Procesamiento de imágenes en GPU – Organización del Computador II

Pazos Méndez, Nicolás Javier y Reyes Mesarra, Darío René

Resumen— **Cuatro algoritmos de procesamiento de imágenes implementados en *OpenCL*, para correr en *GPU*. Se discuten los algoritmos, sus implementaciones y las diferencias de *performance* frente a sus versiones en *C*.**

Keywords— **GPU, OpenCL, image processing, canny, hough, inpainting, lucas-kanade**

I. Introducción

Como trabajo final para la materia "Organización del Computador II", decidimos implementar cuatro filtros de imágenes en *OpenCL*, con el objetivo de familiarizarnos y comprender la programación y arquitectura de placas gráficas, entender su potencial de cómputo y así mismo las complicaciones del cambio de paradigma al trabajar de forma paralela.

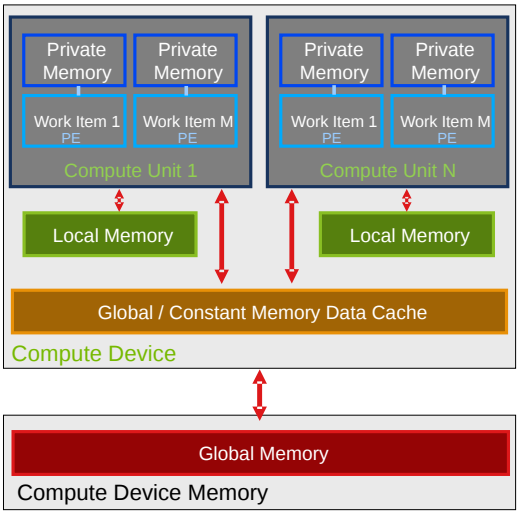


Fig. 1: Jerarquía de memorias en GPU, según OpenCL [1]

II. GPUs y OpenCL

Nos parece importante discutir brevemente algunas propiedades de las *GPUs*, e introducir la nomenclatura que utilizaremos en el resto del informe.

Las placas de video, por naturaleza, trabajan con concurrencia. Pueden llegar a tener decenas de miles de *threads* corriendo en simultáneo. Estos *threads* están dentro de *cores* (núcleos), que pueden llegar a ser unos miles. En *OpenCL*, a un *thread* particular se le pone el nombre de *work-item*.

Los núcleos están agrupados en lo que *OpenCL* llama *work-groups*, que comparten entre sí una memoria llamada *local memory*, y que además están garantizados de poder sincronizarse entre sí (no hay ninguna primitiva de sincronización para *work-items* en *work-groups* diferentes).

Las GPUs tienen distintos niveles de memoria con diferentes velocidades, como se ve en el esquema 1. La primera es la *global memory*, que es accesible por todos los *workers*. La siguiente es la *local memory*, que es más rápida ($\sim \times 15$ usualmente), pero tiene la restricción de que solo es compartida dentro de un mismo *work-group*. Luego, cada *work-item* tiene su propia *private memory*, que a su vez es mucho más rápida que las memorias previamente mencionadas, pero no es compartida en absoluto.

La compleja distribución de memorias en las placas gráficas se debe a que las lecturas y escrituras son operaciones sumamente costosas y no deseadas. Los procesadores de las *GPUs* son muy rápidos para operaciones aritméticas (sobre todo de punto flotante), y se deben priorizar por sobre accesos a memoria. Más costosa aún es la transferencia de datos desde el *host* (el CPU) a la placa gráfica, tanto de lectura como escritura. Veremos más adelante, que muchos de los filtros en *OpenCL* están pensados para minimizar todos estos movimientos. Las *GPUs* también disponen de distintas cachés, pero generalmente su uso se da de forma automática. En *OpenCL* se le llama *kernel* al programa que corre en un *thread*. Se dispone de una cola de ejecución en la que puede encolarse un *kernel* asociado a una cantidad de *workers*. Los *kernels* en la cola de ejecución se ejecutan secuencialmente; para cada uno de ellos, se lanzan tantos *threads* a correr el programa como se haya indicado. Todos corren el mismo código, y es deseable que su ejecución sea idéntica, pues las *GPUs* mejoran su performance si los *instruction pointers* de un grupo son iguales.

