Aceleración en GPU del Proceso de Cálculo de Patrones de Color en Mallas Triangulares Generadas a Partir de Imágenes

Marcos Paulo Yauri Aranibar
Programa Profesional de Ciencia de la Computación
UCSP
Arequipa, Peru
Email: marcos.yauri@ucsp.edu.pe

Alex Cuadros Vargas
Programa Profesional de Ciencia de la Computación
UCSP
Arequipa, Peru
Email: alex@ucsp.edu.pe

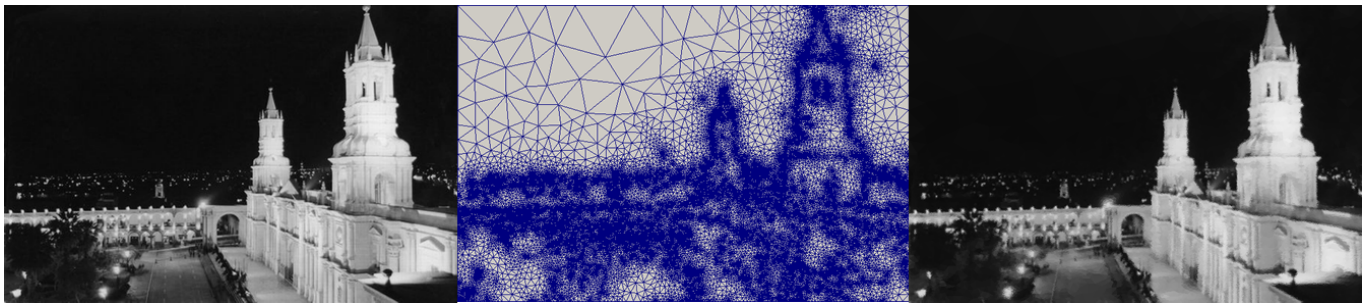

Figura 1. Imagen de prueba: Catedral de Arequipa (izq.), Malla triangular relacionada a la imagen de prueba (medio), Malla coloreada con la media de color obtenida mediante los algoritmos propuestos de barrido simplicial (der.).

***Resumen*—En el área de Computación Gráfica, existen diferentes algoritmos que generan una malla geométrica de triángulos a partir de una imagen. Para poder añadir información representativa de la imagen a esta malla, es necesario evaluar cada triángulo, barriendo cada pixel en él. Este procesamiento se realiza tradicionalmente de manera secuencial, lo cual en términos de tiempo es un proceso costoso. Este costo dificulta el desarrollo, visualización y optimización de los procesos que incluyan este como subproceso. Existen algoritmos consolidados para el barrido de polígonos de manera individual, sin embargo no hay estudios que propongan su optimización en conjuntos de polígonos a gran escala. Se plantea llevar a paralelismo variaciones de un algoritmo secuencial de barrido mallas triangulares, haciendo uso de multi-threading en *CPU* (Central Processing Unit)y masive-threading en *GPU* (Graphics Processing Unit), y evaluar el desempeño de estos nuevos algoritmos.**

***Keywords*-Barrido de polígonos; Generación de Mallas; Triangulación de Delaunay**

***Abstract*—In Computer Graphics there are different algorithms that generate a triangular mesh from an image. In order to add representative information image to this mesh, it is necessary to evaluate each triangle, scanning each pixel in it. This processing is traditionally performed sequentally, which in terms of time is costly. This cost hinders processes development, display, and optimization. There are bound algorithms for scanning individual polygons, however no studies propose large scale polygon sets scanning. We propose taking variations of a sequential triangular mesh scanning algorithm to parallelism using multi-threading *CPU* (Central Processing Unit) and masive-threading on *GPU* (Graphics Processing Unit), and evaluate the performance of these new algorithms.**

