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

DESARROLLO DE UN ALGORITMO PARALELO EN GPU PARA ENCONTRAR PERIODOS DE OBJETOS VARIABLES PARA EL SISTEMA ALERCE.

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MIGUEL ANGEL SEPÚLVEDA HUENCHULEO

PROFESOR GUÍA:
Nancy Hitschfeld

PROFESOR CO-GUÍA:
Pablo Huijse

SANTIAGO DE CHILE
2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL
EN COMPUTACIÓN
POR: MIGUEL ANGEL SEPÚLVEDA HUENCHULEO
FECHA: 2021
PROF. GUÍA: NANCY HITSCHFELD KAHLER

DESARROLLO DE UN ALGORITMO PARALELO EN GPU PARA ENCONTRAR PERIODOS DE OBJETOS VARIABLES PARA EL SISTEMA ALERCE.

El broker astronómico ALeRCE usa un clasificador con machine learning que usa la variabilidad de la magnitud de las fuentes astronómicas, en forma de curvas de luz, para clasificar estas fuentes en un conjunto de clases como binarias eclipsantes, RR Lyrae y `active galactic nuclei`. ALeRCE se encarga también de calcular distintas características de las curvas de luz para el clasificador, y entre estas esta el periodo.

En esta memoria, se optimiza el algoritmo de cálculo de periodo usado en ALeRCE, MHAOV, implementando y diseñando una versión paralela en GPU del mismo algoritmo para mejorar su rendimiento, y usando un algoritmo de post proceso para mejorar su precisión, llamado promediado de subarmónicos. La implementación en GPU de MHAOV, denominada GMHAOV, se realizó en CUDA, y el algoritmo elegido como post proceso para mejorar la precisión fue el promediado de subarmónicos.

La validación de la versión paralela de MHAOV no se logra completar, presentando una desviación promedio del resultado de MHAOV de casi 120 % por cada frecuencia de prueba para datos reales, pero obteniendo una desviación de solo 5.58×10^{-9} para datos generados automáticamente. La causa de esto no se estudia a fondo, pero el efecto de este error en la precisión es de menos de 1 %.

En la paralelización, se obtuvo un speedup máximo de 10 y mínimo de 3 al calcular el periodograma de forma paralela para 100 y 1000 curvas con 150 puntos y 7000 frecuencias de prueba, respectivamente. El algoritmo de promediado de subarmónicos y una modificación de este llamado promediado de armónicos demostraron ser relevantes para una parte de las curvas de luz de binarias eclipsantes y RR Lyrae, llegando a incrementar la precisión en 10 veces en el caso de incluir el promediado de armónicos en GMHAOV, pero siendo contraproducente para algunos casos, como el promediado de subarmónicos en binarias eclipsantes.

Se estudió el código y la teoría de MHAOV, junto con el código de otro periodograma implementado en GPU llamado GCE, y además se investigó como crear interfaces en Python para ejecutar código CUDA, junto con los fundamentos de la detección de señales en curvas de luz.

Como trabajo futuro se propone usar el resultado del promediado de subarmónicos y de armónicos, GCE y GMHAOV para entrenar el clasificador de ALeRCE y evaluar el impacto que tendría en su precisión. Esto permitiría decidir si vale la pena incrementar el tiempo de ejecución del cálculo de periodos para mejorar el clasificador, ejecutando GCE y el promediado de armónicos/subarmónicos junto con GMHAOV, ya que el speedup de GMHAOV y la eficiencia de GCE permitiría hacer esto sin un impacto mayor en el tiempo de ejecución actual del periodograma.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

En el contexto la astronomía observacional, la medición continua de la magnitud de una fuente, como una estrella o una galaxia, permite la construcción de su curva de luz, que caracteriza la variación de la magnitud en función del tiempo. De esta curva de luz, es posible extraer características, o *features*, como su periodo y los parámetros de un ajuste sinusoidal, y usar estas características para clarificarla y hacer conclusiones sobre su naturaleza.El broker astronómico ALeRCE (Automatic Learning for Rapid Classification of Events) tiene el propósito de procesar el flujo de alertas de variabilidad astronómicas proveniente de telescopios de rastreo, calcular características de las curvas de luz asociadas a las alertas y usarlas para clasificar la fuente usando Machine Learning, y así permitirle a los astrónomos evaluar si vale la pena estudiar alguna de estas alertas y dedicarle valioso tiempo de observación. [1]

Una de las características más importantes de las curvas e luz es el periodos, que se calcula mediante un periodograma, que a cada frecuencia de prueba ω le asigna una medida de confianza $\Theta(\omega)$ cuyo valor determina que tan bien se ajusta cada frecuencia a la curva de luz al doblarla, lo cual consiste en definir la fase ϕ, definida por $\phi(t) \equiv t/T + \lfloor t/T \rfloor$ y graficar la magnitud en función de la fase.

El periodograma usado actualmente en ALeRCE es el *Multiharmonic Analysis of Variance* (MHAOV), que ajusta un sinusoide con un determinado número de armónicos a la curva doblada con una frecuencia ω y define como $\Theta(\omega)$ como un valor que define que tan bien se ajusta este sinusoide a la curva de luz. [2]. Actualmente, MHAOV es un algoritmo que se ejecuta completamente en CPU, y es impreciso para ciertos situaciones que se describirán en las siguientes secciones.

1.1. Motivación

El periodo de las curvas de luz una de sus propiedades más importantes ya que permite diferenciar entre objetos con curvas de luz similares pero distintos rangos de periodos y al estudiar la fuente en sí el periodo suele ser relevante para derivar propiedades físicas de las fuentes, como la distancia o su masa de las estrellas, además, el cálculo del periodo es actualmente una de las características mas computacionalmente costosas de calcular del sistema ALeRCE, y es por esto que es de gran importancia mejorar su precisión y eficiencia.

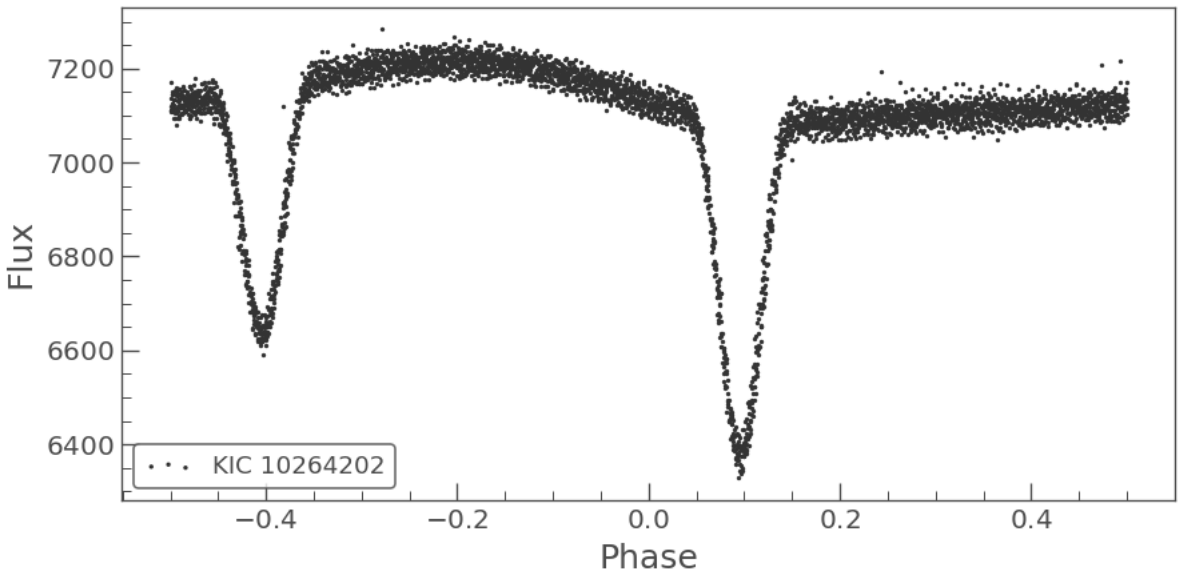

Figura 1.1: Curva de luz de una binaria eclipsante. El flujo del sistema se reduce si es que una de las estrellas oculta a la otra, pero hay una diferencia entre ambos eclipses, donde el flujo se reduce menos si la estrella que se bloquea es la menos brillante.

Podemos identificar dos areas en el que el periodograma usado por ALeRCE se puede mejorar:

1. **Eficiencia:** La velocidad de ejecución del cálculo de características es de gran importancia, debido a que ALeRCE actualmente usa datos del *Zwicky Transient Facility* (ZTF), un sondeo del cielo nocturno que produce 1.4 TB de datos por noche, y el sistema debe ser capaz de mantenerse al ritmo de este volumen de datos. En general, el volumen de datos en la astronomía es cada vez mayor, por ejemplo para el sondeo del LSST, el volumen de datos será cerca de 15 TB por noche, por lo que se debe mejorar continuamente la velocidad de estos algoritmos. Actualmente, MHAOV es uno de los algoritmos que toma más tiempo en el cálculo de características, tomando alrededor del 20 % del tiempo.

   La versión de MHAOV se ejecuta en CPU, pero como es posible calcular la medida de confianza para cada frecuencia de prueba y curva de luz independientemente, el algoritmo se podría ver beneficiado de una implementación paralela en GPU, ya que se debe procesar una gran cantidad de curvas de luz y frecuencias.

2. **Precisión:** Hay fuentes para las cuales MHAOV es particularmente impreciso. Específicamente, MHAOV calcula un periodo igual a la mitad del periodo real para gran parte de las binarias eclipsantes, ya que no es lo suficientemente sensible como para distinguir entre ambos eclipses (ver Fig. **??**). Esto lleva a que el 2 % de las binarias eclipsantes se clasifiquen incorrectamente como estrellas δ-Scuti, ya que estas últimas tienen un rango de periodos menor al de las binarias eclipsantes, pero sus curvas de luz son muy parecidas (ver Fig. 1.2. Debido a la gran cantidad de binarias eclipsantes, la contaminación en las estrellas δ-Scuti es considerable.

