UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA

PARALELIZACIÓN Y OPTIMIZACIÓN DEL ALGORITMO METABAT USANDO CUDA

Por: Jonathan Venegas Caro

Informe de Memoria presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al grado de Ingeniero Civil Informático

Marzo 2024

Concepción, Chile

Profesor Guía: Cecilia Hernández Rivas

A Eiichirō Oda

AGRADECIMIENTOS

Quiero utilizar este espacio para agradecer a mi familia, en especial a mi mamá, Julia Caro, y a mi padrino , Hernán Larraín, por su cariño, esfuerzo y apoyo incondicional durante todo mi trayecto académico y de mi vida, a quienes quiero mucho y considero mis modelos a seguir; A mis compañeros y amigos, en especial la ayuda de Sebastián Lagos y Camilo Ruiz, que me han soportado durante 6 años, y que me tendrán que seguir haciéndolo durante muchos años más.

Agradecer también a mi profesora guía, Cecilia Hernández, por su apoyo y dedicación durante toda la elaboración de este trabajo. Su asesoramiento, sugerencias y correcciones han sido fundamentales para mejorar la calidad de esta memoria. Aprecio enormemente su paciencia y compromiso con mi formación académica.

Adicionalmente se agradece a proyecto FONDOCYT regular 1220960 por el interés y apoyo financiero en la realización de esta memoria de título.

Resumen

En el área de microbiología, el análisis de la secuencia completa del genoma de los microorganismos es fundamental para seguridad de la población. Sin embargo, en algunos casos la obtención de estas secuencias genéticas se dificulta al ser realizadas sobre muestras extraídas desde su ambiente, en estos casos el uso de software proporciona gran ayuda para interpretar la información genética contenida en las muestras. Entre las herramientas de software se encuentra MetaBat, el que define un algoritmo de agrupamiento programado en C++, que tiene por objetivo la reconstrucción de genomas en comunidades microbianas complejas. Dichas comunidades pueden llegar a contener una gran cantidad de información genética, la cual nos puede llevar desde el orden de segundos hasta horas de ejecución. Esta memoria de título se enfoca en la aceleración del algoritmo MetaBat, en su versión número 2, haciendo uso de unidades de procesamiento gráfico (GPU), hardware dedicado que nos permiten realizar tareas paralelas en un tiempo mucho menor en comparación a su ejecución en procesadores (CPU). Para esto se utilizó la plataforma de desarrollo CUDA, que sobre el lenguaje de programación C++ nos permite ejecutar código tanto en procesador como en unidades de procesamiento gráfico. La ejecución de fragmentos de código en unidades de procesamiento gráfico además de diferentes optimizaciones al algoritmo original permitieron obtener una reducción significativa, de hasta un 40 por ciento, en el tiempo necesario para lograr la reconstrucción de los genomas.

Keywords – Microbiología, MetaBat, CUDA

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

Para la Microbiología, obtener la secuencia genómica completa, es decir, determinar el conjunto de material genético de algún microorganismo, es fundamental para su estudio, ayudando a comprender las características, diferencias, evolución y variantes de estos mismos. Además, obtener esta información es de gran utilidad para el diagnóstico, tratamiento y control de enfermedades que estas pueden provocar. Algunos de estos organismos pueden ser analizados con relativa facilidad, debido a que se han logrado *cultivar* en entornos controlados para su posterior análisis, pero para muchos otros este proceso es mucho más complejo. Este último grupo corresponde a organismos a los cuales todavía no ha sido posible mantener fuera de su ambiente natural, y por lo cual su análisis se debe realizar obteniendo muestras en su entorno junto con otros organismos que viven e interactúan con ellos, a causa de esto, obtener específicamente la secuencia genómica de un solo microorganismo, desde este conjunto, es una tarea compleja, principalmente debido a las limitaciones tecnológicas y técnicas actuales.

Para lograr obtener información en estos ambientes se deben utilizar técnicas más complejas que involucran programas computacionales bioinformáticos, que se utilizan para la reconstrucción de estos genomas. Uno de estos programas es MetaBAT, un programa creado por desarolladores e investigadores de La universidad de California, a través del Laboratorio Nacional Lawrence Berkeley, el cuál tiene por objetivo la reconstrucción de comunidades microbianas en ambientes complejos.

1.1. Objetivos generales

La presente memoria de título tiene por objetivo optimizar y refactorizar diferentes secciones de código del algoritmo de MetaBAT para su procesamiento en Unidades de procesamiento gráfico (GPUs), con el fin de mejorar el tiempo de ejecución del programa aprovechando las secciones paralelizables del algoritmo original.

1.1.1. Objetivos específicos

1. Analizar y comprender el funcionamiento del algoritmo original, priorizando las secciones en las que se utiliza paralelismo.
2. Diseñar e implementar un algoritmos paralelos para ser computados en GPU.
3. Optimizar los algoritmos computados en GPU para aprovechar de la mejor forma el potencial que nos ofrece.
4. Analizar y comparar el rendimiento de el algoritmo implementado con el original.

Capítulo 2

Marco teórico

2.1. Conceptos previos

2.1.1. Microbiología

> La microbiología es la ciencia que estudia todos los organismos vivos que son demasiado pequeños para ser visibles a simple vista. Esto incluye bacterias, arqueas, virus, hongos, priones, protozoos y algas, conocidos colectivamente como 'microbios'. Estos microbios desempeñan papeles clave en el ciclo de nutrientes, la biodegradación, la causa y el control de enfermedades, el cambio climático, el deterioro de los alimentos y la biotecnología. Por lo anterior, la microbiología ha demostrado ser una de las disciplinas más importantes en biología y cuya investigación ha sido y sigue siendo fundamental para cumplir con muchas de las aspiraciones y desafíos mundiales actuales, como el mantenimiento de la seguridad alimentaria, hídrica y energética para una población sana en una tierra habitable.
>
> Facultad de ciencias biológicas, Universidad de Concepción (4)

2.1.2. Metagenómica

La metagenómica es un área de la microbiología enfocada en el análisis de grandes cantidades de información genética de microorganismos obtenida en muestras

ambientales, como lo pueden ser muestras de suelo, agua o comunidades microbianas, que pueden contener organismos de múltiples especies.

2.1.3. Shotgun metagenomics

Shotgun metagenomics es una técnica de secuenciación, que tiene por objetivo extraer la información genética de todos los organismos presentes en una muestra sin necesidad de cultivarlos o aislarlos previamente. Esto da como resultado millones de fragmentos de información genética (reads), que deberán ser procesados posteriormente.

2.1.4. Reads

En el contexto de metagenómica, se denomina *reads* a pequeños fragmentos de ADN o ARN, que se obtienen mediante técnicas de secuenciación. Estos *reads* representan partes de los genomas de los microorganismos contenidos en una muestra ambiental. Los *reads* suelen tener una longitud de entre 50 y 300 nucleótidos (Estructura fundamental básica de los ácidos nucleicos representada por A, C G, T y U), dependiendo del método de secuenciación utilizado.

2.1.5. Contigs

Un *contigs* es una secuencia genética incompleta, generalmente de una longitud mayor a 2000 bases, que se obtiene a partir de la unión de *reads*, en un proceso denominado *sequence assembly* en el que se pueden usar múltiples muestras ambientales para mejorar los resultados. Los *contigs* se utilizan para ensamblar genomas completos usando algoritmos de *binning* (*Clustering*).

2.1.6. Abundancia

La abundancia es un valor que cuantifica la presencia relativa de un contig en una muestra genómica. La abundancia es un valor importante para analizar la diversidad y la composición de una muestra genómica, ya que permite identificar los contigs más representativos o abundantes, y también los más raros o escasos.

2.1.7. Metagenomic assembly

Metagenomic assembly hace referencia a un proceso de reconstrucción genómica, el cual recibe como entrada múltiples muestras ambientales. Este proceso da como resultado un archivo con múltiples *contigs* además de un archivo de abundancias en base a las muestras y *contigs* creados. Dentro de los algoritmos de ensamblaje existen los algortimos *Reference assembly* y *de-novo assembly*, en donde el primero utiliza bases de datos de genomas como referencia para agrupar los reads mientras que el segundo lo realiza sin referencias.

2.1.8. Binning

En el contexto de metagenómica el termino Binning, o agrupamiento por su traducción, se refiere al proceso de agrupar contigs provenientes de un algoritmo assembly para posteriormente ser analizados.

2.1.9. Formato FASTA

El formato de archivo FASTA es un formato de archivo de texto utilizado en bioinformática para representar secuencias de material genético, como se puede visualizar en la siguiente imagen:

Figura 2.1.1: Ejemplo de formato FASTA

Este formato contiene un grupo de secuencias genéticas representadas como se muestra en la imagen anterior, cada una de ellas comienzan con el carácter reservado

'>', seguido a esto nos encontraremos el encabezado y la secuencia genómica. El encabezado debe contener al menos el identificador de la secuencia, que se encuentra después del carácter de inicio, el que además puede contener información sobre el organismo e información complementaria. Después del encabezado se encuentra la secuencia genética perteneciente al *read* o *contig* del encabezado, conformada por los nucleótidos Adenina, Citosina, Timina y Guanina representados por sus iniciales **A**, **C**, **T** y **G** respectivamente.

2.2. Computación paralela

La computación paralela se puede definir como la ejecución de un grupo de tareas o instrucciones de forma paralela, es decir que se ejecutan de forma concurrente en un mismo espacio de tiempo. La computación paralela puede ayudar a procesar tareas o procesos que se dividen en tareas más pequeñas las cuales deben coordinarse para completar correctamente el procedimiento requerido. Esto puede alterar, o no, la complejidad de la tarea inicial, y recae en los desarrolladores analizar y determinar, dependiendo del contexto, si el uso de paralelismo es recomendable.

2.2.1. Procesamiento paralelo en GPU

Las unidades de procesamiento gráfico o GPUs por sus siglas en inglés son componentes de hardware optimizadas para procesamiento de tareas en paralelo, teniendo estas una arquitectura con núcleos más pequeños y eficientes que una unidad de procesamiento central o CPU. Esta arquitectura permite realizar una gran cantidad tareas al mismo tiempo, y utilizan el tipo de paralelismo *SIMD* (*Single Instruction, Multiple Data*), que es un tipo de paralelismo de datos en el cual se ejecuta una misma instrucción a diferentes colecciones de datos de forma paralela.

2.2.2. CUDA

La compañía Nvidia, con la intención de incentivar y facilitar el empleo de sus dispositivos GPU, lanzó en 2006 CUDA, una API que nos permite usar los recursos de sus productos utilizando diferentes lenguajes de programación tales como C, C++,

Fortran, Python y MATLAB.

