Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías Industriales.

Cálculo paralelo con CUDA y Matlab en el seguimiento visual de objetos

Autor: Cristian Gil Chicano

Tutor: Manuel Ruiz Arahal

Depto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Cálculo paralelo con CUDA y Matlab en el seguimiento visual de objetos

Autor:
Cristian Gil Chicano

Tutor:
Manuel Ruiz Arahal
Profesor titular

Depto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2015

Trabajo Fin de Grado: Cálculo paralelo con CUDA y Matlab en el seguimiento visual de objetos

Autor: Cristian Gil Chicano

Tutor: Manuel Ruiz Arahal

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2015

El Secretario del Tribunal

A mi familia

A mis maestros

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

1.1. Motivación

Nuestra sociedad está experimentando un cambio radical gracias a la incorporación de la tecnología a nuestra vida diaria. Dicha evolución permite que cada vez sea más común escuchar términos como robots de asistencia para enfermos, domótica e inmótica, UAVs y sistemas de seguridad automáticos entre otros.

Una de las cualidades por las cuales estos sistemas son interesantes es su capacidad de interactuar con los usuarios y con el entorno que los rodea.

La robótica soluciona el problema de la capacidad sensorial utilizando distintos medios, sensores de ultrasonidos, sistemas de escáner por láser y sistemas de visión artificial mediante cámaras.

Los sistemas de percepción artificial permiten discretizar la realidad y convertirla en datos que pueden ser tratados de manera informática. En concreto nos centraremos en los sistemas de visión artificial, los cuales permiten obtener información fiable de manera económica aunque la complejidad y coste computacional de los algoritmos de visión jueguen en su contra.

Por otro lado se están popularizando la utilización de dispositivos y estructuras de programación que permiten la optimización y aceleración de algoritmos y reducción en los tiempos de computación.

En concreto nos centraremos en el estudio de las unidades de procesamiento gráfico, las cuales incorporan la tecnología necesaria para realizar operaciones en paralelo, aplicándolas sobre una técnica utilizada en el seguimiento visual de objetos, el flujo óptico.

1.2. Evolución de la GPU.

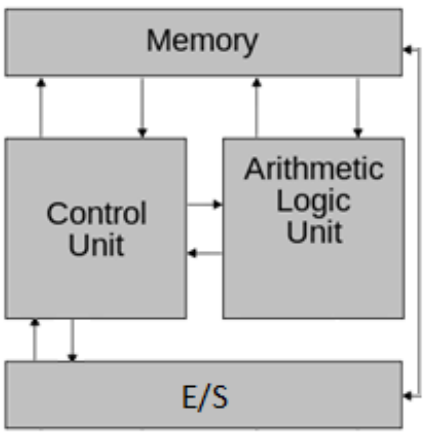


Figura 1.1 Arquitectura von Neumann

Dentro de la actual estructura de un ordenador es imposible no mencionar uno de sus elementos indispensables, el procesador gráfico o GPU. Por otro lado este hecho no fue siempre así.

En los inicios de la informática doméstica, al igual que hoy en día, las bases en las que se sustentaban los ordenadores era la arquitectura de von Neumann.

Como se aprecia en la figura 1.1, la propuesta que desarrolló John von Neumann constaba de 4 pilares indispensables, memoria, ALU, unidad de control y dispositivos de E/S. Esta estructura cumple las funciones de almacenamiento procesado e intercambio de información.

En los primeros ordenadores, la CPU, unidad central de procesamiento, se encargaba de procesar todo tipo de información. Conforme aumentaba la popularidad de los ordenadores y surgían nuevas aplicaciones que necesitaban más recursos, programas de edición gráfica o videojuegos, se hizo necesario el desarrollo de nuevos dispositivos que cumpliesen las nuevas exigencias de los usuarios.

El nacimiento de la GPU como la conocemos actualmente deriva de un componente llamado coprocesador matemático. Este dispositivo se utilizaba para acelerar ciertas operaciones y procesamiento de datos, es decir, como segundo procesador. No fue hasta 1999 cuando NVidia acuñó el término GPU, desde ese instante la mejora de rendimiento se ha ido incrementando gracias, en gran medida, al éxito en el campo recreativo de los videojuegos.

Por otro lado, el objetivo final del diseño de una GPU era resolver los problemas asociados a los gráficos. En el 2007 NVIDIA vuelve a reavivar el concepto de coprocesador matemático fusionándolo con el concepto de GPU. Llegado ese momento nacen las GPUPU, dispositivos capaces de ejecutar código que normalmente está destinado a ser ejecutado en la CPU.

A modo de ejemplo, la tarjeta gráfica de NVIDIA GeForce GTX Titán Z contiene 5760 núcleos capaces de trabajar de manera simultánea.

Por otro lado, existen otros sistemas que permiten la aumentar el rendimiento y la velocidad del software como pueden ser:

-Los procesadores multinúcleo: Dispositivos que combina en un solo circuito varios microprocesadores independientes. Permiten realizar operaciones de manera paralela pero no pueden compararse con la capacidad de una GPU.

-Clúster: Conjunto de múltiples ordenadores conectado mediante una red de alta velocidad de tal manera que la cooperación entre ellos los convierte en una sola unidad, comportándose como un ordenador más potente.

1.3. Seguimiento de objetos:

La finalidad del seguimiento es la estimación de la posición de los objetos en imágenes mediante el uso de cámaras de video. Es un proceso complejo que engloba una multitud de técnicas de visión de alto coste computacional. Las tareas necesarias para realizar el seguimiento son distintas según el campo de aplicación en el que deba desenvolverse el seguidor de objetos pero, a grandes rasgos, podemos diferenciar dos:

1. El reconocimiento de objetos: Para poder localizar correctamente un objeto es indispensable ser capaz de identificarlo dentro de la imagen. No es un proceso que pertenezca al problema de seguimiento de objetos pero es necesario para su funcionamiento.
2. Selección de características: Una vez determinado el objeto será necesario prestar atención a una serie de características visuales que nos permitan realizar el seguimiento. Estas características suelen ser el color, la textura o el flujo óptico. En este trabajo nos centraremos en el estudio del flujo óptico.

Figura 1.2: Seguimiento de objetos

1.1.1 El flujo óptico.

Se define el flujo óptico como el patrón de movimiento aparente entre los objetos, superficies y bordes causado por el movimiento relativo entre el observador y la escena. Este concepto es estudiado por primera vez por el psicólogo James Jerome Gibson a partir de unas grabaciones realizadas al situar una cámara en un avión. De estas grabaciones se desprendía que a partir del estudio del flujo óptico se podía contabilizar el movimiento o reposo de los objetos según la presencia o ausencia del flujo así como el cambio en la orientación del observador con respecto a la escena. En la figura 1.3 se representan dos ejemplos característicos. En el primero apreciamos el campo de desplazamientos de un observador alejándose del horizonte sobre una superficie plana. En el segundo se muestra los resultados obtenidos de una cámara de video colocada en un avión real realizando una maniobra de rotación.

En la actualidad el cálculo del flujo óptico se utiliza en el problema de seguimiento de objetos, como ya hemos comentado, para el cálculo de tiempo hasta colisión, detección de movimiento, estabilización de video, extracción de fondo, odometría visual y compresión de video entre otras.

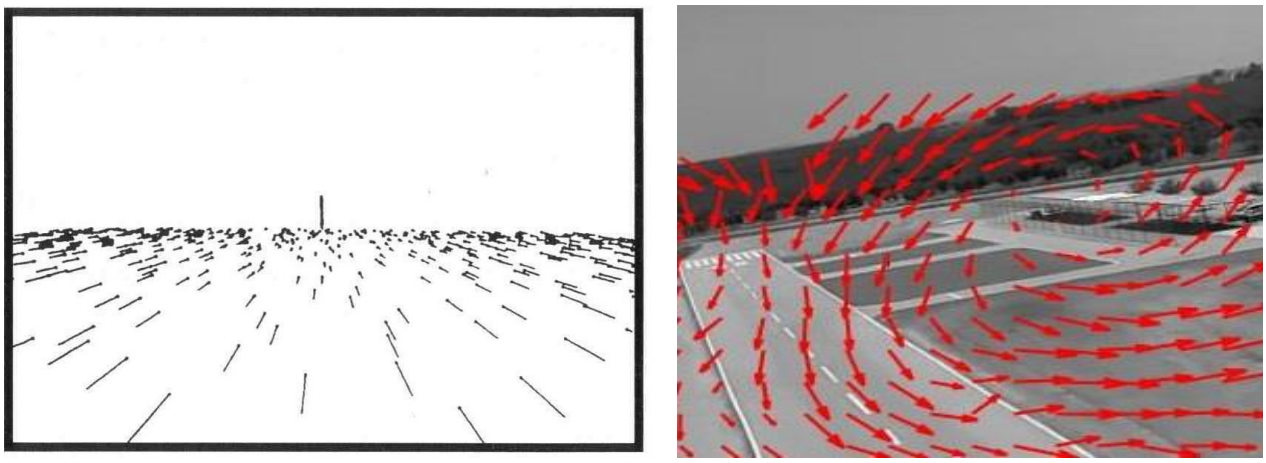


Figura 1.3 Ejemplos de Flujo óptico

La obtención del flujo óptico se realiza a partir de una secuencia de imágenes ordenadas en la que se pueda apreciar el movimiento de los objetos. Partiendo de unas hipótesis impuestas sobre la intensidad de la imagen se desarrolla una formulación matemática que permite calcular el flujo óptico. En nuestro caso utilizaremos el método de Lucas-Kanade con el que se puede calcular un flujo estimado mediante la obtención de las derivadas espaciotemporales de la imagen de intensidad.

1.4. Objetivos

Partiendo de algoritmos para el cálculo del flujo óptico y seguimiento de objetos, se van a realizar implementaciones tanto secuenciales como basadas en la programación paralela.

Los objetivos finales son, realizar un estudio comparativo del tiempo de computación en ambos dispositivos, comprender las diferencias que existen a la hora de implementar las funciones y poner de manifiesto la versatilidad que ofrece el entorno de programación Matlab para gestionar este tipo de programación así como las limitaciones que pueda contener.

2 ARQUITECTURA GPU

En este capítulo procederemos a describir los aspectos generales relacionados con el modelo de programación CUDA desarrollado por NVIDIA.

2.1. Arquitectura CUDA

Ya se ha comentado brevemente la estructura tradicional de von Neumann pero, antes de entrar de lleno en el concepto de GPU vamos a explicar brevemente las características y componentes de un ordenador actual, desde el punto de vista que nos interesa.

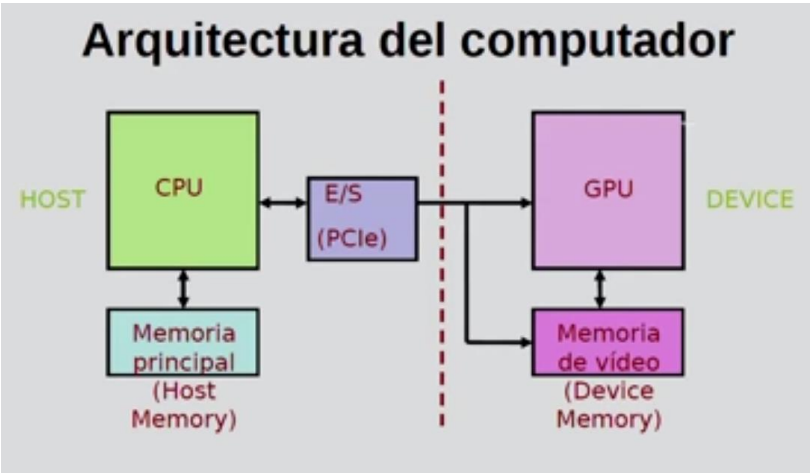


Figura 2.1: Estructura de una computadora convencional.

Como se observa en la figura 2.1 existen principalmente dos partes, una que denominaremos *host* y otra llamada *device*, las cuales podemos asociar al concepto de maestro y esclavo.

En el *host* se encuentra el procesador del ordenador o CPU junto con la memoria principal así como los dispositivos de E/S que permiten la comunicación con el *Device*. En el *Device* se encuentra la tarjeta gráfica junto con una memoria de video asociada.

De manera tradicional se ha estado ejecutando algoritmos sobre el Host utilizando la CPU siguiendo el modelo de ejecución secuencial. Las consecuencias de este sistema a la hora de manejar una cantidad de información elevada es la consecuente ralentización y aumento del tiempo necesario para obtener resultados.

En este paradigma la GPU (Unidad de procesamiento gráfico), dividida en distintas unidades funcionales, estaba enfocada a realizar dos tareas, procesar píxeles y procesar vértices. La GPU por consiguiente resolvía problemas gráficos.

La tecnología CUDA (Arquitectura Unificada de Dispositivos de Cómputo) de NVIDIA mejora dichas funcionalidades incorporando a cada unidad una unidad aritmética lógica (ALU) y una memoria para almacenar el cache necesario para la ejecución de los programas, desarrollando las llamadas GPUPU o GPU de propósito general.

Por consiguiente en la CPU (maestro) se ejecutará código que no necesita ser paralelizado mientras que en la GPU (esclavo) se ejecutará código en múltiples unidades de procesamiento permitiendo velocidades de cómputo muy superiores.

2.2. Jerarquía entre malla, bloques e hilos

Dentro de la GPU podemos encontrar múltiples núcleos de procesamiento capaces de ejecutar programas de manera paralela. Dichos elementos se organizan siguiendo una jerarquización que facilita la programación y el correcto reparto de recursos. El *Device* está dividido en *Grids o* mallas como elemento que contiene a todos los demás, cada malla dispone de distintos bloques y cada bloque está formado por un número de hilos determinado que ejecutan una función denominada Kernel.

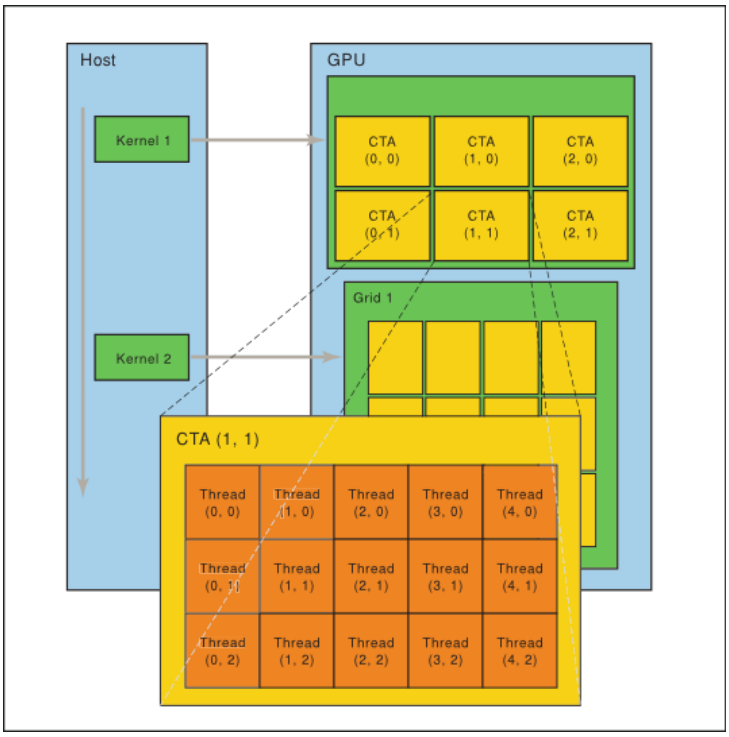


Figura 2.2: Modelo de ejecución de un kernel.

La figura 2.2 muestra el esquema general del modelo de programación CUDA. A primera vista podemos observar los dos elementos principales, host y GPU, representados por dos rectángulos azules. El Host es el encargado de configurar y ordenar la ejecución de los Kernels, los cuales se ejecutarán en dos mallas diferentes. Si prestamos atención al Grid en el cual se ejecuta el Kernel 1 podemos ver que está formado por una matriz de bloques constituida por dos filas y tres columnas. Atendiendo al bloque correspondiente al elemento (1,1) comprobamos que contiene una estructura de hilos, de tres filas y cuatro columnas. En este caso cada hilo o *thread* será el encargado de ejecutar el *Kernel 1*.

2.1.1 Hilos

La función o algoritmo que se desea ejecutar de manera masiva en el d*evice* se denomina *kernel*. Esta función se invoca desde la CPU la cual delega el trabajo en la GPU, creando múltiples hilos encargados de ejecutar el Kernel de manera individual e independiente. Los datos sobre los cuales trabajan cada hilo pueden ser distintos, a este tipo de ejecución se le denomina SPMD o un programa, multiples datos.

Por otro lado cada hilo consta de información que le permite identificarse de manera unívoca a la vez que permite tomar decisiones de control y direccionar la memoria. Dicha información se almacena en un vector de hasta 3 componentes, es decir, se pueden organizar los hilos como un vector, matriz bidimensional o tridimensional de según la dimensión que deseemos y la aplicación a la que esté destinada. Por ejemplo, si se quiere operar sobre cada elemento de un vector tendrá sentido iniciar los hilos como vector, en el caso de tratar una imagen, debido al carácter bidimensional iniciaremos los hilos como una matriz.

Hilo (0)
Hilo (2)
Hilo (3)
Hilo (4)

Hilo (0,0)	Hilo (0,1)	Hilo (0,2)
Hilo (1,0)	Hilo (1,1)	Hilo (1,1)
Hilo (2,1)	Hilo (2,1)	Hilo (2,1)

Figura 2.3: Organización de los hilos. A la izquierda, hilos como vector, a la derecha, estructura matricial

2.1.2 Bloques

Una vez explicado el concepto de hilo se puede entender el concepto de bloque como un conjunto de hilos. Dichos bloques cumplen las siguientes características:

- Pueden contener un conjunto de hilos almacenados como vector, matriz o matriz tridimensional.
- Una vez que se ha ejecutado el kernel no se puede cambiar el tamaño del bloque.
- Todos los hilos de un mismo bloque pueden cooperar entre sí, mientras que hilos de distintos bloques no pueden realizar dicha operación.
- Los bloques a su vez se agrupan en una malla o grid.
- Poseen un identificador bidimensional.
- El identificador de hilo es único dentro de cada bloque.

2.3. Comparativa CPU/GPU

A modo de ejemplo y de manera conceptual vamos a exponer un ejemplo para comprender las diferencias que existen a la hora de ejecutar un algoritmo utilizando únicamente la CPU o la combinación CPU y GPU.

El programa de ejemplo parte de un vector de números aleatorios. En el caso de que alguno de los elementos sea par se sustuirá por cero, en caso contrario no se modificará.

El Algoritmo 1 se ejecuta en su totalidad en la CPU. En la línea (1) generamos un vector V de componentes aleatorias. Con el fin de conocer la naturaleza de cada elemento se utilizará una sentencia condicional que determinará si es par o impar y realizará las operaciones pertinentes. Dichas operaciones se ejecutarán de manera iterativa mediante un bucle for el cual indexará al vector V. Por lo tanto el cuello de botella se da en este punto, ya que la ejecución del algoritmo es secuencial y es necesario para continuar el análisis que se haya completado la operación de la iteración anterior.

Algoritmo 1 Programa íntegramente en la CPU

```
1: Vk ⟵ GENERA_VECTOR_ALEATORIO (n_elementos);
2: for k=1 to n_elementos do
3:     if( Vk % 2 = 0 ) then
4:         Vk=0;
5:     end
6: end
```

El Algoritmo 2 se ejecuta en la CPU hasta la línea número 2. En este punto se crean tantos hilos en la GPU como número de elementos, n_elementos, tenga el vector. Dichos hilos mantienen la estructura del Algoritmo 3. Cada hilo dispondrá de su número de identificación así como del vector original (V_k). La operación que deberá realizar cada hilo es tan simple como analizar y ejecutar las operaciones sobre el elemento del vector que corresponde a su identificador de hilo. Si el identificador de hilo es el cinco, dicho hilo analizará el elemento número cinco. La diferencia es notable comparada con el Algoritmo 1 pues si el vector tuviese 200 elementos estaríamos operando sobre los 200 de manera simultánea, sin la necesidad de esperar la terminación de la vuelta de bucle anterior.

Algoritmo 2 Programa para la ejecución en la GPU

```
1: Vk ⟵ GENERA_VECTOR_ALEATORIO (n_elementos);
2: EJECUTAR_KERNEL_GPU (n_elementos,Vk);
3: Vk ⟵ RECUPERAR_DATOS_GPU;
```

Algoritmo 3 Kernel

```
1: id ⟵ OBTENER_IDENTIFICADOR_HILO();
2: if (Vid % 2=0) then
3:     Vid=0;
4: end
```

3 LENGUAJES DE PROGRAMACIÓN

El lenguaje original utilizado para programar en CUDA es una variación del lenguaje de programación C. Sin embargo es posible utilizar otros como C++, Python, Fortran, Java y MATLAB.

Para este proyecto nos hemos centrado principalmente en MATLAB, ya que está incrementando el número de bibliotecas que facilitan la migración a la GPU. Esto permite programar sobre la GPU sin necesidad de usar CUDA, lenguaje más complejo por las múltiples consideraciones a tener en cuenta relacionadas con la arquitectura y características del dispositivo. No obstante, si MATLAB no cumple o satisface los requisitos necesarios, nos ofrece la posibilidad de utilizar código de C e insertarlo de manera intuitiva.

Por esto la metodología a la hora de diseñar el algoritmo ha sido:

1-Programar en MATLAB utilizando las bibliotecas que ofrece.

2-Programar en C siempre que las bibliotecas fuesen insuficientes.

3.1. CUDA C/C++

El lenguaje de programación CUDA pertenece a los llamados modelos de programación heterogénea, grupo que engloba sistemas en los que encontramos dispositivos de distinta naturaleza trabajando de manera coordinada. En nuestro caso hablamos de la cooperación entre una CPU y una GPU. Al usar este tipo de programación coordinada se debe de utilizar estructuras como la descrita en la figura 3.1 para asegurar el correcto funcionamiento.

Figura 3.1: Metodología de diseño de algoritmos para el modelo CUDA.

Como ya se ha comentado, no explotaremos toda la capacidad de CUDA C ya que MATLAB puede encargarse de prácticamente todos los pasos anteriores de una manera más sencilla.

Utilizaremos CUDA C entonces para desarrollar las funciones que se ejecutarán en la GPU. Observando el diagrama de la figura esto se corresponde principalmente al cuarto paso, programación de los Kernels.

De acuerdo a esto, no se considerarán los aspectos relacionados con la programación del código del host, encargado de la reserva de memoria y transferencia de información entre ambos dispositivos.

En los apartados siguientes se describirá todo lo necesario para entender las operaciones, comandos y estructuras necesarias para programar el kernel utilizando CUDA C.

3.1.1 Declaración de las funciones

Puesto que el objetivo final de este apartado es la obtención del código que se ejecutará en el d*evice* comenzaremos con la declaración de las funciones. En C, para declarar una función se utilizan los comandos void, int, float entre otros según el tipo de valor de retorno. En CUDA además de disponer de dichas etiquetas debemos incorporar uno de los comandos siguientes según el modo de ejecución de las funciones:

Para una ejecución desde el *host*: En el caso de realizar un volcado de datos desde el *host* para que el *device* los procese se utilizará el comando __global__.

__global__ void nombre_función(…)

Para una ejecución desde el *device*: En el caso de que el *kernel* necesite ejecutar alguna función auxiliar sin la necesidad de transferir la información al *host* utilizaremos el comando __device__.

