Aceleración HW de algoritmos de disparidad binocular

Departamento de Arquitectura de Computadores y Automática

Curso 2011 / 2012

Autores:
Nehuén Eloy Benítez
Daniel González Flores
Roberto Ortiz Merino

Directores:
Guillermo Botella Juan
Jose Antonio Martín Hernández

Se autoriza a la Universidad Complutense de Madrid a dinfundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria como el código, la documentación y/o el prototipo desarrollado.

Nehuén Eloy Benítez

Daniel González Flores

Roberto Ortiz Merino

A mis padres,
Nehuén

A Pedro,
Daniel

A mi familia,
Roberto

Agradecimientos

A Pedro Jesús Martín de la Calle por dedicarnos tiempo y compartir su experiencia en programación de GPUs.

A Ricardo Ivan Stepaniuk por compartir sus conocimientos en optimización y arquitecturas paralelas.

Resumen

Dentro de la visión por computador, el campo de la visión estereoscópica es un campo muy amplio y muy estudiado en los últimos años. En este campo se estudia la aproximación de manera fiel una escena tridimensional a partir de dos o más imágenes bidimensionales. Tal y como ocurre en los seres vivos, los ojos están desplazados a una cierta distancia, las cámaras también se deben separar cierto espacio. La computadora debe analizar las imágenes y determinar qué pixel de una corresponde a qué pixel de otra. La distancia entre ellos se conoce como disparidad, y es medida en píxeles. A partir de la disparidad se puede obtener la profundidad de ese punto en la escena tridimensional. Actualmente existen multitud de algoritmos, unos aproximan la escena mejor que otros, pero la mayoría tiene un problema crucial: no llegan a computar la imagen tridimensional en tiempo real y si llegan suele ser a costa de una pérdida en la calidad de la imagen final o utilizando procesadores del segmento más alto. Nuestro proyecto esta orientado a buscar una solución a este problema paralelizando por hardware el cómputo de las funciones más pesadas (en términos de tiempo) de un algoritmo estereoscópico que proporcione imágenes tridimensionales de calidad. Para ello, utilizaremos la potencia de proceso de las GPUs (unidades de procesamiento gráfico) a través de la interfaz de programación que proporciona OpenCL (Open Computing Language).

Palabras clave visión estéreo, estereopsis, visión por computador, escena 3D, disparidad, GPU, OpenCL

Abstract

Inside computer vision, stereoscopic vision is a very wide field and it has been deeply studied in the last years. The main objetive in this field is to approach, in a faithful way, a 3D scene from two or more 2D images. As it happens with living beings, in which the eyes are placed at a certain distance, cameras must be also placed keeping some space. The computer has to analyze both images and determine which pixel from the first, matches a pixel in the second image. The distance between those pixels is called disparity, and it's measured in pixels. Based on the disparity of a pixel and knowing the geometry of the cameras, the depth of that pixel, in the 3D scene, can be computed. Nowadays, there are lots of stereo algorithms, some approaches are better than others, but most have the same problem: they can't compute the 3D scene in real time and, if they do, they do it at the price of lower quality or, by computing with top processors. Our project tries to find a solution to this problem parallelizing by hardware the computation of the heavier functions (speaking in time) of an stereo algorithm which estimates a 3D scene with good quality. To this end, we will use the computational power of the GPUs (Graphics Processing Units) through the API provided by OpenCL (Open Computing Language).

Keywords stereo vision, stereopsis, computer vision, 3D scene, disparity, GPU, OpenCL

Índice general

Introducción

Para llevar a cabo la optimización del algoritmo de visión estereoscópica, primero tuvimos que estudiar bien los conceptos básicos que rodeaban al proyecto. Éstos conceptos no son triviales y para poder realizar nuestro trabajo hemos tenido que invertir timepo y esfuerzo en estudiar y comprender detalladamente cada uno de ellos. De esta forma, en el Capítulo 1 se llevará a cabo una explicación detallada de los conceptos clave que sientan los cimientos sobre los que se construye nuestro proyecto. Además se explicará también el funcionamiento el algoritmo escogido.

Una vez entendida la base sobre la que se trabajará, en el Capítulo 2 se llevará a cabo una explicación de la metodología que se ha empleado para llevar a cabo la optimización del algoritmo. En concreto describiremos los puntos seleccionados para optimizar, las estrategias seguidas y la arquitectura de cada pieza de código.

En el Capítulo 3 se plantearán los experimentos realizados con el código optimizado y sus resultados.

Por último en los Capítulos 4 y 5 se mostrarán las conclusiones y el trabajo futuro respectivamente.

Capítulo 1

Contexto del proyecto

1.1. Visión por Computador

La Visión por Computador es la ciencia que desarrolla la base teórica y algorítmica mediante la que se extrae y analiza información sobre el entorno a partir de una imagen, un conjunto de ellas o una secuencia de las mismas. Los sensores, como tal, no ofrecen información por sí mismos por lo que hace falta procesar los datos obtenidos mediante algoritmos.

La utilización de estos algoritmos se utiliza para campos como la obtención de la profundidad y la estructura tridimensional de una escena. Por lo general se pueden definir tres tipos de niveles de abstracción de la visión por computador: visión a baja escala, media escala y alta escala [24].

La visión a baja escala obtiene información útil como puede ser color, disparidad binocular, procesado de movimiento, etc., de varios canales. Algunos de estos canales pueden ser identificados como campos receptivos que envían información a la retina. Otros, como la disparidad binocular o el procesado de movimiento, son combinaciones de los canales que hemos mencionado previamente.

La visión a media escala integra primitivas procesadas en el nivel previo de abstracción. La información enviada en este nivel corresponde a las inferencias del mundo real como la egomoción y los objetos con movimiento independiente (IMOs). Se denominan acciones causales u objetos candidatos en conexión con cualquier caracterización multimodal. Un ejemplo es la combinación de medidas luminosas para inferir brillo o dar forma mediante sombras.

La visión a alta escala interpreta la escena mediante tareas específicas como el razonamiento relacional, aprendizaje, reconocimiento de objetos, etc.

La escena es vista por una o más cámaras. Es importante que estas cámaras esten calibradas para evitar distorsiones de tipo "barril" o de tipo "cojín" [1] de las imágenes. Una vez colocadas, calibradas y apuntando a la escena, se capturan las imágenes sobre las que se trabajará. Para obtener la información de las imágenes se suelen segmentar obteniendo características como bordes o regiones que sientan un punto sobre el cual comparar unas imágenes con otras. Posteriormente y dependiendo del algoritmo, se realiza un procesado de dichas características tras lo cual se consigue recrear con mayor o menor aproximación la estructura de la escena tridimensional.

Este campo de la Visión por Computador [2] se conoce como Visión Estereoscópica y estudia los métodos y algoritmos que consiguen la obtención de una escena tridimensional a partir de imágenes bidimensionales.

La visión estereoscópica artificial se basa en el modelo estereoscópico biológico, es decir, en el funcionamiento de los ojos. Pongamos que tenemos dos imágenes (que en lo que sigue llamaremos imagen izquierda e imagen derecha) de la misma escena, con un pequeño desplazamiento entre ambas; podemos calcular por triangulación [2] la profundidad en la escena de los objetos deseados.

El trabajo del computador consiste en identificar en ambas imágenes aquellos pixeles que se corresponden con la misma entidad física en la escena 3D, usando para ello algoritmos especializados. La distancia que separa estos pixeles se conoce como disparidad. La medida de la disparidad sirve para obtener la distancia a la que se sitúa físicamente ese objeto en la escena con respecto a las dos cámaras. La correspondencia entre los puntos de ambas imágenes constituye el principal problema dentro del proceso de la visión estereoscópica.

Para resolver este problema hay que determinar mediante algún tipo de procesado qué parejas de puntos de ambas imágenes se corresponden con un mismo punto en la escena real. De una manera más formal, el problema de la correspondencia consiste en identificar una característica en una imagen, por ejemplo un punto de borde o región y determinar qué característica en la otra imagen es la que corresponde a la misma característica en el espacio tridimensional.

Se trata de un problema mal condicionado, pues geométricamente pueden existir infinitas soluciones o que no exista solución (oclusiones), lo que puede dar lugar a ilusiones ópticas (falsas correspondencias). La dificultad de establecer la correspondencia viene determinada

principalmente por el propio sistema, es decir, dadas las imágenes derecha e izquierda de un sistema de visión estéreo hay que tener en cuenta que éstas vienen afectadas por distintos ángulos de vista de la cámara e incluso diferentes iluminaciones.

¿Cómo se hace?

Partimos de un sistema formado por dos cámaras cuyos ejes ópticos son paralelos y se colocan el uno respecto del otro a una distancia denominada línea base. Los ejes de los sistemas están separados a lo largo de la horizontal por lo que un punto de la escena captado por las dos cámaras va a diferir sólo en su componente horizontal.

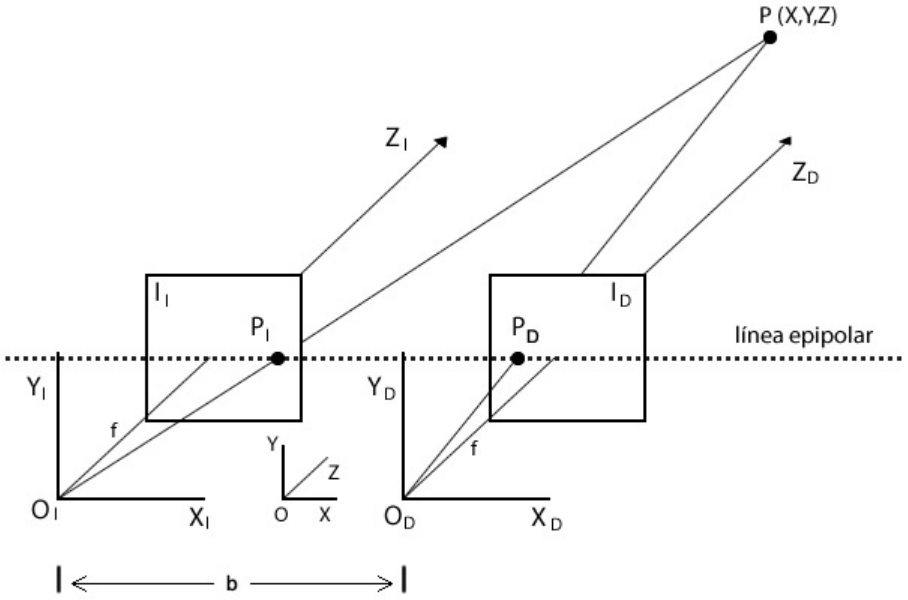


Figura 1.1: Geometría de dos cámaras con los ejes ópticos paralelos

El origen del sistema de coordenadas es **O**, siendo la longitud focal efectiva **f**, y la línea base, **b**. En cada una de las dos cámaras se establece un sistema de coordenadas 3D (X, Y, Z), por lo que PI(X_I, Y_I) y PD(X_D,Y_D) son las proyecciones en las imágenes izquierda y derecha, respectivamente, de un punto P(X, Y, Z) de la escena. Los rayos de proyección P_{OI} y P_{OD} definen un plano de proyección del punto en la escena 3D que recibe el nombre de plano epipolar.

Como consecuencia de la geometría del sistema se obtiene la denominada restricción epipolar, que limita la búsqueda de correspondencia de manera que en el sistema convencional de ejes

paralelos todos los planos epipolares generan líneas horizontales al cortarse con los planos de las imágenes. En un sistema como el anterior definimos la disparidad de un par de puntos emparejados $P_I(X_I , Y_I)$ y $P_D(X_D, Y_D)$ como X_I - X_D. Para establecer la correspondencia de un píxel de la imagen izquierda con uno de la imagen derecha se recorre la línea epipolar que pasa por él y se escoge el píxel que más se le parezca.

Unos términos que aparecerán a lo largo del documento son *imágen objetivo* e *imágen referencia*. La primera es la imágen de la que se parte, que define las coordenadas x e y de un punto en la escena. La segunda, es la que se usa para calcular la coordenada z del mencionado punto.

1.2. Estado del arte

En los últimos años han aparecido multitud de papers y trabajos de investigación relacionados con la visión estereoscópica pero surgen varias preguntas: ¿Cómo o dónde podemos comparar los distintos algoritmos? ¿Cómo saber si uno es mejor que otro? Para esta tarea se han creado varias webs que ofrecen distintos benchmarks para clasificar los algoritmos que van saliendo.

Uno de los más importantes es el benchmark de Middlebury [28]. En su página se muestra una tabla clasificatoria que agrupa los mejores algoritmos en las posiciones más altas. Esta web proporciona un lugar de encuentro dentro del campo de la visión estereoscópica donde poder encontrar los algoritmos.

Los algoritmos estereoscópicos se clasifican mediante la siguiente taxonomía [3] :

- Coste de computación de la correspondencia.
- Agregación de coste.
- Computación / optimización de la disparidad.
- Refinamiento de la disparidad.

Uno de sus principales usos en la actualidad es el campo de la robótica. Los investigadores utilizan la visión estereoscópica para que los robots que diseñan sean capaces de medir la profundidad de la escena que están observando y así poder detectar posibles obstáculos. La NASA por ejemplo utilizó en 2004 visión estereoscópica para la exploración planetaria con su

NASA's Mars Exploration Rover [8] y prevén utilizar en sus próximos lanzamientos algoritmos estereoscópicos más capaces de ayudar al robot a moverse por terrenos difíciles. Al hilo de esta aplicación, en la actualidad también se está intentando utilizar la visión estereoscópica para guiar desde aeronaves hasta turismos.

En el Instituto Tecnológico Karlsruhe (KIT) llevan desde 2006 trabajando en un proyecto denominado AnnieWAY [6] que consiste en lograr que un coche conduzca de forma autónoma basándose en la visión estereoscópica, de momento su proyecto ha obtenido resultados muy satisfactorios. A lo largo del desarrollo del proyecto han encontrado que algoritmos con calificación alta en el benchmark de Middlebury han obtenido resultados no muy satisfactorios cuando han sido aplicados al exterior. Por lo que han decidido crear otro benchmark junto con el Instituto Tecnológico Toyota de Chicago (TTI-C) llamado KITTI [7], cuyo objetivo es que los investigadores tengan más lugares donde probar las capacidades de sus algoritmos de visión estereoscópica.

En el campo aeronáutico se intenta utilizar para resolver los problemas que tienen los sistemas actuales como el GPS o el sistema de altitud AHRS. Estos sistemas son incapaces de llevar a cabo tareas como detectar obstáculos en la trayectoria, detectar terreno bajo, detectar barrancos en el camino o aterrizar en terrenos desconocidos. Para realizar estas tareas, la aeronave debe estar constantemente monitorizando su entorno. Esto se puede hacer utilizando sensores activos como láseres o radares, pero este tipo de dispositivos abultan demasiado y pueden comprometer el sigilo de la nave. Es aquí donde la visión estereoscópica ofrece una salida excelente, ya que obtiene información a través de sensores pasivos. Además, a través de sus algoritmos se puede ser capaz de orientar y manejar una nave de forma autónoma a baja altitud o en terreno aleatorio [9].

También han aparecido nuevas ideas y enfoques teóricos. Por ejemplo emplear aproximaciones ondícula (wavelets) [20] o multi-ondícula (multiwavelets) [20], o la introducción de campos aleatorios de Markov [21] para la estimación de la correspondencia o el empleo de conjuntos difusos de tipo 2 para la estimación y la extracción de las características de interés [9].

1.3. GPU

GPU (unidad de procesamiento gráfico) es un coprocesador dedicado al procesamiento de gráficos y operaciones en coma flotante, para liberar de trabajo a la CPU (unidad de procesamiento central) en determinadas aplicaciones con altas cantidades de operaciones con gráficos como pueden ser videojuegos o aplicaciones 3D interactivas.

Se caracterizan por tener una gran potencia de cálculo gracias a sus multiprocesadores y por la velocidad de su memoria interna, utilizada para almacenar los resultados de las operaciones intermedias. Para aprovechar la enorme potencia de cálculo y el gran número de procesadores, las GPUs modernas dedican la mayor parte de sus transistores a crear unidades de procesamiento más que al control de flujos de datos. Esto proporciona a la GPU la capacidad de ejecutar tanto tareas de renderizado de imágenes como programas de propósito general, todo ello de forma paralela.

Figura 1.2: Comparación Arquitectura CPU y GPU.

En la actualidad cualquier algoritmo que se implemente en una CPU también puede ser implementado en una GPU. Ambas implementaciones no serán igual de eficientes para cada arquitectura, de ahí que se haga una comparativa entre las dos. Los algoritmos a ejecutar en una GPU van a ser versiones paralelas de los que se ejecutan en la CPU. En concreto, los algoritmos con un alto grado de paralelismo, sin necesidad de estructuras de datos complejas, y con una alta intensidad aritmética, son los que mayores beneficios obtienen de su implementación en la GPU.

Muchas aplicaciones que trabajan sobre grandes cantidades de datos se pueden adaptar a un modelo de programación paralela, con el fin de incrementar el rendimiento en velocidad de las mismas. La mayor parte de ellas tienen que ver con el procesamiento de imágenes y datos multimedia. Existen una serie de APIs específicas para llevar a cabo la labor de programar en la GPU como por ejemplo Microsoft Direct Compute, CUDA de NVIDIA u OpenCL del Grupo Khronos.

Se hace patente la importancia de la GPU para nuestro proyecto ya que vamos a utilizar un programa que contiene grandes cantidades de operaciones con imágenes. Para llevar a cabo la implementación utilizamos la API OpenCL.

1.4. OpenCL

1.4.1. Introducción a OpenCL

OpenCL [11] es el primer estándar de programación en paralelo multiplataforma para plataformas consistentes en una CPU, una GPU y otros procesadores ya sea en computadoras personales, dispositivos de mano/empotrados o servidores. Incluye un lenguaje para escribir *kernels* (funciones que se ejecutan en los dispositivos OpenCL) y APIs utilizadas para definir y controlar las plataformas.

La computación en paralelo que ofrece utiliza el paralelismo de tareas y de datos. OpenCL es un estándar abierto mantenido por el consorcio sin ánimo de lucro Grupo Khronos. Ha sido adoptado por Intel [12], AMD (*Advanced Micro Devices*) [13], Nvidia [14], y ARM *Holdings* [15].

OpenCL da acceso a cualquier aplicación a la unidad de procesamiento gráfico para computar procesos no basados en gráficos extendiendo el poder de cómputo de la unidad de procesamiento gráfico más allá de los gráficos. Cabe destacar que OpenCL se puede utilizar en procesadores de FPGAs, lo que ha extendido de forma considerable su uso en la comunidad científica y académica. [4][5]

En general hay múltiples maneras de implementar un algoritmo dado en OpenCL y estas implementaciones pueden comportarse de forma muy distinta según la arquitectura que tenga el dispositivo que va a llevar a cabo la computación. Nosotros hemos seguido el camino que ofrece NVIDIA en su documentación para la implementación de nuestro algoritmo.

Un **host** es el procesador que ejecuta el sistema operativo que puede controlar los dispositivos. La CPU es el único tipo de *host*.

Un **dispositivo** (*device*) es un componente controlado por el *host*. Ejemplo de dispositivos son la CPU, GPU o los dispositivos de aceleración como el Cell Broadband Engine.

Los **contextos** representan un entorno en que los *kernels* se pueden ejecutar. También representan el dominio en el que se define la gestión de la sincronización y la memoria. Las colas de comandos permiten comandos relacionados con la gestión de memoria de sincronización y la ejecución del *kernel*. Estos últimos, se ponen en cola a través del *host* utilizando la API de OpenCL y eliminados de la cola internamente en tiempo de ejecución. Una vez que los

comandos son procesados por una combinación de recursos del *host* y el dispositivo se quitan de la cola.

Los **objetos** de memoria son una abstracción que nos permite un almacenamiento de datos en todos los dispositivos compatibles. Lee y escribe, desde y hacia los objetos de memoria para ponerlos en la cola de comandos (en el *host*) o activa disparadores por la función async_work_group_copy (en el *device*).

Los **programas** de OpenCL son un grupo de funciones regulares compiladas y funciones del *kernel*. Cada uno de los cuales se pueden ejecutar en el dispositivo.

Los *kernels* son funciones especiales en un programa que pueden ser encoladas en la cola de comandos por el *host* y se ejecutadas en el *device*. Son puntos de entrada a un programa de OpenCL. Este uso del *kernel* es distinto de un *kernel* de un sistema operativo. El uso del *kernel* que se está utilizando se deriva de la utilización de la palabra núcleo en el procesamiento de imágenes. En el procesamiento de imágenes un núcleo es una pequeña matriz de píxeles, que se utiliza como un operador durante la convolución de la imagen. En este sentido, los *kernels* de OpenCL pueden ser descritos como operadores que transforman los datos, en paralelo.

Las aplicaciones OpenCL se ejecutan en colecciones de procesadores heterogéneos, típicamente en un *host* CPU y una o varias GPUs. La clave consiste en mantener todos los procesadores ocupados y más en concreto, ocupados en lo que mejor pueden procesar. Ahí es donde los datos deben ser transferidos entre los procesadores y, para algunas aplicaciones, estas transferencias pueden marcar una fracción significativa del tiempo total de ejecución. Por lo tanto aparece la necesidad de minimizar estas transferencias u ocultarlas solapándolas con ejecuciones de *kernel*.

Para minimizarlas hay que favorecer el acceso coalescente [19 Sección 3.2.1] a memoria de los *threads*. Para ocultaras, por ejemplo, cuando uno o todos los *threads* de un *warp* [Explicado en la sección 1.4.4] realizan accesos a memoria, el *warp* completo pasa a segundo plano y se continúan ejecutando los siguientes *warps* planificados ocultando así el acceso a memoria.

Cabe destacar que una aplicación OpenCL es capaz de ejecutarse sobre un amplio abanico de arquitecturas (NVIDIA, ATI, etc) sin tener que cambiar su configuración. Su poder de adaptación es tal que se obtiene alto rendimiento en todas ellas [22].

Hay cinco pasos principales para ejecutar un cálculo de OpenCL [27]:

1. Inicialización: Selección de un dispositivo y crear un contexto en el que ejecutar el cálculo.

```
cl_int err;
cl_context context;
cl_device_id devices;
cl_command_queue cmd_queue;

err = clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &devices, NULL);
context = clCreateContext(0, 1, &devices, NULL, NULL, &err);
cmd_queue = clCreateCommandQueue(context, devices, 0, NULL);
```

Figura 1.3: Inicialización.

2. Asignar los recursos: La asignación de la memoria / almacenamiento que se utilizará en el *device* y volcarlo hacia el dispositivo.

```
cl_mem ax_mem = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                atom_buffer_size, NULL, NULL);

err = clEnqueueWriteBuffer(cmd_queue, ax_mem, CL_TRUE, 0,
                           atom_buffer_size, (void*)ax, 0,NULL,NULL);
clFinish(cmd_queue);
```

Figura 1.4: Asignación de recursos.

3. Creación de programas / *kernels*: Programas y *kernels* leen el código fuente y compilado o cargado como binario.

```
cl_program program[1];
cl_kernel kernel[1];

program[0] = clCreateProgramWithSource(context,1,
                           (const char**)&program_source, NULL, &err);

err = clBuildProgram(program[0], 0, NULL, NULL, NULL, NULL);
kernel[0] = clCreateKernel(program[0], "mdh", &err);
```

Figura 1.5: Creación de *kernels*.

4. Ejecución: se establecen los argumentos al *kernel* y se ejecutan todos los datos.

```
size_t global_work_size[2], local_work_size[2];
global_work_size[0] = nx; global_work_size[1] = ny;
local_work_size[0] = nx/2; local_work_size[1] = ny/2;

err  = clSetKernelArg(kernel[0],  0, sizeof(cl_mem), &ax_mem);

err = clEnqueueNDRangeKernel(cmd_queue, kernel[0], 2, NULL,
                             &global_work_size, &local_work_size,
                             0, NULL, NULL);
```

Figura 1.6: Ejecución.

5. Recoger: Es la parte del proceso que devuelve los resultados a la vuelta y limpiar la memoria.

```
err = clEnqueueReadBuffer(cmd_queue, val_mem, CL_TRUE, 0,
                          grid_buffer_size, val, 0, NULL, NULL);

clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleaseContext(context);
```

Figura 1.7: Recoger.

1.4.2. Un modelo de programación escalable

Para llevar a cabo esta tarea vamos a utilizar el modelo de programación paralela que se ofrece en la documentación de CUDA para OpenCL, que esta diseñado para afrontar el problema desde tres puntos clave: jerarquía de grupos de *threads*, jerarquía de memorias compartidas y sincronización de barreras.

Estos tres puntos proporcionan paralelismo de grano fino para datos y paralelismo de *thread*, anidado dentro de paralelismo de grano grueso y paralelismo de tareas. Este modelo nos ha guiado a la hora de particionar nuestro problema en subproblemas de grano grueso que puedan ser resueltos independientemente en bloques paralelos de *threads (work-groups)* y así cada subproblema pueda resolverse cooperativamente en paralelo entre las *threads* de un mismo bloque.

Este método proporciona escalabilidad automática ya que cada bloque de *threads* puede ser planificado en cualquiera de los procesadores actuales disponibles, ya sea de forma concurrente o secuencial, de manera que una aplicación OpenCL puede ser ejecutada con cualquier número de procesadores como ilustramos en la siguiente imagen. Lo único que necesita saber el sistema es el número de procesadores físicamente disponibles.

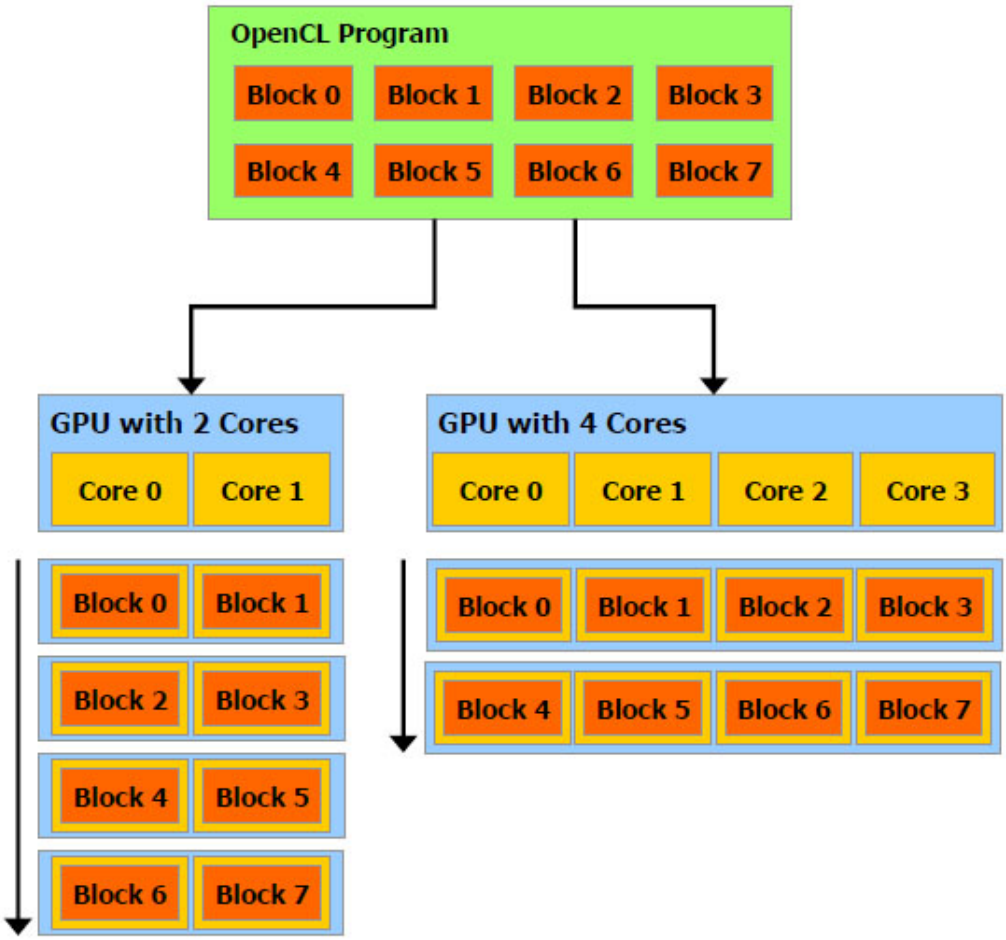


Figura 1.8: La aplicación OpenCL se adapta a los núcleos disponibles.

Como podemos ver en la Figura 1.8, un programa multihilo se particiona en bloques de hilos de forma que se pueden ejecutar independientemente en cualquier procesador de forma que una GPU con más núcleos ejecutará el programa automáticamente en menos tiempo que otras con menos núcleos.

1.4.3. OpenCL en la arquitectura CUDA

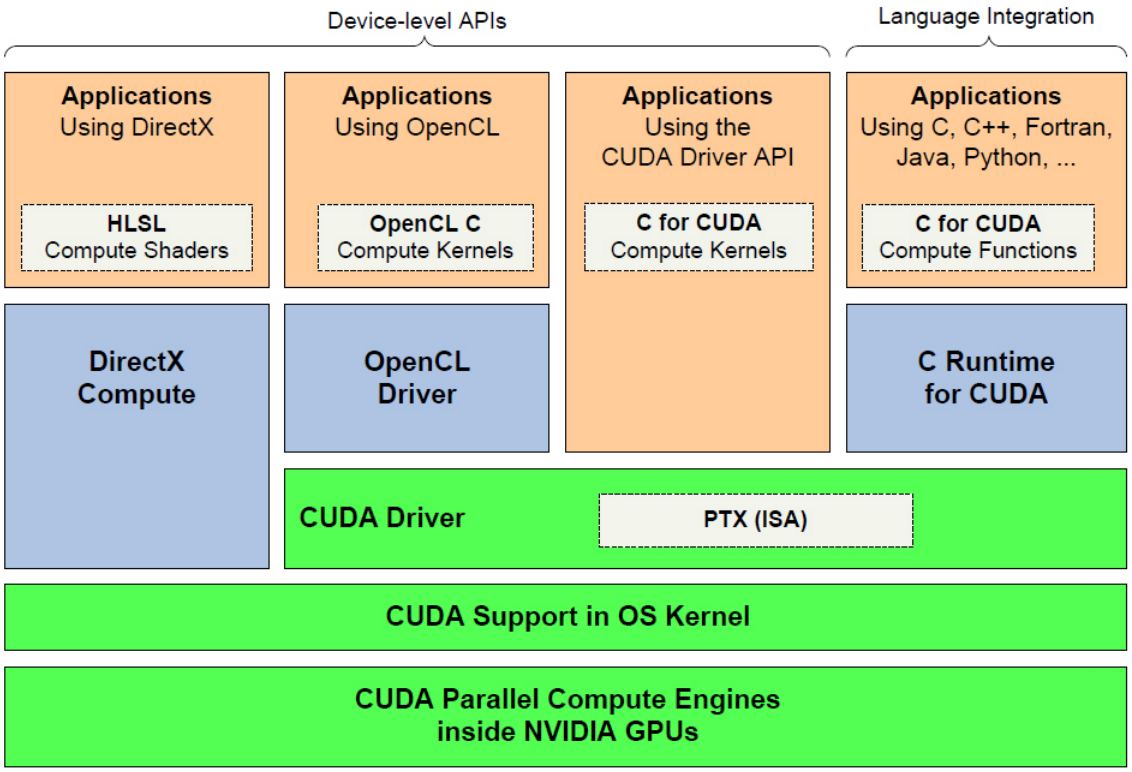


Figura 1.9: OpenCL se puede lanzar sobre el driver provisto por CUDA [17].