Una de las ventajas de CUDA es que puede escalar fácilmente a medida que se mejora el hardware usado. Como se puede ver en la figura 2.2.1 un mismo programa puede ser ejecutado por dos dispositivos con capacidades de cómputo distintas, mejorando claramente en aquella que posee mayor capacidad de procesamiento paralelo:

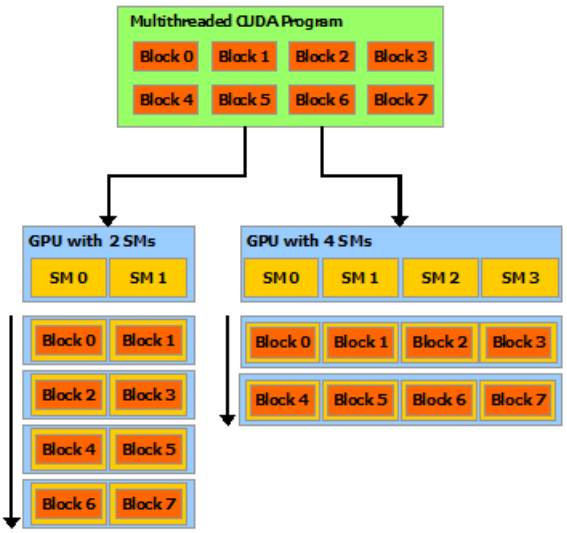

Figura 2.2.1: Escalabilidad de procesamiento, obtenida desde el blog de Nvidia (CPG)

En el contexto de programación en el lenguaje CUDA surgen los siguientes conceptos:

Host es el encargado de administrar los recursos como la memoria y disco duro, entre otros componentes del hardware, en la CPU.

Device o dispositivo, hace referencia al dispositivo o unidad de procesamiento que forma parte de la arquitectura de una tarjeta gráfica (GPU), encargado de ejecutar programas secundarios o fragmentos de código. La ejecución en el dispositivo es manejada desde el *host*.

Kernels Fragmento de código destinado a ser ejecutado en el dispositivo a petición del *host*.

Los *Kernels* generan subprocesos que se rigen bajo un modelo de jerarquía, el cual se enumerará desde el más específico al más general como se detalla a continuación:

1. ***Thread*** o **Hilo**, unidad básica de ejecución paralela que es asignada a un núcleo del dispositivo.

2. ***Block*** o **bloque**, es un conjunto de *threads*. Dentro de este bloque cada *thread* puede es identificado bajo un identificador incremental, y pueden ser organizados hasta en tres dimensiones.

3. ***Grid*** o **grilla**, es un conjunto de *blocks*. Cada bloque en la grilla es identificado bajo un identificador incremental, además de poder ser organizados hasta en tres dimensiones.

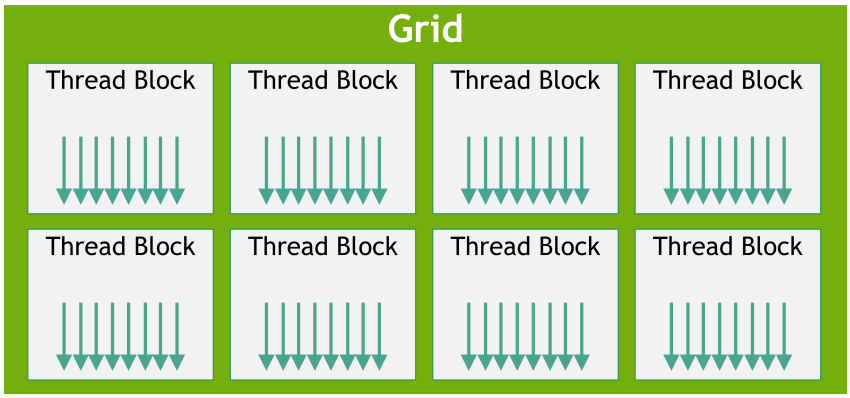

Figura 2.2.2: Jerarquía de subprocesos, obtenida desde el blog de Nvidia (CPG)

2.3. Descripción de MetaBAT

MetaBAT, publicado inicialmente el año 2015 (Kang DD, Froula J, Egan R, Wang Z) y actualizado en 2019 (Kang DD, Li F, Kirton E, Thomas A, Egan R, An H, Wang Z), es un algoritmo de *binning* que tiene por objetivo la reconstrucción de comunidades microbianas, y el cual está enfocado en comunidades complejas (muestras con una gran variedad de genomas). Este algoritmo en particular utiliza la frecuencia de tetranucleótidos (secuencia de 4 nucleótidos) y la abundancia de *contigs* en las muestras para lograr asociar los *contigs*. Adicionalmente MetaBAT requiere de otros algoritmos metagenómicos de reconstrucción (de naturaleza de-novo), desde los cuales obtiene el archivo de ensamblaje y abundancias de las muestras ambientales a procesar.

En la siguiente figura 2.3.1 se presentan las distintas etapas y flujo de datos del algoritmo:

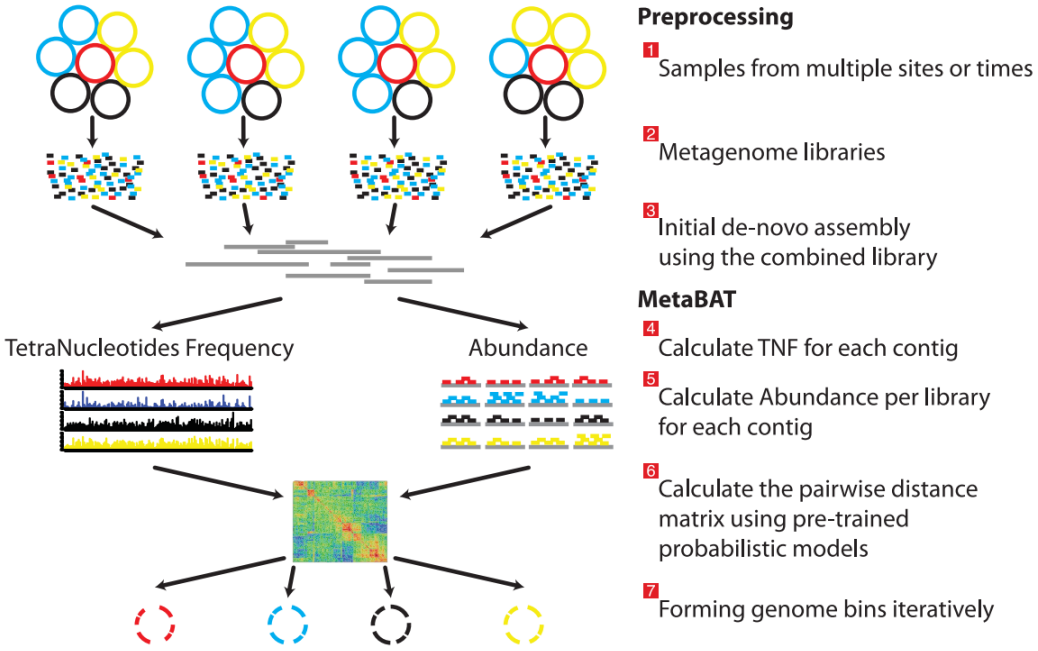

Figura 2.3.1: Flujo de trabajo de MetaBAT **Fuente:** (6)

A continuación, se explicará a grandes rasgos cada una de las etapas que componen el algoritmo de MetaBAT.

2.3.1. Preprocesado

En la figura 2.3.1 se puede apreciar que se debe realizar un preprocesado de los datos de muestreo antes de ser utilizados en el programa de MetaBAT. Más específicamente MetaBAT recibe como entrada un archivo en formato FASTA, un formato de archivo de texto que se utiliza para representar secuencias de bases. Este archivo se obtiene mediante el ensamblaje (de-novo) de *reads* obtenidos en la secuenciación de muestras ambientales, estas muestras pueden haber sido obtenidas en diferentes lugares y/o momentos.

2.3.2. Descripción del Algoritmo

A partir de un archivo de ensamblado podemos comenzar con el funcionamiento de MetaBAT, el cual se divide, a grandes rasgos, en 4 pasos:

2.3.2.1. Cálculo de matriz TNF

Para conseguir la compleja tarea de agrupar aquellos *contig* pertenecientes a un mismo genoma es primordial poder caracterizarlos. Para lograrlo, los desarrolladores e investigadores utilizan una matriz de la frecuencia de tetranucleótidos o TNF (*Tetra Nucleotide Frequency*) por sus siglas en inglés, la cual se obtiene a identificando todos las subcadenas de longitud 4 pertenecientes a una misma secuencia genética. De esta manera se define la matriz TNF como la estructura de datos que contiene los vectores asociados a cada uno de los *contigs*.

2.3.2.2. Cálculo de la abundancia

La abundancia se define como la presencia relativa de un *contig* en una muestra genómica, esta información es generada por los algoritmos utilizados en el preprocesado, y puede ser entregada a MetaBAT para obtener resultados más precisos.

2.3.2.3. Cálculo de distancias

La distancia es una métrica que mide la similitud entre dos *contigs*. El cálculo de distancia varía ligeramente entre la primera y segunda versión del algoritmo. En ambas se usa la abundancia y un modelo probabilístico que está basado en la frecuencia de tetranucleótidos junto con la longitud del *contig*.

2.3.2.4. Generación de bins

Para la generación de los *bins*, se utilizan dos algoritmos diferentes de *clustering* dependiendo de la versión. MetaBAT 1 utiliza un algoritmo propio basado en k-medoids; Mientras que MetaBAT 2 utiliza un algoritmo de *clustering* basado en grafos.

Capítulo 3

Desarrollo

3.1. Análisis preliminar de MetaBAT

Para determinar que secciones del algoritmo son mas relevantes a mejorar se realizó un análisis preliminar. Para ello se midieron los tiempos de ejecución de las secciones principales de MetaBAT 1 y 2. Estas pruebas se ejecutaron utilizando el dataset CAMI_High, creado por la comunidad para comparar los algoritmos metagenómicos (7).

3.1.0.1. Análisis de tiempo de ejecución

Adicionalmente a las etapas presentadas en la ilustración 2.3.1 se midió el tiempo de lectura y, en el caso de MetaBAT 2, el tiempo de cálculo del valor Cutoff[1], que se utiliza posteriormente para la generación del grafo. Por lo tanto, la distribución de tiempos en los diferentes tramos quedaría de la siguiente manera:

[1]Umbral de similitud.