Algunas otras particularidades de las GPUs que no mencionamos en esta sección surgirán en el análisis de los filtros, y discutiremos cómo las aprovechamos o cómo nos perjudicaron en la implementción.

III. Los filtros

En esta sección introduciremos los cuatro algoritmos con los que trabajamos. Vamos a discutir qué consideraciones tu-

vimos al implementar los algoritmos para las GPUs, y cómo es que aprovechamos su arquitectura, además de presentar la performance obtenida comparando con las version implementadas solo en *C*. En la bibliografía a la que recurrimos, que está incluida en el informe, se encuentra la teoría detrás de los algoritmos con profundidad. En cada sección vamos a dar una idea los mismos como contexto para ver qué es lo que logramos optimizar corriéndolos en placas de video.

Todas las mediciones mostradas en las tablas se realizaron sobre una image de 400×400, con un `Intel i7 5500U` [2] como *host* y una GPU `NVIDIA GeForce 940m` [3] como *device*. Se tomaron varias mediciones para obtener valores representativos.

A. *Detección de bordes*

Implementamos el ampliamente conocido algoritmo de detección de bordes propuesto por Canny en 1968 [4]. En pocas palabras, el método propone postprocesar el resultado de una convolución de gradiente, evitando bordes dobles y cortados.

Las convoluciones [5] son operaciones fácilmente optimizables en GPU, pues están compuestas por una operación que se repite por cada punto de la señal (en este caso, la imagen):

$$I(x,y) = \sum_{k_1=-K}^{K} \sum_{k_2=-K}^{K} I(x-k_1, y-k_2) \times kernel(k_1,k_2)$$

Al ser esta la operación para cada pixel (x,y), programar una convolución con *OpenCL* implica implementar esta operación para un pixel en particular, y lanzar tantos *workers* como píxeles tenga la imagen en cuestión, uno por pixel. Sin bien hacer eso ya resulta en una convolución mucho más performante que la obtenida, por ejemplo, en C, pueden tenerse en cuenta algunas cosas más.

Una convolución tiene muchos accesos a memoria repetidos: para cada pixel se consultan todos sus vecinos en un radio K. Un pixel en particular será accedido aproximadamente $4K^2$ veces. Si bien optimizar esto no cambia el orden del tiempo de ejecución de la convolución, vale la pena atacarlo. Para hacerlo, en la implementación de *OpenCL* de las convoluciones nos manejamos con un tipo especial de *buffers* que disponen las placas de video: las texturas (llamadas *images* en *OpenCL*). Las texturas, entre otras propiedades, disponen de cachés de espacialidad en una, dos y tres dimensiones, dependiendo del tipo de textura. Esto quiere decir que, al acceder al pixel (x,y), se traen a la caché píxeles cercanos al mismo; para la convolución, los píxeles que vamos a necesitar. Usando texturas, aumentamos los *hits* a la caché y evitamos más accesos a memoria global.

Un desafío que nos encontramos en la implementación del algoritmo de Canny fue el proceso de *hysteresis* [4], que consta de un ciclo cuya condición de corte es la preservación de la imagen; en otras palabras, se itera hasta que no se realizaron cambios en la imagen. Esta lógica no es tan fácil de traducir a la programación concurrente. En la GPU, el cuerpo del ciclo que itera por la imagen se corre una única vez para cada *worker*, y deberíamos poder determinar si cualquiera de ellos efectuó un cambio en la imagen.

Nuestra solución fue disponer de un booleano en memoria global para que los *workers* fijen en *true* si cambiaron la imagen. Para hacer eso, procuramos no afectar la performance en forma significativa: si todos los N *workers* efectuaran cambios, tendríamos N accesos idénticos a memoria global, todos seteando nuestro booleano en *true*. Para minimizar estos accesos, nos apoyamos nuevamente en la caché:

```
if (cambios == 0) cambios = 1;
```

Con este previo checkeo sobre el booleano, estamos evitando escrituras innecesarias en el caso de que ya fuese *true*. Más aun, la lectura del condicional se aprovecha de la caché, y evitamos tantas lecturas a memoria global.