__device__ void función_auxiliar(…)

3.1.2 Identificación de Hilo

Como se mencionó en el Capítulo 2, existe una serie de parámetros que permite a cada hilo identificarse de manera unívoca dentro de la estructura jerarquizada que presenta la GPU. Dicho mecanismo consiste en la obtención de un entero característico mediante una serie de comandos.

- blockIdx.x, blockIdx.y, blockIdx.z: variables que devuelven el identificador del bloque en la dimensión correspondiente.
- threadIdx.x, threadIdx.y, threadIdx.z variables que devuelven el identificador de hilo en la dimensión correspondiente
- blockDim.x, blockDim.y, blockDim.z variable que devuelve el número de hilos que contiene los bloques en la dimensión correspondiente.

Estas variables se combinan para formar estructuras como la siguiente:

int idx = blockIdx.x*blockDim.x + trheadIdx.x;

identificador (idx)	0	1	2	3	4	5	6	7	8	**9**	10	11
threadIdx.x	**0**	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**	**0**	**1**	**2**
blockIdx.x	bloque 0			bloque 1			bloque 2			bloque 3		

Figura 3.2: Esquema de memoria, identificación de hilos.

Supongamos que el idx tiene asociado el número 9, las variables anteriores tomarían los siguientes valores:

blockIdx.x=3
blockDim.x=3 } idx=3*3+0=9
threadIdx.x=0

3.2. Matlab:

MATLAB es un lenguaje de programación de alto nivel orientado al cálculo numérico, visualización y programación. Permite trabajar de manera interdisciplinar gracias a la cantidad de herramientas que ofrece como puede ser procesamiento de señales e imagen, comunicaciones, sistemas de control y un largo etc. Por consiguiente es una buena opción para nuestro proyecto debido a que tenemos que aunar de manera eficiente el tratamiento de imágenes junto con la programación paralela.

3.2.1 Toolbox:

Se le denomina *toolbox*, (caja de herramientas) a un conjunto de funciones algoritmos y aplicaciones que nos permiten ampliar las funcionalidades básicas del lenguaje de programación de MATLAB de manera relativamente sencilla. Para este proyecto se han utilizado fundamentalmente dos:

-Image Processing Toolbox version 9.2: Orientado al procesamiento, análisis de imágenes, segmentación, mejora y reducción de ruido.

-Parallel Computing Toolbox version 6.6: Orientado a la utilización de procesadores multinúcleo, GPUs y cluster.

3.2.2 Programación CUDA en Matlab

Volviendo a la metodología de programación en CUDA, Figura 3.1, Matlab será el lenguaje de programación principal encargado del tratamiento de los datos, reserva y alojamiento de memoria, inicialización de variables y transferencia de información entre dispositivos.

Vamos a explicar las funciones, algoritmos y consideraciones más relevantes a la hora de programar CUDA sobre MATLAB.

```
Dispositivo =

  CUDADevice with properties:

                      Name: 'GeForce GTX 850M'
                     Index: 1
         ComputeCapability: '5.0'
            SupportsDouble: 1
             DriverVersion: 7
            ToolkitVersion: 6.5000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 2.1475e+09
           AvailableMemory: 1.9931e+09
       MultiprocessorCount: 5
              ClockRateKHz: 901500
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 1
          CanMapHostMemory: 1
           DeviceSupported: 1
            DeviceSelected: 1

fx >>
```

Figura 3.3: Información gpuDevice()

En primer lugar debemos asegurarnos que existe una comunicación ente el dispositivo y el entorno de programación, para ello utilizaremos la función *gpuDevice();* función que nos devuelve la información de los dispositivos conectados.

En nuestro caso dicha función arroja los resultados que podemos apreciar en la figura 3.3.

3.2.2.1 Transferencia de información

Como ya se ha explicado, la GPU consta de una memoria para la ejecución y almacenamiento de los programas y datos necesarios. A la hora de ejecutar un kernel cada hilo necesitará acceso a la información que requiera la función. Este procedimiento se gestiona en MATLAB mediante dos funciones:

A=gpuArray(X) Copia el valor de X a la GPU y devuelve un objeto A que puede ser manipulado por funciones que soporten el tratamiento paralelo. A esos objetos se les denomina de tipo gpuArray.

Res=gather(A); Copia el valor de A desde la GPU a la variable Res de la CPU. En algunos casos esta función no es necesaria puesto que MATLAB ya la contempla en sus funciones.

Como se pudo ver en el algoritmo 2 y 3 la estructura de un programa que se desea programar en la GPU consta de código que se ejecutará en la CPU junto con código que se ejecutara en cada hilo en la GPU. Por otro lado Matlab nos permite ejecutar funciones directamente sin necesidad de preocuparnos por esta estructura.

Diferenciaremos entonces las funciones ya programadas en Matlab o en los toolbox, y funciones o algoritmos que debemos programar por nosotros mismos. En las funciones ya programadas no es necesario realizar prácticamente ningún tratamiento fuera de lo normal mientras que las funciones propias necesitan la configuración de ciertos parámetros adicionales.

3.2.2.2 Funciones preprogramadas

En esta categoría contemplamos las funciones que en condiciones normales pueden ser utilizadas para un tratamiento en la CPU pero que soportan tratamiento en paralelo simplemente por el hecho de trabajar sobre objetos de tipo gpuArray o mediante alguna modificación muy sencilla.

abs	cov	flip	Nan
acos	cross	floor	mpower
atan2	ctranspose	gather	pinv
arrayfun	cumsum	gradient	plot
bitcmp	det	idivide	rand
btainc	diag	ifft	randn
besselj	double	imag	rank
ceil	eig	Inf	size
complex	exp	interp1	sqrt
cond	eye	isreal	tan
conj	factorial	ldivide	transpose
conv	false	length	true
conv2	fft	mat2str	uint8
cos	fft2	max	zeros

Figura 3.4: Muestra de funciones disponibles en el entorno de programación.

Las funciones element-wise permiten operar sobre cada elemento de un vector o matriz. Si queremos realizar una operación aritmética, suma, multiplicación, exponencial… simplemente debemos colocar un "." delante del operador. Al utilizar la GPU podemos realizar el mismo tratamiento pero de manera paralela utilizando:

```
[B1,...,Bm] = arrayfun(func,A1,...,An)
```

Donde func es cualquier función perteneciente al repertorio de Matlab, operación aritmética o funciones escritas por el propio usuario, An son los argumentos de entrada de la función y Bm son las salidas.

Cada núcleo de la GPU realiará dicha operación sobre el elemento que le corresponda, mientras que la CPU tiene que operar sobre cada elemento de manera secuencial.

3.2.2.3 Ejecución de código externo a Matlab

Una vez explotados todos los recursos que ofrece Matlab de manera nativa o mediante los toolbox, podemos utilizar dos mecanismos para introducir programas externos. Disponemos de dos métodos que nos sirven de pasarela entre Matlab y, en nuestro caso, programas en C. Dichos mecanismos son las funciones mex y la utilización de la función parallel.gpu.CUDAKernel.

3.2.2.3.1 Funciones mex

La función mex no es un mecanismo restringido al procesamiento de la GPU sino que se extiende a cualquier código en C externo que queramos incorporar a MATLAB. Este método permite añadir más funciones a las que ya disponemos de serie y utilizarlas exactamente igual que éstas.

El proceso es tedioso ya que hay que adaptar y compilar nuestro código siguiendo una serie de especificaciones determinadas para asegurar que nuestro programa pasa a formar parte de la biblioteca de MATLAB.

3.2.2.3.2 Función parallel.gpu.CUDAKernel y feval

Con estas funciones, pertenecientes al *Parallel computing toolbox*, podemos ejecutar un *kernel* programado en C de manera similar a como lo haríamos en CUDA, pero gestionando los recursos desde el entorno de MATLAB. En primer lugar se crea un objeto en el entorno de trabajo de MATLAB en el que se determinan ciertos parámetros importantes a la hora de ejecutar el código en la GPU. En segundo lugar se ejecutan las funciones relacionadas con dicho objeto introduciendo las variables y atributos de entradas que la función requiera.

La creación del objeto tipo *parallel.gpu.CUDAKernel* se realiza con la siguiente función:

```
KERN = parallel.gpu.CUDAKernel(PTXFILE, CUFILE, FUNC);
```

Esta función crea un objeto (KERN) introduciendo como parámetros de entrada los siguientes argumentos:

- PTXFILE: Archivo de extensión .ptx (***Parallel Thread Execution***). Contiene las instrucciones necesarias para ejecutar el programa enfocándolo a la ejecución sobre los hilos. Usa un lenguaje semejante a ensamblador y se puede obtener tras compilar el archivo .cu.
- CUFILE: archivo de extensión .cu que contiene el kernel escrito en CUDA C.
- FUNC: Nombre de la función dentro de CUFILE que queremos ejecutar.

Una vez creado el objeto es necesario modificar ciertos campos internos para personalizar la ejecución del kernel como el número de hilos y bloques. Dicho procedimiento se lleva a cabo accediendo al objeto como una estructura y dándole valor al campo ThreadBlockSize de la siguiente manera:

NOMBRE DEL OBJETO.ThreadBlockSize=[]

La ejecución de la función relacionada con el objeto parallel.gpu se realiza mediante el uso de la

función feval de la siguiente manera:

[y1, y2] = feval(KERN, x1, x2, x3)

3.3. Ejemplos de programación:

El siguiente programa aplica todos los aspectos anteriores tanto en MATLAB como en CUDA C. Las tareas que se van a realizar consisten en compartir un vector con la GPU, tanto envío como recepción de los datos ya procesados, y la escritura de los identificadores de hilo en cada elemento del vector.

Matlab(host):

```
%Creación del objeto relacionado al archivo Ex_hilos.cu

KERN = parallel.gpu.CUDAKernel('Ex_hilos.ptx','Ex_hilos.cu','indx')

%Utilizamos una operación de matlab ya preprogramada que crea un vector de tamaño 10
donde cada elemento es un uno.

v=ones(10,1,'gpuArray'); %Al colocar el comando 'gpuArray' se crea en la gpu

N_elementos=numel(v); %N° de elementos del vector v

%Numero de hilos:

KERN.ThreadBlockSize = [N_elementos,1,1]; %Creamos un hilo por cada elemento

%Ejecutamos la función, ordenamos a la CPU ejecutar el kernel KERN, ya configurado,
teniendo como dato de entrada el vector v:

result_GPU = feval(KERN,v);

%Copiamos la información de la GPU a la CPU

resultado_CPU=gather(result_GPU);
```

Cuda (device)

Al ser una función de tipo __global__ estamos determinando que la comunicación existente entre la GPU y la CPU será bidireccional, la información resultante será rescatada posteriormente por la CPU. La función indx recibe un puntero al vector v sobre el que trabajará. Obtiene su identificador de hilo y lo almacena en la variable idx para posteriormente introducirlo en la componente idx del vector v.

```
__global__ void indx(double * v)
{
  int idx = blockDim.x * blockIdx.x + threadIdx.x; //identificador de hilo
  v[idx] =idx; //el hilo idx actua sobre la componente v[idx]
}
```

Los resultados obtenidos serán parecidos a lo siguiente:

```
Hilo 0 -> v[0] = 0
Hilo 1 -> v[1] = 1
Hilo 3 -> v[3] = 3
Hilo 5 -> v[5] = 5
```

La ejecución de los hilos no es ordenada, no podemos asegurar que el hilo 1 termine antes que el hilo 2.

4 ALGORITMOS DE VISIÓN ARTIFICIAL

En este capítulo vamos a describir los dos algoritmos y programas estudiados en este trabajo así como las características más importantes y consideraciones a tener en cuenta a la hora de implementarlos en sus respectivos lenguajes de programación. Por otro lado se realizará un pequeño estudio para comprender las diferencias entre los algoritmos programados haciendo uso de la CPU únicamente y los programas que utilizan el cálculo paralelo en la GPU.

4.1 Flujo Óptico

Cuando tratamos con imágenes estamos trabajando sobre proyecciones en dos dimensiones de un entorno tridimensional (Figura 4.1) es decir, no podemos captar toda la información completa.

Dada una secuencia de imágenes podemos estimar la proyección de un movimiento tridimensional sobre el plano de la imagen del observador utilizando diversas técnicas. Como producto de estos tratamientos se puede obtener el campo de velocidades (Figura 4.2) y los desplazamientos de los pixeles dentro de la imagen, información que puede utilizarse para recomponer el movimiento real del observador así como para detección de objetos, segmentación de objetos y detección de colisiones entre otras muchas aplicaciones más.

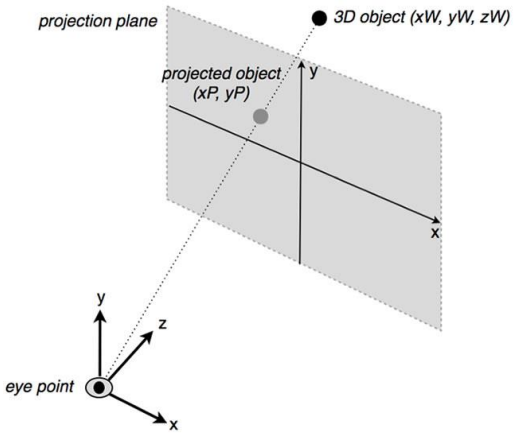


Figura 4.1: Proyección de un punto 3D en el plano de la imagen

Figura 4.2: Flujo óptico

4.2 Descripción del problema:

4.2.1 Visión general

Una de las opciones para la obtención del flujo óptico es el llamado método de Lucas-Kanade con el cual se puede calcular una aproximación bastante buena. En la figura 4.3 podemos apreciar, de manera general, las operaciones que hay que llevar a cabo para el cálculo del campo de velocidades. De una secuencia de imágenes se seleccionan dos consecutivas a las cuales se les realiza un tratamiento visual y una serie de operaciones propias del método LK para obtener el resultado final y mostrarlo si fuese necesario.

Para una correcta comprensión del problema se va a dividir esta sección en tres subapartados relacionados con la información de entrada requerida, los procedimientos más importantes que se llevan a cabo y por último con la forma en la que se muestran los resultados.

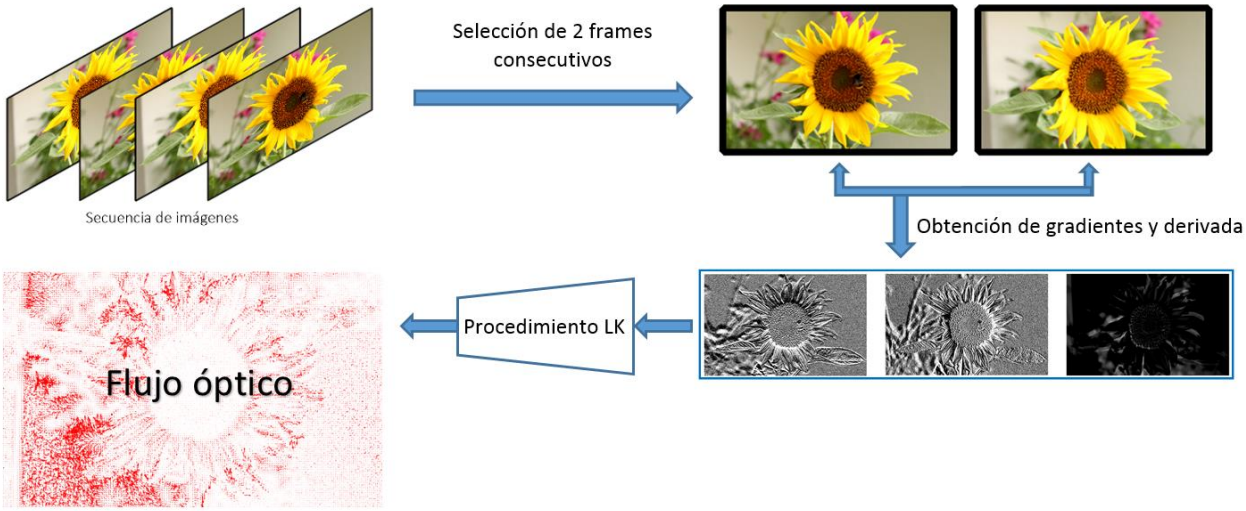


Figura 4.3: Esquema descriptivo del problema de obtención del flujo óptico.

4.2.2 Información de entrada:

El punto desde el que partimos para llevar a cabo el método de Lucas-Kannade es una secuencia de imágenes. El tamaño de ésta es indiferente puesto que, a pesar de analizar un video con un número considerable de frames, el procedimiento se va aplicando sobre un par de imágenes consecutivas. Repitiendo el tratamiento sobre los distintos pares se obtiene un análisis completo de todo el video.

Puesto que al final trabajamos sobre imágenes, es necesario describir y conocer el formato y la estructura de este tipo de archivo. El algoritmo LK procesa un tipo de imagen muy concreta, imágenes en escala de grises o imágenes de intensidad lumínica, por lo que es muy posible que sea necesario realizar un pequeño tratamiento en el caso de no partir de dicha información.

4.2.2.1 Matrices de intensidad, imágenes en escala de grises.

Una imagen digital en escala de grises es una imagen en la que cada pixel toma un valor comprendido entre 0 y 255. Este valor está relacionado con la medida de intensidad que captura el sensor de la cámara para un determinado rango del espectro electromagnético, es decir, capturamos la intensidad lumínica de un color.

A la hora de realizar una captura fiel a la realidad, las cámaras recogen la información de luminosidad de distintas frecuencias para recomponer la imagen siguiendo una serie de modelos o formatos, entre ellos vamos a destacar el utilizado en este proyecto, el modelo RGB.

4.2.2.2 Representación de una imagen RGB.

Es un modelo de color basado en la síntesis aditiva, con el que es posible representar cualquier color mediante la mezcla de los tres colores de luz primarios. Su nombre, hace alusión a los tres colores involucrados, rojo, verde y azul.

Matlab permite el almacenamiento de imágenes utilizando este sistema mediante la creación de una matriz tridimensional compuesta por tres matrices bidimensionales, correspondientes a los niveles de intensidad lumínica de cada color.

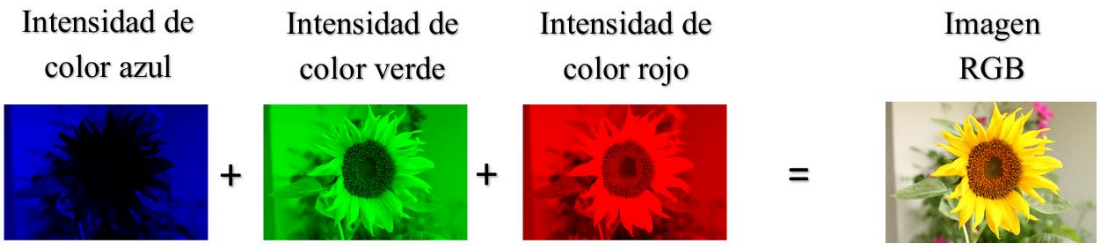


Figura 4.4: Imágenes en formato RGB

Como ya se ha dicho el método LK procesa matrices de intensidad, y el modelo RGB dispone de tres matrices distintas. En el caso de utilizar solamente una de ellas estaríamos perdiendo información importante. Para solventar este problema existen distintos tipos de procedimientos para convertir imágenes RGB a escala de grises que consisten en ponderar la información de las distintas capas para generar una única imagen.

4.2.3 Procesamiento:

Una vez que se dispone de la matriz de intensidad de la imagen se realizan dos procesamientos fundamentalmente:

Selección de pixeles: El algoritmo LK está enfocado a obtener el desplazamiento de los píxeles de la imagen. Para ello es necesario seleccionar que pixeles queremos tratar. El método de selección y el número de pixeles son dos parámetros que pueden afectar al tiempo de ejecución del método.

Obtención de derivadas temporales y gradientes: Estas dos operaciones arrojan información necesaria sobre la que se aplica la formulación.

Método de los mínimos cuadrados: Cuya aplicación permite obtener los valores del desplazamiento de los pixeles de la imagen.

4.2.4 Información de salida:

Puesto que los resultados del algoritmo son los desplazamientos de los píxeles, y teniendo en cuenta que las imágenes son bidimensionales, la información de salida se puede representar gráficamente de manera muy sencilla de manera vectorial. Conocemos el punto de origen, los pixeles ya se han seleccionado previamente, y los valores del módulo del vector, obtenidos tras la resolución por mínimos cuadrados.

Otro forma de representar la información es realizando una tabla de equivalencias, para cada dirección del desplazamiento se utilizará un color determinado y según el módulo del desplazamiento se aplicará mayor o menor intensidad a dicho color.

Puesto que la idea principal es conocer la orientación y movimiento del entorno la representación del flujo óptico más intuitiva seria la representación vectorial.

4.3 Hipótesis de partida:

El método de Lucas-Kanade se basa en una serie de hipótesis para desarrollar toda su formulación. Estas hipótesis condicionan el funcionamiento y los resultados del algoritmo. Las suposiciones más características son las siguientes:

-Conservación de la intensidad: Los valores de intensidad se mantienen constantes entre los fotogramas de la secuencia.

-Conservación del gradiente: Al conservarse los valores de intensidad podemos deducir que también se conservan los valores del gradiente.

-Se consideran que los términos de orden superior a la hora de calcular derivadas espaciotemporales son lo suficientemente pequeños como para ser despreciados.

-Se considera que los desplazamientos entre frames son pequeños.

La suposición de la conservación de la intensidad es muy ambiciosa y no suele cumplirse en la práctica debido a factores como variación de la luminosidad, ruido, movimientos de los objetos o la no idealidad de los dispositivos de captura. A pesar de ello los resultados son válidos ya que, aunque no se cumpla a nivel puntual, se mantiene constante si analizamos la vecindad de dicho punto.

4.4 Aspectos matemáticos:

Basándonos en las hipótesis del apartado 4.3 vamos a desarrollar la formulación utilizada en el método de Lucas-Kanade, subapartado 4.4.1, y además se ejemplificará su funcionamiento para un caso muy sencillo en el subapartado 4.4.2.

4.4.1 Estimación del flujo óptico mediante el gradiente, Método de Lucas-Kanade

El material del que partimos es una secuencia ordenada de imágenes, mínimo dos para poder realizar la operación, cuyo objetivo final es obtener la posición o desplazamiento de los pixeles entre los integrantes de dicha secuencia. Para ello consideraremos las hipótesis descritas en el apartado anterior para simplificar el problema.

4.4.1.1 Ecuación de continuidad y conservación de la intensidad.

En primer lugar asumíamos que las intensidades en cada pixel se conservaban de un frame al siguiente:

$$\boldsymbol{I}(\vec{\boldsymbol{x}}, \boldsymbol{t}) = \boldsymbol{I}(\vec{\boldsymbol{x}} + \vec{\boldsymbol{u}}, \boldsymbol{t} + \mathbf{1}) \tag{4-1}$$

Donde $\boldsymbol{I}(\vec{\boldsymbol{x}}, \boldsymbol{t})$ es la intensidad de la imagen en función del espacio $\vec{\boldsymbol{x}} = (x, y)$, $\vec{\boldsymbol{u}}$ es del desplazamiento y **t** el tiempo, teniendo en cuenta que son elementos discretos. Esta primera hipótesis no se cumple en la mayoría de los casos ya que ni las condiciones del entorno se mantienen inmutables entre las capturas de las imágenes, hay variación de luminosidad, ni los dispositivos de captura son capaces de captar la misma información sobre el mismo objeto ya que no son ideales. A pesar de esto en la práctica dicha suposición arroja buenos resultados puesto que aunque no podamos hablar de conservación a nivel de pixeles, si podemos aceptar que la luminosidad se va a mantener constante dentro de unos márgenes y radio alrededor del pixel.

