UNIVERSIDAD DE ALMERÍA
ESCUELA SUPERIOR DE INGENIERÍA

PROYECTO FIN DE CARRERA
INGENIERÍA INFORMÁTICA

PARALELIZACIÓN DE ALGORITMOS DE CARACTERIZACIÓN DE IMÁGENES. COMPARACIÓN DE TÉCNICAS BASADAS EN GPU FRENTE A LAS LIMITADAS A CPU

Alumno: **Jose Manuel Arrufat González**

Director: **Francisco de Asís Guindos Roja**s

Almería, Noviembre de 2012

Agradecimientos

Es justo dedicar unas breves líneas a todas aquellas personas que me han acompañado durante este largo viaje, que cambia ahora de vagón para realizar el paso de la educación al mundo laboral. Porque se lo merecen.

Me gustaría comenzar agradeciendo todo su apoyo, confianza, aguante y cariño durante todo este tiempo a mis padres y hermana, porque sin ellos no estaría hoy donde estoy, ni sería la persona que he llegado a ser. Gracias.

A mi familia, que no es poca, por creer y confiar siempre en mí, dándome su apoyo desde el principio y animándome a continuar y avanzar siempre, incluso en los momentos en los que nadie, ni si quiera yo, confiaba en que las cosas pudieran salir bien.

A mi pareja, por entender mi situación y comprenderme, y por estar siempre ahí para apoyarme y levantarme cuando era necesario.

A compañeros y amigos, tanto de dentro como de fuera de la universidad, por aguantar nervios, largas temporadas de incomunicación, y charlas monotemáticas realmente aburridas. Mirando atrás, me doy cuenta de lo mucho que han hecho por mí, y de lo grandes que son. No puedo olvidarme de Israel de la Plata, gran amigo y compañero, con el que descubrí que había otra forma de hacer las cosas, y que, trabajando, podemos llegar lejos tan lejos como queramos, él ya lo ha conseguido. Junto a él, tuve también la suerte de compartir concursos y eventos con José Aguado, el cual se ha convertido con el tiempo en mi compañero de trabajo, y que ha tenido que soportar muchas de mis peculiaridades. Gracias por poner siempre la nota de razón y dedicar tu tiempo a escucharme y apoyarme. Además, querría acordarme de Manuel Godoy, compañero y amigo desde que comenzamos juntos la aventura de la Ingeniería Informática en 4º, gracias al cual he aprendido mucho. Gracias por confiar en mí.

A todos aquellos profesores que me han acompañado en este viaje, desde 1º hasta 5º, y durante el Máster de Informática Industrial cursado. En especial a los que me apoyaron y confiaron n mí.

A Francisco de Asís Guindos Rojas, por darme a conocer nuevas tecnologías, su tutela en este PFC, y sus grandes consejos.

A Fernando Reche Lorite, mi actual y primer jefe, porque gracias a su confianza en dos estudiantes con ganas de aprender y de emprender nuevos retos, he podido completar mi formación y he aprendido grandes lecciones.

A todos, gracias.

Índice

I Introducción

II Conceptos Teóricos

Parte I

Introducción

Capítulo 1

Descripción general

Es indudable el auge que están demostrando en los últimos tiempos las tecnologías de múltiples unidades de procesamiento en el ámbito tecnológico de consumo actual, abarcando desde los computadores personales a los dispositivos móviles.

Históricamente, el **paradigma de paralelización ha estado dominado por los procesadores de propósito general**, ya sea en forma de clúster de máquinas, múltiples procesadores colaborando en la misma máquina, o varios núcleos colaborando en un mismo procesador. Este paradigma esta cambiando en la actualidad gracias a la aparición de **nuevas tecnologías de procesamiento paralelo que buscan migrar del hardware convencional a los dispositivos gráficos** ([GLN], [SK], [YZP], [KW]) como arquitecturas de ejecución paralela.

Con este contexto actual se desarrolla un proyecto con fines analíticos que busca **explorar este nuevo paradigma computacional y enfrentarlo al procesamiento paralelo convencional**, en el marco del procesamiento y caracterización de imágenes segmentadas.

Para conseguir este objetivo, en primer lugar se realiza un estudio en profundidad de ambas tecnologías, centrando nuestra atención en dos ejemplos como son **CUDA** y **C#**, sobre las que se desarrollarán un conjunto de descriptores para imágenes segmentadas ([PF]).

Para poder determinar un **marco de usabilidad** de cada una de las tecnologías y establecer un **espacio de aplicación** se llevarán a cabo diferentes experimentos sobre distintas fuentes de datos previamente seleccionadas.

Capítulo 2

Objetivos

Este capitulo se dividirá en dos apartados, detallando los objetivos tanto generales como específicos del proyecto desarrollado.

2.1. Objetivos generales

El objetivo principal del presente proyecto es la **comparación de las nuevas tecnologías de programación paralela que corren sobre procesadores gráficos actuales con las que se ejecutan sobre los procesadores de propósito general**. Estas comparaciones se realizarán utilizando algoritmos de extracción de características de imágenes previamente sometidas a segmentación.

En base a ello, el primer objetivo es el **estudio de distintos algoritmos de extracción de características sobre imágenes segmentadas para evaluar su grado de paralelismo de forma teórica**, así como las distintas posibilidades de diseño paralelo observadas.

Por tanto, debemos realizar un proceso de estudio, caracterización y marcado de distintos algoritmos de extracción de características, atendiendo a los de más común uso y mayor grado de representatividad del segmento analizado, en base a un conjunto conocido de algoritmos. Cada uno de ellos debe ser estudiado con el fin de proponer distintas alternativas de implementación que extraigan todo el paralelismo posible de su definición.

El siguiente punto a llevar a cabo es el paso lógico tras el objetivo anterior. Se trata pues de la implementación de los distintos algoritmos seleccionados, haciendo especial hincapié en todas aquellas alternativas de implementación paralela diseñadas para cada algoritmo, posibilitando de esta manera un estudio en profundidad.

Posteriormente, a efectos de la comparativa, es necesario realizar un proceso de verificación y validación de los diseños teóricos implementados, ante un conjunto de imágenes objetivo previamente seleccionado.

Finalmente, se llevará a cabo una **discusión de la idoneidad de una u otra tecnología para los casos de estudio**, en base a los resultados obtenidos de los procesos de verificación y validación anteriores, generados sobre el conjunto de imágenes predefinido.

2.2. Objetivos específicos

Los objetivos específicos del presente proyecto se enmarcan en los objetivos generales anteriormente descritos.

Relativo al estudio teórico de los distintos candidatos a ser paralelizados, se establecen distintos objetivos específicos como son el **establecimiento de un conjunto básico de algoritmos de estudio** adecuados o el **desarrollo de estrategias de paralelización adecuadas** para cada algoritmo.

En lo relacionado al objetivo que resume el motivo del proyecto, la discusión de las tecnologías para los casos de estudio dados, se presentan distintos objetivos específicos. El primero de ellos es el **uso de tecnologías de carácter novedoso**, como serán en nuestro caso las relativas a plataformas de procesamiento gráfico. El segundo será el **establecimiento de plataformas de pruebas diferenciadas** entre si para dar una idea del alcance real de las tecnologías usadas.

Capítulo 3

Metodología

En este capítulo se indicará la metodología empleada durante el desarrollo del proyecto, enumerando las fases que han tenido lugar durante el mismo, comentando las técnicas usadas en cada etapa.

3.1. Estudio de la tecnología CUDA de NVIDIA y Parallel .NET de Microsoft

En esta primera etapa, se realizará un estudio exhaustivo de las peculiaridades y posibilidades de las dos tecnologías escogidas, que este caso son **CUDA** de n VIDIA y **Parallel .NET** de Microsoft, para la implementación de los algoritmos en su versión paralela.

Dentro de esta fase podemos, por tanto, englobar desde la búsqueda de información de las tecnologías mencionadas hasta la extracción de las necesidades y recursos básicos para la consecución del objetivo especificado anteriormente.

3.2. Estudio de los algoritmos y características a paralelizar

Una vez definidas las fuentes de información, así como realizado el estudio de las características de cada tecnología, se procede al **estudio, comprensión y elección de algoritmos** de extracción de características sobre imágenes segmentadas, basando el conjunto de estudio en los más usualmente utilizados y presentes en conjuntos de extracción automática altamente usados.

3.3. Implementación

En esta fase se aúnan los esfuerzos de estudio tanto de la tecnología como de los algoritmos analizados para realizar implementaciones de los mismos en distintas versiones. Durante este proceso se procede a **implementar de forma tanto secuencial como paralela** las distintas elecciones, además de establecer distintas versiones de implementación paralela para ciertos casos, con el fin de realizar un análisis más profundo de su comportamiento y viabilidad en ambas tecnologías.

3.4. Verificación de resultados

Una vez implementados, los algoritmos deben ser **verificados y validados** de forma concienzuda ante diferentes casos de entrada, con el fin de asegurar la veracidad y validez de los análisis posteriores que se realizarán sobre los mismos.

3.5. Análisis y estadísticas de los resultados obtenidos

Una vez verificados los algoritmos, se procede a realizar distintas ejecuciones ante casos de prueba previamente establecidos, a fin de realizar un **estudio comparativo de eficiencia y rendimiento** entre tecnologías y, de esta forma, evaluar la idoneidad de cada una de ellas ante distintas situaciones.

Capítulo 4

Herramientas de trabajo

A continuación se expondrán sin entrar en detalles tanto las tecnologías como herramientas software y hardware empleadas en el desarrollo de este proyecto. Los motivos de la elección tanto de las tecnologías hardware como software se presentarán en capítulos posteriores, a fin de desarrollar antes en detalle tanto las necesidades como el ámbito del proyecto para que sirvan de base para la elección

4.1. Tecnologías Software

Las tecnologías y herramientas software comentadas a continuación forman parte tanto del ámbito del libre uso, como son ciertos SDK's usados, como del conjunto de software bajo licencia ofrecido a la UAL para su uso gratuito en el entorno académico.

4.1.1. C# y Parallel .NET

El segmento de proyecto correspondiente a la implementación solo CPU se ha realizado sobre C# y su librería de procesamiento paralelo Parallel .NET sobre el Framework 4.0. C# ([CJM], [FR]) es un lenguaje de programación presente desde 2001 y desarrollado por **Microsoft**.

La biblioteca de manejo de procesamiento paralelo Parallel de .NET es la respuesta de Microsoft a la necesidad de una mejor manera de escritura de aplicaciones paralelas. Aunque .NET soporta procesamiento paralelo desde su versión 1.0, estamos en un marco de paralelización clásico, difícil de usar y donde se dedica la mayor parte del esfuerzo a como

manejar los distintos aspectos paralelos de la aplicación, en lugar de centrarse en lo que necesita ser paralelizado.

En el paradigma clásico, el programador crea hilos, que son el motor de ejecución, y somos responsables de su creación, asignación de trabajo y manejo de su existencia. En este modelo creamos un pequeño batallón para ejecutar nuestro programa, le damos órdenes a todos lo soldados, y nos manteneos vigilantes para estar seguros de que hacen lo que se les ordenó. Por el contrario, el modelo propuesto por Parallel .NET cuenta con la unidad básica de la tarea, que describe lo que se quiere hacer. Creamos entonces tareas para cada actividad que queremos realizar, y la biblioteca se encarga de crear hilos y tratar con ellos para que realicen en trabajo de nuestras tareas. Continuando con el símil anterior, entre nuestro batallón de procesamiento y nosotros colocamos una persona de confianza, a la que encargaremos un trabajo y que realizará por nosotros, en todo momento, las acciones de ordenar, vigilar y comprobar, dándonos capacidad de invertir nuestro tiempo en como abordar la tarea, frente al enfoque de como manejar las peculiaridades del tratamiento con hilos.

4.1.2. C++ y CUDA

En cuanto al procesamiento que hace uso del enfoque híbrido GPU y CPU, se ha optado por el lenguaje de programación CUDA acompañado de C++. CUDA son las siglas de Compute Unified Device Architecture y hace referencia a un lenguaje de programación desarrollado por **nVIDIA** y accesible por primera vez en 2007. En cuanto a C++, se postula como el compañero oficial propuesto por nVIDIA para CUDA y data de los años 80, en concreto de 1983, surgido como una extensión de C para el manejo de objetos.

CUDA no es solo un lenguaje de programación, es también una arquitectura gráfica. En generaciones anteriores se dividían los recursos computacionales entre procesamiento de pixeles y de vértices. La llegada de CUDA incluyó un camino unificado de procesamiento, permitiendo a cada una de las unidades aritmético lógicas del chip ser manejadas por una aplicación de computación de propósito general.

Estas unidades aritmético lógicas se construyeron siguiendo los requerimientos de simple precisión de aritmética sobre punto flotante del IEEE, en sus primeras versiones, además de añadir un conjunto de instrucciones de computación general. Además se capacitó a las unidades de ejecución para la lectura y escritura arbitraria en memoria, así como la posibilidad de acceso a un nuevo nivel de memoria denominado memoria caché, siguiendo una típica arquitectura de procesador típica.

De esta manera se concibe una nueva arquitectura gráfica que, sin perder ni un ápice de su capacidad en tareas de procesamiento gráfico tradicional, introduce nuevos conceptos para computación de propósito general real.

4.1.3. Segmentador de imágenes

Para el proceso de análisis de imágenes antes es necesario realizar un proceso de segmentación de las mismas. Para ello se hace uso de un segmentador proporcionado por el director de proyecto que hará sus funciones sobre las imágenes que servirán de fuente para los experimentos.

4.1.4. Entorno y herramientas de desarrollo

Para el desarrollo e implementación del proyecto sobre las tecnologías anteriormente mencionadas es necesario un entorno especialmente adaptado para ello. Ya que se busca realizar un estudio comparativo de rendimiento, es necesario unificar al mayor grado posible el proceso de generación de resultados, por lo que se decide usar el entorno **Visual Studio 2010** como entorno común de desarrollo para los dos grupos tecnologías de implementación presentados anteriormente.

Visual Studio (VS) es un entorno de desarrollo integrado (IDE) para sistemas operativos Windows desarrollado por Microsoft. Soporta varios lenguajes de programación de forma nativa como Visual C++, Visual C# y una amplia selección de lenguajes que operan sobre el Framework .NET. También ofrece soporte para C/C++ y, mediante complementos de nVIDIA, para CUDA.

Desde 2005 se ofrecen versiones Express de forma gratuita, pero para el desarrollo del presente proyecto se optó por usar la versión Ultimate proporcionada por Microsoft gracias al acuerdo con la UAL.

En consonancia con el IDE anterior se han usado las herramientas de nVIDIA para ayuda y mejora de la experiencia en su codificación junto a VS. En concreto se trata de la aplicación nVIDIA **NSight**, que añade potentes herramientas de depuración y análisis del rendimiento para optimizar el funcionamiento tanto de la CPU como de la GPU. Además, permite tareas de compilación en Host y ejecución en remoto para evaluar el rendimiento sobre sistemas externos con baja o ninguna carga gráfica.

4.2. Tecnologías Hardware

En cuanto a las tecnologías hardware usadas, debido a las necesidades del proyecto y a las tecnologías software mencionadas anteriormente se hace necesario con dos tipos de arquitecturas de procesamiento.

4.2.1. Arquitectura CPU

Para poder realizar una comparativa real, siguiendo el objetivo del proyecto de ejecución y prueba en entornos de ámbito doméstico, necesitamos un procesador de gama media de arquitectura multinúcleo. A tal efecto se dispondrá de dos tipos de arquitectura, CPU de sobremesa y de ordenador portátil.

4.2.2. Arquitectura GPU

De igual manera al anterior, se hace necesario un dispositivo multinúcleo o multiprocesador de tipo GPU para realizar las comparativas. Del mismo modo se hará uso de arquitecturas preparadas para funcionamiento en plataformas de sobremesa y en plataformas móviles.

Capítulo 5

Planificación

Debido al carácter del proyecto, donde predomina el marcado carácter investigador, el proceso de planificación resulta harto complicado. Estamos sujetos a variaciones continuas de los plazos y el contenido, fruto de los resultados parciales arrojados por los procesos de búsqueda de información, implementación parcial de fases del proyecto y aparición de nuevas estrategias de paralelización fruto de nuevas investigaciones. Por tanto, aunque se busca establecer una planificación típica modular, en la práctica las distintas fases y etapas del proyecto corren de forma simultánea, apoyándose unas sobre otras, y generando nuevas líneas de trabajo de forma continua.

A continuación se establece una planificación por etapas de las distintas fases a realizar durante el desarrollo del presente proyecto. En primer lugar mostraremos un diagrama de Gantt general para completarlo posteriormente con una breve descripción temporal y conceptual de las distintas fases completadas.

5.1. Diagrama de Gantt

Aquí nos encontramos pues con el diagrama de Gantt del proyecto, realizado sobre la herramienta **Microsoft Proyect 2010**, proporcionada como las anteriores gracias al acuerdo entre la UAL y la citada compañía.

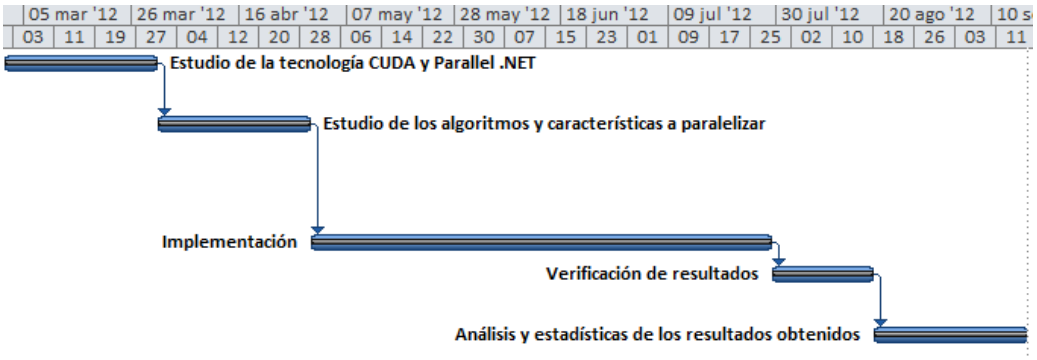


Ilustración 1 - Diagrama de Gantt

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		Estudio de la tecnología CUDA y Parallel .NET	30 días	jue 01/03/12	vie 30/03/12	
2		Estudio de los algoritmos y características a paralelizar	30 días	sáb 31/03/12	dom 29/04/12	1
3		Implementación	90 días	lun 30/04/12	sáb 28/07/12	2
4		Verificación de resultados	20 días	dom 29/07/12	vie 17/08/12	3
5		Análisis y estadísticas de los resultados obtenidos	30 días	sáb 18/08/12	dom 16/09/12	4

Ilustración 5.1 - Detalle Planificación Microsoft Project

Como se puede observar la duración total del proyecto se enmarca en unos 200 días, aproximadamente 7 meses, durante los cuales se van sucediendo las distintas etapas. Se ha estimado que para el comienzo de una etapa la anterior debe estar completamente finalizada y aunque en teoría y práctica es necesario que esto sea así, se producen durante el desarrollo del proyecto flujos de retroalimentación continuos entre las distintas fases mencionadas.

5.2. Estudio de la tecnología CUDA y Parallel .NET

La duración de esta etapa se establece en unos 30 días aproximadamente, durante los cuales se llevan a cabo tareas de búsqueda de información de ambas tecnologías citadas. Dentro de esta información se busca establecer tanto el patrón de viabilidad de uso de ambas en el ámbito general, como las posibilidades reales en entornos y problemas específicos. Además de esto, se establece una primera línea de información sobre la aplicación de ambas tecnologías, en especial la orientada a procesadores gráficos, en el ámbito de las imágenes segmentadas, y más concretamente en la extracción de características.

Durante este período se establece también el ámbito de formación en las tecnologías comentadas, cuya importancia radica en la consecución de la formación necesario para la elaboración de las siguientes fases del presente proyecto, como las relativas al trabajo fin de

master realizado de forma paralela a este ([A], [AAL]), cuyo desarrollo hace uso tanto de la paralelización CPU como GPU estudiadas aquí.

5.3. Estudio de los algoritmos y características a paralelizar

El estudio de los algoritmos conlleva un tiempo de proyecto de otros 30 días. Durante el mismo, se establecen los marcos de elección de los distintos algoritmos susceptibles de ser optimizados mediante procesamiento paralelo. Para ello se lleva a cabo una búsqueda intensiva sobre su idoneidad y uso por la comunidad, centrando la atención en ciertas plataformas como Matlab y sus paquetes de tratamiento digital de imágenes ([PF], [GW]).

5.4. Implementación

Durante la etapa de implementación se consume un tiempo aproximado de 90 días, durante los cuales se llevan a cabo todo tipo de desarrollos de los algoritmos seleccionados, implementando distintas versiones de los mismos sobre las distintas plataformas escogidas, a fin de establecer un criterio comparativo suficiente en las siguientes fases.

Se hace especial hincapié en la diversidad de versiones sobre ciertos algoritmos debido a la naturaleza del proyecto, que busca establecer rangos de trabajo y viabilidad sobre la optimización en ambas plataformas, a fin de obtener una visión clara y real de ambas a su finalización.

5.5. Verificación de resultados

La fase de verificación de resultados conlleva un esfuerzo cuantificado en 20 días aproximadamente, tiempo durante el que se llevan a cabo intensas pruebas de carga sobre los distintos algoritmos implementados, a fin de establecer la corrección en los mismos y poder avanzar a la siguiente etapa.

5.6. Análisis y estadísticas de los resultados obtenidos

La fase final es la correspondiente al análisis y estudio de los resultados obtenidos en la fase anterior, así como la generación de nuevas baterías de pruebas a fin de estudiar puntos o comportamientos considerados interesantes para la obtención de una conclusión fiable.

Esta etapa consume un tiempo aproximado de 30 días, y engloba tanto lo anterior como la finalización de la memoria explicativa del proyecto, dando buena cuenta de todas las resoluciones aportadas por dicho análisis.

Parte II

Conceptos teóricos

Capítulo 6

CUDA

Para una introducción correcta al paradigma de procesamiento GPU se comenzará por una breve introducción, seguido de una exposición desde sus comienzos hasta llegar al estado actual, y así sentar las bases de esta novedosa tecnología en cuanto a su aplicación y explotación. Nos centraremos en la base que nos proporcionan las referencias [KW] y [SK].

6.1. Introducción

Para los programadores **CUDA** y **OpenCL** las unidades de procesamiento gráfico (GPU) son procesadores de computación numérica paralela masiva codificados en C con extensiones. No es necesario entender los algoritmos gráficos o su terminología para ser capaz de programar estos procesadores, como sucedía anteriormente. Aun así, entendiendo y comprendiendo la herencia de los mismos podemos percibir de mejor manera tanto los puntos fuertes como débiles de estos procesadores frente a las principales tecnologías de cómputo. En particular, su historia ayuda a clarificar las principales decisiones de diseño de la arquitectura de las GPU programables modernas: multihilo masivo, memorias cache de bajo tamaño en comparación con las CPU, e interfaz de memoria centrada en el ancho de banda.

6.2. Historia

6.2.1. Evolución del camino de datos gráfico

El camino de datos en hardware gráfico 3D evoluciona desde los caros sistemas de principios de los 80 a las pequeñas estaciones de trabajo a finales de los 90. Durante este periodo, el coste de los mismos cae más de un 99%, mientras que el rendimiento aumenta de 50 millones de pixeles por segundo a más de 1000 millones de pixeles por segundo y de 100 mil vértices a 10 millones de vértices por segundo. Aunque estos avances tienen mucho que ver con la reducción implacable del tamaño de los semiconductores, también son el resultado de innovaciones continuas en los algoritmos gráficos y el diseño del hardware, que da forma a las capacidades hardware nativas e las GPU modernas.

Los avances gráficos más importantes han sido propiciados por la demanda del mercado de alta calidad y gráficos en tiempo real en aplicaciones de propósito general. En un juego, por ejemplo, se necesita mostrar escenas cada vez más complejas en una resolución cada vez mayor a una tasa de al menos 60 frames por segundo. El resultado es que durante los últimos 30 años la arquitectura gráfica ha evolucionado de ser un simple camino de datos para dibujar diagramas de malla de alambre a convertirse en un diseño altamente paralelo consistente en multitud de caminos de datos capaces de mostrar escenas de contenido 3d interactivo.

6.2.1.1. La era de los caminos de datos de función prefijada

Desde el comienzo de los 80 a la finalización de los 90 el hardware gráfico contaba con caminos de datos de función prefijada que podían ser configurados pero no programados. En la misma época, las bibliotecas de interfaz de programación de aplicaciones (API) gráficas se hizo popular. Una API es **DirectX**, propietaria de Microsoft, cuyo componente **Direct3D** provee una interfaz de funciones para el procesador gráfico. La otra API más usada es **OpenGL**, un estándar abierto soportado por múltiples compañías y muy popular en aplicaciones profesionales. Esta era que comentamos se corresponde con las primeras siete generaciones de DirectX.

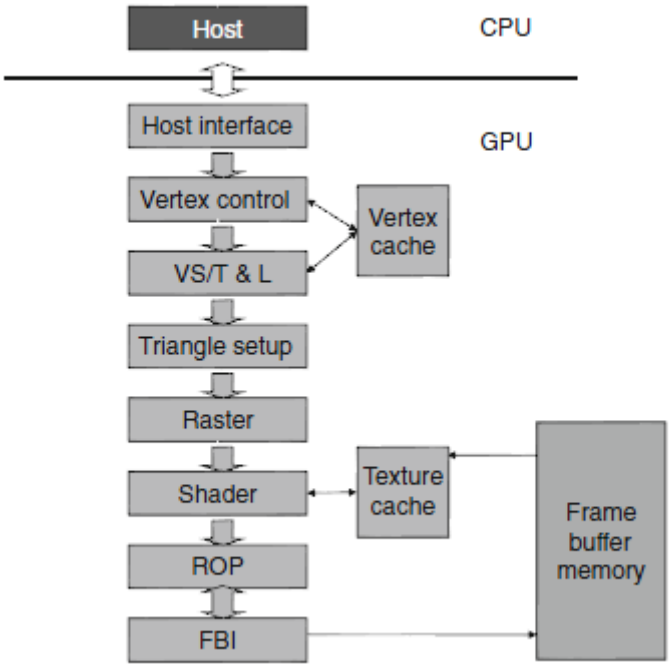


Ilustración 6.1 - Camino de datos nVIDIA GeForce

En la figura 6.1 podemos ver un ejemplo de camino de datos de los primeros procesadores gráficos **GeForce** de nVIDIA. La interface host recibe comandos gráficos y datos desde la CPU, generalmente por parte de programas de aplicación por medio de una API. Una vez ahí realiza el recorrido configurado según los parámetros recibidos y se ejecuta el procesamiento gráfico necesario.

Durante dos décadas, cada generación de hardware y su correspondiente API trajo mejoras continuas sobre varias etapas del camino de datos. Aunque cada generación introdujese recursos hardware adicionales y mayor poder de configuración de las etapas del camino de datos, los desarrolladores fueron haciéndose más sofisticados y requiriendo nuevas características que podrían ser ofrecidas razonablemente con funciones prefijadas. El siguiente paso es obviamente convertir estas etapas del camino de datos en procesadores programables.

6.2.1.2. Evolución de plataformas de gráficos en tiempo real programables

En 2001, la plataforma de **nVIDIA GeForce 3** dio el primer paso hacia la programación real de los procesadores. Expuso al desarrollador lo que habían sido el conjunto de instrucciones internas del motor de vértices en punto flotante, coincidiendo con la liberación de las extensiones sobre el procesador de vértices de **DirectX8** y **OpenGL**. Las siguientes GPUs, junto a **DirectX 9**, extendieron la capacidad de programación y manejo de punto flotante a la etapa completa del procesador de pixeles e hizo que las texturas fuesen accesibles desde esta

etapa. La **ATI Radeon 9700**, presentada en 2002, incluyó procesador de pixeles de 24 bits en punto flotante programable, programable desde DirectX9 y OpenGL. **GeForce FX** incluyo procesadores de pixeles de 32 bits en punto flotante de la misma manera. Estos procesadores fueron la base de un esfuerzo general por unificar la funcionalidad de las etapas del camino de datos de cara al programador de aplicaciones. Las líneas de **GeForce 6800** y **7800** fueron construidas con procesadores separados encargados de los vértices y los pixeles. Fue la **Xbox 360** la que introdujo el primer procesador gráfico unificado en 2005, permitiendo que las operaciones sobre pixeles y vértices se ejecutasen en el mismo procesador.

En los caminos de datos gráficos, algunas etapas manejaban de forma excelente las operaciones en punto flotante en datos completamente independientes, como la transformación de posiciones de los vértices del triangulo o la generación de pixeles coloreados. Esta independencia de datos como la característica dominante de la aplicación es la diferencia clave entre el diseño base de las GPUs y CPUs. Un frame, renderizado en 1/60 de segundo, puede tener un millón de triángulos y seis millones de pixeles. La oportunidad de usar paralelismo hardware para explotar esta independencia de datos es tremenda.

Las funciones específicas ejecutadas en unas pocas etapas del camino de datos gráfico varían dependiendo de los algoritmos usados, variación que motivó a los diseñadores de hardware a hacer estas etapas programables. Dos etapas en particular destacaron, el procesamiento de vértices y el procesamiento de pixeles. Los programas de procesamiento de vértices mapean las posiciones de los vértices del triángulo en la pantalla, alterando su posición, color u orientación. En cuanto al procesamiento de pixeles, el programa calcula los valores de rojo, verde, azul y alpha (RGBA) para construir la imagen en la posición del pixel. Dado que se trabaja con datos independientes entre ellos, se producen resultados independientes, pueden correr en paralelo, y esta propiedad motiva el diseño de caminos de datos programables convertidos en procesadores de computación paralela masiva.

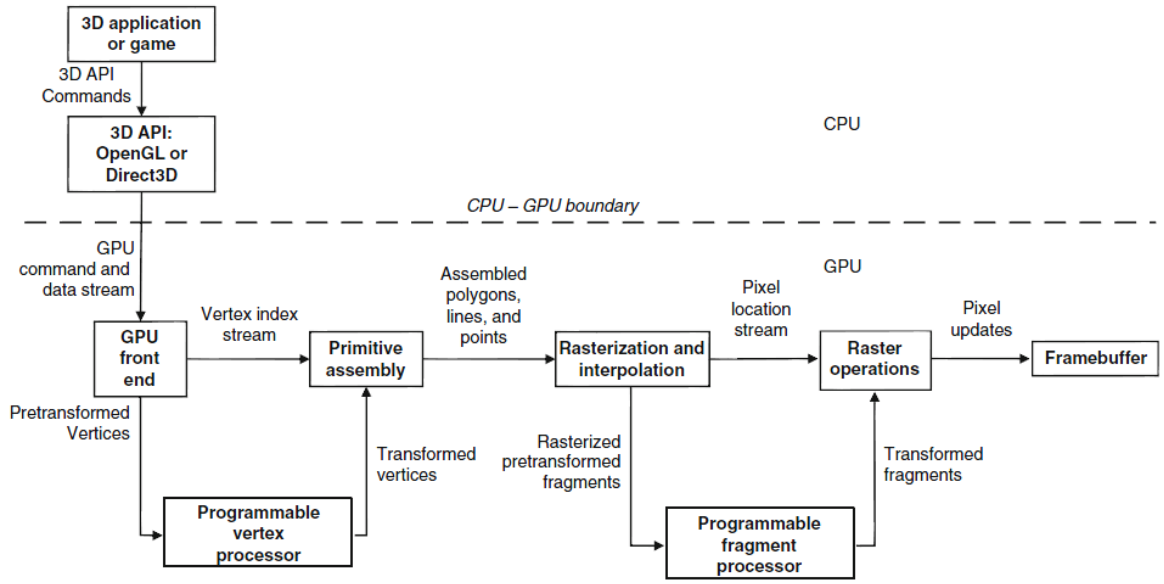


Ilustración 6.2 – Procesador de vértices y procesador de pixeles fragmentado separados

En la figura 6.2 podemos ver un ejemplo de arquitectura que usa un procesador de vértices y un procesador fragmentado de pixeles de forma separada. Entre las etapas de pixel y vértice existen docenas de etapas de funciones prefijadas que realizan tareas bien definidas de forma mucho más eficiente que un procesador programable. Juntos, la mezcla de procesadores programables y funciones están ingeniados para balancear el rendimiento extremo con el control del usuario sobre los algoritmos del procesador gráfico.

Los algoritmos de render comunes realizan una pasada sobre primitivas de entrada y acceden a recursos en memoria de una manera altamente coherente. Estos algoritmos tienden a acceder de forma simultánea a zonas de memoria contiguas, como todos los triángulos o todos los pixeles en un vecindario. Como resultado, estos algoritmos muestran una eficiencia excelente en la utilización del ancho de banda de memoria y son altamente insensibles a la latencia de memoria. Combinado con la carga del procesamiento de pixel que es normalmente limitado computacionalmente, estas características han guiado a las GPUs hacia un camino de evolución distinto al de las CPUs. En particular, mientras que el área muerta de la CPU esta dominada por las memorias caché, las GPUs están dominadas por los caminos de datos en punto flotante y la lógica de funciones predefinida. Las interfaces de memoria GPU enfatizan el ancho de banda sobre la latencia, ya que la latencia puede ser fácilmente oculta con ejecuciones paralelas masivas. De hecho, el ancho de banda es normalmente muy superior al de CPU, superando en los últimos modelos los 100GB/s.

6.2.1.3. Gráficos unificados y procesadores de computación

La GPU de la GeForce 8800, introducida en 2006, mapeaba las etapas programables en un array de procesadores unificados; el camino de datos lógico es físicamente un camino de recurrencia que visita estos procesadores tres veces, con mucha lógica de funciones hardware prediseñadas entre visitas, lo cual podemos ver en la figura 6.3.

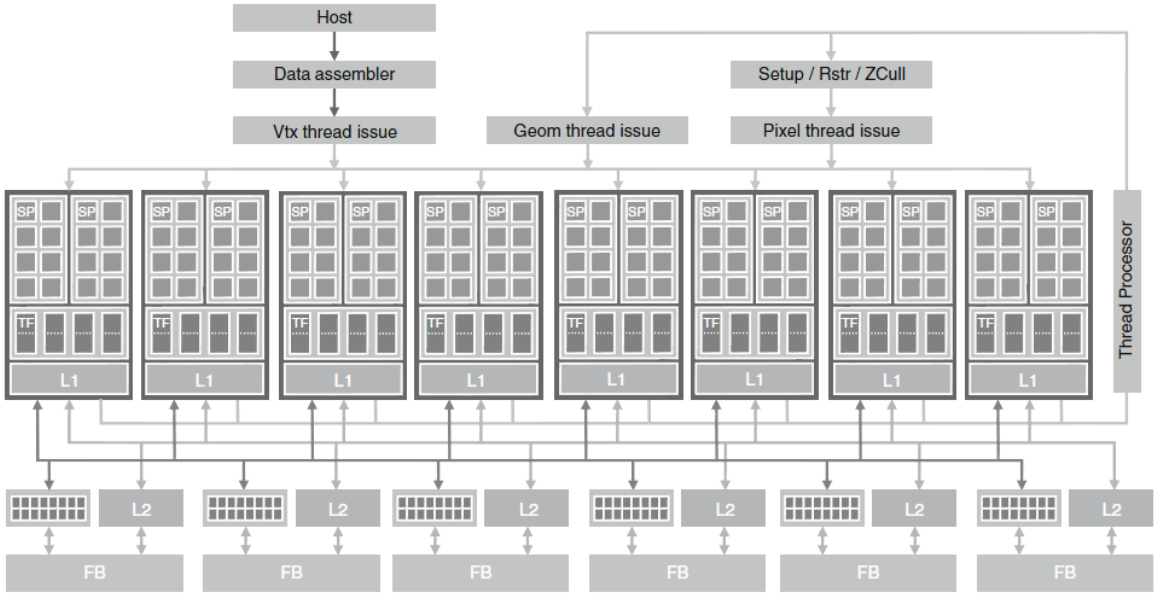


Ilustración 6.3 – Procesador unificado programable en array de GeForce 8800 GTX

Este hardware se corresponde con la API de DirectX 10, donde la funcionalidad del procesamiento de vértices y pixeles ha sido hecha idéntica para el programador. Ahora los desarrolladores cuentan con algoritmos de procesamiento más sofisticados, y esto motiva un fuerte aumento de la tasa de operaciones disponibles.