Con todas estas y algunas otras consideraciones en la implementación para *OpenCL*, obtuvimos la *performance* que expone la tabla 1.

Canny en *C*	`12,7 FPS`
Canny en *OpenCL*	`197,2 FPS`
	`16 X`

Table 1: comparación de *performance* en cuadros por segundo (FPS) para el algoritmo de Canny en sus dos implementaciones

El filtro en GPU es considerablemente más rápido, llegando a frecuencias que algunas aplicaciones considerarían "tiempo real". Sin embargo, esperábamos frecuencias un tanto mayores. Luego de indagar un poco, concluímos que el problema estaba en la *hysteresis*, cuya lógica es computacionalmente intensa y, estructuralmente difícil de traducir a la

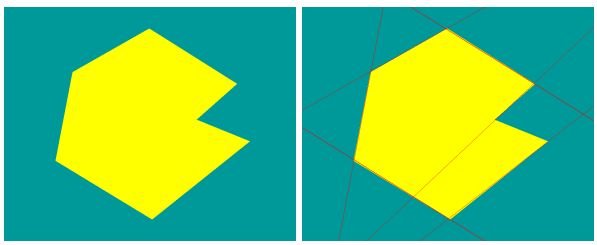
programación concurrente de GPUs: normalmente resulta en alrededor de 50 iteraciones antes de que no haya cambios, lo que en *OpenCL* se traduce en 50 *kernels* encolados para ejecutar. Realizamos una nueva medición quitando la *hysteresis* en ambas implementaciones (que, por cierto, no impacta en forma considerable en la salida del algoritmo), y obtuvimos los de la tabla 2.

Canny en *C*	`20,9 FPS`
Canny en *OpenCL*	`414,9 FPS`
	`20 X`

Table 2: comparación de *performance* en cuadros por segundo (FPS) para el algoritmo de Canny en sus dos implementaciones

En estas nuevas mediciones se ve cómo impacta esa parte del algoritmo en la *performance*, reduciendo a la mitad los cuadros por segundo. Algunas aproximaciones del algoritmo de Canny, saltean o simplifican la *hysteresis*. Nótese además cómo esto evidencia que esa lógica tiene una estructura que no aprovecha tan bien la potencia de cómputo de una GPU, ya que la diferencia *OpenCL–C* aquí resulta mayor.

B. Detección de líneas



El algoritmo de detección de líneas [6] que implementamos está basado en la transformada de Hough [7]. Su idea básica es tomar una imagen binaria que represente bordes, y de todas las posibles líneas que pueden dibujarse sobre la imagen, determinar cuáles tienen la mayor cantidad de píxeles presentes en los bordes. Para obtener los bordes de una imagen, utilizamos el algoritmo de Canny que implementamos previamente.

La transformada de Hough aplicada en este algoritmo involucra un intento de cómputo de operaciones trigonométricas para cada pixel de la imagen, utilizando únicamente información local. Esto, a primera vista, lo hace muy apto para implementar en GPU: el cómputo es paralelizable por pixels, y las operaciones que involucran son muy rápidas.

Cuando comenzamos por implementar el algoritmo en *C*, notamos que el cálculo de estas operaciones resultaba muy costoso, pues se encontraban dentro de varios ciclos anidados; para mejorar la performance, probamos crear un arreglo de operaciones trigonométricas precomputadas sobre un espacio discretizado. Esto aumentó notablemente la rapidez del filtro.

Distinto fue el caso de *OpenCL*. Como sabemos que son operaciones para las cuales están diseñadas y los accesos a memoria son costosos, resultaba más eficiente que cada *worker* realizara las operaciones trigonométricas que necesitara en vez de precomputarlas y pagar una lectura (por supuesto, esto sumado a que lo que eran iteraciones en *C*, ahora son *threads*). Probamos almacenar este precómputo en memoria local, pero aun así resultaba más rápido computarlo *on demand*.

