UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
Departamento de Ingeniería en Informática

Desarrollo e implementación de un *framework* orientado al objeto para síntesis de imágenes en interferometría

Nicolás Salvador Muñoz Zarricueta

Profesor guía: Pablo Román
Profesor co-guía: Fernando Rannou

Trabajo de título presentado en conformidad a los requisitos para obtener el grado de Ingeniero de Ejecución en Computación e Informática

Santiago – Chile
2018

RESUMEN

El problema de la síntesis de imágenes en interferometría es un problema complejo que no posee solución única. En la actualidad la comunidad astronómica utiliza diferentes algoritmos para lograr reconstruir una imagen y poder probar sus hipótesis. Este proceso es lento y muchas veces requiere generar diferentes imágenes mediante el uso de muchos algoritmos y variantes de estos. Los *softwares* actualmente utilizados para generar estas imágenes permiten poca flexibilidad y por lo general están limitados a un conjunto muy pequeño de algoritmos, esto lleva a que los astrónomos intenten desarrollar sus propios algoritmos o variantes de algoritmos existentes. Pese a que los astrónomos poseen conocimientos de programación, la implementación de variantes de algoritmos en plataformas de alto es una tarea de alta complejidad que escapa de estos conocimientos. Es por esto que objetivo de este trabajo de titulación es proveer a la comunidad astronómica de un sistema de software del alto rendimiento que permita de forma sencilla, rápida y robusta la creación y modificación de variantes de algoritmos de síntesis de imágenes en interferometría. Para esto se desarrolla un software del tipo *framework* mediante el uso de la programación orientada al objeto. El sistema de software desarrollado utiliza como base de la implementación el software GPUVMEM, el cual, es una plataforma de alto rendimiento desarrollada en C++ y CUDA para la síntesis de imágenes en interferometría que utiliza datos provistos por el observatorio ALMA y datos simulados. Este nuevo sistema de software es comparado con su versión base en cuanto a rendimiento y calidad de imágenes obteniendo una pérdida de rendimiento computacional menor al 1 % en todos los conjuntos de prueba. Además, esta implementación es validada mediante la implementación de variantes de algoritmos por un desarrollador externo. El tema de este trabajo de título pertenece a la comunidad de astro informática de la USACH.

Palabras Claves: CUDA, C++, Astronomía, OOP, Interferometría, Síntesis de Imágenes, ALMA, Framework, HPC, GPGPU

Dedicado a Sonia Zarricueta, mi madre y compañera incondicional.

AGRADECIMIENTOS

Todo el apoyo entregado durante el desarrollo de este trabajo de titulación y durante mi estadía en la casa de estudios ha sido sin lugar a dudas uno de los principales motivos por los que hoy he podido llegar a este hito tan importante que es la titulación.

Es por esto que les doy las gracias por medio de estas breves palabras por el apoyo y la compañía incondicional de aquellos que han estado a mi lado durante este largo proceso que ha sido mi formación profesional.

En primera instancia a mi madre, hermana y abuelos que son mi círculo más cercano además de ser quienes siempre han estado ahí para acompañarme, guiarme y ayudarme en cada paso que he dado en la vida. A mis más cercanos amigos, por su constante apoyo, compañerismo y amistad que lograron que todas las veces que sentía mal por algún dilema de la vida pudiera salir adelante con una sonrisa. A mi profesor guía, el Dr. Pablo Román, por su gran labor como profesor, por su paciencia, por todo el apoyo brindado en este proceso, por los conocimientos y los consejos entregados, los que me han servido para crecer como persona y profesional. Por otra parte, quiero agradecer el apoyo FONDECYT Regular a través del proyecto 1171841 por el financiamiento parcial entregado para el desarrollo de esta memoria. Y por último quiero agradecer a la Universidad de Santiago de Chile por acogerme dentro de sus aulas y a los funcionarios del Departamento de Informática por toda la ayuda, el cariño y su gran labor.

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE ALGORITMOS

CAPÍTULO 1. INTRODUCCIÓN

En este capítulo, se describen las causales que llevan a desarrollar este trabajo de titulación. Además, se hace una presentación de la solución a desarrollar en este trabajo, definiendo sus objetivos, propósito y características.

1.1 ANTECEDENTES Y MOTIVACIÓN

El presente trabajo nace bajo el contexto de las actividades de síntesis de imágenes en radio interferometría con los datos obtenidos del Observatorio ALMA (Turner (2006)), el cual es un radiotelescopio ubicado en el norte de Chile. Un radiotelescopio recibe señales de frecuencias de radio provenientes del cielo y mediante el proceso de síntesis de imágenes se genera una mapa de emisión de energía.

El observatorio se compone de tres instalaciones distintas: el Centro de Operaciones (*Operation Support Facility*, OSF), el Sitio de Operaciones del conjunto de antenas (*Array Operation Site*, AOS), y la Sede Central en Santiago (Santiago Central Operation, SCO). El AOS es un instrumento astronómico compuesto de un interferómetro que comprende un conjunto de 66 antenas de radiofrecuencia (antenas también llamadas reflectores o radiotelescopios cuando es una única antena) de siete y doce metros de diámetro destinadas a observar longitudes de onda milimétricas y submilimétricas ($10 - 10^3$ GHz). Estas antenas presentan separaciones entre sí que van desde los diez metros a dieciséis kilómetros. Construido en el desierto de Atacama en el norte de Chile a una altitud de 5000 metros sobre el nivel del mar en el llano Chajnantor. El AOS es un instrumento revolucionario que gracias a su resolución y sensibilidad ha provisto a la comunidad astronómica de una nueva manera de observar el universo. Lo que permitiendo que los científicos puedan develar misterios astronómicos existentes desde hace muchos años mediante la exploración de nuevos orígenes cósmicos (Peck & Beasley (2008)). Las señales de radiofrecuencia capturadas por un par de antenas son correlacionadas obteniendo así visibilidades. Estas mediciones corresponden a la transformada de Fourier de la imagen del cielo y serán profundizadas más adelante en el documento. Anteriormente es mencionado que la separación máxima entre las

antenas es de 16 kilómetros, esto implica que la resolución estimada del AOS es equivalente con un hipotético telescopio con una apertura de 16.000 metros de diámetro. Técnicamente es imposible construir y operar una antena de estas dimensiones, por lo que se construyen muchas antenas y se combinan sus mediciones (ALMA (2009)).

La reconstrucción de imágenes de un interferómetro es un problema complejo. Los datos capturados por las antenas del AOS, no corresponden precisamente a imágenes de los objetos estudiados, sino más bien a muestras en el espacio de Fourier (plano (u, v)) de dichas imágenes. El teorema de Van Cittert–Zernike (Thompson et al. (1986)) (eq. 1.1) describe esta relación entre V y la imagen I del cielo. Esta relación es una transformada de Fourier y las coordenadas (u, v) corresponden a los vectores entre las antenas medidos en metros. Esta relación indica que para obtener una imagen del cielo usando las mediciones $V(u, v)$ se debe encontrar la transformada de Fourier inversa. Esto parece simple de hacer, pero dada la naturaleza de los datos muéstrales no es posible utilizar la transformada de Fourier inversa.

$$V(u,v) = \int_{-\infty}^{\infty} I(x,y) e^{2\pi i(ux+vy)} dxdy \tag{1.1}$$

El conjunto muestral se ubica en posiciones no regularmente espaciadas del plano (u,v) y esto implica que el algoritmo FFT (*Fast Fourier Transform*) no puede ser utilizado (Levanda & Leshem (2010)). Para poder utilizar los conocidos algoritmos que implementan la transformada de Fourier se requiere estrictamente que las muestras se ubiquen en un cuadriculado regular. Este problema se conoce como síntesis de imágenes.

La Figura 1.1 ejemplifica lo irregular que resulta ser un conjunto muestral obtenido para uno de los objetos estudiados por ALMA en el plano (u,v), en donde cada punto de este plano representa un valor complejo en el espacio de Fourier.

Es posible observar el efecto que produce la rotación de la tierra sobre los puntos en el plano (u, v) mientras se sigue al objeto estudiado, dicho efecto produce que los puntos en el plano (u, v) se desplacen de manera elíptica. La Figura 1.2 muestra un imagen generada mediante el algoritmo CLEAN (Högbom (1974)) de los datos en el plano (u, v).

Para generar imágenes de alta calidad (Figura 1.3) que sean consistentes

con el objeto estudiado, se deben desarrollar y configurar distintas implementaciones de prototipos de algoritmos ya que la síntesis de imágenes es inestable en relación a los datos. Esto implica que se debe aproximar una solución restringiendo el problema de reconstrucción de la imagen hasta que se obtenga un solución única para dicho conjunto muestral (Levanda & Leshem (2010)).

Actualmente existen dos grandes tipos de algoritmos: aquellos basados en heurísticas *ad-hoc* y aquellos basados en optimización no lineal (Bayesiano). Estos últimos tienen una base estadística sólida y son poco usados por su alto costo computacional.

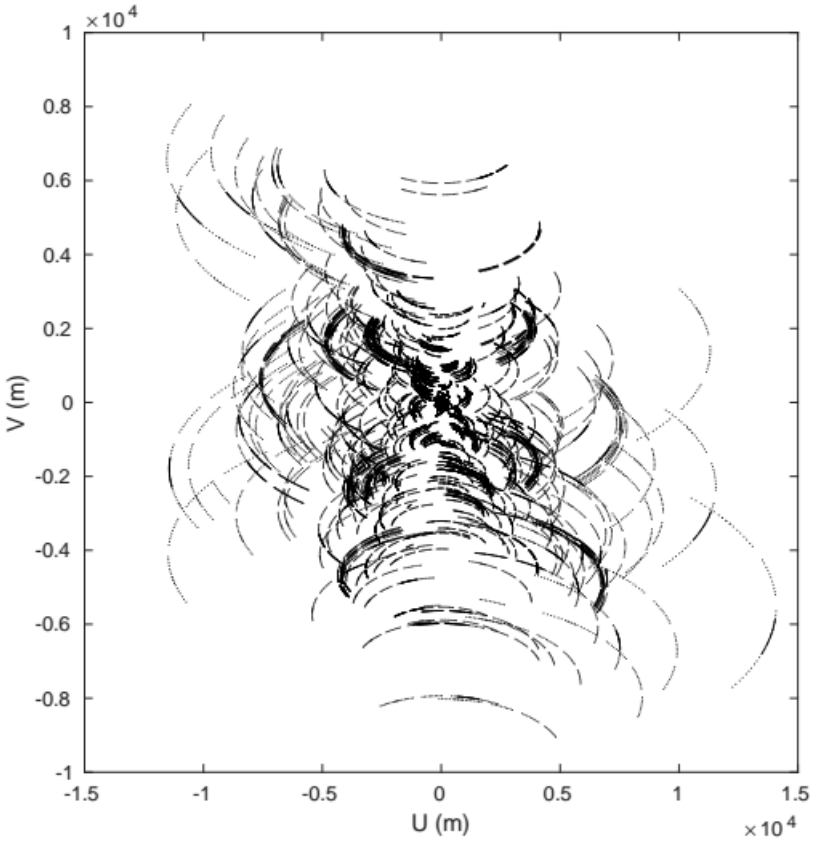

Figura 1.1: HlTau banda 7.
Fuente: ALMA, (ALMA (2014)).

Por otro lado, las tarjetas de vídeo o GPU, son componentes de *hardware* que tradicionalmente sólo han sido usadas en áreas particulares (juegos de computadoras, películas 3D e interfaces gráficas). Ahora último con el auge de la computación científica se han convertido en unidades de propósito general (GPGPU). Esto gracias a que durante

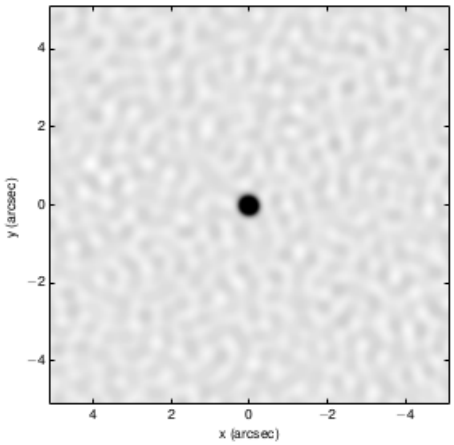

Figura 1.2: Imagen generada usando el algoritmo CLEAN.
Fuente: ALMA.

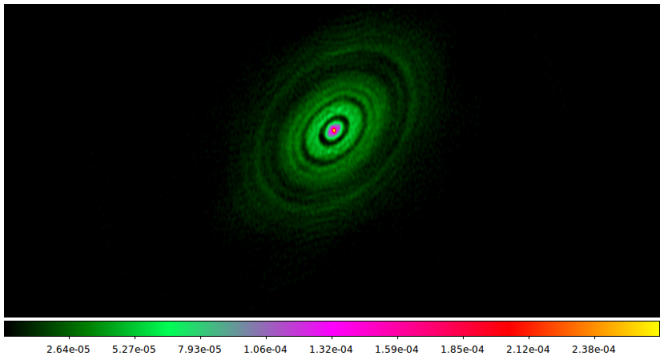

Figura 1.3: Imagen generada usando el algoritmo MEM.
Fuente: Elaboración propia.

los últimos años la comunidad científica ha presentado un notable interés en aprovechar las capacidades de estos *chips* para realizar cómputos de manera paralela. Es así como las GPGPU han pasado de ser usadas solamente en áreas particulares a ser utilizadas para llevar acabo simulaciones, aplicaciones de visión por computador, procesamiento y análisis de señales, procesamiento de imágenes, minería de datos (Owens et al. (2007)) e incluso minería de criptomonedas. Estos dispositivos están diseñados bajo una arquitectura SIMT (*single instruction multiple thread*) para administrar y ejecutar grupos de hebras. Si bien un grupo de hebras puede ejecutar las mismas instrucciones al mismo tiempo, en SIMT las hebras adicionalmente ejecutan estas instrucciones de manera independiente y asíncrona (Cheng et al. (2014)). Gracias a esto último, es que los problemas que pueden ser paralelizados se ven beneficiados en gran medida.

1.2 DESCRIPCIÓN DEL PROBLEMA

El problema de síntesis de imágenes no tiene solución única (Levanda & Leshem (2010)). Esto implica que cuando un astrónomo quiere comprobar alguna hipótesis sobre la imagen interferométrica debería explorar el conjunto más grande posible de soluciones. Estas incertezas se resuelven comparando los resultados de distintos algoritmos y configuraciones de variantes de estos. Cada método de resolución del problema introduce un sesgo propio que el astrónomo debe interpretar para comprobar sus hipótesis.

Antes de las construcción del observatorio ALMA, se disponía de pocos datos interferométricos, por lo que sólo se podía ajustar modelos de pocos parámetros al conjunto de datos. Por ejemplo en el caso de un disco protoplanetario (Marti-Vidal et al. (2014)) éste se modelaba como una elipse y el resultado de este ajuste era el semi-eje mayor, menor e inclinación. Ahora que los radio-interferómetros están en el régimen de alta fidelidad, se pretende realizar este ajuste en función de una imagen en vez de un modelo de pocos parámetros. Aunque el astrónomo aún es quien discierne entre diferentes imágenes.

Los interferómetros no obtienen la imagen del cielo directamente, sino que obtienen un conjunto disperso e irregular de visibilidades, que corresponden a muestras de las transformada de Fourier de la distribución del brillo del cielo en el plano (u, v). A medida que la tierra gira, las antenas actualizan la posición del cielo a la que apuntan para continuar capturando datos del objeto astronómico. Este conjunto muestral es procesado por medio de distintos algoritmos y técnicas para así reconstruir una imagen.

La síntesis de imágenes de interferometría trae consigo una familia de algoritmos para resolver la problemática anteriormente descrita. El algoritmo CLEAN (Högbom (1974)) es rápido y bien conocido por la comunidad astronómica. El sesgo que tiene CLEAN es el de ajustar bien las fuentes puntuales y despreciar las fuentes extendidas. Otro algoritmo basado en optimización es el método de máxima entropía (MEM) (Cornwell & Evans (1985)) que minimiza el error cuadrático sujeto a ciertos niveles de entropía. MEM tiene el sesgo de generar imágenes más atenuadas que CLEAN, pero con mayor resolución.

Mejorar la resolución de las imágenes generadas, implica la utilización de algoritmos y técnicas muchos más complejas tanto en su base teórica como en su complejidad computacional. Esto se traduce en que para obtener mejores imágenes se requiere de más investigación y mejores herramientas computacionales tanto de *software* como de *hardware*. Dado el volumen de datos y la complejidad computacional de los algoritmos utilizados, muchas veces reconstruir una imagen es un proceso que demora desde días hasta meses. Es por esto que se utilizan herramientas y soluciones de computación de alto rendimiento.

La reconstrucción de imágenes usando radiointerferómetros que captan señales del cielo a medida que rota la tierra es un problema *ill-posed* (Chen (2011)), debido a la irregularidad de los muestreos en el dominio de Fourier. Esto se traduce en que los muestreos resultan en sobreposición de ondas en la imagen debido a los altos lóbulos laterales en la respuesta del conjunto de antenas. Como se debe tener en consideración los errores del instrumento en las mediciones, este problema no tiene solución única, ya que existen múltiples imágenes a las que este conjunto puede ser aproximado.