En nuestro caso OpenCL se va a ejecutar sobre arquitectura NVIDIA: La arquitectura CUDA es similar a la arquitectura de OpenCL, aun así los conceptos y los nombres difieren entre ambas. Un dispositivo CUDA está construido alrededor de un array escalable de procesadores que trabajan en paralelo (*Stremaing Multiprocessors*). En cambio para OpenCL un multiprocesador corresponde con el concepto de unidad de cómputo.

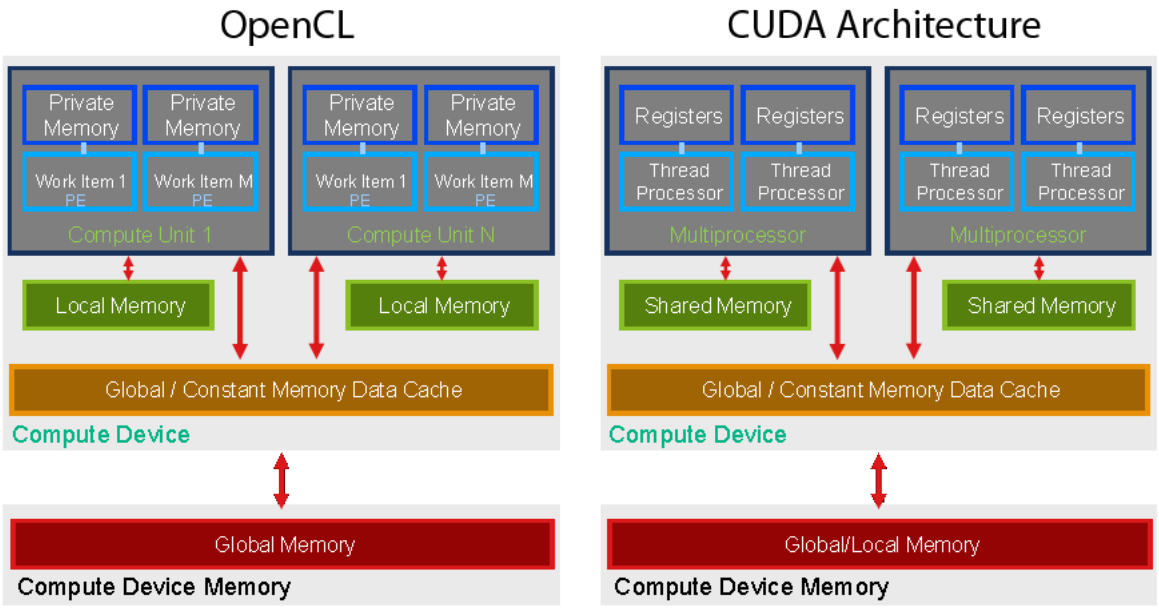


Figura 1.10: Mapeado de OpenCL en la arquitectura CUDA.

Un multiprocesador ejecuta en CUDA un *thread* para cada *work-item* de OpenCL y un bloque de *threads* para cada *work-group* o grupo de trabajo de OpenCL. Un *kernel* se ejecuta sobre un rango N-dimensional (*NDRange*) por un conjunto de unidades de cómputo que ejecutan bloques de *threads*. Todo esto lo podemos observar en la siguiente imagen donde cada uno de los bloques de *threads* que ejecutan un *kernel* es identificada por su ID de *work-group* y cada *thread* por su ID global o por una combinación de su ID local y su ID de *work-group*.

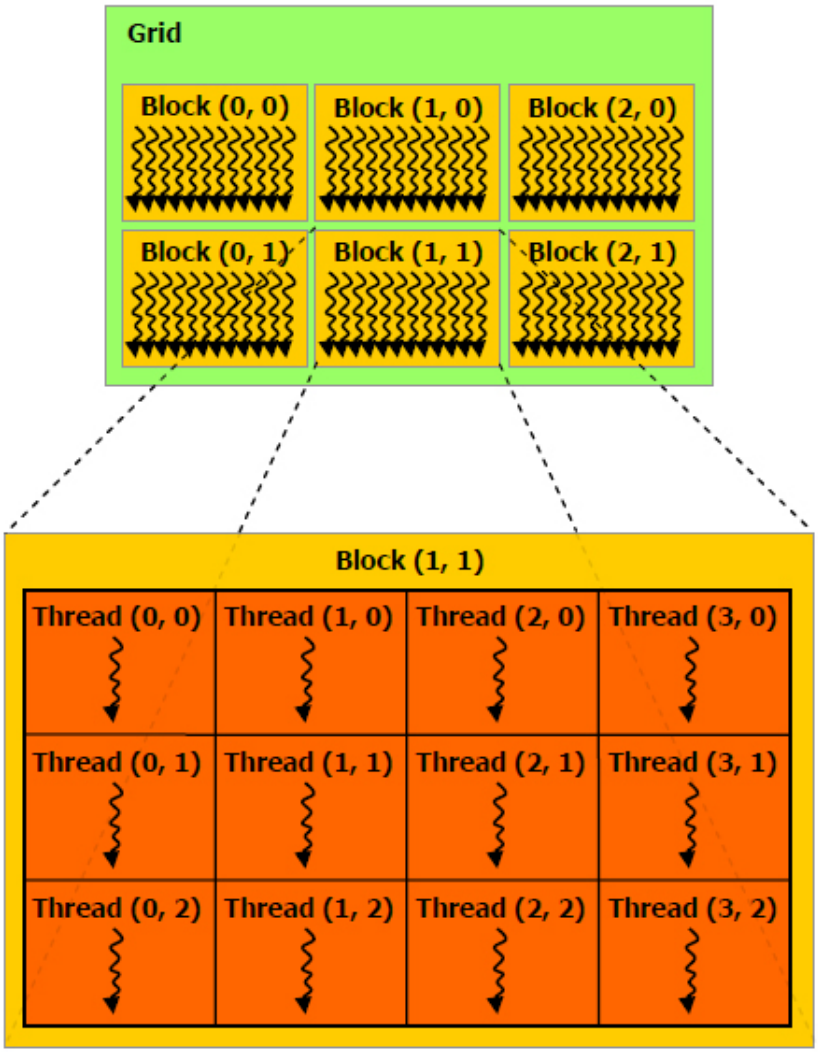


Figura 1.11: Un *kernel* es ejecutado sobre un rango N-dimensional (*NDRange*) por un conjunto de unidades de cómputo que ejecutan bloques de *threads*

Cuadrícula de bloques de *thread* A cada hilo se le asigna un ID único dentro de su bloque. El ID local de un hilo y su ID se relacionan de forma directa: Para un bloque unidimensional, son iguales; para un bloque bidimensional (Dx, Dy), el ID del *thread* en la posición (x,y) será (x + y·Dx); para un bloque de tamaño tridimensional (Dx, Dy, Dz), el ID del *thread* en la posición (x,y) vendrá dado por (x + y·Dx + z·Dx·Dy).

Cuando la aplicación OpenCL invoca un *kernel* desde el host, los work-groups son enumerados y distribuidos como bloques de *threads* a los multiprocesadores con disponibilidad de ejecución. Los hilos de un bloque de *thread* se ejecutan de forma concurrente en un multiprocesador. Cuando el bloque de *thread* ha terminado se lanzan nuevos bloques en los

multiprocesadores que han quedado disponibles.

Un multiprocesador esta diseñado para ejecutar cientos de *threads* de forma concurrente. Para manejar tal cantidad de *threads*, emplea una arquitectura única llamada SIMT (Single-Instruction, Multiple-*thread*) [16].

1.4.4. Arquitectura SIMT

El multiprocesador crea, maneja, planifica y ejecuta grupos de 32 *threads* en paralelo. Estos grupos se denominan *warps*. Las *threads* que componen un mismo *warp* comienzan con el mismo contador de programa del código que se está ejecutando, pero cada una tiene su propio contador de instrucción y su propio registro de estado por lo tanto son libres de realizar saltos y ejecutarse de forma independiente.

Cuando se lanza uno o más bloques de *threads* a un multiprocesador para su ejecución lo primero que hace es particionarlo en nuevos *warps* que son planificados por el planificador de *warp* para su ejecución. La forma en la que un bloque de *thread* es particionado es siempre la misma, cada *warp* contiene *threads* de forma consecutiva incrementando los IDs de los hilos, así por ejemplo el primer *warp* contiene el *thread* 0.

1.5. Efficient Large-Scale Stereo Matching (ELAS)

1.5.1. Introducción

El algoritmo que hemos escogido para acelerar con OpenCL es ELAS (*Efficient Large-Scale Stereo Matching*) [18]. Hemos escogido este por varias razones, se trata de un algoritmo actual (la última actualización es del 14/03/2012) de licencia GNU GPL y en su *paper* se compara con otros algoritmos de la tabla de Middlebury cuya implementación está disponible de forma pública.

La mayoría de algoritmos disponibles se centran en imágenes de baja resolución, en cambio, ELAS se centra en tratar dichas imágenes y las de alta resolución intentando mantener una eficiencia satisfactoria. En las figuras 1.12 y 1.13 podemos ver que, en comparación con otras implementaciones, para imágenes de alta resolución el algoritmo se comporta bien en

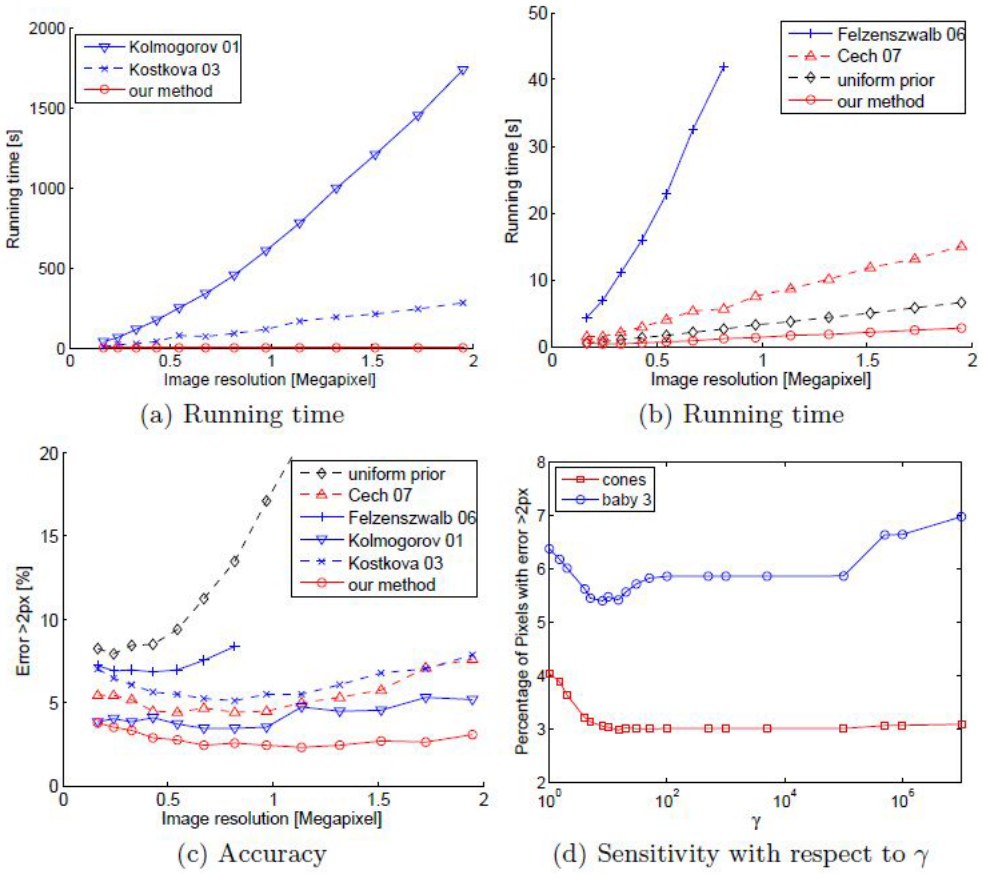


Figura 1.12: Comparación gráfica de ELAS con otros algoritmos.

tiempo de ejecución y los niveles de error se mantienen más o menos idénticos según sube la resolución.

La implementación ELAS viene optimizada para CPU con instrucciones multimedia SSE [29]. Este código ejecutado en un procesador de última generación, proporciona niveles de tiempo de procesado de la escena 3D cercanos a tiempo real. Dado que obtener tal procesador en la actualidad es bastante caro, nuestro objetivo es optimizarlo para GPU y demostrar que es posible conseguir los mismos resultados o incluso mejores sin necesidad de procesadores de última generación.

Ya que la implementación viene optimizada con SSE, la traduciremos a código C plano, lo que añadirá un recurso más a la hora de analizar el algoritmo.

	Cones	Teddy	Art	Aloe	Dolls	Baby3	Cloth3	Lamp2	Rock2
Image width	900	900	1390	1282	1390	1312	1252	1300	1276
Image height	750	750	1110	1110	1110	1110	1110	1110	1110
Support points	3236	3095	6164	6268	8241	5901	6805	4424	6670
Correct points	99.2%	99.1%	99.1%	99.8%	99.4%	98.7%	100.0%	98.9%	100.0%
Triangles	6376	6128	12237	12417	16353	11689	13473	8769	13215
Missed pixels	0.7%	4.0%	5.3%	1.6%	1.1%	1.0%	0.4%	9.7%	0.6%
Non-occluded pixels: Error > 1									
uniform prior	18.0%	37.5%	43.0%	12.8%	33.1%	49.4%	7.6%	74.9%	7.8%
Felzenszwalb 06	15.2%	18.7%	23.3%	12.8%	20.9%	13.0%	6.1%	32.0%	7.6%
Kolmogorov 01	8.2%	16.5%	30.3%	13.5%	28.7%	26.2%	4.3%	65.7%	10.4%
Cech 07	7.2%	15.8%	18.8%	9.2%	19.8%	17.4%	2.8%	36.7%	3.6%
Kostkova 03	7.2%	13.5%	17.9%	7.2%	14.4%	14.2%	2.7%	31.5%	3.0%
our method	**5.0%**	**11.5%**	**13.3%**	**5.0%**	**11.0%**	**10.8%**	**1.4%**	**17.5%**	**1.9%**
Non-occluded pixels: Error > 2									
uniform prior	16.4%	35.0%	41.1%	11.3%	29.6%	46.9%	7.3%	74.2%	7.3%
Felzenszwalb 06	7.8%	11.4%	16.5%	7.8%	10.5%	7.0%	3.5%	26.0%	3.1%
Kolmogorov 01	4.1%	8.1%	21.0%	8.1%	17.0%	19.0%	1.8%	60.7%	6.0%
Cech 07	4.4%	10.2%	11.2%	4.8%	10.6%	9.7%	1.8%	27.1%	2.1%
Kostkova 03	5.3%	10.1%	13.0%	4.8%	8.2%	8.2%	2.2%	26.7%	2.2%
our method	**2.7%**	**7.3%**	**8.7%**	**3.0%**	**5.3%**	**4.5%**	**0.9%**	**10.4%**	**1.0%**

Figura 1.13: Tablas comparativas de ELAS con otros algoritmos.

1.5.2. Explicación del algoritmo

El algoritmo ELAS propone un modelo probabilista generativo para la búsqueda de correspondencia estereoscópica. Este modelo realiza una correspondencia densa utilizando pequeñas ventanas de agregación reduciendo las ambigüedades en las correspondencias.

El algoritmo construye un antecedente en el plano de la disparidad formando una triangulación que utiliza como vértices un conjunto de correspondencias calculadas con alto grado de fidelidad o robustez llamados "*support-points*" (puntos de soporte). Con este antecedente se restringe la búsqueda de correspondencia para el resto de puntos (que no son robustos) a zonas plausibles.

En particular este antecedente se forma al computar por secciones una función lineal inducida por las disparidades de los *support-points* y la malla creada por la triangulación. Además se asume que las imágenes introducidas están rectificadas por lo que la correspondencia se da en la misma línea en las dos imágenes.

Así la taxonomía [3] del algoritmo es la siguiente:

Método	ELAS
Coste de Correspondencia	Diferencia Media Absoluta (MAD)
Coste de Agregación	Región de Soporte Tridimensional
Computación de la Disparidad	Estimación Bayesiana Modal (MAP)
Refinamiento de la Disparidad	Comprobación de Consistencia Izq / Der

1.5.3. Support Points (Puntos de Soporte)

Para escoger los *support-points* seleccionaremos píxeles que pueden ser encontrados con firmeza debido a su textura y su singularidad. Aparece el concepto de "*easy*" (sencillo) y "*hard*" (complejo) aplicado a cada punto de las imágenes. El criterio que determina si un punto es sencillo y por tanto robusto es la consistencia, es decir, las correspondencias tienen que poder ser encontradas de izquierda a derecha y de derecha a izquierda.

Para deshacerse de las correspondencias ambiguas, se eliminan todos los puntos cuyo ratio entre la mejor y la segunda mejor correspondencia exceda el parámetro inicial $\tau = 0.9$ (veremos los parámetros más adelante). Las falsas correspondencias son eliminadas descartando puntos que presentan una disparidad distinta que la de sus vecinos y, para cubrir toda la imagen, se añaden *support-points* en las esquinas de las imágenes, cuyas disparidades son tomadas de las de sus vecinos.

Figura 1.14: Punto sencillo (easy) y punto complejo (hard).

Por ejemplo, los bordes serán puntos de correspondencia sencillos ya que se puede hallar su correspondencia de forma fácil con el filtro de Sobel [2], mientras que los puntos de dentro de

una región serán puntos de correspondencia complejos ya que los puntos vecinos serán muy parecidos y hallar la correspondencia de ese punto en la otra imagen será difícil.

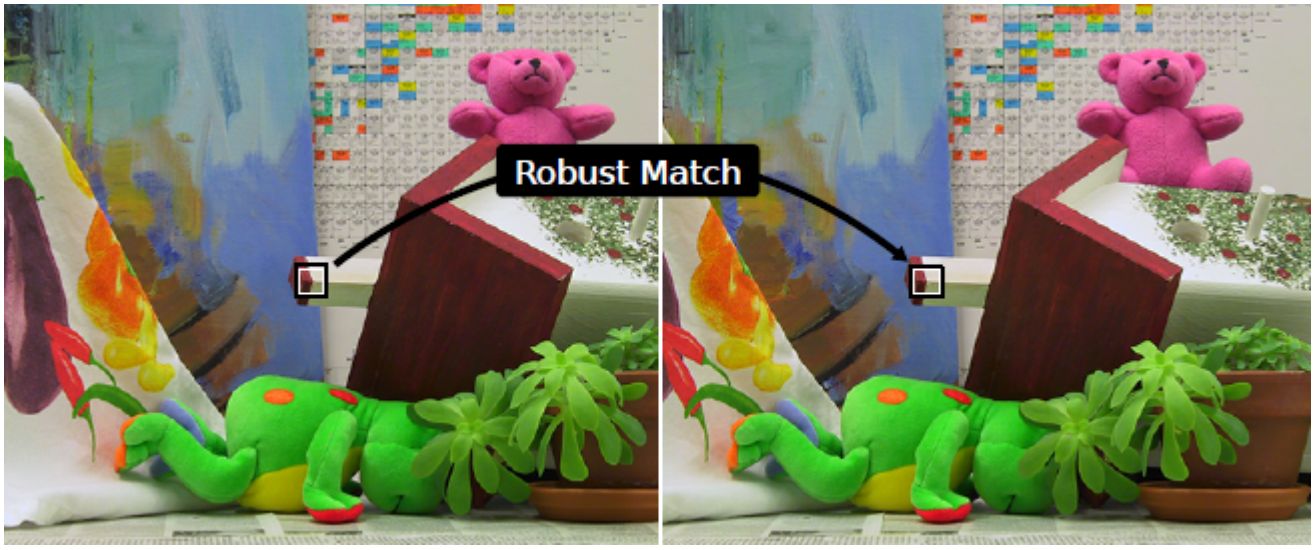


Figura 1.15: Correspondencia robusta del mismo punto en las dos imágenes.

Una vez encontrados el conjunto de *support-points* construiremos la malla 2D donde buscar la correspondencia de los puntos complejos. Esta malla se construye por triangulación de Delaunay [10] y se utiliza para interpolar las disparidades entre los puntos sencillos generando una malla para cada imagen.

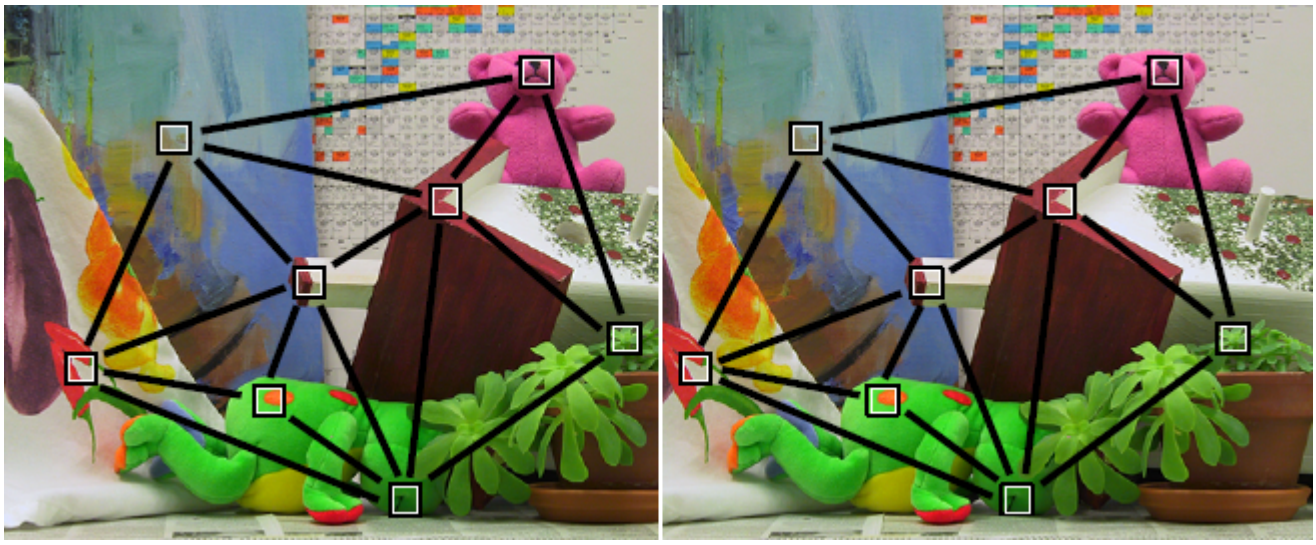
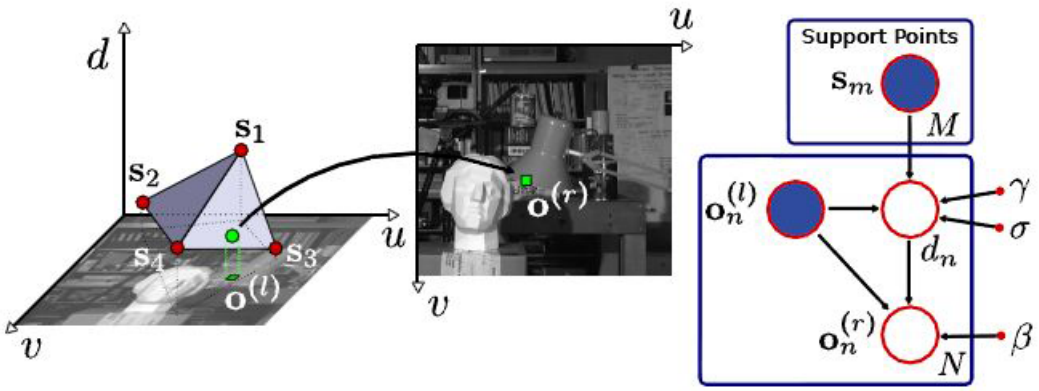


Figura 1.16: Mallas 2D obtenidas por triangulación de Delaunay [10].

1.5.4. Modelo Probabilista Generativo para la correspondencia

Dada una imagen de referencia y los *support-points*, se utiliza el modelo generativo para dibujar la otra imagen. Dado un conjunto $\mathbf{S}$ $\{\mathbf{s}_1,..., \mathbf{s}_M\}$ de *support-points* robustos y un conjunto $\mathbf{O} = \{\mathbf{o}_1, ..., \mathbf{o}_N\}$ de observaciones de la imagen, se asume que las observaciones $\mathbf{O}$ y los *support-points* $\mathbf{S}$ son condicionalmente independientes. Dadas las disparidades de los *support-points*, se formaliza una distribución conjunta formada por un antecedente y una probabilidad característica de la imagen.

En el *paper* original [18] se describe con más en detalle este modelo. Una ventaja de este modelo generativo, es que dados los *support-points* y las observaciones de la imagen izquierda, se pueden obtener muestras de la imagen derecha. Esto puede contemplarse gráficamente en la Figura 1.17.



(a) Sampling process and graphical model

(b) Left image (c) Sample mean (d) Right image

Figura 1.17: (a) Modelo gráfico y proceso de toma de muestras: dado un conjunto de *support-points* $\{\mathbf{s}_1,..., \mathbf{s}_M\}$, para una observación en la imagen de la izquierda (*left*) $\mathbf{o}_n^{(l)}$ se dibuja una disparidad d. Dada dicha observación en la imagen izquierda y la disparidad, podemos dibujar la observación correspondiente en la imagen derecha (*right*) $\mathbf{o}_n^{(r)}$. Repitiendo este proceso 100 veces por cada píxel (d) la computación de la media da como resultado una imagen borrosa (c).

1.5.5. Estimación del mapa de disparidad

Este modelo generativo se utiliza a la hora de estimar el mapa de disparidad. Para ello se utiliza el antecedente y la probabilidad característica de la imágen en una función de energía cuya utilidad reflejaremos en este apartado.

Llegados a este punto hay que aclarar un par de conceptos al lector. El mapa de disparidad de la imagen derecha se obtiene a partir de los *support-points* y las observaciones de la imagen izquierda y con la imagen izquierda ocurre lo mismo pero utilizando los *support-points* y las observaciones en la imagen derecha. Una vez aclarado esto, continuamos suponiendo que queremos hallar la disparidad de la imagen derecha.

Para hallar el mapa de disparidad dadas ambas imágenes, el algoritmo se basa en una estimación *maximum a-posteriori* (MAP) (estimación bayesiana modal en castellano) para computar la disparidad

$$d_n^* = \text{argmax } p(d_n | o_n^{(izq)}, o_1^{(der)}, ..., o_N^{(der)}, \text{S})$$

donde $o_1^{(der)}$, ..., $o_N^{(der)}$ son todas las observaciones de la imagen derecha localizadas en la linea epipolar de $o_n^{(izq)}$.

Las observaciones a lo largo de la linea epipolar de la imagen derecha estan estructuradas de la siguiente forma. Dada una disparidad asociada con $o_n^{(izq)}$, hay un mapa determinista para el cual las observaciones tienen una probabilidad mayor que cero en dicha linea. Así el algortimo captura esta propiedad modelando la distribución sobre todas las observaciones a lo largo de la línea epipolar de la forma:

$$p(o_1^{(der)}, ..., o_N^{(der)} |\ o_n^{(izq)}, d_n) \propto \sum_{i=1}^{N} p(o_i^{(der)} | o_n^{(l)}, d_n)$$

Una vez comprendida toda la información con la que se va a trabajar, vamos a describir finalmente cómo se calcula la disparidad de un punto.

Dado un píxel de la imágen de referencia (en este caso la izquierda), la disparidad de ese punto en la imágen objetivo (imagen derecha) se calcula de la siguiente forma:

1. Nos situamos en la coordenada (x, y) de la imagen objetivo, donde (x, y) son las coordenadas del píxel de la imagen de referencia.

2. Recorremos la linea epipolar desde ese punto hasta la disparidad máxima. Según cual sea la imagen objetivo, la derecha o la izquierda, recorreremos incrementando o decrementando las coordenadas, ya que el píxel correspondiente se encontrará a un lado u otro.

3. La forma de comparar es la siguiente:

    a) Se suma el contenido del Descriptor [Sección 2.3.1, Figura 2.1] para el píxel i de la imagen objetivo.

    b) Se realiza una resta en valor absoluto entre el resultado obtenido en el apartado $3a$ y la suma del descriptor del píxel de la imagen de referencia.

    c) Por último, se seleccionará la disparidad i para la que el resultado de la diferencia en valor absoluto ha dado el número más cercano a 0. Esto es lo que se conoce como función de energía y viene explicada más detalladamente en el *paper* original [18, Sección 3.3 apartado (8)].

Es importante tener en cuenta que esta operación se puede aplicar perfectamente en paralelo para cada píxel ya que los *support-points* desacoplan las diferentes observaciones.

Por último, se calculan las disparidades en el sentido contrario (si se calcula el mapa de disparidad izquierda-derecha, entonces después se calcula el mapa de disparidad derecha-izquierda), y se realiza una comprobación de coherencia para eliminar falsas correspondencias y disparidades en regiones ocultas.

1.5.6. Pseudo código

Algoritmo 1.1 Pseudocódigo ELAS

```
void Elas{
  Reservar memoria
  Inicialización de variables
  Descriptor desc1(...);
  Descriptor desc2(...);

  vector<support_pt> p_support = computeSupportMatches(desc1,desc2);

  vector<triangle> tri_1 = computeDelaunayTriangulation(p_support,izquierda);
  vector<triangle> tri_2 = computeDelaunayTriangulation(p_support,derecha);

  computeDisparityPlanes(p_support,tri_1,izquierda);
  computeDisparityPlanes(p_support,tri_2,derecha);

  createGrid(p_support,disparidad_cuadricula_1,dimension_cuadricula,izquierda);
  createGrid(p_support,disparidad_cuadricula_2,dimension_cuadricula,derecha);

  computeDisparity(p_support,tri_1,disparidad_cuadricula_1,
                   dimension_cuadricula,desc1,desc2,izquierda,Disparidad1);
  computeDisparity(p_support,tri_2,disparidad_cuadricula_2,
                   dimension_cuadricula,desc1,desc2,derecha,Disparidad2);

  leftRightConsistencyCheck(Disparidad1,Disparidad2);

  removeSmallSegments(D1);
  si (¬postprocesar_solo_la_imagen_izquierda)
      removeSmallSegments(D2);

  gapInterpolation(D1);
  si (¬postprocesar_solo_la_imagen_izquierda)
    gapInterpolation(D2);

  si (filtro_de_media_adaptativa) {
    adaptiveMean(D1);
    si (¬postprocesar_solo_la_imagen_izquierda)
    adaptiveMean(D2);
  }
  si (filtro_mediana) {
    median(D1);
    si (¬postprocesar_solo_la_imagen_izquierda)
      median(D2);
  }
  Liberar memoria
}
```

1.5.7. Flow chart

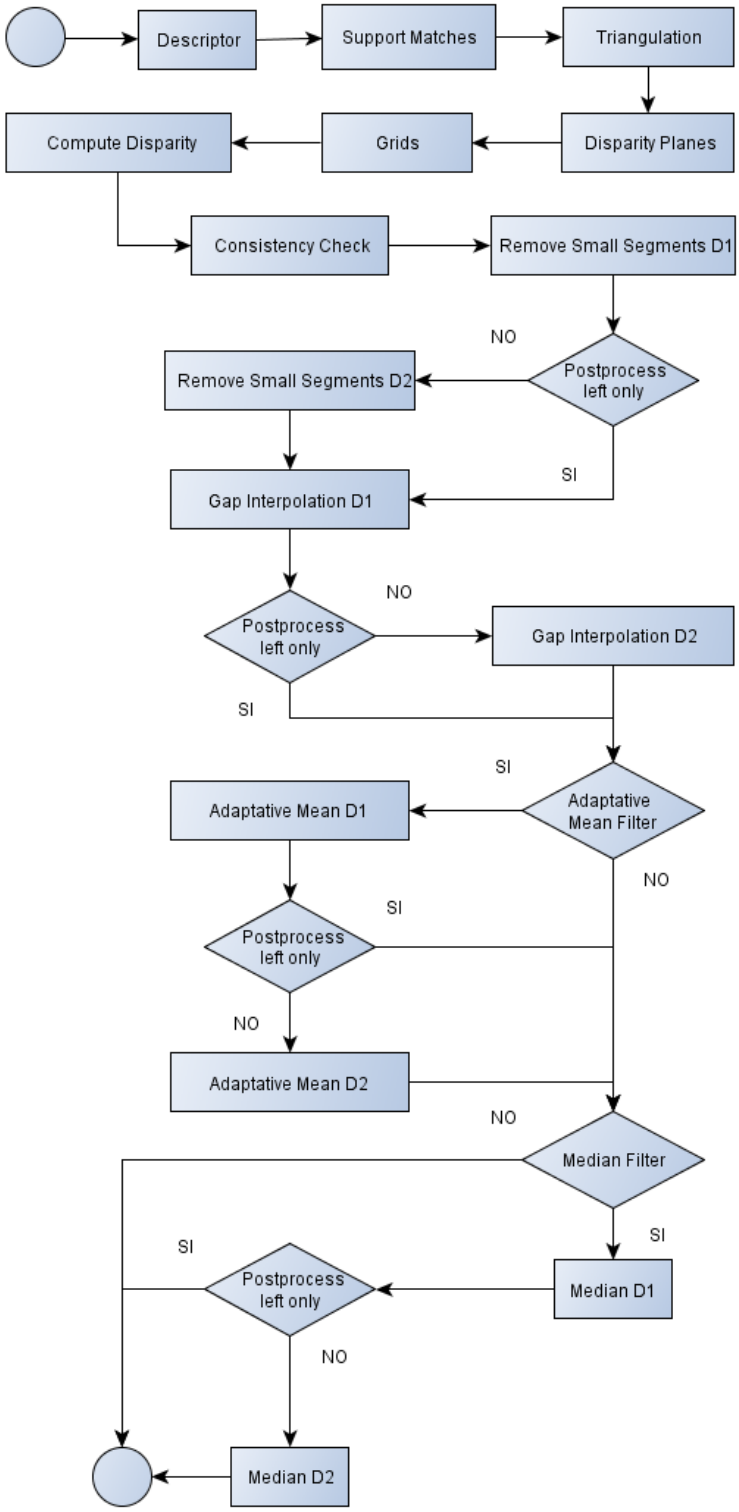


Figura 1.18: Flow chart

Capítulo 2

Metodología

En esta sección vamos a describir la metodología que hemos seguido a lo largo de nuestro proyecto para demostrar que se puede ejecutar el algoritmo ELAS en tiempo real.

