UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA
EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Segmentación de imágenes en CUDA

Julio Barriuso Medrano

Septiembre, 2011

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS

PROYECTO FIN DE CARRERA

Segmentación de imágenes en CUDA

Autor: Julio Barriuso Medrano

Director: Dr. D. José Luis Sánchez García

Septiembre, 2011

Este documento ha sido elaborado en LaTeX, y producido en formato PDF 1.5 por MiKTeX 2.9 y pdfTex-1.40.12

Gran parte de los esquemas del capítulo *implementación en CUDA* son de elaboración propia, usando el paquete TikZ.

TRIBUNAL:

Presidente: _____

Vocal: _____

Secretario: _____

FECHA DE DEFENSA: _____

CALIFICACIÓN: _____

PRESIDENTE	VOCAL	SECRETARIO
Fdo.:	Fdo.:	Fdo.:

Resumen

Para seguir aumentando el rendimiento de las aplicaciones informáticas, ya no basta con procesadores de un solo núcleo. El paralelismo de datos es vital para aprovechar las nuevas soluciones *multi-core*. Dentro de este contexto, la arquitectura CUDA, junto con su API, permiten desarrollar programas de altas prestaciones para GPUs nVIDIA. El campo del tratamiento digital de imágenes es, sin duda, uno de los que más pueden beneficiarse de todo ese potencial. En concreto, la segmentación de imágenes, encargada de separar objetos. En este trabajo se presenta un diseño paralelo del algoritmo de segmentación de imágenes por crecimiento de regiones, implementado en CUDA. Encontrar en éste dicho paralelismo, sin embargo, no será tarea fácil.

Agradecimientos

Quiero expresar mi agradecimiento a mi director de proyecto José Luis Sánchez García, por sus correcciones detalladas, su paciencia, y su buena disposición.

Índice general

Índice de figuras

(En azul, los esquemas de elaboración propia, usando el paquete Tikz)

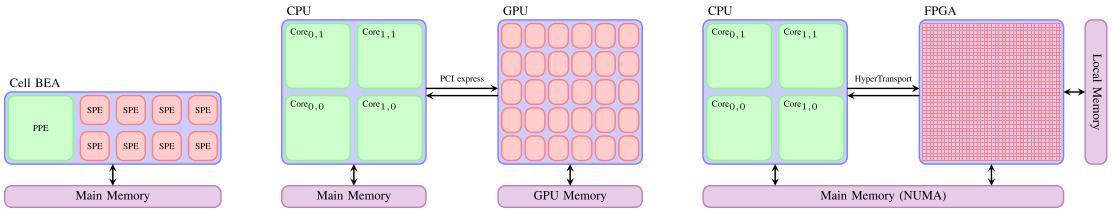
Índice de tablas

Índice de listados

Capítulo 1

Introducción y motivación

En los últimos años, la computación en paralelo ha experimentado un auge notable. Existe un gran interés en mejorar la eficiencia de aplicaciones que, tradicionalmente, han mostrado un rendimiento pobre en su forma secuencial. El aumento de la frecuencia en las CPUs, uno de los principales motores de su avance, ya no es suficiente. Problemas en la disipación de calor y en la escala de integración lo impiden. Por ello, los principales fabricantes han venido introduciendo modelos de varios núcleos desde 2005. Además, junto con los procesadores centrales, se han incluido otros secundarios, conocidos como «aceleradores», dando lugar a la *computación heterogénea*. Los aceleradores tienen un juego de instrucciones distinto al del procesador principal[1], y son de muy diversa índole: coprocesadores (Cray, Cell), procesadores específicos (de señales: DSPs; gráficos: GPUs), o lógica programable (ASIC, FPGA). En la figura 1.1, de [4], podemos ver varios ejemplos. En algunos, los espacios de memoria están juntos, y en otros separados. La interconexión física también es variable[2].



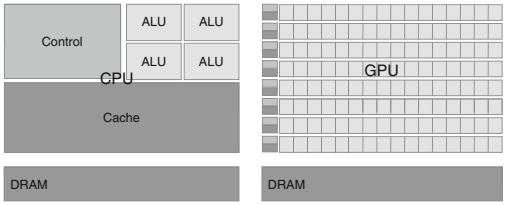
(a) CBEA (b) CPU in combination with GPU (c) CPU in combination with FPGA

Figura 1.1: Tres ejemplos de arquitecturas heterogéneas: Un chip heterogéneo: Cell Broadband Engine (a); dos sistemas heterogéneos: CPU en combinación con GPU (b), y CPU en combinación con FPGA(c).

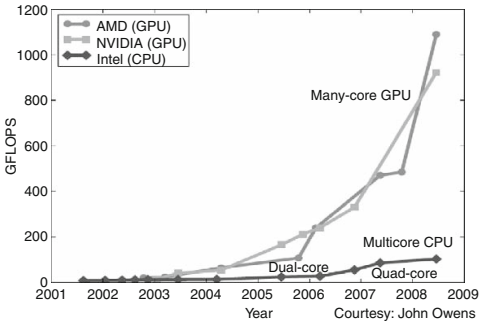
La computación heterogénea con procesadores gráficos requiere que estos dispositivos estén preparados para el desarrollo de aplicaciones de propósito general. Para ello, NVIDIA ha diseñado una nueva arquitectura en sus GPUs, llamada CUDA. ATi, por su parte,

[1] Por tanto, un sistema multi-procesador en el que todas las unidades computacionales son iguales no sería considerado un sistema heterogéneo.

[2] En el caso de GPU, es PCI Express, los FPGA suelen tener sockets compatibles con Intel/AMD, y los coprocesadores están encapsulados en el mismo chip junto con el procesador principal. Este último empaquetamiento es el ideal, ya que las transferencias son mucho más rápidas, y también se está proponiendo para CPU-GPU (AMD Fusion).



(a) La filosofía de diseño de CPUs y GPUs es muy diferente.



(b) La diferencia de rendimiento entre CPUs y GPUs es cada vez más grande.

ofrece una implementación de características similares, Stream[3].

La filosofía de CPU es radicalmente distinta a la de GPU. En CPU, gran parte del chip se dedica a lógica de control, para conseguir ejecución fuera de orden y lograr un diseño superescalar. Además, la memoria caché es amplia, reduciendo así la latencia de instrucciones y datos. Estas técnicas tienen como objetivo la mejora del rendimiento en programas secuenciales, pero no contribuyen por sí mismas a la velocidad de cómputo (GFLOPS). Sin embargo, en GPU la lógica de control y caché son mínimas, y el grueso del chip está ocupado por un gran número de núcleos, que, a pesar de ser individualmente más sencillos que sus homólogos de CPU (figura 1.2a, de [15]), ofrecen en su conjunto más capacidad de procesamiento (figura 1.2b, de [15]). Eso sí, para acceder a todo ese potencial, es necesario ocupar todos los recursos de ejecución. Eso sólo se conseguirá con un programa paralelo, o, en otras palabras, que presente una fuerte independencia de datos.

Hoy en día existen numerosas aplicaciones computacionalmente exigentes que pueden beneficiarse de esa capacidad. En [21] se mencionan algunos sistemas que usan CUDA en la vida real, como la obtención de imágenes 3D a partir de ultrasonidos, y la simulación molecular y de fluidos. Es fácil imaginar otras muchas en un futuro cercano. Por ejemplo, la televisión en alta definición. Gran parte del precio de estos aparatos no está en la pantalla, sino en la electrónica. Todos necesitan aplicar reescalado cuando la resolución de entrada no se corresponde con la nativa. Además, muchos de los modelos actuales incluyen algún sistema de interpolación de imágenes[4]. Otro caso podrían ser los sistemas de videovigilancia, que ya no se limitan a grabar, sino que identifican en tiempo real movimientos de personas, o cuentan su número. En numerosas fábricas el control de calidad se realiza mediante un ordenador conectado a una cámara: pequeñas fracturas, el nivel de agua de un envase, etc.

[3]Tanto CUDA como Stream vienen de la mano de APIs propietarias, aunque existen otras que permiten la programación en ambas, como OpenCL y DirectCompute. En este trabajo se utilizan GPUs nVIDIA y el API de CUDA.

[4]Como por ejemplo *Sony Motionflow*. Estos sistemas evitan el emborronamiento que percibe el ojo humano en las pantallas LCD ante imágenes rápidas en movimiento.

Muchos de los ejemplos anteriores están relacionados con el tratamiento de imágenes. Este trabajo también. En concreto, con una fase muy importante: la segmentación. Su objetivo es localizar objetos en una imagen, separándolos de los demás. Esto es una tarea sencilla para un humano, pero en un programa requiere relacionar un píxel con sus vecinos. Es una operación costosa, que puede ser significativamente acelerada mediante un diseño paralelo con CUDA.

Esta memoria está organizada en dos capítulos introductorios, uno a la segmentación de imágenes y otro a CUDA, para seguir después con la descripción de la implementación propiamente dicha. Posteriormente se verán algunas medidas de rendimiento y conclusiones.

Capítulo 2

Objetivos

El objetivo principal de este trabajo es el diseño de un algoritmo paralelo de segmentación por crecimiento de regiones, usando la plataforma CUDA. Esto no es algo inmediato, pues se hace necesaria la toma de una serie de decisiones para adaptar la versión secuencial. Esas decisiones implican dificultades como la división del trabajo, la comunicación y sincronización de hilos o, como veremos, el sacrificio de algo de exactitud en aras de un mayor rendimiento. Todas las arquitecturas paralelas tienen sus peculiaridades. CUDA no es una excepción. Las GPUs tienen mayor ancho de banda a memoria que los procesadores centrales, pero su aprovechamiento depende de patrones de acceso adecuados. Disponemos de cachés muy rápidas accesibles al programador, pero sólo un uso eficiente mejorará la velocidad de cómputo. A lo largo de este documento se verán algunas de estas cuestiones, explorando las ventajas e inconvenientes de CUDA.

Éste es un trabajo esencialmente práctico. No se pretenden introducir técnicas de segmentación excesivamente complejas. Al lector con conocimientos en tratamiento digital de imágenes, la implementación puede parecerle incluso «simplona». Realmente lo es. Este proyecto se centra más bien en el proceso de adaptación de un algoritmo secuencial a uno paralelo, para una plataforma heterogénea basada en GPU.

Capítulo 3

Segmentación de Imágenes

3.1. Contexto en el tratamiento digital de imágenes

Esta sección nos ayudará a entender qué es la segmentación, y dónde se sitúa dentro del procesamiento de imágenes. El tratamiento digital de imágenes es complejo, y consta de varias etapas (3.1, obtenida de [12]). Las ilustraremos con el ejemplo del OCR, el reconocimiento óptico de caracteres (ver figura 3.2, de [19]). El dominio del problema serían las hojas impresas, y el resultado buscado es un flujo digital de caracteres.

La primera etapa consiste en la adquisición de imágenes. Es necesario convertirla a digital en caso de que la cámara usada sea analógica. Después, entra en juego el preprocesamiento. Su función es «mejorar» la imagen, modificarla para que las fases siguientes encuentren menos dificultades. En nuestro caso (3.2b), consiste en aumento del contraste, eliminación de ruido, y delimitación de zonas cuya textura pueda apuntar a la presencia de caracteres impresos.

El siguiente paso es la segmentación. Básicamente, consiste en dividir una imagen en sus partes constituyentes u objetos. Para el ejemplo de OCR, la idea es separar las letras del fondo (3.2c). Conseguir que este proceso se realice autónomamente, sin intervención humana, es complicado. Se hace necesario buscar el equilibrio entre métodos demasiado costosos en tiempo, y otros demasiado sencillos, que pueden causar inexactitudes que se agravarán en fases posteriores. En la figura 3.2 se muestran otros detalles de la segmentación como detectar componentes conexas 3.2e.

El resultado de la segmentación son datos de píxeles en bruto, normalmente un array de números con las regiones a las que pertenece cada punto. Es necesario convertir los datos a un formato adecuado para poder ser convenientemente procesados. Tenemos que decidir entre una representeción por contornos o por regiones. La representación en contornos es útil para estudiar la forma exterior, como ángulos y salientes. La representación con regiones, por el contrario, describe características internas (textura, estructuración...). En el caso de OCR ambas interesan.

Posteriormente se aplica la descripción, también conocida como selección de rasgos. Esta etapa obtiene información que ayude a distinguir un objeto de otro. Para reconocimiento de caracteres, serían aspectos relativos a la forma o silueta de los glifos: huecos interiores, remaches, etc.

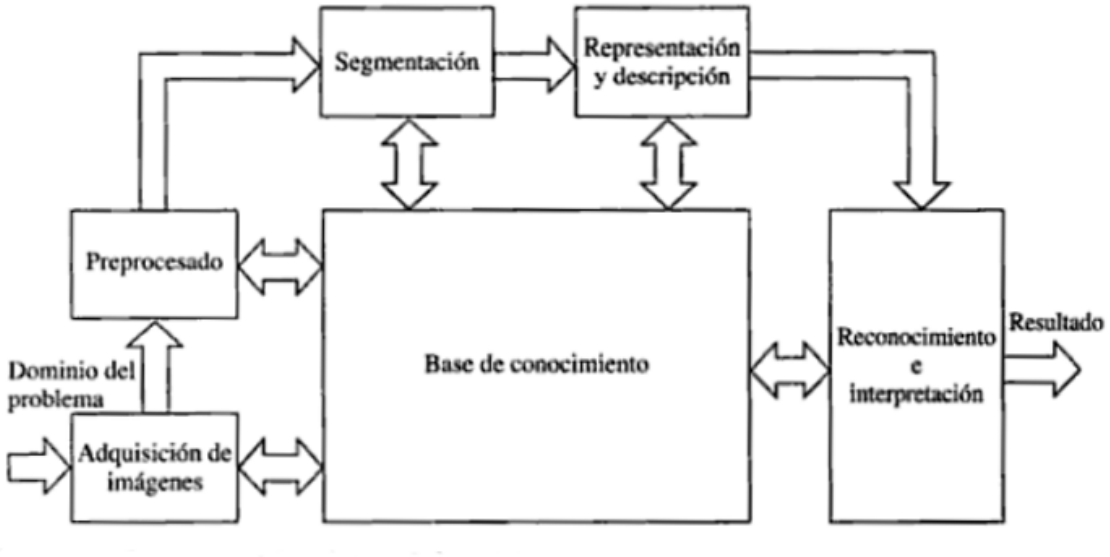


Figura 3.1: Etapas fundamentales del procesamiento digital de imágenes.

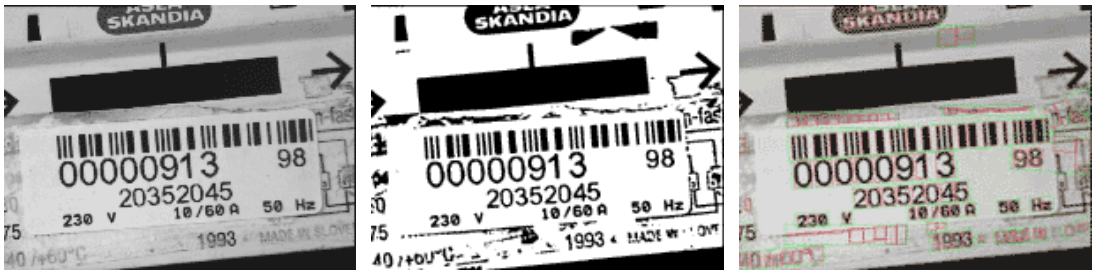
La última etapa es el reconocimiento e interpretación. Con la información de fases anteriores, el reconocimiento etiqueta el objeto. Por ejemplo, reconocer la letra impresa «a» como etiqueta «a»[1]. La interpretación infiere un significado de un conjunto de etiquetas. Al hilo del ejemplo de reconocimiento de caracteres, que, por ejemplo, 5 cifras seguidas sea un código postal en el caso de una carta; tres números separados con barras sean una fecha; un número con dos decimales seguido de «€» represente una cantidad monetaria, etc. En el caso de la figura 3.2g son los datos de voltaje y número de serie en un contador eléctrico.

En el diagrama 3.1 aparece otro elemento que interactúa con todas las fases. Es la base de conocimiento. El conocimiento del dominio del problema está almacenado en ella. En la etapa de adquisición de imágenes, puede ser simplemente imágenes almacenadas. En el preprocesado, zonas de la imagen que se sepan de interés previamente. Para la segmentación en, por ejemplo, un programa de detección de defectos, una lista con las imperfecciones más comunes. En el caso de OCR, todos los caracteres que se quieran reconocer. La base de conocimiento también sirve en caso de realimentación entre distintas fases. Por ejemplo, si en la interpretación se detecta que el día de una fecha tiene tres dígitos, podemos sospechar que se han reconocido erróneamente dos caracteres como tres, y podemos repetir la fase de segmentación para comprobarlo.

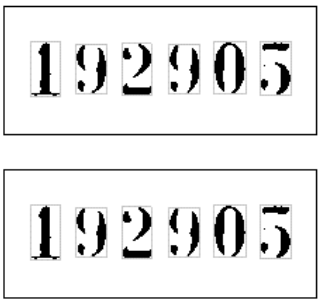
En las figuras 3.3 y 3.4 se muestra otro ejemplo de segmentación, en este caso por regiones (interior coloreado), no por contornos.

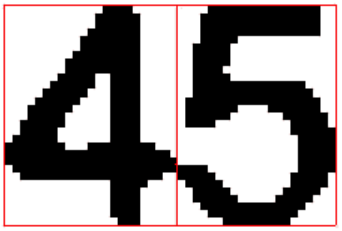
En general, las operaciones del procesamiento de imágenes se pueden clasificar en

[1] En la práctica la etiqueta sería la codificación de la letra «a» en algún código de caracteres como ASCII, Unicode,etc.

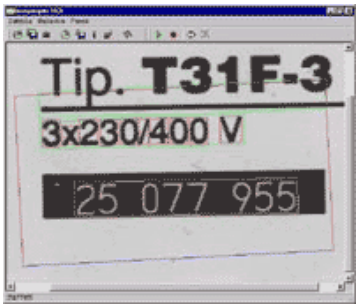


(a) Adquisición **(b)** Preprocesamiento **(c)** Segmentación





(d) Segmentación caracteres **(e)** Conectar **(f)** Reconocimiento caracteres



(g) Interpretación

Figura 3.2: Etapas de procesamiento de caracteres para OCR.

Figura 3.3: Imagen original

Figura 3.4: Imagen segmentada. Las zonas coloreadas representan distintos segmentos

cuatro tipos ([23]):

1. Operaciones de píxel: La salida de un píxel sólo depende de él mismo, no de los demás. La umbralización o *thresholding*, consistente en establecer los píxeles que superan un límite o *threshold* a blanco, y los que no a negro, es una operación de píxel. Otros ejemplos pueden ser el aumento/disminución de brillo, la normalización del contraste, o el negativo (inversión de colores).

2. Operaciones locales, sobre un vecindario: La salida de un píxel depende del valor de los píxeles vecinos. Por tanto, en cada zona los resultados serán distintos. Algunos ejemplos pueden ser la detección de bordes, todo tipo de filtros: suavizado (media y mediana), realce (laplaciana y gradiente).

3. Operaciones geométricas: La salida de un píxel depende de otros píxeles que cumplan una transformación geométrica. Un ejemplo sencillo puede ser voltear una imagen, hacer el simétrico, etc.

4. Operaciones globales: La salida de un píxel depende de *todos* los demás en la imagen. Puede reflejar algún tipo de estadística. Por ejemplo, *distance transformation*, que asigna a cada píxel la distancia mínima entre él y todos los pixeles del fondo.

El tratamiento digital de imágenes tiene múltiples y muy diversas aplicaciones. Se citan a continuación algunas de ellas, y ejemplos de su uso:

- Imagen Satelital: ya sea para uso civil (mapas, meteorología), militar (búsqueda de objetos: personas, carreteras, edificios) o científico (imágenes espaciales del *Hubble*).

- Multimedia y comunicaciones: emisiones de televisión o teleconferencia.

- Medicina: imágenes rayos X, angiograma, tomografía, resonancia magnética, que permiten localizar tumores y otras patologías.
- Radar y sonar, para detección y reconocimiento de diferentes tipos de objetivos y sus maniobras.
- Visión por computadora: identificación y descripción de objetos industriales, reconocimiento facial, control de tráfico en carreteras.

3.2. Estado del arte

En [18] se puede encontrar un repaso a las principales técnicas de segmentación. La elección de una u otra depende de las imágnenes y de la aplicación. El nivel de subdivisión empleada, es decir, cuántas regiones generamos, depende del problema que estemos tratando. En el caso de OCR introducido en la sección anterior, es común encontrarse con documentos de mala calidad, como fotocopia, fax, etc, que tienen caracteres poco legibles. La segmentación puede provocar errores al contar dos caracteres como uno, o uno como dos, etc. Es difícil dividir los caracteres correctamente. Según [23], existen cuatro tipos de técnicas de segmentación:

- Umbralización
- Basados en Bordes
- Basados en Regiones
- Técnicas híbridas

3.2.1. Umbralización.

La umbralización es un método fácil y cómodo para segmentar, basándonos en la diferencia de intensidad de colores entre el fondo y el primer plano. La «diferencia» tiene que ser notable. De lo contrario lo mejor será no utilizar esta técnica y buscar otra. Si esa disimilitud existe, se suele reflejar claramente en el histograma[2], como dos o más picos.

En el histograma de la figura 3.5 se puede observar un máximo a la izquierda, correspondiente al fondo, oscuro, sobre los 90, y otro a la derecha, correspondiente al cuadrado, más claro, con un color de aproximadamente 160. Como se distinguen «picos» bien

[2]El histograma indica las frecuencias de los colores/intensidades.

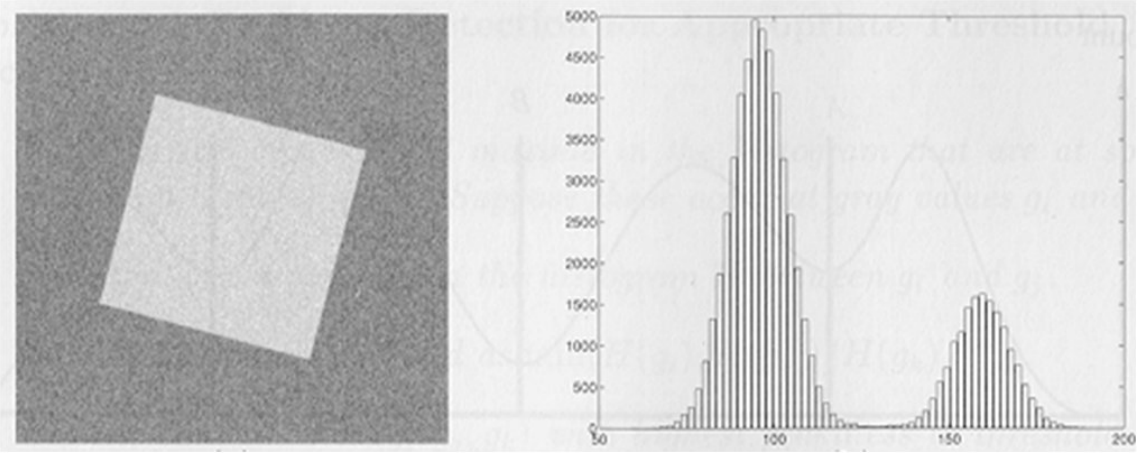


Figura 3.5: Imagen con su histograma a la derecha.

diferenciados, la imagen es una buena candidata a umbralización. Entre éstos es donde se coloca el umbral o *threshold*, para separarlos. En la figura estaría alrededor de 130.

Hemos encontrado un buen umbral «a ojo», pero hacerlo autónomamente no es tan sencillo. Se han propuesto varios métodos para ello. Por ejemplo, [11] utiliza esta expresión:

$$\lambda = \frac{\sigma\sqrt{2\log n}}{\sqrt{n}} \tag{3.1}$$

donde λ es el umbral, σ la desviación típica y n el tamaño total de las muestras.

También existe algún método iterativo:

1. Elegir una estimación inicial del $\lambda_{inicial}$
2. Calcular las medias μ_1 y μ_2 de los grupos que quedan a izquierda y derecha de λ.
3. Calcular el nuevo umbral como $\lambda_{nuevo} = \frac{1}{2}(\mu_1 + \mu_2)$
4. Si la diferencia entre λ_{nuevo} y $\lambda_{inicial}$ es menor que un cierto límite anteriormente prefijado, hemos terminado. De lo contrario volver al paso 1.

3.2.2. Basados en Bordes

Los métodos basados en bordes se apoyan en la discontinuidad entre un píxel y sus vecinos, y usan esa información para separar las regiones. El procedimiento más directo es «detect and link», en el que primero se detectan discontinuidades, y posteriormente se conectan para formar límites más grandes. El problema es que en ocasiones esos límites no están cerrados. Esto es un gran inconveniente para fases posteriores en el procesamiento de la imagen. Para solucionarlo, Yanowitz y Bruckstein [25] plantearon un método de umbralización adaptativa, como los vistos en la subsección anterior, pero que se ayuda del gradiente de

intensidad en los bordes para determinar el umbral inicial. Lo veremos en la subsección de técnicas híbridas.

La detección de discontinuidades es uno de los problemas mejor estudiados en tratamiento digital de imágenes, ya sean puntos, líneas o bordes. El método más común es el llamado filtrado espacial. Consiste en pasar una *máscara* (también conocida como filtro) a través de la imagen, punto por punto. La reacción, o resultado, del filtro en un punto viene dada por una relación predefinida. En el caso del filtrado espacial *lineal*, esa relación es la suma de productos de los coeficientes del filtro y los píxeles correspondientes de la imagen que cubre la máscara (figura 3.6, de [12]).

Siguiendo la figura 3.6, el resultado R de aplicar el filtro es:

$$R = w(-1,-1)f(x-1,y-1) + w(-1,0)f(x-1,y) + \ldots \\ + w(0,0)f(x,y) + \cdots + w(1,0)f(x+1,y) + w(1,1)f(x+1,y+1)$$

Escrito matemáticamente, siendo f la imagen original, w la máscara, g la imagen resultado, con una máscara de tamaño $m \times n$[3], $a = \frac{(m-1)}{2}$ y $b = \frac{(n-1)}{2}$:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t) \tag{3.2}$$

$x = 0, 1, 2, \ldots, M-1$, $y = 0, 1, 2, \ldots, N-1$, siendo la imagen de $M \times N$, para pasar el filtro por todos los puntos de la imagen.

