MODELO DE PROCESAMIENTO PARALELO EN ARQUITECTURAS HETEROGÉNEAS PARA LA CONSTRUCCIÓN DE GRAFOS EN EL ENSAMBLAJE DE-NOVO DE GENOMAS

NELSON ENRIQUE VERA PARRA

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS
FACULTAD DE INGENIERÍA
PROGRAMA DE DOCTORADO EN INGENIERÍA
ÉNFASIS EN CIENCIA DE LA INFORMACIÓN Y EL CONOCIMIENTO
BOGOTÁ, D.C. 2018

MODELO DE PROCESAMIENTO PARALELO EN ARQUITECTURAS HETEROGÉNEAS PARA LA CONSTRUCCIÓN DE GRAFOS EN EL ENSAMBLAJE DE-NOVO DE GENOMAS

Tesis de grado para optar por el título de Doctor en Ingeniería

Presentada por:
M.Sc NELSON ENRIQUE VERA PARRA

Director: PhD JOSÉ NELSON PÉREZ CASTILLO

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS
FACULTAD DE INGENIERÍA
PROGRAMA DE DOCTORADO EN INGENIERÍA
ÉNFASIS EN CIENCIA DE LA INFORMACIÓN Y EL CONOCIMIENTO
BOGOTÁ, D.C. 2018

COMISIÓN DE DOCTORADO

Esta tesis, titulada "MODELO DE PROCESAMIENTO PARALELO EN ARQUITECTURAS HETEROGÉNEAS PARA LA CONSTRUCCIÓN DE GRAFOS EN EL ENSAMBLAJE DE-NOVO DE GENOMAS", escrita por Nelson Enrique Vera Parra, ha sido aprobada en cuanto a estilo y contenido intelectual.

Hemos leído esta tesis y la aprobamos,

_____
Doctor Jurado 1

_____
Doctor Jurado 2

_____
Doctor Jurado 3

_____
Doctor JOSÉ NELSON PÉREZ CASTILLO Director

Fecha de la defensa:

RESUMEN

En el presente proyecto se diseñó un modelo de procesamiento paralelo masivo sobre arquitecturas heterogéneas para acelerar y facilitar el tratamiento de k-mers en los procesos relacionados a la construcción de grafos en el ensamble genómico de-novo. El modelo incluye 3 principales aportes: una nueva estructura de datos denominadas CISK para representar de forma indexada y compacta los super k-mers y sus minimizer de una lectura y dos patrones de paralelización masiva, uno para obtener los m-mers canónicos de un conjunto de lecturas y otro para realizar la búsqueda de super k-mers basados en semillas tipo minimizer.

Durante el proyecto se realizaron 4 procesos de evaluación: - una evaluación preliminar que permitió determinar que el proceso de ensamblaje de-novo es la etapa más compleja y con mayores requerimientos computacionales de un flujo de trabajo típico de lecturas genómicas y trancriptómicas, - una segunda evaluación que evidenció que las tareas asociados al tratamiento de k-mers son procesos que representan cuellos de botella debido a su alta exigencia de memoria, - una tercera evaluación que proyectó a las técnicas de particionamiento en disco basadas en super k-mers por semillas tipo minimizer como candidatas a potencializarlas mediante computación paralela masiva sobre plataformas heterogéneas, - y por último una evaluación al modelo propuesto que mostró sus ventajas obteniendo un speed-up hasta de 6.69x sobre procesos similares en herramientas contadoras de k-mers muy reconocidas que realizan paralelización en CPU.

El código de la implementación del modelo se encuentra disponible en el repositorio https://github.com/BioinfUD/K-mersCL. Esta implementación consta de un código host y dos kernels en OpenCL, uno para minimizer canónicos y otro para signature.

Palabras Claves: Computación heterogénea paralela, grafos de De Bruijn, GPU, ensamblaje genómico de-novo, minimizers, particionamiento en disco basado en semilla, procesamiento paralelo de k-mers.

ABSTRACT

In the present project, a massive parallel processing model on heterogeneous architectures was designed to accelerate and facilitate the processing of k-mers in the tasks related to the construction of graphs in the de-novo genomic assembly. The model includes 3 main contributions: a new data structure called CISK to represent in an indexed and compact way the super k-mers and their minimizers and two massive parallelization patterns, one to obtain the canonical m-mers of a set of reads and another to perform the search for super k-mers based on seeds type minimizer.

During the project, 4 evaluation processes were performed: - a preliminary evaluation that allowed determining that the de-novo assembly process is the most complex stage and with the highest computational requirements of a typical workflow of genomic and transcriptomic reads, - a second evaluation that showed that the tasks associated with the treatment of k-mers are processes that represent bottlenecks due to their high demand of memory, - a third evaluation that allowed select the disk partitioning techniques based on super k-mers using seeds type minimizer as base methodology for the design of massive parallel computing model to process k-mers on heterogeneous platforms, - and finally an evaluation of the proposed model that evidenced its advantages obtaining a speed-up of 4.31x on similar processes in highly recognized k-mers counting tools that perform parallelization in CPU.

The model implementation code is available in the repository https://github.com/BioinfUD/K-mersCL. This implementation consists of a host code and two kernels in OpenCL, one for canonical minimizer and another for signature.

Keywords: De Bruijn graphs, De-novo genomic assembly, GPU, Minimizers, Parallel heterogeneous computing, parallel processing of k-mers, Seed-based data partitioning on disk.

Dedicatoria

A la memoria de Nelly Rengifo por iluminar mis ideas

A mi familia su por su confianza y respaldo

A Tatiana Guevara por su apoyo, compañía, comprensión y amor

Agradecimientos

A la Universidad Distrital Francisco José de Caldas

Al equipo administrativo y académico del Doctorado

A mi director, el Doctor José Nelson Pérez Castillo y al grupo de investigación GICOGE

A mi asesor externo, el Doctor Luis Fernando Cadavid, al grupo de investigación en inmunología Evolutiva e Inmunogenética y al Instituto de Genética de la Universidad Nacional (IGUN)

Al Centro de Computación de Alto Desempeño (CECAD)

Al pupilo Cristian Rojas

A mis amigos, compañeros y colegas de la Universidad.

Tabla de contenido

Lista de tablas

Lista de figuras

Lista de algoritmos

CECAD:	Centro de Cómputo de Alto Desempeño de la Universidad Distrital
CIDC:	Centro de Investigaciones y Desarrollo Científico de la Universidad Distrital
CISK:	Compact and Indexed representation of Super k-mers
Contig:	Segmentos de ADN superpuestos
CPU:	Central Processing Unit
CUDA:	Compute Unified Device Architecture
CUs:	Compute Units
DRAM:	Dynamic Random Access Memory
DSP:	Digital Signal Processor
FPGA:	Field Programmable Gate Array
GPU:	Graphics Processing Unit
Hash:	Funciones tipo resumen
IGUN:	Instituto de Genética de la Universidad Nacional
K-mer:	Sub-secuencia de longitud k
M-mer:	Sub-secuencia de longitud m
OpenCL:	Open Computing Language
Pair-end:	Lectura pareadas – secuenciadas desde ambos extremos
Pes:	Processing Elements
SMs:	Streaming Multiprocessors
Speed-up:	Factor de Aceleración
SVM:	Shared Virtual Memory

1. INTRODUCCIÓN

Secuenciación de nueva generación: Bajo costo de secuenciación / Alto costo de análisis

En la última década se ha dado un cambio en cuanto al uso generalizado de la tecnología de Sanger para la obtención de secuencias de ácidos nucléicos, la cual había predominado durante más de 2 décadas. Ahora, las metodologías de secuenciación de alto rendimiento, permiten la obtención de grandes cantidades de datos con una menor inversión (Metzker, 2010). Mientras que el costo por 1M de bases secuenciadas utilizando la tecnología Sanger era de U$2400, la misma cantidad de bases secuenciadas con metodologías de alto rendimiento tiene un costo que oscila entre U$0.05 y U$0.15. Mientras que en una ejecución en la Tecnología Sanger se obtenía cómo máximo 52.5K bases, actualmente con un secuenciador HiSeq 2500 de Illumina se obtienen hasta 1000G bases por ejecución. Lo prometido actualmente por Illumina es llegar muy pronto a un valor de U$1000 por genoma completo en un tiempo cercano a un día.

La evolución en la tecnología de secuenciación descrita en el párrafo anterior permite el acceso masivo de la comunidad científica y comercial a los servicios de secuenciación, lo que genera que día a día la cantidad de datos biológicos secuenciados crezcan exageradamente. Esa gran montaña de datos biológicos abre la puerta a la evolución de procesos investigativos tales como la medicina personalizada o el entendimiento y aprovechamiento de la biodiversidad de nuestro planeta. Sin embargo obtener grandes cantidades de datos a bajos precios y en poco tiempo es solo el primer paso, porque el siguiente reto es mucho más grande y corresponde al procesamiento de dichos datos para que sean útiles al biólogo o al médico.

El análisis computacional de datos genómicos y transcriptómicos involucra una serie de procesos que se agrupan en 3 categorías: análisis primario, secundario y terciario (Rudy, 2010). El análisis primario normalmente está inmerso en el proceso de secuenciación y es realizado por las máquinas secuenciadoras al final del proceso con el objeto de arrojar los datos en un formato estándar FASTQ (Cock. et al., 2010) con metadatos relacionados a la calidad de las secuencias. El análisis secundario se puede resumir en tres tipos de procesos: el preprocesamiento para limpiar las secuencias y garantizar un nivel de calidad (Castillo, J. N. P., Parra, N. E. V., & Ramirez, L. M. G., 2014), el ensamblaje de las secuencias para reconstruir el genoma o el transcriptoma (Miller, J. R., Koren, S., & Sutton, G., 2010) y la detección de variaciones (Cheung & Spielman, 2009). Por último, el análisis terciario reúne todos los procesos específicos que dependen del tipo de investigación biológica, como por ejemplo: anotación, expresión diferencial, análisis de variaciones genómicas, entre otras.

La mayoría de procesos bioinformáticos nombrados en el párrafo anterior exigen unas características computacionales muy altas, lo cual ha limitado los centros bioinformáticos a centros computacionales de alto desempeño; por ejemplo el proceso de ensamblaje de un conjunto de datos transcripotómico de 13.08 Gbp (107M pares de lecturas de 76bp de longitud) usando una máquina de 32 núcleos de 1.87Ghz y 512GB de RAM requiere hasta 160 horas de ejecución (Henschel, R. et al, 2012). En pocas palabras mientras el costo de secuenciación baja y permite el acceso a datos biológicos a medianos y pequeños investigadores, el costo de análisis computacional sube porque se limita a grandes centros de muy alto desempeño.

Ensamblaje de-novo: El proceso bioinformático de mayor costo computacional

Como etapa preliminar de esta tesis, en el marco del convenio entre el CECAD (Centro de Computación de Alto Desempeño de la Universidad Distrital) y el IGUN (Instituto de Genética de la Universidad Nacional) se desarrolló una evaluación de desempeño computacional de las principales etapas de procesamiento bioinformático con el objeto de identificar los cuellos de botella. Los resultados de esta evaluación evidenciaron que la etapa que más costo computacional exige tanto para datos genómicos como transcriptómicos es el ensamblaje de-novo, adicional a esto es la etapa que presenta menor paralización. (Vera-Parra, Pérez-Castillo & Rojas-Quintero, 2015). En las figuras 1.1 y 1.2 se pueden observar los resultados para el análisis de un transcriptoma, mientras que en las figuras 1.3 y 1.4 se observan para el análisis de un genoma.

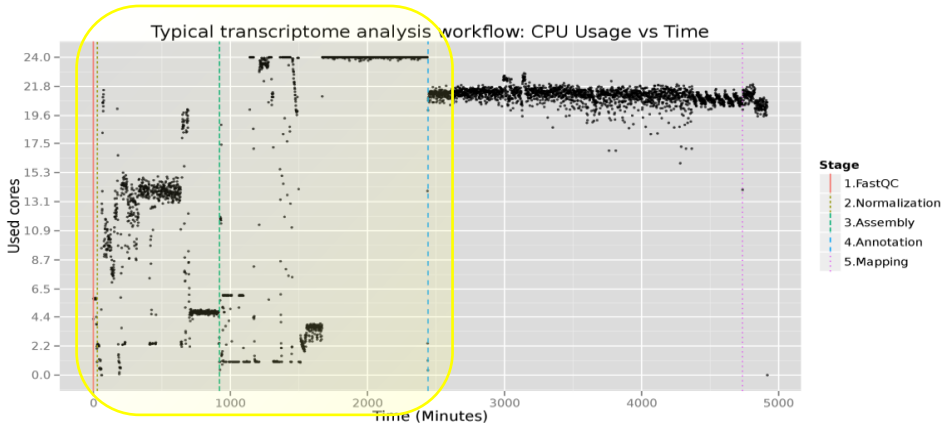


Figura 1.1 Evaluación de un flujo de trabajo típico en el análisis de un transcriptoma - Uso de CPU vs Tiempo. Fuente: Autor.

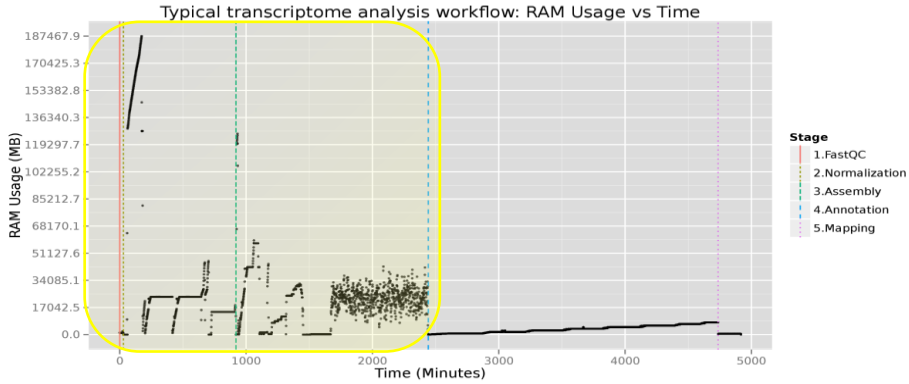


Figura 1.2 Evaluación de un flujo de trabajo típico en el análisis de un transcriptoma - Uso de RAM vs Tiempo. Fuente: Autor.

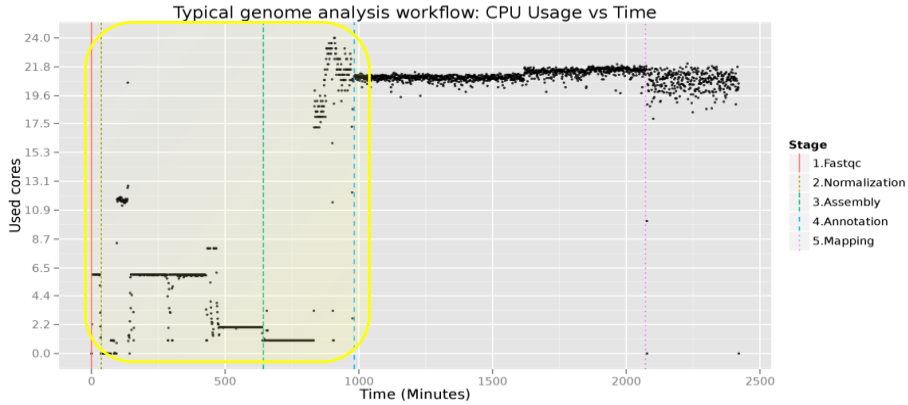


Figura 1.3 Evaluación de un flujo de trabajo típico en el análisis de un genoma - Uso de CPU vs Tiempo. Fuente: Autor.

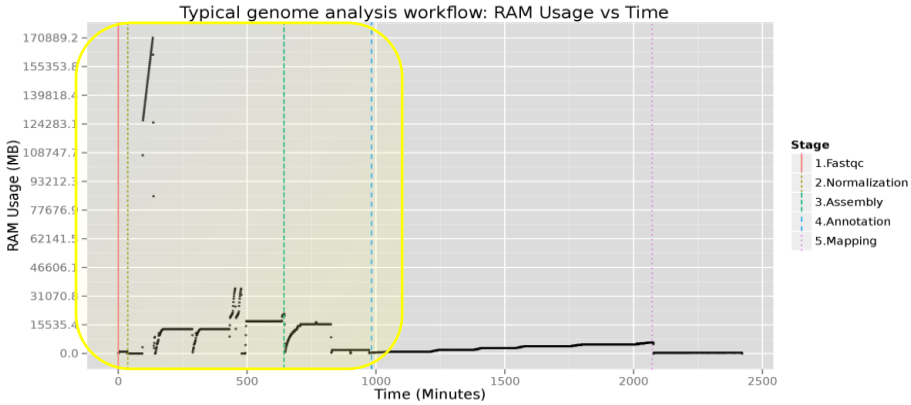


Figura 1.4 Evaluación de un flujo de trabajo típico en el análisis de un genoma - Uso de RAM vs Tiempo. Fuente: Autor.

Debido al tamaño de un transcripto completo y a la tendencia de las tecnologías de secuenciación a aumentar la longitud de la lecturas, el ensamblaje de datos transcriptómicos será un problema cada vez menor (Martin & Wang, 2011), sin embargo debido a la enorme longitud de un genoma completo, el ensamblaje de datos genómicos es actualmente uno de los principales retos de la bioinformática.

Construcción de grafos: La sub-etapa del ensamblaje de-novo de mayor costo computacional

Los resultados de la evaluación descrita en los párrafos anteriores motivaron a realizar una segunda evaluación de desempeño específicamente sobre las etapas del proceso de ensamblaje de-novo para datos genómicos (Vera-Parra, Pérez-Castillo & Rojas-Quintero, 2015). Los resultados arrojados evidencian que la sub-etapa que más costo computacional exige es la de obtención y procesamiento de los k-mers (nodos y aristas del grafo). En la figura 1.5 se puede observar las exigencias computacionales del ensamblador SOAPdenovo2 (Luo & Ruibang et al., 2012).

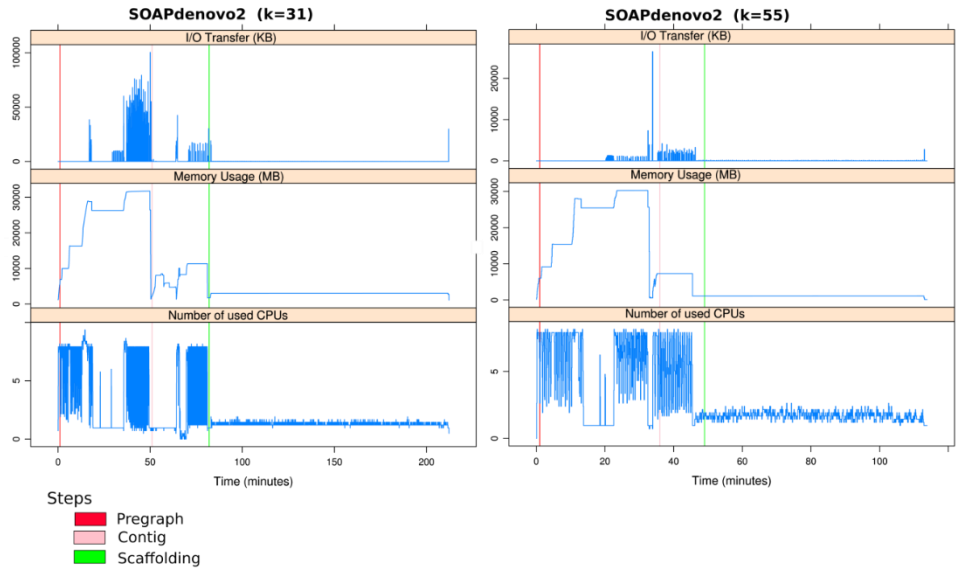


Figura 1.5. Evaluación de desempeño específicamente sobre las etapas del proceso de ensamblaje de-novo. Uso de CPU vs. Tiempo. Fuente: Autor.

Una de las principales razones por la que los tiempos de procesamiento para el ensamblaje de-novo no se han logrado reducir sustancialmente, son los altos limitantes que se han encontrado en el momento de paralelizar el proceso de construcción de grafos para poder realizar la ejecución en plataformas many-thread, como por ejemplo GPU - Unidades de procesamiento gráfico (Wen-Mei, 2011). Los limitantes se centran esencialmente en tres tópicos: la exagerada exigencia de memoria para la representación de los grafos, la alta dependencia

de los procesos que impide su ejecución simultánea y la alta necesidad de comunicación entre hilos.

Paralelización sobre plataformas Many Core: Una solución viable para la aceleración de procesos intensivos.

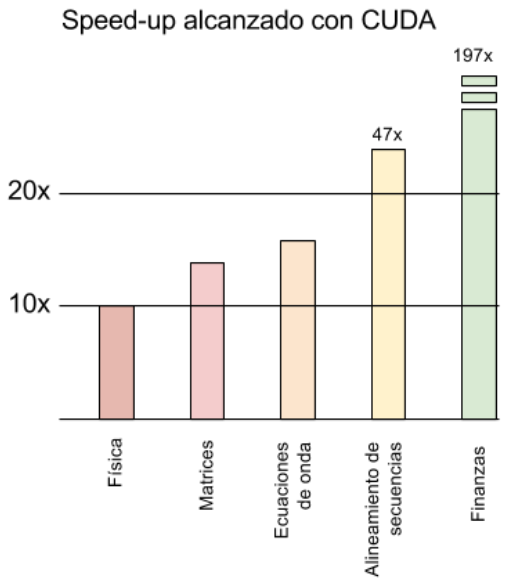


Figura 1.6 Speed-up alcanzado por CUDA. Imagen modificada. Fuente: http://www.beyond3d.com/content/articles/12/5

La figura 1.6 evidencia la aceleración que se logra conseguir cuando se paraleliza de forma eficiente un proceso: hasta 1000% en el área de física, hasta 4700% para el alineamiento de secuencias biológicas y hasta un 19700% en el área de finanzas. En el área de Bioinformática los principales avances se han dado en el alineamiento de secuencias (Schatz MC et al., 2007), (Shi H. et al., 2010), (Trapnell, C., & Schatz, M. C., 2009) y en la bioestadística (Zhou, J., et al., 2011). Con estas cifras es evidente que si se logra conseguir un modelo de procesamiento paralelo para la construcción de grafos, que supere los limitantes de memoria, dependencia y ancho de banda se alcanzará acelerar sustancialmente el proceso de ensamblaje de-novo de genomas.

1.1. Antecedentes

Ensambladores genómicos de-novo Overlap

Los primeros ensambladores que se tiene referencia usaron técnicas de solapamiento de lecturas. En la era de secuenciación de Sanger fueron muy conocidos 5 ensambladores basados en solapamientos de lecturas: Los primeros 3 fueron usados para reconstruir el genoma de una Drosophila: CAP3 (Huang X.

& Madan, 1999), CELERA (Myers E. et al., 2000), ARACHNE (Batzoglou S. et al., 2002). Un cuarto ensamblador tomó mucha importancia cuando logró reconstruir el genoma de un ratón a partir de lecturas Sanger (Mullikin, J. C., & Ning, Z, 2003). Por último un ensamblador de propósito general basado en solapamientos jerárquicos: PHRAP (Bastide, M., & McCombie, W. R. 2007).

A finales de la primera década de este siglo aparecieron los primeros ensambladores de lecturas cortas basados en solapamientos. Principalmente estaban enfocados en el ensamblaje de bacterias: NEWBLER (Chaisson, Mark J., & Pavel A., 2008), CABOG (Miller, J. R., et al 2008), SHORTY (Hernandez, D., et al, 2008), EDENA (Hossain, M. S., Azimi, N., & Skiena, S., 2009), FORGE (DiGuistini, S., et al., 2009).

