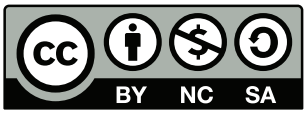
ALGORITMO DE CÁLCULO DE CÁPSULAS CONVEXAS

Trabajo Final de Licenciatura en Ciencias de la Computación

Universidad Nacional de Córdoba
Facultad de Matemática, Astronomía y Física (Córdoba - Argentina)

Autor: Miranda Leonardo

9 de julio de 2015



Índice

1. Introducción **3**

2. Motivación **4**

3. Cápsula Convexa **6**

4. Descripción Algoritmo de Cálculo de Cápsulas Convexas **9**
4.1. Filtro 9
4.2. Convexidad 13

5. Herramientas **16**
5.1. Modelos Computacionales 17
5.2. Cuda 18

6. Implementación **27**
6.1. Diseño 29
6.2. Código 30
6.3. Análisis Empírico 39
6.4. Comparación con la versión secuencial 42

7. Resultados **44**

8. Mejoras **59**

9. Bibliografía **60**

10. Conclusiones **60**

1. Introducción

La teledetección es la técnica utilizada para estudiar un objecto sin necesidad de tener que tocarlo. El escenario en el que nos enfocamos es el de la teledección espacial. Este escenario consiste en estudiar objetos de alguna de estas dos formas:

1. Utilizando la luz solar y un sensor montado sobre una aeronave o satélite, de modo que cada rayo de luz proveniente de sol, que rebote contra el objeto a estudiar y vaya directo al sensor montado sobre la aeronave o satélite, sea digitalizado (componiendo un píxel) y enviado a una estación terrestre donde con cada dato que va llegando (píxel) se va formando la imagen del objeto de interés.

2. Otra forma es similar a la primera, sólo que en vez de usar la luz solar, utilizamos un aparato que emite una señal en alguna banda del espectro. Esta señal es utilizada para pegarle al objeto de interés. Luego la señal que rebota será captado por el sensor montado en alguna aeronave o satélite, digitalizado y enviado a una estación terreste para componer un dato (píxel) de la imagen del objeto de interés.

Una vez definido el escenario previamente descrito, la idea de este trabajo es centrarse en las imágenes recibidas desde un satélite, más concretamente trabajar con la imagen ya guardada en archivo y operar sobre ella. No tendremos que lidiar con radares, sensores o satélites/aeronaves.

2. Motivación

El problema de encontrar la cápsula convexa de conjuntos encuentra su aplicación práctica en el reconocimiento de patrones, procesamiento de imágenes, estadísticas, SIG (Sistemas de Información Geográfica), etc.En particular en la segmentación y clasificación en imágenes de teledetección.
La idea central subyacente al procesamiento de imágenes es la de transformar una imagen por medio de una ecuación o una serie de ecuaciones (usando la computadora), para obtener una nueva imagen digital o pictórica, que a su vez pueda ser manipulada por otros programas computacionales o mostrada en pantalla, por impresora, etc.
Una de las tareas más importantes en el procesamiento y análisis de imágenes es clasificar cada píxel como perteneciente a una cierta categoría de terrenos también llamada "tema"por ejemplo clasificar un píxel perteneciente a;la clase agua o trigo o ciudad o camino etc.
Una persona tratando de clasificar hechos en una imagen, usa elementos de la interpretación visual para identificar grupos de píxeles homogéneos que representen estos hechos o tipos de coberturas de interés. La clasificación digital usa información espectral representada por los valores digitales en una o más bandas, para clasificar cada píxel individual basado en esta información espectral.
Esto podría llamarse la identificación de un patrón espectral.
En cualquier caso el objetivo es asignar todos los píxeles de la imagen a clases o temas particulares. La imagen clasificada que resulta, es un mosaico de píxeles cada uno de los cuales pertenece a un tema particular y es esencialmente un mapa temático de la imagen original.
Cuando hablamos de clases es bueno distinguir entre clases de información y clases espectrales. Las primeras son aquellas categorías de interés que la persona está realmente tratando de identificar en la imagen. Tal como distintos tipos de cultivos, diferentes clases de forestación, tipos de rocas etc. Las clases espectrales son grupos de píxeles que son uniformes (similares) en valores de brillo en las diferentes bandas. El objetivo final es crear una correspondencia entre estas clases espectrales y las clases de información que nos interesan.
Por supuesto es raro que exista una correspondencia uno-uno entre estos dos tipos de clases. Por ejemplo puede haber clases espectrales que no correspondan a ninguna clase temática de interés. Inversamente, clases temáticas amplias como (forestación) podrían tener subclases espectralmente separables. Usando el ejemplo de la forestación, las sub-clases espectrales podrían deberse a variaciones en la edad, especie, densidad, sombra o iluminación. Justamente el trabajo del analista de la imagen es decidir sobre la utilidad de las diferentes clases espectrales con respecto a las clases temáticas de interés.
Según sean los datos que se utilizan para el proceso de clasificación, estos procedimientos se agrupan en tres clases:

1. Reconocimiento de patrones espectrales.
2. Reconocimiento de patrones espaciales.
3. Reconocimiento de patrones temporales.

Los segundos clasifican cada píxel de la imagen en base a sus relaciones espaciales con los píxeles vecinos. Por ejemplo, considerando textura, proximidad,

tamaño, formas, etc., intentan replicar la síntesis realizada por la mente humana en el proceso de interpretación visual.
De hecho, para realizar una clasificación lo más completa y correcta posible es preciso generalmente valerse de una combinación de diferentes técnicas de clasificación.
Los métodos que más se han desarrollado son los de reconocimiento de patrones espectrales que a su vez pueden ser de clasificación supervisada o no supervisada.

Clasificación supervisada Comprende dos etapas: entrenamiento y clasificación propiamente dicha. En la etapa de entrenamiento se identifican áreas representativas (áreas de entrenamiento) y se desarrolla en base a ellas una tabla numérica de los atributos espectrales de cada tipo de cobertura del terreno. En la etapa de clasificación cada píxel de la imagen se coloca dentro de alguna de las categorías determinadas en la etapa anterior en base a la que sea más parecida. Si el píxel no se puede caracterizar claramente, en general se lo rotula como "desconocido". La categoría asignada a cada píxel en esta etapa se registra en la celda correspondiente a una nueva imagen que es la imagen resultante. Esta última, como también es digital, se puede usar en diversas formas en procedimientos posteriores, como por ejemplo, en un S.I.G..

Clasificación no-supervisada Estas técnicas de clasificación involucran algoritmos que examinan los píxeles en una imagen y los colocan en una de las clases determinadas por medio de agrupamientos naturales (clusters) que se hallan presentes en los valores de la imagen. La premisa básica para estas técnicas es que los valores dentro de un cierto tipo de cobertura de terreno estén próximos en el espacio de los valores espectrales, mientras que los datos provenientes de clases diferentes deberían estar comparativamente bien separados. Si bien pueden estar claramente separadas, no conocemos a priori la identidad de cada clase. Para identificarlas se hace necesario tener alguna información sobre el terreno. Los procedimientos exploratorios son especialmente útiles para entender la compleja naturaleza de las relaciones multivariadas. La búsqueda de una estructura "natural"para agrupar los datos es una importante técnica exploratoria. Los métodos de clasificación no supervisada o de agrupamiento no son métodos de clasificación propiamente dicha, son técnicas más primitivas sin ningún supuesto respecto al número de grupos o la estructura de cada uno. La agrupación se realiza basándose en similitudes o distancias. Para realizar estos agrupamientos es indispensable definir lo que se entiende por similar. También se cuenta con técnicas mixtas que se valen de las propiedades discriminatorias de una o varias reglas de clasificación no supervisada con otra u otras supervisadas. Entre ellas está la que consiste en determinar la cápsula convexa de un subconjunto previamente elegido en el espacio (multidimensional) de los valores radiométricos de una imagen multiespectral. Existen varios algoritmos referentes a computación secuencial de la cápsula convexa de un conjunto aunque mucho menor es el número de los que hacen tal cálculo por medio de computación paralela. Las imágenes de teledetección son, en general, de gran tamaño (del orden de 6000x6000 píxeles). Los algoritmos clásicos, secuenciales consumen mucho tiempo. Muchos de estos procedimientos pueden definirse por medio de algoritmos para ejecución simultánea en varios procesadores lo que ahorra el tiempo de cálculo en forma considerable.

3. Cápsula Convexa

El algoritmo que se presenta en este trabajo intenta calcular la cápsula convexa dado un conjunto de puntos de entrada. Definiremos la cápsula convexa de un conjunto de puntos. Previamente necesitaremos otras definiciones que las presentamos a continuación y al final definiremos la cápsula convexa.

Definición 1. *$P_{i,j}$ = Recta formada por los puntos i y j, con $i = (x_1 \in \mathbb{R}, y_1 \in \mathbb{R})$, $j = (x_2 \in \mathbb{R}, y_2 \in \mathbb{R})$*

Definición 2. *$P_{i,j}(v)$ = Valor que se obtiene al valuar en el punto v a la recta formada por los puntos i y j, con $i = (x_1 \in \mathbb{R}, y_1 \in \mathbb{R})$, $j = (x_2 \in \mathbb{R}, y_2 \in \mathbb{R})$, $v \in \mathbb{R}$*

Definición 3. *Dado un conjunto $S = \{S_1,\ldots,S_n\}$, con $S_j = (x_j \in \mathbb{R}, y_j \in \mathbb{R})$, con $j = 0 \ldots n$. Un punto $i \in S$ es un punto extremo si se da una de las cuatro condiciones:*

1. $\exists v = (x_1 \in \mathbb{R}, y_1 \in \mathbb{R}) \in S : \forall k = (x_2 \in \mathbb{R}, y_2 \in \mathbb{R}) \in S : y_2 \geq P_{i,v}(x_2)$

2. $\exists v = (x_1 \in \mathbb{R}, y_1 \in \mathbb{R}) \in S : \forall k = (x_2 \in \mathbb{R}, y_2 \in \mathbb{R}) \in S : y_2 \leq P_{i,v}(x_2)$

3. $i = (x_1 \in \mathbb{R}, y_1 \in \mathbb{R}) \in S : \forall k = (x_2 \in \mathbb{R}, y_2 \in \mathbb{R}) \in S : x_1 \leq x_2$

4. $i = (x_1 \in \mathbb{R}, y_1 \in \mathbb{R}) \in S : \forall k = (x_2 \in \mathbb{R}, y_2 \in \mathbb{R}) \in S : x_1 \geq x_2$

Definición 4. *Dado un conjunto de puntos $P = \{P_1,\ldots,P_n\}$ dado por la figura 1.*

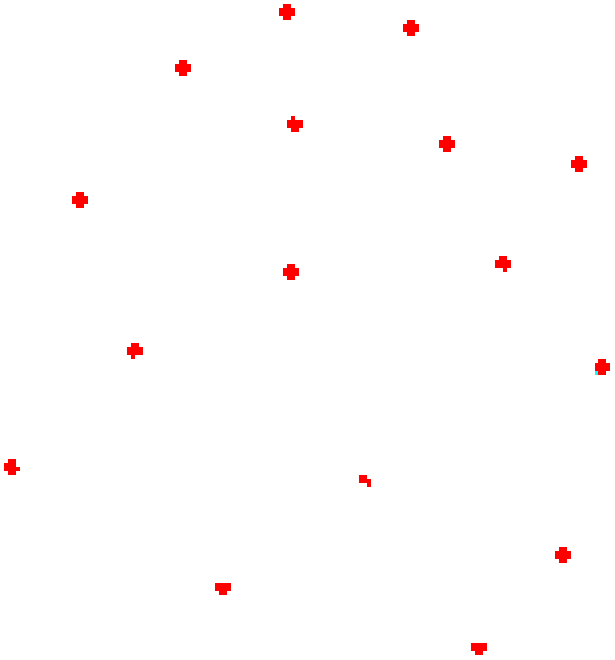


Figura 1: Conjunto de puntos

La cápsula convexa del conjunto de puntos P será el menor subconjunto $S \in P$ que sólo contiene puntos extremos de P.