Por otro lado asumimos que los desplazamientos de los pixeles de la imagen son muy pequeños entre los distintos elementos de la secuencia y que las propiedades de estos se mantienen constantes en una vecindad.

Partiendo de dichas suposiciones y con el fin de obtener el desplazamiento se realiza una aproximación mediante Taylor de primer orden sobre la ecuación (4-1).

$$\boldsymbol{I(x+u,t+1) \approx\ I(x\,,t+\,,t+1) + u\nabla I(x\,,t) + It(x\,,t)} \tag{4-2}$$

Donde $\boldsymbol{\nabla I(\vec{x}\,,t)}$ es el gradiente, $\boldsymbol{I_t}$ es la derivada temporal de la imagen **I** y $\vec{\boldsymbol{u}}$ es el desplazamiento en 2D o velocidad.

Despreciando los términos de orden superior y combinando la ecuación (4-1) y (4-2) obtenemos:

$$\boldsymbol{\nabla I(\vec{x}\,,t)\vec{u} + I_t(\vec{x}\,,t) = 0} \tag{4-3}$$

Podemos reescribir la ecuación anterior como:

$$\boldsymbol{I_x(q_1)u_1 + I_y(q_1)u_2 = -I_t(q_1)} \tag{4-4}$$

Donde $\boldsymbol{f(q_1)}$ es cualquier función centrada en el pixel q_1. Si se cumpliese la hipótesis de conservación de la intensidad a nivel de píxeles la ecuación (4-4) sería suficiente. Como esto no ocurre se tiende a aplicar dicha ecuación sobre los puntos alrededor del pixel central q_1 formándose un sistema de ecuaciones.

4.4.1.2 Vecindad del pixel y aplicación de ecuación de continuidad

Como se ha asumido que la intensidad de la imagen se conserva dentro de una vecindad o radio determinado optaremos por tomar un número de pixeles alrededor de nuestro punto central, dicha vecindad se elegirá en función de una cuadricula de tamaño w puntos.

$$\begin{aligned}
&\boldsymbol{I_x(q_1)u_1 + I_y(q_1)u_2 = -I_t(q_1)}\\
&\boldsymbol{I_x(q_2)u_1 + I_y(q_2)u_2 = -I_t(q_2)}\\
&\boldsymbol{I_x(q_3)u_1 + I_y(q_1)u_2 = -I_t(q_3)}\\
&\dots\\
&\boldsymbol{I_x(q_9)u_1 + I_y(q_9)u_2 = -I_t(q_9)}
\end{aligned} \tag{4-5}$$

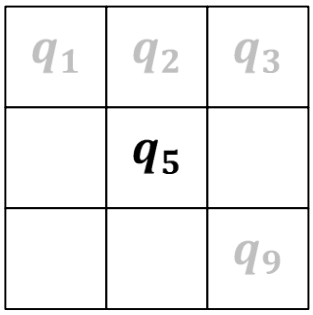


Figura 4.5: Vecindad o cuadricula alrededor del elemento q5 de tamaño 3x3.

El método propuesto por Lucas-Kanade consiste en resolver las ecuaciones anteriores con el fin de obtener el desplazamiento o velocidad $\vec{\boldsymbol{u}}$:

$$A = \begin{bmatrix} \boldsymbol{I_x(q_1)} & \boldsymbol{I_y(q_1)} \\ \boldsymbol{I_x(q_2)} & \boldsymbol{I_y(q_2)} \\ \vdots & \vdots \\ \boldsymbol{I_x(q_n)} & \boldsymbol{I_y(q_n)} \end{bmatrix} \vec{\boldsymbol{u}} = \begin{bmatrix} \boldsymbol{u_1} \\ \boldsymbol{u_2} \end{bmatrix} \qquad b = \begin{bmatrix} \boldsymbol{-I_t(q_1)} \\ \boldsymbol{-I_t(q_2)} \\ \vdots \\ \boldsymbol{-I_t(q_n)} \end{bmatrix} \qquad \text{A}\vec{\boldsymbol{u}} = \text{b} \tag{4-6}$$

Puesto que el sistema presentado está sobredeterminado la solución buscada será una solución de compromiso mediante el método de mínimos cuadrados.

Anticipándonos a una futura implementación del algoritmo se tiende a buscar expresiones susceptibles de ser programadas de manera sencilla, para ello:

$$v = (A^T A)^{-1} A^T b \tag{4-7}$$

De la ecuación (4-6) se deduce que:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i \boldsymbol{I_x(q_i)^2} & \sum_i \boldsymbol{I_x(q_i) I_y(q_i)} \\ \sum_i \boldsymbol{I_x(q_i) I_y(q_i)} & \sum_i \boldsymbol{I_y(q_i)^2} \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i \boldsymbol{I_x(q_i) I_t(q_i)} \\ -\sum_i \boldsymbol{I_y(q_i) I_t(q_i)} \end{bmatrix} \tag{4-8}$$

Resolviendo el sistema obtenemos las siguientes expresiones:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{\sum_i \boldsymbol{I_{xi}}^2 \sum_i \boldsymbol{I_{yi}}^2 - (\sum_i \boldsymbol{I_{xi} I_{yi}})^2} \begin{bmatrix} \sum_i \boldsymbol{I_{xi}}^2 & \sum_i \boldsymbol{I_{xi} I_{yi}} \\ \sum_i \boldsymbol{I_{xi} I_{yi}} & \sum_i \boldsymbol{I_{yi}}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i \boldsymbol{I_{xi} I_{ti}} \\ -\sum_i \boldsymbol{I_{yi} I_{ti}} \end{bmatrix} \tag{4-9}$$

Realizando las operaciones pertinentes sobre la ecuación anterior llegamos al resultado final que puede ser implementado fácilmente en cualquier lenguaje de programación.

$$u = \frac{-\sum_i \boldsymbol{I_{xi}}^2 \sum_i \boldsymbol{I_{xi} I_{ti}} + \sum_i \boldsymbol{I_{xi} I_{yi}} \sum_i \boldsymbol{I_{yi} I_{ti}}}{\sum_i \boldsymbol{I_{xi}}^2 \sum_i \boldsymbol{I_{yi}}^2 - \left(\sum_i \boldsymbol{I_{xi} I_{yi}}\right)^2} \tag{4-10}$$

$$v = \frac{\sum_i \boldsymbol{I_{xi} I_{ti}} \sum_i \boldsymbol{I_{xi} I_{yi}} - \sum_i \boldsymbol{I_{xi}}^2 \sum_i \boldsymbol{I_{yi} I_{ti}}}{\sum_i \boldsymbol{I_{xi}}^2 \sum_i \boldsymbol{I_{yi}}^2 - \left(\sum_i \boldsymbol{I_{xi} I_{yi}}\right)^2}$$

4.4.2 Ejemplo matricial:

Basándonos en la información anterior vamos a analizar un ejemplo de aplicación de lo desarrollado en el apartado 4.4.1, con el fin de comprender el alcance de toda la formulación matemática que exige el método de Lucas-Kanade. Debido a que la idea del ejemplo es meramente explicativa se va a suponer que se cumplen completamente las hipótesis descritas en el apartado 4.3, se recuerda que los resultados aplicados a una imagen real pueden variar.

4.4.2.1 Planteamiento del problema:

La idea principal es muy simple, dadas dos imágenes pertenecientes a una secuencia de movimiento continuo, el método es capaz de identificar el desplazamiento de los píxeles en dichos frames y de mostrarlo en forma de vector.

Para nuestro ejemplo partiremos de dos matrices de 6x6 elementos que simulan las medidas de intensidad de dos imágenes consecutivas y se obtendrá el desplazamiento del pixel correspondiente a las coordenadas (2,2) en la figura 4.6. Los valores de intensidad están comprendidos entre 0 (negro) y 255 (blanco), aunque en las

figuras se representen de distintos colores para facilitar la comprensión. Se ha representado una figura geométrica desplazándose en el espacio desde el punto (2,2) en la imagen 1 hasta el punto (3,2) en la imagen 2, es decir un desplazamiento horizontal de 1 pixel. En la figura 4.6 (a) se muestran los dos frames superpuestos para apreciar mejor el movimiento.

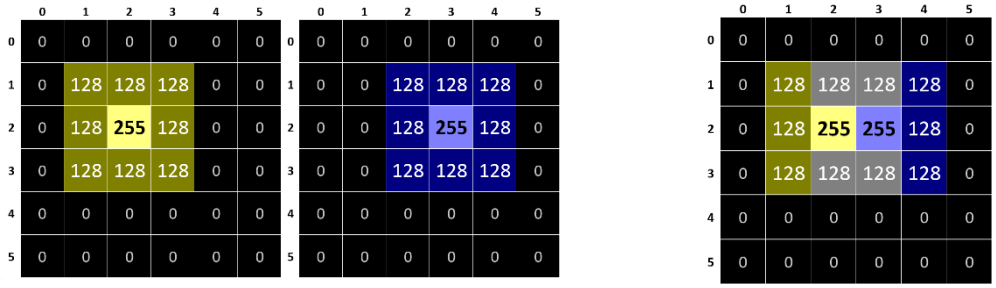


Figura 4.6: img1 e img2 corresponden a la primera y segunda imagen de la secuencia seleccionada. (a)

4.4.2.2 Procesamiento y cálculos:

Recordando lo expuesto en el apartado 4.2 y la formulación del apartado 4.4.1 es necesario obtener los gradientes espaciales y derivada temporal alrededor en una vecindad del píxel seleccionado, que en nuestro caso será de 3x3 elementos. En la figura 4.6 se representa la imagen 1 de la secuencia en formato matricial.

4.4.2.2.1 Cálculo de los gradientes

Existen varios métodos para la obtención de los gradientes, en este caso aplicaremos la formulación siguiente debido a su sencillez y por ser una extensión directa de la formulación del gradiente para un campo escalar:

$$\nabla f(x,y) = \left[\frac{\partial f(x,y)}{\partial x} \quad \frac{\partial f(x,y)}{\partial y}\right]^T$$

Donde $\frac{\partial f(x,y)}{\partial x}$ es la componente en el sentido del eje de abscisas del gradiente y $\frac{\partial f(x,y)}{\partial y}$ su homónimo en el eje de ordenadas. Puesto que nuestro sistema es discreto, se tiende a utilizar las siguientes expresiones aproximadas:

$$\frac{\partial f(x,y)}{\partial x} \approx \Delta_x f(x,y) = f(x,y) - f(x-1,y)$$
$$\frac{\partial f(x,y)}{\partial y} \approx \Delta_y f(x,y) = f(x,y) - f(x,y-1)$$

Nuestra función de partida $f(x,y)$ es la imagen 1 de la secuencia. El procedimiento a seguir es muy simple, a cada elemento de la matriz se le restará el contiguo según la orientación deseada. Por ejemplo, si se desea obtener la componente (2,2) del gradiente en el sentido de abscisas se actuará de la siguiente manera:

$$\Delta_x f(2,2) = f(x,y) - f(x-1,y) = f(2,2) - f(1,2) = 255-128 = 127$$

$$f(x,y) \approx img1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{128} & \mathbf{128} & \mathbf{128} & 0 & 0 \\ 0 & \mathbf{128} & \mathbf{255} & \mathbf{128} & 0 & 0 \\ 0 & \mathbf{128} & \mathbf{128} & \mathbf{128} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

El resultado final tras aplicar las ecuaciones sobre los elementos de la matriz y extrayendo la vecindad deseada 3x3 sobre el pixel (2,2) se refleja en las matrices Gx y Gy:

$$Gx = \begin{bmatrix} 127 & 0 & 0 \\ 127 & \mathbf{127} & -127 \\ 127 & 0 & 0 \end{bmatrix} \qquad Gy = \begin{bmatrix} 127 & 127 & 127 \\ 0 & \mathbf{127} & 0 \\ 0 & -127 & 0 \end{bmatrix}$$

4.4.2.2.2 Cálculo de la derivada temporal:

La derivada temporal está relacionada con la variación de la posición de los pixeles entre la imagen 1 y 2 de la secuencia. Partiendo de la derivada temporal de una función continua obtenemos la discretización aproximada aplicable a nuestro caso:

$$\frac{dF(t)}{dt} \approx F(2) - F(1) = img2 - img1 \tag{4-11}$$

$$\frac{dF(t)}{dt} \approx \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{128} & \mathbf{128} & \mathbf{128} & 0 \\ 0 & 0 & \mathbf{128} & \mathbf{255} & \mathbf{128} & 0 \\ 0 & 0 & \mathbf{128} & \mathbf{128} & \mathbf{128} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{128} & \mathbf{128} & \mathbf{128} & 0 & 0 \\ 0 & \mathbf{128} & \mathbf{255} & \mathbf{128} & 0 & 0 \\ 0 & \mathbf{128} & \mathbf{128} & \mathbf{128} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4-12}$$

Donde F(1) corresponde a la imagen 1 de la secuencia y F(2) corresponde a la imagen 2. Aplicando esta ecuación a las dos matrices de intensidad somos capaces de calcular la siguiente matriz asociada a la vecindad del pixel (2,2):

$$Gt = \begin{bmatrix} -128 & 0 & 0 \\ -128 & \mathbf{127} & 127 \\ -128 & 0 & 0 \end{bmatrix} \tag{4-13}$$

4.4.2.2.3 Sistema de ecuaciones

Una vez calculado los gradientes y derivadas podemos constituir el sistema de ecuaciones 4.6. Puesto que se ha utilizado una vecindad de 9 pixeles, 3x3, el método arroja 9 ecuaciones, es decir, trabajaremos con un sistema sobredimensionado que se resolverá por el método de los mínimos cuadrados:

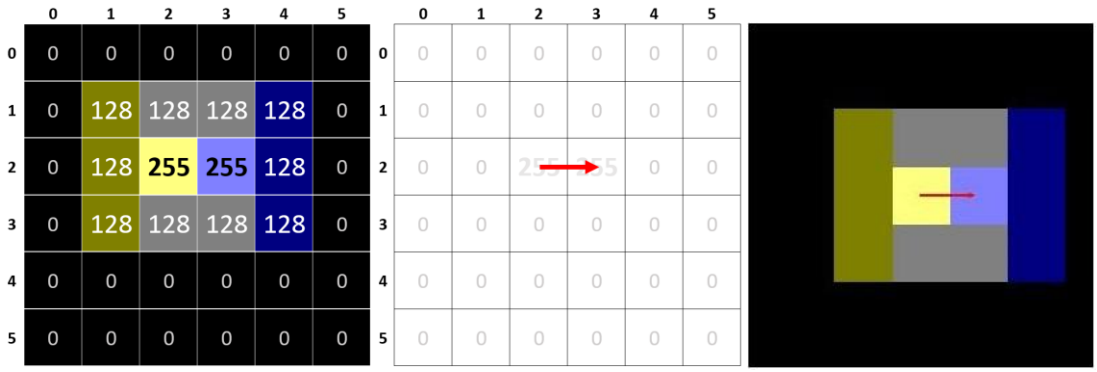
$$\begin{bmatrix} 127 & 0 & 0 \\ 127 & \mathbf{127} & -127 \\ 127 & 0 & 0 \end{bmatrix} u + \begin{bmatrix} 127 & 127 & 127 \\ 0 & \mathbf{127} & 0 \\ 0 & -127 & 0 \end{bmatrix} v = \begin{bmatrix} -128 & 0 & 0 \\ -128 & \mathbf{127} & 127 \\ -128 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 127 & 127 \\ 127 & 0 \\ 127 & 0 \\ 0 & 127 \\ 127 & 127 \\ 0 & -127 \\ 0 & 127 \\ -127 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -128 \\ -128 \\ -128 \\ 0 \\ 127 \\ 0 \\ 0 \\ 127 \\ 0 \end{bmatrix} \tag{4-14}$$

Tras resolver el sistema de ecuaciones 4-14 mediante los mínimos cuadrados llegamos a la conclusión de que el valor de u, desplazamiento horizontal, es de un pixel y el valor de v, desplazamiento vertical, es cero.

4.4.2.3 Resultados:

Conociendo la posición sobre la que se ha realizado los cálculos y disponiendo de los valores del desplazamiento u y v la representación del vector de flujo óptico, origen y componentes vertical y horizontal, queda completamente determinada:



(a) **(b)** **(c)**

Figura 4.7: :(a) Representación de los dos frames superpuestos, (b) Representación del vector de flujo óptico sobre el elemento (2,2), (c) Representación combinada del flujo óptico e imagen superpuestas generada por Matlab.

4.5 Descripción general del algoritmo.

Las expresiones de la sección 4.4 son el fundamento matemático y teórico del algoritmo LK las cuales nos permiten calcular las velocidades de cada pixel en función de la vecindad de estos. A pesar de ello siguen siendo solamente ecuaciones algebraicas en las que se ve implicada información que de partida no es conocida. Para obtener los datos necesarios para resolver el problema anterior se han diseñado dos algoritmos, uno íntegramente en la CPU y otro que utiliza las funcionalidades que ofrece el cálculo paralelo. Vamos a explicar el funcionamiento de ambos algoritmos y las principales diferencias.

En principio ambos programas deben resolver el mismo problema, por lo tanto los dos deben realizar las siguientes tareas básicas:

- INICIALIZACION

Lectura de imágenes ya sea cargándolas de manera independiente o extrayéndolas de un video.

Selección de parámetros como el tamaño de la vecindad w o el número de pixeles a procesar.

Obtención de los gradientes y la derivada temporal.

- OBTENCIÓN DE LAS SOLUCIONES:

Resolver las ecuaciones del apartado anterior para los pixeles determinados.

- MOSTRAR RESULTADOS

Mediante una representación vectorial.

4.5.1 Algoritmo en la CPU

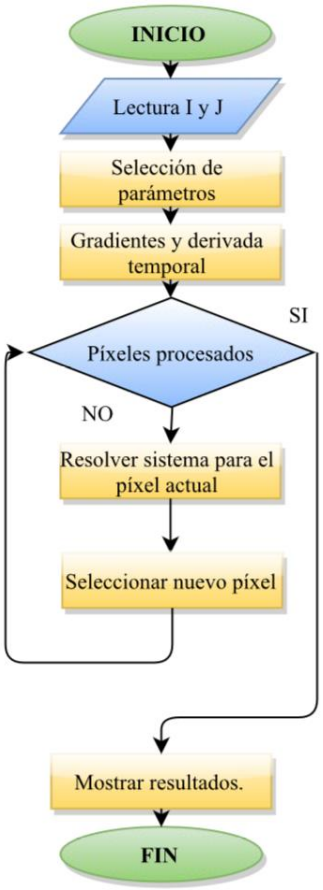


Figura 4.8 : Diagrama del algoritmo implementado en la CPU

Basándonos en la formulación expuesta en el apartado 4.4.4 y utilizando los recursos descritos en los apartados anteriores se ha diseñado la estructura principal que debe tener nuestro programa.

Puesto que el algoritmo está implementado íntegramente en la CPU el lenguaje programación de Matlab cubre todas nuestras necesidades.

Los siguientes subapartados recogerán la información referente a los procesos ilustrados en la Figura 4.8:

- Lectura y tratamiento de imágenes.
- Selección de parámetros.
- Calculo de las derivadas espaciotemporales.
- Aplicación de la formulación del método LK.
- Muestra de los resultados.

4.5.1.1 Lectura I y J

Para poder aplicar la formulación correspondiente necesitaremos dos imágenes como mínimo, por consiguiente, la primera consideración a tener en cuenta es cómo va realizarse el proceso de lectura. Existen distintas formas de obtener las imágenes desde el entorno de programación de Matlab, se pueden importar desde la carpeta de origen que contenga el archivo, se pueden extraer determinados frames de un video o incluso se puede realizar una captura directa desde una cámara conectada al ordenador.

Por otro lado, la información que realmente nos es útil son las matrices de intensidad asociadas a dichas imágenes. La obtención de las matrices se realiza tras la lectura de las imágenes y de aplicar un tratamiento concreto para adecuar la información a nuestras necesidades.

En los apartados obtención de imágenes y tratamiento de imágenes explicaremos los métodos que se han utilizado en este proyecto.

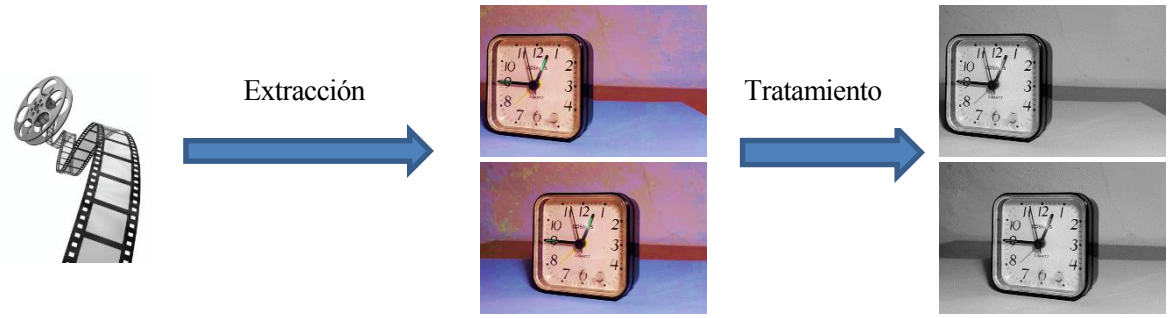


Figura 4.9: Proceso de obtención de las matrices de intensidad desde un video

4.5.1.1.1 Obtención de imágenes:

4.5.1.1.1.1 Extracción de fotogramas desde un archivo de video

Matlab dispone de la función VideoReader que permite importar un video al entorno de programación como una variable de tipo vidObj. El objeto creado contiene información referente al video, nombre, duración, tamaño de imagen, formato entre otras. Además existen funciones diseñadas para manejar este tipo de dato y realizar procesamiento de video:

Para capturar un frame determinado basta con utilizar el comando read() el cual devolverá una imagen en formato RGB.

Combinando ambas funciones podemos descomponer un video en una secuencia de imágenes independientes para tratarlas posteriormente o incluso ir realizando lectura y tratamiento de manera cíclica.

Ejemplo: Captura de un frame

Para el ejemplo supondremos que disponemos de un video en formato mp4 llamado vid1.mp4 y se desea capturar la imagen cuando trascurra 1.5 segundos del inicio. Para ello se podría utilizar el código siguiente:

```
vidorig = VideoReader('vid1.mp4'); %Crea un objeto VidObj asociado al video vid1.mp4

mitiempoini =1.5; %Momento en el que se quiere capturar la imagen
frame ini = fix( mitiempoini*vidorig.FrameRate);%frame correspondiente

fr1=read(vidorig,frame_ini);    %Lectura del primer frame
```

Una vez ejecutado el comando read, el cual recibe por parámetros un objeto vidObj y el número del frame elegido, se almacena la imagen en la variable fr1 en formato RGB que ya estaría lista para su posterior tratamiento.

4.5.1.1.1.2 Lectura de imágenes por archivo

Si disponemos de una imagen que se haya capturado y almacenado previamente, Matlab puede importar dicha imagen con el comando imread(). Esta función crea una matriz tridimensional siguiendo el modelo RGB.

En el caso de disponer de una imagen llamada "imagen.jpg" la lectura se llevaría a cabo simplemente escribiendo el código siguiente:

A = imread("imagen.jpg");

Donde A es el objeto del entorno de programación que almacena la imagen, en nuestro caso, "imagen.jpg".