La actual década se ha caracterizado por ensambladores que buscan optimizar los algoritmos de solapamiento con el fin de reducir el costo computacional: SGA (Simpson, J. T., & Durbin, R., 2012) presenta una estructura de datos comprimida, READJOINER (Gonnella, G., & Kurtz, S., 2012) propone un algoritmo de solapamiento basado en grafos de cadenas para reducir la exigencia de memoria y disminuir el tiempo de procesamiento. SAGE (Ilie, L., Haider, B., Molnar, M., & Solis-Oba, R., 2014) propone un algoritmo basado en grafos de solapamientos de cadenas y técnicas estadísticas como máxima verosimilitud.

Ensambladores genómicos de-novo Kmer - De Bruijn

A comienzos de la década pasada se tiene referencia del primer ensamblador genómico que usa caminos Eulerianos para reconstruir genomas aún a partir de secuencias Sanger: EULER (Pevzner, P. A., Tang, H., & Waterman, M. S., 2001). A finales de la década pasada surgen una serie de ensambladores que proponen los grafos De Bruijn como método para reducir el costo computacional que exigen los algoritmos de solapamiento: VELVET (Zerbino, D. R., & Birney, E. 2008), ALLPATHS (Butler, J. et al., 2008), ALLPATHS 2 (I. Maccallum, D. et al., 2009), ABYSS (Birol, I., et al., 2009).

A comienzos de la actual década se propone un ensamblador que busca optimizar el uso de los grafos de De Bruijn mediante algoritmos iterativos: IDBA (Peng, Y. et al., 2010). En el mismo año se lanza un ensamblador con la capacidad de reconstruir genomas del tamaño del genoma humano: SOAPdenovo (Li, R. et al., 2010). Dos años más adelante se lanza la segunda versión de éste ensamblador: SOAPdenovo2, con la novedad que reduce el consumo de memoria en la construcción de grafos.

Ensambladores genómicos de novo Greedy

Con el objeto de lograr optimizar el proceso de ensamblaje de-novo, a finales de la década anterior se proponen ensambladores genómicos basados en la

metodología Greedy SSAKE (Warren, R. L. et al., 2007), SHARCGS (Dohm, J. C. et al., 2007), VCAKE (Jeck, W. R. et al., 2007) y QSRA (Bryant, D. W., Wong, W. K., & Mockler, T. C., 2009). Al igual que el algoritmo, estos ensambladores se enfocaban en optimizar los procesos locales con la esperanza de alcanzar una optimización global.

Computación Heterogénea en los ensambladores genómicos de-novo

Figura 1.7 Paralelización de la construcción de grafos en el ensamble genómico. Fuente: Autor.

A partir del año 2011 se comienzan a proponer ensambladores genómicos de-novo que utilizan plataformas heterogéneas, principalmente CPU/GPU (figura 1.7):

- GPU-EULER (Mahmood, S. F., & Rangwala, H. 2011) ensamblador basado en caminos de Euler que soporta la representación del grafo De Bruijn en una lista de vértices, una lista de nodos y 2 listas de apuntadores (aristas entrantes y salientes de cada nodo); Usa la librería CUDPP para hacer un estimado de cuanta memoria se requiere para almacenar la información del vértice correspondiente a los nodos de entrada y salida. Sus principales desventajas se evidencian en el mayor tiempo de transferencia de datos, ya que no es riguroso en evitar la sobre transferencia de datos entre host y GPU, y en el mayor uso de memoria (capacidad para menos lecturas) debido al no uso de estructuras de datos simplificadas.

- En el 2013 se propone una estrategia para enviar por tandas las lecturas a procesar en la GPU en el proceso de ensamblaje genómico. El aporte principal se evidencia en la ecuación que permite estimar el número de secuencias a pasar por tanda basado en la memoria disponible de la GPU, el tamaño de las lecturas y la longitud del k-mer (Lu. M, et al., 2013).

- En el mismo año hace su aparición el ensamblador GAGM (Jain, A., Garg, A., & Paul, K. 2013), cuyo principal aporte fue la inclusión de algoritmos de búsqueda binaria para hallar la ubicación de una pareja de k-mers. El propósito de este aporte fue evitar el uso de tablas Hash, debido a la complejidad de su implementación en GPU (causada por la dependencia de sus procesos secuenciales).

- GGAKE (Garg, A., Jain, A., & Paul, K., 2013). Este modelo de ensamblador genómico se basa en la metodología Overlap. Se soporta en la hipótesis de que 2 lecturas que se solapan van a formar un contig; además asume que no hay lecturas, ni k-meros duplicados, lo que lo hace un modelo ideal aún no comprobado en la realidad.

- SWAP Assembler (Meng. et al., 2014). En el siguiente año se propone un ensamblador que postula la construcción del grafo por partes simultáneas mediante una estrategia que permite resolver la interdependencia computacional en la fusión de bordes del grafo que comparten la misma ruta.

- En el mismo 2014 se propone hacer uso de lecturas en formato SeqDB (Formato basado en HDF5) con el objetivo de reducir los requerimientos de memoria. Además involucrar el uso de un filtro Bloom (Bloom, B. H., 1970) para impedir accesos innecesarios a memoria (Georganas, E. et al., 2014). Aunque el principal objetivo se cumple muy bien (reducción de requerimiento de memoria) se descuidan otros aspectos tales como: que el uso de MPI (para distribuir los datos en los nodos) puede causar que mucho tiempo sea invertido en transmisión de los datos a otros nodos (este problema se hace más crítico cuando hay más nodos) y que la conversión de FASTA a SeqDB es un proceso de alta exigencia computacional

- En el año 2015 se propone un ensamblador para datos metagenómicos basado en GPU (Li, D. et al., 2015). Se usan Grafos de De Bruijn Sucintos, (Bowe, A. et al., 2012) los cuales son una forma comprimida de los grafos de De Bruijn, pero suelen ser más difíciles de procesar. Para manejar el ingreso de datos a GPU se envían lecturas por bloques. Adicionalmente incluye una estrategia iterativa que va aumentando el tamaño del K, basándose en que los K pequeños son favorables para filtrar nodos erróneos y los k grandes son útiles para resolver repeticiones. En cada iteración elimina los nodos potencialmente erróneos.

- Entre los avances más recientes sobresale un ensamblador denominado Focus (Warnke-sommer & H. Ali, 2017), el cual propone una metodología de construcción distribuida de grafos de solapamiento

ayudada por meta-datos biológicos que favorece la calidad del proceso distribuido.

- Gerbil (Erbert, Rechner & Müller-Hannemann, 2017). A nivel específico de tratamiento de k-mers (sub-secuencias con las que se representan los nodos y aristas de los grafos) el aporte más reciente ha sido un contador de k-mers basado en tablas hash y particionamiento del conjunto de datos a partir de semillas denominadas minimizers. La herramienta soporta su ejecución en GPUs y se escala al número de éstas disponibles. La paralelización masiva se enfoca específicamente en el paso del conteo. Para favorecer la reducción de latencia por transferencia de datos la tabla hash es llevada a la GPU y se hace una inserción simultánea de tal forma que cada bloque de CUDA se encargue de llevar un paquete de k-mers a la tabla.

1.2. Objetivos

Objetivo General

Crear un modelo de procesamiento paralelo para la construcción de grafos en el ensamblaje de-novo de genomas, utilizando arquitecturas heterogéneas.

Objetivos Específicos

Diseñar un modelo de procesamiento paralelo para la construcción de grafos en el ensamblaje de-novo de genomas, utilizando procesamiento paralelo en arquitecturas heterogéneas. (Fase 1: Diseño)

Desarrollar un núcleo de librerías para la construcción de grafos en el ensamblaje de-novo de genomas, utilizando procesamiento paralelo en arquitecturas heterogéneas. (Fase 2: Implementación)

Evaluar el modelo propuesto mediante la medición del desempeño de una herramienta basada en las librerías desarrolladas. (Fase 3: Evaluación)

Generar proyecciones y prospectivas de uso e innovación (mejoras, optimización y transformación) del modelo propuesto. (Fase 4: Proyección)

1.3. Pregunta problema e hipótesis de la investigación

La pregunta de investigación formulada fue: ¿Que especificaciones debe cumplir un modelo de procesamiento sobre arquitecturas heterogéneas para paralelizar el procesamiento de k-mers en la construcción de grafos para el ensamblaje de-novo de secuencias genómicas de nueva generación?

La hipótesis que se planteó es la siguiente: "Mediante la representación y procesamiento basado en espacios indexados N-dimensionales se obtiene un modelo de paralelización en el tratamiento de k-mers para la construcción de grafos que permite la implementación y ejecución de ensambladores genómicos de-novo en plataformas heterogéneas paralelas."

1.4. Organización de la tesis

Capítulo	Descripción	Fases
1. Introducción	Mediante la introducción se le presenta al lector el contexto de la investigación y los estudios preliminares que justifican los objetivos de la investigación.	Fase 0. Propuesta
2. Evaluando el ensamble genómico de-novo basado en grafos de De Bruijn	En el segundo capítulo se presenta la evaluación que se realizó a los principales ensambladores genómicos de-novo basados en grafos de De Bruijn con el fin de detectar la(s) etapa(s) y/o proceso(s) de mayor complejidad y con mayores requerimientos computacionales, e identificar las metodologías que mejor enfrentan estas dificultades.	Fase 0. Propuesta Fase 1. Diseño
3. El reto de procesar k-mers	De acuerdo a los resultados de la evaluación del capítulo 2, el capítulo 3 se enfoca en evaluar exclusivamente las herramientas que procesan k-mers con el objeto de medir el desempeño de sus diferentes metodologías, haciendo especial seguimiento a aquellas que realizan particionamiento en disco.	Fase 1. Diseño
4. Minimizers para facilitar el procesamiento de k-mers	Basado en los resultados del capítulo anterior, el capítulo 4 presenta un análisis detallado de las metodologías de procesamiento de k-mers soportadas en semillas tipo minimizer.	Fase 1. Diseño
5. Computación heterogénea	El capítulo 5 hace una introducción y presenta los fundamentos de la computación heterogénea incluyendo los dos principales estándares y/o plataformas para la programación paralela sobre dispositivos many core: OpenCL y CUDA.	Fase 1. Diseño Fase 2. Implementación Fase 3. Evaluación
6. Mejorando el desempeño del procesamiento paralelo sobre plataformas many core	Este capítulo documenta el trabajo que se realizó alrededor de la búsqueda, análisis y comparación de metodologías para mejorar el procesamiento paralelos sobre plataformas many core.	Fase 1. Diseño Fase 2. Implementación Fase 3. Evaluación
7. Modelo de procesamiento	Este capítulo representa el corazón de la investigación, en él se presenta el modelo	Fase 1. Diseño

paralelo de k-mers sobre plataformas many core	de procesamiento paralelo de k-mers sobre plataformas many core, aplicando los resultados expuestos en los capítulos 3,4 y 6.	
8. Implementación y evaluación	En este capítulo se implementa el modelo mediante programación heterogénea que involucra un código host y dos kernels.	Fase 2. Implementación Fase 3. Evaluación
9. Conclusiones, principales aportes, recomendaciones y futuros trabajos	En el capítulo 9 se presentan las conclusiones, los aportes y los futuros trabajos de investigación.	Fase 4. Proyecciones

Tabla 1.1 Organización de la tesis

2. EVALUANDO EL ENSAMBLAJE GENÓMICO DE-NOVO BASADO EN GRAFOS DE DE BRUIJN

De acuerdo a lo evidenciado en el capítulo introductorio, el proceso de ensamblaje constituye la etapa de procesamiento bioinfomático de secuenciación de nueva generación con mayor exigencia de recurso computacional. En este capítulo se profundiza en la etapa de ensamblaje y se presentan los resultados del estudio realizado por el autor con el fin de evaluar el desempeño computacional de los principales ensambladores genómicos de-novo basados en grafos de De bruijn e identificar las etapas y sub etapas con mayor exigencia computacional y de evaluar y comparar el desempeño de los diferentes tipos de algoritmos y estructuras de datos empleadas por estos ensambladores.

2.1. Que es el ensamblaje genómico?

El ensamblaje es el proceso de reconstrucción de secuencias genómicas muy grandes a partir de subsecuencias aleatorias llamadas lecturas; dichas lecturas resultan del proceso de secuenciación. Durante el proceso de ensamblaje se agrupan las lecturas en contigs y estos a su vez se agrupan en scaffolds que representan regiones largas de ADN (El-Metwally, 2014). Para llevar a cabo el proceso de ensamblaje existen tres estrategias: La primera se llama “ab-initio” y usa un genoma de referencia para el ordenamiento de las regiones de ADN, la segunda se llama “de novo” la cual trabaja sin genoma de referencia y la tercera es una opción híbrida que combina las anteriores (Miller. et al., 2010). Este capítulo se limita exclusivamente al ensamblaje de novo, debido a que los objetivos de ésta investigación están centrados en este tipo de ensamblaje.

2.2. Cómo se usan los grafos de De Bruijn en el ensamblaje genómico de-novo?

Teoría de grafos de De Bruijn

Los grafos De Bruijn fueron propuestos por el matemático Nicolaas De Bruijn para resolver el problema de encontrar la secuencia cíclica más corta de letras que contenga todas las posibles palabras de tamaño “k” (k-mer) dado un alfabeto. Por ejemplo para un el alfabeto: “A,B” los posibles k-mers con tamaño k=2 serian: AA, AB, BA y BB. La propuesta de De Bruijn fue como primer paso crear un grafo en el cual los nodos corresponden a los posibles (k-1)mers y las aristas corresponden a los k-mers conectando dos (k-1)mers que representan el sufijo y el prefijo de la palabra, y como segundo paso hallar un ciclo euleriano (Compeau. et al., 2011).

Ensamblaje con enfoque de ciclo Hamiltoniano

Este enfoque toma las lecturas, las divide en segmento de tamaño "k" (k-mers) y construye un grafo en el cual los nodos corresponden a los k-mers, conectando los nodos en los que el sufijo del primero sea el prefijo del segundo como se muestra en la figura 2.1. Así se obtiene un grafo direccionado en el cual se busca un ciclo hamiltoniano (un camino que recorra todos los nodos una única vez) (Compeau & Pevzner, 2010).

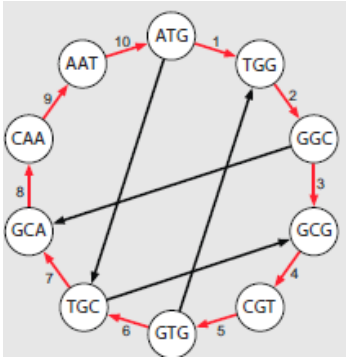


Figura 2.1 Ciclo Hamiltoniano para la secuencia "ATGGCGTGCAATG" (Compeau. et al., 2011)

Ensamblaje con enfoque de ciclo Euleriano

Este enfoque toma las lecturas, las divide en segmentos de tamaño "k" (k-mers) y construye un grafo en el cual los nodos corresponden a los (k-1)mers y las aristas a los k-mers, conectando los nodos en los que el primer nodo corresponde al prefijo del k-mer y el segundo nodo corresponde al sufijo como se muestra en la figura 2.2. Así se obtiene un grafo direccionado y a diferencia del enfoque Hamiltoniano, se busca un ciclo Euleriano (un camino que recorra todas las aristas una única vez) (Idury & Waterman, 1995).

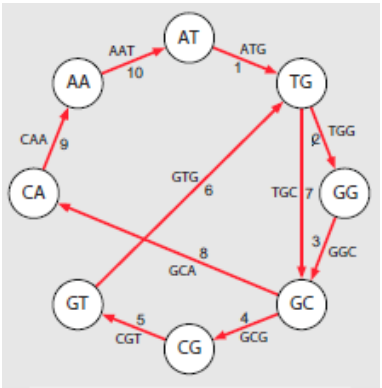


Figura 2.2 Ciclo Euleriano para la secuencia "ATGGCGTGCAATG" (Compeau. et al., 2011)

Este método es usado actualmente por los ensambladores de lecturas cortas ya que al igual que el enfoque hamiltoniano permite llegar a la misma construcción del genoma pero a un costo computacional mucho menor.

2.3. Ensambladores genómicos de-novo basados en grafos de De Bruijn

A continuación se hace una breve descripción de los principales (recientes y más citados) ensambladores genómicos de lecturas cortas basados en grafos de De Bruijn:

ABySS: Fue uno de los primero ensambladores genómico capaz de representar grandes grafos de De Bruijn. Usa una tabla de direccionamiento hash que almacena los k-mers del grafo en sus llaves. Las aristas pueden ser inferidas a partir de los nodos, lo que evita su almacenamiento. Para almacenar cada k-mers se emplean 2K bits, más un adicional de 43 bits de datos asociados. Por consiguiente, el espacio total de memoria usado para almacenar la estructura de datos del grafo de De Bruijn es (l-1 (2k + 64)) bits por k-mer, donde l es el factor de carga de la tabla hash (normalmente 0.8). En la actualidad ABySS no representa una opción eficiente de ensamblaje genómico debido al alto requerimiento de memoria para almacenar la tabla hash distribuida que representa el grafo: para almacenar el grafo correspondiente al conjunto de lecturas de un genoma humano (HapMap: NA18507) se requiere 336GB de memoria (Muggli, et. al., 2017).

Velvet: Es un ensamblador de-novo de lecturas cortas. El algoritmo de Velvet cuenta principalmente con dos etapas: Producción de hash (hashing) y construcción del grafo; estas etapas son realizadas por velveth y velvetg respectivamente. Velveth lee los archivos de secuencias y construye un diccionario con todas las palabras de longitud k, junto con la definición de los alineamientos locales exactos entre esas lecturas. Posteriormente, velvetg lee esos alineamientos, construye un grafo de De Bruijn, remueve los errores, simplifica el grafo y resuelve las repeticiones con base en los parámetros suministrados por los usuarios. Velvet está implementado en C++ y hace uso de la librería OpenMP con el objeto de que algunas etapas del ensamblaje se puedan correr en paralelo en un mismo nodo computacional.

SOAPdenovo2: Es un novedoso método de ensamblaje de lecturas cortas que puede construir un borrador de ensamblado de novo para genomas de tamaño humano. Este algoritmo cuenta con los siguientes pasos: Construcción de grafos, ensamblaje de contigs, mapeado de lecturas paired-end, construcción de scaffold y cierre de brechas. SOAPdenovo2 está implementado en C++ y para algunas de sus etapas de procesamiento hace uso de varios hilos.

Minia: Es un ensamblador de lecturas cortas basado en grafos de De Bruijn, capaz de ensamblar un genoma humano en un computador de escritorio en tan solo un día. La salida de Minia es un conjunto de contigs. Las principales etapas del algoritmo de Minia son: el conteo de k-mers, la construcción del grafo y el recorrido o simplificación del mismo. Para la etapa de conteo de k-mers, minia hace uso de filtros Bloom con el fin de reducir el uso de memoria RAM. Minia está implementado en C++. Cabe notar que a diferencia de Velvet y Abyss, Minia no hace uso de la información paired-end disponible. (Chikhi, R., & Rizk, G., 2012; Salikhov, K., Sacomoto, G., & Kucherov, G., 2013)

EPGA2: Es un ensamblador genómico orientado al uso eficiente de la memoria. Para cumplir este objetivo hace uso de BLESS (Heo et al, 2015) para corrección de errores en las lecturas, DSK (Rizk, Lavenier & Chikhi, 2013) para el conteo de k-mers, y BCALM (Chikhi et al, 2014) para simplificar nodos en el grafo de De Bruijn. Una vez simplificados los nodos y generados los contigs, EPGA2 hace la unión de contigs de manera paralela (Luo et al, 2015).

2.4. Evaluación de desempeño computacional de los principales ensambladores genómicos de-novo basados en grafos de De Bruijn

Como etapa preliminar de esta tesis se realizó una evaluación de desempeño computacional de los principales ensambladores genómicos de-novo basados en grafos de De bruijn con el objeto de identificar las etapas y sub etapas con mayor exigencia computacional y de evaluar y comparar el desempeño de los diferentes tipos de algoritmos y estructuras de datos empleadas por estos ensambladores (Vera-Parra, Pérez-Castillo & Rojas-Quintero, 2015).

Para la evaluación se empleó un conjunto de datos correspondiente a 64587949 lecturas cortas (101bp) paired-end del cromosoma 14 de Homo Sapiens. La evaluación se ejecutó en un equipo de cómputo con sistema operativo Debian Wheezy (AMD64), procesador Intel(R) Xeon(R) cpu E7450 @ 2,40GHz, 24 cores. RAM de 64GB y disco duro HDD de 160GB. Los ensambladores evaluados fueron los mismos descriptos en la sección anterior. Los parámetros medidos fueron la transferencia I/O de datos a disco, el uso de la memoria, el número de cores usados y el tiempo de ejecución (figura 2.3 – 2.8 y tabla 2.1).

Desempeño de Velvet

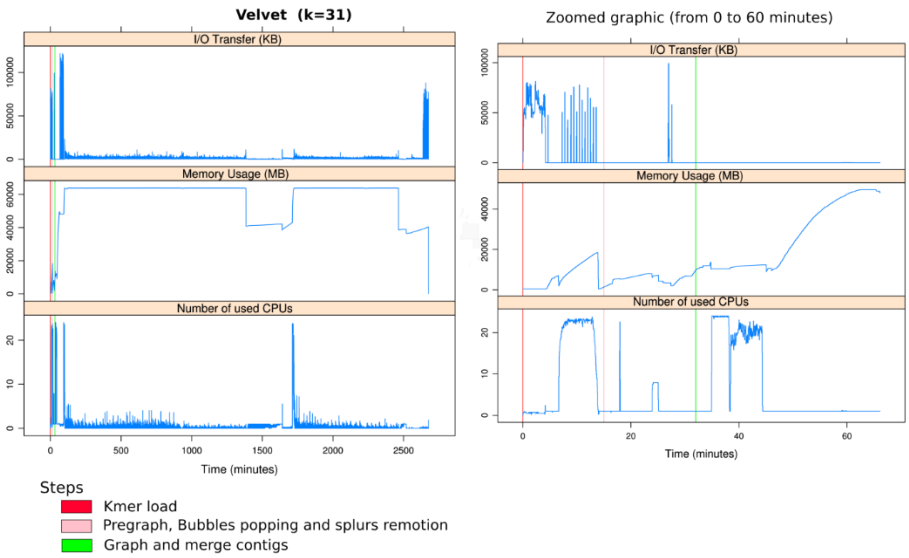


Figura 2.3 Desempeño de Velvet (k=31).Zoom de minuto 1 a 60. Fuente: Autor.

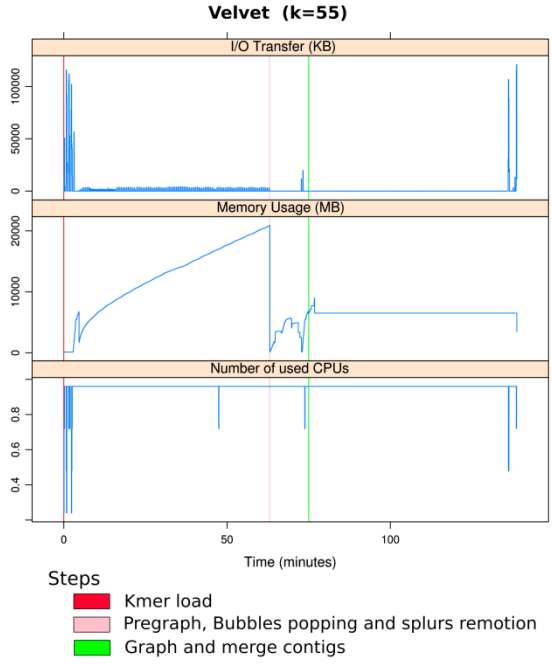


Figura 2.4 Desempeño de Velvet (k=55). Fuente: Autor.

