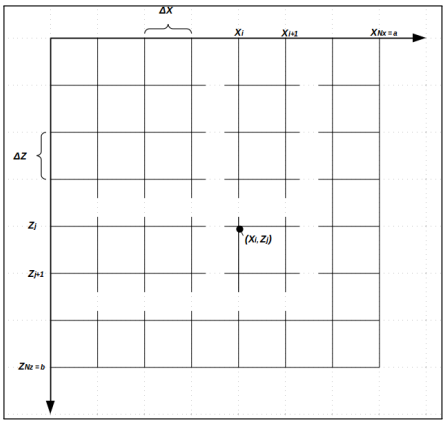

Fuente: http://www.osinerg.gob.pe/ [7].

La solución aproximada entonces estaría denotada por

$$P_{i,j}^{n} = P(x_i, z_j, t_n), \quad i = 0, 1, ..., N_x - 1, \quad j = 0, 1, ..., N_z - 1, \quad n = 0, 1, ..., N_t, \tag{2.10}$$

donde $x_i = i \triangle x, z_j = j \triangle z$ y $t_n = n \triangle t$. ver figura 6.

Figura 6: Discretización espacial.




## 2.5. ECUACIÓN DE ONDA ACÚSTICA ISOTRÓPICA USANDO DIFERENCIAS CENTRADAS

La solución de la ecuación (2.7) se puede aproximar usando diferencias finitas centradas de segundo orden tanto en espacio como en tiempo[11].

$$P_{i,j}^{n+1} = 2P_{i,j}^{n} - P_{i,j}^{n-1} + G^2[P_{i+1,j}^{n} + P_{i-1,j}^{n} + P_{i,j+1}^{n} + P_{i,j-1}^{n} - 4P_{i,j}^{n}] + S_{i,j}^{n} \tag{2.11}$$

donde

$$G = \frac{c_{i,j} \triangle t}{\triangle h} \leq \frac{1}{\sqrt{2}} \tag{2.12}$$

$c_{i,j}$ es la velocidad de propagación en el punto $(x_i, y_j)$, $\triangle h = \triangle x = \triangle z$ es el espaciado de la grilla, $\triangle t$ es el paso de tiempo y $S^n_{i,j}$ es

$$S(x_i, z_j, t_n) = s(t_n)\delta(x_i - x_k, y_i - y_l) \tag{2.13}$$

donde $(x_k, x_l)$ son las coordenadas de los puntos de la grilla que son considerados como fuentes, finalmente teniendo en cuenta las condiciones dadas en la ecuación (2.7) se tiene

$$\begin{aligned} P^n_{i,j} &= 0 \\ P^1_{i,j} &= \begin{cases} S^1_{i,j}, & si \;\; i = k, j = l \\ 0, & si \;\; i \neq k, j \neq l. \end{cases} \end{aligned} \tag{2.14}$$

## 2.6. FRONTERAS NO NATURALES

El dominio espacial para la solución de la ecuación (2.7) se trata de un dominio infinito, pero debido a que la memoria computacional es limitada esto acota la solución a un dominio finito, las condiciones de frontera deben ser tenidas en cuenta en la solución de la ecuación de onda acústica usando alguna técnica de análisis numérico. Se requieren estas condiciones para disminuir los reflejos artificiales en el interior del área de interés [2]. Uno de los métodos usados para incluir las condiciones de frontera en la solución de la ecuación de onda acústica es *Convolutional Perfectly Matched Layer* (CPML)[9]. Ahora para incluir CPML en la ecuación de onda acústica se requiere adicionar dos variables auxiliares en cada dimensión, aplicando este cambio a la ecuación (2.7) tenemos

$$\frac{1}{(c(x,z))^2}\frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial x^2} + S(x,z,t) + \frac{\partial \psi x}{\partial x} + \frac{\partial \psi z}{\partial z} + \zeta x + \zeta z \tag{2.15}$$

La versión discretizada de la ecuación de onda acústica incluyendo CPML esta expresada como sigue

$$P^{n+1}_{i,j} = 2P^n_{i,j} - P^{n-1}_{i,j} + G^2[P^n_{i+1,j} + P^n_{i-1,j} + P^n_{i,j+1} + P^n_{i,j-1} - 4P^n_{i,j}] + S^n_{i,j} + G^2\frac{\triangle h}{2}\psi^n_{st} + (G\triangle h)^2\zeta^n_{st} \tag{2.16}$$

donde

$$\begin{aligned} \psi^n_{st} &= \psi^n_{x,i+1,j} - \psi^n_{x,i-1,j} + \psi^n_{z,i,j+1} - \psi^n_{z,i,j-1} \\ \zeta^n_{st} &= \zeta^n_{x,i,j} + \zeta^n_{z,i,j} \end{aligned} \tag{2.17}$$

Según [9] las variables auxiliares que minimizan el efecto artificial de reflexión en las fronteras están definidas por

$$\psi_q^n = b_q \psi q^{n-1} + a_q \frac{\partial P}{\partial q} \tag{2.18}$$

$$\zeta_q^n = b_q \zeta_q^{n-1} + a_q \left[ \left( \frac{\partial^2 P}{\partial q^2} \right)^n + \left( \frac{\partial \psi_q}{\partial q} \right)^n \right] \tag{2.19}$$

donde $q \in (x, z)$, $\psi_q^0$ y $\zeta_q^0$ son iguales a cero en la primera iteración, $a_q$ y $b_q$ se pueden calcular como sigue.

$$a_q = \frac{d_q}{d_q - \alpha_q}(b_q - 1) \tag{2.20}$$

$$b_q = e^{-(d_q + \alpha_q)\triangle t} \tag{2.21}$$

$$d_q = d_0 V_{max} \left( \frac{f(x)}{Lx} \right)^2 \tag{2.22}$$

$$\alpha_q = \pi f \left( \frac{Lx - f(x)}{Lx} \right) \tag{2.23}$$

Capítulo 3

# UNIDADES DE PROCESAMIENTO GRÁFICO

## 3.1. DEFINICIÓN E HISTORIA

Una unidad de procesamiento gráfico o GPU es un coprocesador dedicado al procesamiento de gráficos u operaciones en coma flotante para aligerar la carga de trabajo del procesador central. El desarrollo de estas se ve influenciado en gran medida por el desarrollo de la industria del videojuego, la cual ejerce una gran presión para mejorar la capacidad de realizar una gran cantidad de cálculos en coma flotante. Esto ha llevado a que los fabricantes de GPUs busquen formas de aumentar el área del chip y la cantidad de energía dedicada a los cálculos. La principal estrategia consiste en explotar un número masivo de flujos de ejecución de tal manera que mientras que unos están en espera para el acceso a memoria, el resto puede seguir ejecutando una tarea pendiente[13].

Las primeras GPUs fueron diseñadas como aceleradores gráficos. Iniciando a finales de 1990, el Hardware se hizo cada vez más programable, llegando finalmente la primera GPU de NVIDIA en 1999. Menos de un año después NVIDIA introdujo el termino GPU. Creadores y diseñadores de juegos no eran los únicos que hacían un trabajo innovador con la tecnología. Investigadores estaban trabajando en el rendimiento de las operaciones de punto flotante. La GPU de propósito general (GPGPU) había nacido.

Pero trabajar con GPGPU no fue nada fácil en aquel entonces incluso para aquellos que conocían lenguajes de programación de gráficos como OpenGL. Los desarrolladores tenían que asignar los cálculos científicos sobre los problemas que podían estar representados por triángulos y polígonos.

La GPGPU estaba prácticamente fuera de los límites de memoria de las ultimas APIs (*Application Programming Interface*) gráficas hasta que un grupo de investigadores de Stanford se dispuso a reinventar la GPU como un *streaming processor*.

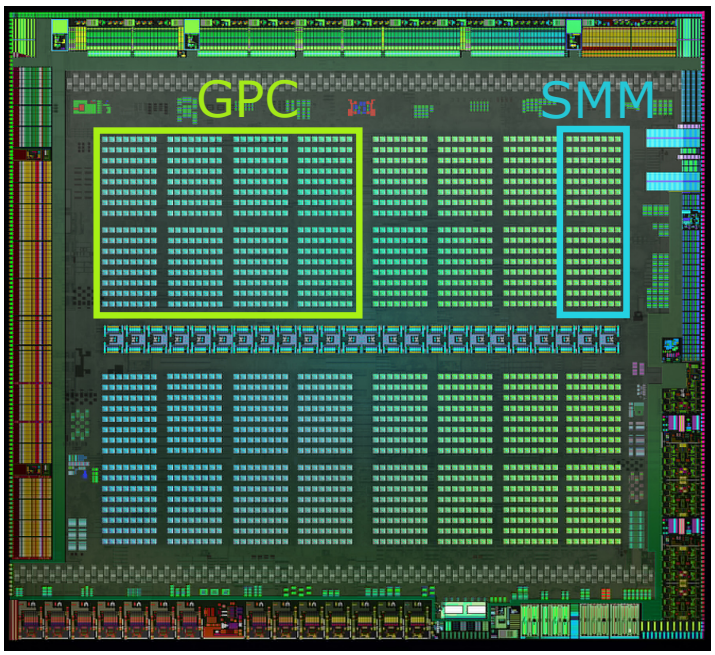
En 2003, un grupo de investigadores dirigido por Lan Buck dio a conocer *Brook* , el primer modelo de programación ampliamente adaptado para extender el lenguaje de programación C con construcciones de datos en paralelo. Usando conceptos tales como *Streaming*, *kernel* y operadores de reducción, el sistema compilador y el tiempo de ejecución *Brook* expone la GPU como un procesador de propósito general en un lenguaje de alto nivel, lo mas importante de los programas en *Brook* era la sencillez de la sintaxis y su desempeño, alrededor de siete veces mayor a codigos existentes similares.

NVIDIA sabía que el Hardware extremadamente rápido tenía que ser acoplado con herramientas de *Software* y *Hardware* intuitivos, e invito a Lan Buck a unirse a la compañía y empezar a desarrollar una solución para implementar sin problemas el lenguaje C en la GPU. Al poner el *Software* y el *Hardware* juntos NVIDIA dio a conocer CUDA (Arquitectura Unificada de Dispositivos de Cómputo) en 2006, siendo esta la primera solución del mundo para uso general sobre una GPU[17].

## 3.2. ARQUITECTURA GPU

La figura 7 muestra la arquitectura Maxwell (*compute capability* 5.$x$) de una GPU NVIDIA. Esta arquitectura está compuesta por un arreglo de *Graphics Processing Clusters* (GPCs), *Streaming Multiprocessor* (SMs) y controladores de memoria.

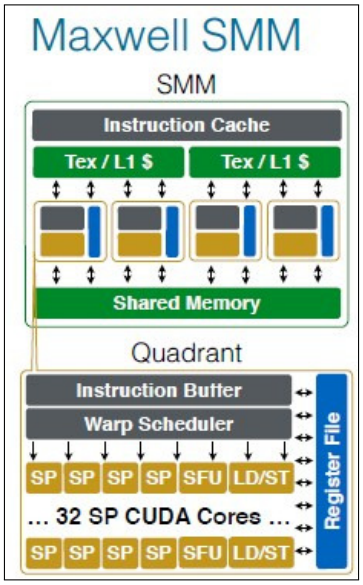
Figura 7: Diagrama de bloques chip GM204 (Geforce GTX 980).




Fuente: NVIDIA Unleashes Maxwell GM204 Based GeForce GTX 980 and GeForce GTX 970 Graphics Cards [24].

El chip GM204 en el caso de la tarjeta Geforce GTX 970 (*compute capability*5,2) está compuesto por 4 *clusters* de procesamiento gráfico (GPCs), 13 multiprocesadores (SMM o SMs) y 4 controladores de memoria (*memory controllers*), cada SMM tiene 128 núcleos CUDA (*cores*), 8 unidades de textura (Tex), su propia región de memoria compartida (*shared memory*), 32 unidades especiales de función (SFU), 4 planificadores de hilos (*warp scheluder*) y unidades de carga y almacenamiento de datos (LD/ST). En total tiene 1664 núcleos y 104 unidades de textura (figura 8). Además esta tarjeta se caracteriza por tener 4 controladores de memoria de 64 bit (256-bit total), cada controlador de memoria tiene 16 ROP (*render output unit*) y 512 KB de L2 cache es decir en total tiene 64 ROPs y 2048 KB de L2 caché[19].

Figura 8: Arquitectura de un SM (*Streaming Multiprocessor*)




Fuente: NVIDIA BRINGS MAXWELL GPUs TO TESLA COPROCESSORS [23].

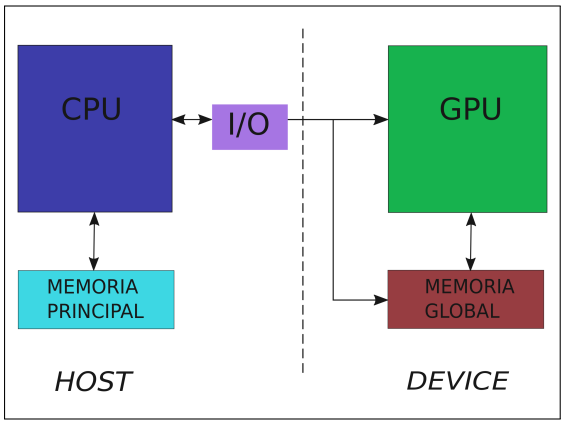
Tabla 1: Comparación entre 3 arquitecturas de GPUs GeForce.

| **GPU specifications** | **GTX 660 Kepler (GK106)** | **GTX 860m Maxwell (GM107)** | **GTX 970 Maxwell (GM204)** |
|---|---|---|---|
| SMs | 5 | 5 | 13 |
| CUDA cores | 960 | 640 | 1664 |
| Base Clock | 980 MHz | 1029 MHz | 1050 MHz |
| GFLOPs | 1881,1 | 1317,1 | 3494 |
| Memory Clock | 6008 MHz | 5000 MHz | 7010 MHz |
| Memory Bandwidth | 144,2 GB/s | 80 GB/s | 224 GB/s |
| TDP | 140 Watts | 40-45 watts | 145 watts |
| Transistors | 2,54 billion | 1,87 billions | 5,2 billions |
| Manufacturing Process | 28 nm | 28 nm | 28 nm |

## 3.3. MODELO DE PROGRAMACIÓN

Primero hay que introducir dos términos importantes, *device* (coprocesador) donde se ejecutan los *threads* o hilos (unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo) y *host* (procesador) donde se ejecuta el programa (figura 9). Esto se conoce como modelo de programación CUDA. CUDA está diseñado para para trabajar con varios lenguajes de programación como por ejemplo C,C++ y Fortran, sin embargo, solo hablaremos de CUDA-C, este permite definir funciones en C llamadas *kernels*, para de esta manera al ser llamados se puedan ejecutar en N hilos diferentes. En principio los *kernels* se ejecutaban de forma secuencial en el *device* pero a partir de la de la arquitectura *Fermi*, es posible ejecutar *kernels* en paralelo.

Figura 9: Interacción CPU (*host*) y GPU (*device*).