En el dominio de frecuencias[4] existe un concepto similar al filtrado espacial, llamado *convolución*. A veces se utiliza ésta como sinónimo de aquél, con expresiones del tipo «máscara de convolución», aunque técnicamente no sea correcto: filtrado espacial pertenece al dominio espacial, y convolución, al de frecuencias.

También existen otros filtros no lineales, es decir, que no utilizan la suma de productos, sino otras operaciones, como calcular la media, mediana, o varianza de los píxeles de la imagen original que cubre la máscara.

Para el propósito de esta sección, lo que me interesa es mostrar la existencia de filtros que nos ayudan a detectar bordes. En la figura 3.7 podemos observar el efecto de máscaras de Robers, Prewitt y Sobel. Explicar cómo se obtienen, sin embargo, requiere cierta formulación matemática y excede el propósito de este trabajo.

[3] Asumimos que tanto m como n son enteros no negativos e *impares*. La imparidad es para que quede siempre un píxel en la mitad, sobre el que la máscara quede centrada.

[4] Las imágenes se pueden tratar en dominio espacial o en dominio de frecuencia. El dominio espacial usa directamente los píxeles. El dominio de frecuencia emplea la *transformada de Fourier* de la imagen.

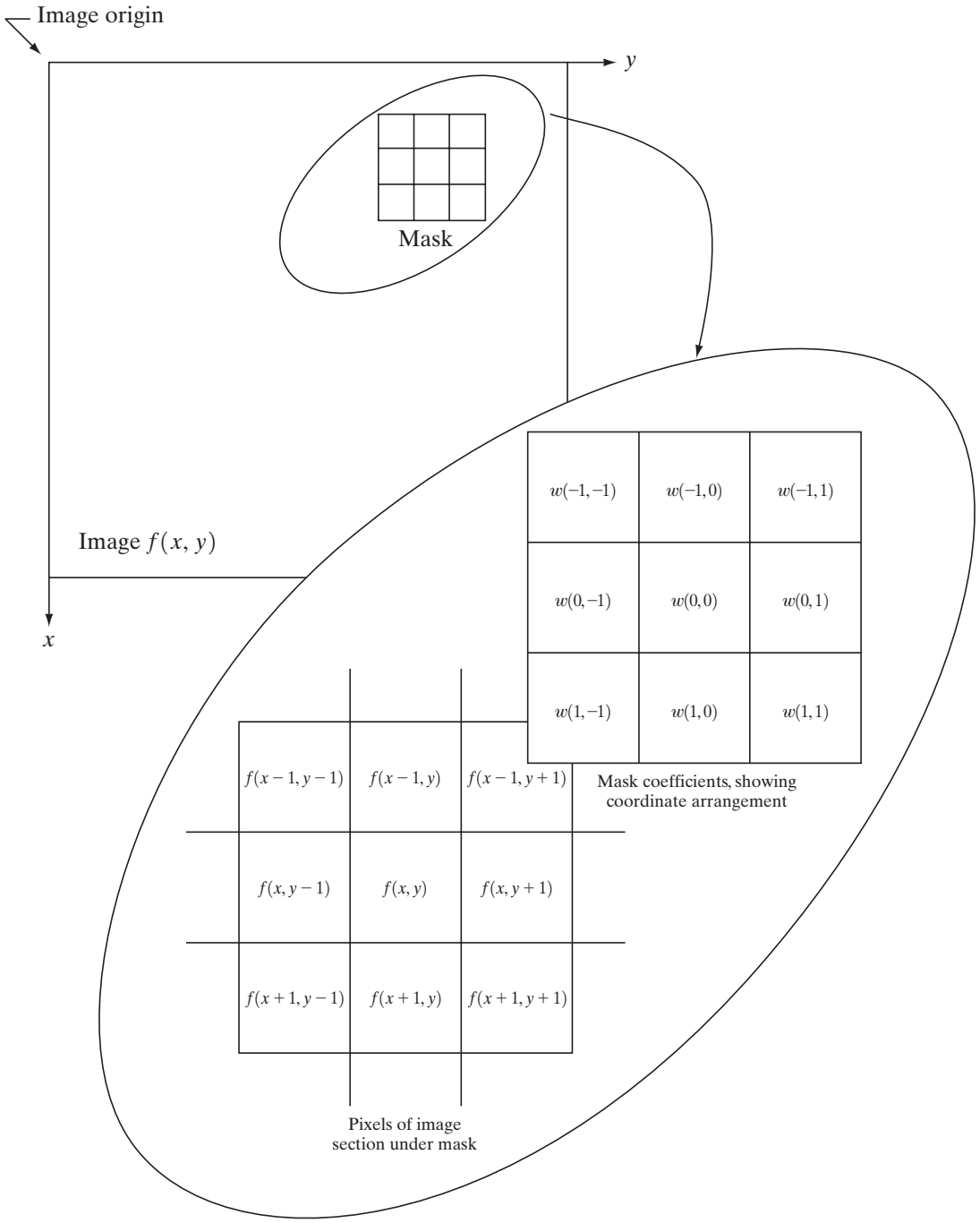


Figura 3.6: Aplicando filtrado espacial a una imagen. La zona aumentada muestra la máscara de 3x3, y los píxeles de imagen original debajo de ella (desplazados para que se puedan ver).

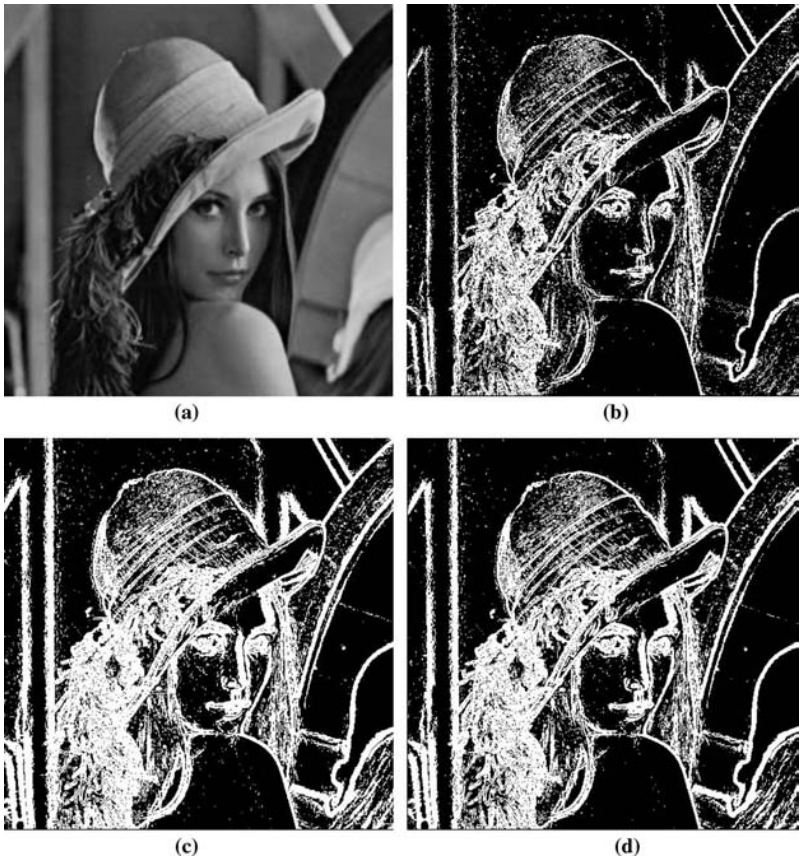


Figura 3.7: La imagen de Lena (a) y las imágenes resultantes de aplicar (b) Roberts, (c) Prewitt, y (d) Sobel.

3.2.3. Basados en Regiones

Los métodos basados en regiones, al contrario que los basados en bordes, se apoyan en la similitud de un píxel con sus vecinos, agrupando puntos de características semejantes (por ejemplo, de intensidad parecida) en regiones. El lugar donde empiece el proceso se llama semilla, y puede ser un píxel o grupo de ellos. Buscar y seleccionar semillas adecuadas es muy importante. Si se eligen descuidadamente y caen en medio de dos regiones, los algoritmos no podrán detectar o corregir el error. También fallan cuando la definición de «uniformidad» es demasiado estricta: por ejemplo, si exigimos que una región tenga una intensidad constante cuando en realidad varía linealmente. Otro inconveniente es la dependecia del orden en que los píxeles se inspeccionan, aunque no es realmente un problema porque las diferencias suelen ser mínimas.

Existen dos enfoques: crecimiento de regiones y división de regiones. Crecimiento hace la suposición inicial de que la imagen está muy segmentada. Al comienzo, los conjuntos examinados, las semillas, son muy pequeñas, y se expanden para incluir zonas homogéneas. El proceso se repite hasta que no quede ningún píxel por clasificar. Veremos crecimiento con más detalle en la sección 3.3.

División de regiones, por el contrario, supone inicialmente que la imagen está poco segmentada. Al principio, toda la imagen constituye la semilla. Si ésta no es homogénea, se divide en un número predeterminado de subregiones, normalmente cuatro. El proceso se repite usando cada subregión como una nueva semilla. El algoritmo termina cuando todas las subregiones son homogéneas. Dado que las semillas son muy grandes, división de regiones se ve menos afectada ante el ruido que crecimiento, donde las semillas son pequeñas, y píxeles vecinos ruidosos pueden alterar el resultado.

En los años 70, Horowitz y Pavlidis [25] diseñaron un algoritmo híbrido, «división, unión y agrupamiento» (*split, merge and group (SMG)*), que combina lo mejor de crecimiento y división de regiones. También es más eficiente. El inconveniente es que en la imagen resultado se observan regiones similares a las estructuras de datos empleadas: un quadtree[5] en imágenes 2D y un K-tree en 3D.

3.2.4. Técnicas híbridas

Hoy en día, se conoce bien que los procedimientos de segmentación vistos anteriormente no siempre producen una segmentación precisa. Por ello existe la tendencia de combinar varias técnicas para lograr mejores resultados. Por ejemplo, se ha mezclado detección de bordes con crecimiento de regiones (Puvlius y Liov; Harris en al; Fan et al). Los

[5] Árbol que siempre tiene cuatro ramas.

métodos más conocidos dentro de esta clase son *morphological watershed*, *variable-order surface fitting*, y *active contour*.

Como ejemplo, vamos a mostrar aquí el que ya aventuramos en 3.2.2, la umbralización adaptativa de Yanowitz y Bruckstein, que usa detección de bordes junto con umbralización. En la figura 3.8, de [25], se puede ver el resultado.

El proceso en concreto consiste en los siguientes pasos:

1. Suavizar la imagen, reemplazando cada píxel con la media de los vecinos.

2. Determinar el gradiente de la imagen suavizada. Se calcula como diferencia de intensidad en una distancia pequeña:

$$G(i,j) = min(I(i,j) - I\left(i+\delta_i, j+\delta_j\right)) \tag{3.3}$$

$$\delta_i = -1, 1, \delta_j = -1, 1 \tag{3.4}$$

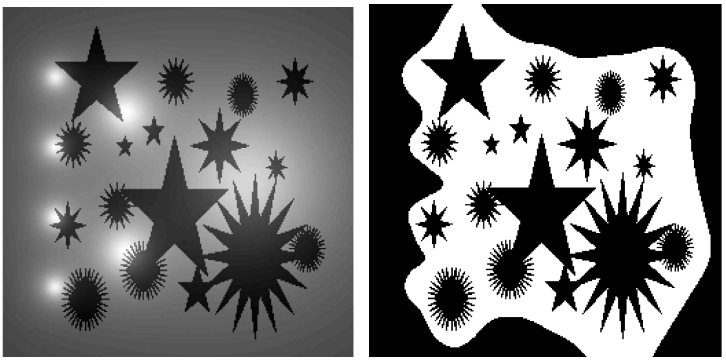
   donde I es la imagen, y G la imagen con las diferencias aplicadas.

3. Quedarse sólo con los puntos de gradiente máximo para hacer más fino el borde, que debería ser de un píxel de ancho.

4. Leer la intensidad de los píxeles en los puntos del borde.

5. Interpolar los valores obtenidos en el paso anterior por toda la imagen. Cada píxel tendrá dos valores: uno el original, y otro el interpolado.

6. Segmentar la imagen empleando umbralización. Si el valor del píxel original es mayor o igual al interpolado, lo establecemos a 1 (blanco). En caso contrario, a 0 (negro). Así pues, los objetos serán blancos, y el fondo, negro.



(a) Imagen Original. **(b)** Imagen resultado.

Figura 3.8: Umbralización adaptativa de Yanowitz y Bruckstein.

3.3. Crecimiento de Regiones

3.3.1. Idea principal del algoritmo de segmentación.

Crecimiento de regiones (*Seeded Region Growing*) es una técnica híbrida de segmentación, propuesta por Adams y Bischof[23] en 1994. Aprovecha que píxeles espacialmente cercanos tienen valores de intensidad similares. Comienza solamente con un píxel, denominado *semilla*, al que se van añadiendo otros próximos progresivamente. El proceso consiste básicamente[6] en tres pasos:

1. Elegir un punto semilla.
2. Comprobar los puntos vecinos y añadirlos a la región si son similares a la semilla.
3. Repetir el segundo paso para cada uno de los nuevos puntos añadidos, hasta que no quede ninguno que poder incluir.

Un esquema de lo anterior:

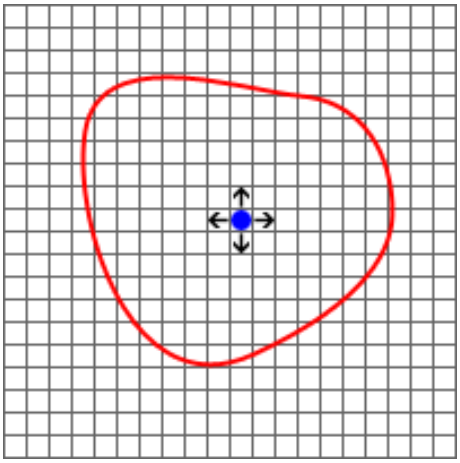


Figura 3.9: Imagen original

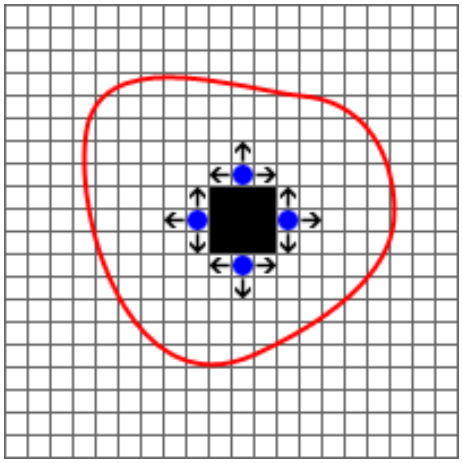


Figura 3.10: Imagen tras haber aplicado una primera iteración de crecimiento de regiones

En la figura 3.9 se puede observar la semilla, de color azul, y las flechas que indican los puntos vecinos[7]. La línea en rojo representa el contorno segmento que deberíamos obtener al final del proceso. A la derecha (figura 3.10) se muestra el estado una vez acabada la primera iteración. Puesto que los puntos vecinos son similares a la semilla (todos están dentro del segmento) se incluyen en la región, coloreada en negro. Al igual que antes, se

[6] Ver el proceso detallado en 3.3.4.

[7] En este caso, la vecindad es de cuatro, por lo que no se consideran puntos diagonales, que sí serían tenidos en cuenta en una vecidad de ocho.

vuelven a comprobar los vecinos de esos nuevos puntos. Este proceso es bastante similar al de una inundación. El agua sale de un manantial (semilla) y va anegando los alrededores paulatinamente: antes los más próximos, después los más lejanos.

3.3.2. El proceso completo de crecimiento de regiones

El proceso completo de crecimiento de regiones consta de más pasos, además de la segmentación propiamente dicha:

1. Convertir la imagen del espacio de color RGB al YCbCr.
2. Selección automática de semillas. Cada semilla forma inicialmente una región.
3. Se emplea segmentación por crecimiento de regiones con la idea vista en la subsección anterior.
4. Finalmente, se aplica un algoritmo para juntar regiones, por el que las más pequeñas se unen a vecinas similares.

En referencia al primer paso, las cámaras digitales recogen imágenes usando el espacio de color RGB. Este formato es adecuado para mostrarlas en un monitor, pero no para su análisis, debido a la alta correlación entre sus componentes. En su lugar, se suele emplear YUV, que separa luminancia (Y) cromancia (U) y saturación (V). Éste, además, se adapta mejor a la visión del ojo humano, por ser más sensible a las variaciones de luminosidad que de color. YCbCr es una versión escalada y desplazada de YUV.

3.3.3. Selección automática de semillas

Los puntos candidatos a semillas deben cumplir tres condiciones. La primera, que sean lo suficientemente similares a sus vecinos. La segunda, que para una región esperada, haya al menos una semilla que la genere. La tercera, que semillas de diferentes regiones deben ser inconexas (es decir, que no estén unidas por otras semillas).

Antes de ver formalmente dichas condiciones, necesitamos dos conceptos, la similitud, y la distancia, de un píxel con sus vecinos. Empecemos por la similitud:

Primero obtenemos la desviación estándar. Considerando un vecindario de 3×3:

$$\sigma_x = \sqrt{\frac{1}{9}\sum_{i=1}^{9}(x_i - \overline{x})^2}$$

donde x puede ser Y, C_b o C_r, y la media $\overline{x} = \frac{1}{9}\sum_{i=1}^{9} x_i$. La desviación total es la suma de las desviaciones de cada componente:

$$\sigma = \sigma_Y + \sigma_C b + \sigma_C r$$

La desviación estándar se normaliza a [0,1] mediante

$$\sigma_N = \sigma / \sigma_{max}$$

donde σ_{max} es el máximo de la desviación estándar en la imagen. La similitud H de un pixel con sus vecinos es «lo contrario» a la desviación:

$$H = 1 - \sigma_N$$

La distancia de un píxel a sus vecinos se calcula como la distancia euclídea relativa, que experimentalmente da mejores resultados que la distancia euclídea normal.

$$d_i = \frac{\sqrt{(Y - Y_i)^2 + (C_b - C_{b_i})^2 + (Cr - C_{r_i})^2}}{\sqrt{Y^2 + C_b^2 + C_r^2}}, i = 1, 2, \ldots, 8 \tag{3.5}$$

Veamos, ahora sí, las condiciones para que un píxel sea semilla:

- Condicion 1: La similitud H debe ser mayor que un umbral, que se calcula mediante el método de Otsu.
- Condicion 2: El máximo de las 8 distancias relativas euclídeas de un punto con sus vecinos $d_{max} = max_{i=1}^{8}(d_i)$ debe ser menor que un umbral[8]. Esta condición comprueba que la semilla no esté en el límite entre dos regiones.

Es posible que se seleccionen varios píxeles cercanos como semillas. En ese caso, se comprueba si son conexos, y si lo son, se tratan como una sola semilla. También puede ocurrir que en una misma región esperada aparezcan varias semillas, que la segmentarán en exceso. Para solucionar este problema, una fase posterior junta las que sean similares.

Valores mayores del umbral de distancias disminuyen el número de semillas, por lo que pueden quedar objetos sin segmentar. Por el contrario, umbrales bajos generan demasiadas, que si son conexas pueden segmentar de menos, formando una sola región cuando en realidad son varias.

[8]En [23] se escoge manualmente un valor de 0,05.

3.3.4. El algoritmo de segmentación

Aquí describo lo mismo que en 3.3.1, pero más detallado y con vistas a la implementación:

1. Asignar una etiqueta indentificativa de región a cada semilla/grupo de semillas conexas.
2. Guardar los vecinos de las regiones (el contorno de éstas) en una lista ordenada T, en orden decreciente de distancias.
3. Mientras T no esté vacía, extraer el primer punto p[9] y comprobar sus 4-vecinos. Si todos ellos tienen etiqueta asignada con el mismo identificador de región, establecer la etiqueta de p a ese identificador. Si, por el contrario, los vecinos pertenecen a regiones distintas, clasificar a p en la región que esté a menor distancia de él[10]. A continuación, actualizar la media de color de la región a la que p ha sido añadido. Incluir los 4-vecinos de p que ni estén previamente en T ni tengan etiqueta, a la lista T, en orden decreciente de distancias.
4. Juntar regiones.

En el paso 3, la distancia entre un punto y la región se calcula de forma parecida a como vimos en la expresión (3.5), solo que en lugar de emplear el color de un punto, empleamos la media de toda la región:

$$d_i = \frac{\sqrt{(Y_i - \overline{Y})^2 + (C_{b_i} - \overline{C_b})^2 + (C_{r_i} - \overline{C_r})^2}}{\sqrt{Y_i^2 + C_{b_i}^2 + C_{r_i}^2}}, i = 1, 2, \ldots, 8 \tag{3.6}$$

donde $(\overline{Y}, \overline{C_b}, \overline{C_r})$ son la media de las componentes Y, C_b, C_r de la región. Por poder, se podría tomar otro criterio aparte de la media para decidir la inclusión de un punto en una región. Por ejemplo, considerar sólo la semilla inicial. Es una condición muy sencilla, pero no suele dar buenos resultados, ya que la región producida es muy sensible a la elección de ésta. Otra opción es comparar los píxeles del borde ya pertenecientes a la región con los vecinos que se está estudiando añadir, pero esto es poco efectivo debido a la alta divergencia que conlleva: si la semilla es parecida a p, y p a q, puede que la semilla y q sean demasiado diferentes como para etiquetarlas en una misma región. La media, sin embargo, es el equilibrio ideal. Proporciona cierta flexibilidad, porque los requerimientos de inclusión cambian, pero a la vez cierto control, ya que los píxeles añadidos hasta entonces hacen de «contrapeso».

[9] El primer punto (elemento) de la lista es el de menor distancia, puesto que la lista T es ordenada. En caso de que haya varios píxeles con el mismo valor mínimo, se elige aquel que tenga como vecina la región de mayor tamaño.

[10] Si todas las regiones están a la misma distancia, incluirlo en la más grande.

3.3.5. Juntar regiones

Antes hemos comentado que la imagen se segmenta en exceso si el umbral de la distancia euclídea es demasiado bajo. Para compensarlo, se juntan las regiones. Existen dos criterios para ello: la media y el tamaño.

Dos regiones se juntan si la diferencia de medias es menor que un umbral. El problema es que la media depende del orden en el que se examinen. Por tanto, se deben comprobar los pares de menor distancia primero. En cada iteración, si se juntan, se recalcula la media de la región resultado de la unión, así como las distancias al resto. El algoritmo termina cuando no quede ningún par con distancia menor que el umbral escogido. Si éste es demasiado alto, se juntarán regiones que no deberían hacerlo. Por el contrario, si es bajo, la imagen seguirá segmentada en exceso, que es precisamente lo que queremos evitar con todo este proceso.

Después, entra en juego el tamaño. Si una región tiene un número de píxeles menor que un cierto umbral, se junta con la vecina de media más cercana. Este proceso termina cuando no quede ninguna de tamaño menor que dicho umbral.

3.3.6. Diferencias entre este algoritmo y el mío

Realmente, el algoritmo que he implementado mantiene muchas de las ideas que acabamos de ver, pero es mucho más descafeinado. Por ejemplo, no se hace una selección previa de semillas. Simplemente divido la imagen en cuadrículas (*tiles*), y el centro de esa cuadrícula se toma como semilla. Al comparar un píxel con sus vecinos, solamente tengo en cuenta la distancia relativa euclídea, y no la similitud H. Esto es para simplificar la implementación. Aparte, hay otros aspectos que, debido a la necesidad de paralelismo, se realizan de forma distinta. Por ejemplo, a la hora de segmentar no utilizo ninguna lista ordenada, ya que eso está pensado para un algoritmo secuencial. En cada iteración, todos los píxeles del contorno se evalúan a la vez, en aras de una mayor velocidad de procesamiento. Todas estas cuestiones se verán con más detalle en el capítulo de implementación.

Capítulo 4

CUDA

Esta sección es una muy breve introducción a los conceptos básicos de CUDA, necesarios para entender la implementación, que se incluye en el siguiente capítulo.

4.1. Introducción

Durante los años 80 y 90, el hardware de las GPUs estaba diseñado como un *pipeline* en el que cada componente tenía una función fija, es decir, que estaba diseñado específicamente para un fin. Esto se puede observar en la figura 4.1, extraída de [15]. Por ejemplo, *vertex control* recibe los tres vértices de un triángulo, y los almacena en la *cache* de vértices. VS/T&L se encarga de aplicarles transformaciones, y de almacenar en cada uno de ellos información de *shading* (coordenadas de texturas), color, e iluminación. «Raster» averigua qué píxeles están contenidos en un triángulo, e interpola la información de los vértices obtenida anteriormente. Para acceder a todo este sistema, se hace uso de las APIs DirectX y OpenGL. Los comandos emitidos por éstas desde la CPU son recogidos por «host interface» en la GPU, que también devuelve los resultados de ejecución.

Esta arquitectura funciona bien, y es bastante *configurable*. Sin embargo, no es *programable*. Se pueden elegir entre multitud de opciones, pero sin crear otras nuevas. Ofrecer una nueva funcionalidad implica normalmente rediseñar el hardware, ya que estaba pensado para una función muy concreta (como las vistas en el párrafo anterior), y no otra. Esto es poco flexible. Los desarrolladores de software necesitan más libertad. La implementación concreta de cada uno de los pasos vistos en la figura 4.1 es distinta para cada algoritmo de renderización. Esto motivó a introducir cierta capacidad de programación en algunas fases del *pipeline*, sobre todo en el *vertex shader* y *pixel shader*. Se proporcionaron, a través de DirectX y OpenGL, los «vertex shader extensions», nuevos juegos de instrucciones del *vertex engine*[1], acceso de los *pixel shader processors*, etc. Para lograr esto, se hace necesario un procesador programable, en contraposición a una circuitería que siempre haga lo mismo.