Desempeño de ABySS

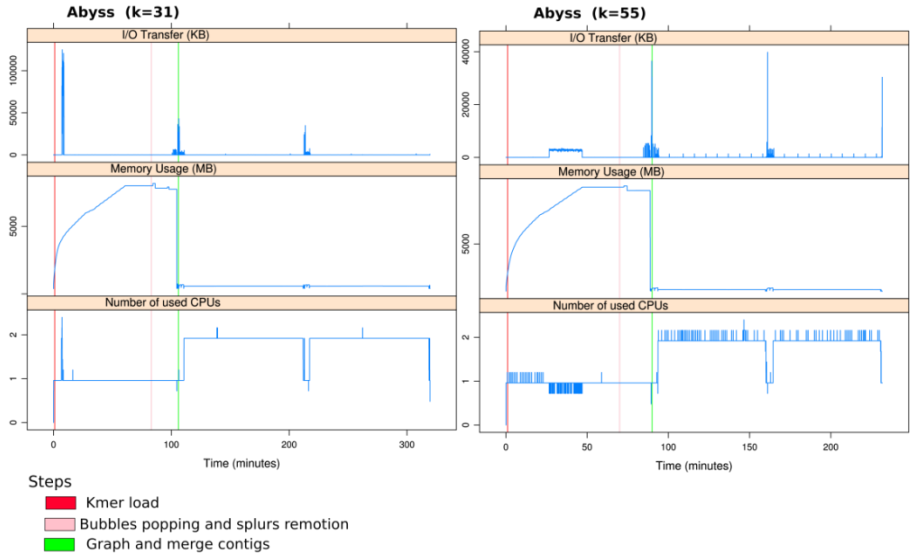


Figura 2.5 Desempeño de ABySS (k=31 y k=55). Fuente: Autor.

Desempeño de Minia

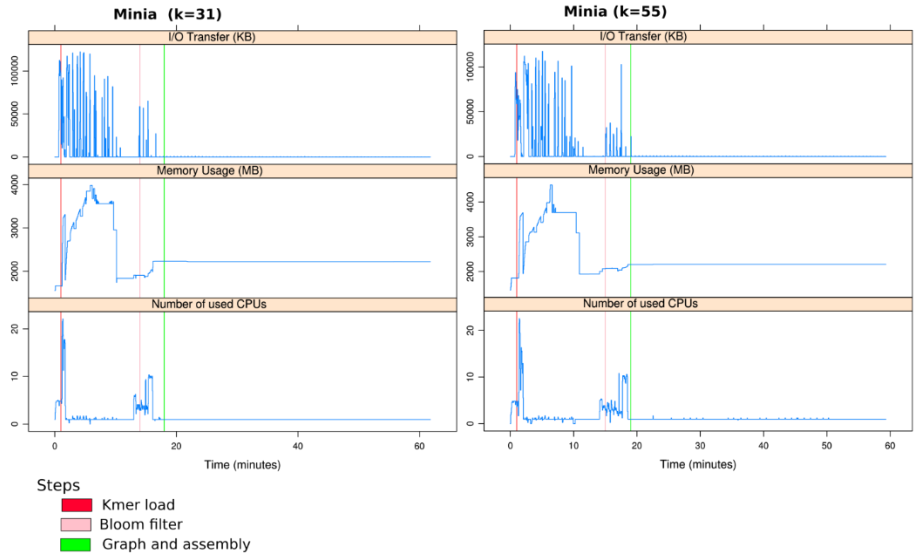


Figura 2.6 Desempeño de Minia (k=31 y k=55). Fuente: Autor.

Desempeño de SOAPdenovo2 por etapas

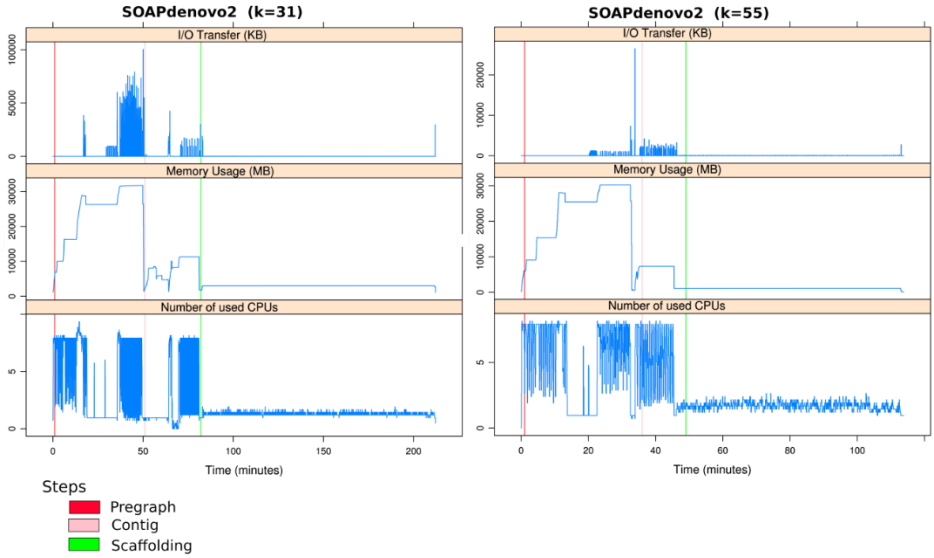


Figura 2.7 Desempeño de SOAPdenovo2 (k=31 y k=55). Fuente: Autor.

Desempeño de EPGA2 por etapas

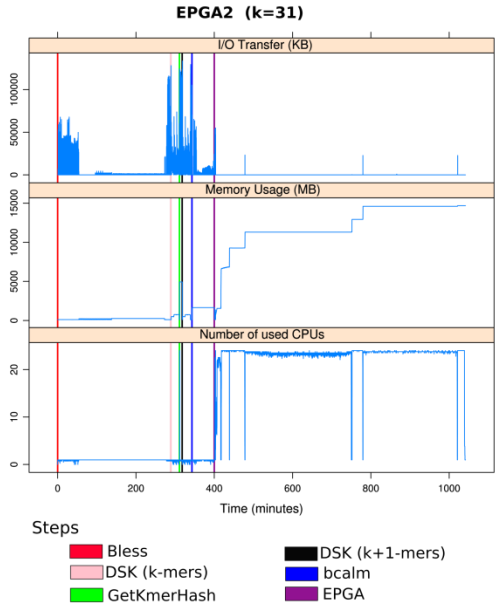


Figura 2.8 Desempeño de EPGA2 (k=31). Fuente: Autor.

Comparación entre ensambladores

		RAM (MB)		Número de CPUs				I/O (KB/S)				Tiempo (Mins)	
		k=31	k=55	k=31		k=55		k=31		k=55			
		Max	Max	Δ	Max	Δ	Max	Δ	Max	Δ	Max	k=31	k=55
Velvet	Kmer load	18458	20893	11	24	1	1	7000	25000	600	37000	15	63
	Pregraph, bubbles popping and splurs remotion	9740	6781	2	22	1	1	250	30000	300	41000	17	12
	Graph and contig merging	63916	6781	2	24	1	1	400	40000	100	7000	2618	70
Abyss	Kmer load	8000	9230	1	3	1	1	450	40000	300	1300	83	70
	Bubbles popping and splurs remotion	8170	9315	1	1	1	1	80	12000	200	13500	23	20
	Graph and merge contigs	574	1733	1	2	2	2	90	13000	60	14000	224	155
Minia	Kmer load	3984	4492	2	20	2	23	5500	40000	5500	39000	15	15
	Bloom filter	2231	2201	3	11	4	10	700	22000	2300	37000	4	4
	Graph and assembly	2231	2206	1	1	1	2	270	33	60	8000	43	40
SOAP Denovo 2	Pregraph	31739	30274	4	10	4	8	1300	33000	100	12500	51	36
	Contig generation	11324	7328	2	9	4	8	300	13000	200	1400	31	13
	Scaffolding	3049	1102	1	2	2	3	40	10000	20	1250	133	65
EPGA2	Bless	258	N/A	1	1	N/A	N/A	800	35000	N/A	N/A	289	N/A
	DSK	4924	N/A	1	1	N/A	N/A	7500	43000	N/A	N/A	54	N/A
	bcalm	1651	N/A	1	1	N/A	N/A	1200	25000	N/A	N/A	57	N/A
	EPGA	14671	N/A	23	24	N/A	N/A	40	20000	N/A	N/A	653	N/A

Tabla 2.1 Evaluación de desempeño por etapas de los principales ensambladores genómicos de-novo basados en grafos de De Bruijn.

Análisis de los resultados

El análisis realizado en este capítulo se enfoca exclusivamente en el tiempo de ejecución y el uso eficiente de la memoria, con el objeto de detectar las sub-etapas que exigen mayores requerimientos e identificar las metodologías que presentan mejor desempeño en dichas métricas. (Si se desea un análisis en otro aspecto se puede remitir al artículo resultado de esta evaluación (Vera-Parra, Pérez-Castillo & Rojas-Quintero, 2015)).

Todas las herramientas presentaron una notoria mayor exigencia de memoria en los procesos asociados a la obtención y el tratamiento de k-mers previo y durante la construcción del grafo. Los tiempos de ejecución por etapa fueron un poco más balanceados presentando en algunos casos mayor exigencia las etapas de reducción y recorrido del grafo.

Si se compara en general el uso de la memoria y los tiempos de ejecución de las herramientas entre sí, la indiscutible ganadora en tiempo de ejecución es Minia (tomó un tiempo menor que 1 hora a diferencia del resto que tomaron entre 3 y 17 horas) y en cuanto a uso eficiente de la memoria hay una especie de empate entre EPGA2, y Minia.

Al analizar las metodologías usadas por cada una de las herramientas se evidencia que las dos herramientas sobresalientes (Minia y EPGA2) presentan algo en común, las dos utilizan técnicas que permiten el procesamiento particionado (dividir el conjunto de datos de tal forma que se pueda procesar cada archivo por separado). La etapa de conteo de k-mers de Minia es realizada por DSK (Rizk, Lavenier & Chikhi, 2013), un contador que particiona el conjunto de datos de tal forma que permite el procesamiento por archivos con tamaño adecuados a los recursos de memoria. Por su parte EPGA2 hace uso de BCALM, una herramienta que representa de forma compacta el grafo involucrando una técnica de particionamiento del conjunto de k-mers basada en minimizers por frecuencia.

2.5. Conclusiones

De acuerdo al análisis anterior se puede concluir que los procesos asociados a la obtención y al tratamiento de k-mers previo y durante la construcción del grafo son tareas intensivas con muy alta exigencia de memoria, lo que los convierte en cuellos de botella en el ensamblaje genómico de-novo. Adicionalmente se pudo notar que aquellas herramientas que incluyeron en algunas de sus etapas algoritmos de procesamiento particionado presentaron un mejor desempeño, lo que convierte a éstas estrategias en un posible horizonte a seguir para superar los cuellos de botella detectados.

Teniendo en cuenta estas conclusiones, el siguiente capítulo se enfocará en desglosar el proceso de ensamblaje para analizar y evaluar específicamente las estructuras de datos y las metodologías usadas en procesos fundamentales en el tratamiento de k-mers (tales como su obtención, representación, búsqueda, comparación, conteo, eliminación, entre otras), haciendo un seguimiento especial a aquellas metodologías basadas en el procesamiento particionado con el objeto de medir su desempeño cuando se aplican específicamente en este tipo de procesos.

3. EL RETO DE PROCESAR K-MERS

En este capítulo se presenta un análisis teórico de la dificultad de procesar k-mers y un análisis práctico mediante los resultados de un estudio desarrolado por el autor con el propósito de evaluar las estructuras de datos y algoritmos usados en el procesamiento de k-mers. La evaluación práctica se realiza mediante herramientas de conteo que involucren las principales técnicas de procesamiento de k-mers.

3.1. ¿Qué es un k-mer?

Los k-mers de una lectura son todas las posibles sub-secuencias de longitud k que se pueden obtener de dicha lectura. En la figura 3.1 se puede observar la obtención de los k-mers de una lectura.

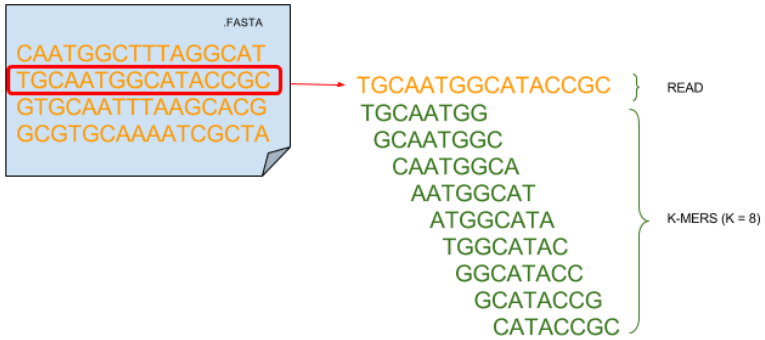


Figura 3.1 K-mers de una lectura. Fuente: Autor.

La cantidad de k-mers obtenidos de una lectura dependerá de la longitud del k-mer y de la longitud de la lectura. Si se tiene una lectura con longitud L se podrán obtener L-K+1 sub-secuencias de longitud K (k-mers). En la figura 3.1 se observa que de una lectura de 16 bases se obtuvieron 9 k-mers de longitud 8.

3.2. ¿Cuál es la importancia de los k-mers?

En el capítulo anterior se describió la forma como en la última década el surgimiento de técnicas basadas en grafos de De Bruijn ha permitido en cierto grado afrontar la problemática que representa la elevada intensidad computacional y la complejidad exigida para realizar un ensamblaje sin un genoma de referencia (de-novo). Actualmente, los ensambladores genómicos

con mayor uso están basados en este enfoque, tales como Velvet, ALLPATHS, ABySS, SOAPdenovo2, MINIA, EPGA2.

Todas las herramientas de ensamblaje de lecturas que se basan en la construcción y recorrido de grafos de De Bruijn para reconstruir un genoma, se fundamentan estrictamente en el procesamiento de k-mers para su funcionamiento, debido a que los nodos o las aristas de dichos grafos son representados por los k-mers generados a partir de las lecturas.

Como consecuencia a lo anterior, la obtención, el almacenamiento y el procesamiento de k-mers se han convertido en una alta prioridad bioinformática especialmente en el campo del ensamblaje de lecturas sin genoma de referencia.

3.3. Por qué es difícil procesar k-mers?

Como se mencionaba en los párrafos anterior, los k-mers son el soporte sobre el cual se fundamentan los métodos de ensamblaje de-novo basados en grafos de De Bruijn, por tal motivo la mayoría de retos computacionales referentes al ensamblaje de-novo se enfocan en la dificultad del procesamiento de k-mers específicamente por la alta demanda de memoria. Este alto requerimiento de memoria se sustenta específicamente por dos aspectos:

- El gran volumen de datos debido a la redundancia: De acuerdo al concepto de k-mers descrito en un párrafo anterior, de una sola lectura de longitud L se obtendrán L-k+1 k-mers; en otras palabras, por cada lectura se pasará de tener L caracteres a tener (L-k+1)*k caracteres; los caracteres adicionales no son caracteres nuevos, son sólo producto de la alta redundancia de datos entre k-mers, por ejemplo dos k-mers contiguos de una lectura, son sub-secuencias con redundancia de k-1 caracteres. En la figura 3.2 se puede observar un ejemplo del tamaño del archivo que se generaría al obtener los k-mers de un conjunto de lecturas: considerando que se usan 2 bits para representar cada base y que no se emplea ninguna técnica de compresión, se pasaría de tener un archivo de 2GB a uno de 100GB.

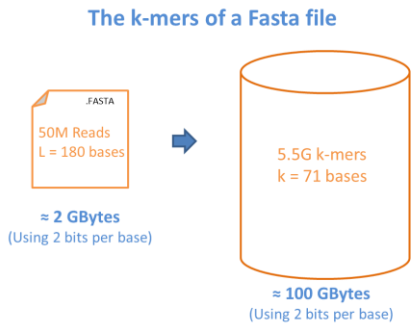


Figura 3.2 Ejemplo de la generación de un gran volumen de datos al obtener los k-mers de un conjunto de lecturas. Fuente: Autor.

- Imposibilidad o dificultad del procesamiento particionado: La mayoría de tareas con k-mers exigen la carga en memoria de todo el conjunto de datos debido a la imposibilidad o alta dificultad del tratamiento por partes. Por ejemplo si se divide el conjunto de datos de acuerdo a una función hash de los k-mers, es muy probable que dos k-mers contiguos de una lectura queden en particiones (archivos) diferentes e imposibiliten tareas que impliquen encontrar k-mer contiguos o exijan un pos-procesamiento muy complejo de fusión de particiones.

3.4. Evaluación de las estructuras de datos para representar k-mers y metodologías para su procesamiento

En los últimos años se han propuesto varias técnicas de estructuras de datos buscando obtener una representación eficiente de k-mers que facilite su almacenamiento (tanto en disco como en memoria) y su agrupación, de tal forma que facilite el procesamiento particionado. El propósito de esta evaluación es medir el desempeño de dichas estructuras y estrategias en procesos fundamentales en el tratamiento de k-mers, haciendo un especial seguimiento a aquellas técnicas que habilitan el procesamiento particionado.

Para cumplir con este propósito se buscó un tipo de herramienta que fuera referente en el campo de procesamiento de k-mers y que involucrara la mayor cantidad de procesos fundamentales. Por este motivo esta investigación optó por evaluar el desempeño de las principales herramientas contadoras de k-mers con el objeto de medir los requerimientos y el desempeño de sus estructuras de datos y algoritmos (Pérez, N., Gutierrez, M., & Vera, N., 2016).

Herramientas contadoras de k-mers evaluadas

Las herramientas a evaluar son: – DSK (Rizk, Lavenier & Chikhi, 2013), particiona los datos en disco y los carga por partes a memoria. – KAnalyze (Audano, & Vannberg, 2014), divide la carga de k-mers a memoria. – MSPKmerCounter (Li, 2015), utiliza MSP (Minimum Substring Partitioning)

el cual usa minimizers como criterio para particionar. – KMC2 (Deorowicz et. al., 2015), utiliza una variación del método MSP, llamada Signatures (un conjunto de minimizers con características enfocadas en lograr un particionamiento más homogéneo). –BFCounter (Melsted, & Pritchard, 2011), utiliza una estructura de datos probabilísticas llamada filtro Bloom. – KHMer (Zhang, Pell, Canino-Koning, Howe, & Brown, 2014), utiliza una estructura de datos probabilística llamada Count-Min Sketch. – Turtle (Roy, Bhattacharya, & Schliep, 2014), utiliza el filtro Bloom junto con una técnica llamada Sort and Compress. – Tallymer (Kurtz, Narechania, Stein, & Ware, 2008), utiliza árboles lcp-interval. – Jellyfish (Marçais & Kingsford, 2011), utiliza una tabla hash con un enfoque lock-free para el acceso concurrente A continuación se presenta cada herramienta con la descripción de su algoritmo de conteo:

DSK: Método para conteo de k-mers que optimiza el uso de la memoria RAM por medio de la partición de lecturas en conjuntos pequeños los cuales son guardados en disco y cargados a memoria en pequeñas cantidades. Este algoritmo se divide en 3 pasos: el primero calcula el número de k-mers, el número de iteraciones para recorrer los k-mers y el número de particiones para guardar en disco. En el segundo paso se recorren todos los k-mers y por medio de una función hash se distribuye una parte de los k-mers en las particiones. El tercer paso consiste en cargar en una tabla hash los k-mers de cada partición e irlos contando, ocupando en memoria únicamente una partición a la vez. Finalmente se repiten los 3 pasos hasta que se hayan insertado todos los k-mers.

KAnalyze: Este algoritmo divide las secuencias en subconjuntos para cagarlos a memoria por partes pequeñas, lo cual permite un consumo mucho menor de memoria. Este método se divide en dos pasos: división (Split) y unión (Merge). El primer paso se encarga de tomar los k-mers y los carga en memoria (en un array) hasta que esté llena, luego el array es ordenado, se cuentan los k-mers y se escriben en disco. El segundo paso lee los archivos de conteo y los carga a memoria con un buffer para no ocupar toda la memoria disponible y los va uniendo en el archivo final de conteo, el cual queda ordenado y en disco.

MSPKmerCounter: Este método se basa en una técnica llamada minimum substring partitioning (MSP) (Li et al., 2013) la cual divide las lecturas en super k-mers, que son cadenas que agrupan k-mers que poseen el mismo minimizer; siendo este minimizer la sub-cadena (de longitud fija menor a “k”) con el menor peso lexicográfico de un k-mer. El almacenamiento de los super k-mers se realiza particionando el disco, de tal forma que aquellos super k-mers que posean el mismo minimizer serán almacenados en la misma partición. A continuación del almacenamiento particionado de los súper-kmers viene una etapa que se encarga de leer estas particiones, dividir los súper-kmers y obtener los k-mers para insertarlos en una tabla hash, hacer el conteo de cada inserción y finalmente guardar el conteo en disco.

KMC 2: Este método utiliza minimizers que cumplan con ciertas características, para reducir la cantidad de éstos y solucionar algunos problemas presentes al utilizarlos (por ejemplo la gran cantidad de minimizers que contienen la cadena AA). Los minimizers seleccionados deben cumplir las siguientes reglas: no empezar con AAA, no empezar con ACA, no contener AA en ningún lado (excepto en el inicio). A los minimizers que cumplen con estas características se les llama Signatures. El procesamiento de los k-mers se divide en dos fases: la distribución y el ordenado. La primera fase carga las lecturas para encontrar las regiones (super k-mers) que tengan la misma Signature y guardarlas en una misma partición de disco identificada con dicha Signature. La segunda fase lee cada archivo, extrae los (k, x)-mers de los super k-mers, los ordena, calcula las estadísticas para los k-mers y finalmente guarda la base de datos de los k-mers (en forma binaria compacta).

BFCounter: Método para identificar, contar y almacenar únicamente los k-mers que se repiten más de una vez en un conjunto de datos de secuencias genómicas. Hace uso de la estructura de datos probabilística llamada filtro Bloom, la cual almacena implícitamente todos los k-mers observados con una gran reducción del espacio en memoria (solo se usa un arreglo: la existencia de un k-mer se registra alterando algunos bits de dicho arreglo, de acuerdo a la salida de unas funciones hash). Este método cuenta con tres pasos: el primero recorre todos los k-mers y utiliza el filtro Bloom para detectar los k-mers vistos más de una vez. El segundo paso se encarga de hacer el conteo de los k-mers anteriormente filtrados. Y finalmente el último paso se encarga de limpiar los posibles falsos positivos que pasaron el filtro.

Khmer: Este método utiliza una estructura de datos probabilísticas llamada Count-Min Sketch (Cormode & Muthukrishnan, 2005), la cual se basa en un filtro Bloom y el uso de múltiples tablas hash para detectar los k-mers que se repiten más de 1 vez y contarlos. Este método se divide en 3 pasos: primero crea una cantidad determinada de tablas hash (según la cantidad máxima de falsos positivos deseados). El segundo paso verifica la repetición de k-mers con el filtro Bloom. Finalmente en el tercer paso se insertan los k-mers y se aumenta el conteo.

Turtle: Es un método de conteo de k-mers basado en un filtro Bloom para detectar los k-mers que se repiten más de 1 vez, de forma similar como lo hace BFCounter. La novedad de Turtle se enfoca en el uso de un método llamado sort and compress (SAC) para hacer el conteo de los k-mers de forma iterativa, de tal manera que en cada iteración se carguen los k-mers (ya filtrados) que quepan en la memoria disponible (o un umbral definido) para proceder a organizarlos y compactar los que sean iguales (haciendo el conteo) tanto de la iteración actual como de las anteriores; así se finaliza una iteración y se pasa a una nueva que usará la memoria liberada con la compresión para cargar otro paquete de k-mers. Finalmente resulta un array con todos los k-mers ordenados y contados.