Fuente: Estudio de rendimiento en GPU [20].

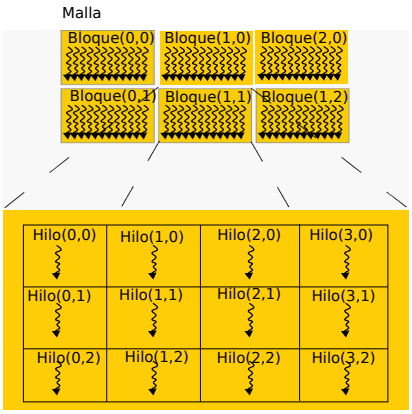
Los hilos se agrupan en arreglos llamados bloques (*blocks*), que pueden ser de 1, 2 o 3 dimensiones. Los bloques a su vez pueden organizarse en arreglos llamados *grid* o malla de bloques, también pueden ser de 1, 2 o 3 dimensiones. Los valores que aparecen entre <<< ... >>> definen las dimensiones del *grid* (dimGrid) y el numero de hilos (dimBlock) de cada bloque.

Listing 3.1: Ejemplo definición *kernel* e invocación .

```
1 /*CODIGO DEVICE--DECLARACION KERNEL*/
2 __global__ void  MatAdd(float *A, float *B, float *C, int nx, int ny){
3       int ix=threadIdx.x + blockIdx.x*blockDim.x;
4       int iy=threadIdx.y + blockIdx.y*blockDim.y;
5       int tid = ix + iy*nx;
6
7       if(ix < nx && iy < ny){
8           C[tid]=A[tid]+B[tid];
9          }
10 }
11 /*CODIGO HOST*/
12 int main(){
13     ...
14        /*Lanzamiento kernel*/
15        MatAdd<<<dimGrid, dimBlock>>>(A_d,B_d,C_d,nx,ny);
16        /*#blocks=dimGrid, #threads=dimBlock*/
17     ...
18 }
```

En la figura 10 cada hilo se identifica por un índice del bloque `blockIdx` y otro índice del propio hilo dentro del bloque `threadIdx`.

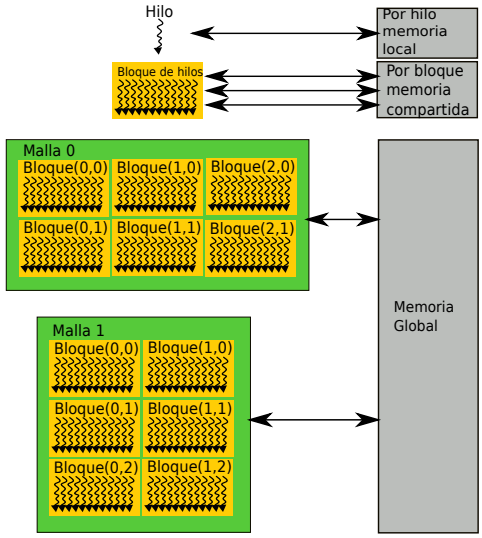
Figura 10: Malla de bloques.




Fuente: CUDA C Programming GUIDE v7.5 [16].

Los hilos acceden a los datos desde múltiples espacios de memoria durante la ejecución del código (figura 11), cada hilo tiene su propia memoria local (*local memory*, $off - chip$). Todos los hilos de un bloque pueden acceder a una clase de memoria especial denominada memoria compartida (*shared memory*, $on - chip$), que tiene el mismo tiempo de vida que la duración del bloque y por ultimo todos los hilos tienen acceso a la memoria global (*device memory*).También existen dos espacios de memoria adicionales que son accesibles por todos los hilos, los espacios de memoria de textura y constantes (ver figura 12) estos espacios junto con la memoria global están optimizados para diversos usos [16].

Figura 11: Jerarquía de memoria.




Fuente: CUDA C Programming GUIDE v7.5 [16].

Tabla 2: Características principales de los tipos de memoria.

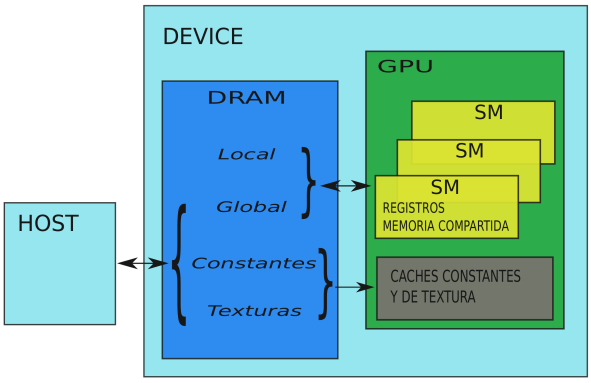
| **Memory** | **Location on/off chip** | **Cached** | **Access** | **Scope** | **Lifetime** |
|---|---|---|---|---|---|
| Register | on | n/a | R/W | 1 thread | Thread |
| Local | off | yes* | R/W | 1 thread | Thread |
| Shared | on | n/a | R/W | All threads in block | Block |
| Global | off | yes** | R/W | All threads + host | Host allocation |
| Constant | off | yes | R | All threads + host | Host allocation |
| Texture | off | yes | R | All threads + host | Host allocation |
| * Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices. of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags. | | | | | |
| ** Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2. | | | | | |

Fuente: CUDA C Best Practices GUIDE v7.5 [15].

### 3.3.1. Acceso a memoria

La arquitectura de la tarjeta gráfica envuelve distintos niveles de memoria algunos *on-chip* (dentro de la GPU) y otros *off-chip* (fuera de la GPU). Desde el punto de vista de la ejecución, las instrucciones a memoria se tratan de forma diferente según a qué nivel se esté accediendo.

Figura 12: Espacios de memoria.




Fuente: CUDA C Best Practices GUIDE v7.5 [15].

#### 3.3.1.1. Memoria compartida

Como se mencionó anteriormente la memoria compartida (*shared memory*) está dentro del *chip*, por esto es mucho más rápida que el espacio de memoria local (*local memory*) y global (*device memory*) e incluso el tiempo de acceso de todos los hilos de un *warp* (conjunto de 32 hilos) accediendo a memoria compartida es tan rápido como el acceso a los registros, siempre y cuando no existan conflictos. La memoria compartida está diseñada en forma de bancos en donde los datos se distribuyen a nivel de palabra. Lo ideal es que el acceso a bancos distintos se pueda realizar de manera simultánea, si esto no sucede y si se accede a datos que están en el mismo banco, se produce conflictos y el acceso se serializa. Esto causa que la petición a memoria sea separada en tantas peticiones como sea necesario para que no existan conflictos, disminuyendo así el ancho de banda efectivo[20].

#### 3.3.1.2. Memoria del dispositivo

En Comparación con la memoria compartida la memoria global es mucho más lenta por que esta fuera del *chip* y es de gran importancia realizar las peticiones de manera eficiente. La memoria global (*device memory*) se divide en tres espacios de memoria separados, memoria global, memoria de textura y memoria de constantes. Tanto la memoria global y de textura son espacios accesibles de lectura y escritura, pero la memoria de constantes es un espacio dedicado únicamente a la lectura (ver tabla 2).

El acceso a memoria global es por segmentos, es decir, incluso si se quiere leer una sola palabra es necesario transferir 32, 64 o 128 bytes a caché, cuando no se utilice todos los datos del segmento se desperdicia ancho de banda. Cuando los hilos acceden de manera ordenada y aprovechan la mayoría de datos transferidos se denomina acceso alineado, unificado o acceso *coalesced*. Realizar accesos no *coalesced* puede producir cuellos de botella por latencia o ancho de banda [16] .

**Memoria local**

Es costoso acceder al espacio de memoria local por que se encuentra dentro del espacio de memoria global. La memoria local se usa de forma automática por el compilador al alojar variables que no caben en registros o que eleven demasiado el número de registros usados por cada hilo (ver tabla 2).

**Memoria constantes**

En el espacio de memoria de constantes una lectura es igual de costosa que una en memoria global sólo si se produce un fallo en caché. En caso de acierto, el coste de acceder a los datos en la caché de constantes es variable, si todos los hilos de medio *warp* acceden a la misma dirección de la caché, el coste es igual que acceder a los registros.

**Memoria de textura**

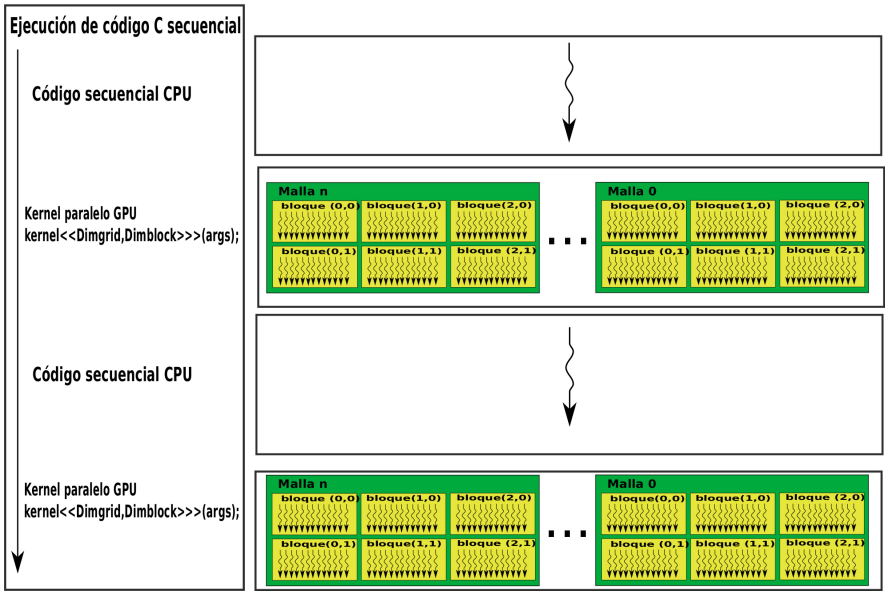
Al igual que la memoria constante, solo accede a memoria global si se produce un fallo en caché. Pero en caso de acierto el coste es distinto. La memoria de texturas esta optimizada para aprovechar la localidad espacial de dos y tres dimensiones. De esta forma, se consigue mejor rendimiento a medida que los hilos del mismo *warp* acceden a direcciones más cercanas de la memoria[20].

## 3.4. PROGRAMACIÓN HETEROGÉNEA

El modelo de programación CUDA asume que los hilos CUDA se ejecutan en un dispositivo (*device*) físicamente separado que opera como un coprocesador al *host* donde se ejecuta el programa C. En este caso un *kernel* es ejecutado en la GPU y el resto de programa C se ejecuta en la CPU[16].

También da por hecho que tanto el dispositivo (*device*) como el *host* tienen su propia memoria dinámica de acceso aleatorio (DRAM) por lo tanto CUDA gestiona los espacios de memoria global, textura y constantes visibles para los *kernels* a través de las llamadas ejecutadas por CUDA. Estas incluyen reserva y liberación de memoria en *device memory*, además se encarga de la transferencia de datos entre *host* y *device*. En la figura 13 se muestra el flujo de programación heterogénea.

Figura 13: Programación Heterogénea.




Fuente: CUDA C Programming GUIDE v7.5 [16].

Los hilos de un mismo *warp* pueden tomar caminos de ejecución diferentes (divergencia de hilos), cuando esto sucede se serializan, ejecutando los hilos de un camino primero y luego los otros. Existen instrucciones para sincronizar todos los hilos de un mismo bloque tales como `__syncthreads()` que actúa como una barrera en la que los *threads* se detienen mientras sus hilos hermanos llegan a donde se invoca esta función. No existe este tipo de herramientas para la sincronización a nivel de malla. *Host* y *device* son asíncronos, la manera de bloquear el *host* hasta que las tareas solicitadas anteriormente en *device* finalicen es mediante el llamado de sincronización explicita `cudaDeviceSynchronize()`.

## 3.5. ANÁLISIS DE DESEMPEÑO

Al desarrollar programas paralelos, es importante evaluar que tan efectivas son las soluciones que se obtienen con estos, en el caso de algoritmos paralelos hay aspectos que no son tan fáciles de medir y es interesante preguntarse si merece la pena el trabajo realizado respecto a los posibles beneficios, para resolver estas preguntas existen diferentes métricas que pueden servir de ayuda.

### 3.5.1. Aceleración

Una de las medidas más importantes para apreciar la calidad de un algoritmo paralelo en multicomputadores y multiprocesadores es la aceleración, la aceleración permite estimar el desempeño de una implementación versus otra.

La aceleración (*Speedup*) que se puede alcanzar al usar cierta mejora está dada:

$$Speedup = \frac{Rendimiento\ sin\ usar\ la\ mejora}{Rendimiento\ al\ usar\ la\ mejora} = \frac{T_a}{T_m} \tag{3.1}$$

Donde:

$T_a$:tiempo de ejecución antiguo.

$T_m$:tiempo de ejecución mejorado.

### Ley de Amdahl

La **Ley de Amdahl**, llamada así por el arquitecto de ordenadores Gene Amdahl, se usa para averiguar la mejora máxima de un sistema cuando solo una parte de éste es mejorado. Establece que: La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente[12].

$$T_m = T_a * \left( (1 - f_m) + \frac{f_m}{Speedup_{mejorado}} \right) \tag{3.2}$$

Según 3.1 tenemos:

$$Speedup_{global} = \frac{1}{(1 - f_m) + \frac{f_m}{Speedup_{mejora}}} \tag{3.3}$$

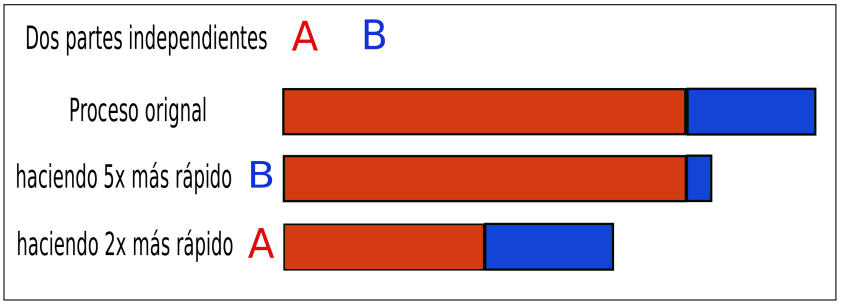
Donde:

$f_m$: es la fracción de tiempo que el sistema usa la parte mejorada.

$Speedup_{mejorada}$: es la aceleración de la parte que puede ser mejorada.

Como ejemplo en la figura 14 se tienen dos procesos independientes, suponiendo que **B** consume el 25 % del tiempo total de cómputo. Haciendo 5 veces más rápido el proceso **B**, el tiempo de cómputo solo se reduce un poco. En contra parte si se logra hacer 2 veces más rápido **A**, el tiempo de cómputo baja más. En conclusión es mejor optimizar aquellas partes del código que consumen un mayor porcentaje de tiempo de cómputo.

Figura 14: Ejemplo Ley de Amdahl.