Con el objetivo de acelerar la implementacion original del algoritmo, comenzamos por detectar cuales eran los puntos mas intensos computacionalmente para después rediseñarlos e implementarlos bajo el paradigma paralelo de OpenCL.

Hemos impuesto como condición que los resultados de ambas implementaciones sean exactamente los mismos. Por ello, a lo largo del codigo hay bloques que se encargan de generar archivos de salida con los datos intermedios. Estos datos son generados con la implementación original y con la optimizada. Para que estos bloques se ejecuten hay que definir al macro DUMPS.

A la hora de plantearnos la forma de diseñar, tuvimos que considerar sobre qué tipo de plataforma íbamos a basar el diseño. No es lo mismo paralelizar para una CPU Multicore (aproximadamente 8 cores) que para una GPU, que tiene cientos de cores [22].

Nuestro diseño esta basado en plataformas con más de 16 cores, ya que nuestro objetivo es lanzar un kernel por cada píxel. Así, si se lanza el programa en una plataforma capaz de ejecutar tantos kernel como píxeles, la paralelización sería óptima.

2.1. Elección de puntos a optimizar

Para realizar una optimización útil, lo primero que hemos hecho ha sido realizar un *profiling* del algoritmo para tratar de encontrar los puntos que presentaban un mayor tiempo de

ejecución. Para ello ejecutamos y medimos los tiempos de ejecución para cada función del algoritmo en distintos ordenadores. Los resultados de los tiempos para las tres imagenes (adjuntas con el proyecto) se pueden observar en los siguientes cuadros.

Funciones	Core i5 M450 2.40GHz	Intel Core 2 Duo T7250 2GHz	Intel Core 2 Quad Q9400 2.66GHz x 4
Descriptor	32.7 ms	81.2 ms	40.8 ms
Support Matches	166.7 ms	312.9 ms	100.5 ms
Delaunay Triangulation	9.0 ms	20.5 ms	9.3 ms
Disparity Planes	11.4 ms	30.3 ms	12.6 ms
Grid	14.5 ms	21.2 ms	11.5 ms
Matching	259.3 ms	487.6 ms	208.5 ms
L/R Consitency Check	29.9 ms	43.6 ms	22.0 ms
Remove Small Segments	59.8 ms	121.7 ms	58.9 ms
Gap Interpolation	15.1 ms	28.4 ms	12.3 ms
Adaptative Mean	91.3 ms	200.9 ms	90.5 ms
Total	**689.8 ms**	**1348.3 ms**	**567.0 ms**

Cuadro 2.1: Tiempos de ejecución - Cones

Funciones	Core i5 M450 2,40 GHz	Intel Core 2 Duo T7250 2 GHz	Intel Core 2 Quad Q9400 2.66GHz x 4
Descriptor	66.9 ms	170.0 ms	86.7 ms
Support Matches	410.0 ms	639.3 ms	253.9 ms
Delaunay Triangulation	17.3 ms	39.9 ms	18.0 ms
Disparity Planes	22.4 ms	58.1 ms	24.0 ms
Grid	29.3 ms	43.6 ms	24.3 ms
Matching	567.7 ms	1100.9 ms	446.1 ms
L/R Consitency Check	73.6 ms	84.1 ms	50.6 ms
Remove Small Segments	134.3 ms	219.6 ms	125.4 ms
Gap Interpolation	38.3 ms	58.2 ms	28.9 ms
Adaptative Mean	201.6 ms	410.9 ms	192.6 ms
Total	**1561.4 ms**	**2824.5 ms**	**1270.5 ms**

Cuadro 2.2: Tiempos de ejecución - Aloe

Funciones	Core i5 M450 2,40 GHz	Intel Core 2 Duo T7250 2 GHz	Intel Core 2 Quad Q9400 2.66GHz x 4
Descriptor	70.0 ms	169.3 ms	91.2 ms
Support Matches	414.2 ms	598.1 ms	224.9 ms
Delaunay Triangulation	18.8 ms	42.7 ms	19.2 ms
Disparity Planes	23.7 ms	63.2 ms	26.5 ms
Grid	30.4 ms	45.8 ms	24.3 ms
Matching	577.9 ms	1079.3 ms	461.0 ms
L/R Consitency Check	78.6 ms	91.5 ms	54.5 ms
Remove Small Segments	149.9 ms	238.9 ms	136.7 ms
Gap Interpolation	40.9 ms	60.4 ms	30.7 ms
Adaptative Mean	210.8 ms	431.4 ms	202.2 ms
Total	**1615.1 ms**	**2820.5 ms**	**1271.2 ms**

Cuadro 2.3: Tiempos de ejecución - Raindeer

Teniendo estos resultados y habiendo estudiado el algoritmo, hicimos las siguientes elecciones:

- Descriptor
  - Genera la estructura inicial que va a ser utilizada por puntos posteriores, es decir, es el primer paso del algoritmo.
  - Es un proceso intenso desde el punto de vista de accesos a memoria. Este es uno de los aspectos más criticos a la hora de hacer implementaciones para GPUs.
- Support Matches
  - Este punto tiene dependencia inmediata con la estructura generada por Descriptor. Así, podemos usar la memoria de la GPU y evitar transacciones con la memoria RAM del **host**.
  - Es una de las funciones que más tiempo requiere.
  - Tras estudiar el algoritmo, evaluamos como factor importante el hecho de que este punto realiza el cálculo de disparidades inicial. Su elección nos permite estudiarlo más en profundidad, aprendiendo la técnica utilizada para llevar a cabo la obtención de dicho mapa.
  - Por otro lado también lo hemos escogido por su naturaleza paralela, ya que se puede calcular la disparidad de cada píxel en paralelo.

2.2. Estrategias de optimización

Principalmente hay tres estrategias distintas a la hora de optimizar un algoritmo [19 Apéndice A.1][23]:

- Maximizar la ejecución en paralelo para alcanzar un máximo de uso de los procesadores.
- Optimizar el uso de la memoria para alcanzar un rendimiento máximo de la misma.
- Optimizar el uso de las instrucciones para maximizar el rendimiento de instrucciones.

Para maximizar la ejecución paralela se comienza estructurando el código de forma que exponga la mayor cantidad posible de datos paralelos. Una vez que el paralelismo ha sido

expuesto, necesita ser mapeado en el hardware tan eficientemente como sea posible. Esto se hace escogiendo de forma cuidadosa el *NDRange* de cada invocación al *kernel*. La aplicación también debería maximizar la ejecución en paralelo a un nivel más alto exponiendo explícitamente la ejecución concurrente en el *device* mediante streams, así como maximizar la ejecución concurrente entre *host* y *device*.

Para optimizar el uso de la memoria hay que empezar minimizando las transferencias de datos entre *host* y *device* debido a que dichas transferencias tienen mucho menos ancho de banda que las transferencias de datos en memoria interna. Los accesos del *kernel* a la memoria global deben ser minimizados maximizando el uso de la memoria compartida en el *device*. Un acceso a memoria global tiene una latencia de entre 400 y 800 ciclos mientras que la latencia de la memoria compartida es 100 veces menor. Además, el ancho de banda de la memoria local es 10 veces mayor que el ancho de banda de la memoria global. Hay ocasiones en que la mejor optimización sería incluso evitar cualquier transferencia de datos re-computando los datos cuando se necesiten en lugar de recibirlos a través de una transferencia.

El ancho de banda efectivo puede variar en un orden de magnitud dado dependiendo en el patrón de acceso para cada tipo de memoria. El siguiente paso en la optimización del uso de la memoria es organizar los accesos a la misma de acuerdo a los patrones de acceso óptimos a memoria. Esta optimización es especialmente importante para el acceso a memoria global ya que la latencia de estos accesos es de varios cientos de ciclos de reloj. Los accesos a memoria compartida, en contraposición, normalmente sólo merece la pena optimizarlos cuando existe un alto grado de conflictos de acceso a memoria.

Para la optimización en el uso de las instrucciones, las instrucciones aritméticas que tienen bajo rendimiento deberían ser evitadas. A cambio se debe utilizar funciones regulares o de precisión simple en vez de funciones de precision doble. Finalmente, se debe prestar una atención particular al control del flujo de instrucciones debido a la naturaleza SIMT de los distintos *devices*.

Así se definen tres categorías de recomendación [*19* Apéndices A.2,A.3 y A.4] separadas por su prioridad; Tácticas de alta prioridad, media y baja. A continuación vamos a citarlas y en el punto 2.3 describiremos cuales hemos aplicado a nuestro proyecto.

2.2.1. Técnicas recomendadas

Técnicas de Alta Prioridad

- Centrarse primero en encontrar formas de paralelizar el código secuencial [19, Sección 1.1.3].
- Utilizar el ancho de banda efectivo de la computación como métrica cuando se mida el rendimiento y los beneficios de la optimización [19 Sección 2.2].
- Minimizar las transferencias de datos entre *host* y *device,* incluso si ello conlleva ejecutar algunos *kernels* en el *device* que no muestran ganancia en rendimiento cuando se comparan con el rendimiento en el *host* CPU [19, Sección 3.1].
- Asegurar que los accesos a memoria son coalescentes siempre que se pueda [19, Sección 3.2.1].
- Minimizar el uso de la memoria global. Elegir accesos a memoria compartida siempre que sea posible [19, Sección 5.2].
- Evitar la ejecución de diferentes segmentos de código dentro de un mismo *warp*. Es decir, si todos los *threads* dentro de un mismo *warp* cumplen condiciones distintas en un *if*, se ejecutará dentro del *warp* el segmento de código del *else* y del *if*; y ésto hay que intentar evitarlo [19 Sección 6.1].
- Utilizar la opción de compilación -cl-mad-enable [19, Capítulo 5].

Técnicas de Media Prioridad

- Utilizar de forma prudente la memoria "pineada" [19, Sección 3.1.1].
- Donde sea posible y para aplicaciones donde sea efectivo, solapa las transferencias de memoria *host* - *device* con computación en el *device* y con actividades asíncronas del *host* [19, Seccióones 3.1.2 y 3.1.3].
- Para aplicaciones donde el resultado de los cálculos se muestran visualmente, utiliza OpenCL-OpenGL or interoperabilidad OpenCL-D3D.
- Los accesos a memoria compartida deben ser diseñados para evitar peticiones serializadas debidas a conflictos de memoria [19, Sección 3.2.2.1].

- Para ocultar la latencia que surja de las dependencias entre registros, hay que mantener al menos una ocupación del 25 % en *devices* con capabilidad de computo menor o igual a 1.1 y del 18.75 % en *device* posteriores [19, Sección 4.3].
- Utilizar la memoria compartida para evitar transferencias redundantes desde memoria global [19, Sección 3.2.2.2].
- El número de *threads* por bloque debería ser un multiplo de 32, ya que esto proporciona una computación óptima y eficiente y facilita la coalescencia [19 Sección 4.4].
- Utilizar la libreria "*math*" nativa siempre que la velocidad supere la precisión [19, Sección 5.1.4].

Técnicas de Baja Prioridad

- Para *kernels* con una larga lista de argumentos, conviene emplazar algunos argumentos en memoria constante para ahorrar en memoria compartida [19 Sección 3.2.2.4].
- Utiliza opraciones *shift* para evitar divisiones caras (en coste) y cálculos de módulo [19, Sección 5.1.1].
- Evita la conversión automática de *doubles* a *floats* [19, Sección 5.1.3].
- Haz que al compilador utilice predicción de saltos en lugar de bucles o sentencias de control [19, Sección 6.2].

Por último, decir que nosotros hemos enfocado el problema desde las dos primeras estrategias (maximizar el paralelismo y optimización de la memoria), ya que, en general, estas dos primeras estrategias resuelven la mayoría de los cuellos de botella.

2.3. Descripción arquitectural

2.3.1. Algoritmo original ELAS

En este apartado vamos a describir de forma más concreta los puntos que vamos a modificar. Cabe mencionar que ELAS trabaja con imágenes .pgm que tienen 8 bits por cada píxel.

Descriptor El *descriptor* de una imagen es una estructura auxiliar interna del algoritmo que optimiza el acceso a la vecindad de cada píxel. En esta fase se crean los descriptores de las imágenes con las que se va a trabajar.

Cada descriptor es un vector de 16 píxeles. La información de los vecinos se guarda del siguiente modo: del elemento 1-5 y 8-12 se guardan los vecinos en la imágen correspondiente al descriptor. Los elementos 6 y 7 almacenan, ambos, el píxel actual. Y los elementos 13-16 almacenan los vecinos del píxel con las mismas coordenadas de la otra imagen. En la Figura 2.1 se muestra este proceso en el caso de la imagen izquierda. Para la imagen derecha se realiza el mismo proceso pero del revés.

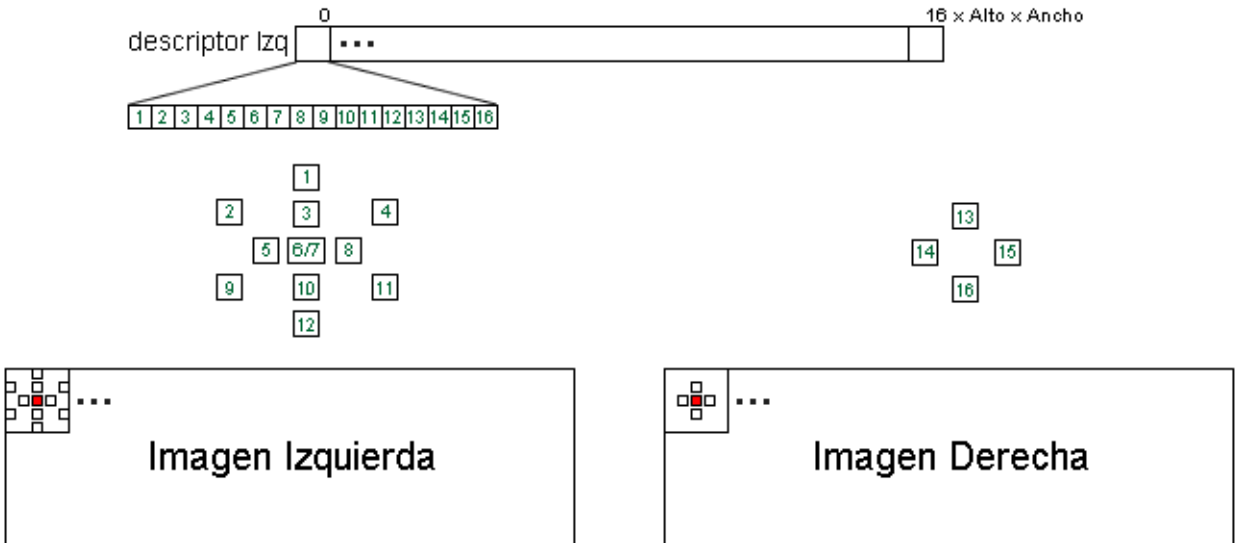


Figura 2.1: Contenido del elemento 0 del descriptor Izquierdo.

Support Points En esta función se crean los *support-points* a partir de los descriptores calculados previamente.

Se empieza creando una cuadrícula de candidatos. El ancho de cada sector de la cuadrícula viene determinado por un parámetro inicial del algoritmo, *candidate_stepsize* [Sección 2.3.2]. La cuadrícula funciona así: Supongamos *candidate_stepsize* $= 5$, entonces los píxeles cuyas coordenadas (x e y) sean múltiplo de 5, formarán los vértices o puntos de cruce de la cuadrícula.

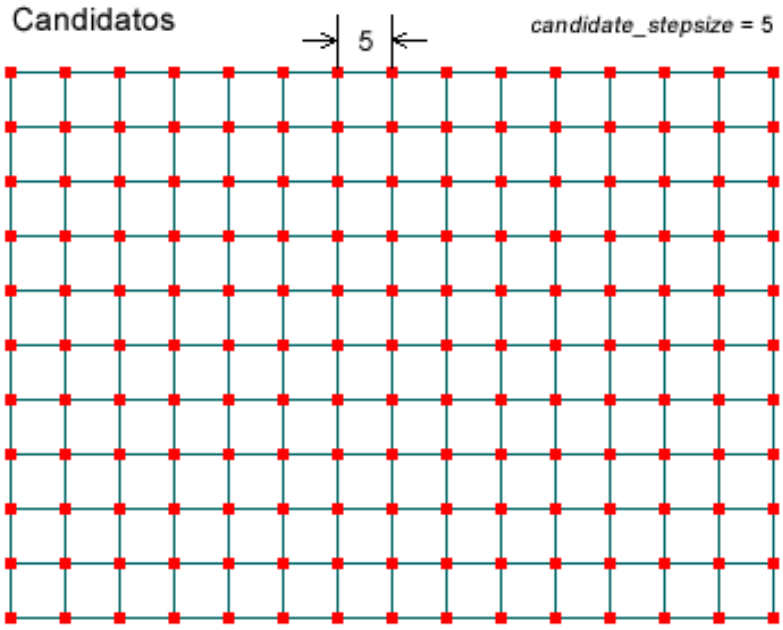


Figura 2.2: Cuadrícula de candidatos para $candidate\_stepsize = 5$.

Para cada candidato, primero se busca la disparidad del candidato de izquierda a derecha y de derecha a izquierda, obteniendo así dos disparidades. Si el valor absoluto de la diferencia entre ambas es menor o igual a la condicion inicial *lr_threshold* [Sección 2.3.2], que es el parámetro mide la consistencia Izquierda-Derecha, se toma por bueno.

Después, se computa la disparidad del modo descrito en el apartado 1.5.5.

Una vez escogidos, se eliminan lo *support-points* inconsistentes y los que aportan información redundante. Por último se transforma le representación de los *support-point* a representación vectorial, para proceder a su triangulación.

Adicionalmente, es importante destacar que este punto es más intenso, en lo que accesos a memoria respecta, que el anterior. Para comparar píxeles se evalúan los descriptores de los cuatro vecinos inmediatos a cada uno. Este proceso ha de repetirse como mucho 255 veces por píxel (el máximo valor de disparidad admitido). En conclusión, por cada píxel han de realizarse, a lo sumo, $255 \times 4 = 1020$ accesos a memoria.

2.3.2. Análisis de parámetros

El programa ELAS utiliza una serie de parámetros de entrada que sientan las condiciones iniciales sobre las que se van a realizar los cálculos. A continuación ofrecemos una breve explicación de cada parámetro junto con sus valores por defecto. En la sección **??**.

Parámetro	**Descripción**	**Valor por defecto**
[disp_min, disp_max]	Rango mínimo y máximo de disparidad. El máximo es 255 ya que el valor se almacena en un byte, cuyo valor entero máximo es 255. Este rango define el espacio sobre el que se buscará un píxel de la imagen de referencia para la imagen objetivo.	[0, 255]
support_threshold	Ratio máximo de unicidad, la mejor correspondencia de support-point contra la segunda mejor. Este ratio determina el nivel de aceptación de la función de energía	0.95
support_texture	Textura mínima para que un pixel se compute como support-points.	10
candidate_stepsize	Tamaño de la cuadrícula sobre la que se buscan los support-points.	5
incon_window_size	Tamaño de ventana para el checkeo de support-points inconsistentes.	5
incon_threshold	Umbral de similitud de disparidad para que un support-point sea considerado consistente.	5
incon_min_support	Mínimo numero de support-points inconsistentes.	5

add_corners	Booleano. Añadir support-points en las esquinas de la imagen utilizando la disparidad de los vecinos.	1
grid_size	Tamaño de la cuadrícula que se utilizará para la extrapolación adicional de support-points	20
beta	Parámetro de parecido entre las imagenes.	0.02
gamma	Constante del antecedente.	5
sigma	Sigma del antecedente.	1
sradius	Radio sigma del antecedente.	3
match_texture	Textura mínima para la busqueda intensa.	0
lr_threshold	Umbral de disparidad para el checkeo de consistencia Izquierda - Derecha.	2
speckle_sim_threshold	Umbral de similitud para la segmentación a puntos.	1
speckle_size	Tamaño máximo del punto.	200
ipol_gap_width	Ancho del espacio de interpolación de huecos pequeños (derecha-izquierda, arriba-abajo)	5000
filter_median	Booleano. Filtro opcional de la mediana.	1
lter_adaptive_mean	Booleano. Filtro opcional de la media adaptativa.	0
postprocess_only_left	Booleano. Salva tiempo post procesando sólo la imagen izquierda.	0
subsampling	Booleano. Salva tiempo calculando las disparidades por cada 2 pixeles.	0

2.3.3. Optimizaciones OpenCL

Descritpor - Primera implementación (A1) Esta implementación realiza una paralelización inicial de la función Descriptor. Al realizar la nueva implementación hemos tenido en cuenta la ventaja de realizar accesos coalescentes a memoria y nos hemos asegurado de que se así fuesen.

Es importante destacar que, en las tarjetas gráficas NVIDIA de computabilidad 1.x, las restricciones para realizar accesos coalescentes son muy estrictas (16). Ésta implementación cumple con todas las restricciones y así es capaz de funcionar de forma óptima en todas las tarjetas compatibles con CUDA.

Para el tamaño de los grupos de trabajo dejamos que el compilador de OpenCL determine las mejores dimensiones dependiendo de en qué dispositivo se va a ejecutar la aplicación.

La forma de trabajo es la siguiente:

1. Se lanza un *thread* por pixel.
2. Este *thread* realiza todos los accesos a memoria global con el fin de tener el valor de intensidad de los vecinos de su pixel.
3. Una vez recolectados estos valores, conforma el vector descriptor del pixel y lo guarda en memoria global.

En la Figura 2.3 puede verse de forma gráfica el proceso.

Para visualizar de forma más clara qué optimizaciones se han llevado a cabo en esta implementación, adjuntamos el Cuadro 2.5.

Optimizaciones de Alta Prioridad	
Paralelizar el código secuencial	Sí
Utilizar el ancho de banda efectivo de la computación como métrica	No
Minimizar las transferencias de datos entre host y device	Sí
Asegurar que los accesos a memoria son coalescentes	Sí
Minimizar el uso de la memoria global	No
Evitar la ejecución de diferentes segmentos de código dentro de un mismo warp	Sí
Utilizar la opción de compilación -cl-mad-enable	Sí

Cuadro 2.5: Optimizaciones aplicadas en la implementación A1.

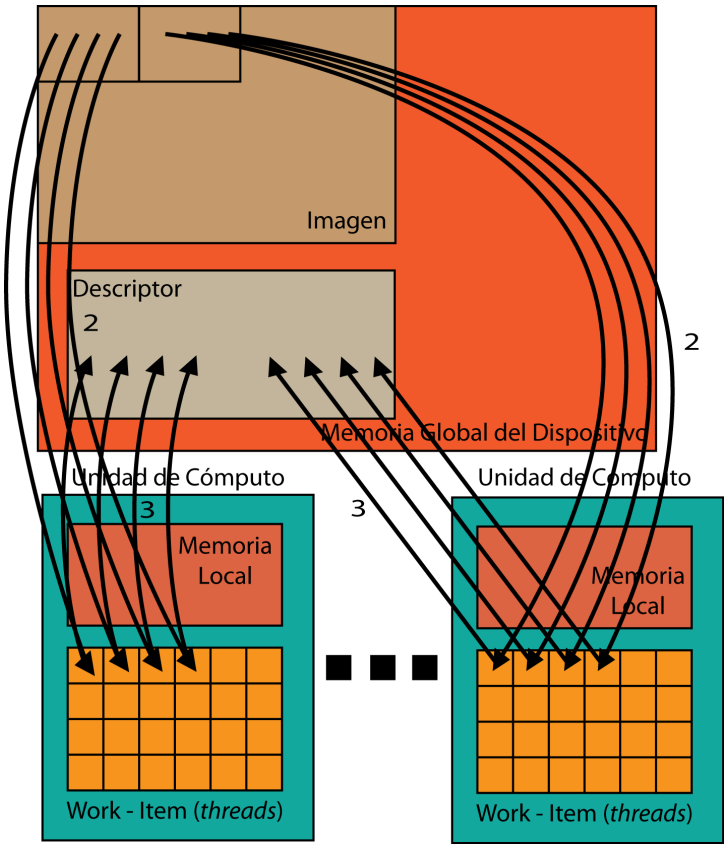


Figura 2.3: Forma de trabajo de la implementación A1.

Descritpor - Segunda implementación (A2) La principal mejora en esta implementación es usar la memoria local de cada unidad de cómputo. Para ello traemos todos los datos a usar como primer paso en la ejecución de todos los *kernels*. Los accesos a memoria local requieren varios ciclos de reloj mientras que el acceso a memoria global requiere cientos de ciclos (600-800).

Para optimizar los accesos a memoria global necesitamos el mayor solapado posible de vecindades para cada píxel. Por ello, la región óptima para traer a memoria local es un rectángulo. Y teniendo en cuenta que las planificaciones en CUDA se realizan en warps, nos aseguramos de que el tamaño de los grupos de trabajo es múltiplo de 32. El valor escogido para los grupos de trabajo es, en dos dimensiones, (32,8).
Al necesitar cada píxel acceder a sus vecinos dentro de una ventana, también tenemos que traer estos vecinos a memoria local, aunque no se vaya a generar el descriptor para ellos.

Debido a que hay *thread*s que se ejecutan antes que otros, hay que forzar la espera de la carga de memoria local. Esto lo hemos hecho usando el mecanismo de barrera provisto en OpenCL.

El llenado de la memoria local se hace siguiendo estos pasos:

- Cargar el píxel que ocupa las coordenadas asignadas para el *thread*.
- Hacer un barrido de los bordes de la imagen para traer los píxeles vecinos que ocupan esas posiciones.
- En las esquinas, realizar cargas especificas de los píxeles que no han sido cargados en el paso anterior.

En la Figura 2.4 se ilustra este proceso. Todo este trabajo esta distribuido de forma tal que se reduce la divergencia entre *thread*s de un warp y se serializa la ejecución de estos lo mínimo posible.

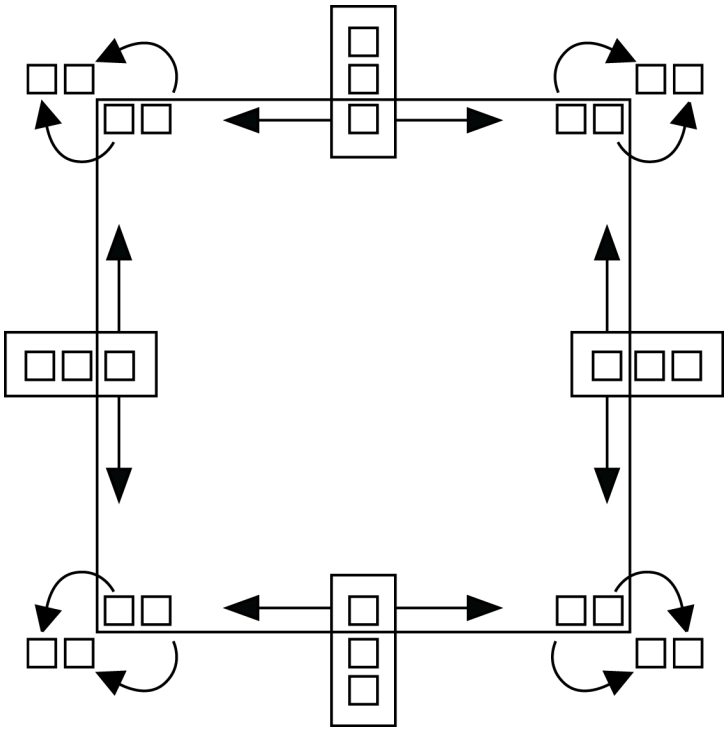


Figura 2.4: Llenado de memoria local en la implementación A2.

La forma de trabajo es la siguiente:

1. Se lanza un *thread* por pixel.
2. Todos los *threads* del mismo *work-group* colaboran para llenar la memoria local de la unidad de cómputo con la region de la imagen que van a necesitar.

3. Cada *thread* accede a los vecinos de su pixel, previamente cargados en memoria local.

4. Una vez formado el vector descriptor de su pixel, se guarda en memoria global.

En la Figura 2.5 puede verse de forma gráfica el proceso.

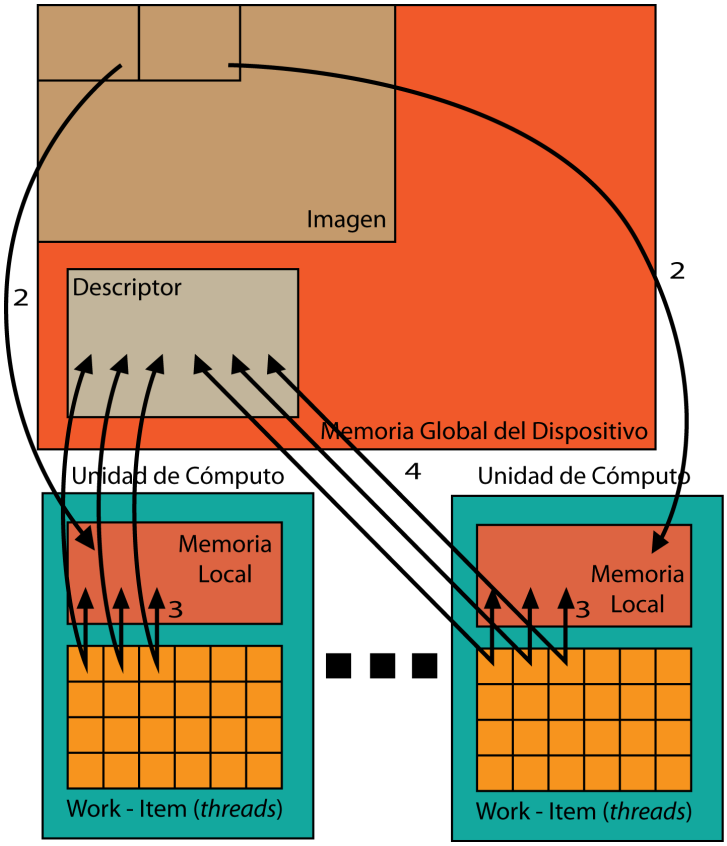


Figura 2.5: Forma de trabajo de la implementación A2.