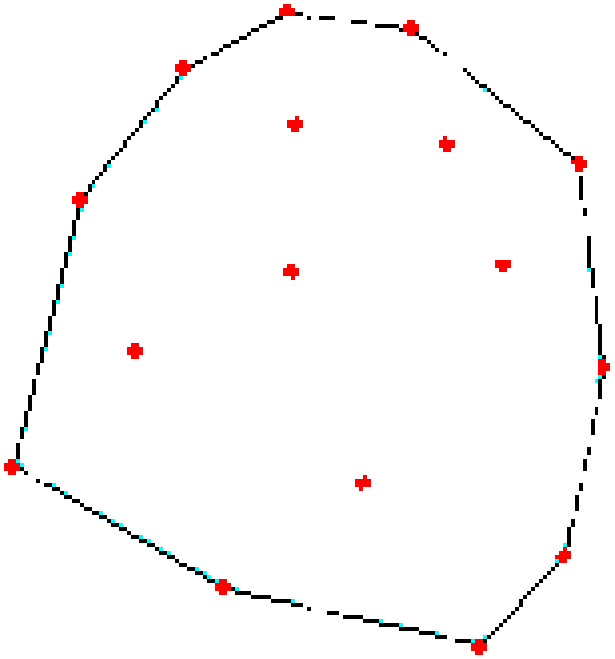


Figura 2: Cápsula Convexa

4. Descripción Algoritmo de Cálculo de Cápsulas Convexas

En esta sección veremos cómo funciona el algoritmo para calcular cápsulas convexas, luego veremos como se implementa utilizando lenguajes de programación CUDA (extensión de C++), C/C++ y Bash. En esencia este algoritmo trata de explotar el beneficio de hacer operaciones en paralelo que nos provea un dispositivo determinado. El algoritmo que calcula la cápsula convexa consta de dos etapas para lograr su objetivo:

1. Filtro
2. Convexidad

4.1. Filtro

En esta primera etapa, que es donde se procesan los puntos por primera vez, el objetivo es tratar de dejar en lo posible sólo los puntos extremos del conjunto de puntos que recibimos como entrada para alivianar la carga al algoritmo que determina los puntos de la cápsula convexa. Supongamos que tenemos el siguiente conjunto de puntos:

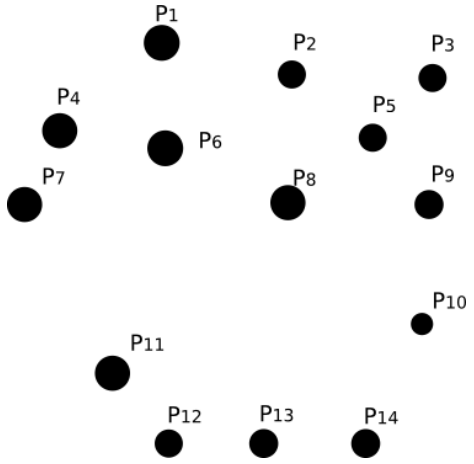


Figura 3: Conjunto de puntos

Supongamos que podemos lanzar M hilos en paralelo, como tenemos N puntos, la idea es asignar a cada hilo una cantidad de puntos de modo tal que queden equitativamente distribuidas entre los M hilos. Si M es mayor a N, entonces asignamos un punto para los primeros N hilos. En la implementación, este caso no se da debido a que sabemos cuantos puntos tenemos y en base a esto podemos ajustar la cantidad de hilos en paralelo que queremos que trabajen. Si M es menor o igual a N, entonces a cada hilo le corresponde procesar por lo menos un punto. Cada hilo trabaja en el conjunto de puntos que se le asignó de la siguiente manera: Para cada punto del conjunto de puntos que le corresponde trabajar, si el punto contiene uno anterior que esté a la misma altura y uno posterior a

la misma altura, entonces ese punto se elimina. Propongo las siguientes definiciones para explicar el pseudocodigo de este filtro y del siguiente que calcula la convexidad.

Definición 5. $\{\}^n_< = \{(x_i, y_i) \in \mathbb{R} \times \mathbb{R} : \forall j \ > \ i : \ (x_j, y_j) \ \in \ \{\}^n_< : \ (y_j \ < \ y_i) \ \vee \ (y_j \ = \ y_i \ \wedge \ x_i \ <= \ x_j)\}, \ n \ \in \ \mathbb{N}, j, i \in [1, \ldots, n]$

Definición 6. S_i = *Elemento i-esimo del conjunto S*

Definición 7. *Dado* $S = (x, y) \in \mathbb{R} \times \mathbb{R}, \ X(S) \ = \ x$

Definición 8. *Dado* $S \ = \ (x, y) \ \in \mathbb{R} \times \mathbb{R}, \ Y(S) \ = \ y$

Definición 9. $modulo : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$.
Calculo del valor entero que resulta del resto de la division entera entre el primer argumento y el segundo

Definición 10. Γ = *Cantidad de hilos que puedo lanzar en paralelo en un dispositivo*

En pseudocódigo el filtro funcionaría así:

Data: S = $\{\}^n_<$, *con* $n \in \mathbb{N}$
Result: $S' \subseteq S$

$M \leftarrow \Gamma$;
$f \leftarrow \#S$;
$K \leftarrow f/M$;

begin En paralelo entre los M hilos
  $id \in [0, \ldots, M]$, tal que no hay dos hilos con el mismo id;
  for $j \leftarrow S_{id}$ **to** S_{id+f} **do**
    $A \leftarrow S_{j-1}$;
    $P \leftarrow S_{j+1}$;
    if $j == S_0 \vee j == S_n$ **then**
      $S' \leftarrow S' \cup \{j\}$;
    end
    else
      if $Y(j) != Y(A) \vee Y(j) != Y(P)$ **then**
        $S' \leftarrow S' \cup \{j\}$;
      end
    end
  end
end

Para el caso de la figura 3 la cantidad de puntos que tenemos es 14, osea N = 14. Para este caso seguramente podremos contar con M = 7 hilos que trabajen en paralelo.
Supongamos que tenemos el conjunto de puntos de la figura 3 ubicados en un arreglo como el que se muestra a continuación donde disponemos que puntos les tocará a cada uno de los M = 7 hilos que pueden trabajar sobre él.

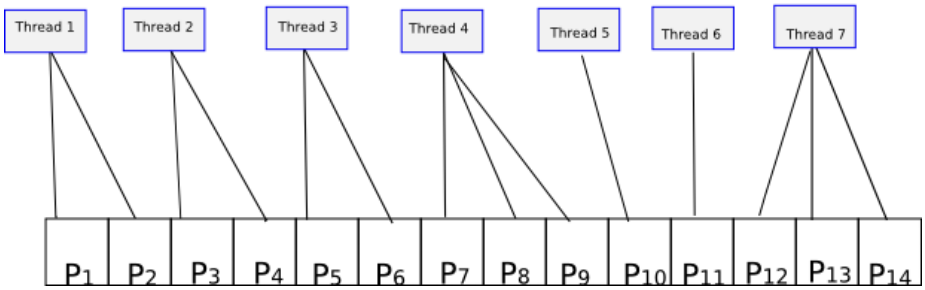


Figura 4: Aplicación del primer filtro a un conjunto de puntos

El resultado que obtenemos al aplicar el primer filtro al conjunto de puntos descritos en la figura 4 es el siguiente:

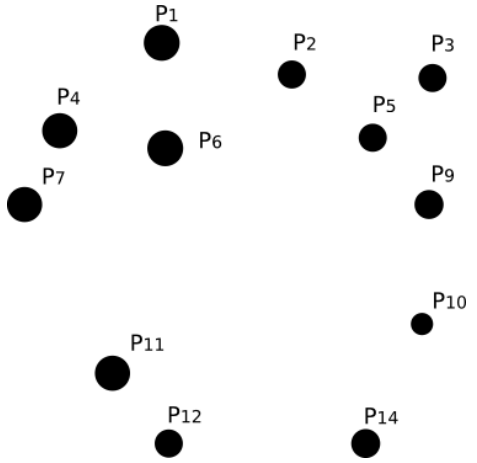


Figura 5: Resultado de aplicación del primer filtro a un conjunto de puntos

Como podemos notar aplicando el filtro los puntos que se descartan en este caso son el punto 8 y el punto 13. A continuación vemos el funcionamiento del algoritmo que termina de calcular la cápsula convexa a partir del resultado que da el filtro.

4.2. Convexidad

A partir del resultado obtenido por el filtro anterior obtenemos un conjunto de puntos mucho mas reducido, del cual todavía tenemos que seguir descartando algunos para obtener nuestra cápsula convexa.
El algoritmo que calcula la cápsula recorrerá todo el conjunto de puntos dejado por el filtro anterior empezando por el mayor punto que está más hacia la izquierda hasta llegar al más chico que se encuentre más hacia la derecha. A los mayores puntos nos referimos siempre a los puntos que están por encima de todos los demás, mientras que a los más chicos nos referimos a los que están por debajo de todos los demás (Puede haber más de un mínimo y mas de un máximo).
El algoritmo procederá de la siguiente manera:

1. Trazará una línea vertical a partir del punto máximo que se encuentre más hacia la izquierda.
2. Luego armará dos variables pivots que tomarán el valor del punto máximo del paso anterior. Con un pivot se analizaran puntos hacia un lado del punto máximo y con el segundo los puntos hacia el lado contrario.
3. Como el máximo cumple la propiedad de ser un punto extremo analizamos los puntos siguientes. Para esto vamos recorriendo los subsiguientes puntos y hacemos lo siguiente:

    a) Para cada pivot hacemos lo siguiente:
        1) Trazamos una recta entre el pivot y el punto a evaluar.
        2) Luego analizamos si todos los demás puntos están por encima por debajo o hacia unos de los dos costados de dicha recta. Si alguna de esta condiciones se da, entonces el punto a evaluar es un punto extremo y lo que hacemos es que el pivot que usamos para el análisis tome dicho valor. En caso contrario el pivot queda con el valor que traía y el punto a evaluar se descarta.
4. Y así procedemos hasta haber analizado todos los puntos

Para el caso de nuestro ejemplo el recorrido de los pivots aplicando el algoritmo es como muestra la siguiente figura:

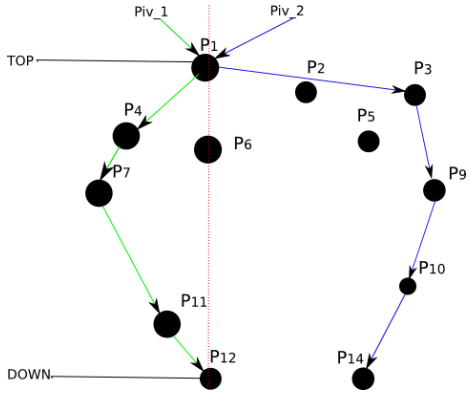


Figura 6: Descripción del proceder del algoritmo para calcular cápsulas convexas

En pseudocódigo del algoritmo sería así:

Data: $S = \{\}^{n}_{<},\ con\ n\ \in \mathbb{N}$
Result: $S' = capsula\ convexa\ de\ S$

$M \leftarrow \Gamma$;
$TOP \leftarrow S_i : \forall S_j : j \neq i\ \wedge\ Y(S_j)\ \leq Y(S_i)\ \wedge X(S_i) \leq X(S_j)$;
$PIVOT\_IZQ \leftarrow TOP$;
$PIVOT\_DER \leftarrow TOP$;
$f \leftarrow \#S$;
$k \leftarrow f/M$;

for $i \leftarrow 0$ **to** M **do**
    $M_i \leftarrow \{S_{(i\times k)}, \ldots, S_{(i\times k)\ +\ k}\}$;
end
$M_n \leftarrow M_n + (modulo(f, M))$;

$witness1 \leftarrow 0$;
$witness2 \leftarrow 0$;

```
foreach e ∈ S − {TOP} do
    x ← X(e);
    y ← Y(e);
    foreach pivot ∈ {PIVOT_IZQ, PIVOT_DER} do
        begin En paralelo entre los M hilos
            i ← El Id del hilo que lo distingue de los demás;
            x_piv ← X(pivot);
            y_piv ← Y(pivot);
            foreach j ∈ M_i do
                x_j ← X(j);
                y_j ← Y(j);
                if x == x_piv then
                    if witness1 == 0 ∧ x_j > x then
                        witness1 ← witness1 + 1;
                    end
                    if witness2 == 0 ∧ x_j < x then
                        witness2 ← witness2 + 1;
                    end
                end
                else if P_{e,pivot}(x_j) < y_j then
                    witness1 ← witness1 + 1;
                end
                else if P_{e,pivot}(x_j) > y_j then
                    witness2 ← witness2 + 1;
                end
            end
        end
        if witness2 == 0 ∨ witness1 == 0 then
            S' = S' ∪ {e};
            if pivot == PIVOT_IZQ then
                PIVOT_IZQ ← pivot;
            end
            else
                PIVOT_DER ← pivot;
            end
        end
        witness2 ← 0;
        witness1 ← 0;
    end
end
```

5. Herramientas

En esta sección daremos un vistazo a las herramientas que utilizaremos para la implementación del algoritmo antes presentado.
La implementación se hizo utilizando el lenguaje C/C++.
Para cargar las imágenes en memoria y poder operar con el algoritmo luego se utilizó una librería llamada 'helper_image'.
Como el hardware que se utilizó para poder hacer operaciones en paralelo se

encuentra en servidores a los cuales hay que conectarse, se empleó el lenguaje script bash para poder automatizar ciertas operaciones que se necesitaban realizar, con el objetivo de que el uso sea más simple para el usuario final.
El dispositivo que utilizamos para hacer operaciones en paralelo son placas de video del fabricante nvidia. Estas placas de video se encuentran en servidores a los cuales tengo acceso.
A continuación hablamos del dispositivo que utilizamos para poder hacer operaciones en paralelo.