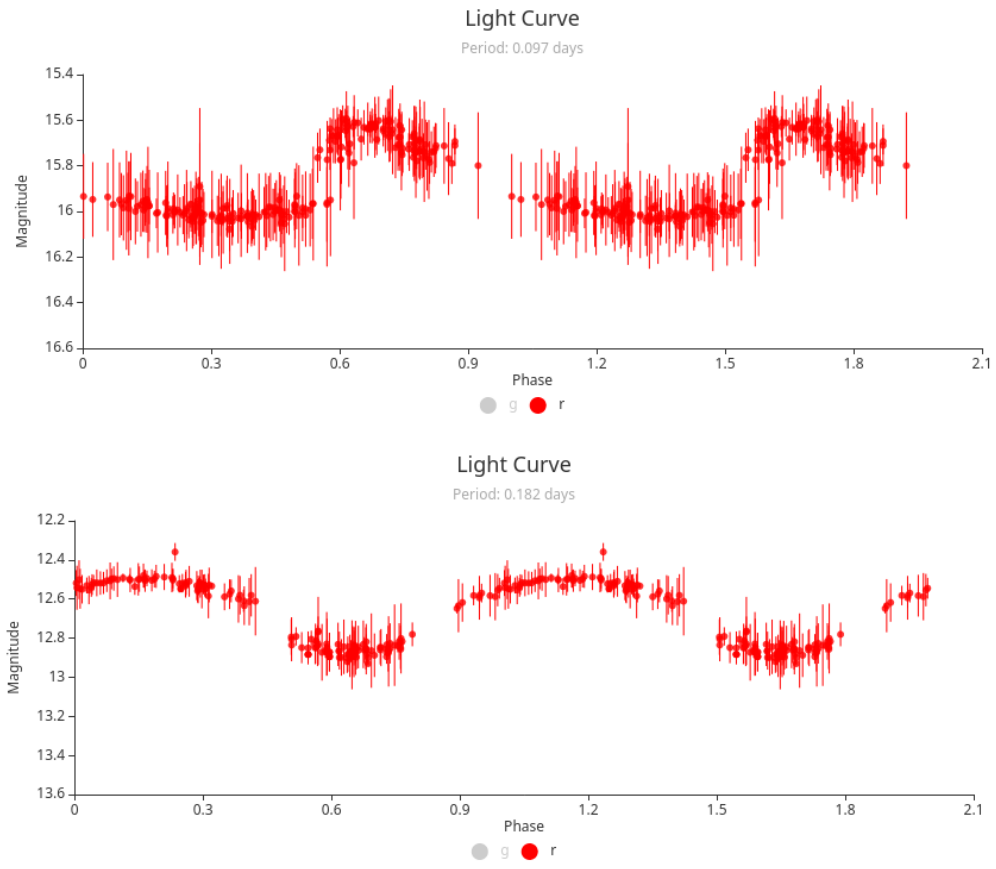

Figura 1.2: Curva de luz de una δ-Scuti (Arriba) y una binaria eclipsante (Abajo). Sus curvas de luz son similares pero sus periodos son muy diferentes (Fuente: ZTF Explorer)

Esto se podría solucionar considerando otros periodogramas simultáneamente en el clasificador, de manera que este pueda aprender cuando darle más importancia al resultado de uno que del otro y así mejorar la clasificación, o hacer algún otro postproceso que pueda determinar si el resultado de un periodograma es correcto o no. La desventaja de esta solución es el costo computacional: calcular otro periodograma podría incrementar considerablemente el tiempo de ejecución usado por el cálculo de los periodos.

1.2. Objetivos

Objetivo General

Desarrollar un algoritmo paralelo en GPU que permita determinar el periodo de un gran volumen de fuentes variables en base a sus curvas de luz, con especial énfasis en aumentar la precisión de este para estrellas binarias eclipsantes, de manera que se logre obtener el periodo real y no alguna fracción de este.

Objetivos Específicos

Para cumplir el objetivo de la memoria, se deben lograr las siguientes metas:

1. Estudiar y analizar algoritmos para obtener el periodo ya existentes, empezando por GCE y MHAOV.

2. Implementar una versión paralela en GPU de MHAOV, denominada GMHAOV, complementando el diseño propio con las técnicas de paralelización usadas en GCE.

3. Validar y comparar el desempeño de los algoritmos, tanto en términos de tiempo de ejecución como en precisión, usando conjuntos de datos con periodo conocido.

4. Evaluar el impacto de incluir el resultado de más de un periodograma en el clasificador de ALeRCE.

5. Verificar que al integrar algoritmo mas preciso en ALERCE es lo suficientemente eficiente en calcular las características y representa una mejora con respecto al algoritmo actual.

1.3. Plan de Trabajo

Ya que este trabajo consistió en desarrollar implementaciones eficientes de algoritmos ya existentes, se pasó por una fase de investigación y evaluación para determinar con qué algoritmos se trabajaría y como se mejorarían. Estos algoritmos son:

- **MHAOV:** El algoritmo siendo usado actualmente por ALeRCE, es necesario saber que tan preciso es y cual es su tiempo de ejecución a fin de compararlo con otros algoritmos. Para mejorar su velocidad de ejecución, se implementó una versión en GPU de este algoritmo, por lo que es muy importante entenderlo a profundidad para poder implementar GMHAOV.

- **GCE:** Un periodograma implementado en GPU que usa entropía condicional, una medida de la correlación entre el magnitud y la fase de una curva de luz doblada, y que es considerablemente más rápido que MHAOV. **??**

- **Métodos de identificar subarmónicos:** A veces, periodogramas como MHAOV identifican subarmónicos del periodo real como se describió en 1.1, por lo que es beneficioso investigar algoritmos que ayuden a identificar estas situaciones para poder encontrar el periodo real.

Se tomó la decisión de implementar GMHAOV en CUDA, ya que se tenía experiencia con el desarrollo de algoritmos en GPU en este lenguaje, se podía usar el código de GCE como base para poder ejecutar código en CUDA desde python, y además se tenía el hardware necesario para esto. El diseño de GMHAOV se hizo tomando ventaja de que el cálculo de la medida de confianza es independiente entre curvas de luz y frecuencias, y de hecho el mismo diseño de MHAOV permitía paralelizar este cálculo para cada punto de cada curva de luz.

GMHAOV se validó usando el código de generación de curvas de luz disponible en el repositorio que contiene MHAOV, y curvas de luz de sets etiquetados (Conjuntos cuyas propiedades son conocidas). Como GMHAOV debe presentar exactamente los mismos resultados

que su versión secuencial, bastó con comparar los valores de $\Theta(\omega)$ para ambas versiones y asegurarse que estén lo suficientemente cerca. Usando el generador de curvas de luz, fue posible evaluar el tiempo de ejecución de GMHAOV y compararlo con GCE y MHAOV en función de el número de frecuencias de prueba y número de curvas de luz.

Con respecto a mejorar la precisión del cálculo del periodo, en especial en el caso de las binarias eclipsantes, se implementó un algoritmo que identifica submúltiplos llamado promediado de subarmónicos, que permite discriminar entre señales falsas y señales reales en el periodograma, pero requiere ordenar los valores de Θ para todas las frecuencias de prueba. Se evalúa el impacto de este postproceso en la precisión del periodograma y además su velocidad de ejecución.

Para estudiar el impacto de los algoritmos implementados en el clasificador, se planea entrenar el clasificador de ALeRCE incluyendo como características el resultado de GMHAOV, GCE y del promediado de subarmónicos. Con esto se espera evaluar el impacto de agregar los resultados de otros clasificadores, en especial con respecto a las binarias eclipsantes.

1.4. Contenido de la memoria

- **Análisis del software:** Se hace un análisis profundo del software que se usó como base en el desarrollo de la solución, incluyendo comparaciones entre sus precisiones y rendimientos.
- **Diseño:** Se describe el diseño de la solución, incluyendo el desarrollo de la interfaz en Python para ejecutar el código de CUDA, el proceso de paralelización de MHAOV y el diseño de un postproceso al periodograma que es capaz de identificar señales falsas.
- **Implementación:** Se muestra la implementación de la solución, explicando en detalle y con secciones de código como se llevó a cabo el diseño del capítulo anterior.
- **Resultados:** Se describen los métodos de evaluación del rendimiento y precisión de la solución, y se presentan sus resultados y comparaciones con la versión secuencial de MHAOV y el impacto del postproceso que identifica señales falsas en el periodograma.
- **Análisis y conclusión:** Se realiza un análisis de los resultados expuestos en la sección anterior, se realizan conclusiones respecto a los objetivos en función de esto y se discute el trabajo futuro.

Capítulo 2

Análisis del software

En este capítulo, se detallan los detalles del pipeline de ALeRCE y de los algoritmos estudiados durante el curso de la memoria, empezando por MHAOV, GCE y la comparación entre estos dos.

2.1. El pipeline de ALeRCE

El sistema de ALeRCE recibe un flujo de aletas desde ZTF, que contiene varios datos de la observación como la magnitud en distintas bandas y su posición. Estos datos se procesan de la siguiente manera, como se observa en la figura 2.1 [3]

1. **S3 upload:** Se suben los paquetes de alertas al servicio de almacenamiento de AWS S3 para uso posterior.

2. **Cross match:** Se usa la posición de la alerta para buscar coincidencias con catálogos externos y obtener más información sobre el objeto.

3. **Stamp classifier:** La imagen de los objetos nuevos son clasificadas.

4. **Preprocessing:** Se hacen correcciones a la magnitud y se calculan estadísticas simples de las curvas de luz.

5. **Light curve features:** Se calculan estadísticas avanzadas para curvas de luz con más de 6 puntos, incluyendo el periodo.

6. **Light curve classifier:** Se clasifica el objeto usando las características calculados y a los datos del objeto adquiridos en el crossmatch.

7. **Outliers:** Se aplica un algoritmo de detección de outliers para identificar curvas de luz que pueden ser particularmente interesantes.

8. **ALeRCE stream:** Se reportan las curvas de luz en un stream incluyendo su clasificación y las características calculadas.