Los astrónomos son profesionales altamente capacitados en proceso de análisis de datos astronómicos, teniendo conocimientos en distintas áreas de este proceso. Esto hace del astrónomo un candidato ideal para el desarrollo y configuración de variantes de algoritmos. Sin embargo su foco es la investigación astronómica para lo cual necesitan un rápido desarrollo de herramientas computacionales. Esto en mayor parte se ve afectado por la dificultad de implementación de las variantes de algoritmos. Las mayor complejidad a la hora de implementar nuevos algoritmos y variantes de los mismos, es la gran dificultad de implementación de estos en plataformas de alto rendimiento.

Si bien la comunidad astronómica necesita reducir la dificultad de experimentar con configuraciones de variantes de algoritmos, también requiere que las variantes a utilizar permitan reutilización y extensión, ya que muchas veces se requiere realizar pequeños ajustes a los algoritmos. Como la reconstrucción de imágenes es un proceso lento y complejo computacionalmente, las variantes de algoritmos deben ser aplicaciones o soluciones del alto rendimiento para obtener resultados de estas configuraciones en el menor tiempo posible.**Proveer al astrónomo de una herramienta de alto rendimiento que permita configurar y crear variantes de algoritmos de síntesis de imágenes de**

manera flexible, robusta y extensible, es la problemática que pretende abordar este trabajo.

1.3 SOLUCIÓN PROPUESTA

La solución propuesta para esta problemática es un *framework* orientado al objeto, implementado con el uso de patrones de diseño basado en la implementación de GPUVMEM (Cárcamo et al. (2018)). GPUVMEM es una solución de computación de alto rendimiento para la síntesis de imágenes en interferometría. El *software* GPUVMEM implementa el método de la máxima entropía para resolver la problemática de la síntesis de imágenes, aprovechando la alta capacidad de computo paralelo que presentan las unidades de procesamiento gráfico (GPU). Este *software* fue desarrollado utilizando la tecnología CUDA 7.0 y programado en el lenguaje C. En la actualidad este *software* está mucho más avanzado y ha sido migrado a la tecnología CUDA 9.2 y reescrito en el lenguaje C++11. Se decide su uso como base de este trabajo de título por ser una plataforma de alto rendimiento validada, en la que actualmente se realizan trabajos investigativos con variantes de algoritmos. Además este *software*, al estar escrito en C++11, es compatible con la programación orientada al objeto.

En la actualidad, uno de los paradigmas de programación más utilizados corresponde al *paradigma de programación orientado al objeto*. Este paradigma permite que el programador diseñe las aplicaciones de una perspectiva distinta. Puesto que, durante el diseño el programador se enfoca primeramente en el descubrimiento de los objetos que constituyen la aplicación antes de preocuparse de las funcionalidades específicas de esta. Esta nueva forma de modelar los problemas del mundo real, genera aplicaciones modulares, extensibles y reutilizables. La principal meta de este trabajo de título consiste en dar solución a la problemática planteada anteriormente, es decir la configuración rápida de variantes de algoritmos de síntesis de imágenes en interferometría, tal que:

- Su implementación permita la reutilización y extensión de las configuraciones de variantes de algoritmos.

- Poder realizar múltiples configuraciones con la menor cantidad de cambios para así reducir el tiempo de implementación.
- Su implementación no disminuye en más de 3 % el rendimiento en comparación a el *software* base GPUVMEM.
- Permitir flexibilidad de configuraciones hasta el nivel de *kernel* GPU.

1.3.1 Características de la solución

1. Proveer de un conjunto de clases, métodos y estructuras estandarizados para la síntesis de imágenes de interferometría.
2. Proveer de un conjunto mínimo de clases asociados a algoritmos iterativos de síntesis de imágenes en interferometría, mediante las especializaciones características de la orientación a objetos.
3. Permitir la configuración de variantes de algoritmos en el *software* base *GPUVMEM*.
4. Establecer resultados a través de *benchmarks*. Esto con fines comparativos e investigativos.

La solución será evaluada mediante la implementación de un sintetizador para la reconstrucción de imágenes mediante el uso de esta herramienta. Luego la solución se evaluará de manera cuantitativa utilizando datos de prueba provistos por el observatorio ALMA, comparando los resultados obtenidos con las implementaciones en el *paradigma imperativo/procedural* de estos algoritmos. Los datos a comparar entre otros, serán resolución, valor cuadrático medio, error cuadrático medio normalizado (ECMn) y rendimiento. Una gran similitud a nivel de rendimiento y parámetros demuestra la correcta implementación de esta solución.

Una segunda evaluación consistirá en la modificación o creación de un nuevo algoritmo basado en una de las implementaciones de los algoritmos para GPU. Por ejemplo, algunas modificaciones que se podrían hacer al algoritmo implementado son: cambio de la función de penalización, cambio del método de interpolación, modificación

del algoritmo de optimización, entre otras. Dichas pruebas las realizará un desarrollador externo al presente trabajo quien realizará una evaluación respecto a la conformidad con la herramienta desarrollada. Estas pruebas son desarrolladas en forma de formulario minimizado basado en la ISO 17.000. Esto con el fin de proveer de la retroalimentación necesaria para establecer una medición de la prueba.

1.3.2 Propósito de la solución

Permitir al astrónomo configurar variantes de algoritmos de manera rápida, robusta, reutilizable y extensible en una plataforma de alto rendimiento. Para así explorar los diferentes conjuntos de imágenes posibles de los objetos o fenómenos astronómicos a estudiar.

1.4 OBJETIVOS Y ALCANCE DEL PROYECTO

1.4.1 Objetivo general

El objetivo general del proyecto es diseñar y desarrollar un *framework* orientado al objeto, implementado con patrones de diseño para la creación, modificación y evaluación de variantes de algoritmos para síntesis de imágenes en interferometría. El sistema estará basado en la implementación del *software* **GPUVMEM** (Cárcamo et al. (2018)) , debe permitir extensión y modificación manteniendo la mayor similitud de rendimiento con el *software* original.

1.4.2 Objetivos específicos

1. Identificar las clases principales en algoritmos basados en optimización.

2. Diseñar las interfaces, es decir, diagrama de clases y sus relaciones.
3. Definir el comportamiento básico de las clases comprendidos en el diagrama de clases.
4. Construir clases especializadas que implementan el comportamiento de las clases base.
5. Evaluar la solución, construyendo una variante de algoritmo mediante el uso del *framework*.
6. Evaluar la solución mediante una prueba de conformidad por parte de un desarrollador externo.

1.4.3 Alcances y limitaciones de la solución

Este trabajo utiliza los lenguajes de programación C++11 y la plataforma de computación paralela CUDA en su versión 9.2. Además la solución está orientada al desarrollo de algoritmos iterativos de síntesis de imágenes con datos obtenidos del observatorio ALMA. En particular se restringe a considerar algoritmos que resuelven un problema de optimización en base a una función objetivo. Junto con, esto una de las principales limitantes para el diseño y desarrollo de la solución, está dada por la máxima pérdida de rendimiento en comparación al *software* base *GPUVMEM*, la cual, se establece en no más de 3 %.

1.5 METODOLOGÍA Y HERRAMIENTAS UTILIZADAS

1.5.1 Metodología

Para el desarrollo del proyecto, se utilizan dos metodologías. Una metodología para el diseño de la arquitectura y otra para la implementación del diseño antes generado.

La metodología escogida para el diseño arquitectural del *software* es Attribute-Driven Design (ADD) en su versión 2.0 (Wojcik et al. (2006)). Esta metodología considera las cualidades de los atributos de software, lo que en este caso es necesario, ya que se necesita que el software sea extensible, flexible y modificable sin tener una pérdida de rendimiento significativa.

La metodología ADD cuenta con cuatro pasos, teniendo tantas iteraciones como sean necesarias. El primer paso consiste en seleccionar un elemento del sistema a diseñar. Como segundo paso se deben identificar los requerimientos arquitecturalmente significativos del elemento seleccionado. El tercer paso consta de generar una solución de diseño para el elemento seleccionado. El cuarto paso consiste de indicar los requerimientos faltantes, y seleccionar la entrada de la siguiente iteración.

Para la implementación de la solución se utiliza la metodología de desarrollo XP (*eXtreme Programming*) Beck & Gamma (2000). El lugar de trabajo (Laboratorio de Optimización y Computación Paralela) permite la interacción con la persona que desarrolló el sistema de *software GPUVMEM*, que es usado como base en esta memoria. Se trabajar directamente con esta persona durante todo el desarrollo, cumpliendo así con la metodología. Además como retroalimentación se tienen reuniones semanales con el profesor guía para mostrar avances y recoger las acotaciones pertinentes.

1.5.2 Herramientas de desarrollo

Para el desarrollo y documentación del trabajo de titulación, son utilizados los siguientes elementos de *hardware* y *software*.

1.5.2.1 Hardware

- Computador Personal del Alumno
  - Procesador Intel Core I7 7500U
  - RAM 8GB

    - GPU Nvidia GeForce 940 mx 2GB
- Computador Laboratorio de Operaciones
    - Procesador AMD Athlon(tm) II X4 630 Processor
    - RAM 16GB
    - 2 GPU Nvidia Quadro M4000 8GB

1.5.2.2 Software

- Ubuntu 16.04
- CUDA 9.2
- C++11
- Atom 1.26.1
- StarUml 3.0.0
- Git
- Casacore
- Cfitsio 3.37

1.6 ORGANIZACIÓN DEL DOCUMENTO

El presente documento se encuentra seccionado en 6 capítulos: Introducción, Estado del Arte, Diseño de la solución, Implementación de la solución, Experimentos y finalmente Conclusiones.

- En el capítulo 1 Introducción, se presentan las motivaciones y antecedentes junto con una descripción del problema. Además se presenta la solución a desarrollar junto con los objetivos y metodologías de trabajo a seguir.

- En el capítulo 2, Marco teórico y estado del arte, se tratan los principales temas que componen este trabajo de título junto con los trabajos relacionados en el área de la síntesis de imágenes en interferometría.
- En el capítulo 3, Diseño de la Solución, se explica cómo se modelan los objetos para incorporar las características (dentro de las limitantes del proyecto) de un sintetizador.
- En el capítulo 4, Implementación de la Solución, se presenta la construcción de los objetos concretos junto con la creación de un sintetizador particular. Además se da una descripción de por qué y cómo se llega a esa implementación.
- En el capítulo 5, Experimentos, se presenta el diseño de los experimentos, detallando la pérdida de rendimiento para todos los casos de prueba presentados. Además se diseña y muestra la prueba de conformidad con el *software*, conociendo así el estado final de este.
- En el capítulo 6, Conclusiones, Se presenta el análisis final de las pruebas obtenidas y se validan los objetivos planteados, para finalmente realizar una evaluación general del proyecto y sugerir posibles trabajos futuros.

CAPÍTULO 2. MARCO TEÓRICO Y ESTADO DEL ARTE

2.1 MARCO TEÓRICO

En este capítulo se muestran los conceptos necesarios necesarios para entender el desarrollo de este trabajo de titulación. Además, se muestra el estado del arte de la disciplina en la que se enmarca este trabajo.

2.1.1 Principios de la interferometría y síntesis de imágenes

La interferometría es una técnica utilizada para obtener imágenes de alta resolución de un objeto o fenómeno astronómico particular. Esto implica la combinación de señales de diferentes antenas separadas físicamente. La imagen de la distribución del brillo del cielo puede recuperarse a través del muestreo de visibilidades complejas en el plano (u, v). En otras palabras, la imagen corresponde a la transformada de Fourier de las visibilidades, donde cada una de estas tiene una amplitud y una fase, representando la transformada de Fourier del brillo y la posición angular de la emisión.

Un interferómetro obtiene un conjunto disperso e irregular de visibilidades, las que corresponden a muestras de las transformada de Fourier de la distribución del brillo del cielo en el plano de la imagen. A medida que la tierra gira, las antenas van actualizando la posición a la que apuntan en el cielo, de esta forma se recolectan nuevos datos dentro de otros puntos del plano de Fourier.

Como se menciona anteriormente, los interferómetros solo capturan un conjunto disperso e irregular de visibilidades. Estas muestras no corresponde a posiciones regulares de una grilla de Fourier, sino que a posiciones arbitrarias. Esto implica que para poder aplicar la transformada inversa en este conjunto de visibilidades, estas deben ser interpoladas en una grilla con espaciamiento regular. Esto introduce factores adicionales de error.

El muestreo incompleto en el plano (u, v) es el origen de las complicaciones en la síntesis de imágenes. De esta manera obtener una imagen del cielo se convierte

en un problema inverso que requiere algoritmos de síntesis especializados (Levanda & Leshem (2010)).

2.1.2 Programación en GPU

Para efectos de este trabajo de título se revisan los conceptos maás importantes de programación en GPU con CUDA, de manera más específica, los conceptos de *kernel*, *grilla*, *bloque*, *warp*, y *hebra*.

La unidad física más pequeña de cómputo en una GPU es la unidad llamada *CUDA core* o *Streaming Processor* (SP), la cual, puede llevar a cabo operaciones aritmético lógicas así como operaciones de punto flotante, aunque esta unidad es mucho más simple que una unidad central de procesamiento (CPU por nombre en inglés) tradicional. Un conjunto de SP forman un *Streaming Multiprocessor* (SM). Todos los SP de un SM ejecutan la misma instrucción. Es por esto que son clasificadas dentro del término *Single Instruction Multiple Thread* (SIMT) (Cheng et al. (2014)). Siguiendo la misma lógica, la unidad más pequeña de ejecución es la hebra. Cada SP de un SM ejecuta secuencialmente una hebra hasta que esta termina o hasta que los datos que requiere no se encuentren disponibles en los registros del SP, en cuyo caso, se realiza un cambio de contexto tal que un nuevo grupo de hebras ingresa al SM.

Toda funcion que se llama desde el *host* (CPU) para que se ejecute en el *device* (GPU) se llama *kernel*. Un *kernel* es la unidad básica de programación en CUDA para GPU. Este es ejecutado paralelamente en el *device* por las hebras de la aplicación. Profundizando en el concepto de *kernel* una **grilla** es la forma lógica de organización de un *kernel*. Las grillas pueden tener distintas geometrías, estás pueden ir desde 1 dimensión hasta 3 dimensiones. En la Figura 2.1 se muestra una grilla de 1 dimensión.

Además la Figura 2.1 muestra que las grillas se dividen en **bloques** de hebras y que de la misma forma, todos los bloques de una grilla tienen la misma dimensión. Es muy importante no confundir la organización lógica de un *kernel* con la arquitectura de GPU, puesto que, los conceptos de grilla y bloque corresponden a una estrategia de solución del *kernel* y no a características físicas de la GPU.

Cuando una grilla junto con sus bloques ha sido organizada, el programador

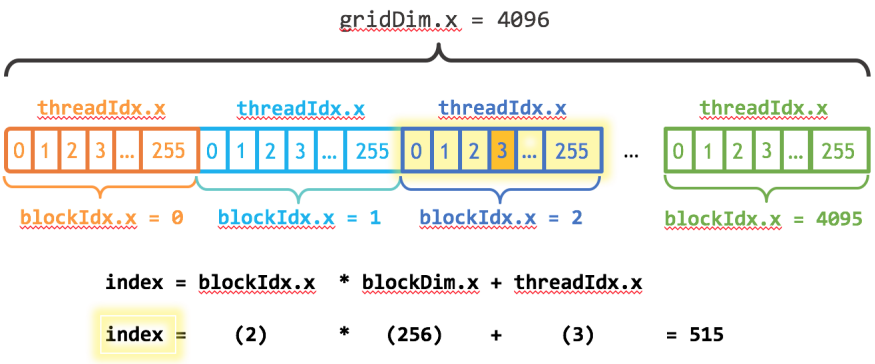

Figura 2.1: Grilla lógica de una dimensión.
Fuente: NVIDIA Developers blog, 2011.

puede hacer uso de diferentes identificadores para encontrar la posición global y local de una hebra, como se muestra en la Figura 2.1. De manera más específica los identificadores son:

- *blockIdx.x, blockIdx.y, blockIdx.z*: el identificador del bloque en una grilla, en cada una de sus dimensiones.
- *threadIDx.x, threadIDx.y, threadIDx.z*: el identificador de una hebra en el bloque, en cada una de sus dimensiones.
- *gridDim.x, gridDim.y, gridDim.z*: el número de bloques en cada dimensión de una grilla
- *blockDim.x, blockDim.y, blockDim.z*: el número de hebras en cada dimensión de un bloque.

Dependiendo de la geometría de los bloques los identificadores de las dimensiones adicionales no son utilizados. En un bloque unidimensional como el que se muestra en la Figura 2.1 el identificador global de una hebra se define como $tid = blockDim.x * blockIdx.x + threadIdx.x$, en donde se multiplican la dimensión del bloque y el identificador del bloque de una hebra en particular. Para luego a este resultado sumarlo con el identificador del bloque de la hebra.

Dentro de la organización de un bloque, se define **warp** como un grupo de 32 hebras en un bloque que se ejecutan a la vez, las cuales, tienen asignados recursos

tales como registros y memoria compartida. Un SP puede alojar varios *warps* en distintos bloques. Según la cantidad de registros que un *kernel* necesite y del tamaño del bloque, un SP tendrá más (o menos) oportunidad de realizar cambios de contexto y, por lo tanto, mostrar una mejora en el rendimiento. Sin embargo, un SP siempre tiene un número máximo de *warps* que puede albergar. Para efectos de este trabajo de título, se utilizan 2 **NVIDIA M4000** con una capacidad de cómputo de 5.2, las cuales, pueden albergar un número máximo de 64 *warps*.

