UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y AUTOMÁTICA

ACELERACIÓN DE ALGORITMOS BIOINSPIRADOS PARA ESTIMACIÓN DE MOVIMIENTO EN HARDWARE PARALELO

TESIS DOCTORAL DE:

FERMÍN AYUSO MÁRQUEZ

DIRIGIDA POR:

GUILLERMO BOTELLA JUAN

CARLOS GARCÍA SÁNCHEZ

Madrid, 2014

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Arquitectura de Computadores y Automática

TESIS DOCTORAL

Aceleración de algoritmos bioinspirados para estimación de movimiento en hardware paralelo

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR:

Fermín Ayuso Márquez

Directores:
Guillermo Botella Juan
Carlos García Sánchez

Madrid, 2013

Aceleración de algoritmos bioinspirados para estimación de movimiento en hardware paralelo

TESIS DOCTORAL

Memoria presentada para obtener el grado de
Doctor en Informática
Fermín Ayuso Márquez

Dirigida por los profesores
Guillermo Botella Juan
Carlos García Sánchez

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid

Diciembre 2013

Aceleración de algoritmos bioinspirados para estimación de movimiento en hardware paralelo.

Memoria presentada por Fermín Ayuso Márquez para optar al grado de Doctor por la Universidad Complutense de Madrid en el programa de doctorado en Ingeniería Informática, realizada bajo la dirección de Guillermo Botella Juan y Carlos García Sánchez.

Madrid, 11 de noviembre de 2013

A mi abuela Paca:

"Con el escudo o sobre el escudo,
pero nunca sin el escudo".

Agradecimientos

Cuando uno se pone a escribir esta parte de la tesis doctoral corre el riesgo de dejarse gente por el camino, así que antes de nada, mis disculpas si me dejo a alguien en el tintero.

En primer lugar quisiera expresar mis más sincero agradecimiento a los profesores Dr. D. Guillermo Botella y Dr. D. Carlos García, mis directores, por su tiempo y dedicación a lo largo del proceso de gestación de esta tesis doctoral y por sus recomendaciones, sugerencias y correcciones, que han hecho mucho mejor este texto. Agradezco toda la confianza que depositaron en mí a la hora de realizar este doctorado.

Al profesor Dr. D. Antonio Plaza y al profesor Dr. D. Manuel Prieto, por darme la oportunidad de iniciarme en el mundo de la investigación y al catedrático Dr. D. Francisco Tirado por abrirme las puertas del Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid, dónde pasé tres años estupendos en los que tuve la ocasión de empezar esta andadura.

A Hugo Vegas, Marco Antonio Gutierrez y Silvio Sepúlveda, compañeros de viaje, en los buenos y malos momentos; durante los años que trabajamos juntos en la Universidad Complutense hemos fraguado una estrecha amistad. A Javier Setoain por aconsejarme y ayudarme con los trámites, sobre todo los de última hora. A los técnicos de *ArTeCS*, Jorge, Dani y Rober, siempre dispuestos a solucionar los problemas técnicos surgidos durante los años que ha durado la realización de este trabajo. Al resto de los compañeros del Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid, por la ayuda prestada siempre que la he necesitado.

A mis padres, Fermín y Paqui Lourdes, por educarme, enseñarme, cuidarme, ayudarme y un largo etcétera. Soy quien soy gracias a ellos. Sin olvidar su labor en la lectura de cada capítulo de esta tesis, corrigiéndome y aconsejándome, aunque no entendieran siempre todo...

A mis hermanos Luis y Carlos; aunque los tengo lejos, siento su apoyo en todo momento. Gracias por estar ahí.

A mi novia Paqui, por comprenderme, intentar ayudarme siempre, facilitarme la vida y, sobre todo, por el tiempo que no le he podido dedicar, hasta ahora. Sin ese apoyo incondicional no hubiera terminado nunca esta tesis.

Quisiera agradecer también a mis tías Juana e Isa y a mis tíos César, Isidoro y Luis, cada uno ha aportado su granito de arena en este trabajo y ha influido positivamente en mi vida. Al resto de la familia, tíos y primos, por quererme, apoyarme y enseñarme lo que es una familia unida.

A mis abuelos, mis tíos Eloy y Celia y a mi prima Esther, que no pueden compartir este momento conmigo. Siempre os llevaré en la memoria.

No me puedo olvidar de Antonio, Jaime, Jesús y Alex, siempre a mi lado a las duras y a las maduras desde hace ya un *porrón* de años.

Finalmente, tengo que agradecer al resto de mis amigos el soportar pacientemente mis ausencias a lo largo de estos últimos años y por su apoyo cuando ha sido necesario, sin olvidarme de mis compañeros de trabajo, que han soportado mi tema de conversación monotemático en los últimos tiempos y me han animado.

Índice

Índice de figuras

Índice de tablas

1. Introducción.

La visión, o sentido de la vista, es una de las principales capacidades sensoriales de muchos animales, además del ser humano, ya que permite tanto extraer de forma continua información del entorno que nos rodea, como reconocer los objetos por sus colores, formas y movimiento. Los seres vivos con capacidad de percepción visual distinguimos la distancia y la profundidad de los objetos a través de sombras, estereopsis, color y texturas. En concreto, la percepción visual permite al ser humano analizar el movimiento de los objetos que nos rodean pudiendo predecir incluso eventos futuros mediante la extrapolación de la información recogida por los sentidos y así evitar situaciones de peligro. Este análisis visual, y posterior anticipación a los acontecimientos, conlleva una serie de operaciones de gran precisión que las personas realizamos de forma inconsciente, ya que hemos vivido con esta capacidad desde nuestro nacimiento.

Sin embargo, la percepción visual tiene un gran coste biológico asociado. Un claro ejemplo es el sistema que posee el ser humano para el procesamiento de la información visual mediante la corteza cerebral, que destaca por ser un sistema extremadamente complejo [1]. De forma particular, las cortezas asociativas ocupan gran cantidad de la corteza cerebral, donde complejas redes neuronales establecen circuitos que permiten asociar diversos aspectos de la percepción sensorial, como la visual [2]. Este ejemplo destaca la gran importancia y la gran cantidad de recursos, en términos de capacidad neuronal, que se necesitan para procesar la información visual con el objetivo de estructurar y tratar toda la información necesaria.

Comprender dicha complejidad e intentar procesar su volumen ha sido el principal objetivo de muchas investigaciones que han derivado en grandes avances en visión estereoscópica, seguimiento de objetos, reconocimiento de patrones, etc. Sin embargo, pese a ser testigos de innumerables avances en los campos relacionados con la visión por computador, se han desarrollado muy pocos modelos capaces de aproximarse a la eficiencia y robustez de los sistemas biológicos.

Desde el punto de vista tecnológico, hemos sido testigos de unos avances sin parangón en las capacidades computaciones disponibles en todos los ámbitos de la computación. A modo de ejemplo, podemos destacar que en la actualidad un ciudadano medio tiene a su alcance las mismas capacidades computaciones en términos de FLOPS (operaciones en punto flotante por segundo) que los grandes supercomputadores de hace tres décadas[12].

El incremento en la capacidad de integración (en la actualidad Samsung ha anunciado un prototipo de *System-on-Chip* en 14nm [3]) ha supuesto la reducción del elemento básico en la construcción de un computador, el transistor, y por ende mayores capacidades computacionales con el mismo área de silicio. Además, los esfuerzos realizados en la reducción del consumo se han traducido en una gama amplia de dispositivos al alcance de cualquiera que abarca desde computadores portátiles, videoconsolas, teléfonos móviles hasta sistemas de

[12] *Linpack* para sistemas Android: `http://www.greenecomputing.com/apps/linpack/linpack-top-10`

monitorización en medicina o tele-detección.

1.1. Motivación.

La estimación de movimiento es un tema muy abordado a lo largo de los últimos años debido a sus múltiples aplicaciones relativas al procesamiento de la señal, como son vídeo-vigilancia, disparidad binocular, *tracking*, etc. Dentro de este tema encontramos tres familias representativas de técnicas de estimación: los modelos de emparejamiento [4], los de energía [5] y los de gradiente [6]. La primera familia aplica plantillas, dando una idea del ajuste entre una determinada ventana de comparación y su ventana objetivo dentro de un espacio de búsqueda dado. Los modelos de energía hacen uso de probabilidades y esquemas bayesianos, de forma que tenemos una distribución final de soluciones válidas (poblaciones de filtros) no únicas. Podríamos resumir estos dos esquemas como una aplicación de plantillas, buscando de forma ideal un ajuste óptimo entre el movimiento y la plantilla, con problemas no triviales como el contraste del estímulo, lo cual haría necesario realizar complejas etapas de normalización.

La filosofía de los modelos de gradiente sin embargo, viene dada por una solución a la ecuación de constricción del flujo óptico:

$$\frac{\partial I}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial I}{\partial t} = 0 \tag{1}$$

Siendo I la intensidad de la imagen, donde la velocidad se calcula directamente a través de sendos cocientes de derivadas temporales y espaciales en cada punto [6]. En esta ecuación se asume la existencia de brillo constante [7, 8], asumiendo superficies *Lambertianas* que reflejan la luz con igual intensidad en todas las direcciones, manteniendo su brillo constante, y la asunción de movimientos suaves [9], es decir, aquellos que se consideran que no son bruscos, siendo la función diferenciable del brillo de la imagen.

Por otra parte, la percepción del movimiento desde el punto de vista sensorial es un tema fundamental para sobrevivir. Existen áreas en el córtex visual cuya única función es detectar movimiento [10]. Uno de los retos actuales que todavía no ha sido resuelto es una explicación plausible de cómo el sistema visual puede calcular la velocidad del movimiento a partir de los cambios espaciales y temporales de la imagen proyectada en la retina [11].

El modelo neuromórfico en el que se basa esta tesis [12] recoge conocimientos de la biología y la fisiología cortical y está basado en una estructura de operadores diferenciales espaciales y temporales que luego se rotan en el espacio, acelerados mediante arquitecturas específicas de *hardware* gráfico [13].

Bajo estas premisas, esta tesis pretende converger hacia el estudio de un modelo de visión por computación basado en el comportamiento de los seres vivos denominado "bioinspirado", explotando un sistema con un acelerador gráfico, siendo este esquema la tendencia actual en sistemas tan dispares como supercomputadores de alto rendimiento o smartphones.

1.2. Visión por computador / Sistemas de visión artificial.

Desde hace unos años, la necesidad de automatizar la estimación de movimiento ha sido un tema muy abordado, ya que tiene múltiples aplicaciones en diferentes campos como la vigilancia, el control de tráfico, la robótica, etc. La visión artificial, también conocida como visión por computador (*Computer Vision*), es un subcampo de la inteligencia artificial que incluye métodos para adquirir, procesar, analizar y entender imágenes y, en general, grandes cantidades de datos procedentes del mundo real, con el objetivo de producir información numérica o simbólica [14, 15, 16]. Un gran reto en este campo ha sido el desarrollo de modelos que repliquen la capacidad de la visión humana por la vía de la percepción electrónica y la comprensión de una imagen [17]. Uno de los propósitos principales de la visión artificial es construir un sistema capaz de interpretar una escena o alguna característica de una imagen, cuyos principales objetivos enumeramos a continuación:

- La detección, segmentación, localización y reconocimiento de ciertos objetos en una imagen, como por ejemplo, el reconocimiento facial.
- La evaluación de resultados.
- El registro de diferentes puntos de vista de la misma escena u objeto.
- La correlación de una escena a un modelo tridimensional, de forma que tal modelo pueda ser utilizado por un robot para navegar por la escena fotografiada.
- El seguimiento de un objeto a través de una secuencia de imágenes.

Estos, y otros objetivos, se consiguen por medio de reconocimiento de patrones, aprendizaje estadístico, geometría proyectiva, procesamiento de imágenes, teoría de gráficos y otros métodos. La visión cognitiva por ordenador está fuertemente relacionada con la psicología cognitiva y la computación biológica.

Dentro del campo de visión por computador existe una clasificación [18] (figura 1) en función de la granularidad del problema que ha sido ampliamente aceptada por la comunidad científica:

- Procesamiento de nivel bajo [19], donde se trabaja directamente con los *pixels* de las imágenes para extraer las propiedades de éstas, como el color, la textura, los bordes, etc. En este nivel se analiza la imagen y conlleva un bajo nivel de construcción de primitivas y codificación.
- Procesamiento de nivel intermedio [20], que consiste en agrupar los elementos obtenidos en el nivel bajo para obtener contornos y regiones, entre otras características. La segmentación y el movimiento se pueden englobar en este nivel.
- Procesamiento de alto nivel [21], donde se reconocen eventos y objetos específicos. Consiste en la interpretación de los datos obtenidos en los niveles anteriores. Para ello, se utiliza la visión basada en modelos y la basada en conocimiento, con problemas que reutilizan primitivas diseñadas en niveles anteriores, con un alto nivel de codificación.

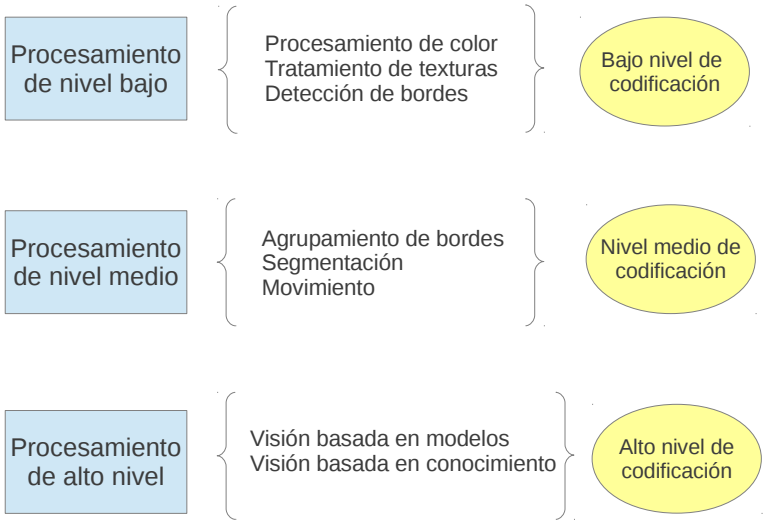


Figura 1: Niveles de procesamiento de la visión computacional [22].

Un reto al que nos enfrentamos es lograr implementar un sistema con estas características cuyo tiempo de respuesta se acerque a los requisitos del tiempo real, porque este tipo de sistemas son ampliamente utilizados en la vida cotidiana y pueden tener un alto impacto tecnológico. A modo de ejemplo, los sistemas basados en visión artificial empleados en campos relacionados con las imágenes médicas se venden actualmente por miles de millones de dólares cada año.

Otro ejemplo, son los sistemas de visión artificial como ayuda a la automatización. Al igual que en una cadena de montaje hay algunos trabajadores que tienen que inspeccionar las piezas para poder juzgar la calidad del producto final, los sistemas de visión, a través de cámaras digitales y software de procesamiento de imágenes, realizan inspecciones y tomas de decisiones basándose en el análisis de imágenes digitales. Estos sistemas están programados para realizar tareas estrictamente definidas, aunque hasta ahora muy pocos tenían ni la inteligencia ni la capacidad de aprendizaje. Lo que se espera en muchos casos de estos sistemas es una alta velocidad en el procesamiento de las imágenes, operabilidad las 24 horas y la posibilidad de repetir las mediciones.

Los componentes básicos de un sistema de visión artificial son los que se detallan a continuación. Algunos de ellos se encuentran en la misma máquina, como es el caso del sensor y la iluminación, que se incorporan a la cámara.

- Sensor óptico. Para determinar cuándo un objeto en la escena se mueve.
- Cámara. Para tomar las imágenes de entrada y poderlas procesar.

- Iluminación. Para resaltar objetos o características de la escena que se quieren observar y minimizar la aparición de otros objetos u otras características que no son interesantes.
- Interfaz entre la cámara y el circuito diseñado/PC, conocida como *framegrabber*. Para transformar la salida de la cámara en formato digital y poder transmitir esta información.
- Software. Para procesar las imágenes recogidas por la cámara. Este tipo de software normalmente requiere realizar varios pasos para procesar dichas imágenes como reducir el ruido o transformar escalas de grises en una combinación de blanco y negro.
- Señales digitales o conexiones de red. Para presentar los resultados.

Uno de los usos principales de los sistemas de visión artificial son la inspección y guiado automático de robots industriales [23]. Las aplicaciones más comunes incluyen garantía de calidad, clasificación, manejo de materiales, orientación del robot y medición óptica.

En este tipo de usos, uno de los problemas más importantes que se necesitan solventar es poder procesar toda la información que reciben de entrada en tiempo real, para así poder responder de una manera adecuada en el tiempo deseado. Para realizar las tareas de procesado de la información recogida por el sistema podrían ser suficientes los procesadores de propósito general, sin embargo, puesto que se espera que las respuestas se realicen en tiempo real, estas tareas pueden hacerse de una manera más rápida haciendo uso de aceleradores *hardware*. Actualmente existen en el mercado aceleradores *hardware* que permiten esta tarea como son las FPGAs[13] (*Field Programmable Gate Array*), las GPUs[14] (*Unidades de Procesamiento Gráfico*) o ciertos circuitos específicos ASIC[15]

1.3. Hitos arquitectónicos en los últimos años.

Gracias a los procesadores de propósito general podemos automatizar y emular la estimación de movimiento realizada por los seres vivos que poseen percepción visual. Estos dispositivos han ido evolucionando su arquitectura a lo largo de los años incorporando mejoras orientadas a la capacidad de cálculo y al consumo, entre otras.

El volumen y la complejidad de los datos que los computadores de hoy en día tienen que procesar sigue aumentando de manera exponencial. A su vez, el rendimiento que se puede lograr aumentando la frecuencia de reloj de un microprocesador ha llegado a los límites físicos, lo que ha provocado la búsqueda de otras soluciones a nivel arquitectónico.

En los últimos años hemos visto la aparición de técnicas arquitectónicas como *SIMD* [24] (del inglés **S***ingle* **I***nstruction,* **M***ultiple* **D***ata*), de microprocesadores con múltiples núcleos (*multicores*) o la posibilidad de utilizar las GPUs, inicialmente pensadas únicamente para el renderizado de gráficos, con el objetivo de acelerar los cómputos necesarios.

[13] Definición FPGA: `http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array`
[14] Nvidia GPU: `http://www.nvidia.es/object/gpu-computing-es`
[15] ASIC: `http://userwww.sfsu.edu/necrc/files/synopsys`

1.3.1. SIMD. Una instrucción, múltiples datos.

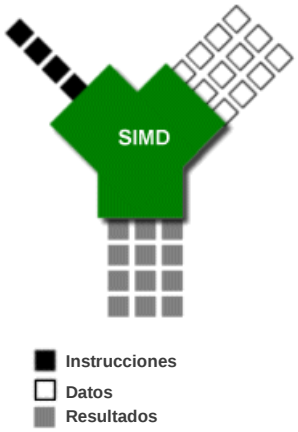


Figura 2: Taxonomía de Flynn [25, 26] del diseño SIMD.

Los repertorios SIMD son un conjunto de instrucciones que permiten acelerar el rendimiento de las aplicaciones, ya que su utilización conlleva la ejecución de menos instrucciones al procesarse varios elementos de datos a la vez (figura 2). Aunque la primera vez que se diseñaron instrucciones SIMD fue en los años 70 [27], en supercomputadores vectoriales como el CDC Star-100[16], no es hasta 1996 cuando se incorporan en computadoras de sobremesa. En concreto, la primera implementación la lleva a cabo Intel en su arquitectura IA-32 en los procesadores *Pentium MMX*[17] agregando 8 nuevos registros a la arquitectura (los llamados registros MMn). Para simplificar el diseño se reutilizaban los ocho registros de la Unidad de Coma Flotante (FPU), por lo que era necesario habilitar el procesador en un modo u otro, haciendo altamente costoso el cambio de modo de ejecución. Entre 1997 y 1998 Intel lanza la segunda versión de MMX (MMX2) para intentar corregir algunas de las limitaciones encontradas en la especificación original.

El fabricante Intel, en 1999, introduce una extensión a MMX, de la arquitectura $\times$86, llamada *SSE* (del inglés *Streaming SIMD Extensions*) [28] en la que se añaden otros 8 nuevos registros de 128 bits cada uno (XMM0 - XMM7) y un registro adicional de control (MXCSR), que trabajan mayormente con datos en coma flotante de precisión simple. Estas instrucciones son especialmente adecuadas para el procesamiento de gráficos 3D, *software* de reconocimiento de voz o para decodificar MPEG2, que es un *codec* de audio y vídeo ampliamente utilizado. A diferencia de MMX, la utilización de SSE no implicaba inhabilitar la FPU.

[16] CDC Star-100: `http://homepages.inf.ed.ac.uk/cgi/rni/comp-arch.pl?Vect/star100.html,Vect/star100cpu-f.html,Vect/menu-cyb.html`

[17] Conjunto de instrucciones MMX: `http://www.plantation-productions.com/Webster/www.artofasm.com/Windows/HTML/TheMMXInstructionSet.html`

Desde su aparición hasta ahora han ido surgiendo diferentes versiones que han ido mejorando dicho repertorio. La versión SSE2, introducida con los *Pentium IV* en 2001, añadía instrucciones para doble precisión (64-bit) y extendía las instrucciones enteras de MMX para operar sobre 128-bit en los registros XMM. La principal ventaja que posee esta versión es que permite trabajar con estos tipos de datos sin emplear las instrucciones de la Unidad de Coma Flotante (FPU) ni la extensión MMX. En total se agregaron con esta nueva versión unas 144 instrucciones, haciendo de esta una de las mejoras más significativas dentro de la tecnología SSE.

La extensión SSE3, también conocida como *Prescott New Instructions* (PNI), se presentó en 2004 con los procesadores *Pentium IV* 5xx y brindaba un nuevo conjunto de instrucciones matemáticas y de manejo de procesos. Con la aparición de los procesadores *Intel Core 2 Duo* y Xeon en 2006, se presenta la versión SSSE3 (*Supplemental SSE3*) para mejorar la velocidad de ejecución proporcionando 32 nuevas instrucciones. En 2007 aparece la extensión SSE4, que añade instrucciones adicionales para enteros, una para el producto escalar, etc. A principios de 2008 Intel lanza AVX (*Advanced Vector Extensions*) [29] como una versión avanzada de SSE que, entre otras mejoras, amplia el *path* de datos de 128 bits a 256 bits, aunque no ha sido implementado por sus procesadores hasta el primer trimestre del año 2011 en su gama de procesadores *Sandy Bridge*. AVX introduce un formato SIMD de 3 y 4 operandos donde el operando destino es diferente de los operandos fuente, haciendo que estas instrucciones sean no destructivas.

En primavera de 2009 se empieza a oír hablar de un nuevo conjunto de instrucciones llamado *Larrabee*[18] (LRBni), en el que Intel introducía un nuevo conjunto de instrucciones vectoriales, añadiendo dos tipos de registros a la arquitectura: 32 nuevos registros vectoriales de 512 bits y 8 nuevos registros para máscaras vectoriales de 16 bits. Las instrucciones pueden tener dos anchos, 16 y 8, de esta manera se puede operar una instrucción LRBni en los registros vectoriales, como se puede observar en la figura 3.

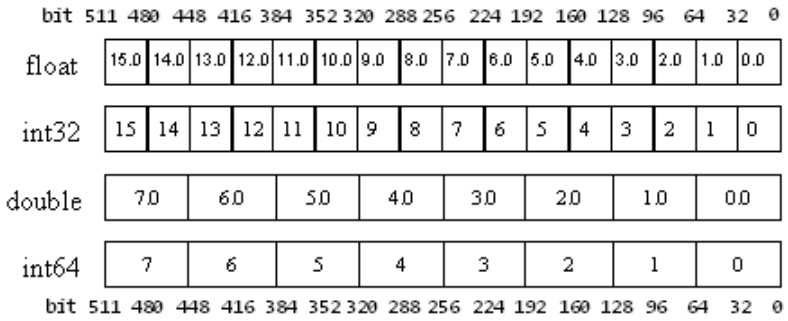


Figura 3: Tipos de datos vectoriales soportados por LRBni.

[18]Información sobre Larrabee: `http://software.intel.com/en-us/articles/larrabee`

Por último, y continuando con la tendencia de añadir más funcionalidades vectoriales, se espera que con *Haswell*[19],(microarquitectura de nuevos procesadores de Intel que tienen prevista su salida a lo largo del 2013), se amplíe el juego AVX (en lo que se conocerá como AVX2 o *Haswell New Instruction*[20]) con nuevas instrucciones que funcionarán también sobre números naturales.

Por su parte, si hablamos de AMD, en 1998 introduce en sus procesadores K6-2 la extensión multimedia *3DNow!*[21], un añadido de instrucciones SIMD para obtener mayor rendimiento en el procesamiento de vectores. Esta extensión constaba de 21 instrucciones que soportaban operaciones en punto flotante tipo SIMD, no siendo hasta 2001 cuando AMD decide incluir SSE en sus procesadores *Athlon XP*. Con la aparición del núcleo *Venice* en 2004 se incluye el conjunto de instrucciones SSSE3 en los procesadores de AMD. Ya en 2007, AMD anuncia el conjunto de instrucciones SSE5 como suplemento a las instrucciones SSE de 128 bits presentes en la arquitectura AMD64. Estas incluyen innovaciones en la arquitectura $\times 86$ como un conjunto de instrucciones condicionales y permutaciones para el movimiento de datos, aunque en 2009 AMD las remplaza por los conjuntos de instrucciones XOP (*eXtended Operations*) y CVT16 (*half-precision floating point converts*), capaces de codificar las instrucciones de forma compatible con el conjuto de instrucciones AVX propuesto por Intel.

La futura extensión del conjunto de instrucciones SIMD para 128 y 256 bits en los microprocesadores $\times 86$ recibe el nombre de FMA ya que soportará operaciones de este tipo (del inglés *Fused Multiply-Add*). Existen dos variantes, la FMA3 (*three-operand Fused Multiply/Add*) y la FMA4 (*four-operand Fused Multiply/Add*), donde la principal diferencia entre ambas radica en que una operación puede tener tres operandos o cuatro, respectivamente. La ventaja de tener tres operandos es la de acortar el código y hacer la implementación *hardware* más sencilla, mientras que con cuatro operandos existe mayor flexibilidad a la hora de programar. Cabe destacar que, aunque ambas versiones son funcionalmente idénticas, no son compatibles.

En enero de 2012 AMD anuncia que los procesadores *Trinity* y *Vishera*, basados en la arquitectura *Piledriver*, soportan FMA3, mientras que Intel no la soportará hasta la aparición de los procesadores de la familia *Haswell*, en 2013, y los de la familia *Broadwell*, en 2014.
Con respecto a la variación FMA4, la arquitectura *Bulldozer* de AMD soporta esta extensión. Actualmente coexisten FMA3 y FMA4, pero es difícil prever cuál de los dos dominará en el futuro. También es posible que los próximos procesadores soporten ambas formas.

Aunque se ha centrado el presente estudio preliminar en AMD e Intel, las extensiones SIMD no son exclusivas de estas tecnologías. La tabla 1 muestra otros fabricantes y alternativas, como Motorola o Sony/Toshiba, que ofrecen instrucciones SIMD con *AltiVec*, para enteros y punto flotante.

[19] Noticia sobre Haswell y AVX2: `http://www.madboxpc.com/intel-haswell-22nm-soportara-instrucciones-avx2/`
[20] Descripción del conjunto de instrucciones AVX2: `http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/`
[21] AMD 3DNow!: `http://www.amd.com/us/products/technologies/3dnow/Pages/3dnow.aspx`

Fabricante	Arquitectura	Tecnología	Año
Intel	P6/Pentium M	MMX, SSE, SSE2, SSE3	1995
	NetBurst	SSE2	2000
	Core	SSE3, SSSE3, SSE4, SSE4.1	2006
	Nehalem	SSE4.2	2007
	Sandy/Ivy Bridge	AVX	2009/2011
	Haswell	AVX2	2013
HP	PA-RISC 2.0	MAX-2 [30] (*MultiMedia Acceleration eXtension*)	1996
AMD	K6	MMX, 3DNow!	1998
	K7	MMX, 3DNow!+, SSE	1999
	K8	MMX, 3DNow!+, SSE2, SSE2, SSE3	2003
	K10/K10.5	MMX, SSE, SSE2, SSE3, SSE4a, Enhanced 3DNow!	2007
	Bulldozer	AVX, SSE4.2, SSE4.1, XOP, CVT16, MMX, SSE, SSE2, SSE3, SSE4a, Enhanced 3DNow!	2011
Compaq (digital)	Alpha	MVI (*Motion Video Instruction*)	Mediados 90's
Motorola	PowerPC G4/G5	Velocity Engine (AltiVec)	1997
ARM	ARMv6	NEON	2002
	PPC970	VMX	2002
Sony/Toshiba	Cell (PPE)	AltiVec	2001

Tabla 1: Resumen de fabricantes y tecnologías con SIMD.

1.3.2. Procesadores con varios núcleos.

Durante décadas, el rendimiento de una CPU se podía mejorar reduciendo el área del circuito integrado, que a su vez suponía una bajada en el coste. De manera alternativa, se podían utilizar mayor número de transistores para una misma zona del circuito, lo que provocó el aumento en su funcionalidad y mayores frecuencias de reloj, pudiendo pasar de los MHz que se obtenían en la década de los 80, hasta llegar a los GHz de los años 2000. Mantenerse al día con la Ley de Moore [31, 32] ha resultado cada vez más difícil, ya que las tecnologías de fabricación de chips se han ido acercando a sus límites físicos. En respuesta, los fabricantes de microprocesadores buscaron otras maneras de mejorar el rendimiento [33]. Como consecuencia del consumo derivado de aumentar la frecuencia de reloj y a la disipación de energía, la computación paralela en procesadores multicore apareció como alternativa, haciendo posible tener más de un procesador en el mismo circuito integrado.

Un procesador multicore es simplemente un chip que contiene más de un núcleo o *core* [34]. Esto permite que el rendimiento potencial del procesador se multiplique por el número de núcleos (siempre y cuando el sistema operativo y el *software* estén diseñados para ello), haciendo posible la compartición entre núcleos de algunos componentes como la *cache*. Otra

ventaja que poseen los procesadores multicore es que, al encontrarse muy cerca físicamente, se pueden comunicar más rápido de lo que lo harían procesadores separados en un sistema multiprocesador, posibilitando así mejoras en el rendimiento global.

IBM lanzó el primer procesador multicore, llamado Power 4 [35], en 2001 para sus servidores RISC, pero no es hasta 2005 cuando AMD e Intel dan a conocer los primeros computadores personales que poseían doble núcleo. A partir de 2009 los sistemas PC o portátiles vienen utilizando los procesadores *dual-core* y los *quad-core*, mientras que en mercados profesionales con sistemas de servidores o estaciones de trabajo, se utilizan procesadores con un número de núcleos que varía entre 4 y 16. Otro ejemplo es el de *Sun Microsystems*, que lanzó los chips Niagara y Niagara 2, los cuales cuentan con un diseño de ocho núcleos además, el Niagara 2 soporta más hilos y funciona a 1,6 GHz.

En 2012 Intel lanzó al mercado la familia de procesadores multicore, conocida con el nombre en clave *Ivy Bridge*, que incluyen, entre otras características, una tecnología de fabricación de los microprocesadores de 22 nm, un *pipeline* de 14 fases (heredado de la arquitectura Intel Core) y transistores Tri-Gate [36], que implica menos del 50 % de consumo energético al mismo nivel de rendimiento respecto de los transistores planos. Actualmente en el mercado podemos encontrar el procesador multicore *Intel Xeon Processor E7-8870*[22], de 32 nm, que posee 10 núcleos y una velocidad de reloj de 2.8 GHz o el modelo *Intel Core i7-3920XM Processor Extreme Edition*[23], de 22 nm, que posee 4 núcleos y una frecuencia de reloj máxima de 3.8 GHz. Por su parte, la compañía AMD sacó la familia de procesadores AMD FX[24] a lo largo del 2012, con hasta 8 *cores* en un mismo procesador y 4 GHz de frecuencia de reloj.

A la próxima generación de microprocesadores *multicore* que Intel lanzará a lo largo del año 2013 se la conoce con el nombre en clave *Haswell*, que incorporarán el conjunto de instrucciones AVX2 (que incluyen *gather*[25], la expansión a 256 bits de la mayoría de las instrucciones enteras de AVX y soporte de FMA3, entre otras características).

[22] Especificaciones del procesador Intel Xeon E7-8870: `http://ark.intel.com/m/products/53580/Intel-Xeon-Processor-E7-8870-30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI`

[23] Características del procesador Intel Core i7-3920XM `http://ark.intel.com/products/64887/intel-core-i7-3920xm-processor-extreme-edition-8m-cache-up-to-3_80-ghz`

[24] Procesadores AMD FX: `http://www.amd.com/es/products/desktop/processors/amdfx/Pages/amdfx.aspx`

[25] Se podrá acceder a la vez a varias posiciones no contiguas en memoria, aumentando considerablemente las capacidades de procesado vectorial de la arquitectura ×86-64

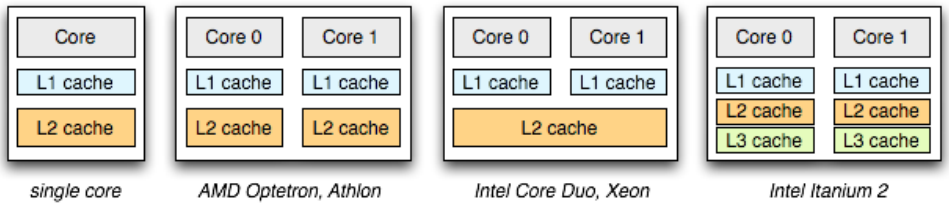


Figura 4: Distintos esquemas *multicore*.

La figura 4 compara de forma esquemática un procesador con un núcleo, con varias variantes de procesadores con dos núcleos, pudiendo estos compartir los distintos niveles de *cache* o no. En este caso, la diferencia entre el procesador AMD Opteron Athlon y el Intel Core Duo Xeon es que el primero posee una *cache* de nivel 2 para cada núcleo mientras que el segundo la comparte entre ambos. La ventaja de compartir niveles de *cache* es que permite una mayor utilización, lo que puede suponer un ahorro de energía, así como mayores tasas de acierto en determinados escenarios. Sin embargo, esa misma memoria *cache* compartida puede crear fácilmente conflictos a la hora de obtener recursos. En el caso del Intel Itanium 2 se puede observar como cada núcleo posee 3 niveles de *cache* (L1, L2 y L3) independientes.

Por su parte, la figura 5 muestra un esquema más actual, el de un procesador *Ivy Bridge*, en el que se pueden observar cuatro núcleos que comparten el tercer nivel de *cache*. Además, en el chip se incluye el procesador gráfico.

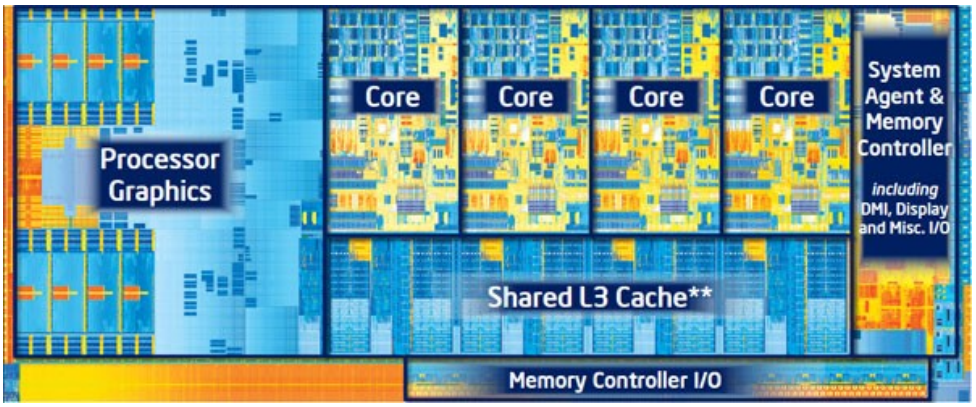


Figura 5: Esquema *multicore* del procesador Ivy Bridge. *Fuente: Intel*.

También son de especial interés los procesadores *multicore* asimétricos que poseen el mismo repertorio de instrucciones, o al menos un conjunto común amplio, haciendo de esta forma

más simple el desarrollo *software*, uno de los principales obstáculos para la adopción de arquitecturas heterogéneas a una escala más global. A este tipo de procesadores se les conoce con el nombre de ASISA (*Asymmetric Single-ISA*) [37] y han sido propuestos como una alternativa más eficiente, en términos de consumo-rendimiento, a las arquitecturas *multicore* homogéneas, entre otros motivos, por la posibilidad de poseer, en un mismo procesador, *cores* con distinto balance consumo-rendimiento en función del tipo de aplicaciones que ejecuten. Actualmente no existen procesadores comerciales de este tipo. La figura 6 compara, con un esquema, una arquitectura homogénea con otra asimétrica.

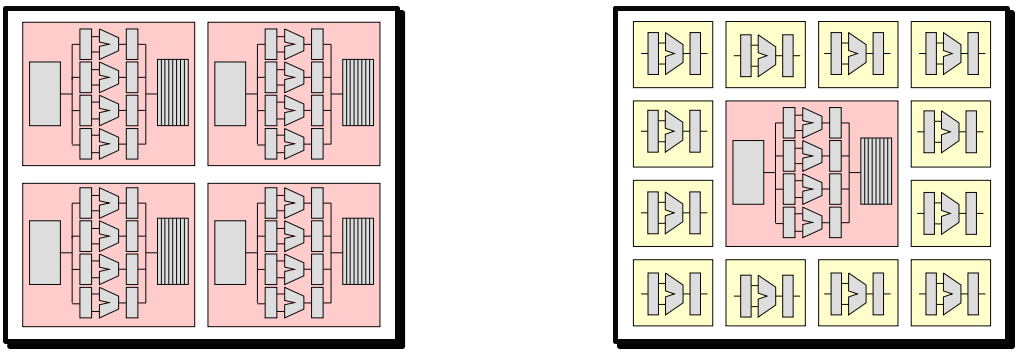


Figura 6: Arquitectura homogénea (izquierda) frente a arquitectura asimétrica (derecha).

1.4. Aceleradores consolidados.

Uno de los aceleradores *hardware* consolidados en el mercado es el gráfico, debido a que se encuentran disponibles en una amplia gama de dispositivos electrónicos. Su relativo bajo coste ha motivado su extensión y aceptación. Su desarrollado ha venido a la par que el incremento de uso gracias a la industria de los videojuegos. Estos dispositivos están basados en sistemas multinúcleo con una jerarquía de memoria compleja. Estas plataformas están diseñadas para aprovechar el alto grado de paralelismo de datos en el renderizado de escenas en 3D. Sin embargo, se pueden utilizar hoy en día como coprocesadores paralelos mediante la ejecución de un alto número de *threads* simultáneamente. A modo de ejemplo, empleando tecnología actual, un chip de *NVIDIA* GK110 [38] (Tesla K20X) alcanza un rendimiento máximo de 3,95 TeraFLOPs en precisión simple con 2688 *cores* incorporando la nueva arquitectura *Kepler*, mientras que un procesador Intel Core i7-3930 [39] únicamente puede completar 154 GigaFLOPs. Esta sorprendente potencia de cómputo ha servido para llamar la atención de muchos programadores y científicos procedentes de múltiples áreas, que están utilizando las GPUs actuales como sistemas paralelos que permiten acelerar sus propias aplicaciones. Como se puede apreciar en la figura 7, el uso de este tipo de arquitecturas está cada vez más extendido, pudiendo encontrar dentro del TOP 500 [40] de supercomputadores un incremento de su utilización comparado con el año anterior. Como ejemplo, si analizamos las características del primer supercomputador del TOP500 (consultado en noviembre del 2012), el Cray Titan, podemos observar que posee 299.008 AMD Opteron *cores* y 18.688 aceleradores GPU de Nvidia del tipo Tesla K20 [41].

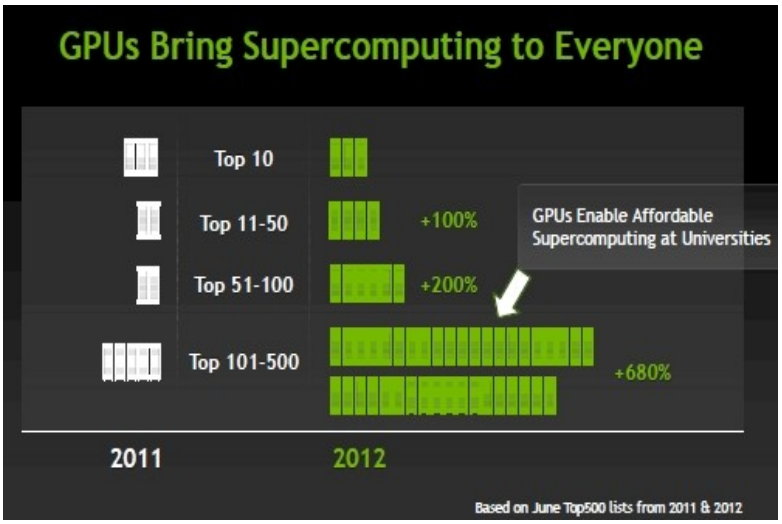


Figura 7: Computación de altas prestaciones: TOP500.

Como complemento al TOP500, podemos consultar también la lista Green500 [42], que clasifica el TOP500 de supercomputadores según la eficiencia energética. El objetivo de "rendimiento a cualquier precio" en operaciones computacionales ha llevado a la aparición de supercomputadores que consumen grandes cantidades de energía eléctrica y que producen tanto calor que deben construirse grandes instalaciones de refrigeración para asegurar su correcto funcionamiento. Para hacer frente a esta tendencia, desde 2007 la lista Green500 premia el rendimiento con eficiencia energética para la supercomputación sostenible. En este ranking podemos observar como el Cray Titan ocupa la tercera posición.

Otro ejemplo de cómo las líneas de investigación actuales persiguen el bajo consumo sería el proyecto europeo Mont-Blanc [43], que tiene como principal objetivo desarrollar un supercomputador energéticamente eficiente con tecnología disponible de bajo consumo, intentando obtener los mejores resultados medidos en MFLOPS/W. Este proyecto se ha decantado por la plataforma Samsung Exynos para construir su prototipo de supercomputador que integrará tecnología de los teléfonos móviles y tablets. Así, el chip escogido es el Samsung Exynos 5 Dual[26], que utilizan tanto el Samsung Chromebook como el Nexus 10, que llevan un procesador Samsung Exynos de dos núcleos ARM Cortex A-15 [44] a 1.7 GHz y una GPU ARM Mali T-604 [45].

[26] Información sobre el chip Exynos 5 Dual de Samsung: `http://www.samsung.com/global/business/semiconductor/minisite/Exynos/products5dual.html`

1.5. Objetivo general, organización de la tesis doctoral y principales contribuciones derivadas.

Bajo las premisas antes expuestas, y ante la necesidad actual de encontrar soluciones eficientes desde el punto de vista energético, en esta tesis se abordará la aceleración hardware de un modelo de flujo óptico robusto fuertemente bioinspirado. La principal motivación de esta tesis se sustenta en la mayor presencia de aceleradores hardware en los dispositivos de la vida cotidiana. A modo de ejemplo, encontramos soluciones arquitectónicas que incorporan GPUs, no solamente en computación de altas prestaciones, sino que hoy en día también está presente hasta en los dispositivos móviles de última generación, ya mencionados al principio de este capítulo.

El siguiente trabajo está organizado en seis capítulos, de forma que en el capítulo 2 se aborda la estimación de movimiento desde el punto de vista de la computación, profundizando en las distintas familias de algoritmos existentes y analizando trabajos e implementaciones previas en distintos tipos de aceleradores. Además, se presentan también los estímulos que han servido para el desarrollo de este trabajo, así como las métricas de error consideradas. El capítulo 3 expone en profundidad el algoritmo en el que se ha basado esta tesis (McGM) y la teoría en la que se basa, describiendo las principales bondades que ofrece respecto a otros algoritmos. En el capítulo 4 se profundiza en los procesadores gráficos o GPUs, su historia y estado del arte, sus últimas arquitecturas, tendencias futuras y su modelos de programación. El capítulo 5 presenta la implementación de McGM en GPUs y se analiza si es conveniente el uso de estas tecnologías para alcanzar requisitos de tiempo real. Como consecuencia de este último estudio, en el capítulo 6 se presentan técnicas de reducción del consumo de memoria para hacer viable su uso en GPUs. Para finalizar el trabajo, en el capítulo 7 se presentan las principales conclusiones de esta tesis, las posibles líneas que podrían surgir a raíz de este trabajo y una breve descripción de la extensión en el ámbito de ingeniería biomédica.

Esta tesis doctoral viene avalada por las publicaciones que se detallan a continuación, ordenadas según su fecha de publicación:

- "GPU-Based Signal Processing Scheme for Bioinspired Optical Flow", F. Ayuso, C. Garcia, G. Botella, M. Prieto, F. Tirado, en *IEEE 21th International Conference on Field Programmable Logic and Applications* (pág. 2011-2011), Septiembre 2011 [46].
- "Pre-procesamiento de Flujo Óptico Robusto en Hardware Gráfico", F. Ayuso, C. Garcia, G. Botella, M. Prieto, F. Tirado, en *XXII Jornadas de Paralelismo* (pág. 323-328), Universidad de la Laguna. Septiembre 2011 [47].
- "GPU-based acceleration of bioinspired motion estimation model", F. Ayuso, C. Garcia, G. Botella, M. Prieto, F. Tirado, en *4th Workshop on Parallel Architectures and Bioinspired Algorithms*, Octubre 2011 [48].
- "GPU-based acceleration of bio-inspired motion estimation model", F. Ayuso, C. Garcia, G. Botella, M. Prieto, F. Tirado, en *Concurrency and Computation: Practice and Experience*,vol. 25 (8): pág. 1037-1056. Octubre 2012 [49].

- "Multi-GPU based on multicriteria optimization for motion estimation system", C. García, G. Botella, F. Ayuso, M. Prieto, F. Tirado, Francisco, en *EURASIP Journal on Advances in Signal Processing*, vol. 2013 (1): pág. 1-12. Febrero 2013 [50].
- "Implementation of a Low-Cost Mobile Devices to support Medical Diagnosis", C. García, G. Botella, F. Ayuso, D. González, M. Prieto, F. Tirado, en *Computational and Mathematical Methods in Medicine*. En Prensa [51].

2. Estimación de movimiento.

2.1. Introducción y estado del arte.

La estimación de movimiento es uno de los problemas fundamentales en el ámbito de la visión por computador o del tratamiento de vídeo digital. El objetivo es calcular el campo de vectores que describe el movimiento aparente entre dos fotogramas de una secuencia. Hablamos de movimiento aparente porque no hay que olvidar que las señales de imagen en movimiento son la proyección en el plano, en instantes discretos de tiempo, de escenas tridimensionales. Como consecuencia se observa una pérdida de información que hace necesaria la distinción entre el movimiento real que se proyecta sobre el plano y el movimiento aparente que, junto con su velocidad, es lo que el observador nota mediante la variación del contenido visual de las imágenes de la secuencia. Un efecto de lo anterior es que, en condiciones en las que no existe suficiente gradiente espacial en la imagen o se encuentran cambios en la iluminación, el flujo óptico (velocidad aparente provocada por los cambios de intensidad de la imagen) y el movimiento real no tienen por qué coincidir, teniendo este proceso una alta complejidad de cálculo en la codificación de vídeo [52]. Como es usual en el procesamiento de señales, nos encontraremos con un compromiso entre el tiempo de procesado y el tamaño de los datos respecto a la calidad del vídeo comprimido. Por ello, en la actualidad, muchas de las investigaciones dentro el campo de la codificación de vídeo se centran en buscar algoritmos que puedan realizar de manera más eficiente la estimación de movimiento.

En este capítulo se examinarán algunos de los algoritmos utilizados con más frecuencia en visión por computador y estimación del flujo óptico, su eficiencia computacional y su inspiración biológica. Se han planteado una gran cantidad de métodos para la estimación del flujo óptico, que derivan de la teoría de la señal, la inteligencia artificial, la robótica, la psicología y la biología. Es posible encontrar una amplia bibliografía al respecto, de la que se hará un estudio lo más descriptivo posible sin tener que exponer todos los algoritmos existentes, puesto que no es el objetivo de este capítulo, con el fin de poder justificar el modelo explicado en el capítulo 3 dentro de un entorno suficientemente clasificado, describiendo aquí sólo las cuestiones más relevantes e importantes dentro de esta disciplina.

Cada campo ofrece su particular motivación y aportación al problema, dando como resultado una gran gama de diversos algoritmos. Sin embargo, pese a la existencia de una gran diversidad, los modelos de flujo óptico se pueden clasificar en tres categorías. Además, se van a repasar las implementaciones existentes hasta ahora de estas familias de algoritmos en distintos tipos de aceleradores como son las FPGAs, las GPUs o los circuitos específicos.

2.2. Familias de algoritmos.

Dependiendo de cada aplicación, es apropiado utilizar una u otra aproximación a la estimación de movimiento. El teorema del muestreo [53] expone que la reconstrucción exacta de una señal periódica continua en banda base a partir de sus muestras, es matemáticamente posible si la señal está limitada en banda y la tasa de muestreo es superior al doble de su

ancho de banda. De esta forma se garantiza que, en el campo del procesamiento de imágenes en movimiento, el desplazamiento entre 2 fotogramas es pequeño si lo comparamos con la escala del patrón de entrada. Si deja de cumplirse este teorema, aparece el fenómeno de *aliasing*[27], lo que en imágenes espacio-temporales produce estructuras sin ninguna relación entre sí o falsas inclinaciones. Como ejemplo de *aliasing* temporal, puede observarse como las aspas de un ventilador a veces parecen girar en sentido inverso del que en realidad lo hacen, cuando se les filma o cuando son iluminadas por una fuente de luz parpadeante. En definitiva, no se pueden estimar desplazamientos largos procedentes de patrones de entrada con escalas pequeñas. Adicionalmente a este problema, se tiene el denominado problema de la apertura[28] [54, 8, 55]. Estos dos problemas (*aliasing* y apertura) conforman el llamado *problema global de la correspondencia* [56].

Conforme a un acuerdo general, existen tres categorías para organizar los algoritmos y técnicas utilizadas para estimar el flujo óptico [57], que son: los modelos de energía, los modelos de emparejamiento y los modelos de gradiente. A continuación se describe cada uno de ellos.

2.2.1. Modelos de emparejamiento.

También conocidos como *Block Matching* o métodos basados en correspondencia [58], el procedimiento de estos modelos consiste en la comparación de posiciones dentro de la estructura de la imagen entre fotogramas contiguos. De esta forma infieren la velocidad a partir del cambio en cada lugar. Probablemente son los métodos más intuitivos para recuperar dirección de movimiento y velocidad [4].

Cada una de las imágenes pertenecientes a una secuencia de vídeo se divide en bloques rectangulares (generalmente cuadrados) denominados macrobloques. El método pretende detectar el movimiento entre imágenes con respecto a los macrobloques que las constituyen. Los bloques del fotograma actual son cotejados con los bloques del fotograma de destino o de referencia (anterior al actual, generalmente el primero), deslizando el actual a lo largo de una región concreta de píxeles del fotograma de destino (figura 8). Los cambios en la posición indican movimiento en el tiempo, es decir, velocidad.

Los criterios de semejanza se utilizan para medir la similitud entre bloques de imágenes. Uno de los más utilizado es el Sumatorio de Diferencias Absolutas (SAD, *Sum of Absolute Differences*) [59], que se define de la siguiente manera:

$$SAD(x,y;u,v) = \frac{1}{N^2}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}|I_t(x,y) - I_{t-1}(x+u,y+v)| \qquad (2)$$

Siendo I_t(x,y) el valor del *pixel* en la coordenada (x,y) en el fotograma t; (u,v) representa el desplazamiento del macrobloque candidato y N el tamaño del bloque. Otros criterios que

[27] Efecto que causa que señales continuas distintas se tornen indistinguibles cuando se muestrean digitalmente.
[28] Aparece cuando se miden las dos componentes de la velocidad bidimensional usando sólo medidas locales.

tienen un uso extendido son la Correlación Cruzada Normalizada [60, 61] (*Normalized Cross Correlation*, NCC) o la Clasificación por Diferencia de Píxel [62] (*Pixel Difference Classification*, PDC).

Uno de estos criterios de semejanza determina la elección del bloque con mayor similitud (o que minimiza un error medido) de entre los candidatos dentro de la ventana de búsqueda de tamaño fijo del fotograma de referencia. Si el bloque elegido no se encuentra en la misma posición en ambos fotogramas, significa que se ha movido. La distancia del bloque coincidente entre el fotograma actual y el de referencia se define como el vector de desplazamiento estimado, y será el que se le asigne a todos los píxeles del macrobloque.

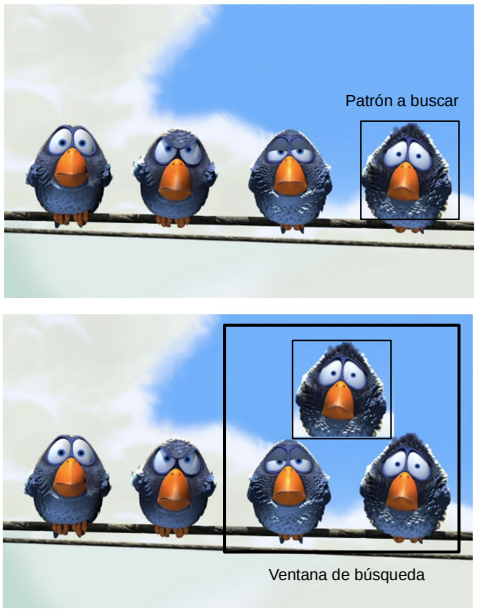


Figura 8: Ejemplo de estimación de movimiento con Block Matching.

En el caso ideal, los píxeles correspondientes de los bloques coincidentes serían exactamente iguales. No obstante, ese caso sucede en muy raras ocasiones, ya que la forma de los objetos en movimiento varía con respecto al punto de vista del observador o la luz reflejada sobre su superficie, y siempre nos veremos afectados por el ruido.

Los algoritmos de *Block Matching* se caracterizan por una búsqueda exhaustiva y un funcionamiento iterativo, lo que los hace ser unos algoritmos con una ejecución muy lenta, requiriendo una cantidad de recursos prohibitiva en muchos casos. Si la imagen es de tamaño M^2, la plantilla de búsqueda es de tamaño N^2, y la ventana de búsqueda es de tamaño L^2 entonces la estimación total de complejidad de cómputo requerida sería del orden de $M^2N^2L^2$.

La búsqueda exhaustiva (FST - *Full Search Technique*) no es el único algoritmo perteneciente a la familia de modelos de emparejamiento. La tabla 2 resume algunos algoritmos que siguen la filosofía de *Block Matching* y su complejidad computacional asociada.

Algoritmo	**Número máximo de puntos de búsqueda**	**Complejidad computacional**
FST	$(2w+1)^2$	$O(N^2)$
TSST [63]	$1+8log_2(w+1)$	$O(log_2 N)$
LOGST [58]	$2+7log_2 w$	$O(log_2 N)$
DS [64]	$9+5log_2 w$	$O(log_2 N)$
CSA [65]	$5+4log_2 w$	$O(log_2 N)$
4SST [66]	$18log_2((w+1)/4)+9$	$O(log_2 N)$

Tabla 2: Algoritmos de la familia de modelos de emparejamiento y su complejidad computacional donde w es el tamaño del desplazamiento.

La figura 9 muestra un ejemplo del algoritmo de búsqueda en 3 pasos (TSST - *Three-Step Search Technique*) donde el primer paso es asignar el tamaño de la ventana de búsqueda, que será la mitad del área de búsqueda. Se seleccionan en cada paso nueve puntos candidatos, incluyendo el punto central y los ocho puntos de comprobación en el límite de la ventana de búsqueda. El segundo paso es mover el centro de búsqueda hacia adelante hasta el punto coincidente con el mínimo valor SAD de la etapa anterior, reduciendo a la mitad el tamaño de la ventana de búsqueda. Por último, en el tercer paso se detiene el proceso de búsqueda. Se reduce de nuevo el paso (hasta valer 1 *pixel*) y se repite el proceso de nuevo.

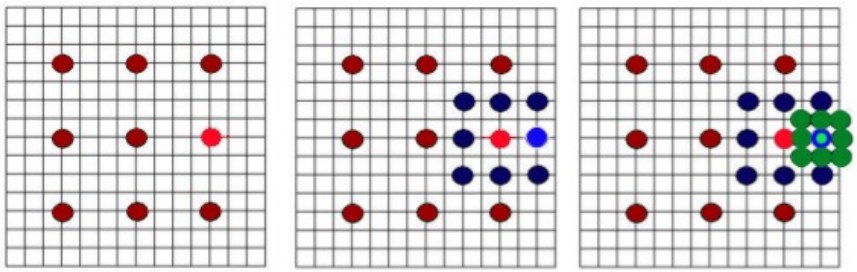


Figura 9: Explicación de los tres pasos del algoritmo TSST.

La búsqueda logarítmica en 2-D (LOGST - *2-D Logarithm Search Technique*) utiliza un patrón de búsqueda transversal en cada etapa en forma de "+", con un tamaño del paso inicial de $d/4$. El tamaño del paso se reduce cuando el punto mínimo de la etapa anterior es el del centro o cuando el punto mínimo actual alcanza el límite de la ventana de búsqueda. Si ninguna de estas dos condiciones se alcanzan, el tamaño del paso sigue siendo el mismo.

Por su parte, el algoritmo de búsqueda en diamante (DS - *Diamond Search*) en cada paso trata de encontrar el punto con menor valor SAD entre sus cuatro puntos adyacentes, siendo

este el centro en el siguiente paso. El criterio que utiliza para parar la búsqueda es almacenar los valores mínimos de SAD de los dos últimos pasos y si el actual es mayor que los dos anteriores, entonces el algoritmo termina.

El algoritmo de búsqueda cruzada (CSA - *Cross-Search Algorithm*) es similar al de la búsqueda logarítmica. En la tabla 2, el término *w* denota el mayor desplazamiento permitido. Los pasos que sigue este algoritmo son los siguientes:

1. El bloque central se compara con el bloque actual y si el valor de SAD es menor que un cierto umbral, se considera que el algoritmo ha alcanzado la condición de parada.

2. Se escoge el primer conjunto de puntos en forma de 'X' alrededor del centro y se mueve el centro al punto con el mínimo SAD. Normalmente el tamaño del paso se escoge como la mitad del desplazamiento máximo.

3. Si el tamaño del paso fuese mayor que 1, se reduce este a la mitad y se vuelve al paso 2. En caso contrario, se va al cuarto paso.

4. Si en la última etapa el punto de mínimo SAD se encuentra abajo a la izquierda o arriba a la derecha, se evalúan los 4 puntos alrededor de este que forman una '+'. Si por el contrario, el punto con mínimo SAD es el de arriba a la izquierda o el de abajo a la derecha, se evalúa el SAD para los 4 puntos alrededor de este que forman una 'X'.

Por último, la búsqueda en cuatro pasos (4SST - *Four-Step Search Technique*) comienza con una comparación de nueve puntos siguiendo los cuatro pasos que se detallan a continuación:

1. Se empieza con un tamaño de paso de 2. Se escogen los nueve puntos alrededor del centro de la ventana de búsqueda y se encuentra el punto con menor SAD. Si resulta que el punto con menor SAD es el del centro, se considera que el algoritmo converge y procesa directamente el cuarto paso. En caso contrario, se continua con el segundo.

2. Se mueve el centro al punto con menor SAD y se mantiene el tamaño de paso a 2. El patrón de búsqueda depende de la posición donde estuviera el punto con el mínimo SAD en el paso anterior (figura 10):
   - Si el punto previo con mínimo SAD está localizado en una esquina del área de búsqueda, se escogen 5 puntos para el patrón de búsqueda.
   - Si por el contrario, el punto está en el eje horizontal o vertical, se seleccionan únicamente 3 puntos.

   Una vez localizado el punto con menor SAD, si se encuentra este en el centro se va al paso cuarto, en otro caso, se continua con el tercer paso.

3. La estrategia para el patrón de búsqueda es la misma, sin embargo esta acaba en el cuarto paso.

4. El tamaño del paso se reduce a 1 y se vuelven a examinar los nueve puntos alrededor del centro.

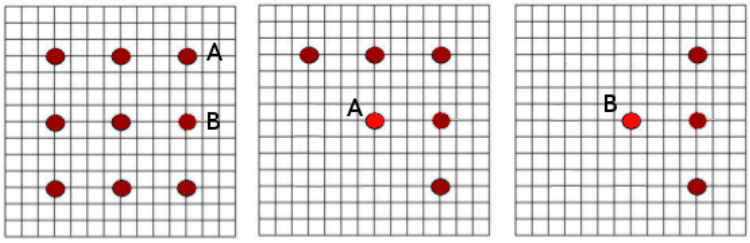


Figura 10: Diferentes casos de selección de bloque en el algoritmo 4SST.

Existen algunos estudios que defienden que los algoritmos de la familia *Block Matching* no son evidentes desde un punto de vista biológico [54] ya que no son capaces de hacer predicciones sobre estímulos complejos (por ejemplo barras verticales posicionadas aleatoriamente), para los cuales, los observadores experimentales perciben diversos movimientos en diferentes posiciones.

Estos algoritmos poseen como principal bondad su simplicidad, lo que ha motivado su estudio durante muchos años por lo que en la actualidad dominan en entornos industriales para control de calidad e inspección. Otra de sus ventajas es su capacidad de trabajo en entornos donde los desplazamientos entre fotogramas son largos (más extensos que unos pocos puntos), aunque esto requiere el procesamiento de largos espacios de búsqueda. Por estas razones se han convertido en una técnica fundamental en la mayoría de los estándares de compresión y codificación de vídeo basados en la compensación de movimiento [67, 68].

2.2.2. Métodos de energía.

Los métodos que pertenecen a esta familia hacen uso de filtros de orientación espacio-temporal para poder reaccionar de forma óptima a velocidades específicas de la imagen. Los bancos de filtros de este tipo se utilizan para responder a una amplia gama de movimientos visuales [5].

