
VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**

**POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN**

# PROGRAMACIÓN DINÁMICA PARALELA EN LAS GPU

# T E S I S

QUE PARA OBTENER EL GRADO DE:

**MAESTRO EN CIENCIAS (COMPUTACIÓN)**

P R E S E N T A:

**MANUEL ALCÁNTARA JUÁREZ**

DIRECTOR DE TESIS:
DR. JOSÉ DAVID FLORES PEÑALOZA, FACULTAD DE CIENCIAS
COTUTOR:
DR. MARIO A. LÓPEZ, UNIVERSITY OF DENVER

MÉXICO, D.F. FEBRERO 2014.

# Agradecimientos


- A mi madre Ma. Teresa Juárez y a mi hermano Daniel Alcántara que son el pilar fundamental en mi vida. Porque siempre han sido la brújala en mi camino, la fuerza para afrontar los obstáculos y la sabiduría para tomar las decisiones correctas.
- A mis tutores el Dr. José David Flores Peñaloza y Dr. Mario A. López, por sus consejos, estímulos, apoyo y amistad brindada durante todo el desarrollo de este trabajo.
- A Violeta Díaz Santos, por ser una de mis mejores amigas, cómplice incondicional en las mejores aventuras, sueños y prospectos de estos últimos cuatro años. Por enseñarme a mirar otro ángulo de la vida que nunca olvidaré.
- A mis amigos Renato Zamudio, Karla Vargas, Norma Trinidad y Armando Vaginas, por hacerme más ameno el sendero brindándome todos esos bellos momentos que hemos pasado juntos, por exhortarme a cumplir siempre mis objetivos e impulsarme a progresar como individuo.
- A CONACYT por haberme apoyado con una beca durante todos mis estudios de maestría.
- A mis sinodales Dr. Fabián García, Dr. Sergio Rajsbaum, Dr. José Galaviz y en especial al Dr. Carlos Velarde por sus comentarios y sugerencias realizadas a este trabajo.
- A todas las personas que directa o indirectamente han hecho posible la realización y culminación de este trabajo.

# Índice general

# Capítulo 1

# Introducción

En los últimos años, el avance que ha tenido la industria de la computación se ha incrementado considerablemente. Tal ha sido el progreso, que hoy en día podemos encontrar procesadores con más de un núcleo y con la capacidad de ejecutar varios miles de millones de operaciones por segundo. También se ha visto una gran proliferación de las unidades de proceso gráfico o GPU por sus siglas en inglés para realizar cómputo de propósito general, ya que estas contienen una gran cantidad de núcleos de procesamiento exclusivamente dedicados a calcular operaciones de punto flotante.

El mayor beneficio de estos sistemas multinúcleo, es la posibilidad de ejecutar simultáneamente dos o más operaciones por diferentes unidades de procesamiento, consiguiendo con ello realizar un trabajo en un menor lapso de tiempo o llevar a cabo más instrucciones en ese mismo período. Sin embargo para poder aprovechar estas características tan importantes que nos ofrecen estos sistemas, es necesario que el programador especifique cuáles son los bloques de código que pueden ejecutarse de manera independiente y a su vez indicar la forma en cómo se combinarán esos resultados.

A menudo la realización de estas tareas suele ser un proceso complicado y a pesar de que se han desarrollado una gran gama de modelos y técnicas, que intentan sobrellevar esta tarea de una manera intuitiva y transparente, sigue siendo necesaria la intervención del programador para que coordine las actividades en las diferentes unidades de procesamiento. Desgraciadamente es demasiado difícil realizar automáticamente el paralelismo y una de las razones de ello, es que para llevar a cabo dicha tarea se tienen que identificar conjuntos de instrucciones que sean independientes entre sí, mediante algún análisis de dependencias, sin embargo muchas de las técnicas que existen actualmente para este propósito, no en todos los casos son correctas o son muy limitadas.

El simple hecho de programar puede considerarse en sí ya una tarea complicada, por lo cual muchas veces el programador deja de lado la labor de paralelizar el código, provocando con esto que no se aprovechen adecuadamente los recursos con los que se cuenta. Es por ello que es necesario ofrecer algunos mecanismos que ayuden al programador a

explotar estas características, sin necesidad de cambiar radicalmente su forma de pensar, con la menor inserción de líneas de código y tratando de evitar lo más posible de que se enfrente a los problemas que surgen cuando se utiliza el cómputo paralelo, como por ejemplo la sincronización o condiciones de carrera. Y es a partir de aquí donde surge el planteamiento de este trabajo.

Como ya se había mencionado con anterioridad, tratar de realizar una automatización del paralelismo en un caso general es muy complicado, por lo cual en este trabajo se acotará el caso de estudio y sólo se enfocará en la programación dinámica, que no es más que un técnica que se utiliza cuando se quiere resolver un problema de optimización, encontrando una posible solución a partir de combinar las soluciones parciales de varios subproblemas. Cuando el programador está utilizando esta técnica, implícitamente define la relación que existe entre los subproblemas, por lo cual con unas pocas líneas de código por parte del programador, se pueden identificar bloques independientes que pueden ser ejecutados simultáneamente, logrando con ello obtener cierto paralelismo y mínimos cambios en el diseño original. A la par se analizará la idea de poder generar una gráfica de dependencias, la cual logre obtener las relaciones de los subproblemas a partir de la definición de los mismos.

Por todo ello el presente trabajo tiene como objetivo principal, el análisis de algunos problemas que se resuelven con programación dinámica, como pauta para la implementación de un marco de trabajo (framework en inglés) que ayude a los programadores a extraer de manera sencilla el paralelismo, explotando las características que en la actualidad ofrecen las GPU, mediante la inserción de pocas líneas de código y tratando de ahorrarle al programador muchos de los detalles técnicos de esta nueva arquitectura.

# Capítulo 2

# Programación dinámica

Uno de los métodos que surgen de manera intuitiva a la hora de resolver un problema, es tratar de descomponerlo en subproblemas más pequeños, de tal manera que sean mucho más fáciles de resolver, para posteriormente combinar los resultados parciales y con ello formar la solución general de nuestro problema principal. Esta es prácticamente la esencia de la programación dinámica, aunque muchas veces se suele confundir con la técnica de *"Divide y vencerás"*[6], ya que son muy parecidas entre sí. Sin embargo la diferencia circunstancial radica en que la programación dinámica se enfoca en resolver problemas cuyos subproblemas no son del todo independientes, es decir el resultado de un subproblema puede depender de la solución de otro. Y es en estos casos en donde la programación dinámica logra obtener un mejor desempeño que la técnica de divide y vencerás, ya que evita que se lleven a cabo cálculos repetidos, guardando para ello las soluciones parciales en una matriz para futuras referencias. [17]

El término programación dinámica originalmente fue introducido por Richard Bellman [3], el cual describe el proceso de resolver problemas donde es necesario encontrar las mejores decisiones una tras de otra. Es por este hecho que es típicamente utilizada en la resolución de problemas de optimización, es decir en problemas en donde se pueden presentar una enorme cantidad de soluciones, cada una de ellas con un valor asociado y lo que se busca es encontrar la solución con un valor óptimo ya sea máximo o mínimo. Esta técnica puede aplicarse a cualquier tipo de problemas siempre y cuando cumplan con el *principio del óptimo* enunciado por Richard Bellman en 1957[4], el cual menciona lo siguiente:

*"Dada una secuencia óptima de decisiones, toda subsecuencia de ella ha de ser también óptima"*

Lo que menciona dicho principio en esencia, es que si de alguna manera se obtuvo una solución óptima para el problema, entonces eso sólo pudo significar que en cada parte en donde se necesitó realizar una decisión, ésta siempre tuvo que ser la mejor entre todo el conjunto de decisiones posibles.

Existen principalmente dos estrategias que se suelen utilizar cuando se está desarrollando la solución a un problema por medio de programación dinámica. La primera es un enfoque *top-down* en la cual el problema principal se descompone en subproblemas más pequeños, los cuales a su vez se van redefiniendo con un mayor detalle hasta que se llega a una especificación lo suficientemente simple para resolverlos sin mayor dificultad. Posteriormente se reconstruye la solución general a partir de estos subproblemas. El segundo es un enfoque que se conoce como *bottom-up*, en él se parte de los problemas más elementales resolviéndolos y enlazándolos para construir el resultado de un problema más grande. El proceso continua hasta que se logre formar la solución del problema general. Escoger el enfoque y diseñar la ecuación funcional de manera eficiente requiere un análisis algorítmico que depende propiamente del problema a resolver. De manera simple pueden utilizarse los siguientes pasos como referencia: [7]:

1. Plantear la solución como una sucesión de decisiones, verificando que cumple con el principio del óptimo.
2. Definir de manera recursiva el valor de una solución óptima.
3. Calcular el valor de la solución óptima apoyándose de una matriz de soluciones parciales, con la finalidad de reutilizar cálculos previos.
4. Construir la solución óptima general haciendo uso de la información previamente calculada.

Como se puede destacar de lo anterior, cuando se está planteando la solución a un problema, este se traduce en definir una función que permita calcular las entradas de una matriz asociada, cuyo tamaño depende del número de subproblemas y su propósito es guardar soluciones parciales. Lo que se espera de esta técnica, es que las llamadas recursivas que realice la función para encontrar una solución óptima, sean simples accesos a localidades de la matriz previamente calculadas. El dominio de la función se encuentra intrínsecamente ligado a los índices y dimensión de la matriz de resultados, con ello en mente se define lo siguiente:

**Definición 2.1.** *Sea*

$$I_{n_1 \times \ldots \times n_k} = \{\bar{x} \in N^k | 0 \leq x_i < n_i,\ \forall i \in [1..k]\}$$

*el* ***dominio acotado de índices k-dimensional****.*

**Definición 2.2.** *Se define la* ***ecuación funcional*** *de un cierto problema, tanto a la matriz asociada como a la función que calcula sus entradas, la cual se representará como:*

$$f_{M_{n_1 \times \ldots n_k}}$$

*donde:*

- *$f$ es la función que calcula la solución de los subproblemas. $f : I_{n_1 \times \ldots \times n_k} \to R$ donde $R$ es un rango arbitrario*

- *$M$ es la matriz asociada*
- *$n_1 \times ... \times n_k$ es la dimensión de la matriz $M$*

A continuación se presentan algunos problemas que se resuelven utilizando la programación dinámica, estos problemas se tomarán como casos de estudio, para posteriormente obtener algunas constantes de paralelización. Debido a que el objetivo de este trabajo no es realizar un análisis minucioso de dichos problemas, muchos de los detalles no se incluyen, como por ejemplo, la comprobación del principio del óptimo.

## 2.1. Coeficientes binomiales

Los coeficientes binomiales o combinaciones son números estudiados en combinatoria, denotados por el símbolo $\binom{n}{m}$ y cuya interpretación corresponde al número de formas en que se puede escoger un conjunto de $m$ elementos a partir de un conjunto dado de $n$ elementos[11].
Lo que se pretende es poder resolver dicho problema pero utilizando para ello la programación dinámica, así que lo primero que se necesita realizar es definir de manera recursiva una solución óptima. El matemático Blaise Pascal realizó una correspondencia entre su triángulo de Pascal y los coeficientes binomiales, lo que permite saber dicha definición recursiva, conocida como identidad de Pascal[29]:

$$\binom{n}{m} = \begin{cases} 1 & \text{si } m = 0 \text{ ó } m = n \\ \binom{n-1}{m-1} + \binom{n-1}{m} & \text{si } 0 < m < n \end{cases} \tag{2.1}$$

Con esta identidad uno se puede dar cuenta de que cada coeficiente binomial puede calcularse como la suma de dos números de la fila anterior, uno directamente encima de él, y el otro justo a la izquierda.

| $M$ | 0 | 1 | 2 | ... | $m-1$ | $m$ |
|---|---|---|---|---|---|---|
| 0 | 1 | | | | | |
| 1 | 1 | 1 | | | | |
| 2 | 1 | 2 | 1 | | | |
| ... | | | | ... | | |
| $n-1$ | | | | | $\binom{n-1}{m-1}$ | $\binom{n-1}{m}$ |
| $n$ | | | | | | $\binom{n}{m}$ |

Tabla 2.1: Dependencias en el problema de los coeficientes binomiales.

De lo anterior se tiene lo necesario para formalizar la ecuación funcional que resuelve el problema de los coeficientes binomiales, que puede expresarse como:

$$BinCoef_{M_{(n+1)\times(m+1)}}(i,j) = \begin{cases} 1 & \text{si } j = 0 \text{ ó } i = j \\ M[i-1,j-1] + M[i-1,j] & \text{si } 0 < j < i \\ 0 & e.o.c. \end{cases} \tag{2.2}$$

Una de las maneras en que se puede ir llenando la matriz es por filas de arriba hacia abajo, esto debido a la configuración de las dependencias que presenta el problema.

Una implementación en pseudocódigo puede ser la siguiente:

```
BinCoef(i, j, M[0..n, 0..m]){
  if j == 0 o i == j then
    M[i, j]:= 1;
  else if j < i then
    M[i, j]:= M[i-1, j-1] + M[i-1, j];
  else
    M[i, j]:= 0;
  end if
}

fillBinCoef(M[0..n, 0..m]){
  for i:= 0 to n do
    for j:= 0 to m do
      BinCoef(i, j, M);
    end for
  end for
}
```

## 2.2. Problema de la mochila (0-1)

Supongamos que tenemos $n$ objetos, $x_1, x_2, .., x_n$ donde $x_i$ tiene asociado un valor $p_i \in \mathbb{R}^+/0$, un peso $w_i \in \mathbb{N}/0$ y tenemos una mochila que puede soportar un peso $W \in \mathbb{N}/0$, lo que se quiere saber es que objetos se deben de meter en la mochila de tal forma que no sobrepasen $W$ y se maximice su valor. En términos formales, lo que se busca es determinar un subconjunto $T \subseteq \{1, 2, \ldots, n\}$ tal que[18]:

$$\sum_{i \in T} p_i \quad \text{sea máxima, sujeto a} \quad \sum_{i \in T} w_i \leq W \tag{2.3}$$

A este problema también se le conoce como *mochila* (0-1) y para hallar su solución primero se crea una matriz $M[0..n, 0..W]$, en donde $M[i, j]$ contendrá el valor máximo que se puede obtener con los primeros $i$ objetos y una capacidad $j$ para la mochila. El valor de la solución general del problema se encuentra en $M[n, W]$.
Para calcular la solución óptima de manera recursiva para $M[i, j]$ se toman las siguientes decisiones:

1. Si el peso del objeto $x_i$ sobrepasa el límite $j$ de la mochila, simplemente no hay que considerarlo y la solución está dada por el óptimo de los primeros $i-1$ objetos con una mochila de capacidad $j$, que por definición se encuentra en $M[i-1, j]$.

2. Si no sucede el caso anterior, entonces se considera el valor óptimo entre agregar o no el objeto. El mejor valor cuando no se considera el objeto se encuentra en

$M[i-1,j]$ por el mismo argumento que el caso anterior.
Por otro lado cuando se agrega el objeto se tiene que calcular la solución óptima de los primeros $i-1$ objetos pero utilizando una mochila de capacidad $j-w_i$ (ya que se está agregando el peso del nuevo objeto) que por definición se encuentra en $M[i-1,j-w_i]$ y a ello sólo hay que sumarle la ganancia del objeto $x_i$, es decir $p_i + M[i-1,j-w_i]$.

| $M$ | 0 | 1 | ... | $j-w_i$ | ... | $j$ | ... | $W$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | | 0 | | 0 | | 0 |
| 1 | 0 | | | | | | | |
| ... | | | ... | | | | | |
| $i-1$ | 0 | | | | ... | | | |
| $i$ | 0 | | | | ... | $M[i,j]$ | | |
| ... | | | | | | | | |
| $n$ | 0 | | | | | | | |

Tabla 2.2: Dependencias en el problema de la mochila (0-1).

De lo anterior se desprende que la ecuación funcional para el problema de la mochila (0-1) se puede definir como:

$$mochila_{M_{(n+1)\times(W+1)}}(i,j) = \begin{cases} 0 & \text{si } i=0 \text{ ó } j=0 \\ M[i-1,j] & \text{si } w_i > j \\ max(M[i-1,j], p_i + M[i-1,j-w_i]) & \text{si } w_i \leq j \end{cases}$$

La forma de cómo se tiene que ir llenando la matriz para obtener el resultado deseado es por filas de arriba hacia abajo.

Una implementación en pseudocódigo es la siguiente:

```
mochila(i, j, M[0..n, 0..W], w[1..n], p[1..n]){
  if i==0 o j==0 then
    M[i, j]:= 0;
  else
    if w[i] > j then
      M[i, j]:= M[i-1, j];
    else
      M[i, j]:= max(M[i-1, j], p[i]+M[i-1, j-w[i]]);
    end if
  end if
}

fillMochila(M[0..n, 0..W], w[1..n], p[1..n]){
  for i:=0 to n do
    for j:=0 to W do
      mochila(i, j, M, w, p);
    end for
  end for
}
```

## 2.3. La subsecuencia común más larga (LCS)

**Definición 2.3.** *Dado un conjunto finito* $\Gamma$*, una* ***secuencia*** *es una disposición finita y ordenada de símbolos pertenecientes a ese conjunto.*
$\Gamma = \{a, g, t, c\}$ *entonces* $\langle a, g, c, g, t, a, g\rangle$ *y* $\langle g, t, c, a, g, a\rangle$ *son secuencias del conjunto* $\Gamma$*.*

**Definición 2.4.** *La* ***longitud de una secuencia*** *es el número de símbolos que la componen.* $|\langle a, g, c, g, t, a, g\rangle| = 7$ *y* $|\langle g, t, c, a, g, a\rangle| = 6$*.*

**Definición 2.5.** *Una* ***subsecuencia*** *de cualquier secuencia* $X$*, es aquella que resulta de eliminar cero o más símbolos de* $X$*.*
$X = \langle a, g, c, g, t, a, g\rangle$ *entonces* $\langle a, c, g, a\rangle$*,* $\langle g, g, t, g\rangle$ *y* $\langle c, t\rangle$*, son posibles subsecuencias de* $X$*.*

**Definición 2.6.** *Dadas dos secuencias* $X$ *y* $Y$*, se dice que* $Z$ *es una* ***subsecuencia común*** *si es subsecuencia de* $X$ *y* $Y$*.*

Lo que se busca es encontrar la subsecuencia común de longitud máxima de dos secuencias. Por ejemplo si $X = \langle a, g, c, g, t, a, g\rangle$ y $Y = \langle g, t, c, a, g, a\rangle$ entonces $Z = \langle g, c, g, a\rangle$ es la subsecuencia común a $X$ y $Y$ más larga con longitud 4, ya que no existe ninguna subsecuencia común a $X$ y $Y$ de longitud 5. [12, 13]

Sea $X = \langle x_1, x_2, \ldots, x_n\rangle$, $Y = \langle y_1, y_2, \ldots, y_m\rangle$ secuencias y sea $Z = \langle z_1, z_2, \ldots, z_k\rangle$ la subsecuencia común más larga de $X$ y $Y$, cuya estructura óptima puede hallarse como sigue [5]:

1. Si $x_n = y_m$ entonces $z_k = x_n = y_m$ y $Z_{k-1}$ es la subsecuencia común más larga de $X_{n-1}$ y $Y_{m-1}$.
2. Si $x_n \neq y_m$ y $z_k \neq x_n$ entonces $Z$ es la subsecuencia común más larga de $X_{n-1}$ y $Y$.
3. Si $x_n \neq y_m$ y $z_k \neq y_m$ implica que $Z$ es la subsecuencia común más larga de $X$ y $Y_{m-1}$.