Parte importante de este trabajo de título esta dado por la utilización de la programación en CUDA. Como se menciona en Cárcamo et al. (2018) el problema de la síntesis de imágenes es *embarrassingly parallel*, lo que implica que se requiere muy poco esfuerzo para separar el problema en múltiples tareas paralelas. Además la ejecución de estas tareas paralelas no tienen dependencia, necesidad de comunicación o de resultados entre ellas. Esto las convierte en candidatas perfectas para la utilización de SIMT.

2.1.3 Programación orientada a objetos

La programación orientada a objetos (POO) (Rentsch (1982)) es un paradigma de diseño y programación que se enfoca principalmente en modelar y programar sistemas del mundo real, tal y como ellos ocurren en la realidad. En otras palabras, es definir los sistemas como un conjunto de objetos que interactúan entre sí. El diseñador se enfoca principalmente en las interacciones y fenómenos en lugar del cómo estos ocurren.

En el marco de la programación orientada a objetos, se definen 5 características de esta técnica, las cuales, se resumen a continuación.

- Abstracción: La abstracción es el mecanismo mediante el cual se reduce la complejidad a una entidad. Esto permite representar las características esenciales de un objeto sin preocuparse de las características restantes.
- Encapsulamiento: Es la propiedad que permite asegurar que el contenido de información de un objeto esté oculto al mundo exterio., Es así como un Objeto A no tenga acceso al comportamiento de un Objeto B, y viceversa. En otras

palabras es el proceso de ocultar todas las implementaciones de características y comportamientos propios de un objeto.

- Modularidad: Propiedad que permite subdividir una aplicación en unidades más pequeñas llamadas módulos. En donde cada uno debe ser tan independiente como sea posible de la aplicación en la que está incluido y a su vez de otros módulos. Básicamente es dividir la aplicación en módulos débilmente acoplados que puedan ser compilados de manera independiente, pero que mantengan relación o conexiones con distintos módulos.
- Jerarquía: Es una propiedad que describe y permite un orden en las abstracciones. Las 2 jerarquías más importantes dentro de un sistema de complejo son la estructura de clases o generalización/especialización, y estructura de objetos o agregación. El primer tipo es lo que se conoce como herencia de clases, donde una clase permite compartir su estructura o comportamiento en una o múltiples clases (herencia simple y herencia múltiple respectivamente). El segundo tipo se refiere al agrupamiento físico de estructuras que se interrelacionan lógicamente.
- Polimorfismo: El polimorfismo es la capacidad de los objetos de comportarse de distinta manera según el objeto con el cual interactúan.

Es gracias a estas características que la POO permite abordar proyectos de gran tamaño disminuyendo la complejidad de programación, a la vez que permite la reutilización de código aumentando la flexibilidad de las aplicaciones.

2.1.4 Framework Orientado al Objeto

Para definir un "*framework* orientado al objeto", es necesario introducir el concepto de *framework*. Un *framework* en el desarrollo de *software* es una estructura conceptual y tecnológica de asistencia definida, la cual, normalmente cuenta con módulos concretos de *software*. Estos pueden servir como base de la organización y desarrollo arquitectural de un *software*. Típicamente incluyen soporte y bibliotecas de *software* entre otras herramientas.

Un *framework* tiene como objetivo ofrecer una funcionalidad definida y auto contenida. Estos están construidos utilizando patrones de diseños y su principal característica es su alta cohesión y bajo acoplamiento. Para lograr esto se construyen módulos de *software* que vinculan las necesidades del sistema con las funcionalidades que este presta. De esta manera el desarrollo de módulos debe asegurar que estos sufran pocos o ningún cambio a lo largo del ciclo de vida del *framework*. Esto permite la portabilidad entre distintos sistemas.

Un patrón de diseño es el esqueleto de las soluciones comunes a problemas comunes en el desarrollo de *software*. En otras palabras, brindan una solución ya probada y documentada a a problemas del desarrollo de *software* que están sujetos a contextos similares. Los patrones de diseño cuentan con diferentes elementos que se deben tener en consideración, esto son:

- Su nombre.
- El problema: cuando aplicar un patrón.
- La solución: descripción abstracta del problema.
- Las consecuencias: costos y beneficios de implementarlo.

Además, un patrón de diseño puede ser clasificado en los siguientes grupos:

- Patrones Creacionales: Sirven para la inicialización y configuración de objetos.
- Patrones Estructurales: Tienen como objetivo separar la interfaz de la implementación. Se ocupan de cómo las clases y objetos se agrupan, para formar estructuras más grandes.
- Patrones de Comportamiento: Se usan para describir la comunicación entre clases u objetos.

Cuando se habla de "*framework* orientado al objeto"se refiere a la construcción de un *framework* utilizando las características inherentes de la programación orientada al objeto. En otras palabras, el desarrollo de los módulos de *software* se realiza en base a objetos que se relacionan entre sí. Lo que define una estructura casi estática de objetos con funcionalidad definida, los cuales, interactúan entre ellos para entregar las

funcionalidades clave del sistema de *software* o proyecto a desarrollar. Gracias a estas características un "*framework* orientado al objeto"simplifica el diseño de aplicaciones.

2.2 ESTADO DEL ARTE

En Levanda & Leshem (2010) se presenta el problema que actualmente tiene la disciplina de la radioastronomía, el cual es, la necesidad de nuevas técnicas de síntesis de imágenes. Esto debido al constante aumento del tamaño de los datos. Se presenta también los algoritmos existentes que resuelven este problema, mencionando que el algoritmo CLEAN es el algoritmo más comúnmente usado.

En Román (2012) se menciona la necesidad de investigar y mejorar los algoritmos actuales. Para esto se realiza una prueba de concepto, en donde se utiliza CLEAN y una propuesta de reconstrucción de imágenes llamada “Home made”. En la comparación de ambos se muestran las deficiencias del algoritmo CLEAN, ya que el algoritmo “Home made” obtiene una mejor calidad de imagen, medida en rango dinámico, y la nula presencia de *artifacts* que si están presentes en la imagen obtenida mediante CLEAN. Esto plantea la necesidad del continuo mejoramiento de los algoritmos actuales.

En el paquete de software CASA (Common Astronomy Software Applications) (McMullin et al. (2007)) se puede encontrar la implementación de los métodos CLEAN y MEM (método de máxima entropía). Este software sólo sirve para aplicar los algoritmos de síntesis de imágenes, y no permite modificarlos para proporcionar mejoras. Además, este paquete de *software* no es un plataforma de alto rendimiento.

En el paquete de software MIRIAD (Zhao (2013)) para el SMA (o Submillimeter Array es un radio interferómetro ubicado en Hawaii perteneciente al Smithsonian Astrophysical Observatory) se encuentran implementados los algoritmos CLEAN y MEM. Este paquete de software sólo permite la ejecución de la deconvolución mediante estos métodos. Este *software* no es una herramienta de alto rendimiento.

El software Difmap Pearson et al. (1994) es un software para producir imágenes con información proveniente de radio-interferómetros. Este utiliza el algoritmo CLEAN de síntesis de imágenes para la deconvolución. Difmap no permite la modificación de algoritmos, sólo permite la ejecución de estos.

GPUVMEM (Cárcamo et al. (2018)) es un software de alto rendimiento que aprovecha la capacidad de cómputo paralelo de las GPU para resolver la problemática de síntesis de imágenes para el caso del algoritmo MEM. En particular este software fue desarrollado en el Departamento de Ingeniería Informática de la Universidad de Santiago de Chile (DIINF), y será considerado como base para el desarrollo de la solución. Esta solución se plantea como una herramienta de alto rendimiento debido a que resuelve el problema de síntesis de imágenes para grandes conjuntos de datos, sólo siendo limitado por la cantidad de memoria (VRAM) disponible en las GPUs.

CAPÍTULO 3. DISEÑO DE LA SOLUCIÓN

En este capítulo, se presenta el diseño realizado para implementar el *framework* de síntesis de imágenes en interferometría. Como base de este diseño, se realiza un análisis del código de GPUVMEM, para así lograr una definición completa de los elementos que deben interactuar en el *framework*.

3.1 DESCRIPCIÓN DEL CICLO DE VIDA DE GPUVMEM

Como se mencionó anteriormente, GPUVMEM es una plataforma de alto rendimiento para la síntesis de imágenes en interferometría que implementa el algoritmo MEM (método de máxima entropía). Este programa realiza distintas rutinas implementadas como *kernels* a lo largo de todo el código. Estas rutinas son distintas entre sí y tiene distintos objetivos a lo largo del ciclo de vida de la aplicación, por lo que no son consideradas como objetos durante el diseño. Cabe destacar que la arquitectura CUDA no permite que una rutina (o función) que realiza cómputos en GPU sea encapsulado dentro de una clase, en otras palabras, un *kernel* no puede ser un método de una clase. Teniendo lo anterior en consideración, se describe el flujo de información a lo largo del ciclo de vida del programa GPUVMEM, en donde, mediante reiteradas reuniones e iteraciones de la metodología ADD se definen las clases principales del *framework*. El ciclo mencionado se muestra a continuación.

GPUVMEM es un sintetizador de imágenes (clase denominada **Synthesizer**) de imágenes radio interferométricas. Este sintetizador en su versión más usada es del tipo MFS (*multi-frequency synthesis*) (Sault & Wieringa (1994)), el cual, es un algoritmo de carácter iterativo que implementa la lógica de la reconstrucción de imágenes a partir de un conjunto de datos de múltiples frecuencias. Este conjunto de datos es obtenido desde la información almacenada en directorios de extensión *MS* mediante la ejecución de múltiples rutinas de I/O (clase denominada **Io**). Los directorios *MS* toman su nombre del inglés *Measurement Sets* (McMullin et al. (2007)). Dentro del programa GPUVMEM se utiliza la biblioteca de funciones para el procesamiento de datos radio astronómicos CASACORE (van Diepen (2015)), la cual se utiliza dentro de las rutinas de I/O.

El conjunto de los datos cargando por las rutinas de I/O mencionadas anteriormente es llamado visibilidades (clase denominada **Visibilities**), que corresponden a las mediciones de los interferómetros. Este conjunto consiste en varias estructuras de datos que representan las mediciones en el espacio de *Fourier* del objeto o fenómeno astronómico. A las visibilidades se les puede aplicar un filtro (clase denominada **Filter**), que por lo general consiste en transportar las visibilidades a una grilla o modificar su valores para eliminar valores no representativos del conjunto de datos (*Outliners*). Luego mediante diferentes funciones en el *host* y *kernels* en *device*, se utilizan estos datos para calcular variables de configuración, las cuales, serán referenciadas a lo largo del ciclo de vida del sintetizador.

Posteriormente, se recogen ciertos parámetros de entrada para definir una imagen inicial (clase denominada **Image**). A continuación, todos los datos calculados, las visibilidades generadas y la imagen inicial son enviados a memoria de *device* para ser minimizados (o maximizados) por un optimizador (clase denominada **Optimizator**), el cual en la última entrega del sistema de *software* GPUVMEM consiste en un método de optimización tipo gradiente conjugado que utiliza el método de Fletcher-Reeves-Polak-Ribiere (Press et al. (2007)). Mediante este método se intenta minimizar una función objetivo (clase denominada **ObjectiveFunction**). Esta función objetivo está definida en base a distintos elementos o componentes aditivos (objetos denominados **Fi**). La ejecución y recolección de resultado de los distintos componentes aditivos es coordinada por las rutinas de la función objetivo.

Los elementos de la función objetivo o componentes aditivos contienen la lógica del cálculo matemático en GPU. Algunos de estos componentes deben implementar lógica adicional como es el caso de Chi cuadrado, en donde se utiliza un procesador de imágenes (clase denominada **ImageProcessor**), el cual se encarga de realizar funciones específicas no relacionadas con el comportamiento esencial de un componente aditivo (modificación de la imagen, recómputo de la imagen, cálculo de variables necesarias para continuar la ejecución, etc.). Al terminar la rutina de minimización, se reconstruye la imagen final en formato *FITS*, nuevamente haciendo uso de una rutina de I/O.

Durante el flujo de información descrito, se identifican diferentes problemáticas de rigidez en el *softtware* GPUVMEM. Este *software* tiene 3 versiones de código, en

las cuales, se utilizan sintetizadores de distinto tipo, esto hace que los programadores deban realizar diferentes implementaciones si se desea utilizar algún tipo de sintetizador en particular. Por otro lado, este *software* sólo permite utilizar un método de optimización. Este método de optimización no es más que una compleja función, en la cual, muy poco de esta permite modificación y existe 1 versión de esta función para cada tipo de sintetizador. Además, este optimizador sólo permite trabajar con una función objetivo de 2 elementos o componentes aditivos. Si se necesita utilizar más de 2 elementos aditivos en la función objetivo, se debe volver a implementar el optimizador, los *kernels* que se utilizan para el cálculo del gradiente total de la función objetivo, los *kernels* utilizados para el cálculo del valor total de la función objetivo y ajustar los *kernels* utilizados para el cálculo del valor de la componente aditiva y su gradiente.

3.1.1 Consideraciones de diseño

Dentro del diseño de este *framework* se tienen las siguientes consideraciones de diseño:

1. Las clases deben ser diseñadas procurando no perder ninguna funcionalidad del *software* base GPUVMEM.
2. Modificar lo menos posible los *kernels* implementados.
3. Se debe procurar que el diseño de este *software* no comprenda una perdida de rendimiento mayor al 3 %.
4. Utilizar patrones de diseño siempre y cuando estos no impliquen cambios sustantivos en el código original de GPUVMEM.

3.2 DEFINICIÓN DE LAS CLASES DE LOS OBJETOS

En la esta sección se muestra el diseño de las clases nombradas en las descripción del ciclo de vida de GPUVMEM. ordenadas desde clases que representan

almacenes de datos, coordinadores, y por último las clases que realizan operaciones complejas con los datos.

3.2.1 Diseño de la clase Visibilities

Es una clase que funciona como almacén de datos para las visibilidades y su información asociada. Luego de ser procesadas y utilizadas para el cálculo de variables de configuración, estas son enviadas a memoria de *device*. Esta clase está definida como un contenedor, por lo que su diseño no contempla posterior especialización.

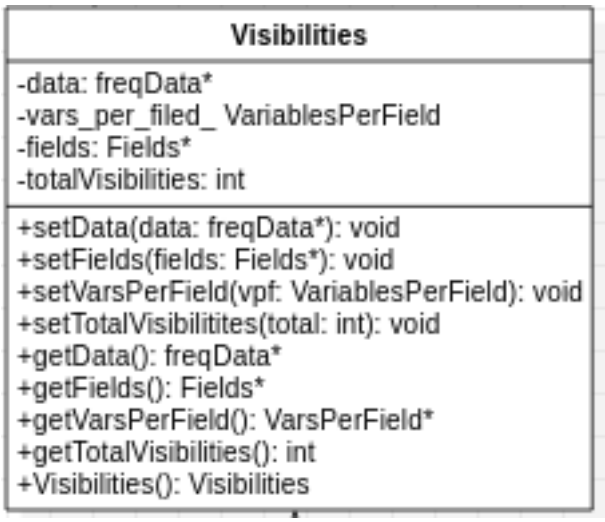

Figura 3.1: Diseño de la clase Visibilitites.
Fuente: Elaboración propia, 2018.

Al ser un contenedor, los métodos de esta clase, son sólo *seters y geters*.

3.2.2 Diseño de la clase Image

Image es la clase encargada de contener la imagen inicial junto con la información asociada a dicha imagen. Es importante destacar que una imagen pueden estar compuesta de múltiples imágenes. Desde un punto de vista más computacional, una imagen es un arreglo unidimensional de números que representan una matriz, es

por esto que dentro de este arreglo se pueden definir *n* imágenes y acceder a ellas mediante un índice. Esto es posible gracias a que todas las imágenes involucradas en el cálculo son de las mismas dimensiones. Se define esta clase como un contenedor, por lo cual no se diseña pensando en especialización posterior.

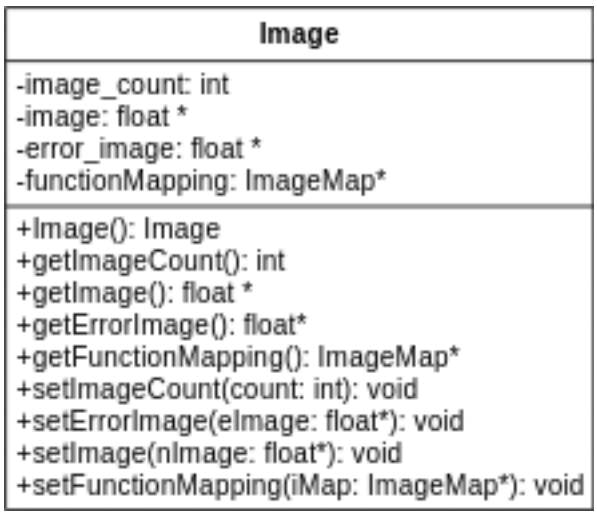

Figura 3.2: Diseño de la clase Image.
Fuente: Elaboración propia, 2018.

Los métodos de este objeto, no son más que *seters y geters*.