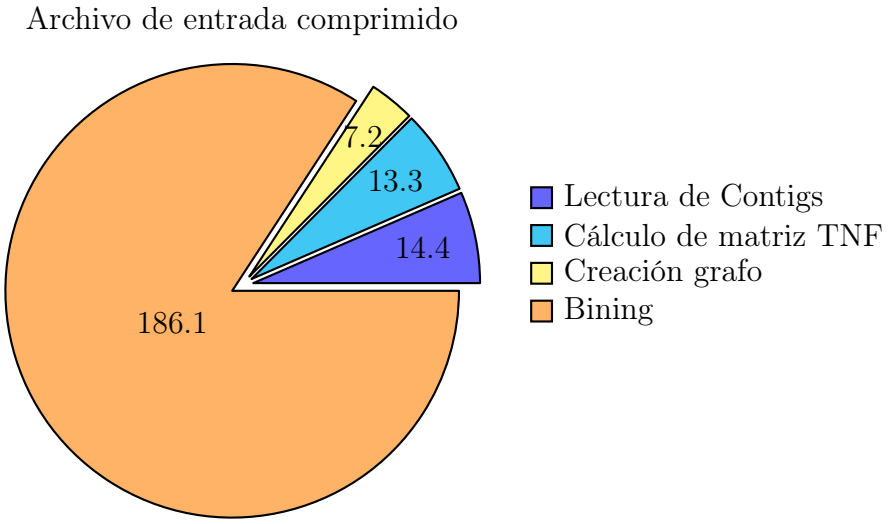

Figura 3.1.1: Distribución de tiempos de MetaBAT 1, en segundos

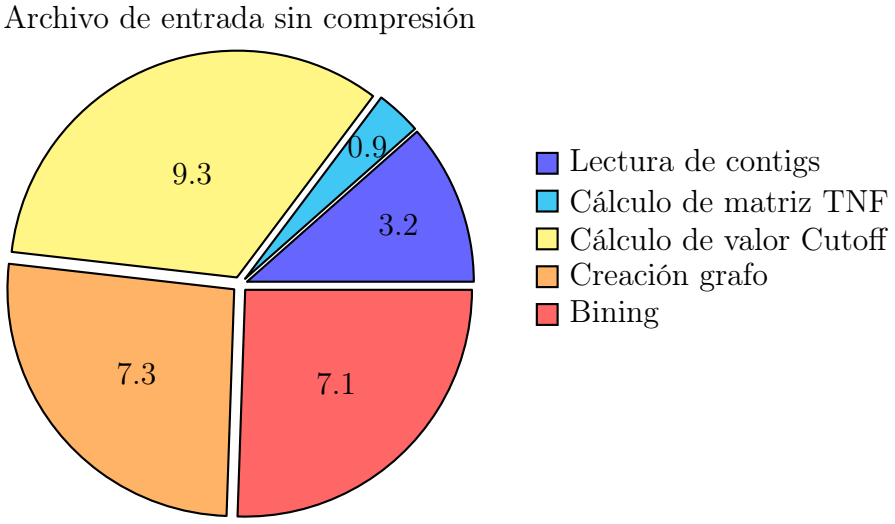

Figura 3.1.2: Distribución de tiempos de MetaBAT 2 en segundos

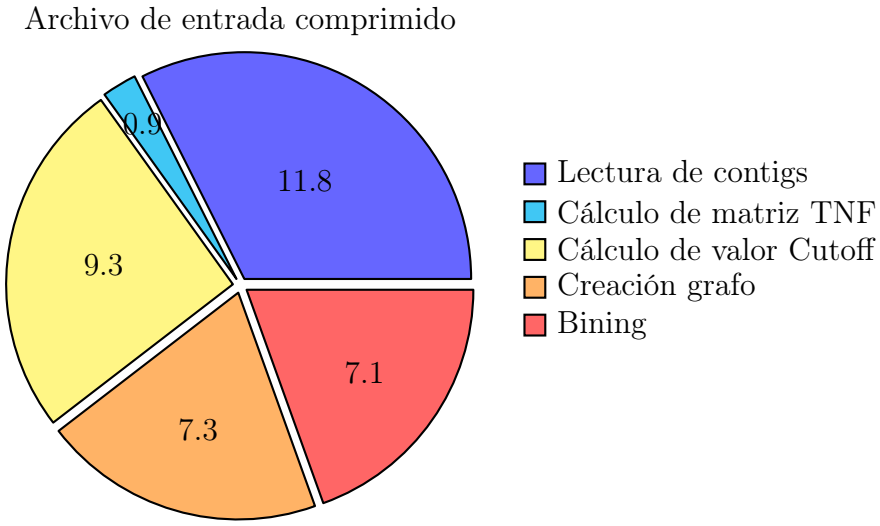

Figura 3.1.3: Distribución de tiempos de MetaBAT 2 en segundos

MetaBAT 1 demoró 221 segundos en completar su ejecución mientras que MetaBAT 2 tan solo 36 y 28 segundos, con y sin utilizar archivos de entrada comprimidos respectivamente. Con esto se pudo reducir, usando en ambos casos datos comprimidos, el tiempo de procesamiento total en un 83 % al utilizar la segunda versión de MetBat en comparación a la inicial.

Por otra parte, en los gráficos anteriores se aprecia que para MetaBAT 1 la etapa más lenta es el proceso de *Binnig*, mientras que en MetaBAT 2, sin considerar la lectura, son las etapas de cálculo de valor *cutoff* y la creación grafo.

3.2. Áreas de enfoque

Aunque en un inicio el objetivo de este proyecto fue mejorar y/o adaptar diferentes secciones algoritmo MetaBAT 1, el análisis de tiempos realizado derivó en enfocar estos esfuerzos a optimizar y mejorar el algoritmo de MetaBAT 2, dado que es una versión posterior y que presenta una ventaja significativa a su predecesor.

Las secciones de MetaBAT 2 que se decidió mejorar en este trabajo fueron las siguientes:

1. Lectura de *contigs*.

2. Cálculo de matriz TNF.
3. Cálculo de valor *cutoff*.
4. Creación del grafo.

Para cada una de estas secciones se analizó si su cálculo en GPU es beneficioso en comparación de ser realizadas en CPU, adaptando y/o optimizando cada una de ellas en caso de ser posible. Este análisis y posterior modificación se realizó a MetaBAT su versión v2.16-4-g40efa2d disponible en (REP).

3.3. Análisis del Algoritmo MetaBAT 2

Para lograr adaptar y/o optimizar el código de MetaBAT 2, se realiza un análisis detallado de las secciones críticas que se han identificado previamente, con el fin de determinar las mejores formas de abordar cada una de ellas.

3.3.1. Lectura de *contigs*

La lectura de *contigs* es la etapa inicial del algoritmo, que lee y almacena la información de los *contigs* a procesar. Para esto se debe proporcionar al programa un archivo en formato FASTA, el cual puede opcionalmente estar comprimido con el algoritmo GZIP. La información relacionada a estos *contigs* se almacena en 6 estructuras, los cuales guardarán la siguiente información:

1. contigs, encargado de almacenar el identificador interno para los *contigs* con secuencias de longitud mayor o igual a minContig.
2. contig_names, encargado de almacenar las etiquetas de los *contigs* con secuencias de longitud mayor o igual a minContig.
3. seqs, encargado de almacenar las secuencias de aminoácidos de los *contigs* con secuencias de longitud mayor o igual a minContig.
4. small_contigs, encargado de almacenar el identificador interno para los *contigs* con secuencias de longitud menor a minContig y mayor o igual 1000.

5. small_contig_names, encargado de almacenar las etiquetas de los *contigs* con secuencias de longitud menor a minContig y mayor o igual 1000.

6. small_seqs,encargado de almacenar las secuencias de aminoácidos de los *contigs* con secuencias de longitud menor a minContig y mayor o igual 1000.

La variable minContig puede ser modificada con argumentos de entrada, pero por defecto su valor es 2500. Adicionalmente se establece que su valor debe ser igual o mayor a 1500.

En este procedimiento se filtran los *contigs* en base a la longitud de su secuencia de aminoácidos, utilizando el parámetro *mincontig* anteriormente mencionado. Descartando todos aquellos *contigs*de longitud menor a 1000 bases.

El pseudo código del proceso de lectura se puede ver a continuación:

Algoritmo 1: Lectura, MetaBAT 2

Input: Nombre del archivo fasta

Result: Estructuras pobladas

```
1  nobs ← 0
2  nobs1 ← 0
3  kseq ← kseq_init(inFile)
4  while kseq_reader(kseq) do
5      if kseq.seq.length ≥ minContig then
6          nobs ← nobs + 1
7          contigs[kseq → name] ← nobs
8          contig_names.push_back(kseq → name)
9          seqs.push_back(kseq → seq)
10     else if kseq.seq.length ≥ 1000 then
11         nobs1 ← nobs1 + 1
12         small_contigs[kseq → name] ← nobs1
13         small_contig_names.push_back(kseq → name)
14         small_seqs.push_back(kseq → seq)
```

Para lograr obtener la información desde el archivo de entrada se utiliza la estructura

llamada Kseq_t, que hace uso de la biblioteca ZLIB, para poder leer archivos comprimidos en formato GZIP.

3.3.2. Cálculo de matriz TNF

El cálculo de la matriz TNF se realiza para todos los *contigs* almacenados en la estructura contigs_name, que son aquellos que tienen una longitud mayor o igual a min_contig. Para obtener esta matriz es necesario recorrer la secuencia de cada uno de los Contigs, analizando cada una sus subcadenas de largo 4 contenidas en este. Cada una de estas subcadenas se clasifica y aumenta el contador de una de las 136 posiciones del vector. Luego de recorrer la secuencia completa se realiza una normalización del vector, esto con el objetivo de que la representación del vector TNF sea independiente del largo del Contig. La matriz resultante de este cálculo es una matriz de tamaño *contigs* x 136.

El procedimiento puede ser visualizado en el siguiente pseudo código:

Algoritmo 2: Cálculo de Matriz TNF, MetaBAT 2

Input: Estructuras y nobs

Result: Matriz TNF

```
1  for r ← 0 to nobs − 1 do in parallel
2      seq ← seqs[r]
3      for i ← 0 to seq.length() − 4 do
4          tnNum ← tnToNumber(seq + i)
5          tnIdx ← TNLookup[tnNum]
6          if tnIdx < 136 then
7              TNF(r, tnIdx) ← TNF(r, tnIdx) + 1
8      rsum ← 0,0
9      for c ← 0 to 135 do
10         rsum ← rsum + TNF(r, c) × TNF(r, c)
11     sqrt(rsum)
12     for c ← 0 to 135 do
13         TNF(r, c) ← TNF(r, c) ÷ rsum
```

En el algoritmo anterior se hace uso de la función $tnToNumber()$, la cual recibe como parámetro un puntero a una cadena de caracteres y devuelve la representación numérica del tetranucleótido de los primeros caracteres. También se utiliza la estructura $TNLookup$, la cual almacena los identificadores de cada tetranucleótido, si alguno de los tetranucleótidos contiene un caracter no válido entonces entregará el valor 136 para que la subcadena actual sea ignorada.