6.2.1.4. GPGPU, un paso intermedio

Mientras el hardware gráfico evoluciona hacia la unificación de procesadores, cada vez se parecía más a computadores paralelos de altas prestaciones. Desde DirectX 9, algunos investigadores toman conciencia del aumento de rendimiento de las GPUs y comienzan a explorar sus capacidades para resolver problemas intensivos en cómputo. Para poder realizar esto, un desarrollador tenía que convertir su problema en primitivas de procesamiento gráfico para que la computación fuese posible lanzarla desde OpenGL o DirectX. Para ejecutar varias instancias simultáneas, por ejemplo, debía escribirse como un procesamiento de pixeles. Los datos de entrada debían guardarse como imágenes de texturas y emitidas a la GPU como triángulos. La salida consistía en un conjunto de pixeles generados por las operaciones sobre ellos.

Por tanto la arquitectura de procesadores como un array se presentó demasiado restrictiva para aplicaciones genéricas. En particular, la salida de los programas son pixeles simples cuya localización en memoria es predeterminada, ya que el array de procesadores ha sido diseñado con capacidades de lectura y escritura en memoria muy restrictivas. En la figura 6.4 se puede ver la limitación de acceso a memoria de os primeros procesadores en array programables.

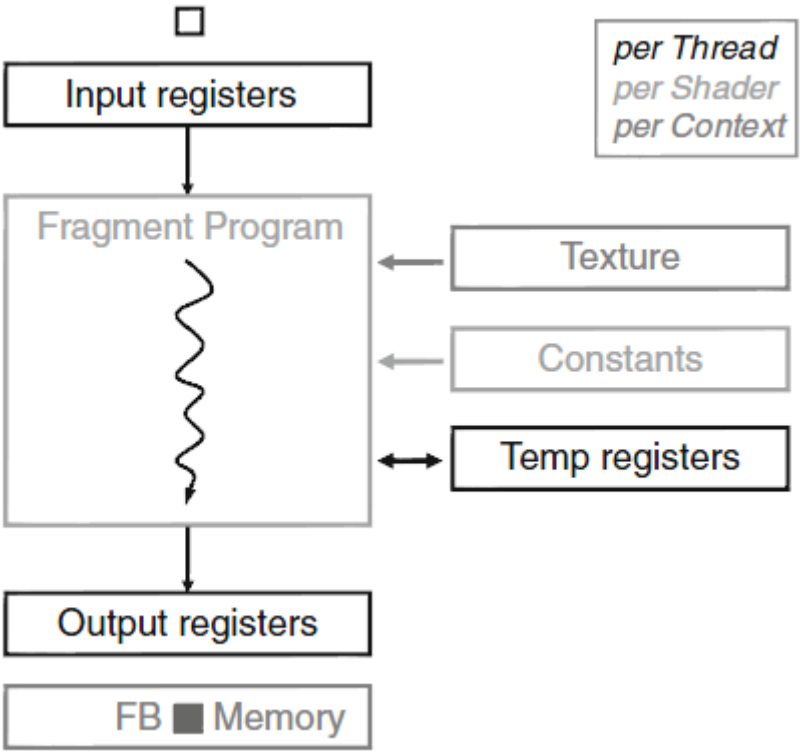


Ilustración 6.4 – Capacidades restringidas de entrada y salida

Además, la única manera de que un resultado pasase de una ejecución a otra era escribiendo todos los resultados paralelos en un buffer de pixeles del frame, usar después este buffer como una textura de entrada para la siguiente etapa de computación.

6.2.2. Computación GPU

Mientras desarrollaban la arquitectura GPU **Tesla**, nVIDIA se dio cuenta de que su potencial podría ser mejor aprovechado si los desarrolladores pudiesen pensar en la GPU como si fuese un procesador convencional.

Para la generación gráfica de **DirectX 10** había comenzado a trabajar en un procesador de enteros y punto flotante de alta eficiencia que pudiese correr una gran variedad de trabajos para soportar el camino de datos gráfico. Además, los procesadores de vértices y pixeles se convirtieron en procesadores totalmente programables con una amplia memoria de instrucciones, cache de instrucciones y control lógico de secuencia de instrucciones. El coste de estos recursos hardware adicionales se redujo introduciendo muchos procesadores para compartir entre ellos la cache de instrucciones y la lógica de control. nVIDIA añadió además instrucciones de escritura y lectura de memoria con capacidad de direccionamiento aleatorio para soportar os requerimientos de programas C. Para aplicaciones de propósito general, la arquitectura Tesla introduce un modelo de programación paralela con jerarquía de hilos paralelos, barreras de sincronización y operaciones atómicas. Además también se desarrolló el compilador de **CUDA C/C++**, bibliotecas y software de tiempo de ejecución para que los

desarrolladores pudiesen acceder al nuevo modelo de computación paralela y desarrollasen aplicaciones. **Ya no es necesario, por tanto usar las API de DirectX u OpenGL**.

6.2.2.1. GPUs escalables

La escalabilidad ha sido una característica atractiva de los sistemas gráficos desde el principio. Al comienzo, los sistemas gráficos de trabajo daban a los clientes la elección del poder de trabajo sobre pixeles variando el número placas de procesamiento de pixeles instaladas. Hasta mediados de los 90 la escalabilidad en los gráficos de ordenadores personales era inexistente. En 1998, 3dfx introdujo el escalado multiplaca con la interfaz SLI, lo que supone el inicio de la escalabilidad en gráficos de sobremesa. Tras adquirir 3dfx en 2001, nVIDIA continuó con la idea de usar SLI, proporcionando el salto necesario para realizar escalabilidad de forma transparente tanto al usuario como al desarrollador.

Con el cambio a la trayectoria multinúcleo, las CPUs aumentan la escalabilidad añadiendo núcleos, en lugar de solo aumentar la velocidad de un núcleo. Los programadores se ven forzados a encontrar segmentos de paralelismo en 4 u 8 núcleos, actualmente en consumo de sobremesa, para obtener el máximo rendimiento de estos procesadores. Por tanto las aplicaciones deben ser rescritas a menudo para introducir más tareas paralelas para cada aparición de núcleos en los procesadores. En contraste, el modelo GPU de computación multihilo fomenta el uso masivo de paralelismo de datos de grano fino en CUDA. Un soporte eficiente de gestión de hilos en la GPU permite a las aplicaciones exponer una carga paralela mucho mayor de los recursos de ejecución hardware disponibles, que explotan más del paralelismo expuesto para obtener mayor rendimiento. Este es el modelo de programación paralela para gráficos y computación paralela transparente y escalable. Los gráficos o programas CUDA son escritos una vez y se ejecutan en una GPU con cualquier número de núcleos sin necesidad de adaptación.

6.3. Cuando y porqué CUDA

Como ya hemos visto, CUDA es una plataforma de computación paralela y un modelo de programación llevado a cabo por nVIDIA. Engloba desde el hardware y su diseño hasta el lenguaje de programación y sus diversas herramientas de compilación y explotación.

Para un programador CUDA, el sistema de computación consiste en un host, con la arquitectura tradicional CPU, y uno o más dispositivos, que son procesadores paralelos masivos equipados con un alto número de unidades de ejecución aritmética.

En base a lo anterior, podemos definir el cuando CUDA. Si contamos con un sistema compuesta por nuestra unidad central de proceso, a la que se le suman dispositivos de cálculo paralelo masivo, que además proporcionan herramientas de uso para el desarrollador, es directamente lógico pensar en usar CUDA en aplicaciones de propósito general donde realizamos tareas intensivas en cálculo o intensivas iterativas. Donde usaremos procesamiento GPU entonces, pues será usado en aquellas secciones de nuestro programa en las que observemos capacidades paralelas e independencia de datos.

Conocemos por tanto qué es CUDA, que nos puede ofrecer y cuando podemos aplicarlo, ahora la pregunta es porqué aplicarlo. Hay muchos ejemplos válidos que nos pueden ayudar a entender por qué, pero en líneas generales podemos hablar de la búsqueda de la eficiencia y le rendimiento. Cuando nos proponemos a realizar el diseño e implementación de una aplicación rara vez se tiene en cuenta el efecto sobre el hardware y su castigo constante a la unidad de proceso. Por tanto la respuesta a nuestra pregunta tiene dos posibles sentidos. En primer lugar, el sentido del rendimiento, buscamos acelerar al máximo ciertas tareas o dar una respuesta en tiempo determinado, para lo que necesitamos una capacidad de la que nos disponemos en un ordenador personal en forma de múltiples procesadores convencionales. Por otro loado tenemos la eficiencia, buscamos realizar tareas de forma eficiente y transparente al usuario, lo que incluye dar una experiencia de uso fluida y dinámica. Por tanto hay tareas intensas en computación que necesitamos alejar del motor del ordenador personal, para evitar congestiones indeseadas, y usamos un dispositivo que tenemos al alcance y que, de forma convencional, se desperdicia su capacidad en alto grado.

En otros casos, sirve para llevar a cabo tareas fuera del alcance de un computador personal, incluso de un computador de altas prestaciones de gama media. Como ejemplo podemos citar [Referencia Trabajo Fin de Master y artículo] donde se consigue traer de vuelta un método de computación altamente seguro en el campo de la criptografía abandonado 20 años atrás debido a la imposibilidad de computar su resultado en un tiempo aceptable, solo estando al alcance de ciertas arquitecturas de altas prestaciones no fácilmente escalables.

6.4. Cuando no CUDA

También hay que destacar las limitaciones de la tecnología, ya que no todo es aplicable en según que contextos.

Si tratamos con datos con alta dependencia, operaciones no intensivas, o intensivas sobre este tipo de datos el acercamiento a esta tecnología es altamente difícil y costoso, añadiendo que con estas características su uso queda totalmente descartado. En este caso lo que buscamos es una arquitectura con núcleos que procesen a alta velocidad, de los cuales probablemente usemos nada más que uno y necesitaremos todo lo que pueda darnos. A este respecto las arquitecturas mono procesador tradicionales o multinúcleo son las idóneas para nosotros.

Igualmente, existen otras limitaciones básicas. CUDA no admite de ninguna manera procesamiento recurrente de ningún tipo, y cualquier esfuerzo en este sentido ha de ser rediseñado totalmente para poder adaptarse a la plataforma, en [referencia tfm y artículo] podemos encontrar un ejemplo de todo esto, con el costo que ello conlleva.

Igualmente, existe otra salvedad importante, aunque está siendo corregida en estos momentos, y es el uso de aritmética de enteros grandes. Si bien la aritmética en punto flotante ha ido mejorando en la plataforma hasta legar al nivel de arquitecturas CPU la aritmética de enteros grandes es un puto en contra. Actualmente existe una iniciativa para transportar **GNU MP** al entorno CUDA y, de hecho, está desarrollada y liberada, aunque aún en proceso de documentación. Esto es todo lo que resta a la plataforma para acerarse a mundo de la computación paralela con paso firme y seguro.

6.5. Alternativas

Existen otras alternativas a la plataforma CUDA, de hecho existen dos alternativas, las denominadas **OpenCL** y **ATI Stream**. La primera propone un lenguaje multiplataforma al más puro estilo Java, pero ofrece un pobre rendimiento en comparación con las otras dos mencionadas, o legando sacar todo el jugo y rendimiento que las plataformas gráficas nos ofrecen. En cuanto a la alternativa de ATI, competidora de nVIDIA, presenta mejor rendimiento que OpenCL pero no llega a los límites de madurez y eficiencia ofrecidos por CUDA por tanto se descartan de plano para una ejecución de alto rendimiento

Otro factor importante es que estamos hablando de plataformas gráficas de juego, sobre las que impera el monopolio nVIDIA con lo cual no se puede obviar como base el dispositivo gráfico más presente en ordenador personales si se busca una solución para el publico en general.

Capítulo 7

Parallel .NET

En este capitulo se introducirá de forma concisa la tecnología Parallel de .NET y sus aspectos más destacados, así como el porqué de su elección ante otras alternativas también válidas. Nos centraremos en las referencias [CJM] y [FR] para este capítulo.

7.1. Introducción

Los equipos de varios procesadores se están convirtiendo ahora en los equipos estándar mientras que se reducen los aumentos de velocidad de los equipos con un único procesador. La clave para las mejoras de rendimiento es, por lo tanto, ejecutar un programa en varios procesadores en paralelo., Lamentablemente, todavía es muy difícil escribir algoritmos que realmente saquen partido de estos procesadores múltiples. De hecho, la mayoría de aplicaciones usan sólo un único núcleo y no perciben mejoras de velocidad cuando se ejecutan en un equipo multinúcleo. Por tanto, debemos escribir nuestros programas de una manera nueva.

7.2. Qué es Parallel .NET

Parallel .NET es un nuevo paradigma de programación paralelo más conocido como **TPL** o **Task Parallel Library**, Biblioteca de tareas paralelas, diseñada por **Microsoft** sobre su **Framework de .NET** para facilitar la escritura de código administrado que pueden usar automáticamente los procesadores múltiples. Con esta biblioteca, se puede expresar de forma conveniente el paralelismo posible en el código secuencial existente, donde las tareas paralelas expuestas se ejecutarán simultáneamente en todos los procesadores disponibles.

Generalmente, esto tiene como resultado unas aceleraciones significativas. Este es el comienzo de la nueva manera de escribir programas paralelos.

Se trata de un componente integrado en el **.NET Framework 4.0**, pero usable desde la versión 3.5, capaz de gestionar de forma totalmente autónoma, si así se desea, el manejo, configuración y supervisión de los hilos necesarios para ejecutar una tarea concreta.

Su uso es realmente abrumador por sencillez y potencia. Por ejemplo, supongamos que contamos con un bucle de repetición que eleva al cuadrado los elementos de una matriz como el del fragmento 7.1.

```
for (int i = 0; i < 100; i++) {
  a[i] = a[i]*a[i];
}
```

Fragmento 7.1 – Bucle de repetición secuencial

Dado que las iteraciones son independientes entre sí, es decir, las iteraciones posteriores no leen las actualizaciones de estada realizadas por iteraciones anteriores, puede usarse TPL para expresar el posible paralelismo con una llamada al método For paralelo, tal como se en el fragmento 7.2.

```
Parallel.For(0, 100, delegate(int i) {
  a[i] = a[i]*a[i];
});
```

Fragmento 7.2 – Bucle de repetición paralelo basado en TPL

Como vemos, es una simple expresión estática con tres argumentos, donde el último es una expresión del delegado, que captura el cuerpo del bucle sin modificar de la figura, lo que hace que sea especialmente fácil experimentar con la introducción de la función de simultaneidad en un programa.

La biblioteca contiene algoritmos más sofisticados para facilitar una **distribución dinámica** del trabajo y se adapta automáticamente a la **carga de trabajo** y al equipo en particular. Mientras tanto, los primitivos de la biblioteca sólo expresan el paralelismo potencial, pero no lo garantizan. Por ejemplo, en un equipo de un único procesador, los bucles de repetición paralelos se ejecutan de manera secuencial, coincidiendo casi con el rendimiento de código estrictamente secuencial. Sin embargo, en un equipo de doble núcleo, la biblioteca usa dos subprocesos de trabajo para ejecutar el bucle en paralelo, en función de la carga de trabajo y de la configuración. Esto significa que hoy en día es posible insertar paralelismo en el código y las aplicaciones usarán varios procesadores automáticamente cuando estos se encuentren disponibles. Al mismo tempo, el código todavía tendrá un rendimiento adecuado en equipos más antiguos con un único procesador.

Si recordamos lo expuesto en el capitulo anterior, la tecnología CUDA se jactaba de ofrecer al desarrollador el nivel de paralelismo y adaptación que faltaba frente al desarrollo paralelo tradicional, siendo esta una forma totalmente escalable, adaptable y funcional sin preocupaciones sobre la arquitectura base. Como podemos ver, la alternativa presentada por

Microsoft **potencia todas esas características introducidas por CUDA**, desde el manejo de hilos, a la gestión de trabajo y carga de los elementos de computación, hasta el nivel de ejecución por tareas como interfaz de abstracción del nivel más básico de manejo de recursos.

Lamentablemente, todo tiene sus límites, y la biblioteca no facilita la sincronización correcta de código paralelo que usa memoria compartida. Es todavía responsabilidad del programador asegurarse de que haya código que pueda ejecutarse en paralelo sin riesgos. Otros mecanismos, tales como bloqueos, todavía son necesarios para proteger las modificaciones simultáneas en la memoria compartida. Sin embargo, TPL si ofrece algunas abstracciones que facilitan la sincronización de manera eficiente y segura.

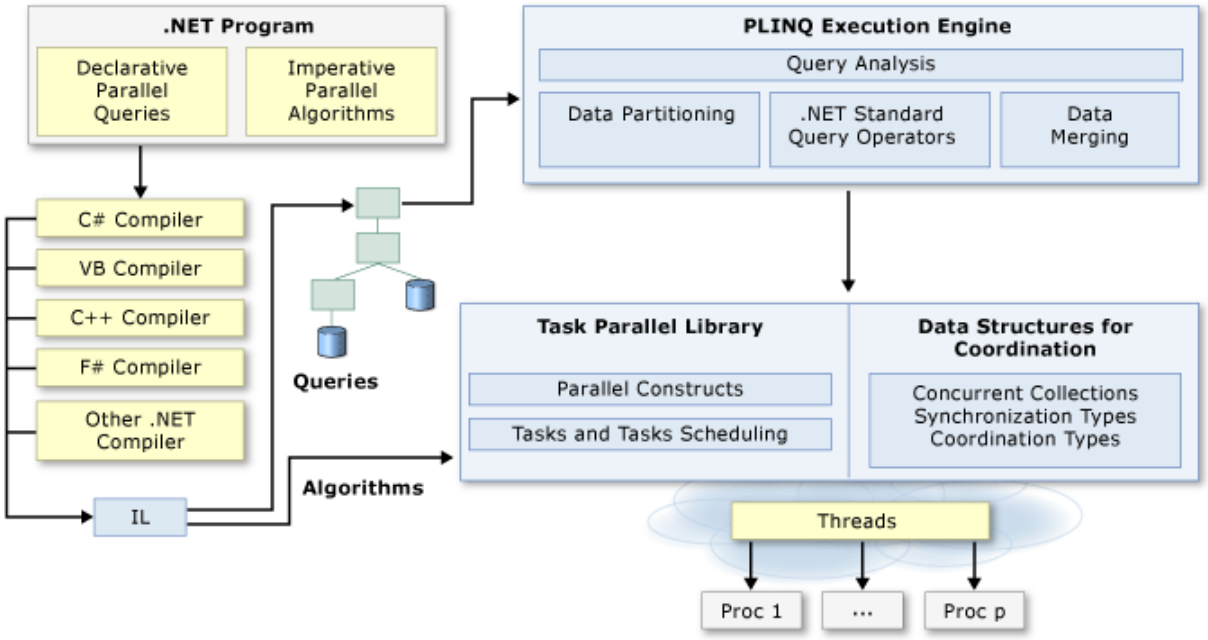


Ilustración 7.1 – Estructura aplicación .NET Paralela

A fin de clarificar un poco más el uso de esta tecnología, la figura 7.1 presenta una vista de alto nivel arquitectura de programación paralela existente en el Framework 4 de .NET, el que se va a usar en la implementación del presente proyecto. En nuestro caso, contaremos con un programa .NET, como se puede observar en la parte superior izquierda. Haciendo uso del compilador de C# se generan las instrucciones pertinentes al desarrollo paralelo de nuestra aplicación. En este caso no habrá ninguno tipo de tratamiento de datos sobre bases de datos, por lo que no se hace uso de LINQ, o su interfaz paralela PLINQ. Estas instrucciones son las que la TPL y sus estructuras de datos de coordinación manejarán aislando al programador, si él lo desea, de cualquier tratamiento pormenorizado de los hilos que llevarán a cabo la acción programada. Una forma más simple de lo anterior puede observarse en la figura 7.2.

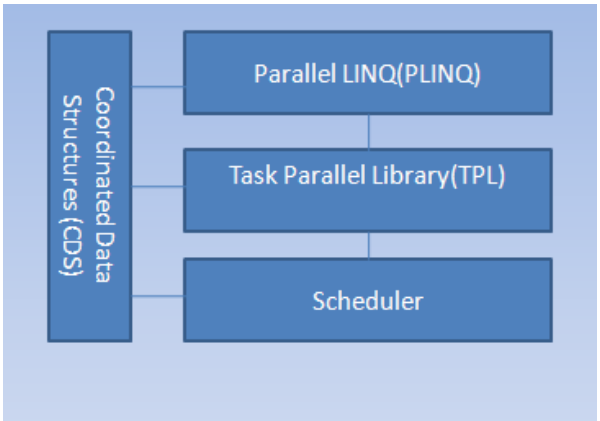


Ilustración 7.2 – Estructura de alto nivel del componente paralelo del Framework de .NET

7.3. Cuando y porqué Parallel .NET

En base a todo lo expuesto anteriormente, queda bastante definido el ámbito de actuación de Parallel .NET. El cuando usarlo es directamente fruto de esas características anteriores, y se resume en una frase: cuando buscamos paralelizar para obtener rendimiento sin perder tiempo en bajar al nivel de gestión de recursos. Por tanto, si contamos con un problema a resolver, donde la plataforma .NET es una opción de desarrollo por supuesto, con indicios de posible actuación paralela, y queremos que sea altamente portable, sobre plataformas que soporten .NET, y escalable en cualquier arquitectura, esta es nuestra opción sin lugar a dudas. Si por otro lado buscamos mejorar nuestra aplicación .NET añadiendo algún tipo de paralelismo para acelerar su funcionamiento, podemos hacer uso de la TPL directamente sobre nuestro código secuencial sin inducir grandes cambios ni restructuración de nuestra aplicación.

Porque, por tanto, usar Parallel .NET, ya que sabemos cuando podemos usarlo. En primer lugar la pregunta que surge es porque directamente .NET y no otras opciones como pueden ser Java, C o C++. La respuesta a esto es bastante clara, buscamos en primer lugar una solución escalable, adaptable y eficiente. En cuanto a la eficiencia, es obvio que la mejor solución pasa por C + C++, pero cuando hablamos de escalabilidad y adaptabilidad, Java y .NET ofrecen grandes recursos. Entre estas dos, la que mejor eficiencia ofrece es .NET aunque cuenta con el lastre de ser solo para plataformas Windows. Bien, esto realmente no es cierto, ya que se encuentra activa la iniciativa **Mono** actualmente para portar, de hecho con bastante éxito, aplicaciones .NET a plataforma Linux. En caso de no convencer esa solución, decir que la plataforma oficial de desarrollo de aplicaciones para una consola portátil de bastante actualidad, la **PS VITA** de Sony, es Mono, la adaptación de .NET para Linux. A esto debemos sumarle que se busca una comparativa sobre computadores personales, y este ámbito está altamente dominado por la plataforma Windows. Si queremos ofrecer alguna funcionalidad acelerada para nuestros clientes, esta es siempre nuestra primera opción, pues buscamos abarcar la mayor cuota de mercado posible, y a este respecto es necesario decir que la integración del Framework de .NET con el sistema operativo Windows, así como la

comunicación y el manejo conjunto de los recursos ofrece unos valores de eficiencia, rendimiento y escalabilidad que no pueden ofrecer otras plataformas sobre Windows, siempre dibujando este esquema desde una visión de alto nivel, por supuesto.

Un ejemplo de desarrollo sobre .NET con grandes resultados en eficiencia y escalabilidad esta presenta en [Referencia Trabajo Fin de Master y artículo], donde se consiguen resultados muy interesantes sobre arquitecturas personales sin modificar y sistemas operativos sin alterar, simulando una ejecución sobre un sistema cliente con una carga de trabajo intrínseca relativa a un uso medio.

7.3. Cuando no Parallel .NET

Como todas las tecnologías, hay momentos para aplicarlas y momento para retirarlas de nuestras opciones. En este caso, como se expone anteriormente, es necesario tratar con operaciones independientes sobre los datos, para evitar situaciones de generación de errores indeseados. Esto es aplicable a esta y a cualquier tipo de desarrollo paralelo, pues es la base de la paralelización.

De igual manera, debemos tener claros los ámbitos de actuación de nuestra aplicación, teniendo en cuenta que esta debe estar dirigida al publico general sobre plataforma Windows, o tener en cuenta durante su desarrollo una versión para Mono sobre Linux.

Capítulo 8

Estado del arte

En cuanto a los esfuerzos realizados en el terreno investigador respecto a la cuestión de la paralelización de algoritmos de extracción de características, se va a realizar una división de las distintas fuentes consultadas para mostrarlas en tres subgrupos. En primer lugar, se comentarán algunos esfuerzos llevados a cabo sobre el terreno de las implementaciones solo CPU. Posteriormente pasaremos a evaluar algunas fuentes interesantes que buscan la optimización desde un enfoque híbrido, usando según necesidad tanto tecnologías CPU como GPU. Por último, mostraremos todo lo relativo a las implementaciones basadas en procesamiento paralelo solo GPU como otra de las alternativas posibles.

8.1. Procesamiento solo CPU

En cuanto a las alternativas puramente CPU las referencias, si nos limitamos únicamente al procesamiento de extracción de características, es algo limitada y de cierta antigüedad, por lo que comentaremos solo una fuente en este caso.

Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks [GO]

En esta referencia se estudia e implementa un diseño paralelo de tareas de extracción de características sobre imágenes junto a un balanceo de carga de las unidades de procesamiento. Las tareas implementadas son la transformada de Hough y Momentos de la región.

Se basa en una comparativa entre un enfoque MIMD y SIMD sobre arquitecturas multiprocesador, que no multinúcleo, de factura comercial y alejadas del alcance de un hogar medio. En torno a estas máquinas se genera un algoritmo de control de la carga y balanceo de uso de recursos de procesamiento y memoria, con el fin de obtener el mejor resultado en rendimiento y eficiencia en el cálculo de las tareas sobre imágenes anteriormente

mencionadas. Los resultados obtenidos apoyan el uso del algoritmo de control de carga diseñado y empleado.

Aun así, queda fuera totalmente del ámbito y objetivo del presente proyecto, ya que se emplean arquitecturas complejas fuera del ámbito del ordenador personal y, por tanto, no se trata de una solución adaptable.

8.2. Procesamiento híbrido CPU y GPU

En lo que concierne a un procesamiento totalmente híbrido, aunando las capacidades de procesamiento paralelo de la arquitectura tanto CPU como GPU contamos con una referencia que creemos realmente interesante.

A Fast Feature Extraction in Object Recognition Using Parallel Processing on CPU and GPU [KP]

Esta referencia presenta una técnica de procesamiento paralelo para la extracción de características en tiempo real durante el proceso de reconocimiento de objetos por parte de robots autónomos móviles, basándose en la utilización de procesador convencional y procesador gráfico combinando tecnologías como OpenMP, SSE (Extensión SIMD de streaming) y CUDA.

El procesamiento CPU paralelo conlleva una serie de pasos necesarios, debido además al propósito de detección de objetos objeto del artículo. En primer lugar se realiza una configuración espacial de la escala de forma paralela, calculando el determinante de la matriz Hessiana de la imagen, a fin de encontrar los puntos clave candidatos. Esta operación, el cálculo del hessiano es invariante respecto cambios de escala de la imagen y se consigue mediante la búsqueda de características estables a través de todas las posibles escalas.

Una vez realizado esto, se realiza una detección paralela de los puntos clave mencionados, buscando como se ha comentado la estabilidad de estos puntos a lo largo de distintos factores de escala. Una vez realizado todo este proceso, se procede a construir los descriptores usando instrucciones SSE, entrando aquí en juego el paradigma SIMD.

En cuanto a la extracción de características en base a procesadores gráficos, es interesante consultar la figura 8.1.

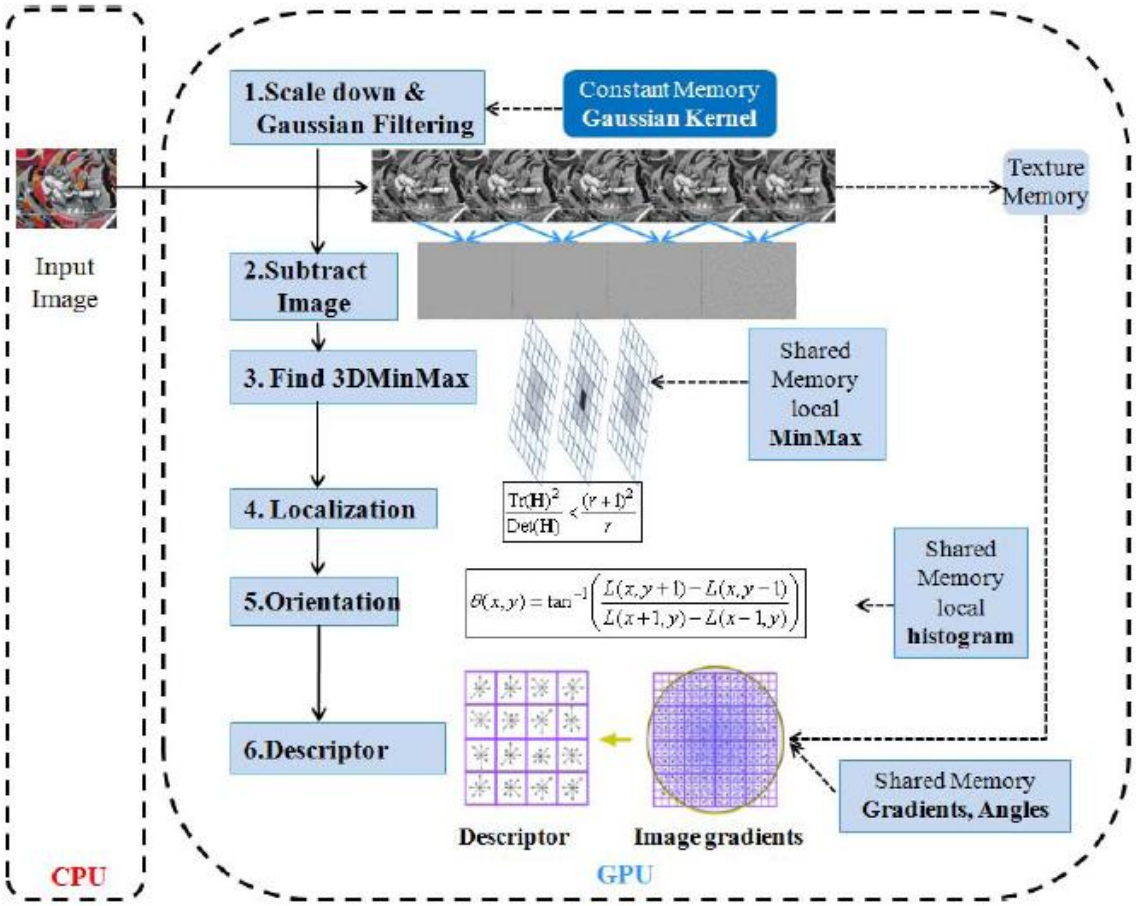


Ilustración 8.1 – Bucle de repetición secuencial

Como podemos ver, el único procesamiento que realiza la CPU es el transito de la imagen de la memoria local a la memoria del dispositivo gráfico. Una vez allí, se hace uso de las distintas características que ofrece la arquitectura CUDA para acelerar el proceso. Como podemos ver, en las distintas etapas presentes entran en juego zonas de memoria como la memoria de constantes, la memoria shared local o general y la memoria de texturas. Esto supone un aprovechamiento realmente alto de la arquitectura, pues es el uso de estas zonas predeterminadas de memoria para usos gráficos lo que confiere mayor rendimiento a cualquier posible ejecución CUDA, aprovechando tanto los recursos de procesamiento como, de esta manera, las distintas jerarquías de memoria existentes.

8.3. Procesamiento solo GPU

Ahora vamos a mostrar algunas referencias de implementación paralela limitada a GPU sobre la extracción de características de gran interés. Es interesante comentar que todas las referencias que incluyen computación GPU presentadas aquí o en su lugar correspondiente no se remontan a más de 3 años atrás, debido a lo novedoso de la tecnología empleada.

GP-GPU Implementation of the "Local Rank Differences" Image Feature [HJZ]

Un tema popular en la detección de objetos y el reconocimiento de patrones es el uso de clasificadores estadísticos. El rendimiento en velocidad de estos clasificadores depende en mucho en las características de bajo nivel que estén usando. La característica objeto de estudio es una alternativa al wavelet de Haar. Esta es perfecta para su implementación en FPGAs o hardware especializado, pero se demuestra en esta referencia que funciona de forma altamente eficiente en procesadores gráficos usados para computación de propósito general.

Dejando a un lado los detalles de implementación, se obtienen resultados ciertamente interesantes en el uso de la tecnología de procesamiento gráfico frente al procesamiento convencional, mostrando una aceleración superior a 10 conforme aumenta el tamaño de la imagen y los requisitos especificados.

Accelerating MATLAB Image Processing Toolbox Functions on GPUs [KDY]

La idea que se propone en este artículo es remplazar algunas implementaciones nativas de MATLAB, uno de las plataformas mas usadas en computación científica, con programas de código abierto procesando sobre GPU. Aunque no trata directamente sobre extracción de características si que realiza mejoras sobre el procesamiento de imágenes y linda de forma directa con los algoritmos y los objetivos del proyecto que se está presentando.

La novedad introducida radica tanto en la mejora de una biblioteca de funciones de Matlab que no había sido mejorada anteriormente por aplicaciones de computación GPU, como en el uso de las dos tecnologías comentadas en capítulos anteriores para desarrollo GPU, CUDA y OpenCL. A este respecto se presenta un análisis bastante exhaustivo de rendimiento de ambas plataformas gráficas.

A fin de abarcar el mayor número de algoritmos y realizar diseños acordes a las necesidades, se opta por dividirlos en cuatro grupos principales: independencia de datos, compartición de datos, dependencia de algoritmos y dependencia de datos. Cada categoría presenta por tanto una forma distinta de abordar los algoritmos y un uso distinto de las posibilidades de la tecnología de computación gráfica en cada caso.

A fin de obtener mejor calidad en los resultados, las implementaciones realizadas cuentan también con distintas estrategias de optimización, mostrando así la viabilidad de unas u otras según variables como el tipo de algoritmo o la carga de cada uno.

Repasando estos resultados podemos ver como se detalla de forma muy clara tanto el tiempo de procesamiento real de la GPU como el tiempo perdido en el movimiento de los datos desde la memoria del host a la memoria del dispositivo gráfico y su retorno. A este respecto, se introduce un retardo enorme frente al tiempo de computación, generando por ejemplo bajadas del speedup en procesamiento del 215x hasta el 11x por estos tránsitos de información.

Por tanto presenta a las alternativas GPU como una mejora, incluso ante entornos bien depurados y probados como es Matlab, y ratifica de forma notable la potencia y viabilidad de los sistemas de procesamiento gráfico como parte del nuevo paradigma de computación de altas prestaciones.

Capítulo 9

Estructura de las imágenes

En este capítulo se comentará tanto el formato de imagen elegido para la prueba de los distintos algoritmos de extracción de características que se comentarán más adelante, como la segmentación realizada y en base a que recursos ha sido llevaba a cado.

9.1. Formato de imagen

9.1.1. Introducción

El formato **BMP** ([M]) es el formato nativo de imágenes en los sistemas operativos **Microsoft** Windows. Este soporta imágenes de 1, 4, 8, 16, 24 y 32 bits por pixel, aunque archivos BMP usando 16 o 32 bits por pixel no son comúnmente encontrados.

Durante los años han existido diferentes versiones incompatibles entre sí del formato BMP, aunque nos centraremos en el formato BMP en uso desde la versión 3 de Windows.

9.1.2. Ordenación de los datos

En el formato BMP se guardan enteros multibyte comenzando por el bit menos significativo.

La estructura del archivo BMP es muy simple, como podemos ver en la figura 9.1.

