Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería
Escuela de Graduados

ACELERANDO EL CÁLCULO DEL CONVEX HULL CON UN ALGORITMO PARALELO EN GPU

Tesis de Magíster en Informática

Alan Edward Keith Paz

Valdivia, abril 2022

Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería

Escuela de Graduados

ACELERANDO EL CÁLCULO DEL CONVEX HULL CON UN ALGORITMO PARALELO EN GPU

Alan Edward Keith Paz

Certificación de Aprobación:

Dr. Héctor Ferrada Escobar
Profesor Patrocinante
Instituto de Informática, UACh

Dr. Eliana Scheihing García
Profesora Informante
Instituto de Informática, UACh

Dr. Cristóbal Navarro Guerrero
Profesor Copatrocinante
Instituto de Informática, UACh

Dr. Nancy Hitschfeld Kahler
Profesora Informante Externo
Departamento de Ciencias de la Computación,
Universidad de Chile

Dr. Enrique Suárez Silva
Decano
Facultad de Ciencias de la Ingeniería, UACh

Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería

Escuela de Graduados

ACELERANDO EL CÁLCULO DEL CONVEX HULL CON UN ALGORITMO PARALELO EN GPU

Esta tesis es presentada a la Escuela de Graduados de la Facultad de Ciencias de la Ingeniería de la Universidad Austral de Chile, como parte de los requisitos para la obtención del grado de Magíster en Informática. Certifico que el contenido de este trabajo es original, excepto cuando se hace referencia al trabajo de otros y, para que quede constancia, firma en Valdivia, el 7 de abril de 2022,

_____

Alan Edward Keith Paz

Accelerating the Convex Hull Calculation with a Parallel GPU Algorithm*

Alan Keith Paz
Universidad Austral de Chile
alan.keithpaz@gmail.com
Valdivia, Chile

Resumen

The calculation of the convex hull is a fundamental problem in computational geometry, optimization and mathematics, contributing in many fields, especially in those time-sensitive ones (Collision detection, image processing for virtual reality, etc). For n entry points, it can not be calculated in less than $O(n \log n)$ time in the RAM model[1] and there are many algorithms that calculate it in $(n \log n)$ which is theoretically optimal. The work of Ferrada et al. [25] performs a preliminary stage to filter the points that will be considered to compute the Convex Hull, which in practice significantly accelerates the calculation time of the convex hull for most input sets by dividing it into subsets and using heaps. Through High-Performance Computation, we seek to design and implement a parallel algorithm version, to speed up the calculation of Convex Hull, improving efficiency in the computation time. We evaluate its efficiency contrasting existing solutions with the proposed solution using performance measures.

Keywords: Computational Geometry; Convex Hull; Parallel Algorithm; Speedup

1. Introducción

El problema de calcular el convex hull de un conjunto de puntos es fundamental en matemáticas, geometría computacional, gráfica computacional y modelamiento de formas. El cálculo de convex hull es usado frecuentemente para detección de colisiones, cálculo de interferencias, análisis de formas, reconocimiento de patrones, estadísticas, sistemas de información geográfica, etc.

La convexidad es una propiedad geométrica muy importante. Un conjunto geométrico es convexo si para todo par de puntos pertenecientes a este, el segmento de línea recta que los une está completamente dentro del conjunto [1]. Uno de los primeros problemas identificados en el campo de la geometría computacional es el de calcular la forma convexa más pequeña, llamada convex hull, que encierra un conjunto de puntos [2]. Dado un conjunto de puntos en un espacio en d dimensiones, el convex hull es su mínimo subconjunto convexo que contiene todos los puntos [3], tal como se muestra en la Figura 1.

Definición **1.** *Dado un conjunto de puntos finito S, se dice que CH(S) es la envolvente o cierre convexo de S si:*

- *CH(S) es convexo.*
- *S está contenido en CH(S).*
- *CH(S) es el menor conjunto convexo que contiene a S.*

*Supported by Universidad Austral de Chile

[1]The complexities described in this document are based in the RAM model, which can perform any arithmetic operation on $\omega = \Omega(\log n)$-bit integers in constant time.

Para un conjunto de n puntos $\mathbf{P}$, el convex hull se define como la intersección de todos los conjuntos convexos que contienen a $\mathbf{P}$. El problema de encontrar el convex hull $CH(\mathbf{P})$, consiste en hallar los puntos de $\mathbf{P}$ que son vértices de $CH(\mathbf{P})$ y forman su frontera [24]. Esto se puede expresar matemáticamente con la expresión 1.

$$CH(\mathbf{P}) = \left\{ \sum_{i=1}^{n} \alpha_i P_i | P_i \in \mathbf{P}, \alpha_i \geq 0, \alpha_i \in \mathbb{R}, \sum_{i=1}^{n} \alpha_i = 1 \} \right\} \quad (1)$$

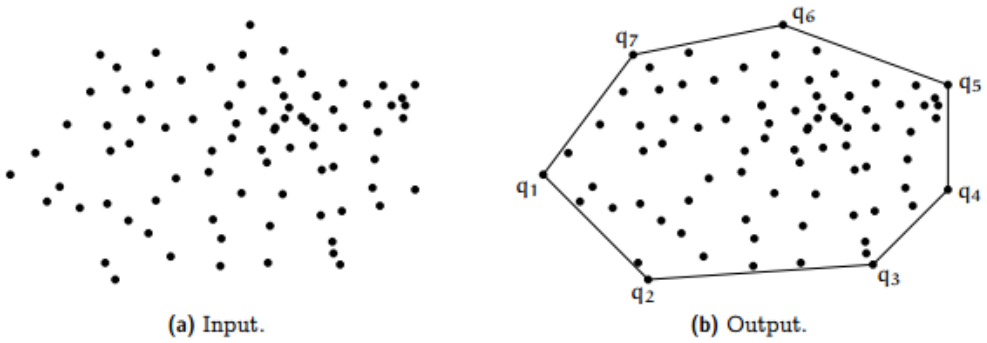

Figura 1: Cálculo del Convex Hull

Para el caso de 2 dimensiones ($2D$), el convex hull corresponde al polígono convexo más pequeño que contiene a todos los puntos del conjunto. En palabras simples, el convex hull, para el caso de $2D$, puede ser definido intuitivamente como "si S es un conjunto finito de puntos en el plano e imaginamos rodear el conjunto con una banda elástica; cuando se suelte el elástico este asumirá la forma del convex hull" [2].

El cálculo del convex hull suele realizarse directamente sobre los puntos de entrada. Sin embargo, en muchos casos – dependiendo de la instancia de entrada – es más conveniente realizar un filtrado de puntos, descartando elementos que son evidentemente innecesarios para el proceso final. Lo anterior da lugar a las siguientes dos etapas secuenciales:

1. **Filtrado**. Este paso consiste en tratar de descartar el máximo número de puntos que no pertenecen al convex hull, puntos que no son parte del polígono convexo que rodea el conjunto completo de puntos, o que están sobre alguno de los lados de este polígono y no son vértices del segmento de recta.

2. **Convex hull**. Recibe como entrada los puntos que sobrevivieron a la etapa previa y calcula el convex hull con alguno de los algoritmos tradicionales. De esta forma, el posterior cálculo del convex hull es realizado con un número reducido de puntos, siempre y cuando el paso anterior haya logrado descartar algunos puntos, haciendo que el proceso del algoritmo de cálculo sea más rápido y fácil. El cálculo en sí es realizado por algoritmos conocidos como Graham Scan [5], Jarvis March [6] u otro.

1.1. Algoritmos para el cálculo del convex hull

El problema de calcular el convex hull de un conjunto de puntos ha sido ampliamente estudiado en la geometría computacional. Se han propuesto muchos algoritmos teóricos óptimos para conjuntos de puntos de baja y alta dimensión. Este fue uno de los primeros problemas en el campo a ser estudiado desde el punto de vista de la complejidad computacional, determinando que su tiempo asintótico es $O(n \log n)$, para n puntos de entrada. Esto nos permite establecer la siguiente sentencia: **Dado un**

conjunto de n puntos. Es posible calcular su envolvente convexa en tiempo óptimo de $O(n \log n)$ [1][4]. Además, mediante un proceso lineal, es posible reducir el problema de ordenar n puntos a encontrar el convex hull en este mismo conjunto; confirmando que la envolvente convexa se puede calcular en un tiempo óptimo de $O(n \log n)$, ya que esta es la cota inferior al problema de ordenamiento.