La complejidad algorítmica de la lectura es $O(N \times M)$, donde N es la cantidad de *contigs* almacenada en seqs y M la longitud promedio de los contigs.

3.3.3. Cálculo de *Cutoff*

La variable Cutoff representa el valor de la distancia máxima entre dos *contigs* necesaria para ser almacenada en nuestro grafo, por esto mismo es necesario calcularla antes de crear nuestro grafo. La obtención de este valor se realiza de forma iterativa, tomando un subconjuntos aleatorios de hasta un máximo de 2500 *contigs* y obteniendo la distancia entre este subconjunto y el conjunto de todos los *contigs*[2]. Opcionalmente se puede entregar el valor de *cutoff* como parámetro. La complejidad de este algoritmo es $O(N)$, donde N es es la cantidad de *contigs* almacenados en seqs.

El algoritmo para obtener el valor Cutoff se puede ver en los siguientes dos pseudocódigos, almacenada en la variable pTNF:

[2] *Contigs* con longitud mayor a *minContig*

Algoritmo 3: Cálculo de valor cutoff, MetaBAT 2

Input: Matriz TNF

Result: Valor cutoff

```
1  if pTNF < 1 then
2      for i ← 0 to 10 do
3          _minp ← gen_tnf_graph_sample(maxP)
4          if _minp < 701 then
5              _minp ← 700
6          pTNF ← pTNF + _minp
7          if i == 1 & pTNF ÷ 2 < 701 then
8              pTNF ← 700
9              break
10         if i == 9 then
11             pTNF ← pTNF ÷ 10
12 else
13     pTNF ← pTNF × 10
14 pTNF ← pTNF ÷ 1000
```

El algoritmo anterior muestra cómo se puede calcular el valor de corte óptimo para una muestra de datos utilizando la función $gen\_tnf\_graph\_sample()$. Esta función recibe como parámetro la variable $maxP$, que indica el porcentaje mínimo de cobertura que se desea obtener. La función devuelve el valor de corte más alto que cumple con este criterio, es decir, que al menos el $maxP$ por ciento de la muestra tiene un valor mayor o igual al corte.

Algoritmo 4: Valor de corte para muestra, MetaBAT 2

Input: valor de cobertura mínima

Result: Valor cutoff

```
1  _nobs ← min(2500, nobs)
2  idx ← random_unique()
3  matrix ← ∅
4  for j ← 0 to nobs − 1 do in parallel
5      for i ← 0 to _nobs − 1 do
6          matrix ← 1 − cal_tnf_dist(idx[i], idx[j])
7  p ← 999
8  while p > 700 do
9      cutoff ← p ÷ 100 counton ← parallel_get_nodes_over_cutoff(matrix)
10     cov ← counton ÷ _nobs
11     if cov ≥ coverage then
12         if cov − coverage > coverage − previous_cov then
13             p ← previous_p
14         break
15     p = p − valor_aleatorio()
16 return p
```

Para seleccionar una muestra aleatoria de los Contigs, se utiliza la función *random_unique*(), que devuelve un vector de la misma longitud que el número total de Contigs. Cada elemento del vector es un índice que corresponde a uno de los Contigs. Esta función permite obtener una muestra representativa y sin repetición de los Contigs.

Luego de esto se crea la variable *matrix*, que almacena la similitud de la totalidad de los *contigs* con respecto a los primeros 2500 *contigs* contenidos en nuestra muestra aleatoria. Con esta variable es posible determinar el punto de corte óptimo logra la cobertura deseada.

Algoritmo 5: Cálculo de distancia TNF, MetaBAT 2

Input: Índices de dos *contigs*(r1 y r2)

Result: Valor de distancia TNF entre dos *contigs*

```
1 d ← DistanciaEuclidiana(TNF[r1], TNF[r2])
2 ctg1 ← log(min(seqs[r1].length, 500000))
3 ctg2 ← log(min(seqs[r2].length, 500000))
4 prob ← Modelo1(d, ctg1, ctg2)
5 if prob ≥ 0,1 then
6     prob ← Modelo2(d, ctg1, ctg2)
7     if prob < 0,1 then
8         prob ← 0,1
9 return prob
```

Para calcular la distancia TNF entre dos *contigs* es necesaria la distancia euclidiana entre sus vectores TNF y el logaritmo de sus longitudes, estos se utilizan en los dos modelos logísticos ajustados por los desarrolladores de MetaBAT.

3.3.4. Cálculo de grafo de probabilidad

Para crear el grafo de probabilidad que servirá de base para la generación de *bins* se emplea una estructura de grafo dirigido que representa las similitudes entre los *contigs*, donde la similitud representa la distancia entre dos *contigs* basada en la matriz TNF (Algoritmo 5). Adicionalmente, durante la creación del grafo se utilizan diferentes estrategias para lograr optimizar el cómputo y la utilización de memoria. La complejidad de este algoritmo es $O(N^2)$, donde N es es la cantidad de *contigs* almacenados en seqs.

El pseudocódigo de la creación del grafo se encuentra en el algoritmo 6:

Algoritmo 6: Cálculo de Grafo de probabilidad, MetaBAT 2

Input: Matriz TNF y cutoff

Result: Grafo de probabilidad

```
1  TILE ← optimal_subset_size()
2  for ii ← 0 to (nobs − 1) ÷ TILE do in parallel
3      edges ← ∅
4      for jj ← 0 to (nobs − 1) ÷ TILE do
5          for i ← ii to TILE do
6              for j ← jj to TILE do
7                  if i == j then
8                      continue;
9                  sTNF ← 1 − cal_tnf_dist(i, j)
10                 if sTNF > cutoff & (edges[i − ii].size() < maxEdges ||
                    sTNF > edges[i − ii].top()) then
11                     if edges[i − ii].size() == maxEdges then
12                         edges[i − ii].pop()
13                     edges[i − ii].push(sTNF)
14     for k ← 0 to TILE do
15         while !edges[k].empty() do
16             graph.insert(edges[k].top())
17             edges[k].pop()
```

Antes de comenzar con la creación del grafo se obtiene y almacena en la variable $TILE$ el subconjunto óptimo que computará cada hilo de procesamiento. Este valor se obtiene en base al tamaño de caché del procesador y de las estructuras utilizadas. Luego, cada hilo se encarga de calcular y guardar en la estructura de datos del grafo, las $maxEdges$ probabilidades más elevadas para cada uno de los nodos que pertenecen al subconjunto. Este proceso se realiza con la ayuda de la estructura *edges*, la cual contiene una cola de prioridad para cada uno de los nodos del subconjunto.

3.4. Mejoras del Algoritmo, MetaBAT 2

Para lograr los mejores resultados, se ha analizado y experimentado diferentes estrategias y enfoques para cada uno de las secciones a mejorar, dando como resultado un flujo de ejecución como se muestra en la Figura 3.4.1. Se despliegan los fragmentos que son ejecutados en $HOST$ (CPU), $DEVICE$ (GPU) o en ambas simultáneamente:

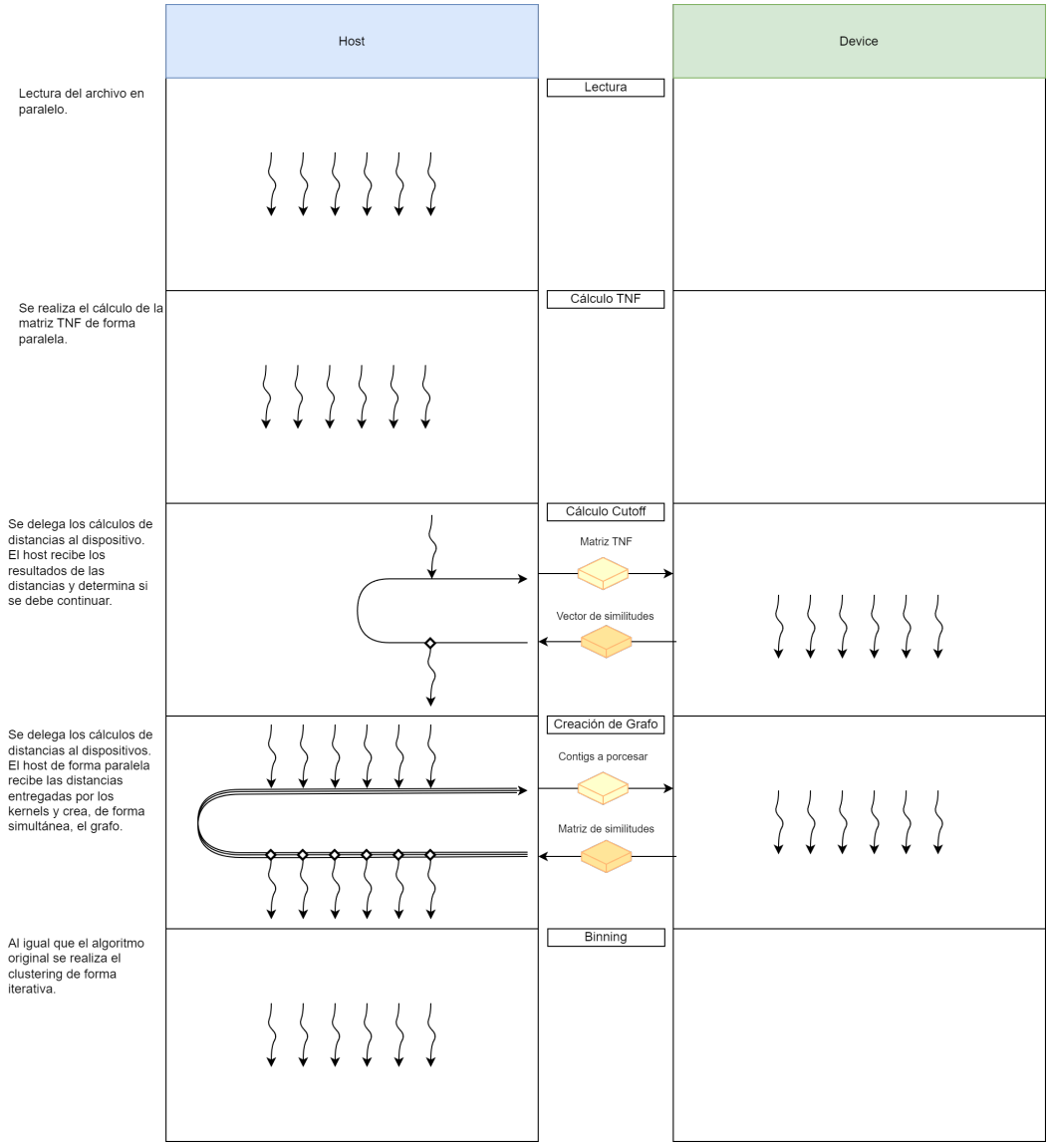

Figura 3.4.1: Flujo de trabajo propuesto

3.4.1. Lectura de *contigs*

En primer lugar tenemos los cambios realizados a la lectura del archivo *assembly*, que contiene la información de los *contigs*, que será leído para ser almacenado en memoria. El algoritmo original de MetaBAT 2 tiene la capacidad de leer el archivo *assembly* tanto comprimido como sin comprimir, en el caso de leer un archivo comprimido usa la biblioteca ZLIB.