5.1. Modelos Computacionales

Definición 11. *Un modelo computacional será una máquina virtual que provee operaciones a nivel de programación, donde dichas operaciones son implementadas en todas las arquitecturas subyacentes.*
Las propiedades fundamentales de un modelo de computación paralela es:

1. *Independencia de la arquitectura subyacente*
2. *Facilidad de programación, que quiere decir que el modelo permite al programador abstraerse de las características físicas del entorno de ejecución ocultando:*

   a) *Descomposición*
   b) *Asignación a procesadores*
   c) *Comunicación*
   d) *Sincronización*

PRAM (Parallel Random Access Machine):
Es un modelo de computación paralela, donde un conjunto de procesadores se encuentran conectados a una memoria global a través de una unidad de acceso a memoria como muestra la figura A. Esta unidad de acceso a memoria global asume que cada procesador hace sus operaciones sobre esta memoria global en O(1).
Esto hace que las operaciones de escritura y lectura puedan traer problemas no deseados (condición de carrera). Ese problema hace que haya variantes del modelo PRAM determinados por como están definidas las escrituras y las lecturas sobre la memoria global.

1. Modos de definir lectura:

   a) ER (Exclusive Reads): Sólo puede hacer una lectura a la vez sobre la memoria global.
   b) CR (Concurent Reads): Se pueden hacer varias lecturas al mismo tiempo por cada ciclo de reloj.

2. Modos de definir escritura:

a) EW (Exclusive Writes): Sólo se puede hacer una escritura a la vez sobre la memoria global.

b) CW (Concurrent Writes):Se pueden hacer varias escrituras al mismo tiempo por cada ciclo de reloj. Aquí se distinguen varias formas de definir las escrituras concurrentes.

En base a combinaciones entre los items de Modos de definir lectura y Modos de definir escritura se definen las variantes del modelo computacional PRAM mas populares:

1. CREW
2. EREW
3. CRCW

Algo importante para notar y destacar dentro de este modelo es que los procesos que corren en distintos procesadores utilizan la memoria global para comunicarse, no existe otro mecanismo.
Ahora veamos un modelo de computación paralela que se corresponde con dicha variante del modelo PRAM. Si bien este modelo presenta además de la memoria global (implementada con DRAM), memoria compartida, memoria constante, son sólo recursos extras con los que cuenta el programador. El modelo de computación paralela al que nos referimos es CUDA. Como los accesos en una memoria DRAM son concurrentes tanto para lectura como escritura, esto nos dice que CUDA será un modelo de computación paralela PRAM CRCW.

5.2. Cuda

El modelo computacional CUDA se basa en GPU (Graphics Process Unit) que a diferencia de una CPU (Central Process Unit) presenta una arquitectura diferente y se especializa en cálculos de operaciones en coma flotante, y por ende no esta preparada para atender demandas de sistemas operativos, dispositivos I/O como si lo esta una CPU.
Alrededor del año 1996 ya en las computadoras actuales se combina lo mejor de ambos mundos, múltiples procesadores en combinación con GPU en un mismo chasis. Años mas tarde Nvidia agrega hardware a los nuevos chips para lanzar una API para poder programar sobre una GPU. A ésta nueva API se la llama CUDA.
CUDA es un modelo computacional PRAM CRCW, específicamente una extensión del lenguaje C++ para que soporte computación paralela.
Explicaremos la estructura de un programa CUDA de aquí en mas a través de un ejemplo. Realizaremos la multiplicación de 2 matrices M y N para obtener como resultado una matriz P.

Estructura programa Cuda

Un programa cuda consta de varias etapas, donde se ejecutan funciones tanto en un host (CPU) como en una GPU. Un programa en cuda consta de

operaciones en código ANSI C como así también código ANSI C con extensiones. Es por esto que el compilador NVCC (Nvidia C Compiler) divide el proceso de compilación en dos etapas, en la primera compila todas las operaciones o procedimientos del lenguaje C/C++ y luego compila el código que está escrito para ejecutarse en el dispositivo GPU. Este último código que es como dijimos antes código ANSI C con extensiones, dichas extensiones son palabras claves que sirven para identificar a las funciones de tratamiento de datos en paralelo (Kernels), que son las funciones a ejecutar en la GPU.
Un programa en cuda comienza ejecutándose en un host simple (CPU). Cuando en este código se ejecuta una función Kernel, las operaciones de esta función se realizan en la GPU a través de muchos hilos que ejecutarán la misma función kernel, no necesariamente al mismo tiempo. A ese conjunto de hilos que ejecutan una función kernel se la denomina GRILLA. Una vez que todas los hilos terminan su ejecución, se retorna al código que se estaba ejecutando en la CPU hasta una nueva llamada a una función kernel.
Veamos ahora como hacer la multiplicación de matrices en una CPU común y luego veamos como hacer un programa similar en una GPU a través de CUDA. Proponemos el siguiente algoritmo:

```
void MatrixMultiplication(float *M, float *N, float *P, int Width){
        for (int i=0; i < Width; ++i){
                for (int j=0; j < Width; ++j){
                        float sum = 0;
                        for (int k=0; k < Width; ++k){
                                float a = M[i*Width + k];
                                float b = N[k*Width + j];
                                sum += a * b;

                        }
                        P[i*Width + j] = sum;
                }
        }
}
```

cuya forma de operar es como muestra la siguiente figura:

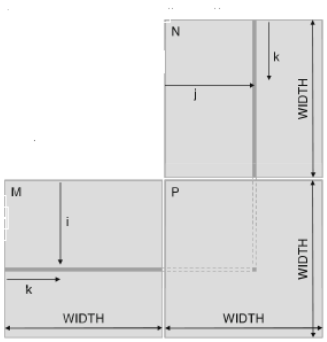


Figura 7: Multiplicación de Matrices versión CPU

Los dos loops de este algoritmo que iteran sobre las variables i,j indican que fila de la matriz N y que columna de la matriz M respectivamente se están tomando para hacer el producto punto y así generar un valor de P. El loop más interno es el que computa el producto punto entre las filas y columnas indicadas por los índices i,j. El lector debe saber notar que los puntos están ubicados en un arreglo de sólo una dimensión.
Ahora veamos como implementar este mismo algoritmo en CUDA.
Para esto tenemos que hacerlo en 3 etapas:

1. Movemos datos de la memoria del host a la memoria del dispositivo, para esto necesitamos reservar memoria en el dispositivo a través de la función

cudaMalloc(). Con esta función reservamos memoria para las matrices N, M, P.

2. Llamamos a la función kernel que se ejecutará en la GPU para computar el producto. Esta función la tenemos que declarar nosotros y veremos como.

3. Copiamos el resultado que se encuentra en memoria del dispositivo (GPU) a memoria del host (CPU) a través de la función cudaMemCpy()

Estos pasos estarán dentro de la nueva versión de la función MatrizMultiplication(). Esta función quedaría de la siguiente manera:

```
void MatrixMultiplication(float *M, float *N, float *P, int Width){

        int size = Width*Width*sizeof(float);
        float *Md, Nd, Pd;

        1.// Transferir M y N a la memoria de GPU
        cudaMalloc((void**)&Md, size);
        cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
        cudaMalloc((void**)&Nd, size);
        cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

        // Reservar memoria para P en GPU
        cudaMalloc((void**)&Pd, size);

        2.//Aqui va la invocacion a la funcion Kernel
        ...
        3.//Transferir el resultado que esta en P desde GPU a CPU
        cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

        //Libero memoria que esta en GPU
        cudaFree(Md);cudaFree(Nd);cudaFree(Pd);


}
```

Como podemos notar la función cudaMemCpy() no varía mucho con respecto a la función memcpy() de ANSI C salvo por la constante que se la pasa como cuarto parámetro que indica de dónde a dónde se transfieren los datos, esto puede ser en dos sentidos, de host a dispositivo y de dispositivo a host. La función cudaMalloc() como se puede ver en el código anterior se presenta un formato diferente con respecto a su par malloc() de ANSI C.
En este código presentado anteriormente se necesita alocar memoria del dispositivo en arreglos de valores flotantes.
Para esto como se puede observar la función cudaMalloc() toma como primer argumento una dirección o puntero al puntero que apunta a los datos en si (valores en coma flotante). Ese primer argumento de la función se corresponde con el segundo argumento de la función malloc() de ANSI C.
La función cudaFree() es similar a free() de ANSI C.
Hasta aquí los formatos de las funciones cuda que hemos utilizado son:

1. cudaMalloc((void**)P, size_t size)
2. cudaMemCpy((void*) dest, (void*) source, size_t size, DIRECCIóN)
3. cudaFree((void*) P)

Donde DIRECCIóN ∈ {cudaMemCpyHostToDevice, cudaMemCpyDeviceToHost}
Ahora veamos como hacer el paso 2 del código CUDA presentado anteriormente.
Para eso presentamos una función kernel denominada MatrixMulKernel() que

completará esta etapa:

```
__global__ MatrixMulKernel(float *M, float *N, float *P, int Width){

        //Identificador thread en 2D
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        float Pvalue = 0.0;

        for (int k=0; k < Width; ++k){
                float Mdelement = Md[ty*Width + k];
                float Ndelement = Nd[k*Width + tx];
                Pvalue += Mdelement * Ndelement;
        }
        Pd[ty*Width + tx] = Pvalue;
}
```

Como vemos la función esta declarada con una palabra clave en el prototipo que es __global__ . Hay 3 tipos de palabras claves que son las siguientes:

1. __global__ : La función va a ser lanzada desde código de la CPU (host) para ser ejecutada en la GPU.
2. __device__: La función sólo puede ejecutarse en la GPU y puede ser lanzada o llamada por funciones que se ejecutan en la GPU.
3. __host__ : La función sólo puede ser ejecutada en la CPU y ser llamada por funciones que se ejecutan en la CPU. Además agregamos que si una función en un programa CUDA no tiene ninguna de estas palabras claves en el prototipo, entonces quiere decir que esta función se ejecutará en el host (CPU).