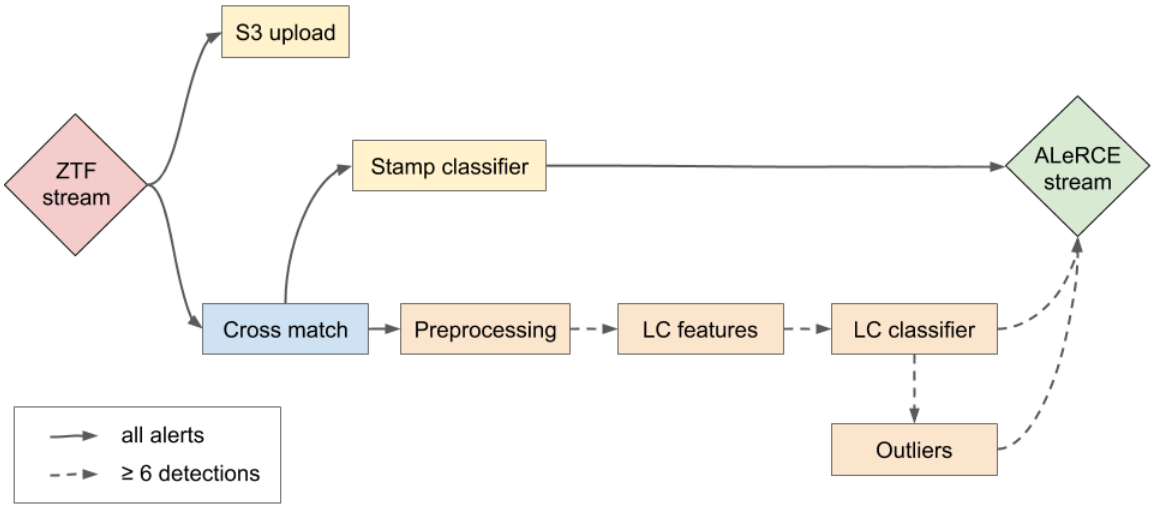

Figura 2.1: Pipeline de ALeRCE [1]. Entre el procesamiento que se le hace al stream de alertas de ZTF, está el tratamiento a las curvas de luz, donde después de una corrección, se calculan características que serán usadas para la clasificación de las curvas de luz.

2.2. Arquitectura de MHAOV

MHAOV usa como medida de confianza que tan bien se ajusta un sinusoide con un determinado número de armónicos a la forma de la curva. En su base, consiste en asumir que las observaciones X son la suma de la señal F y el error E, de tal manera que la observación k es $X_k = F_k + E_k$. La forma de la señal $F^{(N)}$ es de una serie de fourier con N armónicos. Con el fin de descomponer la señal en sus componentes, definimos el polinomio $\Psi_{2N}(z)$, de manera de que para una frecuencia de prueba ω, los argumentos del polinomio tienen la forma $z_k = e^{i\omega}$, y se asocian a la k-ésima observación, y su valor para z_k es $\Psi_{2N}(z_k) = z_k^N F(t_k))$. Notamos que en un espacio de Hilbert un polinomio de grado N puede ser descompuesto en función de una base ortonormal del espacio $\{\Phi\}_{n=1,..,N}$

$$\Psi_N(z) = \sum_{n=1}^{N} c_n \Phi_n(z). \tag{2.1}$$

Donde el producto interno está definido como

$$(\Phi, \Psi) = \sum_{k=1}^{K} g_k \Phi(z_k)\overline{\Psi(z_k)} \tag{2.2}$$

$$g_k \approx \frac{1}{\mathrm{Var}\{X_k\}}.$$

Esta base se puede obtener de manera iterativa

$$\tilde{\Phi}_0(z) = 1 \tag{2.3}$$

$$\tilde{\Phi}_{n+1}(z) = z\tilde{\Phi}_n - \alpha_n z^n \overline{\tilde{\Phi}_n(z)} \tag{2.4}$$

$$\Phi_n(z) = \frac{\tilde{\Phi}_n(z)}{\sqrt{(\tilde{\Phi}_n, \tilde{\Phi}_n)}} \tag{2.5}$$

$$\alpha_n = \frac{(z\tilde{\Phi}_n, \tilde{\Phi}_n}{(z^n \overline{\tilde{\Phi}_n}, \tilde{\Phi}_n)} \tag{2.6}$$

$$c_n = \frac{(\Psi, \tilde{\Phi}_n)}{\sqrt{(\tilde{\Phi}_n, \tilde{\Phi}_n)}}. \tag{2.7}$$

Para el MHAOV, la medida de confianza se basa en estimadores de la varianza de F y E, y es $\Theta \equiv \widehat{\mathrm{Var}}\{F\}/\widehat{\mathrm{Var}}\{E\}$, que tiene la forma (Para mas detalle ver [2])

$$\Theta(\omega) = \frac{(K - 2N - 1)\sum_{n=0}^{2N} |c_n|^2}{2N[(X, X) - \sum_{n=0}^{2N} |c_n|^2]}. \tag{2.8}$$

El procedimiento para calcular la medida de confianza es entonces el siguiente:

1. Fijar los valores iniciales para la recurrencia, $n = -1$, $\tilde{\Phi}_{-1} = 1/z$, y $\alpha_{-1} = 0$.
2. Usar (2.4) para encontrar el siguiente valor de $\tilde{\Phi}_n$
3. Usar (2.2) para calcular los valores de (2.6) y (2.7).
4. Volver al paso 2, iterando esto N veces.
5. Calcular $\Theta(\omega)$ usando 2.8.

Algorithm 1 MHAOV

$wmean \leftarrow$ weighted_mean(mag, err, N) ▷ weighted mean
for $i \leftarrow 1$ to N **do**
  $wvar \leftarrow wvar + (mag[i] - wmean)^2/err[i]^2$ ▷ weighted variance (X, X)
end for
$\Theta \leftarrow 0$
for $i \leftarrow 1$ `to` N **do** ▷ Inicialización de variables
  $\phi \leftarrow 2\pi(mjd[i] * \omega - \lfloor mjd[i] * \omega \rfloor)$
  $z_r = \cos(\phi)$ ▷ Inicialización de z
  $z_i = \sin(\phi)$
  $zn_r = 1$
  $zn_i = 0$
  $p_r = 1/err[i]$
  $p_i = 0$
  $cf_r = (mag[i] - wmean)\cos(K * \phi)/err[i]$
  $cf_i = (mag[i] - wmean)\sin(K * \phi)/err[i]$
end for
for $j \leftarrow 1$ to $2K + 1$ **do**
  $sn, al_r, al_i, sc_r, sc_i \leftarrow 0$
  for $i \leftarrow 1$ `to` N **do**
    $sn \leftarrow sn + {p_r}^2 + {p_i}^2$
    $al_r \leftarrow al_r(z_r[i] \cdot p_r[i] - z_i[i] \cdot p_i[i])/err[i]$ ▷ Siguiente valor de de α
    $al_i \leftarrow al_i + (z_r[i] \cdot p_i[i] + z_i[i] \cdot p_r[i])/err[i]$
    $sc_r \leftarrow sc_r + p_r[i] \cdot cf_r[i] + p_i[i] \cdot cf_i[i]$ ▷ Siguiente valor de c
    $sc_i \leftarrow sc_i + p_r[i] \cdot cf_i[i] - p_i[i] \cdot cf_r[i]$
  end for
  $sn \leftarrow \max(sn, 10 \cdot -9)$
  $al_r \leftarrow al_r/sn$
  $al_i \leftarrow al_i/sn$
  $\Theta \leftarrow \Theta + ({sc_r}^2 + {sc_i}^2)/sn$
  for $i \leftarrow 1$ to N **do**
    $s_r \leftarrow al_r \cdot zn_r[i] - al_i \cdot zn_i[i]$
    $s_i \leftarrow al_r \cdot zn_i[i] + al_i \cdot zn_r[i]$
    $tmp \leftarrow p_r[i] \cdot z_r[i] - p_i[i] \cdot z_i[i] - s_r \cdot p_r[i] - s_i \cdot p_i[i]$
    $p_i \leftarrow p_r[i] \cdot z_i[i] - p_i[i] \cdot z_r[i] - s_r \cdot p_i[i] - s_i \cdot p_r[i]$ ▷ Actualizar el valor de Φ_n
    $p_r \leftarrow tmp$
    $tmp \leftarrow zn_r[i] \cdot z_r[i] - zn_i[i] \cdot z_i$
    $zn_i \leftarrow zn_i[i] \cdot z_r[i] + zn_r[i] \cdot z_i$ ▷ Actualizar el valor de z^n
    $zn_r \leftarrow tmp$
  end for
end for
return $(K - 2N - 1) \cdot \Theta/(2N \cdot \max(wvar - \Theta, 10^{-9}))$

2.3. Arquitectura de GCE

2.4. Comparación de MHAOV vs GCE

Se realizó una comparación de MHAOV y GCE usando tests previamente escritos por Pablo Huijse, usando sets etiquetados, que son archivos con información sobre objetos, incluyendo su clase y periodo. Los tests miden el tiempo de ejecución y la precisión de cada algoritmo usando curvas de luz con mas de 20 puntos en sus curvas de luz, y fueron realizados en un computador con un procesador Intel Core i7-9750H de 12 núcleos lógicos y una GPU Nvidia GeForce 1660 Ti con 6GB de memoria. MHAOV fue ejecutado con 8 threads en paralelo.

Para el primer test, se usaron alrededor de 8500 estrellas RR Lyrae, que tienen curvas suaves y periodos marcados, y en las tablas 2.1 y 2.2 se encuentran los resultados.

Tabla 2.1: Tiempos de ejecución total y por curva para ambos algoritmos, usando el mismo conjunto de datos.

Algoritmo	Tiempo de ejecución [s]	Tiempo por curva [ms]
GCE	85	32
MHAOV	629	242

Tabla 2.2: Clasificación en porcentajes para los valores del periodo calculado respecto al real. Acierto significa que estos sin similares, Múltiplo que el calculado es un múltiplo del original, Submúltiplo que es una fracción del original, Alias que es un alias del original (Es decir, que se ajusta igualmente bien a los datos debido a su naturaleza discreta), y Otro para cualquier otro valor del periodo calculado.