### 3.5.2. *Profiler CUDA*

*Profiling*, consiste en obtener información de desempeño durante la ejecución de un algoritmo en la GPU. Para esto la GPU dispone de un grupo de contadores de *Hardware* para registrar ciertos aspectos de bajo nivel de los *Kernels* ejecutados. Además de los contadores, el *profiler* CUDA proporciona información acerca del tiempo de ejecución, *occupancy*, uso de registros, uso de memoria compartida entre otros. NVIDIA ofrece un conjunto de herramientas de *profiling* que permiten comprender y optimizar una aplicación implementada en la GPU[18].

#### Terminología

Un **evento** es una actividad o acción que ocurre en *device*. Cuando ocurre un evento en la ejecución de un *kernel* el contador de *Hardware* incrementa su valor en 1. Para ver la lista de eventos disponibles en una determinada GPU NVIDIA, se usa `nvprof --query -events`.

Una **métrica** es una característica de una aplicación que se calcula a partir de uno o más eventos. Para ver la lista de métricas disponible en una determinada GPU NVIDIA, se usa `nvprof --query -metrics`.

Tabla 3: Métricas usadas en el trabajo.

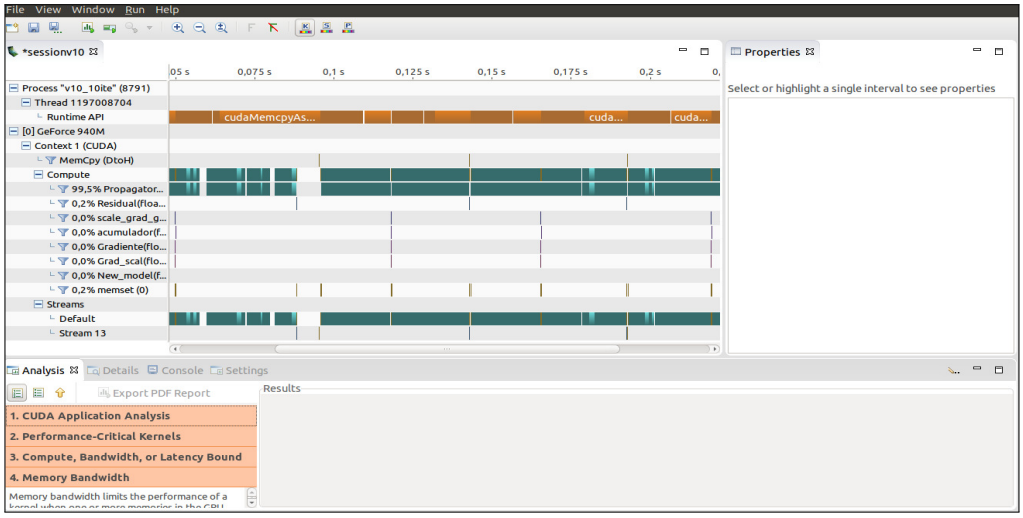
| **Metric name NVVP** | **Metric name nvprof** | **Description** |
|---|---|---|
| Achieved occupancy | achieved_occupancy | Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor. |
| Global load transactions | gld_transactions | Number of global memory load transactions. |
| Global store transactions | gst_transactions | Number of global memory store transactions. |
| Global load throughput | gld_throughput | Global memory load throughput. |
| Global store throughput | gst_throughput | Global memory store throughput |
| Shared store transactions | shared_store_transactions | Number of shared memory store transactions. |
| shared load transactions | shared_load_transactions | Number of shared memory load transactions. |
| Device memory read throughput | dram_read_throughput | Device memory read throughput. |
| Device memory write throughput | dram_write_throughput | Device memory write throughput. |
| Global load efficiency | gld_efficiency | Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage. |
| Global store efficiency | gst_efficiency | Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage. |

Fuente: Profiler User's Guide [18].

### NVIDIA visual Profiler (NVVP)

NVIDIA visual Profiler permite visualizar y analizar el rendimiento de una aplicación ejecutada en una GPU NVIDIA. NVVP muestra la línea de tiempo de las actividades en CPU y GPU de determinada aplicación, facilitando identificar oportunidades de mejora de rendimiento. Además, NVVP permite analizar la aplicación en busca de posibles cuellos de botella y sirve de guía para tomar medidas para reducir o eliminar estos [18].

Figura 15: Entorno NVIDIA visual Profiler.




### NVIDIA Profiling (nvprof)

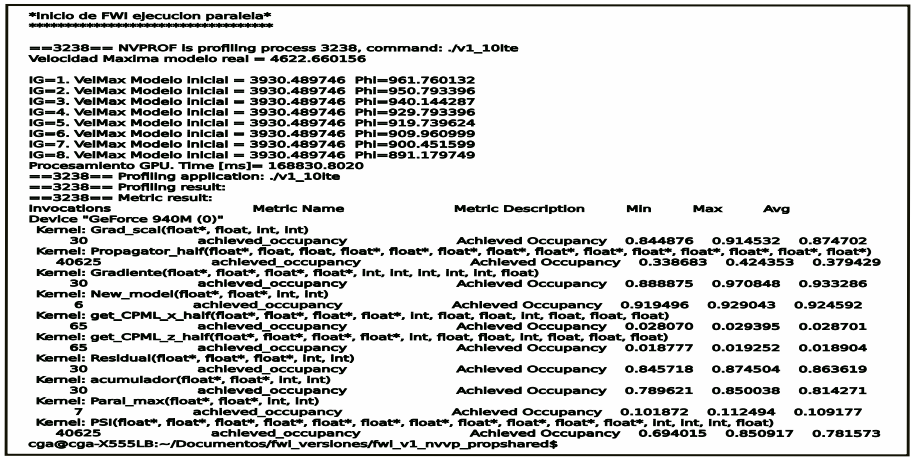
Usando `nvprof` se puede recolectar información de dos tipos[18].

- Permite la obtención de una línea de tiempo de las actividades relacionadas con CUDA, tanto en la CPU y la GPU, incluyendo la ejecución de los *Kernel*, las transferencias de memoria y llamadas a la API de CUDA.
- También permite recopilar eventos o métricas para los *Kernel* ejecutados.

La línea de comando para usar `nvprof` es:

```
nvprof [options] [CUDA-application] [application-arguments]
```

Figura 16: Entorno NVIDIA Profiling.




## 3.6. OPTIMIZACIÓN DE ALGORITMOS EN GPUs

Debido a que la GPU (*device*) está unida a la CPU (*host*) mediante el bus PCI-Express, la transferencia de datos entre CPU y GPU puede resultar costosa. Para que una aplicación sea acelerada de manera correcta en una GPU debe cumplir algunos criterios.

- Que las tareas sean masivamente paralelas. Dividir el problema en muchos subproblemas independientes.
- Que sea computacionalmente intenso.
- Que el tamaño de los datos utilizados por el *kernel* sea limitado. Para poder ser alojado en la GPU.

### 3.6.1. Optimización en CUDA

Mediante la optimización se propone como objetivo mejorar el rendimiento de una aplicación con respecto a versiones anteriores. Se debe llegar a un equilibrio entre paralelización y los demás recursos puestos a disposición para no crear nuevos cuellos de botella. Algunas recomendaciones son:

- Para lograr el máximo rendimiento con CUDA, se debe lograr paralelizar el código.
- Reducir las transferencias de datos entre *host* y *device*, incluso si es necesario ejecutar algunos *Kernels* directamente en la GPU aunque no muestren un mayor rendimiento comparados con la ejecución en la CPU.
- Reducir el uso de la memoria Global. En su lugar se usará la memoria compartida

- Evitar bifurcaciones o diferentes rutas de ejecución dentro del mismo *warp*.
- El acceso a la memoria compartida debe ser diseñado para evitar solicitudes en serie debido a que surgen conflictos de banco.
- Usar la memoria compartida para evitar transferencias redundantes desde la memoria global.
- El número de hilos por bloque debe ser múltiplo de 32 para ofrecer mayor eficiencia de cálculo y permitir el amalgamado de accesos (*coalesced*).
- Usar la biblioteca rápida de matemáticas siempre que la velocidad compense la precisión.

Capítulo 4

# ANÁLISIS Y RESULTADOS

## 4.1. ANÁLISIS PREVIO

La implementación de la FWI consta de una parte serial ejecutada en la CPU y un conjunto de *Kernels* que se ejecutan en la GPU, específicamente en la GTX 970. El análisis de desempeño y las propuestas están enfocadas en gran medida a lograr una mejora del tiempo de ejecución (disminución del tiempo) mediante la optimización de los segmentos del código que se ejecutan en la GPU, debido a que estos representan un alto costo computacional.

Tabla 4: Parámetros del modelo.

| **Parámetro** | **Descripción** | **Valor** |
|---|---|---|
| Nt | número de pasos de tiempo | 625 |
| dt | ancho de paso de tiempo | 4 $ms$ |
| tend | tiempo de observación | 2.5 |
| Nx | número de pasos espaciales en $x$ | 210 |
| dx | ancho de pasos espaciales en $x$ | 25 |
| Ny | número de pasos espaciales en $z$ | 68 |
| dy | ancho de pasos espaciales en $z$ | 25 |
| Ns | número de fuentes | 5 |
| Frec | frecuencia | 3 Hz |
| CPMLimit | ancho área CPML | 20 |

El algoritmo 1 presenta la estructura general de la FWI, las ecuaciones nombradas como *kernel* hacen referencia a las funciones invocadas por el *host* y ejecutadas sobre la GPU, las demás lo hacen exclusivamente en *host*.

El ciclo $for$ (línea 1) representa el número de veces (iteraciones) que se ejecuta el código, de la línea 3 a la 12 se realiza la propagación donde además del *kernel* de Propagator_half están los relacionados con el cálculo de las variables auxiliares para las condiciones de frontera no naturales. De la línea 13 a 20 se calculan los residuales (fuentes de la retro-propagación) y la función objetivo. Posteriormente

**Algorithm 1** Inversión de onda completa $FWI(m, obs, s, inf, \alpha)$. Adaptado de [1]

**for** i $\leftarrow$ 0,iG **do** ▷ iG, Número de iteraciones
Propagación
  **for** j $\leftarrow$ 0, Ns **do** ▷ Ns, Número de fuentes en superficie
    $a_q = \frac{d_q}{d_q - \alpha_q}(b_q - 1)$ ▷ Kernels $get\_CPML\_x\_half$ y
    $b_q = e^{-(d_q+\alpha_q)\triangle t}$ $get\_CPML\_z\_half$
    **for** n,$\leftarrow$ 0,Nt **do** ▷ Nt, Número de pasos de tiempo
      $\psi_q^n = b_q \psi q^{n-1} + a_q \frac{\partial P}{\partial q}$ ▷ Kernel $PSI$

      $\zeta_q^n = b_q \zeta_q^{n-1} + a_q \left[ \left( \frac{\partial^2 P}{\partial q^2} \right)^n + \left( \frac{\partial \psi_q}{\partial q} \right)^n \right]$ ▷ Kernel $Propagator\_half$
      $\frac{1}{(c(x,z))^2} \frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial x^2} + S(x,z,t) + \frac{\partial \psi x}{\partial x} + \frac{\partial \psi z}{\partial z} + \zeta x + \zeta z$
    **end for**
  **end for**
  $acum \leftarrow 0$
  **for** j $\leftarrow$ 0,Ns **do**
    $Residual_j(x,t) \leftarrow Mod_j(x,t) - Obs_j(x,t)$ ▷ Kernel $Residual$
    $norm \leftarrow \|Residual_j(x,t)\|_2$
    $acum \leftarrow \frac{1}{2} norm + acum$

  **end for**
  $\Phi(i) \leftarrow acum$
Retro-propagación
  **for** j $\leftarrow$ 0,Ns **do**
    $a_q = \frac{d_q}{d_q - \alpha_q}(b_q - 1)$ ▷ Kernels $get\_CPML\_x\_half$ y
    $b_q = e^{-(d_q+\alpha_q)\triangle t}$ $get\_CPML\_z\_half$
    **for** n $\leftarrow$ 0,Nt **do**
      $\psi_q^n = b_q \psi q^{n-1} + a_q \frac{\partial P}{\partial q}$ ▷ Kernel $PSI$

      $\zeta_q^n = b_q \zeta_q^{n-1} + a_q \left[ \left( \frac{\partial^2 P}{\partial q^2} \right)^n + \left( \frac{\partial \psi_q}{\partial q} \right)^n \right]$ ▷ Kernel $Propagator\_half$
      $\frac{1}{(c(x,z))^2} \frac{\partial^2 P}{\partial t^2} = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial x^2} + Residual(x,z,Nt-n) + \frac{\partial \psi x}{\partial x} + \frac{\partial \psi z}{\partial z} + \zeta x + \zeta z$

    **end for**
  $g(x,z) = -(\frac{1}{c(x,z)^3}) * \int_0^T q_s(x,z,Nt-n) \frac{\partial^2 p_s(x,z,t)}{\partial t^2} dt$ ▷ Kernel $Gradiente$

  $g(x,z) = k * g(x,z)$ ▷ Kernel $Grad\_scale$
  $g'(x,z) = \sum_j g(x,z)$ ▷ Kernel $acumulador$
  **end for**
  $c(x,z) = c(x,z) - g'(x,z)$ ▷ Kernel $New\_Model$

**end for**
$c_{end} \leftarrow c(x,z)$
**return** $\phi, c_{end}$ ▷ Resultados FWI

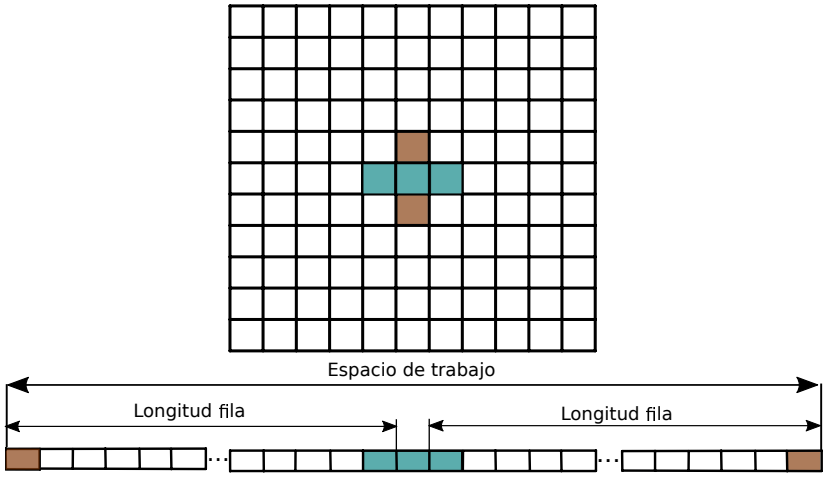
se calcula la retro-propagación (líneas 23 a 31), con estos datos y los obtenidos en la propagación se resuelve el gradiente y con este se actualiza el modelo de velocidades (línea 37).

### 4.1.1. Kernels presentes en la implementación de la FWI

**Propagator_half**

A priori este kernel es el de mayor importancia en el algoritmo, en parte debido a que se ejecuta un número considerable de veces mayor en comparación a otros *kernels*. Para resolver las ecuaciones 2.11 y 2.19, necesita realizar una gran cantidad de accesos a memoria global por hilo (*thread*), además algunos de estos no son los más adecuados para aprovechar la jerarquía de memoria de la GPU (ver figura 17), presenta sincronizaciones (`__syncthreads ()`) innecesarias.