4.5.1.1.2 Tratamiento de imágenes:

4.5.1.1.2.1 Conversión RGB-Gray, obtención de la matriz de intensidad

Como ya se ha comentado, es muy posible que dispongamos de partida de una imagen en formato RGB. El procedimiento seguido para calcular la matriz de intensidad es crear una matriz bidimensional donde cada elemento se va a obtener como una ponderación de los elementos de las tres capas, rojo, verde y azul de la imagen original. Dichas ponderaciones están relacionadas con la mayor o menor contribución de cada color a la intensidad de la imagen.

Matlab dispone de una función, rgb2gray, que devuelve la matriz de intensidad a partir de una imagen RGB siguiendo el siguiente procedimiento:

```
Imgray = 0.2989 * R + 0.5870 * G + 0.1140 * B
```

Donde R, G, B son los valores de intensidad de cada elemento para cada capa.

4.5.1.1.2.2 Cambio de tamaño

Existen dos casos en los que puede ser aconsejable o incluso necesario modificar el tamaño de la imagen:

```
B = imresize(A,scale)
```

-Imágenes de grandes dimensiones: Pueden provocar tiempos de cómputo muy elevados ya que la cantidad de píxeles se aumenta conforme lo hace el tamaño. Es de esperar que en el caso de la CPU este problema sea más determinante que en la GPU.

-Secuencias con desplazamientos muy grandes: El método de LK partía de la hipótesis de pequeños desplazamientos, si no se cumple dicha premisa el método comienza a arrojar resultados poco fiables. Al reducir el tamaño de la imagen también lo hace los desplazamientos lo que mejora en cierta manera el funcionamiento.

Matlab dispone de la función imresize(), capaz de devolver la imagen reducida o ampliada según un factor determinado introducido por el usuario. La operación que realiza por defecto es una interpolación bicubica aunque dispone de muchos más procedimientos. Por otro lado es una función que soporta el tratamiento paralelo.

4.5.1.2 Selección de parámetros:

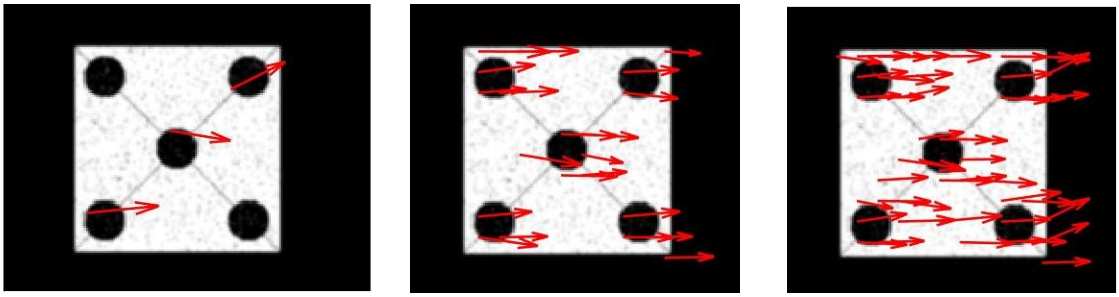
Llegados a este punto debemos determinar dos parámetros fundamentales en el método LK, el tamaño de la vecindad y el número de puntos a computar.

4.5.1.2.1 Tamaño de la vecindad

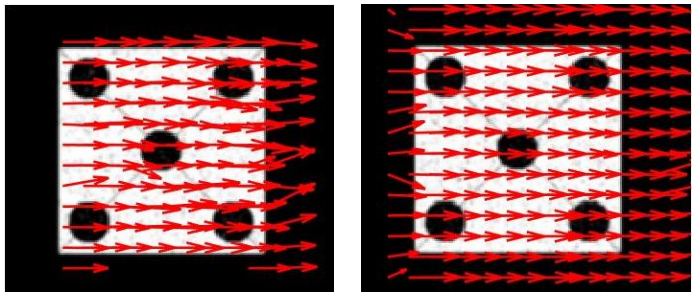
El cual determina el número de ecuaciones que se obtendrán en cada punto en el que se aplique el método, (Figura 4.3, ecuación 4-5). Si se utiliza una vecindad de 5x5 píxeles, estaríamos tratando con un sistema de 25 ecuaciones, si se utiliza una vecindad de 7x7 píxeles, el número de ecuaciones ascendería a 49, es decir una variación muy pequeña en el tamaño de la vecindad puede suponer un aumento en tiempo de cómputo considerable. Por otro lado la selección de un tamaño excesivamente grande puede desembocar en errores en los resultados, a continuación se mostrará un ejemplo de ello:

Ejemplo: Relación entre el tamaño de la ventana y los resultados

Supongamos que disponemos una secuencia en la que se observa la representación de un dado realizando un desplazamiento puramente horizontal. Comenzaremos variando el tamaño de la ventana y analizaremos el resultado obtenido:



3x3 5x5 7x7



10x10 45x45

Figura 4.10: Variación de los resultados en función del tamaño de la ventana

La utilización de una ventana 3x3 supone resolver 9 ecuaciones por pixel solamente, lo que conlleva un tiempo de procesamiento menor. A pesar de esto se puede observar que los resultados obtenidos se desvían de lo esperado puesto que los vectores no son fieles al movimiento real. Los tamaños 5x5, 7x7 y en concreto 10x10 arrojan buenos resultados a cambio de verse incrementado el número de ecuaciones del sistema. Si nos excedemos en el tamaño de la ventana la formulación comienza a fallar pues se localizan desplazamientos de píxeles que realmente no se están moviendo, en contrapartida al computar los resultados con más ecuaciones el resultado se asemeja más al movimiento que describe el dado. Este último caso se aprecia perfectamente para una ventana de 45x45. Por consiguiente será necesario utilizar una ventana de un tamaño tal que aporten unos resultados en el tiempo adecuado y con una precisión acorde a la aplicación que se quiera desarrollar.

4.5.1.2.2 Modo de selección de los píxeles.

Como ya se ha mencionado el método LK utiliza información derivada de un pixel determinado y de su vecindad. A la hora de seleccionar los píxeles a tratar debemos decidir cómo se va a realizar dicho proceso. En nuestro caso se va a crear una rejilla de puntos equidistantes separados por un número de pixeles determinados. A la hora de seleccionarlos solamente será necesario introducir el espacio entre píxeles que se desea.

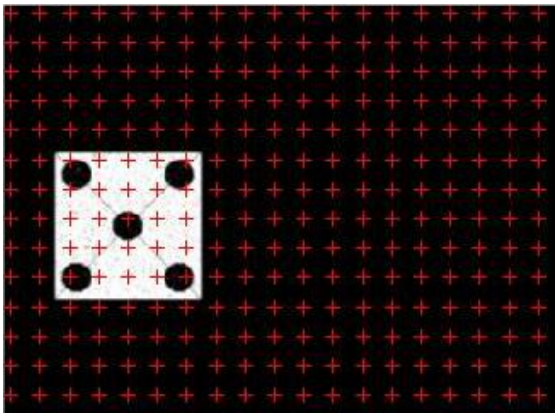
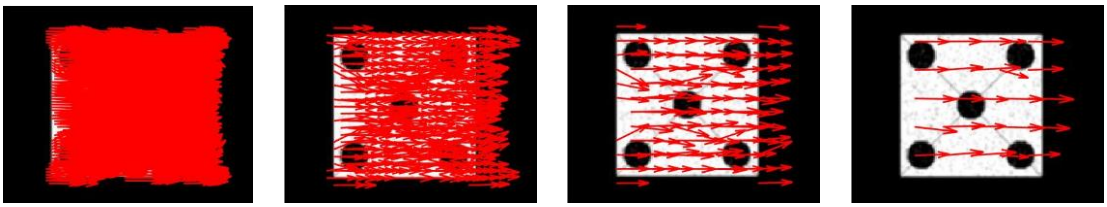


Figura 4.11: Rejilla de puntos equidistantes separados 20 pixeles entre si.

Es evidente que cuanto más pequeña sea la distancia entre píxeles mayor será el número de sistemas de ecuaciones que hay que resolver. Por otro lado si la representación gráfica no se realiza correctamente es posible que no se aprecien bien los resultados de manera visual debido a la alta densidad de información que contiene la imagen. Este efecto se ilustra en las siguientes imágenes.



Cada 3 pixeles Cada 5 pixeles Cada 10 pixeles Cada 20 pixeles

Figura 4.12: Resultados en función de la distancia entre píxeles.

4.5.1.3 Gradientes y derivada temporal:

En el apartado 4.4.2 se obtuvieron una serie de expresiones discretas a partir de la formulación continua para el cálculo del gradiente y derivada temporal.

$$\frac{\partial f(x,y)}{\partial x} \approx \Delta_x f(x,y) = f(x,y) - f(x-1,y) \quad (4\text{-}11)$$

$$\frac{\partial f(x,y)}{\partial y} \approx \Delta_y f(x,y) = f(x,y) - f(x,y-1) \quad (4\text{-}12)$$

$$\frac{dF(t)}{dt} \approx F(2) - F(1) \quad (4\text{-}13)$$

Estas expresiones son la base de varios algoritmos que resuelven el problema de la obtención de las derivadas espaciotemporales. En este apartado explicaremos las utilizadas en este proyecto.

4.5.1.3.1 Resolución iterativa.

Dicha implementación consiste simplemente en utilizar dos bucles que van a indexar la matriz correspondiente realizando las operaciones descritas en las ecuaciones (4-11) (4-12) y (4-13) para obtener los resultados deseados. El problema de esta opción es que al ser una operación centrada en dos píxeles, en el caso de ser una imagen con ruido, es posible que los resultados no sean correctos. Aun así este procedimiento se podría utilizar como primera aproximación, como ya se ha hecho en el ejemplo matricial.

4.5.1.3.2 Por convolución.

La convolución es una operación matemática muy utilizada en el tratamiento de imágenes. Consiste en aplicar un filtro a una matriz, en nuestro caso una matriz de intensidad, utilizando otra denominada "kernel".

Existen varios métodos diferentes para obtener el gradiente y cada uno utiliza un kernel diferente. Entre ellos podemos mencionar al operador sobel, al operador Prewitt, o incluso filtros derivados de las ecuaciones (4-11) y (4-12).

$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$S_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$P_x = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}$	$P_y = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}$
(a)	(b)	(c)	(d)

Figura 4.13: (a) Máscaras para la obtención del gradiente en el sentido de la x y de la y según el operador Sobel. (c-d) Mascaras para la obtención del grandiente en el sentido de la x y de la y según el operador Prewitt

Por comodidad y eficiencia se ha decidido utilizar la función imgradientxy de las bibliotecas de Matlab debido a los múltiples métodos para el cálculo del gradiente que dispone y por ser compatible con el cómputo paralelo.

Tras seleccionar como matriz de convolución el operador sobel e introduciendo la primera matriz de intensidad de nuestra secuencia de ejemplo obtendríamos los siguientes resultados:

```
[Ix_m,Iy_m]=imgradientxy(I);
```

Donde Ix_m corresponde al gradiente en sentido del eje x e Iy_m corresponde al gradiente en el sentido del eje y.

(a) (b)

Figura 4.14: Gradiente para la primera matriz de intensidad de la secuencia de dos imágenes

El Cálculo de la derivada temporal no es más que una resta matricial entre las dos matrices de intensidad. Para nuestro ejemplo de partida operaríamos con la imagen del dado en la posición inicial y la imagen una vez desplazado horizontalmente hacia la derecha.

```
It_m = J-I;
```

Donde It_m correspondo a la derivada temporal, I a la imagen inicial y J a la imagen del dado tras su movimiento.

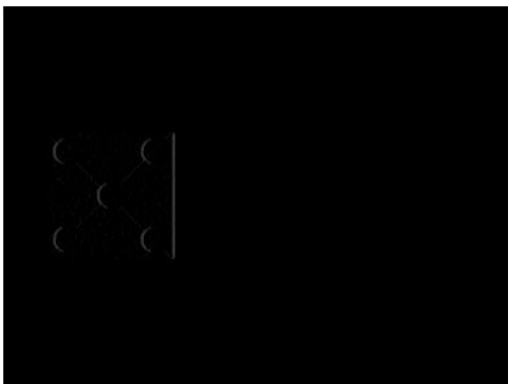


Figura 4.15: Derivada temporal, resta entre las dos imágenes de la secuencia.

4.5.1.4 Procesado de píxeles

Disponemos de todos los datos necesarios para aplicar las ecuaciones descritas en la sección 4.2, además de tener seleccionados los píxeles sobre los que se desea obtener el flujo óptico.

La operación a realizar es muy sencilla, recorrer todos los puntos de manera secuencial y resolver los sistemas de ecuaciones en mínimos cuadrados que se forman en cada uno de ellos para obtener la solución aproximada de la velocidad.

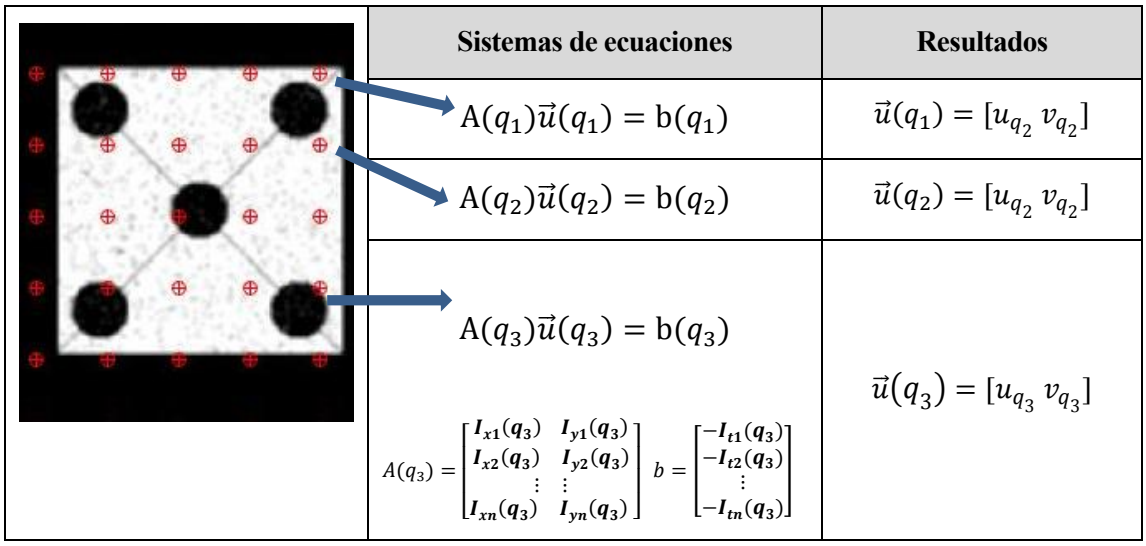


Figura 4.16: Diferentes sistemas de ecuaciones para distintos píxeles de la imagen.

4.5.1.5 Mostrar resultados

Como resultado final obtenemos los vectores de velocidad centrados en cada punto de la rejilla. Para representar dicho campo Matlab nos ofrece la posibilidad de utilizar el comando quiver.

```
quiver(x,y,u,v)
```

Donde x e y son las coordenadas de los píxeles seleccionados y u y v las componentes horizontal y vertical que hemos obtenido tras resolver los sistemas de ecuaciones por mínimos cuadrados.

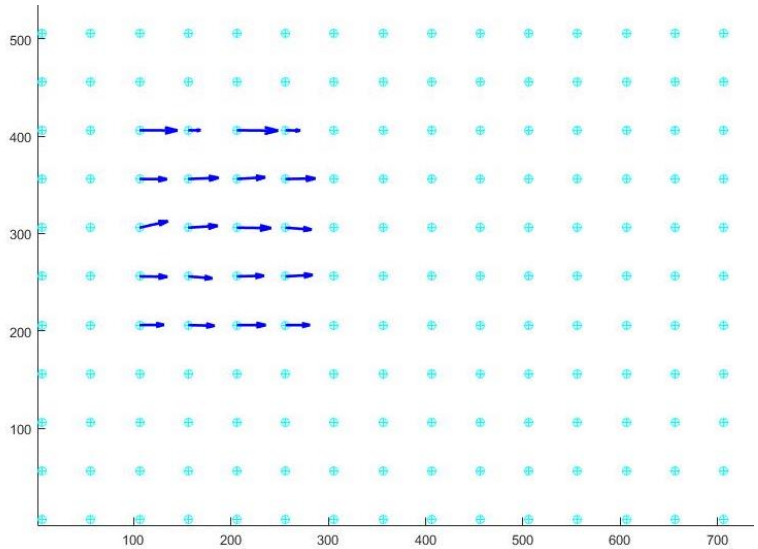


Figura 4.17: Representación de los resultados obtenida mediante el comando quiver.

4.5.2 Algoritmo en la GPU:

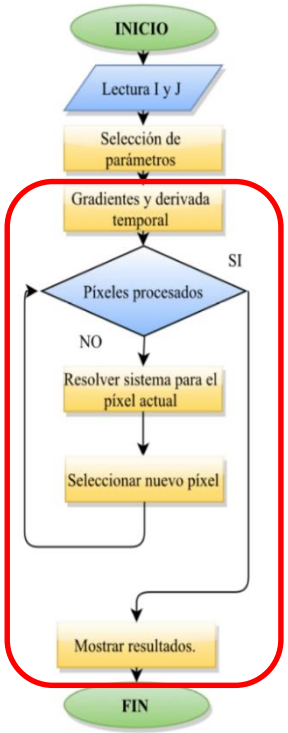
Como ya se explicó en el capítulo 2, la programación paralela consta de dos partes fundamentales, la que se diseña de cara a la ejecución en la CPU y la que se diseña de cara a la ejecución en la GPU. En la CPU encontraremos código que por su naturaleza no puede ser paralelizado o que cuya función es configurar la ejecución de los programas en la GPU. Por otro lado en la GPU encontraremos código destinado a ser ejecutado por los múltiples hilos de manera paralela. Por consiguiente podemos distinguir dos tareas fundamentales a llevar a cabo una vez diseñado e implementado el algoritmo secuencial en la CPU:

- Caracterización de las operaciones que son susceptibles de ser paralelizadas migrándolas a la GPU.
- Determinar qué operaciones se implementarán en Matlab y cuales en C.
- Incorporación de código encargado de la configuración de la GPU.

En los siguientes apartados nos centraremos en este aspecto y desarrollaremos las características más importantes del algoritmo en la GPU.

4.5.2.1 Caracterización de código susceptible a ser paralelizado.

Las entidades principales sobre las que estamos trabajando son imágenes, concretamente sobre los valores de intensidad de sus píxeles. Muchas de las operaciones que se están aplicando consisten en repetir ciertas modificaciones variando únicamente los datos de partida, ejecutamos el mismo código sobre distintos píxeles. Para la ejecución secuencial este proceso se vale de una serie de bucles que indexan los valores de las matrices, en el caso de la programación paralela podemos ejecutarlas de manera paralela. Por consiguiente la mayor parte del tratamiento de imágenes y la formulación referente al método LK se pueden migrar a la GPU.



Operaciones en la GPU
Obtención de la matriz de intensidad/conversión rgb-gris
Modificar tamaño
Obtención de los gradientes
Obtención de las derivadas
Aplicar formulación LK

Figura 4.18: Operaciones del algoritmo secuencial que pueden paralelizarse

4.5.2.2 Estructura del programa. MATLAB vs CUDA C

Una vez realizada la distinción sobre lo que se puede paralelizar y lo que no, es necesario analizar los recursos de los que disponemos para implementar el algoritmo. Por un lado Matlab nos ofrece un lenguaje sencillo con multitud de funciones ya programadas así como de las herramientas de los toolbox. Cuando Matlab no sea suficiente podemos crear funciones y programas en CUDA C que son perfectamente compatibles con el lenguaje anterior. Por otro lado algunas funciones de Matlab están preparadas para conmutar su ejecución y migrarla a la GPU. Conociendo todo lo anterior repartiremos las acciones entre los distintos lenguajes según la figura siguiente:

MATLAB		**CUDA C (GPU)**
CPU	GPU	
Lectura de imágenes	Cálculo de gradientes	Formulación CUDA C
Selección de parámetros	Cálculo de derivadas	
Mostrar resultados	Tratamiento de imagen	
Configuración de la ejecución del código de CUDA C		

Figura 4.19: Distribución de las tareas entre los distintos lenguajes de programación.

4.5.2.3 Descripción general:

El algoritmo y la implementación aquí descrita parte en su mayoría del algoritmo secuencial, es por eso que apartados como lectura, selección de parámetros, tratamiento de imágenes y cálculo de gradiente se mantienen prácticamente inalterables. Si es cierto que a pesar de ser muy parecidos es necesario configurar ciertos parámetros para indicar a las funciones que utilicen la GPU, siempre que estén diseñadas para dicha tarea.

Para el caso de las funciones programadas íntegramente en CUDA C, Figura 4.20, será necesario realizar una nueva implementación además de incorporar en MATLAB funciones relacionadas con la configuración de este nuevo tipo de código.

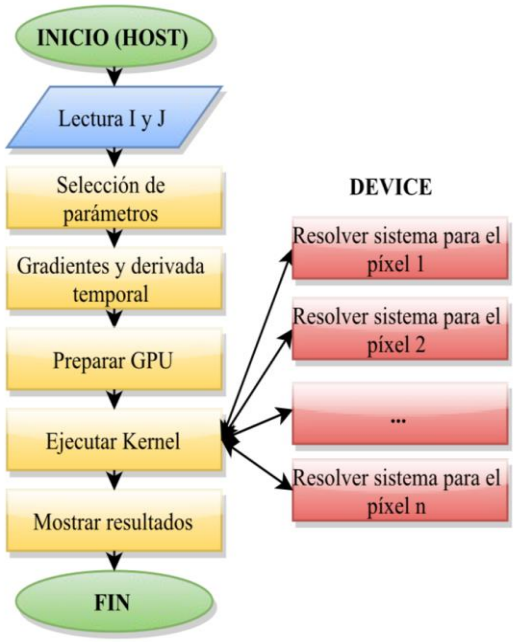


Figura 4.20: Diagrama del algoritmo (LK) orientado a la programación en la GPU.

4.5.2.3.1 Configuración de funciones de MATLAB para el cálculo paralelo.

Como se aprecia en la figura 4.18, las funciones que anteriormente ejecutábamos sobre la CPU pueden adaptarse fácilmente para la ejecución en la GPU. Por norma general esta adaptación es tan simple como cambiar el tipo del objeto que se introduce en la función. Si disponemos de una imagen "I" almacenada en la memoria de la CPU, utilizando el siguiente comando crearemos un objeto en la memoria de la GPU:

```
gpuarrayI=gpuArray(I);
```

Al utilizar el nuevo objeto *gpuarray* en las funciones siguientes habremos conmutado el funcionamiento de la CPU a la GPU. Es decir, si utilizamos un objeto almacenado en la memoria del host, la función se ejecutara en la CPU, si utilizamos un objeto almacenado en el device, la función se ejecutará en la GPU:

-Para el cálculo de los gradientes:
```
[gpuarrayGx,gpuarrayGy]=imgradientxy(gpuarrayI,__)
```

-Para la conversión de formato:
```
I = rgb2gray(gpuarrayRGB)
```

-Para el cambio de tamaño:
```
gpuarrayB = imresize(gpuarrayA,scale)
```

4.5.2.3.2 Configuración y ejecución de CUDA C desde MATLAB.

4.5.2.3.2.1 Configuración:

Como vamos a ejecutar programas en la GPU necesitamos configurar ciertos parámetros extra en comparación con el procedimiento en la CPU:

-Número de *threads*: Se seleccionarán tantos hilos como pixeles contenga la rejilla, siempre que nuestra GPU pueda procesarlos. En el caso de que el número de píxeles sea superior a la capacidad de cómputo paralelo se irán ejecutando por tandas. Recordando el apartado 3.3.8, el comando que nos permite realizar dicha función es parallel.gpu.CUDAKernel.

