Ingeniare. Revista Chilena de Ingeniería
ISSN: 0718-3291
facing@uta.cl
Universidad de Tarapacá
Chile

Aracena-Pizarro, Diego; Daneri-Alvarado, Nicolás
Detección de puntos claves mediante SIFT paralelizado en GPU
Ingeniare. Revista Chilena de Ingeniería, vol. 21, núm. 3, diciembre, 2013, pp. 438-447
Universidad de Tarapacá
Arica, Chile

Detección de puntos claves mediante SIFT paralelizado en GPU

Keypoint detection through SIFT parallelization on GPU

Diego Aracena-Pizarro[1] Nicolás Daneri-Alvarado[1]

Recibido 25 de octubre de 2012, aceptado 12 de agosto de 2013
Received: October 25, 2012 Accepted: August 12, 2013

RESUMEN

Este trabajo presenta una optimización del método de detección de puntos SIFT (*Scale Invariant Feature Transform*), mediante su paralelización empleando una GPU (Unidad de Procesamiento Gráfico), aprovechando los múltiples núcleos de esta para dividir los procesos utilizando las API CUDA (Arquitectura Unificada de Dispositivos de Cómputo). El objetivo es acelerar el tiempo de cómputo, que es una variable crítica para todo el proceso de detección de puntos característicos. La estrategia utilizada se basa en dos premisas: el balance carga y la distribución de cálculo. Cada hilo realizará las operaciones necesarias para el cálculo de SIFT y así obtener los descriptores necesarios de acuerdo con un umbral apropiado. Dentro de SIFT se trabajó paralelizando el proceso de asignación de orientación, una de las etapas de SIFT consiste en la acumulación de todas las orientaciones en torno a una región de un punto clave, asignándose a cada pixel en la ventana un subproceso, centrada en la ubicación de un punto clave. Las pruebas se realizaron con un *notebook* con procesador Core 2 Duo 2.2Ghz, 3GB de RAM y una VGA GeForce 8600GT (32 núcleos) de 512MB. De los resultados obtenidos se observa que se logra un rendimiento en cuanto a velocidad del orden de 42,5 milisegundos en promedio, considerando todas las pruebas realizadas y todas las resoluciones trabajadas (320x240, 480x360, 640x480, 800x600, 1024x768, 1280x960), donde la paralelización de SIFT no muestra pérdidas significativas de puntos claves en comparación con la versión secuencial.

Palabras clave: SIFT, paralelismo, GPU, CUDA, puntos clave.

ABSTRACT

This paper presents an optimization method for detecting SIFT points (Scale-invariant feature transform), by using a GPU parallelization, taking advantage of multiple cores of it, to divide the processes using the API's CUDA. The goal is to accelerate the computation time, which is a critical variable for the entire process of key-point's detection. The strategy used is based on two assumptions, the load balance and distribution of calculation. Each thread will perform the operations required for calculating SIFT and obtain the necessary descriptors according to an appropriate threshold. Parallelizing the process of assignment of orientation, which consists of the accumulation of all the orientations concerning a region of a key-point, assigned to each pixel in the window a thread, centered on the location of a key point, was worked inside SIFT.
The tests were performed with a Notebook with Core 2 Duo 2.2GHz, 3GB of RAM and a GeForce 8600GT VGA (32 Cores) 512MB. The results obtained show that performance is achieved in terms of speed of the order 42.5 millisecond on average, considering all tests and all resolutions worked (320x240, 480x360, 640x480, 800x600, 1024x768, 1280x960), where the parallelization of SIFT, shows no significant loss of key points, compared to the sequential version.

Keywords: SIFT, Parallelism, GPU, CUDA, keypoints.

[1] Escuela Universitaria de Ingeniería Industrial, Informática y Sistemas. Área de Ingeniería en Computación e Informática. Universidad de Tarapacá. Arica, Chile. E-mail: daracena@uta.cl; ndaneria@gmail.com

INTRODUCCIÓN

Este trabajo corresponde a una sección de una implementación mayor: realizar una propuesta paralela para la reconstrucción de objetos y superficies 3D a partir de una secuencia de imágenes con vistas debidamente controladas mediante técnicas de visión, aprovechando la tecnología de alto desempeño en GPU y optimizando el proceso de detección de puntos característicos a través de SIFT y *Delaunay*, en este artículo se hablará sobre la parte de SIFT.

El algoritmo SIFT [3] es un método para extraer puntos característicos invariantes y distintivos de una imagen que pueden ser usados para mejorar la correspondencia entre dos vistas diferentes de un objeto o una escena.

Los puntos característicos obtenidos por SIFT son invariantes a escala y rotación de la imagen, y son mostradas para proporcionar un *matching* robusto a pesar de que exista un rango amplio de distorsiones afines, cambios en la vista 3D, adición de ruido a la escena y cambios en la iluminación.