3.2.3 Diseño de la clase Io

El diseño de la clase *Io* es necesario para la recolección y entrega de información, permitiendo flexibilidad en este proceso. Aunque las visibilidades e imágenes tienen un formato estándar dentro de la disciplina, varían caso a caso, por lo cual, pueden implementar diferentes formatos de almacenamiento y salida. Esta clase es una interfaz para permitir especialización posterior.

En la Figura 3.3 se puede ver la clase *Io* junto con sus métodos más importantes, los cuales son:

- virtual void read(): Método encargado de realizar la lectura de archivos. Se deja la implementación de este método a criterio del programador, ya que si bien MS (se usa en GPUVMEM) es un estándar, esto puede cambiar en el tiempo.

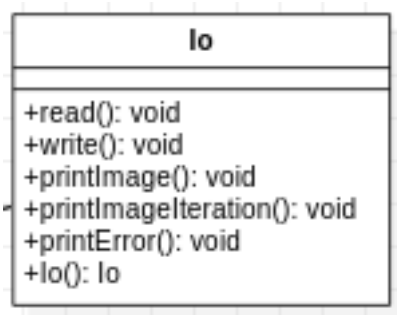

Figura 3.3: Diseño de la clase Io.
Fuente: Elaboración propia, 2018.

- virtual void write(): Es el método encargado de las rutinas de escritura. También la implementación de este método se deja a criterio del programador, por los mismos motivos que read().
- virtual void printImage(): Se encarga de escribir las imágenes en el disco. Si bien existe el estándar *FITS*, éste puede cambiar en el tiempo. Es por esto que la implementación de este método se deja a criterio del programador.
- virtual void printImageIteration(): Se comporta igual al método anterior, sólo que además contempla datos respectivos a la iteración en la que la imagen se encuentra.

3.2.4 Diseño de la clase Filter

La clase *Filter* es necesaria para poder aplicar distintas conversiones a los datos almacenados. Estas conversiones permiten al programador trabajr con subconjuntos de las visibilidades, preprocesar las visibilidades para el minar los *Outliners*, ajustar las visibilidades a una grilla (proceso conocido como *gridding*), etc. Debido a esto, la clase *Filter* está diseñada como una interfaz lo que permite flexibilidad en el conjunto de datos a procesar.

En la Figura 3.4 se puede ver la clase *Filter* junto con sus métodos más importantes, los cuales son:

- virtual void applyCriteria(): Método encargado de realizar la selección o procesa-

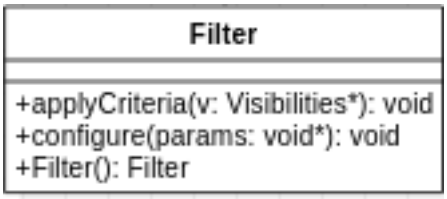

Figura 3.4: Diseño de la clase Filter.
Fuente: Elaboración propia, 2018.

miento de datos. Se deja la implementación a criterio del programador, ya que esta puede realizar la tarea que el programador necesite.

- virtual void configure(): Es el un *setter* encargado de adicionar parámetros a un filtro. También se deja a criterio del programador, ya que dependiendo de que filtro se implemente, los parámetros de configuración del mismo cambiarán.

3.2.5 Diseño de la clase Synthesizer

La clase *Synthesizer* (sintetizador en inglés) esta encargada de la coordinación de los diferentes elementos que componen el flujo general de la ejecución del programa y, además, contiene una instancia de las clases necesarias para lograr esta tarea. Si bien el comportamiento de un sintetizador no presenta cambios por lo general, este es diseñado como una interfaz para modificar las subrutinas asociadas a las tareas de coordinación. Es importante destacar que la implementación de los métodos de esta interfaz son responsabilidad del programador.

A pesar de que existen lineamientos generales de implementación para esta clase, muchas tareas e incluso llamadas a *kernels* pueden variar, es más, incluso las llamadas para distribuir, tratar y transportar los datos a memoria de *device* puede ser de una manera distinta.

En la Figura 3.5 se puede ver la definición de la clase *Synthesizer*, la cual fue diseñada como coordinador del flujo de los componentes. De esta manera la clase tiene asociados distintos métodos que realizan operaciones de memoria tanto en GPU como en CPU. Sus principales métodos son:

- virtual void setDevice(): Este método permite realizar la carga de las imágenes

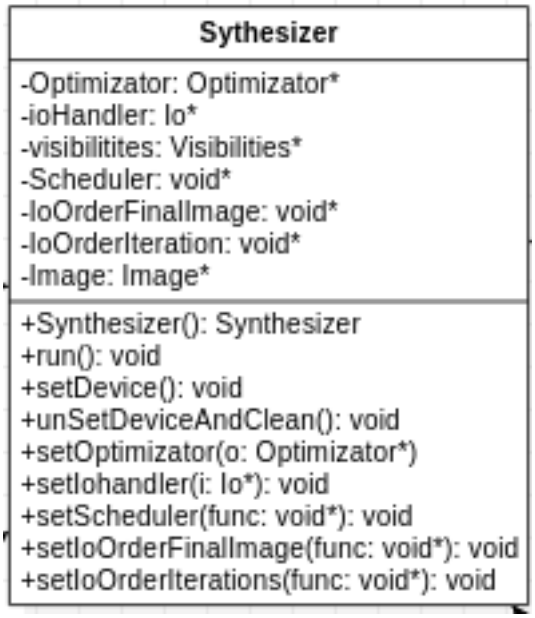

Figura 3.5: Implementación de la clase Sythesizer.
Fuente: Elaboración propia, 2018.

(instancia de clase Image), visibilidades (Instancia de la clase Visibilities) y otras variables en memoria de GPU.

- virtual void run(): Es el método encargado de ejecutar las instrucciones que llevan a cabo el flujo del *Synthetizer*.
- virtual void unSetDevice(): Se encarga de liberar la memoria de *device*, y la memoria del *host*.

3.2.6 Diseño de la clase Optimizator

El optimizador u *Optimizator* por su nombre en inglés, es la clase encargada de coordinar la rutina de minimización en base a una función objetivo (*Objective Function*). Esta clase está definida como una interfaz para permitir su posterior especialización.

En la Figura 3.6 se ven los principales métodos de la clase *Optimizator*, los que en más detalle son:

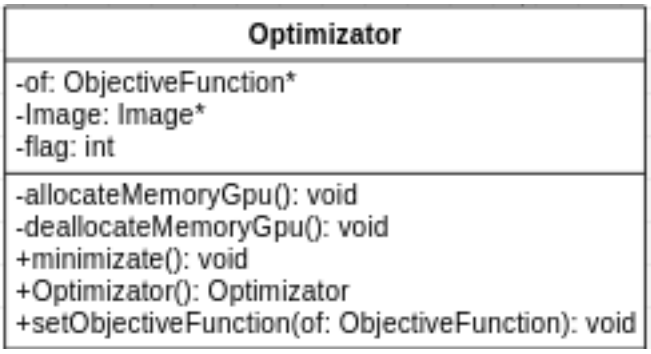

Figura 3.6: Diseño de la clase Optimizator.
Fuente: Elaboración propia, 2018.

- virtual void allocateMemoryGPU: Este método se encarga de transportar todos los atributos propios de un *Optmizador* a la memoria de *device*.
- virtual void deAllocateMemoryGPU: Similar al método anterior, pero en vez de transportar la memoria a *device*, la libera.
- virtual void minimizate(): Es el método donde se realiza toda la lógica propia de la minimización de la función objetivo.

Cabe destacar que la implementación de todos los métodos de esta interfaz quedan como tarea del programador. Lo anterior es porque cada método de optimización posee un algoritmo distinto.

3.2.7 Diseño de la clase ObjectiveFunction

En Cárcamo et al. (2018) se describen distintas técnicas de deconvolución utilizadas para resolver el problema de la síntesis de imágenes, y se define el uso del método de la máxima entropía como técnica a implementar en la plataforma GPUVMEM. Este método es implementado utilizando el método del gradiente conjugado para optimizar la función objetivo que se muestra en la ecuación 3.1 en donde S representa la máxima entropía y λ es un factor de penalización para S.

$$\phi = \chi^2 + \lambda S \tag{3.1}$$

Donde S se define como:

$$S = \sum_i I_i log \frac{I_i}{M} \tag{3.2}$$

Donde I es la imagen y M representa el menor valor de intensidad de un píxel. En el caso de que los datos provengan de un interferómetro, se tiene que:

$$\chi^2 = \sum \frac{\left| V_j^m - V_j^o \right|^2}{\sigma_j^2} \tag{3.3}$$

Donde la sumatoria recorre todas las visibilidades y el numerador de la fracción representa la diferencia entre las visibilidades modelo, creadas luego de realizar una transformada de Fourier a la imagen, llevar las visibilidades observadas a una grilla regular y luego realizar una interpolación bilineal y, las visibilidades observadas. Así, esta diferencia es llamada visibilidad residuo y se debe calcular el módulo del complejo para luego elevarlo al cuadrado. Por otra parte, σ_j es la varianza de cada visibilidad en función del ruido termal de las antenas.

Como se mencionó anteriormente, la minimización realizada por el optimizador, está basada en una función objetivo, la cual es la encargada de coordinar la ejecución de las rutinas de los diferentes componentes aditivos (*Fi*) que contiene. Estos componentes aditivos son los que se muestran en la ecuación 3.1, que conceptualmente, son regularizadores de la optimización. Esta clase no es diseñada como una interfaz sino más bien tiene un comportamiento híbrido entre contenedor y coordinador.

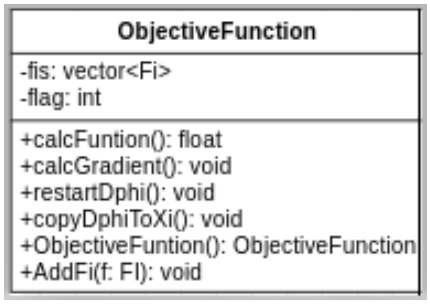

Figura 3.7: Diseño de la clase ObjectiveFunction.
Fuente: Elaboración propia, 2018.

Sus métodos pueden ser vistos en la Figura 3.7, de los cuales los principales son:

- float calcFunction(): Método encargado de coordinar la ejecución del cálculo del componente aditivo de la función objetivo *Fi*. Para luego recoger los valores de salida de las ejecuciones de los distintos componentes aditivos (*Fi*) y entregar la suma de todos los valores recogidos.

- void calcGradient(): Es el método que, de la misma manera que el método anterior, coordina la ejecución del cálculo del gradiente de cada componente de la función objetivo *Fi* y guardar sus resultados.

Es importante destacar que el *software* GPUVMEM sólo contempla la ejecución de 2 *Fi*, aunque actualmente tiene implementados alrededor de 5 *kernels* que pueden ser utilizados. Es por esto que el diseño de esta clase contempla el poder coordinar la ejecución y cálculo de múltiples *Fi* y sus gradientes. Permitiendo de esta manera al astrónomo realizar pruebas con variantes de algoritmos más complejas, que no estén limitados a la utilización 2 *Fi*.

3.2.8 Diseño de la clase Fi

Como se mencionó, un *Fi* corresponde a un regularizador de la función objetivo de carácter aditivo. La clase *Fi* es diseña con el propósito de ejecutar los diferentes *kernels* que procesan los datos e imágenes. La ejecución de estos *kernels* es lo que más tiene impacto en el rendimiento de la aplicación (en especial los gradientes), ya que, estos se ejecutan iterativamente durante todo el proceso de minimización. Para esto se define como una interfaz ya que las diferentes implementaciones de *Fi* pueden variar en complejidad dependiendo de la cantidad de cálculos asociados y elementos necesarios para llevar a cabo estos costosos cálculos.

La implementación objeto *Fi* que se aprecia en la Figura 3.8 muestra los métodos del objeto *Fi*. De los métodos mostrados, los principales métodos son:

- virtual float calcFi(): Es el método que se encarga de invocar a los *kernels* de GPU para el cálculo del valor del elemento. Este cálculo es utilizado en el cómputo de valor total de la función objetivo.

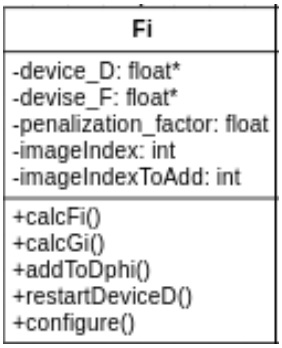

Figura 3.8: Diseño de la clase Fi.
Fuente: Elaboración propia, 2018.

- virtual void calcGi(): Se encarga de invocar a los *kernels* GPU para el cálculo del gradiente del elemento. Este cálculo se utiliza en el cómputo del gradiente total de la función objetivo.

3.2.9 Diseño de la clase ImageProcessor

ImageProcessor o procesador de imágenes es una clase encargada de agregar funcionalidades a los elementos *Fi*. debido a lo anterior esta clase se hace necesaria, ya que algunas implementaciones de *Fi* requieren cálculos particulares para entregar resultados o incluso modifican el contenido de la imagen durante su ejecución. La ejecución de estos cálculos particulares no corresponden a tareas propias de un *Fi* por lo que se diseña como una clase adicional que puede o no ser instanciada dentro de estos.

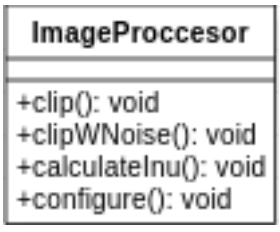

Figura 3.9: Diseño de la clase ImageProcessor.
Fuente: Elaboración propia, 2018.

En la Figura 3.9 se ve la implementación del objeto *ImageProcessor*, el cual tiene como métodos más importantes:

- virtual void clip(): Encargado de reiniciar los valores puntuales de las imágenes en función de valores calculados por el *Synthesizer*. Es utilizado
- virtual void clipWNoise(): Similar al método anterior con la diferencia de que este método incorpora el ruido de las imágenes al cómputo.
- virtual void calculateInu(): Se encarga de calcular un valor particular necesario para el cálculo de la trasformada de Fourier (FFT).

3.3 DIAGRAMAS

3.3.1 Diagrama de clases del framework

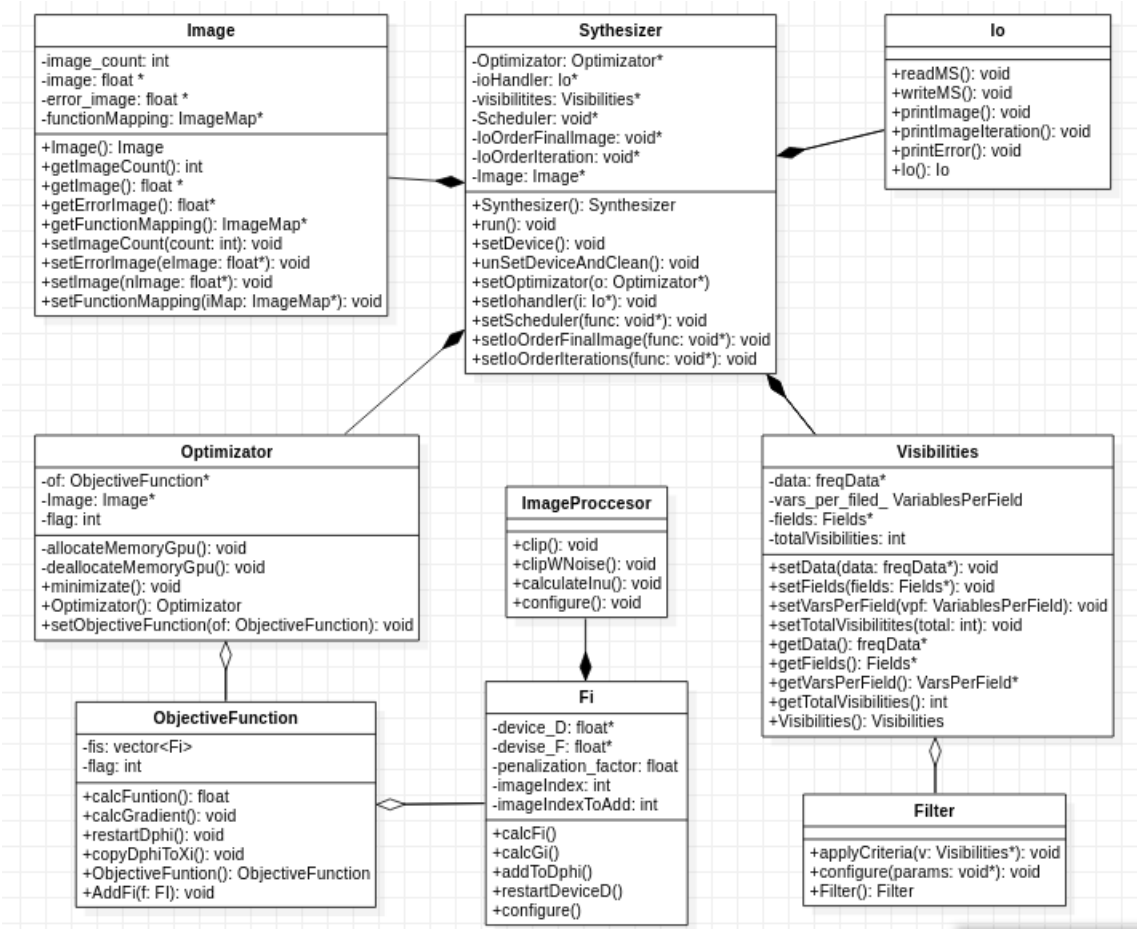

Figura 3.10: Diagrama de clases del framework.
Fuente: Elaboración propia, 2018.

