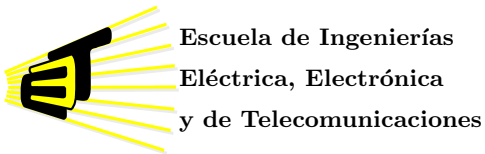
PREDICCIÓN DEL TIEMPO DE EJECUCIÓN DE UNA IMPLEMENTACIÓN IN-PLANE DE LA ECUACIÓN DE ONDA ACÚSTICA 3D EN UNA GPU USANDO UN MODELO ANALÍTICO

Xiomara Ribero Figueroa



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

PREDICCIÓN DEL TIEMPO DE EJECUCIÓN DE UNA IMPLEMENTACIÓN IN-PLANE DE LA ECUACIÓN DE ONDA ACÚSTICA 3D EN UNA GPU USANDO UN MODELO ANALÍTICO

Xiomara Ribero Figueroa

Trabajo de investigación presentado como requerimiento parcial para optar por el título de Ingeniera Electrónica

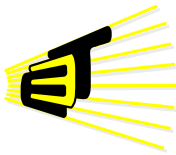
Director:

MIE. Dorfell Leonardo Parra Prada

Co-Directores:

PhD(c). William Alexander Salamanca Becerra

PhD. Ana Beatriz Ramirez Silva



Escuela de Ingenierías
Eléctrica, Electrónica
y de Telecomunicaciones

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

2016

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

Pag.

ÍNDICE DE TABLAS

Pag.

ÍNDICE DE ANEXOS

Pag.

RESUMEN

TÍTULO:

Predicción del tiempo de ejecución de una implementación In-plane de la ecuación de onda acústica 3D en una GPU usando un modelo analítico*.

AUTOR:

XIOMARA RIBERO FIGUEROA**

PALABRAS CLAVES:

GPU, Modelo, Tiempo de Ejecución, Stencils, In-plane, CUDA.

DESCRIPCIÓN

El siguiente trabajo de investigación presenta una propuesta enfocada en predecir el tiempo de ejecución por medio del modelo propuesto por Tang et al [2013], contrastándolo con el tiempo obtenido por medio de la implementación del método In-plane en una GPU. Inicialmente se estudian las diferentes formas de carga In-plane propuestas por Tang et al [2013] en el artículo [9]. Luego se implementó la solución de la ecuación de onda acústica con estas diferentes formas de carga (forma de carga clásica, forma de carga vertical, forma de carga horizontal y forma de carga completa). Después se comparó el tiempo de ejecución de cada una de las implementaciones, posteriormente se analizaron los diferentes parámetros y variables del modelo propuesto en [9], para al final estimar el tiempo de ejecución usando las ecuaciones descritas en el modelo. Finalmente se compara este tiempo con el obtenido de la implementación del método In-plane carga completa.

Se obtuvo que el modelo no es aplicable para la implementación de la ecuación de onda a través del método In-plane carga completa pues no contempla el efecto que genera la forma como se almacenan y se acceden a los datos en memoria compartida y el artículo no presenta muchos detalles de la implementación.

*Trabajo de grado.

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

ABSTRACT

TITLE:

Prediction of the execution time of an In-plane 3D acoustic wave equation implementation on a GPU using an analytical model*.

AUTHOR:

XIOMARA RIBERO FIGUEROA**

KEYWORDS:

GPU, Model, Execution time, Stencils, In-plane, CUDA.

DESCRIPTION

This research work presents the prediction of an In-plane implementation runtime by means of the analytical model proposed by Tang et al [2013] and compares it with the implementation time. Initially the different load forms of the In-plane method proposed by Tang et al [2013] in [9] are studied.Then, the In-plane method has four forms of load (standard load form, vertical load form, horizontal load form and full load form), these forms of load are implementated for the acoustic wave equation solution in a Nvidia K40 GPU. Following,the execution times of each of the implementations are taken and the these times are compared. Subsequently, the different parameters and variables of the analytical model proposed are analyzed and computed to obtain the estimated execution time by using the equations described in the model. Finally, the estimated times by the model are compared with the execution times taken from the full-load implementation of the in-plane method.

It was obtained that the model is not applicable for the implementation of the wave equation through the full-load implementation of the in-plane method. because it does not contemplate the effect that generates the way in which the data is stored and accessed in shared memory and The article does not present many details of the implementation.

*Degree work

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

INTRODUCCIÓN

La computación de alto rendimiento es usada para simular modelos complejos como la dinámica de fluidos, difusión de calor, procesamiento de datos sísmicos, etc. Los avances tecnológicos en este campo tienen la tendencia a aumentar la capacidad de cómputo y disminuir el consumo de potencia. También existen diferentes métodos que buscan disminuir el tiempo de acceso a memoria lo que lleva a un menor consumo de potencia.

Estos métodos se denominan métodos de segmentación de dominio y son estrategias de diferentes formas en las cuales podemos distribuir la manera de como se va a procesar un volumen de datos. Entre las estrategias de implementación se encuentran las estrategias 3-D, 2.5-D e In-plane. La estrategia 3-D es la estrategia tradicional donde cada sub-bloque es procesado por un bloque de *threads*. La estrategia 2.5-D tiene como principio que las cargas redundantes en la dirección Z puedan reducirse ya que todo el bloque de datos no necesita ser cargado al mismo tiempo durante el cálculo de los elementos dentro del bloque. La estrategia In-plane busca mejorar la eficiencia de la memoria usando el paralelismo a nivel de memoria para reducir el número de instrucciones de carga [9]. En [9] propone un modelo que permite predecir los tiempos que se obtendrían al usar una implementación basada en el método In-plane.

El objetivo de esta tesis es estimar la viabilidad de hallar el tiempo de ejecución del cálculo del *stencil* 3D de un laplaciano de la onda acústica en una GPU mediante lectura de datos In-plane. Para realizar este objetivo, primero se realizara el estudio de las diferentes formas de carga del método In-plane. Después, cada una de las formas de carga serán implementadas. Finalmente se usará el modelo analítico propuesto por Tang et al [2013] [9] y se compararán los tiempos estimados por el modelo con los de la implementación In-plane forma de carga completa.

Capítulo 1

MARCO TEÓRICO

El uso de sistemas de cómputo para la realización de simulaciones involucra conceptos de programación, modelado físico, etc. Algunos de los conceptos más relevantes empleados para el desarrollo de este trabajo se presentan a continuación.

1.1. ECUACIÓN DE ONDA ACÚSTICA CON DENSIDAD CONSTANTE

Algunos fenómenos físicos pueden ser simulados usando ecuaciones. Por ejemplo, la propagación de una onda acústica en un medio de densidad constante puede ser representado por la ecuación 1.1 en el caso de una dimensión.

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

donde $p(x,t)$ representa la posición del objeto en la dirección x en el tiempo t y v es la velocidad a la cual se mueve. Si el medio donde la onda se propaga es tridimensional podemos usar la ecuación de onda acústica con densidad constante que se encuentra descrita en la ecuación 1.2 donde $p(x,y,z,t)$ representa la intensidad de presión de la onda en el espacio y el tiempo, v representa la velocidad de propagación de la onda y esta depende del medio en el que se propaga la onda.

$$\frac{1}{v^2}\frac{\partial^2 p(x,y,z,t)}{\partial t^2} = \nabla^2 p(x,y,z,t) = \frac{\partial^2 p(x,y,z,t)}{\partial x^2} + \frac{\partial^2 p(x,y,z,t)}{\partial y^2} + \frac{\partial^2 p(x,y,z,t)}{\partial z^2}. \tag{1.2}$$

Donde ∇^2 es conocido como el laplaciano.

1.2. DISCRETIZACIÓN DE LA ECUACIÓN DE ONDA ACÚSTICA USANDO FDTD

El método de diferencia finitas en el dominio del tiempo (*Finite Difference Time-Domain* o en sus siglas FDTD) es uno de los más usados para discretizar la ecuación de la onda. Dicho método transforma la ecuación diferencial en una ecuación de diferencias finitas o un sistema de ecuaciones en diferencias finitas. Para ello toma las derivadas y las aproxima por medio de la serie de Taylor, mientras más términos de la serie se tomen, la aproximación será más precisa.
Para discretizar la ecuación de onda es necesario conocer los tipos de esquemas de FDTD que existen:

1. **Esquema de diferencias finitas hacia atrás:** Se le llama de esta forma ya que usa los datos $(x - \Delta x)$ y (x) para estimar la derivada. Su representación se presenta en la ecuación 1.3.

$$\frac{\partial^2 f(x,t)}{\partial x^2} \cong \frac{C_{-\Delta x} f(x - \Delta x, t) + C_0 f(x,t)}{\Delta x^2} \tag{1.3}$$

   En la ecuación 1.4 se muestra un ejemplo de las diferencias finitas hacia atrás de orden 4.

$$\frac{\partial^2 f(x,t)}{\partial x^2} \cong \frac{C_{-4} f(x-4,t) + C_{-3} f(x-3,t) + C_{-2} f(x-2,t) + C_{-1} f(x-1,t) + C_0 f(x,t)}{\Delta x^2} \tag{1.4}$$

2. **Esquema de diferencias finitas hacia delante:** Se le llama de esta forma ya que usa los datos (x) y $(x + \Delta x)$ para estimar la derivada. Su representación se presenta en la ecuación 1.5.

$$\frac{\partial^2 f(x,t)}{\partial x^2} \cong \frac{C_0 f(x,t) + C_{\Delta x} f(x + \Delta x, t)}{\Delta x^2} \tag{1.5}$$

   En la ecuación 1.6 se muestra un ejemplo de las diferencias finitas hacia adelante de orden 4.

$$\frac{\partial^2 f(x,t)}{\partial x^2} \cong \frac{C_0 f(x,t) + C_1 f(x+1,t) + C_2 f(x+2,t) + C_3 f(x+3,t) + C_4 f(x+4,t)}{\Delta x^2} \tag{1.6}$$

3. **Esquema de diferencias finitas centradas:** Es la media de las diferencias hacia adelante y hacia atrás. Usa los datos $(x - \Delta x)$, (x) y $(x + \Delta x)$ para estimar la derivada y viene dada por la siguiente ecuación.

$$\frac{\partial^2 f(x,t)}{\partial x^2} \cong \frac{C_{-\Delta x} f(x - \Delta x, t) + C_0 f(x,t) + C_{\Delta x} f(x + \Delta x, t)}{\Delta x^2} \tag{1.7}$$

   En la ecuación 1.8 se muestra un ejemplo de las diferencias finitas centradas de orden 4.

$$\frac{\partial^2 f(x,t)}{\partial x^2} \cong \frac{C_{-2} f(x-2,t) + C_{-1} f(x-1,t) + C_0 f(x,t) + C_1 f(x+1,t) + C_2 f(x+2,t)}{\Delta x^2} \tag{1.8}$$

El esquema seleccionado para las implementaciones es el de diferencias finitas centradas.
La ecuación de la onda 3D discretizada para un orden 2 en el espacio y un orden 2 en el tiempo, se muestra en la ecuación 1.9. Donde C_l representan los coeficientes de la aproximación descritos en [5] (los cuales dependen del esquema de diferencias y del orden de la derivada) y $P^n_{i,j,k}$ es el valor discretizado de la intensidad de presión para el tiempo t y las dimensiones x, y, z donde i, j, k representan los vectores unitarios de las respectivas dimensiones.

$$\begin{gathered}\frac{C_{-1}P^n_{i-1,j,k} + C_0P^n_{i,j,k} + C_1P^n_{i+1,j,k}}{\Delta x^2} + \frac{C_{-1}P^n_{i,j-1,k} + C_0P^n_{i,j,k} + C_1P^n_{i,j+1,k}}{\Delta y^2} + \\ \frac{C_{-1}P^n_{i,j,k-1} + C_0P^n_{i,j,k} + C_1P^n_{i,j,k+1}}{\Delta z^2} = \frac{1}{v^2}\frac{P^{n-1}_{i,j,k} - 2P^n_{i,j,k} + P^{n+1}_{i,j,k}}{\Delta t^2}\end{gathered} \tag{1.9}$$

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

$$\begin{gathered}P^{n+1}_{x,y,z} = -P^{n-1}_{x,y,z} + 2P^n_{x,y,z} + \\ \Delta t^2 v^2 (\frac{C_{-1}P^n_{i-1,j,k} + C_0P^n_{i,j,k} + C_1P^n_{i+1,j,k}}{\Delta x^2} + \frac{C_{-1}P^n_{i,j-1,k} + C_0P^n_{i,j,k} + C_1P^n_{i,j+1,k}}{\Delta y^2} + \\ \frac{C_{-1}P^n_{i,j,k-1} + C_0P^n_{i,j,k} + C_1P^n_{i,j,k+1}}{\Delta z^2})\end{gathered} \tag{1.10}$$

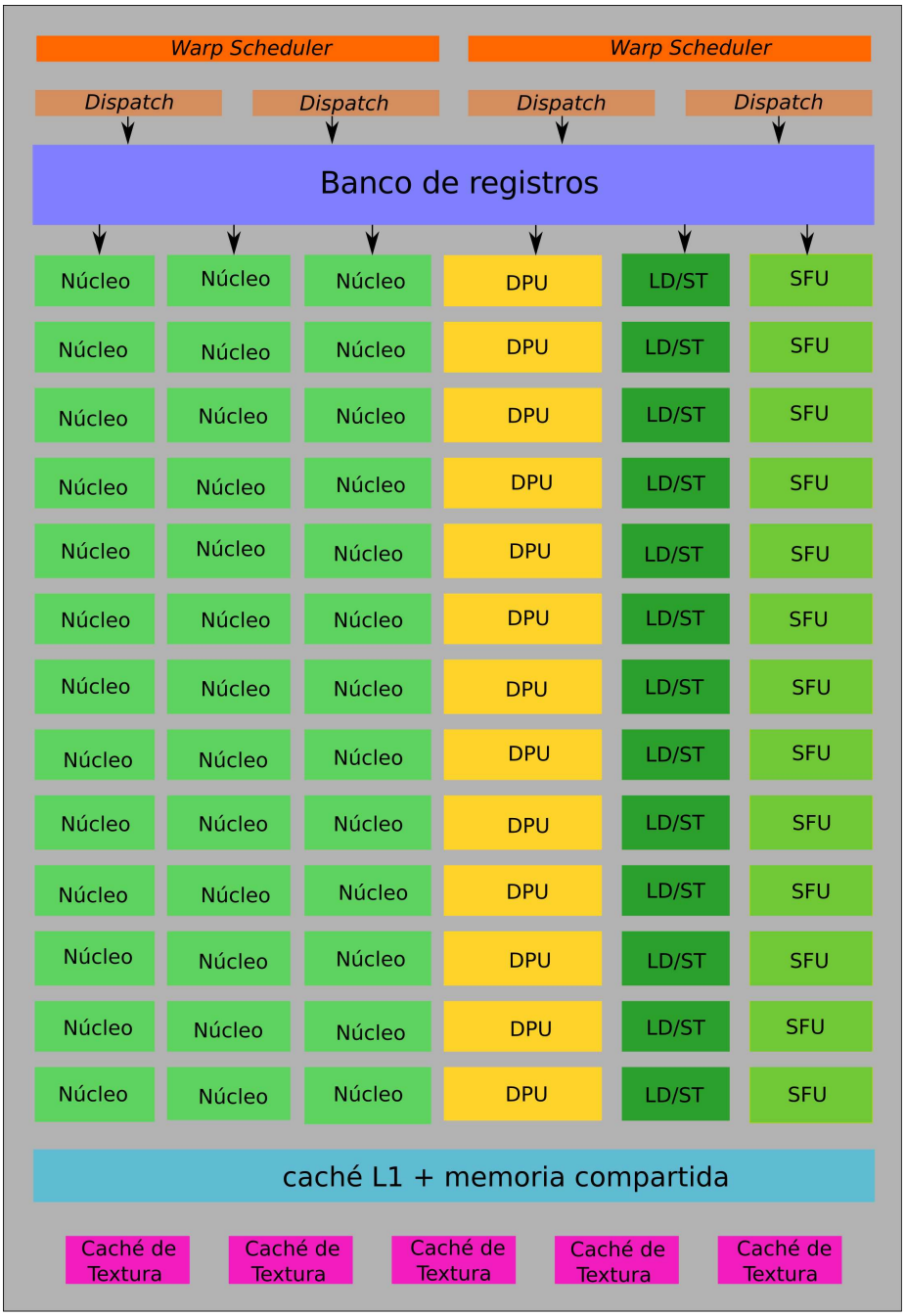
La ecuación discretizada 1.9 puede ser implementada en plataformas de cómputo basadas en CPU o GPU entre otros.