Tallymer: Este método hace uso de enhanced suffix array y de árboles tipo lcp-interval (Abouelhoda, Kurtz, & Ohlebusch, 2004) para guardar implícitamente el número de ocurrencias de las subcadenas de una secuencia. Este método divide la secuencia en secciones más pequeñas (que no se solapan) para que puedan ser procesadas por un equipo con pocos recursos en memoria RAM. Luego guarda estas secuencias en un enhanced suffix array y a partir de este array construye el árbol lcp-interval. Luego se hace un conteo en cada una de estas secciones a partir del árbol.

Jellyfish: Este método hace uso de una tabla hash y un enfoque lock-free, lo cual permite el acceso y actualización de la tabla por varios hilos al mismo tiempo, con una técnica llamada lock free-queues basada en la operación compare-and-swap (CAS) (Ladan-Mozes & Shavit, 2004). El algoritmo se divide en dos pasos: el primero se encarga de buscar el k-mer y obtener su llave (si ya ha sido insertado) o insertar el k-mer en la tabla hash con una estrategia de reprobado si no existe previamente; en caso de que se genere una colisión, el algoritmo vuelve a intentar insertar el k-mer una cantidad de veces indicada por max_reprobe. El segundo paso se encarga de aumentar la cuenta del k-mer, igualmente con una operación CAS.

Metodología de la evaluación

La evaluación se realiza empleando un conjunto de gran tamaño con lecturas cortas "paired-end" pertenecientes al cromosoma 14 del humano, el cual contiene 36.504.800 lecturas con un promedio de 101 pares de bases por lectura. Los parámetros medidos son: tiempo de ejecución, consumo de memoria RAM, número y porcentaje de cores usados y accesos de lectura y escritura al disco (Figuras 3.3 – 3.6). Todas las medidas se hicieron para dos longitudes de k-mers, k=31 y k=55. El equipo de cómputo sobre el cual se ejecutó esta evaluación contó con las siguientes características: Procesador Intel(R) Xeon(R) cpu E7450 @ 2,40GHz, 24 cores. RAM: 64GB. Disco duro: 160GB HDD.

Resultados y análisis

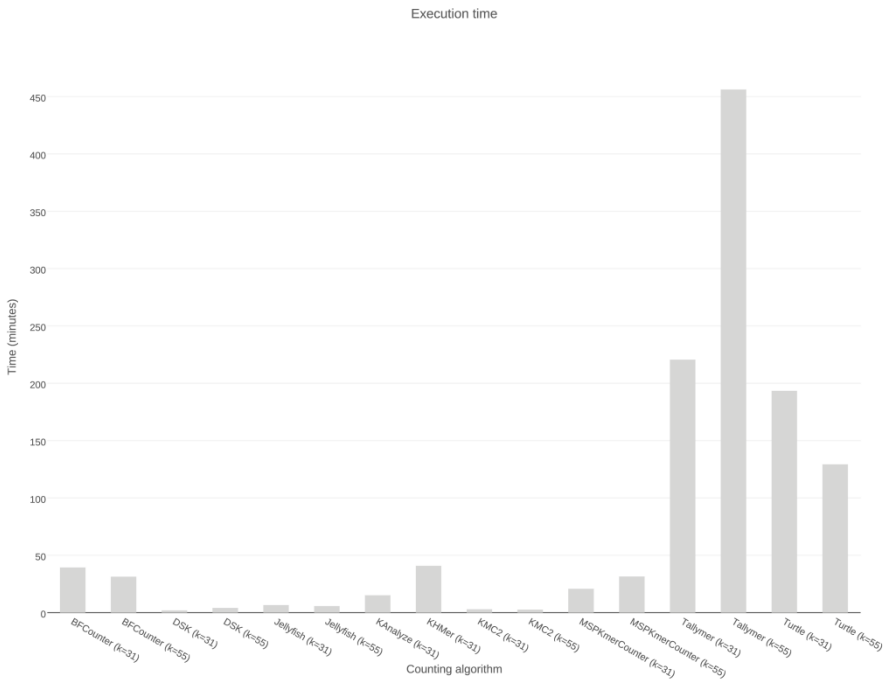


Figura 3.3 Tiempo de ejecución de las herramientas contadoras de k-mers. Fuente: Autor.

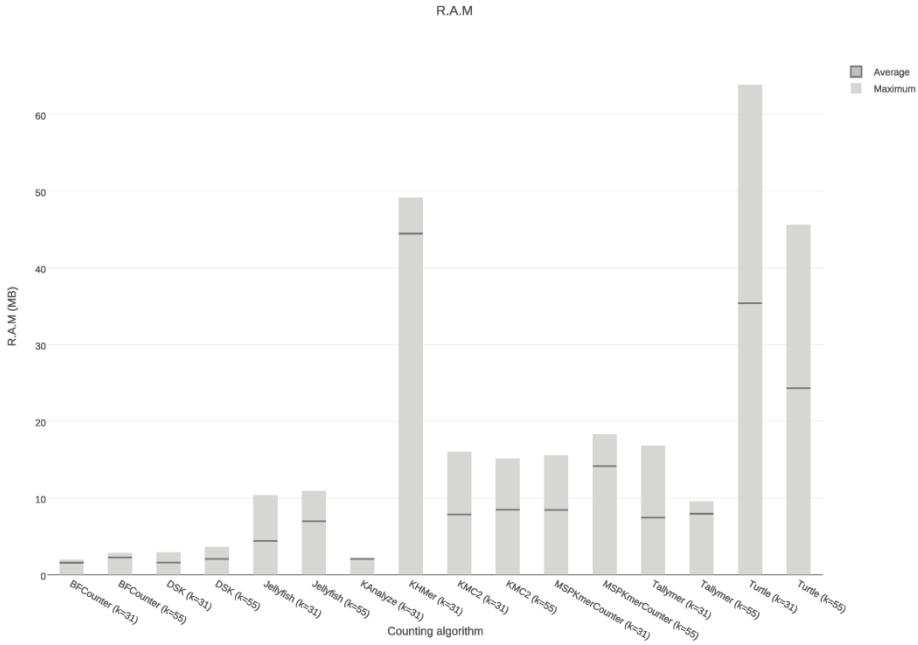


Figura 3.4 Uso de la RAM de las herramientas contadoras de k-mers. Fuente: Autor.

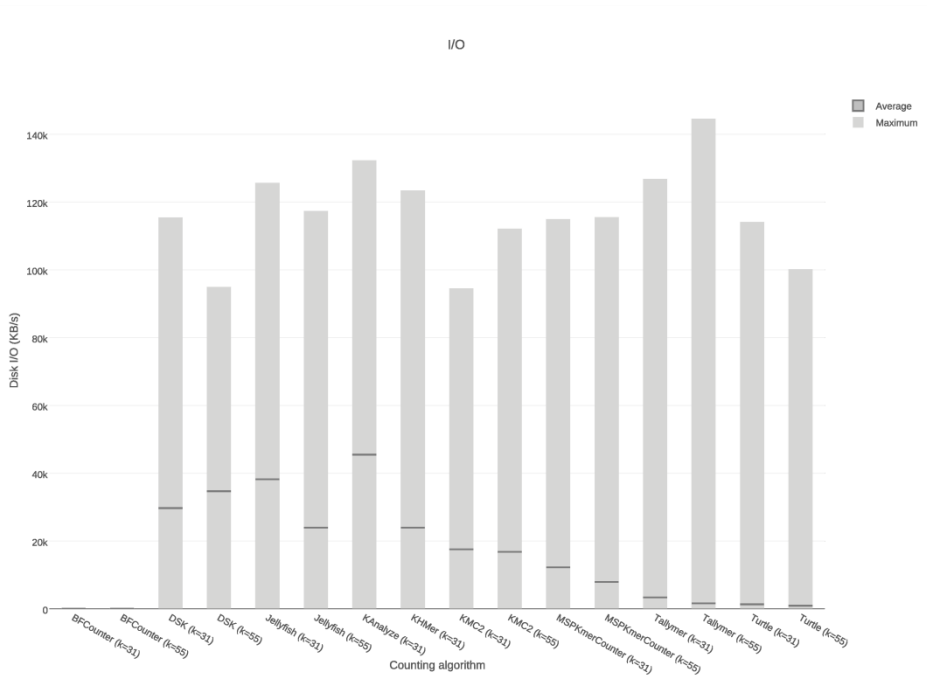


Figura 3.5 Transferencia I/O de datos al disco de las herramientas contadoras de k-mers. Fuente: Autor.

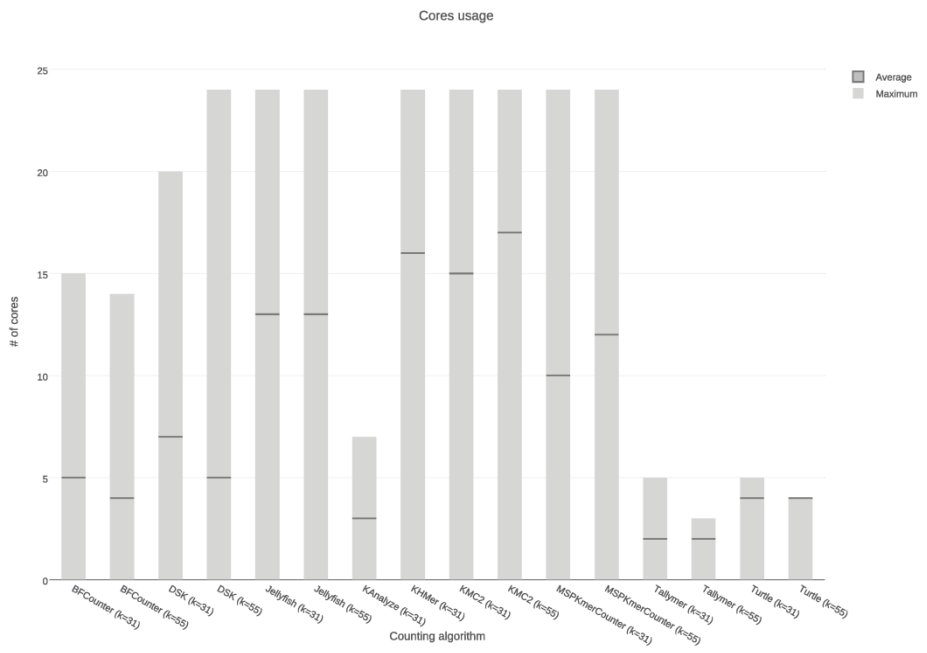


Figura 3.6 Número de cores usados por las herramientas contadoras de k-mers. Fuente: Autor.

Herramientas basadas en Particionamiento en disco: Las herramientas basadas en particionamiento en disco evaluadas fueron: DSK, KMC2, MSPKCounter y KAnalyze. Se evidenció que las herramientas que utilizan este enfoque fueron las que tardaron mucho menos en la ejecución del algoritmo de conteo. Por otro lado, esta técnica tuvo acceso a disco (operaciones de I/O) de una manera más intensiva que el resto de técnicas. En cuanto a uso de RAM se notó un consumo moderado (entre 15 y 19 GB) para las herramientas que utilizaron minimizers y un bajo consumo (entre 2 y 4 GB) para las herramientas que no los utilizaron para el particionado. Finalmente, se notó un buen paralelismo en CPU por parte de las herramientas que usan esta técnica, con la excepción de KAnalyze.

Herramientas basadas en filtros Bloom: Las herramientas basadas en la estructura de datos de filtros Bloom evaluadas fueron: BFCounter, Turtle y KHMer. - Se evidencio que las técnicas que utilizan esta estructura de datos reducen significativamente el consumo de memoria, sin embargo cuando se combina con otros métodos con el objeto de paralelizar su ejecución (múltiples tablas hash) la exigencia de memoria aumenta. - Las técnicas que utilizan este filtro mostraron un tiempo de ejecución aceptable, sin embargo, cuando se establece un umbral máximo de consumo de memoria mediante la técnica SAC el tiempo de ejecución incrementa.

Herramienta basada en lpc-interval: La herramienta basada en árboles lpc-interval evaluada fue Tallymer. Se evidenció que esta técnica fue la que mayor tiempo de ejecución tuvo para el genoma escogido. Adicionalmente se evidenció que tiene un consumo moderado de memoria, utilizando entre 10 y 16 GB de RAM. En cuanto al acceso a disco, se notó un bajo promedio de operaciones de lectura y escritura. Finalmente, esta técnica tuvo un bajo paralelismo al utilizar únicamente entre 3 y 5 núcleos de los 24 disponibles.

Herramienta basada en Tablas Hash – lock Free: La herramienta basada en tablas hash con un enfoque lock-free evaluada fue Jellyfish. Se evidenció que esta técnica tiene un tiempo de ejecución corto, tardando alrededor de 7 minutos en procesar el genoma seleccionado. Por otro lado, este enfoque tiene un consumo moderado de RAM, utilizando cerca de 11 GB de memoria. En cuanto al acceso a disco, se notó un uso intensivo, comparable con la técnica de particionado en disco. Finalmente, esta técnica posee un buen paralelismo.

3.5. Conclusiones y oportunidades de mejoras para superar el reto de procesar k-mers

Después de analizar los resultados de la evaluación realizada, la pregunta a contestar es: ¿Cuál de las estructuras de datos y/o metodologías que presentaron mejor desempeño, ofrecen mayor potencial en el sentido de mejorarlas y así

generar un mayor y mejor impacto en el procesamiento de k-mers enfocado a la construcción de grafos en el ensamble genómico de-novo?

La respuesta a la anterior pregunta son las técnicas de procesamiento particionado usando semillas tipo minimizer, y la justificación es la siguiente:

- Las técnicas de procesamiento particionado presentan un excelente desempeño tanto en el tiempo de ejecución como en el uso eficiente de la memoria, lo que representa un muy buen punto de partida.
- Estas técnicas no solo favorecen a un tipo específico de procesamiento. El hecho de poder procesar de forma independiente un conjunto de datos por archivos menores no solo es favorable en el tratamiento de k-mers sino que podría generalizarse para favorecer cualquier otro proceso donde se manejen conjuntos de datos de tamaños elevados que se requieran cargar completos a la memoria.
- Este tipo de técnicas son muy jóvenes en el campo del tratamiento de k-mers lo que lleva a suponer que existen muchos factores por madurar y mejorar.
- A pesar de su corto tiempo en el campo del procesamiento de k-mers ya existen una gran cantidad de aplicaciones en diferentes áreas lo que implicaría un mayor impacto en caso de una mejoría. (En la evaluación anterior se pudo notar que este tipo de técnica era el que presentaba más herramientas).
- La obtención de semillas tipo minimizer es un proceso intensivo por el hecho de tener que comparar los pesos lexicográficos de cada uno de los m-mers de cada uno de los k-mers en cada una de las lecturas. Aparentemente sería una desventaja, pero realmente representa una oportunidad muy grande de mejoría mediante la paralelización masiva.

4. MINIMIZERS PARA FACILITAR EL PROCESAMIENTO DE K-MERS

En este capítulo se presenta un análisis teórico del uso de las semillas tipo minimizers para facilitar el procesamiento de k-mers desde dos enfoques: Estructura de datos "Super k-mers" y criterio de particionamiento. Adicional a los conceptos fundamentales, se presenta también las variaciones de minimizers necesarias para aumentar y/o mejorar su funcionalidad en el campo práctico, tales como minimizers canónicos, signatures y minimizers por frecuencia.

4.1. Que son los minimizers?

Los minimizers son propuestos inicialmente como una técnica para reducir los requerimientos de almacenamiento en procesos de comparación de secuencias biológicas (Roberts, et al. 2004), mediante la estrategia de reducir la redundancia presentada por la técnica "seed-and-extend" (Altschul et al., 1997; Lipman & Pearson, 1985; Pearson & Lipman, 1988). A partir del 2013, los minimizers hacen su incursión en tareas de tratamiento de k-mers en procesos de ensamblaje

de-novo: - contadores de k-mers KMC2 y KMC3 (Kokot, Długosz & Deorowicz, 2017) y MSPKmerCounter, - constructor de grafos MSPGraphBuilder (Li, et al., 2013), - compactador de grafos BCALM y BCALM2 (Chikhi, Limasset & Medvedev, 2016) y - ensamblador EPGA2.

Un minimizer de un k-mer es la sub-secuencia de longitud m (m-mer, donde m<k), que al comparar todos los m-mers posibles de dicho k-mer, presenta el menor valor de acuerdo a un criterio de comparación, que normalmente es el peso lexicográfico. Ver figura 4.1.

Figura 4.1 Minimizer de un k-mer. Fuente: Autor.

4.2. Que aportan los minimizers al procesamiento de k-mers?

De acuerdo a la definición del párrafo anterior, se puede afirmar que el minimizer de un k-mer es una sub-secuencia única y aquellos k-mers que sean contiguos en una lectura tienen una probabilidad muy alta de presentar el mismo minimizer; por esta razón pueden ser vistos como "semillas" y usados para enfrentar los dos retos del procesamiento de k-mers presentados en el capítulo anterior: el alto volumen de datos debido a la redundancia y la imposibilidad o dificultad del tratamiento particionado (Vera N., Gutierrez M. & Perez J., 2016).

Estructura de datos para reducir la redundancia

Los minimizers son usados para definir estructuras de datos donde NO se almacenan todos los k-mers de una lectura sino que se fusionan aquellos que son contiguos y presenten el mismo minimizer. El producto de esta fusión son sub-secuencias llamadas super k-mers (Li, et al., 2013). En la figura 4.2 se puede observar cómo se pasa de 7 k-mers (112 bases) a 2 super k-mers (37 bases).

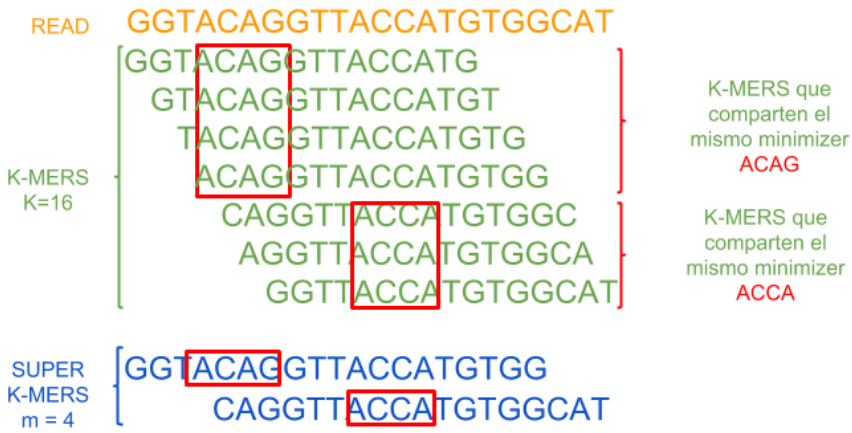


Figura 4.2 K-mers y super k-mers de una lectura (k = 16, m = 4). Fuente: Autor.

Criterio de partición del conjunto de datos

A pesar de que la fusión de k-mers en super k-mers reduce la redundancia, es muy probable que en el momento de tratar con dichos super k-mers el proceso exija la obtención de todos sus k-mers, lo que llevaría a demandar la misma memoria que si inicialmente se hubieran obtenido y almacenado todos los k-mers de las lecturas del conjunto de datos; por esta razón, los minimizers son usados también como criterio para dividir el conjunto de datos, creando particiones (archivos en disco) que contengan todos los super k-mers que posean el mismo minimizer, de tal forma que dichos archivos se puedan cargar en memoria y procesar por separado con la garantía de que no va a existir el mismo k-mer en diferente partición y que todos los k-mers contiguos que posean el mismo minimizers van a estar en la misma partición. Ver figura 4.3.

Figura 4.3 Distribución de super k-mers en particiones. Fuente: Autor.

4.3. Minimizers canónicos

Si la técnica de particionamiento se realiza para ejecutar un tipo de procesamiento que tiene en cuenta el complemento inverso de las secuencias y sub-secuencias, no se podría aplicar el criterio de minimizer tal como se explicó arriba, debido a que un k-mer podría quedar en una partición y su complemento inverso en otra.

Para solucionar lo anterior se hace una pequeña modificación al criterio de selección del minimizer de tal forma que se sigue escogiendo el de menor peso lexicográfico pero ahora no solo de los m-mers sino que también se incluyen sus complementos inversos (ver nota). Otra forma de definirlo es decir que se selecciona el de menor peso lexicográfico entre todos los m-mers canónicos del k-mer (ver nota).

Figura 4.4 Minimizer canónico (m=4) de un k-mer (k=16). Fuente: Autor.

En la figura 4.4 se observa la diferencia entre minimizer y minimizer canónico. Mientras que el minimizer del k-mer mostrado es ACCG (porque es el m-mer de menos peso lexicográfico), el minimizer canónico es AACC debido a que es la sub-secuencia de longitud 4 con menor peso lexicográfico entre todos los m-mers y sus complementos inversos; en este caso AACC corresponde al complemento inverso de GGTT.

Nota: El complemento inverso de un m-mer es el resultado de invertir la secuencia y reemplazar cada base por su complemento (A < - > T, C < - > G), por ejemplo el complemento inverso del m-mer GGTT es AACC.

4.4. Minimizers para una distribución más homogénea

Si el criterio de selección de los minimizer es exclusivamente el peso lexicográfico de los m-mers, los tamaños de cada de una de las particiones dependen directamente de la frecuencia de aparición de los minimizer, lo que significa que son definidos por las tendencias en las secuencias biológicas, por ejemplo los minimizer con varias As pueden ser muy comunes, mientras que minimizer con varias Ts pueden ser muy escasos. Lo anterior podría llevar a

distribuciones no muy homogéneas que generen archivos grandes y pequeños que no optimizan el uso de la memoria.

Con el objeto de conseguir distribuciones más homogéneas o distribuciones que favorezcan la estrategia de los procesos, se han propuesto variante de los minimizer que utilizan otro criterio diferente al peso lexicográfico o que adicionan restricciones.

Signatures

Con el propósito de conseguir distribuciones más homogéneas, los autores de los contadores de k-mers KMC 2 y KMC 3 han propuesto una alternativa de minimizer canónico donde se vetan algunos de ellos de acuerdo a unas características específicas con el propósito de que aquellos minimizer permitidos tengan frecuencias de aparición no tan diferentes. Los minimizer vetados son aquellos que cumplen las siguientes características: empezar con AAA, empezar con ACA, contener AA en algún lado (excepto en el inicio).

Figura 4.5 Signature (m=4) de un k-mer (k=16). Fuente: Autor.

En la figura 4.5 se observa como el minimizer canónico ACAG es vetado por iniciar por ACA y por consiguiente la signature es ACCA que es el complemento inverso de TGGT.

Minimizers por frecuencia

Los autores de BCALM y BCALM 2 han propuesto una alternativa de minimizers canónico que no se basa en el peso lexicográfico sino en la frecuencia de los m-mers. Con esto buscan una distribución de tal forma que a través del desarrollo de su algoritmo los archivos tiendan a homogenizar sus tamaños. BCALM es una herramienta compactadora del grafo de De Bruijn, cuyo algoritmo permite ir compactando gradualmente por archivos de tal forma que el resultado de un archivo pasa al otro y así sucesivamente hasta llegar a un archivo final donde se halla el resultado final. Se inicia por lo archivos cuyos minimizer presentaron la menor frecuencia, de tal forma que al ir pasando los resultados de archivo a archivo los tamaños se vayan homogenizando.

5. COMPUTACIÓN HETEROGÉNEA PARALELA

En este capítulo se realiza una revisión conceptual de la computación heterogénea y de sus estándares y modelos de programación más comunes: OpenCL y CUDA. Se aboradan los modelos de plataforma, ejecución y memoria de OpenCL y se exponen los fundamentos de CUDA en cuanto a modelo de programación, jerarquía de hilos, jerarquía de memoria y programación heterogénea.

5.1. ¿Qué es la computación heterogénea paralela?