En el Cuadro 2.6 se pueden ver las optimizaciones aplicadas.

Optimizaciones de Alta Prioridad	
Paralelizar el código secuencial	Sí
Utilizar el ancho de banda efectivo de la computación como métrica	No
Minimizar las transferencias de datos entre host y device	Sí
Asegurar que los accesos a memoria son coalescentes	Sí
Minimizar el uso de la memoria global	Sí
Evitar la ejecución de diferentes segmentos de código dentro de un mismo warp	Sí
Utilizar la opción de compilación -cl-mad-enable	Sí

Optimizaciones de Media Prioridad	
Utilizar de forma prudente la memoria "pineada"	No
Solapar transferencias de memoria *host-device* con computación en el *device*	No
Los accesos a memoria compartida deben ser diseñados para evitar peticiones serializadas debidas a conflictos de memoria	Sí
Utilizar la memoria compartida para evitar transferencias redundantes desde memoria global	Sí
El número de *threads* por *workgroup* debería ser un multiplo de 32, ya que esto proporciona una computación óptima, eficiente y facilita la coalescencia	Sí
Utilizar la libreria "math" nativa siempre que la velocidad supere la precisión	No

Cuadro 2.6: Optimizaciones aplicadas en la implementación A2.

Support Matches - Primera implementación (B1) En esta implementación hemos paralelizado el apartado en cuestión del algoritmo junto con algunas optimizaciones. Nos hemos asegurado de que los accesos a memoria son siempre coalescentes y así reducir al cantidad de peticiones separadas que se han de realizar a la memoria global. Esta implementación también realiza los accesos a memoria cloalescentes en tarjetas NVIDIA con computabilidad 1.x.

La forma de trabajo es la siguiente:

1. Se lanza un *thread* por píxel.
2. Cada *thread* adquiere los descriptores de la vecindad de su pixel.
3. De forma iterativa y aumentando el valor de disparidad, cada *thread* compara su pixel con uno en la imagen de referencia hasta encontrar una coincidencia.
4. Si se encuentra una coincidencia se guarda en memoria global el valor de disparidad, y en caso contrario, se guarda un valor que indica que no se ha encontrado.

En la Figura 2.6 puede verse de forma gráfica el proceso.

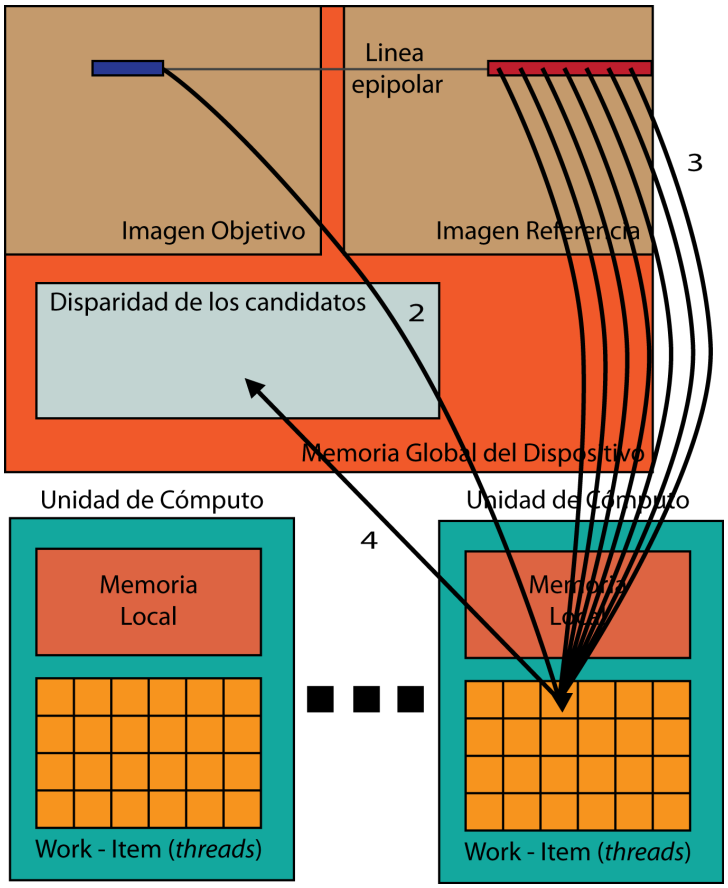


Figura 2.6: Forma de trabajo de la implementación B1.

En el Cuadro 2.7 se pueden ver las optimizaciones aplicadas.

Optimizaciones de Alta Prioridad	
Paralelizar el código secuencial	Sí
Utilizar el ancho de banda efectivo de la computación como métrica	No
Minimizar las transferencias de datos entre host y device	Sí
Asegurar que los accesos a memoria son coalescentes	Sí
Minimizar el uso de la memoria global	No
Evitar la ejecución de diferentes segmentos de código dentro de un mismo warp	No
Utilizar la opción de compilación -cl-mad-enable	Sí

Cuadro 2.7: Optimizaciones aplicadas en la implementación B1.

Es importante hacer notar que, tanto la implementación B1 como B2, tiene a su vez, incluída las mejoras realizadas en la implementación A1.

Support Matches - Segunda implementación (B2) - Parcial Esta optimización esta en estado parcial de implementación. En ella, el cálculo de los valores de disparidad es correcto en solo un sentido (al computar las disparidades para la imagen izquierda). Los tiempos de ejecución son correctos, ya que se relaliza el cálculo en los dos sentidos, aunque los resultado no son consistentes con la implementación original. Adicionalemente hay discrepancias con las disparidades calculadas para los píxeles cuyos threads ocupan las posiciones finales e iniciales de los grupos de trabajo. Véase 5.

La principal mejora en esta implementación es, también, usar la memoria local de cada unidad de cómputo para traer previamente los datos a usar. No obstante, debido a particularidades en la forma de funcionar, hubo que realizar un diseño más complejo.

Para entender el por qué del diseño es necesario profundizar un poco en cómo actúa el algoritmo. Debido a que el método de paralelización, para el punto anterior, ha sido también crear un *thread* por píxel, evaluaremos el funcionamiendo desde este punto de vista:

- Cada *thread* tiene que acceder a los descriptores de los vecinos del pixel que le corresponde, en la imagen *objetivo*.
- Teniendo esto, de forma iterativa, tiene que buscar qué pixel, en la imagen de *referencia*, guarda mayor similitud con el que le corresponde.

  Para esto, tiene que acceder a los descriptores de los vecinos de cada píxel candidato.

  Adicionalemente, al traer los datos a memoria local, se ha de tener en cuenta cuándo los datos existen y cuándo no. Por ejemplo, en píxeles cerca de los bordes de la imagen.

Ahora, supongamos que tenemos dos píxeles, ambos de la imagen *objetivo*, cuyas posiciones son cercanas. Cuando para estos píxeles se acceda a los posibles candidatos en la imagen de referencia, habrá muchos que sean candidatos comunes. Se hace evidente que este es un comportamiento candidato a optimizar.

Como caso inicial analicemos el comportamiento del *work-group* que se encargará de calcular las disparidades para puntos en el medio de la imagen izquierda. Así, evitamos considerar el problema de los bordes de la imagen.

Para que el *thread* encargado de calcular la disparidad del píxel más a la derecha pueda acceder a los candidatos de la imagen de *referencia*, estos tienen que estar previamente en la memoria local. Llamamos a este conjunto de píxeles, *Base*. Así, como paso inicial todos los *threads* del *workgroup* realizarán la tarea de traer la *Base* a memoria local de forma conjunta.

La ejecución de los threads esta organizada, por cuestiones de optimización (1.4.4), en grupos de 32. Entonces, una vez esté la *Base* en memoria local, los 32 threads tiene que acceder al píxeles que les corresponden, tanto en la imagen *objetivo* como en la de *referencia*. Estos píxeles se cargan previamente en memoria local.
Una vez ejecutados los primeros 32 *threads*, el *thread* más a la izquierda del segundo grupo de 32 tendrá los candidatos ya en memoria local. De esta forma, cada *workgroup* serializa su ejecución en grupos de 32 threads.

La decisión de secuencializar grupos de 32 threads ha sido tomada por dos razones:

- Debido a el solapado de accesos que hay a la imagen de referencia por parte de los *threads*, cada grupo, al traer lo píxeles que ocupan las mismas coordenadas (de la imagen de *referencia*), le ahorra trabajo a los threads del segundo grupo, y así sucesivamente.
- La secuencialización permite que se pueda reutilizar espacio en memoria local. Esto deja como única restricción para definir el tamaño de los *workgroups* la cantidad de memoria *privada* del dispositivo que va a realizar los cálculos.

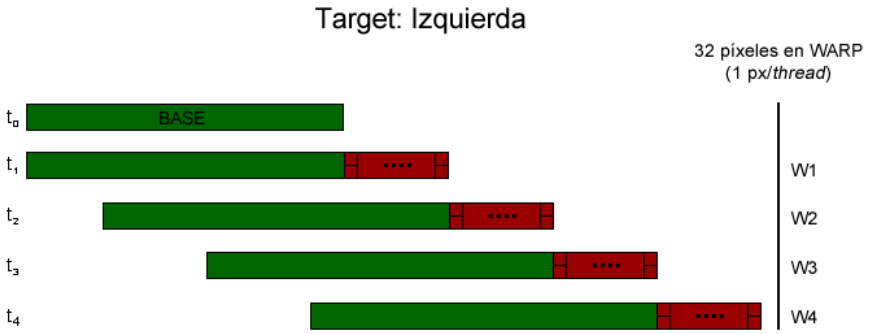
  Teniendo en cuenta que el valor de disparidad máximo es 255 se deduce que hay que cargar en memoria local la misma cantidad de píxeles. Adicionalmente, 32 píxeles van a ser cargados por los threads del primer grupo (*warp* en el caso de NVIDIA). De este primer grupo, el *thread* responsable del píxel más a la derecha, leerá el valor de sólo 223 píxeles de la *Base*. Para el segundo grupo, el *thread* responsable del píxel más a la derecha sólo leerá 191 píxeles de la *Base*, y asi sucesivamente.

Las firguras 2.7 y 2.8 intentan acalarar de manera grafica este proceso.

Una complicación que aparece y afecta al código fuertemente es que, al comparar píxeles, la vecindad que se utiliza tiene un salto determinado. Concretando más, si el pixel a comparar tiene coordenadas (x,y), se accederá a los siguientes píxeles: $(x - u\_step, y - v\_step)$, $(x + u\_step, y - v\_step)$, $(x - u\_step, y + v\_step)$, $(x + u\_step, y + v\_step)$. u_step y v_step tienen valor 2. Repasando la explicación previa teniendo en cuenta esto, se puede apreciar a qué puntos afecta y qué cambios hay que hacer.

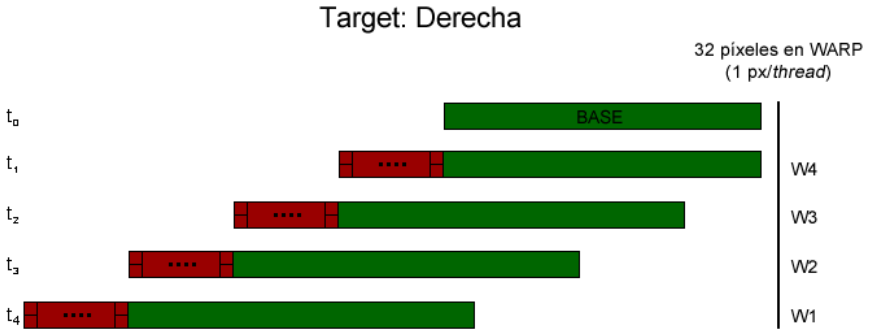
La forma de trabajo es la siguiente:

1. Se lanza un *thread* por píxel.



Como primer paso todos los threads del workgroup traen la Base. Cuando se ejecuta el primer grupo de 32 threads, primer warp, se accede a toda la base. Al ejecutarse el segundo grupo, solo se accede a una porción de esta. Así se ve que, es posible usar una estructura circular, siempre y cuando haya espacio para almacenar la base y los píxeles necesarios para un grupo.

Figura 2.7: Llenado de la memoria local para la implementación B2.



Cuando la imagen objetivo (*target*) es la derecha, lo grupos se ejecutan de derecha a izquierda.

Figura 2.8: Llenado de la memoria local para la implementación B2.

2. Se trae la Base y los píxeles a usar de la imagen de referencia a la memoria local. Todo esto de forma colaborativa y coalescente entre threads.

3. Se traen todos los píxeles necesarios de la imagen objetivo a memoria local.

4. Se realiza la iteración, calculado el valor de disparidad y se guarda en memoria global.

En la Figura 2.9 puede verse de forma gráfica el proceso.

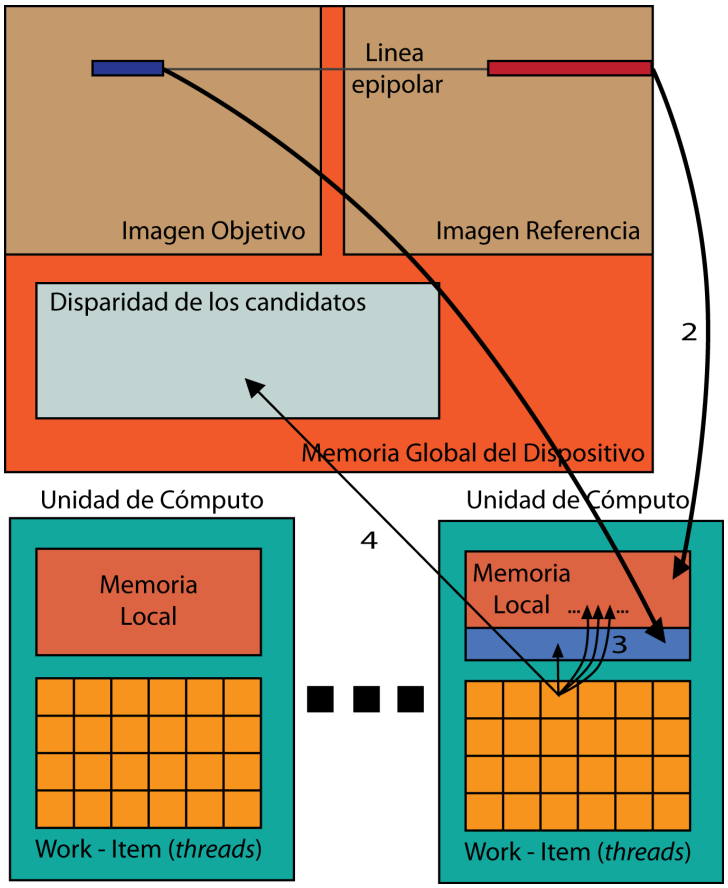


Figura 2.9: Forma de trabajo de la implementación B2.

En el Cuadro 2.8 se puede ver un resumen de las optimizaciones de esta implementación.

Optimizaciones de Alta Prioridad	
Paralelizar el código secuencial	Sí
Utilizar el ancho de banda efectivo de la computación como métrica	No
Minimizar las transferencias de datos entre host y device	Sí
Asegurar que los accesos a memoria son coalescentes	Sí
Minimizar el uso de la memoria global	Sí
Evitar la ejecución de diferentes segmentos de código dentro de un mismo warp	Sí
Utilizar la opción de compilación -cl-mad-enable	Sí

Optimizaciones de Media Prioridad	
Utilizar de forma prudente la memoria "pineada"	No
Solapar transferencias de memoria *host-device* con computación en el *device*	No
Los accesos a memoria compartida deben ser diseñados para evitar peticiones serializadas debidas a conflictos de memoria	Sí
Utilizar la memoria compartida para evitar transferencias redundantes desde memoria global	Sí
El número de *threads* por *workgroup* debería ser un multiplo de 32, ya que esto proporciona una computación óptima, eficiente y facilita la coalescencia	Sí
Utilizar la libreria "math" nativa siempre que la velocidad supere la precisión	Sí

Cuadro 2.8: Optimizaciones aplicadas en la implementación B2.

2.3.4. Código en C Plano y SSE

Descriptor El código de la función que genera el descriptor es común en implementaciones. Esto es debido a que no se puede acelerar el cómputo del descriptor usando instrucciones SIMD. En el Apéndice B.1 se adjunta el código principal para el cómputo del descriptor.

Support Matches La implementación distribuida por el autor del algoritmo está acelerada con instrucciones SSE. Para tener una perspectiva más amplia, eliminamos las optimizaciones SSE del algoritmo original. Así el tiempo de computación se incrementó considerablemente. Ambas implementaciones estan adjuntas en los Apéndices B.2 y B.3.

El cómputo de los puntos de soporte es realizado en dos pasos.
Primero se invoca a la funcion *computeSupportMatches* quien se encarga parcialmente de verificar la robustez en el valor de disparidad de un punto en el espacio. En resumen, esto lo que hace:

- Compara los valores resultados del cálculo de la disparidad de izquierda a derecha y de derecha a izquierda.

- Aplica las resitricciones definidas en el parámetro *lr_ threshold.*
- Elimina los puntos con disparidad inconsistente, véase parametro *incon_ threshold.*
- Elimina los puntos con disparidades redundantes, véase parámetro *redun_ threshold.*
- Cambia la representación de los puntos de soporte de matricial a vectorial.
- Si el parámetro *add_ corners* esta activado, agrega como puntos de soporte las esquinas, con un valor de disparidad igual al del punto de soporte más cercano.

Como función auxiliar, se invoca a *computeMatchingDisparity,* que es la función encargada de computar el valor de disparidad para un determinado píxel. Esta función ha de verificar que:

- El píxel a computar esté dentro de un area en la que tenga siempre vecinos para comparar si un píxel es o no igual a otro en la imagen de referencia.
- El píxel a computar tenga un mínimo de textura previo al cálculo de su disparidad, véase parámetro *support_ texture.*
- Iterar sobre todos los posibles candidados y seleccionar aquel con menor diferencia. Verificando también que el valor de disparidad a devolver no sea ruido en la imagen, véase parametro *support_ threshold.*

2.3.5. Código GPU (OpenCL)

El código en OpenCL para todas las implementaciones estan en el Apéndice C.

Todo código escrito para ejecutar en OpenCL esta compuesto de dos partes: código que se ejecuta en el procesador anfitrión, *código host,* y código a ser ejecutado en el dispositivo, *código de kernels.*

El código host ha de encargarse, entre otras cosas (1.4.1), de definir el rango de ejecución para los *kernels* y cómo van a ser agrupados estos, es decir, definir el tamaño de los workgroups.

Para las implementaciones A1 y A2, se define ,como rango para aplicar los *kernels*, toda la imagen.

El objeto con el que se parte, en el cálculo de los puntos de soporte, es una matriz cuyas dimensiones son las de la imagen inicial dividas por *candidate_ stepsize*. Por tanto, para las implementaciones B1 y B2, el rango de ejecución es el mismo que las dimensiones de esta matriz.

Descriptor - A1 El código de esta implementación es bastante simple de seguir. La división de los grupos de trabajo esta hecha por OpenCL ayudándose del compilador distribuido con los drivers de la tarjeta gráfica en la que se vaya a ejecutar la aplicación.

El código para esta implementación se encuentra en el Apéndice C.1.

Descriptor - A2 Para esta implementación hemos definido un tamaño de grupo de trabajo de 32 x 8. Es importante tener en cuenta que la instrucción *barrier* bloqueará todos los threads que hayan llegado a ese punto hasta que el resto también llegue.

El código para esta implementación se encuentra en el Apéndice C.2.

Support Matches - B1 En esta implementación hemos fijado un tamaño de grupo de trabajo de 64 x 1. Esto es así ya que, por defecto, el valor de *candidate_ stepsize* es 5, y el solapamiento vertical que se puede conseguir es mucho menor que el solapamiento horizontal. Esta es la razón de maximizar el tamaño de las columnas usadas y minimizar el tamaño de las filas.

El código para esta implementación se encuentra en el Apéndice C.3.

Support Matches - B2 Debido al nuevo mecanismo para utilizar la memoria local, el tamaño de los grupos de trabajo se calculan de una forma más compleja. El valor de éste, en bytes, cumple la siguiente ecuación:

$$wgs = (wgw, 1)$$

$$wgw = SNAP\left(\text{mín}\left(\left\lfloor \frac{lms - bw \times 2 \times 16}{css \times 2 \times 16 + 4 \times 16} \right\rfloor, 32\right), 32\right)$$

$$SNAP\,(x,y) = \begin{cases} x & x \bmod y = 0 \\ x - (x \bmod y) + y & otherwise \end{cases}$$

Siendo,

wgs el tamaño de los grupos de trabajo.

wgw el ancho de los grupos de trabajo.

lms el tamaño de memoria local del dispositivo (como mínimo 16KB en las tarjetas NVIDIA).

bw el ancho de la Base.

css el valor del parámetro *candidate_ stepsize*.

Las multiplicaciones por 16 son debidas al tamaño de los descriptores. Las operaciones con 32 como operando son para asegurar que el ancho de los grupos de trabajo es multiplo de 32. Debido a que solo se traen a memoria local las vecindades de los píxeles a comparar, y estos solo estan en dos líneas de la image de referencia, ciertos terminos hay que multiplicarlos por 2. Finalmente, también se traen a memoria local 4 descriptores por píxel a comparar, de la imagen objetivo.

El código para esta implementación se encuentra en el Apéndice C.4.

Capítulo 3

Experimentación

Para evaluar la respuesta de las diferentes optimizaciones han de aplicarse una serie de pruebas que aporten datos significativos de los cambios realizados. Adicionalmente, es necesario realizar todas las pruebas en diferentes plataformas, para tener así una referencia de cómo escalan las diferentes implementaciones.

Para la experimentación de las implementaciones A1 y A2 hemos evaluado la respuesta del algoritmo ante diferentes tamaños de imagen.

Para la experimentación de las implementaciones B1 y B2 hemos preparado una batería de diferentes configuraciones de parámetros con el fin de evaluar como responde el algoritmo a la variación de ellos. Estos parámetros los almacenamos en un archivo .cvs para así poder tratarlos mediante cualquier aplicación de hojas de cálculo.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		disp_min	disp_max	support_threshold	support_texture	candidate_stepsize	icon_window_size	incon_threshold	incon_min_support	add_corners	grid_size	beta	gamma	sigma
2	1	0	255	0.85	10	5	5	5	5	0	20	0.02	3	1
3	2	1	254	0.85	10	5	5	5	5	0	20	0.02	3	1
4	3	2	253	0.85	10	5	5	5	5	0	20	0.02	3	1
5	4	3	252	0.85	10	5	5	5	5	0	20	0.02	3	1
6	5	4	251	0.85	10	5	5	5	5	0	20	0.02	3	1
7	6	5	250	0.85	10	5	5	5	5	0	20	0.02	3	1
8	7	6	249	0.85	10	5	5	5	5	0	20	0.02	3	1
9	8	7	248	0.85	10	5	5	5	5	0	20	0.02	3	1
10	9	8	247	0.85	10	5	5	5	5	0	20	0.02	3	1
11	10	9	246	0.85	10	5	5	5	5	0	20	0.02	3	1
12	11	10	245	0.85	10	5	5	5	5	0	20	0.02	3	1
13	12	11	244	0.85	10	5	5	5	5	0	20	0.02	3	1
14	13	12	243	0.85	10	5	5	5	5	0	20	0.02	3	1
15	14	13	242	0.85	10	5	5	5	5	0	20	0.02	3	1
16	15	14	241	0.85	10	5	5	5	5	0	20	0.02	3	1
17	16	15	240	0.85	10	5	5	5	5	0	20	0.02	3	1
18	17	16	239	0.85	10	5	5	5	5	0	20	0.02	3	1
19	18	17	238	0.85	10	5	5	5	5	0	20	0.02	3	1
20	19	18	237	0.85	10	5	5	5	5	0	20	0.02	3	1

Figura 3.1: Base de datos de parámetros con la que vamos a trabajar.

3.1. Experimentación de implementaciones A1 y A2

Debido al papel que cumple la función *Descriptor* en el algoritmo, una configuración diferente del algoritmo no habría aportado información significativa. Para evaluar la respuesta de ambas implementaciones hemos medido los tiempos de ejecución en imágenes con diferentes tamaños.

Las imágenes a probar tienen las características expuestas en el Cuadro 3.1.

3.2. Experimentación de implementaciones B1 y B2

Hemos seleccionado los parámetros que influyen de forma significativa en el tiempo de ejecución del punto cambiado. Y ejecutamos ambos algoritmos con dos imágenes. Para poder tener así una visión más amplia de su funcionamiento.

Nombre	Ancho (píxeles)	Alto (píxeles)	Píxeles en total
Cones	900	750	675000
Baby	1312	840	1102080
Bowl	1250	1110	1387500
Rocks	1276	1110	1416360
Aloe	1282	1110	1423020
Lamp	1300	1110	1443000
Cloth	1300	1110	1443000
Raindeer	1342	1110	1489620

Cuadro 3.1: Imágenes de prueba para implemetaciones A1 y A2.

A continuación vamos a detallar los parámetros y describiremos los motivos por los que los hemos seleccionado.

disp_min y disp_max Como ya hemos visto, estos dos parámetros marcan el rango de disparidad en el que se va a intentar buscar un punto de la imágen objetivo en la imágen de referencia. Por lo tanto alterar estos parámetros va a afectar de forma significativa el tiempo de ejecución de la aplicación.

Cuanto más pequeño sea el rango, el punto de la imágen objetivo se buscará en un espacio más pequeño de la imagen de referencia. Esto influirá significativamente en el resultado final, ya que, si restringimos el rango, puede ser que puntos que se encuentran cercanos a la cámara no sean encontrados al estar en un rango de disparidad mayor al impuesto.

Hay que tener en cuenta que el algoritmo impone una disparidad mínima de 11 (distancia 0..10).

support_texture Modificar este parámetro influirá en que el algoritmo decida computar la disparidad de un punto dado o no, según si tiene textura suficiente. Aumentando este parámetro, exigiremos que un punto necesite estar en una zona con una textura más definida que antes. Lo que producirá que puntos que antes se computaban se dejen de computar.

La textura de un píxel se calcula en función del valor de intensidad de sus vecinos (En la figura 2.1 se muestran los vecinos junto con el píxel).

Este parámetro interactua en una de las funciones más pesadas del algoritmo, por lo que resulta interesante seleccionarlo para la experimentación, veremos pues como influye este parámetro en el tiempo de ejecución al incrementarlo (exigir textura más definida) y reducirlo (exigir textura menos definida).

candidate_stepsize Modificar este parámetro va a afectar de forma directa a la cantidad de *support-points* ya que, cuanto más grande sea el paso, el número de candidatos totales será menor, y viceversa.

Esto influye directamente en la cantidad total de *support-points,* que se calculan en funcion de los candidatos, lo que hace que este parámetro sea interesante para el paso de experimentación.

lr_threshold Por último, este parámetro marca el umbral con el que se decide si un punto es robusto o no. En base a su robustez, se añadirá a la lista de *support-points*. Por tanto, disminuir este parámetro, reducirá la cantidad de *support-points* tomados y viceversa. Veremos cómo incidirá esto en los tiempos de ejecución de las funciones.

El resto de parámetros no se han tomado en consideración a la hora de experimentar, ya que interactúan con el tiempo de ejecución en un grado menor que los escogidos.

3.2.1. Bateria de test

Las baterías de test sobre las que hemos ejecutado el algoritmo se pueden observar en el siguiente cuadro.

Tests	**disp_max**	**disp_min**	**support_texture**	**candidate_stepsize**	**lr_threshold**
1	[133..255]	[0..122]	10	5	2
2	255	[0..244]	10	5	2
3	[10..255]	0	10	5	2
4	255	0	[1..100]	5	2
5	255	0	10	5	[1..10]
6	255	0	10	[1..50]	2

3.3. Resultados y análisis crítico

A continuación mostramos los tiempos obtenidos para las diferentes implementaciones con dos gráficas. Las gráficas parciales son aquellas dónde solo se ha medido el tiempo de ejecución

del punto modificado en el algoritmo. Las gráficas totales son aquellas que miden el tiempo de ejecución del algoritmo completo. Así se puede ver el grado de impacto de la optimización en el algoritmo final.

Los valores de tiempo para las implementaciones en SSE y C plano se han obtenido en una una CPU Intel Core i5 M450 a 2,40 GHz.

Los tiempos para las implementaciones en OpenCL han sido obtenidos tras la ejecución en dos tarjetas gráficas diferentes:

- NVIDIA GT 320M
- NVIDIA GTX 275

Para más información, véase Apéndice A.

3.3.1. Resultados e interpretación de implementación A1

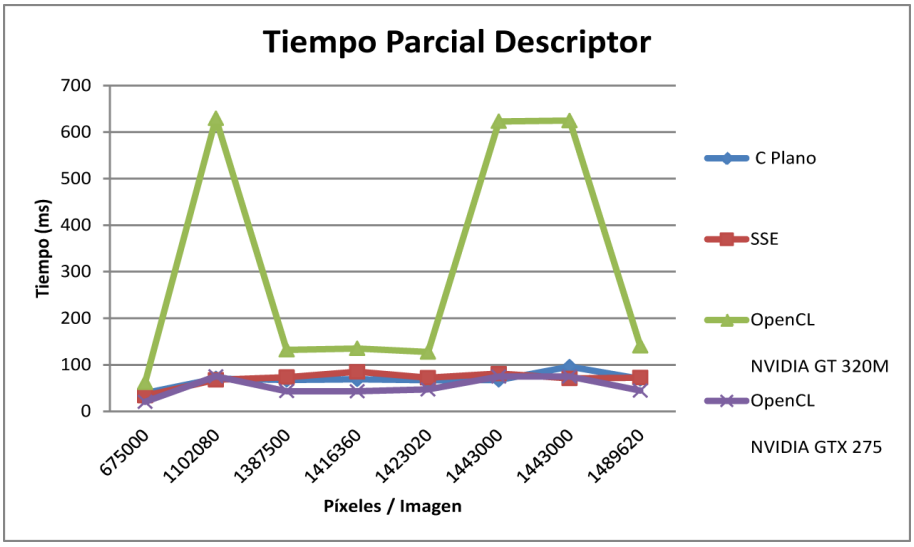


Figura 3.2: Impacto en tiempo de ejecución. Parcial.

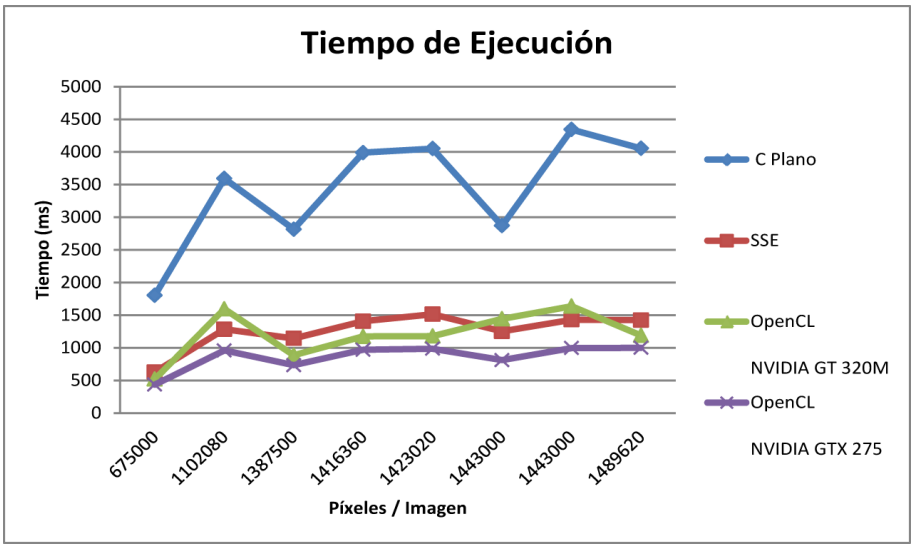


Figura 3.3: Impacto en tiempo de ejecución. Total.