1.3. UNIDADES DE PROCESAMIENTO GRÁFICO (GPU)

En aplicaciones donde el volumen de datos es grande (e.g. RTM3D), las plataformas actuales de cómputo como las *Central processing units* (CPUs) presentan un *throughput* bajo (del orden de los 400 Gflops/s en una CPU con microarquitectura Sandy Bridge [2]). Plataformas como las *Graphics processing units* (GPUs) presentan un *throughput* mayor (ordenes 3050 Gflops/s en una GPU con arquitectura Kepler [2]).

Las GPUs (Graphic Processing Units), también conocidas como unidades de procesamiento gráfico, son circuitos integrados que disponen de muchos núcleos funcionando de manera concurrente. Los núcleos se agrupan en SM (*Streaming Multiprocessors*), además, cada SM dispone de unidades de memoria (registros, caché L1, memoria compartida) y unidades funcionales (SFU, DPU, LD/ST, *warp scheduler*). La tabla 1 tiene una breve descripción de algunas de las unidades funcionales de una GPU. La disposición de estas unidades en un SM dentro de la GPU se muestra en la Figura 1.

Figura 1: Unidades Funcionales en un SM, arquitectura *kepler*.



Fuente: Adaptado de [1].

Tabla 1: Unidades funcionales de una GPU.

Modulo	Función
Banco de registros	Son usados para la ejecución de los *threads*.
cahé L1	Reduce la latencia a la memoria local o global.
Memoria compartida	Rápido intercambio de datos entre *threads*.
SFU	Núcleos o unidades de funciones especiales que ejecutan operaciones de punto flotante con precisión sencilla para funciones trascendentales, cada SFU ejecuta una instrucción por *thread* y por ciclo de reloj.
DPU	Núcleos que ejecutan operaciones de punto flotante con precisión doble (número de instruciones por *thread* y por ciclo de reloj).
Warp scheduler	Realiza cambios de contextos rápidos entre *threads* y entrega instrucciones a los *warps* que están listos para ejecutar.
Caché de Textura	Aumenta el ancho de banda de la memoria de textura.
LD/ST	Son elementos por los cuales los SMs pueden leer o escribir en la memoria global para la ejecución de los *threads*.
Núcleo	Puede ejecutar instrucciones de punto flotante y de enteros de precisión simple.
Dispatch	Envían instrucciones de ejecución.

Fuente: Tomado de [2].

1.3.1. Modelo de programación CUDA

CUDA (*Compute Unified Device Architecture*) es un modelo de programación creado por NVIDIA para realizar cálculos en paralelo. Algunos componentes del modelo de programación CUDA son: *Thread*, bloque de *Thread*, *Grid* y *Warp*.

***Thread*:** Encargado de ejecutar la instrucciones sobre un dato o conjuntos de datos en específico.

Bloque de *Threads*: Agrupación de *threads*, máximo 1024. Los *threads* en un bloque pueden intercambiar datos dentro de memoria compartida trabajando eficientemente. Los *threads* en diferente bloques solo pueden compartir datos entre ellos dentro de la caché L2 y memoria global.

***Grid*:** Agrupación de bloques de *threads*, define el tamaño del *kernel*.

El *kernel* constituye una parte fundamental del sistema operativo.Es el principal responsable de facilitar a los distintos programas acceso seguro al hardware de la computadora, es el encargado de gestionar recursos a través de servicios de llamada al sistema. También se encarga de decidir qué programa podrá usar un dispositivo de hardware y durante cuánto tiempo ya que el acceso al hardware es limitado. Los *kernels* tienen como funciones básicas garantizar la carga y la ejecución de los subprocesos y proponer una interfaz entre el espacio núcleo y los programas del espacio del usuario.

***Warp*:** Es un grupo de 32 *threads*, que son lanzados simultáneamente.

Memoria compartida

Debido a que está en el chip, la memoria compartida es mucho más rápida que la memoria local y la memoria global. La memoria compartida es asignada por bloque de *threads*, por lo que todos los *threads* en un mismo bloque tienen acceso a la memoria compartida.
Los *threads* pueden acceder a los datos en memoria compartida cargados desde memoria global por otros *threads* dentro del mismo bloque de *threads*.

Cuando existe un intercambio de datos entre *threads* se deben evitar las condiciones de carrera. Digamos que dos *threads* A y B cada uno carga un elemento de datos de la memoria global y lo almacenan en la memoria compartida. A continuación, A quiere leer el elemento B de la memoria compartida, y viceversa. Supongamos que A y B son los *threads* en dos *warps* diferentes. Si B no ha terminado de escribir su elemento antes de que A intente leerlo, tenemos una condición de carrera, lo que puede conducir a un comportamiento indefinido y resultados incorrectos.

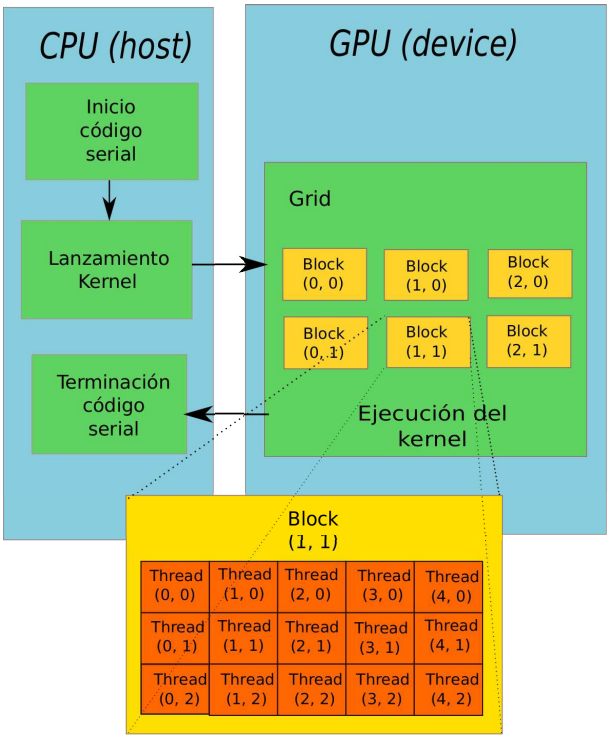
Para asegurar unos resultados correctos, hay que sincronizar los *threads*. CUDA proporciona una simple barrera de sincronización primitiva, la cual puede ser invocada por *syncthreads()*. Un *thread* solo puede ejecutar instrucciones hasta *syncthreads()*, donde debe esperar a que los otros *threads* en su bloque ejecuten hasta la barrera *syncthreads()* para poder seguir ejecutando el programa. Por lo tanto, podemos evitar la condición de carrera que se ha descrito más arriba, llamando *syncthreads()* después de almacenar los datos en la memoria compartida y antes de de que cualquier *thread* cargue datos desde la memoria compartida.

1.3.2. Programación Heterogénea

El proceso de la programación heterogénea se muestra en la Figura 2. En la programación heterogénea la CPU (*host*) se encarga de ejecutar la parte secuencial del algoritmo, mientras que la GPU (*device*) es la encargada de las operaciones de cómputo intensivo que pueden ejecutarse en paralelo. La programación heterogénea permite aprovechar la capacidad que tiene la GPU de realizar cálculos en paralelo [2].

La ejecución del algoritmo comienza en la CPU. Luego un *kernel* se lanza desde la CPU hacia la GPU. En este lanzamiento se especifican los parámetros tales como el número de *threads*, el número de bloques de *threads* y la cantidad de memoria compartida. Cuando la GPU termina la ejecución del *kernel*, los resultados se regresan a la CPU para continuar con la ejecución.

Figura 2: Programación Heterogénea.



Fuente: Adaptado de [2].

1.3.3. Indexado en CUDA

Es la operación encargada de asignar un *thread* a cada dato a calcular (i.e. dato de salida). Es decir, en el indexado en CUDA asignamos los índices de los *threads* encargados de calcular cada uno de los elementos que conforman la salida.

A cada subproceso que ejecuta el *kernel* se le asigna un único ID de *thread*, el cual es accesible dentro del *kernel* a través de las variables $threadIdx$, $blockDim$ y $blockIdx$ que define CUDA. Estas variables predefinidas son de tres dimensiones (DIM3) [2]. De modo que los *threads* se pueden identificar usando un índice de *thread* de una, dos, o tres dimensiones, formando un bloque de una, dos, o tres dimensiones de *threads*, llamada secuencia de *threads*.

Esto proporciona una manera natural de invocar el cálculo a través de los elementos de un dominio tal como un vector, matriz o volumen. Cabe resaltar que el programador debe especificar el número de *threads* por bloque de *threads* y número de bloques de *threads* por *grid* [2].

Tabla 2: Funciones para el indexado de los *threads*.

Comando	Descripción
$threadIdx.x, y, z$	Índice o identificador del *thread* dentro de su bloque de *thread*.
$blockIdx.x, y, z$	Índice o identificador del bloque de *threads* en la *grid*.
$blockDim$	Identificador que contiene el tamaño de cada bloque de *threads*.

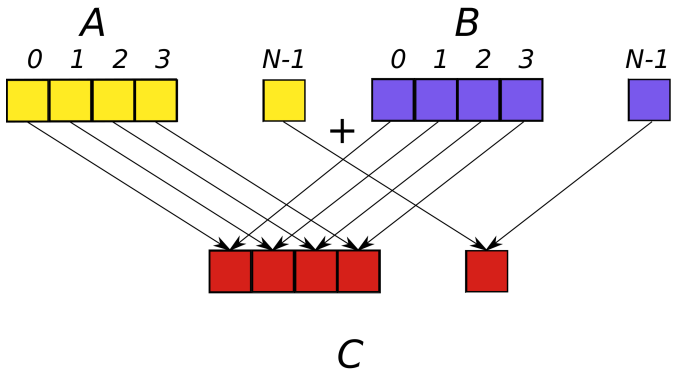
Fuente: Tomado de [6].

Como ejemplo se muestra una suma de vectores usando memoria global y memoria compartida. Se tienen dos vectores de entrada A y B de N elementos y se quiere calcular el vector de salida C.

Suma de vectores en memoria global

Para la creación del kernel se propone que cada *thread* calcule un único elemento de C como se muestra en la Figura 3.

Figura 3: suma de vectores.



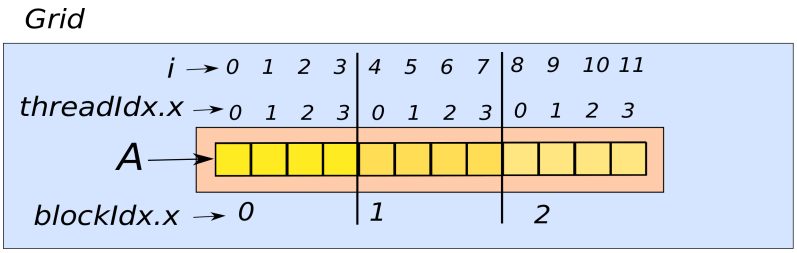
Se crea el kernel que realice lo propuesto, el cual se muestra en el cuadro de código 1.1 se muestra el *kernel*, en la línea 1 se guarda el nombre de cada uno de los vectores, en la línea 2 realiza el indexado global. Con el if se realiza la suma y se guarda en el vector C. En este caso se trabaja solo en la componente X, se debe tener en cuenta que las funciones para el indexado de los *threads* se muestran en la Tabla 2.

Cuadro de Código 1.1: Código del *kernel* que suma dos vectores en memoria global. Adaptado de [7].

```
1 __global__ void VecAdd(float * A, float * B, float * C, int N){
2 int i = blockDim.x * blockIdx.x + threadIdx.x;
3 if (i < N){
4 C[i] = A[i] + B[i]; } }
```

En la Figura 4 se muestra el indexado del vector A para la suma de vectores en memoria global enviando tres bloques de *threads* ($blockIdx.x = [0, 1, 2]$) con N igual a doce y $blockDim.x$ igual a cuatro ($threadIdx.x = [0, 1, 2, 3]$).

Figura 4: Indexado de un vector en memoria global.



Fuente: Adaptado de [4].

Suma de vectores en memoria compartida

Se puede visualizar en el cuadro de código 1.2, en la línea 1 se guarda el nombre de cada uno de los vectores, en las líneas 2 y 3 se realizan el indexado global y el indexado en memoria respectivamente. En la línea 5 se realiza la reserva de memoria compartida en forma estática, con las líneas 6 y 7 se realiza la copia de los vectores A y B de memoria global a compartida (As y Bs). Con el if se realiza la suma y se guarda en el vector Cs que se encuentra en memoria compartida y luego con la última línea se guarda el valor final en memoria global.

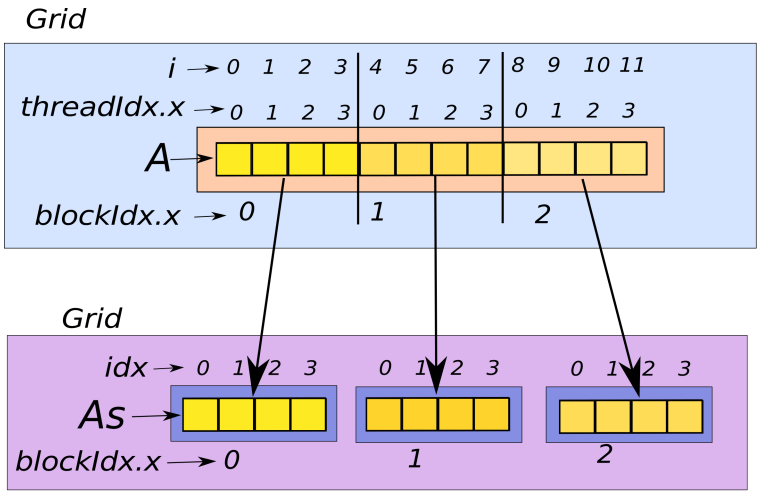
Cuadro de Código 1.2: Código del *kernel* que suma dos vectores en memoria compartida

```
1  __global__ void VecAdd(float * A, float * B, float * C){
2  int i = blockDim.x * blockIdx.x + threadIdx.x;
3  int idx = threadIdx.x;
4  Ns = blockDim.x;
5  __shared__float As[Ns],Bs[Ns],Cs[Ns];
6  As[idx] = A[i];
7  Bs[idx] = B[i];
8  if (idx < Ns){
9  Cs[idx] = As[idx] + B[idx]; }
10 C[i] = Cs[idx];}
```

En la Figura 5 se muestra tanto el indexado del vector A para la suma de vectores en memoria global como el indexado en memoria compartida del vector As enviando tres bloques de *threads* con

blockDim.x igual a cuatro.

Figura 5: Indexado de un vector en memoria compartida.



Lanzamiento del Kernel

El kernel suma VecAdd es lanzado por la declaración:

```
1 VecAdd<<< n_blocks, block_size >>>;
```

La información entre <<<, , >>> es la configuración de ejecución, que dicta cómo muchos de los *threads* del dispositivo ejecutan el *kernel* en paralelo. En el modelo de programación CUDA hablamos de lanzar un *kernel* con una *grid* de bloques de *thread*. El primer argumento en la configuración de ejecución especifica el número de bloques de *threads* en la *grid* y el segundo especifica el número de *threads* en un bloque de *threads*.

Los bloques de *threads* y las *grids* pueden ser de una, dos o tres dimensiones que se pueden describir mediante el macro DIM3 (un macro sencillo definido por CUDA con las direcciones x, y, z respectivamente), para este caso se usa una dimensión.