3.3.2 Diagrama de secuencia General

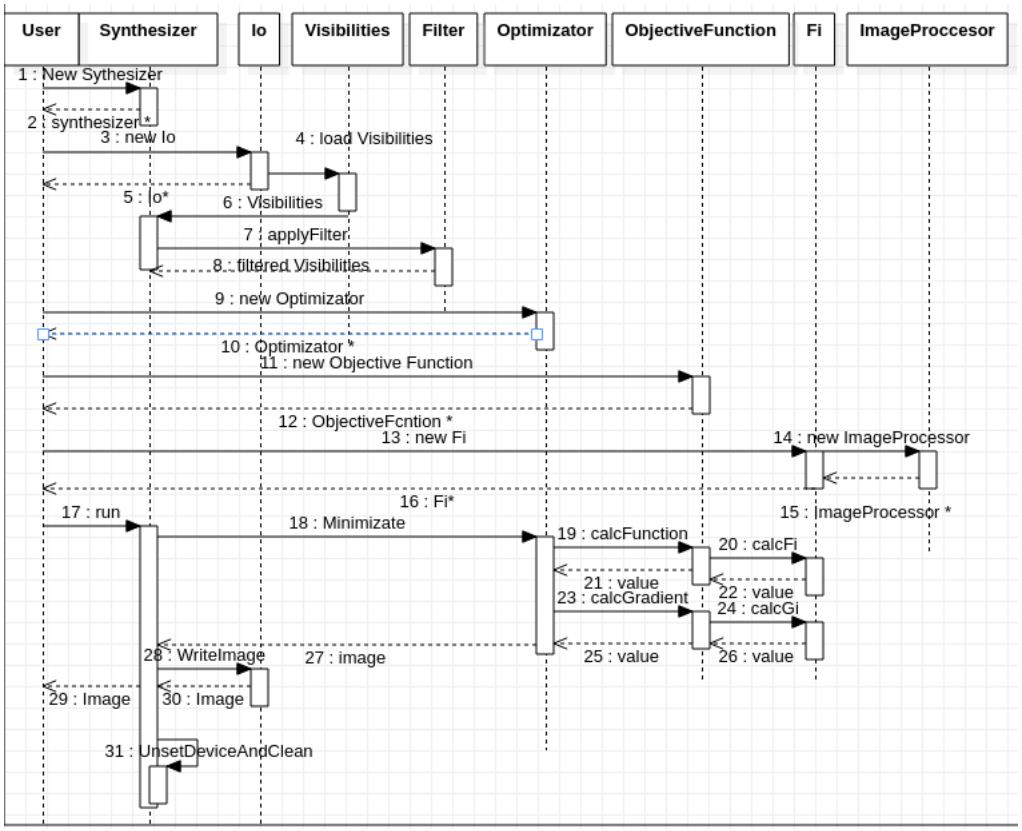

Figura 3.11: Diagrama de secuencia del framework.
Fuente: Elaboración propia, 2018.

3.3.3 Diagrama de arquitectura del framework

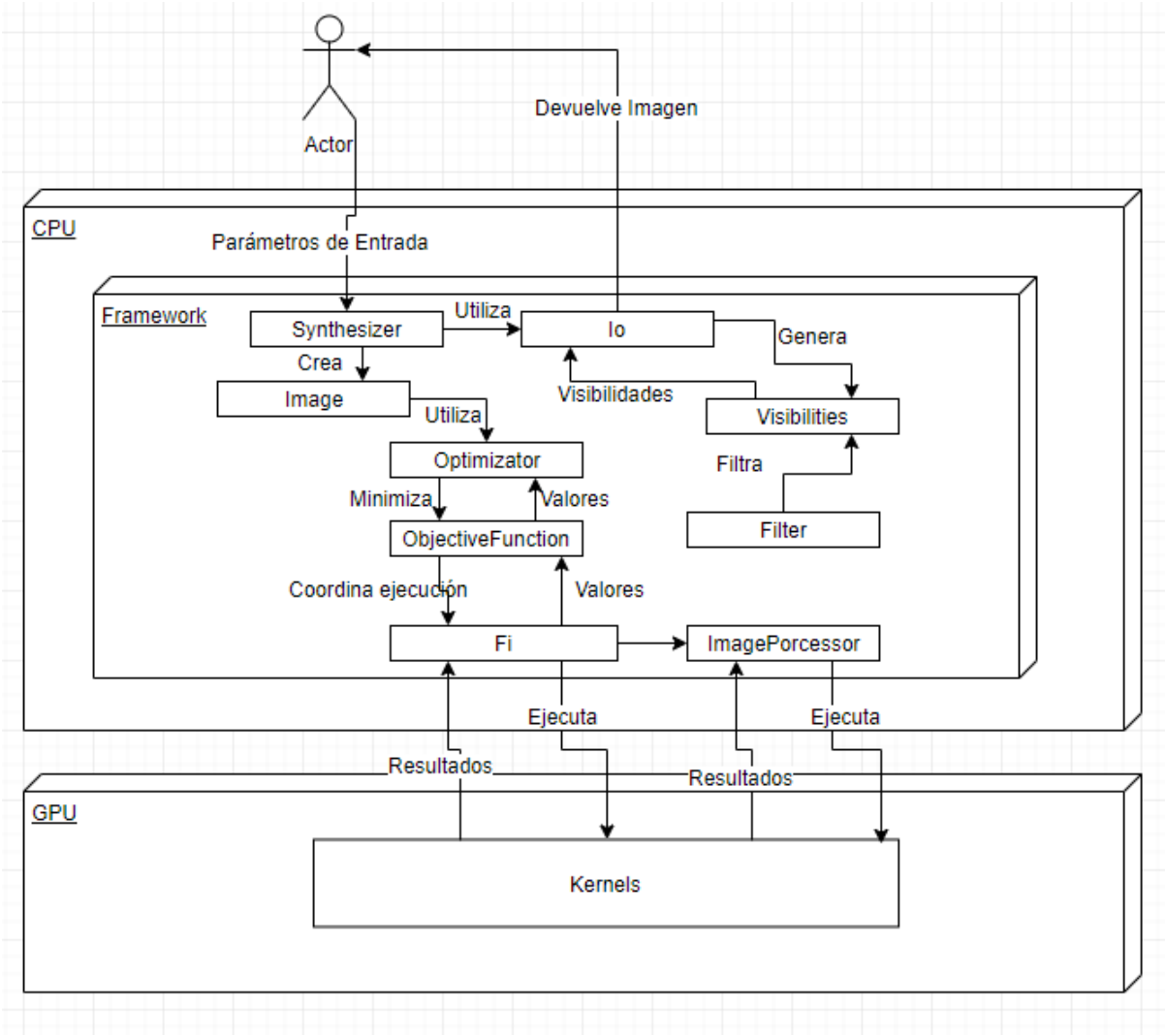

Figura 3.12: Diagrama de arquitectura del framework.
Fuente: Elaboración propia, 2018.

CAPÍTULO 4. IMPLEMENTACIÓN DE LA SOLUCIÓN

La implementación de la solución tiene 2 partes. Primero, la implementación del patrón de diseño creacional para las clases base definidas en el capítulo anterior junto con las clases necesarias para esto. Para luego, mostrar la implementación detallada de estas clases junto a sus especializaciones.

4.1 IMPLEMENTACIÓN DEL PATRÓN DE DISEÑO *FACTORY METHOD*

Para esta implementación se decide en el uso del patrón de diseño creacional *Factory Method* en todas las clases. Este patrón de diseño consiste en centralizar la construcción de los objetos de un subtipo de una clase determinada en una clase constructora. De esta manera se oculta al usuario la diversidad de casos particulares que se puede prever, para elegir el subtipo a crear. Parte del principio de que las subclases determinan la clase a implementar. A continuación se muestra un ejemplo de este patrón.Esto requiere primeramente implementar un patrón de diseño *Singleton* (Alexandrescu (2001)) que pueda instanciar a cada una de las fabricas de objetos. El patrón de diseño *Singleton* garantiza la existencia de una única instancia para una clase y la creación de un mecanismo global para el acceso de ella. Restringe la instanciación de una clase o valor de un tipo a un solo objeto. En el código mostrado en el algoritmo 4.1 se ve que la clase *Singleton* es un *template*, por lo que puede instanciar cualquier clase.

De esta manera se proceden a generar distintas clases *factory*, las cuales, en tiempo de ejecución son instanciadas mediante el método **Instance()** de la clase *Singleton*. De esta forma se asegura que existe sólo una instancia en memoria de cada una de las clases *factory*. En el algoritmo 4.2 se muestra el código generalizado para la implementación de la clase *factory*.

En donde ***Object**** es la clase a instanciar y para dar más claridad al código, se reemplaza la palabra *Object* con el nombre de cada clase. En el algoritmo 4.3 se muestra la implementación de un *factory* particular para la clase *Synthesizer* donde se realiza el cambio mencionado.

Así, cada especialización creada por el programador debe ser registrada

Algoritmo 4.1: Código clase Singleton

```
1
template<class T>
class Singleton
{
public:
static T& Instance()
{
        static T instance;
        return instance;
}
private:
Singleton(){
};
Singleton(T const&)          = delete;
void operator=(T const&) = delete;
};
```

Algoritmo 4.2: Código clase ObjectFactory.

```
1
class ObjectFactory
{
public:
  typedef: Object* (*CreateObjectCallback)();
private:
  typedef std::map<int. CreateObjectCallback> CallbackMap;
public:
  bool RegisterObject(int ObjectId, CreateObjectCallback CreateFn){
    return callbacks_.insert(CallbackMap::value_type(ObjectId, createFn)).
        second;
  }

  bool UnregisterObject(int ObjectId){
    return callbacks_.erase(ObjectId) == 1;
  }
  Object* CreateObject(int ObjectId)
  {
          CallbackMap::const_iterator i = callbacks_.find(ObjectId);
          if (i == callbacks_.end())
          {
                  // not found
                  throw std::runtime_error("Unknown Object ID");
          }
          // Invoke the creation function
          return (i->second)();
  };
private:
  CallbackMap callbacks_;
}
```

en su correspondiente *factory* para poder ser instanciada. Esto se realiza mediante la creación de un *namespace* privado en donde se registra la especialización de la clase en el *factory* correspondiente a su clase padre. En el algoritmo 4.4, se muestra la rutina de

Algoritmo 4.3: Código clase SynthesizerFactory.

```
1
class SynthesizerFactory
{
public:
typedef Synthesizer* (*CreateSynthesizerCallback)();
private:
typedef map<int, CreateSynthesizerCallback> CallbackMap;
public:
// Returns true if registration was succesfull
bool RegisterSynthesizer(int SynthesizerId, CreateSynthesizerCallback CreateFn)
{
        return callbacks_.insert(CallbackMap::value_type(SynthesizerId,
            CreateFn)).second;
};

bool UnregisterSynthesizer(int SynthesizerId)
{
        return callbacks_.erase(SynthesizerId) == 1;
};

Synthesizer* CreateSynthesizer(int SynthesizerId)
{
        CallbackMap::const_iterator i = callbacks_.find(SynthesizerId);
        if (i == callbacks_.end())
        {
                // not found
                throw std::runtime_error("Unknown Synthesizer ID");
        }
        // Invoke the creation function
        return (i->second)();
};

private:
CallbackMap callbacks_;
};
```

registro para un clase especializada en la clase *factory* correspondiente a su clase padre, usando los nombres *especializedObject* y *Object* respectivamente.

Algoritmo 4.4: Código de registro para una clase especializada.

```
1
namespace {
Object* CreateEspecializedObject()
{
        return new especializedObject;
}
const int ObjectId = 0;
const bool Registered = Singleton<ObjectFactory>::Instance().RegisterObject(
    ObjectId, CreateEspecializedObject);
};
```

4.2 IMPLEMENTACIÓN DE LAS CLASES Y SUS ESPECIALIZACIONES

Como segunda parte de la implementación, se tiene la implementación tanto de la clases diseñadas en el capítulo anterior como las especializaciones de estas. Estas son ordenadas según el nivel de valor que entregan a las funcionalidades del *framework*. De esta manera las implementaciones se muestran el orden: *Fi, ImageProcessor, ObjectiveFunction, Optimizator, Synthesizer, Image, Io, Visibilities, Filter*.

4.2.1 Implementación de la clase Fi

La clase *Fi* comprende la ejecución de la mayor cantidad de *Kernels* de GPU, esto hace una de las clases más complejas de implementar ya que CUDA no está pensado para ser utilizado en el paradigma orientado al objeto. Es por esto que junto a la implementación de la clase *Fi*, se propone un estándar de nombramiento para funciones intermedias encargadas de llamar a los *kernels* específicos de cada especialización de *Fi*.

Este estándar consiste en agregar el prefijo *link* a la función intermedia que llamada a los *Kernels*, por ejemplo, para invocar al *kernel* **chi2Kernel**, se crea una función intermedia que invoca a dicho *kernel*, la cual es nombrada *linkChi2Kernel*. Para efectos de este trabajo de título se implementan cinco *Fi*. Estos *Fi* corresponden a los posibles elementos que el programa GPUVMEM utilizaba como miembros de su función objetivo. Para realizar la implementación de los elementos de la función objetivo presentes en el programa GPUVMEM, se crean las clases especializadas mostradas desde la Figura 4.1 hasta la Figura 4.5.

Estas especializaciones están basadas en las siguientes ecuaciones:

- Clase Chi2: $\chi^2 = \sum \frac{\left|V_j^m - V_j^o\right|^2}{\sigma_j^2}$
- Clase Entropy: $S = \sum_i I_i log \frac{I_i}{M}$
- Clase Laplacian: $L = \sum_{i,j}^{N-1}((I_{k+1,j} - 2I_{k,j} + I_{k-1,j}) + (I_{k,j+1} - 2I_{k,j} + I_{k,j} - 1))^2$
- Clase QuadraticPenalization: $QP = \frac{\sum_{i,j}(I_{i+1,j}-I_{i,j})^2+(I_{i-1,j}-I_{i,j})^2+(I_{i,j+1}-I_{i,j})^2+(I_{i,j-1}-I_{i,j})^2}{2}$

- Clase TotalVariation: $TV = \sqrt{\sum_{i,j}(I_{i+1,j} - I_{i,j})^2 + (I_{i,j+1} - I_{i,j})^2}$

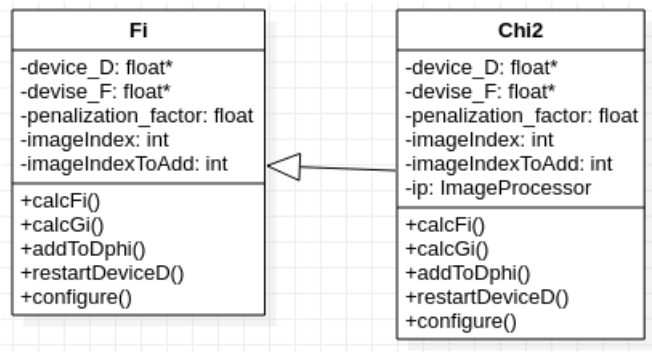

Figura 4.1: Implementación de la clase Chi2.
Fuente: Elaboración propia, 2018.

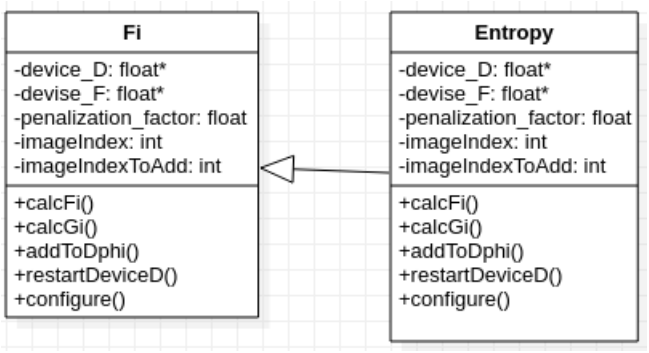

Figura 4.2: Implementación de la clase Entropy.
Fuente: Elaboración propia, 2018.

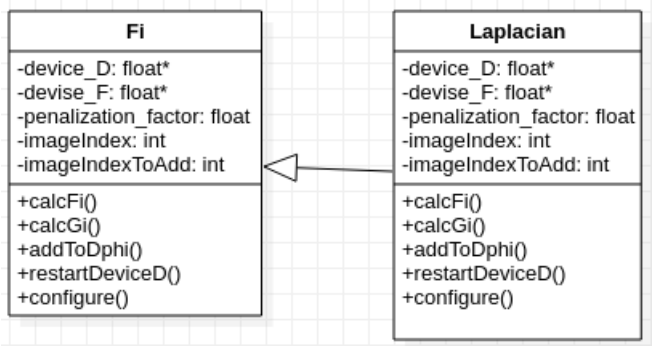

Figura 4.3: Implementación de la clase Laplacian.
Fuente: Elaboración propia, 2018.

Todas las implementaciones de *Fi* mostradas anteriormente tienen estructura similar debido a que en general para las implementaciones heredadas de GPUVMEM solamente es necesario modificar la llamada al *kernel* creando una función intermedia.