A través de la historia de la computación, el paradigma de desarrollo y evolución de los procesadores se había enfocado en el aumento de su capacidad de cómputo mediante el incremento de la frecuencia de reloj, con el objeto de ejecutar una mayor cantidad de instrucciones en el menor tiempo posible. Sin embargo, desde 2003 debido al consumo de energía y los problemas de disipación de calor que limitan la construcción de procesadores que aumenten la frecuencia de reloj y el nivel de actividades productivas que puede ejecutarse en cada periodo de reloj en un único procesador, se cambió el enfoque integrando múltiples unidades de procesamiento en un mismo chip para aumentar el poder de procesamiento (de Antonio & Marina, 2005). Gracias al desarrollo de estos procesadores se abrió la posibilidad de resolver problemas computacionales que antes hubieran sido imposibles (Alba, 2005). Estos problemas deben ser solucionados de una manera distinta a como se resuelven linealmente, tomando un problema cualquiera se divide en un conjunto de sub-problemas para resolver éstos simultáneamente sobre diferentes unidades de procesamiento.

De acuerdo a lo expuesto en el párrafo anterior, en la actualidad el desarrollo de sistemas de procesamiento se ha enfocado en producir dispositivos con la capacidad de ejecución simultánea de dos manera diferentes: La primera opción es el diseño de CPUs multi-core, optimizadas para reducir el tiempo de ejecución de procesos secuenciales (lactency cores); la segunda opción, es el diseño de sistemas de procesamiento many-thread, como por ejemplo las GPUs (Unidades de Procesamiento Gráfico) optimizadas para mejorar el desempeño (menos tiempo y menos consumo de energía eléctrica) en la ejecución de procesos paralelizables (throughput cores). Debido a que la mayoría de problemas computacionalmente intensivos poseen procesos tanto secuenciales como paralelizables, en los últimos años se ha iniciado el proceso de integración de los sistemas multi-core y los sistemas many-thread en plataformas computacionales denominadas heterogéneas (Kirk & Wen-mei, 2012).

Una plataforma de computación heterogénea se define como un sistema conformada por lo menos de dos tipos diferentes de procesadores, normalmente, con el objeto de incorporar capacidades de procesado especializadas para realizar tareas particulares (Amar Shan, 2006). Un sistema heterogéneo se conforma habitualmente por una o más CPU(s) que cumple(n) la función de unidad de procesamiento principal (llamado generalmente Host) y uno o más dispositivos de procesamiento diferentes, como por ejemplo GPUs (Graphics Processing Units), DSPs (Digital Signal Processors), FPGAs (Field Programmable Gate Arrays), que cumple(n) la función de aceleradores (ver figura 5.1). También se puede encontrar la integración de dos o más tipos de procesadores en un solo Chip, por ejemplo un APU (accelerated processing unit) es un microprocesador que integra una CPU multinúcleo y una GPU mediante un bus de alta velocidad.

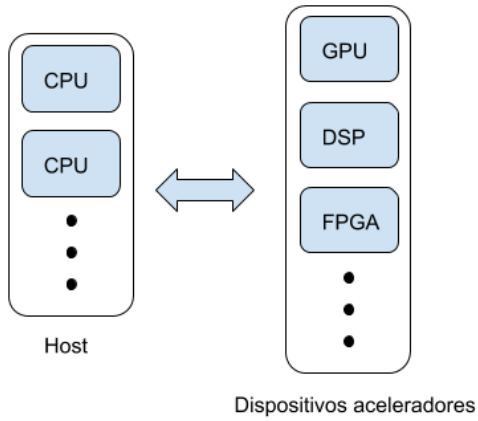


Figura 5.1 Plataforma heterogénea típica. Fuente: Autor.

Así como la heterogeneidad entre dispositivos de procesamiento representa una ventaja al ofrecer capacidades de procesado especializadas para realizar tareas particulares, también representa una gran desventaja desde el punto de vista del desarrollo. La heterogeneidad entre dispositivos de procesamiento se centra principalmente en la diferencia entre arquitecturas de conjuntos de instrucciones ISA (Instruction Set Architecture), por tal motivo cada uno de los tipos de dispositivos podrá contar con modelos, paradigmas y herramientas de programación totalmente diferentes, lo que conlleva a procesos de desarrollo separados con tortuosas integraciones.

Los limitantes en la integración de procesos de desarrollo para los diferentes tipos de dispositivos que pueden estar involucrados en un sistema heterogéneo se han comenzado a mitigar con la creación de estándares de plataformas y modelos de programación tales como CUDA y OpenCL.

5.2. Fundamentos de OpenCL

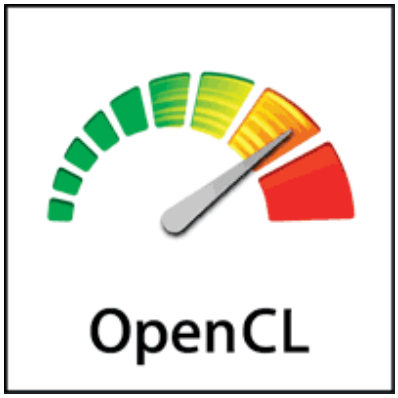


Figura 5.2 Logotipo de OpenCL. Fuente: https://www.khronos.org/opencl/

OpenCL (Open Computing Language) es un estándar abierto y libre para la programación paralela de propósito general sobre plataformas heterogéneas compuestas por CPUs (host) y otros dispositivos (aceleradores) tales como GPUs, DSPs, FPGAs, entre otros (ver logo en la figura 5.2). El objetivo de OpenCL es abstraer cualquier sistema heterogéneo y presentarlo como una única jerarquía de 4 modelos (plataforma/memoria/ejecución/programación) que permita integrar el proceso de desarrollo (OpenCL 2.2 API Specification, 2017).

OpenCL consiste en un framework conformado por una API (Application Programming Interface) y un lenguaje de programación multiplataforma. Este framework presenta las siguientes características:

- Soporta ambos modelos de programación paralela: datos y tareas
- Utiliza un subconjunto de ISO C99 con extensiones para el paralelismo
- Define requisitos numéricos consistentes basados en IEEE 754
- Define un perfil de configuración para dispositivos portables y embebidos
- Permite la interacción eficiente con OpenGL, OpenGL ES y otras APIs gráficas

Como se mencionó arriba, OpenCL se define totalmente mediante una jerarquía de 4 modelos: plataforma, memoria, ejecución y programación. A continuación se describen los modelos de plataforma, de ejecución y de memoria (la parte básica del modelo de programación se explica en la parte II del presente libro mediante ejemplos fundamentales).

Modelo de plataforma

El modelo de plataforma de OpenCL consiste en un host conectado a uno o varios dispositivos OpenCL (ver figura 5.3). Un dispositivo está conformado por uno o más unidades de cómputo CUs (Compute Units), que a su vez están divididas en elementos de procesamiento PEs (Processing Elements).

Una aplicación OpenCL está conformada por dos tipos de códigos: un código de host que se ejecuta sobre las CPUs de acuerdo al modelo nativo de su plataforma y un código kernel que es suministrado del host a los dispositivos como comandos para que sean ejecutados por los elementos de procesamiento.

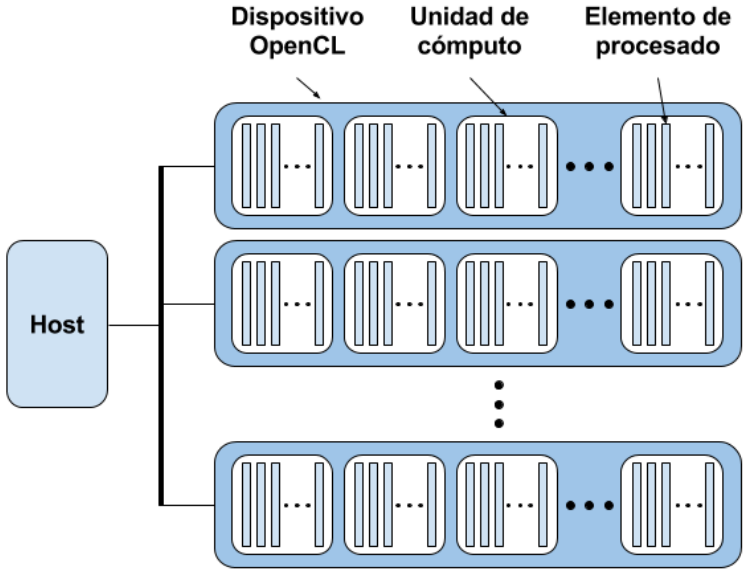


Figura 5.3 Modelo de plataforma de OpenCL. Fuente: Autor.

Modelo de ejecución

El modelo de ejecución de OpenCL se basa en dos unidades de ejecución: Un programa host que se ejecuta en la(s) CPU(s) y un programa kernel que se ejecuta en el(los) dispositivo(s) OpenCL. El trabajo de cómputo programado en los kernels es realizado por ítems de trabajo, los cuales se reúnen en grupos de trabajo para la ejecución de dicho trabajo. La interacción entre el host y los dispositivos se habilita mediante un contexto y se realiza a través de comandos.

Contexto: Los kernels son ejecutados bajo un ambiente definido por un contexto el cual es creado y administrado por el host. Dicho ambiente incluye los siguientes recursos:

- Dispositivos: Uno o más dispositivos OpenCL.
- Objetos kernel: Funciones OpenCL (con sus argumentos asociados) que se ejecutan en los dispositivos.
- Objetos programa: Programa fuente y ejecutable que implementa a los kernels.
- Objetos memoria: Variables visibles al host y a los dispositivos.

Cola de comandos: El contexto descrito arriba habilita al host para que interactúe con los dispositivos, esta interacción se realiza mediante una cola de comandos (command-queue). Existirá una cola de comandos por cada dispositivo con el que se requiera interactuar. De acuerdo a la funcionalidad cumplida, los comandos son clasificados en tres tipos: Comandos para poner en cola un kernel que se va a ejecutar (comandos Kernel-enqueue), comandos para la transferencia de datos (comandos memory) y comandos para procesos de sincronización (comandos synchronization).

Adicional a las colas que reciben los comandos enviados por el host, existen también colas alojadas en el lado de los dispositivos que reciben comando enviados por un kernel que se encuentre ejecutándose en un dispositivo. Estos comandos son usados para que un kernel (padre) tenga la capacidad de enviar a ejecución otro kernel (hijo).

Todos los comandos, sin importar que provengan del host o de un kernel pasan por seis estados: en cola (queued), suministrado (submitted), listo (ready), corriendo (running), finalizado (ended) y completo (complete). Los comandos comunican sus estados a través de objetos denominados eventos.

Múltiples colas de comandos pueden estar presentes en un mismo contexto y múltiples colas de comandos puede ejecutar comandos de forma independiente. Para la sincronización de comandos provenientes de varias colas el host utiliza los eventos para definir puntos de sincronía.

Instancia de un kernel: Se le llama instancia de un kernel a la integración de 3 componentes: el kernel, los valores de los argumentos asociados a dicho kernel y un espacio indexado definido cuando el kernel es enviado para su ejecución.

Cuando una instancia de un kernel es enviada para su ejecución, cada función de dicho kernel es ejecutada por un punto del espacio indexado definido; estas funciones son llamadas ítems de trabajo (work-items) y son integradas en grupos de trabajo (work-groups) para su administración por parte del dispositivo. Un ítem de trabajo se puede identificar de dos formas: mediante un ID global basado en las coordenadas dentro del espacio indexado o mediante un ID local a un grupo de trabajo.

Espacio indexado - NDRange: El espacio indexado definido para un kernel en OpenCL es llamado NDRange y corresponde a un espacio N-dimensional, donde N puede tomar el valor de 1 o 2 o 3. El NDRange se conforma de grupos de trabajo que a su vez están conformados por ítems de trabajo. Estos ítems de trabajo se pueden sincronizar entre sí exclusivamente entre un mismo grupo de trabajo; la sincronización entre ítems de diferentes grupos de trabajos es imposible debido al modelo de memoria, explicado más adelante. En la figura 5.4 se muestra un ejemplo de un NDRange con N=2.

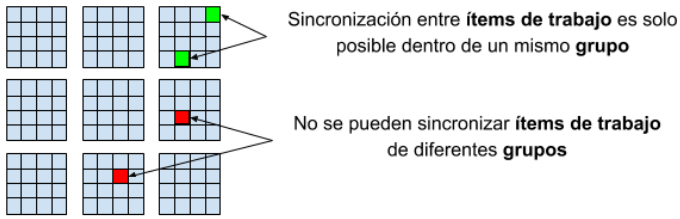


Figura 5.4 Ejemplo de un NDRange con N=2. (Imagen modificada de https://handsonopencl.github.io/)

Un NDRange es definido por 3 arreglos de enteros, cada uno con tamaño N (un valor por cada dimensión):

- Un arreglo para indicar el tamaño del espacio indexado en cada dimensión. En el ejemplo mostrado en la Figura 5.4 será un arreglo de dos valores [12,12].
- Un arreglo para indicar el valor inicial del índice en cada dimensión. Este arreglo representa una especie de offset en el índice de cada dimensión (este valor es denominado F). Por defecto los índices se inician en cero (0).
- Un arreglo para indicar el tamaño de los grupos de trabajo en cada una de las dimensiones. En el ejemplo mostrado en la Figura 5.4 será un arreglo de dos valores [4,4].

El ID global de cada uno de los ítems de trabajo será una tupla N-dimensional, con valores en el rango, F a F más el número de ítems en cada dimensión menos uno (1).

El tamaño de los grupos de trabajo no necesariamente debe ser el mismo en todas las dimensiones. En el caso que en alguna de las dimensiones el tamaño global no sea divisible por el tamaño del grupo de trabajo se generarán dos regiones, una con grupos de trabajo con el tamaño definido por el programador y otra región con grupos de trabajo de tamaño menor para completar exactamente el tamaño global requerido. Esto se puede presentar en cualquiera de las dimensiones, de tal manera que si se tiene un arreglo con N = 3, podrían existir 8 tamaños diferentes de grupos de trabajo.

La forma de identificar cada uno de los ítems y grupos de trabajo y la forma de representar los diferentes tamaños en un NDRange es ilustrado en la Figura 5.5.

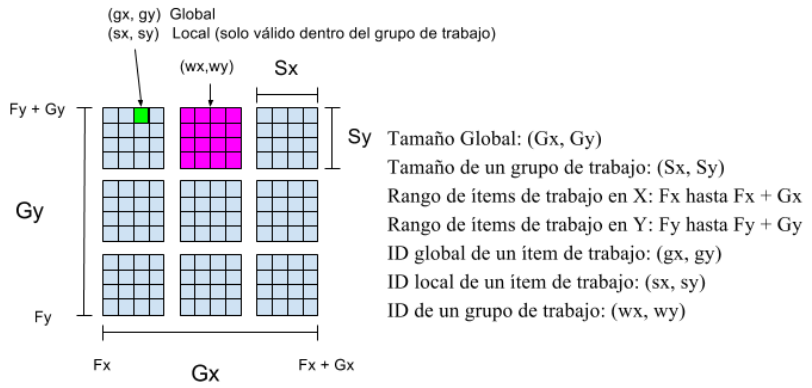


Figura 5.5 Identificadores y tamaños en un NDRange. Fuente: Autor.

El ID global de un ítem de trabajo se puede obtener a partir de su ID local, siempre y cuando se tenga el offset, el ID y el tamaño del grupo de trabajo:

(gx, gy) = (wx*Sx + sx + Fx, wy*Sy + sy + Fy)

La cantidad de grupos de trabajo puede obtenerse a partir del tamaño global y el tamaño del work-group:

(Wx, Wy) = (ceil(Gx/Sx), ceil(Gy/Sy))

Si se desea obtener el ID del work-group al cual pertenece un ítem de trabajo se debe contar con el ID global y local del ítem, con el offset y con el tamaño del work-group:

(wx, wy) = ((gx – sx – Fx) / Sx, (gy – sy – Fy) / Sy)

Modelo de memoria

El modelo de memoria de OpenCL define la estructura, el contenido y el comportamiento de la memoria expuesta por una plataforma OpenCl y usada por un programa OpenCL. Este modelo se conforma de 4 componentes:

- Regiones de memoria: Las diferentes memorias visibles tanto para el host como para los dispositivos que comparten un contexto.
- Objetos de memoria: Es el contenido de la memoria global. Son objetos definidos por la API de OpenCL.
- Memoria Virtual Compartida SVM (Shared Virtual Memory): Es un espacio virtual direccionado expuesto tanto para el host como para los dispositivos dentro de un contexto.
- Modelo de consistencia: Reglas que definen cuáles valores son observados cuando múltiples unidades de ejecución cargan datos de la

memoria. También se determinan las operaciones atomic/fence que restringen el orden de las operaciones de memoria y definen las relaciones de sincronización.

Regiones de memoria: Desde el punto de vista de OpenCL existen dos tipos de memoria: Memoria de host y memoria de dispositivo (ver figura 5.6).

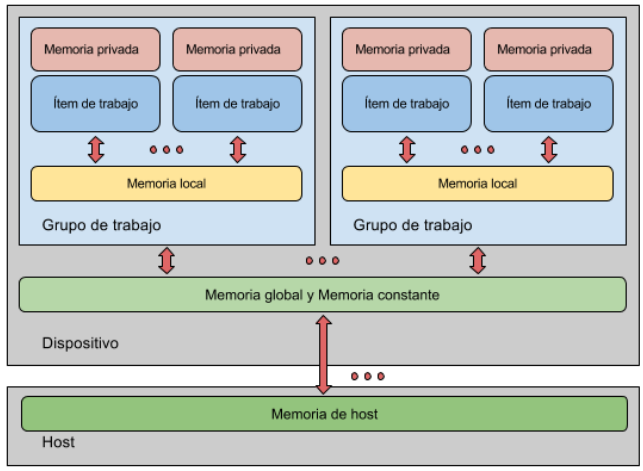


Figura 5.6 Tipos y regiones de memoria desde el punto de vista de OpenCL. Fuente: Autor.

Memoria de host: Es la memoria directamente disponible para el host (la estructura y comportamiento de esta memoria no es definida por OpenCL). La transferencia de objetos de memoria entre el host y los dispositivos se realiza a través de funciones de la API de OpenCL o mediante una interfaz de memoria virtual compartida.

Memoria de dispositivo: Memoria directamente disponible para los kernels ejecutados en un dispositivo OpenCL. La memoria de dispositivo se divide en cuatro (4) regiones:

- Memoria global: Esta región de memoria permite el acceso para lectura/escritura a todos los ítems de trabajo de todos los grupos de trabajo que se ejecutan sobre algún dispositivo dentro de un contexto.
- Memoria constante: Una región de la memoria global que se mantiene constante durante la ejecución de una instancia de kernel. El host asigna e inicializa objetos de memoria colocados en la memoria constante.
- Memoria local: Una región de memoria local a un grupo de trabajo. Esta región de memoria se puede utilizar para asignar las variables que son compartidos por todos los ítems de trabajo pertenecientes a un grupo.
- Memoria privada: Una región de memoria exclusiva a un ítem de trabajo. Una variable definida en una memoria privada de un ítem de trabajo no será visible para ningún otro ítem.

Las memorias locales y privadas siempre están asociados con un dispositivo específico. Las memorias globales y constantes, se comparten entre todos los dispositivos dentro de un contexto dado. Un dispositivo de OpenCL puede incluir una memoria caché para apoyar el acceso eficaz a estas memorias compartidas.

Objetos de memoria: El contenido de la memoria global son objetos de memoria. OpenCL define tres tipos de objetos de memoria: búfer (buffer), imagen (image) y tubería (pipe).

- Búfer: Objeto almacenado como un bloque contiguo de memoria y usado como un objeto de propósito general para mantener datos en un programa OpenCL. Los valores almacenados en un búfer pueden ser de tipos primitivos (enteros, flotantes), vectores o estructuras de datos definidas por el usuario. Un búfer puede ser manipulado a través de punteros de forma similar como se manipula un bloque de memoria en C.
- Imagen: Objeto usado para almacenar imágenes de una, dos o tres dimensiones en los formatos estándar utilizados por las aplicaciones gráficas. Un objeto imagen es un tipo de dato opaco que no puede ser visto directamente a través de punteros en el código del dispositivo.
- Tubería: Objeto usado para organizar los datos en una estructura de tipo FIFO (first in, first out) con el propósito de facilitar el paso de datos procesados de un kernel a otro.

La asignación y acceso a los objetos de memoria dentro de las diferentes regiones de memoria varía entre el host y los ítems de trabajo ejecutándose sobre un dispositivo (ver tabla 5.1).

	Host		**Kernel**	
	Asignación	**Acceso**	**Asignación**	**Acceso**
Global	Asignación dinámica	Acceso de lectura/escritura a búfer e imágenes pero no a tuberías.	Asignación estática por las variables del programa	Acceso de lectura/escritura
Constante	Asignación dinámica	Acceso de lectura/escritura	Asignación estática	Acceso a solo lectura
Local	Asignación dinámica	Sin acceso	Asignación estática. Asignación dinámica para kernel hijos	Acceso de lectura/escritura. Sin acceso a memoria local de kernel hijos

Privada	Sin asignación	Sin acceso	Asignación estática	Acceso de lectura/escritura

Tabla 5.1 Modo como los objetos de memoria son asignados y accedidos por el host y/o una instancia de un kernel.

Memoria virtual compartida (SVM - Shared Virtual Memory): Una de las grandes novedades de OpenCL 2.0 es el soporte a una memoria virtual compartida, llamada SVM por su sigla en inglés. SVM extiende la región de memoria global dentro de la memoria de host, permitiendo el direccionamiento virtual compartido entre el host y todos los dispositivos dentro de un contexto.

Hay tres tipos de SVM definidos por OpenCL:

- SVM - Búfer de grano grueso: Permite que el direccionamiento virtual compartido ocurra con una granularidad de objetos de memoria búfer; esto es similar a lo que se hacía sin contar con una SVM. La diferencia entre un SVM búfer de grano grueso y un No-SVM búfer es simplemente que el host y los dispositivos comparten punteros de memoria virtual.
- SVM - Búfer de grano fino: Permite que el direccionamiento virtual compartido ocurra con una granularidad de cargas o almacenamientos individuales dentro de los bytes de un objeto de memoria búfer.
- SVM - Sistema de grano fino: Permite que el direccionamiento virtual compartido ocurra con una granularidad de cargas o almacenamientos dentro de los bytes en cualquier parte de la memoria de host.

En la tabla 5.2 se realiza un resumen de los tipos de SVM y sus principales características, tales como la granularidad de lo compartido, el modo de asignación de memoria y el mecanismo para asegurar la consistencia.

	Granularidad de lo compartido	**Modo de asignación de memoria**	**Mecanismo para asegurar la consistencia**	**Actualizaciones explícitas entre el host y los dispositivos?**
Búfer No-SVM	Objetos de memoria OpenCL (búfer)	clCreateBuffer (función que crea un objeto de memoria búfer)	Puntos de sincronización del host sobre el mismo o entre dispositivos	Si, a través de comandos *Map* y *Unmap*
SVM - Búfer de grano grueso	Objetos de memoria OpenCL (búfer)	clSVMAlloc (función para asignar un búfer SVM, que puede ser compartido por el host y todos los dispositivos en un	Puntos de sincronización del host entre dispositivos	Si, a través de comandos *Map* y *Unmap*

		contexto OpenCL)		
SVM - Búfer de grano fino	Bytes dentro de los objetos de memoria OpenCL (búfer)	clSVMAlloc	Puntos de sincronización más operaciones *atomics*	No
SVM - Sistema de grano fino	Bytes dentro de la memoria de host (sistema)	Mecanismos para asignar memoria de host (Ej: malloc)	Puntos de sincronización más operaciones *atomics*	No

Tabla 5.2 Tipos de SVM y sus principales características.