También agregamos que podemos tener funciones declaradas con __host__ y __device__, lo que indica que se van a crear dos versiones de esta función. Una que puede ser llamada y ejecutada en el host y otra que puede ser llamada y ejecutada en el dispositivo (GPU).
Como vemos la función del ejemplo MatrixMulKernel() será lanzada desde código del host y ejecutada en el dispositivo por muchos hilos (GRILLA) que ejecutarán la misma función. La pregunta que puede surgir aquí es, si todos los hilos ejecutan la misma función. Como las distinguimos?.
Para distinguirlas se usan dos registros especiales que indican las coordenadas de un hilo, threadIdx.x, threadIdx.y, estos registros tendrán valores diferentes según la thread que esté ejecutando la función MatrixMulKernel(). De esta manera aprovechando los valores de estos registros, cada hilo tomará una columna de la matriz N indicada por el registro threadIdx.x y una fila de la matriz M indicada por el registro threadIdx.y. De esta manera como se observa tendremos tantos hilos en la grilla como productos de filas y columnas necesitemos.
Otra observación que podemos hacer es que los registros threadIdx.x y threadIdx.y sirven para guiar a los hilos sobre la porción de los datos en la que tienen que

operar. Esta forma de trabajar se la conoce como SPMD (Simple Process Múltiple Data) que quiere decir que el mismo procedimiento se ejecuta sobre distintos porciones de datos.
Hasta aquí tenemos nuestro ejemplo de programa en CUDA que realiza la multiplicación de matrices.
Veamos ahora como están organizadas las grillas que hemos mencionado antes para saber que podemos aprovechar de su estructura en nuestros programas.
Cuando una función kernel es lanzada, esta construye una grilla que es un conjunto de bloques donde cada bloque contiene hilos que ejecutan la función kernel propiamente dicha. Dicho esto aparecen nuevas variables para poder identificar a los hilos dentro de un bloque utilizando además las variables threadIdx.x, threadIdx.y, las variables blockIdx.x y blockIdx.y. Con estas variables podremos identificar hilos y guiarlas por los datos que tienen que manejar de modo similar a como hacemos con las variables threadIdx.x, threadIdx.y.
Una pregunta que surge en esta instancia es. Se puede determinar la cantidad de bloques que queremos en una grilla como así también la cantidad de hilos en cada bloque ?.
La respuesta es si, en el momento de llamado de la función kernel podemos especificar con parámetros de configuración la estructura de la grilla. Estos parámetros de configuración aparecen entre símbolos <<<>>> separados todos los parámetros por comas. Los datos que determinan la estructura de los bloques como así también de la grilla son tipos de datos ya definidos en CUDA cuya estructura en C se llama dim3. Esta estructura contiene 3 campos (x,y,z) para determinar la estructura de la grilla o bloque, dependiendo de lo que quiera especificar la variable de tipo dim3 definida por el programador.

Sintaxis:

```
<funcion kernel><<<DimGrilla, DimBloque>>>(PARAM_1,...,PARAM_N)
```

Donde DimGrilla y DimBloque son variables de tipo dim3 que especifican la cantidad de bloques en la grilla y la cantidad de hilos en cada bloque respectivamente.
Por ejemplo si queremos una grilla de un bloque y 512 hilos en cada bloque tenemos que especificar el siguiente código:

```
dim3 DimGrid(1,1,1);
dim3 DimBlock(32,16,1);

KernelFunction<<<DimGrid,DimBlock>>>(PARAM_1,..., PARAM_N);
```

En este caso estamos especificando que el bloque lucirá como una matriz ya que tendrá componentes (x,y). Si queremos podemos especificar el bloque sin imponer coordenadas (x,y) de la siguiente manera:

```
dim3 DimGrid(1,1,1);
dim3 DImGrid(512,1,1);
KernelFunctionName<<<DimGrid, DimBlock>>>(PARAM_1,..,PARAM_N);
```

O lo que es equivalente al código anterior es:

```
KernelFunctionName<<<1, 512>>>(PARAM_1, ..., PARAM_N);
```

Una barrera que sirve para sincronizar hilos de un mismo bloque es la función __syncthreads() que es una barrera que suspende al hilo que ha hecho la llamada en ese punto hasta que todas los demás hilos del mismo bloque lleguen al mismo punto.
Una pregunta que puede surgir ahora es:
Cómo hacer para comunicar o coordinar hilos de distintos bloques o iguales bloques?
Podemos sincronizar las hilos de igual o distinto bloque a través de las distintas memorias que tenemos disponible. En CUDA tenemos registros, memoria compartida, memoria global, memoria constante.
Para determinar a que tipo de memoria pertenece una variable declarada. Sólo hay que anteponerle alguna de las siguientes palabras claves:

1. __shared__
2. __constant__
3. __device__
4. Ninguna de las anteriores para lograr tener registros que sólo están en memoria específica de cada hilo.

Las variables que tiene en su declaración la palabra __shared__ son variables que tienen como scope todo el bloque, es decir todos los hilos de ese bloque comparten esta variable que es una sola (lo cual es ideal para sincronizar hilos de un mismo bloque). La duración de esta variable está dado por lo que dure la función kernel. Un detalle importante a tener en cuenta con esta memoria es que es más pequeña que la memoria global pero es más rápido de acceder.
Las variables que tiene la palabra clave __constant__ son variables que son vistas por todos los hilos de todas las grillas, deben ser declaradas fuera del cuerpo de cualquier función kernel. Su duración es la duración de la aplicación entera.
Las variables que en su declaración tengan la palabra clave __device__ solamente son variables ubicadas en la memoria global, vistas por todos los hilos de todas las grillas. Su duración es la duración de la aplicación entera.
Las variables que están declaradas en el cuerpo de una función kernel sin ninguna palabra clave serán registros de los hilos.

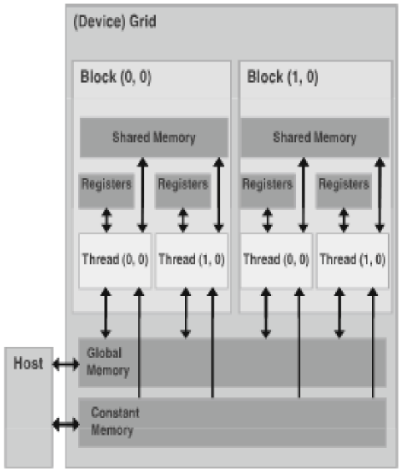


Figura 8: Estructura Grilla en una GPU

6. Implementación

En esta sección explicaremos más en detalles la implementación del algoritmo de cálculo de cápsulas convexas presentado al principio de este trabajo.
Para empezar, como la idea es utilizar una placa de video nvidia para realizar todas aquellas tareas que requieran mucha capacidad de cómputo, donde dicha placa puede residir o no en la computadora donde ejecutamos nuestra aplicación. Necesitaremos implementar una comunicación con un servidor (en el cual reside una placa de video nvidia), subir datos a dicho servidor, compilar un binario en dicho servidor, ejecutar el binario y finalmente traer los resultados de nuevo a nuestra máquina local. Por esta razón implementaremos un script en bash que se encargará de subir a un servidor que contenga una placa de video nvidia los siguientes datos:

1. Una imagen en formato ppm
2. Un archivo txt que contendrá los valores de los píxeles a buscar dentro de la imagen anterior. Este archivo forma parte de un parámetro de la aplicación.
3. Conjunto de archivos para compilar un binario en el servidor

Una vez subidos los datos antes mencionados, la siguiente etapa de la aplicación consta en compilar un binario en el servidor ingresado por el usuario y ejecutarlo. Lo que hace este binario en el servidor es buscar en una imagen ingresada por el usuario como parámetro de la aplicación, pixeles que coincidan en su valor con alguno contenido en el archivo de texto (que también ingreso como parámetro el usuario). Al resultado de esa búsqueda (que se realiza utilizando la capacidad de cómputo de la placa de video) se le calcula la cápsula convexa. A dicha cápsula convexa obtenida como resultados se la guarda en una imagen llamada cápsula.ppm, también se guardan dos archivos mas que son el log de la aplicación, y el resultados de la bbúsqueda realizada.
Una vez que el binario compilado terminó su ejecución, la siguiente etapa de la aplicación, consta en traer todos los archivos generados por el binario en la etapa anterior para que se puedan analizar los resultados.

Como podemos notar, hay dos partes en la aplicación bien marcadas, una es el script de bash que interactua con el usuario, sube los datos necesarios a un servidor y pone a correr en dicho servidor a un binario que se encargará de aplicar el algoritmo de cálculo de cápsulas convexas presentados al principio de este documento.

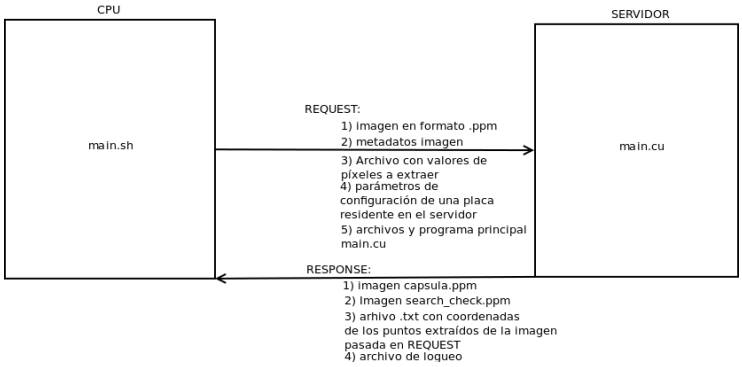


Figura 9: Descripción

Ahora nos enfocamos en el binario que se compila en un servidor ingresado por el usuario, ya que el trabajo mas grande se encuentra en dicho código fuente. Como se menciono antes, el binario implementa el algoritmo de cálculo de cápsulas convexas que se explica al principio de este documento. Aquí detallaremos la forma de proceder del binario para que el lector pueda entender con mas facilidad el código.
El código del binario es implementado usando el lenguaje C/C++.Este binario, toma como parámetro el archivo que contiene valores de píxeles en el formato $[x, y, z]$ con $x,\ y,\ z\ \in [0, ..255]$. También toma los metadatos de la imagen fuente, como el alto,ancho y cantidad de canales dentro de la imagen.
Luego lo que éste binario realiza en primera instancia es abrir la imagen y cargarla en memoria, procedimiento que lo logra a través de una librería externa denominada cutil. La imagen es cargada en memoria en un arreglo de una dimensión. Luego el siguiente paso del binario es buscar los píxeles de interés, que son todos los píxeles dentro de la imagen fuente, cuyo valor este contenido en el archivo de texto previamente mencionado. Para esto se copia el arreglo donde esta ubicada la imagen fuente, a la memoria de una GPU. Allí se lanza una grilla que buscará de forma paralela todos los píxeles de interés. Luego todos los índices de los píxeles de interés son almacenados por los hilos de la grilla que realiza la búsqueda en otro arreglo. Este último arreglo una vez finalizada la búsqueda es copiado a memoria de CPU. Con éste último arreglo copiado en memoria de la CPU, tenemos los índices de los píxeles de interés. Lo que se hace luego es generar las coordenadas cartesianas de los puntos de interés en base a los índices obtenidos del resultado de la búsqueda. Las coordenadas cartesianas son almacenadas en un vector de pares de flotantes. En En este último tipo de dato de C/C++ tenemos las coordenadas cartesianas almacenadas. Este vector de pares, se encuentra encapsulado dentro de una clase llamada “set”.
La clase set es la clase principal de este binario ya que encapsula todas las operaciones sobre un conjunto de puntos. A nosotros nos interesa sólo un método, que es justamente el que calcula la cápsula convexa. Ese método dentro de la clase set llamado get_convex_hull, es el más importante y su forma de proceder es la siguiente:

1. En primera instancia se copian los puntos que están en un vector de pares de flotantes a un arreglo de flotantes. éste arreglo tiene un tamaño de 3

veces la cantidad de puntos que hay. La razón es que además de guardar las dos coordenadas cartesianas, contiene una tercera componente inicializada en cero para indicar que este punto no es es parte de la cápsula convexa.

2. Luego se calcula cual es la mejor grilla para operar dentro del conjunto de puntos almacenados en el arreglo anterior.

3. Una vez logrados los pasos anteriores, se copian los puntos a memoria de GPU y se llama a una función que lanza una grilla y ejecuta el filtro según el algoritmo que se presentó al principio de este documento. ésta función pondrá un 1 en la tercera componente de cada punto si dicho punto pasa el filtro.

4. Luego del paso anterior los puntos almacenados en el arreglo mencionado en el primer punto contendrán puntos de la forma:
$[x_1, y_1, z_1] \dots [x_n, y_n, z_n]$, *con* $z_i \in [0, 1]$. Dónde el valor 1 de la componente z_i indica que el punto paso el filtro.

5. Una vez pasado el paso anterior, tenemos en el arreglo del primer paso un conjunto de puntos más reducido, o sea sólo tenemos que tener en cuenta todos los puntos que tienen el valor 1 en su tercera componente (que son los que han pasado el filtro). la tercera y última función llamada convex_hull_filter (que es un método privado) computa la cápsula convexa teniendo en cuenta sólo los puntos que han pasado el filtro, o sea, como dijimos anteriormente, sólo se tiene en cuenta los puntos que tiene el valor 1 en su tercera componente. Esta última función calculará la cápsula convexa según el algoritmo presentado al principio y colocará un valor 2 en la tercera componente de todos los puntos que forman la cápsula convexa.

