UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

OPTIMIZACIÓN DE PROCESO DE DETECCIÓN DE PARTÍCULAS A PARTIR DE IMÁGENES DE VIDEO MEDIANTE PARALELIZACIÓN

MEMORIA PARA OPTAR AL TITULO DE INGENIERO CIVIL EN COMPUTACIÓN

JUAN SEBASTIÁN SILVA LEAL

PROFESORES GUÍAS:
NICOLÁS MUJICA FERNÁNDEZ
NANCY HITSCHFELD KAHLER

MIEMBRO DE LA COMISIÓN:
JAVIER BUSTOS JIMENEZ

SANTIAGO DE CHILE
ABRIL 2012

Resumen

La detección de objetos a partir de imágenes se ha convertido en una herramienta muy poderosa para diferentes disciplinas. El Laboratorio de Materia Fuera del Equilibrio del Departamento de Física de la Facultad cuenta con una implementación en C del Método χ^2 usando bibliotecas ad-hoc compatibles con Mac OSX para detectar partículas en sistemas granulares cuasi-bidimensionales compuestos por miles de partículas de acero de 1 mm de diámetro, pudiendo detectar partículas en una imagen de 1 MegaPixel en alrededor de 10 segundos. Sin embargo, estas imágenes provienen de videos que se desean analizar y en una sesión de trabajo se puede requerir analizar alrededor de unas 100.000 imágenes en total, por lo cual el procesamiento y posterior análisis de estas imágenes de video tiene una duración de varios días. Es por esto que fue necesario agilizar de alguna manera este procesamiento de imágenes y generar una solución robusta.

El objetivo principal de la memoria consistió en reducir los tiempos de detección de partículas generando un nuevo software basado en el anterior, facilitando extensiones futuras, y utilizando el máximo poder de cómputo disponible en el laboratorio.

El alumno ideó como solución un sistema distribuido haciendo uso de todos los computadores disponibles para el procesamiento de imágenes, reimplementando el código del software, en ese entonces utilizado, de C a C++ utilizando patrones de diseño para facilitar futuras extensiones del software y threads con el fin de aumentar el rendimiento de este. También se agregó tecnología CUDA para el procesamiento de datos reduciendo de forma considerable los tiempos de ejecución.

Como resultado final de la memoria, se logró obtener un speedup de alrededor de $5x$ haciendo uso de distribución de carga computacional, uso de procesos en paralelo, hilos de ejecución y tecnología CUDA, además se logró una solución más robusta y extensible para futuros cambios o generación de nuevos algoritmos de procesamiento.

Todo el proceso de investigación, desde la obtención de datos hasta la validación de la hipótesis, lleva mucho tiempo, en donde la detección de partículas es solo una parte de todo el calculo computacional que se debe realizar, por lo que se aconseja implementar en lenguajes no interpretados y más rápidos, como por ejemplo C++, otras etapas de cálculo de datos y además, en lo posible, distribuir el computo y usar CUDA.

Dedicado a mi Padre Gerardo Silva

Agradecimientos

Agradezco a toda mi familia por el apoyo recibido en todos estos años de estudio, a mi Papá Gerardo Silva por preocuparse de mí en todo momento, a Andrea Guerra quien me ayudó mucho en corregir este documento y en su soporte.

Agradezco también a la profesora Nancy quien siempre da su apoyo y tiempo a todos sus alumnos, a Nicolás Mujica por su indiscutida confianza en el desarrollo de esta memoria, a Gustavo Castillo quien me ayudo completamente en todo el transcurso del desarrollo ofreciendo soporte, conocimientos y tiempo para ayudar a comprender y entender el problema y la solución necesaria, también agradezco a Cristóbal Navarro por su tiempo y dedicación en tratar de ayudarme a resolver una parte del desarrollo en CUDA.

Agradezco de todas maneras el financiamiento por parte del proyecto Anillo de Investigación en Ciencia y Tecnología *"Dynamics of disordered and heterogeneous media, ACT127"* por medio de Nicolas Mujica y la compra del equipo de trabajo que se utilizo en el desarrollo e implementación de la solución gracias al proyecto Fondecyt Nº 1090188: "On the criticality of the solid-liquid-like transition in vibrated quasi-two-dimensional granular media".

Índice

Índice de tablas.

Índice de Figuras

Capítulo 1

Introducción

La detección de objetos a partir de imágenes se ha convertido en una herramienta muy poderosa para diferentes disciplinas como la Física, Biología, Astronomía, entre otras. Las técnicas son variadas en complejidad y en velocidad. La elección de una u otra depende de la forma del objeto, la calidad de la imagen y de la densidad de objetos a detectar en una imagen.

1.1. Contexto

Dos ejemplos de algoritmos de detección de partículas son: "Detección por máximos de intensidad"[1] y "Detección por ajuste de partícula ideal mediante la convolución de la imagen experimental con una imagen de partícula teórica", en adelante Método χ^2 [1]. Ambos algoritmos se encuentran implementados mayormente en $MatLab$ por sus autores.

El Laboratorio de Materia Fuera del Equilibrio del Departamento de Física de la Facultad trabaja con una implementación en C del Método χ^2, modificado para detectar partículas en sistemas muy densos en donde el Método χ^2 original no podía detectar muchas de ellas, todo esto usando bibliotecas ad-hoc, en adelante software $PTrack$ [2]. El objetivo es detectar partículas en sistemas granulares cuasi-bidimensionales compuestos por miles de partículas de acero de 1 mm de diámetro[Figura 2], pudiendo detectar todas o la mayoría de éstas partículas en una imagen de 1 MegaPixel en alrededor de 10 segundos, usando un solo computador para el procesamiento. Sin embargo, estas imágenes provienen de videos que se desean analizar, así en un video pueden existir alrededor de 3.000 imágenes y en una sesión de trabajo se puede requerir analizar alrededor de unas 100.000 imágenes en total, por lo cual el procesamiento y posterior análisis de estas imágenes de video tiene una duración de varios días.

Estas imágenes son tomadas a través de una cámara de alta velocidad y alta resolución emplazada encima del contenedor de partículas mediante un trípode. El contenedor se retro-ilumina tornando las partículas de color negro[Figura 1] y el fondo de color blanco; tales imágenes son transferidas al computador por red. El experimento a rasgos generales consiste en hacer vibrar las partículas en determinadas frecuencias y ver que les sucede, por ejemplo, por que se forman grupos densos de partículas.

[1] Ver más en http://physics.georgetown.edu/matlab/

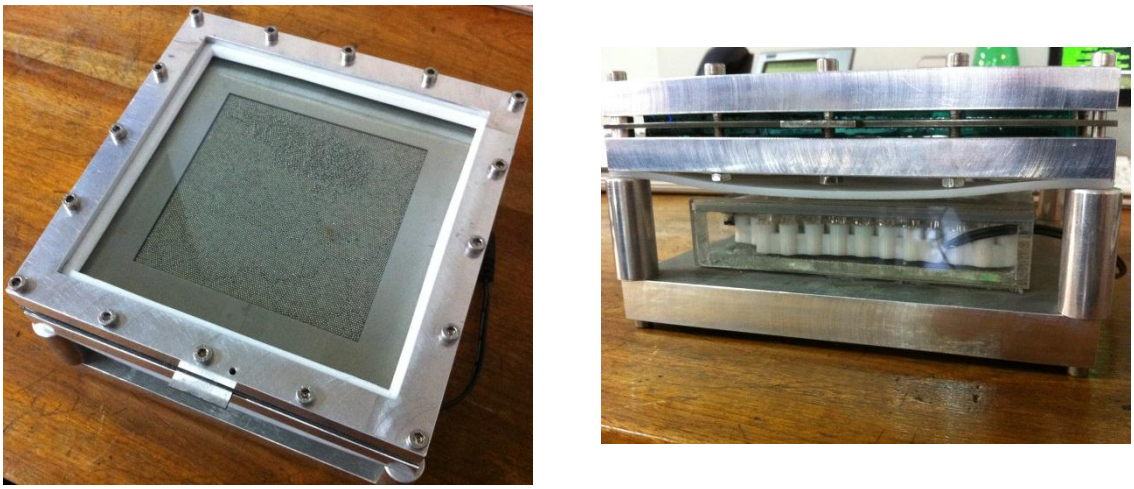


Figura 1: Contenedor de partículas y retro-iluminación.

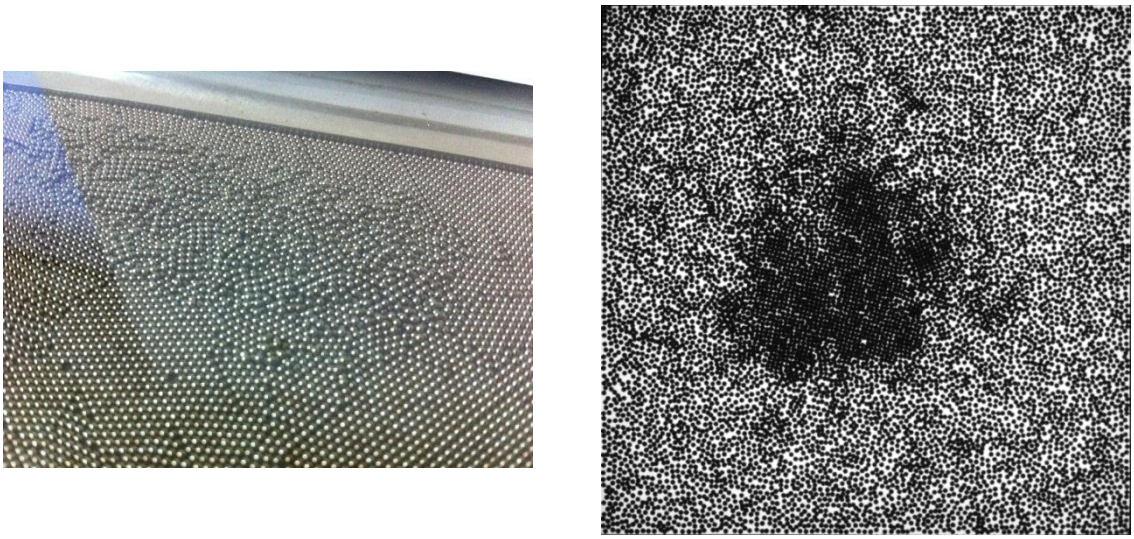


Figura 2: Sistema granular cuasi-bidimensional.

1.2. Problema

El software $PTrack$ generalmente se ejecuta dentro de funciones de MATLAB por parte del investigador. Para tratar de acelerar los cálculos, se lanzan varias instancias de MATLAB procesando videos diferentes dejando el computador realizando cálculos todo un fin de semana. Es dentro de estas instancias que al procesar ciertas imágenes, $PTrack$ deja de funcionar y se detiene el procesamiento de imágenes produciéndose un error en MATLAB, generando una detención en los cálculos hasta que el investigador a inicio de semana inspeccionaba el resultado de los cálculos y al ver el error debía lanzar el procesamiento de imágenes desde el punto donde se había producido el error.

Todo lo anterior causa mucha pérdida de tiempo para el investigador que debe estar constantemente revisando el progreso del procesamiento para evitar detenciones prolongadas de los cálculos, produciendo también una incerteza de la completitud de los resultados y un gasto de tiempo enorme para obtener los datos.

1.3. Motivación

Se desea reducir al máximo el tiempo de procesamiento de datos con el fin de poder procesar y analizar lo más pronto posible y así utilizar de buena manera el tiempo del investigador. Además se desea eliminar los errores durante el procesamiento y si es que los hay, estos sean automáticamente solucionados y así poder tener los resultados completos al momento de querer verlos.

Por otra parte el software $PTrack$ original, creado por Mauricio Cerda se basaba en el Método χ^2 original el cual se desempeñaba bien para configuraciones de densidades bajas, luego tanto $PTrack$ como el Método χ^2fue modificado posteriormente por Scott Waitukaitis para poder procesar imágenes con alta densidad de partículas las cuales no se detectaban bien con el Método χ^2 original. Así $PTrack$ quedó con dos opciones de procesamiento de imágenes, una para imágenes normales y otra para imágenes con alta densidad de partículas. Esta última se ocupaba generalmente para el procesamiento de imágenes, así el nuevo diseño de software debía soportar creación de nuevos algoritmos y una fácil modificación de éstos.

1.4. Objetivo General

El objetivo principal de la memoria consiste en reducir al máximo los tiempos de detección de partículas de imágenes de video en una sesión de trabajo, permitiendo al investigador, (el cual deja procesando las imágenes y posteriormente recupera las posiciones de las partículas de cada una de las imágenes) tener los resultados, en la medida de lo posible, de un día para otro. También es muy deseable tener un software que pueda ser fácilmente extensible para adaptar e implementar nuevos algoritmos de detección de partículas pues el software existente ya ha sufrido cambios con éste fin y va a seguir sufriendo modificaciones en un futuro cercano.

1.5. Objetivos Específicos

Los objetivos específicos de la realización de ésta memoria son los siguientes:

- Portar el código del software original $PTrack$ a lenguaje C++.
- Implementar una arquitectura de solución utilizando patrones de diseño[3][16] y elementos que ayuden a una fácil extensión futura del software, reutilización de código y faciliten el uso del software.
- Generar una solución distribuida[11] con el fin de aprovechar todo el poder de cómputo disponible dentro del laboratorio, siendo tolerante a fallas y con una configuración relativamente sencilla.
- Aprovechar todo el poder de cómputo disponible en cada computador del laboratorio utilizando varios hilos de ejecución y varias instancias de procesamiento.
- Aprovechar la tecnología CUDA[10] para darle una gran aceleración a los cálculos[5][6][7][8][9].
- Documentar todo el desarrollo para posteriores modificaciones de software, instalación y configuración en distintos computadores y aprender su modo de uso.

1.6. Metodología

Con el fin de ir solucionando los problemas desde el día uno de trabajo y no solamente al finalizar la memoria, es que se decidió por empezar con la Distribución de cómputo haciendo uso de $PTrack$ en 3 computadores en conjunto con la ejecución de varias instancias. Así se lograron ver resultados notables desde el primer mes de la realización de esta memoria. Luego se implementó la versión en C++ (ahora en adelante $PTrack2$) haciendo uso de varios threads para procesar ciertas funciones y ayudando a la optimización mediante Flags de compilación que aumentaron el rendimiento notablemente. Finalmente, usando un computador nuevo solicitado por el alumno, se agregó a $PTrack2$ la tecnología CUDA mediante el desarrollo de una biblioteca bien encapsulada para así tocar lo menos posible la última versión de $PTrack2$.

1.7. Contenido de la memoria

Dentro del Capítulo 2 se explican la mayoría de los conocimientos necesarios para entender el resto del documento y el desarrollo de ésta memoria. Dentro de los conocimientos necesarios se explica que es la tecnología CUDA incluyendo conceptos de programación asociados a conceptos físicos propios del hardware, modelo de programación, ejemplo de código y algunos conceptos técnicos propios de la evolución de ésta tecnología. Por otra parte se explica que son los Patrones de diseño orientados a objetos y algunos patrones que se utilizaron para el desarrollo de la memoria. Finalmente se explica la teoría e implementación del algoritmo de reconocimiento de partículas utilizado y su posterior modificación y objetivo de la alteración.

El Capítulo 3 contiene una especificación del problema a resolver y las causas de estos problemas, se describe además la arquitectura del software antiguo $PTrack$, también se describen las alternativas de solución ideadas y las escogidas

La solución implementada se describe dentro del Capítulo 4 en donde se explica que cambios sufrió $PTrack$ antes de ser remplazado. Se presenta la arquitectura de harware utilizada, la arquitectura de software ideada, el software $PTrack2$ en detalle utilizando diagramas de clases con su arquitectura, estructuras de datos importantes y funcionamiento. También se describe el sistema distribuido realizado en conjunto con su funcionamiento y configuración. Se describen también algunos scripts importantes para el uso casual del software y la biblioteca desarrollada para trabajar usando CUDA con su diagrama de clases y estructuras de datos. Finalmente se describen los elementos de ayuda al desarrollo como la descripción propia de cada función y los elementos de ayuda al usuario como la autocompletación y wiki.

El Capítulo 5 contiene los resultados de la solución desarrollada y su validación, esto contempla la descripción de una plataforma de pruebas, la metodología usada y sus resultados validando los resultados por medio de exactitud y resultados estadísticos equivalentes.

Finalmente el Capítulo 6 contiene las conclusiones de todo el desarrollo de la memoria describiendo logros, resultados y trabajos futuros.

Capítulo 2

Antecedentes

Para la realización de esta memoria se tienen en cuenta varios conceptos y tecnologías relativamente nuevas que se explicarán en este capitulo.

2.1. Tecnología CUDA[10]

El poder de cómputo general existente dentro de cada computador, laptop e incluso celulares de hoy en día se basa en el uso de múltiples núcleos de procesamiento en un solo chip llamado CPU o *Central Processor Unit* por sus siglas en inglés. Agregar múltiples núcleos generalmente aumenta el rendimiento sin tener la necesidad de aumentar la frecuencia con que se trabaja generando un menor consumo energético y una temperatura de trabajo menor. Existe una variada gama de arquitecturas de CPU y la más común es la llamada x86 creada por la compañía *Intel*.

Sin embargo la adición de múltiples núcleos de procesamiento no es algo actual pues mucho antes de que se masificase para uso en computadores, ésta ya existía dentro de las tarjetas de video, elemento fundamental para desplegar en una pantalla información, sin embargo no podía usarse para realizar cálculos generales y se encontraba orientada solamente al despliegue de datos en pantalla, acelerando el dibujo de gráficos en 2 y 3 dimensiones y haciendo uso de todo tipo de filtros de imagen y render. Con la masificación de juegos de video que requerían una gran aceleración de video y elementos visuales de gran complejidad es que este tipo de procesadores llamados GPU o *Graphics processing unit* fue tomando gran importancia y aumentando exorbitantemente el poder de cálculo de éstos. Es así como desde el año 2008 se crearon dos tipos de tecnología para aprovechar el poder de cómputo de las GPU para realizar cálculos generales, CUDA para tarjetas de video Nvidia, y OpenCL iniciada por Apple, apta para cualquier tipo de tarjeta de video.

La tecnología CUDA hace posible la programación de funciones de tipo general usando la GPU como unidad de procesamiento y facilitando el desarrollo gracias a una extensión del lenguaje de programación C llamada *C para CUDA,* así para un programador familiarizado con este lenguaje, no le va a resultar extraña la forma de programar. Además, Nvidia alienta y apoya la adopción de ésta tecnología sacando al mercado servidores de cómputo llamados TESLA diseñados para cómputo de alto desempeño pudiendo entregar un rendimiento de hasta 10^2 veces para aplicaciones de variada índole.

[2] Según sitio http://www.nvidia.com/object/tesla_computing_solutions.html

2.1.1. Conceptos de programación

CUDA introduce varios conceptos de programación asociados a su arquitectura dentro de los cuales se distingue ***Device***, ***Host*** y ***Kernel***.

i. Device

A la unidad de procesamiento grafica o GPU se le llama Device o dispositivo. Puede haber tantos dispositivos como tarjetas de video tenga el computador distinguiéndose cada una por un identificador, nombre, poder de cómputo, etc. Además se puede seleccionar uno u otro para realizar cálculos y almacenar memoria.

ii. Host

A la unidad central de procesamiento se le denomina Host, es aquí donde se ejecuta cualquier código escrito normalmente por el programador, sin restricciones y como complemento al código ejecutado por en dispositivo.

iii. Kernel

Los Kernel son fragmentos de código que se ejecutan en paralelo dentro del dispositivo. Solo un kernel es ejecutado a la vez y muchos threads ejecutan el código del kernel. La principal diferencia entre los threads de CUDA y los de CPU, son que los threads de CUDA son extremadamente ligeros, requieren muy pocos recursos computacionales para su creación y pueden ser conmutados rápidamente. Otro punto es que CUDA ocupa miles de threads para obtener eficiencia mientras que CPUs de múltiples núcleos solo pueden usar unos pocos.

iv. CUDA Threads

Un kernel de CUDA es ejecutado por un arreglo, matriz o cubo de threads, donde cada thread corre el mismo código y se diferencian por un identificador o threadID, el cual se usa para procesar direcciones de memoria y tomar decisiones de control.

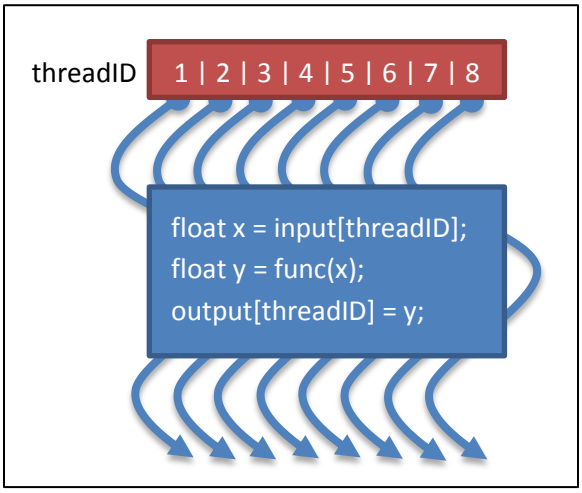


Figura 3: Arreglo de threads de CUDA.

v. *CUDA Block*

Un bloque de CUDA son las dimensiones de threads que son ejecutados, así éste puede ser un arreglo, una matriz o un cubo. Un bloque se ejecuta hasta que el último thread termine, luego la GPU es libre de ejecutar un nuevo bloque. En la Figura 4 se aprecian las distintas formas que puede tener un bloque. Cada número representa un thread dentro del bloque.

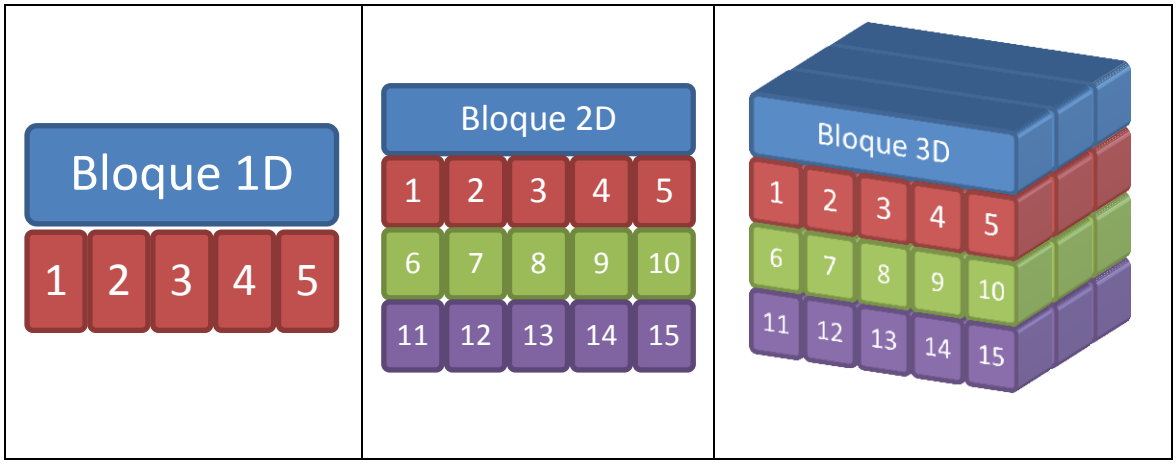


Figura 4: Bloques de threads de CUDA, 1 2 y 3 Dimensiones.

vi. *CUDA Grid*

Una grilla o cuadrícula son las dimensiones de los bloques que son ejecutados, éste puede ser un arreglo o una matriz.

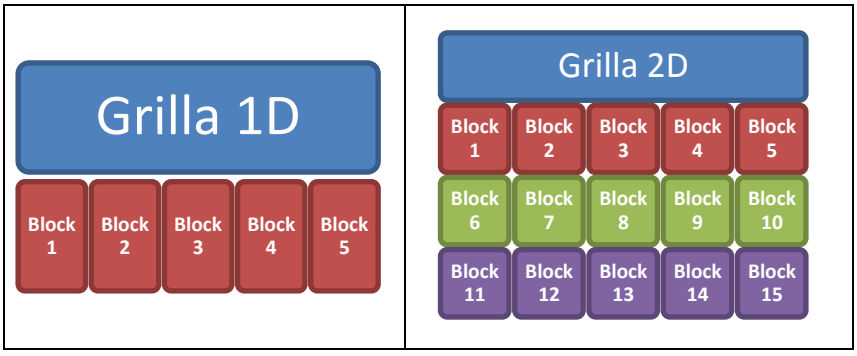


Figura 5: Grilla de bloques de CUDA, 1 y 2 Dimensiones.

La siguiente ilustración refleja las opciones de ejecución de un kernel dentro de un dispositivo. Cada bloque posee un identificador de acuerdo a la posición en la grilla, así también un Thread.

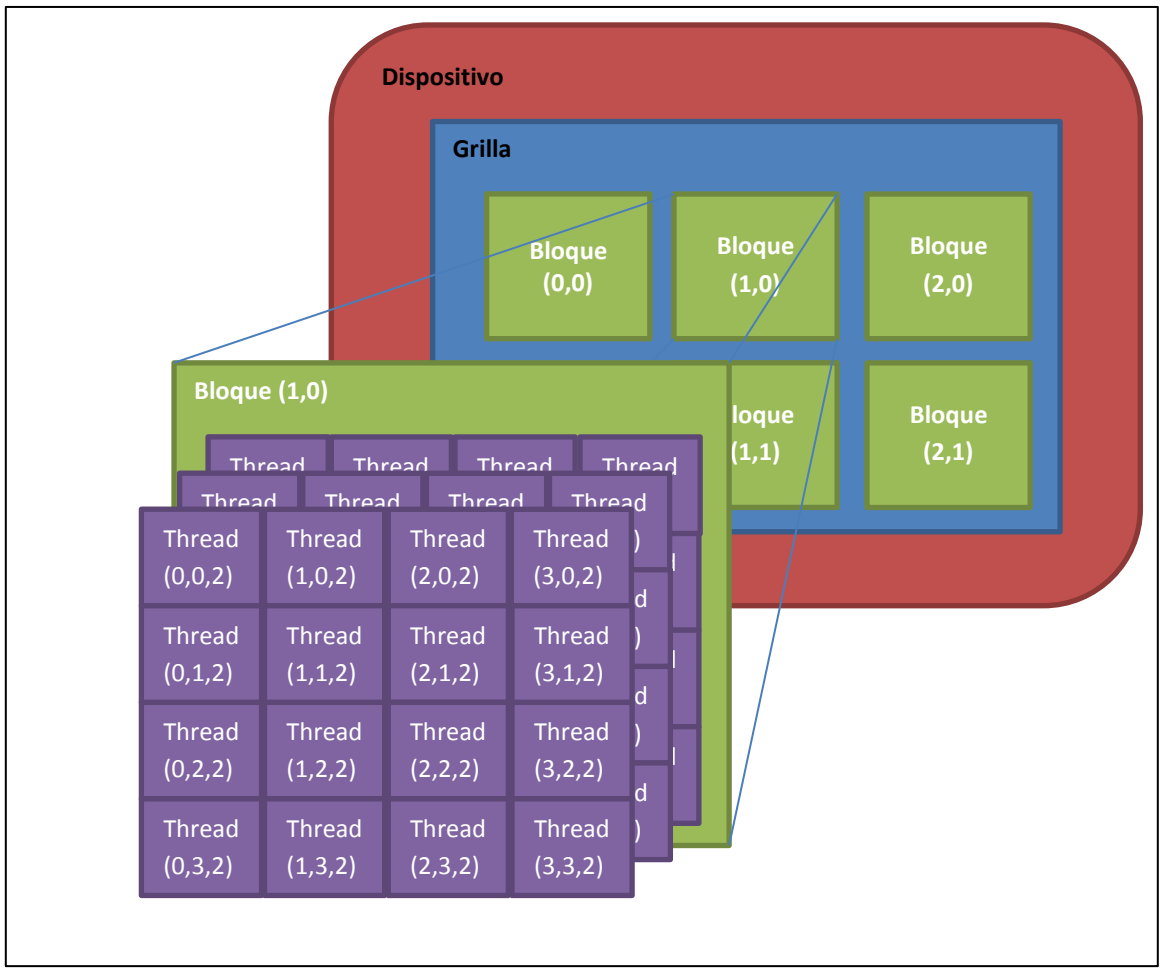


Figura 6: Dimensiones de una grilla, bloque y threads dentro de un dispositivo.

vii. Registros

Los registros son los elementos más pequeños de memoria dentro de la arquitectura, se asignan por thread y tienen un ciclo de vida igual al thread.

viii. Memoria local

Al igual que los registros, la memoria local es asignada a cada thread y respetan el ciclo de vida del mismo, sin embargo este tipo de memoria se encuentra físicamente en la memoria DRAM del dispositivo.

ix. Memoria compartida

La memoria compartida es asignada a cada bloque de threads y se encuentra físicamente en la tarjeta de video. Esta memoria tiene un ciclo de vida igual al ciclo de un bloque y cada thread puede leer y escribir en ella posibilitando la cooperación de threads dentro de un bloque (Ver Figura 7).

x. Memoria global (dispositivo)

Esta memoria es accesible por todos los threads al igual que el Host, la vida útil de este tipo de memoria respeta las reservas y liberación de memoria y es controlado por el programador.

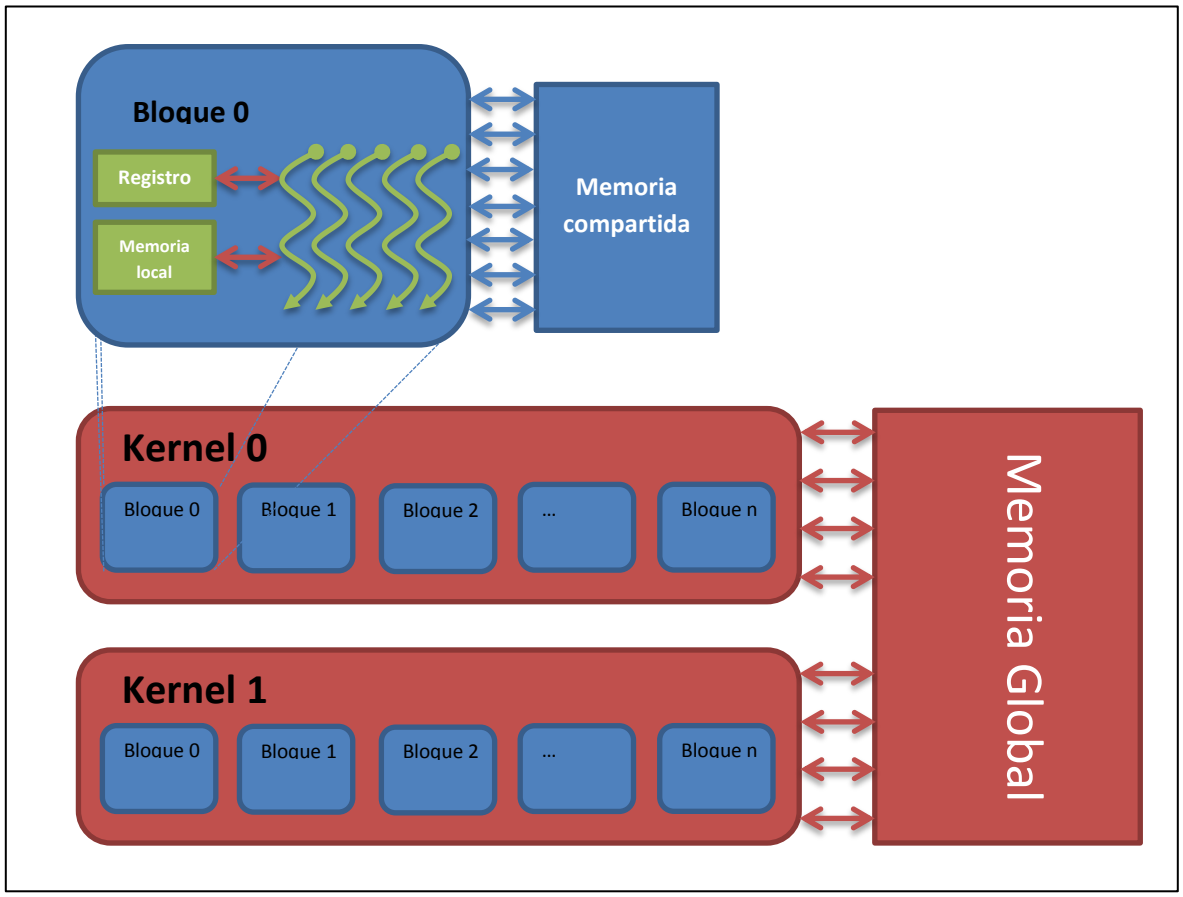


Figura 7: Modelo de memoria en CUDA.