Algoritmo	Acierto	Múltiplo	Submúltiplo	Alias	Otro
GCE	69.5	1.7	1.8	1.7	25.3
MHAOV	88.8	1.0	0.2	1.1	8.7

La segunda prueba se hizo con binarias eclipsantes, y la comparación de la precisión de ambos algoritmos está en la tabla 2.3. Notemos que si multiplicamos los periodos obtenidos por 2, los resultados cambian a los resultados de la tabla 2.5, pero aún así observamos cierto grado de contaminación, y al hacer esto estamos obteniendo el periodo equivocado para binarias eclipsantes donde la diferencia de periodos es significativa.

Tabla 2.3: Clasificación en porcentajes para los valores del periodo calculado respecto al real para las binarias eclipsantes.

Algoritmo	Acierto	Múltiplo	Submúltiplo	Alias	Otro
GCE	0.42	93.30	0.00	0.00	6.28
MHAOV	11.1	55.14	0.32	0.56	32.92

Tabla 2.4: Comparación entre ambos algoritmos para las tres situaciones: usando RR Lyrae, binarias eclipsantes, y multiplicando por 2 el periodo obtenido por los algoritmos para binarias eclipsantes.

Prueba	Ambos aciertan [%]	MHAOV acierta, GCE no [%]	GCE acierta, MHAOV no [%]	Ninguno acierta [%]
RR Lyrae	61.40	27.40	3.40	7.78
Binarias Eclipsantes	0.20	0.22	10.86	88.72
Binarias Eclipsantes (Ajustado)	10.46	82.46	0.60	6.48

Tabla 2.5: Precisión de MHAOV después de multiplicar los periodos obtenidos por 2

Algoritmo	Acierto	Múltiplo	Submúltiplo	Alias	Otro
2 × GCE	54.50	0.30	11.88	1.30	32.02
2 × MHAOV	92.92	0.32	0.42	0.14	6.20

Notamos que GCE, aunque es considerablemente más rápido que MHAOV, es significativamente menos preciso, incluso para el caso de submúltiplos, donde se esperaba que redujera estos casos ya que es un algoritmo que no ajusta una sinusoidal a diferencia de MHAOV. Además, el aporte de GCE con respecto a calcular el periodo de curvas correctamente donde MHAOV no lo hace es pequeño para las RR Lyrae, solo 3.4 % y 0.60 % para las binarias eclipsantes, pero esto puede ser significativo por la gran cantidad de binarias eclipsantes que hay. Se concluye a partir de estas pruebas que no se puede simplemente reemplazar MHAOV por GCE en ALeRCE, ya que se perderá demasiada precisión, pero implementar una versión paralela de MHAOV tan rápida como GCE y tan precisa como MHAOV resultará beneficioso.

Capítulo 3

Diseño

A continuación se detalla el detalle del diseño de la solución, empezando por explicar el proceso de paralelización de MHAOV y luego el detalle del algoritmo de promediado de subarmónicos.

3.1. Ejecución de código de CUDA usando python

Ejecutar código de CUDA directamente desde python requiere del paquete `cupy`, para enviar las curvas de luz a la memoria de la GPU, y de `cython`, para realizar llamadas a la función en C que ejecuta el kernel desde python. Para poder compilar el código, se debe escribir un script en python que incluya correctamente las librerias, cambiar el método de compilación para incluir CUDA y que indique las arquitecturas para las cuales se debe realizar la documentación. Afortunadamente, el código de GCE tiene una estructura muy similar a la que se planeaba, ya que GCE ofrece una interfaz en python desde la cual se pueden hacer llamadas al kernel, por lo que se usó el código de GCE como base para el script de compilación y para la clase de python encargada de la preparación de los datos para ser enviados a la memoria de la GPU.

3.2. Paralelización de MHAOV

Como se expuso en la sección 1.1, MHAOV puede ser paralelizado a nivel de las curvas de luz y las frecuencias de prueba, ya que los valores $\Theta(\omega)$ son independientes entre sí. Adicionalmente, de la descripción del algoritmo en la sección 2.2 se observa que los ciclos que iteran por el largo de la curva de luz también se pueden paralelizar, para el primer y segundo ciclo la paralelización es trivial, y para el segundo ciclo se puede calcular el aporte de cada ciclo a los valores de las variables, y luego usar reducción para sumar estos aportes y usar los resultados para calcular el incremento en Θ y para el tercer ciclo que una vez más es trivialmente paralelizable ya que los ciclos no dependen entre sí.

Así, se requiere coordinar los threads en ciertos puntos del kernel para evitar data races, y a continuación se presenta el pseudo-código del kernel de GMHAOV, siendo análogo al código presentado en 2.2:

Algorithm 2 GMHAOV

function GMAHOVKERNEL(lc_i, ω_i, i)
  if $i == 0$ **then**
    $\Theta \leftarrow 0$
  end if
  $wmean \leftarrow$ cálculo paralelo de los aportes de cada punto y suma por reducción
  ___syncthreads()
  $wmvar \leftarrow$ cálculo paralelo de los aportes de cada punto y suma por reducción
  ___syncthreads()
  $z_r,\ z_i,\ z_n,\ p_r,\ p_i,\ cf_r,\ cf_i \leftarrow$ valores inciales según $mjd[lc_i, i]$, $err[lc_i, i]$ y $mag[lc_i, i]$
  ___syncthreads()
  for $j \leftarrow 0$ to $2K + 1$ **do**
    $al_r,\ al_i,\ sc_r,\ sc_i,\ sn \leftarrow$ suma de los aportes de cada punto por reducción
  ___syncthreads()
    if $i == 0$ **then**
      $\Theta \leftarrow \Theta + ({sc_r}^2 + {sc_i}^2)/sn$
    end if
    $s_r,\ s_i,\ p_r,\ p_i,\ zn_i,\ zn_r \leftarrow$ Valores actualizados usando $al_r,\ al_i,\ zn_r,\ zn_i,\ p_r,\ p_i,\ z_r$ y z_i
  ___syncthreads()
  end for
  ___syncthreads()
  $\Theta \leftarrow (K - 2N - 1) \cdot \Theta/(2N \cdot \max(wvar - \Theta, 10^{-9}))$
end function

Como las curvas de luz son independientes entre sí y estas tienen en promedio 150 puntos, se pueden guardar completamente en memoria compartida para cada bloque y así solo acceder a memoria global solo para obtener las curvas y su número de puntos. Además de las curvas de luz, se deberá usar la memoria local para la reducción de 8 variables, usando 8 arreglos del largo de la curva de luz más larga.

3.3. Promediado de subarmónicos

El promediado de subarmónicos es descrito en [4], pero el código no está disponible así que se tuvo que diseñar este algoritmo en base a la descripción en el paper citado. La idea del promediado de subarmónicos, es que si en el periodograma se tiene una señal real en $\Theta(\omega)$, entonces el periodograma tendrá un peak relevante en $\frac{\omega}{2}$. De esta manera, si se promedia el valor de Θ en ω y en $\frac{\omega}{2}$, se debe seguir teniendo un valor significativo, de lo contrario se tenía una señal falsa y la frecuencia real está en otra parte.

Para poder implementar esto como algoritmo, es necesario ordenar los valores de Θ para obtener las frecuencias más relevantes. El costo computacional de esto se puede reducir buscando maximos o mínimos locales de Θ escaneando el arreglo de valores y marcando aquellos que son mayores que sus vecinos, lo cual tiene un costo computacional de $\mathcal{O}(N_\omega)$, con N_ω la cantidad de frecuencias de prueba, y luego ordenando estos valores. El costo computacional seguirá siendo de $\mathcal{O}(N_\omega \ln N_\omega)$, pero el ordenamiento podrá ignorar la gran mayoría de las frecuencias. Luego se elijen las frecuencias con mayor valor de Θ, y se marcan

como señales significantes.

Se establece un criterio simple para determinar si una señal significante es real: Si existe otra señal significante cerca de la mitad de su frecuencia asociada, entonces se promedian los valores de Θ y se le asigna como puntaje a la frecuencia. Si no existe esta señal, entonces su puntaje es nulo. Así se elije la frecuencia asociada a la señal con el mayor puntaje y se eliminan las señales falsas. Asociarle este puntaje al resultado del periodograma es conveniente pues se puede usar posteriormente como característica del clasificador.

Capítulo 4

Implementación

En este capítulo se detalla la implementación de la solución, empezando por como se preparan los datos en python para ser cargados en la memoria de la GPU, la implementación del kernel y finalmente el promediado de subarmónicos.

4.1. Preparación de los datos en python

Se crea la clase de python GMHAOV, basada en GCE, con una función que recibe las curvas de luz, las frecuencias de prueba y el tamaño de batch, según el cual se dividen las curvas de luz y se mandan en partes para no sobrecargar la memoria. Se ignoran varias funciones de GCE que no se reimplementaron en GMHAOV.

```
def batched_run_const_nfreq(self, lightcurves, batch_size, freqs):
...
    # split by light curve batches
    split_inds = []
    i = 0
    while i < len(lightcurves):
        i += batch_size
        if i >= len(lightcurves):
            break
        split_inds.append(i)

    lightcurves_split_all = np.split(lightcurves, split_inds)
...
    bf = []
    per_vals = []
    for i, light_curve_split in iterator:

        # run one light curve batch
        out = self._single_light_curve_batch(
            light_curve_split,
            freqs)
...
```

Como se usa CUDA para el procesamiento de las curvas de luz, es necesario determinar el largo de cada curva de luz y el largo máximo, para que se pueda realizar el cálculo del

periodograma sin considerar datos que pueden contaminar las curvas de luz y para poder determinar el tamaño que cada bloque debe tener, pues estos deben tener a lo menos un thread por punto de la curva de luz. Luego se crean arreglos para los tiempos, magnitudes y errores para las curvas de luz, donde cada curva tiene un largo equivalente al de la curva más larga, y las entradas adicionales se rellenan con 0s.