Basic BMP File Format			
Name		Size	Description
Header		14 bytes	Windows Structure: BITMAPFILEHEADER
	Signature	2 bytes	'BM'
	FileSize	4 bytes	File size in bytes
	reserved	4 bytes	unused (=0)
	DataOffset	4 bytes	File offset to Raster Data
InfoHeader		40 bytes	Windows Structure: BITMAPINFOHEADER
	Size	4 bytes	Size of InfoHeader =40
	Width	4 bytes	Bitmap Width
	Height	4 bytes	Bitmap Height
	Planes	2 bytes	Number of Planes (=1)
	BitCount	2 bytes	Bits per Pixel 1 = monochrome palette. NumColors = 1 4 = 4bit palletized. NumColors = 16 8 = 8bit palletized. NumColors = 256 16 = 16bit RGB. NumColors = 65536 (?) 24 = 24bit RGB. NumColors = 16M
	Compression	4 bytes	Type of Compression 0 = BI_RGB no compression 1 = BI_RLE8 8bit RLE encoding 2 = BI_RLE4 4bit RLE encoding
	ImageSize	4 bytes	(compressed) Size of Image It is valid to set this =0 if Compression = 0
	XpixelsPerM	4 bytes	horizontal resolution: Pixels/meter
	YpixelsPerM	4 bytes	vertical resolution: Pixels/meter
	ColorsUsed	4 bytes	Number of actually used colors
	ColorsImportant	4 bytes	Number of important colors 0 = all
ColorTable		4 * NumColors bytes	present only if Info.BitsPerPixel <= 8 colors should be ordered by importance
	Red	1 byte	Red intensity
	Green	1 byte	Green intensity
	Blue	1 byte	Blue intensity
	reserved	1 byte	unused (=0)
	repeated NumColors times		
Raster Data		Info.ImageSize bytes	The pixel data

Ilustración 9.1 – Estructura de imagen BMP

Cabecera del archivo

Cada BMP comienza con una estructura denominada **BITMAPFILEHEADER**, cuya función principal es servir de firma para identificar el formato del fichero.

Es necesario tener en cuenta tres cosas para asegurar que estamos leyendo un archivo BMP:

- Los primeros dos bytes deben contener los caracteres ASCII “B” y “M”.
- El tamaño real del archivo en bytes debe ser igual al valor del campo **FileSize**.
- El campo **reserved** debe ser cero.

La cabecera también especifica la localización de los datos de los pixeles en el archivo. Cuando decodificamos un BMP debemos usar el campo **DataOffset** para determinar la distancia desde el inicio del archivo donde comienzan estos datos de los pixeles. Algunas aplicaciones introducen dicha información tras la estructura **BITMAPINFOHEADER** o la

paleta, en caso de que esta este presente. Aun así, también es común que se introduzcan bytes de relleno entre estas estructuras y los datos de pixeles, por lo que es necesario recurrir al campo DataOffset para determinar el número de bytes que se encuentran entre el **BITMAPFILEHEADER** y dichos datos.

Cabecera de la imagen

Esta estructura denominada BITMAPINFOHEADER sigue a la cabecera de archivo. Esta estructura nos da las dimensiones y la profundidad de bits de la imagen y nos dice si existe compresión. En caso de que el valor del alto de la imagen sea negativo, esto nos dice que los datos de los pixeles están ordenados de arriba abajo, en lugar de lo normal, de abajo a arriba. En estos casos las imágenes no suelen presentar compresión.

Paleta de Colores

La paleta de colores sigue a la estructura anterior y presenta cuatro valores que representan los colores azul, verde y rojo, mas un valor reservado que suele y debe ser cero.

Datos de los pixeles

Las filas de pixeles se ordenan de abajo hacia arriba. El numero de filas de datos esta dado por el atributo de altura de la imagen en su cabecera. El tamaño de las filas se determina por el número de bits y el ancho de la imagen. El número de bytes por fila se redondea al alza a un múltiplo de 4.

El formato depende el número de bits por pixel:

- De 1 a 4 bits por pixel. Cada byte de datos se subdivide en de 8 a 2 campos cuyos valores representan un índice en la paleta de colores. El bit más significativo representa al pixel más a la izquierda.

- 8 bits por pixel. Cada pixel en la fila se representa por 1 byte que es un índice a la paleta de colores

- 16 bits por pixel. Cada pixel se representa por un valor entero de 2 bytes. Si el valor del campo de compresión es 0, la intensidad de cada color se representa por 5 bits, con el bit más significativo sin usar.

- 24 bits por pixel. Cada pixel se representa por 3 bytes consecutivos que especifican el azul, verde y rojo, respectivamente.

- 32 bits por pixel. Cada pixel se forma con un entero de 4 bytes donde, si la compresión está desactivada, los tres bytes de menor orden representan los valores de 8 bits del azul, verde y rojo. El byte de mayor orden no se usa, por lo que es igual al formato de 24 bits por pixel pero con un byte al final.

Capítulo 10

Representación y descripción de imágenes segmentadas

A continuación, procederemos a detallar la base teórica de los distintos algoritmos de extracción de características de imágenes elegidos para su implementación de forma paralela ([PF], [GW]). Estos se agruparán en varias categorías, atendiendo a su naturaleza y a la característica que se busca obtener.

10.1. Descriptores Simples

Los descriptores simples son aquellos cuya complejidad computacional no esta compuesta de más de una o dos operaciones aritméticas simples, es decir, la aplicación de los operadores de suma, resta, multiplicación o división. Su baja complejidad es un factor importante a la hora de comparar resultados de ejecución paralela con ejecución secuencial, ya que proporciona una medida de aceleración para el peor caso. Es decir, proporcionan el marco más favorable para un procesamiento secuencial.

10.1.1. Primer punto

El primer punto de una región se define como el primer punto que se encuentra al realizar un barrido secuencial por la imagen de izquierda a derecha en horizontal y de arriba abajo en vertical.

Ilustración 10.1 – Ejemplo primer punto

En las figuras 10.1 y 10.2 podemos ver definido el primer punto, en color rojo, para dos regiones distintas, definidas por el color negro. Como se puede apreciar, es simplemente realizar el barrido comentado sobre la imagen, de izquierda a derecha, hasta que nos encontramos con el primer pixel de una región no explorada anteriormente, en este momento podemos definir a dicho pixel como el primer punto de la región mencionada. El resultado arrojado por la búsqueda del primer punto es un par (x,y) con las coordenadas del mismo.

Por tanto, para poder definir el primer punto de todas las regiones de una imagen segmentado debemos explorar toda la imagen, en caso de desconocer el número de regiones, o explorar la imagen hasta definir tantos primeros puntos como regiones contenga nuestra imagen. Este punto es interesante de cara a una implementación secuencial eficiente.

10.1.2. Área

El área de una región se define como el número de pixeles contenidos en su contorno, es decir, el número total de pixeles que constituye dicha región. Esta característica es invariante a traslación y rotación, aunque no a cambios de escala, por lo que solo es útil si no se realizan cambios de tamaño.

Ilustración 10.2 – Ejemplo área

Para el ejemplo anterior de la región en L invertida, podemos ver como la región definida por el color negro, se colorea de forma completa en el cálculo del área. Por tanto para realizar el cálculo del área de una región es necesario recorrerla en su totalidad para realizar un conteo del número de pixeles que la definen de forma completa. El resultado de este proceso será por tanto un número natural.

Por tanto, para poder definir el total del área de todas las regiones es necesario realizar un barrido de la imagen al completo, pues se deben contar todos los pixeles de cada una de ellas. Este es un proceso de poca carga de procesamiento pero altamente tedioso, por lo que ofrece un buen marco de referencia para introducir paralelismo.

10.1.3. Perímetro

El perímetro viene dado por el número total de pixeles que configuran el contorno de la región, teniendo especial cuidado con los que conforman bordes diagonales a los que hay que ponderar por la raíz cuadrada de 2.

En términos de vecindad, podemos decir que el perímetro está formado por un número de puntos con al menos un cero en su vecindad, ya sea 4-vecindad u 8-vecindad, introduciendo el factor de ponderación anterior para los límites diagonales de la región.

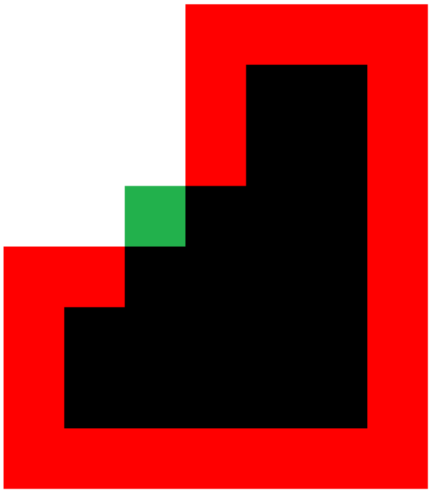


Ilustración 10.3 – Ejemplo perímetro

En este caso, siguiendo con el mismo ejemplo, podemos ver en rojo los pixeles que forman parte del contorno, contando como un pixel completo, y en verde podemos ver los pixeles, en este caso solo uno, que forman bordes diagonales. Por tanto el resultado final sería un número real, formado por el conteo de los pixeles más rojos más la raíz cuadrada de dos, en este ejemplo.

Si queremos realizar un cálculo del perímetro de todas las regiones de la imagen debemos, entonces, realizar un barrido completo de las regiones para destacar los puntos pertenecientes al subconjunto perímetro o límite de la región, clasificarlos y, finalmente, realizar el conteo mencionado anteriormente.

10.1.4. Densidad

El área u opacidad de una región de define de la siguiente forma:

$$D(X) = \frac{L(X)^2}{A(X)}$$

La densidad es una cantidad adimensional y es mínima para una región en forma de disco. Su valor será menor en tanto la región se aproxime más a una forma circular, ya que el circulo es la forma geométrica plana con densidad mínima. Como podemos ver, además, siempre será mayor que 0, ya que tanto el área como el perímetro de una región han de ser siempre valores positivos mayores que 1.

Si, en este caso, buscamos analizar la densidad de todas las regiones de la imagen, nos encontramos con que antes debemos haber realizado el cálculo tanto del área como del perímetro de dichas regiones, con lo que nos encontramos con una dependencia clara.

10.1.5. Volumen

La definición del volumen para nuestro caso se establece como la suma de las intensidades de los pixeles de la región a analizar, donde dicha intensidad se establece como su nivel de gris. Ya que estamos en el contexto de las imágenes segmentadas, es interesante observar que para poder obtener el nivel de gris, es necesario antes contar con la imagen original, para obtener así el nivel de gris asociada a cada pixel marcado como pixel de la región en la imagen segmentada.

$$V(X) = \sum_{i,j} I(x_i, y_j)$$

En este caso, estamos realizando una medida global de todos los niveles de gris de la imagen, mientras que si estuviésemos en el caso de una imagen binarizada, obtendríamos directamente el área de la región.

Por tanto, para calcular el volumen de todas las regiones de una imagen segmentada debemos realizar un barrido completo de la imagen e ir realizando sumatorias para los distintos segmentos, obteniendo de esta manera todos los volúmenes de una sola pasada, o un solo barrido.

10.1.6. Volumen^2

En este caso, el volumen al cuadrado presenta una variante del volumen anterior, modificando simplemente el sumatorio, donde en este caso no se suman las intensidades, si no el cuadrado de las mismas, dejando la ecuación de la siguiente manera:

$$V(X) = \sum_{i,j} I(x_i, y_j)^2$$

Ya que estamos hablando del mismo descriptor anterior, el modo de obtención del mismo es idéntico, así como el proceso general para todas las regiones.

10.1.7. Diámetro Equivalente

Cuando realizamos el cálculo del diámetro equivalente lo que buscamos es aproximar una región dada al área de un círculo de la siguiente manera:

$$DE(X) = \sqrt{\frac{4 * A(X)}{\pi}}$$

Esta característica permite, por tanto, comparar distintas regiones de manera que sean fácilmente diferenciables, asumiendo que no existen cambios de escala.

En el procesamiento actual podemos observar otra dependencia, en este caso solo del área, en el cálculo de este descriptor. Por tanto para poder procesar el diámetro equivalente de las regiones de una imagen segmentada, es necesario obtener antes el valor del área de la misma.

10.2. Descriptores basados en los niveles de gris

Este tipo de descriptores puede ser usado cuando no se realizan cambios de contraste, permitiendo analizar las regiones frente a cambios como traslaciones, rotaciones y cambios de escala. Como estamos hablando de descriptores de niveles de gris sobre una imagen segmentada, es en este momento donde es necesario el procesamiento tanto sobre la imagen segmentada como sobre la imagen original de forma completa, y no solo para algunas características como en el caso anterior.

10.2.1. Nivel mínimo de gris

El nivel mínimo de gris no es otra cosa que el menor valor de pixel existente en la región analizada.

Por tanto, para la obtención del nivel mínimo de gris de una región, se necesario explorar la región al completo para obtener el valor que minimiza a todos los de la región, generando así una cota inferior del nivel de gris para la misma.

Para el caso de la obtención de todos los niveles mínimos de gris de todas las regiones de la imagen, no es necesario más que hacer, por tanto, un barrido de la imagen junto a una comparativa de los valores de pixel de la imagen original para obtener estas cotas inferiores.

10.2.2. Nivel máximo de gris

Al igual que el anterior, se busca un valor que acote los niveles de gris de una determinada región, aunque en este caso esa cota sea superior. Buscamos entonces el nivel de gris mayor

de la región, esto es, el más alto valor de pixel en la imagen original asociado a la región de la imagen segmentada.

Así pues, para llevar a cabo este cálculo es necesario realizar, como anteriormente, un barrido de la imagen para así poder comparar los niveles de gris y obtener el mayor de ellos, ya sea para una sola región, o para obtener de forma conjunta todos los niveles máximos de gris de la imagen.

Como podemos observar, una vez obtenidos los niveles máximo y mínimo de gris, obtenemos cotas tanto superior como inferior de la intensidad en la región estudiada, pudiendo así hacer un análisis del contenido de la región en cuanto a nivel de intensidad más completo.

10.2.3. Nivel de gris medio

El nivel de gris medio de una región se considera como el nivel que se encuentra en la mitad de todos los valores de gris de la región de la imagen, y puede definirse de la siguiente manera:

$$M(X) = \frac{\sum_{i,j} I(x_i, y_j)}{A(X)} = \frac{V(X)}{A(X)}$$

Por tanto, se trata de una simple ecuación donde el área de la región divide al volumen de la misma para obtener el nivel medio de gris. Esto genera, como podemos ver, otra dependencia clara. Para poder realizar el cálculo del nivel de gris medio, necesitamos conocer tanto el área de la región como el volumen de la misma.

De esta manera, calcular el nivel de gris medio de una imagen completa pasa por realizar simplemente los cálculos de división de volumen y área tantas veces como regiones existan en la imagen estudiada.

10.2.4. Desviación típica

Siguiendo la definición formal de la desviación típica o desviación estándar, y aplicándola a niveles de gris, podemos obtener una medida de dispersión de estos niveles de la siguiente manera:

$$\sigma^2(X) = \sqrt{\frac{\sum_{i,j}(I(x_i, y_j) - M(X))^2}{A(X)}}$$

Para este caso generamos una dependencia tanto con el área de la región como con el nivel de gris medio introducido anteriormente. En base a estos dos valores es necesario realizar un barrido total de la región pixel a pixel, observando tanto la imagen segmentada como la original para obtener el operando que será dividido por el área de la región. Una vez obtenido, será tan sencillo como realizar tantas divisiones como regiones en total existan en la imagen.

10.2.5. Baricentro de los niveles de gris

En geometría, el baricentro o centroide de una superficie es un punto tal que cualquier recta que pasa por él, divide a dicha superficie en dos partes de igual momento respecto a dicha recta. Si nos acercamos al campo de las imágenes segmentadas, podemos decir que el baricentro de los niveles de gris determina donde se encuentra el punto medio de los niveles de gris de una región concreta, de la forma siguiente:

$$B(X) = \frac{\sum_{i,j}(I(x_i, y_j) * x_i)}{A(X)}$$

$$B(Y) = \frac{\sum_{i,j}(I(x_i, y_j) * y_j)}{A(X)}$$

De esta forma obtenemos tanto la coordenada x como la coordenada y del baricentro de la región.

Por tanto, la búsqueda de los baricentros de las regiones de una imagen segmentada son dependientes del área de la región, y es necesario un barrido de la imagen original en base a la imagen segmentada para obtener de esta manera cada baricentro.

10.3. Descriptores basados en el rectángulo circunscrito

El rectángulo circunscrito es aquel que circunscribe a la región que contiene de forma completa, como se pude ver en las figura 10.3.

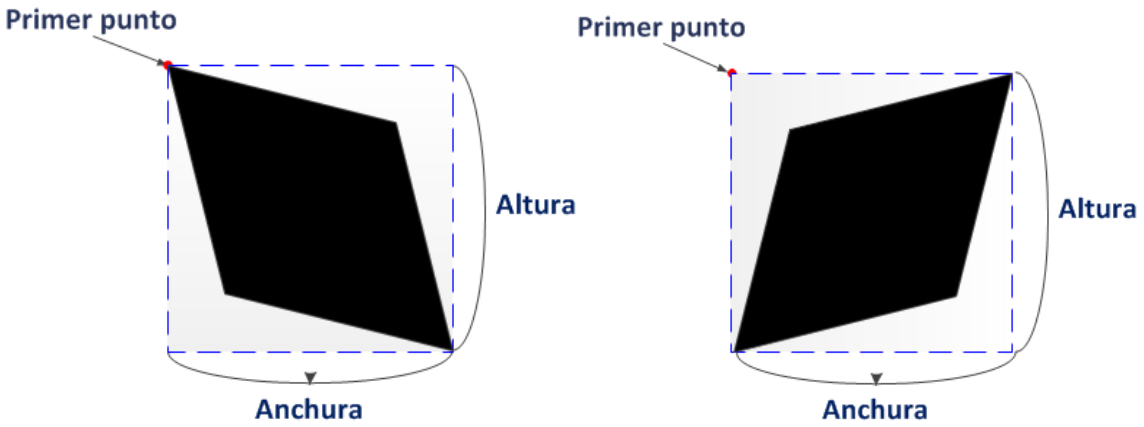


Ilustración 10.4 – Ejemplo Rectángulo circunscrito

En este caso, como podemos ver, no es necesario el uso de la imagen original, como en el apartado anterior, por lo que el procesamiento esta centrado en la imagen segmentada únicamente.

10.3.1. Primer punto

Anteriormente se ha presentado el primer punto de una región, pero en este caso el primer punto se va a referir al rectángulo circunscrito. Se define como el primer punto que podemos encontrar al hacer un barrido por la imagen de izquierda a derecha y de arriba abajo del rectángulo que rodea a una región dada.

Dos ejemplos de primer punto se pueden ver en la figura 10.3 de forma clara. En el primer ejemplo de la figura el primer punto coincide con el primer punto dela región, mientras que en el segundo ejemplo el primer punto de la región es totalmente distinto al primer punto del rectángulo circunscrito, que es lo usual.

10.3.2. Altura

La altura del rectángulo circunscrito de una región se define como el número de pixeles del contorno de uno de los lados del rectángulo circunscrito paralelo al eje y de la imagen, es decir, la altura del rectángulo en número de píxeles.

10.3.3. Anchura

En cuanto a la anchura, siguiendo el principio de la altura, se trata del número de pixeles del contorno de uno de los lados del rectángulo circunscrito paralelo al eje x de la imagen, o sea, la base del rectángulo en número de pixeles.

10.3.4. Área

Ya que estamos hablando de un rectángulo, el área del mismo no puede ser otra que el producto base por altura de un rectángulo. En este caso, la altura es la altura calculada anteriormente, y la base se corresponde, como hemos dicho, con la anchura del rectángulo circunscrito.

10.3.5. Extendido

El extendido busca dar una proporción del número de pixeles encerrados en el rectángulo en función del área de la región, determinado mediante la siguiente ecuación:

$$E(X) = \frac{A(X)}{Área\ del\ rectángulo\ circunscrito}$$

En este caso, existe una dependencia total del área de la región y el área del rectángulo circunscrito para este descriptor, por lo que su cálculo es una mera secuencia de divisiones, tantas como regiones a estudiar contenga la imagen objeto del estudio.

10.4. Puntos extremos

Los puntos extremos de una región son aquellos que acotan externamente de manera aproximada al perímetro. Se trata de una cota aproximada porque solo se llevará a cabo con ocho puntos. Estos descriptores son útiles, por ejemplo, para aproximar una solución en un tiempo mucho menor para el cálculo del número de Euler que explorando todo el perímetro de la región.

De forma gráfica, los ocho puntos extremos de una región a calcular pueden verse representados en la [figura].

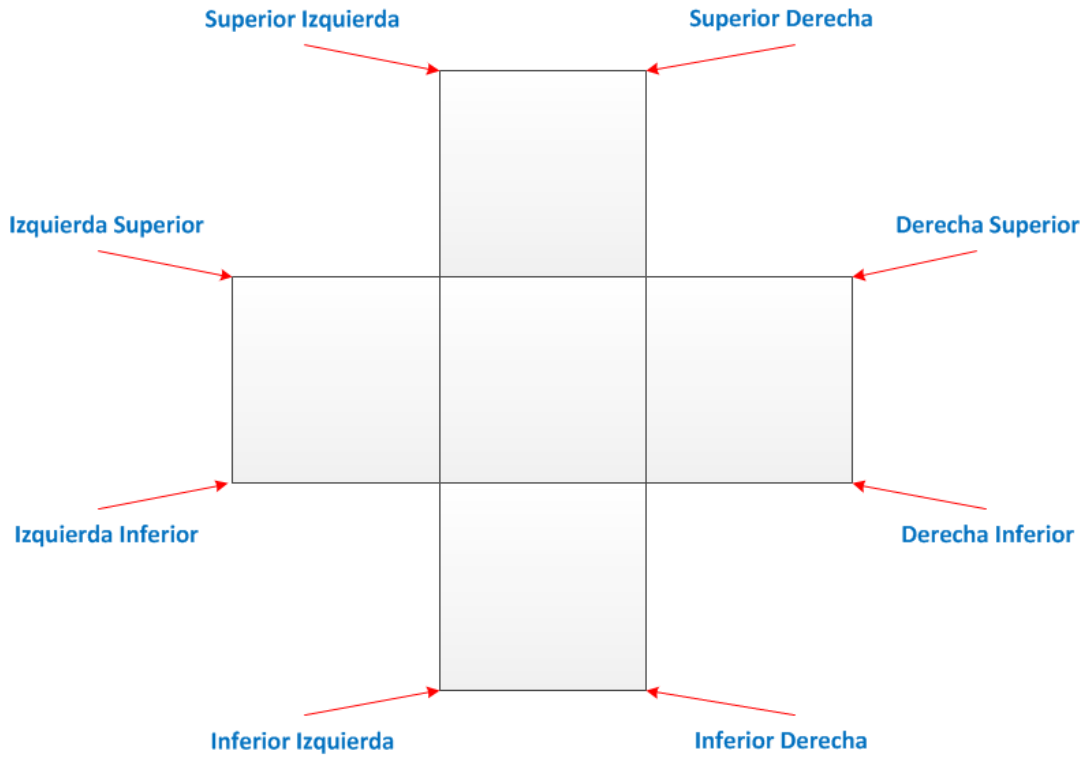


Ilustración 10.5 – Ejemplo Puntos extremos

Aunque puede darse el caso de varios puntos coincidan en la región a estudiar, sirva de ejemplo la figura 10.5.

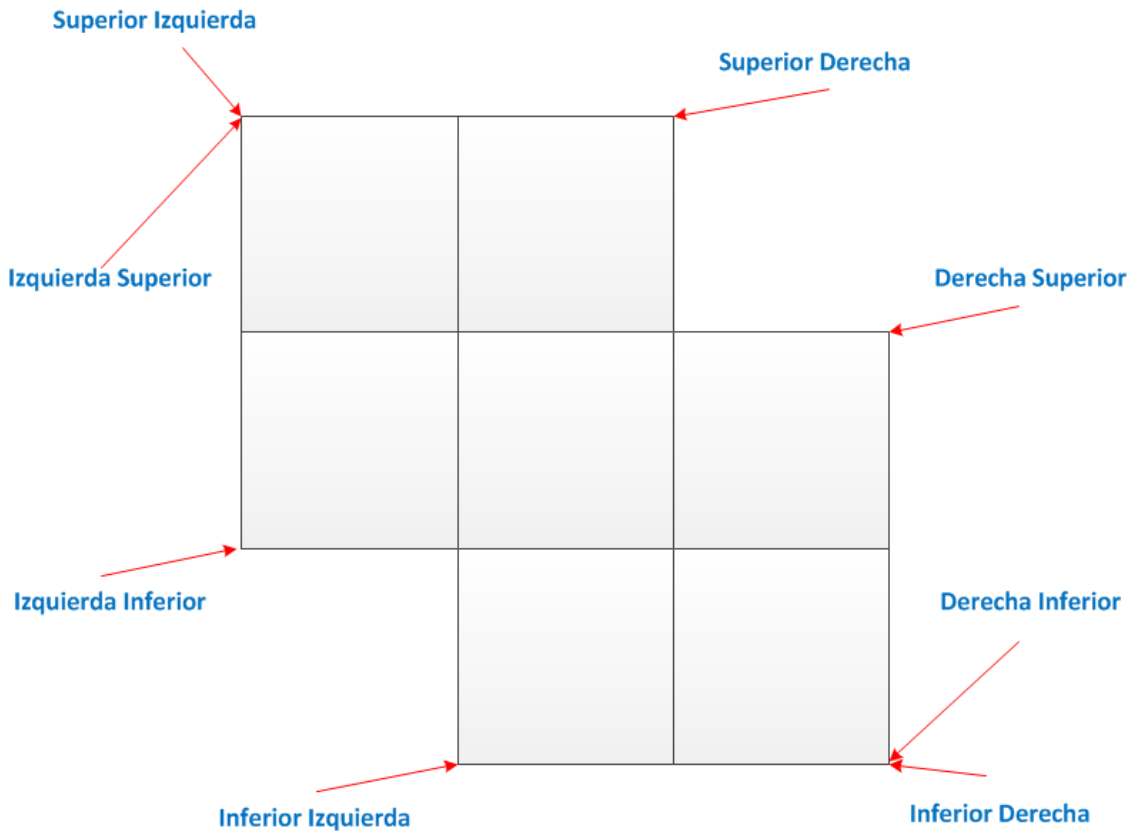


Ilustración 10.6 – Ejemplo Puntos extremos

Por tanto, podemos ver en la figura 10.6 como es posible que coincidan en espacio dos o más puntos de la región como puntos extremos, aunque esto no quiere decir, como vemos en la figura 1, que estos sean por definición el mismo punto.

10.4.1. Superior izquierda

Se trata del punto que se encuentra en la parte más alta, arriba, y más a la izquierda de la imagen. Siguiendo un barrido de izquierda a derecha y de arriba abajo sería el primer punto que nos encontrásemos.

10.4.2. Superior derecha

Se trata del punto que se encuentra en la parte más alta, arriba, y más a la derecha de la imagen. Siguiendo el barrido comentado en el punto anterior, este punto lo encontraríamos a la misma altura, valor de y, que el punto superior izquierda, y sería el último punto de la región que encontrásemos a ese nivel.

10.4.3. Inferior izquierda

En este caso se trata del punto opuesto, no de forma literal, al punto superior izquierda. Es el punto que se encuentra en la parte más baja y más a la izquierda de la región. Siguiendo con nuestro barrido, se trataría del punto más a la izquierda, es decir, el primero que encontraríamos, en la última fila, nivel de y, donde se encuentre la región analizada.

10.4.4. Inferior derecha

Se trata ahora del punto que se encuentra en la parte más baja y más a la derecha de la región. Por tanto, como en el caso del superior derecha, estaríamos a la misma altura, nivel de y, que en el punto inferior izquierda, y este sería el último punto que encontraríamos a ese nivel y, general, de la región.

10.4.5. Izquierda superior

El punto izquierda superior es el punto que se encuentra en la parte más a la izquierda y más arriba de la región. En este caso, en nuestro barrido, sería el punto que se encuentre en la menor columna, nivel de x, y, de todos los de dicha columna, el primero que encontraríamos.

10.4.6. Izquierda inferior

El punto izquierda inferior es el punto que se encuentra en la parte más a la izquierda y más baja de la región. Siguiendo nuestro camino por la imagen, se trataría de un punto al mismo nivel de x que el izquierda superior, pero en este caso en vez de ser el primer punto que nos encontramos, es el último punto de la región encontrado.

10.4.7. Derecha superior

Este punto se encuentra en el lado opuesto al punto izquierda superior. Es, por tanto, el punto que se encuentra más a la derecha y más arriba de la región. De forma análoga que en el punto izquierda superior pero de forma opuesta, en nuestro barrido sería el punto a nivel de x, columna, de mayor nivel y, sobre toda la columna, sería el primero que nos encontremos.

10.4.8. Derecha inferior

Se trata del punto que se encuentra en la parte más a la derecha y más baja de la región. Sobre nuestro recorrido en la imagen, sería el punto que se encontraría en la misma columna, a nivel de x, del derecha superior pero, a diferencia de este, se trataría del punto que nos encontraríamos en último lugar.

Parte III

Proyecto desarrollado

Capítulo 11

Diseño e Implementación

En este capitulo, procederemos a comentar los detalles de diseño e implementación del proyecto al completo. En primer lugar, daremos algunas pinceladas sobre la implementación de la lectura de la imagen, tanto en C# como CUDA y las distintas diferencias que podemos apreciar entre los dos enfoques. Posteriormente pasaremos a dar cuenta de la implementación de los distintos algoritmos de representación comentados en el capitulo anterior, en ambas tecnologías, así como las diferencias importantes entre ambas. Finalmente, comentaremos de forma breve los elementos de medición temporal usados para cada implementación y los detalles más relevantes de las estrategias de medición tomadas.

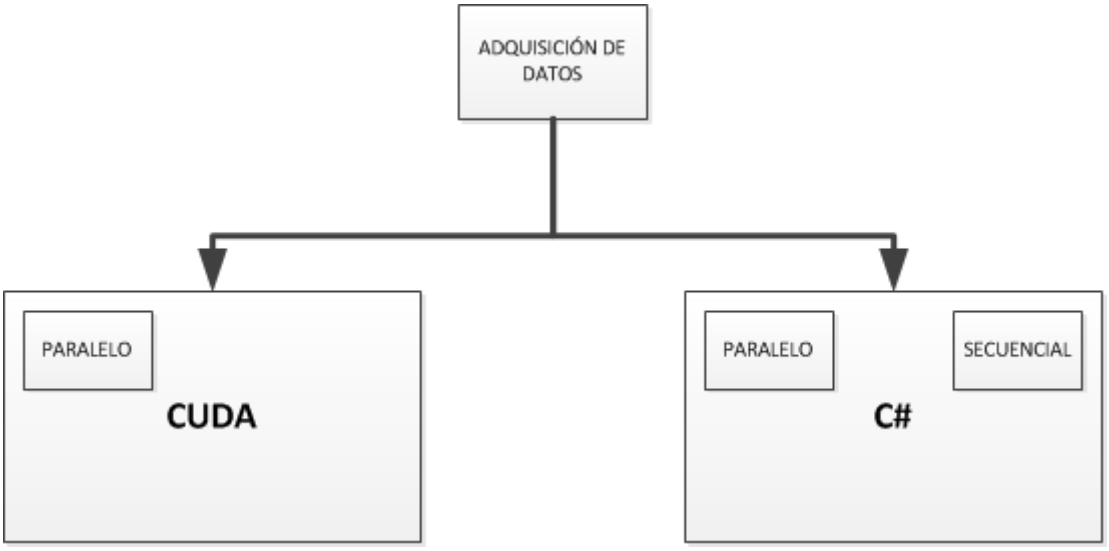


Ilustración 11.1 – Estructura de la implementación.

El desarrollo seguirá el esquema de la figura 11.1, donde se puede observar como existen dos grandes grupos de implementaciones, uno dedicado a cada tecnología. Además, nos encontramos con un primer grupo extra en el que se encuentra el proceso de adquisición de datos, que en este caso no es más que la lectura de la imagen bmp. Estos grupos principales se corresponden pues con CUDA y C#. En el primero nos encontramos con todas las

implementaciones paralelas realizadas sobre el dispositivo gráfico, y en el segundo nos encontramos tanto con las distintas versiones de implementación paralela como con la implementación secuencial realizada, que servirá de referencia para los speedup de los distintos descriptores desarrollados durante la fase de análisis de resultados.

11.1. Lectura de la imagen

Como hemos comentado en el capitulo correspondiente, el formato elegido para las imágenes fuente no es otro que el bmp, y para poder trabajar con dicho formato hemos de establecer un procedimiento de lectura del mismo común y único para ambas plataformas. No se ha querido usar en ningún momento ningún tipo de librería externa que condicione el proceso de lectura de la imagen, a fin de ofrecer las mismas posibilidades y funciones a ambas plataformas por igual. Esto es así debido a que el objetivo del presente proyecto, no es llevar a cabo una aplicación de caracterización rápida de imágenes, ni una librería altamente funcional y eficiente, si no establecer un marco de estudio sobre la idoneidad de dos tecnologías enfrentadas a un mismo problema, y para ello debemos darles los mismos datos, las mismas fuentes y las mismas armas de lectura de las mismas.

Por un lado contamos con C++ y CUDA como primera tecnología base, y por otro tenemos a C# y TPL. En cuanto a C++, y ya que no queremos, ni debemos, usar librerías externas propias para cada lenguaje, no tenemos ningún medio básico de lectura de imágenes bmp que pueda ser visto desde C# también, por tanto nos obliga a generar de forma propia ese marco de lectura común. En el caso de C# contamos con funciones nativas realmente bien implementadas que ofrecen una excelente relación de eficiencia y tiempo de procesado en cuanto a la lectura de las imágenes bmp, pero es solo para esta tecnología, con lo que nos vemos obligados a desarrollar el marco de lectura para C++ sobre C# igualmente.

Volviendo atrás al capítulo 9 podemos consultar la figura 9.1, que volvemos a mostrar a continuación para ilustrarnos.

Basic BMP File Format			
Name		Size	Description
Header		14 bytes	Windows Structure: BITMAPFILEHEADER
	Signature	2 bytes	'BM'
	FileSize	4 bytes	File size in bytes
	reserved	4 bytes	unused (=0)
	DataOffset	4 bytes	File offset to Raster Data
InfoHeader		40 bytes	Windows Structure: BITMAPINFOHEADER
	Size	4 bytes	Size of InfoHeader =40
	Width	4 bytes	Bitmap Width
	Height	4 bytes	Bitmap Height
	Planes	2 bytes	Number of Planes (=1)
	BitCount	2 bytes	Bits per Pixel 1 = monochrome palette. NumColors = 1 4 = 4bit palletized. NumColors = 16 8 = 8bit palletized. NumColors = 256 16 = 16bit RGB. NumColors = 65536 (?) 24 = 24bit RGB. NumColors = 16M
	Compression	4 bytes	Type of Compression 0 = BI_RGB no compression 1 = BI_RLE8 8bit RLE encoding 2 = BI_RLE4 4bit RLE encoding
	ImageSize	4 bytes	(compressed) Size of Image It is valid to set this =0 if Compression = 0
	XpixelsPerM	4 bytes	horizontal resolution: Pixels/meter
	YpixelsPerM	4 bytes	vertical resolution: Pixels/meter
	ColorsUsed	4 bytes	Number of actually used colors
	ColorsImportant	4 bytes	Number of important colors 0 = all
ColorTable		4 * NumColors bytes	present only if Info.BitsPerPixel <= 8 colors should be ordered by importance
	Red	1 byte	Red intensity
	Green	1 byte	Green intensity
	Blue	1 byte	Blue intensity
	reserved	1 byte	unused (=0)
	repeated NumColors times		
Raster Data		Info.ImageSize bytes	The pixel data

Ilustración 11.2 – Estructura de imagen BMP

Como podemos ver, la estructura del fichero se compone de la cabecera del mismo fichero, la cabecera de la imagen y la tabla de colores y el contenido de los pixeles, y todo codificado en un archivo binario. Por tanto para llevar a cabo el proceso de lectura debemos leer nuestro archivo de forma binaria, extrayendo así los datos necesarios del mismo.

En primer lugar comenzamos por la cabecera, lo primero y más importante es comprobar que comienza por BM, en sus dos primeros bytes, para cerciorarnos de que el tipo de archivo es correcto. Recordemos ahora que estamos trabajando sobre archivos BMP con paleta de colores y sin compresión, como se comentó en el capitulo anteriormente citado. Posteriormente obtenemos el tamaño del archivo, a modo de información de comprobación de que todo es correcto. Los siguientes cuatro bytes se desechan por inútiles, y guardamos los siguientes cuatro como la posición dentro del archivo a partir de la cual están los datos de los píxeles de la imagen.