Los puntos se distinguen claramente, en el sentido en que un único punto se puede corresponder correctamente con una alta probabilidad, contra una gran base de datos de puntos de muchas imágenes. Se puede dividir la estructura de SIFT en cuatro módulos:

- **Detección de escala-espacio (DEE)**: La primera fase de computación busca en todas las escalas y localizaciones de la imagen. Se implementa eficientemente al emplear la función de diferencias gaussianas para identificar los puntos de interés que son invariantes a escala y orientación.
- **La localización del "punto clave" (LK)**: En cada localización candidata se adecua un modelo cuadrático detallado para determinar la localización y la escala. Los puntos claves se seleccionan basándose en su estabilidad.
- **Asignación de orientación (AO)**: Una o más orientaciones se asignan a cada localización de los puntos claves, basándose en la dirección del gradiente de la imagen. Todas las futuras operaciones se ejecutan con los datos de la imagen que ha sido transformada de acuerdo con la orientación, escala y localización asignada para cada característica, de este modo se proporciona invariancia a estas transformaciones.
- **Descriptor del "punto clave" (DK)**: Los gradientes de la imagen son medidos en la escala seleccionada en la región alrededor de cada punto clave. Estos son transformados a una representación que permite, para niveles significativos, distorsión de la forma local y cambios en la iluminación.

Existen distintas propuestas en la literatura del proceso de detección de puntos característicos, las cuales presentan diferentes alternativas de solución basadas en técnicas de visión, en nuestro caso se abordarán las propuestas que utilizan SIFT y sugieren o implementan una estrategia paralela.

En Martín [4] el objetivo es la obtención de mapas de disparidad en un sistema de visión estéreo. Para ello se utiliza CUDA, que permite paralelizar operaciones para reducir en varios órdenes de magnitud los tiempos de cálculo. La propuesta apunta hacia implementaciones en sistemas tales como vehículos inteligentes, donde prima el tiempo de cálculo para la toma de decisiones en tiempo real, de modo que se pueda avisar al conductor o al sistema inteligente que conduce el vehículo.

En Zhang [5] se propone dos algoritmos paralelos para SIFT por GPU (CUDA) y por CPU (OpenMP). El resultado muestra una mejor aplicación de SIFT paralelo en GPU que en procesadores quad-core. También se lleva a cabo una escalabilidad y rendimiento de la memoria sobre un sistema de 8 núcleos en CPU y en un chip de 32 núcleos para GPU. El análisis ayuda a identificar las posibles causas de los cuellos de botella, y se sugieren vías para la mejora de la escalabilidad.

En Geys [6] se presenta un algoritmo para la eficiencia del cálculo de profundidades y síntesis de vista. El objetivo principal es la generación en línea de interpolaciones realistas de vistas en una escena dinámica. Las entradas son videos procedentes de dos o más cámaras calibrados. La eficiencia se logra mediante el uso paralelo de la CPU y la GPU en una aplicación multihilo. Las imágenes de entrada son proyectados en un plano de barrido a través de un espacio 3D, acelerando las transformaciones a través de GPU. Se observó una correlación de medida calculada de manera simultánea para

todos los pixeles en el plano y se compara en las posiciones diferentes del plano. Finalmente se aplica un algoritmo de min-cut/max-flow implementado en la CPU, para una optimización global.

La propuesta de Wu [1] se adopta como base de este trabajo, en conjunto con la implementación en Sinha [7], que describe la implementación del algoritmo KLT[8] para *tracking* y SIFT implementado por GPU mostrando la conveniencia de utilización en sistemas para su análisis en tiempo real, liberando la CPU para otras tareas de cálculo.

El trabajo se centra en el balance de carga en GPU, ya que es uno de los factores más importantes que influyen en la escalabilidad del rendimiento en una aplicación paralela. A cada punto clave se asigna una orientación y genera un descriptor, sólo después de detectar puntos claves en una escala. En este caso pueden encontrarse pocos puntos claves para el paso siguiente, quedando poco para las iteraciones siguientes de octavas y escalas. El desequilibrio de carga ocurre en los pasos de "Asignación de Orientación" y "Generar Descriptor" (el módulo AO-DK) para cada punto clave. Por otra parte, como la imagen se reduce en la muestra, el número de puntos claves detectados en cada escala disminuye.

PARALELIZACIÓN MEDIANTE CUDA

El algoritmo con que se trabajó tiene un alto componente de procesos que se ejecutan de manera iterativa e independiente del volumen de datos, es decir, es un trabajo *pixel* a *pixel.* Esta propiedad permite modificar los procesos con facilidad, para poder paralelizarlos por medio de GPU, por dos razones:

1. Tomar ventaja de la gran cantidad de unidades de procesamientos que posee una tarjeta gráfica, en comparación a paralelizar solo con CPU.
2. Aprovechar de investigar una tecnología novedosa en la actualidad y ver el potencial que se puede lograr con ello.