Figura 17: Asignación de stencil en la memoria usando indexado por filas.




Acceder a memoria global puede tomar hasta 400-600 ciclos de reloj [15] lo que produce una alta latencia es decir, los hilos tardan mas accediendo a los datos que procesándolos, la latencia se puede ocultar siempre y cuando hayan suficientes instrucciones aritméticas independientes y se ejecuten una gran cantidad de bloques (*blocks*), sin embargo, lo mejor es evitar el acceso a memoria global siempre y cuando sea posible.

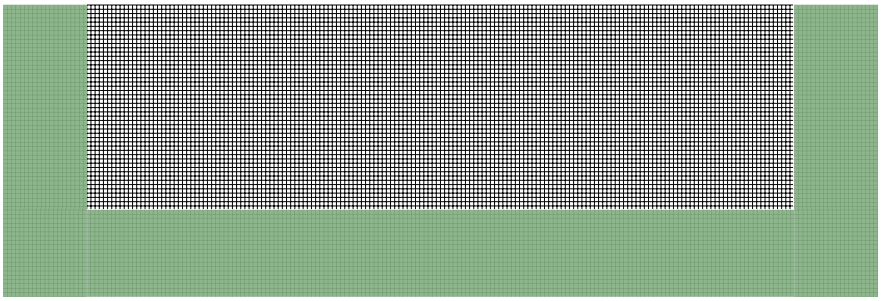
**PSI**

Este *kernel* se ejecuta en igual proporción que el Propagator_half por lo que se supone que es el segundo en importancia, resuelve únicamente la ecuación 2.18 y este es diferente de cero dentro de las fronteras no naturales (CPML), en la figura 18 se observa el área correspondiente a las fronteras no naturales, presenta sincronizaciones (`__syncthreads ()`) innecesarias.

**get_CPML_x_half y get_CPML_z_half**

En comparación a los dos *kernels* anteriormente discutidos, el número de invocaciones es mucho menor, resuelven las ecuaciones de la 2.20 a la 2.23, la diferencia entre los dos es que uno se encarga de la coordenada espacial $x$ y el otro en $z$, el número de hilos lanzados en estos *kernels* son muy pocos, por tanto no se debería apreciar una ventaja significativa al resolverlos en GPU en lugar de la CPU a excepción de evitar copias entre *host* y *device*.

Figura 18: Tamaño modelo (210x68) con CPML de 20 puntos (área sombreada de color verde).



**Residual**

Desde el punto algebraico, realiza una operación punto a punto entre dos vectores, al tener estas características la lectura de memoria global es completamente *coalesced*, al igual que la escritura, no se ejecuta muchas veces por lo que su importancia es significativamente menor a los $Kernels$ anteriores.

**Paral_max**

A simple vista este kernel no muestra una buena implementación en paralelo (gran serialización por hilo). Presenta escrituras a memoria global innecesarias que pueden hacerse en memoria compartida. Sin embargo, representa una importancia menor respecto a PSI y Propagator_half.

**Acumulador**

Al igual que el *kernel* residual, realiza una operación punto a punto entre dos matrices, la lectura y escritura a memoria global es *coalesced*, también tiene menor importancia que PSI y Propagator_half.

**Grad_scal**

Lleva a cabo una operación entre un escalar y una matriz, el acceso a memoria global es *coalesced*, el escalar se guarda en memoria global lo cual no es ideal para valores que tienen que acceder una gran cantidad de hilos, utilizar memoria constante para guardar el escalar puede ayudar a disminuir el tiempo de ejecución, aunque sería necesario realizar una copia a memoria constante cada vez que se vaya a lanzar el *kernel* . Su importancia es reducida en comparación a PSI y Propagator_half.

**New_model**

Actualiza el modelo de velocidades, es de los *kernels* con menos lanzamientos que se realizan por iteración, por lo que su importancia es menor. Por el tipo de operación, soluciona la ecuación 2.2 se supone que su acceso a memoria global es *coalesced*.

**Nota**

Las propuestas realizadas tienen como objetivo mejorar el desempeño del algoritmo (entiéndase como mejora una disminución en el tiempo de ejecución) con el tamaño del modelo establecido previamente sin comprometer este en un modelo mayor, es decir, sin aumentar el consumo de ancho de banda excesivamente, que puede producir un cuello de botella cuando se tiene una mayor cantidad de datos (entiéndase como modelo mayor a un modelo con más puntos).

### 4.1.2. *Profiling* versión original

El algoritmo FWI es un método iterativo esto quiere decir que se ejecuta el mismo código varias veces en este caso 300, no existen diferencias representativas entre una u otra iteración por ello al realizar el análisis de desempeño mediante la herramienta NVIDIA visual profiler se tomaron datos únicamente de dos iteraciones (iG=10 a iG=11), esto con el objetivo de alivianar el trabajo para el $sotfware$, no hay grandes variaciones entre los valores de las métricas obtenidas para 2 o más iteraciones.

Tabla 5: Prioridades versión original FWI

| **Kernel** | **Importancia** | **Duración promedio** | **Invocaciones** | **Registros** |
|---|---|---|---|---|
| Propagator_half | 68.8 % | 13.862 $\mu$s | 12500 | 41 |
| Psi | 28.7 % | 5.825 $\mu$s | 12500 | 26 |
| Gradiente | 2.1 % | 543.535 $\mu$s | 10 | 23 |
| get_CPML_x_half | 0.1 % | 9.006 $\mu$s | 20 | 36 |
| get_CPML_z_half | 0.1 % | 8.107 $\mu$s | 20 | 38 |
| Residual | 0.0 % | 9.513 $\mu$s | 10 | 8 |
| Paral_max | 0.0 % | 42.064 $\mu$s | 2 | 11 |
| acumulador | 0.0 % | 4.064 $\mu$s | 10 | 8 |
| Grad_scal | 0.0 % | 2.934 $\mu$s | 10 | 8 |
| New_model | 0.0 % | 3.264 $\mu$s | 2 | 8 |

La suposición realizada en análisis previo es acertada, los *kernels* PSI y Propagator_half representan más del 90 % del tiempo de ejecución total sobre la tarjeta gráfica, Propagator_half es el *kernel* con mayor importancia en la FWI. Respecto a la utilización de la GPU ($ComputeUtilization$) este es relativamente bajo relacion al tiempo total de ejecución (55.5 %).

**Limite de ocupación teórica en `Propagator_half`**

La NVIDIA GeForce GTX 970 cuenta con 65536 registros por SM lo que permite una máxima cantidad de 65536 por bloque ejecutando un único bloque por SM, además la cantidad de hilos máximos por SM son 2048. Anteriormente se mencionó que el número de registros por hilo son 41, los bloques utilizados son de un tamaño de 32x16 (512 hilos) es decir que cada bloque utiliza 20992 registros más 3584 registros adicionales. Lo que limita el lanzamiento de solo 2 bloques (1024) hilos y por tanto solo se alcanza una ocupación teórica del 50 %.

Se limitó el número de registros (32 registros por hilo) usando la bandera `-maxrregcount` al momento de la compilación, esto con el fin de permitir una ocupación teórica del 100 %, la ocupación medida aumento a un valor de 43.2 % pero el tiempo promedio del *kernel* aumento a 36.426 $\mu$s lo que incrementa el tiempo total de ejecución. Se concluye que disminuir el número de registros no representa una mejora para el algoritmo en este punto, puesto que la ocupación sigue siendo baja, esto es debido a que el *kernel* no ejecuta suficientes bloques para ocultar la latencia por memoria y operaciones [18], se recomienda que el tamaño de la malla (*grid*) sea suficientemente grande para llenar a GPU con múltiples olas de bloques, la tarjeta gráfica puede ejecutar simultáneamente 4 bloques por cada SM entonces el *kernel* debería ejecutar múltiplos de 52 bloques con el fin de ocultar la latencia. Disminuir el tamaño de los bloques no es una solución factible puesto que al hacer esto los SMs podrán ejecutar más bloques simultáneamente, en conclusión el tamaño del modelo es pequeño para GPU.

## 4.2. OPTIMIZACIONES

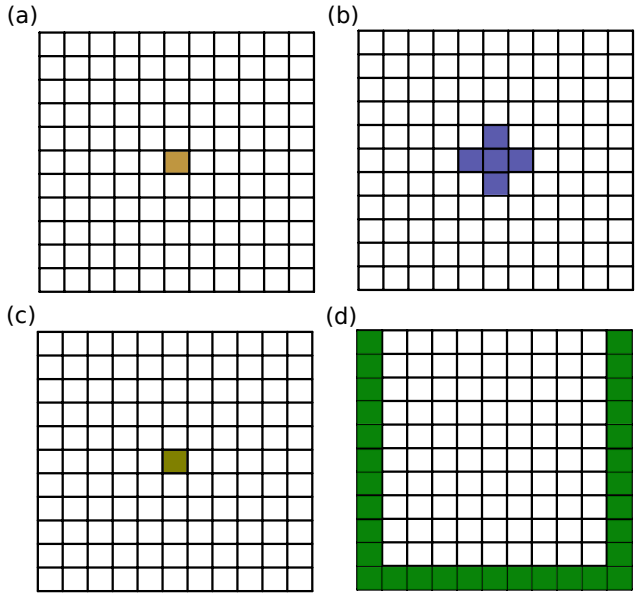
### 4.2.1. Primera fase: Disminución de accesos a memoria global haciendo uso de memoria compartida

Basándose en el análisis inicial se determina que:

- El kernel con mayor importancia es el Propagator_half.
  - Realiza demasiados accesos de lectura y escritura a memoria global.
  - Su ocupación es baja (39.87 %), en parte debido a que utiliza bastantes registros por hilo (41 registros por hilo), limitando la ocupación teórica al 50 %.

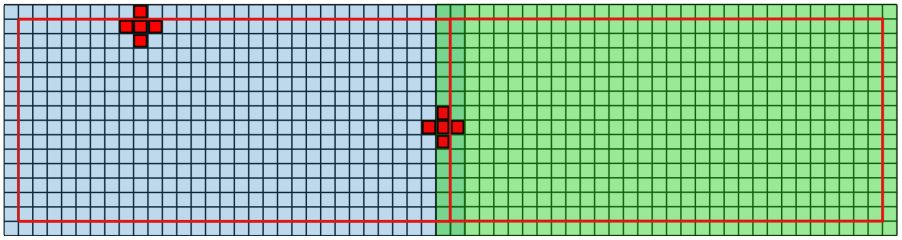
Con el fin de evitar la gran cantidad de accesos a memoria global presentes se propone utilizar memoria compartida para calcular, guardar y almacenar datos que posteriormente serán usados o que se utilizaran varias veces en el *kernel*, por ejemplo la matriz presente (ecuación 2.16).

Figura 19: Cálculo del campo de presión (propagación). a) Punto en pasado b) Puntos en presente. c) Punto en futuro calculado a partir de los valores en pasado y presente. d) Fronteras no naturales (CPML) .




Cada hilo solo se encarga de traer de memoria del dispositivo un solo dato a memoria compartida, esto implica que los hilos que se encuentren en los bordes del bloque no realizan el cálculo del *stencil* y solo son utilizados para traer el dato de memoria . Para un tamaño de bloque $N$x$M$ (tamaño original del bloque), solo podrán ser solucionados $(N-1)$x$(M-1)$ puntos, y por esto es necesario lanzar más bloques. En la figura 20 dentro de los recuadros rojos se ven los accesos realizados a memoria compartida por los hilos adyacentes a los bordes.
Se eliminan las llamadas a `__syncthreads()` innecesarias.

Figura 20: Bloques de 32x16 hilos (NxM) adyacentes, dentro del área encerrada por la linea roja los elementos que en realidad se usan para el cálculo de la propagación.

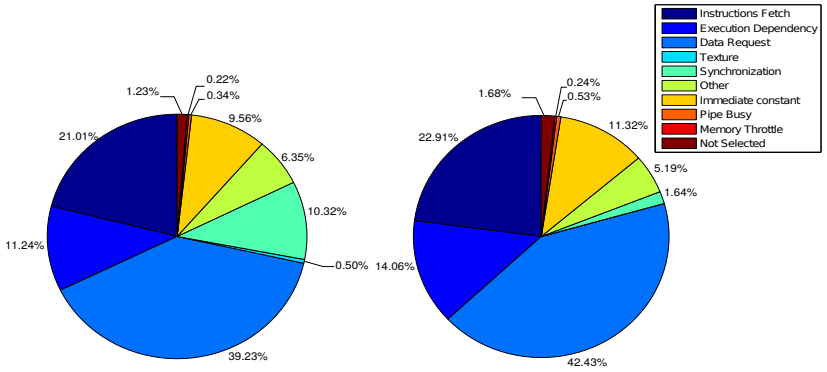


Se realiza la comparación de algunas métricas seleccionadas para evaluar el efecto de los cambios realizados respecto a la FWI original (tabla 6).

Tabla 6: Comparación de métricas Propagator_half original y propuesto con memoria compartida.

| Metric | Propagador original | | | Propagador shared memory | | |
|---|---|---|---|---|---|---|
| | Min | Max | Avg | Min | Max | Avg |
| Achieved occupancy | 0.369 | 0.417 | 0.3987 | 0.328 | 0.418 | 0.365 |
| Global load transactions | 104962 | 105583 | 105269 | 62964 | 63382 | 63172 |
| Global store transactions | 845 | 8448 | 8473 | 8383 | 8409 | 8395 |
| Global load throughput | 81.315GB/s | 105.291GB/s | 93.296GB7s | 46.714GB/s | 61.316GB/s | 53.737GB/s |
| Global store throughput | 12.117GB/s | 15.667GB/s | 13.888GB/s | 13.185GB/s | 17.238GB/s | 15.136GB/s |

Figura 21: *Issue stall reasons*, a la izquierda Propagator_half original, la derecha propuesta .




De la tabla 6 y la figura 21 se concluye que:

- Disminuyo el tiempo de ejecución del *kernel*, por consiguiente el tiempo de ejecución total.
- Disminuyen tanto las transacciones (carga y escritura) a memoria global como el ancho de banda utilizado por el *kernel* por consiguiente no compromete este cuando se usen modelos más grandes.
- Disminuye la ocupación debido a disminución de hilos por bloque que realizan el cálculo de la propagación.
- Aumenta la cantidad de registros por hilo, continuando limitada la ocupación por esta razón (44 registros por hilo).
- Al eliminar las sincronizaciones el porcentaje de paradas (*issue stall reasons*) provocadas por estas se reducen dramáticamente.
- Las paradas por dependencia de datos (*execution dependency*) a pesar de haber disminuido las transacciones a memoria global siguen siendo el factor más importante.