El problema de utilizar la biblioteca ZLIP es que, si bien nos permite aceptar archivos de entrada comprimidos, nos limita a solo poder hacer lecturas secuenciales.

Para poder realizar esta tarea de forma paralela se propone una nueva estrategia de lectura, en la cual se procesa el archivo *assembly* sin comprimir. De esta forma se puede cargar la totalidad del archivo desde el disco a la memoria RAM, mediante funciones de bajo nivel de C++ esta se puede realizar este proceso de forma paralela.

La nueva lectura del algoritmo funciona de la siguiente forma:

Primero se carga a memoria el archivo fasta utilizando la función pread, esta función nos permite cargar secciones continuas de disco a RAM. Luego, podemos además utilizar threads para que cada hilo de procesamiento cargue una fracción del archivo.

Posteriormente cada hilo se encarga de extraer la información relacionada a los *contigs* de cada fracción del archivo. Para lograr esto de forma eficiente, en términos de memoria, se hace uso de la estructura de datos *STRING_VIEW* proporcionados por C++17, la cual nos permite almacenar referencias a cadenas de texto a diferencia de los *STRINGS* anteriormente utilizados que realizan una copia de la información. De esta forma, aunque se aumentará el uso de memoria debido a tener almacenado toda información sobre *contigs*(incluidos los que se ignoran), se utilizará una cantidad de memoria similar al algoritmo original.

El algoritmo resultante se describe en Algoritmo 7:

Algoritmo 7: Lectura MetaBAT 2 CUDA

Input: Nombre del archivo

Result: Estructuras pobladas

```
1  nobs ← 0
2  nobs1 ← 0
3  mem ← leer_archivo_paralelo(inFile, numThreads)
4  size_per_thread ← (mem.size ÷ numThreads)
5  for t ← 0 to numThreads − 1 do in parallel
6      thread_mem ← mem + size_per_thread × t
7      i ← 0
8      while i < size_per_thread do
9          if thread_mem[i] ='>' then
10             name ← get_name(thread_mem + i)
11             i ← i + name.size
12             seq ← get_seq(thread_mem + i)
13             i ← i + seq.size
14             if seq.size ≥ minContig then
15                 nobs ← nobs + 1
16                 contigs[name] ← nobs
17                 contig_names.push_back(name)
18                 seqs.push_back(seq)
19             else if seq.size ≥ 1000 then
20                 nobs1 ← nobs1 + 1
21                 small_contigs[name] ← nobs1
22                 small_contig_names.push_back(name)
23                 small_seqs.push_back(seq)
24     i ← i + 1
```

3.4.2. Cálculo de matriz TNF

Para el cálculo de la matriz TNF, en un principio se refactorizó el código de MetaBAT 2 para su procesado en GPU. En este caso se enviaban bloques de información correspondiente a las secuencias de *contigs* almacenados en seqs (mayores a minContig), y el *host* recibía la matriz TNF del bloque.

Este enfoque del problema no resultó ser beneficioso debido a que los tiempos de transferencia de información desde el *host* a GPU eran muy altos.

Debido a esto se decidió optimizar la implementación original, minimizando las consultas a memoria y cambiando el uso de estructuras auxiliares. Dando como resultado el Algoritmo 8.

Algoritmo 8: Cálculo de Matriz TNF MetaBAT 2 CUDA

Input: Estructuras y nobs

Result: Matriz TNF

```
1  for r ← 0 to nobs − 1 do in parallel
2      tnf_temp ← ∅
3      tn_temp ← first3(seq[i])
4      for i ← 3 to seq.length() − 1 do
5          tn_temp ← next_tn(tn_temp, seq[i])
6          tn ← get_tn(tn_temp)
7          if tnIdx = 256 then
8              continue
9          tnf_temp[tn] ← tnf_temp[tn] + 1
10     rsum ← 0,0
11     for c ← 0 to 135 do
12         rsum ← rsum + tnf_temp[c] × tnf_temp[c]
13     sqrt(rsum)
14     for c ← 0 to 135 do
15         TNF(r, c) ← tnf_temp[c] ÷ rsum
```

3.4.3. Cálculo de *Cutoff*

En la sección del cálculo del parámetro *cutoff* se cambió la estructura que almacena las distancias a un vector. Este vector tiene un tamaño igual a los *contigs* utilizados como muestra, y en el se almacenan las similitudes más altas para cada *contig* seleccionado en la muestra aleatoria. Esto nos permite optimizar el uso de memoria y procedimientos futuros de esta misma sección.

El vector de similitudes máximas es calculado por la GPU, dado que la similitud en base a TNF es computacionalmente cara y altamente paralelizable.

Ahora bien, como nuestro objetivo es devolver al *host* los valores de similitud más altos para cada *contig* de la muestra, se utiliza una estrategia de paralelización por bloques. Cada bloque tiene por objetivo encontrar la similitud más alta del *contig* asignado. Dentro de cada bloque, los threads se reparten el cálculo de un conjunto de valores de similitud, y luego se realiza una operación de reducción para obtener el máximo valor entre todos los threads del bloque.

El Algoritmo 9 muestra la función gen_tnf_graph_sample ejecutada en el *host*.

Algoritmo 9: Valor de corte para muestra, MetaBAT 2 CUDA

Input: Valor de cobertura mínima

Result: Valor cutoff

```
1  _nobs ← min(2500, nobs)
2  idx ← random_unique()
3  max_prob ← tnf_max_prob_sample(idx)
4  max_prob ← sort(max_prob)
5  p ← 999
6  counton ← 0
7  while p > 700 do
8      cutoff ← p ÷ 100
9      while max_prob[counton] ≥ cutoff & counton < max_prob.size do
10         counton ← counton + 1
11     cov ← counton ÷ _nobs
12     if cov ≥ coverage then
13         if cov − coverage > coverage − previous_cov then
14             p ← previous_p
15         break
16     p = p − valor_aleatorio()
17 return p
```

Como se observa en el Algoritmo 9, después de obtener nuestro arreglo max_prob se ordenan de forma descendiente. Esto nos permite recudir la cantidad de cálculos realizados para obtener el valor *counton*, teniendo que recorrer a lo más una sola vez el arreglo.

Por otro lado, el pseudo código del cálculo del vector arreglo max_prob se puede ver en el Algoritmo 10.

Algoritmo 10: Kernel de cálculo de distancias máximas, MetaBAT 2 CUDA

Input: Matriz TNF y vector de contigs

Result: Arreglo de similitudes máximas

```
1  shared_max ← ∅
2  contig_idx ← blockIdx
3  local_max ← −1
4  simPerThread ← nobs ÷ blockDim
5  for i ← threadIdx × simPerThread to (threadIdx + 1) × simPerThread do
6      dist ← cal_tnf_pre_dist(contig[contig_idx], contig[i])
7      if dist > local_max then
8          local_max ← dist
9  shared_max[threadIdx] ← local_max
10 syncThreads()
11 max_reduction(shared_max)
12 syncThreads()
13 if threadIdx = 0 then
14     max_dist[blockIdx] ← 1 − (1/(1 + exp(shared_max[0])))
```

El Algoritmo 10 corresponde a un *kernel* que es ejecutado por la GPU. Cada bloque procesado por el kernel tiene por objetivo obtener solo un valor de distancia. Para ello se divide el trabajo entre hebras y se utiliza memoria compartida para intercambiar información entre las pertenecientes a un mismo bloque. También se usa la función cal_tnf_pre_dist, que es una modificación de la función cal_tnf_dist [5] para obtener un resultado parcial de la distancia TNF. El Algoritmo 11 presenta esta función.

Algoritmo 11: Kernel de cálculo de distancia TNF parcial, MetaBAT 2 CUDA

Input: Índices de dos *contigs*(r1 y r2)

Result: Valor de distancia TNF entre dos Contigs

```
1 d ← DistanciaEuclidiana(TNF[r1], TNF[r2])
2 ctg1 ← log(min(seqs[r1].length, 500000))
3 ctg2 ← log(min(seqs[r2].length, 500000))
4 preProb ← Modelo1(d, ctg1, ctg2)
5 if preProb ≥ ln(9) then
6     preProb ← Modelo2(d, ctg1, ctg2)
7     if preProb < ln(9) then
8         preProb ← ln(9)
9 return preProb
```

El Algoritmo 11 ejecutado en el dispositivo nos permite ahorrar trabajo innecesario, calculando solo el valor final de la distancia mas cercana a 1[3].

3.4.4. Cálculo de grafo de probabilidad

Para esta última etapa se propone utilizar tanto los recursos de GPU como de CPU. Aunque esta vez se superpone el procesamiento de estos, esto nos permite reducir aun más el tiempo necesario para la construcción del grafo de probabilidad. Por un lado, el dispositivo(GPU) tendrá por objetivo calcular las distancias entre los *contigs*; mientras que el host(CPU) usa esta información obtenida por la GPU para filtrar e insertar las distancias a la estructura del grafo como se muestra en el Algoritmo 12.

[3]1 es el valor máximo que se puede obtener.

