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

PROFESORA GUÍA:
Nancy Hitschfeld Kahler

PROFESOR CO-GUÍA:
Pablo Huijse Heise

MIEMBROS DE LA COMISIÓN:
Luis Mateu Brulé
Eric Tanter

SANTIAGO DE CHILE
2022

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL
EN COMPUTACIÓN
POR: MIGUEL ANGEL SEPÚLVEDA HUENCHULEO
PROF. GUÍA: NANCY HITSCHFELD KAHLER
2022

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, o periodograma, usado en ALeRCE, MHAOV, que calcula una medida de confianza para un conjunto de frecuencias de prueba en base a la curva de luz de un objeto. Esta optimización se logra 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. La implementación en GPU de MHAOV, denominada GMHAOV, se realizó en CUDA.

La validación de GMHAOV presenta una desviación promedio del resultado de MHAOV de casi 6.3 % para datos reales, pero obtiene una desviación de solo 5.58×10^{-9} para datos generados sintéticamente. La causa de esto no se estudia a fondo, pero esta discrepancia disminuye la precisión del cálculo del periodo en solamente un 1 %.

En la paralelización, se obtuvo un speedup máximo de GMHAOV respecto a MHAOV de 1.5 para 7000 frecuencias de prueba, y de 7.5 para 700 frecuencia de prueba. Los algoritmos de postproceso que se usaron para refinar la frecuencia, que descarta frecuencias erróneas entregadas por el periodograma y encuentra la que tiene mayor probabilidad de ser la frecuencia real, 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, pero siendo contraproducente para algunos casos, como con las 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 de la refinación de periodos, 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 los algoritmos de refinación de periodos 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 de la astronomía observacional, la medición continua de la magnitud de una fuente, como por ejemplo una estrella o una galaxia, permite la construcción de su curva de luz [1]. Una curva de luz es una serie de tiempo de magnitud que permite caracterizar la variabilidad de la fuente en cuestión [2]. A partir de la curva de luz es posible extraer características físicas de la fuente, hacer predicciones sobre su comportamiento futuro y/o clasificar la fuente en alguna de las categorías conocidas. Algunas de estas tareas pueden realizarse de forma automática por medio de modelos de regresión y clasificación ajustados en base a datos, paradigma conocido como Machine Learning [3, 4].

El broker astronómico ALeRCE (Automatic Learning for Rapid Classification of Events) [5] tiene el propósito de procesar el flujo de alertas proveniente de telescopios sinópticos de rastreo, tales como el *Zwicky Transient Facility* (ZTF) [6] o el futuro *Vera C. Rubin Observatory* (VRO) con su proyecto principal el *Legacy Survey of Space and Time* (LSST) [7]. En este contexto una alerta corresponde a un cambio en brillo o posición de una fuente detectado por el telescopio en base a comparaciones con imágenes de referencia [8]. En promedio ZTF produce 5 alertas por segundo, sin embargo se espera una tasa de 350 alertas por segundo para el caso de VRO, por lo que el procesamiento realizado por un broker debe ser no sólo robusto sino también eficiente y escalable [7].

El procesamiento que realiza ALeRCE incluye el cálculo de características a partir de las curvas de luz asociadas a las alertas. Estas características son luego utilizadas como entrada para un modelo de Machine Learning que predice la clase a la cual pertenece la fuente [9]. Esta información procesada es luego distribuida a la comunidad científica permitiendo que los astrónomos filtren y seleccionen las alertas asociadas a las fuentes de su interés para enfocar sus análisis y hacerles seguimiento [5].

Una de las características más importantes para la clasificación de curvas e luz es su período [9]. En particular, el período es clave para distinguir correctamente ciertos tipos de estrellas variables pulsantes [10] y sistemas eclipsantes que presentan variaciones regulares en su magnitud [2]. Ciertamente las herramientas más ampliamente utilizadas para realizar análisis frecuencial y estimación de período son la transforma de de Fourier y la función de autocorrelación. Sin embargo estas herramientas no pueden utilizarse directamente en datos con frecuencias de muestreo irregulares, como lo son las curvas de luz astronómicas. Las técnicas de interpolación no son una opción en el caso de las curvas de luz, pues en general las diferencias de magnitud entre instantes sucesivos no cambia monótonicamente. En particular, algunos tipos de fuentes variables pueden presentar oscilaciones con períodos mucho menores a la frecuencia de muestreo promedio [11].

Por las razones mencionadas anteriormente astrónomos y matemáticos han desarrollado nuevos estimadores para realizar análisis frecuencial que son específicos para series de tiempo con muestreo irregular [12–15]. Este estimador, denominado generalmente periodograma, puede describirse como una función $\Theta(\omega)$ evaluada sobre una frecuencia de prueba ω. En general la función $\Theta(\omega)$ indica la fuerza o intensidad de un comportamiento con período ω^{-1} en una curva de luz en particular. Luego encontrar la frecuencia principal o fundamental de una curva de luz se reduce a evaluar $\Theta(\omega)$ en una grilla de frecuencias candidatas apropiada.

La diferencia central entre los periodogramas está en la definición de $\Theta(\omega)$. A grandes rasgos se puede hacer la distinción entre periodogramas basados en ajustes de mínimos cuadrados [12, 14] y periodogramas basados en la transformación conocida como *epoch folding* [13, 15]

$$\phi(t) \equiv t/T + \lfloor t/T \rfloor, \tag{1.1}$$

donde $T = \omega^{-1}$ y t es un instante de observación de una curva de luz. El resultado de aplicar esta transformación es un diagrama de fase o "curva de luz doblada". En general, si el período candidato que se usa para doblar la curva es cercano al período real de la fuente entonces el diagrama de fase presentará un patrón ordenado y representativo de la periodicidad del objeto astronómico. En caso contrario el diagrama de fase será similar a ruido sin correlación.

El periodograma utilizado actualmente en ALeRCE es el *Multiharmonic Analysis of Variance* (MHAOV) [16], el cual se basa en ajustar un modelo sinusoidal con un determinado número de armónicos sobre la curva doblada con una frecuencia ω. En este caso $\Theta(\omega)$ se interpreta como un métrica asociada a la bondad del ajuste entre el modelo sinusoidal y la curva de luz. [16]. La implementación actual del algoritmo MHAOV se ejecuta completamente en CPU. Adicionalmente se ha detectado que este algoritmo es impreciso para ciertos situaciones que se describirán en las siguientes secciones.

1.1. Motivación

El período de una curva de luz es una de sus propiedades más importantes, ya que permite diferenciar entre objetos con curvas de luz similares pero distintos rangos de períodos [11]. Adicionalmente al período de una fuente es relevante para derivar otras propiedades físicas, como su distancia a la Tierra [17] o la masa de las estrellas de un sistema binario. Por otro lado el cálculo del período es actualmente una de las rutinas computacionales más costosas del sistema ALeRCE. Por esto es de gran importancia mejorar su precisión y eficiencia.

Podemos identificar dos áreas en las que el periodograma usado por ALeRCE puede ser mejorado:

1. **Eficiencia:** El cálculo de características es el paso computacional más demandante del pipeline del sistema ALeRCE [5]. Actualmente ALeRCE procesa alertas de ZTF, sondeo que produce 1.4 TB de datos por noche. El sistema debe ser capaz de mantenerse al ritmo de este volumen de datos. Si bien el sistema actual es capaz de manejar este volumen de alertas es necesario considerar que nuevos sondeos incrementarán exponencialmente esta cantidad. Por ejemplo para el sondeo 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

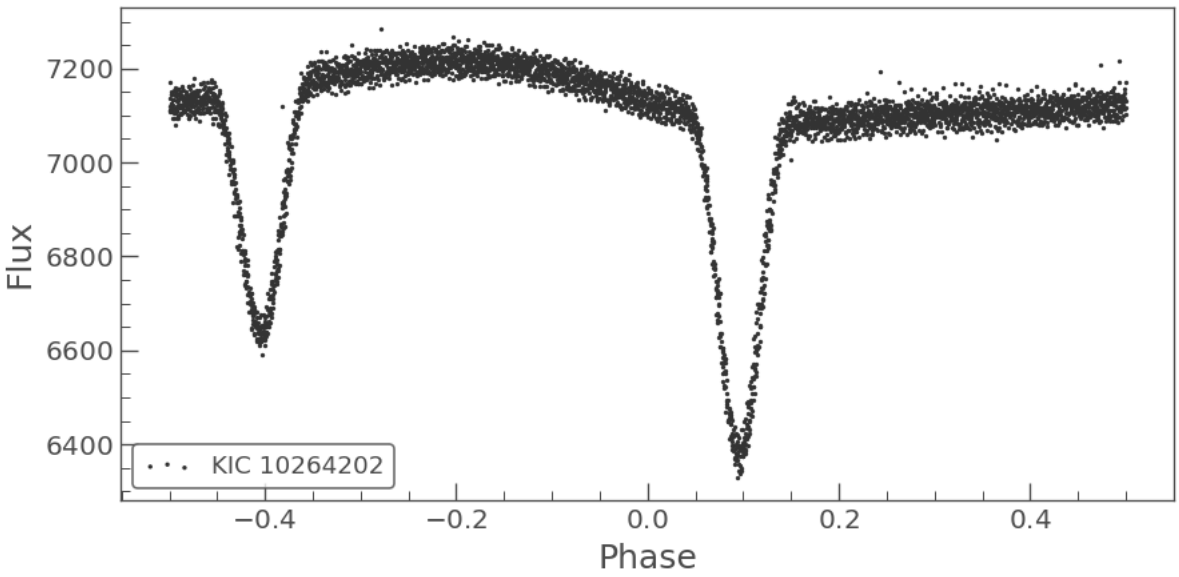

Figura 1.1: Curva de luz de una binaria eclipsante. El flujo del sistema se reduce cuando una de las estrellas oculta a la otra, pero hay una diferencia entre ambos eclipses, donde el flujo se reduce menos si la estrella que se bloquea es la menos brillante.

cálculo de características, tomando alrededor del 20 % del tiempo. Cabe destacar que la implementación de MHAOV utilizada por ALeRCE se ejecuta en CPU y se paraleliza a nivel de curvas de luz.

Sin embargo debido a que $\Theta(\omega)$ puede también calcularse para cada frecuencia de prueba de forma independiente podemos suponer que el algoritmo se podría ver beneficiado de una implementación masivamente paralela en base a GPU.