De lo anterior se puede formular la solución al problema general, construyendo una matriz $M[0 \ldots n, 0 \ldots m]$ en donde la entrada $M[i, j]$ guarda la longitud de la subsecuencia común mas larga de $X_i$ y $Y_j$.

| $M$ | | - | $y_1$ | $y_2$ | $\ldots$ | $y_{m-1}$ | $y_m$ |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | $\ldots$ | $m-1$ | $m$ |
| - | 0 | 0 | 0 | 0 | $\ldots$ | 0 | 0 |
| $x_1$ | 1 | 0 | | | | | |
| $x_2$ | 2 | 0 | | | | | |
| $\ldots$ | $\ldots$ | | | | ... | | |
| $x_{n-1}$ | $n-1$ | 0 | | | | | |
| $x_n$ | $n$ | 0 | | | | | ? |

Tabla 2.3: Dependencias en el problema de la subsecuencia común más larga.

La ecuación funcional que resuelve el problema se define como:

$$LCS_{M_{(n+1)\times(m+1)}}(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ ó } j = 0 \\ 1 + M[i-1, j-1] & \text{si } x_i = y_j \\ \text{máx}(M[i-1, j], M[i, j-1]) & e.o.c. \end{cases} \tag{2.4}$$

Una implementación en pseudocódigo es la siguiente:

```
1  LCS(i, j, M[0..n,0..m], X[1..n], Y[1..m]){
2    if i==0 o j==0 then
3      M[i,j]:= 0;
4    else if X[i] == Y[j] then
5      M[i,j]:= 1 + M[i-1, j-1];
6    else
7      M[i,j]:= max(M[i-1, j], M[i, j-1]);
8    end if
9  }
10
11 fillLCS(M[0..n,0..m], X[1..n], Y[1..m]){
12   for i:= 0 to n do
13     for j:= 0 to m do
14       LCS(i, j, M, X, Y);
15     end for
16   end for
17 }
```

## 2.4. Multiplicación en cadena de matrices (MCM)

Dada una secuencia de $n$ matrices:

$$A_0 A_2 \dots A_{n-1}$$

y un conjunto de enteros positivos $d_0, d_1, \dots, d_n$, donde la dimensión de la matriz $A_i$ es $d_i \times d_{i+1}$. Se requiere encontrar el orden en que se debe de multiplicar dicha secuencia de tal manera que el número total de multiplicaciones que se tengan que realizar sea mínimo.

Se debe recordar que cuando se multiplican matrices no necesariamente cuadradas sus dimensiones deben coincidir, es decir, si tenemos una matriz $A$ con dimensión $p \times q$ ($p$ renglones y $q$ columnas), solo se podrá multiplicar por una matriz $B$ con dimensión $q \times r$, dejando como resultado una matriz $C$ de dimensión $p \times r$. En otras palabras el número de columnas de $A$ debe de coincidir con el número de renglones de la matriz $B$. Formalmente el producto de matrices se puede definir como[9, 14]:

$$C[i,j] = \sum_{k=0}^{q-1} A[i,k] \cdot B[k,j] \quad \text{con} \quad 0 \leq i < p \quad \text{y} \quad 0 \leq j < r \tag{2.5}$$

Hay que notar que la matriz $C$ contiene $p \cdot r$ entradas y para encontrar cada valor se necesita de $q$ multiplicaciones, por lo tanto para calcular el producto de dos matrices se requieren $p \cdot q \cdot r$ multiplicaciones que es equivalente a la multiplicación de sus dimensiones. Otro factor importante que hay que tener tener en cuenta, es que la multiplicación de matrices es una operación asociativa pero no conmutativa, por lo que podemos parentizar la cadena de matrices como queramos, pero no podemos alterar el orden en que se encuentran dadas.

**Ejemplo 2.1.** *$A_0$ de dimensión $8 \times 3$, $A_1$ de dimensión $3 \times 7$ y $A_2$ de dimensión $7 \times 4$.*

$$\#Multiplicaciones(A_0 \cdot (A_1 \cdot A_2)) = (3 \cdot 7 \cdot 4) + (8 \cdot 3 \cdot 4) = 180$$

$$\#Multiplicaciones((A_0 \cdot A_1) \cdot A_2) = (8 \cdot 3 \cdot 7) + (8 \cdot 7 \cdot 4) = 392$$

Para fines prácticos de este problema sólo se calcula el mínimo número de multiplicaciones que son necesarias para multiplicar la cadena de matrices, dejando a un lado la forma de parentizarla. Para ello se van a ir guardando las soluciones de los subproblemas en una matriz $M$, en donde $M[i,j]$ va a denotar el mínimo número de multiplicaciones necesarias para calcular $A_i \dots A_j$, con $0 \leq i \leq j < n$.

- Claramente si $i = j$ entonces la secuencia sólo contiene una matriz y el número de multiplicaciones es igual a 0, ya que en realidad no hay nada que multiplicar, por lo tanto $M[i,j] = 0$.
- Si $i < j$ entonces se quiere el mínimo número de multiplicaciones necesarias para calcular $A_i \dots A_j$. Esto se puede obtener separando la cadena para cada $k$, donde

$i \leq k < j$ y multiplicando $(A_i \dots A_k) \times (A_{k+1} \dots A_j)$, pero estos valores por definición deben de encontrarse en $M[i,k]$ y $M[k+1,j]$ respectivamente.
Ahora como $A_i \dots A_k$ es una matriz de dimensión $d_i \times d_{k+1}$ y $A_{k+1} \dots A_j$ es de dimensión $d_{k+1} \times d_{j+1}$, entonces $d_i \cdot d_{k+1} \cdot d_{j+1}$ es el número de multiplicaciones necesarias para llevar a cabo su producto. Por lo tanto la solución óptima con el mínimo número de multiplicaciones para $(A_i \dots A_k) \times (A_{k+1} \dots A_j)$ está dada por el mínimo valor de $M[i,k] + M[k+1,j] + d_i \cdot d_{k+1} \cdot d_{j+1}$ con $i \leq k < j$.[10]

| $M$ | 0 | 1 | ... | $i$ | ... | $j$ | ... | $n-1$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | | | | | | | $M[1,n]$ |
| 1 | | 0 | | | | | | |
| ... | | | ... | | | | | |
| $i$ | | | | 0 | ... | $M[i,j]$ | | |
| ... | | | | | | | | |
| $n-1$ | | | | | | | | |

Tabla 2.4: Dependencias en el problema de la multiplicación en cadena de matrices.

De lo anterior se formaliza la ecuación funcional para el problema de la multiplicación en cadena de matrices, la cual queda definida por:

$$MCM_{M_{n\times n}}(i,j) = \begin{cases} 0 & \text{si } j \leq i \\ \min\limits_{i \leq k < j} (M[i,k] + M[k+1,j] + d_i \cdot d_{k+1} \cdot d_{j+1}) & \text{si } i < j \end{cases}$$

Y una posible implementación en pseudocódigo es la siguiente:

```
MCM(i, j, M[0..(n-1), 0..(n-1)], d[0..n]){
  if j <= i then
    M[i,j]:= 0;
  else
    minimun:= INFINITY;
    for k:= i to j-1 do
      aux:= M[i,k]+M[k+1,j]+d[i]*d[k+1]*d[j+1];
      if aux < minimun then
        minimun:= aux;
      end if
    end for
    M[i,j]:= minimun;
  end if
}

fillMCM(M[0..(n-1), 0..(n-1)], d[0..n]){
  for diag:= 0 to (n-1) do
    for id:= 0 to n-diag-1 do
      MCM(id, id+diag, M, d);
    end for
  end for
}
```

# Capítulo 3

# Cómputo en las GPU

Las unidades de proceso gráfico o GPU por sus siglas en inglés, tienen su origen a mediados de los años 90 y surgen de la necesidad de una creciente demanda de mejores gráficos, tanto para videojuegos, simulaciones espaciales, proyectos militares, robótica y mejores imágenes médicas, por lo cual los desarrolladores de hardware se vieron obligados a tratar de proveer alguna solución, provocando con ello una gran competencia de ideas, desde la forma en cómo implementar hardware, hasta el uso de diferentes técnicas para generar imágenes a partir de modelos, lo que actualmente se conoce como *rendering*[28]. Una de las primeras instancias a las cuales se recurrió, fue incrementar en cada generación la velocidad de los procesadores, provocando que se redujera el tiempo de las aplicaciones, sin embargo esta optimación fue siendo cada vez menos viable debido al consumo de energía y a los problemas de disipación de calor [15]. Otra de las ideas que más prosperó, fue introducir un microprocesador que ayudara con las funciones más básicas e intensivas del procesador principal, llevándolas a cabo por su propia cuenta y liberándolas por completo del CPU, con el fin de que este pudiera dedicarse a otras tareas, a este tipo de microprocesador se le llamo *coprocesador.*[31]

Debido a que el coprocesador es el que realizaba muchos de los cálculos más intensivos para los gráficos, los vendedores de hardware empezaron a incorporar múltiples unidades de proceso (conocidos como núcleos de procesamiento), en el mismo chip para incrementar el poder de cómputo y fue este cambio lo que propició un enorme impacto en la evolución de la tecnología, ya que de esta manera se dotaba al coprocesador de un modelo completamente diferente de cómputo [30]. Conforme la tecnología siguió avanzando, se iban delegando más y más funciones del procesador hacia el coprocesador haciéndolo cada vez más indispensable. Al principio las operaciones más importantes que se llevaban a cabo en el coprocesador eran filtrar texturas y procesar la geometría del entorno, hasta que en agosto de 1999 la empresa NVIDIA presentó la GeForce 256 acuñándole por primera vez en la historia el término de GPU[20] e incorporando funciones de transformación e iluminación, lo que causó una gran revolución.

Cada generación que salía al mercado incluía nuevas mejoras, haciendo cada vez más

potentes a las GPU, lo que dio pauta a que algunos científicos centraran su atención en ellas, con el fin de poderlas utilizar en sus proyectos de investigación, entre otras cosas a la aceleración de sus aplicaciones. Aunque esta tarea era posible realizarla en una GPU llegando a reportarse un rendimiento considerable, existía un gran problema, el cual se encontraba relacionado a la flexibilidad que tenían los usuarios al programar este tipo de dispositivos, ya que se encontraban muy limitados en las cosas que podían realizar, debido a que tenían que utilizar las API[1] de programación para gráficos como OpenGL, aunque la aplicación en sí sólo hiciera uso de los núcleos de procesamiento para llevar a cabo cálculos. Esto limitó considerablemente todo el potencial de las GPU para poder llevar a cabo cómputo científico.[24]

Debido a que se tenía poca flexibilidad a la hora de programar aplicaciones de propósito general para estos dispositivos, los desarrolladores de hardware, empresas de cómputo y la comunidad científica, empezaron a tomar diferentes acciones para que las GPU fueran completamente programables, con ello se empezaron a desarrollar diferentes API y lenguajes de programación para dar solución al problema. Entre los más populares se encuentran, *ATI Stream Computing* de ATI comprada por AMD, *Microsoft Direct Compute* de Microsoft, *CUDA* de NVIDIA y *OpenCL* que es de código abierto y no tiene restricciones de hardware o sistema operativo como los anteriores.[25]

A partir de ese momento, las aplicaciones ya no solo son capaces de utilizar el procesador principal para ejecutarse, sino que pueden auxiliarse de las GPU para llevar a cabo su cometido, así las partes del código que requieren de gran desempeño computacional pueden delegarse a la GPU, mientras que el resto del código puede seguirse ejecutando en el CPU, con esta combinación se logra obtener un rendimiento mucho mayor que utilizando sólo el procesador. Al uso del GPU para este tipo de aplicaciones se le llamo cómputo de propósito general en unidades de proceso gráfico o *GPGPU* por sus siglas en inglés.

La gran popularidad que han tenido las GPU de NVIDIA, el desarrollo de CUDA C que es una extensión del lenguaje C para poder programar estos dispositivos, las grandes ventajas que se ofrecen actualmente, la imposibilidad de abarcar cada uno de los modelos de programación y arquitecturas que circulan actualmente por el mercado, son los principales motivos por los cuales el desarrollo de este trabajo se centra utilizando únicamente las GPU de la compañía NVIDIA. Proponiendo como trabajo futuro la migración del conocimiento adquirido utilizando otro modelo de programación y/o arquitectura.

## 3.1. CUDA

Desde que comenzó la revolución de las tarjetas gráficas, la atención para utilizar este tipo de dispositivos se dirigió a las aplicaciones y procesamiento de gráficos tridimensio-

[1] API es la abreviatura de *Application Programming Interface* o Interfaz de Programación de Aplicaciones el cual es un conjunto de funciones y procedimientos que pueden ser utilizados por otro software.

nales, por lo cual se empezaron a desarrollar chips especialmente dedicados a funciones de transformación, iluminación y filtro de texturas, culminando en la incorporación de los *pixel shaders* y los *vertex shaders*, que procesan de manera separada cada pixel y cada vértice, lo cual brinda a los programadores una mayor libertar a la hora de diseñar gráficos tridimensionales. En forma general un *vertex shader* es una función que recibe como parámetro un vértice y lo transforma de tal manera que repercute en la geometría del objeto al que pertenece, mientras que un *pixel shader* se encarga de producir un color para cada pixel en la pantalla.

La idea general de los pixel shaders es utilizar una posición $(x, y)$ de la pantalla junto con alguna información adicional, combinarlos de alguna manera y generar un color final para el pixel; la información adicional puede ser un color, una textura o simplemente valores numéricos. Debido a este hecho los investigadores se dieron cuenta que los colores finales podían representar cualquier cosa, pudiendo programar los pixel shaders para realizar cómputo general sobre esta información. Sin embargo a pesar de este descubrimiento las tarjetas gráficas siguieron construyéndose sólo con soporte para pixel y vertex shaders. No fue hasta el año 2006 que la compañía NVIDIA lanzó la GeForce 8800 GTX, la primera GPU construida con la arquitectura CUDA[2][22]. Esta arquitectura incluye varios componentes diseñados exclusivamente para el cómputo en GPU, aligerando muchas de las limitaciones que impedían que los procesadores gráficos de las generaciones anteriores, fueran utilizados legítimamente para el cálculo de propósito general.

### 3.1.1. ¿Qué es CUDA?

CUDA es la arquitectura que desarrolló NVIDIA para permitir que las GPU pudieran ser utilizadas en aplicaciones de propósito general. En esta arquitectura los recursos de cómputo no sólo se dividen en vertex y pixel shaders, sino que se añade un shader unificado que permite a cada unidad aritmética lógica (ALU) en el chip ser utilizada con el fin de efectuar cálculos generales. Para que las operaciones de cómputo fueran compatibles con el estándar que se tenía en las CPU, las ALU fueron construidas siguiendo los requerimientos de la IEEE[3] para las operaciones de punto flotante de precisión simple y se diseñó un conjunto de nuevas instrucciones que facilitaban a los programadores el desarrollo de aplicaciones sobre esta arquitectura, como por ejemplo, instrucciones que permitían el libre acceso a la memoria para realizar operaciones de escritura y de lectura, así como acceso a una memoria caché administrada por software, conocida como memoria compartida (*shared memory* en inglés). [26]

Aunque por fin se desarrollaban avances en el soporte al hardware de las GPU para realizar cómputo general, seguía siendo necesario que los programadores escribieran su código empleando un lenguaje orientado a gráficos y utilizaran las API como OpenGL y DirectX para poder acceder a estas propiedades. NVIDIA se dio cuenta de esto, por

[2]Acrónimo para Compute Unified Device Architecture.
[3]Siglas en inglés para Institute of Electrical and Electronics Engineers.

lo que decidió añadir un conjunto pequeño de instrucciones al lenguaje C, para poder explotar las características especiales de CUDA. A este nuevo lenguaje se le llamo *CUDA C* y poco tiempo después se liberó un compilador que lo soportaba. Gracias a esto ya no era necesario que los desarrolladores tuvieran algún conocimiento previo de OpenGL o de programación de gráficos, ya que el lenguaje fue desarrollado especialmente para facilitar el cómputo de propósito general en las GPU desarrolladas por la empresa NVIDIA.

## 3.2. Arquitecturas CUDA

Tanto los CPU como las GPU son unidades de procesamiento, cuya función principal es recibir datos y procesarlos por medio de un conjunto de instrucciones preestablecidas. Una de las diferencias más notorias entre ambos radica en que el primero se encuentra integrado con pocas ALU, mientras que en una GPU se puede contar con cientos o incluso miles de estas. Sin embargo lo más trascendental es que a la hora de programar estos dispositivos implica un cambio de paradigma, es por ello que tenemos que desmenuzar por completo el trasfondo de lo que realmente es una GPU, adentrarnos en el diseño y la estructura operacional fundamental y conocer la forma en cómo se interconectan e interactúan los diferentes componentes de hardware que integran el dispositivo. En otras palabras tenemos que comprender su arquitectura. Todo este conocimiento servirá para entender realmente las cosas que puede realizar una GPU, como es que las puede realizar y tener en mente todas las limitaciones que ello conlleve.

### 3.2.1. Arquitectura G80

La arquitectura G80 fue la primera en adoptar el nuevo modelo de programación para cómputo de propósito general en las GPU y fue lanzada con la serie 8 de GeForce, en particular con la tarjeta gráfica GeForce 8800. A partir de ella fue que se empezaron a desarrollar avances, mejoras e innovaciones, que hasta el día de hoy han culminado en la creación de la arquitectura Kepler, que es con la que actualmente son construidas las nuevas tarjetas gráficas. Es por este motivo que es necesario observar sus características más importantes y que han perdurado en la evolución de las arquitecturas.

Las unidades básicas de las que se encuentra construida una GPU son los *streaming processors* (SP) o *CUDA cores* y que en una definición simple son microprocesadores que contienen dos ALU, una para operaciones con punto flotante y otra para enteros. El principal objetivo de un SP, es que pueda ser utilizado para realizar una gran cantidad de operaciones matemáticas, por lo cual no es necesario incluir una memoria caché dentro de él. Por si solo un SP puede resultar inservible, pero si se agrupan una gran cantidad de ellos se puede empezar a realizar algo mucho más productivo, en particular aquellas aplicaciones cuyas representaciones sean muy paralelizables. Para esta arquitectura se agruparon 8 de estos SP en un arreglo, que se le llamo *streaming multiprocessor* (SM) y se adicionaron dos nuevos microprocesadores llamados unidades de funciones especiales o SFU por su nombre en inglés. Cada SFU se encuentra integrado por cuatro ALU de

punto flotante y es utilizado para realizar operaciones específicas (sin, cos, tan), interpolación y filtrado de texturas. Aunado a esto se incluyó en cada SM un pequeño caché L1 tanto para instrucciones como para datos, 16 Kb de memoria compartida, 8192 registros de 32 bits y una unidad que despacha las instrucciones (*Instruction Fetch/Dispatch*) a todos los SP y SFU en el SM.