Esta familia aborda el problema de la estimación de movimiento como un problema de estimación global bayesiano. Para ello, se establece un criterio de **máximo a posteriori** que permita maximizar la probabilidad del campo de movimiento, dada la observación de la intensidad de la imagen en la próxima trama. La estimación del máximo a posteriori puede lograrse por medio de métodos estocásticos de relajación, tales como el temple simulado, que garanticen la convergencia hacia un máximo global.

Se usan filtros orientados espacio-temporales que responden de forma óptima a determinadas velocidades. Las estructuras utilizadas para este tipo de procesamiento son bancos de filtros en paralelo que se activan para un rango determinado de valores.

El gran inconveniente de éste tipo de técnicas es su lentitud de procesado, ya que se pretende segmentar el campo de movimiento en regiones de movimiento coherente trabajando *pixel* a

pixel, y que a su vez cada *pixel* tenga asociada una distribución de velocidades en lugar de una velocidad específica.

Este tipo de algoritmos requieren dos funciones de densidad de probabilidad:

- La probabilidad condicional de la intensidad de imagen observada dado el campo de movimiento, también llamado modelo *likelihood* o modelo de observaciones.
- La probabilidad a priori de los vectores de movimiento o modelo de campo de movimiento.

Sea la intensidad de campo en un *pixel* x y el *frame* k, $s_k(x)$ donde $d(x) = (d_1(x), d_2(x))$ denota el vector desplazamiento. En general, cuando se obtiene vídeo, este está corrompido levemente por la incorporación de ruido de la forma $g_k(x) = s_k(x) + v_k(x)$. En general, el máximo a posteriori para dos *frames* consecutivos $g_k(x)$ y $g_{k-1}(x)$ viene descrita por la ecuación (3).

$$(d_1, d_2) = \arg max_{d_1 d_2} p\left(d_1 d_2 | g_k g_{k-1}\right) \tag{3}$$

El diseño de filtros orientados espacio-temporales normalmente se realiza en el dominio de la frecuencia. Si examinamos la relación entre un patrón bidimensional que se traslada y su transformada de Fourier (4), encontramos que $\hat{I}_0(\bar{k})$ es la transformada de Fourier del patrón estático, $\bar{x} = (x, y)$ es la posición de la imagen, $\bar{k} = (k_1, k_2)$ es la frecuencia espacial, t es el tiempo, ω es la frecuencia temporal y $\bar{v} = (u, v)$ la velocidad de la imagen.

$$\begin{aligned} I\left(\bar{x}, t\right) &= I_0\left(\bar{x} - \bar{v}t\right) \\ \hat{I}(\bar{k}, \omega) &= \hat{I}_0(\bar{k}) \cdot \delta(\omega + v \cdot k) \end{aligned} \tag{4}$$

La presencia de la función delta de Dirac [69] $\delta(\omega + v \cdot k)$ obliga a encontrarse en un plano alrededor del origen a todos los valores distintos de cero. Esto demuestra que es posible diseñar un algoritmo de movimiento en el plano de Fourier haciendo uso de bancos de filtros sintonizados capaces de responder de forma correcta a planos específicos en el espacio de frecuencias.

Una de las principales diferencias entre algoritmos de esta familia se encuentra en el criterio a la hora de diseñar los bancos de filtros. La figura 11 sirve de ejemplo simple, en la que se puede observar un borde unidimensional que se traslada y su transformada de Fourier asociada. Para este caso, las componentes de Fourier están restringidas a una línea en el espacio de frecuencias.

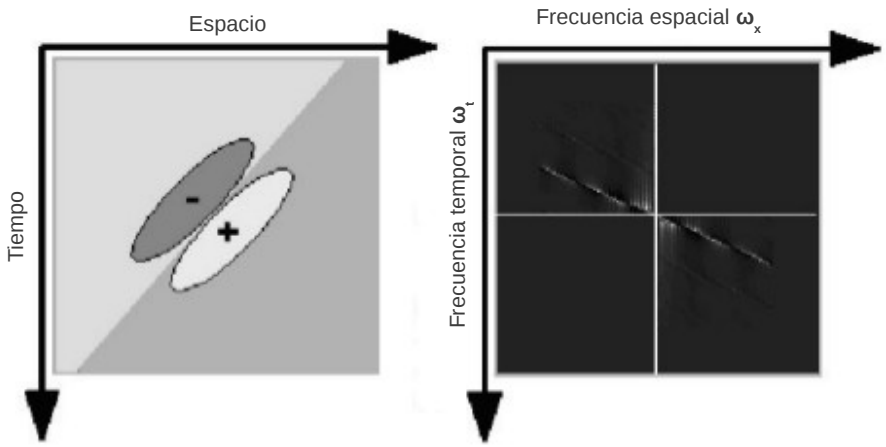


Figura 11: Filtro espacio-temporal orientado que responde a un borde en movimiento.

Adelson y Bergen [54] expusieron un sistema por el cual se puede estimar el flujo mediante la energía de movimiento, sintetizando a partir de unos pocos filtros separables un filtro espacio-temporal orientado. La explicación de cómo es posible usar una combinación de salidas de filtros para estimar la velocidad se muestra en la figura 12, en la que se dispone de cuatro unidades de procesamiento de entrada y cuyas respuestas son A, B, A', B'. Estos filtros corresponden con derivadas Gaussianas espacio-temporales. La siguiente etapa consiste en sumar o restar siempre en cuadratura[29] (con fase de 90°) para formar filtros direccionales en parejas, que sean sensibles a movimiento hacia la izquierda y la derecha y cuyas salidas son I1,I2,D1,D2. A continuación, se calcula la energía de movimiento que corresponde a cada dirección sumando los cuadrados de las respuestas de las diferencias de los filtros, imitando así el comportamiento de las células complejas [70]. La etapa siguiente, llamada también etapa de oponencia, toma la diferencia entre dos energías, siendo la energía de oponencia $Eo = E$d-Ei = 4(A'B-AB'). Con ello se resalta, de manera específica, una dirección y además anula la respuesta a imágenes estacionarias. Cabe destacar que la energía de oponencia no estima velocidad, en parte porque varía mucho con el contraste.

[29] Propiedad por la que dos filtros forman un par, donde la mínima respuesta de un filtro coincide con la máxima respuesta en el otro.

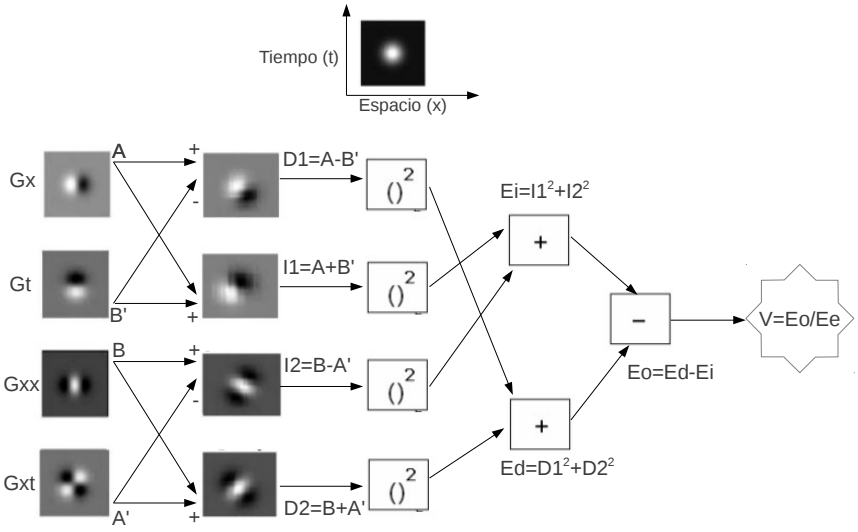


Figura 12: Modelo de energía de movimiento.

La situación ideal sería que la relación entre energía estática y energía de movimiento (Eo/Ee) fuera igual a la velocidad verdadera V para todos los patrones posibles de movimiento. La energía estática vendría expresada por la ecuación 5, donde A y B son los filtros espaciales no direccionales que se han mostrado anteriormente (misma respuesta temporal pero desfasados 90º).

$$Ee = 4(A^2 + B^2) \Rightarrow Eo/Ee = (A'B - AB')/(A^2 + B^2) \tag{5}$$

Los modelos basados en frecuencia o de energía de movimiento y los modelos de gradiente son similares en muchos aspectos, ya que ambos sistemas usan bancos de filtros para obtener el movimiento, es decir, la orientación espacio-temporal. La principal diferencia se encuentra en que los filtros usados en modelos de energía están diseñados para responder a orientaciones específicas espacio-temporales más que para un cociente de filtros.

2.2.3. Métodos diferenciales o de gradiente.

La metodología de los modelos de gradiente se basa en utilizar derivadas de la intensidad de la imagen en el espacio y el tiempo. Las combinaciones y proporciones de estas derivadas producen medidas explícitas de velocidad [6, 71].

El enfoque basado en el gradiente es un enfoque diferencial que recurre a estimaciones de las variaciones espacio-temporales de las intensidades de los píxeles de las imágenes para obtener un vector de velocidad instantánea que permita determinar el campo de flujo óptico. Estos métodos se han convertido en el principal acercamiento en las aplicaciones de visión por computador debido a que su computación es eficaz, en el sentido de que produce una buena estimación del flujo óptico.

Las técnicas y algoritmos que pertenecen a esta familia actúan usando derivadas de la intensidad de la imagen en el espacio y en el tiempo. La velocidad se obtiene a partir de cocientes de las medidas anteriores.

Los métodos de energía y de emparejamiento, que básicamente funcionan aplicando plantillas, tienen como fin obtener mecanismos que respondan bien al movimiento de un estímulo respecto a una dirección y con una velocidad específica, dando como resultado, y de forma ideal, un ajuste entre el movimiento y la plantilla.

Sin tener en cuenta otras dificultades de menor calado, se debe señalar que la respuesta de estos dos métodos depende fuertemente del contraste del estímulo, necesitando etapas de normalización complejas que no evitan completamente este efecto aunque lo atenúan [72]. Además, se debe calcular la velocidad en posteriores etapas, habitualmente mediante poblaciones de filtros, ya que las plantillas no proporcionan directamente el movimiento, como se ha explicado en este capítulo.

Por contra, los métodos diferenciales o de gradiente tienen una filosofía dispar puesto que, mediante un cociente de derivadas espaciales y temporales en cada punto, calculan directamente la velocidad. Además, esta filosofía viene dada por una solución a la ecuación de constricción del flujo óptico (1). Como el contraste varía de la misma forma en el numerador y en el denominador, el resultado no debería verse afectado por éste, obteniendo como resultado un sistema que no varía con el contraste y que además no conlleva un sobrecoste adicional.

Hay que destacar esta cualidad a pesar de que deje de cumplirse en zonas de patrones con frecuencias temporales y espaciales bajas, en el que no hay información suficiente para disparar un cierto umbral de activación. En 1995, Johnston [72] realizó un estudio de la velocidad estimada en función de dicho umbral para una población de filtros derivativos. Además, al problema descrito anteriormente hay que añadirle otro, que aparece con la dependencia del patrón estático frente al dinámico en las tres familias de modelos.

Como consecuencia de todo esto, se necesita desarrollar otros modelos más robustos y elaborados frente a estas consideraciones que solucionan esta dependencia del patrón estático de tal forma que introducen una etapa de filtrado pasa banda previa al procesamiento. Ejemplos de estos modelos son versiones previas al McGM [73] o el de Lucas y Kanade [71]. Por contraposición, no todo son ventajas, ya que este uso añade nuevas contrapartidas no deseadas:

- Se necesitan realizar operaciones de inversión de matrices grandes (Lucas y Kanade [71]), que no son justificables desde el punto de vista biológico. Por este motivo se ha decidido adoptar en este trabajo un esquema que se basa en operaciones que sólo utilizan respuestas de filtros orientados, que son calculadas mediante operaciones de multiplicación, suma y división.
- Se elimina información potencialmente útil al suprimir las frecuencias inferiores, que provoca que se reduzca la señal en patrones con un movimiento lento asociado.

- Si se tiene en cuenta que el umbral inferior de movimiento en la fóvea humana es $0{,}02^o/seg.$ [74], (lo que atañe a una orientación en el espacio-tiempo de 1^o, si se escala para que $1^o/seg.$ equivalga a 45^o de orientación), es difícil justificar un filtro temporal plausible biológicamente que sea capaz de eliminar la influencia del patrón estático, al mismo tiempo que puede mantener esta alta sensibilidad a movimientos lentos. Esto indica que en sistemas biológicos se elude la influencia del patrón estático mediante mecanismos especiales del córtex en etapas posteriores, en lugar de operaciones de filtrado en etapas previas. Existe una consistencia fisiológica y neuronal a nivel de primates [75], que deduce que una lesión en el área cerebral de movimiento (V5-MT) aumenta la sensibilidad al ruido dinámico y estático con tan sólo un pequeño efecto en el umbral de detección de filtros de bajo nivel [76, 77]. Este tipo de modelos proporciona una justificación convincente para la sensibilidad al patrón estático que muestran pacientes que presentan una lesión de la clase de ceguera al movimiento.

Así pues, la eliminación del patrón estático haciendo uso del filtrado temporal no es factible desde un punto de vista biológico, obteniendo como conclusión que si se desea un sistema biológicamente apropiado, es esencial que se adopte una estrategia en la cual la etapa de independencia al patrón estático se ubique al final del mismo.

2.3. Implementaciones existentes en aceleradores.

Las técnicas para calcular flujo óptico a menudo buscan un compromiso entre la precisión y la eficiencia [78]. Esta compensación surge porque las técnicas más precisas tienden a tener mayores requerimientos computacionales. Teniendo en cuenta similares recursos computacionales, algunas técnicas hacen estimaciones de movimiento más lentas aunque más precisas mientras que otras hacen cálculos más rápidos pero menos precisos. La precisión permite evaluar la calidad de los resultados obtenidos mientras que con eficiencia nos referimos tanto al tiempo en el que se obtienen los datos como a los recursos computacionales utilizados para ello.

Dentro de la estimación de movimiento, y más concretamente en el marco de este trabajo, es interesante obtener resultados precisos y de manera eficiente en tiempo real, haciendo uso de técnicas que permitan su adaptación a problemas reales. Con este objetivo, vamos a hacer un repaso de los estudios realizados en los últimos años relacionados con la estimación de movimiento en aceleradores, tanto aquellos trabajos que hacen uso de *hardware* gráfico como de circuitos específicos o de FPGAs.

2.3.1. FPGAs.

Uno de los tipos de aceleradores que se utilizan para la estimación de movimiento son las FPGAs. Es por ello por lo que en la literatura se encuentran múltiples trabajos que relacionan ambos conceptos. La tabla 3 expone algunos de los realizados en los últimos años, mostrando además a qué familia pertenecen los algoritmos implementados, el rendimiento obtenido y el tipo de FPGA utilizado.

Autor	Familia	Algoritmo	fps (resolución)	FPGA	Año
Niitsuma [79]	*Block Matching*	*FST*	30 (640×480)	Virtex-II XC2V6000 [80]	2004
Babionitakis [81]	*Block Matching*	varios	30 (1024×768)	Virtex-II Pro 40	2008
Asano [82]	*Block Matching*	*FST*	30 (1920×1080)	Virtex-5 [83]	2010
Akin [84]	*Block Matching*	*OBT* [85]	83 (1920×1080)	-	2010
Ho [86]	*Block Matching*	*MFHME* [87]	30 (1920×1080)	-	2011
Núñez-Yáñez [88]	*Block Matching*	*FME* [89]	62 (1920×1080)	Virtex-5	2012
Wei [90]	*Gradiente*	*Tensores* [91, 92]	64 (640×480)	Virtex-II Pro XC2VP30 [80]	2007
Díaz [93]	*Gradiente*	*Lucas-Kanade*	170 (800×600)	Virtex-II XC2V6000-4	2008
Botella [94]	*Gradiente*	*McGM*	177 (128×96)	Virtex-E 2000eBG560 [95]	2010
Bahar [96]	*Gradiente*	*Horn&Schunck*	1029.9 (240×320)	Cyclone II [97]	2012
Barranco [98]	*Gradiente*	*Lucas-Kanade*	270 (640×480)	Virtex-4 XC4vfx100 [99]	2012
Norouznezhad [100]	*Energía*	*Gabor filter* [101]	30 (40×480)	Xilinx Virtex-5 XC5VSX50T	2010
Tomasi [102]	*Energía*	*Phase-Based* [103]	36 (512×512)	Virtex-4 XC4vfx100	2010
Tomasi [104]	*Energía*	*Phase-Based* [105]	22.8 (512×512)	Virtex-4 XC4vfx100	2012

Tabla 3: Estado del arte para algoritmos de estimación de movimiento en FPGAs.

Niitsuma y Maruyama [79] implementan el algoritmo de búsqueda exhaustiva en una FPGA con el objetivo de calcular el movimiento de objetos y la distancia a la que estos se encuentran combinando el flujo óptico con la visión en estéreo. El rendimiento obtenido por este sistema es de 30 *frames* por segundo (fps) para una resolución, o tamaño de fotograma, de 640×480. Otro trabajo relacionado con la estimación de movimiento mediante *Block Matching* haciendo uso de FPGAs es el expuesto por Babionitakis *et al.* en [81], donde se propone una arquitectura que soporta de forma eficiente un conjunto de algoritmos de *Block Matching* que se utilizan en la actualidad. Además, el diseño propuesto ejecuta los diferentes algoritmos proporcionando un conjunto de instrucciones comunes a todos estos algoritmos y sólo unas pocas instrucciones específicas a cada uno de ellos. Los resultados que obtienen en una FPGA son de 30 fotogramas por segundo con un tamaño de fotograma de 1024×768.

En la estimación de movimiento mediante *Block Matching*, el fotograma actual se divide en macrobloques, y se busca la mejor correspondencia de bloque para cada macrobloque en la zona de búsqueda del fotograma de referencia. Sin embargo, Asano *et al.* [82] proponen un método por el cual la dirección de exploración del macrobloque en el fotograma actual, y la dirección de exploración de coincidencia en el área de búsqueda, se optimicen con el fin de reducir el acceso a los bancos de memoria externa que almacenan el marco de referencia, y a los cacheados en la memoria interna que representan la zona de búsqueda. Mediante la reducción de ambos accesos a memoria, es posible obtener un alto rendimiento en una FPGA de tamaño pequeño consiguiendo obtener 30 fps para una resolución de 1920×1080, consiguiendo, además, reducir el número de bancos de memoria requeridos para procesar estos datos en tiempo real. Para ello, muestran una aproximación para la estimación de movimiento mediante una búsqueda completa con tamaño de bloque variable (VSBME) en dicho dispositivo.

Teniendo en cuenta las implementaciones en FPGA de modelos de emparejamiento podemos hablar también del trabajo realizado por Akin *et al.* [84], referente a arquitecturas *hardware* de alto rendimiento para algoritmos de estimación basados en la transformada de un bit (1BT), donde se exponen varios modelos de arquitecturas que consiguen obtener resultados de forma más rápida y haciendo uso de menos memoria. Además, proponen por primera vez

un *hardware* reconfigurable para la estimación de movimiento con fotograma de referencia múltiple basado en 1BT mediante el cual el número y la selección de fotogramas de referencia se puede configurar de manera estática basándose en los requisitos de la aplicación, con el fin de estimar el rendimiento y la complejidad computacional de la estimación de movimiento. El rendimiento alcanzado en este trabajo es de 83 fps (con una resolución igual a 1920×1080). Por su parte, Ho *et al.* [86] presentan una implementación en FPGA del algoritmo MFHME (*multi-frame hierarchical motion estimation*) con la que se consiguen alcanzar los 62 fps para conversiones de vídeo de alta definición (1920×1080). El sistema diseñado es escalable y obtiene alto rendimiento para vídeo en tiempo real con una alta precisión.

Para terminar con los algoritmos referentes a los modelos de emparejamiento, Núñez-Yáñez *et al.* [88] presentan una implementación flexible y escalable para la estimación de movimiento que es capaz de soportar los requerimientos para procesar vídeo en alta definición haciendo uso del *codec* H.264 AVC[30]. Para ello, optimizan el algoritmo de *fast block matching* obteniendo un rendimiento de 62 fps para resoluciones de 1920×1080.

Con respecto a los trabajos relacionados con los modelos de gradiente, merece la pena destacar el realizado por Wei *et al.* [90], en el que se explica como se ha implementado y modificado un algoritmo de flujo óptico basado en tensores para adaptarlo a una FPGA con el objetivo de evitar pequeños obstáculos y posibilitar la navegación de vehículos no tripulados. El rendimiento obtenido es de 64 fps, para un tamaño de fotogramas de 640×480, y con un error angular medio de 12,9º.

Con respecto al algoritmo de Lucas & Kanade, haciendo uso de FPGAs, el trabajo llevado a cabo por Díaz *et al.* [93] consiste en la implementación del algoritmo de Lucas & Kanade en FPGA con la capacidad de procesar hasta 170 fps a una resolución de 800×600 *pixels* en una arquitectura escalable, modular y versátil. Por otro lado, Barranco *et al.* [98] proponen una arquitectura paralela para la estimación de movimiento que es capaz de alcanzar 270 fps para una resolución de 640×480 en el mejor de los casos en una implementación mono-escala [106] y 32 fps para una implementación multi-escala [107, 108], obteniendo un error angular medio de 4,55º en el mejor de los casos.

Con respecto a la implementación del modelo McGM, Botella *et al.* exponen en su trabajo [94] una arquitectura personalizable basada en *hardware* reconfigurable con las propiedades de la secuencia del movimiento cortical. El rendimiento obtenido es de 177 fps con una resolución igual a 128×96 y con un error angular medio de 7,2º. El enfoque mostrado en este trabajo es apropiado en situaciones en las que la luminosidad es muy variable, en entornos ruidosos o en la investigación sobre el sistema perceptivo de los seres humanos.

Otro algoritmo implementado en FPGAs, perteneciente a los modelos de gradiente, se presenta en el trabajo realizado por Bahar y Karimian [96], donde exponen la utilización de la FPGA Cyclone II para acelerar los cálculos relativos al algoritmo de flujo óptico de Horn

[30] Estándar para la compresión de vídeo: `http://www.itu.int/rec/T-REC-H.264`

& Schunck. La implementación que proponen utiliza unidades paralelas de almacenamiento para facilitar el acceso a memoria y los cálculos, mejorando la eficiencia del algoritmo y reduciendo significativamente de esta forma los ciclos de reloj necesarios, llegando a ser de tan sólo un ciclo. De esta manera llegan a alcanzar un rendimiento máximo de 1029,9 fps para una resolución de 240×320, adecuado para la detección de movimiento en tiempo real, con un error angular medio de 6,96º.

Para terminar con esta subsección, se presentan algunos de los trabajos realizados sobre FPGAs en los que se implementan modelos de energía. En [100] se presenta un sistema, implementado en FPGA, para el seguimiento de objetos haciendo uso de filtros de Gabor a través de varios canales de la imagen de entrada. El sistema es capaz de obtener un rendimiento de 30 fps para una resolución de entrada de 640×480.

Otro ejemplo es el expuesto por Tomasi *et al.* [102, 104] que, por partida doble, presentan dos trabajos en los que se implementan algoritmos basados en fase en FPGA. En [102] se presenta una arquitectura en la que se implementa un modelo basado en fase multi-escala que es capaz de procesar imágenes con una resolución de 512×512 de manera precisa a 36 fps. El algoritmo calcula la fase para ocho orientaciones diferentes en un modelo piramidal y propaga la información a través de un número adecuado de escalas en función del tamaño de la imagen de entrada y el tamaño del filtro. En [104] se presenta una arquitectura para la extracción de características de visión de bajo nivel que incluyen el flujo óptico multi-escala, la disparidad, la energía, la orientación y la fase. De las soluciones que se presentan, la de alto rendimiento es capaz de computar 165 GOPS[31] con un consumo de energía de 30 GOPS/W con una frecuencia de reloj de 50 MHz. El sistema consigue un rendimiento de 22,8 fps con fotogramas de tamaño 512×512.

2.3.2. GPUs.

El *hardware* gráfico es un acelerador en pleno auge, tal y como se explicó en el capítulo 1. Debido a que hoy en día están disponibles en dispositivos de bajo coste, su aplicación a técnicas de estimación de movimiento con el objetivo de obtener resultados de manera más rápida, buscando obtener estos en tiempo real, ha sido, y es, una fuente de estudio a tener en cuenta. Debido a que esta tesis sustenta sus bases en este tipo de aceleradores, a continuación la tabla 4 muestra una serie de trabajos relacionados con la aplicación de GPUs para estimar el movimiento. La mayoría de los trabajos expuestos utilizan una arquitectura CUDA, salvo en los realizados por Jinglin Zhang *et al.* [109], en el que la arquitectura seleccionada ha sido OpenCL, Duvenhage *et al.* [110], que se decantan por OpenGL, y el de Pauwels y Van Hulle [111], que no especifica arquitectura alguna.

[31] *Giga Operations Per Second* - 10^9 operaciones por segundo.

Autor	Familia	Algoritmo	fps (resolución)	GPU	Año
Kiss [112]	*Block Matching*	*Crosby* [113]	33 (200×200)	GTX 285 [114]	2009
Zhang [115]	*Block Matching*	*BNM* [116]	15 (1280×768)	Tesla C1060 [117]	2010
Monteiro [118]	*Block Matching*	*FST*	256 (1280×768)	GTX 480 [119]	2011
Ranft [120]	*Block Matching*	*Raw Block Matching* [121]	N/D	GTX 470 [122]	2011
Zhang [109]	*Block Matching*	*FST*	89 (1280×720)	Radeon HD 6870 [123]	2012
Rodríguez-Sánchez [124]	*Block Matching*	-	6 (1920×1080)	GTX 480	2012
Monteiro [125]	*Block Matching*	*FST* *DS*	33 (1280×720) 310 (1280×720)	GTX 480	2012
Vu [126]	*Block Matching*	*HS* [127]	16 (1280×720)	Tesla C2050 [128]	2012
Chase [129]	*Gradiente*	*Tensor* [90]	150 (640×480)	8800 GTX [130]	2008
Marzat [131]	*Gradiente*	*Lucas-Kanade*	47 (316×252)	Tesla C870 [117]	2009
Duvenhage [110]	*Gradiente*	*Lucas-Kanade*	30 (512×512)	GTX285 [114]	2010
Phull [132]	*Gradiente*	*RLCT* [133]	465 (1024×768)	280 GTX [134]	2010
del Riego [135]	*Gradiente*	*HLK* [107]	30 (640×480)	9500 GT [136]	2011
Hegner [137]	*Gradiente*	*Pyramid Algorithm* [138]	36 (512×512)	GTX 260 [139]	2011
Ohmura [140]	*Gradiente*	*Lucas-Kanade*	40 (512×384)	Tesla C1060 [117]	2011
Shiralkar [141]	*Gradiente*	*SOM* [142]	20 (640×480)	GTX480 [119]	2012
Pauwels [111]	*Energía*	*Phase-Based* [103]	128 (316×252)	8800 GTX [130]	2008
Sundaram [143]	*Energía*	*LDOF* [144]	1.84 (640×480)	GTX 480 [119]	2010
Gwosdek [145]	*Energía*	*Euler-Lagrange*	N/D	GTX 480 [119]	2012
Abramov [146]	*Energía*	*Phase-Based* [147]	4.3 (640×512)	GT 240M [148]	2012

Tabla 4: Resumen de trabajos relacionados con la estimación de movimiento en GPUs. N/D: no definido.

Kiss *et al.* exponen en [112] una manera de optimizar el rendimiento en técnicas de *Block Matching* para eco-cardiografías 3D haciendo uso de un modelo SIMD. Para ello comparan una implementación del algoritmo desarrollado en [113] haciendo uso de instrucciones SSE con una implementación en GPU utilizando CUDA obteniendo un rendimiento 26 veces superior (26×) la segunda implementación con respecto a la primera. En ambos casos el tiempo en el que se ejecuta el algoritmo se ve reducido, lo que supone una mejora al aplicarlo en entornos clínicos.

En el trabajo realizado por Zhang *et al.* [115] se presenta una implementación en GPU del algoritmo *Best Neighborhood Matching* [116] (BNM), un método efectivo para recuperar imágenes dañadas por una mala transmisión en la red, y se aplica a imágenes con color de alta definición (con resoluciones mayores que 1280×768). El rendimiento obtenido es de 21× con respecto a una versión secuencial del mismo algoritmo ejecutada en una CPU.

El estudio llevado a cabo por Ranft *et al.* [120] se centra en tres plataformas *hardware* diferentes para paralelizar estimaciones basadas en emparejamiento, una *multicore* con dos núcleos, una con GPU y otra con un núcleo ×64, llegando a la conclusión de que la GPU es la más rápida de las tres (entre 1.3× y 4.3× más rápida).

Con respecto al algoritmo *FST*, Monteiro *et al.* [118] presentan una implementación de dicho algoritmo haciendo uso de CUDA con el objetivo de obtener un alto rendimiento a la hora de estimar movimiento en la codificación de vídeo. En este trabajo se estudia el rendimiento obtenido comparándolo con una implementación que hace uso de OpenMP[32] y otra secuencial ejecutada en la CPU, obteniendo un rendimiento de 66× y 600× respectivamente. Además,

[32] `http://openmp.org/wp/`

en [125] el trabajo se centra en presentar un método eficiente para mapear algoritmos de estimación de movimiento en GPU utilizando el modelo de programación de CUDA. Los algoritmos escogidos para realizar el estudio han sido el *FST* y el *DS*. Para comparar los resultados obtenidos, implementan también una versión de los algoritmos usando OpenMP y MPI[33] con la finalidad de tener una versión *multicore* y otra distribuida, respectivamente. La solución proporcionada haciendo uso de CUDA consigue obtener mayor rendimiento que las versiones de OpenMP y MPI. Así, para el algoritmo *FST*, el rendimiento obtenido con respecto a una versión secuencial es de 154× para la implementación CUDA, 13× para la MPI y 1× para OpenMP. Además, si se compara la versión de CUDA con las versiones de MPI y OpenMP se obtienen unos *speedups* de 14× y 77× respectivamente. Con respecto al algoritmo *DS* se obtienen unos *speedups* de 62×, 1× y 0.5× para las versiones CUDA, MPI y OpenMP respectivamente. Además, la versión CUDA alcanza un *speedup* de 77× con respecto a la versión MPI y de 191× con respecto a la versión OpenMP.

Jinglin Zhang, Nezan y Cousin introducen en [109] una implementación para la estimación de movimiento basada en computación paralela heterogénea. Haciendo uso de OpenCL[34] (*Open Computing Language*), proponen un método para determinar la distribución de carga de trabajo en un sistema heterogéneo que cuenta con una CPU y una GPU, consiguiendo *speedups* entre ×100 y ×150 comparado con la misma implementación en una sola CPU. Para las pruebas, comparan implementaciones de OpenCL en CPU, en una tarjeta NVIDIA y en otra ATI. En esta última es donde obtienen los mejores resultados con 89 fps en resoluciones de 1280×720. Además, cabe destacar que el criterio para seleccionar el mejor vector de movimiento se realiza mediante el algoritmo SAD.

En [124], Rodríguez-Sánchez *et al.* muestran un método para la estimación de movimiento en la codificación de vídeo 3D de alta definición haciendo uso de GPU, que permite reducir el tiempo de ejecución hasta en un 98 % además de ser energéticamente más eficiente y requerir menor energía que el método secuencial. Dicho método es una extensión del *codec* H.264/AVC para soportar vídeo 3D, *Multiview Video Coding* [149] (MVC). Los resultados obtenidos en este trabajo muestran un *speedup* de 67× para el algoritmo de estimación de movimiento propuesto y de 10× para la codificación de vídeo completa.

Para finalizar con los algoritmos de *Block Matching* en GPU, el trabajo realizado por Vu, Yang y Bhuyan [126] presenta una aproximación eficiente y dinámica basada en GPU para el algoritmo *Hierarchical Search* [127]. Para ello, implementan un esquema de selección fijo y otro dinámico con el que obtienen unos *speedup* iguales a 200× y 250× respectivamente.

Con respecto a los estudios realizados sobre GPU referentes a algoritmos de la familia de gradiente, Chase *et al.* [129] presenta una comparación entre la utilización de una FPGA (*Xilin Virtex-II Pro XC2VP30*) y una GPU (*NVIDIA GeForce 8800 GTX*) para el cálculo del flujo óptico, llegando a la conclusión de que la implementación realizada en la FPGA

[33] `http://www.mcs.anl.gov/research/projects/mpi/`
[34] `http://www.khronos.org/opencl/`

requiere mayor tiempo de desarrollo (del orden de 12×). Para ello, implementan una versión del algoritmo descrito en [90] en ambas plataformas. Además, con la realizada en GPU, se alcanzan los 150 fps con un tamaño de fotograma de 640×480, mientras que con la FPGA se obtienen 64 fps para la misma resolución. Por su parte, Marzat *et al.* [131] implementan una versión piramidal del algoritmo de Lucas & Kanade haciendo uso de CUDA con el que consiguen acelerar los cálculos 100× con respecto a la versión en CPU, obteniendo además 15 fps para una resolución de 640×480.

En el trabajo realizado por Duvenhage *et al.* [110] se presenta la implementación en GPU del algoritmo de Lucas & Kanade con el fin de emplearse en aplicaciones de estabilización de imagen. Para realizar esta tarea, implementan el algoritmo haciendo uso de *OpenGL Shading Language*[35] (GLSL), alcanzando, con la configuración mejor, 30 fps para un tamaño del fotograma de 512×512.

Phull *et al.* [132] desarrollan, haciendo uso de CUDA, el algoritmo descrito en [133] con el que consiguen optimizar y acelerar dicho algoritmo obteniendo *speedups* de 25× con respecto a la implementación en CPU y 236× teniendo en cuenta la implementación en CPU del algoritmo piramidal KLT [150], con el que se realiza la comparación.

Hoy en día existen muchos dispositivos en los que se pueden encontrar una cámara embebida, como en portátiles, videoconsolas o teléfonos móviles, además de una GPU para manejar gráficos 3D y otras tareas relacionadas. Haciendo uso de la dupla cámara+GPU, del Riego *et al.* [135] implementan, en un procesador gráfico, una versión paralela del algoritmo de flujo óptico *Hierarchical Lucas-Kanade* [107] (HLK). En dicha implementación obtienen 30 fps, el máximo soportado por la cámara utilizada en sus experimentos, además de conseguir reducir el porcentaje de uso de CPU, ya que los cálculos se realizan en GPU.

El trabajo desarrollado por Hegner *et al.* [137] presenta la implementación, en una GPU, del algoritmo de estimación de flujo óptico desarrollado en [138] y que utiliza filtros direccionales. Con él, obtienen resultados, para una secuencia de imágenes de entrada de tamaño 240×256, miles de veces más rápido que la correspondiente implementación en MATLAB[36]. El número de fps varía en función de las resoluciones que utilizan como datos de entrada, obteniendo 148 fps para resoluciones de 240×256 y de 36 fps para las de 512×512.

La simulación numérica para el procesamiento visual del cerebro humano es una de las aplicaciones que más tiempo consumen. En esta línea, Ohmura [140] y su equipo de trabajo muestran técnicas para acelerar un programa que simula del procesamiento visual realizando la estimación de movimiento mediante el algoritmo de Lucas & Kanade. Para ello, paralelizan cálculos de convolución en un clúster de de GPUs Tesla C1060 y C2070 [117]. Esta investigación incluye varias mejoras, como la asignación eficiente de datos entre la memoria global y la memoria compartida de la GPU, el cálculo en cada GPU de múltiples convoluciones

[35] `http://www.opengl.org/documentation/glsl/`
[36] `http://www.mathworks.com/products/matlab/index.html`

para los mismos datos de entrada y la división en regiones de la imagen de entrada para que dichas regiones se ejecuten en paralelo (haciendo uso de MPI).

Por su parte, Shiralkar *et al.* [141] presentan una versión paralela del algoritmo *Self-Organizing Map* [142] (SOM). Para el tamaño de fotograma más pequeño con el que se realizan las pruebas en este trabajo (320×240), el rendimiento que se obtiene es de 140 fps, mientras que para tamaños mayores obtienen 40 fps y 20 fps (para resoluciones de 512×284 y 640×480 respectivamente). Además, dependiendo del estímulo de entrada, alcanzan *speedups* entre 130× y 145×.

Para terminar con los trabajos relacionados con la estimación de movimiento en GPUs repasamos algunos de los realizados tomando como punto de partida los algoritmos basados en modelos de energía. Pauwels y Van Hulle proponen en [111] la utilización de una arquitectura GPU donde se implementa, haciendo uso de CUDA, un algoritmo de flujo óptico basado en fase [103]. Con la implementación en GPU obtienen alrededor de los 40 fps para resoluciones de 640×512 y un *speedup* de 150×. Este algoritmo de flujo óptico gira en torno a una medida de fiabilidad que evalúa la consistencia de la información de fase en el tiempo, y se caracteriza por su sencillez, robustez y velocidad, aunque no incorpora medidas de estabilización, aspecto crucial cuando se tienen en cuenta resoluciones mayores.

En [143], Sundaram *et al.* proponen un método para calcular la trayectoria de puntos mediante el algoritmo *Large Displacement Optical Flow* (LDOF) [144]. Dicha implementación se ejecuta 78× más rápida que la correspondiente versión en C++. Gwosdek *et al.* [145] presentan una implementación en GPU para el cálculo del flujo óptico en el sistema *Euler-Lagrange* [8, 151]. En este trabajo se ha utilizado el enfoque propuesto en [152] para reducir al mínimo el modelo de flujo óptico, consiguiendo obtener resultados óptimos en menos de un segundo para secuencias de tamaño 640×480. Para resoluciones más altas (1024×768), el *speedup* alcanzado varía entre 23× y 28× dependiendo del estímulo de entrada utilizado.

Para finalizar, el estudio desarrollado por Abramov *et al.* [146] se basa en un modelo de segmentación espacio-temporal en aplicaciones robóticas móviles. Para ello, mapean en GPU el algoritmo basado en fase desarrollado en [147] comparando los resultados obtenidos en una GPU Nvidia GeForce GT 240M y otra Nvidia GeForce GTX 295[37], la primera instalada en un portátil y la segunda en un sobremesa. Aunque los resultados obtenidos con la primera tarjeta son del orden de 2.1× más lentos que los obtenidos con la segunda, sigue siendo posible procesar varios fotogramas por segundo para las resoluciones consideradas en este estudio (160×128, 320×256, y 640×512).

2.3.3. Circuitos específicos (ASIC).

En este apartado se muestran algunos trabajos relacionados con la estimación de movimiento haciendo uso de circuitos específicos, ASIC (ver síntesis en la tabla 5).

[37] `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-295/specifications`

Autor	Familia	Algoritmo	fps (resolución)	Año
Warrington [153]	*Block Matching*	*VBSME*	30 (720×480)	2007
Verma [154]	*Block Matching*	*varios*	30 - 60 (varias)	2008
Sebastião [155]	*Block Matching*	*varios*	25 (352×288)	2008
Ndili [156]	*Block Matching*	*HMDS* [157]	-	2011
Dhahri [158]	*Block Matching*	*4SST*	-	2011
Stocker [159]	*Gradiente*	*Horn & Schunck*	-	2006

Tabla 5: Estimación de movimiento en circuitos específicos - Estado del arte.

A través de la síntesis de una arquitectura ASIC, Warrington *et al.* [153] proponen una arquitectura de alto rendimiento para VBSME (*Variable Block Size Motion Estimation*), de la familia de *Block Matching*, que soporta múltiples referencias a fotogramas (MRF). Para habilitar el mejor rendimiento en cuanto a ratio-distorsión para diferentes contenidos de vídeo, la arquitectura permite seleccionar entre una búsqueda espacial de movimiento en alta resolución sobre un marco de referencia, o una búsqueda MRF con una resolución espacial menor. Comparando este sistema con un algoritmo de búsqueda exhaustiva se obtiene una reducción en el *bitrate* y en la complejidad computacional. Además, el algoritmo implementado consigue un rendimiento similar al obtenido con la búsqueda exhaustiva a 30 fps con una resolución de 720×480.

Otro estudio interesante es el llevado a cabo por Verma y Akoglu [154], donde se propone una arquitectura híbrida reconfigurable de grano grueso sobre un mecanismo NoC[38]. Para ello hacen uso de VBSME así como otros algoritmos de *Block Matching* (FST, HS, *Big Hexagon Search* -BHS, SS [160] y DS). El diseño propuesto utiliza un alto nivel de paralelismo y una reutilización intensiva de los datos. Además, permite cualquier tamaño de bloque, lo que, a priori, supone un problema desde la perspectiva ASIC. Otro punto interesante es que el sistema desarrollado necesita una frecuencia de reloj superior a 0,8 MHz, para mantener los 30 fps para tamaños pequeños de fotograma (176×144), y a 29.16 MHz para soportar los 60 fps requeridos para alta definición (1280×720).

Por su parte, Sebastião *et al.* [155] comparan dos implementaciones para codificación de vídeo, una en FPGA y otra en ASIC, realizadas ambas con IP *core*[39]. Centran su atención en la estimación de movimiento en tiempo real mediante distintos algoritmos de la familia *Block Matching* (FST, TSST y DS), ya que supone el mayor coste computacional de este tipo de sistemas. En este caso, la implementación ASIC es más adecuada para dispositivos que utilizan baterías mientras que la capacidad de reconfiguración de la FPGA permite que la estimación de movimiento adapte dinámicamente el codificador de vídeo a las características de la aplicación. Escogiendo como base dos formatos[40] diferentes de fotograma, CIF

[38] http://en.wikipedia.org/wiki/Network_On_Chip
[39] http://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core
[40] http://www.springerreference.com/docs/html/chapterdbid/10489.html

(352×288) y QCIF (176×144), el máximo ratio por fotograma conseguido en cada uno de los casos es:

- Para el algoritmo FST se consiguen 2,07 fps en la FPGA y de 3,33 fps en ASIC para el formato CIF y 8,29 fps (FPGA) y 13.30 fps (ASIC) para el formato QCIF.
- En el caso del algoritmo TSST, en la FPGA se consiguen 21,12 fps (CIF) y 84,49 fps (QCIF) mientras que para el ASIC el ratio es de 33,88 fps (CIF) y 135,52 fps (QCIF).
- Para la implementación en ASIC, el algoritmo DS presenta un ratio de 23,53 fps y 94,12 fps para los formatos CIF y QCIF respectivamente, y de 14,67 fps y de 56,68 fps para la realizada en la FPGA.

Ndili y Ogunfunmi [156] proponen un algoritmo eficiente de estimación de movimiento [157] sobre una arquitectura SoC[41] con el objetivo de procesar vídeos de calidad alta en dispositivos móviles de bajo consumo. Los resultados obtenidos alcanzan hasta los 15456,05 Kb/s. en el caso de una secuencia de entrada de 90 fotogramas con un tamaño de 1280×720 (720p) cada uno.

Como último ejemplo de aplicación de algoritmos de *Block Matching* sobre ASIC, Dhahri *et al*. [158] proponen una arquitectura paralela para estimación de movimiento mediante la técnica de búsqueda en cuatro pasos (4SST). Para ello desarrollan un método que usa 9 elementos de procesamiento, 2 memorias locales (una para el bloque referencia y otra para el área de búsqueda), una unidad de comparación y contadores para controlar las direcciones de memoria permitiendo así procesar *pixels* en paralelo y obteniendo resultados en tiempo real, aunque no ofrecen datos precisos del rendimiento obtenido.

Con respecto a la implementación de modelos de gradiente en arquitecturas ASIC, Alan A. Stocker introduce en [159] un nuevo sensor analógico VLSI (del inglés *Very Large Scale Integrated*[42]) que estima el flujo óptico en dos dimensiones visuales. Su arquitectura computacional consiste en una red de dos capas de unidades de movimiento, conectadas localmente con el fin de estimar de manera conjunta el campo de flujo óptico óptimo. Para ello implementa el algoritmo de Horn & Schunck [8] con una restricción parcial que mide la distancia del flujo óptimo estimado con respecto a un vector de movimiento de referencia.

2.4. Estímulos y métricas relacionadas con el flujo óptico.

En este apartado se presentan los estímulos de entrada que se han tenido en cuenta en este trabajo para poder comprobar y comparar los resultados obtenidos. Posteriormente, se explican las métricas relacionadas con el flujo óptico que han sido consideradas a la hora de comparar y de medir la calidad de los datos resultantes.

[41] http://www.springerreference.com/docs/html/chapterdbid/311302.html
[42] http://www.springerreference.com/docs/html/chapterdbid/311381.html

2.4.1. Estímulos de flujo óptico.

Las principales ventajas de los estímulos sintéticos son que los campos de movimiento de dos dimensiones y las propiedades de la escena se pueden controlar y probar de manera metódica. En particular, al tener conocimiento sobre el movimiento en 2-D se puede cuantificar el rendimiento. Sin embargo, cabe destacar que este tipo de estímulos son señales limpias, es decir, no implican oclusión, sombras, transparencias, etc. y por lo tanto la medida de rendimiento se debe tomar como una medida optimista.

Además, debido al problema de la apertura [161, 162], también es necesario generar estímulos sintéticos con el objetivo de poder evaluar el rendimiento de los algoritmos de flujo óptico. Ejemplos de ello son las traslaciones de senos y senos compuestos, ya que tienen suficientes frecuencias espaciales y temporales como para ser considerados buenos candidatos. A continuación se presentan algunos de los estímulos utilizados en esta tesis.

Como punto de partida, se puede hablar del estímulo sintético genérico en el que se han tenido en cuenta dos escenarios de entrada (figura 13), una translación de un seno en la dirección del eje x con una velocidad $v = (-1, 0)$ y la translación de dos sinusoidales de onda plana con una velocidad $v = (1, 1)$.

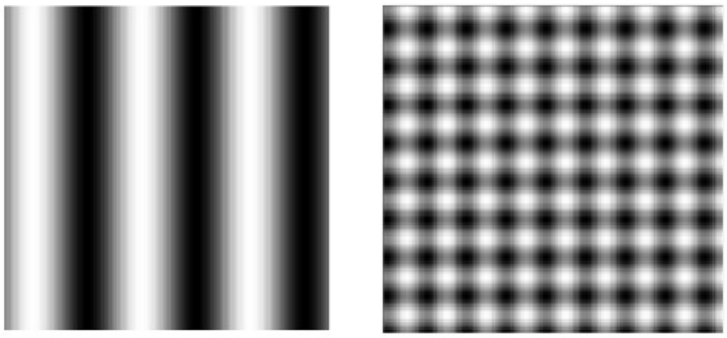


Figura 13: Estímulos sintéticos utilizados para evaluar algoritmos. Izquierda: translación seno. Derecha: translación rejilla.

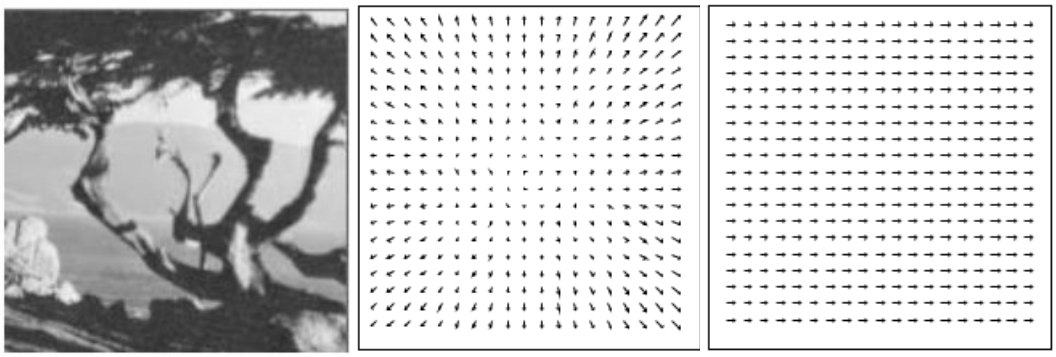
La entrada sinusoidal viene representada por la ecuación 6. Esta señal permite una detección muy precisa de los bordes y una diferenciación numérica.

$$sin(k_1 x + \omega_1 t) + sin(k_2 x + \omega_2 t) \tag{6}$$

Se suelen tener en cuenta también estímulos de entrada más complejos, como los representados en la figura 14, que incluyen las secuencias sintéticas "*Diverging Tree*" y "*Translating Tree*"[43], creadas por David Fleet en la Universidad de Toronto [163]. El estímulo "*Diverging Tree*" muestra un movimiento expansivo de un árbol,como el del zoom de las cámaras, con

[43] Disponibles en la siguiente dirección web: `http://www2.fz-juelich.de/icg/icg-3/Mitarbeiter/Scharr/Testdata`

un rango de velocidades asimétrico dependiendo de la posición del *pixel* (nulo en el centro y 2 *pixels*/*frame* y 1.4 *pixels*/fotograma en los márgenes derecho e izquierdo respectivamente). El estímulo denominado "*Translating Tree*" muestra un movimiento de translación de un árbol con un rango de velocidades asimétrico dependiendo de la posición del *pixel* (desde 0 hasta 1.73 *pixels*/fotograma y desde 0 hasta 2.3 *pixels*/*frame* en los bordes izquierdo y derecho respectivamente). Es decir, la sensación es que el movimiento se produce a lo largo del eje de las X (figura 14 (III)).



(I) (II) (III)

Figura 14: Textura utilizada en el '*Diverging Tree*' y el '*Translating Tree*' con su representación del movimiento para ambos casos.

La secuencia conocida como "*Yosemite*"[44] (figura 15), creada por Lynn Quam en el *Stanford Research Institute*, es un caso algo más complejo. El movimiento en la parte superior derecha es divergente, principalmente. Las nubes se trasladan hacia la izquierda con una velocidad de 1 *pixel*/*frame* mientras que las velocidad en la parte inferior izquierda están en torno a los 4 *pixels*/*frame*. Esta secuencia es un reto debido al rango de velocidades y a la oclusión de bordes entre las montañas y el horizonte. Además, cabe destacar la existencia de un alto grado de aliasing en la parte inferior de las imágenes, lo que puede causar mediciones poco exactas con la mayoría de los métodos de estimación de flujo óptico.
Además de utilizar estímulos sintéticos como entrada para los algoritmos de flujo óptico, estos se han de probar con secuencias de imagen real. Entre las más utilizadas se encuentra la secuencia "*Hamburg taxi*"[45] (figura 16) en la que se puede observar una escena real realizada con una cámara estática y en la que aparecen cuatro objetos en movimiento: un taxi girando en la esquina; un coche en la parte inferior izquierda, que se mueve de izquierda a derecha; una furgoneta que lleva una dirección de derecha a izquierda, en la parte inferior derecha; y un peatón en la parte superior izquierda. El movimiento de avance real para esta secuencia no está disponible, ya que no es una secuencia sintética.

[44] Disponible en: `http://cs.brown.edu/~black/images.html`
[45] Accesible desde: `http://i21www.ira.uka.de/image_sequences/`

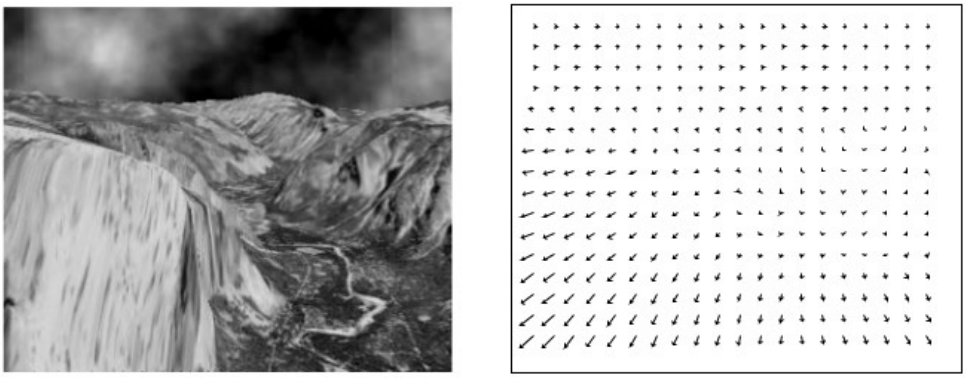


Figura 15: Fotograma de la secuencia '*Yosemite*' y su campo de flujo correcto.

Figura 16: Captura de pantalla de la secuencia real '*Hamburg taxi*'.

2.4.2. Métricas relacionadas.

Con el fin de medir el error respecto a la implementación presentada en este trabajo, se hace necesario usar una métrica de error. Una de las métricas más aceptadas es la de Barron [57]. La ecuación (7) muestra la desviación a partir de la orientación espacio-temporal correcta, la velocidad de un vector de dirección 3D unitario. Este vector incluye tanto el módulo (velocidad) como la fase (dirección) en un único valor, reduciendo el aumento de los errores de dirección para velocidades pequeñas.

$$\vec{v} = \frac{1}{\sqrt{u^2 + v^2 + 1}} (u, v, 1)^T \tag{7}$$

El ángulo ψ_E entre la velocidad obtenida v_e y la correcta v_c se representa en la ecuación (8).

$$\psi_E = \arccos (\vec{v_c} \cdot \vec{v_e}) \tag{8}$$

Aunque probablemente la métrica de Barron es la más utilizada en el ámbito de la estimación de movimiento, existen otras métricas que se utilizan en la comunidad de los Sistemas de

Visión y que se han de tener en cuenta con el fin de aumentar la visibilidad y la generalidad de los resultados obtenidos.
Una métrica de error alternativa, propuesta por Galvin *et al.* [164] y reflejada en las ecuaciones 9 y 10, modela, por una parte, la magnitud de la diferencia entre la medida correcta y la estimada:

$$\Psi_M = \|\overrightarrow{v_c} - \overrightarrow{v_e}\| \tag{9}$$

y por otra, una medida del error normal a la dirección del gradiente de la intensidad g, siendo esta última un buen indicativo de la interacción con el problema de la apertura:

$$\Psi_N = \left\|(\overrightarrow{v_c} - \overrightarrow{v_e}) \cdot \hat{g}^{\perp}\right\| \tag{10}$$

En el trabajo de Otte y Nagel [165] se destaca el hecho de la asimetría y el sesgo de vectores de flujo óptico amplios. En base a esto, se propone un nuevo indicador que representa la diferencia de magnitud entre el vector de flujo bidimensional (v_c) y el estimado (v_e) como se muestra en la ecuación 11.

$$\Psi_{O\&N} = \left\|o\hat{f}v_c - o\hat{f}v_e\right\| \tag{11}$$

McCane *et al.* [166] ponen de manifiesto que la métrica descrita no es suficiente debido a que no tiene en cuenta el error cometido en regiones de flujos pequeños. Por ello proponen dos métricas con el objetivo de solventar este problema. La primera métrica es la diferencia del ángulo entre el vector tridimensional v_c y el estimado v_e utilizando la métrica de Barron (ecuación 12) pero remplazando la tercera componente por δ. Este umbral modula el error teniendo en cuenta que en las zonas de flujo más pequeño es menos significativo que en las zonas de flujo grande.

$$\Psi_{McCane_A} = cos^{-1}\left(\hat{v_c}, \hat{v_e}\right) \tag{12}$$

La segunda métrica propuesta es la normalización del vector diferencia entre los vectores de flujo tridimensional estimado y correcto. El factor de normalización es la magnitud del flujo correcto, teniéndose en cuenta el efecto de los flujos pequeños utilizando un umbral significativo T como se muestra en la ecuación 13. El efecto de este umbral se traduciría en un error normalizado igual a la unidad.

$$\Psi_{McCane_B} = \begin{cases} \frac{\|v_c - v_e\|}{\|v_c\|} & if \quad \|v_c\| \geq T \\ \frac{\|v_e - T\|}{\|T\|} & if \quad \|v_c\| < T \leq \|v_e\| \\ 0 & if \quad \|v_c\| < T > \|v_e\| \end{cases} \tag{13}$$

La figura 17 representa un esquema de diferentes criterios de medida del error. En ella se incluye (a) la medida basada en los criterios de Barron , (b) la medida basada en los criterios de Galvin usual y perpendicular y (c) la medida basada en diferencias de módulo y fase de vectores teóricos y experimentales.

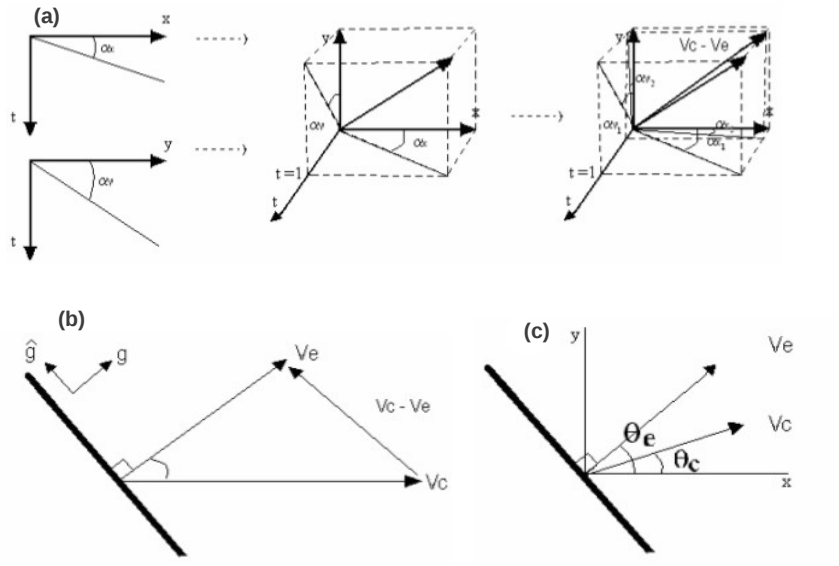


Figura 17: Esquema de diferentes criterios de medida del error.

3. McGM como caso de estudio.

Según se ha estudiado en el capítulo 2, existen tres estrategias para el calculo del flujo óptico: los métodos de correlación, o *Block Maching*, los métodos de energía y los diferenciales o de gradiente.

En este capítulo se va a exponer una alternativa a los tres modelos previos cuyo objetivo es cumplir con los requisitos de: invarianza al patrón estático sin perder información y al contraste, consistencia biológica (uso de filtros espaciales y temporales que se encuentran en el ser humano) y el uso de operaciones realizadas mediante dichos filtros. De esta manera, se diseña un modelo capaz de cumplir estas condiciones y que posee propiedades que no se encuentran en otros modelos de gradiente convencionales que son: capacidad para separar las componentes diferenciales del flujo óptico sin necesidad de recurrir a métodos iterativos, consistencia matemática frente a regiones donde escasamente hay contraste, diversidad para fundamentar procesamiento de movimiento de segundo orden[46] y robustez frente a ruido (dinámico y estático), entre otras. Es por ello por lo que se va a describir la estructura matemática del modelo multicanal de gradiente o McGM [72, 169] (*Multi-channel Gradient Model*).

3.1. Introducción.

El modelo McGM expande la ecuación básica de conservación de la intensidad para flujo óptico, de manera que expresa la imagen como un desarrollo de Taylor e introduce derivadas de órdenes superiores en la descripción. Debido a esto, se obtiene un método idóneo capaz de analizar la estructura espacio-temporal de una secuencia de imágenes consiguiendo a su vez una mayor robustez. A través del cociente de la derivada temporal y espacial de varios términos del desarrollo de Taylor (computando este desarrollo para un número de direcciones que se corresponden a las orientaciones de las columnas encontradas en el córtex visual primario), se obtiene una medida de la velocidad como función de la orientación del filtro.

Esta estrategia de múltiples medidas ayuda a reducir el ruido además de permitir separar la componente de translación de la diferencial. Puesto que cada cociente individual está mal condicionado en puntos donde el denominador se hace nulo, se combinan las derivadas de varios términos de este desarrollo y se integran en un volumen espacio-temporal. A partir de todos estos datos, se producen las primitivas de velocidad inversa y velocidad que, a su vez, generan la estimación de velocidad en fase y módulo para cada punto calculado, para al final obtener un mapa denso de información. A modo de resumen, la figura 18 muestra un esquema simple de las etapas del algoritmo que se van a explicar a continuación.

[46] Movimiento de segundo orden: aquel en el que el movimiento de contorno se define por el contraste, la textura, el parpadeo o alguna otra cualidad que no hace aumentar la luminancia o el movimiento de energía del espectro de Fourier del estímulo [167, 168].

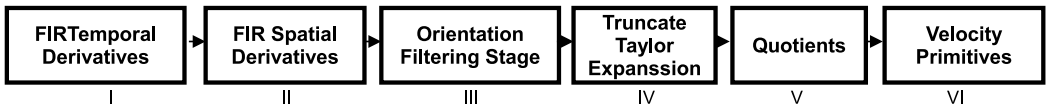


Figura 18: Esquema de las etapas del Modelo Multicanal de Gradiente (McGM).

3.2. Etapa I. Filtros temporales.

Hess y Snowden investigaron el procesamiento visual humano con una serie de experimentos [170] encontrando evidencias de 3 canales temporales: un canal paso baja, un canal paso banda con frecuencia central aproximada a 10 Hz y otro paso banda con frecuencia central a 18 Hz. Esos 3 canales se pueden modelar por diferenciación de una Gaussiana en el dominio del logaritmo del tiempo (figura 19).

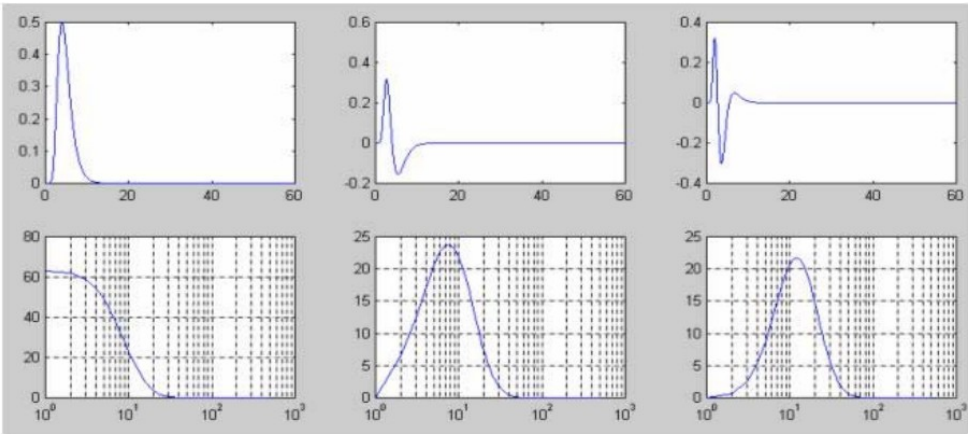


Figura 19: Representación de los tres canales temporales encontrados en el humano [171] desde el punto de vista de su respuesta impulsiva (arriba) y su comportamiento en frecuencia (abajo).