***Keywords*-Polygon scan; Mesh generation; Delaunay triangulation**

I. Introducción

En los últimos años, los *GPU* (*Graphics Processing Unit*) han surgido como dispositivos poderosos de cómputo que pueden soportar soluciones paralelas de propósito general en el trato de cantidades masivas de datos haciendo uso de sus cientos de procesadores. Entre las plataformas de computación paralela en *GPU*, la arquitectura *CUDA* [1](*Compute Unified Device Architecture*) provee un modelo de programación intuitivo y escalable basado en el lenguaje de programación *C*: *C-CUDA*. Debido a la practicidad de uso de *C-CUDA*, muchos algoritmos consolidados de diferentes campos de la ciencia y la ingeniería han sido rediseñados con la finalidad de acelerar sus tiempos de ejecución. En ese momento nace el campo de la computación general en hardware gráfico *GPGPU* (*General-Purpose Computing on Graphics Processing Units*) [2]. Sin embargo, la eficiencia en el tiempo de las aplicaciones *C-CUDA* depende de considerar varias propiedades técnicas de la arquitectura *GPU* durante el desarrollo e implementación. Sin un cuidado delicado de estas consideraciones, los programas paralelos en *C-CUDA* pueden ser incluso más lentos que sus pares secuenciales.

Por otro lado, junto con este desarrollo del procesamiento en *GPU*, surgieron nuevas técnicas, que mezclan geometría e imágenes, tales como el algoritmo *Imesh* [3] capaz de representar una imagen a partir de cientos, o miles, de triángulos. Luego de generar una geometría ajustada a los rasgos de la imagen, Imesh necesita barrer el interior de cada triángulo con la finalidad de insertar en la malla información de la imagen. Este procesamiento de barrido en el algoritmo Imesh, es realizado secuencialmente en *CPU*, y representa un tiempo considerable en el tiempo total de la técnica. Siendo así, esta demora dificulta labores de depuración y optimización del proceso, así como la visualización de resultados y el desarrollo en general. Es importante considerar que aunque existen estudios consolidados sobre la optimización de barrido de triángulos individuales [4] [5], no existen estudios para la optimización del barrido de grandes cantidades de ellos ni en *CPU* ni en *GPU*.

Contribución: De esta forma, esta investigación propone el uso de paralelismo para disminuir el tiempo de los procesos de barrido, comparando el comportamiento de cuatro alternativas de algoritmos basados en: 1) Procesamiento secuencial en *CPU*, 2) Descomposición en triángulos en *CPU*, 3) Descomposición en líneas en *CPU* y 4) Descomposición en líneas en *CPU+GPU*. Para una mejor comparación de estos procesos, utilizaremos un conjunto de mallas triangulares sintéticas, con una cantidad de elementos creciente, desde pocos elementos hasta grandes cantidades de elementos. En las pruebas realizadas en este trabajo el proceso de barrido estará enfocado en calcular la media de colores en el interior de un triángulo, sin embargo este barrido podría ser empleado para calcular diferentes funciones o medidas estadísticas como varianza, media, entre otras. En nuestros experimentos observamos que nuestra propuesta resulta ser hasta 4 veces más rápida con respecto al enfoque original bidimensional. Aunque esta diferencia de tiempo, en un contexto bidimensional, no es muy grande, en nuestros experimentos observamos que mientras el número de elementos aumenta, es mayor la proporción de diferencia de tiempo entre los procesos de *CPU* y *GPU*. Este hecho es interesante puesto que nuestro objetivo a futuro es acelerar el proceso con mallas tridimensionales donde se espera una cantidad de elementos mucho mayor.

A. Trabajos Relacionados

Los trabajos están organizados en dos bloques, el primero relacionado a la paralelización de algoritmos secuenciales haciendo uso de *massive-threading GPU* y el segundo, el relacionado con el barrido de triángulos y mallas triangulares.

Dentro del primer bloque están los trabajos relacionados a la paralelización de algoritmos regulares, es decir que no muestran dependencia de datos, y que por lo tanto son fácilmente traducidos a un entorno *multi-threading* o *massive-threading*, el primero de ellos [6] realiza una introducción a *GPGPU*, analizando el rendimiento de *CUDA*, concluye que la mejora en velocidad con respecto a *CPU* depende del porcentaje del problema que es inherentemente paralelo y de la intensidad aritmética presente en él. El siguiente trabajo [7] presenta un sistema de partículas adaptado para la extracción de superficies en *GPU*, concluyen que a mayor cantidad de partículas (elementos procesados), mayor es la rapidez de *GPU* comparada al procesamiento en *CPU*, debemos destacar que el sistema de partículas en el cual se basa no es inherentemente paralelizable, pero el autor lo soluciona añadiendo el concepto de *bins* (colectores) para evitar conflictos de compartición de datos entre las partículas. En el trabajo de Rueda[8] se evalúa el comportamiento de dos algoritmos geométricos para mallas 3D, los resultados obtenidos son mejores para mallas con cantidades mayores de triángulos. Estos algoritmos no necesitaron de sincronización para acceder a la memoria y disponían de independencia de datos, lo que facilitó el trabajo independiente de todas las hebras. Estas propuestas de paralelización de algoritmos conocidos no se limitan al campo de la computación gráfica, sino que lo trascienden, acorde con la generalidad de la *GPGPU*; por ejemplo tenemos trabajos [9] que muestran la paralelización de una aplicación financiera basada en el método de *Monte-Carlo*, demostrando que el rendimiento en *GPU*, usando *CUDA* es asciende a $190\,\%$ comparado con el rendimiento en *CPU*; o la propuesta de paralelización de la clusterización de Markov con aplicación en la biocomputación, que nuevamente tiene un desempeño notablemente mejor que su par en *CPU* [10].