Como se puede apreciar en la Figura 3.2, se producen dos grandes picos. Atribuímos esto a que la división de grupos de trabajo hecha por OpenCL no es adecuada a la tarjeta gráfica en la que se ejecuta. Como se verá en la implementación A2, estos picos han sido reducidos al fijar el tamaño de los grupos de trabajo.

Como hecho adicional, se produce el resultado esperado al ejecutar el mismo código escrito en OpenCL sobre dos tarjetas gráficas con potencia de cálculo diferente.

3.3.2. Resultados e interpretación de implementación A2

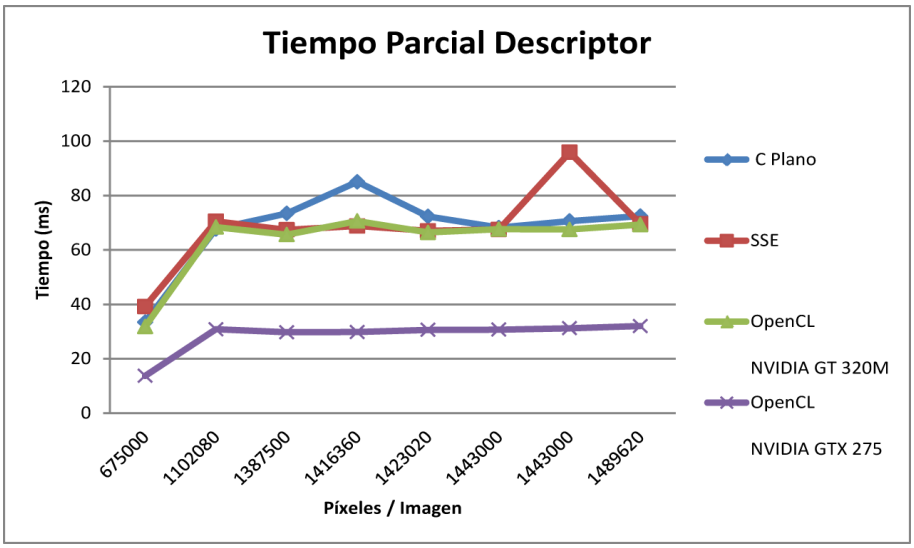


Figura 3.4: Impacto en tiempo de ejecución. Parcial.

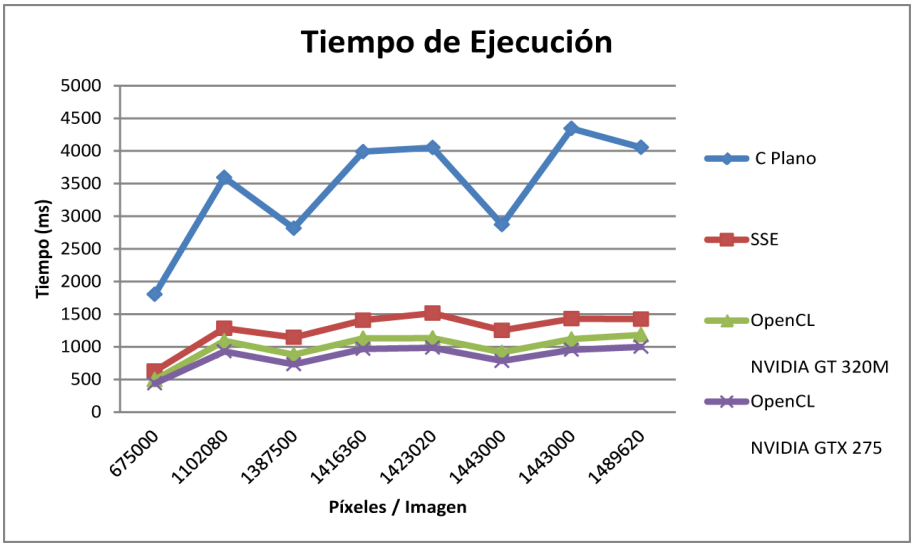


Figura 3.5: Impacto en tiempo de ejecución. Total.

Aunque, en el tiempo de ejecución total, la diferencia entre las dos ejecuciones de OpenCL y la SSE sea pequeña, ésta se puede apreciar mejor en la Figura 3.4. Aquí se ve que la ejecución del código OpenCL en una tarjeta gráfica poca capacidad se asemeja a las implementaciones base. También se hace evidente el incremento que hay, en rendimiento, al ser ejecutado el mismo algoritmo en una tarjeta gráfica más potente y que el tiempo de ejecución de éste, tarda menos de la mitad.

3.3.3. Resultados e interpretación de implementación B1

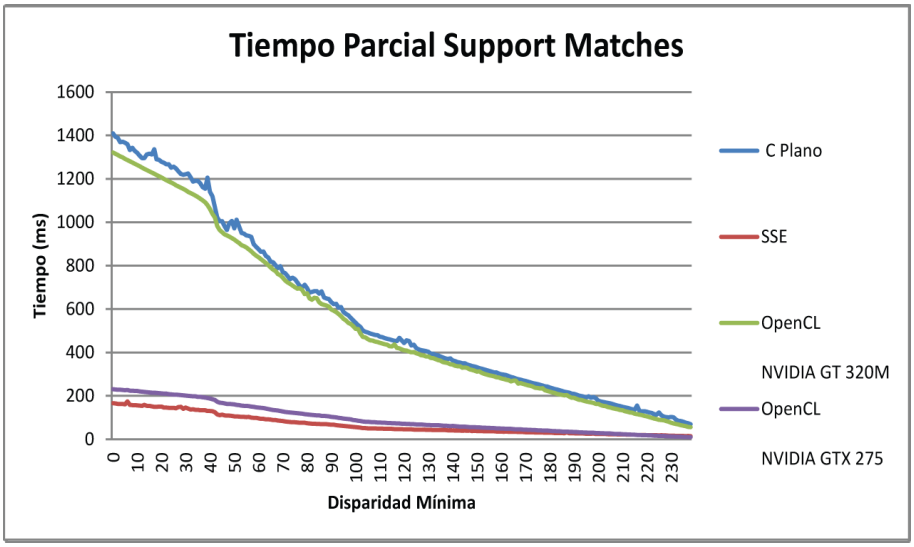


Figura 3.6: Imagen Cones. Variación de Disparidad Mínima. Impacto parcial.

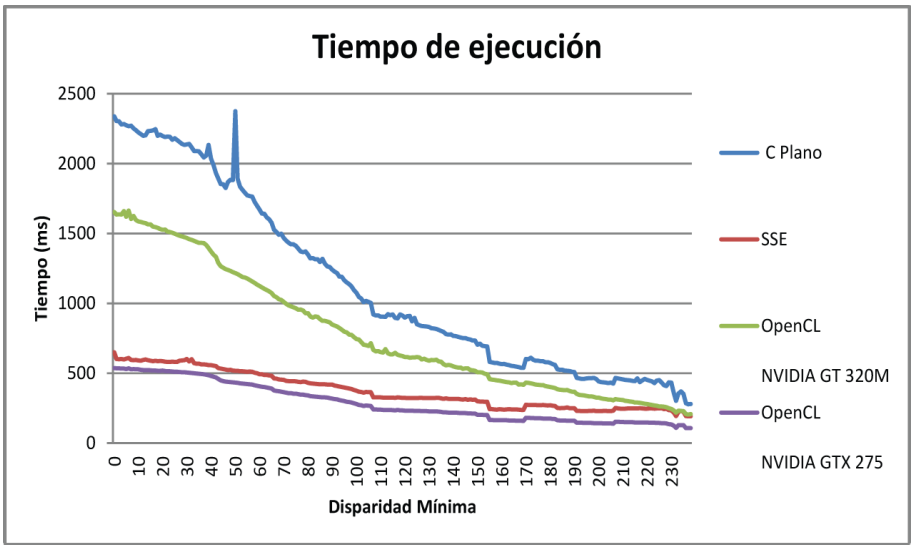


Figura 3.7: Imagen Cones. Variación de Disparidad Mínima. Impacto Total.

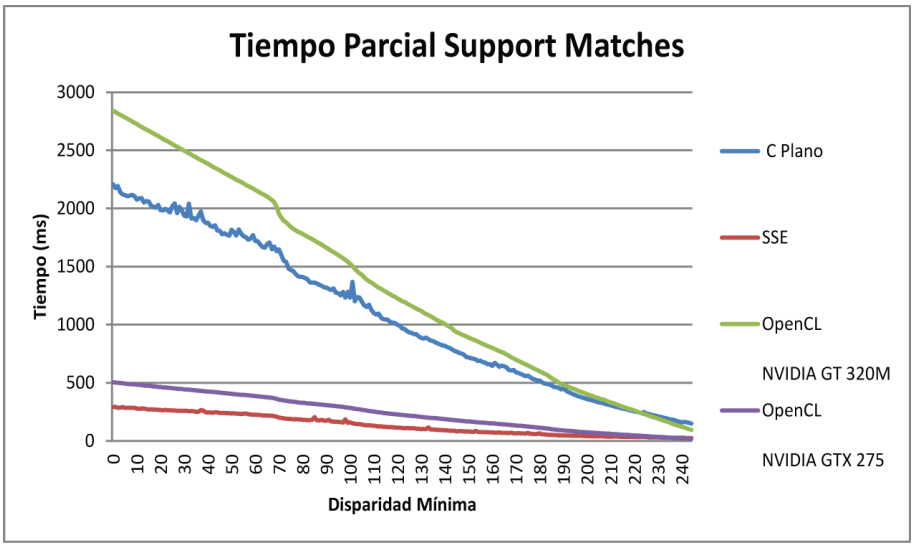


Figura 3.8: Imagen Raindeer. Variación de Disparidad Mínima. Impacto parcial.

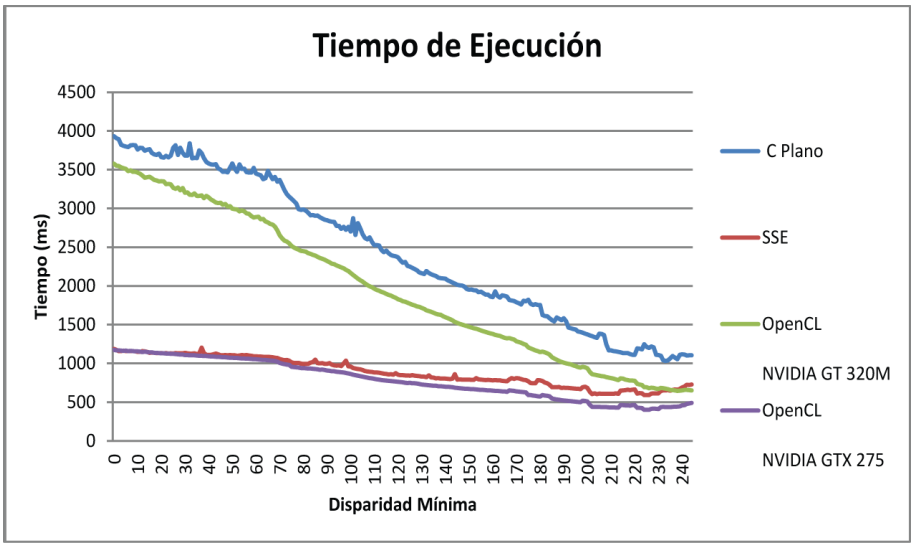


Figura 3.9: Imagen Raindeer. Variación de Disparidad Mínima. Impacto Total.

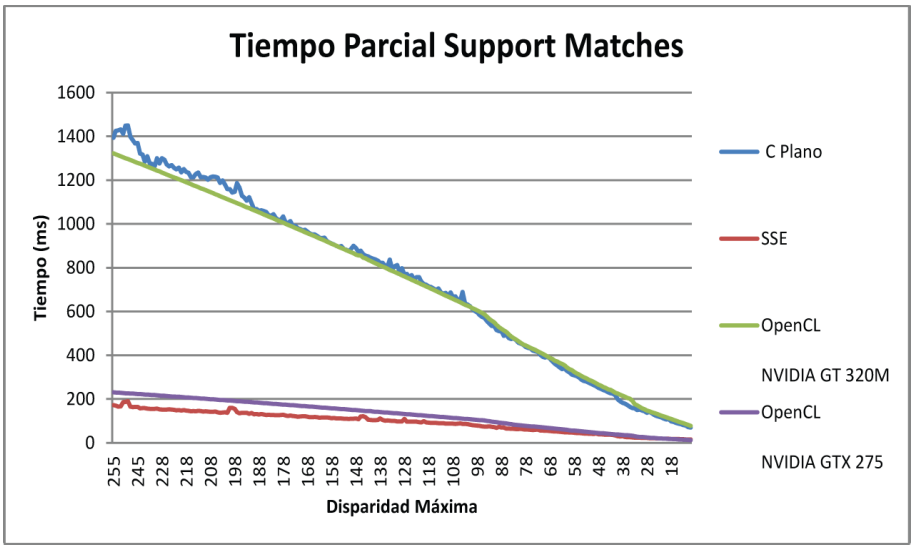


Figura 3.10: Imagen Cones. Variación de Disparidad Máxima. Impacto parcial.

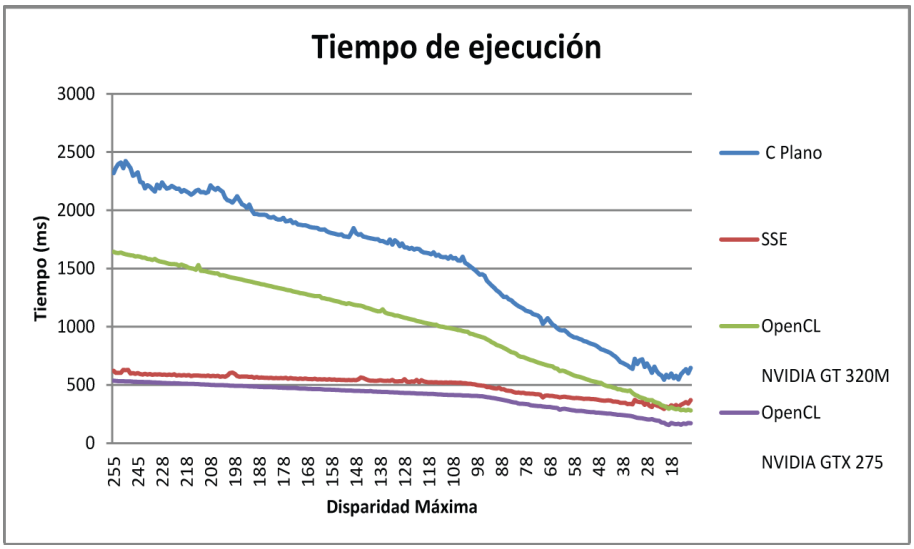


Figura 3.11: Imagen Cones. Variación de Disparidad Máxima. Impacto Total.

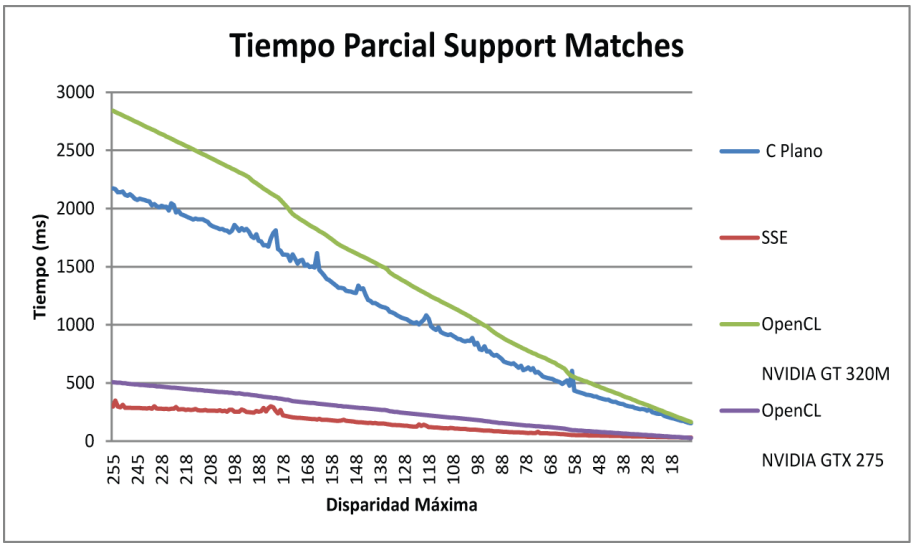


Figura 3.12: Imagen Raindeer. Variación de Disparidad Máxima. Impacto parcial.

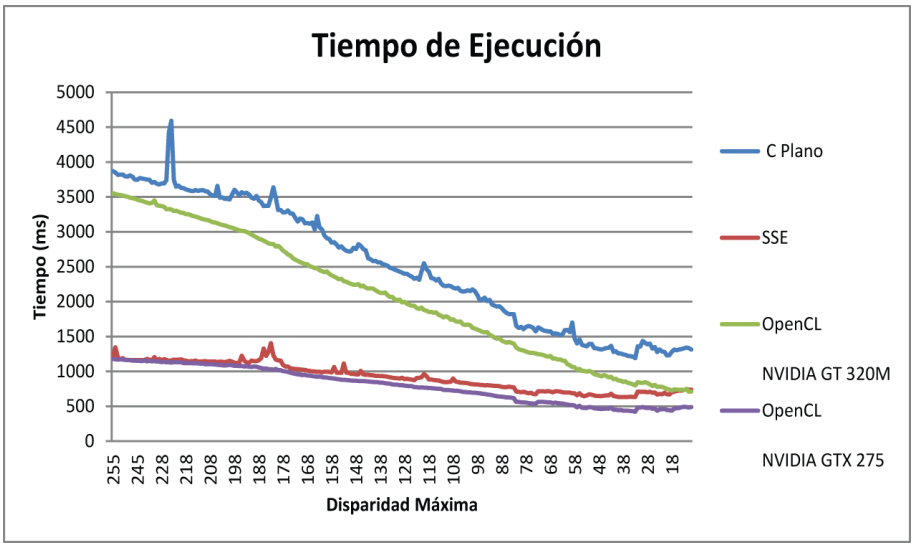


Figura 3.13: Imagen Raindeer. Variación de Disparidad Máxima. Impacto Total.

En las anteriores imágenes se puede ver que la mejora obtenida al optimizar la implementación del algoritmo en C Plano mediante instrucciones SSE es comparable a la ejecución del mismo código escrito en OpenCL en dos dispositivos diferentes.

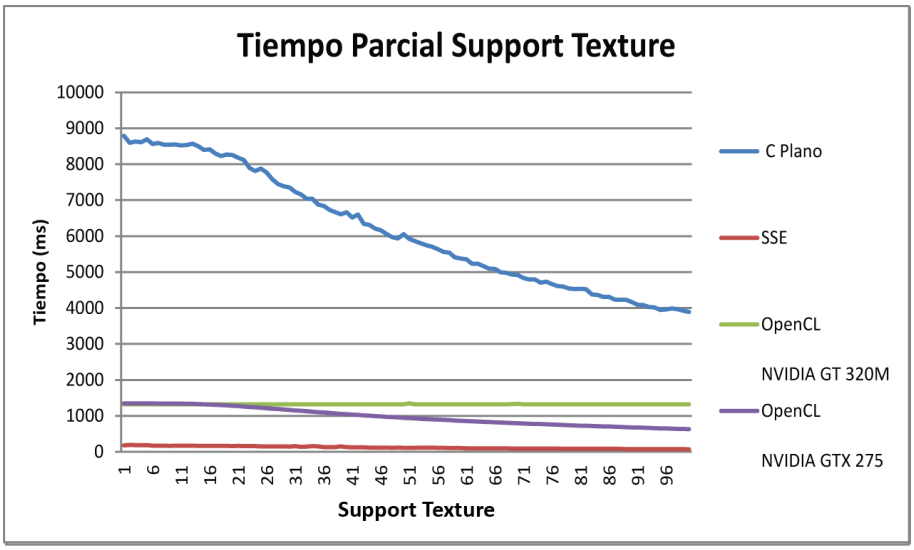


Figura 3.14: Imagen Cones. Variación de Support Texture. Impacto parcial.

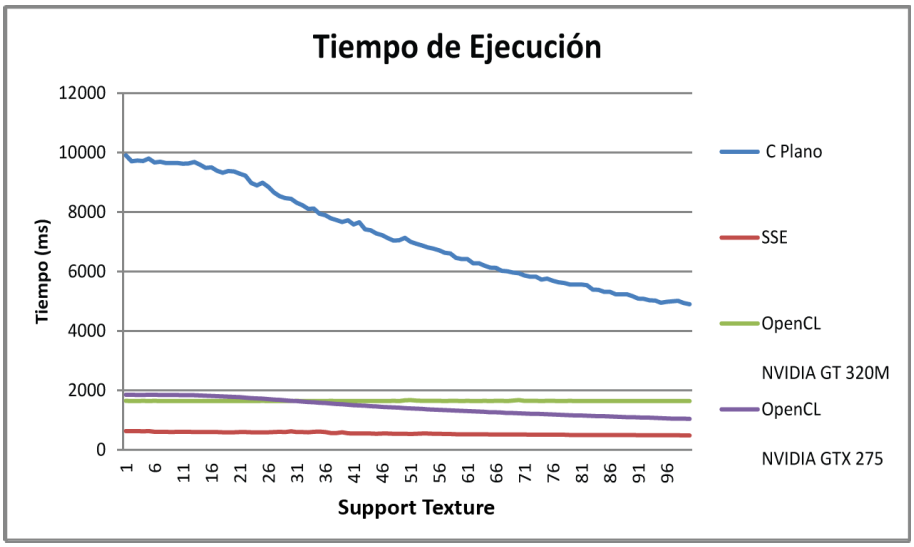


Figura 3.15: Imagen Cones. Variación de Support Texture. Impacto Total.

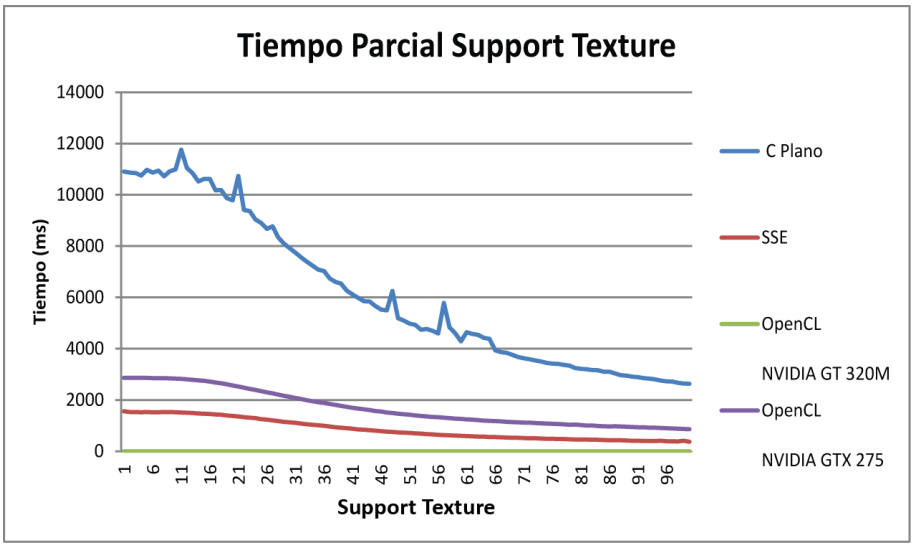


Figura 3.16: Imagen Raindeer. Variación de Support Texture. Impacto parcial.

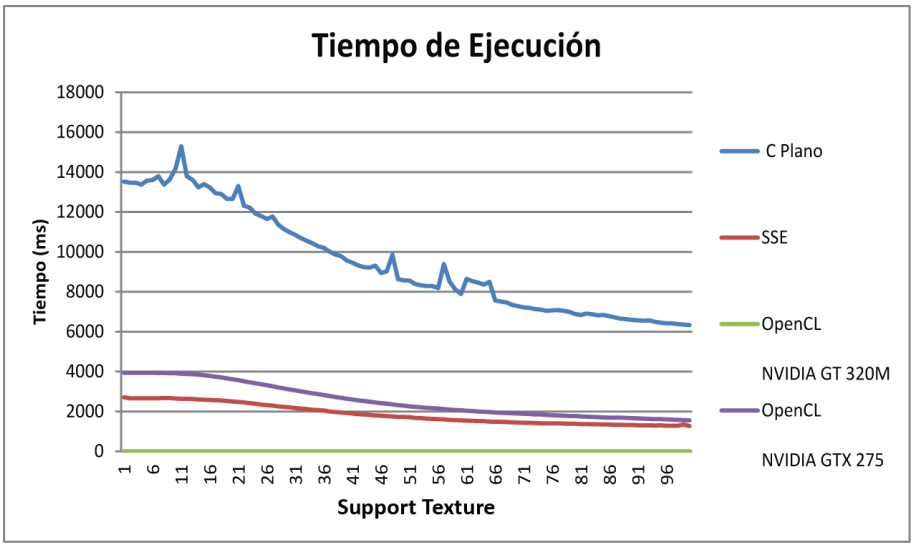


Figura 3.17: Imagen Raindeer. Variación de Support Texture. Impacto Total.

Ante la variación de el valor mínimo de textura admitido para el cómputo de un punto de soporte, resulta evidente que ambas ejecuciones OpenCL se comportan peor que la optimización original con instrucciones SSE.

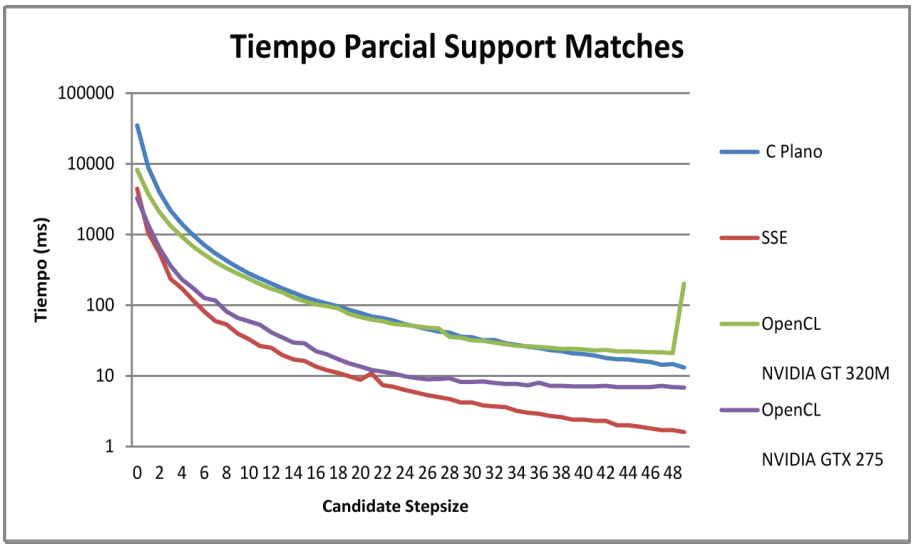


Figura 3.18: Imagen Cones. Variación de Candidate Stepsize. Impacto parcial.

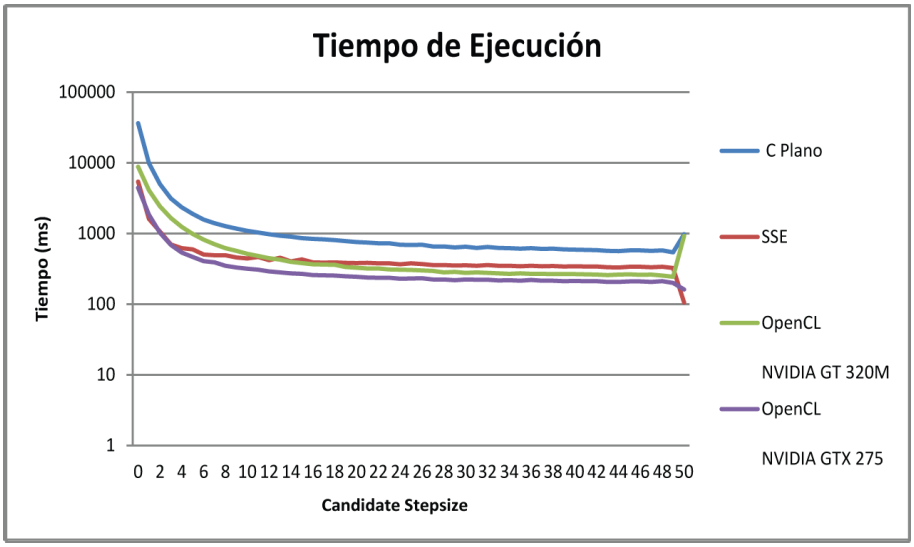


Figura 3.19: Imagen Cones. Variación de Candidate Stepsize. Impacto Total.

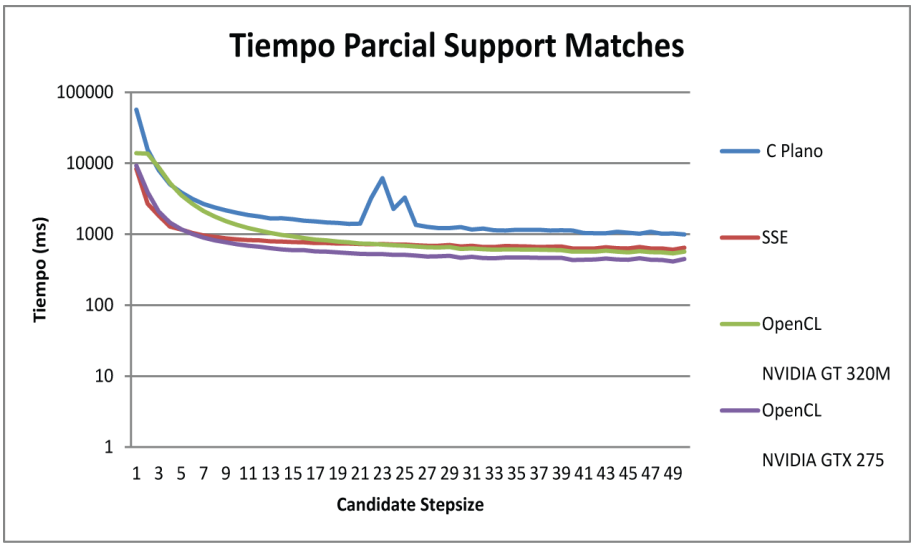


Figura 3.20: Imagen Raindeer. Variación de Candidate Stepsize. Impacto parcial.