El problema de encontrar el convex hull de un conjunto de puntos $\mathbf{P} = p_1, p_2, ..., p_n$ genera una secuencia de puntos como salida, entonces:

$$sort(n) \ \leq \ CH(P) + O(n)$$

$$CH(P) \ \geq \ sort(n) - O(n)$$

Como $sort(n)$ toma $O(n \log n)$, entonces cualquier algoritmo que resuelva $CH(n)$ toma al menos $O(n \log n)$ en el peor caso.

De hecho, soluciones algorítmicas eficientes fueron propuestas incluso antes de que se acuñara el término "geometría computacional". Esto, junto con su extenso análisis, refleja la importancia tanto teórica como práctica del problema [7].

Básicamente hay dos clases principales de algoritmos para encontrar el convex hull, los algoritmos de recorrido y los algoritmos incrementales [8].

- Algoritmos de recorrido: Primero encuentran algún nodo del convex hull y luego intentan identificar los vértices y lados restantes del polígono atravesándolo de alguna manera. Esta operación se conoce como pivoteo en el algoritmo símplex para programación lineal, y por esta razón también se conoce como algoritmos pivotantes. En esta categoría se encuentran el algoritmo gift wrapping de Chand and Kapur [9], algoritmo de Seidel [10] y el algoritmo de búsqueda inversa de Avis y Fukuda [11].

- Algoritmos incrementales: Calculan el vértice al intersectar secuencialmente los espacios intermedios definidos. Un símplex inicial se construye a partir de un subconjunto de $d + 1$ semiespacios y se calculan sus vértices y esqueleto. Los semi espacios adicionales se introducen secuencialmente y la descripción del vértice y el esqueleto se actualizan en cada etapa. Esencialmente, tal actualización equivale a identificar y eliminar todos los vértices que no están contenidos en el nuevo espacio medio, introduciendo nuevos vértices para todas las intersecciones entre los bordes y el hiperplano delimitador del nuevo espacio intermedio, y generar los nuevos bordes entre estos nuevos vértices [8]. Algoritmos de este tipo son el método de Seidel [12], el algoritmo aleatorio de Clarkson y Shor [13] y el algoritmo no aleatorizado de Chazelle [14]

En la Tabla 1 podemos ver los tiempos de los algoritmos más conocidos para dos dimensiones. Consideremos n = número de puntos en el conjunto de entrada, h = número de vértices del polígono de salida. Nótese que $h \leq n$, por lo que $nh \leq n^2$.

1.2. Aplicaciones del Convex Hull

Muchos problemas pueden ser reducidos a un convex hull, ejemplos de esto son la triangulación de Delaunay, los diagramas de Voronoi, diagramas de potencia, análisis de datos de espectrometría, generación de mallas, búsqueda de archivos, análisis de clusters, detección de colisiones, cristalografía, metalurgia, planificación urbana, cartografía, integración numérica, estadísticas y más (Aurenhammer [1991], citado por Barber [15])

.

Con el fin de hacer hincapié en la utilidad del cálculo del convex hull se describe brevemente en la Tabla 2 algunas de las aplicaciones más actuales de este.

Algoritmo	Tiempos	Autor
Fuerza Bruta	$O(n^4)$	Anónimo
Gift Wrapping	$O(nh)$	Chand & Kapur, 1970
Graham Scan	$O(n \log n)$	Graham, 1972
Jarvis March	$O(nh)$	Jarvis, 1973
QuickHull	$O(nh)$	Eddy, 1977, Bykat, 1978
Divide-and-Conquer	$O(n \log n)$	Preparata & Hong, 1977
Monotone Chain	$O(n \log n)$	Andrew, 1979
Incremental	$O(n \log n)$	Kallay, 1984
Marriage-before-Conquest	$O(n \log n)$	Kirkpatrick & Seidel, 1986

Tabla 1: Tiempos de algoritmos para el cálculo del convex hull en $2D$.

1.3. Algoritmo Heaphull

El trabajo de Ferrada y otros [25] está basado en el desarrollo de una técnica de optimización que reduce el costo computacional para construir el convex hull. Este método preprocesa el conjunto de puntos de entrada, filtrando todos los puntos dentro de un polígono de ocho vértices en tiempo $O(n)$ y devuelve un conjunto reducido de puntos candidatos para encontrar el convex hull, los cuales se encuentran semi ordenados y distribuidos en cuatro colas de prioridad.

Los resultados experimentales muestran que en el peor de los casos (cuando todos los puntos se encuentran sobre una circunferencia) un pequeño desplazamiento radial aleatorio de los puntos hace de este método el más rápido. Además, al aumentar la magnitud de este desplazamiento, el rendimiento del método propuesto escala a un ritmo más rápido que las otras implementaciones testeadas. En términos de eficiencia de memoria, esta implementación logra usar de 3 a 6 veces menos memoria que otros métodos.

Este algoritmo es esencialmente una técnica de filtrado que se conforma de cuatro etapas relevantes, (1) construcción de un octágono con los puntos extremos, (2) filtrado de puntos y agrupación en colas de prioridad, (3) ordenamiento parcial y (4) cálculo del convex hull. Las primeras tres etapas toman tiempo $O(n)$, mientras que la etapa de cálculo del convex hull toma $O(n' \log n')$, donde n' es el tamaño del conjunto filtrado $P' \subseteq P$. Se puede ver el algoritmo en 1.

Algorithm 1 heaphull. To compute the convex hull $CH[1..h]$ in $2D$

Require: n floating points, in $2D$, stored in an array $P[1..n]$.
Ensure: an integer array $CH[1..h]$ with the h points of the hull in counter-clockwise order.

1: **procedure** HEAPHULL(P, n)
2: $E \leftarrow$ FINDEXTREMES(P,n) ▷ find the 8 extreme points in $O(n)$ time
3: $n_1 = n_2 = n_3 = n_4 = 0$ ▷ counter for the 4 priority queues Q_i
4: **for** $j = 1$ `to` n **do**
5: **if** $P[j]$ is outside the convex octagon $CP(E)$ **then**
6: $i \leftarrow$ FINDQUEUE(P, n, j) ▷ find Q_i for $P[i]$ ($O(1)$ time)
7: store j in the priority queue Q_i
8: $n_i \leftarrow n_i + 1$
9: $CH \leftarrow \emptyset$
10: **for** $i = 1$ `to` 4 **do**
11: $CH \leftarrow CH \cup$ HULL(Q_i, n_i) ▷ CH for points of Q_i in $O(n_i \log n_i)$ time
12: **return** CH ▷ the convex hull $CH[1..h]$, with $h \leq \sum_{i=1}^{4} n_i = n' \leq n$

Área	Descripción
Química	Se obtienen diagramas de fase en sistemas multicomponentes a partir de la generación de una malla de puntos, calculando la energía libre de Gibbs de cada uno y construyendo el convex hull de aquellos con mínima energía libre de Gibbs en cada malla. [16]
Máquinas de soporte vectorial (SVM)	Se define el convex hull flexible y se emplea para solucionar problemas de clasificación en entrenamiento de SVM y se extiende a problemas de clasificación multiclase [17].
Obtención de estructuras cristalinas teóricas	Se combinan las relaciones entalpia-presión con el convex hull para identificar las fases más estables a presión cero. [18]
Definición de contornos de siluetas	Reconocimiento de actividades de personas a través de cámaras. Se utiliza el convex hull para encontrar el polígono más pequeño que envuelve una silueta humana determinada [19].
Aprendizaje supervisado	Aplicación del método de Jarvis para identificación de convex hull con el fin de entrenar SVM en problemas de clasificación [20].
Selección de modelos	Por medio de convex hull se identifican los modelos sobre la frontera del mismo que afectan la bondad de ajuste de forma significativa [21].
Análisis envolvente de datos	Se utilizan convex hull para caracterizar completamente las caras donde se encuentran las unidades eficientes. De esta forma se identifican pares de referencia que pueden permanecer ocultos con los métodos de solución tradicionales [22]
Análisis de formas	Tiene aplicaciones en campos como el reconocimiento de matrículas de automóviles [23]

Tabla 2: Nuevas aplicaciones del convex hull [24].

1.4. Uso de programación paralela y GPU para acelerar el cálculo del Convex Hull

Existen varios métodos para acelerar un programa y uno de ellos es la programación paralela, esta consiste en dividir una tarea en varias tareas más pequeñas e independientes entre sí, y ejecutarlas simultáneamente. En un escenario ideal, el tiempo dedicado a una tarea podría reducirse proporcionalmente en relación con el número de trabajos concurrentes dedicados a ella. Para esto, actualmente podemos utilizar la CPU y la GPU, obteniendo mejores resultados en cada una según la naturaleza de los datos y del algoritmo a paralelizar.