5.3. Fundamentos de CUDA

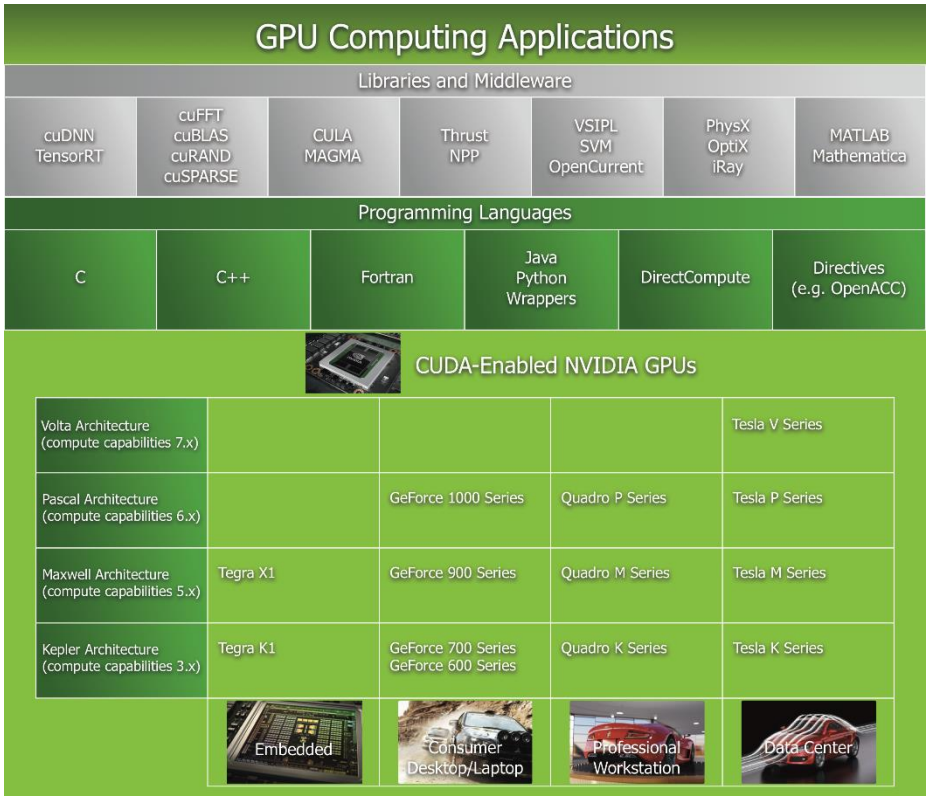


Figura 5.7 Lenguajes de programación y APIs soportados por CUDA. Fuente: http://docs.nvidia.com/cuda/cuda-c-programming-guide

CUDA es una plataforma de computación paralela de propósito general y un modelo de programación. Su principal objetivo es habilitar el uso de GPUs NVIDIA para soluciona problemas computacionales complejos de una forma más eficiente que como se hace sobre una CPU (CUDA C Programming Guide, 2017). CUDA incluye un entorno de software que permite a los desarrolladores usar C como un lenguaje de alto nivel. También soporta otros lenguajes de programación y APIs como se puede observar en la figura 5.7.

Modelo de programación de CUDA

El modelo de programación de CUDA se soporta sobre 3 abstracciones claves: jerarquía de grupos de hilos, memorias compartidas y barreras de sincronización, que se presentan al programador como un conjunto mínimo de extensiones de lenguaje. Estas abstracciones guían al programador a dividir el problema en sub-problemas gruesos que pueden resolverse de forma independiente en paralelo mediante bloques de hilos, y cada sub-problema en piezas más finas que se pueden resolver cooperativamente en paralelo por todos los hilos dentro del bloque.

El modelo es escalable de forma automática, en el sentido que los bloques de hilos no van sujetos al número de multiprocesadores de la GPU. La ejecución de los bloques se adapta al número de multiprocesadores disponibles. Ver figura 5.8.

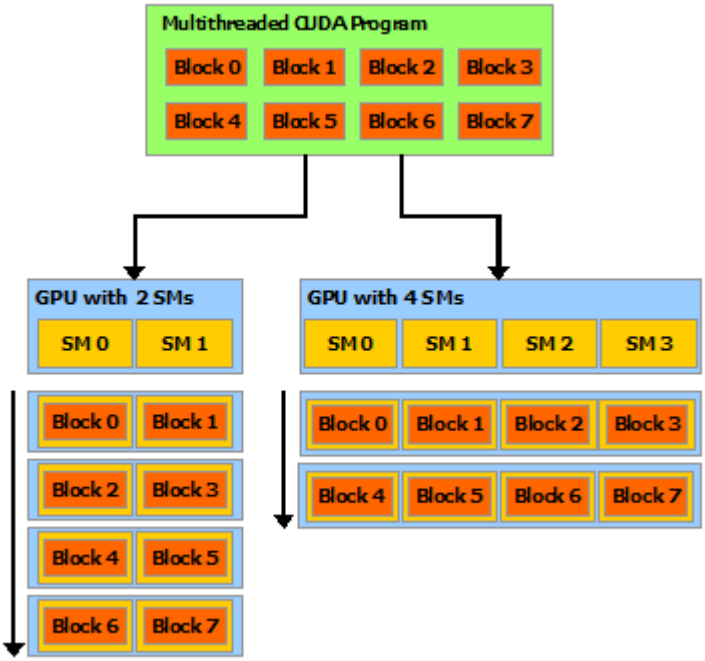


Figura 5.8 Escalabilidad automática de CUDA: Los bloques de hilos se distribuyen de forma homogénea entre los SMs (Streaming Multiprocessors). Fuente: http://docs.nvidia.com/cuda/cuda-c-programming-guide

Kernels: CUDA extiende C de tal forma que el programador pueda definir funciones denominadas kernels, que cuando sean llamadas, se ejecuten N veces en paralelo por N diferentes Hilos CUDA. El número de hilos a ejecutar la

función se define en el momento de llamar el kernel. En la figura 5.9 se puede observar un ejemplo de definición y llamado de un kernel.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figura 5.9 Ejemplo de definición y llamado de un kernel. Fuente: http://docs.nvidia.com/cuda/cuda-c-programming-guide

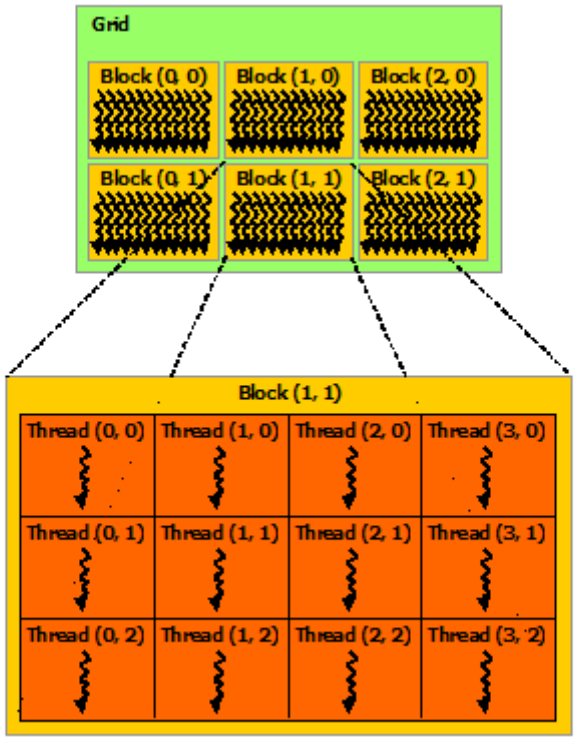


Figura 5.10 Malla de bloques de hilos. Fuente: http://docs.nvidia.com/cuda/cuda-c-programming-guide

Jerarquía de hilos: En CUDA los hilos se pueden agrupar en bloques de 1, 2 o 3 dimensiones y a su vez esos bloques se pueden agrupar en mallas de 1, 2 o 3 dimensiones. En la figura 5.10 se puede observar una grilla de 2 dimensiones conformada por bloques de hilos también de 2 dimensiones.

El número de hilos por bloque y el número de bloques por malla se determinan en el momento de llamar el kernel. Dentro del kernel tanto los bloques como los hilos tienen un identificador que se puede acceder a través de una variable (built-in). Para el caso de los bloques la variable es blockIdx y para el caso de los hilos es threadIdx. Adicionalmente se puede acceder a la dimensión de los bloques mediante la variable blockDim.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Figura 5.11 Ejemplo de definición y llamado de un kernel con una malla bidimensional conformada por bloques bidimensionales de hilos. Fuente: http://docs.nvidia.com/cuda/cuda-c-programming-guide

En el ejemplo de la figura 5.11 se definen bloques de tamaño 16x16 (256 hilos), que se agrupan en una malla bidimensional definida de tal forma que hallan suficientes bloques como para disponer de un hilo por cada elemento de la matriz a procesar.

Jerarquía de memoria: Las memorias con las cuales se cuenta en CUDA se organizan de forma jerárquica de acuerdo a su visibilidad. Cada hilo tiene una memoria privada de uso exclusivo, cada bloque de hilos tiene una memoria compartida a la cual pueden acceder todos los hilos de un bloque pero no los de otros bloques, por último todos los hilos sin importar de que bloque sean pueden acceder a una memoria denominada global. Adicional a esta memoria global existen otras dos memorias de acceso general para todos los hilos pero únicamente para su lectura, estas memoras son la de textura y la constante.

En la figura 5.12 se pueden observar los diferentes tipos de memoria en CUDA con su visibilidad por parte de los hilos, los bloques y las mallas.

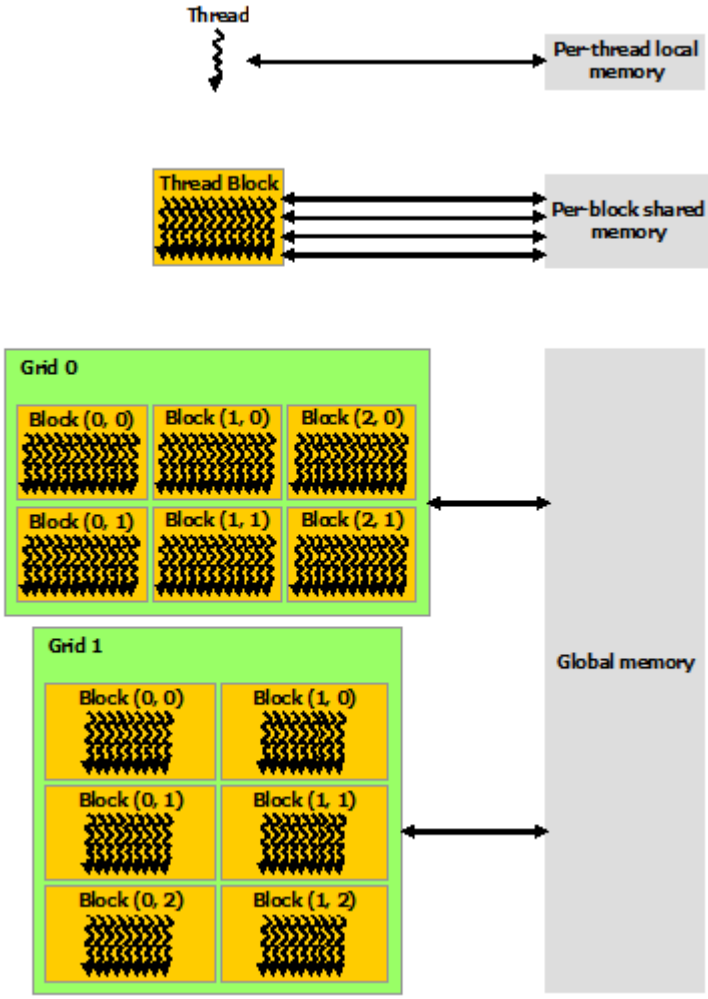


Figura 5.12 Jerarquía de memoria en CUDA. Fuente: http://docs.nvidia.com/cuda/cuda-c-programming-guide. Fuente: Autor.

Programación heterogénea: El modelo de programación de CUDA asume que sus hilos se ejecutan en un dispositivo separado físicamente que actúa como co-procesador del host donde se ejecuta el programa C desde el cual se llaman los kernels. Para el caso de tener una CPU y una GPU, esta última actuará como co-procesador del host CPU.

El modelo también asume que tanto la GPU como la CPU poseen su propio espacio de memoria independiente en la DRAM y se refiere a estos espacios como memoria de dispositivo y memoria de host respectivamente. En la figura 5.13 se puede observar el concepto de programación heterogénea: un programa en C que ejecuta de forma secuencial código serial que es ejecutado en el host y código paralelo que es ejecutado en el dispositivo (GPU).

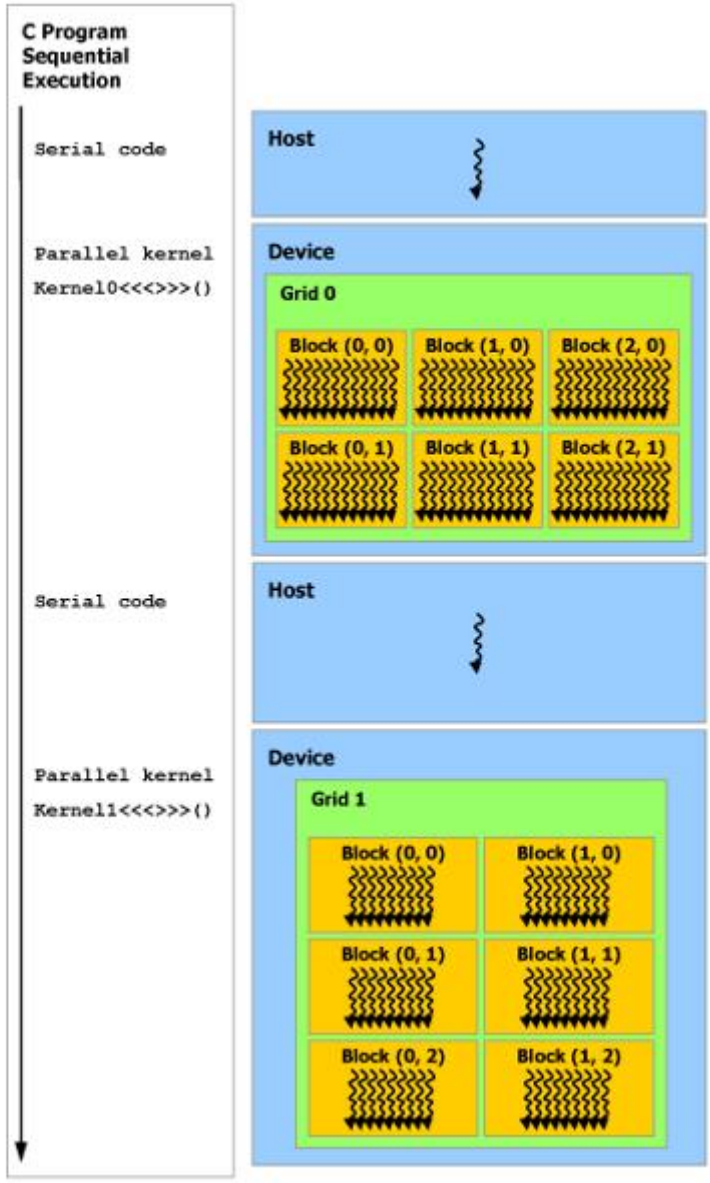


Figura 5.13 Programación heterogénea. Fuente: http://docs.nvidia.com/cuda/cuda-c-programming-guide. Fuente: Autor.

6. MEJORANDO EL DESEMPEÑO DEL PROCESAMIENTO PARALELO SOBRE PLATAFORMAS MANY CORE

En este capítulo se presenta un análisis teórico acerca del desempeño del procesamiento paralelo sobre plataformas Many Core. Adicionalmente el autor presenta una guía en forma de "mejores prácticas" para aumentar este desempeño mediante dos enfoques estratégicos: maximizando el speed-up y maximizando la intensidad operacional.

6.1. ¿Cuáles son las métricas de desempeño usadas en el procesamiento paralelo?

Tiempo de procesamiento

Es el tiempo empleado en la ejecución de un código para realizar una tarea específica. El principal objetivo de la programación paralela es precisamente realizar las mismas tareas pero en un menor tiempo.

Speed-up

El término speed-up hace referencia al factor en que se aumenta la velocidad de procesamiento (o se disminuye el tiempo de procesamiento) de una tarea específica (Karp, A. H., & Flatt, H. P., 1990). La forma de medir el speed-up depende del escalonamiento: la Ley de Amdahl propone un escalonamiento fuerte donde el problema es fijo, mientras que la Ley de Gustafson propone un escalonamiento débil donde el problema se puede redimensionar (Hill, M. D., 2008).

Escalamiento fuerte: Es una forma de medir como decrece el tiempo de procesamiento para un problema global fijo a medida que se aumenta el número de procesadores. Este escalonamiento se calcula mediante la Ley de Amdahl, la cual dice que el máximo speed-up que se puede alcanzar debido a la paralelización de porciones de programas seriales depende de la cantidad de procesadores disponibles y del porcentaje de código que se logra paralelizar.

$S = (s + p) / (s + p / N)$
$S = 1 / (s + p / N)$
S -> Máximo speed-up
s -> Porción de tiempo gastado (por un procesador serial) para ejecutar la parte serial de un programa
p -> Porción de tiempo gastado (por un procesador serial) para ejecutar la parte paralelizable de un programa
N -> Número de procesadores

s+p = 1 -> Tiempo total de ejecución del programa en un procesador serial
s+p/N -> Tiempo total de ejecución del programa en un sistema paralelo

Ecuación 6.1

En el caso de la paralelización de programas mediante GPUs el número de procesadores es alto y el término p/N tiende a 0 (cero), por tal motivo la ecuación se puede aproximar a S = 1/s, lo que indica que el máximo speed-up depende exclusivamente del porcentaje de código que se logre paralelizar.

Escalamiento débil: Es una forma de medir la variación del tiempo de procesamiento para un problema global variable que aumenta a medida que se agregan procesadores al sistema los cuales adicionan una porción fija del problema (la que se encargan de procesar). Este escalonamiento se calcula mediante la Ley de Gustafson, la cual define el máximo speed-up de un programa como:

$$S = (s' + Np') / (s' + p')$$
$$S = s' + Np'$$
$$S = N + (1 - N)s'$$

S -> Máximo speed-up
s' -> Porción de tiempo gastado (por el sistema paralelo) para ejecutar la parte serial de un programa
p' -> Porción de tiempo gastado (por el sistema paralelo) para ejecutar la parte paralelizable de un programa
N -> Número de procesadores
s'+p' = 1 -> Tiempo total de ejecución del programa en un sistema paralelo
s'+Np' -> Tiempo total de ejecución del programa en un procesador serial

Ecuación 6.2

En el caso de la paralelización de programas mediante GPUs el número de procesadores es alto y el máximo speed-up se puede aproximar a S = N(1 - s'), lo que significa que a medida que la porción serial de un programa sea reducida, el valor máximo del speed-up se aproxima al número de procesadores.

Ancho de banda

El ancho de banda es la tasa a la cual se transfieren los datos en un dispositivo CUDA/OpenCL. Existe un ancho de banda teórico que depende de las características del dispositivo sin embargo en la práctica este ancho de banda es diferente esencialmente por la estrategia que se utilice para hacer uso de las diferentes memorias, este ancho de banda es llamado efectivo y depende del tiempo real que se emplee para hacer la lectura y escritura de todos los datos en un kernel específico (CUDA C Best Practices Guide, 2017).

Ancho de banda teórico: El pico teórico del ancho de banda de un dispositivo CUDA/OpenCL depende de sus especificaciones; se halla multiplicando la frecuencia del reloj por el ancho de la interfaz de la memoria.

TBW = Frecuencia del reloj (Hz) x Ancho de la interfaz (Bytes)

Ecuación 6.3

Ancho de banda efectivo: Depende de la cantidad de datos accedidos por el programa y el tiempo que toma dichas transferencias.

EBW = (Datos leídos (Bytes) + Datos escritos (Bytes))/Tiempo

Ecuación 6.4

Intensidad operacional/aritmética

El modelo de Roofline define la intensidad operacional como la relación entre el trabajo (cantidad de operaciones por segundo) y el tráfico de datos de la memoria (Williams, S., Waterman, A., & Patterson, D., 2009).

$I = W/Q$

I -> Intensidad Operacional (FLOPS/Bytes)
W -> Trabajo (FLOPS)
Q -> Tráfico de la memoria (Bytes)

Ecuación 6.5

La intensidad operacional se debe interpretar como el número de operaciones realizadas por segundo por cada Byte de datos transferido entre las memorias. Este parámetro en gran parte define el desempeño de un kernel, debido a que representa una medida de que tanto trabajo se puede realizar por segundo y que tanta latencia se puede tener debido al tráfico de datos por la memoria.

6.2. Estrategias para mejorar el desempeño

Maximizar el speed-ud

De acuerdo a lo expuesto en la sección anterior referente al speed-up, para maximizar dicho factor de aceleración, se debe maximizar la cantidad de código paralelizado. Y para lograr paralelizar de forma eficiente la mayor cantidad de procesos es recomendable tener presente y aplicar algoritmos que permitan paralelizar disminuyendo la cantidad de pasos y la cantidad de operaciones.

Algunos de estos algoritmos son: Convolución, reducción, escaneo, conteo, histograma y operaciones atómicas.
Convolución: Es una operación matemática en la que dadas dos funciones o señales se busca generar una tercera que representa superposición de ambas.

Esta operación es muy usada en física, electrónica, óptica, estadística, entre otros. Este patrón busca generar la salida de esta operación de forma paralela haciendo uso de las matrices para representar las funciones, señales o imágenes a procesar y los divide en sub-matrices para hacer las operaciones en hilos individuales (Fialka, O., & Cadik, M., 2006).

Reducción: Este patrón busca optimizar una operación que calcule un solo resultado de un conjunto de datos, esta operación puede ser una suma, un producto, un promedio o un valor máximo, entre otras. La técnica usada por este patrón es dividir el conjunto de datos en grupos más pequeños para ejecutar la operación en cada grupo al mismo tiempo y utilizar los resultados de estos grupos como un nuevo conjunto de datos y volver a aplicar la reducción hasta que se llegue al resultado único. Actualmente existen diferentes técnicas para resolver problemas de reducción (Yu & Rauchwerger, 2014).

Escaneo: Este patrón hace uso del algoritmo "all-prefix-sums" que se utiliza frecuentemente para ordenamiento, comparación de cadenas, evaluación polinomial, construcción de grafos, entre otros (Chang, L., & Gómez-Luna, J., 2017). El escaneo utiliza una operación en la cual dado un arreglo de tipo:

Dado [a0, a1, a2,…, aN-1] se calcula:
[a0, (a0 ⊕ a1),…, (a0 ⊕ a1 ⊕… ⊕ aN-1)]

Ecuación 6.6

La anterior operación es un "Escaneo inclusivo" ya que incluye en la suma la misma posición que se calcula. También existe una operación llamada "Escaneo exclusivo" en el cual se calcula la suma sin tener en cuenta la posición actual del recorrido, esta operación calcula el siguiente resultado:

[0, a0, (a0 ⊕ a1),…, (a0 ⊕ a1 ⊕… ⊕ aN-2)]

Ecuación 6.7

Histograma: Los histogramas son conjuntos de datos muy usados en cálculos estadísticos y en otras áreas que hacen uso de estos datos como telecomunicaciones, tratamiento de imágenes, seguridad, entre otros. Este patrón busca extraer características notables de conjuntos de datos muy grandes de forma paralela dividiendo los datos de entrada en elementos mucho más pequeños y analizando cada uno por medio de un hilo recolectando la información relevante, por ejemplo un histograma de frecuencia calcularía la cantidad de veces que aparece un dato en la cadena (Ramtin Shams & R. A. Kennedy., 2007).