Se lanza el *kernel* con un número de bloques de *threads*, para determinar el número de bloques de *threads* necesarios (n_blocks) para procesar todos los elementos del modelo se usa la ecuación 1.11.

$$((Nx - 1)/4) \; = \; n_blocks, \tag{1.11}$$

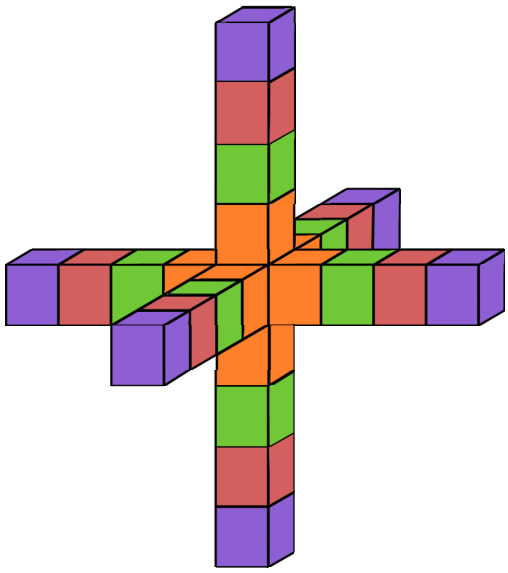
```
1     int block_size = 4;
2     int n_blocks = (((Nx-1)/4));
```

1.3.4. *Stencils*

Existen diversas aplicaciones tanto en la ingeniería como en la ciencia donde es necesario la resolución de ecuaciones en derivadas parciales. La implementación de la solución de ecuaciones parciales es compleja y computacionalmente costosa. Por esta razón se calcula una solución de forma aproximada usando el método de diferencias finitas. Las diferencias finitas se implementan usando *stencils* los cuales son patrones de cómputo en los que cada elemento de salida se calcula usando sus elementos vecinos a los cuales se les da un peso de acuerdo a su proximidad [8]. En la Figura 6 se puede observar un *stencil* de orden 8.

Para la ejecución en paralelo de un modelo se utiliza el método de segmentación de dominio el cual es una estrategia de descomposición que divide la *grid* en secuencias de *threads* y registra los bloques de *threads* para evitar fallos en la caché de último nivel y explotar paralelismo a nivel de datos. Este método se utiliza para mejorar la localización en la caché tanto en CPU como en GPU con arquitecturas multinúcleos y la reutilización de datos [9].

Figura 6: *Stencil* de Orden 8.



Fuente: Adaptado de [7].

1.4. MÉTODOS DE SEGMENTACIÓN DE DOMINIO

Los métodos de segmentación de dominio son diferentes formas en las cuales podemos distribuir la manera como se va a procesar un volumen de datos debido a que no siempre se tiene el número

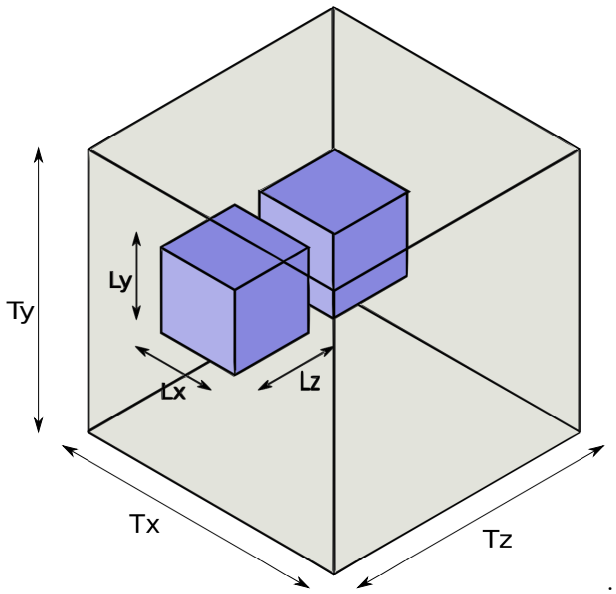
suficiente de núcleos para asignarle a cada núcleo un dato, entonces se deben organizar los recursos para reutilizarlos y así procesar volúmenes de datos más grandes que el número de núcleos que hay en la GPU.
Muchos de los parámetros (entradas al modelo) utilizados para realizar este método son dependientes de factores relacionados con la arquitectura de la GPU, tales como el número de registros y limitaciones en los tamaños de la caché.

Entre los diferentes métodos de segmentación se encuentra 3D, 2.5-D e In-plane.

1.4.1. Método de segmentación de dominio 3-D

En este método también conocido como método tradicional, el modelo de tamaño (T_x, T_y, T_z) se descompone en bloques más pequeños de tamaño (L_x, L_y, L_z), cada sub-bloque es procesado por un bloque de *threads*. Cada sub-bloque contiene un volumen de datos que necesita tener acceso al bloque incluyendo las regiones de *halo*, (Regiones de *halo* son los valores que se encuentran por fuera del tamaño del bloque de *thread* pero que son necesarios para calcular el *stencil*) adicionales en las seis caras del bloque. El método de segmentación 3-D es mostrado en la Figura 7.

Figura 7: Método de segmentación de dominio espacial 3-D.

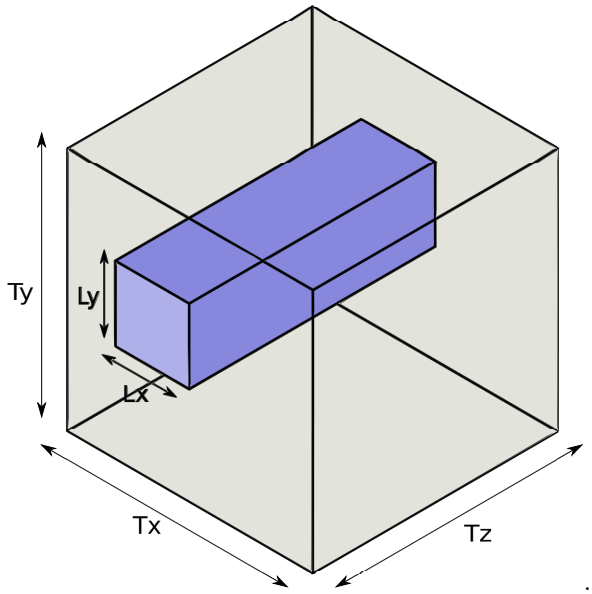


Fuente: Adaptado de [9].

1.4.2. Método de segmentación de dominio 2.5-D

El principio fundamental de este método es que las cargas redundantes de las regiones de *halo* en la dirección Z pueden reducirse ya que todo el bloque de datos no necesita ser cargado al mismo tiempo durante el cálculo de los elementos dentro del bloque. En este método cada *thread* almacena una copia de los elementos en los registros. Los *threads* atraviesan el plano Z, los registros se mueven y actualizan de forma constante, de tal manera que solo algunos puntos son guardados. En la Figura 8 se muestra el método de segmentación 2.5-D.

Figura 8: Método de segmentación de dominio espacial 2.5-D.



Fuente: Adaptado de [9].

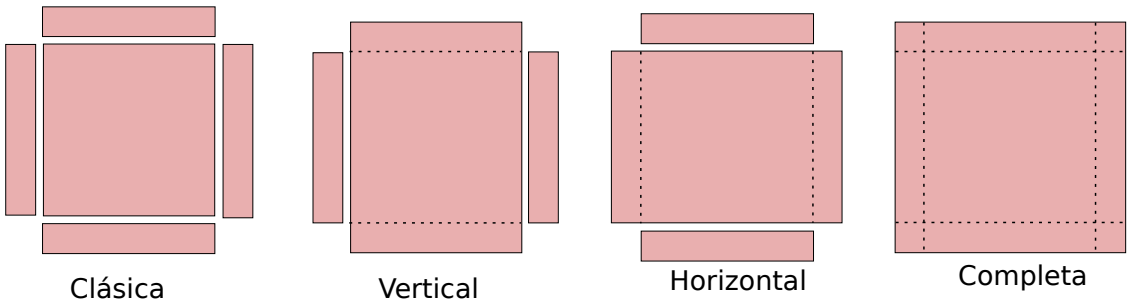
Otra opción reciente en el estado del arte es el método de segmentación de dominio In-plane el cual se estudiará en el siguiente capítulo.

Capítulo 2

MÉTODO DE SEGMENTACIÓN DE DOMINIO IN-PLANE

En este método se divide el cubo en planos y cada plano se carga en bloques de *threads* (donde los bloques de *threads* del plano se cargan en paralelo) en la memoria compartida. Un total de elementos de salida r se almacenan en la caché. Después de cargar cada plano, los elementos de salida anteriores se actualizan y se desplazan más lejos.
Existen algunas formas en que los datos de entrada se pueden cargar con el método In-plane. Estos modelos se muestran en la Figura 9.

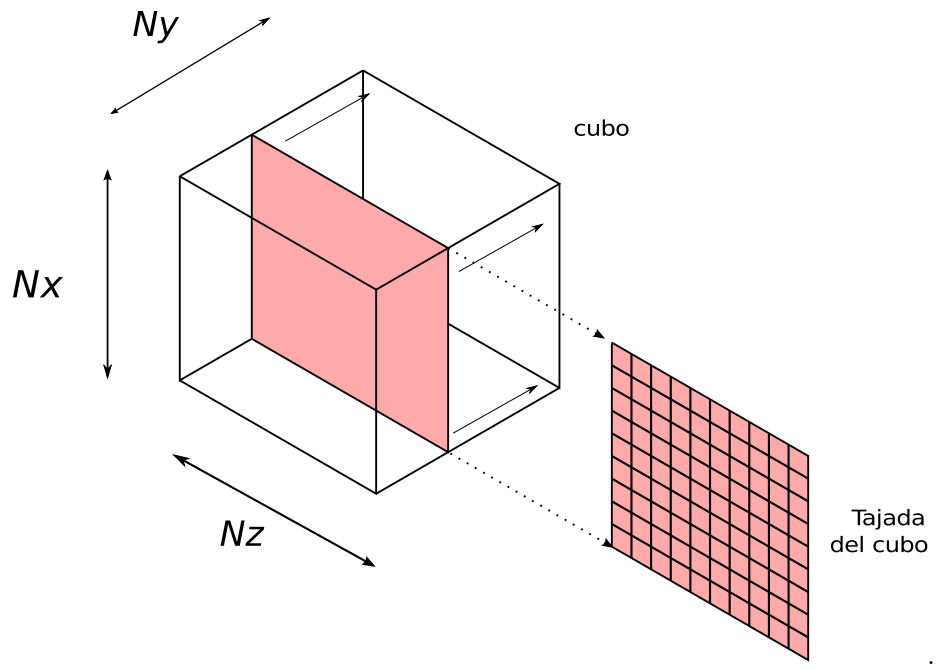
Figura 9: Diferentes formas en que los datos de la *grid* se pueden cargar con el método In-plane.



Fuente: Adaptado de [9].

El método In-plane se enfoca en cargar un plano e ir sumando los aportes de los demás planos con respecto al espacio (en Y), como se muestra en la Figura 10. En la carga de cada plano interactúan una cantidad de bloques de *threads*.

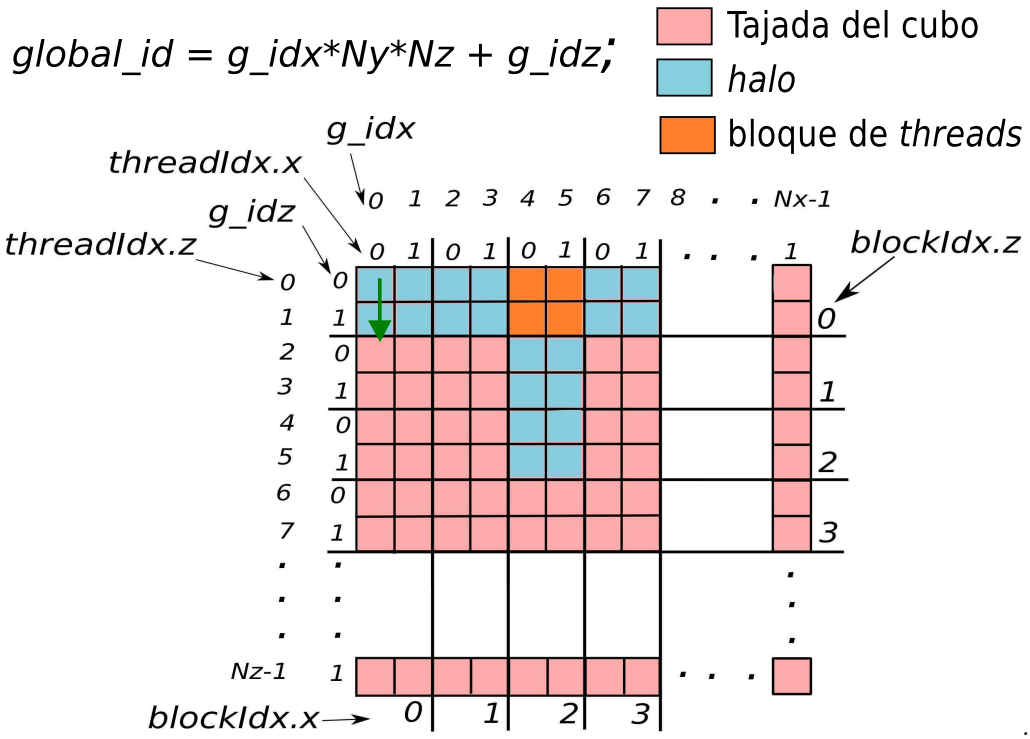
Figura 10: Dinámica método In-plane.



2.1. IN-PLANE FORMA DE CARGA CLÁSICA

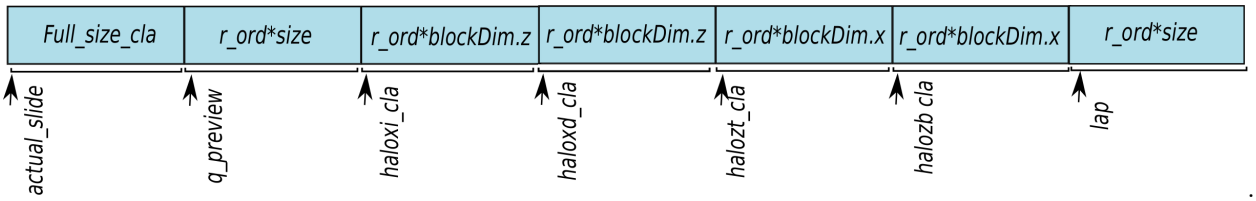
En esta variante de carga del método In-plane se utiliza la memoria compartida de la GPU para almacenar el bloque de *threads* y los *halos* en diferentes variables cada uno. Las variables definidas en el código del *device* no necesitan ser especificadas como variables del *device*, ya que residen en el.
Primero se realiza el indexado global para identificar algunas variables. En la Figura 11 se ve como es el indexado global de la tajada, también se toma como ejemplo el cuadro naranja que es el bloque de *threads* (a analizar) junto con los *halos* que se encuentran de color azul. Las funciones $threadIdx.z$ y $threadIdx.x$ son usadas para generar los índices en la dirección Z y X respectivamente. Mientras que las variables g_idx y g_idz crean los índices para cada uno de los bloques en las respectivas direcciones y por último la variable $global\_id$ representa el indexado global, el indexado global se realiza en la dirección de la flecha verde.

Figura 11: Indexado de una tajada en memoria global.



Antes de cargar el bloque de *threads* (a analizar) a la memoria compartida se debe asignar la memoria compartida, en la Figura 12 se muestra la asignación de la memoria compartida, para este caso la memoria compartida se divide en siete partes, una para guardar el bloque de *threads* ($Full\_size\_cla$), cuatro para guardar cada uno de los *halos* ($r\_ord * blockDim.z$, $r\_ord * blockDim.z$, $r\_ord * blockDim.x$, $r\_ord * blockDim.x$), una para guardar el cálculo del laplaciano enunciado en el capitulo uno ($r\_ord * size$) y la otra ($r\_ord * size$) para guardar los bloques de *threads* de las demás tajadas de plano necesarios para el calculo. En la Figura, $size$ representa el tamaño en X por el tamaño en Z del bloque (a analizar), r_ord es tamaño del *halo* y $Full\_size\_cla$ en este caso es igual a $size$. Los valores de $size$ y r_ord son iguales para todas las formas de carga.

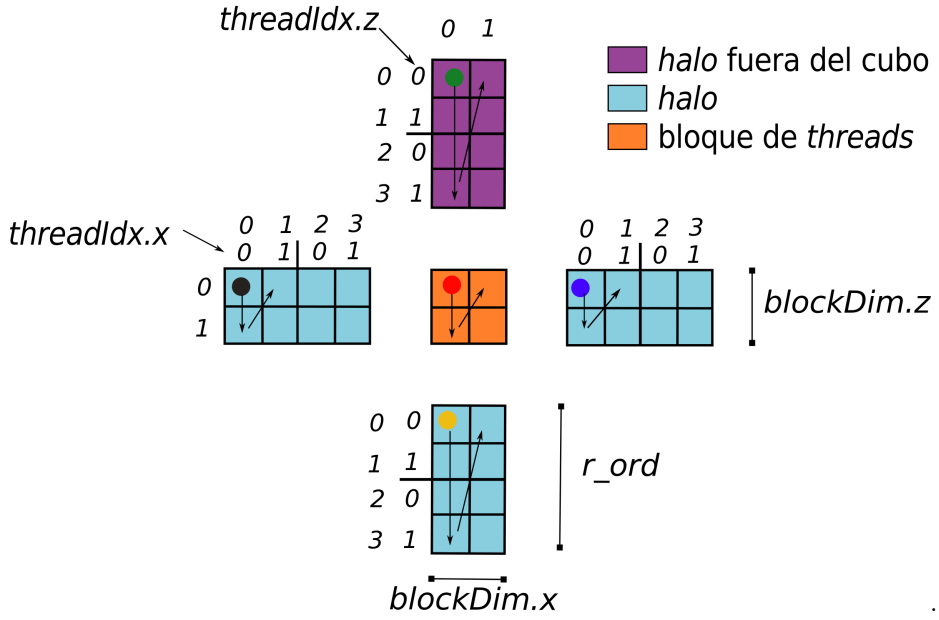
Figura 12: Asignación de la memoria compartida, carga clásica.