Las GPU (Graphic Process Unit) se han utilizado para proporcionar soluciones eficientes para diversas aplicaciones, como la simulación de partículas, el modelado molecular y el procesamiento de imágenes. Es una arquitectura masiva de multi-threaded que incluye cientos de cores (elementos de procesamiento), donde cada core es un procesador pipeline multi-etapa. Los cores se agrupan para generar un multiprocesador simétrico (SM) de instrucción única de datos múltiples (SIMD por sus siglas en inglés), es decir, los núcleos en un SM ejecutan la misma instrucción en diferentes elementos de datos. Cada core tiene su propio conjunto de registros y memoria local limitada, y los cores dentro del mismo SM tienen memoria compartida limitada.

Para medir la mejora lograda se utiliza el speedup, este es definido como el proceso realizado para mejorar el rendimiento de un sistema que procesa un problema determinado. En otras palabras, es la mejora en la velocidad de ejecución de una tarea realizada en dos arquitecturas similares con diferentes

recursos, la cual se denota con la siguiente expresión matemática:

$$S = \frac{L_{old}}{L_{new}}$$

donde L_{old} corresponde al tiempo de ejecución del programa antiguo y L_{new} es el tiempo de ejecución del nuevo programa. Estos tiempos pueden ser medidos con respecto a las observaciones mínima, máxima, la media o la mediana de estas.

En este artículo buscamos alcanzar un speedup significativo del algoritmo de filtrado, es decir un S grande y mayor que 1, mediante paralelización en GPU, teniendo en cuenta que el algoritmo a utilizar logra los mejores tiempos para el cálculo del convex hull, debido a su uso eficiente de las estructuras de datos y su impacto en la etapa de filtrado.

Para paralelismo en CPU es ampliamente utilizado OpenMP, que es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influyen el comportamiento en tiempo de ejecución.

Para paralelismo en GPU nos enfocamos en CUDA. Esta plataforma de computo en paralelo, incluye un compilador y un conjunto de herramientas de desarrollo creadas por nVidia que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPU de nVidia. CUDA explota las ventajas de las GPU frente a las CPU utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos. Por ello, si una aplicación está diseñada utilizando numerosos hilos que realizan tareas independientes (que es lo que hacen las GPU al procesar gráficos), una GPU podrá ofrecer un mayor rendimiento.

Este trabajo tiene como objetivo de investigación responder a las siguientes preguntas:

- ¿Cuánto mejora el desempeño de la etapa de filtrado al paralelizar en GPU el algoritmo heaphull?
- Al paralelizar heaphull ¿Se logra que el algoritmo sea más eficiente que otros que ya usan paralelismo?

El resto de este artículo se divide en 5 secciones. En la primera sección realizamos una revisión del trabajo relacionado, considerando especialmente los algoritmos que realizan un esfuerzo por paralelizar la etapa de filtrado, para esto los clasificamos en principales algoritmos secuenciales tradicionales, algoritmos secuenciales que explotan el filtrado previo, algoritmos paralelos de filtrado en CPU y algoritmos paralelos de filtrado en GPU. En la segunda sección definimos en detalle la metodología de investigación implementada. En la sección 3 presentamos el diseño del algoritmo paralelo en CPU y en GPU. En la sección 4 revisamos las métricas obtenidas y comparamos las medidas de eficiencia del algoritmo (speedup, eficiencia y comparación entre CPU y GPU). Finalmente en la sección 5 presentamos las conclusiones.

2. Trabajo Relacionado

Entre 1972 y 1989 fueron publicados 16 algoritmos lineales que encuentran el convex hull y 7 de ellos fueron encontrados incorrectos posteriormente. También hay muchos métodos prácticos conocidos e implementaciones de software robustas que abordan este problema de manera secuencial, desde 1970 se han desarrollado varios algoritmos clásicos [26], como Graham scan [5], Gift wrapping [6], Incremental method [39], Divide-and-Conquer [38], Monotone chain [40], and Quick-Hull [15].

2.1. Etapa de filtrado

La atención principal al elegir un algoritmo para el cálculo del convex hull se presta al tiempo de ejecución. Realizar un filtrado eficiente del conjunto de puntos puede tener un gran impacto en el tiempo final de cálculo. Vyšniauskaitė [44] en 2006 propuso una idea de la filtración a priori de puntos, presentado dos nuevos algoritmos basados en la búsqueda de puntos extremos y en la subdivisión del conjunto de puntos en arreglos más pequeños.

También Sharif [45] el 2011 propone un método híbrido para calcular el convex hull, basado en dos algoritmos ya existentes, es decir, Quickhull y GrahamScan, intentando eliminar las deficiencias en las dos técnicas mencionadas anteriormente usando el primero para realizar un filtrado y realizando el cálculo final con el segundo de estos.

2.2. Algoritmos Paralelos

Encontrar el convex hull de grandes conjuntos de puntos es en general costoso desde el punto de vista computacional. Una estrategia efectiva para tratar este problema es calcular el convex hull en paralelo [26]. Para mejorar la eficiencia computacional de este cálculo, se han hecho algunas contribuciones valiosas rediseñando e implementando algoritmos de convex hull secuenciales en paralelo, explotando el potencial de cálculo masivo de las GPU. La mayoría de estas implementaciones se desarrollan en base al algoritmo QuickHull [26].

2.2.1. Paralelismo en CPU

Como se muestra la publicación de Chen y otros [27] el convex hull puede ser calculado en PRAM (Máquina de acceso aleatorio paralelo).

Ya en 1987 Goodrich [34] desarrolló un algoritmo paralelo para encontrar el convex hull que se ejecuta en tiempo $O(log\ n)$ utilizando $O(n/log\ n)$ procesadores en el modelo computacional CREW (concurrent read and exclusive write) PRAM, que es óptimo. Una de las técnicas que utilizan para alcanzar estos límites óptimos es el uso de una estructura de datos paralela que llaman "hull tree".

Miller [31] en 1988 presentó algoritmos paralelos para identificar los puntos extremos del convex hull, concentrándose en el desarrollo de estos en tiempo polilogarítmico para una variedad de máquinas paralelas y analizándolos usando notación O. Asimismo, Liu [30] en 2015 hace una revisión completa de los algoritmos paralelos existentes a la fecha, categorizándolos según su arquitectura y considerando sus tiempos de ejecución en notación O y el número de procesadores necesarios para esto.

Berkman [28] en 1996 desarrolló un algoritmo paralelo para encontrar el convex hull en tiempo $O(log\ log\ n)$ usando $n/log\ log\ n$ procesadores en una arquitectura PRAM CRCW (Concurrent read concurrent write) común. Para romper la barrera de tiempo $\Omega(log\ n/log\ log\ n)$ requerida para generar el convex hull, introduce una estructura de datos, un árbol balanceado de altura doblemente logarítmica y usa este para representarlo. El algoritmo demuestra el poder del "paradigma doblemente logarítmico de divide y vencerás".

Nakagawa [35] presentó una implementación de algoritmo paralelo simple para calcular el convex hull y evaluar el rendimiento en los procesadores de cuatro núcleos duales, logrando un factor de aceleración de aproximadamente 7 utilizando 8 procesadores. Como el factor de aceleración de más de 8 no es posible, su implementación paralela para calcular el convex hull es casi óptima.

Waghmare [33] presenta un algoritmo de convex hull usando clustering K-means, en el que los puntos en 2D se agrupan en clusters diferentes y luego se calculan los convex hull para cada uno de estos. El algoritmo se implementa en modo MPI, OpenMP e híbrido. Los resultados indican que el

enfoque híbrido supera al enfoque MPI y OpenMP por separado.

2.2.2. Paralelismo en GPU

El trabajo relacionado en computación paralela para el cálculo del convex hull en GPU muestra que se puede alcanzar mejor velocidad y rendimiento, como se evidencia en el trabajo de Tang et al. [41], y Jiayin et al [26]. Estos algoritmos basados en GPU trabajan sobre la optimización del filtrado de puntos, realizando la etapa posterior de cálculo en forma secuencial.

Srungarapu et al. [50] propuso en 2011 una implementación optimizada en GPU para encontrar el convex hull para conjuntos de puntos de dos dimensiones. Su implementación trata de minimizar el impacto de los patrones de acceso irregular a los datos, logrando un speedup de hasta 14 veces sobre la implementación estándar secuencial en CPU.

