UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN

ACELERACIÓN DE METABAT MEDIANTE LA INTEGRACIÓN DE COMPRESORES GPU NVIDIA

POR

Hernán Vicente Caro Quijada

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero(a) Civil Informático

Profesor Guía

Cecilia Hernández

Marzo 2024

Concepción (Chile)

AGRADECIMIENTOS

Se agradece a proyecto FONDECYT regular 1220960 por el apoyo financiero para la ejecución de la memoria.

A la Dra. Cecilia Hernández se le agradece su disposición y orientación que han sido fundamentales en la realización de este trabajo.

A mis padres, familia y amigos agradezco profundamente su apoyo, respaldo y buenos deseos brindados durante la realización de la memoria y a lo largo de mi vida.

Resumen

Un problema del área de la Metagenómica es la identificación de los diversos genomas encontrados en muestras obtenidas del ambiente, esto se debe a que las muestras contienen una gran cantidad de material genético que puede pertenecer a diversos microorganismos. MetaBAT es una herramienta desarrollada en C++ utilizada para resolver dicho problema, ya que recibe como entrada las muestras procesadas en forma de contigs, para posteriormente entregar como resultado clusters de contigs, donde cada cluster constituye las secuencias genómicas de un genoma. El aporte de esta herramienta es de utilidad en el contexto ecológico y en particular, en el estudio de microorganismos del ser humano, el trabajo realizado por los investigadores de esta área puede resultar muy beneficioso para lograr el diagnóstico, tratamiento y control de enfermedades que pueden provocar dichos microorganismos.

Cómo se puede presumir los datasets utilizados por MetaBAT son de gran magnitud, es por ello que en esta memoria de título se llevó a cabo el análisis e integración de compresores de datos acelerados por GPU NVIDIA, con el propósito de disminuir el tiempo de compresión y descompresión de estos. Además, se abordó la fase de clustering en MetaBAT con un algoritmo igualmente acelerado por GPU NVIDIA, para disminuir el tiempo de procesamiento del clustering a la vez que se conserva una calidad del resultado similar a la original.

Palabras clave – Paralelismo, Aceleración, GPU, NVIDIA, MetaBAT, Compresión, Clustering, Metagenómica, ADN

Tabla de Contenidos

Lista de Tablas

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introducción

MetaBAT es una herramienta desarrollada en C++ utilizada ampliamente en el área de la Metagenómica para el clustering de muestras de ADN obtenidas del ambiente [4]. Estas muestras contienen una gran cantidad de material genético perteneciente a diversos microorganismos y el uso de MetaBAT ayuda a la identificación de los diversos genomas encontrados en el ambiente. Para la utilización de MetaBAT, primero se necesita obtener la secuenciación de las muestras y el ensamblaje de las lecturas (reads) obtenidas para conseguir lecturas más largas denominadas contigs. MetaBAT utiliza como entrada los contigs y obtiene clusters de contigs donde cada cluster constituye las secuencias genómicas de un genoma. La identificación de estos genomas es de utilidad en el contexto ecológico y para la identificación de la microbiota (conjunto de organismos que coexisten en un órgano) de un organismo como el humano. En particular, en el estudio de la microbiota humana en algún órgano como el intestino, el trabajo realizado por los investigadores puede resultar muy beneficioso para lograr el diagnóstico, tratamiento y control de enfermedades que pueden provocar los microorganismos.

Sin embargo, MetaBAT demanda una gran cantidad de recursos para llevar a cabo su función. Primero los datasets de entrada son de gran tamaño, usualmente en el orden de los gigabytes, los cuales usan formato FASTA. Segundo, para ser procesados en un menor tiempo, en la mayoría de sus fases utiliza paralelismo a nivel de CPU usando OpenMP.

El rendimiento de esta herramienta repercute de forma significativa en el trabajo que llevan a cabo sus usuarios. Por lo tanto, en la memoria de título desarrollada por *Jonathan Venegas*, se

ha utilizado paralelismo en GPU NVIDIA usando CUDA [8]. Este enfoque es promisorio dado que las GPU son cada vez más comunes de encontrar en computadores de propósito general incluyendo laptops.

Para contextualizar el trabajo de esta memoria de título primero se presenta lo realizado en el trabajo de *Jonathan* en la aceleración de MetaBAT. En particular la versión MetaBAT2:

1. Lectura de dataset de entrada FASTA sin comprimir, incorporando paralelismo en CPU para la obtención de los contigs y sus etiquetas.
2. El cálculo de la TNF (Frecuencia de Tetranucleótidos), acelerado con multithreading en CPU.
3. La formación del pregrafo usado por MetaBAT, acelerado a través del uso de GPU.
4. La creación del grafo usado por MetaBAT en el proceso de clustering, acelerado con GPU.

En esta memoria de título se ha decidido explorar aquellas etapas que no fueron abordadas en el trabajo de *Jonathan*. Definiendo el énfasis en la primera etapa que corresponde a la manipulación de los archivos de entrada, donde se pretende utilizar compresores desarrollados en CUDA para acelerar la obtención de datos y reducir el espacio de almacenamiento en disco.

1.1. Objetivos

A continuación, se describe el objetivo general y los específicos de este trabajo.

1.1.1. Objetivo General

El objetivo general del trabajo consiste en analizar el uso de compresores de datos desarrollados en GPU NVIDIA usando CUDA para su integración a la herramienta de análisis metagenómico MetaBAT2 para mejorar el desempeño en la obtención del archivo de entrada reduciendo el uso de espacio en disco.

1.1.2. Objetivos Específicos

1. Explorar distintas herramientas y bibliotecas de compresión desarrolladas para GPU NVIDIA, tales como CURC y nvCOMP, para seleccionar las más adecuadas para ser utilizadas en MetaBAT2.
2. Implementar e integrar las herramientas y bibliotecas seleccionadas en (1) en MetaBAT2 usando paralelismo en CPU y GPU a través de CUDA/C++.
3. Realizar una evaluación experimental que compare el impacto en tiempo de cómputo y razón de compresión usando la integración obtenida en (2).
4. Evaluar y aplicar paralelismo en CPU y GPU dentro de la fase de clustering de MetaBAT2.

1.2. Antecedentes del Problema

La entrada de MetaBAT2 consiste en dos archivos, donde el primero de ellos corresponde a un archivo FASTA comprimido en GZIP el cual almacena en su interior contigs, es decir, segmentos con conjuntos de secuencias de ADN de diversos microorganismos expresadas en forma de caracteres los cuales a su vez representan las siglas de las bases nitrogenadas (A, G, C, T). Además, cada contig va acompañado de una etiqueta que va inserta en el mismo archivo. El segundo archivo, denominado archivo de abundancia, almacena la media y la varianza de la profundidad de cobertura base de cada contig existente en el primer archivo. El uso del archivo de abundancia es opcional para MetaBAT2.

Aquí lo que resalta es el primer archivo, dado que descomprimido, tiene por lo general un tamaño en el orden de gigabytes que crece a medida que aumenta el número y largo de los contigs. Esto se debe tener en cuenta ya que por lo general se tendrán varios archivos de este tipo en un mismo computador. Más importante aún, este volumen de datos afecta directamente en el desempeño de MetaBAT2, dado que internamente lo lee y lo descomprime para posteriormente procesarlo. La descompresión propiamente tal se realiza a través de las funciones de la biblioteca Z-Lib y kseq, las cuales en conjunto realizan la descompresión al mismo tiempo que la lectura del archivo usando recursos de un solo hilo de CPU.

Por lo tanto, este archivo FASTA presenta dos aristas importantes, la primera es reducir su espacio y la segunda corresponde al tiempo de descompresión del archivo. Luego, se pretende usar un compresor/descompresor mejor a Z-Lib para con ello minimizar el espacio y tiempo de cómputo de esta etapa.

Capítulo 2

Revisión del estado del arte

2.1. Paradigma de programación en las GPU

De acuerdo a NVIDIA, la computación acelerada por GPU es el uso de una Unidad de Procesamiento de Gráficos (GPU) junto a una CPU para acelerar el funcionamiento de las aplicaciones de aprendizaje profundo, análisis e ingeniería[1].

La Arquitectura Unificada de Dispositivos de Cómputo o CUDA, por sus siglas en inglés, es la plataforma que permite desarrollar aplicaciones en C/C++ con las GPU de NVIDIA. Si bien existen otros enfoques unificados para la programación de GPU de otros proveedores como OpenCL y OpenACC, CUDA es hoy en día el marco de paralelización predominante en GPU [11].

La enorme capacidad de cómputo de las GPU se puede explicar por el gran número de unidades de procesamiento, que normalmente supera unos pocos miles de núcleos. A diferencia de las CPU multinúcleo que poseen unas pocas decenas [11]. Cabe señalar que las unidades de procesamiento de una GPU son más simples y en general alcanzan una menor potencia que los núcleos de una CPU.

Luego para aprovechar al máximo las GPU se deben tener en cuenta algunos factores:

- **Organización de las subtareas:** Al paralelizar se divide el trabajo de una tarea en

[1] https://www.nvidia.com/es-la/drivers/what-is-gpu-computing/

múltiples subtareas que realizan parte de ese trabajo de manera simultánea. Una hebra o thread corresponde a un subproceso individual que ejecuta una única subtarea durante un tiempo determinado. Las múltiples unidades de cómputo de las GPU son capaces de ejecutar muchas hebras simultáneamente.

El kernel es el código a ser ejecutado en la GPU, en este código se define el trabajo de las subtareas. Cuando se ejecuta el kernel, de acuerdo a lo definido por el desarrollador, las subtareas se organizarán en una grilla. Esta grilla está compuesta por bloques donde cada bloque tiene un número igual de hebras. Es importante señalar que la cantidad y tamaño de los bloques es un parámetro que repercute finalmente en el desempeño obtenido.

- **Comunicación entre host y device:** Un ejemplo básico de comunicación entre el host y el device es la llamada al kernel. Esta llamada siempre se realiza desde el host (CPU) y, si es sincrónica, la hebra de CPU padre espera por el resultado del kernel que se ha alojado en el device (GPU), para poder continuar su ejecución.

  Esta comunicación también se realiza con los conjuntos de datos a procesar. Los datos del disco que se han guardado en la RAM del host, primero deben ser copiados a la memoria del device para poder ser procesados por la GPU. En CUDA esta copia puede realizarse en ambas direcciones, es decir, desde el host al device y desde el device al host, la cual tiene un costo implícito.

  La memoria principal del device, que es donde se realiza la copia al host, es la memoria global, cuya capacidad varía según el modelo de la GPU. Además de la memoria global, en la GPU se encuentran otras unidades de almacenamiento más veloces: la siguiente más veloz es la memoria compartida, la cual pertenece a un bloque y es compartida por las hebras de dicho bloque, le sigue la memoria local que corresponde a la memoria propia de cada hebra y la memoria constante que se aloja junto a la global pero es de solo lectura. El alojamiento de los datos en estas memorias repercutirá finalmente en el desempeño del kernel.

A continuación, se describirán la biblioteca de compresión nvCOMP desarrollada por NVIDIA y la biblioteca CURC.

2.2. nvCOMP

Como se indica en su sitio web[2], nvCOMP es una biblioteca desarrollada por NVIDIA, la cual proporciona compresión y descompresión con compatibilidad solo para las GPU de NVIDIA. Cuenta además con interfaces de compresión genéricas para permitir a los desarrolladores utilizar distintos compresores GPU de alto rendimiento en sus aplicaciones, esto a través de API que son accesibles una vez es integrada su biblioteca.

Es importante señalar que, si bien es posible obtener y hacer uso de esta biblioteca y sus API de forma gratuita a través de un registro en su sitio web, actualmente con las últimas versiones de nvCOMP (a partir de la versión 2.3 lanzada en abril de 2022) el código fuente ya no está disponible de manera pública, es decir, el código fuente de la biblioteca es totalmente privado. Esto produce que su exploración sea limitada, más teniendo en cuenta que al ser un desarrollo reciente (lanzado en julio de 2020) la información que podemos obtener se limita a lo que encontramos en su sitio web y otros sitios que han probado esta herramienta. Por otro lado, esta situación obliga a utilizar un compilador intermediario para lograr incorporar esta biblioteca a cualquier programa. En particular, en este caso, como MetaBAT2 está basado en C++ es necesario usar CMake como compilador para incorporar nvCOMP en MetaBAT2.

NvComp proporciona dos tipos de API. La primera corresponde a la API de bajo nivel, orientada a la compresión por lotes que requiere empaquetar varios fragmentos independientes que pueden o no estar contiguos en memoria. La segunda corresponde a la API de alto nivel, la cual requiere menos parámetros y especificación técnica en comparación a las de bajo nivel, dado que están orientadas para comprimir un único buffer [12].

En unas primeras pruebas utilizando esta herramienta se pudo corroborar su correcto funcionamiento y que además permite comprimir distintos tipos de archivos. Eso si, debido a la amplia gama de compresores que posee es necesario un análisis experimental del set completo de compresores disponibles. Para este trabajo en particular se considerarán todos los existentes en la última versión disponible de nvCOMP a la fecha de la realización de este trabajo, la cual es la 3.0.4 lanzada en octubre de 2023.

[2] https://developer.nvidia.com/nvcomp

El análisis técnico y el detalle de cada compresor y su viabilidad para el caso particular estará descrito en el Capítulo 3.

2.3. CURC

CURC fue desarrollado en mayo de 2022 y como indica en su artículo [14], es un compresor de lectura libre de referencias acelerado por GPU y CPU orientado a la compresión y descompresión de archivos FASTQ. A diferencia de los archivos FASTA, los archivos FASTQ almacenan reads, que son lecturas mas cortas que los contigs y además, junto a cada read, se almacena la calidad del mismo.

CURC, a diferencia de nvCOMP, no es una biblioteca o una API que se pueda integrar en otro programa, ya que es un programa diseñado para ser compilado y se puede usar a través de la línea de comandos. Si bien en un comienzo se pensó que CURC podría utilizarse con archivos en formato FASTA debido a la similitud de este formato con FASTQ, finalmente CURC no posee dicha compatibilidad.

No obstante, se decidió probar CURC con archivos FASTQ, el cual funcionó correctamente logrando una razón de compresión aparente de 80 veces. CURC únicamente comprime los reads contenidos en el archivo FASTQ, por lo que si consideramos lo que comprime realmente versus lo que descomprime tenemos que, sin preservar el orden de los reads, la razón de compresión es de 30 y preservando el orden de los reads es de 11. Esto es muy bueno teniendo en cuenta que la razón de compresión que se tiene con GZIP es de aproximadamente 3.5, eso sí, con GZIP no se pierde la integridad del archivo original como si ocurre con CURC.

Considerando lo anterior y el potencial de CURC para con MetaBAT, se pueden generar conjeturas tales como:

- Existen conversores «FASTA to FASTQ» y es posible utilizarlos junto con otra herramienta que se encargue de adaptar los datasets de MetaBAT, para lograr comprimir dichos datasets con CURC. Sin embargo, esto aumenta el tiempo de compresión y además, dado que no se comprimen las etiquetas de los contigs, si se quiere tener el archivo FASTA original con sus etiquetas, el usuario deberá conservar una copia de este, lo que produce mayor ocupación

del almacenamiento en disco.

- CURC es de código libre, lo que abre las puertas para modificarlo e intentar adaptarlo para que funcione con archivos FASTA. Esto al parecer es posible pero requiere una modificación a fondo de CURC, pues todos sus métodos están pensados para procesar archivos FASTQ. De acuerdo a lo visto, el uso de múltiples librerías y la extensión del código fuente demandan un tiempo de análisis y desarrollo que no es prometedor.

En conclusión, se ha descartado continuar por la vía de desarrollo con CURC, a favor del desarrollo con nvCOMP debido a la revisión y conjeturas realizadas.

Capítulo 3

Gestión de compresión/descompresión usando nvCOMP

En el capítulo 2 se menciona que se decide utilizar nvCOMP y evaluar todos los compresores que proporciona. En este capítulo se describen los experimentos realizados para evaluar todos los compresores disponibles.

La primera evaluación consistió en usar un archivo FASTA de entrada de 100 MB y usar la API de alto nivel con el compresor LZ4, la cual logró comprimir con éxito. Sin embargo al utilizar un archivo de entrada de 1 GB, la API arrojó una advertencia indicando que la memoria disponible en GPU era insuficiente, tal como se muestra en la Figura 3.1.

```
WARNING: In nvcompBatchedGdeflateCompressGetTempSize: Might not have enough memory available on this GPU.
    Require 5.50082 GB but detected only 0 GB available
terminate called after throwing an instance of 'std::runtime_error'
  what():  Encountered Cuda Error: 2: 'out of memory'.
Aborted
```

Figura 3.1: Mensaje de error arrojado por LZ4 con un archivo de 1 GB

La GPU utilizada en esta pruebas en particular es una «NVIDIA RTX 3050 Laptop»[1] con 4 GB de memoria dedicada, una cantidad insuficiente en este caso pues se requerían adicionalmente 5.5 GB para poder realizar la compresión de acuerdo al mensaje entregado.

En una segunda prueba, con un archivo de 2 GB, la advertencia se volvió a mostrar, ahora

[1] https://www.nvidia.com/es-es/geforce/laptops/compare/30-series/

indicando que se requerían 11.0 GB de memoria adicionales. Al revisar la tabla comparativa de rendimiento de los compresores de nvCOMP con diferentes datasets en su sitio web (Figura 3.2), se pudo comprobar que utilizaron unas GPU de alta gama orientadas a estaciones de trabajo y centros de datos, estas son la «NVIDIA H100 PCIe»[2], «NVIDIA A100»[3], «NVIDIA A30»[4] y «NVIDIA A10»[5] cuyas memorias de GPU dedicada son de 80 GB para las dos primeras y 24 GB para las dos últimas, lo que dada la evidencia, permitiría comprimir un archivo FASTA de hasta 7 GB con LZ4.