Basándose en la tajada actual del cubo, se necesita saber cuál bloque se desea cargar tanto en la dirección X como en la dirección Z, se debe tener en cuenta el indexado de cada bloque en memoria compartida.

Si los *halos* o parte de ellos no se encuentran dentro del cubo, al cargar a memoria compartida esta parte se carga como valor cero ya que se necesitan los *halos* completos para así poder realizar el cálculo, esto se muestra en la Figura 13 por el color morado en el *halo*. Para el indexado de los *halos* se debe tener en cuenta que son dos veces más grandes en una dirección ya sea en X o Z que el bloque principal. Para la carga de los *halos* de memoria global a memoria compartida se parte de la ubicación del bloque principal en memoria global (se toma como referencia el punto rojo en el centro de la figura), los demás puntos hacen referencia al inicio del indexado de cada uno de los elementos. Para determinar la ubicación del *halo* de la izquierda se le resta *r_ord* en la dirección X, la ubicación del *halo* de la derecha se obtiene sumando *r_ord* y *blockDim.x* en la dirección X. En el caso del *halo* superior se resta *r_ord* en la dirección Z y para la ubicación del *halo* inferior se suma *r_ord* y *blockDim.z* en la dirección Z.

Figura 13: Dinámica de carga e indexado del bloque (a analizar) y de los *halos* en memoria compartida, carga clásica.



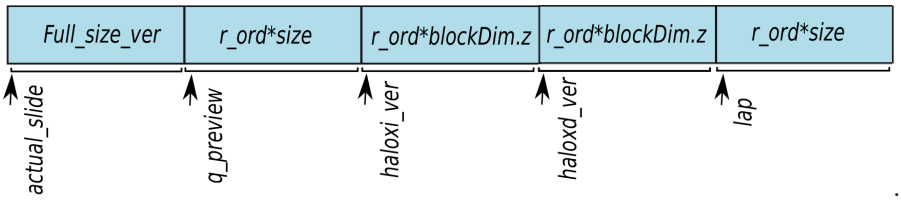
2.2. IN-PLANE FORMA DE CARGA VERTICAL

Otra de las variantes en la carga del método In-plane es la carga vertical. En esta se utiliza la memoria compartida de la GPU para almacenar cada bloque de la tajada junto con los *halos* superior e inferior en una misma variable y los *halos* en la dirección X se almacena cada uno por aparte en variables

diferentes.
En la Figura 11 se ve como es el indexado global de la tajada, al igual que en la variante de carga anterior esta Figura se toma como referencia. En la Figura 14 se realiza la asignación de la memoria compartida.

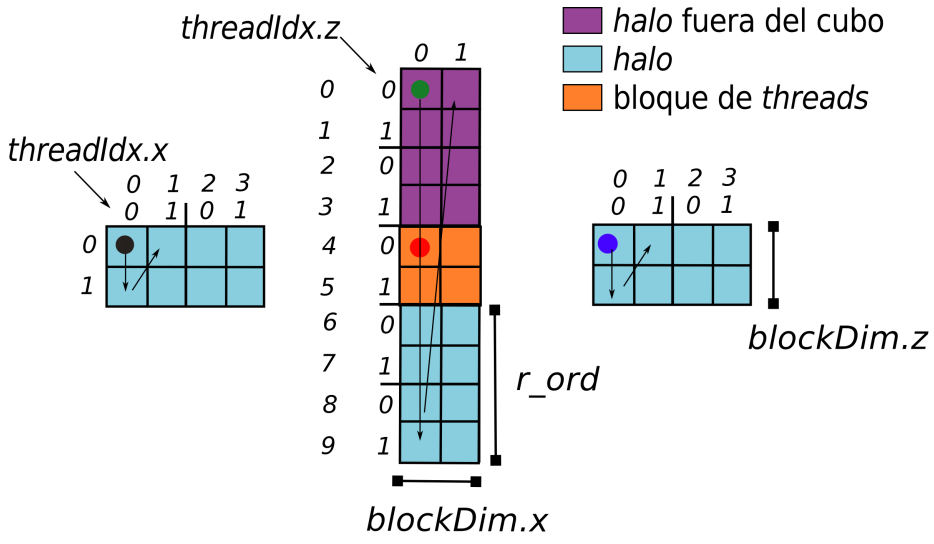
La memoria compartida se divide en cinco partes, una para guardar el nuevo bloque ($Full\_size\_ver$), dos para guardar los *halos* de la derecha y de la izquierda ($r\_ord * blockDim.z$, $r\_ord * blockDim.z$), una para guardar el cálculo del laplaciano ($r\_ord * size$) y la otra ($r\_ord * size$) para guardar los guardar los nuevos bloques de las demás tajadas de plano necesarios para el calculo final, donde $Full\_size\_ver$ representa el tamaño en X del bloque (a analizar) por el tamaño de la carga del bloque (a analizar) junto con los *halos* en Z.
La diferencia con la anterior variante es que al cargar los *halos* superior e inferior junto con el bloque se quitan dos particiones.

Figura 14: Asignación de la memoria compartida, In-plane carga vertical.



Basándose en la tajada actual del cubo, se necesita saber cúal bloque se desea cargar tanto en la dirección X como en la dirección Z, además, se debe tener en cuenta el indexado de cada bloque en memoria compartida, en este caso el bloque (a analizar) junto con los *halos* superior e inferior forman un nuevo bloque.
Para la carga tanto del nuevo bloque como de los demás *halos* de memoria global a memoria compartida se parte de la ubicación del bloque (a analizar) en memoria global (se toma como referencia el punto rojo en el centro de la Figura 15), los demás puntos hacen referencia al inicio del indexado de cada uno de los elementos.
Para determinar la ubicación del nuevo bloque se resta r_ord en la dirección Z, se debe tener en cuenta también el indexado del nuevo bloque en memoria compartida. La ubicación del *halo* de la izquierda se obtiene restando r_ord en la dirección X, para definir la ubicación del *halo* de la derecha se le suma r_ord y $blockDim.x$ en la dirección X.

Figura 15: Dinámica de carga e indexado del nuevo bloque y de los demás *halos* en memoria compartida, carga vertical.



2.3. IN-PLANE FORMA DE CARGA HORIZONTAL

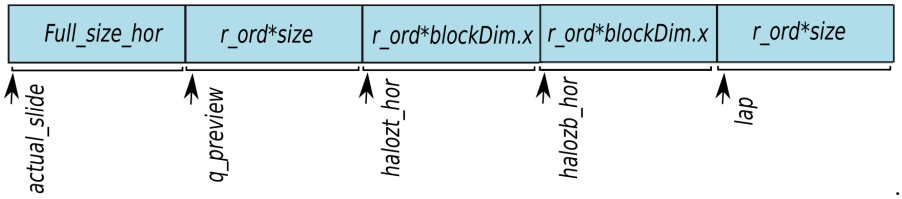
Otra de las variantes en la carga del método In-plane es la carga horizontal. En esta se utiliza la memoria compartida de la GPU para almacenar cada bloque de la tajada junto con los *halos* de la izquierda y de la derecha en una misma variable y los *halos* en la dirección Z se almacena cada uno por aparte en variables diferentes.

En la Figura 11 se ve como es el indexado global de la tajada, al igual que en las variantes de carga anteriores esta Figura se toma como referencia. En la Figura 16 se realiza la asignación de la memoria compartida.

La memoria compartida se divide en cinco partes, una para guardar el nuevo bloque ($Full\_size\_hor$), dos para guardar los *halos* de la superior e inferior ($r\_ord * blockDim.x$, $r\_ord * blockDim.x$), una para guardar el cálculo del laplaciano ($r\_ord * size$) y la otra ($r\_ord * size$) para guardar los guardar los nuevos bloques de las demás tajadas de plano necesarios para el calculo final, donde $Full\_size\_hor$ representa el tamaño de la carga del bloque (a analizar) junto con los *halos* en X por el tamaño en Z del bloque (a analizar).

La diferencia es que en esta variante se cargan los *halos* de la izquierda y de la derecha junto con el bloque (a analizar) en vez de los *halos* superior e inferior.

Figura 16: Asignación de la memoria compartida, In-plane carga horizontal.

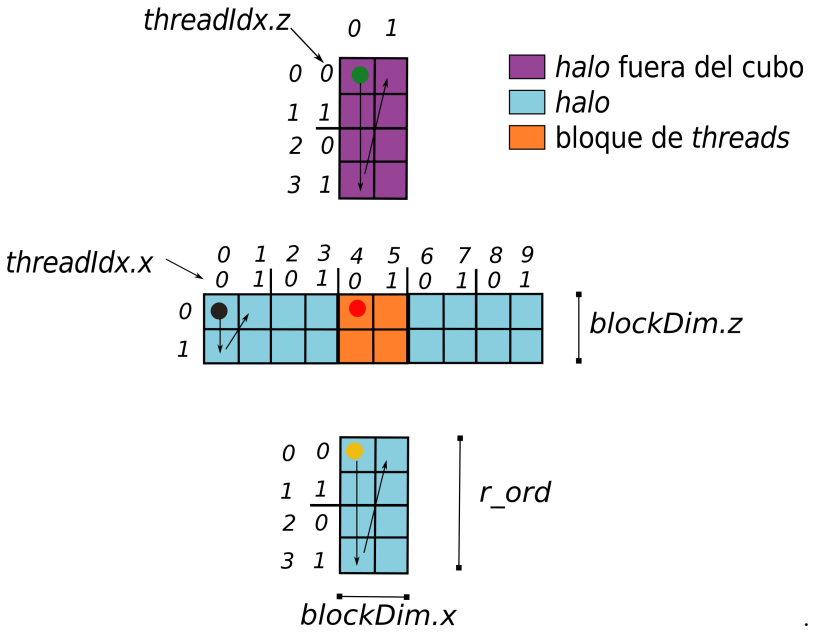


Basándose en la tajada actual del cubo, se necesita saber cúal bloque se desea cargar tanto en la dirección X como en la dirección Z, además, se debe tener en cuenta el indexado de cada bloque en memoria compartida, en este caso el bloque (a analizar) junto con los *halos* de la izquierda y de la derecha forman un nuevo bloque.

Para la carga tanto del nuevo bloque como de los demás *halos* de memoria global a memoria compartida se parte de la ubicación del bloque principal en memoria global (se toma como referencia el punto rojo en el centro de la Figura 17), los demás puntos hacen referencia al inicio del indexado de cada uno de los elementos.

Para determinar la ubicación del nuevo bloque se resta r_ord en la dirección X, se debe tener en cuenta el indexado de este en memoria compartida. La ubicación del *halo* superior se adquiere restando r_ord en la dirección Z, para determinar la ubicación del *halo* inferior se suma r_ord y $blockDim.x$ en la dirección Z.

Figura 17: Dinámica de carga e indexado del nuevo bloque y de los demás *halos* en memoria compartida, carga horizontal.

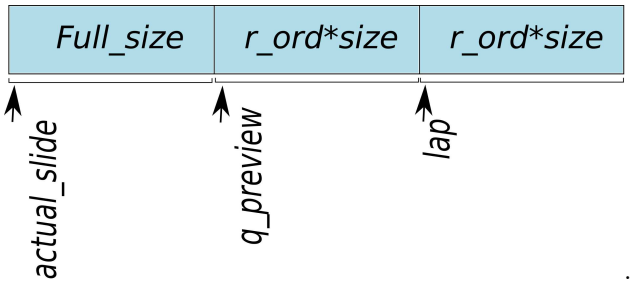


2.4. IN-PLANE FORMA FORMA DE CARGA COMPLETA

La última de las variantes en la carga del método In-plane es la carga completa. En esta se utiliza la memoria compartida de la GPU para almacenar tanto los *halos* como el bloque (a analizar) y algunos elementos adicionales (estos elementos adicionales se usan por facilidad al indexar), en este caso se almacena todo en conjunto.
En la Figura 11 se ve como es el indexado global de la tajada, al igual que las demás variantes de carga esta Figura se toma como referencia. En la Figura 18 se realiza la asignación de la memoria compartida.

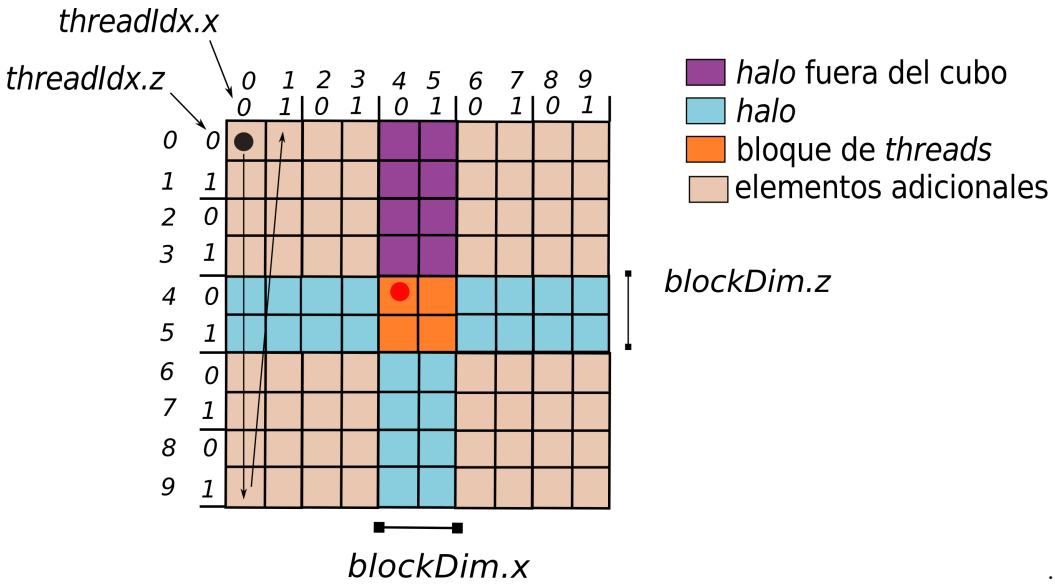
La memoria compartida se divide en este caso en tres partes, una para guardar el nuevo bloque ($Full\_size$), una para guardar el cálculo del laplaciano ($r\_ord * size$) y la otra ($r\_ord * size$) para guardar los guardar los nuevos bloques de las demás tajadas de plano necesarios para el calculo final, donde $Full\_size$ representa el tamaño de la carga del bloque (a analizar) junto con los *halos* en X por el tamaño de la carga del bloque (a analizar) junto con los *halos* en Z. La diferencia con las demás variantes es que se carga todo en conjunto.

Figura 18: Asignación de la memoria compartida, In-plane carga completa.



Al cargar el plano en memoria compartida se debe tener en cuenta el indexado de cada bloque a cargar en memoria compartida.
Para la carga del nuevo bloque de memoria global a memoria compartida se parte de la ubicación del bloque (a analizar) en memoria global (se toma como referencia el punto rojo que se encuentra en el centro de la Figura 19), los demás puntos hacen referencia al inicio del indexado de cada uno de los elementos.
Para determinar la ubicación del nuevo bloque se resta r_ord tanto en la dirección X como en la dirección Z, el indexado del nuevo bloque en memoria compartida se utiliza para determinar esta ubicación.

Figura 19: Dinámica de carga e indexado del nuevo bloque en memoria compartida forma de carga completa.



Capítulo 3

IMPLEMENTACIÓN MÉTODO IN-PLANE

En esta sección se presentan las implementaciones de las cuatro formas de carga de memoria del método In-plane (clásica, vertical, horizontal y completa), en cada una de las implementaciones se realiza una breve explicación y se muestra una parte del pseudo-código. Esta parte del pseudo-código es la que diferencia las implementaciones de las formas de carga.
Los códigos completos de cada una de las implementaciones se encuentran anexados en el apéndice que se encuentra al final de este libro.