Estos filtros se modelan como derivadas de una Gaussiana en el espacio temporal logarítmico ($log(tiempo)$), con $\alpha = 10$, $\tau = 0{,}2$, tal y como se describe en la ecuación 14.

$$n\acute{u}cleo = \frac{e^{-(log(t/\alpha)/\tau)^2}}{\sqrt{\pi}\alpha e^{(\frac{\tau^2}{4})}} \tag{14}$$

El modelo hace uso de un filtro FIR (*Finite Impulse Response*) con el objetivo de implementar las derivadas temporales. No se obtendrán salidas hasta que se hayan procesado tantos fotogramas en el filtro como longitud de éste por la naturaleza del proceso de filtrado, es decir, si se tiene un filtro temporal de longitud 15, hasta el instante $t = 15$ (con 15 imágenes) no será posible obtener la respuesta al fotograma dado por la posición α.

Si realizamos este triple proceso convolutivo desde el primer fotograma ($t = 0$) hasta el

que ocupe la última posición menos la longitud del filtro ($t = n - L + 1$), conseguimos un conjunto de derivadas temporales de orden 0, 1 y 2 capaces de actuar como primeros canales de información, pudiéndose extraer a partir de ellos cada una de las variaciones espaciales asociadas a la estructura de la imagen.
El algoritmo 1 muestra el pseudocódigo de los cálculos realizados en la etapa de filtrado temporal.

Algorithm 1 $temp\_filt = etapaI(frames, nFrames, L, nTemp\_filters, \alpha, \tau)$

for $tf = 0$ to $nTemp\_filters$ **do**
  $T\_filters(tf) = obtener\_filtro\_temporal(\alpha, \tau)$ {Obtención de filtros}
  for $fr = 0 \leq nFrames - L$ **do**
    $frame = obtener\_frames(frames, fr)$
    for all $p = pixel \in frame$ **do**
      $temp\_filt[tf][fr] \leftarrow [frames(p[0 : L - 1]) \otimes T\_filters]$ {Convolución de p y T_filters}
    end for
  end for
end for

A continuación, se procede a diseñar y justificar todas las operaciones espaciales que irán encadenadas a su vez a las convoluciones temporales abordadas en este apartado.

3.3. Etapa II. Filtros espaciales.

En el dominio espacial, la forma de los campos receptivos de las células en el córtex visual primitivo puede ser modelada con derivadas Gaussianas [171]. A medida que el orden de diferenciación aumenta, las Gaussianas se ajustan a frecuencias espaciales más elevadas, obteniendo un rango de canales espaciales independientes entre sí, que han sido verificados experimentalmente, como se ilustra en la figura 20 y se modela según la expresión 15:

$$\frac{d^n}{dx^n}(G_0) = \frac{d^n}{dx^n}\left(\frac{e^{-\frac{x^2+y^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}\right) = H_n\left(\frac{x}{\sqrt{2\sigma}}\right) H_n\left(\frac{y}{\sqrt{2\sigma}}\right)\left(\frac{-1}{\sqrt{2\sigma}}\right)^{2n}\left(\frac{e^{\frac{-x^2+y^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}\right) \quad (15)$$

Donde σ representa la anchura de la Gaussiana y H_n es el polinomio de Hermite de orden *n*. La convolución se realiza de forma separable, usando derivadas tomadas en filas y columnas respectivamente, debido a la separabilidad de la Gaussiana bidimensional. Este tipo de filtros son muy útiles en el campo de la visión artificial, especialmente en aplicaciones relativas al análisis de orientación local, filtrado angular adaptativo, detección de bordes o formación de imágenes tridimensionales a partir de sombras bidimensionales.

El proceso total equivale primero: a convolucionar la imagen con una Gaussiana, segundo: se convoluciona este resultado previo con el operador derivada. Este proceso puede llevarse a cabo gracias a la linealidad de los filtros. Sin embargo, este esquema se comporta como

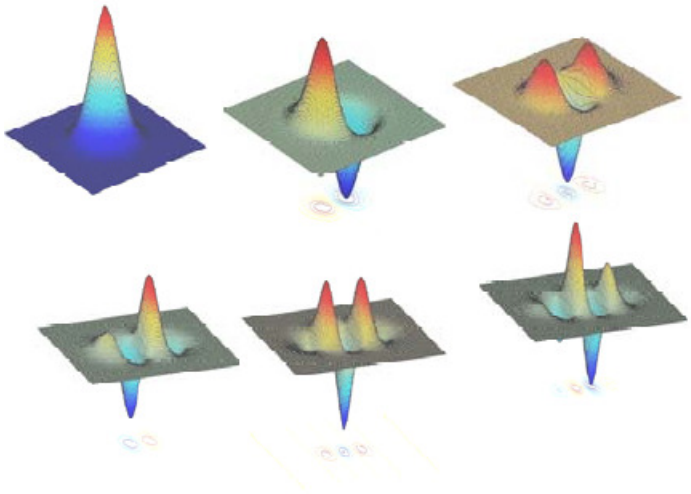


Figura 20: Representación de Gaussiana bidimensional y sus diferentes derivadas teniendo en cuenta la ecuación 1. La fila superior representa las derivadas de orden 0, 1 y 2. Fila inferior, derivadas de orden 3, 4 y 5.

un filtro paso alta, que responde bien a partir de determinadas frecuencias espaciales. El producto de los dos es un filtro paso banda, cuya frecuencia de corte inferior viene limitada por la asociada a la Gaussiana y la superior viene limitada por cada una de las derivadas. El pseudocódigo de esta etapa se presenta en el algoritmo 2.

Algorithm 2 $spat\_filt = etapaII(temp\_filt, nFrames, L, nTemp\_filters, nSpat\_filters)$

```
for sf = 0 to nSpat_filters do
    S_filt[sf] = obtener_filtro_espacial(sf) {Obtención de filtros}
end for
for tf = 0 to nTemp_filters do
    for fr = 0 ≤ nFrames − L do
        frame = obtener_frames(temp_filt, tf, fr)
        for all p = pixel ∈ frame do
            for sf = 0 to nSpat_filters do
                spat_filt[sf][tf][p] ← conv2D(temp_filt[tf][fr][p], S_filters(sf))
            end for
        end for
    end for
end for
```

3.4. Etapa III. Orientación de filtros.

El modelo multicanal de gradiente utiliza unos cuantos filtros orientados que deben intentarse evitar en sistemas de tiempo real por la alta carga de las convoluciones. En el contexto computacional, los filtros separables son deseables y ventajosos [172], ya que son necesarios

para generar un conjunto básico de respuestas. Este proceso ha de ser compatible con que se mantengan unas cotas de eficiencia adecuada en el proceso de orientación.

La etapa de orientación de filtros en el espacio (*steering*) representa la proyección de los filtros calculados previamente en diferentes orientaciones en el espacio. En la ecuación 16 se representa la expresión general de las diferentes rotaciones de la Gaussiana y sus derivadas. Se denomina n y m el orden de diferenciación en las direcciones x e y, θ el ángulo proyectado, D el operador derivativo y G_0 la expresión de la Gaussiana. Esta expresión se resume en expresar cada filtro orientado como una combinación lineal de los filtros de su mismo orden de diferenciación, como se demuestra en el trabajo de G. Botella [172].

$$G^{\theta}_{n,m} = \left[\sum_{k=0}^{n} \binom{n}{k} (D_x cos\theta)^k (D_y sin\theta)^{n-k}\right] \cdot \left[\sum_{i=0}^{m} \binom{m}{i} (-D_x sin\theta)^i (D_y cos\theta)^{m-i}\right] G_0 \tag{16}$$

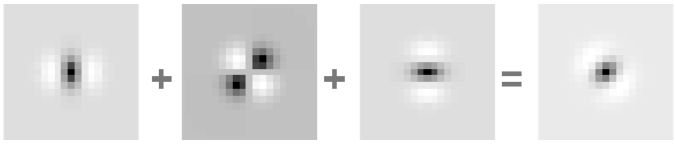


Figura 21: Esquema de orientación de filtros de segundo orden (45^o).

Además, la figura 21 muestra un ejemplo de orientación de filtros, como una combinación lineal de los tres filtros de segundo orden. Es posible apreciar dos cruces por cero de los lóbulos de las funciones representadas. En consonancia, en la figura 22 se ilustra un desarrollo de filtros orientados de primer orden[47], así como las contribuciones propias de cada filtro de la base.

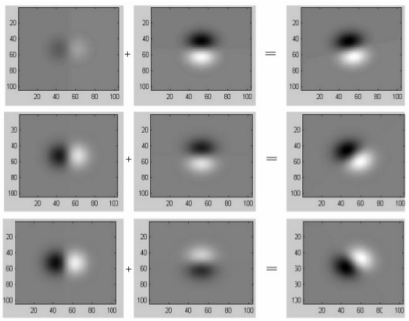


Figura 22: Ejemplos de rotación de filtros orientados para varios ángulos avanzando según el sentido contrario al de las agujas del reloj. De arriba a abajo, implementación de filtros de 1^{er} orden para 105^o, 135^o, 215^o respectivamente.

[47] Percepción de primer orden de movimiento se refiere a la percepción del movimiento de un objeto que difiere en luminancia de su fondo.

El pseudocódigo de esta etapa se muestra en el algoritmo 3.

Algorithm 3 $R^{\theta} = etapaIII(spac\_filt, G^{\theta}, nFrames, L, nTemp\_filts, nSpat\_filts, nOrth\_Orders, n\theta s)$

```
for θ = 0 to nθs do
  for oo = 0 to nOrth_Orders do
    for sf = 0 to nSpat_filts do
      for tf = 0 to nTemp_filts do
        for fr = 0 ≤ nFrames − L do
          frame = obtener_frames(spac_filt, oo, sf, tf, fr)
          for all p = pixel ∈ frame do
            I = spac_filt[sf][tf][p]
            R^θ[oo][sf][tf][p] ← [G^θ ⊗ I] {Convolución G y I}
          end for
        end for
      end for
    end for
  end for
end for
```

3.5. Etapa IV. Desarrollo de Taylor.

En esta etapa se realiza un desarrollo de Taylor usando cada filtro orientado en la etapa anterior. Esta función representa una estructura robusta que recoge toda la información espacio-temporal de las secuencias, aproximando un *pixel* genérico por el conjunto de las derivadas de los vecinos. Se puede expresar con la ecuación (17).

$$I\left(x+p, y+q, t+r\right) = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n}\frac{p^i q^j r^k}{i!j!k!}\frac{\partial^n}{\partial x^i \partial y^j \partial t^k} I\left(x, y, t\right) \tag{17}$$

Las tres derivadas de Taylor han sido construidas en una imagen de considerable tamaño usando el conjunto completo de filtros de respuestas básico. De acuerdo con el modelo original [169], la expansión ha sido truncada después del tercer orden en la dirección primaria y del segundo orden en las direcciones ortogonal y temporal.

En el algoritmo 4 se representa el pseudocódigo de esta etapa.

Algorithm 4 $I^{\theta} = etapaIV(R^{\theta}, nFrames, L, nTemp\_filts, nSpat\_filts, nOrth\_Orders, n\theta s)$

for $\theta = 0$ to $n\theta s$ **do**
  for $sf = 0 \leq nSpat\_filts$ **do**
    for $tf = 0 \leq nTemp\_filts$ **do**
      for $fr = 0 \leq nFrames - L$ **do**
        $frame = obtener\_frames(R^{\theta}, \theta, sf, tf, fr)$
        for all $p = pixel \in frame$ **do**
          $//Tercer\ orden\ en\ dirección\ primaria$
          $//Segundo\ orden\ en\ direcciones\ ortogonal\ y\ temporal$
          $I^{\theta} = TaylorTruncation(R^{\theta})$
        end for
      end for
    end for
  end for
end for

3.6. Etapa V. Cocientes.

Esta es la última etapa derivada para el cálculo de ruta común. Las próximas etapas implementan la estimación del módulo y la fase con expresiones distintas. El objetivo de esta etapa es calcular los cocientes de cada componente del sexteto de la siguiente expresión:

$$\left.\begin{matrix} X = \partial I/\partial x \\ Y = \partial I/\partial y \\ T = \partial I/\partial t \end{matrix}\right| \rightarrow \begin{matrix} XX & XY & XT \\ YY & YT & TT \end{matrix} \left| \rightarrow \begin{matrix} YT/TT & XY/XX & XT/XX \\ YT/YY & XY/YY & XT/TT \end{matrix} \right. \tag{18}$$

Las operaciones realizadas en esta etapa pueden estudiarse en el pseudocódigo que se representa en el algoritmo 5.

Algorithm 5 $[XX^{\theta}, XY^{\theta}, XT^{\theta}, YY^{\theta}, TY^{\theta}, TT^{\theta}] = etapaV(I^{\theta}, nFrames, L, n\theta s)$

for $\theta = 0$ to $n\theta s$ **do**
  for $t\_Taylors$ **do**
    for $y\_Taylors$ **do**
      for $x\_Taylors$ **do**
        for $fr = 0 \leq nFrames - L$ **do**
          $frame = obtener\_frames(I, \theta, t\_Taylor, y\_Taylor, fr)$
          for all $p = pixel \in frame$ **do**
            $X = \delta I^{\theta}/\delta x$
            $Y = \delta I^{\theta}/\delta y$
            $T = \delta I^{\theta}/\delta t$
            $XX^{\theta} = calcular\_derivada(X, X)$
            $XY^{\theta} = calcular\_derivada(X, Y)$
            $XT^{\theta} = calcular\_derivada(X, T)$
            $YY^{\theta} = calcular\_derivada(Y, Y)$
            $YT^{\theta} = calcular\_derivada(Y, T)$
            $TT^{\theta} = calcular\_derivada(T, T)$
          end for
        end for
      end for
    end for
  end for
end for

3.7. Etapa VI. Primitivas de velocidad.

Las etapas anteriores calculan la información visual considerando una representación de Taylor para cada *pixel* y calculando la velocidad para un rango de orientaciones a fin de simular las columnas de orientación que se encuentran en la corteza estriada [169]. Esto se realiza rotando el sistema y los filtros derivados Gaussianos de manera coordinada (etapa de orientación) a un número de direcciones primarias. Después de esto, se toman las medidas de velocidad, paralela y ortogonal, a las direcciones primarias para producir un vector de medidas de velocidad, cuyos componentes son la velocidad y la velocidad ortogonal:

$$\hat{s} = \left(\hat{s}_{\parallel}, \hat{s}_{\perp}\right) \tag{19}$$

Las medidas en bruto de la velocidad también están condicionadas mediante la inclusión de las medidas de la estructura de la imagen $X\Delta Y/X\Delta X$ y $X\Delta Y/Y\Delta Y$ donde los vectores finales condicionados de velocidad vienen dados por:

$$\hat{s}_{\parallel} = \sqrt{\frac{2}{\Sigma}} \left[\frac{XT}{XX} \left(1 + \left(\frac{XY}{XX} \right)^2 \right)^{-1} \right] \qquad \hat{s}_{\perp} = \sqrt{\frac{2}{\Sigma}} \left[\frac{YT}{YY} \left(1 + \left(\frac{XY}{YY} \right)^2 \right)^{-1} \right] \tag{20}$$

donde $\sum$ es el número de orientaciones en las que se evalúa la velocidad. También se calcula la velocidad inversa $\breve{s}_{\parallel}$ siguiendo la ecuación (21):

$$\breve{s}_{\parallel} = \sqrt{\frac{2}{\Sigma}} \left[\frac{XT}{TT} \right] \qquad \breve{s}_{\perp} = \sqrt{\frac{2}{\Sigma}} \left[\frac{YT}{TT} \right] \tag{21}$$

La velocidad inversa se calcula usando diferentes términos de los utilizados para calcular la velocidad, y por tanto constituye una medición independiente adicional. Finalmente, el módulo del movimiento se calcula a través de un cociente de determinantes:

$$M\acute{o}dulo^2 = \left[\begin{array}{cc} \hat{s}_{\parallel} \cos\theta & \hat{s}_{\parallel} \sin\theta \\ \hat{s}_{\perp} \cos\theta & \hat{s}_{\perp} \sin\theta \\ \hline \hat{s}_{\parallel}\breve{s}_{\parallel} & \hat{s}_{\parallel}\breve{s}_{\perp} \\ \hat{s}_{\perp}\breve{s}_{\parallel} & \hat{s}_{\perp}\breve{s}_{\perp} \end{array} \right] \tag{22}$$

La dirección del movimiento se extrae calculando las medidas de fase que se combinan a través de todas las medidas relacionadas con la velocidad, puesto que están en fase:

$$Fase = \arctan \left(\frac{\left(\breve{s}_{\parallel} + \hat{s}_{\parallel}\right) \sin\theta + \left(\breve{s}_{\perp} + \hat{s}_{\perp}\right) \cos\theta}{\left(\breve{s}_{\parallel} + \hat{s}_{\parallel}\right) \cos\theta - \left(\breve{s}_{\perp} + \hat{s}_{\perp}\right) \sin\theta} \right) \tag{23}$$

Por último, en el algoritmo 6 se expresan, en pseudocódigo, las operaciones realizadas en esta etapa.

Algorithm 6 $[fase, módulo] = etapaVI(nFrames, L, n\theta s, XX^{\theta}, XY^{\theta}, XT^{\theta}, YY^{\theta}, YT^{\theta}, TT^{\theta})$

```
for fr = 0 ≤ nFrames − L do
  frame = obtener_frames(stageV, fr)
  for all p = pixel ∈ frame do
    módulo_{fr,p} = fase_{fr,p} = 0 //Inicio
    for θ = 0 to nθs do
      ŝ = calcular_velocidad(XX^θ, XY^θ, XT^θ, YY^θ, YT^θ, TT^θ)
      módulo_{fr,p} = módulo_{fr,p} + obtener_módulo(ŝ, θ)
      fase_{fr,p} = fase_{fr,p} + obtener_ángulo(ŝ, θ)
    end for
  end for
end for
```

3.8. Publicaciones relacionadas.

En los últimos años se pueden encontrar en la bibliografía estudios relacionados con la estimación de movimiento que utilizan el algoritmo McGM con el objetivo de determinar el flujo óptico.

El primero del que hay que hablar es el realizado por Johnston *et al.* [73] en el que se presenta un modelo computacional del análisis de algunos patrones del movimiento de primer orden y de segundo orden realizado por células simples y complejas. También Bruce, Green y Georgeson abordan este algoritmo en [10]. Por su parte, McOwan *et al.* exponen en [173] un enfoque neuromórfico multidiferencial para la detección de movimiento haciendo uso del modelo multicanal de gradiente.

En el año 2003, Johnston, McOwan y Benton hacen uso del modelo en [12] para el cálculo de flujo de imágenes sobre una región limitada de la imagen. En este caso, el modelo recupera información precisa del flujo óptico sin recurrir a la regularización y es resistente al ruido. La teoría requiere una jerarquía de componentes de cómputo que se asignen bien a las propiedades conocidas de las neuronas sensibles al movimiento y a la arquitectura computacional del sistema visual.

Otro ejemplo más actual lo podemos encontrar en el trabajo realizado por Anderson y McOwan [174], en el que presentan un sistema de reconocimiento facial en tiempo real automático y en múltiples etapas. En este trabajo el algoritmo McGM se utiliza para determinar el flujo óptico de la cara, es decir, una vez que se ha localizado esta en la escena, el algoritmo determina el movimiento de la cara. La información que se obtiene se utiliza con el objetivo del reconocimiento facial, ya que, es posible observar las emociones básicas mediante los patrones del movimiento facial, independientemente de quien expresa la emoción.

Liang *et al.* [175] describen una extensión espectral del algoritmo McGM para abordar distinción de color. Para ello, extienden las series de derivadas Gaussianas, en el espacio de color, y las integran dentro del modelo multicanal de gradiente.

4. Unidades de procesamiento gráfico.

Las unidades de procesamiento gráfico, comúnmente conocidas por las siglas GPU, son procesadores paralelos optimizados para acelerar cálculos gráficos. Fueron diseñadas específicamente para realizar una gran cantidad de cálculos en coma flotante necesarios para el renderizado de gráficos 3D. Sin embargo, las GPUs modernas son masivamente paralelas y totalmente programables, tanto es así, que la capacidad de realizar los cálculos en punto flotante de forma paralela en una GPU moderna es de varios órdenes de magnitud superior que en una CPU. Además, las GPUs pueden encontrarse en un amplio rango de sistemas, desde ordenadores de sobremesa y portátiles hasta en dispositivos móviles o supercomputadores.

En este capítulo nos vamos a centrar en la evolución histórica de las GPUs, abordando la tecnología relacionada con el procesamiento gráfico con la que nos encontramos en la actualidad, así como las iniciativas y herramientas software para favorecer su explotación desde el punto de vista del programador.

4.1. Historia y estado del arte de las GPU.

La aparición de las primeras tarjetas gráficas data de finales de los años 60 cuando, en lugar de impresoras, se empiezan a utilizar los monitores como elementos de visualización. En estas primeras tarjetas solo era posible visualizar texto, aunque con la aparición posterior de chips gráficos, como el *Motorola 6845*, se empezó a dotar a los equipos de capacidades gráficas.

En 1987 se dan a conocer las primeras tarjetas gráficas que poseían una resolución de 640×480 y 256 colores, las conocidas VGA (del inglés *Video Graphics Array*), que tuvieron una aceptación masiva, provocando que muchas compañías trabajaran sobre esta tarjeta para mejorar su resolución y el número de colores. Sin embargo, no fue hasta 1995 cuando la evolución de este tipo de *hardware* dio un giro importante. Hasta esa fecha, las mejoras iban encaminadas a soportar mayores resoluciones y colores, pero gracias a que los videojuegos comenzaban a popularizarse, con la aparición de las videoconsolas, los requisitos de mayor realismo gráfico supusieron mayores demandas computaciones y comenzaron a aparecer las primeras tarjetas 2D/3D, que cumplían con el estándar SVGA (una evolución de VGA) implementando además algunas funciones 3D que las hacían mucho más potentes.

Los chips gráficos empezaron como procesadores gráficos de funciones fijas, pero se hicieron cada vez más programables y potentes desde el punto de vista computacional. Entre los años 1999 y 2000, científicos del sector informático y de otras disciplinas empezaron a utilizar las GPU para acelerar diversas aplicaciones científicas, ya que hasta ahora su principal uso estaba orientado al mundo de los videojuegos. Fue el nacimiento de un nuevo concepto denominado GPGPU o la utilización de la GPU para aplicaciones de propósito general.

Aunque los usuarios lograban alcanzar un rendimiento muy superior a las CPUs de la época (por encima de 100x con respecto a las CPU en algunos casos), el problema era que el concepto GPGPU precisaba el uso de APIs de programación de gráficos, como OpenGL y Cg, para

explotar el paralelismo de las GPU. Eso limitaba el acceso de la comunidad científica a las enormes capacidades computaciones de este tipo de *hardware*. NVIDIA advirtió el potencial que supondría facilitar la programación de estos dispositivos, de modo que invirtió grandes esfuerzos para que la GPU fuera totalmente programable y ofreció un proceso totalmente transparente para desarrolladores con lenguajes familiares como C/C++, Fortran, OpenCL, DirectCompute, etc. Como consecuencia, en el año 2006 NVIDIA presenta la arquitectura CUDA (del inglés *Compute Unified Device Architecture*) a la par que la arquitectura gráfica G80 y la primera tarjeta que la soportaba (el modelo *GeForce 8800 GTX*), para alcanzar el objetivo propuesto en la computación a nivel de GPU: realizar la computación de carácter general de forma trivial. Desde entonces, esta arquitectura ha ido evolucionando hasta el modelo actual: *Kepler* [38].

Sin embargo, CUDA es un modelo que solo puede utilizarse en arquitecturas NVIDIA, por lo que a finales del 2008 aparece un estándar para la computación GPGPU llamado *OpenCL* [176], desarrollado por el grupo *Khronos* y cuyo fin es el de ser utilizado en plataformas heterogéneas con CPUs, GPUs, DSPs (Procesadores Digitales de Señal) o FPGAS entre otros procesadores.

La computación usando GPUs ha ido creciendo en importancia cada vez más rápido. Actualmente, dicha tecnología ha ido ganando presencia en las supercomputadoras más rápidas del mundo para incrementar sus prestaciones; la amplia mayoría de las universidades de todo el mundo incorporan entre sus estudios de informática computación paralela utilizando GPU y centenares de miles de desarrolladores las utilizan.

Desde 2003 y gracias en gran parte a la industria de los videojuegos, las tarjetas gráficas han ido evolucionando hasta convertirse en aceleradores consolidados. El microprocesador de dichas tarjetas, o Unidad de Procesado Gráfico, ha liderado la carrera de rendimiento en lo que a operaciones en punto flotante se refiere, tal y como muestra la figura 23, en la que se puede observar cómo la GPU GTX680 de NVIDIA alcanza un rendimiento de hasta 3 trillones de operaciones en punto flotante por segundo (3 TeraFLOPS) [177].

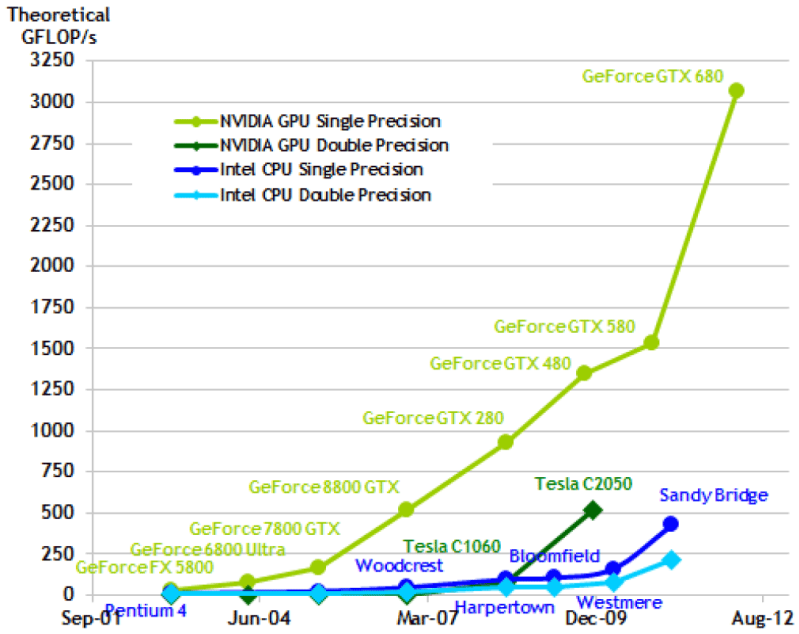


Figura 23: Comparativa del rendimiento en GFLOPS entre CPUs y GPUs.

La razón de esta diferencia en el rendimiento entre CPU y GPU es que la GPU está especializada en computación intensiva y computación masivamente paralela, que es exactamente lo que trata de explotar el renderizado de gráficos. Además, están diseñadas de forma que se destinan más transistores a la lógica computacional dotándola de más unidades de cómputo en lugar de a la jerarquía de memoria (*caches*) y control de flujo. Más concretamente, una GPU es eficiente a la hora de solucionar problemas que pueden poseer un alto grado de paralelismo; el mismo flujo de ejecución es ejecutado sobre un volumen de datos grande en paralelo con alta intensidad aritmética. Este hecho ha motivado que en algunos autores como Henessy&Patterson [178] describan la arquitectura GPU junto a los antiguos procesadores vectoriales por su analogía en la explotación del paralelismo SIMD.

Con este nuevo tipo de arquitectura presente, aparecen nuevas dificultades para el desarrollador de *software*. Ahora, no sólo se debe dominar la programación paralela tradicional, sino que se debe aprender a desarrollar aplicaciones para estas nuevas arquitecturas específicas, masivamente paralelas, y así aprovechar toda la potencia que ofrecen las soluciones *hardware* de hoy en día. Aún más complicado es obtener soluciones óptimas, aprovechar las ventajas de la jerarquía de memoria, evitar y/u ocultar los cuellos de botella, etc.

En la actualidad existen diferentes soluciones a la hora de acelerar nuestras aplicaciones mediante GPUs. Estas abordan tanto soluciones *hardware* como la utilización de distintos lenguajes de programación.

Soluciones hardware: Históricamente AMD/ATI y NVIDIA han sido las compañías que más hincapié han hecho en el desarrollo de *hardware* gráfico, poniendo a disposición de la comunidad científica diferentes soluciones con el objetivo de acelerar los cálculos realizados en las CPUs. En la actualidad, ambas permiten el acceso al conjunto de instrucciones nativas y a la memoria de los elementos de cálculo masivamente paralelos de las GPUs, convirtiendo a estas en poderosas arquitecturas programables.

Para competir con CUDA de NVIDIA, en el año 2006 AMD presenta la biblioteca *Close to Metal* (CTM) para la programación de propósito general en sus unidades de procesamiento gráfico. CTM era una capa fina de abstracción que permitía el acceso directo al juego de instrucciones de las GPUs y su sistema de memoria. Debido a su escaso éxito, en 2008 AMD/ATI decidió terminar con su desarrollo y apoyar el proyecto de OpenCL. Ese mismo año presentó *ATI Stream*, un marco de desarrollo que permite trabajar sobre OpenCL como lenguaje de programación para utilizar las GPUs como procesadores de propósito general. Por su parte, NVIDIA ofrece, desde 2006, CUDA para un propósito similar, siendo este, además de un marco que permite el uso de OpenCL, un lenguaje específico para el procesado GPGPU sobre tarjetas NVIDIA.

Lenguajes: Hoy en día coexisten dos lenguajes principales para poder programar en GPUs, *CUDA* y *OpenCL*. *CUDA* es la solución propuesta por NVIDIA y que funciona solo en el *hardware* fabricado por esta compañía mientras que *OpenCL* busca ser un estándar que permita ejecutar código en plataformas heterogéneas y de fabricantes diferentes.

En un esfuerzo con el fin de facilitar las tareas a los programadores para que aprovechen las ventajas de la computación paralela, NVIDIA junto a Cray Inc, The Portland Group (PGI) y CAPS Enterprise anunciaron en 2011 un nuevo estándar de programación paralela denominado OpenACC[48]. Este nuevo estándar de programación para computación paralela es abierto y está diseñado para permitir a millones de científicos y programadores técnicos tomar ventaja, de manera fácil, del poder de cómputo de sistemas heterogéneos, como sistemas formados por GPU/CPU o GPGPU.

Por otra parte, OpenACC es totalmente compatible e interoperable con la arquitectura de programación paralela CUDA, que está diseñada para permitir un control detallado sobre el acelerador con el objetivo de ajustar el rendimiento máximo.

En las siguientes secciones se abordan estas soluciones en más detalle.

4.2. Paradigma de programación.

4.2.1. CUDA. Arquitectura y modelo de programación.

Las siglas *CUDA* hacen referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA que permiten a los programadores usar una serie de

[48] Página oficial de OpenACC: `http://www.openacc.org/`

lenguajes de programación, como C/C++, Fortran, Python o Java, para codificar algoritmos en GPUs de NVIDA.

El objetivo de CUDA es intentar explotar las ventajas de las GPU frente a las CPU haciendo uso del paralelismo que ofrecen sus múltiples núcleos. Así, si una aplicación está diseñada utilizando numerosos hilos que realizan tareas independientes, una GPU podrá ofrecer un gran rendimiento en diversos campos.

4.2.1.1. Arquitectura de una GPU.

El modelo de programación CUDA presenta la GPU como un coprocesador que puede ejecutar *kernels* en paralelo y ofrece extensiones para el lenguaje C para (a) mapear los datos de la GPU, (b) transferir datos entre la GPU y la CPU y (c) lanzar dichos *kernels*.

Un *kernel* CUDA ejecuta unas líneas de código sobre un gran número de hilos en paralelo. Este tipo de sistemas explotan el concepto SIMT (*Single Instruction Mutiple Threads*), en la que una misma instrucción es ejecutada por muchos hilos, o *threads*, con datos de entrada distintos. Las tareas son organizadas mediante bloques CUDA, donde se pueden llegar a lanzar hasta 1024 hilos que pueden cooperar entre sí por compartición de datos a través de una memoria local de baja latencia y la sincronización mediante barreras. Diferentes bloques CUDA sólo se pueden coordinar a través de los accesos a una memoria global de alta latencia.

La figura 24 muestra un esquema del *hardware* en una GPU. Los núcleos de la GPU (procesadores) se organizan en varios multiprocesadores. Cada uno de estos núcleos integra sus propias unidades funcionales y un registro de gran tamaño que tiene capacidad para la ejecución de cientos de hilos concurrentes. Cada multiprocesador posee una unidad de instrucción (*Instruction Unit*) que controla el lanzamiento de los hilos, y una memoria local compartida. La jerarquía de memoria también incluye memoria caché de sólo lectura para acelerar el acceso a las texturas y las constantes. La abstracción del bloque CUDA está estrechamente relacionada con esta organización: cada bloque de CUDA es ejecutado por un multiprocesador que, dependiendo de la disponibilidad de recursos, pueden mapear varios bloques al mismo tiempo.

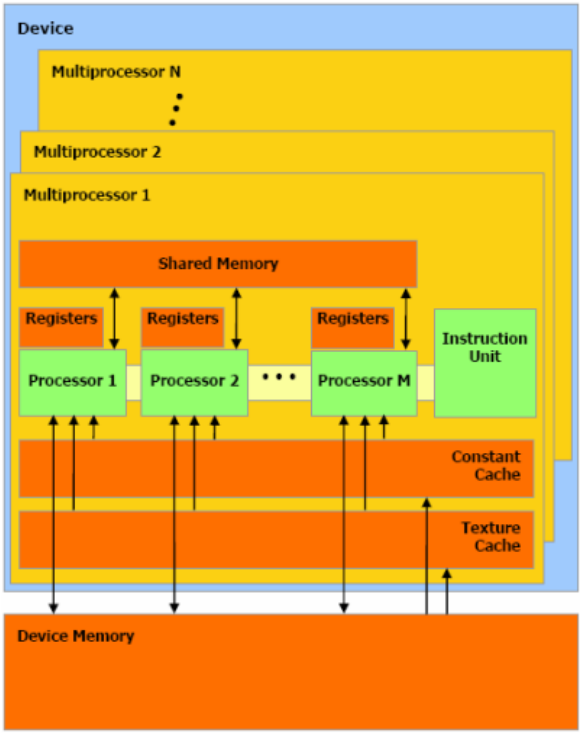


Figura 24: Modelo de programación CUDA. GPU como un coprocesador que integra varios multiprocesadores y una jerarquía de memoria compleja[49].

La unidad de ejecución no es el hilo individual, sino un grupo de hilos llamados *warp*. En cada ciclo el planificador elige el siguiente *warp* a ejecutar de forma semejante a la planificación de un procesador *multithreading* de grado fino.

Uno de lo factores más significativos que afectan al rendimiento final es el uso eficiente de la jerarquía de memoria. A pesar de que este tipo de hardware permite la ocultación de accesos a memoria de alta latencia al igual que un procesador *Multithreading* de grano fino, el acceso simultáneamente de un número alto de *threads* a la DRAM plantea un enorme desafío. Debido a este problema, el uso eficiente de la memoria local compartida y las texturas de sólo lectura son fundamentales para lograr buenos rendimientos en muchos algoritmos. Además, el acceso de los *threads* de un *warp* debe de ser alineado, porque se traduce en un único acceso a memoria reduciendo significativamente la contención con memoria [179].

4.2.1.2. El modelo de programación CUDA.

La programación paralela de propósito general en GPUs es un fenómeno relativamente nuevo. En sus orígenes, las GPUs eran bloques de *hardware* optimizados para un pequeño conjunto de operaciones con gráficos. Con el paso de los años, los programadores fueron reclamando una mayor flexibilidad para programar en este tipo de dispositivos. Es por ello por lo que en

[49] Figura extraída de la guía de programación de CUDA [177]

2006, NVIDIA introduce la arquitectura CUDA y las herramientas necesarias para realizar el cálculo en paralelo de datos en una GPU de una forma más sencilla.

El modelo de programación CUDA extiende el lenguaje de programación C/C++ permitiendo que el programador defina funciones en este lenguaje, llamadas *kernels*, que van a ser ejecutadas N veces en paralelo para los N hilos diferentes. El modelo CUDA asume que los hilos se ejecutan en una unidad física distinta al procesador (*host*), que actúa como coprocesador (dispositivo o *device*) donde se ejecuta el programa, como muestra la figura 25. En principio, los *kernels* se ejecutan secuencialmente en el coprocesador, aunque a partir de la aparición de la arquitectura *Fermi* también es posible la ejecución dinámica a nivel de kernel pudiendo coexistir su ejecución en el tiempo.

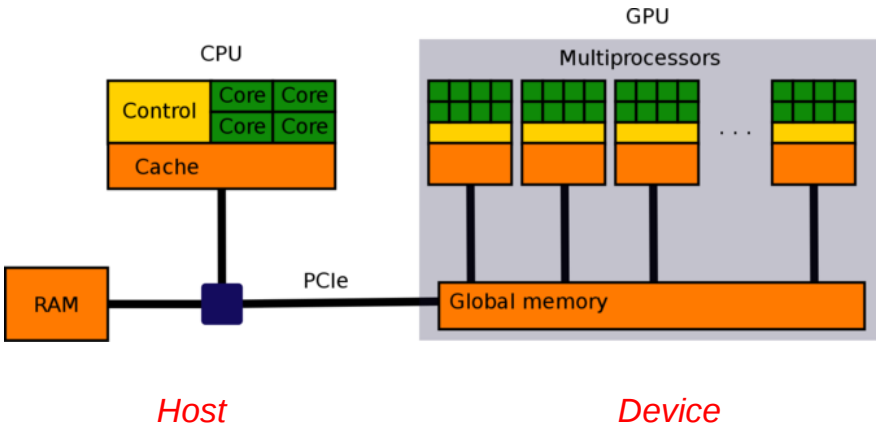


Figura 25: Arquitectura CPU (*host*) - GPU (*device*)[50].

A modo de ejemplo, se muestra parte de código para ilustrar cómo se define un *kernel* y cómo este es llamado desde un programa.

```
//definición del kernel
__global__ void vecAdd(float *A, float *B, float *C)
{
 int i = threadIdx.x;
```

[50]Figura extraída de la guía de programación de CUDA [177]

```
 C[i] = A[i] + B[i];
}

int main(){
 ...
 vecAdd<<<1,N>>>(A,B,C); //invocación del kernel con N hilos
 ...
}
```

Existe una jerarquía de hilos perfectamente definida en CUDA. Estos se agrupan en vectores, que reciben el nombre de bloques, pudiendo ser unidimensionales, bidimensionales o tridimensionales, de forma que definen bloques de hilos de una, dos o tres dimensiones según se quiera representar el espacio del problema. Sólo los hilos que pertenecen al mismo bloque pueden compartir datos y sincronizar sus ejecuciones, no permitiéndose la cooperación entre hilos de distintos bloques. A la organización de bloques se denomina *grid*, que a su vez puede ser de una o dos dimensiones. Como hemos visto en el código de ejemplo, los valores entre $<<<$ ..... $>>>$ sirven para configurar el *kernel* y definen la dimensión del *grid* y el número de hilos de cada bloque. La figura 26 ilustra la organización de los hilos en un *grid* de 2x3 bloques y 3x4 hilos. Puede observarse que cada hilo queda perfectamente definido por un identificador de bloque y el identificador del propio hilo dentro del bloque. Estos identificadores se suelen utilizan como índices para definir el conjunto de datos que procesa cada hilo. Si tomamos como ejemplo el código anterior, cada hilo tomará un valor del vector A, en la posición *i*, y otro en la misma posición del vector B y almacenará la suma en el vector C.

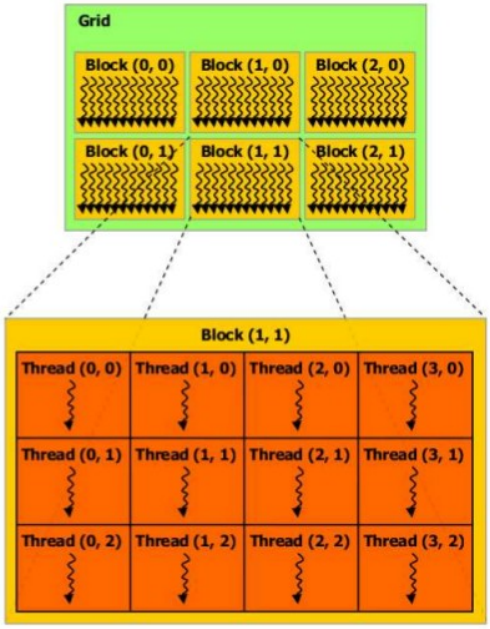


Figura 26: Jerarquía de hilos en CUDA[51].

En CUDA, existe una jerarquía de memoria con distintos espacios de memoria: local, compartida y global. Cada hilo tiene acceso a estos tipos; en primer lugar, a la memoria local, que se aloja en la memoria principal de la GPU (*off-chip*). Para que todos los hilos de un mismo bloque puedan colaborar entre si, estos tienen acceso a una región de memoria compartida (*on-chip*) que posibilita esta labor. Cabe destacar que la memoria compartida tiene el mismo tiempo de vida que el bloque de hilos. Para finalizar, todos los hilos tienen acceso a la misma memoria global (*device memory*). La figura 27 resume gráficamente la jerarquía de memoria de CUDA.

Además, existen dos espacios de memoria adicionales y accesibles por todos los hilos: la memoria de constantes y la de texturas. Junto con la memoria global, están optimizadas para diferentes usos de memoria y son persistentes a través de los lanzamientos de *kernels* de una misma aplicación. Adicionalmente, la memoria de texturas ofrece distintos tipos de direccionamiento, como filtro de datos, para algunos tipos de datos específicos.

También se puede considerar la memoria de la CPU como un nivel más externo de memoria, teniendo siempre en cuenta que las transferencias entre CPU-GPU suponen un cuello de botella, por lo que en muchos casos es de vital importancia minimizar estas transacciones.

[51] Figura extraída de la guía de programación de CUDA [177]
[52] Figura extraída de la guía de programación de CUDA [177]

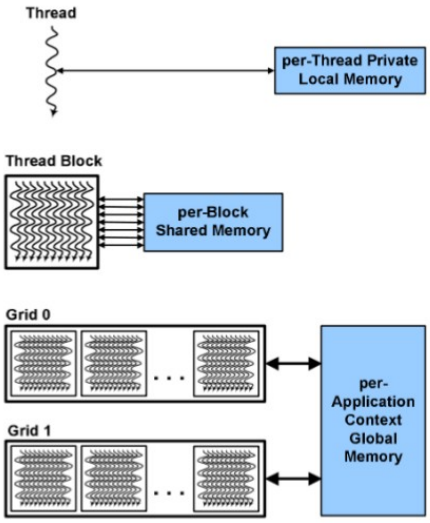


Figura 27: Jerarquía de memoria en CUDA[52].

4.2.1.3. El modelo de ejecución.

El planificador de hilos es el encargado de la ejecución de los hilos en la GPU, distribuyendo bloques de hilos entre los multiprocesadores. En las primeras versiones, cada multiprocesador puede ejecutar un total de ocho bloques simultáneamente. Además, el planificador ha de mantener una lista de los bloques planificados y debe encargarse de ir asignando bloques a los multiprocesadores según vayan terminando.

El multiprocesador tiene que crear, gestionar y ejecutar los hilos de forma concurrente en el hardware sin un sobrecoste relativo a la planificación. Para ello mapea cada hilo a un procesador o *CUDA core*, ejecutando cada uno de ellos de forma independiente con su propia dirección de instrucción y registros de estado. A este modelo de ejecución se lo conoce por el nombre de SIMT (Single Instruction Multiple Threads). Lo primero que hace el multiprocesador es dividir los bloques de hilos en grupos de 32 hilos, que reciben el nombre de *warps*, siendo esta la unidad de planificación. Cuando se tiene que lanzar una nueva instrucción, la unidad de planificación selecciona uno de estos *warps* que están disponibles y lanza la misma instrucción para todos los hilos que pertenecen al *warp* seleccionado. Hay que tener en cuenta que las instrucciones de salto pueden suponer un problema puesto que los hilos de un mismo *warp* pueden tomar caminos de ejecución distintos. Si se da este caso, la ejecución se realiza en serie, ejecutando primero los hilos de un camino y a continuación los hilos del otro. Debido a esto, existen instrucciones que permiten la sincronización de todos los hilos de un mismo *warp* (y no de bloques distintos, puesto que son independientes entre sí), haciendo que bloques enteros detengan su ejecución.

4.2.1.4. El modelo de memoria.

Las aplicaciones CUDA pueden acceder a diferentes tipos de memoria. Para cada tipo hay una serie de ventajas y desventajas que hay que tener en cuenta. Por ejemplo, la memoria global es un espacio de direcciones grande, pero la latencia de acceso es muy alta. Sin embargo, la memoria compartida tiene una latencia de acceso a memoria bastante baja, pero el espacio de direcciones es más pequeño si lo comparamos con la memoria global. Es importante conocer las diferencias entre estas memorias y cómo usar unas u otras decisiones que pueden afectar al rendimiento. Por ello, el escoger dónde almacenar los datos y cuándo, se convierte en una tarea muy relevante.

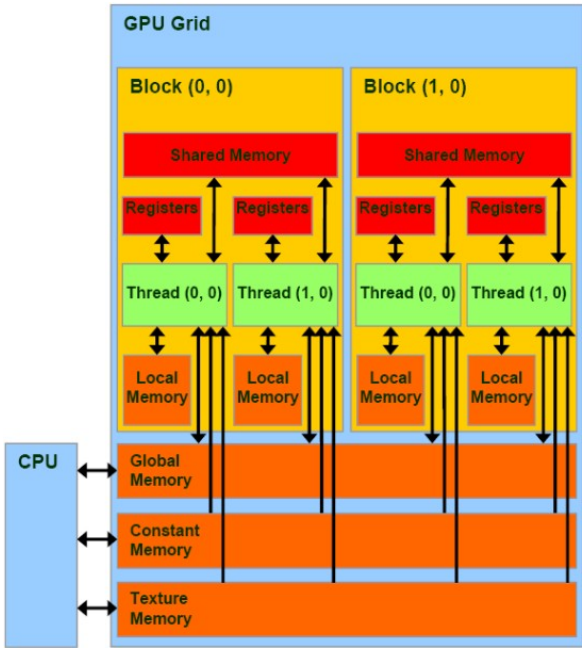


Figura 28: Modelo de memoria de CUDA[53].

Los distintos tipos de memoria disponible son: registros, memoria compartida, memoria local, memoria global y memoria de constantes.

En los dispositivos con la prestación de cómputo (o *compute capability* en la jerga CUDA) 1.x hay dos zonas donde la memoria puede residir, en la memoria caché o en la memoria del dispositivo (*device memory*). A la primera se la considera *on-chip* y los accesos a esta son muy rápidos. La memoria compartida y la de constantes se almacenan en la memoria *on-chip*. La memoria del dispositivo se considera *off-chip* y los accesos son mucho más lentos que a la primera. La memoria global y la local se almacenan en la memoria *off-chip*.

En los dispositivos con prestación de cómputo 2.x hay un banco de memoria adicional que se almacena en cada SP (*streaming processor*). Se la considera una caché L1 y su espacio de

[53]Figura extraída de la guía de programación de CUDA [177]

memoria es relativamente pequeño y su latencia de acceso muy lenta. La figura 28 resume el modelo de memoria de CUDA donde se dispone de:

- **Registros**: las variables escalares que se declaran en el ámbito de una función *kernel* y no están iniciadas con todos los atributos se almacenan por defecto en los registros. Estos registros tienen una latencia de acceso muy rápida, pero el número de registros que están disponibles por bloque es limitado. Las matrices que se declaran en la función del *kernel* también se almacenan en los registros, pero sólo si el acceso a los elementos de la matriz se realiza mediante índices constantes (es decir, el índice que se utiliza para acceder a un elemento de la matriz no es una variable, y por lo tanto el índice puede ser determinado en tiempo de compilación). Actualmente no es posible realizar acceso aleatorio para registrar variables. Las variables en esta memoria son privadas para el hilo. Los hilos que pertenecen al mismo bloque tendrán versiones particulares de cada variable de registro. Esta memoria sólo existe mientras el hilo existe. Una vez que el hilo finaliza la ejecución, ya no se puede acceder de nuevo a una variable de registro de dicho hilo. Cada vez que se invoca la función *kernel* se debe inicializar la variable. Esto puede parecer obvio, porque el alcance de la variable está dentro de la función *kernel*, pero no es cierto para todas las variables declaradas en la función *kernel*, como veremos con la memoria compartida. Las variables declaradas en los registros pueden ser leídas y escritas dentro del *kernel*; además, la lectura y la escritura no necesitan ser sincronizadas.

- **Memoria local**: cualquier variable que no cabe en el espacio de registros permitidos para el *kernel* se almacena en la memoria local. La memoria local tiene el mismo acceso que la latencia de memoria global (es decir, lenta). Los accesos a la memoria local se almacenan sólo en una memoria cercana (rápida) a la GPU en aquellas capacidad de cómputo 2.x o superior [180]. Al igual que los registros, la memoria local es privada para cada hilo, y por lo tanto debe inicializarse antes de que se utilice. Las variables almacenadas en la memoria local tienen la vida útil del hilo. Una vez que el hilo se haya ejecutado, la variable local ya no es accesible. No se puede realizar una declaración de variable con cualquier atributo, pero el compilador pondrá automáticamente las declaraciones de variables en la memoria local en las siguientes condiciones:
  - Las matrices que son accedidas con índices en tiempo de ejecución. Esto es, el compilador no puede determinar los índices en tiempo de compilación.
  - Grandes estructuras o matrices que consumen demasiado espacio al registrarse.
  - Cualquier variable declarada que supera el número de registros para ese *kernel*.

  Una variable en la memoria local puede ser leída y escrita en el *kernel* y el acceso a la memoria local no necesita ser sincronizado al ser empleada en el ámbito de un hilo.

- **Memoria compartida**: acceder a la memoria compartida es muy rápido (unas 100 veces más que a la memoria global) aunque cada SP tiene una cantidad limitada de espacio de dirección de memoria compartida. La memoria compartida se debe declarar dentro del alcance de la función del *kernel* teniendo el tiempo de vida del bloque. Cuando un bloque termina la ejecución, la memoria compartida que se definió en el

núcleo ya no es accesible. La memoria compartida puede ser tanto de lectura como de escritura dentro del *kernel*. Si se modifica el contenido de la memoria compartida es recomendable realizar una sincronización para cerciorarse de que el resto de hilos leen el valor correcto. Dado que el acceso a la memoria compartida es más rápido que el acceso a la memoria global, es más eficiente realizar una copia de los datos desde la memoria global a la memoria compartida que se utilizará dentro del *kernel* si existe una alta tasa de localidad. Para declarar de una variable en memoria compartida se antepone a esa variable la palabra `__shared__`.

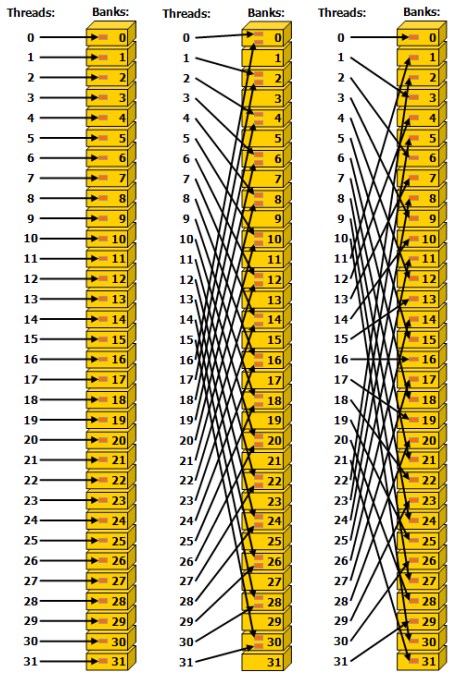


Figura 29: Patrones de acceso que no provocan conflicto. Acceso lineal de los hilos a palabras de 32 bits, con un avance de uno, dos y tres respectivamente[54].

En la figura 29 se muestran ejemplos de accesos que no provocan conflictos ya que cada hilo accede a un banco distinto. También cabe destacar que la memoria compartida implementa un mecanismo de distribución por el cual varios hilos pueden leer una palabra de 32 bits de forma simultánea en la misma petición de lectura. Gracias a esto se reduce el número de conflictos sobre un banco al que varios hilos piden el mismo dato. Por otro lado, la figura 30 muestra algunos ejemplos de accesos irregulares a memoria compartida que no provocan conflictos.

- **Memoria global**: la latencia de acceso a memoria global es muy alta (unas 100 veces más lenta que la memoria compartida), pero su espacio de direccionamiento es mucho

[54] Figura extraída de la guía de programación de CUDA [177]
[55] Figura extraída de la guía de programación de CUDA [177]

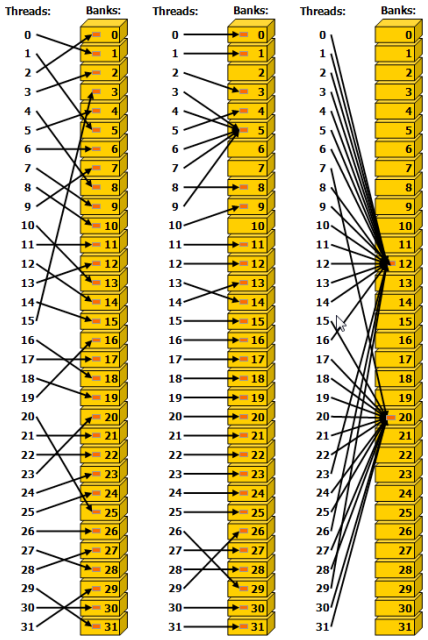


Figura 30: Patrones de acceso irregular a memoria a palabras de 32 bits. Acceso por permutación aleatoria, acceso de los hilos 3, 4, 6, 7 y 9 a la misma palabra del banco 5 e hilos que acceden a la misma palabra de un banco respectivamente[55].

mayor (hasta 6 GB en las modernas NVIDIA Tesla K20X). A diferencia de las memorias vistas hasta ahora, la memoria global se puede leer y escribir desde el *host* usando la función de C `cudaMemcpy`. Además, esta memoria tiene el mismo tiempo de vida que la aplicación y es accesible para todos los hilos de todos los *kernels*. Hay que tener cuidado al leer o escribir en la memoria global debido a que la ejecución de subprocesos no se pueden sincronizar entre diferentes bloques. La memoria global se declara en el *host* utilizando la función de C `cudaMalloc` y se libera mediante `cudaFree`.

- **Memoria de constantes**: al igual que las variables globales, las variables constantes se deben declarar en el ámbito global (fuera del alcance de cualquier función *kernel*). Comparten los mismos bancos de memoria que la memoria global, pero a diferencia de esta, sólo se puede declarar una cantidad pequeña de este tipo de memoria (64KB). La latencia de acceso a esta memoria es considerablemente más rápida que a la memoria global, pero a diferencia de la memoria global, la de constantes sólo es de lectura dentro del *kernel*. Se puede utilizar en el *host* mediante las funciones `cudaMemcpyToSymbol` (para escribir en la memoria) y `cudaMemcpyFromSymbol` (para leer de la memoria). No es posible asignar dinámicamente el almacenamiento de la memoria constante (el tamaño de los buffers de memoria de constantes debe ser declarado y determinado estáticamente en tiempo de compilación).

- **Memoria de texturas**: realizar las lecturas a través de la memoria de texturas puede

tener algunos beneficios que la conviertan en una alternativa mejor a la memoria global o de constantes:

- Si las lecturas no se ajustan a los patrones de acceso a la memoria global o a la de constantes, es posible obtener mayor ancho de banda explotando las ventajas de localidad en la memoria de texturas.
- Los enteros de 8 y 16 bits se pueden convertir a punto flotante de 32 bits (en rangos de [0.0, 1.0] o [-1.0, 1.0]).
- Los datos pueden distribuirse a través de variables separadas en una única instrucción.
- La latencia provocada por el cálculo de direcciones se oculta mejor y además puede que mejore el rendimiento de las aplicaciones que acceden a los datos de forma aleatoria.

La tabla 6 resume los distintos tipos de memoria y sus propiedades.

Memoria	Localización	Cacheada	Acceso	Alcance	Tiempo de vida
Registros	caché	n/a	Host:NO Kernel:R/W	hilo	hilo
Local	device	1.x:NO 2.x:SI	Host:NO Kernel:R/W	hilo	hilo
Compartida	caché	n/a	Host:NO Kernel:R/W	bloque	bloque
Global	device	1.x:NO 2.x:SI	Host:R/W Kernel:R/W	aplicación	aplicación
Constantes	device	SI	Host:R/W Kernel:R	aplicación	aplicación

Tabla 6: Los distintos tipos de memoria en CUDA y sus características.

4.2.1.5. Fermi y Kepler.

Actualmente, la demanda de cómputo paralelo de gran rendimiento ha provocado que el uso de GPU como aceleradores de cálculo aumente en muchas áreas científicas, como son la medicina, las finanzas, la ingeniería, etc. Es por ello por lo que NVIDIA ha revisado y redefinido las GPU dando como resultado las nuevas tecnologías gráficas. Cada familia nueva tiene diferentes características y capacidades de cómputo. Las familias existentes para computación de altas prestaciones son Tesla (2008), Fermi (2010) y Kepler (2012), y se prevé el lanzamiento de Maxwell (2014) con memoria virtual unificada y Volta (*sin fecha aún*), con módulos de memoria apilados denominados "stacked DRAM" para alcanzar anchos de banda de hasta 1TB/s.

4.2.1.5.1. Arquitectura Fermi.

La primera GPU basada en *Fermi*, implementada con 3 billones de transistores, ofrece hasta 512 *cores* CUDA, donde cada uno de estos ejecuta una instrucción de punto flotante o entero por ciclo de reloj y por hilo. Los 512 *cores* CUDA están organizados en 16 multiprocesadores paralelos (SMs), que se muestran en la figura 31, de 32 *cores* cada uno de ellos. La GPU cuenta con seis particiones de memoria de 64-bit, una interfaz de memoria de 384 bits, que soporta hasta un total de 6 GB de memoria DRAM de tipo GDDR5, una interfaz PCI-Express que conecta la GPU con la CPU y utiliza GigaThread y un programador general que distribuye hilos de bloques a los hilos programados del SM.

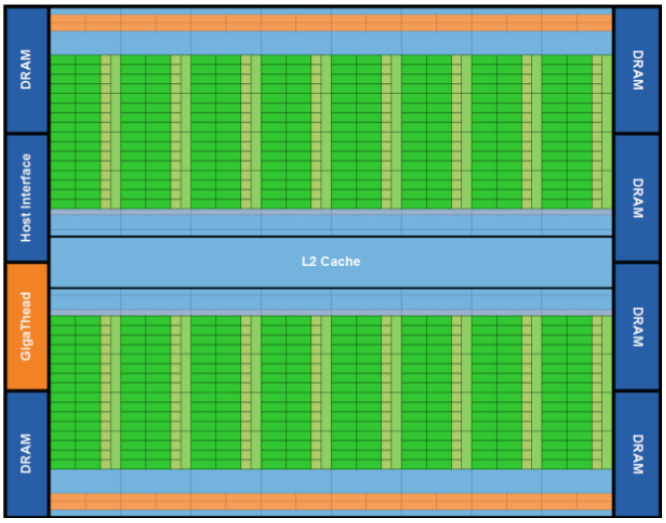


Figura 31: Los 16 SM de Fermi se posicionan alrededor de una caché L2 común. Cada SM es una banda vertical rectangular que contiene una porción naranja (planificador y envío), una porción verde (unidades de ejecución) y porciones azul claro (fichero de registro y caché L1)[56].

En la arquitectura *Fermi*, NVIDIA introduce lo que llaman la tercera generación de SMs, con una serie de innovaciones en la arquitectura que no sólo hacen que sean más potentes sino que también son más eficientes y fáciles de programar.

Cada SM (figura 32) posee 32 procesadores CUDA, un incremento cuatro veces superior con respecto al diseño de anteriores SM. Cada uno de estos procesadores posee una unidad aritmético-lógica (ALU) y una unidad de punto flotante (FPU). Mientras que las GPUs anteriores usaban la aritmética de punto flotante IEEE 754-1985, *Fermi* implementa el nuevo estándar de punto flotante IEE 754-2008. Otras características de la nueva arquitectura de los SM son:

- Poseen 16 unidades de carga/almacenamiento, permitiendo que las direcciones fuente y destino puedan ser calculadas por hasta 16 hilos en cada ciclo.
- Tienen cuatro Unidades de Función Especial (SFUs) que ejecutan instrucciones como `sin`, `cos`, o `sqrt`.
- Están diseñados específicamente para aritmética de doble precisión, ofreciendo mejores rendimientos que en tarjetas anteriores [181].

Las GPU *Fermi* poseen un planificador dual de *warp* (**Dual Warp Scheduler**) que permite seleccionar dos *warps* a los que se les asocia una instrucción a cada uno para un grupo de 16 *cores*, 16 unidades de carga/almacenamiento o 4 SFUs. Como los *warps* se ejecutan de forma independiente, el planificador no necesita comprobar las dependencias de cada

[56] Figura extraída del documento de NVIDIA [181].
[57] Figura extraída del documento de NVIDIA [181].

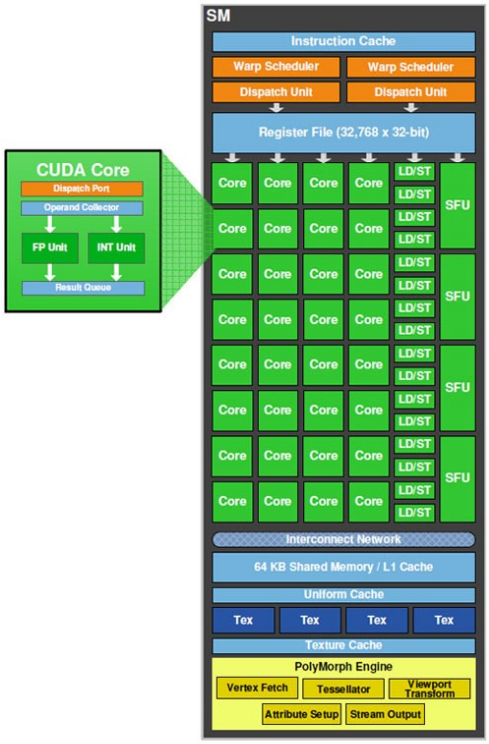


Figura 32: Arquitectura de un SM (Streaming Multiprocessor) en Fermi[57].

flujo de instrucción. De esta forma, se pueden emitir en dual más instrucciones, como dos instrucciones con enteros, dos instrucciones en punto flotante o una mezcla de instrucciones de entero, punto flotante, carga, almacenamiento y SFU. Las instrucciones de doble precisión no soportan una emisión dual con ninguna otra operación. La figura 33 muestra un esquema del planificador dual.