Agrupando varios SM se tiene lo que se llama un grupo de textura/proceso o TPC. En la arquitectura G80 cada TPC se integró con dos SM y se agregó una unidad lógica de control junto con un bloque de textura, el cual incluía instrucciones de filtrado, direccionamiento de texturas, así como una memoria caché L1 exclusiva para texturas. El esquema modular continúa agrupando 8 TPC para construir un *Streaming Proccessor Array (SPA)*, al que finalmente se le agrega una unidad caché L2 de textura, un planificador (scheduler en inglés) cuya función consiste en distribuir el trabajo a todo el arreglo de TPC al cual se le nombró *GigaThread* y una memoria RAM propia del dispositivo, conocida como *DRAM*. Todo este conjunto de componentes integran lo que es una GPU con arquitectura G80. Las nuevas generaciones que ha desarrollado NVIDIA contienen en esencia este mismo esquema modular, inclusive algunos mismos componentes pero con actualizaciones importantes. [1, 19]

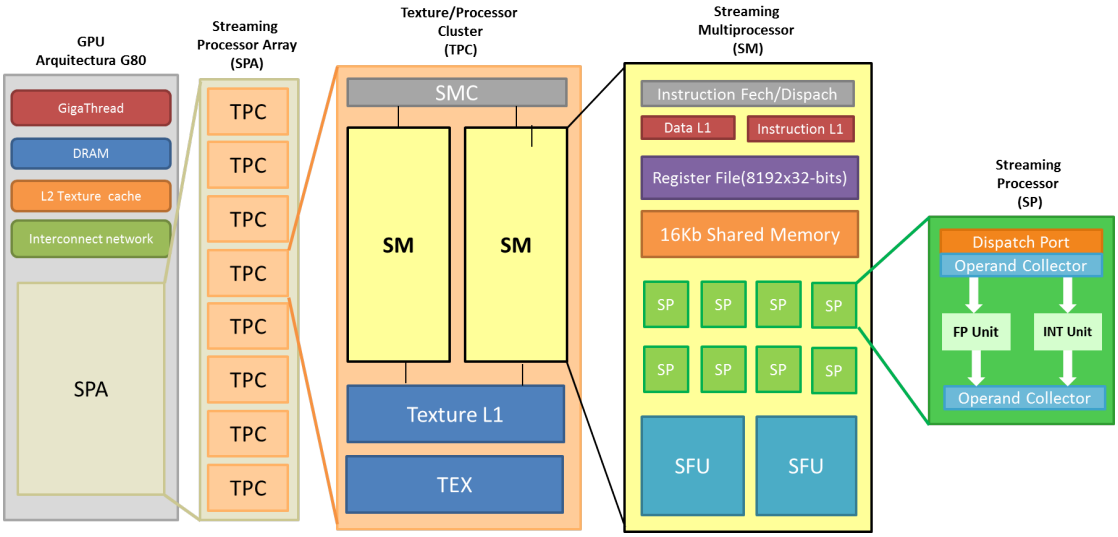


Figura 3.1: Principales componentes que integran la arquitectura G80.

### 3.2.2. Arquitectura Tesla

Esta arquitectura fue lanzada junto con la tarjeta GeForce GTX 280 en junio del 2008 y contempla una actualización a la arquitectura anterior. Esta nueva generación llamada *Tesla* (En honor al físico Nikola Tesla), en su versión *GT200* incorpora 3 SM por cada TPC y 10 TPC por cada GPU, como consecuencia se incrementó el número

total de SP de 128 a 240. También se duplicó el número de registros pasando de 8192 a 16384 y la caché de textura L2 finalizando con 256Kb, lo que permitió cargar más hilos de ejecución (threads en inglés) en cada TPC, logrando realizar mayor trabajo en forma paralela. [16]

### 3.2.3. Arquitectura Fermi

Después de haber presentado las dos primeras arquitecturas, NVIDIA se concedió un tiempo para analizar hacia dónde iba a encaminar la siguiente generación de CUDA, por lo cual apoyándose ampliamente en los comentarios de los usuarios, rediseñó varios de los componentes e introdujo muchas mejoras significativas que hasta el momento se siguen utilizando en la fabricación de las GPU. A esta nueva arquitectura se le conoció con el nombre de *Fermi* (en distinción al físico Enrico Fermi) y fue lanzada junto con las tarjetas GeForce de la serie 400 en marzo de 2010. Los cambios que tuvieron los SP fueron relativamente pocos, los cuales consisten en la adopción del nuevo estándar para punto flotante (IEEE 754-2008), precisión de 32 bits para todas las instrucciones en la ALU de enteros y soporte para operaciones como *shift, move y compare* entre otras. El componente con el mayor número de cambios en su diseño original y el que presentó las características más innovadoras fue el SM, las cuales se describen a continuación[22, 33]:

- Se realizó un incremento radical en el número de SP pasando de 8 a 32.
- Se agregaron dos nuevos SFU teniendo 4 en total.
- Se volvió a duplicar el número de registros llegando a 32768.
- Se incrementó a 64 Kb la memoria compartida, realizando la innovación de poder ser configurable. Se puede optar por la opción de asignar 48 Kb a memoria compartida y 16 Kb a un caché L1 o viceversa.
- Se añadieron 16 unidades *load/store* que permiten calcular direcciones de origen y destino de 16 hilos por ciclo de reloj, permitiendo cargar y almacenar datos para cada dirección ya sea a memoria DRAM o a caché.
- Se incluyeron 4 unidades de textura junto con una caché unificada.
- Se agregó un módulo con funciones especializadas para gráficos conocido como *polymorph engine*.
- Se creó un planificador de trabajo en grupos de 32 hilos llamado *warp scheduler* y se incluyeron dos de estos por cada SM, lo que permite que dos grupos de 32 hilos puedan ser ejecutados de manera concurrente.

Debido a estos enormes cambios que tuvieron los SM, algunos componentes tuvieron que suprimirse, cambiaron de nombre o se les agregaron nuevas innovaciones. Los cambios que surgieron en algunos componentes y las características más importantes que adoptó la nueva arquitectura se enuncian a continuación:

1. Los TPC ya no tenían cabida por lo que se decidió quitarlos.
2. Los SPA cambiaron de nombre a grupos de proceso gráfico o GPC por sus siglas en inglés.
3. Se incluyeron operaciones atómicas como *add, min, max, compare-and-swap* entre otras, para proporcionar un mecanismo que permitiera obtener resultados correctos cuando se trabajaba con operaciones de lectura y escritura ejecutando múltiples hilos concurrentes sobre un mismo intervalo de memoria.
4. La repartición de trabajo se realiza en dos niveles, en el primero el trabajo global es distribuido por el planificador GigaThread mandando bloques completos de hilos a los diferentes SM, mientras que el segundo nivel se realizaba en el propio SM, distribuyendo grupos de 32 hilos en las diferentes unidades de ejecución.
5. Finalmente se realizó una mejora significativa en la velocidad de cambios de contexto y se permitió la ejecución de hasta 16 *kernels* de manera concurrente en el mismo dispositivo. En la siguiente sección describiremos propiamente lo que es un kernel.
6. Cada GPU se construía con los GPC colocados alrededor de una memoria caché L2 común para todos.

En su versión *GF100* cada GPC consistía de 4 SM y contenía 768 Kb de memoria caché L2. Fermi dio un giro revolucionario a la capacidad y el propósito de las GPU, lo cual definió la nueva dirección a la que NVIDIA encaminaría sus nuevas generaciones de arquitecturas.


Figura 3.2: Esquema general de un SM en la arquitectura Fermi y forma de agrupar los componentes en su versión GF100.

### 3.2.4. Arquitectura Kepler

La arquitectura de la serie 600 de NVIDIA, se encuentra diseñada sobre muchos de los principios establecidos por su antecesora. Se le dio el nombre de *Kepler* en distinción al matemático Johannes Kepler, es considerada una de las de más alto rendimiento y eficiencia en consumo de energía. La tarjeta gráfica que introduce Kepler como arquitectura base fue la *GeForce GTX 680*, fue puesta a la venta en marzo de 2012 y su versión *GK104* se encuentra compuesta por 4 GPC cada uno con 2 SM de nueva generación que fueron renombrados a *SMX*. Las innovaciones que presentaron los nuevos SMX se enuncian a continuación[21]:

- Incrementaron el número de SP de manera exorbitante llegando hasta los 192.
- Se duplicó el número de unidades load/store teniendo en total 32.
- Las unidades de textura pasaron de 4 a 16 y las SFU de 4 a 32.
- Se agregaron dos nuevos planificadores para grupos de hilos.
- Se volvió a incrementar el número de registros beneficiándose ahora de 65536.

- Se agregó una nueva configuración para la memoria caché L1, contando con 3 opciones, 16 Kb, 32 Kb y 48 Kb.

La siguiente versión de Kepler cuyo código es el *GK110*, fue incluida en la serie 700 de GeForce realizando su debut con la tarjeta *GeForce GTX 780* en mayo del 2013. Tuvo algunos pequeños cambios en el diseño de los SMX, cuyos más relevantes se enuncian en seguida[23]:

- Se quitó el caché de textura, reemplazándolo por 48 Kb de memoria caché de sólo lectura aunque siguió siendo accesible exclusivamente para las unidades de textura.
- Se quitó el módulo polymorph engine.
- Se agregaron 64 unidades de punto flotante de precisión doble o *DP Units*

Finalmente una de las innovaciones más relevantes en esta arquitectura, fue la inclusión del *paralelismo dinámico* el cual añade la capacidad de generar nuevo trabajo por sí mismo, poder sincronizarlo y administrarlo a través de hardware dedicado, sin la necesidad de involucrar al CPU.


Figura 3.3: Esquema general de los componentes que integran los SMX para las versiones GK104 Y GK110.

En marzo de 2013 NVIDIA anunció que la siguiente generación de CUDA será conocida como *Maxwell* en honor al físico escocés James Clerk Maxwell, se espera que sea la primera en introducir un CPU ARM integrado y sea presentada al público a principios del 2014 [27].

Para indicar la generación de la arquitectura en que está basado un cierto chip de una GPU, NVIDIA creó el concepto de *Compute Capability* o *CC*, el cual es un número decimal con dos dígitos separados por un punto, en donde el primero indica la generación de la arquitectura con cambios mayores, mientras que el segundo se designa para cambios menores realizados en ella, denotamos como versiones. En general entre más grande sean los números de cada componente, indica que el GPU consta de características e innovaciones superiores.

A continuación se presenta una tabla que resume las propiedades más importantes de cada arquitectura. Debemos aclarar que cada generación descrita anteriormente puede contener muchas más versiones de las que aquí se anunciaron. Sin embargo para los propósitos de este trabajo, sólo se enumeraran las que se consideran más importantes debido al impacto que tuvieron al incrementar el desempeño computacional, en consecuencias de su diseño, innovaciones realizadas o características específicas.

| GPU | G80 | GT200/Tesla | GF100/Fermi | GK104/Kepler | GK110/Kepler |
|---|---|---|---|---|---|
| Por cada unidad SM: | | | | | |
| CUDA cores (SP) | 8 | 8 | 32 | 192 | 192 |
| SFU | 2 | 2 | 4 | 32 | 32 |
| Registros | 8192 | 16384 | 32768 | 65536 | 65536 |
| Shared Memory/L1 | 16Kb/0Kb | 16Kb/0Kb | 16Kb/48Kb<br>48Kb/16Kb | 16Kb/48Kb<br>32Kb/32Kb<br>48Kb/16Kb | 16Kb/48Kb<br>32Kb/32Kb<br>48Kb/16Kb |
| Load/Store Units | | | 16 | 32 | 32 |
| Texture Units | 0 | 0 | 4 | 16 | 16 |
| Warp schedulers | 1 | 1 | 2 | 4 | 4 |
| Caché L2 | 128Kb(Textura) | 256Kb(Textura) | 768Kb | 512Kb | 1536Kb |

Figura 3.4: Resumen de las características más importantes para cada arquitectura.

## 3.3. Modelo de programación

Un modelo de programación consta de diferentes mecanismos (métodos, propiedades, eventos, interfaces, etc.) que se le proveen al desarrollador para que pueda expresar la lógica de un programa. Para utilizar CUDA se tuvo que extender el lenguaje de programación C, para que permitiera definir código que sería ejecutado por la GPU. A este nuevo lenguaje se le llamo *CUDA C* y se creó un compilador que lo reconociera, al

cual se le conoce con el acrónimo de *nvcc* proveniente de NVIDIA CUDA Compiler. De ahora en adelante llamaremos por *dispositivo* a la tarjeta gráfica y *equipo anfitrión* a la computadora donde se encuentra conectado el dispositivo como coprocesador.

El desarrollo de un programa en el que parte de su código queremos que sea ejecutado por el dispositivo, se basa en la especificación de dos conjuntos de instrucciones. Por un lado las que queremos que se ejecuten propiamente en el dispositivo y aquellas que tienen o pueden ser ejecutadas por el equipo anfitrión. El principal objetivo al modificar el lenguaje C, fue que este pudiera aceptar un nuevo tipo de función llamada *kernel*, la cual tiene como propósito mandar a realizar el trabajo en el dispositivo y todo el código que contenga será ejecutado de manera paralela por múltiples hilos. El modelo de programación define y establece la forma en la que es ejecutado un kernel dentro del dispositivo, es por ello que en las próximas secciones se pretende describir con mayor detalle.

### 3.3.1. Jerarquía de hilos

A continuación se describen los diferentes componentes, en los que se divide un kernel para su ejecución:

- Un programa diseñado para ser ejecutado por el dispositivo, se encuentra compuesto por uno o varios *kernels* que pueden ejecutarse de manera concurrente dependiendo de su arquitectura.
- Cada kernel es ejecutado como una malla de bloques (grid en inglés), que se encuentra organizado en dos o tres dimensiones y cuyos valores son declarados por el usuario respetando siempre los límites que establece cada arquitectura. Las dimensiones además de definir el número de bloques que contendrá la malla, son utilizadas para poder asociarle un identificador único a cada bloque y puede ser accedido por medio de la variable *blockIdx* indicando la dimensión que se requiere. Varias mallas de bloques pueden ser sincronizadas a través de una serialización de kernels independientes.
- Un *bloque* a su vez es un conjunto de hilos, que se encuentran dispuestos en tres dimensiones y al ser definidas por el programador queda automáticamente establecido el número de hilos que contendrá cada bloque. Debemos aclarar que las diferentes arquitecturas definen un número máximo de hilos por bloque, lo que conlleva a que muchas de las configuraciones en las que puede ser definida la dimensión no serán válidas. Al igual que sucede en una malla, cada bloque asocia un identificador único a los hilos y puede ser accedido mediante la variable *threadIdx* indicando la dimensión requerida. Los hilos en un mismo bloque pueden interactuar y cooperar entre ellos a través de memoria compartida y barreras de sincronización para llevar a cabo su cometido, lo que se realiza mediante la función *__syncthreads()*.

- Un *hilo de ejecución* o simplemente hilo es la unidad básica de ejecución.
- Internamente y por parte del sistema, cada bloque es dividido en grupos de 32 hilos consecutivos de un bloque para todas las arquitecturas, que de ahora en adelante llamaremos *warps*. Estos son lo que serán ejecutados físicamente por cada SM, por ello son la unidad básica de planificación.

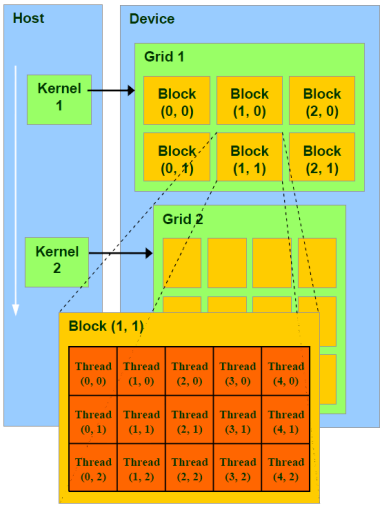


Figura 3.5: Representación de mallas y bloques de dimensión dos en la jerarquía de hilos.

### 3.3.2. Planificación y ejecución de un kernel

Una vez que entendemos la forma en como es dividido un kernel para ser ejecutado por el dispositivo, podemos comprender el procedimiento en que los diferentes elementos de la jerarquía son planificados y distribuidos entre los componentes de hardware, lo que nos brinda una visión general de lo que realmente sucede al ejecutar nuestro programa. Al conocer nuestra arquitectura y la forma de distribución de trabajo, podemos desarrollar kernels que aprovechen eficientemente los recursos con los que se cuenta.

- Cuando se manda a ejecutar un kernel, este es visto inmediatamente como una malla de bloques, con dimensiones definidas por el programador. Esta malla será ejecutada por completo en el dispositivo.
- Posteriormente los bloques son distribuidos secuencialmente hacia los diferentes SM quedando asignados al correspondiente. Lo que provoca que toda la ejecución de un bloque se lleve a cabo en un mismo SM, sin la posibilidad de migrar hacia otro.
  Potencialmente cada SM pude albergar a más de un bloque, lo que implica que múltiples bloques pueden ejecutarse de forma paralela. Aunque esta característica

queda limitada por los recursos y capacidades definidas en cada arquitectura. El planificador global puede asignar libremente la ejecución de los bloques, ya que cada uno es independiente, por lo que nunca debe de suponerse que se ejecutarán en un cierto orden o por algún SM.

- Una vez que los bloques son asignados a un SM, este los divide en un conjunto de warps y su ejecución es programada por los planificadores contenidos en el SM. En ciertas arquitecturas es posible una ejecución paralela de los warps.
  Al igual que sucede en los bloques, los warps pueden ser calendarizados en cualquier momento, incluso aun cuando se encuentran en ejecución. El planificador es libre de interrumpirla para darle tiempo de procesamiento a otro warp.

- Cuando un warp es seleccionado por el planificador, mapea los hilos de los que se compone a los SP y estos a su vez ejecutan de manera paralela una instrucción común para todos. Con ello el modelo de ejecución se transforma en un SIMD (Single Instruction Multiple Data), es decir una sola instrucción aplicada a diferentes valores.
  Si algún hilo en el warp diverge por motivos como condicionales o dependencia de datos, se crean distintas ramas de ejecución. Las cuales se serializan en el warp, deshabilitando en cada caso aquellos hilos que no se encuentran en la misma rama, realizándolas de manera separada hasta que estas vuelvan a converger. Las divergencias se producen sólo dentro de un warp; diferentes warps pueden manejar y ejecutar independientemente sus ramas de ejecución.