En un momento dado del algoritmo, se debe crear un contador matricial cuyas celdas se incrementan de forma individual dependiendo de ciertas propiedades de cada pixel de la imagen. Una vez construido este contador nos interesa encontrar las celdas cuyos contadores son mayores. Para hacer esto (sin entrar en detalles algorítmicos), nos era necesario conocer el valor máximo del contador. Todo este procedimiento de la formación y análisis del contador es fácil e intuitivo de implementar en un lenguaje como *C*, de forma secuencial. Cuando introducimos el paralelismo, sin embargo, debimos tener en cuenta cuestiones de concurrencia y performance.

En primer lugar, en la construcción del contador, debimos ser cuidadosos al incrementar los valores y manejar la concurrencia de alguna forma: si fuese una simple lectura, adición y escritura, dos *workers* podrían intentar hacerlo en simultáneo para la misma dirección, y se perdería un conteo. *OpenCL* ofrece varios recursos para tratar estos problemas; entre otras cosas, existe un conjunto de operaciones atómicas de todo tipo, que contiene la siguiente función:

```
int atomic_inc (__global int *p )
```

Como sugiere su nombre, esta operación suma 1 al entero almacenado en `p`, lo que nos solucionó el problema de la construcción del contador. Nótese que esta operación conlleva un acceso a memoria global, que sabemos es costoso. Consideramos evitar o minimizar estos accesos, pero no parecía posible por la naturaleza del algoritmo; entre otras particularidades, no resultan fácilmente predecibles las celdas del contador que incrementará un *worker* particular, y menos aún la relación de ellas con las las de sus *workers* vecinos, si se quisiera hacer algún tipo de reducción utilizando memoria local.

La determinación del máximo valor del contador puede hacerse de varias maneras. Una forma sería programar un *kernel* que implemente una reducción: como la operación `max` es conmutativa y asociativa, puede reducirse el tamaño de la matriz iterativamente, tomando máximos parciales. Esta opción

tiene el *overhead* que trae cada nuevo *kernel* a ejecutar: establecer sus parámetros (tráfico *host–device*), encolarlo a la cola de ejecución, etc. En nuestro caso, tenemos la posibilidad de calcular el máximo en el mismo *kernel* en el que construímos el contador. Simplemente utilizando la operación atómica de *OpenCL* `atomic_max`, cada *worker* puede actualizar el máximo (único para todos) con el mayor valor visto entre todas las celdas que incrementó. Esto no agrega muchos más accesos a memoria global (uno por *worker*), considerando los realizados en la construcción del contador.

Para este filtro obtuvimos los siguientes resultados:

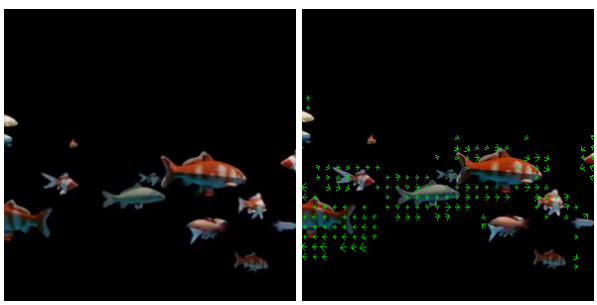
Hough en *C*	`12,9 FPS`
Hough en *OpenCL*	`153,7 FPS`
	`12 X`

Table 3: comparación de *performance* en cuadros por segundo (FPS) para el algoritmo de detección de líneas en sus dos implementaciones

Como era de esperarse, la velocidad de este filtro es menor a la de Canny, pues incluye una llamada al mismo. La diferencia obtenida con *OpenCL* no es tan grande como la obtenida en el filtro anterior. Esto, probablemente se deba a los caóticos accesos a memoria global que mencionamos que ocurren al construir el contador. No solo son numerosos, sino que no están relacionados entre *workers*: las GPUs pierden mucha *performance* cuando los accesos a memoria de sus *threads* no son cercanos ni siguen un patrón lineal aparente.