Una vez pasada la cabecera, pasamos a la cabecera de información de la imagen vemos el tamaño de la misma y obtenemos el ancho y alto de la imagen, parámetros que guardaremos en variables para su uso más adelante durante los distintos algoritmos, y aun aquí mismo para obtener la matriz de píxeles. A continuación nos encontramos con dos bytes que nos indican el número de planos de la imagen, uno en todo caso, y otros dos bytes que nos dan información del número de bits por pixel. El siguiente campo es la compresión, donde esperamos encontrar el valor cero, que indica BI_RGB, es decir, sin compresión. El siguiente parámetro es el tamaño de la imagen comprimida, por lo tanto lo desechamos, y los dos siguientes nos dan la resolución de la imagen en pixel por metro en horizontal y en vertical. A continuación, tenemos el número de colores usados y de colores importantes.

Una vez finalizada esta cabecera, leemos la paleta de colores, que nos da una lectura de los valores RGB a los que apuntaran los pixeles contenidos a continuación. En nuestro caso estamos trabajando con imágenes en blanco y negro, por lo que los valores de RGB darán todos el mismo valor, el tono de gris usado.

Una vez terminado todo esto nos encontramos con los datos de los píxeles de la imagen, lugar en el que tenemos que tener en cuenta dos cosas realmente importantes.

Lo primero es que los pixeles están almacenados de izquierda a derecha y de abajo a arriba, no de arriba abajo como cabría esperar en una lectura de fichero convencional, y además, hay que tener cuidado con el fenómeno de padding o relleno que se produce debido a que los datos contenidos han de ser múltiplo de cuatro. Por tanto si al leer este fragmento del archivo de imagen no tenemos en cuenta el relleno, podremos caer en leer datos basura y tomarlos como pixeles reales de la imagen, cuando no son más que información de relleno sin valor que debemos desechar.

Todo esto da lugar a un procedimiento que podemos llamar lectura, que se realiza dos veces durante la carga de las fuentes, una para la lectura de la imagen segmentada, donde podremos realizar el procesamiento directo sobre las regiones, y la imagen original, donde acudiremos para cualquier procesado que necesite de los niveles de gris de la imagen para cada región concreta.

11.2. Algoritmos de representación y caracterización

A continuación, vamos a presentar los detalles de diseño e implementación de los distintos algoritmos expuestos anteriormente en ambas tecnologías, tanto CUDA junto a C++ como C#. La organización será por conjunto de algoritmos, por tanto tendremos cuatro categorías, dentro de cada cual se expondrá el desarrollo primero sobre una plataforma, y luego sobre la otra, comentando diferencias y similitudes, así como detalles importantes a tener en cuenta entre las dos, de cara a extraer de forma posterior resultados concretos.

Como se ha comentado en el apartado 11.1 no se va a hacer uso de ningún tipo de librería externa, así que a partir de aquí, y en base a los datos obtenidos de las imágenes fuente, mediante el proceso de lectura de imagen anterior, es donde se pondrán al descubierto las armas, metodologías y puntos débiles de cada tecnología, con el fin de obtener lo mejor de cada una en todo momento.

11.2.1. Descriptores simples

Respecto a los descriptores simples, recordemos que se trataba del área, el primer punto, el perímetro, la densidad, el volumen, el volumen cuadrado y el diámetro equivalente.

Es interesante comenzar anotando que para cada uno de los descriptores anteriores, no solo se ha implementado una versión paralela sobre CUDA y C#, si no que además se ha codificado una versión secuencial sobre C#, a fin de tener una pequeña medida de mejora sobre lo que podría ser un procesamiento secuencial estándar del mismo.

Se va a explicar como una serie de pequeños apartados, uno por descriptor, donde se enumerarán las características de la implementación sobre cada tecnología. Los detalles de cada implementación tecnológica se explicarán siempre más a fondo en el primer descriptor en el que aparezcan, asumiendo así su conocimiento en los siguientes.

11.2.1.1. Área

Como se ha comentado, se va a realizar una implementación secuencial por descriptor, además de las implementaciones paralelas, por lo que comenzamos con la secuencial en primer lugar.

Basándonos en lo expuesto en el capítulo teórico, capítulo 10, para calcular el área de una región es necesario realizar un barrido por la región, y si queremos hacerlo para todas las regiones a la vez, debemos barrer la imagen al completo. Por tanto la implementación secuencial no es más que un bucle de repetición, donde para cada elemento de la imagen, se contabilizará un punto para el área del segmento al que pertenezca.

Vista la sencillez de la implementación secuencial, que nos acompañará en casi todas las implementaciones secuenciales, procedemos a las versiones paralelas del cálculo del área. Comenzaremos por el lenguaje C# debido a que se ha utilizado como primera plataforma de implementación y banco de pruebas, en parte por las complicaciones que se derivan de los procesos de depuración sobre CUDA, y en parte por ser una tecnología más amigable.

Implementación Paralela C# - Evolución del algoritmo paralelo

Como hemos dicho C# va a ser nuestro banco de pruebas, y en concreto el cálculo del área como primera característica implementada llevará el peso de las pruebas iniciales, tanto en estrategia como en implementación.

a. Paralelización con recurso compartido

La primera idea de paralelización es lanzar tantos hilos a ejecución como filas contenta nuestra imagen, es decir, tantos hilos como la propiedad alto de la imagen leída de archivo, y contener un vector con tamaño igual al número de segmentos, en el que cada pixel sume uno al conteo de la región a la que pertenece. Esta idea es interesante si buscamos explotar todo el paralelismo posible de la máquina y la fuente de datos, pero nos encontramos con un problema claro, el acceso simultaneo a un recurso compartido. Por tanto para poder usar esta opción de paralelización simple por filas, o paralelización total por pixel, debemos implantar un mecanismo de seguridad de acceso al recurso compartido.

Hasta ahora, hemos comentado el lanzamiento de diversos hilos a la vez, y el uso de un control de seguridad de acceso al recurso compartido, pero antes de continuar, debemos saber como podemos realizar esta implementación sobre C#, para comprender de mejor manera la bondad de una alternativa frente a otra.

Como se comentó en el capítulo de recursos software y en la explicación de las tecnologías, se ha elegido C# por su facilidad, seguridad y robustez en el manejo de hilos y su orientación a tareas sobre .NET. Bien, para realizar por tanto la creación de esos diversos hilos, vamos a hacer uso de cierta funcionalidad de C#, en concreto el bucle de repetición paralelo, como podemos ver en el siguiente fragmento de código.

```
Parallel.For(0, alto, delegate(int i)
{
   //Contenido del búcle
});
```

Fragmento 11.1 – Bucle paralelo en C#

Como podemos ver, mantiene casi intacta la estructura de un bucle de repetición convencional, pero nos posibilita abstraernos del manejo de hilos y centrarnos en el paradigma de las tareas. En este bucle, estamos diciendo al framework de .NET que queremos que se creen tareas de 0 a alto exclusive, numeradas por el delegado i, y cada tarea ejecutará el contenido del bucle de repetición de forma independiente.

Ahora necesitamos conocer el proceso de seguridad que podemos usar para asegurar un acceso exclusivo al recurso compartido. En este caso vamos a usar un cerrojo, y así evitamos cualquier acceso indeseado.

```
lock (result)
{
   //Proceso a ejecutar
}
```

Fragmento 11.2 – Cerrojo en C#

Ahora bien, conocemos los dos recursos que vamos a usar, pero podemos vislumbrar ya un motivo para cambiar de estrategia. No tiene sentido alguno, realizar un procesamiento secuencial de las filas de la imagen, generando un gran conjunto de hilos que intentaran actualizar a la vez una variable, si serializamos el acceso a la misma de esta forma, pero por otro lado, necesitamos asegurar el acceso seguro, por lo que esta alternativa presenta su capitulación.

b. Paralelización por segmentos

La idea era interesante, generar un alto grado de paralelismo por filas y conseguir un acceso rápido a los pixeles, pero no era la forma adecuada. Podemos presentar por tanto otra alternativa, en la que perdemos grados de paralelismo pero ganamos en seguridad. Vamos a aprovechar que estamos trabajando con imágenes segmentadas, y realicemos el paralelismo en lugar de sobre la imagen, sobre los segmentos. De esta manera, se propone un nuevo enfoque paralelo por segmentos, haciendo uso del bucle de repetición paralelo introducido anteriormente.

En este caso, la repetición no se hará de 0 hasta el alto de la imagen, si no hasta el número de segmentos de nuestra imagen fuente segmentada. Además, el recurso compartido anterior, ahora no presenta problemas, pues gracias al control que proporciona TPL, podemos leer de la posición del vector resultados, de tamaño el número de segmentos, y escribir en ella, debido a que somos los únicos interesados en la escritura en esa posición concreta. No existen por tanto problemas de acceso a recurso compartido, ni problemas de caché por escritura y lectura del recurso, gracias también a la gestión realizada por TPL y .NET.

Así pues, acabamos de eliminar la problemática del acceso al recurso compartido de una manera bastante simple, pero nos encontramos con un problema, acabamos de perder un alto grado de paralelismo, tanto como alto de la imagen, para reducirlo considerablemente al número de segmentos de la misma. Por tanto, aunque es un buen enfoque, que guardamos para la sesión de pruebas de rendimiento, es necesaria la búsqueda de una alternativa, cuyo grado de paralelismo sea considerablemente más alto.

c. Paralelización por filas y segmentos

El sentido común nos dicta que si en la alternativa a. teníamos un alto grado de paralelismo, y en la b. una solución al problema del recurso compartido, intentemos generar un híbrido que solucione ambos problemas de manera eficiente.

Por tanto, ignorando las leyes establecidas por la genética, y quedándonos solo con lo mejor de sus dos padres, generamos un nuevo enfoque híbrido. En primer lugar, buscamos alto grado de paralelismo, y consideramos que tanto paralelismo como el alto de la imagen, puede

ser un grado suficiente, por tanto, no hemos de abandonar esa idea. El bucle de repetición iterará de forma paralela, por tanto sobre la variable del alto de la imagen. En cuanto al problema del recurso compartido, tuvo su solución generando una dependencia del hilo escritor sobre únicamente una posición del recurso. Ya que en este caso hay tantos hilos escritores como filas en nuestra imagen, el recurso de escritura contará con tantas zonas de escritura como filas tenga la imagen. Este recurso, además, se organizará en segmentos, con lo que cada fila de la imagen, cada hilo de la tarea por tanto, contará con un vector con tantas posiciones como segmentos contenga la imagen y, en cada uno de esas posiciones, irá escribiendo conforme avance por su pequeño conjunto de datos de exploración. De esta manera, cada vez que una fila de la imagen, como tarea ejecutándose, explore un pixel de si misma, aumentará el conteo de pixeles de la región a la que pertenece dicho pixel. Una vez finalizado el proceso nos encontraremos con tantas estructuras de tamaño el número de segmentos como alta sea la imagen, y solo restará sumar las posiciones primera a última de las mismas, iterando sobre el número de segmentos, obteniendo finalmente una estructura donde quedará registrada el área de cada segmento, y todo ello sin perder un ápice de paralelización, ni poner en peligro la estructura de datos. Obviamente, el último recorrido para la obtención de la estructura final, también es paralelo, iterando como hemos dicho sobre los segmentos.

De esta manera, podemos fijar por tanto una estrategia de acercamiento a la paralelización evitando cualquier problema de pérdida de grado de paralelismo o pérdida de fiabilidad en los resultados por mal control de las estructuras compartidas. Este será, por tanto, el esquema a usar en las sucesivas paralelizaciones llevadas a cabo, tanto en C# como en CUDA, para cada descriptor, cada uno con sus peculiaridades por su puesto, salvo que su cálculo exija otros procedimientos.

Implementación Paralela CUDA

Ahora entramos en un paradigma distinto completamente al anterior, donde los recursos y la forma de usarlos nada tienen que ver. En C# comprobamos como con una simple modificación en nuestro bucle de repetición cotidiano podíamos conseguir un procesamiento paralelo de forma simple, sencilla y segura. En este caso requiere algo más de trabajo conseguir los resultados deseados, y a veces es necesario varias los enfoques para adaptarlos a esta arquitectura [Referencia Articulo Mio].

Al igual que sobre C# teníamos código paralelo y código secuencial, en CUDA podemos tener código host, fragmentos que se ejecutarán sobre nuestro procesador convencional, y código device, fragmentos que se ejecutarán sobre nuestro procesador gráfico, y es importante conseguir dividir bien ambos paradigmas, para que el balanceo sea adecuado. Por ello, es necesario conocer un poco más como podemos definir código device dentro de un código host, en nuestro caso será C++, y como interactuar con él.

En primer lugar debemos saber que las variables y conjuntos de datos que declaremos en nuestro código host, son distintas de las del código device, por tanto, si queremos que nuestro procesador gráfico trabaje con los píxeles de la imagen, es necesario antes traspasar esta información del host al device. Como es lógico, para poder obtener el resultado de las

diversas operaciones realizadas en el device, es necesario volver a traer esa información de vuelta. Todo este proceso conlleva un costo de tiempo, ya que estamos lanzando información desde nuestra memoria local hacia el dispositivo gráfico, con lo cual cuanto más rápido sea el bus y la tecnología usada, así como la memoria, menos tiempo perderemos. Es decir, debemos evitar intercambios continuos de información, e intentar usar estas funciones de transito lo menos posible.

Estamos hablando de mandar y recibir datos, pero CUDA es muy similar a C, y para poder enviar un dato, antes debemos reservar memoria para ello. Para centrar un poco todo lo comentado, vamos a mostrar unos ejemplos de como reservar memoria, mandar datos al device y traerlos de vuelta.

```
cudaMalloc((void**)&d_area, nSegmentos*sizeof(int));
```

Fragmento 11.3 – Reserva de memoria en CUDA

Si en C usamos un malloc, en CUDA, usaremos un cudaMalloc, en el que debemos especificar para qué variable estamos reservando memoria, y el tamaño de la reserva, como cualquier reserva de memoria host normal.

Para el caso del envío de datos tanto de ida como de vuelta, vamos a presentar un pequeño ejemplo, para así poder ilustrarlo de forma correcta.

```
cudaMemcpy(d_area, area, nSegmentos* sizeof(int), cudaMemcpyHostToDevice);
```

Fragmento 11.4 – Copia de memoria host a memoria device en CUDA

```
cudaMemcpy(d_area, area, nSegmentos* sizeof(int), cudaMemcpyDeviceToHost);
```

Fragmento 11.5 – Copia de memoria device a memoria host en CUDA

El fragmento 11.4 nos muestra como copiar una estructura de nuestra memoria la del device. Si en C usamos memcpy, en CUDA usaremos cudaMemcpy. Como vemos tanto la sentencia de llevar datos como la de traerlos presentan la misma estructura, cambiando simplemente la cadena última, donde indicamos la dirección de copia. Recordemos que las variables con d_ delante son variables del dispositivo, y se nombran así para diferenciarlas, por lo que primero ubicamos la variable del device, seguidamente la del host, y a posteriori el tamaño de la variable y la dirección de copia.

Por tanto, ya conocemos someramente la forma de realizar una copia de una variable de un lugar a otro de nuestra arquitectura de procesamiento, así como reservar memoria para esas variables que vamos a copiar o usar. Ahora nos queda conocer como debemos hacer para ejecutar esos fragmentos de código en nuestro device y poder darle instrucciones de trabajo.

Los fragmentos de código device o CUDA se organizan en kernels, y a continuación se presenta un pequeño ejemplo de kernel.

```
extern "C"
__global__ void d_inicializarResultado(const int tamanY, const int
nSegmentos, int* resultado)
{
	int idx = blockDim.x * blockIdx.x + threadIdx.x;
	if (idx < tamanY)
	{
		for (int i = 0; i < nSegmentos; i++)
		{
			resultado[idx*nSegmentos + i] = 0;
		}
	}
}
```

Fragmento 11.6 – Ejemplo de kernel en CUDA

Se trata de una función, definida por una etiqueta delante de ella. En este caso se etiqueta global, es decir, puede ser lanzada desde host y desde device, si la etiqueta contuviese device, se trataría de una función solo para uso desde el mismo código CUDA. En todo caso, continuamos con la función, que podemos ver muestra la forma normal de una función en C, ya que CUDA no deja de ser código C, donde definimos el nombre de la función y las variables de entrada. Los detalles internos de la función no son importantes, pero lo que si es necesario comentar es que este enfoque es muy similar al enfoque C#, donde el interior de un bucle de repetición paralelo realizaba una tarea, tantas veces como indicásemos. En este caso, lo que se encuentra dentro del kernel CUDA, se ejecutará tantas veces como hilos de ejecución creemos al llamar a dicho kernel.

Para poder llamar al kernel, el necesario seguir los pasos descritos en el siguiente fragmento de ejemplo.

```
d_calcularArea<<<blocksPerGrid, threadsPerBlock>>>(d_indice, nSegmentos,
altoImagen, anchoImagen, d_imagen, d_resultado);
cudaThreadSynchronize();
```

Fragmento 11.7 – Llamada de kernel en CUDA

Este ejemplo muestra el procedimiento para el cálculo de áreas de las regiones de una imagen segmentada. Como podemos ver, es una llamada corriente a una función C, con una salvedad, y son los parámetros que aparecen entre el nombre de la función y los argumentos de la misma. Se trata de los parámetros del número de hilos o ejecuciones que queremos que se realicen en el device. La estructura de hilos en CUDA es algo diferente a los conjuntos de hilos que podemos formar en nuestras implementaciones paralelas host.

Cuando un kernel se lanza, este se ejecuta como una malla de hilos paralelos. En la figura [] e lanzamiento del kernel 1 crea la malla 1. Cada malla de hilos CUDA se compone normalmente de miles de hilos GPU de baja carga por cada invocación de kernel. La creación de suficientes hilos para utilizar de forma completa el hardware requiere un alto grado de paralelización, por tanto no es fácil conseguirlo.

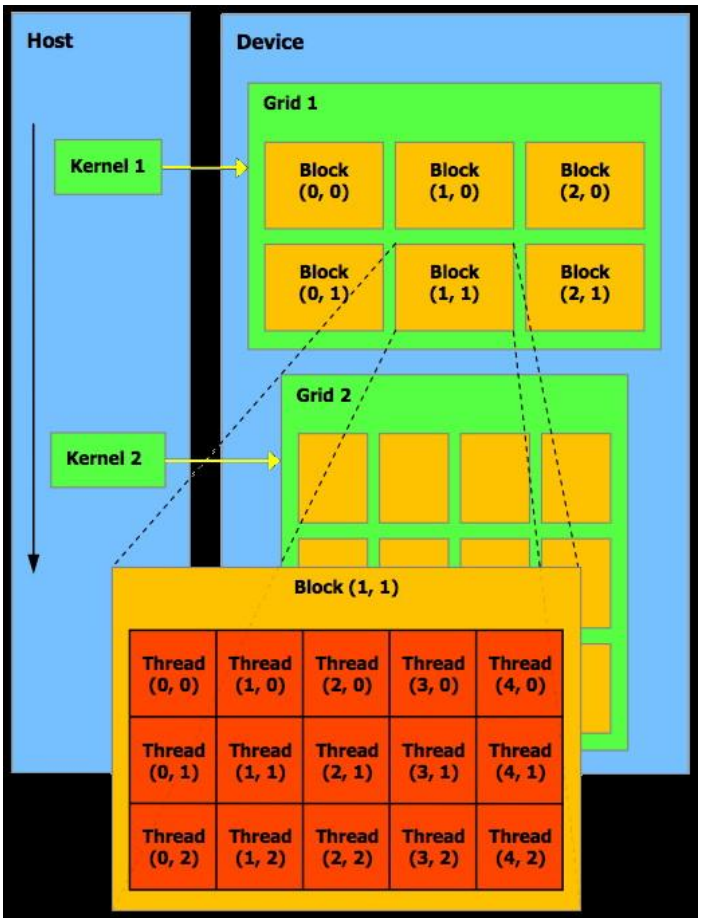


Ilustración 11.3 – Estructura de ejecución CUDA

Los hilos de una malla están organizados en una jerarquía de dos niveles, como vemos en la figura 11.3. En el nivel más alto, cada malla consiste en uno o más bloques de hilos. Todos los bloques de una malla tienen el mismo número de hilos, donde cada bloque se organiza en un array tridimensional de hilos, con un máximo de 512 hilos por bloque.

Ahora que conocemos estos datos, podemos pasar a esos dos parámetros que aparecen en la invocación de un kernel. El primero es el número de bloques por malla que vamos a usar y el segundo es el número de hilos que se usarán. Si no se va a explotar la totalidad de los recursos, y a fin de obtener un mayor paralelismo para cantidades pequeñas de datos, se suele definir un total de 256 hilos por bloque, y el número de bloques por malla es variable, dependiendo del total de elementos que vayamos a procesar, a fin de no generar más bloques de los necesarios.

Además, podemos observar otra instrucción debajo de la invocación del kernel, esta es una llamada de sincronización CUDA que debemos usar cada vez que queremos sincronizar todos los procesadores y la memoria. Si no la realizamos, podemos intentar, por ejemplo, traer un resultado del dispositivo gráfico antes de que todos los procesadores gráficos hayan terminado su trabajo, por tanto es una instrucción de sincronización importante, como las que se usan en programación hilada convencional.

Por tanto, y finalizando toda la introducción sobre CUDA, ahora conocemos como reservar memoria para un recurso, como llevar y traer ese recurso al dispositivo gráfico, como es un kernel de ejecución y como podemos invocar ese kernel para que se produzca un procesamiento.

Ahora que ya tenemos todo claro, podemos proceder a explicar los detalles de la implementación del área de todas las regiones de una imagen segmentada en procesadores gráficos.

Para el cálculo del área, en primer lugar debemos llevar a cabo el paso de datos de los pixeles de la imagen hacia el procesador gráfico, con el coste de tiempo que ello conlleva. En la sección de resultados, se ofrecerán estos contando con el tiempo de transito a memoria y obviándolo, para observar su impacto en el rendimiento. Una vez tenemos allí los recursos necesarios, procedemos a llamar al procedimiento de inicializar resultado mostrado anteriormente. Este es común para todos los descriptores y se encarga de preparar la estructura de salida de resultados antes de cada ejecución. Se escoge este enfoque debido a que es más rápido que su procesamiento en host y su carga en memoria del device.

Una vez todo está listo, sincronizamos el dispositivo gráfico y procedemos a ejecutar el kernel del cálculo del área. Cuando finaliza, nos traemos el resultado a la memoria convencional y procesamos el conjunto de resultados para obtener el resultado final. Este enfoque es, como hemos dicho antes, el enfoque de procesamiento híbrido con paralelización por filas de la imagen y regiones.

Con esto queda explicado el algoritmo del cálculo de área tanto en CPU como en GPU, además de los pormenores interesantes de ambas tecnologías, necesarios para comprender el uso de una u otra alternativa en la implementación. A partir de aquí, nos limitaremos a explicar de forma somera la implementación sin volver a entrar en estos pormenores.

11.2.1.2. Primer punto

El proceso de cálculo del primer punto se detallará más adelante, durante la explicación de los puntos extremos, debido a la conexión entre todos ellos. De todas maneras, se ha realizado una implementación tanto secuencial como paralela que procedemos a explicar.

Para realizar el cálculo del primer punto de forma secuencial, se necesario hacer un barrido de la imagen, de izquierda a derecha y de arriba abajo, hasta encontrar todos los primeros puntos de todas las regiones.

La forma escogida de paralelización es la mencionada en el apartado anterior, realzando el paralelismo por segmentos y filas, debido a las mejoras que representa con respecto a las alternativas anteriormente propuestas.

Implementación Paralela C#

La implementación en C# consta de las tres versiones comentadas, la paralelización total, la paralelización por regiones o segmentos, y la paralelización híbrida, que se presenta como la mejor de las opciones de paralelización para este descriptor concreto.

Implementación Paralela CUDA

En cuanto a la implementación CUDA, se sigue el mismo esquema anterior, donde primero se realiza una inicialización de los vectores de resultados, uno para cada coordenada del punto, para posteriormente realizar la invocación del kernel y traer los datos de vuelta para procesarlos en host y ofrecer el resultado del cálculo del primer punto.

11.2.1.3. Perímetro

La implementación del descriptor perímetro en forma secuencial ha sido realizada mediante un barrido de la imagen segmentada completo a fin de identificar todos los puntos del perímetro de forma correcta para cada región.

Implementación Paralela C#

En este caso también presentamos la versión paralela híbrida donde realizamos el mismo proceso que en los descriptores anteriores, calculando de forma paralela por filas y regiones para, una vez finalizado el proceso, realizar un proceso de agrupación de resultados para obtener el resultado final.

Implementación Paralela CUDA

La implementación CUDA sigue el patrón establecido en las anteriores, realizando los pasos de inicialización, proceso y paso de información para la compactación posterior de resultados.

11.2.1.4. Densidad

La implementación del cálculo de la densidad es un proceso mucho más sencillo y rápido, ya que depende totalmente de los descriptores área y perímetro anteriormente calculados. De esta manera en su forma secuencial solamente realiza un número de divisiones igual al número de regiones contenidas en la imagen.

Implementación Paralela C#

Esta implementación paralela resulta extremadamente sencilla de acuerdo a la implementación secuencial, por lo que se usa el método de paralelización directa sobre segmentos, la única opción por la naturaleza del descriptor.

Implementación Paralela CUDA

Para la implementación CUDA no es necesario realizar grandes procesos de cómputo, por lo que resulta un kernel de ejecución realmente simple, donde además no es necesario compactar resultados a su finalización. En este caso se pueden aprovechar los resultados anteriores, en caso de haberlos mantenido en la memoria del device.

11.2.1.5. Volumen

El descriptor volumen necesita de la imagen original, por lo que es el primer descriptor que hará uso de ella. Su implementación secuencial realiza un barrido por la imagen original realizando la suma expuesta por el algoritmo.

Implementación Paralela C#

Con este descriptor se vuelve a optar por el método de paralelización híbrida expuesto anteriormente, con la necesidad de la compactación de resultados a la finalización del procesamiento paralelo.

Implementación Paralela CUDA

Para realizar la implementación sobre el dispositivo gráfico antes hay que recordar cargar en la memoria gráfica la imagen original, junto a la segmentada ya cargada, en caso de que no se hubiese realizado con anterioridad. Para este descriptor volvemos al esquema de ejecución híbrido usado anteriormente, con la compactación de resultados al final del procesamiento paralelo.

11.2.1.6. Volumen^2

El descriptor volumen^2 es una variación del anterior, por lo que su procesamiento es idéntico teniendo en cuenta las variaciones en la sumatoria propuesta, por lo que no detallará de forma explícita.

11.2.1.7. Diámetro equivalente

El diámetro equivalente vuelve a los pasos de la densidad, ofreciendo un algoritmo basado en una característica anterior donde el procesamiento se centra en los resultados ya calculados de las regiones. De esta manera, el procesamiento secuencial realiza tantas iteraciones como número de segmentos para completar el proceso.

Implementación Paralela C#

Se trata en este caso de un procesamiento paralelo realmente simple y corto en el número de elementos paralelos a ejecutar, además del abandono del enfoque híbrido por el de regiones debido a la naturaleza del descriptor.

Implementación Paralela CUDA

Procesamiento CUDA simple donde no se aprovecha el paralelismo hardware debido a la falta de paralelismo en el algoritmo, que se acota en el número de regiones de la imagen procesada. El abandono en esta implementación del algoritmo híbrido no hace necesaria la compactación de resultados.

11.2.2. Descriptores basados en los niveles de gris

Respecto a los descriptores basados en los niveles de gris, recordemos que se trataba del nivel de gris medio, el nivel mínimo y máximo de gris, la desviación típica y el baricentro de los niveles de gris. Todos estos descriptores harán uso de la imagen original junto a la segmentada, debido a que se centran en la información que esta proporciona en conjunto a la segmentación.

Se va a explicar como una serie de pequeños apartados, uno por descriptor, donde se enumerarán las características de la implementación sobre cada tecnología.

11.2.2.1. Nivel mínimo y máximo de gris

Los descriptores del nivel mínimo y máximo de gris no son más que el menor y mayor valor de gris de la región estudiada. Para la implementación secuencial se realiza un barrido total de la imagen, calculando el valor mínimo y máximo para cada región. Estos dos descriptores han sido implementados de manera conjunta.

Implementación Paralela C#

Volvemos con este descriptor al enfoque híbrido tan utilizado, donde es necesario calcular este valor mínimo y máximo por filas y segmentos para finalmente proceder a compactar resultados y devolver el resultado final.

Implementación Paralela CUDA

Con estos descriptores implementados de manera conjunta volvemos al enfoque híbrido en CUDA, generando los resultados y compactándolos tras el procesamiento paralelo para ofrecer ambas características de forma conjunta.

11.2.2.2. Nivel de gris medio

El nivel de gris medio no es más que la operación división entre el volumen y el área de cada región, por lo que su procesamiento secuencial es realmente simple y corto en el procesamiento.

Implementación Paralela C#

Se retrocede a un enfoque basado en el paralelismo de regiones con este descriptor dependiente únicamente de las características volumen y área anteriormente calculadas, por lo que no es necesaria compactación de resultados.

Implementación Paralela CUDA

Se vuelve al enfoque de regiones, donde solo es necesario el volumen y área de las regiones calculado anteriormente para el procesamiento paralelo del descriptor actual.

11.2.2.3. Desviación típica

El cálculo de la desviación típica es dependiente del área de la región y del valor de gris medio, pero aun así es también dependiente de la intensidad de cada pixel por lo que es necesario un barrido completo de la imagen para su implementación secuencial.

Implementación Paralela C#

Se introduce el enfoque hibrido en este descriptor debido a su dependencia de la intensidad de cada pixel, y se toma como parámetro el área de cada región y su valor de gris medio para el calculo de la desviación típica. Debido a que la dependencia con el área es para todo el valor de cada región, es necesario dividir el procesamiento en dos bloques, donde el primero realiza

el calculo del operando superior de la división y el segundo divide el conjunto de resultados compactados por el área de la región.

Implementación Paralela CUDA

Debido a la naturaleza del descriptor, es necesario usar tanto el enfoque hibrido como el enfoque por regiones. De esta manera primero se realiza el procesamiento del operador superior de forma paralela, se compactan resultados, y finalmente se realiza el procesamiento paralelo del conjunto de la ecuación para ofrecer el resultado final sin mayor compactación.

11.2.2.4. Baricentro de los niveles de gris

El cálculo del baricentro de los niveles de gris es equivalente al procesamiento realizado para la desviación típica. En la implementación secuencial se realiza un barrido generando el sumatorio de X e Y para cada región, y finalmente se realiza la división por el área de la región.

Implementación Paralela C#

Equivalente al descriptor anterior, se realiza primero un procesamiento del operador superior, en base a la intensidad de la imagen original, para posteriormente, una vez compactados los resultados, se procede a la división de cada resultado de región por su área, generando el resultado final.

Implementación Paralela CUDA

De forma totalmente semejante, se realiza un procesamiento en dos fases donde primero se procesa el primero operando, se compactan resultados, se devuelven al device y posteriormente se trae el resultado final tras el procesamiento paralelo en el device.

11.2.3. Descriptores basados en el rectángulo circunscrito y puntos extremos

En este punto se realiza un análisis conjunto de los descriptores basados en el rectángulo circunscrito y los puntos extremos de la región, debido a que su procesamiento común se beneficia del procesamiento de los primeros para obtener de forma simple los segundos.

11.2.3.1. Primer punto, ancho, alto, área y extendido del rectángulo circunscrito

Para calcular estos descriptores es necesario realizar un barrido total de la imagen, para posteriormente extraer tanto el ancho, como el alto, el área y el extendido.

Implementación Paralela C#

Realizamos un procesamiento híbrido en primer lugar obtener los puntos primeros y últimos de cada región para cada fila para, posteriormente, obtener los puntos que delimitan la región, los cuatro que necesitamos, y de forma paralela calcular todas las características del rectángulo circunscrito buscadas.

Implementación Paralela CUDA

Para realizar este procesamiento paralelo es necesario en primer lugar realizar una invocación de kernel con modelo híbrido para obtener los puntos más a la derecha e izquierda, primeros e últimos, de cada región por filas. Una vez tenemos esto, podemos obtener de forma directa, los puntos que delimitan la región en forma de Derecha superior, Superior izquierda, Inferior Izquierda e izquierda superior. Estos son los puntos más a la derecha de la región, más a la izquierda, más arriba y más abajo para, posteriormente con ellos, realizar un procesamiento por regiones paralelo y obtener de cada región el primer punto, el ancho y alto del rectángulo, su área y el extendido del mismo de forma directa y simple.

11.2.3.2. Superior izquierda, superior derecha, inferior izquierda, inferior derecha, izquierda superior, izquierda inferior, derecha superior y derecha inferior de la región

En base a lo calculado anteriormente, durante el procesamiento en que se obtenían todos los puntos necesarios, se añaden clausulas de comparación para, de forma conjunta obtener los cuatro puntos restantes no obtenidos.

Implementación Paralela C#

Durante el proceso en que se buscan los cuatro puntos más importantes para el calculo de los descriptores del rectángulo, se añaden cuatro clausulas extras de comparación que extraen de forma directa los puntos restantes, obteniendo los ocho puntos extremos incluso antes de conseguir las características del rectángulo circunscrito.

Implementación Paralela CUDA

Durante el paso host de compactación de resultados donde se obtienen cuatro puntos que coinciden directamente con cuatro de los ocho puntos extremos de la región, se añaden cuatro clausulas extras y así se obtienen los ocho puntos extremos de forma paralela y conjunta con los descriptores del rectángulo circunscrito.

11.3. Elementos de medición temporal

Para establecer durante la etapa de extracción de resultados unas medidas fiables y confiables se han usado las herramientas nativas ofrecidas por las distintas plataformas para medición de tiempos de ejecución.

Implementación C#

En la plataforma C# se han usado las estructuras de DateTime donde podemos extraer mediciones de tiempo a nivel de milisegundos, de forma que las medidas son totalmente fiables y de suficiente precisión para el caso que nos ocupa.

```
DateTime inicio = DateTime.Now;
//Procesamiento
DateTime final = DateTime.Now;
TimeSpan duracion = final - inicio;
double segundosTotales = duracion.TotalMilliseconds;
```

Fragmento 11.8 – Medición temporal en C#

Implementación CUDA

En este caso se han usado las directivas CUDA de grabación de eventos, sincronización de hilos, para evitar que se muestree un tiempo de finalización antes de la finalización real, y directivas de sincronización de eventos para asegurar que se ejecuta el evento de sincronización de tiempos cuando debe hacerlo.

```
cudaEventRecord(start, 0);
//Procesamiento
cudaThreadSynchronize();
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
cudaThreadSynchronize();
```

Fragmento 11.9 – Medición temporal en CUDA

Parte IV

Resultados

Capítulo 12

Experimentos y resultados

Durante el siguiente capítulo se van a presentar los distintos resultados obtenidos en las ejecuciones de los algoritmos implementados comentados anteriormente sobre las dos tecnologías objetivo CUDA y C# presentadas en los capítulos de conceptos teóricos. Para ello, primero haremos un repaso detallado del hardware de procesamiento objeto de los distintos experimentos, o ejecuciones. De la misma manera, se presentarán las distintas fuentes de imagen a usar como datos de entrada para estas pruebas.

12.1. Entorno de pruebas

Nuestro entorno de pruebas se compone de dos equipos de procesamiento totalmente diferenciados en cuanto a sus características. El primero de ellos es un ordenador personal al uso, equipado con un procesador de gama media y una tarjeta gráfica de características suficientes para pertenecer al mismo rango. Contando con que el equipo tiene una antigüedad de aproximadamente 3 años, cuenta con una capacidad de proceso a la altura de la mayoría de los equipos actuales de sobremesa. Por otro lado, contaremos con un equipo portátil, de gama media igualmente, con procesador y dispositivo gráfico de tipo móvil adaptado al bajo consumo, frente al rendimiento del equipo de sobremesa. La elección de los recursos resulta realmente importante pues, si bien no estamos en la gama más baja a nivel tecnológico, se ha intentado reunir dos plataformas de pruebas cuyas especificaciones no sobresalgan demasiado de la media de equipos consumidos en la actualidad. Es importante recalcar esto pues vamos a presentar una comparativa de dos tecnologías de procesamiento paralelo sobre las que se ha hecho especial hincapié en, sobre todo, su capacidad de procesamiento alejadas de estaciones o servidores de alto rendimiento, a los que estamos acostumbrados cuando se habla de ejecuciones paralelas de gran eficiencia.