En resumen la malla de bloques se mapea al dispositivo, el conjunto de bloques a los SM y los hilos a los SP. El número de bloques/warps que pueden residir y ser procesados por un SM, se encuentra en función de la cantidad de registros, configuración de lanzamiento y memoria compartida que utiliza el kernel. Cada arquitectura define entre otras cosas un número máximo de bloques, warps e hilos residentes por SM. Si no existen suficientes recursos disponibles por SM para procesar al menos un bloque, la ejecución completa del kernel fallará. La siguiente tabla muestra los diferentes valores que definen las arquitecturas:

| Especificaciones | Compute Capability | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1.0 | 1.1 | 1.2 | 1.3 | 2.x | 3.0 | 3.5 |
| Dimensiones máximas en el Grid | ($2^{16}$-1, $2^{16}$-1) | | | | ($2^{16}$-1, $2^{16}$-1, $2^{16}$-1) | ($2^{31}$-1, $2^{16}$-1, $2^{16}$-1) | |
| Dimensiones máximas en el Bloque | (512, 512, 64) | | | | (1024, 1024, 64) | | |
| Número de threads por Bloque | 512 | | | | 1024 | | |
| Tamaño del warp | 32 | | | | | | |
| Número máximo de bloques residentes por SM | 8 | | | | | 16 | |
| Número máximo de warps residentes por SM | 24 | | | 32 | 48 | 64 | |
| Número máximo de threads residentes por SM | 768 | | | 1024 | 1536 | 2048 | |
| Número máximo de registros de 32-bits por thread | 128 | | | | 63 | | 255 |
| Número máximo de registros de 32-bits por SM | 8K | | | 16K | 32K | 64K | |
| Cantidad máxima de memoria compartida por SM | 16K | | | | 48K | | |

Figura 3.6: Recursos y características que se definen en el compute capability.

### 3.3.3. Jerarquía de memoria

Los hilos son los elementos capaces de acceder a los diferentes tipos de memoria definidos en una arquitectura, en donde cada una de ellas está destinada a un cierto propósito y contiene características particulares que determinan su visibilidad y forma en la que puede ser accedida. Por lo cual dependiendo del nivel en la jerarquía de hilo, se puede acceder o no a cierto tipo de memoria. A continuación se detalla la jerarquía de la memoria:

- *Registros*: es la memoria más rápida de lectura y escritura, ya que se encuentra directamente integrada en los SM. A cada hilo se le asocia un conjunto privado de registros, que pude accederlos en cualquier momento y no necesita la intervención del programador ya que son manejados por el sistema.

- *Memoria local*: A su vez cada hilo contiene su propia memoria local privada de lectura/escritura, en donde guarda entre otras cosas variables locales, llamadas a funciones y su contexto de ejecución. Esta memoria es administrada por el sistema.

- *Memoria compartida*: Este tipo de memoria es una caché L1 de lectura/escritura, que es visible por todos los hilos que conforman un bloque, lo que implica que algún hilo residente en otro bloque nunca podrá acceder o modificar la misma región de memoria compartida. Esto provee una excelente forma en que los hilos de un mismo bloque pueden comunicarse y colaborar. Todos los valores contenidos en la

memoria compartida son persistentes hasta que se termina de ejecutar por completo el bloque. Para crear este tipo de memoria se utiliza la directiva *__shared__*.

- *Memoria global*: Es la memoria más abundante en la GPU, llegando inclusive a los gigabytes para algunos modelos de tarjetas gráficas. Es de lectura/escritura utilizada principalmente para guardar y recuperar los datos que fueron procesados en el GPU. Su contenido es visible y puede ser modificado por todos los hilos que componen un kernel. Sirve como puente para intercambiar información con el equipo anfitrión, es por ello que éste debe de encargarse de crearla, inicializarla y liberarla mediante las funciones *cudaMalloc*, *cudaMemcpy* y *cudaFree* respectivamente. Los valores contenidos en esta memoria son persistentes hasta que se liberen los recursos.

- *Memoria constante:* Es una memoria especializada que puede ayudar en la mejora del rendimiento en las aplicaciones. Como su nombre lo indica su propósito es albergar datos que no cambiarán en el transcurso de la ejecución, por lo que es exclusivamente de sólo lectura y su contenido debe de ser inicializado por el equipo anfitrión. Tiene mecanismos de caché que hacen eficiente la lectura, lo que ocasiona que en algunas situaciones su uso provoque un mejor desempeño. Este tipo de memoria puede ser accesible por todos los hilos de una malla y puede ser creada utilizando la directiva *__constant__*.

- *Memoria de textura:* Al igual que la memoria constante, la memoria de textura es también de sólo lectura y aunque en principio está diseñada para utilizarse con gráficos, puede ser utilizada con eficiencia en aplicaciones de propósito general. Sus mecanismos de caché están optimizados para patrones de acceso espacial en dos dimensiones. Las lecturas tienen algunas ventajas como por ejemplo, diferentes modos de acceso e interpolación que se pueden utilizar sin ningún costo adicional.

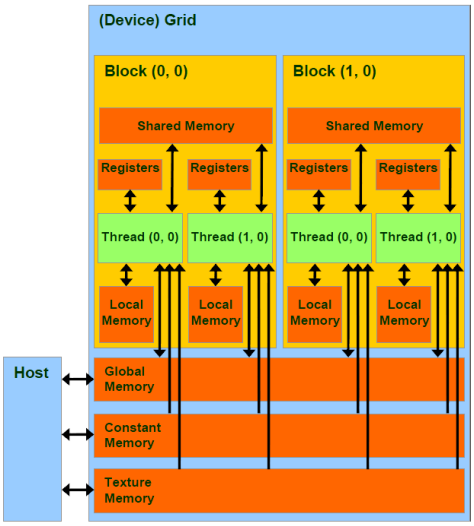


Figura 3.7: Visibilidad y acceso de los diferentes tipos de memoria por los hilos.

### 3.3.4. Esquema general de programación

Como se había mencionado anteriormente, CUDA C es una extensión del lenguaje C, por lo que el código de un programa se basa completamente en su sintaxis, salvo algunas directivas que se agregaron para indicar que partes se ejecutaran en el dispositivo. La definición de un kernel se realiza mediante la declaración de una función, la cual se le antecede la directiva *__global__*. Los kernels pueden invocarse en cualquier momento por el equipo anfitrión de la misma forma que cualquier función, excepto que después del nombre siguen los símbolos $<<<, >>>$ en donde la primera componente declara las dimensiones de la malla y la segunda las dimensiones que contendrá cada bloque.
CUDA C también permite definir funciones ejecutadas en el dispositivo a las cuales se les antepone la directiva *__device__*. Al usar estas dos directivas se debe de tener en cuenta las siguientes consideraciones:

- Una función ejecutada en el equipo anfitrión puede recurrir a una función global, lo que se le conoce como lanzamiento del kernel. Pero no puede invocar a una función en el dispositivo.
- Una función global no puede invocar a una ejecutada en el equipo anfitrión, pero si a una en el dispositivo. Sólo para CC 3.5 se permite lanzar funciones globales, lo que se conoce como *dynamic parallellism*.
- Una función en el dispositivo puede ejecutar algún otra del mismo tipo, pero no una declarada en el equipo anfitrión. Únicamente se permite la invocación a una función global a partir de CC 3.5.

El esquema general para desarrollar un programa en CUDA se desprende de los siguientes pasos:

- Declarar todas las funciones globales y aquellas ejecutadas en el dispositivo que describen las instrucciones a realizar en la GPU.
- Asignar memoria en el dispositivo por medio de la función *cudaMalloc*, con el objetivo de intercambiar datos entre el equipo anfitrión y el dispositivo.
- Copiar toda la información inicial que requieran los kernels para llevar a cabo su labor, del equipo anfitrión al dispositivo.
- Ejecutar los kernels definiendo su configuración de lanzamiento.
- Copiar los resultados generados por el dispositivo de regreso al equipo anfitrión.
- Liberar la memoria y los recursos utilizados.

```
__device__ deviceA(){...}
__global__ kernelA(T* Data){
  ...
  deviceA();
  ...
}
__global__ kernelB(){...}

int main(){
  ...
  T* Data;
  cudaMalloc(&Data, sizeData);
  ...
  cudaMemcpy(Data, src, sizeToCpy, cudaMemcpyHostToDevice);
  ...
  dim3 gridDim(gx, gy, gz);
  dim3 blockDim(bx, by, bz);
  kernelA<<<gridDim, blockDim>>>(Data);
  ...
  cudaMemcpy(Data, res, sizeToCpy, cudaMemcpyDeviceToHost);
  ...
  kernelB<<<...>>>();
  ...
  cudaFree(Data);
  ...
}
```

## 3.4. Consideraciones de rendimiento

Uno de los principales objetivos a la hora de concebir un programa en un GPU, es que pueda ser capaz de realizar su trabajo en una menor cantidad de tiempo. Esto

debido simplemente a la idea de que se cuenta con cientos o incluso miles de pequeñas ALU que cooperan entre sí, dividiéndose el trabajo total a realizar.
La cantidad de trabajo útil realizado por el sistema dependiendo del tiempo y de los recursos utilizados, define el rendimiento del equipo. Para conocer esta proporción se creó una medición que da a conocer el factor de velocidad que se logró obtener y se le conoce como *speedup*. La cual se refiere a que tan rápido se ejecutó un programa en paralelo contra el correspondiente algoritmo secuencial y se define mediante la siguiente fórmula[2]:

$$velocidad = \frac{T_s}{T_p} \tag{3.1}$$

donde:

- $T_s$ es el tiempo que tardo el algoritmo secuencial.
- $T_p$ es el tiempo que tardo el algoritmo paralelo.

La idea para tener un mejor rendimiento en un programa es tratar de aumentar el factor de velocidad, lo que implica reducir $T_p$ lo más que se pueda. Para lograr este cometido se pueden considerar estas tres estrategias básicas:

1. Maximizar el rendimiento en la memoria.
2. Optimizar el uso de instrucciones.
3. Maximizar la ejecución en paralelo aumentando la ocupación.

Se debe hacer énfasis en que todas estas optimizaciones son responsabilidad única del programador, que debe de tenerlas en cuenta a la hora de escribir el código. Ya que aunque suenen muy simples pueden repercutir enormemente en el desempeño del programa. En las siguientes secciones se describen con mayor detalle.

### 3.4.1. Maximizar el rendimiento en la memoria

La mejor forma de maximizar el rendimiento en la memoria, es sin duda reducir las trasferencias a memorias de baja latencia, que son aquellas que toman mucho tiempo en trasladar información de un lado a otro, como por ejemplo la memoria global. El consejo es que cuando se tiene un programa que es altamente dependiente en los datos y realiza numerosas peticiones a memoria global, se puede auxiliar de memorias mucho más rápidas como la caché L1, L2 o la memoria compartida. Ya que estas se encuentran prácticamente en el chip y el tiempo necesario para accederlas es casi nulo. También puede utilizarse el conocimiento que se tiene acerca de la lógica del programa, para diseñar warps cuyos hilos lean en localidades aproximadas en memoria, así el caché L1 contendrá los datos sin necesidad de recurrir a la memoria global.

Otra de las causas de un mal rendimiento a causa de la memoria, es la trasferencia de datos entre el equipo anfitrión y el dispositivo, lo que implica que la copia de información demorará el inicio de la ejecución y la recuperación de resultados. Algunas de las soluciones para enfrentar este problema pueden ser:

- Modificar el código de tal manera que no sean necesarios demasiados datos iniciales o que estos puedan generarse directamente en el dispositivo.
- Realizar una sola trasferencia grande en vez de muchas pequeñas, ya que cada transferencia tiene asociada una sobrecarga.
- CUDA puede declarar memoria en el equipo anfitrión que sea libre de paginación, lo que implica que las copias entre ambos dispositivos pueden realizarse concurrentemente.
- En dispositivos integrados donde el CPU y GPU se encuentran en el mismo chip, la memoria puede ser mapeada, es decir no es necesario reservarla en el dispositivo o realizar copias entre ellos. Esto puede traducirse en un rendimiento considerable.

Finalmente puede utilizarse la memoria constante o de textura para datos que no son modificados durante la ejecución del programa, así se puede usar su caché reduciendo la demanda del ancho de banda en la memoria global.

### 3.4.2. Optimizar el uso de instrucciones

Para optimizar las instrucciones en un programa, se debe de tener un amplio conocimiento de la lógica de este, ya que es necesario realizar modificaciones en el código que pueden influir en el resultado.
Las instrucciones de precisión doble requieren un mayor número de ciclos de reloj para su ejecución que las operaciones de precisión simple, por lo cual como primera mejora podría realizarse el cambio siempre y cuando no se altere el resultado final. A su vez CUDA ofrece funciones comunes optimizadas para el dispositivo como por ejemplo *__fdivide(x,y)* que realiza la operación $\frac{x}{y}$ o *__powf(x,y)* que calcula $x^y$, por ello es conveniente utilizar estas funciones siempre que sea posible.
Instrucciones sin dependencia de datos pueden intercalarse entre aquellas que si la tienen, con el fin de esconder la latencia, es decir de ocultar el retardo producido por la demora en la trasmisión de la información proveniente de la memoria.

Las divergencias que suceden en los hilos de un mismo warp ocasionadas por instrucciones como *if* o *switch* también reducen el rendimiento, porque las ramas de ejecución deben de ser serializadas disminuyendo la cantidad de trabajo hecho por ciclo de reloj. Para evitar este inconveniente se puede utilizar el identificador único del hilo para minimizar el número de ramas divergentes, haciendo que hilos de un mismo warp ejecuten una sola de estas. Cuando no hay forma de evitar la divergencia, debe considerarse utilizar lo menos posible la instrucción de sincronización *__syncthreads*, ya que fuerza al multiprocesador a parar y en consecuencia a ser improductivo.

### 3.4.3. Maximizar la ocupación en GPU

Como se había mencionado anteriormente las arquitecturas definen ciertas constantes que son limitantes en los SM. Como el programa se divide y se mapea a los diferentes

componentes del dispositivo, si se logra maximizar estas constantes se consigue incrementar el nivel de paralelismo. En consecuencia mantener los multiprocesadores ocupados el mayor tiempo posible, puede aumentar el rendimiento. La ocupación se encuentra directamente relacionada con el número de warps residentes en el SM, es decir aquellos a los que se le han asignado recursos de hardware y se encuentran preparados para ser ejecutados.
El número de bloques/warps que pueden residir en un SM, depende de sus características físicas, de la configuración de lanzamiento y los recursos necesarios por cada kernel. Para conocer estos últimos valores, se puede agregar la bandera *-ptxas-options=-v* cuando se compila el programa con *nvcc*, el cual reportará la cantidad de registros, memoria local, memoria compartida y memoria constante necesarios para su ejecución. NVIDIA proporciona una hoja de cálculo para asistir a los programadores en la elección de los mejores parámetros de lanzamiento de kernel, llamada *CUDA Ocupancy Calculator*. La cual puede calcular la relación que existe entre el número de warps activos al ejecutar el kernel, contra el número máximo de warps residentes en el SM. Con esta herramienta se puede ajustar los parámetros de lanzamiento para conseguir una mejor ocupación, que podría originar un mejor rendimiento.

En resumen si un kernel realiza una gran cantidad de operaciones matemáticas, entonces sólo es necesario unos cuantos hilos para ocultar la latencia de las instrucciones, con ello se tiene una baja ocupación sin comprometer el rendimiento[32]. Por otro lado si se realizan numerosos accesos a memoria, se necesita una mayor cantidad de hilos para asegurar que en cualquier momento algunos de ellos se encuentren preparados para ser ejecutados. Ya que la mayoría se quedarán bloqueados esperando información proveniente de la memoria para terminar sus operaciones, lo que sugiere una mayor ocupación.
Para un mejor rendimiento en el marco de trabajo de programación dinámica, se recomienda una mayor ocupación, porque al calcular el valor correspondiente a una casilla de la matriz asociada se generarán dependencias, las cuales se traducirán en accesos a memoria. Esta técnica para ocultar latencia es muy común, tanto que es utilizada en la arquitectura x86_64 (Intel), MIPS (MIPS), Power 5 (IBM), UltraSpark (Sun/Oracle), etc.

# Capítulo 4

# Programación dinámica paralela en las GPU

Día a día el ámbito científico debe enfrentarse a problemas cuyas soluciones o aproximaciones requieren una cantidad enorme de cálculos matemáticos, como por ejemplo la predicción del clima. En la práctica todas estas operaciones son realizadas por computadoras, pero debido a la demanda por procesar grandes cantidades de información en un tiempo razonable y a las restricciones físicas que se han alcanzado, el interés por el cómputo paralelo a aumento considerablemente. Con ello el uso de las GPU para realizar cómputo de propósito general se ha popularizado e incluso se ha logrado reportar un factor de 300 de velocidad por encima de un CPU convencional[8]. Sin embargo a pesar de todos estos avances, sigue siendo labor del programador migrar el código para que pueda ser ejecutado en paralelo por una GPU, tarea que no en todos los casos resulta sencilla. Por ello es necesario ofrecer algunos mecanismos que ayuden en esta tarea.

En este capítulo se expone y se desarrolla el objetivo principal de este trabajo, exhibiendo en su contenido dos ideas que surgen para obtener cierto paralelismo, en problemas cuya solución se puede expresar por medio de la programación dinámica.
La primera idea se basa en generar una gráfica de dependencias a partir de la especificación de la ecuación funcional, con la que se podría obtener un paralelismo casi automático. Sin embargo aunque esta idea parezca bastante alentadora, discutiremos su poca o incluso nula factibilidad, ya que en algunos casos generar dicha gráfica es tan tardado como resolver el problema mismo.
La segunda idea se centra en la definición de ciertos patrones que actúan en la forma de llenar la matriz asociada. Por medio de un pequeño análisis del problema se puede escoger el patrón que mejor se adapte, es decir aquel que incremente su nivel de paralelismo. Estos patrones se incluyen en un marco de trabajo para que pueda ser utilizado por los desarrolladores.
Como se puede observar, esta última idea busca abstraer los detalles técnicos en una arquitectura de GPU, brindando al programador un mecanismo rápido y sencillo para adaptar un problema de programación dinámica de tal manera que explote y aproveche

las características que actualmente ofrece el cómputo de propósito general en las GPU. Se pretende que con sólo unas pocas líneas de código el programador se encuentre disfrutando de los beneficios del cómputo paralelo, sin la necesidad de un amplio conocimiento en este tema.

## 4.1. Gráfica de dependencias

Como se expuso en el capítulo 2, la ecuación funcional de un problema de programación dinámica consta de dos partes fundamentales. En primer lugar una matriz que guarda los resultados de los subproblemas, y segundo una función que indica cómo llenarla. Para cada entrada de la matriz, o se puede calcular su valor de manera independiente o necesita el resultado de algunas otras casillas resueltas previamente. Estas dependencias generan un orden en el cual deben de resolverse los subproblemas, así que a continuación se formalizan algunos conceptos.