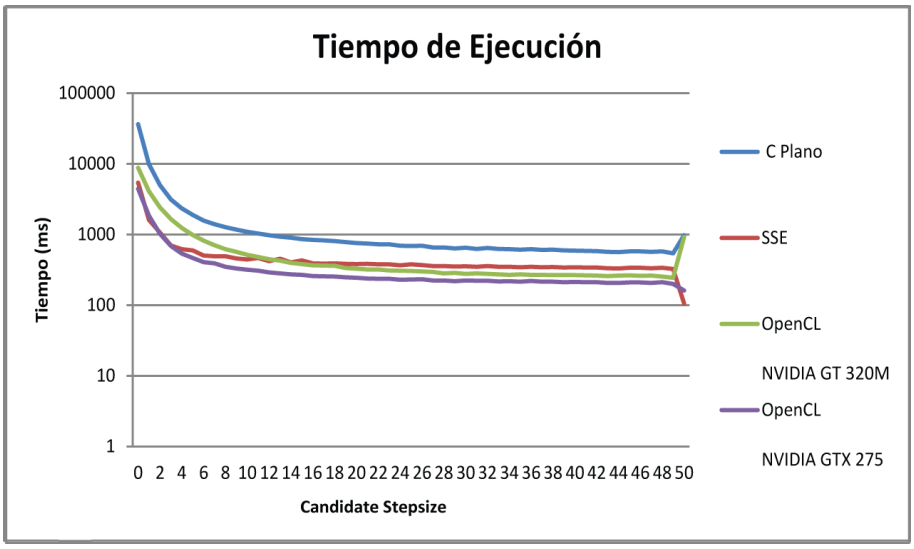


Figura 3.21: Imagen Raindeer. Variación de Candidate Stepsize. Impacto Total.

Es importante notar que todos los tiempos de ejecución para las variaciones del parámetro *candidate_stepsize* estan en escala logarítmica. Esto es debido al crecimiento exponencial que presenta el algoritmo cuando el valor de este parámetro es cercano a 0.

Especialmente en la Figura 3.18, resulta visible la similitud de los tiempos de ejecución en valores menores que 20. El código OpenCL ejecutado en una tarjeta con poca capacidad produce resultados similares a la implementación en C Plano, y OpenCL en una tarjeta gráfica potente es similar a la implementación optimizada con SSE.

El fenómeno que ocurre con valores cercanos a 50 requiere posterior profundización. Hay un evidente cambio en el comportamiento, no sólo en las implementaciones de OpenCL, sino también en la distribuída por el autor.

3.3.4. Resultados e interpretación de implementación B2

Es importante destacar que la implementación en estado parcial. Hemos decidido realizar las gráficas de los tiempos ya que el cálculo de las disparidades es correcto, aunque solo en un sentido. Esto sirve como confirmación de que las transferencias de datos entre memoria global y local tienen efecto, y los resultados de tiempo son válidos.

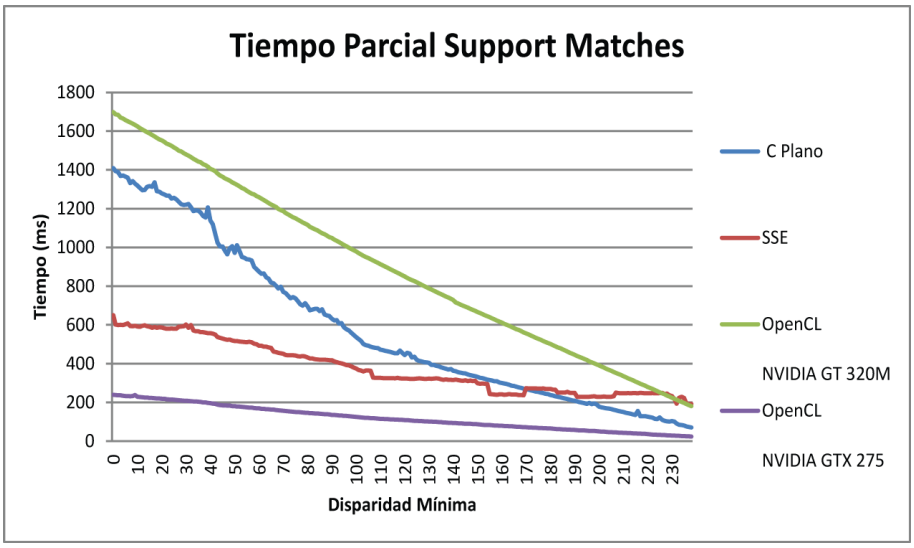


Figura 3.22: Imagen Cones. Variación de Disparidad Mínima. Impacto parcial.

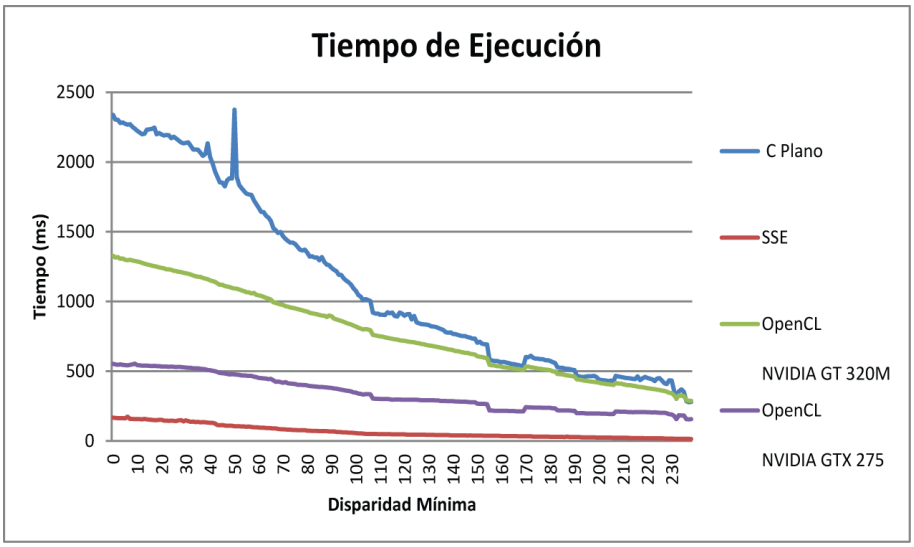


Figura 3.23: Imagen Cones. Variación de Disparidad Mínima. Impacto Total.

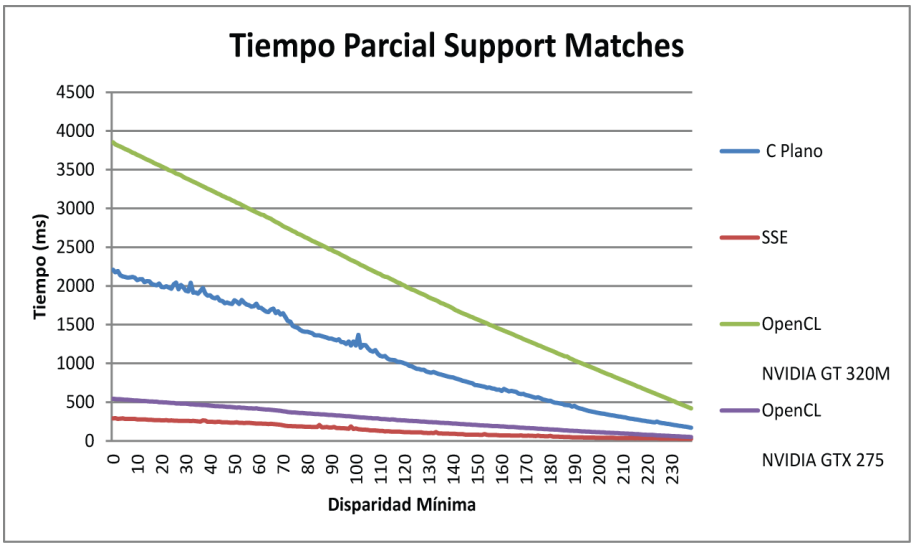


Figura 3.24: Imagen Raindeer. Variación de Disparidad Mínima. Impacto parcial.

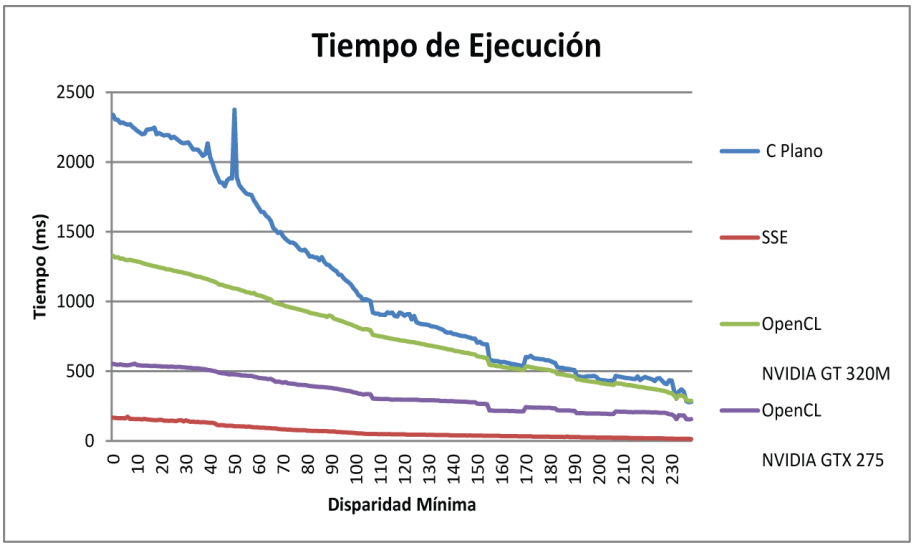


Figura 3.25: Imagen Raindeer. Variación de Disparidad Mínima. Impacto Total.

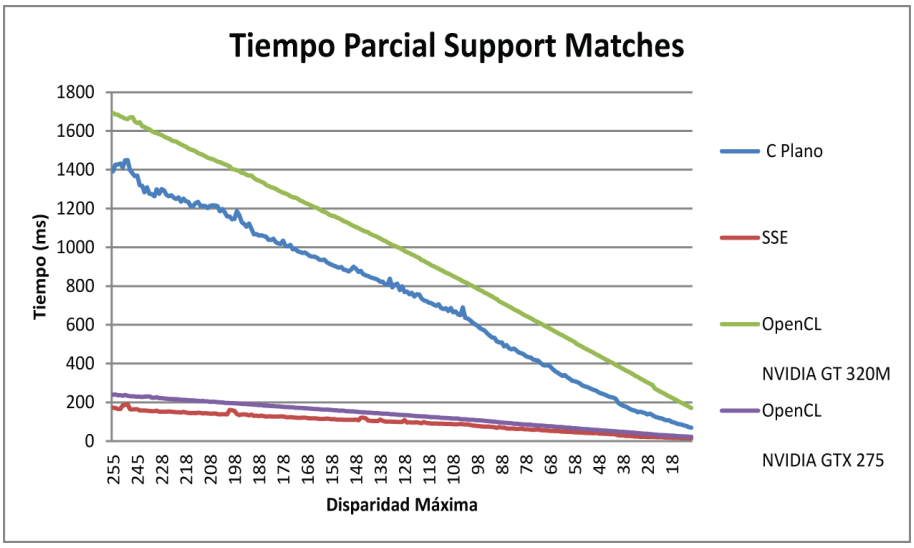


Figura 3.26: Imagen Cones. Variación de Disparidad Máxima. Impacto parcial.

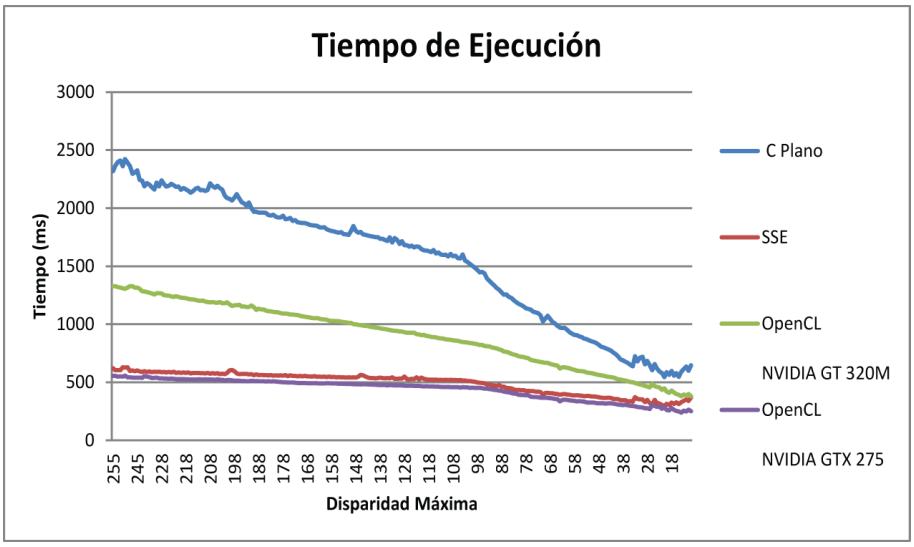


Figura 3.27: Imagen Cones. Variación de Disparidad Máxima. Impacto Total.

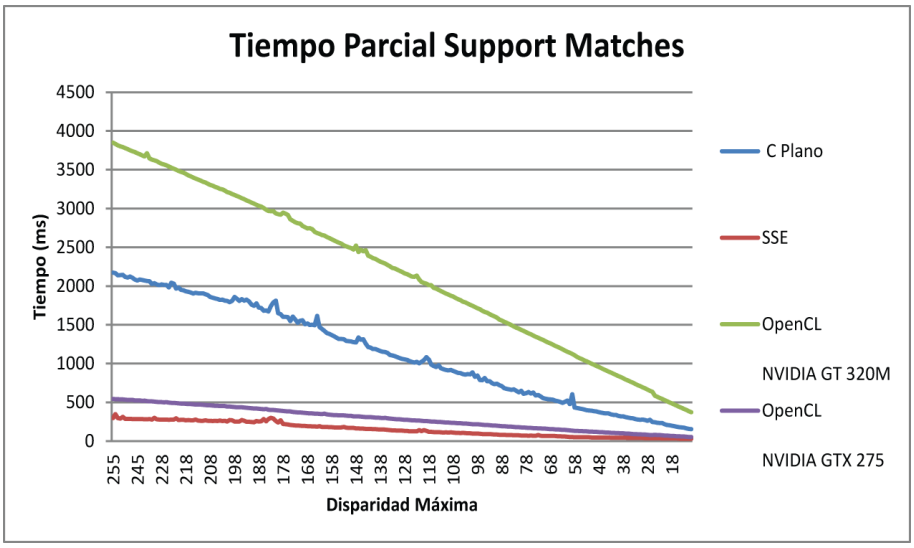


Figura 3.28: Imagen Raindeer. Variación de Disparidad Máxima. Impacto parcial.

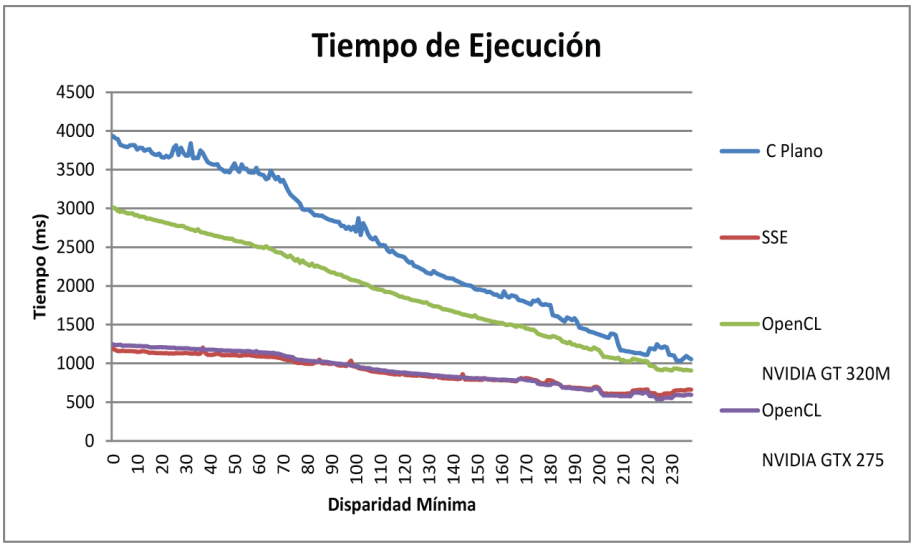


Figura 3.29: Imagen Raindeer. Variación de Disparidad Máxima. Impacto Total.

Tanto en las variaciones realizadas en el Máximo de Disparidad, como en el Mínimo de Disparidad, los resultados de las ejecuciones de las implementaciones en OpenCL muestran, nuevamente, una variación gran variación dependiendo de en qué dispositivo se utilicen. Aunque las mejoras en comparación con las implementaciones en C Plano y SSE son mínimas.

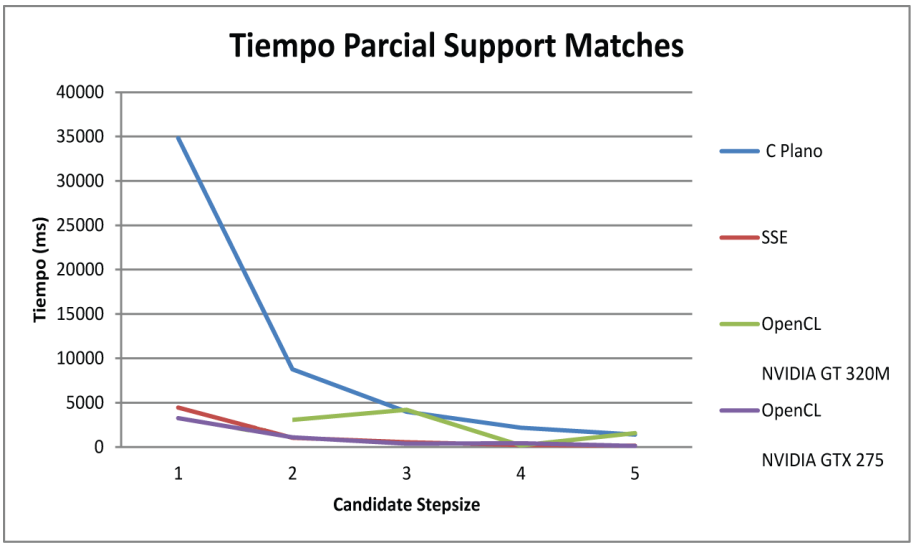


Figura 3.30: Imagen Cones. Variación de Candidate Stepsize. Impacto parcial.

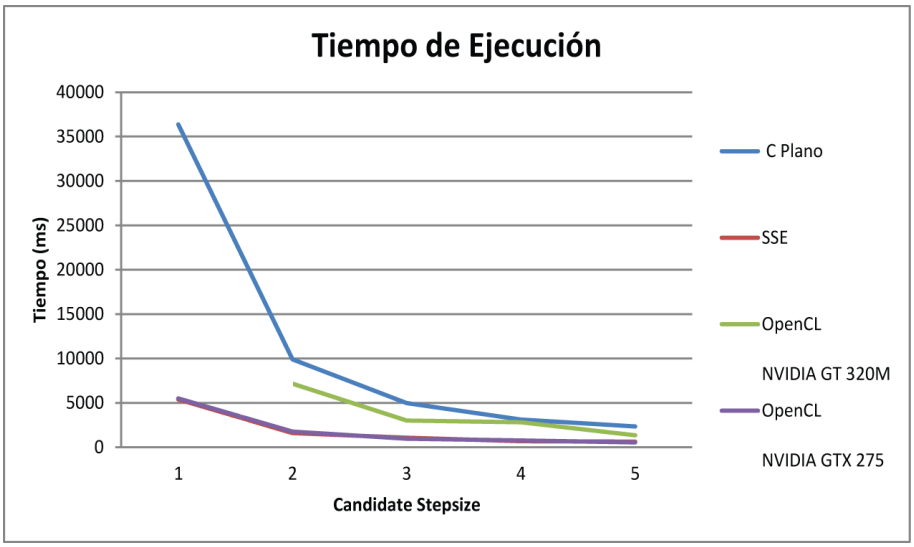


Figura 3.31: Imagen Cones. Variación de Candidate Stepsize. Impacto Total.

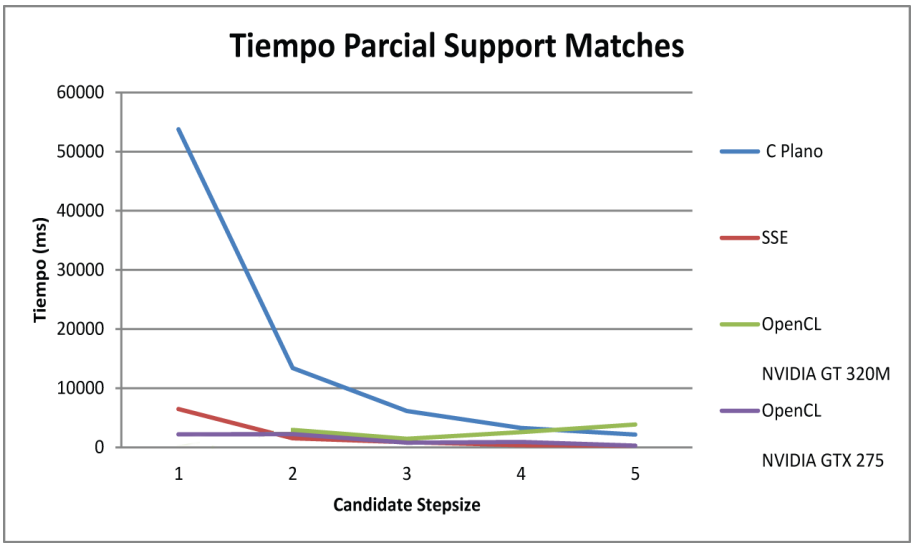


Figura 3.32: Imagen Raindeer. Variación de Candidate Stepsize. Impacto parcial.

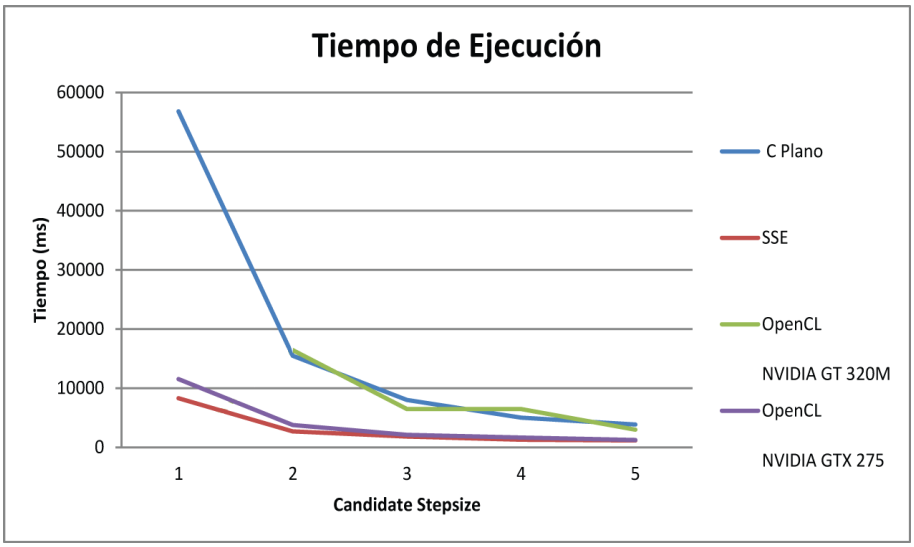


Figura 3.33: Imagen Raindeer. Variación de Candidate Stepsize. Impacto Total.

El hecho de que las gráficas de variación de *candidate_stepsize* tengan tan pocos valores, se debe a que la actual implementación no respondía de forma adecuada a los valores de entrada y por tanto no se pudo obtener los resultados de su tiempo de ejecución.

Aun siendo pocos valores, no se puede apreciar una gran diferencia de la implementación en OpenCL con respeco a las implementaciones en C Plano y SSE.

Capítulo 4

Conclusiones

Todas las implementaciones OpenCL producen el resultado esperado al ser ejecutadas en tarjetas gráficas con diferente potencia. Lo que justifica el esfuerzo de paralelización del algoritmo.

Adicionalmente, las ventajas de utilizar de forma eficiente la memoria local son claras al estudiar las gráficas del Apartado 3.3.

Aunque la mejora A2 realizada al primer punto seleccionado para optimizar, Descriptor, presenta una mejora importante, la repercusión final en el tiempo de ejecución del algoritmo no es de gran impacto. Aun así, ejecución de la implementación A2 se lleva a cabo, siempre, en tiempos menores a 1 segundo, cuando la implementación optimizada con SSE llega a tardar 1.5 segundos.

En general, no se puede afirmar que las implementaciones en OpenCL B1 y B2 sean mejores que la implementación original optimizada para SSE. Sólo que guardan un alto grado de similitud la implementación en SSE y la implementación en OpenCL ejecutada en un tarjeta gráfica con una gran capacidad de cálculo.

Es necesario lograr una implementación más robusta de la optimización B2. Con los datos adquiridos hasta el momento, no se puede apreciar ninguna mejora significativa, y tampoco es posible afirmar que esta implementación no se comporta mejor bajo ciertas circunstancias que la implementación en SSE.

Evaluando el diseño de B2, hay razones para justificar un comportamiento mejor cuando los valores de *candidate_stepsize* son pequeños. No obstante, usar valores pequeños para este parámetro va en contra del objetivo que tiene la función *computeSupportMatches* (1.5.6).

Capítulo 5

Trabajo futuro

Profundizar en el fenómeno que causa el drástico cambio tiempos de ejecución cuando el valor de *candidate_ stepsize* se aproxima a 50 (3.19).

Finalizar la implementación de B2. Para poder así evaluar su comportamiento ante los diferentes parámetros y elaborar una conclusión sobre la optimización aplicada.

Analizar qué imágenes provocan que el algoritmo ELAS responda de forma más rápida o más lenta.

Implementar diferentes puntos del algoritmo de forma paralela, para intentar así mejorar su rendimiento.

Explotar el paralelismo intrínseco del algoritmo ELAS. Como se puede ver en el Apartado 1.5.6, es posible realizar cálculos de forma paralela a nivel de tarea. De esta forma se podría hacer uso simultaneo de todos los dispositivos detectados por OpenCL, es decir, la CPU y todas las GPUs disponibles.

Coordinar el trabajo con el autor del algoritmo, Andreas Geiger. Quién en la presentación de éste, expresó su interés por realizar una implementación en GPUs del algoritmo.

Apéndice A

Tarjetas gráficas utilizadas

Para las implementaciones de OpenCL se ha ejecutado el programa en dos tarjetas gráficas diferentes. Aquí se adjuntan las características técnicas de ambas.

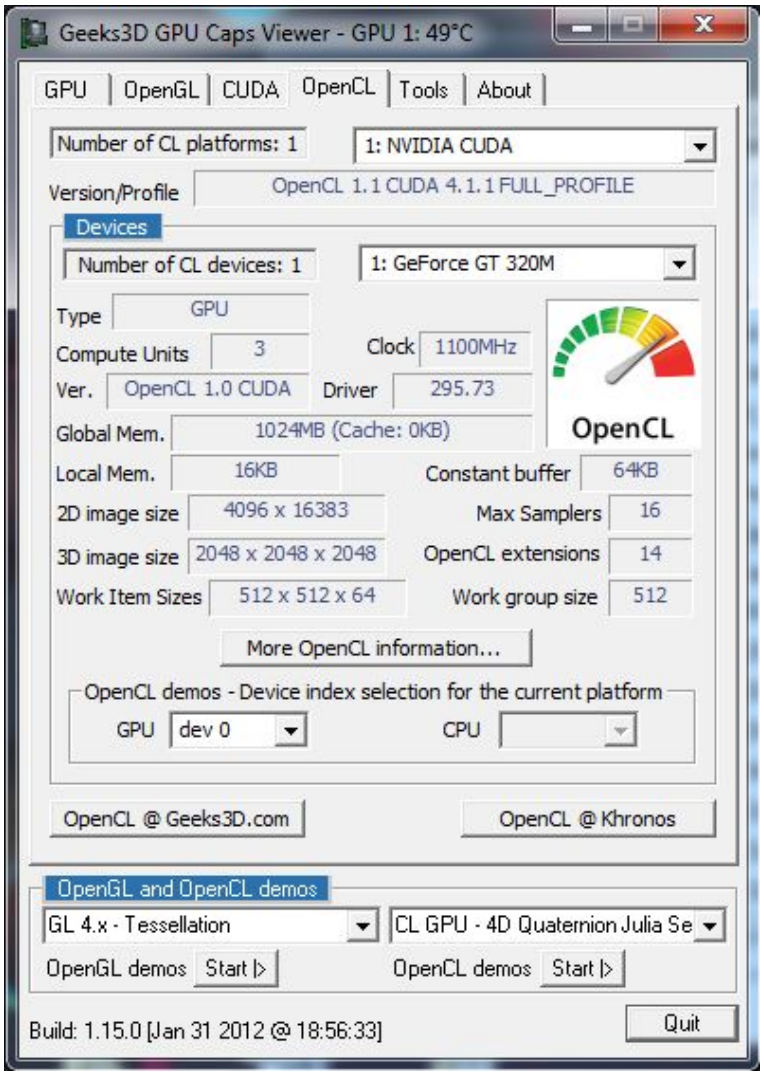


Figura A.1: NVIDIA GT 320M

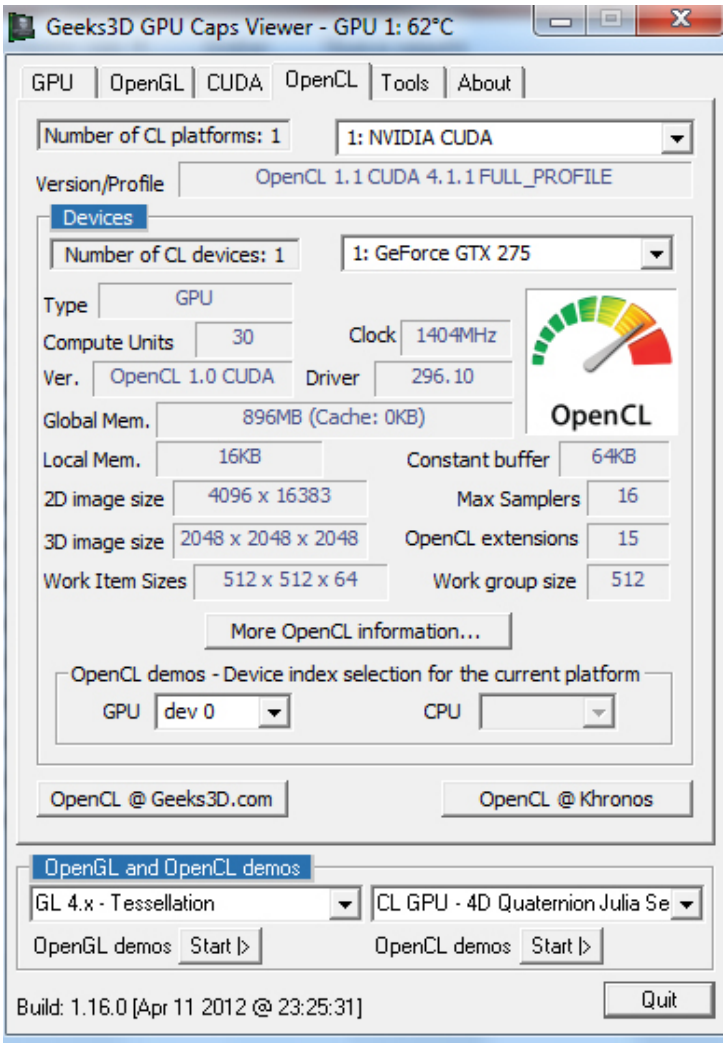


Figura A.2: NVIDIA GTX 275

Apéndice B

Código C Plano y SSE

B.1. Descriptor

Común a ambas implementaciones.

```
1  void Descriptor::createDescriptor (uint8_t* I_du,uint8_t* I_dv,int32_t width,int32_t height,
       int32_t bpl,bool half_resolution)
2  {
3    uint8_t *I_desc_curr;
4    uint32_t addr_v0,addr_v1,addr_v2,addr_v3,addr_v4;
5
6    // do not compute every second line
7    if (half_resolution) {
8
9      // create filter strip
10     for (int32_t v=4; v<height-3; v+=2) {
11       addr_v2 = v*bpl;
12       addr_v0 = addr_v2-2*bpl;
13       addr_v1 = addr_v2-1*bpl;
14       addr_v3 = addr_v2+1*bpl;
15       addr_v4 = addr_v2+2*bpl;
16
17       for (int32_t u=3; u<width-3; u++) {
18         I_desc_curr = I_desc+(v*width+u)*16;
19         *(I_desc_curr++) = *(I_du+addr_v0+u+0);
20         *(I_desc_curr++) = *(I_du+addr_v1+u-2);
21         *(I_desc_curr++) = *(I_du+addr_v1+u+0);
22         *(I_desc_curr++) = *(I_du+addr_v1+u+2);
23         *(I_desc_curr++) = *(I_du+addr_v2+u-1);
24         *(I_desc_curr++) = *(I_du+addr_v2+u+0);
25         *(I_desc_curr++) = *(I_du+addr_v2+u+0);
26         *(I_desc_curr++) = *(I_du+addr_v2+u+1);
27         *(I_desc_curr++) = *(I_du+addr_v3+u-2);
```

```
28          *(I_desc_curr++) = *(I_du+addr_v3+u+0);
29          *(I_desc_curr++) = *(I_du+addr_v3+u+2);
30          *(I_desc_curr++) = *(I_du+addr_v4+u+0);
31          *(I_desc_curr++) = *(I_dv+addr_v1+u+0);
32          *(I_desc_curr++) = *(I_dv+addr_v2+u-1);
33          *(I_desc_curr++) = *(I_dv+addr_v2+u+1);
34          *(I_desc_curr++) = *(I_dv+addr_v3+u+0);
35        }
36      }
37    // compute full descriptor images
38    } else {
39      // create filter strip
40      for (int32_t v=3; v<height-3; v++) {
41        addr_v2 = v*bpl;
42        addr_v0 = addr_v2-2*bpl;
43        addr_v1 = addr_v2-1*bpl;
44        addr_v3 = addr_v2+1*bpl;
45        addr_v4 = addr_v2+2*bpl;
46        for (int32_t u=3; u<width-3; u++) {
47          I_desc_curr = I_desc+(v*width+u)*16;
48          *(I_desc_curr++) = *(I_du+addr_v0+u+0);
49          *(I_desc_curr++) = *(I_du+addr_v1+u-2);
50          *(I_desc_curr++) = *(I_du+addr_v1+u+0);
51          *(I_desc_curr++) = *(I_du+addr_v1+u+2);
52          *(I_desc_curr++) = *(I_du+addr_v2+u-1);
53          *(I_desc_curr++) = *(I_du+addr_v2+u+0);
54          *(I_desc_curr++) = *(I_du+addr_v2+u+0);
55          *(I_desc_curr++) = *(I_du+addr_v2+u+1);
56          *(I_desc_curr++) = *(I_du+addr_v3+u-2);
57          *(I_desc_curr++) = *(I_du+addr_v3+u+0);
58          *(I_desc_curr++) = *(I_du+addr_v3+u+2);
59          *(I_desc_curr++) = *(I_du+addr_v4+u+0);
60          *(I_desc_curr++) = *(I_dv+addr_v1+u+0);
61          *(I_desc_curr++) = *(I_dv+addr_v2+u-1);
62          *(I_desc_curr++) = *(I_dv+addr_v2+u+1);
63          *(I_desc_curr++) = *(I_dv+addr_v3+u+0);
64        }
65      }
66    }
67  }
```