El segundo grupo de trabajos relacionados se refiere a los algoritmos de barrido de triángulos, uno de ellos hace la evaluación de un conjunto de algoritmos de barrido de primitivas geométricas, para obtener resultados de su eficiencia computacional [4], algunos de estos algoritmos son paralelizables. En la siguiente investigación [5], se trabajan diferentes técnicas de barrido de triángulos haciendo uso de coordenadas homogéneas, aprovechando la linealidad de procesamiento que estas ofrecen. Sin embargo, estas técnicas evalúan su desempeño en elementos individuales. Aunque existen algoritmos consolidados para el barrido de triángulos, no hay estudios que evalúen su optimización para el tratamiento de conjuntos de gran cantidad de triángulos, pudiendo resultar que un algoritmo válido para la evaluación de un elemento, no lo sea para la evaluación de un conjunto de esos mismos elementos.

Existen investigaciones [7][8][9] en las que podemos observar el mejor rendimiento del procesamiento en *GPU* para cantidades masivas de procesos elementales, lo cual nos invita a pensar que la solución planteada en esta investigación resultará positiva, brindando una notable mejora con respecto a su par en *CPU*.

II. Marco Teórico

A. Imesh[3]

Imesh, es un algoritmo concebido para crear mallas triangulares directamente a partir de imágenes de entrada no preprocesadas.

El algoritmo Imesh se ejecuta en tres etapas diferenciadas: construcción de la malla, particionamiento de la malla y mejora de calidad de la malla.

- Construcción de la malla: Comienza a partir de una imagen no preprocesada, y tiene como objetivo representar la

imagen de entrada con una malla de *Delaunay*. Especificamente la imagen es representada utilizando patrones de color calculados a partir de los pixeles contenidos dentro de cada triangulo de la malla.

- Particionamiento de la malla: Esta segunda etapa tiene por objetivo descomponer la malla generada en el proceso anterior, demanera que se pueda distinguir diferentes estructuras contenidas en la malla. Este problema es semejante al problema de segmentación de imagenes.
- Mejora de calidad de la malla: Esta última etapa es utilizada para insertar criterios de calidad en la malla particionada por el proceso anterior. Para esto utiliza ideas de algoritmos de refinamiento de calidad *Delaunay*.

En la primera etapa, una vez obtenida la malla a partir de la imagen de entrada se hace necesario el barrido de los triangulos a fin de calcular la media del patrón de color de cada pixel contenido en el triangulo, este cálculo secuencial dependiendo de la cantidad de células es notablemente tardío. Es importante resaltar que el objetivo principal de este trabajo es reducir los tiempos de barrido de mallas, contribuyendo con el procesamiento de mallas en general, como en el caso de Imesh [3].

B. Boost C++

Boost para *C++* provee alrededor de 80 librerías de propósito general, estructuras bien diseñadas y herramientas que pueden ser utilizadas en un amplio rango de aplicaciones. Dentro de ellas la librería *Boost.Thread* hace posible el uso de hebras, la interfaz provee la capacidad de aplicar hebras con datos compartidos en cualquier aplicación independientemente del sistema operativo que se esté usando. Las hebras pueden ser creadas con funciones definidas por el usuario y miembros de clase. Está pensada como una plataforma de investigación para facilitar la experimentación y proveer implementaciones sólidas para resolver problemas de gran escala en otras áreas de aplicación [11]. *C++* estándar no contiene soporte nativo para *multi-threading*, lo que significa que no es posible escribir código portable con hebras como se escribiría con otras librerías estándar como *string*, *vector*, *list* y otras. Actualmente diez de las librerías de Boost han sido incluidas como parte de librería estándar de *C++11*, antes conocido como *C++0x*, y varias más han sido propuestas para la versión *TR2*.