6. Por último lo que hace el binario es generar una imagen llamada capsula.ppm con todos los puntos que tienen el valor 2 en su tercera coordenada. El método privado que realiza dicha tarea es save_points.

Con esta forma de proceder explicada, el lector podrá entender el código con mas facilidad.

6.1. Diseño

El diseño del binario que calcula la cápsula convexa utilizando una GPU es:

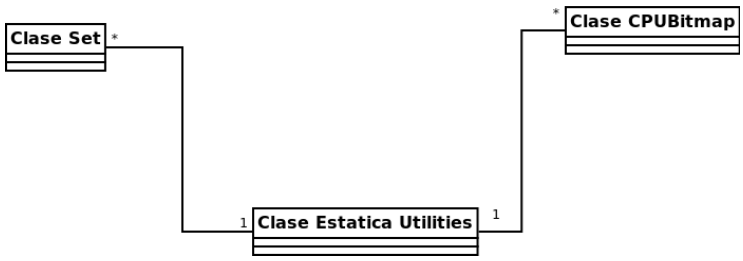


Figura 10: Diseño UML del binario que calcula la cápsula convexa

6.2. Código

El algoritmo de cálculo de cápsulas convexas implementado en C/C++ tiene su principal estructura en las siguientes funciones:

```
1
2  unsigned char *Set::get_convex_hull(regions reg,
3                                      int w,
4                                      int h,
5                                      int chanels,
6                                      int grid_max,
7                                      int block_max,
8                                      int make_union_flag){
9
10         int k=0;
11         int arg3 = 0;
12         int arg6 = 0;
13
14         std::ofstream p;
15         unsigned char *hull_set = NULL; //Contendra el resultado
16         float *dev_ordered_pixels_by_x_axis = NULL;
17         int *dev_regs    = NULL;
18         time_t mytime;
19
20         float *ordered_pixels_by_x_axis = NULL;
21         int *regs = NULL;
22         int in = 0;
23
24         mytime = time(NULL);
25         printf("START TIME: ");
26         printf(ctime(&mytime));
27         printf("\n");
28
29         cudaProfilerStart();
30
31         if(points.size() == 0){
32                 return hull_set;
33         }
34
35         //Init
36
37         ordered_pixels_by_x_axis = (float*)calloc(points.size()*chanels,
38                                                   sizeof(float));
39
40         cudaMalloc((void**)&dev_ordered_pixels_by_x_axis,
41                                    points.size()*chanels*sizeof(float));
42
43         hull_set = (unsigned char*)calloc(w*h*4,sizeof(unsigned char));
44
```

```
45          memset((void*)ordered_pixels_by_x_axis,
46                  -1,
47                  points.size()*chanels*sizeof(float));
48
49
50
51
52          p.open("puntos.txt", ios::out);
53          if (!p.is_open()){
54                  std::cout << "ERROR: Al abrir archivo para esc" << endl;
55                  return 0;
56          }
57
58          //Ubicando puntos en arrelgo para poder usar en GPU
59          for(mapFloat::iterator j = points.begin(); j!=points.end();j++){
60
61                  ordered_pixels_by_x_axis[k] = j->first;
62                  ordered_pixels_by_x_axis[k+1]= j->second;
63                  ordered_pixels_by_x_axis[k+2] = 1;//Punto extremo
64                  k = k + 3;
65
66          }
67          std::cout << std::endl;
68          p.close();
69
70          //Ubicando puntos en GPU
71          printf("CPU>>>Copiando puntos en  GPU\n\n");
72          CUDA_SAFE_CALL(cudaMemcpy(dev_ordered_pixels_by_x_axis,
73                                    ordered_pixels_by_x_axis,
74                                    points.size()*3*sizeof(float),
75                                    cudaMemcpyHostToDevice));
76
77
78          //Alocando regiones
79          regs = (int*)calloc(reg.size()*2,sizeof(int) );
80          CUDA_SAFE_CALL(cudaMalloc(&dev_regs, reg.size()*2*sizeof(int)));
81
82          //Ubicando regiones en arreglo para poder usar en GPU
83
84          for(regions::iterator r = reg.begin(); r !=reg.end();r++){
85                  regs[in+1]= r->second;
86                  regs[in]  = r->first;
87                  in        = in +2;
88          }
89          std::cout << std::endl;
90
91          //Copiando regiones
92          printf("Copiando regiones\n\n");
93          CUDA_SAFE_CALL(cudaMemcpy(dev_regs,
94                                    regs,
```

```
95                                  reg.size()*2*sizeof(int),
96                                  cudaMemcpyHostToDevice));
97
98
99
100         if(points.size() > 2){
101
102                 arg3 = reg.size();
103                 arg6 = points.size();
104
105
106                 hull(ordered_pixels_by_x_axis,
107               dev_ordered_pixels_by_x_axis,
108                         arg6,
109                         dev_regs,
110                         arg3,
111                         grid_max,
112                         block_max);
113
114
115                 /*Valores inicales*/
116
117
118                 arg6 = arg6*chanels;
119                 arg3 = arg3*2;
120                 convex_hull_filter(ordered_pixels_by_x_axis,
121                                    dev_ordered_pixels_by_x_axis,
122                                         arg6,//Es multiplicado por 3 arriba
123                                         dev_regs,
124                                         arg3, //Es multiplicado por dos arriba
125                                         grid_max,
126                                         block_max
127                                    );
128
129
130
131         }
132         //Guardando puntos
133         save_points(ordered_pixels_by_x_axis,
134             dev_ordered_pixels_by_x_axis,
135                     points.size(),
136                     dev_regs,
137                     arg3,
138                     hull_set,
139                     w,
140                     make_union_flag,
141                     THREAD_MAX,
142                     BLOCK_MAX);
143
144         //Clean up
```

```
145         CUDA_SAFE_CALL(cudaFree(dev_regs));
146         dev_regs = NULL;
147         free(regs);
148         regs = NULL;
149         free(ordered_pixels_by_x_axis);
150         ordered_pixels_by_x_axis = NULL;
151         mytime = time(NULL);
152         printf("END TIME: ");
153         printf(ctime(&mytime));
154 
155         cudaProfilerStop();
156 
157         return hull_set;
158 }
```

Como se puede ver esta función además de tomar los argumentos de la imagen. toma otros parámetros para poder operar sobre el hardware que permite hacer operaciones en paralelo.
Como habíamos visto antes, el algoritmo de cálculo de cápsulas convexas consiste en aplicar un filtro en primera instancia para obtener en lo posible puntos extremos o un conjunto mas reducido que los contenga y luego aplicar otro procedimiento para encontrar finalmente la cápsula convexa. En este código la función que ejecuta el filtro se llama hull() y su cuerpo es el siguiente:

```
1 
2 static void hull(float *set,
3                   float *dev_set,
4                   int set_size,
5                   int *dev_regs,
6                   int regions_size,
7                   int grid_max,
8                   int block_max){
9 
10 
11 
12 
13         //Calculo de la grilla adecuada
14 
15         dim3 block(THREAD_MAX,1,1);
16         dim3 grid(BLOCK_MAX,1,1);
17 
18 
19         //Algoritmo que calcula los puntos extremos
20         extreme_points<<<grid,block>>>(dev_set,
21                                        set_size,
22                                        dev_regs,
23                                        regions_size
24                                        );
25 
26         CUDA_SAFE_CALL(cudaDeviceSynchronize());
```

```
27
28          //Copiando resultado a CPU
29          CUDA_SAFE_CALL(cudaMemcpy(set,
30                                    dev_set,
31                                    3*set_size*sizeof(float),
32                                    cudaMemcpyDeviceToHost));
33
34  }
```

La función que calcula la cápsula convexa una vez aplicado el filtro que obtiene los puntos extremos es convex_hull_filter() y tiene como cuerpo:

```
1   static void convex_hull_filter(float *set,
2                                  float *dev_set,
3                   int set_size,
4                   int *regions,//Memory on GPU
5           int regions_size,
6                   int block_max,
7                   int thread_max
8                   ){
9
10  int j = 0;
11  int min_x_cord;
12  int max_x_cord;
13  int min_y_cord;
14  int max_y_cord;
15  int current_r = 0;
16  int current_l = 0;
17
18  int res = 0;
19  int res_izq = 0;
20  int center_x_coord = 0;
21  int center_y_coord = 0;
22  int *dev_res =  NULL;
23  int *dev_res_izq = NULL;
24  int *dev_index  = NULL;
25  int *dev_center_x_coord = NULL;
26  int *dev_center_y_coord = NULL;
27  int *dev_current_r = NULL;
28  int *dev_current_l = NULL;
29
30
31
32  /*Obteniendo cuatro puntos extremmos*/
33  min_x_cord = get_min_x_cord(set,set_size);
34  max_x_cord = get_max_x_cord(set,set_size);
35  min_y_cord = set[1];
36  max_y_cord = set[set_size-2];
37
38  //INICIALIZANDO
```

```
39 current_r = 0;//En cero se encuentra el minimo punto de la capsula convexa
40 current_l = 0;
41
42 center_x_coord = set[0]; //Desvalanseo la b\'usqueda de convexidad
43 center_y_coord = set[1];
44
45
46
47 CUDA_SAFE_CALL(cudaMalloc(&dev_res, sizeof(int)));
48 CUDA_SAFE_CALL(cudaMalloc(&dev_res_izq, sizeof(int)));
49 CUDA_SAFE_CALL(cudaMalloc(&dev_index, sizeof(int)));
50 CUDA_SAFE_CALL(cudaMalloc(&dev_center_x_coord, sizeof(int)));
51 CUDA_SAFE_CALL(cudaMalloc(&dev_center_y_coord, sizeof(int)));
52
53
54 CUDA_SAFE_CALL(cudaMalloc(&dev_current_l, sizeof(int)));
55 CUDA_SAFE_CALL(cudaMalloc(&dev_current_r, sizeof(int)));
56
57
58
59 CUDA_SAFE_CALL(cudaMemcpy(dev_center_x_coord,
60                           &center_x_coord,
61                           sizeof(int),
62                           cudaMemcpyHostToDevice));
63
64 CUDA_SAFE_CALL(cudaMemcpy(dev_center_y_coord,
65                           &center_y_coord,
66                           sizeof(int),
67                           cudaMemcpyHostToDevice));
68
69
70 for(j = 0;j < set_size; j = j+3 ){
71
72         if(set[j+2] == 1){
73
74                 if((set[j+1] == min_y_cord)
75                         || (set[j+1] == max_y_cord)
76                         || (set[j] == min_x_cord)
77                         || (set[j] == max_x_cord) ){
78
79                         set[j+2] = 2;//Punto de la capsula convexa
80
81                         if(set[j] >= set[current_r]){
82                                 current_r = j;
83                         }else if((set[j] <= set[current_l])){
84                                 current_l = j;
85                         }
86                 }else{
87
88                         CUDA_SAFE_CALL(cudaMemcpy(dev_res,
```

```
89                                          &res,
90                                          sizeof(int),
91                                        cudaMemcpyHostToDevice));
92
93                    CUDA_SAFE_CALL(cudaMemcpy(dev_res_izq,
94                                          &res_izq,
95                                          sizeof(int),
96                                        cudaMemcpyHostToDevice));
97
98                    CUDA_SAFE_CALL(cudaMemcpy(dev_index,
99                                          &j,
100                                         sizeof(int),
101                                       cudaMemcpyHostToDevice));
102
103                   CUDA_SAFE_CALL(cudaMemcpy(dev_current_l,
104                                         &current_l,
105                                         sizeof(int),
106                                       cudaMemcpyHostToDevice));
107
108                   CUDA_SAFE_CALL(cudaMemcpy(dev_current_r,
109                                         &current_r,
110                                         sizeof(int),
111                                       cudaMemcpyHostToDevice));
112
113                   dim3 bl(thread_max,1,1);
114                   dim3 gri(block_max,1,1);
115
116
117                           set[j],set[j+1],
118                           set[current_l],set[current_l+1],
119                           set[current_r],set[current_r+1]);
120
121                   belong_to_convex_hull<<<gri,bl>>>(dev_set,
122                                                     dev_index,
123                                                     dev_res,
124                                                     dev_res_izq,
125                                                  dev_center_x_coord,
126                                                  dev_center_y_coord,
127                                                     dev_current_r,
128                                                     dev_current_l,
129                                                     regions,
130                                                     regions_size,
131                                                     set_size
132                                            );
133
134                   CUDA_SAFE_CALL(cudaMemcpy(&res,
135                                         dev_res,
136                                         sizeof(int),
137                                       cudaMemcpyDeviceToHost));
138
```

```
                        CUDA_SAFE_CALL(cudaMemcpy(&res_izq,
                                                  dev_res_izq,
                                                  sizeof(int),
                                                cudaMemcpyDeviceToHost));


                        if((res == 0) && (res_izq != 0)){
                                set[j+2] = 2;
                                current_r = j;

                        }else if((res_izq == 0) && (res != 0)){
                                set[j+2] = 2;
                                current_l = j;
                        }else if((res == 0) && (res_izq == 0) ){
                                set[j+2] = 2;
                                if(set[j] > center_x_coord){
                                        current_r = j;
                                }else{
                                        current_l = j;
                                }
                        }
                        res = 0;
                        res_izq = 0;
                }
        }
}

//Clean up
CUDA_SAFE_CALL(cudaFree(dev_res));
CUDA_SAFE_CALL(cudaFree(dev_index));
CUDA_SAFE_CALL(cudaFree(dev_center_x_coord));
CUDA_SAFE_CALL(cudaFree(dev_center_y_coord));



dev_res = NULL;
dev_index = NULL;
dev_center_x_coord = NULL;
dev_center_y_coord = NULL;


}
```