Cuando los accesos a memoria de *threads* cercanos son a direcciones de memoria cercanas y de forma ordenada (por ejemplo, el *worker* de `id = (x,y)` accede al pixel (x,y)), la velocidad aumenta considerablemente por varias razones. La razón principal es que la memoria suele traerse en bloques, por lo que puede aprovecharse un único *fetch* para abastecer varios *workers*. Otra razón es el mejor aprovechamiento de la caché: la caché local a un grupo de *workers* se aprovecha mucho mejor si sus accesos son cercanos. Además, algunas GPUs implementan predicción de accesos a memoria para adelantar las lecturas. A todo esto se lo conoce con el nombre de *coalescing memory access*.

C. Detección de movimiento



Este filtro consiste en determinar en cada frame de video adonde se están moviendo los objetos. El algoritmo que utilizamos fue el método de Lucas-Kanade, introducido por primera vez en un paper de 1981 [8]. Para la implementación nos basamos en un documento de Intel [9] que describe el algoritmo en detalle.

Dado dos frames consecutivos I y $I+1$, el core del algoritmo consiste en resolver la siguiente ecuación para cada punto (i, j) del video:

$$S * \mu = t$$

$$S = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \qquad t = \begin{bmatrix} \sum -I_x I_t \\ \sum -I_y I_t \end{bmatrix}$$

Donde las sumatorias están definidas sobre un área de tamaño fijo alrededor de (i, j) y ...

- I_x e I_y son los gradientes de I
- I_t es la diferencia entre I e $I+1$ (gradiente temporal)
- La incógnita μ es la velocidad a la que se mueve el pixel (i, j) del cuadro I

Para calcular los gradientes, convolucionamos con los kernels de Sobel, que son perfectamente paralelizables como vimos en Canny. Lo mismo ocurre con la diferencia de imágenes, la cual podemos calcular independientemente para todos los puntos del video. El algoritmo tiene sus complejidades, pero al final del dia lo que vamos a terminar haciendo es resolver la ecuación previamente descripta una y otra vez, con lo cual ganamos mucho pasándolo a GPU donde podemos paralelizar el trabajo de los píxeles.

Además de resolver la ecuación, el algoritmo requiere submuestrear los frames de entrada múltiples veces, para conseguir dichos frames en distintas resoluciones. Esto también involucra suavizar las imágenes con un filtro gaussiano antes de reducir sus tamaños para evitar que perdamos información. Esto implica que tuvimos que introducir varios buffers para guardar los resultados de estas operaciones. Algo importante que tuvimos que hacer es asegurarnos de que todo esto se hiciese completamente en el device, de lo contrario la penalidad de mover datos entre el device y el host sería contraproducente. Lo que hacemos es mover la imagen al device al principio, y generar el resto las imágenes y buffers en el device mismo. Recién cuando el algoritmo termina es que vuelven a interactuar el host y el device, para escribir el resultado del algoritmo (el optical flow) a memoria.

Los resultados que obtuvimos en las version de C y OpenCL pueden ver en la tabla 3 4.

Hay dos observaciones que podemos hacer a partir de estos resultados.

Optical Flow en *C*	1,01 FPS
Optical Flow en *OpenCL*	96,9 FPS
	96 X

Table 4: comparación de *performance* en cuadros por segundo (FPS) para el algoritmo de Optical Flow en sus dos implementaciones

En principio, los FPS del algoritmo en C son mucho menores a los de Canny y Hough. El trabajo extra es debidoa que el algoritmo itera muchas veces sobre si mismo para conseguir buenos resultados.