CUDA es una extensión de C que permite aprovechar la capacidad de procesamiento paralelo de las GPU NVidia. El objetivo que persigue la programación CUDA es sustituir los grandes bucles con los que se trabaja en algoritmos secuenciales, siempre que permitan una estructura de ejecución simultánea. Las funciones que realizan este cometido se denominan *kernels*.

Por otro lado, ya que se pretende sustituir un bucle de muchas iteraciones por un número equivalente de ejecuciones paralelas, se necesita controlar y planificar por orden de precedencia la ejecución de los procesos y subprocesos. Esto se obtiene gracias a los denominados *Grid*, *Block* y *Thread*:

- ***Thread***: es el hilo básico de ejecución, equivalente a una iteración del bucle en el caso serializado.
- ***Block***: matriz de una a tres dimensiones cuyo contenido son hilos o *threads*.
- ***Grid***: arreglos de *blocks*. Consta de hasta dos dimensiones.

De esta manera, a través de la posición del *thread* dentro de su *block* y de éste dentro del *grid*, se identifica el hilo de manera única.

Para empezar a ejecutar el algoritmo SIFT en la GPU se requiere un número de pasos de inicialización, en primer lugar, la imagen de entrada se carga en memoria, ésta se convierte en un arreglo unidimensional, por tanto, una llamada a los datos [xi + ancho*yi] proporciona acceso a los elementos de la imagen p (xi, yi) de acuerdo con ancho de la imagen, de esta manera la imagen puede ser transmitida a la GPU de manera eficiente.

Fase inicial de asignación. Esta etapa depende de una serie de parámetros SIFT, el tamaño de la imagen de entrada y el número de octavas, las diferencias gaussianas se calculan en función de estos parámetros, estos datos se utilizan tanto para asignar memoria lineal en la GPU, como para identificar el desplazamiento y el tamaño de cada imagen que necesita el algoritmo, la memoria se asigna antes de la etapa de generación espacial y se conserva a lo largo el funcionamiento del algoritmo SIFT, además, si se ejecuta para múltiples imágenes, estos datos se reutilizan para todas las imágenes del mismo tamaño. Este proceso se denomina preasignación de memoria (Figura 1).

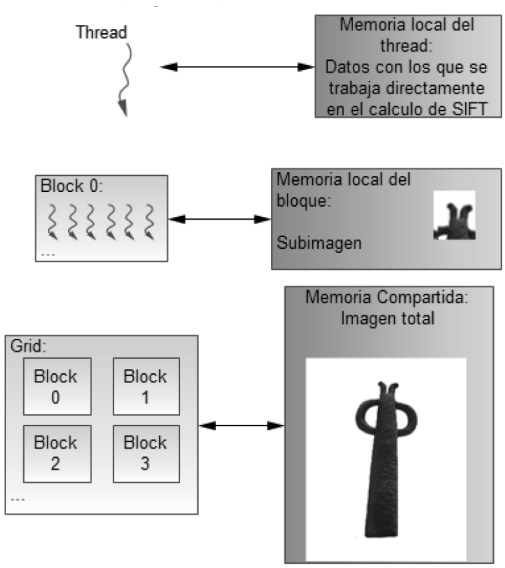

Figura 1. Distribución de datos en memoria.

PROGRAMACIÓN

A partir de este punto se desarrollarán los algoritmos necesarios para el cálculo SIFT. Para ello se tienen en cuenta todos los conceptos teóricos explicados anteriormente. Además, se presentan todas las posibilidades que se adoptaron para la resolución del mismo y el criterio de elección que se consideró para cada una de ellas.

CÓDIGO HOST (CPU)

Inicialmente se cargan las imágenes en la GPU aplicando el siguiente procedimiento:

1. Cargar las dos imágenes en la CPU.
2. Se reserva memoria para 2 imágenes a la GPU.
3. Se transfieren los arreglos a la memoria de la GPU.
4. Se pasan estos arreglos a texturas para optimizar su acceso y también para poder hacer otras operaciones, como interpolaciones.
5. Se procede a la creación de varios *grids* (bloques de hilos), para que se realicen desde la CPU las llamadas a los *kernels* necesarios (Figura 2).

Las funciones importantes utilizadas en este procedimiento son:

CUT_SAFE_CALL(cutLoadPGMub(image_path, &h_data, &ancho, &alto));

Esta función se utiliza para la carga de la imagen al puntero **h_data** de tipo *unsigned char*.

CUDA_SAFE_CALL(cudaMallocArray(&cu_array, &channelDesc, ancho, alto));

Esta función se utiliza para reservar memoria en la GPU, en un arreglo de las dimensiones de las imágenes.