Algoritmo 12: Cálculo de Grafo de probabilidad MetaBAT 2 CUDA

Input: Matriz TNF y cutoff

Result: Grafo de probabilidad

```
1  TILE ← optimal_subset_size()
2  floor_pre_prob ← log((1 ÷ (1 − cutoff)) − 1)
3  for ii ← 0 to (nobs − 1) ÷ TILE do in parallel
4      edges ← ∅
5      cont ← 0
6      matrix_distances[cont] ← get_graph_distances(ii, 0, floor_pre_prob)
7      for jj ← 0 to (nobs − 1) ÷ TILE do
8          matrix_distances[cont + 1] ←
            get_graph_distances(ii, jj + TILE, floor_pre_prob)
9          for i ← ii to TILE do
10             for j ← jj to TILE do
11                 if i == j then
12                     continue;
13                 sTNF ← matrix_distances[cont](i, j)
14                 if sTNF > cutoff & (edges[i − ii].size() < maxEdges ||
                    sTNF > edges[i − ii].top()) then
15                     if edges[i − ii].size() == maxEdges then
16                         edges[i − ii].pop()
17                     edges[i − ii].push(sTNF)
18         cont ← cont + 1
19     for k ← 0 to TILE do
20         while !edges[k].empty() do
21             graph.insert(edges[k].top())
22             edges[k].pop()
```

Durante la ejecución del Algoritmo 12 cada hilo de procesamiento tiene en memoria a lo más 2 matrices de distancias, las que se calculan en GPU de forma asíncrona. De esta forma mientras se filtran las distancias de una matriz se calcula simultáneamente

las siguientes distancias a procesar.

El Algoritmo 13 se presenta el *Kernel* llamado en la creación del grafo. En este *kernel* tambien se utiliza la función $cal\_tnf\_pre\_dist$ presentada en el cálculo de *cutoff*, que nos permite calcular la distancias solo cuando serán mayor al *cutoff*, y en caso de ser menor se retorna 0.

Algoritmo 13: Kernel de cálculo de distancias en GPU, MetaBAT 2 CUDA

Input: Índices de *contigs* y valor de corte mínimo

Result: Matriz de distancias

1 $pob_index \leftarrow threadIdx + blockDim \times blockIdx$
2 $ctg1 \leftarrow getContig1(pob\_index)$
3 $ctg1 \leftarrow getContig2(pob\_index)$
4 $preProb \leftarrow cal\_tnf\_pre\_dist(ctg1, ctg2)$
5 **if** $preProb > floor\_pre\_prob$ **then**
6 $\quad matrix \leftarrow 1 - (1 \div (1 + exp(preProb)))$
7 **else**
8 $\quad matrix \leftarrow 0$

Capítulo 4

Evaluación experimental

Para evaluar la efectividad de las modificaciones realizadas al algoritmo, se comparó el desempeño y resultados de ambos algoritmos, para luego presentar un resumen y un análisis de los datos obtenidos.

Esta evaluación se realizó a MetaBAT 2 en su versión v2.16-4-g40efa2d y la versión modificada de esta misma, a la que se referirá en las pruebas como MetaBAT 2 CUDA. En ambos se utilizó el compilador *NVCC*, propia de NVIDIA, utilizando los mismos argumentos de compilación [A1.1].

4.1. Datasets

Para realizar la evaluación se utilizaron un total de cuatro conjuntos de datos, todos ellos obtenidos desde el portal de NERSC (National Energy Research Scientific Computing Center) (DAT). Estos *dataset* son:

- Cami High, *dataset* sintético creado por la comunidad con el objetivo de evaluar el rendimiento de diferentes algoritmos metagenómicos.
- Case 1, *dataset* creado por los desarrolladores de MetaBAT, que contiene muestras de metagenomas de diferentes ambientes y organismos, para evaluar el rendimiento del algoritmo en datasets pequeños.
- Case 2, *dataset* creado por los desarrolladores de MetaBAT, que contiene

muestras de metagenomas de diferentes ambientes y organismos, para evaluar el rendimiento del algoritmo en datasets medianos.

- Case 3, *dataset* creado por los desarrolladores de MetaBAT, que contiene muestras de metagenomas de diferentes ambientes y organismos, para evaluar el rendimiento del algoritmo en datasets grandes.

Utilizando los *datasets* presentados anteriormente se pretende evaluar como escalan ambos algoritmos a medida que se aumenta el número de contigs a procesar. La cantidad de contigs y tamaño de cada uno de los datasets se puede ver en la Tabla 4.1.1:

Dataset	Tamaño (MB)	N° Contigs [1]
Cami High	2804	30841
Case 1	267	26803
Case 2	4730	230079
Case 3	8770	733876

Cuadro 4.1.1: Detalle de datasets

[1]Contigs con un más de 2500 bases

4.2. Pruebas

Para analizar el impacto de las mejoras aplicadas en el desarrollo de este proyecto, se realizó una comparación entre la implementación propuesta y el algoritmo original de MetaBAT 2, en ambos casos utilizando los conjuntos de datos sin comprimir. De esta manera, se podrá apreciar y contrastar el rendimiento y la calidad de los resultados obtenidos de forma imparcial entre ambas metodologías.

Las pruebas se realizaron en un servidor equipado con un procesador Intel(R) Core(TM) i9-10980XE, el cual cuenta con 18 núcleos y 36 hilos, acompañado con una tarjeta gráfica Nvidia RTX A2000 con 12 GB de memoria dedicada.

Todas las pruebas que se enseñarán a continuación se ejecutaron utilizando la cantidad máxima de threads posibles, 36 en este caso, para realizar las tareas paralelizadas en el host (CPU).

4.2.1. Cami High

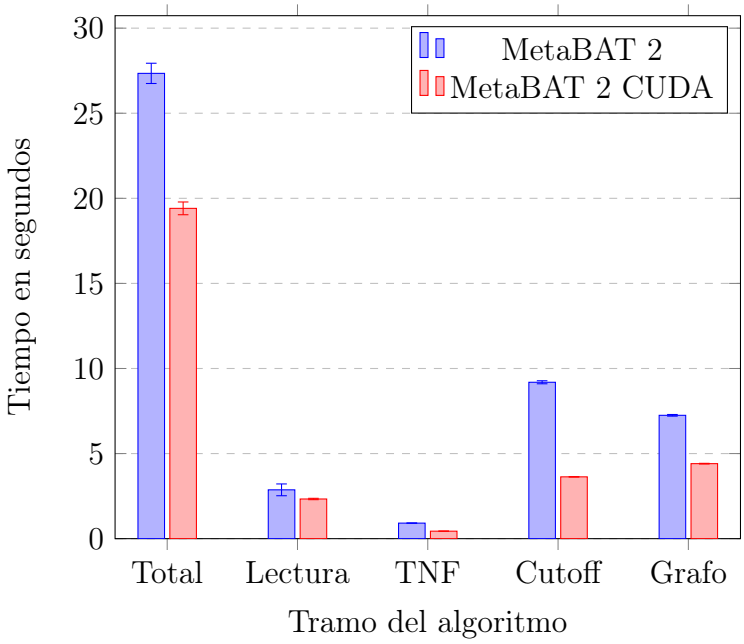

Figura 4.2.1: Gráfico de tiempos para Cami High sin usar abundancia

Tramo	MetaBAT 2	MetaBAT 2 CUDA
Lectura	2.86	2.33
TNF	0.91	0.44
Cutoff	9.19	3.63
Grafo	7.24	4.40
Total	27.34	19.41

Cuadro 4.2.1: Tabla de tiempos en segundos para Cami High sin usar abundancia

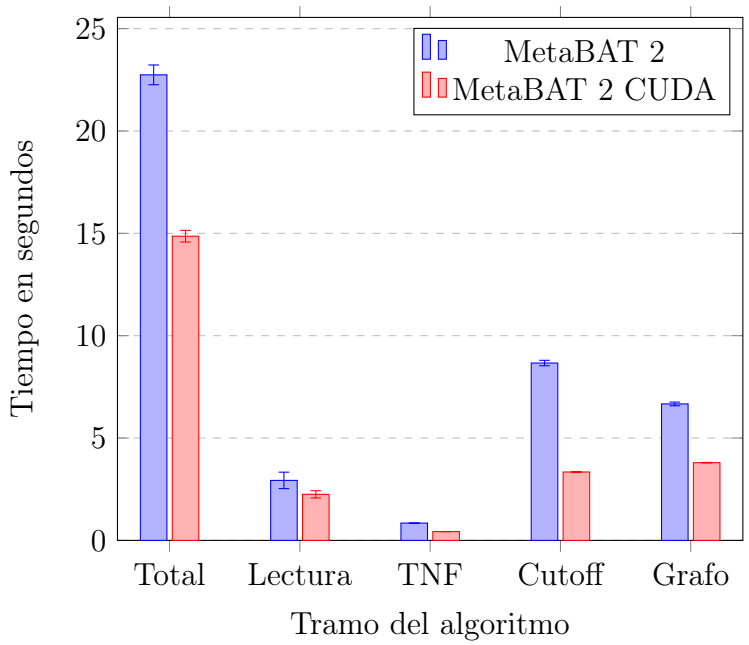

Figura 4.2.2: Gráfico de tiempos para Cami High usando abundancia

Tramo	MetaBAT 2	MetaBAT 2 CUDA
Lectura	2.93	2.25
TNF	0.84	0.42
Cutoff	8.66	3.34
Grafo	6.66	3.79
Total	22.74	14.86

Cuadro 4.2.2: Tabla de tiempos en segundos para Cami High usando abundancia

4.2.2. Case 1

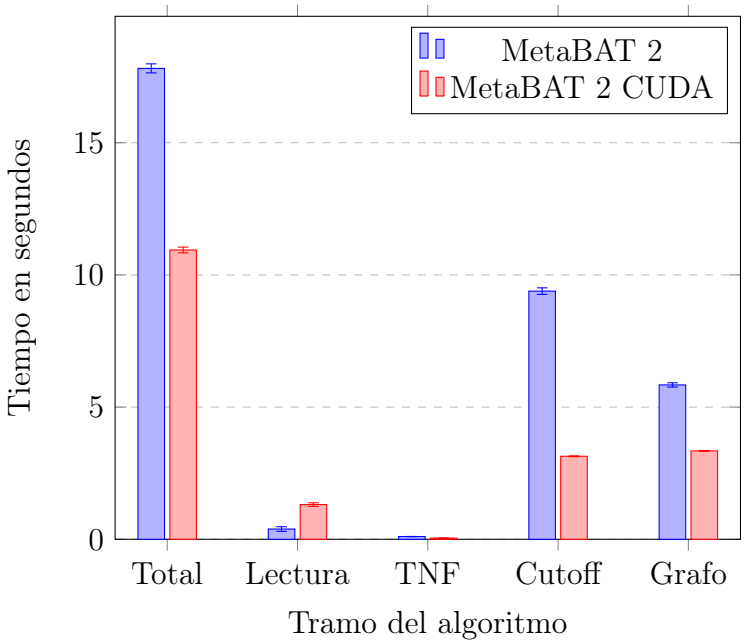

Figura 4.2.3: Gráfico de tiempos para Case 1 sin usar abundancia

Tramo	MetaBAT 2	MetaBAT 2 CUDA
Lectura	0.38	1.31
TNF	0.10	0.04
Cutoff	9.38	3.14
Grafo	5.83	3.34
Total	17.81	10.94