2. **Precisión:** Existen fuentes para las cuales MHAOV es particularmente impreciso. Específicamente, MHAOV calcula un periodo igual a la mitad del período real para gran parte de los objetos de tipo binaria eclipsante. Esto se debe a que MHAOV no es lo suficientemente sensible como para distinguir entre ambos eclipses (ver Fig. 1.1). 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.

   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 u otro y así mejorar la clasificación. Otra posible solución es realizar un post-proceso que pueda determinar si el resultado de un periodograma es correcto o no. La desventaja de la primera solución es el costo computacional: calcular otro periodograma podría incrementar considerablemente el tiempo de ejecución necesario para el cálculo de los períodos.

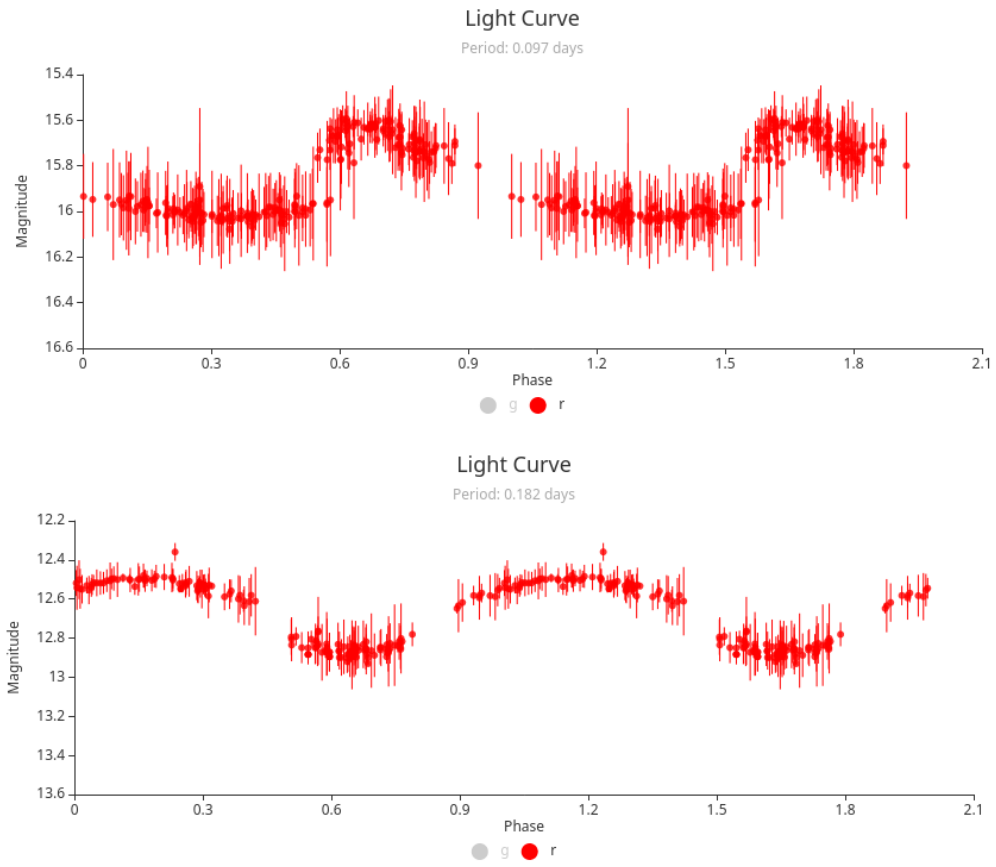

Figura 1.2: Curva de luz de una δ-Scuti con periodo de 0.097 dias (Arriba) y una binaria eclipsante con periodo de 0.182 días (Abajo). Sus curvas de luz son similares pero sus periodos son muy diferentes (Fuente: ZTF Explorer)

1.2. Objetivos

Objetivo General

Desarrollar un algoritmo paralelo en GPU que permita determinar el período 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 propusieron las siguientes metas:

1. Estudiar y analizar algoritmos para obtener el período 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 período conocido.

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

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

1.3. Metodología y resolución

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 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. [15]

- **Métodos para identificar subarmónicos:** Como se describió en la sección 1.1, periodogramas como MHAOV suelen identificar subarmónicos del periodo real para algunos objetos astronómicos, por lo que es beneficioso investigar algoritmos que puedan identificar cuando pasa esto y poder encontrar el período real.

Se tomó la decisión de implementar GMHAOV en CUDA [18], ya que se tenía experiencia con el desarrollo de algoritmos en GPU en este lenguaje. Esto además permite utilizar el código de GCE como referencia para poder ejecutar código en CUDA desde python. El diseño de GMHAOV se realizó en base a 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.

La implementación de 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 del número de frecuencias de prueba y número de curvas de luz.

Con respecto a mejorar la precisión del cálculo del período, en especial en el caso de los estrellas binarias eclipsantes, se implementó un algoritmo que identifica submúltiplos llamado promediado de subarmónicos. Este algoritmo 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 periodogramas, como el de GCE, en especial con respecto a las binarias eclipsantes.

1.4. Contenido de la memoria

- **Antecedentes:** Se explican los fundamentos de la computación en GPU, la arquitectura y su modelo de memoria.
- **Análisis del software:** Se hace un análisis detallado del software que se utilizó 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 concluye respecto a los objetivos y se discute el trabajo futuro.

Capítulo 2

Antecedentes

En este capítulo se presentan conceptos y técnicas que son necesarios para poder entender la solución presentada en este documento. Se empieza por explicar los conceptos fundamentales de la paralelización en GPU, la arquitectura usada en la solución, la manera en la que se maneja la memoria en CUDA y finalmente se habla sobre la técnica de reducción, que permite reducir un arreglo, usando suma o resta por ejemplo, de manera paralela.

2.1. Paralelización en GPU

La fortaleza de las GPU es que contienen miles de núcleos, donde cada uno puede tratar con un sub-problema pequeño, sacrificando poder de computación de un núcleo individual por poder de paralelización. Sin embargo, una desventaja de las GPU es que, debido a la gran cantidad de unidades de procesamiento, no se deja mucho espacio al caché, como se observa en la Figura 2.1 por lo que los accesos a memoria son muy costosos y se tienen que manejar eficientemente. En una CPU, en cambio, se tiene una pequeña cantidad de núcleos a cambio de que el poder de computación de un núcleo individual es mucho más alto que del de una GPU. [19]

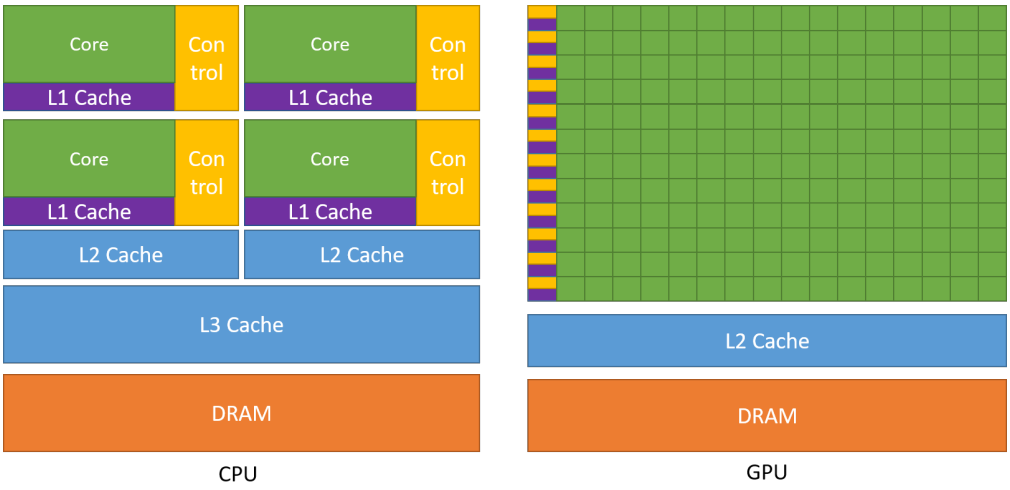

Figura 2.1: Distribución de las unidades de procesamiento y memoria en la CPU y GPU. En la GPU, las unidades de control y memoria usan mucho menos espacio que las de procesamiento en comparación con la CPU.

Un buen ejemplo de un problema que se beneficia del uso de una GPU en vez de una CPU es la suma de vectores, que es un problema al que se le dice data-paralel, pues la suma de cada componente de los vectores se puede realizar de forma paralela al resto. Este problema es ideal para la GPU pues todas las unidades de procesamiento realizan la misma tarea de baja complejidad y su acceso a la memoria al recuperar los valores a sumar es óptimo pues maximiza el ancho de banda usado.

2.2. Arquitectura de CUDA

Para la computación en GPU, las dos alternativas más populares son CUDA y OpenCL. La gran desventaja de CUDA es que es exclusivo para las tarjetas de video NVIDIA más modernas, mientras que OpenCL es compatible con una gran cantidad de tarjetas de video, e incluso puede ser ejecutado en CPU. Sin embargo, la documentación y herramientas de CUDA son mucho más extensivas que las de OpenCL. Además, el memorista cuenta con experiencia en CUDA y tanto el memorista como ALeRCE cuentan con GPUs de NVIDIA, por lo que la tecnología elegida para el desarrollo de la solución fue CUDA.

CUDA permite la creación de funciones en C++ llamadas kernel, que se ejecutan paralelamente por todos los núcleos de la GPU. Cada thread puede acceder a información sobre él mismo para determinar con qué datos hará qué computaciones. Como se observa en el Código 2.1, cada thread puede consultar su id.

Código 2.1: Kernel que suma los componentes i de los arreglos A y B y lo guarda en el arreglo C

```
1 __global__ void VecAdd(float* A, float* B, float* C)
2 {
3     int i = threadIdx.x;
4     C[i] = A[i] + B[i];
5 }
```

Cada thread puede ser identificado en un índice de threads de 1, 2 o 3 dimensiones. Los threads se organizan en bloques de las dimensiones correspondientes, permitiendo una asociación natural entre los threads y arreglos, matrices y volumenes, pero con la restricción de que el número de threads en un bloque no puede superar cierto número, el cual es 1024 para las GPUs de los últimos años, como la usada en el desarrollo y pruebas en esta memoria. Finalmente, todos los bloques de threads se organizan en una grilla de bloques, como se observa en la figura 2.2.

CUDA permite sincronización de la ejecución a nivel de bloques, deteniendo la ejecución de cada thread hasta que todos lleguen a una llamada de la función `__syncthreads()`. Esto implica una disminución en la eficiencia, pero esta sincronización a veces es necesaria para evitar data races.