Tabla 7: Prioridades primera fase.

| **Kernel** | **Importancia** | **Duración promedio** | **Invocaciones** |
|---|---|---|---|
| Propagator_half | 65.2 % | 11.624 $\mu$s | 12500 |
| Psi | 31.8 % | 5.680 $\mu$s | 12500 |
| Gradiente | 2.4 % | 535.676 $\mu$s | 10 |

No se presentan grandes cambios en las prioridades de los *kernels*, el Propgator_half sigue siendo el de mayor relevancia. Se reduce el *compute utilization* debido a la disminución del tiempo de ejecución en la GPU.

Tabla 8: Aceleración fase 1

| **Versión FWI** | **Utilización de la GPU** | **Tiempo total** | **Aceleración** |
|---|---|---|---|
| Original | 55.5 % | 49.996 s | |
| Primera fase | 52.5 % | 46.422 s | 1.076x |

### 4.2.2. Segunda fase: Unión de *kernels* propagador y psi

Existe una dependencia entre los *kernels* Propagator_half y Psi. Con el fin de disminuir los accesos a memoria global y evitar el tiempo entre lanzamientos, se opta por realizar un solo kernel que realice las tareas de las dos funciones previas, una posible desventaja de la implementacion propuesta es que el número de hilos que resuelven la propagación disminuya, se soluciona solo $(N-2)$x$(M-2)$ puntos y por consiguiente es necesario lanzar más bloques. Se extiende el uso de memoria compartida a aquellas variables que se tienen que acceder varias veces en este nuevo *kernel* (se referirá a él como Prop_psi_sh), están son las variables auxiliares de las fronteras no naturales (ecuaciones 2.18 y 2.19). Otras desventaja es la necesidad de usar sincronización entre el cálculo de $\psi$ y $\zeta$ para evitar condiciones de carrera.

Figura 22: Al interior del recuadro de negro los datos que pueden ser calculados de Psi y en el recuadro rojo los datos que pueden ser calcular de la propagación y zeta.

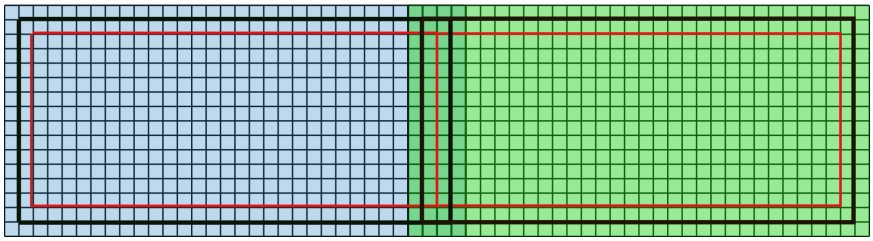


Tabla 9: Métricas Propagator_half con memoria compartida y Psi original.

| Metric | Propagador shared memory | | | Psi | | |
|---|---|---|---|---|---|---|
| | Min | Max | Avg | Min | Max | Avg |
| Achieved occupancy | 0.328 | 0.418 | 0.365 | 0.363 | 0.467 | 0.441 |
| Global load transactions | 62964 | 63382 | 63172 | 15366 | 15366 | 15366 |
| Global store transactions | 8383 | 8409 | 8395 | 1932 | 1932 | 1932 |
| Global load throughput | 46.714GB/s | 61.316GB/s | 53.737GB/s | 24.161GB/s | 39.001GB/s | 32.698GB/s |
| Global store throughput | 13.185GB/s | 17.238GB/s | 15.136GB/s | 5.672GB/s | 9.156GB/s | 7.677GB/s |

Tabla 10: Métricas *kernels* Prop_psi_sh fase 2

| Metric | Prop_psi_sh | | |
|---|---|---|---|
| | Min | Max | Avg |
| Achieved occupancy | 0.59 | 0.693 | 0.663 |
| Global load transactions | 35548 | 35602 | 35573 |
| Global store transactions | 8327 | 8330 | 8328 |
| Global load throughput | 30.931GB/s | 43.41GB/s | 39.02GB/s |
| Global store throughput | 14.441GB/s | 20.241GB/s | 18.208GB/s |

De las tablas 9 y 10 se concluye que:

- Se logra disminuir los accesos a memoria global, especialmente los de lectura respecto al Propagator_half, aun teniendo que resolver más cálculos debido a la unión de las dos funciones. Reducción del ancho de banda total utilizado.
- Aumento de la ocupación por:
  - El uso de registros por hilo es menor (31 registros por hilo) permitiendo una ocupación teórica de 100 %.
  - Se deben lanzar más bloques en consecuencia la GPU ejecuta más hilos, sin embargo, esto realmente no significa una mejora.

Disminuye a casi la mitad el porcentaje de paradas por dependencia de datos (*data request*) lo que quiere decir que los hilos no pierden tanto tiempo esperando los datos que necesitan para ejecutar las tareas asignadas (ver figura 23) . Respecto a paradas por dependencia de ejecución *execution dependency* estas se deben a que hay una dependencia entre las variables que tienen que resolver los hilos, en otras palabras es necesario primero resolver las variables auxiliares $\psi$ y $\zeta$ en ese orden para poder calcular después el campo de presión futuro.

Tabla 11: Prioridades fase 2

| Kernel | Importancia | Duración promedio | Invocaciones |
|---|---|---|---|
| Prop_psi_sh | 95.6 % | 11.570 $\mu$s | 12500 |
| Gradiente | 3.6 % | 538.693 $\mu$s | 10 |

Figura 23: *Issue stall reasons*, a la izquierda Prop_psi_sh original, la derecha Propagator_half fase 1.

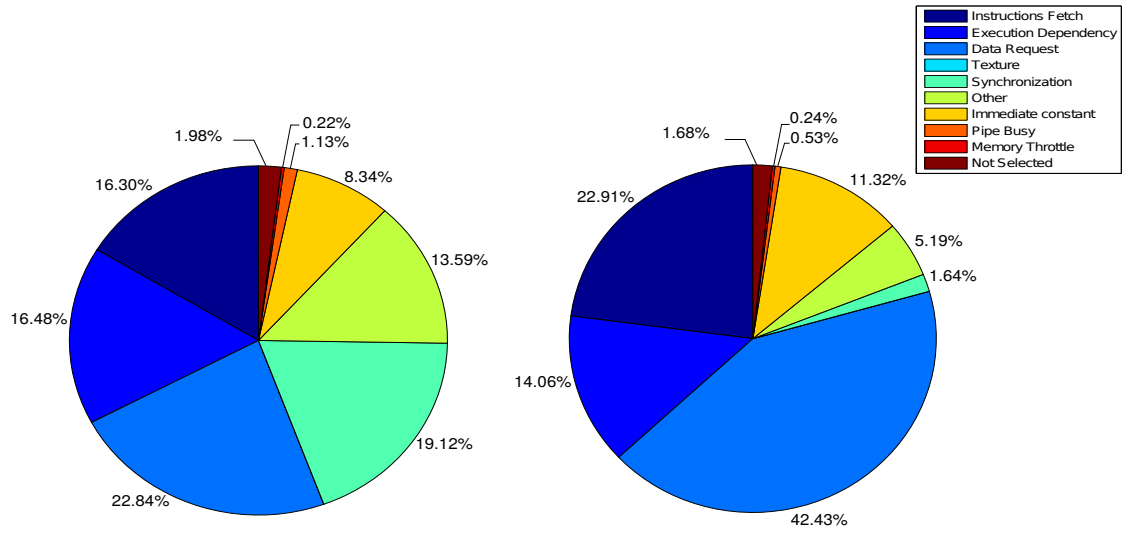



Tabla 12: Aceleración fase 2

| **Versión FWI** | **Utilización de la GPU** | **Tiempo total** | **Aceleración** |
|---|---|---|---|
| Original | 55.5 % | 49.996 s | |
| Primera fase | 52.5 % | 46.422 s | 1.076x |
| Segunda fase | 43.5 % | 36.866 s | 1.356x |

De las tablas 11 y 12 se determina que:

- Una considerable mejora en el tiempo de ejecución gracias a:
  - Se omite el tiempo de lanzamiento entre los *Kernels*.
  - Tiempo promedio de ejecución de este nuevo *kernel* es menor, respecto a suma de los dos anteriores.
- Disminución del *compute utilization*
  - Al disminuir el tiempo de ejecución total de la GPU, de igual manera disminuye el porcentaje entre el tiempo de ejecución de la GPU y el tiempo total.
  - El tiempo Serial, es decir, el tiempo en CPU presenta una pequeñá disminuciòn, mucho menos notable que el de la GPU, esta posiblemente se produce por la reducción de llamadas a la API de CUDA (invocaciones de *kernels* y otros).

### 4.2.3. Tercera fase: Reducción del tiempo de ejecución en CPU

Basándose en el hecho de que el lanzamiento de los *kernels* en el $default\ stream$ tienen sincronización implícita con CPU y que las funciones de asignación de un valor (`memset`) y copias entre *host* y *device* (`cudaMemcpy`) utilizadas en el algoritmo también son síncronas se puede evitar el uso de banderas de sincronización (`cudaDeviceSynchronize`) lo que debería disminuir el tiempo de ejecución en CPU y por consiguiente aumentar el *compute utilization*[16].
Al transferir las variables auxiliares a memoria compartida solo es necesario leer, operar y escribir en los espacios de memoria en donde estas variables son diferentes de cero(ver figura 18). Este cambio se evidencia en la disminución de transacciones de memoria global y compartida (tabla 13).

Tabla 13: Comparación de métricas nuevo *kernel* (Prop_psi_sh_less) y Prop_psi_sh (fase 2)

| **Metric** | **Prop_psi_sh** | | | **Prop_psi_sh_less** | | |
|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Avg** | **Min** | **Max** | **Avg** |
| Achieved occupancy | 0.59 | 0.693 | 0.663 | 0.528 | 0.647 | 0.609 |
| Global load transactions | 35548 | 35602 | 35573 | 29215 | 29269 | 29240 |
| Global store transactions | 8327 | 8330 | 8328 | 6331 | 8561 | 7445 |
| Global load throughput | 30.931GB/s | 43.41GB/s | 39.02GB/s | 26.486GB/s | 39.99GB/s | 34.552GB/s |
| Global store throughput | 14.441GB/s | 20.241GB/s | 18.208GB/s | 12.048GB/s | 22.099GB/s | 17.511GB/s |
| Shared store transactions | 5924 | 5924 | 5924 | 3933 | 3933 | 3993 |
| shared load transactions | 11864 | 11895 | 11879 | 10060 | 10091 | 10075 |

De las tablas 14 y 15 se determina que:

- Se reduce el tiempo ejecución total en gran medida por la reducción del tiempo de ejecución en CPU.
- Aumenta el *compute utilization* esto es de esperar pues disminuye el tiempo en CPU.

Tabla 14: Prioridades fase 3

| **Kernel** | **Importancia** | **Duración promedio** | **Invocaciones** |
|---|---|---|---|
| Prop_psi_sh_less | 95 % | 10.088 $\mu$s | 12500 |
| Gradiente | 4.1 % | 538.153 $\mu$s | 10 |

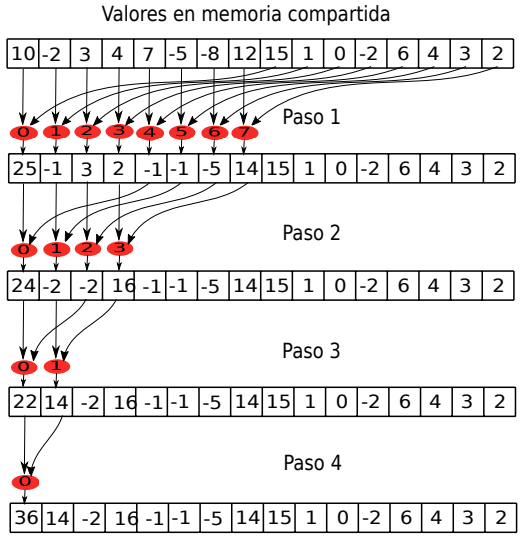
Tabla 15: Aceleración fase 3

| **Versión FWI** | **Utilización de la GPU** | **Tiempo total** | **Aceleración** |
|---|---|---|---|
| Original | 55.5 % | 49.996 s | |
| Primera fase | 52.5 % | 46.422 s | 1.076x |
| Segunda fase | 43.5 % | 36.866 s | 1.356x |
| Tercera fase | 75.5 % | 23.328 s | 2.143x |

### 4.2.4. Cuarta fase: Producto punto de forma paralela

Dentro de la implementación de la FWI se encuentra la función *scale_grad* que realiza la operación de producto punto, la cual se puede realizar de manera paralela. Además la matriz necesaria para este cálculo se copiaba de *device* a *host*, esto no es recomendable, por ello se opta por realizar el producto punto en la GPU.

Figura 24: Sumatoria de manera paralela por bloque.




Es necesario ejecutar esta operación en dos pasos, primero cada bloque realiza el producto entre cada posición de sus datos (ver figura 24) y realiza una parte de la sumatorio de ellos, luego se vuelve a ejecutar con los resultados obtenidos anteriormente utilizando solo un bloque, esto es necesario debido a que no hay forma de sincronizar los bloques entre sí. Teóricamente al hacer la paralelización de la

Tabla 16: Aceleración fase 4

| **Versión FWI** | **Utilización de la GPU** | **Tiempo total** | *Aceleración* |
|---|---|---|---|
| Original | 55.5 % | 49.996 s | |
| Primera fase | 52.5 % | 46.422 s | 1.076x |
| Segunda fase | 43.5 % | 36.866 s | 1.356x |
| Tercera fase | 75.5 % | 23.328 s | 2.143x |
| Cuarta fase | 75.2 % | 23.233 s | 2.151x |

función debería aumentar el *compute utilization* sin embargo, se presenta una pequeña disminución en este probablemente debido al tiempo de preparación para el lanzamiento del *kernel*. Aunque la función paralelizada es alrededor de 6 veces más rápida esto no provoca cambios grandes en el tiempo de ejecución esto como consecuencia de la baja prioridad de este *kernel* dentro del código (prioridad menor al 0.1 %).

Tabla 17: Aceleración scale_grad

| Funciones scale_grad | Tiempo | Aceleración |
|---|---|---|
| Serial | 53.3 $\mu$ s | |
| Paralelo | 4.28$\mu$ s | 6.226x |

La Aceleración en este caso es igual $tiempo\_serial/(2 * tiempo\_paralelo)$, esto es por que para obtener el resultado al ejecutar una vez la función serial hay que ejecutar dos veces el *kernel*.

### 4.2.5. Reorganización del código FWI

Al reservar los espacios de memoria (`cudaMalloc()`) necesarios en la GPU para la implementación de la FWI del algoritmo 1 basta con hacerlo al inicio del bucle de iteraciones ($iG$) esto para algunas variables y liberar el espacio de memoria (`cudaFree()`) después que este termine, inicializando en ceros los espacios de memoria cuando sea necesario (`cudaMemset*()`) dentro de la ejecución del código. La variable de velocidad máxima necesaria para resolver las ecuaciones 2.20 y 2.21 originalmente es actualizada en cada iteración y por tanto estas ecuaciones también. Para efectos de las características del modelado sísmico esta velocidad es constante para cada iteración dando como resultado valores de $a\_q$ y $b\_q$ iguales en cada iteración por ello solo es necesario calcularlas al inicio del FWI.