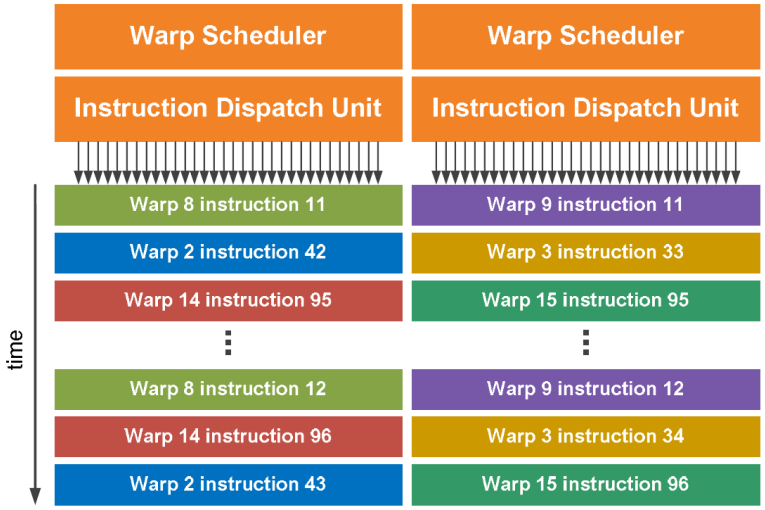


Figura 33: Planificador Dual de *Warps* (Dual Warp Scheduler)[58].

Sin embargo, una de las claves que han hecho que el rendimiento y la programabilidad de aplicaciones en GPU haya mejorado es tener una memoria compartida *on-chip*. Mientras que en tarjetas anteriores solo existían 16KB de memoria compartida para cada SM, en arquitecturas *Fermi* cada SM posee 64KB de memoria *on-chip* que se pueden configurar y dividir para tener una memoria compartida de 48KB y una caché L1 de 16KB o viceversa. La tabla 7 compara tarjetas anteriores con la tecnología *Fermi*.

GPU	G80	GT200	Fermi
Transistors	681 millon	1.4 billon	3.0 billon
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops/clock	246 FMA ops/clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops/clock	512 MAD ops/clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16KB	16KB	Configurable 48KB or 16KB
L1 Cache (per SM)	None	None	Configurable 48KB or 16KB
L2 Cache	None	None	768KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Tabla 7: Resumen de las distintas características de las arquitecturas G80, GT200 y Fermi.

Además, la arquitectura *Fermi* soporta la ejecución concurrente de *kernels*, es decir, distintos *kernels* de la misma aplicación se pueden ejecutar al mismo tiempo, como muestra la

[58]Figura extraída del documento de NVIDIA [181].

figura 34. Esto permite a los programas que ejecutan un número pequeño de *kernels* utilizar la GPU entera. Sin embargo, *kernels* que pertenecen a aplicaciones distintas pueden seguir ejecutándose secuencialmente con una buena eficiencia, gracias al mejorado cambio de contexto.

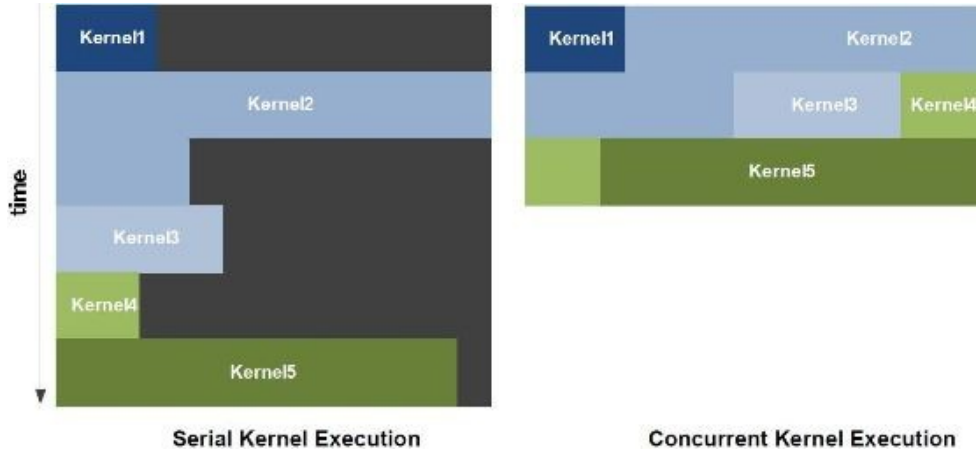


Figura 34: Ejecución de kernels en serie (izquierda) y en paralelo (derecha)[59].

Conclusiones:
Durante dieciséis años, NVIDIA se ha dedicado a la construcción de los procesadores gráficos más rápidos del mundo. Mientras las tarjetas G80 fueron pioneras en la arquitectura de computación GPU y las GT200 un refinamiento de las anteriores, sus diseños estaban profundamente arraigados en el mundo de los gráficos. La aparición de la arquitectura *Fermi* representa una nueva dirección para NVIDIA. Lejos de ser simplemente el sucesor de las GT200, *Fermi* es el resultado de un replanteamiento radical de la función, el propósito y la capacidad de las GPUs.

En lugar de tomar la ruta fácil de añadir unidades de ejecución, el equipo de *Fermi* ha abordado algunos de los problemas más difíciles de la computación GPU. La importancia de la localidad de datos se reconoce a través de los dos niveles de jerarquía de caché y su *path* de memoria combinada de carga/almacenamiento. El rendimiento en doble precisión se eleva a niveles de supercomputación, haciendo que la ejecución de operaciones atómicas sea hasta veinte veces más rápido. Sin embargo, una nueva revisión de la arquitectura de las GPU por parte de NVIDIA ha dado lugar a las arquitecturas *Kepler*.

4.2.1.5.2. Arquitectura Kepler.

Las tarjetas con arquitectura *Kepler* GK110 [38] aumentan el número de transistores hasta los 7.1 billones, haciendo que sean, además de las tarjetas más rápidas, las arquitecturas más

[59]Figura extraída del documento de NVIDIA [181].

Figura 35: Diagrama de la arquitectura Kepler GK110.

completas jamás construidas (figura 35). Dependiendo de la versión, poseen entre 7 y 15 multiprocesadores SMX dotados de 192 *cores* cada uno. *Kepler* añade algunas características innovadoras relacionadas con el rendimiento en el cálculo. Proporcionan un *throughput* de doble precisión de más de 1 TeraFlop (en formato IEEE-754 de 64 bits) con una eficiencia DGEMM superior al 80 % frente al 60-65 % que proporcionaban las *Fermi*. Si por ejemplo, se dispusieran 10 *racks* de servidores, se podrían alcanzar 1 PetaFLOP. Además, la arquitectura *Kepler* ofrece un gran salto hacia adelante en lo que a eficiencia energética se refiere, mejorando el rendimiento por vatio en 3x con respecto a su antecesora, la *Fermi*.

Aunque todos los modelos de *Fermi* y *Kepler* incorporan corrección de errores ECC en DRAM, anchura de 64 bits en el bus de direcciones y anchura de 64 bits en el bus de datos por cada controlador (6 controladores para 384 bits, salvo la versión GF104 que tiene 4), si comparamos la memoria y el transporte de datos vemos que la memoria integrada en cada SMX respecto a los multiprocesadores de Fermi duplica el tamaño y el ancho de banda del banco de registros, el ancho de banda de la memoria compartida y el tamaño y ancho de banda de la memoria caché L1. Además, la memoria interna (caché L2) es de 1.5 Mbytes y la externa (DRAM) es de tipo GDDR5 y anchura de 384 bits (aunque la frecuencia y el tamaño dependen de la tarjeta gráfica). Con respecto a la interfaz con el *host*, la arquitectura *Kepler* implementa la versión 3.0 de PCI-Express y diálogos más directos entre la memoria de vídeo de varias GPUs. La figura 36 muestra la diferencia en la jerarquía de memoria entre *Fermi* y *Kepler*.

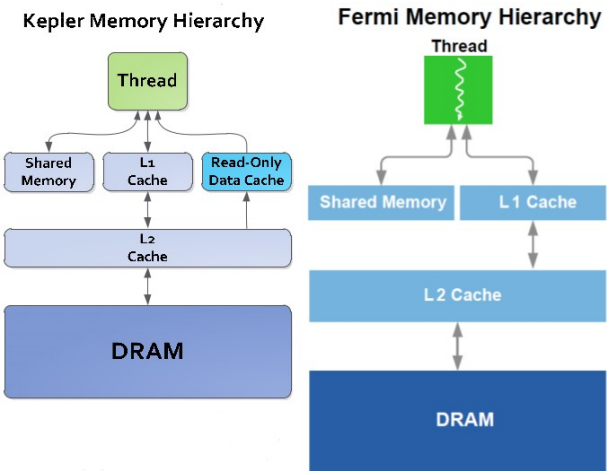


Figura 36: Diferencia en la jerarquía de memoria: Kepler vs. Fermi.

Las nuevas características permiten un incremento en la utilización de la GPU, simplifican los diseños de los programas paralelos y ayudan en el despliegue de las GPU. Algunas de estas características son:

- **Paralelismo Dinámico**. Simplifica la programación en la GPU, ya que facilita la aceleración de bucles anidados paralelos, lo que significa que una GPU puede iniciar nuevos subprocesos de forma dinámica por sí misma, sin necesidad de volver a la CPU. En la figura 37 se puede observar el paralelismo dinámico introducido en la arquitectura *Kepler*. Mientras que en versiones anteriores sólo la CPU podía generar trabajo en la GPU, en la arquitectura *Kepler* la GPU puede generar trabajo por si sola de forma simultánea, dinámica e independiente.

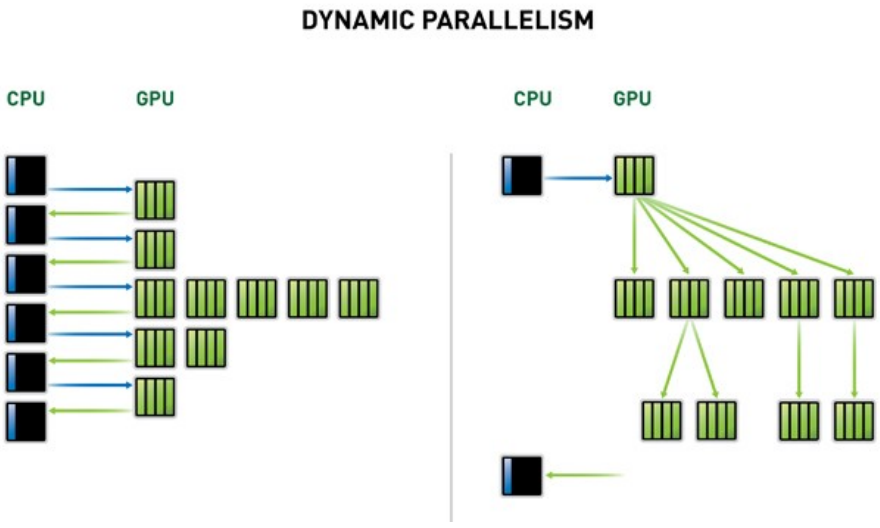


Figura 37: Diferencia entre versiones anteriores y Kepler con respecto al paralelismo dinámico.

- **Hyper-Q**. Reduce el tiempo de inactividad de la CPU al permitir que múltiples núcleos de ésta utilicen una misma GPU *Kepler*, lo que mejora drásticamente su capacidad para ser programada y la eficiencia. Se pueden ejecutar hasta 32 *kernels* procedentes de varios procesos de CPU de forma simultánea, lo que incrementa el porcentaje de ocupación temporal de la GPU. La diferencia entre *Fermi* y *Kepler* en este sentido se muestra en la figura 38.

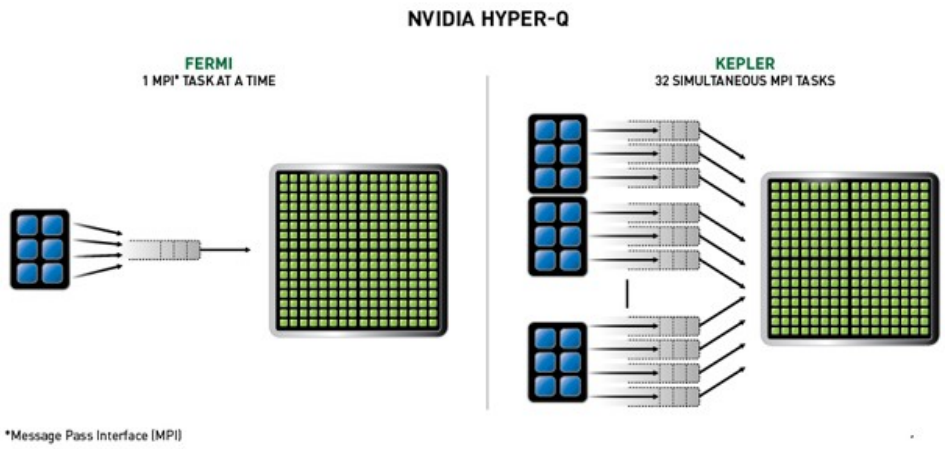


Figura 38: Diferencia entre Fermi y Kepler con respecto a la introducción de Hyper-Q.

- **SMX**. Proporciona mayor velocidad de procesamiento y eficiencia energética gracias a su innovador multiprocesador de streaming (SM), que permite dedicar más espacio a los núcleos de procesamiento que a la lógica de control. La figura 39 muestra un esquema comparativo de los SM de una *Fermi* con respecto a los SMX de una *Kepler*.

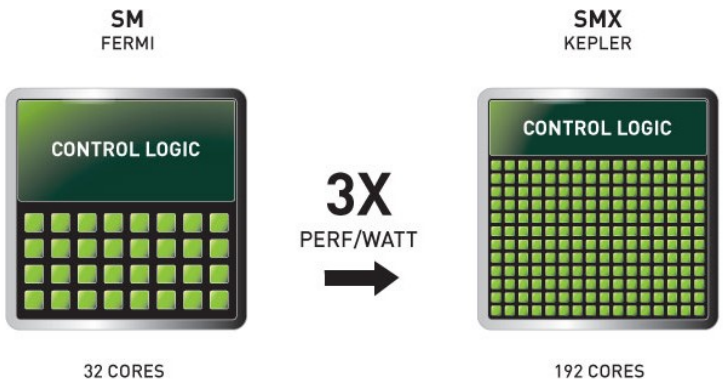


Figura 39: SM en Fermi (izquierda) y SMX en Kepler (derecha).

- **GPUDirect**. Soporta RDMA (*Remote Direct Memory Access*), lo que permite transferencias directas entre GPUs y dispositivos de red y degrada menos el ancho de banda de la memoria de vídeo GDDR5.

Conclusiones:
Con el lanzamiento de *Fermi* en 2010, NVIDIA introdujo una nueva era en la computación de alto rendimiento (HPC) de la industria basada en un modelo de computación híbrida, donde CPUs y GPUs trabajan juntas, para resolver cargas de trabajo de computación intensiva. Ahora, con la nueva GPU Kepler GK110, NVIDIA una vez más sube el listón para la industria de HPC.

Kepler GK110 fue diseñado desde el principio para maximizar el rendimiento computacional con una eficiencia energética excepcional. La arquitectura tiene muchas innovaciones como SMX, paralelismo dinámico e Hyper-Q que hacen la computación híbrida mucho más rápida, más fácil de programar y adaptable a un conjunto más amplio de aplicaciones.

4.2.2. Breve introducción a OpenCL.

Open Computing Language [176] (OpenCL) es un lenguaje de programación para crear aplicaciones y tareas con paralelismo de datos y que se pueden ejecutar tanto en unidades de procesamiento gráfico como en unidades centrales de procesamiento. La especificación fue creada originalmente por Apple y desarrollada en conjunto con AMD, IBM, Intel y NVIDIA[60]. La principal motivación para la aparición de OpenCL fue la necesidad de tener una plataforma de desarrollo estándar para las distintas plataformas de computación paralela existentes. Al igual que para la programación paralela en CPUs existen estándares como OpenMP, se ha observado que es necesario buscar uno que pueda usarse en plataformas heterogéneas.

El modelo de paralelismo de datos en OpenCL es similar al de CUDA. Un programa escrito en OpenCL tiene dos partes: los *kernels* que se ejecutan en uno o varios dispositivos, y el programa *host* que gestiona la ejecución de los *kernels*. Para tener una idea de la similitud de conceptos, la tabla 8 muestra la relación de algunos conceptos de OpenCL y los correspondientes en CUDA. Como puede observarse, algunos son idénticos mientras que otros se denominan de forma diferente.

OpenCL	CUDA
Kernel	*Kernel*
Work item	*Hilo*
Work group	*Bloque*
NDRange	*Grid*
Host	*Host*

Tabla 8: Conceptos de OpenCL y CUDA.

[60] Lista completa de compañías que conforman el *working group* de OpenCL: `http://www.khronos.org/conformance/adopters/conformant-companies`

Existen también similitudes con respecto a la arquitectura, puesto que al igual que CUDA, el sistema posee un *host*, la CPU, y uno o varios dispositivos OpenCL. La figura 40 muestra, de manera conceptual, un dispositivo OpenCL, en el que se puede observar la presencia de las unidades de cómputo (CUs, *Compute Units*) que corresponden en CUDA a los multiprocesadores SM (en azul oscuro), y los elementos de procesamiento (PEs, *Processing Elements*), que tienen su homólogo en los SPs de CUDA (donde se encuentran los *work-items*), donde se realizan los cálculos. El concepto de memoria global y de constantes es similar al de CUDA. Sin embargo, la memoria local tendría su correspondencia en CUDA con la memoria compartida y la memoria privada de OpenCL es la memoria local de CUDA.

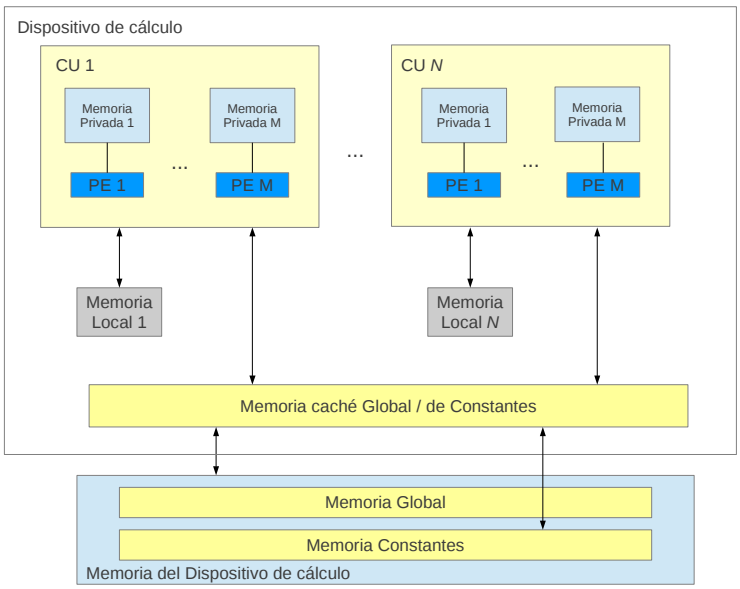


Figura 40: Esquema de la arquitectura OpenCL.

Una aplicación OpenCL se ejecuta en un *host* de acuerdo con los modelos nativos a la plataforma de dicho *host*. La aplicación OpenCL envía comandos desde el *host* para realizar los cálculos en los PEs dentro de un dispositivo OpenCL. Los PEs dentro de una CU ejecutan una sola corriente de instrucciones SIMD, o como unidades SPMD (*Single Program Multiple Data*) donde cada PE mantiene su propio contador de programa. En la figura 41 puede observarse el modelo de la plataforma OpenCL, con el *host* y N dispositivos OpenCL, cada uno de ellos con sus CUs y sus PEs.

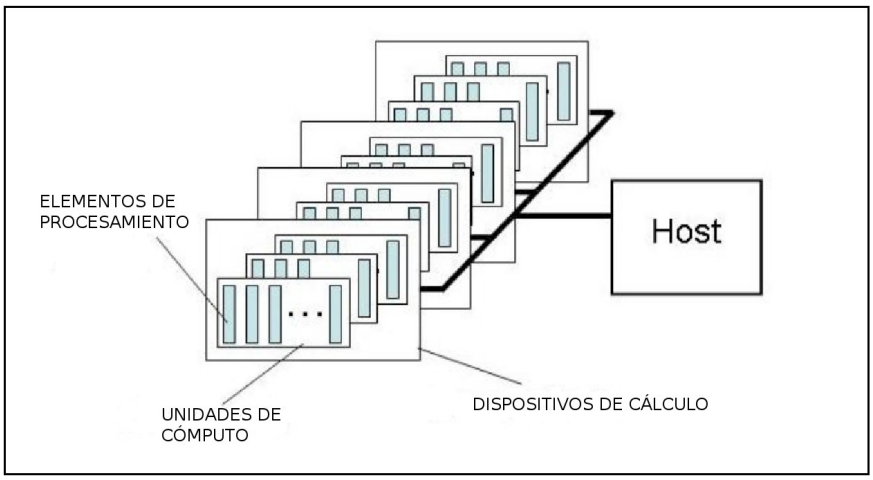


Figura 41: Modelo de plataforma en OpenCL, con un *host* y varios dispositivos de cálculo.

Con el objetivo de describir las ideas principales que están detrás de OpenCL, a continuación se exponen los diferentes modelos que lo componen: el modelo de ejecución, el de programación y el de memoria.

4.2.2.1. Modelo de ejecución en OpenCL.

Como OpenCL no sólo está dirigido a GPUs sino también a otros aceleradores, como CPUs multinúcleo, se proporciona flexibilidad en el tipo de *kernel* que se especifica. Estos pueden idearse para tareas en paralelo, pensadas para CPUs, o para datos en paralelo, lo que coincide con la arquitectura de GPUs.
El modelo de ejecución está compuesto por dos componentes: un programa *host*, que ejecuta el programa central que gestiona el resto de dispositivos, y los *kernels*, ejecutados en los dispositivos OpenCL.

Los *kernels* son porciones de código y que pueden encontrar su similitud con las funciones C. La ejecución de estos puede realizarse en orden o no dependiendo de los parámetros pasados al sistema cuando se forma la cola con el *kernel* para la ejecución. Los eventos se proporcionan de manera que el desarrollador puede comprobar el estado de las solicitudes de ejecución de *kernel* y otras solicitudes de rutina.

El núcleo del modelo de ejecución de OpenCL se define por cómo se ejecutan los *kernels*. Cuando el *host* suministra uno de ellos, se define un espacio de índices. Una instancia del *kernel* se ejecuta para cada punto de este espacio. A esta instancia del *kernel* se le conoce con el nombre de *work-item* y se define por su punto en el espacio de índices, que proporciona un ID global a cada *work-item*. Cada uno de ellos ejecutan el mismo código, pero con una ruta de ejecución específica, pudiendo variar los datos para cada *work-item*.

Los *work-items* se organizan en *work-groups*, que proporcionan una descomposición del espacio de índices de grano más grueso. A cada *work-group* se le asigna un ID único con la misma dimensionalidad que usan los *work-items* que lo componen. A cada *work-item* se le asigna un identificador local único dentro de un *work-group* de forma que podría ser identificado bien por su ID global, bien por la combinación de su ID local y el ID de su *work-group*. Los *work-items* de un determinado *work-group* se ejecutan de manera concurrente en los PEs de una CU.

Al espacio de índices en OpenCL se le conoce con el nombre de *NDRange*, que es un espacio N-dimensional donde *N* es uno, dos o tres. Un *NDRange* se define por un array de enteros de longitud *N* indicando el alcance del espacio de índice en cada dimensión a partir de un índice *F* dado (normalmente 0).

En términos de organización, el dominio de la ejecución de un *kernel* se define por un dominio de dimensión *N*. Esto permite que el sistema conozca la magnitud del problema que el usuario quiere que el *kernel* aplique. Cada elemento en el dominio de ejecución es un elemento de trabajo y OpenCL proporciona la capacidad de agrupar los elementos del trabajo en grupos para propósitos de sincronización y comunicación.

El programa *host* se puede escribir en cualquier lenguaje soportado por el sistema, aunque los más utilizados son C/C++.

4.2.2.2. Modelo de programación en OpenCL.

El modelo de programación de OpenCL define varios tipos de paralelismo, de tareas, de datos y los que combinan a ambos, siendo el paralelismo de datos el fundamento principal de OpenCL.

- **Paralelismo de tareas**. El modelo de programación de OpenCL con paralelismo de tareas define un modelo en el que una única instancia de un *kernel* se ejecuta de manera independiente de su número de índice. Es equivalente a ejecutar un *kernel* en una CPU con un *work-group* que contiene un solo *work-item*. Con este tipo de modelo los usuarios pueden expresar paralelismo usando tipos de vectores de datos implementados en el dispositivo, encolando múltiples tareas o *kernels* nativos desarrollados mediante la utilización de un modelo de programación ortogonal en OpenCL [182].
- **Paralelismo de datos**. El modelo de programación con paralelismo de datos define un cálculo en términos de una secuencia de instrucciones aplicado a múltiples elementos de un objeto de memoria. El espacio de índices asociado con el modelo de ejecución de OpenCL define los *work-items* y como los datos se mapean en ellos. En un modelo de paralelismo de datos más estricto existe una correspondencia uno a uno entre el *work-item* y el elemento en un objeto de memoria sobre el que un *kernel* puede ejecutarse en paralelo. Sin embargo, en OpenCL se implementa una versión más relajada donde esa correspondencia uno a uno no es necesaria.

OpenCL proporciona un modelo jerárquico para el paralelismo de datos donde existen dos formas de especificar la subdivisión. En el *modelo explícito* un programador define el número total de *work-items* a ejecutar en paralelo y puede describir como estos se distribuyen entre los *work-groups*. En el *modelo implícito*, el programador sólo especifica el número de *work-items* que se ejecutarán en paralelo dejando a la implementación OpenCL la división de estos en los correspondientes *work-groups*.

Una característica del modelo de programación con paralelismo de datos es que múltiples *kernels* pueden acceder a una misma región de memoria.

Cuando se utiliza el modelo de paralelismo de datos, se debe declarar la consistencia de memoria de manera explícita haciendo uso de dominios de sincronismo. En OpenCL existen dos: mediante cola de comandos, que permite administrar el paralelismo entre diferentes dispositivos, y mediante *work-items* en un *work-group*, controlando así la ejecución de *kernels* en un mismo dispositivo. La figura 42 ilustra la organización de los hilos en OpenCL pudiendo observarse similitudes con la de CUDA, donde el *NDRange* equivaldría al *Grid* de CUDA, el *Work Group* al bloque CUDA y los *Work Items* a los hilos CUDA.

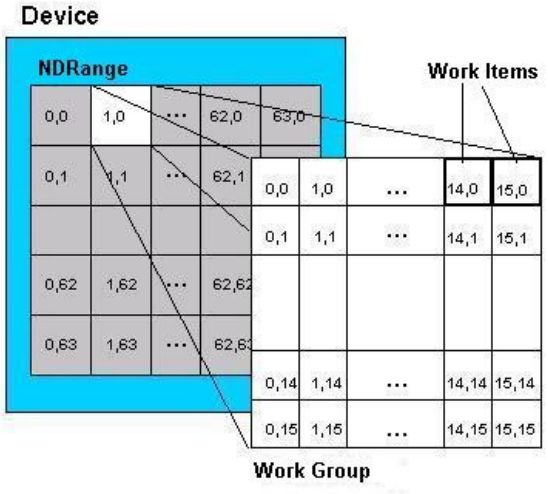


Figura 42: Jerarquía de hilos en OpenCL.

A continuación se muestra, a modo de ejemplo, una porción de código que sirve para ilustrar cómo se define un *kernel* en OpenCL y cómo se llama desde el programa *host*.

```
//definición del kernel
__kernel void vecAdd(__global const float *a,
                     __global const float *b,
                     __global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a [gid] + b[gid];
}
```

```
//se ejecuta el kernel en el host
int main()
{
   ...
   //llamada al kernel
   kernel = clCreateKernel(program, "vecAdd");
   ...
}
```

4.2.2.3. Modelo de memoria en OpenCL.

Los *work-items* que se ejecutan en un *kernel* tienen acceso a cuatro regiones de memoria distintas.

- **Memoria global**: esta región de memoria permite el acceso en modo lectura/escritura a todos los *work-items* de todos los *work-groups*. Los *work-items* pueden leer y escribir cualquier elemento en esta región de memoria aunque, dependiendo de las capacidades del dispositivo, estas escrituras y lecturas pueden necesitar ser cacheadas.
- **Memoria de constantes**: es una región de memoria global que se mantiene constante durante la ejecución de un *kernel*. El *host* asigna e inicializa objetos en esta región de memoria.
- **Memoria local**: es una región de memoria para los *work-group*. Esta región de memoria puede usarse para asignar variables que se comparten entre *work-items* de un mismo *work-group*. Puede implementarse como regiones de memoria dedicada en dispositivos OpenCL. De manera alternativa, la región de memoria local puede ser mapeada en secciones de la memoria global.
- **Memoria privada**: esta memoria se refiere a la región privada de cada *work-item*. Las variables definidas en la memoria privada de un *work-item* no son visibles ni accesibles por ningún otro *work-item* diferente al que la definió.

El modelo de memoria se muestra en la figura 43. En ella se puede observar una memoria privada, disponible y accesible sólo por el *work-item* (este concepto es similar al de *thread* en CUDA) al que está asociada. La memoria local se utiliza para lectura y escritura y es accesible para variables compartidas por distintos *work-items* pertenecientes al mismo grupo de trabajo (*work-group*, que su equivalente en CUDA es el bloque). La memoria global, que representa a la memoria de cada dispositivo, está disponible para lectura y escritura desde cualquier *work-item* o *work-group*. La memoria de constantes, que es una región de la memoria global y es sólo de lectura. Es constante durante la ejecución del *kernel* y puede ser leída y escrita por el *host*. Por último, está la memoria asociada a la CPU (que actúa como *host*).

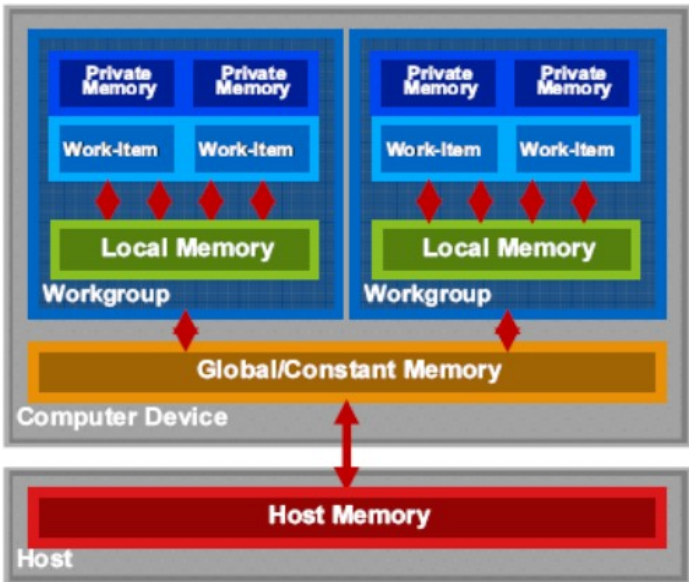


Figura 43: Jerarquía de memoria en OpenCL.

La tabla 9 describe si el *kernel* o el *host* pueden asignar desde una región de memoria, el tipo de asignación (en tiempo de compilación o en tiempo de ejecución) y el tipo de acceso permitido, es decir, si el *kernel* o el *host* pueden leer y/o escribir a una región de memoria.

	Mem. global	**Mem. constantes**	**Mem. local**	**Mem. privada**
Host	Asignación dinámica Acceso R/W	Asignación dinámica Acceso R/W	Asignación dinámica Sin acceso	Sin asignación Sin acceso
Kernel	Sin asignación Acceso R/W	Asignación estática Acceso solo lectura	Asignación estática Acceso R/W	Asignación estática Acceso R/W

Tabla 9: Tipo de asignación y acceso permitido a cada tipo de memoria por parte del *kernel* y el *host* en OpenCL.

4.2.2.4. Principales diferencias con CUDA.

Al igual que CUDA, OpenCL aborda las jerarquías de memoria en su totalidad y la ejecución de datos paralelos. Se basa en gran medida en la experiencia obtenida con el API de CUDA. Por otro lado, OpenCL es una plataforma más compleja, ya que su objetivo es dar soporte multiplataforma. Por este motivo las aplicaciones que hacen uso de OpenCL tienen que estar preparadas para soportar una diversidad de *hardware* mayor. Así, su modelo de gestión de los dispositivos, el de compilación de los *kernels* y la ejecución de estos resulta mucho más compleja que los correspondientes en CUDA.

A continuación, vamos a enumerar algunas de las principales diferencias entre CUDA y OpenCL.

- La principal diferencia radica en la ventaja de que OpenCL es un estándar abierto, a diferencia de CUDA que es propietario, lo que permite que cualquier proveedor pueda dar soporte en sus productos. Por ejemplo, Intel da soporte OpenCL para sus últimos procesadores [183] y para el acelerador Intel-Xeon Phi [61].

- Otra de las diferencias existentes es la terminología, lo que puede acarrear alguna confusión. Por ejemplo, el término de hilo, o *thread*, tiene su correspondencia en OpenCL con el término *work-item*. Además, en OpenCL el concepto de *warp* de CUDA no existe.

- CUDA es más eficiente tanto a la hora de compilar (obteniendo un código optimizado y mejorado) como cuando se trata de gestionar las transferencias de memoria en tiempo de ejecución [184, 185].

- OpenCL en teoría es portable, es decir, válido para cualquier arquitectura, por lo que se podrá ejecutar en múltiples y diferentes dispositivos (GPU, CPU, FPGA, etc.). Sin embargo, es necesario realizar ciertos cambios en el código cuando se desea cambiar de plataforma debido a la capa específica que gestiona el interfaz con cada arquitectura.

4.2.3. Métodos para la programación basada en directivas.

Con el auge de los aceleradores gráficos se hace necesario ofrecer mayores facilidades de programación con el objetivo de poder sacar un mejor partido a esta tecnología. El uso de directivas para guiar al compilador a generar código eficiente no es nueva, siendo una solución intermedia a la compilación automática que facilita el proceso de portabilidad de código. Existen innumerables compiladores que soportan este tipo de programación, destacando el paradigma de programación paralela OpenMP[62], empleado para el desarrollo de un código paralelo en sistemas de memoria compartida. El estándar OpenMP aparece como punto al que convergen todos los fabricantes de sistemas de memoria compartida a principio de los años 2000, como demanda de la comunidad de programadores con el fin de unificar las soluciones que ofrecían los fabricantes de sistemas como Silicon-Graphics [186], IBM, Sun-Microsystems, HP, Intel o Cray entre otros. En esta sección presentamos las principales iniciativas en el ámbito de programación basadas en directivas para GPU, destacando OpenACC [63] como solución unificada que se encuentra en un punto de partida similar al de OpenMP en sus inicios.

4.2.3.1. OpenHMPP.

OpenHMPP [64] nace de la idea de afianzarse como un estándar para la computación heterogénea cuyas siglas HMPP se refieren a *Hybrid Multicore Parallel Programming*. Basado en

[61] OpenCL v1.2 para Intel Xeon-Phi: `http://software.intel.com/en-us/blogs/2012/11/12/introducing-opencl-12-for-intel-xeon-phi-coprocessor`
[62] Especificaciones del estándar de programación paralela OpenMP: `www.openmp.org`
[63] Especificaciones del estándar OpenACC `www.openacc-standard.org`
[64] Directivas de OpenHMPP: `http://www.caps-entreprise.com/openhmpp-directives/`

un conjunto de directivas, está diseñado para la explotación de aceleradores *hardware* sin la complejidad asociada a la programación GPU.

El modelo de programación de directiva basada en OpenHMPP permite al programador indicar mediante una sintaxis sencilla aquellas partes del código que se desea que se ejecuten en el acelerador y de esta manera eliminar la ardua tarea de programar la reserva de memoria en el dispositivo y efectuar las transferencia de datos. El modelo es el producto iniciado por CAPS (*Compiler and Architecture for Embedded and Superscalar Processors*) en un proyecto común de INRIA (Instituto Nacional de Investigación en Informática y Automática francés -*Institut National de Recherche en Informatique et en Automatique*-), CNRS (Centro Nacional de Investigaciones Científicas francés) y la Universidad de Rennes 1. En la actualidad se encuentra disponible en el compilador comercial CAPS y coexiste con el estándar OpenACC.
Está basado en el concepto *codelet* o fragmento de código con las siguientes características:

- Es una función pura.
- No contiene declaraciones de variables estáticas o volátiles, ni se refiere a ninguna variable global, excepto si han sido declarados por una directiva "residente" HMPP.
- No contiene ninguna llamada a función con un cuerpo invisible; básicamente no se pueden invocar funciones *inline*.
- Cada llamada a una función debe referirse a una función pura; no se permiten punteros a función.
- Se invoca una subrutina que no devuelve ningún valor.
- El número de argumentos debe ser fijo.
- No se permite recursividad.

El modelo de memoria es semejante a la de una acelerador con espacios de memoria independientes entre sí. El siguiente ejemplo muestra la codificación de un *codelet* para el cómputo de la multiplicación matriz-vector, el grupo de *codelets* al que pertenece (*grp_label*), viene reflejado con el nombre *simple1*, su identificador único (*codelet_label*) como *codelet_matvec*, y el argumento *outv* que indica que su contenido hay que copiarlo de entrada y salida en el acelerador.

```
/* declaration of the codelet */
 #pragma hmpp simple1 codelet_matvec, args[outv].io=inout, target=CUDA
 static void matvec(int sn, int sm, float inv[sm], float inm[sn][sm], float *outv){
     int i, j;
     for (i = 0 ; i < sm ; i++) {
       float temp = outv[i];
       for (j = 0 ; j < sn ; j++) {
         temp += inv[j] * inm[i][ j];
     }
    outv[i] = temp;
  }
```

4.2.3.2. Acelerador PGI.

El modelo de directivas propuesto por PGI[65] aparece en el 2008 y puede considerarse el "padre" del estándar unificado OpenACC. Proponía un conjunto de directivas al estilo del paradigma de programación OpenMP, que ayudasen al programador a especificar regiones susceptibles de transformarse en *kernels* para la GPU. La mayoría de las directivas de este modelo son opcionales y se utilizan para mejorar el rendimiento. La única directiva que es obligatoria es `acc region` que indica qué porción de código contiene bucles con *kernels*. Además, el compilador PGI mapea el paralelismo de bucles a la arquitectura haciendo uso del módulo *Planner* [187].

Este modelo permitía su uso en códigos en Fortran y en C y en ambos casos se genera un binario tanto para la versión GPU como para la de CPU. El formato de las directivas es muy sencillo, válido para la gestión de datos y la ejecución de los kernels:

```
#pragma acc directive-name [clause [,clause]...] new-line
```

4.2.3.3. hiCUDA.

El modelo de alto nivel hiCUDA[66] (*high-level* CUDA) proporciona al desarrollador un conjunto de directivas que pueden usarse en operaciones CUDA. Los *kernels* se extraen de forma automática del fichero fuente original y las iteraciones se distribuyen entre los hilos y bloques de acuerdo con las cláusulas de particionado de bucles. Otra característica es que para gestionar llamadas a memoria, las rutinas se remplazan por directivas, así el usuario no tiene la necesidad de hacer un seguimiento del dispositivo de punteros.

Desarrollado en la Universidad de Toronto en 2009 [188, 189], trata de llenar el hueco creado por los compiladores de alto nivel que no daban soporte para la paralelización en GPUs, salvo la versión 1.1 del compilador de PGI.

El hecho de que las directivas sean una traducción directa al modelo de programación de CUDA obliga al usuario a conocer en más detalle la plataforma de destino. Además, esto puede dificultar la portabilidad de dichos códigos a diferentes tipos de aceleradores.

El resultado de compilar con hiCUDA no es un binario, sino un directorio que contiene un fichero con código CUDA y varios ficheros cabecera necesarios. Con estos se obtiene el fichero binario final, una vez que se han compilado con las herramientas de NVIDIA.

A modo de ejemplo presentamos a continuación un código para la multiplicación de matrices empleando el modelo hiCUDA.

[65] Más información sobre *PGI Accelerator v1.1*: `http://www.pgroup.com/lit/whitepapers/pgi_accelerator.pdf`

[66] El proyecto *hiCUDA*: `http://www.eecg.utoronto.ca/~tsa/hicuda/`

```
#pragma hicuda global alloc A[*][*] copyin
#pragma hicuda global alloc B[*][*] copyin
#pragma hicuda global alloc C[*][*]

#pragma hicuda kernel matrixMul tblock(4,2) thread(16,16)
// C = A * B
#pragma hicuda loop_partition over_tblock over_thread
    for (i = 0; i < 64; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
        for (j = 0; j < 32; ++j) {
            float sum = 0;
            for (kk = 0; kk < 128; kk += 32) {
#pragma hicuda shared alloc A[i][kk:kk+31] copyin
#pragma hicuda shared alloc B[kk:kk+31][j] copyin
#pragma hicuda barrier
                for (k = 0; k < 32; ++k) {
                    sum += A[i][kk+k] * B[kk+k][j];
                }
#pragma hicuda barrier
#pragma hicuda shared remove A B
            }
            C[i][j] = sum;
        }
    }
#pragma hicuda kernel_end
#pragma hicuda global copyout C[*][*]
#pragma hicuda global free A B C
```

4.2.3.4. OpenMPC.

OpenMPC[67] es una extensión de OpenMP para programar en GPUs [190, 191] que aparece en el año 2009, dotando a ésta de un conjunto de directivas y variables de entorno para controlar parámetros importantes relacionados con CUDA. Aborda dos cuestiones importantes para la programación GPGPU: la capacidad de ajuste y la programabilidad. Además, proporciona a los programadores una interfaz con el usuario que se abstrae del modelo de programación de CUDA.

En esta misma línea encontramos recientemente (Marzo 2013) una propuesta del grupo de trabajo para la versión 4.0 de OpenMP[68] que están discutiendo propuestas para dotar al estándar OpenMP de extensiones para aceleradores que incluyan directivas con funcionalidades similares a las descritas en esta sección. La API nueva contendría unos constructores

[67] *OpenMPC*: `https://engineering.purdue.edu/paramnt/OpenMPC/`
[68] OpenMP v4 Public Review RC 2 `http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf`

target, donde se especificarían el emplazamiento de los datos mediante la directiva `#pragma omp target data`, de igual modo se posibilitaría la actualización de información en el *host* mediante la directiva `#pragma omp target update`.

4.2.4. OpenACC.

Con respecto a los modelos de programación basados en directivas, destacamos el caso de *OpenACC*. Desarrollado por las compañías Cray, NVIDIA, CAPS y PGI, que trata de consolidarse como un estándar de programación para computación paralela, está diseñado para simplificar la programación de sistemas heterogéneos CPU/GPU. En el ámbito universitario se encuentra el compilador accULL [192], de la Universidad de La Laguna, que actualmente soporta dicho estándar, estando disponible para el público[69].

Con la misma filosofía que OpenMP, OpenACC describe una colección de directivas del compilador para bucles específicos y regiones de código (en C, C++ o Fortran) que se van a ejecutar en el dispositivo, proporcionando portabilidad entre sistemas operativos, CPUs y aceleradores.

Los compiladores OpenACC analizan automáticamente la estructura de todo el programa y los datos (las porciones divididas por el conjunto de directivas que indican cuáles son las que se van a ejecutar en CPU y cuáles en el acelerador) y definen y generan una asignación optimizada de los bucles con el objetivo de usar automáticamente los núcleos paralelos, las capacidades *hardware* de los hilos y capacidades vectoriales SIMD de los aceleradores modernos. Además de las directivas y `#pragmas` que especifican las regiones de código o funciones que deben acelerarse, se proporcionan otras directivas de grano fino para controlar la asignación de los bucles, la asignación de memoria, y la optimización de la jerarquía de memoria del acelerador.

4.2.4.1. El modelo de ejecución en OpenACC.

El modelo de ejecución de OpenACC es similar al de CUDA u OpenCL, donde el *host* está unido al dispositivo acelerador, ejecutando gran parte de la aplicación en el *host* y las regiones paralelas en el acelerador, bajo supervisión del *host* [193]. Incluso en las regiones potenciales a ejecutarse en el dispositivo, el *host* debe orquestar la ejecución mediante la reserva de memoria en el acelerador, inicializando la transferencia de datos y enviando código al dispositivo para después pasar los argumentos que se necesitan para la región paralela. También se ha de encolar el código del dispositivo, esperando a que se completen las tareas de este, de manera que los datos se transfieran de nuevo al *host* y desasignando la memoria previamente asignada. En la mayoría de los casos, el *host* puede encolar una secuencia de operaciones que se van a ejecutar, una tras otra, en el dispositivo acelerador. La figura 44 muestra, de forma esquemática, el modelo de ejecución en OpenACC.

[69] Compilador accULL: `http://cap.pcg.ull.es/accULL`

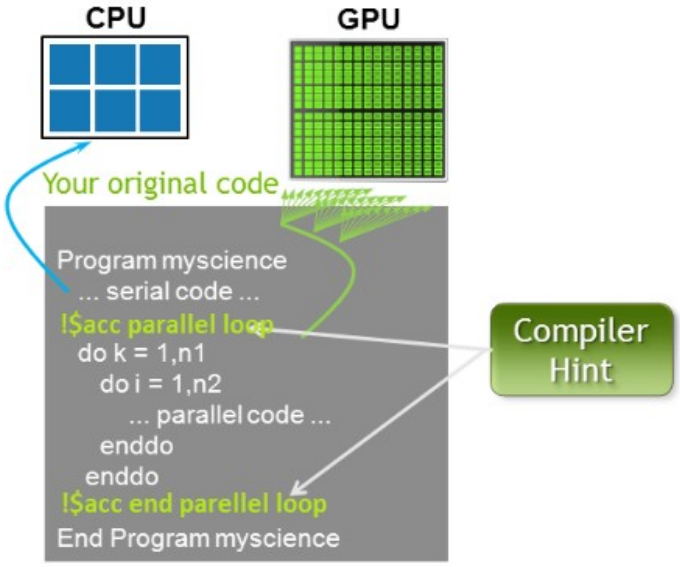


Figura 44: Modelo de ejecución en OpenACC.

El modelo de ejecución en el dispositivo expone múltiples niveles de paralelismo (de grano grueso, de grano fino, operaciones vectoriales o SIMD), así que es responsabilidad del programador conocer las diferencias entre, por ejemplo, un bucle totalmente paralelizable y un bucle vectorizable que requiere sincronización entre declaraciones. En este caso, un bucle totalmente paralelizable puede programarse para una ejecución paralela de grano grueso, mientras que para el caso de bucles con dependencias hay que dividirlos para permitir este tipo de ejecución o programarlos para que se ejecuten en una única unidad de ejecución haciendo uso de paralelismo de grano fino o secuencial.

OpenACC permite tres niveles de paralelismo mediante *gangs*, *workers* y *vector*. Cabe destacar que, dependiendo del compilador y de la arquitectura, los conceptos *worker* y *vector* pueden variar. Por ejemplo, si asumimos una arquitectura CUDA, para el compilador de PGI[70] un *vector* es similar al hilo de CUDA y un *worker* tendría su equivalente en el *warp* mientras que para el compilador de CAPS [71] sería al contrario: un *vector* tiene su equivalente en el *warp* de CUDA y el *worker* es el hilo. Además, un *gang* está formado por uno o más *workers*/*vectors*. El paralelismo a nivel de *gang* es de grano grueso. De esta forma pueden ejecutarse en el dispositivo un número N de *gangs*. Teniendo en cuenta la definición del compilador de CAPS para los *workers* y *vectors*, el paralelismo a nivel de *worker* es de grano fino y el paralelismo a nivel de *vector* es para las operaciones vectoriales y SIMD de un *worker*. En la tabla 10 puede observarse la equivalencia de términos entre OpenACC, CUDA y OpenCL y en la figura 45 puede observarse la relación de los términos expuestos anteriormente.

[70]Compilador PGI: `http://www.pgroup.com/resources/accel.htm`
[71]Compilador CAPS: `http://www.caps-entreprise.com/products/caps-compilers`

CUDA	OpenCL	OpenACC
host	*host*	*host*
thread	*work-item*	*vector*
block	*work-group*	*gang*
warp	-	*worker*
device	*device*	*device*

Tabla 10: Equivalencia de términos en OpenACC, CUDA y OpenCL.

En la tabla 10, los términos relacionados con OpenACC se refieren a la definición dada por el compilador de PGI y con respecto a una arquitectura CUDA. Como se ha mencionado antes, dependiendo del compilador, estos términos podrían variar.

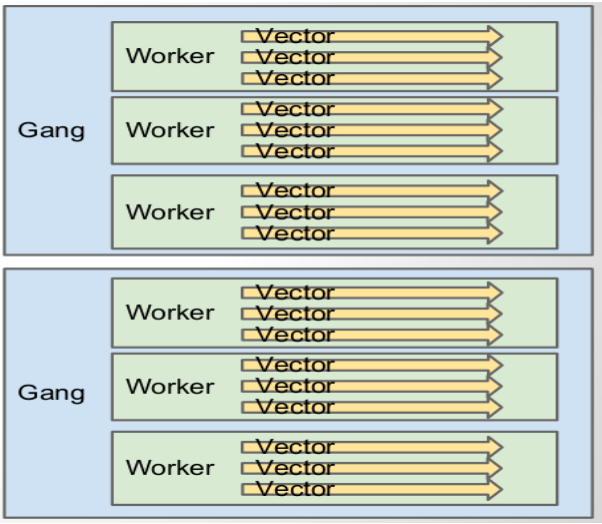


Figura 45: Relación entre *gang*, *worker* y *vector* en OpenACC según la definición de términos de PGI.

Cuando se ejecuta una región de código en el dispositivo, se lanzan uno o varios *gangs*, cada uno de ellos con uno o más *workers*, que a su vez cada uno de estos puede contener una o más rutas vectoriales. Los *gangs* empiezan a ejecutarse en modo GR (*gang-redundant*), que significa que una ruta vectorial de un *worker* en cada *gang* ejecuta el mismo código de manera redundante. Cuando el programa alcanza un bucle marcado como *gang-level work-sharing*, el programa empieza a ejecutarse en modo GP (*gang-partitioned*), donde las iteraciones del bucle se particionan o distribuyen entre *gangs* para una ejecución paralela real, pero con un solo *worker* por cada *gang* activo. En cualquiera de los dos modos anteriores, con un solo *worker* activo, se dice que el programa está en modo WS (*work-single*). Si sólo hay una ruta vectorial activa, el programa se encuentra en modo VS (*vector-single*). De esta forma, si el programa llega a una sección de código donde un bucle está marcado como compartición a nivel de *worker* (*worker-level sharing*), el programa se ejecuta en modo WP (*worker-partitioned*), haciendo que las iteraciones del bucle se distribuyan entre los *workers* de un mismo *gang*. Si el bucle estuviera marcado como GP y WP a la vez (partición a nivel de

gang y de *worker*) las iteraciones del bucle se distribuirían entre todos los *workers* de todos los *gangs*. Otra forma de marcar un bucle es en modo VP (*vector-partitioned*) indicando que las iteraciones de dicho bucle se ejecutarán haciendo uso de instrucciones vectoriales o SIMD a través de las rutas vectoriales.

El siguiente código muestra un ejemplo de multiplicación de matrices haciendo uso de OpenACC, donde se inicializan las matrices en la CPU y se realizan los cálculos en la GPU.

```
// Inicialización de matrices.
 for (i = 0; i < SIZE; ++i) {
   for (j = 0; j < SIZE; ++j) {
     a[i][j] = (float)i + j;
     b[i][j] = (float)i - j;
     c[i][j] = 0.0f;
   }
 }
//Multiplicación de matrices
#pragma acc kernels copyin(a,b) copy(c)
 for (i = 0; i < SIZE; ++i) {
   for (j = 0; j < SIZE; ++j) {
     for (k = 0; k < SIZE; ++k) {
      c[i][j] += a[i][k] * b[k][j];
     }
   }
 }
```

Donde con `#pragma acc kernels` se indica que es una región que se va a enviar a la GPU, con `copyin()` se va a copiar la región contigua de memoria desde la CPU al acelerador y con `copy()` la región de memoria contigua se va a copiar de la CPU al acelerador y viceversa. Como sabemos, el mayor cuello de botella se encuentra en las transferencias de CPU a GPU, por lo tanto, en este ejemplo los encontraríamos en la copia de datos desde la CPU a la GPU y viceversa. Para evitar estos cuellos de botella, OpenACC proporciona una serie de directivas que se explican a continuación. Con `create()` se crean e inicializan las matrices en la GPU y con `copyout()` se indica que la transferencia se hará de la GPU a la CPU, evitando así tener que transferir datos de CPU a GPU. Además, para indicar al compilador que ignore el análisis de dependencia y confíe en que el programador sabe que no existen dependencias, se puede utilizar `#pragma acc loop independent`. De forma inversa, se puede indicar al compilador que genere código que se va a ejecutar en secuencial en la GPU con `#pragma acc loop seq`. Si hacemos uso de estas directivas, se puede producir un código que será más efectivo que el que se ha mostrado con anterioridad:

```
#pragma acc kernels create(a[0:size][0:size], b[0:size][0:size])
 copyout(c[0:size][0:size])
  {
    // Inicialización de matrices.
    #pragma acc loop independent
    for (i = 0; i < size; ++i) {
     #pragma acc loop independent
     for (j = 0; j < size; ++j) {
      a[i][j] = (float)i + j;
      b[i][j] = (float)i - j;
      c[i][j] = 0.0f;
     }
    }

   // Calcular la multiplicación de matrices
   #pragma acc loop independent
   for (i = 0; i < size; ++i) {
    #pragma acc loop independent
    for (j = 0; j < size; ++j) {
     #pragma acc loop seq
     for (k = 0; k < size; ++k) {
      c[i][j] += a[i][k] * b[k][j];
     }
    }
   }
  }
```

Estas y otras opciones se pueden consultar en el manual de OpenACC [193].

OpenACC también ofrece interoperabilidad con CUDA. En el ejemplo que se presenta a continuación se muestra la implementación de una convolución 1D haciendo uso de OpenACC en conjunción con CUDA. Más concretamente con CUFFT[72], una librería para realizar cálculos relacionados con la transformada rápida de Fourier y que hace uso de CUDA. En este ejemplo se calcula la convolución en la frecuencia del espacio donde CUFFT se utiliza para transformar la señal de entrada en el dominio de la frecuencia. La función que multiplica los coeficientes y normaliza los resultados es la que se realiza haciendo uso de OpenACC.

```
//Se transforma la señal
 error = cufftExecC2C(plan, (cufftComplex *)d_signal,
                      (cufftComplex *)d_signal, CUFFT_FORWARD);
 error = cufftExecC2C(plan, (cufftComplex *)d_filter_kernel,
                      (cufftComplex *)d_filter_kernel, CUFFT_FORWARD);
```

[72]CUFFT: `https://developer.nvidia.com/cufft`

```
//Se multiplican los coeficientes y se normaliza el resultado
 complexPointwiseMulAndScale(new_size, (float *restrict)d_signal,
                             (float *restrict)d_filter_kernel);

//Se transforma la señal de vuelta
 error = cufftExecC2C(plan, (float *restrict)d_signal,
                      (float *restrict)d_signal, CUFFT_INVERSE);
```

En este caso, es en la función `complexPointwiseMulAndScale` donde se utiliza OpenACC ya que tiene que ejecutarse en el dispositivo, pues los datos se encuentran almacenados allí. Con la cláusula `deviceptr` se indica que los punteros que se pasan como parámetro son punteros que se encuentran en la memoria del dispositivo, así que no es necesario que los datos se muevan entre el *host* y el dispositivo.

```
void complexPointwiseMulAndScale(int n, float *restrict signal,
                                 float *restrict filter_kernel)
{
  #pragma acc data deviceptr(signal, filter_kernel)
     {
  #pragma acc kernels loop independent
        for (int i = 0; i < n; i++)
        {
           float ax = signal[2*i];
           float ay = signal[2*i+1];
           float bx = filter_kernel[2*i];
           float by = filter_kernel[2*i+1];
           float s = 1.0f / n;
           float cx = s * (ax * bx - ay * by);
           float cy = s * (ax * by + ay * bx);
           signal[2*i] = cx;
           signal[2*i+1] = cy;
        }
     }
}
```

Para sacar mayor partido de los sistemas híbridos CPU/GPU, OpenACC se puede combinar con OpenMP para explotar mayor paralelismo de una manera simple, obteniendo así aceleraciones mayores, ya que se hace uso de la capacidad de ejecución en paralelo tanto de las CPUs como de las GPUs.

4.2.4.2. El modelo de memoria en OpenACC.

La principal diferencia entre un programa que se ejecuta solo en un *host* y otro que se ejecuta en una combinación de *host*+acelerador es que la memoria del acelerador puede estar completamente separada de la memoria del *host*, como sucede en la mayoría de los sistemas con GPUs, donde el hilo que se ejecuta en el *host* no puede escribir ni leer de manera directa en la memoria del dispositivo. Todos los movimientos desde y hacia la memoria del *host* o del dispositivo deben realizarse por el hilo del *host* mediante llamadas al sistema que mueven

de manera explícita los datos entre memorias separadas, usando normalmente transferencias DMA (*Direct Memory Access*). De forma similar, se asume que el acelerador no puede leer o escribir en la memoria del *host*, aunque algunos dispositivos aceleradores si lo soportan.

En el modelo OpenACC, el movimiento de datos entre memorias puede realizarse de manera implícita y ser gestionado por el compilador basándose en directivas especificadas por el programador. Sin embargo, el programador debe tener cuidado por varias razones. Una de ellas es que el ancho de banda entre la memoria del *host* y la memoria del dispositivo determina el nivel de intensidad computacional requerido para que las aceleraciones de una región de código sean efectivas. Además, el tamaño de memoria del dispositivo acelerador puede no permitir la carga de regiones de código que operan sobre grandes cantidades de datos.

OpenACC expone las memorias separadas a través de la utilización de un entorno de datos del dispositivo acelerador. Los datos del dispositivo tienen una vida útil explícita, desde el momento en el que se asignan o se crean hasta que se eliminan. Si el dispositivo comparte la memoria física y virtual con el hilo local, el entorno de datos del dispositivo se compartirá con el hilo local. En este caso, la implementación no necesita crear copias nuevas de los datos ni moverlos. Si el dispositivo cuenta con una memoria separada del hilo local, la implementación asignará el dato nuevo en la memoria del dispositivo y realizará una copia de este en la memoria local del entorno del dispositivo.

Algunos aceleradores actuales tienen una caché administrada por *software*, otros poseen una administrada por *hardware*, y la mayoría tienen caches *hardware* que se pueden utilizar sólo en ciertas situaciones y están limitadas a sólo lectura. En los modelos de programación de bajo nivel, como CUDA y OpenCL, es responsabilidad del programador gestionar estas *caches*. En el modelo OpenACC, estas memorias caché las gestiona el compilador con ayuda del programador en forma de directivas.

4.3. Otras soluciones: Xeon Phi.

En 2011 Intel presenta los procesadores **Xeon Phi**, que son la primera arquitectura MIC (*Intel Many Integrated Core*) en convertirse en un producto comercial y que, junto a los procesadores Xeon (conectados con estos mediante un bus *PCI Express*), forman una arquitectura heterogénea de alto rendimiento. Aunque por si mismas son un procesador, Intel las considera un co-procesador ya que las aplicaciones con un solo hilo tienen un rendimiento mejor en los procesadores Xeon. Así, las aplicaciones pueden mejorar su rendimiento haciendo uso de los procesadores Xeon junto a los co-procesadores Xeon Phi. Estas aplicaciones deben utilizar un co-procesador cuando pueden explotar un alto grado de paralelismo. La figura 46 muestra la arquitectura de un *core* en la que se observa capacidad para *multithreading*, una unidad vectorial y compatibilidad para $\times$86-64.

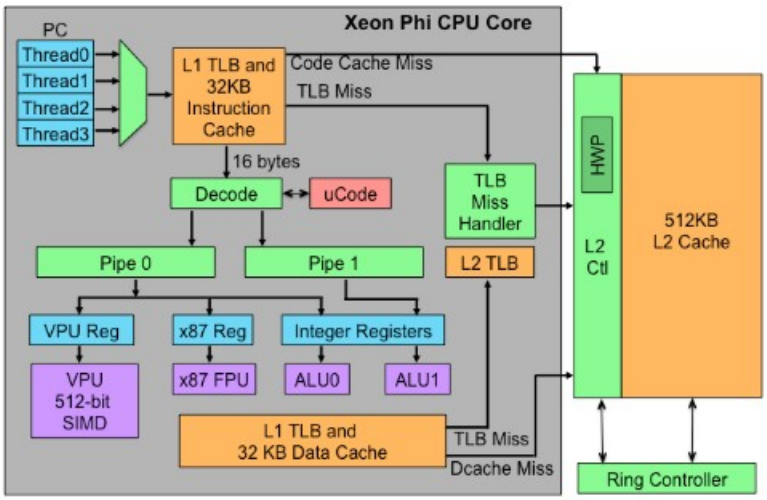


Figura 46: Microarquitectura de un *core* Xeon Phi.

La arquitectura Xeon Phi está fabricada en 22nm y ofrece un rendimiento de doble precisión superior al TeraFLOP (un trillón de operaciones en coma flotante por segundo). Las aplicaciones objetivo para el co-procesador Xeon Phi son aquellas que poseen problemas masivamente paralelos y que en la actualidad se resuelven haciendo uso de grandes *clusters*. Así, la arquitectura del co-procesador está diseñada de forma similar a un *cluster*, con más de 50 núcleos conectados mediante una estructura de anillo. La figura 47 muestra el diagrama de bloques de un co-procesador Xeon Phi donde pueden observarse los núcleos, cada uno con su caché L2 totalmente coherente, interconectados por un bus bidireccional en anillo además de los controladores de memoria de tipo GDDR5 y la lógica de entrada/salida para el bus PCIe.