3.1. IMPLEMENTACIÓN IN-PLANE FORMA DE CARGA CLÁSICA

Después de realizar la partición de la memoria compartida se realiza la carga de los diferentes elementos (bloque principal y *halos*). A continuación se muestran los diferentes pseudo-códigos usados para la carga del plano a memoria compartida.

3.1.1. Pseudo-código carga bloque principal

Representa la carga del bloque principal a la memoria compartida, en la primera línea se puede observar el indexado del bloque en memoria compartida. El primer if se realiza para tener la certeza de que el indexado del bloque se encuentra dentro del tamaño de este. Con la tercera línea se calcula la ubicación del bloque a cargar en memoria global, esto con el fin de cargar el bloque que se necesita y no otro bloque. Con el segundo if se busca cargar bloques que se encuentren dentro del tamaño del modelo, si esto no es así se debe cargar valores de cero, como se muestra en el Algoritmo 1.

Algoritmo 1 Pseudo-código carga bloque principal.

```
1: posición = indexado del bloque en memoria compartida
2: if posición < full_size_cla then
3:     x , z = índices en X y Z para ubicar el bloque principal en memoria global
4:     if x >= 0; x < Nx; z >= 0; z < Nz then
5:         cargar el bloque principal a memoria compartida
6:     else
7:         guardar valores de cero
8:     end if
9: end if
```

3.1.2. Pseudo-código carga *halos* en X

El pseudo-código para la carga de los *halos* izquierdo y derecho es similar al pseudo-código anterior. El indexado en cada uno de los *halos* es igual en memoria compartida y la diferencia se encuentra a la hora de calcular la ubicación de cada uno de ellos en la memoria global, esto se muestra en el Algoritmo 2

Algoritmo 2 Pseudo-código carga de los *halos* en X.

```
1: halox = indexado halos derecho e izquierdo en memoria compartida
2: if halox < full_size_cla then
3:     xi , zi = índices en X y Z para ubicar el halo izquierdo en memoria global
4:     xd , zd = índices en X y Z para ubicar el halo derecho en memoria global
5:     if xi >= 0; xi < Nx; zi >= 0; zi < Nz then
6:         cargar el halo izquierdo a memoria compartida
7:     else
8:         guardar valores de cero
9:     end if
10:    if xd >= 0; xd < Nx; zd >= 0; zd < Nz then
11:        cargar el halo derecho a memoria compartida
12:    else
13:        guardar valores de cero
14:    end if
15: end if
```

3.1.3. Pseudo-código carga *halos* en Z

El pseudo-código para la carga de los *halos* inferior y superior es similar al pseudo-código de la sección 3.1.2., pero en este los *halos* se encuentran ubicados en diferentes lugares de la memoria global y el indexado de los *halos* en memoria compartida cambia. Esto se muestra en el Algoritmo 3.

Algoritmo 3 Pseudo-código carga de los *halos* en Z.

```
1: haloz = indexado halos superior e inferior en memoria compartida
2: if haloz < full_size_cla then
3:     xs , zs = índices en X y Z para ubicar el halo superior en memoria global
4:     xin , zin = índices en X y Z para ubicar el halo inferior en memoria global
5:     if xs >= 0; xs < Nx; zs >= 0; zs < Nz then
6:         cargar el halo superior a memoria compartida
7:     else
8:         guardar valores de cero
9:     end if
10:    if xin >= 0; xin < Nx; zin >= 0; zin < Nz then
11:        cargar el halo inferior a memoria compartida
12:    else
13:        guardar valores de cero
14:    end if
15: end if
```

3.2. IMPLEMENTACIÓN IN-PLANE FORMA DE CARGA VERTICAL

Después de realizar la partición de la memoria compartida se realiza la carga de los diferentes elementos (bloque principal junto con los *halos* superior e inferior y los *halos* de la izquierda y de la derecha). La carga de los *halos* de la izquierda y de la derecha es igual que en la implementación anterior.
A continuación se muestra el pseudo-código usado para la carga del bloque principal junto con los *halos* superior e inferior a memoria compartida.

3.2.1. Pseudo-código carga nuevo bloque

Representa la carga del bloque principal junto con los *halos* superior e inferior a la memoria compartida, en la primera línea se puede observar el indexado del nuevo bloque. Este indexado cambia con respecto a la carga del bloque principal mostrado en la implementación anterior ya que su tamaño -que es más grande en la dirección Z- pasa de ser el tamaño del bloque a ser el tamaño del bloque más dos veces el tamaño del *halo*. El primer if se realiza para tener la certeza de que el indexado anterior se encuentre dentro del tamaño anteriormente mencionado, si esto se cumple se ubica cual es el bloque y los *halos* a cargar, si estos se encuentran dentro del tamaño del modelo se cargan, sino se cargan valores de cero, como se muestra en el Algoritmo 4.

3.3. IMPLEMENTACIÓN IN-PLANE FORMA DE CARGA HORIZONTAL

Después de realizar la partición la memoria compartida se realiza la carga de los diferentes elementos (bloque principal junto con los *halos* de la izquierda y de la derecha y los *halos* inferior y superior).

Algoritmo 4 Pseudo-código carga bloque principal junto con los *halos* superior e inferior.

```
1: posición.1 = indexado bloque junto a halos superior e inferior en memoria compartida
2: if posición.1 < full_size_ver then
3:     x , z = índices en X y Z para ubicar el bloque junto con los halos en memoria global
4:     if x >= 0; x < Nx; z >= 0; z < Nz then
5:         cargar el bloque junto con los halos a memoria compartida
6:     else
7:         guardar valores de cero
8:     end if
9: end if
```

La carga de los *halos* de la izquierda y de la derecha es igual que en la implementación anterior. A continuación se muestra el pseudo-código usado para la carga del bloque principal junto con los *halos* de la izquierda y de la derecha a memoria compartida.

3.3.1. Pseudo-código carga bloque principal junto con los *halos* en Z

Representa la carga del bloque principal junto con los *halos* de la izquierda y de la derecha a la memoria compartida, en la primera línea se puede observar el indexado del nuevo bloque. Este indexado cambia con respecto a la carga del bloque principal mostrado en la implementación In-plane forma de carga clásica, ya que su tamaño - que es más grande en la dirección X- pasa de ser el tamaño del bloque a el tamaño del bloque más dos veces el tamaño del *halo*. El primer if se realiza para tener la certeza de que el indexado anterior se encuentre dentro del tamaño anteriormente mencionado, si esto se cumple se ubica cual es el bloque y los *halos* a cargar, si estos se encuentran dentro del tamaño del modelo se cargan, sino se cargan valores de cero, como se muestra en el Algoritmo 5.

Algoritmo 5 Pseudo-código carga bloque principal junto con los *halos* de la izquierda y de la derecha.

```
1: posición.2 = indexado bloque junto a halos derecho e izquierdo en memoria compartida
2: if posición.2 < full_size_hor then
3:     x , z = índices en X y Z para ubicar el bloque junto con los halos en memoria global
4:     if x >= 0; x < Nx; z >= 0; z < Nz then
5:         cargar el bloque junto con los halos a memoria compartida
6:     else
7:         guardar valores de cero
8:     end if
9: end if
```

3.4. IMPLEMENTACIÓN IN-PLANE FORMA DE CARGA COMPLETA

Después de realizar la partición de la memoria compartida se realiza la carga de los diferentes elementos (bloque principal junto con todos los *halos*y otros elementos más). A continuación se muestra el pseudo-

código usado para la carga del bloque principal junto con los *halos* a memoria compartida.

3.4.1. Pseudo-código carga bloque principal junto con los *halos*

Representa la carga del bloque principal junto con todos los *halos* a la memoria compartida, en la primera línea se puede observar el indexado del nuevo bloque. Este indexado cambia con respecto a la carga del bloque principal mostrado en la implementación In-plane forma de carga clásica ya que su tamaño es más grande tanto en la dirección X como en la dirección Z, es decir, pasa de ser el tamaño del bloque a ser el tamaño del bloque más dos veces el tamaño del *halo* en cada dirección. El primer if se realiza para tener la certeza de que el indexado anterior se encuentre dentro del tamaño anteriormente mencionado, si esto se cumple se ubica cual es el bloque y los *halos* a cargar, si estos se encuentran dentro del tamaño del modelo se cargan, sino se cargan valores de cero, como se muestra en el Algoritmo 6.

Algoritmo 6 Pseudo-código carga *halos* en x

```
1: posición.3 = indexado bloque junto con halos en memoria compartida
2: if posición.3 < full_size then
3:     x , z = índices en X y Z para ubicar el bloque junto con los halos en memoria global
4:     if x >= 0; x < Nx; z >= 0; z < Nz then
5:         cargar el bloque junto con los halos a memoria compartida
6:     else
7:         guardar valores de cero
8:     end if
9: end if
```

3.5. COMPARACIÓN FORMAS DE CARGA MÉTODO IN-PLANE

Las implementaciones fueron realizadas en una GPU Nvidia K40. Las especificaciones para esta GPU se muestran en la Tabla 3.

Tabla 3: Especificaciones técnicas GPU K40.

Especificaciones técnicas GPU Nvidia K40		Capacidad de cómputo 3.5
Número de SMs en una GPU	SM	15
Ancho de banda de la memoria global	BW	288,384 GB/s
Máximo de memoria compartida por SM.	Smem	48KiB = 49152 $Byte$
Número máximo de registros por SM	Reg	64K, 65536
Número máximo de bloques de *threads* por SM	BLKSM	16
Máximo número de *warps* por SM	WarpSM	64
Frecuencia del reloj en una GPU.	Clock	745 MHz
Tamaño del *warp*		32 *threads*

Fuente: Adaptado de [2].

Con un modelo de entrada de tamaño [$Nx = 512$, $Ny = 256$, $Nz = 512$]. Los resultados de las diferentes implementaciones se muestran en las diferentes tablas que se encuentran a continuación, donde ($Ty = 1$).
En la Tabla 4 se muestran los tiempos tomados para las diferentes implementaciones tomando como tamaño del bloque en la dirección Z un *thread* y cambiando los valores del tamaño del bloque en X. Acá se puede observar que el menor tiempo de ejecución para todas las implementaciones se da cuanto el tamaño del bloque es [32,1,1].

Tabla 4: Tiempo de ejecución del *kernel* de cada una de las implementaciones con tamaño de bloque uno en Z y tamaño variable en X.

T_x	T_z	In-plane completa tiempo [s]	In-plane vertical tiempo [s]	In-plane horizontal tiempo [s]	In-plane clásica tiempo [s]
1	1	22.2032	9.9860	9.8375	8.2401
2	1	8.3582	4.7686	4.2853	3.6243
4	1	3.6592	2.3522	1.9746	1.7037
8	1	1.8259	1.2306	0.9775	0.8742
16	1	1.0588	0.6675	0.5203	0.4533
32	1	0.7081	0.3642	0.2923	0.2675

En la Tabla 5 se muestran los tiempos tomados para las diferentes implementaciones tomando como tamaño del bloque en la dirección Z dos *threads* y cambiando los valores del tamaño del bloque en X.

Acá se puede observar que el menor tiempo de ejecución para todas las implementaciones se da cuanto el tamaño del bloque es [32,1,2].

Tabla 5: Tiempo de ejecución del *kernel* de cada una de las implementaciones con tamaño de bloque dos en Z y tamaño variable en X.

T_x	T_z	In-plane completa tiempo [s]	In-plane vertical tiempo [s]	In-plane horizontal tiempo [s]	In-plane clásica tiempo [s]
1	2	8.2123	4.2748	4.5778	3.8397
2	2	3.4004	2.1601	2.1161	1.8003
4	2	1.6120	1.0689	0.9831	0.8480
8	2	0.8498	0.5504	0.4890	0.4337
16	2	0.5216	0.2882	0.2520	0.2290
32	2	0.5053	0.2381	0.2027	0.1761

En la Tabla 6 se muestran los tiempos tomados para las diferentes implementaciones tomando como tamaño del bloque en la dirección Z cuatro *threads* y variando los valores del tamaño del bloque de en X. Acá se puede observar que el menor tiempo de ejecución para todas las implementaciones se da cuanto el tamaño del bloque es [16,1,4].

Tabla 6: Tiempo de ejecución del *kernel* de cada una de las implementaciones con tamaño de bloque cuatro en Z y tamaño variable en X.

T_x	T_z	In-plane completa tiempo [s]	In-plane vertical tiempo [s]	In-plane horizontal tiempo [s]	In-plane clásica tiempo [s]
1	4	3.5076	1.9563	2.2272	1.8708
2	4	1.5940	0.9923	1.0207	0.8651
4	4	0.7860	0.4998	0.4893	0.4212
8	4	0.4434	0.2512	0.2379	0.2186
16	4	0.3498	0.1801	0.1673	0.1466
32	4	0.4476	0.2016	0.1707	0.1525

En la Tabla 7 se muestran los tiempos tomados para las diferentes implementaciones tomando como tamaño del bloque en la dirección Z ocho *threads* y modificando los valores del tamaño del bloque de en X. Acá se puede observar que el menor tiempo de ejecución para todas las implementaciones se da cuanto el tamaño del bloque es [8,1,8].

Tabla 7: Tiempo de ejecución del *kernel* de cada una de las implementaciones con tamaño de bloque ocho en Z y tamaño variable en X.

T_x	T_z	In-plane completa tiempo [s]	In-plane vertical tiempo [s]	In-plane horizontal tiempo [s]	In-plane clásica tiempo [s]
1	8	1.7474	0.9622	1.1257	0.9342
2	8	0.8282	0.4803	0.5155	0.4373
4	8	0.4389	0.2361	0.2456	0.2168
8	8	0.2836	0.1454	0.1428	0.1261
16	8	0.3733	0.1638	0.1552	0.1394
32	8	0.5587	0.2124	0.1882	0.1695

En la Tabla 8 se muestran los tiempos tomados para las diferentes implementaciones tomando como tamaño del bloque en la dirección Z dieciséis *threads* y variando los valores del tamaño del bloque en X.

Aquí se puede observar que el menor tiempo de ejecución para la forma de carga completa se da cuando el tamaño del bloque es [4,1,16]. Para las demás implementaciones el menor tiempo se da cuanto el tamaño del bloque es [8,1,16].

Tabla 8: Tiempo de ejecución del *kernel* de cada una de las implementaciones con tamaño de bloque dieciséis en Z y tamaño variable en X.

T_x	T_z	In-plane completa tiempo [s]	In-plane vertical tiempo [s]	In-plane horizontal tiempo [s]	In-plane clásica tiempo [s]
1	16	1.0059	0.4832	0.5669	0.4681
2	16	0.5075	0.2349	0.2576	0.2247
4	16	0.3387	0.1583	0.1599	0.1385
8	16	0.3677	0.1496	0.1483	0.1342
16	16	0.5021	0.1704	0.1682	0.1520
32	16	0.7886	0.2372	0.2335	0.2172

En la Tabla 9 se muestran los tiempos tomados para las diferentes implementaciones tomando como tamaño del bloque en la dirección Z treinta y dos *threads*, variando los valores del tamaño del bloque en X.

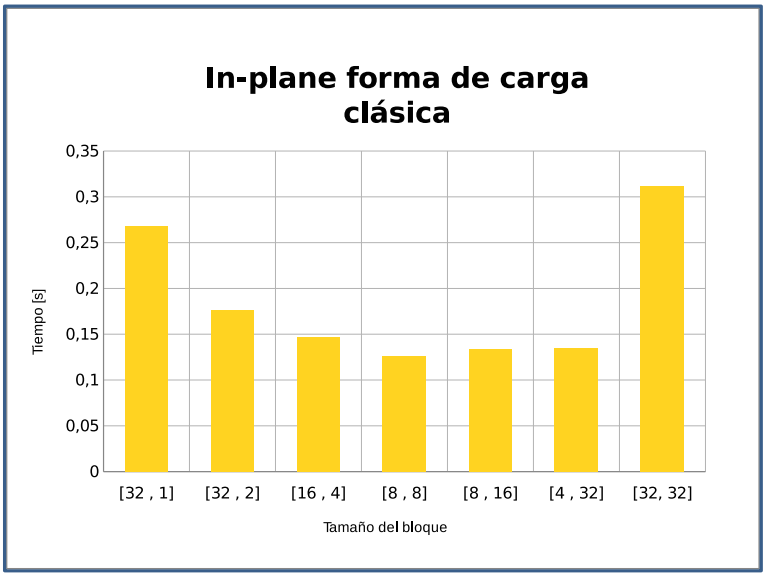
Aquí se puede visualizar que para cada una de las implementaciones de los diferentes métodos de carga el mejor resultado en tiempo, se obtiene con un tamaño de bloque de [4,1,32].