Algunas personas intentaron desarrollar aplicaciones no gráficas, de propósito general, al darse cuenta del potencial de las GPUs. Sin embargo, la capacidad de programación en

[1] Parte del VS/T&L

ese momento sólo estaba pensada para fuciones 3D, usando las APIs DirectX y OpenGL. Una aplicación de propósito general tenía que ser «adaptada», como si fuese un problema gráfico, para poder ser implementada. Por ejemplo, para ejecutar en paralelo varias instancias de una misma función, había que reescribirla como un *pixel shader*. Los datos a procesar tenían que ser guardados como una textura, y enviadas a la GPU como triángulos. Los datos devueltos eran un conjunto de píxeles, que debían ser procesados de alguna manera para obtener un resultado final. No hace falta decir que todo esto es tremendamente difícil, por lo que no tuvo mucho éxito.

4.1.1. Surgimiento de la arquitectura unificada

En un principio, los procesadores programables que hemos comentado estaban separados, según la fase del *pipeline* a la que estuviesen destinados. Por ejemplo, en GeForce 6800 y 7800, había uno para vértices y otro para píxeles. Sin embargo, los principales fabricantes apostaron por juntarlos en un solo grupo o *array*, dando lugar a lo que se conoce como arquitectura *unificada*. XBox 360, con su GPU Xenos de ATi, fue la primera en utilizarla, con un mismo procesador calculando tanto píxeles como vértices. Existen varios motivos que dieron lugar esta transición: uno es una mejor distribución de carga de trabajo entre las distintas etapas del *pipeline* gráfico. Esto es importante porque, como comentamos anteriormente, algoritmos de renderización distintos requieren diferentes implementaciones de cada una de las fases, con exigencias de rendimiento que pueden ser muy diversas. Una segunda utilidad es la mayor facilidad para incrementar la frecuencia de reloj respecto a diseños con procesadores separados, por la distancia que tienen que recorrer las señales eléctricas. Pero sin duda el gran atractivo es la escalabilidad. En el procesamiento de gráficos 3D, existe una notable independencia de datos en algunas fases del *pipeline*. Por ejemplo, una misma transformación geométrica que se realiza sobre multitud de vértices. Esa independencia favorece la implantación de procesadores masivamente paralelos. Con sólo aumentar el número de núcleos, mejoramos automáticamente el rendimiento, y sin necesidad de hacer cambios profundos en el diseño hardware. Esto es beneficioso para el mercado, ya que facilita la creación de varias gamas de GPUs (móviles, portátiles, sobremesa, etc), que funcionan de la misma forma, con el mismo software, pero con rendimiento distinto.

Las GPUs G80, entre la que se encuentra GeForce 8800, fueron las primeras de nVIDIA en implantar arquitectura unificada: las partes programables del *pipeline* gráfico se ejecutan en un mismo conjunto o *array* de procesadores. En la figura 4.2 se puede observar un camino de datos circular que pasa tres veces por ellos: primero «Vertex thread issue», después «Geometry Thread Issue», y por último «Pixel Thread Issue»[2]. Los resultados de una

[2]Son las tres fases programables: vertex shading, geometry processing, y pixel processing.

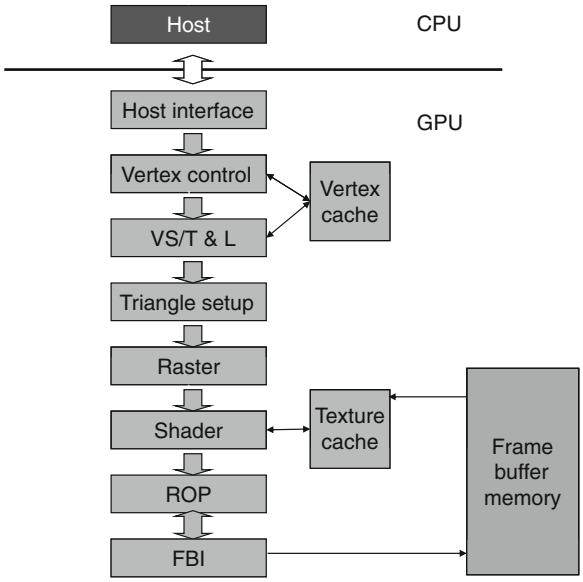


Figura 4.1: Pipeline gráfico antiguo. Hardware específico se implementaba para cada una de las fases del proceso de renderizado y sólo tenía esa función.

vuelta sirven como entrada en la siguiente, con partes fijas (no programables) entre medias: por ejemplo, antes de la etapa de procesamiento de pixels, es necesaria Triangle Setup, Raster y Z-Culling[3].

Sin embargo, nVIDIA no sólo introdujo la arquitectura unificada, sino que además se añadieron características destinadas a la computación de propósito general. Se incorporó un *shader processor* totalmente programable, con una gran memoria y caché de instrucciones[4]; nuevas instrucciones de carga y escritura sobre cualquier posición de memoria; un modelo de ejecución con jerarquías de hilos, barreras de sincronización, operaciones atómicas; ALUs que cumplían la especificación IEEE para punto flotante de simple precisión... A esta arquitectura, nVIDIA la bautizó como CUDA: COMPUTE UNIFIED DEVICE ARCHITECTURE. Una **A**rquitectura **U**nificada para **D**ispositivo[5] orientada a **C**omputación de propósito general. Junto con ella, los compiladores, librerías y runtime API nos permiten programar para los nuevos procesadores gráficos.

[3]En el esquema aparece abreviado como Setup/Rstr/ZCull.

[4]Estos elementos hardware eran compartidos por todos los *shader processors*, por lo que el coste económico no era elevado, y además se favorecía el paralelismo de datos, al aplicar el mismo programa shader sobre múltiples vértices.

[5]Dispositivo o *device* es como la literatura de CUDA denomina a la GPU.

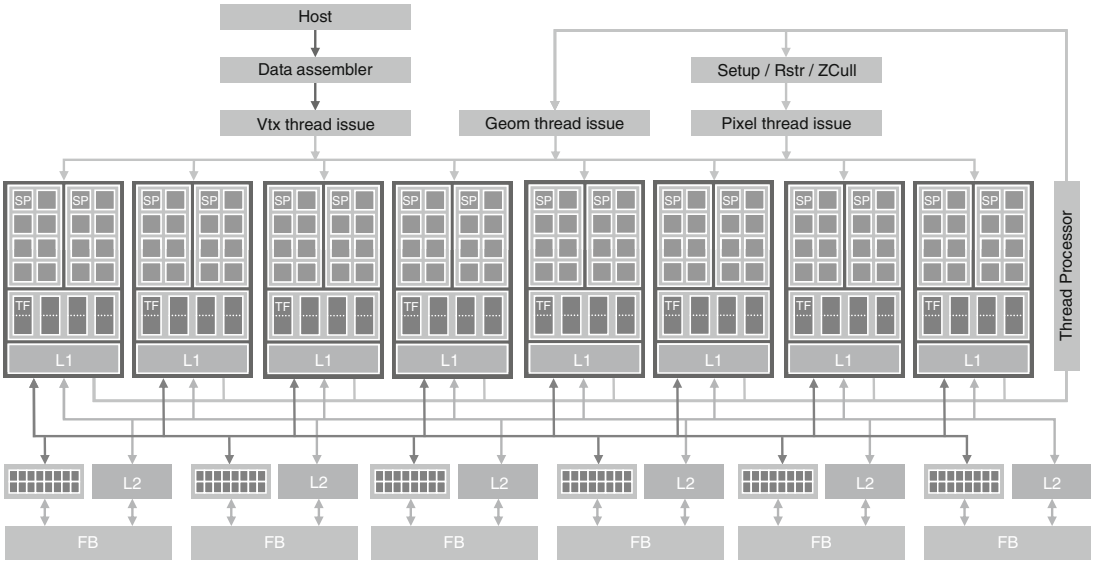


Figura 4.2: Pipeline gráfico nuevo, utilizando arquitectura unificada (CUDA). El mismo array de procesadores unificados sirve para tres fases: vertex shading, geometry processing, y pixel processing. Los datos dan, por tanto, tres vueltas.

4.2. Programación en CUDA

A pesar de todas las nuevas características de la arquitectura CUDA, seguía sin haber ningún método para acceder a ellas, aparte de las APIs gráficas. Por ello, NVIDIA no sólo introdujo hardware para computación, sino que también puso a disposición de los programadores las herramientas de desarrollo adecuadas. Para llegar a más gente, se tomó un lenguaje bien conocido en la industria como es C, y se le añadieron un pequeño número de extensiones o palabras clave para acceder a las funcionalidades que ahora el hardware permitía. Pocos meses después del lanzamiento de GeForce 8800GTX, nVIDIA distribuyó un compilador para «CUDA C», además de *drivers* para varios sistemas operativos. En esta sección veremos algunas nociones y aspectos prácticos necesarios para la programación en CUDA.

En CUDA, se denomina *host* a la CPU y *device* o dispositivo a la GPU, que, como hemos visto, es un procesador masivamente paralelo. Para sacarle partido necesitamos paralelismo o, lo que es lo mismo, independencia, de datos. Un ejemplo clásico es la multiplicación de matrices (ver figura 4.3). En ella, un elemento de la matriz resultado P es el producto escalar de una fila de M y una columna de N. Todos los elementos de P se pueden calcular simultáneamente, ya que son independientes uno de otro.

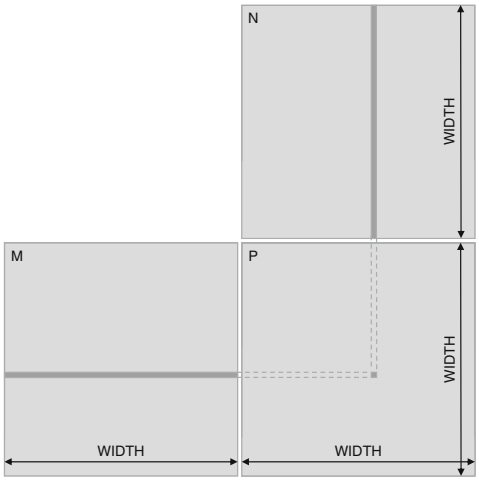


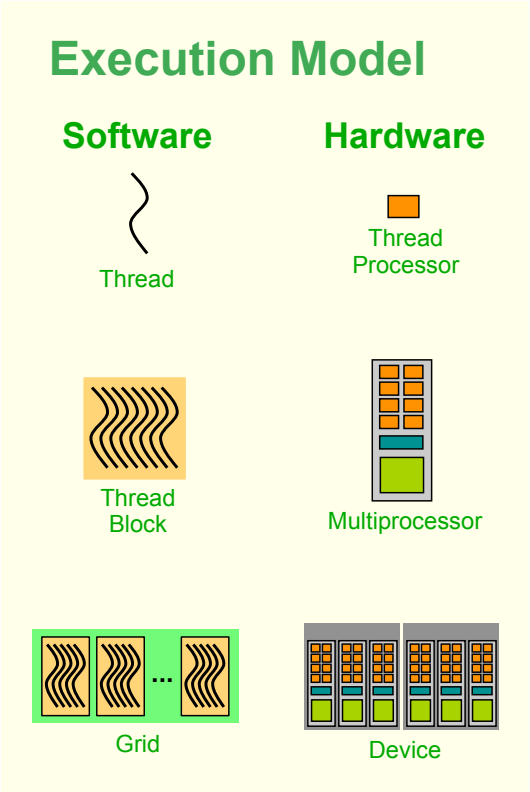
Figura 4.3: Paralelismo (independencia) de datos en la multiplicación de matrices.

Si un algoritmo no presenta en absoluto paralelismo, lo mejor es utilizar solamente CPU, mucho más eficiente para código en serie, y con más lógica de control (predictores de salto, planificación dinámica, etc). Sin embargo, en la práctica solemos encontrarnos con una situación intermedia, en la que ciertas partes del algoritmo presentan paralelismo y otras no. Esto no supone un problema porque el API de CUDA está perfectamente preparado para mezclar ejecución de CPU y GPU. Esto se conoce como *computación heterogénea*.

Un *kernel* es una función que se ejecuta en el dispositivo (GPU). En el código

se indica con una extensión del lenguaje C, el calificador `__global__`. Cada kernel es ejecutado por múltiples hilos en paralelo, que recorren el mismo código. Este estilo de programación es conocido como SPMD (Single-Program, Multiple-Data), muy común en computadores paralelos masivos. SPMD es distinto a SIMD (Single-Instruction, Multiple-Data). En SPMD las unidades de procesamiento ejecutan el mismo *programa* sobre muchos datos. No tiene por qué ser la misma instrucción. Por el contrario, en SIMD todos tienen que ejecutar exactamentae la misma instrucción en cualquier instante de tiempo.

Los hilos se organizan en *bloques*, que pueden estar organizados en una, dos o tres dimensiones. Los bloques, a su vez, forman el *grid*, de una o dos dimensiones. Los hilos se identifican dentro del bloque con estructuras predefinidas: threadIdx.x en una dimensión; threadIdx.x y threadIdx.y en dos; threadIdx.x, threadIdx.y y threadIdx.z para tres. Tenemos esta misma organización para blockIdx, threadDim y blockDim. Éstas dos últimas devuelven el número de hilos por bloque y número de bloques por grid respectivamente, en la dimensión indicada en los campos de la estructura. Se debe indicar en la llamada al kernel, utilizando otra extensión de C, la *configuración de ejecución*, que básicamente es el número de bloques en el grid, el número de hilos por bloque, y las dimensiones en las que se organizan.

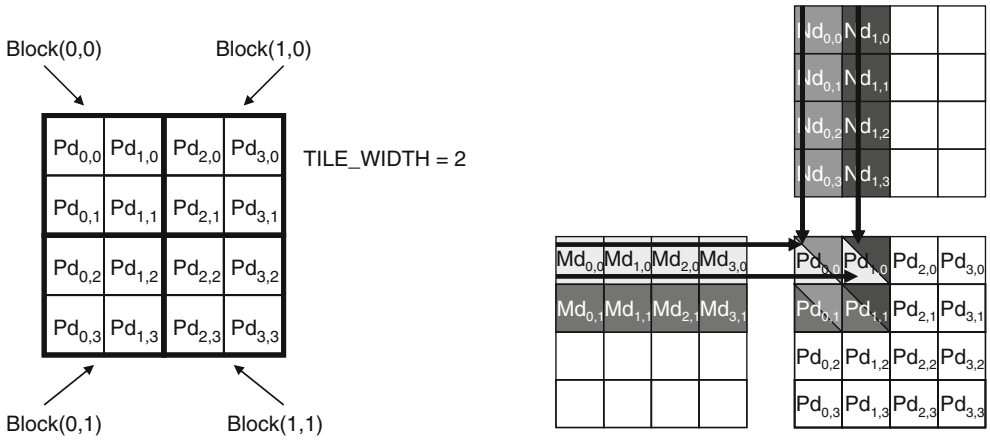


La figura 4.4 muestra en su parte izquierda el modelo de ejecución de CUDA [3], organizado en hilos, bloques y grid. A la derecha aparece el hardware sobre el que se ejecuta cada uno: los hilos en los thread processors, también conocidos como Stream Processors (SPs); los bloques en los multiprocesadores o Stream Multiprocessors (SMs); y los bloques restantes del grid van llenando todos los SMs disponibles en el dispositivo. Como curiosidad, en la parte indicada como «device» aparecen dos bloques grises coneniendo tres SMs cada uno. Son los llamados *building blocks*, y tienen que ver con el encapsulamiento y montaje físico de los chips realizados en fábrica. El número de SMs por building block varía de una GPU a otra.

Figura 4.4: Modelo de ejecución en CUDA.

Veamos una vez más el ejemplo de multiplicación de matrices para aplicar todo lo visto en esta sección:

En 4.5a podemos observar una matriz, que hemos particionado en cuatro bloques



(a) Particionamiento de la matriz en 4 bloques de hilos

(b) Multiplicación de matrices en uno de los bloques

Figura 4.5: Multiplicación paralela de matrices.

de hilos/*tiles*[6] de cuatro hilos/elementos cada uno. A la derecha, en 4.5b, las operaciones realizadas por el bloque superior izquierdo. M y N son las dos matrices a multiplicar, y P la matriz resultado. Md, Nd y Pd son elementos de dichas matrices. `TILE_WIDTH` es el ancho del *tile*, y Width el ancho total de la matriz[7]. El funcionamiento del kernel (listado 4.1) es sencillo: se calcula la fila y columna[8] del elemento Pd resultante, que coinciden, respectivamente, con la fila de M y la columna de N que vamos a procesar. Se recorre horizontalmente la fila de M, y verticalmente la de N, multiplicando los valores y sumándolos en la variable automática Pvalue. Finalmente, ésta se guarda en memoria global, ya sea para devolver el resultado a CPU, o para un procesamiento futuro por parte de otro kernel.

Tenemos que ser conscientes de que *todos* los elementos de la matriz resultado se están calculando en paralelo, *simultáneamente*. En este caso, no es necesario ningún tipo de sincronización y/o comunicación, ya que los datos son totalmente independientes unos de otros.

La llamada al kernel (listado 4.2) se compone de:

- **Configuración de ejecución**, entre comillas latinas triples. En primer argumento es dimGrid, el número de bloques de hilos por grid, de dos dimensiones (`Width /`

[6]Tile o «baldosa» es cada una de las partes en las que hemos subdividido la matriz. Tile es una zona física, mientras que «bloque de hilos» tiene un sentido lógico. Los hilos pertenecientes al bloque operan sobre los elementos del tile.

[7]Como utilizamos un hilo por cada elemento de la matriz, `TILE_WIDTH` se podría sustituir por blockDim, y Width por gridDim*blockDim. Esto no podría hacerse en caso de que un hilo se ocupase de varios elementos.

[8]Como lógicamente estamos trabajando en dos dimensiones, pero los punteros utilizados son de una dimensión (un nivel de indirección), «aplanamos» (*flatten*) las dos dimensiones a una, haciendo Row*Width + Col.

`TILE_WIDTH, Width / TILE_WIDTH`. El segundo, dimBlock, es el número de hilos por bloque, también de dos dimensiones (`TILE_WIDTH, TILE_WIDTH`).

- **Parámetros** pasados al kernel, igual que en las funciones tradicionales de C. Éste no es el caso, pero si pasamos punteros a memoria, tienen que apuntar a memoria *del dispositivo*[9], no a memoria *del host*.

Listado 4.1: Kernel de multiplicación de matrices usando varios bloques

```
__global__ void MatrixMulKernel (float* Md, float* Nd, float* Pd, int
    Width)
{
  //Calculate the row index of the Pd element and M
  int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
  //Calculate the column index of Pd and N
  int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

  float Pvalue = 0;
  //each thread  computes one element of the block sub-matrix
  for (int k=0; k<Width; k++)
    Pvalue += Md[Row*Width + k] * Nd[k*Width + Col];

  Pd[Row*Width + Col] = Pvalue;
}
```

Listado 4.2: Código host para lanzar el kernel del listado 4.1

```
  //Setup the execution configuration
  dim3 dimGrid (Width / TILE_WIDTH, Width / TILE_WIDTH);
  dim3 dimBlock (TILE_WIDTH, TILE_WIDTH);

  //Launch the device computation threads!
  MatrixMulKernel <<<dimGrid, dimBlock>>> (Md, Nd, Pd, Width);
```

4.3. Asignación y Planificación

Cuando se lanza un kernel, el runtime sigue estos pasos:

1. Planificación de bloques: Asignación de bloques a multiprocesadores (figura 4.6).
2. Planificación de hilos por grupos de *warps* (figura 4.7).

[9]Se reserva mediante la llamada cudaMalloc del API de CUDA.

4.3.1. Planificación de bloques

CUDA reparte todos los bloques del grid entre los SMs libres, siguiendo un sistema *round-robin*. Por ejemplo, GT200 tiene 30 multiprocesadores. Si el grid consta de 30 bloques, se asignará uno a cada SM. Si por el contrario, tenemos que distribuir 60, tocarán a 2 por multiprocesador. En todas las GPUs de nVIDIA, el número máximo de bloques por SM es 8, siempre que se satisfagan los recursos exigidos por cada uno, es decir, que *entre todos* no superen el número máximo de hilos, registros o memoria compartida por multiprocesador. Si esto no se cumple, CUDA calcula el número de bloques máximo que pueda asignar al SM para que se dé a *todos* los bloques lo que piden. Por ejemplo, supongamos que la restricción son los hilos, y que andamos sobrados de registros y memoria compartida. En GT200 el número máximo de hilos por multiprocesador es 1024. Si lanzamos bloques de 512 hilos[10], CUDA asignará sólo 2 bloques por SM, ya que 2x512=1024. Con 256 hilos por bloque, habrá 4 bloques/SM; para 128 hilos/bloque, 8 bloques/SM. Para 64 hilos/bloque serían 16, pero como el máximo de bloques por SM es 8, nos quedamos en 8. Cambiemos el supuesto: ahora vamos bien de hilos y registros, pero no de memoria compartida. Con 16KB por multiprocesador, si un solo bloque reserva los 16KB, sólo se podrá asignar un bloque por SM. Si por el contrario, la reserva es de 8KB, habrá 2 bloques/SM. Para 4KB, 4 bloques/SM, etc. La limitación de número de registros es similar. Por supuesto, el número *máximo* de bloques por multiprocesador viene restringido por la *mayor* de las restricciones.

En GT200, el número total máximo de bloques *asignados*, contando todos los multiprocesadores, es 8 max bloques/SM*30SMs = 240. Sin embargo, CUDA permite lanzar en la configuración del kernel muchos más que esos 240 bloques, ¡en concreto, 65535×65535!. A los bloques que en un momento determinado están asignados a un multiprocesador se les denomina *activos* o *residentes*. En otras palabras, aquellos de entre los que el *scheduler* puede elegir para cambiar la ejecución. En GT200 serían como mucho 240 al mismo tiempo, en un instante concreto. Según vayan terminando, el *scheduler* asigna otros pendientes de ejecución.

Un bloque no puede migrar entre multiprocesadores, está atado al que le fue asignado en un principio hasta que acabe. Esto es lógico, pues si el planificador suspende su ejecución, al volver necesitará recuperar exactamente el mismo estado en el que se encontraba anteriormente. Esa información estaría guardada en los registros, memoria compartida, etc, por lo que si fuese enviado a un nuevo SM no encontraría dichos datos, o habría que estar migrándolos, lo cual introduciría complejidad y retardos.

[10]Máximo de hilos por bloque.

4.3.2. Planificación de hilos. Tolerancia a la latencia.

En el paso anterior se han asignado bloques enteros a multiprocesadores. Ahora necesitamos saber cómo se planifican los hilos dentro del multiprocesador. No se pueden asignar individualmente. El *scheduler* los planifica en grupos llamados *warps*. Hasta ahora los *warps* han sido de 32 hilos, pero eso es una decisión de implementación hardware, y por tanto podría cambiar en el futuro. Los hilos del *warp* tienen identificadores (thread ID[11]) consecutivos, es decir, del 0 al 31 forman el primero, del 32 al 63 el segundo, etc. Al igual que con los bloques, aquí tambien existe el concepto de warps activos o residentes. Siguiendo con GT200, el número máximo de warps activos por SM es 32 (1024 hilos). Es sorprendente que se permita planificar entre tantos hilos cuando dentro del SM tan sólo hay 8 SPs (ver figura 4.7). Parece demasiado. Sin embargo, es una de las grandes bazas de CUDA. Cuando los hilos de un warp generan una operación de alta latencia, como un acceso a memoria global o aritmética en punto flotante, se deja el warp en suspensión, y se selecciona otro que haga algo útil mientras tanto. Esto permite lo que se conoce como *ocultación de latencia* (latency hiding). No se introduce ningún coste temporal adicional, es el *zero-overhead thread scheduling*, que podríamos traducir como «sobrecarga nula por planificación de hilos». Esto no ocurre así con los hilos de CPU, donde se requiere un cambio de contexto del sistema operativo, lo que implica gastar muchos ciclos en guardar el estado de la CPU (registros, program counter, etc) a memoria. Sin embargo, en los SMs, los registros y memoria se reparten entre todos los bloques asignados, es decir, coexisten[12]. No hace falta guardar ningún contexto, ya está ahí. Para cambiar de warp, simplemente se modifica una lista con los hilos activos y los pendientes de ejecución, y eso es prácticamente instantáneo. Resumiendo, se suele dcir que los hilos de CUDA son mucho más «livianos» que sus homólogos de CPU.

La figura 4.8 muestra la planificación de warps dentro de un multiprocesador. El tiempo avanza de izquierda a derecha. Al principio, se selecciona el warp 1 del bloque 1 para ejecución. En la instrucción 7 accede a memoria global, una operación de alta latencia, por lo que el planificador pone el warp en suspensión, y selecciona uno nuevo, el warp 1 del bloque 2[13]. La instrucción 3 del nuevo warp genera otra instrucción de alta latencia. El *scheduler* pone también a éste en suspensión. Mientras tanto, la instrucción 7 del warp 1 bloque 1 continúa progresando. Aparece marcado en la parte superior del cronograma como «TB1, W1 stall[14]». Hasta que acabe, otros warps siguen entrando, para ocupar los recursos de

[11] Thread ID será lo mismo que theadIdx.x si trabajamos en una sola dimensión, threadIdx.y*blockDim.x + threadIdx.x en dos dimensiones, y threadIdx.z*blockDim.x*blockDim.y + threadIdx.y*blockDim.x + threadIdx.x en tres dimensiones.

[12] Esto limita, como hemos comentado anteriormente, el número de bloques por SM según los recursos que éstos reserven.

[13] Nótese que se escogen warps de *cualquiera* de los bloques residentes o activos.

[14] *Stall* significa parada.

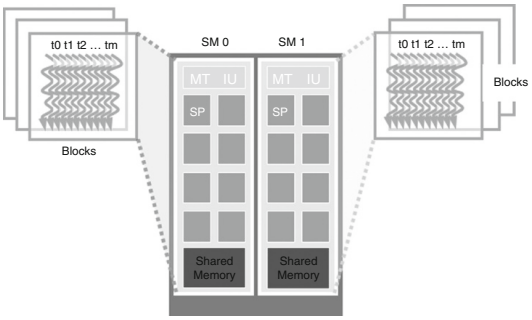


Figura 4.6: Los bloques se asignan a los multiprocesadores.

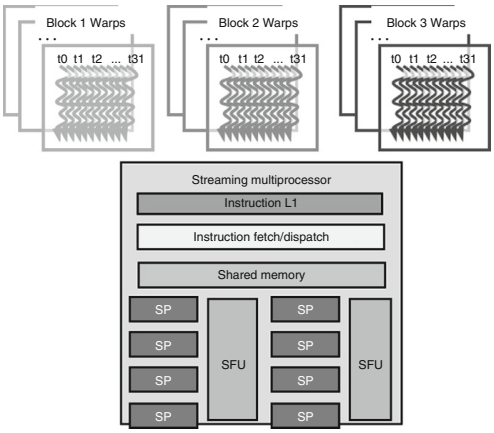


Figura 4.7: Dentro de cada SM, se planifica a nivel de warps.