		H100 PCIe			A100			A30			A10		
		Ratio	Compression Throughput (GB/s)	Decompression Throughput (GB/s)	Ratio	Compression Throughput (GB/s)	Decompression Throughput (GB/s)	Ratio	Compression Throughput (GB/s)	Decompression Throughput (GB/s)	Ratio	Compression Throughput (GB/s)	Decompression Throughput (GB/s)
Data analytics: INT columns													
	lz4	35.82	244.54	472.78	35.82	200.97	369.79	35.82	130.15	224.72	35.82	107.22	228.71
	snappy	16.01	81.88	225.15	16.01	52.69	204.29	16.01	27.20	116.39	16.01	28.26	128.33
	cascaded	46.52	459.72	904.24	46.52	382.10	730.33	46.52	204.29	399.18	46.52	306.64	401.68
	gdeflate-high-throughput	29.60	96.13	278.39	29.60	69.81	221.16	29.60	43.92	120.99	29.60	36.85	160.43
	gdeflate-high-compression	43.53	0.21	282.89	43.53	0.21	217.86	43.53	0.11	118.27	43.53	0.11	155.84
	gdeflate-entropy-only	1.59	97.10	60.03	1.59	47.13	46.53	1.59	28.14	25.70	1.59	28.92	36.47
	bitcomp-default	21.72	684.67	361.31	21.72	483.95	351.20	21.72	257.90	195.94	21.72	359.04	262.75
	bitcomp-sparse	1.00	708.59	595.60	1.00	659.84	556.92	1.00	342.58	307.51	1.00	179.62	218.01
	deflate	40.53	85.45	234.77	40.53	62.92	188.14	40.53	38.52	107.62	40.53	30.69	132.51
	ans	1.58	244.23	350.64	1.58	207.85	269.36	1.58	111.93	145.60	1.58	98.12	161.32
	zstd	94.97	88.74	405.05	94.97	64.24	281.14	94.97	36.11	167.45	94.97	36.86	161.32
Silesia													
	lz4	1.97	9.58	43.79	1.97	7.11	34.13	1.97	4.67	22.07	1.97	3.66	26.59
	snappy	1.99	13.18	40.20	1.99	9.00	40.70	1.99	5.21	25.84	1.99	5.09	27.51
	cascaded	1.08	66.96	262.50	1.08	44.14	172.06	1.08	24.36	98.61	1.08	21.71	89.22
	gdeflate-high-throughput	2.37	7.86	60.77	2.37	5.74	38.34	2.37	3.73	22.82	2.37	2.87	32.41
	gdeflate-high-compression	3.01	0.31	52.44	3.01	0.27	32.93	3.01	0.16	20.20	3.01	0.15	29.22

Figura 3.2: Tabla comparativa de rendimiento con distintos tipos de datasets para nvCOMP

Además del hecho de que existen datasets de entrada con un tamaño superior a los 7 GB, tenemos que no es común encontrar las gráficas antes descritas en los computadores de propósito general. Esto evidencia que se requiere de un software que gestione la compresión y la descompresión usando las API de nvCOMP a la vez que permita dividir los datasets de entrada en particiones lo suficientemente pequeñas para ser utilizadas en las GPU de NVIDIA, en particular, aquellas que se encuentran en computadores generales y cuentan con capacidades de memoria de GPU en un rango de entre 4 y 12 GB. Este último dato fue deducido de acuerdo con un estudio realizado en junio de 2023 por un sitio web [1] dedicado a tecnología que se basa en las búsquedas de Google para determinar cuáles son las tarjetas gráficas más populares (más buscadas) a partir de 2017.

[2] https://www.nvidia.com/es-la/data-center/h100/
[3] https://www.nvidia.com/es-la/data-center/a100/
[4] https://www.nvidia.com/es-la/data-center/products/a30-gpu/
[5] https://www.nvidia.com/es-la/data-center/products/a10-gpu/

3.1. Gestor de compresión y descompresión

Como se menciona en la revisión del estado del arte, nvCOMP tiene dos tipos de API, esto para cada uno de los algoritmos de compresión que posee en su biblioteca. Luego, para diseñar e implementar el módulo que gestione la compresión y descompresión de archivos por particiones se decidió usar las API de alto nivel. La idea es realizar la compresión y descompresión de una única partición a la vez de manera que la API pueda procesarla con una cantidad de memoria de GPU reducida y a la vez que mejore el tiempo de procesamiento.

3.1.1. Descripción general del gestor de compresión

El gestor de compresión divide el archivo de entrada en múltiples particiones. La Figura 3.3 proporciona un esquema de proceso completo y el Algoritmo 1 presenta el algoritmo general. Cada partición (P_i) se lee usando paralelismo multi-hilo en CPU, donde cada partición se divide en porciones que llamamos chunks y cada hilo procesa un chunk. El tamaño de cada chunk está determinado por el tamaño de la partición y los hilos disponibles en la CPU. Una vez leída la partición actual se copia a memoria de la GPU. Luego, se ejecuta un algoritmo de compresión de nvCOMP para la partición P_i, mientras múltiples hilos de CPU escriben los chunks de la partición anterior (P_{i-1}) comprimida y ya copiada al host de forma asincrónica. Este procedimiento continúa iterativamente para procesar todas las particiones, es decir, para $1 < i \leq n$. Como se observa en la figura, la primera partición (P_1) no posee partición P_{i-1} por lo que no se crean hilos en la CPU para escritura asincrónica. Finalmente, el gestor de compresión escribe un archivo de metadata en disco con los largos de las particiones comprimidas (líneas 28 a la 34 en Algoritmo 1).

Algorithm 1 Algoritmo del gestor de compresión con nvCOMP

Input Archivo F de extensión .fasta o .fa y algoritmo seleccionado por el usuario *alg*
Output Archivo *F'* de extensión .*«alg»* (t.q. length(F') $\leq$ length(F))

```
F' ← ∅
offsetR ← 0
offsetW ← 0
arrThreads ← ∅
arrSizePart ← ∅
partSize ← getBestPartSize()                ▷ Definido de acuerdo a mejor desempeño
do
    S ← ∅
    partSizeR ← 0
    if length(F) >= offsetR + partSize then
        partSizeR ← partSize
    else if offsetR < length(F) then
        partSizeR ← length(F) − offsetR
    end if
    S ← generatePartition(F, offsetR, partSizeR)
    offsetR ← offsetR+ length(S)
    if length(S) > 0 then
        S_device ← ∅
        cudaMemcpy(S_device ,S, length(S), cudaMemcpyHostToDevice)
        free(S)
        C ← Compress(S_device, alg)
        th ← createThread(function that writes partition, F', C, offsetW)
        arrThreads.insert(th)
        offsetW ← offsetW+length(C)
        arrSizePart.insert(length(C))
    end if
while length(S) > 0
metadata ← ∅
for i ← 0 to length(arrSizePart) - 1 do
    metadata.insert(arrSizePart[i])
end for
metadata.insert(length(F))
join(arrThreads)
pwrite(F', metadata, offsetW)
```

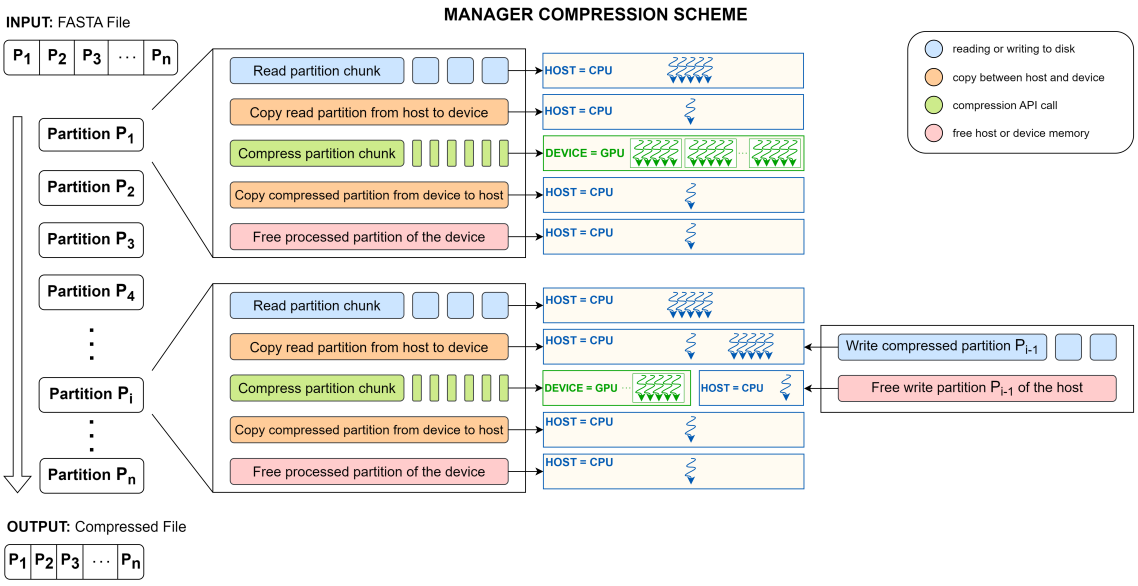


Figura 3.3: Flujo de procesamiento del Gestor de compresión

3.1.2. Descripción general del gestor de descompresión

La gestión de descompresión es análoga a la de compresión, tal como se muestra en la Figura 3.4 y el algoritmo correspondiente en el Algoritmo 2. Para la descompresión se usa el metadata generado en la etapa de compresión conteniendo los largos de las particiones comprimidas.

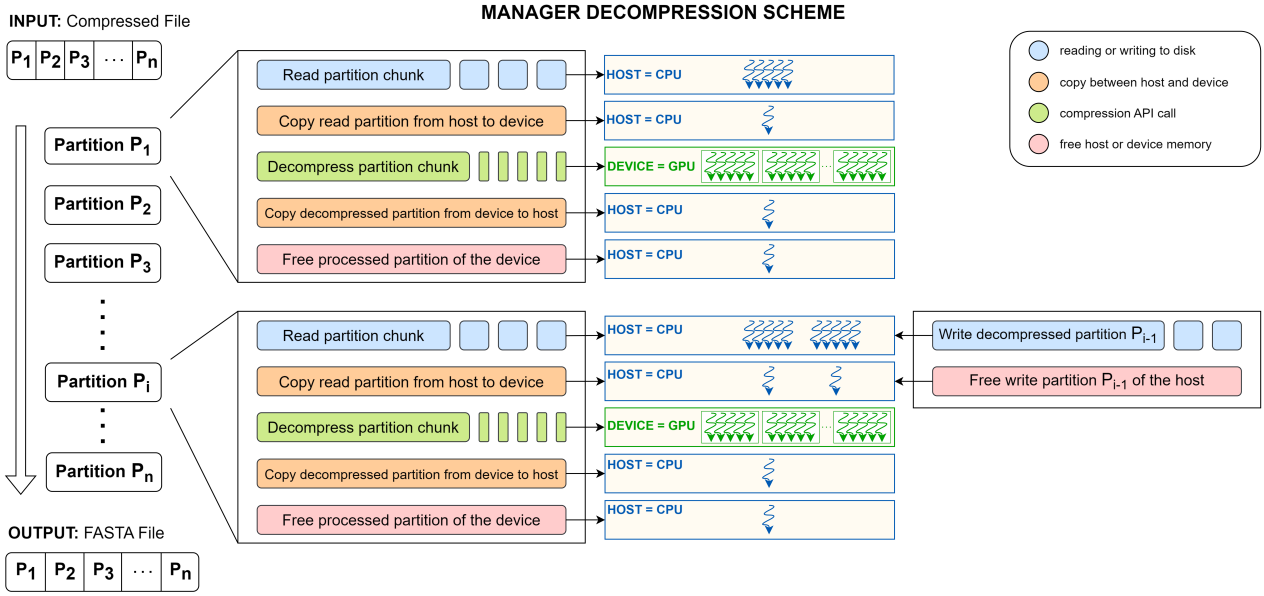


Figura 3.4: Flujo de procesamiento del Gestor de descompresión

Algorithm 2 Algoritmo del gestor de descompresión con nvCOMP

Input Archivo *F'* de extensión *.«alg»*
Output Archivo F de extensión .fasta (t.q. length(F') $\leq$ length(F))

1: $alg \leftarrow$ getExtension(F')
2: $F \leftarrow \varnothing$
3: *offsetR* $\leftarrow 0$
4: *offsetW* $\leftarrow 0$
5: $arrThreads \leftarrow \varnothing$
6: *arrSizePart* $\leftarrow$ getMetadata(*F'*, *offsetR*)
7: *partSize* $\leftarrow$ getBestPartSize() ▷ Definido de acuerdo a mejor desempeño
8: **for** $i \leftarrow 0$ to length($arrSizePart$) - 1 **do**
9: $C \leftarrow$ readPartition(*F'*, *offsetR*, *arrSizePart*[i])
10: *offsetR* $\leftarrow$ *offsetR*+ length(C)
11: $C_{device} \leftarrow \varnothing$
12: cudaMemcpy(C_{device} ,C, length(C), cudaMemcpyHostToDevice)
13: free(C)
14: $S \leftarrow$ Decompress(C_{device}, alg)
15: $th \leftarrow$ createThread(*function that writes partition*, F, S, *offsetW*)
16: $arrThreads$.insert(th)
17: *offsetW* $\leftarrow$ *offsetW*+length(S)
18: **end for**
19: join($arrThreads$)

Aunque existen otras formas de ordenar el flujo de procesamiento incluyendo utilizar particiones más pequeñas para realizar múltiples llamadas a la API o realizar la lectura y la escritura de una partición completa sin divisiones. Durante el periodo de desarrollo, se determinó experimentalmente que la manera presentada proporciona mejor desempeño.

Por último, cabe mencionar que el Gestor de compresión y descompresión no solo está orientado a funcionar con las API de alto nivel de nvCOMP, sino que además, es posible adaptarlo fácilmente a cualquier otro compresor. Más adelante se presentará una alternativa que se adapta a este esquema para pasar a ser una opción elegible más.

A continuación, se presenta con mayor detalle a los componentes más importantes del gestor de compresión y descompresión.

3.1.3. Compresión y descompresión

La partición leída es copiada de la memoria del host a la memoria del device, luego se define el tamaño de los chunks que considerará la API y el tipo de dato al que corresponde la partición a procesar.

La API a su vez se encarga de estimar el tamaño de salida y de realizar la acción de comprimir (Algoritmo 3) o descomprimir (Algoritmo 4) propiamente tal, esto utilizando múltiples hebras de GPU las cuales se encargarán de comprimir/descomprimir cada chunk de la partición para retornar al final del proceso una partición terminada, la cual es copiada a la memoria del host.

Algorithm 3 Algoritmo de compresión usando la API de nvCOMP

Function Compress(S_{device}, *alg*)
Input Bytes de la partición leída S en la GPU y algoritmo seleccionado por el usuario *alg*
Output Bytes de la partición comprimida C (t.q. length(C) $\leq$ length(S))

1: *stream* $\leftarrow$ getCudaStream()
2: *chunkSize* $\leftarrow 64KB$ ▷ Suggested by nvCOMP
3: *dataType* $\leftarrow$ nvcompTypeChar
4: *nvcompAPIManager* $\leftarrow$ getNvcompAPIManager(*chunkSize*, *dataType*, *stream*, *alg*)
5: $C_{device} \leftarrow$ *nvcompAPIManager*.compress(S) ▷ Kernel call
6: cudaStreamDestroy(*stream*)
7: cudaFree(S)
8: *compSize* $\leftarrow$ *nvcompAPIManager*.getCmpSize()
9: cudaMemcpy(C, C_{device}, *compSize*, cudaMemcpyDeviceToHost)
10: cudaFree(C_{device})

Algorithm 4 Algoritmo de descompresión usando API de nvCOMP

Function Decompress(C_{device}, *alg*)
Input Bytes de la partición comprimida C en la GPU y algoritmo asociado *alg*
Output Bytes de la partición descomprimida S (t.q. length(C) $\leq$ length(S))

1: *stream* $\leftarrow$ getCudaStream()
2: *chunkSize* $\leftarrow 64KB$ ▷ Suggested by nvCOMP
3: *dataType* $\leftarrow$ nvcompTypeChar
4: *nvcompAPIManager* $\leftarrow$ getNvcompAPIManager(*chunkSize*, *dataType*, *stream*, *alg*)
5: $S_{device} \leftarrow$ *nvcompAPIManager*.decompress(C) ▷ Kernel call
6: cudaStreamDestroy(*stream*)
7: cudaFree(C)
8: *dcpSize* $\leftarrow$ *nvcompAPIManager*.getDcpSize()
9: cudaMemcpy(S, S_{device}, *dcpSize*, cudaMemcpyDeviceToHost)
10: cudaFree(S_{device})

3.1.4. Lectura del archivo de entrada

El gestor creado lee una partición del archivo de acuerdo a un tamaño tentativo (Algoritmo 5), por ejemplo 500 MB, usando la función de lectura con paralelismo en CPU *pread* de la biblioteca «unistd.h»[6] junto con «OpenMP» [2]. Luego, a partir de esos 500 MB se procede a leer de a 1 MB hasta encontrar el símbolo que indica el inicio de un nuevo contig ('>') o el final del archivo ('\0'), esto para evitar que las particiones queden con contigs incompletos.

En el caso de la descompresión el archivo de metadata contiene información acerca del tamaño exacto de cada una de las particiones a descomprimir. Luego, al igual que con el Algoritmo 5 se procede a leer la partición en paralelo.

Algorithm 5 Algoritmo de lectura por particiones del archivo FASTA

Function generatePartition(F, *offsetR*, *partSizeR*)
Input Archivo F de extensión .fasta o .fa, tamaño de partición a leer *partSizeR* y cantidad de bytes leídos *offsetR*
Output Bytes del partición leída S

1: *numThreads* ← getCPUthreadFraction() ▷ The one that produces better performance
2: *chunkSize* ← *partSizeR*/*numThreads*
3: *arrOffset* ← getOffset(*numThreads*, *chunkSize*, *offsetR*)
4: *totalBytesRead* ← 0
5: $S \leftarrow \varnothing$
6: #pragma omp parallel for
7: **for** $i \leftarrow 0$ to *numThreads* − 1 **do in parallel**
8: *localSizeChunk* ← *chunkSize*
9: **if** $i =$ *numThreads* − 1 **then**
10: *localSizeChunk* ← *partSizeR* − (*chunkSize* ∗ (*numThreads* − 1))
11: **end if**
12: *bytesRead* ← pread(F, S + i * *chunkSize*, *localSizeChunk*, *arrOffset*[i])
13: **if** *bytesRead* > 0 **then**
14: #pragma omp critical
15: *totalBytesRead* ← *totalBytesRead* + *bytesRead*
16: **end if**
17: **end for**
18: $S \leftarrow$ readUntilFound(F, S, *totalBytesRead*, ['>','\0'])

[6] https://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html

3.1.5. Escritura del archivo de salida

De forma análoga a la lectura, la escritura del archivo de salida se realiza en paralelo, con la diferencia de que el proceso completo se realiza en una hebra independiente con la finalidad de que la hebra principal no sea interrumpida (modalidad asincrónica). En este caso cada chunk de la partición se escribe con *pwrite*, que pertenece igualmente a la biblioteca «unisdth».

3.2. Datos de Prueba

Para realizar las pruebas de rendimiento del gestor de compresión y descompresión fue necesario seleccionar un grupo de archivos FASTA dentro del repositorio oficial de MetaBAT[7]. La elección de estos archivos se hace en base a la magnitud de su tamaño, el tiempo de procesamiento que le toma a MetaBAT realizar su lectura para obtener la colección de contigs que porta y el tiempo que le toma al compresor, en este caso GZIP, para comprimir y guardar el archivo comprimido (cuya extensión es «.gz») en disco.

Tabla 3.1: Análisis preliminar de los datasets de prueba

FASTA File Name	Decompressed Size (MB)	Compressed Size (MB)	Ratio	Compression Time GZIP (s)	Decompression Time GZIP (s)	Read Time MetaBAT (s)
CAMI_high	2673.77	779.16	3.432	241.297	16.205	11.531
case2_assembly	4509.97	1313.97	3.432	364.767	24.326	19.510
nielsen	6472.12	1848.84	3.501	551.779	33.492	26.616
case3_assembly	8363.13	2441.04	3.426	722.192	44.023	35.279

La Tabla 3.1 presenta los datasets utilizados para la experimentación. Dentro de este set de datos de prueba se encuentra «CAMI_high» un archivo de Critical Assessment of Metagenome Interpretation (CAMI) [6] que es el más «pequeño» del set, con un tamaño de 2.7 GB estando descomprimido. Mientras que el más «grande» es «case3_assembly.fa» un archivo FASTA con un tamaño de 8.2 GB estando descomprimido. Los archivos intermedios «case2_assembly» y «nielsen» se eligieron con el objetivo mantener un tamaño de archivo descomprimido equidistante que facilite las posteriores comparaciones.