Operaciones atómicas: Este patrón se centra en la operación de "Read-modify-write" orientado a computación paralela y su objetivo es prevenir "Data race" (donde los hilos compiten por el acceso a los datos) y pérdidas de valores (donde

los hilos acceden indiscriminadamente y alteran los datos sin un orden específico) para asegurar la integridad del resultado de las operaciones. Esto se logra creando orden de acceso a los datos de salida por medio de operaciones atómicas, evitando que dos hilos modifique un valor y dando acceso a un hilo hasta que la operación atómica del otro sea terminada (Nickolls. et al., 2008).

Maximizar la intensidad operacional

En la sesión anterior se definió la intensidad operacional como el número de operaciones realizadas por segundo por cada Byte de datos transferido entre las memorias. Por este motivo para maximizar este parámetro se debe aumentar el trabajo (cantidad de operaciones por segundo), reducir al mínimo posible el tráfico de datos a través de la memoria y hacer que ese tráfico sea lo más rápido posible. Para conseguir eso se deben seguir las siguientes prácticas (CUDA C Best Practices Guide, 2017; NVIDIA_OpenCL_BestPracticesGuide, 2016; OpenCL Optimization Guide, 2016).

Minimizar la transferencia de datos entre host y dispositivo: El ancho de banda dentro del dispositivo es mucho más alto que el ancho de banda entre dicho dispositivo y el host, por este motivo la heterogeneidad de un programa CUDA/OpenCL debe estar enfocada no exclusivamente a dividir lo que es paralelizable y lo que no, sino también a minimizar las conmutaciones entre host y dispositivo que impliquen transferencia de datos. En otras palabras cada paso de dato del host al dispositivo se debe justificar con una relación alta entre las operaciones a ejecutar y la cantidad de datos transferidos, de no ser así se debe evaluar la opción de ejecutar la tarea en el host así el proceso sea paralelizable.

Aumentar el ancho de banda en la transferencia de datos entre host y dispositivo: Adicional a minimizar la cantidad de datos transferido entre host y dispositivo se pueden utilizar técnicas y tecnologías recientes para reducir el tiempo que se toma dicha transferencia de datos. A continuación se listan las principales técnicas y/o tecnologías introducidas por CUDA y/o OpenCL o por los fabricantes de las GPUs para mejorar el ancho de banda entre host y dispositivo.

- Memoria “Pinned” (page-locked): Tanto CUDA como OpenCL permite hacer uso de memoria que no requiere hacer una búsqueda de una dirección física por cada palabra que vaya a ser transferida, sino que fija toda una página en una dirección física. El uso de esta memoria (pinned) aporta un significativo aumento en el ancho de banda en la transferencia de datos entre host y dispositivo. Para OpenCL se deben revisar los comandos Map/Unmap y para CUDA se deben revisar los comandos cudaHostAlloc y cudaMemcpyAsync.
- Solapamiento entre transferencia de datos y procesamiento: Una de las ventajas que presenta el hecho de hacer uso de la memoria “pinned” es la transferencia de datos de forma asincrónica que habilita la posibilidad

de solapar parcialmente la transferencia de datos con las tareas de cómputo, como se ve en la figura 6.1.

Figura 6.1 Solapamiento entre transferencia de datos y ejecución. Fuente: Autor.

- Acceso directo entre host y dispositivo: OpenCL en sus últimas versiones (desde la 2.0) presenta la posibilidad de usar una memoria virtual compartida (SVM - Shared Virtual Memory) que representa un espacio direccionado visible tanto para el host como para el dispositivo, similar a extender una porción de la memoria global hasta el host, lo que permite que los ítems de trabajo puedan acceder directamente a la memoria del host. Al igual que OpenCL, CUDA desde la versión 2.2 dispone de una característica que habilita los hilos (ítems para OpenCL) para acceder directamente a la memoria del host. Esta característica se llama Zero Copy. Algunos dispositivos soportan un modo de direccionamiento especial denominado Direccionamiento Virtual Unificado (UVA - Unified Virtual Addressing). Con UVA tanto la memoria del host como las memorias de todos los dispositivos instalados comparten un mismo espacio virtual direccionado.

Hacer uso eficiente de la estructura jerárquica de la memoria: Tanto CUDA como OpenCL presentan un modelo de memoria basado en una estructura jerárquica donde el tamaño de la memoria y su visibilidad es inversamente proporcional a la velocidad de acceso. Es decir la memoria más grande y de acceso general es la más lenta y la memoria más pequeña y de acceso restringido a sólo un ítem de trabajo es la más rápida. Si se organizan los principales tipos de memoria que presenta un dispositivo CUDA/OpenCL de mayor a menor velocidad en el acceso se verá algo así:

Para CUDA: Registros > Memoria compartida >> Memoria global
Para OpenCL: Memoria privada > Memoria local >> Memoria global

De acuerdo a lo anterior se debe minimizar los accesos a la memoria global y enfocarse en hacer uso eficiente de la memoria local (compartida para CUDA) y en la memoria de uso exclusivo de los hilos, de acuerdo como lo permita la naturaleza del algoritmo a implementar.

Hacer uso de patrones "coalesced" para acceder a la memoria global: La memoria global aporta una latencia tan alta que no es suficiente con minimizar el

acceso a esta sino que se debe pensar también en usar patrones de acceso que permitan leer o escribir en bloque varias de sus posiciones. Esos patrones son normalmente llamados “Coalesced”.

Las arquitecturas sobre las cuales se basan la mayoría de GPUs actuales permiten hacer transferencias de datos desde y hacia la memoria global en bloques hasta de 128-bytes en UNA SOLA TRANSACCIÓN.

El truco consiste en crear patrones donde los hilos accedan a posiciones de memorias consecutivas de tal forma que todas aquellas que pertenezcan a un mismo bloque puedan ser leídas/escritas en una sola transacción, a continuación se describen algunos de los más frecuentes patrones de accesos usados:

- Compacto sin corrimiento: Es un patrón donde la posición de memoria accedida corresponde exactamente al índice del hilo en alguna de sus dimensiones. Este es el patrón que más ancho de banda alcanza. En la figura 6.2 se ilustra como un conjunto de hilos puede acceder a la memoria global en una sola transacción debido a que todas las posiciones de memoria corresponden a una misma línea de caché.

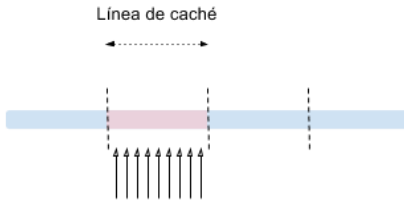


Figura 6.2 Patrón de acceso compacto sin corrimiento. Fuente: Autor.

- Compacto con corrimiento: Es un patrón donde la posición de memoria accedida corresponde al índice del hilo en alguna de sus dimensiones más un corrimiento específico. Este patrón presenta una leve disminución del ancho de banda con respecto al anterior, sin embargo esta reducción no es progresiva de acuerdo al valor del corrimiento sino que presenta un comportamiento cíclico. En la figura 6.3 se ilustra como un conjunto de hilos para acceder a la memoria global usando este patrón requiere dos transacción debido a que las posiciones de memoria corresponden a dos líneas de caché.

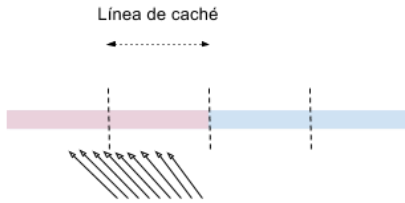


Figura 6.3 Patrón de acceso compacto con corrimiento. Fuente: Autor.

- Saltos: Mediante este patrón se accede a posiciones de memoria separadas por espacios constantes. En otras palabras es como si se accediera a la memoria en saltos donde la distancia entre salto y salto es la misma. El ancho de banda disminuye a medida que la separación entre posiciones de memoria aumenta. En la figura 6.4 se ilustra como un conjunto de hilos para acceder a la memoria global requiere tres transacción debido a que las posiciones de memoria están separadas ocupando tres líneas de caché.

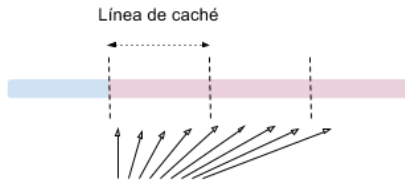


Figura 6.4 Patrón de acceso por saltos. Fuente: Autor.

- Aleatorio: La forma menos eficiente para acceder a la memoria global es aleatoriamente. Al no poder compactar los accesos a memoria se eleva drásticamente el número de transacciones necesarias. En la figura 6.5 se ilustra como un conjunto de hilos requiere varias transacciones para acceder de forma aleatoria a la memoria global debido a que las posiciones pertenecen a varias líneas de caché.

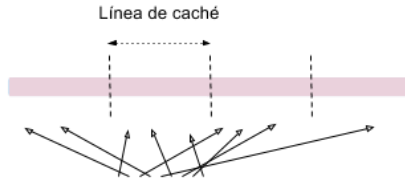


Figura 6.5 Patrón de acceso aleatorio. Fuente: Autor.

Compartir memoria entre hilos: En las sesiones anteriores se hizo hincapié en la relación inversamente proporcional entre la visibilidad de la memoria y su velocidad de acceso. De acuerdo a esto se podría pensar que lo ideal sería siempre usar la memoria de registro/privada por ser la que presenta menor latencia, sin embargo su reducido tamaño y su visibilidad exclusiva a un hilo limita la posibilidad de reuso de sus datos.

Por lo expuesto en el párrafo anterior la memoria exclusiva a cada hilo se debe restringir a usos donde definitivamente no es necesario compartir los datos con otros hilos, donde el hilo va a realizar más de una operación con los datos copiados y donde el tamaño de estos datos es adecuado para el tamaño de dicha memoria. En cualquier otro caso la memoria que debe tener prioridad en su uso deber ser la compartida por los siguientes aspectos:

- Presenta una latencia mucho menor a la memoria global.
- Evita la redundancia de acceso a la memoria global. Si varios hilos requieren un mismo dato de la memoria global, se accede una única vez para copiarlo a la memoria local y allí queda disponible para ser accedido por múltiples hilos.
- Facilita el acceso a la memoria global con patrones compactos. Si la naturaleza del algoritmo no facilita el acceso a la memoria global con un patrón compacto, se utiliza la memoria local como intermediaria para hacer una copia inicial en esta donde se utilice un patrón compacto, luego se reorganizan los datos y se ponen a disposición de los hilos de acuerdo a la naturaleza del algoritmo (se debe recordar que los patrones de acceso compacto se requieren para acceder a la memoria global mas no para acceder a la memoria local).

Evitar colisiones de bloque en el acceso a la memoria local: La estrategia usada por los dispositivos para alcanzar un alto ancho de banda en los accesos concurrentes a la memoria local es dividir dicha memoria en bloques de tal forma que aquellas posiciones que no correspondan al mismo bloque puedan ser accedidas simultáneamente. El acceso a la memoria local se realiza de forma simultánea siempre y cuando las posiciones accedidas no correspondan a un mismo bloque, si es así los accesos se serializan. En la figura 6.6 se puede notar como en el caso de la derecha dos hilos acceden simultáneamente a posiciones del mismo bloque lo que genera la colisión.

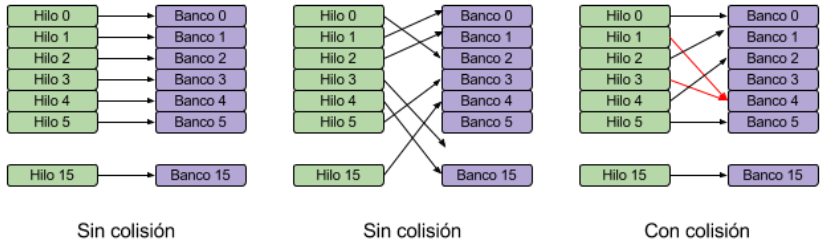


Figura 6.6 Colisión de bloque. Fuente: Autor.

El consejo para evitar colisiones es conocer la información referente a: - la forma como se asignan las palabras a los bancos (por ejemplo palabras consecutivas de 32 o 64 bits son asignadas a bancos consecutivos), - el número de bancos de memoria, - el ancho de banda por banco, - la forma como son agrupados los hilos para ser ejecutados, y usar dicha información para garantizar que dos o más hilos que se ejecutan concurrentemente no vayan a acceder a posiciones de memoria que correspondan al mismo banco de memoria.

Mejorar la configuración de ejecución: Uno de los factores determinantes para aumentar la intensidad operacional es asegurar un alto nivel de ocupancia de los recursos disponibles en los dispositivos. La ocupancia es una métrica que determina el porcentaje de uso del hardware disponible.

En gran parte la responsabilidad de mantener dicha ocupancia en un nivel alto reside en los parámetros de configuración de la ejecución, principalmente de la definición del tamaño de los espacios de trabajo. El tamaño de los bloques (CUDA) o grupos de trabajo (OpenCL) no debe ser definido exclusivamente por la naturaleza de la problemática y/o los datos a tratar; por el contrario se debe dar una alta prioridad a definir un tamaño basado en la forma como son ejecutados los hilos. A continuación se listan algunas recomendaciones para tener en cuenta en el momento de seleccionar el tamaño del grupo/bloque:

- Debe ser múltiplo del tamaño de los grupos que se crean para la ejecución (en términos de CUDA: warp), así se evitan grupos poco poblados que generan un desperdicio de cómputo.
- Debe ser mínimo de 64 hilos, pues se tomaría el riesgo de sub-utilizar en gran medida un multiprocesador (en términos de CUDA) o una unidad de cómputo (en término de OpenCL).
- No debe ser tan grande porque se aumenta la latencia en el momento de sincronización. Un tamaño adecuado debe estar entre 128 y 256 hilos.

Existen otras configuraciones a tener en cuenta como por ejemplo: - Evitar la definición de más de un contexto debido a que los recursos son asignados concurrentemente pero no pueden ser usados de la misma forma y se requiere

una conmutación entre contextos. - Evaluar la opción de kernels concurrentes (de la misma forma como se puede realizar transferencia de datos concurrentemente con ejecución, se puede ejecutar simultáneamente dos o más kernels).

Generar mejoras a nivel de instrucciones: Tener el conocimiento de la forma como se ejecutan las instrucciones en CUDA o en OpenCL permite tomar decisiones referentes a sustituir instrucciones por otras que generen el mismo resultado pero con menos costo computacional. Un caso para tener muy en cuenta son las operaciones con flotantes de precisión sencilla; estas operaciones presentan un alto desempeño, por este motivo se deben sustituir operaciones de alto costo computacional por aquellas basadas en flotantes de precisión sencilla, por ejemplo: reemplazar divisiones y módulos por corrimientos.

Tener en cuenta el control de flujo: Uno de los principales factores generadores de latencia puede llegar hacer un mal diseño de control de flujo, esencialmente en el manejo de las divergencias, tal como no evitar hilos divergente en un mismo grupo de ejecución o no evitar la sincronización de grupos/bloques de trabajo que posean hilos divergentes

7. MODELO DE PROCESAMIENTO PARALELO DE K-MERS SOBRE PLATAFORMAS HETEROGÉNEAS

7.1. Introducción al modelo

En los capítulos 3 y 4 se evidenció la disminución considerable de requerimientos de memoria que aportan los métodos de representación de lecturas en super k-mers y la partición del conjunto de datos de acuerdo a los minimizers canónicos / signatures; sin embargo estas ventajas obtenidas referentes a memoria se deben pagar de cierta forma en intensidad de cómputo en el proceso de identificación y obtención de super k-mers debido a la complejidad e intensidad de la tarea de encontrar y comparar el minimizer canónico / signatures de cada posible k-mer de cada lectura. El modelo propuesto en este documento pretende afrontar la intensidad de las tareas enunciadas en el párrafo anterior mediante procesamiento paralelo ejecutado en plataformas heterogéneas multi core / many core.

Este trabajo propone una metodología simplificada para modelar procesos sobre plataformas heterogéneas multi core / many core, basado exclusivamente sobre los 3 grandes retos que debe enfrentar este tipo procesamiento:

1. Estructuras de datos eficientes que permitan hacer buen uso de las plataformas many core a pesar de sus limitaciones de memoria y sus latencias.

2. Distribución adecuada de tareas para el procesamiento en las plataformas multi core o many core de acuerdo a sus exigencias de memoria y de cómputo, y bajo la necesidad de minimizar la transferencia de datos entre plataformas.

3. Modelo de procesamiento paralelo masivo diseñado bajo el enfoque de maximizar el speed-up con patrones de paralelización eficientes y maximizar la intensidad operacional mediante la disminución del acceso a las memorias que presentan mayor latencia. Este modelo se define mediante una descripción

general y 3 componentes: - Uso de la estructura jerárquica de la memoria, - Espacios indexados de procesamiento y - Algoritmos de paralelización masiva definidos mediante funciones indexadas con respeto a los espacios de procesamiento.

Homologación de términos OpenCL / CUDA

Teniendo en cuenta que el modelo está planteado para plataformas many core, tomando como referencia GPUs, y que los dos principales estándares y/o plataforma de cálculo paralelo para éstos dispositivos son CUDA y OpenCL, se considera pertinente antes de iniciar con la descripción del modelo, realizar una homologación de términos entre estos dos ambientes de tal forma que lo expuesto en el modelo sea fácilmente interpretado para ser implementado bajo cualquiera de estos dos estándares y/o plataformas.

	OpenCL	CUDA	Modelo
Jerarquía de procesos	Ítem de trabajo	Hilo	Hilo
	Grupo de trabajo	Bloque	Espacio local
	NDRange	Malla	Espacio global
Jerarquía de memoria	Memoria privada	Memoria local	Memoria privada
	Memoria local	Memoria compartida	Memoria local
	Memoria global	Memoria global	Memoria global

Tabla 7.1 Tabla de homologación de términos CUDA y OpenCL

7.2. Estructuras de datos

Representación de lecturas

El modelo representa cada una de las bases de una lectura mediante su representación numérica base 4:

A -> 0, C -> 1, G -> 2, T -> 3.

Por ende una lectura será una secuencia numérica base 4, donde cada número corresponde a cada base.

Ejemplo: CTAAACGGTCAATG -> 13000122310032

Representación vectorial de una lectura: Una lectura es representada mediante un vector cuyos elementos corresponden al equivalente numérico base 4 de cada base, la longitud del vector será el tamaño de la lectura.

Read[ ]:

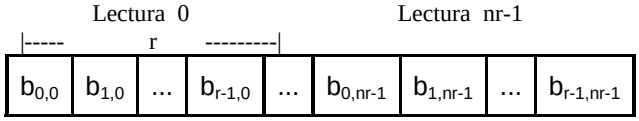
b_0	b_1	...	b_{r-1}

r-> longitud de una lectura
Read[i] = bi -> representación numérica (base 4) de la base i de la lectura R.

El modelo soportará lecturas con longitud hasta de 1024 bases. Las razones para fijar este valor y los beneficios que aporta son explicados en la estructura de datos CISK.

Representación vectorial unificada de un conjunto de lecturas: La representación de un conjunto de lecturas se realiza mediante un único vector constituido por la concatenación de los vectores que representan cada una de las lecturas. Debido a que todas las lecturas de un conjunto poseen la misma longitud el direccionamiento de este vector unificado no presenta ninguna complejidad.

Reads_Set[ ]:



r-> longitud de una lectura, nr-> número de lecturas
*Reads_Set[j*r+i] = bi,j -> representación numérica (base 4) de la base i de la lectura j*

Representación de m-mers canónicos

Retomando los conceptos definidos en los capítulos anteriores, un m-mer es un trozo de lectura de longitud m (conformada por m bases). El modelo representa un m-mer mediante un valor decimal equivalente a la conversión de la representación numérica base 4 de dicha secuencia a la representación numérica base 10. Si se tiene el m-mer = b_0, b_1 ,..., b_{m-1} su valor decimal será:

$$M\text{-}mer_{10} = b_0 \times 4^{(m-1)} + b_1 \times 4^{(m-2)} + \ldots + b_{m-1} \times 4^{(0)}$$

Ecuación 7.1

Por ejemplo, el valor decimal del m-mer CATG es (1 x 64 + 0 x 16 + 3 x 4 + 2 x 1) : 78

El complemento inverso de un m-mer es el resultado de invertir la secuencia y reemplazar cada base por su complemento (A < - > T, C < - > G), por ejemplo el complemento inverso del m-mer GCTA es TAGC. La representación decimal del complemento inverso de un m-mer se puede obtener de forma directa a partir del m-mer modificando la ecuación definida arriba. Si se tiene el m-mer = b0, b1 ,..., bm-1 el valor decimal de su complemento inverso será:

$$Rcm\text{-}mer_{10} = \sim b_{m-1} \times 4^{(m-1)} + \ldots + \sim b_1 \times 4^{(1)} + \sim b_0 \times 4^{(0)}$$

Ecuación 7.2

Por ejemplo, el valor decimal del complemento inverso del m-mer GCTA es (3 x 64 + 0 x 16 + 2 x 4 + 1 x 1) : 201

Un m-mer canónico es el menor entre el m-mer y su complemento inverso tomando como valor de comparación sus representaciones numéricas (en cualquier base), es decir su peso lexicográfico. A continuación se expone un ejemplo para ilustrar el proceso de obtención de un m-mer canónico:

M-mer (m = 5): TGACG
Representación decimal del m-mer: 902
Complemento inverso del m-mer: CGTCA
Representación decimal del complemento inverso: 436
El m-mer canónico es CGTCA debido a que tiene el menor valor numérico.

Los m-mers de una lectura son todas las subsecuencias de longitud m que se pueden obtener de ésta, por ejemplo los m-mers (con m = 4) de la lectura GGCTATCGAC son: GGCT, GCTA, CTAT, TATC, ATCG, TCGA, CGAC. Por tal motivo la representación numérica decimal de los m-mers de una lectura será una secuencia de números decimales donde cada uno de ellos corresponde a un m-mer. Por ejemplo los m-mers de la lectura mencionada serán: 167, 156, 115, 205, 54, 216, 97. Esto igual aplica para los m-mers canónicos de una lectura, con la diferencia de que a cada uno de los m-mers se le obtiene la representación decimal de su canónico.

Representación vectorial de los m-mers canónicos de una lectura: Como se mencionaba arriba, si a cada uno de los m-mers de una lectura se le aplica el procedimiento de m-mer canónico y se obtiene su representación decimal, el resultado será una secuencia de números donde cada uno de ellos será el menor valor entre cada m-mer y su complemento inverso. El modelo almacena esa secuencia de números en un vector donde cada elemento será la representación numérica decimal de los m-mers canónicos de una lectura; la longitud de este vector corresponde al número de m-mers obtenidos de una lectura (r - m + 1), donde r es la longitud de la lectura.

CM-mers[ ]:

cmr_0	cmr_1	...	cmr_{nmr-1}

nmr -> número de m-mers por lectura; nmr = r-m+1
CM-mers[i] = cmri -> representación numérica (base 10) del m-mer canónico i de la lectura procesada.

Representación de Super k-mers: CISK (Compact and Indexed representation of Super k-mers)

Debido a que el proceso de obtención de super k-mers es una tarea que se puede realizar por tandas (no se requiere cargar todo el conjunto de datos a la memoria, ni se requiere mantener todo el resultado en la memoria), resulta ser un proceso con alta exigencia de cómputo más no de memoria, normalmente las herramientas seriales o paralelizadas en CPU utilizan buffer temporales de tamaño adecuado para la memoria, los cuales se van llenando a medida que se van identificando los super k-mers y tan pronto se llenan se realiza su escritura en disco.

En el caso de la implementación en dispositivos many core, las cosas no resultan tan sencillas desde el punto de vista de la memoria, primero que todo porque estos dispositivos son limitados en dicho recurso y segundo porque el procesamiento en tandas implicaría realizar varias transferencias de host a memoria del dispositivo lo que generaría una alta latencia. De acuerdo a esto se debe buscar la forma de cargar todo el conjunto de datos al dispositivo en un solo llamado al kernel o por lo menos en la mínima cantidad de llamadas posibles. El problema no radica en el tamaño del archivo de entrada sino en el de salida, pues ya no serán las lecturas sino todos los super k-mers encontrados y sus semillas (y aunque los super k-mers no presentan tanta redundancia como los k-mers si generan un considerable aumento en el conjunto de datos).