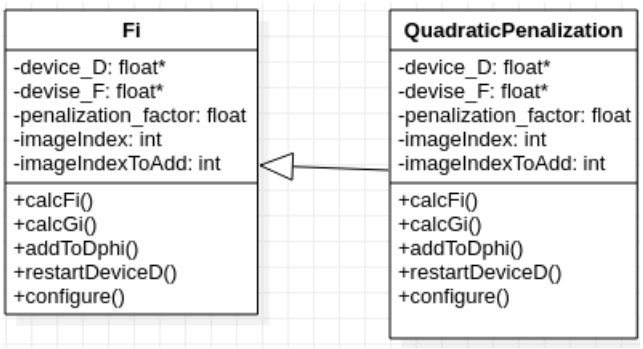

Figura 4.4: Implementación de la clase QuadraticPenalization.
Fuente: Elaboración propia, 2018.

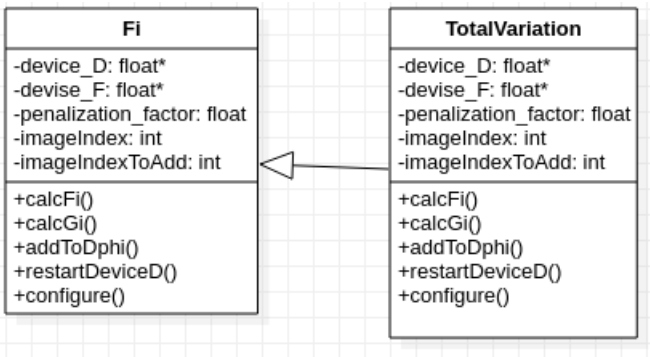

Figura 4.5: Implementación de la clase TotalVariation.
Fuente: Elaboración propia, 2018.

Debido a esta implementación se ha definido una convención de programación para *kernels*. Esta indica que para programar un nuevo *kernel*, siempre debe tenerse en consideración que los *Fi* pueden aplicarse a una imagen en cualquier índice. De esta manera un *kernel* implementado bajo esta convención se muestra en el algoritmo 4.5, en donde el *int image* representa el índice de la imagen y los *long* **M** y **N** representan las dimensiones de cada imagen (recordar que las imágenes están almacenadas en un arreglo unidimensional y todas las imágenes tienen las mismas dimensiones.)

Si bien las implementaciones de la mayoría de los *Fi* se ve simple, no necesariamente lo es para los futuros *Fi*. Un ejemplo de cómo la complejidad de un *Fi* difiere para cada implementación, es el caso de clase *Chi2* (Figura 4.1). En esta clase es necesaria la adición de otra clase para realizar ciertas tareas.

Las tareas mencionadas anteriormente que requieren otra clase para ser atendidas, están fuertemente acopladas en el *software* GPUVMEM, particularmente con la clase *Chi2*. Que estas tareas estén acopladas con la clase *Chi2* no implica que sólo

Algoritmo 4.5: Código de kernel para el cálculo de la máxima entropía implementado bajo la convención de índices.

```
1
__global__ void SVector(float *S, float *noise, float *I, long N, long M, float
    noise_cut, float MINPIX, float eta, int image)
{
        int j = threadIdx.x + blockDim.x * blockIdx.x;
        int i = threadIdx.y + blockDim.y * blockIdx.y;

        float entropy = 0.0;
        if(noise[N*i+j] <= noise_cut) {
                entropy = I[N*M*image+N*i+j] * logf((I[N*M*image+N*i+j] /
                    MINPIX) + (eta + 1.0));
        }

        S[N*i+j] = entropy;
}
```

son aplicables a dicha clase incluso es más, estás tareas podrían ser realizadas *n* veces durante la ejecución de cada uno de los *Fi* que la clase *ObjectiveFunction* contenga. Esta clase representa el grado máximo de flexibilidad dentro del *framework*, ya que cambiar un *Fi* tiene un impacto sustantivo en los resultados de las ejecuciones. Además, el *software* GPUVMEM sólo provee de la posibilidad de utilizar 1 *Fi* además de *Chi2*, de esta manera el hecho de poder agregar más *Fi* a la función objetivo, es lo que más impacto tiene sobre las funcionalidades del *framework*.

4.2.2 Implementación de la clase ImageProccesor

La especialización de esta clase está fuertemente ligada a la clase *Fi*, ya que define ciertas tareas necesarias de tratamiento de imágenes específicas para los cálculos particulares de algunas de las especializaciones de la clase *Fi*.

Las tareas de tratamiento de imágenes son por lo general llamadas a *kernels* que realizan alguna operación sobre la imagen. Estos *kernels* son dependientes del tipo de sintetizador, aunque de todas maneras apuntan a resolver tareas definidas y comunes en la síntesis de imágenes en interferometría. Especializaciones de la clase *ImageProcessor* pueden eventualmente agregar otro tipo de funcionalidades a la ejecución de cada especialización de *Fi*.

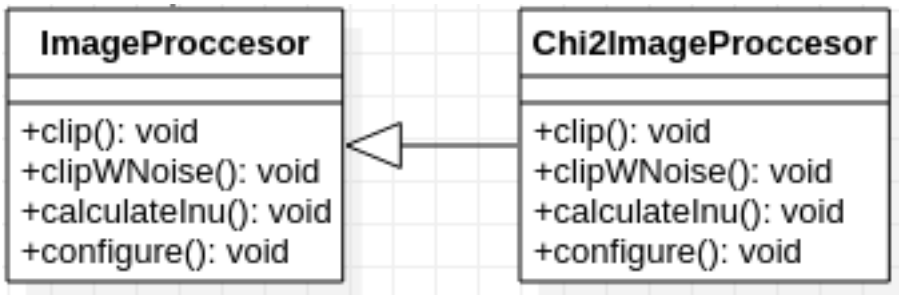

Figura 4.6: Implementación de la clase Chi2ImageProccesor.
Fuente: Elaboración propia, 2018.

4.2.3 Implementación de la clase ObjectiveFunction

La clase *ObjectiveFunction* tiene un comportamiento híbrido entre coordinador de ejecuciones y contenedor, es por esto que no se realiza un implementación de la interfaz. Su estructura está definida por:

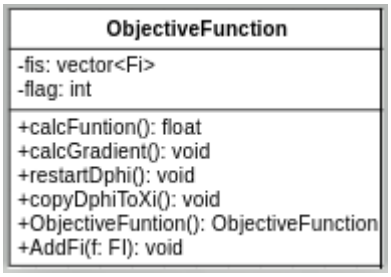

Figura 4.7: Implementación de la clase ObjectiveFunction.
Fuente: Elaboración propia, 2018.

Esta implementación coordina la ejecución de los distintos *Fi*, recoge los valores de estas ejecuciones y reporta estos resultados al *Optimizator* para realizar la minimización correspondiente de la imagen. Las implementaciones de los métodos más importantes de esta clase son mostrados, de forma simplificada, a continuación:

Algoritmo 4.6: Método calcFunction().

```
1
1 void calcfunction(){
2     float value = 0.0;
3     for(vector<Fi*>::iterator it = fis.begin(); it != fis.end(); it++)
4         value += (*it)->calcFi(p);
5     return value;
6 }
```

Algoritmo 4.7: Método calcGradient().

```
1
1 void calcGradient(){
2     restartDPhi();
3     for(vector<Fi*>::iterator it = fis.begin(); it != fis.end(); it++){
4         (*it)->calcGi(p, xi);
5         (*it)->addToDphi(dphi);
6     }
7     copyDphiToXi(xi);
8 }
```

4.2.4 Implementación de la clase Optimizator

La clase *Optimizator* implementa el método de optimización que posee nuestro *Synthesizer*. Esta tarea es de suma importancia ya que en ella se centra la lógica iterativa de los algoritmos de síntesis de imágenes en interferometría. De esta manera la especialización de esta interfaz está dada por la implementación del gradiente conjugado (*Conjugate Gradient* por su nombre en inglés). Siendo esta especialización de la siguiente forma:

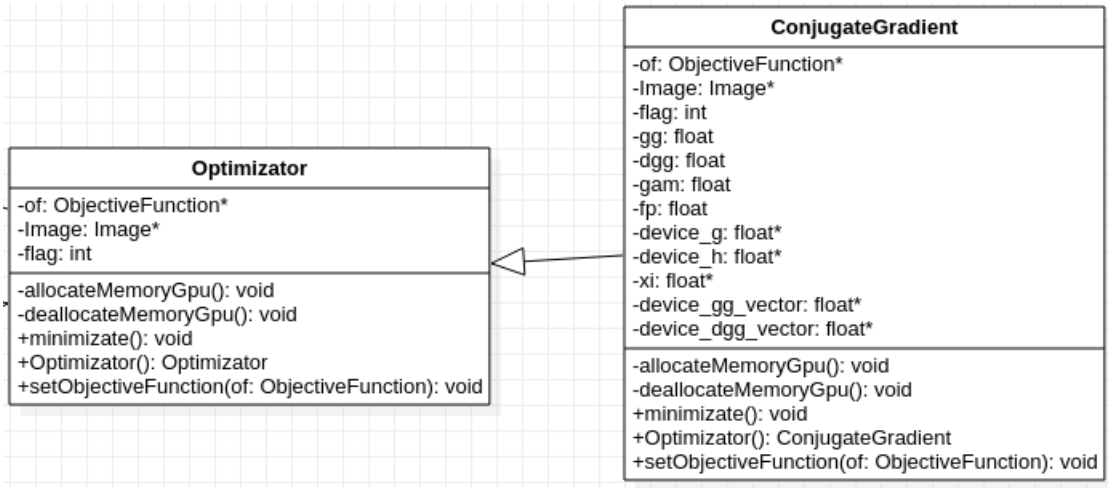

Figura 4.8: Implementación de la Interfaz Optimizator.
Fuente: Elaboración propia, 2018.

Esta implementación es la misma que la del *software GPUVMEM* y está basada en el método de *Fletcher-Reeves-Polak-Ribiere* (Press et al. (2007)).

4.2.5 Implementación de la clase Synthesizer

La clase *Synthesizer* implementa la lógica de la coordinación de la ejecución de los componentes del sistema, puesto que define las interacciones entre los datos en memoria de GPU y los métodos de los componentes del sistema. Dentro de esta clase es donde se incorpora la mayor cantidad de lógica del código base. De esta manera se implementa un *Synthesizer* MFS (*multi-frequency synthesis*).

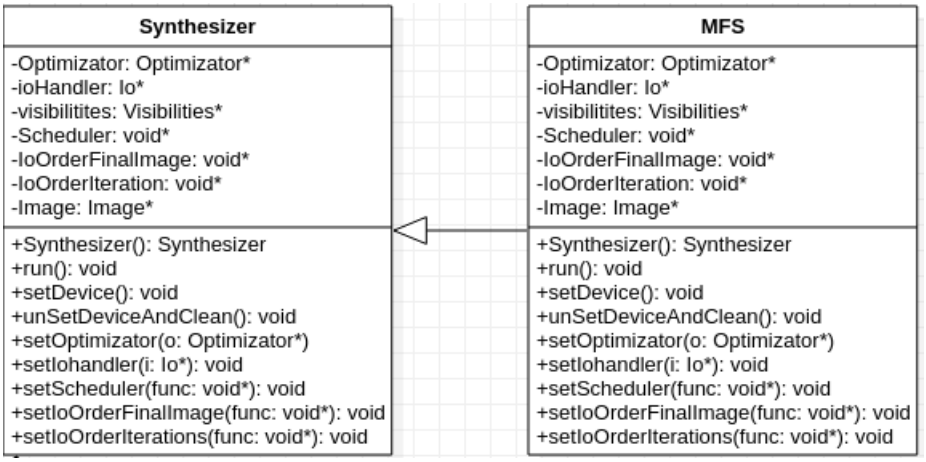

Figura 4.9: Especialización de la clase Synthesizer.
Fuente: Elaboración propia, 2018.

A partir de esta interfaz (Figura 4.9) se ha implementado el primer elemento que compone el *framework*, en otras palabras su coordinador. A continuación se presentan los algoritmos de manera resumida para los métodos más importantes de esta especialización.

En el algoritmo 4.9 se presenta el flujo normal de configuración de un sintetizador, en donde al instanciar la clase *Io*, esta concentra la lógica de la carga de datos del sistema. De esta manera el programa empieza a manejar los datos recibidos y coordina la creación de los objetos mínimos para su funcionamiento. Se debe tener en cuenta que existe una limitante de implementación en CUDA, la cual no permite utilizar *kernel* cómo métodos. Esto se traduce en que dentro del método **configure()** existen llamadas directas a *kernels* para configurar las variables necesarias.

En Algoritmo 4.10 es donde la ejecución de los primeros *Kernels* ocurre. Estos *Kernels* ejecutan para las tareas de pre-procesamiento particulares de un

Algoritmo 4.8: Método configure().

Entrada: int argc, char* argv.

Salida: void.

```
1: if Leer entradas del programa then
2:     Configurar variables necesarias
3:     if iohandler == NULL then
4:         iohandler = default
5:     end if
6:     Crear Visibilidades
7: else
8:     return Exit(-1)
9: end if
10: return
```

Algoritmo 4.9: Método setDevice().

Entrada:

Salida: void.

```
1: if Pre-procesamiento de datos then
2:     Crear imagen inicial
3:     if Envío de datos a memoria GPU then
4:         return
5:     end if
6: else
7:     return Exit(-1)
8: end if
9: return
```

sintetizador *MFS*. Es aquí donde el tiempo de ejecución empieza a contar con propósito de evaluar el rendimiento del sistema de *software*.

En Algoritmo 4.11, se ven las primeras tareas de coordinación de ejecución del sistema. Es aquí donde el sistema comienza su ejecución más costosa. Debido a que el orden de minimización es dependiente a la cantidad de imágenes y el objeto del cielo a estudiar, se la posibilidad de modificar este orden mediante el atributo *scheduler*

Algoritmo 4.10: Algoritmo método run().

Entrada:

Salida: void.

```
1: if optimizator != NULL then
2:    if scheduler == NULL then
3:       Cargar Minimización por defecto
4:       optimizator->minimizate()
5:       iohandler->printImage()
6:    else
7:       Utilizar scheduler de minimización
8:    end if
9: else
10:   return Exit(-1)
11: end if
12: return
```

Algoritmo 4.11: Algoritmo método unSetDevice().

Entrada:

Salida: void.

```
1: Limpiar memoria GPU y CPU
2: Destruir Objetos
3: return
```

4.2.6 Implementación de la clase Image

Como se mencionaba anteriormente la clase *Image* como contenedora de datos no requiere un modelo escalable, por lo cual ,no ha sido diseñada un interfaz para su diversificación. Como se ha visto anteriormente, su estructura está compuesta por la siguiente clase:

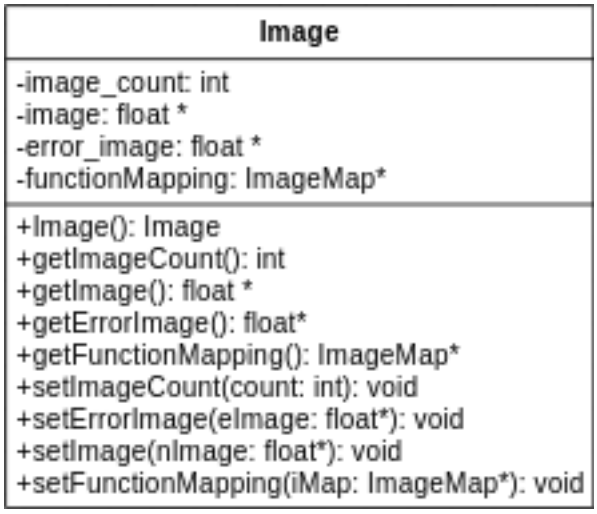

Figura 4.10: Implementación de la clase Image.
Fuente: Elaboración propia, 2018.

4.2.7 Implementación de la clase IO

La implementación de la clase *IO* parte de la necesidad de generar un modelo en el cual se permita leer información de distinta manera y almacenarlos en un formato que el *framework* logre manejar. Debido a lo mencionado anteriormente es que se ha realizado una abstracción de esta funcionalidad y se ha especializado el objeto de manera que la lectura de archivos no influencie al flujo normal de información en la plataforma.

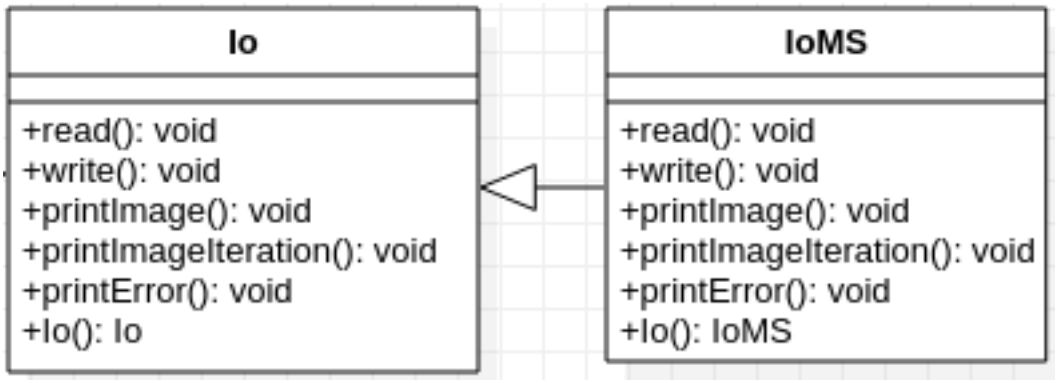

Figura 4.11: Implementación de la interfaz IOMS.
Fuente: Elaboración propia, 2018.