Sistema de sobremesa

En cuanto al sistema de sobremesa, como hemos comentado se trata de un pc que tras tres años de vida ha caído a la gama media de procesamiento comparada con la tecnología existente. En la siguiente tabla podemos comprobar su composición.

Sistema Host	
Placa Base	Asus P6T
Memoria Ram	Kingston DDR3 1333 Mhz 2GB x 2 (PC3-10700)
Procesador	Intel Core i7 920
Disco Duro	Seagate 1TB(7200rpm)
Tarjeta Gráfica	NVIDIA Geforce GTX 260 – Asus ENGTX260 GL+
Sistema Operativo	Windows 7

Tabla 12.1 – Especificaciones Sistema Sobremesa

Si entramos más en detalle sobre cada una de las unidades de proceso, podemos ver las siguientes tablas de especificaciones.

CPU	
Denominación	Intel Core i7 920
Consumo Máximo	130W
Tecnología	45 nm
Frecuencia de Trabajo	2.67GHz
Número de Núcleos	4
Número de Hilos por procesador	2
Número de Hilos totales	8
Frecuencia Máxima Turbo (Solamente un procesador)	2.93Ghz
Cache	8MB
Velocidad del Bus Frontal	4.8 GT/s
Ancho de Banda de memoria máximo	25.6 GB/s
Velocidad de Memoria	1066 Mhz (DDR3)
Interfaz de Memoria	36 bits

Tabla 12.2 – Especificaciones del Procesador del Sistema Sobremesa

Tarjeta Gráfica	
Denominación	Asus EN260GTX GL+
Consumo Máximo	182W
Tecnología	55 nm
Número de Procesadores Shader	216 unificados
Número de Multiprocesadores	27
Número de procesadores por multiprocesador	8
Frecuencia de Trabajo del motor	576 Mhz
Frecuencia de los procesadores Shader	1242 MHz
Ancho de Banda de memoria máximo	113.7 GB/s
Velocidad de Memoria	1999 Mhz (DDR3)
Tamaño Memoria	896 MB
Interfaz de memoria	448 bits

Tabla 12.3 – Especificaciones del Dispositivo Gráfico del Sistema Sobremesa

Las principales diferencias entre las unidades de proceso que se van a utilizar destacan en el número de unidades de proceso, su velocidad, y la velocidad del acceso a memoria básicamente. En el caso de la CPU, contamos con pocos núcleos e hilos hardware, pero su velocidad de procesamiento es mucho más alta que la de los procesadores GPU, que por otro lado son muchos más, ordenados en una arquitectura multiprocesador, y con un acceso a memoria mucho más rápido, con lo que se reducen las latencias enormemente.

Todas las pruebas van a ser realizadas sobre este sistema tomándolo como un entorno normal de trabajo. Es decir, no se han desconectado pantallas de la tarjeta gráfica para liberarla de trabajo, con el fin de no darle ventaja, y tampoco se han desconectado servicios innecesarios de Windows. Se trata de un arranque normal de un ordenador personal, donde las únicas aplicaciones corriendo en primer plano serán las necesarias para realizar las ejecuciones, pero manteniendo el entramado en segundo plano intacto. De esta manera se pretende emular el funcionamiento en un entorno real, donde difícilmente se dispondrá de un equipo de forma exclusiva.

Sistema portátil

En cuanto al sistema portátil, se trata de una composición con unos 2 años de antigüedad, que se puede considerar también en la gama media de los equipos actualmente suministrados por los fabricantes. El equipo portátil va a ser usado de dos maneras, en primer lugar como entorno de pruebas tanto C# como CUDA de ejecución local, y en segundo lugar y haciendo uso de la herramienta NSight de CUDA sobre Visual Studio 2010, como entorno de ejecución remoto, siendo la ejecución programada desde el pc de sobremesa y ejecutada sobre este dispositivo portátil.

Sistema Host	
Equipo portatil	Sony Vaio VPCF13C5E
Placa Base	Sony VAIO sin especificar
Memoria Ram	Kingston DDR3 1333 Mhz 6GB (PC3-10700)
Procesador	Intel Core i5 560M
Disco Duro	Samsung 500TB (7200rpm)
Tarjeta Gráfica	NVIDIA Geforce GT 425M
Sistema Operativo	Windows 7

Tabla 12.4 – Especificaciones del Sistema Portátil

Si entramos más en detalle sobre cada una de las unidades de proceso, podemos ver las siguientes tablas de especificaciones.

CPU	
Denominación	Intel Core i5 560M
Consumo Máximo	35W
Tecnología	32 nm
Frecuencia de Trabajo	2.66GHz
Número de Núcleos	2
Número de Hilos por procesador	2
Número de Hilos totales	4
Frecuencia Máxima Turbo (Solamente un procesador)	3.2Ghz
Cache	3MB
Velocidad del Bus Frontal	2.5 GT/s
Ancho de Banda de memoria máximo	17.1 GB/s
Velocidad de Memoria	1066 Mhz (DDR3)
Interfaz de Memoria	36 bits

Tabla 12.5 – Especificaciones del Procesador del Sistema Portátil

Tarjeta Gráfica	
Denominación	NVIDIA Geforce GT 425M
Consumo Máximo	182W
Tecnología	40 nm
Número de Procesadores Shader	96 unificados
Número de Multiprocesadores	12
Número de procesadores por multiprocesador	8
Frecuencia de Trabajo del motor	560 Mhz
Frecuencia de los procesadores Shader	1120 MHz
Ancho de Banda de memoria máximo	25.6 GB/s
Velocidad de Memoria	800 Mhz (DDR3)
Tamaño Memoria	1024 MB
Interfaz de memoria	128 bits

Tabla 12.6 – Especificaciones del Dispositivo Gráfico del Sistema Portátil

En este caso podemos obtener las mismas conclusiones que el caso del equipo sobremesa en cuanto a las diferencias encontradas entre las características del procesamiento convencional frente al procesamiento gpu.

Como se comentó, este sistema se usará de forma doble, tanto para las pruebas en forma local como para las pruebas CUDA de forma remota, para así observar las diferencias que podemos encontrar en este tipo de procesamiento. El modelo ideal sería un procesamiento en local sobre una tarjeta gráfica de base CUDA no usada como dispositivo de visionado principal por el equipo, y usada solo y exclusivamente como elemento de procesamiento, para así poder observar el rendimiento real total de la misma. De igual manera, sería interesante hacer ejecuciones sobre los procesadores convencionales con sistemas operativos de baja carga computacional y preparados para legar todo la capacidad de cómputo al proceso paralelo, pero, reiteramos, no es ese el objetivo primordial.

12.2. Conjunto de datos de prueba

En cuanto al conjunto de imágenes fuente que conforman el conjunto de datos de prueba, se ha escogido un grupo de cuatro imágenes, siendo todas estas de distintos tamaños, formas y con características diferenciadas. Sobre cada una de ellas, se ha llevado a cabo un proceso de segmentación, y fruto de lo anterior se exponen a continuación las cuatro imágenes fuente y segmentadas que serán la base de los resultados próximos.

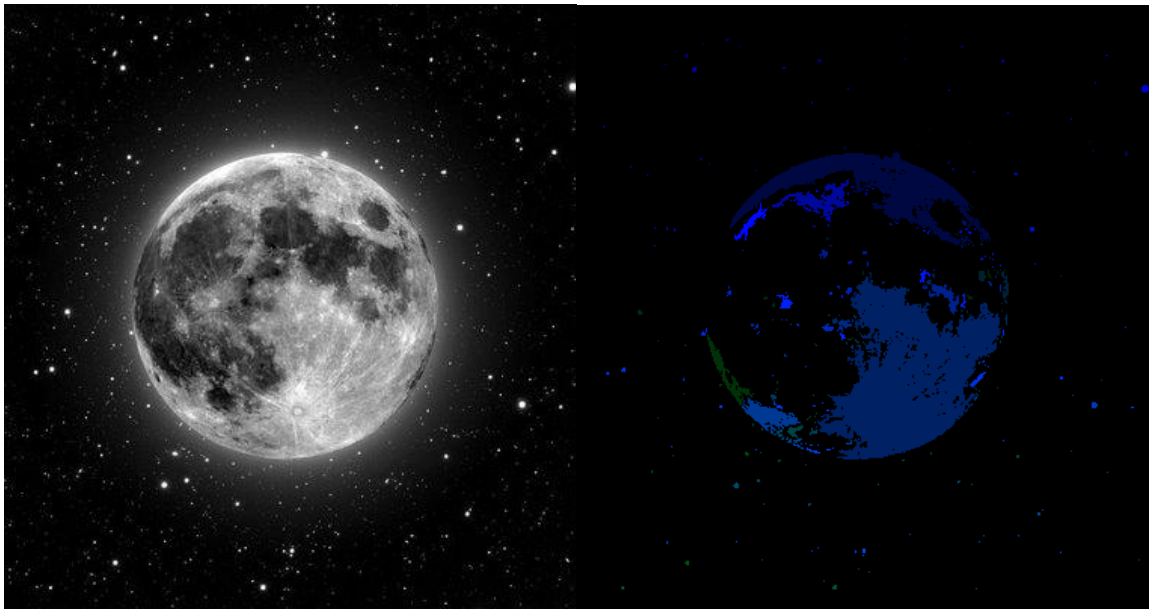


Ilustración 12.1 – Imagen fuente Luna y estrellas

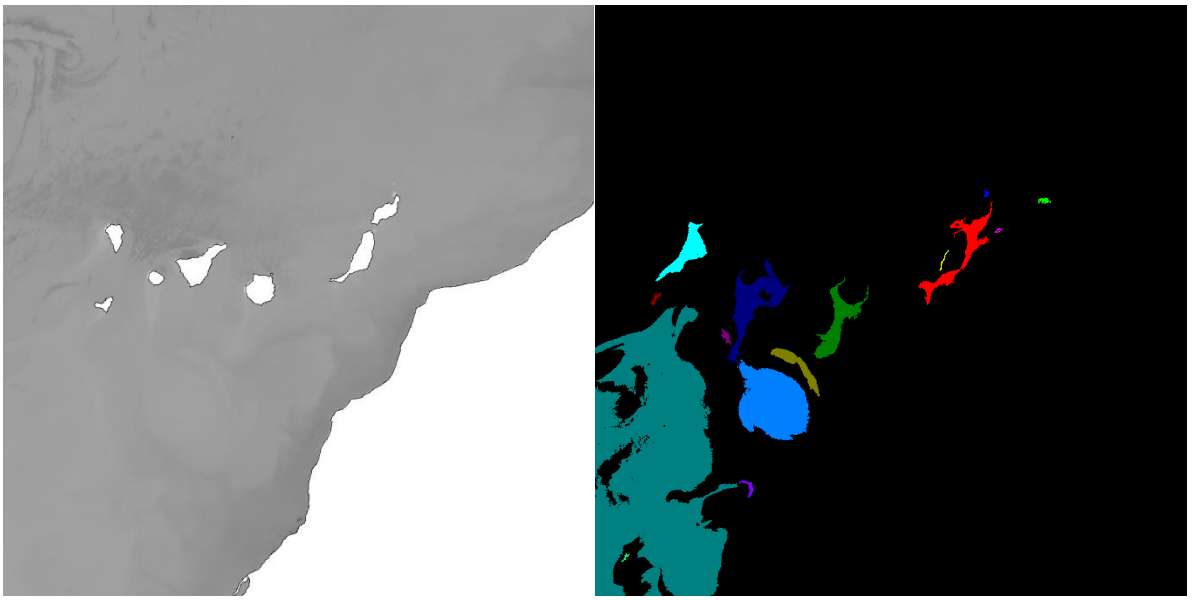


Ilustración 12.2– Imagen fuente Geo

Ilustración 12.3– Imagen fuente Lena

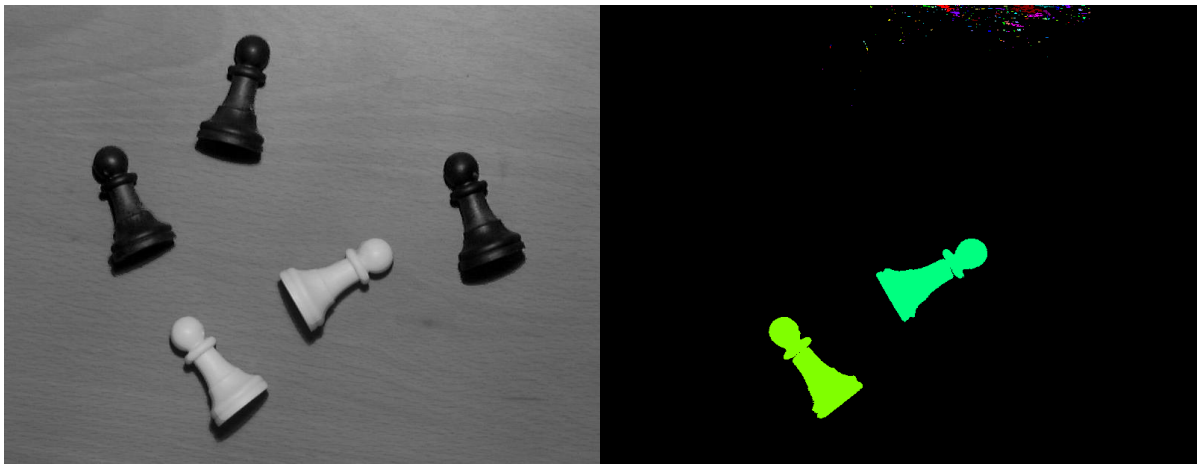


Ilustración 12.4– Imagen fuente Ajedrez

Como resumen, presentamos la siguiente tabla, donde damos cuenta de cada imagen exponiendo las distintas características interesantes, como el alto, ancho y número de regiones en su versión segmentada.

Imagen	Alto	Ancho	Nº Regiones
Luna y Estrellas	400	380	169
Geo	1024	1024	16
Lena	512	512	216
Ajedrez	615	800	185

Tabla 12.7 – Datos de las imágenes fuente usadas

12.3. Resultados

A continuación se presentan los resultados obtenidos de las distintas ejecuciones realizadas sobre las tecnologías empleadas, incluyendo tanto las ejecuciones paralelas como las segmentadas para cada una de las imágenes presentadas en el punto anterior. Estos resultados se agrupan en categorías, atendiendo a los grupos de descriptores usados durante los capítulos anteriores.

Para mostrar estos resultados haremos uso de tablas y gráficas donde mostraremos las distintas ejecuciones realizadas para cada imagen y cada descriptor de forma detallada.

Las distintas ejecuciones se han realizado sobre los equipos anteriormente mostrados, y siempre desde el entorno de desarrollo Visual Studio 2010, sobre una carga de trabajo del sistema operativo de un ordenador personal convencional. Todas las aplicaciones de uso común se encuentran cerradas, a excepción del entorno de desarrollo, pero se ha ejecutado un arranque del sistema completo, sin apagar ningún tipo de servicio ni característica ejecutada en un arranque convencional. Por lo tanto, estos resultados que van a exponerse dan una medida de las posibilidades reales de eficiencia y rendimiento en situaciones de uso común.

Para obtener estos tiempos se ha hecho un procesamiento en bucle de 1000 iteraciones, para extraer de esa manera el tiempo medio de cómputo para cada descriptor, expresando todos los tiempos finales en milisegundos.

12.3.1. Descriptores Simples

En este apartado, buscamos presentar de una forma detallada el comportamiento de los descriptores simples tanto paralelos como secuenciales ante el conjunto de imágenes de prueba definido anteriormente. Debido a la naturaleza del procesamiento sobre CUDA, vamos a realizar una evolución en la comparativa de tiempos de ejecución, desde el solo procesamiento, hasta el procesamiento y demás tareas de carga de memoria y sincronización, a fin de detallar los posibles puntos fuertes o débiles de las implementaciones.

12.3.1.1. Implementación C#

Comenzaremos mostrando los resultados sobre la base de pruebas que ha resultado la tecnología C# sobre el equipo sobremesa y, posteriormente, sobre el equipo portátil, incluyendo en los primeros casos implementados las distintas estrategias paralelas definidas en capítulos anteriores: paralelo por pixel, paralelo por segmentos y paralelo híbrido. De esta forma, se enfrentarán al procesamiento secuencial de los mismos descriptores, para posteriormente compararlos con la implementación de procesamiento gráfico y evaluar los resultados obtenidos.

Ejecución Sobremesa

En primer lugar, vamos a mostrar, en tablas separadas, los resultados para cada descriptor de las implementaciones realizadas en la tecnología C# y sus resultados en el equipo de sobremesa, tanto en secuencial como en paralelo.

Primer punto (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	44,0725	29,9917	1816,4438	16,7409
Geo	80,2045	133,1376	292,4467	20,3711
Lena	78,6544	53,9530	4060,5522	29,6216
Ajedrez	138,5379	93,5253	6185,2237	45,1325

Tabla 12.8 – Ejecución descriptor Primer Punto C# en Sobremesa

Siguiendo el patrón de documentación del capítulo anterior, vamos a comenzar por extraer todos los detalles posibles de estos primeros resultados sobre un descriptor, en este caso el

primer punto, para comentarlos en detalle, y dar algunas pinceladas que servirán de apoyo a los siguientes resultados presentados.

Si recordamos, este descriptor requería para ser calculado de un barrido de la imagen en su forma segmentada, hasta encontrar tantos primeros puntos como regiones contenidas en la imagen, por lo que se trata de un recorrido pixel a pixel de la misma. En base a eso, observamos la primera columna, donde podemos ver las cuatro imágenes fuente usadas en este procesamiento, y en la primera fila las distintas ejecuciones lanzadas. Se trata de las implementaciones secuencial, paralela por pixel, paralela por regiones de la imagen y paralela híbrida en base a filas y regiones de la imagen. Se trata por tanto de las implementaciones comentadas al detalle en el capítulo 11.

Centrándonos en los tiempos presentados por las distintas ejecuciones, podemos observar como en tres de los cuatro casos, la implementación paralela por pixel resulta de entrada más ventajosa que la implementación secuencial, salvo en el caso de la fuente Geo. Volviendo la vista a la tabla 12.7 podemos ver que se trata de la imagen con mayor tamaño y menos número de regiones. Eso afecta en esta ejecución paralela por pixel al número de hilos lanzados de forma unísona a procesamiento, motivo que se presenta como favorito para la explicación de este aumento del tiempo de procesamiento, añadido al hecho de que se efectúa un cerrojo sobre la variable compartida que provoca una entrada secuencial al mismo.

La siguiente columna de resultados nos ofrece una implementación paralela por regiones de la imagen, que arroja un balance realmente negativo a todas luces. Nos encontramos con un procesamiento de lentitud exagerada, debido quizás a que contamos con tantos hilos como regiones de la imagen, realizando un procesamiento de la imagen por separado, lo que genera grandes tiempos de procesamiento. Es interesante observar como es ahora la fuente Geo la que presenta menor tiempo de procesamiento, fruto probablemente del menor número de regiones presente en la misma.

El último conjunto de resultados contenido en la última columna pertenece a la implementación paralela por filas y regiones, o implementación híbrida como también ha sido denominada. En este caso, comparando con las implementaciones paralelas anteriores, se observa una mejora interesante, siendo el speedup de más de 100 en el caso paralelo por regiones y de hasta 6 en el caso paralelo por pixel. Esto la posiciona como la mejor opción viable de implementación paralela para competir contra la implementación secuencial de la misma. En este caso, podemos observar un speedup que va desde 2,7 a 3,9, según la imagen fuente usada para realizar la comparativa.

Por tanto, en base a los resultados observados, la mejor implementación para el cálculo del área es la implementación paralela por fila y región, produciéndose además una mejora sustancial sobre el caso secuencial estudiado.

Área (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	41,5223	64,3636	150,4686	11,2806
Geo	57,8133	101,2257	129,1673	16,9309
Lena	73,9242	114,8565	330,9789	20,7811
Ajedrez	128,0473	199,0313	532,6704	35,2320

Tabla 12.9 – Ejecución descriptor Área C# en Sobremesa

Al igual que el descriptor anterior, este requería un barrido de la imagen que, en este caso, si se hará completo en cualquier caso, pero que conlleva menor procesamiento, pues solo es necesario realizar un acumulado del número de píxeles por región de la imagen segmentada fuente. Por tanto operamos sobre la imagen segmentada.

En cuanto a los tiempos visibles, podemos observar como, a diferencia del descriptor anterior, el caso paralelo por pixel no mejora en manera alguna a la implementación secuencial realizada. Esto puede ser debido a que el proceso de creación y ejecución del conjunto de hilos necesario, unido al procesamiento simple en base de conteo realizado por cada uno de ellos y al acceso serializado al recurso compartido. En cuanto al caso paralelo por región, podemos ver como, de nuevo, se trata del peor caso paralelo, sea cual sea la referencia de imagen fuente que tomemos. Se produce un procesamiento lento en este caso, movido por que se produce una multiplicación de hilos realizando una tarea pesada, como es el barrido de una imagen. De igual manera, el resultado de mayor productividad no es otro que el paralelo por fila y región, exhibiendo una mejora sustancial tanto frente a otras implementaciones paralelas como a la implementación secuencial. Concretamente, se produce un speedup de 3,6 de media, teniendo en cuenta a todos los casos de prueba.

Perímetro (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	42,0724	31,2417	133,2276	11,7006
Geo	267,8953	67,1838	301,6872	63,0836
Lena	71,2240	32,0718	280,4960	20,1111
Ajedrez	131,7075	49,2428	452,3458	33,8919

Tabla 12.10 – Ejecución descriptor Perímetro C# en Sobremesa

Este descriptor vuelve a necesitar un barrido completo de la imagen segmentada, añadiendo un procesamiento extra fruto del cálculo del tipo de borde que genera el perímetro.

En base a los resultados presentados, podemos ver como, vuelta al primer descriptor, una implementación secuencial por pixel mejora a una implementación secuencial del descriptor, en algunos casos en mayor medida y en otros en menor, pero existe una mejora continua. El caso paralelo por región demuestra ser, valga la redundancia, un caso perdido en cuanto a eficiencia y rendimiento, exhibiendo unos tiempos de procesamiento que, en el mejor de los casos, quedan relativamente cerca, pero por encima, de los arrojados por una ejecución secuencial.

Como viene siendo habitual, es la implementación híbrida la que ofrece los mejores resultados, mejorando por mucho la implementación secuencial y las implementaciones paralelas. En este caso el speedup respecto a la implementación secuencial se consigna en 3,8 de media.

Densidad (ms)		
Imagen	***Secuencial***	***Paralelo por región***
Luna y Estrellas	0,0100	2,0001
Geo	0,0100	1,0001
Lena	0,0200	2,0001
Ajedrez	0,0100	2,0001

Tabla 122.11 – Ejecución descriptor Densidad C# en Sobremesa

Este es un caso interesante y distinto a los anteriormente comentados, por lo que servirá de base para los siguientes descriptores cuyo procesamiento es similar. En este caso, lo que necesitamos para poder calcularlo es exclusivamente el resultado de dos descriptores anteriores, por lo que su procesamiento se centra en realizar una operación por región de la imagen segmentada. Por tanto se trata de un procesamiento relativamente simple, ante un conjunto de datos significativamente más reducido que el conjunto de la imagen segmentada.

En este caso solo aparecen dos implementaciones, secuencial y paralela por región, debido a que por las fuentes usadas, no tiene sentido otro tipo de paralelización.

Por ello, cuando observamos los tiempos, vemos claramente como cualquier intento de implementación paralela se topa de frente con unos resultados realmente inalcanzables por parte de la implementación secuencial. Esto es así debido a que el proceso de creación, ejecución y, de forma general, manejo de hilos para la implementación paralela resultan más costosos que el procesamiento en sí realizado, por lo que no es viable para un pequeño conjunto de datos.

Volumen (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	42,6624	85,7149	131,8975	13,2507
Geo	67,4138	131,0174	181,4403	20,4511
Lena	76,0743	148,7885	275,4457	22,7913
Ajedrez	133,7176	242,3338	441,9652	37,5621

Tabla 12.12 – Ejecución descriptor Volumen C# en Sobremesa

El descriptor volumen necesita para su procesamiento tanto de la imagen original como de la imagen segmentada, debido a que se basa en un sumatorio de las intensidades de los pixeles contenidos en la región estudiada. Por tanto para su procesamiento es necesario un barrido total de la imagen segmentada, y un acceso constante a la imagen original.

En este caso volvemos a contar con unas implementaciones paralelas por pixel y región que arrojan tiempos de procesamiento medio muy superiores a la implementación secuencial realizada. Por tanto, nos centramos en la implementación por fila y región, cuyo rendimiento

mejora de forma significativa al procesamiento secuencial, arrojando un speedup de 3,3 de media.

Volumen2 (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	44,4725	94,3153	137,4978	15,2408
Geo	80,3745	144,2982	198,1713	24,0713
Lena	79,3645	154,7588	276,0357	24,1613
Ajedrez	139,7679	255,2846	451,4658	39,2922

Tabla 12.13 – Ejecución descriptor Volumen Cuadrático C# en Sobremesa

El descriptor del volumen cuadrático es una modificación del descriptor volumen donde el sumatorio se realiza sobre la intensidad al cuadrado, por lo que tanto el procesamiento como las necesidades de procesamiento son las mismas.

En cuanto a tiempos se mantiene la tónica del descriptor anterior al que modifica, como no podía ser de otra manera, por lo que la única opción paralela válida en cuanto a rendimiento es la paralela por fila y región. Esta consigue arrojar un speedup sobre la implementación secuencial de 3,3 de media, como en el caso anterior.

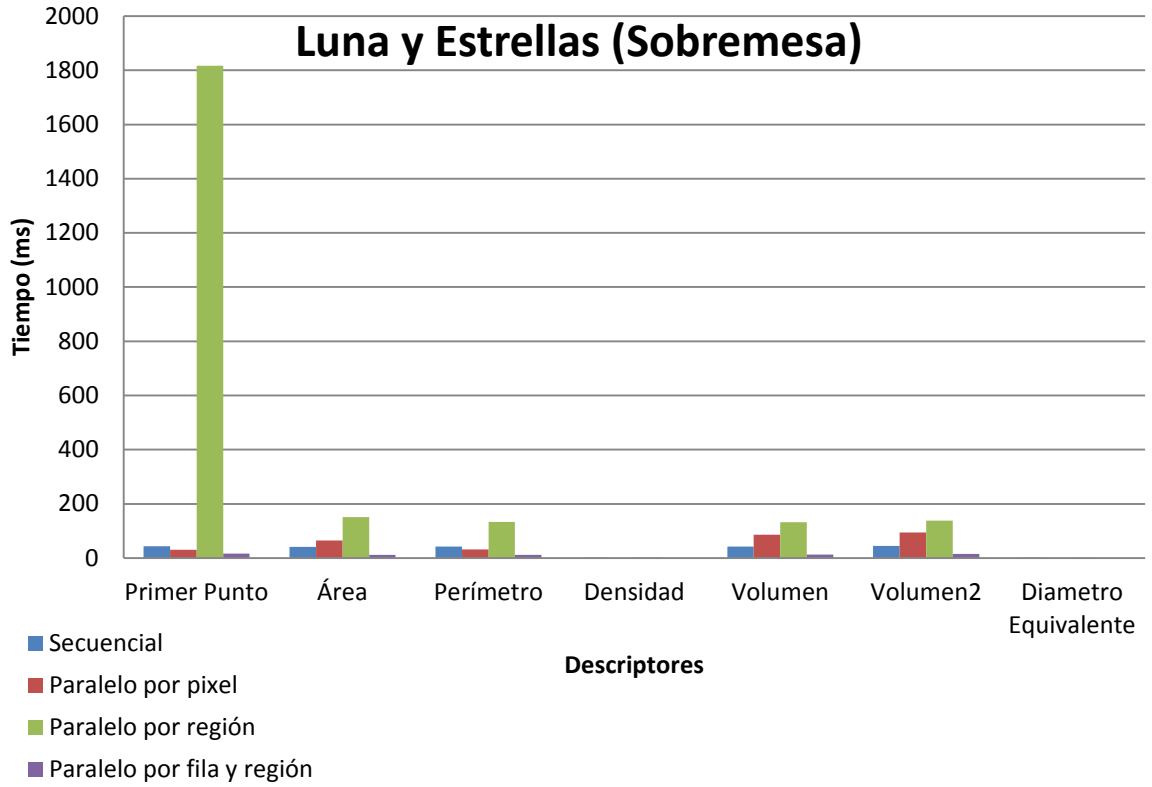
Diámetro Equivalente (ms)		
Imagen	***Secuencial***	***Paralelo por región***
Luna y Estrellas	0,0200	0,0500
Geo	0,0100	0,0100
Lena	0,0100	0,0200
Ajedrez	0,0100	0,0300

Tabla 12.14 – Ejecución descriptor Diámetro Equivalente C# en Sobremesa

Para finalizar los descriptores simples en la implementación C# sobre el equipo de sobremesa nos encontramos con el descriptor de diámetro equivalente, que presenta las mismas características en cuanto a necesidad que la densidad, por lo que no es necesario más que un par de los descriptores anteriormente calculados.

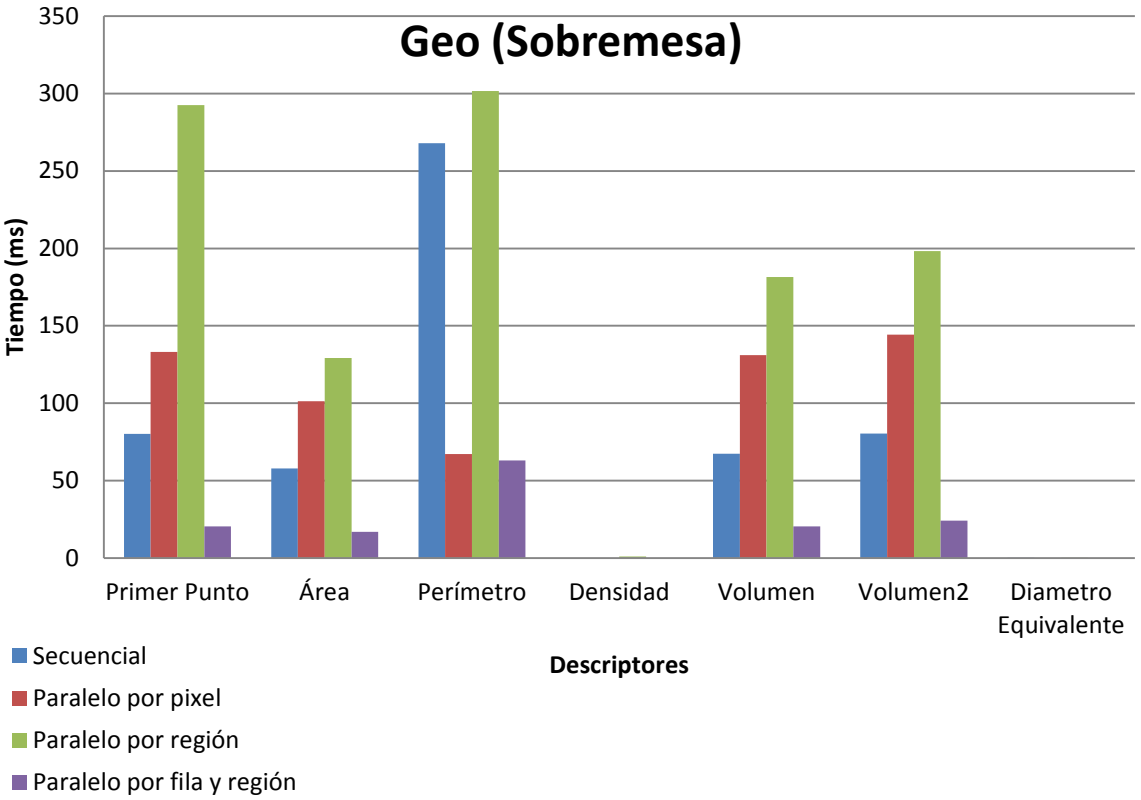
En base a esto, y revisando los tiempos conseguidos, podemos ver como no es viable una implementación paralelo para mejorar una implementación secuencial, al menos no con un número de regiones tan bajo como el que estamos manejando en estas fuentes.

Como resumen de todo lo expuesto anteriormente, se presentan algunas gráficas, organizadas por fuente, donde podemos ver cada uno de los descriptores, con sus correspondientes implementaciones, y sus tiempos de ejecución para cada una de las imágenes fuente usadas.



Gráfica 132.1 – Descriptores Simples C# en Sobremesa sobre Luna y Estrellas

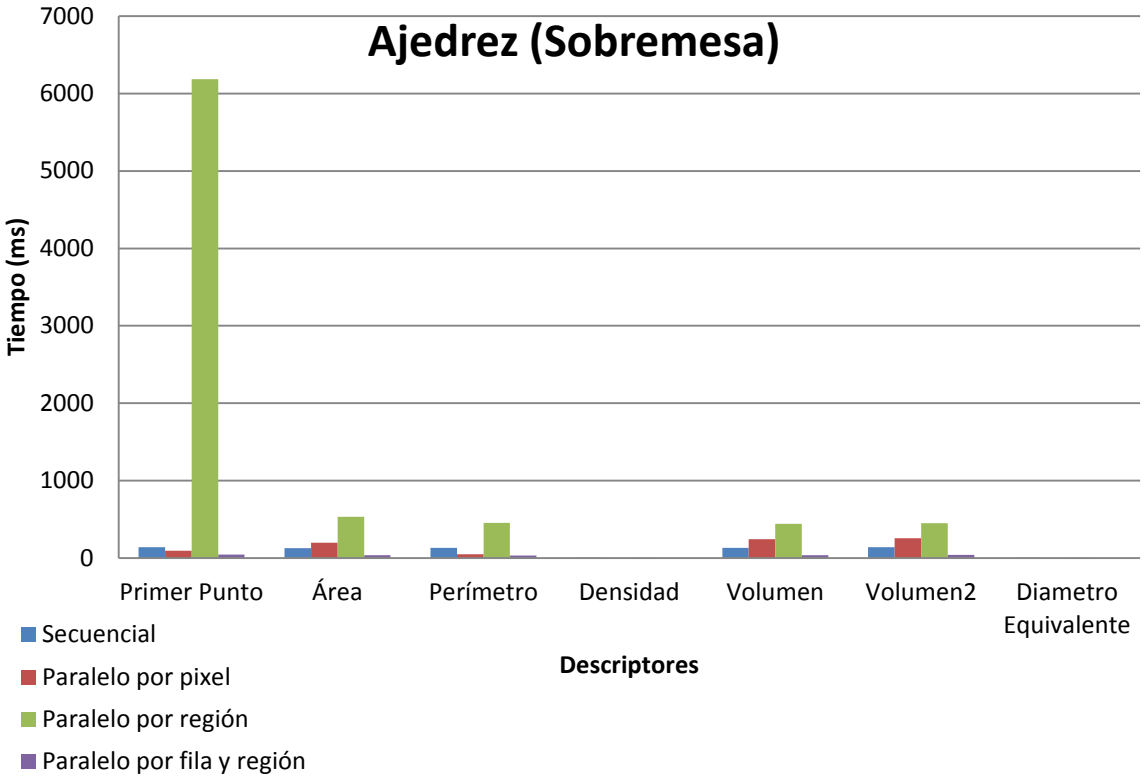
De esta forma podemos ver más claramente como, para el caso de la fuente Luna y Estrellas, los menores tiempos de ejecución de forma general son arrojados por la implementación paralela por fila y región, mientras que el peor caso lo arroja de forma constante la implementación por región. De igual manera, se ve como las veces en que la implementación paralela por pixel mejora a la implementación secuencial son menos que a la inversa, por lo que estas dos opciones paralelas que conforman la evolución hacia la paralelización híbrida quedan descartadas de forma abrumadora. También podemos observar, de forma simbólica, los casos densidad y diámetro equivalente, donde sus tiempos de ejecución son tan pequeños, sobre en comparación con la paralelización por región del primer punto, que pasan desapercibidos.