C. CUDA

CUDA es la arquitectura de computación paralela desarrollada por *NVIDIA*. Permite el incremento dramático de la capacidad de programación usando los núcleos de un *GPU*. Su rango de uso no se limita a la computación gráfica, sino que va a todo tipo de aplicaciones como la biología computacional o la química a esto se llama Computación de Propósito General *GPGPU*.

Las tarjetas gráficas más actuales dedican la mayor parte de sus transistores a crear unidades de procesamiento más que al control de flujos de datos. En esta línea, una tarjeta gráfica de coste medio puede tener alrededor de 128 procesadores, con capacidad de ejecutar tanto tareas inherentes al renderizado de imágenes como programas de propósito general, todo ello de forma paralela.

Estas condiciones de ejecución hacen atractivo establecer una competición entre el rendimiento de una *CPU* y una *GPU*. En términos generales, los algoritmos a ejecutar en una *GPU* van a ser versiones paralelas de los que se ejecutan en la *CPU*, pues es esta la forma de conseguir un mayor rendimiento de la misma. Ver figura: 2.

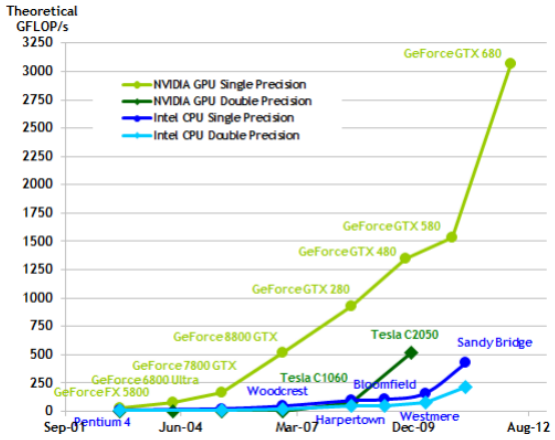

Figura 2. Operaciones de punto flotante por segundo para *CPU* y *GPU* [1]

Por otra parte, 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, como la codificación/decodificación de vídeos y audios, el escalado de imágenes, y el reconocimiento de patrones en imágenes.

El anfitrión es el computador *CPU* al cual está conectado el dispositivo. El dispositivo es un *GPU* conectado a un anfitrión para ejecutar la parte de la aplicación que es de computación intensiva y datos paralelos. *Kernel* es una función llamada desde el *CPU* anfitrión y ejecutada en paralelo en el dispositivo *CUDA* por cientos o miles de hebras. Una aplicación o librería consiste en uno o varios *kernel*. [12]

Un programa *CUDA* usa kernels para operar en el flujo de datos. Ejemplos de flujo de datos son vectores de elementos o un conjunto de píxeles. *CUDA* provee tres mecanismos para paralelizar programas: Jerarquía del grupo de hebras, memoria compartida y barrera de sincronización.

En la figura 3, observamos que la jerarquía en *CUDA* es como sigue:

- Las hebras son el nivel más elemental de la arquitectura.
- Un bloque está compuesto de varias hebras.
- Una grilla está compuesta de varios bloques de hebras concurrentes.
- La memoria local es solo accesible a la hebra propietaria.
- La memoria compartida es accesible por todas las hebras en un mismo bloque.
- La memoria global es accesible por todos los bloques.

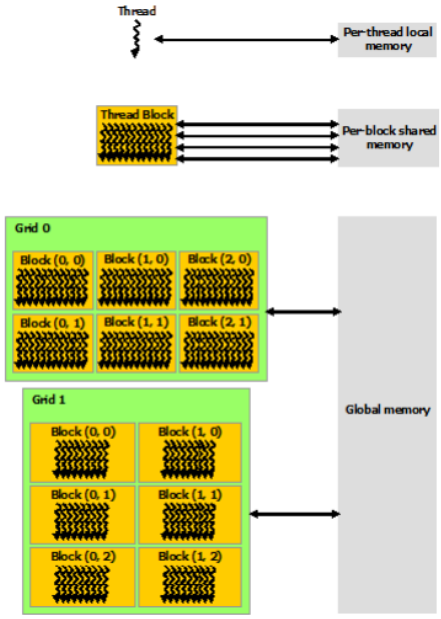

Figura 3. Jerarquía de memoria en *CUDA* [1]