Tabla 9: Tiempo de ejecución del *kernel* de cada una de las implementaciones con tamaño de bloque treinta y dos en Z y tamaño variable en X.

T_x	T_z	In-plane completa tiempo [s]	In-plane vertical tiempo [s]	In-plane horizontal tiempo [s]	In-plane clásica tiempo [s]
1	32	0.6698	0.2367	0.2852	0.2411
2	32	0.4823	0.1749	0.1861	0.1596
4	32	0.4293	0.1483	0.1533	0.1343
8	32	0.5363	0.1595	0.1627	0.1483
16	32	0.7601	0.1988	0.2053	0.1998
32	32	1.2614	0.3025	0.3262	0.3118

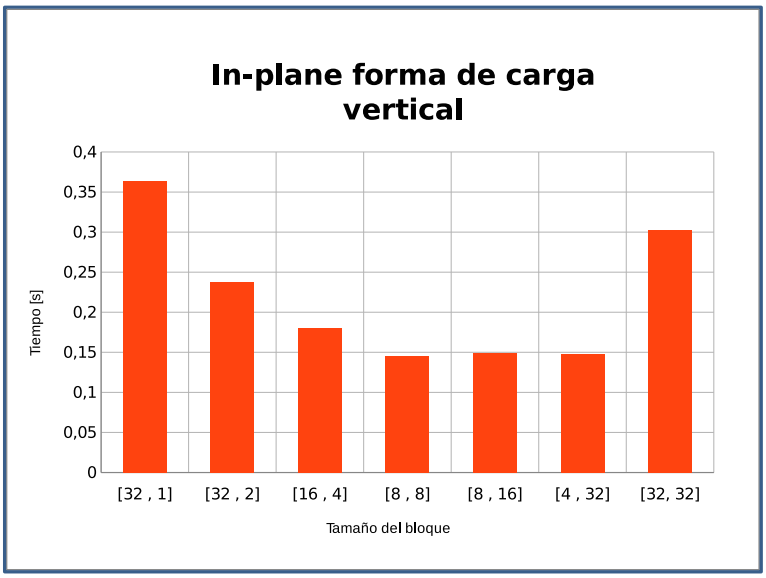
Cabe aclarar que para cada medida de tiempo se hicieron 500 tomas de tiempo y se verificó su distribución para encontrar la medida más probable.
De los datos de las anteriores tablas se obtienen las siguientes Figuras, las cuales muestran la tendencia de los resultados obtenidos en las diferentes implementaciones.

Figura 20: In-plane clásica, Tx, Tz, tiempo ejecución.



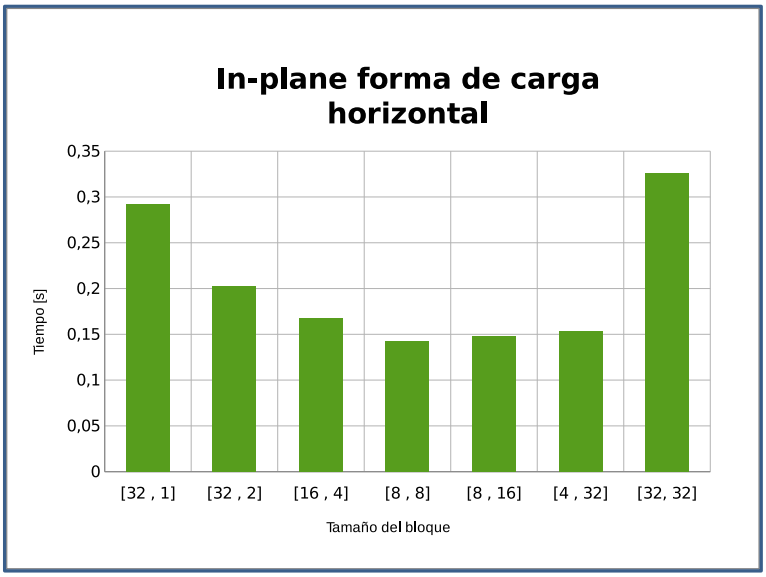
En la Figura 20 se muestra la tendencia que tienen los datos obtenidos en la implementación de la forma de carga clásica donde se puede visualizar que el menor tiempo se da cuando el tamaño del bloque es [8,1,8].

Figura 21: In-plane vertical, Tx, Tz, tiempo ejecución.



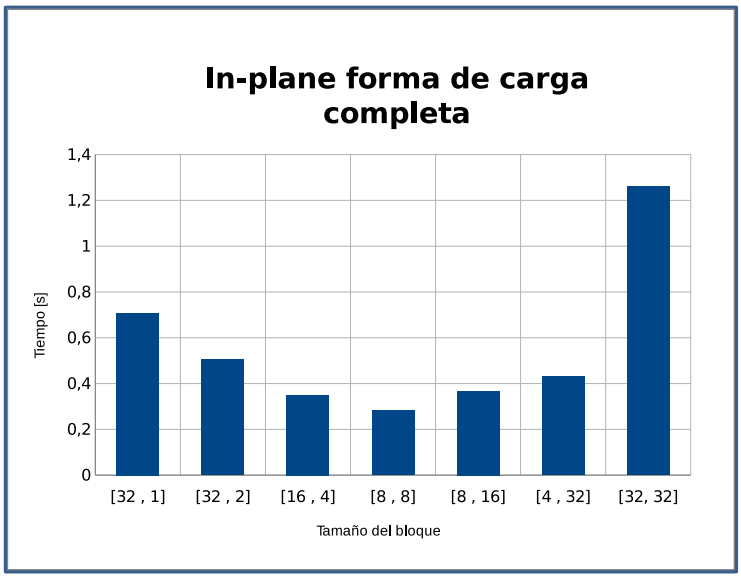
En la Figura 21 se muestra la tendencia que tienen los datos obtenidos en la implementación de la forma de carga vertical donde se puede visualizar que el menor tiempo se obtiene cuando el tamaño del bloque es [8,1,8] al igual que para la implementación anterior.

Figura 22: In-plane horizontal, Tx, Tz, tiempo ejecución.



En la Figura 22 se ve la tendencia que tienen los datos obtenidos en la implementación de la forma de carga horizontal, se puede ver que para el tamaño de bloque [8,1,8] se obtiene el menor tiempo.

Figura 23: In-plane completa, Tx, Tz, tiempo ejecución.



En la Figura 23 se ve la tendencia que tienen los datos obtenidos en la implementación de la forma de carga horizontal donde el menor tiempo de ejecución se da para el tamaño de bloque [8,1,8], que puede notar que esto se da en todas las implementaciones, si se retoma la Tabla 7 se puede notar que el menor tiempo de ejecución si comparamos las diferentes implementaciones, se da para la implementación de la forma de carga clásica.

En el siguiente capítulo se presentará el modelo analítico propuesto por Tang et al [2013] [9] para predecir el tiempo de ejecución de una implementación In-plane de carga completa. Además, se realizará una comparación entre los tiempos estimados por el modelo y los resultados de la implementación In-plane presentados en este capítulo.

Capítulo 4

MODELO IN-PLANE CARGA COMPLETA

4.1. DESCRIPCIÓN DEL MODELO

El artículo presentado por Tang et al [2013] muestra un modelo que determina la eficiencia del cálculo de una tajada en la ejecución de la implementación de un *stencil* por medio del método In-plane carga completa [9].

Las ecuaciones que describen el modelo se presentan a continuación. El primer valor que es necesario determinar es el número total de bloques que contiene cada plano o tajada del cubo y esta descrito por la ecuación 4.1, la cual relaciona el tamaño del plano ($L_x * L_z$) con el tamaño de cada uno de los bloques multiplicado por un escalar ($(T_x * R_x) * (T_z * R_z)$), R_x y R_z se usan para mejorar el tiempo de paralelismo aumentando el área de la cuadrícula que cada bloque de *threads* calcula en ambas dimensiones.

$$Blks = \frac{L_x * L_z}{(T_x * R_x) * (T_z * R_z)} \tag{4.1}$$

La ecuación 4.2 representa el número de bloques activos que son los bloques que pueden estar realizando una acción al mismo tiempo en cada uno de los SMs, se calcula relacionando el mínimo número de recursos de la GPU con los recursos usados por el *kernel*.

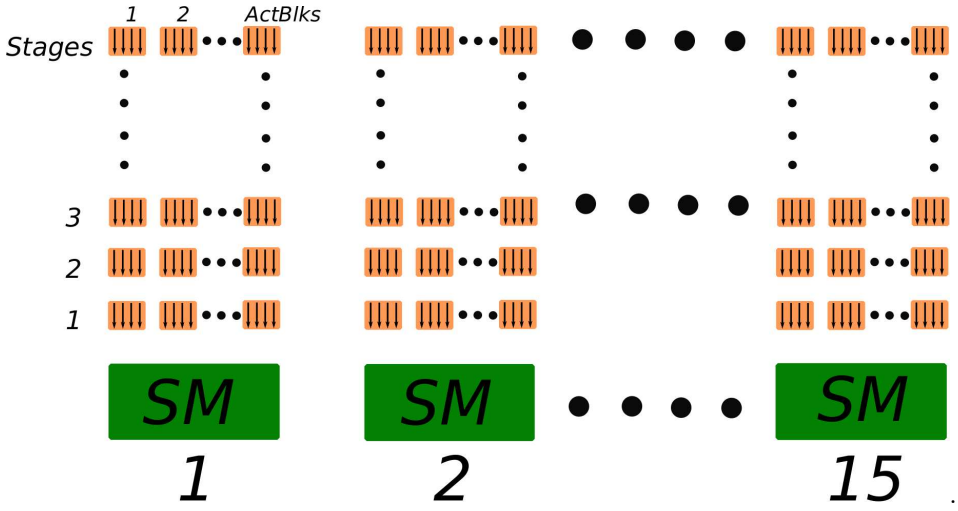
$$ActBlks = min(\lfloor\frac{Reg}{K_R}\rfloor, \lfloor\frac{Smem}{K_S}\rfloor, \lfloor\frac{WarpSM}{WarpBlk}\rfloor, BlkSM) \tag{4.2}$$

Stages es el número de etapas necesarias para que todos los SMs en la GPU pueden ejecutar el número total de bloques que contiene cada plano, relaciona el número de bloques que contiene cada tajada

del cubo con el número de SMs en la GPU y el número de bloques activos en cada SM. El cálculo de los *stages* se puede observar en la ecuación 4.3. En la Figura 24 se muestra una representación de los *stages* y los bloques activos.

$$Stages = \lceil \frac{Blks}{SM * ActBlks} \rceil \tag{4.3}$$

Figura 24: Representación gráfica de los bloques activos ($ActBlks$) y de los $Stages$.



El número de bloques activos por SM para el último *stage* se calcula con la ecuación 4.4. Esta ecuación relaciona el número total de *stages* con el número de bloques que contiene cada tajada y el número de SMs en la GPU.

$$RemBlks = \lceil \frac{Blks - (Stages - 1) * ActBlks * SM}{SM} \rceil \tag{4.4}$$

La ecuación 4.5 busca determinar el tiempo requerido para que un bloque de *threads* acceda a los datos de la memoria global, relaciona la latencia global con la frecuencia del reloj de la GPU y el número total de bytes leídos y escritos para cada plano del *stencil* con el ancho de banda por SM.

$$T_m = \frac{Lat}{Clock} + \frac{BytesBlk}{BW_{SM}} \tag{4.5}$$

El tiempo de cómputo se ve representado por la ecuación 4.6, la cual determina el tiempo requerido para que un bloque de *threads* finalice el cálculo del *stencil* y relaciona los bloques activos en un SM, el número de *warps* por bloque y los *ops* con la frecuencia del reloj.

$$T_c = \frac{ActBlks * Ops * R_x * RY * WarpBlk}{Clock} \tag{4.6}$$

La ecuación 4.7 determina el tiempo por *stage* (i.e. el tiempo que se demora en ejecutar los bloques activos en los SMs), relacionando el tiempo de cómputo T_c y el tiempo de memoria T_m con la función f(arg). La función f(arg) representa el traslape entre el tiempo de ejecución y las lecturas de memoria que existen dento de un SM [9]. Cabe resaltar que esta función varia entre uno y el argumento de dicha función y además el artículo propuesto por Tang et al [2013] no da una referencia de como se calcula el valor de dicha función.

$$T_s = f(ActBlks) * T_m + ActBlks * T_c \tag{4.7}$$

T_l determina el tiempo que se demora el último *stage* en ejecutar los bloques activos en los SMs que estén siendo utilizados, debido a que puede ser que todos los SMs no se ocupen. Este valor se calcula con la ecuación 4.8 la cual relaciona las ecuaciones relaciona el tiempo de memoria T_m y el tiempo de cómputo T_c con el número de bloques activos para el último *stage* y la función $f(arg)$.

$$T_l = f(RemBlks) * T_m + RemBlks * T_c \tag{4.8}$$

La ecuación 4.9 calcula el rendimiento y relaciona el número de bloques para calcular un plano con el tiempo necesario para calcular estos puntos, esta ecuación se da en $Mpoints/s$.

$$Perf = \frac{L_x * L_z}{T_s * (Stages - 1) + T_l} \tag{4.9}$$

Las ecuaciones anteriormente mostradas en este capítulo son las ecuaciones usadas en el modelo presentado en el artículo [9], basándose en ellas se puede proponer la ecuación 4.10 con la cual se determina el tiempo total de ejecución, ya que los tiempos hallados anteriormente son los tiempos de ejecución de una sola tajada.
La ecuación 4.10 consta de el tiempo de ejecución de una tajada multiplicado por el número de tajadas que contiene el modelo.

$$T_{total} = (T_s * (Stages - 1) + T_l) * L_y \tag{4.10}$$

En la Tabla 10 se realiza una breve descripción de los parámetros usados en el modelo.

Tabla 10: Parámetros utilizados en el modelo propuesto por Tang et al [2013].

Parámetros	Descripción
L_x	Tamaño del cubo en la dirección X.
L_y	Tamaño del cubo en la dirección Y.
L_z	Tamaño del cubo en la dirección Z.
T_x	Tamaño del bloque en la dirección X.
T_z	Tamaño del bloque en la dirección Z.
R_x	Factor de escala en la dirección X.
R_z	Factor de escala en la dirección Z.
SM	Número de SMs en una GPU.
$Smem$	Máximo memoria compartida usada por SM.
Reg	Número máximo de registros por SM.
BW_{SM}	Ancho de banda de la memoria global fuera del *chip* por SM y esta dado por BW/SM.
BLK_{SM}	Número máximo de bloques de *threads* por SM.
$WarpSM$	Máximo número de *warps* por SM.
K_R	Número de registros utilizados por el *kernel*.
K_S	Cantidad de memoria compartida utilizada por el *kernel*.
$WarpBlk$	Número de *warps* en un bloque de *threads*.
Lat	Latencia global en ciclos.
$BytesBlk$	Número total de bytes leídos y escritos para cada plano del *stencil*.
Ops	Número de $flops$ requeridos para un orden particular del *stencil*.
$Clock$	Frecuencia del reloj en la GPU.

4.2. PREDICCIÓN TIEMPO DE EJECUCIÓN

Para la validación del modelo propuesto, primero se calculan los valores de los parámetros usados en el modelo, luego se evalúan estos valores en las ecuaciones propuestas en el modelo, después se determina el tiempo total de ejecución con la ecuación 4.10 y se compara con los tiempos obtenidos de la implementación. Cabe aclarar que no se compara la eficiencia ya que no es el parámetro objetivo a comparar.

Algunos de valores de los parámetros descritos en la Tabla 10 son tomados de las especificaciones de la

GPU Tesla K40 mostrados en la Tabla 3, a continuación se presentará la explicación de cómo realizar el cálculo de los demás parámetros.

Los valores de $L_x,\ L_z,\ L_y,\ T_x,\ T_z$ son valores asignados en la implementación.

En este caso las variables R_x y R_z se usan como valores constantes de uno ya que lo que se quiere es determinar el tiempo de ejecución.

La variable BW_{SM} se calcula con el ancho de banda de la memoria global fuera del *chip* que se muestra en la Tabla 3 dividido en el número total de SMs también descrito en el misma Tabla.