Gráfica 12.2 – Descriptores Simples C# en Sobremesa sobre Geo

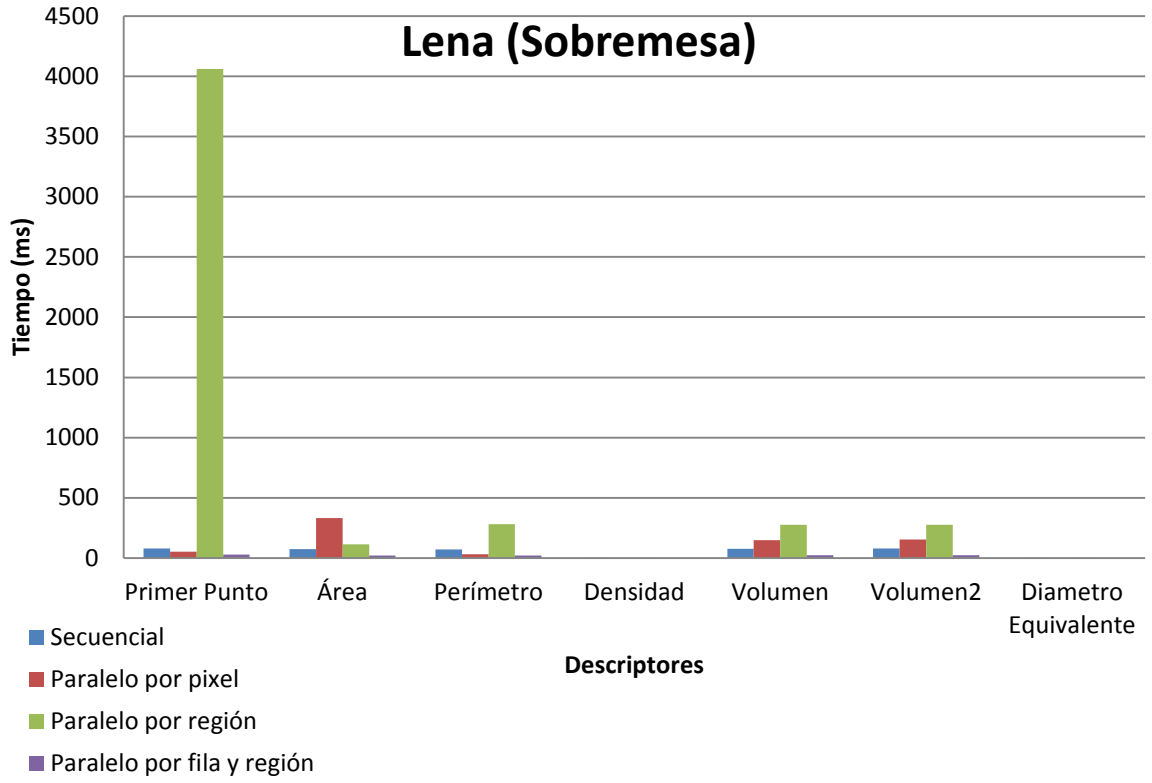
Igual que en el caso anterior, puede observarse como el peor parado es de forma constante es el caso paralelo por región, seguido de cerca por el paralelo por pixel. Se confirma, de forma clara, que no se trata de implementaciones que ofrezcan una gran eficiencia o rendimiento, y que hay que descartarlas en un enfrentamiento secuencial frente a paralelo, so pena de perder estrepitosamente en la mayoría de los casos.

Como pudimos ver en las tablas anteriores, y comprobamos de forma visual en la gráfica 12.2, la implementación híbrida de procesamiento paralelo por fila y región es la que ofrece mejores tiempos de ejecución medios y mejor rendimiento general en cualquier descriptor en que haya sido aplicada.



Gráfica 42.3 – Descriptores Simples C# en Sobremesa sobre Ajedrez

Como en los casos anteriores, podemos ver la misma tendencia bajista para el rendimiento de las alternativas paralelas, a excepción de la paralela por fila y región, y como esta es la única que presenta batalla, realizando un gran papel, a la implementación secuencial, y arrojando un speedup continuo frente a ella.



Gráfica 12.4 – Descriptores Simples C# en Sobremesa sobre Lena

Visualmente no obtenemos datos que cambien un ápice lo comentado en las gráficas anteriores, por lo que no es necesario comentar nada más al respecto, sirva como nuevo ejemplo del rendimiento de los distintos algoritmos implementados.

Ejecución Portátil

Ahora vamos a realizar un análisis parecido para el caso del entorno de pruebas portátil, pero obviando cual comentario relativo al algoritmo en sí y centrándonos en el efecto que produce pasar de una plataforma sobremesa de alto rendimiento a un procesador de perfil bajo para un dispositivo preparado para la movilidad, al menos en teoría. Se aprovechará este apartado para hacer una breve reseña sobremesa frente a portátil.

Primer punto (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	42,1200	34,3200	3517,8061	28,0800
Geo	70,3001	176,2803	564,7209	37,4400
Lena	73,3201	59,2801	7936,6339	45,2400
Ajedrez	127,9202	102,9601	11715,7205	74,8801

Tabla 12.15 – Ejecución descriptor Primer Punto C# en Portátil

Manteniendo lo obtenido en la implementación sobremesa, obtenemos una mejora de rendimiento frente a la implementación secuencial por parte de la implementación paralela por pixel en tres de las fuentes, exactamente las mismas. En cuanto a la diferencia de tiempos entre portátil y sobremesa, el equipo móvil presente una mejora en cuanto al tiempo

secuencial, pero cae estrepitosamente en lo relativo al rendimiento paralelo, donde es barrido por el entorno de pruebas de sobremesa. El speedup de sobremesa frente a portátil en este plano para la implementación por fila y región, la de mejor rendimiento, es de 1,7 de media, con lo que podemos comprobar que el cambio de arquitectura se deja notar. Es interesante comentar que el hecho de que la arquitectura portátil quede por encima de la sobremesa en cuanto a la implementación secuencial tiene su causa en las tablas de definición de ambos procesadores mostradas al inicio del capítulo. Si consultamos las tablas 12.2 y 12.5 podemos observar como la frecuencia de trabajo base del procesador portátil es igual a la del procesador de sobremesa, pero la frecuencia turbo es 300 Mhz superior en el modelo portátil. Dado que estamos haciendo uso de una implementación secuencial, todo el procesamiento recae sobre un solo hilo, que se ejecuta en un solo procesador, por lo que se activa el modo turbo del procesador Intel y se produce ese aumento de rendimiento que arroja estos resultados. En cuanto a porqué, entonces, es más rápido el procesador de sobremesa en el caso paralelo, podemos atender al número de unidades de procesamiento, o núcleos, de cada procesador. Si hacemos esto veremos que el procesador de sobremesa dobla los núcleos del procesador portátil, y ya que a frecuencia de trabajo normal están igualados, esto unido a la arquitectura sobremesa de i7 frente a i5 consigue la mejora que podemos observar en cuanto a rendimiento se refiere.

Ya que las tablas siguientes han sido comentadas al detalle para el caso sobremesa, solo se harán comentarios en las que presenten datos interesantes o anómalos susceptibles de una revisión más profunda.

Área (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	37,4400	65,5201	285,4805	26,4601
Geo	53,0400	127,9202	191,8803	32,7600
Lena	70,2001	109,3002	622,4410	37,4400
Ajedrez	118,5602	187,2003	984,3617	63,9601

Tabla 12.16 – Ejecución descriptor Área C# en Portátil

Perímetro (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	39,0000	35,9800	241,8004	18,7200
Geo	243,3604	127,9202	347,8806	121,6802
Lena	63,9601	40,5600	521,0409	34,3200
Ajedrez	123,2402	76,4401	834,6014	62,4001

Tabla 12.17 – Ejecución descriptor Perímetro C# en Portátil

Densidad (ms)		
Imagen	***Secuencial***	***Paralelo por pixel***
Luna y Estrellas	0,0100	0,0321
Geo	0.0100	0.0201
Lena	0,0156	0,0312
Ajedrez	0,0100	0,0312

Tabla 152.18 – Ejecución descriptor Densidad C# en Portátil

En este descriptor ocurre algo curioso. Para el caso secuencial los valores se mantienen como en la implementación de sobremesa, pero los valores de procesamiento paralelo se alejan de los valores del experimento sobremesa, consiguiendo acercarse muchísimo más a los tiempos ofrecidos por la implementación secuencial. Según el razonamiento anterior esto no debería ser posible, debido a la capacidad de proceso paralelo para el caso sobremesa es mucho mayor. Este hecho debe estar causado por el bajo tamaño del conjunto de datos a procesar, que es posible que beneficie a una arquitectura móvil que resulta menos compleja en cuanto a sus caminos de datos que una arquitectura sobremesa, pero no deja de ser una conjetura ante un resultado anómalo, que se repite en el caso del diámetro equivalente, presentado más abajo.

Volumen (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	40,5600	76,4401	224,6403	23,4000
Geo	65,5201	218,4003	185,6403	40,5600
Lena	71,7601	141,9602	488,2808	39,0000
Ajedrez	121,6802	248,0404	772,2013	68,6401

Tabla 12.19 – Ejecución descriptor Volumen C# en Portátil

Volumen2 (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	40,5600	81,1201	226,2003	23,4000
Geo	73,3201	238,6804	191,8803	46,8000
Lena	73,3201	145,0802	496,0808	42,1200
Ajedrez	129,4802	263,6404	786,2413	71,7601

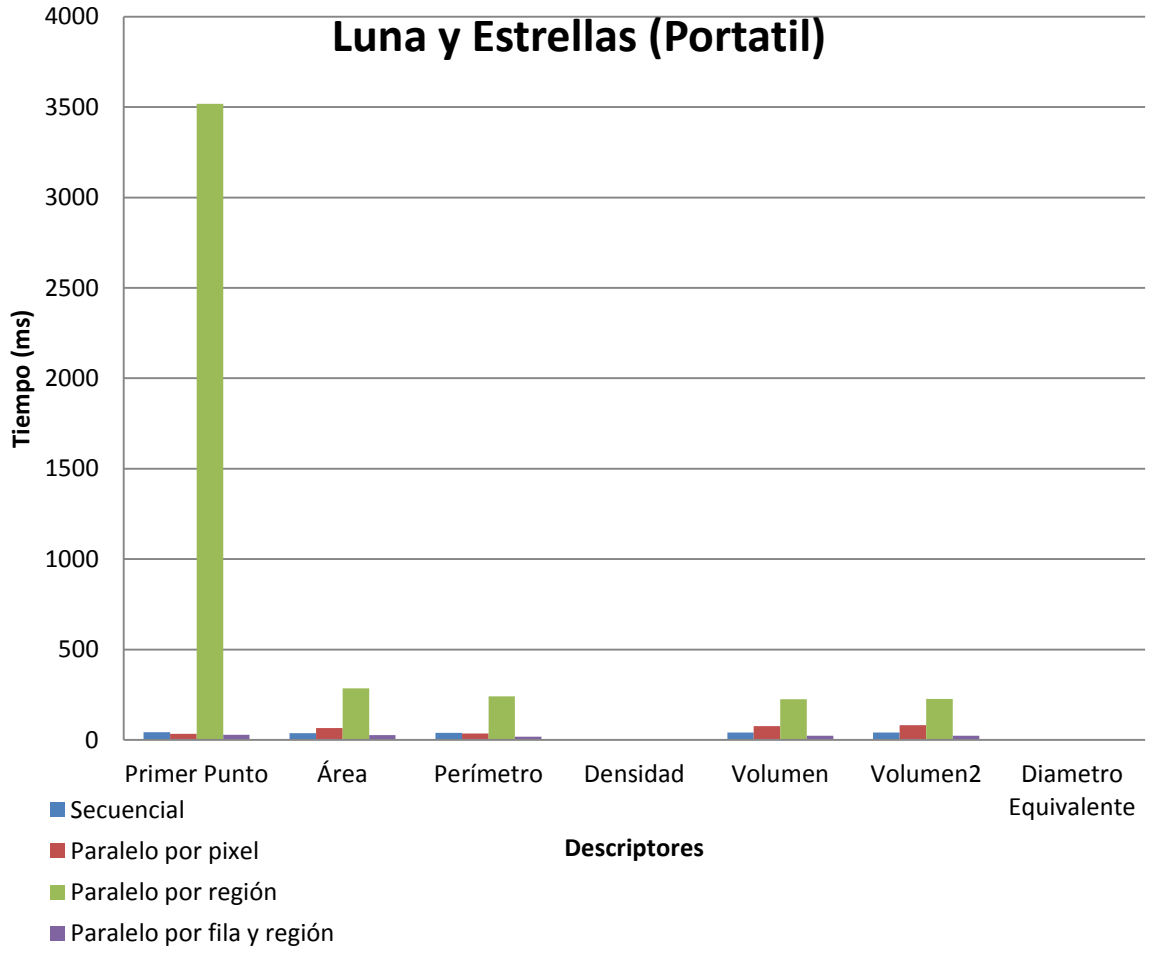
Tabla 62.20 – Ejecución descriptor Volumen Cuadrático C# en Portátil

Diámetro Equivalente (ms)		
Imagen	***Secuencial***	***Paralelo por pixel***
Luna y Estrellas	0,0200	0,0300
Geo	0,0100	0,0100
Lena	0,0100	0,0156
Ajedrez	0,0100	0,0156

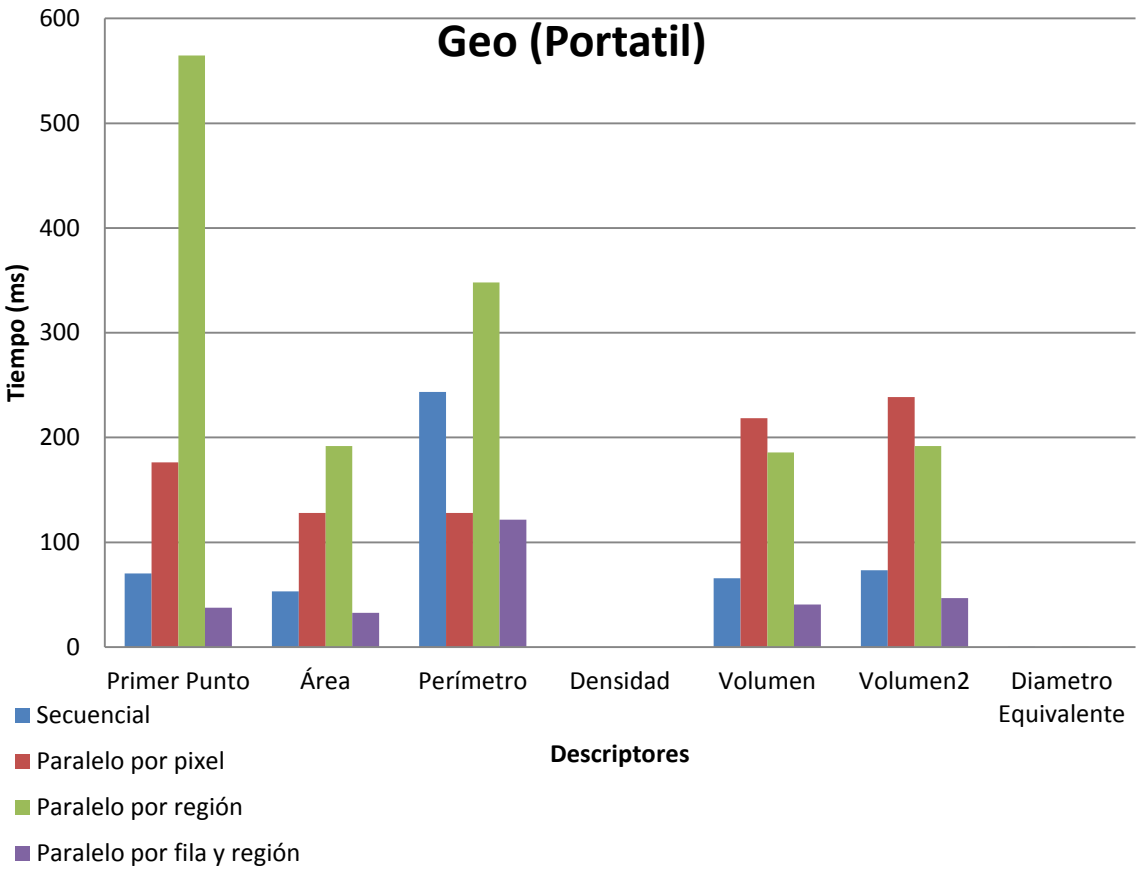
Tabla 12.18 – Ejecución descriptor Diámetro Equivalente C# en Portátil

Como podemos ver, en este descriptor se observan también esos resultados anómalos para el caso paralelo que observamos en la implementación paralela portátil frente a la implementación paralela sobremesa.

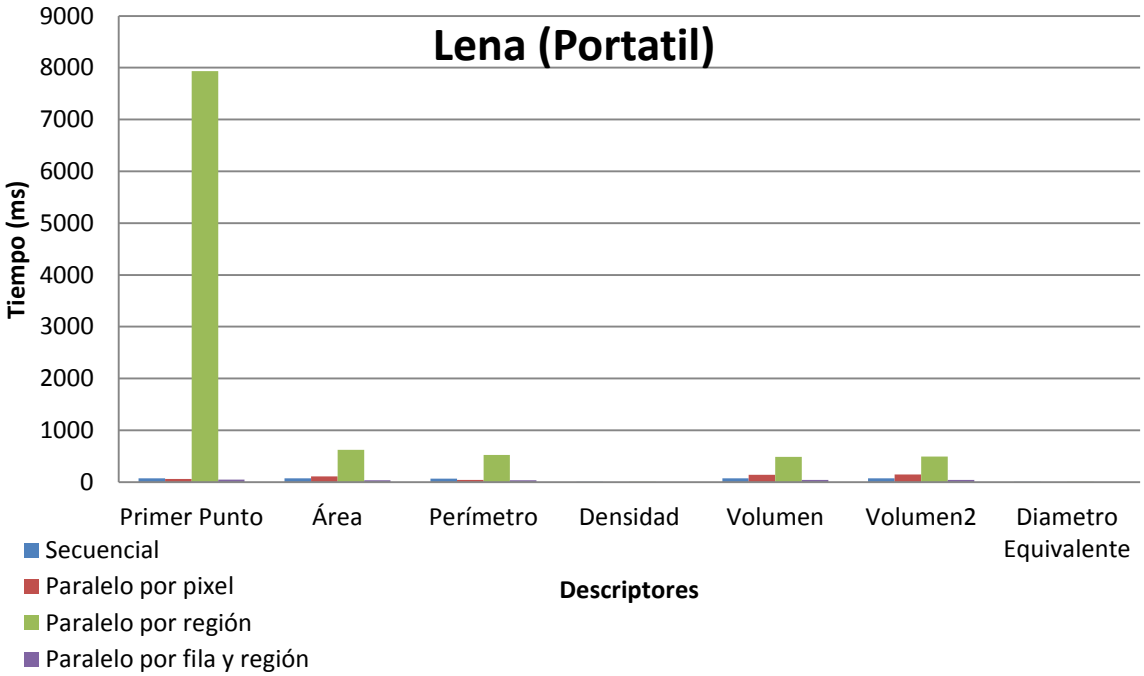
Al igual que para la implementación sobremesa, se presentan algunas gráficas que ayudan a resumir de forma visual todos los datos expuestos en las tablas anteriores. En este caso no van a ser comentadas debido a que exponen los mismos resultados obtenidos para el caso sobremesa pero con una breve mejora en los procesos secuenciales y un empeoramiento ya descrito para los casos paralelos.



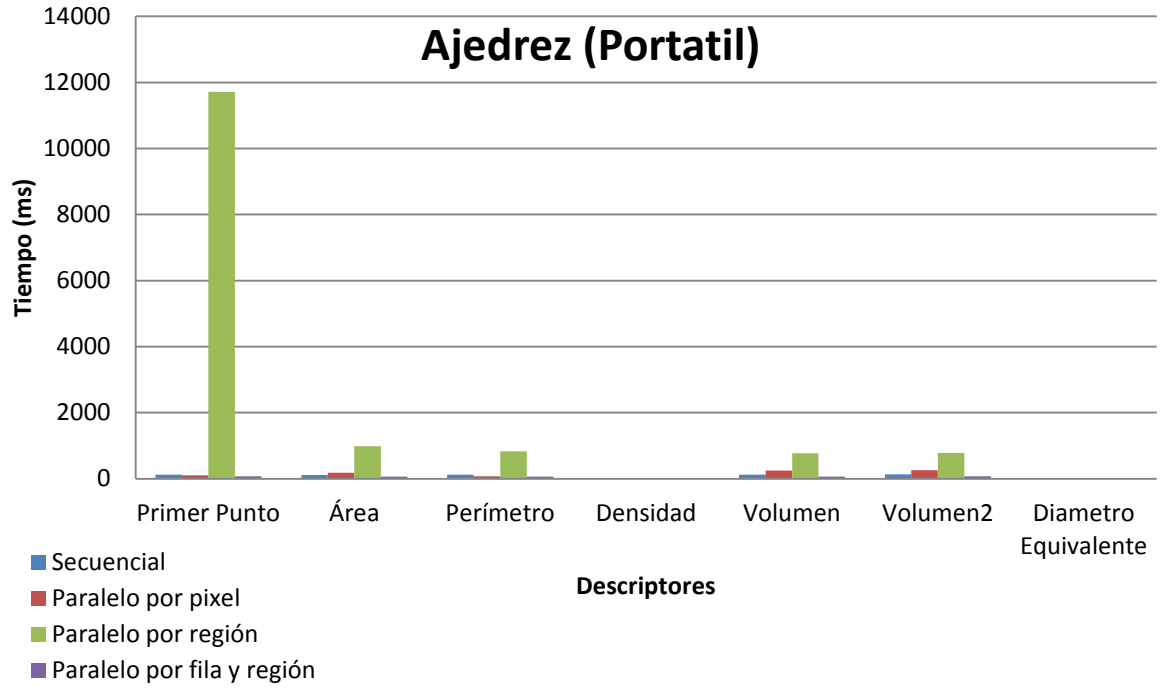
Gráfica 12.5 – Descriptores Simples C# en Portátil sobre Luna y Estrellas



Gráfica 12.6 – Descriptores Simples C# en Portátil sobre Geo



Gráfica 12.7 – Descriptores Simples C# en Portátil sobre Lena



Gráfica 12.8 – Descriptores Simples C# en Portátil sobre Ajedrez

12.3.1.2. Implementación CUDA

Una vez comentado a fondo el caso C# para los descriptores simples, pasamos a detallar los resultados de las pruebas realizadas sobre CUDA en base a las mismas arquitecturas de prueba y las mismas fuentes de datos. Debido a que en CUDA solo se ha optado por implementar una versión de paralelización por algoritmo, comentadas en detalle en el capítulo 11, nos encontraremos a continuación con una tabla resumen para cada plataforma. Más un detalle de ejecución realizando una división de tiempos de ejecución y de labores de memoria.

Ejecución Sobremesa

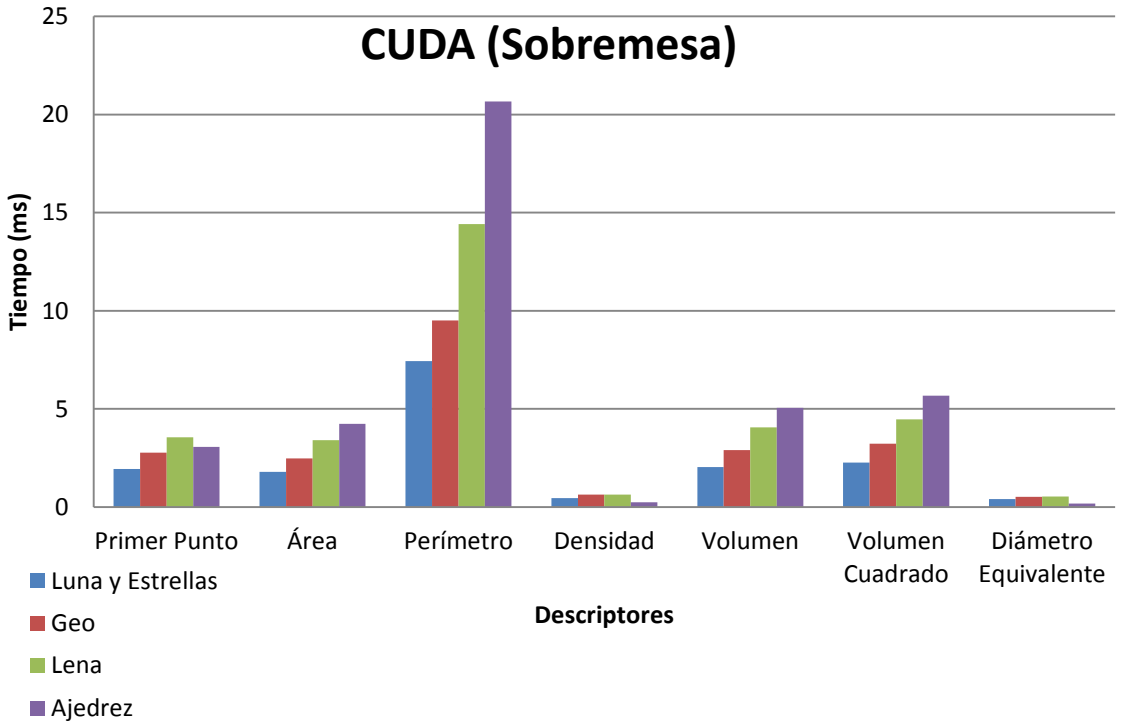
Para la plataforma sobremesa vamos a realizar dos análisis completamente distintos, en primer lugar un estudio de tiempos completos de ejecución para los distintos descriptores implementados en cada imagen fuente, con su correspondiente ayuda visual en forma de gráfica, para posteriormente presentar los datos resultado de dividir el tiempo de ejecución en procesamiento tanto de cpu como de gpu frente al tiempo usado para realizar gestiones de memoria, como el transporte de datos de la memoria host a device y viceversa.

CUDA Sobremesa (ms)							
Imagen	***Primer Punto***	***Área***	***Perímetro***	***Densidad***	***Volumen***	***Volumen Cuadrado***	***Diámetro Equivalente***
Luna y Estrellas	1,9395	1,7875	7,4366	0,4586	2,0385	2,2622	0,4095
Geo	2,7651	2,4736	9,5024	0,6326	2,8920	3,2233	0,5229
Lena	3,5568	3,4044	14,4244	0,6297	4,0629	4,4637	0,5331
Ajedrez	3,0617	4,2377	20,6684	0,2357	5,0564	5,6651	0,1789

Tabla 12.19 – Ejecución CUDA en Sobremesa

Viendo los resultados de la tabla 12.19, y si volvemos la vista atrás a la tabla 12.8, podemos ver como para el caso C# sobremesa, la mejor implementación del primer punto en la imagen Luna y Estrellas ofrecía un tiempo de ejecución de más de 16 ms, mientras que en este caso, el mismo descriptor sobre la misma fuente en CUDA ofrece un tiempo medio de ejecución de menos de 2 ms. En este caso estamos obteniendo un speedup de más de 8 puntos sobre la implementación paralela que usa un procesador convencional.

Si observamos las otras características vemos que el resultado es similar en todas ellas, salvo en el caso de la densidad y el diámetro equivalente, donde, por lo comentado en el caso paralelo C#, obtenemos menor rendimiento que en cualquier implementación secuencial típica.



Gráfica 12.9 – Descriptores Simples CUDA en Sobremesa

De forma gráfica, podemos observar como todos los descriptores ofrecen unos tiempos de ejecución realmente bajos, muy lejos de lo obtenido para los casos C#.

Ahora se hace necesario observar el mismo caso anterior pero tomando en consideración tanto los tiempos de ejecución como los tiempos de gestión de memoria, sobre los que obtenemos los siguientes datos.

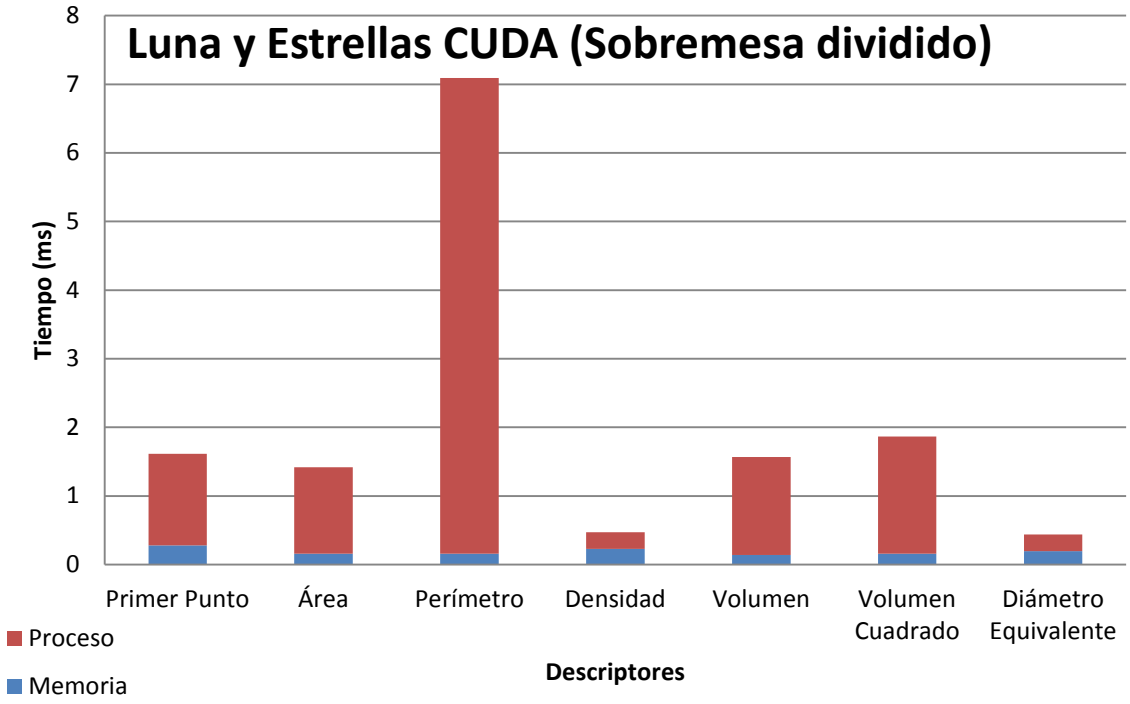
CUDA Sobremesa (ms)							
Imagen	***Primer Punto***	***Área***	***Perímetro***	***Densidad***	***Volumen***	***Volumen Cuadrado***	***Diámetro Equivalente***
L. y E. Memoria	0,2816	0,1591	0,1594	0,2291	0,1407	0,1582	0,1965
L. y E. Proceso	1,3308	1,2603	6,9345	0,2400	1,4279	1,7056	0,2406
Geo Memoria	0,1175	0,0706	0,0696	0,1378	0,0628	0,0631	0,1196
Geo Proceso	2,8671	4,1772	20,5333	0,1010	4,8599	5,5333	0,1042
Lena Memoria	0,4106	0,2192	0,2363	0,2915	0,2292	0,2233	0,2623
Lena Proceso	1,7605	1,6363	8,7582	0,2915	1,9768	2,2464	0,2992
Ajedrez Memoria	0,4653	0,2691	0,3420	0,3174	0,2475	0,2688	0,3029
Ajedrez Proceso	2,4644	2,5185	13,9822	0,2836	3,1303	3,5711	0,2941

Tabla 12.20 – Ejecución CUDA en Sobremesa a Trozos

Estructurado por filas, la primera fila de cada imagen fuente presenta los tiempos dedicados a memoria y la segunda fila muestra los tiempos de procesamiento. Aunque trabajamos en milisegundos y observemos tiempos de trabajo en memoria de 0,2 ms, tengamos en cuenta que para el caso del primer punto en Luna y Estrellas, por ejemplo, el tiempo de memoria supone un 18% del tiempo total de ejecución del descriptor. Lo indicado como tiempos de memoria, no son más que los tiempos de llevar y traer datos de la memoria host a la memoria device, mientras que lo etiquetado como proceso, se refiere a todo el tiempo de procesamiento, ya sea host o device, para llevar a cabo la tarea y generar un resultado.

De forma general, los tiempos de gestión de memoria oscilan entre 0,2 y 0,5 milisegundos, siendo siempre una parte inferior al 25% del tiempo total de ejecución del descriptor, salvo en dos casos muy concretos. En los descriptores Densidad y Diámetro Equivalente, que tantos resultados diferentes nos han ofrecido, podemos ver que el tiempo de gestión de memoria llega a superar en algunos casos al tiempo de procesamiento. Estos dos descriptores presentan un buen ejemplo de cuando no merece la pena realizar un proceso paralelo para su resolución, y menos ser llevados a una tecnología de procesamiento gráfico como esta.

Ahora se mostrará una gráfica de una de las fuentes a modo de ejemplo visual para ver claramente como se distribuye el tiempo de ejecución entre el tiempo de cómputo en procesador gráfico frente al tiempo de espera por gestión de la memoria.



Gráfica 12.10 – Descriptores Simples sobre Luna y Estrellas en CUDA en Sobremesa

De forma gráfica, podemos ver para la fuente Luna y Estrellas como el tiempo de proceso es muy superior al tiempo empleado en los procesos de memoria para todos los descriptores implementados, excepto para la densidad y el diámetro equivalente, donde se emplea casi el mismo tiempo en cada una de las dos tareas del procesamiento.

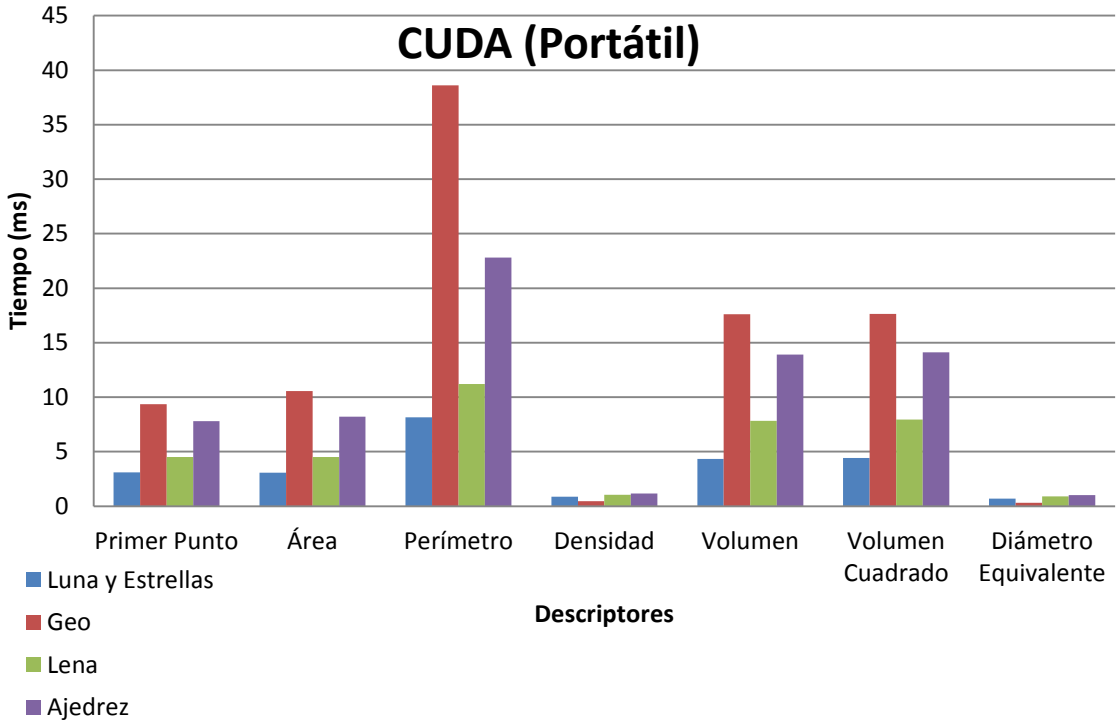
Ejecución Portátil

Para el caso de ejecución sobre portátil realizamos el mismo análisis y ejecuciones, obteniendo la siguiente tabla de resultados.

CUDA Portátil (ms)							
Imagen	***Primer Punto***	***Área***	***Perímetro***	***Densidad***	***Volumen***	***Volumen Cuadrado***	***Diámetro Equivalente***
Luna y Estrellas	3,1144	3,0742	8,1642	0,8711	4,3305	4,4255	0,6840
Geo	9,3641	10,5609	38,6001	0,4743	17,5993	17,6268	0,3201
Lena	4,5226	4,5235	11,2033	1,0553	7,8453	7,9642	0,8959
Ajedrez	7,8015	8,2251	22,8001	1,1570	13,921	14,121	1,0159

Tabla 12.21 – Ejecución CUDA en Portátil

Como podemos observar, se mantiene la misma tónica de resultados que en el caso sobremesa, pero con tiempos más elevados, llegando a estar estos por encima del doble en algunos casos.