En cuanto al ambiente utilizado en las pruebas este se encuentra en un servidor cuyas especificaciones se encuentran en la Tabla 3.2.

[7] https://portal.nersc.gov/dna/RD/Metagenome_RD/MetaBAT/Files/

Tabla 3.2: Especificaciones del servidor y ambiente utilizado para las pruebas

Specifications	Details
OS	Ubuntu 20.04.6 LTS
CPU	Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz
CPU RAM (GB)	128
GPU	NVIDIA RTX A2000
GPU Architecture	Ampere
GPU memory (GB)	6
CUDA cores	3,328
CUDA version	12.1
Compiler	CMake 3.28.0

3.3. Análisis y Evaluación Experimental de nvCOMP

Actualmente se tienen diversas implementaciones de algoritmos de compresión y descompresión ya existentes dentro de la biblioteca de nvCOMP. En esta sección se proporciona el procedimiento de evaluación y una descripción general de cada algoritmo.

Cada compresor se evaluó experimentalmente utilizando el gestor de compresión, donde se definieron distintas configuraciones con fracciones de hebras (TF) de CPU utilizadas para la lectura y escritura, y con diversos tamaños de partición. Para ello se varió el TF como $\{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1\}$ y tamaños de particiones $\{100, 256, 512, 768, 1024\}$. El objetivo fue encontrar la configuración que permite obtener el mejor tiempo de cómputo para cada algoritmo de compresión disponible en nvCOMP. Para cada dataset descrito en la Tabla 3.1, el procedimiento para seleccionar aquella configuración es el siguiente:

1. **Obtención de tiempos promedios de compresión y descompresión por configuración, dataset y algoritmo:** Se ejecuta el gestor, el cual realiza la compresión y descompresión de un dataset S_i. Tanto la compresión como la descompresión se ejecutó 20 veces manteniendo una configuración k definida por un tamaño de partición y fracción de hebras de CPU para la lectura y la escritura. Luego, en Eqs. 3.1 y 3.2 se define $T_c(k, a, S_i)$ como el tiempo promedio para una configuración k, algoritmo a, y dataset S_i. Así mismo, $T_d(k, a, S_i)$ denota el tiempo de descompresión promedio por configuración k, algoritmo a y dataset S_i.

$$T_c(k, a, S_i) = \frac{\sum_{j=1}^{20} compTime(k, a, S_i)}{20} \tag{3.1}$$

$$T_d(k, a, S_i) = \frac{\sum_{j=1}^{20} descompTime(k, a, S_i)}{20} \tag{3.2}$$

2. **Obtención de tiempos promedios de compresión y descompresión por configuración y algoritmo de nvCOMP:** En este caso, el objetivo es determinar el tiempo promedio para todos los datasets. Para ello se obtienen los tiempos promedios conseguidos por algoritmo usando todos los datasets tal como se presenta en Eqs. 3.3 y 3.4.

$$AVG_c(k, a) = \frac{\sum_{i=1}^{4} T_c(k, a, S_i)}{4} \tag{3.3}$$

$$AVG_d(k, a) = \frac{\sum_{i=1}^{4} T_d(k, a, S_i)}{4} \tag{3.4}$$

3. **Obtención de mejor configuración por algoritmo en términos de tiempos de cómputo:** En este caso, para cada algoritmo se determina la mejor configuración para compresión/descompresión en términos del tiempo de cómputo. Para ello, primero se pondera con mayor peso el tiempo de descompresión por sobre el de compresión y se obtiene el mínimo (T_a en Eq. 3.6), dado que la descompresión es relevante para la ejecución de MetaBAT.

$$T(k) = 0.2 \times AVG_c(k, a) + 0.8 \times AVG_d(k, a) \tag{3.5}$$

$$T_a = \min_k(T(k)) \tag{3.6}$$

Cada algoritmo de compresión y descompresión tiene una configuración con mejor desempeño que representa el rendimiento de dicho algoritmo y permite compararlo a posteriori con los demás. Sin embargo, el tiempo de compresión y descompresión del algoritmo no son los únicos parámetros que se utilizan para evaluar cada uno de ellos, si no que se considera también

el ratio o razón de compresión del algoritmo, su máximo uso de memoria de GPU dedicada para la compresión (MUC) y la descompresión (MUD), el speedup o comparación de tiempo de compresión y descompresión respecto al algoritmo de referencia GZIP y la configuración propiamente tal.

3.3.1. LZ4

Algoritmo de compresión de datos sin pérdidas[8] y de uso general, que se centra en la velocidad de compresión y descompresión. Pertenece a la familia de algoritmos de codificación LZ (Lempel-Ziv) [15] que proveen algoritmos de compresión sin pérdidas genéricos basados en diccionarios que explotan las regularidades de los datos, en particular LZ4 se basa en la variante LZ77.

Tabla 3.3: Tres mejores configuraciones de LZ4

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Comp. Time (s)	Average Comp. Time (s)	Comp. Speedup	Max. GPU MUD (GB)	Decomp. Time (s)	Average Decomp Time (s)	Decomp. Speedup
1/4	256	CAMI_high	1.438	496	720	1.0	5.393	10.724	44.7	0.6	3.714	7.221	4.4
1/4	256	case2_assembly	1.468	512	755	1.0	8.811	10.724	41.4	0.6	5.975	7.221	4.1
1/4	256	nielsen	1.486	520	774	1.0	12.457	10.724	44.3	0.6	8.363	7.221	4.0
1/4	256	case3_assembly	1.435	515	772	1.0	16.235	10.724	44.5	0.6	10.832	7.221	4.1
1/2	256	CAMI_high	1.438	503	709	1.0	5.319	10.656	45.4	0.6	3.772	7.280	4.3
1/2	256	case2_assembly	1.468	512	748	1.0	8.809	10.656	41.4	0.6	6.028	7.280	4.0
1/2	256	nielsen	1.486	522	765	1.0	12.387	10.656	44.5	0.6	8.459	7.280	4.0
1/2	256	case3_assembly	1.435	519	770	1.0	16.110	10.656	44.8	0.6	10.862	7.280	4.1
1/2	768	CAMI_high	1.438	504	678	2.8	5.307	10.427	45.5	1.4	3.946	7.371	4.1
1/2	768	case2_assembly	1.468	523	728	2.8	8.624	10.427	42.3	1.4	6.192	7.371	3.9
1/2	768	nielsen	1.486	537	761	2.8	12.055	10.427	45.8	1.4	8.501	7.371	3.9
1/2	768	case3_assembly	1.435	532	771	2.8	15.722	10.427	45.9	1.4	10.843	7.371	4.1

En la Tabla 3.3 se presentan las tres mejores configuraciones obtenidas en la evaluación experimental del algoritmo LZ4, siendo la primera de ellas (primeras 4 filas) la que entrega el mejor desempeño (configuración de referencia). También, es posible ver que con un mismo dataset la razón de compresión no varía. Esto se debe a que la diferencia en el tamaño de los archivos comprimidos al variar el tamaño de la partición aproximada, está en el orden de los kilobytes, situación que se repite con todos los algoritmos probados. Por este motivo, la comparativa en la razón de compresión se realiza únicamente entre algoritmos distintos.

Para LZ4 en particular la razón de compresión es de 1.4 y se obtiene su mejor desempeño usando una cuarta parte de las hebras de CPU para leer y otra cuarta parte para escribir, esto comprimiendo el archivo de a particiones de aproximadamente 256 MB. El máximo uso de

[8] https://lz4.org/

memoria de GPU es de 1 GB al comprimir y 0.6 GB al descomprimir, es importante notar que de haber escogido la configuración con particiones de 768 MB no solo hubiera aumentado el consumo de memoria de GPU a más del doble, sino que también los tiempos de compresión y descompresión. Aquí se debe aclarar que la diferencia en tiempos de compresión y descompresión promedio entre configuraciones para un mismo algoritmo es poco significativa cuando se ordenan y se comparan en un subconjunto contiguo, teniendo diferencias en el orden de décimas de segundo, no así cuando se comparan los extremos, es decir la peor con la mejor configuración, donde la diferencia está en el orden de los segundos.

Por último, el speedup de LZ4 señala que es 40 veces más rápido en comprimir y 4 veces más rápido en descomprimir que GZIP para el mismo dataset.

3.3.2. Snappy

Algoritmo de compresión y descompresión de datos de nivel de bytes desarrollado por Google, es de uso general al igual que LZ4 por lo que en cierta medida son parecidos. Este algoritmo no apunta a una alta razón de compresión ni a una compatibilidad en particular, en cambio apunta a ser veloz con una razón de compresión razonable[9].

Tabla 3.4: Configuración de referencia de Snappy

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Max. GPU MUD (GB)	Comp. Speedup	Decomp. Speedup
1/4	512	CAMI_high	2.040	510	802	1.8	0.9	46.0	4.9
1/4	512	case2_assembly	2.002	522	846	1.8	0.9	42.2	4.6
1/4	512	nielsen	2.043	522	889	1.8	0.9	44.5	4.6
1/4	512	case3_assembly	2.001	521	888	1.8	0.9	45.0	4.7

En la Tabla 3.4 se puede ver su desempeño, el cual es similar a LZ4, salvo en su razón de compresión es 30 % superior y el tamaño de partición que logra el mejor desempeño es de 512 MB.

3.3.3. GDeflate

Esquema de compresión de datos de alto rendimiento, escalable y optimizado para GPU desarrollado por NVIDIA [13], está basado en Deflate RFC 1951 y su objetivo inicial era

[9] https://github.com/google/snappy/

mejorar la transmisión de archivos de juegos y aplicaciones de gran volumen.

Durante las pruebas fue revelado que GDeflate presenta tres ajustes en sus API de nvCOMP que son las siguientes:

- **Ajuste 0**: Produce un alto rendimiento a cambio de una baja razón de compresión.
- **Ajuste 1**: Produce un bajo rendimiento a cambio una alta razón de compresión.
- **Ajuste 2**: Balance en rendimiento y razón de compresión.

Para las pruebas de rendimiento fue utilizado el **Ajuste 2**, puesto que produce un rendimiento mucho mayor al **Ajuste 1** y la razón de compresión prácticamente no se ve afectada.

Tabla 3.5: Configuración de referencia de GDeflate

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Max. GPU MUD (GB)	Comp. Speedup	Decomp. Speedup
1/4	512	CAMI_high	3.649	1070	862	2.6	0.8	96.6	5.2
1/4	512	case2_assembly	3.068	1024	888	2.6	0.8	82.8	4.8
1/4	512	nielsen	3.110	1049	938	2.6	0.8	89.4	4.9
1/4	512	case3_assembly	3.332	1083	963	2.6	0.8	93.5	5.1

En la Tabla 3.5 se puede ver que su razón de compresión llega a ser tan buena como la de GZIP y su rendimiento es claramente superior, siendo alrededor de 80 veces más rápido en comprimir y 5 veces más rápido en descomprimir. Además, es interesante mencionar que para un mismo tamaño de partición aproximado utiliza más recursos de GPU que Snappy.

3.3.4. Deflate

Compresor y descompresor que también es un formato de compresión de datos, utiliza internamente una implementación del algoritmo LZ77 junto con una codificación de Huffman. Actualmente existe una implementación de Deflate dentro de GZIP para la compresión de algunos archivos.

Al igual que con GDeflate, Deflate presenta los mismos tres ajustes en sus API de nvCOMP, sin embargo fue imposible utilizar el **Ajuste 2** debido a un error desconocido que es producido por la API en la compresión. Por este motivo, se optó por escoger el **Ajuste 0** que si bien produce una menor razón de compresión, es más veloz, que es lo que finalmente se busca rescatar de los algoritmos de nvCOMP.

Tabla 3.6: Configuración de referencia de Deflate

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Max. GPU MUD (GB)	Comp. Speedup	Decomp. Speedup
1/2	256	CAMI_high	2.175	474	710	2.9	0.5	42.8	4.3
1/2	256	case2_assembly	2.186	479	738	2.9	0.5	38.8	4.0
1/2	256	nielsen	2.227	483	755	2.9	0.5	41.2	3.9
1/2	256	case3_assembly	2.164	481	753	2.9	0.5	41.5	4.0

En la Tabla 3.6 se puede ver que Deflate tiene un comportamiento similar a Snappy en cuanto a rendimiento y razón de compresión. Donde difieren es en su uso máximo de memoria de GPU, ya que Deflate ocupa 3 veces más memoria para comprimir utilizando un mismo tamaño de partición aproximado.

3.3.5. Zstandard (Zstd)

Algoritmo de compresión sin pérdidas desarrollado por Meta (Facebook)[10], está basado en la codificación de Huffman y utiliza además el codificador de entropía FSE basado en ANS, combinación que promete conseguir una mayor rapidez y mejor razón de compresión respecto a Z-Lib.

Para esta implementación de Zstd en nvCOMP, se omitieron pruebas con particiones superiores a 256 MB, pues se producían errores frecuentes con la API en la descompresión, algo que no ocurrió con ninguno de los otros algoritmos.

Tabla 3.7: Configuración de referencia de Zstd

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Max. GPU MUD (GB)	Comp. Speedup	Decomp. Speedup
1/2	100	CAMI_high	3.305	498	741	0.7	0.5	45.0	4.5
1/2	100	case2_assembly	3.234	515	767	0.7	0.5	41.7	4.1
1/2	100	nielsen	3.265	502	768	0.7	0.5	42.8	4.0
1/2	100	case3_assembly	3.219	500	769	0.7	0.5	43.2	4.0

En la Tabla 3.7 se puede ver que Zstandard tiene una razón de compresión similar a GDeflate donde por ejemplo con el archivo «CAMI_high» este valor es superior y con «case_3» ligeramente inferior. Sin embargo, el rendimiento es claramente inferior a GDeflate, siendo más parecido a lo que vemos con LZ4 pero con una mejor razón de compresión.

[10] https://github.com/facebook/zstd

3.3.6. Cascaded

Esquema de compresión en cascada ideado para cargas de trabajo analíticas desarrollado por NVIDIA[11]. Su esquema de compresión es flexible en el sentido que puede ser modificado a bajo nivel para adaptarse manualmente a los datos a comprimir.

A este compresor no se le realizaron pruebas de rendimiento debido a que la razón de compresión obtenida en ensayos preliminares con los datasets de prueba fue prácticamente 1, es decir que no lograba comprimir. Revisando la documentación de Cascaded se logró comprender el por qué y es que este compresor se basa en RLE, un esquema de codificación muy simple que comprime valores repetidos en un solo par, algo que puede funcionar muy bien para datos numéricos, pero que en este caso no funcionó ya que los archivos FASTA se pueden considerar mayormente texto.

3.3.7. Bitcomp

Compresor patentado por NVIDIA[12] diseñado para una compresión en GPU eficiente y orientado a aplicaciones de computación científica. Ofrece compresión con y sin pérdidas, ya que está orientado a datos numéricos.

Tabla 3.8: Configuración de referencia de Bitcomp

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Max. GPU MUD (GB)	Comp. Speedup	Decomp. Speedup
1/4	512	CAMI_high	1.992	941	833	1.6	0.9	84.9	5.1
1/4	512	case2_assembly	1.807	956	870	1.6	0.9	77.3	4.7
1/4	512	nielsen	1.873	982	908	1.6	0.9	83.8	4.7
1/4	512	case3_assembly	1.931	998	922	1.6	0.9	86.2	4.9

Pese a estar orientado a enteros y flotantes, en la Tabla 3.8 se puede ver que Bitcomp tiene una razón de compresión cercana a Snappy, la cual es superior a LZ4 pero con un rendimiento mayor a ambos, pues tiene un throughput de compresión y descompresión cercano a GDeflate pero usando menos recursos de GPU.

[11] https://github.com/NVIDIA/nvcomp/blob/main/doc/cascaded_overview.md
[12] https://developer.nvidia.com/nvcomp

3.3.8. ANS

Sistemas Numéricos Asimétricos (ANS) es un método de codificación de entropía basado en bits que combina la velocidad de codificación de Huffman con una razón de compresión de codificación aritmética [3]. Este método para codificar y decodificar utiliza un alfabeto definido y una función de distribución de probabilidad asimétrica que se ajusta a dicho alfabeto. Al codificar ANS utiliza una representación numérica de la entrada, donde cada valor de esta entrada de forma iterativa pasa por la función de codificación, la cual realiza una ecuación con la distribución de probabilidad asimétrica y también el resultado anterior, generando un nuevo valor resultante cada vez. Al completar la codificación se obtiene finalmente un único valor cuya representación en bits es menor a la de la entrada. La distribución de probabilidad asimétrica le permite alcanzar una mayor razón de compresión que con una distribución simétrica.

Tabla 3.9: Configuración de referencia de ANS

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Max. GPU MUD (GB)	Comp. Speedup	Decomp. Speedup
1/4	512	CAMI_high	3.921	1185	905	2.7	0.8	106.9	5.5
1/4	512	case2_assembly	3.075	1111	941	2.7	0.8	89.8	5.1
1/4	512	nielsen	3.127	1137	978	2.7	0.8	97.0	5.1
1/4	512	case3_assembly	3.428	1181	1005	2.7	0.8	102.0	5.3

En la Tabla 3.9 se puede ver la variante de ANS de nvCOMP logra las razones de compresión más altas, y además, es el algoritmo de compresión/descompresión de nvCOMP que logra el throughput más alto, tanto en comprimir como en descomprimir, siendo ligeramente superior a GDeflate logrando un speedup de 100x en la compresión y de 5x en la descompresión respecto a GZIP. Esto lo convierte en el compresor de referencia de nvCOMP para archivos FASTA.

Capítulo 4

Compresor Personalizado CFA

En este capítulo se describe el diseño e implementación de dos nuevos algoritmos de compresión y descompresión para archivos FASTA. Ambos algoritmos explotan la repetición de caracteres contiguos y usan la capacidad disponible en representación ASCII para realizar la codificación. Ambos algoritmos usan paralelismo en GPU y al igual que las API de nvCOMP se integran en el gestor de compresión y descompresión descrito en la Sección 3.1.

4.1. ASCII

El Código Estándar estadounidense para el Intercambio de Información o ASCII por sus siglas en inglés, es un código de caracteres basado en el alfabeto latino, tal como se usa en inglés moderno[1]. ASCII utiliza solo 7 bits para representar caracteres, pero dado que los computadores usan por lo general múltiplos de 8 bits para representar información, entonces ASCII se representa finalmente en 8 bits. Al conjunto de caracteres que no pertenecen a ASCII pero que se pueden representar en esos 8 bits se denomina popularmente como ASCII Extendido.

4.2. CFA1

El primer algoritmo lo denominamos CFA1, el cual utiliza una función de codificación $g()$ para comprimir y una de decodificación $h()$ para descomprimir. El algoritmo codifica caracteres a

[1] https://es.wikipedia.org/wiki/ASCII

subcadenas de texto de largo tres, las cuales conforman a todas las combinaciones que se pueden obtener con las 4 bases nitrogenadas del ADN más el carácter de salto de línea.