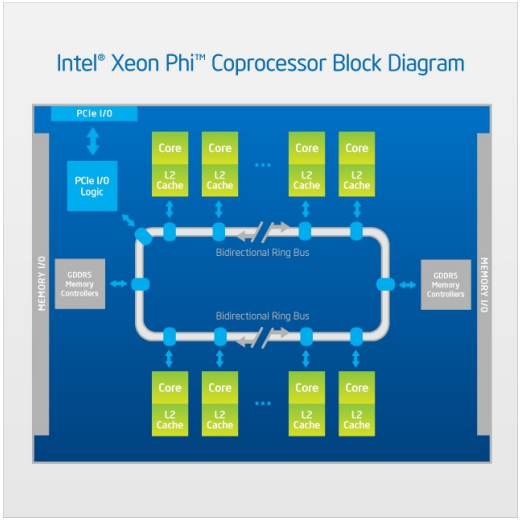


Figura 47: Diagrama de bloque de un co-procesador Xeon Phi.

Se puede considerar que cada núcleo del co-procesador Xeon Phi es una unidad de ejecución multi-hilo completamente funcional. La figura 48 muestra el esquema de un núcleo de este co-procesador, en la que se puede observar la unidad escalar, basada en la familia de procesadores Pentium, que tiene dos *pipelines* de emisión dual con instrucciones escalares, y una nueva unidad vectorial de 512 bits con un conjunto de instrucciones SIMD y 32 registros vectoriales de 512 bits. Además, hay 4 hilos *hardware* por núcleo, donde cada hilo emite instrucciones en turno siguiendo una política *round-robin*.

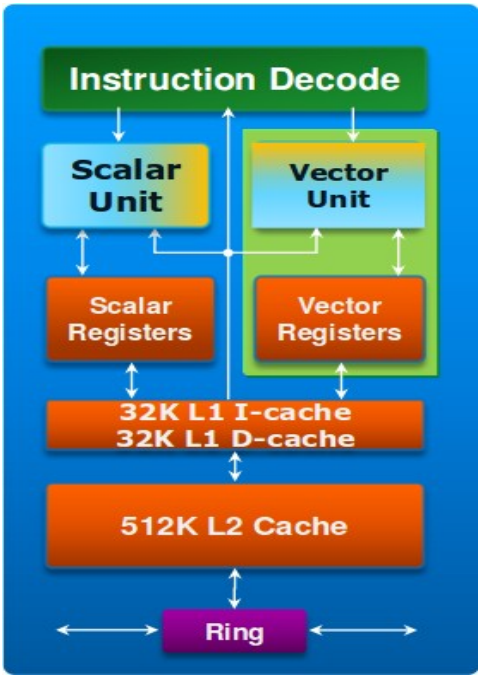


Figura 48: Esquema de un núcleo del co-procesador Xeon Phi.

La Unidad Vectorial (VPU) es uno de los componentes más importantes de los núcleos del co-procesador Xeon Phi, ya que ofrece un conjunto de instrucciones SIMD de 512 bits nuevo, al que se le conoce como IMCI (*Initial Many Core Instructions*), además de poseer operaciones de máscara, *gather*, *scatter* similares a los procesadores vectoriales antiguos. Estas unidades vectoriales pueden ejecutar 16 u 8 operaciones por ciclo (de precisión simple y doble, respectivamente). Además, proporcionan soporte para números enteros y para instrucciones FMA por lo que se pueden ejecutar 32 o 16 operaciones (de simple y doble precisión) por ciclo. En la figura 49 se puede observar un esquema de la Unidad Vectorial.

Estas unidades vectoriales son muy eficientes en términos de energía para computación de alto rendimiento, ya que una única operación puede codificar una gran cantidad de trabajo sin incurrir en los costes de energía asociados a la hora de buscar, decodificar y retirar muchas instrucciones.

Se han introducido una serie de mejoras para poder soportar las instrucciones SIMD que,

entre otras acciones, ayudan a vectorizar los saltos condicionales, mejorando así la eficiencia global. Además, soportan las instrucciones *gather* y *scatter* permitiendo de esta manera mantener el código vectorizado, sobre todo en códigos con patrones de acceso irregulares. Otra ventaja es que cuentan con una Unidad Matemática Extendida (UEM) que puede ejecutar operaciones como la raíz cuadrada, lo que permite que estas operaciones sean realizadas de manera vectorial con gran ancho de banda. La UEM funciona mediante el cálculo de las aproximaciones polinómicas de estas funciones.

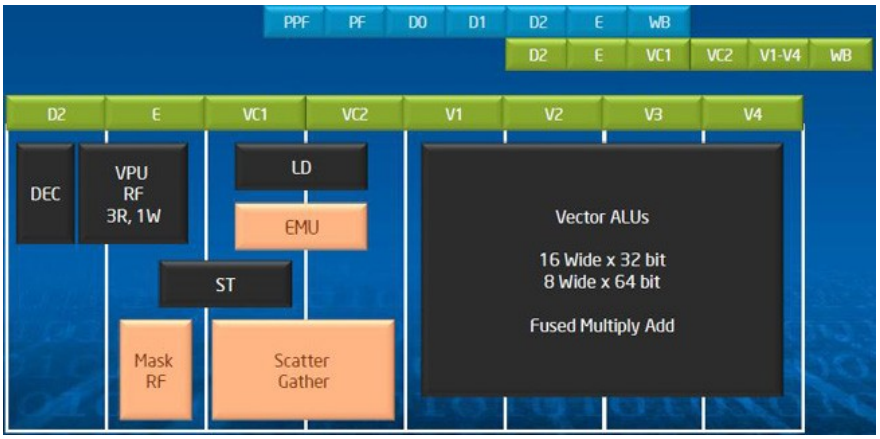


Figura 49: Unidad Vectorial del co-procesador Xeon Phi.

Los co-procesadores Xeon Phi ofrecen la completa capacidad de utilizar las mismas herramientas, los mismos modelos y lenguajes de programación que un procesador Xeon. Sin embargo, como están diseñados para altos grados de paralelismo, algunos modelos son más interesantes que otros. Por ejemplo, para programadores de C++ se recomienda hacer uso de Intel TBB[73], Intel Clik Plus[74] y OpenMP [194] .

Actualmente, el modelo del co-procesador Intel Xeon Phi 5110P posee, entre otras características; 60 núcleos con una frecuencia de reloj de 1053 GHz y 240 hilos, 8 GB de memoria y un ancho de banda máximo de 320 GB/seg e instrucciones SIMD de 512 bits.

[73]Intel TBB: `http://software.intel.com/en-us/intel-tbb`
[74]Intel Clik Plus: `http://software.intel.com/en-us/intel-cilk-plus`

5. Estudio de viabilidad de McGM con GPU's.

En este capítulo se describe una implementación específica y eficiente en GPU de un modelo de flujo óptico basado en un modelo de gradiente. Este esquema se particulariza por usar un sistema neuromórfico para la estimación del movimiento, es decir, realiza la extracción de movimiento de manera robusta, como se vio en el capítulo 3. Además, posee muchas características que aumentan la capacidad de estimación cuando son comparados con otros modelos ópticos pertenecientes a la familia de algoritmos que utilizan modelos de gradiente. La implementación que se presenta en este capítulo ha sido desarrollada haciendo uso de GPUs y diseñada específicamente para este modelo, pudiéndose reutilizar en muchas aproximaciones de bajo nivel. Los resultados observados indican que este tipo de aceleradores son altamente recomendados para llevar a cabo esta tarea. Además, se han analizado las salidas obtenidas en comparación con las generadas por CPU para estudiar la precisión del sistema construido.

Para concluir el estudio, se ha implantado en un sistema empotrado basado en GPU con el objetivo de estudiar su posible implementación.

5.1. Introducción.

El modelo McGM, descrito en el capítulo 3, fue desarrollado como parte de un esfuerzo de investigación destinado a mejorar nuestra comprensión del sistema visual humano. Gracias a ello, dicho modelo también nos permite hacer predicciones que pueden probarse a través de la experimentación psicofísica, como ilusiones de movimiento independientes que se observan en experiencias estudiadas con seres humanos [72]. La contribución de este trabajo sigue las líneas necesarias, con el objetivo de realizar una implementación eficiente para modelos de flujo óptico basados en técnicas de gradiente, haciendo uso de plataformas de *hardware* gráfico, explotando diferentes niveles de paralelismo (a nivel de bloque y de *pixel*) en cada etapa del algoritmo y haciendo uso de manera eficiente de la jerarquía de memoria (combinando la memoria global con la compartida de las GPUs). El resultado de esta optimización en todas las etapas es una mejora notable del rendimiento (pasando por ejemplo de $\times 23$ a $\times 82$ dependiendo de la etapa y obteniendo un rendimiento global de $\times 32$ cuando se usa una imagen con una resolución de 256×256). Esta plataforma es particular, entre otros enfoques de visión de bajo nivel, para un esquema de estimación de movimiento bioinspirado.

5.2. Implementación en la GPU.

Para el desarrollo de este trabajo se ha aplicado la descripción del algoritmo McGM [169, 175] propuesto por Johnston y se han añadido variaciones específicas para mejorar el rendimiento y la viabilidad para una implementación en GPU.

La implementación del algoritmo McGM para este estudio se ha realizado haciendo uso de la tecnología Tesla de NVIDIA. Este sistema contiene varias GPUs que pueden ser programadas haciendo uso del paradigma CUDA. Tal y como se ha visto en el capítulo 4, CUDA es un

conjunto de herramientas proporcionadas por NVIDIA [179] que, entre otras cosas, incluye un compilador específico para *hardware* gráfico. En este capítulo se explica el esquema que se ha llevado a cabo para implementar el algoritmo McGM en una GPU. Se ha asumido que los datos de entrada (la información de los fotogramas) se obtiene en tiempo real desde un dispositivo externo, por ejemplo, una cámara, y que el sistema se puede alimentar con un número de fotogramas considerable.

A continuación se abordará la descripción del mapeo del algoritmo McGM en una GPU. Con el objetivo de facilitar la tarea de programación del algoritmo. Cada etapa descrita en el capítulo 3 corresponde con un *kernel* para la GPU, salvo para las etapas IV y V que se han mapeado en uno solo para simplificar y facilitar la codificación.

5.2.1. Etapa I: Filtrado temporal.

La primera etapa desarrollada es la del filtrado temporal, donde los filtros temporales han sido cuantificados como $k(t)$ en la expresión (14) del capítulo 3. En este punto, es necesario hacer una discretización de los valores del caso continuo y un posterior ecualizado de forma que la integral de las diferentes funciones se ajuste a cero. Para llevar a cabo su implementación en *hardware* gráfico, se han examinado diferentes esquemas en la GPU con el objetivo de quedarse con los ratios de eficiencia válidos, es decir, escoger la solución más adecuada. La primera versión almacena los coeficientes de los filtros usando distintos niveles de la jerarquía de memoria: global, de texturas o de constantes. La segunda versión conserva la información de N *frames*, o fotogramas, en la memoria global o en la memoria compartida. Por último, una tercera versión ha permitido examinar el rendimiento que se alcanza cuando se reutiliza la información almacenada en la memoria de constantes.

Los N fotogramas involucrados en esta etapa se dividen en bloques de 16×16, cada uno de los cuales se mapea en un multiprocesador. El procesado de estos fotogramas en bloques de 16×16 lo realizan, además, 16×16 hilos, de forma que cada hilo realice los cálculos equivalentes al procesado de un *pixel* de salida.

Como salida de esta etapa se obtiene un suavizado temporal (filtrado de paso baja) y las derivadas de primer y segundo orden (filtrado de paso banda) de la secuencia original, como se observa en la figura 50.

El algoritmo 7 muestra el código para el *kernel* de esta primera etapa, en la que se realizan los cálculos necesarios para filtrado temporal en la GPU, almacenando los datos en *field*.

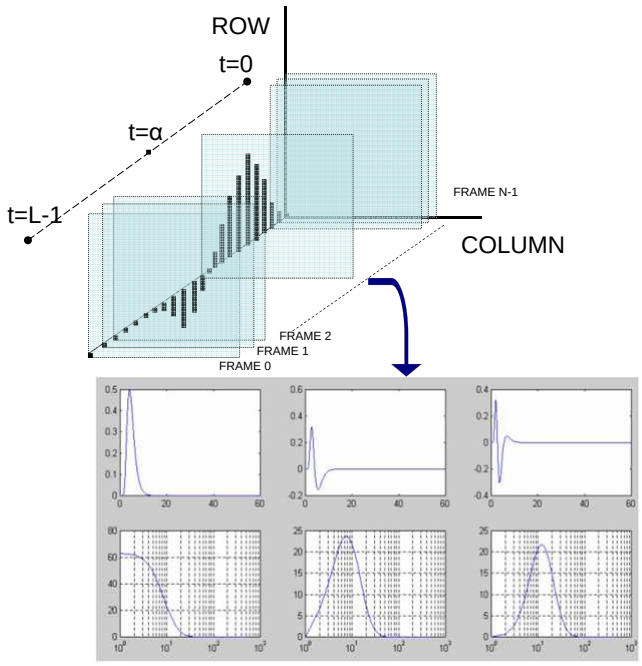


Figura 50: Paso de una realización de filtro temporal, (siendo t=0 el primer fotograma, N el último y L la longitud del filtro) que nos da una respuesta para el fotograma α.

Algorithm 7 $field$ = Filtrado_Temporal(frames, N_{frames}, Tam_{filts}, $N_{temporal\_filters}$)

$id = (blockIdx.y * BLOCK\_SIZE + threadIdx.y) * size_{frame}X + blockId.x * BLOCK\_SIZE + threadId.x$
$size_{frame} = size_{frame}X * size_{frame}Y$
for $i = 0$; $i < N_{frames} - Tam_{filts}$; $i++$ **do**
  for $t = 0$; $t < Tam_{filts}$; $t++$ **do**
    $val_frame =$frames$[size_{frame} * (i + t) + id]$
    $dev0+ = filtros\_t[(Tam_{filts} - 1) - t] * val\_frame$
    $dev1+ = filtros\_t[(2 * Tam_{filts} - 1) - t] * val\_frame$
    $dev2+ = filtros\_t[(3 * Tam_{filts} - 1) - t] * val\_frame$
  end for
  $field[i * size_{frame} * N_{temporal\_filters} + id] = dev0$
  $field[i * size_{frame} * N_{temporal\_filters} + 1 + id] = dev1$
  $field[i * size_{frame} * N_{temporal\_filters} + 2 + id] = dev2$
end for

Los parámetros de entrada del *kernel* son: los fotogramas (*frames*), almacenados en memoria

de GPU, y el número de estos (N_{frames}), el tamaño del filtro (Tam_{filts}) y el número de filtros temporales ($N_{temporal\_filters}$). Además, *id* representa al identificador de cada hilo, que realiza los cálculos correspondientes a cada *pixel*. Además, la variable *filtros_t* estará almacenada en memoria global, de constantes o compartida dependiendo de la versión desarrollada, de entre las distintas explicadas con anterioridad.

5.2.2. Etapa II: Filtrado espacial.

Durante esta etapa, la del filtrado espacial, se realizan las convoluciones espaciales mostradas en el modelo. Previamente se aplica un filtro de paso baja para suavizar las imágenes y mejorar los resultados de las etapas posteriores. A cada una de las tres salidas temporales obtenidas en la etapa anterior se le aplica una "pirámide" espacial de varios filtros (como ejemplo se muestran 10 en la figura 51).

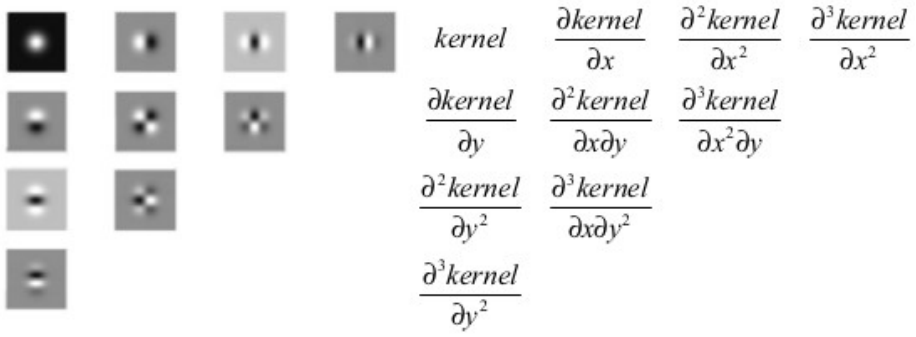


Figura 51: Pirámide de filtros diferenciales espaciales de orden 0 hasta 3.

La estructura está constituida por diferentes derivadas gaussianas, donde cada diagonal está compuesta usando el mismo orden de diferenciación que una función espacial básica (señalada como *kernel* en la figura 51).

La implementación en GPU se basa en la creación de dos *kernels* diferentes con el objetivo de reducir el orden computacional. La convolución de una máscara con una imagen normalmente tiene una complejidad de $O(n^2)$. Sin embargo, si se realiza una convolución separada (ya que la gaussiana es una función separable), es posible reducir la complejidad a $O(2n)$.

La estructura principal del *kernel* se ha realizado tomando como ejemplo el código de la convolución proporcionado por el SDK de NVIDIA [195], teniendo en cuenta que en este caso no se realiza una convolución centrada, como sí se hace en el ejemplo proporcionado por el SDK. Se han escogido valores altos para el tamaño de *blockid.x* o *blockid.y*, dependiendo de la dirección en la que se realice el filtrado (la nomenclatura de CUDA se refiere a los números de bloques en cada dirección), con el objetivo de aprovechar el mayor paralelismo posible explotando la reutilización de datos intrínseco a la convolución. Los coeficientes de los filtros a aplicar han sido almacenados en la memoria de texturas de la GPU, con menor tiempo de latencia, y se aborda el cálculo de forma que la información de salida quede almacenada

consecutivamente en memoria global según el orden que se procesa.

Para entender mejor la implementación de esta etapa, el algoritmo 8 muestra el pseudo-código del código desarrollado, donde *basis* es la salida de la etapa II, *field* es la salida de la etapa I (ver sección 5.2.1), que a su vez es la entrada de datos para la etapa II, la variable *filtros_s* se refiere a los filtros espaciales y $spac_{filts}$ al tamaño de estos. Las convoluciones separables ($Convolution2D_{columnas}$ y $Convolution2D_{filas}$) son *kernels* independientes que se procesan en la GPU siguiendo el esquema antes mencionado.

Algorithm 8 basis = Filtrado_Espacial($field$, filtros, $spac_{filts}$)

for all $i = 0$; $i < size_{field}$; $++i$ **do**
  $frame_i = field[i]$
  for all $j = 0$; $j < spac_{filts}$; $++j$ **do**
    temp=$Convolution2D_{columnas}(frame_i, filtros_j)$
    for all $k = 0$; $k \leq spac_{filts} - j$; $++k$ **do**
      basis$_{i,j,k}$=$Convolution2D_{filas}(temp, filtros_k)$
    end for
  end for
end for

Como puede observarse, para cada fotograma obtenido en la etapa anterior (*field*), se aplica una pirámide de convoluciones separable, por columnas primero y luego por filas, para obtener las convoluciones espaciales del modelo, que son almacenadas en *basis*.

5.2.3. Etapa III: Orientación de filtros.

Una vez obtenidos los resultados de los cálculos realizados en las etapas anteriores, se procede a realizar la orientación de filtros. Sin embargo, las operaciones que conllevan la orientación, se suelen evitar en los sistemas de visión por computación [196] debido a su alto coste computacional, lo que conlleva un alejamiento de los requisitos de tiempo real. En esta tercera etapa es necesario procesar un filtrado con orientación θ para cada uno de los filtrados procesados previamente, en las etapas anteriores. Gracias a la linealidad y a propiedad conmutativa de la convolución, es posible sintetizar el filtro orientado F^θ a partir del conjunto de filtros de su base (mismo orden de diferenciación) y, además, también se puede obtener su respuesta orientada R^θ a partir de su conjunto de respuestas $R_1, ..., R_n$, tal y como muestra la expresión siguiente (24):

$$R^\theta = F^\theta \otimes I = (K_1F_1 + K_2F_2 + ... + K_nF_n) \otimes I = K_1R_1 + K_2R_2 + ... + K_nR_n \quad (24)$$

donde el peso de los coeficientes está representado por $K_1, ..., K_n$, I representa la imagen de entrada que va a ser orientada y θ es el grado de orientación a ser proyectado.

El esquema de procesamiento a realizar en la GPU es similar al efectuado en la primera etapa, donde se aplica un *kernel* para realizar todas las operaciones de multiplicación y suma. En

este caso, los coeficientes de las rotaciones se almacenan en la memoria compartida, porque por un lado ofrecen mejores resultados de rendimiento y por otro el volumen de datos de los filtros impide su almacenamiento en la escasa memoria de constantes.

En el algoritmo 9, implementado como un *kernel* CUDA, se observa el procedimiento llevado a cabo en esta etapa, que utiliza los datos de salida de la etapa II (ver sección 5.2.2) como entrada en esta (*basis*) y donde existe de nuevo un alto paralelismo a nivel de *pixel*. El resultado de esta etapa se almacena en *convresult*. Este código se ejecuta para cada uno de los ángulos proporcionados como parámetro de entrada del algoritmo McGM.

Algorithm 9 convresult = Orientación(basis, filtros_orientados, $spac_{filts}$, ángulo)

```
filts_shared=Mem_Compartida(filtros_orientados, ángulo)
for frame = 0 ; i ≤ all_frames ; ++frame do
  for temp = 0 ; temp ≤ temp_filts ; ++temp do
    for orden = 0 ; orden ≤ spac_filts ; ++orden do
      for kval = 0 ; kval ≤ orden ; ++kval do
        conv_result =Calcular_Orden_Ortogonal_0(basis,filts_shared,kval,frame,temp,order)
      end for
    end for
    for orden = 1 ; orden ≤ spac_filts ; ++orden do
      for kval = 0 ; kval ≤ orden ; ++kval do
        conv_result =Calcular_Orden_Ortogonal_1(basis,filts_shared,kval,frame,temp,order)
      end for
    end for
    for orden = 2 ; orden ≤ spac_filts ; ++orden do
      for kval = 0 ; kval ≤ orden ; ++kval do
        conv_result =Calcular_Orden_Ortogonal_2(basis,filts_shared,kval,frame,temp,order)
      end for
    end for
  end for
end for
```

5.2.4. Etapa IV - V: Desarrollo de Taylor y cálculo de coeficientes.

La implementación de las etapas IV y V del algoritmo McGM se llevan a cabo conjuntamente. En esta cuarta etapa se realizan dos tareas: la primera de ellas es calcular y almacenar los pesos que después se van a utilizar para el desarrollo de Taylor y la segunda de las tareas es construir el desarrollo de Taylor en sí.

La mayor desventaja en la implementación de esta etapa reside en el hecho de que el proceso conlleva varios sumatorios encadenados que, desde el punto de vista computacional, supone varias operaciones de reducción, que dificultan la explotación del paralelismo a nivel de datos.

Sin embargo, existen diferentes estrategias [195] que logran resultados relativamente satisfactorios en una GPU pese a la necesidad de introducir etapas de sincronización. En nuestro caso, con el fin de evitar la ineficiencia de la reducción, se ha explotado un paralelismo de grano más grueso, lo que permite ocultar algunas de las operaciones de reducción. El esquema de paralelización se basa en un particionado a nivel de *pixel*, que permite ocultar la sincronización de las operaciones de reducción.

Resumiendo, el *kernel* implementado explota el paralelismo a nivel de *pixel* ocultando la dependencia de datos en las operaciones de reducción.

Al igual que en la etapa anterior, el código mostrado en el algoritmo 10 se ejecuta para cada ángulo, manteniendo el paralelismo a nivel de *pixel*, donde nf es una variable que depende del número de ángulos, que se calcula como $\sqrt{\frac{2}{\alpha}}$. Los pesos que se utilizan para el cálculo en esta etapa se almacenan en memoria de constantes para obtener mejores resultados de rendimiento.

Algorithm 10 intp = EtapaIV_V(convresult, t_{ord}, y_{ord}, x_{ord}, ángulo, nf)

vectores st_1, st_2, st_3, st_4, st_5, st_6
Reservar_Mem_Compartida(st_1, st_2, st_3, st_4, st_5, st_6)
w = Leer_Pesos_de_Mem_Constantes();
for all $i = 0$; $i \leq$ all_frames ; $++i$ **do**
  for all $t = 0, y = 0, x = 0$; $t < t_{ord}, y < y_{ord}, x < x_{ord}$; $++t, ++y, ++x$ **do**
    st_1 $+= convresult[x] * convresult[t] * w[t, y, x]$ $\{XT^{\theta}\}$
    st_2 $+= convresult[t] * convresult[t] * w[t, y, x]$ $\{TT^{\theta}\}$
    st_3 $+= convresult[x] * convresult[x] * w[t, y, x]$ $\{XX^{\theta}\}$
    st_4 $+= convresult[y] * convresult[t] * w[t, y, x]$ $\{YT^{\theta}\}$
    st_5 $+= convresult[y] * convresult[y] * w[t, y, x]$ $\{YY^{\theta}\}$
    st_6 $+= convresult[x] * convresult[y] * w[t, y, x]$ $\{XY^{\theta}\}$
  end for
  intp[0] $= (st_1/st_2) * nf$
  intp[1] $= (st_1/st_3) * (st_6/st_3) * nf$
  intp[2] $= (st_4/st_2) * nf$
  intp[3] $= (st_4/st_5) * (st_6/st_5) * nf$
  intp[4] $= (st_6/st_5)$
  intp[5] $= (st_6/st_3)$
end for

5.2.5. Etapa VI: Cálculo de las primitivas de velocidad.

De acuerdo a la descripción realizada en la sección 3.7, en esta etapa se calculan las primitivas de la velocidad (módulo y ángulo) para cada *pixel* del fotograma. En esta etapa existe de nuevo un paralelismo a nivel de *pixel* que puede ser explotado, por lo que el particionado realizado es análogo al explotado en etapas previas.

Para concluir, el código representado en el pseudo-código 11 calcula la fase y el módulo para cada *pixel* de los fotogramas, haciendo uso de los datos obtenidos en las etapas anteriores, donde α representa el número de ángulos y cn y sn son vectores almacenados en memoria de constantes que tienen los valores del coseno y el seno para un ángulo dado.

Algorithm 11 [vel, ang] = Fase_Modulo(intp,α)

M[α], M$_o$[α]
top[2][2], bottom[2][2]
for all $i = 0$; $i < N_{frames}$; $++i$ **do**
  for all $th = 0$; $th < \alpha$; $++th$ **do**
    M[th]=$intp[0] * (-intp[1])$
    M$_o$[th]=$intp[2] * (-intp[3])$
    bottom[0][0] += $intp[0] * intp[1]$
    bottom[0][1] += $intp[1] * intp[2]$
    bottom[1][0] += $intp[3] * intp[0]$
    bottom[1][1] += $intp[3] * intp[2]$
    top[0][0] += $intp[0] * cn[th]$
    top[0][1] += $intp[1] * sn[th]$
    top[1][0] += $intp[3] * cn[th]$
    top[1][1] += $intp[3] * sn[th]$
  end for
  vel[i][x,y] = $\sqrt{\frac{top[0][0]*top[1][1]-top[0][1]*top[1][0]}{bottom[0][0]*bottom[1][1]-bottom[0][1]*bottom[1][0]}}$
  for all $j = 0$; $j < \alpha$; $++j$ **do**
    Mcn += M[i] * cn[i]
    Msn += M[i] * sn[i]
    M$_o$cn += M$_o$[i] * cn[i]
    M$_o$sn += M$_o$[i] * sn[i]
  end for
  ang[i][x,y] = $(180/\pi)$ * atan2((Mcn - M$_o$sn),(Msn + M$_o$cn)) + 90
end for

5.3. Resultados.

En esta sección se van a presentar las pruebas realizadas y los resultados obtenidos para la implementación del algoritmo McGM en GPU. En primer lugar, y con el fin de entender mejor los resultados recogidos, se presenta el entorno de trabajo en el que se han desarrollado las pruebas. A continuación, se explica el rendimiento obtenido en cada una de las etapas del algoritmo por separado, comparándose los resultados con los obtenidos en una implementación en CPU. Por último, se presentan los resultados globales para la implementación del algoritmo McGM en GPU.

5.3.1. Entorno de trabajo.

Los resultados han sido obtenidos haciendo uso de un sistema basado en la tecnología *Tesla*. Este sistema contiene dos procesadores Intel Xeon E5530 con 4 *cores* (2.40 GHz con 8MB de memoria caché y tecnología *Hyperthreading*) conectados a una GPU Tesla C1060. El sistema operativo que utiliza el entorno es Debian con el *kernel* 2.6.38. El compilador utilizado ha sido el *g++* de GNU en su versión v.4.5.2 con las opciones de compilación *-O3 -m64* y la versión de CUDA v.2.3. La tarjeta gráfica tiene la capacidad 1.3, lo que significa que posee 240 procesadores unificados que permiten la ejecución de *1024* hilos en cada multiprocesador. La jerarquía de memoria posee 4GB de memoria global, 16KB de memoria compartida y 64KB de memoria de constantes.

5.3.2. Resultados de rendimiento.

El objetivo principal es analizar el rendimiento y los beneficios que pueden aportar las arquitecturas de *hardware* gráfico aplicadas en algoritmos de estimación de movimiento.

Con respecto a la etapa del filtrado temporal, se han implementado cuatro alternativas dependiendo de en qué parte de la jerarquía de memoria de la GPU se han almacenado los datos:

- *Base*: Llamamos así a la primera implementación desarrollada donde los datos de la secuencia de entrada (todos los fotogramas) y el conjunto de filtros a aplicar se almacenan en memoria global. En el capítulo 3 se describe el proceso computacional de esta implementación.
- *Global*: En esta implementación la secuencia de datos se encuentra almacenada en memoria global pero los coeficientes de los filtros se almacenan en memoria de constantes.
- *Shared*: Se puede decir que es una aproximación híbrida, sólo los primeros N fotogramas que se van a procesar se almacenan en memoria compartida aunque los coeficientes de los filtros permanecen en memoria de constantes.
- *Shared-optimizada*: Los coeficientes de los filtros continúan almacenados en memoria de constantes y los N primeros fotogramas en memoria compartida. La diferencia con respecto a la implementación que se ha llamado *shared* es que se diseña una estructura de *buffer*. Esta estructura funciona de la siguiente manera: a) en el instante inicial ($t=0$) se encuentran almacenados en la memoria compartida los $N+1$ primeros fotogramas. b) En el siguiente instante ($t=1$) se sustituye el primer fotograma almacenado (el de índice 0) por el siguiente fotograma a procesar ($N+2$). c) Se repite el proceso hasta que se han procesado todos los fotogramas. De esta forma se reducen el número de accesos y copias en la memoria compartida. Además, se reutiliza la información de muchos fotogramas que se van a volver a procesar entre varios instantes.

En la figura 52 se muestran los resultados obtenidos para cada caso de estudio. Todos ellos se muestran con respecto a la implementación de esta etapa en CPU y variando el tamaño

de los datos de entrada (*size*) y el tamaño de la ventana para la convolución temporal (N=7, 9 y 15).

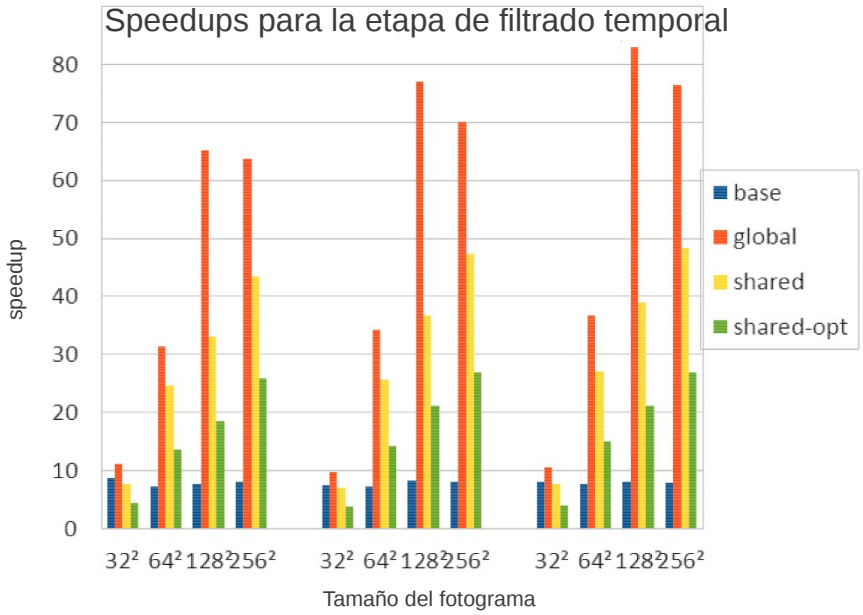


Figura 52: *Speedup* obtenidos en la etapa de filtrado temporal para varios tamaños de entrada y ventana de filtro (N)

Como se muestra en la figura 52, los *speedup* son notables para cada versión de esta etapa implementada en la GPU. Sin embargo, los mejores resultados se obtienen cuando los coeficientes de los filtros se almacenan en la memoria de constantes y los fotogramas en la memoria global (versión *Global*), alcanzando un rendimiento de $\times 85$.

Aunque a priori las versiones *Shared* y *Shared-optimizada* deberían mostrar mejores resultados, ya que la latencia de acceso a memoria compartida es menor que a memoria global, se puede explicar que esto no es así porque no hay implicadas suficientes operaciones ni reutilización de datos, por lo que debido al escaso número de datos a procesar, los datos se pueden mapear directamente sobre los registros del multiprocesador (16K registros de 32 bits). Por ejemplo, para $N = 15$, el *kernel* trabaja con 3840 elementos/bloque, menos de los 16K registros disponibles. Además, se puede observar en la figura 52 cómo el rendimiento se incrementa conforme el tamaño de problema aumenta. La explicación en este caso es sencilla; como el número de bloques CUDA es mayor, la ocupación de la GPU también será mayor.

A continuación se va a tener en cuenta la etapa de filtrado espacial. Se ha optado por un tamaño de ventana para los filtros temporales de N=*15* con el objetivo de poder comparar con la etapa anterior. Se ha analizado el efecto de modificar el tamaño de ventana para los filtros espaciales y el rendimiento de esta etapa teniendo en cuenta los tamaños para cada fotograma de los experimentos anteriores.

La figura 53 muestra los *speedup* obtenidos con cuatro órdenes para filtros espaciales y con diferentes valores para el tamaño de ventana (7, 9, 15 y 31). El orden de la derivada espacial no afecta al rendimiento, lo que es coherente con otros resultados observados. El rendimiento es mejor conforme el filtro crece ya que se explota mayor paralelismo en el cálculo de la convolución. Los rendimientos obtenidos alcanzan aceleraciones de hasta $\times 25$.

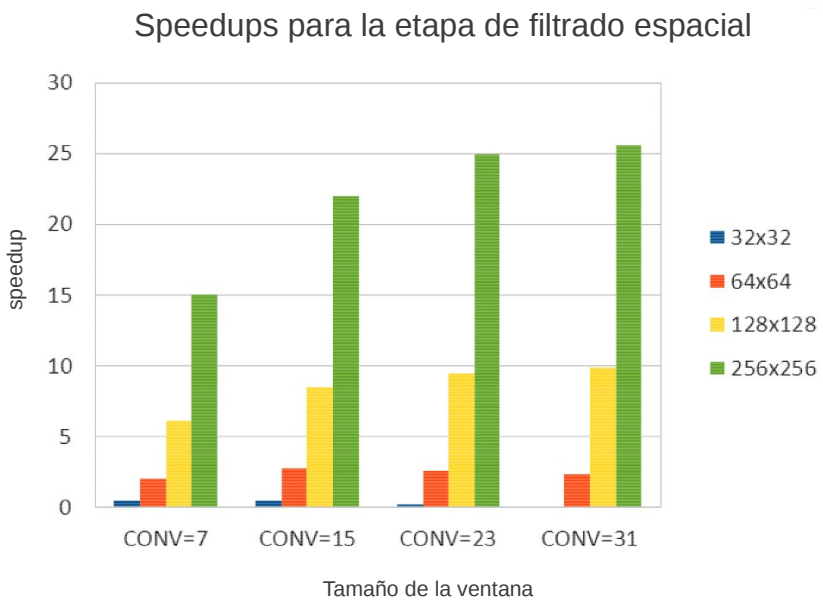


Figura 53: *Speedup* para la etapa de filtrado espacial para varios tamaños de fotograma y de ventana de filtro.

La siguiente etapa a analizar es la de orientación. El cómputo se basa en formar combinaciones lineales de todas las derivadas espaciales calculadas anteriormente. Por lo tanto, la operación principal a completar se hará teniendo en cuenta las derivadas espaciales y aplicándoles una función de ponderación. La información de entrada se ha almacenado en memoria global de la GPU.

El conjunto de datos de entrada se ha mapeado directamente en los registros de la arquitectura, al igual que en la etapa de filtrado temporal, ya que el volumen de datos no es demasiado grande. También hay que tener en cuenta que los pesos de las orientaciones se han almacenado en memoria compartida.

Los resultados obtenidos para diferentes tamaños de fotograma y distintas orientaciones se muestran en la figura 54, donde los ángulos considerados han sido 60^o, 30^o y 15^o que corresponden a los valores 6, 12 y 24 para las orientaciones, respectivamente. Hay que destacar que, salvo para tamaños pequeños de los fotogramas (64×64 *píxeles*), el número de ángulos apenas supone un impacto importante en el rendimiento final. La configuración que muestra

los mejores resultados sería aquella que utiliza 24 orientaciones (cada 15^o) y una resolución grande para los fotogramas (256×256) debido a que hay mayor paralelismo.

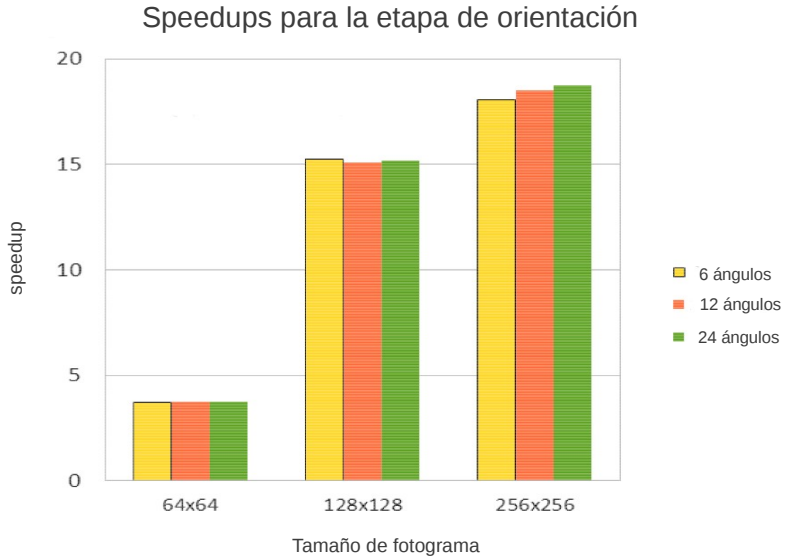


Figura 54: *Speedup* para la etapa de orientación para varios tamaños del fotograma y varias orientaciones.

Los resultados para la etapa del desarrollo de Taylor se muestran en la figura 55. Cabe destacar que en esta etapa se ha explotado el paralelismo a nivel de *pixel* con el objetivo de evitar etapas de sincronización asociadas a operaciones de reducción debido a que son costosas. Esto explica los resultados recogidos, obteniendo ratios de rendimiento desde $\times 5$ hasta $\times 38$ para resoluciones de 64×64 y 256×256 respectivamente.

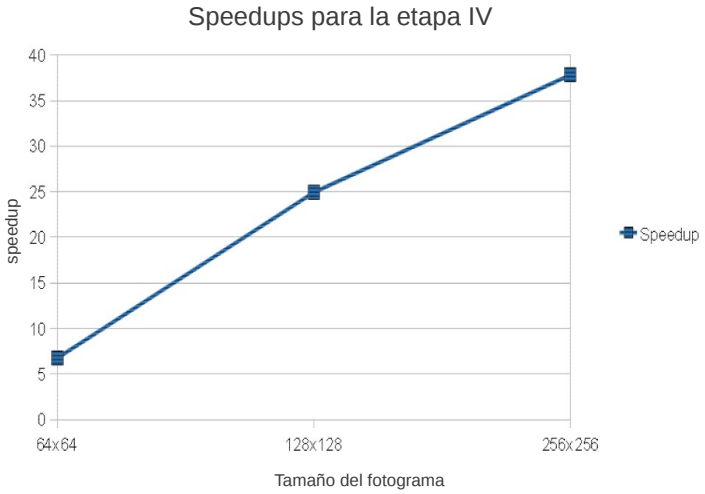


Figura 55: *Speedup* para la etapa del desarrollo de Taylor variando la resolución.

Para finalizar, los resultados obtenidos al extraer las primitivas de velocidad, que engloba a las etapas V y VI y donde se obtienen los resultados finales del módulo y fase, se exponen en la figura 56. En esta etapa se obtiene un rendimiento de $\times 23$ para un tamaño de resolución de 256×256.

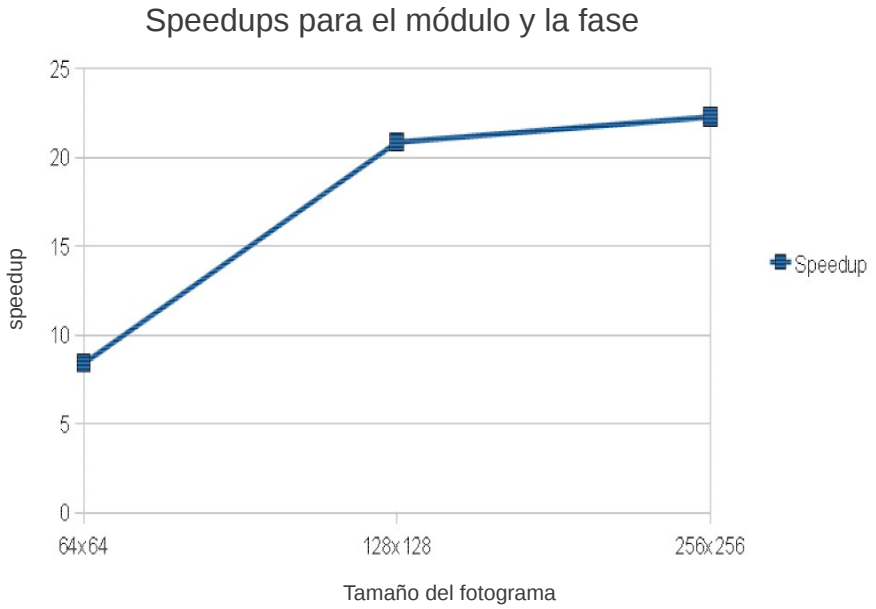


Figura 56: *Speedup* obtenido para las etapas de extracción de primitivas de velocidad teniendo en cuenta varias resoluciones.

En la tabla 11 se expone el rendimiento para cada etapa y el general para distintas resoluciones de los fotogramas. Para este experimento, la configuración del modelo inicial incluye 3 derivadas temporales y una ventana para la convolución temporal de 15 fotogramas (Etapa I), 5 derivadas espaciales y una ventana para la convolución espacial separable de 31 *pixels* (Etapa II), y 12 ángulos de dirección (Etapa III). La fila CPU se refiere a los resultados obtenidos en el sistema con un procesador Intel Xeon y haciendo uso de un solo *core*. La fila *Best* CPU hace referencia a la mejor configuración conseguida utilizando diversos *cores*. La explotación de paralelismo en este caso se ha realizado haciendo uso de OpenMP. Por último, la fila GPU muestra los resultados obtenidos en la GPU Tesla C1060.

Respecto a la eficiencia en un sistema *multicore*, hay que señalar que para una resolución pequeña, como la de 32×32, la explotación de múltiples *cores* no reporta gran beneficio, ya que sólo es rentable con dos *cores*. Esto se debe al escaso grado de paralelismo de datos disponible en esta resolución. Sin embargo, para otras resoluciones mayores, las aceleraciones obtenidas con múltiples *cores* son mayores, aunque no muy impresionantes. Aunque la configuración que ofrece mejores resultados corresponde a la de máxima utilización del sistema (16 hilos), su escalabilidad en términos de eficiencia es aún así pobre (con un promedio del 30 %). Este comportamiento proviene del hecho de que en el sistema *multicore* se comparten

algunos recursos críticos como el acceso al bus de memoria y algunos niveles de la jerarquía de memoria, que actúan como cuello de botella debido a la competencia por dichos recursos.

	Init. GPU (s/pixel)	Filt. Temporal (Mpixel/s)	Filt. Espacial (Mpixel/s)	Orientación (Mpixel/s)	Taylor (Mpixel/s)	Velocidad (Mpixel/s)	**Total etapas (Mpixel/s)**	**Total (fps)**
CPU (32^2)		10.30	21.63	90.99	217.87	247.24	**6.14**	**6327**
Best CPU (32^2)		13.20	12.48	130.75	369.79	289.62	**6.26**	
GPU (32^2)	3.28E-5	124.69	0.62	4.66	8.04	50.40	**0.50**	**375.7**
CPU (64^2)		12.85	1.44	2.09	3.78	20.14	**0.64**	**195.2**
Best CPU (64^2)		51.30	1.30	9.03	16.76	72.18	**1.66**	
GPU (64^2)	9.08E-6	495.98	2.55	15.77	25.54	169.08	**2**	**296.4**
CPU (128^2)		13.97	0.92	1.20	2.06	11.53	**0.39**	**30.72**
Best CPU (128^2)		70.87	0.77	5.04	14.23	66.59	**1.2**	
GPU (128^2)	7.21E-6	1166.12	8.79	36.17	51.49	240.65	**6.03**	**210.8**
CPU (256^2)		21.50	1.05	1.30	1.70	12.98	**0.41**	**8.631**
Best CPU (256^2)		154.75	1.11	3.89	7.95	93.27	**1.15**	
GPU (256^2)	2.25E-6	1724.63	27.56	47.62	64.20	289.21	**13**	**99.64**

Tabla 11: Comparación del rendimiento obtenido en la versión GPU frente a la versión CPU.

La segunda columna de la tabla 11 corresponde al coste de la inicialización de la GPU, que incluye: la inicialización del *hardware*, la reserva de memoria y la información enviada, medido todo ello en segundos/*pixel*. Dicha medida puede parecer insignificante, sin embargo, el impacto con respecto a todo el proceso no lo es, ya que supone entre un 7 % y un 15 % del tiempo total de ejecución. El resto de columnas, salvo la última, corresponden a cada una de las etapas analizadas, y los valores mostrados corresponden a *Mpixels*/segundo. Es importante destacar que la fase de orientación (Etapa III), es la más costosa de largo (alrededor de un 30 % del tiempo medio total) por lo que el rendimiento global estará limitado por esta etapa. Por último, la columna *Total etapas* muestra el rendimiento global de la implementación en GPU frente a la realizada en CPU.

La GPU ofrece un rendimiento bastante pobre para una resolución pequeña (32×32), viéndose superado por su homónimo CPU. Este hecho viene motivado por la pequeña cantidad de datos que hay que procesar, lo que favorece su procesado en la CPU, ya que se pueden almacenar todos los cómputos intermedios en el primer nivel de la memoria caché, ofreciendo una alta localidad de datos tanto espacial como temporal. Sin embargo, para resoluciones más grandes, donde el volumen de datos supone el reemplazo de bloques en todos niveles de la jerarquía de memoria de la CPU, encontramos que el rendimiento es peor, de forma que las versiones para GPU superan sobradamente el rendimiento ofrecido por la CPU. Hay que destacar que para las últimas pruebas realizadas, con una resolución de 256×256, se alcanzan 12 *Mpixel*/s, lo que significa que se cumplen sobradamente los requisitos para tiempo real, lográndose procesar hasta 185 fotogramas por segundo.

A modo de resumen, la tabla 12 pone de relieve las ventajas computaciones que ofrece el uso de aceleradores gráficos mostrando las aceleraciones obtenidas para las diferentes resoluciones contempladas en este estudio. Teniendo en cuenta el resultado total de todas las

etapas, se logran unas aceleraciones globales de hasta $\times 32$ para una resolución 256×256, esperando alcanzar mayores aceleraciones incluso para resoluciones mayores ya que el grado de paralelismo a explotar será mayor.

Resolución de la película	64×64	128×128	256×256
Speedup (config. normal CPU)	3.125	15.46	31.70
Speedup (mejor config. CPU)	1.195	5.03	11.27

Tabla 12: *Speedups* obtenidos comparando versiones de GPU y versiones de CPU.

5.3.3. Resultados visuales.

Para demostrar los resultados visuales del modelo completo se han utilizado los estímulos expuestos en el capítulo 2. En primer lugar, se ha realizado una prueba preliminar para la implementación GPU con el estímulo de entrada representado en la figura 13, donde se han tenido en cuenta los dos escenarios de entrada representados en ella. A continuación, se ha probado con el estímulo real (representa la explosión de un gas) expuesto en la figura 57, donde la parte baja de dicha figura muestra las dos salidas del algoritmo implementado (la fase a la izquierda y el módulo a la derecha), pudiéndose apreciar la obtención de movimiento realizada por el algoritmo.

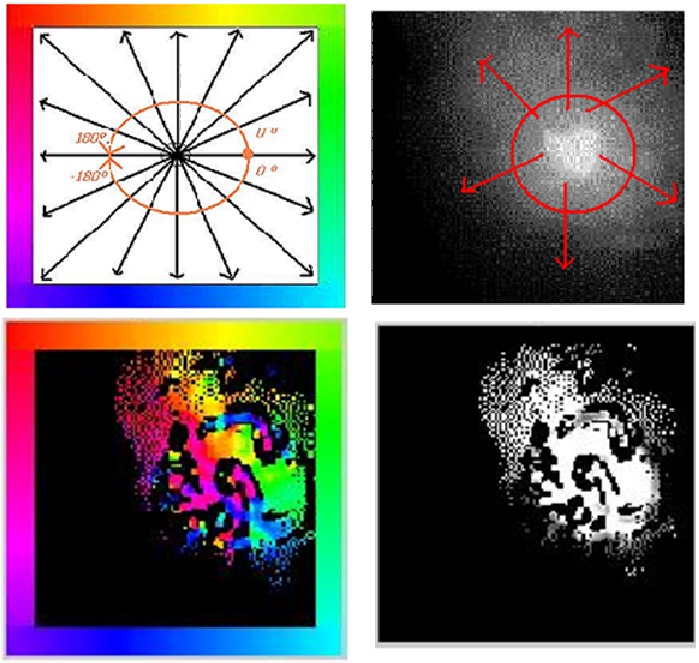


Figura 57: Estímulo de entrada. Salidas para el Módulo y Fase del algoritmo.

En esta figura 57, las líneas rojas denotan la naturaleza expansiva del movimiento en el fotograma original. La fase se muestra con un código de color en los límites del fotograma

y el módulo queda representado como un rango lineal intenso de escala de grises, donde el negro significa que no hay movimiento y el blanco una alta velocidad.

Se han tenido en cuenta también estímulos de entrada más complejos, como los representados en la figura 58, las secuencias sintéticas "*Diverging Tree*" y "*Translating Tree*" (explicados en el capítulo 2 y en la figura 14). Al igual que para el estímulo anterior, la figura 58 muestra en un código de color la fase y el módulo en la escala de grises. Estos resultados son los que se observan a la derecha de cada imagen de entrada, quedando patente en el caso del "*Diverging Tree*" la sensación de zoom (los colores tienden a los extremos) y para el caso del "*Translating Tree*" el movimiento de translación hacia la derecha de la imagen.

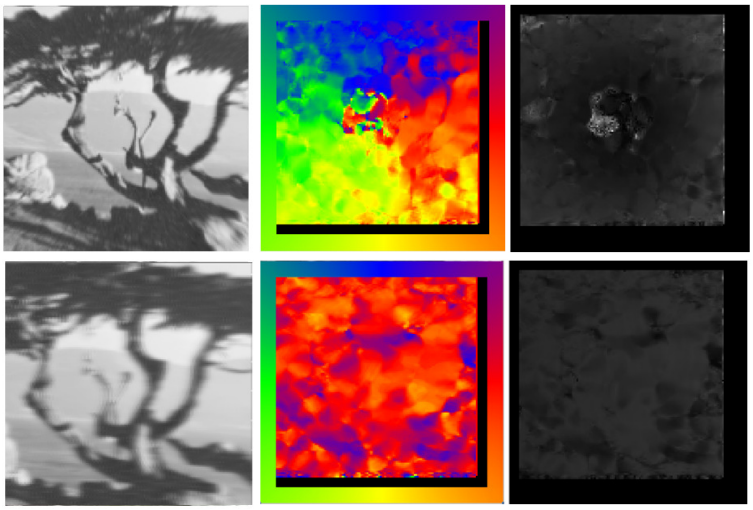


Figura 58: Secuencia del "*Diverging Tree*" (*arriba*) y del "*Translating Tree*" (*abajo*).

Por último se ha testeado el algoritmo desarrollado para hardware gráfico con el estímulo de entrada real como la secuencia "*Hamburg taxi*", representado en la figura 16. El resultado visual del movimiento puede observarse en la figura 59.

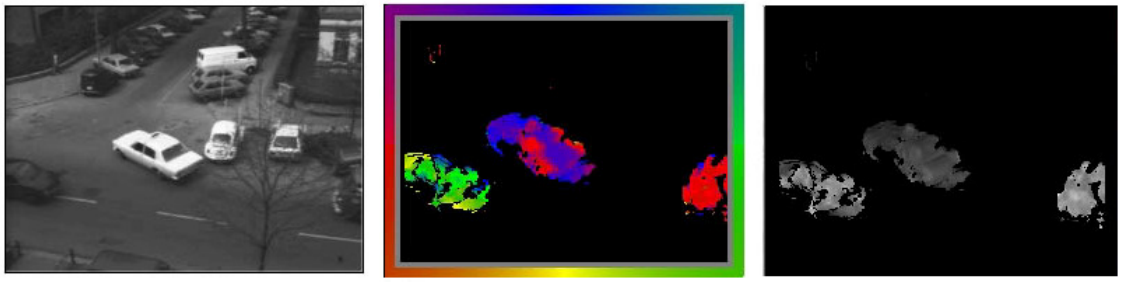


Figura 59: Estímulo real del taxi, de la secuencia "*Hamburg taxi*", y la fase y el módulo obtenidos con la implementación en GPU del algoritmo McGM.

Con el objetivo de cuantificar el error relativo en la implementación GPU, se hace necesario

usar una métrica como la de Barron, descritas en la ecuaciones (7, 8).

Esta medida del error se ha calculado para cada *pixel* para el cual se ha recuperado una medida de la velocidad o el error promedio calculado. El conjunto de parámetros utilizados para el algoritmo McGM ha sido $\sigma = 1{,}5$, $\alpha = 10$, $\tau = 0{,}25$, y una zona de integración $p, q, r = 11 \times 11 \times 11$ desde la expresión (14) a la expresión (17). Los errores medios obtenidos utilizando la métrica de Barron para los estímulos "Diverging Tree" y "Translating Tree" han sido $0{,}01$ y $0{,}40$ radianes, respectivamente.

Antes de comparar la implementación realizada con otras existentes en flujo óptico, se va a analizar el impacto de la variación del número de orientaciones en la estimación de movimiento para el algoritmo McGM. La figura 60 muestra el valor de velocidad media para un número de orientaciones determinado empleando el estímulo sintético de "*Translating Tree*". La posibilidad de obtener diferente información en cada uno de los diferentes canales orientados es una de las características que hacen robusto a este modelo. Se puede concluir que el sistema desarrollado es robusto con un número mínimo de orientaciones mayor a 18 porque la velocidad obtenida aparece estable e invariante.

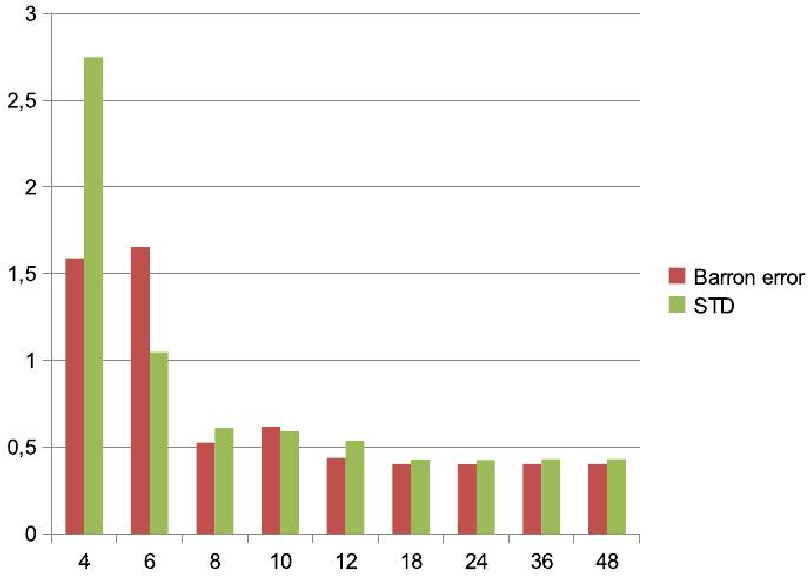


Figura 60: Velocidad medida vs. orientaciones.

5.3.4. Comparación con otras implementaciones de flujo óptico.

Los datos representados en la tabla 13 comparan la estimación de movimiento como resultado de aplicar otros algoritmos empleados en flujo óptico. El estímulo de entrada corresponde a un estímulo sintético de translación de una rejilla similar al de la figura 13, expuesto en el capítulo 2. La tabla muestra el error angular y la desviación estándar correspondiente a los algoritmos Horn & Schunck, Horn & Schunck modificado, Lucas & Kanade y nuestra

implementación desarrollada del algoritmo McGM en GPU en este trabajo.

Método	Error Angular	Desviación Estándar
Horn & Schunck [8]	0.97	2.62
Modified Horn & Schunck [197]	0.73	0.94
Lucas & Kanade [6]	2.47	0.16
McGM (GPU versión) [169]	0.40	0.43

Tabla 13: Errores (en grados) en el estímulo de la translación de una rejilla.

Empleando como estímulos de entrada los conocidos "*Diverging Tree*" y "*Translating Tree*", la implementación McGM en GPU produce una media de diferencias en magnitud de $0{,}291$ y un error angular de $12{,}47° \pm 18{,}51$. Si se añade un umbral en el valor de la primera derivada temporal, se puede llegar a disminuir este error hasta $11{,}29°$ para una densidad del 64 %. Algunos de los mejores resultados publicados de otros algoritmos para esta secuencia han sido realizados con versiones donde el umbral de la densidad de salida era bajo. Por tanto, el algoritmo McGM en su versión GPU es consecuente con estos resultados y estima la velocidad de todas las regiones con la misma precisión. Además, el umbral no mejora de forma significativa los resultados, habiendo logrado un implementación robusta ante diferentes condiciones en el entorno. Queda demostrado que la velocidad calculada es prácticamente uniforme pero la dirección varía, obteniendo de esta forma valores altos del error. Por último, la tabla 14 muestra una media del error angular en grados.

	'Diverging Tree'		'Translating Tree'	
	Error Angular	Desviación Estándar	Error Angular	Desviación Estándar
Horn & Schunck [8]	12.77	12	38.99	27
Uras et al. [198]	5.75	4.01	0.71	0.81
Lucas & Kanade [6]	8.07	11	11.69	18
Fleet & Jepson [163]	1.24	0.9	0.36	0.41

Tabla 14: Resultados para los estímulos '*Diverging Tree*' y '*Translating Tree*'.

5.3.5. Extensión a sistemas empotrados basados en GPU.

Con el fin de completar el estudio de viabilidad de la implementación del algoritmo McGM en *hardware* gráfico, se ha considerado su portabilidad a un sistema empotrado de alto rendimiento y energéticamente eficiente. Para ello se ha escogido la arquitectura CARMA[75], ya que cumple estas características.

[75] `http://www.nvidia.com/object/seco-dev-kit.html`

CARMA es una plataforma de desarrollo cuya arquitectura se basa en ARM con CUDA, creada para soportar el alto crecimiento de iniciativas de computación energéticamente eficientes. Posee una CPU (NVIDIA Tegra 3 ARM Cortex A9 Quad-Core) con 2GB de memoria y una GPU (NVIDIA QUADRO 1000M con 96 CUDA *cores* y capacidad de computo 2.1) con otros 2 GB de memoria. Presenta un rendimiento de 270 GFlops en precisión simple.

La creciente demanda de energía y el aumento de los costes están obligando al mercado de la computación de alto rendimiento (HPC) a encontrar soluciones más eficientes en términos de energía. Además, los desarrollos están orientados cada vez más a arquitecturas híbridas capaces de combinar núcleos ARM con avanzados aceleradores GPU. Gracias a las arquitecturas híbridas basadas en ARM, es posible paralelizar una aplicación reduciendo al menos un orden de magnitud el consumo de energía en comparación con las arquitecturas basadas en x86. La plataforma CARMA ayudará a los desarrolladores, investigadores y diseñadores de soluciones a construir sistemas simples, económicos, eficientes y escalables.

La escalabilidad es una de las características que hacen de CARMA una plataforma ideal para el desarrollo, no sólo independiente, sino también para las implementaciones en clúster de investigación.

	Filt. Temporal (Mpps.)	Filt. Espacial (Mpps.)	Orientación (Mpps.)	Taylor (Mpps.)	Velocidad (Mpps.)	**Total etapas (Mpps.)**	**Total etapas (fps.)**
GPU CARMA (32^2)	113.77	0.052	4.19	19.91	55.65	**0.05**	**48,82**
GPU CARMA (64^2)	338.51	0.42	3.76	15.06	85.34	**0.37**	**90,33**
GPU CARMA (128^2)	487.04	1.39	3.19	9.73	63.38	**0.87**	**53,1**
GPU CARMA (256^2)	513.77	2.97	2,87	8,14	53,42	**1,2**	**18,31**

Tabla 15: Rendimiento obtenido en la arquitectura CARMA para cada una de las etapas.