2.1.2. Modelo de programación

CUDA establece un modelo de programación basado en la idea de Kernel, Grilla, Bloques y Threads. Un kernel es ejecutado por una cuadrícula la cual contiene varios bloques, los bloques son un conjunto de threads que pueden cooperar entre sí compartiendo datos vía memoria compartida y sincronizar su ejecución. Threads de diferentes bloques no pueden cooperar entre sí. Este punto es importante pues dependiendo de la complejidad del problema es que pueden ser de utilidad pocos bloques de gran tamaño o varios bloques de tamaño pequeño, teniendo el control el programador para ello. Además, cada función programada en CUDA se debe compilar usando el compilador propio llamado **NVCC** el cual se enlaza con **GCC4.4**[3], versión soportada hasta la fecha de término del desarrollo de la memoria.

[3] Compilador de código en lenguaje C y C++.

i. Ejecución de código en GPU

Los Kernel son funciones en C con algunas restricciones, éstas son:

- Solo se puede acceder a memoria de GPU.
- Debe tener retorno de tipo void.
- No se aceptan argumentos variables (varargs).
- No se acepta recursión.
- No se aceptan variables estáticas.

Los argumentos de la llamada a función son automáticamente copiados desde la memoria de la CPU a la memoria de la GPU.

ii. Calificadores de función

Un Kernel debe establecer una calificación de la función que representa e indicar desde donde se puede invocar la función. Así estos calificadores son los siguientes:

- **__global__** : Este tipo de función se invoca desde el interior del código del host y no puede ser llamado desde el código del dispositivo. Solo puede retornar void.
- **__device__** : Este tipo de función puede ser llamada desde otras funciones dentro del código del dispositivo y no puede ser llamada dentro del código del host.
- **__host__** : Solo puede ser ejecutada y llamada dentro del host.

Los calificadores **__host__** y **__device__** pueden ser combinados usando sobrecargas de función y el compilador generará código tanto para CPU como para GPU.

iii. Ejemplo de código

El siguiente ejemplo representa la suma de dos vectores, el resultado se almacena en otro vector.

```
// Código en dispositivo
__global__ void VecAdd(float* A, float* B, float* C, int vec_size){
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        if (i < vec_size)
                C[i] = A[i] + B[i];
}

// Código en Host
int main() {
        …
        // Invocar el Kernel con 256 threads por bloque
        int threadsPerBlock = 256;
        int blocksPerGrid = (N + threadsPerBlock – 1) / threadsPerBlock;
        VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, vec_size);
        …
}
```

2.1.3. Conceptos Técnicos

La Tecnología CUDA tiene sus inicios en el año 2008 y desde entonces ha ido evolucionando incorporando nuevas funcionalidades, nuevas tecnologías de hardware y elementos útiles para la programación. En este sentido se definió un sistema que representa las funcionalidades que están presentes y pueden ser utilizadas en una tarjeta de video con esta tecnología, este sistema se llama ***Compute Capability***[13](pág. 170) y las funcionalidades para cada versión se definen en la siguiente tabla:

Soporte para funcionalidades	Compute Capability			
	1.1	1.2	1.3	2.x
Funciones atómicas sobre enteros operando en palabras de 32-bits en memoria global.	Si			
atomicExch() operando en valores de punto flotante de 32-bits en memoria global.				
Funciones atómicas sobre enteros operando en palabras de 32-bits en memoria compartida.	No	Si		
atomicExch() operando en valores de punto flotante de 32-bits en memoria compartida.				
Funciones atómicas sobre enteros operando en palabras de 64-bits en memoria compartida.				
Función de votación Wrap.		No	Si	
Operaciones de doble precisión de punto flotante.				
Funciones atómicas sobre enteros operando en palabras de 64-bits en memoria compartida.				
Adición atómica de punto flotante operando en palabras de 32-bits en memoria global y compartida.				
_ballot()		No		Si
_threadfence_system()				
_syncthreads_count()				
_syncthreads_and()				
_syncthreads_or()				
Funciones de superficie				
Grilla 3D de bloques de threads				

Tabla 1: Compute Capability

2.2. Patrones de diseño orientados a objetos

En desarrollo de software, un patrón de diseño es, como bien dice su nombre, un patrón a seguir para resolver un problema común pudiendo adaptarse a otro de similares características. Este patrón tiene la cualidad de poder resolver de forma eficiente un determinado tipo de problema.

Durante el desarrollo de esta memoria, se ocuparon los siguientes patrones de diseño que fueron considerados utilizables para resolver distintos problemas que se vislumbraban durante el desarrollo:

2.2.1. Strategy

Strategy define una familia de algoritmos, encapsulando cada uno y haciéndolos intercambiables, así el algoritmo usado varia independientemente de cómo se use.

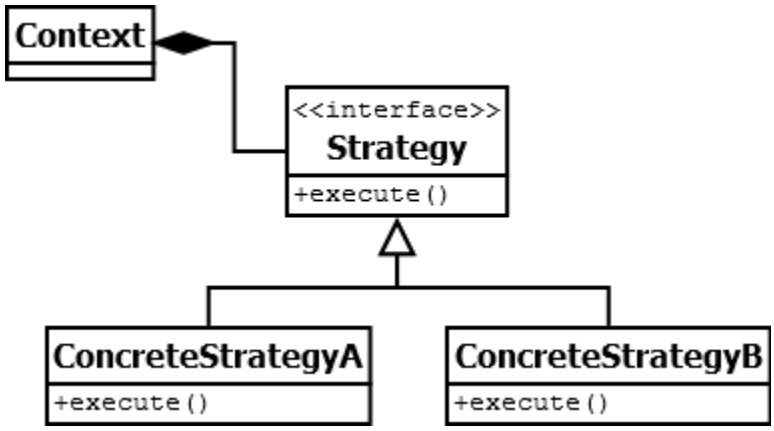


Figura 8: Diagrama de clases para patrón Strategy.

2.2.2. Factory method

Factory method define una interfaz para crear objetos, pero delega a las subclases decidir que clase instanciar.

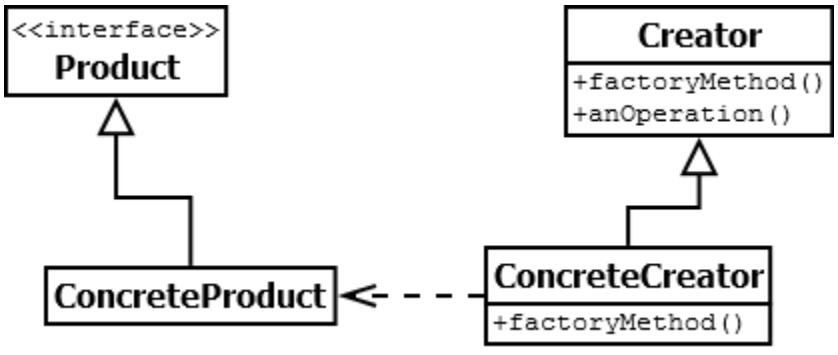


Figura 9: Diagrama de clases para patrón Factory Method.

2.2.3. Singleton

Singleton asegura que una clase tenga una y solo una instancia de ésta y provee un punto de acceso global a ella.

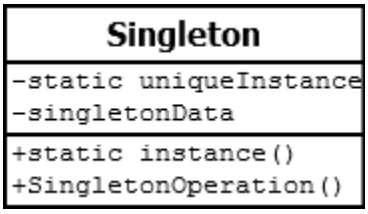


Figura 10: Diagrama de clases para patrón Singleton.

2.2.4. Object Pool

Este patrón de diseño es poco frecuente[4] y la idea es que si una instancia de una clase se puede re usar, entonces se debe evitar la creación de instancias reusando ésta. Generalmente se usa como un Singleton contenedor de datos.

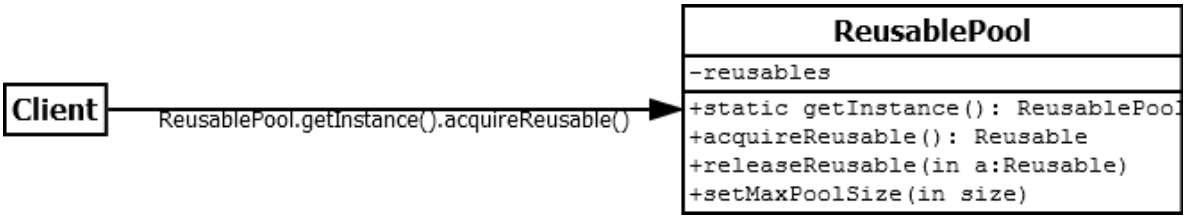


Figura 11: Diagrama de clases para patrón Object Pool.

[4] [16] http://sourcemaking.com/design_patterns/object_pool

2.3. Métricas orientadas a objetos

Para poder medir de cierta forma la pulcritud y buena codificación de un software son útiles las métricas y en especial métricas orientadas a objetos. A continuación se describen algunas métricas utilizadas.

2.3.1. Weighted Methods per Class [WMC]

Representa un indicador de esfuerzo para desarrollar y mantener una clase. La idea es que el número de métodos como la complejidad de cada uno determina la complejidad del objeto completo. Mientras mayor WMC, mayor esfuerzo de desarrollar y mantener la clase. Mientras menor WMC, mayor re-uso de la clase.

El número recomendable de atributos (variables de instancia y métodos) es 20, pero puede extenderse hasta 40. Si una clase tiene más de 40, debe revisarse. La descripción de cada método debe ocupar a lo más de una página.

2.3.2. Depth of Inheritance Tree [DIT]

Intenta representar el re-uso y el esfuerzo para entender una clase. Mientras más profunda se ubica la clase es más compleja de entender, más esfuerzo se necesita para desarrollar y mantener, por otro lado, mayor es su re-uso.

2.3.3. Coupling Between Objects [CBO]

Mide el grado de relación entre dos objetos, sin considerar la relación de herencia. Dos objetos están relacionados entre si, si cualquiera de los dos actúa sobre el otro. El acoplamiento excesivo entre objetos fuera de la jerarquía de herencia va en contra del re uso y de la modularidad. La mantención de estos objetos requiere harto esfuerzo pues dado que muchas veces se deben revisar también los objetos relacionados. Mientras más acoplamiento exista, menos modulares y re-usables serán los objetos, así también, más esfuerzo en mantener y extender el código asociado.

2.3.4. Number of Children [NOC]

Representa el número de subclases directas. Mientras mayor NOC, mayor re-uso de una clase y mayor impacto tiene dentro del desarrollo.

2.3.5. Lack of Cohesion in Methods [LCOM]

Representa el grado de similitud entre métodos, se mide en la cantidad de variables de instancia que comparten. A mayor LCOM, menor es la calidad de la clase. Este indicador va de 0 a 100.

2.3.6. Number of Methods [RFC]

Mide la complejidad de la comunicación entre componentes u objetos. A mayor RFC, mayor tiempo se destinara a depuración y prueba de código.

2.4. Algoritmo de reconocimiento de partículas

Para entender el funcionamiento del software desarrollado es necesario comprender la teoría detrás en la cual ésta basado el desarrollo. Los algoritmos implementados son ***Ajuste de mínimos cuadrados basado en convolución*** de ahora en adelante $Algoritmo\ \chi^2$ y ***Ajuste de mínimos cuadrados basado en convolución para imágenes con alta densidad de partículas*** de ahora en adelante $Algoritmo\ \chi^2 HD$.

2.4.1. Ajuste de mínimos cuadrados basado en convolución[1]

Teoría

Este algoritmo se basa en la generación de una partícula ideal a ser detectada dentro de la imagen a procesar, escrito por el Professor Mark D. Shattuck [1]. Se asumen que la forma de la imagen que se desea procesar es:

$$I_c(\vec{x}) = \sum_{n=1}^{N} I_p(\vec{x} - \vec{x}_n; D, \dots) \tag{1}$$

Donde N es el número de partículas y en éste caso la forma de la partícula se puede representar con la siguiente función:

$$I_p(\vec{x}, D, w) = \frac{\left[1 - \tanh\left(\frac{|\vec{x}| - \frac{D}{2}}{w}\right)\right]}{2} \tag{2}$$

El parámetro w determina qué tan fuerte cambia la función, en principio, w se puede relacionar al foco de la imagen. El parámetro D representa el diámetro de la partícula. Luego para determinar la posición más acertada de la partícula se usa ajuste de mínimos cuadrados definiendo la función χ^2 como sigue:

$$\chi^2(\vec{x}_0; D, w) = \int W(\vec{x} - \vec{x}_0)\left[I(\vec{x}) - I_p(\vec{x} - \vec{x}_0; D, w)\right]^2 d\vec{x} \tag{3}$$

Donde $W(\vec{x} - \vec{x}_0)$ es una función de peso. El dominio de integración está dentro del área de la imagen experimental I. Sin embargo, el dominio de $\vec{x}_0$ es mayor. En general, si el tamaño de la imagen es L_x por L_y y el tamaño de I_p es s_x por s_y, el rango de integración es $[0, L_x]$ y $[0, L_y]$ pero el rango de $\vec{x}_0$ es $[-s_x, L_x + s_x]$ y $[-s_y, L_y + s_y]$.

Cuando $\vec{x}_0$ se encuentre en la posición de una partícula, entonces χ^2 será mínimo. Por lo tanto si se minimiza χ^2 a través de $\vec{x}_0$ se puede encontrar la posición de la partícula. De hecho, habrá un mínimo en χ^2 en la posición de cada partícula. Si la imagen es exactamente igual a la representada por $I_c(\vec{x})$ entonces χ^2 será 0 para cada $\vec{x}_0 = \vec{x}_n$.

Entonces el proceso de encontrar las partículas es equivalente a encontrar todos los mínimos de χ^2 , lo cual se puede realizar de una manera fácil mediante convolución. Para esto se expande la función χ^2 como sigue:

$$\chi^2(\vec{x}_0; D, w) = \int W(\vec{x} - \vec{x}_0)\left[I(\vec{x})^2 - 2I(\vec{x})I_p(\vec{x} - \vec{x}_0; D, w) + I_p(\vec{x} - \vec{x}_0; D, w)^2\right]d\vec{x} \quad (4)$$

$$\chi^2(\vec{x}_0; D, w) = I^2 \otimes W - 2I \otimes \left(WI_p\right) + \langle WI_p{}^2\rangle \quad (5)$$

Donde:

$$f \otimes g = [f \otimes g](\vec{x}_0) = \int f(\vec{x})g(\vec{x} - \vec{x}_0)d\vec{x} \quad (6)$$

Y

$$\langle f\rangle = 1 \otimes f \quad (7)$$

El cual no es simplemente constante, sino una función de $\vec{x}_0$ ya que el dominio de integración es sobre la imagen solamente. Si W o I_p no son simétricos entonces se debe tener cuidado al evaluar la ecuación 6 (La cual es una relación cruzada) usando convolución, definido como:

$$f * g = [f * g](\vec{x}) = \int f(\vec{x})g(\vec{x}_0 - \vec{x})d\vec{x} = f(\vec{x}) \otimes g(-\vec{x}) \quad (8)$$

Existen muchas opciones para la función de peso. Si

$$W = 1;$$

Entonces

$$\chi^2(\vec{x}_0; D, w) = \int I^2 d\vec{x} - 2I \otimes I_p + \langle I_p{}^2\rangle \quad (9)$$

En este caso, el primer término no depende de $\vec{x}_0$ y el último término solo depende de $\vec{x}_0$ cerca del borde de la imagen, luego $I \otimes I_p$ será máximo para $\vec{x}_0$ en la posición de la partícula. El hecho de que relación cruzada es máxima cerca del centro es el punto de partida de muchas técnicas de detección de partículas. Esta técnica funciona bien con partículas bien separadas y con buena relación señal/ruido. Esto es debido a que la función peso es muy amplia, haciendo que el ajuste sea sensible al hecho que la partícula ideal tiene ceros en todo su alrededor, en contraste con la imagen real en donde otras partículas se encuentran alrededor. La elección de W también produce una cima muy amplio haciéndola sensible al ruido.

Para minimizar completamente χ^2 se puede encontrar una función de peso compacta que produce mejores resultados. Típicamente se usa:

$$W = I_p$$

Entonces

$$\chi^2(\vec{x}_0; D, w) = I^2 \otimes I_p - 2I \otimes {I_p}^2 + \langle {I_p}^3 \rangle \tag{10}$$

El primer término de χ^2 muestra que solamente el área de tamaño I_p alrededor de cada punto $\vec{x}_0$ es importante. El último término es constante excepto cerca de los bordes de la imagen. Cerca del borde, ésta y todos los términos se ven disminuidos debido al hecho de que hay menos y menos superposición entre la imagen experimental I y la imagen de partículas ideales I_p. Así, para normalizar este efecto en el límite, se divide todo por el último término y se define la nueva función χ^2:

$$\chi^2(\vec{x}_0; D, w) = \frac{I^2 \otimes I_p - 2I \otimes {I_p}^2}{\langle {I_p}^3 \rangle} + 1 \tag{11}$$

Lo que es igual a:

$$\chi^2(\vec{x}_0; D, w) = \frac{I^2 \otimes W - 2I \otimes WI_p}{\langle W{I_p}^2 \rangle} + 1 \tag{12}$$

Luego, la ecuación normalizada de χ^2 sigue mínima en la posición de la partícula y puede ser usada para encontrar partículas cuyo centro se encuentre fuera de la imagen.

Para encontrar las cimas dentro de la imagen se obtienen los máximos (mínimos en una imagen normalizada) locales dentro de un área determinada, teniendo un mínimo de intensidad necesaria y una separación mínima de otras cimas, todo medido en pixeles discretos. Esto da como resultado una precisión de 1 pixel.

Para obtener una mejor precisión acerca de donde se encuentran los centros de las partículas, también se puede utilizar el ajuste de mínimos cuadrados para mejorarlos. Para realizar esto, se debe ver una función χ^2 modificada como sigue:

$$\chi^2(\vec{x}_n; D, w) = \int [I(\vec{x}) - I_c(\vec{x}, \vec{x}_n)]^2 \, d\vec{x} \tag{13}$$

Donde:

$$I_c(\vec{x}, \vec{x}_n) = \sum_n W_n(\vec{x}) I_p(\vec{x} - \vec{x}_n; D, w) \tag{14}$$

Y la función $W_n(\vec{x})$ es tal que es igual a 1 dentro del volumen de Voronoi de la partícula n y cero por fuera. $W_n(\vec{x})$ se encarga de partículas que se sobreponen.

Para encontrar el centro de las partículas se desea minimizar el error producido por la diferencia entre la imagen experimental y la imagen producida por las partículas ideales encontradas, lo cual es equivalente a resolver la siguiente ecuación:

$$\frac{\delta \chi^2(\vec{x}_n^*; D, w)}{\delta \vec{x}_n^*} = 0 \tag{15}$$

Para $\vec{x}_n^*$. Debido a que se tiene una buena precisión de pixel, se puede resolver mediante el método de Newton.

Implementación [2]

A continuación se describen los pasos generales de la implementación en C de éste algoritmo.

Implementación algoritmo Chi2	
1	Normalizar imagen
2	Generar imagen de partícula ideal
3	Obtener imagen Chi2 mediante convolución
4	Obtener los mínimos locales, como lista, de la imagen Chi2 filtrando por un mínimo de intensidad, una distancia mínima.
5	Validar mínimos locales por un mínimo de separación entre ellos.
6	Generar matriz auxiliar conteniendo distancia en X e Y al mínimo más cercano y el índice del mínimo dentro de la lista
7	Calcular imagen diferencia entre imagen original normalizada e imagen generada a partir de las partículas encontradas usando la matriz auxiliar
8	Obtener error cuadrado de la diferencia
9	**Iterar N veces o hasta que el delta del error cuadrado sea menor o igual a 1:**
10	Mover partículas a su centro estimado mediante método de Newton
11	Repetir pasos 6 y 7
12	Calcular diferencia entre error cuadrado anterior y el nuevo
13	Agregar áreas de Voronoi a las partículas
14	Establecer estado de materia según área de Voronoi

Un paso importante a destacar es la obtención de los mínimos locales, el que genera las partículas detectadas. Primero se recorre toda la imagen Chi2 en busca de mínimos locales que cumplan un mínimo de intensidad de imagen y un área específica dentro de la cual son mínimos, luego éstos mínimos son validados verificando que todos tengan una distancia de separación mínima, los que no cumplen con la mínima distancia son eliminados según intensidad de imagen, es decir, el mínimo con menor intensidad de imagen es eliminado. Se le dice mínimo local pues representa un mínimo en intensidad de imagen, sin embargo la imagen que se entrega se encuentra con colores invertidos, vale decir, el negro pasa a ser blanco y el blanco a negro entonces más exactamente se debería hablar de un mínimo local de intensidad de imagen original.

2.4.2. Ajuste de mínimos cuadrados basado en convolución para imágenes con alta densidad de partículas [4]

Teoría

Para mejorar los resultados del Método χ^2 en su versión original sobre imágenes de partículas muy densas, Scott Waitukaitis extendió el algoritmo χ^2 para mejorar los resultados.

El principal problema del algoritmo original, fue que no se detectaban todas las partículas existentes en sistemas muy densos, por lo que una primera aproximación, fue generar una nueva imagen para ser reprocesada. Esta imagen se produce tomando la original y restándole la generada por el procesamiento, con lo que queda una nueva imagen con partículas no reconocidas. Así, estas nuevas partículas reconocidas en la nueva imagen son agregadas a la solución original. El problema que se observó con esta solución fue que se reconocían muchas partículas inexistentes, por lo que una solución robusta fue aplicar filtros para deshacerse del exceso de partículas detectadas. El filtro consistió en obtener la región de Voronoi[5] de cada partícula detectada y el brillo del pixel central de cada una de ellas. Una partícula "falsa" tiende a estar encerrada por un espacio más pequeño que las partículas "verdaderas", por lo tanto el área de la región de Voronoi de las partículas "falsas" es más pequeña que el área de las "verdaderas" y además, están ubicadas donde el brillo (en este caso, la oscuridad) no corresponde al centro de una partícula. Así el área de la región de Voronoi debe ser más grande que 50 $pixeles^2$(para este tipo de experimentos[6]) y el brillo del centro de la partícula debe estar dentro de 3 σ de la media (obtenido a través del Histograma y encaje Gausiano de todos los valores).

También se imponen criterios de los estados posibles de las partículas, los cuales pueden ser líquidos o solidos dependiendo del área de la región de Voronoi de cada una, siendo ésta líquida si es mayor a 75 $pixeles^2$ o sólida si es menor. Este valor se obtuvo en una imagen en la cual se encuentran muy bien definidos los estados sólidos y líquidos en donde se obtuvieron las áreas de las regiones de Voronoi y la frecuencia de partículas dentro de un rango de área, con lo que se generó un gráfico (Figura 12) en donde se vislumbraron dos picos consistentes con la cantidad de partículas en estado sólido y líquido y un valle entre estos picos el cual fue de 75 $pixeles^2$.

En el gráfico se vislumbra un máximo de alrededor de 60 $pixeles^2$ que coincide con el valor de estado solido de las partículas, siendo mayor que el valor mínimo posible. Este valor se explica si las partículas están levemente separadas y no en contacto por vibración, así un 10% del diámetro de la partícula es suficiente para explicar el valor obtenido.

[5] Ver más en http://mathworld.wolfram.com/VoronoiDiagram.html

[6] $Area_{voronoi} Minima = \left(\frac{\sqrt{2}}{2} d\right)^2 = \frac{1}{2} d^2$, donde $d = Diámetro\ de\ la\ partícula\ en\ pixeles = 10_{px} \Rightarrow 50 px^2$

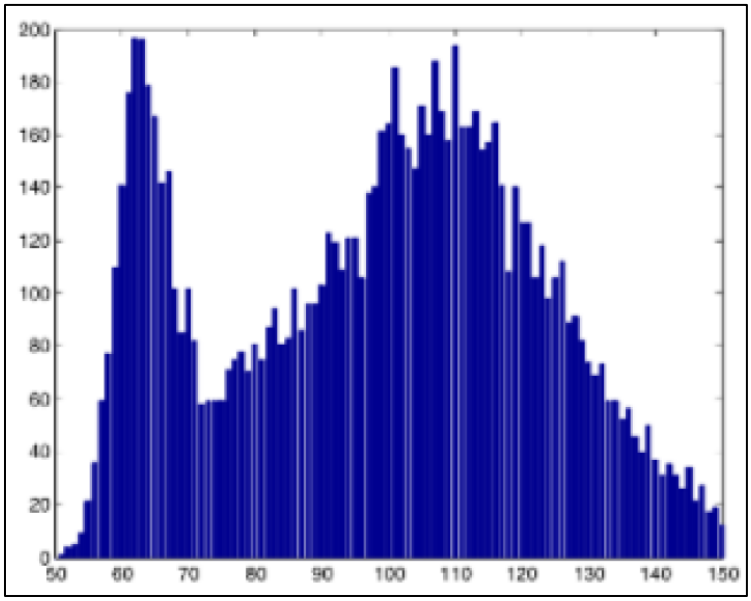


Figura 12: Cantidad de partículas (Eje Y) vs Área de Voronoi (Eje X).

Implementación[3][2][4]

A continuación se describen los pasos generales de la implementación en C de este algoritmo.

Implementación algoritmo Chi2HD	
1	Normalizar imagen
2	Generar imagen de partícula ideal
3	Obtener imagen Chi2 mediante convolución
4	Obtener los mínimos locales, como lista, de la imagen Chi2 filtrando por un mínimo de intensidad y distancia
5	Validar mínimos locales por un mínimo de separación entre ellos
6	Generar matriz auxiliar conteniendo distancia en X e Y al mínimo más cercano y el índice del mínimo dentro de la lista
7	Calcular imagen diferencia entre imagen original normalizada e imagen generada a partir de las partículas encontradas usando la matriz auxiliar
8	Obtener error cuadrado de la diferencia
9	**Iterar N veces o hasta que el numero de partículas encontradas sea igual a 0:**
10	Generar imagen escalada del paso 7
11	Obtener imagen Chi2 mediante convolución usando la imagen escalada
12	Repetir paso 4 y 5 usando la nueva imagen Chi2
13	Eliminar partículas detectadas fuera de la imagen
14	Agregar partículas detectadas a la lista
15	Repetir pasos 6 y 7
16	**Iterar N veces o hasta que el delta del error cuadrado sea menor o igual a 1:**
17	Mover partículas a su centro estimado mediante método de Newton
18	Repetir paso 6 y 7
19	Calcular diferencia entre error cuadrado anterior y el nuevo
20	Calcular valores de Mu y Sigma para ajuste gaussiano
21	Calcular mínimo de intensidad a partir de Mu y Sigma
22	Calcular áreas de Voronoi para cada partícula
23	Eliminar partículas con intensidad de imagen y área de Voronoi menor a la mínima necesaria
24	Repetir paso 8
25	Recalcular áreas de Voronoi
26	Establecer estado de materia según área de Voronoi

2.5. Factor de Estructura Estático [21]

El factor de Estructura Estático $S(k)$ es la intensidad de las fluctuaciones de densidad en el espacio de Fourier. Más específicamente es una medida de fluctuaciones de densidad en el espacio de momentum $\delta\rho(\vec{k}) = \rho(\vec{k}) - \langle\rho(\vec{k})\rangle$, donde $\langle\rho\left(\vec{k}\right)\rangle$ es el promedio de configuraciones de cada componente de Fourier. En la práctica, el promedio de las configuraciones se remplaza por un promedio temporal de imágenes debido a que el sistema es ergodico[7]. Así el Factor de Estructura Estático está definido por:

$$S(\vec{k}) = \frac{\langle\delta\rho(\vec{k})\delta\rho(\vec{k})^*\rangle}{N} = \frac{\langle\rho(\vec{k})\rho(\vec{k})^*\rangle - \langle\rho(\vec{k})\rangle\langle\rho(\vec{k})^*\rangle}{N} = \frac{\langle|\rho(\vec{k}) - \langle\rho(\vec{k})\rangle|^2\rangle}{N}$$

Donde $*$ es el complejo conjugado. Dado que el sistema en estudio es isotrópico, se tiene que $S(\vec{k}) = S(k)$. La Figura 13 muestra el Factor de Estructura Estático medido en el sistema en estudio, aquí se aprecia que $S(k)$ muestra un comportamiento considerado usual esperado para líquidos con orden de corto alcance, además se observa un primer máximo local o pre-peak conocido como First Sharp Diffraction Peak o FSDP.

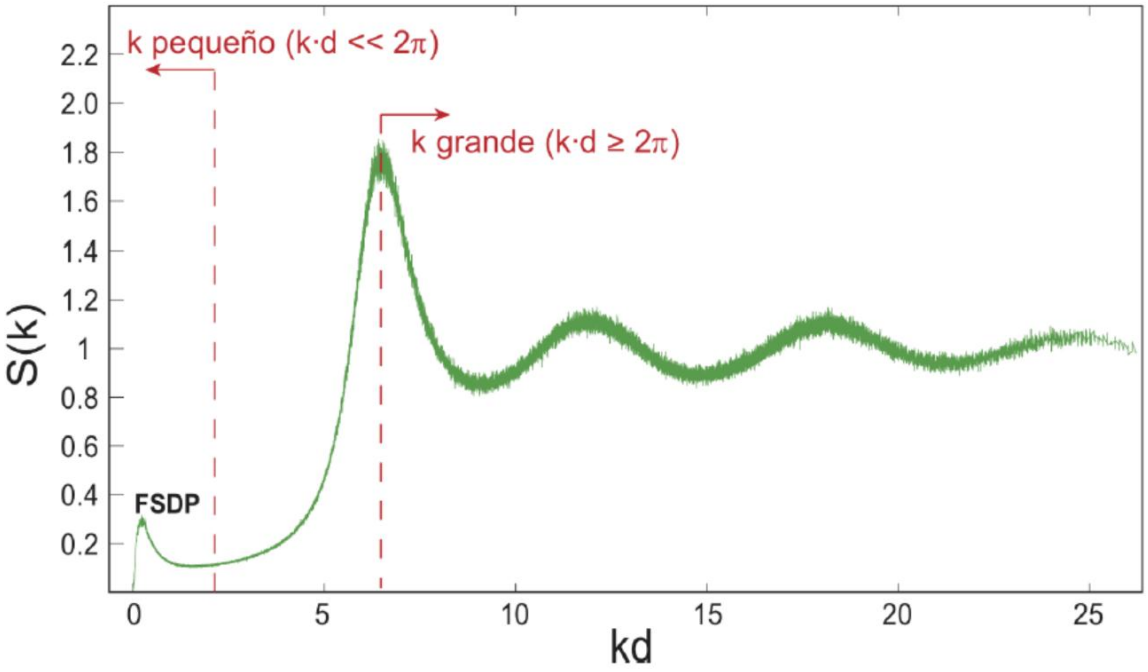


Figura 13: Factor de estructura estático S(k) versus kd donde d es el diámetro de la partícula.