4.5.2.3.2.2 Ejecución de un Kernel:

Una vez creado el objeto asociado a nuestro kernel y configurado el número de hilos necesarios se ejecutan en la GPU mediante el comando feval. Al terminar dicho procedimiento podemos recuperar los datos utilizando el comando gather.

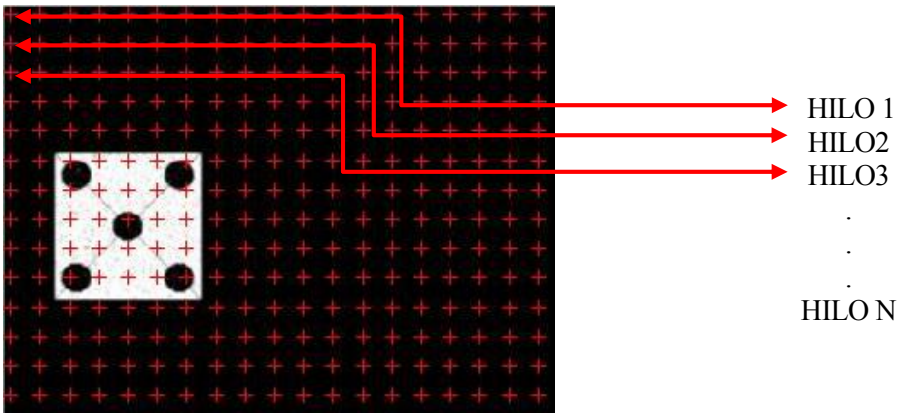


Figura 4.21: Cada hilo se encargará de ejecutar las funciones correspondientes a un punto de la rejilla

4.5.2.4 Funciones programadas en CUDA C

Aquí reside la diferencia principal entre el algoritmo secuencial y el paralelo. Mientras que el algoritmo en la CPU debe esperar la resolución de un pixel para continuar con el siguiente, en el algoritmo que utiliza la GPU se resuelven todos los sistemas de manera paralela, una vez se haya programado ciertos parámetros previos. Por consiguiente, El objetivo de cada hilo es realizar la misma operación que el algoritmo en la CPU realizaba en cada vuelta de bucle, resolver el sistema de ecuaciones para cada pixel.

La información que recibe cada hilo será la necesaria para la resolución del sistema de ecuaciones, tamaño de la vecindad, gradientes, derivadas y localización del pixel sobre el que se va a trabajar.

El Algoritmo 3 escrito en pseudocódigo resume a grandes rasgos la estructura que tendrá el programa una vez implementado en C.

Algoritmo 3: Resuelve_sistema_ecuaciones(I_x,I_y,I_t,w,q_i,q_j)

01- idx ⟵ OBTENER_IDENTIFICADOR_HILO();
02- Asociamos el pixel al hilo, obtención de la fila: f= q_i(idx);
03- Asociamos el pixel al hilo, obtención de la columna: c=q_j(idx);
04- **for** i= f-w/2 **to** f+w/2 **do**
05- **for** j=c-w/2 **to** c+w/2 **do**

Recorremos la vecindad del punto (f,c)

06- Ex=Ex+$I_x(i,j)^2$
07- Ey=Ey+ $I_y(i,j)^2$
08- Exy=Exy+ $I_x(i,j)$ $I_y(i,j)$;

Información referente a la matriz A:

$$A = \begin{bmatrix} Ex & Exy \\ Exy & Ey \end{bmatrix}$$

09- b_1= b_1+ $I_x(i,j)$ $I_t(i,j)$;
10- b_2= b_2+ $I_y(i,j)$ $I_t(i,j)$;

Vector b del sistema Ax=b

11- **end for**
12- **end for**
13- Resolución del sistema, donde x=desplazamiento: Ax=b.

4.6. Estimación iterativa, método de Kanade-Lucas-Tomashi triangular

4.6.1 Descripción general de lalgoritmo

Este método parte de las mismas hipótesis que el basado en el gradiente, es decir, las ecuaciones descritas son válidas y se utilizan de manera parecida. Por otro lado, la resolución del sistema y la implementación varía sustancialmente puesto que, mientras que en la propuesta anterior se utilizaba la resolución por mínimos cuadrados, en el algoritmo LKT triangular se realizan una serie de operaciones iterativas, tratamiento de imagen y selección de pixeles bien condicionados, que arrojan mejores resultados incluso para amplios desplazamientos entre los pixeles.
En este apartado vamos a explicar las características más importantes del algoritmo LKT triangular valiéndonos del diagrama de la figura 4.6.

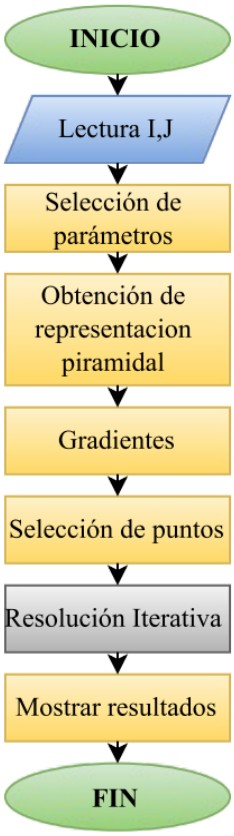


Figura 4.22: Diagrama del algoritmo (LKT) orientado a la programación en la CPU.

4.6.1.1 Lectura de las imágenes I y J

Obtención de imágenes y tratamientos asociados para calcular los niveles de intensidad de las imágenes. No hay cambios sustanciales con respecto al algoritmo LK.

4.6.1.2 Selección de parámetros

En este apartado comenzamos a observar diferencias entre el algoritmo LK y el algoritmo LKT triangular puesto que en el primero no se utilizaban los niveles piramidales ni la resolución iterativa. El tamaño de la vecindad se rige por las mismas normas expuestas en el método LK.
En resumen, los parámetros a seleccionar son los siguientes:

- Tamaño de la vecindad (w).
- Número de niveles piramidales a computar (L_m).
- Número máximo de iteraciones.
- Número de píxeles.

4.6.1.3 Representación piramidal de imágenes

La representación piramidal consiste en generar una colección de imágenes de distinta resolución partiendo de la original. Es un proceso iterativo en el que por cada ciclo se reduce la imagen ¼ (Figura 4.23) de su resolución inicial, donde cada pixel del nivel superior va a contener un valor igual al valor medio de una vecindad del nivel inferior.
El parámetro L_m contiene el nivel más alto que se quieren computar, considerando el nivel 0 como la imagen original y el nivel L_m la imagen más pequeña obtenida.

Figura 4.23: Representación piramidal de una imagen. I^3 es la imagen de menor resolución e I^0 imagen de mayor resolución (original).

El beneficio y la motivación del uso de la representación piramidal es ser capaces de detectar grandes desplazamientos de pixeles, mayores que la ventana (w). A pesar de esto si se calculan excesivos niveles es posible que el nivel más alto no sea procesable porque la ventana sea más grande que la imagen. L_m tiende a tomar valores comprendidos entre dos y tres en la práctica.

De entre los diferentes métodos que existen, Matlab nos ofrece la posibilidad de utilizar una representación piramidal gaussiana mediante la ejecución del comando impyramid.

4.6.1.4 Cálculo de los gradientes

Llegados a este punto, en el algoritmo LK, realizábamos el cálculo de los gradientes y derivadas temporales. En el caso del algoritmo LKT, realizamos el cálculo de los gradientes de igual manera pero sobre todos los niveles piramidales.
La obtención de la derivada temporal es relativamente diferente en el método LK. Al ser un proceso iterativo obtenemos un desplazamiento diferente en cada iteración, la derivada temporal se calcula realizando las diferencias entre la vecindad del punto inicial y la vecindad del punto calculado en la iteración actual.

4.6.1.5 Selección de puntos bien condicionados:

En el algoritmo LK seleccionábamos los píxeles utilizando una rejilla de puntos equidistantes. Según [1] Es más eficiente seleccionar una serie de puntos bien condicionados que probablemente arrojen unos resultados mejores. Matlab implementa dicho procedimiento en la función corner.

En la figura 4.24 podemos ver la comparativa entre el método de selección utilizado en el algoritmo LK y el utilizado en el algotirmo LKT. En rojo se aprecia la malla de puntos equidistantes y en azul los puntos pertenecientes a píxeles bien condicionados.

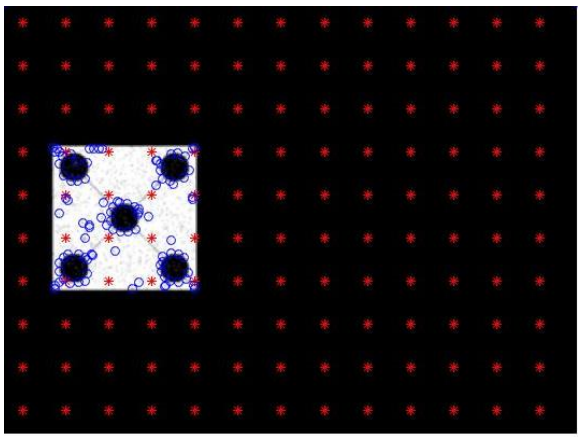


Figura 4.24: Puntos seleccionados por rejilla(rojos) y puntos bien condicionados (azul).

4.6.1.6 Interpolación

Para no perder precisión en cada iteración del método es necesario utilizar los valores correctos de intensidad de la imagen. Es posible que en el proceso iterativo sea necesario operar con algún valor de intensidad que esté comprendido entre dos píxeles, puesto que es una entidad discreta, en el caso de tener que medir la intensidad de un punto situado entre píxeles, vamos a utilizar la interpolación bilineal.

Las ecuaciones necesarias para realizar dicho procedimiento son las siguientes:

$$x = xo + \alpha_x \quad y = yo + \alpha_y$$

$$I^L(x,y) = (1-\alpha_x)(1-\alpha_y)I^L(xo,yo) + \alpha_x(1-\alpha_y)I^L(xo+1,yo) + (1-\alpha_x)\alpha_y IL(xo,yo+1) + \alpha_x\alpha_y I^L(xo+1,yo+1).$$

Por otro lado Matlab nos ofrece el comando interp2.

4.6.1.7 Resolución iterativa

En este punto estamos en disposición de comenzar la resolución de los sistemas de ecuaciones. El objetivo principal del algoritmo es la resolución de los sistemas de ecuaciones asociados a los píxeles de manera iterativa sobre los distintos niveles piramidales de la imagen, de tal manera que la información que se obtiene de cada nivel se utiliza en el siguiente.

Algoritmo 4: Resolución iterativa del sistema de ecuaciones Ax=b

01- Inicialización de la estimación inicial: $g^{Lm} = [g_x^{Lm} \; g_y^{Lm}]^T = [0\;0]^T$

02- **for** L= L_m **to** 0 **do**

03- Situamos los puntos en nivel actual L: $u^L = [q_x \; q_y]^T = u/2^L$

04- Calculamos o extraemos el gradiente correspondiente al nivel L.

05- Construimos la matriz de gradientes espaciales: $A = \begin{bmatrix} \sum_i \boldsymbol{I_x(q_i)^2} & \sum_i \boldsymbol{I_x(q_i)I_y(q_i)} \\ \sum_i \boldsymbol{I_x(q_i)I_y(q_i)} & \sum_i \boldsymbol{I_y(q_i)^2} \end{bmatrix}$

06- Inicializamos la variable iterativa: $\vec{v}^0 = [0\;0]^T$

07- **for** k= 1 **to** Maximas_iteraciones **do**

08- Calculamos la derivada temporal: $\boldsymbol{I_t = I^L(x,y) - J^L(x + g_x^L + v_x^{k-1}, y + g_y^L + v_y^{k-1})}$

09- Construimos el vector b: $b_k = \begin{bmatrix} -\sum_i \boldsymbol{I_x I_t} \\ -\sum_i \boldsymbol{I_y I_t} \end{bmatrix}$

10- Obtención del flujo óptico (LK) $\vec{x} = A^{-1}b_k$

11- Actualizamos el valor de v: $\vec{v}^k = \vec{v}^{k-1} + \vec{x}^k$

12- **Endfor K**

13- Flujo óptico final para el nivel L: $d^L = \vec{v}^k$

14- Guess for next level L-1: $g^{L-1} = [g_x^{L-1} \; g_y^{L-1}]^T = 2(g^L + d^L)$

15- **Endfor L**

16- Vector de flujo óptico: $d = g^0 + d^0$

Puesto que el procedimiento de resolución es análogo para todos los píxeles, vamos a aplicarlo solamente a un punto.

En primer lugar inicializamos el valor del flujo óptico a cero.

$$g^{Lm} = [g_x^{Lm} \; g_y^{Lm}]^T = [0\;0]^T$$

Donde g^{Lm} es el valor del flujo óptico en el nivel mayor que corresponde con la imagen de menor resolución.

El orden en el que se procesan los niveles piramidales es de menor resolución a mayor resolución. Como los puntos bien condicionados se calculan haciendo uso de la imagen de mayor resolución es necesario redimensionar las posiciones para localizarlos en el nivel actual correspondiente. Esta operación se realiza utilizando la fórmula:

$$u^L = \frac{u}{2^L}$$

Donde $u = (q_x, q_y)$ es la posición del pixel en el nivel 0, y u^L es la posición en el nivel actual L.
Una vez situado el punto en el nivel correspondiente construimos la matriz A.

Ya disponemos de la información necesaria para realizar el proceso iterativo. Al igual que en el algoritmo LK, el objetivo final es resolver el sistema Ax=b con la particularidad de que b va variando y tendiendo a unos resultados óptimos. Dicha variación se debe a que el cálculo de la derivada temporal depende del resultado que arroja la iteración anterior y de la estimación realizada en el nivel anterior:

$$\boldsymbol{I_t = I^L(x,y) - J^L\left(x + g_x^L + v_x^{k-1}, y + g_y^L + v_y^{k-1}\right)}$$

Como se observa, el valor de J^L está en función de variables que no tienen por que ser enteros. Este hecho nos obliga a utilizar la interpolación bilineal para obtener resultados más precisos.

Una vez determinado el vector b, se resuelve el sistema de ecuaciones y se actualizan para la siguiente iteración.
Al finalizar el proceso iterativo se actualiza el valor de flujo óptico estimado para su utilización en el siguiente nivel piramidal. Al igual que se hizo con los puntos bien condicionas es necesario redimensionar las posiciones para que sean válidas en el análisis de la imagen del nivel inferior. La ecuación utilizada para realizar este procediendo es:

$$g^{L-1} = [g_x^{L-1} \;\; g_y^{L-1}]^T = 2(g^L + d^L)$$

Una vez procesadas todos los niveles piramidales, llegamos a la obtención del vector de flujo óptico.

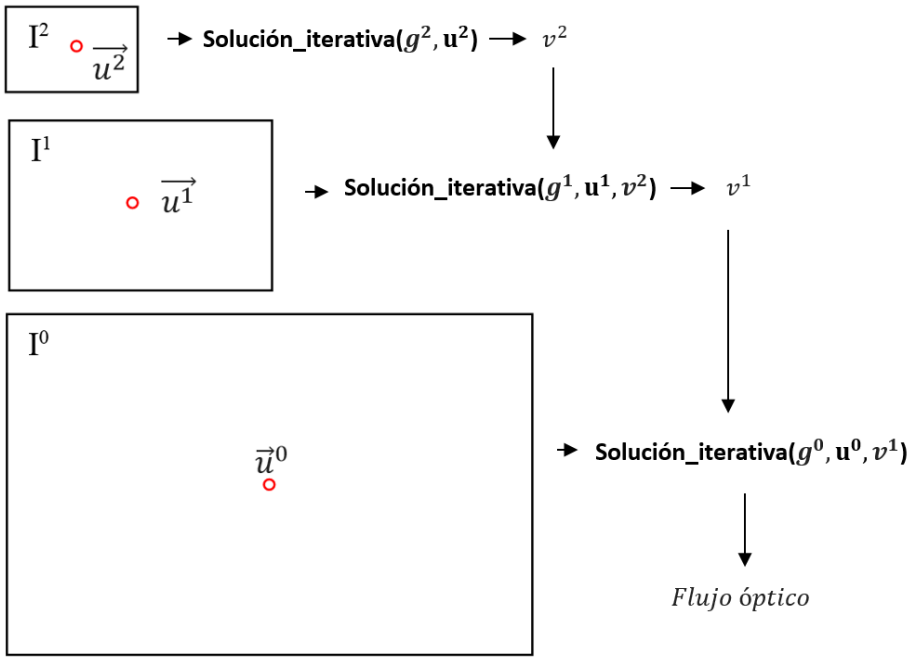


Figura 4.25: Diagrama explicativo del método LKT triangular

4.6.1.8 Mostrar resultados:

Al igual que en el algoritmo LK, los resultados se mostraran haciendo uso de la función quiver que nos muestra el campo de velocidades asociado a los puntos seleccionados al comienzo del proceso.

4.6.1.9 Algoritmo en la CPU

El algoritmo implementado en la CPU cumple íntegramente los explicado en el apartado 4.3.1.

4.6.1.10 Algoritmo en la GPU

En el caso del algoritmo paralelizado vamos a migrar cierta parte del procesamiento a la GPU. Al igual que la paralización del algoritmo LK, vamos a ejecutar un número determinado de hilos que van a encargarse de resolver los sistemas de ecuaciones, con la particularidad de que en el algoritmo LKT hay que realizar esta operación para cada nivel piramidal. Haciendo alusión al algoritmo 4, la programación en el device debe encargarse de las líneas comprendidas entre la 03-014.

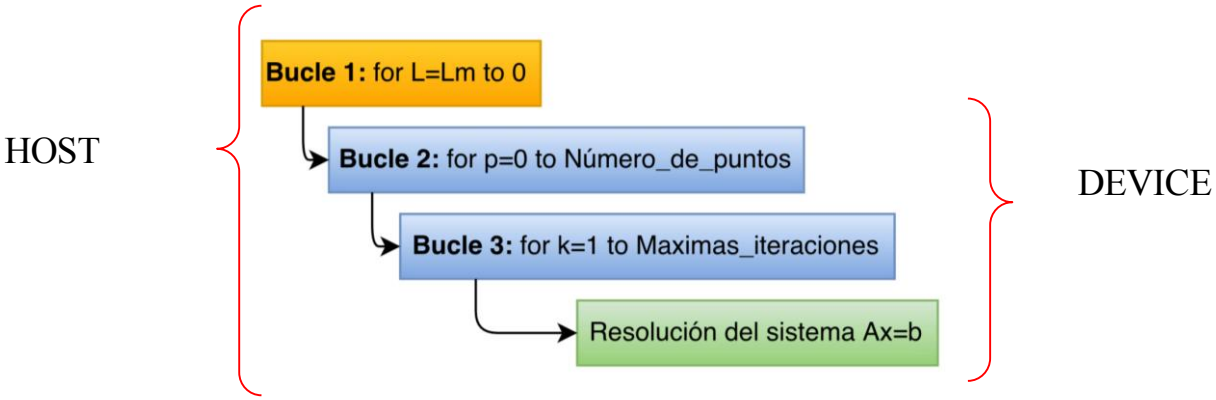


Figura 4.26: Estructura de los bucles en la CPU/GPU

5 PROYECTO INFORMÁTICO

En el proyecto informático se van a tratar aspectos referentes directamente al código y a la propia implementación de los algoritmos. Así mismo, se va a sintetizar toda la información con el fin de llegar a comprender todo lo que se ha ido exponiendo a lo largo del trabajo.

La estructura de este capítulo consta de dos partes fundamentales correspondientes al algoritmo LK y al algoritmo LKT. Dentro de cada parte podemos acceder al desarrollo de las dos implementaciones que se han estudiado, una puramente secuencia basada en la programación en la CPU y otra basada en la utilización de CUDA C para el cálculo paralelo en la GPU.

Como ya se ha mencionado, la práctica totalidad de los programas se han construido utilizando MATLAB como lenguaje principal, excluyendo el código ejecutado en la GPU en el que se ha utilizado CUDA C.

5.1 Método Lucas-Kanade

5.1.1 Diagrama de flujo

En el capítulo 4 se desarrollaron las características principales del algoritmo LK tanto para la implementación secuencial como paralela y, en el capítulo 3, se explicaron las características más importantes de la programación tanto en Matlab como en CUDA C. Por otro lado, A nivel de código, es necesario repartir las operaciones para diseñar un programa legible y bien estructurado. Podemos dividir el proceso en tres partes bien diferenciadas, Figura 5.1, inicialización, procesado de píxeles y muestra o almacenamiento de los resultados. En ambas implementaciones optaremos por realizar dos funciones, la función principal LK_main y la función encargada de aplicar el método correspondiente, LK_CPU/GPU:

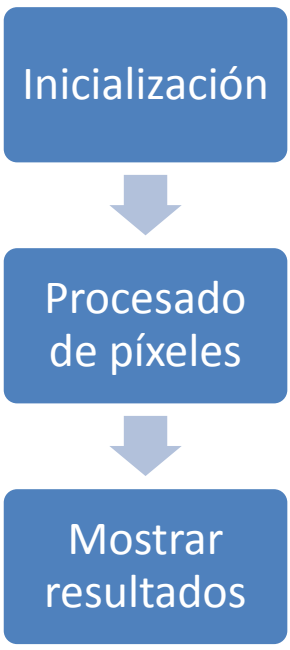


Figura 5.1: Diagrama general del algoritmo LK

La función LK_main deberá realizar:

- Lectura de imágenes ya sea de manera independiente o extrayéndolas de una secuencia mayor.
- Tratamiento de imagen y obtención de los valores de intensidad
- Selección e inicialización de parámetros como tamaño de la vecindad o número de píxeles a procesar

La función LK_CPU LK_GPU deberá llevar a cabo el método de Lucas-Kanade descrito en la sección 4.2.

- LK_CPU se utilizará para el procesamiento completamente secuencial mientras
- LK_GPU se utilizarpa para el procesamiento combinado con la GPU.

5.1.2 Jerarquía de funciones

En este apartado describiremos la jerarquía de ejecución de las funciones más importantes para el caso secuencial y paralelizado así como las características y diferencias más llamativas.

PROGRAMA SECUENCIAL	PROGRAMA PARALELIZADO
1. LK_main: a. [u,v] = LK_CPU(im1,im2,ww,'grid');	1. LK_main: a. [u,v] = LK_GPU(im1,im2,ww,'grid'); i. lk_kernel.cu
1.LK_main: • Encargada de iniciar las variables, cargar imágenes y mostrar resultados. • Llama a la función LK_CPU la cual devuelve el valor del flujo óptico calculado en la CPU. a.LK_CPU: • LK_main invoca esta función y le entrega las variables: -im1/im2: Valores de intensidad de las imágenes. -ww: ancho de la ventana o vecindad. -Parámetro referente a la selección de los puntos.	1.LK_main: • Encargada de iniciar las variables, cargar imágenes y mostrar resultados. • Llama a la función LK_GPU la cual devuelve el valor del flujo óptico utilizando una llamada a un kernel en la GPU. a.LK_GPU: • Invoca las mismas variables que el proceso secuencial. • Configura y ejecuta un kernel, lk_kernel.cu, que procesará la información en la GPU. i.Lk_kernel.cu: • Kernel ejecutado en la GPU. • A diferencia de las otras funciones se programa en CUDA C.