Un aspecto a considerar es que las GPU siguen un modelo de mismas instrucciones, datos distintos, o SIMD (*S*ame Instructions Multiple Data). Esto significa que todos los threads deben ejecutar las mismas instrucciones, lo que se traduce en un impacto en la eficiencia en el caso de los condicionales, pues los threads deben ejecutar todas las instrucciones independiente del resultado del condicional. En el peor de los casos, la mitad de los threads desperdician tiempo en ejecutar instrucciones cuyos resultados se descartan.

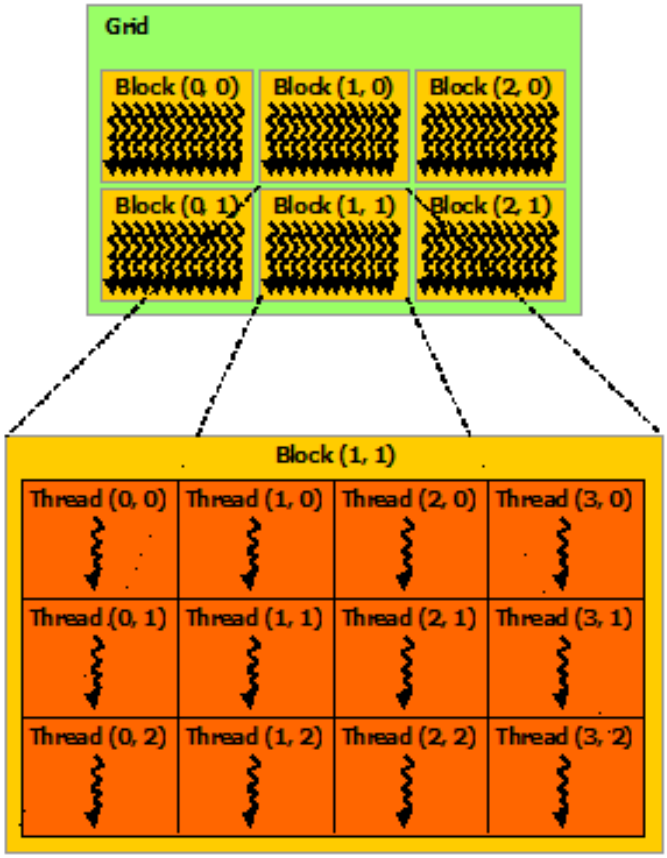

Figura 2.2: Una grilla se divide en bloques, y cada bloque se subdivide en threads.

2.3. Modelo de memoria de CUDA

En CUDA, cada thread puede acceder a distintos tipos de memoria. La memoria local de los threads es solo visible a este y es la de acceso más rápido, mientras que cada bloque tiene memoria que es visible a todos los threads del bloque. Finalmente, hay una memoria global accesible por todos los threads, pero el acceso es considerablemente más lento. La jerarquía de memoria se aprecia en la Figura 2.3.

La CPU solo es capaz de enviar datos a la memoria global, por lo que el kernel es el encargado de acceder a esta y guardar el resultado de la computación en el mismo. Como el acceso a memoria global es el más lento, este se debe realizar de forma eficiente aprovechando el ancho de banda de la GPU.

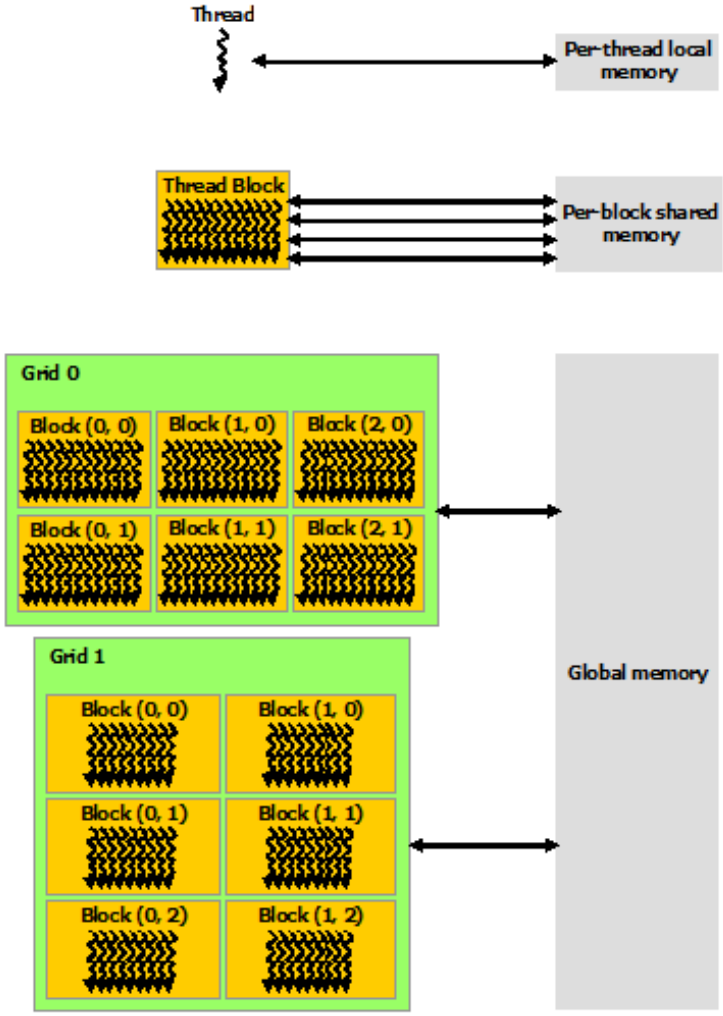

Figura 2.3: Jerarquía de memoria de CUDA.

2.4. Reducción en GPU

Una técnica de paralelización muy usada es la reducción, que, acorde a su nombre, sirve para reducir un o múltiples arreglos a un solo valor usando una operación. Un ejemplo sería sumar todos los valores de un arreglo.

En la figura 2.4 se encuentra un ejemplo de reducción para un arreglo en la memoria de un bloque. En cada paso, se van acumulando los valores por cada thread hasta que en el paso final el resultado de la reducción se encuentra en el primer elemento del arreglo. Existen otros métodos de reducción pero se usa este por ser el más simple de implementar [20].

Si es que se quiere reducir un arreglo que se procesa en varios bloques, es buena idea empezar reduciendo las partes del arreglo que corresponden a cada bloque, y operar los resultados individuales de todos los bloques en CPU, pues CUDA no ofrece una manera de sincronizar bloques.

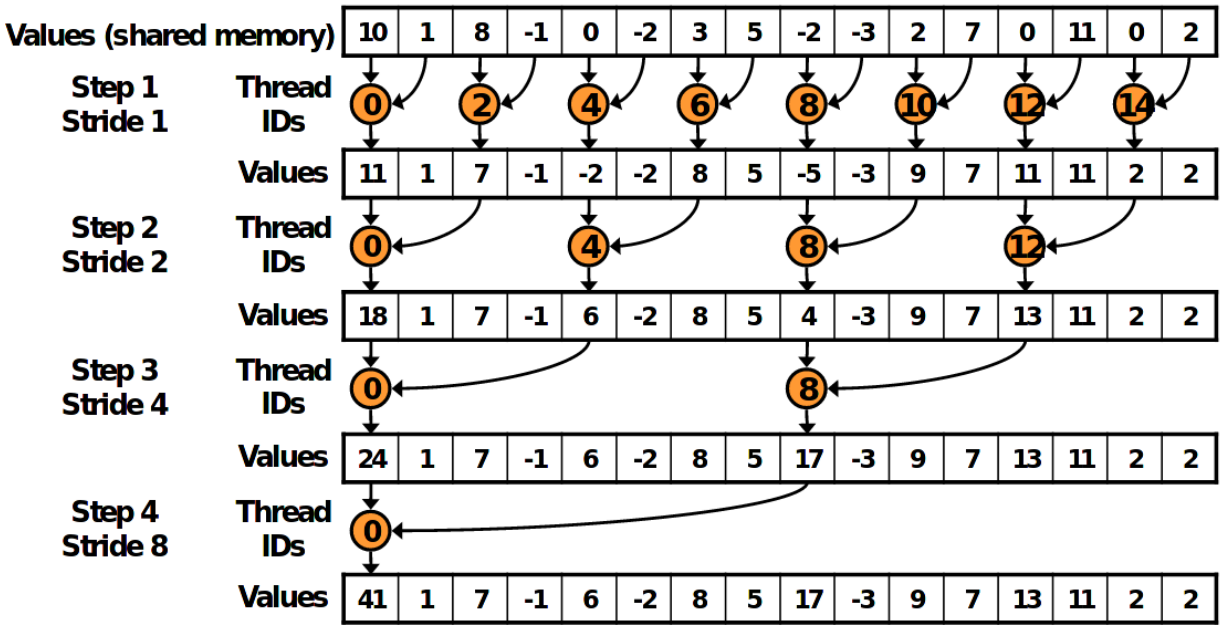

Figura 2.4: Pasos de la reducción, [20].

Capítulo 3

Análisis del software

En este capítulo, se entregan los detalles del pipeline de ALeRCE y de los algoritmos estudiados durante el curso de la memoria, es decir MHAOV y GCE.

3.1. El pipeline de ALeRCE

El sistema de ALeRCE recibe un flujo de alertas desde ZTF, que contiene varios datos de la observación como la magnitud en distintas bandas y la posición del objeto, entre otros. La Figura 3.1 muestra un esquema del pipeline de ALeRCE. A continuación se describe cada uno de los bloques diagrama:

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

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

3. **Stamp classifier:** Las imágenes de los objetos nuevos son clasificadas utilizando un modelo de aprendizaje profundo [21].

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

5. **Light curve features:** Se calculan las características para las curvas de luz con más de 6 puntos. Entre ellas se incluye el período.

6. **Light curve classifier:** Se clasifica el objeto con las características calculadas en el paso anterior utilizando un ensamble de árboles de decisión con compensación de desbalance de clases [9].

7. **Outliers:** Se aplica un algoritmo de detección de outliers (manuscrito en preparación) para identificar las curvas de luz que pueden ser particularmente interesantes y que no correspondan a las clases incluidas en el dataset de entrenamiento de los clasificadores.

8. **ALeRCE stream:** Se reportan las curvas de luz a la comunidad en un stream. Esto incluye la clasificación de las curvas de luz y las características calculadas.