**Definición 4.1.** *Sea $f_{M_{n_1\times\cdots\times n_k}}$ la ecuación funcional de un problema de programación dinámica. Se dice que $\Gamma_f(\bar{x}) = \{\bar{y_0}, \ldots, \bar{y_m}\}$ es el conjunto de **dependencias directas** de $\bar{x}$ asociadas a $f$ si y sólo si $f(\bar{x})$ necesita $f(\bar{y_i})$ $\forall_{i\in[0\ldots m]}$ para realizar su cómputo. Con $\bar{x}, \bar{y_i} \in I_{n_1\times\cdots\times n_k}$ $\forall_{i\in[0\ldots m]}$, $\Gamma_f : I_{n_1\times\cdots\times n_k} \to 2^{I_{n_1\times\cdots\times n_k}}$.*

**Definición 4.2.** *Se denota como $\Gamma_f^+(\bar{x})$ al conjunto de **dependencias absolutas** de $\bar{x}$ asociadas a $f$, donde $\Gamma_f^+(\bar{x}) = \Gamma_f(\bar{x}) \cup \Gamma_f^+(\bar{y})$, $\forall_{\bar{y}\in\Gamma_f(\bar{x})}$.*

**Definición 4.3.** *Se dice que $\bar{x}$ es una **dependencia cíclica** si $\bar{x} \in \Gamma_f^+(\bar{x})$*

**Definición 4.4.** *Sea $f_{M_{n_1\times\cdots\times n_k}}$ la ecuación funcional de un problema de programación dinámica. La **gráfica de dependencias** asociada a $f$ denotada por $G_f = (V, E)$, tiene como conjunto de vértices a $V = \{\bar{x} | \bar{x} \in I_{n_1\times\cdots\times n_k}\}$ y como conjunto de aristas a $E = \{(\bar{x}, \bar{y}) | \bar{x}, \bar{y} \in V,\ \bar{y} \in \Gamma_f(\bar{x})\}$.*

**Definición 4.5.** *En una gráfica de dependencias el **exgrado** de un vértice $\bar{x}$ es el número de aristas que tienen a $\bar{x}$ como vértice inicial, denotado por $g^+(\bar{x})$.*

**Definición 4.6.** *En una gráfica de dependencias se dice que el vértice $\bar{x}$ es un **caso base** si y sólo si $g^+(\bar{x}) = 0$.*

**Definición 4.7.** *Sea $G_f = (V, E)$, la **profundidad de un vértice** $\bar{x}$ es igual a la longitud de la trayectoria más larga desde $\bar{x}$ a un caso base, denotada por $\mathcal{D}(\bar{x})$. Con ello, $\mathcal{D}(G_f) = max_{\bar{x}\in V}(\mathcal{D}(\bar{x}))$ es la profundidad de la gráfica $G_f$.*

Con estas definiciones hay que observar que una ecuación funcional (definición 2.2) no puede tener dependencias cíclicas, ya que si esto no se cumple, existirían subproblemas para los cuales es imposible obtener su resultado, así la gráfica de dependencias resulta una gráfica dirigida acíclica. También se puede notar que el conjunto de los casos base son instancias de subproblemas independientes entre sí, porque no requieren de la solución de algún otro para calcular su resultado, lo que implica que pueden resolverse de manera

paralela por diferentes hilos. Posteriormente podemos eliminar estos casos base de la gráfica original y así obtener una nueva gráfica de dependencias que representa el estado actual de la resolución del problema original. Este proceso continúa hasta que la gráfica sea nula, es decir que el conjunto de vértices sea vacío lo que resultaría en la resolución de todas las instancias de los subproblemas.

**Definición 4.8.** *Sea $G_f = (V, E)$, se llama **etapa** a cualquier $S_i$, donde $S_i$ es el conjunto de vértices de $G_f$ tal que su profundidad es $i$, es decir $S_i = \{x \in V | \mathcal{D}(x) = i\}$*

**Definición 4.9.** *Sea $G_f = (V, E)$, se dice que la secuencia de etapas $S_0 \dots S_k$ de $G_f$ es mínima si $k = \mathcal{D}(G_f)$*

Al generar la secuencia de etapas mínima de una gráfica de dependencias, se obtiene un orden en el cual pueden resolverse los subproblemas de manera paralela. A continuación se formaliza esta idea, presentando un algoritmo que genera dicha secuencia:

**Algorithm 1** Secuencia de etapas mínima de una gráfica de dependencias

**Require:** $G_f = (V, E)$.
1: $P := \emptyset$
2: $R := V$
3: $i := 0$
4: **while** $R \neq \emptyset$ **do**
5: $H := G[R]$ {Gráfica inducida por los vértices en $R$}
6: $S_i := \{x | x$ es un caso base de $H\}$
7: $P := P \cup S_i$
8: $R := V/P$
9: $i := i + 1$
10: **end while**
11: **return** $S_0, S_1, \dots, S_{i-1}$

Dado el algoritmo anterior se demostrará que genera la mínima secuencia de etapas para una gráfica de dependencias.

**Observación 4.1.** *Sea $G_f = (V, E)$, $\bar{x}$ es un caso base de $G_f$ si y sólo si $\mathcal{D}(\bar{x}) = 0$*

*Demostración.* $\Rightarrow$)
Como $\bar{x}$ es un caso base, por definición $g^+(\bar{x}) = 0$, lo que implica que no existe arista de la forma $(\bar{x}, \bar{y})$ en $G_f$, por lo que ningún vértice es alcanzable desde $\bar{x}$. Entonces la trayectoria más larga desde $\bar{x}$ a cualquier otro vértice es 0, en particular a el mismo que es un caso base, por lo tanto $\mathcal{D}(\bar{x}) = 0$.
$\Leftarrow$)
Se sabe que $\mathcal{D}(\bar{x}) = 0$. Si suponemos que existe una arista de la forma $(\bar{x}, \bar{y})$ en $G_f$, entonces podemos tomar la trayectoria más larga de $\bar{y}$ a un caso base. Sea $\bar{y}y_0 \dots y_k$ esa trayectoria. Si a continuación construimos lo siguiente $t = \bar{x}yy_0 \dots y_k$ nos damos cuenta

de que $t$ es una trayectoria de $\bar{x}$ a un caso base con una longitud al menos de uno, lo que contradice la hipótesis. Por lo tanto no existe arista de la forma $(\bar{x}, \bar{y})$ en $G_f$, entonces $g^+(\bar{x}) = 0$ lo que implica que $\bar{x}$ es un caso base. □

**Lemma 4.1.** *Sea $G_f = (V, E)$, al finalizar la iteración $i$-ésima del algoritmo 1 sobre $G_f$, $S_i$ contendría todos los vértices que se encuentran a profundidad $i$ en $G_f$, es decir $S_i$ es una etapa.*

*Demostración.* Por inducción sobre $i$.
**Caso base:**
Al realizar la iteración 0, se tiene que al principio $R = V$, entonces al ejecutar la instrucción 5, $H = G_f$ y $S_0$ es el conjunto de todos los casos base de $G_f$, pero por la observación 4.1 $S_0$ es equivalente al conjunto de todos los vértices cuya profundidad en $G_f$ es cero, es decir $S_0 = \{x \in V | \mathcal{D}(x) = 0\}$.
Cuando se ejecutan las siguientes líneas (7 a 9) se tiene que $P = S_0$, $R = V/S_0$ e $i = 1$, por lo que al finalizar la iteración 0, $S_0$ es una etapa.
**Hipótesis de Inducción:** Suponemos que al finalizar la iteración $k$, $S_j$ es una etapa para $\forall j <= k$.
**Paso inductivo:** Se demostrará que al finalizar la iteración $(k+1)$, $S_{k+1}$ es una etapa.
Por hipótesis de inducción al finalizar la iteración $k$, $S_j$ es una etapa $\forall_{j \in [0...k]}$. Observemos que $P = \cup_{j=0}^{k} S_j$ , por lo que contiene a todos los vértices que están a profundidad a lo más $k$ en $G_f$ y como $R$ es el complemento de $P$ entonces $R$ contiene a todos los vértices que están a profundidad estrictamente mayor que $k$ en $G_f$.
Al iniciar la iteración $(k+1)$, $H$ es la gráfica inducida por $R$, y $S_{k+1}$ son los casos base de $H$, vamos a ver que todos los vértices de profundidad $(k+1)$ en $G_f$ están contenidos en $S_{k+1}$ y sólo esos.
Sea $u \in V$ tal que $\mathcal{D}(u) = k + 1$, como se había observado $R$ contiene a los vértices que están a profundidad estrictamente mayor que $k$ en $G_f$ por lo que $u \in R$. Ahora supongamos que existe la arista $(u, v)$ contenida en $H$ para algún $v \in R$, entonces la profundidad de $v$ en $G_f$ también es mayor que $k$, por lo que $\mathcal{D}(v) = l$ con $l > k$. Sea $t = vv_1 \dots v_{l-1}$ la trayectoria más larga de $v$ a un caso base de $G_f$, entonces podemos construir a $t' = uvv_1 \dots v_{l-1}$ una trayectoria de va desde $u$ a un caso base cuya longitud es al menos $k + 2$, pero se había supuesto $\mathcal{D}(u) = k + 1$, por lo que se tiene una contradicción. Por lo tanto no existe arista $(u, v)$ contenida en $H$ para algún $v \in R$.
Como no existe tal arista quiere decir que $g^+(u) = 0$ en $H$, es decir $u$ es un caso base de $H$ y por lo tanto $u \in S_{k+1}$.
Ahora sea $u \in V$ tal que $\mathcal{D}(u) = l$ con $l > k + 1$, entonces $u \in R$ y $g^+(u) \neq 0$ en $G_f$ ya que en particular $l > 1$, lo que implica que existe una arista del tipo $(u, v)$ en $G_f$. Si suponemos que no existe una arista $(u, v)$ contenida $H$ para algún $v \in R$, entonces $v$ tiene que estar en el conjunto de vértice del complemento de $H$ que es $P$, pero este contiene todos los vértices cuya profundidad es a lo más $k$ en $G_f$, por lo que todas las trayectorias de $u$ hacia un caso base de $G_f$ tendrían longitud a lo más $(k + 1)$, contradiciendo la hipótesis. Por lo tanto existe al menos una arista $(u, v)$ contenida en $H$ con $v \in R$. Pero si existe esta arista quiere decir que $g^+(u) \neq 0$ en $H$, por lo que $u$ no es un

caso base de $H$ y por lo tanto $u \notin S_{k+1}$.
Al ejecutar las instrucciones restantes (líneas 7 a 9) no modifican a $S_{k+1}$, por lo tanto al finalizar la iteración $(k+1)$, $S_{k+1} = \{x \in V | \mathcal{D}(x) = k+1\}$. □

**Corolario 4.1.** *Sea $G_f = (V, E)$, si $\mathcal{D}(G_f) = k$ entonces al ejecutar el algoritmo 1 sobre $G_f$ este generará la secuencia de etapas $S_0 \dots S_k$ para la gráfica.*

*Demostración.* Ya que $\mathcal{D}(G_f) = k$, por definición existe $\bar{x} \in V$ tal que $\mathcal{D}(\bar{x}) = k$, por el Lemma 4.1 al finalizar la $k$-ésima iteración del algoritmo $\bar{x} \in S_k$ por lo que $S_k \neq \emptyset$ y $P = \bigcup_{j=0}^{k} S_j$. Entonces $P$ contiene a todos los vértices con profundidad menor o igual a $k$ en $G_f$, lo que implica que $R = V/P = \emptyset$, ya que si no fuera de esta manera, $R$ contendría a todos los vértices de $G_f$ con profundidad mayor que $k$, lo que contradice la hipótesis $\mathcal{D}(G_f) = k$. Por lo tanto al iniciar la iteración $(k+1)$ la condición $R \neq \emptyset$ no se cumple, terminando el algoritmo y regresando la secuencia $S_0 \dots S_k$ □

Una vez que se tiene la secuencia de etapas mínima de la gráfica de dependencias se puede proceder a llenar la matriz asociada de la ecuación funcional en forma paralela, a lo cual podemos proceder como sigue:

**Algorithm 2** Llenar en paralelo la matriz asociada de una ecuación funcional.

**Require:** $f_{M_{n_1 \times \cdots \times n_k}}$ la ecuación funcional y $S_0, \dots, S_z$ la secuencia de etapas mínima de la gráfica de dependencias.

```
1: for i = 0 to z do
2:    do parallel
3:    take x̄ ∈ S_i do M[x̄] = f(x̄)
4:    end do
5: end for
```

Si $\bar{x}$ es el vértice de la gráfica de dependencias que representa la solución del problema original, entonces surge la idea de que no es necesario construir toda la gráfica, sino que puede ser redefinida de la siguiente manera, $G_f(V', E')$ donde $V' = \Gamma_f^+(\bar{x})$ y $E' = \{(u, v) | u, v \in V', v \in \Gamma_f(u)\}$. Es claro que esta nueva gráfica de dependencias que llamaremos *generada por la solución*, es una subgráfica de la que se definió originalmente. De manera natural estas dos gráficas son potencialmente la misma, como por ejemplo en el problema de la longitud de la subsecuencia común más larga.

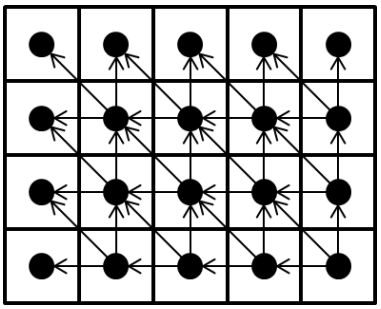

Figura 4.1: La gráfica de dependencias y la gráfica generada por la solución, son exactamente la misma para el problema de la subsecuencia común más larga.

Como se puede observar, generar la gráfica de dependencias y obtener la secuencia de etapas, demandan tanto memoria como tiempo adicional de procesamiento. Por ello es necesario analizar estos dos parámetros para determinar la factibilidad de construir explícitamente la gráfica de dependencias.

### 4.1.1. Consideración en memoria

Generar la gráfica de dependencias requiere de memoria adicional, la cual se encuentra ligada al número de vértices y al número de aristas que contiene. De la forma en cómo se definió la gráfica de dependencias el número de nodos es igual a la cardinalidad del conjunto de todos los índices de la matriz asociada, por lo cual si esta tiene dimensión $n_1 \times \cdots \times n_k$ el número de nodos en la gráfica es igual a $|V| = n_1 * \cdots * n_k$.
El número máximo de aristas que puede tener una gráfica dirigida acíclica con $n$ vértices es $\frac{n*(n-1)}{2}$. La explicación surge a partir de que cualquier gráfica dirigida tiene a lo más $n^2$ aristas. Sin embargo debido a que no existen ciclos, es necesario quitar todas las aristas lazo (aquellas hacia un mismo nodo) y para cualesquiera dos vértices sólo una de las aristas $(x, y)$ o $(y, x)$ puede ser incluida, lo cual genera $|E| = \frac{n^2-n}{2} = \frac{n*(n-1)}{2}$ número de aristas.
Si se considera que cada vértice guarda un valor que requiere memoria constante $O(k)$, entonces generar la gráfica de dependencias requiere a lo más $O(|V| + |E| + k * |V|) \cong O(|V| + \frac{|V|*(|V|-1)}{2} + k * V) \cong O(|V|^2)$ memoria.
El mejor caso es cuando cada vértice no tiene dependencias, es decir no existen aristas en la gráfica, por lo tanto se requiere de al menos $O(|V|)$ memoria.

### 4.1.2. Consideración en tiempo

Llenar la matriz asociada secuencialmente, implica al menos recorrer cada una de sus entradas calculando su respectivo valor. Por lo cual si se suma el costo en tiempo de evaluar cada casilla, se obtiene el tiempo total de procesamiento. Con ello se tiene que $\sum_{\bar{x} \in V} c(\bar{x}) = W$, donde $V \leq W$ ya que el costo de calcular cada casilla es al menos tiempo constante.

Por otro lado si se genera la gráfica de dependencias explícitamente, se tiene que crear todos sus nodos y aristas, que por lo discutido en la sección anterior se encuentran en el rango de $V$ a $V^2$. Esto mismo sucede al generar la secuencia de etapas, ya que al final del proceso se habrán examinado todos los vértices para su respectiva clasificación y para cada uno de ellos se eliminaron todas sus dependencias en la gráfica. Finalmente se procede a llenar la matriz asociada, pero la secuencia de etapas sólo genera un orden de los vértices, lo que implica que aún se tiene que recorrer toda la matriz calculando sus entradas, que se había supuesto en el orden de $W$.
Con esto último es fácil observar que generar la gráfica de dependencias secuencialmente no tiene un impacto en la reducción del tiempo de procesamiento, más bien genera un costo administrativo que se encuentra en el rango de $V$ a $V^2$.

Suponiendo que se tiene un número ilimitado de hilos, se puede construir la gráfica de dependencias lanzando exactamente $V$, donde cada uno construirá su vértice respectivo y creará sus dependencias. Por lo tanto en el peor de los casos construir la gráfica en paralelo nos toma $O(|V|)$. Por un momento se dará por hecho que ya se generó la secuencia de etapas con longitud $k$, lo que implica que calcular las entradas de la matriz asociada en paralelo toma al menos $O(k*w+|V|)$ donde $w$ es el máximo costo en tiempo al evaluar un casilla.
Por otro lado si se crean exactamente $V$ hilos en la matriz, donde cada uno verificará si sus dependencias se encuentran resueltas y en dado caso proceder a calcular el resultado de su casilla, entonces en $k$ iteraciones se habrán resuelto todas las entradas de la matriz. Ya que debe de corresponder al número de etapas que generó la gráfica de dependencias, por lo que llenar la matriz asociada de esta manera toma $O(k*(|V|+w))$. Si $k$ es constante entonces $O(k*w+|V|) \cong O(k*(|V|+w))$. Si no lo es entonces su valor máximo es $|V|$ ya que corresponde al número máximo de etapas que se pueden generar, pero si esto sucede entonces cada vértice sólo tiene una dependencia que apunta exactamente al vértice de la etapa anterior, por lo tanto se tiene que $O(|V|*w+1) \cong O(|V|*(1+w))$. De esta manera se muestra que generar la gráfica de dependencias en paralelo sigue teniendo un costo administrativo y consumo de memoria, lo cual puede suprimirse iterando directamente sobre la matriz.
Se concluye que generar la gráfica de dependencias, no tiene un impacto en la reducción del tiempo de procesamiento al llenar la matriz asociada de una ecuación funcional.

**Algorithm 3** Llenar en paralelo la matriz asociada de una ecuación funcional.

**Require:** $f_{M_{n\times m}}$

1: $V := n * m$
2: **while** $V > 0$ **do**
3: **do parallel**
4: $(i, j) := (\frac{threadIdGlobal}{m}, \text{threadIdGlobal}\,\%m)$
5: **if** (i, j)$\neq$ NULL and all dependencies of $(i, j)$ are done **then**
6: $M[i, j] := f(i, j)$
7: $V := V - 1$
8: **end if**
9: **end while**

## 4.2. Marco de trabajo

En este capítulo se definirán ciertos patrones que ayudan a calcular los valores de la matriz asociada de una ecuación funcional de manera paralela. Basándose en la naturaleza de su recursión pero sin la necesidad de construir explícitamente la gráfica de dependencias; ahorrando con ello tiempo y memoria. En términos prácticos y de simplicidad, este trabajo se restringirá únicamente en ecuaciones funcionales cuyas matrices asociadas son bidimensionales, aunque esto no debe de ser un factor limitante ya que se podría transformar una matriz $k$ dimensional en otra bidimensional, simplemente encontrando las funciones de redireccionamiento en el dominio de los índices.