5.1.3 Código y descripción del programa

5.1.3.1 Programa íntegramente en la CPU:

Código del programa basado en la programación en MATLAB tradicional. Comprende las funciones LK_main, LK_CPU. El programa lee dos imágenes de una secuencia y realiza todo el procesamiento pertinente para obtener el flujo óptico.

5.1.3.1.1 LK_main

Descripción y código del programa principal.

Codigo 5.1: Algoritmo LK íntegramente programado en la CPU

Inicialización del entorno

```
clear all      %Borramos las variables existentes
clc            %Limpiamos el entorno
warning('off') %Cancelamos posibles avisos que ralenticen la ejecución
```

Cargamos las imágenes

```
load AVION; %En este caso abrimos las imágenes ya almacenadas

%%Procedimiento alternativo:
%fr1=imread('ima1.jpg'); %Almacenamos en fr1 la imagen de nombre 'ima1.jpg'
%fr2=imread('ima2.jpg'); %Almacenamos en fr2 la imagen de nombre 'ima2.jpg'
```

Tratamiento de la imagen para obtener las intensidades

```
r=3;                          %Factor para reducir o ampliar la imagen un 1/r
I = im2double(rgb2gray(fr1)); %Conversion de la imagen 1 a gris
im1=imresize(I, 1/r);         %Reducción de I un 1/r
J = im2double(rgb2gray(fr2)); %Conversion de la imagen 2 a gris
im2=imresize(J, 1/r);         %Reducción de J un 1/r
ww=7;                         %Tamaño de la ventana
```

Comienza el analisis

```
[u,v] = LK_CPU(im1,im2,ww,'grid'); %Ejecutamos el algoritmo Lucas-Kanade
```

Mostrar resultados

```
imshow(im2);        %Muestra la imagen
hold on;
quiver(u,v,50,'r')  %Muestra el campo de velocidades.
```

5.1.3.1.2 LK_CPU

Descripción y código de la función LK_CPU.

Codigo 5.2: Función LK_CPU programado en la CPU

```
function [u,v] = LK CPU(I,J,ww,selector,W)
%Algoritmo para obtener el flujo optico mediante el método de Luca-Kanade
%   I es la 1ª imagen de la secuencia
%   J es la 2ª imagen de la secuencia
%   w es el tamaño de la ventana
%   W ponderaciones de cada pixel.
%   [u,v] matrices donde almacenamos las componentes de velocidad
```

Selección del método a utilizar para la selección de los píxeles

```
if nargin==4
    W=eye(ww^2); %En el caso de no utilizar ponderaciones W=identidad.
end
if strcmp(selector,'grid')==1 || strcmp(selector,'deci')==1
    n=10;                   %Seleccióna el punto cada 10 píxeles
if strcmp(selector,'deci')==1
    n=round(size(I,1)/10); %Divide automáticamente en una rejilla de 10 elementos
end
```

Cálculo de los gradientes y derivada temporal mediante convolución

```
h = fspecial('sobel');              %h=mascara para obtener los gradientes
Ix_m = imfilter(I,h','replicate');  %Realiza la convolución y obtiene Ix
Iy_m = imfilter(J,h,'replicate');   %Realiza la convolución y obtiene Iy
It_m =  I-J;                        %Derivada temporal
u = zeros(size(I));                 %Matriz que almacenaremos los valores de u
v = zeros(size(J));                 %Matriz que almacenaremos los valores de v
w=floor(ww/2);                      %Radio de la vecindad
```

Bucle que recorre los píxeles de la imagen y resolución del sistema de ecuaciones

```
for i = w+1:n:size(Ix_m,1)-w-n
    for j = w+1:n:size(Ix m,2)-w-n
        Ix = Ix m(i-w:i+w, j-w:j+w); %Extracción de la vecindad para el punto (i,j)
        Iy = Iy m(i-w:i+w, j-w:j+w);
        It = It m(i-w:i+w, j-w:j+w);
        Ix=Ix';
        Iy=Iy';
        It=It';
        Ix = Ix(:);             %Convertimos Ix,Iy e It en vectores columna
        Iy = Iy(:);
        It = -It(:);
        A=[Ix Iy];               %Formamos la matriz A
        nu=(A'*W*A)\(A'*W*It);  %Resolvemos el sistema
        %Eliminamos valores que puedan dar resultados mal determinados
        a=eig(A'*W*A);          %Cálculo de los autobalores
        if(a(1)>0 && a(2)>0)    %Si autovalores>0
            u(i,j)=nu(1);       %guardamos la velocidad
            v(i,j)=nu(2);
```

```
        else
            u(i,j)=0;
            v(i,j)=0;
        end
    end
end
    %Eliminamos los terminos NaN
    u(isnan(u))=0; v(isnan(v))=0; %isnan=1 si el valor es NaN
    end
end
```

5.1.3.2 Programa CPU-GPU

Código del programa basado en la programación en MATLAB y CUDA C. Comprende las funciones LK_main, LK_GPU, lk_kernel.cu y otras funciones auxiliares.

El programa lee dos imágenes de una secuencia y realiza todo el procesamiento pertinente para obtener el flujo óptico.

5.1.3.2.1 LK_main

Descripción y código del programa principal.

Código 5.3: Programa LK para la programación paralela

Inicialización del entorno

```
clear all      %Borramos las variables existentes
clc            %Limpiamos el entorno
warning('off') %Cancelamos posibles avisos que ralenticen la ejecución
```

Cargamos las imágenes

```
load workspace; %En este caso abrimos las imágenes ya almacenadas

%%Procedimiento alternativo:
%fr1=imread('ima1.jpg'); %Almacenamos en fr1 la imagen de nombre 'ima1.jpg'
%fr2=imread('ima2.jpg'); %Almacenamos en fr2 la imagen de nombre 'ima2.jpg'
```

Tratamiento de la imagen para obtener las intensidades

```
r=1;                                  %Factor que reducir la imagen un 1/r
I1 = im2double(rgb2gray(gpuArray(fr1)));%Conversion de la imagen a gris
im1=imresize(I1, 1/r);                 %Reducción de I un 1/r
I2 = im2double(rgb2gray(gpuArray(fr2)));%Conversion de la imagen 2 a gris
im2=imresize(I2, 1/r);                 %Reducción de J un 1/r
ww=7;                                 %Tamaño de la ventana
```

Comienza el analisis

```
[u,v] = LK_GPU(im1,im2,ww,'grid'); %Ejecutamos el algoritmo Lucas-Kanade
```

Mostrar resultados

```
imshow(im2);
hold on;
quiver(u,v,30,'r')
```

5.1.3.2.2 LK_GPU

Descripción y código de la función LK_GPU.

Código 5.4: Función LK_GPU para la programación paralela

```
function [u,v] = LK_GPU(I,J,ww,selector,W)
%Algoritmo para obtener el flujo optico mediante el método de Lucas-Kanade
%Algoritmo para obtener el flujo optico mediante el método de Lucas-Kanade
%   I es la 1ª imagen de la secuencia
%   J es la 2ª imagen de la secuencia
%   w es el tamaño de la ventana
%   W ponderaciones de cada pixel.
%   [u,v] matrices donde almacenamos las componentes de velocidadth=.01;
```

Selección del método a utilizar para la selección de píxeles

```
if nargin==4
    W=eye(ww^2);
end
if strcmp(selector,'grid')==1 || strcmp(selector,'deci')==1
    n=10;                   %Seleccióna el punto cada 10 píxeles
if strcmp(selector,'deci')==1
    n=round(size(I,1)/10); %Divide automáticamente en una rejilla de 10 elementos
end
```

Cálculo de los gradientes y derivada temporal

```
h = fspecial('sobel');                          %h=mascara para obtener los gradientes
Ix = im2double(imfilter(gpuArray(I),h','replicate'));%Convolución, obtención de Ix
Iy = im2double(imfilter(gpuArray(I),h,'replicate')); %Convolución, obtención de Ix
It=im2double(J-I);                                %Derivada temporal
%Inicializamos el campo de velocidades a cero
u=zeros(size(Ix));      %Matriz donde almacenaremos los valores de u
v=zeros(size(Ix));      %Matriz donde almacenaremos los valores de v
w=floor(ww/2);          %Radio de la vecindad
```

Inicializamos el objeto asociado al kernel programado en CUDA C

```
k = parallel.gpu.CUDAKernel('lk_kernel.ptx','lk_kernel.cu','indx');
```

Selección de pixeles mediante una rejilla

```
[X,Y]=meshgrid(1:n:size(I,1),1:n:size(I,2));
x=X(:);         %Los convertimos en un vector por comodidad
y=Y(:);
tam=size(x,1);  %número de pixeles totales que se van a computar
nr=tam/1024;    %relación (N° de pixeles/N° máximo hios)
```

Ejecución del kernel, configuración de los Hilos

```
if(nr<1)        %(N° de pixeles/N° máximo hios)<1, < 1024 hilos, 1 sola ejecución
    k.ThreadBlockSize = [tam,1,1]; %N° de hilos a ejecutar
    [u,v] = feval(k,u,v,Ix,Iy,It,x,y,size(I,1),w); %Ejecutamos 1024 hilos
else            %(N° de pixeles/N° máximo hios)>1, ejecutar por tandas
    ll=[1 1025:1025:tam];
    hl=[1024 2049:1025:tam];
    for l=1:floor(nr);
    k.ThreadBlockSize = [1024,1,1];
    [u,v] = feval(k,u,v,Ix,Iy,It,x(ll(l):hl(l)),y(ll(l):hl(l)),size(I,1),w);
    end
    k.ThreadBlockSize = [numel(x(hl(l):end)),1,1];
    [u,v] = feval(k,u,v,Ix,Iy,It,x(hl(l):end),y(hl(l):end),size(I,1),w);
end

%Eliminamos los terminos NaN
 u(isnan(u))=0;
 v(isnan(v))=0;
end
```

5.1.3.2.3 Kernel

Código del kernel programado en CUDA C

Código 5.5: kernel en C ejecutado por LK_GPU

```
__global__  void  indx(double  *  u,double  *  v,double  *Ix,double*Iy,double  *It,double
*fil,double *col,int N,int w)
{
     int idx = blockDim.x*blockIdx.x+threadIdx.x; //N° id del hilo
     int f; //fila del píxel
     int c; //columna del píxel
     int i,j; //variables auxiliares para indexar matrices
      //VARIABLES PARA LOS SUMATORIOS, inicializamos a cero//
     double Gx=0;
     double Gy=0;
     double Gt=0;
     double det=0;
     double gxx,gxy,gyy,ey,ex;
     gxx=0;
     gyy=0;
     gxy=0;
     ex=0;
     ey=0;
     //Asignamos al hilo(idx) la posición del pixel(idx)
     f=fil[idx];
     c=col[idx];
     for(i=0;i<w;i++) //Montamos el sistema Ax=b
     {
        for(j=0;j<w;j++)
        {
        Gx=*(Ix+(f-1+i)*N+(c-1+j));
        Gy=*(Iy+(f-1+i)*N+(c-1+j));
        Gt=*(It+(f-1+i)*N+(c-1+j));
        gxx=gxx+Gx*Gx;
        gyy=gyy+Gy*Gy;
        gxy=gxy+Gx*Gy;
        ex=Gx*Gt+ex;
        ey=Gy*Gt+ey;
        }
```

```
        }
        det=gxx*gyy-gxy*gxy; //Determinante de la matriz A
        u[f*N+c] = (gyy*ex - gxy*ey)/det; //valor de desplazamiento horizontal
        v[f*N+c]=(gxx*ey - gxy*ex)/det;    //valor de esplazamiento vertical
}
```

5.1.4 Método de empleo

Para utilizar la función LK_CPU y LK_GPU solamente es necesario disponer de:

[u,v] = LK_xPU(I,J,ww,selector,W)

-I y J: funciones de intensidad de las imágenes de la secuencia almacenadas como matrices.

-ww: Ancho de la ventana de integración o de la vecindad.

-selector: Selector puede tomar dos variables 'deci', para realizar una selección de puntos cada 10 píxeles, o 'grid' seleccionando los puntos cada n píxeles donde n es un parámetro configurable.

-W: W factores para realizar el método ponderado.

La captura de las imágenes podemos realizarla de diversas maneras:

- Cargándolas desde un archivo preconfigurado,
- Cargandolas desde una ubicación en el ordenador mediante el comando imread('nombre.jpg').
- Cargando un video como variable del entorno de Matlab y extrayendo la secuencia deseada.

5.2 Metodo Lucas-Kanade-Tomashi

5.2.1 Diagrama de flujo

En el capítulo 4 se desarrollaron las características principales del algoritmo LKT tanto para la implementación secuencial como paralela y, en el capítulo 3, se explicaron las características más importantes de la programación tanto en Matlab como en CUDA C. Por otro lado, A nivel de código, es necesario repartir las operaciones para diseñar un programa legible y bien estructurado. Podemos dividir el proceso en tres partes bien diferenciadas, Figura 5.1, inicialización, procesado de píxeles y muestra o almacenamiento de los resultados. En ambas implementaciones optaremos por realizar dos funciones, la función principal LKT_main y la función encargada de aplicar el método correspondiente, LKT_CPU/GPU:

5.2.2 Jerarquía de

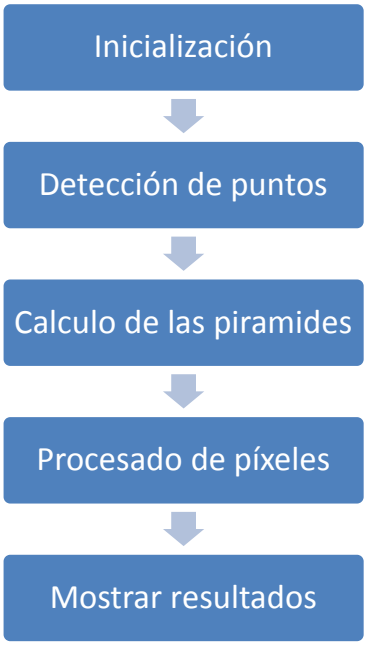


Figura 5.2: Diagrama general del algoritmo LKT

La función LKT_main deberá realizar:

- Lectura de imágenes ya sea de manera independiente o extrayéndolas de una secuencia mayor.
- Tratamiento de imagen y obtención de los valores de intensidad
- Selección e inicialización de parámetros como tamaño de la vecindad o número de píxeles a procesar
- Búsqueda de los puntos bien condicionados

La función LKT_CPU LKT_GPU deberá realizar:

- Calculo de las representaciones piramidales.
- Cálculo de los gradientes y derivadas temporal.

funciones

En este apartado describiremos la jerarquía de ejecución de las funciones más importantes para el caso secuencial y paralelizado así como las características y diferencias más llamativas.

PROGRAMA SECUENCIAL	PROGRAMA PARALELIZADO
1. LKT_main: a. LKT_CPU(fr1,fr2,Lm,X1,Y1,th,winR,maxIter)	1. LKT_main: a. LKT_GPU(fr1,fr2,Lm,C,w,maxIter,k) i. LKT_kernel.cu 1. rejilla 2. interp
1.LKT_main: • Encargada de iniciar las variables, cargar imágenes y mostrar resultados. • Llama a la función LK_CPU la cual devuelve el valor del flujo óptico calculado en la CPU. a.LKT_CPU: • LKT_main invoca esta función y le entrega las variables: -fr1/fr2: Imágenes de la secuencia. -Lm: parámetro del proceso iterativo. -winR: Radio de la ventana de integración o vecindad -maxIter: número de iteraciones máximas. -X1,Y1: posición de los puntos en la primera imagen de la secuencia	1.LKT_main: • Encargada de iniciar las variables, cargar imágenes y mostrar resultados. • Llama a la función LK_GPU la cual devuelve el valor del flujo óptico utilizando una llamada a un kernel en la GPU. a.LKT_GPU: • Invoca las mismas variables que el proceso secuencial. • Configura y ejecuta un kernel, lk_kernel.cu, que procesará la información en la GPU. i.lkt_kernel.cu: • Kernel ejecutado en la GPU. • A diferencia de las otras funciones se programa en CUDA C. • Se vale de dos funciones programadas en C: -interp: Interpolación bilineal. -rejilla: Selecciona puntos de la vecindad de un píxel.

5.2.3 Código y descripción del programa

5.2.3.1 Programa íntegramente en la CPU:

5.2.3.1.1 LKT_main

Descripción y código del programa principal.

Código 5.6: programa LKT íntegramente programado en la CPU

Inicialización del entorno

```
clear all        %Eliminamos las variables anteriores
clc              %Limpiamos el entorno
reset(gpuDevice);%Reseteamos la GPU
warning('off');
```

Cargamos las imágenes

```
load PAR;
%%Procedimiento alternativo:
%fr1=imread('ima1.jpg'); %Almacenamos en fr1 la imagen de nombre 'ima1.jpg'
%fr2=imread('ima2.jpg'); %Almacenamos en fr2 la imagen de nombre 'ima2.jpg'
```

Parámetros

```
w=11;        %Tamaño de la vecindad
th=.01;
Lm=3;        %Numero de niveles
maxIter=50;  %Número de iteraciones
N_puntos=200;%Número de puntos máximo bien condicionaods
```

Puntos bien condicionados, método shi-Tomasi

```
[C] = corner(rgb2gray(fr1),'MinimumEigenvalue',N_puntos);
X1=C(:,1);
Y1=C(:,2);
```

Comienzo del proceso LKT

```
[X2,Y2,d] = LKT_CPU(fr1,fr2,Lm,X1,Y1,th,w,maxIter);
```

Mostramos los resultados

```
u=X2-X1;
v=Y2-Y1;
imshow(fr1) %Mostramos la imgágen
hold on
plot(X1,Y1,'b*');   %Mostramos los puntos
[u,v]=arrayfun(@flechas,X2,Y2,u,v); %Mostramos el campo
plot(X2,Y2,'go')    %Mostramos los puntos
```

5.2.3.1.2 LKT_CPU

Descripción y código de la función LKT_CPU.

Código 5.7: función LKT_CPU programada para la ejecución en la CPU

```
function [X2,Y2,d] = LKT_CPU(fr1,fr2,Lm,X1,Y1,th,winR,maxIter)
%Algoritmo para calcular el flujo óptico mediante LKT piramidal
%   +Npuntos: Numero de puntos donde se calculará el flujo optico
%   +Lm: Numero de niveles piramidales
%   +Icell: Celda donde se almacenan todos los niveles piramidales 1ª imagen
%   +Jcell: Celda donde se almacenan todos los niveles piramidales 2ª imagen
%   +X2: desplazamiento del punto en la 2ª imagen
%   +Y2  ""
%   +th: valor
%   +winR: tamaño de la ventana a muestrear
%   +maxIter: maximo número de iteraciones
```

Tratamiento de imágenes y cálculo piramidal

```
img1 = im2double(rgb2gray(fr1));  %Conversion de la imagen a gris
img2 = im2double(rgb2gray(fr2));
Icell = pyr(img1,Lm);
Jcell = pyr(img2,Lm);
```

Parámetros del método

```
X2 = X1/2^Lm;   %Localizamos los puntos en el nivel mas alto
Y2 = Y1/2^Lm;   %Localizamos los puntos en el nivel mas alto
Npuntos=size(X2,1);   %Variable que almacena el n° de puntos
h = fspecial('sobel');%Cálculo del operador sobel
d=zeros(Npuntos,2);   %Inicializamos el vector desplazamiento
```

Inicio de las iteraciones

```
for level = Lm:-1:1

    %Seleccionamos las imagenes piramidales si Lm=3--> 2 1 0
    img1 = Icell{level}; %Rescatamos la imágen piramidal del nivel "level"
    img2 = Jcell{level}; %Rescatamos la imágen piramidal del nivel "level"
    [M N] = size(img1);  %Tamaño de la imagen img1

    %Calculamos los gradientes mediante el operador sobel
    img1x = imfilter(img1,h','replicate');
    img1y = imfilter(img1,h,'replicate');

    %Recorremos los puntos que hemos seleccionado anteriormente
    for p = 1:Npuntos
        xt = X2(p)*2; %Actualizamos la posicion al siguiente nivel
        yt = Y2(p)*2; %Actualizamos la posicion al siguiente nivel
        dt=d(p,:)*2;  %Actualizamos el desplazamiento al siguiente nivel

        %Generar una rejilla alrededor del pixel
        [iX iY oX oY isOut] = genMesh(xt,yt,winR,M,N);
        if isOut, continue; end
```

```
            %Extraemos los gradientes
            Ix = interp2(iX,iY,img1x(iY,iX),oX,oY);
            Iy = interp2(iX,iY,img1y(iY,iX),oX,oY);
            I1 = interp2(iX,iY,img1(iY,iX),oX,oY);
           %Resolución iterativo
            for q = 1:maxIter
                  [iX iY oX oY isOut] = genMesh(xt,yt,winR,M,N); %Genera una rejilla
                  if isOut, break; end %si fuera de la imagen cancelamos el punto

                  %Interpolamos los puntos y calculamos la derivada temporal
                  It=interp2(iX,iY,img2(iY,iX),oX,oY) - I1;
                  %Resolución del sistema para el actual pixel
                  b=It(:);
                  A=[Ix(:),Iy(:)];
                  vel=A\b;
                  %Actualizamos las variables del proceso iterativo
                  xt = xt+vel(1);
                  yt = yt+vel(2);
                  dt=vel'+dt;
                  %%%%%%%%%%%%%%%%%%%%%%%%%%%%

                  if max(abs(vel))<th, break; end %velocidad>umbral lo cancelamos
             end
           %Actualizamos las variables para el siguiente nivel
            X2(p) = xt;
            Y2(p) = yt;
      d(p,:)=dt;
      end
end
end
```

5.2.3.2 Programa CPU-GPU

5.2.3.2.1 LKT_main

Descripción y código del programa principal.