Esto se puede definir de la siguiente manera, dado el lenguaje $L = \{A, C, G, T, \backslash n\}$ se construye el conjunto C compuesto por subcadenas s^i de largo 3, las cuales conforman a todas las combinaciones que se pueden generar de L. Dado que $|L| = 5$ y $|s^i| = 3$, entonces $|C| = |L|^{|s^i|} = 5^3 = 125$ que corresponde a la cantidad de combinaciones contenidas en C (Ecuación 4.1).

$$C = \{s^1\{A, A, A\},\ s^2\{A, A, C\},\ s^3\{A, A, G\},\ s^4\{A, A, T\},\ s^5\{A, A, \backslash n\},\ ..,\ s^{125}\{\backslash n, \backslash n, \backslash n\}\} \tag{4.1}$$

La función de codificación $g(s^i)$ utiliza los caracteres de lo que conocemos como ASCII Extendido, ya que estos caracteres no son utilizados en los archivos FASTA. De esta manera, a cada subcadena s^i de C se le asignará un carácter c del código ASCII Extendido.

$$g(s^i) = c \tag{4.2}$$

Por otro lado, la función de decodificación corresponde a la inversa a la Ecuación 4.2, es decir, $h^{-1}(s^i) = g(s^i)$.

$$h(c) = s^i \tag{4.3}$$

Este método codificación y decodificación permite a CFA1 comprimir y descomprimir respectivamente sin pérdidas, pues que la función de codificación se aplica sobre texto en ASCII y la de decodificación únicamente decodifica caracteres de ASCII Extendido. Por otro lado, esta forma posibilita dividir un archivo FASTA en chunks para codificar y decodificar cada chunk por separado sin afectar el resultado final, lo que permite a CFA1 ser un algoritmo de compresión/descompresión paralelizable.

En un mejor caso se puede suponer que se tiene un archivo FASTA F compuesto por n contigs de largo k y m etiquetas de largo l, siendo k un número divisible por 3 y además los caracteres

de los contigs están compuestos únicamente por el lenguaje L. Inicialmente $|L| = |n| \cdot k + |m| \cdot l$, luego podemos tomar cada contig y transformarlo en una nueva cadena de texto componiendo subcadenas de largo 3 del contig y codificándolas con la función de la Ecuación 4.2. De esta manera, los n contigs de largo k se convierten en n nuevas cadenas de texto de largo $\frac{k}{3}$, permitiendo que el tamaño del nuevo archivo comprimido F' sea de cardinalidad $|F'| = |n| \cdot \frac{k}{3} + |m| \cdot l$ que es menor a la cardinalidad $|F|$. Dado que n y k son muy grandes con respecto a las etiquetas, se puede alcanzar teóricamente una razón de compresión cercana a 3.

Luego, se puede inferir que lo mejor sería tomar subcadenas de un mayor largo para lograr una razón de compresión más alta. Sin embargo, a través de este método no es posible ya que el código ASCII Extendido está conformado por solo 128 caracteres. También se puede creer que utilizando los 128 caracteres restantes del código ASCII (256 caracteres en total) se puede tener una razón de compresión cercana a 4 (con $|s^i| = 4$). No obstante, esto tampoco se puede lograr a la práctica debido a que las etiquetas utilizan esta parte de los caracteres de ASCII y las funciones de codificación y decodificación no podrían diferenciar entre etiquetas y contigs pues se estaría utilizando todos los caracteres existentes, algo que también impide definir caracteres a modo de separadores.

4.2.1. Algoritmo de Compresión de CFA1

El algoritmo de compresión CFA1 se divide en componentes. El primero se encarga de comunicar el kernel en GPU con el gestor de compresión (Algoritmo 6) y el segundo es el kernel de procesamiento (Algoritmo 7).

En el Algoritmo 6 se define el número de hebras y bloques, en este caso el kernel computa 2048 hebras simultáneamente, las cuales están repartidas en 256 bloques. En cada hebra se comprime un chunk de la partición S, donde el tamaño del chunk es el largo de la partición S dividido por el número de hebras. Como resultado se obtiene la secuencia codificada, C, la cual se copia al host para continuar con el proceso del Gestor, tal como aparece en la Sección 3.1.1.

Algorithm 6 Algoritmo de compresión CFA1

Function CompressCFA1(S_{device})
Input Bytes de la partición leída S en la GPU S_{device}
Output Bytes de la partición comprimida C (t.q. length(C) $\leq$ length(S_{device}))

1: $threads \leftarrow 2048$
2: $chunkSize \leftarrow \text{length}(S)/threads$
3: $gridDim \leftarrow \text{dim3}(256,1,1)$
4: $blockDim \leftarrow \text{dim3}(threads/256,1,1)$
5: $C_{device} \leftarrow \varnothing$
6: $args \leftarrow \{threads, chunkSize, S_{device}, C_{device}\}$
7: cudaLaunchKernel(CFA1compKernel, $gridDim, blockDim, args$) ▷ Kernel call
8: cudaDeviceSynchronize()
9: cudaMemcpy(C, C_{device}, length(C_{device}), cudaMemcpyDeviceToHost)
10: cudaFree(C_{device})

En cada hebra del kernel (Algoritmo 7) se construye un arreglo que permite distinguir los caracteres del lenguaje $L = \{A, C, G, T, \backslash n\}$ que se encuentran en la partición S y que de existir 3 de ellos consecutivos en S, es posible codificarlos a través de la función de codificación de la Ecuación 4.2. Para ello utiliza la interpretación numérica del carácter en el código ASCII. Los caracteres de S que no permitan generar estas subcadenas de largo 3 codificables, se agregan sin modificación en la salida.

Algorithm 7 Kernel de CFA1 para la compresión

Function CFA1compKernel(*threads*, *chunkSize*, S_{device}, C_{device})

```
1: id ← blockIdx.x * blockDim.x + threadIdx.x
2: it ← id * chunkSize
3: L ← ∅
4: for i ← 0 to 127 do
5:     L[i] ← −1
6: end for
7: L[65] ← 0                          ▷ Set {A,C,G,T,\n} with an identifier other than -1
8: L[67] ← 1
9: L[71] ← 2
10: L[84] ← 3
11: L[10] ← 4
12: k ← (id + 1) * chunkSize
13: if id = threads − 1 then
14:     k ← length(S_device)
15: end if
16: for i ← id * chunkSize to k − 1 do
17:     if i + 2 < k then
18:         c1 ← L[S_device[i]]
19:         c2 ← L[S_device[i + 1]]
20:         c3 ← L[S_device[i + 2]]
21:         if (c1 ≠ −1) & (c2 ≠ −1) & (c3 ≠ −1) then
22:             C_device[it] ← (25 * c1) + (5 * c2) + c3 + 128          ▷ Encode
23:             i ← i + 2                ▷ At the end of the cycle it adds 1 again
24:         else
25:             C_device[it] ← S_device[i]
26:         end if
27:     else
28:         C_device[it] ← S_device[i]
29:     end if
30:     it ← it + 1
31: end for
```

4.2.2. Algoritmo de Descompresión de CFA1

De forma análoga a la Sección 4.2.1 anterior, el algoritmo de descompresión de CFA1 se divide en dos partes.

En el Algoritmo 8 se define el número de hebras, bloques y tamaño del chunk tal como sucede en el Algoritmo 6. Sin embargo, aquí se diferencia entre los chunks de la partición comprimida C y los chunks descomprimidos de S, pues se espera que estos últimos sean a lo más 3 veces

más grandes que los chunks de C.

Algorithm 8 Algoritmo de descompresión CFA1

Function DecompressCFA1(C_{device})
Input Bytes de la partición comprimida C en la GPU C_{device}
Output Bytes de la partición descomprimida S (t.q. length(C_{device}) $\leq$ length(S))

1: $threads \leftarrow 2048$
2: $chunkSizeIn \leftarrow \text{length}(C_{device})/threads$
3: $chunkSizeOut \leftarrow 3 * chunkSizeIn$
4: $gridDim \leftarrow \text{dim3}(256,1,1)$
5: $blockDim \leftarrow \text{dim3}(threads/256,1,1)$
6: $S_{device} \leftarrow \varnothing$
7: $args \leftarrow \{threads, chunkSizeIn, chunkSizeOut, C_{device}, S_{device}\}$
8: cudaLaunchKernel(CFA1decompKernel, $gridDim, blockDim, args$) ▷ Kernel call
9: cudaDeviceSynchronize()
10: cudaFree(C_{device})
11: cudaMemcpy(S, S_{device}, length(S_{device}), cudaMemcpyDeviceToHost)
12: cudaFree(S_{device})

El kernel (Algoritmo 9) utiliza la función de la Ecuación 4.3 para decodificar cada chunk de la partición C. Para ello, en cada hebra se construye un arreglo con las subcadenas de largo 3 que se presentan en la Ecuación 4.1 y que utiliza dicha función, para posteriormente aplicar la función a los bytes de cada chunk de C y eventualmente encontrar uno que pueda ser decodificado agregando los bytes de la subcadena correspondiente a un chunk de S.

Algorithm 9 Kernel de CFA1 para la descompresión

Function CFA1decompKernel(*threads*, *chunkSizeIn*, *chunkSizeOut*, C_{device}, S_{device})

```
1: id ← blockIdx.x * blockDim.x + threadIdx.x
2: it ← id * chunkSizeOut
3: str ← "ACGT\n"
4: l ← 0
5: h ← ∅                                          ▷ Create function h() (Ecuación 4.3)
6: for i ← 0 to 4 do
7:     for j ← 0 to 4 do
8:         for k ← 0 to 4 do
9:             h[l] ← str[i]
10:            h[l + 1] ← str[j]
11:            h[l + 2] ← str[k]
12:            l ← l + 3
13:        end for
14:    end for
15: end for
16: k ← (id + 1) * chunkSizeIn
17: if id = threads − 1 then
18:    k ← length(C_device)
19: end if
20: for i ← id * chunkSizeIn to k − 1 do
21:    if C_device[i] < 128 then
22:        S_device[it] ← C_device[i]
23:        it ← it + 1
24:    else
25:        key ← 3 * (C_device[i] − 128)                              ▷ Decode
26:        S_device[it] ← h[key]
27:        S_device[it + 1] ← h[key + 1]
28:        S_device[it + 2] ← h[key + 2]
29:        it ← it + 3
30:    end if
31: end for
```

4.2.3. Resultados de la Evaluación Experimental de CFA1

En la Tabla 4.1 se muestra el desempeño de CFA1 con la mejor configuración encontrada. Aquí se revela que la razón de compresión alcanzada por CFA1 es mayor a 2.6 con todos los datasets de prueba, lo que es mayor a lo obtenido con las API de LZ4 (Algoritmo 3.3.1), Snappy (Algoritmo 3.3.2), Bitcomp (Algoritmo 3.3.7) y Deflate (Algoritmo 3.3.4) de nvCOMP. En particular, con el archivo CAMI_high se puede ratificar la razón de compresión teórica que puede llegar a ser muy cercana a 3, tal como se presenta en la hipótesis teórica al inicio de esta sección.

Con respecto al tiempo de compresión este logra ser 80 veces más rápido que GZIP y 4 veces más rápido en la descompresión.

Tabla 4.1: Configuración de referencia de CFA1

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Max. GPU MUD (GB)	Comp. Speedup	Decomp. Speedup
1/4	512	CAMI_high	2.998	998	669	1.1	0.8	90.1	4.1
1/4	512	case2_assembly	2.662	948	691	1.1	0.8	76.7	3.7
1/4	512	nielsen	2.634	957	708	1.1	0.8	81.6	3.7
1/4	512	case3_assembly	2.851	1007	723	1.1	0.8	86.9	3.8

4.3. CFA2

Dado que con CFA1 se logra una razón de compresión y un tiempo de compresión/descompresión competitivos respecto a las otras API de nvCOMP, con CFA2, el segundo compresor de archivos FASTA, se tiene como objetivo aumentar la razón de compresión respecto a CFA1 utilizando un compresor adicional perteneciente a nvCOMP. Este objetivo tiene la limitación de que los compresores de nvCOMP y en general, no permiten comprimir caracteres que pertenezcan a ASCII Extendido.

CFA2, continúa el camino de CFA1 de la Sección 4.2, pues utiliza las mismas funciones de codificación (Ecuación 4.2) y decodificación (Ecuación 4.3). Pero en este caso utiliza un lenguaje $L' = \{A, C, G, T\}$ del cual se construye un conjunto C' compuesto por subcadenas s^i de largo 3, las cuales conforman a todas las combinaciones que se pueden generar de L', siendo $L' \subset L$ y $C' \subset C$. La cantidad de combinaciones en este caso corresponde a la cardinalidad $|C'| = |L'|^{|s^i|} = 4^3 = 64$, que es menor a la vista en la Sección 4.2.

$$C' = \{s^1\{A, A, A\},\ s^2\{A, A, C\},\ s^3\{A, A, G\},\ s^4\{A, A, T\},\ s^5\{A, G, A\},\ ..\ ,\ s^{64}\{T, T, T\}\} \tag{4.4}$$

Escoger un conjunto más pequeño de combinaciones tiene tres motivos que están relacionados:

- El primero es que con ello es posible utilizar la parte del código ASCII (no de ASCII Extendido) que no contiene al alfabeto para poder codificar las subcadenas de C' (Ecuación

4.4) a estos caracteres.

- El segundo motivo es que en un primer paso de la compresión de CFA2, este aplica la función de la Ecuación 4.2 únicamente en los contigs del archivo FASTA para poder obtener de esta manera un archivo comprimido parcial. El contenido del archivo comprimido parcial está completamente dentro del código ASCII y, por lo tanto, es posible utilizarlo en un segundo compresor en el paso siguiente.
- El último motivo es que los contigs que poseen los archivos FASTA no siempre contienen únicamente a los caracteres de L', sino que puede contener otros caracteres del alfabeto, por lo tanto, estos no pueden ser utilizados para ser asociados a las subcadenas de C'.

4.3.1. Algoritmo de Compresión de CFA2

El Algoritmo 10 realiza la llamada a un primer kernel que corresponde justamente al que aplica la función de codificación de la Ecuación 4.2, utilizando las subcadenas de C'. Este comprime la partición S generando una nueva partición comprimida parcial que será entregada al siguiente algoritmo de compresión (Algoritmo 3) que utiliza las API de nvCOMP, el cual generará la partición comprimida final que será retornada al gestor de compresión.

Algorithm 10 Algoritmo de compresión CFA2

Function CompressCFA2(S_{device})
Input Bytes de la partición leída S en la GPU S_{device}
Output Bytes de la partición comprimida C (t.q. length(C) $\leq$ length(S_{device}))

1: $threads \leftarrow 2048$
2: $chunkSize \leftarrow \text{length}(S)/threads$
3: $gridDim \leftarrow \text{dim3}(256,1,1)$
4: $blockDim \leftarrow \text{dim3}(threads/256,1,1)$
5: $C'_{device} \leftarrow \varnothing$
6: $args \leftarrow \{threads, chunkSize, S_{device}, C'_{device}\}$
7: cudaLaunchKernel(CFA2compKernel, $gridDim, blockDim, args$) ▷ Call CFA2 kernel
8: cudaDeviceSynchronize()
9: cudaFree(S_{device})
10: $alg \leftarrow$ *algorithm with which you get the best ratio*
11: $C \leftarrow$ Compress(C'_{device}, alg) ▷ Call nvCOMP (kernel)

En cada hebra del kernel (Algoritmo 11) se construye un arreglo que permite distinguir los caracteres del lenguaje L' que se encuentran en la partición S y, que de existir 3 de ellos

consecutivos en S, es posible codificarlos con Ecuación 4.2 que también se construye en cada hebra en forma de arreglo. Los caracteres de S que no permitan generar estas subcadenas de largo 3 codificables se agregan a la salida.

Algorithm 11 Kernel de CFA2 para la compresión

Function CFA2compKernel($threads$, $chunkSize$, S_{device}, C'_{device})

$id \leftarrow$ blockIdx.x * blockDim.x + threadIdx.x
$it \leftarrow id * chunkSize$
$L' \leftarrow \varnothing$
for $i \leftarrow 0$ to 127 **do**
    $L'[i] \leftarrow -1$
end for
$L'[65] \leftarrow 0$ ▷ Set {A,C,G,T} with an identifier other than -1
$L'[67] \leftarrow 1$
$L'[71] \leftarrow 2$
$L'[84] \leftarrow 3$
$g \leftarrow$ generateEncodingArray() ▷ Encoding function
$k \leftarrow (id + 1) * chunkSize$
if $id = threads - 1$ **then**
    $k \leftarrow$ length(S_{device})
end if
for $i \leftarrow id * chunkSize$ to $k - 1$ **do**
    if $i + 2 < k$ **then**
        $c_1 \leftarrow L'[S_{device}[i]]$
        $c_2 \leftarrow L'[S_{device}[i + 1]]$
        $c_3 \leftarrow L'[S_{device}[i + 2]]$
        if $(c_1 \neq -1)$ & $(c_2 \neq -1)$ & $(c_3 \neq -1)$ **then**
            $C_{device}[it] \leftarrow g[(16 * c_1) + (4 * c_2) + c_3]$ ▷ Encode
            $i \leftarrow i + 2$ ▷ At the end of the cycle it adds 1 again
        else
            $C_{device}[it] \leftarrow S_{device}[i]$
        end if
    else
        $C_{device}[it] \leftarrow S_{device}[i]$
    end if
    $it \leftarrow it + 1$
end for

El procedimiento de compresión de CFA2 se muestra de forma resumida en el esquema de la Figura 4.1.

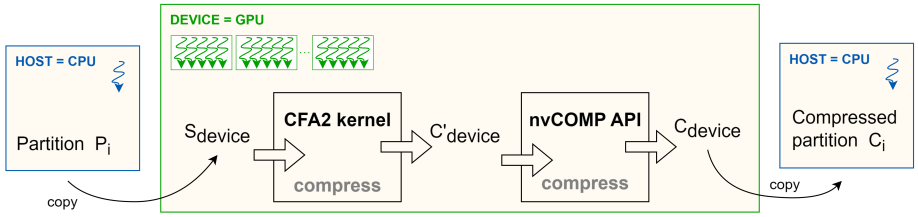


Figura 4.1: Esquema de compresión de CFA2

4.3.2. Algoritmo de Descompresión de CFA2

El Algoritmo 12 llama primero a la función de descompresión de nvCOMP para descomprimir parcialmente una partición del archivo FASTA, la cual se pasa al kernel de CFA2 que se encarga de decodificar dicha partición. El resultado obtenido se devuelve al Gestor de compresión.