El K_R se calcula multiplicando el número de registros (los cuales se restringen en la implementación a 31 registros) por el tamaño del bloque en X y en Z ($T_x * T_z$).

El K_S se calcula con la ecuación 4.11, en este caso como la asignación de la memoria compartida se realizó de forma dinámica, el valor de K_S se cálculo y asignó en la implementación a la hora de lanzar el *kernel*.

$$K_S = ((T_x * 2 * r\_ord) + (T_z * 2 * r\_ord) + (2 * r\_ord * T_x * T_z)) * 4 \tag{4.11}$$

El valor de $WarpBlk$ se calcula con el número de *threads* por bloque sobre el número máximo de *threads* por $Warp$ (Tabla 3).

Existen dos valores de latencia, la latencia de escritura y la latencia de lectura, Lat se tomó como el peor de los casos entre las dos latencias, en este caso el peor de los casos es la latencia de lectura estas se muestran en [10].

Para hallar el valor de $BytesBlk$ se debe calcular el número de bytes leídos y escritos, estos se calculan con la ecuación 4.11. El número de bytes leídos es igual al tamaño del bloque junto con los *halos* en X por el tamaño del bloque junto con los *halos* en Z, el número de bytes escritos es igual al tamaño del bloque en la dirección X por el tamaño del bloque en la dirección Z. El valor final de la ecuación es igual a la suma de estos dos valores multiplicado todo por cuatro.

$$BytesBlk = (((T_x * 2 * r\_ord) * (T_z * 2 * r\_ord)) + (T_x * T_z)) * 4 \tag{4.12}$$

La variable Ops se obtiene del artículo propuesto por Tang et al [2013] [9].

Ahora se presenta la aplicación del modelo para un valor específico de tamaño de bloque ($T_x = 8$ y $T_z = 8$), en la Tabla 11 se muestran los parámetros calculados para este ejemplo.

Tabla 11: Parámetros calculados para el ejemplo $T_x = 8$ y $T_z = 8$.

Parámetros	Valor
L_x	512
L_z	512
BW_{SM}	(288,384 GB/s)/15 = 19,2256 GB/s
K_R	31*8*8 = 1984
K_S	((8+8)*(8+8)+(8*(8*8)))*4 = 3072
$WarpBlk$	(8*8)/32=2
Lat	430
$BytesBlk$	(((8+8)*(8+8))*(8*8))*4 = 1280
Ops	33

Después de hallar el valor de los diferentes parámetros usados en el modelo para este ejemplo se pasa a evaluar el modelo, se remplazan los valores de los parámetros en las ecuaciones y se calcula el valor de cada una de ellas. Se toman cuatro cifras significativas para los diferentes cálculos.

También cabe resaltar que el T_s y el T_l se toman para el peor de los casos ($f(arg) = arg$) debido a que no se conoce la función para calcular el valor de $f(arg)$.

$$Blks = \frac{512 * 512}{(8 * 1) * (8 * 1)} = 4096 \tag{4.13}$$

$$ActBlks = min(\lfloor\frac{64k}{1984}\rfloor, \lfloor\frac{49152}{3072}\rfloor, \lfloor\frac{64}{2}\rfloor, 16) = 16 \tag{4.14}$$

$$Stages = \lceil\frac{4096}{15 * 16}\rceil = 18 \tag{4.15}$$

$$RemBlks = \lceil\frac{4096 - (18 - 1) * 16 * 15}{15}\rceil = 2 \tag{4.16}$$

$$T_m = \frac{430}{745\ MHz} + \frac{1280\ B}{19,2256\ GB/s} = 0,6436\ us \tag{4.17}$$

$$T_c = \frac{16 * 33 * 1 * 1 * 2}{745\ MHz} = 1,4174\ us \tag{4.18}$$

$$\begin{aligned} &1.\ T_s = f(16) * 0,6436\ us + 16 * 1,4174\ us \\ &2.\ T_{smax} = 16 * 0,6436\ us + 16 * 1,4174\ us = 32,976\ us \end{aligned} \tag{4.19}$$

$$\begin{aligned} &1.\ T_l = f(2) * 0,6436\ us + 2 * 1,4174\ us \\ &2.\ T_{lmax} = 2 * 0,6436\ us + 2 * 1,4174\ us\ =\ 4,122\ us \end{aligned} \tag{4.20}$$

$$T_{total} = (32,976\ us * (18-1) + 4,122\ us) * 256\ =\ 0,1446\ s \tag{4.21}$$

Ahora replicando los cálculos anteriores se puede determinar los tiempos de ejecución para diferentes configuraciones y compararlos con los tiempo obtenidos en las implementaciones, esto se muestra en la Tabla 12. Además cabe resaltar que la comparación se hace a través de la ecuación 4.22.

$$\frac{|T_{implementación} - T_{modelo}|}{T_{implementación}} * 100 \tag{4.22}$$

Tabla 12: Comparación de tiempos obtenidos con el modelo contra tiempo obtenidos en la implementación.

T_x	T_z	Tiempo de la implementación [s]	Tiempo del modelo [s]	Error %
32	1	0.7081	0.1917	72.9276
32	2	0.5053	0.1405	72.1947
16	4	0.3498	0.1450	58.5477
8	8	0.2836	0.1446	49.0127
4	16	0.3387	0.1450	57.1893
4	32	0.4293	0.0803	81.2951
32	32	1.2614	0.0117	99,0725

De los resultados anteriores se puede determinar que el modelo no se ajusta a la implementación debido a que se obtienen errores mayores a un 40 %, esto puede ser porque el modelo no contempla el efecto que genera la forma como se almacenan y se acceden a los datos en memoria compartida. Otra razón es que el artículo no presenta muchos detalles de la implementación. Cabe aclarar que se intentó comunicar con los autores pero no se logró tal comunicación.

Capítulo 5

DISCUSIÓN Y CONCLUSIONES

5.1. DISCUSIÓN

El objetivo principal de este proyecto fue predecir el tiempo de ejecución por medio del modelo propuesto por Tang et al [2013], contrastándolo con el tiempo obtenido por medio de la implementación del método In-plane forma de carga completa en una GPU. Para ello se realizaron las implementaciones de la solución de la ecuación de onda acústica 3D en las cuatro formas de carga que presenta el artículo, luego se compararon las diferentes formas de carga y los tiempos de ejecución que se obtuvieron para cada implementación. Comparando el tiempo de ejecución de cada una de ellas se obtuvieron menores tiempos para la implementación del método In-plane forma de carga clásica. Después se realizaron los análisis de cada uno de los parámetros y ecuaciones usadas en el modelo propuesto, encontrando que algunos parámetros no estaban lo suficientemente especificados, lo que llevó ha asumir la forma de calcular estos valores. También se analizó que el modelo solo estaba propuesto para el método de carga In-plane completo y para determinar el valor de ejecución del cálculo de una sola tajada o plano del modelo, así que se propuso una ecuación donde se calcula el tiempo de ejecución del cubo completo. Por último, se predijo el tiempo de ejecución del cubo completo usando el modelo y se comparó este tiempo con el tiempo obtenido por la implementación dando como resultados errores de más del 49 %, debido a que no se tienen conocimientos a cerca de como realizo la implementación por Tang et al [2013] .

5.2. CONCLUSIONES

- Las formas presentadas en el artículo [9] tienen ventajas frente a una solución común debido a que se pueden usar tamaños de datos más grandes con menos recursos.
- La diferencia fundamental entre cada una de las formas de carga radica en la forma que se realiza el indexado de cada elemento en memoria compartida y la forma como se cargan dichos

elementos.

- De los resultados de los tiempos de ejecución en la comparación de las diferentes implementaciones, se obtuvo como resultado que el menor tiempo de ejecución se daba con la implementación del método In-plane carga clásica.
- Solo se puede realizar la comparación del tiempo obtenido para la implementación del método In-plane forma de carga completa ya que modelo propuesto por Tang et al [2013] se basa únicamente en este método de carga.
- El artículo [9] no proporcionaba la información suficiente de ciertos parámetros ya que los dejaba al libre entendimiento de cada persona, lo cual pudo ser un factor del no ajuste del modelo a la implementación.
- De los resultados obtenidos en el capítulo cuatro se puede determinar que el modelo no se ajusta a la implementación debido a que se obtienen errores mayores a un 49 %, esto puede ser a que el modelo no contempla el efecto que genera la forma como se almacenan y se acceden a los datos en memoria compartida. Otra razón es que el artículo no presenta muchos detalles de la implementación.

REFERENCIAS

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

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

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

[4] HERNÁNDEZ, Francisco Javier, EJEMPLOS DE PROGRAMACIÓN EN CUDA. Guanajuato, Gto. Noviembre de 2012.

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

[6] WILT, Nicholas. The CUDA handbook: A comprehensive guide to GPU programming. Pearson Education, 2013.

[7] BORGES, L., and P. Thierry. 3D finite differences on multi-core processors. http://software. intel. com/en-us/articles/3d-finite-differences-on-multi-core-processors) 2.3 (2011): 4-2.

[8] GARCÉS, Bernardo, HERRERO, José Ramón y OTERO, Beatriz. Operación stencil en CUDA. (2011).

[9] TANG, Wai Teng, et al. "Optimizing and auto-tuning iterative stencil loops for GPUs with the In-plane method." Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013.

[10] PARRA, Dorfell. Analytical model to estimate the execution time of a 3D acoustic wave eqution implementation using FDTD in a GPU, Tesis de Maestría. Bucaramanga. Universidad Industrial de Santander, 2016. 90p.

BIBLIOGRAFÍA

BORGES, L., and P. Thierry. 3D finite differences on multi-core processors. http://software. intel. com/en-us/articles/3d-finite-differences-on-multi-core-processors) 2.3 (2011): 4-2.

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

GARCÉS, Bernardo, HERRERO, José Ramón y OTERO, Beatriz. Operación stencil en CUDA. (2011).

HERNÁNDEZ, Francisco Javier, EJEMPLOS DE PROGRAMACIÓN EN CUDA. Guanajuato, Gto. Noviembre de 2012.

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

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

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

PARRA, Dorfell. Analytical model to estimate the execution time of a 3D acoustic wave eqution implementation using FDTD in a GPU, Tesis de Maestría. Bucaramanga. Universidad Industrial de Santander, 2016. 90p.

TANG, Wai Teng, et al. Optimizing and auto-tuning iterative stencil loops for GPUs with the In-plane method. Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013.

WILT, Nicholas. The CUDA handbook: A comprehensive guide to GPU programming. Pearson Education, 2013.

ANEXO A. CÓDIGO IMPLEMENTACIÓN FORMA DE CARGA CLÁSICA

```
__global__ void stencil_in_plane_clasica(int Nx, int Ny, int Nz, float* p, float*
    q, float* c, float dt, float  dx, float dy, float dz, int Ord, float* coef ){
    const int g_idx  =  threadIdx.x + blockIdx.x*blockDim.x;
    const int g_idz  =  threadIdx.z + blockIdx.z*blockDim.z;
    const int idx  =  threadIdx.x;
    const int idz  =  threadIdx.z;
    const int full_bdz_cla = blockDim.z;
    const int full_bdx_cla = blockDim.x;
    const int r_ord = Ord / 2;
    const int id_cla = (idx)*full_bdz_cla + idz;
    const int l_id_cla = idx * blockDim.z + idz;
    int global_id = g_idx*Ny*Nz + g_idz;
    const int size = blockDim.x * blockDim.z;
    const int full_size_cla = full_bdx_cla* full_bdz_cla;
    int idy, i, posicion, temp1, temp1, temp3, haloxi, haloxd, halozt, halozb;
    int x, z , xi, xd, zi, zd, xt, zt, xb, zb;
    float d2x, d2y, d2z;
    bool validw = true;
    if ((g_idx >= Nx) ||  (g_idz >= Nz))
        validw = false;
    extern __shared__ float s[];
    float *actual_slide = s;
    float *q_preview = &actual_slide[full_size_cla];
    float *haloxi_cla= &q_preview[r_ord*size];
    float *haloxd_cla= &haloxi_cla[r_ord*blockDim.z];
    float *halozt_cla= &haloxd_cla[r_ord*blockDim.z];
```

```
26      float *halozb_cla= &halozt_cla[r_ord*blockDim.x];
27      float *lap = &halozb_cla[r_ord*blockDim.x];
28      for(i=0; i<r_ord;i++){
29          q_preview(i) = 0;
30          lap(i) = 0;
31      }
32      temp1 = ((full_size_cla - 1)/ size) + 1;
33      temp2 = ((r_ord*blockDim.z -1)/size) + 1;
34      temp3 = ((r_ord*blockDim.x -1)/size) + 1;
35  for(idy=0; idy<Ny; idy++){
36  __syncthreads();
37     for(i=0; i< temp1; i++){
38         posicion = i*size + (idx*blockDim.z + idz);
39        if(posicion<full_size_cla){
40           x =  (posicion/full_bdz_cla);
41           z =  blockIdx.z*blockDim.z + (posicion - x*full_bdz_cla);
42           x += blockIdx.x*blockDim.x;
43        if(x>=0 && x<Nx && z>=0 && z<Nz){
44          actual_slide[posicion] = Id(q,x,idy,z);
45           }else{
46              actual_slide[posicion] = 0;
47                  } } }
48  __syncthreads();
49     for(i=0; i< temp2; i++){
50         haloxi = i*size + (idx*blockDim.z + idz);
51         haloxd = i*size + (idx*blockDim.z + idz);
52        if(haloxi < r_ord*bdx ){
53           xi =  (haloxi/blockDim.z);
54           zi =  blockIdx.z*blockDim.z + (haloxi - xi*blockDim.z);
55           xi += blockIdx.x*blockDim.x - r_ord;
56           xd =  (haloxd/blockDim.z);
57           zd =  blockIdx.z*blockDim.z + (haloxd - xd*blockDim.z);
58           xd += blockIdx.x*blockDim.x + blockDim.x;
59        if(xi>=0 && xi<Nx && zi>=0 && zi<Nz){
60          haloxi_cla[haloxi] = Id(q,xi,idy,zi);
61           }else{
62              haloxi_cla[haloxi] = 0;
63                  }
64        if(xd>=0 && xd<Nx && zd>=0 && zd<Nz){
65          haloxd_cla[haloxd] = Id(q,xd,idy,zd);
```

```
66          }else{
67             haloxd_cla[haloxd] = 0;
68                 }  } }
69 __syncthreads();
70    for(i=0; i< temp3; i++){
71        halozt = i*size + (idx*blockDim.z + idz);
72        halozb = i*size + (idx*blockDim.z + idz);
73       if(halozt < r_ord*bdz){
74          xt =  (halozt/r_ord);
75          zt =  blockIdx.z*blockDim.z + (halozt - xt*r_ord)- r_ord;
76          xt += blockIdx.x*blockDim.x;
77          xb =  (halozb/r_ord);
78          zb =  blockIdx.z*blockDim.z + (halozb - xb*r_ord)+ blockDim.z;
79          xb += blockIdx.x*blockDim.x;
80       if(xt>=0 && xt<Nx && zt>=0 && zt<Nz){
81         halozt_cla[halozt] = Id(q,xt,idy,zt);
82          }else{
83             halozt_cla[halozt] = 0;
84                 }
85       if(xb>=0 && xb<Nx && zb>=0 && zb<Nz){
86         halozb_cla[halozb] = Id(q,xb,idy,zb);
87          }else{
88             halozb_cla[halozb] = 0;
89                 } } }
90 __syncthreads();
91   if(validw){
92       d2x = 0; d2y = 0; d2z = 0;
93    for(i = 0; i<=r_ord; i++){
94       if(i-bdx+idx>=0){
95         d2x += coef[i] * haloxd_cla[idz+(i-blockDim.x+idx)*blockDim.z];
96            }else{
97             d2x += coef[i] * actual_slide[id+i*full_bdz_cla];
98                 }
99       if(i-bdz+idz>=0){
100        d2z += coef[i] * halozb_cla[idx*r_ord+(i-blockDim.z+idz)];
101           }else{
102            d2z += coef[i] * actual_slide[id+i];
103                }
104         }
105   for(i = 1; i<=r_ord; i++){
```

```
106       if(idx-i<0){
107         d2x += coef[i] * haloxi_cla[idz+(r_ord+idx-i)*blockDim.z];
108            }else{
109             d2x += coef[i] * actual_slide[id-i*full_bdz_cla];
110                 }
111       if(idz-i<0){
112         d2z += coef[i] * halozt_cla[idx*r_ord+(r_ord+idz-i)];
113            }else{
114             d2z += coef[i] * actual_slide[id-i];
115                 }
116          }
117       d2y += coef[0] * actual_slide[id];
118       for(i = 0; i<r_ord; i++){
119         d2y += coef[i+1] * q_preview(i);
120            }
121       for(i = 0; i<r_ord; i++){
122         lap(i) += (coef[i+1] * actual_slide[id])/dy/dy;
123            }
124       if(idy>=r_ord){
125         p[global_id]= -p[global_id] + 2*q_preview(r_ord-1) + (lap(r_ord-1) * dt*dt
              *c[global_id]*c[global_id]);
126         global_id += Nz;
127            }
128       for(i=r_ord-1; i>0; i--){
129         q_preview(i) = q_preview(i-1);
130         lap(i) = lap(i-1);
131            }
132       q_preview(0) = actual_slide[id];
133       lap(0) = d2x/dx/dx + d2y/dy/dy + d2z/dz/dz;
134         } }
135       for (i=r_ord-1; i>=0; i--){
136          if (validw)
137              p[global_id]=-p[global_id] + 2*q_preview(i) +(lap(i) * dt*dt*c[
                   global_id]*c[global_id]);
138         global_id += Nz;
139     } }
```