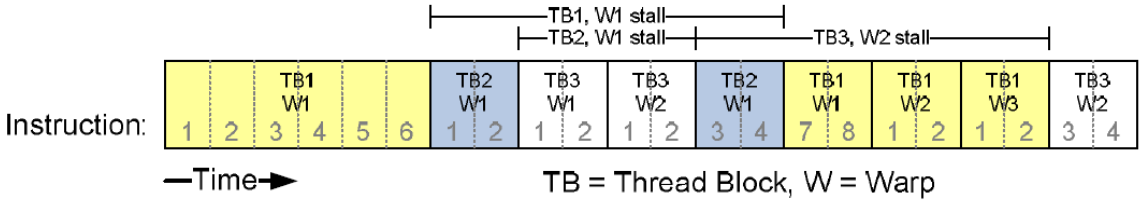


Figura 4.8: Ejemplo de planificación de warps

ejecución haciendo algo útil. Sin este sistema, habríamos perdido 8 unidades de tiempo.

Antes comentamos que CUDA era SPMD. Sin embargo, ésa no es toda la verdad. Cuando un warp se selecciona, todos los hilos que lo componen ejecutan la misma instrucción[15], es decir, SIMD, una sola instrucción sobre múltiples datos. En definitiva, podemos decir que CUDA es SPMD *entre warps* y SIMD *dentro del warp*, como se comenta en [17].

4.3.3. Divergencia de hilos

En caso de que aparezcan instrucciones condicionales if/while/switch, etc, se produce la llamada divergencia de hilos. En ese caso, se particiona el warp en varios caminos. Éstos se ejecutan en serie, aunque todos los hilos que pertenezcan al mismo siguen avanzando en paralelo. Esto se implementa mediante una máscara de bits. Por ejemplo, en una instrucción condicional IF/THEN/ELSE, se apila una máscara para que funcionen (desenmascarar) los hilos que han ido por el ELSE, y a continuación, prosiguen los que fueron por el THEN. Los SPs destinados a ejecutar los hilos del ELSE simplemente se desactivan (de ahí la pérdida de

[15] Salvo que se produzca divergencia de hilos. Ver subsección 4.3.3

eficiencia). Cuando en la ejecución llegamos a la instrucción ELSE, se desapila la máscara, y sucede lo contrario: los hilos del ELSE funcionan y los del THEN no. Si hay IFs anidados se siguen apilando máscaras a un nivel más profundo. El peor caso sería la divergencia absoluta (listado 4.3).

Listado 4.3: Divergencia total entre los hilos de un warp

```
switch(threadIdx.x){
  case 0: ...break;
  case 1: ...break;

  case 31: ...break;
}
```

En el listado 4.3, cada hilo del warp ejecutaría un código distinto, por lo que no hay en absoluto paralelismo.

4.4. Memoria

La mayor limitación de las aplicaciones en CUDA es el acceso a memoria global. Y ello a pesar de la ocultación de la latencia comentada en secciones anteriores. Existe un concepto importante, llamado *compute to global memory access (CGMA) ratio*, es decir, la relación entre el número de operaciones en punto flotante realizadas por cada acceso a memoria global. CUDA provee de otro tipo de memorias más rápidas, destinadas a conseguir aumentar el CGMA, es decir, a conseguir más operaciones con menos accesos a global. La idea es eliminar cargas redundantes entre hilos, y conseguir así un mayor reaprovechamiento.

En la figura 4.9 podemos observar los diversos tipos de memoria en CUDA. Abajo encontramos la global y la constante. Ambas son accesibles desde el host mediante llamadas al API. Los registros son de pequeño tamaño y se encuentran dentro del chip. Son de muy alta velocidad y se asignan a cada uno de los hilos. Un hilo no puede acceder a los registros de otro, es decir, son privados (ver tabla 4.1, columna visibilidad). Si en un kernel declaramos un solo registro, y dicho kernel tiene, por ejemplo, 1 millón de hilos, se crearán un millón de copias para que cada hilo use la suya propia. Su función es almacenar valores frecuentemente utilizados. No es necesario ningún cualificador especial para declararlos: al igual que ocurre con el código para CPU, son variables automáticas[16].

La memoria compartida (declarada mediante `__shared__`) es visible, o *compartida*, por todos los hilos de un mismo bloque. Todos los bloques del kernel tienen su propia

[16]Hay que tener cuidado con las variables automáticas de arrays. Para evitar un gasto excesivo de registros, se guardan en memoria global en lugar de en registros, aunque siguen siendo privadas al hilo. En la tabla 4.1 esto aparece indicado como «Local (Global Privada)»

copia privada de las variables compartidas. También es más rápida que la memoria global. Como ya se indicó en el primer párrafo, es muy común guardar partes de memoria global en compartida, para aprovechar localidad espacial/temporal de los programas. En mi código de crecimiento de regiones, la aprovecho para guardar los *labels* o etiquetas de los puntos[17], ya que es información a la que acceden varios hilos, y además a lo largo de varias iteraciones. Es ineficiente utilizar memoria compartida para datos que sólo empleemos una vez, ya que, al tiempo de acceso a memoria global, tenemos que sumar el de compartida, aunque sea leve. Sólo merece la pena cuando existe algún tipo de reutilización.

Las variables definidas como constantes, con el cualificador `__constant__`, son visibles para todo el grid. Se declaran fuera del cuerpo de cualquier función, normalmente al principio del fichero. La memoria constante es una *cache* de memoria global, de 65 KB. Al igual que su homóloga de CPU, el trasiego de datos se controla enteramente por hardware. Es de sólo lectura desde el dispositivo, es decir, sólo podemos escribir en ella desde el host, mediante llamadas al API. Realmente, la memoria compartida que hemos comentado en el párrafo anterior es también una *cache*, con la diferencia de que es el propio programador quien la gestiona, y sí es modificable desde el código de la GPU.

Las variables declaradas con el cualificador `__device__` son globales a todo el grid, y se guardan en memoria global. Una utilidad de ésta es la transferencia de grandes cantidades de datos entre host y dispositivo[18]. El otro propósito tiene mucho que ver con la columna «Tiempo de vida» de la tabla 4.1. Tanto las variables privadas de los hilos, como las compartidas dentro de un mismo bloque, duran lo que dure el kernel. Todo lo que queramos hacer con ellas tiene que hacerse *antes* de que acabe el kernel. Esto no es preocupante, porque normalmente los resultados de memoria compartida se almacenan en global, ya sea para devolver los datos al host, o para que otro kernel siga operando con ellos. Y es que en memoria global (también en constante), los datos se conservan a lo largo de toda la aplicación. Aunque un kernel acabe y otro empiece, la información seguirá estando ahí. Ésa es una característica importante, ya que pueden darse circunstancias que nos obliguen a dividir el código en varios kernels: una puede ser utilizar una configuración de ejecución[19] cambiante. Otra es la *sincronización implícita*, como veremos posteriormente, en el capítulo dedicado a la implementación de crecimiento de regiones en CUDA, en concreto en 5.5.2.

[17] Recordemos que los labels guardan información de la región a la que pertence un punto.

[18] Digo «grandes cantidades» porque si los parámetros que se pasan al kernel son de pequeño tamaño, se almacenan en memoria compartida. Por el contrario, si su tamaño crece demasiado, como en estructuras pasadas por valor, el compilador los guarda en memoria global.

[19] Recordemos, el número de hilos por bloque y el de bloques por grid.

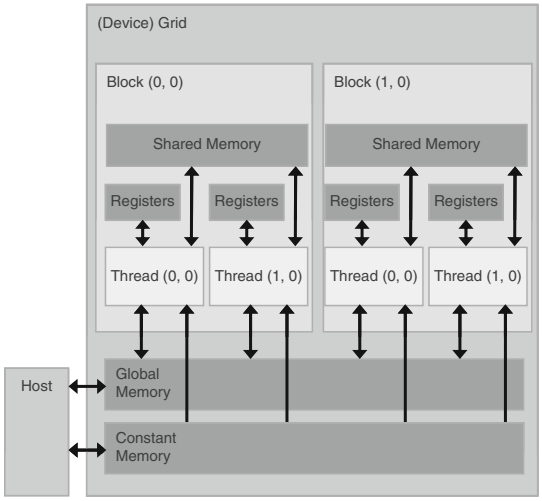


Figura 4.9: Modelo de memoria en CUDA.

Declaración de Variable	Memoria	Visibilidad	Tiempo de Vida
Variables automáticas que no sean arrays (escalares)	Registros	Hilos	Kernel
Variables automáticas de array	Local (Global privada)	Hilo	Kernel
`__shared__`	Compartida	Bloque	Kernel
`__device__`	Global	Grid	Aplicación
`__constant__`	Constante	Grid	Aplicación

Tabla 4.1: Cualificadores de tipo en CUDA.

4.4.1. Striding

Striding es un patrón de acceso a memoria en el que un hilo salta un número de posiciones igual al número total de hilos, o un múltiplo de éste. Es decir, si se tienen:

- N datos en memoria en un array A, indexado desde $A[0]$ hasta $A[N-1]$
- p hilos, desde T_0 hasta T_{p-1}
- $N > p$

En *striding*, un hilo T_i accede a la(s) posición(es) $A[i + sp]$, siendo:

$$0 \leq i \leq p - 1 \tag{4.1}$$

$$0 \leq s < \left\lceil \frac{N}{p} \right\rceil \tag{4.2}$$

$$i + s \cdot p < N \tag{4.3}$$

El techo (*ceiling*) de la fracción $\frac{N}{p}$, indicado como $\left\lceil \frac{N}{p} \right\rceil$ en (4.2), y la condición (4.3), son necesarios para poder recorrer todo el espacio del array A, desde A[0] hasta A[N-1], sin pasarnos de N cuando $\frac{N}{p}$ no es entero y $s = \left\lceil \frac{N}{p} \right\rceil$.

En CUDA, el número total de hilos p se puede calcular fácilmente. Dependiendo de si estamos aplicando striding en memoria global o compartida:

- En memoria compartida:
  - blockDim.x, si sólo trabajamos en una dimensión.
  - blockDim.x* blockDim.y, con dos dimensiones.
  - blockDim.x* blockDim.y*blockDim.z, con 3 dimensiones.
- En memoria global:
  - blockDim.x*gridDim.x, si sólo trabajamos con una dimensión.
  - blockDim.x*blockDim.y*gridDim.x*gridDim.y, con dos dimensiones.
  - blockDim.x*blockDim.y*blockDim.z*gridDim.x*gridDim.y, con tres dimensiones[20].

[20]No existen 3 dimensiones para el grid, pero sí para el bloque

Por supuesto, para el caso de memoria global existen otras posibilidades resultado de combinación de dimensiones: bloques de una dimensión con grid de dos dimensiones, grid de una dimensión con bloques de dos dimensiones, bloques de tres dimensiones con grid de una dimensión, etc, etc.

Para entender por qué necesitamos esta técnica, debemos fijarnos en la condición $N > p$, que quiere decir que el número de posiciones de memoria a acceder es mayor que el número de hilos. Por tanto, si se desea cubrirlas todas, un hilo tendrá que saltar a varias. Puede parecer extraño que en CUDA puedan llegar a faltar hilos, considerando la cantidad ingente que se puede lanzar, pero es que la cantidad de memoria global disponible también es enorme, con hasta 6 GB en modelos como Tesla C2070. Además, pese a ser posible, lanzar más hilos de los necesarios para la ocultación de la latencia es ineficiente, pese al buen trabajo del *scheduler* hardware.

Sin embargo, este problema es mucho más común con memoria compartida. En muchas GPUs, el número máximo de hilos por bloque es 512, mientras que la memoria compartida es de 16KB. Según de qué tipo sea el array guardado:

- De byte o char, tendremos 16*1024=16384 posiciones.
- De int, tendremos 16*1024/4=4096 posiciones.
- De int2[21], tendremos 16*1024/8=2048 posiciones.
- De int4, tendremos 16*1024/16=1024 posiciones.

Parece que se necesite el tipo de datos que se necesite, nunca podremos cubrir todo ese espacio de memoria con un acceso por hilo, al haber sólo 512 hilos (512 posiciones a hilo por posición). Harán falta varios accesos por hilo.

4.4.2. Coalescencia

La coalescencia (*coalescing*) constituye uno de los factores más importantes en el rendimiento de CUDA. Es la unión de varios accesos a memoria por parte de hilos pertenecientes a un *half-warp*,[22] o *warp*,[23] en una sóla transacción, en el mejor de los casos. Esto supone un ahorro considerable de tráfico en contraposición a la serialización de los accesos.

[21] CUDA tiene tipos del estilo tipo1,tipo2,tipo3,tipo4, que son estructuras con varios componentes. Por ejemplo, int4 tiene 4 enteros que se acceden con variable.x, variable.y, variable.z y variable.w

[22] En el caso de GPUs con capacidad computacional 1.x

[23] En el caso de GPUs con capacidad computacional 2.x

En CUDA aparece el concepto de segmentos de memoria. Un segmento de memoria es un grupo de bytes que está alineado con su tamaño (su dirección de comienzo es un múltiplo de éste). Por ejemplo, segmentos de 128 bytes: 0-127,128-255,256-383. Segmentos de 64: 0-63,128-191,192-255. Segmentos de 32: 0-31,32-63,64-95. La coalescencia se basa en la capacidad de traer de memoria un segmento entero en una sola transacción.

Se ha mencionado «en el mejor de los casos» porque deben cumplirse las siguientes condiciones:

- Para los datos:
  - El tamaño de los tipos, según la capacidad computacional:
    - 1.0 y 1.1: 4, 8 ó 16 bytes.
    - 1.2 y 1.3: 1, 2, 4, 8 ó 16 bytes.
  - Alineamiento, es decir, que la dirección de memoria donde está almacenada una variable sea múltiplo de su tamaño.
- Un patrón de acceso favorable.

Siempre que utilicemos los tipos predefinidos por CUDA, no hay que preocuparse por las dos primeras restricciones.[24]. En caso de utilizar tipos propios, como estructuras, se deben emplear directivas `__align__(número de bytes)`. Del «patrón de acceso favorable» se encarga el programador. Los requisitos para cumplir esta última condición difieren según la capacidad computacional:

Para 1.0 ó 1.1:

- Si el tamaño de palabra es:
  - 4 bytes: las 16 palabras accedidas por los 16 hilos tienen que estar en un mismo segmento de 16*4=64 bytes.
  - 8 bytes: todas las palabras en un segmento de 16*8=128 bytes.
  - 16 bytes: las primeras 8 palabras en un segmento de 128 bytes y el resto en otro segmento, también de 128 bytes.
- Hilos seguidos acceden a palabras seguidas: al hilo k-ésimo de un half-warp la palabra k-ésima del segmento. Se permite que haya hilos sin accesos.

[24]El alineamiento no sólo se cumple para variables estáticas, sino también para las reservas dinámicas mediante la función del API `cudaMalloc`, muy parecida al `malloc` de C. Siempre, se insiste, que se utilicen los tipos predefinidos.

Para 1.2 ó 1.3, las condiciones se relajan:

Al contrario que con 1.0/1.1, los hilos pueden acceder a cualquier palabra dentro de un mismo segmento. No es necesario que el hilo k-ésimo acceda a la posición k-ésima. Ante una falta de alineamiento, al acceder a distintos segmentos, se mantiene la coalescencia por cada uno de ellos. Es peor que transferir un sólo segmento, pero mejor que serializar. Esto último explicado formalmente sería:

1. Encontrar el segmento accedido por el hilo de menor *thread ID*[25] dentro del half-warp. El tamaño de dicho segmento será, según el tamaño de palabra accedida:
   - 32 bytes, para palabras de 1 byte.
   - 64 bytes, para palabras de 2 bytes.
   - 128 bytes, para palabras de 4, 8 y 16 bytes.
2. Reducir el tamaño de la transacción, mirando las posiciones accedidas por los demás hilos del half-warp:
   a) Si el segmento es de 128 bytes y sólo se utiliza la parte superior o la inferior, reducir la transacción a 64 bytes.
   b) Si la transacción es de 64 bytes (por ser originariamente un segmento de ese tamaño o bien después de la reducción del paso anterior) y sólo se usa la parte superior o inferior, reducir a 32 bytes.
3. Marcar los hilos atendidos.
4. Repetir todos los pasos anteriores hasta que todos los hilos restantes del half-warp hayan sido antendidos. Esto es necesario por si otros hilos de dicho half-warp acceden a otro segmento distinto.

Veamos un ejemplo (figura 4.10). Para capacidad computacional que no sea 2.x, tenemos que fijarnos en cada half-warp, 16 hilos, del 0 al 15, y otros 16 del 16 al 31.

- «Aligned and sequential»
  - 1.0 y 1.1: Hilo k-ésimo accede a posición k-ésima, por lo que no hay problema: coalescencia para cada half-warp.

[25] Thread ID será lo mismo que theadIdx.x si trabajamos en una sola dimensión, threadIdx.y*blockDim.x + threadIdx.x en dos dimensiones, y threadIdx.z*blockDim.x*blockDim.y + threadIdx.y*blockDim.x + threadIdx.x en tres dimensiones.

    - 1.2 y 1.3: Según el protocolo que acabamos de ver, palabras de 4 bytes corresponden a segmentos de 128 bytes, pero el tamaño de la transacción se reduce, ya que todos los hilos de cada half-warp utilizan solamente los 64 bytes inferiores (128-191, ambos inclusive).

- «Aligned and non-sequential»

    - 1.0 y 1.1: Hay dos hilos que acceden a posiciones desordenadas, por lo que se invalida la regla del hilo k-ésimo, posición k-ésima. Todos los accesos son serializados.
    - 1.2 y 1.3: No hay ningún cambio respecto a «Aligned and sequential». No importa que los hilos accedan desordenadamente.

- «Misaligned and sequential»

    - 1.0 y 1.1: Por la misma razón que el caso anterior, todos los accesos se serializan.
    - 1.2 y 1.3: Nuevamente, siguiendo el protocolo, palabras de 4 bytes corresponden a segmentos de 128 bytes. El hilo de thread ID más bajo, el 0, accede a la posición 132 (128+4bytes/palabra). Los demás están dentro del rango 128-191, menos el último, que ya lee en la posición 192, perteneciente a la siguiente mitad del segmento de 128 bytes. Por tanto, no podemos reducir a 64, se transmiten los 128. Para el siguiente half-warp, los hilos 16 al 30 están en el semisegmento 192-255, y el 31º ya accede al siguiente. En un principio se podría pensar que simplemente, al igual que antes, se genera una transacción de 128. Sin embargo, eso no puede ser, ya que éstas sólo ocurren en direcciones alineadas a 128 bytes, es decir: 0-127, 128-255, 256-383 ... Por tanto, se produce una de 64 y, para el último hilo, ahora sí, otra que inicialmente es de 128 pero se reduce a 64, y finalmente a 32. Resumiendo, para el primer half-warp, una de 128, para el segundo half-warp, una de 64 y otra de 32.

Dos dimensiones: padding

Todo esto está muy bien, pero si queremos utilizar un array de dos dimensiones, como las imágenes con las que se trabaja aquí, no basta con una llamada a cudaMalloc, ya que ésta solo alinea en una dimensión, no sabe nada del ancho y el alto.

Para leer píxeles de la imagen se utiliza el siguiente código, siendo *Row* la fila, y *Col* la columna, a la que se accede (se comentó brevemente esta técnica de aplanamiento o *flattening*, por la que convertimos dos dimensiones lógicas a una dimensión real, en el ejemplo de multiplicación de matrices visto en 4.2).

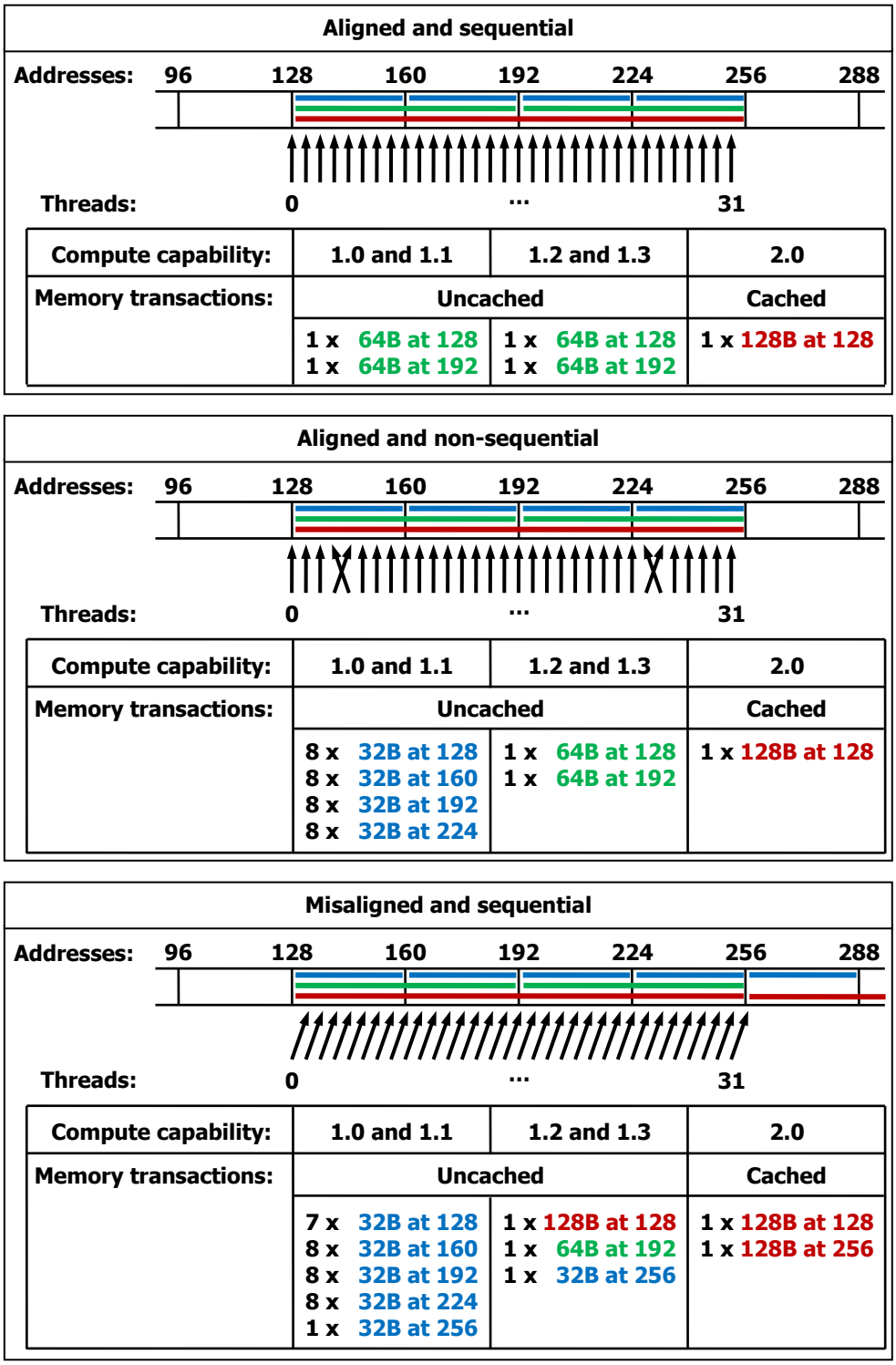


Figura 4.10: Un ejemplo de acceso a memoria global por un *warp*. El tamaño de palabra accedido por los hilos es de 4 bytes.

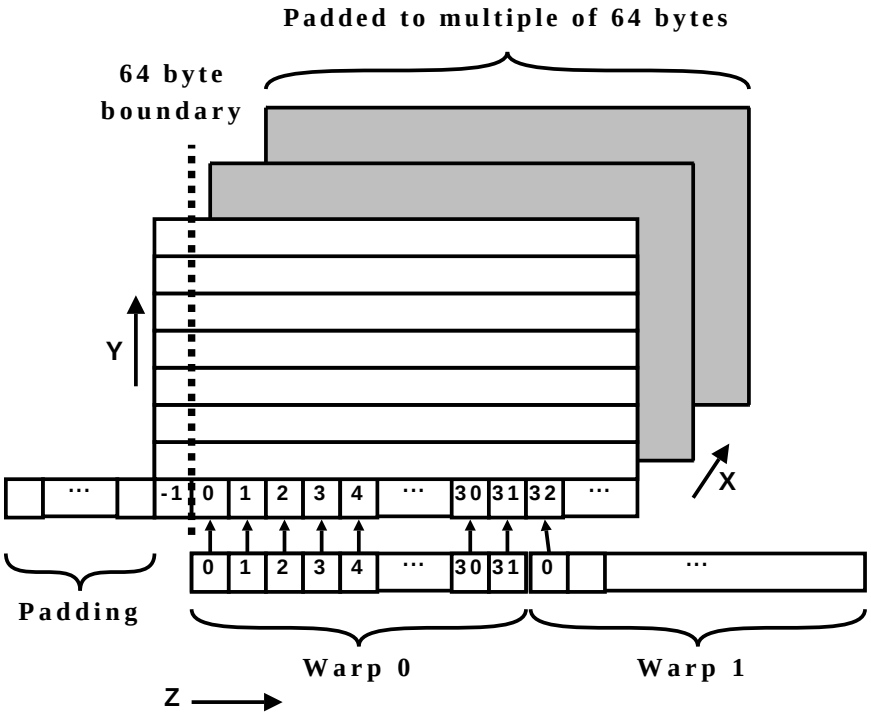


Figura 4.11: Un ejemplo de coalescencia para 3 dimensiones, pero que nos sirve para entender 2 dimensiones también.