CUDA_SAFE_CALL(cudaMemcpyToArray(cu_array, 0, 0, h_data, size, cudaMemcpyHostToDevice));

Esta función se utiliza para pasar la imagen que está almacenada en el arreglo en memoria de nuestro programa, al arreglo creado anteriormente (**cu_array**).

CUDA_SAFE_CALL(cudaBindTextureToArray(tex, cu_array, channelDesc));

Esta función vincula el arreglo almacenado en la memoria de la GPU, a una textura (**tex**), tipo de dato utilizado para tratar imágenes por CUDA.

Una vez llegado a este punto se procede a crear el *grid* de bloques de hilos mediante el uso de:

dim3 dimBlock(a, b, c);

Esta función genera un bloque de hilos de dimensiones (a, b, c) en (x, y, z), respectivamente.

dim3 dimGrid(d, e, f);

Esta función genera un *grid* de bloques de dimensiones (d, e, f) en (x, y, z), respectivamente. Una vez que se configuran estos parámetros se puede llamar al primer *kernel* que se ejecutará en el *dispositivo*.

ESTRATEGIA

Para enfocar la programación se propuso un par de alternativas:

1. Lanzar tantos hilos por etapa de SIFT por cada pixel que se desee tratar. Además se seleccionarán distintas dimensiones de bloques de hilos, así como el tamaño del *grid* (nótese que en principio para una GPU ideal sin restricción

de memoria, así como de hilos a lanzar, se obtendrán mejores resultados lanzando bloques con un número de hilos múltiplo de 16 [2]). El inconveniente que puede presentar este tipo de enfoque, es que se deberán sincronizar los hilos de cada bloque al momento en el cual quieran acceder a memoria compartida, con lo que se incrementará el tiempo de ejecución.

2. Lanzar tantos hilos como pixeles tiene la imagen. Cada hilo realizará las operaciones necesarias para la obtención de los cálculos.

Se debe aclarar que el método utilizado es el denominado 2, ya que los tiempos son aproximadamente 17 veces menores. Además, después de diversas pruebas se concluyó que la GPU gestiona mejor los bloques de una dimensión, que los de dos o tres.

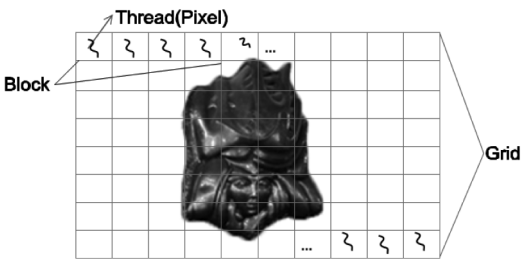

Figura 2. Distribución de trabajo de los *threads*.

Según [2] es recomendable lanzar bloques múltiplos de 64 hilos, para que sea más eficiente la gestión de memoria y por lo tanto los tiempos sean menores. Otra conclusión que se obtuvo fue que el tiempo era menor cuanto más se acerca el número de hilos a un múltiplo de 64. En vista de los resultados, se dedujo que el óptimo era escoger un bloque unidimensional múltiplo del ancho de la imagen.

Posteriormente se probaron distintos tamaños para el bloque de hilos, tales como 64, 128, 256 y 512 hilos. Los resultados que se obtuvieron fueron muy similares en todos los casos. Para el caso de 512 hilos, aunque en la GPU utilizada se pueden lanzar hasta 512 hilos simultáneamente, debido a la memoria necesaria por hilo, no es viable esta solución y se obtuvieron errores en la ejecución.

Se hace necesario aclarar que el tamaño del *grid* será el correspondiente para tratar la imagen completa y dependerá del tamaño de la misma.

Una vez finalizada esta fase y en vista de las pocas diferencias encontradas entre los distintos tamaños de bloque y de *grid* para la aplicación de dicho filtro, se propuso que fuese más funcional el programa y que se autoajustase a cualquier tamaño de imagen. Para ello se definió una variable llamada **ANCHO_BLOQUE**, la cual contiene la dimensión **x** del bloque de hilos a lanzar, y el *grid* lanzará los bloques necesarios para cubrir la dimensión de la imagen a procesar.

De este modo, aunque las diferencias son de menos de 1ms, apenas se verá afectado el resultado final, ya que el tiempo total del cálculo de SIFT será sobre dos órdenes de magnitud mayor, de este modo el *software* adquirirá una mayor flexibilidad al poder trabajar con distintos tamaños de imagen, sin modificar la programación.

Existe un problema derivado de esta estrategia: al almacenar la imagen en subimágenes de la misma, existe una pérdida de pixeles que no se analizan, estos se ubican donde se realiza el corte para obtener cada subimagen, estos pixeles llamados pixel frontera (PF) al igual que cualquier pixel, dentro de la imagen puede contener puntos de interés, por lo que se debe plantear una estrategia para solucionar esta pérdida, para ello se almacenan estos puntos en el momento de la partición de la imagen, se utiliza:

cudaMallocArray(&cu_frontera,&channelDesc,N_Cuadrante, Ancho_alto);

Donde:
N_Cuadrante: Número de subimagen a la que corresponden los PF, comenzando el conteo desde el primer cuadrante en la esquina superior izquierda
Ancho_alto: Tamaño de la cantidad de puntos reservados por cuadrante sumando el ancho y alto de la subimagen.