ANEXO B. CÓDIGO IMPLEMENTACIÓN FORMA DE CARGA VERTICAL

```
__global__ void stencil_in_plane_vertical(int Nx, int Ny, int Nz, float* p,  float
    * q, float* c, float dt, float  dx, float dy, float dz, int Ord, float* coef){
    const int g_idx  =  threadIdx.x + blockIdx.x*blockDim.x;
    const int g_idz  =  threadIdx.z + blockIdx.z*blockDim.z;
    const int idx  =  threadIdx.x;
    const int idz  =  threadIdx.z;
    int bdx = blockDim.x;
    int bdz = blockDim.z;
    const int full_bdz_ver = blockDim.z + Ord;
    const int full_bdx_ver = blockDim.x;
    const int r_ord = Ord/2;
    const int id_ver = (idx)*full_bdz_ver + idz+r_ord;
    const int l_id_ver = idx * blockDim.z + idz;
    int global_id = g_idx*Ny*Nz + g_idz;
    const int size = blockDim.x * blockDim.z;
    const int full_size_ver = full_bdx_ver * full_bdz_ver;
    int idy, i, posicion, temp1, temp2, haloxi, haloxd;
    int x, z , xi, xd, zi, zd;
    float d2x, d2y, d2z;
    bool validw = true;
    if ((g_idx >= Nx) ||  (g_idz >= Nz))
        validw = false;
    extern __shared__ float s[];
    float *actual_slide = s;
    float *q_preview = &actual_slide[full_size_ver];
    float *haloxi_ver= &q_preview[r_ord*size];
```

```
26      float *haloxd_ver= &haloxi_ver[r_ord*blockDim.z];
27      float *lap = &haloxd_ver[r_ord*blockDim.z];
28      for(i=0; i<r_ord;i++){
29          q_preview(i) = 0;
30          lap(i) = 0;
31      }
32      temp1 = ((full_size_ver - 1)/ size) + 1;
33      temp2 = ((r_ord*blockDim.z -1)/size) + 1;
34      for(idy=0; idy<Ny; idy++){
35  __syncthreads();
36      for(i=0; i< temp1; i++){
37        posicion = i*size + (idx*blockDim.z + idz);
38         if(posicion<full_size_ver){
39            x =  (posicion/full_bdz_ver);
40            z =  blockIdx.z*blockDim.z + (posicion - x*full_bdz_ver) - r_ord;
41            x += blockIdx.x*blockDim.x;
42             if(x>=0 && x<Nx && z>=0 && z<Nz){
43                actual_slide[posicion] = Id(q,x,idy,z);
44                  }
45                else{
46                  actual_slide[posicion] = 0;
47                } } }
48  __syncthreads();
49     for(i=0; i< temp2; i++){
50        haloxi = i*size + (idx*blockDim.z + idz);
51        haloxd = i*size + (idx*blockDim.z + idz);
52           if(haloxi < r_ord*bdx){
53              xi =  (haloxi/blockDim.z);
54              zi =  blockIdx.z*blockDim.z + (haloxi - xi*blockDim.z);
55              xi += blockIdx.x*blockDim.x - r_ord;
56              xd =  (haloxd/blockDim.z);
57              zd =  blockIdx.z*blockDim.z + (haloxd - xd*blockDim.z);
58              xd += blockIdx.x*blockDim.x + blockDim.x;
59                if(xi>=0 && xi<Nx && zi>=0 && zi<Nz){
60                  haloxi_ver[haloxi] = Id(q,xi,idy,zi);
61                    }else{
62                         haloxi_ver[haloxi] = 0;
63                    }
64               if(xd>=0 && xd<Nx && zd>=0 && zd<Nz){
65                 haloxd_ver[haloxd] = Id(q,xd,idy,zd);
```

```
                }else{
                    haloxd_ver[haloxd] = 0;
                     } } }
__syncthreads();
   if(validw){
      d2x = 0; d2y = 0; d2z = 0;
        for(i = 0; i<=r_ord; i++){
            if(i-bdx+idx>=0){
               d2x += coef[i] * haloxd_ver[idz+(i-blockDim.x+idx)*blockDim.z];
               }
               else{
                 d2x += coef[i] * actual_slide[id+i*full_bdz_ver];
                   }
               d2z += coef[i] * actual_slide[id+i];
             }
        for(i = 1; i<=r_ord; i++){
            if(idx-i<0){
               d2x += coef[i] * haloxi_ver[idz+(r_ord+idx-i)*blockDim.z]; // -x
                 }
                 else{
                   d2x += coef[i] * actual_slide[id-i*full_bdz_ver]; // -x
                     }
               d2z += coef[i] * actual_slide[id-i]; // -z
              }
            d2y += coef[0] * actual_slide[id];
            for(i = 0; i<r_ord; i++){
                d2y += coef[i+1] * q_preview(i);
            }
            for(i = 0; i<r_ord; i++){
                lap(i) += (coef[i+1] * actual_slide[id])/dy/dy;
            }
            if(idy>=r_ord){
                p[global_id]= -p[global_id] + 2*q_preview(r_ord-1) +\
                            (lap(r_ord-1) * dt*dt*c[global_id]*c[global_id]);
                global_id += Nz;
            }
            for(i=r_ord-1; i>0; i--){
                q_preview(i) = q_preview(i-1);
                lap(i) = lap(i-1);
            }
```

```
106             q_preview(0) = actual_slide[id];
107             lap(0) = d2x/dx/dx + d2y/dy/dy + d2z/dz/dz;
108         } }
109     for (i=r_ord-1; i>=0; i--){
110         if (validw)
111             p[global_id]=-p[global_id] + 2*q_preview(i) +\
112                         (lap(i) * dt*dt*c[global_id]*c[global_id]);
113         global_id += Nz;
114     } }
```

ANEXO C. CÓDIGO IMPLEMENTACIÓN FORMA DE CARGA HORIZONTAL

```
1  __global__ void stencil_in_plane_horizontal (int Nx, int Ny, int Nz, float* p,
       float* q, float* c, float dt, float  dx, float dy, float dz, int Ord, float*
       coef){
2      const int g_idx  =  threadIdx.x + blockIdx.x*blockDim.x;
3      const int g_idz  =  threadIdx.z + blockIdx.z*blockDim.z;
4      const int idx  =  threadIdx.x;
5      const int idz  =  threadIdx.z;
6      const int full_bdz_hor= blockDim.z;
7      const int full_bdx_hor = blockDim.x + Ord;
8      const int r_ord = Ord/2;
9      const int id_hor = (idx+r_ord)*full_bdz_hor+ idz;
10     const int l_id_hor = idx * blockDim.z + idz;
11     int global_id = g_idx*Ny*Nz + g_idz;
12     const int size = blockDim.x * blockDim.z;
13     const int full_size_hor = full_bdx_hor* full_bdz_hor;
14     int idy, i, posicion,temp1, temp3, halozt, halozb;
15     int x, z , xt, xb, zt, zb;
16     float d2x, d2y, d2z;
17
18     bool validw = true;
19     if ((g_idx >= Nx) ||  (g_idz >= Nz))
20         validw = false;
21
22     extern __shared__ float s[];
23     float *actual_slide = s;
24     float *q_preview = &actual_slide[full_size_hor]; ]
```

```
25      float *halozt_hor= &q_preview[r_ord*size];
26      float *halozb_hor= &halozt_hor[r_ord*blockDim.x];
27      float *lap = &halozb_hor[r_ord*blockDim.x];
28      for(i=0; i<r_ord;i++){
29          q_preview(i) = 0;
30          lap(i) = 0;
31      }
32      temp1 = ((full_size_hor - 1)/ size) + 1;
33      temp3 = ((r_ord*blockDim.x -1)/size) + 1;
34  for(idy=0; idy<Ny; idy++){
35   __syncthreads();
36    for(i=0; i< temp1; i++){
37       posicion = i*size + (idx*blockDim.z + idz);
38         if(posicion<full_size_hor){
39           x =  (posicion/full_bdz);
40           z =  blockIdx.z*blockDim.z + (posicion - x*full_bdz);
41           x += blockIdx.x*blockDim.x - r_ord;
42        if(x>=0 && x<Nx && z>=0 && z<Nz){
43            actual_slide[posicion] = Id(q,x,idy,z);
44          }else{
45             actual_slide[posicion] = 0;
46              } } }
47   __syncthreads();
48        for(i=0; i< temp3; i++){
49            halozt = i*size + (idx*blockDim.z + idz);
50            halozb = i*size + (idx*blockDim.z + idz);
51            if(halozt < r_ord*bdz){
52                 xt =  (halozt/r_ord);
53                 zt =  blockIdx.z*blockDim.z + (halozt - xt*r_ord)- r_ord;
54                 xt += blockIdx.x*blockDim.x;
55                 xb =  (halozb/r_ord);
56                 zb =  blockIdx.z*blockDim.z + (halozb - xb*r_ord)+ blockDim.z;
57                 xb += blockIdx.x*blockDim.x;
58                 if(xt>=0 && xt<Nx && zt>=0 && zt<Nz){
59                         halozt_hor[halozt] = Id(q,xt,idy,zt);
60                 }else{
61                         halozt_hor[halozt] = 0;
62                         }
63                   if(xb>=0 && xb<Nx && zb>=0 && zb<Nz){
64                         halozb_hor[halozb] = Id(q,xb,idy,zb);
```

```
65                    }else{
66                         halozb_hor[halozb] = 0;
67                    } } }
68   __syncthreads();
69     if(validw){
70        d2x = 0; d2y = 0; d2z = 0;
71           for(i = 0; i<=r_ord; i++){
72              d2x += coef[i] * actual_slide[id+i*full_bdz];
73                 if(i-bdz+idz>=0){
74                   d2z += coef[i] * halozb_hor[idx*r_ord+(i-blockDim.z+idz)];
75                   }else{
76              d2z += coef[i] * actual_slide[id+i];
77                 }
78       }
79           for(i = 1; i<=r_ord; i++){
80               d2x += coef[i] * actual_slide[id-i*full_bdz];
81                 if(idz-i<0){
82                    d2z += coef[i] * halozt_hor[idx*r_ord+(r_ord+idz-i)];
83                   }else{
84                         d2z += coef[i] * actual_slide[id-i];
85                 }
86       }
87             d2y += coef[0] * actual_slide[id];
88               for(i = 0; i<r_ord; i++){
89                 d2y += coef[i+1] * q_preview(i);
90                }
91             for(i = 0; i<r_ord; i++){
92                 lap(i) += (coef[i+1] * actual_slide[id])/dy/dy;
93               }
94             if(idy>=r_ord){
95                 p[global_id]= -p[global_id] + 2*q_preview(r_ord-1) + (lap(r_ord-1)
                        * dt*dt*c[global_id]*c[global_id]);
96                 global_id += Nz;
97               }
98             for(i=r_ord-1; i>0; i--){
99                 q_preview(i) = q_preview(i-1);
100                lap(i) = lap(i-1);
101            }
102            q_preview(0) = actual_slide[id];
103            lap(0) = d2x/dx/dx + d2y/dy/dy + d2z/dz/dz;
```

```
104          } }
105  for (i=r_ord-1; i>=0; i--){
106     if (validw)
107        p[global_id]=-p[global_id] + 2*q_preview(i) + (lap(i) * dt*dt*c[global_id]*
               c[global_id]);
108        global_id += Nz;
109     } }
```

ANEXO D. CÓDIGO IMPLEMENTACIÓN FORMA DE CARGA COMPLETA

```
1  __global__ void stencil_in_plane_completa (int Nx, int Ny, int Nz, float* p,float*
       q, float* c, float dt, float  dx, float dy, float dz, int Ord, float* coef){
2      const int g_idx  =  threadIdx.x + blockIdx.x*blockDim.x;
3      const int g_idz  =  threadIdx.z + blockIdx.z*blockDim.z;
4      const int idx  =  threadIdx.x;
5      const int idz  =  threadIdx.z;
6      const int full_bdz_com = blockDim.z + Ord;
7      const int full_bdx_com = blockDim.x + Ord;
8      const int r_ord = Ord / 2;
9      const int id_com = (idx+r_ord)*full_bdz + idz+r_ord;
10     const int l_id_com = idx * blockDim.z + idz;
11     int global_id = g_idx*Ny*Nz + g_idz;
12     const int size = blockDim.x * blockDim.z;
13     const int full_size_com = full_bdx_hor* full_bdz_hor;
14     int idy, i, posicion, temp;
15     int x, z;
16     float d2x, d2y, d2z;
17     bool validw = true;
18     if ((g_idx >= Nx) ||  (g_idz >= Nz))
19         validw = false;
20     extern __shared__ float s[]; // size = full_size_com + r_ord + r_ord
21     float *actual_slide = s;
22     float *q_preview = &actual_slide[full_size_com];
23     float *lap = &q_preview[r_ord*size];
24     for(i=0; i<r_ord;i++){
25         q_preview(i) = 0;
```

```
26          lap(i) = 0;
27      }
28      temp = (full_size_com-1 / size)+1;
29      for(idy=0; idy<Ny; idy++){
30  __syncthreads();
31      for(i=0; i< temp; i++){
32          posicion = i*size + (idx*blockDim.z + idz);
33              if(posicion<full_size){
34                 x =  (posicion/full_bdz);
35                 z =  blockIdx.z*blockDim.z + (posicion - x*full_bdz) - r_ord;
36                 x += blockIdx.x*blockDim.x - r_ord;
37                   if(x>=0 && x<Nx && z>=0 && z<Nz){
38                      actual_slide[posicion] = Id(q,x,idy,z);
39                   }else{
40                      actual_slide[posicion] = 0;
41                   } } }
42  __syncthreads();
43    if(validw){
44      d2x = 0; d2y = 0; d2z = 0;
45        for(i = 0; i<=r_ord; i++){
46           d2x += coef[i] * actual_slide[id+i*full_bdz_com];
47           d2z += coef[i] * actual_slide[id+i];
48         }
49        for(i = 1; i<=r_ord; i++){
50           d2x += coef[i] * actual_slide[id-i*full_bdz_com];
51           d2z += coef[i] * actual_slide[id-i];
52         }
53      d2y += coef[0] * actual_slide[id];
54        for(i = 0; i<r_ord; i++){
55           d2y += coef[i+1] * q_preview(i);
56         }
57       for(i = 0; i<r_ord; i++){
58           lap(i) += (coef[i+1] * actual_slide[id])/dy/dy;
59         }
60       if(idy>=r_ord){
61          p[global_id]= -p[global_id] + 2*q_preview(r_ord-1) +(lap(r_ord-1) * dt*dt*
                c[global_id]*c[global_id]);
62          global_id += Nz;
63        }
64       for(i=r_ord-1; i>0; i--){
```

```
65          q_preview(i) = q_preview(i-1);
66          lap(i) = lap(i-1);
67         }
68     q_preview(0) = actual_slide[id];
69     lap(0) = d2x/dx/dx + d2y/dy/dy + d2z/dz/dz;
70         }  }
71       for (i=r_ord-1; i>=0; i--){
72          if (validw)
73             p[global_id]=-p[global_id] + 2*q_preview(i) + (lap(i) * dt*dt*c[
                   global_id]*c[global_id]);
74             global_id += Nz;
75           }}
```