La segunda observación que podemos hacer es que la ganancia que tenemos en la versión en OpenCL con respecto a la versión en C, es mucho mayor que en los anteriores algoritmos. Hay varios motivos por los que esto puede ocurrir. En principio, tanto en Canny/Hough como en Lukas-Kanade hay mínimas lecturas/escrituras de host a device; pero como Lukas-Kanade itera, terminá computando más que los otros dos algoritmos. Esto implica que el ratio de trabajo sobre accesos a memoria es mayor en Lukas-Kanade, con lo cual termina siendo más beneficioso cuando lo pasamos a GPU. Otro motivo que encontramos es que el algoritmo lidia mayor que nada con floats, mientras que hay partes de Canny y Hough que utilizan aritmética entera; y las GPUs suelen estar optimizadas para operar con floats (por ejemplo, para las placas NVIDIA se recomienda en la Programming Guide de CUDA utilizar floats antes que enteros [10]).

D. Remoción de objetos (inpainting)

El objetivo de este filtro es eliminar partes indeseadas de una imagen. Toma como entrada una imagen enmascarada, y rellena inteligentemente la máscara de manera tal que no se note que alguna vez hubo algo. Para lograr esto, la idea es utilizar la información y los patrones que contiene la imagen.

Hay muchos algoritmos que resuelven este problema. El que elegimos es de un paper de 2004, de Criminisi et al. [11]. El algoritmo consiste en ir rellenando la máscara de a pequeños parches: elegimos un parche de la máscara y lo rellenamos con un parche en la imagen. Repetimos este procedimiento hasta que la máscara quede completamente rellena.

Para elegir qué parte de la máscara queremos rellenar, el paper propone maximizar cierta función sobre los píxeles de la máscara. Esta función requiere solo información local, con lo cual la surge una oportunidad de paralelización, haciendo que cada *work-item* se encargue de calcular la función para un pixel en particular.

Una vez que encontramos qué parte de la máscara queremos rellenar, debemos buscar un parche en la imagen con el cual rellenarlo. La situación es similar a la anterior: vamos a querer minimizar una función sobre todos los píxeles que no están en la máscara. Esta función también va a depender de información local, con lo cual podemos aplicar la misma estrategia que antes.

La gran ganancia que tenemos al pasar el algoritmo a GPU estaría en estas dos operaciones que podemos paralelizar para cada pixel.

El detalle que nos queda es como encontrar máximos y mínimo de esta función a través de la imagen. Una idea es utilizar *atomic_max* como hicimos en Hough. El problema es que este built-in es solo para enteros en *OpenCL 1.2*. Hay una forma de implementarla utilizando la única función atómica para floats *atomic_xchg*, pero no es trivial e impone una penalidad en performance.

La alternativa que consideramos es realizar las reducciones en el device sincronizando los *workers*. Esto es, que los threads vayan calculando el máximo de a pares, hasta que un solo thread termine quedándose con el máximo de toda la imagen. La restricción que tenemos es que solo podemos sincronizar *workers* de un mismo *work-group*. Esto implica dos cosas:

- Como los *workers* solo interactuan con los de su propio *work-group*, podemos aprovechar y utilizar la memoria local que tiene una latencia menor a la memoria global
- Como no podemos sincronizar threads de distintos work-groups, lo máximo que podremos reducir en el device es obteniendo el máximo de cada *work-group*. Deberíamos pasar esos resultados al host para que este calcule el máximo final.

Haciendo pruebas, obtuvimos que la reducción en el device, incluso utilizando memoria local, recién empezaba a superar a la reducción en el host para un número muy alto de píxeles. Por esa razón es que terminamos optando por hacer las reducciones enteramente en el host, pero en otros contextos, podría tener sentido optar por la alternativa en el device.

Otra cosa surgió al pasar el algoritmo de C a OpenCL es que en la versión en C habiamos dividido la elección de elegir que parte de la máscara rellenar en dos. Esto es porque en realidad no queremos buscar en toda la máscara, sino solo en el borde de la másacara. Por ello, primero calculábamos el borde (marcando cada pixel con un booleano), y después

buscábamos qué pixel en el borde queríamos rellenar. Nuestra primera versión del algoritmo en OpenCL tenía dos kernels que hacian ambas operaciones por separado. Lo que nos dimos cuenta es que si bien una operación dependía de la otra, podiamos calcular ambas para un solo pixel sin depender del resto de la imagen (salvo vecinos). Con lo cual, pudimos combinar ambos kernels, reduciendo el overhead de inicialización de threads. Esto tiene impacto debido a que ejecutamos nuevos kernels por cada parche que rellenamos de la máscara (para tener una referencia, en una máscara de 90*90 píxeles, se necesitan al menos 200 parches para rellenarla completamente, con lo cual terminamos inicializando muchos kernels).