Tabla 18: Aceleración FWI reorganizada.

| Versión FWI | Utilización de la GPU | Tiempo total | Aceleración |
|---|---|---|---|
| Original | 55.5 % | 49.996 s | |
| Primera fase | 52.5 % | 46.422 s | 1.076x |
| Segunda fase | 43.5 % | 36.866 s | 1.356x |
| Tercera fase | 75.5 % | 23.328 s | 2.143x |
| Cuarta fase | 75.2 % | 23.233 s | 2.151x |
| FWI cuarta fase reorganizada | 82.5 % | 21.198 s | 2.358x |

### 4.2.6. Quinta fase: Disminución de los requerimientos de memoria RAM (DRAM)

En el *kernel* gradiente el acceso de lectura a los datos almacenados en las películas de propagación ($p_s$) y de retro-propagación ($q_s$) como tal es *coalesced*, sin embargo, los hilos deben acceder continuamente a memoria global lo que puede ocasionar embotellamientos por ancho de banda de DRAM, por ello el *kernel* está limitado por el ancho de banda de la GPU, esto produce que no se puedan proporcionar los datos a la velocidad requerida por el *kernel* (figura 25). La propuesta presentada tiene como objetivo disminuir el ancho de banda utilizado y los requerimientos de memoria RAM, es decir, la cantidad de memoria RAM que se utiliza en la ejecución de la FWI [1] (ecuación 4.1). Al momento de realizar la retro-propagación se puede calcular directamente la integral presente en la ecuación 2.3 y así evitar el

uso del espacio de memoria reservado para $q_s$ (ver figura 25).

$$Ram\ size = \frac{\beta}{1024^2 * 8}(N_s * N_t * N_{traza} + N_t + 11 * N_x * N_z + N_x * N_t + 2 * N_x * N_z * N_t + 4 * N_x + 4 * N_z + N_t * N_{traza}) \tag{4.1}$$

Con $N_{traza}$=170 y $\beta$=32 y los demás parámetros dados en la ecuación 4.1 el requerimiento de memoria RAM original es de 71.631 MB.

Figura 25: Películas de Propagación ($p_s$) y retro-propagación ($q_s$) originales.

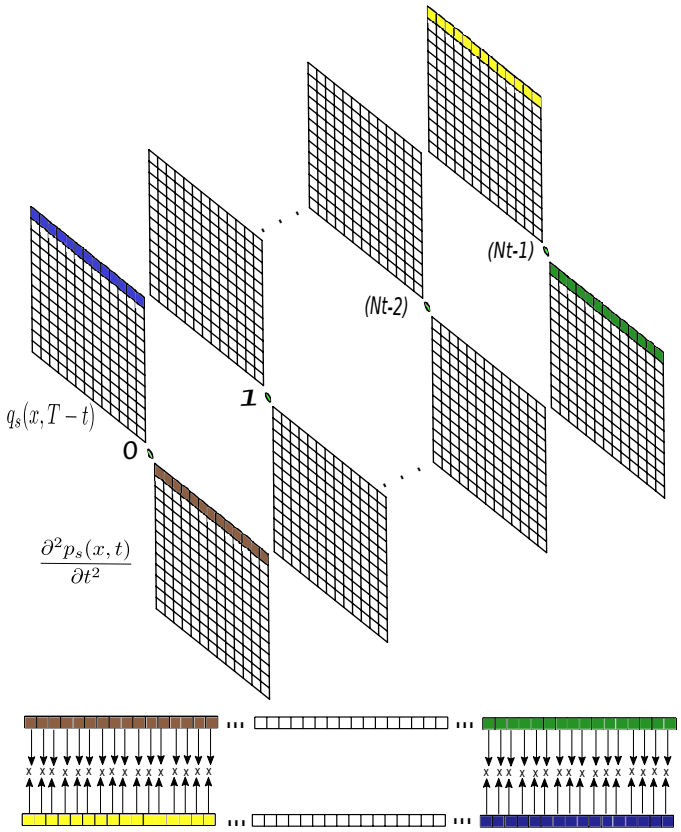



Tabla 19: Comparación de métricas Gradiente original y modificado.

| Metric | Gradiente original | | | Gradiente modificado | | |
|---|---|---|---|---|---|---|
| | Min | Max | Avg | Min | Max | Avg |
| Device memory read throughput | 133.417GB/s | 134.474GB/s | 133.883GB/s | 287.746MB/s | 14.628GB/s | 3.417GB/s |
| Device memory write throughput | 1.381GB/s | 1.498GB/s | 1.437GB/s | 6.141GB/s | 12.547GB/s | 9.806GB/s |

El ancho de banda utilizado por el Gradiente original era de 135.32 GB/s lo cual representa un poco más del 60 % de la capacidad máxima del ancho de banda de la memoria DRAM, el cambio realizado disminuyo considerablemente el anterior valor y no compromete el desempeño del propagador (tabla 20). El nuevo requerimiento de memoria DRAM es de 37.584 MB.

Tabla 20: Comparación de métricas Prop_psi_sh_less mas guardado de películas (Prop_psi_sh_less+p).

| Metric | Prop_psi_sh_less | | | Prop_psi_sh_less+p | | |
|---|---|---|---|---|---|---|
| | Min | Max | Avg | Min | Max | Avg |
| Device memory read throughput | 3.796GB/s | 34.193GB/s | 14.578GB/s | 1.103GB/s | 44.070GB/s | 11.414GB/s |
| Device memory write throughput | 6.232GB/s | 21.030GB/s | 14.692GB/s | 6.228GB/s | 22.730GB/s | 14.638GB/s |

Tabla 21: Prioridades fase 5

| Kernel | Importancia | Duración promedio | Invocaciones |
|---|---|---|---|
| Prop_psi_sh_less+p | 99.4 % | 10.376 $\mu$s | 12500 |
| Gradiente | 0.0 % | 4.217 $\mu$s | 10 |

**Variación de tamaño de bloque**

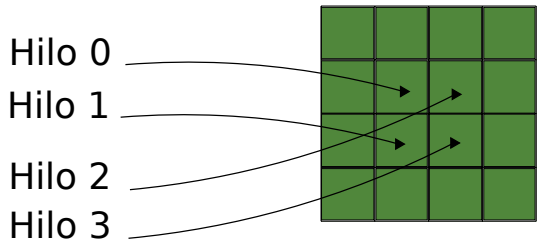
Se utilizaron diferentes tamaños de bloques para elegir el que diera el menor resultado de tiempo, el tamaño original 32x16 presento el mejor desempeño de todos.

### 4.2.7. Sexta fase: Uso de memoria de textura

El uso de espacios de memorias especiales en las condiciones adecuadas puede acelerar dramáticamente algunas aplicaciones. La memoria de textura al igual que la memoria de constantes es de tipo solo lectura y puede mejorar el desempeño del tráfico de lecturas cuando estas tienen ciertos patrones de acceso específico. Esta fue originalmente diseñada para aplicaciones gráficas pero también puede ser utilizada en algunas aplicaciones de computo[22].

Como la memoria de constantes, la memoria de textura está asignada a caché dentro del chip (*on-chip*), en algunas situaciones puede proveer una ancho de banda efectivo mayor, reduciendo las solicitudes de memoria por fuera del chip (*off-chip*), es decir, a DRAM. La asignación de caché de esta memoria está diseñado para aplicaciones gráficas donde los patrones de acceso a memoria exhiben una gran cantidad de localidad espacial.
Los hilos en la figura 26 acceden a cuatro espacios de memoria que no son consecutivos, así que estos no serian asignados a caché en una memoria global, la memoria de textura fue diseñada para acelerar este tipo de accesos.

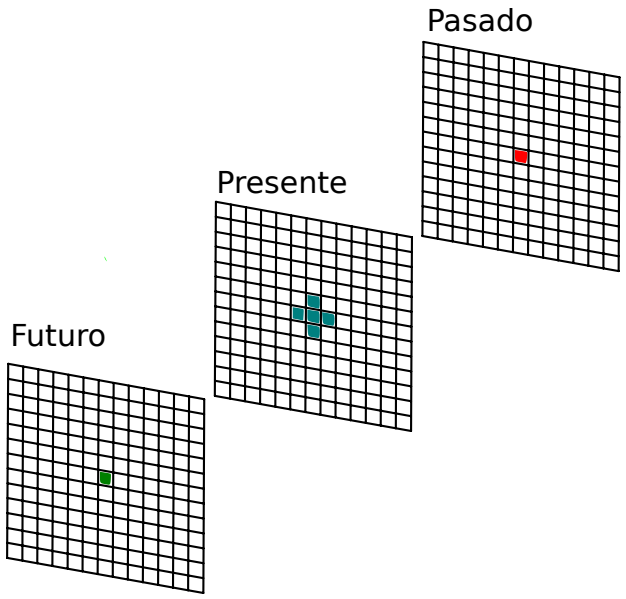
Figura 26: Mapeo de hilos en una región de dos dimensiones de la memoria.




Fuente: CUDA by example [22].

La propuesta en este caso, es utilizar la memoria de textura para realizar algunas de las lecturas que se utilizan para resolver la ecuación 2.7, específicamente las de la matriz presente la cual tiene una acceso de tipo stencil von Neumman de 5 puntos como se ve en la figura 26. Al utilizar memoria de

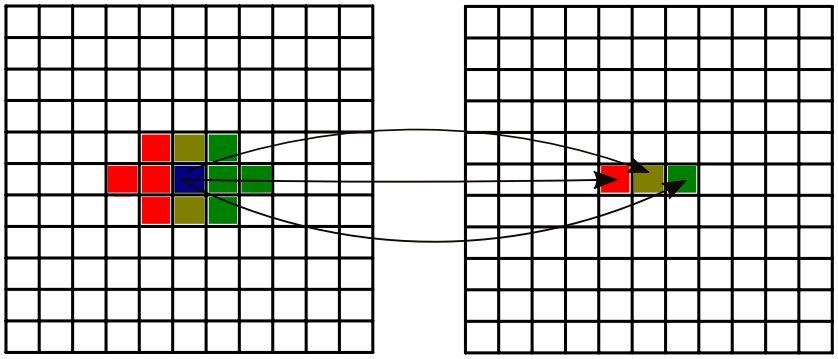
Figura 27: Accesos de hilos en los espacios de memoria para el cálculo de la propagación.




textura se obtienen dos ventajas a simple vista.

Los bancos de la memoria compartida tienen un tamaño de 4 bytes, los datos almacenados son de tipo flotante de precisión simple, cada banco almacena un dato. Al tener este patrón de accesos, tres hilos podrían solicitar acceso al mismo dato en memoria compartida y por consiguiente puede producirse conflictos de bancos, estos conflictos incrementarían si el *stencil* es de un orden mayor.

Figura 28: Conflicto producido por el acceso de tres hilo al mismo dato.



Como segunda ventaja el porcentaje de número de hilos por bloque que se utilizan para solucionar la propagación aumenta y además se ahorra el uso de la barrera de sincronización que se necesita al traer los datos de global a compartida para evitar condiciones de carrera y por consiguiente menos bloques se lanzan.

N $*$ M total de hilos por bloque.

(N $-$ 2) $*$ (M $-$ 2) hilos que se usan resolver el *stencil* en memoria compartida.

(N $-$ 1) $*$ (M $-$ 1) hilos que se usan para resolver el *stencil* en memoria de textura.

Luego el tamaño de la malla está dado por las ecuaciones 4.2 y 4.3.

$$usando\ memoria\ compartida = enterosuperior\left(\frac{N_x * N_z}{N-2}\right) * enterosuperior\left(\frac{N_x * N_z}{M-2}\right) \quad (4.2)$$

$$usando\ memoria\ de\ textura = enterosuperior\left(\frac{N_x * N_z}{N-1}\right) * enterosuperior\left(\frac{N_x * N_z}{M-1}\right) \quad (4.3)$$

#### 4.2.7.1. *Texture reference* API y *texture object* API

La memoria de texturas es leída por el *kernel* utilizando funciones de la GPU. El proceso de lectura de textura se denomina $fetch$, cada uno de estos procesos se especifican con un parámetro llamado *texture object* para el API *texture object* o *texture reference* para el API $texture\ reference$[16].

Listing 4.1: Definición estructura *texture object*. Fuente: CUDA C Programming GUIDE v7.5 [16]

```
1 struct cudaTextureDesc
2 {
3 enum cudaTextureAddressMode addressMode[3];
4 enum cudaTextureFilterMode filterMode;
5 enum cudaTextureReadMode readMode;
6 int sRGB;
7 int normalizedCoords;
8 unsigned int maxAnisotropy;
9 enum cudaTextureFilterMode mipmapFilterMode;
10 float mipmapLevelBias;
11 float minMipmapLevelClamp;
12 float maxMipmapLevelClamp;
13};
```

Listing 4.2: Definición estructura *texture reference*. Fuente: CUDA C Programming GUIDE v7.5 [16]

```
1 struct textureReference
2{
3 int normalized;
4 enum cudaTextureFilterMode filterMode;
5 enum cudaTextureAddressMode addressMode[3];
6 struct cudaChannelFormatDesc channelDesc;
7 int sRGB;
8 unsigned int maxAnisotropy;
9 enum cudaTextureFilterMode mipmapFilterMode;
10 float mipmapLevelBias;
11 float minMipmapLevelClamp;
12 float maxMipmapLevelClamp;
13}
```

*Texture object* solo es soportada para dispositivos con compute *capability* 3.x o mayor, es decir, arquitectura *Kepler* en adelante, y *texture reference* es soportado por todos los dispositivos.

**Diferencias**

*Texture reference* es creado en la compilación y las texturas son especificadas en tiempo de ejecución realizando un enlazado (*bind*) entre estas y el *Texture reference*. Puede ser un poco tedioso de usar pues es necesario realizar en enlace (*bind*) y desenlace (*unbind*) manualmente, además *texture reference* solo puede ser declarado como constante y no puede ser pasado como argumento a los *kernels*.

*Texture object* también llamados textura sin enlazado porque no requiere hacer un enlace y desenlace manual, este utiliza una nueva clase llamada `cudaTextureObject_t` que permite pasar como si se tratase de un puntero a los *Kernels* lo que permite que en tiempo de compilación no sea necesario saber que texturas serán usadas y en qué orden lo que permite una ejecución más dinámica (disminución del tiempo de lanzamiento entre *Kernels* usando este tipo de textura en comparación a *texture reference*) y una programación más sencilla [21].

**Implementación**

En base a lo anterior se realizan dos versiones (Prop_psi_sh_less+p) utilizando textura, mediante la API *texture reference* y otra mediante API *texture object* (solo para matriz presente). Como criterio de selección se toma el tiempo de ejecución total con el mejor tamaño de bloque.