Originalmente, GCE realizaba otros cálculos relacionados al binning de las magnitudes y los tiempos, además de considerar los valores de $\dot{p}$, pero esto fue removido y el código fue adaptado para funcionar sin esto.

```

# determine maximum length of light curves in batch
# also find minimum and maximum magnitudes for each light curve
max_length = 0
number_of_pts = np.zeros((len(light_curve_split),)).astype(int)
for j, lc in enumerate(light_curve_split):
    number_of_pts[j] = len(lc)
    max_length = max_length if len(lc) < max_length else len(lc)
light_curve_arr = np.zeros((len(light_curve_split), max_length, 3))
logging.info(f"Number of points {number_of_pts}")

# populate light_curve_arr
for j, lc in enumerate(light_curve_split):
    light_curve_arr[j, : len(lc)] = np.asarray(lc)

# separate time and mag info
light_curve_times = light_curve_arr[:, :, 0]

light_curve_mags = light_curve_arr[:, :, 1]

light_curve_errs = light_curve_arr[:, :, 2]
```

Luego se aplanan los arreglos para formar tres arreglos muy largos, y se cargan en la memoria de la GPU usando la función de cupy `xp.asarray` junto con el arreglo con las frecuencias de prueba, el arreglo que contendrá los resultados del periodograma y el arreglo que contiene el número de armónicos con el que se correrá el periodograma para cada curva. Luego se llama al envoltorio de la función que llama al kernel con los arreglos creados y otros datos como el número de frecuencias y el número de curvas de luz.

```
        # flatten everything
        light_curve_times_in = (
            xp.asarray(light_curve_times).flatten().astype(self.dtype)
        )

        light_curve_mags_in = xp.asarray(light_curve_mags.flatten()).astype(
            self.dtype
        )

        light_curve_errs_in = xp.asarray(light_curve_errs.flatten()).astype(
            self.dtype
        )
        number_of_pts_in = xp.asarray(number_of_pts).astype(xp.int32)
```

```
14
15      freqs_in = xp.asarray(freqs).astype(self.dtype)
16
17      per_vals_out_temp = xp.zeros(
18          (len(freqs_in) * len(light_curve_times),), dtype=self.dtype
19      )
20
21      max_num_pts_in = number_of_pts_in.max().item()
22      Nharmonics = xp.ones(len(light_curve_times), dtype=xp.int32)
23
24      self.gmhaov_func(
25          per_vals_out_temp,
26          freqs_in,
27          len(freqs_in),
28          light_curve_times_in,
29          light_curve_mags_in,
30          light_curve_errs_in,
31          number_of_pts_in,
32          Nharmonics,
33          len(light_curve_times),
34          max_num_pts_in
35      )
36      per_vals_out_temp = per_vals_out_temp.reshape(
37          len(light_curve_times), len(freqs_in)
38      )
```

En la última linea se cambia la forma del arreglo con los resultados del periodograma de manera que sea un arreglo de los valores de Θ para cada curva de luz. El envoltorio de la función que llama al kernel está escrito en Cython, y usa una decoración que reemplaza los arreglos que recibe la función como argumentos por sus punteros en memoria. En este caso, como los arreglos están en memoria de la GPU, estos son reemplazados por los punteros a su localización en la GPU. Luego, la función en CUDA `run_gmhaov`, encargada de hacer las llamadas al kernel, es ejecutada con estos punteros como argumentos.

```
1  @pointer_adjust
2  def run_gmhaov_wrap(per, freqs, num_freqs, mjds, mags, errs, num_pts_arr,
       ↪ Nharmonics_arr, num_lcs, num_pts_max,  wmeans_arr, wvars_arr):
3      cdef size_t per_in = per
4      cdef size_t freqs_in = freqs
5      cdef size_t num_freqs_in = num_freqs
6      cdef size_t mags_in = mags
7      cdef size_t mjds_in = mjds
8      cdef size_t errs_in = errs
9      cdef size_t num_pts_arr_in = num_pts_arr
10     cdef size_t Nharmonics_arr_in = Nharmonics_arr
11     cdef size_t num_lcs_in = num_lcs
12     cdef size_t num_pts_max_in = num_pts_max
13     run_gmhaov(<fod *> per_in, <fod *> freqs_in, num_freqs_in, <fod *> mags_in, <fod *>
       ↪ mjds_in, <fod *> errs_in,
14                         <int *> num_pts_arr_in, <int *> Nharmonics_arr_in, num_lcs,
       ↪ num_pts_max_in, <float *> wmeans_arr_in, <float *> wvars_arr_in)
```

4.2. Paralelización

Como la memoria local usada por cada bloque es dinámica, y los arreglos relevantes ya están en la memoria de la GPU, `run_gmhaov` es una función relativamente corta, pues no necesita mover objetos desde la memoria del host a la del dispositivo. El tamaño ded la grilla se escoje como el número de frecuencias en el eje x y el número de curvas de luz en el eje y, y la memoria de cada bloque es el espacio que ocupará cada curva de luz más los arreglos necesarios para hacer las reducciones descritas en **??**.

Como siempre es conveniente tener potencias de 2 como tamaño de bloque, se aproxima el largo máximo de las curvas de luz a su próxima potencia de 2 y se elije esto como tamaño de bloque, con un valor máximo de 1024, de manera que cada bloque pueda procesar completamente cada curva de luz. Para curvas de luz con más de 1024 puntos, el kernel se debería encargar de cubrir toos los puntos.

```
1  void run_gmhaov(fod *d_per, fod *d_freqs, int num_freqs, fod *d_mag, fod *d_mjd, fod *
       ↪ d_err,
2              int *num_pts_arr, int *Nharmonics_arr, int num_lcs, int num_pts_max, float
       ↪ *wmeans_arr, float *wvars_arr){
3      dim3 griddim(num_freqs, num_lcs, 1);
4
5      // determine shared memory allocation size
6      size_t numBytes = sizeof(fod)*num_pts_max + // magnitude values
7                  sizeof(fod)*num_pts_max + // error values
8                    sizeof(fod) * num_pts_max + // time values
9                    5 * sizeof(fod) * num_pts_max + // sn, scr, sci, alr, ali to sum
10                   3 * sizeof(fod) * num_pts_max; // wvar, wmean, w_sum to sum
11
12     int v = num_pts_max;
13     // Source for rounder: https://graphics.stanford.edu/~seander/bithacks.html#
       ↪ RoundUpPowerOf2
14     v--;
15     v |= v >> 1;
16     v |= v >> 2;
17     v |= v >> 4;
18     v |= v >> 8;
19     v |= v >> 16;
20     v++;
21     if (v > 1024) v = 1024;
22     kernel<<<griddim, v, numBytes>>>(d_per,
23                                      d_freqs, num_freqs,
24                                      d_mag, d_mjd,
25                                      d_err, num_pts_arr,
26                                      Nharmonics_arr, num_lcs,
27                                      num_pts_max, wmeans_arr, wvars_arr);
28     cudaDeviceSynchronize();
29     gpuErrchk(cudaGetLastError());
30 }
```

El kernel empieza por crear el arreglo compartido `mag_share`, que tiene el espacio necesario para guardar todos los arreglos locales que se usan en cada bloque. Se definen entonces el resto de los arreglos como punteros a las secciones que le corresponden en memoria, listos

para ser rellenados por los datos en memoria global, y finalmente se declaran las cantidades que se comparten en todo el bloque.

```
1
2  ___global___ void kernel(fod* ___restrict___ d_per,
3                           fod* d_freqs, int num_freqs,
4                           fod* d_mag, fod* d_mjd,
5                           fod* d_err, int* num_pts_arr,
6                           int* Nharmonics_arr, int num_lcs,
7                           int num_pts_max, fod* wmeans_arr, fod* wvars_arr)
8     {
9     /* Assign the magnitude,error and time values to sections of the shared array */
10    extern ___shared___ fod mag_share[];
11    fod *error_share = (fod*) &mag_share[num_pts_max];
12    fod *time_share = (fod*) &error_share[num_pts_max];
13
14    fod *sn_sum = (fod*) &time_share[num_pts_max];
15
16    fod *scr_sum = (fod*) &sn_sum[num_pts_max];
17    fod *sci_sum = (fod*) &scr_sum[num_pts_max];
18    ...
19    ___shared___ fod sn;
20    ___shared___ fod scr;
21    ___shared___ fod sci;
22    ...
```

En caso de que se haya elegido un tamaño de grilla en el eje y menor al número de curvas de luz, se introduce un ciclo que se asegura que todas las curvas de luz sean procesadas, empezando por lc_i = blockIdx.y. Luego se carga en memoria la curva de luz actual de forma paralela, y se itera en frecuencias en el eje x de la misma manera que se hace en el eje y, empezando por f_i = blockIdx.x. Se definen las variables que se usarán para el cálculo de Θ, y uno de los threads inicializa a $\Theta = 0$.

```
1     for (int lc_i = blockIdx.y; lc_i < num_lcs; lc_i += gridDim.y){
2        int num_pts_current = num_pts_arr[lc_i];
3           int current_index = lc_i * num_pts_max + i;
4           mag_share[i] = d_mag[current_index];
5           error_share[i] = d_err[current_index];
6           time_share[i] = d_mjd[current_index];
7        }
8        ___syncthreads();
9        for (int fi = blockIdx.x; fi < num_freqs; fi += gridDim.x) {
10          int idx = threadIdx.x;
11          fod zr, zi, znr, zni, pr, pi, cfr, cfi;
12
13          if (idx == 0) aov = 0;
14          ...
```

Un problema con la implementación de GMHAOV es que si la curva de luz tiene un largo mayor que el tamaño de bloque máximo de 1024, no es posible cubrirla en el periodograma y el algoritmo deja de funcionar. El algoritmo toma $i = idx$, el id del thread en el bloque, y

calcula *wmean* y *wvar* encontrando el aporte de cada valor a estos valores y luego usando reducción para encontrar sus valores finales.

```
...
w_sum_sum[i] = 1.0 / powf(abs(error_share[i]), 2.0f);
w_mean_sum[i] = mag_share[i] / powf(abs(error_share[i]), 2.0f); }
___syncthreads();
for(int size = NUM_THREADS/2; size > 0; size/=2){
   if(idx < size && (idx + size < num_pts_arr[lc_i])){
      w_sum_sum[idx] += w_sum_sum[idx + size];
      w_mean_sum[idx] += w_mean_sum[idx + size];
   }
   ___syncthreads();
}

if (idx == 0) {
   w_sum = w_sum_sum[0];
   w_mean = w_mean_sum[0];
   w_mean = w_mean / w_sum;
   wmeans_arr[lc_i] = w_mean;
}
___syncthreads();
...
```