III. Propuesta

La propuesta de este trabajo es realizar un estudio para determinar cuál es la mejor opción en términos de tiempo de ejecución, para calcular patrones de color de un conjunto grande de triángulos contenidos en una malla, la misma que fue generada a partir de una imagen bidimensional. Para esto analizaremos 4 opciones descritas a continuación. En los procesos descritos a continuación se utilizará partes con *multi-threading*, para lo cual utilizaremos la librería *Boost.Thread* para *C++* (ver II-B), y en otras partes se utilizará procesos con *massive-threading* para la cual utilizaremos *C-CUDA* (ver II-C).

Procesamiento Secuencial en CPU: En esta primera técnica se analiza cada triángulo aplicando el clásico algoritmo *polygon scan* [3] . Esta opción es la que se utiliza dentro del algoritmo *Imesh* para obtener la media de color de los píxeles contenidos en cada triángulo de la malla generada, y es precisamente el proceso que se pretende mejorar. En este algoritmo primeramente se obtienen los puntos extremos del triángulo tanto en el eje-x como en el eje-y para luego proceder con el barrido, partiendo desde estos extremos (ver Fig.4)

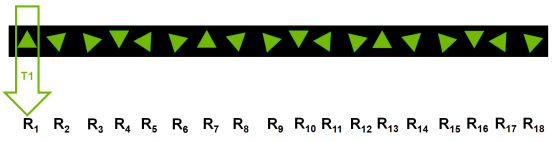

Figura 4. Procesamiento Secuencial en *CPU*

Descomposición en Triángulos en CPU: Esta opción fue pensada para una ejecución *multi-threading* en *CPU*. Esta técnica consiste en la división equitativa de la cantidad de triángulos a procesar entre el número de hebras, de acuerdo a la capacidad del procesador utilizado. Sin embargo, cada hebra aplica la misma idea de la primera opción, sin modificación, a cada triángulo (ver Fig.5).

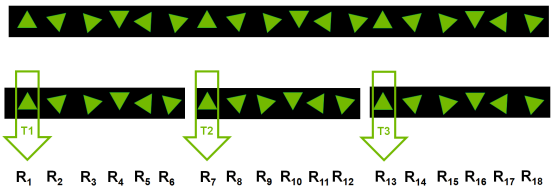

Figura 5. Descomposición en Triángulos en *CPU*

Descomposición en líneas en CPU: Esta propuesta se ejecuta en tres etapas diferenciadas. a) La primera etapa consiste en la descomposición de cada triángulo en segmentos de línea horizontales, b) una segunda etapa realiza el barrido de cada segmento de línea y c) la tercera compone los resultados de los barridos de cada línea en los resultados totales de cada triángulo. Estas etapas se realizan de manera concurrente con multi-threading en *CPU* (ver Fig.6).

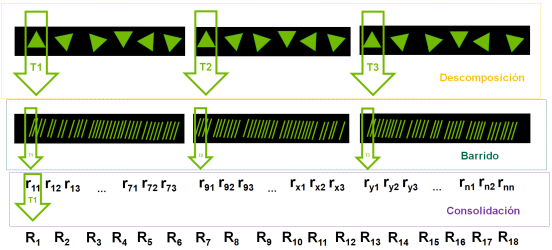

Figura 6. Descomposición en líneas en *CPU*

Descomposición en líneas en CPU+GPU: Esta propuesta está basada en una modificación de la tercera propuesta de la siguiente manera. Las etapas 3a y 3c quedan iguales. Por otro lado, en la etapa 3b se realiza el barrido de líneas utilizando paralelismo masivo en *GPU*, una línea por hebra (ver Fig.7)

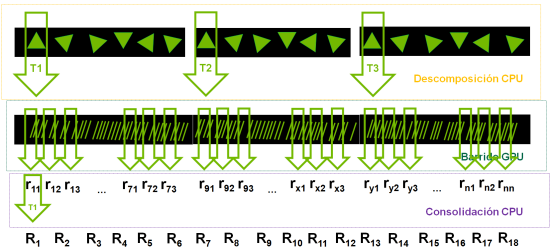

Figura 7. Descomposición en líneas en *CPU+GPU*

A. Evaluación

El desempeño de cada algoritmo será medido en términos de tiempo de ejecución. Las curvas de desempeño obtenidas serán comparadas a fin de obtener el punto de equilibrio entre ellas. Este punto de equilibrio nos indicará el tipo de procesamiento más adecuado para cierta cantidad de elementos.