B.2. Support Matches - C Plano

```
1  vector<Elas::support_pt> Elas::computeSupportMatches (uint8_t* I1_desc,uint8_t* I2_desc)
2  {
3    // be sure that at half resolution we only need data
4    // from every second line!
5    int32_t D_candidate_stepsize = param.candidate_stepsize;
6
7    if (param.subsampling)
8      D_candidate_stepsize += D_candidate_stepsize%2;
9
10   // create matrix for saving disparity candidates
11   int32_t D_can_width  = 0;
12   int32_t D_can_height = 0;
13   for (int32_t u=0; u<width;  u+=D_candidate_stepsize)
14     D_can_width++;
15   for (int32_t v=0; v<height; v+=D_candidate_stepsize)
16     D_can_height++;
17   int16_t* D_can = (int16_t*)calloc(D_can_width*D_can_height,sizeof(int16_t));
18
19   // loop variables
20   int32_t u,v;
21   int16_t d,d2;
22
23   // for all point candidates in image 1 do
24   for (int32_t u_can=1; u_can<D_can_width; u_can++)
25   {
26     u = u_can*D_candidate_stepsize;
27     for (int32_t v_can=1; v_can<D_can_height; v_can++)
28     {
29       v = v_can*D_candidate_stepsize;
30       // initialize disparity candidate to invalid
31       *(D_can+getAddressOffsetImage(u_can,v_can,D_can_width)) = -1;
32
33       // find forwards
34       d = computeMatchingDisparity(u,v,I1_desc,I2_desc,false);
35       if (d>=0)
36       {
37         // find backwards
38         d2 = computeMatchingDisparity(u-d,v,I1_desc,I2_desc,true);
39         if (d2>=0 && abs(d-d2)<=param.lr_threshold)
40           *(D_can+getAddressOffsetImage(u_can,v_can,D_can_width)) = d;
41       }
42     }
43   }
44
45   // remove inconsistent support points
46   removeInconsistentSupportPoints(D_can,D_can_width,D_can_height);
47
48   // remove support points on straight lines, since they are redundant
49   // this reduces the number of triangles a little bit and hence speeds up
50   // the triangulation process
51   removeRedundantSupportPoints(D_can,D_can_width,D_can_height,5,1,true);
52   removeRedundantSupportPoints(D_can,D_can_width,D_can_height,5,1,false);
```

```
53
54    // move support points from image representation into a vector representation
55    vector<support_pt> p_support;
56    for (int32_t u_can=1; u_can<D_can_width; u_can++)
57      for (int32_t v_can=1; v_can<D_can_height; v_can++)
58        if (*(D_can+getAddressOffsetImage(u_can,v_can,D_can_width))>=0)
59          p_support.push_back(support_pt(u_can*D_candidate_stepsize,
60            v_can*D_candidate_stepsize,
61            *(D_can+getAddressOffsetImage(u_can,v_can,D_can_width))));
62    // if flag is set, add support points in image corners
63    // with the same disparity as the nearest neighbor support point
64    if (param.add_corners)
65      addCornerSupportPoints(p_support);
66
67    // free memory
68    free(D_can);
69
70    // return support point vector
71    return p_support;
72  }
```

```
1  inline int16_t Elas::computeMatchingDisparity (const int32_t &u,const int32_t &v,uint8_t*
       I1_desc,uint8_t* I2_desc,const bool &right_image)
2  {
3    const int32_t u_step      = 2;
4    const int32_t v_step      = 2;
5    const int32_t window_size = 3;
6
7    int32_t desc_offset_1 = -16*u_step-16*width*v_step;
8    int32_t desc_offset_2 = +16*u_step-16*width*v_step;
9    int32_t desc_offset_3 = -16*u_step+16*width*v_step;
10   int32_t desc_offset_4 = +16*u_step+16*width*v_step;
11
12   uint8_t *mm1,*mm2,*mm3,*mm4,*mm5,*mm6;
13   mm1=(uint8_t*)malloc(16*sizeof(uint8_t));
14   mm2=(uint8_t*)malloc(16*sizeof(uint8_t));
15   mm3=(uint8_t*)malloc(16*sizeof(uint8_t));
16   mm4=(uint8_t*)malloc(16*sizeof(uint8_t));
17   mm5=(uint8_t*)malloc(16*sizeof(uint8_t));
18   mm6=(uint8_t*)malloc(16*sizeof(uint8_t));
19   // check if we are inside the image region
20   if (u>=window_size+u_step && u<=width-window_size-1-u_step &&
21       v>=window_size+v_step && v<=height-window_size-1-v_step)
22   {
23
24     // compute desc and start addresses
25     int32_t  line_offset = 16*width*v;
26     uint8_t *I1_line_addr,*I2_line_addr;
27     if (!right_image) {
28       I1_line_addr = I1_desc+line_offset;
29       I2_line_addr = I2_desc+line_offset;
30     } else {
31       I1_line_addr = I2_desc+line_offset;
32       I2_line_addr = I1_desc+line_offset;
33     }
```

```
34      // compute I1 block start addresses
35      uint8_t* I1_block_addr = I1_line_addr+16*u;
36      uint8_t* I2_block_addr;
37
38      // we require at least some texture
39      int32_t sum = 0;
40      for (int32_t i=0; i<16; i++)
41        sum += abs((int32_t)(*(I1_block_addr+i))-128);
42      if (sum<param.support_texture)
43        return -1;
44
45      // load first blocks to xmm registers
46      memcpy(mm1,I1_block_addr+desc_offset_1,16);
47      memcpy(mm2,I1_block_addr+desc_offset_2,16);
48      memcpy(mm3,I1_block_addr+desc_offset_3,16);
49      memcpy(mm4,I1_block_addr+desc_offset_4,16);
50
51      // declare match energy for each disparity
52      int32_t u_warp;
53
54      // best match
55      int16_t min_1_E = 32767;
56      int16_t min_1_d = -1;
57      int16_t min_2_E = 32767;
58      int16_t min_2_d = -1;
59      // get valid disparity range
60      int32_t disp_min_valid = max(param.disp_min,0);
61      int32_t disp_max_valid = param.disp_max;
62      if (!right_image) disp_max_valid = min(param.disp_max,u-window_size-u_step);
63      else              disp_max_valid = min(param.disp_max,width-u-window_size-u_step);
64
65      // assume, that we can compute at least 10 disparities for this pixel
66      if (disp_max_valid-disp_min_valid<10)
67        return -1;
68      // for all disparities do
69      for (int16_t d=disp_min_valid; d<=disp_max_valid; d++) {
70        // warp u coordinate
71        if (!right_image) u_warp = u-d;
72        else              u_warp = u+d;
73        // compute I2 block start addresses
74        I2_block_addr = I2_line_addr+16*u_warp;
75        // compute match energy at this disparity
76        // http://www.rz.uni-karlsruhe.de/rz/docs/VTune/reference/vc250.htm
77        memcpy(mm6,I2_block_addr+desc_offset_1,16);
78        uint16_t aux1=0;
79        uint16_t aux2=0;
80        for (int i = 0; i<16; i++){
81        if (i<8)
82          aux1+=abs(extract_8(mm1,i) - extract_8(mm6,i));
83        else
84          aux2+=abs(extract_8(mm1,i) - extract_8(mm6,i));
85        mm6[i] = (uint8_t)0;
86        }
87        ((uint16_t*)mm6)[0]=aux1;
88        ((uint16_t*)mm6)[4]=aux2;
```

```
89          memcpy(mm5,I2_block_addr+desc_offset_2,16);
90          aux1=0;
91          aux2=0;
92          for (int i = 0; i<16; i++){
93          if (i<8)
94            aux1+=abs(extract_8(mm2,i) - extract_8(mm5,i));
95          else
96            aux2+=abs(extract_8(mm2,i) - extract_8(mm5,i));
97          }
98          ((uint16_t*)mm6)[0]=extract_16(mm6,0)+aux1;
99          ((uint16_t*)mm6)[4]=extract_16(mm6,4)+aux2;
100         memcpy(mm5,I2_block_addr+desc_offset_3,16);
101         aux1=0;
102         aux2=0;
103         for (int i = 0; i<16; i++){
104         if (i<8)
105           aux1+=abs(extract_8(mm3,i) - extract_8(mm5,i));
106         else
107           aux2+=abs(extract_8(mm3,i) - extract_8(mm5,i));
108         }
109         ((uint16_t*)mm6)[0]=extract_16(mm6,0)+aux1;
110         ((uint16_t*)mm6)[4]=extract_16(mm6,4)+aux2;
111         memcpy(mm5,I2_block_addr+desc_offset_4,16);
112         aux1=0;
113         aux2=0;
114         for (int i = 0; i<16; i++){
115         if (i<8)
116           aux1+=abs(extract_8(mm4,i) - extract_8(mm5,i));
117         else
118           aux2+=abs(extract_8(mm4,i) - extract_8(mm5,i));
119         }
120         ((uint16_t*)mm6)[0]=extract_16(mm6,0)+aux1;
121         ((uint16_t*)mm6)[4]=extract_16(mm6,4)+aux2;
122         sum = extract_16(mm6,0) + extract_16(mm6,4);
123         // best + second best match
124         if (sum<min_1_E) {
125           min_1_E = sum;
126           min_1_d = d;
127         } else if (sum<min_2_E) {
128           min_2_E = sum;
129           min_2_d = d;
130         }
131       }
132       // check if best and second best match are available and if matching ratio is sufficient
133       if (min_1_d>=0 && min_2_d>=0 && (float)min_1_E<param.support_threshold*(float)min_2_E)
134         return min_1_d;
135       else
136         return -1;
137
138     } else
139       return -1;
140   }
```

B.3. Support Matches - SSE

```
1  vector<Elas::support_pt> Elas::computeSupportMatches (uint8_t* I1_desc,uint8_t* I2_desc)
2  {
3    // be sure that at half resolution we only need data
4    // from every second line!
5    int32_t D_candidate_stepsize = param.candidate_stepsize;
6
7    if (param.subsampling)
8      D_candidate_stepsize += D_candidate_stepsize%2;
9
10   // create matrix for saving disparity candidates
11   int32_t D_can_width  = 0;
12   int32_t D_can_height = 0;
13   for (int32_t u=0; u<width;  u+=D_candidate_stepsize)
14     D_can_width++;
15   for (int32_t v=0; v<height; v+=D_candidate_stepsize)
16     D_can_height++;
17   int16_t* D_can = (int16_t*)calloc(D_can_width*D_can_height,sizeof(int16_t));
18
19   // loop variables
20   int32_t u,v;
21   int16_t d,d2;
22
23   // for all point candidates in image 1 do
24   for (int32_t u_can=1; u_can<D_can_width; u_can++)
25   {
26     u = u_can*D_candidate_stepsize;
27     for (int32_t v_can=1; v_can<D_can_height; v_can++)
28     {
29       v = v_can*D_candidate_stepsize;
30       // initialize disparity candidate to invalid
31       *(D_can+getAddressOffsetImage(u_can,v_can,D_can_width)) = -1;
32
33       // find forwards
34       d = computeMatchingDisparity(u,v,I1_desc,I2_desc,false);
35       if (d>=0)
36       {
37         // find backwards
38         d2 = computeMatchingDisparity(u-d,v,I1_desc,I2_desc,true);
39         if (d2>=0 && abs(d-d2)<=param.lr_threshold)
40           *(D_can+getAddressOffsetImage(u_can,v_can,D_can_width)) = d;
41       }
42     }
43   }
44
45   // remove inconsistent support points
46   removeInconsistentSupportPoints(D_can,D_can_width,D_can_height);
47
48   // remove support points on straight lines, since they are redundant
49   // this reduces the number of triangles a little bit and hence speeds up
50   // the triangulation process
51   removeRedundantSupportPoints(D_can,D_can_width,D_can_height,5,1,true);
52   removeRedundantSupportPoints(D_can,D_can_width,D_can_height,5,1,false);
```

```

  // move support points from image representation into a vector representation
  vector<support_pt> p_support;
  for (int32_t u_can=1; u_can<D_can_width; u_can++)
    for (int32_t v_can=1; v_can<D_can_height; v_can++)
      if (*(D_can+getAddressOffsetImage(u_can,v_can,D_can_width))>=0)
        p_support.push_back(support_pt(u_can*D_candidate_stepsize,
          v_can*D_candidate_stepsize,
          *(D_can+getAddressOffsetImage(u_can,v_can,D_can_width))));
  // if flag is set, add support points in image corners
  // with the same disparity as the nearest neighbor support point
  if (param.add_corners)
    addCornerSupportPoints(p_support);

  // free memory
  free(D_can);

  // return support point vector
  return p_support;
}
```

```
inline int16_t Elas::computeMatchingDisparity (const int32_t &u,const int32_t &v,uint8_t*
    I1_desc,uint8_t* I2_desc,const bool &right_image) {
  const int32_t u_step      = 2;
  const int32_t v_step      = 2;
  const int32_t window_size = 3;
  int32_t desc_offset_1 = -16*u_step-16*width*v_step;
  int32_t desc_offset_2 = +16*u_step-16*width*v_step;
  int32_t desc_offset_3 = -16*u_step+16*width*v_step;
  int32_t desc_offset_4 = +16*u_step+16*width*v_step;
  __m128i xmm1,xmm2,xmm3,xmm4,xmm5,xmm6;
  // check if we are inside the image region
  if (u >= window_size + u_step &&
      u <= width - window_size - 1 - u_step &&
      v >= window_size + v_step &&
      v <= height - window_size - 1 - v_step) {
    // compute desc and start addresses
    int32_t  line_offset = 16*width*v;
    uint8_t *I1_line_addr,*I2_line_addr;
    if (!right_image) {
      I1_line_addr = I1_desc+line_offset;
      I2_line_addr = I2_desc+line_offset;
    } else {
      I1_line_addr = I2_desc+line_offset;
      I2_line_addr = I1_desc+line_offset;
    }
    // compute I1 block start addresses
    uint8_t* I1_block_addr = I1_line_addr+16*u;
    uint8_t* I2_block_addr;
    // we require at least some texture
    int32_t sum = 0;
    for (int32_t i=0; i<16; i++)
      sum += abs((int32_t)(*(I1_block_addr+i))-128);
    if (sum<param.support_texture)
      return -1;
```

```
34      // load first blocks to xmm registers
35      xmm1 = _mm_load_si128((__m128i*)(I1_block_addr+desc_offset_1));
36      xmm2 = _mm_load_si128((__m128i*)(I1_block_addr+desc_offset_2));
37      xmm3 = _mm_load_si128((__m128i*)(I1_block_addr+desc_offset_3));
38      xmm4 = _mm_load_si128((__m128i*)(I1_block_addr+desc_offset_4));
39      // declare match energy for each disparity
40      int32_t u_warp;
41      // best match
42      int16_t min_1_E = 32767;
43      int16_t min_1_d = -1;
44      int16_t min_2_E = 32767;
45      int16_t min_2_d = -1;
46      // get valid disparity range
47      int32_t disp_min_valid = max(param.disp_min,0);
48      int32_t disp_max_valid = param.disp_max;
49      if (!right_image) disp_max_valid = min(param.disp_max,u-window_size-u_step);
50      else              disp_max_valid = min(param.disp_max,width-u-window_size-u_step);
51      // assume, that we can compute at least 10 disparities for this pixel
52      if (disp_max_valid-disp_min_valid<10)
53        return -1;
54      // for all disparities do
55      for (int16_t d=disp_min_valid; d<=disp_max_valid; d++) {
56        // warp u coordinate
57        if (!right_image) u_warp = u-d;
58        else              u_warp = u+d;
59        // compute I2 block start addresses
60        I2_block_addr = I2_line_addr+16*u_warp;
61        // compute match energy at this disparity
62        xmm6 = _mm_load_si128((__m128i*)(I2_block_addr+desc_offset_1));
63        xmm6 = _mm_sad_epu8(xmm1,xmm6);
64        xmm5 = _mm_load_si128((__m128i*)(I2_block_addr+desc_offset_2));
65        xmm6 = _mm_add_epi16(_mm_sad_epu8(xmm2,xmm5),xmm6);
66        xmm5 = _mm_load_si128((__m128i*)(I2_block_addr+desc_offset_3));
67        xmm6 = _mm_add_epi16(_mm_sad_epu8(xmm3,xmm5),xmm6);
68        xmm5 = _mm_load_si128((__m128i*)(I2_block_addr+desc_offset_4));
69        xmm6 = _mm_add_epi16(_mm_sad_epu8(xmm4,xmm5),xmm6);
70        sum  = _mm_extract_epi16(xmm6,0)+_mm_extract_epi16(xmm6,4);
71        // best + second best match
72        if (sum<min_1_E) {
73          min_1_E = sum;
74          min_1_d = d;
75        } else if (sum<min_2_E) {
76          min_2_E = sum;
77          min_2_d = d;
78        }
79      }
80      // check if best and second best match are available and if matching ratio is sufficient
81      if (min_1_d>=0 && min_2_d>=0 && (float)min_1_E<param.support_threshold*(float)min_2_E)
82        return min_1_d;
83      else
84        return -1;
85    } else
86      return -1;
87  }
```

Apéndice C

Código OpenCL

C.1. Código optimización A1

```
1  //////////////////////////////////////////////////////////////////////////////
2  // Common definitions
3  //////////////////////////////////////////////////////////////////////////////
4  typedef char  int8_t;
5  typedef short int16_t;
6  typedef int     int32_t;
7  typedef long  int64_t;
8  typedef uchar uint8_t;
9  typedef ushort  uint16_t;
10 typedef uint  uint32_t;
11 typedef ulong uint64_t;
12 //////////////////////////////////////////////////////////////////////////////
13 // Kernels
14 //////////////////////////////////////////////////////////////////////////////
15 __kernel void CreateDescriptor( __global uint8_t* d_I_du,
16   __global uint8_t* d_I_dv, int32_t width, int32_t height, int32_t bpl,
17   __global uint8_t* d_I_desc)
18 {
19   int32_t u = get_global_id(0) + 3;
20   int32_t v = get_global_id(1) + 3;
21   int32_t index = (v*width+u)*16;
22   uint32_t addr_v0,addr_v1,addr_v2,addr_v3,addr_v4;
23
24   addr_v2 = v*bpl;
25   addr_v0 = addr_v2-2*bpl;
26   addr_v1 = addr_v2-1*bpl;
27   addr_v3 = addr_v2+1*bpl;
28   addr_v4 = addr_v2+2*bpl;
29   d_I_desc[index++] = d_I_du[addr_v0+u+0];
30   d_I_desc[index++] = d_I_du[addr_v1+u-2];
31   d_I_desc[index++] = d_I_du[addr_v1+u+0];
32   d_I_desc[index++] = d_I_du[addr_v1+u+2];
```

```
33      d_I_desc[index++] = d_I_du[addr_v2+u-1];
34      d_I_desc[index++] = d_I_du[addr_v2+u+0];
35      d_I_desc[index++] = d_I_du[addr_v2+u+0];
36      d_I_desc[index++] = d_I_du[addr_v2+u+1];
37      d_I_desc[index++] = d_I_du[addr_v3+u-2];
38      d_I_desc[index++] = d_I_du[addr_v3+u+0];
39      d_I_desc[index++] = d_I_du[addr_v3+u+2];
40      d_I_desc[index++] = d_I_du[addr_v4+u+0];
41      d_I_desc[index++] = d_I_dv[addr_v1+u+0];
42      d_I_desc[index++] = d_I_dv[addr_v2+u-1];
43      d_I_desc[index++] = d_I_dv[addr_v2+u+1];
44      d_I_desc[index++] = d_I_dv[addr_v3+u+0];
45  }
```

C.2. Código optimización A2

```
1  __kernel void CreateDescriptor(__global uint8_t* d_I_du, __global uint8_t* d_I_dv, int32_t
       width, int32_t height, int32_t bpl, __global uint8_t* d_I_desc, __local uint8_t* du,
       __local uint8_t* dv)
2  {
3    // There are local and global coordinates!
4    int32_t u = get_global_id(0) + 3;
5    int32_t v = get_global_id(1) + 3;
6    int32_t lu = get_local_id(0) + 2; // x va de 2 a 33 y el array es de [0,35]
7    int32_t lv = get_local_id(1) + 2; // y va de 2 a 9  y el array es de [0,11]
8
9    if ((u>width-4) || (v>height-4))
10     return;
11
12   // Own coordinates
13   du[GET_OFFSET(lu,lv,36)] = d_I_du[GET_OFFSET(u,v,bpl)];
14
15   // Lateral sweep
16   if (lu < 4 )    du[GET_OFFSET(lu-2,lv   ,36)] = d_I_du[GET_OFFSET(u-2,v   ,bpl)];
17   if (lu > 31)    du[GET_OFFSET(lu+2,lv   ,36)] = d_I_du[GET_OFFSET(u+2,v   ,bpl)];
18   if (lv < 4 )    du[GET_OFFSET(lu   ,lv-2,36)] = d_I_du[GET_OFFSET(u   ,v-2,bpl)];
19   if (lv > 7 )    du[GET_OFFSET(lu   ,lv+2,36)] = d_I_du[GET_OFFSET(u   ,v+2,bpl)];
20
21   // Corners
22   if (lu < 5  && lv < 3 )
23     du[GET_OFFSET(lu-2,lv-1,36)] = d_I_du[GET_OFFSET(u-2,v-1,bpl)];
24   if (lu > 29)
25     du[GET_OFFSET(lu+2,lv-1,36)] = d_I_du[GET_OFFSET(u+2,v-1,bpl)];
26   if (lu < 5  && lv > 7 )
27     du[GET_OFFSET(lu-2,lv+1,36)] = d_I_du[GET_OFFSET(u-2,v+1,bpl)];
28   if (lu > 29  && lv > 7 )
29     du[GET_OFFSET(lu+2,lv+1,36)] = d_I_du[GET_OFFSET(u+2,v+1,bpl)];
30
31   barrier(CLK_LOCAL_MEM_FENCE);
32   int32_t index = (v*width+u)*16;
33   uint32_t addr_v0,addr_v1,addr_v2,addr_v3,addr_v4;
34
35   addr_v2 = lv*36;
36   addr_v0 = addr_v2-2*36;
37   addr_v1 = addr_v2-1*36;
38   addr_v3 = addr_v2+1*36;
39   addr_v4 = addr_v2+2*36;
40
41   d_I_desc[index++] = du[addr_v0+lu+0];
42   d_I_desc[index++] = du[addr_v1+lu-2];
43   d_I_desc[index++] = du[addr_v1+lu+0];
44   d_I_desc[index++] = du[addr_v1+lu+2];
45   d_I_desc[index++] = du[addr_v2+lu-1];
46   d_I_desc[index++] = du[addr_v2+lu+0];
47   d_I_desc[index++] = du[addr_v2+lu+0];
48   d_I_desc[index++] = du[addr_v2+lu+1];
49   d_I_desc[index++] = du[addr_v3+lu-2];
50   d_I_desc[index++] = du[addr_v3+lu+0];
```

```
51      d_I_desc[index++] = du[addr_v3+lu+2];
52      d_I_desc[index++] = du[addr_v4+lu+0];
53      d_I_desc[index++] = d_I_dv[((v-1)*bpl)+u+0];
54      d_I_desc[index++] = d_I_dv[(v*bpl)+u-1];
55      d_I_desc[index++] = d_I_dv[(v*bpl)+u+1];
56      d_I_desc[index++] = d_I_dv[((v+1)*bpl)+u+0];
57  }
```

C.3. Código optimización B1

```
1  // REMEMBER: MACROS CANNOT BE OVERLOADED!!
2  // We use so many macros because we need to save on local variables and it is
3  // better to calculate everything each time.
4  /// This means groups of 16 and image width 32.
5  #define GET_OFFSET_G16_W32(x,y) \
6    (((y)<<5+(x))<<4)
7  #define GET_OFFSET_G16(x,y,width) \
8    (((y)*(width)+(x))<<4)
9  #define GET_OFFSET(x,y,width) \
10   ((y)*(width)+(x))
11 #define GET_OFFSET_G(x,y,group_size,width) \
12   ((y)*(group_size)*(width)+(x)*(group_size))
13 #define ADD_OFFSET(base,x,y,width) \
14   ((base)+(y)*(width)+(x))
15 #define ADD_OFFSET_W(base,x,y,word,width) \
16   ((base)+(y)*(word)*(width)+(x)*(word))
17
18 //-------------------------------------------------------------------------
19 //-- Support points
20 // WARNING: If you change this, dont forget to update its equivalent on elas.h
21 typedef struct {
22   // used by ComputeSupportMatches
23   int32_t candidate_stepsize;
24   int32_t D_can_width;
25   int32_t D_can_height;
26
27   // used by computeMatchingDisparity
28   int32_t image_width;
29   int32_t image_height;
30   int32_t disp_min;
31   int32_t disp_max;
32   int32_t lr_threshold;
33   float   support_threshold;
34   int32_t support_texture;
35 } sm_params_t;
36
37 #define U_STEP 2
38 #define V_STEP 2
39 #define WINDOW_SIZE 3
40 #define MIN_1_E = (32767);
41 #define MIN_1_D = (-1);
42 #define MIN_2_E = (32767);
43 #define MIN_2_D = (-1);
44 #define DESC_OFFSET(dx,dy,width) \
45   ((dx)*16*U_STEP+(dy)*(width)*V_STEP)
46 #define DESC_OFFSET_1  DESC_OFFSET(-1,-1,params->image_width)
47 #define DESC_OFFSET_2  DESC_OFFSET(+1,-1,params->image_width)
48 #define DESC_OFFSET_3  DESC_OFFSET(-1,+1,params->image_width)
49 #define DESC_OFFSET_4  DESC_OFFSET(+1,+1,params->image_width)
50
51 #define I1_DESC ((!right_image) ? d_I_desc1 : d_I_desc2)
52 #define I2_DESC ((!right_image) ? d_I_desc2 : d_I_desc1)
```

```
53 #define I1_LINE_ADDR  ADD_OFFSET(d_I_desc_target,0,v,16,params->image_width)
54 #define I2_LINE_ADDR  ADD_OFFSET(d_I_desc_ref,0,v,16,params->image_width)
55 #define LINE_INDEX ((v)*(params->image_width))
56 // These macros do not take into account the size of the word:
57 //   IOFFSET means INDEX OFFSET
58 #define GET_IMAGE_IOFFSET(x,y) \
59   GET_OFFSET(x,y,params->image_width)
60 #define ADD_IMAGE_IOFFSET(base,x,y) \
61   ADD_OFFSET(base,x,y,params->image_width)
62 #define GET_IMAGE_DESC_IOFFSET(x,y) \
63   GET_OFFSET_W(x,y,16,params->image_width)
64 #define ADD_IMAGE_DESC_IOFFSET(base,x,y) \
65   ADD_OFFSET_W(base,x,y,16,params->image_width)
66
67 //TODO: code this better!
68 uint16_t addAllTerms(uchar16 vector)
69 {
70   uint16_t accumulator = 0;
71   accumulator += vector.s0;
72   accumulator += vector.s1;
73   accumulator += vector.s2;
74   accumulator += vector.s3;
75   accumulator += vector.s4;
76   accumulator += vector.s5;
77   accumulator += vector.s6;
78   accumulator += vector.s7;
79   accumulator += vector.s8;
80   accumulator += vector.s9;
81   accumulator += vector.sA;
82   accumulator += vector.sB;
83   accumulator += vector.sC;
84   accumulator += vector.sD;
85   accumulator += vector.sE;
86   accumulator += vector.sF;
87   return accumulator;
88 }
89
90 int16_t computeMatchingDisparity( __global __const uint8_t * d_I_desc1, __global __const
       uint8_t * d_I_desc2, __global __const sm_params_t * params, int32_t u, int32_t v, bool
       right_image)
91 {
92   // check if we are inside the image region
93   if (u>=WINDOW_SIZE+U_STEP &&
94   u<=params->image_width-WINDOW_SIZE-1-U_STEP &&
95   v>=WINDOW_SIZE+V_STEP &&
96   v<=params->image_height-WINDOW_SIZE-1-V_STEP)
97   {
98     // we require at least some texture
99     int32_t sum = 0;
100    for (int32_t i=0; i<16; i++)
101      sum += abs(((int32_t) I1_DESC[GET_IMAGE_DESC_IOFFSET(u,v)+i]) - 128);
102    if (sum<params->support_texture)
103      return -1;
104
105    // get valid disparity range
```

```
106       int32_t disp_min_valid = max(params->disp_min,0);
107       int32_t disp_max_valid = params->disp_max;
108       if (!right_image)
109         disp_max_valid = min(disp_max_valid,u-WINDOW_SIZE-U_STEP);
110       else
111         disp_max_valid = min(disp_max_valid,
112       params->image_width-u-WINDOW_SIZE-U_STEP);
113
114       // assume, that we can compute at least 10 disparities for this pixel
115       if (disp_max_valid-disp_min_valid<10)
116         return -1;
117
118       uchar16 xmm1,xmm2,xmm3,xmm4,xmm5;
119       // load first blocks to xmm registers
120       xmm1 = vload16(0,&(I1_DESC[GET_IMAGE_DESC_IOFFSET(u - U_STEP, v - V_STEP)]));
121       xmm2 = vload16(0,&(I1_DESC[GET_IMAGE_DESC_IOFFSET(u + U_STEP, v - V_STEP)]));
122       xmm3 = vload16(0,&(I1_DESC[GET_IMAGE_DESC_IOFFSET(u - U_STEP, v + V_STEP)]));
123       xmm4 = vload16(0,&(I1_DESC[GET_IMAGE_DESC_IOFFSET(u + U_STEP, v + V_STEP)]));
124
125       // declare match energy for each disparity
126       int32_t u_warp;
127
128       // best match
129       int16_t min_1_E = 32767;
130       int16_t min_1_d = -1;
131       int16_t min_2_E = 32767;
132       int16_t min_2_d = -1;
133
134       // for all disparities do
135       for (int16_t d=disp_min_valid; d<=disp_max_valid; d++) {
136         // warp u coordinate
137         if (!right_image) u_warp = u-d;
138         else              u_warp = u+d;
139
140         sum=0;
141         // compute match energy at this disparity
142         xmm5 = vload16(0,&(I2_DESC[GET_IMAGE_DESC_IOFFSET(u_warp - U_STEP, v - V_STEP)]));
143         xmm5 = abs_diff(xmm1, xmm5);
144         sum += addAllTerms(xmm5);
145         xmm5 = vload16(0,&(I2_DESC[GET_IMAGE_DESC_IOFFSET(u_warp + U_STEP, v - V_STEP)]));
146         xmm5 = abs_diff(xmm2, xmm5);
147         sum += addAllTerms(xmm5);
148         xmm5 = vload16(0,&(I2_DESC[GET_IMAGE_DESC_IOFFSET(u_warp - U_STEP, v + V_STEP)]));
149         xmm5 = abs_diff(xmm3, xmm5);
150         sum += addAllTerms(xmm5);
151         xmm5 = vload16(0,&(I2_DESC[GET_IMAGE_DESC_IOFFSET(u_warp + U_STEP, v + V_STEP)]));
152         xmm5 = abs_diff(xmm4, xmm5);
153         sum += addAllTerms(xmm5);
154
155         // best + second best match
156         if (sum<min_1_E)
157         {
158           min_1_E = sum;
159           min_1_d = d;
160         }
```

```
161         else
162           if (sum<min_2_E)
163           {
164             min_2_E = sum;
165             min_2_d = d;
166           }
167       }
168
169       // check if best and second best match are available and if matching ratio
170       // is sufficient
171       if (min_1_d>=0 && min_2_d>=0 &&
172         isless((float) min_1_E,(float) (params->support_threshold)*min_2_E))
173         return min_1_d;
174       else
175         return -1;
176     } else
177     return -1;
178   }
179
180   __kernel void ComputeSupportMatches(__global uint16_t* d_D_can, __global __const uint8_t *
        d_I_desc1, __global __const uint8_t * d_I_desc2, __global __const sm_params_t * params)
181   {
182     int32_t u_can = get_global_id(0);
183     int32_t v_can = get_global_id(1);
184     if (u_can > params->D_can_width || v_can > params->D_can_height)
185       return;
186     int32_t u = u_can * params->candidate_stepsize;
187     int32_t v = v_can * params->candidate_stepsize;
188     int16_t d,d2, result;
189
190     // initialize disparity candidate to invalid
191     result = -1;
192     // find forwards
193     d = computeMatchingDisparity(d_I_desc1, d_I_desc2, params, u, v, false);
194
195     if (d>=0)   {
196       // find backwards
197       d2 = computeMatchingDisparity(d_I_desc1, d_I_desc2, params, u-d, v, true);
198
199       if (d2>=0 && abs(d-d2)<=params->lr_threshold)
200         result = d;
201     }
202     d_D_can[GET_OFFSET(u_can,v_can,params->D_can_width)] = result;
203   }
```