Código 5.8: función LKT_main programada para la ejecución en la GPU

Inicialización del entorno

```
clear all        %Eliminamos las variables anteriores
clc              %Limpiamos el entorno
reset(gpuDevice);%Reseteamos la GPU
warning('off');
```

Cargamos las imágenes

```
load LKT2;

%%Procedimiento alternativo:
%fr1=imread('ima1.jpg'); %Almacenamos en fr1 la imagen de nombre 'ima1.jpg'
%fr2=imread('ima2.jpg'); %Almacenamos en fr2 la imagen de nombre 'ima2.jpg'
```

Parámetros

```
w=11;       %Tamaño de la vecindad 5x5 numeros impares asonsejable
Lm = 4;     %Número de niveles
maxIter=50; %Número de iteraciones
N_puntos=20;
```

Puntos bien condicionados

```
[C] = corner(rgb2gray(fr1),'MinimumEigenvalue',N_puntos)-1; %restamos uno para
compatibilizar con C#
X1=C(:,1);
Y1=C(:,2);
```

Creamos el objeto k asociado al kernel

```
k = parallel.gpu.CUDAKernel('LKT_kernel.ptx','LKT_kernel.cu','main_LKT');
```

Comienzo del proceso LKT

```
[C2]=LKT_GPU(fr1,fr2,Lm,C,w,maxIter,k);
```

Mostramos los resultados

```
X2=C2(:,1);
Y2=C2(:,2);
u=X2-X1;
v=Y2-Y1;
```

Mostramos los resultados

```
imshow(fr2)

hold on
plot(X1,Y1,'b*');
[u,v]=arrayfun(@flechas,X2,Y2,u,v);
plot(C2(:,1),C2(:,2),'go')
```

5.2.3.2.2 LKT_GPU

Descripción y código del algoritmo LKT_GPU

Código 5.9: función LKT_GPU programada para la ejecución en la GPU

```
function [C]=LKT_GPU(fr1,fr2,Lm,C,w,maxIter,k)
%Algoritmo para calcular el flujo óptico mediante LKT piramidal utilizando
%CUDA
%   +Npuntos: Numero de puntos donde se calculará el flujo optico
%   +Lm: Numero de niveles piramidales
%   +Icell: Celda donde se almacenan todos los niveles piramidales 1ª imagen
%   +Jcell: Celda donde se almacenan todos los niveles piramidales 2ª imagen
%   +X2: desplazamiento del punto en la 2ª imagen
%   +Y2  ""
%   +th: valor
%   +winR: tamaño de la ventana a muestrear
%   +maxIter: maximo número de iteraciones C=C/(2^Lm);
```

Tratamiento de imágenes y cálculo piramidal

```
img1 = im2double(rgb2gray(fr1));  %Conversion de la imagen a gris
img2= im2double(rgb2gray(fr2));
Icell = pyr(img1,Lm);
Jcell = pyr(img2,Lm);
```

Parámetros del método

```
h = fspecial('sobel');  %Cálculo del operador sobel
```

Inicio de las iteraciones

```
    for level = Lm:-1:1

     %Seleccionamos las imagenes piramidales si Lm=3--> 2 1 0
     img1=Icell{level};
     img2=Jcell{level};

     %Calculamos los gradientes mediante el operador sobel
     img1x = imfilter(img1,h','replicate');
     img1y = imfilter(img1,h,'replicate');
```

```
     %Configuramos el kernel, numero de hilo, bloques....
     k.ThreadBlockSize=[size(C,1) 1 1];
     %Inicializamos los vectores u y v que almacenan la salida del kernel
     u=zeros(size(C,1),1);
     v=u;
     %Ejecutamos los múltiples hilos en la GPU

[x,y]=feval(k,u,v,img1,img2,img1x,img1y,C(:,2),C(:,1),size(img1x,1),size(img1x,2),5,w)

     %Eliminamos posibles valores erroneos
     mask_nan=1-((isnan(x)+isnan(y))>0);%valores que no son Nan
     x2 = gather(x(mask_nan==1));
     y2 = gather(y(mask_nan==1));

     C=[x2 y2]
    end
end
```

5.2.3.2.3 Kernel

El kernel programado para el algoritmo LKT está formado por tres funciones:

-main_LKT: Función principal, resolución iterativa para un pixel.

-rejilla: Calcula si el pixel está dentro de los límites posibles de la imagen.

-interp: Calcula la interpolación bilineal.

Función auxiliar para la interpolación bilineal:

Dado un punto cualquiera, (x, y), la función de intensidad de la imagen y el número de filas la función interp devuelve el valor de intensidad para dicho punto mediante un proceso de interpolación.

Recordemos que la etiqueta __device__ implica que la función solo es accesible para la GPU.

Código 5.10: Función de interpolación

```
__device__ void interp(double y,double x,double *I,double *res,int N)
{
//COLUMNA*N_filasl+FILA
double ax;
double ay;
int xo=x;
int yo=y;

ax=x-xo;
ay=y-yo;

*res=(1-ax)*(1-ay)*I[yo*N+xo]+ax*(1-ay)*I[yo*N+xo+1]+(1-
ax)*ay*I[(yo+1)*N+xo]+ax*ay*I[(yo+1)*N+xo+1];
}
```

Función auxiliar rejilla:

Dado el radio de la vecindad, la posición del píxel, fila y columna, y las dimensiones de la imagen, la función rejilla determina si dicha vecindad está dentro o fuera de los límites de la imagen.
El objetivo de esta función es asegurar que los procesos se están realizando con los datos correctos y no con información que no pertenecen a nuestro programa.

Código 5.11: Funcion rejilla

```
__device__ void rejilla(int w,int f,int c,int M,int N,int*isOut)
{//M=filas N=columnas
    int hi,hd; //posiciones de los píxeles mas alejados del pixel central, dirección horizontal
    int vu,vd; //posiciones de los pízeles mas alejados del píxel central, dirección vertical
    hi=c-w;    // posición del pixel-radio de la ventana,límite izquierdo
    hd=c+w;    //posición del pixel+radio de la ventana,límite derecho
    vu=f+w;    //posición del pixel+radio de la ventana,límite superior (up)
    vd=f-w;    //posición del pixel+radio de la ventana,límite inferior (down)
    *isOut=0;  //isOut=0 significa que el píxel está dentro de los limites de la imagen
    if(hi<0 || hd>N) // la componente horizontal tiene que estar entre [0,N_filas]
    {
     *isOut=1; //isOut=1, el píxel está fuera de los límites, valores no válidos para procesar.
    }
    if(vd<0 || vu>M) //La componente vertical tiene que estar entre [0,N_columnas]
    {
     *isOut=1;
    }
}
```

Función principal main_LKT:

Las variables de entrada de esta función son:

- double *u,double*v: Punteros a un vector donde se almacenará los valores de desplazamiento
- double *I,double *J: Punteros a las imágenes.
- double *Ix,double*Iy: Punteros al gradiente especial en x e y.
- double *fil,double *col: Punter a un puntero donde se almacenan las componentes de la posición de los puntos que se van a procesar.
- int M,int N: Tamaño de la imagen.
- int maxIter: Número máximo de iteraciones.

Código 5.12: Kernel LKT programado en C

```
int w __global__ void main_LKT(double *u,double*v,double *I,double *J,double *Ix,double*Iy,double
*fil,double *col,int M,int N,int maxIter,int w)
{
     int idx = blockDim.x*blockIdx.x+threadIdx.x; //Identificador de hilo
     int i,j,q;  /*indexar*/
     int ww;     //Radio de la vecindad
     double f,c; //variable que almacena la fila y la columna
     double f1,c1;
     int isOut;  //variable para calcular si esta fuera de la imagen
     double Jaux;
     double Iaux;
     /*Sumatorios*/
     double Gx=0;
     double Gy=0;
     double Gt=0;
     double det=0;
     double gxx,gxy,gyy,ey,ex;
     gxx=0;
     gyy=0;
     gxy=0;
     ex=0;
     ey=0;
          //Asociamos al thread su punto correspondiente
```

```
      c=2*(col[idx]);    //Ejemplo, el hilo 0 procesara el punto 0, el hilo 1 procesara el pixel 1
      f=2*(fil[idx]);
      f1=f;
      c1=c;
       /*Comprobamos si el punto está fuera de los límites de la imagen */
      ww=(w-1)/2;
      rejilla(ww,f,c,M,N,&isOut);
      if(isOut==1) //Si el punto está fuera no realizamos el tratamiento y borramos el resultado
      {
         u[idx] = 0;
         v[idx]=0;

      }
      else
      {/***********Proceso iterativo***********/
         for(q=0;q<maxIter;q++)
         {
         //Comprobamos si estamos dentro de la imagen
         rejilla(ww,f,c,M,N,&isOut);
         if(isOut==1) break;

         for(i=0;i<w;i++)
         {
             for(j=0;j<w;j++)
             {
             //Ix[((int)c-ww+j)*M+(int)f-ww+i]=aux; //Comprobamos si elige la vecindad correcta
             interp(c1-ww+j,f1-ww+i,Ix,&Gx,M);
             interp(c1-ww+j,f1-ww+i,Iy,&Gy,M);
             interp(c-ww+j,f-ww+i,J,&Jaux,M);
             interp(c1-ww+j,f1-ww+i,I,&Iaux,M);
             Gt=-Iaux+Jaux;

             gxx=gxx+Gx*Gx;
             gyy=gyy+Gy*Gy;
             gxy=gxy+Gx*Gy;
             ex=Gx*Gt+ex;
             ey=Gy*Gt+ey;
             //Para comprobar que suma correctamente, lo hace Ix[((int)c-ww+j)*M+(int)f-ww+i]=gxx;
             //aux=aux+1
             }
         }
         det=gxx*gyy-gxy*gxy; //Cálculo del determinante
         c = c+((gyy*ex - gxy*ey)/det);  //componente c del desplazamiento
         f= f+((gxx*ey - gxy*ex)/det);   //componente f del desplazamiento

        }
        u[idx]=c;
        v[idx]=f;
      }/************************************/

}
```

5.2.4 Método de empleo

Para utilizar la función LKT_CPU y LKT_GPU solamente es necesario disponer de:

[C]=LKT_GPU(fr1,fr2,Lm,C,w,maxIter,k)

[X2, Y2]=LKT_CPU(fr1,fr2,Lm,X1,Y1,th,winR,maxIter)

-fr1 y fr2: Imágenes secuencia almacenadas por ejemplo en formato .jpg

-w: Ancho de la ventana de integración o de la vecindad.

-Lm: Máximo nivel piramidal deseado.

-C o X1, Y1: Puntos bien condicionados almacenados como vector.

-maxIter: Número de iteraciones que se desea realizar.

La captura de las imágenes podemos realizarla de diversas maneras:

- Cargándolas desde un archivo preconfigurado,
- Cargándolas desde una ubicación en el ordenador mediante el comando imread('nombre.jpg').
- Cargando un video como variable del entorno de Matlab y extrayendo la secuencia deseada.

6 RESULTADOS

6.1 Entorno de las pruebas.

La GPU utilizada en este trabajo es la GeForce GTX 850M, dispositivo de gama media de la compañía NVIDIA enfocado al procesado de gráficos para ordenadores portátiles. Es una de las primeras tarjetas gráficas basadas en la nueva estructura Maxwell, tecnología usada en las gráficas más potentes del mercado actual.

El campo donde más se utiliza este tipo de dispositivo, y para el cual está enfocado, es el de los videojuegos, debido al aumento de su popularidad y de la creciente complejidad que están adquiriendo. Gracias al desarrollo en este campo, las tarjetas gráficas son cada vez más un recurso viable para la optimización de algoritmos.

Las características más importantes de nuestro dispositivo son:

Codename	N15P-GT
Architecture	Maxwell
Pipelines	640 - unified
Core Speed *	876 MHz
Memory Speed *	2000 - 5000 MHz
Memory Bus Width	128 Bit
Memory Type	DDR3, GDDR5
Max. Amount of Memory	4096 MB
Shared Memory	no
DirectX	DirectX 12 (FL 11_0), Shader 5.0
Transistors	1870 Million
technology	28 nm
Features	Battery Boost, GameStream, ShadowPlay, GPU Boost 2.0, Optimus, PhysX, CUDA, SLI, GeForce Experience
Notebook Size	medium sized
Date of Announcement	12.03.2014

Figura 6.1 Características de la GPU

6.2 Primeros resultados (algoritmo LK):

A la hora de realizar las pruebas del algoritmo basado en el método de Lucas-Kanade vamos a prestar atención

a dos aspectos principalmente, al correcto funcionamiento de los programas y al tiempo de cómputo de las dos modalidades de funcionamiento planteadas, secuencial y paralela. Por consiguiente en los apartados siguientes, Resultados gráficos y Comparativas de tiempo vamos a abordar ambos aspectos.

6.2.1 Resultados gráficos:

Para comprobar el correcto funcionamiento del programa se utilizó una secuencia de imágenes con un movimiento muy concreto, un desplazamiento horizontal. La imagen se ha diseñado expresamente para que, en el caso de no existir ningún fallo de implementación, el programa arroje resultados óptimos. En la figura 6.1 podemos ver tres frames correspondientes al video completo.

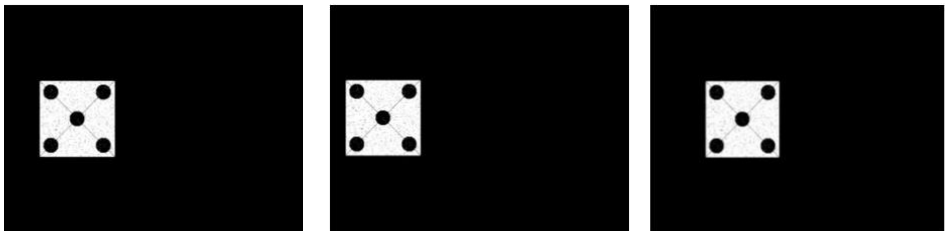


Figura 6.2: Secuencia de un dado moviéndose horizontalmente

6.2.1.1 Primeros resultados:

En un primer momento se aplicó directamente la formulación básica de Lucas-Kanade sobre varios pares de imágenes del video. Como se puede apreciar en la figura 6.3 encontramos una mayoría de vectores cuya dirección y sentido muestran un desplazamiento horizontal de los píxeles. Por otro lado existe un porcentaje reducido, pero considerable, que no presenta un comportamiento coherente. A continuación se muestran los parámetros utilizados para la simulación junto con los resultados:

Tamaño de la ventana: 20 píxeles

Distancia entre píxeles: 20 píxeles

Tamaño de la imagen: 696x977

Nº Puntos: 1715

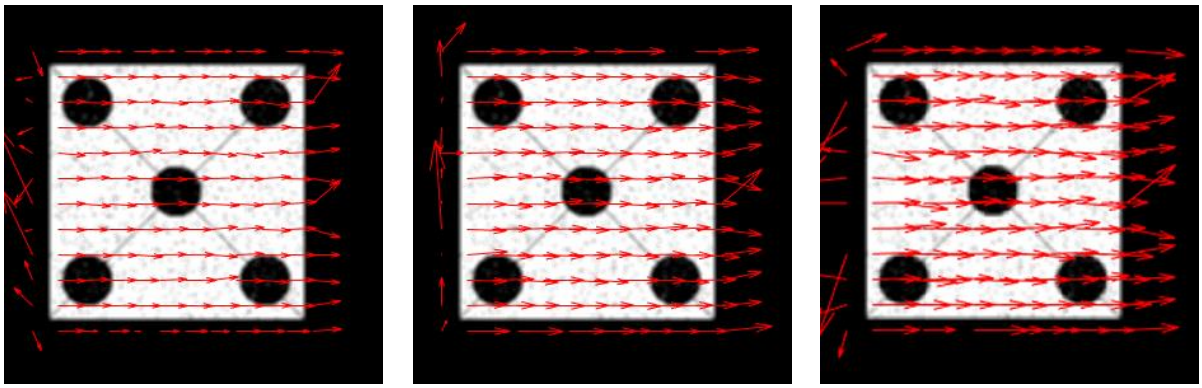


Figura 6.3: Resultados del algoritmo LK sin restricción en los autovalores.

6.2.1.2 Con selección de autovalores

Con el fin de obtener mejores resultados se aplicó una serie de restricciones nuevas. Recordemos que para

cada pixel se resolvía un sistema de ecuaciones $\boldsymbol{Ax} = \boldsymbol{b}$, un buen criterio para comprobar que el resultado será correcto o no es analizar los autovalores de A, si no pertenecen a un cierto rango determinado se considera no válidos y se descartan.

Como se puede apreciar en la figura 6.3 los desplazamientos son prácticamente horizontales en todos los puntos calculados reduciéndose los vectores incorrectos en comparación con la figura 6.3.

Tamaño de la ventana: 20 Nº Puntos: 1715

Puntos seleccionados cada 20 px Tamaño de la imagen: 696x977

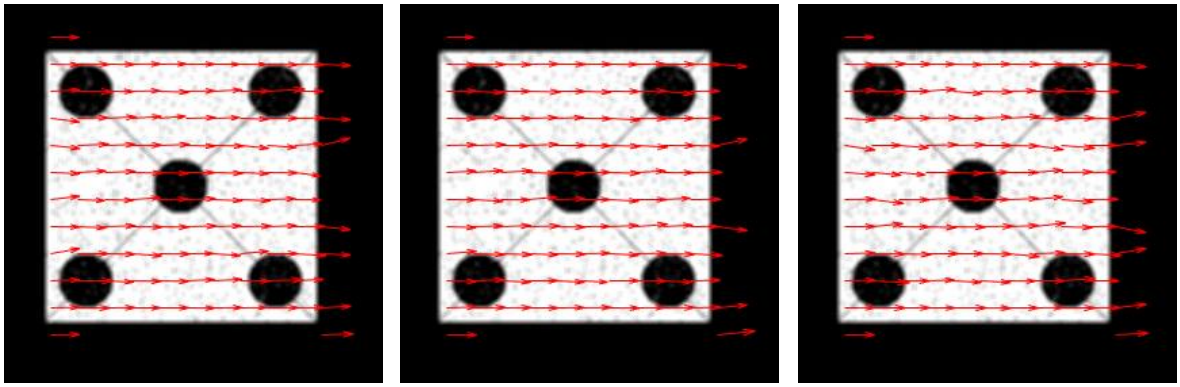


Figura 6.3: Resultados del algorimo LK con restricción en sus autovalores.

6.2.2 Prueba en un entorno real

Para este ensayo se ha utilizado un video de una cámara colocada sobre un vehículo aéreo sobrevolando una pista de aterrizaje. Las características y los resultados de la prueba son las siguientes:

Tamaño de la ventana: 20 píxeles Tamaño de la imagen: 696x977

Distancia entre píxeles: 20 píxeles Nº Puntos: 1715

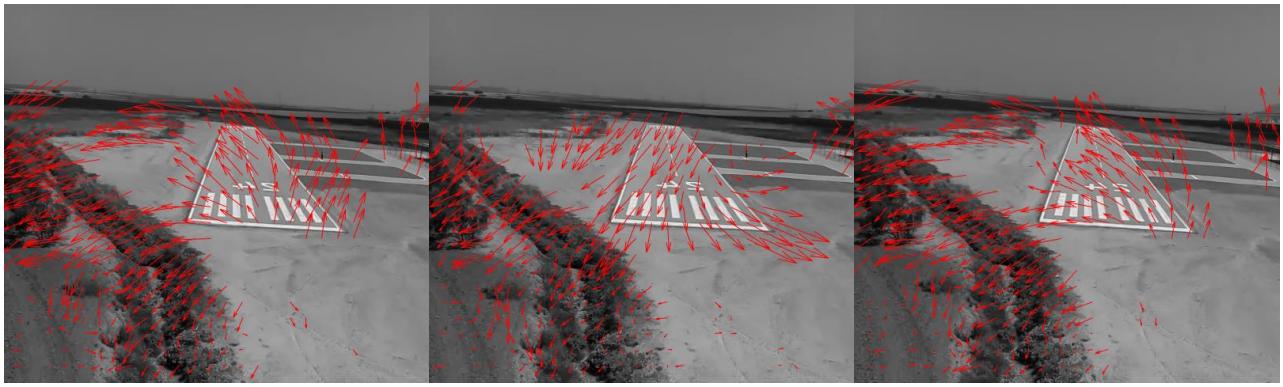


Figura 6.4: Resultados del algoritmo para un entorno real.

6.2.3 Comparativa de tiempo algoritmo GPU/CPU

Tras ejecutar la implementación secuencial y paralela se han medido los tiempos de computación en función de los números de píxeles seleccionados en la rejilla.

El tamaño de la vecindad que se ha utilizado es de 7x7 pixeles, lo que arroja unas 49 ecuaciones por punto. El

tamaño de la imagen utilizado ha sido 1600x1800 pixeles.

Los resultados obtenidos se han resumido en las tres gráficas siguientes. Como se puede observar el tiempo de procesamiento aumenta de manera exponencial en la implementación secuencial y se mantiene prácticamente constante en la implementación paralela.

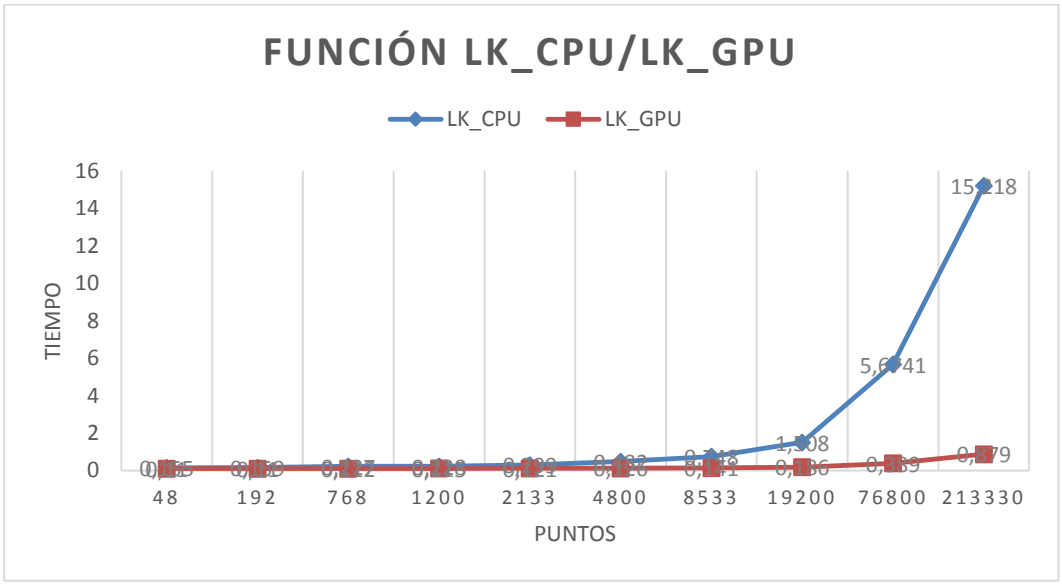


Figura 6.5: Comparativa en tiempo de la función LK_CPU y LK_GPU, las cuales implementan la formulación correspondiente al método Lucas-Kanade

.

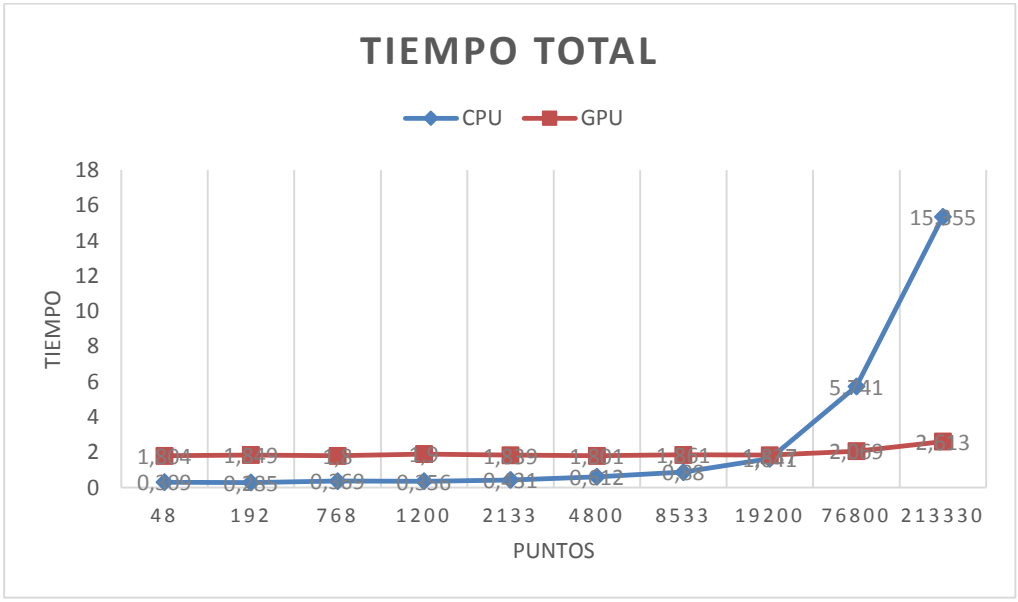


Figura 6.6: Comparativa en tiempo de la función principal, la cual engloba todo el procesamiento de imágenes y video además de las funciones LK_CPU y LK_GPU en cada caso.