Algorithm 12 Algoritmo de descompresión CFA2

Function DecompressCFA2(C_{device}, alg)
Input Bytes de la partición comprimida C_{device} y algoritmo de compresión alg de nvCOMP
Output Bytes de la partición descomprimida S (t.q. length(C_{device}) $\leq$ length(S))

1: $C'_{device} \leftarrow$ Decompress(C_{device}, alg) ▷ Call nvCOMP (kernel)
2: $threads \leftarrow 2048$
3: $chunkSizeIn \leftarrow$ length(C'_{device})$/threads$
4: $chunkSizeOut \leftarrow 3 * chunkSizeIn$
5: $gridDim \leftarrow$ dim3(256,1,1)
6: $blockDim \leftarrow$ dim3($threads$/256,1,1)
7: $S_{device} \leftarrow \varnothing$
8: $args \leftarrow \{threads, chunkSizeIn, chunkSizeOut, C'_{device}, S_{device}\}$
9: cudaLaunchKernel(CFA2decompKernel, $gridDim, blockDim, args$) ▷ Call CFA2 kernel
10: cudaDeviceSynchronize()
11: cudaFree(C'_{device})
12: cudaMemcpy(S, S_{device}, length(S_{device}), cudaMemcpyDeviceToHost)
13: cudaFree(S_{device})

En cuanto al kernel (Algoritmo 13), este en cada hebra construye un arreglo que permite distinguir los caracteres utilizados en la codificación y un segundo arreglo que representa a la propia función de decodificación. Posteriormente procesa los caracteres del chunk utilizando un mecanismo que le permite distinguir entre aquellos caracteres que forman parte de un contig y los que no, pues solo decodificará aquellos si pertenezcan a un contig.

Algorithm 13 Kernel de CFA2 para la descompresión

Function CFA2decompKernel($threads$, $chunkSizeIn$, $chunkSizeOut$, C'_{device}, S_{device})

1: $id \leftarrow$ blockIdx.x * blockDim.x + threadIdx.x
2: $it \leftarrow id * chunkSizeOut$
3: $conv \leftarrow$ generateConvArray() ▷ According to the ASCII characters used
4: $h \leftarrow$ generateDecodingArray ▷ Decoding function (similar to Algorithm 9)
5: $k \leftarrow (id + 1) * chunkSizeIn$
6: **if** $id = threads - 1$ **then**
7: $k \leftarrow$ length(C'_{device})
8: **end if**
9: **for** $i \leftarrow id * chunkSizeIn$ to $k - 1$ **do**
10: **if** IsCharOfContig($C_{device}[i]$) $\neq$ true **then**
11: $S_{device}[it] \leftarrow C'_{device}[i]$
12: $it \leftarrow it + 1$
13: **else**
14: $key \leftarrow conv[C'_{device}[i]]$
15: $S_{device}[it] \leftarrow h[3 * key]$ ▷ Decode
16: $S_{device}[it + 1] \leftarrow h[(3 * key) + 1]$
17: $S_{device}[it + 2] \leftarrow h[(3 * key) + 2]$
18: $it \leftarrow it + 3$
19: **end if**
20: **end for**

El procedimiento de descompresión de CFA2 se muestra de forma resumida en el esquema de la Figura 4.2.

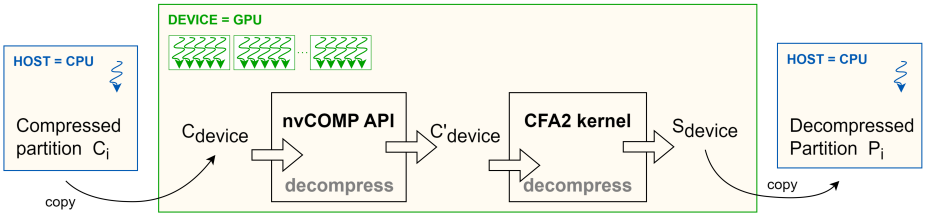


Figura 4.2: Esquema de descompresión de CFA2

Dado que los chunks pueden cortar a la partición en cualquier punto, en la implementación, el mecanismo que se utiliza para distinguir los caracteres que pertenecen a un contig es el siguiente:

- En cada hebra se comienza a leer el chunk de la partición en reversa hasta encontrar el inicio o un carácter de salto de línea.
- Luego lee un siguiente carácter y verifica si corresponde o no al inicio de una etiqueta.

- Al identificar si es o no una etiqueta procede a ajustarse para comenzar a decodificar línea a línea solo aquellas que sean contigs.

Esto es posible ya que tanto el carácter de salto de línea como el carácter que indica el inicio de una etiqueta son caracteres reservados y, por lo tanto, no forman parte del conjunto de caracteres utilizados para la codificación.

4.3.3. Resultados de la Evaluación Experimental de CFA2

Para cumplir con el objetivo planteado en un inicio se realiza una evaluación preliminar, donde se incorporan las API de nvCOMP dentro de CFA2 y se obtiene la razón de compresión que genera cada una de ellas con los datasets de prueba. Este resultado se muestra en la Tabla 4.2, donde la mejor razón de compresión se obtiene utilizando Zstandard (Zstd).

Tabla 4.2: Comparativa de razón de compresión de CFA2 con diferentes API de nvCOMP

nvCOMP API	CAMI_high	case2_assembly	nielsen	case3_assembly	Average
ZSTD	4.124	3.953	4.071	3.974	4.030
LZ4	3.003	2.948	3.029	2.976	2.989
Snappy	2.989	2.928	3.000	2.957	2.968
Gdeflate	4.108	3.399	3.428	3.709	3.661
Deflate	4.131	3.830	3.993	3.942	3.974
Cascaded	2.997	2.629	2.622	2.843	2.773
Bitcomp	3.656	3.131	3.135	3.393	3.329
ANS	4.083	3.385	3.412	3.688	3.642

Esta razón de compresión lograda en CFA2 con Zstd es de 4.030 en promedio y es la más alta que se logró obtener de todas las evaluaciones incluido el capítulo 3, siendo la segunda más alta la obtenida con GZIP con un valor de 3.448 y la tercera la obtenida con ANS de nvCOMP (Sección 3.3.8) con un valor de 3.388.

Tabla 4.3: Configuración de referencia de CFA2

CPU TF R/W	Approx. Partition Size (MB)	FASTA File Name	Ratio	Comp. Throughput (MB/s)	Decomp. Throughput (MB/s)	Max. GPU MUC (GB)	Max. GPU MUD (GB)	Comp. Speedup	Decomp. Speedup
1/4	768	CAMI_high	4.122	764	620	1.6	1.9	69.0	3.8
1/4	768	case2_assembly	3.952	736	655	1.6	1.9	59.6	3.5
1/4	768	nielsen	4.071	756	697	1.6	1.9	64.5	3.6
1/4	768	case3_assembly	3.974	763	699	1.6	1.9	65.9	3.7

En cuanto al desempeño de CFA2 este se obtuvo utilizando Zstd en la evaluación experimental,

donde los resultados con la configuración de referencia o la que produce el mejor desempeño se presentan en la Tabla 4.3. Aquí se puede observar que CFA2 tiene un uso de memoria de GPU similar a CF1 pero CFA1 es más rápido tanto en comprimir como en descomprimir, lo que tiene sentido ya que CFA2 realiza una compresión/descompresión adicional. Sin embargo, este valor del throughput es competitivo con otros compresores disponibles en nvCOMP.

Capítulo 5

Análisis Comparativo

En este capítulo se sintetizan y complementan los resultados obtenidos en los capítulos 3 y 4, en los cuales se presenta una revisión y una evaluación experimental individual de los algoritmos de compresión/descompresión utilizados con los datasets de prueba de archivos FASTA mostrado en la Tabla 3.1. El objetivo es comparar el desempeño de cada uno de estos algoritmos, tanto en la compresión como en la descompresión.

Para ello se presentan gráficos que evidencian las magnitudes más importantes de esta comparativa. Particularmente, se presenta el *throughput* o tasa de transferencia efectiva a la que se comprime/descomprime un archivo FASTA; el *speedup* o aceleración de los algoritmos de compresión/descompresión acelerados por GPU con respecto al algoritmo de referencia GZIP; el uso máximo de memoria de GPU dedicada utilizada por estos algoritmos; y el *ratio* o razón de compresión que pueden llegar a conseguir. Siempre considerando la configuración de referencia de cada algoritmo.

En la Figura 5.1 se observa el throughput conseguido por cada algoritmo con los diferentes datasets. Aquí nos podemos percatar fácilmente que pese a que los datasets son de distinto tamaño (Tabla 3.1), el throughput posee un comportamiento lineal en el desempeño de los algoritmos para diferentes tamaños de archivos FASTA. Además, es notorio que todos los algoritmos acelerados por GPU tienen un throughput mayor con respecto a GZIP. En el gráfico superior se muestra el throughput obtenido en la compresión, donde el valor más alto lo consigue ANS, seguido de GDeflate. Según lo mostrado, es posible agrupar los algoritmos acelerados por

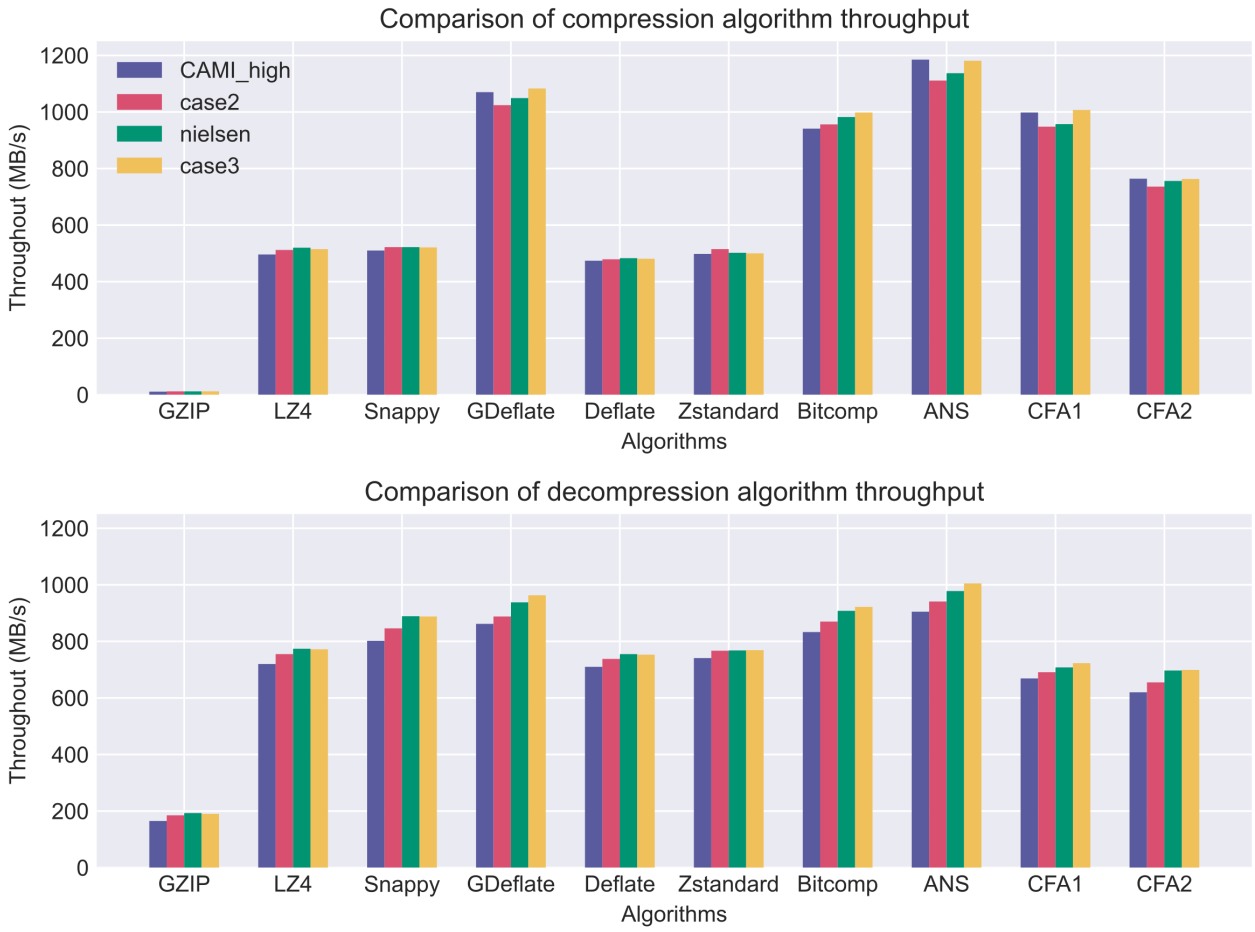


Figura 5.1: Throughput en la compresión y descompresión agrupado por algoritmo

GPU con un desempeño similar, formando así dos grupos. El primer grupo (1) se compone de LZ4, Snappy, Deflate y Zstandard, los cuales alcanzan un throughput de alrededor de 500 MB/s. El segundo (2) se compone por ANS primero, luego GDeflate, CFA1 y Bitcomp, los cuales alcanzan un throughput de alrededor de 1000 MB/s, el doble de lo que consigue el primer grupo. Luego, queda CFA2 cuyo throughput se encuentra a la mitad de estos dos grupos. En el gráfico inferior se muestra también el throughput de los algoritmos para la descompresión. Aquí el valor más bajo corresponde a la compresión con GZIP con alrededor de 200 MB/s, mientras que los algoritmos acelerados por GPU tienen un throughput que se encuentra entre los 600 MB/s y los 1000 MB/s, una diferencia menor entre sí con respecto a la compresión, lo que revela que todos pertenecen a un grupo en común.

Los mismos resultados de la Figura 5.1 se muestran en la Figura 5.2, pero aquí los resultados

se agrupan por dataset. En el gráfico de la izquierda se ven más claramente los grupos (1) y (2) mencionados anteriormente, los cuales se ubican en los cuartiles Q1 y Q3 respectivamente. También, en el gráfico de la derecha, es más notorio que los algoritmos de descompresión acelerados por GPU pertenecen a un grupo en común pues mantienen un throughput muy similar entre ellos.

3.1)

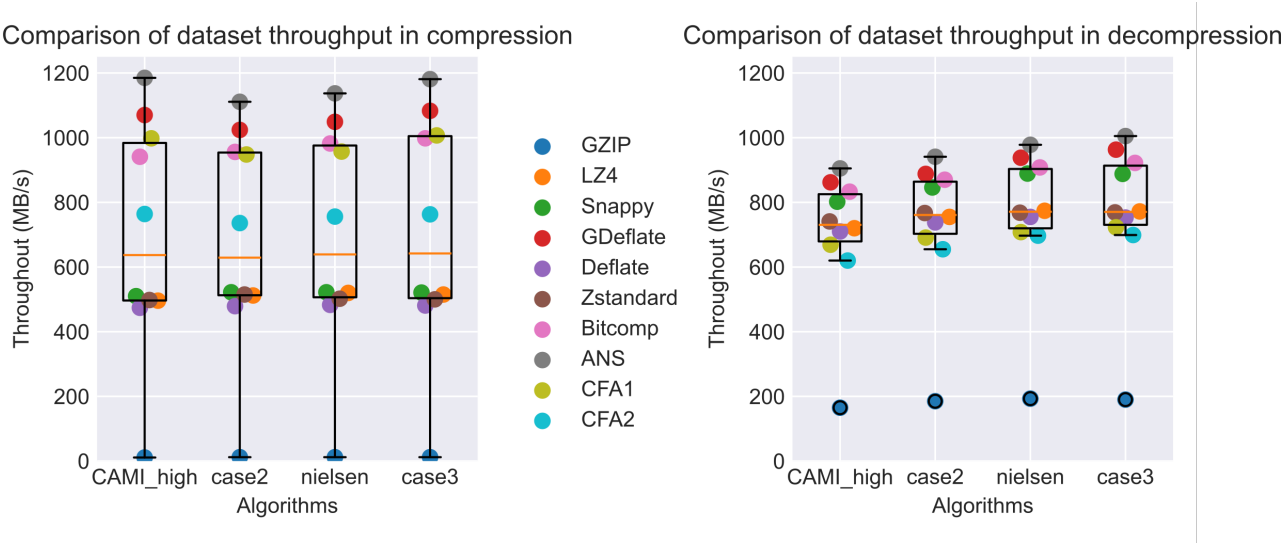


Figura 5.2: Throughput en la compresión y descompresión agrupado por dataset

En la Figura 5.3 se expone el speedup logrado por los algoritmos acelerados por GPU respecto a GZIP, donde en todos existe un incremento en cuanto a la velocidad de compresión y descompresión. Luego, esto se traduce a un menor tiempo de cómputo para obtener el mismo resultado. En el gráfico inferior se puede ver que la velocidad alcanzada en la descompresión es de 3 a 5 veces más rápida con respecto a la obtenida con GZIP. Mientras que, en el gráfico superior, se puede ver un aumento más significativo logrando una aceleración en la compresión de hasta 100 veces con ANS, siendo la menor la obtenida con Deflate, la cual es de 40x.

El uso máximo de memoria de GPU utilizado por los algoritmos acelerados por GPU se presenta en la Figura 5.4, donde se comparan utilizando tamaños de partición de 128MB, 256 MB y 512 MB. En ambos gráficos es posible observar que todos los algoritmos consumen una cantidad de memoria de GPU que aumenta a medida que crece el tamaño de la partición a comprimir/descomprimir. En la figura superior se tiene que el consumo más alto es producido por

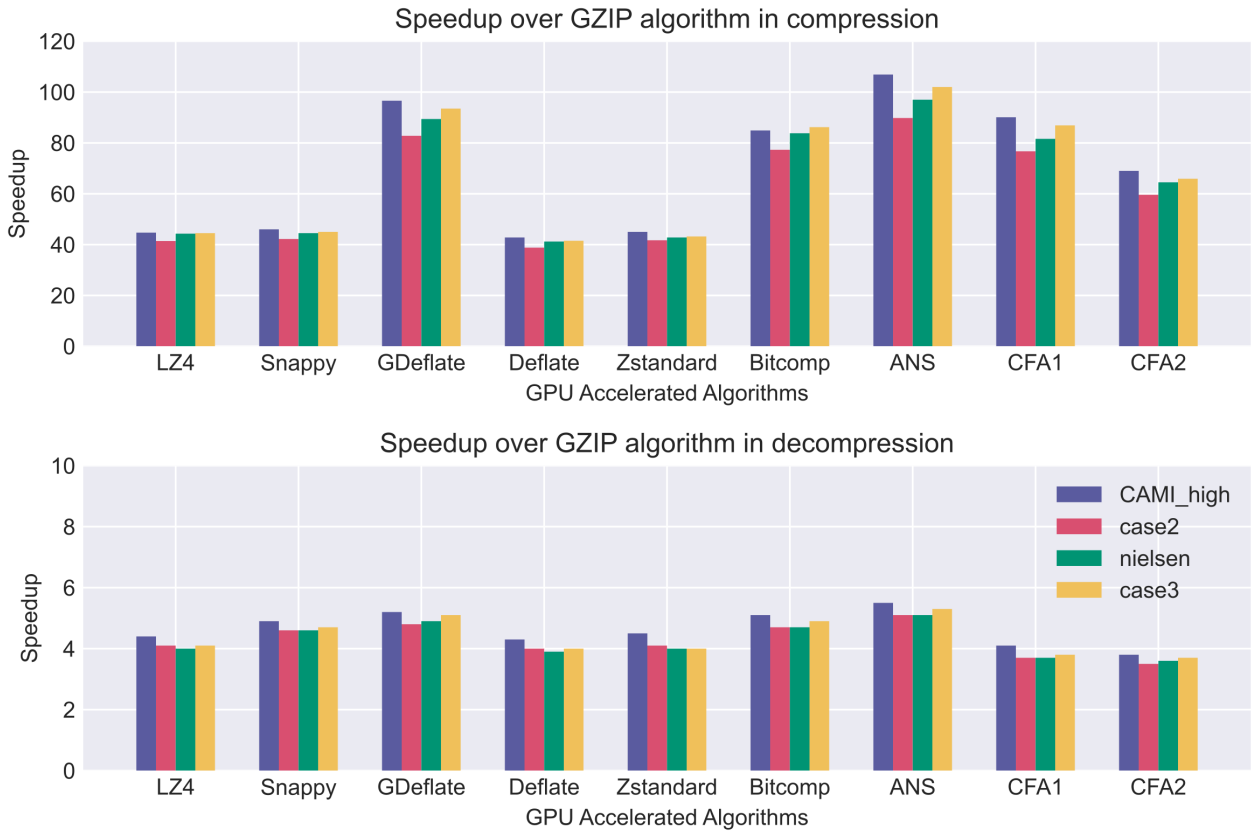


Figura 5.3: Speedup en la compresión y descompresión con respecto a GZIP

Deflate y el menor el producido por CFA1 en la compresión. Mientras que, en la descompresión, el consumo máximo de memoria de GPU es bastante similar entre algoritmos.