Cuadro 4.2.3: Tabla de tiempos en segundos para Case 1 usando abundancia

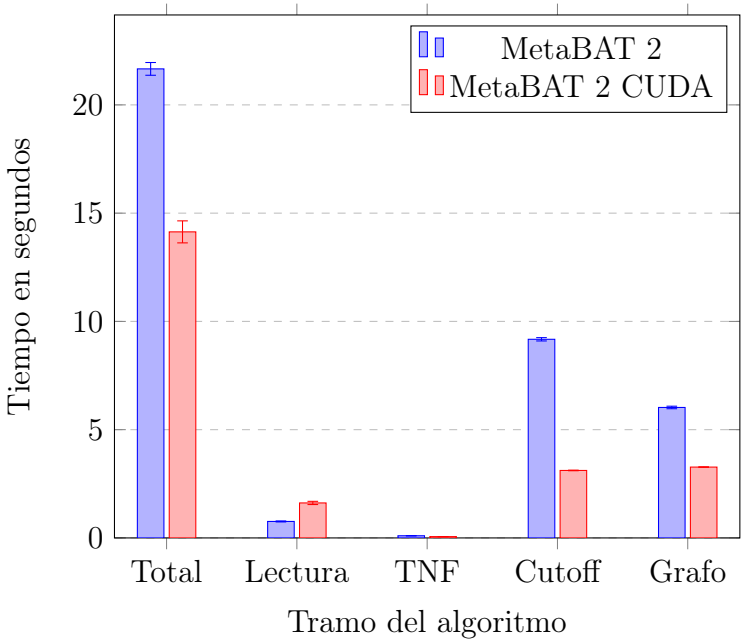

Figura 4.2.4: Gráfico de tiempos para Case 1 usando abundancia

Tramo	MetaBAT 2	MetaBAT 2 CUDA
Lectura	0.76	1.61
TNF	0.10	0.06
Cutoff	9.17	3.11
Grafo	6.02	3.34
Total	21.66	14.13

Cuadro 4.2.4: Tabla de tiempos en segundos para Case 1 usando abundancia

4.2.3. Case 2

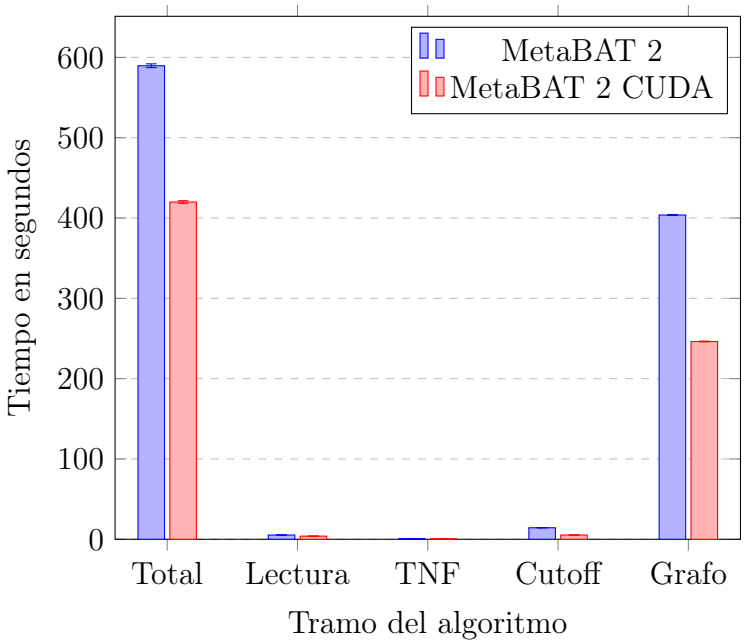

Figura 4.2.5: Gráfico de tiempos en segundos para Case 2 sin usar abundancia

Tramo	MetaBAT 2	MetaBAT 2 CUDA
Lectura	5.30	3.91
TNF	0.53	0.36
Cutoff	14.34	5.33
Grafo	403.76	246.25
Total	589.58	419.95

Cuadro 4.2.5: Tabla de tiempos en segundos para Case 2 sin usar abundancia

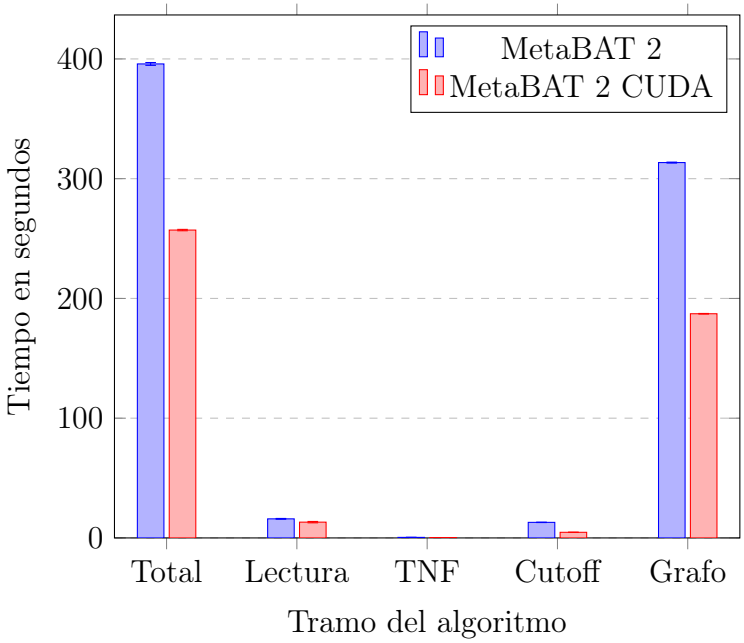

Figura 4.2.6: Tiempos para Case 2 usando abundancia

Tramo	MetaBAT 2	MetaBAT 2 CUDA
Lectura	15.93	13.18
TNF	0.45	0.27
Cutoff	13.00	4.70
Grafo	313.44	187.20
Total	395.82	257.08

Cuadro 4.2.6: Tabla de tiempos en segundos para Case 2 usando abundancia

4.2.4. Case 3

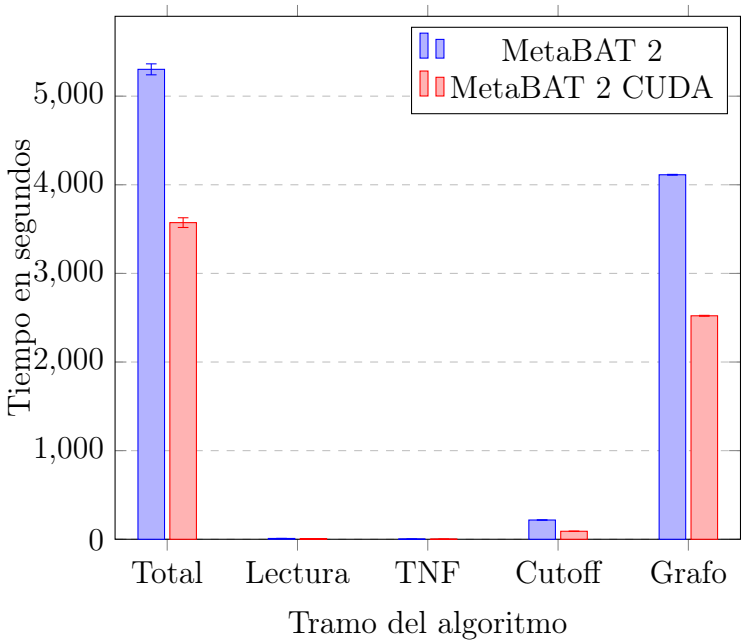

Figura 4.2.7: Gráfico de tiempos para Case 3 sin usar abundancia

Tramo	MetaBAT 2	MetaBAT 2 CUDA
Lectura	9.21	6.28
TNF	1.73	1.05
Cutoff	217.22	90.31
Grafo	4113.03	2521.75
Total	5302.73	3572.94

Cuadro 4.2.7: Tabla de tiempos en segundos para Case 3 sin usar abundancia

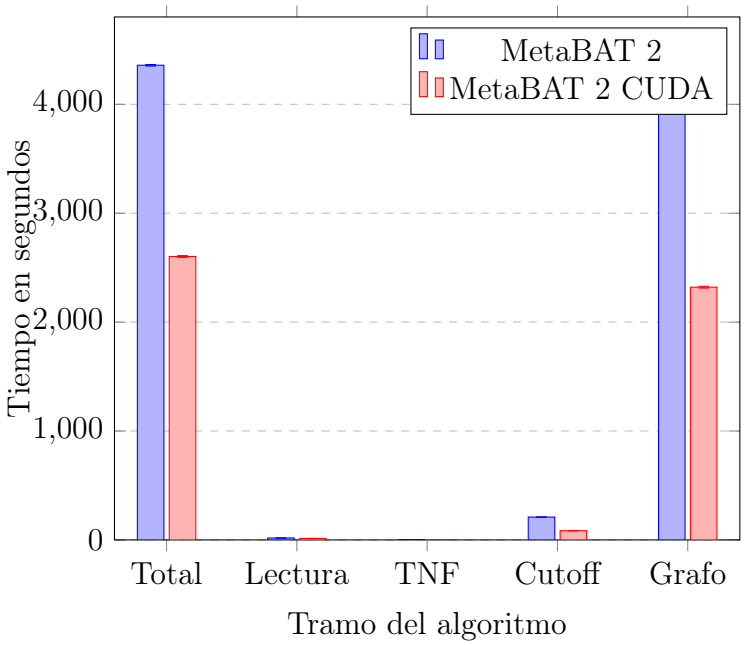

Figura 4.2.8: Gráfico de tiempos para Case 3 usando abundancia

Tramo	MetaBAT 2	MetaBAT 2 CUDA
Lectura	17.86	13.47
TNF	1.66	0.91
Cutoff	209.85	84.74
Grafo	3941.46	2320.97
Total	4358.84	2602.66

Cuadro 4.2.8: Tabla de tiempos en segundos para Case 2 usando abundancia

4.3. Verificación

Debido a que en este trabajo se ha modificado el algoritmo de MetaBAT 2, es de suma importancia verificar que los resultados entregados por la nueva implementación, y comprobar que sean consistentes con la salida del programa original.

Como se explicó previamente en el análisis del algoritmo MetaBAT, el programa genera como salida una serie de archivos, conteniendo cada uno de ellos la agrupación de *contigs* que corresponden a un mismo organismo.