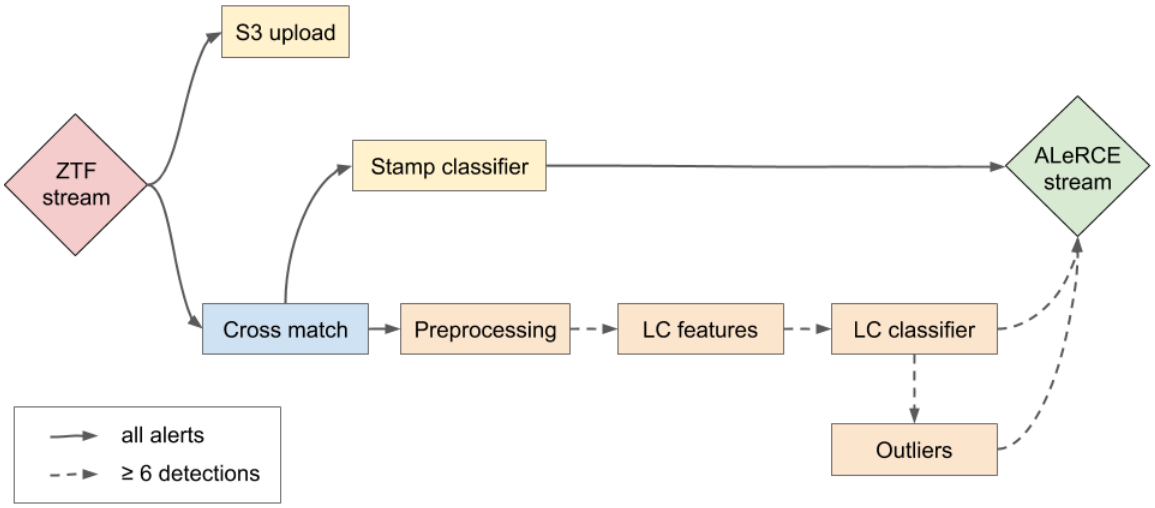

Figura 3.1: Pipeline de ALeRCE [5]. En el procesamiento realizado al stream de alertas de ZTF se encuentra el cálculo de características (LC features) a partir de las curvas de luz corregidas. El período se calcula en dicha etapa del pipeline.

3.2. Diseño de MHAOV

El algoritmo MHAOV usa como medida de confianza la calidad del ajuste de una serie trigonométrica con un determinado número de armónicos a las magnitudes de la curva de luz. El algoritmo asume 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 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 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 $\{\Phi\}_{n=1,..,N}$

$$\Psi_N(z) = \sum_{n=1}^{N} c_n \Phi_n(z), \tag{3.1}$$

donde el producto interno está definido como

$$(\Phi, \Psi) = \sum_{k=1}^{K} g_k \Phi(z_k)\overline{\Psi(z_k)} \tag{3.2}$$

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

Esta base se puede obtener de manera iterativa como se muestra a continuación

$$\tilde{\Phi}_0(z) = 1 \tag{3.3}$$

$$\tilde{\Phi}_{n+1}(z) = z\tilde{\Phi}_n - \alpha_n z^n \overline{\tilde{\Phi}_n(z)} \tag{3.4}$$

$$\Phi_n(z) = \frac{\tilde{\Phi}_n(z)}{\sqrt{(\tilde{\Phi}_n, \tilde{\Phi}_n)}} \tag{3.5}$$

$$\alpha_n = \frac{(z\tilde{\Phi}_n, \tilde{\Phi}_n}{(z^n\overline{\tilde{\Phi}_n}, \tilde{\Phi}_n)} \tag{3.6}$$

$$c_n = \frac{(\Psi, \tilde{\Phi}_n)}{\sqrt{(\tilde{\Phi}_n, \tilde{\Phi}_n)}}. \tag{3.7}$$

Para MHAOV, la medida de confianza se basa en estimadores de la varianza de F y E, y se define como $\Theta \equiv \widehat{\text{Var}}\{F\}/\widehat{\text{Var}}\{E\}$. Si consideramos la descomposición en funciones base anterior la expresión tiene la forma (Para mas detalle ver [16])

$$\Theta(\omega) = \frac{(K - 2N - 1)\sum_{n=0}^{2N} |c_n|^2}{2N[(X, X) - \sum_{n=0}^{2N} |c_n|^2]}. \tag{3.8}$$

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 (3.4) para encontrar el siguiente valor de $\tilde{\Phi}_n$
3. Usar (3.2) para calcular los valores de (3.6) y (3.7).
4. Volver al paso 2, iterando esto N veces.
5. Calcular $\Theta(\omega)$ usando 3.8.

Algorithm 1 MHAOV

$wmean \leftarrow \text{weighted\_mean}(mag, err, N)$ ▷ 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 $(K - 2N - 1) \cdot \Theta/(2N \cdot \max(wvar - \Theta, 10^{-9}))$

3.3. Diseño de GCE

El algoritmo de Conditional Entropy (CE) se basa en determinar si existe una correlación entre la magnitud y la fase para una curva doblada de acuerdo a cierta frecuencia. El algoritmo empieza por normalizar los valores de la magnitud y doblar la curva respecto a un periodo p, de forma que los valores de magnitud $m(\phi_i)$, con ϕ definido de acuerdo a la ecuación 1.1, se encuentren en un cuadrado unitario en el espacio (m, ϕ) que se particiona en k secciones en magnitud y en l secciones en fase. Esto se hace para poder calcular la entropía condicional H_c, que se define como

$$H_c = \sum_{i,j} p(m_i, \phi_j) \ln \left(\frac{p(\phi_j)}{p(m_i, \phi_j)} \right) \tag{3.9}$$

donde $p(m_i, \phi_j)$ es la probabilidad de ocupación para la i-ésima partición en magnitud y la j-ésima partición en fase, y $p(\phi_j)$ es la probabilidad solo para la j-ésima partición en fase. Para GCE, se usan particiones rectangulares y la probabilidad de ocupación se calcula contando la cantidad de puntos por partición y dividiendo esta cantidad por la cantidad total de puntos [15].

Como el cálculo de H_c se realiza para cada frecuencia de prueba y para cada curva, GPU-Accelerated Conditional Entropy (GCE) paraleliza este proceso, calculando una combinación de curva y frecuencia de prueba para cada thread de la GPU. GCE se puede encontrar en este repositorio .

3.4. Comparación de MHAOV vs GCE

Se realizó una comparación de MHAOV y GCE usando sets etiquetados, que son archivos con información sobre objetos, incluyendo su clase y periodo. El propósito de esta composición fue evaluar el impacto de crear un periodograma con la precisión de MHAOV pero con un tiempo de ejecución comparable con el de GCE.

Los tests realizados midieron el tiempo de ejecución y la precisión de cada algoritmo usando curvas de luz con mas de 20 puntos, 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 en CPU, mientras que GCE fue ejecutado en GPU.

Para el primer test, se usaron alrededor de 8500 estrellas RR Lyrae, que tienen curvas suaves y periodos marcados, y en las tablas 3.1 y 3.2 se encuentran los resultados.

Tabla 3.1: 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 3.2: Clasificación en porcentajes para los valores del periodo calculado respecto al real. Acierto significa que estos son 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 3.3. Notemos que si multiplicamos los periodos obtenidos por 2, obteniendo el periodo original si es que se había obtenido la mitad de este inicialmente, los resultados cambian a los resultados de la Tabla 3.5, pero aún así observamos cierto grado de contaminación, y al hacer esto estamos obteniendo el periodo equivocado para binarias eclipsantes donde la diferencia entre los eclipses es muy significativa.

Tabla 3.3: 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 3.4: 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 3.5: 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

De las Tablas 3.1 a 3.5, notamos que GCE es más rápido pero significativamente menos preciso que MHAOV. Esto incluye el caso de submúltiplos, donde se esperaba que GCE redujera dichos casos puesto que es un algoritmo que no ajusta un serie de Fourier. Al no ajustar una serie trigonométrica se espera que su desempeño sea superior en curvas de luz con formas no sinusoidales como lo son las estrellas binarias eclipsantes.

La Tabla 3.4 muestra que el porcentaje de curvas de luz de GCE acierta y MHAOV falla es del 3.4 % para RR Lyrae y 0.60 % para las estrellas binarias eclipsantes despúes de ajustar el periodo. Este último valor pudiera ser significativo debido la gran cantidad de binarias eclipsantes que existen. De estas pruebas se concluye que no es adecuado simplemente reemplazar MHAOV por GCE en ALeRCE, ya que se perderá demasiada precisión. Por otro lado implementar una versión paralela de MHAOV, que sean tanto o más rápida como GCE, será claramente beneficioso.

Capítulo 4

Diseño

A continuación se describe 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.

4.1. 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. 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.

4.2. 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 3.2 se observa que los ciclos que iteran por el largo de la curva de luz también se pueden paralelizar.

En el Algoritmo 2 se presenta en pseudocódigo una paralelización de MHAOV. En las lineas 2 a 3 se inicializa el valor de Θ, y luego se calcular el valor de $w$mean y $w$mvar, que requieren sumar los aportes de cada punto de la curva de luz a ellos, usando reducción. Esto requiere sincronizar el bloque en la linea 6 pues el valor de $w$mvar depende del valor de $w$mean. Para el primer ciclo de las lineas 6 a 16 del Algoritmo 1, se inicializa el valor de cada arreglo por hilo en la linea 9 del algoritmo **??**, el ciclo sobre j de MHAOV de la linea 17 del Algoritmo 1 no se paraleliza pues se ejecuta no más de 10 veces en la práctica y los resultados de un paso dependen del anterior. Para el ciclo de la linea 19 del Algoritmo 1, se puede calcular el aporte de cada paso a los valores de las variables, y luego utilizar reducción para sumar estos aportes y usar los resultados para calcular el incremento en Θ luego de

sincronizar el bloque para no obtener valores indeterminados. Todo esto corresponde a las lineas 12 a 19 del Algoritmo 2. Para el ciclo de la linea 30 del Algoritmo 1, simplemente se actualizan los valores que corresponden a cada hilo, ya que ellos son independientes entre sí.