Esta última función como vimos contiene el llamado a otra función importante que aplica el algoritmo para saber si un punto pertenece a la cápsula convexa o no. Esa función es:

```

__global__ static void belong_to_convex_hull(float *set,
```

```
                                        int *index,
                                        int *res,
                                        int *res_izq,
                                        int *center_x_coord,
                                        int *center_y_coord,
                                        int *current_r,
                                        int *current_l,
                                        int *regions,
                                        int size_regions,
                                        int set_size){


int my_region = (blockIdx.x*blockDim.x*2) + threadIdx.x*2;
int start = regions[my_region];
int end = regions[my_region + 1];
int k = 0;
float a = 0.0;
float b = 0.0;
float rect_value = 0.0;
int i = 0, j = 0;


for(k = start*CHANELS; k < end*CHANELS; k = k+3 ){

        if( set[k+2] > 0 ){

                if(set[*index] == set[*current_r]){

                        if(set[k] > set[*index]){
                                *res = 1;
                        }
                }else{
                        j = *index;
                        i = *current_r;

                        //Recta
                        a = (float)(-set[j+1] + set[i+1]);
                        b = (float)(set[j]*(-set[i+1]));
                        b = (float)(b - set[i]*(-set[j+1]));
                        a = a/(float)(set[j] - set[i]);
                        b = b/(float)(set[j] - set[i]);
                        rect_value = a*set[k] + b;

                        if((rect_value < 0) && (-set[k+1] > rect_value)
                                && (-(set[k+1] + rect_value) > EPSIL0N)
                                && (set[(*index)] > set[*current_r])){

                                *res = 1;

```

```
53                          }else if((rect_value < 0) &&
54                                  (-set[k+1] < rect_value) &&
55                                  (set[k+1] + rect_value  > EPSIL0N)
56                                  && (set[*index] < set[*current_r])){
57
58                                  *res = 1;
59                          }
60
61                  }
62                  if(set[*index] == set[*current_l]){
63
64                          if(set[k] < set[*index]){
65                                  *res_izq = 1;
66                          }
67                  }else{
68                          j = *index;
69                          i = *current_l;
70
71                          //Recta
72                          a = (float)(-set[j+1] + set[i+1]);
73                          b = (float)(set[j]*(-set[i+1]));
74                          b = (float)(b - set[i]*(-set[j+1]));
75                          a = a/(float)(set[j] - set[i]);
76                          b = b/(float)(set[j] - set[i]);
77                          rect_value = a*set[k] + b;
78
79                          if((rect_value < 0) && (-set[k+1] > rect_value)
80                                  && (-(set[k+1] + rect_value) > EPSIL0N)
81                                  && (set[*index] < set[*current_l])){
82                                          *res_izq = 1;
83
84                          }else if((rect_value < 0) &&
85                                  (-set[k+1] < rect_value) &&
86                                  (set[k+1] + rect_value > EPSIL0N) &&
87                                  (set[*index] > set[*current_l])){
88                                          *res_izq = 1;
89                          }
90                  }
91          }
92
93  }
94
95  }
```

6.3. Análisis Empírico

En esta sección estudiaremos el orden del algoritmo de cálculo de cápsula convexas que se presentó al principio de este documento. Realizaremos el análisis empírico o experimental. Para varios tamaños de entrada veremos cuanto tiempo

le lleva computar la capsula a nuestro algoritmo. El cómputo se realiza en un servidor con 16461112 KiB de RAM, 8 cores y una placa de video nvidia Tesla k40.
A través de la siguiente tabla mostramos los tiempo obtenidos:

Resolución imagen	**Cantidad de puntos**	**tamaño entrada en MB**	**Tamaño bloquex-Cant hilos por bloque**	**Tiempo en segundos**
400x300	6180	0,070724487 MB	49x128	0.037562 seg
602x401	9408	0,107666016 MB	37x256	0.048438 seg
284x279	20578	0,235496521 MB	80x256	0.097639 seg
293x400	26756	0,30619812 MB	105x256	0.103763 seg
1000x1000	47670	0,545539856 MB	94x512	0.251681 seg
334x535	49612	0,567764282 MB	97x512	0.251817 seg
640x320	102649	1,174724579 MB	100x1024	0.549431 seg
428x600	120339	1,377170563 MB	117x1024	1.211706 seg
857x2000	723621	8,28118515 MB	64x1024	15.733267 seg

Como podemos notar hay algunos saltos en el tiempo de aproximadamente 4 veces cuando se pasa de un tamaño de entrada de 0,567764282 MB a otro de 1,174724579 MB y del doble cuando se pasa de un tamaño de entrada de 1,174724579 MB a otro de 1,377170563 MB. Por dichos saltos que se repiten en diversas muestras podemos ver que el orden del algoritmo será cuadrático.

6.4. Comparación con la versión secuencial

En esta sección realizaremos comparaciones con la versión secuencial del algoritmo de calculo de cápsulas convexas, y lo analizaremos con un profiler. Para empezar mediremos el tiempo que tarda en ejecutarse la función get_convex_hull cuando lo ejecutamos con una grilla de tamaño igual a 1 y con una grilla de un tamaño mucho mas grande que uno. Las comparaciones se realizaron en un servidor con sistema operativo Debian 3.13.10-1 (2014-04-15) x86_64 GNULinux, 8 CPUś, 16461112 KiB RAM y una placa aceleradora nvidia Tesla k40. A través de la siguiente tabla mostramos las comparaciones :

Resolució imagen	**Cantidad de puntos**	**tamaño entrada en MB**	**Tamaño bloque**	**Tiempo paralelo**	**Tiempo secuencial**
400x300	6180	0,070724487 MB	49x128	0.037562 seg	3.025768 seg
602x401	9408	0,107666016 MB	37x256	0.048438 seg	5.336587 seg
284x279	20578	0,235496521 MB	80x256	0.097639 seg	10.368445 seg
293x400	26756	0,30619812 MB	105x256	0.103763 seg	17.718184 seg
1000x1000	47670	0,545539856 MB	94x512	0.251681 seg	63.354965 seg
334x535	49612	0,567764282 MB	97x512	0.251817 seg	55.791456 seg
640x320	102649	1,174724579 MB	100x1024	0.549431 seg	63.415893 seg
428x600	120339	1,377170563 MB	117x1024	1.211706 seg	135.073684 seg
857x2000	723621	8,28118515 MB	64x1024	15.733267 seg	40 min 23 seg

Para obtener información de las funciones mas utilizadas de nuestro código lo analizamos con el profiler de nvidia:

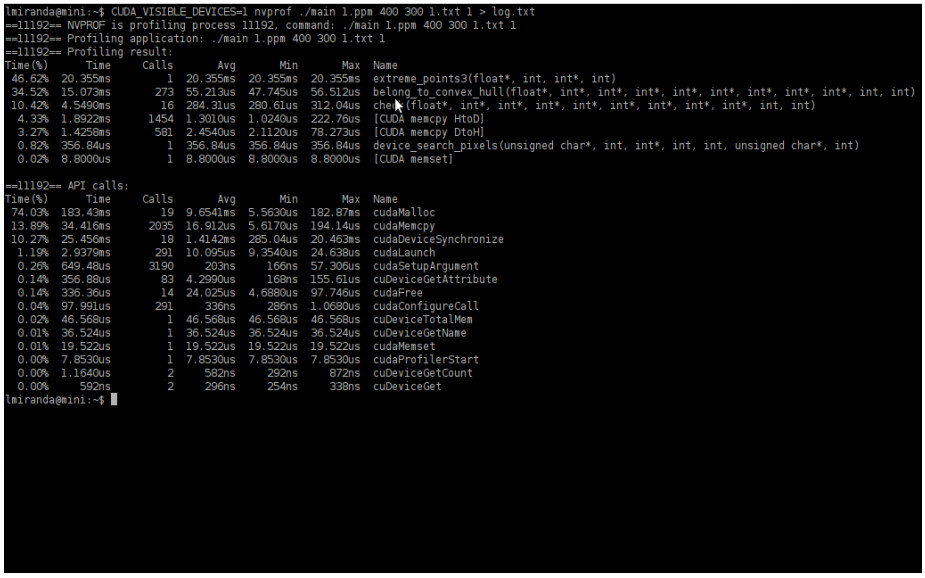


Figura 11: Análisis con profiler nvidia

Como se puede ver éste informe nos cuenta cuales son las funciones mas ejecutadas en nuestro algoritmo, para este caso las funciones más llamadas son 'belong_to_convex_hull' y 'cudaMemCpy'. Esto es coherente siendo que la primera función sirve para determinar si un punto va a formar parte de la cápsula convexa y por ende se llama para cada punto que se quiera analizar, para nuestro caso sería la mayoría de los puntos de entrada. Por otra parte la función cudaMemCpy sirve para copiar datos de la memoria de nuestra CPU a la memoria de la GPU y se utiliza principalmente para copiar a memoria de GPU cada punto que se quiere analizar junto con los pivots actualizados, procedimiento que se lleva a cabo para cada punto que se quiere determinar si pertenece o no a la cápsula convexa. Por lo cual podemos decir que es coherente el análisis que nos esta dando esta herramienta.

7. Resultados

En esta sección presentamos resultados obtenidos con la implementación del algoritmo de cálculo de cápsulas convexas antes presentado. Para eso presentaremos una serie de imágenes para de las cuales se extraen puntos para luego calcularle su cápsula convexa. Los puntos que se extraen de una imagen son los que se muestran de color verde. Estas imágenes utilizadas son las imágenes que se utilizaron para realizar la comparación con la versión secuencial.