La performance final que conseguimos fue:

Inpainting en *C*	0,01 FPS
Inpainting en *OpenCL*	0,79 FPS
	53 X

Table 5: comparación de *performance* en cuadros por segundo (FPS) para el algoritmo de Inpainting en sus dos implementaciones

Podemos ver que este es el filtro más lento de los cuatro. El motivo es que tenemos que procesar la imágen entera una y otra vez hasta rellenar la máscara, lo que resulta en muchas iteraciones. En este algoritmo también notamos un gran aumento en la performance al pasarlo a GPU. Para esto encontramos la misma explicación que en Lukas-Kanade, debido a la naturaleza iterativa del algoritmo, terminamos computando más que leyendo/escribiendo a memoria. Perdemos un poco de performance por acceder al host para realizar las reducciones.

IV. Notas finales

En el desarrollo de este trabajo final pudimos comprobar empíricamente el aumento de performance que podemos conseguir explotando el poder de las GPUs. Destacamos lo significativo del aumento, variando desde 12 hasta 96 veces más rápido. Más allá de los números, pudimos ver filtros lentos implementados de forma secuencial llegando a ser practicamente real-time aprovechando las placas de video de forma paralela.

Logramos entender un poco más algunas consideraciones a tener en cuenta al correr programas en la GPU. Cuestiones como ser cuidadosos al mover datos entre los distintos niveles de memoria, cuanto hacer en el host y cuanto en los kernels, y como compartir información entre las distintas partes de nuestros programas.

Para interactuar con las GPUs, tuvimos una buena experiencia con OpenCL, cuya portabilidad nos permitió utilizar los mismos fuentes para desarrollar y probar los algoritmos en dos máquinas muy distintas corriendo en diferentes sistemas operativos.

En conclusión, programar para placas de video puede ser accesible y proveer gran beneficio en contextos donde la performance sea crítica. Nos llevamos esta experiencia como una herramienta que potencia nuestra capacidad de afrontar futuros problemas, y como un aporte a nuestro conocimiento acerca de la capacidad que tienen los procesadores modernos.

References

1. Nvidia . OpenCL Optimization https://www.nvidia.com/content/gtc/documents/1068_gtc09.pdf.
2. Intel . Intel Core i7-5500U https://ark.intel.com/es/products/85214/Intel-Core-i7-5500U-Processor-4M-Cache-up-to-3-00-GHz-.
3. NVIDIA . GeForce 940m https://www.geforce.com/hardware/notebook-gpus/geforce-940m.
4. Canny J.. A Computational Approach To Edge Detection *IEEE Trans. Pattern Analysis and Machine Intelligence.* 1968.
5. Wikipedia . Núcleo de convolución https://es.wikipedia.org/wiki/N%C3%BAcleo_(procesamiento_digital_de_im%C3%A1genes).
6. Richard O. Duda Peter E. Hart. Use of the Hough transformation to detect lines and curves in pictures *Communications of the ACM.* 1972.
7. Hough P.V.C. Method and means for recognizing complex patterns *U.S. Patent 3,069,654.* 1962.
8. Lucas Bruce D., Kanade Takeo. An Iterative Image Registration Technique with an Application to Stereo Vision *Proceedings of Imaging Understanding Workshop.* 1981.
9. Bouguet Jean-Yves. Pyramidal Implementation of the Lucas Kanade Feature Tracker Description of the algorithm *Intel Corporation, Microprocessor Research Labs.* 2000.
10. NVIDIA Corporation . NVIDIA CUDA C Programming Guide http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.
11. A. Criminisi P. Perez, Toyama K.. Region Filling and Object Removal by Exemplar-Based Image Inpainting *IEEE Transactions on Image Processing, Vol. 13, No. 9.* 2004.