Una malla triangular convecional no supera la cantidad de 4000 triángulos aproximadamente, es por ello que se elaboran mallas triangulares sintéticas de 100000, 150000, 200000 y 800000 elementos, a fin de lograr una evaluación y comparación más consistente del desempeño de cada algoritmo.

IV. Implementación

Todos los algoritmos han sido implementados bajo el lenguaje de programación *C++*. Para los algoritmos concurrentes en *CPU*, simple y por descomposición, se ha utilizado la librería *Boost.Thread*. Para el algoritmo concurrente híbrido en *CPU* y *GPU* se ha utilizado *C-CUDA*. Dado que tanto *CPU* como *GPU* poseen varios núcleos o procesadores, los algoritmos han sido afinados para alcanzar el uso de el 100 % de la capacidad de ambos.

A fin de validar los cuatro algoritmos de barrido se ha utilizado una función simple de cálculo de la media aritmética del valor de color de cada pixel de la imagen usando su respectiva malla triangular. El resultado obtenido tras la ejecución de los cuatro algoritmos en la Figura 1, es el mismo para los cuatro y guarda coherencia con el resultado esperado.

V. Experimentos y Resultados

Se presentan los resultados de ejecutar los diferentes algoritmos en las mallas bidimensionales obtenidas sintéticamente, principalmente enfocado en comparar los tiempos computacionales de ejecución y las curvas descritas por el comportamiento de cada algoritmo.

En la Figura 8, podemos ver el comportamiento de los cuatro diferentes algoritmos ejecutados, mientras que en el Cuadro I observamos los resultado obtenidos para cada algoritmo. Destaca el algoritmo de Descomposición en líneas en *CPU* (Exp.3) por sus altos tiempos de ejecución, mucho mayor que los otros tres. Para una mejor visualización de los otros tres algoritmos, en la Figura 9 se omite el comportamiento del algoritmo por descomposición. Tanto el algoritmo de descomposición en líneas en *CPU* (Exp.2) como el algoritmo de Descomposición en líneas en *CPU+GPU* (Exp.4) muestran comportamientos similares, mientras que el Procesamiento secuencial en *CPU* (Exp.1) tiene un elevado costo en tiempo comparado con los anteriores. También es cierto, que el algoritmo Descomposición en líneas en *CPU* (Exp.3), solo encuentra razón de ser como precursor del algoritmo Descomposición en líneas en *CPU+GPU*(Exp.4).

Para realizar un mejor análisis comparativo de los algoritmos Descomposición en triángulos en *CPU* (Exp.2) y por Descomposición en líneas en *CPU+GPU* (Exp.4), se realizan nuevas pruebas, en un conjunto más amplio de mallas sintéticas (ver Fig.10). El algoritmo por Descomposición en triángulos en *CPU* muestra un mejor desempeño para mallas de menos de 6250 células aproximadamente, a partir de esta cantidad el algoritmo por Descomposición en líneas en *CPU+GPU* tiene mejor desempeño como se observa en el gráfico 11.