Notar que es necesario coordinar los threads en ciertos puntos del kernel para evitar data races. A continuación se presenta el pseudo-código del kernel de GMHAOV, siendo análogo al código presentado en la Sección 3.2:

Algorithm 2 GMHAOV

1: **function** GMAHOVKernel(lc_i, ω_i, i)
2: **if** $i == 0$ **then**
3: $\Theta \leftarrow 0$
4: **end if**
5: $wmean \leftarrow$ cálculo paralelo de los aportes de cada punto y suma por reducción
6: ___syncthreads()
7: $wmvar \leftarrow$ cálculo paralelo de los aportes de cada punto y suma por reducción
8: ___syncthreads()
9: $z_r,\ z_i,\ z_n,\ p_r,\ p_i,\ cf_r,\ cf_i \leftarrow$ valores inciales según $mjd[lc_i, i]$, $err[lc_i, i]$ y $mag[lc_i, i]$
10: ___syncthreads()
11: **for** $j \leftarrow 0$ to $2K + 1$ **do**
12: $al_r,\ al_i,\ sc_r,\ sc_i,\ sn \leftarrow$ suma de los aportes de cada punto por reducción
13: ___syncthreads()
14: **if** $i == 0$ **then**
15: $\Theta \leftarrow \Theta + ({sc_r}^2 + {sc_i}^2)/sn$
16: **end if**
17: $s_r,\ s_i,\ p_r,\ p_i,\ zn_i,\ zn_r \leftarrow$ Valores actualizados usando $al_r,\ al_i,\ zn_r,\ zn_i,\ p_r,\ p_i,\ z_r$ y z_i
18: ___syncthreads()
19: **end for**
20: $\Theta \leftarrow (K - 2N - 1) \cdot \Theta/(2N \cdot \max(wvar - \Theta, 10^{-9}))$
21: **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 acceder a memoria global sólo 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.

4.3. Promediado de subarmónicos

El promediado de subarmónicos es descrito en [15], pero el código no está disponible así que se tuvo que implementar 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 máximos 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 significativas.

Se establece un criterio simple para determinar si una señal significativa es real: Si existe otra señal significativa 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 en el clasificador.

Capítulo 5

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.

5.1. 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, la cantidad de curvas de luz para las cuales se calculará el periodograma y el tamaño de lote en el que se dividirán las curvas de luz para no sobrecargar la memoria, como se observa en el Código 5.1. Se ignoran varias funciones de GCE que no se reimplementaron en GMHAOV, ya que las interfaces que ofrecen son compatibles con cualquier periodograma, como por ejemplo la función que encuentra el máximo o mínimo del mismo.

Código 5.1: Primera parte de la inicialización

```
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 = 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)
```

```
22 ...
```

Antes de realizar el cálculo del periodograma es necesario determinar el largo de cada curva de luz y el largo máximo entre todas las curvas de luz usando el ciclo de la linea 6 del bloque de código 5.2. Estos valores son necesarios para determinar el númedo de threads de cada bloque, pues estos deben tener a lo menos un thread por punto de la curva de luz. Luego en las lineas 17 a 21 se crean arreglos para los tiempos, magnitudes y errores para las curvas de luz, donde cada arreglo tiene un largo equivalente al de la curva más larga, y las entradas adicionales se rellenan con ceros.

Originalmente, GCE realizaba otros cálculos relacionados al binning de las magnitudes y los tiempos, además de considerar los valores de la derivada del periodo $\dot{p}$, pero esto fue removido ya que MHAOV no es un algoritmo incluya el cálculo de este valor, y el código fue adaptado para funcionar sin esto.

Código 5.2: Segunda parte de la inicialización

```
1
2  # determine maximum length of light curves in batch
3  # also find minimum and maximum magnitudes for each light curve
4  max_length = 0
5  number_of_pts = np.zeros((len(light_curve_split),)).astype(int)
6  for j, lc in enumerate(light_curve_split):
7      number_of_pts[j] = len(lc)
8      max_length = max_length if len(lc) < max_length else len(lc)
9  light_curve_arr = np.zeros((len(light_curve_split), max_length, 3))
10 logging.info(f"Number of points {number_of_pts}")
11
12 # populate light_curve_arr
13 for j, lc in enumerate(light_curve_split):
14     light_curve_arr[j, : len(lc)] = np.asarray(lc)
15
16 # separate time and mag info
17 light_curve_times = light_curve_arr[:, :, 0]
18
19 light_curve_mags = light_curve_arr[:, :, 1]
20
21 light_curve_errs = light_curve_arr[:, :, 2]
```

Luego en la lineas 2 a 10 del Código 5.3 se aplanan los arreglos para formar tres arreglos que contienen los datos de todas las curvas, y se cargan en la memoria de la GPU usando la función de cupy `xp.asarray` junto con el arreglo que contiene 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, el cual por ahora se inicializa con solo unos. En la linea 24 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.

Código 5.3: Tercera parte de la inicialización

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

```
4         )
5
6         light_curve_mags_in = xp.asarray(light_curve_mags.flatten()).astype(
7             self.dtype
8         )
9
10        light_curve_errs_in = xp.asarray(light_curve_errs.flatten()).astype(
11            self.dtype
12        )
13        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 linea 36 del Código de 5.3 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.

Código 5.4: Wrapper en Cython

```
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
```

```
    cdef size_t mags_in = mags
    cdef size_t mjds_in = mjds
    cdef size_t errs_in = errs
    cdef size_t num_pts_arr_in = num_pts_arr
    cdef size_t Nharmonics_arr_in = Nharmonics_arr
    cdef size_t num_lcs_in = num_lcs
    cdef size_t num_pts_max_in = num_pts_max
    run_gmhaov(<fod *> per_in, <fod *> freqs_in, num_freqs_in, <fod *> mags_in, <fod *>
     ↪ mjds_in, <fod *> errs_in,
                        <int *> num_pts_arr_in, <int *> Nharmonics_arr_in, num_lcs,
     ↪ num_pts_max_in, <float *> wmeans_arr_in, <float *> wvars_arr_in)
```

5.2. Paralelización

Como la memoria local usada por cada bloque es variable para cada ejecución, y los arreglos relevantes ya están en la memoria de la GPU, `run_gmhaov` es una función relativamente corta, pues no necesita reservar el espacio que cada arreglo usará en memoria local ni mover objetos desde la memoria del host a la del dispositivo. El tamaño de la grilla, determinado en la linea 3 del Código 5.5, se escoge como el número de frecuencias en el eje x y el número de curvas de luz en el eje y. La memoria de cada bloque, calculada en la linea 6, es el espacio que ocupará cada curva de luz más los arreglos necesarios para hacer las reducciones descritas en la Sección 2.1.

Como siempre es conveniente tener potencias de 2 como tamaño de bloque, se aproxima el largo máximo de las curvas de luz a su potencia de 2 superior más cercana y se elije esta como tamaño de bloque, considerando un valor máximo de 1024. El objetivo es que cada bloque pueda procesar completamente cada curva de luz. Para curvas de luz con más de 1024 puntos el kernel podría procesarla completamente, pero en esta implementación se asume que todas las curvas tienen menos de 1024 puntos.

Código 5.5: Wrapper del kernel

```
void run_gmhaov(fod *d_per, fod *d_freqs, int num_freqs, fod *d_mag, fod *d_mjd, fod *
 ↪ d_err,
          int *num_pts_arr, int *Nharmonics_arr, int num_lcs, int num_pts_max, float
 ↪ *wmeans_arr, float *wvars_arr){
    dim3 griddim(num_freqs, num_lcs, 1);

    // determine shared memory allocation size
    size_t numBytes = sizeof(fod)*num_pts_max + // magnitude values
                      sizeof(fod)*num_pts_max + // error values
                        sizeof(fod) * num_pts_max + // time values
                        5 * sizeof(fod) * num_pts_max + // sn, scr, sci, alr, ali to sum
                        3 * sizeof(fod) * num_pts_max; // wvar, wmean, w_sum to sum

    int v = num_pts_max;
    // Source for rounder: https://graphics.stanford.edu/~seander/bithacks.html#
     ↪ RoundUpPowerOf2
    v--;
    v |= v >> 1;
```

```
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    v++;
    if (v > 1024) v = 1024;
    kernel<<<griddim, v, numBytes>>>(d_per,
                                     d_freqs, num_freqs,
                                     d_mag, d_mjd,
                                     d_err, num_pts_arr,
                                     Nharmonics_arr, num_lcs,
                                     num_pts_max, wmeans_arr, wvars_arr);
    cudaDeviceSynchronize();
    gpuErrchk(cudaGetLastError());
}
```

En el Código 5.6, 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 define 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.

Código 5.6: Inicialización de memoria en el kernel del algoritmo

```

__global__ void kernel(fod* __restrict__ d_per,
                       fod* d_freqs, int num_freqs,
                       fod* d_mag, fod* d_mjd,
                       fod* d_err, int* num_pts_arr,
                       int* Nharmonics_arr, int num_lcs,
                       int num_pts_max, fod* wmeans_arr, fod* wvars_arr)
    {
    /* Assign the magnitude,error and time values to sections of the shared array */
    extern __shared__ fod mag_share[];
    fod *error_share = (fod*) &mag_share[num_pts_max];
    fod *time_share = (fod*) &error_share[num_pts_max];

    fod *sn_sum = (fod*) &time_share[num_pts_max];

    fod *scr_sum = (fod*) &sn_sum[num_pts_max];
    fod *sci_sum = (fod*) &scr_sum[num_pts_max];
    ...
    __shared__ fod sn;
    __shared__ fod scr;
    __shared__ fod sci;
    ...
```

En caso 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 local 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 $\Theta = 0$.

Código 5.7: Inicialización de variables en el kernel

```
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 en las lineas 2 a la 11 del código 5.8.

Código 5.8: Cálculo de wmean y wvar en el kernel

```
1           ...
2           w_sum_sum[i] = 1.0 / powf(abs(error_share[i]), 2.0f);
3           w_mean_sum[i] = mag_share[i] / powf(abs(error_share[i]), 2.0f); }
4           ___syncthreads();
5           for(int size = NUM_THREADS/2; size > 0; size/=2){
6               if(idx < size && (idx + size < num_pts_arr[lc_i])){
7                   w_sum_sum[idx] += w_sum_sum[idx + size];
8                   w_mean_sum[idx] += w_mean_sum[idx + size];
9               }
10              ___syncthreads();
11          }
12
13          if (idx == 0) {
14              w_sum = w_sum_sum[0];
15              w_mean = w_mean_sum[0];
16              w_mean = w_mean / w_sum;
17              wmeans_arr[lc_i] = w_mean;
18          }
19          ___syncthreads();
20          ...
```

Luego se procede de según lo indicado en el Algoritmo 2, presentado en 3.2, siendo la diferencia principal que el algoritmo calcula el aporte de cada punto a los valores de α, c y sn desde las lineas 13 a 14 del código 5.9 y los utiliza para obtener sus valores finales usando reducción entre las lineas 23 y 32. 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 en la linea 46. Finalmente se actualizan los valores de s_r, si_i, zn_r, zn_i, p_r y p_i entre las lineas 49 y 58.