En 2011 Jurkiewicz y Danilewski [49] presentaron una propuesta de implementación de algoritmo de cálculo del convex hull implementada en su totalidad en GPU, sin filtrado y basado en 3 algoritmos secuenciales distintos, con pruebas experimentales con conjuntos de entrada de 10^7 puntos, obteniendo resultados en reducción de tiempo de hasta 100 veces para el máximo de puntos al compararse con una librería que utiliza algoritmos similares, pero que su desempeño se encuentra muy por debajo de la media.

Mei G. [48] en 2016 propuso una versión paralela en CUDA (GPU) para el cálculo de convex hull para dos [47] y tres [46] dimensiones, alcanzando aceleraciones de aproximadamente 4x en promedio y $5x \sim 6x$ en los mejores casos, la cual llama CudaChain. Realizando solo una mejora de filtrado, afirma que el 95 % de puntos de entrada se pueden descartar en la mayoría de las pruebas experimentales. Anterior a esto, usó GPUs para mejorar la fase de filtrado en la búsqueda de puntos extremos sobre datos sintéticos y con el uso de la biblioteca qhull (implementación de Quickhull en C++), la idea básica era descartar los puntos que se ubican dentro de un polígono convexo formado por 16 extremos [51].

Por otro lado, Stein [32] en 2012 hizo una versión paralela para el convex hull en 3 dimensiones que se basa en el enfoque QuickHull y comienza construyendo un tetraedro inicial utilizando cuatro puntos extremos, descarta los puntos internos y distribuye los puntos externos a las cuatro caras y continúa luego iterativamente. En su trabajo, Stein afirma que su implementación superó al Qhull basado en CPU en 30 veces para 10 millones de puntos y 40 veces para 20 millones de puntos.

Dunlaing [52] en su artículo de 2012 presenta una implementación CUDA del algoritmo de convex hull PRAM de Wagener en R^2, con varias restricciones sobre la entrada de datos y con un análisis teórico del speedup.

En 2020 Masnadi [53] presenta ConcurrentHull, una nueva técnica basada en el filtrado de puntos para conjuntos de datos 2D y 3D. Su implementación, que es una combinación de filtrado, divide y vencerás, y computación paralela, permite ser empleada en un entorno de computación distribuida. Su algoritmo tiene una versión para CPU y otra para GPU (CUDA). Los resultados muestran que tiene una ganancia de rendimiento con grandes tamaños de datos de entrada y tiene la ventaja de poder manejar grandes conjuntos de datos.

Si bien las soluciones propuestas informan un speedup importante para las condiciones establecidas para las pruebas, nuestro trabajo busca no solo mejorar los tiempos de ejecución si no ponemos especial atención en el porcentaje de filtrado a obtener en la fase paralela, lo cual tiene el mayor impacto en el tiempo de cálculo posterior. Nuestro trabajo compara las dos soluciones más actuales con la nuestra, comparando la solución en el peor caso y en el caso promedio.

3. Metodología de Investigación

Para responder a las preguntas de investigación, en nuestra metodología nos proponemos diseñar algoritmos de búsqueda de puntos extremos y filtrado de la entrada en paralelo, tanto en CPU como en GPU, los cuales serán implementados. Luego de comparar ambas estrategias experimentalmente, seleccionaremos la mejor de ellas, que por hipótesis creemos que debe ser paralelismo en GPU, para finalizar construyendo una implementación que calcule el convex hull en 2D, descartando el método de filtrado más lento. Para el análisis de los resultados obtenidos, comparamos los tiempos de ejecución, el speedup y la eficiencia de los algoritmos implementados en función del tiempo de ejecución y contrastamos con las soluciones más nuevas.

Se realiza una versión paralela en CPU con OpenMP y una versión en GPU con CUDA. Esto para determinar empíricamente que la versión en GPU arroja mejores tiempos que la versión en CPU.

Para el desarrollo de esta solución paralelizamos la fase de filtrado en OpenMP para CPU y en CUDA para GPU, para luego calcular el convex hull con los puntos restantes en forma secuencial en CPU, comparando los speedup obtenidos.
Los pasos a seguir son:

1. Revisión y reconocimiento de secciones paralelizables y/o otras técnicas de HPC que pueden ser aplicadas al paralelizar nuestro algoritmo.
   a) Profiling del filtrado del algoritmo heaphull
   b) Revisión de bloques de código con bucles (altamente paralelizables)
2. Construcción de algoritmo paralelo en CPU usando OpenMP en lenguaje C++ y optimización de este.
3. Construcción del algoritmo paralelo en CUDA para GPU y optimización de este.
4. Cálculo de porcentajes de filtrado, medidas de speedup y eficiencia para evaluar y analizar condiciones de borde de la versión codificada.
5. Comparación de resultados con soluciones existentes (ver trabajo relacionado) más significativas en cuanto a speedup y más recientes, tanto en filtrado como en el proceso completo de encontrar el convex hull.

Usamos datos sintéticos generados a partir de distribución normal (que es el caso más usual en el cálculo del Convex Hull). Incluimos un segundo caso donde el dataset de entrada corresponde a nuestro peor caso para el cálculo del Convex Hull, esto es, donde todos los puntos se encuentran sobre una misma circunferencia (o muy cerca de ella) y por tanto no hay manera posible de descartar puntos en la etapa preliminar de filtrado.

4. Paralelización del Algoritmo Heaphull

La implementación del algoritmo se realizó en el lenguaje de programación C++, utilizando OpenMP para CPU y CUDA para GPU. El primer paso es encontrar los cuatro puntos extremos en las direcciones norte, sur, este y oeste y formar el cuadrilátero que definen.

Ahora, tenemos puntos distribuidos en cuatro regiones no conectadas entre sí, trataremos cada una de estas regiones de manera independiente. En cada uno de ellos encontramos el punto más cercano a la esquina conformada por las coordenadas cada par de puntos puntos, obteniendo así cuatro puntos nuevos y un octágono que dividirá los puntos en ocho regiones que tratamos individualmente siguiendo la misma regla anterior.

Para acelerar la ejecución, buscamos segmentos paralelizables, identificando los bloques de ejecución asociados con las repeticiones. Una mejora en estas dimensiones puede influir significativamente en el tiempo de ejecución final.

Para la creación de los puntos para las pruebas, se utiliza una distribución normal, generando $2x10^8$ puntos con $\mu = 0{,}5$ y desviación estándar = 0,1, estableciendo esta preferencia en el momento de ejecutar el programa.

Se realizan dos versiones paralelas en CPU, una con la reducción provista por OpenMP para mínimos y máximos y otra usando la técnica de Divide y Vencerás, siendo esta última la con mejores resultados experimentales y con la que se obtiene un speedup de casi $3x$ para el filtrado. Esta última se contrasta con la versión en GPU, la cual arroja resultados iniciales con tiempos entre 8 y 11 veces menores a la versión secuencial; debido a nuestros resultados experimentales nos quedamos con la GPU para realizar nuestra implementación paralela.

4.1. Paralelización del Filtrado de Puntos en GPU

Para la versión en GPU se realiza la reducción de mínimos y máximos, así como la reducción de las distancias de manhattan usando el enfoque de Shuffle Warp Reduce.

El algoritmo sigue los pasos de la versión tradicional de heaphull para el filtrado de puntos, codificando una versión en paralelo de estos mismos. A diferencia de la versión tradicional de heaphull, que implementa la búsqueda de los 8 puntos en cada paso del bucle, en la versión GPU se realiza una reducción para encontrar los primeros 4 puntos extremos para luego en un segundo paso se realiza los cálculos para todos los puntos restantes para encontrar los siguientes 4 puntos aprovechando la alta velocidad de cálculo de la GPU (no hay una diferencia significativa en tiempo al realizar este paso). Esto es debido a que la búsqueda de los puntos secundarios requiere de los primeros puntos, lo cual equivale a un paso secuencial.

En el segundo paso usamos distancias de Manhattan, es decir, para cada esquina $c_i, 1 \leq i \geq 4$, encontramos el punto en P que minimiza la suma de las distancias vertical y horizontal a este esquina. En algunos casos la distancia de Manhattan no proporcionará el punto más cercano a la esquina en forma exacta como al calcular la distancia euclidiana, aunque esto tienden a ocurrir con baja frecuencia, y cuando ocurre, se puede ver que el punto elegido todavía se encuentra muy cerca del más cercano usando distancia Euclidiana [25]. La ventaja es que al usar la distancia de Manhattan realizamos un cálculo más simple, no requiriendo raíces cuadradas y manteniendo un tiempo de cálculo constante.

En el paso siguiente realizamos el descarte de los puntos que no sobreviven al filtrado y creamos las colas. Para la creación de las 4 colas usamos un arreglo, donde asignamos el número de cola (1, 2, 3 o 4) a la posición donde se encuentra un punto que si pertenece a la cola respectiva y un 0 cuando ha sido descartado.