Dado que el algoritmo de MetaBAT utiliza valores aleatorios para su funcionamiento, la verificación de la salida se realizó usando las mismas semillas[2], de esta forma se deberían obtener los mismos resultados. Se analizó tanto la cantidad de *bins* generados, como la similitud de los archivos de salida.

Mediante un programa externo creado en el lenguaje Python se comprobó que, para cada uno de los *datasets* presentados, ya utilizando o no los archivos de abundancia correspondiente a cada uno, tanto la cantidad de *contigs* generados como los archivos de salida son idénticos. Por lo cual se puede decir que ambos algoritmos son equivalentes.

4.4. Resultados

Para cuantificar la mejora proporcionada por el algoritmo implementado en CUDA se utilizará la aceleración, representando relación entre el algoritmo modificado en comparación a la implementación original, y la cuál se define como:

$$\text{Aceleración} = \frac{\text{Tiempo algoritmo original}}{\text{Tiempo del nuevo algoritmo}}$$

dada está medida, las aceleraciones proporcionadas por el nuevo algoritmo para cada uno de los datasets sería las siguientes:

Dataset	Sin abundancia	Con abundancia
Cami High	1.40	1.53
Case 1	1.62	1.53
Case 2	1.40	1.53
Case 3	1.48	1.67

Cuadro 4.4.1: Aceleración del algoritmo

Lo que también se puede representar en una mejora porcentual como se muestra el la siguiente tabla:

[2]Valor inicial utilizado para inicializar el generador de números pseudoaleatorios

Dataset	Sin abundancia	Con abundancia
Cami High	29 %	34 %
Case 1	38 %	34 %
Case 2	28 %	35 %
Case 3	32 %	40 %

Cuadro 4.4.2: Reducción de tiempo porcentual

Basándonos en los gráficos presentados en las pruebas, las etapas que más contribuyen a la reducción de tiempos es la obtención del valor de corte (Tramo *Cutoff*) y la creación del grafo (Tramo Grafo), secciones en las cuales se alcanza una reducción de tiempo promedio de 62.47 % y 41.42 % respectivamente en comparación al algoritmo original, siendo está ultima aun más significativa a medida que aumenta el tamaño de *contigs* a procesar.

También puede apreciar que la aceleración, en general, es mayor cuando se utiliza el archivo de abundancia. Esto de debe a que, al utilizar la abundancia, se filtran aquellos *contigs* que tienen una baja presencia en el archivo, lo que reduce la cantidad de nodos y aristas que será necesario procesar en las etapas de creación de grafo y *binning*.

Con respecto a los datos y resultados obtenidos por el algoritmo original de MetaBAT 2 y su versión modificada para CUDA son idénticos, como se muestra en la sección de verificación. Ambos algoritmos producen la misma cantidad de *bins* y archivos de salida. Esto indica que la adaptación a CUDA no afecta la calidad ni la precisión del algoritmo, sino que solo mejora su rendimiento y velocidad.

Capítulo 5

Conclusión

Como se ha podido comprobar a través de los resultados obtenidos, la utilización de diferentes enfoques de programación, como lo es el computo a través de unidades de procesamiento gráfico, resultó claramente beneficioso para la aceleración del algoritmo MetaBat 2. Permitiéndonos aprovechar de mejor manera los recursos disponibles y reducir los tiempos de cómputo.

Gracias a esto se han logrando cumplir satisfactoriamente los objetivos propuestos en esta memoria de titulo, utilizando diferentes técnicas de optimización, procesamiento en GPU y solapamiento de trabajo en CPU y GPU. Logrando reducir, en su conjunto, el tiempo de procesamiento hasta en un 40 por ciento en comparación al algoritmo original, el cual ya contaba con optimizaciones y paralelizaciones realizadas por los desarrolladores e investigadores de instituciones involucradas en su desarrollo. De esta manera, se ha logrado contribuir al avance del campo de la metagenómica, facilitando el análisis de grandes conjuntos de datos de secuenciación.

Bibliografía

[CPG] *CUDA C++ Programming Guide*. NVIDIA. Disponible en https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[DAT] *Datasets de Pruebas*. National Energy Research Scientific Computing Center. Disponible en https://portal.nersc.gov/dna/RD/Metagenome_RD/MetaBAT/Files/.

[REP] *Repositorio de MetaBAT*. berkeleylab. Disponible en https://bitbucket.org/berkeleylab/metabat.

[4] (2023). *Microbiología - Facultad de Ciencias Biológicas Universidad de Concepción*. Universidad de Concepción. Disponible en https://cienciasbiologicasudec.cl/departamentos/microbiologia/.

[5] Kang DD, Froula J, Egan R, Wang Z (2015). Metabat, an efficient tool for accurately reconstructing single genomes from complex microbial communities. *PeerJ*.

[6] Kang DD, Li F, Kirton E, Thomas A, Egan R, An H, Wang Z (2019). Metabat 2: an adaptive binning algorithm for robust and efficient genome reconstruction from metagenome assemblies. *PeerJ*.

[7] Sczyrba, A., Hofmann, P., Belmann, P., Koslicki, D., Janssen, S., Dröge, J., Gregor, I., Majda, S., Fiedler, J., Dahms, E., Bremges, A., Fritz, A., Garrido-Oter, R., Jørgensen, T. S., Shapiro, N., Blood, P. D., Gurevich, A., Bai, Y., Turaev, D., DeMaere, M. Z., Chikhi, R., Nagarajan, N., Quince, C., Meyer, F., Balvočiūtė, M., Hansen, L. H., Sørensen, S. J., Chia, B. K., Bertrand, D., Froula, J., Wang, Z., Egan, R. W., Kang, D., Cook, J., Deltel, C., Beckstette, M., Lemaitre, C., Peterlongo, P., Rizk, G., Lavenier, D., Wu, Y. W., Singer, S. W., Jain, C., Strous, M., Klingenberg, H., Meinicke, P., Barton, M. D., Lingner, T., Lin, H.-H., Liao, Y.-C., Silva, G. G. Z., Cuevas, D. A., Edwards, R. A., Saha, S., Piro, V. C., Renard, B. Y., Pop, M., Klenk, H., Göker, M., Kyrpides, N. C., Woyke, T., Vorholt, J. A., Schulze-Lefert, P., Rubin, E. M., Darling, A. E., Rattei, T., and McHardy, A. C. (2017). Critical Assessment of Metagenome Interpretation—A benchmark of metagenomics software. *Nature Methods*, 14(11):1063–1071.

Apéndice A

Anexos

A1. Parametros de compilación

Argumentos
–default-stream per-thread
-std=c++17
-lboost_program$_o ption$
-lboost_filesystem
-lboost_graph
-lboost_serialization
-O3
-arch=native
-use_fast_math
-Xcompiler=march=native -pthread -fopenmp"

Cuadro A1.1: Parámetros de compilación para pruebas

A2. Tablas de verificación

A2.1. Cami High

Semilla	Bins Metabat 2	Bins Metabat 2 CUDA	Similitud de salida
434612491	284	284	100 %
282576595	287	287	100 %
538520321	284	284	100 %
340863112	282	282	100 %
804425096	281	281	100 %
643540902	281	281	100 %
139671728	281	281	100 %
328954229	281	281	100 %
386315070	281	281	100 %
917311112	289	289	100 %

Cuadro A2.1: Bins generados para Cami High sin usar abundancia.

Semilla	Bins Metabat 2	Bins Metabat 2 CUDA	Similitud de salida
547550071	725	725	100 %
961841042	726	726	100 %
585792072	726	726	100 %
767491016	725	725	100 %
457848830	725	725	100 %
767116819	724	724	100 %
365813617	723	723	100 %
901965894	722	722	100 %
523711998	725	725	100 %
926567801	728	728	100 %

Cuadro A2.2: Bins generados para Cami High usando abundancia.

A2.2. Case 1

Semilla	Bins Metabat 2	Bins Metabat 2 CUDA	Similitud de salida
734804257	56	56	100 %
161809931	57	57	100 %
48752322	56	56	100 %
626539115	56	56	100 %
911579261	57	57	100 %
921068153	57	57	100 %
171694025	57	57	100 %
640761160	56	56	100 %
44836503	57	57	100 %
116997218	56	56	100 %

Cuadro A2.3: Bins generados para Case 1 sin usar abundancia

Semilla	Bins Metabat 2	Bins Metabat 2 CUDA	Similitud de salida
401767737	103	103	100 %
613429792	103	103	100 %
603095903	104	104	100 %
202887033	105	105	100 %
839133831	104	104	100 %
329219492	106	106	100 %
263500992	104	104	100 %
541598217	104	104	100 %
719474621	105	105	100 %
282151686	104	104	100 %

Cuadro A2.4: Bins generados para Case 1 usando abundancia

A2.3. Case 2

Semilla	Metabat 2	Metabat 2 CUDA	Similitud de salida
878346181	150	150	100 %
963348715	150	150	100 %
560636253	150	150	100 %
538219179	150	150	100 %
868358024	150	150	100 %
890974843	150	150	100 %
961922000	150	150	100 %
219891135	150	150	100 %
384467160	150	150	100 %
238493292	150	150	100 %

Cuadro A2.5: Bins generados para Case 2 sin usar abundancia

Semilla	Metabat 2	Metabat 2 CUDA	Similitud de salida
689860067	480	480	100 %
794905832	480	480	100 %
602859138	480	480	100 %
412035450	480	480	100 %
838031620	480	480	100 %
563948780	480	480	100 %
314584548	480	480	100 %
166610643	480	480	100 %
449038647	480	480	100 %
485311951	480	480	100 %

Cuadro A2.6: Bins generados para Case 2 sin usar abundancia

A2.4. Case 3

Semilla	Metabat 2	Metabat 2 CUDA	Similitud de salida
223057545	348	348	100 %
893872570	342	342	100 %
947040296	342	342	100 %
362573142	341	341	100 %
790273756	338	338	100 %
792699031	341	341	100 %
280106340	340	340	100 %
871283339	341	341	100 %
367116889	340	340	100 %
433141497	346	346	100 %

Cuadro A2.7: Bins generados para Case 3 sin usar abundancia

Semilla	Metabat 2	Metabat 2 CUDA	Similitud de salida
709405180	1283	1283	100 %
238977916	1286	1286	100 %
57571408	1286	1286	100 %
398474619	1283	1283	100 %
595427019	1283	1283	100 %
441025202	1282	1282	100 %
815504665	1285	1285	100 %
712847646	1284	1284	100 %
758976167	1283	1283	100 %
123170595	1286	1286	100 %

Cuadro A2.8: Bins generados para Case 3 sin usar abundancia