En el gráfico de la Figura 5.5 se muestra la razón de compresión promedio alcanzada por los algoritmos acelerados por GPU y el máximo consumo de memoria de GPU utilizado por los mismos para alcanzar ese valor. Tal valor es un promedio entre el utilizado en la compresión y el utilizado en la descompresión para un tamaño de partición de 256 MB. Aquí es interesante resaltar que CFA1 es el que menos memoria de GPU usa y produce una razón de compresión que está en la media de los algoritmos presentados. Por otro lado, Deflate es el que usa mayor memoria, y esta por detrás de CFA1 en cuanto a razón de compresión. Esto revela que no es necesario utilizar más memoria para obtener una mejor razón de compresión en archivos FASTA. Adicionalmente se destaca a CFA2, pues siendo el que mejor razón de compresión produce esta solo un paso por delante de CFA1 en cuanto a consumo de memoria de GPU.

En la Figura 5.6 se presenta el promedio ponderado del throughput (Ecuación 3.5) comparado con

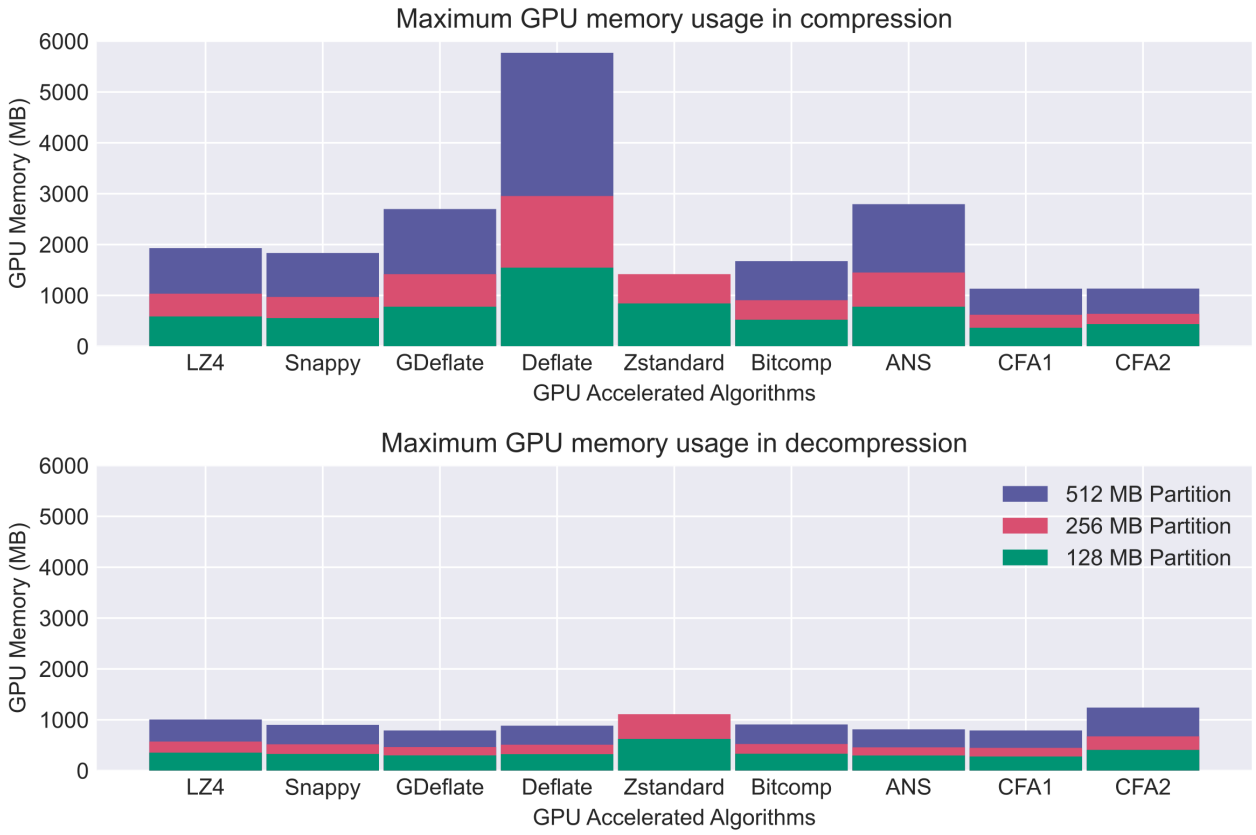


Figura 5.4: Máximo uso de memoria de GPU con distintos tamaños de partición

la razón de compresión promedio que llegan a obtener los algoritmos de compresión/descompresión. Aquí al igual que en la Figura 5.5 tenemos que una mejor razón de compresión no necesariamente sacrifica el desempeño obtenido del compresor. Esto se ve claramente con CFA2, que es mejor a GZIP tanto en razón de compresión como en el throughput alcanzado. También se destaca a GDeflate y ANS que son los más veloces y pese a esto mantienen una razón de compresión muy cercana a GZIP.

Por último, en el Anexo 1 y el Anexo 2 se muestra más a detalle del tiempo de compresión y descompresión de cada algoritmo versus el tamaño del archivo comprimido que generan, esto para cada dataset de prueba.

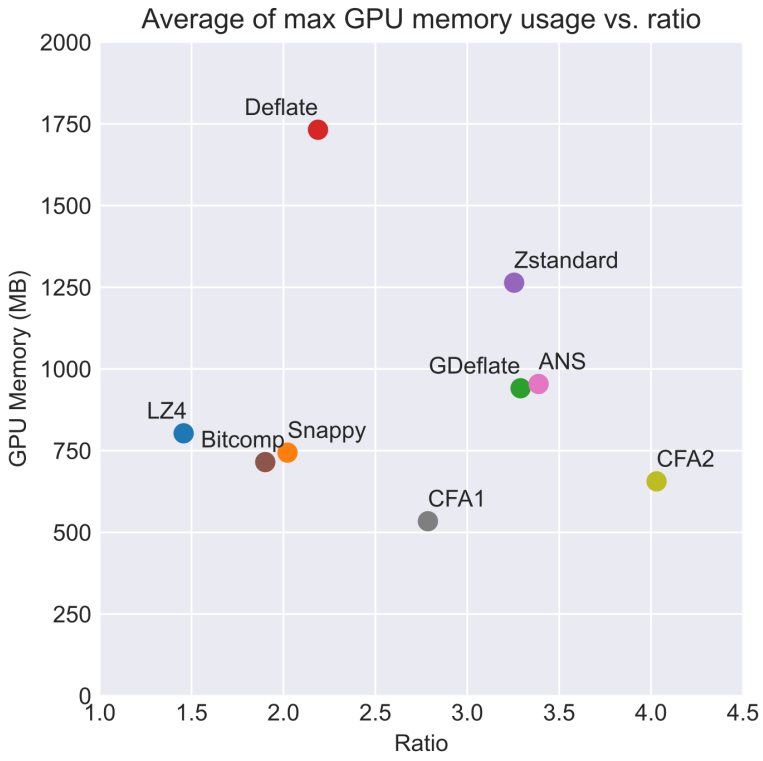


Figura 5.5: Promedio del máximo uso de memoria de GPU vs. razón de compresión

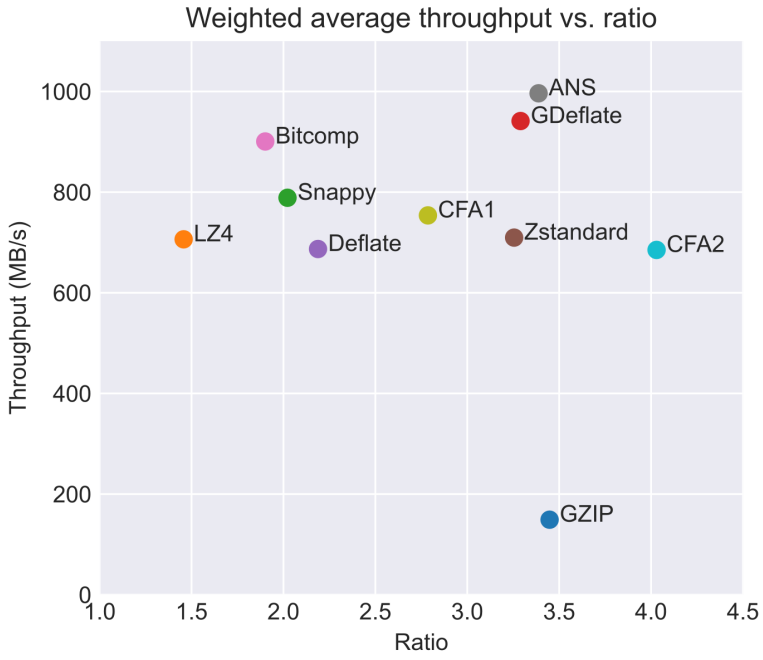


Figura 5.6: Promedio ponderado del throughput vs. razón de compresión

Capítulo 6

Integración con MetaBAT

Como se mencionaba en la Introducción, la edición de MetaBAT a utilizar para la integración corresponde a la desarrollada por *Jonathan Venegas*, la cual a su vez está basada en la versión de MetaBAT 2 publicada en el repositorio oficial de MetaBAT a mediados de diciembre de 2023[1].

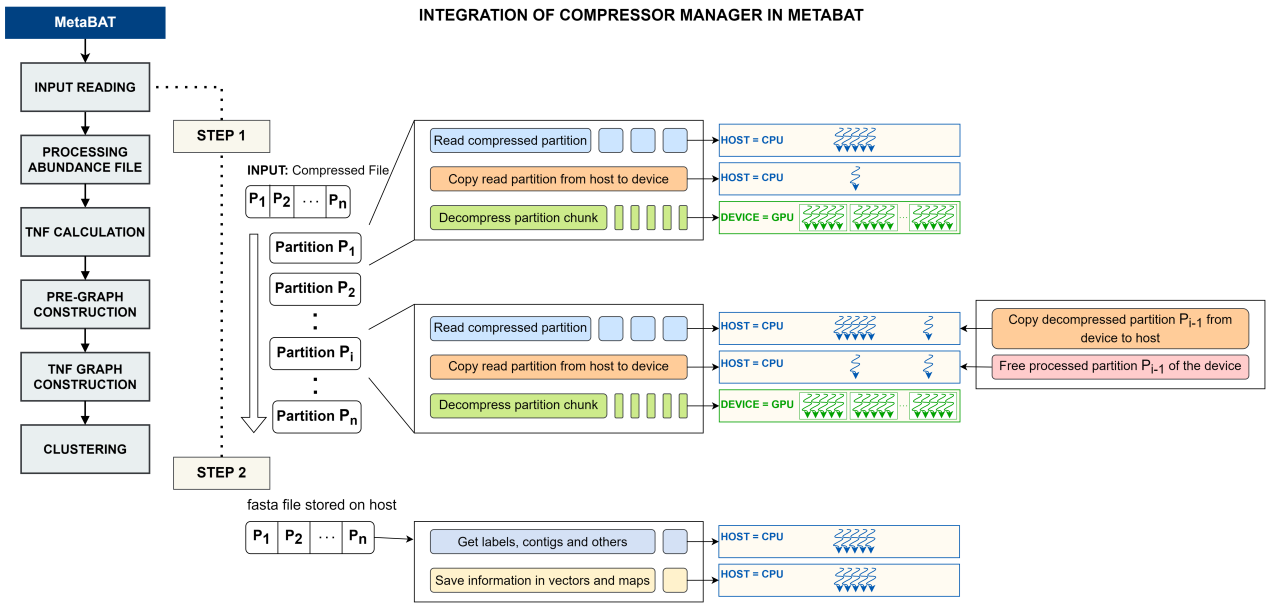


Figura 6.1: Integración del Gestor de compresión y descompresión en MetaBAT2

La integración de los algoritmos de compresión/descompresión presentados en el Capítulo 3 y el Capítulo 4 se realiza sobre la fase global de lectura del archivo de entrada, que está compuesta por dos etapas. La primera etapa es la lectura del archivo FASTA sin comprimir, la cual se

[1] https://bitbucket.org/berkeleylab/metabat/src/40efa2dc296404c7ab6735276a0a8dcd5da4e709/src/

realiza con múltiples hebras de CPU hasta obtener el archivo completo en memoria del host. La segunda etapa, que de igual manera se realiza utilizando paralelismo en CPU, almacena las etiquetas y contigs contenidos en vectores y mapas.

En este caso la integración, como se puede ver en la Figura 6.1 se realiza en la primera etapa, donde se reemplaza la lectura por una llamada al Gestor de compresión y descompresión visto en la Sección 3.1. El Gestor se transforma en una biblioteca más de MetaBAT, que en este caso, permite descomprimir el archivo sin escritura y en cambio, almacena cada una de las particiones en el host, obteniendo el mismo resultado que proveía la primera etapa anteriormente.

La evaluación de los algoritmos de compresión/descompresión termina en su integración con MetaBAT. En este caso, los resultados mostrados en la Figura 6.2 comparan el tiempo de ejecución en la lectura sin usar compresión (versión de *Jonathan*), usando Z-Lib y kseq (MetaBAT 2 sin modificar) y utilizando los compresores acelerados por GPU. Como se puede observar la versión con Z-Lib, es decir, la que descomprime los archivos comprimidos con GZIP, es la más lenta con todos los datasets. Seguido están las versiones que utilizan compresores acelerados por GPU, las cuales tienen un resultado relativamente similar entre sí, logrando un speedup de 2.5x en promedio con respecto a Z-Lib en todos los archivos. Por último, la versión sin compresión es la más veloz de todas, logrando un speedup de 5.5x en promedio con respecto a Z-Lib, pero perdiendo los beneficios propios de la compresión.

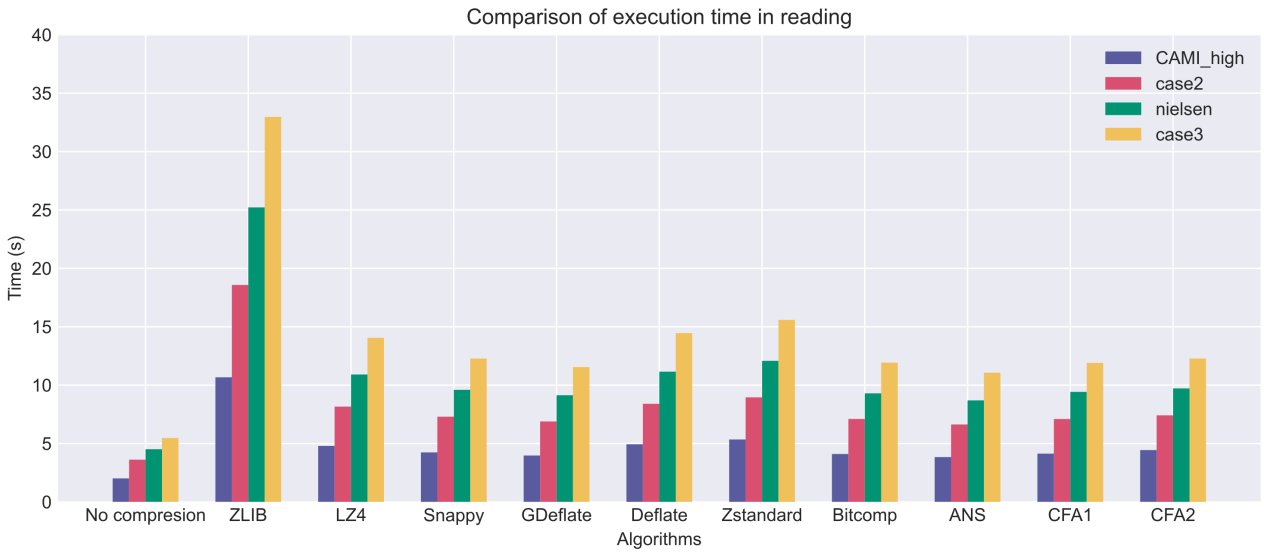


Figura 6.2: Comparación del tiempo de ejecución en la lectura

Capítulo 7

Etapa de clustering en MetaBAT

En este capítulo se pretende abordar el Objetivo Específico 4 el cual apunta a indagar y aplicar paralelismo en CPU y GPU en la etapa de clustering de MetaBAT2 para mejorar su desempeño.

7.1. Antecedentes

Dado un conjunto de datos no uniforme, el proceso de agrupar los datos del conjunto se llama agrupamiento o clasificación de datos y a los grupos resultantes se le denominan clusters [10]. Este proceso lo lleva a cabo un algoritmo que se encarga de agrupar el conjunto de datos utilizando una medida de similitud o distancia definida.

En MetaBAT2 el clustering se realiza en la fase final del algoritmo principal, donde la entrada corresponde al conjunto de contigs representados en un grafo dado en las etapas anteriores. En tal grafo cada vértice representa a un contig del dataset y existe una arista si dos contigs tienen una similitud mayor a un umbral la cual determina el peso de la arista. En el artículo de MetaBAT [4, pág. 6] se señala que esta etapa es realizada por un algoritmo de clustering k-medoid modificado, sin embargo tanto en la implementación[1] de MetaBAT1 como MetaBAT2 se puede encontrar que el algoritmo de clustering que se encarga de esta etapa es una variante de Label Propagation.

Label Propagation es un tipo de algoritmo de aprendizaje automático que realiza la agrupación

[1] https://bitbucket.org/berkeleylab/metabat/src/40efa2dc296404c7ab6735276a0a8dcd5da4e709/src/

de vértices de un grafo con aristas por medio del uso de reasignación de etiquetas asociadas a los vértices. Para ello en un inicio cada vértice es su propia etiqueta y en las etapas posteriores se realiza la propagación de dichas etiquetas entre los vecinos de cada vértice utilizando un criterio definido, lo que resulta finalmente en un grafo etiquetado, cuyos vértices que compartan la misma etiqueta forman un cluster.