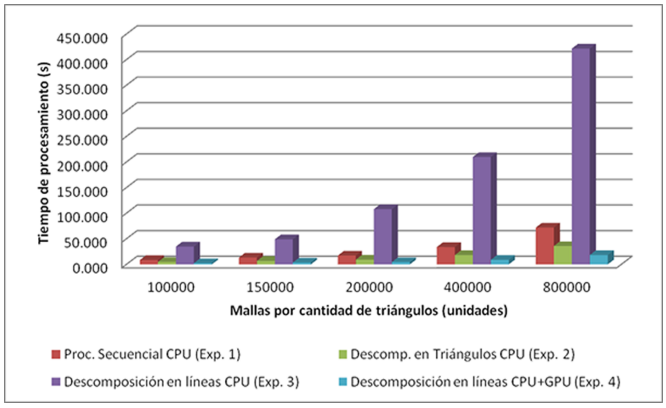

Figura 8. Comparación de Resultados

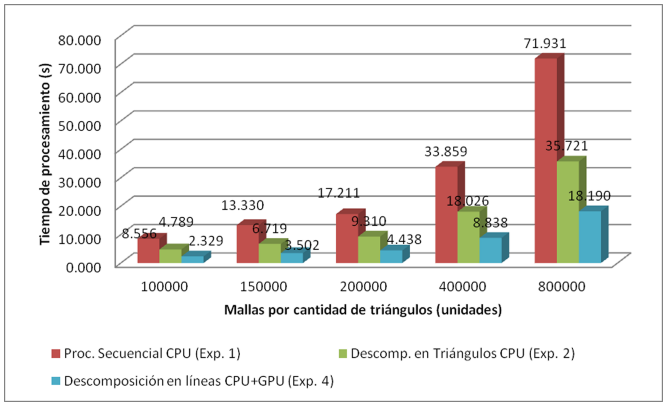

Figura 9. Comparación de Resultados

A. Comparación de Desempeños

Todos los resultados fueron obtenidos de la ejecución de las implementaciones en un computador con *CPU* de doble procesador *2128Mhz*, *64-bit*, y *4063 MB* de *RAM*, y *GPU GeForce 9600M GT*, de 64 núcleos *CUDA*.

En el Cuadro II, observamos la comparativa de mejora (*speedup*) de todos los algoritmos respecto al Algoritmo Procesamiento secuencial en *CPU*. El mejor speedup lo sostiene el Algoritmo por Descomposición en líneas en *CPU+GPU* (Exp.4), siendo en promedio 3,8 veces de mejor rendimiento que el Procesamiento Secuencial (Exp.1).

Cuadro I
Comparación de Resultados

Cant.	Exp. 1(s)	Exp. 2(s)	Exp. 3(s)	Exp. 4(s)
3125	0.305	0.172	1.467	0.200
6250	0.872	0.272	2.349	0.260
12500	1.522	0.753	3.539	0.321
25000	2.593	1.085	9.393	0.595
50000	5.666	2.350	16.287	1.162
100000	8.556	4.789	34.730	2.329
150000	13.330	6.719	49.072	3.502
200000	17.211	9.310	107.848	4.438
400000	33.859	18.026	209.750	8.838
800000	71.931	35.721	421.590	18.190

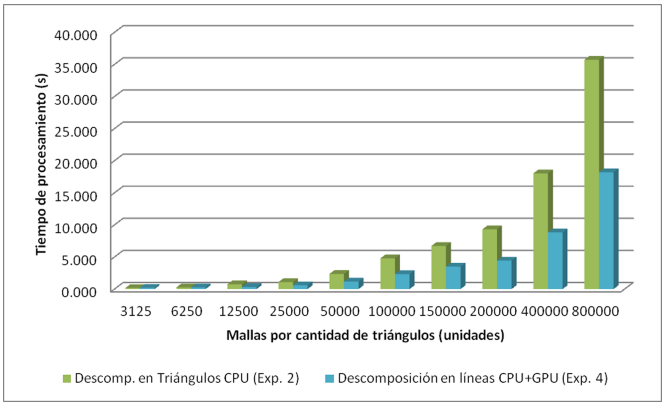

Figura 10. Comparación de Resultados Alg. Descomposición en triángulos en *CPU* y Descomposición en líneas en *CPU+GPU*

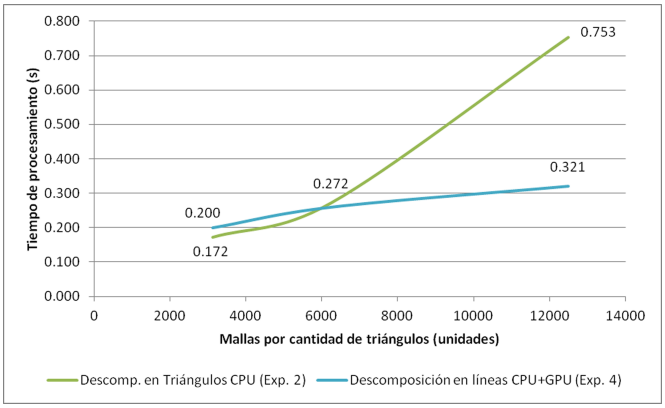

Figura 11. Curvas de comportamiento Alg. Descomposición en triángulos en *CPU* y Descomposición en líneas en *CPU+GPU*

VI. Conclusión

En esta investigación se presenta un nuevo algoritmo de procesamiento *CPU+GPU* relacionado al procesamiento de mallas triangulares a partir de imágenes. Este nuevo algoritmo adapta la versión secuencial para ejecución en multiprocesadores, finalmente muestra un notable mejor desenvolvimiento que llega a ser 4 veces más rápido que el procesamiento secuencial original, haciendo uso del $100\,\%$ de los procesadores *CPU* y *GPU*.