Listado 4.4: Acceso coalescente a los pixels usando el pitch adecuado

```
mapaBitsDevice[Row*devicePitch+Col]
```

Para que estos accesos sean coalescentes, tanto el ancho de la imagen como el del bloque deben ser múltiplos del half-warp para GPUs de capacidad 1.x, y del warp en el caso de 2.x. Sin embargo, restringir tanto el tamaño de la imagen sería limitar demasiado el programa. En caso de que el ancho no cumpla esa condición, se puede aumentar hasta que lo haga. Ese aumento extra es lo que se conoce como *padding*. El ancho total, suma del original más el padding, se denomina *pitch*. La función `cudaMallocPitch` se encarga, dado el ancho y el alto, de reservar memoria utilizando este sistema, devolviendo, además, el pitch adecuado. Éste es necesario dentro del kernel para obtener la indirección correcta, tal y como podemos observar en el listado 4.4 de arriba.

En la figura 4.11, extraída de [6], se muestra una estructura en tres dimensiones alineada. También sirve para mostrar nuestro caso, en 2D, fijándonos sólo en los ejes z-y.

4.5. *Sincronización*

CUDA permite a los hilos de un mismo bloque coordinar su trabajo mediante una función barrera de sincronización, `__syncthreads()`. El hilo que la ejecute suspenderá

su actividad en el lugar de la llamada, hasta que todos los hilos del bloque lleguen a dicho punto. De esta forma, se puede asegurar que todos hayan completado una fase antes de que ninguno pase a la siguiente. Esto permite que las escrituras a memoria compartida y global realizadas por un hilo sean visibles por el resto de hilos pertenecientes al mismo bloque.

Aunque a nivel lógico, al programar, se piense en los hilos como si su ejecución ocurriera en paralelo, a nivel de hardware no tiene por qué ocurrir así. Como vimos en la subsección 4.3.2, no todos los hilos empiezan o terminan a la par. De ahí la necesidad de una barrera de sincronización.

4.5.1. Tropezando con la misma piedra otra vez

`Syncthreads` es una instrucción peligrosa. No se debe incluir dentro de una sentencia condicional como `if/while` cuando ésta produzca una divergencia de hilos[26]. La razón es que sólo los que cumplan la condición ejecutarán syncthreads, los demás terminarán sin pasar por él. Como la sincronización espera a todos, se produce una situación de bloqueo. En la práctica esto puede producir uno de estos dos efectos indeseables, dependiendo de la versión del compilador, del driver y de la fase lunar:

- Se produce un «cuelgue», y aunque el sistema operativo sigue funcionando, no se puede matar el proceso. Directamente es necesario un *hard reset*.
- Resultados incorrectos. Trabajando con imágenes, como es el caso de este trabajo, hay zonas de la imagen que parecen no ser afectadas por el algoritmo.

La restricción mencionada en esta sección desaparece cuando no se produce divergencia de hilos, o, dicho de otro modo, cuando la condición no depende, ni directa ni indirectamente, de `threadIdx` o `blockIdx`. De forma más sencilla, cuando o bien todos los hilos cumplen, o bien todos inclumplen, la condición. Aunque estas reglas son sencillas, es fácil olvidarlas durante la programación, tropezando de nuevo con la misma piedra.

Queda claro que podemos sincronizar dentro de un bloque, pero ¿y entre bloques?. Desafortunadamente, CUDA no proporciona ningún método de sincronización global (ver figura 4.12). Esto es así por diseño. Se permite una escabilidad amplia, con miles de bloques y millones de hilos, pero es imposible que el hardware pueda sincronizar todo eso, ya que por cada SM hay como mucho 8 bloques de hilos (1536 hilos) activos o residentes. Todas estas cuestiones se discuten en profundidad en [10], [5] y [26]. Sin embargo, podemos usar *sincronización global implícita*, consistente en dejar que el kernel termine y empezar otro[27].

[26] Unos ejecutarán unas instrucciones y otros otras

[27] En la guía de CUDA también aparece otro método consistente en hacer que el último bloque de hilos haga algo en especial; véase «Memory Fence Functions» en [9]

Figura 4.12: ¡No hay sincronización global!

Es ineficiente, pero no hay otro remedio. Esto lo veremos en el capítulo de implementación, subsección 5.5.2.

4.6. Operaciones atómicas

En los programas secuenciales existen operaciones que no debemos utilizar en código paralelo. Veamos un ejemplo con la suma:

```
x++;
```

La sentencia anterior aumenta en uno el valor de x, pero realmente consta de tres pasos:

1. Leer el valor de x.
2. Sumar uno al valor leído en el primer paso.
3. Escribir el resultado en x.

Este proceso se cataloga dentro de los conocidos como *lectura-modificación-escritura*.

Consideremos ahora dos hilos «A» y «B» que realizan sendos incrementos sobre *la misma variable* `x`. Ambos tienen que seguir los pasos descritos anteriormente. Supongamos que el valor original de x es 7. Dado que es incrementado por dos hilos, el valor esperado de x sería 9, tal y como se muestra en la tabla 4.2. Sin embargo, cada hilo va a su propio ritmo, por lo que existen otras múltiples ordenaciones que producen resultados incorrectos (tabla 4.3).

Lo que necesitamos para solventar el problema del anterior ejemplo es una manera de realizar *lectura-modificación-escritura* sin que ningún otro hilo nos interrumpa, es decir, sin que ningún otro hilo lea o escriba en `x` hasta que hayamos terminado con la operación. Las operaciones que cumplen esa condición se denominan *atómicas*.

Paso	Valor de x
1. El hilo A lee el valor de x	A lee 7 de x
2. El hilo A incrementa en uno el valor leído	A calcula 7+1=8
3. El hilo A escribe el resultado en x.	$x \leftarrow 8$
4. El hilo B lee el valor de x	B lee 8 de x
5. El hilo B incrementa en uno el valor leído	B calcula 8+1=9
6. El hilo B escribe el resultado en x	$x \leftarrow 9$

Tabla 4.2: Valor esperado de x con dos hilos incrementando su valor.

Paso	Valor de x
1. El hilo A lee el valor de x	A lee 7 de x
2. El hilo B lee el valor de x	B lee 7 de x
3. El hilo A incrementa en uno el valor leído	A calula 7+1=8
4. El hilo B incrementa en uno el valor leído	B calcula 7+1=8
5. El hilo A escribe el resultado en x	$x \leftarrow 8$
6. El hilo B escribe el resultado en x	$x \leftarrow 8$

Tabla 4.3: Dos hilos incrementan el valor de x con operaciones intermedias intercaladas.

4.7. Comunicación

Existen otras librerías/lenguajes de programación paralelos aparte de CUDA. Por ejemplo, MPI [15], para computación con clusters, y OpenMP [15], en sistemas multi-procesador de memoria compartida. Precisamente a través de memoria compartida es como OpenMP realiza la comunicación entre hilos. MPI, por el contrario, utiliza el concepto de paso de mensajes. De hecho, sus siglas así lo indican: (*Message Passing* Interface). Un hilo deja un mensaje en un «buzón», y otro lo recoge.

Paso de mensajes exige más esfuerzo al programador, aunque en la práctica escala mejor. MPI se ha llegado a implementar en sistemas de hasta 100.000 nodos. OpenMP es más sencillo, pero tiene problemas a partir de los 200, debido a tiempos excesivos en planificación

de hilos y coherencia de *caches*[28]. En muchas ocasiones, el bus de memoria del sistema es compartido entre los procesadores, y el ancho de banda se convierte en el cuello de botella.

CUDA, al igual que OpenMP, utiliza memoria compartida. Sin embargo, existen grandes diferencias entre ambas: la de OpenMP es la principal del sistema, la RAM[29], lenta y de gran tamaño. Todos los procesadores la comparten. En CUDA, es muy reducida, de apenas 16 ó 48KB, rápida, y sólo la comparten los *Stream Processors* pertenecientes a un mismo *Stream Multiprocessor*, normalmente 8 ó 16. Al ser privada a ese grupo de SPs, los creadores de CUDA eliminan ¡fácilmente! el problema de coherencia de memoria compartida, y lo hacen así porque quieren, **por diseño**, porque esa arquitectura funciona muy bien para gráficos 3D. Muerto el perro, se acabó la rabia. Esto supone una mayor escalabilidad, pero *limita* los problemas que podemos tratar en CUDA: sólo aquellos que presenten un *alto* nivel de paralelismo, o, lo que es lo mismo, de independencia de datos.

En CUDA realmente podemos comunicar los bloques mediante memoria global. Pero no podemos sincronizar globalmente[30], entre todos los bloques de hilos, así que de nada nos sirve. La memoria global está ahí más que nada para dar datos de entrada y almacenar los de salida. También es posible usar operaciones atómicas con ella, a pesar de su lentitud.

[28] Cuando un procesador escribe en su *cache*, se debe informar a las demás *caches* restantes que tengan copia.

[29] Realmente una parte de ella.

[30] Aunque sí implícitamente, como comentamos en la sección anterior

Capítulo 5

Segmentación por crecimiento de regiones en CUDA

Como se ha visto anteriormente, el algoritmo de segmentación por crecimiento de regiones es relativamente sencillo. Sin embargo, la elaboración y diseño de una versión paralela no lo es tanto. Hay que tomar una serie de decisiones sobre particionamiento de la imagen, sincronización y comunicación de hilos, mezcla de regiones..., que se verán con detalle en este capítulo.

5.1. División del trabajo

Antes de empezar con crecimiento de regiones, es necesario pensar en cómo dividir la imagen en bloques. Básicamente, podemos elegir entre tener pocos bloques con muchos hilos, o muchos bloques con menos hilos. A priori, siendo CUDA programación paralela masiva, podría pensarse que la segunda opción es más favorable[1]. Sin embargo, otros factores puede inclinar la balanza del lado contrario:

- Cada multiprocesador tiene una cantidad limitada de registros y memoria compartida disponible. Hay que repartir estos recursos entre todos los bloques activos. Si éstos declaran variables de gran tamaño o reservan demasiada memoria compartida, la GPU limitará su número para dar a todos lo que piden, reduciéndose por tanto la ocupación.

- Los bloques grandes permiten sincronizaciones más amplias. Si se necesitan muchas, esto puede ser importante.

- Si todos los *warps* piden los mismos datos, podemos guardarlos en memoria compartida[2]. En un bloque grande, muchos hilos podrán acceder a ésta, ahorrándonos accesos

[1] Pero con unos mínimos. Hay un límite en el número de bloques activos por multiprocesador. Actualmente es de 8 para todos los procesadores gráficos. Para lograr una ocupación máxima, los bloques deberían ser mayores de 3 *warps*, para capacidad computacional 1.0 y 1.1, 8 *warps*, para 1.2 y 1.3, y 6 *warps*, para 2.1 y 2.2.

[2] CUDA garantiza que un mismo bloque será asignado a un multiprocesador en concreto. Por tanto, no hay peligro de ir accediendo a memorias compartidas de diferentes bloques

a memoria global. Con bloques demasiado pequeños, cada uno tendrá que cargar los mismos datos por separado, reduciéndose el redimiento.

Veamos los resultados de cada procedimiento. Como ahora tan sólo estamos probando patrones de acceso a píxeles, he utilizado la modificación del brillo, mucho más sencilla que implementar directamente crecimiento de regiones. Posteriormente aplicaremos sobre éste el patrón de acceso que aquí elaboremos.

Figura 5.1: Brillo del 40 % para un tamaño de bloque de 22x22=484 pixels (1 píxel por hilo)

Figura 5.2: Brillo del 40 % para un tamaño de bloque de 22x22 aplicando streading (varios píxeles por hilo)

En las figuras 5.1 y 5.2 podemos comprobar el resultado de aplicar un 40 % de brillo a la imagen. A la izquierda, un hilo se ocupa de un solo píxel; a la derecha, de varios. En ambos casos, se utilizan 22x22 hilos por bloque. Esta elección es debida al límite `maxThreadsPerBlock`, de 512 ($22 \times 22 = 488 < 512 < 23 \times 23$).

Veamos cómo se procede con varios píxeles por hilo. En la figura 5.2, se puede observar la división de la zona con brillo en *tiles* de 22x22, individualmente del mismo tamaño que en 5.1. Siguiendo la imagen 5.3 y el código del listado 5.2 se puede estudiar cómo se recorre horizontalmente la imagen, saltando de *tile* en *tile* mediante `Columm+=blockDim.x`. Cuando se termina una fila, nos desplazamos verticalmente un *tile* usando `Row+=blockDim.y`. Reposicionamos la columna con `Columm = 26*blockIdx.x + threadIdx.x`. El ancho del área con brillo es `TAMSHAREDLADO=126`. Se ha tomado esta cifra por ser 126*126 el máximo de memoria compartida reservable.

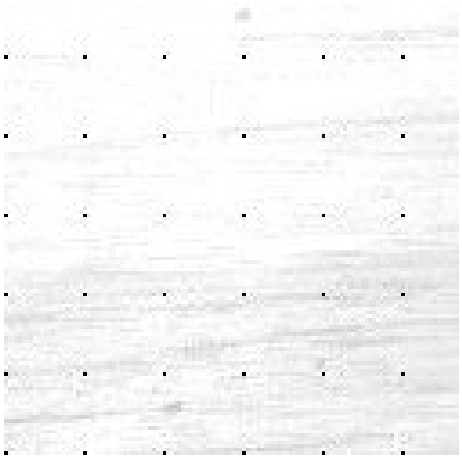


Figura 5.3: Detalle de la zona con brillo de la figura 5.2. En negro, se muestran los píxeles modificados por el hilo(0,0)

Algunas notas sobre la terminología usada:

- En este trabajo punto y píxel son sinónimos.
- Cuando hablo de «bloques» me refiero a bloques *de hilos*, no bloques *de píxeles*.
- Los tiles son zonas de la imagen sobre las que opera un bloque de hilos en concreto. Con «opera» quiero decir tanto escribir como leer: aunque hayamos establecido una relación 1:1 entre píxeles e hilos, los puntos de los bordes no tienen hilos, al no tener vecinos. Esos píxeles perimetrales no se escriben, pero sí se leen. En el código, tenemos tiles de 22x22, pero bloques de hilos de 20x20.

Listado 5.1: Bloque pequeño (1 hilo por píxel)

```
__global__ void fnIMG_BrilloOnDevice(int brillo)
{

  __shared__ BYTE mapaBits[22][22];
  long long tid;  int tx=threadIdx.x; int ty=threadIdx.y;

  int Row = blockIdx.y*blockDim.y + threadIdx.y;
  int Column = blockIdx.x*blockDim.x + threadIdx.x;

  tid = Row*maxRow + Column;
  ...Aplicamos subida brillo...
  __syncthreads();
  datImDG.m_pchMapaBits[tid] = mapaBits[tx][ty];

}
```

Listado 5.2: Bloque pequeño (varios hilos por píxel)

```
__global__ void fnIMG_BrilloOnDevice(int brillo)
{

  __shared__ BYTE mapaBits[TAMSHAREDLADO][TAMSHAREDLADO];
  long long tid = 0;  int cx; int cy;

  int Row = 126*blockIdx.y + threadIdx.y;
  int Column = 126*blockIdx.x + threadIdx.x;

  for (cy=threadIdx.y;cy<126;cy+=blockDim.y){
    for (cx=threadIdx.x;cx<126;cx+=blockDim.x){
      tid = Row*maxCol + Column;
      ...Aplicamos subida brillo...
      __syncthreads();
      datImDG.m_pchMapaBits[tid] = mapaBits[cx][cy];
      Column+=blockDim.x;
    }
    Column = 126*blockIdx.x + threadIdx.x;
    Row+=blockDim.y;
  }

}
```

En la figura 5.4 se muestra el rendimiento de estos dos procedimientos para las GPUs GeForce 210 y GeForce GTX 460. Como se puede observar, no ocurre nada especialmente remarcable. En ambos dispositivos se obtienen tiempos similares. Por ser más sencilla la de un hilo por píxel, es la que se ha usado en la implementación de crecimiento de regiones con CUDA.

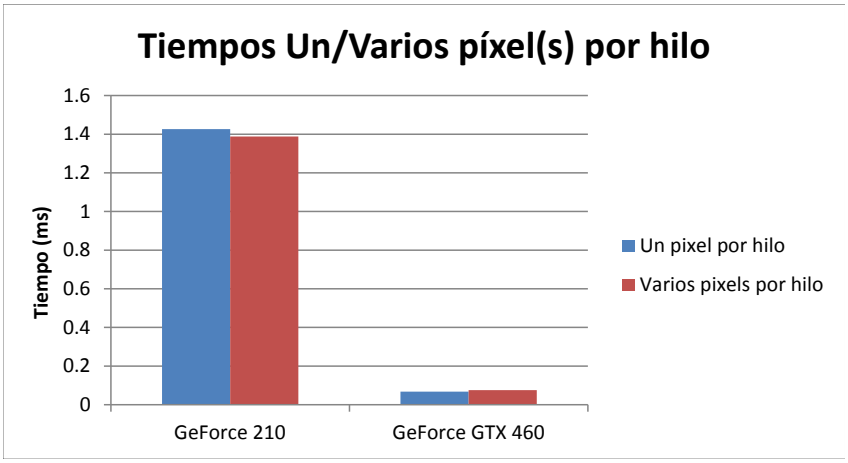


Figura 5.4: Tiempos para el procedimiento de un píxel por hilo, y varios píxeles por hilo, en dos GPUs diferentes.

5.2. Expansión desde la semilla

Si recordamos la subsubsección 3.3.4, se comentó al final la posibilidad de varios criterios de similitud para decidir la inclusión de un píxel en una región. Algunos sólo tenían en cuenta el valor de la semilla, y otros toda la región, como la media, y se argumentó su elección por ser el equilibrio ideal. En la sección 3.3, se mencionó como segundo paso de crecimiento de regiones «Comprobar los puntos vecinos y añadirlos a la región si son similares a la semilla.». Nótese que la sentencia está en plural. Si entendemos esa pluralidad como paralelismo, la frase anterior es correcta o no según el criterio elegido: en caso de tomar el valor inicial de la semilla como referente, sí lo es, porque la condición es estática. No ocurre lo mismo con la media de la región, dado que cambia *con cada píxel* añadido a ella. Dicho de otra forma, es dependiente del orden en el que se comprueban los píxeles vecinos. Por tanto, si queremos una implementación totalmente correcta, deberíamos añadir el píxel con valor *más similar de todos los vecinos*, o, en otras palabras, aquel que tenga la diferencia *mínima*. En [22] esto se implementa como una lista de píxeles vecinos ordenada, en la que los puntos de menor distancia se sitúan los primeros. Sin embargo, ese diseño es secuencial. En mi programa, al igual que en otros aprovechamientos paralelos de crecimiento de regiones [14], y en aras de una mayor velocidad, he optado por procesar todos los vecinos a la vez, aun a sabiendas de que no es la forma de proceder exacta.

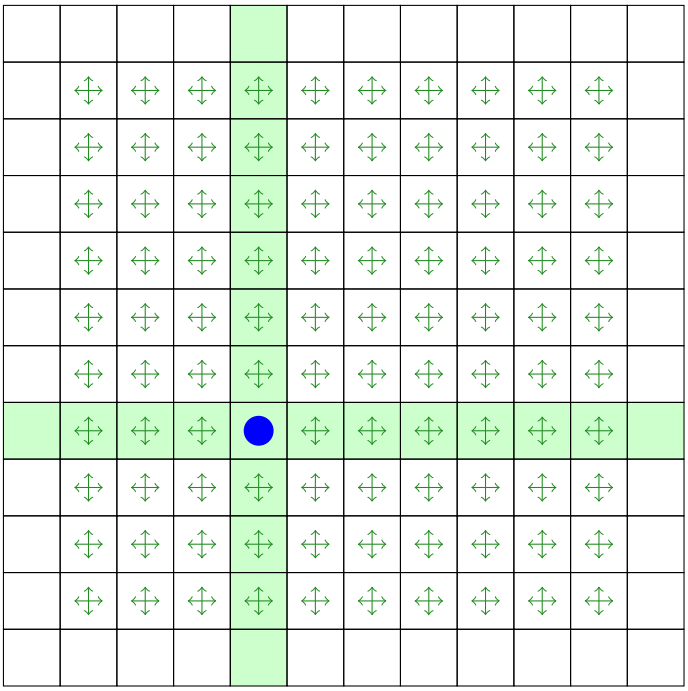


Figura 5.5: Implementación paralela del algoritmo de Crecimiento de Regiones.

Una implementación paralela podría basarse en el esquema mostrado en la figura 5.5.

La cuadrícula representa a un bloque, y cada una de las celdas a un píxel/hilo[3]. En el punto azul, intersección de la fila y columna verdes, se encuentra la semilla. La dirección de las flechas indica en qué dirección «mira» cada hilo para ver si el vecino ya ha sido incluido en la región. En el código, esto se comprueba mediante un array de *labels* o etiquetas, del mismo tamaño que la imagen, en el que 0 significa no pertenencia, y 1 pertenencia, a la región.

No hay hilos para los puntos de los bordes porque éstos sólo tienen 2 ó 3 vecinos, pero nunca los 4 que hemos considerado en crecimiento de regiones. En el código habría que hacer distinciones según threadIdx a la hora de leer las etiquetas de los vecinos, ya que si alguno de los 4 no existe, se producirá una violación de acceso a memoria, con la consiguiente parada del kernel. El problema de este diseño es que se generan espacios sin rellenar entre los *tiles*, pero esto se solucionará posteriormente, en la sección 5.3.

En la figura 5.6 se puede observar el resultado de aplicar la implementación. En la 5.7 muestra un detalle de ésta. Se ha señalado con un cuadrado rojo los píxeles que cubre un bloque. Las zonas en blanco son las regiones que se han expandido a partir de la semilla (en azul). Para el bloque marcado, podemos observar que la región crece desde la mesa, y se detiene al llegar al plato. En este caso se obtienen buenos resultados debido al alto contraste entre la madera, oscura, y la porcelana, blanca. Sin embargo, todavía estamos lejos de obtener un resultado como el de las imágenes de las figuras 3.3 y 3.4: hay espacios entre las regiones y, lo más importante, están todavía separadas. Las siguientes secciones solucionarán estos inconvenientes.

5.2.1. Un intento incorrecto de mejorar el algoritmo

Las figuras 3.9 y 3.10 podrían llevarnos a pensar que las regiones *siempre* se expanden desde la semilla con dirección a los vértices, como se muestra en el esquema 5.8. Con este supuesto, se llega al diagrama 5.9. Es igual que el esquema de la figura 5.5, pero ahora hay 4 cuadrantes con origen en la semilla, que determinan el sentido de las flechas. Con esto se ganaría en eficiencia, ya que pasamos de acceder de cuatro puntos vecinos a tan sólo dos. Sin embargo, la figura 5.10 muestra que este planteamiento no es correcto para algunos casos:

[3]Estamos utilizando un píxel por hilo, como se adelantó al final de la sección 5.1.

Figura 5.6: Imagen obtenida tras aplicar crecimiento de regiones con una semilla por cada bloque.

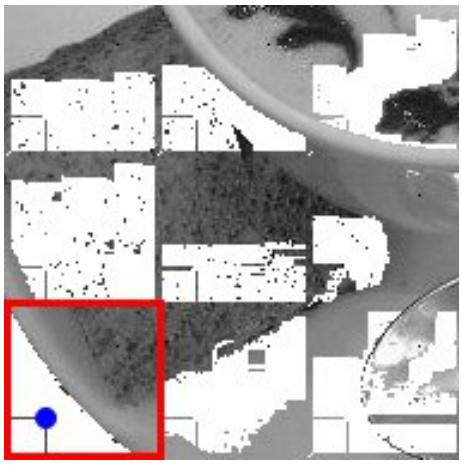


Figura 5.7: Detalle de la imagen 5.6. La semilla (en azul) del bloque marcado en rojo está situada sobre la mesa. Se expande hasta llegar al plato.

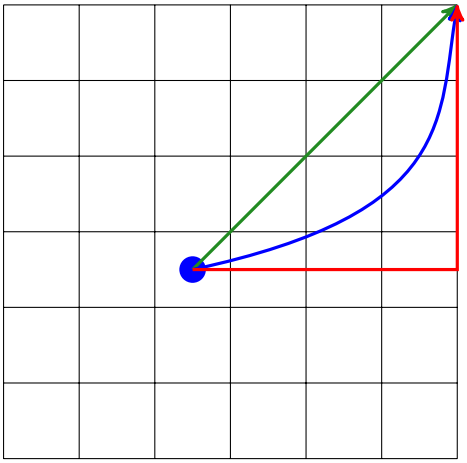


Figura 5.8: Tres posibles direcciones de propagación, de la semilla a los vértices. Este supuesto es incorrecto, *no siempre* las regiones avanzan en ese sentido.