Luego, de vuelta en la CPU, se crean las 4 colas con los respectivos índices guardados para ser entregadas a la sección correspondiente del algoritmo secuencial heaphull.

El Algoritmo 2 describe el proceso de esta primera etapa.

Para el desarrollo de esta versión se debe tener en cuenta las limitaciones de la programación en GPU, como el hecho de no poder realizar la búsqueda de extremos y el filtrado en un mismo paso, pues se requieren pasos secuenciales (en vez de esto se utilizan tres kernel distintos). También la asignación de memoria dinámica no es posible realizarla como en la versión CPU, lo cual dificulta el uso eficiente de memoria.

Algorithm 2 GPUfilter. Para calcular el octágono de filtrado de $P[1..n]$ en $2D$

Require: n valores de punto flotante, en $2D$, almacenados en un arreglo $P[1..n]$.
Ensure: un arreglo de enteros $F[1.,8]$ con los índices del octágono de filtrado de P. Un arreglo de char con la cola correspondiente Qi.

```
1: procedure GPUFILTER(P, n)
2:     E ← FINDEXTREMES(P,n)                    ▷ encuentra los 8 puntos en tiempo O(n)
3:     Sea Q[] un arreglo de n enteros
4:     for j = 1 to n do
5:         if P[j] está fuera del octágono CP(E) then
6:             i ← FINDQUEUE(P, n, j)           ▷ encuentra Q_i para P[i] (tiempo O(1))
7:             Q[j] = i
8:     return Q
```

5. Análisis de Resultados

A fin de comparar experimentalmente el desempeño de nuestra solución paralela en GPU, incluimos como baseline las implementaciones de: el heaphull tradicional [25], el CudaChain [48] y el ConcurrentHull [53]; este último permite el manejo de una mayor cantidad de puntos y consigue mejores resultados en esas condiciones.

Para las pruebas se utilizó un computador con Procesador Intel®Core™i5-8300H CPU (2.30GHz × 8), 8 GB de memoria RAM, tarjeta gráfica NVIDIA GeForce GTX 1050 Ti, CUDA Version 11.6.112 en Ubuntu 20.04.4 LTS.

5.1. Análisis de los datos (Tiempos de Ejecución)

Para cada uno, creamos de 10^4 hasta 10^8 puntos (el valor máximo se elige por asuntos de tiempo de ejecución y capacidad de memoria de la GPU). Corremos 100 veces cada prueba y tomamos los tiempos promedio de ejecución en cada una.

Los tiempos promedio de búsqueda de puntos extremos para las pruebas experimentales en las versiones de heaphull secuencial, paralelo en CPU y paralelo en GPU se pueden ver en la tabla 3 y en el gráfico de la Figura 2. Debido a los resultados experimentales no se seguirá considerando la versión paralela en CPU, ya que hemos comprobado que la versión en GPU logra un speedup de al menos 3x sobre esta, por lo que solo se usará esta implementación.

Puntos	Tiempo CPU	Tiempo CPU Paralelo	Tiempo GPU
10^4	0,0460	0,1000	0,0433
10^5	0,4600	0,4000	0,0814
10^6	5,0120	2,4000	0,8505
10^7	56,8020	18,0000	6,9785
10^8	793,7420	289,1300	67,0598

Tabla 3: Tiempo promedio de búsqueda de puntos extremos en ms.

5.1.1. Desempeño en el caso Promedio (Distribución Normal)

En la tabla 4 se pueden ver los tiempos de cálculo de la versión secuencial de heaphull, CudaChain, ConcurrentHull y nuestra implementación en GPU, en milisegundos para 10^4 a 10^8 puntos. Nuestra solución es la que obtiene mejores tiempos en los casos con 10^5 puntos o más. En la Figura 3 podemos ver

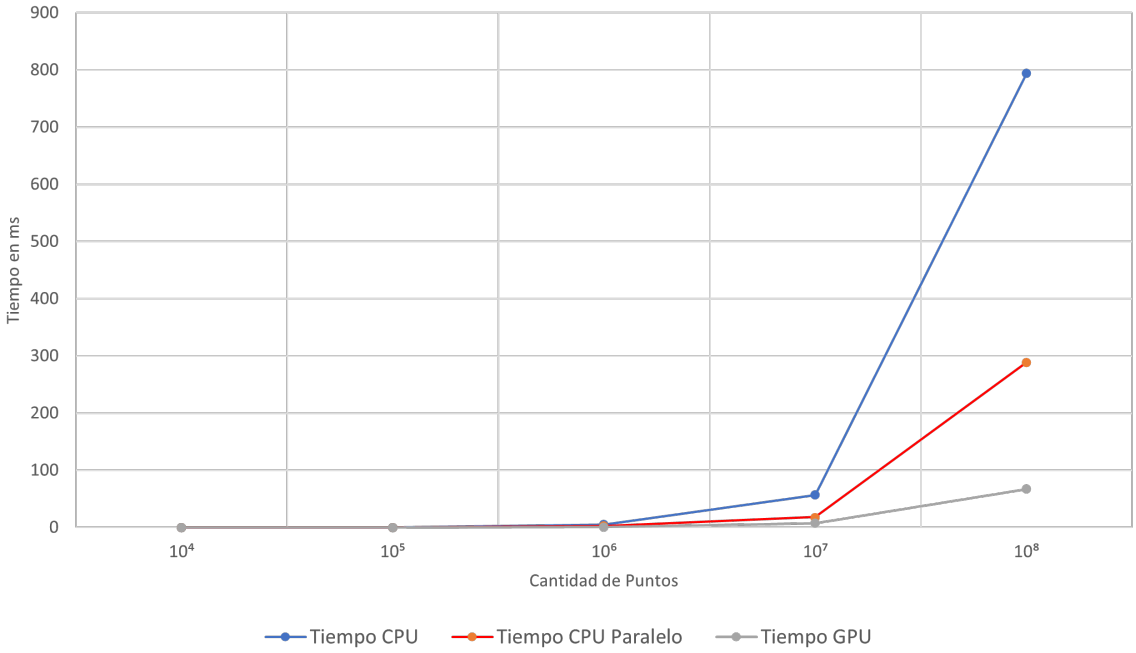

Figura 2: Tiempos promedio de filtrado para las versiones secuencial, paralela en CPU y paralela en GPU.

el gráfico de los tiempos en milisegundos para el cálculo de convex hull en las cuatro implementaciones anteriores.

Puntos	**Heaphull**	**CudaChain**	**ConcurrentHull**	**GPU heaphull**
10^4	0,1201	6,4803	882,0300	0,1575
10^5	1,2014	12,0930	902,0700	0,9940
10^6	12,0824	24,1600	956,3200	7,0058
10^7	139,9730	224,2051	1211,6300	35,1825
10^8	2054,5416	1517,1612	2201,5500	464,8182

Tabla 4: Tiempo promedio cómputo del convex hull en el caso promedio.

El Speedup obtenido se puede ver en la tabla 5 y en la figura 5, pudiendo identificarse que al trabajar con 10^8 puntos se logra una disminución de tiempo de hasta $4,4$ veces respecto al algoritmo secuencial de heaphull y de 3,2 respecto a CudaChain, el cual ya logra un speedup de hasta 6x comparado con la librería de Quickhull (Qhull en C++). No se considera en el gráfico el speedup frente a ConcurrentHull de menos de 10^7 puntos para poder mantener una escala apreciable.

En relación al porcentaje de filtrado de puntos, heaphull secuencial y la versión en GPU de este logran porcentajes para puntos provenientes de una distribución normal sobre el 99,99 % en promedio (igual porcentaje para mismos puntos de entrada). En el caso de CudaChain logra un filtrado de 98,86 % en promedio en el mejor caso (10^8 puntos de entrada) y en el peor caso para este (10^4 puntos de entrada) llega a un 91,6 %, mientras que heaphull en sus dos versiones logra un 99,87 % para esa misma cantidad de puntos.