Los patrones que se desarrollaron surgen de manera natural al resolver muchos problemas con programación dinámica, los cuales se encapsulan en un marco de trabajo lo más general posible, con la finalidad de ofrecer un mecanismo a los programadores para que las ecuaciones funcionales puedan ser ejecutadas de manera paralela utilizando las GPU. Se debe de tener en cuenta que la idea principal de este trabajo no es caracterizar a todos los problemas de programación dinámica, por lo que pueden existir ecuaciones funcionales que no se adecuen a nuestros patrones.

Lo primero que se tiene que hacer es utilizar un dominio de índices bidimensionales, los cuales utilizando notación anterior pueden representarse por medio de:

$$I_{n\times m} = \{(x, y)|x, y \in N, x < n, y < m\} \tag{4.1}$$

Posteriormente se define el concepto de *regiones adyacentes*, para un $(x_1, y_1) \in I_{n\times m}$ como sigue:

- $Left((x_1, y_1)) = \{(x_2, y_2) \in I_{n\times m}|x_2 = x_1, y_2 < y_1\}$
- $Right((x_1, y_1)) = \{(x_2, y_2) \in I_{n\times m}|x_2 = x_1, y_2 > y_1\}$
- $Up((x_1, y_1)) = \{(x_2, y_2) \in I_{n\times m}|x_2 < x_1, y_2 = y_1\}$

- $Down((x_1, y_1)) = \{(x_2, y_2) \in I_{n\times m} | x_2 > x_1, y_2 = y_1\}$
- $UpLeft((x_1, y_1)) = \{(x_2, y_2) \in I_{n\times m} | x_2 < x_1, y_2 < y_1\}$
- $UpRight((x_1, y_1)) = \{(x_2, y_2) \in I_{n\times m} | x_2 < x_1, y_2 > y_1\}$
- $DownLeft((x_1, y_1)) = \{(x_2, y_2) \in I_{n\times m} | x_2 > x_1, y_2 < y_1\}$
- $DownRight((x_1, y_1)) = \{(x_2, y_2) \in I_{n\times m} | x_2 > x_1, y_2 > y_1\}$

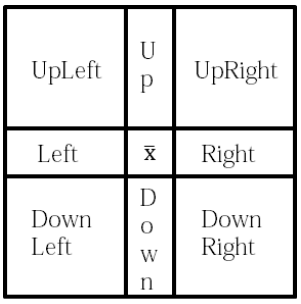


Figura 4.2: Especificación de regiones para un índice $\bar{x}$ bidimensional.

En las próximas dos secciones se definen clases de ecuaciones funcionales, cuyo conjunto de dependencias absolutas siempre se encuentra en ciertas regiones previamente establecidas. De esta manera se obtiene una estructura en sus dependencias, la cual puede ser utilizada para identificar subproblemas independientes y así poder resolverlos de manera paralela. Se presentarán en total ocho clases de funciones, que corresponden a las formas más naturales en las que se puede ir llenando los valores de la matriz asociada, que son: por renglones, columnas o diagonales, en cualquiera de sus dos direcciones. Cada patrón está asociado a una clase, por lo cual se dirá que una ecuación funcional cumple con ese patrón, si se encuentra contenida en su clase correspondiente.

### 4.2.1. Patrones por renglones/columnas

Las primeras clases a presentar, corresponden a las ecuaciones funcionales cuya matriz asociada puede llenarse de manera paralela, iterando por renglones (de arriba hacia abajo y viceversa) o columnas (de izquierda a derecha y a la inversa). Las cuales se precisan de la siguiente manera:

- RUD $=\{f_{M_{n\times m}} | \forall_{\bar{x}\in I_{n\times m}}, \Gamma_f^+(\bar{x}) \subseteq UpLeft(\bar{x}) \cup Up(\bar{x}) \cup UpRigth(\bar{x})\}$
- RDU $= \{f_{M_{n\times m}} | \forall_{\bar{x}\in I_{n\times m}}, \Gamma_f^+(\bar{x}) \subseteq DownLeft(\bar{x}) \cup Down(\bar{x}) \cup DownRigth(\bar{x})\}$
- CLR $= \{f_{M_{n\times m}} | \forall_{\bar{x}\in I_{n\times m}}, \Gamma_f^+(\bar{x}) \subseteq UpLeft(\bar{x}) \cup Left(\bar{x}) \cup DownLeft(\bar{x})\}$
- CRL $= \{f_{M_{n\times m}} | \forall_{\bar{x}\in I_{n\times m}}, \Gamma_f^+(\bar{x}) \subseteq UpRight(\bar{x}) \cup Right(\bar{x}) \cup DownRigth(\bar{x})\}$

Para establecer las ideas se desprende de lo anterior, que por ejemplo *RUD* es la clase que contiene a todas las ecuaciones funcionales, tal que para cualquier índice en la matriz asociada, su conjunto de dependencias absolutas siempre se encuentra en los renglones superiores. El nombre que le asignó a cada clase va a corresponder al patrón correspondiente.

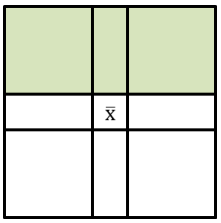


Figura 4.3: Estructura de las dependencias absolutas de una ecuación funcional que cumple el patrón *RUD*.

Una vez que se sabe que estructura tiene la ecuación funcional, hay que especificar como resolver sus instancias de subproblemas de manera paralela. Se describirá este proceso sólo para el patrón *RUD*, ya que los demás son equivalentes a éste en sus rotaciones a 90 grados y por lo tanto los argumentos son análogos.

Lo primero en observar es que si se tiene una ecuación funcional que cumpla con el patrón *RUD* y se llena la matriz de tal manera que para cualquier subproblema residente en el renglón $i$-ésimo, ya se han calculado todas las soluciones de los renglones anteriores; entonces las dependencias de estos subproblemas estarán completamente resueltas y se podrá calcular su valor inmediatamente. Esta idea sugiere que las filas deben de procesarse secuencialmente y los subproblemas integrantes de una misma pueden llevarse a cabo de manera paralela.
El patrón RUD implementado en CUDA, será una función global que recibe la ecuación funcional y el identificador del renglón que se va a procesar, teniendo como efecto calcular todos los valores contenidos en esa misma fila. En teoría cada vez que se ejecuta esta función, deben de lanzarse tantos hilos como columnas en la matriz, con la finalidad de aumentar el nivel de paralelismo. Sin embargo, en la práctica esto no siempre es posible, debido a que las dimensiones de una matriz para un problema real, llegan a superar los límites físicos de los dispositivos. Para solucionar este inconveniente un mismo hilo puede encargarse de calcular más de un resultado. Otra de los aspectos a tomar en cuenta, es que tanto la matriz como la función que calcula sus entradas deben de residir en el dispositivo, realizando a su paso las inicializaciones y copias respectivas, trabajo administrativo que se intenta esconder en el marco de trabajo.

A continuación se muestra la implementación en seudocódigo del patrón RUD. Por las razones que se discutieron en la sección 3.4.3, los parámetros de lanzamiento del kernel deben de ajustarse de tal manera que se tenga la mejor ocupación posible.

```
__device__ T f_eq(int i, int j){...}

__global__ void RUD(f, M, row){
  int id = threadIdx.x + blockIdx.x*blockDim.x;
  for(;id < M.m; id+= blockDim.x*gridDim.x)
    M[row, id] = f(row, id);
}

int main(){
  M_dev[n,m];
  for(int i=0; i < n; i++)
    RUD<<<...>>>(f_eq, M_dev, i);
  return EXIT_SUCCESS;
}
```

### 4.2.2. Patrones por diagonales

Las siguientes clases, abstraen el concepto de aquellas ecuaciones funcionales cuya matriz asociada puede llenarse iterando por diagonales, ya sea dirección *noreste-suroeste*, *noroeste-sureste* y sus viceversas. Estas nuevas clases se definen formalmente como sigue:

- SONE $= \{f_{M_{n\times m}} | \forall_{\bar{x}\in I_{n\times m}}, \Gamma_f^+(\bar{x}) \subseteq Left(\bar{x}) \cup DownLeft(\bar{x}) \cup Down(\bar{x})\}$
- NESO $= \{f_{M_{n\times m}} | \forall_{\bar{x}\in I_{n\times m}}, \Gamma_f^+(\bar{x}) \subseteq Up(\bar{x}) \cup UpRight(\bar{x}) \cup Rigth(\bar{x})\}$
- NOSE $= \{f_{M_{n\times m}} | \forall_{\bar{x}\in I_{n\times m}}, \Gamma_f^+(\bar{x}) \subseteq UpLeft(\bar{x}) \cup Up(\bar{x}) \cup Left(\bar{x})\}$
- SENO $= \{f_{M_{n\times m}} | \forall_{\bar{x}\in I_{n\times m}}, \Gamma_f^+(\bar{x}) \subseteq Right(\bar{x}) \cup DownRight(\bar{x}) \cup Down(\bar{x})\}$

Al igual que en la sección anterior sólo se expondrá la implementación para el patrón *NOSE*, ya que los demás son equivalentes a éste. La diagonal $i$-ésima va a corresponder a todos los subproblemas que al sumar el identificador de su renglón más el de su columna es igual a $i$. Si se tiene una ecuación funcional que cumple el patrón *NOSE*, entonces se puede llenar su matriz asociada de tal manera que antes de procesar la diagonal $i$-ésima, se calculen todos los resultados de las diagonales anteriores. Esto se debe a que si se tiene un índice $(x, y)$ en la diagonal $i$-ésima, se cumple que $x + y = i$, entonces cualquier otro índice $(x', y')$ en sus regiones de dependencias, satisfacen $x' <= x$ y $y' < y$ o $x' < x$ y $y' <= y$, lo que implica que en ambos casos $x' + y' < x + y = i$. Por lo tanto cualquier subproblema en las regiones de dependencias, pertenece a una diagonal anterior y ya se ha resuelto previamente.

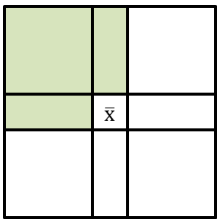


Figura 4.4: Estructura de las dependencias absolutas de una ecuación funcional que cumple el patrón *NOSE*.

Para realizar la implementación en CUDA, se tiene que procesar secuencialmente el conjunto de diagonales empezando por la cero y terminando en la $(n + m) - 2$. Al igual que en la sección anterior el patrón *NOSE* es una función global que recibe la ecuación funcional y el identificador de la diagonal a procesar. En seguida se introduce el seudocódigo para el patrón *NOSE*:

```
1  __device__ T f_eq(int i, int j){...}
2
3  __global__ void NOSE(f, M, diag){
4    int id = threadIdx.x + blockIdx.x*blockDim.x;
5    id+=(diag<M.m) ? 0: (diag -(M.m-1));
6    for(;(diag-id>=0)&& id < M.n; id+= blockDim.x*gridDim.x)
7      M[id, diag-id] = f(id, diag-id);
8  }
9
10 int main(){
11   M_dev[n,m];
12   for(int i=0; i <= (n+m)-2; i++)
13     NOSE<<<...>>>(f_eq, M_dev, i);
14   return EXIT_SUCCESS;
15 }
```

### 4.2.3. Patrones en algunos algoritmos de programación dinámica

Los patrones presentados pueden ser utilizados en muchos problemas de programación dinámica, y para ejemplo se muestran algunos de ellos junto con su respectivo patrón, en un breve compendio extraído de la amplia gama de posibilidades. Las descripciones de dichos problemas pueden encontrarse en [18]. De los cuarenta y siete problemas totales que aborda el libro, se pudieron identificar fácilmente a treinta y uno de ellos que cumplen con alguno de los patrones presentados.

| **Nombre del Problema** | **Patron** |
|---|---|
| Binomial Coefficient | RUD |
| 0/1 Knapsack Problem | RUD |
| Longest Common Subsequence | NOSE |
| Matrix Chain Multiplication Problem | SONE |
| Edit Distance Problem | NOSE |
| Optimal Allotment | RDU |
| Optimal Alphabetic Radix-Code Tree | SONE |
| Assembly Line Balancing | RDU |
| Optimal Assignment Problem | RDU |
| Optimal Binary Search Tree Problem | SONE |
| Optimal Covering Problem | RUD |
| Deadline Scheduling Problem | RDU |
| Discounted Profits Problem | RDU |
| Fibonacci Recurrence Relation | CLR |
| Integer Linear Programming | RDU |
| Inventory Problem | RDU |
| Optimal Investment Problem | RDU |
| Investment: Winning in Las Vegas Problem | RDU |
| COV as KSINT Problem | RDU |
| Integer Knapsack Problem | RUD |
| Minimum Maximum Problem | RDU |
| Minimum Weight Spanning Tree Problem | CRL |
| The Game of NIM | CLR |
| Optimal Distribution Problem | RDU |
| Optimal Production Problem | RDU |
| Reliability Design Problem | RUD |
| Replacement Problem | CRL |
| Stagecoach Problem | RDU |
| Segmented Curve Fitting Problem | RUD |
| Process Scheduling Problem | RDU |
| Transportation Problem | RDU |
| Shortest Path in an Cyclic Graph | CLR |

## 4.3. Descripción de la implementación del marco de trabajo

La idea principal del marco de trabajo es proveer un mecanismo que por medio de la definición de una ecuación funcional y la especificación de un patrón, se puedan calcular las entradas de la matriz asociada de manera paralela en un GPU. Las funciones que implementan los patrones deben de realizar todas las acciones necesarias para llevar a

cabo este cometido, ocultando la mayor parte de los aspectos técnicos de la arquitectura. En esta sección se exhibirán los principales elementos que constituyen el marco de trabajo, junto con una breve descripción de su propósito.
El primero de ellos es una clase de C++ que representa una matriz con dimensión $n \times m$, la cual servirá para albergar todos los resultados de los subproblemas. Esta matriz será manipulada en el dispositivo, pero si el usuario lo quiere puede copiarse al equipo anfitrión, por lo que contiene métodos que pueden ser utilizados en los dos ámbitos. De manera adicional, se incorpora una pequeña flexibilidad por medio de la opción *type*, que indica la forma en cómo se guardará la matriz, de esta manera hilos con índices consecutivos acceden siempre a localidades contiguas de memoria, con la finalidad de aprovechar la caché. En seguida se ilustra el diagrama de la clase *Matrix*:

```
1  enum {ROWS, COLUMS, DIAGSNOSE, DIAGSSONE};
2
3  template <typename T>
4  class Matrix{
5    private:
6      //Forma de guardar la matrix que dependera de la forma de iterar
7      unsigned int type;
8
9    public:
10     //Espacio de memoria para guardar los resultados
11     T* M;
12
13     //Dimensiones de la matriz
14     unsigned int n;
15     unsigned int m;
16
17     // Constuctor de la matriz
18     Matrix(const unsigned int n, const unsigned int m, unsigned int type);
19
20     // Establece un valor en la posicion (i,j) de la matriz
21     inline __device__ __host__ void setAt(unsigned int i, unsigned int j, T
           value);
22
23     // Obtiene el valor de la posicion (i,j) en la matriz
24     inline __device__ __host__ T getAt(unsigned int i, unsigned int j);
25
26     // Redirecciona la posicion (i,j) a un indice unidimensional
27     inline __device__ __host__ int getIndex(unsigned int i, unsigned int j);
28
29     // Libera memoria
30     ~Matrix();
31 };
```

Como segundo elemento se creó una estructura llamada *Vars*, la cual guarda algunos

apuntadores a las variables más importantes utilizadas en el marco de trabajo, como por ejemplo la dirección de memoria del arreglo de la matriz o el apuntador hacia los parámetros extra en el dispositivo.

```
1  template <typename F_TYPE, typename T, class U>
2  struct Vars{
3    //Parametros extras en el Host.
4    U* ex_p_host;
5    //Tamanio de los parametros extras.
6    size_t size_ex_p;
7    //Parametros extras en el Device.
8    U* ex_p_device;
9    //Memoria para la Matrix en el Host
10   Matrix<T> *M_host;
11   //Memoria para la Matrix en el Device
12   Matrix<T> *M_device;
13   //Memoria para el arreglo en el Device.
14   T *arr_M_device;
15   //Funcion a ejecutar en el device
16   F_TYPE* dev_f;
17 };
```

Posteriormente se declararon tres funciones que se encargan de realizar la parte administrativa del marco de trabajo. Una de ellas se ocupa de los parámetros extra, que es información adicional requerida en la ecuación funcional. Entre sus deberes se encuentra reservar memoria para los parámetros extra en el dispositivo y llevar a cabo la copia pertinente de información.
La siguiente se ocupa de crear la matriz que se alojará en el dispositivo e inicializar todos los apuntadores incluidos en la estructura *Vars*, para luego regresar una instancia como resultado.
Finalmente la última función, sincroniza todos los kernels que se hayan ejecutado y libera los recursos utilizados. A esta función se le añade la característica de copiar tanto los parámetros extra, como la matriz de resultados de regreso al equipo anfitrión, por medio de la opción *copyBack_ex_p* y *copyBack_Matrix* respectivamente. De esta manera se pueden preservar los posibles cambios realizados. Las firmas de las tres funciones se muestran en seguida:

```
1  /*Recibe los parametros extras declarados en Host, junto con su tamanio y
       regresa un apuntador de los mismos pero en el Device*/
2  template <class U>
3  U* manage_ExtraParams(U* ex_p_host, size_t size_ex_p);
4
5  /*Inicializa todos los apuntadores para una instancia de Vars, la cual se
       regresa como resultado*/
6  template <typename F_TYPE, typename T, class U>
7  Vars<F_TYPE, T, U>* setUpVars(DimMatrix<T>* dim, F_TYPE* f, U* ex_p_host,
       size_t size_ex_p, unsigned int type);
8
9  /*Sincroniza los kernels, libera los recursos y si es necesario copia
       resultados devuelta al Host*/
10 template <typename F_TYPE, typename T, class U>
11 Matrix<T>* synchCopyFree(Vars<F_TYPE, T, U>* vars, bool copyBack_Matrix,
       bool copyBack_ex_p);
```