Este máximo local ha sido relacionado con orden cristalino de mediano alcance en sistemas amorfos que se aproximan a la transición de tipo vidrio. El FSDP es graficado y comparado para verificar la exactitud de la solución implementada para el cálculo de las posiciones de las partículas.

[7] Un sistema se dice Ergodico si se asume que recorre todo un espacio de fases si se espera un tiempo infinito.

Capítulo 3

Especificación del problema

El software $PTrack$ implementa el $Algoritmo\ \chi^2$ y el $Algoritmo\ \chi^2 HD$ para detectar partículas en imágenes de baja y alta densidad respectivamente[8], sin embargo, la implementación de estos algoritmos no era robusta produciendo en algunas ocasiones detención del software, ciclos infinitos y aumento desmesurado e infinito de memoria y en consecuencia detención total de los cálculos programados dentro del computador en el cual se procesaban las imágenes. Además la implementación de las funciones asociadas a los cálculos no se aprovechaba ninguna mejora posible como por ejemplo optimización de las estructuras de datos, manejo de múltiples hilos de ejecución y optimización dentro del compilador. Otro punto importante es que $PTrack$ originalmente calculaba las áreas de Voronoi solamente para filtrar partículas no para establecer el estado de la materia de ésta por lo que ese estado se debía recalcular en MATLAB. Con todo lo anterior en conjunto, los tiempos de procesamiento se dilataban mucho más de lo que podía ser.

Para el procesamiento de videos se utilizaba MATLAB como lanzador en secuencia para procesar todas las imágenes necesarias y para procesar todos los videos se utilizaba varias instancias de MATLAB trabajando en paralelo consumiendo muchos recursos del sistema.

La arquitectura de software no se encontraba muy clara y solamente poseía 4 archivos con funciones que trabajaban en conjunto siendo estos los siguientes:

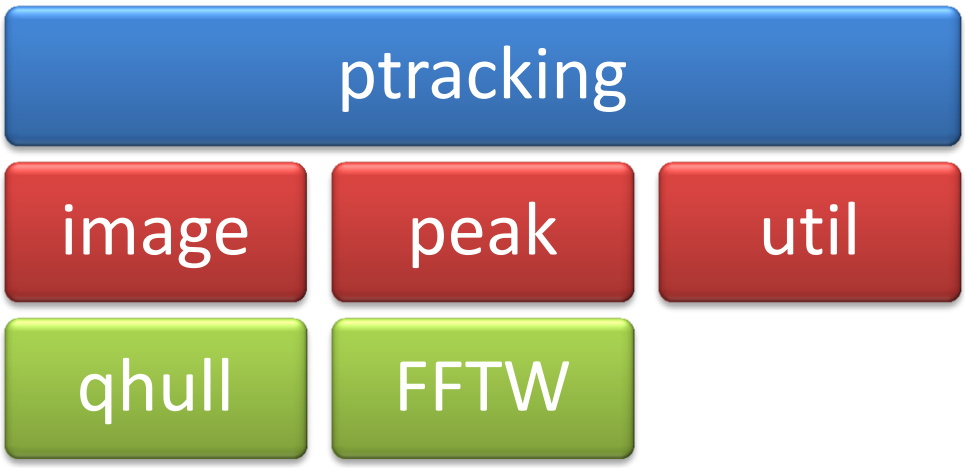


Figura 14: Arquitectura de PTrack.

[8] Ver Capítulo 2, sección 2.3.

Primeramente el archivo ***ptracking*** se encargaba de establecer todos los argumentos necesarios para la ejecución de uno u otro algoritmo diferenciándose solamente en la ejecución de una u otra función y se encargaba de llamar a las demás funciones de los archivos. Todo lo anterior dentro de una gran función main. El objetivo del archivo ***image*** era el tratamiento de imágenes, desde la lectura, normalización, generación de convolución, generación de diferencia entre 2 imágenes y obtención de imagen chi2. El archivo ***peak*** contenía funciones para trabajar con las partículas detectadas trabajando en conjunto con las funciones definidas en el archivo ***image***. Por último el archivo ***util*** poseía funciones para reservar, establecer y liberar memoria de matrices de números de doble precisión y enteros.

3.1. Alternativas de solución

A continuación, se mencionan las alternativas de solución propuestas para los problemas mencionados anteriormente considerando el contexto y los recursos disponibles.

3.1.1. Reimplementación en C++

Como parte de la solución completa, se planeó reimplementar el programa en ese entonces en uso, de lenguaje C a C++ usando patrones de diseño[3][16] y enfocando el desarrollo a uno orientado a objetos para facilitar la implementación de nuevos algoritmos.

3.1.2. Distribución de Computo

Para acelerar el procesamiento de datos y existiendo la posibilidad de usar más computadores para procesar y mejorar el rendimiento, es que se planeó hacer uso de distribución de carga, vale decir, hacer uso de varios computadores en conjunto para procesar todos los datos necesario[11], además se logró vislumbrar que el procesamiento de una imagen es independiente a las demás, por lo que la posibilidad de procesar en computadores independientes era factible.

3.1.3. Procesamiento por lotes

Ejecutar en un mismo computador múltiples instancias de programa, en vez de ejecutar secuencialmente, ayuda en cierta manera a acelerar el procesamiento de datos, así mismo como se intentaba realizaba en MATLAB ejecutando varias instancias de MATLAB ejecutando $PTrack$. Sin embargo, esto sirve solamente si no se sobrecarga el computador y si el programa que se ejecuta se encuentra bien encapsulado y no depende de tantas variables externas de su ambiente de ejecución, como por ejemplo, leer y escribir desde un único archivo.

3.1.4. Aceleración de cálculo usando varios hilos de ejecución

En un programa pueden existir varios hilos de ejecución o threads (de ahora en adelante), así se pueden ejecutar varias funciones en paralelo sin la necesidad de crear nuevas instancias del software. Como se vio al analizar el código existente en ese entonces, todas las tareas se realizaban secuencialmente, sin embargo muchas podían ser ejecutadas simultáneamente debido a la independencia de los datos que se trabajan, por lo que los algoritmos utilizados eran altamente paralelizables.

3.1.5. Aceleración de cálculo usando CUDA

Además de utilizar threads en el desarrollo de software, el alumno decidió tener como propuesta el uso de tecnología CUDA[10], la cual utiliza el poder de cómputo que poseen las tarjetas de video Nvidia, así se podía lograr un gran desempeño demostrado en muchos papers[7] y artículos en internet[5][6][8][9], facilidad de programación con respecto a otro lenguaje del mismo estilo y un aprovechamiento del poder que posee la tarjeta de video. Además, considerando la experiencia del alumno con esta tecnología es que se consultó por la incorporación de CUDA a la propuesta.

3.1.6. Compra de nuevos computadores

Con el fin de actualizar la tecnología existente en el laboratorio y potenciar varias soluciones propuestas, es que se solicitó la compra de un computador nuevo con partes que se pudieron encontrar en el mercado, escogiendo los mejores componentes con la mejor relación precio/rendimiento y acompañado de una tarjeta de video Nvidia con tecnología CUDA con la misma relación anterior.

3.2. Alternativas escogidas

Considerando todas las propuestas de solución y vislumbrando que todas eran factibles y no eran mutuamente excluyentes es que se decidió incorporar todas las propuestas teniendo la oportunidad de obtener y utilizar todos los recursos disponibles para agilizar el procesamiento.
Para una buena implementación en C++ del nuevo software se fijaron las siguientes premisas:

1. El procesamiento de imágenes debe ser lo más rápido posible y notoriamente menor a la forma antigua de procesar las imágenes, pudiendo facilitar al investigador procesar los datos obtenidos de un día para otro.
2. El nuevo software debe ser lo suficientemente robusto como para no detenerse o quedar en ciclos infinitos al procesar una imagen.
3. El desarrollo de nuevos algoritmos de detección de partículas tiene que ser lo más fácil, menos invasivo y más compatible posible.

Para cumplir la robustez necesaria de todo el sistema en su conjunto, se fijaron las siguientes premisas para el sistema distribuido:

1. La especificación de cuáles computadores se utilizan, si se debe procesar con CUDA y cuántas imágenes simultáneamente pueden ser procesadas, debe ser flexible y parametrizable.
2. La distribución de cómputo debe ser tolerante a errores de distinto tipo.
3. El agregar nuevos computadores para procesar debe ser lo más fácil posible.

Capítulo 4

Descripción de la solución

La solución completa planteada en esta memoria se puede dividir en 2 partes:

1. Reimplementación en C++ de $PTrack$ usando patrones de diseño, programación orientada a objetos y tecnología CUDA. Este nuevo desarrollo se llamará de ahora en adelante $PTrack2$.
2. Repartición de cómputo mediante un sistema distribuido. Este sistema se llamará de ahora en adelante $PTrack2Bash$.

Cabe destacar que durante el desarrollo se usó el sistema de administración de proyectos llamado **trac** que se integra a un sistema de versionamiento, en este caso ***Subversion***, e incluye una wiki para documentar el desarrollo. La principal razón del uso de Subversion fue para evitar mezclas de código fuente, pérdida de código, simple distribución del software, ordenar las ramas de desarrollo y establecer las versiones estables haciendo uso de *Branches* y *Tags*. Además se consideró la experiencia con este sistema de versionamiento.

Toda la documentación relacionada a la instalación, configuración y uso del software desarrollado se escribió en la wiki donde se puede ver desde cualquier lugar[9] por internet. Como complemento a la wiki, se generó un ***man page***[10] con la documentación necesaria para correr todo el software el cual se puede leer tipeando en un terminal ***"man PTrack2"***. Además se integró una API generada con ***Doxygen***[17] para desarrollos futuros en donde se especifica la relación y uso de las clases así como también la especificación de cada función que se utiliza.

Para facilitar el uso de $PTrack2$ se generó un script de auto-compleción que se ejecuta con ***Bash_completion***, el cual ayuda a ver las opciones disponibles del programa mediante el tecleo doble de la tecla "tab" y una vez escrito el sufijo del argumento éste se completa presionando una vez la tecla "tab", con esto se evita confusión al ingresar los parámetros, facilita la escritura de éstos y se hace más rápida la ejecución.

El proceso de desarrollo se realizó en 3 etapas:

1. Implementación de un sistema distribuido usando el software antiguo $PTrack$.
2. Desarrollo de $PTrack2$ sin uso de CUDA y $PTrack2Bash$.
3. Extensión de $PTrack2$ utilizando CUDA y optimización de $PTrack2Bash$.

[9] El repositorio Subversion y trac se encuentran en: http://trac.assembla.com/particle-tracking-2

[10] Man page es una reducción de Manual Page, el cual es una documentación de software para el programa de unix **man** el cual presenta estos datos en pantalla.

El fin de realizar en este orden el desarrollo, fue para colaborar desde el primer día a la reducción en los tiempos de procesamiento, para así, aportar directa, continua e incrementalmente a la mejora de todo el proceso de detección de partículas.

La primera etapa de desarrollo contempló la base para $PTrack2Bash$ pues en esencia era el mismo software en donde solo se cambiaba la ejecución de $PTrack2$ en vez de $PTrack$. También se instaló este sistema en los computadores disponibles y se puso en marcha, lo cual ayudó al avance de los cálculos. Además se documentó y facilitó la instalación de $PTrack$ escribiendo en la wiki del proyecto y modificando el archivo de compilación Makefile para que instalara el software en el sistema. Este desarrollo se encuentra temporalmente en el servidor privado del alumno[11].

La segunda etapa contempló el desarrollo de $PTrack2$, la modificación del sistema distribuido anterior creándose $PTrack2Bash$, la verificación de la correctitud de los datos generados y la instalación y puesta en marcha del sistema. Aquí se generó la primera documentación de $PTrack2$ en la wiki del proyecto.

La tercera etapa contempló el desarrollo de la parte en CUDA de $PTrack2$, pruebas de exactitud y correctitud de los datos procesados usando CUDA, optimización de $PTrack2Bash$, documentación de la API, documentación completa en la wiki, *man page*, *bash completion* y afinación de detalles como limpieza de código, refactorización, entre otros.

En los siguientes puntos se describe el resultado final del sistema por lo que los pasos intermedios se omiten.

4.1. Modificación de $PTrack$

Para poder ejecutar de buena manera $PTrack$ y suplir los requerimientos de los investigadores, éste sufrió dos cambios importantes.

Primero se modificó para recalcular las áreas de Voronoi para cada partícula, una vez filtradas las partículas mal detectadas, y establecer el estado de éstas, ya sea sólido o líquido exportando tanto las posiciones como los estados físicos de cada partícula al archivo de salida. Cabe destacar que el cálculo de las áreas de Voronoi para establecer el estado físico de cada partícula se realizaba en MATLAB una vez detectadas las partículas por parte de $PTrack$.

El segundo cambio que sufrió fue la modificación para poder ejecutarse en 64 bits ya que su versión original solo se podía ejecutar en 32bits. Si bien $PTrack$ se podía compilar y ejecutar en 64 bits, los resultados obtenidos eran erróneos principalmente debido a que no se iniciaban de forma completa los valores de determinadas matrices, así en 32 bits los valores no iniciados eran 0 y en 64 bits eran valores basura así los cálculos quedaban corruptos.

[11] El repositorio para la etapa 1 se encuentra en: http://myatom.dyndns-server.com/trac/ptrack/

4.2. Arquitectura de Hardware

Para poder procesar datos en varios computadores a la vez y aprovechar así su poder de cómputo, es que se decidió montar un sistema distribuido como **cluster**, es decir, varios computadores conectados con similares arquitecturas que trabajan en conjunto y que vistos desde el equipo ejecutor del software, funcionan como un solo equipo. Estos computadores tienen un ***Sistema operativo de red*** el cual, en este caso, se basan en UNIX los que son *Ubuntu Linux* y *Mac OSX*. En este sentido, al hacer uso de un sistema operativo de red es que el sistema es escalable pudiendo agregar mayor poder de cómputo fácilmente.

Para poder comunicarse entre sí, se ocupa una interfaz de red común, los mensajes son pasados punto a punto como cliente servidor y se utiliza el protocolo SSH.

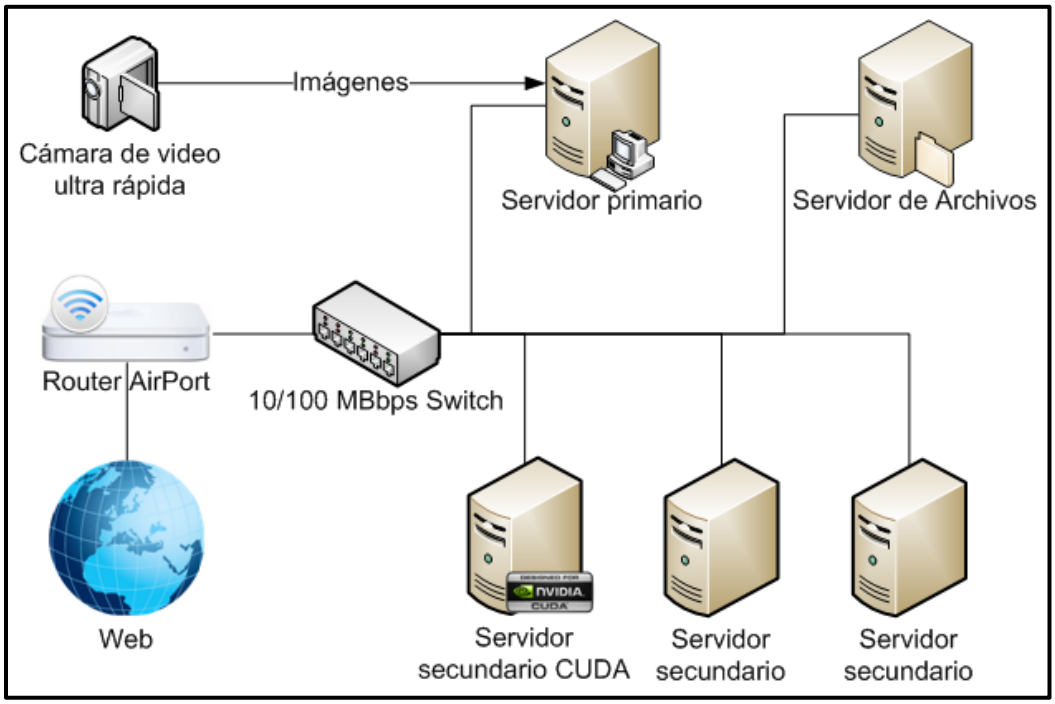


Figura 15: Arquitectura del sistema.

Las imágenes a ser procesadas se captan a través de una cámara de alta velocidad conectada al servidor primario, estas imágenes son almacenadas y luego distribuidas a cada servidor secundario. Si no se tiene espacio suficiente en el disco duro del servidor primario, las imágenes se almacenen en un NAS o servidor de archivos. Todos los computadores se encuentran conectados a un Switch FastEthernet con una velocidad soportada de 10-100 MegaBits por segundo y a su vez, el switch se encuentra conectado a un router Apple Airport con acceso a internet.

Cada servidor del cluster tiene diferentes características, las cuales se especifican a continuación:

Servidor Primario - *Granos2D*			
Modelo:	Mac Pro	**Número total de núcleos:**	8
Modelo de Procesador:	Intel Xeon E5520	**Número total de threads:**	16
Número de procesadores:	2	**Memoria:**	6 GB
Velocidad del procesador:	2.26 GHz	**Sistema operativo:**	Mac OSX 10.11

Tabla 2: Configuración Servidor primario - Granso2D

Servidor Secundario - *Cuasi1D*			
Modelo:	Mac Pro	**Número total de núcleos:**	4
Modelo de Procesador:	Intel Xeon 5150	**Número total de threads:**	4
Número de procesadores:	2	**Memoria:**	5 GB
Velocidad del procesador:	2.66 GHz	**Sistema operativo:**	Mac OSX 10.11

Tabla 3: Configuración Servidor Secundario - Cuasi1D

Servidor Secundario - *LMFE*			
Modelo:	Mac Pro	**Número total de núcleos:**	4
Modelo de Procesador:	Intel Xeon 5150	**Número total de Threads:**	4
Número de procesadores:	2	**Memoria:**	4 GB
Velocidad del procesador:	2.66 GHz	**Sistema operativo:**	Mac OSX 10.11

Tabla 4: Configuración Servidor Secundario - LMFE

Servidor Secundario CUDA – *Granos2D-i7*			
Modelo de Procesador:	Intel Core i7-2600K	**Modelo Tarjeta de video:**	GTX 590
Número de procesadores:	1	**Dispositivos CUDA:**	2
Velocidad del procesador:	3.40 GHz	**Velocidad de GPU:**	630 MHz
Número total de núcleos:	4	**Núcleos CUDA:**	512 x 2
Número total de Threads:	8	**Memoria de video:**	1.536 GB x 2
Memoria:	8 GB	**Compute Capability:**	2.0
Sistema operativo:	Ubuntu Linux 11.10 x64	**GFLOPS[12] estimados:**	2488 [20]

Tabla 5: Configuración Servidor Secundario - Granos2D-i7

Este último computador fue comprado a petición del alumno ya que dentro del laboratorio no existía un computador con las características necesarias para desarrollar usando CUDA.

Las principales ventajas que se tienen al idear el sistema de esta forma son:

1. Reutilización de computadores existentes.
2. Los servidores secundarios no son exclusivamente para el procesamiento de datos, aprovechando así el uso de éstos por parte de otros investigadores.

[12] El poder de cómputo de cualquier procesador puede expresarse en Giga FLOPS o GFLOPS en donde un FLOPs es una operación de punto flotante por segundo, generalmente este valor es estimado y representa el potencial de cálculo que posee un procesador.

4.3. Arquitectura de Software

El software desarrollado se presenta de manera general con la siguiente estructura:

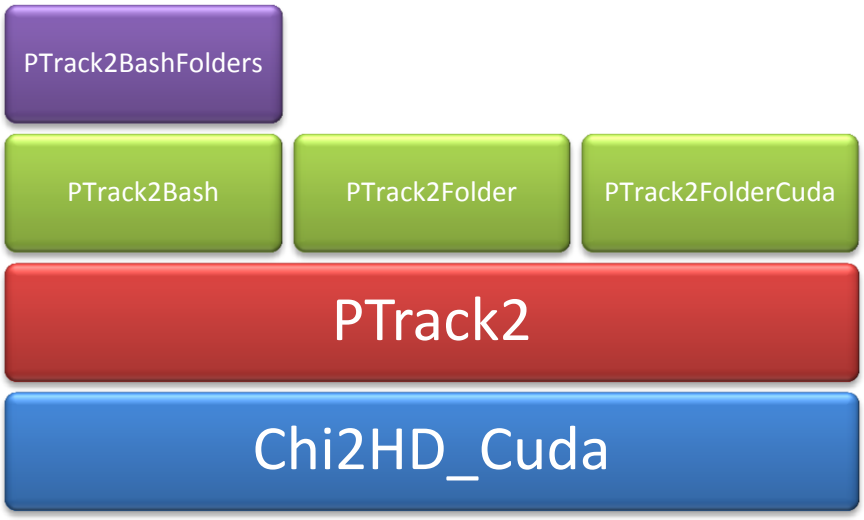


Figura 16: Arquitectura de software.

Para la versión de $PTrack2$ usando CUDA se utiliza una biblioteca creada por el alumno llamada $Chi2HD\_Cuda$ en la cual se encuentran las funciones necesarias para ejecutar el algoritmo chi2hd usando esta tecnología. Arriba de esta biblioteca se encuentra el programa central $PTrack2$ que es el que detecta las partículas en las imágenes.

Encima de $PTrack2$ se encuentran tres scripts, uno es $PTrack2Bash$, el sistema que distribuye el procesamiento de las imágenes a calcular en diversos computadores y haciendo uso de múltiples instancias de $PTrack2$. El segundo es $PTrack2Folder$ que se utiliza para procesar en un solo computador una carpeta con imágenes haciendo uso de múltiples instancias de $PTrack2$ y el tercero es $PTrack2FolcerCuda$ el cual es lo mismo que $PTrack2Folder$ solo que usa la versión CUDA de $PTrack2$.

Al tope de ésta arquitectura se encuentra $PTrack2BashFolders$, un script que hace uso de $PTrack2Bash$ para procesar múltiples carpetas con imágenes.

Cada una de las partes de la solución completa se describe en los siguientes puntos.

4.4. $PTrack2$

El programa $PTrack2$ es la base de todo el sistema y presenta una serie de avances con respecto a su predecesor $PTrack$. En primera instancia se tiene un desempeño general mucho mejor, además se puede seleccionar de manera sencilla qué algoritmo de detección de partículas se desea ejecutar teniendo hasta el momento 3 opciones: el algoritmo original de detección de partículas $Algoritmo\ \chi^2$, el algoritmo $Algoritmo\ \chi^2$ modificado para detectar partículas en estado muy denso $Algoritmo\ \chi^2 HD$ y la implementación de $Algoritmo\ \chi^2 HD$ usando CUDA.

Otro avance que presenta es la extensibilidad de éste, pudiendo implementar nuevos algoritmos de una manera rápida y sencilla, además de corregir algunos errores presentes en su predecesor, evitando ciclos infinitos y caídas por variables no chequeadas.

Por otro lado se presentan algunas opciones para filtrar partículas falso positivo mediante intensidad de imagen y área de Voronoi, incluyendo éstas dos juntas al igual que en el algoritmo $Algoritmo\ \chi^2 HD$ original. También presenta la opción de optimizar en ciertas ocasiones la detección en imágenes con partículas en estado muy denso, pudiendo detectar más, posponiendo la validación por distancia entre ellas una vez detectadas todas las posibles partículas[13] y optimizadas sus posiciones. Además se presenta la opción de poder ejecutar el algoritmo seleccionado sin usar threads si aplica al caso, esto con la finalidad de poder optimizar la ejecución bajo ciertos casos en que el procesador tenga menos núcleos de procesamiento o no tenga una tecnología que permita ejecutar de una manera óptima estos threads, como por ejemplo la tecnología HT de Intel.

Otras opciones agregadas tienen un propósito de depuración y chequeo como el despliegue de imagen con las partículas detectadas y el despliegue de texto en distintos niveles para depuración, información, errores y otros.

Algunos elementos que permitieron aumentar el rendimiento de $PTrack2$ fueron primeramente CUDA, en segundo lugar optimizaciones a las estructuras de datos y manejo de éstas y finalmente el uso de Flags[18] de compilación que ayudaron a que el software funcionase a una velocidad mucho mayor haciendo provecho de las tecnologías e instrucciones especificas del procesador donde se ejecutará el software. Los Flags que se utilizaron son los siguientes:

- ***O3:*** Este flag de optimización establece una serie de otros flags para acelerar los cálculos. O2 y O3 son flags estándar para distribuir un software a producción.
- ***march=native:*** Genera instrucciones para el tipo de procesador específico de la máquina.
- ***mfpmath=sse:*** Genera aritmética de punto flotante para hacer uso del set de instrucciones SSE del procesador.
- ***ftree-vectorize:*** Realiza vectorización de ciclos en instrucciones tipo árbol.
- ***funroll-loops:*** Desenvuelve los ciclos en que se puede determinar su número de iteraciones en tiempo de compilación.

[13] Verificar Antecedentes punto 2.3.2. sección Implementación.

- ***ffast-math:*** Establece una serie de otros flags que tienen como efecto acelerar los cálculos, sin embargo, puede producir errores en resultados que requieran una implementación exacta de las normas IEEE o ISO para funciones matemáticas.

Todos los flags de compilación que se utilizaron no produjeron errores en los cálculos y aceleraron de forma drástica el rendimiento de $PTrack2$. Estos solo son algunos de los flags que se pueden utilizar y todo depende del problema a considerar y de la aplicabilidad de éstos.

4.4.1. Arquitectura de software

$PTrack2$ utiliza varias bibliotecas para su funcionamiento, cada una con una función en específico, presentando así la siguiente arquitectura de software:

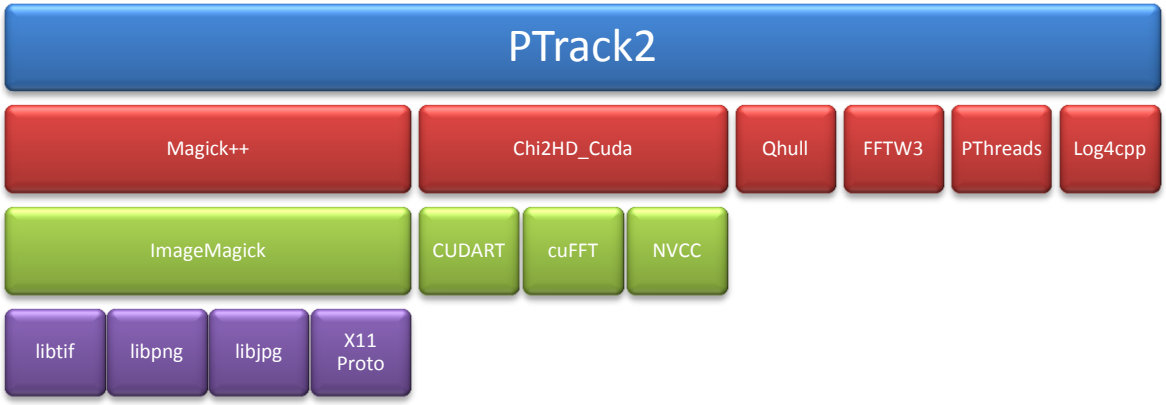


Figura 17: Arquitectura de software de PTrack2.

Magick++

Magick++ es la API orientada a objetos usando C++ de la biblioteca de procesamiento de imágenes ***ImageMagick*** la cual posee una infinidad de funciones[14], transformaciones y utilidades para trabajar con imágenes, además posee una buena documentación. Magick++ se utilizó para leer los archivos de imágenes en formato TIF, pudiendo leer también archivos PNG y JPG en caso de querer procesar una imagen en distinto formato, también se utilizó para desplegar imágenes de apoyo y dibujar dentro de esas imágenes de apoyo. Esta biblioteca depende directamente de ImageMagick y a su vez ésta depende de las bibliotecas de codificación y decodificación de imágenes TIF, PNG, JPG entre otras, las cuales no son necesarias para su funcionamiento pero sí para la apropiada decodificación de ese tipo imágenes. Para desplegar imágenes en pantalla también se apoya en el uso de la biblioteca X11 Proto.

[14] Ver ejemplos en http://www.imagemagick.org/script/examples.php

Qhull

Qhull es una aplicación para resolver el problema de la cerradura convexa. Además presenta varias opciones de ejecución como por ejemplo el calculo del Área de Voronoi para cada partícula detectada dentro de una imagen.

FFTW3

La biblioteca *Fast Fourier Transform of the West*, es una biblioteca muy usada para calcular transformadas de Fourier rápidamente. En el caso de $PTrack2$, se usa para calcular la convolución de imágenes entre la imagen original y la imagen generada de una partícula ideal.

PThreads

PThreads es una biblioteca estándar que se utiliza para realizar tareas en paralelo haciendo uso de procesos livianos o threads que comparten memoria y almacenamiento permanente. $PTrack2$ utiliza esta biblioteca para hacer cálculos en paralelo en muchas funciones, por ejemplo, en el cálculo de convolución se utiliza para realizar 2 cálculos de convolución en paralelo.

Log4cpp[19]

Log4cpp es una biblioteca para realizar registros o log creada en C++. La principal función que realiza dentro de $PTrack2$ es la de manejar los registros de las etapas de ejecución para depuración, información, errores o marcar momentos en el tiempo.

Chi2HD_Cuda

Esta biblioteca, programada por el alumno, implementa funciones propias para la ejecución del $Algoritmo\ \chi^2 HD$ usando tecnología CUDA. Así, por ejemplo, para el cálculo de convolución se utiliza **cuFFT**[14] que es una biblioteca para el cálculo de transformadas de Fourier implementada en CUDA. La descripción detallada de esta biblioteca se presenta más adelante.

4.4.2. Diagrama de clases

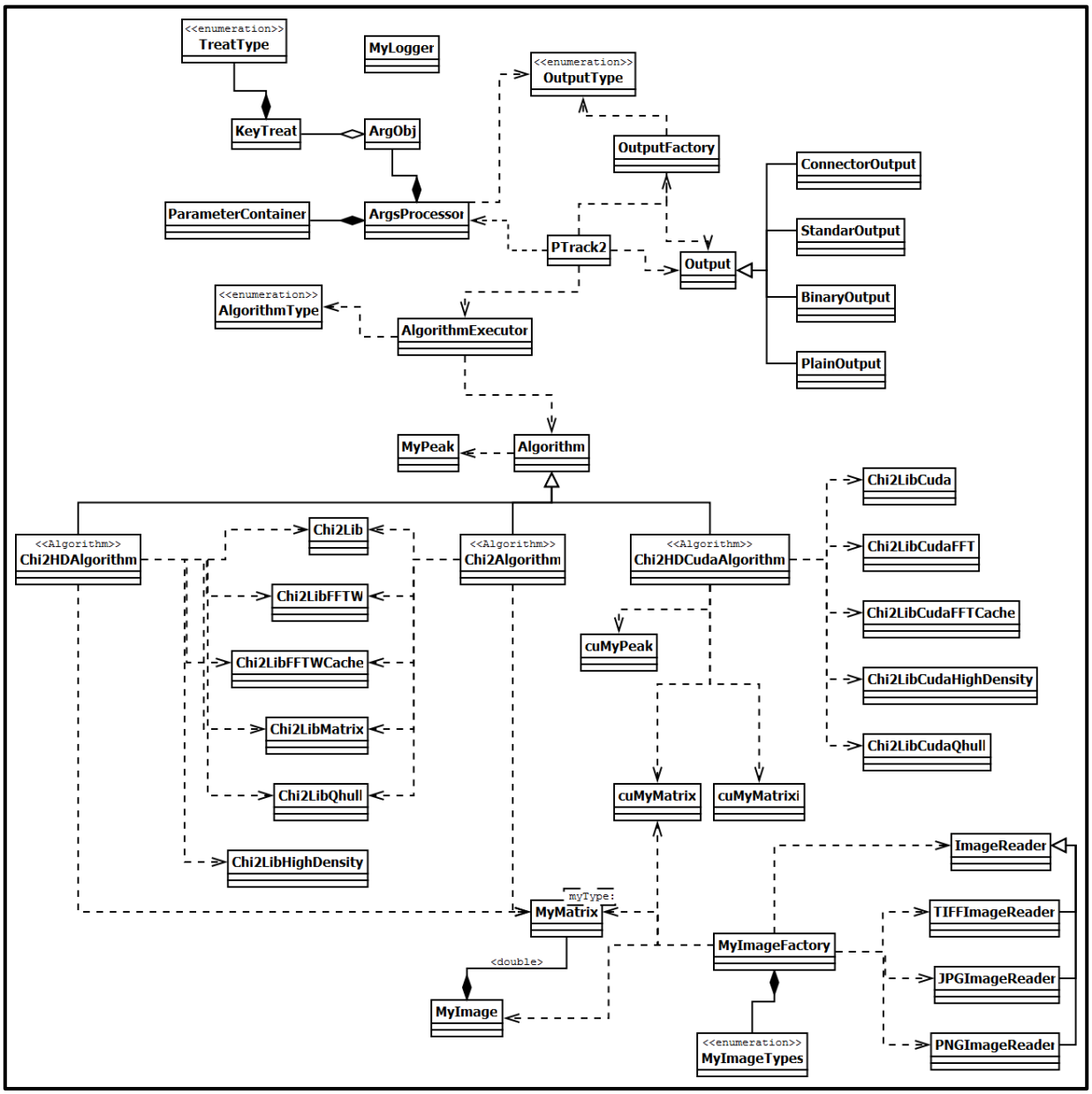


Figura 18: Diagrama de clases simplificado.