Código 5.9: Loop principal del kernel

```
1       fod arg = d_freqs[fi] * time_share[i];
2           fod phi = 2 * PI * (arg - floor(arg));
3           zr = cos(phi);
4           zi = sin(phi);
5           znr = 1;
6           pr = 1 / error_share[i];
7           zni = 0;
8           pi = 0;
9           fod factor = (mag_share[i] - w_mean) / error_share[i];
10          cfr = factor * cos(Nharmonics_arr[lc_i] * phi);
11          cfi = factor * sin(Nharmonics_arr[lc_i] * phi);
12
13          for (int j = 0; j < (2 * Nharmonics_arr[lc_i] + 1); j++){
14              ___syncthreads();
15              sn_sum[i] = powf(fabsf(pr), 2.0f) + powf(fabsf(pi), 2.0f);
16
17              scr_sum[i] = pr * cfr + pi * cfi;
18              sci_sum[i] = pr * cfi - pi * cfr;
19
20              alr_sum[i] = (zr * pr - zi * pi) / error_share[i];
21              ali_sum[i] = (zr * pi + zi * pr) / error_share[i];
22              ___syncthreads();
23              for(int size = NUM_THREADS/2; size > 0; size/=2){
24                  if(idx < size && (idx + size < num_pts_arr[lc_i])){
25                      sn_sum[idx] += sn_sum[idx + size];
26
27                      scr_sum[idx] += scr_sum[idx + size];
28                      sci_sum[idx] += sci_sum[idx + size];
29
30                      alr_sum[idx] += alr_sum[idx + size];
31                      ali_sum[idx] += ali_sum[idx + size];
32                  }
33                  ___syncthreads();
34              }
35              if (idx == 0) {
36                  sn = sn_sum[0];
37
38                  scr = scr_sum[0];
39                  sci = sci_sum[0];
40
41                  alr = alr_sum[0];
42                  ali = ali_sum[0];
43
44                  if (sn < 1e-9) sn = 1e-9;
45                  alr = alr / sn; ali = ali / sn;
46                  aov += (powf(fabsf(scr), 2.0f) + powf(fabsf(sci), 2.0f))/sn;
47              }
```

```
            __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, en el Código 5.10 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.

Código 5.10: Resultado del periodograma

```
        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);
        }
```

5.3. Promediado de subarmonicos

El periodograma entrega el arreglo `thetas` que contiene los valores de $\Theta(\omega)$ para cada curva, que puede ser usado para encontrar los máximos locales más relevantes del periodograma. 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.

Código 5.11: Obtención de candidatos a frecuencia

```
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 **thetas** permite identificar señales reales como se describió en 4.3, 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 la 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 este puntaje está dentro de un 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 vez de tener un corte dado por el usuario.