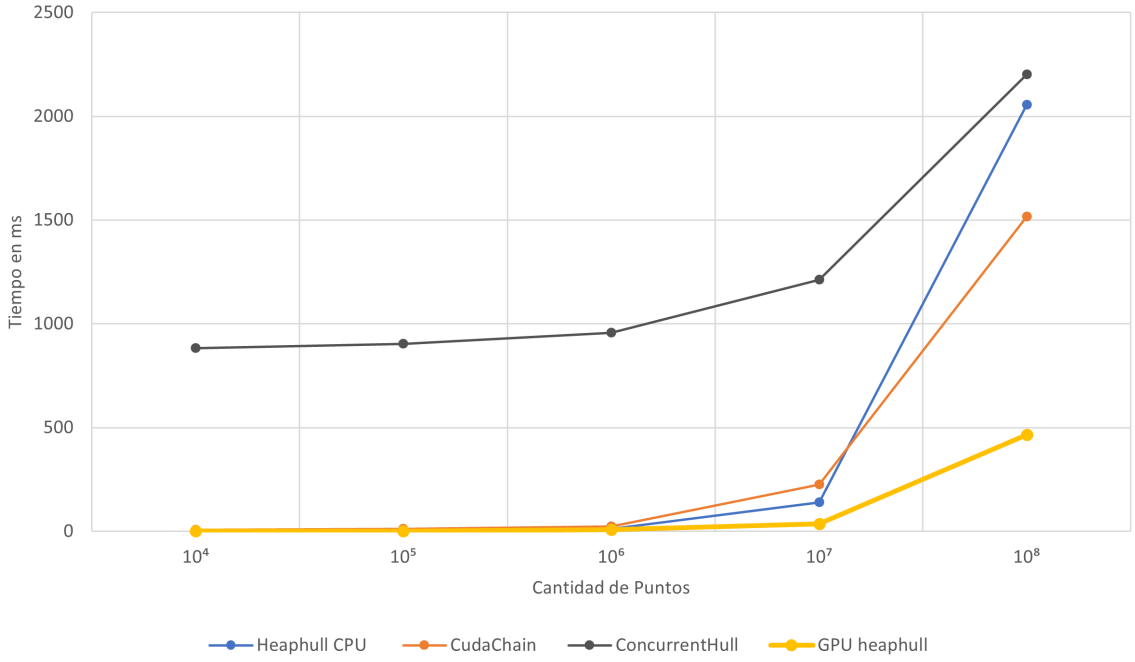

Figura 3: Tiempos para el cómputo del convex hull en el caso promedio

Puntos	Heaphull	CudaChain	ConcurrentHull
10^4	0,7625	41,1448	5600, 1905
10^5	1,2087	12,1660	907,5151
10^6	1,7246	3,4486	136,5040
10^7	3,9785	3,5303	34,4384
10^8	4,4201	3,2640	4,7364

Tabla 5: Speedup solución desarrollada sobre heaphull, CudaChain y ConcurrentHull, caso promedio.

5.1.2. Desempeño en el Peor Caso (Puntos en la Circunferencia)

Para el peor caso (los puntos sobre la circunferencia) con 10^7 puntos nuestra implementación paralela en GPU demora en encontrar el convex hull 6104, 58 ms, sin embargo con 2 % de distorsión el tiempo disminueye a 4973, 3 ms; el primero representa un speedup de 0,96 respecto del algoritmo secuencial de heaphull, sin embargo en el segundo caso se obtiene un speedup de 1,41. Para 10^8 puntos el speedup es de 1, 04 en el peor caso. En la tabla 6 se encuentran los tiempos de ejecución en el peor caso. En la figura 5 se puede ver el gráfico con los tiempos de para cada versión.

Con una distorsión de sólo 2 % los porcentajes de filtrado para 10^8 puntos se mantienen en 10.50 % en promedio, con tiempos de ejecución del algoritmo completo similares al caso promedio.

La versión codificada del algoritmo tiene un comportamiento con un speedup no relevante cuando se trata de cantidades de puntos inferiores a los 10^7 puntos. Además, cuando se trata del peor caso no hay una mejora significativa respecto a la versión secuencial.

En general, los mejores tiempos se logran al trabajar con 10^7 y 10^8 puntos, en estos casos el speedup es mayor a 4x.

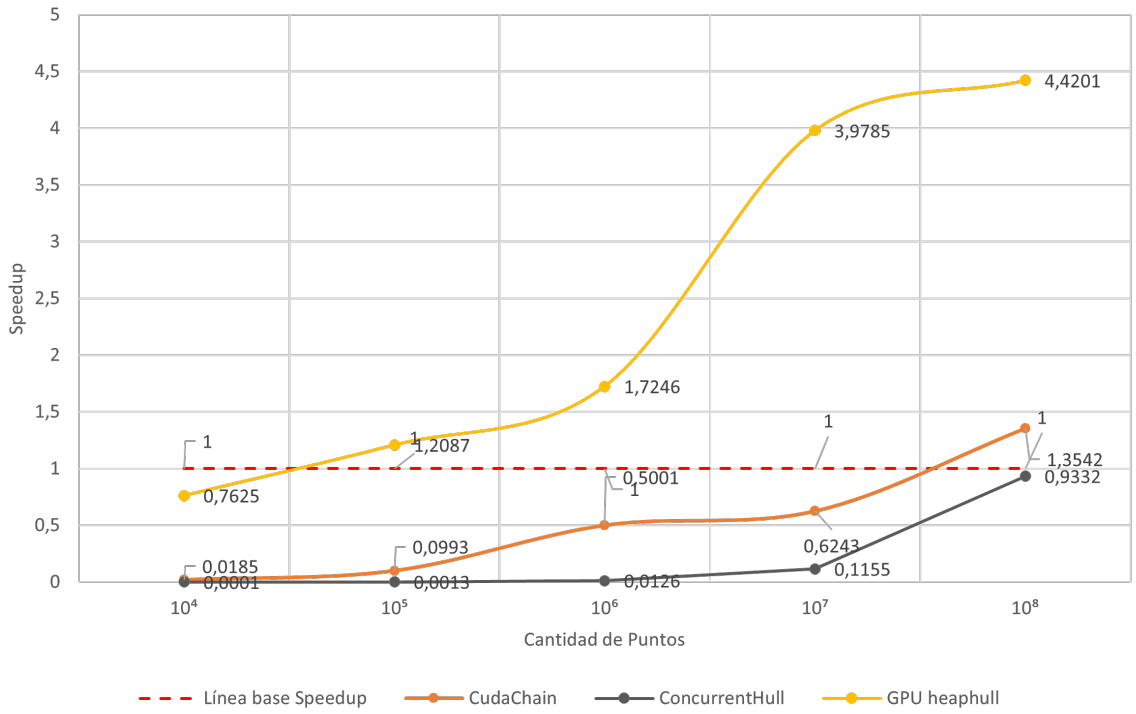

Figura 4: Speedup sobre heaphull secuencial de CudaChain, ConcurrentHull e implementación GPU Heaphull para el caso promedio.

Puntos	Heaphull	CudaChain	ConcurrentHull	GPU Heaphull
10^4	1,0153	86,7300	90,0223	1,6400
10^5	15,0671	112,5680	96,3250	23,6950
10^6	241,7010	445,2650	1256,3600	427,0750
10^7	5861,8900	6459,5251	6201,5522	6104,5800
10^8	95278,1000	121426,0021	123045,2000	91559,8000

Tabla 6: Tiempos para el cómputo del convex hull en el peor caso.

6. Discusión y Conclusiones

Nuestro trabajo presenta un implementación paralela en GPU basada en el algoritmo heaphull que mejora el tiempo de filtrado y el tiempo total de cálculo del convex hull manteniendo las premisas del algoritmo tradicional, haciendo un uso eficiente de memoria y conservando un tiempo total $O(n \log n)$ para el caso de $2D$. El algoritmo se basa en realizar un filtrado con un octágono que permite descartar todos los puntos que se encuentran en su interior y lograr en el caso promedio filtrar el $99,99\,\%$ de los puntos del conjunto de entrada, quedando una cantidad muy reducida de puntos para el cálculo del convex hull y conservando el eficiente filtrado del algoritmo secuencial de heaphull.

El mayor impacto en desempeño se logra al trabajar con grandes volúmenes de puntos de entrada (sobre 10^7), siendo un aporte importante en esta arista para el cálculo del convex hull.

En comparación con otras implementaciones en GPU se obtienen mejores tiempos totales, considerando que se está trabajando sobre un algoritmo que ya resulta ser el más rápido secuencial y en $2D$, esto es un importante aporte al desarrollo futuro de implementaciones que puedan requerir el cálculo en tiempo real de grandes volúmenes de puntos.