En el diagrama de clases se pueden distinguir algunos patrones de diseño que se implementaron para facilitar la extensión y las futuras modificaciones que pudiese sufrir el software. A continuación se enuncian los patrones de diseño que se usaron, dónde se usaron y porqué.

Singleton

Para permitir el uso global del registro de los pasos que se están ejecutando en la aplicación, es que se decidió usar un Singleton en la clase **MyLogger**, permitiendo la configuración de éste una sola vez para luego poder hacer uso de ésta clase desde cualquier parte de la aplicación sin configuración necesaria. También se utilizó Singleton para la clase de procesamiento de argumentos **ArgsProcessor** para permitir a futuro una sola instancia sin tener que crearla nuevamente y procesar por más de una vez los argumentos ingresados al programa. Además para evitar reservas, liberación de memoria y guardar cálculos para posteriores llamadas a **Chi2LibFFTW** y **Chi2LibCudaFFT** se decidió usar singleton para almacenar datos como cache en las clases **Chi2LibFFTWCache** y **Chi2LibCudaFFTCache**.

Factory Method

Factory method se utilizó principalmente para crear las imágenes mediante la clase **MyImageFactory** y objetos de salida de datos mediante la clase **OutputFactory**, así independiente a la implementación, el objeto creado se puede utilizar haciendo las mismas llamadas a funciones que antes se utilizaban.

Strategy

Este patrón de diseño se utilizó para ejecutar y configurar los distintos algoritmos de procesamiento de imágenes, facilitando así la generación, ejecución y configuración de nuevos algoritmos a implementar en el futuro. La clase de ejecución de Strategy es **AlgorithmExecutor** mientras que las clases que ejecutan algoritmos deben extender e implementar funciones de la clase **Algorithm**. Con todo lo anterior, es posible implementar un nuevo algoritmo modificando solamente una clase (AlgorithmExecutor) y extendiendo e implementando la clase Algorithm.

Object Pool

Chi2LibFFTWCache y **Chi2LibCudaFFTCache** ocupan este patrón de diseño para mantener en memoria Matrices que se ocupan de manera recurrente en los cálculos de transformadas de Fourier, así se evita crear y eliminar matrices que se volverán a ocupar y también evita cálculos innecesarios para obtener resultados constantes.

4.4.3. Extensión de $PTrack2$

Extender $PTrack2$ para implementar nuevos algoritmos para procesar imágenes es posible y relativamente sencillo mediante la modificación de una clase y extensión de otra y los pasos para esto se explican a continuación.

Primeramente es necesario extender la clase *Algorithm* ocultando la implementación por defecto de la función **static ArgObj myArgs();** en la cual se devuelven los parámetros aceptados por este algoritmo para su ejecución, así como el tratamiento de cada uno de ellos. También es necesario implementar la función **virtual void setData(ParameterContainer *pc);** que establece los datos necesarios para la ejecución y **virtual vector<MyPeak> run();** que ejecuta el algoritmo y devuelve las partículas encontradas.

Una vez extendida la clase *Algorithm* es necesario establecer el tipo de algoritmo generado agregando un nuevo tipo a la enumeración *AlgorithmType*. Luego se deben modificar 3 funciones de la clase *AlgorithmExecutor* los cuales son **static vector<ArgObj> getAllArguments(); static AlgorithmType translate(string alg);** y **void select(AlgorithmType type);** En donde la primera entrega todos los argumentos de los algoritmos disponibles para ser chequeados al momento de procesar los parámetros ingresados por el usuario, la segunda función enlaza el texto ingresado como algoritmo a ejecutar y el tipo efectivo del algoritmo y la ultima establece la clase necesaria para ejecutar el algoritmo.

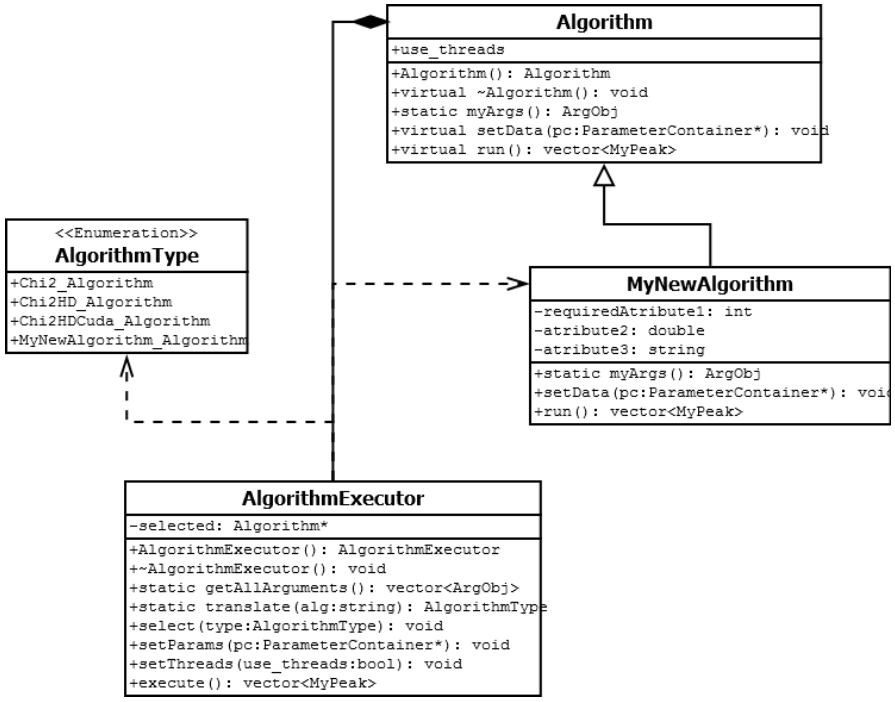


Figura 19: Extensión de PTrack2.

4.4.4. Estructuras de datos

Es importante describir algunas estructuras de datos que se implementaron con el fin de entender un poco el funcionamiento de $PTrack2$.

ArgsProcessor trabaja con una serie de clases y estructura de datos dentro de las que se destaca **ArgObj** el cual es un almacén de parámetros aceptados por cierto algoritmo, incluyendo el argumento clave para ejecutar el algoritmo, descripción y un ejemplo de cómo se debe convocar. Los parámetros aceptados por algún algoritmo se almacenan como un vector de **KeyTreat**, ésta estructura guarda la llave del parámetro, una descripción, un ejemplo de cómo se debe convocar de ser el caso y una lista de cómo debe ser procesado.

El cómo debe ser procesado un argumento se encuentra establecido por la enumeración **TreatType** pudiendo ser uno o más de los siguientes:

- ***Require_Treat***: Establece que un argumento es requerido para la ejecución del algoritmo, sin el argumento con éste tratamiento, el programa no puede ejecutarse.
- ***Followed_String_Treat***: Establece que un argumento debe estar seguido por un string.
- ***Followed_Int_Treat***: Establece que un argumento debe estar seguido por un entero.
- ***Followed_Double_Treat***: Establece que un argumento debe estar seguido por un número decimal.
- ***Exist_Treat***: Establece que un argumento requiere solamente su existencia para ser tratado.
- ***Output_Treat***: Establece un parámetro de tipo de salida de datos.

Otras estructuras de datos importantes son **MyPeak** y **MyMatrix**. La primera almacena los datos relevantes acerca de una partícula detectada, como posición, intensidad de imagen chi2, área de Voronoi y estado de materia entre otros. La segunda es una clase tipo template que representa una matriz de datos, generalmente de tipo numérico, en ésta estructura se almacenan los datos de la imagen de entrada, la imagen chi2 y algunas matrices auxiliares entre otras.

4.4.5. Funcionamiento del software

Al inicio de su ejecución, $PTrack2$ procesa los argumentos entregados chequeando primeramente que se haya seleccionado un algoritmo a ejecutar, luego se chequea que los argumentos sean aceptados por el algoritmo, arrojando una advertencia de no existir o deteniendo el programa si un parámetro no contiene los datos necesarios (ejemplo: un parámetro seguido de un número). Luego de Verificar los parámetros éstos se interpretan y se almacenan según las reglas definidas según el algoritmo.

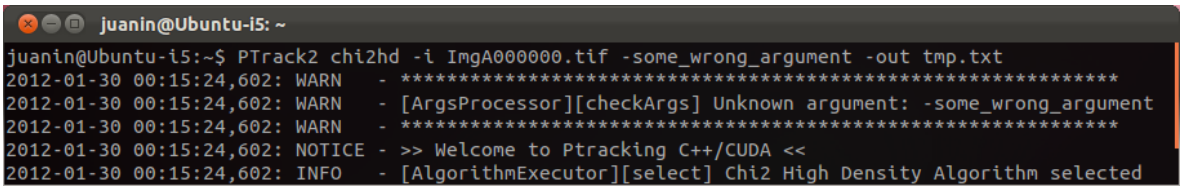


Figura 20: Alerta de argumento desconocido.

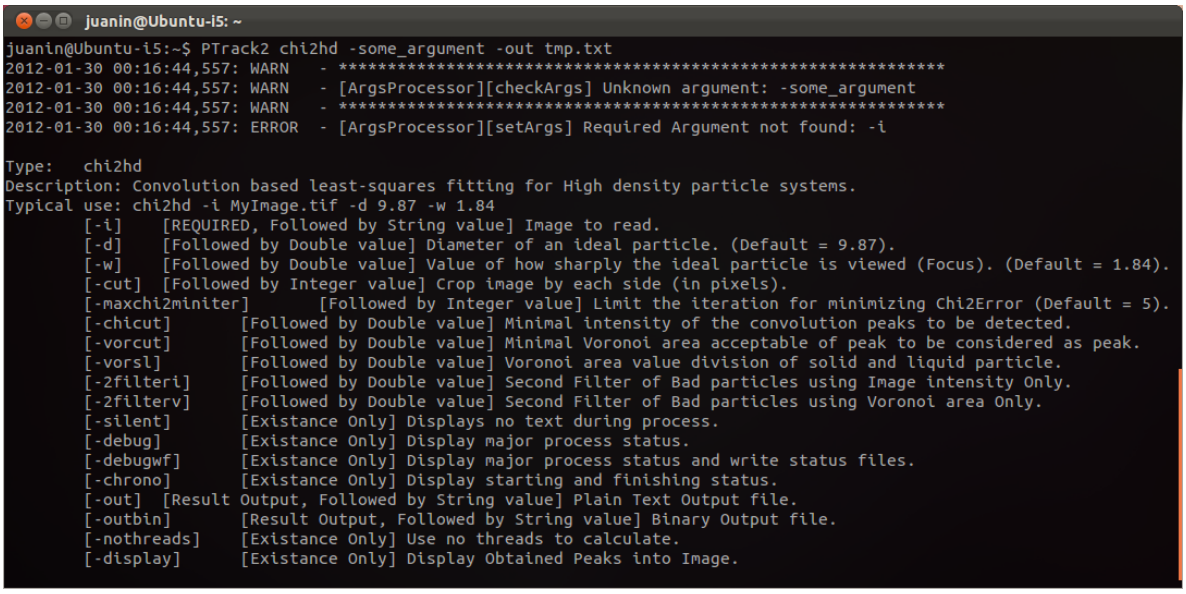


Figura 21: Argumento faltante para ejecutar el algoritmo.

A continuación se establece el nivel de registro que se va a utilizar, ocupando generalmente el modo silencioso, sin registro en pantalla, o el de depuración, presentando todos los registros de ejecución en pantalla. Una vez realizado todo lo anterior, se selecciona el algoritmo que se va a ejecutar, se establecen los parámetros necesarios y requeridos por el algoritmo y luego se ejecuta.

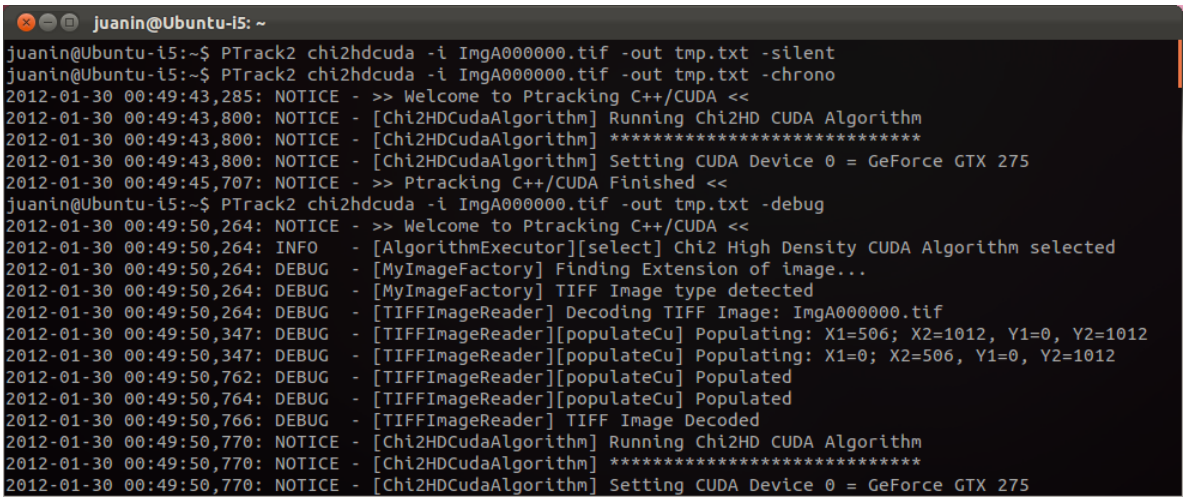


Figura 22: Distintos niveles de registro de ejecución.

Una vez obtenidas las posiciones de las partículas que el algoritmo logró detectar, se procede a seleccionar el método de salida de datos. Además si se seleccionó el modo de despliegue de imagen, se le muestra al usuario la imagen procesada con las posiciones detectadas dentro de ésta.

A continuación se describe en pseudocódigo el funcionamiento del software en general.

Implementación PTrack2	
1	Procesar argumentos de entrada
2	Establecer el nivel de log deseado
3	Seleccionar algoritmo a ejecutar
4	Establecer parámetros para el algoritmo
5	Seleccionar el uso de threads
6	Ejecutar algoritmo y obtener resultados como un vector de Peaks
7	Seleccionar tipo de salida de datos
8	Escribir vector de Peaks a la salida de datos seleccionada
9	Desplegar y guardar imagen si se desea

Un ejemplo del despliegue de imagen es el siguiente:

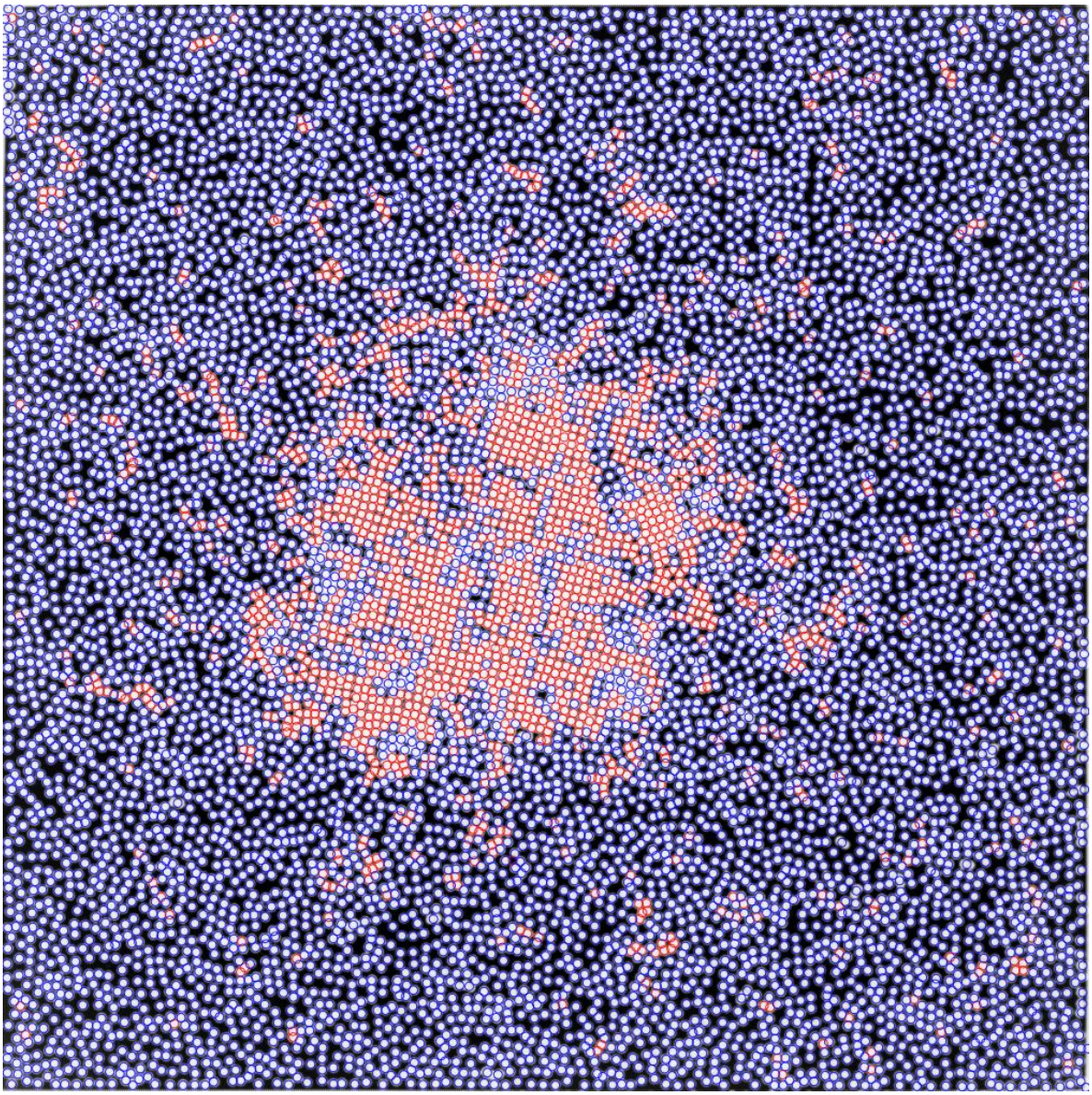


Figura 23: Ejemplo de despliegue de imagen.

En la imagen de ejemplo se pueden visualizar las partículas detectadas sobrepuestas a la imagen original con colores invertidos; de color azul se encuentran las partículas en estado líquido y en rojo las de estado sólido diferenciadas por el área de Voronoi de cada una. En este ejemplo, el área de Voronoi límite entre sólido y líquido se asignó por defecto como 75.0 $pixeles^2$, los valores de D y W se asignaron por defecto como 9.87 y 1.84 respectivamente.

En la siguiente imagen, se puede ver un ejemplo de la ejecución validando solo una vez la distancia entre partículas.

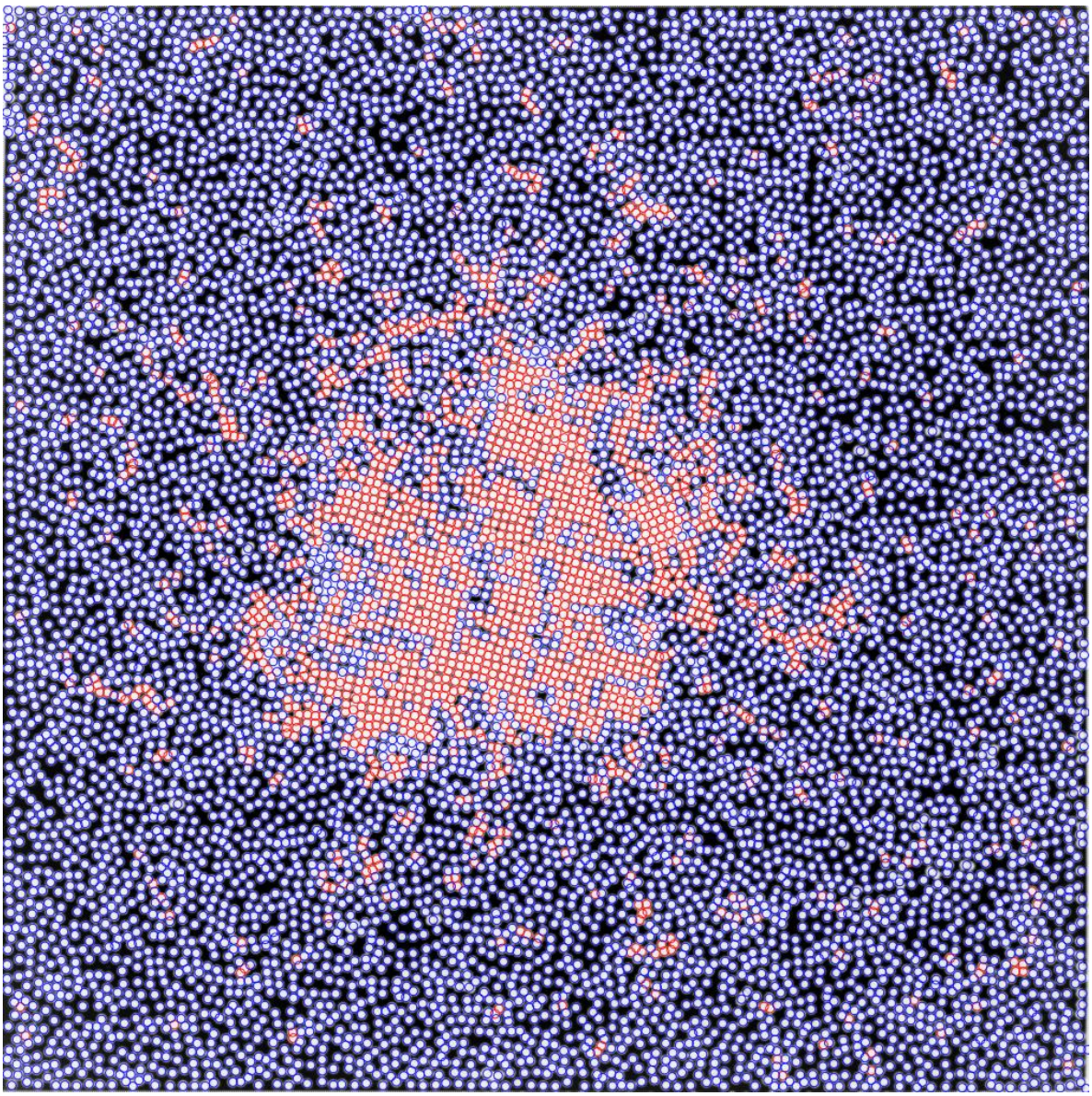


Figura 24: Ejemplo de una sola validación de partículas por distancia entre ellas.

En esta imagen se pueden apreciar más partículas detectadas dentro del área más densa, sin embargo en las áreas menos densas se puede apreciar que algunas partículas detectadas en una ejecución normal ya no están, por lo tanto se puede aconsejar el uso de esta modificación cuando se tenga este tipo de áreas más densas en las imágenes.

En una ejecución normal, como se visualiza en la ilustración 21, se detectan 10.084 partículas mientras que con ésta modificación se logran detectar 10.113 lo que implica un aumento de 29 partículas, todas validadas mediante área de Voronoi e intensidad de imagen, por lo que no son falsos positivos.

4.4.6. Métricas

A continuación se presentan algunas métricas de $PTrack2$ obtenidas con el programa Understand[15]:

Métrica	Valor
Número de clases	35
Número de líneas	7.793
Número de líneas de código	4.048
Número de líneas de comentario	2.296
Número de funciones	298
Ratio de líneas de comentario con líneas de código	0.57

Tabla 6: Métricas generales del proyecto PTrack2.

Para visualizar de cierta manera la complejidad del desarrollo, se generó un diagrama que representa todas las clases organizadas por la estructura de directorios. El tamaño de la clase representa la cantidad de líneas de código mientras que el color más fuerte representa el WMC de la clase[16].

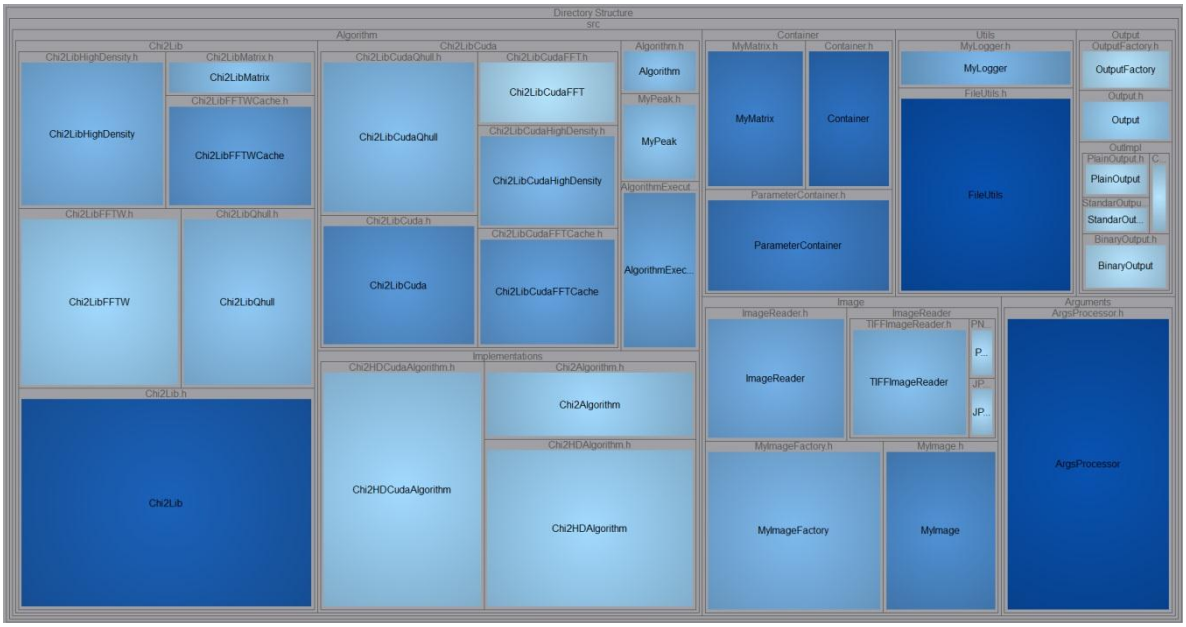


Figura 25: Mapa de métricas para PTrack2, líneas de código – cantidad de métodos.

De la Figura 25 se desprende que las clases más complejas (más grandes y con color más fuerte) son Chi2Lib, ArgsProcessor y FileUtils. De estas tres clases las más importantes son Chi2Lib la cual tiene la mayoría de las funciones para detectar partículas y ArgsProcessor que interpreta los argumentos de ejecución hacia los parámetros necesarios para cada algoritmo de forma automática.

[15] Ver más en http://www.scitools.com/index.php
[16] Ver Antecedentes punto 2.3.1

A continuación se presentan algunas métricas asociadas al desarrollo orientado a objetos.

Clase	LCOM	DIT	CBO	NOC	RFC	WMC	Líneas de Código
Algorithm	100	0	3	3	5	5	22
AlgorithmExecutor	22	0	6	0	9	9	80
ArgsProcessor	80	0	7	0	20	20	336
BinaryOutput	0	1	2	0	4	1	27
Chi2Algorithm	33	1	9	0	8	3	103
Chi2HDAlgorithm	33	1	12	0	8	3	64
Chi2HDCudaAlgorithm	38	1	12	0	8	3	66
Chi2Lib	0	0	3	0	18	18	71
Chi2LibCuda	0	0	3	0	11	11	15
Chi2LibCudaFFT	0	0	4	0	1	1	8
Chi2LibCudaFFTCache	22	0	1	0	10	10	15
Chi2LibCudaHighDensity	0	0	4	0	7	7	18
Chi2LibCudaQhull	0	0	1	0	6	6	19
Chi2LibFFTW	66	0	4	0	3	3	37
Chi2LibFFTWCache	22	0	3	0	10	10	17
Chi2LibHighDensity	0	0	5	0	7	7	21
Chi2LibMatrix	0	0	1	0	5	5	3
Chi2LibQhull	0	0	1	0	5	5	19
ConnectorOutput	0	1	0	0	4	1	1
Container	75	0	0	0	16	16	19
FileUtils	0	0	2	0	20	20	233
ImageReader	0	0	4	3	8	8	118
JPGImageReader	0	1	0	0	10	2	7
MyImage	50	0	4	0	12	12	127
MyImageFactory	0	0	7	0	7	7	197
MyLogger	54	0	2	0	7	7	48
MyMatrix	38	0	0	0	14	14	92
MyPeak	62	0	0	0	5	5	39
Output	0	0	1	4	3	3	26
OutputFactory	0	0	2	0	1	1	25
ParameterContainer	14	0	2	0	14	14	122
PlainOutput	0	1	1	0	4	1	15
PNGImageReader	0	1	0	0	10	2	7
StandarOutput	0	1	1	0	4	1	12
TIFFImageReader	0	1	3	0	14	6	89
Promedio	20,3	0,3	3,1	0,3	8,5	7,1	60,5

Tabla 7: Métricas de clases para PTrack2.

De la tabla anterior se puede concluir que en PTrack2 tiene en promedio buenos indicadores salvo en determinadas clases como por ejemplo Algorithm el cual presenta un LCOM de 100, sin embargo ésta clase intenta ser una interfaz por lo que posee muy pocas líneas de código y no representa mayor problema entenderla. También ArgsProcessor tiene un LCOM de 80 pues contiene mucha lógica y por motivos de tiempo no se refactorizo a un patrón de diseño acorde ,como por ejemplo *Interpreter.*

4.5. *PTrack2Bash*

Para realizar la distribución de $PTrack2$ con el fin de ejecutar cálculos en distintos computadores, se pensó en programar un script teniendo las siguientes ventajas:

1. Programación no invasiva. No se debía tocar el código de $PTrack2$ para realizar la distribución.
2. Reutilizable. Este script podía ser reutilizado para los nuevos programas que se pensaran distribuir, solo se necesitaba un leve cambio en algunas líneas de código.
3. No necesita compilación. Un script es interpretado, por lo que no necesita compilación y cualquier cambio efectuado en él es reflejado en la siguiente ejecución sin necesidad de realizar otras operaciones.
4. Puede realizar llamadas a programas del sistema. Las llamadas a programas dentro del sistema son muy útiles pues eliminan mucha programación e interacción del software.
5. Simple y rápido de programar.