Una característica interesante que presenta esta nueva técnica, es que puede ser fácilmente extendida a mallas tridimensionales, considerando que las células constitutivas de estas mallas son tetraedros y que estos pueden ser descompuestos en triángulos y finalmente en segmentos de linea, este será un trabajo futuro de esta investigación.

La técnica propuesta utiliza la nueva tecnología existente para procesamiento multiprocesador, la mejora acelerada de esta tecnología acarreara consigo un mejor desempeño de esta técnica. La segmentación de datos utilizada hace que no se requiera comunicación entre procesadores, lo cual permite aprovechar la totalidad de la capacidad de cómputo, evitando cuellos de botella y uso de semáforos.

Esta técnica puede ser utilizada en diferentes tipos de procesamiento de malla que necesiten recorrer el interior de los elementos de la malla como *Imesh*, facilitando una mayor rapidez de ejecución y depuración.

Como se observa en los comentarios anteriores, el tipo de abordaje descrito en este trabajo presenta resultados promisorios para aplicaciones de procesamiento de mallas, abriendo el horizonte para nuevos desafíos en el procesamiento de imágenes.

Cuadro II
Comparación de Rendimientos Respecto a Procesamiento Secuencial

Cant.	Speedup Exp.2	Speedup Exp.3	Speedup Exp.4
100000	1.787	0.246	3.674
150000	1.984	0.272	3.806
200000	1.849	0.160	3.878
400000	1.878	0.161	3.831
800000	2.014	0.171	3.954
Prom.	1.902	0.202	3.829

Referencias

[1] NVIDIA, *NVIDIA CUDA Programming Guide 5.0*. NVIDIA Corporation, 2012.

[2] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[3] A. Cuadros-Vargas, M. Lizier, R. Minghim, and L. Nonato, "Generating segmented quality meshes from images," Ph.D. dissertation, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10851-008-0105-2

[4] L. Huang, D. Crisu, and S. Cotofana, "Heuristic algorithms for primitive traversal acceleration in tile-based rasterization," 2008.

[5] M. Olano and T. Greer, "Triangle scan conversion using 2d homogeneous coordinates," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. ACM, 1997, pp. 89–95.

[6] E. Wynters, "Parallel processing on nvidia graphics processing units using cuda," *J. Comput. Sci. Coll.*, vol. 26, no. 3, pp. 58–66, Jan. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1859159.1859173

[7] M. Kim, G. Chen, and C. Hansen, "Dynamic particle system for mesh extraction on the gpu," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 38–46. [Online]. Available: http://doi.acm.org/10.1145/2159430.2159435

[8] A. Rueda and L. Ortega, "Geometric algorithms on CUDA," in *International Conference on Computer Graphics Theory and Applications*, Jan. 2008, pp. 107–112. [Online]. Available: http://www.gvu.gatech.edu/~{}jarek/graphics/reading/cuda.pdf

[9] M. Lee, J.-h. Jeon, J. Bae, and H.-S. Jang, "Parallel implementation of a financial application on a gpu," in *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ser. ICIS '09. New York, NY, USA: ACM, 2009, pp. 1136–1141. [Online]. Available: http://doi.acm.org/10.1145/1655925.1656132

[10] A. Bustamam, K. Burrage, and N. A. Hamilton, "Fast parallel markov clustering in bioinformatics using massively parallel computing on gpu with cuda and ellpack-r sparse format," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 9, no. 3, pp. 679–692, May 2012. [Online]. Available: http://dx.doi.org/10.1109/TCBB.2011.68

[11] S. Koranne, *Handbook of Open Source Tools*. Springer, 2010. [Online]. Available: http://books.google.es/books?id=ukXrNh2g6fQC

[12] F. Gebali, *Algorithms and Parallel Computing*, ser. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, 2011. [Online]. Available: http://books.google.com.pe/books?id=3g6lrxrd4wsC