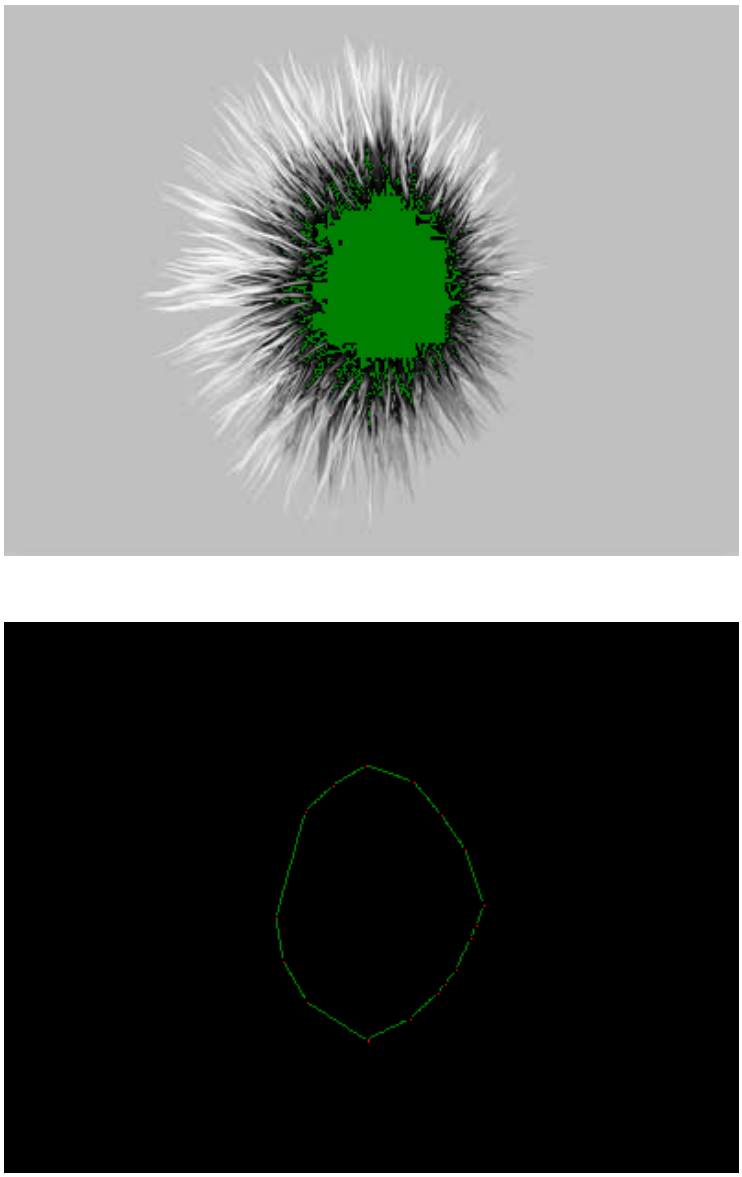


Figura 12: Resultado

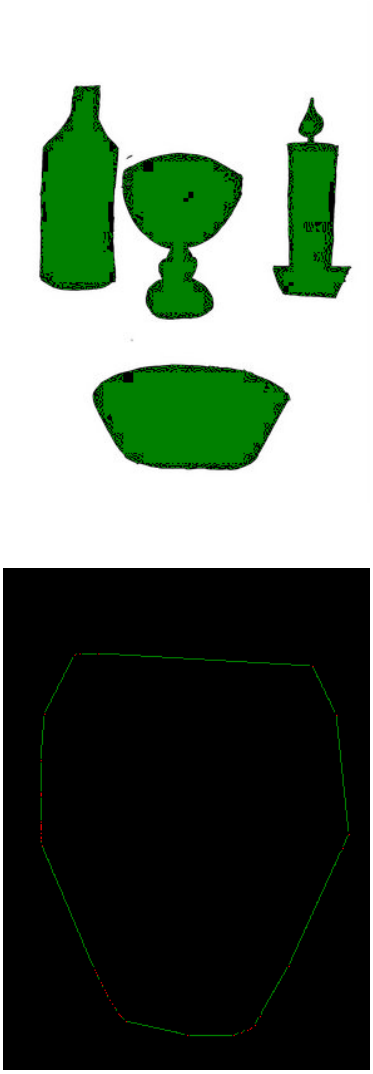


Figura 13: Resultado

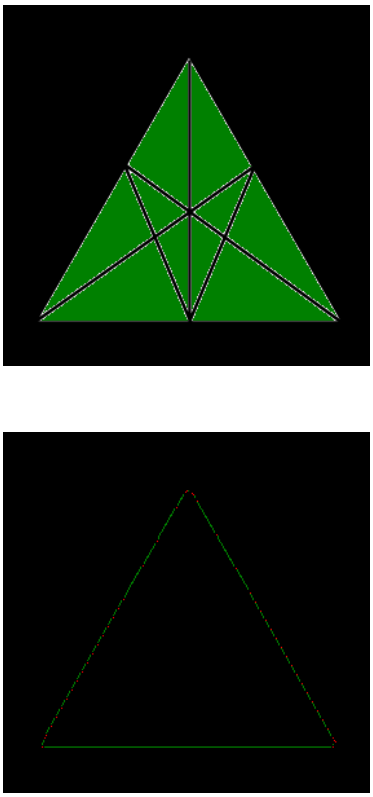


Figura 14: Resultado

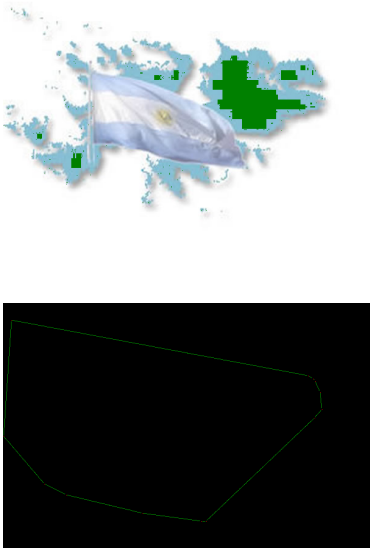


Figura 15: Resultado

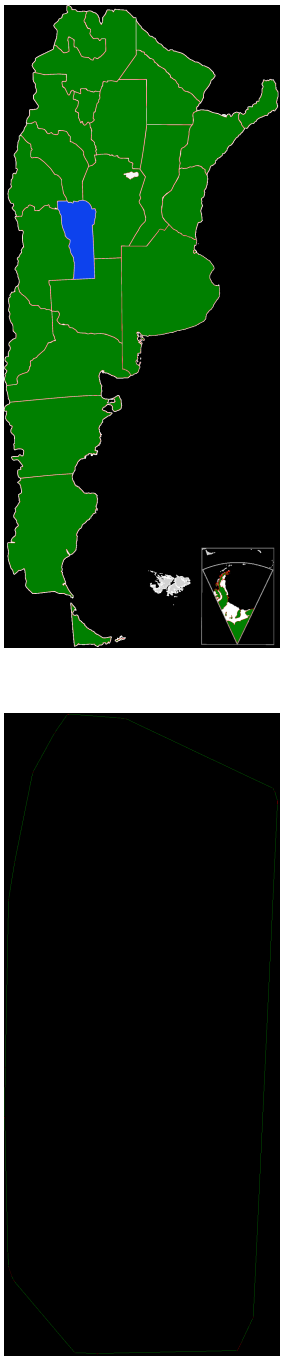


Figura 16: Resultado

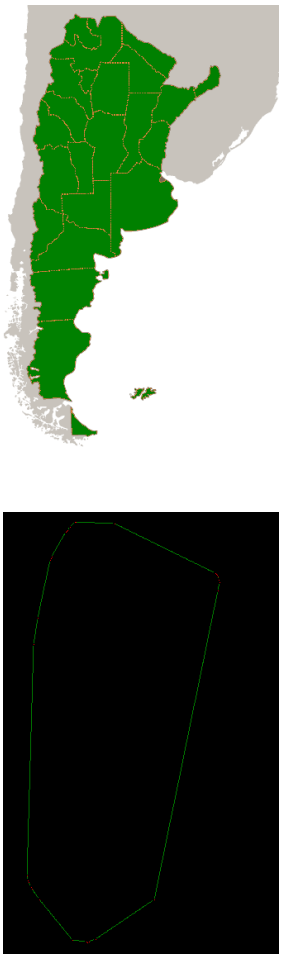


Figura 17: Resultado

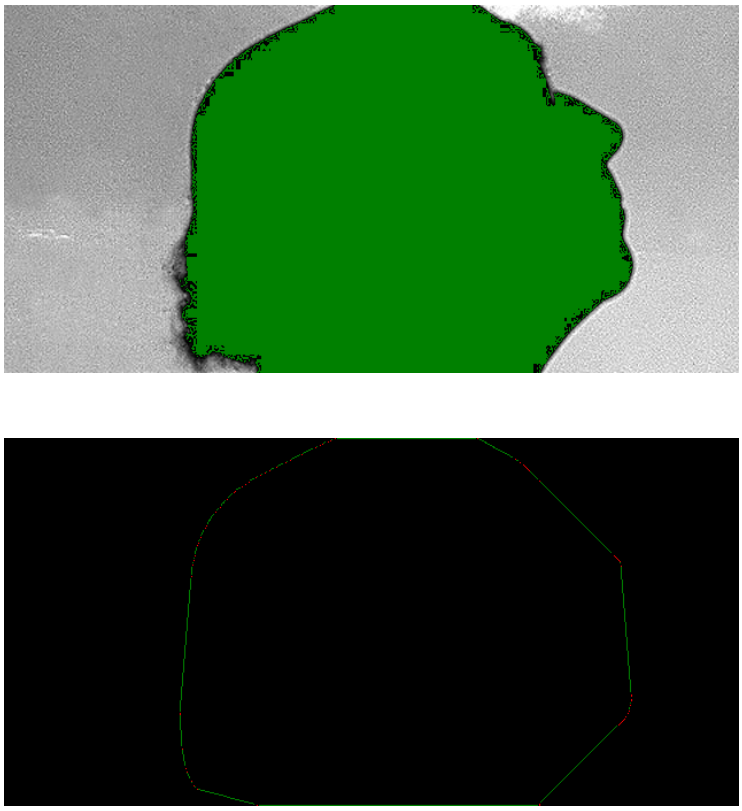


Figura 18: Resultado

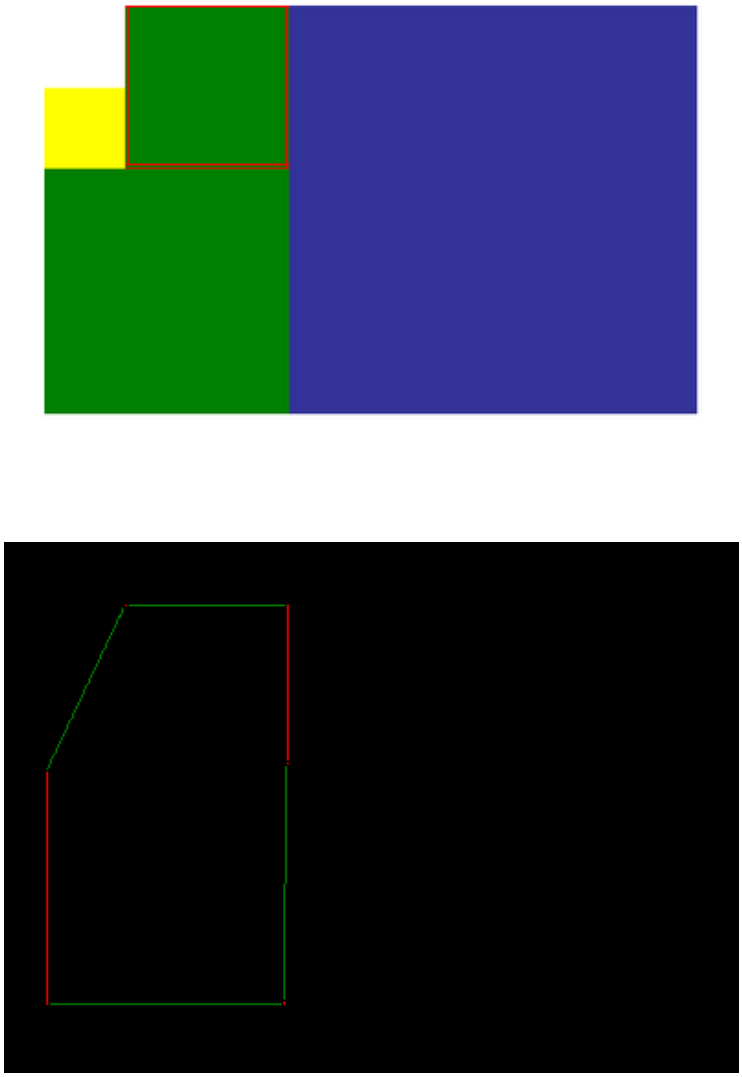


Figura 19: Resultado

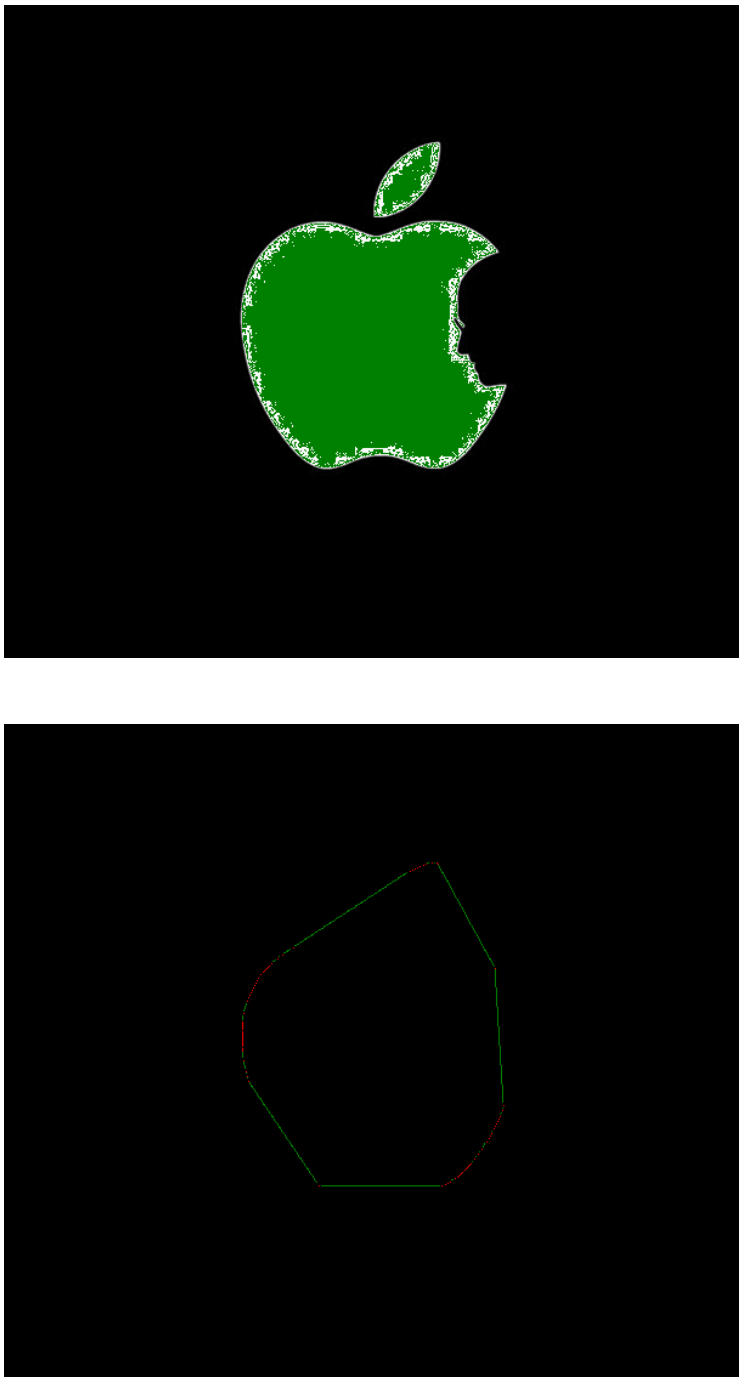


Figura 20: Resultado

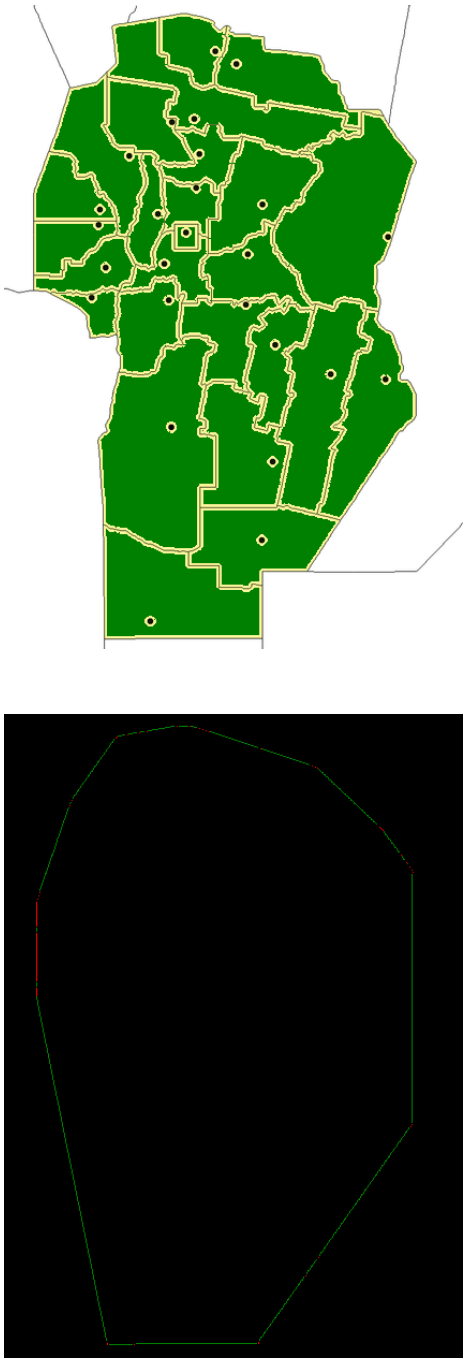


Figura 21: Resultado

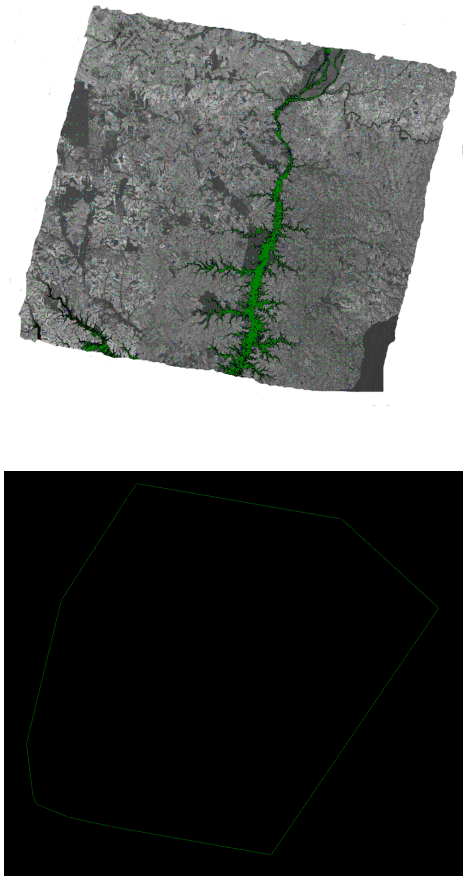


Figura 22: Resultado

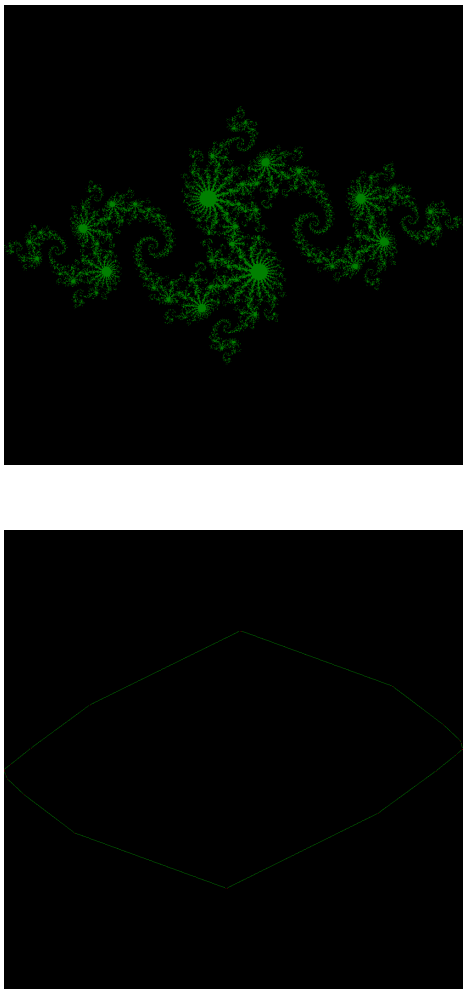


Figura 23: Resultado

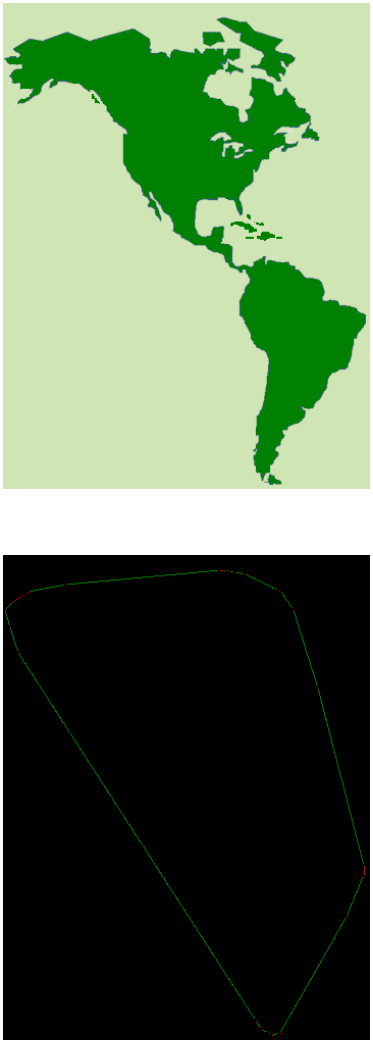


Figura 24: Resultado

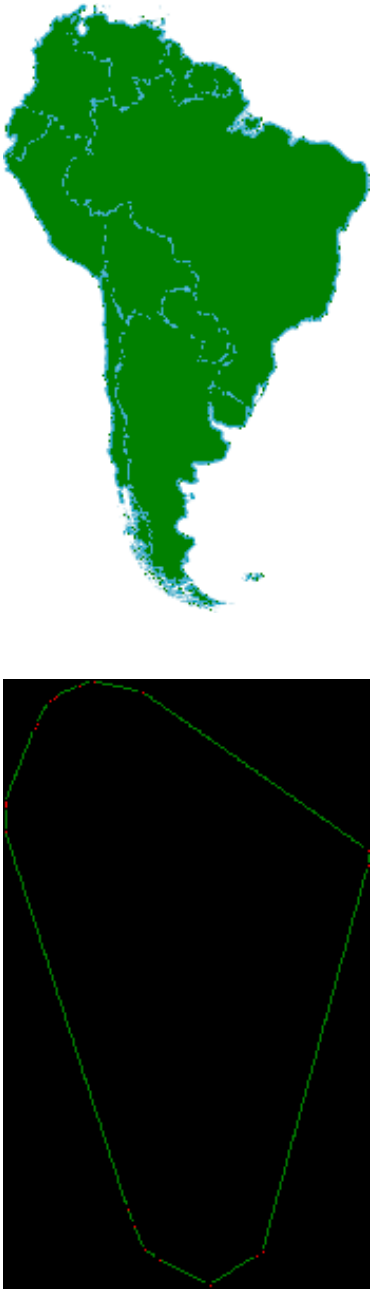


Figura 25: Resultado

8. Mejoras

1. Calcular la ocupación con el objetivo de eliminar parámetros de la aplicación.
2. Permitir múltiples formatos.
3. Balanceo de la carga de puntos extras en búsquedas y otras operaciones.

9. Bibliografía

1. 'Cuda by example' - Jason Sanders/Edward Kandrot
2. 'Programming Massively parallell processors' - David B. Kirk/Wen Mei W. Hwu
3. 'Algorithms Secuencial and Parallel a Unified Approach' - Russ Muller/Laurence Boxer
4. 'An Investigation of Graham Scan and Jarvis March' - Chris Harrison, http://www.chrisharrison.net/index.php/Research/ConvexHull

10. Conclusiones

Para concluir el hecho de poder emplear un dispositivo especial para hacer cómputos en paralelo y desarrollar en él un algoritmo que aproveche esta capacidad, logrando resultados eficientes , me hace concluir en que resolver el problema del cálculo de cápsulas convexas encuentra su futuro, al menos en el área de teledetección, en las arquitecturas con múltiples procesadores y no con simples procesadores que puedan realizar millones de instrucciones por segundo. Esta conclusión hace hincapié en la gran agilidad que tienen ciertos dispositivos con múltiples procesadores para operar de manera interconectada con grandes cantidades de datos.