Tabla 22: Comparación APIs de memoria de textura utilizando tamaño de bloque 32x8.

| **API *texture*** | **Achieved occupancy Avg** | **Utilización de la GPU** | **Tiempo total** | **Tiempo promedio kernel** |
|---|---|---|---|---|
| Reference | 0.462 | 62.7 % | 20.057 s | 9.701 $\mu$ s |
| Object | 0.458 | 81.2 % | 20.180 s | 9.820 $\mu$ s |

De la tabla 26:

- El tamaño óptimo de bloque para las dos implementaciones es 32x8
- *Texture reference* presenta un mejor desempeño que *texture object*.
- Del *compute utilization* se puede inferir que *texture object* permite una ejecución más dinámica, es decir lanzamientos más seguidos. Se comprobó la hipótesis mediante la línea de tiempo (ver figuras 32 y 30).
- Se referirá como Prop_tex_ref a esta nueva modificación realizada.

Figura 29: Intervalo de tiempo entre lanzamientos del *kernel* con API *texture reference*.

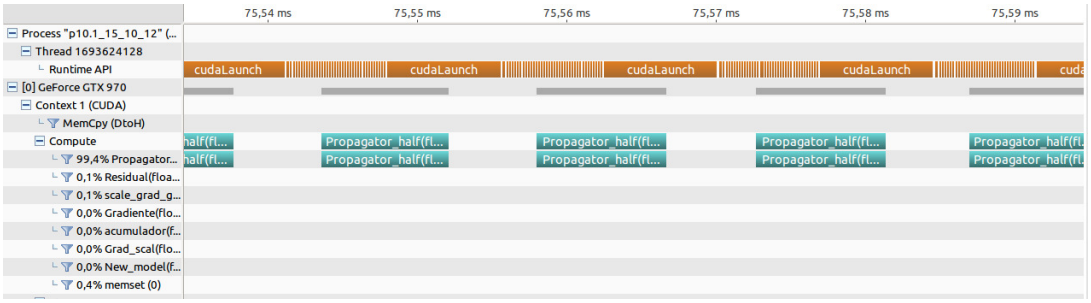
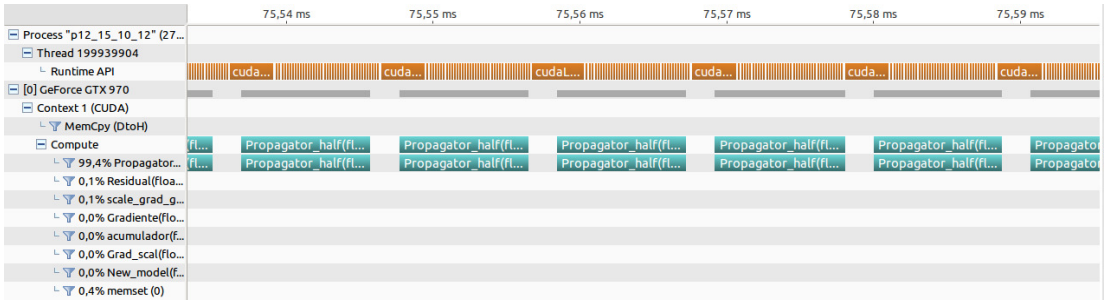



Figura 30: Intervalo de tiempo entre lanzamientos del *kernel* con API *texture object*.




#### 4.2.7.2. Comparación entre implementaciones con memoria compartida y uso de memoria de textura mediante API *texture reference*

Las implementaciones que presentan el uso de memoria de textura tienen un mejor desempeño en comparación con las que usan únicamente memoria compartida, posiblemente por dos razones:

- Mayor porcentaje de hilos por bloque solucionando la propagación.
- Se evitan la mayoría de conflictos de banco al utilizar memoria de textura.

Tabla 23: Aceleración sexta fase.

| Versión FWI | Utilización de la GPU | Memoria DRAM | Tiempo total | Aceleración |
|---|---|---|---|---|
| Original | 55.5 % | 71.631MB | 49.996 s | |
| Primera fase | 52.5 % | 71.631MB | 46.422 s | 1.076x |
| Segunda fase | 43.5 % | 71.631MB | 36.866 s | 1.356x |
| Tercera fase | 75.5 % | 71.631MB | 23.328 s | 2.143x |
| Cuarta fase | 75.2 % | 71.631MB | 23.233 s | 2.151x |
| FWI cuarta fase reorganizada | 82.5 % | 71.631MB | 21.198 s | 2.358x |
| Quinta fase | 82.5 % | 37.584MB | 21.200 s | 2.358x |
| Sexta fase(Texture object) | 81.2 % | 37.584MB | 20.180 s | 2.477x |
| Sexta fase(Texture reference) | 62.7 % | 37.584MB | 20.057 s | 2.492x |

Tabla 24: Lista de prioridades implementación *texture reference*.

| **Kernel** | **Prioridad** |
|---|---|
| Prop_tex_ref | 99.4 % |
| Residual | 0.1 % |
| Scale_grad_gpu | 0.1 % |
| Gradiente | 0.0 % |
| New_model | 0.0 % |
| Memset | 0.4 % |

## 4.3. ANÁLISIS DE *KERNELS* DE BAJA PRIORIDAD Y OTROS

Las funciones de GPU mostradas en la tabla 24 excluyendo *Prop_tex_ref* representan menos del 1 % del tiempo total de ejecución en la GPU, luego mejoras en estos *kernels* representan un impacto mínimo en el desempeño del algoritmo, tomando como referencia la implementación con memoria compartida y las de memoria de textura, además como se mencionó en el análisis previo la mayoría de estos *kernels* realizan funciones simples y presentan accesos a memoria *coalesced*, el acceso coalesced o unificado se puede comprobar mediante la eficiencia de carga y escritura a memoria global (*global memory load efficiency y global memory write efficiency*).

Tabla 25: *Global memory load efficiency y global memory load efficiency* de *kernels* de las implementaciones con memoria compartida y de textura.

| **Kernel** | **Global memory load efficiency** | | | **Global memory write efficiency** | | |
|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Avg** | **Min** | **Max** | **Avg** |
| Residual | 82.530 % | 82.543 % | 82.535 % | 82.543 % | 82.543 % | 82.543 % |
| Scale_grad_gpu | 87.500 % | 100.000 % | 93.750 % | 12.500 % | 12.500 % | 12.500 % |
| Acumulador | 83.333 % | 83.333 % | 83.333 % | 83.333 % | 83.333 % | 83.333 % |
| Grad_scale | 70.454 % | 70.454 % | 70.454 % | 83.333 % | 83.333 % | 83.333 % |
| New_model | 83.144 % | 83.144 % | 83.144 % | 83.144 % | 83.144 % | 83.144 % |

De la tabla 25 todos los *kernels* presentan valores altos de eficiencia excepto la escritura en el *scale_grad_gpu* esto es de esperar puesto que esta función realiza una única escritura por bloque de un dato tipo flotante de precisión simple (4 bytes) y los accesos que se realizan deben ser de 32 bytes.

$$global\ memory\ write\ efficiency = \frac{transactions\_per\_seconds\_request}{transactions\_per\_seconds\_required} * 100 \tag{4.4}$$

Entonces según ecuación 4.4 la eficiencia de escritura es (4/32)*100 = 12.5 % .

### 4.3.1. Memoria de constantes

Existe un total de 64 KB de memoria de constantes en el *device* el espacio de esta memoria tiene asignado caché L1, lo que significa que el costo de acceder a ella es la lectura de memoria de *device* solo cuando no se presenta un acierto en caché. Su característica principal es que esta optimizada para

cuando todos los hilos del mismo *warp* acceden a la misma dirección [15]. Los parámetros de la tabla 4 se llevaron a este espacio de memoria pero no se presentó una mejora en el desempeño luego se descarta el uso de esta.

## 4.4. USO DE BIBLIOTECA RÁPIDA DE MATEMÁTICAS

Es recomendable el uso de la librería de matemática rápida para disminuir el tiempo de ejecución de los *kernels* siempre y cuando no afecte el modelo final del algoritmo. La GPU soporta algunas funciones de la librería de matemáticas estándar de C, C++ [16], la librería rápida solo afecta los segmentos del código ejecutados en la tarjeta gráfica (funciones intrínsecas), generalmente estas funciones toman menos tiempo en ejecutarse que su contra-parte estándar sin embargo, tienen una precisión menor.

Tabla 26: Funciones afectada al usar `--use_fast_math`.

| **Operador/función** | **función en GPU** |
|---|---|
| x/y | __fdividef(x,y) |
| sinf(x) | __sinf(x) |
| cosf(x) | __cosf(x) |
| tanf(x) | __tanf(x) |
| sincosf(x,sptr,cptr) | __sincosf(x,sptr,cptr) |
| logf(x) | __logf(x) |
| log2f(x) | __log2f(x) |
| log10f(x) | __log10f(x) |
| expf(x) | __expf(x) |
| exp10f(x) | __exp10f(x) |
| powf(x,y) | __powf(x,y) |

Tomada de [16]

Una forma de utilizar estas funciones intrínsecas es mediante la bandera `--use_fast_math` al momento de la compilación. Las funciones afectadas en la ejecución de la FWI son `x/y`, `expf(x)`, `powf(x,y)` y `logf(x)`

En la figura 31 a simple vista no es perceptible el cambio debido al uso de matemática rápida, la curva de la función objetivo permite visualizar un poco mejor la diferencia entre los dos modelos finales, y calculando la norma L2 de los dos modelos finales (tabla 27), se llegan a valores del mismo orden con una diferencia del 0.022 % por lo que se puede aceptar el uso de matemática rápida.

Tabla 27: Norma L2 de los modelos finales.

| **Versión FWI** | $l^2$norm |
|---|---|
| original | $1{,}7751 * 10^4$ |
| texture+fast math | $1{,}7746 * 10^4$ |

Figura 31: Modelo Marmousi . a) Modelo Marmousi original. b) Modelo Marmousi inicial. c) Modelo Marmousi FWI original. d) Modelo Marmousi usando memoria de textura y matemática rápida.

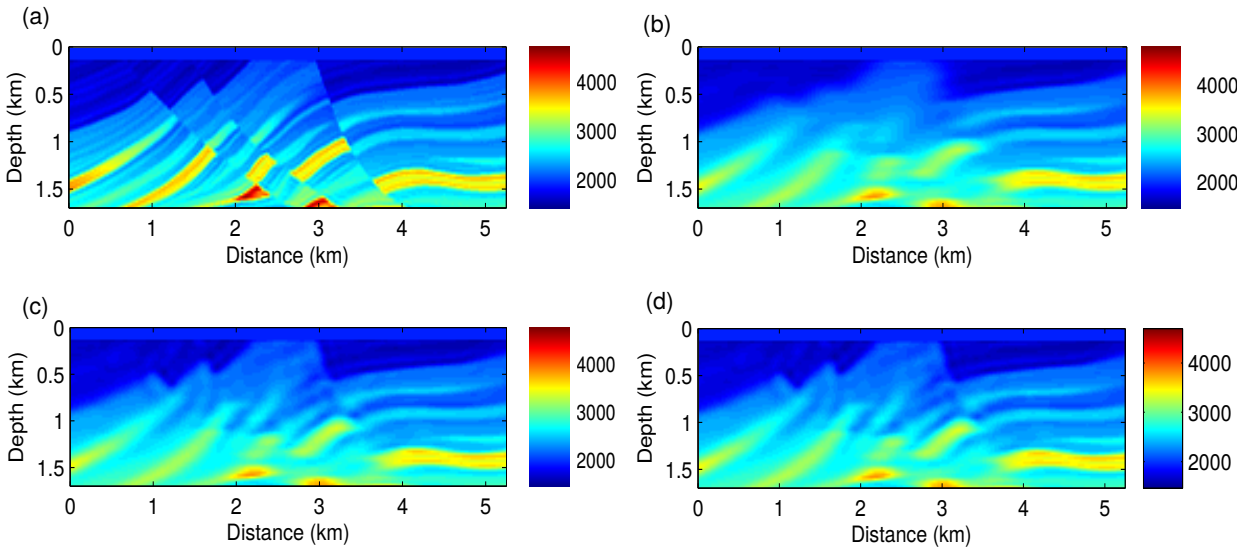



Figura 32: Curvas función objetivo, color rojo FWI original y color azul FWI usando memoria de textura y matemáticas rápida.

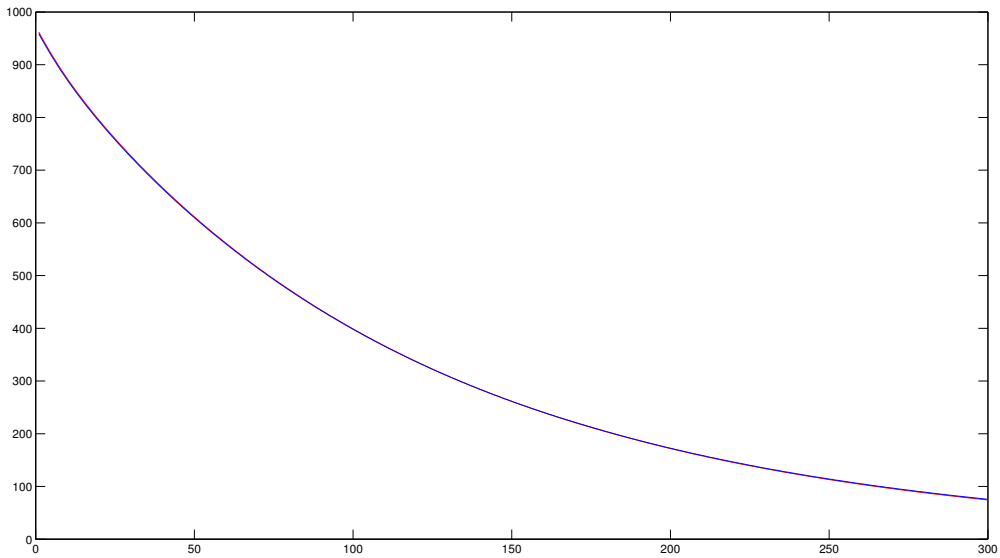


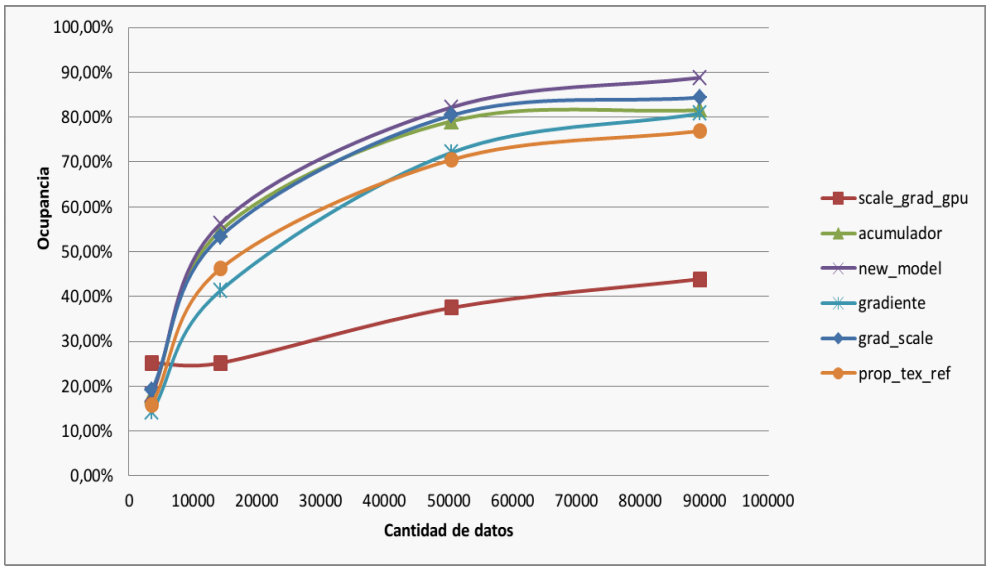
Tabla 28: Aceleración alcanzada.

| Versión FWI | Utilización de la GPU | Memoria DRAM | Tiempo total | Aceleración |
|---|---|---|---|---|
| Original | 55.5 % | 71.631MB | 49.996 s | |
| Primera fase | 52.5 % | 71.631MB | 46.422 s | 1.076x |
| Segunda fase | 43.5 % | 71.631MB | 36.866 s | 1.356x |
| Tercera fase | 75.5 % | 71.631MB | 23.328 s | 2.143x |
| Cuarta fase | 75.2 % | 71.631MB | 23.233 s | 2.151x |
| FWI cuarta fase reorganizada | 82.5 % | 71.631MB | 21.198 s | 2.358x |
| Quinta fase | 82.5 % | 37.584MB | 21.200 s | 2.358x |
| Sexta fase(Texture object) | 81.2 % | 37.584MB | 20.180 s | 2.477x |
| Sexta fase(Texture reference) | 62.7 % | 37.584MB | 20.057 s | 2.492x |
| Texture reference fast math | 44.5 % | 37.584MB | 15.150 s | 3.300x |

## 4.5. VARIACIÓN DEL TAMAÑO DEL MODELO

Una hipótesis acerca del bajo valor de ocupación es que esta se debe a que el volumen de datos a procesar es bajo y por ello no se aprovechan del todo los recursos de la GPU, la prueba para validar esta hipótesis es trabajar con modelos con más puntos, en la figura 33 se puede ver que para la mayoría de *kernels* la ocupación incrementa proporcionalmente al incrementar el número de datos hasta llegar a un punto de saturación donde la limitante tiene como causas diversas razones como divergencia de hilos, dependencia de datos, entre otros.