Los resultados obtenidos de portar el algoritmo a esta arquitectura se muestran en la tabla 15. La configuración escogida es similar al caso anterior, con 50 fotogramas de tamaño variable, entre 32×32 y 256×256, 3 filtros temporales con una ventana de tamaño igual a 15 *pixels*, 6 filtros espaciales de 31 *pixels* y 12 ángulos. En dicha tabla pueden observarse que los valores obtenidos, medidos en Mpixels/s, siguen la misma línea que los observados en la tabla 11, pudiéndose comprobar cómo las etapas que más tiempo y recursos consumen son la segunda y la tercera (la de filtrado espacial y la de orientación de filtros). Cabe destacar que los resultados son peores si los comparamos con los obtenidos en el experimento anterior, debido a que esta arquitectura es menos potente y más limitada en recursos. A su favor tiene que es portable, por lo que podría ser útil en determinadas aplicaciones, incorporando este sistema, por ejemplo, en un robot.

5.4. Conclusiones.

Se ha presentado una implementación en GPU eficiente para un algoritmo neuromórfico con una serie de características adecuadas para entornos complejos o ruidosos, que podría ser

aplicada en campos como la robótica, navegación, seguridad, vigilancia, etc. Además, este sistema también se puede utilizar en el campo de la investigación neurocientífica, extendiendo la complejidad de algoritmos bio-inspirados.

La utilización de *hardware* gráfico como un co-procesador para flujo óptico puede ser una elección asequible e interesante, con la que se pueden lograr *speedups* de $\times 32$. Sin embargo, el coste de intercambiar información entre la CPU y la GPU y la administración de la memoria de la GPU pueden reducir la aceleración ligeramente. Hay que agregar que, para obtener ventaja de la computación paralela de la GPU, es crucial que se distribuyan estas tareas de una manera adecuada y que, adicionalmente, se exploten de forma eficiente la compleja jerarquía de memoria, lo que no resulta una tarea sencilla en muchos casos.

El estudio presentado se ha centrado en la evaluación del rendimiento de la implementación del algoritmo McGM en GPU, sin embargo, debido a la naturaleza expansiva de este, a medida que se avanza a través de las etapas, los requisitos de memoria aumentan. La configuración que se ha tenido en cuenta y que tiene mayor consumo utiliza 3.5GB de memoria, un valor cercano al limite de en una GPU. Aunque hoy en día la capacidad de la memoria de las GPU aumenta, este problema persistirá para tamaños mayores de tamaños de fotogramas (mayor resolución). Debido a estas limitaciones, parece interesante estudiar mecanismos que reduzcan la cantidad de datos y diseñar estrategias para explotar paralelismo entre varias etapas, como analogía con un procesador segmentado.

Se debe enfatizar que, la estimación de movimiento para sistemas bio-inspirados en plataformas GPU es una alternativa a considerar en términos de *Mpixel*/segundo, especialmente comparado con otros sistemas específicos utilizados para este tipo de algoritmos. Además, las tendencias actuales parecen indicar que estos *ratios* aumentarán a medida que la tecnología futura mejore.

6. Técnicas de reducción en el consumo de memoria del algoritmo McGM.

6.1. Introducción.

Como se ha visto a lo largo de este trabajo, las GPUs ofrecen un alto rendimiento y eficiencia energética para un número considerable de aplicaciones que hacen uso del paralelismo de datos. En el capítulo 5 se pudo ver cómo la versión en GPU del algoritmo McGM alcanzaba un *speedup* de $\times 32$ usando esta tecnología. Sin embargo, conforme se van realizando las operaciones de procesamiento de señal, el consumo de memoria que se requiere crea cuellos de botella que pueden influir en el rendimiento de la solución. En este capítulo se presenta una mejora para reducir este problema, ya que limita la viabilidad a la hora de estimar el movimiento utilizando este tipo de aceleradores. El objetivo es encontrar una solución de compromiso entre el consumo de los recursos, la precisión alcanzada en la estimación de movimiento y un uso eficiente del paralelismo. Para ello se ha desarrollado un esquema con varios niveles de paralelismo, donde el nivel más fino se vincula al paralelismo de datos, y el más grueso se alcanza gracias a la explotación de un sistema con múltiples GPUs (multi-GPU).

La estimación y la compensación de movimiento son cruciales para la codificación multimedia, que se caracteriza por la demanda de una gran cantidad de memoria, y por realizar operaciones complejas desde el punto de vista computacional. A modo de ejemplo, en el proceso de compresión de un vídeo en formato MPEG, la estimación de movimiento es la parte que más tiempo consume [199], pudiendo alcanzar hasta un 90 % del tiempo total de ejecución [200, 201].

Por otro lado, en el ámbito de este tipo de algoritmos, el uso de las GPUs como aceleradores está limitado por restricciones relacionadas con la memoria DDR. En la literatura podemos encontrar soluciones a este este problema. Mattes [202] propone reutilizar datos con el objetivo de minimizar el tráfico de memoria entre la GPU y la CPU. Otros abordan esta problemática proponiendo técnicas basadas en compresión de información [203] para conseguir algoritmos más eficientes en términos de consumo de memoria.

En nuestro caso, el objetivo principal que se desea abordar en este capítulo es la construcción de un sistema de estimación de movimiento inteligente, con capacidades de auto-adaptación que, sin abandonar los requisitos de tiempo real, minimice el consumo de recursos sin que se vea afectada la calidad de precisión en el proceso de estimación.

6.2. Optimización y algoritmos genéticos.

En el capítulo 5 se ha estudiado la posibilidad de utilizar los sistemas GPUs y los beneficios potenciales que representan estos dispositivos en contextos donde se aplica el algoritmo McGM. Como se ha observado, se alcanzan *speedups* de $\times 32$ para películas con una resolución de 256^2 *pixels* por fotograma. Por tanto, cabe destacar que este sistema, basado

en un esquema de estimación de movimiento haciendo uso de GPUs, es una alternativa a considerar en términos de *Mpixel*/s. en comparación con otros sistemas. Sin embargo, las características de este algoritmo crean un cuello de botella debido a que los requisitos de memoria aumentan en cada etapa. Esta desventaja limita la viabilidad de usar GPUs con el propósito de estimar movimiento. Si tenemos en cuenta la configuración que más memoria usa en el estudio realizado en el capítulo 5, se consumen 3.5GB de memoria global, que está cerca del límite de una sola GPU. Aunque la capacidad de memoria en GPUs es mayor cada día, este problema se puede seguir encontrando conforme aumente la resolución de las películas a procesar.

Por tanto, el objetivo es explorar mecanismos que permitan reducir la cantidad de datos sin perder los requisitos de eficiencia ni de precisión. Para resaltar el problema de la memoria en GPU, nos referimos a los resultados que ya mostramos en la tabla 11, donde se presentaba un resumen de los resultados del rendimiento obtenido usando el algoritmo McGM en un dispositivo gráfico. Cabe agregar que la implementación en GPU cumple ampliamente los requisitos de tiempo real en todas las configuraciones de resolución consideradas. Mientras los procesadores de propósito general sólo pueden alcanzar *ratios* de tiempo real en vídeos con resoluciones pequeñas, los sistemas basados en GPU permiten mayores resoluciones de películas donde hay disponible mayor capacidad de memoria DDR.

6.2.1. Optimización multiobjetivo.

Los problemas de optimización intentan localizar una solución capaz de representar el valor óptimo para una función objetivo. Sin embargo, en muchas ocasiones, nos enfrentamos a problemas que requieren la optimización simultánea de más de un objetivo, lo que se conoce como las optimizaciones multiobjetivo [204, 205]. En términos matemáticos se puede expresar un problema de optimización multiobjetivo de la siguiente forma:

$$\begin{aligned} Minimize\ (f_1(x), f_2(x), ..., f_k(x))^T \\ subject\ to\ x\ \epsilon\ X \end{aligned} \tag{25}$$

en la que $K \geq 2$ es el número de objetivos y X es el conjunto factible de los vectores de decisión definidos por las funciones de restricción. Se dice que un elemento $x^*\epsilon X$ es una solución factible y al vector que contiene los elementos factibles se denomina vector objetivo. Como es de esperar, en las optimizaciones multiobjetivo no suele existir una solución que minimice todas las funciones objetivo de manera simultánea, por ello se presta especial atención a las soluciones óptimas de Pareto [206], es decir, soluciones que no se pueden mejorar en ninguno de los objetivos sin perjudicar al menos a uno del resto de objetivos. Por tanto, se dice que una solución factible $x^1\epsilon X$ domina a otra $x^2\epsilon X$ si:

1. $f_i(x^1) \leq f_i(x^2)$ para todos los valores de $i\epsilon 1, 2, ..., k$ y
2. $f_j(x^1) < f_j(x^2)$ para al menos un índice $j\epsilon 1, 2, ..., k$

Cuando, dada una solución $x^1 \epsilon X$ no existe otra que la domine se la denomina óptimo de Pareto. Al conjunto de estas soluciones se las conoce con el nombre de frente de Pareto[76].

Por consiguiente, en nuestro caso nos proponemos buscar solución a una optimización con tres objetivos a minimizar: el tiempo de ejecución, la utilización de memoria y la pérdida de precisión.

6.2.2. Algoritmos genéticos.

Los algoritmos genéticos [207] reciben su nombre debido a que se inspiran en la evolución biológica y su base genético-molecular, haciendo evolucionar una población de individuos mediante acciones aleatorias (mutaciones y recombinaciones genéticas) y selecciones respecto a un criterio, con el objetivo de decidir cuáles son los individuos más aptos, que se irán manteniendo, y los menos aptos, que se van a ir descartando. Un algoritmo genético es un método de búsqueda dirigida basado en probabilidad, es decir, al aumentar el número de iteraciones, la probabilidad de tener el óptimo de una población tiende a uno.
La manera de proceder de un algoritmo genético es la siguiente:

1. Se genera aleatoriamente una población inicial, construida por un conjunto de propiedades (cromosoma o genotipo) que representan las posibles soluciones al problema.

2. Se evalúa cada uno de los cromosomas de la población aplicándole una función de aptitud para saber cómo de buena es dicha solución.

3. Una vez que se sabe la aptitud de cada cromosoma, se eligen los que serán cruzados en la siguiente generación, en la que existe mayor probabilidad de escoger a los cromosomas con mejor aptitud.

4. En la recombinación se opera sobre dos cromosomas para generar individuos con las características combinadas de sus cromosomas padre.

5. También se procede a una mutación, o modificación al azar, del cromosoma de los individuos para permitir abarcar un mayor espacio de búsqueda.

6. Una vez que se han cruzado y mutado los individuos, se procede a seleccionar a los mejores, que formarán parte de la siguiente generación.

7. El algoritmo se detendrá cuando se alcance una solución óptima. Bien por requisitos o porque en ocasiones no existe o se desconoce, se suelen utilizar otros criterios de parada para el algoritmo, como son detenerlo pasado un número determinado de iteraciones o cuando no se produzcan cambios en la población.

Así, la idea es utilizar un algoritmo genético para minimizar los tres objetivos descritos anteriormente y poder obtener una solución perteneciente al frente de Pareto.

[76] `http://en.wikipedia.org/wiki/Pareto_front`

6.2.3. Aplicación en el McGM.

Para reducir el consumo de memoria, una solución que se podría emplear sería no almacenar ciertos cómputos intermedios, volviendo a calcularlos cuando fuese necesario. Esta propuesta conlleva implícitamente reducir el rendimiento siendo únicamente válida bajo condiciones de tiempo real. Las etapas del algoritmo McGM que mayores requerimientos de memoria conllevan son la de filtrado espacial y la de orientación (explicadas en las secciones 3.3 y 3.4 respectivamente), tal y como se ha puesto de manifiesto anteriormente. Por un lado, una solución rápida y eficiente para reducir las necesidades de memoria es realizar la etapa de orientación con menos θ ángulos, efecto que conlleva una pérdida de precisión. Por otro lado, es posible usar derivadas numéricas [208], en lugar de las gaussianas, durante la etapa de filtrado espacial porque precisarán menos operaciones, permitiendo recalcular las derivadas de manera más rápida cuando fuese necesario. Esta alternativa se basa en el hecho de que, al no tener que almacenar los datos de los cálculos intermedios, se reduce gran cantidad de memoria pudiendo recalcular estos datos cuando sean necesarios. Esta alternativa se fundamenta en las propiedades conmutativas de la convolución ,particularizada para la etapa del filtrado espacial: $I \otimes G_x = I \otimes (G_x \otimes D_x) = (I \otimes G_0) \otimes D_x$. El número de operaciones realizadas en $(I \otimes G_0) \otimes D_x$ es menor que su homónimo en el filtrado con derivadas gaussianas, lo que hace que la convolución se realice de forma más rápida.

Con el objetivo de evaluar la degradación en la precisión, la tabla 16 muestra el error obtenido calculando $G_0 \otimes D_x$. La degradación del filtro se evalúa mediante la diferencia $|(G_0 \otimes D_G) - (G_0 \otimes D_N)|$, donde D_G y D_N corresponden respectivamente a los filtros Gaussianos y Numéricos. Como se puede apreciar, la pérdida de precisión no es muy elevada en filtros con 9-31 *pixels*, obteniendo una media del 3 % en el error. Así pues, gracias a este estudio preliminar se puede afirmar que, acorde a los resultados encontrados, las derivadas basadas en filtrados numéricos a penas introducen errores considerables siendo más eficientes desde el punto de vista del número de operaciones que conllevan.

	x'	x''	x'''	x^{IV}	x^{V}
Degradación filtro	0.003825	0.009415	0.018444	0.033701	0.060134

Tabla 16: Degradación en la precisión usando derivadas numéricas en lugar de gaussianas para las derivadas desde el primer hasta el quinto orden.

A pesar de que la degradación en la precisión con filtros numéricos no es grande, se han llevado a cabo experimentos para evaluar la pérdida de precisión en el conjunto del algoritmo McGM completo. Para evaluar la degradación del mismo, se han utilizado las secuencias sintéticas "*Diverging Tree*" y "*Translating Tree*" descritas en el capítulo 2. Como métrica para evaluar los errores se ha empleado la de Barron, ya que es una de las métricas más aceptadas dentro de la literatura especializada.

En la tabla 17 se muestran los errores obtenidos cuando, para el filtrado espacial, se utilizan derivadas numéricas en lugar de gaussianas. Las columnas $O(h)$, $O(h^2)$, $O(h^3)$ y $O(h^4)$ denotan el error observado cuando el filtrado espacial se realiza con derivadas numéricas en

el primer, segundo, tercer y cuarto orden de precisión. Además, $\#\theta$ significa el número de θ ángulos proyectados en la etapa de orientación.

	$O(h)$	$O(h^2)$	$O(h^3)$	$O(h^4)$	$\#\theta/2$	$\#\theta/4$
Diverging Tree	0.9297	0.4982	0.4432	0.4020	0.0008	0.0122
Translating Tree	1.5185	0.7965	0.7762	0.6903	0.0015	0.0296

Tabla 17: Degradación total medida como error medio absoluto del ángulo Barron.

Como puede observarse, el experimento realizado con la secuencia "*Diverging Tree*" se comporta razonablemente bien con las derivadas numéricas, reduciendo el impacto con un orden más alto de precisión. Sin embargo, en el caso de la secuencia "*Translating Tree*" el algoritmo es más vulnerable con las derivadas numéricas que con la variación en el número de ángulos. Debido a esta disparidad observada en la tabla 17 es aconsejable explorar el espacio de soluciones factibles con cualquier combinación de derivadas numéricas y número de ángulos. Por este motivo, puede ser interesante la utilización de algoritmos genéticos para dicha exploración debido al gran número de parámetros a configurar, por un lado los relativos al algoritmo McGM en sí, y por otro los relacionados con la demanda de recursos.

6.3. Descripción de la optimización con algoritmos genéticos.

El uso de algoritmos genéticos surge a partir de la no viabilidad de explorar un espacio de soluciones amplio. En este contexto, el objetivo es encontrar un compromiso para reducir el uso de memoria en GPUs sin perder precisión, de forma que se permita un sistema de estimación de movimiento con capacidad de auto-adaptarse a cambios en las condiciones del entorno y en un tiempo razonable, es decir, que sea capaz de dar una respuesta en tiempo real.

El propósito de la optimización con varios objetivos [209] es optimizar de manera simultánea varios objetivos que podrían ser inconsistentes entre sí. Teniendo en cuenta estos problemas, se han de destacar también algunas ventajas y desventajas entre las diferentes variables involucradas. Los 3 objetivos a minimizar considerados en el contexto de este trabajo se exponen en la ecuación 26:

$$\begin{aligned} Minimize\ z = (f_1(x), f_2(x), f_3(x)) \\ subject\ to\ x\ \epsilon\ X \end{aligned} \tag{26}$$

Donde **z** es el vector objetivo que contiene los 3 objetivos a minimizar: tiempo de ejecución (f_1), uso de memoria (f_2) y pérdida de precisión (f_3). **z** es el vector de decisión y X es la región factible en el espacio de decisión, que corresponde a todas las configuraciones del algoritmo McGM, con respecto a la decisión de realizar las derivadas numéricas o gaussianas, y al número de ángulos involucrados. En la terminología de los algoritmos genéticos, a la x se la conoce como cromosoma. Así, en este trabajo:

- D_x corresponde a la derivada a utilizar en el filtrado espacial. Esta información se encuentra almacenada en un *array* bi-dimensional cuyos valores determinan si se van

a utilizar las derivadas gaussianas o las numéricas. Las posiciones de este array bi-dimensional están relacionadas con el orden de la derivada.

- θ es el número de ángulos en la etapa de orientación, determinado por un número entero.

6.4. Implementación en GPU.

A lo largo de los últimos años se han desarrollado un gran número de algoritmos evolutivos multi-objetivo [210, 211, 212]. En el tutorial [213] puede encontrarse una revisión de este tipo de algoritmos, pudiéndose observar cómo los autores proporcionan una revisión de las características más relevantes. Entre los algoritmos evolutivos multi-objetivo más empleados se encuentra el algoritmo elitista de ordenamiento no dominado del inglés *nondominated sorting genetic algorithm*(NSGA). Sin embargo este tipo de algoritmo presentan una serie de limitaciones entre las que destacamos:

1. Altos requerimientos computacionales en la ordenación de las soluciones no-dominadas.
2. La ausencia del elitismo [214].
3. La necesidad de introducir un parámetro [215] para determinar las soluciones similares que han de ser desechadas.

Por este motivo la comunidad científica ha propuesto una variante más moderna manteniendo las ventajas de dicha aproximación: NSGA-II [216]. En esta tesis se ha utilizado el NSGA-II por poseer las siguientes características:

- No es necesario estudiar el impacto de las funciones fitness $f_i(x)$ y por lo tanto no es necesario asignar los pesos.
- En la fase de ordenación requiere menor complejidad computacional.
- Tiene un buen comportamiento y destaca por su habilidad de encontrar un conjunto de soluciones cercanas al óptimo de Pareto en un número pequeño de iteraciones.
- Es un algoritmo muy utilizado y por lo tanto ha sido probado exitosamente.

El algoritmo NSGA-II está basado en un procedimiento de selección rápida no-dominada donde se lleva a cabo una estimación rápida de la distancia al objetivo involucrado, haciendo uso de un operador de comparación [216]. El algoritmo se puede resumir en los siguientes pasos:

1. Se crea una población de manera aleatoria inicialmente.
2. Esta población se ordena siguiendo el esquema de no-dominación.
3. Se asigna una aptitud (*fitness*) que significa que cada individuo de la población ocupa un puesto en algún nivel, siendo el primer nivel (o frontera de Pareto) el más deseable.

4. Se realiza una selección mediante torneo binario además de una combinación.

5. Se lleva a cabo la fase de mutación.

6. Se obtiene una población R que proviene de combinar la población antigua con la nueva obtenida. El tamaño de esta población combinada es de 2 veces el tamaño de la población inicial.

7. Dependiendo de los valores obtenidos con el algoritmo McGM, se reorganizan en niveles los individuos de la población R y se ordenan con respecto al esquema de no-dominación.

8. Se obtiene una nueva población con el tamaño de la inicial.

La parte más costosa en términos computacionales del algoritmo genético es la ordenación rápida no-dominada, ya que conlleva clasificar cada individuo de la población. Por este motivo, esta tarea se lleva a cabo en un sistema multi-GPU ya que es más eficiente que realizarlo en una CPU, desde el punto de vista computacional. Sin embargo, la mayoría de las operaciones del algoritmo genético se realizan en CPU debido a que su demanda computacional es baja.

Clasificar a un individuo de la población conlleva ejecutar el algoritmo McGM con la configuración que indica el cromosoma (qué tipo de derivación, gaussiana o numérica, se va a realizar en la etapa de filtrado espacial y cuántos ángulos se van a utilizar en la etapa de orientación). Cabe destacar que se han explotado varios niveles de paralelismo: el nivel más grueso, donde la ordenación no-dominada se lleva a cabo en las GPUs, y el nivel más fino, el disponible en cada etapa del algoritmo McGM. El algoritmo 12 resume la implementación paralela donde el tamaño de la población (*pop_size*), el número de generaciones (*ngens*) y la probabilidad de mutación (*%mutation*) son parámetros de entrada del algoritmo genético. Se ha utilizado *OpenMP* para distribuir la ordenación no-dominada entre los distintos dispositivos disponibles (con la directiva *#pragma omp parallel*). Esta implementación genera soluciones óptimas de Pareto con un conjunto de estimaciones de movimiento teniendo en cuenta el tiempo de ejecución, la precisión del error a nivel de *pixel* y el uso de memoria de la GPU. Esta característica permite elegir una de las mejores soluciones, teniendo en cuenta los recursos computacionales disponibles, favoreciendo de esta manera ajustes dinámicos dependiendo de las condiciones actuales.

Algorithm 12 pareto_front = multiGPU_NSGAII(pop size, ngens, %mutation)

$pop = random\_init\_Population(pop\_size)$ %NSGAII - 1^{st} stage
$Fronts = McGM\_fast\_nondominated\_sort(pop)$ %NSGAII - 2^{nd} & 3^{rd} stages
while $gen < ngens$ **do**
  #pragma parallel for shared(Fronts,R) private(i,GPU_{th},p_1,p_2)
  for all $i = 0; i \leq pop\_size; i = i + 1$ **do**
    $p_1, p_2 = select\_parents(pop(i))$
    $newpop(i).x = crossover(p_1, p_2)$ %NSGAII - 4^{th} stage
    $newpop(i).x = mutation(\ \%muration)$ %NSGAII - 5^{th} stage
    $R(i) = newpop(i) \bigcup pop(i)$ %NSGAII - 6^{th} stage
    $GPU_{th} = threadID();$
    $Fronts = McGM\_fast\_nondominated\_sort(GPU_{th}, R(i))$ %NSGAII - 7^{th} stage
  end for
  $k = 0$
  $pop = 0$
  %NSGAII - 8^{th} stage
  while $sizeof(pop) < pop\_size$ **do**
    $front_k = get\_fronts(Fronts, k)$
    $pop = pop \bigcup get\_Relements\_in\_front(R, front_k)$
    $k = k + 1$
  end while
  $gen = gen + 1$
end while
$pareto\_front = get\_front(Fronts, 0)$

6.5. Resultados.

En este apartado se muestran los resultados obtenidos en un entorno multi-GPU, aplicando el algoritmo genético NSGA-II multi-objetivo para encontrar una solución óptima que permita ejecutar el algoritmo de estimación de movimiento McGM en GPU, de manera que sea posible reducir el consumo de memoria provocado en las etapas más costosas, sin perder precisión en los resultados y manteniendo el requisito de obtener estos en tiempo real. Para ello, primero se describe el entorno en el que se han desarrollado las pruebas para después mostrar los resultados obtenidos, tanto a nivel del algoritmo genético con 3 objetivos, como a nivel del algoritmo McGM con las configuraciones óptimas obtenidas gracias al NSGA-II.

6.5.1. Entorno de pruebas.

Los sistemas GPU utilizados están basados en la tecnología *Tesla*. El primero de ellos contiene dos procesadores *Intel Xeon* E5645 con seis *cores* (2.40 GHz con 12MB de memoria caché y tecnología *Hyper-threading*) y dos GPUs *Tesla* M2070. El segundo sistema está equipado con dos procesadores *Quad Intel Xeon* E5530 (2.40 GHz con 8MB de memoria caché y tecnología *Hyper-threading*) conectados a cuatro GPUs *Tesla* C1060. En ambos casos el sistema

operativo es Debian con el kernel v.2.6.38, el compilador utilizado es el g++ de GNU versión v.4.4 con los *flags* de compilación *-O3 - m64 -fopenmp* y el SDK de CUDA C/C++ versión v.4.2 con los *flags -O3 -fopenmp -arch sm_20/13* activos.

El sistema basado en *Tesla* M2070 incorpora la tecnología *Fermi*, pero debido al escaso número de dispositivos disponibles, el estudio de escalabilidad se ha completado con un sistema con 4 *Tesla* C1060, lo que permite realizar proyecciones de las tasas de eficiencia paralela en sistemas más modernos.

6.5.2. Resultados multicriterio.

Los algoritmos genéticos multi-objetivos se utilizan para buscar soluciones óptimas en un espacio de búsqueda grande. En el contexto de este trabajo, se han empleado para alcanzar un conjunto de soluciones óptimas para permitir reducir el uso de memoria de las GPUs cuando se utiliza el algoritmo McGM y en el que no se desea perder precisión de una manera significativa en la estimación de movimiento. Como ya se ha mencionado previamente, las pruebas se han realizado haciendo uso de las secuencias de entrada "*Diverging Tree*" y "*Translating Tree*", ampliamente aceptadas en este área.

El primer experimento realizado ha sido evaluar la convergencia del algoritmo genético y el conjunto de soluciones óptimas alcanzadas. Para este propósito se ha utilizado la métrica de la distancia Euclídea entre soluciones consecutivas, como se describe en [216]. El algoritmo genético implementado incorpora una condición para parar basada en esta métrica cuando se han llevado a cabo un cierto número de iteraciones sin que se hayan producido cambios en la población, con el objetivo de asegurar que las soluciones no-dominadas convergen en el frente óptimo de Pareto.

La evolución del conjunto de soluciones no-dominadas a lo largo de las iteraciones, asociado con una estricta condición de parada, se muestra en la figura 61. Con el objetivo de facilitar la visualización sólo se muestra la reducción de memoria en la GPU y la diferencia en el error, aunque el algoritmo genético también optimiza el tiempo de estimación de movimiento. El error de Barron Ψ_E corresponde a la diferencia de la media del error de Barron con respecto a su homólogo en el McGM original.
En la figura 61 el número de puntos en el frente óptimo es un subconjunto final de las soluciones, debido a que las componentes referentes al tiempo de ejecución no se han tenido en cuenta. Además se puede observar cómo para la secuencia "*Translating Tree*" sólo se muestra un 15 % de parejas de Pareto, mientras que para la secuencia "*Diverging Tree*" se muestra un 40 %.
Otro factor a destacar es que el tamaño de la población se ha fijado en 500 con un 1 % de probabilidad de mutación. Los resultados obtenidos indican que después de un cierto número de generaciones, el algoritmo genético apenas mejora las soluciones no dominantes, aunque reporta nuevos pares.

El tamaño de la población sólo afecta al tiempo de ejecución final, alcanzando resultados de

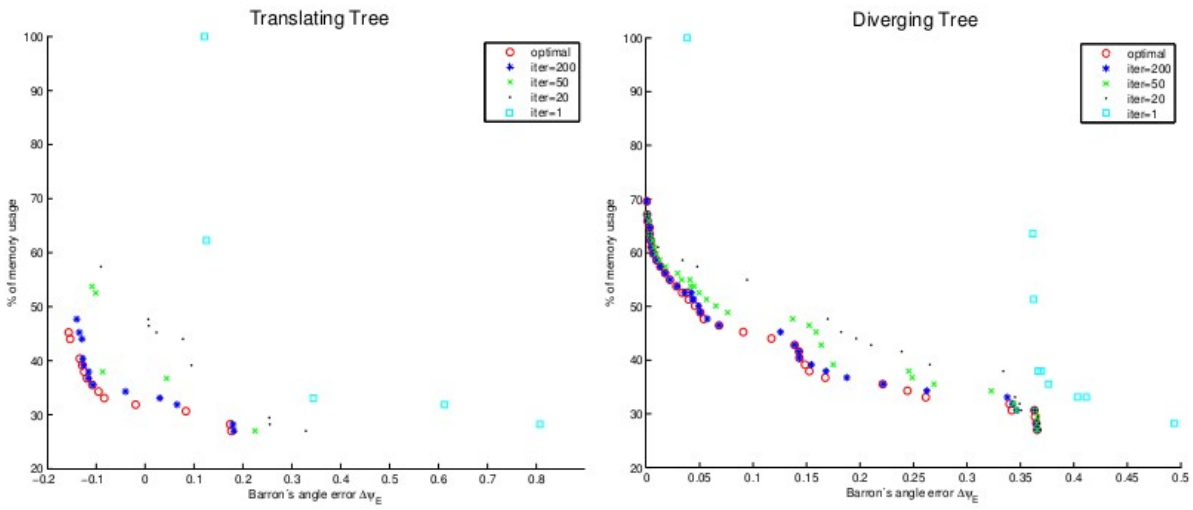


Figura 61: Evolución del conjunto de soluciones en el algoritmo genético para las secuencias *Diverging Tree* y *Translating Tree*.

una solución óptima con calidad similar. Empíricamente, el 1 % de mutaciones reporta mejor rendimiento en el algoritmo genético. Escoger valores más altos supone variaciones significativas entre generaciones consecutivas, lo que implica que se necesitan más generaciones para alcanzar el criterio de convergencia. En particular, mayores valores para el porcentaje de mutación implican un elevado número de iteraciones que varía entre el 15 % y el 320 %.

Como se puede observar en la figura 61, se generan soluciones óptimas con una reducción significativa en los requisitos de memoria, logrando soluciones más precisas que el algoritmo McGM original para la secuencia "*Translating Tree*".

6.5.3. Resultados en la multi-GPU.

Los tiempos medidos utilizando la mejor configuración para el algoritmo genético en el sistema *Tesla* M2070 se muestran en la tabla 18. Esta configuración hace referencia a una población con 500 individuos, un 1 % de mutación y una condición de convergencia estricta para encontrar el frente de Pareto. Se ha utilizado como prueba la secuencia "*Diverging Tree*" ya que los resultados obtenidos son similares a los observados para la secuencia "*Translating Tree*". Hay que señalar que esta elección sólo afecta al número de generaciones para alcanzar una solución óptima. Como se esperaba, la evaluación *fitness* es la etapa más costosa del algoritmo genético, no siendo muy relevante la sobrecarga en el intercambio de información entre el *host* y el *device*. De esta forma, se obtienen unos *speedups* de $\times 1.79$ para dos GPUs.

Tesla M2070	$t_{CPU}(s)$	$t_{GPU}(s)$	$t_{Comm}(s)$
1 GPU	1.24	22495.6	869.5
2 GPUs	124.2	12464.9	447.4

Tabla 18: Tiempos de ejecución con Multi-GPU en el sistema *Tesla* M2070.

Se han obtenido resultados similares con un número mayor de dispositivos gráficos. La tabla 19 muestra aún más aceleraciones cuando se habilitan dos GPUs. Además se puede observar cómo los ratios de escalabilidad se mantienen de manera satisfactoria con 4 GPUs alcanzando $\times$3.71 de *speedup*. Los resultados computacionales muestran que esta implementación multi-GPU es eficiente en términos de escalabilidad (95 % usando 2 GPUs y 93 % usando 4) y la tendencia indica que los tiempos de convergencia del algoritmo genético serían menores teniendo más recursos disponibles. Podemos concluir que la escalabilidad obtenida para los algoritmos genéticos es útil a la hora de resolver problemas de esta naturaleza. Los buenos resultados de rendimiento se deben tanto a una carga de trabajo equilibrada como al bajo coste involucrado en el intercambio de datos.

Tesla C1060	$t_{CPU}(s)$	$t_{GPU}(s)$	$t_{Comm}(s)$
1 GPU	1.18	23748.0	2025.8
2 GPUs	278.52	12613.1	1022.5
4 GPU	153.28	6248.4	513.5

Tabla 19: Tiempos de ejecución con Multi-GPU en el sistema *Tesla* C1060.

Por otra parte, el uso de múltiples niveles de paralelismo permiten multiplicar las aceleraciones. En primer lugar, los *speedups* alcanzados en el sistema multi-GPU pueden ser de hasta $\times$3.71 con 4 GPUs habilitadas. En segundo lugar, las aceleraciones de hasta $\times$32 se pueden obtener explotando el paralelismo de datos en una GPU. Por una lado, la combinación de ambas aceleraciones permite reducir el tiempo de exploración para alcanzar una solución óptima en un 99.2 % comparado con un procesador de propósito general. Por el otro, el uso de un sistema multi-GPU no sólo permite obtener mayores tasas de FLOPS que en una CPU, sino que en términos de consumo de energía (MFLOPS/vatio) también es mejor.

Aunque el tiempo de búsqueda en los algoritmos genéticos es importante, su uso favorece la obtención de soluciones casi óptimas que cumplen los requisitos de tiempo de respuesta o consumo de recursos, y conforme evoluciona el algoritmo genérico, la búsqueda se va refinando gradualmente. Esta característica, junto con la posibilidad de reducir el tamaño de la población, supone una disminución impresionante en el número de simulaciones que abre la posibilidad de construir un sistema inteligente que se autocorrija/adapte dependiendo de requerimientos específicos y/o cambios sustanciales en el entorno.

6.5.4. Resultados visuales.

Para terminar, se presentan los resultados visuales obtenidos con las dos secuencias de prueba. La figura 62 muestra las principales diferencias en los resultados obtenidos para la secuencia

"*Diverging Tree*". La salida original del algoritmo McGM se muestra en la parte de arriba de la figura, en el centro y abajo se muestran los resultados obtenidos a partir de las configuraciones dadas por el algoritmo genético para unas reducciones de memoria entre el 75 % y el 50 %. También se muestra el tiempo llevado a cabo para realizar la estimación de movimiento (ME_{time}). La fase (la dirección de los *pixels*) se representa con el código de color que se muestra en los bordes del fotograma y el módulo o velocidad se representa con una escala de grises.

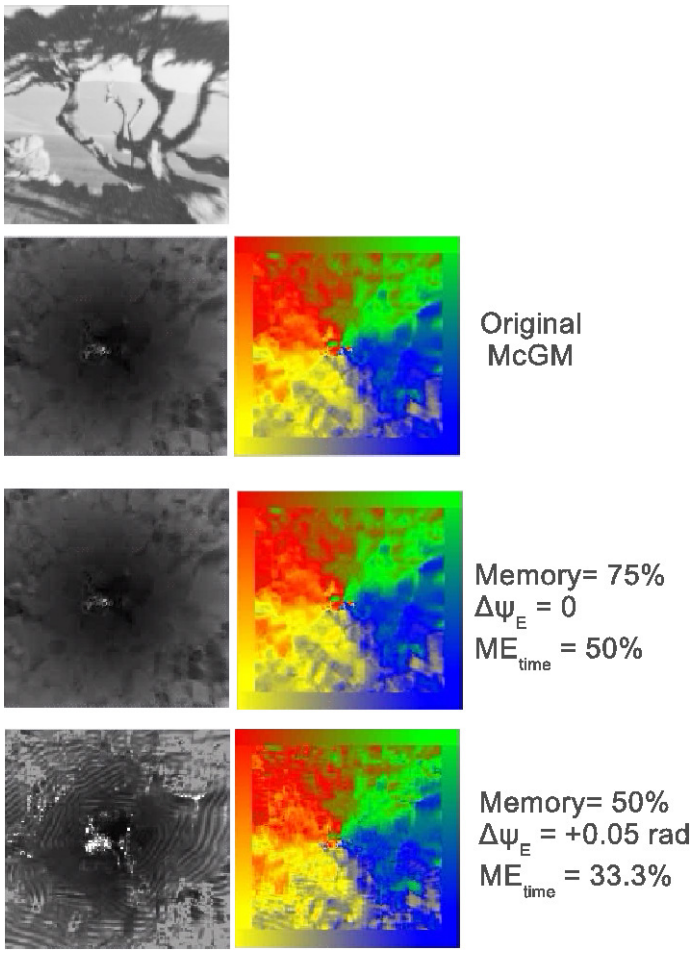


Figura 62: Resultados visuales para el algoritmo McGM original y los resultados obtenidos con el algoritmo genético con una reducción en la utilización de memoria GPU del 75 % (centro) y del 50 % (abajo) para el estímulo *Diverging Tree*.

De forma similar, la figura 63 muestra las soluciones obtenidas para la secuencia "*Translating Tree*".

Con el estímulo de entrada "*Diverging Tree*" se ha obtenido una reducción en el uso de memoria del 75 % con la misma precisión usando la métrica de Barron y un 50 % en el tiempo de ejecución comparándolo con el algoritmo original. Sin embargo, la configuración que

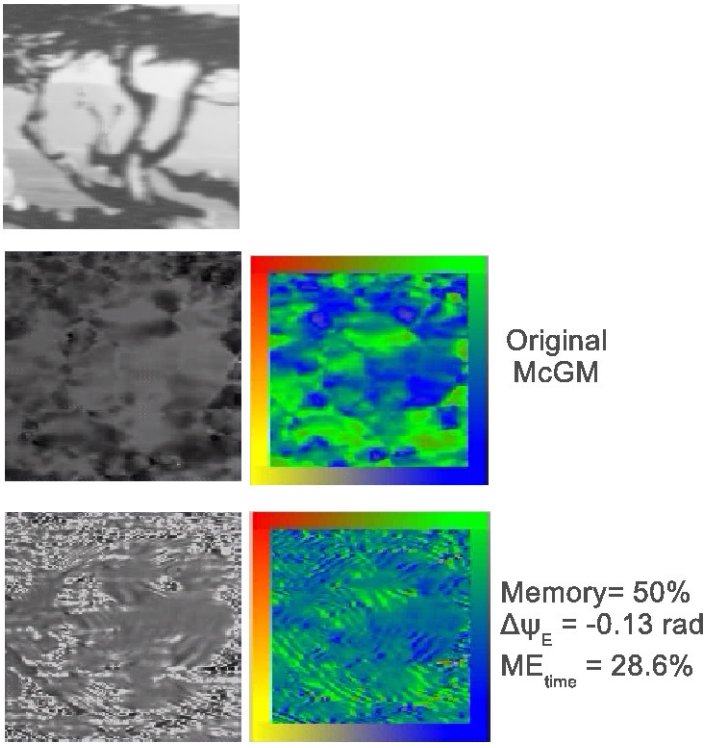


Figura 63: Módulo y fase para el estímulo *Translating Tree* obtenidos con el algoritmo McGM original y el obtenido haciendo uso del algoritmo genético con una reducción en el consumo de memoria de la GPU del 50 %.

consigue reducir la memoria en un 50 % degrada la precisión un 22 % con un $\times$3.3 de *speedup*.

Para la secuencia "*Translating Tree*", se ha encontrado una solución que requiere la mitad del uso de memoria, siendo más precisa (con un error de Barron de 0.13 radianes menos que el original) y $\times$3.5 veces más rápida.

6.6. Otras métricas de error.

A pesar de la popularidad de la métrica de Barron dentro de la comunidad científica en el contexto de estimación de movimiento, algunos autores [165, 166] señalan métricas específicas debido a la asimetría y sesgo de grandes vectores de flujo.

La tendencia del algoritmo NSGA-II teniendo en cuenta las métricas de McCane y Otte&Nagel se muestran en la figura 64. Al igual que en pruebas anteriores, las secuencias utilizadas para llevar a cabo el estudio han sido "*Translating Tree*" y "*Diverging Tree*".

Métrica Error	Secuencia Prueba	Memoria	ME_{time}	Precisión ($\Delta\Psi$)
Barron	*Translating Tree*	50 %	28.6 %	-0.13
		75 %	50.0 %	0.00
	Diverging Tree	50 %	33.3 %	0.05
$McCane_A$	*Translating Tree*	50 %	26.9 %	-0.12
		75 %	49.1 %	0.00
	Diverging Tree	50 %	28.9 %	0.05
$McCane_B$	*Translating Tree*	50 %	29.6 %	-0.11
		75 %	49.2 %	0.00
	Diverging Tree	50 %	34.5 %	0.09
Otte&Nagel	*Translating Tree*	50 %	25.3 %	-0.21
		75 %	49.1 %	0.00
	Diverging Tree	50 %	35.7 %	0.12

Tabla 20: Mejor configuración lograda para una reducción del 75 % y del 50 % de los requisitos de memoria para las métricas de McCane y Otte&Nagel.

La tabla 20 resume las principales configuraciones obtenidas tras la ejecución del algoritmo genético. Los resultados observados son consistentes con independencia de la métrica utilizada. Mientras que para la secuencia *Translating Tree* se mejora la efectividad de la estimación de movimiento con una reducción significativa del uso de memoria, para la prueba realizada con *Diverging Tree* no se observa degradación utilizando cualquiera de las métricas cuando el uso de memoria es del 75 %. Desde el punto de vista del tiempo de ejecución, los resultados de rendimiento son los esperados. Por un lado, la reducción de requisitos de memoria del 50 % se traduce en *speedups* entre $\times$3.3 y $\times$4. Por otro, El 75 % del consumo de memoria presenta una media del 50 % en el tiempo de ejecución de la estimación de movimiento.

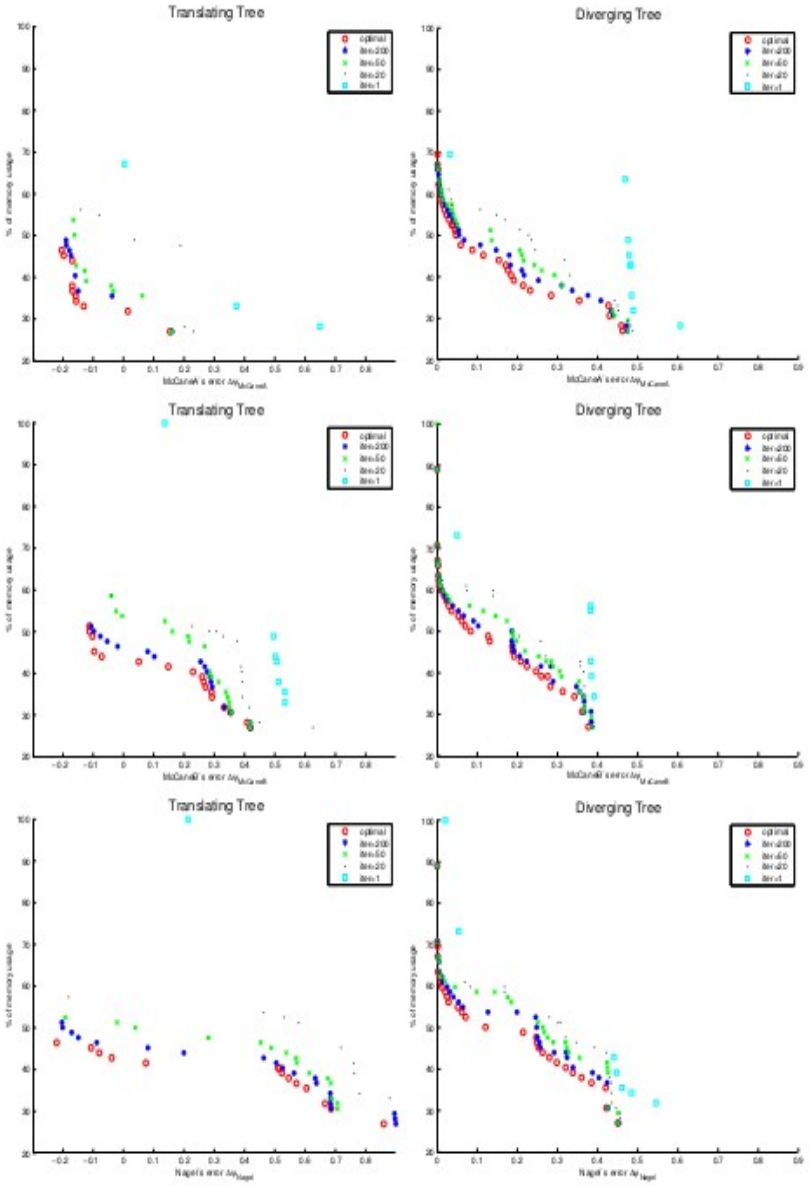


Figura 64: Evolución en el algoritmo genético para *Diverging Tree* y *Translating Tree* con las métricas de McCane y Otte&Nagel. El frente de Pareto muestra la configuración óptima dependiendo de la métrica aplicada.

6.7. Conclusiones.

En este capítulo se ha presentado una aproximación paralela para subsanar el problema del consumo de memoria en las GPU que ocurría en la implementación previa, como se ha descrito en 5.4. El problema descrito supone la principal motivación de utilizar algoritmos evolutivos para resolver problemas de optimización con varios criterios. El uso de algoritmos genéticos en sistemas multi-GPU permite explorar de forma rápida soluciones factibles con cualquier conjunto de datos de entrada. La elección del algoritmo NSGA-II viene motivada por los buenos resultados obtenidos con pocas iteraciones y que se encuentran cerca del frente óptimo de Pareto.

Desde el punto de vista de llegar a una solución que cumpla los requisitos de consumo de memoria, se ha observado:

- Para la secuencia de prueba *Diverging Tree* se ha conseguido una reducción del 75 % en la utilización de memoria con la misma precisión para todas las métricas consideradas y un 50 % en el tiempo de ejecución del algoritmo McGM comparado con el original. La configuración que reduce el uso de memoria en un 50 % presenta una degradación en la precisión entre el 15 % y el 25 % con unos *speedups* que varían entre $\times 2.8$ y $\times 3.5$.
- Para la secuencia *Translating Tree* una configuración permite reducir a la mitad los requerimientos de memoria siendo más precisa en términos de error y siendo entre $\times 3.3$ y $\times 4$ veces más rápida la estimación.

Desde el punto de vista de la eficiencia utilizando un sistema multi-GPU se ha observado:

- Un rendimiento satisfactorio con cuatro GPUs habilitadas que permite obtener *speedups* de $\times 3.71$.
- La implementación realizada es una aproximación escalable debido a que se consigue balancear la carga de trabajo y un bajo impacto en la comunicación entre el *host* y el *device*.
- Los *speedups* de $\times 3.71$ alcanzados con un sistema multi-GPU unidos a los $\times 32$ obtenidos al ejecutar el algoritmo McGM en GPU se han conseguido por medio de la explotación del paralelismo de datos en GPU.
- Es una buena alternativa a tener en cuenta en términos de consumo de energía (MFLOPS/vatio).

Debido a estos alentadores resultados, se puede concluir que es posible la construcción un sistema inteligente que se auto-corrija/adapte a las necesidades específicas o variaciones ambientales conforme evoluciona el algoritmo genético.

7. Conclusiones y trabajo futuro.

7.1. Resumen y conclusiones.

Como se ha puesto de manifiesto en esta tesis, la estimación de movimiento es un problema fundamental dentro del campo de la visión por computador y del tratamiento de vídeo digital, cuyo objetivo es el cálculo del campo de vectores que describen el movimiento aparente entre dos fotogramas de una secuencia.

Como es habitual en el procesado de señales, existe un compromiso entre el tiempo en el que se realiza este proceso y el tamaño de los datos en relación a la calidad del vídeo. Por ello, en la actualidad, muchas de las investigaciones dentro el campo de la codificación de vídeo y del cálculo del flujo óptico se centran en buscar algoritmos que puedan realizar, de manera más eficiente, la estimación de movimiento, un proceso que, en muchos casos, requiere una gran cantidad de cálculos complejos.

En el capítulo 2 se ha expuesto una clasificación de estos algoritmos en tres grandes familias: los modelos de energía, los modelos de gradiente y los modelos de emparejamiento, además de un estudio de los trabajos realizados en los últimos años al aplicarlos sobre aceleradores como las GPUs, los circuitos integrados o las FPGAs, obteniendo una visión global y actualizada de técnicas para calcular flujo óptico de manera eficiente en tiempo real.

Dentro del abanico de algoritmos de flujo óptico, se ha escogido el algoritmo McGM, puesto que cumple con los requisitos de: a) uso de filtros temporales y espaciales encontrados en el ser humano y b) uso de operaciones realizadas mediante dichos filtros e invarianza al contraste y al patrón estático sin perder información. Otra de las características del algoritmo McGM es que los cálculos que realiza son potencialmente paralelizables, por ello, los aceleradores *hardware* son una alternativa a tener en cuenta para realizar los cálculos de una manera más rápida.

Continuando con la línea de investigación propuesta por G. Botella en 2007 [172], se ha implementado el algoritmo McGM de flujo óptico bioinspirado en *hardware* gráfico, perteneciente a la familia de modelos de gradiente. Este algoritmo presenta una serie de ventajas con respecto a otros de su clase, como son la consistencia matemática frente a regiones donde no hay apenas contraste, la capacidad de separar las componentes diferenciales del flujo óptico sin recurrir a métodos iterativos, la versatilidad para justificar procesamiento de movimiento de segundo orden, robustez frente a ruido estático y dinámico, entre otras.

Para su implementación en *hardware* gráfico se ha escogido la arquitectura de cálculo paralelo CUDA, de Nvidia, de entre las alternativas propuestas en el capítulo 4, debido a su amplia aceptación en la comunidad científica. Esta arquitectura aprovecha la potencia de las GPUs para proporcionar un incremento del rendimiento del sistema, de forma que intenta explotar las ventajas de las GPUs frente a las CPUs de propósito general, utilizando el paralelismo que ofrecen sus múltiples núcleos.

El estudio inicial para comprobar la viabilidad de la implementación en *hardware* gráfico ha consistido en examinar la implementación de las primeras etapas del algoritmo intentando explotar la jerarquía de memoria de estos dispositivos y así obtener una primera idea de las ventajas de su aplicación. Los resultados obtenidos mostraban que el uso de GPUs como acelerador para aplicaciones de flujo óptico es interesante y abordable.

Las pruebas preliminares, que incluyen ejecuciones del algoritmo en GPU con variación en los parámetros de entrada, mostraron las aceleraciones obtenidas con respecto a sus ejecuciones homólogas en CPU, pudiéndose observar una mejora sustancial a la hora de obtener los datos. Tras analizar los resultados obtenidos en el capítulo 5, que implicaban comparar el rendimiento en GPU con el obtenido en CPU (con un solo *core* y con múltiples *cores*), se llega a la conclusión de que la utilización de GPUs con este fin es una alternativa a considerar, obteniendo unas aceleraciones de $\times 32$ y teniendo en cuenta siempre que la administración de memoria de la GPU y el intercambio de información entre esta y la CPU pueden reducir estos valores ligeramente. Por contrapartida, se ha observado que, debido a la naturaleza expansiva del algoritmo McGM, los requisitos de memoria aumentan a medida que se avanza a través de las etapas de este, lo que limita al algoritmo.

Las pruebas realizadas sobre la arquitectura CARMA, con el fin de estudiar el impacto de la implementación de esta solución en un sistema empotrado de alto rendimiento y energéticamente eficiente, ponen de manifiesto que los resultados obtenidos no son tan buenos como en un sistema de propósito general compuesto por CPU + GPU, aunque sí cumplen los requisitos para que pueda ser una alternativa a tener en cuenta a la hora de implantarlo en un sistema empotrado con requisitos de tiempo real.

Para intentar solucionar el problema de los requisitos de memoria anteriormente citado, y con el objetivo de construir un sistema capaz de autoadaptarse a ciertos cambios o necesidades, en el capítulo 6 se ha presentado una alternativa que busca equilibrar el consumo de los recursos con una adecuada precisión en el cálculo de los resultados, haciendo uso del paralelismo de manera eficiente. Para ello se ha utilizado el algoritmo genético NSGA-II, del que se han expuesto sus principales características, en un entorno con múltiples GPUs (multi-GPU), que permite explorar de manera más rápida soluciones factibles con cualquier conjunto de datos de entrada. Los *speedups* obtenidos con multi-GPU (de $\times 3.71$), unidos a los obtenidos al ejecutar el algoritmo McGM en GPU y la posibilidad de reducir los requerimientos de memoria, permiten a la solución expuesta ser una buena alternativa a tener en cuenta en términos de consumo de energía.

Así pues, las principales contribuciones de este trabajo son:

- Una visión global del estado del arte de la implementación de algoritmos de estimación de movimiento y flujo óptico en aceleradores *hardware*, en la que se puede comprobar como es un tema muy abordado en la actualidad y con múltiples aplicaciones en el mundo real.

- Se ha presentado una implementación en *hardware* gráfico del algoritmo McGM que pone de manifiesto que se pueden obtener resultados óptimos para su utilización en sistemas con requisitos de tiempo real.

- Puesto que una de las limitaciones del algoritmo McGM es su característica expansiva en lo que a consumo de memoria se refiere, se ha presentado como posible solución un sistema que utiliza el algoritmo genético multiobjetivo NSGA-II para minimizar dicho consumo de memoria sin perjudicar la precisión ni el tiempo de ejecución del algoritmo para seguir manteniendo las ventajas obtenidas con el uso de GPU sobre este tipo de problemas.

Por todo esto, podemos concluir que el empleo de GPUs con el fin de calcular el flujo óptico con el algoritmo McGM en secuencias de vídeo aparece como una solución válida en términos de rendimiento y consumo de recursos *hardware*.

7.2. Proyección de futuro.

El trabajo realizado permite pensar en posibles mejoras y aplicaciones que hagan uso de las conclusiones aquí obtenidas. Entre las líneas de investigación que podrían continuarse a raíz de este estudio se deberían tener en cuenta las siguientes ideas:

- Aplicar de manera óptima la implantación del modelo completo de procesado de movimiento de gradiente, multicanal y multiescala, considerando su representación en colores, ya que no existe actualmente ninguna implementación del mismo que haga uso de *hardware* gráfico, según nuestro conocimiento. Aunque en el trabajo de Xuefeng Liang *et al.* [175] se aborda el campo del color, no se aplica en el área del *hardware* gráfico, pudiendo comparar los resultados obtenidos y la viabilidad de su implantación.

- Reutilizar el sistema desarrollado en el capítulo 6 con un pronosticador del entorno, para permitir auto-reconfiguración en tiempo real dependiendo de los recursos disponibles en la plataforma y de determinadas limitaciones externas, esperando contribuir en las nuevas tendencias de sistemas de visión, útiles en muchas aplicaciones del mundo real.

- Realizar una comparación completa entre la implementación del algoritmo McGM en GPU y la realizada en FPGA, anteriormente implementada por G.Botella en [172] y estudiar las ventajas e inconvenientes que cada una de las soluciones aporta.

- Estudiar la viabilidad de una implementación del algoritmo haciendo uso de directivas OpenACC y comparar los resultados obtenidos con los mostrados en el trabajo. De esta forma se podría observar si este tipo de directivas pueden aplicarse en este área, puesto que una de sus ventajas es que permiten a los programadores de GPGPU el desarrollo de una manera más sencilla.

- Implementar el algoritmo McGM en estéreo para poder captar y medir, entre otras características, la profundidad de los distintos objetos que aparecen en las secuencias de vídeo. Para llevar a cabo este estudio sería necesario ejecutar en paralelo el algoritmo McGM en dos dispositivos distintos, con el fin de emular la visión binocular del

ser humano. Teniendo dos imágenes tomadas desde posiciones ligeramente diferentes, obtenidas por separado por cada uno de estos dispositivos, el sistema sería capaz de reconstruir la distancia (y por lo tanto la profundidad) analizando la disparidad o el paralelismo entre las imágenes observadas.

- Extender el algoritmo McGM al dominio multi-espectral (varios canales de información) con importantes aplicaciones en campos tan diversos como el de imágenes por satélite, aviónica, seguridad, médico. Adicionalmente, se podrá abordar el diseño de aceleradores eficientes a esta extensión teniendo en cuenta no solamente el rendimiento ofrecido sino cuestiones relativas al consumo energético o precio final. Según el estado del arte actual, no existen modelos ni aceleradores que aborden estas cuestiones de forma favorable.

Se espera que este sistema contribuya a las nuevas tendencias de visión artificial, muy útiles para muchas aplicaciones del mundo real, ya que al estar implementado en GPU, las cuales se encuentran disponibles en una amplia gama de dispositivos electrónicos, podría aplicarse en múltiples campos y aplicaciones.

Un posible uso podría ser la adaptación a teléfonos móviles de última generación mediante una aplicación de *tracking*, que permitiera capturar vídeo y procesarlo sin sobresaltos o reduciendo la cantidad de ellos. Otra posible aplicación sería integrarlo en robótica con el objetivo de simular el comportamiento del procesamiento visual humano en robots. En el campo de defensa permitiría tener otros sistemas inteligentes alternativos de detección, por ejemplo, para descubrir misiles enemigos en tiempo real.

En el campo de la vídeo-vigilancia, este sistema podría aplicarse en situaciones que requieran un cierto grado de seguridad, para detectar intrusos en determinadas zonas controladas, o para conocer la velocidad de los objetos que aparecen en escena, permitiendo detectar excesos de velocidad en la misma, pudiendo en ambos casos comunicarlo a los organismos de seguridad correspondientes.

Un sistema de este tipo podría incluirse en simuladores civiles o de defensa sin un gran coste económico asociado para permitir que estos detectaran situaciones aleatorias, permitiendo al simulador tomar decisiones de trayectoria en tiempo real.

Como se puede observar, existen múltiples campos diferentes en los que se podrían aplicar los conocimientos presentados en este trabajo.

7.3. Extensión del algoritmo como herramienta de diagnosis médica.

Las imágenes médicas se han convertido en una herramienta de diagnóstico absolutamente esencial para las prácticas clínicas. En la actualidad, hay múltiples patologías que pueden ser detectadas con una precocidad nunca antes conocida empleando computadores. Su uso cada vez mayor proviene de la generación de imágenes en ordenador antes de la cirugía. El análisis del movimiento, en particular, juega un papel importante en el análisis de las actividades o

comportamientos de los objetos activos en la medicina. En esta subsección se presenta una posible extensión de esta tesis en el contexto de imágenes médicas.

Un modelo basado en un algoritmo de estimación de movimiento de la familia del gradiente ha sido publicado [51], una revista de computación en el ámbito médico cuyo trabajo es trasladable a un entorno como el desarrollado en esta tesis doctoral, donde se explota el paralelismo en un acelerador gráfico.

Este sistema se basa en una segmentación dinámica basada en histograma (método Otsu) con el método de estimación de movimiento Lucas&Kanade [71]. El método de Lucas y Kanade es un algoritmo bien conocido, al tiempo que añade algunas variaciones para mejorar la viabilidad de la implementación de hardware. Se presenta un esquema simplificado del algoritmo, de la siguiente manera. El modelo de Lucas y Kanade calcula flujo óptico utilizando una técnica de gradiente que hace uso de filtros derivados espacio-temporales. El modelo viene de la conservación de la intensidad básica sobre el tiempo.

Este sistema está basado en una implementación de un sistema híbrido de bajo coste, especialmente diseñado en escenarios médicos en los que se procesan las imágenes médicas para ayudar en los diagnósticos médicos y ayuda en la toma de decisiones. Ha sido utilizado para imágenes MRI de mama (*Magnetic Resonace Imaging*) sobre la base de la estimación de movimiento densa, que pueden ayudar a los especialistas en la prestación de atención rápida al diagnóstico de tumores relacionados con la mama.

Apéndice

A. Accelerating bioinspired algorithms for motion estimation in parallel hardware: A Summary in English.

This chapter is a summary of the PhD. thesis titled "*Accelerating bioinspired algorithms for motion estimation in parallel hardware*" . The summary is structured as follows. In Section A.1 we introduce this thesis, presenting the main motivation. In Section A.2 we summarize the objectives that had been planned at first and the others that have came up during development. Section A.3 is a description of the results we obtained applying GPU to bioinspired algorithms, more specifically, how good is the implementation in GPU of the McGM algorithm. To conclude, in Section A.4 we present the main conclusions of this work.

A.1. Introduction.

A considerable amount of useful information can be processed from time-varying images for several applications—such as robot navigation, biomedical assistance, surveillance, tracking, sport monitoring, video compression, *etcetera*. The motion estimation process determines motion vectors, describing the transformation from a two-dimensional image to another, normally from contiguous frames in a video sequence. The motion relies on three dimensions, but the images are a projection of the three-dimensional scene onto a two-dimensional plane, therefore posing a mathematically ill-posed problem. The motion vectors can relate to the entire image (global motion estimation) or specific parts, such as shaped patches, rectangular blocks, or even per pixel. To overcome these handicaps, external knowledge is necessary regarding the behavior of objects, such as rigid body constraints or several other models that might approximate the motion of a real video camera. These models consist of the motion of rotation, translation, and zoom, in all three dimensions.

Optical flow is a term that has been widely linked with motion estimation, despite their not being the same concept. The brightness pattern in the image moves as the object that appears moves. Optical flow is the apparent motion of the brightness pattern of entities—surfaces, edges, and objects in a visual scene—caused by the relative motion between an observer (an eye or a camera) and the scene [8]. Ideally, optical flow would correspond with the estimated motion field, despite the fact that apparent motion can be caused by lighting changes without any actual motion. One example of this would be a rotating sphere under constant illumination, which delivers a non-null value of motion estimation but a zero value of optical flow. Similarly, a static sphere with changing light will deliver optical flow; meanwhile, the motion field is null [217]. Optical flow techniques such as motion detection, object segmentation, time-to-collision, focus of expansion calculations, motion compensated encoding, and stereo disparity measurement utilize this motion of the objects' surfaces and edges. Therefore, if we want to recover the bi-dimensional optical flow (two components), we need to add an additional constraint to the environment, because only one independent component can be recovered; therefore, the problem is mathematically ill-posed [8, 218], which is called '*the*

aperture problem'. The optical flow error measures have to be applied to synthetic sequences, where every pixel's motion is known in advance, due to the previously mentioned facts.

There is a general agreement about the organization of algorithms and techniques used to estimate optical flow. They generally fall into one of three categories:

- *Differential or gradient techniques* work using derivatives of image intensity in space and time. Combinations and ratios of these derivatives yield explicit measures of velocity [71, 6]. The gradient-based family can estimate vector motion of every single pixel, giving a dense representation of the processed frame. There are several examples of video compression using gradient based algorithm [219]. Recursive algorithms belonging to this family do not have to transmit motion information. Nevertheless, this algorithm family has the drawback of large motion vectors (severe motion), noisy images, and changes in illumination.

- *Motion energy methods* use space-time oriented filters tuned to respond optimally to specific image velocities. Banks of such filters are used to respond to a range of visual motion possibilities [5]. Energy models are probabilistic, delivering a population of solutions that do not indicate motion itself and are not usually used for multimedia purposes.