En un principio se utilizó lenguaje **BASH**, sin embargo este no posee utilidad alguna para crear procesos ligeros, solo se pueden crear procesos pesados lo que utilizaba muchos recursos del computador y además no se pueden compartir datos en estos procesos pesados que no fuesen en algún archivo de datos. Por estos motivos es que posteriormente se optimizó el script usando lenguaje **PERL** utilizando procesos livianos y mejorando el algoritmo de distribución. La elección de este lenguaje y no otro como **Python**, fue debido a la similitud entre PERL y BASH haciendo más fácil el paso de un lenguaje a otro.

4.5.1. Arquitectura de software

Este script hace uso de algunos programas que deben estar instalados y bien configurados. Además de un archivo de configuración llamado *PTrack2Bash.conf*.

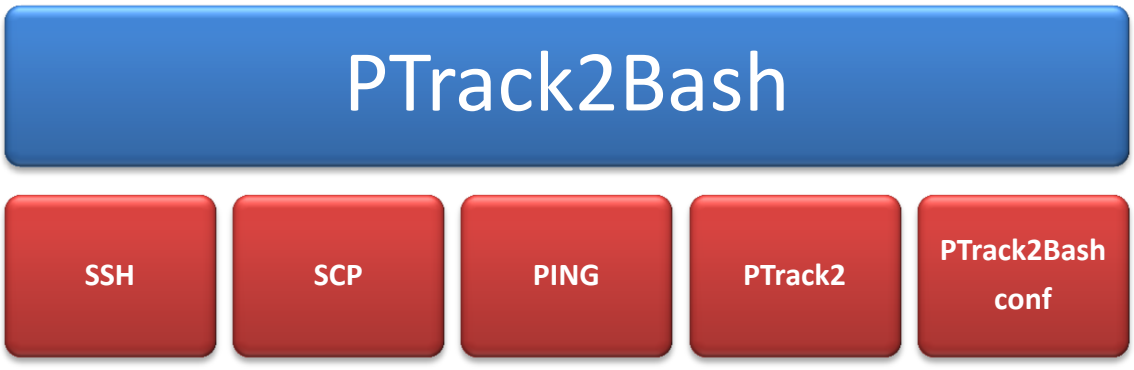


Figura 26: Arquitectura de PTrack2Bash.

4.5.2. Algoritmo de distribución

El algoritmo de distribución que se creó se puede describir mediante el siguiente cuadro escrito en lenguaje normal:

Algoritmo de distribución	
1	Leer archivo de configuración
2	**Por cada servidor en archivo de configuración:**
3	Verificar que responda mediante ping
4	*Si no responde:*
5	Eliminar servidor para uso
6	*Si responde:*
7	Verificar que se pueda conectar mediante SSH usando archivo de llave pública
8	*Si responde:*
9	Almacenar configuración de servidor
10	*Si no responde:*
11	Eliminar servidor para uso
12	Verificar que exista la carpeta de imágenes a leer
13	Agregar todas las imágenes dentro de la carpeta a un Stack global
14	**Por cada servidor ejecutar lo siguiente en paralelo:**
15	Obtener un paquete de imágenes a procesar
16	*Si el paquete de imágenes es mayor a 0:*
17	**Por cada imagen del paquete ejecutar en paralelo:**
18	Enviar imagen a servidor
19	Procesar imagen en servidor
20	Esperar a que los procesos terminen
21	Recuperar archivos de salida
22	Eliminar imágenes enviadas y archivos de salida dentro del servidor
23	Verificar correspondencia de archivos de salida con imágenes
24	*Si no existe correspondencia con ninguna imagen:*
25	Repetir pasos 3 a 11
26	*En caso contrario:*
27	Terminar procesamiento en servidor
28	Esperar el procesamiento en los servidores

Este algoritmo crea varios consumidores de imágenes, en este caso cada servidor de procesamiento, y una estructura de alimentación, que es el Stack de imágenes, visible y modificable por todos los servidores. Así cada servidor se alimenta según su ritmo de trabajo sin necesidad de esperar al resto. Las imágenes se consumen mediante un paquete de imágenes el cual se obtiene a través de una única función que obtiene las imágenes del Stack. Esta función evita condiciones de carrera obteniendo un paquete continuo de imágenes del Stack mediante una sección crítica. Una vez que las imágenes son procesadas y se obtienen los archivos de salida respectivos, se chequea la correspondencia entre un archivo de salida con una imagen mediante otra función similar a la obtención de paquete de imágenes, los archivos tienen como nombre ***xysl####*** y ***ImgA######.tif*** (donde # representa un número) respectivamente, por lo que al Verificar se comparan los números y si no se encuentra una correspondencia, la imagen es devuelta al Stack de imágenes para ser procesada y se deja un registro en otro Stack Historico, si en este último Stack se encuentra mas de N veces la imagen, ésta no se devuelve al Stack de imágenes y pasa a ser una imagen imposible de procesar quedando el registro de esto en un

archivo. En caso de que no exista correspondencia alguna entre imagen y archivo de salida, se verifica que el servidor se encuentre vivo mediante ping y se pueda conectar mediante SSH, en caso de fallar cualquiera de las dos verificaciones, las imágenes vuelven al Stack de imágenes y se elimina el servidor para procesamiento.

4.5.3. Funcionamiento del script

El script $PTrack2Bash$ utiliza primeramente SCP para poder enviar la imagen a procesar al servidor y rescatar los resultados del proceso, luego utiliza SSH para ejecutar $PTrack2$ en el servidor y eliminar las imágenes enviadas y los archivos temporales. Cabe destacar que en cada uno de los servidores de procesamiento, deben estar instalados $PTrack2$, SSH y SCP. Además estos deben tener configurado SSH para poder conectarse usando un archivo llave pública, sin necesidad de escribir una contraseña, y aceptar un mínimo de conexiones persistentes simultáneamente que va a depender de qué tantos archivos se puedan procesar en paralelo en ese servidor en específico.

El archivo de configuración que se creó, indica las direcciones IP de los servidores a conectar, la cantidad de imágenes en paralelo que se van a procesar por cada servidor, el nombre de usuario que se utilizará para conectar, si un servidor trabaja con CUDA y cuántos dispositivos de CUDA se pueden utilizar para procesar las imágenes. Un ejemplo de archivo de configuración se puede ver en el apéndice B[17].

Al procesar con más de un dispositivo CUDA, las imágenes se asignan una por cada dispositivo usando Round Robin o planificación cíclica pudiendo un dispositivo procesar más de una imagen al mismo tiempo, sin embargo se debe tener cuidado al asignar la cantidad de imágenes a procesar en paralelo mediante CUDA pues $PTrack2$ se limita a usar la memoria del dispositivo sin manejar errores de falta de memoria u otros errores que se presenten en la ejecución, en cuyo caso presenta un mensaje de error y la posible explicación de este, sin embargo no los resuelve por sí solo. Es por este motivo que se deben realizar pruebas antes de asignar la cantidad de imágenes en paralelo que puede procesar cada dispositivo CUDA.

El script guarda todo el registro del procesamiento en un archivo llamado *process-log.txt*, también guarda el registro de las imágenes que fallaron en ser procesadas en otro archivo llamado *failed-img-log.txt*. Al dejar de funcionar un servidor y éste ser eliminado para el procesamiento, se deja un mensaje mediante salida estándar y en los dos archivos de registro. El tiempo total de procesamiento se despliega en la salida estándar y al final del archivo de registro de procesamiento.

[17] Información más detallada en https://trac.assembla.com/particle-tracking-2/wiki/config#ConfigurarPTrack2Bash

4.6. *PTrack2BashFolders*

Este script es fundamental para procesar las imágenes que los investigadores obtienen pues, como ya se dijo en la especificación del problema, los investigadores deben procesar miles de imágenes de varios videos por lo que es necesario procesar varias carpetas con imágenes. Es por esta razón que se decidió crear esta utilidad que procesa cada carpeta necesaria en secuencia, evitando tener que lanzar manualmente los procesamientos de cada carpeta por separado. Este script se escribió en ***BASH*** y no ha sido necesario cambiarlo a otro lenguaje debido a su simplicidad.

Viendo ejemplos de carpetas con imágenes se identificó que cada carpeta de video posee un prefijo de la forma ***Acq_A_###*** en donde el símbolo ***#*** representa un dígito y el prefijo puede variar pero la cantidad de dígitos se mantiene, también dentro de éstas carpetas se encuentra generalmente una llamada ***ImgA*** en donde se encuentran las imágenes a procesar y generalmente los resultados se guardan en otro carpeta al mismo nivel que la de imágenes llamada ***XYSL***. Además, en cada video se tienen parámetros distintos de la variable D (Diámetro) y w (Foco) de la partícula ideal[18].

Para ejecutar este script es necesario ingresar el prefijo de cada carpeta y un número inicial y final para procesar. También es necesario crear un archivo de configuración llamado ***PTrack2BashFolders.conf***[19] y ubicarlo dentro de la carpeta en donde se va a ejecutar este script. Dentro del archivo de configuración se encuentran indicados los parámetros D y w de cada carpeta de imágenes, la carpeta general en la cual se encuentran las imágenes (ImgA) y la carpeta de salida de datos (XYSL). Los parámetros D y w se distinguen mediante un índice que identifica a la carpeta de video asociada con estos parámetros, por ejemplo para la carpeta ***Acq_A_003*** se tiene la variable ***Data_D [3]= 8.38433*** y ***Data_W [3]= 1.36541.*** Un ejemplo de archivo de configuración se presenta en el apéndice B.

[18] Ver Antecedentes - Ajuste de mínimos cuadrados basado en convolución[1]

[19] Más información en https://trac.assembla.com/particle-tracking-2/wiki/config#ConfigurarPTrack2BashFolders

La estructura necesaria para ejecutar $PTrack2BashFolders$ y la estructura encontrada es la siguiente:

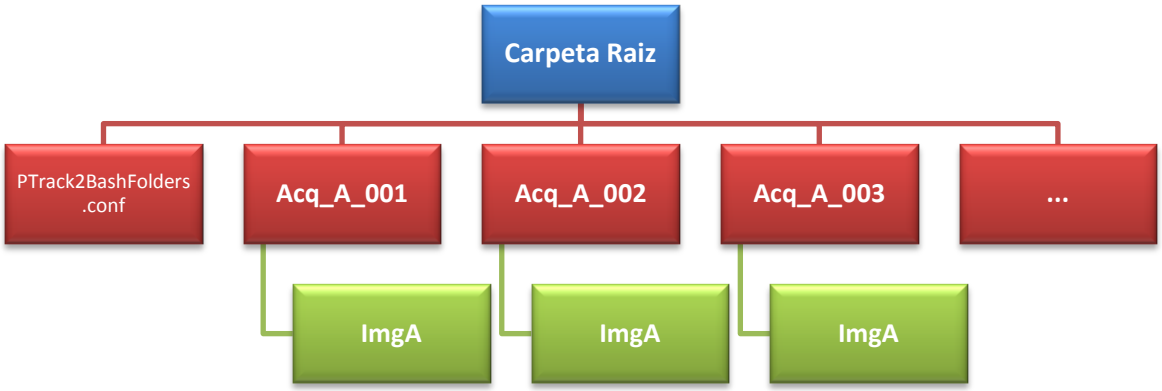


Figura 27: Estructura de archivos para ejecutar PTrack2BashFolders.

Las carpetas principales deben poseer un sufijo seguido de 3 dígitos, y dentro de éstas debe existir una carpeta de imágenes, la que debe tener el mismo nombre para todas las demás carpetas a procesar.

4.7. $PTrack2Folder$ - $PTrack2FolderCuda$

Este script se generó para facilitar el procesamiento de una carpeta de imágenes en particular y trabaja solo en un computador usando múltiples instancias de $PTrack2$ por lo que no requiere de archivo de configuración ni de otros computadores para el cálculo. Como parámetros de ejecución se requiere la carpeta de imágenes a procesar, la carpeta de salida de datos, los parámetros D y w y finalmente la cantidad de procesos en paralelo que se desean ejecutar. Para la versión en CUDA se debe establecer como último parámetro la cantidad de dispositivos CUDA a utilizar.

4.8. Biblioteca $Chi2HD\_Cuda$

La biblioteca $Chi2HD\_Cuda$ fue creada especialmente para hacer uso de la tecnología CUDA y ser ocupada para los cálculos en la implementación del algoritmo *Chi2HD* teniendo las funciones y las estructuras de datos lo más encapsuladas posibles con el fin de no utilizar el compilador ***NVCC*** en la generación de $PTrack2$, así se tiene código independiente al compilador [13] además de evitar problemas de compilación y enlace con otras bibliotecas utilizadas.

4.8.1. Estructuras de datos

Todas las estructuras de datos que se utilizan tienen un carácter Dual, vale decir, al crearlas residen en memoria del dispositivo pero si se desea se pueden copiar a memoria de Host[15] para poder trabajar con ellas fuera de la biblioteca. La dualidad tiene sentido para poder realizar cálculos simples o modificaciones pequeñas que no tiene sentido ejecutarlas en el dispositivo. También posibilita el control de memoria utilizable por el dispositivo, ya que ésta es muy limitada.

Las principales estructuras son ***cuMyMatrix***, ***cuMyMatrixi*** y ***cuMyPeakArray***. Las dos primeras representan una matriz de puntos flotantes y de enteros. La principal razón de por qué se encuentran separadas y no se creó una estructura tipo template fue debido a la conservación del encapsulamiento. Una estructura template usando CUDA necesariamente debe ser compilada usando ***NVCC***, en cambio, usando estructuras separadas y bien tipadas, la compilación con NVCC no es necesaria. La tercera estructura, *cuMyPeakArray*, pretende ser un arreglo dinámico de partículas detectadas, en donde se puede agregar al final otra estructura similar, ser ordenada y filtrada dejando solamente las partículas válidas. La estructura que representa la partícula detectada es de tipo ***cuMyPeak*** en donde se almacena la posición, la intensidad de imagen chi2 y el estado de materia entre otras cosas.

4.8.2. Diagrama de clases

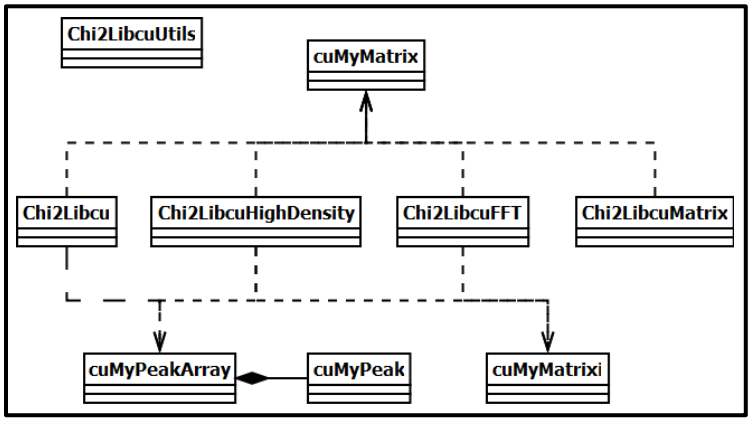


Figura 28: Diagrama de clases para la biblioteca Chi2HD_cuda simplificado.

La clase ***Chi2LibcuUtils*** se usa en todas las demás clases, ya que es la que se encarga de definir la dimensión de la grilla y los bloques para ejecutar un *Kernel*. Además se encarga de sincronizar la ejecución de los *Kernel* y manejar errores de memoria. Las demás clases son colecciones de funciones estáticas que se utilizan para ejecutar el algoritmo *Chi2HD*, exceptuando las estructuras de datos ya mencionadas anteriormente.

Muchas de las funciones que se implementaron en CUDA se pudieron reducir de forma directa para su ejecución, mientras que otras, en particular las de generación de matrices auxiliares, se lograron obtener los resultados requeridos mediante múltiples llamadas a la función, logrando sobrepasar problemas de sincronización y Race conditions.

4.8.3. Codificación en CUDA

A continuación se presentan algunos Kernels de ejecución y su símil escrito en C++. Estos ejemplos representan los tres tipos de codificación en CUDA que se utilizaron en el desarrollo, siendo el primero el más común y el último una única vez. Generalmente cada Kernel se ejecuta con la cantidad máxima de threads disponibles ajustando la dimensión de la grilla de ejecución para adaptarse a la cantidad de elementos a procesar. Las dimensiones se pueden calcular como sigue:

$$Tamaño_{Arreglo} \leq Threads_{max} * Dimension_{Grilla}$$

Generación de la imagen de partícula ideal

La generación de la imagen de partícula ideal se realiza creando una matriz de datos de tamaño $N \times N$ en donde cada celda se le asigna un valor específico de acuerdo a su ubicación y los parámetros entregados. En la versión de C++ se recorre cada elemento de la matriz usando dos ciclos *for*. En CUDA se recorre cada elemento de un arreglo unidimensional en forma paralela que es mapeado a una matriz posteriormente.

Implementación en C++

```
1  MyMatrix<double> Chi2Lib::generateKernel(unsigned int ss, unsigned int os, double d, double w){
2
3      MyLogger::log()->debug(
                   "[Chi2Lib][generateKernel] Building kernel with ss=%i; os=%i; d=%f; w=%f",
                   ss,os,d,w);
4
5      MyMatrix<double> kernel(ss,ss);
6      for(unsigned int x=0; x < ss; ++x)
7          for(unsigned int y=0; y < ss; ++y){
8                  double absolute  = abs(sqrt((x-os)*(x-os) + (y-os)*(y-os)));
9                  double ipfval    = (1.0 - tanh((absolute - d/2.0)/w))/2.0;
10                 kernel.at(x,y)   = ipfval;
11         }
12
13     MyLogger::log()->debug("[Chi2Lib][generateKernel] Kernel successfully built");
14     return kernel;
15 }
```

Implementación en CUDA

```
1   __global__ void __gen_kernel(
            float* arr, unsigned int size, unsigned int ss, unsigned int os, float d, float w){
2           int idx = blockIdx.x * blockDim.x + threadIdx.x;
3           if(idx < size){
4                   float absolute   = abs(sqrtf( (idx%ss-os)*(idx%ss-os) + (idx/ss-os)*(idx/ss-os) ));
5                   arr[idx]         = (1.0f - tanhf((absolute - d/2.0f)/w))/2.0f;
6           }
7   }
8
9   cuMyMatrix Chi2Libcu::gen_kernel(unsigned int ss, unsigned int os, float d, float w){
10          cuMyMatrix kernel(ss,ss);
11          dim3 dimGrid(_findOptimalGridSize(kernel.size()));
12          dim3 dimBlock(_findOptimalBlockSize(kernel.size()));
13          __gen_kernel<<<dimGrid, dimBlock>>>(kernel.devicePointer(), kernel.size(), ss, os, d, w);
14          checkAndSync();
15
16          return kernel;
17  }
```

Búsqueda y asignación de mínimos locales

Para encontrar los peaks resultantes de las imágenes Chi2 se verifica que cada celda de la matriz que representa la imagen tenga un valor mayor que cierta tolerancia y luego se verifica que en la vecindad de esa celda el valor sea el máximo.

En la implementación en C++ el algoritmo es básicamente el mismo recorriendo toda la imagen Chi2 usando dos ciclos *for* y para verificar si una celda es un mínimo local se utiliza la misma técnica recorriendo la vecindad de la celda, así si una celda representa un mínimo local las coordenadas y el valor de la celda se almacenan en un vector.

En la implementación en CUDA esto es un poco diferente ya que se realiza en forma paralela, entonces, se verifica que cada celda de la imagen sea un mínimo local asignando a una matriz auxiliar de tipo boolean el valor true en caso afirmativo, aumentando un contador de forma atómica en 1 y false en caso contrario teniendo ambas matrices los mismos índices, así si la matriz auxiliar con índice *J* indica true, la celda de la matriz Chi2 con índice *j* representa un mínimo local. Luego de completar los valores de la matriz auxiliar se Crea un arreglo de Peaks con tamaño igual al del contador atómico, luego se reinicia el contador a cero y se recorre en forma paralela la matriz auxiliar verificando si el valor de esta es true, creándose un peak y guardándolo en la posición que indique el contador para luego ser incrementado en 1 de manera atómica. Así se crea el arreglo de Peaks de forma secuencial pero obteniendo los Peaks en forma paralela.

Implementación en C++

```
1   vector<MyPeak> Chi2Lib::getPeaks(
            MyMatrix<double> *img, int threshold, int mindistance, int minsep, bool use_threads){
2
3       MyLogger::log()->debug(
            "[Chi2Lib][getPeaks] Obtaining peaks threshold=%i; mindistance=%i; minsep=%i",
            threshold, mindistance, minsep);
4
5       vector<MyPeak> peaks;
6       for(unsigned int x=0; x < img->sX(); ++x){
7           for(unsigned int y=0; y < img->sY(); ++y){
8                   double img_xy = img->getValue(x,y);
9                   if(img_xy > threshold){
10                          if(findLocalMinimum(img, x, y, minsep)){
11                                  MyPeak local(x,y, img_xy);
12                                  peaks.push_back(local);
13                          }
14                  }
15          }
16      }
17
18      unsigned int detected = peaks.size();
19      MyLogger::log()->info(
                    "[Chi2Lib][getPeaks] Peaks detected: %i of %i", detected, img->sX()*img->sY());
20          // Validar peaks ...
21  }
22
23  bool Chi2Lib::findLocalMinimum(
                            MyMatrix<double> *img, unsigned int imgX, unsigned int imgY, int minsep){
24
25      for(int localX = minsep; localX >= -minsep; --localX){
26          for(int localY = minsep; localY >= -minsep; --localY){
27                  if(!(localX == 0 && localY == 0)){
28                          int currentX = (imgX+localX);
29                          int currentY = (imgY+localY);
30                          if(currentX < 0)
31                                  currentX = img->sX() + currentX;
32                          if(currentY < 0)
33                                  currentY = img->sY() + currentY;
34                          currentX = (currentX)% img->sX();
35                          currentY = (currentY)% img->sY();
36                          if(img->at(imgX, imgY) <= img->getValue(currentX, currentY))
37                                  return false;
38                  }
39          }
40      }
41      return true;
42  }
```

Implementación en CUDA

```
1   __device__ bool __findLocalMinimum(
                    float* arr, unsigned int sizeX, unsigned int sizeY,
                    unsigned int imgX, unsigned int imgY, unsigned int idx, int minsep){
2
3           for(int localX = minsep; localX >= -minsep; --localX){
4                   for(int localY = minsep; localY >= -minsep; --localY){
5                           if(!(localX == 0 && localY == 0)){
6                                   int currentX = (imgX+localX);
7                                   int currentY = (imgY+localY);
8
9                                   if(currentX < 0)
10                                          currentX = sizeX + currentX;
11                                  if(currentY < 0)
12                                          currentY = sizeY + currentY;
13
14                                  currentX = (currentX)% sizeX;
15                                  currentY = (currentY)% sizeY;
16                                  if(arr[idx] <= arr[currentX*sizeX+currentY]){
17                                          return false;
18                                  }
19                          }
20                  }
21          }
22
23          return true;
24  }
25
26  __global__ void __findMinimums(
                            float* arr, unsigned int sizeX, unsigned int sizeY,
                            int threshold, int minsep, bool* out, int* counter){
27
28          int idx = blockIdx.x * blockDim.x + threadIdx.x;
29          if(idx >= sizeX*sizeY)
30                  return;
31
32          int imgX = (int)floorf(idx/sizeY);
33          int imgY = idx%sizeX;
34
35          if(arr[idx] > threshold){
36                  if(__findLocalMinimum(arr, sizeX, sizeY, imgX, imgY, idx, minsep)){
37                          out[idx] = true;
38                          atomicAdd(&counter[0], 1);
39                  }else{
40                          out[idx] = false;
41                  }
42          }
43  }
44
45  __global__ void __fillPeakArray(
                            float* img, bool* peaks_detected, unsigned int sizeX, unsigned int sizeY,
                            cuMyPeak* peaks, int* counter){
46          int idx = blockIdx.x * blockDim.x + threadIdx.x;
47          if(idx < sizeX*sizeY && peaks_detected[idx]){
48                  cuMyPeak peak;
49                  peak.x = (int)floorf(idx/sizeY);
50                  peak.y = idx%sizeX;
51                  peak.chi_intensity = img[idx];
52                  peak.fx = peak.x;
53                  peak.fy = peak.y;
54                  peak.dfx = peak.dfy = 0;
55                  peak.solid = false;
56                  peak.valid = true;
57                  peaks[atomicAdd(&counter[0], 1)] = peak;
58          }
59  }
```

```
60  cuMyPeakArray Chi2Libcu::getPeaks(cuMyMatrix *arr, int threshold, int mindistance, int minsep){
61          DualData<bool> minimums(arr->size(), false);
62          DualData<int> counter;
63          // Encontrar Minimos
64          dim3 dimGrid(_findOptimalGridSize(arr->size()));
65          dim3 dimBlock(_findOptimalBlockSize(arr->size()));
66
67          __findMinimums<<<dimGrid, dimBlock>>>(
68                                      arr->devicePointer(), arr->sizeX(), arr->sizeY(),
69                                      threshold, minsep, minimums.devicePointer(),
70                                      counter.devicePointer());
71          checkAndSync();
72
73          // Contador de datos
74          counter.copyToHost();
75
76          // Alocar datos
77          cuMyPeakArray peaks(counter[0]);
78          counter.reset(0);
79
80          __fillPeakArray<<<dimGrid, dimBlock>>>(
81                          arr->devicePointer(), minimums.devicePointer(), arr->sizeX(), arr->sizeY(),
82                          peaks.devicePointer(), counter.devicePointer());
83          checkAndSync();
84          return peaks;
85  }
```

Cálculo de diferencia entre imagen original y generada mediante los peaks encontrados

El cálculo de la diferencia se realiza recorriendo toda la imagen original, generando los valores de las celdas correspondientes de una imagen generada mediante los peaks encontrados y restando ese valor al valor de la celda actual de la imagen original generando una matriz de diferencia, además se suma cada valor cuadrado de la diferencia a una variable para obtener el error cuadrático. De esta forma en esencia nunca se genera una imagen sino que solo se generan los valores de ésta.

La implementación en C++ posee dos maneras de realizar el cálculo, una manera secuencial y otra mediante el uso de threads en donde se calcula en dos hilos de ejecución la mitad de los valores de la diferencia por cada hilo, acelerando un poco el proceso de generación de la matriz de diferencia.

La implementación en CUDA realiza en forma paralela éste proceso, sin embargo el cálculo del error cuadrático requiere de una técnica llamada reducción en donde cada bloque de ejecución almacena en un arreglo en memoria compartida (Shared Memory) los valores cuadrados de las celdas de diferencia generadas por ese bloque, luego la mitad de los threads realiza una suma del valor asignado por ese thread más el valor asignado por el thread con índice mitad más uno de la dimensión de threads, los threads se sincronizan y luego la mitad de estos siguen sumando el valor actual más el valor actual del thread con índice mitad más uno partido en dos y así consecutivamente. Hasta llegar a que el thread con índice cero ha sumado todo el arreglo de memoria compartida, así ese valor se asigna a un arreglo de memoria global con índice igual al bloque ejecutado. Una vez ejecutados todos los bloques se procede a sumar de forma secuencial en Host el arreglo de memoria global. Esto se puede explicar de mejor manera en la siguiente forma.

Supongamos un arreglo de memoria compartida de tamaño 8 llamado a con los siguientes valores

Índice	0	1	2	3	4	5	6	7
Valor	5	10	15	20	25	30	35	40

Dentro de la ejecución del kernel estos valores se pueden sumar de manera secuencial por un solo thread o bien aplicando una reducción en donde se aprovecha el paralelizmo de CUDA. Así cada thread se mapea con el arreglo de memoria compartida de manera que el índice del thread calza con el índice del arreglo en memoria compartida, por lo que al empezar la secuencia los valores corresponden al mismo arreglo.

Thread	0	1	2	3	4	5	6	7
Iteración 0	$a[0] = 5$	$a[1] = 10$	$a[2] = 15$	$a[3] = 20$	$a[4] = 25$	$a[5] = 30$	$a[6] = 35$	$a[7] = 40$

Luego la mitad de los threads suman el valor propio dentro del arreglo más el valor del índice propio más el de la mitad más uno.

Thread	0	1	2	3	4	5	6	7
Iteración 0	$a[0] = 5$	$a[1] = 10$	$a[2] = 15$	$a[3] = 20$	$a[4] = 25$	$a[5] = 30$	$a[6] = 35$	$a[7] = 40$
Iteración 1	$a[0]$ $+=$ $a[0+4]$	$a[1]$ $+=$ $a[1+4]$	$a[2]$ $+=$ $a[2+4]$	$a[3]$ $+=$ $a[3+4]$	-	-	-	-

Luego la mitad de los threads que se ejecutaron suman nuevamente dividiendo el índice por la mitad, en este caso 4/2 = 2.

Thread	0	1	2	3	4	5	6	7
Iteración 0	$a[0] = 5$	$a[1] = 10$	$a[2] = 15$	$a[3] = 20$	$a[4] = 25$	$a[5] = 30$	$a[6] = 35$	$a[7] = 40$
Iteración 1	$a[0]$ $+=$ $a[0+4]$	$a[1]$ $+=$ $a[1+4]$	$a[2]$ $+=$ $a[2+4]$	$a[3]$ $+=$ $a[3+4]$	-	-	-	-
Iteración 2	$a[0]$ $+=$ $a[0+2]$	$a[1]$ $+=$ $a[1+2]$	-	-	-	-	-	-

Ahora se repite salvo que el índice cambia a 2/2 = 1

Thread	0	1	2	3	4	5	6	7
Iteración 0	$a[0] = 5$	$a[1] = 10$	$a[2] = 15$	$a[3] = 20$	$a[4] = 25$	$a[5] = 30$	$a[6] = 35$	$a[7] = 40$
Iteración 1	$a[0]$ $+=$ $a[0+4]$	$a[1]$ $+=$ $a[1+4]$	$a[2]$ $+=$ $a[2+4]$	$a[3]$ $+=$ $a[3+4]$	-	-	-	-
Iteración 2	$a[0]$ $+=$ $a[0+2]$	$a[1]$ $+=$ $a[1+2]$	-	-	-	-	-	-
Iteración 3	$a[0]$ $+=$ $a[0+1]$	-	-	-	-	-	-	-

Luego si nos devolvemos en la secuencia el valor final de $a[0]$ queda como:

$$\boldsymbol{a[0] = a[0] + a[1]}$$

Pero $a[0] = a[0] + a[2]$ y $a[1] = a[1] + a[3]$

$$\boldsymbol{a[0] = (a[0] + a[2]) + (a[1] + a[3])}$$

Así mismo $a[0] = a[0] + a[4]$; $a[2] = a[2] + a[6]$; $a[1] = a[1] + a[5]$ y $a[3] = a[3] + a[7]$

$$\boldsymbol{a[0] = \big((a[0] + a[4]) + (a[2] + a[6])\big) + \big((a[1] + a[5]) + (a[3] + a[7])\big)}$$

Luego

$$\boldsymbol{a[0] = a[0] + a[4] + a[2] + a[6] + a[1] + a[5] + a[3] + a[7]}$$

$$\boldsymbol{a[0] = 5 + 25 + 15 + 35 + 10 + 30 + 20 + 40}$$

$$\boldsymbol{a[0] = 180}$$

Así la suma se realizó paralelamente en $\mathcal{O}(\log N)$ con $N = 8$ que es el tamaño del arreglo.