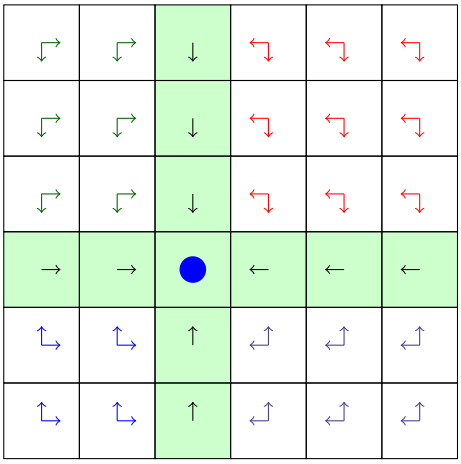
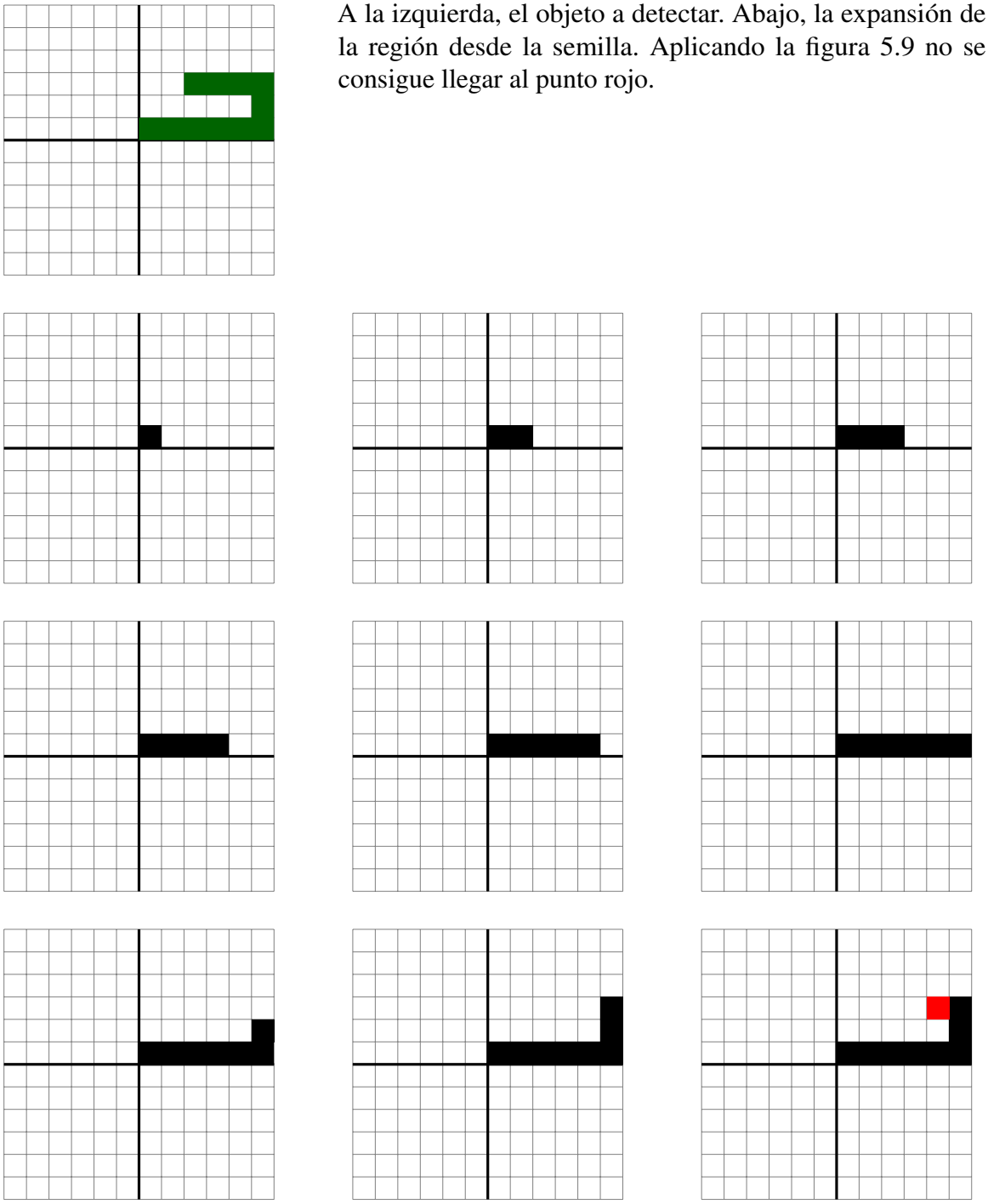


Figura 5.9: Implementación paralela según el supuesto incorrecto de la figura 5.8.



A la izquierda, el objeto a detectar. Abajo, la expansión de la región desde la semilla. Aplicando la figura 5.9 no se consigue llegar al punto rojo.

Figura 5.10: Este ejemplo demuestra que el planteamiento de las figuras 5.8 y 5.9 es incorrecto. Se debería obtener la región en verde (arriba), pero paso a paso comprobamos que ni siquiera se llega al punto rojo. Según el esquema 5.9, dicho punto sólo comprueba el vecino inferior e izquierdo, pero en este caso la región está a la derecha, por lo que falla.

5.2.2. Sincronización

Si recordamos la sección 4.5, la sincronización es necesaria para coordinar el flujo de ejecución, y permite que escrituras a memoria compartida y global realizadas por un hilo sean visibles por el resto de hilos pertenecientes al mismo bloque. Veamos un ejemplo en el código: en él utilizo un array de etiquetas o *labels*, inicializados a indexBloque para el punto semilla, y a 0x00 para el resto. Posteriormente, aparece un `__syncthreads()` (ver listado 5.3). Éste es necesario porque, de no utilizarse, un hilo «rápido»[4] que ya esté en el código posterior podría leer los valores previos a la inicialización (que son *basura*), ya que todavía el «lento» no ha tenido tiempo de escribir.

Listado 5.3: La inicialización de los labels es visible por los hilos del bloque al ejecutar syncthreads

```
if (threadIdx.x==semilla.x && threadIdx.y==semilla.y){
      Inicializar a indexBloque
}
else{
    for (i=0;threadIdx.x + i<DIMTILE && threadIdx.y + i<DIMTILE;i+=
       DIMBLOQUE){
        Inicializar a 0x00
    }
}
__syncthreads();
```

En la sección 4.5.1 mencionamos que no debemos incluir barreras de sincronización en sentencias condicionales, bajo peligro de «cuelgues» del sistema o resultados incorrectos. Esta circunstancia obliga, en ocasiones, a elaborar código reduntante:

Como los bloques no encajan exactamente en el tamaño de la imagen, si queremos que se cubra toda, habrá bloques que «sobresalgan». El código presentado a continuación evita que esos salientes produzcan violaciones de acceso a memoria. Sin embargo, en la versión actual del programa he desechado este método, porque añade demasiada lógica (instrucciones if), contrario a la filosofía SIMD. Por ello en la figura 5.14 los bordes aparecen sin modificar.

Por ejemplo, en lugar de este código (que es incorrecto):

Listado 5.4: No se debe utilizar `syncthreads` dentro de `if`

```
if (Col<ancho && Row<alto){
    colorPuntoActual = datImDG.m_pchMapaBits[Row*ancho_padding+Col];
```

[4] Recordemos que, tal y como se vio en 4.3.2, no todos los hilos empiezan o terminan a la par.

```
    __syncthreads();    //Incorrecto. En general, no se debe utilizar
        syncthreads dentro cuando se produzca divergencia de hilos
    //Código Crecimiento Regiones donde se utiliza colorPuntoActual en
        algún momento.
}
```

se debe usar este otro, duplicando la sentencia condicional:

Listado 5.5: Utilización correcta de `syncthreads`

```
if (Col<ancho && Row<alto)
    colorPuntoActual = datImDG.m_pchMapaBits[Row*ancho_padding+Col];

__syncthreads();

if (Col<ancho && Row<alto){
    //Código Crecimiento Regiones donde se utiliza colorPuntoActual en
        algún momento.
}
```

Nótese que la sincronización es necesaria porque tenemos que asegurar que el «Código de Crecimiento de Regiones» utiliza el valor actualizado de `colorPuntoActual`, que es: `datImDG.m_pchMapaBits[Row*ancho_padding+Col]`.

Dado que no debemos añadir `syncthreads` dentro de `if`, tenemos que extraerlo afuera, con el consiguiente engorro de volver a repetir `if (Col<ancho && Row<alto)`.

5.2.3. Cálculo de la media

Mediante operaciones atómicas

En la sección 4.6 se introdujo el concepto de operaciones atómicas. En el código de crecimiento de regiones, las utilicé en un principio para el cálculo de la media de intensidad de la región. Debido al pobre rendimiento obtenido, en la versión final opté por la técnica de reducción paralela (5.2.3), mucho más rápida. Aun así, veamos cómo se haría con atómicas:

Necesitamos dos variables: la suma total del valor de todos los píxeles de la región (`numerador_media`), y su número (`elementos_media`). Cada una se incrementa atómicamente (ver listado 5.6).

Listado 5.6: Variables incrementadas atómicamente

```
atomicAdd(&elementosMedia,1);
atomicAdd(&numeradorMedia, colorPuntoActual);
```

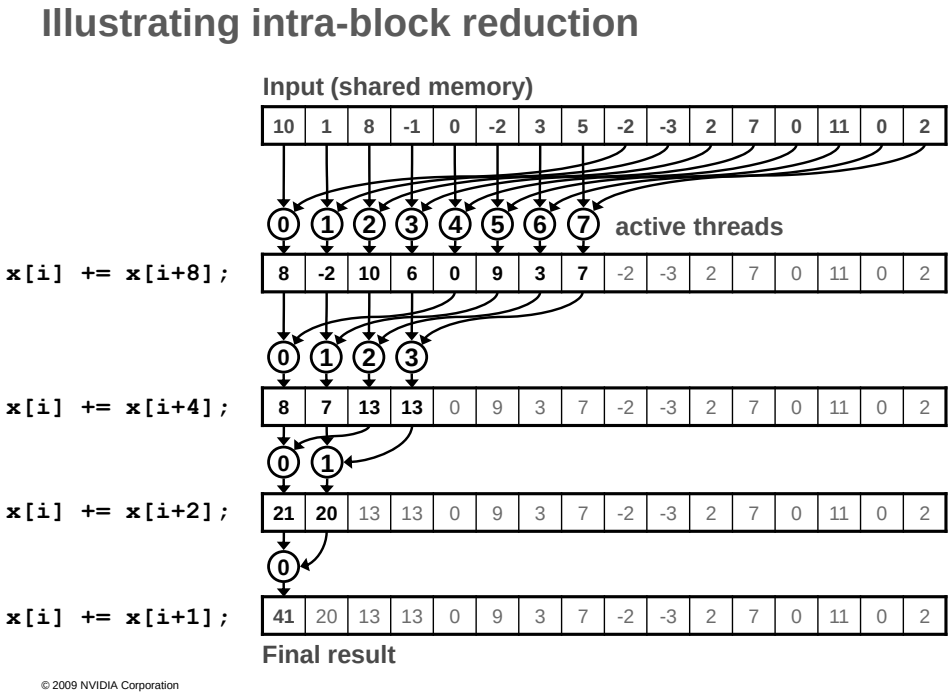


Figura 5.11: Esquema del algoritmo de reducción paralela.

La relación entre éstas es la media:

```
media = numerador_media/elementosMedia
```

Mediante reducción paralela

La reducción paralela es una técnica que nos permite procesar[5] un array en tan sólo $\log_2(tamaño)$ iteraciones. En la figura 5.11 podemos observar que cada hilo suma un elemento con el que esté a una distancia la mitad de la longitud. De esta forma, los resultados se van acumulando en la mitad izquierda, que se procesa en la iteración siguiente, y así sucesivamente. Al terminar, la suma total queda depositada en el primer elemento del array. Aquí no son necesarias operaciones atómicas porque las posiciones a las que los hilos acceden no se solapan. Sólo se requieren cuando varios hilos acceden a la misma posición.

Al igual que se comentó en 5.2.3, necesitamos dos variables numerador y denominador de la media (en el listado 5.7 numeradorMedia y elementosMedia, respectivamente), pero en esta ocasión no son simples enteros, sino arrays de enteros, de tamaño el tamaño del tile. En cada iteración de crecimiento de regiones se suman mediante reducción, y la división

[5]Ese procesamiento puede consistir en la suma, la multiplicación, el máximo... En este caso se quiere la suma.

de los respectivos primeros elementos nos da la media actualizada (listado 5.8). Dado que en reducción el vector se sobreescribe, es necesario mantener una copia, para manejar datos correctos en la siguiente iteración.

Listado 5.7: Reducción paralela para el cálculo de la media de color de la región

```
int mitad = 512 / 2;
while (mitad != 0){
  if (indexHilo < mitad){
    numeradorMedia[indexHilo].x += numeradorMedia[indexHilo + mitad].x;
    numeradorMedia[indexHilo].y += numeradorMedia[indexHilo + mitad].y;
    numeradorMedia[indexHilo].z += numeradorMedia[indexHilo + mitad].z;
    elementosMedia[indexHilo] += elementosMedia[indexHilo + mitad];
  }
  __syncthreads();
  mitad/=2;
}
```

Si utilizamos memoria compartida, es necesaria una barrera de sincronización entre iteraciones (`__syncthreads()` en el listado 5.7), para que las posteriores lean los valores correctos y actualizados de las anteriores.

Listado 5.8: Cálculo de la media de color de la región a partir de los resultados de la reducción

```
media_Y = numeradorMedia[0].x/elementosMedia[0];
media_Cb = numeradorMedia[0].y/elementosMedia[0];
media_Cr = numeradorMedia[0].z/elementosMedia[0];
```

Una observación importante es que el código de reducción aquí presentado sólo es válido para tamaños que sean potencias de dos. Si la longitud del tile no cumple esta condición, el tamaño numeradorMedia y elementosMedia debe redondearse a la potencia de dos más próxima.

La versión de crecimiento de regiones con reducción paralela supone una ventaja notable respecto a la versión con operaciones atómicas, como se verá en el capítulo de rendimiento.

5.3. Quitando espacios

La división de imagen utilizada deja espacios entre los tiles/regiones[6], como se puede observar en la figura 5.6. Esta vista no está mal para comprobar cómo evoluciona cada una de las regiones sin que nos moleste la vista las vecinas. Sin embargo, si queremos lograr

[6]Recuérdese que hasta ahora cada tile representa una región

resultados como los de la imagen 3.4 de los pimientos, debemos eliminarlos, ya que el tamaño de un *tile* no suele ser suficiente para cubir el objeto a segmentar.

Por tanto, vamos a utilizar otro diseño que no presente ese problema (figura 5.12):

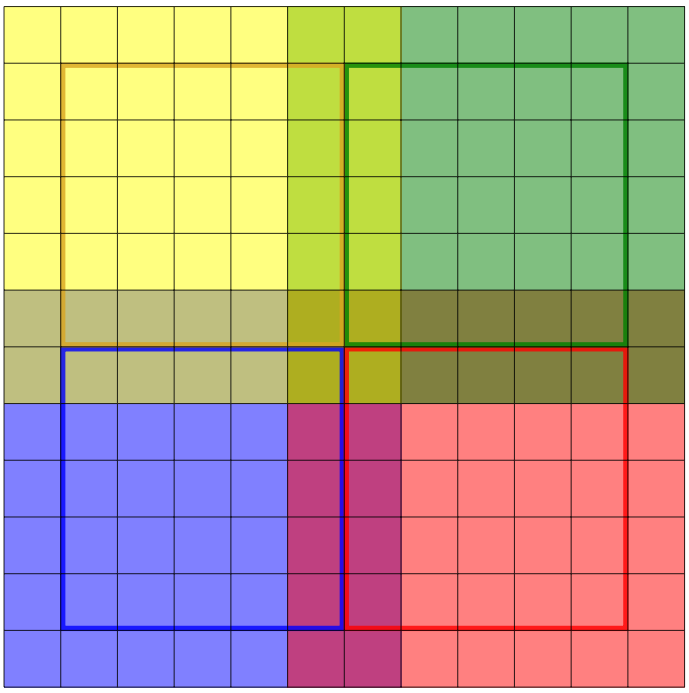


Figura 5.12: Diseño con solapamiento de *tiles*.

En el esquema se pueden observar cuatro *tiles* de color violeta, naranja, verde y amarillo, de tamaño 7×7, que se solapan los unos a los otros en 2 filas y 2 columnas. Los recuadros internos representan bloques de hilos, de 5×5. Al igual que antes, no existen hilos para los puntos de los bordes, pero ahora, debido al solapamiento, el relleno es continuo, sin espacios intermedios.

En el código, debemos realizar dos desplazamientos:

- De una posición para los hilos, tanto en filas como en columnas, para situar los hilos en los recuadros interiores.

Listado 5.9: Desplazamiento de una posición tanto en filas como en columnas para que los hilos se encarguen de los recuadros interiores

```
int tx = threadIdx.x;
int ty = threadIdx.y;
tx++;
ty++;

int Row = DIMTILE * blockIdx.y + ty;
int Col = DIMTILE * blockIdx.x + tx;
```

- De dos posiciones para las filas y columnas, para lograr el solapamiento de *tiles*.

Listado 5.10: Solapamiento de *tiles* en dos filas y dos columnas

```
Row -= 2*blockIdx.y;
Col -= 2*blockIdx.x;
```

5.4. Visión general

Recapitulemos, tanto lo que ya hemos hecho como lo que queda por hacer:

1. Formar las regiones iniciales, en nuestro caso aplicando crecimiento de regiones. Se deben inicializar las etiquetas de las regiones[7] con el índice de bloque. Como todavía no se han mezclado, todas serán distintas entre ellas. Éste es el trabajo realizado hasta ahora (ver figura 5.14).
2. Para cada región:
   a) Comprobar si las regiones adyacentes son similares.
   b) En caso de que así sea, juntarlas igualando sus etiquetas.
3. Repetir el paso 2 hasta que no queden regiones por mezclar.

Este cuadro guarda un gran parecido con el que se expuso en la sección 3.3. Realmente es lo mismo, con la salvedad de que ahora se trabaja con regiones en lugar de puntos.

[7] Otras implementaciones utilizan un *Region Adjacency Graph* o Grafo de Adyacencia de Regiones, pero aquí utilizo etiquetas por ser un sistema más sencillo.

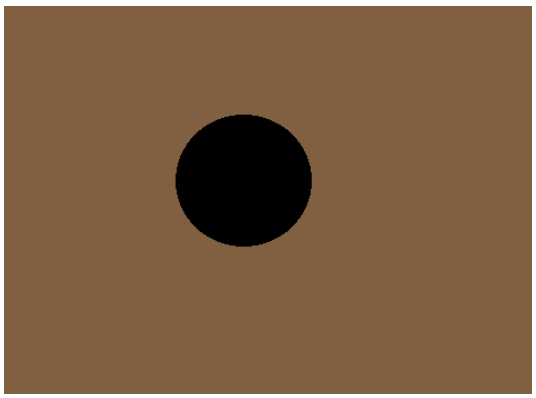


Figura 5.13: Imagen original.

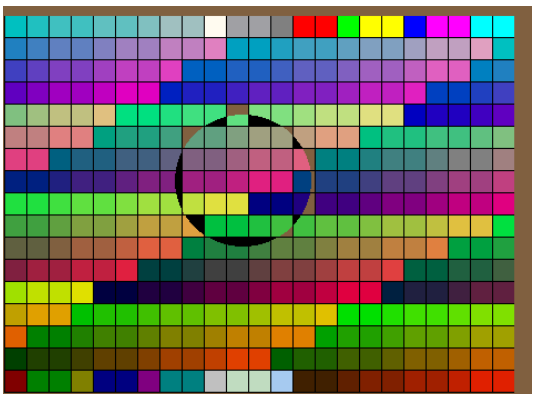


Figura 5.14: Hemos aplicado crecimiento de regiones a la figura 5.13 en cada tile. Hasta ahora, cada tile simboliza una región, y están coloreados con sus etiquetas iniciales, que son consecutivas.

5.5. Juntando regiones

Al igual que entonces, debemos definir qué entendemos por «similitud». Y, al igual que entonces, existen varias posibilidades:

- Teniendo en cuenta toda la región:
  - Intensidad media.
  - Ajuste de superficies (*surface fitting*) para determinar si las regiones se pueden aproximar como una superficie común.
  - Test de Hipótesis. Asume que las intensidades se ajustan a algún tipo de distribución de probabilidad; normalmente la gaussiana. Las regiones se juntan si tienen la misma distribución. Por ejemplo, suponiendo la gaussiana, mezclamos dos regiones si ambas se ajustan a una sola $N(\mu, \sigma)$; las mantenemos separadas si una encaja con $N(\mu_1, \sigma_1)$ y la otra con $N(\mu_2, \sigma_2)$.
- Teniendo en cuenta los bordes de la región:
  - Debilidad/Fortaleza de los fronteras entre regiones.

Para la implementación en CUDA se ha empleado este último criterio. La idea es combinar las regiones si la frontera entre ellas es «débil». Débil significa que las diferencias de intensidad a uno y otro lado sean menores que un cierto límite T_1. Veamos las expresiones:

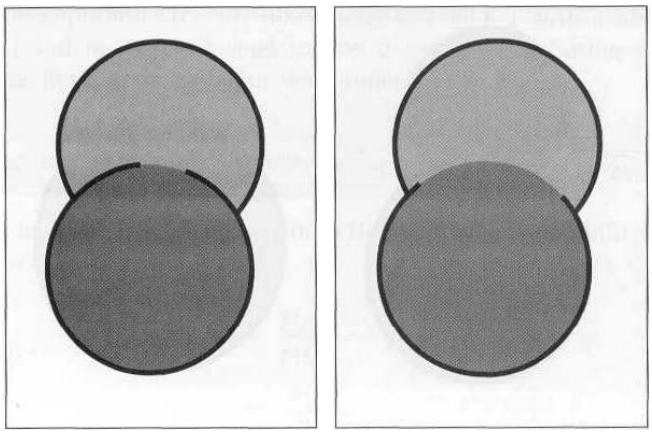
$\frac{W}{S} > T_2$

donde T_2 es otro umbral, distinto de T_1; W es la longitud de la parte débil de la frontera, y S puede ser dos cosas:

- $S = min(S_1, S_2)$, el menor de los perímetros de las regiones que estamos considerando juntar.

- La longitud de la frontera (la longitud del borde común).

Veamos la subfiguras 5.15a y 5.15b para entender mejor estos conceptos. En ellas se pueden observar dos regiones en forma de círculo, que estamos decidiendo juntar o no usando el criterio de borde débil. Los contornos en negro simbolizan un borde fuerte, y su ausencia borde débil (la longitud de éste es W). La expresión anterior no es más que un porcentaje. Según tomemos S, podría ser «el porcentaje del borde de la región (de menor perímetro) que es débil» o bien «el porcentaje de la frontera (la intersección de las dos circunferencias) que es débil». En cualquier caso, para 5.15a no juntaremos las regiones, ya que el borde es mayoritariamente fuerte. Sin embargo, sí que lo haremos en 5.15b, por ser borde mayoritariamente débil.



(a) Borde fuerte **(b)** Borde débil

Figura 5.15: Comparación de bordes fuertes y débiles.

5.5.1. Borde débil *versus* media

Al comienzo de esta sección se ha citado la intensidad media de las regiones como posible técnica para juntarlas. El problema de ese método es la dependencia del orden. Podemos acabar obteniendo resultados dispares, sobre todo si las regiones son similares.

Sin embargo, utilizando bordes débiles, el orden carece de importancia: dos regiones se unirán o no, sean procesadas antes o después, sin afectarles lo que haya ocurrido a su alrededor. Esta independencia de datos es ideal a la hora de paralelizar. Para una región en concreto, no necesitamos examinar todas las vecinas en búsqueda de la más adecuada, sino

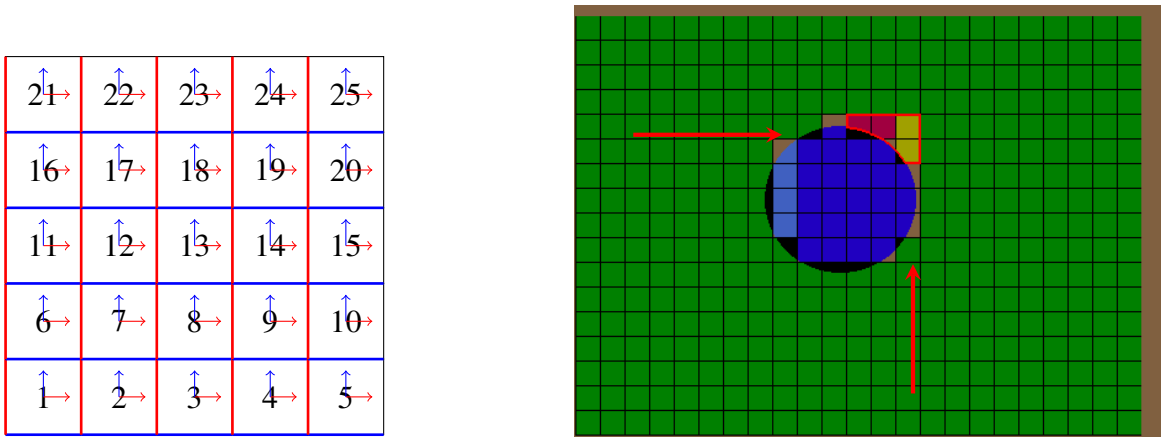


Figura 5.16: Esquema de propagación superior derecha y resultado. La dirección de propagación «choca» con el círculo generando un ángulo muerto (zona rodeada en rojo).

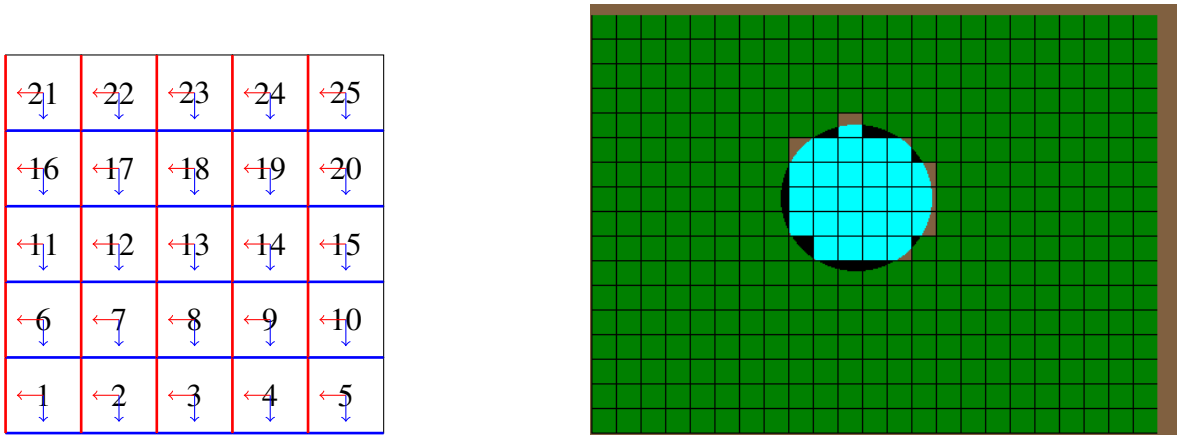


Figura 5.17: Esquema de propagación inferior izquierda (una vez aplicado superior derecha) y resultado.

que directamente podemos utilizar un orden prefijado para la dirección de propagación[8].

5.5.2. Dirección de propagación

En mi caso, empecé utilizando derecha y superior. El resultado obtenido es el mostrado en la figura 5.16. Podemos observar que se crea un «ángulo muerto», señalado en rojo, al noreste del círculo. Esa zona forma parte del fondo, pero, en lugar de mezclarse con la región en expansión a la que pertenece éste, aparecen como regiones individuales, en púrpura y verde pistacho. El problema es que la expansión de regiones «choca» con el objeto, dejando algunas zonas sin «inundar». Por solventarlo, tenemos que aplicar otra pasada en el sentido contrario al esquema 5.16, es decir, inferior e izquierda (5.17).