Código 5.12: Búsqueda de la frecuencia final

```
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 6

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.

6.1. Validación de GMHAOV

GMHAOV se validó usando el generador de curvas de luz incluido con MHAOV, y se calculó la raíz de la suma de los cuadrados de las diferencias relativas entre los resultados de MHAOV y GMHAOV, y ese valor se dividió por la cantidad de curvas de luz y la cantidad de frecuencias de prueba para obtener el error relativo promedio por curva por valor de Θ. Con 1000 curvas generadas sintéticamente por el generador de P4J [1] y 10000 frecuencias de prueba, este valor corresponde a 4.8×10^{-7}, 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 en el capítulo 3.4, y 10000 frecuencias de prueba, se obtuvo un valor del error cuadrático de 0.063, lo que corresponde a un error promedio de un 6.3 % para cada valor del periodograma. Esto es considerable, y al inspeccionar en qué valores hay una diferencia importante, se observa que hay ciertas curvas de luz problemáticas donde la diferencia entre ambos algoritmos es considerablemente mayor que para otras curvas. Sin embargo, estas curvas problemáticas no presentan ninguna 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.

[1] Repositorio de P4J que incluye el generador

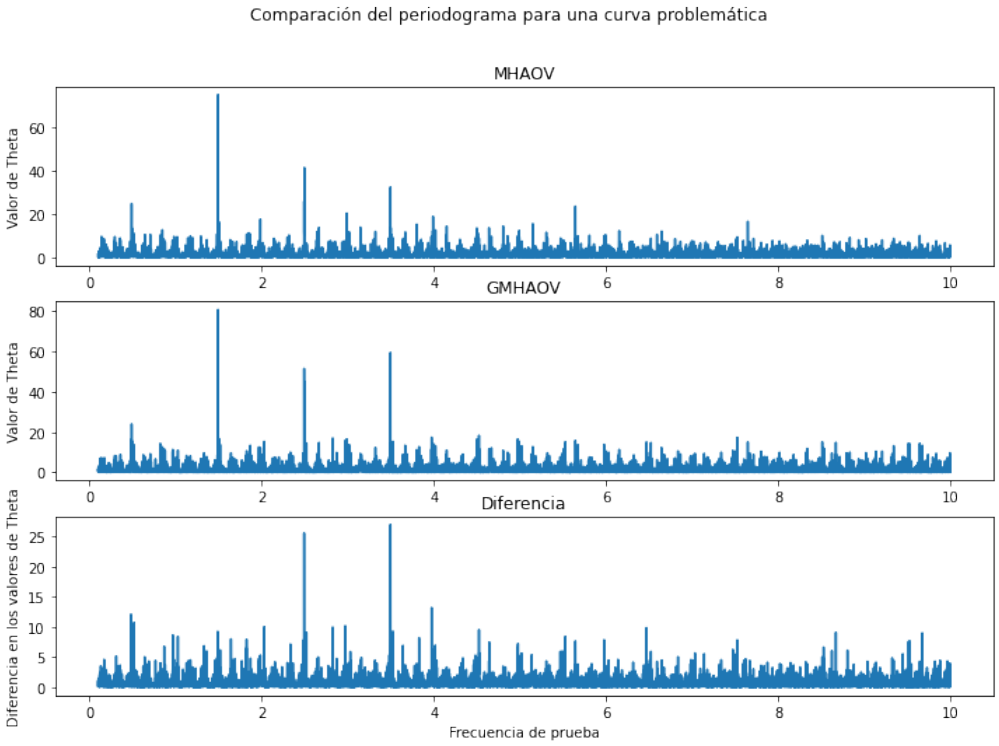

Figura 6.1: Resultado del periodograma para MHAOV (Arriba) y GMHAOV (Centro), y la diferencia entre ambos (Abajo).

En la Figura 6.1 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 significativamente la frecuencia que corresponde al máximo. De hecho al comparar la precisión de ambos algoritmos, ésta solamente disminuye desde 80 % a 79 % para RR Lyrae.

6.2. Efecto del 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 la Sección 3.4.

6.2.1. Promediado de subarmónicos (PS)

A continuación se presentan los efectos de introducir el promediado de subarmónicos en el cálculo del periodo.

6.2.1.1. RR Lyrae

En la Tabla 6.1, se observa que si bien PS logra obtener el período correcto para el 0.10 % de los objetos, los objetos para los que ya no se tiene el período correcto son casi el doble para ambos algoritmos. De la Tabla 6.2 se puede notar que al aplicar PS en GCE, la cantidad de submúltiplos, o fracción de curvas para las cuales se calcula una fracción, como la mitad o un tercio, del periodo original se reduce casi en un 50 %, pero la cantidad de múltiplos, o fracción de curvas para las cuales se calcula un múltiplo del periodo, como el doble o el triple,

casi se triplica, mientras que en GMHAOV los múltiplos incrementan levemente mientras que los submúltiplos se eliminan completamente.

Tabla 6.1: 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 6.2: 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

6.2.1.2. Binarias eclipsantes

De la Tabla 6.3, se observa que al usar PS, se deja de acertar para menos objetos tanto en GMHAOV como en GCE, y en la Tabla 6.3 los aciertos de GCE disminuyen al introducir PS, y los de GMHAOV se mantienen igual, pero la cantidad de submúltiplos disminuye en GCE. En este caso, no hay mayor beneficio en introducir PS.

Tabla 6.3: 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 6.4: 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

6.2.2. Efecto del promediado de armónicos (PA)

Como se observó en algunos periodogramas, usualmente hay una señal significativa 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).

6.2.2.1. RR Lyrae

De la Tabla 6.5, se reduce la cantidad de múltiplos como es de esperar, pero el efecto negativo del PA lleva a su efecto neto sea negativo.

Tabla 6.5: 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

6.2.2.2. 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 6.6: 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

6.3. Comparación de rendimiento

Para la evaluación del rendimiento, se utilizó 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 del promediado de subarmónicos en el tiempo de ejecución de GMHAOV para $N = 10, \ldots, 10^4$ curvas de luz con 150 puntos[2] cada una y una resolución de frecuencias de fres $= 10^{-2}, 10^{-3}, 10^{-4}$ y 10^{-5} desde las frecuencias 0.2 a 0.9. MHAOV se ejecuta de manera completamente secuencial para poder tener una medición adecuada del speedup.

Para cada combinación de valores de prueba, se mide el tiempo de ejecución de MHAOV, GMHAOV, GCE y el promediado de subarmónicos. Los resultados se encuentran en la figura 6.2. El tiempo de ejecución por curva de luz se muestra en la Figura 6.3

[2] El largo promedio de las curvas de luz de ALeRCE.

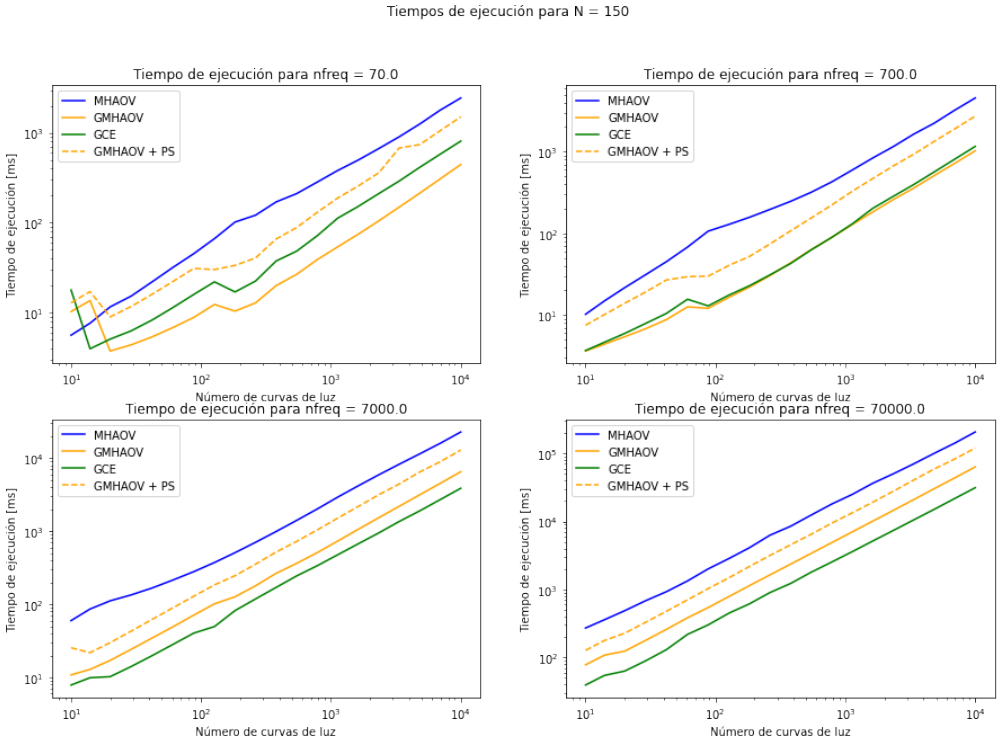

Figura 6.2: Resultados de las pruebas para cada combinación de frecuencias y número de curvas de luz simultaneas. La línea punteada representa el tiempo de ejecución de MHAOV sumado al tiempo de ejecución del promediado de subarmónicos.

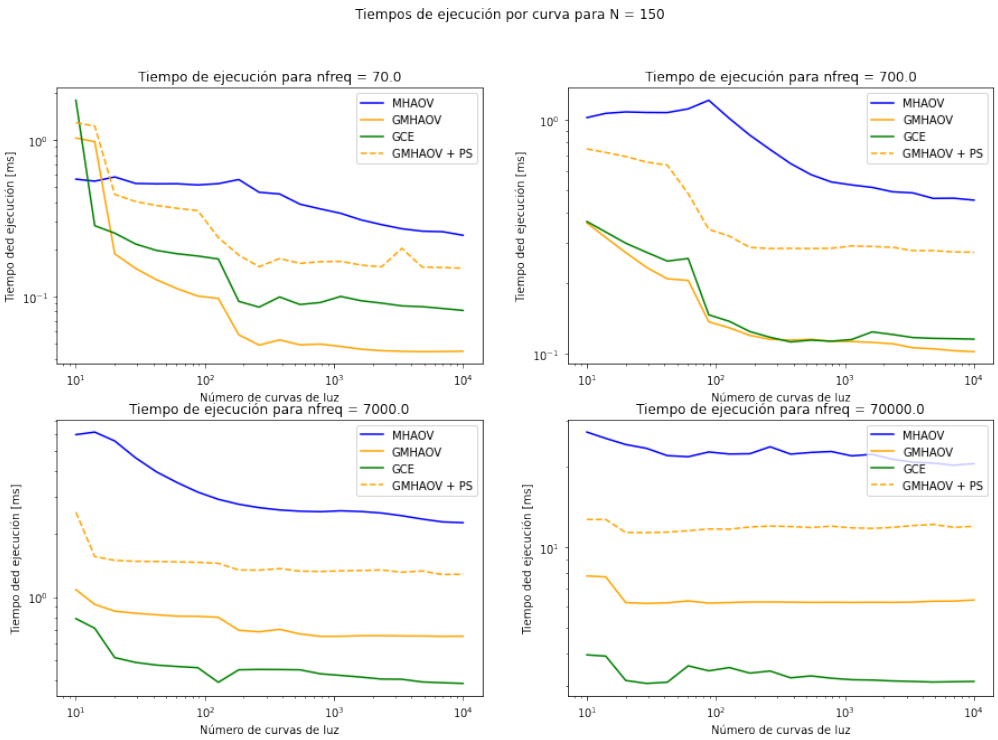

Figura 6.3: Tiempo de ejecución por curva para cada combinación de frecuencias y número de curvas de luz simultaneas.

Para todas las resoluciones de frecuencia, GMHAOV es considerablemente más rápido que MHAOV, y comparable con 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 6.2), pero sigue siendo menor. De todas maneras, para una alta resolución de frecuencias se logra un speedup de 1.5 y de casi 10 para resoluciones más bajas.

Esta tendencia queda más clara con la Figura 6.3, 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, con GMHAOV representando una mejora sustancial del tiempo de ejecución por curva.

Es importante destacar, sin embargo, que el tiempo de ejecución de tanto GMHAOV como el de 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 6.4. El tiempo de ejecución por curva y por frecuencia de prueba de GCE disminuye con mayor rapidez que GMHAOV, como se nota en la Figura 6.4, lo que explica su mejor desempeño a mayor resolución frecuencial.

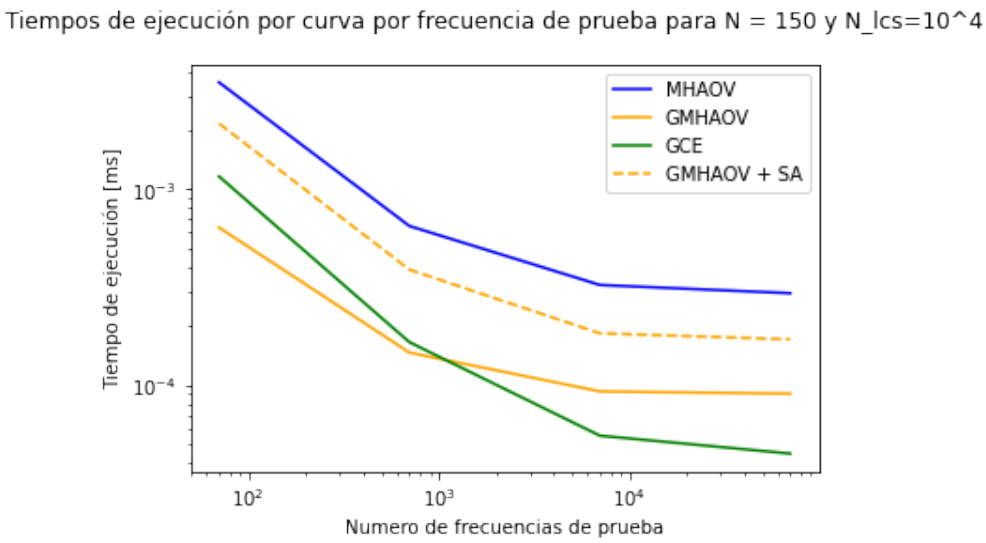

Figura 6.4: Variación del tiempo de ejecución por curva y por frecuencia de prueba en función de la cantidad de frecuencia de pruebas para $N = 10^4$

Si se ejecuta MHAOV de forma paralela en CPU, dividiendo todas las curvas en 8 tandas y ejecutandolas en 8 núcleos separados, se obtienen los resultados de las figuras **??**, **??** y 6.7. En la Figura 6.6, para un número de curvas del orden de 10^{-4}, tiempo de ejecución de MHAOV es comparable con el de GMHAOV a partir de las 700 frecuencias de prueba, rango en el que está el número de frecuencias de prueba usadas en la práctica. La razón de esto queda más claro en la Figura 6.5, donde se observa que para resoluciones de frecuencia más altas, MHAOV tiene un menor tiempo de ejecución por curva que GMHAOV. Finalmente, en la figura 6.7, el tiempo de ejecución por curva por frecuencia de prueba disminuye más rápido para MHAOV que para GMHAOV, pero no es claro si esta tendencia continua para un mayor número de frecuencias de prueba.