Gráfica 12.11 – Descriptores Simples CUDA en Portátil

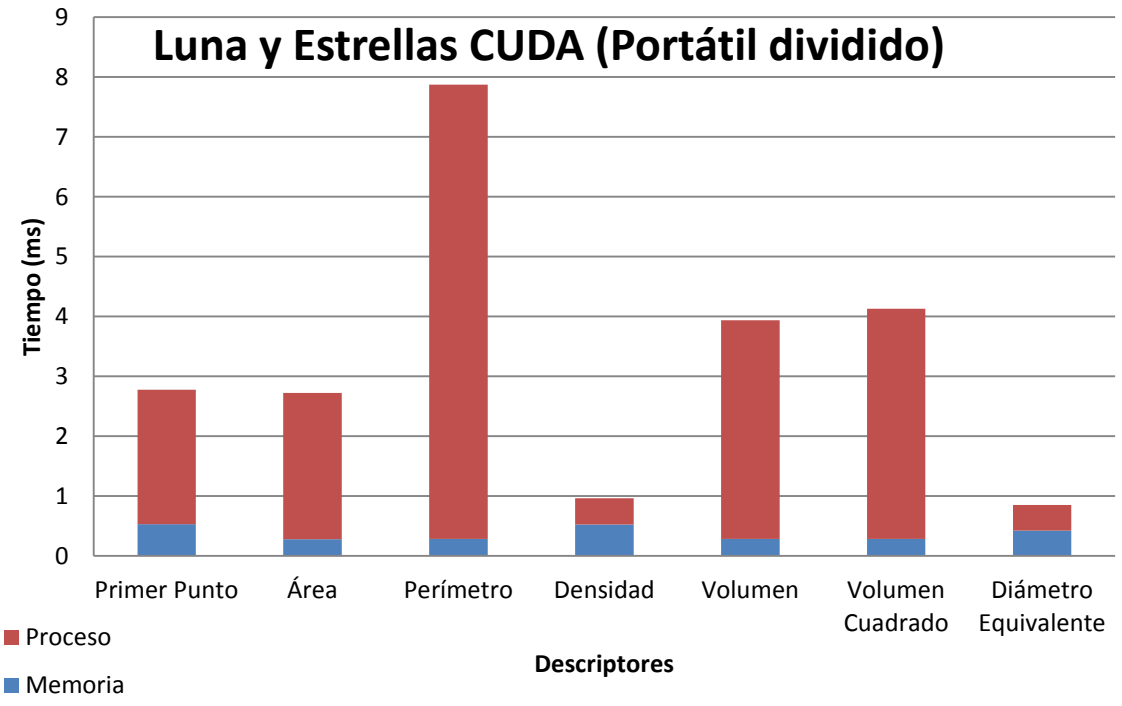
De forma visual podemos ver como en el caso portátil no se mantiene la misma tendencia respecto a sobremesa. En este caso, la fuente Geo se ve realmente penalizada respecto a sus tiempos en el dispositivo gráfico de sobremesa. Al tratarse de la imagen con mayor tamaño, las características que hacen uso de la misma para realizar un barrido son las que se ven penalizadas. Esto puede ser debido directamente al número de procesadores disponibles para ejecución, que se ve reducido de 216 en el sobremesa a 96 para el caso móvil, si recordamos las tablas 12.3 y 12.6.

Como en el caso sobremesa, vamos a presentar ahora un pequeño resumen de los tiempos de ejecución divididos en tiempos de gestión de memoria y tiempos de procesamiento, a fin de observar si se mantiene el patrón anterior.

CUDA Portátil (ms)							
Imagen	***Primer Punto***	***Área***	***Perímetro***	***Densidad***	***Volumen***	***Volumen Cuadrado***	***Diámetro Equivalente***
L. y E. Memoria	0,5260	0,2768	0,2841	0,5217	0,2846	0,2847	0,4204
L. y E. Proceso	2,2500	2,4451	7,5862	0,4391	3,6519	3,8415	0,4307
Geo Memoria	0,7636	0,4103	0,4025	0,6087	0,3940	0,3903	0,5197
Geo Proceso	3,2000	3,6378	10,2590	0,5056	5,8028	5,8606	0,5020
Lena Memoria	0,8543	0,4596	0,4549	0,6809	0,4538	0,4429	0,5521
Lena Proceso	6,4499	7,0526	21,5836	0,6050	11,4049	11,4922	0,5678
Ajedrez Memoria	0,2894	0,1572	0,1542	0,3683	0,1575	0,1575	0,2839
Ajedrez Proceso	8,8991	10,5474	38,5341	0,1814	17,3226	17,3554	0,1940

Tabla 12.22 – Ejecución CUDA en Portátil a Trozos

En el caso portátil, podemos ver como los tiempos de carga en memoria son claramente mayores, debido a la menor velocidad de transmisión entre el dispositivo gráfico y el host, y como los tiempos de procesamiento son, como hemos visto en la tabla 12.21, también mayores.



Gráfica 12.12 – Descriptores Simples sobre Luna y Estrellas en CUDA en Portátil

De forma gráfica, podemos ver como se mantiene el patrón visto en la implementación dividida, donde el tiempo de procesamiento es mucho mayor que el de gestión de memoria, salvo en los dos casos densidad y diámetro equivalente ya estudiados.

12.3.1.3. Comparativa C# y CUDA

Una vez comentados todos los resultados tanto de C# como de CUDA para la implementación de los descriptores simples, vamos a presentar una comparativa para los cuatro casos de imágenes fuente en forma de tablas, y posteriormente una gráfica de una de ellas que ayude a visualizar las diferencias en cuanto a tiempos de procesamiento se refiere. Tanto las implementaciones C# como CUDA se refieren a los experimentos realizados sobre el equipo sobremesa.

Luna y Estrellas (ms)							
Tecnología	*Primer Punto*	*Área*	*Perímetro*	*Densidad*	*Volumen*	*Volumen Cuadrado*	*Diámetro Equivalente*
CUDA	1,9395	1,7875	7,4366	0,4586	2,0385	2,2622	0,4095
C#	16,7409	11,2806	11,7006	2,0001	13,2507	15,2408	0,0500

Tabla 12.23 – Ejecución Descriptores Simples sobre Luna y Estrellas en CUDA y C#

Geo (ms)							
Tecnología	*Primer Punto*	*Área*	*Perímetro*	*Densidad*	*Volumen*	*Volumen Cuadrado*	*Diámetro Equivalente*
CUDA	2,7651	2,4736	9,5024	0,6326	2,8920	3,2233	0,5229
C#	20,3711	16,9309	63,0836	1,0001	20,4511	24,0713	0,01

Tabla 12.24 – Ejecución Descriptores Simples sobre Geo en CUDA y C#

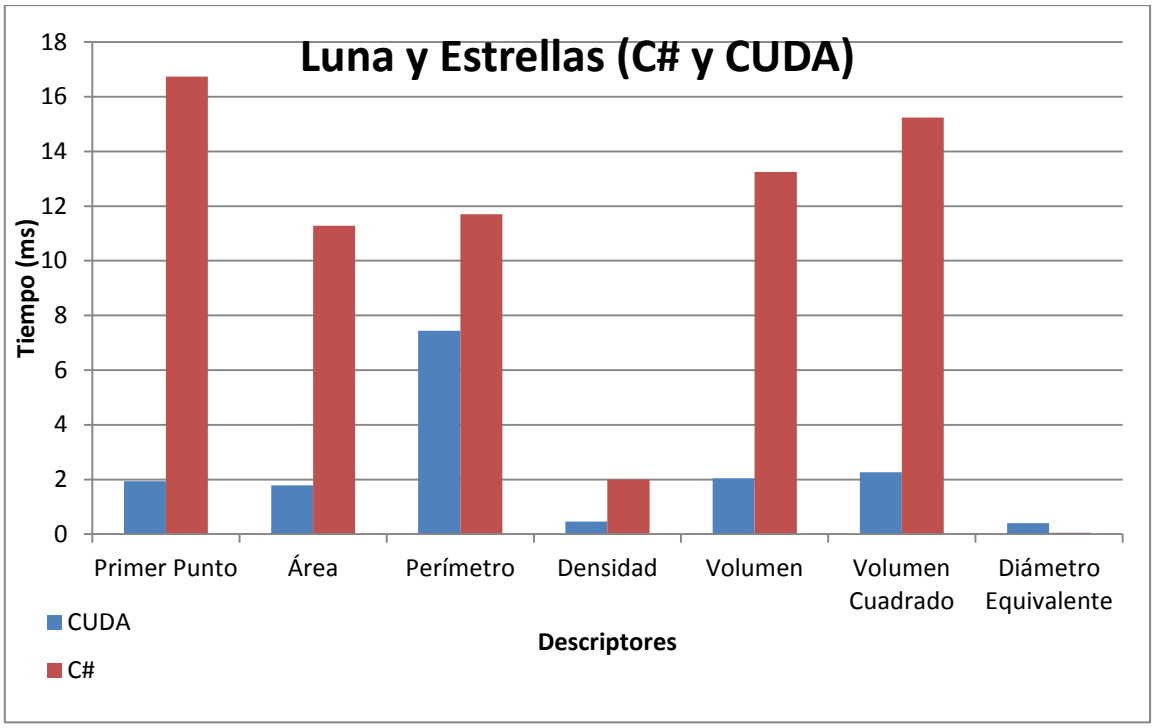
Lena (ms)							
Tecnología	*Primer Punto*	*Área*	*Perímetro*	*Densidad*	*Volumen*	*Volumen Cuadrado*	*Diámetro Equivalente*
CUDA	3,5568	3,4044	14,4244	0,6297	4,0629	4,4637	0,5331
C#	29,6216	20,7811	20,1111	2,0001	22,7913	24,1613	0,02

Tabla 12.25 – Ejecución Descriptores Simples sobre Lena en CUDA y C#

Ajedrez (ms)							
Tecnología	*Primer Punto*	*Área*	*Perímetro*	*Densidad*	*Volumen*	*Volumen Cuadrado*	*Diámetro Equivalente*
CUDA	3,0617	4,2377	20,6684	0,2357	5,0564	5,6651	0,1789
C#	45,1325	35,232	33,8919	2,0001	37,5621	39,2922	0,03

Tabla 12.26 – Ejecución Descriptores Simples sobre Ajedrez en CUDA y C#

Como puede verse, la diferencia de tiempos es abismal entre las dos implementaciones, generando un gran rendimiento por parte de CUDA en casi todos los descriptores. El único descriptor donde la implementación CUDA cede terreno a C# es en el diámetro equivalente, donde la tecnología gráfica se ve muy penalizada por los tiempos de gestión de memoria.



Gráfica 12.13 – Descriptores Simples sobre Luna y Estrellas en CUDA y C# en Sobremesa

Visualmente es más sencillo ver como la implementación gráfica consigue mucho mayor rendimiento en la ejecución de casi todas las características como se ha dicho, y como pierde contra el procesador convencional en un descriptor de baja carga computacional.

Por tanto, para el caso de los descriptores simples queda bastante claro que la mejor opción es la implementación sobre procesador gráfico, salvo en el caso de la densidad y el diámetro equivalente, donde lo mejor es usar la implementación secuencial, salvo que nuestra imagen segmentada contenga del rango de miles de segmentos, para empezar a dar cuanta de un nivel de procesamiento interesante para el dispositivo gráfico.

12.3.2. Descriptores basados en niveles de gris

Ahora describiremos como en el apartado anterior los resultados obtenidos de los experimentos para los descriptores basados en niveles de gris. Debido a que en el apartado 12.3.1 se desarrolló con todo lujo de detalles la problemática de cada caso interesante tanto sobre CUDA como sobre C#, y se estudió y analizó todo rasgo de interés, los siguientes resultados se expondrán tomando como base esos resultados y explicaciones, evitando recrearnos en detalles ya expuestos con claridad.

Los descriptores analizados serán el nivel de gris medio, los niveles de gris máximo y mínimo, la desviación típica y el baricentro de los niveles de gris.

12.3.2.1. Implementación C#

A continuación se expondrán los resultados obtenidos por los experimentos realizados para los descriptores basados en niveles de gris sobre C#, tanto en el entorno de sobremesa como portátil, para su análisis.

Ejecución Sobremesa

Presentamos una tabla por descriptor donde se detallan los tiempos obtenidos en milisegundos para los distintos algoritmos y sus distintas implementaciones tomando como objetivo las cuatro imágenes fuente.

Nivel de gris medio (ms)		
Imagen	***Secuencial***	***Paralelo por región***
Luna y Estrellas	0,0100	0,1000
Geo	0,0001	0,0500
Lena	0,0100	0,2800
Ajedrez	0,0100	0,1300

Tabla 12.27 – Ejecución nivel de gris medio C# en Sobremesa

Este descriptor presenta las mismas características que el descriptor densidad o el descriptor diámetro equivalente, por lo que cualquier ejecución secuencial supera en rendimiento a nuestros esfuerzos paralelos de forma abrumadora. Por tanto en este caso el mejor rendimiento proviene de la implementación secuencial del descriptor.

Nivel de gris máximo y mínimo (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	89,0150	143,9482	165,4094	24,9114
Geo	147,0684	204,2316	298,5670	39,8622
Lena	158,2490	247,6741	327,7987	45,5626
Ajedrez	278,7559	423,6542	564,6522	75,7243

Tabla 12.28 – Ejecución nivel de gris máximo y mínimo C# en Sobremesa

Para este caso del nivel de gris máximo y mínimo, volvemos al patrón de tres implementaciones paralelas y una secuencial donde la implementación supera a las implementaciones paralelas por pixel y por región, y donde el mayor rendimiento lo ofrece la alternativa híbrida paralela por fila y región. De hecho ofrece un speedup de hasta 3,7 frente a la implementación secuencial, mientras ocurre la perdida de rendimiento de las otras alternativas paralelas.

Desviación típica (ms)				
Imagen	*Secuencial*	*Paralelo por pixel*	*Paralelo por región*	*Paralelo por fila y región*
Luna y Estrellas	92,9953	160,2691	1417,0810	23,5113
Geo	189,9808	203,3716	378,6216	47,7527
Lena	165,7694	273,8556	3807,0277	42,9324
Ajedrez	292,5967	446,1455	5730,3277	73,5242

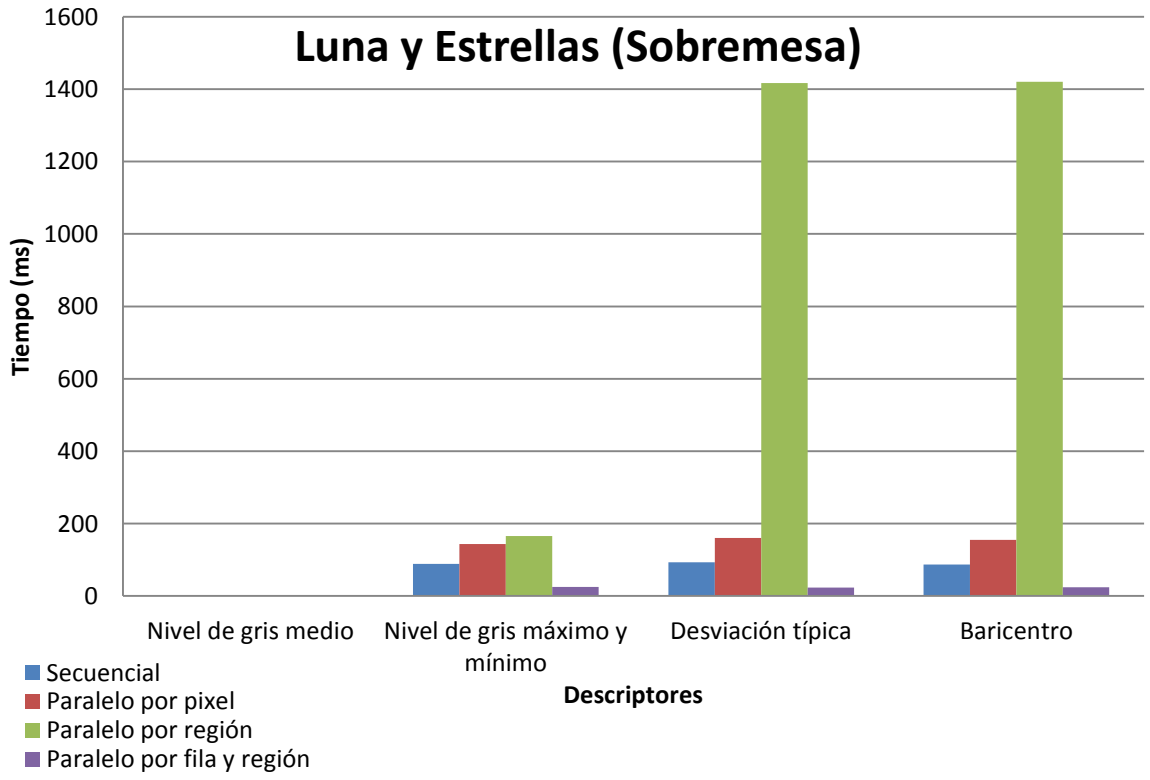
Tabla 12.29 – Ejecución desviación típica C# en Sobremesa

Como en el caso anterior, la mejor opción en cuanto al tiempo medio es la implementación paralela por fila y región seguida de la implementación secuencial, consiguiendo un speedup de hasta 3,9.

Baricentro (ms)				
Imagen	*Secuencial*	*Paralelo por pixel*	*Paralelo por región*	*Paralelo por fila y región*
Luna y Estrellas	87,2249	155,4088	1420,7312	23,98137
Geo	146,0583	207,6718	319,0582	39,3822
Lena	155,5888	263,5250	3803,0175	43,2424
Ajedrez	275,6857	422,9541	5831,2735	74,4542

Tabla 12.30 – Ejecución baricentro C# en Sobremesa

Volvemos a obtener el mismo patrón para el descriptor baricentro, donde el speedup conseguido por la implementación paralela por fila y región es en este caso de 3,7. Las demás alternativas paralelas siguen estando, como era de prever, por debajo de la implementación secuencial.



Gráfica 12.14 – Descriptores basados en niveles de gris sobre Luna y Estrellas C# en Sobremesa

Gráficamente más sencillo observar como la implementación que consigue el mejor rendimiento es la más que probada y avalada opción paralela por fila y región. Por tanto no cambia nada respecto a lo analizado sobre los descriptores simples, y esto corrobora lo ya comentado.

Ejecución Portátil

Para el caso portátil seguiremos el mismo esquema anterior, dando breves comentarios y análisis sobre las distintas tablas de resultados y la gráfica final que dará una perspectiva general de los datos obtenidos.

Nivel de gris medio (ms)		
Imagen	***Secuencial***	***Paralelo por región***
Luna y Estrellas	0,0100	0,1000
Geo	0,0001	1,5600
Lena	0,0100	0.7001
Ajedrez	0,0100	0.5401

Tabla 12.31 – Ejecución nivel de gris medio C# en Portátil

Como en el caso sobremesa, el nivel de gris medio es un descriptor de poca carga de trabajo e ínfimo conjunto de datos de partida, lo que le confiere un plus en la implementación secuencial.

Nivel de gris máximo y mínimo (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	81,2201	138,8402	254,2804	46,8000
Geo	134,1602	258,9604	338,5205	76,4401
Lena	146,6402	248,0404	528,8409	84,2401
Ajedrez	251,1604	438,3607	882,9615	138,8402

Tabla 12.32 – Ejecución nivel de gris máximo y mínimo C# en Portátil

Para el caso del nivel de gris máximo y mínimo, la tónica se mantiene para las implementaciones paralelas por pixel y región, donde su rendimiento es mucho menor que el ofrecido por la implementación secuencial. El listón del procesamiento paralelo lo coloca de nuevo la versión paralela por fila o región, o híbrida, consiguiendo un speedup de 1,9 de máximo.

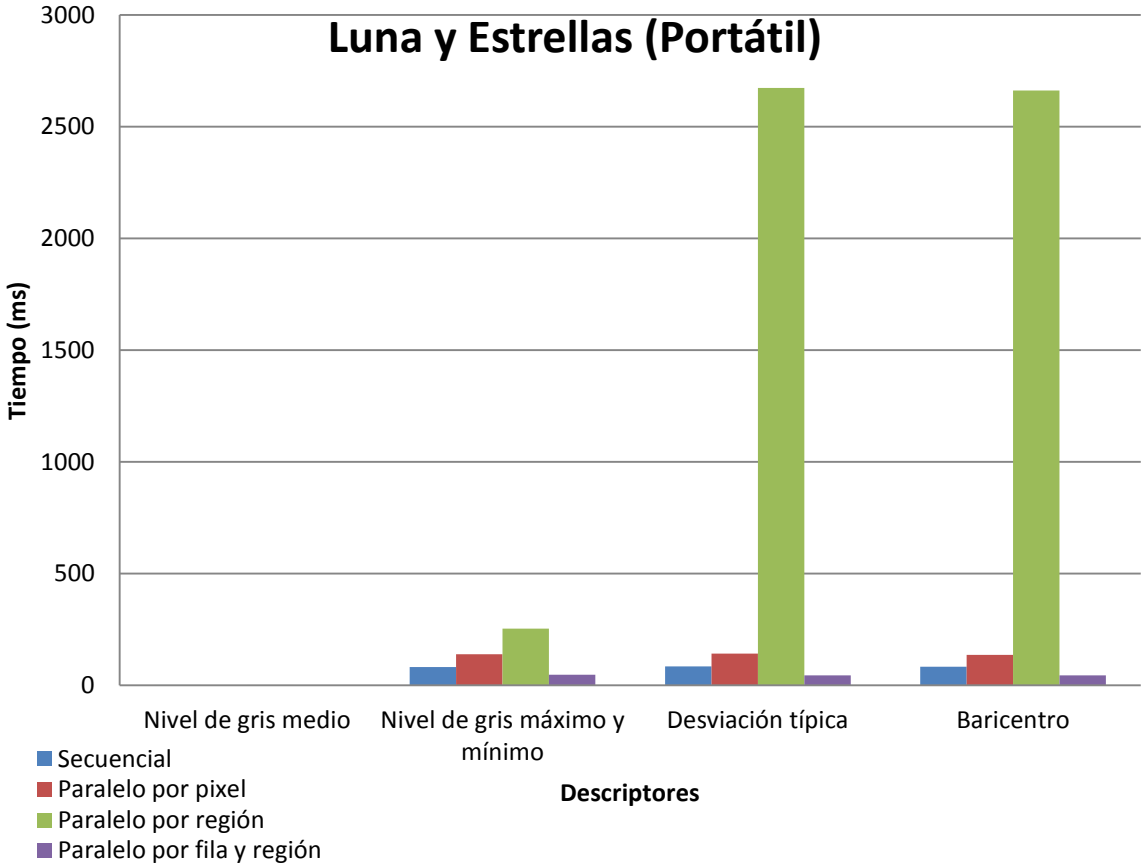
Desviación típica (ms)				
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por región***	***Paralelo por fila y región***
Luna y Estrellas	84,2401	141,9602	2672,284	45,2400
Geo	171,6003	382,2006	404,0407	90,4801
Lena	152,8802	258,9604	7280,5327	79,5601
Ajedrez	271,4404	463,3208	10871,6591	137,2802

Tabla 12.33 – Ejecución desviación típica C# en Portátil

Baricentro (ms)				
Imagen	*Secuencial*	*Paralelo por pixel*	*Paralelo por región*	*Paralelo por fila y región*
Luna y Estrellas	82,6801	135,7202	2661,3646	45,2400
Geo	135,7202	349,4406	391,5606	78,0001
Lena	145,0802	251,1604	7268,0527	79,5601
Ajedrez	252,7204	443,0407	10899,7391	137,2802

Tabla 12.34 – Ejecución baricentro C# en Portátil

Los casos desviación típica y baricentro proporcionan el mismo marco de referencia que los anteriores, dominado completamente por la implementación paralela por fila y región, donde el speedup conseguido es de 1,9 y 1,9 de máximo respectivamente.



Gráfica 12.15 – Descriptores basados en niveles de gris sobre Luna y Estrellas C# en Portátil

Como se puede observar en la gráfica, claramente se ve como, al igual que en los casos anteriores, la mejor opción es el procesamiento paralelo por fila y región para todos los casos menos para el descriptor del nivel de gris medio. Este, por sus características de procesamiento concretas, resulta imbatible para los casos de prueba establecidos mediante un algoritmo paralelo, como se ha podido comprobar en este y en los casos anteriores de densidad y diámetro equivalente.

12.3.2.2. Implementación CUDA

A continuación se expondrán los resultados obtenidos por los experimentos realizados para los descriptores basados en niveles de gris sobre CUDA, tanto en el entorno de sobremesa como portátil, para su análisis.

Ejecución Sobremesa

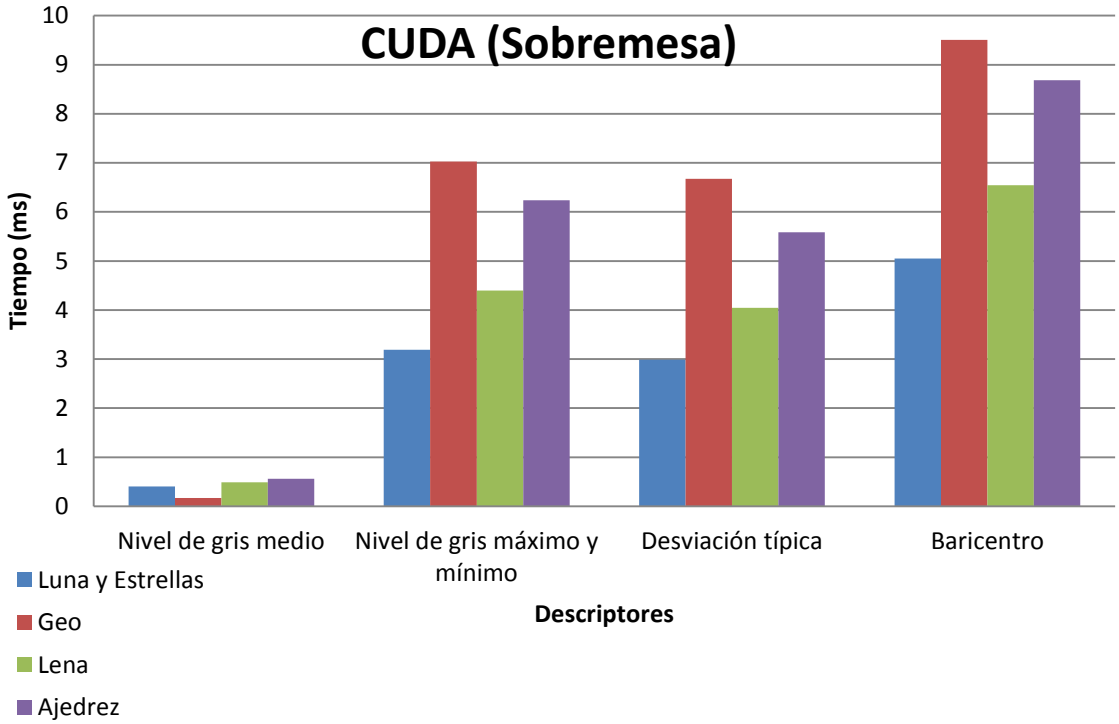
Presentamos una tabla por descriptor donde se detallan los tiempos obtenidos en milisegundos para los distintos algoritmos y sus distintas implementaciones tomando como objetivo las cuatro imágenes fuente.

CUDA Sobremesa (ms)				
Imagen	***Nivel de gris medio***	***Nivel de gris máximo y mínimo***	***Desviación típica***	***Baricentro***
Luna y Estrellas	0,4023	3,1907	2,9902	5,0507
Geo	0,1677	7,0232	6,6750	9,5046
Lena	0,4907	4,3978	4,0449	6,5434
Ajedrez	0,5573	6,2386	5,5870	8,6853

Tabla 12.35 – Ejecución CUDA en Sobremesa

Como podemos ver los tiempos de ejecución vuelven a ser extremadamente bajos en comparación con las implementaciones presentadas sobre la plataforma C#. Si volvemos a la tabla 12.28 podemos ver como bajamos para la imagen Lena de 45 ms para el cálculo del nivel de gris máximo y mínimo, el mejor caso, hasta 6 ms. Esto puede mostrar el alcance real del rendimiento que pude ofrecer CUDA. De igual manera, el mayor tiempo obtenido en estos resultados por el algoritmo C# paralelo por fila y región es de 75 ms para el cálculo del nivel de gris máximo y mínimo, misma tabla, para la imagen ajedrez, que en CUDA ofrece un tiempo de procesamiento de otros 6 ms.

Podemos ver mejor estos datos mediante un ejemplo visual en forma de gráfica.



Gráfica 12.16 – Descriptores basados en niveles de gris sobre CUDA en Sobremesa

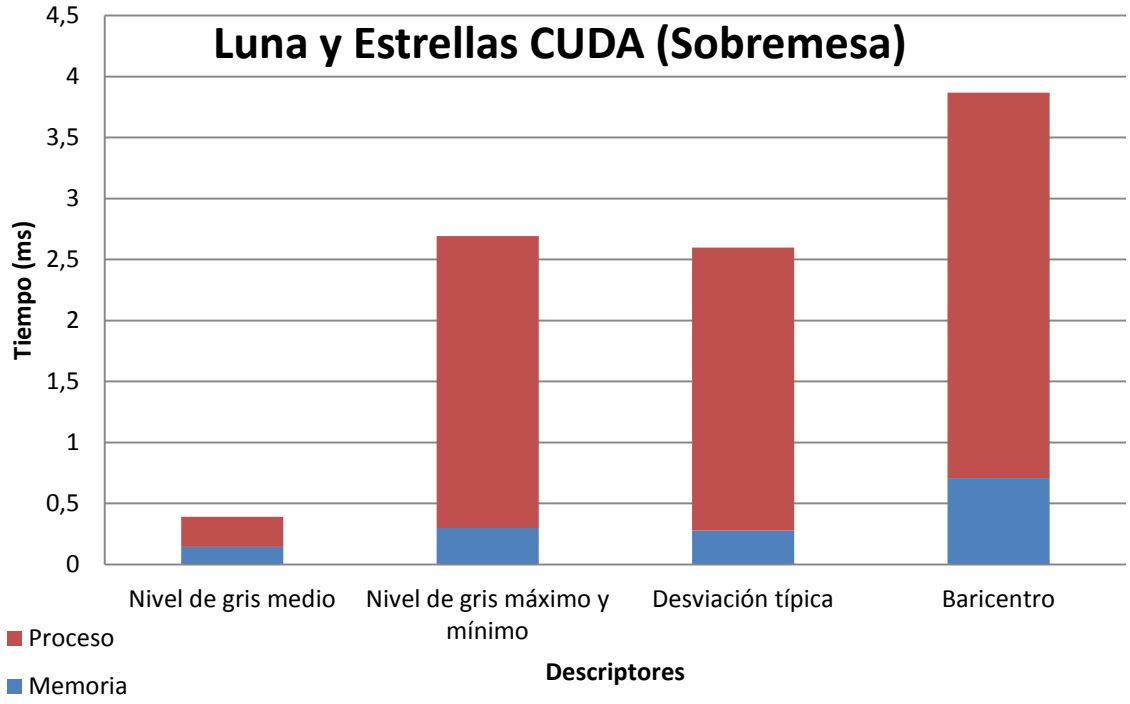
Aquí se puede comprobar como la fuente que ofrece de forma general mejores resultados es la fuente Geo, de la que obtenemos mejores resultados. Esta presenta el mayor tamaño pero también el menor número de segmentos.

A fin de conseguir un poco más de detalle para estas ejecuciones vamos a presentar, como en el caso anterior, una tabla de tiempos de procesamiento teniendo en cuenta por un lado las gestiones de memoria y por otro el procesamiento puro.

CUDA Sobremesa (ms)				
Imagen	*Nivel de gris medio*	*Nivel de gris máximo y mínimo*	*Desviación típica*	*Baricentro*
L. y E. Memoria	0,1445	0,2997	0,2769	0,7075
L. y E. Proceso	0,2460	2,3933	2,3221	3,1600
Geo Memoria	0,2324	0,5033	0,3941	1,1154
Geo Proceso	0,3060	4,8123	4,5430	6,1929
Lena Memoria	0,2013	0,4320	0,3468	0,9896
Lena Proceso	0,2818	3,1314	3,1010	4,1958
Ajedrez Memoria	0,0601	0,1228	0,2087	0,3893
Ajedrez Proceso	0,1048	6,8338	6,4353	9,2161

Tabla 12.36 – Ejecución CUDA en Sobremesa a Trozos

En esta ejecución dividida en procesamiento y memoria podemos ver el mismo patrón que en los descriptores simples, donde el tiempo de memoria es muy inferior al tiempo de procesamiento, salvo en el caso del nivel de gris medio, que se encuentra a la altura como hemos dicho de otros descriptores simples donde el tiempo de memoria se igualaba e incluso superaba al tiempo de procesamiento, como ocurre por ejemplo en la imagen ajedrez.



Gráfica 12.17 – Descriptores basados en niveles de gris sobre Luna y Estrellas en CUDA en Sobremesa

De forma gráfica podemos comprobar lo dicho anteriormente. Por un lado podemos ver como el tiempo de procesamiento sigue siendo mucho mayor que el tiempo de gestión de memoria, como viene pasando en los distintos ejemplos mostrados, y por otro, el nivel de gris medio sigue mostrando su línea de tiempo de procesamiento igual o similar al tiempo de gestión de memoria.

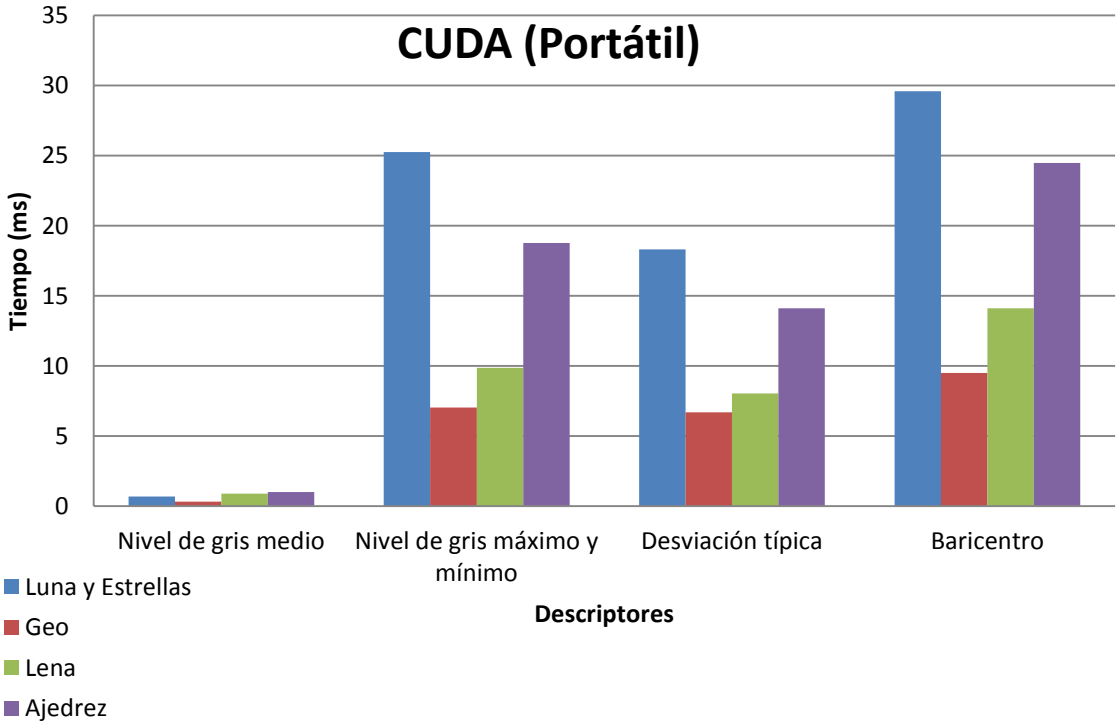
Ejecución Portátil

Presentamos una tabla por descriptor donde se detallan los tiempos obtenidos en milisegundos para los distintos algoritmos y sus distintas implementaciones tomando como objetivo las cuatro imágenes fuente.