Figura 33: Variación de la ocupación al incrementar la cantidad de datos.



Capítulo 5

# CONCLUSIONES

- La implementación con mejor desempeño es aquella con el uso de la API *texture reference* (sexta fase), con la que se logra un *speedup* de 2.492x y al utilizar matemática rápida un valor máximo de 3.3x, un requerimiento de memoria de 37.584 MB, 1.9 veces menos que la versión original, sin presentar limitaciones de ocupación.
- Es de importancia mantener el número de registros utilizados por hilo en un valor que no limite la ocupación teórica, puesto que incurrir en esto es impedir la total utilización de los recursos de la GPU. En modelos más grandes o en *kernels* que tengan duraciones mucho más largas que las presentadas en este trabajo tener una ocupación limitada puede comprometer considerablemente el desempeño de la aplicación, una opción es el uso de banderas de compilación que limiten el uso de registros.
- La cantidad de datos del modelo utilizado no permite el uso de la totalidad de los recursos disponibles de la GPU, esto es reflejado en los valores de ocupación. La tarjeta tiene la capacidad de ejecutar modelos mucho más grandes siempre y cuando estos no sobrepasen las especificaciones de memoria DRAM (4 GB para GTX 970), en tarjetas de computo científico como la Tesla K40 cuenta con más capacidad de DRAM (12 GB) y una cantidad de SMs (15 SMs) mayor, lo que permite un volumen de datos mucho mayor.
- Escoger un método adecuado para la implementación de una función en la GPU es igual de importante que tener en cuenta las características de la arquitectura de esta, existen algoritmos que a simple vista no presentan un alto grado de paralelización pero realizando una análisis más profundo se pueden obtener propuestas de solución que se ajusten a las características del *Hardware* usado, un claro ejemplo es la implementación del *scale_grad_gpu* en este trabajo.

Capítulo 6

# RECOMENDACIONES

- Los estudios geofísicos reales pueden requerir recursos de memoria que sobrepasen la capacidad de la GPU, para estos casos se requiere realizar implementaciones que tengan en cuenta transferencias continuas de datos entre *host* y *device*, CUDA posee funciones y otros recursos como el uso de *stream*, copias asíncronas, paralelismo dinámico entre otras que permiten abordar este tipo de problemas.
- Mayor paralelismo no siempre se traduce en mejor desempeño por que se pueden producir cuellos de botella por ancho de banda, para implementaciones con derivadas y *stencil* de mas puntos que se compruebe que tienen limitación por ancho de banda, una posible solución seria resolver más de un punto por hilo lo que conlleva más trabajo por hilo y una presión menor sobre recursos de memoria.
- Se recomienda el uso de memorias especiales (texturas, constantes, compartida, etc.) cuando el problema presente características que ameriten su uso.
- Se recomienda el uso de la libreria de matemática rápida siempre que el aumento de error producido por esta no sea un factor que impida llegar a la solucion esperada.

# REFERENCIAS


[1] ABREO, David L. Implementación de una inversión de onda completa (FWI) 2-D sobre una arquitectura GPU. Trabajo de grado Ingeniero Electrónico. Bucaramanga: Universidad Industrial de Santander. Facultad de Ingenierías Fisicomecánicas. Escuela de Ingenierías Eléctrica, Electrónico y de Telecomunicaciones, 2015. p. 32-33.

[2] ABREO, Sergio A; RAMIREZ, Ana; REYES, Oscar M y GONZÁLES, Herling. Using GPUs to speed up the 2D full Waveform Inversion: A practical point of view. *Journal of oil, gas and alternative energy sources*. 2015, Vol 4, p. 15-18.

[3] PLESSIX, Rene E. A review of the adjointstate method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International*,2006, nro. 167. p. 495–503.

[4] TARANTOLA, Albert; VALETTE, Bernard. Generalized non-linear inverse problems solved using the least-squares criterion. *Reviews of Geophysics and Space Physics*. 1992, Nro. 2. p. 219-232.

[5] MA, Yong. Full waveform inversion with image-guided gradient. Tesis de Maestría. Golden, Colorado: Colorado School of Mines. Center for Wave Phenomena, 2010. p. 4-13.

[6] SNIEDER, Roel. The role of nonlinearity in inverse problems. Inverse Problems. Netherlands: Ultrecht University. Department of Geophysics, 1998. p. 387-404.

[7] ORGANISMO SUPERVISOR DE LA INVERSIÓN EN ENERGÍA Y MINERÍA [sitio web]. Disponible en:
http://www.osinerg.gob.pe/newweb/pages/GFH/1652.htm

[8] OSPINA, Saúl B. Propagación de ondas sísmicas y migración. Tesis de Maestría. Bogotá D.C.: Universidad Nacional de Colombia. Faculta de Ciencias. Departamento de Matemáticas, 2011. p. 33-38.

[9] PASALIC, Damir; MCGARRY, Ray. Convolutional perfectly matched layer for isotropic and anisotropic acoustic wave equations. *2010 SEG Annual Meeting. Society of Exploration Geophysicists*, 2010.

[10] RYAN, Harold. Ricker, Ormsby, Klauder, Butterworth-A Choice of Wavelets [en línea]. Disponible en:
http://74.3.176.63/publications/recorder/1994/09sep/sep94-choice-of-wavelets.pdf

[11] LEDESMA, Antonio y BERNAL, Omar. Introducción al Metodo de Diferencias Finitas y su Implementación Computacional. México: Universidad Nacional de México. Facultad de Ciencias. 2015. p. 2-3.

[12] HOEGER, Herbert. Introducción a la Computación Paralela. Venezuela: Centro Nacional de Cálculo Científico Universidad de Los Andes, 1997. p. 32-33.

[13] JIMÉNEZ, Daniel; GUIM, Francesc y RODERO, Ivan. Arquitecturas basadas en computación gráfica (GPU). Barcelona: Eureca Media, SL, 2012. 19 p.

[14] LÁSZLÓ, Endre; SZOLGAY, Péter y NAGY, Zoltán. Analysis of GPU based CNN implementation. *13th Internatioal Workshop on Cellular Nanoscale Networks and Their Application (CNNA)*. 2012, p. 1-5.

[15] NVIDIA Corporation. CUDA C Best Practices GUIDE v7.5 [en línea]. Disponible en:
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[16] NVIDIA Corporation. CUDA C Programming GUIDE v7.5 [en línea]. Disponible en:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[17] NVIDIA Corporation. GPU Computing: The Revolution [en línea]. Disponible en:
http://www.nvidia.com/object/cuda_home_new.html.

[18] NVIDIA Corporation. Profiler User's Guide [en línea]. Disponible en:
http://docs.nvidia.com/cuda/profiler-users-guide/.

[19] NVIDIA Corporation. Whitepaper NVIDIA GeForce GTX 980 [en línea]. Disponible en:
http://international.download.nvidia.com/geforce-com/international/pdfs/
GeForce_GTX_980_Whitepaper_FINAL.PDF

[20] REIMÚNDEZ, Carlos J. Estudio de rendimiento en GPU. Tesis de Maestría. España: Universidad Complutense de Madrid. Facultad de Informática, 2010. 85 p.

[21] CLARK, M. CUDA Pro Tip: Kepler Texture Objects Improve Performance and Flexibility [en línea]. Disponible en:
https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility

[22] SANDERS, Jason y KANDROT,Edward, CUDA by example [en línea]. Disponible en:
http://www.nvidia.com. p. XIII-XIV, 115-137.

[23] MORGAN, Timothy. NVIDIA BRINGS MAXWELL GPUs TO TESLA COPROCESSORS. Disponible en:
http://www.nextplatform.com/2015/11/10/nvidia-brings-maxwell-gpus-to-tesla-coprocessors/

[24] MUJTABA, Hassan. NVIDIA Unleashes Maxwell GM204 Based GeForce GTX 980 and GeForce GTX 970 Graphics Cards. Disponible en:
http://wccftech.com/nvidia-unleashes-maxwell-gm204-geforce-gtx-980-gtx-970-graphics-cards/

# BIBLIOGRAFÍA

ABREO, David L. Implementación de una inversión de onda completa (FWI) 2-D sobre una arquitectura GPU. Trabajo de grado Ingeniero Electrónico. Bucaramanga: Universidad Industrial de Santander. Facultad de Ingenierías Fisicomecánicas. Escuela de Ingenierías Eléctrica, Electrónico y de Telecomunicaciones, 2015. p. 32-33.

ABREO, Sergio A; RAMIREZ, Ana; REYES, Oscar M y GONZÁLES, Herling. Using GPUs to speed up the 2D full Waveform Inversion: A practical point of view. *Journal of oil, gas and alternative energy sources*. 2015, Vol 4, p. 15-18.

CLARK, M. CUDA Pro Tip: Kepler Texture Objects Improve Performance and Flexibility [en línea]. Disponible en:
https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility.

HOEGER, Herbert. Introducción a la Computación Paralela. Venezuela: Centro Nacional de Cálculo Científico Universidad de Los Andes, 1997. p. 32-33.

JIMÉNEZ, Daniel; GUIM, Francesc y RODERO, Ivan. Arquitecturas basadas en computación gráfica (GPU). Barcelona: Eureca Media, SL, 2012. 19 p.

LÁSZLÓ, Endre; SZOLGAY, Péter y NAGY, Zoltán. Analysis of GPU based CNN implementation. *13th Internatioal Workshop on Cellular Nanoscale Networks and Their Application (CNNA)*. 2012, p. 1-5.

LEDESMA, Antonio y BERNAL, Omar. Introducción al Metodo de Diferencias Finitas y su Implementación Computacional. México: Universidad Nacional de México. Facultad de Ciencias. 2015. p. 2-3.

MA, Yong. Full waveform inversion with image-guided gradient. Tesis de Maestría. Golden, Colorado: Colorado School of Mines. Center for Wave Phenomena, 2010. p. 4-13.

MORGAN, Timothy. NVIDIA BRINGS MAXWELL GPUs TO TESLA COPROCESSORS. Disponible en:
http://www.nextplatform.com/2015/11/10/nvidia-brings-maxwell-gpus-to-tesla-coprocessors/

MUJTABA, Hassan. NVIDIA Unleashes Maxwell GM204 Based GeForce GTX 980 and GeForce GTX 970 Graphics Cards. Disponible en:
http://wccftech.com/nvidia-unleashes-maxwell-gm204-geforce-gtx-980-gtx-970-graphics-cards/

NVIDIA Corporation. CUDA C Best Practices GUIDE v7.5 [en línea]. Disponible en:
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

NVIDIA Corporation. CUDA C Programming GUIDE v7.5 [en línea]. Disponible en:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

NVIDIA Corporation. GPU Computing: The Revolution [en línea]. Disponible en:
http://www.nvidia.com/object/cuda_home_new.html.

NVIDIA Corporation. Profiler User's Guide [en línea]. Disponible en:
http://docs.nvidia.com/cuda/profiler-users-guide/.

NVIDIA Corporation. Whitepaper NVIDIA GeForce GTX 980 [en línea]. Disponible en:
http://international.download.nvidia.com/geforce-com/international/pdfs/
GeForce_GTX_980_Whitepaper_FINAL.PDF

ORGANISMO SUPERVISOR DE LA INVERSIÓN EN ENERGÍA Y MINERÍA [sitio web]. Disponible en:
http://www.osinerg.gob.pe/newweb/pages/GFH/1652.htm

OSPINA, Saúl B. Propagación de ondas sísmicas y migración. Tesis de Maestría. Bogotá D.C.: Universidad Nacional de Colombia. Faculta de Ciencias. Departamento de Matemáticas, 2011. p. 33-38.

PASALIC, Damir; MCGARRY, Ray. Convolutional perfectly matched layer for isotropic and anisotropic acoustic wave equations. *2010 SEG Annual Meeting. Society of Exploration Geophysicists*, 2010.

PLESSIX, Rene E. A review of the adjointstate method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International*,2006, nro. 167. p. 495–503.

REIMÚNDEZ, Carlos J. Estudio de rendimiento en GPU. Tesis de Maestría. España: Universidad Complutense de Madrid. Facultad de Informática, 2010. 85 p.

RYAN, Harold. Ricker, Ormsby, Klauder, Butterworth-A Choice of Wavelets [en línea]. Disponible en: http://74.3.176.63/publications/recorder/1994/09sep/sep94-choice-of-wavelets.pdf

SANDERS, Jason y KANDROT,Edward, CUDA by example [en línea]. Disponible en: http://www.nvidia.com. p. XIII-XIV, 115-137.

SNIEDER, Roel. The role of nonlinearity in inverse problems. Inverse Problems. Netherlands: Ultrecht University. Department of Geophysics, 1998. p. 387-404.

TARANTOLA, Albert; VALETTE, Bernard. Generalized non-linear inverse problems solved using the least-squares criterion. *Reviews of Geophysics and Space Physics*. 1992, Nro. 2. p. 219-232.