Luego se procede de forma análoga al pseudocódigo presentado en 2.2, con la diferencia principal siendo que el algoritmo calcula el aporte de cada punto a los valores de α, c y sn y los usa para obtener sus valores finales usando reducción. Es necesario sincronizar el bloque después de inicializar las variables, al terminar de calcular los aportes de cada punto, como parte de la reducción, y después de calcular el aporte del ciclo a Θ.

```
fod arg = d_freqs[fi] * time_share[i];
   fod phi = 2 * PI * (arg - floor(arg));
   zr = cos(phi);
   zi = sin(phi);
   znr = 1;
   pr = 1 / error_share[i];
   zni = 0;
   pi = 0;
   fod factor = (mag_share[i] - w_mean) / error_share[i];
   cfr = factor * cos(Nharmonics_arr[lc_i] * phi);
   cfi = factor * sin(Nharmonics_arr[lc_i] * phi);

   for (int j = 0; j < (2 * Nharmonics_arr[lc_i] + 1); j++){
      ___syncthreads();
      sn_sum[i] = powf(fabsf(pr), 2.0f) + powf(fabsf(pi), 2.0f);

      scr_sum[i] = pr * cfr + pi * cfi;
      sci_sum[i] = pr * cfi - pi * cfr;

      alr_sum[i] = (zr * pr - zi * pi) / error_share[i];
      ali_sum[i] = (zr * pi + zi * pr) / error_share[i];
      ___syncthreads();
```

```
            for(int size = NUM_THREADS/2; size > 0; size/=2){
                if(idx < size && (idx + size < num_pts_arr[lc_i])){
                    sn_sum[idx] += sn_sum[idx + size];

                    scr_sum[idx] += scr_sum[idx + size];
                    sci_sum[idx] += sci_sum[idx + size];

                    alr_sum[idx] += alr_sum[idx + size];
                    ali_sum[idx] += ali_sum[idx + size];
                }
                ___syncthreads();
            }
            if (idx == 0) {
                sn = sn_sum[0];

                scr = scr_sum[0];
                sci = sci_sum[0];

                alr = alr_sum[0];
                ali = ali_sum[0];

                if (sn < 1e-9) sn = 1e-9;
                alr = alr / sn; ali = ali / sn;
                aov += (powf(fabsf(scr), 2.0f) + powf(fabsf(sci), 2.0f))/sn;
            }
            ___syncthreads();
            fod sr, si;
            fod tmp;
            sr = alr * znr - ali * zni;
            si = alr * zni + ali * znr;
            tmp = pr * zr - pi * zi - sr * pr - si * pi;
            pi = pr * zi + pi * zr + sr * pi - si * pr;
            pr = tmp;
            tmp = znr * zr - zni * zi;
            zni = zni * zr + znr * zi;
            znr = tmp;
        }
```

Finalmente, el primer thread calcula el valor final de Θ, y se coloca en el arreglo que contiene el resultado del periodograma para todas las curvas.

```
        if (idx == 0) {
            fod d1 = 2 * Nharmonics_arr[lc_i];
            fod d2 = num_pts_arr[lc_i] - Nharmonics_arr[lc_i] * 2 - 1;
            d_per[lc_i * num_freqs + fi] = d2 / d1 * aov / max(wvar - aov, 1e-9);
        }
```

4.3. Promediado de subarmonicos

El periodograma entrega el arreglo **thetas** que contiene los valores de $\Theta(\Omega)$ para cada curva, el cual puede ser usado para encontrar los mayores locales más relevantes. Para esto, se usa la función **argrelextrema** del paquete Scipy, que encuentra los índices para los cuales Θ es mayor que sus vecinos a cierta distancia. Estos máximos se ordenan según su valor asociado de Θ y la función entrega el arreglo con los índices de las frecuencias que representan señales significativas para la curva de luz. Esta función se llama para todas las curvas y los resultados se colocan en un arreglo que contiene el índice de todas las señales significativas, ordenadas por su valor en el periodograma.

```
def find_local_maxima(theta, n_local_optima=10, order=2):
    local_optima_index = argrelextrema(theta, np.greater, order=order)[0]
    if(len(local_optima_index) < n_local_optima):
        print("Warning: Not enough local maxima found in the periodogram")
    # Keep only n_local_optima
    best_local_optima = local_optima_index[np.argsort(theta[local_optima_index])][::-1]
    if n_local_optima > 0:
        best_local_optima = best_local_optima[:n_local_optima]
    else:
        best_local_optima = best_local_optima[0]
    return best_local_optima
def get_significant_signals(thetas, n_significant_signals = 10, order=2,
    ↪ signal_diff_tolerance = 0.2, sign=-1, spacing=100):
    significant_signals_arr = []
    for theta in thetas:
        local_optima_index = find_local_maxima( -sign * theta, n_significant_signals, order)
        significant_signals_arr.append(local_optima_index)
    return significant_signals_arr
```

Este arreglo permite identificar señales reales como se describió en 3.3, iterando sobre las señales significativas de todas las curvas de luz. Primero se identifica si es que hay alguna otra señal significativa en la mitad de la frecuencia, si no lo hay entonces la señal es falsa y se ignora. En el caso contrario, se le asigna un puntaje igual al promedio del valor de Θ en la frecuencia de la señal y de su primer subarmónico. Si es que este puntaje está en cierto rango de tolerancias ajustable, entonces esta señal se identifica como asociada a la frecuencia real y se pasa a la siguiente curva. Se pueden ignorar las siguientes señales pues como están ordenadas son más débiles que la primera señal real.

En caso de que ninguna de las señales significativas pasen esta prueba, entonces se elige como frecuencia real la primera de la lista. Es importante notar que al asignarle un puntaje a la frecuencia elegida por el periodograma, este se puede usar como característica para el clasificador en ves de tener un corte dado por el usuario.

```
    def get_best_indices(significant_signals_arr, thetas, tol, freqs_test, score_tol_lower=0,
    ↪ score_tol_upper=np.inf):
    best_indices = []
    for lc_i, significant_signals in enumerate(significant_signals_arr):
        scores = []
        for s_i, signal_index in enumerate(significant_signals):
            is_freq_mult = np.abs(freqs_test[signal_index] / 2 - freqs_test[significant_signals])
```

```
    ↪ < tol
            significant_signals_index = np.where(is_freq_mult)[0]
            if len(significant_signals_index) > 0:
                theta_freq_mult = thetas[lc_i][significant_signals[significant_signals_index[0]]]
                theta_freq = thetas[lc_i][signal_index]
                score = (theta_freq + theta_freq_mult)/2
                if score_tol_lower < score < score_tol_upper:
                    # this is the correct signal
                    best_indices.append(signal_index)
                    break
            if s_i == len(significant_signals) - 1:
                best_indices.append(significant_signals[0])
    return best_indices
```

Capítulo 5

Resultados

En esta sección se detallan los resultados de la validación y evaluación de los algoritmos, y se describe la comparación de rendimiento entre ellos. Todas las pruebas fueron realizadas en un computador con un procesador Intel Core i7-9750H de 12 núcleos lógicos y una GPU Nvidia GeForce 1660 Ti con 6GB de memoria.

5.1. Validación de GMHAOV

GMHAOV se validó usando el generador de curvas de luz incluido con MHAOV, y se calculó la suma de los cuadrados de las diferencias relativas entre los resultados de MHAOV y GMHAOV divido por la cantidad de curvas de luz y la cantidad de frecuecias relativas para obtener el error relativo cuadrático promedio por curva por valor de Θ. Con 100 curvas de 150 puntos y 700 frecuencias de prueba, este valor corresponde a 5.58×10^{-9}, lo cual se encuentra dentro de los rangos aceptables.

Usando 1000 curvas de luz de RR Lyrae con las que se evaluó inicialmente MHAOV y GCE, como se describió en el capítulo 2.4 y 700 curvas de luz, se obtuvo un valor de 1.51, lo que indica un error promedio de cerca de un 120 % al calcular su raiz cuadrada. Esto es considerable, y al inspeccionar en qué valores hay una diferencia problemática, y se observa que hay ciertas curvas de luz problemáticas donde la diferencia entre ambos algoritmos es considerablemente mayor que para otras curvas, pero estas curvas problemáticas no presentan ningúna diferencia inmediatamente evidente del resto, ni en su forma ni en su cantidad de puntos, por lo que sería necesario estudiar este problema a fondo.

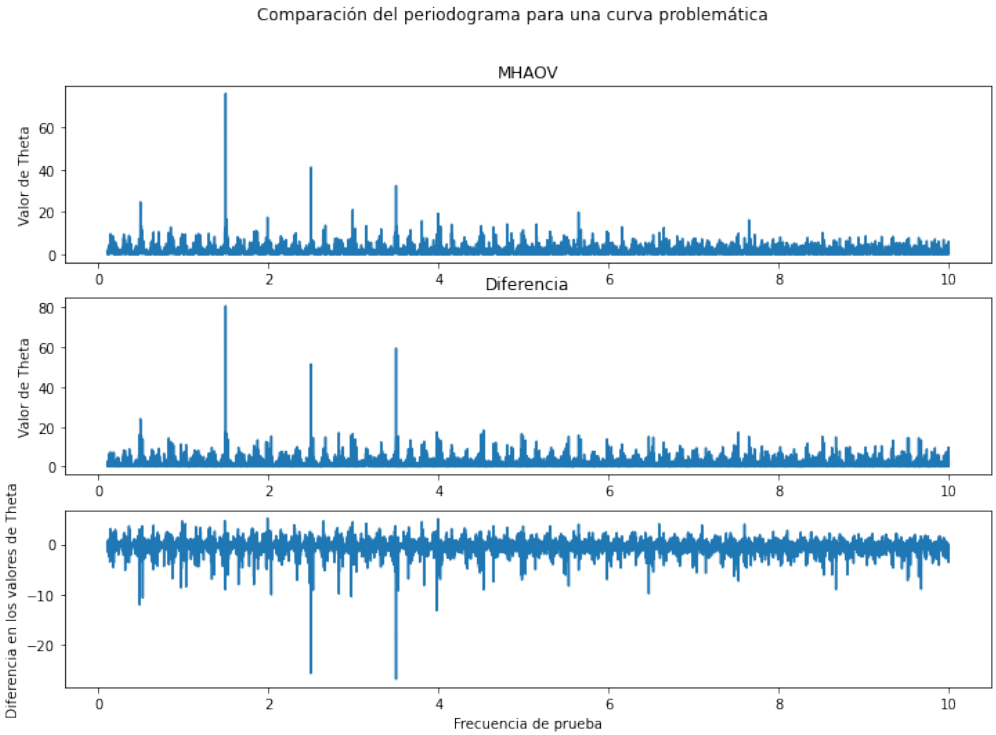

Figura 5.1: Resultado del periodograma para MHAOV (Arriba) y GMHAOV (Centro), y la diferencia entre ambos (Abajo).