Al contrario que muchas otras librerías paralelas, y como ya se ha indicado, CUDA

[8]Desafortunadamente, encontré el método de borde débil demasiado tarde, y no me ha dado tiempo a implementarlo, así que utilizo la media.

no dispone de sincronización global (sincronización entre bloques de hilos). Es el precio a pagar a cambio de un alto nivel de paralelismo. De esta forma, el runtime no tiene ningún tipo de restricción respecto al orden de ejecución de los bloques de hilos, ya que no tienen que esperarse entre ellos. Esta independencia permite que diferentes gamas de GPUs puedan ejecutar el mismo código a diferentes velocidades: CUDA podrá lanzar muchos bloques simultáneamente, ocupando todos los SMs que le sea posible. Si hubiesen esperas entre bloques, se reduciría el paralelismo, desaprovechando esos recursos de ejecución.

Aunque no existe ningún método explícito de sincronización global, sí que hay uno implícito: dejar que el kernel acabe[9]. De esta manera, todos los hilos mueren y terminan las escrituras pendientes a memoria global. Como los bloques sólo comparten ésta, las dependencias de datos son importantes. Según dónde se produzan:

- En un bucle interno del kernel, donde el procesamiento en una iteración dependa de los datos generados en la anterior: podemos sacar ese bucle al código de la CPU, para que llame varias veces a dicho kernel.

- Fuera de un bucle interno del kernel: simplemente extraemos el código dependiente como un nuevo kernel.

Todas estas consideraciones son muy importantes a la hora de implementar el algoritmo. Realmente, está dividido en 3 kernels (ver tabla 5.1). El primero ejecuta crecimiento de regiones sobre cada tile, con un bloque de hilos por tile. También guarda en memoria global la información de borde débil. El segundo kernel junta regiones utilizando el procedimiento visto en la sección anterior. Sin embargo, sólo utiliza un bloque. Esto es ineficiente, pero no hay otro remedio: aunque todos los bloques puedan acceder a memoria global, no sirve de nada sin una sincronización entre ellos. No podemos empezar a juntar regiones hasta que estemos seguros de que los datos necesarios, los concernientes al borde débil, hayan sido escritos a memoria global. Esto se hace dejando que termine el kernel, y lanzando el código para juntar regiones como otro nuevo. De la misma manera, no podemos colorear la imagen hasta que la unión de regiones finalice. Esto exige la separación en un tercer kernel. Éste sí puede volver a usar todos los bloques que se utilizaron en el primero, porque toda la información requerida (etiquetas de puntos y regiones) ya está escrita en memoria, y no es necesaria ninguna sincronización adicional.

[9]En la guía de CUDA también aparece otro método consistente en hacer que el último bloque de hilos haga algo en especial; véase «Memory Fence Functions» en [9]

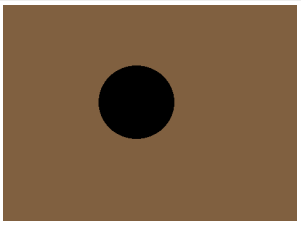
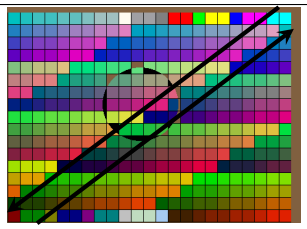
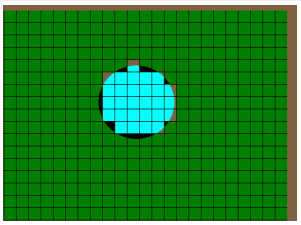
<table>
<tr><td>Fases</td><td>  </td><td colspan="2">  </td><td>  </td></tr>
<tr><td>Kernel aplicado sobre imagen</td><td>region Growing OnDevice1</td><td>region Growing OnDevi-ce2</td><td>region Growing OnDevi-ce3</td><td rowspan="2">Ninguno
(ya hemos acabado)</td></tr>
<tr><td>Número bloques de cada kernel</td><td>Varios bloques</td><td>Un sólo bloque</td><td>Varios Bloques</td></tr>
</table>

Tabla 5.1: Kernels aplicados sobre la imagen y número de bloques de cada kernel, para cada una de las fases del algoritmo de crecimiento de regiones.

5.5.3. Aplicando Striding

En la figura 5.18 se puede observar el funcionamiento de regionGrowingOnDevice2, en la que se ha aplicado *striding*. La numeración del 1 al 6 representa el orden en el que el bloque de hilos encargado de juntar regiones se desplaza por la memoria compartida para cubrir toda la imagen, señalada en rojo. Al igual que ocurría con los esquemas 5.16 y 5.17 de la sección 5.5, necesitamos una propagación derecha-arriba y otra izquierda-abajo. De la misma forma que en 5.3 esquema 5.12, existe un solapamiento, pero con dos diferencias:

- El fin es distinto: No se trata de eliminar espacios, sino de que el bloque «arrastre» lo que ha producido con anterioridad.
- El solapamiento es tan sólo de una posición, no de dos.

5.6. Estructuras de arrays *versus* arrays de estructuras

Un asunto recurrente en CUDA es la mejora que las estructuras de arrays suponen respecto a los arrays de estructuras[10]. La razón es la coalescencia.

[10]Si utilizamos memoria dinámica, como en los listados que aparecen a continuación, en lugar de arrays tenemos punteros, pero para lo que queremos comentar da lo mismo.

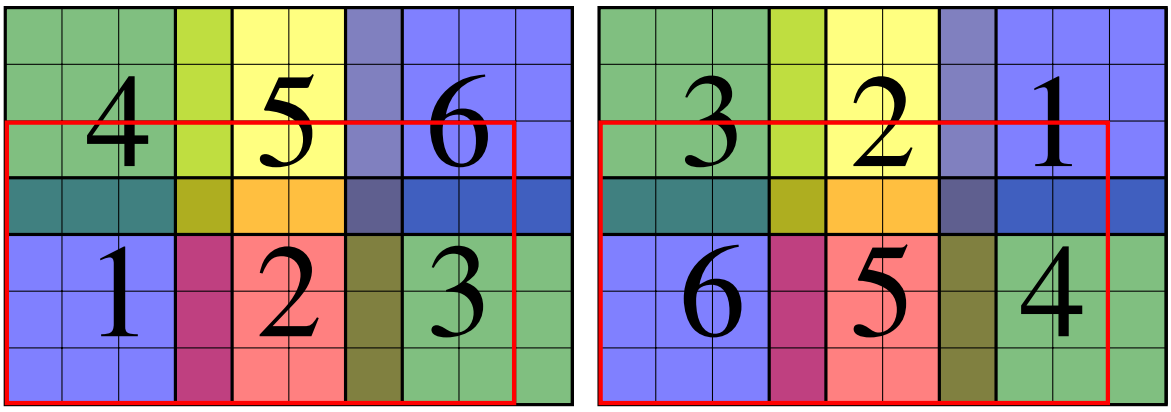


Figura 5.18: Striding aplicado en regionGrowingOnDevice2.

Veamos primero lo que no se debe hacer, arrays de estructuras:

Listado 5.11: Arrays de estructuras

```
typedef struct tagRGBTriplet{
  BYTE red;
  BYTE green;
  BYTE blue;
} AOSRGB;

AOSRGB* punteroAosRGB;
malloc(punteroAosRGB,NUMERO_PIXELES_IMAGEN*sizeof(AOSRGB);
```

Supongamos el caso en el que tengamos que calcular la media de la componente *R*. La figura 5.19[11] muestra la disposición de la memoria y el acceso de los hilos, señalados con flechas rojas.

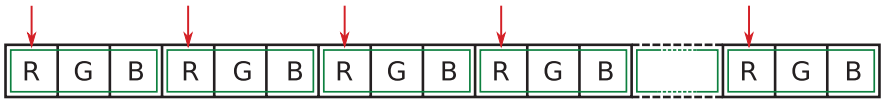


Figura 5.19: Array de estructuras

Los hilos saltan de tres en tres posiciones de memoria, por lo que los accesos no son coalescentes. Sin embargo, usando arrays de estructuras(listado 5.12), la cosa cambia.

Listado 5.12: Estructura de arrays

```
typedef struct tagRGBTriplet{
  BYTE* red;
  BYTE* green;
  BYTE* blue;
```

[11] Extraida de [14].

```
} SOARGB;

SOARGB punteroSoaRGB;
malloc(punteroSoaRGB.red,NUMERO_PIXELES_IMAGEN*sizeof(BYTE));
malloc(punteroSoaRGB.green,NUMERO_PIXELES_IMAGEN*sizeof(BYTE));
malloc(punteroSoaRGB.blue,NUMERO_PIXELES_IMAGEN*sizeof(BYTE));
```

Ahora hilos consecutivos acceden a posiciones consecutivas de memoria, y por tanto, hay coalescencia (figura 5.20[12].).

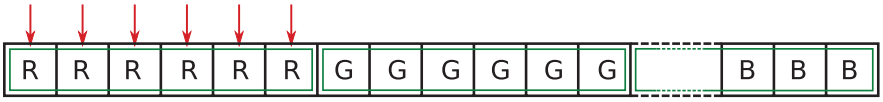


Figura 5.20: Estructuras de arrays

[12]También extraída de [14]

Capítulo 6

Rendimiento

Para la evaluación de rendimiento he empleado el depurador *Parallel Nsight* que nVIDIA ofrece para *Visual Studio*, concretamente la función *Application Trace*. También se pueden medir tiempos usando directamente el API de CUDA, insertando en el código «eventos» `cudaEvent_t`. Sin embargo, en *Parallel Nsight* todo se realiza automáticamente, sin necesidad de modificar el código. Además, es más poderoso, muestra más información, y los datos están perfectamente organizados.

En la figura 6.1 se muestra una captura de pantalla de la página principal de *Application Trace*. En el menú desplegable se pueden observar todas las posibilidades que proporciona. Nos interesa sobre todo los tiempos de cada kernel, y el cronograma. En todo este capítulo he empleado la misma imagen de prueba círculo.bmp, de tamaño 477×350, que incluyendo el *padding* de memoria de GPU, pasa a ser de 512×350. La configuración utilizada ha sido: 50 iteraciones crecimiento regiones, 50 iteraciones juntar regiones, 0.2 umbral crecimiento regiones, 0.2 umbral juntar regiones.

6.1. Atómicas *versus* reducción paralela

En 5.2.3 se comentó la mejora sustancial de la reducción paralela frente a las operaciones atómicas en el cálculo de la media de color de la región. Veamos sin más dilación los resultados de los tiempos, para GTX 460:

Podemos observar una aceleración significativa de `regionGrowing OnDevice` en los datos de la figura 6.3 frente a los recogidos en 6.2. Concretamente, 3.57 veces más rápido. No está mal.

Nsight también dispone de un cronograma que nos permite hacernos una idea global del rendimiento. Podemos comprobar la aceleración gráficamente[1] en las figuras 6.4 y 6.5:

[1] **Ambos** *timelines* tienen aproximadamente una amplitud de 0.15 segundos, para poder hacer así una comparación justa «a ojo».

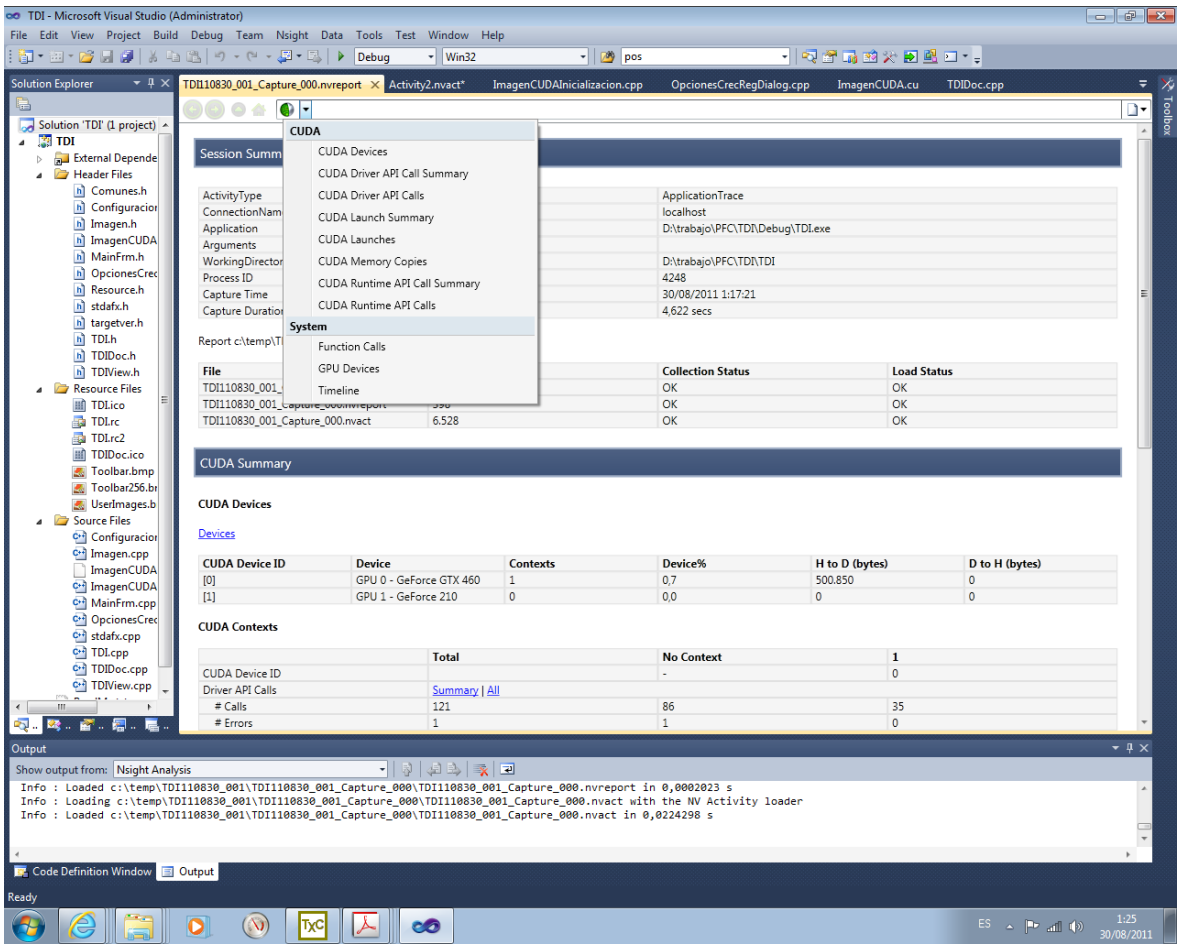


Figura 6.1: Application Trace del depurador Parallel Nsight, dentro de Visual Studio 2010. En el menú desplegable se muestran todas las posibilidades.

Top Device Functions by Total Time

Launch Summary | All

Name	Launches	Device%	Total (µs)	Min (µs)	Avg (µs)	Max (µs)
regionGrowingOnDevice	1	2,4	115.604,604	115.604,604	115.604,604	115.604,604
regionGrowingOnDevice2	1	0,0	1.299,850	1.299,850	1.299,850	1.299,850
RGBtoYCbCr	1	0,0	163,444	163,444	163,444	163,444
YCbCrtoRGB	1	0,0	135,473	135,473	135,473	135,473
regionGrowingOnDevice3	1	0,0	132,655	132,655	132,655	132,655

Figura 6.2: Tiempos de los kernels para operaciones atómicas.

Top Device Functions by Total Time

Launch Summary | All

Name	Launches	Device%	Total (µs)	Min (µs)	Avg (µs)	Max (µs)
regionGrowingOnDevice	1	0,6	32.377,218	32.377,218	32.377,218	32.377,218
regionGrowingOnDevice2	1	0,0	1.299,729	1.299,729	1.299,729	1.299,729
RGBtoYCbCr	1	0,0	164,171	164,171	164,171	164,171
YCbCrtoRGB	1	0,0	135,784	135,784	135,784	135,784
regionGrowingOnDevice3	1	0,0	132,708	132,708	132,708	132,708

Figura 6.3: Tiempos de los kernels para reducción paralela.

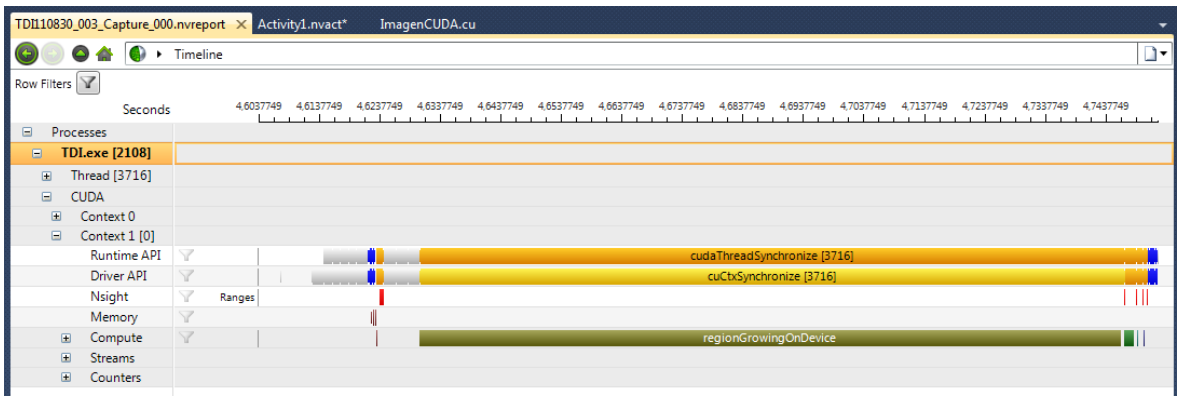


Figura 6.4: Cronograma para operaciones atómicas.

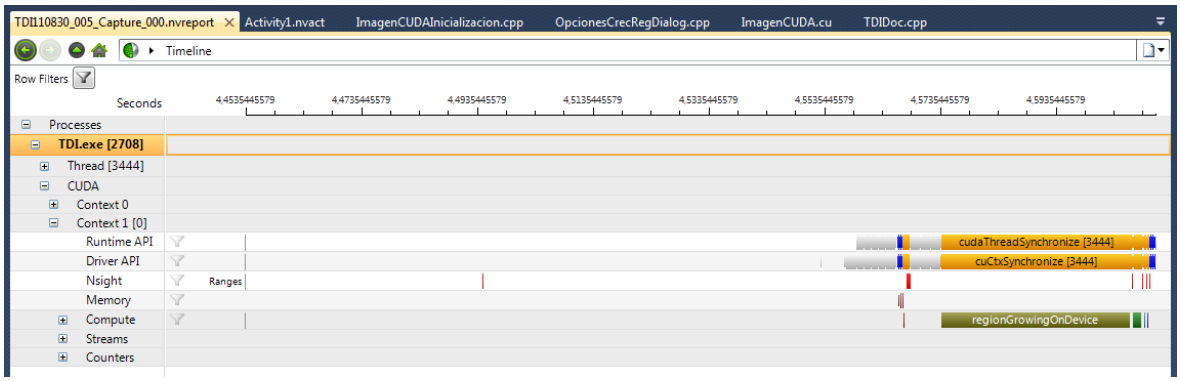


Figura 6.5: Cronograma para reducción paralela.

6.2. Influencia de las dimensiones de bloque en la coalescencia

Las dimensiones del bloque influyen de manera muy notable en el número de segmentos de memoria accedidos, así como en su eficacia[2]. En la subfigura 6.6a podemos observar, arriba, una fila de la imagen[3], separada en segmentos de 32 bytes[4] de color rojo y amarillo. Abajo aparecen, diferenciados en verde y azul, bloques de hilos de 20×20 (por hacer el análisis más sencillo, me he centrado aquí en el caso de los kernels `RGBToYCbCr` e `YCbCrToRGB`, que no tienen solapamiento, donde tanto bloques como tiles tienen las mismas dimensiones de 20×20).

El bloque con una cruz roja se ha ampliado abajo, en la subfigura 6.6b[5]. Está a caballo

[2]Cantidad de información del segmento que realmente se utiliza.

[3]Puesto que están alineadas mediante cudaMallocPitch, todas las filas de la imagen siguen esa misma distribución. Recordemos, además, que el ancho pasa de 477 a 512, por el *padding*.

[4]Puesto que los hilos acceden a datos de tamaño byte, según la reglas de coalescencia vistas en 4.4.2, el tamaño mínimo de segmento es de 32 bytes.

[5]Nótese que, aunque en el código se utilicen hilos de dos dimensiones, internamente se trabaja con una dimensión (thread ID), que es lo que importa para la coalescencia, y lo que se presenta en el esquema

(a) Arriba, una fila de la imagen. La sucesión rojo-amarillo distingue segmentos de 32 bytes. Abajo, bloques de hilos de 20x20, diferenciados en colores verde y azul.

28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3
28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3
28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

(b) Bloque señalado con una cruz roja en 6.6a, ampliado.

Figura 6.6

entre dos segmentos de 32 bytes. La división se ha marcado con una línea verde. Los distintos colores representan el tamaño de la coalescencia[6]: rojo para transacciones de segmentos de 64 bytes, y azul para las de 32. En éste último, las distintas tonalidades simbolizan accesos a segmentos distintos, porque se pasa a otro *half-warp*[7] justo cuando cruzamos la línea verde. Si, por el contrario, el half-warp está a caballo entre los dos segmentos de 32 bytes, se genera una sola transacción de 64. Por ejemplo, en primera fila empezando por abajo, el half-warp (hilos 0 al 15) está entre dos segmentos de 32 bytes (separados por la línea verde), por lo que se genera una sola transacción de 64 (rojo). Los hilos 16 al 19 pertenecen a otro half-warp y

[6]Como estamos trabajando en dos dimensiones, la coalescencia se da entre hilos de la misma fila, no entre hilos de distintas filas.

[7]Estamos suponiendo dispositivos de capacidad computacional 1.x, por lo que los accesos coalescentes son por *half-warp*. En el caso de 2.x, son por *warp*.

sólo acceden al segmento derecho; es una transacción de 32 (azul). En la segunda fila, también empezando por abajo, dos half-warps distintos acceden a segmentos distintos, y por tanto son dos transacciones de 32 bytes (en azul, diferenciadas por la tonalidad). El número total de transacciones es[8] $15 \times 64B + 25 \times 32B$.

Volviendo a la subfigura 6.6a, vemos que algunos bloques no acceden a varios segmentos, como el caso que acabamos de ver, sino a uno sólo. La secuencia la podemos observar en la fila superior del cuadro 6.1:

1	2	1	2	2	1	2	1	1	2	1	2	2	1	2	1	1	2	1	2	2	1	2	1	1	1
-	12	-	4	16	-	8	-	-	12	-	4	16	-	8	-	-	12	-	4	16	-	8	-	-	12

Tabla 6.1: En la fila superior, el número de segmentos a los que accede cada uno de los bloques. En la inferior, el offset de los hilos con respecto al inicio del segmento (no se ha incluido («-») el de los que acceden a un solo segmento, ya que no lo necesitamos para saber el número de transacciones.

Los bloques que sólo acceden a un segmento generan $20 \times 64B$ transacciones, y los que acceden a dos, $15 \times 64B + 25 \times 32B$, como acabamos de ver[9]. Si hacemos cuentas con la secuencia anterior, el número total por fila de bloques, y por imagen total[10], será:

$$\left.\begin{array}{l} 13\frac{bloques\ que\ acceden\ 1\ segmento}{fila\ imagen} \cdot (20\frac{filas}{bloque} \cdot 2\frac{halfwarps}{fila\ bloque} \times 32B) \\ = 520 \times 32B \\ 12\frac{bloques\ que\ acceden\ 2\ segmentos}{fila\ imagen} \cdot (15 \times 64B + 25 \times 32B) \\ = 180 \times 64B + 300 \times 32B \end{array}\right\} Total\ fila$$

En la ecuación superior, he puesto 13 y no 14 (bloques por fila de imagen) porque, recordemos, mi programa no cubre toda la imagen (ver 5.2.2). La explicación del paréntesis $(20\frac{filas}{bloque} \cdot 2\frac{halfwarps}{fila\ bloque} \times 32B)$ es que el bloque tiene 20 filas, con dos *half-warps* por fila del bloque (compruébese en la subfigura 6.6b), que hacen sendas transacciones al mismo segmento. Calculemos ahora cuántas se producen en toda la imagen:

$$Total\ fila = 180 \times 64B + 820 \times 32B$$
$$Total\ imagen = \left\lfloor \frac{350}{20} \right\rfloor \cdot (180 \times 64B + 820 \times 32B) = 3060 \times 64B + 13940 \times 32B$$

[8] `número«B»` significa una transacción de ese número de *bytes*

[9] Da la casualidad de que son las mismas para los segmentos con un offset de 4, 8, 12 y 16. El lector puede comprobarlo haciendo un estudio como el de la subfigura 6.6b

[10] Recordemos que estamos utilizando la imagen de prueba circulo.bmp, con dimensiones 477×350, con *padding* 512×350.

Utilizo el truncamiento por abajo (*floor*) porque, nuevamente, mi programa no cubre toda la imagen (ver 5.2.2).

Si el tamaño de bloque fuese de 16x16, tendríamos la distribución de la figura 6.7, y el número de transacciones sería:

$$\begin{aligned} Total\ fila &= 32\frac{bloques\ que\ acceden\ 1\ segmento}{fila\ imagen} \cdot (16\frac{filas}{bloque} \cdot 1\frac{halfwarps}{fila\ bloque} \times 32B) \\ &= 512 \times 32B \\ Total\ imagen &= \left\lfloor \frac{350}{16} \right\rfloor \cdot (512 \times 32B) = 10752 \times 32B \end{aligned}$$

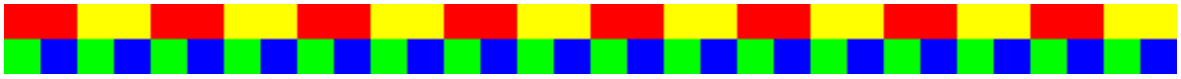


Figura 6.7: Arriba, fila de la imagen. Abajo, bloques de hilos de 16×16.

Comparemos el rendimiento real en GeForce 210[11], para la imagen de prueba circulo.bmp:

Top Device Functions by Total Time

Launch Summary | All

Name	Launches	Device%	Total (µs)	Min (µs)	Avg (µs)	Max (µs)
regionGrowingOnDevice	1	8,9	756.675,423	756.675,423	756.675,423	756.675,423
regionGrowingOnDevice2	1	0,2	16.466,287	16.466,287	16.466,287	16.466,287
regionGrowingOnDevice3	1	0,0	1.408,604	1.408,604	1.408,604	1.408,604
RGBtoYCbCr	1	0,0	1.001,821	1.001,821	1.001,821	1.001,821
YCbCrtoRGB	1	0,0	968,317	968,317	968,317	968,317

Figura 6.8: Rendimiento con bloques de 16x16.