CUDA Portátil (ms)				
Imagen	***Nivel de gris medio***	***Nivel de gris máximo y mínimo***	***Desviación típica***	***Baricentro***
Luna y Estrellas	0,6818	25,2560	18,3077	29,5917
Geo	0,3038	7,0232	6,6750	9,5046
Lena	0,8738	9,8618	8,0274	14,1052
Ajedrez	0,9914	18,7768	14,1015	24,4736

Tabla 12.37 – Ejecución CUDA en Portátil

Echando un vistazo rápido a la tabla 12.37 podemos ver como los tiempos a priori son bastante más elevados que para el caso sobremesa, como esperábamos que ocurriese. Aun así, esta ejecución sigue ofreciendo mayor rendimiento y mejores resultados que las implementaciones realizadas sobre C#



Gráfica 12.18 – Descriptores basados en niveles de gris sobre CUDA en Portátil

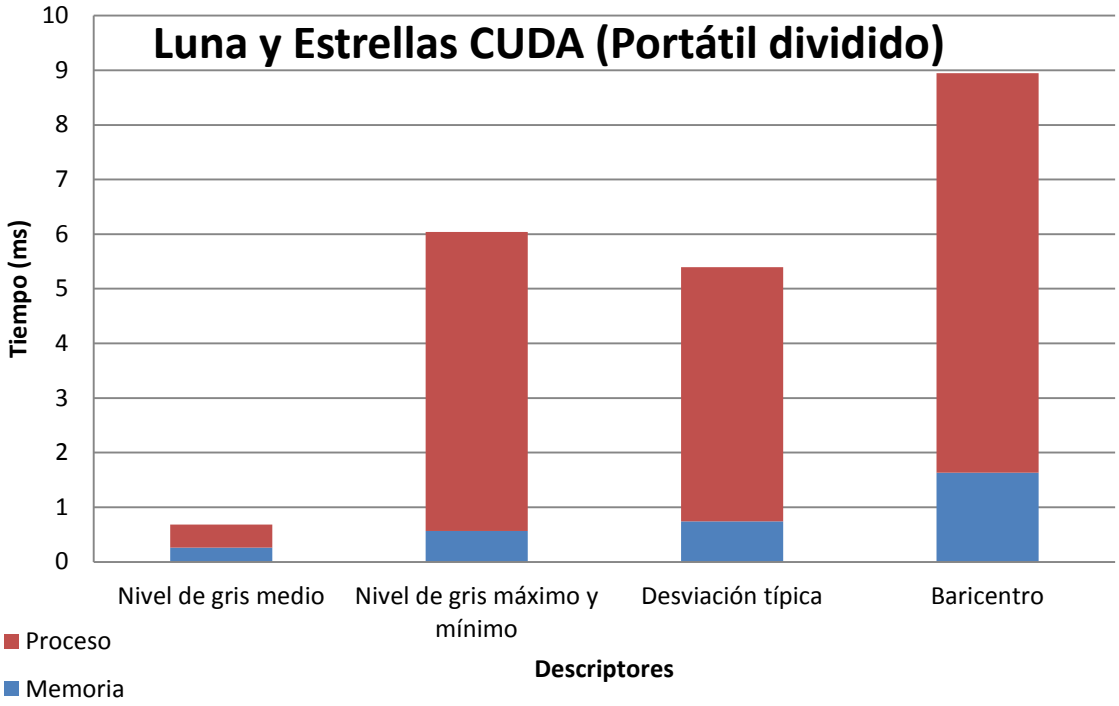
De forma gráfica podemos observar que el rendimiento de la implementación CUDA aun sobre un dispositivo portátil es superior a la ofrecida por C# en una arquitectura sobremesa.

A fin de obtener algo más de detalle sobre estos resultados vamos a presentar los datos de ejecución divididos en tiempo de gestión de memoria y tiempo de proceso.

CUDA Portátil (ms)				
Imagen	***Nivel de gris medio***	***Nivel de gris máximo y mínimo***	***Desviación típica***	***Baricentro***
L. y E. Memoria	0,2587	0,5690	0,7396	1,6337
L. y E. Proceso	0,4273	5,4703	4,6545	7,3157
Geo Memoria	0,1454	0,3073	0,5887	1,0707
Geo Proceso	0,1526	24,7926	17,7588	28,4807
Lena Memoria	0,3516	0,7799	0,8316	2,0447
Lena Proceso	0,5015	8,2691	6,8032	10,9322
Ajedrez Memoria	0,3952	0,8920	0,8989	2,3043
Ajedrez Proceso	0,5735	16,6356	12,4997	20,6852

Tabla 12.38 – Ejecución CUDA en Portátil a Trozos

En la tabla 12.38 podemos ver claramente los tiempos dedicados a memoria y proceso por cada descriptor y cada fuente, quedando patente que el mayor tiempo es empleado en procesamiento. De igual manera, se ve como el nivel de gris medio mantiene su igualdad entre los tiempos de memoria y procesamiento.



Gráfica 12.19 – Descriptores basados en niveles de gris sobre Luna y Estrellas en CUDA en Portátil

Visualmente, gracias a la gráfica 12.19, podemos observar lo comentado anteriormente, donde el tiempo dedicado a memoria es una pequeña parte del tiempo total dedicado a la ejecución de cada descriptor.

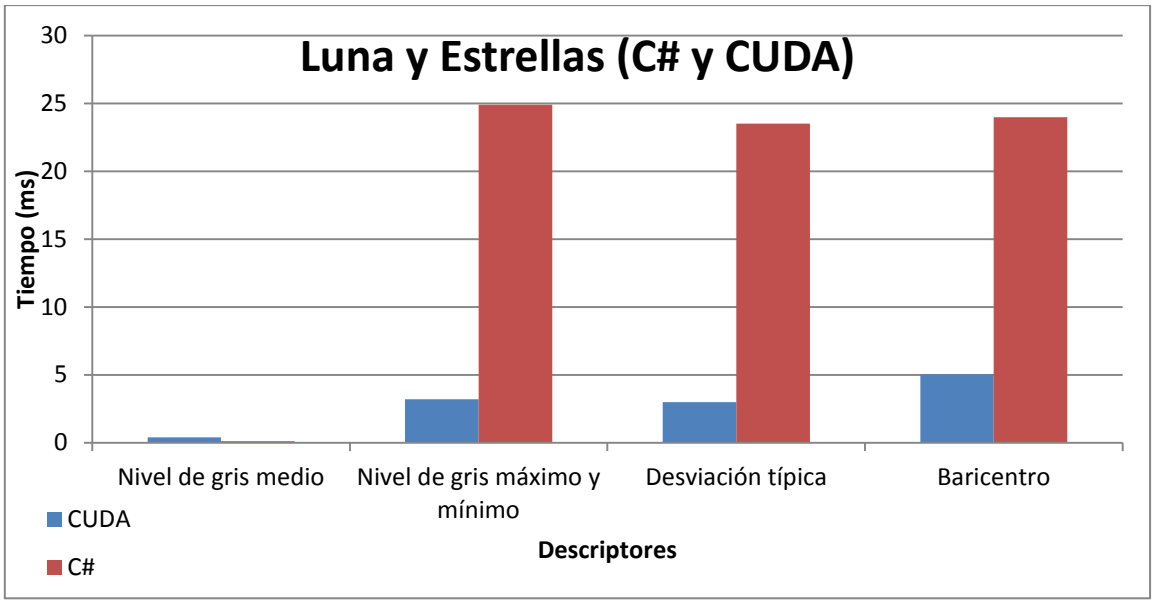
12.3.2.3. Comparativa C# y CUDA

Con motivo de ver de forma más clara la diferencia de rendimiento entre C# y CUDA para los descriptores basados en niveles de gris se presentan la siguientes tabla comparativa sobre una de las fuentes de imagen usadas.

Luna y Estrellas (ms)				
Tecnología	***Nivel de gris medio***	***Nivel de gris máximo y mínimo***	***Desviación típica***	***Baricentro***
CUDA	0,4023	3,1907	2,9902	5,0507
C#	0,1000	24,9114	23,5113	23,98137

Tabla 12.39 – Ejecución Descriptores basados en niveles de gris sobre Luna y Estrellas en CUDA y C#

Como en los casos anteriores, en cualquier descriptor que requiera un barrido de la imagen la implementación CUDA se hace con el mando en lo que a rendimiento se refiere, mientras que los descriptores con baja carga de trabajo y conjunto de datos de origen pequeño son susceptibles de tardar demasiado tiempo para competir con la implementación paralela C#



Gráfica 12.20 – Descriptores basados en niveles de gris sobre Luna y Estrellas en CUDA y C# en Sobremesa

De forma visual podemos observar sobre la gráfica 12.20 como el rendimiento esta claramente de parte de la implementación CUDA, salvo en el primer descriptor comentado, donde la implementación sobre procesador convencional genera un mejor rendimiento.

12.3.3. Descriptores basados en el rectángulo circunscrito y Puntos extremos

Como se comentó anteriormente, en el capitulo 11, la implementación de los descriptores basados en el rectángulo circunscrito y los puntos extremos se realizó de forma conjunta, de tal manera que un solo algoritmo extrae a la vez todas estas características. Recordemos que esto es así debido a que codificando el rectángulo circunscrito se observó que parte de los procesos necesarios para calcularlo eran parte de los resultados finales de los puntos extremos, por lo que dividir el procedimiento para calcular ambos conjuntos de descriptores de forma separada exigía realizar el mismo procesamiento dos veces, por lo que incurría en una perdida de eficiencia.

12.3.3.1. Implementación C#

A continuación se expondrán los resultados obtenidos por los experimentos realizados para los descriptores basados en niveles de gris sobre C#, tanto en el entorno de sobremesa como portátil, para su análisis.

Ejecución Sobremesa

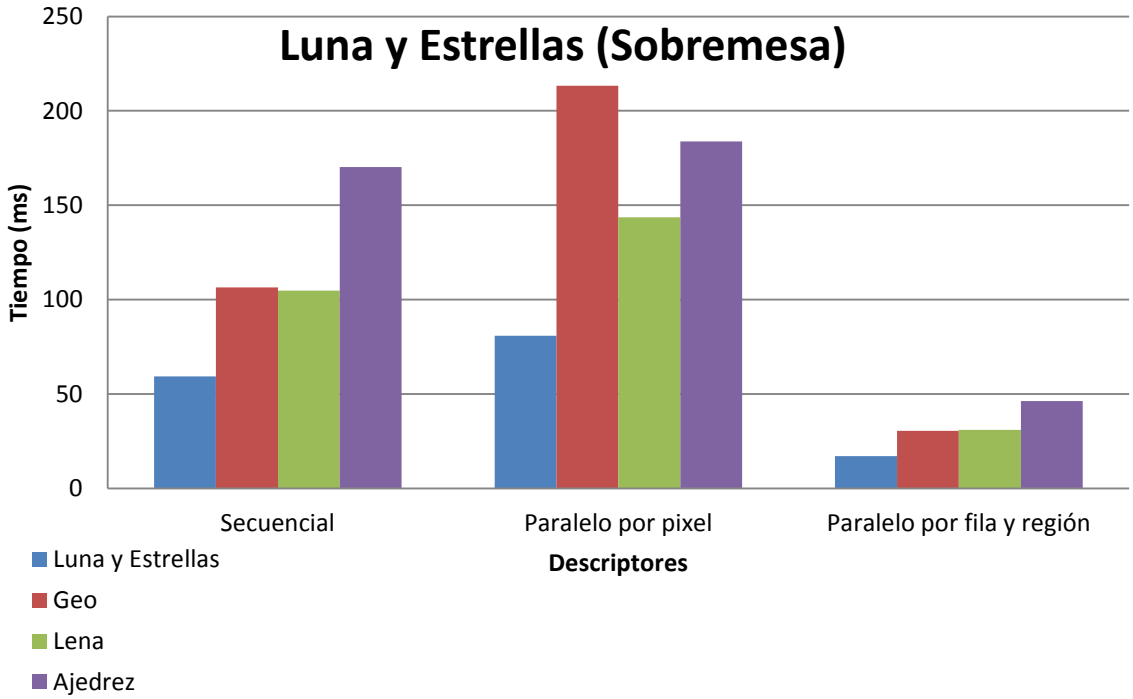
Presentamos una tabla donde se detallan los tiempos obtenidos en milisegundos para los distintos algoritmos y sus distintas implementaciones tomando como objetivo las cuatro imágenes fuente.

Rectángulo circunscrito y Puntos extremos (ms)			
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por fila y región***
Luna y Estrellas	59,3133	80,9546	17,1309
Geo	106,4860	213,3622	30,5617
Lena	104,6959	143,5482	31,0417
Ajedrez	170,1597	183,8505	46,2226

Tabla 12.40 – Ejecución rectángulo circunscrito y puntos extremos C# en Sobremesa

Para esta implementación se ha optado por no realizar la paralelización por regiones debido a sus pésimos resultados durante todos los descriptores implementados, dejando solo una implementación secuencial y las implementaciones paralelas paralela por pixel y paralela por fila y región.

Siguiendo la tónica de todas las pruebas presentadas, el diseño paralelo por fila y región es el que obtiene la mejoría sobre la implementación secuencial, por lo que es el elegido para posteriores comparativas con la estrategia CUDA.



Gráfica 12.21 – Descriptores basados en rectángulo circunscrito y puntos extremos en C# en Sobremesa

De forma visual podemos apreciar en la gráfica los mismos datos comentados anteriormente, donde la implementación que ofrece mejor rendimiento es la paralela por fila y región, frente a la perdida de rendimiento respecto a la implementación secuencial de la alternativa paralela por pixel.

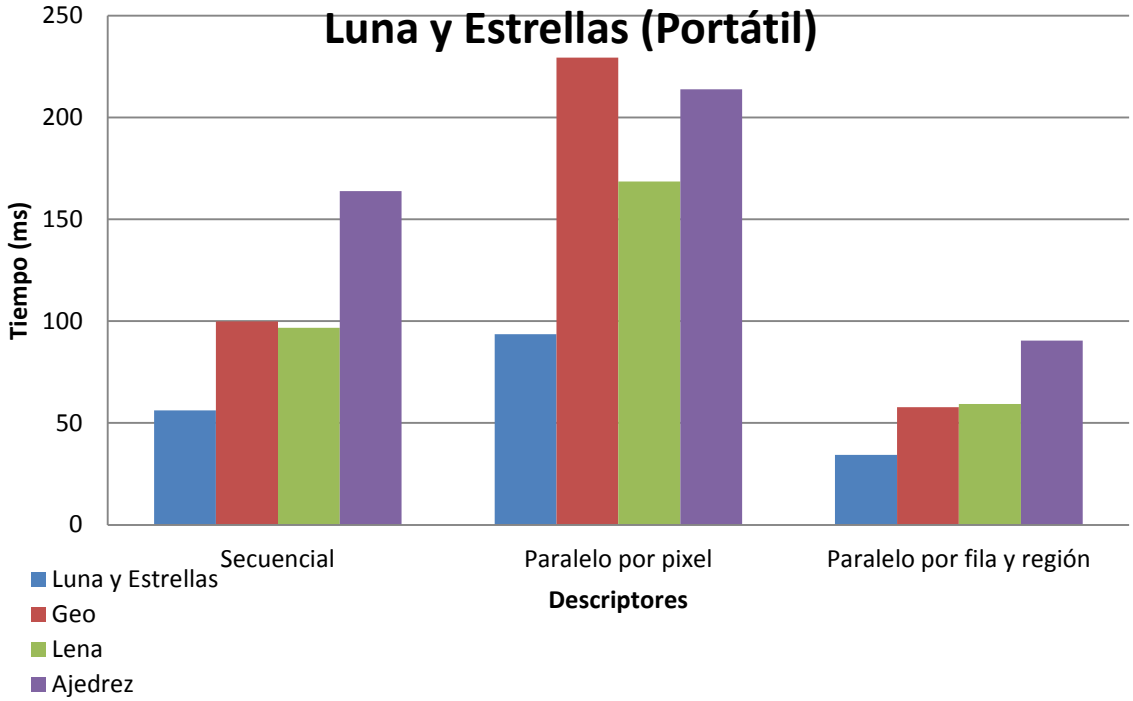
Ejecución Portátil

Presentamos una tabla donde se detallan los tiempos obtenidos en milisegundos para los distintos algoritmos y sus distintas implementaciones tomando como objetivo las cuatro imágenes fuente.

Rectángulo circunscrito y Puntos extremos (ms)			
Imagen	***Secuencial***	***Paralelo por pixel***	***Paralelo por fila y región***
Luna y Estrellas	56,1601	93,6001	34,3200
Geo	99,8401	229,3204	57,7201
Lena	96,7201	168,4802	59,2801
Ajedrez	163,8002	213,7203	90,4801

Tabla 12.41 – Ejecución rectángulo circunscrito y puntos extremos C# en Portátil

De forma análoga a la anterior podemos ver los resultados en la tabla 12.41 para la implementación sobre portátil realizada. Obtenemos los mismos datos y las mismas conclusiones que anteriormente pero con tiempos algo superiores.



Gráfica 12.22 – Descriptores basados en rectángulo circunscrito y puntos extremos en C# en Portátil

De forma complementaria a la anterior podemos ver como la mejor implementación es la relativa al diseño paralelo por fila y región, pero con tiempos más altos en comparación con su homónima de sobremesa.

12.3.3.2. Implementación CUDA

A continuación se expondrán los resultados obtenidos por los experimentos realizados para los descriptores basados en niveles de gris sobre CUDA, tanto en el entorno de sobremesa como portátil, para su análisis.

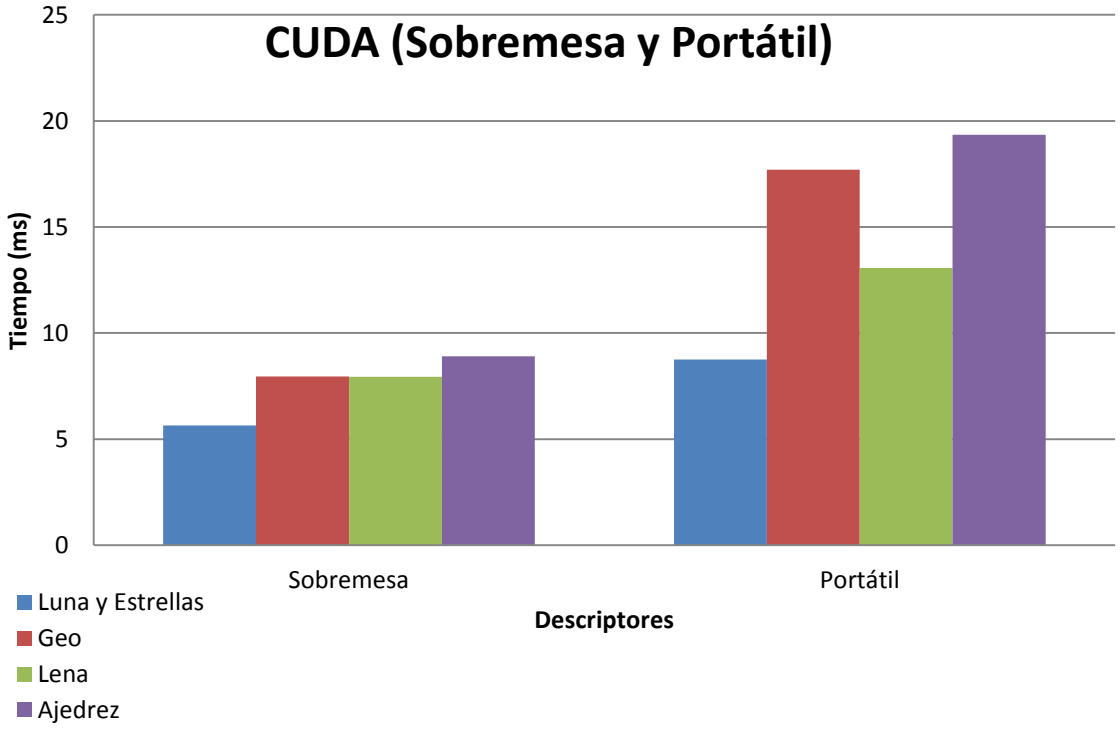
Ejecución Sobremesa y Portátil

Debido a que solo contamos con un descriptor complejo, fruto de aunar todos los que corresponden a las dos categorías de rectángulo circunscrito y puntos extremos, en este caso realizaremos el análisis de las plataformas sobremesa y portátil en conjunto.

Rectángulo circunscrito y Puntos extremos (ms)		
Imagen	***Sobremesa***	***Portátil***
Luna y Estrellas	5,6356	8,7448
Geo	7,9540	17,6979
Lena	7,9431	13,0669
Ajedrez	8,9043	19,3529

Tabla 12.42 – Ejecución rectángulo circunscrito y puntos extremos CUDA

Como podemos ver en la tabla 12.42 los tiempos son muy inferiores a los de C#, y además los tiempos en sobremesa son también inferiores a los de portátil. Con esto podemos deducir que la mejor plataforma es la plataforma sobremesa, como ya sabíamos, pero también que no existen limitaciones sobre su funcionamiento en plataformas de menor capacidad, como la portátil. De igual manera podemos observarlo en la gráfica 12.23.

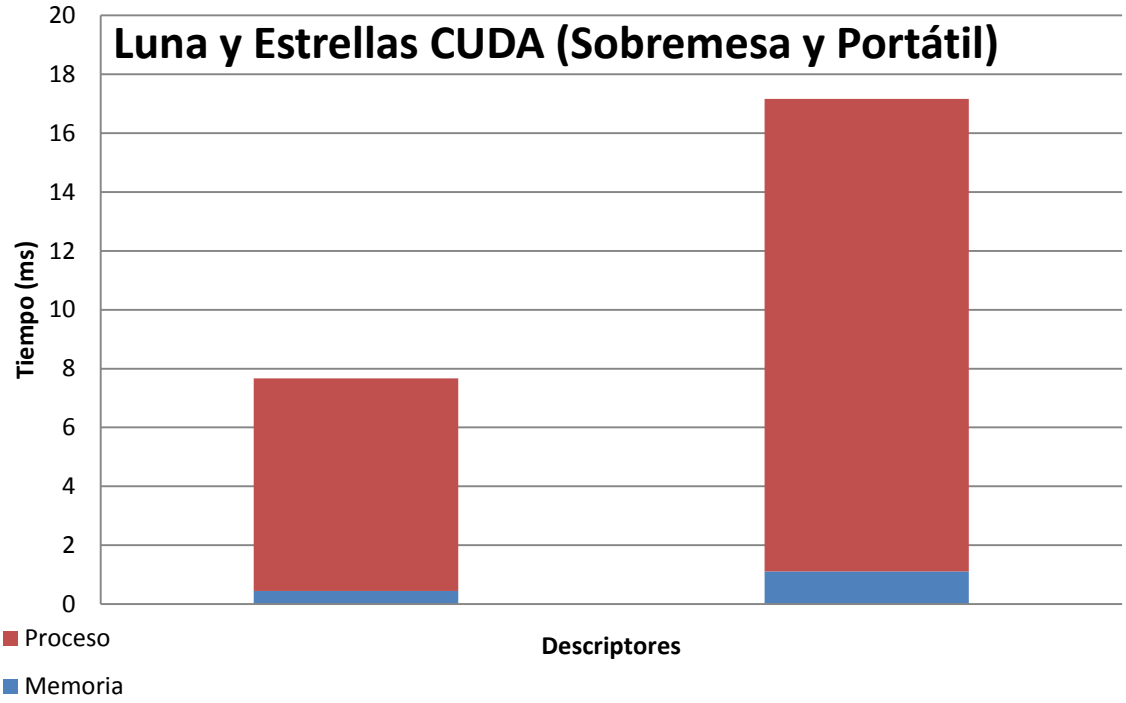


Gráfica 12.23 – Descriptores basados en rectángulo circunscrito y puntos extremos en CUDA en Portátil y Sobremesa

CUDA Sobremesa y Portátil(ms)		
Imagen	***Sobremesa***	***Portátil***
L. y E. Memoria	0,4505	1,1099
L. y E. Proceso	7,2211	16,0497
Geo Memoria	0,7448	1,6796
Geo Proceso	2,5486	5,3481
Lena Memoria	1,0510	2,0745
Lena Proceso	3,5389	8,0076
Ajedrez Memoria	1,2013	2,3059
Ajedrez Proceso	4,9275	14,5042

Tabla 12.43 – Ejecución rectángulo circunscrito y puntos extremos CUDA a Trozos

Según la tabla 12.43 de ejecución dividida para las implementaciones CUDA sobre portátil y sobremesa, vemos como se mantiene la tónica donde el tiempo de memoria es siempre una fracción pequeña del tiempo de ejecución total. Este hecho está presente tanto en sobremesa como portátil, diferenciando los tiempos de carga en memoria y procesamiento, que son más altos en la plataforma móvil. Igualmente podemos verlo de forma gráfica en la gráfica 12.23.



Gráfica 12.23 – Descriptores basados en rectángulo circunscrito y puntos extremos sobre Luna y Estrellas en CUDA en Portátil y Sobremesa

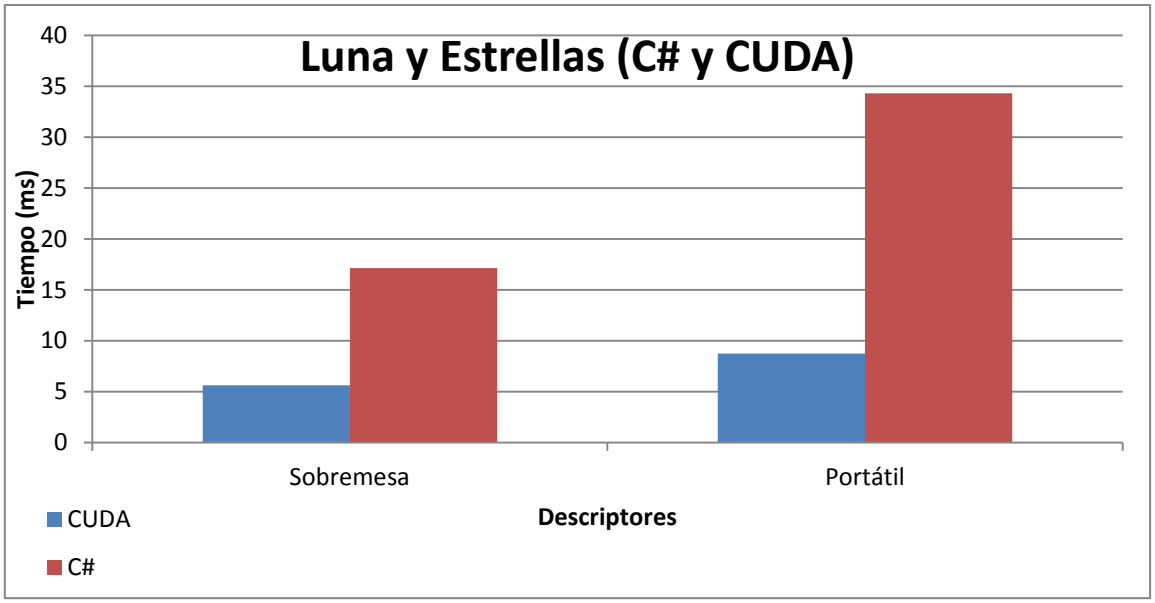
12.3.3.3. Comparativa C# y CUDA

Con motivo de ver de forma más clara la diferencia de rendimiento entre C# y CUDA para los descriptores basados en el rectángulo circunscrito y los puntos extremos se presentan la siguientes tabla comparativa sobre una de las fuentes de imagen usadas.

Luna y Estrellas (ms)		
Tecnología	***Sobremesa***	***Portátil***
CUDA	5,6356	8,7448
C#	17,1309	34,3200

Tabla 12.44 – Ejecución Descriptores basados en rectángulo circunscrito y puntos extremos sobre Luna y Estrellas en CUDA y C#

Como podemos ver, la implementación CUDA barre nuevamente a los esfuerzos paralelos sobre C#. En este caso comparamos también con los resultados sobre portátil, donde podemos ver como sigue la misma tónica, pero con tiempos más elevados. De la misma forma podemos observarlo en la gráfica 12.24,



Gráfica 12.24 – Descriptores basados en rectángulo circunscrito y puntos extremos sobre Luna y Estrellas en CUDA y C# en Sobremesa

Parte V

Conclusiones

Capítulo 13

Conclusiones

El motivo principal de este estudio comparativo de las tecnologías C# paralelo y CUDA era demostrar que esta última presenta un gran potencial como tecnología de procesamiento paralelo novedosa, y que es aplicable al campo del procesamiento de imágenes. C# ha servido como banco de pruebas, además de proporcionar una medida de comparación útil para el objeto del estudio. Hemos contado pues con una tecnología de paralelización de última generación y actual como es C# con su TPL, usada en aplicaciones de propósito general sobre plataformas Windows, por lo que esta altamente orientada al usuario final. Con estos datos y este banco de pruebas realizamos los estudios comparativos, que enfrentan a una tecnología de factura aún más novedosa que la TPL de C#, debido a la arquitectura que la sostiene, e igualmente disponible para su aplicación frente al público general.

Todo esto sirve para definir el marco de trabajo fijado y utilizado durante todo el proceso. En base a él obtenemos unos grandes resultados que muestran las diferencias en tiempo de ejecución medio entre las tecnologías comparadas pero, sobre todo, nos ofrecen una visión del potencial real de la tecnología de procesamiento gráfico CUDA, cuya capacidad de paralelización crece de forma constante con cada nueva remesa de dispositivos que inundan el mercado, lo que hace mejorar cada vez más los resultados aquí expuestos sin necesidad de realizar modificación alguna.

Por tanto, podemos concluir de forma categórica que la tecnología CUDA no solo es aplicable en el pequeño campo de la caracterización de imágenes segmentadas estudiado, si no que es muy recomendable usarla en cualquier implementación que busque la eficiencia y el rendimiento, y donde el tiempo de respuesta sea un factor crítico.

De igual manera, podemos concluir que el potencial de dicha tecnología es realmente elevado, siendo sus aplicaciones enormes, como se puede ver en las referencias [meter referencias tanto de master como de articulo y demás referencias del estado del arte interesantes], y ofertando una capacidad de procesamiento a un coste relativamente bajo en comparación con otras alternativas multiprocesador.

Por tanto, se considera con todo que se han alcanzado los objetivos propuestos a comienzos del proyecto, haciendo realizado un análisis detallado de las tecnologías usadas y probado su correcto desempeño de las tareas implementadas y su idoneidad, o no, en diversos campos de actuación.

Capítulo 14

Trabajos futuros

Los siguientes pasos a llevar a cabo en el presente proyecto se enmarcan en dos caminos diferenciados, pero todos dirigidos al mismo fin. En primer lugar sería interesante hacer un estudio de la última versión liberada del SDK de CUDA, en la cual se introducen nuevos conceptos de programación dedicados al aprovechamiento de la capacidad tanto de procesamiento como de acceso a memoria proporcionada por el hardware, de esta manera nos brinda un mayor control y mejores herramientas para el desempeño de las tareas de procesamiento gráfico. De igual manera, también proporciona nuevas funcionalidad y soporta mejores estándares de procesamiento aritmético para sus últimos dispositivos, lo que da una nueva vuelta de tuerca a la potencia de esta tecnología. Por otro lado, sería una buena estudiar el hardware que hace posible la ejecución CUDA al detalle, desde sus inicios a las ultimas series 600 fabricadas, a fin de desarrollar un marco de computación específico para cada plataforma concreta, sea una seria u otra o sea dispositivo sobremesa o portátil. De esta manera, y en consonancia con las nuevas características añadidas al sdk CUDA se podría generar una implementación que no solo fuese capaz de realizar las tareas implementadas en cualquier dispositivo CUDA, si no que además se adaptase de forma dinámica al uso de los recursos presentes en cada dispositivo, como por ejemplo una memoria compartida de segundo nivel presente solo en algunos dispositivos, y así poder además extraer el mayor rendimiento de cada uno en base a sus posibilidades reales.

Referencias

[A] J. M. Arrufat **Implementación eficiente del Teorema Chino del Resto**. Proyecto Fin de Master, Master en Informática Industrial, Universidad de Almería, Almería, Septiembre 2012, (2012).

[AAL] J. M. Arrufat, J.A. Älvarez-Bermejo y J.A. López-Ramos **Una implementación paralela del CRA con aplicaciones criptgráficas**. VIII Jornadas de Matemática Discreta y Algorítmica, Almería 2012, pp.267-274, (2012).

[CJM] Colin Campbell, Ralph Johnson, Ade Miller, Stephen Toub **Parallel Programming With Microsoft .NET, Design Patterns for Decomposition and Coordination on Multicore Architectures**, Microsoft (2010).

[FM] J. Fung, S. Mann **Using graphics devices in reverse: GPU-based Image Processing and Computer Vision.** Multimedia and Expo, 2008 IEEE International Conference on, pp:9-12, (2008).

[FR] Adam Freeman **Pro .NET 4 Parallel Programming in C#**, Apress (2010).

[GLN] Garland, M, Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Yao Zhang, Volkov, V. **Parallel Computing Experiences with CUDA**. Micro, IEEE, Volume 28, Issue 4, pp.13-27,(2008).

[GO] D. Gerogiannis, S.C. Orphanoudakis **Load balancing requirements in parallel implementations of image feature extraction tasks.** Parallel and Distributed Systems, IEEE Transactions on. Volume 4, Issue 9, pp:994-1013, (1993).

[GW] Rafael C. Gonzalez, Richard E. Woods **Digital Image Processing**, Capítulo 11, Prentice Hall (2002).

[K] O. Kao **Parallel and distributed methods for image retrieval with dynamic feature extraction on cluster architectures.** Database and Expert Systems Applications, 12th International Workshop on, pp:110-114, (2001).

[KDY] Jingfei Kong, Martin Dimitrov, Yi Yang, Janaka Liyanage, Lin Cao, Jacob Staples, Mike Mantor, Huiyang Zhou **Accelerating MATLAB Image Processing Toolbox functions on GPUs.** General-Purpose Computation on Graphics Processing Units, 3rd Workshop on, pp:75-85, (2010).

[KP] Junchul Kim, Eunsoo Park, Xuenan Cui, Hakil Kim, William A. Gruver **A Fast Feature Extraction in Object Recognition Using Parallel processing on CPU and GPU.** Systems, Man and Cybernetics. SMC 2009. IEEE International Conference on, pp:3842-3847, (2009).

[KSH] In Kyu Park, N. Singhal, Man Hee Lee, Cho Sungdae, C.W. Kim **Design and Performance Evaluation of Image Processing Algorithms on GPUs.** Parallel and Distributed Systems, IEEE Transactions on. Volume 22, Issue 1, pp:91-104, (2011).

[KW] David B. Kirk, Wen-mei W. Hwu **Programming Massively Parallel Processors. A Hands-on Approach**, Elsevier, (2010).

[HJZ] Adam Herout, Radovan Josth, Pavel Zemcik, Michal Hradis **GP-GPU Implementation of the "Local Rank Differences" Image Feature.** Computer Vision and Graphics, International Conference on, pp:380-390, (2008).

[M] John Miano **Compressed image file formats: JPEG, PNG, GIF, XBM, BMP**, Capítulo 2, Addison Wesley Longman (1999).

[MM] M. S. Mousavi, R.J. Schalkoff **A Parallel Distributed Algorithm for Feature Extraction and Disparity analysis of Computer images.** Parallel and Distributed Processing, 1990, IEEE Symposium on, pp:428-435, (1990).

[NV] nVIDIA **CUDA C Best Practices Guide**, nVIDIA (2012).

[NVC] nVIDIA **NVIDIA CUDA C Programming Guide**, nVIDIA (2012).

[PF] José Antonio Piedra Fernández **Caracterización e Interpretación de Imágenes Segmentadas.** Proyecto Fin de Carrera. Ingeniería Informática. Junio 2001, (2001).

[PPC] Rafael Palomar, José M. Palomares, José M. Castillo, Joaquín Olivares, Juan Gómez-Luna **Parallelizing and optimizing LIP-canny using NVIDIA CUDA.** Industrial engineering and other applications of applied intelligent systems, 23rd International conference on. Volume 3, pp:389-398, (2010).

[SK] Jason Sanders, Edward Kandrot **CUDA By Example. An introduction to General-Purpose GPU Programming**, Adisson-Wesley (2010).

[YZP] Zhiyi Yang, Yating Zhu, Yong Pu **Parallel Image Processing Based on CUDA.** Computer Science and Software Engineering, 2008 International Conference on. Volume 3, pp:198-201, (2008).