Como se observa en la Figura 3, para cada cuadrante se van almacenando sucesivamente sus fronteras inferior y derecha.

Posteriormente se realiza un análisis individual de cada PF, para determinar si es un punto de interés, tal cual como se realiza con cualquier pixel de la imagen.

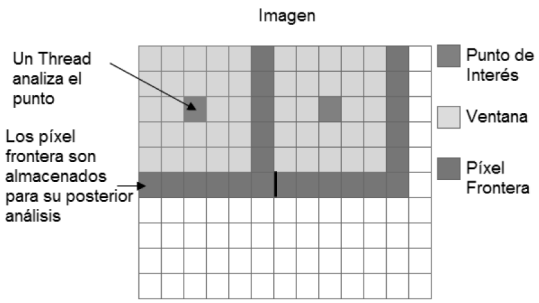

Figura 3. Estrategia límites de subimagen.

OBTENCIÓN DE DESCRIPTORES

Se realiza el cálculo de la magnitud del gradiente y orientaciones necesarias para esta etapa. Como estas dos operaciones son altamente paralelizables, son implementadas sobre todas las imágenes de espacio-escala gaussianas.

Sin embargo, mediante la utilización de los desplazamientos incluidos en la estructura de datos descriptores, la magnitud del gradiente y la orientación sólo se calcula para imágenes que contienen descriptores, esta técnica aumenta el rendimiento del algoritmo, cuando los intervalos de cada octava están obligados a ser grandes. A menudo se da el caso de que los descriptores no estarán presentes en los intervalos cercanos y por lo tanto los cálculos realizados en estos niveles serán excesivos.

Por lo tanto, un *kernel* debe realizar las operaciones de gradiente en la GPU para cada intervalo encontrado que contenga descriptores. Una vez más, todos los *threads* se utilizan para cargar una parte de la imagen, como una plataforma de anchura a memoria compartida.

El paso siguiente consiste en la asignación y el bloque de tamaños de Grid para la tarea de orientación. Como se describe en la aplicación de la CPU, la asignación de orientación consiste en la acumulación de todas las orientaciones en torno a una región de un punto clave p (x, y), donde el tamaño de la región está determinado por σ. Un acercamiento quizás ingenuo a la asignación de hilos para esta operación, es asignar un hilo para cada punto clave detectado. Sin embargo, esto significa que habría una reducción del número de bloques del *grid*.

En su lugar se asigna un subproceso a cada píxel en la ventana de ambas regiones magnitud y orientación, centrada en punto clave de la ubicación. Este enfoque entrega varios beneficios. En primer lugar, ya que cada subproceso se asigna de esta manera, las regiones centradas al punto de interés son proporcionadas al hilo a cargo de su ejecución, para la extracción de información de manera rápida. En segundo lugar, el número de subprocesos que se ejecutan al mismo tiempo es mayor, satisfaciendo los requisitos de acceso a la memoria. Las dimensiones de la *grid* y los bloques se conforman de la siguiente manera:

```
wblock = wwind +3;
hblock = hwind +3;
n(grid) = nextr;
```

Donde wwind = hwind son el ancho de la ventana la orientación y la altura y Nextr es el número de los descriptores de la imagen.

A continuación, una invocación de *syncthreads*() asegura que todos los datos dentro de cada región de ventana se han cargado correctamente. El hilo conductor de cada bloque es entonces responsable de calcular el histograma de las orientaciones, que se obtiene y almacena en la memoria local. Las principales orientaciones en el histograma entonces se encuentran en una situación similar como se ejecuta en CPU, y se agrega a la estructura de datos de puntos claves previamente descrito.

PRUEBAS

Las imágenes fueron captadas con una cámara digital Sony DSC-S930, de 10.1 megapixeles y zoom óptico de 3X, y trabajadas en un *notebook* Dell Vostro 1500, procesador Core 2 Duo 2.2Ghz, 3GB de RAM y una VGA GeForce 8600GT de 512MB.

Se realizaron tres pruebas para analizar la ganancia obtenida con el algoritmo realizado, cabe destacar que en ambas pruebas se utiliza un umbral de 32 y una ventana de 64x64 para el algoritmo de SIFT, ya que el paralelismo que se realizó funciona de manera óptima al trabajar con múltiplos de 64.