- *Pattern matching methods* operate by comparing the positions of image structure between adjacent frames and inferring velocity from the change in location. These are probably the most intuitive methods [4]. This methods are well known as *Block Matching* algorithms and have the pros of robustness, low cost VLSI implementation (because of their regular parallel procedure), and low overhead (since they contain one vector per block). Nevertheless, there are many cons, since a block may contain several moving objects and fail for zoom, rotational motion, local deformation, and blocking artifact. Additionally, they usually estimate the motion error by minimizing a metric, which does not release the true movement, etc.

The particular implementation of the algorithm used in this paper belongs to this last family and is based on Johnston's work [169, 173, 175]. The multi-channel gradient model (McGM) was developed as part of a research effort aimed at improving our understanding of the human visual system and it is a neuromorphic algorithm fitted to allow the construction of viable, highly robust, front-end processors for image recognition systems [94]. The model also allows us to make predictions that can be tested through psychophysical experimentation as separate motion illusions that are observed by humans in experiments[74]. This contribution outlines an efficient implementation for optical flow gradient-based models using Graphic Hardware platforms by exploiting different parallelism levels (pixel and block levels) in each stage of the algorithm and also by efficiently using the memory hierarchy (using combinations of global and shared graphic processing unit (GPU) memory to process the data). The result of this optimization at every stage is a notable throughput enhancement (for example, going from 23$\times$ to 82$\times$, depending on the stage computed and delivering a global throughput of 32$\times$ when using a 256$\times$256 input frame resolution). This platform is particularized, among

other low-level vision approaches, for a bio-inspired motion estimation scheme.

Furthermore, motion estimation and compensation are crucial for multimedia coding characterized by high memory requirements and computation complexity. When considering MPEG processing, motion estimation is acknowledged as the most time-consuming [199], creating up to 90 % of the total execution time [200, 201]. Additionally, motion estimation has several applications regarding multimedia scope as segmentation, extraction of 3D structure, pattern tracking, filtering, compression, and deblurring.

A.1.1. Evolution and bio-inspired algorithms for vision processing.

Nature has engineered many solutions for visual processing, each of which is adapted to the needs of the specific organism involved. For example, invertebrates accomplish interception, navigation, obstacle avoidance, and recognition in a way that performs robustly, frequently using less than a million neurons [220]. These organisms evolved with ad hoc visual algorithms and neuronal architectures highly efficient in terms of energy wasted, robustness, speed, and space. Properties such as these have driven researchers to try and emulate the robustness and efficiency of natural solutions for both software and hardware [221, 222, 223]. Regarding these systems, the neuromorphic approximations [224] are based on how the nervous systems create physical architectures and computations, attending to the information coding, morphology, robustness against damage, *et cetera*. For example, autonomous robots can navigate very successfully using neuromorphic systems based on models of insect vision [225, 226].

Many designs and solutions come from myriad research fields such as biological modeling, artificial intelligence, signal processing, and robotics. Perhaps the best algorithm for optical flow would require a set of characteristics such as those found in mammals, where huge computational resources are required. As technology advances alongside GPU systems, more evolved and versatile biological systems can be developed, which would allow solutions to more problematic and demanding visual tasks. For example, the implementation of a neuromorphic optical flow algorithm with highly demanding computational resources using GPU systems, enabling real-time performance because of the massively parallel architecture and the capability of processing floating-point systems.

A.1.2. Graphic hardware paradigm.

Graphic processing units are available in low cost devices, thanks, in great part, to the entertainment industry. These devices are based on multi-core systems with a complex memory hierarchy. These platforms are designed to take advantage of the inherent parallelism of data rendering 3D scenes. However, currently, these platforms are used as parallel coprocessors, executing a high number of threads simultaneously.

The increased computing capabilities of Graphics Processing Units (GPUs) in recent years has increased their use as accelerators in many areas such as scientific simulations, computer vision, bioinformatics, cryptography, and finance, among others. This increase is largely due

to impressive performance rates. For example, one of the latest GPUs from Nvidia (NVIDIA Corp., Sta. Clara, CA, USA), the K20X, achieves 3.95 TeraFLOPS in single precision with 1006 cores and also incorporates the newer *Kepler* architecture, whereas an Intel i7-3930 microprocessor can only complete 154 GigaFLOPs. Current trends seem to indicate that this capacity will grow even more with the incorporation of 22nm and 28nm technologies. Recently, for example, AMD announced its Radeon 8000 Series, branded as Sea Island, and Intel is manufacturing Knights Corner products. However, key points that dramatically affect performance rates include the efficient use of the memory hierarchy and the exploitation of parallelism capabilities.

The increased demand for information to be processed also plays a role, because the use of these devices as accelerators is limited due to DDR memory restrictions. To solve this problem, research [202] has often proposed a data reuse alternative with the aim of minimizing the memory traffic between GPU and CPU. Another approach in the field of rendering meshes can be found in [203] a solution that uses more efficient algorithms in terms of memory consumption alongside other techniques based on simplification or information compression. The GPU memory reduction proposed here is addressed using a motion estimation scenario, which, to the best of our knowledge, doesn't exist as a solution in any of the current literature.

A.1.3. The Compute Unified Device Architecture.

The Compute Unified Device Architecture (CUDA) programming paradigm represents the GPU as a coprocessor, which can execute parallel kernels and can also offer extensions for the C language. This paradigm maps the data from the GPU, transfers the data between the GPU and CPU, and runs the kernels. A CUDA kernel runs several code lines over many parallel threads. This kind of system uses the concept of Single Instruction Multiple Threads, so a single instruction is executed from many threads with different input data. The tasks are organized through CUDA blocks, which make it possible to run up 1024 cooperating threads, thanks to a low-latency local memory and the use of synchronization tokens. Some CUDA blocks can only be coordinated through the global memory with high latency. Processors are grouped into multiprocessors, which are assigned to a particular task [179]. Each processor has its own functional unit and large register capabilities, as well as constant and local memories, which allow the execution of thousands of concurrent threads. Each multiprocessor shares a low-latency shared memory. The memory hierarchy also includes read-only cache memory to accelerate access to the textures and constants. Each CUDA block is executed by one multiprocessor, which can map several blocks to the present multiprocessor, depending on the availability of resources. The instruction unit controls threads execution. The subtask creation and the programming are performed entirely in hardware. The programming unit is not an individual thread, but a group of threads called a warp. During each cycle, the scheduler chooses the next warp to execute in the same way a fine-grain multi-threading processor would be scheduled. One of the strongest points affecting the final throughput is the efficient use of the memory hierarchy. Because this hardware allows the hiding of the high latency memory access, as does the multi-threading fine grain processor, the simultaneous access from the threads to the DRAM presents a challenge. Due to this drawback, the

efficient use of the local shared memory and the texture memory is key in many algorithms to enhance the throughput. Additionally, the access from the threads of a warp must be aligned, because only one memory access is translated, slightly reducing the contention with memory [179].

A.2. Objetives.

The main objetive of this thesis is the viability study of an implementation of the McGM algorithm in graphic hardware. This section moves through a specific neuromorphic model, which is analyzed with its GPU implementation.

This method has a disadvantage relating with memory compsuption, as will be seen below. So, in this chapter we will present the motivation of this study where multi-objective optimization is used to try to solve this problem.

A.2.1. MultiChannel Gradient Model (MCGM)

This original algorithm was proposed by Johnston et al. We have applied Johnston's description of the McGM model [169, 175], while adding many specific variations to improve the viability of the GPU implementation. McGM algorithm could be divided in some different stages. Figure 65 shows a simplified scheme of the processing pipe to be completed.

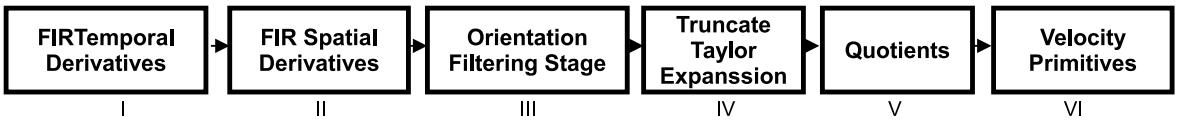


Figura 65: Scheme of the multichannel gradient model with several stages.

A.2.1.1. Stage I. Temporal filtering

We took, as starting point, the work performed by Hess and Snowden [170] regarding temporal processing in human beings. We modeled three different temporal channels: one low-pass filter and two band-pass filters with a central frequency of 10 and 18 Hz, respectively. Where α represents the peak of the log-time Gaussian and τ represents its spread. These channels were accomplished using a Gaussian differentiation in the log-time domain, as shown in Figure 66.

$$nucleo = \frac{e^{-(log(t/\alpha)/\tau)^2}}{\sqrt{\pi}\alpha e^{(\frac{\tau^2}{4})}} \tag{27}$$

A.2.1.2. Stage II. Spatial filtering.

Attending to the space domain, the shape of the receptive fields from the primitive visual cortex were modeled using either Gabor function, where their impulse responses were defined by harmonic functions multiplied by a Gaussian; or, alternatively, they could be modeled with a derivative set of Gaussians [227]. The Gaussian is a unique function in many ways and is

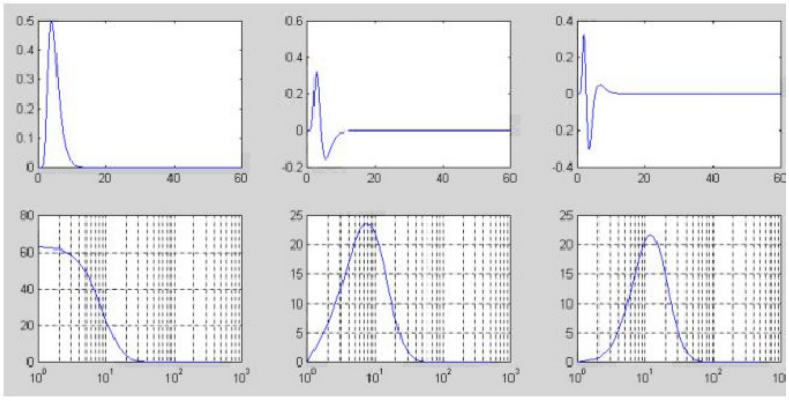


Figura 66: Three visual channels found in the human being, representing the impulse and frequency response, respectively.

of particular importance to biology.

As far as the differentiation orders rising, the Gaussians were fitted and tuned to higher spatial frequencies. Finally, a range of independent channels was constructed, as shown in Figure 67.

The *nth* Gaussian derivative can be expressed as a Hermite polynomial multiplied by the original Gaussian: (σ is the standard deviation of the Gaussian, and the scale factor ensures the function integrates to unity).

$$\frac{d^n}{dx^n}(G_0) = \frac{d^n}{dx^n}\left(\frac{e^{-\frac{x^2+y^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}\right) = H_n\left(\frac{x}{\sqrt{2\sigma}}\right) H_n\left(\frac{y}{\sqrt{2\sigma}}\right)\left(\frac{-1}{\sqrt{2\sigma}}\right)^{2n}\left(\frac{e^{\frac{-x^2+y^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}\right) \quad (28)$$

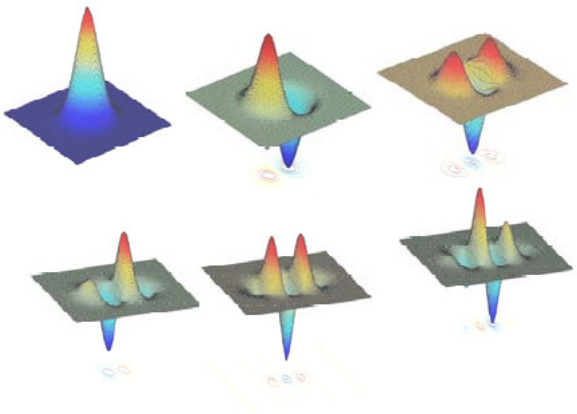


Figura 67: Bi-dimensional Gaussian and its different derivatives, from zero-order to fifth-order.

A.2.1.3. Stage III. Steering filtering.

The steering stage represents our approach to projecting the space-temporal filters, calculated in previous stages, under the different orientations. Calling *n* and *m* the order in *x* and *y* directions, respectively, θ the angle projected, and D the derivative operator, the general expression was derived as a linear combination of a filter belonging to the same order basis.

$$G^{\theta}_{n,m} = \left[\sum_{k=0}^{n} \binom{n}{k} (D_x cos\theta)^k (D_y sin\theta)^{n-k}\right] \cdot \left[\sum_{i=0}^{m} \binom{m}{i} (-D_x sin\theta)^i (D_y cos\theta)^{m-i}\right] G_0 \tag{29}$$

Also, Figure 68 shows an example of steering filtering, as a linear combination of three filters regarding second order. It is possible to appreciate two zero-crossings of the lobes from the functions represented below.

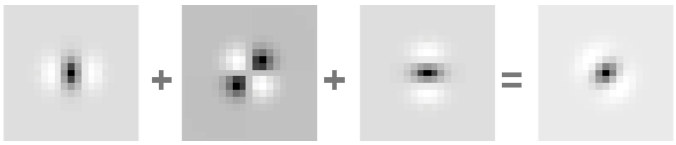


Figura 68: Steering schema for second order filters (45^o).

A.2.1.4. Stage IV. Taylor truncation.

At this stage, a truncated Taylor expansion was performed, using each oriented filter previously calculated. This function represents a robust structure that gathers all space-temporal information sequences, approximating one generic pixel by the set of derivatives from the neighborhood. It can be written as follows:

$$I(x+p, y+q, t+r) = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} \frac{p^i q^j r^k}{i!j!k!} \frac{\partial^n}{\partial x^i \partial y^j \partial t^k} \tag{30}$$

The three Taylor expansion derivatives were constructed in one large image using the completed set of basis filter responses. According to the original model [169], the expansions were truncated after the third order in the primary direction and the second order in the orthogonal and temporal directions.

A.2.1.5. Stage V. Quotients.

This was the last stage derived from the common pathway calculation. The next stage is where the modulus and phase estimation was implemented using separate expressions. The goal here was to compute a quotient for every sextet's component:

$$\left.\begin{matrix} X = \partial I/\partial x \\ Y = \partial I/\partial y \\ T = \partial I/\partial t \end{matrix}\right| \rightarrow \left.\begin{matrix} XX & XY & XT \\ YY & YT & TT \end{matrix}\right| \rightarrow \begin{matrix} YT/TT & XY/XX & XT/XX \\ YT/YY & XY/YY & XT/TT \end{matrix} \tag{31}$$

A.2.1.6. Stage VI. Velocity primitives

The previous stages computed the visual information using a Taylor representation for each pixel and calculated the speed for a range of orientations to simulate the orientation columns found in the striate cortex [169]. This was achieved by rotating the coordinate system and Gaussian derivative filters (Steering Stage) to a number of primary directions. Next, the speed measurements were taken, both parallel and orthogonal, of the primary directions to yield a vector of speed measurements, whose components are speed and orthogonal speed:

$$\hat{s} = \left(\hat{s}_{\parallel}, \hat{s}_{\perp}\right) \tag{32}$$

The raw measurements of speed were also conditioned by including the measurements of the image structure $X\Delta Y/X\Delta X$ y $X\Delta Y/Y\Delta Y$ where the final conditioned speed vectors

$$\hat{s}_{\parallel} = \sqrt{\frac{2}{\Sigma}}\left[\frac{XT}{XX}\left(1+\left(\frac{XY}{XX}\right)^2\right)^{-1}\right] \qquad \hat{s}_{\perp} = \sqrt{\frac{2}{\Sigma}}\left[\frac{YT}{YY}\left(1+\left(\frac{XY}{YY}\right)^2\right)^{-1}\right] \tag{33}$$

resulted in the number of orientations $\sum$ at which speed was evaluated. Inverse speed was also calculated:

$$\breve{s}_{\parallel} = \sqrt{\frac{2}{\Sigma}}\left[\frac{XT}{TT}\right] \qquad \breve{s}_{\perp} = \sqrt{\frac{2}{\Sigma}}\left[\frac{YT}{TT}\right] \tag{34}$$

The inverse speed was evaluated using different terms from those used to compute speed and necessitated an additional independent measurement. Finally, the motion modulus was calculated through a quotient of determinants:

$$Modulus^2 = \begin{bmatrix} \hat{s}_{\parallel}\cos\theta & \hat{s}_{\parallel}\sin\theta \\ \hat{s}_{\perp}\cos\theta & \hat{s}_{\perp}\sin\theta \\ \hline \hat{s}_{\parallel}\breve{s}_{\parallel} & \hat{s}_{\parallel}\breve{s}_{\perp} \\ \hat{s}_{\perp}\breve{s}_{\parallel} & \hat{s}_{\perp}\breve{s}_{\perp} \end{bmatrix} \tag{35}$$

The direction of motion was extracted by calculating a measurement for phase that was combined across all speed-related measures:

$$phase = \arctan\left(\frac{\left(\breve{s}_{\parallel}+\hat{s}_{\parallel}\right)\sin\theta + \left(\breve{s}_{\perp}+\hat{s}_{\perp}\right)\cos\theta}{\left(\breve{s}_{\parallel}+\hat{s}_{\parallel}\right)\cos\theta - \left(\breve{s}_{\perp}+\hat{s}_{\perp}\right)\sin\theta}\right) \tag{36}$$

A.2.2. GPU Implementation.

This implementation was based on Tesla technology from NVIDIA. The system contained several GPUs that could be programmed using the CUDA paradigm. CUDA is a toolset from NVIDIA [179], which includes a specific compiler for graphic hardware. In this subsection, the mapping scheme was explained using the algorithm of a single GPU implementation. A real-time feeding by an external device (e.g., a camera) was assumed. To facilitate an algorithm-programming task, each stage corresponded with a GPU execution *kernel*.

A.2.2.1. Stage I.

The temporal filters were quantized from the $k(t)$ expression 27. After this process, an equalization development was performed, ensuring that the summing of all values became null.

Different GPU schemes were examined with the aim of retaining valid efficiency rates. The first version stored the filter coefficients using different memory hierarchy levels: global, texture, or constant. The second version conserved the information from the *N* frames in global memory or shared memory. The third version examined the throughput reached when reusing information from shared memory.

The *N* frames involved at this stage were sliced into 16×16 size blocks, which were mapped into a multiprocessor. The processing of these frames was performed, therefore, in 16×16 threads. Thus, a single thread calculated all operations for an individual output pixel. The output of this stage delivered a temporal blurring (low-pass filtering) and two first-temporal derivatives (band pass filtering), as shown in Figure 69.

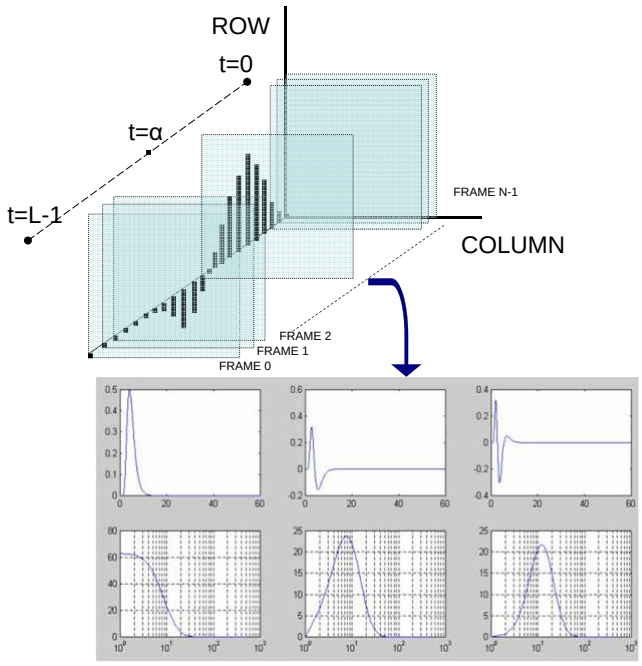


Figura 69: Temporal filtering performed by a linear convolution. The parameters used are as follows: the first frame ($t = 0$), the last frame ($t = N - 1$) and the filter length (L). A response for the α frame was delivered by ($t = L - 1$).

A.2.2.2. Stage II.

During Stage II, the spatial convolutions shown in the model were performed. A low-pass filter was applied to smooth out some of the high-pass components of the captured information from the external periphery. Starting with the three different outputs previously performed, a pyramidal set of many filters was built, as shown in Figure 70.

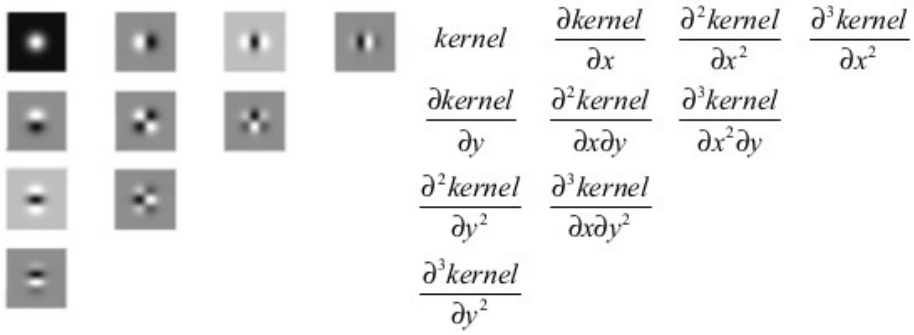


Figura 70: Pyramidal structure of spatial filtering from zero-order to third-order.

The structure was constituted of different Gaussian derivatives, where each diagonal was composed using the same differentiation order as a spatial basic function (denoted as kernel in Figure 70). The GPU implementation was based on the creation of two different kernels to reduce the computational order. The usual convolution of using a mask with an image has a complexity of $O(n^2)$. However, if a separable convolution is taken into account (because the Gaussian is a separable function), it was possible to reduce the order to $O(2n)$.

The main kernel structure implementation was established from the SDK convolution code from NVIDIA [195], although it differed as not centered convolution was applied. A higher *blockid.x* or *blockid.y* size was chosen, depending on filter direction (CUDA nomenclature refers to the numbers of blocks in each direction), in order to exploit as much parallel grain as possible. The corresponding filter coefficients were gathered in the texture memory. The computation was performed in the way that output information is kept consecutively, according to the differentiation order processed.

A.2.2.3. Stage III.

Usually, steering filters are avoided in computer vision systems because real-time is so demanding because of the high cost of convolutions [196]. Stage III required a filter with orientation θ from the previously constructed basic filter bank. Using the linear property of the convolution as a main advantage, it was possible to synthesize the oriented filter G^θ , from its basic set of filters and, therefore, synthesize the oriented response R^θ from R_1 , ..., R_n:

$$R^\theta = F^\theta \otimes I = (K_1F_1 + K_2F_2 + ... + K_nF_n) \otimes I = K_1R_1 + K_2R_2 + ... + K_nR_n \quad (37)$$

where the weight coefficients were represented with K_1 , ..., K_n , I as the input image to be steered, and θ as the orientation degree to be projected. The GPU calculation was similar to Stage I, where a single kernel was applied that computed all multiplication and addition operations. In this case, the steering coefficients were located in the shared memory, which yielded better performance rates.

A.2.2.4. Stage IV.

Two tasks were performed in Stage IV. The first task was to calculate and store all weights that would be used in the truncated Taylor expansion. The second task was to build the Taylor expansion itself.

The main drawback of this implementation stage lies in the fact that computation involved several chained summations. This type of process was not particularly successful on a GPU system, because it relied on a reduction operation.

However, there were different strategies [195] that achieved satisfactory speedups on a GPU, based on several synchronization schemes. In our case, a coarser grain parallelism was exploited, which allowed the avoidance of reduction operations. Pixel-level partitioning was also applied, which hid inefficient synchronization barriers due to reduction operations.

To summarize, the *kernel* implemented performed all computations at pixel level in a parallel way; therefore, one single *thread* was managed for each pixel. Note that there were no data dependencies, so the computation was done with as small granularity as possible.

A.2.2.5. Stage V + VI.

Because of the results obtained in the Stage VI of the McGM algorithm (explained in A.2.1.6), the velocity modulus (the speed of each pixel) and the angle (the movement direction of each pixel) was calculated for each single pixel of each frame. Therefore, this stage retained pixel-grain-level exploitation.

As it will be shown in the results section A.3, this implementation has a disadvantage refering to de memory consumption. The algorithm features create a bottleneck, specifically when memory requirements are increased in each stage, with an upward trend. This disadvantage limits GPU viability. Attending to the largest memory usage configuration considered in the implementation, 3.5 GB of global memory was used, which was close to the capacity limit of a single GPU. Although the memory capacity is greater for GPUs nowadays, this problem is still present with larger data input resolutions.

Trying to solve the problem mentioned before, we propose a solution using multiobjetive optimization and genetic algorithms that alows us to tune the McGM algorithm with multiple criteria, analyzing mechanisms to reduce the data amount without losing the accuracy and efficiency requirements.

A.2.3. Multi-criteria motivation for tuning McGM.

In order to reduce algorithm memory consumption, we could afford not to store, as a particular solution, some of the temporary data computations, recalculating when necessary at the expense of reducing performance throughput under real time conditions. The most memory-demanding stages in the McGM algorithm correspond to the Spatial Filtering and Steering stages. On the one hand, an efficient way to reduce memory necessities was to perform the Steering stage with less θ angles at the expense of accuracy degradation. On the other hand, it was possible to use a numerical derivative [208] instead of the Gaussian counterpart in the Spatial Filtering stage in order to allow faster derivative recalculation. This alternative scheme was based on the fact of not requiring intermediate data computation storage by saving a huge amount of memory and to recalculate whenever data were used. A simple numerical differentiating filter was used based on the convolution commutative properties: $I \otimes G_x = I \otimes (G_0 \otimes D_x) = (I \otimes G_0) \otimes D_x$. The number of operations performed in $(I \otimes G_0) \otimes D_x$ are smaller than the Gaussian derivative filtering, making the convolution process faster.

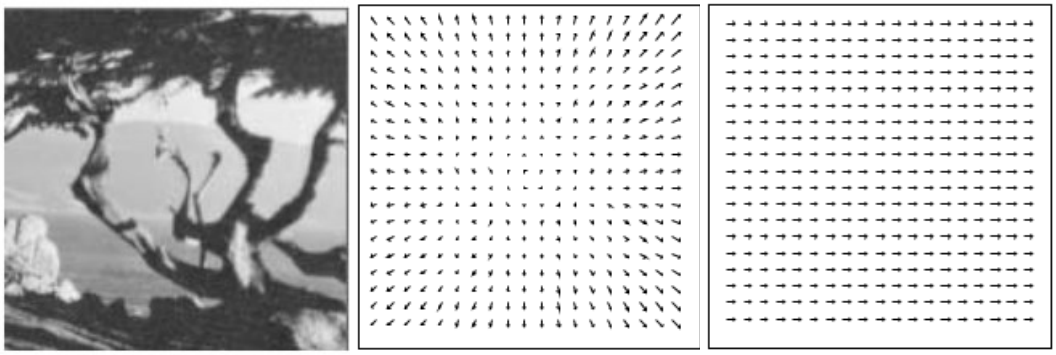
	x'	x''	x'''	x^{IV}	x^{V}
Filter degradation	0.003825	0.009415	0.018444	0.033701	0.060134

Tabla 21: Filter accuracy degradation using a numerical derivative instead of the Gaussian counterpart for first, second,... to the fifth derivative order.

Table 21 shows the error in computing $G_0 \otimes D_x$ to evaluate accuracy degradation. Filter degradation denotes $|(G_0 \otimes D_G) - (G_0 \otimes D_N)|$ difference where D_G and D_N corresponds to Gaussian and Numerical derivative filtering, respectively. As can be appreciated, loss of accuracy is not so important for 9–31 pixel filtering, reaching a maximum of 3 % error. A priori, we may conclude that performing numerical derivatives rarely creates considerable error.

Despite the unimportance of degraded filtering accuracy, an experiment comparing motion estimation degradation is carried out to evaluate the loss of accuracy in the overall algorithm. As benchmarks, we have used a couple of synthetic sequences widely accepted in this context: the *'diverging tree'* and the *'translating tree'*, both created by David Fleet at Toronto University [163] and presented in the Figure 71. The *'diverging tree'* shows an expansive motion of a tree (in camera zoom mode) with an asymmetric velocity range depending on the pixel position (null in the central focus and 1.4 pixels/frame and 2 pixel/frames in the left and right boundaries, respectively). The *'translating tree'* shows the translational motion of a tree with an asymmetric velocity range depending on the pixel position (zero to 1.73 pixel/frames and zero to 2.3 pixel/frames in the left and right border, respectively).
For an error metric, we used Barron [57], considered to be one of the most accepted metrics in the specialized literature.

Barron Equation (38) shows deviation from the correct space-time orientation, the velocity being a 3D unit direction vector. This vector wraps both modulus (speed) and phase (direc-



(I) (II) (III)

Figura 71: '*Diverging Tree*' and '*Translating Tree*' with their movement.

tion) in a single value reducing and reduce the rise of directional errors for small velocities.

$$\vec{v} = \frac{1}{\sqrt{u^2 + v^2 + 1}} (u, v, 1)^T \tag{38}$$

Since the vector is self-normalized, the angle between the measured velocity v_e and the correct one v_c is given by Equation (39). This error measurement is calculated for every pixel for which a velocity measurement was recovered.

$$\psi_E = \arccos (\vec{v_c} \cdot \vec{v_e}) \tag{39}$$

Table 22 shows an error in Barron's angle when used as a numerical derivative instead of a Gaussian counterpart in spatial filtering with a significant θ angles reduction in the steering stage. Columns $O(h)$, $O(h^2)$, $O(h^3)$, and $O(h^4)$ denote the observed error of Barron's angle when performed with numerical derivatives with first-, second-, third- and fourth-accuracy order, respectively. $\#\theta$ is related to the maximum number of θ angles projected in the Steering stage. The table shows the impact of half- or quarter-θs.

	$O(h)$	$O(h^2)$	$O(h^3)$	$O(h^4)$	$\#\theta/2$	$\#\theta/4$
Diverging Tree	0.9297	0.4982	0.4432	0.4020	0.0008	0.0122
Translating Tree	1.5185	0.7965	0.7762	0.6903	0.0015	0.0296

Tabla 22: Overall degradation measured as mean absolute error of Barron's angle.

As observed, the *'diverging tree'* experiment behaves reasonably well with numerical derivatives reducing their impact with a higher order of accuracy. Nevertheless, in the *'translating tree'* experiment, the algorithm is more vulnerable to numerical derivatives than the number of angles variation. Due to this disparity observed in Table 22, it is advisable the space of feasible solutions with any set of input data be explored. Given the large number of parameters to configure, on one hand relative to the McGM algorithm, and on the other hand those based on available resources, the use of genetic algorithms (GAs) can be useful to reduce

time-consuming exploration.

A.2.3.1. Multi-criteria optimization description.

The use of GAs arises from non-viability exploration with a large solution space. In our context, the target is to find a compromise in the reduction of the GPU's memory usage with negligible accuracy degradation that allows motion estimation system self-adaptation under appreciable environmental conditions and changes in a reasonable time.

The goal of the multi-objective optimization [209] is to simultaneously optimize several objectives that could be inconsistent. Considering the problems, some trade-offs among the different variables involved also need to be considered. In our context, we consider the following three-objective minimization problem:

$$\begin{aligned} Minimize\ z = (f_1(x), f_2(x), f_3(x)) \\ subject\ to\ x\ \epsilon\ X \end{aligned} \tag{40}$$

where *z* is the objective vector with 3 objectives to be minimized: execution time f_1 , memory usage f_2 , and loss of accuracy f_3 ; *z* is the decision vector, and X is the feasible region in the decision space, which corresponds to all possible McGM configurations with respect to the derivative decision and the number of angles involved. In GA terminology, *x* corresponds to a chromosome. In our context: D_x corresponds to the derivative to be computed in spatial filtering. This information is stored in a two-dimensional array whose values determine the way their derivative is computed by means of Gaussian or order-numerical differentiation. The two-dimensional array position is related to the derivative order.

The number of θ angles to be performed in the steering stage, which can be assigned as an integer.

A.2.3.2. Our multi-GPU implementation.

Over the last few years, a great number of multi-objective evolutionary algorithms have been developed [211, 212, 210]. A revision of the GA can be found in a tutorial [213], where the authors provide the revision's more relevant features.

For this study, we have chosen the NSGA-II [216] for its following advantages:

- Weights are not required, so it is not necessary to study the impact of $f_i(x)$ and assign them.
- Its computational requirement is *one*, which presents less computational complexity.
- Its 'good' behavior and ability to find a set of solutions near the true Pareto-optimal with few iterations.
- It's widely used and amply tested.

The NSGA-II is based on a fast non-dominated sorting procedure where a fast crowded distance estimation is carried out. It involves a simple crowded comparison operator [216]. The NSGA-II algorithm could be summarized in the next stages:

1. Initially, a random population is created in *pop*.
2. The population is sorted based on the non-domination scheme.
3. It is assigned a fitness, which means every individual of the population is ranked into levels. First-level or Pareto-front is most preferable.
4. A binary tournament selection and combination is carried out.
5. A mutation phase is done.
6. A combined population *R* comes from the union of an old *pop* with the new one *new_pop*. The population *R* is size $2 * pop\_size$.
7. *R* is ranked by means of the McGM algorithm and sorted according to a non-domination scheme.
8. New population *pop* is made from size *pop_size*.

The fast non-dominated sorting is the most computational cost part of the GA, because it involves ranking every individual of the population. We urge that this task be performed entirely on multi-GPUs since this is more efficient than using a CPU, from computational point of view. Most GA operators are executed in CPU due to its low computational demand.

To rank an individual of the population means to compute the McGM algorithm with chromosome configuration, to compute the derivatives in Spatial Filtering, and to compute the number of angles in the Steering Stage to be performed. Several levels of parallelism are exploited: a coarser level, where non-dominate sorting is evaluated in parallel on several GPUs, and finer level by means of data parallelism exploitation available in each stage of the McGM algorithm. Algorithm 13 summarizes our parallel implementation where *pop_size*, *ngens* and *%mutation* are GA input parameters which correspond to population size, number of generations, and mutation probability, respectively. The OpenMP paradigm is used to distribute a non-dominated sort across multiple devices by means of `#pragma omp parallel for` directives. Our implementation generates Pareto-optimal solutions with a set of motion estimation execution time, accuracy pixel error, and GPU memory usage points. This feature allows the choice of one of the best solutions, taking into account the available computational resources favoring the dynamic tuning depending on current conditions.

Algorithm 13 $pareto\_front = multiGPU\_NSGAII(popsize, ngens, \%mutation)$

$pop = random\_init\_Population(pop\_size)$ %NSGAII - 1^{st} stage
$Fronts = McGM\_fast\_nondominated\_sort(pop)$ %NSGAII - 2^{nd} & 3^{rd} stages
while $gen < ngens$ **do**
  #pragma parallel for shared(Fronts,R) private(i,GPU_{th},p_1,p_2)
  for all $i = 0; i \leq pop\_size; i = i + 1$ **do**
    $p_1, p_2 = select\_parents(pop(i))$
    $newpop(i).x = crossover(p_1, p_2)$ %NSGAII - 4^{th} stage
    $newpop(i).x = mutation(\%muration)$ %NSGAII - 5^{th} stage
    $R(i) = newpop(i) \bigcup pop(i)$ %NSGAII - 6^{th} stage
    $GPU_{th} = threadID()$;
    $Fronts = McGM\_fast\_nondominated\_sort(GPU_{th}, R(i))$ %NSGAII - 7^{th} stage
  end for
  $k = 0$
  $pop = 0$
  %NSGAII - 8^{th} stage
  while $sizeof(pop) < pop\_size$ **do**
    $front_k = get\_fronts(Fronts, k)$
    $pop = pop \bigcup get\_Relements\_in\_front(R, front_k)$
    $k = k + 1$
  end while
  $gen = gen + 1$
end while
$pareto\_front = get\_front(Fronts, 0)$

A.3. Results.

In this section the results obtained in this thesis will be presented. First of all, we are going to shown the outcomes for the McGM implementation in GPU and its consecuences. Then, the results to solve the main problem found in this implementation wil be exposed.

A.3.1. The McGM implementation in GPU.

A.3.1.1. Work environment.

To implement the McGM algorithm in a GPU, we used a system based on Tesla technology. The system contained 2 Intel Xeon E5530 processors with 4 cores (2.40 GHz with 8 MB cache memory and Hyperthreading technology), connecting to a Tesla C1060 GPU. The operating system used was Debian with a kernel version 2.6.38; the compiler was GNU Compiler Collection v.4.5.2, using the compilation options *O3 m64*; and CUDA v.2.3 was also used. This graphic card had a CUDA capability of 1.3, which meant it had 240 processing unified processors, which allowed the execution of 1024 threads in each multiprocessor. The memory hierarchy was 4 GB for global memory, 16 KB for shared memory, and 64 KB for constant memory.

A.3.1.2. Throughtput results.

The aim of the work was to analyze the throughputs as a potential benefit of specific graphic hardware architecture mapping in the framework of motion estimation algorithms. Regarding the temporal filtering, four alternative implementations were considered and are described as follows:

- *Base*: Baseline, where the input sequence data (all gathered frames) and the filter set are saved in the global memory. The data partitioning and computation process was described in Section A.2.2.1.
- *Global*: The sequence data is saved in the global memory and the filter coefficients in the constant memory.
- *Shared*: This is a hybrid approach. The N frames to be processed are saved in the shared memory; nevertheless, the filter coefficients remain in the constant memory.
- *Shared-optimized*: The spatial filter coefficients are continually saved in the constant memory, and the N frames go to shared memory, designing a loop-buffer structure. The buffer works as follows: (i) At $t = 0$ is initialized with the frame data index from first frame until $N + 1$. (ii) At $t = 1$, where the next one replaces the memory content of the first, frame by frame, $N + 2$. In this way, the number of access copies and data are reduced in the shared memory. It is suitable, thus, to exploit the information reusing.

Figure 72 shows the accelerations obtained for each case study. All throughputs were compared with a single CPU implementation, yielding a plethora of input resolution (size) and $N = 7$, 9 and 15 (the window size of temporal convolution).
As shown in Figure 72, the acceleration was noticeable for every GPU implementation version considered. Nevertheless, the best results were achieved when filter coefficients were stored in the constant memory, and the frames were stored in the global memory (Global version), reaching a throughput of $\times$85.

Shared and shared-optimized versions should have yielded better results because the latency access to shared memory is shorter than global memory, which could be explained by not having enough operations involved, which permitted the hosting of data directly in the multiprocessor registers (16K registers of 32 bits). For example, with $N = 15$, the kernel operates with 3840 elements/block, which is significantly lower than the 16K architecture registers. In Figure 72, the throughputs increased as frame size increased. This result was consistent with the awaited output, because the number of CUDA-blocks was longer, so the use of GPU resources was also longer.

With regard to spatial filtering, we assigned temporal filtering to the window with $N = 15$ to be able to compare the results with previous stages. Taking the size frame from the previous experiments, the effect on the spatial window size and the final order derivatives in the global throughput were analyzed.

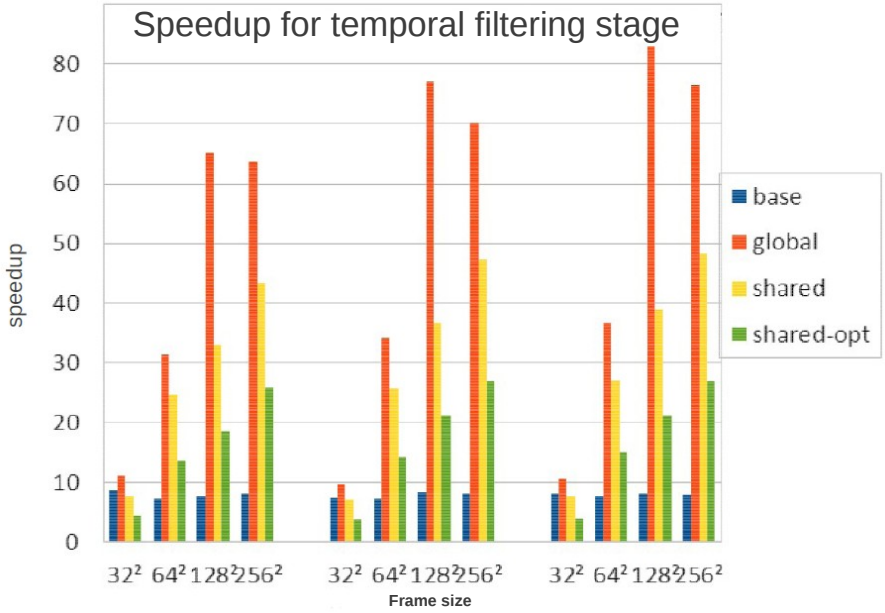


Figura 72: Speedup from the temporal filtering stage for several frame resolution sizes and filter window sizes (N).

Figure 73 shows the accelerations with fourth-order spatial filters. The window size was set with different values (7, 9, 15 and 31). The spatial derivative order did not affect the throughput, which was also consistent with other observed results. The throughput was better as the filter increased, because the parallelism grain in the convolution computation was exploited. Accelerations up to $\times 25$ were reached.
The next analysis corresponded to the steering stage in charge of orienting derivatives. The computation was based on forming a linear combination from all previous spatial derivatives. Therefore, taking the spatial derivatives and applying a weighting function was the main operation to complete. The input information was stored in a global memory.

At the temporal filtering stage, the input data set was mapped directly over the architecture registers, because the data volume was not too large. Shared memory was used to store the steering weights.

Figure 74 shows the results for the different frame resolutions and orientations. The angles considered were 60°, 30° and 15° , which corresponds to 6, 12, and 24 orientations, respectively. It should be noted that except for small frames (64×64 pixels), the number of angles barely made an impact on the final performance. A 24-angle configuration (each 15°) and larger frame resolutions (256×256) delivered the best results because the parallelism rate was greater.

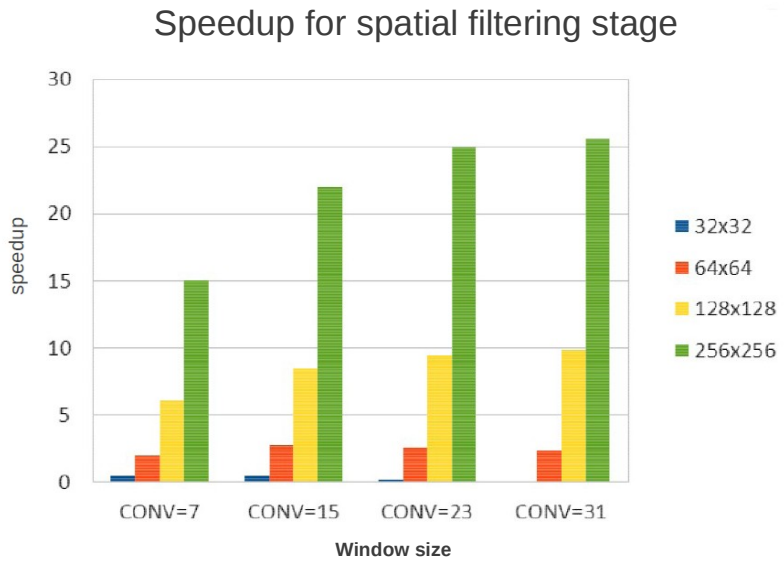


Figura 73: Speedup of the spatial filtering stage for several sizes of frame resolution and filter window size.

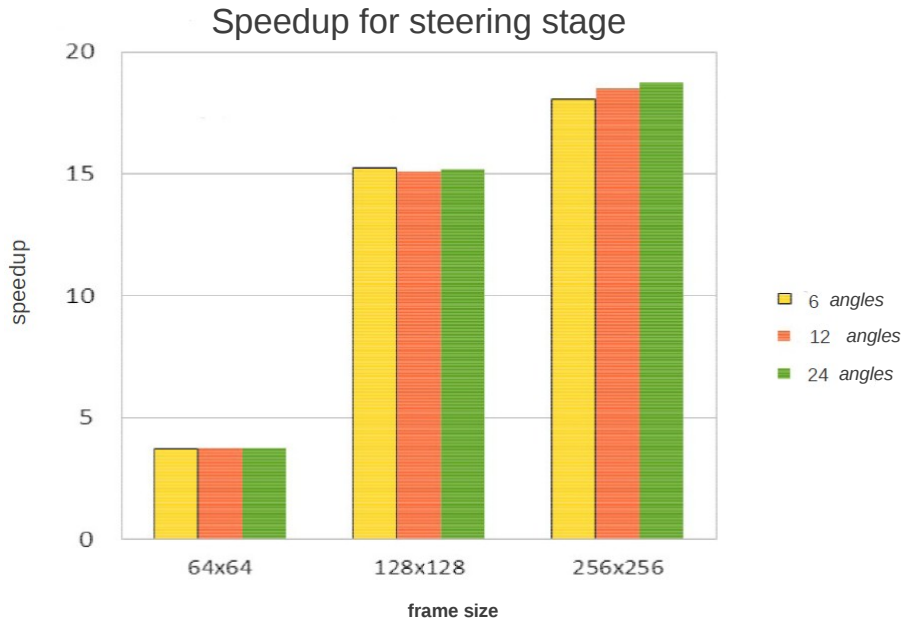


Figura 74: Speedup for the steering stage for several frame resolution sizes and orientations.

Figure 75 illustrates the acceleration for the Taylor expansion stage. We would like to emphasize that the pixel level parallelism grain was exploited to avoid expensive synchronization stages involved in reduction operations. This fact motivated incremental throughput rates

from $\times 5$ to $\times 38$ for 64×64 and 256×256 resolutions, respectively.

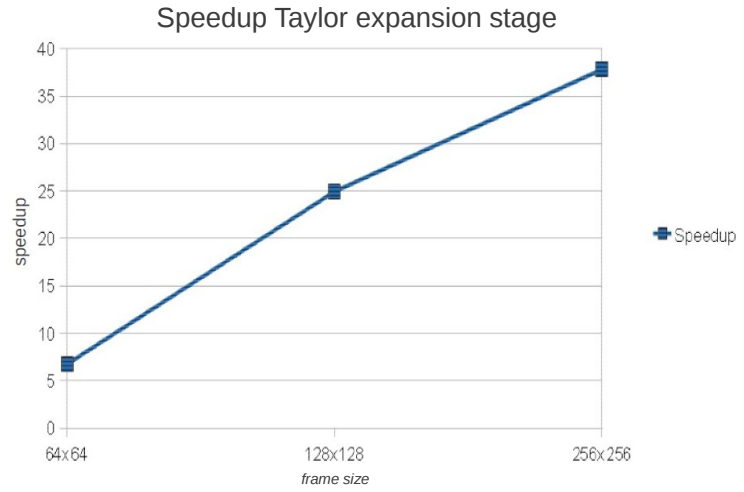


Figura 75: Speedup from the Taylor expansion stage using different resolutions.

Figure 76 presents the extraction of velocity primitives (stages V+VI) which delivers the final modulus and phase pixel values. In this stage, a throughput of $\times 23$ at 256×256 resolution size could be reached.

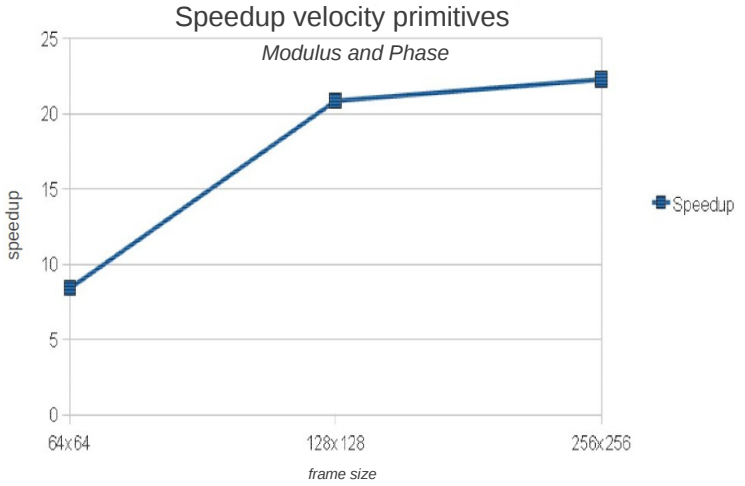


Figura 76: Speedup from velocity primitives with different resolutions.

	Init. GPU (s/pixel)	Temp. filtering (Mpixel/s)	Spatial filtering (Mpixel/s)	Steering (Mpixel/s)	Taylor (Mpixel/s)	Velocity (Mpixel/s)	**Total stages (Mpixel/s)**	**Total (fps)**
CPU (32^2)		10.30	21.63	90.99	217.87	247.24	**6.14**	**6327**
Best CPU (32^2)		13.20	12.48	130.75	369.79	289.62	**6.26**	
GPU (32^2)	3.28E-5	124.69	0.62	4.66	8.04	50.40	**0.50**	**375.7**
CPU (64^2)		12.85	1.44	2.09	3.78	20.14	**0.64**	**195.2**
Best CPU (64^2)		51.30	1.30	9.03	16.76	72.18	**1.66**	
GPU (64^2)	9.08E-6	495.98	2.55	15.77	25.54	169.08	**2**	**296.4**
CPU (128^2)		13.97	0.92	1.20	2.06	11.53	**0.39**	**30.72**
Best CPU (128^2)		70.87	0.77	5.04	14.23	66.59	**1.2**	
GPU (128^2)	7.21E-6	1166.12	8.79	36.17	51.49	240.65	**6.03**	**210.8**
CPU (256^2)		21.50	1.05	1.30	1.70	12.98	**0.41**	**8.631**
Best CPU (256^2)		154.75	1.11	3.89	7.95	93.27	**1.15**	
GPU (256^2)	2.25E-6	1724.63	27.56	47.62	64.20	289.21	**13**	**99.64**

Tabla 23: GPU, graphic processing unit; CPU, central processing unit.

Table 23 compares the performance for each stage and an overall throughput for frame size resolutions in a GPU versus CPU. For this experiment, the starting model configuration featured included three temporal derivatives and a temporal convolution window of 15 frames (Stage I); five spatial derivatives and a spatial separable convolution window of 31 pixels (Stage II); and, finally, 12 steered angles (Stage III). CPU row refers to the results obtained in the system based on Intel Xeon processors using a single core. Best CPU refers to the best configuration achieved using several cores. The exploitation of multicore parallelism has been performed by means of OpenMP directives, which are directly supported by the GNU Compiler C++ compiler. And GPU row shows performance rates observed in a Tesla C1060 GPU.

Foremost, we would like to note that for a small resolution such as 32×32, the exploitation of multicore is negligible, it is only worth the use of two cores. This is motivated by scarce data parallelism degree available in this resolution films. However, for other resolutions, multicore speedups are significant but not impressive. Although, the configuration that offers better results corresponds to maximum utilization (16 threads), its scalability in terms of efficiency is poor (efficiency on average is 30 %). This behavior comes from the fact that multicore system shares some critical resources such as last level cache or bus interconnection, which acts as bottleneck due to resources competition.

Finally, the Total stages column indicates the complete performance of the GPU versus CPU. Regarding CPU versions, acceptable performance was only achieved for small resolution sizes (32×32). In this case, the data information to be performed was so small that it could be stored in the first levels of cache memory, which offered good performance rates in terms of Mpixel/s due to high spatial and temporal data locality. For other resolutions, GPU versions improved on their CPU counterparts by far, because cache misses are often due to a large amount of processed data. For the last tests performed (the 256×256 resolution), it should be noted that almost 12 Mpixel/s were achieved, which means that out of 185 frames per second so real-time requirements were achieved.

Movie resolution	64×64	128×128	256×256
Speedup (single CPU)	3.125	15.46	31.70
Speedup (best CPU config.)	1.195	5.03	11.27

Tabla 24: Speedups comparing single CPU configuration and best CPU configuration. GPU, graphic processing unit; CPU, central processing unit.

Table 24 summarizes the speedup obtained for the different resolutions studied. According to total stages results, a successful throughput of ×32 for 256×256 resolution was observed. We would like to emphasize that GPU's version beats by far CPU's counterpart even in its best configuration. For higher resolutions, greater acceleration was expected, because a higher parallelism degree could be exploited.

A.3.1.3. Visual results.

To demonstrate the visual results of the complete model, Figure 77 is used as an example of real stimuli (a gas explosion), where red lines denoted the expansive nature of the motion in the original frame. The motion phase was shown and color-coded from the boundary frame. The modulus was represented by a linear intensity range of gray scales (black means no motion, whereas white is high speed). The lower part of Figure 77 showed the two outputs (phase at left, and modulus at right) of the algorithm implemented. Indeed, the motion recovery of the present model could then be fully appreciated.

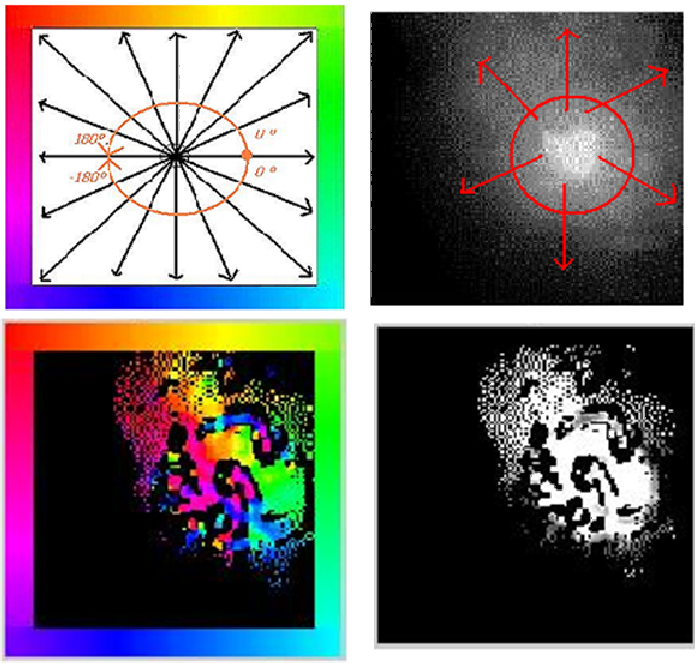


Figura 77: Input stimuli. Modulus and phase outputs of the algorithm.

It was necessary to use an error metric to measure the relative error of the GPU implementa-

tion. One of the most accepted metrics in the specialized literature was one from Barron [57], exposed in equations (38, 39).

This error measurement was calculated for every pixel for which a velocity measure was recovered or the average error computed. The GPU implementation was preliminarily tested with synthetically generated stimuli. Two test scenes were used (Figure 78), a translating sine wave moving in the x-direction with a velocity $v = (-1, 0)$ and a translating plaid with velocity $v = (1, 1)$.

The parameter set used was $\sigma = 1{,}5$, $\alpha = 10$,$\tau = 0{,}25$, and an integration zone of $p, q, r = 11 \times 11 \times 11$ following from (27) to (30) equations. The average errors for these stimuli were 0.01 and 0.40, respectively.

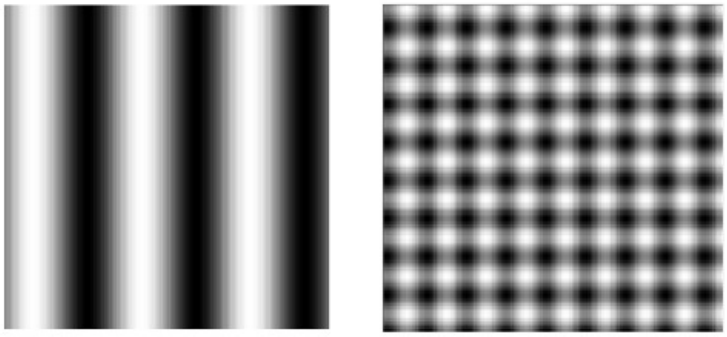


Figura 78: Stimuli used in evaluating the model.

A.3.1.4. Comparison with other optical flow implementations.

The errors for the McGM with 24 orientations were noted. All algorithms performed very well within this sequence. Figure 79 shows the speed measurements for the number of orientations taken. The ability to obtain differing information from each orientated channel was one of the points that enhanced the robustness of the model. It is possible to see how the error kept approximately constant for 18 with a minimal configuration of orientations.

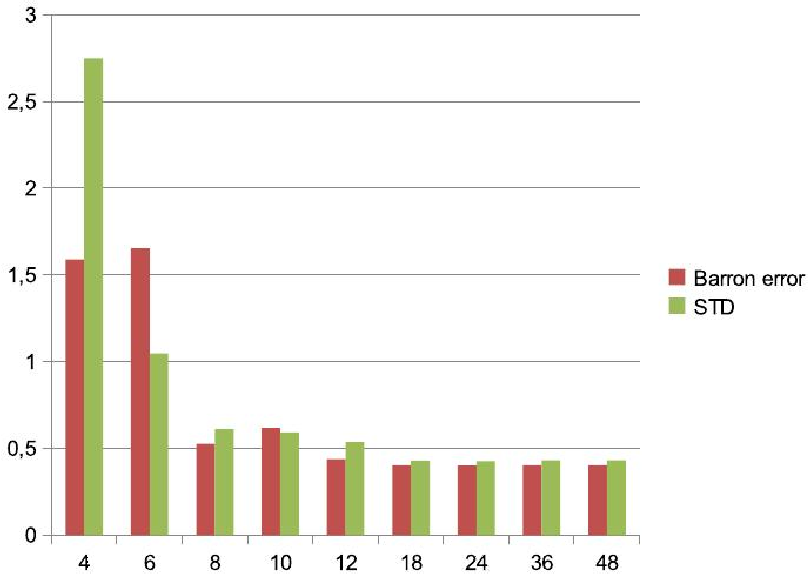


Figura 79: Speed measures versus orientations taken.

The results from Table 25 show that the present implementation outperformed all of the other algorithms for this simple sequence. The single versions of McGM reports the same error than the GPU versions because all floating points operations have been mapped in the GPU without accuracy loss.

Method	Angular Error	Standard deviation
Horn & Schunck [8]	0.97	2.62
Modified Horn & Schunck [197]	0.73	0.94
Lucas & Kanade [6]	2.47	0.16
McGM (GPU version) [169]	0.40	0.43

Tabla 25: Translating plaid errors in degrees.

We encountered more complex stimuli, presented in Figure 80, including the famous synthetic sequences called the *'Diverging Tree'* and the *'Translating Tree'*, explained in the chapter A.2.

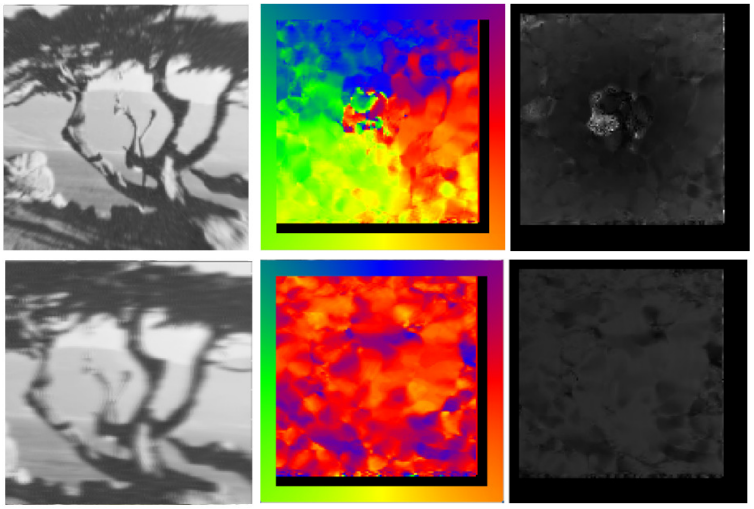


Figura 80: The temporal blur of the *'Diverging Tree'* (upper) and *'Translating Tree'* (lower) sequence, and the final phase and modulus of the graphic processing unit implementation.

We observed an average difference magnitude of 0.291 and an average angular error of $12.47° \pm 18.51$. If we threshold in the value of the first temporal derivative, we could decrease the error until $11.29°$ with a density of 64 %. Some of the best results reported for other algorithms for this sequence were with threshold versions with low output density. The McGM was consistent with its results and estimated the velocity for all regions with the same accuracy; the threshold did not improve results significantly. The computed velocity was practically uniform, but the direction was varied. Consequently, the errors were high. Table 26 shows an average angular error measured in degrees.

	'Diverging Tree'		'Translating Tree'	
	Angular error	Standard deviation	Angular error	Standard deviation
Horn & Schunck [8]	12.77	12	38.99	27
Uras et al. [198]	5.75	4.01	0.71	0.81
Lucas & Kanade [6]	8.07	11	11.69	18
Fleet & Jepson [163]	1.24	0.9	0.36	0.41

Tabla 26: '*Diverging Tree*' and '*Translating Tree*' errors measured in degrees .

Finally, Figure 81 shows a real scene performed with a static camera with its motion segmented as output. The true ground motion was not available for this sequence, because it was not synthetic.

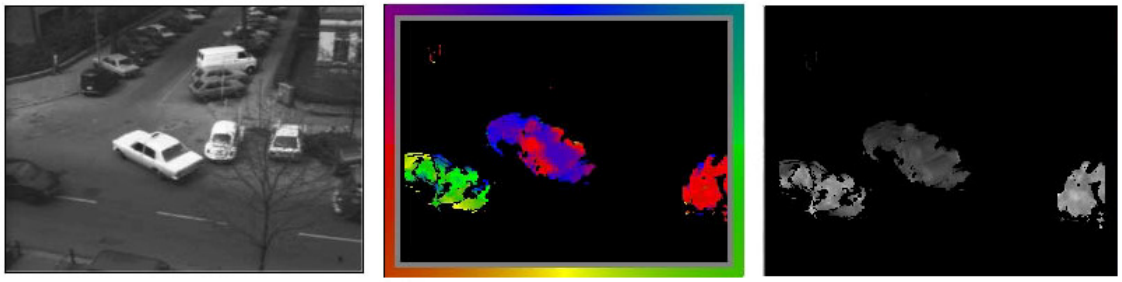


Figura 81: Taxi sequence. Real sequence.

A.3.2. Multicriteria optimization results.

Potential benefits of GPUs in the McGM context have been explored, where we studied the viability in these novel devices. Throughput results with respect to a single CPU were satisfactory enough in terms of performance, achieving $\times 32$ speedups for 256^2 resolution movies.