Implementación en C++

```
1  double Chi2Lib::computeDifference(
                          MyMatrix<double> *img, MyMatrix<double> *grid_x, MyMatrix<double> *grid_y,
                          double d, double w, MyMatrix<double> *diffout, bool use_threads){
2      if(use_threads){
3          PartitionDiff p1;
4          p1.x1 = 0; p1.x2 = img->sX()/2;
5          p1.y1 = 0; p1.y2 = img->sY();
6          p1.d = d; p1.w = w;
7          p1.img = img; p1.grid_x = grid_x; p1.grid_y = grid_y;
8          p1.diffout = diffout;
9
10         PartitionDiff p2;
11         p2.x1 = img->sX()/2; p2.x2 = img->sX();
12         p2.y1 = 0; p2.y2 = img->sY();
13         p2.d = d; p2.w = w;
14         p2.img = img; p2.grid_x = grid_x; p2.grid_y = grid_y;
15         p2.diffout = diffout;
16
17         pthread_t thread1, thread2;
18         pthread_create(&thread1, NULL, computeDifferenceThread, (void *)&p1);
19         pthread_create(&thread2, NULL, computeDifferenceThread, (void *)&p2);
20
21         pthread_join(thread1, NULL);
22         pthread_join(thread2, NULL);
23
24         MyLogger::log()->info("[Chi2Lib][computeDifference] Chi2 Error: %f", p1.err + p2.err);
25         return p1.err + p2.err;
26
27     }else{
28         MyLogger::log()->debug("[Chi2Lib][computeDifference] Computing Chi2 Difference");
29         double chi2err = 0.0;
30         for(unsigned int x=0; x < img->sX(); ++x){
31             for(unsigned int y=0; y < img->sY(); ++y){
32                 double x2y2 = sqrt(1.0*grid_x->getValue(x,y)*grid_x->getValue(x,y)
                                       + 1.0*grid_y->getValue(x,y)*grid_y->getValue(x,y));
33                 double temp = ((1.0-tanh( (x2y2-d/2.0)/w )) - 2.0*img->getValue(x,y))/2.0;
34
35                 diffout->at(x,y) = temp;
36                 chi2err += temp*temp;
37             }
38         }
39         MyLogger::log()->info("[Chi2Lib][computeDifference] Chi2 Error: %f", chi2err);
40         return chi2err;
41     }
42 }
```

```
43  void * Chi2Lib::computeDifferenceThread( void* ptr){
44      PartitionDiff *part = (PartitionDiff*) ptr;
45      MyMatrix<double> *img = part->img;
46      MyMatrix<double> *grid_x = part->grid_x;
47      MyMatrix<double> *grid_y = part->grid_y;
48      double d = part->d;
49      double w = part->w;
50      MyMatrix<double> *diffout = part->diffout;
51
52      MyLogger::log()->debug("[Chi2Lib][computeDifferenceThread] Computing Chi2 Difference");
53      double chi2err = 0.0;
54      for(unsigned int x=part->x1; x < part->x2; ++x){
55          for(unsigned int y=part->y1; y < part->y2; ++y){
56              double x2y2 = sqrt(1.0*grid_x->getValue(x,y)*grid_x->getValue(x,y)
                                + 1.0*grid_y->getValue(x,y)*grid_y->getValue(x,y));
57              double temp = ((1.0-tanh( (x2y2-d/2.0)/w )) - 2.0*img->getValue(x,y))/2.0;
58
59              diffout->at(x,y) = temp;
60              chi2err += temp*temp;
61          }
62      }
63      MyLogger::log()->debug("[Chi2Lib][computeDifferenceThread] Chi2 Error: %f", chi2err);
64
65      part->err = chi2err;
66      return 0;
67  }
```

Implementación en CUDA

```
1   __global__ void __computeDifference(
                    float* img, float* grid_x, float* grid_y, float d, float w, float* diffout,
                    unsigned int size, float* sum_reduction){
2       extern __shared__ float sharedData[];
3       int idx = blockIdx.x * blockDim.x + threadIdx.x;
4       unsigned int tid = threadIdx.x;
5
6       float temp = 0;
7       if(idx < size){
8           float x2y2 = sqrtf(1.0f*grid_x[idx]*grid_x[idx] + 1.0f*grid_y[idx]*grid_y[idx]);
9           temp = ((1.0f-tanhf( (x2y2-d/2.0)/w )) - 2.0f*img[idx])/2.0f;
10          diffout[idx] = temp;
11      }
12
13      // Calcular la suma cuadrada
14      sharedData[tid] = temp*temp;
15      __syncthreads();
16      for(unsigned int s=blockDim.x/2; s>0; s>>=1){
17          if(tid < s)
18              sharedData[tid] += sharedData[tid + s];
19          __syncthreads();
20      }
21
22      if(tid == 0){
23          sum_reduction[blockIdx.x] = sharedData[0];
24      }
25  }
26
27  float Chi2Libcu::computeDifference(
28                          cuMyMatrix *img, cuMyMatrix *grid_x, cuMyMatrix *grid_y, float d, float w,
29                          cuMyMatrix *diffout){
30      unsigned int griddim = _findOptimalGridSize(img->size());
31      unsigned int blockdim = _findOptimalBlockSize(img->size());
32      DualData<float> sum_reduction(griddim);
33      sum_reduction.reset(0);
34
35      dim3 dimGrid(griddim);
36      dim3 dimBlock(blockdim);
37      __computeDifference<<<dimGrid, dimBlock, blockdim*sizeof(float)>>>(
38                          img->devicePointer(), grid_x->devicePointer(), grid_y->devicePointer(),
39                          d, w, diffout->devicePointer(), img->size(),
40                          sum_reduction.devicePointer());
41      checkAndSync();
42      sum_reduction.copyToHost();
43
44      float total = 0;
45      for(unsigned int i=0; i < sum_reduction.size(); ++i){
46          total = total + sum_reduction[i];
47      }
48      return total;
49  }
```

Generación de Matrices auxiliares

Las matrices auxiliares son una parte importante para el proceso de detección de partículas, estas matrices auxiliares intentan representar las partículas detectadas dentro del espacio de imagen, cada matriz almacena la distancia en X, distancia en Y e índice del peak más cercano. La generación de las matrices se inicia recorriendo cada peak detectado, por cada peak se recorre una vecindad cercana, y se calcula la distancia entre el peak y la posición actual recorrida de la vecindad, si la distancia es menor a la distancia actual de la vecindad se almacenan las nuevas distancias en las matrices respectivas y el índice del peak dentro del arreglo de peaks. Así se tiene una representación dividida por dimensiones.

La implementación en C++ es directa y secuencial, sin embargo se pudo acelerar el proceso usando dos threads limitando las asignaciones a la mitad de las matrices auxiliares.

La implementación en CUDA tiene la desventaja de que en cada ejecución muchos valores se sobreponen entre si debido a las altas condiciones de paralelización dejando distancias no necesariamente menores. Sin embargo se pudo sobrepasar este problema mediante la ejecución de varias iteraciones, así en una pasada quedan algunos valores mal asignados, luego en una segunda pasada, la mayoría de los valores están correctos debido a que los valores se encuentran actualizados y existen menos probabilidades de tener race conditions. Así por cada pasada sucesiva los problemas de llenado se van corrigiendo obteniendo buenos valores para ser usados. Estas pasadas sucesivas si bien afectan el rendimiento redujeron el tiempo de codificación y resolución del problema, siendo el tiempo de ejecución, igualmente mucho menor que la versión secuencial y paralela implementada en C++.

Implementación en C++

```
void Chi2Lib::generateGridImpl(
        vector<MyPeak> *peaks, unsigned int shift, MyMatrix<double> *img, int x1, int x2, int y1,
        int y2, MyMatrix<double> *grid_x, MyMatrix<double> *grid_y, MyMatrix<int> *over){
    unsigned int half=(shift+2);
    MyLogger::log()->debug("[Chi2Lib][generateGridImpl] Half=%i, SS=%i", half, shift);
    unsigned int counter = 0;
    int currentX, currentY;
    double currentDistance = 0.0;
    double currentDistanceAux = 0.0;

    if(!peaks->empty())
    for(int npks = peaks->size()-1; npks >= 0; npks--){
        for(unsigned int localX=0; localX < 2*half+1; ++localX)
            for(unsigned int localY=0; localY < 2*half+1; ++localY){
                MyPeak currentPeak = peaks->at(npks);
                currentX = (int)round(currentPeak.px) - shift + (localX - half);
                currentY = (int)round(currentPeak.py) - shift + (localY - half);

                if( x1 <= currentX && currentX < x2 && y1 <= currentY && currentY < y2 ){
                    currentDistance =
                    sqrt(grid_x->getValue(currentX, currentY)*grid_x->getValue(currentX, currentY)
                       + grid_y->getValue(currentX, currentY)*grid_y->getValue(currentX, currentY));

                    currentDistanceAux =
                         sqrt(1.0*(1.0*localX-half+currentPeak.x - currentPeak.px)*(1.0*localX-
                         half+currentPeak.x - currentPeak.px) + 1.0*(1.0*localY-half+currentPeak.y -
                         currentPeak.py)*(1.0*localY-half+currentPeak.y - currentPeak.py));

                    if(currentDistance >= currentDistanceAux){
                        over->at(currentX, currentY) = npks+1;
                        grid_x->at(currentX, currentY) =
                                    (1.0*localX-half+currentPeak.x)-currentPeak.px;
                        grid_y->at(currentX, currentY) =
                                    (1.0*localY-half+currentPeak.y)-currentPeak.py;
                        counter++;
                    }
                }
            }
    }
    MyLogger::log()->debug("[Chi2Lib][generateGridImpl] Auxiliary Matrix Generation Complete");
    MyLogger::log()->debug("[Chi2Lib][generateGridImpl] Total pgrid= %f; Counter= %i",
                                            1.0*counter/peaks->size(), counter);
}
void Chi2Lib::generateGrid(
        vector<MyPeak> *peaks, unsigned int shift, MyMatrix<double> *img, MyMatrix<double> *grid_x,
        MyMatrix<double> *grid_y, MyMatrix<int> *over,                    bool use_threads){
    MyLogger::log()->debug("[Chi2Lib][generateGrid] Generating Auxiliary Matrix");
    unsigned int maxDimension = img->sX() > img->sY() ? img->sX() : img->sY();
    MyLogger::log()->debug("[Chi2Lib][generateGrid] Grid Size: %ix%i", grid_x->sX(), grid_x->sY());
    grid_x->reset(maxDimension); grid_y->reset(maxDimension);
    over->reset(0);
    MyLogger::log()->debug("[Chi2Lib][generateGrid] Matrix Reset");
    if(use_threads){
        PartitionGrid p1; p1.shift = shift;
        p1.peaks = peaks; p1.grid_x = grid_x; p1.grid_y = grid_y; p1.over = over;
        p1.img = img;
        p1.x1 = 0; p1.x2 = img->sX()/2;
        p1.y1 = 0; p1.y2 = img->sY();

        PartitionGrid p2; p2.shift = shift;
        p2.peaks = peaks; p2.grid_x = grid_x; p2.grid_y = grid_y; p2.over = over;
        p2.img = img;
        p2.x1 = img->sX()/2; p2.x2 = img->sX();
        p2.y1 = 0; p2.y2 = img->sY();

        pthread_t thread1, thread2;
        pthread_create(&thread1, NULL, generateGridThread, (void *)&p1);
        pthread_create(&thread2, NULL, generateGridThread, (void *)&p2);

        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
    }else
        generateGridImpl(peaks, shift, img, 0, img->sX(), 0 , img->sY(), grid_x, grid_y, over);
}
void *Chi2Lib::generateGridThread( void* ptr){
    PartitionGrid* part = (PartitionGrid *)ptr;
    generateGridImpl(part->peaks, part->shift, part->img, part->x1, part->x2, part->y1, part->y2,
                        part->grid_x, part->grid_y, part->over);
    return 0;
}
```

Implementación en CUDA

```
__global__ void __generateGrid(
        cuMyPeak* peaks, unsigned int peaks_size, unsigned int shift, float* grid_x, float* grid_y,
        int* over , unsigned int sizeX, unsigned int sizeY){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx >= peaks_size)
        return;

    unsigned int half=(shift+2);
    int currentX, currentY;
    double currentDistance = 0.0;
    double currentDistanceAux = 0.0;

    if(peaks_size != 0){
        cuMyPeak currentPeak = peaks[idx];
        for(unsigned int localX=0; localX < 2*half+1; ++localX){
            for(unsigned int localY=0; localY < 2*half+1; ++localY){
                currentX = (int)round(currentPeak.fx) - shift + (localX - half);
                currentY = (int)round(currentPeak.fy) - shift + (localY - half);

                if( 0 <= currentX && currentX < sizeX && 0 <= currentY && currentY < sizeY ){
                    int index = currentX+sizeY*currentY;

                    currentDistance =
                            grid_x[index]*grid_x[index] + grid_y[index]*grid_y[index];

                    currentDistanceAux =
                         1.0f*(1.0f*localX-half+currentPeak.x - currentPeak.fx)*(1.0f*localX-
                         half+currentPeak.x - currentPeak.fx) +
                         1.0f*(1.0f*localY-half+currentPeak.y - currentPeak.fy)*(1.0f*localY-
                         half+currentPeak.y - currentPeak.fy);

                    if(currentDistance >= currentDistanceAux){
                        over[index] = idx+1;
                        grid_x[index] = (1.0f*localX-half+currentPeak.x)-currentPeak.fx;
                        grid_y[index] = (1.0f*localY-half+currentPeak.y)-currentPeak.fy;
                    }
                }
            }
        }
    }
}

void Chi2Libcu::generateGrid(
                 cuMyPeakArray* peaks, unsigned int shift, cuMyMatrix* grid_x, cuMyMatrix* grid_y,
                 cuMyMatrixi* over){
    unsigned int maxDimension = grid_x->sizeX() > grid_x->sizeY() ?
                                                             grid_x->sizeX() : grid_x->sizeY();
    grid_x->reset(maxDimension);
    grid_y->reset(maxDimension);
    over->reset(0);

    dim3 dimGrid(_findOptimalGridSize(peaks->size()));
    dim3 dimBlock(_findOptimalBlockSize(peaks->size()));
    __generateGrid<<<dimGrid, dimBlock>>>(
                 peaks->devicePointer(), peaks->size(), shift, grid_x->devicePointer(),
                 grid_y->devicePointer(), over->devicePointer(), grid_x->sizeX(), grid_x->sizeY());
    checkAndSync();
    // Con una sola pasada hay una desviación estándar de 3.06 en los peaks totales

    __generateGrid<<<dimGrid, dimBlock>>>(
                 peaks->devicePointer(), peaks->size(), shift, grid_x->devicePointer(),
                 grid_y->devicePointer(), over->devicePointer(), grid_x->sizeX(), grid_x->sizeY());
    checkAndSync();

    __generateGrid<<<dimGrid, dimBlock>>>(
                 peaks->devicePointer(), peaks->size(), shift, grid_x->devicePointer(),
                 grid_y->devicePointer(), over->devicePointer(), grid_x->sizeX(), grid_x->sizeY());
    checkAndSync();

    __generateGrid<<<dimGrid, dimBlock>>>(
                 peaks->devicePointer(), peaks->size(), shift, grid_x->devicePointer(),
                 grid_y->devicePointer(), over->devicePointer(), grid_x->sizeX(), grid_x->sizeY());
    checkAndSync();
    // Con 4 Pasadas hay una desviación estándar de 0.42-0.48 en los peaks totales, siendo
imperceptible los cambios.
}
```

4.8.4. Métricas

A continuación se presentan algunas métricas de la biblioteca $Chi2HD\_Cuda$:

Métrica	Valor
Número de clases	8
Número de líneas	2.691
Número de líneas de código	1.261
Número de líneas de comentario	958
Número de funciones	138
Ratio de líneas de comentario con líneas de código	0.76

Tabla 8: Métricas generales del proyecto Chi2HD_Cuda.

A continuación se presenta el mismo diagrama presentado para $PTrack2$ ahora para $Chi2HD\_Cuda$. El tamaño de la clase representa la cantidad de líneas de código mientras que el color más fuerte representa la cantidad de métodos.

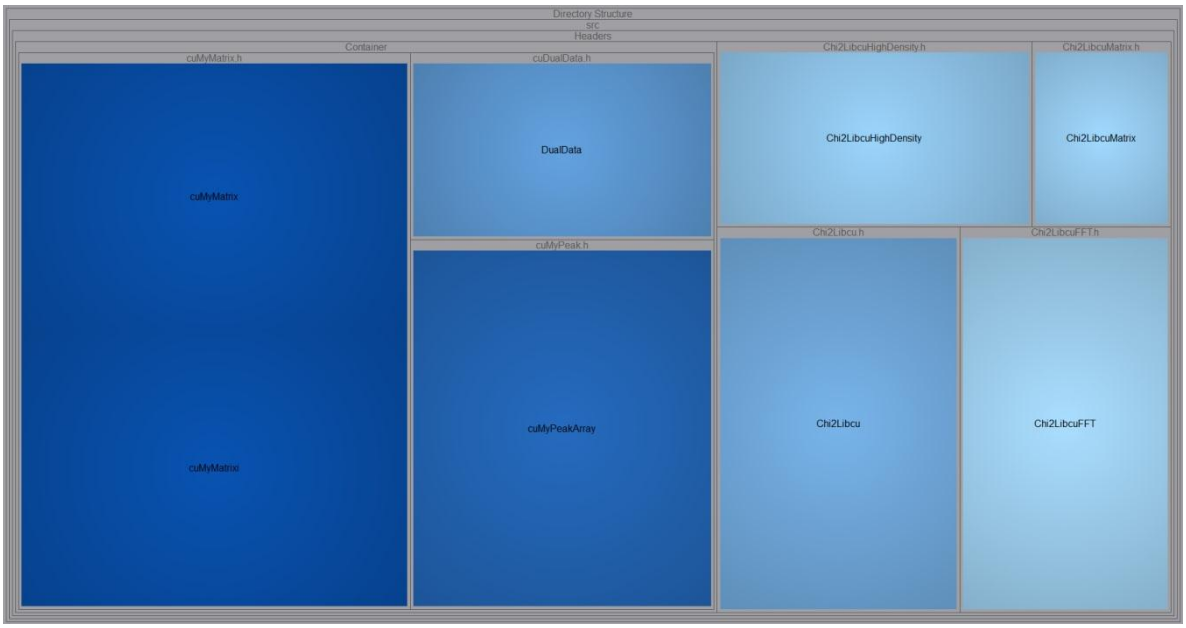


Figura 29: Mapa de métricas para Chi2HD_Cuda, líneas de código – cantidad de métodos

En la Figura 29 se aprecia que las clases cuMyMatrix, cuMyMatrixi y cuMyPeakArray son las más complejas debido al manejo de memoria manual que se tiene usando CUDA, así se tienen muchos métodos para pasar datos de memoria del dispositivo a memoria de host sin poder reutilizar estos métodos en otras clases debido a la conservación de la encapsulación para poder utilizar GCC en vez de NVCC.

A continuación se presentan algunas métricas asociadas al desarrollo orientado a objetos.

Clase	LCOM	DIT	CBO	NOC	RFC	WMC	Líneas de Código
Chi2Libcu	0	0	5	0	10	10	127
Chi2LibcuFFT	0	0	1	0	3	3	110
Chi2LibcuHighDensity	0	0	4	0	5	5	78
Chi2LibcuMatrix	0	0	2	0	4	4	33
cuMyMatrix	63	0	0	0	26	26	149
cuMyMatrixi	59	0	0	0	26	26	155
cuMyPeakArray	46	0	1	0	22	22	154
DualData	43	0	0	0	13	13	76
Promedio	26,4	0,0	1,6	0,0	13,6	13,6	110,3

Tabla 9: Métricas generales del proyecto Chi2HD_Cuda.

En general, las métricas para la biblioteca son mayores que en el proyecto PTrack2 debido al uso de CUDA y todo lo que conlleva trabajar manualmente con memoria sin poder reutilizar métodos, así mismo se tienen más líneas de código y éste se vuelve más complejo. Sin embargo, los valores de las métricas no son tan altos para el tipo de tecnología utilizada.

4.9. Elementos de ayuda al desarrollo

Todo el código de $PTrack2$ y el de la biblioteca $Chi2HD\_Cuda$ se encuentra debidamente documentado mediante páginas HTML generadas con *Doxygen* y descritas dentro de cada función y cada clase en los archivos de encabezado de todo el código fuente.

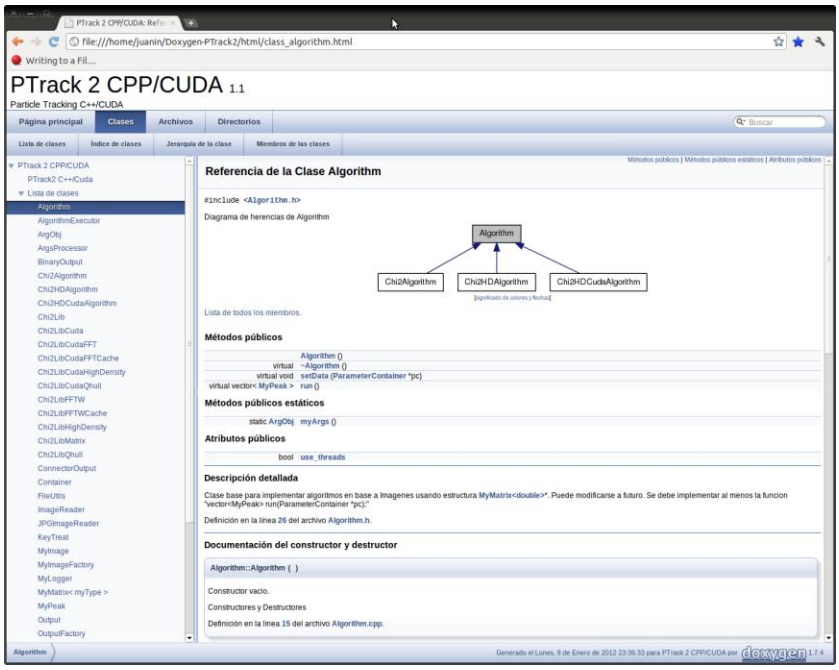


Figura 30: Documentación mediante Doxygen.

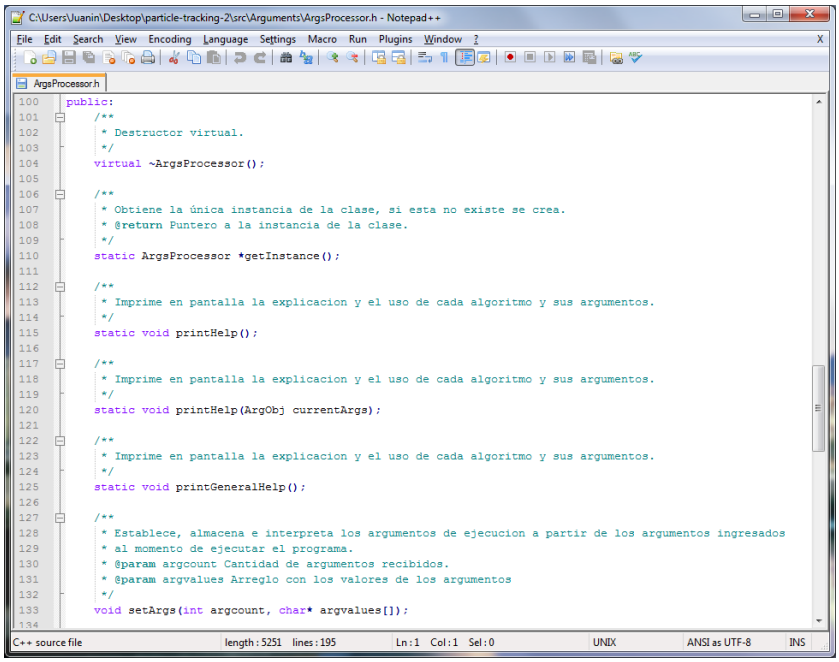


Figura 31: Documentación en archivos header.

4.10. Elementos de ayuda al usuario

El software desarrollado presenta varias ayudas al usuario, como por ejemplo la wiki ubicada en https://trac.assembla.com/particle-tracking-2/wiki, en donde se encuentra toda la documentación necesaria como guías para seguir desarrollando el software, instalación, configuración, modo de uso y errores frecuentes que pueden ocurrir producto de algún error al momento de ingresar los parámetros de ejecución.

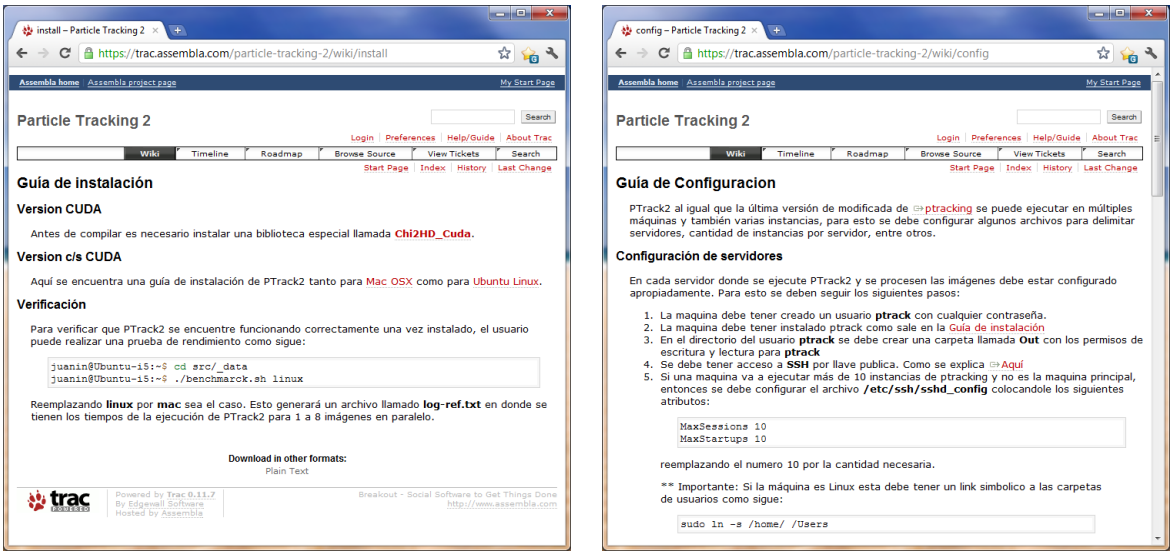


Figura 32: Páginas de instalación y configuración de la wiki.

También se encuentra documentado el modo de uso mediante *man page* para obtener ayuda acerca de los parámetros de ejecución de $PTrack2$.

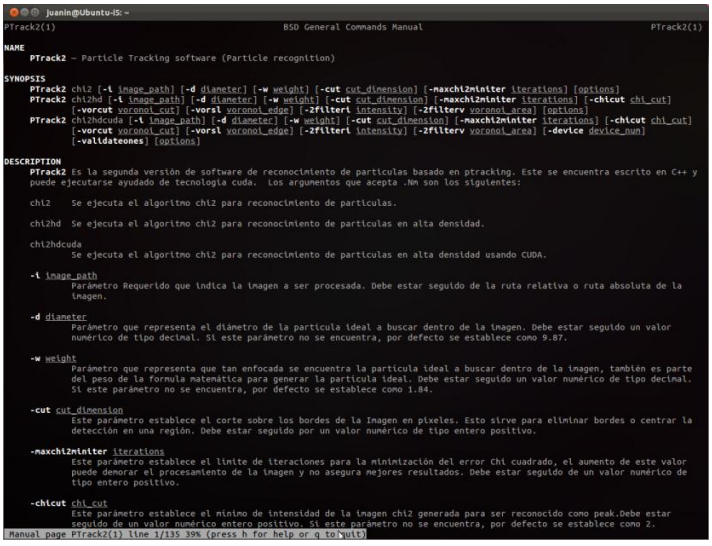


Figura 33: Documentación mediante man page.

Para evitar errores al escribir los distintos argumentos que acepta $PTrack2$ se complementó toda la ayuda anterior agregando un script de auto-completión usando *Bash Completion*, desplegando en pantalla las distintas opciones disponibles mientras se está escribiendo la ejecución del programa.

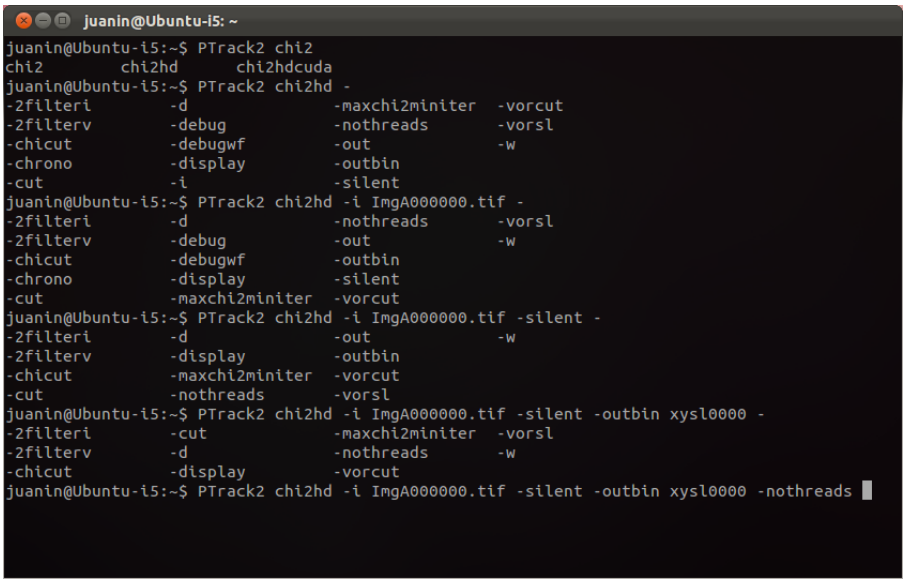


Figura 34: Auto compleción al momento de ejecutar.

A medida que se va escribiendo y se presiona la tecla “tab” aparecen las opciones disponibles, por ejemplo, al escribir *PTrack2* y presionar “tab” se escribe de inmediato *chi2* pues es el sufijo de todas las opciones disponibles en ese punto. Luego, al presionar 2 veces “tab” se despliegan las 3 opciones disponibles que en este caso son *chi2*, *chi2hd* y *chi2hdcuda*. Una vez escrito, en este caso, *chi2hd* se empiezan a desplegar las opciones asociadas a la ejecución de éste algoritmo y mientras se van escribiendo más opciones éstas ya no van a estar disponibles, como por ejemplo al escribir la opción *–silent* la cual esta dentro de un grupo excluyente el cuales es *–debug –debugwf –silent* y *–chrono* que establecen el nivel de despliegue de texto en pantalla, por lo tanto al escribir cualquiera de éstos, el grupo completo ya no va a estar disponible.

Para ayudar en la ejecución de los scripts, éstos se pueden ejecutar sin parámetro y en este caso se desplegará un mensaje de ayuda en donde se describe cada argumento necesario y su significado.