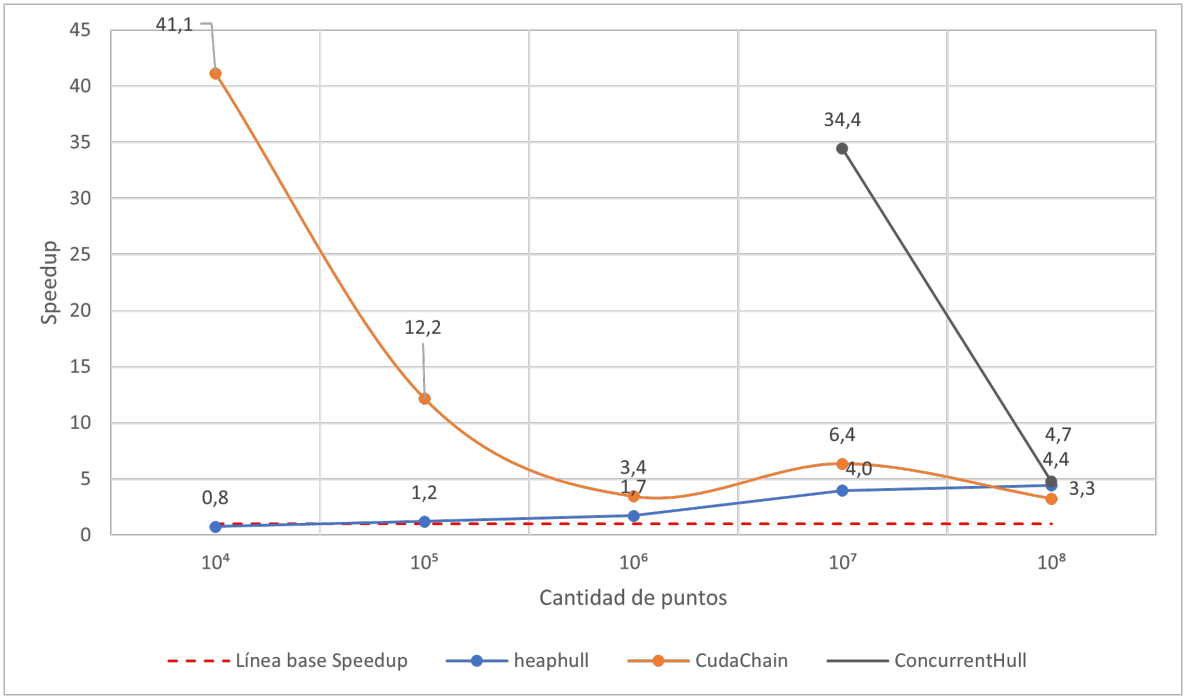

Figura 5: Speedup implementación desarrollada sobre heaphull, CudaChain y ConcurrentHull, caso promedio.

Dentro de las limitaciones de la solución propuesta se encuentra el uso de memoria de GPU, el cual se ve aumentado por la necesidad de crear arreglos para almacenar los índices que son resultado de las reducciones para poder conservar el arreglo original de puntos de entrada.

El trabajo futuro es la implementación del cálculo final del convex hull con algoritmos que sean paralelizables (Jarvis March, Quickhull, Kirkpatrick–Seidel, Chan's algorithm) para comprobar si alguno de ellos, a pesar de aumentar la complejidad del algoritmo, puede lograr mejores tiempos aún al aprovechar el paralelismo de la GPU. Por otra parte, la implementación de la solución para más dimensiones puede ser un importante aporte, por sobre todo en $3D$, pero se debe considerar que el tamaño requerido de memoria crece en potencias según la cantidad de dimensiones, lo cual puede dificultar el cálculo de grandes volúmenes de datos en la GPU.

También en el trabajo futuro está, siguiendo el modelo de heaphull, el uso eficiente de memoria, lo cual podría ser un importante aporte de la mano de grandes volúmenes de datos, logrando data más compacta y facilitando el cálculo de mayores números de datos.

Todo lo anterior requiere un tiempo de desarrollo de la investigación mayor a lo que abarca este trabajo, debiendo resolver dificultades asociadas al manejo de la memoria dinámica en GPU y la reducción de velocidad con la que se ve sancionada la ejecución de instrucciones secuenciales, complejizando el desarrollo de un algoritmo de convex hull íntegro en la GPU.
...

Referencias

[1] Cormen T H., Leiserson Ch. E., Rivest R. L. and Stein C.: Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill (2001). Section 33.3: Finding the convex hull, 947–957.

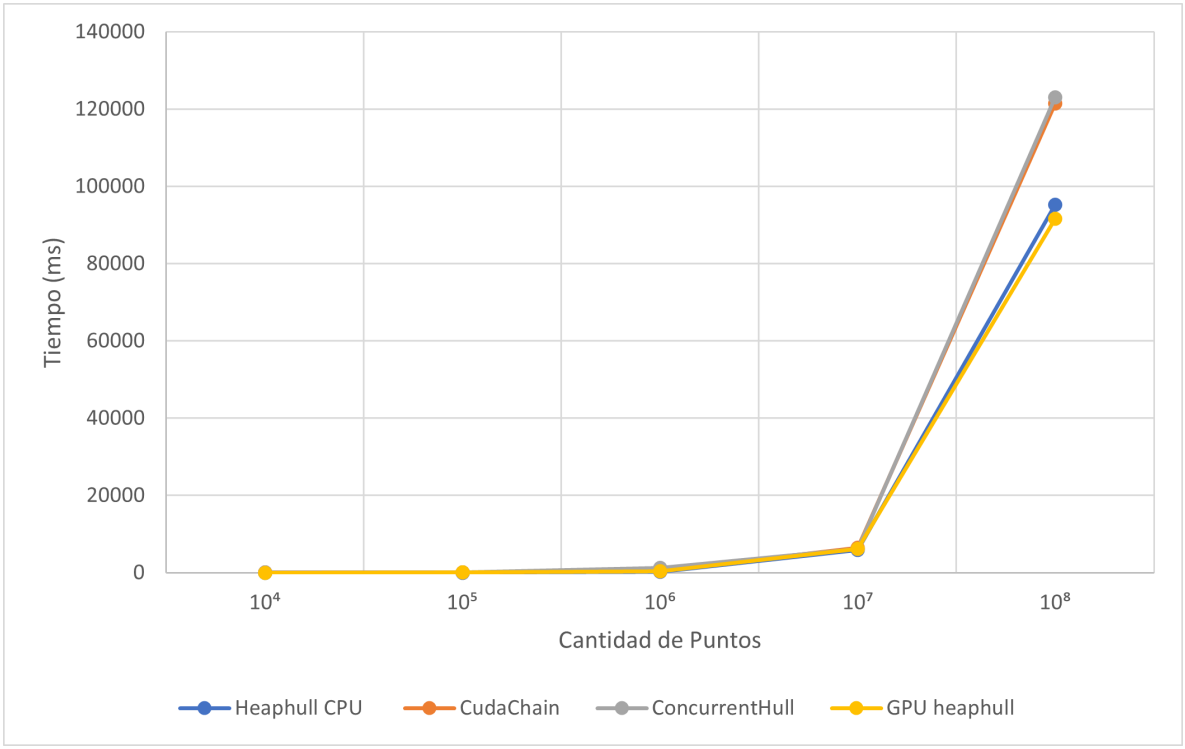

Figura 6: Tiempos de cómputo del convex hull en el peor caso

[2] Preparata F.P., Shamos M.I. (1985) Convex Hulls: Basic Algorithms. In: Computational Geometry. Texts and Monographs in Computer Science. Springer, New York, NY

[3] Chul E. Kim.: A linear time convex hull algorithm for simple polygons, Computer Science Technical Report Series, University of Maryland (1980)

[4] Akl, S. G., "Optimal parallel algorithms for computing convex hulls and for sorting", Computing (1984) 33: 1. https://doi.org/10.1007/BF02243071, pages 1-11.

[5] Graham, R. L.: An efficient algorithm for determining the convex hull of a planar set, Information Processing Letters I (1972) 132-133.

[6] Jarvis, R. A.: On the identification of the convex hull of a finite set of points in the plane. Information Processing Letters, 2(1):18–21 (1973)

[7] Kirkpatrick, David G.; Seidel, Raimund (1986). "The ultimate planar convex hull algorithm". SIAM Journal on Computing. 15 (1): 287–299.

[8] Avis D., Bremner D., Seidel R.: How good are convex hull algorithms?, Computational Geometry, Elsevier, Volume 7, Issues 5–6, (1997) 265-301.

[9] Chand D., Kapur S.: An algorithm for convex polytopes, J. ACM, 17 (1970) 78-86.

[10] Seidel R.: Output-size sensitive algorithms for constructive problems in computational geometry, Ph.D. Thesis, Dept. Comput. Sci., Cornell University (1986).

[11] Avis D., Fukuda K.: A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra, Discrete Comput. Geom., 8 (1992), 295-313.