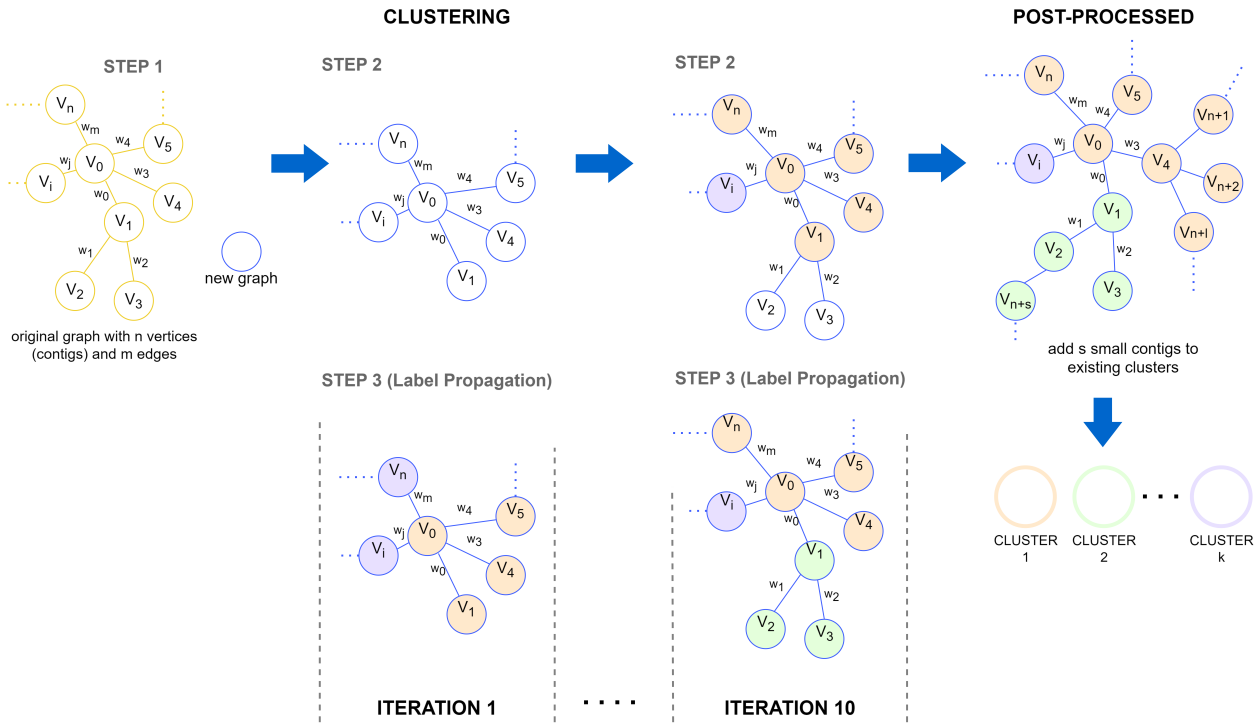


Figura 7.1: Etapa de clustering y postprocesado en MetaBAT2

La etapa de clustering y el postprocesado en MetaBAT2 se muestra en la Figura 7.1 y funciona de la siguiente manera:

1. Se crea un nuevo grafo y se etiquetan los vértices del grafo original (el que se ha formado de etapas anteriores).

2. Se reúne un grupo de aristas no visitadas del grafo original cuyos vértices no comparten la misma etiqueta y se agregan dichos vértices al nuevo grafo.

3. Se ejecuta el algoritmo de label propagation utilizando el nuevo grafo, el cual reasigna la etiqueta de cada vértice que se ha agregado hasta el momento. La reasignación consiste en puntuar a los vecinos de un vértice de acuerdo al peso de cada arista que forma con ellos,

para luego con la ayuda de la implementación de Chi-cuadrado de la biblioteca Boost[2] del Método de Fisher determinar al vecino mejor puntuado. Luego, la etiqueta del vecino mejor puntuado es reasignada a dicho vértice.

4. Se repiten los pasos 2-3 en 10 iteraciones. En la última iteración se completa la revisión de las aristas del grafo original y con ello termina el etiquetado de los vértices (Fin de la etapa de clustering). A continuación, se realiza un postprocesado que consiste en un clustering con los considerados «small contigs», que serán añadidos a los clusters encontrados en la etapa de clustering. Por último, se lleva a cabo la construcción de cada cluster de acuerdo a las etiquetas asignadas.

El desempeño de la etapa de clustering de MetaBAT2, en la cual realiza 10 iteraciones de label propagation, se puede ver en la Tabla 7.1.

Tabla 7.1: Etapa de Clustering en MetaBAT2 utilizando los datasets de prueba

FASTA File Name	Decompressed Size (MB)	Vertices	Edges	Clustering Time (s)
CAMI_high	2673.77	30,841	1,722,890	6.059
case2_assembly	4509.97	230,079	16,867,970	164.786
nielsen	6472.12	449,706	23,484,116	498.981
case3_assembly	8363.13	733,876	52,770,200	900.815

Cabe señalar que el algoritmo de clustering de MetaBAT2 es iterativo y todas sus etapas dependen del resultado anterior, por este motivo y pese a los esfuerzos, no fue posible realizar un algoritmo paralelizado que entregue el mismo resultado basándose en él. Debido a esto se realiza una búsqueda de algoritmos alternativos de label propagation acelerados por CPU y GPU que mejoren el rendimiento a la vez que conserven la mayor similitud posible en los clusters resultantes.

7.2. Revisión estado del arte

Durante la revisión del estado del arte se pudo encontrar que en distintas aplicaciones se ha utilizado un algoritmo de label propagation acelerado por CPU y GPU para clustering de grafos

[2] https://live.boost.org/doc/libs/1_84_0/libs/math/doc/html/math_toolkit/dist_ref/dists/chi_squared_dist.html

en común [5]. Este algoritmo recibe como entrada un archivo con las aristas del grafo, las cuales representa de forma conveniente en un formato CSR (Compressed Sparse Row). Esta representación del grafo en formato CSR le permite llevar a cabo el proceso de label propagation paralelo de una forma más eficiente. Luego la salida de este algoritmo es la lista de los vértices etiquetados. También cabe señalar que este algoritmo no presenta la característica de utilizar aristas con pesos.

La implementación del algoritmo label propagation paralelo[3] ofrece diferentes combinaciones de métodos y políticas para llevar a cabo su procesamiento. Los métodos que presenta son dos, uno que usa primitivas de datos paralelos (0) y otro que usa tablas hash sin bloqueo (1). Luego para cada método ofrece las mismas políticas, las cuales se presentan a continuación:

- **Política 0:** Paralelismo en GPU en el núcleo.
- **Política 1:** Paralelismo en GPU fuera del núcleo sin superposición.
- **Política 2:** Paralelismo en GPU fuera del núcleo con superposición.
- **Política 3:** Paralelismo híbrido en CPU y GPU.
- **Política 4:** Con método (0) paralelismo en GPU en el núcleo con carga desequilibrada y con método (1) paralelismo en múltiples GPU fuera del núcleo con superposición.
- **Política 5:** Paralelismo en múltiples GPU (solo para método (1)).

El método (0) utiliza primitivas de datos paralelos a través de la librería moderngpu la cual se encuentra modificada para el caso de uso. En unas primeras pruebas fue posible observar que este método no funciona correctamente, pues no realizaba el algoritmo label propagation. Al realizar una revisión sobre la biblioteca modificada se pudo encontrar que estaba diseñada para funcionar bajo ciertas arquitecturas de GPU de NVIDIA, las cuales eran algo más antiguas y lamentablemente no se encontraba «Ampere» que es la que posee el servidor utilizado para las pruebas (Tabla 3.2). Luego se intentó actualizar la biblioteca de moderngpu[4] a una versión más reciente que si soportaba la arquitectura indicada pero no fue posible integrarla a la implementación de label propagation paralelizado puesto que presenta una muy baja escalabilidad.

[3] https://github.com/ykzw/galp
[4] https://github.com/moderngpu/moderngpu/wiki

Por este motivo se decide descartar realizar pruebas con el método (0).

Por último en el artículo de label propagation paralelo [5, pág. 575] se hace uso de dos métricas para comparar la precisión en la similitud entre los clusters obtenidos por esta implementación y un grupo de clusters referencia. Estas métricas son NMI (Normalized Mutual Information) que se basa en un enfoque teórico de la información y ARI (Adjusted Rand Index) que utiliza conteo de pares y tiene en cuenta el hecho de que una coincidencia entre ambos grupos de clusters se puede deber a una casualidad.

7.3. Evaluación Experimental

Con el objetivo de evaluar el desempeño del algoritmo de label propagation paralelizado mostrado en la Sección 7.2 en MetaBAT2, es que finalmente se integra el mismo a MetaBAT2. En esta integración se modifica la entrada de label propagation paralelizado para que reciba directamente las aristas del grafo generado en la etapa anterior a la de clustering.

Los datasets utilizados en la evaluación experimental son los que se muestra en la Tabla 7.1 utilizando el ambiente señalado en la Tabla 3.2.

Además, se ha fijado la semilla 1 (–seed=1) en MetaBAT2 al momento de evaluar el clustering original y el clustering con label propagation paralelizado, para que ambos estuvieran en igualdad de condiciones.

7.3.1. Evaluación 1

La primera parte de la evaluación consiste en ver la similitud entre los clusters obtenidos con MetaBAT2 y label propagation paralelizado. Para ello se ejecuta el algoritmo de label propagation paralelizado utilizando el método (1) y las políticas (0), (1), (2) y (3) con un rango de número de iteraciones de entre 10 y 200, que aumentan de 10 en 10, esto pretendiendo que con más iteraciones el resultado puede mejorar.

Las métricas utilizadas son las vistas en la Sección 7.2 (NMI y ARI). Adicionalmente, agregamos otras dos métricas, Precision y Recall, las que evalúan la similitud entre grupos de clusters basándose en verdaderos positivos y falsos negativos [7] a las cuales se les ha fijado un umbral

de la medida Overlap Score (OS) de 0.8 para las pruebas.

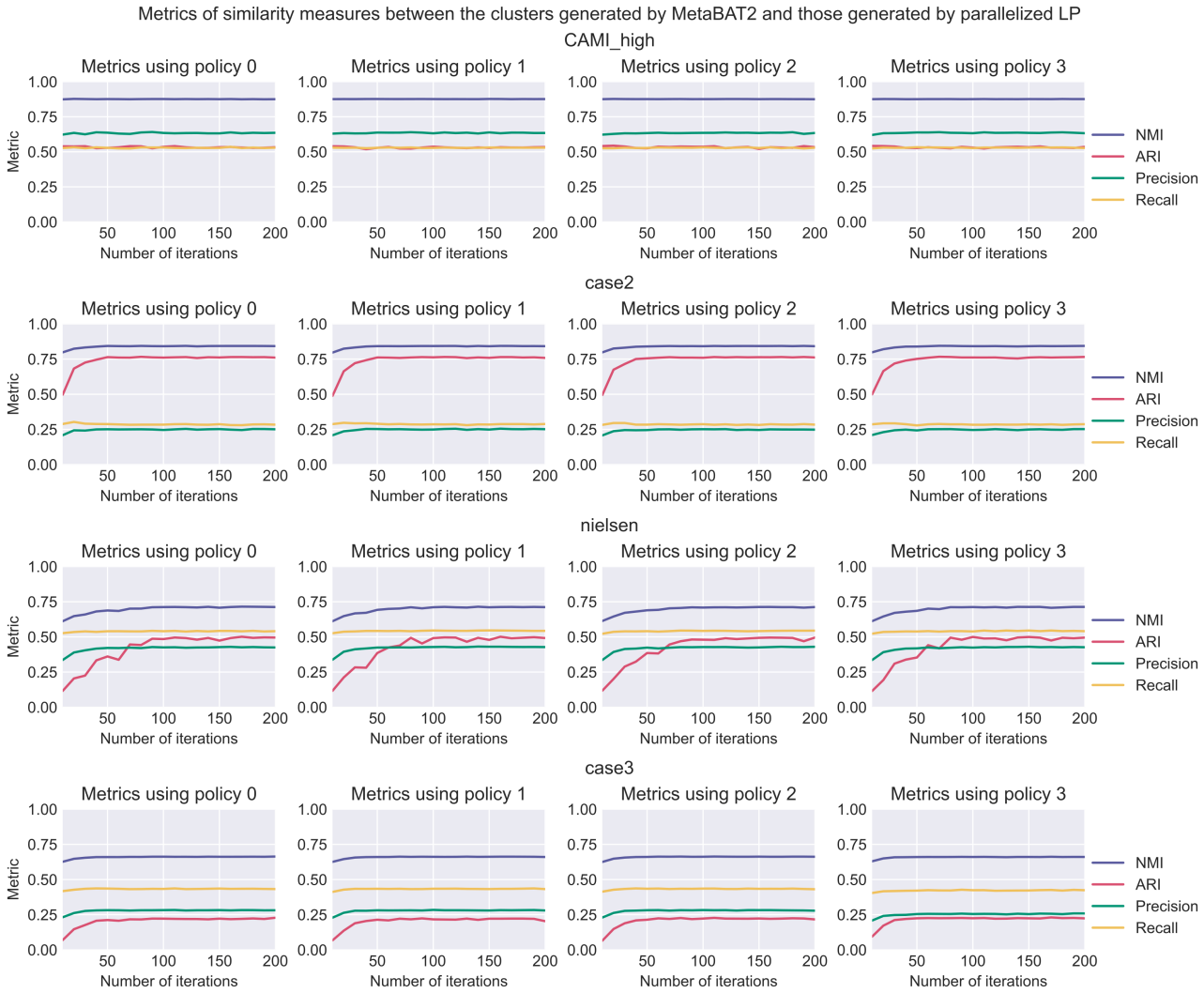


Figura 7.2: Métricas de similitud obtenidas con label propagation paralelizado

En la Figura 7.2 se pueden ver los resultados en promedio de la similitud de cada configuración de label propagation paralelizado con respecto a los grupos de clusters de referencia. Aquí se puede observar que en cada uno de los datasets, al variar la política, los resultados cambian de forma inapreciable. Esto indica que la política no influye en el grado de similitud alcanzado. También se puede ver que el valor de NMI es el más alto obtenido en cada dataset y que el valor de Recall en cada dataset no aumenta al incrementar el número de iteraciones.

Para Cami_high el NMI es de 0.88 seguido de Precision 0.63 y Recall y ARI de 0.53, es interesante mencionar que estos valores prácticamente no varían al aumentar el número de iteraciones en este caso. Respecto a la similitud encontrada, esta se puede considerar regular pues todas las

métricas están en un valor sobre 0.5.

En case2 se aprecia que ARI aumenta de 0.49 a 0.76 entre 10 y 50 iteraciones, estancándose hasta la iteración 200, lo que revela una mejoría de ARI producto de aumentar el número de iteraciones, mientras que las otras métricas aumentan ligeramente su valor. NMI y ARI están sobre 0.76 lo que indica un buen grado de similitud según estas métricas sin embargo Precision y Recall están entre 0.25 y 0.28 lo que indica todo lo contrario, pero dado que estas últimas métricas son más estrictas, se puede considerar que la similitud encontrada en general es de regular a mala.

En nielsen la mejoría de ARI, NMI y Precision es más notoria, ARI parte con un valor de 0.11 y alcanza 0.49 con 110 iteraciones, NMI parte con 0.61 y llega a 0.71 con 90 iteraciones y Precision de 0.33 a 0.42 en 50 iteraciones. Aquí Recall, ARI y Precision están entre 0.42 y 0.54 y NMI alcanza un valor de 0.71, lo que indica en general un grado de similitud de regular a bueno.

En case3 tal como sucede en case2, los resultados mejoran ligeramente al incrementar el número de iteraciones. Aquí Precision y ARI se encuentran entre 0.22 y 0.25, mientras que Recall llega a 0.42 y NMI a 0.66 lo que indica en general un grado de similitud de regular a malo.

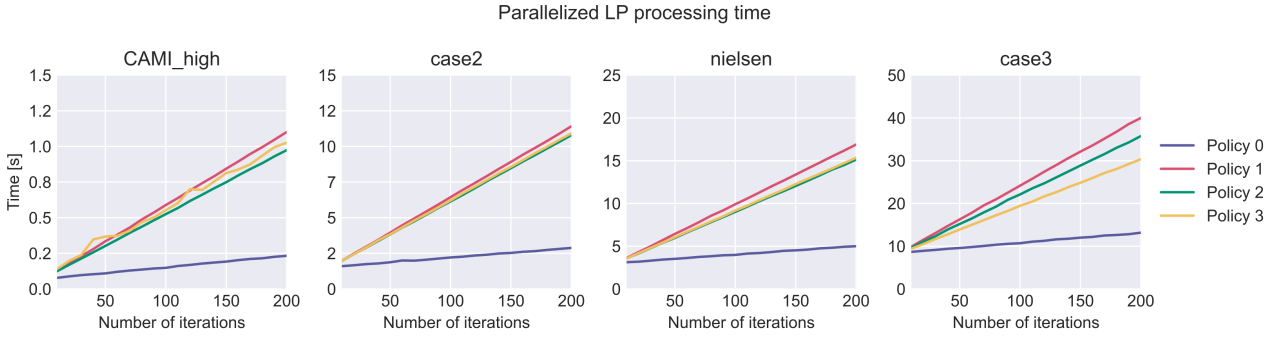


Figura 7.3: Tiempo de procesado de label propagation paralelizado

En la Figura 7.3 se puede ver el tiempo de procesado en promedio que se obtiene con label propagation paralelizado en cada dataset variando la política utilizada. Aquí se puede ver un patrón donde la política (0) es la más veloz incluso al ir aumentando el número de iteraciones, la cual es seguida de lejos por la política (3) que utiliza GPU + CPU y luego las políticas (2) y (1). En todos los casos se sigue un patrón lineal donde aumenta el tiempo al incrementar el número de iteraciones.

En CAMI_high el tiempo de procesamiento va de los 78 ms a 1.1 s de 10 a 200 iteraciones, mientras que con case3 que es el dataset más grande, este tiempo va de 8.7 s a 40 s.

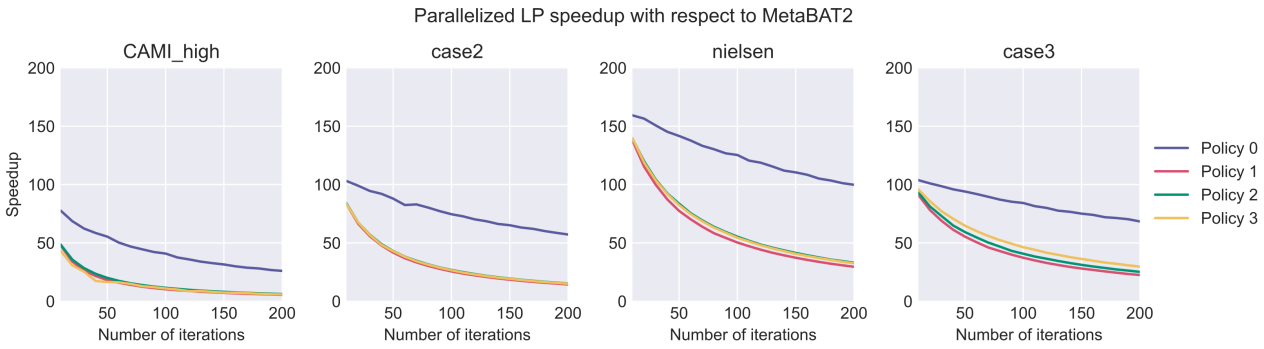


Figura 7.4: Speedup de label propagation paralelizado

En la figura 7.4 se muestra el speedup alcanzado por label propagation paralelizado respecto a la etapa de clustering de MetaBAT2 (Tabla 7.1). Aquí a medida que se incrementa el número de iteraciones el speedup va decreciendo, esto debido a que incrementa el tiempo, siendo el speedup más bajo el obtenido con CAMI_high con un valor de 5.5x.