Como último elemento se muestran las funciones que implementan los patrones descritos en las secciones 4.2.1 y 4.2.2. Se debe recordar que estos corresponden a las formas más naturales de iterar la matriz asociada en cualquiera de sus dos direcciones. Por ejemplo el patrón RUD y RDU iteran sobre renglones, pero el primero lo hace de arriba hacia abajo y el otro de abajo hacia arriba. Pensando en ello se crearon cuatro funciones, que reciben como parámetro el identificador de un renglón, columna o diagonal tanto de inicio como de final, y dependiendo del orden se itera en una dirección o en su contraria. Estas funciones inicializan todo lo necesario para ejecutar los kernels que calculan los valores de la matriz asociada siguiendo un cierto patrón. Cada llamada a kernel llena de forma paralela el renglón, columna o diagonal que le sea especificado.

```
1  //Patron por diagonales, Noroeste-Sureste(NOSE) o Sureste-Noroeste(SENO)
2  template <typename F_TYPE, typename T, class U>
3  Matrix<T>* dp_by_diagonals_NO_SE(DimMatrix<T>* dim, F_TYPE* f, void
       (*f_result)(Matrix<T>*, U*), U* ex_p_host, size_t size_ex_p, int
       diagInit, int diagEnd, bool copyBack_Matrix, bool copyBack_ex_p);
4
5  template <typename F_TYPE, typename T, class U>
6  __global__ void dp_by_diagonals_NO_SE_stage(F_TYPE dev_f, Matrix<T>*
       M_device, U* ex_p_device, int diag);
7
8  //Patron por diagonales, Suroeste-Noreste(SONE) o Noreste-Suroeste(NESO)
9  template <typename F_TYPE, typename T, class U>
10 Matrix<T>* dp_by_diagonals_SO_NE(DimMatrix<T>* dim, F_TYPE* f, void
       (*f_result)(Matrix<T>*, U*), U* ex_p_host, size_t size_ex_p, int
       diagInit, int diagEnd, bool copyBack_Matrix, bool copyBack_ex_p);
11
12 template <typename F_TYPE, typename T, class U>
13 __global__ void dp_by_diagonals_SO_NE_stage(F_TYPE dev_f, Matrix<T>* M, U*
       ex_p, int diag);
14
15 //Patron por renglones, Arriba-Abajo(RUD) o Abajo-Arriba(RDU)
16 template <typename F_TYPE, typename T, class U>
17 Matrix<T>* dp_by_rows(DimMatrix<T>* dim, F_TYPE* f, void
       (*f_result)(Matrix<T>*, U*), U* ex_p_host, size_t size_ex_p, int
       rowInit, int rowEnd, bool copyBack_Matrix, bool copyBack_ex_p);
18
19 template <typename F_TYPE, typename T, class U>
20 __global__ void dp_by_rows_stage(F_TYPE dev_f, Matrix<T>* M_device, U*
       ex_p_device, int row);
21
22 //Patron por columnas, Izquierda-Derecha(CLR) o Derecha-Izquierda(CRL)
23 template <typename F_TYPE, typename T, class U>
24 Matrix<T>* dp_by_colums(DimMatrix<T>* dim, F_TYPE* f, void
       (*f_result)(Matrix<T>*, U*), U* ex_p_host, size_t size_ex_p, int
       colInit, int colEnd, bool copyBack_Matrix, bool copyBack_ex_p);
25
26 template <typename F_TYPE, typename T, class U>
27 __global__ void dp_by_colums_stage(F_TYPE dev_f, Matrix<T>* M_device, U*
       ex_p_device, int colum);
```

## 4.4. Problema de la mochila (0-1) como caso de estudio detallado

Para esclarecer las ideas y ejemplificar el uso del marco de trabajo, se procederá a detallar la implementación para el problema de la mochila (0-1). Como primer paso se presenta el código secuencial, que se basa en el seudocódigo de la sección 2.2:

```
1  void knapsack(unsigned int i, unsigned int j, Matrix<unsigned int>* M,
       unsigned int * ex_p){
2      if(i == 0 || j == 0){
3        M->setAt(i, j, 0);
4      }else{
5        unsigned int wi = ex_p[i-1];
6        unsigned int p1 = M->getAt(i-1, j);
7        unsigned int p2 = (wi > j)? p1 : M->getAt(i-1, j-wi) +
             ex_p[(M->n-1)+(i-1)];
8        M->setAt(i, j,max(p1, p2));
9    }
10 }
11 void fillKnapsack(Matrix<unsigned int>* M, WeightProfits* ex_p){
12     for(unsigned int i = 0; i < M->n; i++)
13       for(unsigned int j = 0; j < M->m; j++)
14         knapsack(i, j, M, ex_p);
15 }
16 int main(){
17     unsigned int ex_p[2*(n-1)];
18     //Llenar los parametros extras con los weigth y profits
19     Matrix<unsigned int>* M_host = new Matrix<unsigned int>(n, W, ROWS);
20     M_host->M = new unsigned int[n*W];
21     fillKnapsack(M_host, ex_p);
22   //do something with matrix M
23     delete M_host;
24     return EXIT_SUCCESS;
25 }
```

Para ejecutar el problema de la mochila utilizando el marco de trabajo, primero es necesario identificar si cumple con algún patrón definido, es decir si pertenece a su clase correspondiente. Debido a la estructura de sus dependencias, el problema de la mochila cumple con el patrón RUD, lo que se demostrará en seguida.

**Lemma 4.2.** *La ecuación funcional $f_{M_{n\times m}}$ para el problema de la mochila (0-1) cumple con el patrón RUD.*

*Demostración.* Se tiene que ver que para cualquier índice $\bar{x} \in I_{n\times m}$,

$$\Gamma_f^+(\bar{x}) \subseteq UpLeft(\bar{x}) \cup Up(\bar{x}) \cup UpRight(\bar{x})$$

Supongamos que $I_{n\times m} \neq \emptyset$ es decir que la matriz no es nula y vamos a demostrarlo por inducción sobre el índice de los renglones.
**Caso base:** $i = 0$
Sea $(0, y) \in I_{n\times m}$ un índice cualquiera perteneciente al renglón 0. Entonces se tiene que:

$$\Gamma_f^+((0,y)) = \emptyset = UpLeft((0,y)) \cup Up((0,y)) \cup UpRight((0,y))$$

, por lo tanto se cumple la propiedad.
**Hipótesis de inducción:** Para cualquier índice en el renglón $k$-ésimo se cumple la

propiedad.
**Paso inductivo:** Se demostrará la propiedad para cualquier índice en el renglón $(k+1)$.
Sea $(k+1, y) \in I_{n\times m}$ un índice en el renglón $k+1$.
**Caso 1)**
Si $y = 0$ entonces:

$$\Gamma_f^+((k+1,0)) = \emptyset \subseteq UpLeft((k+1,0)) \cup Up((k+1,0)) \cup UpRight((k+1,0))$$

, por lo tanto se cumple la propiedad.
**Caso 2)**
Si $y \neq 0$ y $w_{k+1} \leq y$ entonces $\Gamma_f((k+1,y)) = \{(k,y),(k,y-w_{k+1})\}$, pero estos dos índices estan en el renglón $k$ por lo que cumplen la hipótesis de inducción, por lo tanto:

$$\Gamma_f^+((k,y)) \subseteq UpLeft((k,y)) \cup Up((k,y)) \cup UpRight((k,y))$$

$$\subseteq UpLeft((k+1,y)) \cup Up((k+1,y)) \cup UpRight((k+1,y))$$

, lo mismo sucede para el otro índice.

También es fácil observar que:

$$\Gamma_f((k+1,y)) \subseteq UpLeft((k+1,y)) \cup Up((k+1,y)) \cup UpRight((k+1,y))$$

, por lo tanto:

$$\Gamma_f^+((k+1,y)) \subseteq UpLeft((k+1,y)) \cup Up((k+1,y)) \cup UpRight((k+1,y))$$

y se cumple la propiedad.
**Caso 3)**
Si $y \neq 0$ y $y < w_{k+1}$ la demostración es análoga al caso anterior.
Por lo tanto la ecuación funcional para el problema de la mochila (0-1) cumple con el patrón RUD. □

Una vez que se sabe que el problema de la mochila (0-1) cumple el patrón RUD, sólo es necesario invocar la función pertinente, pasándole como parámetro la ecuación funcional. El siguiente código ilusta la manera de realizarlo:

```
1  __device__ void knapsack(unsigned int i, unsigned int j, Matrix<unsigned
       int>* M, unsigned int* ex_p){
2    if(i == 0 || j == 0){
3      M->setAt(i, j, 0);
4    }else{
5      unsigned int wi = ex_p[i-1];
6      unsigned int p1 = M->getAt(i-1, j);
7      unsigned int p2 = (wi > j)? p1 : M->getAt(i-1, j-wi) +
           ex_p[(M->n-1)+(i-1)];
8      M->setAt(i,j, max(p1, p2) );
9    }
10 }
11
12 __device__ void (*fillKnapsack)(unsigned int, unsigned int,
       Matrix<unsigned int>*, unsigned int*) = knapsack;
13 //Funcion que recupera la solucion
14
15 __global__ void objects(Matrix<unsigned int>* M, unsigned int* ex_p){...}
16
17 int main(){
18   unsigned int ex_p[2*n-1)];
19   //Llenar los parametros extras con los weigth y profits
20   DimMatrix<unsigned int>* dim = new DimMatrix<unsigned int>(n, W);
21   dp_by_rows(dim, &fillKnapsack, &objects, ex_p, (2*(n-1))*sizeof(unsigned
         int), false, false);
22   delete dim;
23   return EXIT_SUCCESS;
24 }
```

La línea 11 es un variable estática, que tiene como objetivo dejar un rastro en el equipo anfitrión de la dirección de memoria de la función que calcula las entradas de la matriz asociada. Cada que se quiera utilizar un patrón debe de declararse una de estas variables, ya que CUDA no permite utilizar directamente el apuntador de función declarada en el dispositivo dentro de código ejecutado en CPU. Como se puede observar se está absorbiendo casi la totalidad de la parte administrativa de CUDA, lo cual se quiere hacer énfasis mostrando la misma implementación pero sin utilizar el marco de trabajo:

```
__device__ void knapsack(unsigned int i, unsigned int j, unsigned int n,
    unsigned int m, unsigned int* M, unsigned int* ex_p){
  if(i == 0 || j == 0){
    M[i*m + j]= 0;
  }else{
    unsigned int wi = ex_p[i-1];
    unsigned int p1 = M[(i-1)*m + j];
    unsigned int p2 = (wi > j)? p1 : M[(i-1)*m + (j-wi)] +
        ex_p[(n-1)+(i-1)];
    M[i*m+j]=max(p1, p2);
  }
}

__global__ void rows_stage(unsigned int n, unsigned int m, unsigned int*
    M, unsigned int * ex_p, int row){
  int id = threadIdx.x + blockIdx.x*blockDim.x;

  for(;id < m; id+= blockDim.x * gridDim.x)
    knapsack(row, id, n, m, M, ex_p);
}

int main(){
  unsigned int ex_p[2*n-1)];
  //Llenar los parametros extras con los weigth y profits
  unsigned int* ex_p_dev;
  cudaAssert( cudaMalloc((void**)&ex_p_dev, 2*(n-1)*sizeof(unsigned int)) );
  cudaAssert( cudaMemcpy((void*)ex_p_dev, (void*)ex_p,
      2*(n-1)*sizeof(unsigned int), cudaMemcpyHostToDevice) );

  //Creando un arreglo que representa la matriz en el Device
  unsigned int* M_dev;
  cudaAssert( cudaMalloc((void**)&M_dev, (n*m)*sizeof(unsigned int)) );
  int nb = min( (int) ceil(m/((float)min(m,1024))), 65535 );
  for(int row=0; row < n; row++)
    rows_stage<<<nb, min(m,1024)>>>(n, m, M_dev, ex_p_dev, row);

  //Creando un arreglo que representa la matriz en el Host
  unsigned int* M_host = new unsigned int[n*m];
  cudaAssert( cudaMemcpy((void*)M_host, (void*)M_dev, (n*m)*sizeof(unsigned
      int), cudaMemcpyDeviceToHost) );

  //Hacer algo con la matriz
  delete M_host;
  cudaAssert( cudaFree(ex_p_dev) );
  cudaAssert( cudaFree(M_dev) );
}
```

# Capítulo 5

# Resultados

Para evaluar el desempeño que se obtiene al utilizar el marco de trabajo, se crearon tres implementaciones para cada caso de estudio, las cuales corresponden a la secuencial, a la que utiliza el código realizado en este trabajo y una implementación específicamente elaborada para el problema que se ejecuta en GPU. Los resultados se presentan como gráficas, donde se muestra el tiempo de ejecución en función del tamaño de la matriz; y debido a que es posible llevar a cabo la recuperación de la solución tanto en GPU como en CPU, se incluyen ambos casos para analizar si es conveniente o no la transferencia de memoria entre el dispositivo y el equipo anfitrión. Para la realización de las pruebas, el Dr. Mario A. López catedrático e investigador de la universidad de Denver, proporcionó un equipo de cómputo con las siguientes características:

- Procesador Intel Xeon E5530 a 2.4Ghz, 4 núcleos, 8 hilos.
- 32Gb de memoria RAM.
- GPU Tesla K20m:
  - Compute Capability 3.5.
  - 5120Mb RAM a 2600Mhz.
  - 320 bits en bus de memoria.
  - 2496 CUDA cores.

## 5.1. Pruebas de rendimiento

Las pruebas de rendimiento consistieron en medir el tiempo de ejecución en milisegundos, como función del tamaño de la matriz para cada caso de estudio descrito en el capítulo 2. Asociado a esto se incluye una gráfica que muestra la velocidad alcanzada en aquellas instancias que se ejecutan utilizando el GPU, esto con la finalidad de conocer que tanto se aceleró nuestra aplicación utilizando el marco de trabajo y que tanto se aleja de una implementación hecha a la medida. Para facilitar las cosas se utilizaron instancias cuya matriz asociada es cuadrada, por lo que el muestreo en el eje $x$ no es constante, sino

que es ajustado al valor más próximo, provocando pequeñas fluctuaciones en la gráfica. Para cada muestra se realizaron quince ejecuciones obteniendo un promedio de ellas, lo que brinda una mayor certeza en los resultados obtenidos.

Como es de esperar, para los tres primeros casos de estudio, el tiempo de procesamiento crece de manera lineal mientras incrementamos el tamaño de la matriz asociada. Para el problema de la multiplicación en cadena de matrices, el tiempo crece en forma más pronunciada, sin embargo esto es completamente lógico debido a que el algoritmo tiene una complejidad cúbica. Los resultados también muestran que recuperar la solución en el dispositivo cuando la matriz asociada tiene un tamaño considerable, provoca un mejor desempeño que aquellas implementaciones que llevan a cabo la copia de la matriz y posteriormente recuperan la solución en el equipo anfitrión.
Finalmente se observó que las implementaciones creadas especificamente para el problema superan en gran manera la velocidad alcanzada por el marco de trabajo, aunque esto no es de sorprender, ya que este contiene cierta sobrecarga de operaciones y trabajo administrativo que puede ser omitido en una implementación a la medida.

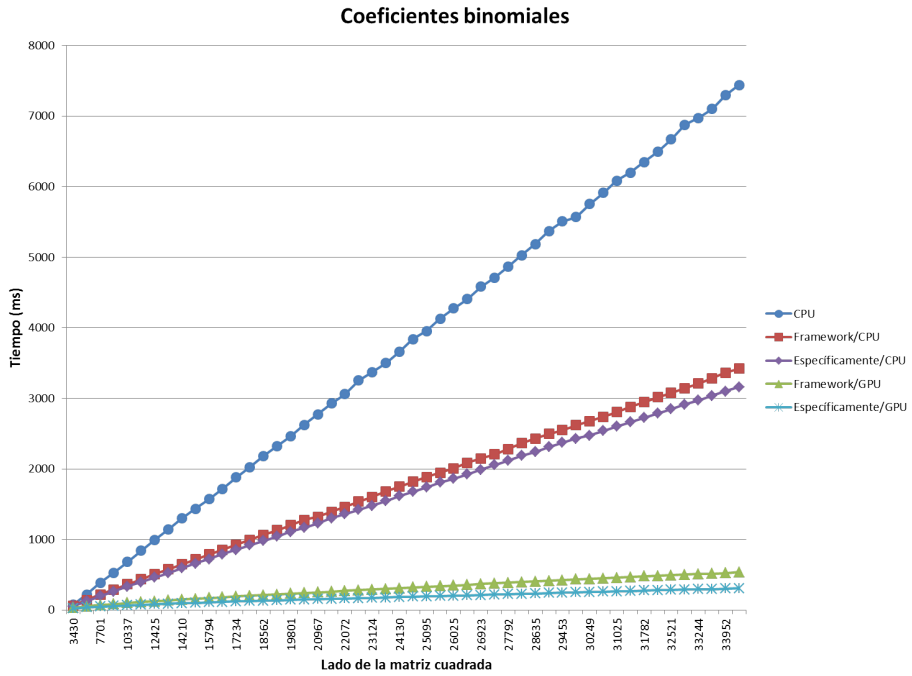


Figura 5.1: Tiempo de ejecución vs. tamaño de la matriz en el problema de los coeficientes binomiales.

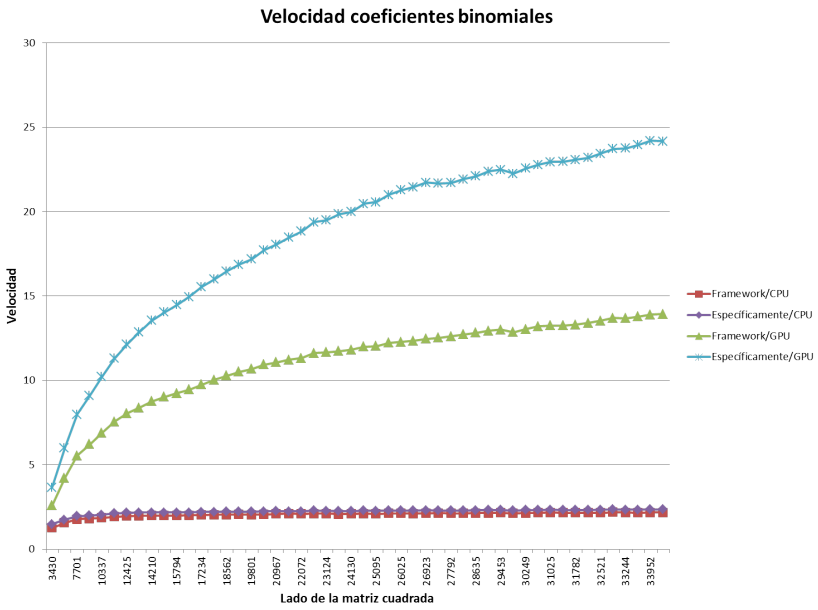


Figura 5.2: Velocidad alcanzada en el problema de los coeficientes binomiales.

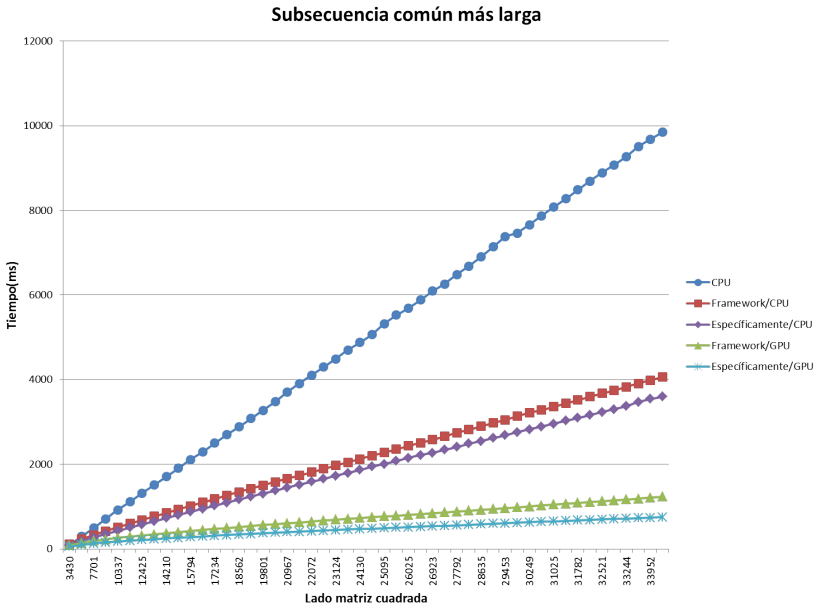


Figura 5.3: Tiempo de ejecución vs. tamaño de la matriz en el problema de la subsecuencia común más larga.