C.4. Código optimización B2

```
1  // REMEMBER: MACROS CANNOT BE OVERLOADED!!
2  // We use so many macros because we need to save on local variables and it is
3  // better to calculate everything each time.
4  /// This means groups of 16 and image width 32.
5  #define GET_OFFSET_G16_W32(x,y) \
6    (((y)<<5+(x))<<4)
7  #define GET_OFFSET_G16(x,y,width) \
8    (((y)*(width)+(x))<<4)
9  #define GET_OFFSET(x,y,width) \
10   ((y)*(width)+(x))
11 #define GET_OFFSET_G(x,y,group_size,width) \
12   ((y)*(group_size)*(width)+(x)*(group_size))
13 #define ADD_OFFSET(base,x,y,width) \
14   ((base)+(y)*(width)+(x))
15 #define ADD_OFFSET_W(base,x,y,word,width) \
16   ((base)+(y)*(word)*(width)+(x)*(word))
17
18 //-------------------------------------------------------------------------------
19 //-- Support points
20 //---------------------------------------
21
22 //-- Common
23 #define U_STEP 2
24 #define V_STEP 2
25 #define WINDOW_SIZE 3
26 // These macros do not take into account the size of the word:
27 //   IOFFSET means INDEX OFFSET
28 #define GET_IMAGE_IOFFSET(x,y)
29   GET_OFFSET(x,y,params->image_width)
30 #define ADD_IMAGE_IOFFSET(base,x,y)
31   ADD_OFFSET(base,x,y,params->image_width)
32 // Descriptor is a structure of 8bit elements grouped in 16.
33 #define GET_DESC_IOFFSET(x,y) \
34   GET_OFFSET_G(x,y,16,params->image_width)
35 #define ADD_IMAGE_DESC_IOFFSET(base,x,y) \
36   ADD_OFFSET_W(base,x,y,16,params->image_width)
37 //---------------------------------------
38 //-- Used by ComputeMatchingDisparity
39 #define TI_DESC ((!right_image) ? d_I_desc1 : d_I_desc2) // Target image
40 #define RI_DESC ((!right_image) ? d_I_desc2 : d_I_desc1)  // Reference image
41 #define IS_INSIDE_IMAGE(u,v) \
42       (u >= WINDOW_SIZE + U_STEP && \
43       u <= params->image_width - WINDOW_SIZE - 1 - U_STEP && \
44       v >= WINDOW_SIZE + V_STEP && \
45       v <= params->image_height - WINDOW_SIZE - 1 - V_STEP)
46 //---------------------------------------
47 ///@warning If you change this, don't forget to update its equivalent on elas.h
48 typedef struct {
49   // used by ComputeSupportMatches
50   int32_t candidate_stepsize;
51   int32_t D_can_width;
52   int32_t D_can_height;
```

```
53    // used by computeMatchingDisparity
54    int32_t image_width;
55    int32_t image_height;
56    int32_t disp_min;
57    int32_t disp_max;
58    int32_t lr_threshold;
59    float   support_threshold;
60    int32_t support_texture;
61  } sm_params_t;
62
63  ///@todo Code this better!
64  uint16_t addAllTerms(uchar16 vector) {
65    // I don't use :
66    //   sum = dot(v, (float4)(1))
67    // because dot works with float/double/half not uchar.
68
69    uint16_t accumulator = 0;
70    accumulator += vector.s0;
71    accumulator += vector.s1;
72    accumulator += vector.s2;
73    accumulator += vector.s3;
74    accumulator += vector.s4;
75    accumulator += vector.s5;
76    accumulator += vector.s6;
77    accumulator += vector.s7;
78    accumulator += vector.s8;
79    accumulator += vector.s9;
80    accumulator += vector.sA;
81    accumulator += vector.sB;
82    accumulator += vector.sC;
83    accumulator += vector.sD;
84    accumulator += vector.sE;
85    accumulator += vector.sF;
86    return accumulator;
87  }
88
89  ///@file
90  ///
91  ///@par Support points section
92  ///
93  ///@par Cache Matrix:
94  ///   Each work group uses a certain amount of pixels that must be fetched
95  ///   from the reference image.
96  ///   To speed up the process of accessing them we created the structure
97  ///   Cache Matrix.
98  ///   Since the comparison for pixel similarity is done used the Descriptor
99  ///   structure, the elements of the Cache Matrix must be the same as those
100 ///   in the Descriptor, vectors of 16 uint8_t.
101 ///   REMEMBER: All information stored is from the reference image.
102 ///
103 ///@par Acronyms used in Macros:
104 ///   - DESC: References the Descriptor structure, they always reference a
105 ///     vector of 16 uint8_t.
106 ///   - WG: Work Group
107 ///   - CM: Cache Matrix. (Thus, references a vector of 16 uint8_t)
```

```
108 ///   - BASE: References the pixels that have to be fetched for computing the
109 ///         first disparity of each work group. Also called Base Section.
110 ///   - IMAGE: Is related to the actual image. (Thus, image coordinates)
111 ///   - WIT: Warp Iteration. The task of fetching the additional pixels'
112 ///       descriptors, besides of the Base, to the Cache Matrix, is
113 ///       sequentialized by the warps. This sequentialization is done by a loop
114 ///       named Warp Iteration.
115 ///   - IC: Image coordinates.
116 ///   - RC: Relative coordinates. Mostly used to reference the Cache Matrix.
117 ///   - WI_CM_OWN: Makes reference to the section of the Cache Matrix used by
118 ///     the current warp.
119 // -- Common
120 #define WARP_SIZE            32
121 #define SNAP(a,b)  (((((int)a) %((int)b)) == 0) ? ((int)a) : (((int)a) - (((int)a) %((int)
        b)) + ((int)b)))
122 #define IMAGE_XCOORDINATE(u) ((u) * params->candidate_stepsize)
123 #define IMAGE_YCOORDINATE(y) ((y) * params->candidate_stepsize)
124 #define WG_LEFTEST_GLOBAL_ID  \
125   (get_global_id(0) - get_local_id(0))
126 #define WG_RIGHTEST_GLOBAL_ID \
127   ((int) get_global_id(0) - (int) get_local_id(0) + (int) get_local_size(0) - 1)
128
129 ///@warning Remember the work group size must be x*32 in the first dimension.
130 #define WG_WARPS (get_local_size(0) / WARP_SIZE)
131 #define WARP_ID (convert_int(floor((float) get_local_id(0) / WARP_SIZE)))
132
133 // -- Cache Matrix - Common
134 // CM_WIDTH =
135 //     Math.floor[(LocalMemory - Externally_Needed_LocalMemory) /
136 //                (16 * CACHEMATRIX_ROWS)]
137 //#define CM_WIDTH  (Math.floor((LOCALMEMORY_SIZE-16)/(sizeof(uchar16)*2)))
138 //#define CM_WIDTH  (SNAP(params->disp_max,32) + params->candidate_stepsize*(int)
        get_local_size(0))
139
140 // The Descriptor always receives image coodinates and always returns a vector 16 of uint8_t
141 ///@todo  Try using: ADD_DESC_OFFSET
142 #define GET_CM_IOFFSET(x,y) \
143   ((y >> 1) * cm_width + (x %cm_width))
144 #define CACHEMATRIX_GET(x,y) \
145   cache_matrix[GET_CM_IOFFSET(x,y) << 4]
146
147 /// The offset get multiplied by 16 in the vloadn/vstoren call =).
148 #define CM_SET_FROM_DESC(x,y,desc,desc_x,desc_y) \
149       vstore16( vload16(GET_IMAGE_IOFFSET((desc_x), desc_y),desc), \
150       GET_CM_IOFFSET((x),(y)), \
151       cache_matrix)
152
153 // -- Cache Matrix - Base section
154 #define THREAD_BASE_ASIGNEE(i) (wg_size * ((int)(i)) + wi_lid)
155 // -- Cache Matrix - Warp's own section
156 #define WI_CM_OWN_WIDTH (cm_wit_right_ic - cm_wit_left_ic)
157 #define THREAD_OWN_ASIGNEE (params->candidate_stepsize * get_global_id(0))
158 // -- Disparity computing
159 #define ADD_DESC_OFFSET(desc, dx, dy) \
160   ( (desc + ((params->image_width * dy + dx) * (16))) )
```

```
161
162 ///@todo Make sure all possible divisions are shifts. For example x/32 should be x>>5.
163 ///@param u_displacement Displacement on the actual image!
164 ///                      (not on the D_can matrix)
165 ///@param right_image Defines whether the right image is to be considered the
166 ///         target image or not.
167 ///@param target_cache Used for fetching the target image pixels.
168 ///                    Treat as vector of arrays instead of array of vectors.
169
170 int16_t computeMatchingDisparityForCandidate(__global __const uint8_t * d_I_desc1, __global
      __const uint8_t * d_I_desc2, __global __const sm_params_t * params, int32_t u_can,
      int32_t u_displacement, int32_t v_can, bool right_image, __local uint8_t* cache_matrix,
      __local uint8_t* target_cache)
171 {
172   int32_t u = (u_can * params->candidate_stepsize) + u_displacement;
173   int32_t v = (v_can * params->candidate_stepsize);
174   uint16_t wi_lid   = get_local_id(0);
175
176   // work item local id
177   uint16_t wg_size  = get_local_size(0);
178
179   // work group size
180   uint32_t cm_width = (SNAP(params->disp_max,32) + params->candidate_stepsize*wg_size);
181
182   // check if we are inside the image region
183   if (!IS_INSIDE_IMAGE(u,v))
184     return -1;
185
186   //{ Cache Matrix - Base section fetching
187   //---------------------------------------
188
189     int32_t cm_base_left_ic, cm_base_right_ic;
190     int32_t cm_base_left_rc, cm_base_right_rc;
191     int32_t wit_init, wit_direction, wit_final;
192     uint16_t cm_base_width = 0;
193
194     if (right_image)
195     {
196       cm_base_left_ic   = IMAGE_XCOORDINATE(WG_RIGHTEST_GLOBAL_ID)+1;
197       cm_base_right_ic  = min((int32_t) IMAGE_XCOORDINATE(WG_RIGHTEST_GLOBAL_ID) + params->
            disp_max, (int32_t) params->image_width - WINDOW_SIZE - U_STEP);
198       cm_base_width   = max((cm_base_right_ic - cm_base_left_ic + 1), 0);
199
200       cm_base_left_rc = cm_width - cm_base_width;
201       cm_base_right_rc  = cm_width - 1;
202
203       wit_init    = WG_WARPS - 1;
204       wit_direction = -1;
205       wit_final   = 0;
206     }
207     else
208     {
209       cm_base_left_ic   = max((int32_t) IMAGE_XCOORDINATE(WG_LEFTEST_GLOBAL_ID) - params->
            disp_max, (int32_t) WINDOW_SIZE + U_STEP);
210       cm_base_right_ic  = IMAGE_XCOORDINATE(WG_LEFTEST_GLOBAL_ID) - 1;
```

```
211
212         cm_base_width = max((cm_base_right_ic - cm_base_left_ic + 1), 0);
213
214         cm_base_left_rc   = 0;
215         cm_base_right_rc  = max((int32_t) cm_base_width - 1, 0);
216
217         wit_init       = 0;
218         wit_direction = 1;
219         wit_final      = WG_WARPS - 1;
220       }
221
222       // Order memory loads
223       read_mem_fence(CLK_GLOBAL_MEM_FENCE);
224       uint16_t cm_base_fetching_loops = convert_int(floor( SNAP(cm_base_width, get_local_size
              (0)) / (float) get_local_size(0)));
225
226       // fetch base to cache matrix, performed by the whole workgroup
227       if (cm_base_width > 0)
228         for (int i = 0; i < cm_base_fetching_loops; i++)
229           if ((wg_size * i + wi_lid) < cm_base_width)
230           {
231             CM_SET_FROM_DESC(cm_base_left_rc + THREAD_BASE_ASIGNEE(i),0, RI_DESC,
                    cm_base_left_ic + THREAD_BASE_ASIGNEE(i), IMAGE_YCOORDINATE(get_global_id(1))
                    - V_STEP);
232             CM_SET_FROM_DESC(cm_base_left_rc+THREAD_BASE_ASIGNEE(i),1,RI_DESC, cm_base_left_ic
                    +THREAD_BASE_ASIGNEE(i), IMAGE_YCOORDINATE(get_global_id(1)) + V_STEP);
233           }
234     //}
235
236     //{ Synchronize all threads
237
238       read_mem_fence(CLK_GLOBAL_MEM_FENCE);
239       barrier(CLK_LOCAL_MEM_FENCE);
240
241     //}
242
243     //{ Warp Iteration (Progressive fetches to Cache Matrix)
244     //------------------------------------------------------------
245
246       uint16_t wit_index = wit_init;
247
248       while ( ( right_image && wit_index >= wit_final) || (!right_image && wit_index <=
              wit_final))
249       {
250         if ( wit_index == WARP_ID )
251         {
252           //{ Fetch own section to the Cache Matrix
253
254             int32_t cm_wit_left_ic, cm_wit_right_ic;
255             int32_t cm_wit_left_rc, cm_wit_right_rc;
256
257             if (right_image)
258             {
259               // Always take into account the reasons for the signs
```

```
260             cm_wit_left_ic  = max((int32_t) IMAGE_XCOORDINATE(WG_RIGHTEST_GLOBAL_ID - (
                    WARP_SIZE * (WARP_ID + 1)) + 1) - U_STEP, (int32_t) WINDOW_SIZE + U_STEP);
261             cm_wit_right_ic   = IMAGE_XCOORDINATE(WG_RIGHTEST_GLOBAL_ID - (WARP_SIZE * (
                    WARP_ID)));
262
263             cm_wit_left_rc    = cm_base_left_rc - (WI_CM_OWN_WIDTH * WARP_ID) -
                    WI_CM_OWN_WIDTH;
264             cm_wit_right_rc   = cm_base_left_rc - (WI_CM_OWN_WIDTH * WARP_ID) - 1;
265             cm_wit_left_rc    = cm_wit_left_rc  % cm_width;
266             cm_wit_right_rc   = cm_wit_right_rc % cm_width;
267           }
268           else
269           {
270             cm_wit_left_ic  = IMAGE_XCOORDINATE(WG_LEFTEST_GLOBAL_ID + (WARP_SIZE * WARP_ID)
                      + 1);
271             cm_wit_right_ic   = min((int32_t) IMAGE_XCOORDINATE(WG_LEFTEST_GLOBAL_ID + (
                    WARP_SIZE * (WARP_ID + 1)) - 1) + U_STEP, (int32_t) params->image_width -
                    WINDOW_SIZE - U_STEP);
272
273             cm_wit_left_rc    = cm_base_right_rc + (WI_CM_OWN_WIDTH * WARP_ID) + 1;
274             cm_wit_right_rc   = cm_base_right_rc + (WI_CM_OWN_WIDTH * WARP_ID) +
                    WI_CM_OWN_WIDTH - 1;
275             cm_wit_left_rc    = cm_wit_left_rc  % cm_width;
276             cm_wit_right_rc   = cm_wit_right_rc % cm_width;
277           }
278
279           ///@todo This wont work for the right image.
280
281           int can_stepsize = params->candidate_stepsize;
282
283           #define cm_x(offset)  ((cm_base_width + can_stepsize * wi_lid + (offset)) %
                  cm_width)
284           for (int i = 0; i < can_stepsize; i++)
285           {
286             #define cm_y 0
287             vstore16( vload16(GET_IMAGE_IOFFSET(u + i, v - V_STEP), RI_DESC), cm_y *
                    cm_width + cm_x(i), cache_matrix);
288
289             #define cm_y 1
290             vstore16( vload16(GET_IMAGE_IOFFSET(u + i, v + V_STEP), RI_DESC), cm_y *
                    cm_width + cm_x(i),  cache_matrix);
291           }
292
293         //}
294
295         //{ If needed, fetch own section from ref and compute disparity
296           bool compute_disparity = true;
297           int i;
298           //{ Check condition
299           //
300           //   Local min & max disparities (for the thread's pixel)
301           //   Disparities are always relative to the pixels.
302           //
303           // get valid disparity range
304           int32_t local_disp_min_valid = max(params->disp_min,0);
```

```
305             int32_t local_disp_max_valid = params->disp_max;
306
307             // u, WINDOW_SIZE and U_STEP are present because the use we give to it later
308             //   if (right_image)
309             //     u_warp = u+d;
310             //   ==>  u_warp = params->image_width - WINDOW_SIZE - U_STEP;
311             //   else
312             //     u_warp = u-d;
313             //   ==> u_warp = WINDOW_SIZE+U_STEP;
314
315             // if the target picture is the right one then, I have to look for
316
317             // pixels same coordinates a little to the right on the reference image
318             if (right_image)
319               local_disp_max_valid = min(local_disp_max_valid, params->image_width-u-
                      WINDOW_SIZE-U_STEP);
320             else
321               local_disp_max_valid = min(local_disp_max_valid,u-WINDOW_SIZE-U_STEP);
322
323             // assume, that we can compute at least 10 disparities for this pixel
324             if (local_disp_max_valid-local_disp_min_valid<10)
325               compute_disparity = false;
326             // we require at least some texture
327             int32_t sum;
328             uchar16 xmm5;
329             // This is the same as: xmm5 = vload16(0, ADD_DESC_OFFSET(TI_DESC, u, v));
330             xmm5 = vload16(GET_IMAGE_IOFFSET(u, v),TI_DESC);
331             xmm5 = abs_diff(xmm5, (uchar16)(128));
332             sum  = addAllTerms(xmm5);
333
334             if (sum < params->support_texture)
335               compute_disparity = false;
336
337             //}
338
339             if (compute_disparity)
340             {
341               //{ Fetch own region from the Target image
342
343               vstore16( vload16(GET_IMAGE_IOFFSET(u - U_STEP, v - V_STEP), TI_DESC), (0<<5) +
                      wi_lid, target_cache);
344               vstore16( vload16(GET_IMAGE_IOFFSET(u + U_STEP, v - V_STEP), TI_DESC), (1<<5) +
                      wi_lid, target_cache);
345               vstore16( vload16(GET_IMAGE_IOFFSET(u - U_STEP, v + V_STEP), TI_DESC), (2<<5) +
                      wi_lid, target_cache);
346               vstore16( vload16(GET_IMAGE_IOFFSET(u + U_STEP, v + V_STEP), TI_DESC), (3<<5) +
                      wi_lid, target_cache);
347
348               //}
349
350               //{ Compute disparity
351
352               // declare match energy for each disparity
353               int32_t u_warp;
354               // best match
```

```
355             int16_t min_1_E = 32767;
356             int16_t min_1_d = -1;
357             int16_t min_2_E = 32767;
358             int16_t min_2_d = -1;
359
360             uint16_t cm_x;
361
362             // for all disparities do
363             for (int16_t d=local_disp_min_valid; d<=local_disp_max_valid; d++)
364             {
365
366               // warp u coordinate
367               if (right_image)
368                 u_warp = (cm_wit_left_rc + THREAD_OWN_ASIGNEE) + d;
369               else
370                 u_warp = max((int) cm_base_width, 0) + params->candidate_stepsize * wi_lid -
                        d;
371
372               sum=0;
373               //{ For all terms abs_diff(target, ref) - Upper left
374
375                 #define cm_y 0
376                 #define cm_x ((cm_base_width + can_stepsize * wi_lid + (-U_STEP -d)) %
                        cm_width)
377                 #define tc_y 0
378                 #define tc_x wi_lid
379
380                 i = 0;
381                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
382                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
383                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
384                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
385
386                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
387                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
388                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
389                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
390
391                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
392                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
393                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
394                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
395
```

```
396                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
397                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
398                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
399                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
400
401               //}
402
403               //{ For all terms abs_diff(target, ref) - Upper right
404
405                 #define cm_y 0
406                 #define cm_x ((cm_base_width + can_stepsize * wi_lid + (+U_STEP -d)) %
                        cm_width)
407                 #define tc_y 1
408                 #define tc_x wi_lid
409
410                 i = 0;
411                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
412                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
413                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
414                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
415
416                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
417                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
418                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
419                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
420
421                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
422                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
423                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
424                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
425
426                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
427                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
428                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
429                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
```

```
430
431             //}
432
433             //{ For all terms abs_diff(target, ref) - Botoom left
434
435               #define cm_y 1
436               #define cm_x ((cm_base_width + can_stepsize * wi_lid + (-U_STEP-d)) %
                      cm_width)
437               #define tc_y 2
438               #define tc_x wi_lid
439
440               i = 0;
441               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
442               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
443               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
444               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
445
446               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
447               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
448               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
449               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
450
451               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
452               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
453               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
454               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
455
456               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
457               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
458               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
459               sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i   ] -
                      cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
460
461             //}
462
463             //{ For all terms abs_diff(target, ref) - Botoom right
464
465               #define cm_y 1
466               #define cm_x ((cm_base_width + can_stepsize * wi_lid + (+U_STEP-d)) %
                      cm_width)
```

```
467                 #define tc_y 3
468                 #define tc_x wi_lid
469
470                 i = 0;
471                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
472                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
473                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
474                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
475
476                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
477                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
478                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
479                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
480
481                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
482                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
483                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
484                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
485
486                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
487                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
488                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
489                 sum += abs( target_cache[( ((tc_y<<(5)) + tc_x)   << 4 ) + i  ] -
                        cache_matrix[( (cm_y*cm_width + cm_x) << 4 ) + i++]);
490
491               //}
492
493               // best + second best match (And-Or Selector)
494
495               bool c_value;
496               uint16_t c_mask;
497
498               c_value = sum<min_1_E;
499               c_mask = -c_value;
500
501               min_1_E  = (int16_t) ((sum & c_mask) | (min_1_E & ~c_mask));
502               min_1_d  = (int16_t) ((d   & c_mask) | (min_1_d & ~c_mask));
503
504               c_mask   = ~c_mask;
505               c_value  = (sum<min_2_E);
```

```
506                 c_mask  &= (-c_value);
507
508                 min_2_E = (int16_t) ((sum & c_mask) | (min_2_E & ~c_mask));
509                 min_2_d = (int16_t) ((d   & c_mask) | (min_2_d & ~c_mask));
510
511               }
512
513               // check if best and second best match are available and if matching
514               // ratio is sufficient
515               if (min_1_d>=0 && min_2_d>=0 && isless((float) min_1_E,(float) (params->
                      support_threshold)*min_2_E))
516                 return min_1_d;
517               else
518                 return -1;
519               //}
520             }
521           //}
522         }
523         // Wait for barrier
524         barrier(CLK_GLOBAL_MEM_FENCE);
525         // Increment warp loop index
526         wit_index = wit_index + wit_direction;
527       }
528     //}
529
530     return -1;
531   }
532
533   ///@warning Work group size MUST be {WARP_SIZE * x, 1} for this kernel to work.
534   __kernel void ComputeSupportMatches(__global uint16_t* d_D_can, __global __const uint8_t *
          d_I_desc1, __global __const uint8_t * d_I_desc2, __global __const sm_params_t * params,
          __local uint8_t* cache_matrix, __local uint8_t* target_cache)
535   {
536     int32_t u_can = get_global_id(0);
537     int32_t v_can = get_global_id(1);
538
539     if (u_can > params->D_can_width || v_can > params->D_can_height)
540       return;
541
542     // disparity compute variables
543     int16_t d, d2, result;
544
545     // initialize disparity to invalid
546     result = -1;
547
548     // find forwards
549     d = computeMatchingDisparityForCandidate(d_I_desc1, d_I_desc2, params, u_can, 0, v_can,
            false, cache_matrix,target_cache);
550
551     if (d>=0)   {
552       // find backwards
553       d2 = computeMatchingDisparityForCandidate(d_I_desc1, d_I_desc2, params, u_can, -d, v_can
              , true, cache_matrix, target_cache);
554
555       if (d2>=0 && abs(d-d2)<=params->lr_threshold)
```

```
556             result = d;
557         }
558         d_D_can[GET_OFFSET(u_can,v_can,params->D_can_width)] = result;
559     }
```

Bibliografía / Referencias

1. "Decentering distortion of lenses.". Photogrammetric Engineering. 7: 444–462, Brown DC (1966).

2. "Visión por computador: imágenes digitales y aplicaciones" / Gonzalo Pajares Martinsanz, Jesús Manuel, Paracuellos del Jarama (Madrid), 2008.

3. D. Scharstein and R. Szeliski. "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms". International Journal of Computer Vision, 47(1/2/3):7-42, April-June 2002. Microsoft Research Technical Report MSR-TR-2001-81, November 2001.

4. "Implementing FPGA Design with the OpenCL Standard", Altera, Nov 2011
http://www.altera.com/literature/wp/wp-01173-opencl.pdf

5. The International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, Jul 2012
http://ersaconf.org/ersa-news/

6. AnnieWay
http://www.mrt.kit.edu/annieway/team.html

7. KITTI Benchmark
http://www.cvlibs.net/datasets/kitti/

8. NASA's Mars Exploration Rover
http://www2.ece.ohio-state.edu/ion/documents/IEEE_aero.pdf

9. Advances in Theory and Applications of Stereo Vision Edited by Asim Bhatti, InTech, 08/06/2011

10. B. Delaunay: Sur la sphère vide, Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk, 7:793–800, 1934

11. OpenCL™ http://www.khronos.org/opencl/

12. Intel & OpenCL http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/

13. AMD & OpenCL http://developer.amd.com/zones/OpenCLZone/Pages/default.aspx

14. NVIDIA & OpenCL http://www.nvidia.es/object/cuda_opencl_new_es.html

15. ARM & OpenCL http://blogs.arm.com/multimedia/215-gpu-computing-advance-with-new-opencl-api/

16. OpenCL Programming Guide for the CUDA Architecture

17. NVIDIA CUDA Architecture Overview - http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf

18. ELAS - ACCV'10 Proceedings of the 10th Asian conference on Computer vision - Volume Part I, Pages 25-38, Springer-Verlag Berlin, Heidelberg ©2011

19. NVIDIA OpenCL Best Practices Guide

20. Stereo Correspondence Estimation based on Wavelets and Multiwavelets Analysis Asim Bhatti and Saeid Nahavandi, Intelligent Systems Research Lab., Deakin University, Australia, 2008 http://cdn.intechopen.com/pdfs/5764/InTech-Stereo_correspondence_estimation_based_on_wavelets_and_multiwavelets_analysis.pdf

21. Continuous Markov Random Fields for Robust Stereo Estimation Koichiro Yamaguchi Tamir Hazan, David McAllester, Raquel Urtasun, Toyota Technological Institute at Chicago, April 9, 2012
http://arxiv.org/pdf/1204.1393v1.pdf

22. Heterogeneous Computing with OpenCL, Gaster & Howes & Kaeli & Schaa, 1st Edition, 07 Oct 2011, pag. 15.

23. OpenCL Optimizations, Dr. Gernot Ziegler, Developer Technology (Compute) CINECA, Bologna/Italy| Oct 12th, 2011
http://corsi.cineca.it/courses/scuolaAvanzata/Ziegler-OpenCL/Part2_OpenCL_Optimization.pdf

24. FPGA-Based Multimodal Embedded Sensor System Integrating Low- and Mid-Level Vision Guillermo Botella, José Antonio Martín H., Matilde Santos y Uwe Meyer-Baese - http://www.mdpi.com/1424-8220/11/8/8164

25. CSC/ECE 506 Spring 2011/ch2a mc
http://pg-server.csc.ncsu.edu/mediawiki/index.php/CSC/ECE_506_Spring_2011/ch2a_mc#Introduction

26. http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-AMD-APP-SDK.aspx

27. OpenCL Fundamentals, David W. Gohara, Ph.D. Wednesday, August 26, 2009

28. Middlebury Stereo http://vision.middlebury.edu/stereo/

29. SSE **http://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms/**

Índice de figuras

Índice de cuadros