[12] Seidel R.: A convex hull algorithm optimal for point sets in even dimensions, Technical Report, University of British Columbia, Dept. of Computer Science, (1981)

[13] Clarkson, K.L. and Shor, P.W.: Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental; Proc. 4th Ann. ACM Sympos. Comput. Geom. (1988), 12-17.

[14] Chazelle, B.: An optimal convex hull algorithm in any fixed dimension, Discrete Comput. Geom., 10 (1993), 377-409

[15] Barber, C. Bradford and Dobkin, David P. and Huhdanpaa, Hannu.: The Quickhull Algorithm for Convex Hulls, ACM Trans. Math. Softw.,22(4),469-483(1996)

[16] Voskov, A., A. Dzuban., A. Maksimov, TernAPI program for the calculation of ternary phase diagrams withisolated miscibility gaps by the convex hull method, Fluid Phase Equilibria: 388, 50-58 (2015)

[17] Zeng, M., Y. Yang., J. Zheng., J. Cheng, Maximum margin classification based on flexible convex hulls, Neurocomputing: 149, 957-965 (2015)

[18] Zhang, X., E. Zhao., Z. Wu., K. Li., Q. Hou, Phase stability and mechanical properties of ruthenium borides from first principles calculations, Computational Materials Science: 95, 377-383 (2014)

[19] Youssef, M., Asari, V. Human action recognition using hull convexity defect features with multimodality setups. Pattern Recognition Letters: 34(15), 1971-1979 (2013)

[20] López, A., X. Li., W. Yu, Convex and concave hulls for classification with support vector machine. Neurocomputing: 122, 198-209 (2013)

[21] Wilderjans, T.,E. Ceulemans., K. Meers, CHull:A generic convex-hull-based model selection method. Behavior research methods, 45(1), 1-15 (2013)

[22] Soltanifar, M., G. R. Jahanshahloo., F. H. Lotfi., S. M. Mansourzadeh, On efficiency in convex hull of DMUs, Applied Mathematical Modelling: 37(4), 2267-2278 (2013)

[23] Martinsky, O., Algorithmic and mathematical principles of automatic number plate recognition systems, Brno Univesity of Technology (2007)

[24] Buitrago, Oscar Y, Ramírez, Andrés L, & Britto, Rodrigo A. Nuevo Algoritmo para la Construcción de la Envolvente Convexa en el Plano. Información tecnológica, 26(4), 137-144. (2015) https://dx.doi.org/10.4067/S0718-07642015000400017

[25] Héctor Ferrada, Cristobal Navarro and Nancy Hitschfeld: A Filtering Technique for Fast Convex Hull Construction in IR 2 . Computational and Applied Mathematics Journal, 364:112298, 2019.

[26] Jiayin, Zhang & Mei, Gang & Xu, Nengxiong & Yang, Kun.: A Novel Implementation of QuickHull Algorithm on the GPU. arxiv (2015)

[27] Chen W., Nakano K., Masuzawa T., and Tokura N., "Optimal parallel algorithms for computing convex hulls," IEICE Transactions, vol. J75-D1, no. 9, pp. 809–820, 1992.

[28] Berkman O., Schieber B., Vishkin U., "A fast parallel algorithm for finding the convex hull of a sorted point set", Internat. J Comput. Geom. Appl. 6, pp. 231-242, 1996.

[29] Liu, G. & Chen, C. J.: A new algorithm for computing the convex hull of a planar point set. Zhejiang Univ. - Sci. A (2007) 8: 1210.

[30] Liu, Jigang, "Parallel Algorithms for Constructing Convex Hulls."(1995). LSU Historical Dissertations and Theses. 6029. https://digitalcommons.lsu.edu/gradschool_disstheses/6029

[31] Miller R. and Stout Q. F. , "Efficient parallel convex hull algorithms,ïn IEEE Transactions on Computers, vol. 37, no. 12, pp. 1605-1618, Dec. 1988. doi: 10.1109/12.9737

[32] Stein A., Geva E., El-Sana J.: CudaHull: Fast parallel 3D convex hull on the GPU, Computers & Graphics, Volume 36, Issue 4 (2012) Pages 265-271,ISSN 0097-8493, https://doi.org/10.1016/j.cag.2012.02.012.

[33] V. N. Waghmare and D. B. Kulkarni, "Convex Hull Using K-Means Clustering in Hybrid (MPI/OpenMP) Environment,"2010 International Conference on Computational Intelligence and Communication Networks, Bhopal, 2010, pp. 150-153. doi: 10.1109/CICN.2010.40

[34] Goodrich Michael T., Finding the convex hull of a sorted point set in parallel, Information Processing Letters, Volume 26, Issue 4, (1987), Pages 173-179, ISSN 0020-0190, https://doi.org/10.1016/0020-0190(87)90002-0.

[35] M. Nakagawa, D. Man, Y. Ito and K. Nakano, "A Simple Parallel Convex Hulls Algorithm for Sorted Points and the Performance Evaluation on the Multicore Processors,"2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, Higashi Hiroshima, 2009, pp. 506-511.

[36] S. G. Akl and G. T. Toussaint.: "A fast convex hull algorithm," Information Processing Letters, vol. 7, no. 5, pp. 219–222, Aug. 1978.

[37] Toussaint G., Akl S. and Devroye L., Efficient convex hull algorithms for points in two and more dimensions, Technical Report No. 78.5, McGill University (1978).

[38] F. P. Preparata and S. J. Hong.: Convex hulls of finite sets of points in two and three dimensions. Commun. ACM 20, 2 (February 1977), 87-93.

[39] Kallay, M.: The complexity of incremental convex hull algorithms in R^d, Information Processing Letters, Volume 19, Issue 4, 1984, Page 197.

[40] Andrew, A. M.: Another efficient algorithm for convex hulls in two dimensions, Information Processing Letters, 9(5):216–219 (1979).

[41] Tang, Min & Zhao, Jie-Yi & Tong, Ruo-feng & Manocha, Dinesh.: GPU accelerated convex hull computation. Computers & Graphics. 36. 498–506 (2012)

[42] Orlowski M.: On the conditions for success of sklansky's convex hull algorithm, Pattern Recognition, Elsevier, 16(6) (1983), 579-586

[43] Sklansky J.: Measuring concavity on a rectangular mosaic, IEEE Trans. Comput. 21 (1972), 1355-1364.

[44] Vyšniauskaitė, Laura & Šaltenis, Vydūnas. (2006). A priori filtration of points for finding convex hull. Technological and Economic Development of Economy. 12. 341-346. 10.3846/13928619.2006.9637764.

[45] Sharif, Muhammad. (2011). A new approach to compute convex hull. Innovative Systems Design and Engineering. 2. 187-193.

[46] Mei, Gang & Xu, Nengxiong. (2015). CudaPre3D: An Alternative Preprocessing Algorithm for Accelerating 3D Convex Hull Computation on the GPU. Advances in Electrical and Computer Engineering. 15. 35-44. 10.4316/AECE.2015.02005.

[47] Mei, Gang & Sixu, Guo. (2017). CudaPre2D: A Straightforward Preprocessing Approach for Accelerating 2D Convex Hull Computations on the GPU*. 10.1109/PDP2018.2018.00119.

[48] Mei, Gang. (2016). CudaChain: an alternative algorithm for finding 2D convex hulls on the GPU. SpringerPlus. 5. 1-26. 10.1186/s40064-016-2284-4.

[49] Jurkiewicz, T., & Danilewski, P. (2011). Efficient Quicksort and 2D Convex Hull for CUDA, and MSIMD as a Realistic Model of Massively Parallel Computations.

[50] S. Srungarapu, D. P. Reddy, K. Kothapalli and P. J. Narayanan, “Fast Two Dimensional Convex Hull on the GPU,"2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications, Singapore, 2011, pp. 7-12. doi: 10.1109/WAINA.2011.64

[51] Mei, Gang. (2014). A Straightforward Preprocessing Approach for Accelerating Convex Hull Computations on the GPU. arXiv preprint arXiv:1405.3454.

[52] Dunlaing, Colm O.: CUDA implementation of Wagener's 2D convex hull PRAM algorithm, Computer Science (2012), arXiv, http://arxiv.org/abs/1203.5004v2

[53] Masnadi, Sina & LaViola, Joseph. (2020). ConcurrentHull: A Fast Parallel Computing Approach to the Convex Hull Problem. 10.1007/978-3-030-64556-4_46.

[54] Touati Sid, Worms Julien, Briais Sébastien. The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation. Concurrency and Computation: Practice and Experience, Wiley, 2013, 25 (10), pp.1410-1426. 10.1002/cpe.2939. hal-00764454.