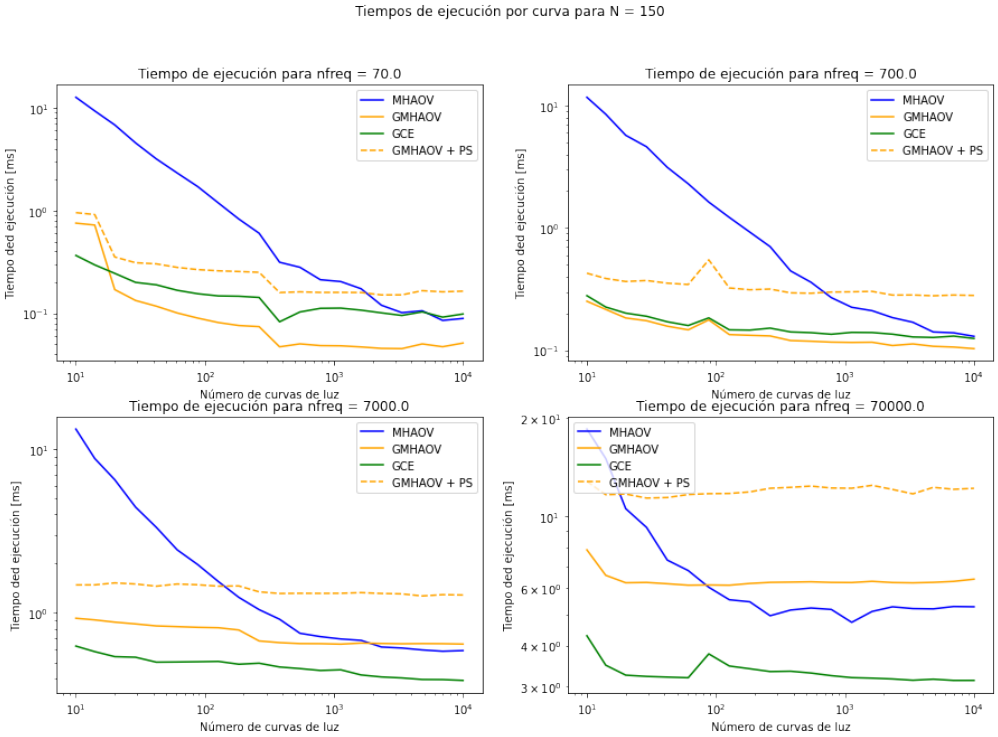

Figura 6.5: Tiempo de ejecución por curva para cada combinación de frecuencias y número de curvas de luz simultaneas, introduciendo paralelismo en MHAOV.

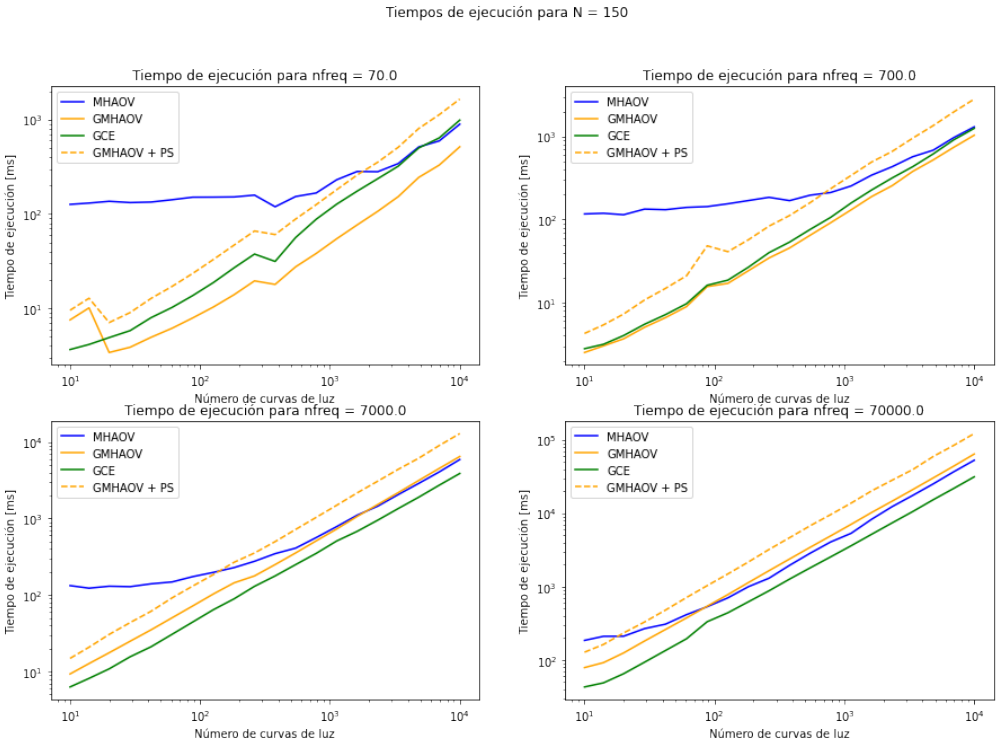

Figura 6.6: Tiempo de ejecución total para cada combinación de frecuencias y número de curvas de luz simultaneas, introduciendo paralelismo en MHAOV.

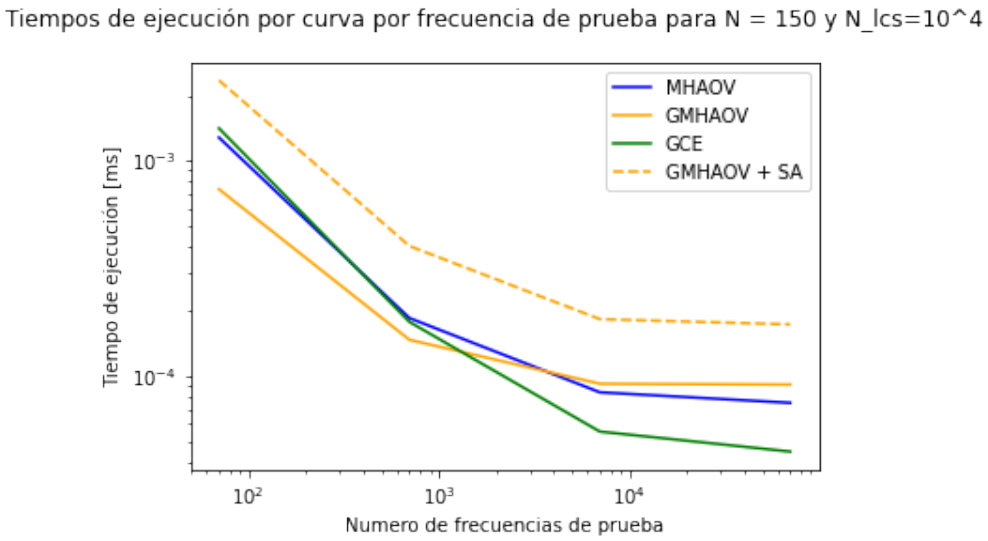

Figura 6.7: Variación del tiempo de ejecución por curva y por frecuencia de prueba en función de la cantidad de frecuencia de pruebas para $N = 10^4$, introduciendo paralelismo en MHAOV.

Capítulo 7

Análisis y conclusión

En este capítulo se concluye partir de los resultados, y se discute el trabajo futuro.

7.1. Análisis de los resultados

Es de gran importancia determinar qué características de las curvas de luz problemáticas descritas en 6.1 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 6.2.2, se observa que considerar el PS o PA puede ser beneficioso en algunos casos, si se incluye el puntaje asociado a la frecuencia corregida, calculado en el Código 5.12, como característica en el 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 6.3, la reducción en tiempo de ejecución de GMHAOV con respecto a MHAOV es tal que el tiempo de cálculo de los puntajes sumado al de GMHAOV es siempre menor al de MHAOV a menos que se procesen una cantidad de curvas del orden de 10^4 y se ejecute MHAOV de forma paralela con una CPU moderna. La decisión de implementar GMHAOV en el sistema de ALeRCE depende entonces de las capacidades de hardware de ALeRCE y de cuanto es posible optimizar GMHAOV.

De las pruebas de rendimiento se puede concluir que GMHAOV presenta un compromiso entre tiempo de ejecución y precisión del período recuperado superior a GCE, por lo que se espera sea un valioso aporte para el pipeline de ALeRCE.

7.2. Conclusión y trabajo futuro

En este trabajo se logró implementar un periodograma en GPU que es capaz de encontrar el período en un conjunto extenso de curvas de luz de forma paralela. Además se estudió un algoritmo que puede potencialmente identificar aquellos casos en que el máximo del periodograma corresponde a una señal falsa, por lo que el objetivo general fue cumplido. Sin embargo, por términos de tiempo no se lograron cumplir los objetivos 4 y 5, pues no se pudo evaluar el impacto de usar los resultados de GMHAOV, GCE, y 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 está presente 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 el proceso de clasificación. 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.

También es importante notar que la manera en la que GMHAOV paraleliza el algoritmo requiere una gran sincronización a nivel de bloque, lo que implica un impacto en la eficiencia de la implementación. Es importante considerar en el futuro una implementación como la de GCE, descrita en la Sección 3.3, donde cada thread procesa una curva de luz, en vez de procesar cada curva a nivel de bloque. La ventaja de esto sería que se vuelve innecesario sincronizar los bloques, pero cada thread debe realizar muchos más cálculos.

Finalmente, debido a que 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 el tiempo de ejecución. Sin embargo, es importante considerar que quizás sea posible optimizar aún más GMHAOV y obtener mejores resultados al compararlo con MHAOV ejecutado en CPU.

Bibliografía

[1] B. D. Warner *et al.*, *A practical guide to lightcurve photometry and analysis*, vol. 300. Springer, 2006.

[2] J. R. Percy, *Understanding variable stars*. Cambridge University Press, 2007.

[3] N. M. Ball and R. J. Brunner, "Data mining and machine learning in astronomy," *International Journal of Modern Physics D*, vol. 19, no. 07, pp. 1049–1106, 2010.

[4] E. D. Feigelson and G. J. Babu, "Big data in astronomy," *Significance*, vol. 9, no. 4, pp. 22–25, 2012.

[5] 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.

[6] E. C. Bellm, S. R. Kulkarni, T. Barlow, U. Feindt, M. J. Graham, A. Goobar, T. Kupfer, C.-C. Ngeow, P. Nugent, E. Ofek, *et al.*, "The zwicky transient facility: Surveys and scheduler," *Publications of the Astronomical Society of the Pacific*, vol. 131, no. 1000, p. 068003, 2019.

[7] Ž. Ivezić, S. M. Kahn, J. A. Tyson, B. Abel, E. Acosta, R. Allsman, D. Alonso, Y. AlSayyad, S. F. Anderson, J. Andrew, *et al.*, "Lsst: from science drivers to reference design and anticipated data products," *The Astrophysical Journal*, vol. 873, no. 2, p. 111, 2019.

[8] E. Bellm, R. Blum, M. Graham, L. Guy, Ž. Ivezić, W. O'Mullane, M. Patterson, J. Swinbank, and B. Willman, "Plans and policies for lsst alert distribution," *Large Synoptic Survey Telescope (LSST) Data Management*, 2019.

[9] P. Sánchez-Sáez, I. Reyes, C. Valenzuela, F. Förster, S. Eyheramendy, F. Elorrieta, F. Bauer, G. Cabrera-Vives, P. Estévez, M. Catelan, *et al.*, "Alert classification for the alerce broker system: The light curve classifier," *The Astronomical Journal*, vol. 161, no. 3, p. 141, 2021.

[10] M. Catelan and H. A. Smith, *Pulsating Stars*. John Wiley & Sons, 2014.

[11] L. Eyer and N. Mowlavi, "Variable stars across the observational hr diagram," in *Journal of Physics: Conference Series*, vol. 118, p. 012010, IOP Publishing, 2008.

[12] J. D. Scargle, "Studies in astronomical time series analysis. ii-statistical aspects of spectral analysis of unevenly spaced data," *The Astrophysical Journal*, vol. 263, pp. 835–853, 1982.

[13] A. Schwarzenberg-Czerny, "On the advantage of using analysis of variance for period search," *Monthly Notices of the Royal Astronomical Society*, vol. 241, no. 2, pp. 153–165, 1989.

[14] M. Zechmeister and M. Kürster, "The generalised lomb-scargle periodogram-a new formalism for the floating-mean and keplerian periodograms," *Astronomy & Astrophysics*, vol. 496, no. 2, pp. 577–584, 2009.

[15] 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.

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

[17] M. Catelan, B. J. Pritzl, and H. A. Smith, "The rr lyrae period-luminosity relation. i. theoretical calibration," *The Astrophysical Journal Supplement Series*, vol. 154, no. 2, p. 633, 2004.

[18] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 11.5.50," 2021.

[19] NVIDIA, "Cuda c++ programming guide v11.5.0," 2021.

[20] M. Harris, "Optimizing parallel reduction in cuda," 2007.

[21] R. Carrasco-Davis, E. Reyes, C. Valenzuela, F. Förster, P. A. Estévez, G. Pignata, F. E. Bauer, I. Reyes, P. Sánchez-Sáez, G. Cabrera-Vives, *et al.*, "Alert classification for the alerce broker system: The real-time stamp classifier," *arXiv preprint arXiv:2008.03309*, 2020.