En la figura 5.1 se pueden apreciar que la diferencia entre ambos periodogramas es notable, en especial en los peaks que están desplazados levemente, pero no es suficiente como para cambiar a las principales señales significativas, y de hecho al comparar la precisión de ambos algoritmos, esta solamente disminuye desde 80 % a 79 % para RR Lyrae.

5.2. Efecto de el promediado de subarmónicos

Se evaluó el efecto en la precisión del GMHAOV y GCE al introducir el promediado de subarmónicos usando los conjuntos de datos mencionados en 2.4.

5.2.1. Promediado de subarmónicos

5.2.1.1. RR Lyrae

En la tabla 5.1, se observa que si bien PS logra obtener el periodo correcto para algunos objetos, los objetos para los que ya no se tiene el periodo correcto son casi el doble para ambos algoritmos. De la tabla 5.2 se puede notar que al aplicar PS en GCE, la cantidad de submúltiplos se reduce casi en un 50 %, pero la cantidad de múltiplos casi se triplica, mientras que en GMHAOV los múltiplos incrementan levemente mientras que los submúltiplos se eliminan completamente.

Tabla 5.1: Diferencias en la precisión de GCE y GMHAOV al aplicar promediado de subarmonicos (PS).

Prueba	Ambos aciertan [%]	Acierta con PS [%]	Deja de acertar con PS [%]	No acierta nunca [%]
GMHAOV	90.15	0.10	0.20	9.55
GCE	52.10	3.70	6.70	37.50

Tabla 5.2: Detalles de las diferencias en la precisión de GCE y GMHAOV al aplicar promediado de subarmonicos (PS).

Prueba	Match	Multiplo	Submultiplo	Alias	Other
GMHAOV	90.35	0.75	0.10	1.45	7.35
GMHAOV con PS	90.25	0.95	0.00	1.45	7.25
GCE	58.80	2.10	4.45	3.60	31.05
GCE con PS	55.80	5.90	2.45	4.05	31.80

5.2.1.2. Binarias eclipsantes

De las tablas 5.4 y 5.3, se observa que en este caso no es beneficioso considerar el PS

Tabla 5.3: Detalles de las diferencias en la precisión de GCE y GMHAOV al aplicar promediado de subarmonicos (PS) para binarias elipsantes.

Prueba	Match	Multiplo	Submultiplo	Alias	Other
GMHAOV	0.20	93.55	0.00	0.00	6.55
GMHAOV con PS	0.20	92.55	0.00	0.00	7.25
GCE	9.85	44.40	0.30	0.95	44.50
GCE con PS	7.75	45.56	0.10	0.80	45.85

Tabla 5.4: Diferencias en la precisión de GCE y GMHAOV al aplicar promediado de subarmonicos (PS) para binarias elipsantes.

Prueba	Ambos aciertan [%]	Acierta con PS [%]	Deja de acertar con PS [%]	No acierta nunca [%]
GMHAOV	0.20	0.00	0.00	99.80
GCE	7.2	0.55	2.65	89.60

5.2.2. Efecto de el promediado de armónicos

Como se observó en algunos periodogramas, usualmente hay una señal significante en el doble de la frecuencia en vez de la mitad, así que se decide probar el promediado con la medida de confianza en el doble de la frecuencia, y le denominaremos a esto promediado de armónicos (PA).

5.2.2.1. RR Lyrae

De la tabla 5.5, se reduce la cantidad de múltiplos como es de esperar, pero el efecto negativo del PA lleva a su efecto neto sea negativo.

Tabla 5.5: Detalle de la precisión para GMHAOV y GCE al aplicar PA para RR Lyrae.

Prueba	Match	Multiplo	Submultiplo	Alias	Other
GMHAOV	90.35	0.75	0.1	1.45	7.35
GMHAOV con PA	90.40	0.25	0.20	1.75	7.40
GCE	58.80	2.10	4.45	3.60	31.05
GCE con PA	57.95	1.85	5.95	3.50	30.75

5.2.2.2. Binarias Eclipsantes

En el caso de las binarias eclipsantes, al aplicar PA la precisión incrementa más de 10 veces para GMHAOV y más de 3 veces para GCE. La cantidad de múltiplos se reduce significativamente, sin aumentar proporcionalmente la cantidad de submúltiplos.

Tabla 5.6: Detalle de la precisión para GMHAOV y GCE al aplicar PA para binarias eclipsantes.

Prueba	Match	Multiplo	Submultiplo	Alias	Other
GMHAOV	0.20	93.25	0.00	0.00	6.55
GMHAOV con PA	3.30	89.96	0.00	0.20	6.90
GCE	9.85	44.40	0.30	0.95	44.50
GCE con PA	32.30	21.85	0.85	1.40	43.60

5.3. Comparación de rendimiento

Para la evaluación del rendimiento, se usó el generador de curvas de luz incluido en el repositorio de MHAOV, que permite generar curvas de luz con ruido simulado y una determinada cantidad de puntos fácilmente. Se evaluó el rendimiento de MHAOV, GMHAOV, GCE y el impacto el promediado de subarmónicos en el tiempo de ejecución de GMHAOV

para $N = 10, \ldots, 10^4$ curvas de luz con 150 puntos cada una, el promedio para ALeRCE y una resolución de frecuencias de fres $= 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}$ en un rango de 0.7. MHAOV se lanza dividiendo las curvas en 8 y ejecutandolo de forma paralela en CPU.

Para cada combinación de valores de prueba, se mide el tiempo de ejecución de MHAOV, GMHAOV, GCE y el promediado de subarmónicos, y los resultados se encuentran en la figura 5.2. El tiempo de ejecución por curva se encuentra en la figura 5.3

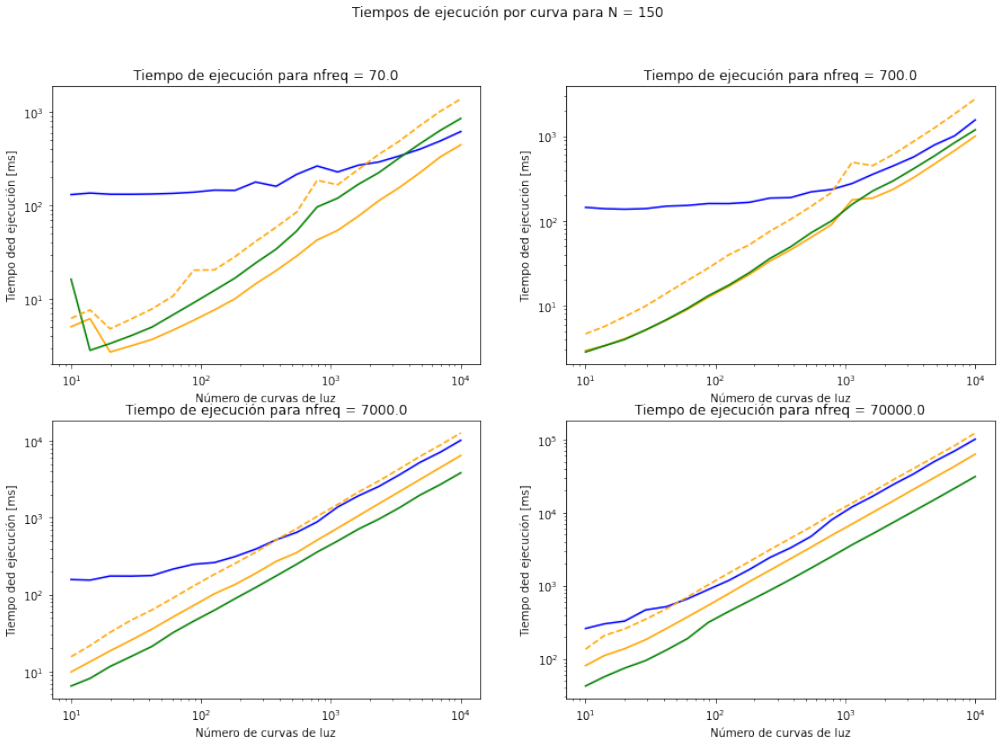

Figura 5.2: Resultados de las pruebas para cada combinación de frecuencias y número de curvas de luz simultaneas. La linea punteada representa el tiempo de ejecución de MHAOV sumado al tiempo de ejecución del promediado de subarmónicos.

Incluso para resoluciones muy bajas de frecuencia, el tiempo de ejecución de los algoritmos paralelos es significativamente menor a $MHAOV$ para una cantidad de curvas menor a 10^4, pero para resoluciones más cercanas a las usadas en la práctica, MHAOV nunca tiene un tiempo de ejecución mejor que GMHAOV y GCE, y GMHAOV se ejecuta más rápido que GCE. Sin embargo, si se le suma el tiempo de ejecución del promediado de subarmónicos, el tiempo de ejecución de GMHAOV es comparable al de su versión secuencial (ver Figura 5.2).

Esta tendencia queda más clara con la figura **??**, donde para resoluciones de frecuencia más bajas el tiempo por curva de MHAOV llega a ser mejor que el de GCE, pero a medida que aumenta la resolución de frecuencia la eficiencia por curva de MHAOV se hace cada vez menor que la de los algoritmos en GPU, con GMHAOV representando una mejora sustancial del tiempo de ejecución por curva.