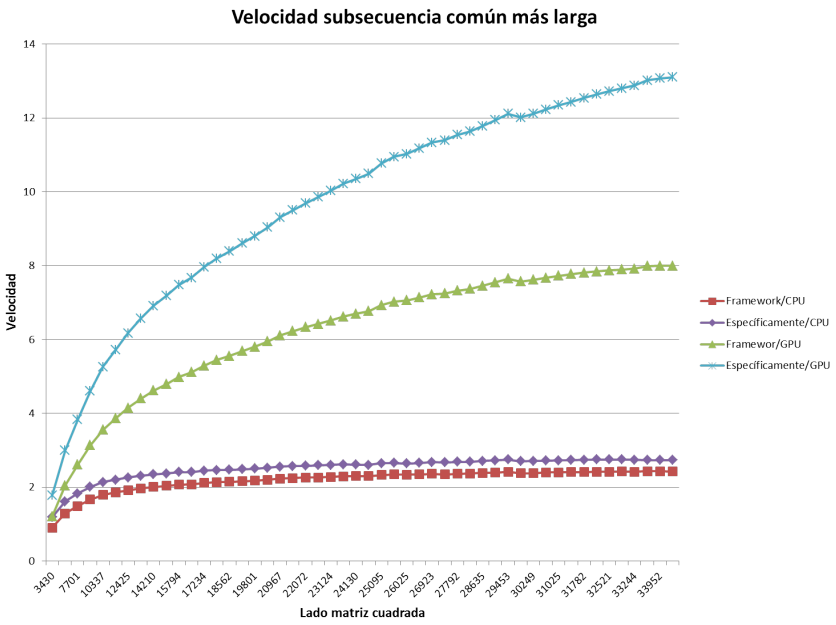


Figura 5.4: Velocidad alcanzada en el problema de la subsecuencia común más larga.

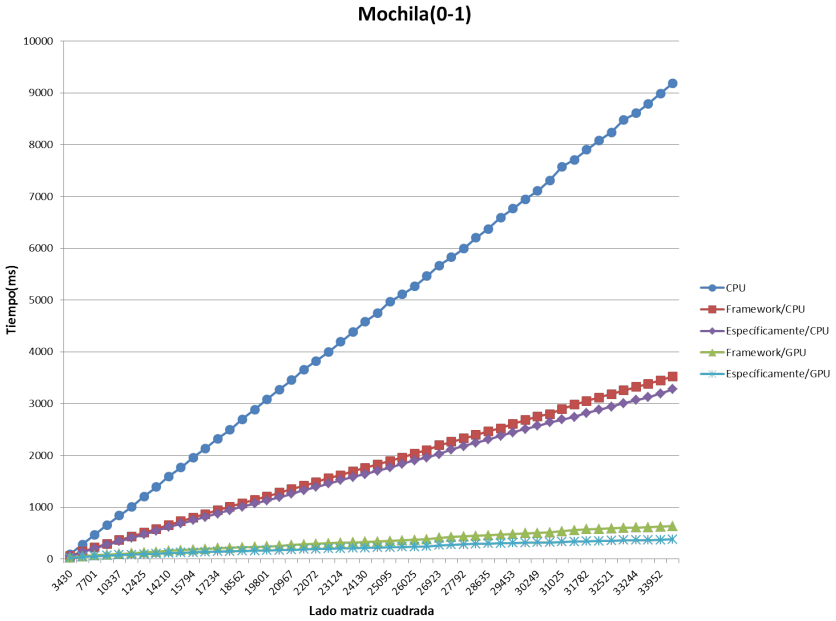


Figura 5.5: Tiempo de ejecución vs. tamaño de la matriz en el problema de la mochila (0-1).

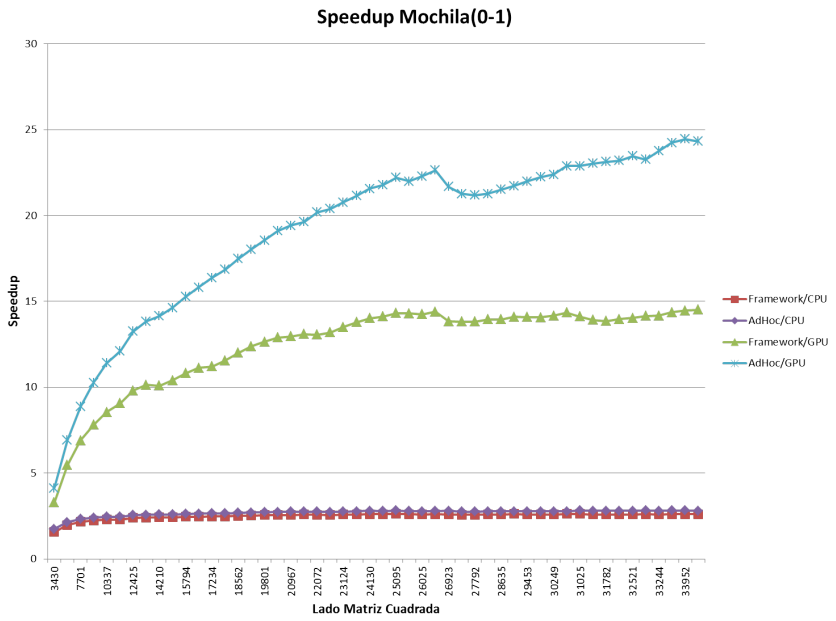


Figura 5.6: Velocidad alcanzada en el problema de la mochila (0-1).

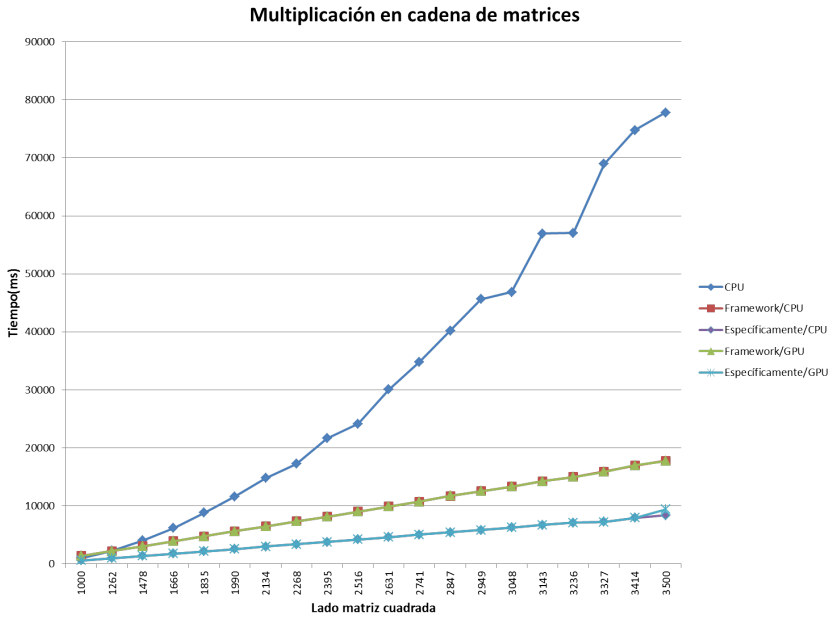


Figura 5.7: Tiempo de ejecución vs. tamaño de la matriz en el problema de la multiplicación en cadena de matrices.

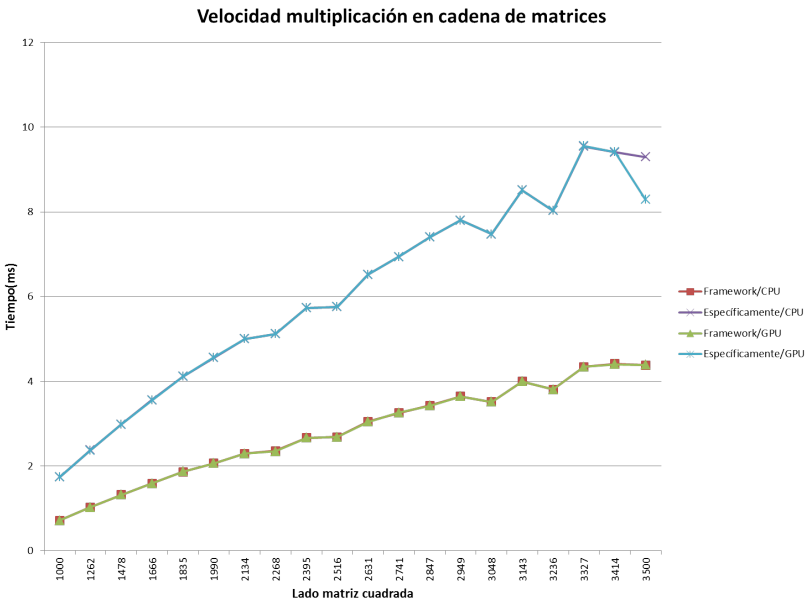


Figura 5.8: Velocidad alcanzada en el problema de la multiplicación en cadena de matrices.

# Capítulo 6

# Conclusiones

El objetivo principal de este trabajo ha sido exponer dos ideas con las cuales se puede realizar programación dinámica paralela utilizando las GPU. Esto se llevó a cabo con la finalidad de ofrecer a los programadores un mecanismo que sea de gran utilidad, al facilitar la obtención de cierto paralelismo en una ecuación funcional, sin la necesidad de tener un conocimiento profundo de la arquitectura y modelo de programación en tarjetas gráficas CUDA. Se espera que con unas pequeñas modificaciones en el código, este se encuentre listo para aprovechar los enormes beneficios que actualmente ofrecen las GPU.

El primer enfoque se basa en la creación de una gráfica de dependencias, en la cual cada vértice representa un subproblema a resolver, identificado por un índice de la matriz asociada y las aristas indican las dependencias que existen entre ellos. Al generar la secuencia mínima de conjuntos de vértices independientes entre sí, es decir que no existen dependencias entre ellos, se puede proceder a calcular los resultados de los subproblemas de manera paralela. Sin embargo se mostró que este enfoque es muy poco viable, debido a la gran sobrecarga administrativa que se genera al construir explícitamente la gráfica de dependencias.

La segunda propuesta consistió en definir ocho patrones, los cuales aprovechan cierta estructura en las dependencias de la ecuación funcional, para iterar en una cierta forma y dirección sobre la matriz asociada, cumpliendo con ello las relaciones de dependencia. En este punto se creó un marco de trabajo para CUDA que encapsula la esencia de estos patrones en funciones, a las cuales se les pasa como parámetros, la matriz asociada, la función de llenado, información adicional, entre otras cosas y se tiene como resultado la ejecución en paralelo utilizando la GPU. Una vez que se tuvo la implementación del marco de trabajo, se midió el tiempo de procesamiento y la velocidad alcanzada en ciertos casos de estudios predefinidos, de tal manera que se pudiera observar el desempeño que se logró alcanzar y que tan lejos se encuentra de una implementación diseñada específicamente para el problema. A través de estas pruebas se pudo observar que el marco de trabajo logró reducir el tiempo en la obtención de resultados en los cuatro casos de estudio y con mínimos ajustes en el código.

Finalmente al concluir toda esta investigación y realizar las pruebas pertinentes se ha llegado a las siguientes conclusiones:

- El modelo de programación en CUDA actualmente ofrece un entorno accesible a los programadores, para que puedan desarrollar aplicaciones de propósito general en las GPU.
- Los problemas resueltos con ayuda de la programación dinámica son potenciales candidatos para ser paralelizables y sus implementaciones en CUDA no implican cambios radicales en el código.
- No importa el número de CUDA cores que se tenga, el rendimiento que se obtiene al paralelizar un algoritmo se encuentra limitado por la parte secuencial.
- Si no se cuidan aspectos importantes como los límites físicos de la arquitectura, la ocupación, accesos optimizados de caché, transferencia de memoria, las implementaciones que se migren a CUDA podrían tener un desempeño desfavorable en consideración con su contraparte en CPU.
- Para aquellas implementaciones con una inmensa cantidad de accesos a memoria y pocos cálculos aritméticos, se recomienda una mayor ocupación en los SM, con la finalidad de ocultar la latencia en las peticiones de memoria.
- Generar la gráfica de dependencias explícita para obtener paralelismo en programación dinámica no es recomendable, debido a la cantidad de memoria adicional necesaria para mantenerla y el tiempo adicional de procesamiento.
- La secuencia de etapas que se generan al utilizar la gráfica de dependencias, sólo proporciona una forma en la que un problema puede ser ejecutado en paralelo. Sin embargo pueden existir diferentes alternativas que brinden un mejor nivel de paralelismo.
- El marco de trabajo propuesto en este trabajo ofrece una opción bastante aceptable, para ejecutar problemas de programación dinámica utilizando las GPU con arquitectura CUDA. Obteniendo un rendimiento favorable y sin muchos reajustes en el código.
- Una considerable cantidad de problemas resueltos con programación dinámica, se ajustan en alguno de los ocho patrones que se han definido en esta tesis.
- El intercambio de grandes volúmenes de información entre el dispositivo y el equipo anfitrión cuando se tiene poca intensidad aritmética, ocasiona que se vea considerablemente mermado el desempeño que se puede lograr al utilizar las GPU. Es por este motivo que se recomienda no llevar a cabo esta labor siempre que sea posible.

- El uso del marco de trabajo genera cierta sobrecarga administrativa, que en una implementación específicamente para el problema puede suprimirse, consiguiendo un mejor desempeño.
- El ancho de banda de la memoria en las GPU puede ser un factor limitante en implementaciones con alta demanda de accesos a memoria, ya que puede actuar como un cuello de botella.

Con las pruebas realizadas se pudo observar que las implementaciones especialmente diseñadas de los casos de estudio, tienen un factor de velocidad de casi el doble en comparación de aquellas que utilizan el marco de trabajo. Por esta razón es necesario seguir buscando diferentes alternativas de optimización con la finalidad de lograr mejores resultados.

Una de las mejoras que se pueden hacer al marco de trabajo es añadir patrones, de tal manera que se abarque un amplio panorama de problemas de programación dinámica. Otra posible mejora que puede realizarse es utilizar clases de C++ en vez de funciones, así se puede administrar de mejor manera cada patrón en sus aspectos específicos, para reducir un poco la carga administrativa.
Finalmente el código del marco de trabajo puede reescribirse para que utilice el estándar OpenCL. De esta manera puede desarrollarse código portable, es decir capaz de ejecutarse indistintamente en diferentes arquitecturas de GPU.

# Bibliografía


[1] Nvidia G80 Architecture and CUDA Programming. School of Electrical Engineering and Computer Science, University of Central Florida.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] R. Bellman and E. Lee. History and development of dynamic programming. *Control Systems Magazine, IEEE*, 4(4):24–28, 1984.

[4] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, USA, 1957.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[6] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Education, 2006.

[7] Juan F. Díaz. Programación dinámica [online]. Agosto 2002. URL: `http://ocw.univalle.edu.co/ocw/ingenieria-de-sistemas-telematica-y-afines/fundamentos-de-analisis-y-diseno-de-algoritmos/material/sesion8jfd.pdf` [cited 17/Marzo/2013].

[8] Qianqian Fang and David A. Boas. Monte carlo simulation of photon migration in 3d turbid media accelerated by graphics processing units. *Opt. Express*, 17(22):20178–20190, Oct 2009.

[9] Carlos A. Felippa. *Introduction to Finite Element Method*. Department of Aerospace Engineering Sciences and Center for Aerospace Structures, University of Colorado Boulder, Colorado, USA, 2004.

[10] R.G. García and A.V. Moreno. *Técnicas de diseño de algoritmos*. Manuales Universidad Series. Universidad de Málaga, Servicio de Publicaciones e Intercambio Científico, 1997.

[11] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[12] Ronald I. Greenberg. Bounds on the number of longest common subsequences. *CoRR*, cs.DM/0301030, 2003.

[13] Costas S. Iliopoulos and M. Sohel Rahman. Algorithms for computing variants of the longest common subsequence problem. In *In ISAAC*, pages 399–408. Springer, 2006.

[14] Ilse C. F. Ipsen. *Numerical Matrix Analysis: Linear Systems and Least Squares*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[15] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[16] Anand Lal Shimpi and Derek Wilson. Nvidia's 1.4 billion transistor gpu: Gt200 arrives as the geforce gtx 280 & 260 [online]. 2008. URL: `http://www.anandtech.com/show/2549/2` [cited 29/Agosto/2013].

[17] A. Levitin. *Introduction to the Design and Analysis of Algorithms*. Always learning. Addison Wesley, 2011.

[18] A. Lew and H. Mauch. *Dynamic Programming: A Computational Tool*. Studies in Computational Intelligence. Springer, 2010.

[19] Nicolás Guil Mata. Arquitectura y programación de procesadores gráficos. Dpto. de Arquitectura de Computadores, Universidad de Málaga, España.

[20] NVIDIA. Geforce 256: The world's first gpu [online]. URL: `http://www.nvidia.com/page/geforce256.html` [cited 18/Mayo/2013].

[21] NVIDIA. NVIDIA GeForce GTX 680. The fastest, most efficient GPU ever built. [online]. URL: `http://international.download.nvidia.com/webassets/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf` [cited 29/Agosto/2013].

[22] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi [online]. URL: `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf` [cited 29/Agosto/2013].

[23] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110 [online]. URL: `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf` [cited 29/Agosto/2013].

[24] NVIDIA. ¿qué es el gpu computing? [online]. URL: `http://www.nvidia.es/object/gpu-computing-es.html` [cited 18/Mayo/2013].

[25] Will Ramey. Languajes, apis and development tools [online]. 2010. URL: http://www.nvidia.com/content/GTC-2010/pdfs/2004_GTC2010.pdf [cited 18/Mayo/2013].

[26] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[27] Anton Shilov. Nvidia: Next-Generation Maxwell Architecture Will Break New Grounds. [online]. Abril 2013. URL: http://www.xbitlabs.com/news/graphics/display/20130412175120_Nvidia_Next_Generation_Maxwell_Architecture_Will_Break_New_Grounds.html [cited 29/Agosto/2013].

[28] Graham Singer. The history of the modern graphics processor [online]. Marzo 2013. URL: http://www.techspot.com/article/650-history-of-the-gpu/ [cited 18/Mayo/2013].

[29] Pedro D. Sánchez Salazar. Apuntes de combinatoria para la olimpiada de matematicas [online]. Marzo 2002. URL: http://www.cimat.mx/~amor/Omi/Entrenamiento/Combinatoria/Notas_de_Combinatoria.pdf [cited 17/Mayo/2013].

[30] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[31] Sv Community. Gpu: ya no solo a los gamers les interesa [online]. 2009. URL: http://www.svcommunity.org/forum/articulos/gpu-ya-no-solo-a-los-gamers-les-interesa-(parte-1-conociendo-la-historia)/1 [cited 17/Mayo/2013].

[32] V. Volkov. Better performance at lower occupancy. In *GPU Technology Conference*, 2010. URL: http://www.cs.berkeley.edu/~{}volkov/volkov10-GTC.pdf.

[33] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *Micro, IEEE*, 31(2):50–59, 2011.