En la primera se realizó una comparación para calcular la mejora en la rapidez del proceso al usar paralelismo, utilizando CUDA. Los resultados

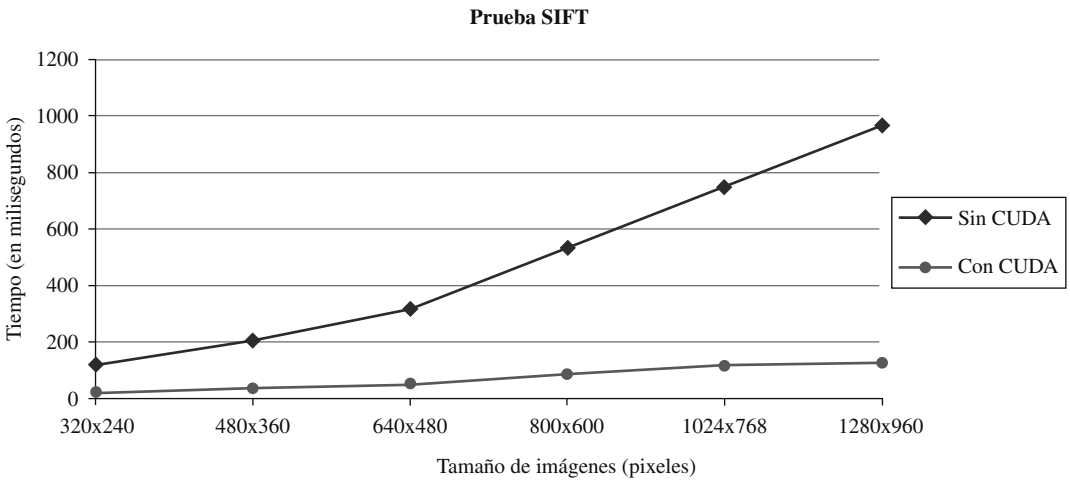

Figura 4. Gráfico de rendimiento.

obtenidos se pueden observar tanto en la Figura 4 como en la Tabla 1.

Tabla 1. Tiempo de reconstrucción en cuevas.

Resolución	Sin CUDA (ms)	Con CUDA (ms)
320x240	121	23
480x360	205	35
640x480	317	51
800x600	533	86
1024x768	748	117
1280x960	967	127

La imagen utilizada (Figura 5) para las pruebas corresponde a cuevas ubicadas en las costas de Arica.

Figura 5. Cueva costas de Arica.

La ganancia experimentada es altísima, dando un *speedup* de 7,6 en el mejor de los casos y 5,2 en el peor de ellos.

Para comprobar que la ganancia obtenida es meramente gracias al algoritmo desarrollado, también se realizó el cálculo de puntos obtenidos para demostrar que no había pérdida de ellos. Como se puede observar en el gráfico la cantidad de puntos obtenidos de manera secuencial es similar al aplicar paralelismo, como se ve en la Figura 6 y la Tabla 2.

Se realizó una segunda prueba, esta vez la imagen corresponde a las Presencias Tutelares (Figura 8) ubicadas fuera de Arica. Los resultados se pueden observar tanto en la Figura 7 como en la Tabla 2.

Tabla 2. Tiempo en Presencias Tutelares.

Resolución	Sin CUDA (ms)	Con CUDA (ms)
320x240	51	12
480x360	72	16
640x480	99	21
800x600	133	29
1024x768	172	38
1280x960	217	47

La ganancia experimentada en este caso no es tan alta como en el caso anterior, pues sólo se logra un *speedup* que varía entre 4 y 5, debido principalmente a que la prueba realizada se puede contar con pocos puntos de interés, en comparación con la prueba anterior, pero de todos modos sigue siendo un resultado muy bueno.

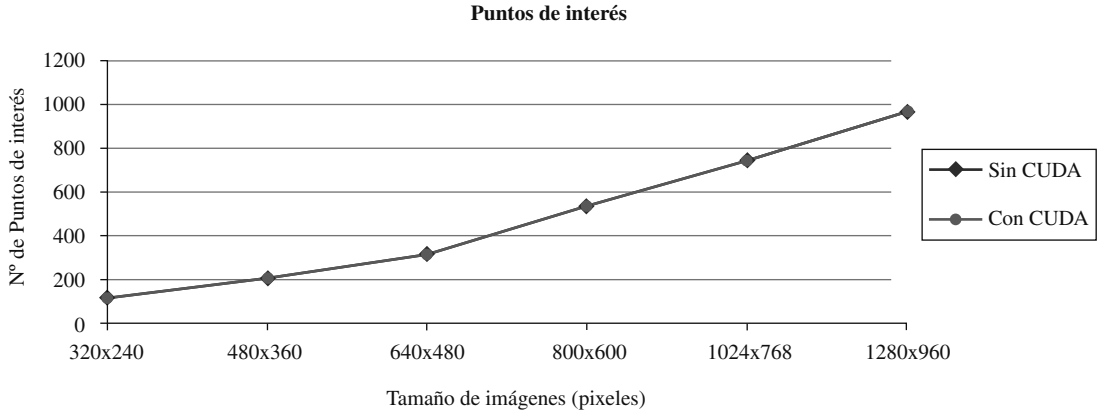

Figura 6. Gráfico de puntos de interés obtenidos.