Cabe destacar que el formato estándar de almacenamiento de archivos astronómicos es *MS*, por lo que la implementación de los métodos de esta especialización se basan estrechamente en la implementación base *GPUVMEM*.

4.2.8 Implementación de la clase Visibilities

Al igual que la clase anterior, la clase *Visibilities* también se comporta como un contenedor de datos, por lo cual no se ha desarrollado una interfaz para la especialización de la Clase. Como se ha visto anteriormente su estructura está dada por la siguiente clase:

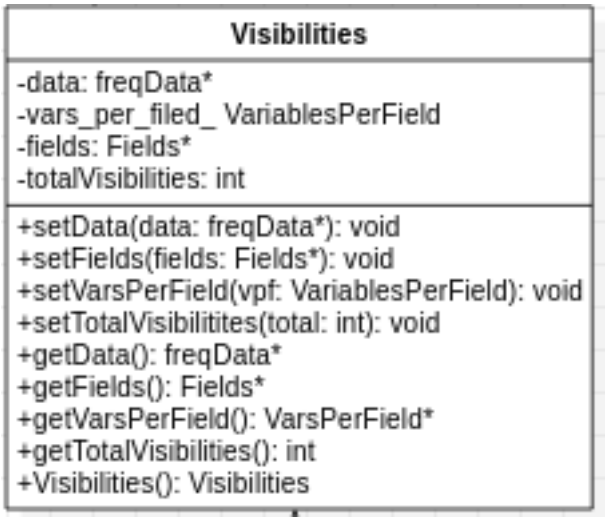

Figura 4.12: Implementación de la clase Visibilities.
Fuente: Elaboración propia, 2018.

4.2.9 Implementación de la clase Filter

La clase *Filter* es la encargada del pre-procesamiento de visibilidades. Es implementada bajo el patrón de diseño *Filter*, en el cual, los filtros heredan de una clase padre *Filter* e implementan su método *applyCriteria*. Este método es donde se concentra toda la lógica de pre-procesamiento de las visibilidades. Como implementación particular, tal y como se muestra en la Figura 4.13, se tiene la clase *gridding*. La clase *gridding* transporta las visibilidades a una grilla regular y es el único filtro presente en el *software* GPUVMEM.

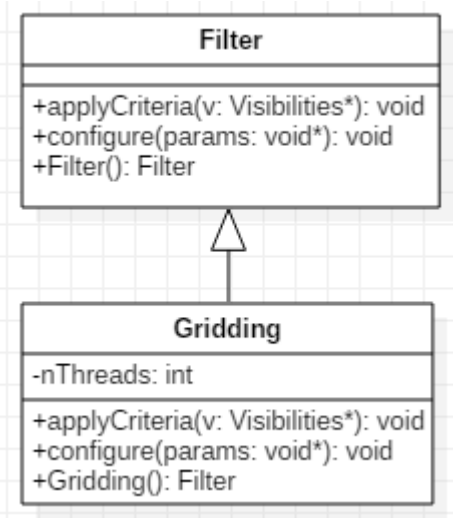

Figura 4.13: Implementación de la clase Filter.
Fuente: Elaboración propia, 2018.

CAPÍTULO 5. EXPERIMENTOS

En este capítulo se exponen los resultados obtenidos a partir de 3 casos de prueba realizados mediante el uso del *framework* diseñado e implementado, los cuales son presentados a continuación. El objetivo es comprobar la correctitud y eficiencia computacional de la implementación.La primera evaluación consiste en la comparativa en cuanto a similitud de imágenes y rendimiento computacional de la plataforma en comparación a su versión base. Una segunda evaluación que consiste en estimar la conformidad y facilidad de utilización de la herramienta desarrollada. Esto lo hace un desarrollador externo, quien configurará variantes del sintetizador GPUVMEM.

5.1 MATERIALES Y MÉTODOS

Para realizar el primer conjunto de pruebas se usa la implementación de GPUVMEM en base al *framework* desarrollado y la implementación base de GPUVMEM.

Se hace uso de distintos conjuntos de datos, de tamaño variable, provistos con el *software* GPUVMEM a modo de *SelfTest* además, se utiliza un conjunto de datos de mayor tamaño y cantidad de frecuencias. Este último, al día de la escritura de este documento, es usado como caso de estudio en la comunidad de Astro informática del Departamento de Ingeniería en Informática de la Universidad de Santiago de Chile.

A excepción del conjunto de datos de mayor tamaño, todas las pruebas son realizadas en el computador personal del estudiante. El conjunto de datos de mayor tamaño es procesado en el Computador del Laboratorio de Operaciones del Departamento de Ingeniería en Informática de la Universidad de Santiago de Chile. Este computador lleva por nombre *forker2* y es utilizado frecuentemente para realizar pruebas de las implementaciones y mejoras de la plataforma GPUVMEM. Las características de este computador son detalladas en la tabla 5.1.

Como última evaluación se realiza un cuestionario basado en la ISO 17.000 para evaluar la conformidad con la herramienta desarrollada. Este cuestionario es respondido por un desarrollador externo quien implementa variantes del sintetizador GPUVMEM. El desarrollador designado para esta prueba conoce la plataforma GPUVMEM y

regularmente la utiliza para realizar reconstrucciones de imágenes. Además, se realiza una medición de los tiempo que el desarrollador externo demora en realizar estas nuevas implementaciones.

La tabla 5.1 muestra en detalle las características del computador utilizado.

Tabla 5.1: Características de *forker2*

Parámetro	Valor
Nombre Equipo	forker2
Tipo de Procesador	AMD Athlon(tm) II X4 630
Número de Procesadores	1
Frecuencia del Procesador	2.8GHz
Núcleos del Procesador	4
Tamaño de la cache	2MB
Memoria RAM	16GB
Tipo de GPU	Nvidia Quadro M4000
Número de GPUs	2
Memoria VRAM Total	16GB

Fuente: Elaboración Propia, 2018

Los experimentos del primer conjunto de pruebas realizadas corresponden a la ejecución de ambas plataformas con una configuración equivalente para los distintos conjuntos de datos.

La configuración escogida contempla las clases especializadas implementadas en el capitulo anterior. Es importante recordar que GPUVMEM sólo permite el cálculo de una función objetivo utilizando los *kernels* de **Chi cuadrado** y **máxima entropía** (Cárcamo et al. (2018)). De esta manera la configuración definida para el *framework* es la de un sintetizador de tipo *MFS*, el cual, realiza los cálculos con una imagen para los conjuntos de datos pequeños y con 2 imágenes para el conjunto de datos de mayor envergadura. La recolección de datos será mediante la clase *IOMS* (carga archivos de tipo MS y como salida entrega archivos de tipo *FITS*). Finalmente un *Optimizator* de tipo gradiente conjugado que minimiza una clase *ObjectiveFunction* configurada con los *Fi Chi2* y *Entropy* que corresponden a los *kernels* de **Chi cuadrado** y **máxima entropía** respectivamente. Esta configuración es equivalente a la configuración por defecto del *software* GPUVMEM. Para ambas configuraciones se utilizan los mismos valores de entrada.

El peso en disco de los conjuntos de datos utilizados en el desarrollo de las pruebas son los que se muestran en la tabla 5.2:

Tabla 5.2: Conjuntos de datos provistos por ALMA

Conjunto de mediciones	Peso en disco
antennae	$9{,}4$ MB
co65	63 MB
FREQ87	9.9 MB
selfcalband9	10 MB
HlTauB6	1.6 GB

Fuente: Elaboración Propia, 2018

Estos conjuntos se ejecutan 10 veces cada uno a excepción de *HLTauB6*, el cual, se ejecuta sólo 5 veces. Se usa el promedio de tiempo entre las ejecuciones como resultado de las pruebas para cada conjunto de datos.

Además se guardan las imágenes generadas por ambos programas, se comparan visualmente y se comparan a nivel de parámetros. Para la comparación a nivel de parámetros se debe definir una unidad comparativa debido a que las imágenes pueden diferir según la cantidad de operaciones de punto flotante involucradas en el cálculo de cada imagen, particularmente el caso de *HlTauB6*, el cual, al ser un conjunto de datos que debe calcularse con más de una imagen involucra más operaciones de punto flotante. Para efectos de esta comparación se define como criterio de comparación el Error Cuadrático Medio normalizado (ECMn), el cual está definido por las ecuaciones 5.1 y 5.2:

$$ECM = \sqrt{\frac{[\sum_{k=1}^{N}(x_k - X_k)^2]}{N}} \tag{5.1}$$

$$ECMn = \frac{ECM}{max_x - min_x} \tag{5.2}$$

En donde:

- N es la cantidad de muestras recolectadas.
- *x* es el vector de parámetros de la imagen generada por el *framework*.
- *X* es el vector de parámetros de la imagen generada con GPUVMEM.

Por otra parte, el segundo experimento consiste en evaluar la conformidad con la herramienta desarrollada. Esto implica que un desarrollador externo propone

e implementa diferentes configuraciones para la prueba de funcionalidades de la plataforma. Las configuraciones realizadas por el desarrollador son las mencionadas en la tabla 5.3.

Tabla 5.3: Configuraciones generadas por el desarrollador externo

Objeto	Modificación
ObjectiveFunction	Se Agrega más de un Fi
IO	Se implementa una clase IO que no utiliza CASACORE
Sythesizer	Eliminación de la opción MultiGPU
Fi	Implementación de un Fi de prueba
Optimizator	Modificación de la implementación del gradiente conjudado

Fuente: Elaboración Propia, 2018

5.2 RESULTADOS

5.2.1 Resultados Tiempo de ejecución

La tabla 5.4 despliega los tiempos de ejecución para todos los casos considerados y contrasta esta información con los tiempos de ejecución de GPUVMEM. Además, en la tabla 5.5 se muestra el porcentaje de pérdida de rendimiento para conjunto de datos. La diferencia de rendimiento en todos los casos no es mayor al 1 % y es importante destacar que en algunas ocasiones el rendimiento del *framework* presenta un *Speedup* con respecto a la plataforma GPUVMEM. Según estos resultados la ejecución de tareas en segundo plano puede afecta el rendimiento de ambas plataformas. En el caso de *HlTauB6* esto no ocurre, ya que al ser ejecutado en la segunda GPU del computador *forker2*, las tareas en segundo plano como el entorno gráfico no afectan el rendimiento de los sistemas de *software*. Sin embargo, el porcentaje de pérdida de rendimiento se mantiene constante para este conjunto de datos.

Se aprecia que los resultados no son dependientes del tamaño de los conjuntos de datos.Además se aprecia que la cantidad de instrucciones adicionales que deben se ejecutadas por el programa no son dependientes de la cantidad de iteraciones del programa, más bien son dependientes de las referencias a atributos de las clases

definidas.

Tabla 5.4: Tiempos de ejecución para los conjuntos de datos

Conjunto de datos	GPUVMEM	framework
co65	57.891 s	58.148 s
selfcalband9	79.059 s	86,17382 s
freq87	105.295 s	107.395 s
antennae	597.291 s	592.576 s
HlTauB6	6658.584 s	6671.931 s

Fuente: Elaboración Propia, 2018

Tabla 5.5: Porcentaje de pérdida de rendimiento

Conjunto de datos	% pérdida de rendimiento
co65	0.8
selfcalband9	0.9
freq87	0.2
antennae	0.09
HlTauB6	0.02

Fuente: Elaboración Propia, 2018

5.2.2 Resultados reconstrucción de imágenes

Las imágenes generadas para todos los conjuntos de datos, excepto el conjunto de datos *HlTauB6*, son idénticas tanto en la imagen visible como en los parámetros de las imágenes con excepción del conjunto de datos *HlTauB6*. Para este caso las imágenes generadas son distintas al igual que sus parámetros, esto se debe a que para el caso de conjuntos de datos de múltiples frecuencias existe la ejecución de un *Kernel* adicional. Este *Kernel* adicional contiene múltiples operaciones de punto flotante lo que hace que los resultados varíen entre cada ejecución incluso para la plataforma GPUVMEM. Las imágenes generadas se pueden ver en las figuras 5.1 y 5.2. Los resultados de parámetros se muestran en la tabla 5.6.

Tabla 5.6: Error cuadrático medio normalizado de los parámetros de las imágenes generadas por el framework

Conjunto de datos	ERMn
co65	0
selfcalband9	0
freq87	0
antennae	0
HlTauB6	$1{,}438 \times 10^{-3}$

Fuente: Elaboración Propia, 2018

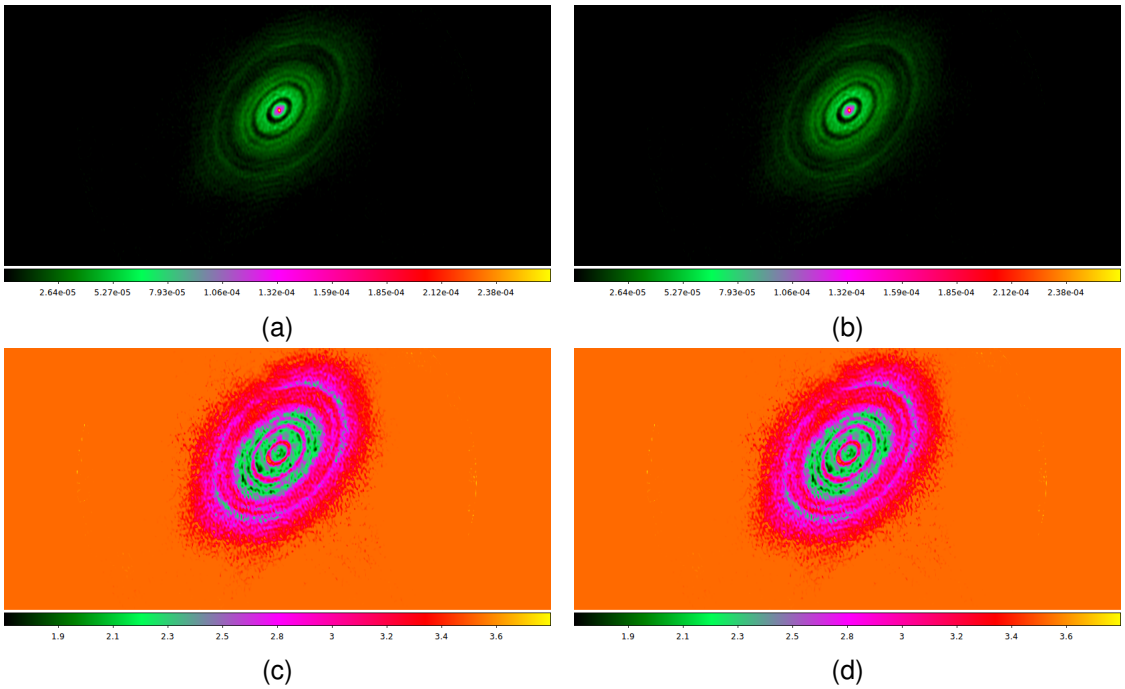

Figura 5.1: A la derecha las imágenes creadas con GPUVMEM y a las izquierda las creadas con el framework para HlTauB6

5.2.3 Resultados medición de conformidad

Los resultados de la medición de conformidad de la implementación de distintos componentes para el *framework* se ve en la tabla 5.7. En esta tabla se muestra la satisfacción con la herramienta en función de un coeficiente calculado al terminar de contestar el cuestionario. Para esto se dividen las preguntas en grupos de 5. Las respuestas de estas preguntas tienen un valor de 1 a 5 en función de la conformidad, siendo muy disconforme 1 y muy conforme 5. Se calcula el promedio de cada grupo de

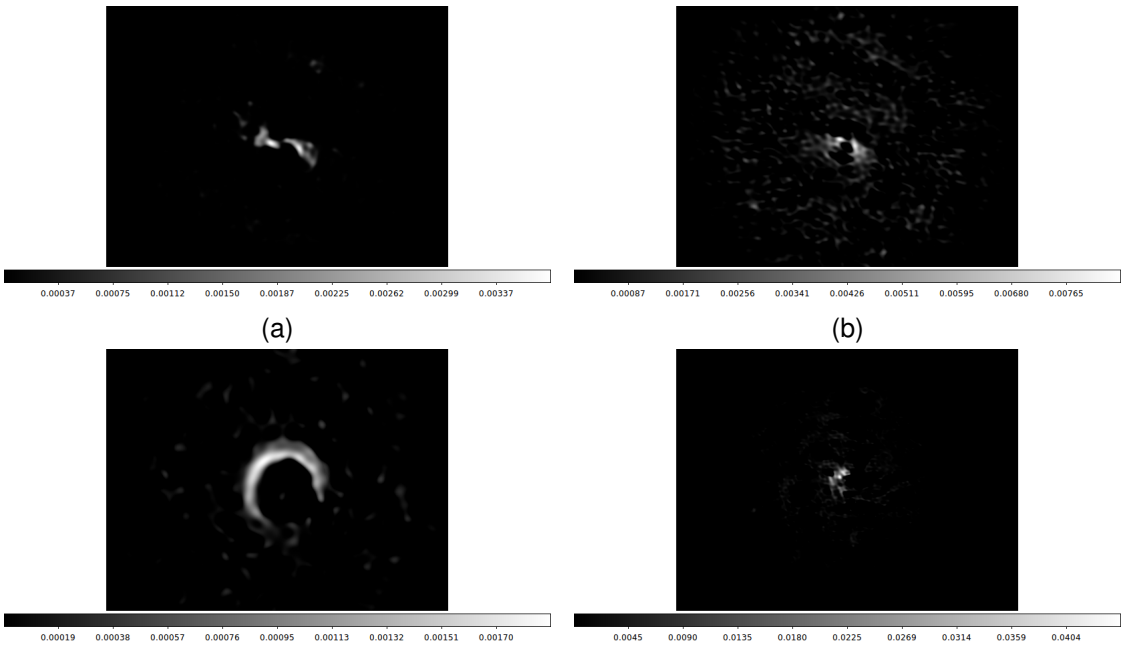

Figura 5.2: (a) imagen generada de co65. (b) imagen generada para freq87. (c) imagen generada para selfcalband9. (d) imagen generada para antennae.

preguntas, si cada uno de estos promedios es mayor a 4 se dice que el desarrollador está conforme.