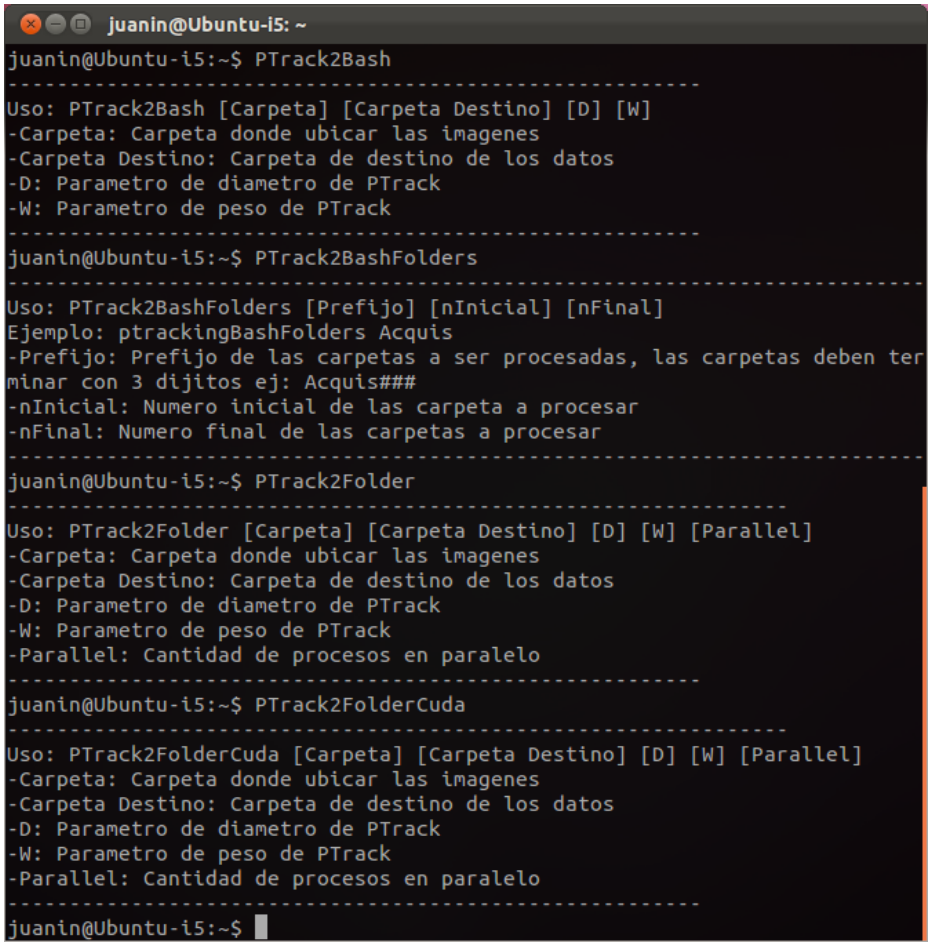


Figura 35: Ayuda de ejecución de los scripts.

Capítulo 5

Validación de la solución

Para validar la solución implementada se compararon 4 factores:

1. Tiempo de procesamiento por imagen.
2. Tiempo total de procesamiento por video.
3. Tiempo total de procesamiento de una sesión de trabajo.
4. Exactitud de los datos de salida.

Todos estos factores tienen como objetivo principal comparar el software $PTrack2$ contra el software antiguo $PTrack$. Como segundo objetivo se desea comparar el sistema distribuido ideado como solución funcionando con ambos software. Debido a que la manera antigua de obtener los datos no era estable y los tiempos de procesamiento no eran medibles, es que no se puede realizar una comparación directa con la nueva solución, sin embargo se puede tener un tiempo estimativo basándose en la experiencia del investigador con más práctica en éstos experimentos. Como último y no menor objetivo se desea verificar que los resultados obtenidos por la nueva solución sean lo más exactos posibles.

Cabe destacar que cada sesión de trabajo puede contener una cantidad variable de videos, así mismo un video puede contener una cantidad variable de imágenes y estas imágenes pueden tener distintas formaciones de partículas por lo que el tiempo en procesar cada imagen es variable y así mismo el tiempo en procesar un video y una sesión de trabajo. En este sentido se tiene que las mediciones realizadas solo son estimativas y en ningún caso finales, por lo cual éstas se deben tomar como mediciones de referencia con motivos comparativos solamente.

5.1. Plataforma de pruebas

Para las pruebas por imagen y por video se eligió como computador de pruebas el comprado a petición del alumno y descrito en la sección 4.4.1. Esto debido a que se podía ejecutar $PTrack2$ en todas sus variantes y también $PTrack$ por lo que se tiene una plataforma estándar para poder comparar. Cabe destacar que el computador posee dos discos duros, el principal es un disco de estado sólido o SSD cuyo rendimiento es superior a otros discos duros convencionales pero con menor espacio de almacenamiento, y otro convencional de 1 TeraByte. Las pruebas de rendimiento en el caso de imagen por imagen se realizaron haciendo uso del disco SSD, mientras que para las pruebas por video se utilizó el disco convencional. Para las pruebas por sesión de trabajo se utilizaron todos los computadores disponibles descritos en la sección 4.1.

5.2. Metodología

Para las mediciones de tiempo de procesamiento por imagen se seleccionó una única imagen a procesar siendo copiada y renombrada al momento de trabajar con múltiples procesos evitando un posible cuello de botella al tratar de leer la imagen por todas las instancias del software a probar, así cada instancia del software lee una imagen independiente. Se procesaron desde 1 a 8 imágenes simultáneamente ejecutando un total de 3 iteraciones. Se ejecutó $PTrack2$ usando threads, sin uso de threads, usando CUDA y $PTrack$. Se probaron desde 1 hasta 8 imágenes en paralelo debido a que ese es el valor máximo que soporta $PTrack2$ usando CUDA con el computador de pruebas, con más imágenes el programa puede dejar de funcionar por falta de memoria en dispositivo. Este límite va enlazado exclusivamente con la tarjeta de video usada para los cálculos.

Para las mediciones de tiempo de procesamiento por video se estableció el video número 20 tomado el 12 de Agosto del 2011 con una cantidad de 3.272 imágenes. La particularidad de éste video es que hay una gran cantidad de partículas formadas densamente por lo que la detección de partículas dentro de la formación densa es importante y generalmente el proceso de detección es mas lento. Cada video se procesó 3 veces usando $PTrack2$ con threads, $PTrack2$ usando CUDA y $PTrack$, procesando 8 imágenes en paralelo por cada configuración y modo de ejecución.

Para medir los tiempos en procesar una sesión de trabajo usando el sistema distribuido para $PTrack$ y $PTrack2$, se utilizaron como datos las imágenes de la sesión de trabajo tomada el día 9 de Agosto del 2011 teniendo 31 videos, cada video contiene 3.285 imágenes lo que hace un total de 101.835 imágenes a procesar. Estas imágenes se procesaron usando los 4 computadores disponibles en el laboratorio para este propósito, utilizando las siguientes configuraciones:

	Cantidad de imágenes			
Computador	**Configuración 1** *PTrack2*	**Configuración 2** *PTrack2*	**Configuración 3** *PTrack2*	**Configuración 4** *PTrack*
Granos2D	8	8	0	8
Cuasi1D	4	4	0	4
LMFE	4	4	0	4
Granos2D-i7	8 usando CUDA	8	8 usando CUDA	8

Tabla 10: Configuración para pruebas de procesamiento de sesión de trabajo

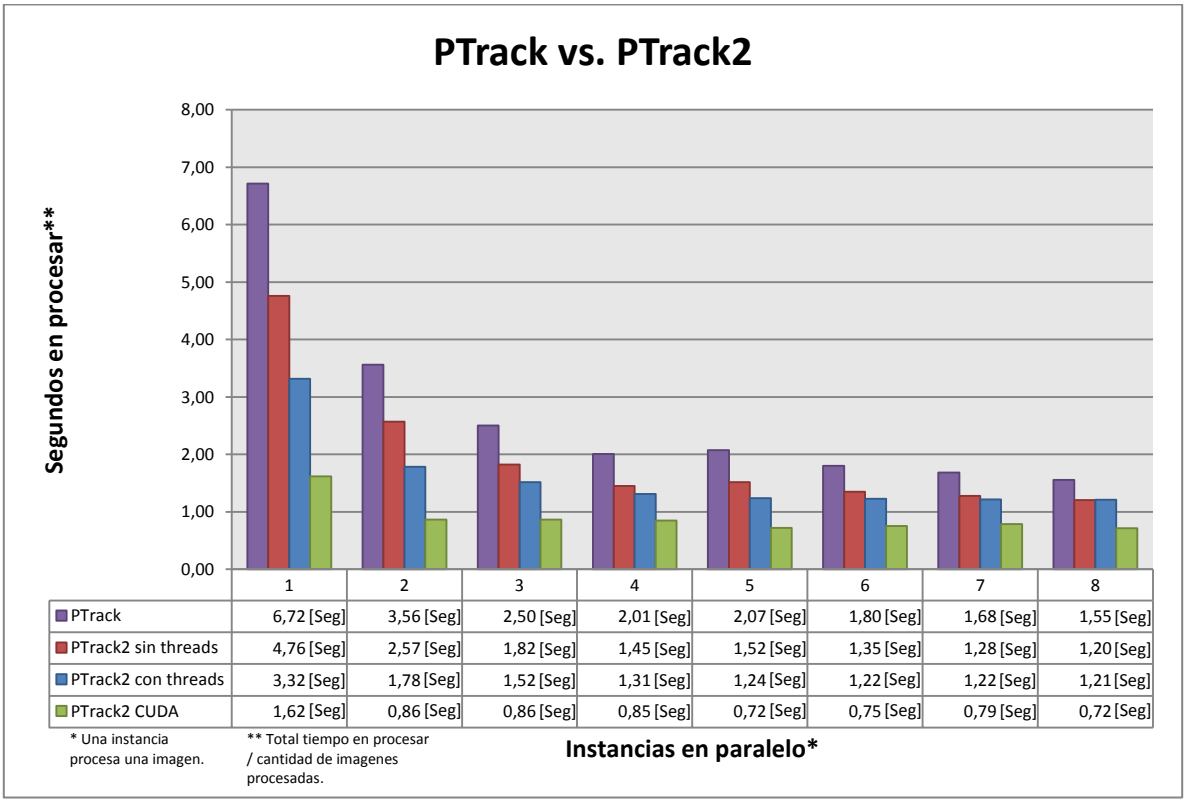
Con esta configuración es que se puede apreciar la diferencia entre ambos programas y también visualizar el efecto que produce el uso de CUDA en el procesamiento de imágenes.

5.3. Resultados

A continuación se presentan y discuten los resultados de las pruebas que se hicieron. Los datos completos y más detallados se pueden encontrar en el apéndice A.

5.3.1. Tiempo de procesamiento por imagen

Los resultados de la comparación entre $PTrack$ y $PTrack2$ fueron los siguientes:



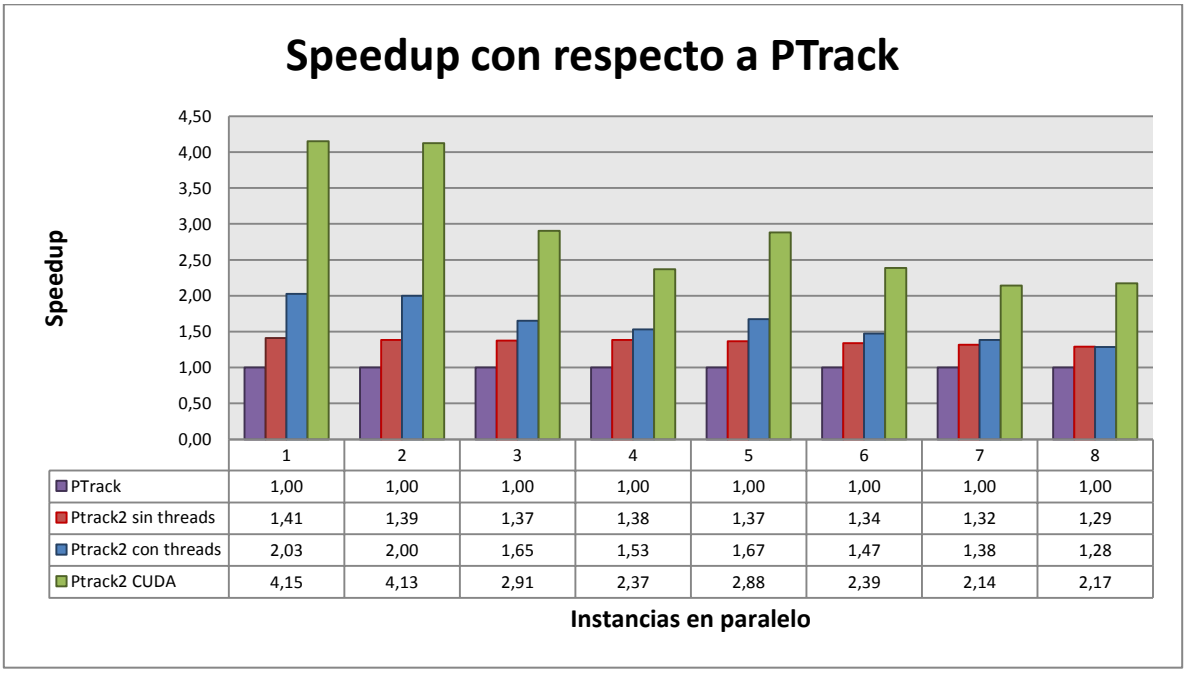
	1	2	3	4	5	6	7	8
PTrack	6,72 [Seg]	3,56 [Seg]	2,50 [Seg]	2,01 [Seg]	2,07 [Seg]	1,80 [Seg]	1,68 [Seg]	1,55 [Seg]
PTrack2 sin threads	4,76 [Seg]	2,57 [Seg]	1,82 [Seg]	1,45 [Seg]	1,52 [Seg]	1,35 [Seg]	1,28 [Seg]	1,20 [Seg]
PTrack2 con threads	3,32 [Seg]	1,78 [Seg]	1,52 [Seg]	1,31 [Seg]	1,24 [Seg]	1,22 [Seg]	1,22 [Seg]	1,21 [Seg]
PTrack2 CUDA	1,62 [Seg]	0,86 [Seg]	0,86 [Seg]	0,85 [Seg]	0,72 [Seg]	0,75 [Seg]	0,79 [Seg]	0,72 [Seg]

Figura 36: Resultados de PTrack y PTrack2 en procesar imágenes.

Con estos resultados se puede concluir que bajo todas las pruebas $PTrack2$ demostró ser más rápido que su predecesor. Usando CUDA para los cálculos es que se visualiza una mejora drástica en los tiempos de ejecución llegando a una diferencia máxima de 5 segundos. Por otra parte, se puede ver que el uso de threads por parte de $PTrack2$ ayuda significativamente a mejorar el desempeño, sin embargo a medida que se aumentan las imágenes que se procesan en paralelo se va estrechando éste aumento de rendimiento hasta llegar a las 8 imágenes en que prácticamente la diferencia entre usar y no usar threads es nula.

También se puede notar que desde procesar 1 imagen hasta procesar 4 la tendencia en los tiempos es a bajar, sin embargo al procesar 5 se encuentra una leve reducción o incluso aumento del tiempo y esto se explica principalmente a la cantidad de núcleos físicos que posee el procesador que justamente son 4, por lo que agregar una instancia más se produce una sobrecarga en el uso del procesador.

En la siguiente ilustración se muestra el speedup[20] respecto a $PTrack$:



	1	2	3	4	5	6	7	8
PTrack	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
Ptrack2 sin threads	1,41	1,39	1,37	1,38	1,37	1,34	1,32	1,29
Ptrack2 con threads	2,03	2,00	1,65	1,53	1,67	1,47	1,38	1,28
Ptrack2 CUDA	4,15	4,13	2,91	2,37	2,88	2,39	2,14	2,17

Figura 37: Speedup por imágenes con respecto a PTrack.

Aquí se puede visualizar el speedup de la ejecución de $PTrack2$ usando CUDA los cuales van desde 2,17 a 4,15 en comparación a su predecesor $PTrack$. También se puede notar los valores al ejecutar $PTrack2$ usando threads desde 1,28 a 2,03 y sin threads desde 1,29 a 1,41.

Se aprecia de forma más clara la diferencia entre usar y no usar CUDA al momento de ejecutar $PTrack2$, en donde puede significar una diferencia de speedup de un 0,88 a un 2,12.

[20] Speedup se puede definir como $S_n = \frac{T_r}{T_n}$ donde T_r es el tiempo de ejecución del software a comparar y T_n es el tiempo de ejecución de la nueva implementación del software.

5.3.2. Tiempo total de procesamiento por video

Al procesar el video seleccionado para las pruebas los resultados fueron los siguientes:

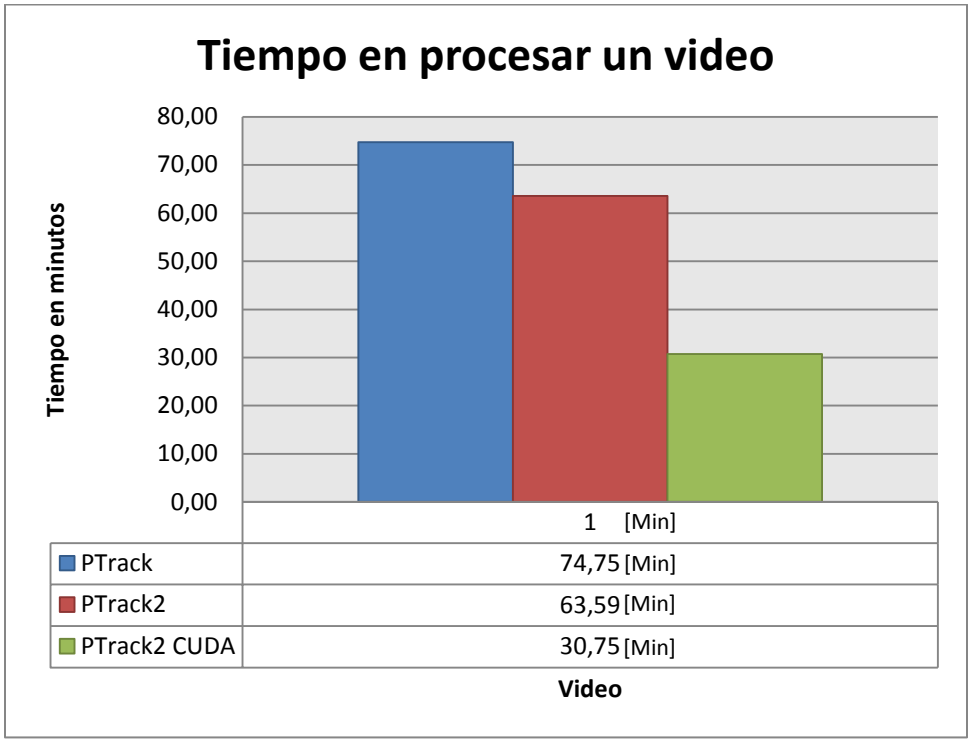


Figura 38: Resultados de procesar un video completo.

Al ver estos resultados se puede apreciar una leve diferencia entre procesar los datos usando $PTrack$ y $PTrack2$ sin usar CUDA, la cual es de unos 11 minutos en contraste con la abismante diferencia usando CUDA que es de unos 44 minutos. Tampoco es de desmerecer la disminución de 11 minutos, pues este recorte de tiempo se propaga al procesar varios videos, por lo que se pueden ahorrar horas de trabajo usando el nuevo software.

También se puede hacer un pequeño cálculo de cuánto fue el promedio de tiempo de procesamiento por imagen en un total de 3.272 imágenes se tiene:

	Tiempo total [s]	Tiempo promedio por imagen [s]
$PTrack$	4.485	1,37
$PTrack2$	3.815	1,17
$PTrack2$ CUDA	1.845	0,56

Tabla 11: Tiempos promedios de procesamiento por imagen en video.

Los resultados fueron menores a la prueba anterior pero mantuvieron una magnitud relativa similar, lo cual indica una relación constante respecto de las pruebas anteriores, teniendo así un buen punto de referencia para poder extrapolar datos y poder predecir los tiempos de ejecución para posteriores cálculos.

El tiempo promedio por imagen menor se explica debido a que en las pruebas anteriores se procesa una imagen densa la cual generalmente tiene un tiempo de procesamiento mayor, en contraste, el video posee muchas imágenes distintas por lo que los tiempos en procesar cada imagen son distintos entre si, pudiendo ser algunos menores y otros mayores.

En el siguiente gráfico se representa la diferencia porcentual comparada con $PTrack$ en el procesamiento del video.

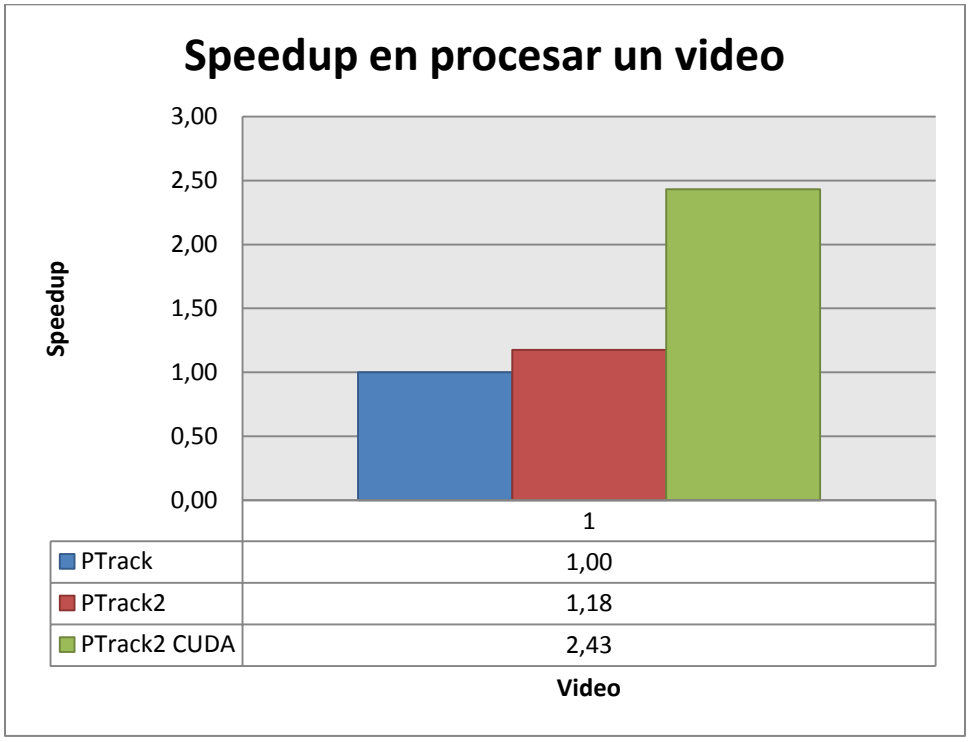


Figura 39: Speedup al procesar un video.

En éste gráfico se puede apreciar el aumento de rendimiento del 18% usando $PTrack2$ y el del 143% usando $PTrack2$ con CUDA. Estos resultados ratifican las ventajas del desarrollo usando CUDA para el procesamiento de datos y el aumento de rendimiento en general de $PTrack2$ ante su predecesor.

5.3.3. Tiempo por sesión de trabajo

Como se explicó anteriormente, los tiempos de procesamiento con la manera antigua de procesar las imágenes no eran medibles de una forma exacta por lo que se tomó como estimación un tiempo de referencia de 2 días y medio en el mejor de los casos, esto es unas 60 horas de trabajo.

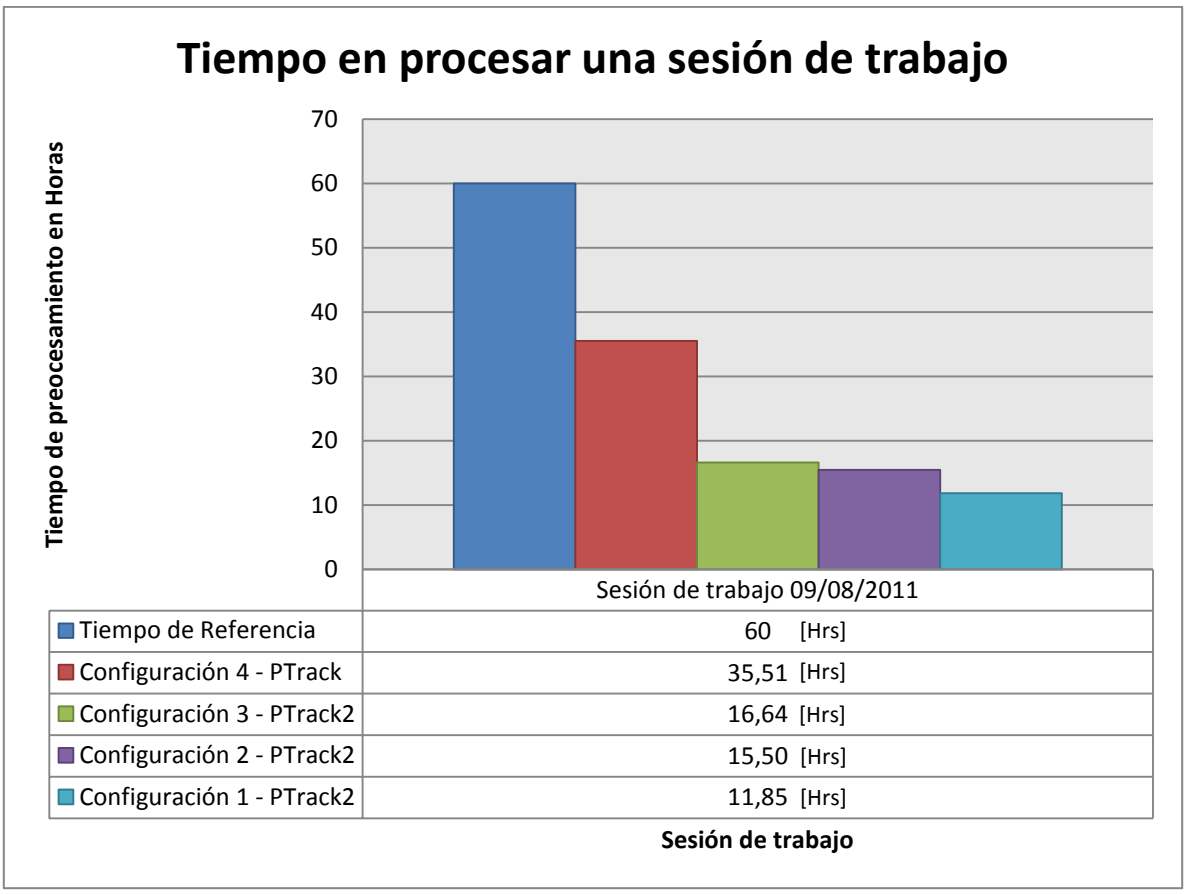


Figura 40: Resultados de procesar una sesión de trabajo.

En principio se puede apreciar la gran diferencia entre el tiempo de referencia y el tiempo usando la configuración 1, siendo ésta diferencia de alrededor de 48 horas lo cual representa una mejora extraordinaria con respecto al tiempo de referencia.

A continuación se puede apreciar una mejora significativa de unas 24 horas la configuración 4, sin embargo los problemas de detención que presenta $PTrack$ hicieron que en la ejecución de la prueba no se pudiesen procesar unas 1.907 imágenes lo cual produce en promedio una pérdida de alrededor de 46 imágenes por video, en consecuencia le da trabajo al investigador para rellenar los resultados faltantes de estas imágenes con el resultado inmediatamente anterior y una mayor pérdida de tiempo. Esto no pasa en ningún caso usando $PTrack2$ pues este procesa todas las imágenes sin ninguna falla.

Aún con todos los problemas presentados usando $PTrack$ existe una diferencia de unas 23 horas con respecto a la ejecución de la configuración 4 contra la configuración 1, unas 20 horas contra la configuración 2 y unas 18 horas contra la configuración 3.

Con estos datos se vislumbra una gran influencia de CUDA en los tiempos de procesamiento alcanzando resultados un poco más lentos al usar solo un computador usando CUDA que 4 computadores usando solo CPU. Esto lleva a pensar que los resultados podrían mejorar aún más usando otra tarjeta de video igual a la usada para alcanzar un rendimiento mucho mayor que cualquiera de los resultados obtenidos evitando así tener que coordinar el uso de 4 computadores y remplazar estos por el uso de solo uno, exclusivamente para este proceso.

Calculando el tiempo promedio de procesamiento por imagen de un total de 101.835 imágenes se tienen los siguientes resultados:

	Tiempo total [s]	Tiempo por imagen [s]
Tiempo de referencia	216.000	2,12
Configuración 4 - PTrack	127.820	1,26
Configuración 3 – PTrack2	59.914	0.59
Configuración 2 – PTrack2	55.812	0,55
Configuración 1 – PTrack2	42.647	0,42

Tabla 12: Tiempos de procesamiento por imagen en sesión de trabajo.

En esta tabla se puede apreciar que el uso de más computadores para el procesamiento comparado con solo uno usando CUDA influye en la disminución de unos 200 milisegundos por imagen, lo cual finalmente aporta en la disminución en varias horas de procesamiento, pero no es una magnitud significativa en relación a la cantidad de recursos comprometidos siendo una tarjeta de video contra 3 computadores con doble procesador. Así mismo el procesamiento usando $PTrack$ sigue teniendo más menos la misma magnitud relativa que el procesamiento por video mostrado en el punto anterior.

A continuación se presentan los resultados porcentuales de los tiempos obtenidos:

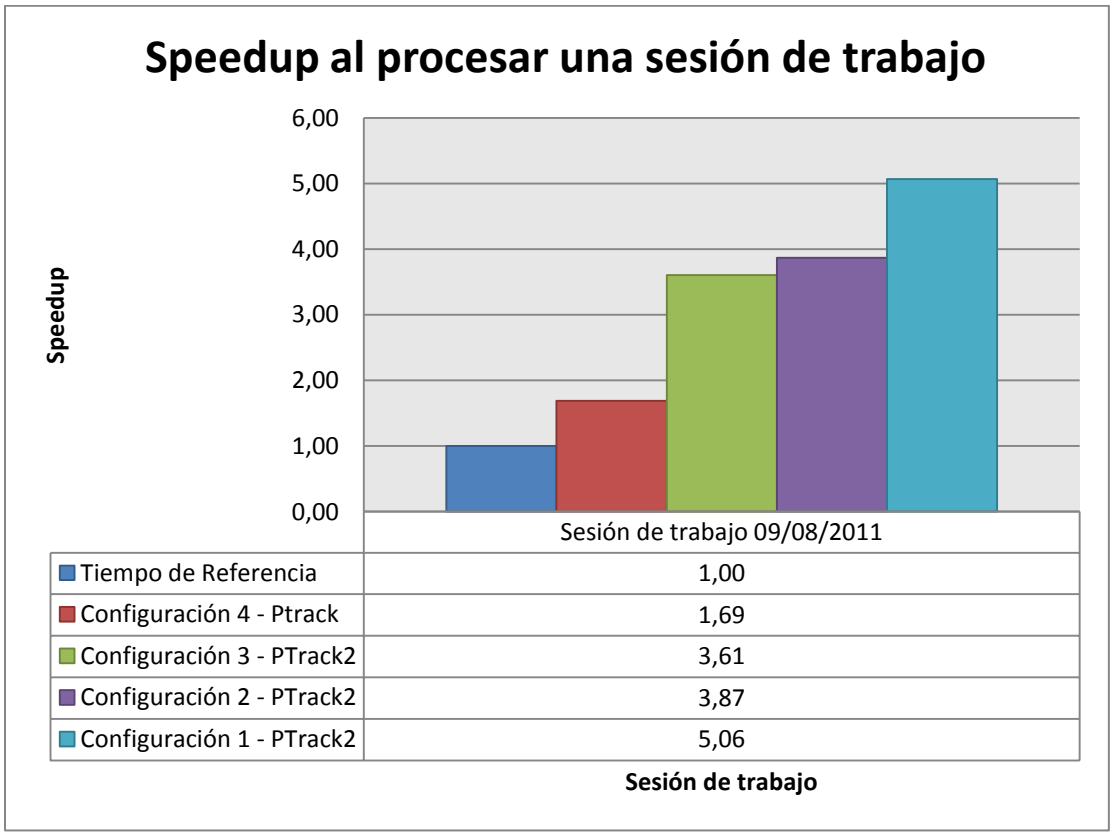


Figura 41: Resultados porcentuales al procesar una sesión de trabajo.

Con este gráfico se puede apreciar el speedup logrado el cual va desde un 1,69 hasta un 5,06 pudiendo llegar, en teoría, aún más lejos usando más dispositivos CUDA como se mencionó anteriormente. También se puede ver la diferencia de solo un 0,26 respecto a la ejecución de $PTrack2$ usando solo CUDA en un computador (Configuración 3) y $PTrack2$ usando 4 computadores utilizando solo CPU (Configuración 2), lo cual evidencia aún más el poder de esta tecnología.