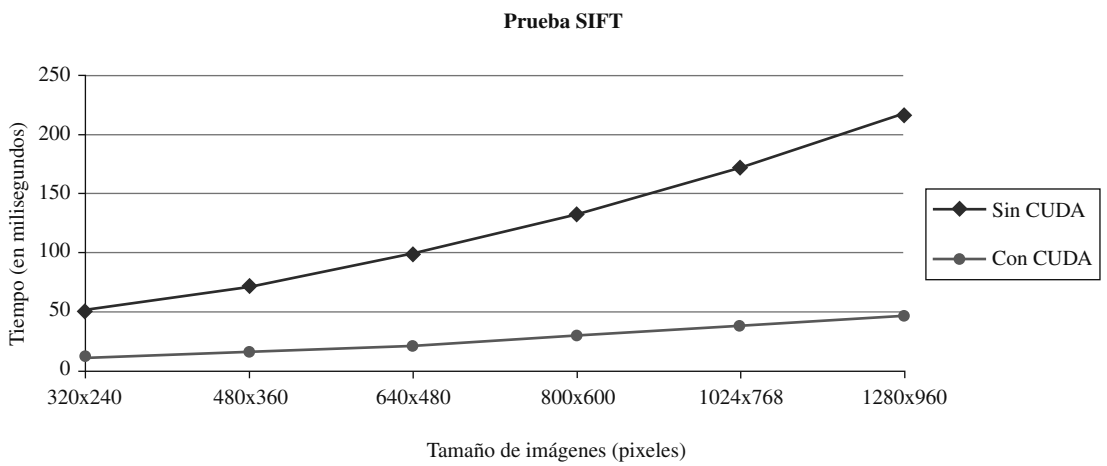

Figura 7. Gráfico de rendimiento.

Figura 8. Presencias Tutelares.

Para comprobar que la ganancia obtenida es meramente gracias al algoritmo desarrollado, también se realizó el cálculo de puntos obtenidos de manera secuencial, para demostrar que no había pérdida de ellos. Como se puede observar en el gráfico de la Figura 9, la cantidad de puntos es similar al aplicar paralelismo.

Tanto en la Figura 5 como en la Figura 9 se puede apreciar que no existe gran disparidad entre los puntos obtenidos por SIFT en paralelo y los obtenidos por SIFT secuencial; como se explicó antes, esto es de vital importancia porque demuestra que el algoritmo no altera en gran medida el funcionamiento del SIFT original, esto es debido que las mejoras realizadas apuntan a una subdivisión del trabajo realizando un paralelismo sobre las funciones de SIFT y no

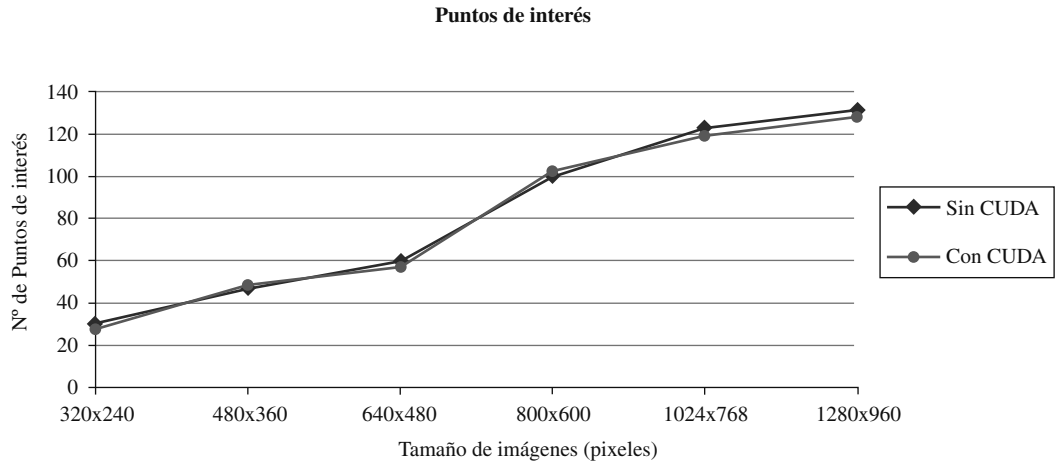

Figura 9. Gráfico de puntos de interés obtenidos.

sobre las instrucciones, además de una partición de memoria sin intervenir la estructura de SIFT como se ha visto en otros trabajos anteriores, donde son paralelizadas instrucciones individuales.