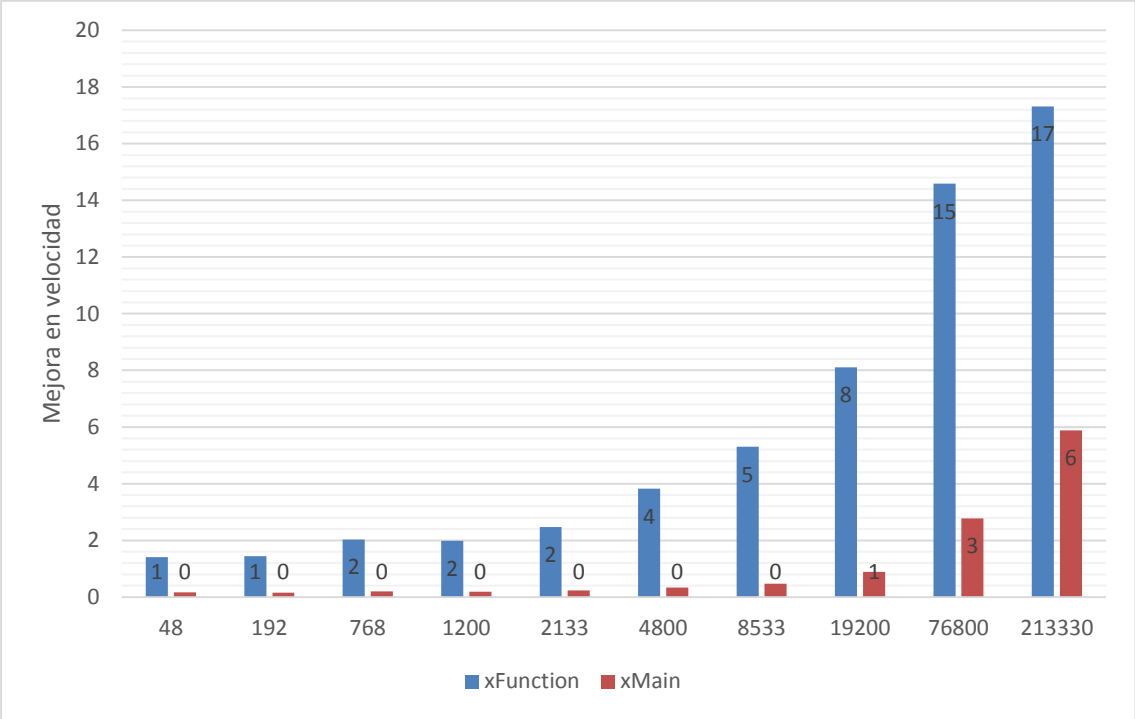


Figura 6.7 Factor de mejora de la función LK_main y LK_xPU

6.3 Resultados finales (algoritmo LKT):

A la hora de realizar las pruebas del algoritmo basado en el método LKT vamos a prestar atención a dos aspectos principalmente, al correcto funcionamiento de los programas y al tiempo de cómputo de las dos modalidades de funcionamiento planteadas, secuencial y paralela. Por consiguiente en los apartados siguientes, Resultados gráficos y Comparativas de tiempo vamos a abordar ambos aspectos.

6.3.1 Resultados gráficos:

Para comprobar el correcto funcionamiento del programa se utilizó la misma secuencia de imágenes que en el apartado anterior pero describiendo en este caso una trayectoria paralela. En la figura 6.1 podemos ver tres frames correspondientes al video:

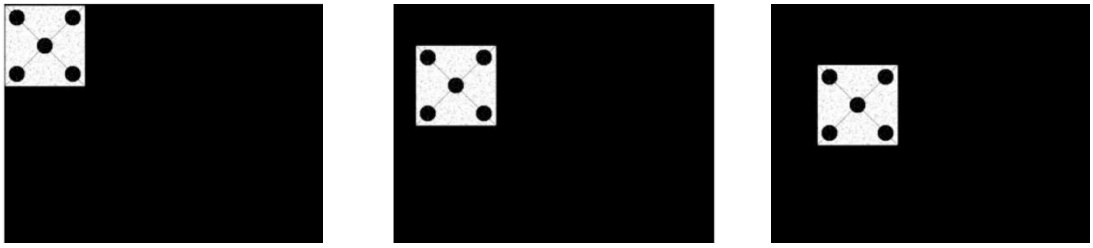


Figura 6.8 Secuencia de un dado moviéndose siguiendo una trayectoria paralela

En la Figura 6.10 podemos apreciar la secuencia anterior junto con los puntos bien condicionados y los vectores de desplazamiento de los píxeles. A la izquierda se ha realizado el experimento para 20 puntos y en la derecha para 200.

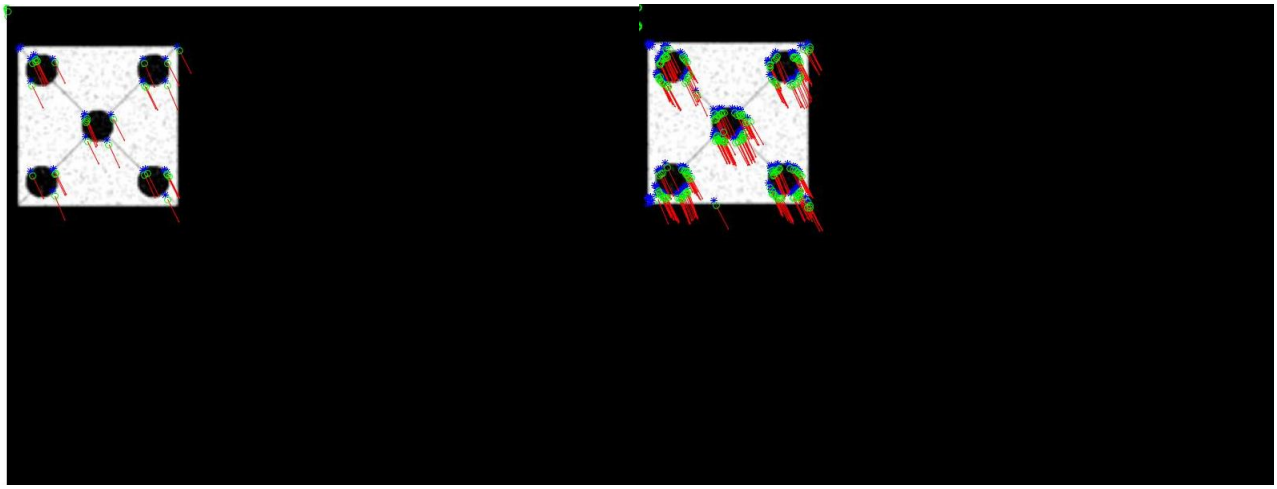


Figura 6.9: Resultados gráficos del método LKT para 20 y 200 puntos.

En la secuencia siguiente podemos apreciar las imágenes recogidas desde la cámara de un vhiculo arereo mientras aterriza sobre una pista. El algoritmo LKT utiliza puntos bien condicionados para ejecutar el algoritmo, de ahí que la mayor parte de los vectores partan de las esquinas de las señales identificadivas del suelo.

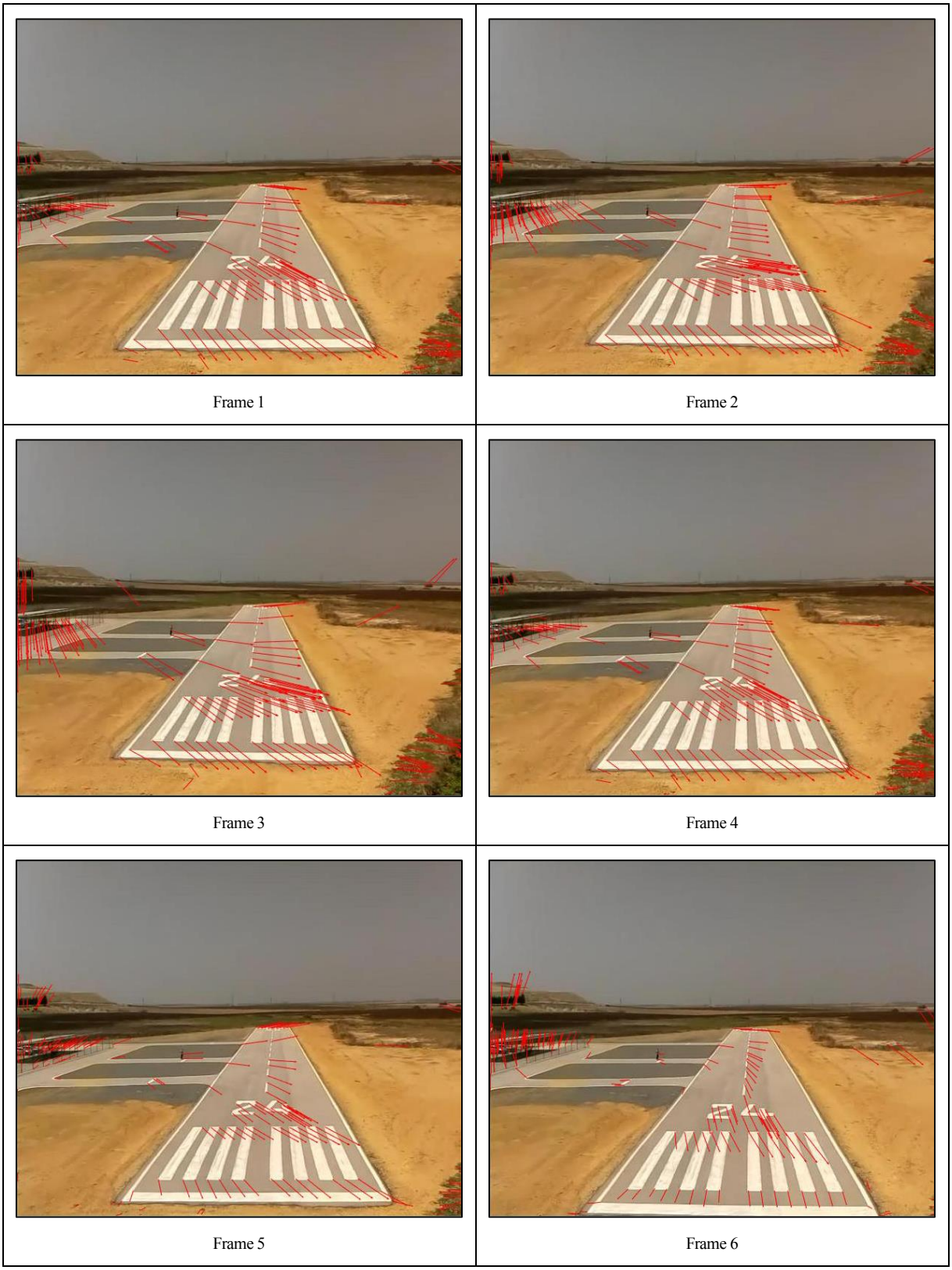


Figura 6.10: Algoritmo LKT aplicado al aterrizaje de un avión

Conforme el avión se coloca de manera paralela a la pista los vectores de desplazamiento se tornan cada vez más verticales.

6.3.2 Comparativa de tiempo algoritmo GPU/CPU:

Tras ejecutar la implementación secuencial y paralela se han medido los tiempos de computación en función del número de píxeles bien condicionados que busca el método.

El tamaño de la vecindad que se ha utilizado es de 7x7 pixeles, lo que arroja unas 49 ecuaciones por punto. El tamaño de la imagen utilizado ha sido 1600x1800 pixeles.

Los resultados obtenidos se han resumido en las tres gráficas siguientes. Como se puede observar el tiempo de procesamiento aumenta de manera exponencial en la implementación secuencial y se mantiene prácticamente constante en la implementación paralela.

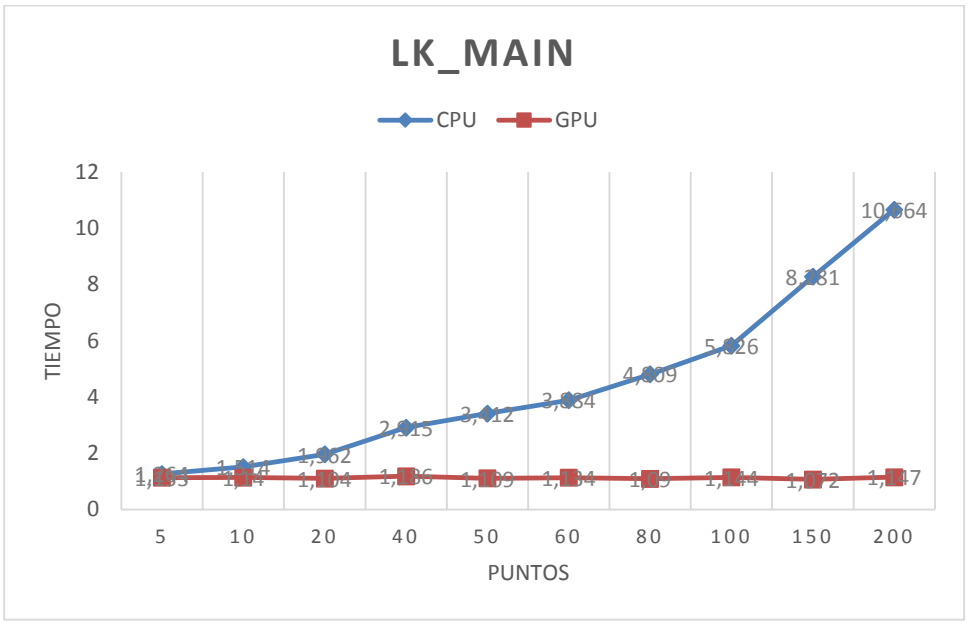


Figura 6.11: Factor de mejora de la función LKT_main

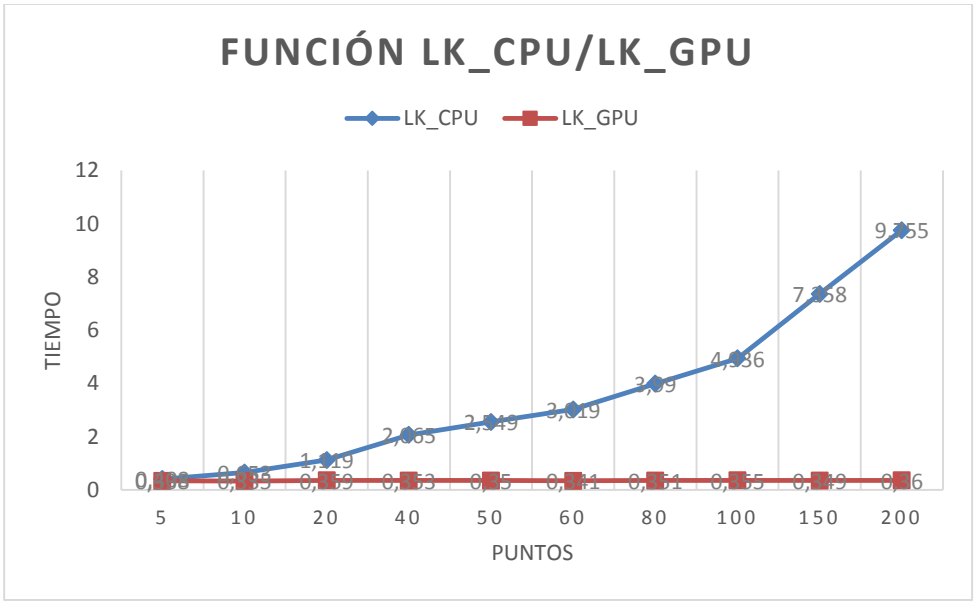


Figura 6.12: Factor de mejora de la función LKT_main y LKT_xPU

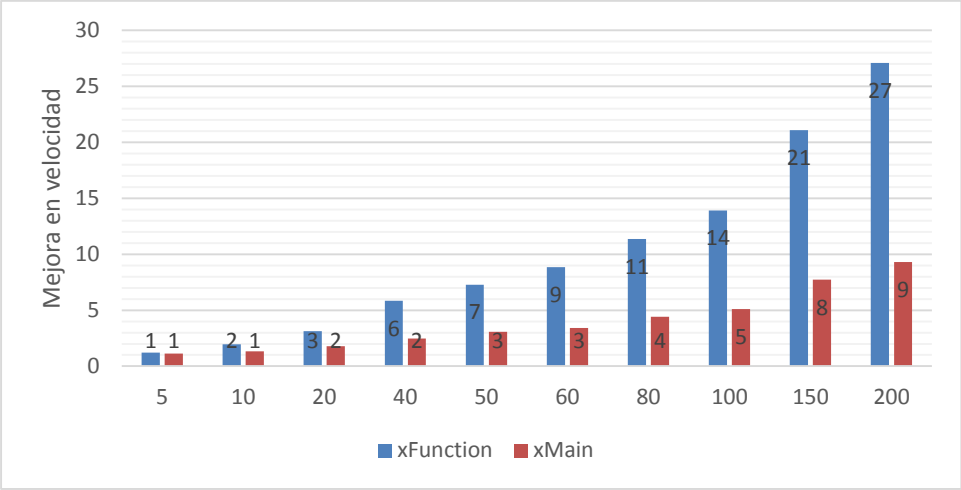


Figura 6.13 Factor de mejora de la función LK_main y LK_xPU

7 CONCLUSIONES Y FUTURAS MEJORAS

La aplicación de estos programas optimizados estaba enfocada a ser aplicada a un problema concreto, posicionamiento y orientación de vehículos aéreos para el aterrizaje automático. El algoritmo Lucas-Kande-Tomasi ofrece buenos resultados a la hora de calcular el Flujo Óptico y desplazamientos entre frames de una secuencia y, gracias a la paralelización usando la GPU, la mejora del rendimiento es notable. Los programas aquí explicados arrojan unos resultados que, tras un posible procesamiento e integración, pueden ofrecer información concreta de la orientación.

Por otro lado el entorno visual donde nos encontramos impone ciertas barreras y dificultades a la hora de aplicar correctamente los algoritmos, por ejemplo, tras el procesamiento de los datos para la selección de puntos bien condicionados es muy común que el programa considere algunos que en realidad no lo son, fenómeno que ocurre en gran medida tras la filmación de texturas como agua o vegetación.

El algoritmo LKT programado en este proyecto consideraba solo desplazamientos puros de los píxeles de la imagen, en futuras mejoras puede considerarse la introducción de modelos de desplazamiento complejos y transformaciones geométricas que ayuden a obtener resultados gráficos más precisos.

A pesar de que la optimización reduce en gran medida el procesado de píxeles, Matlab no deja de ser un intermediario entre el usuario y una programación en CUDA C pura, es posible, una vez comprendido el funcionamiento de los algoritmos de visión, que la migración del código de Matlab a CUDA C arroje mejores resultados.

REFERENCIAS

[1] BOUGUET, Jean-Yves. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel Corporation*, 2001, vol. 5, p. 1-10.

[2] Center for research in computr vision. Optical Flow: 16/09/2015, 01:31:18, Disponible en : http://crcv.ucf.edu/source/optical

[3] Jianbo Shi and Carlo Tomasi. Good features to track, Proc. IEEE Comput. Soc. Conf. Comput. Vision and Pattern Recogn. 1994, pages 593–600.

[4] Mathworks. Image Processing Toolbox, 16 Septiembre 2015. Disponible en: http://es.mathworks.com/products/image/features.html#caracter%C3%ADsticas-principales

[5] Mathworks.Parallel Computing Toolbox, http://es.mathworks.com/products/parallel-computing/

[6] NVIDIA DEVELOPER ZONE. CUDA C Programming Guide, 1 Septiembre 2015: Disponible en: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3lGW2a7Bg

[7] David J. Fleet, Yair Weiss .Optical Flow Estimation. En N.Paragios, Y. Chen, O. Faugeras. Mathematical Models in Computer Vision: The Handbook. Springer, 2005, pp. 239-258

[8] Cyril Zeller, NVIDIA Corporation. CUDA C/C++ Basics, 23 de noviembre 2011, 07:03. Disponible en: http://www.nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf

[9] Shan, Amar. Heterogeneous Processing: a Strategy for Augmenting Moore's Law. Linux Journal. 2006

[10] NVIDIA DEVELOPER ZONE. An Easy Introduction to CUDA C and C++, 31 Octubre 2012. Disponible en: http://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/

[11] Mohamed Zahra, Lecture 5: CUDA Threads, 2 Noviembre 2012. Disponible en: http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture5.pdf

[12] Intro to image processing with CUDA. Disponible en http://supercomputingblog.com/cuda/intro-to-image-processing-with-cuda/

[13] Intro to image processing with CUDA. Disponible en http://supercomputingblog.com/cuda/intro-to-image-processing-with-cuda/

[14] Mark Ebersole, Getting Started with CUDA C/C++. Disponible en: http://on-demand.gputechconf.com/gtc/2013/presentations/S3049-Getting-Started-CUUA-C-PlusPlus.pdf

[15] MathWorks, Run CUDA or PTX Code on GPU. Disponible en: http://es.mathworks.com/help/distcomp/run-cuda-or-ptx-code-on-gpu.html

[16] BERENGUEL, M.; RUBIO, FR; VALVERDE, A; LARA, PJ; ARAHAL, M.R.; CAMACHO, EF; LOPEZ, M. An artificial vision-based control system for automatic heliostat positioning offset correction in a central receiver solar power plant. Solar Energy, 2004, vol. 76, no 5, p. 563-575.

[17] GAVILAN, Francisco; ARAHAL, Manuel R.; IERARDI, Carmelina. Image Deblurring in Roll Angle Estimation for Vision Enhanced AAV Control. IFAC-PapersOnLine, 2015, vol. 48, no 9, p. 31-36.

[18] HORMIGO, Alberto Monino; RUBIO, Francisco Rodrıguez; ARAHAL, Manuel Ruiz. Monitorización y Predicción de la Radiación Solar Mediante Visión del Paso de Nubes.

[19] TORAL, Sergio, et al. Improved sigma–delta background estimation for vehicle detection. Electronics letters, 2009, vol. 45, no 1, p. 32-34.

[20] LIMON, D.; ALVARADO, I; ALAMO, T; ARAHAL, M.R.; CAMACHO, E.F. Robust control of the distributed solar collector field ACUREX using MPC for tracking. En Proceedings of 17th IFAC World Congress, Seoul, Korea. 2008. p. 958-963.

[21] TORAL, Sergio L.; VARGAS, Manuel; BARRERO, Federico. Embedded multimedia processors for road-traffic parameter estimation. Computer, 2009, no 12, p. 61-68.

[22] TORAL, S. L.; BARRERO, F.; VARGAS, M. Development of an embedded vision based vehicle detection system using an ARM video processor. En Intelligent Transportation Systems, 2008. ITSC 2008. 11th International IEEE Conference on. IEEE, 2008. p. 292-297.

[23] VARGAS, Manuel, et al. A license plate extraction algorithm based on edge statistics and region growing. En Image Analysis and Processing–ICIAP 2009. Springer Berlin Heidelberg, 2009. p. 317-326.

ÍNDICE DE CÓDIGO