Al analizar el caso con 50 iteraciones podemos observar que la política (0) con los cuatro datasets es de 55 a 141 veces más rápida que el clustering de MetaBAT2, mientras que las políticas (1), (2) y (3) son de 16 a 84 veces más rápidas. Si tenemos en cuenta los gráficos de la Figura 7.2 donde a partir de estas 50 iteraciones las métricas de similitud convergen para todas las políticas por igual, se puede considerar entonces que se alcanza el grado de similitud más alto con este número de iteraciones y que además es posible lograrlo con la política (0) hasta 141 veces más rápido. Por ejemplo, al usar este ajuste con case3 el tiempo de cómputo se reduce de 15 min con el clustering original de MetaBAT2 a 10 s con label propagation paralelizado.

7.3.2. Evaluación 2

CheckM [9] es un programa que provee un conjunto de herramientas para evaluar la calidad de los genomas recuperados, proporcionando estimaciones sólidas de la integridad y contaminación[5].

La segunda parte de la evaluación consiste en utilizar CheckM en conjunto con un benchmark de MetaBAT2 para evaluar la calidad de los clusters obtenidos con MetaBAT2 y label propagation

[5] https://github.com/Ecogenomics/CheckM/wiki/Introduction#about

paralelizado. Esto se lleva a cabo de forma similar a la explicada en el repositorio de MetaBAT[6], es decir, se evalúa cada set de clusters con las herramientas de CheckM, las cuales generan un archivo de resultado en formato «.txt» con la calidad encontrada en cada set, posteriormente cada archivo de resultado es procesado por el benchmark de MetaBAT2, el cual finalmente entrega datos acerca del Precision y Recall del clustering.

Tabla 7.2: Calidad del clustering sin archivo de abundancia

FASTA File Name	MetaBAT2 Clustering											Parallelized Label Propagation using policy 0 and 50 iterations										
CAMI_high		Recall											Recall									
	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
	0.6	149	149	144	141	140	136	128	124	113	97	0.6	140	140	138	136	136	136	133	130	126	108
	0.7	148	148	143	140	139	135	128	124	113	97	0.7	140	140	138	136	136	136	133	130	126	108
	0.8	146	146	141	138	137	133	128	124	113	97	0.8	140	140	138	136	136	136	133	130	126	108
	0.9	143	143	138	135	134	130	125	121	111	95	0.9	137	137	135	134	134	134	131	128	124	106
	0.95	137	137	132	129	128	125	120	116	107	92	0.95	133	133	131	130	130	130	127	124	120	102
	0.99	100	100	95	92	92	89	84	80	74	64	0.99	96	96	94	93	93	93	90	87	85	71
case2_assembly		Recall											Recall									
	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
	0.6	64	64	59	51	47	40	26	19	12	5	0.6	45	45	44	39	38	34	32	27	12	6
	0.7	53	53	48	41	37	31	18	13	7	4	0.7	38	38	37	34	33	29	27	22	8	5
	0.8	42	42	37	31	28	24	15	11	6	3	0.8	27	27	26	23	22	19	19	15	5	4
	0.9	32	32	28	22	20	16	9	6	2	1	0.9	19	19	18	15	14	12	12	9	1	0
	0.95	29	29	25	19	17	13	8	5	2	1	0.95	15	15	14	11	10	8	8	6	0	0
	0.99	10	10	8	7	6	4	1	1	1	1	0.99	3	3	2	0	0	0	0	0	0	0
nielsen		Recall											Recall									
	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
	0.6	240	240	203	173	137	100	67	35	13	2	0.6	316	316	270	233	193	149	95	50	16	3
	0.7	232	232	195	165	129	93	61	33	13	2	0.7	308	308	262	225	186	143	91	49	16	3
	0.8	225	225	188	158	125	90	58	32	13	2	0.8	295	295	249	212	174	131	82	44	14	3
	0.9	207	207	170	143	114	82	53	29	12	2	0.9	273	273	227	191	158	118	75	38	11	2
	0.95	171	171	136	111	84	58	31	18	8	0	0.95	221	221	180	147	120	84	54	26	7	0
	0.99	77	77	49	35	19	14	10	8	5	0	0.99	96	96	61	43	29	19	12	9	4	0
case3_assembly		Recall											Recall									
	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
	0.6	136	136	124	107	80	64	48	30	15	9	0.6	212	212	198	186	170	147	113	84	43	21
	0.7	122	122	110	93	66	53	39	27	12	7	0.7	187	187	174	162	146	126	97	71	37	17
	0.8	103	103	91	74	51	40	29	19	9	5	0.8	162	162	149	137	121	102	74	52	27	11
	0.9	79	79	67	50	30	19	10	3	2	1	0.9	100	100	90	79	65	50	29	14	5	2
	0.95	57	57	49	35	22	13	5	1	0	0	0.95	63	63	53	44	33	24	10	3	1	0
	0.99	21	21	17	12	8	5	0	0	0	0	0.99	32	32	23	15	7	5	2	0	0	0

En la Tabla 7.2 se muestran los resultados del clustering obtenidos, sin usar archivo de abundancia, donde se utilizó el clustering original de MetaBAT2 y el de label propagation paralelizado con la mejor configuración encontrada en la evaluación anterior (Sección 7.3.1). Tal como se discutió anteriormente, la mejor configuración consiste en la política 0 y 50 iteraciones. Aquí en CAMI_high se puede ver que los clusters encontrados con Precision >=0.99 y Recall 0.95 son 64 con el clustering de MetaBAT2 y 71 con el de label propagation paralelizado, lo que indica un incremento en la cantidad de clusters que presentan la calidad más alta posible. Esto se repite con nielsen y case3, donde con Recall >=0.8 y Precision 0.8 (calidades consideradas buenas), la cantidad de clusters encontrados por label propagation paralelizado es igual e incluso superior

[6] https://bitbucket.org/berkeleylab/metabat/wiki/Best%20Binning%20Practices

en la gran mayoría de casos a la encontrada con el clustering de MetaBAT2. Mientras tanto con case2 los resultados obtenidos de ambos algoritmos de clustering son más bien similares, e incluso algunos clusters de Precision entre 0.95 y 0.99 se pierden al utilizar label propagation paralelizado.

Tabla 7.3: Calidad del clustering con archivo de abundancia

Files	MetaBAT2 Clustering											Parallelized Label Propagation using policy 0 and 50 iterations										
		Recall											Recall									
	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
case2_assembly + case2_depth.txt	0.6	192	192	158	135	111	93	76	57	29	17	0.6	58	58	55	50	47	44	39	32	20	9
	0.7	191	191	157	134	110	92	75	57	29	17	0.7	52	52	49	44	41	38	33	27	15	7
	0.8	185	185	151	128	104	87	71	54	26	17	0.8	46	46	43	38	35	33	29	23	11	6
	0.9	173	173	140	118	96	80	64	47	24	17	0.9	33	33	31	27	24	23	20	15	7	4
	0.95	150	150	117	98	77	64	50	38	23	16	0.95	22	22	20	17	15	14	13	9	4	2
	0.99	72	72	46	33	23	17	13	8	4	1	0.99	7	7	6	4	3	3	3	2	1	1
		Recall											Recall									
	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95	Precision	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.95
case3_assembly + case3_depth.txt	0.6	875	875	796	725	638	541	442	345	202	78	0.6	252	252	242	228	205	179	144	105	57	25
	0.7	848	848	769	699	612	517	423	329	192	75	0.7	223	223	213	199	177	153	121	87	49	20
	0.8	787	787	708	638	552	460	370	282	160	55	0.8	182	182	172	158	136	113	87	63	36	14
	0.9	640	640	562	493	414	329	247	184	100	35	0.9	115	115	106	93	74	56	36	22	9	4
	0.95	483	483	410	347	279	206	145	107	64	24	0.95	64	64	57	47	35	24	12	5	2	0
	0.99	198	198	155	114	78	47	26	18	16	7	0.99	28	28	22	14	7	5	2	0	0	0

Si bien el archivo de abundancia es opcional, se realizaron pruebas con case2 y case3 de forma idéntica a la mostrada en el repositorio de MetaBAT, esto para comparar el desempeño del clustering de MetaBAT2 con el de label propagation paralelizado al usar el archivo de abundancia. En las pruebas realizadas el grafo generado en ambos casos es algo más pequeño, con case2 se generan 13.839.753 aristas, mientras que con case3 son 50.189.985 aristas. En la Tabla 7.3 se muestran los resultados de la calidad obtenida del experimento donde en prácticamente todos los casos label propagation paralelizado tiene una cantidad de clusters inferior a la encontrada por el algoritmo de clustering original de MetaBAT2. Esto se puede deber principalmente a que en la implementación de MetaBAT2 el peso de las aristas cambia y se adapta a la información proporcionada por el archivo de abundancia, mientras que label propagation paralelizado no considera los pesos de las aristas. Luego, es promisorio incorporar esta característica en label propagation paralelizado en un futuro pues podría alcanzar e incluso aumentar la calidad del clustering cuando se presenta el archivo de abundancia en MetaBAT2.

Capítulo 8

Conclusiones

En esta memoria de título se ha presentado un gestor de compresión y descompresión acelerado por CPU y GPU para la obtención de los datasets de MetaBAT2. El gestor de compresión permite utilizar diferentes compresores, en particular los de nvCOMP y otros dos compresores desarrollados durante la memoria (CFA1 y CFA2) vistos en el Capítulo 4. Este gestor permite realizar una compresión más rápida que GZIP con todos los compresores, al mismo tiempo que logra alcanzar e incluso superar la razón de compresión de GZIP con los compresores GDeflate, ANS y CFA2. Además, la integración de este gestor en MetaBAT2 permite realizar el proceso global de lectura en menor tiempo ya que logra descomprimir más rápido.

En la implementación se ha dejado al usuario la elección del compresor a utilizar, aunque se recomienda el uso de ANS y CFA2 debido a sus ventajas en tiempos de compresión y razón de compresión respectivamente. En cuanto a la descompresión tanto fuera como dentro de MetaBAT2 no existe una marcada diferencia entre compresores del gestor, por lo que se debieran tomar en cuenta otros factores para la elección de compresores, como su razón de compresión, tiempo de compresión y uso de memoria GPU.

También se ha logrado evaluar e integrar un algoritmo de clustering de grafos acelerado por GPU y CPU que usa el mismo tipo de algoritmo de clustering que utiliza MetaBAT2 (label propagation). Con este algoritmo se obtuvo un clustering de similitud regular con respecto al generado por MetaBAT2, de acuerdo a las métricas de la Sección 7.3.1. En términos de la calidad de los resultados (Sección 7.3.2) el clustering paralelo fue igual incluso mejor al de MetaBAT2

sin usar archivo de abundancia. Mientras que cuando se utilizó el archivo de abundancia esta calidad del clustering disminuyó notoriamente pues encontró menos clusters en todos los niveles de calidad Precision/Recall. En cuanto al tiempo de cómputo del algoritmo se pudo ver que con la mejor configuración logra una aceleración de hasta 141x con respecto al algoritmo de clustering de MetaBAT2.

Finalmente, en la implementación se ha dejado al usuario la elección de utilizar o no el algoritmo de clustering paralelizado, pudiendo modificar la política y el número de iteraciones. Si el usuario no usa el archivo de abundancia es recomendable utilizar este algoritmo, esto sin dejar de lado las recomendaciones acerca de las mejores prácticas para el clustering que provee MetaBAT.

Referencias

[1] Raul Bilc. *Study: Most searched GPUs in 2022*. 2023. URL: https://www.razzem.com/study-the-most-searched-graphics-cards-in-2022/?utm_source=ixbtcom.

[2] Robit Chandra et al. *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558606718.

[3] Jarek Duda. *Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding*. 2014. arXiv: 1311.2540 [cs.IT].

[4] Dongwan D. Kang et al. «MetaBAT, an efficient tool for accurately reconstructing single genomes from complex microbial communities». En: *PeerJ* 3 (2015), e1165. ISSN: 2167-8359. DOI: 10.7717/peerj.1165.

[5] Yusuke Kozawa, Toshiyuki Amagasa e Hiroyuki Kitagawa. «GPU-Accelerated Graph Clustering via Parallel Label Propagation». En: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. CIKM '17. Singapore, Singapore: Association for Computing Machinery, 2017, págs. 567-576. ISBN: 9781450349185. DOI: 10.1145/3132847.3132960.

[6] Fernando Meyer et al. «Tutorial: assessing metagenomics software with the CAMI benchmarking toolkit». En: *Nature protocols* 16.4 (2021), págs. 1785-1801.

[7] Tamás Nepusz, Haiyuan Yu y Alberto Paccanaro. «Detecting overlapping protein complexes in protein-protein interaction networks». En: *Nature Methods* (2012), págs. 471-472. DOI: 10.1038/nmeth.1938.

[8] NVIDIA. *CUDA C Programming Guide*. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[9] Donovan H. Parks et al. «CheckM: assessing the quality of microbial genomes recovered from isolates, single cells, and metagenomes». En: *Genome Research* (2015), págs. 1043-1055. DOI: 10.1101/gr.186072.114.

[10] Satu Elisa Schaeffer. «Graph clustering». En: *Computer Science Review* 1.1 (2007), págs. 27-64. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2007.05.001.

[11] Bertil Schmidt et al. «Chapter 7 - Compute Unified Device Architecture». En: *Parallel Programming*. Ed. por Bertil Schmidt et al. Morgan Kaufmann, 2018, págs. 225-285. ISBN: 978-0-12-849890-3. DOI: 10.1016/B978-0-12-849890-3.00007-1.

[12] Erik Schmidt y Nikolay Sakharnykh. *Accelerating Lossless GPU Compression with New Flexible Interfaces in NVIDIA nvCOMP*. 2022. URL: https://developer.nvidia.com/blog/accelerating-lossless-gpu-compression-with-new-flexible-interfaces-in-nvidia-nvcomp/.

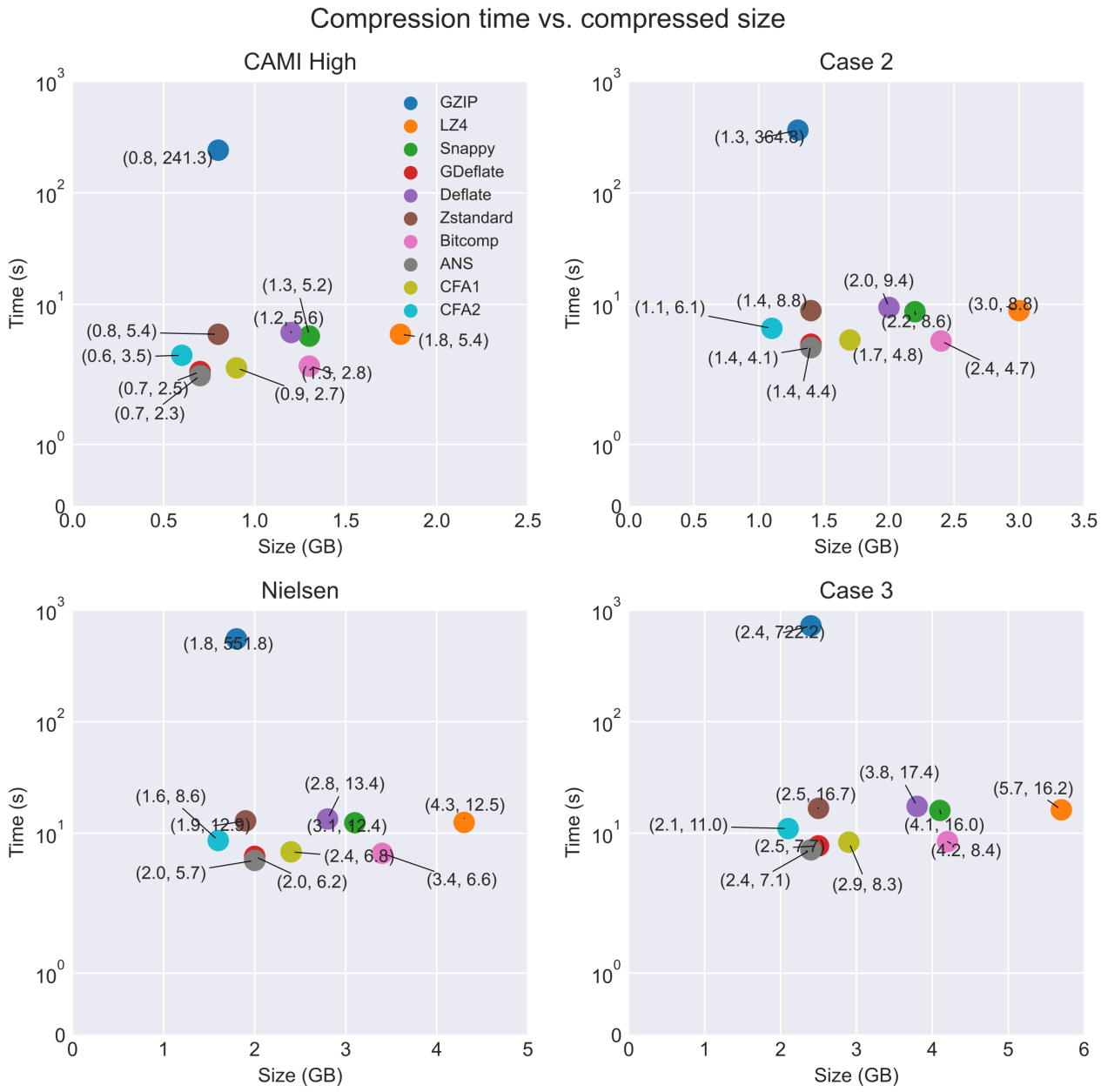
[13] Yury Uralsky. *Accelerating Load Times for DirectX Games and Apps with GDeflate for DirectStorage*. 2022. URL: https://developer.nvidia.com/blog/accelerating-load-times-for-directx-games-and-apps-with-gdeflate-for-directstorage/.

[14] Shaohui Xie et al. «CURC: a CUDA-based reference-free read compressor». En: *Bioinformatics* 38.12 (2022), págs. 3294-3296. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btac333.

[15] J. Ziv y A. Lempel. «A universal algorithm for sequential data compression». En: *IEEE Transactions on Information Theory* 23.3 (1977), págs. 337-343. DOI: 10.1109/TIT.1977.1055714.

Anexos

1. Tiempo de compresión vs. tamaño del archivo comprimido



2. Tiempo de descompresión vs. tamaño del archivo comprimido