Para realizar la última prueba se captaron imágenes del Mural Amerindio de la Universidad de Tarapacá (Figura 10), registrando en este caso una secuencia de vistas del Mural con motivos indígenas, en un ambiente no preparado. Luego fueron editadas para trabajar con ellas bajo una situación ideal seleccionando una sección del mural correspondiente a un rostro.

Figura 10. Rostro amerindio.

Los resultados se pueden observar en la Figura 11 y la Tabla 3.

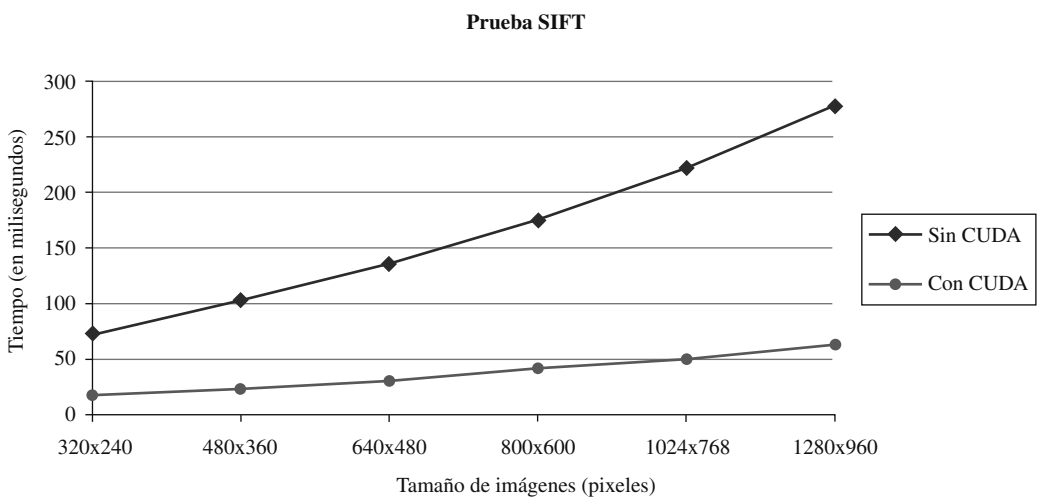

Figura 11. Gráfico de Rendimiento.

Tabla 3. Tiempo en prueba con imagen rostro.

Resolución	Sin CUDA (ms)	Con CUDA (ms)
320x240	73	18
480x360	103	23
640x480	136	31
800x600	175	42
1024x768	222	50

CONCLUSIONES

En lo concerniente a la paralelización a la vista de estos resultados, se puede afirmar, sin ánimo de generalizar, que el presente algoritmo de SIFT mejora en gran medida su rendimiento al paralelizarlos y ejecutarlos en una tarjeta gráfica, además los resultados obtenidos, en cuanto a puntos detectados, no se diferencian de los obtenidos en forma secuencial, es decir, la cantidad de puntos de interés no se ve afectada, pero a diferencia del método secuencial se cuenta con una gran ganancia en los tiempos de cálculo.

El volumen de puntos de interés detectados es relevante para el proceso de reconstrucción, por lo que el uso del método SIFT tiene ventajas sobre otros detectores de puntos característicos, además influye directamente en todo el resto de las etapas, más aún la calidad del *matching* realizado con ellos, ayuda a formar un modelo cercano al real y una buena calidad de reconstrucción 3D final.

TRABAJO FUTURO

En lo que respecta al paralelismo se debe centrar en una experimentación con diferentes estrategias de distribución y agrupamiento de núcleos, los que permitan mejorar el desempeño.

REFERENCIAS

[1] C. Wu. "SiftGPU: A GPU Implementation of Scale Invariant Feature Transform (SIFT)". September, 2008. Date of visit: August, 2010. URL: http://cs.unc.edu/~ccwu/siftgpu/

[2] Nvidia Corporation. NVIDIA CUDA Programming Guide (version 2.3.1). Chap. 5, pp. 52-53. 2009.

[3] D.G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". International Journal of Computer Vision. Vol. 60, pp. 91-110. 2004.

[4] A. Martín. "Generación de mapas de disparidad, utilizando CUDA". Universidad Carlos III de Madrid. Madrid, España. 2009.

[5] Q. Zhang, Y. Chen, Y. Zhang and Y. Xu. "SIFT implementation and optimization for multi-core systems". IPDPS 2008. 2008.

[6] I. Geys and L.V. Gool. "View synthesis by the parallel use of GPU and CPU". Image and Vision Computing. 2007.

[7] S.N. Sinha, J.-M. Frahm, M. Pollefeys and Y. Genc. "GPU-Based Video Feature Tracking and Matching". EDGE 2006. Workshop on Edge Computing Using New Commodity Architectures, Chapel Hill. May, 2006.

[8] C. Tomasi and T. Kanade. Detection and Tracking of Point Features. Carnegie Mellon University Technical Report CMU-CS-91-132. April, 1991.