Top Device Functions by Total Time

Launch Summary | All

Name	Launches	Device%	Total (µs)	Min (µs)	Avg (µs)	Max (µs)
regionGrowingOnDevice	1	6,3	548.177,446	548.177,446	548.177,446	548.177,446
regionGrowingOnDevice2	1	0,1	8.677,509	8.677,509	8.677,509	8.677,509
regionGrowingOnDevice3	1	0,0	1.472,924	1.472,924	1.472,924	1.472,924
RGBtoYCbCr	1	0,0	1.100,733	1.100,733	1.100,733	1.100,733
YCbCrtoRGB	1	0,0	1.061,661	1.061,661	1.061,661	1.061,661

Figura 6.9: Rendimiento con bloques de 20x20.

Comparando los resultados de 6.8 y 6.9, podemos comprobar que con 20x20 los tiempos son peores para `RGBToYCbCr` e `YCbCrToRGB`, como corroboran nuestros cálculos del número de transacciones.

Resumiendo, esta sección sirve para destacar que la coalescencia no sólo consiste en alinear los datos a los segmentos[12], sino que además la alineación de los hilos respecto a

[11] No lo pruebo con GTX 460 porque ésta tiene *cache* y alteraría los resultados.

[12] Mediante cudaMallocPitch.

dichos segmentos también es relevante.

Como curiosidad, en `regionGrowingOnDevice` ocurre lo contrario: un bloque grande mejora los resultados. Atribuyo este fenómeno a dos causas:

1. Un bloque mayor genera menos solapamientos, y por tanto menor desaprovechamiento para cubrir la imagen.

2. Como se emplea reducción, será mejor utilizar más hilos por bloque, ya que conforme incrementamos en iteraciones, el número de hilos menor que la variable *mitad* se reduce exponencialmente, es decir, hay cada vez más hilos inactivos.

Capítulo 7

Conclusiones

En este capítulo incluyo algunas reflexiones acerca del desarrollo de este proyecto. Como se comentó al principio, cada arquitectura tiene sus peculiaridades, y CUDA no es una excepción.

7.1. Memoria

Uno de los inconvenientes de CUDA es que no se puede reservar memoria dinámicamente en código del dispositivo, dentro del kernel, salvo para dispositivos de capacidad computacional 2.x. Y en estos sólo para memoria global, no compartida. Hubiese sido beneficioso puesto que podríamos guardar el contorno de la región[1] en lugar de todos los píxeles que la contienen, ahorrando así memoria.

7.2. Código C y C++

Si el lector echa una ojeada al código, puede que se sorprenda al ver que se utilizan algunas estructuras de C (`struct`) en el código C++, en lugar de clases. Esto es debido a que el código de dispositivo es esencialmente C, y no admite clases. Por tanto, los datos que se comparten entre CPU y GPU o son tipos simples o estructuras de C.

7.3. Sincronización global

A lo largo de este documento he resaltado en varias ocasiones el problema de la sincronización global. Para lograrla en CUDA, tenemos que recurrir a la parada del kernel, o bien a utilizar funciones de *memory fence*. En caso de que se produzcan datos dependientes entre bloques, uno final debe, él sólo, operar con los resultados obtenidos por los demás. Ni que decir tiene que esto es ineficiente, pues se desaprovechan gran parte de los recursos de ejecución. No hay otro remedio, salvo que ese último trabajo lo haga la CPU, claro.

[1] Como se explica en 3.3.4.

Ya se comentó en la subsección 4.5.1 que esto es así por diseño. CUDA permite una gran escalabilidad, con hasta miles de bloques y millones de hilos, pero el número de los que realmente son activos/residentes es muchísimo menor, los justos y necesarios para ocultar la latencia a memoria. Si se quisiese implementar una sincronización global, *todos* los hilos lanzados tendrían que ser residentes. Y eso supondría un coste elevado del hardware que, por otra parte, no mejoraría gráficos 3D, el principal interés, no lo olvidemos, de los fabricantes de GPUs. No estoy diciendo que esto sea ni *bueno* ni *malo*. Como todo en la vida, es bueno o malo según las necesidades de cada programa. En aplicaciones con una total independecia de datos, como el clásico ejemplo de la multiplicación de matrices, es beneficioso. En el caso de mi programa, pues... supone más bien un problema. Realmente, he realizado un algoritmo de crecimiento de regiones *local*, dentro de cada bloque, utilizando memoria compartida, y luego confío que el kernel para juntar regiones haga un buen trabajo. En [14] utilizan desde el principio un algoritmo *global*, con memoria global, aunque, eso sí, deben parar el kernel entre cada iteración. Esto es mucho más fiel a la idea original vista en 3.3.4, y produce resultados mucho mejores que los míos, pero también es más lento. Además, la media de regiones la calculan en la CPU, con el trasiego de información que ello supone.

Personalmente, desde la ignorancia de no haber trabajado con otros APIs/plataformas paralelos como OpenMP o MPI, creo que sería mejor probar crecimiento de regiones en un sistema que soporte sincronización global. Comparando mi trabajo con el de [14], me da la impresión de que, en CUDA, con este algoritmo, o bien se implementa de forma global y se obtienen buenos resultados con un rendimiento bajo, o bien se sacrifican los resultados con un desarrollo local, a cambio de mayor velocidad.

Apéndice A

Sobre el programa

El programa está destinado a la plataforma Windows. Para su programación se ha empleado la librería MFC[1]. En este trabajo, se ha usado básicamente para proporcionar una interfaz gráfica y leer/escribir ficheros. Para ejecutarlo, se necesita:

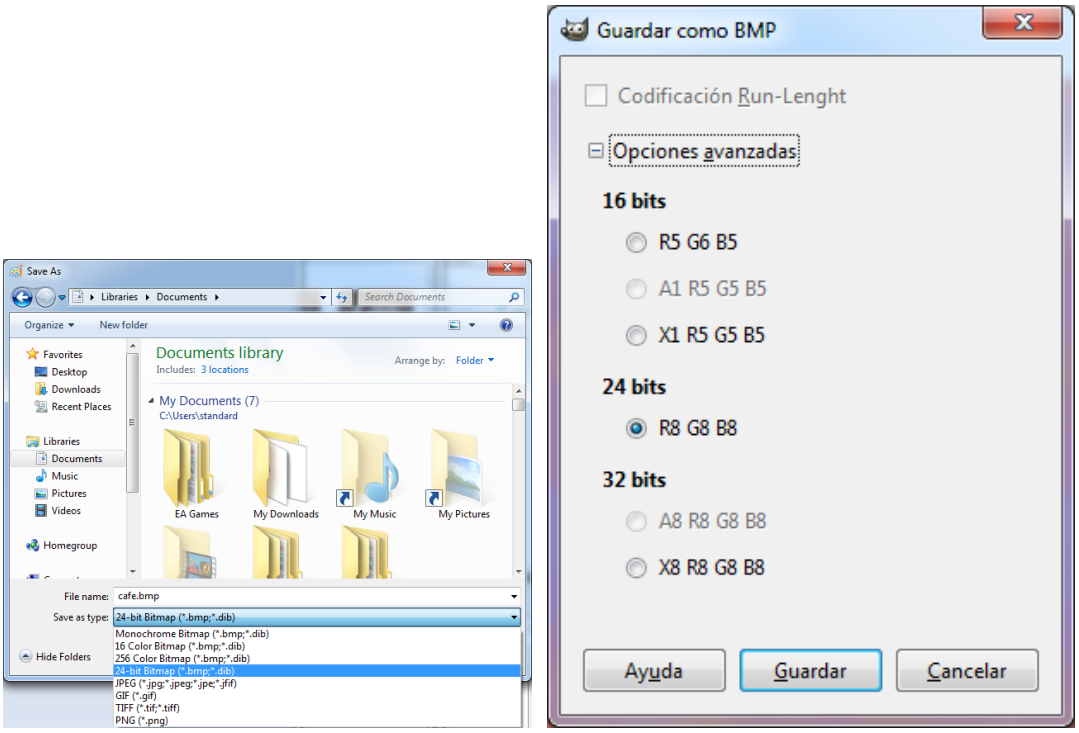
- Una GPU nVIDIA con «compute capability» 1.0 o superior.
- En un mismo directorio:
  - El ejecutable `TDI.exe`.Se proporcionan dos versiones, de 32 y 64 bits.
  - La librería `cudart32_40_17.dll` y `cudart64_40_17.dll`. Son las librerías de 32 y 64 bits, respectivamente, del «CUDA Runtime API», versión 4.0.17.
- La versión 270 del driver de nVIDIA (270.x) o superior. No hace falta que sea el «developer driver».

Para ejecutarlo, no es necesario tener instalado el *Toolkit* o el *SDK* de *CUDA*, tan sólo la versión mencionada del driver.

IMPORTANTE: Windows Vista y 7 incorporan un sistema de detección y recuperación ante «cuelgues» de la GPU, denominado TDR[2]. Con este mecanismo, la GPU es reiniciada si no responde durante dos segundos. De esta forma, el usuario puede seguir trabajando sin necesidad de resetear todo el sistema. Es una buena característica, pero en CUDA puede ser un inconveniente: si sólo disponemos de una GPU en el equipo, ésta estará inevitablemente conectada al monitor, por lo que un kernel que se ejecute durante más de dos segundos será detenido cuando TDR reinicie la GPU. Dependiendo del tamaño de la imagen y la GPU utilizada, mi programa puede llegar a durar más de dos segundos. Si eso sucede y el lector desea probar el programa, deberá desactivar *temporalmente* TDR. Para ello se proporcionan dos entradas de registro: `DisableTDRWpfHardwareAcceleration.reg`, y `EnableTDRWpfHardwareAcceleration.reg`, que deshabilitan y habilitan, respectivamente, TDR. Para añadir la entrada de registro, simplemente se hace doble clic sobre

[1] *Microsoft Foundation Classes*

[2] WDDM Timeout Detection and Recovery.



(a) Diálogo Guardar en Paint. **(b)** Diálogo Guardar en GIMP.

Figura A.1: 24 bits por píxel es la opción por defecto en muchos programas para guardar ficheros BMP.

el icono del fichero `.reg`. Es *estrictamente necesario* reiniciar el sistema a continuación. Permítame el lector insistir en este punto. Las entradas de registro sólo tienen efecto una vez hemos reiniciado *Windows*.

NOTA: Este programa utiliza compilación JIT (Just-In-Time). Esto quiere decir que, cuando en el programa presionamos el menú «Crec Reg CUDA», el runtime de CUDA convierte el código `ptx` en binario. Esto puede causar un leve retraso en la primera ejecución, que desaparece en las siguientes.

El funcionamiento es bastante sencillo:

1. Abrimos una imagen BMP desde el icono correspondiente de la barra de herramientas, o bien desde el menú, escogiendo `Archivo->Abrir...` (Figura A.2). Los ficheros BMP son normalmente de 1, 4, 8 ó 24 bits por píxel. Sin embargo, mi programa *sólo* admite los de 24 bits por píxel, que dicho sea de paso son los más comunes (ver figura A.1).

2. Seleccionamos `Segmentación->Crec Reg CUDA` desde el menú. (Figura A.3).

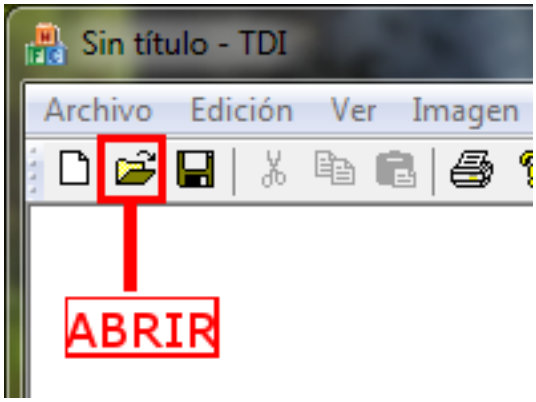


Figura A.2: Primer Paso. Abrir Imagen

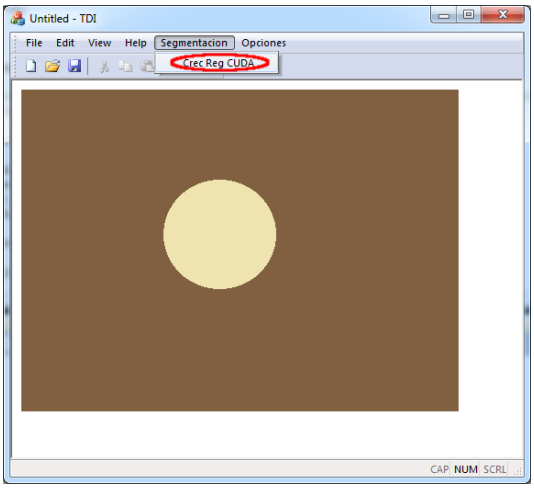


Figura A.3: Segundo paso. Seleccionar opción «Crecimiento Regiones CUDA».

A.1. Selección de dispositivos y diagnóstico

En el programa se incluye un menú Opciones->Configuración CUDA. Al acceder a él, se presenta la caja de diálogo de la figura A.5. La lista «GPU seleccionada» muestra la GPU seleccionada por defecto, que es la más potente[3] de las detectadas en el sistema. Si se despliega la lista haciendo click sobre ella, aparecen las demás si las hubiese, en orden decreciente de potencia. El usuario puede seleccionarlas si así lo desea. Asimismo, en la parte inferior hay una caja de texto con una serie de diagnósticos, por si hubiese algún problema durante la detección. Por ejemplo, en mi sistema se muestra esto:

```
Diagnóstico de CUDA
=======================================================

Detección del driver de NVIDIA:                          PASS
Versión del runtime CUDA en el driver NVIDIA:            PASS
Ausencia de emulación por CPU -- Dispositivo [0]:        PASS
Capacidad computacional -- Dispositivo [0]:              PASS
Timeout -- Dispositivo [0]:                              PASS
Ausencia de prohibición para cómputo -- Dispositivo [0]: PASS
Ausencia de emulación por CPU -- Dispositivo [1]:        PASS
Capacidad computacional -- Dispositivo [1]:              PASS
Timeout -- Dispositivo [1]:                              PASS
Ausencia de prohibición para cómputo -- Dispositivo [1]: PASS
```

[3]Considero potencia como $Número\ multiprocesadores \times frecuencia$

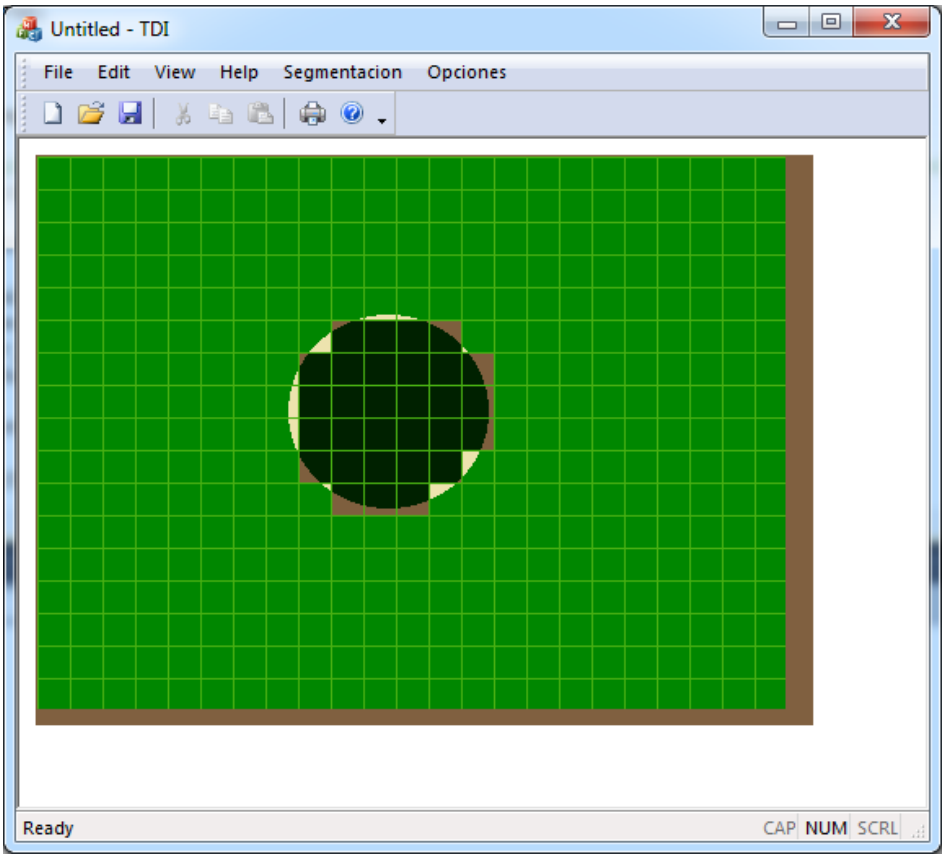


Figura A.4: Imagen tras seleccionar «Crecimiento Regiones CUDA».

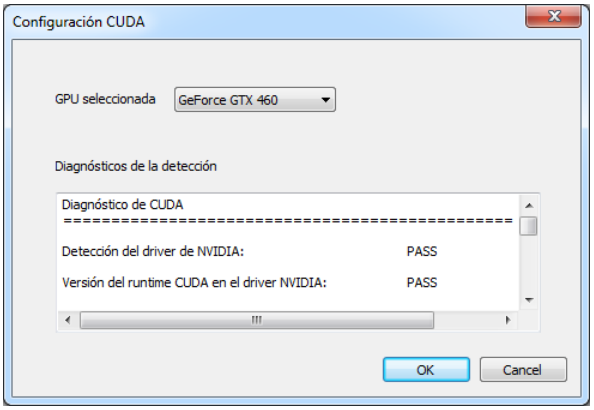


Figura A.5: Caja de diálogo Configuración CUDA

Apéndice B

Ficheros del CD-ROM

Consultar el fichero LEEME.txt del CD-ROM incluido en la contraportada.

En el CD-ROM se incluyen la memoria y el código.

Todo el código es mío, con dos puntualizaciones:

- Lectura y escritura de ficheros BMP: He utilizado el código de [13], con las necesarias adaptaciones para que funcione con mis estructuras SOARGB y SOAYCbCr.
- Detección y diagnóstico de dispositivos CUDA: He utilizado el código de [16], con las necesarias adaptaciones para que funcione con mi caja de diálogo «Configuración CUDA».

Apéndice C

Sistema Utilizado

Hardware	
Componente	Modelo
Procesador	Core 2 Duo E6600 @ 2.40GHz
Memoria	Kingston KVR533D2N4K2/4G 4GB PC2-4200 CL4
Placa	`ASUS P5W DH Deluxe/WIFI-AP <GREEN>` 90-MBB2X0-G0EAYZ
Tarjetas Gráficas	Sparkle SXX460768D5NM GeForce GTX 460 768MB PCI-E MSI VN210-MD512H GeForce 210 512MB PCI-E

Software	
Herramienta	Nombre
Sistema Operativo	Windows 7 Professional 64-bit
CUDA Runtime	4.0.17
Driver nVIDIA	270.81 (developer driver)
Entorno de desarrollo	Visual Studio 2010 Professional Version 10.0.30319.1 RTMRel
Compilador CUDA	nvcc 4.0, V0.2.1221
Depurador CUDA	Parallel Nsight 2.0 `Build# 2.0.11140.2`

Apéndice D

Punteros a memoria compartida y el depurador

Todos conocemos las bondades de los punteros. Son útiles, por ejemplo, para intercambiar (*swap*) zonas de memoria. Además, son indispensables en las reservas dinámicas de memoria[1].

En CUDA hay que tener muy presente una diferencia: en qué espacio de memoria se almacena un puntero, y a qué espacio de memoria apunta. Cuando trabajamos desde código host haciendo llamadas `cudaMalloc`, realmente utilizamos punteros almacenados en memoria principal del sistema, que apuntan a memoria del dispositivo[2]. En código del dispositivo, dentro del kernel, se pueden todas las combinaciones entre memoria `__device__`,`__shared__` y `__local__`[3]. En la figura D.1 se presenta un caso en el que una variable `__local__` apunta a `__shared__`. En CUDA no existen (ni tampoco en C) cualificadores de punteros para indicar a dónde apuntan. El compilador nvcc se encarga de llevar cuenta de ello[4]. Sin embargo, el depurador desconoce esta información, y es necesario pasarle un cualificador de la forma (`__shared__ <tipo_variable>puntero`), como se puede observar abajo a la derecha en la figura D.1. Sin esto, el valor es deconocido (indicado como `???`).

[1] Cosa que también se puede hacer con memoria compartida en CUDA, usando `extern __shared__ <variable>[]`, y añadiendo la cantidad en bytes como tercer parámetro en la configuración de ejecución del kernel.

[2] En este caso, podemos saber la dirección del puntero, pero no podemos acceder directamente a memoria del dispositivo sin utilizar funciones del API como `cudaMemcpy`.

[3] Ésta última para variables automáticas del kernel.

[4] Salvo que se confunda, entonces muestra la advertencia `Warning: Cannot tell what pointer points to, assuming global memory space`

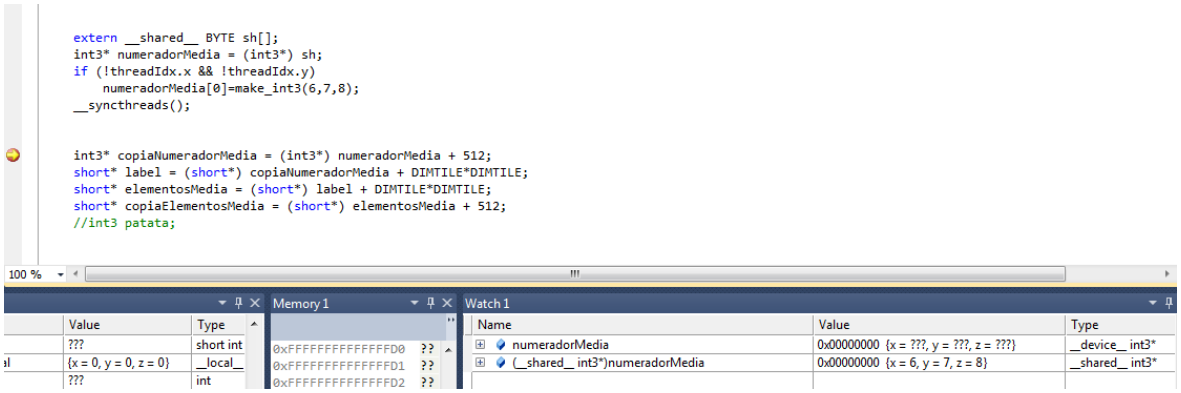


Figura D.1: Variables automáticas apuntando a shared

Bibliografía

[1] Bebis. Cs474/674 course lecture: Segmentation. http://www.cse.unr.edu/~bebis/CS474/Lectures/RegionBasedSegmentation.ppt.

[2] Bebis. Cs791e course notes. http://www.cse.unr.edu/~bebis/CS791E/Notes/RegionSplitMerge.pdf.

[3] Oster Brent and Greg Ruetsch. Getting started with cuda, nvision '08. http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf.

[4] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing, 2010.

[5] Wu chun Feng and Shucai Xiao. To gpu synchronize or not gpu synchronize? In *International Symposium on Circuits and Systems (ISCAS 2010)*, pages 3801–3804, 2010.

[6] J Cohen and M Jeroen Molemaker. A fast double precision cfd code using cuda. *Aerospace*, di(21):2237–2341, 2009.

[7] NVIDIA Corporation. Cuda api reference manual, version 4.0. http://developer.nvidia.com/cuda-toolkit-40.

[8] NVIDIA Corporation. Cuda c best practices guide, version 4.0. http://developer.nvidia.com/cuda-toolkit-40.

[9] NVIDIA Corporation. Nvidia cuda c programming guide, version 4.0. http://developer.nvidia.com/cuda-toolkit-40.

[10] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[11] David L. Donoho, Iain M. Johnstone, Gerard Kerkyacharian, and Dominique Picard. Density estimation by wavelet thresholding. *Ann. Statist*, pages 508–539, 1996.

[12] R.C. González and R.E. Woods. *Tratamiento digital de imágenes*. Addison Wesley Iberoamericana, 1996.

[13] Andreas Hartl. Bitmap tutorial: Loading and saving bitmaps. `http://tipsandtricks.runicsoft.com/Cpp/BitmapTutorial.html`.

[14] Markus Hofmann and Tobias Binna. *Massive Parallel Image Processing*. Bachelor of science fho in informatik, student research project, HSR Hochschule für Technik Rapperswil, Switzerland, 2010.

[15] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1 edition, February 2010.

[16] Usuario de nVIDIA forums MisterAnderson42. Código de selección de gpu, para la aplicación hoomd-blue. `http://forums.nvidia.com/index.php?showtopic=157185`.

[17] John Owens. Data level parallelism: Dlp project. `http://www.nvidia.com/content/cudazone/cudau/courses/ucdavis/projects/DLP_Project_Assignment_S09.pdf`.

[18] N R Pal and S K Pal. A review on image segmentation techniques. *Pattern Recognition*, 26(9):1277–1294, 1993.

[19] Peter Rogelj. Industrial ocr. `http://vision.fe.uni-lj.si/research/indOCR/index.html`.

[20] Bryan S. Morse. Image segmentation. `http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MORSE/region.pdf`.

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

[22] Frank Y. Shih and Shouxian Cheng. Automatic seeded region growing for color image segmentation. *Image and Vision Computing*, 23(10):877 – 886, 2005.

[23] F.Y. Shih. *Image Processing and Pattern Recognition: Fundamentals and Techniques*. John Wiley & Sons, 2009.

[24] Mike Spann. Image segmentation. `http://www.eee.bham.ac.uk/spannm/Teaching%20docs/Computer%20Vision%20Course/Image%20Segmentation.ppt`.

[25] L. Spirkovska. A summary of image segmentation techniques, 1993.

[26] Shucai Xiao and Wu chun Feng. Inter-block gpu communication via fast barrier synchronization. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010)*, pages 1–12, 2010.