Es importante destacar, sin embargo, que el tiempo de ejecución en GPU y del promediado de subarmónicos depende de la cantidad de curvas de una forma casi lineal, y tiene una dependencia más fuerte de la cantidad de frecuencias de prueba, como se observa en la figura 5.4. El tiempo de ejecución por curva por frecuencia de prueba de GCE disminuye con mayor rapidez que GMHAOV, lo que explica su mejor desempeño para una mayor resolución de

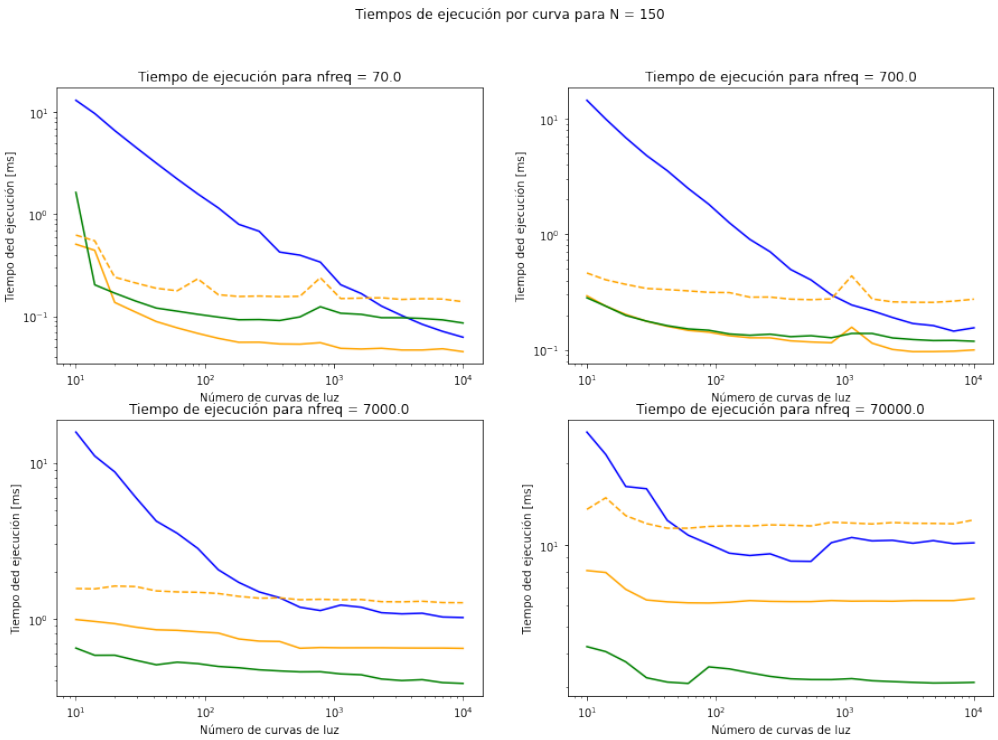

Figura 5.3: Tiempo de ejecución por curva para los algoritmos, incluyendo el tiempo de GMHAOV junto con el promediado de subarmónicos.

frecuencias.

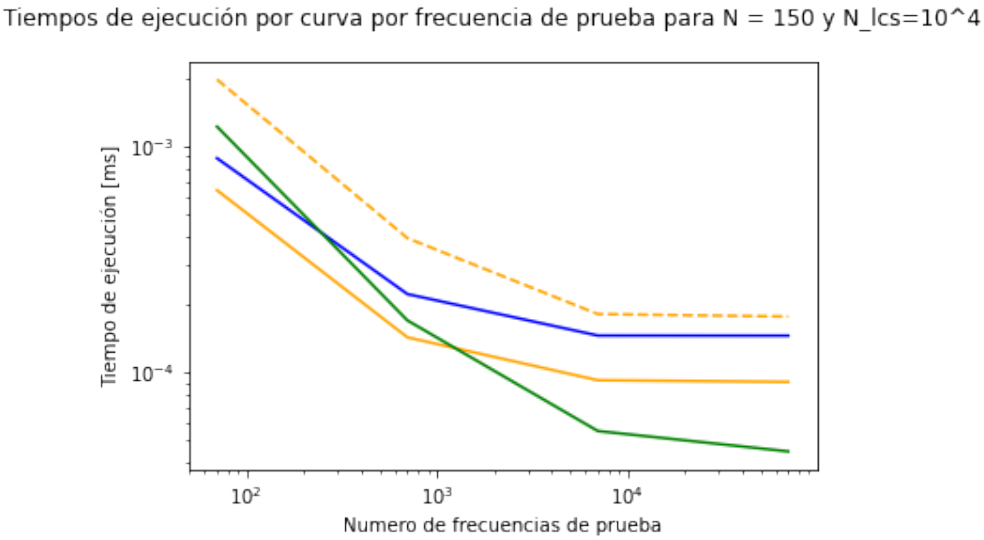

Figura 5.4: Variación del tiempo de ejecución por curva por frecuencia de prueba en función de la cantidad de frecuencia de pruebas para $N = 10^4$

Capítulo 6

Análisis y conclusión

En este capitulo se hacen conclusiones a partir de los resultados, y se discute el trabajo futuro.

6.1. Análisis de los resultados

Es de gran importancia determinar que características de las curvas de luz problemáticas descritas en 5.1 causan la diferencia en los periodogramas obtenidos por MHAOV y su versión en GPU. Sin embargo, el efecto de esta diferencia, si bien es significativo, no cambia completamente los resultados del periodograma, al menos con los tipos de curvas con las que se hicieron pruebas.

De los resultados de la sección 5.2.2, se observa que considerar el PS o PA puede ser beneficioso en algunos casos, y se cree que si se le da el puntaje asociado a la frecuencia elegida por el periodograma al clasificador, este podrá aprender a discriminar en que situaciones es bueno considerar este puntaje y en cuales se puede ignorar. Además, por lo descrito en 5.3, GMHAOV es una mejora tan significativa que si se incluye el cálculo de estos puntajes el tiempo de ejecución será menor al de MHAOV a menos que se procesen una cantidad de curvas del orden de 10^4.

Además, de las pruebas de rendimiento se puede concluir que a pesar de que GCE es más rápido que GMHAOV, la precisión de MHAOV es mucho mejor por lo que la implementación de esta versión del algoritmo de calculo de periodos reduciría significativamente el tiempo de ejecución.

6.2. Conclusión y trabajo futuro

En este trabajo se logró implementar un periodograma en GPU capaz de encontrar el periodo de una gran cantidad de curvas de luz de forma paralela, y además se estudio un posible algoritmo que puede identificar en algunos casos que el periodograma identifica una señal falsa como la frecuencia, por lo que el objetivo general fue cumplido. Sin embargo, por términos de tiempo no lograron cumplir los objetivos 4 y 5, pues no se pudo evaluar el impacto de usar los resultados de GMHAOV y GCE, y quizás los puntajes obtenidos usando PR y PA como características del clasificador de alerce. Es difícil predecir el impacto que tendrían la inclusión de estas estadísticas en el clasificador, ya que es un algoritmo de machine learning, pero si este algoritmo logra aprender cuando darle más peso a cada una de estas

características, la mejora en la clasificación podría ser significativa para las clases relevantes.

A pesar de que el efecto de las imprecisiones en GMHAOV que no están presentes en su versión secuencial no sea severo, es importante identificar su causa pues puede serlo para objetos en los que no se realizaron pruebas y afectar severamente el clasificador. Si es que se logra encontrar la causa de los problemas mencionados, se debe llevar a cabo la implementación y testing de GMHAOV en el sistema de ALeRCE. Esto puede ser relativamente simple, ya que GMHAOV ofrece una interfaz en Python similar a la de MHAOV, pero se requeriría una validación más profunda del algoritmo.

Finalmente, debido a GMHAOV tiene un speedup significativo, estudiar el efecto del número de armónicos usados en el cálculo del periodograma tanto en su precisión como en su eficiencia puede solucionar el problema de la precisión sin un impacto muy grande en la eficiencia.

Bibliografía

[1] F. Förster, G. Cabrera-Vives, E. Castillo-Navarrete, P. A. Estévez, P. Sánchez-Sáez, J. Arredondo, F. E. Bauer, R. Carrasco-Davis, M. Catelan, F. Elorrieta, S. Eyheramendy, P. Huijse, G. Pignata, E. Reyes, I. Reyes, D. Rodríguez-Mancini, D. Ruz-Mieres, C. Valenzuela, I. Alvarez-Maldonado, N. Astorga, J. Borissova, A. Clocchiatti, D. D. Cicco, C. Donoso-Oliva, M. J. Graham, R. Kurtev, A. Mahabal, J. C. Maureira, R. Molina-Ferreiro, A. Moya, W. Palma, M. Pérez-Carrasco, P. Protopapas, M. Romero, L. Sabatini-Gacitúa, A. Sánchez, J. S. Martín, C. Sepúlveda-Cobo, E. Vera, and J. R. Vergara, "The automatic learning for the rapid classification of events (alerce) alert broker," 2020.

[2] A. Schwarzenberg-Czerny, "Fast and statistically optimal period search in uneven sampled observations," *The Astrophysical Journal*, vol. 460, apr 1996.

[3] P. Sánchez-Sáez, I. Reyes, C. Valenzuela, F. Förster, S. Eyheramendy, F. Elorrieta, F. E. Bauer, G. Cabrera-Vives, P. A. Estévez, M. Catelan, G. Pignata, P. Huijse, D. D. Cicco, P. Arévalo, R. Carrasco-Davis, J. Abril, R. Kurtev, J. Borissova, J. Arredondo, E. Castillo-Navarrete, D. Rodriguez, D. Ruz-Mieres, A. Moya, L. Sabatini-Gacitúa, and C. Sepúlveda-Cobo, "Alert classification for the alerce broker system: The light curve classifier," 2020.

[4] M. J. Graham, A. J. Drake, S. G. Djorgovski, A. A. Mahabal, and C. Donalek, "Using conditional entropy to identify periodicity," *Monthly Notices of the Royal Astronomical Society*, vol. 434, pp. 2629–2635, 07 2013.