Tabla 5.7: Configuraciones generadas por el desarrollador externo

Objeto	Modificación	Conforme
ObjectiveFunction	Se Agrega más de un Fi	Si
IO	Se implementa una clase IO que no utiliza CASACORE	Si
Sythesizer	Eliminación de la opción MultiGPU	Si
Fi	Implementación de un Fi de prueba	Si
Optimizator	Modificación de la implementación del gradiente conjugado	Si

Fuente: Elaboración Propia, 2018

Cada implementación fue probada con una ejecución del *framework* en el conjunto de datos *co65* para así evaluar la estabilidad de la plataforma. Las ejecuciones de la plataforma sólo son evaluados con el criterio, ejecuta o no ejecuta, ya que las variaciones implementadas son de prueba y no constituyen un trasfondo investigativo que respalde estos cambios o implementaciones. Estos resultados pueden verse en la tabla 5.8.

En la tabla 5.9 puede verse el tiempo que demoró el desarrollador en realizar

Tabla 5.8: Resultados de ejecución de las configuraciones

Objeto	Modificación	Ejecuta
ObjectiveFunction	Se Agrega más de un Fi	Si
IO	Se implementa una clase IO que no utiliza CASACORE	Si
Sythesizer	Eliminación de la opción MultiGPU	Si
Fi	Implementación de un Fi de prueba	Si
Optimizator	Modificación de la implementación del gradiente conjudado	Si

Fuente: Elaboración Propia, 2018

cada una de estas implementaciones, contrastado con el tiempo que le lleva realizar esto en *software* GPUVMEM. Es importante destacar que el desarrollador ha sido sometido a un día laboral completo de entrenamiento en el uso del *framework*.

Tabla 5.9: Tiempos de desarrollo del desarrollador externo

Modificación	GPUVMEM	framework
Se Agrega más de un Fi	1 hrs.	5 min.
Se implementa una clase IO que no utiliza CASACORE	1 hrs.	45 mins.
Eliminación de la opción MultiGPU	15 mins.	15 mins.
Implementación de un Fi de prueba	30 mins.	10 mins.
Modificación de la implementación del gradiente conjugado	30 mins.	20 mins.

Fuente: Elaboración Propia, 2018

Los resultados muestran que utilizando el *framework* el tiempo de desarrollo generalmente es menor al tiempo de desarrollo utilizando *software* base. Considerando el entrenamiento en el *framework* que tuvo el desarrollador la curva de aprendizaje para utilizar este *software* es pequeña.

CAPÍTULO 6. CONCLUSIONES

El presente trabajo de titulación ha permitido generar mediante la programación orientada a objetos, una aplicación del tipo *framework*. Esta aplicación tiene como finalidad de entregar las herramientas básicas necesarias para la creación y configuración de variantes de algoritmos de síntesis de imágenes en interferometría. El diseño de esta herramienta, se ha logrado mediante el análisis y modelado de un sintetizador usando la programación orienta a objetos. De esta manera, se han identificado (dentro de las limitantes de este trabajo) las clases y procesos más relevantes de la problemática de síntesis de imágenes.

Durante el diseño de la solución, la identificación de clases y comportamientos en proceso de síntesis de imágenes del *software* GPUVMEM han sido de gran relevancia para este trabajo. Esto debido a que separar las tareas que realizan cada una de las clases que interactúan dentro de la aplicación, se permite realizar un análisis específico de la lógica de funcionamiento y comunicación entre estas. De esta manera se facilitan las tareas de adición y modificación del funcionamiento interno de cada una de las clases sin que esto impacte en la definición de las clases ya diseñadas. Este proceso es el que ha permitido entregar modularidad en la solución implementada.

Por otra parte, es necesario mencionar que el rendimiento siempre ha sido un factor importante a considerar durante todo el el proceso de diseño de la solución. Como esta solución se basa en una plataforma de alto rendimiento, el resultado debe también debe ser una plataforma de alto rendimiento de características similares. Esto limita bastante las posibilidades del diseño de las clases, ya que muchas veces se tiene que simplificar una clase o simplemente no permitir mayores niveles de flexibilidad en estas. Esto tiene directa relación con la forma en la que CUDA implementa e interactúa con la programación orientada a objetos. Es por esto que muchas funciones que deberían ser métodos de una clase, deben ser implementados como métodos virtuales o incluso formar parte de los métodos de otra clase.

Es necesario destacar que trabajar con CUDA utilizando programación orientada a objetos, presenta diversas dificultades. Estas dificultades se ven reflejadas tanto al diseñar la aplicación como al implementarla. El simple hecho de utilizar CUDA implica pensar las aplicaciones de otra manera ya que, su arquitectura está diseñada en

base al modelo de ejecución SIMT (*Single Instruction, Multiple Threads*) (Nickolls et al. (2008)). Esto cambia completamente la sintaxis del lenguaje de programación C++11 a un nivel tal que los ciclos *for*, muchas veces dejen de existir para código iterativo. Si bien CUDA permite almacenar y trabajar con objetos dentro de la memoria de la GPU, estos sólo pueden ser utilizados como variables (como un *struct* en lenguaje de programación C)y no pueden tener llamadas a *kernels* como métodos.

Así mismo durante este trabajo de titulación se implementan funcionalidades adicionales para la plataforma GPUVMEM. Estas funcionalidades adicionales son parte importante del diseño de las clases, puesto que mientras estas se diseñan, se les agrega lógica adicional o se crean nuevas clases que permiten incorporar estas nuevas características.

De esta manera es como se ha logrado diseñar e implementar mediante la programación orientada a objetos un *framework* que permite la configuración de variantes de algoritmos de síntesis de imágenes en interferometría. Dentro de sus características están el diseño modular y encapsulamiento de las propiedades de cada una de las clases que componen la aplicación. Esto permite que la creación y configuración de variantes de algoritmos de síntesis de imágenes sea rápida sencilla y ordenada de acuerdo a un formato estándar. Además la jerarquización de las clases permite la incorporación de nuevas clases o la extensión y especialización de estas. Esto logra que la aplicación desarrollada sea flexible. Por otra parte, se ha logrado entregar flexibilidad a nivel de *kernels* mediante la implementación de convenciones de llamadas y de desarrollo de estos. Esto permite a la aplicación realizar tareas muy similares al *software* base, con perdidas mínimas de rendimiento.

A partir del *framework* desarrollado, se ha implementado un sintetizador de imágenes radio interferométricas basado en el *software* GPUVMEM. Esto permite reconstruir imágenes con los datos provistos por el observatorio ALMA. Las imágenes generadas mediante esta implementación son comparadas con las imágenes obtenidas al utilizar el *software* base de este trabajo de título. Estas comparaciones son a nivel visual y de diferencias de los parámetros que acompañan a las imágenes generadas. Estas diferencias son calculadas mediante los métodos de error cuadrático medio y error cuadrático medio normalizado entre los parámetros generados por ambos *softwares*. Además, a partir de los datos generados por las repetidas ejecuciones de los conjuntos

de prueba, se ha medido, registrado y comparado el rendimiento computacional de ambas implementaciones. De esta manera se he evaluado el porcentaje de pérdida de rendimiento asociado al *framework* desarrollado.

En sí, él rendimiento obtenido en la implementación del sintetizador GPUVMEM en su versión *framework* no ha disminuido significativamente. La pérdida de rendimiento existe y es más notoria en ejecuciones que realizan mayor cantidad de iteraciones. A pesar de los diversos beneficios que trae consigo la programación orientada a objetos, existe un factor en particular a considerar con respecto al rendimiento de las aplicaciones desarrolladas en este paradigma. Es el factor que corresponde a que la tasa de *Miss* en la memoria caché del procesador, es cerca de un 80 % a 100 % más alta en los programas orientados al objeto, tanto para las instrucciones como para los datos (Radhakrishnan et al. (1997)).

Este factor es uno de los principales problemas de rendimiento en las implementaciones basadas en la programación orientada a objetos y particularmente las desarrolladas en lenguaje de programación C++. La gran cantidad de *Miss* en la caché con respecto a las instrucciones, se debe a problemas de localidad temporal a causa de los llamados entre métodos de diferentes objetos. Por otra parte, otra gran cantidad de *Miss* en la caché con respecto a los datos, se atribuye al gran tamaño de pila almacenada para las estructuras de datos en C++. En general el problema del rendimiento de las aplicaciones orientadas a objetos en C++ se resume en una alta tasa de *Miss* en la caché, lo cual es posible que en futuras versiones del lenguaje de programación sean solventadas con mejoras a nivel de compilación o de *Hardware*. Con estás mejoras se espera que el costo/beneficio de añadir flexibilidad a el *software* GPUVMEM implique una menor pérdida de rendimiento y así en trabajos posteriores poder realizar mejores y más complejas implementaciones.

Considerando los resultados obtenidos durante los experimentos, se ha logrado cumplir con el objetivo principal de este trabajo de titulación, el cual, resumidamente consiste en desarrollar un *framework* orientado al objeto para síntesis de imágenes en interferometría. Este *software* entrega las herramientas necesarias para la configuración o creación de variantes de algoritmos para síntesis de imágenes y por otra parte, permite que las implementaciones realizadas sean extensibles y flexibles. Además, con los resultados obtenidos, se ha logrado cumplir el propósito de esta implementación. Entregando

una herramienta de alto rendimiento que permite la implementación robusta, rápida, extensible y reutilizable de creación o configuración de variantes de algoritmos para síntesis de imágenes. Esta aplicación es distribuida como la nueva versión oficial de GPUVMEM.

Si bien GPUVMEM no es una plataforma masivamente utilizada, la comunidad de astroinformática de la Universidad de Santiago de Chile ocupa esta herramienta para realizar reconstrucciones de imágenes a pedido de astrónomos colaboradores. El desarrollador externo mencionado en este documento, realiza esta tarea para los astrónomos de manera regular. Como se ve en los resultados, el tiempo de desarrollo ha disminuido y no deben realizarse *forks* del código fuente para modificar un algoritmo. Esto muestra que la herramienta es de utilidad para los astrónomos y cumple su propósito a cabalidad.

Finalmente cabe señalar que en la actualidad tanto el lenguaje de programación C++ como la plataforma CUDA siguen evolucionando y es posible como trabajo a futuro, mediante una actualización, este *framework* pueda permitir mayor flexibilidad con una menor perdida de rendimiento. Dada esta evolución, puede que sea posible implementar *Kernels* a nivel de atributos de objetos o permitir nuevos niveles de flexibilidad para los *kernels*. Además, se propone como trabajo a futuro, el desarrollo de un lenguaje de programación interpretado interno del *framework*, que permita la creación o configuración de *kernels* para los regularizadores de optimización (Fi) en tiempo de ejecución. Esto permitiría mayores posibilidades a la hora de definir una función objetivo junto con menor dificultad de implementación de estos.

GLOSARIO

- Interferometría: Es una familia de técnicas utilizadas que consiste en combinar ondas provenientes de diferentes receptores para obtener una imagen de mayor resolución.
- Framework: Es un conjunto estandarizado de conceptos, prácticas y criterios para enfrentar y resolver nuevos problemas de índole similar a la que originó este conjunto.
- Framework Orientado al objeto: Es un framework que implementa las especializaciones características de la programación orientada a objetos.
- GPUVMEM: Herramienta de Software de alto rendimiento para la síntesis de imágenes en interferometría.
- Lenguaje de Programación: Un lenguaje de programación es un lenguaje formal que especifica una serie de instrucciones para que una computadora produzca diversas clases de datos. Los lenguajes de programación pueden usarse para crear programas que pongan en práctica algoritmos específicos los cuales controlan el comportamiento físico y lógico de una computadora.
- fork de código: se define como la creación de un proyecto en una dirección distinta de la principal u oficial tomando el código fuente del proyecto ya existente.

REFERENCIAS BIBLIOGRÁFICAS

Alexandrescu, A. (2001). *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley.

ALMA (2009). How alma works. `https://www.almaobservatory.org/en/about-alma-at-first-glance/how-alma-works/`. Recuperado: 2018-09-05.

ALMA (2014). Hltau band 7. `https://almascience.nrao.edu/almadata/sciver/HLTauBand7/HLTau_Band7_ReferenceImages.tgz`. Recuperado: 2018-09-05.

Beck, K., & Gamma, E. (2000). *Extreme programming explained: embrace change*. addison-wesley professional.

Cárcamo, M., Román, P. E., Casassus, S., Moral, V., & Rannou, F. R. (2018). Multi-gpu maximum entropy image synthesis for radio astronomy. *Astronomy and computing*, *22*, 16–27.

Chen, W. (2011). The ill-posedness of the sampling problem and regularized sampling algorithm. *Digital Signal Processing*, *21*(2), 375–390.

Cheng, J., Grossman, M., & McKercher, T. (2014). *Professional Cuda C Programming*. John Wiley & Sons.

Cornwell, T. J., & Evans, K. (1985). A simple maximum entropy deconvolution algorithm. *Astronomy and Astrophysics*, *143*, 77–83.

Högbom, J. (1974). Aperture synthesis with a non-regular distribution of interferometer baselines. *Astronomy and Astrophysics Supplement Series*, *15*, 417.

Levanda, R., & Leshem, A. (2010). Synthetic aperture radio telescopes. *IEEE Signal Processing Magazine*, *27*(1).

Marti-Vidal, I., Vlemmings, W., Muller, S., & Casey, S. (2014). Uvmultifit: A versatile tool for fitting astronomical radio interferometric data. *Astronomy & Astrophysics*, *563*, A136.

McMullin, J., Waters, B., Schiebel, D., Young, W., & Golap, K. (2007). Casa architecture and applications. In *Astronomical data analysis software and systems XVI*, vol. 376, (p. 127).

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 classes*, (p. 16). ACM.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., & Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, vol. 26, (pp. 80–113). Wiley Online Library.

Pearson, T., Shepherd, M., Taylor, G., & Myers, S. (1994). Automatic synthesis imaging with difmap. In *Bulletin of the American Astronomical Society*, vol. 26, (p. 1318).

Peck, A., & Beasley, A. (2008). High resolution sub-millimeter imaging with alma. In *Journal of Physics: Conference Series*, vol. 131, (p. 012049). IOP Publishing.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press.

Radhakrishnan, R., Muthiah, A., & John, L. (1997). Performance impact of object oriented programming.

Rentsch, T. (1982). Object oriented programming. *ACM Sigplan Notices*, *17*(9), 51–57.

Román, P. E. (2012). El problema de síntesis de fourier en radio-astronomía y su resolución mediante métodos variacionales. *Revista del Instituto Chileno de Investigación de Operaciones*, *2*(1), 13–19.

Sault, R., & Wieringa, M. (1994). Multi-frequency synthesis techniques in radio interferometric imaging. *Astronomy and Astrophysics Supplement Series*, *108*, 585–594.

Thompson, A. R., Moran, J. M., Swenson, G. W., et al. (1986). *Interferometry and synthesis in radio astronomy*. Springer.

Turner, J. (2006). Atacama large millimeter array. *IAU Special Session*, *1*.

van Diepen, G. (2015). Casacore table data system and its use in the measurementset. *Astronomy and Computing*, *12*, 174–180.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., & Wood, B. (2006). Attribute-driven design (add), version 2.0. Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

Zhao, J.-H. (2013). Miriad for sma data reduction. *Astronomical Data Analysis Software and Systems XXII*, *475*, 181.

ANEXO A. FORMULARIO ISO 1700 MINIMIZADO

Formulario respondido por el desarrollador externo para evaluar la facilidad de implementación y la conformidad para con el sistema de *software*

	A	B	C	D	E	F
1	Pregunta	Muy disconforme	Disconforme	Ni conforme ni disconforme	Conforme	Muy conforme
2	El sistema desarrollado funciona como promete					
3	Las funcionalidades son suficientes					
4	Cumple con todas las exigencias de calidad					
5	No precenta nigún defecto Estructural					
6	El código es legible					
7	Similitud con la plataforma GPUVMEM					
8	Facilidad de extensión					
9	Implementación simple de objetos de prueba					
10	Permite ejecución en batch					
11	Tiene rendimiento aproximado a GPUVMEM					
12	La estructura de clases es fácil de enteder					
13	Las herramientas provistas son suficientes para el desarrollo					
14	No existen fugas de memoria					
15	El diseño de las interfaces es suficiente					
16	La implementación GPUVMEM framework satisface sus expectativas					
17	La información y los mensajes provistos por aplicación son suficientes					
18	El tiempo para adecuarse a la plataforma es menor a 3 hrs					

Figura A.1: Formulario de conformidad.
Fuente: Elaboración propia, 2018.