We would like to emphasize that this particular GPU-based motion estimation scheme is an alternative to consider in terms of Mpixel/s compared to other purpose systems used for such motion-estimation algorithms, although the algorithm features create a bottleneck, specifically when memory requirements are increased in each stage, with an upward trend. This disadvantage limits GPU viability. Attending to the largest memory usage configuration considered in the study presented before, 3.5 GB of global memory was used, which was close to the capacity limit of a single GPU. Although the memory capacity is greater for GPUs nowadays, this problem is still present with larger data input resolutions.

The scope of this study is to explore mechanisms in order to reduce the data amount without losing the efficiency and accuracy requirements. To highlight the memory handicap of GPUs, Table 23 shows the summarized performance results observed using the McGM algorithm in a graphic device compared with a single CPU. The performance observed at each stage of the algorithm is shown in Mpixel/s (Mpps), and the global throughput with a particular model configuration, as follows: three temporal derivatives, a temporal convolution window of 15 frames, five spatial derivatives, a spatial separable convolution window of 31 pixels, and 12 angles steered. Moreover, as shown in this table, GPU implementation amply fulfilled real-time requirements in all of the resolution configurations considered. This is further shown in the last column, which corresponds to overall performance, which was measured in frames per second (fps). While general-purpose processors can only reach real-time rates for small video resolutions, GPU-based systems enabled higher resolution movies where more DDR memory capacity was available.

A.3.2.1. Work environment.

The systems used are based on Tesla technology. The first one consists of 2 Intel Xeon E5645 processors with six cores (2.40 GHz with 12 MB cache memory and Hyper-threading technology) and 2 Tesla M2070 GPUs. The second one is equipped with 2 Quad Intel Xeon

E5530 processors (2.40 GHz with 8 MB cache memory and Hyper-threading technology), connecting with 4 Tesla C1060 GPUs. In both cases, the operating systems are Debian 2.6.38 kernels; the compiler used is a GNU g++ v.4.4 with compilation flags *-O3 -m64 -fopenmp* and CUDA C/C++ SDK v.4.2 with *-O3 -fopenmp -arch sm_20/13* flags enabled.

The system based on Tesla M2070 incorporates Fermi technology, but due to a scarce number of devices available, a scalability study has been completed with a system based on 4 Tesla C1060 GPUs that allow projections be made of parallel efficiency rates in more modern systems.

A.3.2.2. Multicriteria results.

Multi-objective GAs are used to look for optimal solutions in a huge search space. In our context, they are employed to achieve a set of optimal solutions that reduce the GPU's memory usage in the McGM algorithm without losing significant accuracy in the motion estimation scenario. As previously mentioned, the tests were performed using the '*diverging tree*' and the '*translating tree*' benchmarks, which are widely accepted in this area.

The first experiment was to evaluate both the convergence of the GA and the set of optimal solutions reached. For this purpose, a Euclidean distance metric between consecutive solutions was employed as described in [216]. The GA implemented incorporated a stop condition based on a Euclidean metric when a certain number of iterations remained invariant to ensure the non-dominant solutions converged to the optimal Pareto-front.

Figure 82 shows the evolution of the set of non-dominated solutions throughout the iterations with a severe stop condition. To facilitate its visualization, only the GPU's memory reduction and the error difference were included, although the GA also optimizes the motion estimation time. Barron's angle error Ψ_E corresponds to the difference of mean Barron error with respect to the original McGM counterpart. The number of points in the optimal front is a subset of the final solution because execution time components have been removed. This figure shows only 15 % of Pareto-pairs for the '*translating tree*' while those in the '*diverging tree*' are displayed at 40 %.

In this experiment, the population size was fixed to 500 with 1 % mutation probability. The results obtained indicated that after a certain number of generations, the GA barely improved the non-dominant solutions, although it reported new pairs.

Population size only affects the final execution time, achieving results of an optimal solution with similar quality. Empirically, 1 % of mutations reported better GA performance. Higher mutation rates only suppose significant variations between consecutive generations, which means higher generations are necessary to reach the convergence criterion. Particularly, greater mutation rates suppose a higher number of iterations, which varies between 15 to 320 %.

As shown in Figure 82, optimal solutions are generated with significant reduction in memory

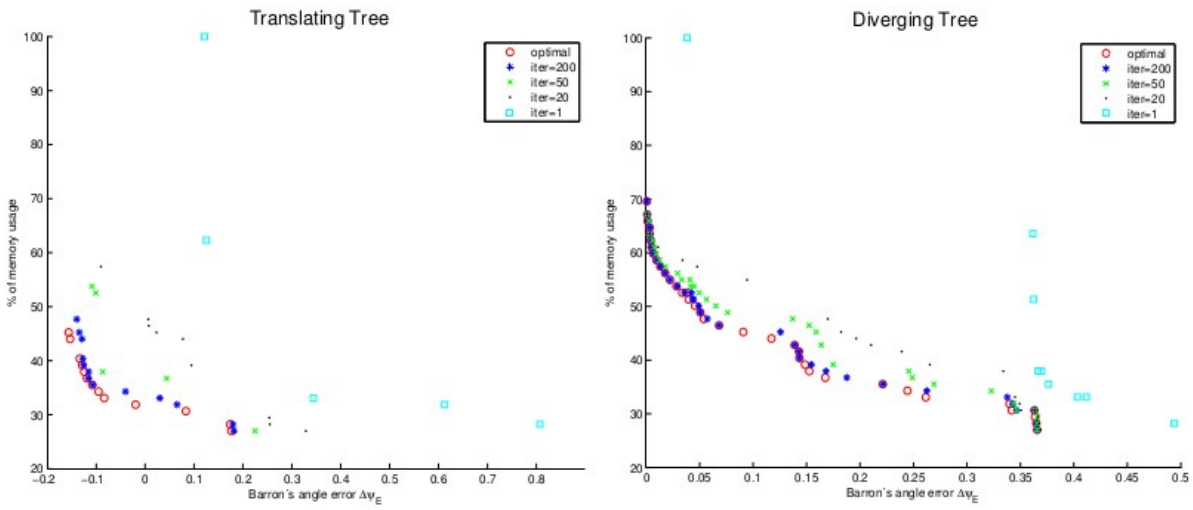


Figura 82: Evolution of the average objective in the GA for '*diverging tree*' and *translating tree*' sequences.

requirements, achieving even more accurate solutions than the original McGM's algorithm for the '*translating tree*' benchmark.

A.3.2.3. Multi-GPU results.

Table 27 shows GA time measured in seconds in a system based on the Tesla M2070 for the best GA configuration: 1 % of mutation rate configuration, 500 individuals and a severe convergence condition to find the Pareto-front. The '*diverging tree*' was used as a benchmark, although similar performance rates were observed with the '*translating tree*'. Note that the benchmark choice only affects the number of generations processed to reach an optimal solution. As expected, the fitness evaluation is the most costly stage of the GA by far. The information exchange overhead between host and devices is not so relevant, which reports satisfactory speedups of $\times$1.79 for 2 GPUs.

Tesla M2070	$t_{CPU}(s)$	$t_{GPU}(s)$	$t_{Comm}(s)$
1 GPU	1.24	22495.6	869.5
2 GPUs	124.2	12464.9	447.4

Tabla 27: Multi-GPU execution times for Tesla M2070 based system.

Analogous results were obtained in a system with a larger number of graphic devices. Table 28 shows even higher accelerations when 2 GPUs are enabled. Furthermore, it is noticeable that scalability rates remain satisfactory with 4 GPUs, achieving $\times$3.71 speedups. Computational results show that our multi-GPU implementation is efficient in terms of scalability (95 and 93 % using 2 and 4 GPUs, respectively), and the tendency indicates that GA convergence times would be even lower if more computational resources were available. We can conclude that this successful scalability makes GAs useful for solving problems of this nature. These

good performance results are due to both a well-balanced workload and low overhead involved in data exchange.

Tesla C1060	$t_{CPU}(s)$	$t_{GPU}(s)$	$t_{Comm}(s)$
1 GPU	1.18	23748.0	2025.8
2 GPUs	278.52	12613.1	1022.5
4 GPU	153.28	6248.4	513.5

Tabla 28: Multi-GPU execution times for Tesla C1060 based system.

Moreover, the use of multiple levels of parallelism reports multiplicative accelerations: first, the speedups achieved in the multi-GPU system, which can be up to $\times$3.71 with 4 GPUs enabled; and second, the accelerations up to $\times$32 the can be achieved by exploiting the data parallelism on a GPU. On one hand, the combination of both accelerations allows the reduction of exploration time to reach an optimal solution in 99.2 % compared with a general-purpose processor. On the other hand, the use of a multi-GPU system not only reports greater FLOPS rates than a CPU, but it is also beneficial in terms of power consumption (MFLOPS/watt).

Moreover, although GA search times are important, their use encourages getting suboptimal solutions that meet the requirements of response time or resource consumption, and as GAs evolve, they are gradually refined. This feature, coupled with the chance of a population size reduction, supposes an impressive simulation times decrease which opens the possibility to build an intelligent system that auto-corrects/adapts depending on the specific requirements or substantial environment changes.

A.3.2.4. Visual result.

Finally, visual results are presented for both benchmarks considered. Figure 83 shows the main differences in motion estimation outputs in the '*diverging tree*' benchmark. The original McGM outputs appear at the top of the figure; in the center and at the bottom their counterparts with GA solutions for 75 and 50 % memory requirements. It is also remarked the motion estimation time consumption (ME_time). The motion phase (the direction of the pixels) is color-coded from the boundary frame (each particular color pixel points outward to the border color frame). The modulus or velocity magnitude is represented by a linear intensity of gray scales.

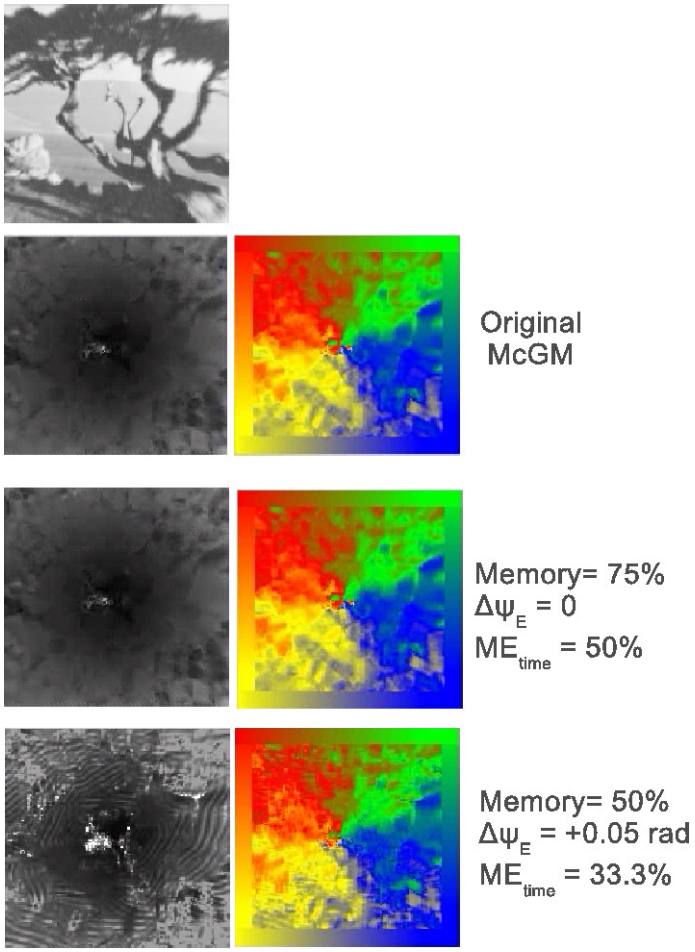


Figura 83: The temporal blur of the original stimulus, modulus, and motion phase with the original McGM algorithm (top), and a z GA solution of 75 % (center) and 50 % (bottom) of memory usage for the '*diverging tree*'.

Similarly, Figure 84 displays GA solutions for the '*translating tree*' benchmark.
For the '*diverging tree*', a reduction of 75 % in memory usage returns the same precision using the Barron metric and 50 % of the McGM execution time (ME_{time}) compared to the original algorithm. However, the configuration that reduces memory usage by 50 % degrades the accuracy in 22 % with a speedup of $\times$3.3.

For '*translating tree*' benchmark, a solution with half of memory requirements is more accurate (Barron's error is 0.13 radians less than the original McGM) and $\times$3.5 faster.

Despite Barron metric's popularity by the scientific community in the context of motion estimation, some authors [165, 166] point out specific performances due to its asymmetry and its bias of large flow vectors.

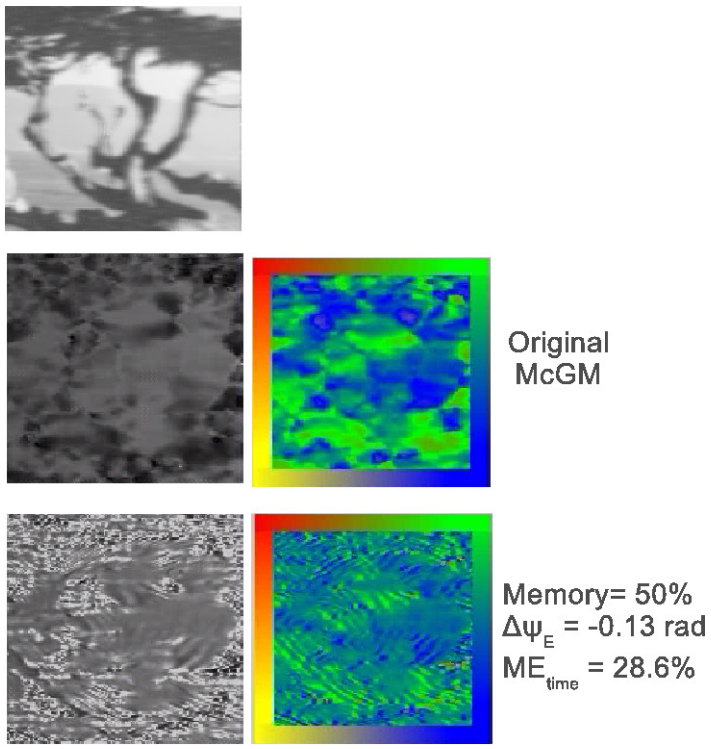


Figura 84: The temporal blur of the original stimulus, modulus, and motion phase with the original McGM algorithm (top), and a z GA solution of 50 % (bottom) of memory usage for the '*translating tree*'.

A.3.2.5. Other error metrics.

Although Barron's metric [57] is probably the most used in the motion estimation scope, there are other metrics used by Machine Vision community that must be taken into account in order to enhance the visibility and generality of the results obtained.

Otte and Nagel [165, 166] remarked the fact of asymmetry and bias for extensive optical-flow vectors. Based on this drawback, it is proposed a new metric which accounts the magnitude difference between bidimensional ground truth flow vector (ofv_c) and the estimated one (ofv_e) as shown in the Equation (41):

$$\Psi_{O\&N} = \left\| o\hat{f}v_c - o\hat{f}v_e \right\| \tag{41}$$

McCane et al. [166] claims this is not sufficient due it gets discount error in regions of small flows. They propose two metrics in order to overcome these problems. The first metric is the angle difference between the correct tridimensional v_c vector and the estimated one v_e used in the Barron's metric (Equation (42)) but the third component is replaced by δ. In our experiments we assign $\delta = 0{,}75$. This threshold modulates the error considering less significant in zones of small flow than in zones of large flow.

$$\Psi_{McCane_A} = cos^{-1}\left(\hat{v_c}, \hat{v_e}\right) \tag{42}$$

An additional metric is proposed, such as the normalize magnitude of the vector difference between the estimated and the correct tridimensional flow vectors. The normalization factor is the magnitude of the correct flow, it is taken into account the effect of small flows using a significance threshold T as shown in Equation (43). It is chose T to be 0.5 pixels. The effect of this threshold would result in a normalized error equal to the unity.

$$\Psi_{McCane_B} = \begin{cases} \frac{\|v_c - v_e\|}{\|v_c\|} & if \quad \|v_c\| \geq T \\ \frac{\|v_e - T\|}{\|T\|} & if \quad \|v_c\| < T \leq \|v_e\| \\ 0 & if \quad \|v_c\| < T > \|v_e\| \end{cases} \tag{43}$$

Figure 85 shows the trend of the NSGA-II algorithm considering McCane and Otte&Nagel metrics. '*Translating*' and '*diverging tree*' sequences are also used as benchmarks.

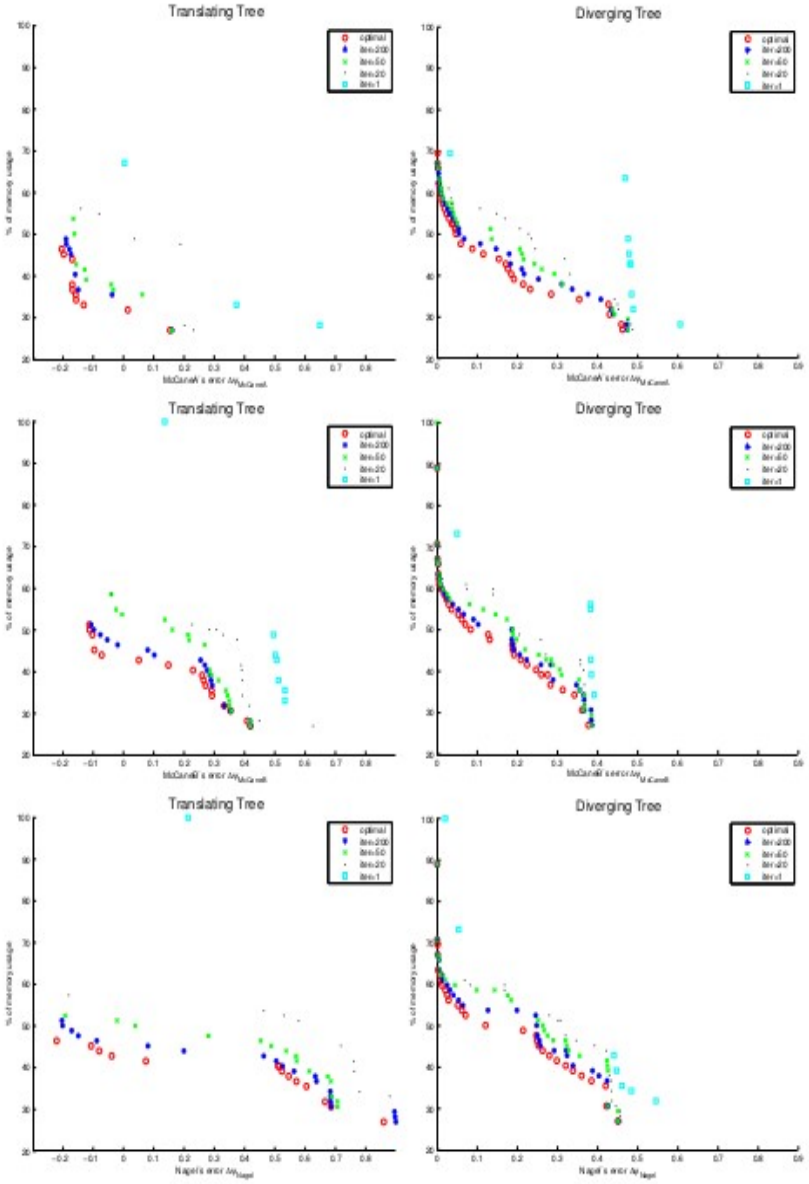


Figura 85: Evolution of the average objective in the GA for Diverging and Translating Tree sequences for McCane and Otte&Nagel metric. The Pareto-front show the optimal configuration depending to the metric applied.

For the sake of clarity, Table 29 summarizes the main successful configuration reported by GA execution. Results observed are consistent with regardless of the metric used. Meanwhile for '*translating tree*' improves Motion Estimation effectiveness with a significant reduction of memory requirements, for '*translating tree*' no degradation is observed for 75 % memory usage for any metric performed. From the viewpoint of the execution times, performance

results are as expected. On the one hand a reduction in memory requirements by 50 % are translated into speedups from 3.3× to 4×. On the other hand, 75 % of memory consumption reports an average of 50 % in motion estimation execution time.

Error Metric	Benchmark	Memory	ME_{time}	Accuracy ($\Delta\Psi$)
Barron	*Translating Tree*	50 %	28.6 %	-0.13
		75 %	50.0 %	0.00
	Diverging Tree	50 %	33.3 %	0.05
$McCane_A$	*Translating Tree*	50 %	26.9 %	-0.12
		75 %	49.1 %	0.00
	Diverging Tree	50 %	28.9 %	0.05
$McCane_B$	*Translating Tree*	50 %	29.6 %	-0.11
		75 %	49.2 %	0.00
	Diverging Tree	50 %	34.5 %	0.09
Otte&Nagel	*Translating Tree*	50 %	25.3 %	-0.21
		75 %	49.1 %	0.00
	Diverging Tree	50 %	35.7 %	0.12

Tabla 29: Best configuration achieved for a reduction of 75 and 50 % memory requirements using McCane and Otte&Nagel metric.

A.4. Conclusions.

The efficient GPU implementation of a neuromorphic algorithm was presented with a set of characteristics suitable for complex or noisy environments that could be applied in tracking robotics, navigation, security, surveillance, and so on. This system could also be used in the field of neuroscience research, extending the complexity of bio-inspired algorithms.

The use of graphics hardware as an optical flow coprocessor was an interesting and affordable choice, with possible speedups of ×32. However, overhead related with GPU memory management and the exchange information process between host and device could reduce acceleration rates slightly. Moreover, to take advantage of GPU parallel computation, it is crucial that tasks be distributed in a suitable way and, additionally, that complex memory hierarchy be exploited efficiently, which is not a straightforward task in most cases.

First study was focused on performance evaluation; however, due to the expansive nature of the algorithm, as long as the stages are carried out, memory requirements increase. Attending to the most consuming configuration considered, 3.5 GB of global memory were used, which is close to the limit on a single GPU. Although memory GPU capacity is greater nowadays, this problem will still present larger frame resolutions. Therefore, we are currently exploring mechanisms to reduce the data amount and design strategies to exploit the parallelism between several stages as analogy with a pipelined processor.

Therefore, a new and highly parallel approach is presented to overcome the GPU memory

usage problems that occurred in our previous implementation of a well-known neuromorphic motion estimation algorithm. This context provides the main motivation for using evolutionary algorithms to solve multi-criteria optimization problems. The use of GAs based on a multi-GPU scheme allowed for quick exploration of feasible solutions with any set of input data. The choice of NSGA-II is motivated by the good results observed in a few iterations and a near-optimal Pareto-front.

From the viewpoint of reaching a solution that meets the requirements of memory consumption, we observed:

- For '*diverging tree*', a reduction of 75 % in memory usage returns the same precision as the all metrics considered and 50 % of the McGM execution time compared to the original algorithm. A configuration that reduces memory usage by 50 % degrades the accuracy from 15 to 25 % with a range of speedup which varies from $\times 2.8$ to $\times 3.5$.
- For *'translating tree'*, a configuration that has half of the memory requirements is more accurate in terms of error and is between $\times 3.3$ to $\times 4$ faster.

From the point of view of multi-GPU efficiency is observed:

- Successful performance of $\times 3.71$ speedups are archived when four GPUs are enabled.
- Our implementation is a scalable approach due to both a well-balanced workload and low-impact communication between host and device.
- A found multiplicative effect: $\times 3.71$ speedups in a multi-GPU system by $\times 32$ acceleration by means of exploiting the data parallelism on a GPU. An impressive GA time in reaching an optimal solution in 99.2 % compared with a CPU.
- An alternative to be considered in terms of power consumption (MFLOPS/watt).

Because of these encouraging results, the possibility exists for building an intelligent system that auto-corrects/adapts depending on specific requirements or environmental condition variations as the GA evolves.

We are also working in a complete comparison of the present neuromorphic implementation with the FPGA implementation previously implemented in [94] and constructing a design based on real-time feedback. The circuit, in that way, is able to self-adapt the constraint of the environment and predict its evolution, trading off between accuracy and efficiency.

We would like to emphasize that bio-inspired motion estimation based on GPU platforms is an alternative to consider in terms of Mpixel/s, especially compared with other specific purpose systems used for such algorithms. In addition, current trends seem to indicate that these ratios will increase as future technology improves.

In considering future work, the focus would be on the optimal implementation of the spectral extension of the model [175], addressing color distinction, which, to the best of our knowledge, is not in existence with any hardware implementation that makes use of graphic hardware.

Reusing the auto-correct/adapt system with an environment predictor is also another field to study, with the possibility of real-time execution and self reconfiguration depending on the external constraints and resources available in the platform. This system is expected to contribute to the new machine vision trends, useful for many real-world applications.

Referencias

[1] J.M. Giménez-Amaya. Anatomía funcional de la corteza cerebral implicada en los procesos visuales. *Rev. Neurol*, 30(7):656–662, 2000.

[2] F. Reinoso-Suárez and C. Cavada. Confluencia de redes neuronales implicadas en el procesamiento de la memoria en primates. *Progresos en Biología Celular*, 1:263–7, 1993.

[3] Samsung Delivers Strong 14nm FinFET Logic Process and Design Infrastructure for Advanced Mobile SoC Customers. `http://www.samsung.com/global/business/semiconductor/news-events/press-releases/detail?newsId=12461`, 2012.

[4] H. Oh and H. Lee. Block-Matching algorithm based on an adaptive reduction of the search area for motion estimation. *Real-Time Imaging*, 6(5):407–414, October 2000.

[5] C. Huang and Y. Chen. Motion Estimation Method Using a 3D Steerable Filter. *Image and Vision Computing*, 13(1):21–32, 1995.

[6] S. Baker and I. Matthews. Lucas-Kanade 20 Years On: A Unifying Framework. *International Journal of Computer Vision*, 56(3):221–255, 2004.

[7] Berthold KP Horn. Determining lightness from an image. *Computer Graphics and Image Processing*, 3(4):277–299, 1974.

[8] B.K.P. Horn and B.G. Schunck. Determining Optical Flow. *Artificial Intelligence*, 17:185–203, 1981.

[9] R. Paquin and Eric Dubois. A Spatio-Temporal Gradient Method for Estimating the Displacement Field in Time-Varying Imagery. *Computer Vision, Graphics and Image Processing*, 21:205–221, 1983.

[10] V. Bruce, P.R. Green, and M.A. Georgeson. Visual Perception: Physiology, Psychology & Ecology. *third ed., Laurence Erlbaum Associates, Hove*, 1996.

[11] K. Claeys, D. Lindsey, E. De Schutter, and G.A. Orban. Higher order motion region in human inferior parietal lobule. *Evidence from fMRI.Neuron*, 40:631–642, 2003.

[12] A. Johnston, P.W. McOwan, and C.P. Benton. Biological computation of image motion from flows over boundaries. *J Physiol. Paris*, 97:325–334, 2003.

[13] [on line]. `http://developer.nvidia.com/cuda-downloads`.

[14] Linda G. Shapiro and George C. Stockman. *Computer Vision*. Prentice Hall, 2001.

[15] Tim Morris. *Computer Vision and Image Processing*. Palgrave Macmillan, 2004.

[16] Bernd Jähne and Horst Haußecker. *Computer Vision and Applications, A Guide for Students and Practitioners*. Academic Press, 2000.

[17] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision*. Thomson, 2008.

[18] Edward H Adelson and James R Bergen. The plenoptic function and the elements of early vision. *Computational models of visual processing*, 1(2), 1991.

[19] William T. Freeman, Egon C. Pasztor, and Owen T. Carmichael. Learning low-level vision. *International journal of computer vision*, 40(1):25–47, 2000.

[20] Nakayama K. *Mid-level vision*. MIT Press, Cambridge, 1999.

[21] Shimon Ullman and Greg J Power. High-Level Vision: Object-Recognition and Visual Cognition. *Optical Engineering*, 36(11):3224–3224, 1997.

[22] L.E. Sucar and G. Gómez. Introduction to Computer Vision. `http://ccc.inaoep.mx/~esucar/Libros/vision-sucar-gomez.pdf`.

[23] Fred D. Turek. Machine Vision Fundamentals, How to Make Robots See. *NASA Tech Briefs*, 35(6):60–62, June 2011.

[24] David A. Patterson and John L. Hennessy. *Computer organization and design - the hardware / software interface (3. ed.)*. Morgan Kaufmann, 2007.

[25] Michael Flynn. Some computer organizations and their effectiveness. *IEEETC:JOURNAL*, 21(9):948–960, 1972.

[26] R. Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16, 1990.

[27] Neil B. MacDonald. An Overview of SIMD Parallel Systems, 1992.

[28] M. Hassaballah, S. Omran, and Y. B. Mahdy. A review of SIMD multimedia extensions and their usage in scientific and engineering applications. *The Computer Journal*, 51(6):630–649, 2008.

[29] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel White paper*, 2008.

[30] Ruby B. Lee and Jerome C. Huck. 64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture. In *COMPCON*, pages 152–160, 1996.

[31] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[32] Gordon E Moore. Excerpts from a conversation with Gordon Moore: Moore's Law. `ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf`, 2005.

[33] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[34] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[35] J.M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[36] B. Doyle, B. Boyanov, S. Datta, M. Doczy, S. Hareland, B. Jin, J. Kavalieros, T. Linton, R. Rios, and R. Chau. Tri-Gate fully-depleted CMOS transistors: fabrication, design and layout. In *VLSI Technology, 2003. Digest of Technical Papers. 2003 Symposium on*, pages 133–134, 2003.

[37] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 64–75, June 2004.

[38] NVIDIA Corporation. Whitepaper: NVIDIA's Next Generation. CUDA Compute Architecture: Kepler GK110, 2012.

[39] Intel. Intel Core i7-3900 Desktop Processor Series. `http://download.intel.com/support/processors/corei7/sb/core_i7-3900_d.pdf`, 2013.

[40] [on line]. Top500 Supercomputing. `http://www.top500.org/`, December 2012.

[41] [on line]. Titan supercomputer. `http://www.olcf.ornl.gov/titan/`, November 2012.

[42] [on line]. Green500 Supercomputing. `http://www.green500.org/`, December 2012.

[43] [on line]. Mont-Blanc Project. `http://www.montblanc-project.eu/`.

[44] ARM. Cortex-A15 Processor. `http://www.arm.com/products/processors/cortex-a/cortex-a15.php`, 2013.

[45] ARM. GPU Mali-T604. `http://www.arm.com/products/multimedia/mali-graphics-plus-gpu-compute/mali-t604.php`, 2013.

[46] F. Ayuso, C. Garcia, G. Botella, M Prieto, and F Tirado. GPU-Based Signal Processing Scheme for Bioinspired Optical Flow. pages 2011–2011, FPL 2011, 09/2011 2011.

[47] F. Ayuso, G. Botella, C. Garcia, M Prieto, and F Tirado. Pre-procesamiento de Flujo óptico Robusto en Hardware gráfico. In *XXII Jornadas de Paralelismo*, pages 323–328. Universidad de La Laguna, Universidad de La Laguna, 09/2011 2011.

[48] F. Ayuso, G. Botella, C. Garcia, M. Prieto, and F. Tirado. GPU-Based Acceleration of Bioinspired Motion Estimation Model. In *WPABA 2011*, Galveston Island, Texas, USA, 10/2011 2011.

[49] F. Ayuso, G. Botella, C. Garcia, M Prieto, and F Tirado. GPU-Based Acceleration of Bioinspired Motion Estimation Model. *Concurrency and Computation: Practice and Experience*, 25(8):1037–1056, 10/2012 2012.

[50] Carlos Garcia, Guillermo Botella, Fermin Ayuso, Manuel Prieto, and Francisco Tirado. Multi-GPU Based on Multicriteria Optimization for Motion Estimation System. *EURASIP Journal on Advances in Signal Processing*, 2013(1):1–12, 02/2013 2013.

[51] Carlos García, Guillermo Botella, Fermín Ayuso, Diego González, Manuel Prieto, and Francisco Tirado. Implementation of a Low-Cost Mobile Devices to support Medical Diagnosis. *Computational and Mathematical Methods in Medicine*, In Press, 2013.

[52] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi. Video coding with H.264/AVC: tools, performance, and complexity. *Circuits and Systems Magazine, IEEE*, 4(1):7–28, 2004.

[53] H. Nyquist and C. E. Shannon. Nyquist–Shannon sampling theorem. `http://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem`, December 2012.

[54] E.H. Adelson and J.R. Bergen. Spatiotemporal Energy Models for the Perception of Motion. *Journal of the Optical Society of America A*, 2(2):284–299, 1985.

[55] H. Wallach. On Perceived Identity: 1. The Direction of Motion of Straight Lines. *In On Perception, New York: Quadrangle*, 1976.

[56] Ting-Chuen Pong and B.G. Kaiser. A hierarchical approach to the correspondence problem. *Systems, Man and Cybernetics, IEEE Transactions on*, 19(2):271–276, 1989.

[57] J.L Barron, D.J. Fleet, and S.S. Beauchemin. Performance of optical flow techniques. *INTERNATIONAL JOURNAL OF COMPUTER VISION*, 12:43–77, 1994.

[58] J. R. Jain and A. K. Jain. Displacement measurement and its application in interframe image coding. *IEEE Trans. Communications*, 29:1799–1806, 1981.

[59] Anastasis A. Sofokleous. Review: H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia. *Comput. J*, 48(5):563, 2005.

[60] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York, 1974.

[61] R. C. Gonzalez and R. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 2001.

[62] H. Gharavi and M. Mills. Blockmatching motion estimation algorithms-new results. *Circuits and Systems, IEEE Transactions on*, 37(5):649–651, 1990.

[63] T. Koga, Kazumoto Iinuma, Yukihiko Iijima, A. Hirano, and Tatsuo Ishiguro. Motion-compensated interframe coding for video conferencing. In *Proc. NTC*, pages G5.3.1–G5.3.5, 1981.

[64] Shan Zhu and Kai-Kuang Ma. A new diamond search algorithm for fast block matching motion estimation. In *Information, Communications and Signal Processing, ICICS*, volume 1, pages 292–296, 1997.

[65] M. Ghanbari. The cross-search algorithm for motion estimation [image coding]. *Communications, IEEE Transactions on*, 38(7):950–953, 1990.

[66] Lai-Man Po and Wing-Chung Ma. A novel four-step search algorithm for fast block motion estimation. *Circuits and Systems for Video Technology, IEEE Transactions on*, 6(3):313–317, 1996.

[67] M. Accame, F.G.B. De Natale, and D.D. Giusto. High Performance Hierachical Block-based Motion Estimation for Real-Time Video Coding. *Real-Time Imaging*, 4:67–79, 1998.

[68] F. Defaux and F. Moscheni. Motion Estimation Techniques for Digital TV: A Review and a New Contribution. *Proceedings of the IEEE 83*, 6:858–876, 1995.

[69] Jerome Spanier and Keith B Oldham. *An atlas of functions*. Taylor & Francis/Hemisphere, 1987.

[70] R. L. De Valois and K. K. De Valois. *Spatial Vision*. Oxford University Press, Oxford, UK, 1988.

[71] B.D. Lucas and T. Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision (IJCAI). In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 674–679, April 1981.

[72] A. Johnston and C.W.G. Clifford. Perceived Motion of Contrast modulated Gratings: Prediction of the McGM and the role of Full-Wave rectification. *Vision Research*, 35:1771–1783, 1995.

[73] A. Johnston, P.W. McOwan, and H. Buxton. A computational model of the analysis of first and second order motion by simple and complex cells. *Proc. R. Soc. London*, B 250:297–306, 1992.

[74] A. Johnston and C.W.G. Clifford. A Unified Account of Three Apparent Motion Illusions. *Vision Research*, 35(8):1109–1123, 1994.

[75] W.T. Newsome and E.B. Pare. A selective impairment of motion perception following lessons of middle temporal visual area (MT). *Journal of neuroscience*, 8:2201–2211, 1988.

[76] R.H. Hess, C.L. Baker, and J. Zihl. The motion-blind patient: low-level spatial and temporal filters. *J. Neuroscience*, 9:1628–1640, 1989.

[77] T. Pasternak and W.H. Merigan. Motion perception following lesions of the superior temporal sulcus in the monkey. *Cerebral. Cortex*, 4:247–259, 1994.

[78] Hongche Liu, Tsai-Hong Hong, Martin Herman, Ted Camus, and Rama Chellappa. Accuracy vs Efficiency Trade-offs in Optical Flow Algorithms. *Computer Vision and Image Understanding*, 72(3):271–286, 1998.

[79] Hiroaki Niitsuma and Tsutomu Maruyama. Real-time detection of moving objects. In *Field Programmable Logic and Application*, pages 1155–1157. Springer, 2004.

[80] Xilinx Inc. FPGA Virtex-II & Virtex-II Pro - Complete Data Sheet. `http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf`.

[81] Konstantinos Babionitakis, Gregory Doumenis, George Georgakarakos, George Lentaris, Kostantinos Nakos, Dionysios I. Reisis, Ioannis Sifnaios, and Nikolaos Vlassopoulos. A real-time motion estimation FPGA architecture. *J. Real-Time Image Processing*, 3(1-2):3–20, 2008.

[82] S. Asano, Zheng Zhi Shun, and T. Maruyama. An FPGA implementation of full-search variable block size motion estimation. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 399–402, December 2010.

[83] Xilinx Inc. FPGA Virtex-5 - Data Seets Documentation. `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`.

[84] A. Akin, G. Sayilar, and I. Hamzaoglu. High performance hardware architectures for one bit transform based single and multiple reference frame motion estimation. *Consumer Electronics, IEEE Transactions on*, 56(2):1144–1152, May 2010.

[85] B. Natarajan, V. Bhaskaran, and K. Konstantinides. Low-complexity block-based motion estimation via one-bit transforms. *Circuits and Systems for Video Technology, IEEE Transactions on*, 7(4):702–706, 1997.

[86] Huong Ho, R. Klepko, Nam Ninh, and Demin Wang. A high performance hardware architecture for multi-frame hierarchical motion estimation. *Consumer Electronics, IEEE Transactions on*, 57(2):794–801, 2011.

[87] Demin Wang, Liang Zhang, and André Vincent. Motion-Compensated Frame Rate Up-Conversion - Part I: Fast Multi-Frame Motion Estimation. *TBC*, 56(2):133–141, 2010.

[88] José L. Núñez-Yañez, A. Nabina, E. Hung, and G. Vafiadis. Cogeneration of Fast Motion Estimation Processors and Algorithms for Advanced Video Coding. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(3):437–448, 2012.

[89] Yu-Wen Huang, Ching-Yeh Chen, Chen-Han Tsai, Chun-Fu Shen, and Liang-Gee Chen. Survey on block matching motion estimation algorithms and architectures with new results. *Journal of VLSI signal processing systems for signal, image and video technology*, 42(3):297–320, 2006.

[90] Zhaoyi Wei, Dah-Jye Lee, B. Nelson, M. Martineau, Zhaoyi Wei, Dah-Jye Lee, and M. Martineau. A Fast and Accurate Tensor-based Optical Flow Algorithm Implemented in FPGA. In *Applications of Computer Vision, 2007. WACV '07. IEEE Workshop on*, pages 18–18, 2007.

[91] Björn Johansson and Gunnar Farnebäck. A theoretical comparison of different orientation tensors. In *Proceedings SSAB02 Symposium on Image Analysis*, pages 69–73. Citeseer, 2002.

[92] G. Farnebäck. Fast and accurate motion estimation using orientation tensors and parametric motion models. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, volume 1, pages 135–139, 2000.

[93] Javier Díaz, Eduardo Ros, Rodrigo Agís, and Jose Luis Bernier. Superpipelined high-performance optical-flow computation architecture. *Computer Vision and Image Understanding*, 112(3):262–273, 2008.

[94] G. Botella, A. García, M. Rodriguez, E. Ros, U. Meyer-Bäse, and M.C. Molina. Robust Bioinspired Architecture for Optical-Flow Computation. *IEEE Trans. VLSI Syst.*, 18(4):616–629, 2010.

[95] Xilinx Inc. FPGA Virtex-E - Field Programmable Gate Arrays. `http://www.xilinx.com/support/documentation/data_sheets/ds022.pdf`.

[96] M.R.B. Bahar and G. Karimian. High performance implementation of the Horn and Schunck optical flow algorithm on FPGA. In *Electrical Engineering (ICEE), 2012 20th Iranian Conference on*, pages 736–741, May 2012.

[97] [on-line] Altera Corporation. Cyclone II. `http://www.altera.com/devices/fpga/cyclone2/cy2-index.jsp`.

[98] F. Barranco, M. Tomasi, J. Diaz, M. Vanegas, and E. Ros. Parallel Architecture for Hierarchical Optical Flow Estimation Based on FPGA. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(6):1058–1067, 2012.

[99] Xilinx Inc. FPGA Virtex-4 - Family Overview. `http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf`.

[100] Ehsan Norouznezhad, Abbas Bigdeli, Adam Postula, and Brian C. Lovell. Object tracking on FPGA-based smart cameras using local oriented energy and phase features. In *Proceedings of the Fourth ACM/IEEE International Conference on Distributed Smart Cameras*, ICDSC '10, pages 33–40, New York, NY, USA, 2010. ACM.

[101] Dennis Gabor. Theory of communication. Part 1: The analysis of information. *Electrical Engineers-Part III: Radio and Communication Engineering, Journal of the Institution of*, 93(26):429–441, 1946.

[102] M. Tomasi, M. Vanegas, F. Barranco, J. Diaz, and E. Ros. High-Performance Optical-Flow Architecture Based on a Multi-Scale, Multi-Orientation Phase-Based Model. *Circuits and Systems for Video Technology, IEEE Transactions on*, 20(12):1797–1807, 2010.

[103] T. Gautama and M.M. Van Hulle. A phase-based approach to the estimation of the optical flow field using spatial filtering. *Neural Networks, IEEE Transactions on*, 13(5):1127–1136, 2002.

[104] M. Tomasi, M. Vanegas, F. Barranco, J. Daz, and E. Ros. Massive Parallel-Hardware Architecture for Multiscale Stereo, Optical Flow and Image-Structure Computation. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(2):282–294, 2012.

[105] Silvio P. Sabatini, Giulia Gastaldi, Fabio Solari, Karl Pauwels, Marc M. Van Hulle, Javier Diaz, Eduardo Ros, Nicolas Pugeault, and Norbert Krüger. A compact harmonic code for early vision based on anisotropic frequency channels. *Computer Vision and Image Understanding*, 114(6):681–699, 2010. ¡ce:title¿Special Issue on Multi-Camera and Multi-Modal Sensor Fusion¡/ce:title¿.

[106] Javier Díaz, Eduardo Ros, Rodrigo Agís, and Jose Luis Bernier. Superpipelined high-performance optical-flow computation architecture. *Computer Vision and Image Understanding*, 112(3):262–273, 2008.

[107] Jean-Yves Bouguet. Pyramidal Implementation of the Affine Lucas Kanade Feature Tracker - Description of the algorithm. *Intel Corporation*, 2001.

[108] Simon Baker and Iain Matthews. Equivalence and efficiency of image alignment algorithms. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–1090. IEEE, 2001.

[109] Jinglin Zhang, J.-F. Nezan, and J.-G. Cousin. Implementation of Motion Estimation Based on Heterogeneous Parallel Computing System with OpenCL. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 41 –45, June 2012.

[110] Bernardt Duvenhage, J. P. Delport, and Jason de Villiers. Implementation of the Lucas-Kanade image registration algorithm on a GPU for 3d computational platform stabilisation. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '10, pages 83–90, New York, NY, USA, 2010. ACM.

[111] K. Pauwels and M.M. Van Hulle. Realtime phase-based optical flow on the GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, june 2008.

[112] G. Kiss, E. Nielsen, F. Orderud, and H.G. Torp. Performance optimization of block matching in 3D echocardiography. In *Ultrasonics Symposium (IUS), 2009 IEEE International*, pages 1403–1406, 2009.

[113] Jonas Crosby, Brage H. Amundsen, Torbjørn Hergum, Espen W. Remme, Stian Langeland, and Hans Torp. 3-D Speckle Tracking for Assessment of Regional Left Ventricular Function. *Ultrasound in Medicine & Biology*, 35(3):458–471, 2009.

[114] NVIDIA Geforce GTX 285. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-285`.

[115] Guangyong Zhang, Liqiang He, and Yanyan Zhang. Parallel Best Neighborhood Matching Algorithm Implementation on GPU Platform. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1140–1145, 2010.

[116] Zhou Wang, Yinglin Yu, and D. Zhang. Best neighborhood matching: an information loss restoration technique for block-based image coding systems. *Image Processing, IEEE Transactions on*, 7(7):1056–1061, 1998.

[117] NVIDIA Tesla Supercomputing Solutions. `http://www.nvidia.com/object/tesla-supercomputing-solutions.html`.

[118] E. Monteiro, B. Vizzotto, C. Diniz, B. Zatt, and S. Bampi. Applying CUDA Architecture to Accelerate Full Search Block Matching Algorithm for High Performance Motion Estimation in Video Encoding. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pages 128–135, 2011.

[119] NVIDIA Geforce GTX 480. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications`.

[120] B. Ranft, T. Schoenwald, and B. Kitt. Parallel matching-based estimation - a case study on three different hardware architectures. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 1060–1067, 2011.

[121] J.-B. Note, M. Shand, and J.E. Vuillemin. Real-Time Video Pixel Matching. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, 2006.

[122] NVIDIA Geforce GTX 470. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-470`.

[123] AMD Radeon HD 6870. `http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/hd-6870/Pages/amd-radeon-hd-6870-overview.aspx`.

[124] R. Rodríguez Sánchez, J.L. Martínez, G. Fernandez Escribano, J.L. Sanchez, and J.M. Claver. A Fast GPU-Based Motion Estimation Algorithm for HD 3D Video Coding.

In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 166–173, 2012.

[125] E. Monteiro, M. Maule, F. Sampaio, C. Diniz, B. Zatt, and S. Bampi. Real-time block matching motion estimation onto GPGPU. In *Image Processing (ICIP), 2012 19th IEEE International Conference on*, pages 1693–1696, 2012.

[126] Dung Vu, Yang Yang, and L. Bhuyan. An efficient dynamic multiple-candidate motion vector approach for GPU-based hierarchical motion estimation. In *Performance Computing and Communications Conference (IPCCC), 2012 IEEE 31st International*, pages 342–351, 2012.

[127] Y.L. Chan and W.C. Siu. Adaptive multiple-candidate hierarchical search for block matching algorithm. *Electronics Letters*, 31(19):1637–, 1995.

[128] Especificaciones técnicas de la GPU NVIDIA Tesla C250. `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`.

[129] J. Chase, B. Nelson, J. Bodily, Zhaoyi Wei, and Dah-Jye Lee. Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 173–182, 2008.

[130] NVIDIA GeForce 8800 gtx. `http://www.nvidia.com/page/geforce_8800.html`.

[131] Julien Marzat, Yann Dumortier, André Ducrot, et al. Real-time dense and accurate parallel optical flow using CUDA. In *7th International Conference WSCG*, 2009.

[132] R. Phull, P. Mainali, Qiong Yang, H. Sips, and G. Lafruit. Robust Low Complexity Feature Tracking using CUDA. In *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, pages 362–367, 2010.

[133] P. Mainali, Qiong Yang, G. Lafruit, R. Lauwereins, and L. Van Gool. Robust low complexity feature tracking. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 829–832, 2010.

[134] NVIDIA GeForce 280 gtx. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280`.

[135] Rafael del Riego, José Otero, and José Ranilla. A low-cost 3D human interface device using GPU-based optical flow algorithms. *Integrated Computer-Aided Engineering*, 18:391–400, 2011.

[136] NVIDIA GeForce 9500 GT. `http://www.geforce.com/hardware/desktop-gpus/geforce-9500-gt`.

[137] Robert Hegner, Ivar Austvoll, Tom Ryen, and Guido M Schuster. Efficient Implementation of Optical Flow Algorithm based on Directional Filters on a GPU Using CUDA. EUSIPCO, 2011.

[138] Robert Hegner and Guido Schuster. *Efficient implementation and evaluation of methods for the estimation of motion in image sequences*. PhD thesis, R. Hegner, 2010.

[139] NVIDIA GeForce GTX 260. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-260`.

[140] J. Ohmura, A. Egashira, S. Satoh, T. Miyoshi, H. Irie, and T. Yoshinaga. Multi-GPU Acceleration of Optical Flow Computation in Visual Functional Simulation. In *Networking and Computing (ICNC), 2011 Second International Conference on*, pages 228–234, 30 2011-dec. 2 2011.

[141] Manish P. Shiralkar and Robert J. Schalkoff. A self-organization based optical flow estimator with GPU implementation. *Mach. Vis. Appl*, 23(6):1229–1242, 2012.

[142] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[143] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense Point Trajectories by GPU-Accelerated Large Displacement Optical Flow. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, volume 6311 of *Lecture Notes in Computer Science*, pages 438–451. Springer Berlin Heidelberg, 2010.

[144] T. Brox, C. Bregler, and J. Malik. Large displacement optical flow. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 41–48, 2009.

[145] Pascal Gwosdek, Henning Zimmer, Sven Grewenig, Andrés Bruhn, and Joachim Weickert. A Highly Efficient GPU Implementation for Variational Optic Flow Based on the Euler-Lagrange Framework. In KiriakosN. Kutulakos, editor, *Trends and Topics in Computer Vision*, volume 6554 of *Lecture Notes in Computer Science*, pages 372–383. Springer Berlin Heidelberg, 2012.

[146] A. Abramov, K. Pauwels, J. Papon, F. Worgotter, and B. Dellen. Real-Time Segmentation of Stereo Videos on a Portable System With a Mobile GPU. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(9):1292–1305, 2012.

[147] Karl Pauwels, Norbert Krüger, Markus Lappe, Florentin Wörgötter, and Marc M Van Hulle. A cortical architecture on parallel hardware for motion processing in real time. *Journal of vision*, 10(10), 2010.

[148] NVIDIA GeForce GT 240M - Specifications. `http://www.geforce.com/hardware/notebook-gpus/geforce-gt-240m/specifications`.

[149] Multiview Coding Using AVC. ISO/IEC/JTC1/SC29/WG11, January 2006.

[150] Carlo Tomasi and Takeo Kanade. *Detection and tracking of point features*. School of Computer Science, Carnegie Mellon Univ., 1991.

[151] Hans-Hellmut Nagel and Wilfried Enkelmann. An Investigation of Smoothness Constraints for the Estimation of Displacement Vector Fields from Image Sequences. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(5):565–593, 1986.

[152] Henning Zimmer, Andrés Bruhn, Joachim Weickert, Levi Valgaerts, Agustín Salgado, Bodo Rosenhahn, and Hans-Peter Seidel. Complementary Optic Flow. In Daniel Cremers, Yuri Boykov, Andrew Blake, and FrankR. Schmidt, editors, *Energy Minimization Methods in Computer Vision and Pattern Recognition*, volume 5681 of *Lecture Notes in Computer Science*, pages 207–220. Springer Berlin Heidelberg, 2009.

[153] S. Warrington, S. Sudharsanan, and Wai-Yip Chan. Architecture for Multiple Reference Frame Variable Block Size Motion Estimation. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 2894–2897, May 2007.

[154] R. Verma and A. Akoglu. A coarse grained and hybrid reconfigurable architecture with flexible NoC router for variable block size motion estimation. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[155] N. Sebastião, T. Dias, N. Roma, P. Flores, and L. Sousa. Application Specific Programmable IP Core for Motion Estimation: Technology Comparison Targeting Efficient Embedded Co-Processing Units. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 181–188, September 2008.

[156] O. Ndili and T. Ogunfunmi. Efficient fast algorithm and FPSoC for integer and fractional motion estimation in H.264/AVC. In *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, pages 407–408, January 2011.

[157] O. Ndili and T. Ogunfunmi. Hardware-oriented Modified Diamond Search for motion estimation in H.246/AVC. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pages 749–752, 2010.

[158] S. Dhahri, A. Zitouni, and R. Tourki. A parallel processing architecture for FSS block-matching motion estimation. In *Systems, Signals and Devices (SSD), 2011 8th International Multi-Conference on*, pages 1–5, March 2011.

[159] Alan A. Stocker. Analog Integrated 2-D Optical Flow Sensor. *Analog Integrated Circuits and Signal Processing*, 46(2):121–138, 2006.

[160] Th Zahariadis and D Kalivas. A spiral search algorithm for fast estimation of block motion vectors. *Signal Processing VIII, theories and applications. Proceedings of the EUSIPCO*, 96:3, 1996.

[161] Ellen C. Hildreth. The Computation of the Velocity Field. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 221(1223):189–220, 1984.

[162] Ellen C. Hildreth. Computations underlying the measurement of visual motion. *Artificial Intelligence*, 23(3):309–354, 1984.

[163] David J. Fleet. *Measurement of Image Velocity*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[164] Ben Galvin, Brendan McCane, Kevin Novins, David Mason, and Steven Mills. Recovering motion fields: An evaluation of eight optical flow algorithms. In *British machine vision conference*, volume 1, pages 195–204. sn, 1998.

[165] M. Otte and H. H. Nagel. Estimation of Optical-Flow Based on Higher-Order Spatiotemporal Derivatives in Interlaced and Noninterlaced Image Sequences. *Artificial Intelligence*, 78(1-2):5–43, October 1995.

[166] B. McCane, K. Novins, D. Crannitch, and B. Galvin. On benchmarking optical flow. *Computer Vision and Image Understanding*, 84(1):126–143, October 2001.

[167] Patrick Cavanagh and George Mather. Motion: The long and short of it. *Spatial vision*, 4(2-3):2–3, 1989.

[168] Charles Chubb, George Sperling, et al. Drift-balanced random stimuli- A general basis for studying non-fourier motion perception. *Optical Society of America, Journal, A: Optics and Image Science*, 5:1986–2007, 1988.

[169] A. Johnston, P. W. McOwan, and C. Benton. Robust Velocity Computation from a Biologically Motivated Model of Motion Perception. *Proceedings of the Royal Society of London*, B 266:509–518, 1999.

[170] R.F. Hess and R.J. Snowden. Temporal frequency filters in the human peripheral visual field. *Vision Research*, 32:61–72, 1992.

[171] J.J. Koenderink and A.J. Van Doorn. Representation of Local Geometry in the Visual System. *Biological Cybernetics*, 63:291–297, 1987.

[172] G. Botella. *Implementación en hardware reconfigurable de un modelo de flujo óptico robusto*. PhD thesis, Universidad de Granada, 2007.

[173] Peter W. McOwan, Christopher Benton, Jason Dale, and Alan Johnston. A Multi-Differential Nneuromorphic Approach to Motion Detection. *International Journal of Neural Systems*, 09(05):429–434, 1999.

[174] K. Anderson and P. W. McOwan. Real-Time Automated System for the Recognition of Human Facial Expressions. *IEEE Trans. Systems, Man and Cybernetics*, 36(1):96–105, February 2006.

[175] Xuefeng Liang, Peter W. McOwan, and Alan Johnston. Biologically inspired framework for spatial and spectral velocity estimations. *J. Opt. Soc. Am. A*, 28(4):713–723, Apr 2011.

[176] [on line]. http://www.khronos.org/opencl/.

[177] NVIDIA Corporation. CUDA C Programming Guide. (PG-02829-001_v5.0). Available from: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, December 2012.

[178] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[179] NVIDIA Corporation. CUDA: Compute Unified Device Architecture. http://developer.nvidia.com/object/cuda.html, December 2012.

[180] NVIDIA Corporation. CUDA C Best Practices Guide. (dg-05603-001_v5.0). Available from: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, December 2012.

[181] NVIDIA Corporation. Whitepaper: NVIDIA's Next Generation. CUDA Compute Architecture: Fermi, 2009.

[182] KOW Group et al. OpenCL 1.2 specifications, 2012.

[183] OpenCL Programmability on 4th Generation Intel Core Processors. http://software.intel.com/sites/default/files/article/182971/hsw-graphics-productbrief.pdf, 2013.

[184] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, December 2010.

[185] Jianbin Fang, Ana Lucia Varbanescu, and Henk J. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In Guang R. Gao and Yu-Chee Tseng, editors, *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan*, pages 216–225. IEEE, September 2011.

[186] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[187] Michael Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.

[188] Tianyi David Han and Tarek S Abdelrahman. hicuda: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[189] T.D. Hart and T.S. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 2010.

[190] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to GPGPU: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009.

[191] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[192] Ruymán Reyes, Iván López-Rodríguez, Juan J. Fumero, and Francisco de Sande. accULL: An OpenACC implementation with CUDA and OpenCL support. In *Euro-Par 2012 Parallel Processing*, pages 871–882. Springer, 2012.

[193] The OpenACC Aplication Programming Interface. Version 2.0. [on line] `http://openacc.org/sites/default/files/OpenACC-2.0-draft.pdf`, March 2013.

[194] Michaela Barth, KTH Sweden, Mikko Byckling, CSC Finland, Nevena Ilieva, NCSA Bulgaria, Sami Saarinen, Michael Schliephake, Volker Weinberg, and LRZ Germany. Best Practice Guide Intel Xeon Phi v0.1. March 2013.

[195] NVIDIA Depeloper Zone. CUDA C/C++ SDK CODE Samples. `http://developer.nvidia.com/cuda-cc-sdk-code-samples`, January 2013.

[196] William T. Freeman and Edward H. Adelson. The Design and Use of Steerable Filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13:891–906, 1991.

[197] Z.-E. Baarir and F. Charif. Fast modified Horn & Schunck method for the estimation of optical flow fields. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 283–288, October 2011.

[198] S. Uras, F. Girosi, A. Verri, and V. Torre. A computational approach to motion perception. *Biological Cybernetics*, 60:79–87, 1988. 10.1007/BF00202895.

[199] M. Shaaban, S. Goel, and M. Bayoumi. Motion estimation algorithm for real-time systems. *Signal Processing Systems (SiPS), 2004 IEEE Workshop*, pages 257–262, 2004.

[200] Jung-Yup Kang, Sandeep Gupta, Saurabh Shah, and Jean-Luc Gaudiot. An Efficient PIM (Processor-In-Memory) Architecture for Motion Estimation. In *ASAP*, pages 282–292. IEEE Computer Society, 2003.

[201] Jung-Yup Kang, Sandeep K. Gupta, and Jean-Luc Gaudiot. An Efficient Data-Distribution Mechanism in a Processor-In-Memory (PIM) Architecture Applied to Motion Estimation. *IEEE Trans. Computers*, 57(3):375–388, 2008.

[202] L. Mattes and S. Kofuji. Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids. In *2010 International Conference on Microwave and Millimeter Wave Technology (ICMMT)*, pages 1536–1539, 2010.

[203] Yuan Zhou and Michael Garland. Interactive point-based rendering of higher-order tetrahedral data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):2006, 2006.

[204] Ralph E. Steuer. *Multiple Criteria Optimization: Theory, Computation, and Application*. Krieger Malabar, 1989.

[205] Yoshikazu Sawaragi, Hirotaka Nakayama, and Tetsuzo Tanino. *Theory of Multiobjective Optimization*, volume 176. Academic Press New York, 1985.

[206] VV Podinovskii and VD Nogin. Pareto-optimal solutions of multicriteria problems. *Moscow: Sci*, 1982.

[207] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[208] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706, October 1988.

[209] A. Konak, D. W. Coit, and A. E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, sep 2006.

[210] Carlos A. Coello Coello. 20 Years of Evolutionary Multi-Objective Optimization: What Has Been Done and What Remains to be Done. In Gary Y. Yen and David B. Fogel, editors, *Computational Intelligence: Principles and Practice*, chapter 4, pages 73–88. IEEE Computational Intelligence Society, Vancouver, Canada, 2006, ISBN 0-9787135-0-8.

[211] Carlos M. Fonseca and Peter J. Fleming. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 416–423, San Mateo, California, 1993. University of Illinois at Urbana-Champaign, Morgan Kauffman Publishers.

[212] N. Srinivas and Kalyanmoy Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. Technical report, Department of Mechanical Engineering, Indian Institute of Technology, Kanpur, India, 1993.

[213] Eckart Zitzler, Marco Laumanns, and Stefan Bleuler. A Tutorial on Evolutionary Multiobjective Optimization. In Xavier Gandibleux, Marc Sevaux, Kenneth Sörensen, and Vincent T'kindt, editors, *Metaheuristics for Multiobjective Optimisation*, pages 3–37, Berlin, 2004. Springer. Lecture Notes in Economics and Mathematical Systems Vol. 535.

[214] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evol. Comput.*, 8(2):173–195, June 2000.

[215] Carlos M. Fonseca and Peter J. Fleming. Multiobjective optimization and multiple constraint handling with evolutionary algorithms-part i: A unified formulation. *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans*, 28:26–37, 1998.

[216] Kalyanmoy Deb, Samir Agrawal, Amrit Pratab, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In Marc Schoenauer, Kalyanmoy Deb, Gunter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 849–858, Paris, France, 2000. Springer. Lecture Notes in Computer Science No. 1917.

[217] Berthold K. Horn. *Robot Vision*. McGraw-Hill Higher Education, 1st edition, 1986.

[218] David Fleet and Yair Weiss. Optical flow estimation. In *IEEE Transactions on Image Processing*, volume 19, pages 1–10. Springer, 2005.

[219] Yu M Chi, Trac D Tran, and Ralph Etienne-Cummings. Optical flow approximation of sub-pixel accurate block matching for video coding. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 1, pages I–1017. IEEE, 2007.

[220] G. Indiveri and R.J. Douglas. ROBOTIC VISION: Neuromorphic Vision Sensor. *Science*, 288:1189–1190, May 2000.

[221] C. M. Higgins. Sensory architectures for biologically inspired autonomous robotics. *The Biological Bulletin*, 200(2):235–242.

[222] Hongche Liu, Tsai-Hong Hong, Martin Herman, and Rama Chellappa. A general motion model and spatio-temporal filters for computing optical flow. *International Journal of Computer Vision*, 22(2):141–172, 1997.

[223] Reid R. Harrison and Christof Koch. An Analog VLSI Model of the Fly Elementary Motion Detector. In *in Advances in Neural Information Processing Systems 10*, pages 880–886. MIT Press.

[224] C. Mead. Neuromorphic Electronic Systems. *Proceedings of the IEEE*, 78(10):1629–36, 1990.

[225] Thomas S. Collett. Insect Vision: Controlling Actions through Optic Flow. *Current Biology*, (18):R615–R617, September.

[226] K. Weber, S. Venkatesh, and M.V. Srinivasan. Insect Inspired Behaviours for the Autonomous Control of Mobile Robots. pages 226–248. Oxford University Press, 1997.

[227] J.J. Koenderink. Optic Flow. *Vision Research*, 26(1):161–180, 1986.