Estos resultados reflejan muy bien el cumplimiento de los objetivos planteados para reducir el tiempo de procesamiento de las imágenes, pues ésta es una prueba real de imágenes capturadas y no solamente imágenes de pruebas.

5.4. Exactitud

Para verificar que los datos fuesen lo más exactos posibles se compararon los archivos de salida de varias imágenes procesadas con anterioridad por $PTrack$ y los datos de salida de $PTrack2$ verificándose que estos eran iguales, sin embargo para los resultados de $PTrack2$ usando CUDA los resultados tenían, en algunas ocasiones, leves variaciones en la precisión de la posición de las partículas y en la cantidad de partículas, por lo que se decidió verificar que los datos fuesen aceptables siendo éstos procesados por el investigador Gustavo Castillo, quien confirmó que los datos en su completitud eran estadísticamente iguales resultando aceptables para el trabajo.

La aceptación del uso de los resultados de $PTrack2$ usando CUDA se validó haciendo cálculos de videos completos y obteniendo el Factor de Estructura Estático[21] y específicamente el FSDP para cada uno de ellos, usando los datos de salida de $PTrack$ y $PTrack2$ usando CUDA. Los datos fueron graficados y comparados teniendo una curva representando cada programa usado, así se visualizaron dos curvas que se sobreponen entre si, con excepción de los gráficos con datos de videos en que el sistema se encuentra muy fluidizado, vale decir, no poseen formaciones densas de partículas, sin embargo la diferencia en la curva no presentó mayores variaciones considerables como se muestra a continuación:

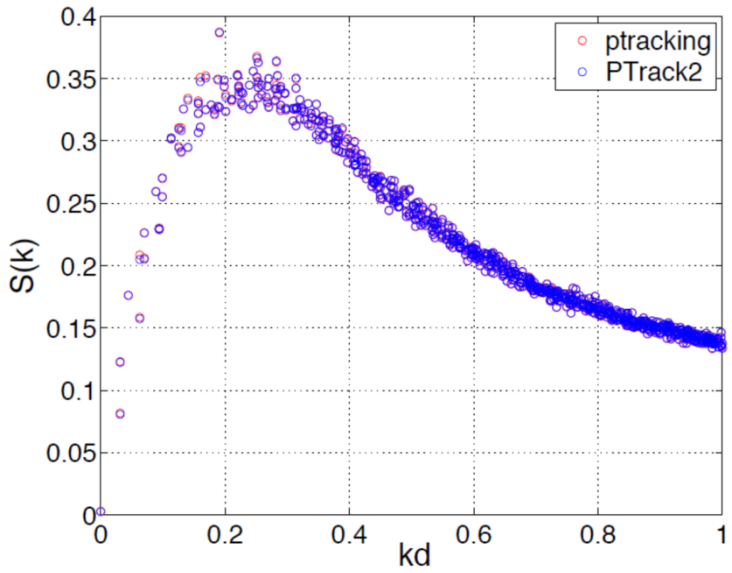


Figura 42: FSDP del Factor de Estructura Estático usando datos de salida de PTrack y PTrack2 usando CUDA.

En el gráfico se puede apreciar la curva de color azul representando los datos obtenidos de $PTrack2$ usando CUDA y en color rojo los datos de $PTrack$. En el eje vertical se encuentran los valores del FSDP del Factor de Estructura Estático según el valor k y en el eje horizontal la variación del valor k.

[21] Ver Punto 2.4. del Capítulo de Antecedentes.

Las variaciones de los resultados de $PTrack2$ usando CUDA se explican principalmente al uso de punto flotante en las estructuras de datos usadas y en el algoritmo implementado para realizar Transformadas de Fourier, el cual difiere entre las bibliotecas cuFFT y FFTW. Lo primero con el fin de conseguir un mayor rendimiento y compatibilidad con más tarjetas de video con tecnología CUDA ya que para procesar números con doble precisión es necesario tener un dispositivo con capacidades *Compute Capability 2.x*, vale decir, tarjetas de video de última generación las cuales se cumplen en la GTX590 pero no en el dispositivo que posee el alumno actualmente.

Capítulo 6

Conclusiones

El alumno logró remplazar con éxito el antiguo software $PTrack$ por un sistema distribuido de cómputo usando como núcleo de procesamiento el nuevo software $PTrack2$ implementado en C++ utilizando patrones de diseño, threads y tecnología CUDA para acelerar los cálculos.

El impacto tanto de la solución completa como de las iteraciones realizadas reflejó un aporte significativo en la reducción de tiempos de procesamiento de datos, también significó un aprendizaje por parte de los investigadores acerca de cómo trabajar con el nuevo sistema, siendo ésta última rápida y de fácil comprensión gracias a la documentación de apoyo y a las ayudas que presenta cada script y software desarrollado.

El tiempo de procesamiento de los datos de una sesión de trabajo logró reducirse a alrededor de unas 11 horas lo que significa un aumento en rendimiento de alrededor de un 400% lo que da la posibilidad de dejar realizando cálculos en la noche para en la mañana tener los resultados listos, sin problemas de detención o errores que detengan éste procesamiento teniendo así un sistema autónomo. Gracias a su diseño orientado a objetos y uso de patrones de diseño el software puede extenderse e implementar nuevos algoritmos modificando unas pocas clases teniendo un impacto mínimo y facilitando la implementación de estos algoritmos mediante pasos de parámetros simples y fáciles de entender.

El sistema distribuido contempló la configuración mediante un archivo por lo cual agregar un nuevo servidor se realiza modificando el archivo y configurando de buena manera el computador objetivo siguiendo los pasos necesarios descritos en la documentación, por lo que escalar el sistema se puede realizar con relativa facilidad. En cualquier momento en que llegase a fallar un servidor de procesamiento éste es eliminado y los datos son redistribuidos a los demás servidores, lo cual lo hace tolerante a fallas.

Los resultados obtenidos ratifican la validez de la solución teniendo un fuerte apoyo de la tecnología CUDA, sin la cual los resultados hubiesen sido buenos pero no excelentes. Es más, usando solo tecnología CUDA en un solo computador con dos dispositivos CUDA equivale al trabajo del mismo software usando 4 computadores usando CPU por lo que estos resultados pueden ser mejorados agregando otros 2 dispositivos CUDA y lograr un máximo en rendimiento usando solo un computador, evitando coordinación del uso de los equipos teniendo un solo equipo exclusivo para este tipo de cálculos.

Al ver que los resultados se ven fuertemente sustentados en la tecnología CUDA se pudo haber pensado en generar el software inmediatamente usando ésta tecnología, ahorrando trabajo en la generación de un sistema distribuido y evitando la escritura de código que no se utilizará, pero esto no es tan factible pues el sistema distribuido ayudó desde un inicio a la reducción de los tiempos de ejecución aumentando la producción de las investigaciones, además el sistema ayuda a dar flexibilidad a la solución teniendo más opciones de dónde procesar a la hora de ejecutar el software y ayudando a distribuir los datos que se van a procesar evitando tener que configurar el computador con CUDA para aceptar la cámara de video, instalar el software para capturar éstas imágenes y evitando también el cambio de paradigma de los investigadores al tener que utilizar un nuevo sistema operativo desconocido, pues este computador con CUDA utiliza Linux en vez de Mac OSX.

Si bien los resultados son muy buenos, esto representa solamente una parte de todo el proceso desde la captura de información hasta la generación de resultados a partir de una serie de procesamiento de datos, como son la detección de partículas. Existe una serie de pasos que toman horas en procesar usando MATLAB y programas pequeños escritos en C, por lo que se recomienda aumentar el rendimiento de estos pasos trasladando éstas funciones de MATLAB a implementaciones en lenguajes de programación más rápidos y así mismo los pequeños programas escritos en C paralelizarlos y optimizarlos. En este sentido se puede extender $PTrack2$ para poder adaptarse a estos procesos o bien complementar MATLAB con el uso de CUDA para acelerar un poco estos cálculos.

Bibliografía

[1] PROFESSOR MARK D. SHATTUCK. *Particle Tracking Algorithm* [En Línea] <http://gibbs.engr.ccny.cuny.edu/technical/Tracking>. [Consulta: 6 Junio 2011]

[2] MAURICIO CERDA. *Implementación en C de algoritmo de reconocimiento de partículas del Professor Mark D. Shattuck* [En Línea] <http://www.dfi.uchile.cl/~nmujica/ptracking_0.9beta> [Consulta: 23 Marzo 2011]

[3] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995

[4] SCOTT WAITUKAITIS. *High Density Peaks variant for ptracking* [En Línea] <http://www.dcc.uchile.cl/~jsilva/program_summary.pdf> [Consulta: 16 Septiembre 2011]

[5] HUGH MERZ. *CUFFT 1.1 / 2.0 vs FFTW 3.1.2 (x86_64) vs FFTW 3.2 (Cell) comparison* [En Línea] Desktop Engineering , 1 de Septiembre 2010. <http://www.sharcnet.ca/~merz/CUDA_benchFFT/> [Consulta: 6 Mayo 2011]

[6] PETER VARHOL*, GPU vs. CPU Computing* [En Línea] Desktop Engineering en Línea <http://www.deskeng.com/articles/aaayet.htm> [Consulta: 10 Mayo 2011]

[7] VICTOR W LEE, CHANGKYU KIM, JATIN CHHUGANI, MICHAEL DEISHER, DAEHYUN KIM, ANTHONY D. NGUYEN, NADATHUR SATISH, MIKHAIL SMELYANSKIY, SRINIVAS CHENNUPATY, PER HAMMARLUND, RONAK SINGHAL y PRADEEP DUBEY. Throughput Computing Lab, Intel Corporation and Intel Architecture Group, Intel Corporation. *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU.* ISCA'10, ACM New York. (Doi: 10.1145/1815961.1816021), 2010

[8] NVIDIA CUDA DEVELOPER ZONE. *CUDA Toolkit 3.2 math Library Performance* [En Línea] <http://developer.download.nvidia.com/compute/cuda/3_2/docs/CUDA_3.2_Math_Libraries_Performance.pdf> [Consulta: 18 Mayo 2011]

[9] NVIDIA CUDA DEVELOPER ZONE. *CUDA Toolkit 4.0 math Performance Report* [En Línea] <http://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDA_4_0_Math_Libraries_Performance_6_14.pdf> [Consulta: 5 Diciembre 2011]

[10] JUAN SILVA, *CUDA Overview and Programming model* [En Línea] <http://www.dcc.uchile.cl/~jsilva/cc68w-es.pdf> [Consulta: 26 Junio 2011]

[11] JAVIER BUSTOS JIMÉNES. *Course notes. CC51S: Distributed Systems* [En Línea] <https://www.u-cursos.cl/ingenieria/2010/2/CC51S/1/material_docente> [Consulta: 21 Septiembre 2010]

[12] ROB FARBER. *CUDA, Supercomputing for the masses* [En Línea] Dr. Dobb's. 15 de Abril 2008. <http://www.drdobbs.com/high-performance-computing/207200659> [Consulta: 21 Junio 2010]

[13] NVIDIA CUDA DEVELOPER ZONE. *Nvidia CUDA C Programming guide* [En Línea] <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf> [Consulta: 5 Diciembre 2011]

[14] NVIDIA CUDA DEVELOPER ZONE. *Nvidia cuFFT library* [En Línea] <http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUFFT_Library.pdf> [Consulta: 5 Diciembre 2011]

[15] NVIDIA CUDA DEVELOPER ZONE. *Nvidia CUDA Library Documentation* [En Línea] **<**http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/html/> [Consulta: 5 Diciembre 2011]

[16] SOURCE MAKING. *Design Patterns* [En Línea] <http://sourcemaking.com/design_patterns> [Consulta: 16 Octubre 2011]

[17] DIMITRI VAN HEESCH. *Doxygen Documentation* [En Línea] <http://www.stack.nl/~dimitri/doxygen/> [Consulta: 3 Enero 2012]

[18] GCC, GNU COMPILER COLLECTION. *GCC Optimization Flags* [En Línea] <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options> [Consulta: 26 Octubre 2011]

[19] LOG4CPP – *Log4cpp Documentation* [En Línea] <http://log4cpp.sourceforge.net/api/index.html> [Consulta: 1 Octubre 2011]

[20] WIKIPEDIA – *GeForce 500 Series* [En Línea] <http://en.wikipedia.org/wiki/GeForce_500_Series> [Consulta: 14 Julio 2011]

[21] GUSTAVO CASTILLO B. *Fluctuaciones de Densidad en la Vecindad de una Transición de fase Solido-Liquido Granular.* Proyecto de Tesis Doctoral. **Abril, 2012**

Apéndice

Apéndice A

A continuación se presentan los resultados completos de las pruebas realizadas para comparar $PTrack$, $PTrack2$ y $PTrack2$ usando CUDA.

Tiempos de procesamiento por imagen

Todos los tiempos están medidos en segundos.

PTrack

Iteración 1

Cantidad de imágenes	1	2	3	4	5	6	7	8
	6,70	7,11	7,43	7,77	7,85	7,93	8,06	12,27
		7,13	7,47	7,78	7,93	8,49	10,82	12,30
			7,48	8,33	8,44	10,20	11,01	12,35
				8,36	9,83	10,63	11,54	12,37
					10,35	10,66	11,68	12,38
						10,73	11,69	12,39
							11,82	12,40
								12,41
Promedio	6,70	7,12	7,46	8,06	8,88	9,77	10,95	12,36
Desviación Estándar	0,00	0,01	0,03	0,33	1,14	1,24	1,33	0,05
Promedio por imagen	6,70	3,56	2,49	2,09	2,07	1,79	1,69	1,55
Desviación Estándar por imagen	0,00	0,01	0,01	0,08	0,23	0,21	0,19	0,01

Iteración 2

Cantidad de imágenes	1	2	3	4	5	6	7	8
	6,73	7,11	7,49	7,83	7,83	8,03	8,04	12,31
		7,11	7,51	7,83	7,85	9,13	10,82	12,33
			7,51	7,85	7,87	9,50	10,92	12,35
				7,85	10,32	10,69	11,64	12,36
					10,34	10,78	11,70	12,39
						10,97	11,71	12,39
							11,78	12,46
								12,46
Promedio	6,73	7,11	7,50	7,84	8,84	9,85	10,94	12,38
Desviación Estándar	0,00	0,00	0,01	0,01	1,36	1,16	1,34	0,05
Promedio por imagen	6,73	3,55	2,50	1,96	2,07	1,83	1,68	1,56
Desviación Estándar por imagen	0,00	0,00	0,00	0,00	0,27	0,19	0,19	0,01

Iteración 3

Cantidad de imágenes	1	2	3	4	5	6	7	8
	6,72	7,11	7,50	7,81	7,84	7,95	8,00	12,31
		7,14	7,52	7,85	7,85	7,95	10,77	12,34
			7,52	7,85	7,88	10,59	10,94	12,35
				7,87	10,38	10,60	11,39	12,37
					10,40	10,74	11,67	12,39
						10,74	11,73	12,41
							11,74	12,41
								12,42
Promedio	6,72	7,12	7,52	7,85	8,87	9,76	10,89	12,38
Desviación Estándar	0,00	0,02	0,01	0,02	1,39	1,41	1,33	0,04
Promedio por imagen	6,72	3,57	2,51	1,97	2,08	1,79	1,68	1,55
Desviación Estándar por imagen	0,00	0,01	0,00	0,01	0,28	0,23	0,19	0,00

Promedio

Cantidad de imágenes	1	2	3	4	5	6	7	8
	6,72	7,11	7,47	7,80	7,84	7,97	8,03	12,30
		7,12	7,50	7,82	7,87	8,52	10,80	12,32
			7,51	8,01	8,06	10,10	10,96	12,35
				8,03	10,18	10,64	11,53	12,37
					10,36	10,73	11,68	12,38
						10,81	11,71	12,40
							11,78	12,42
								12,43
Promedio	6,72	7,12	7,49	7,92	8,86	9,79	10,93	12,37
Desviación Estándar	0,00	0,01	0,02	0,12	1,30	1,27	1,33	0,05
Promedio por imagen	6,72	3,56	2,50	2,01	2,07	1,80	1,68	1,55
Desviación Estándar por imagen	0,00	0,01	0,01	0,03	0,26	0,21	0,19	0,01

PTrack2 sin threads

Iteración 1

Cantidad de imágenes	1	2	3	4	5	6	7	8
	4,82	5,18	5,22	5,79	5,95	6,24	6,80	9,62
		5,19	5,45	5,80	5,97	6,26	8,37	9,64
			5,46	5,81	6,09	7,99	8,37	9,65
				5,81	7,49	8,03	8,71	9,65
					7,58	8,04	8,72	9,68
						8,12	8,93	9,68
							8,94	9,68
								9,70
Promedio	4,82	5,19	5,38	5,80	6,62	7,44	8,40	9,66
Desviación Estándar	0,00	0,01	0,14	0,01	0,84	0,93	0,75	0,03
Promedio por imagen	4,82	2,60	1,82	1,45	1,52	1,35	1,28	1,21
Desviación Estándar por imagen	0,00	0,00	0,05	0,00	0,17	0,15	0,11	0,00

Iteración 2

Cantidad de imágenes	1	2	3	4	5	6	7	8
	4,76	4,99	5,33	5,73	5,93	6,28	6,52	9,57
		5,01	5,41	5,77	5,94	6,35	8,37	9,57
			5,44	5,78	6,06	7,86	8,40	9,58
				5,80	7,61	7,91	8,86	9,58
					7,62	8,05	8,90	9,59
						8,08	8,92	9,60
							9,00	9,60
								9,63
Promedio	4,76	5,00	5,39	5,77	6,63	7,42	8,42	9,59
Desviación Estándar	0,00	0,02	0,06	0,03	0,90	0,86	0,88	0,02
Promedio por imagen	4,76	2,51	1,81	1,45	1,52	1,35	1,29	1,20
Desviación Estándar por imagen	0,00	0,01	0,02	0,01	0,18	0,14	0,13	0,00

Iteración 3

Cantidad de imágenes	1	2	3	4	5	6	7	8
	4,69	5,20	5,46	5,66	5,94	6,24	6,38	9,34
		5,21	5,47	5,74	5,97	6,24	8,21	9,44
			5,48	5,75	6,05	7,94	8,34	9,51
				5,79	7,54	7,95	8,83	9,53
					7,57	8,02	8,83	9,54
						8,03	8,85	9,56
							8,85	9,56
								9,57
Promedio	4,69	5,21	5,47	5,74	6,62	7,40	8,33	9,51
Desviación Estándar	0,00	0,01	0,01	0,06	0,86	0,90	0,90	0,08
Promedio por imagen	4,69	2,61	1,83	1,45	1,51	1,34	1,26	1,20
Desviación Estándar por imagen	0,00	0,00	0,00	0,01	0,17	0,15	0,13	0,01

Promedio

Cantidad de imágenes	1	2	3	4	5	6	7	8
	4,76	5,12	5,34	5,72	5,94	6,25	6,57	9,51
		5,14	5,45	5,77	5,96	6,28	8,31	9,55
			5,46	5,78	6,07	7,93	8,37	9,58
				5,80	7,55	7,96	8,80	9,59
					7,59	8,04	8,82	9,60
						8,08	8,90	9,61
							8,93	9,61
								9,63
Promedio	4,76	5,13	5,41	5,77	6,62	7,42	8,39	9,58
Desviación Estándar	0,00	0,01	0,07	0,03	0,87	0,90	0,84	0,04
Promedio por imagen	4,76	2,57	1,82	1,45	1,52	1,35	1,28	1,20
Desviación Estándar por imagen	0,00	0,01	0,02	0,01	0,17	0,15	0,12	0,01

PTrack2 con threads

Iteración 1

Cantidad de imágenes	1	2	3	4	5	6	7	8
	3,26	3,50	4,13	5,05	5,94	7,26	8,37	9,38
		3,54	4,50	5,20	6,02	7,27	8,39	9,58
			4,51	5,21	6,10	7,32	8,43	9,63
				5,22	6,12	7,33	8,45	9,64
					6,17	7,33	8,46	9,64
						7,36	8,49	9,67
							8,50	9,69
								9,72
Promedio	3,26	3,52	4,38	5,17	6,07	7,31	8,44	9,62
Desviación Estándar	0,00	0,03	0,22	0,08	0,09	0,04	0,05	0,11
Promedio por imagen	3,26	1,77	1,50	1,31	1,23	1,23	1,21	1,22
Desviación Estándar por imagen	0,00	0,01	0,07	0,02	0,02	0,01	0,01	0,01

Iteración 2

Cantidad de imágenes	1	2	3	4	5	6	7	8
	3,30	3,46	4,16	5,20	6,08	7,13	8,22	9,44
		3,64	4,58	5,22	6,12	7,25	8,33	9,44
			4,59	5,23	6,18	7,27	8,37	9,53
				5,25	6,18	7,27	8,40	9,58
					6,22	7,30	8,43	9,61
						7,33	8,45	9,61
							8,50	9,62
								9,63
Promedio	3,30	3,55	4,44	5,22	6,16	7,26	8,39	9,56
Desviación Estándar	0,00	0,13	0,25	0,02	0,06	0,07	0,09	0,08
Promedio por imagen	3,30	1,82	1,53	1,31	1,24	1,22	1,21	1,20
Desviación Estándar por imagen	0,00	0,07	0,08	0,00	0,01	0,01	0,01	0,01

Iteración 3

Cantidad de imágenes	1	2	3	4	5	6	7	8
	3,39	3,41	4,26	5,14	5,94	7,14	8,32	9,46
		3,52	4,53	5,19	5,99	7,22	8,37	9,55
			4,54	5,22	6,03	7,27	8,41	9,57
				5,25	6,12	7,31	8,43	9,59
					6,18	7,31	8,46	9,63
						7,34	8,49	9,64
							8,54	9,66
								9,69
Promedio	3,39	3,46	4,45	5,20	6,05	7,27	8,43	9,60
Desviación Estándar	0,00	0,08	0,16	0,05	0,10	0,07	0,08	0,07
Promedio por imagen	3,39	1,76	1,51	1,31	1,24	1,22	1,22	1,21
Desviación Estándar por imagen	0,00	0,04	0,05	0,01	0,02	0,01	0,01	0,01

Promedio

Cantidad de imágenes	1	2	3	4	5	6	7	8
	3,32	3,46	4,18	5,13	5,99	7,18	8,30	9,43
		3,57	4,54	5,20	6,04	7,25	8,36	9,52
			4,55	5,22	6,10	7,29	8,40	9,57
				5,24	6,14	7,30	8,43	9,60
					6,19	7,31	8,45	9,62
						7,34	8,48	9,64
							8,51	9,65
								9,68
Promedio	3,32	3,51	4,42	5,20	6,09	7,28	8,42	9,59
Desviación Estándar	0,00	0,08	0,21	0,05	0,08	0,06	0,07	0,09
Promedio por imagen	3,32	1,78	1,52	1,31	1,24	1,22	1,22	1,21
Desviación Estándar por imagen	0,00	0,04	0,07	0,01	0,02	0,01	0,01	0,01

PTrack2 usando CUDA

Iteración 1

Cantidad de imágenes	1	2	3	4	5	6	7	8
	1,63	1,66	1,68	1,99	2,61	2,81	2,95	3,74
		1,72	2,73	3,17	2,76	2,97	3,30	4,04
			2,76	3,29	2,97	3,08	4,61	4,16
				3,38	3,32	4,03	5,15	4,35
					3,58	4,33	5,33	5,28
						4,48	5,48	5,48
							5,53	5,65
								5,80
Promedio	1,63	1,69	2,39	2,96	3,05	3,62	4,62	4,81
Desviación Estándar	0,00	0,04	0,61	0,65	0,40	0,75	1,07	0,82
Promedio por imagen	1,63	0,86	0,92	0,85	0,72	0,75	0,79	0,72
Desviación Estándar por imagen	0,00	0,02	0,20	0,16	0,08	0,12	0,15	0,10

Iteración 2

Cantidad de imágenes	1	2	3	4	5	6	7	8
	1,61	1,71	1,92	2,02	2,94	2,78	3,15	4,12
		1,74	2,12	2,94	2,98	2,97	3,32	4,36
			2,44	3,33	3,17	3,54	4,55	4,39
				3,42	3,47	4,06	4,74	4,60
					3,63	4,48	5,29	5,53
						4,53	5,44	5,59
							5,57	5,66
								5,68
Promedio	1,61	1,73	2,16	2,93	3,24	3,73	4,58	4,99
Desviación Estándar	0,00	0,02	0,26	0,64	0,30	0,75	0,99	0,68
Promedio por imagen	1,61	0,87	0,81	0,85	0,73	0,76	0,80	0,71
Desviación Estándar por imagen	0,00	0,01	0,09	0,16	0,06	0,13	0,14	0,08

Iteración 3

Cantidad de imágenes	1	2	3	4	5	6	7	8
	1,61	1,66	1,97	2,02	2,71	2,70	2,98	3,80
		1,72	2,18	2,91	2,88	3,15	3,33	4,14
			2,56	3,28	2,94	4,01	3,44	4,28
				3,36	3,41	4,19	4,62	4,73
					3,56	4,37	5,08	5,37
						4,57	5,30	5,52
							5,39	5,65
								5,69
Promedio	1,61	1,69	2,24	2,89	3,10	3,83	4,30	4,90
Desviación Estándar	0,00	0,05	0,30	0,61	0,37	0,74	1,02	0,76
Promedio por imagen	1,61	0,86	0,85	0,84	0,71	0,76	0,77	0,71
Desviación Estándar por imagen	0,00	0,02	0,10	0,15	0,07	0,12	0,15	0,09

Promedio

Cantidad de imágenes	1	2	3	4	5	6	7	8
	1,62	1,68	1,86	2,01	2,75	2,77	3,03	3,88
		1,73	2,34	3,00	2,87	3,03	3,32	4,18
			2,58	3,30	3,03	3,54	4,20	4,28
				3,39	3,40	4,09	4,83	4,56
					3,59	4,39	5,23	5,39
						4,53	5,41	5,53
							5,50	5,65
								5,72
Promedio	1,62	1,70	2,26	2,93	3,13	3,72	4,50	4,90
Desviación Estándar	0,00	0,04	0,39	0,64	0,36	0,75	1,03	0,75
Promedio por imagen	1,62	0,86	0,86	0,85	0,72	0,75	0,79	0,72
Desviación Estándar por imagen	0,00	0,02	0,13	0,16	0,07	0,12	0,15	0,09

Tiempos de procesamiento por video

Todos los tiempos están medidos en minutos.

Iteración	$PTrack$ [m]	$PTrack2$ [m]	$PTrack2$ CUDA [m]
1	74,63	63,62	30,87
2	74,83	63,55	30,77
3	74,78	63,60	30,62
Promedio	74,75	63,59	30,75
Desviación Estándar	0,10	0,03	0,13
Promedio por imagen	1,37 [s]	1,17 [s]	0,56 [s]

$PTrack$ tuvo 2 fallas en la iteración 2 y 3 teniendo un total de 4 fallas, lo cual quiere decir que no se pudo procesar un promedio de 1.3 imágenes en las pruebas con $PTrack$.

Tiempos de procesamiento por sesión de trabajo

La configuración se presenta en el capitulo 5.2.

Video	Configuración 4 $PTrack$ [m]	Configuración 3 $PTrack2$ [m]	Configuración 2 $PTrack2$ [m]	Configuración 1 $PTrack2$ [m]
1	47,22	31,57	28,90	22,13
2	46,82	31,67	29,08	22,42
3	50,07	31,32	28,77	22,40
4	48,57	31,52	28,78	22,08
5	49,07	31,40	28,83	22,40
6	50,78	31,63	28,93	22,42
7	53,50	31,53	29,03	22,27
8	51,32	31,53	28,78	22,23
9	55,22	31,63	29,17	22,42
10	57,38	31,90	29,18	22,60
11	57,08	31,87	29,55	22,70
12	62,72	32,02	29,20	22,60
13	65,53	31,75	29,45	22,48
14	63,68	31,63	29,15	22,40
15	66,90	31,97	29,67	22,67
16	71,13	31,98	29,77	23,07
17	69,50	31,87	29,72	22,83
18	70,83	32,30	29,90	22,88
19	76,42	32,53	30,40	23,27
20	76,27	32,33	30,47	23,13
21	76,07	32,43	30,53	23,25
22	76,03	32,87	30,52	23,18
23	130,33	32,78	30,78	23,67
24	76,53	32,72	30,80	23,28
25	75,53	32,70	30,82	23,37
26	79,08	32,63	31,18	23,50
27	88,40	32,83	31,30	23,67
28	79,30	33,13	31,73	23,67
29	99,23	33,33	32,07	23,95
30	80,12	33,73	32,07	23,88
31	79,70	33,45	31,67	23,97
Total	2130,33	998,57	930,20	710,78
Desviación Estándar	17,69	0,66	1,06	0,58
Tiempo en horas	35,51 [Hrs]	16,64 [Hrs]	15,50 [Hrs]	11,85 [Hrs]

Apéndice B

A continuación se presentan ejemplos de archivos de configuración para $PTrack2Bash$ como para $PTrack2BashFolders$

Configuración para $PTrack2Bash$

```
#----------------------------------------
# Ejemplo de archivo de configuración
# Modifíquelo a gusto
#----------------------------------------
# IP de servidores
# $HOST ya se encuentra en las variables
# y representa el computador en que se esta
# ejecutando este programa ahora.
$CUASI1D            ='172.17.55.7';
$GRANOS2D           ='172.17.55.91';
$MPLMFE             ='172.17.55.65';
$COREI7             ='172.17.55.236';
#----------------------------------------
# Arreglo de servidores
@SERVER             = ($HOST, $CUASI1D, $MPLMFE, $COREI7);
@PARALLEL_LIMIT     = (8, 4, 4, 8);
@CUDA               = (0, 0, 0, 1);
@CUDA_DEV           = (0, 0, 0, 2);
#----------------------------------------
# Limite de intentos por imagen
$TRY_LIMIT=3;
#----------------------------------------
# Usuario para conectarse a las otras maquinas
$USER="ptrack";
```

Cada variable representa un elemento de configuración

- La variable ***@SERVER*** representa los servidores que se ocuparan para el procesamiento.
- ***@PARALLEL_LIMIT*** representa la cantidad de imágenes en paralelo que se procesaran en cada servidor.
- ***@CUDA*** establece si dentro de un servidor se ejecutará $PTrack2$ usando CUDA o no, 1 y 0 respectivamente.
- ***@CUDA_DEV*** establece cuántos dispositivos CUDA se utilizarán si se aplica.
- ***$TRY_LIMIT*** establece el número de intentos en tratar de procesar una imagen, hasta el momento, usando $PTrack2$ no se ha presentado ningún problema en todas las imágenes procesadas por lo que ésta variable va a ser eliminada en versiones posteriores.
- ***$USER*** establece el nombre de usuario para conectarse a los servidores.

Configuración para *PTrack2BashFolders*

```
#-----------------------------------------
# Ejemplo de archivo de configuración
# Modifíquelo a gusto
#-----------------------------------------
# Carpeta de imágenes
IMG_FOLDER="ImgA"
# Carpeta de destino de datos
DEST="XYSL"
#-----------------------------------------
# Valores de D
Data_D[0]=8.63224
Data_D[1]=8.69859
Data_D[2]=8.59508
#-----------------------------------------
# Valores de W
Data_W[0]=1.30087
Data_W[1]=1.29494
Data_W[2]=1.27761
```

Cada variable representa lo siguiente:

- ***IMG_FOLDER*** representa la carpeta dentro de la cual ubicar las imágenes a ser procesadas.
- ***DEST*** representa el nombre de la carpeta de destino de los resultados.
- ***Data_D[X]*** representa el valor de D para la carpeta con numeración X.
- ***Data_D[W]*** representa el valor de W para la carpeta con numeración X.