Para afrontar este problema el modelo propone una nueva estructura de datos que permite representar los super k-mers y minimizers canónicos / signatures de un conjunto de lecturas con una baja demanda de memoria haciendo uso de un arreglo de bits donde compacta el valor decimal de cada minimizer canónico / signature junto a una representación indirecta (mediante índices y longitudes) de cada súper k-mer, de tal forma de que en la GPU se detectan los super-kmers y sus semillas, más no se extraen de forma explícita.

Esta estructura de datos facilita la división del proceso de obtención y distribución de super k-mers en subprocesos de los cuales unos tienen una alta exigencia de cómputo y moderado requerimiento de memoria (obtención de la estructura de datos mencionada) y otros tienen una baja exigencia de cómputo pero un alto requerimiento de memoria (hacer explícito la obtención y distribución de los super k-mers). Esta división de procesos convierte al problema de obtención y distribución de k-mers en un candidato adecuado para ser solucionado mediante computación heterogénea donde las plataformas multi core no presentan alta capacidad de cómputo pero ofrecen gran recurso de memoria y las plataformas many core no presentan buenas características de memoria pero ofrecen una muy alta capacidad de cómputo.

El modelo propone usar palabras de 32 bits para representar cada uno de los super k-mers de una lectura de hasta 1024 bp, con k hasta de 129 bases y minimizers canónicos / signature hasta de 7 bases (m = 7). La estructura de dicho arreglo es la siguiente: - Los primeros 14 bits (los de mayor peso) son

usados para alojar el valor decimal del minimizer canónico / signature, por esta razón el modelo soporta un valor de m hasta de 7 bases. - Los siguientes 10 bits almacenan el índice que indica la posición de la base donde inicia el súper k-mer dentro de la lectura, por esta razón el modelo soporta lecturas hasta de 1024 bp. - Los últimos 8 bits son usados para almacenar la longitud del súper k-mer; de acuerdo al concepto de súper k-mer, su máxima longitud será cuando su primer k-mer presente en su extremo derecho el minimizer canónico / signature que comparte, y su último k-mer lo presente en su extremo izquierdo, de esta forma la máxima longitud será dos veces k menos m. Teniendo en cuenta que con 8 bits se representa una longitud de súper k-mer de hasta 256 bases, la máxima longitud de k-mer soportada por el modelo será de 129 bases. Ver figura 7.1.

Figura 7.1 CISK: Estructura de datos para representar super k-mers y minimizers canónicos / signatures. Fuente: Autor.

Los valores límites de longitudes de lectura, de k-mers y de minimizers canónicos / signatures soportados por el modelo fueron seleccionados bajo las siguientes argumentaciones: - Las lecturas de los principales secuenciadores (tales como Illumina, Roche, Ion, Solid) presentan un tamaño máximo alrededor de 700 bp (el modelo garantiza longitudes hasta de 1024 bp). - Los valores de k normalmente usados son 21, 31, 41, 51, 61, 71 y 81, el máximo valor recomendado es de 127 y el valor ideal está alrededor de 71 (Chikhi & Medvedev, 2014; Wick, 2016). - El valor de m determina el número de particiones (archivos en disco) en las cuales se distribuyen los super k-mers (# particiones = 4^m), con un número de bases de hasta 7 el modelo asegura un número máximo de particiones de 16384, reduciendo así el tamaño de cada uno de los archivos y la exigencia de memoria al procesarlos (Deorowicz, et al., 2015; Kokot, Długosz & Deorowicz, 2017).

Representación vectorial de los CISKs de una lectura: La estructura de datos CISK garantiza que el modelo pueda representar los super k-mers y minimizers canónicos / signatures de una lectura en un vector de skr elementos, donde skr es el número de super k-mers hallados en la lectura y cada elemento corresponde a un arreglo CISK (32 bits).

CISKs_R[ ]:

skr -> Cantidad de super k-mers hallados en una lectura
CISKs_R[i]= ciski representación del súper k-mer i (y su minimizer) de la lectura correspondiente.

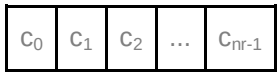
Representación vectorial unificada de los CISKs de un conjunto de lecturas. Debido a que la cantidad de super k-mers hallados en cada lectura puede ser diferente de lectura a lectura, la longitud de cada vector que almacena los CISKs de cada lectura de un conjunto, no es un valor constante y puede variar de vector a vector. Debido a lo anterior, el planteamiento de una estructura de datos que unifique todos los CISKs de un conjunto de lecturas puede resultar complejo por la dificultad de direccionamiento o ineficiente por la existencia de elementos vacíos.

El modelo propone una representación vectorial unificada para todos los CISKs de un conjunto de lecturas, basado únicamente en dos vectores con el objeto de reducir la complejidad del direccionamiento y de evitar la ineficiencia: Un vector constituido por la concatenación de todos los vectores que almacenan los CISKs de cada lectura y un segundo vector que contiene el número de super k-mers hallados en cada lectura.

CISKs_Set[ ]:

C [ ] -> Counters [ ]
nr = Número de lecturas en el conjunto
CISKs_Set [sumatoria(Counter[0] hasta Counter[j-1]) + i] = cisk -> Arreglo de bits CISK ubicado en la posición i del subvector j (representa el súper k-mer i y su minimizer de la lectura j).

Counters[ ]:



nr = Número de lecturas en el conjunto
Counters[i] = ci -> número de super k-mers hallados en la lectura i.

7.3. Distribución de tareas Multi core / Many Core

Inicialmente se identifican y clasifican los procesos y subprocesos involucrados en la obtención y distribución de los super k-mers de un conjunto de lecturas haciendo uso de la estructura de datos descrita anteriormente. La clasificación se realiza de acuerdo a la estimación de su exigencia de cómputo y de memoria

basado en el análisis de sus algoritmos. Se puede notar que gracias a la estructura de datos propuesta, los subprocesos que más exigencia de cómputo presentan son a su vez los que demandan una cantidad de memoria baja (lo cual los hace apropiados para ser acelerados en una plataforma many core). Ver tabla 7.1

Procesos	Subprocesos	Requerimiento de cómputo	Requerimiento de memoria
Carga y conversión numérica de las lecturas	Lectura del archivo FASTA	Bajo	Bajo
	Conversión numérica de las bases	Bajo	Bajo
Búsqueda, identificación y representación compacta e indexada de super k-mers	Obtención del valor decimal de los m-mer de las lecturas	Alto	Moderado
	Búsqueda de los minimizers canónicos / signatures de los k-mers	Muy alto	Bajo
	Identificación y representación compacta e indexada de los super k-mers de las lecturas	Alto	Bajo
Obtención explícita y distribución de super k-mers	Obtención explícita de super k-mers	Bajo	Moderado/Alto
	Distribución de super k-mers	Bajo	Moderado/Alto

Tabla 7.2 Requerimientos computacionales de los procesos y subprocesos en la obtención y distribución de super k-mers

A continuación los procesos y subprocesos se agrupan de acuerdo a su conveniencia para su ejecución en CPU o en una plataforma many core, de la siguiente manera: Aquellos que presentan un alto requerimiento de cómputo pero una moderada o baja exigencia de memoria son candidatos para ser acelerados mediante una plataforma many core, y aquellos que presentan una

exigencia baja de cómputo pero una alta demanda de memoria son candidatos para ser ejecutados en CPU.

De acuerdo a lo anterior se plantea un modelo general de procesamiento heterogéneo que consiste en un flujo de procesos que indica cuáles tareas deben realizarse en CPU y cuales en una plataforma many core para obtener y distribuir los super k-mers de un conjunto de lecturas. Ver figura 7.2.

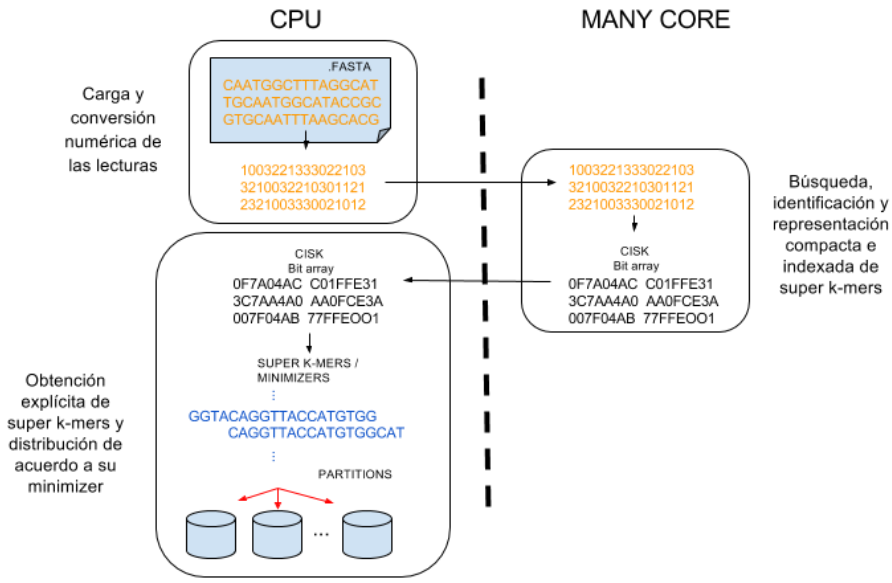


Figura 7.2 Modelo general de procesamiento heterogéneo para la obtención y distribución de super k-mers en plataformas heterogéneas. Fuente: Autor.

7.4. Modelo de procesamiento paralelo masivo sobre plataformas many core

Descripción general del modelo

El modelo describe las directrices para hallar los super k-mers de un conjunto de lecturas y representarlos mediante la estructura de datos descrita anteriormente (CISK), usando procesamiento paralelo masivo sobre plataformas many core (tomando como referencia GPUs, pero extensible a cualquier otra plataforma many core). El modelo se diseñó bajo el enfoque de reducir el uso de memoria y de maximizar la intensidad operacional disminuyendo el acceso a la memoria local. Para cumplir con este propósito se plantean cuatro etapas; una inicial y una final para la transferencia de datos entre memoria global y local y dos etapas principales encargadas de la obtención de m-mers canónicos y de la búsqueda de super k-mers. Ver figura 7.3.

Uso de la estructura jerárquica de la memoria

En la figura 7.4 se puede observar el uso que el modelo hace de la estructura jerárquica de la memoria para almacenar y procesar los datos. Se pueden detallar el flujo de los datos a través de cada una de las etapas: los datos de entrada, los de salida y los de uso interno. Un detalle particular de la figura 7.4 son los fondos en cada una de las memorias: la memoria global tiene un fondo sólido (sin renglones ni cuadrículas) que indica que es una sola memoria para todos los hilos de procesamiento, la memoria local tiene un fondo con rectángulos que indica que existe una memoria local por cada espacio local indexado de procesamiento y que cada uno de esos datos existe en cada una de las memorias locales, finalmente la memoria privada tiene un fondo cuadriculado que indica que existe una memoria privada por cada hilo y que cada uno de esos datos existe en cada una de las memorias privadas.

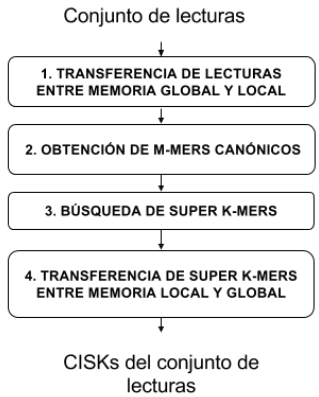


Figura 7.3 Etapas de procesamiento del modelo. Fuente: Autor.

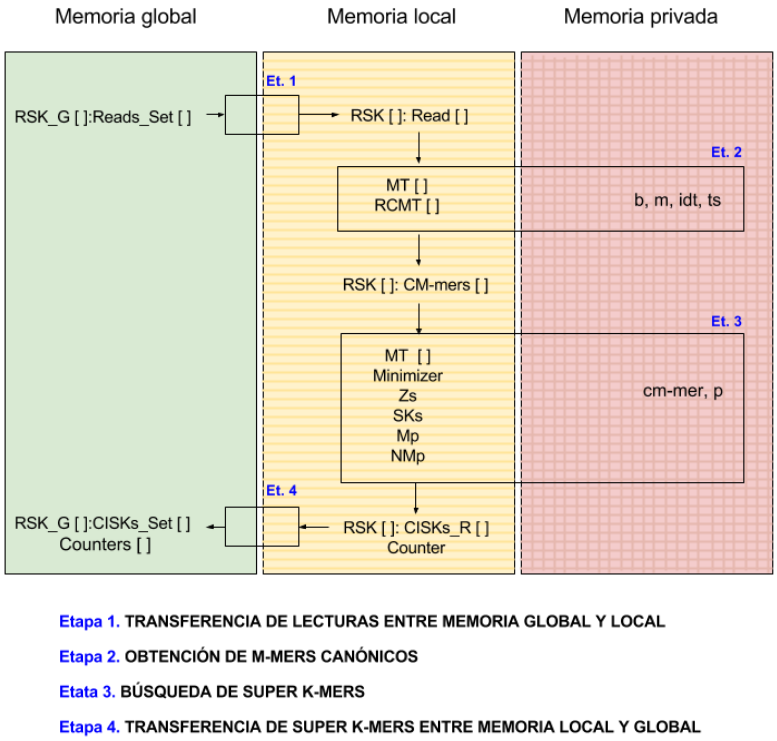


Figura 7.4 Uso de la estructura jerárquica de la memoria. Fuente: Autor.

En la tabla 7.3 se hace una breve descripción los datos mencionados en la figura 7.4:

Tipo de memoria	Espacio	Uso
Global	RSK_G[]	Arreglo que contiene el vector Reads_Set[] o el vector CISKs_Set[].
	Counters[]	Arreglo que contiene el número de super k-mers hallados en cada una de las lecturas.
Local	RSK[]	Arreglo que contiene el vector Read[] o el vector CM-mers[] o el vector CISKs_R[].
	MT[]	Arreglo que se usa internamente en dos etapas: En la etapa 2 contiene el valor decimal del "actual" m-mer calculado por cada uno de los tiles; en la etapa 3 contiene los mínimos de cada uno de los bloques.
	RCMT[]	Arreglo que contiene el valor decimal del complemento inverso del "actual" m-mer calculado por cada uno de los tiles.
	Minimizer	"Actual" minimizer canónico.
	Counter	Conteo de super k-mers que se van encontrando.
	Zs	Índice del inicio de la "actual" zona de evaluación.
	SKs	Índice del inicio del "actual" súper k-mer.
	Mp	Índice del inicio de la "actual" semilla (minimizer canónico / signature)
	NMp	Índice del inicio de la "nueva" semilla (minimizer canónico /

		signature) Memoria privada
Privada	b	Valor numérico base 4 de la base leída por el hilo
	m	Longitud del m-mer
	idt	Índice del tile
	ts	Tamaño del tile
	cm-mer	Valor decimal del m-mer canónico leído por el hilo
	p	Posición del último m-mer canónico leído por el hilo (dentro del vector Cm-mers[])

Tabla 7.3 Espacios de memoria usados por el modelo

Nota: Las notaciones definidas acá son usadas en las imágenes y algoritmos del capítulo.

Espacios indexados de procesamiento

En el dispositivo many core a usar para la implementación del modelo se debe definir un espacio global de procesamiento bidimensional conformado por una cantidad de espacios locales unidimensionales de tal forma que cada uno de estos se encargue de procesar una lectura del conjunto. Ver figura 7.5

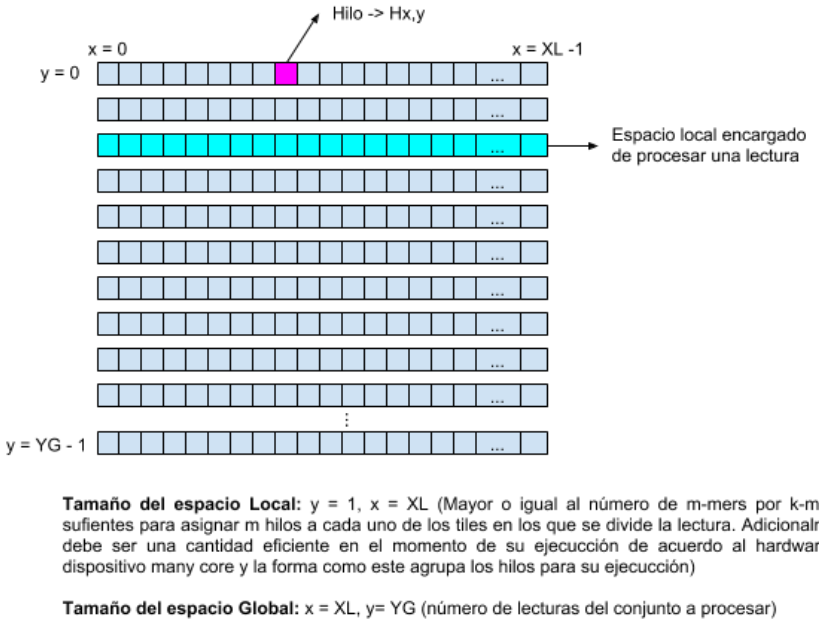


Figura 7.5 Espacios indexados de procesamiento. Fuente: Autor.

Nota: Las notaciones definidas acá son usadas en las imágenes y algoritmos del capítulo.

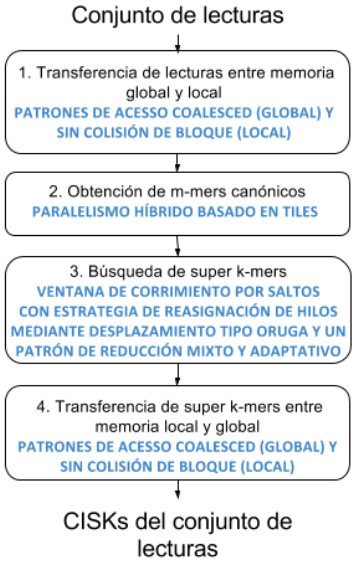


Figura 7.6 Estrategias de paralelización usadas por cada una de las etapas de procesamiento para la obtención y distribución de super k-mers. Fuente: Autor.

Descripción general de los algoritmos de paralelización masiva usados por etapa

Como se mencionó anteriormente, el proceso de obtención y distribución de super k-mers se dividió en 4 etapas, las cuales fueron diseñadas con el objeto de reducir el uso de memoria y de maximizar la intensidad operacional disminuyendo el acceso a la memoria local. En la figura 7.6 se menciona la estrategia de paralelización usada en cada una de esas etapas.

El transporte de datos entre la memoria global y la local y viceversa se lleva a cabo empleando patrones de acceso coalesced que facilitan la transferencia de bloques de memoria en una sola transacción. Para la obtención de m-mers canónicos se utiliza un algoritmo de procesamiento paralelo con granularidad híbrida basado en tiles y para la búsqueda de los super k-mers se utiliza un algoritmo de ventana de corrimiento por saltos con una estrategia de reasignación de hilos por desplazamiento tipo oruga y un patrón de reducción mixto y adaptativo. A continuación se explican cada una de estas estrategias haciendo énfasis en las etapas 2 y 3, debido a que es allí donde se presenta el mayor aporte mediante la propuesta de patrones nuevos de paralelización para plataformas many core.

Algoritmos de la Etapa 1 y 4 - Transferencia de datos entre la memoria global y memoria local y viceversa, usando patrones coalesced y sin colisión de bloque

El modelo se diseña bajo el enfoque de incrementar la intensidad operacional mediante la reducción de la latencia haciendo uso eficiente de la estructura jerárquica de memoria de las plataformas many core. Se plantea reducir el acceso a las memorias con mayor costo de latencia tal como la global y promover el uso de memorias más rápidas como la local o en el mejor de los casos, la privada. Para cumplir este fin, el modelo propone una etapa inicial y una final de transferencia de datos de la memoria global a la local y viceversa, con el objeto de que la memoria global solo sea accedida dos veces: un acceso al inicio del proceso con el objeto de copiar las lecturas a la memoria local y dejarlas allí disponibles para las dos etapas principales de procesamiento (obtención de k-mers canónicos y búsqueda de super k-mers), y un acceso al final para transferir los resultados (super k-mers en formato CISK) desde la memoria local hacia la global.

Lecturas de la memoria global a la local: Cada uno de los espacios de procesamiento local se encarga de hacer la extracción de una lectura del vector Reads_set [ ] almacenado en el espacio de la memoria global RSK_G [ ] y de transferirla hacia la memoria local.

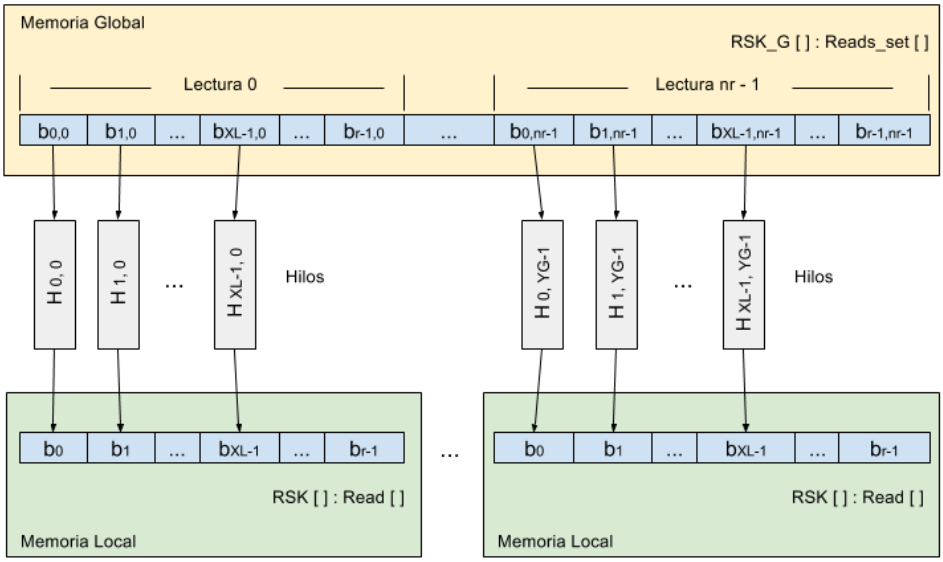


Figura 7.7 Transferencia de lecturas entre la memoria global y la local. Fuente: Autor.

Debido a que el tamaño del espacio local se define de tal forma que solo hayan m hilos por cada uno de los tile en los que se divide una lectura, y los tiles tienen tamaños superiores a m, el número de hilos de un espacio local sería inferior al número de bases de una lectura y por ende se debe hacer un proceso iterativo por

segmentos de la lectura, donde en cada iteración todos los hilos de un espacio local de procesamiento acceden de forma compacta (hilos contiguos, posiciones de memoria contiguas) a un segmento de la lectura y lo transfieren a la memoria local. La forma como se escribe en la memoria local también es contigua sin riesgo a que haya conflicto de bloque debido a que cada posición del vector Read [ ] almacenado en el espacio RSK [ ] de la memoria local tiene 32 bits correspondiente a un bloque en la mayoría de los dispositivos many core actuales. Ver Figura 7.7

Super k-mers de la memoria local a la global: Cada uno de los espacios de procesamiento local copia el vector CISKs_R [ ] resultante de cada lectura y lo sobreescribe en el segmento donde se encontraba la lectura en el vector de entrada en la memoria global (el espacio siempre va a ser suficiente debido a que nunca van a haber más super k-mers que bases en una lectura). La transferencia se realiza de igual forma que la descrita arriba, los hilos acceden de forma compacta tanto a la memoria local como a la global. Si el número de hilos por espacio local indexado de procesamiento es menor al número de super k-mers hallados en esa lectura el proceso de transferencia se hace de manera iterativa, de no ser así el proceso se realiza en una solo iteración tal como lo muestra la figura 7.8.

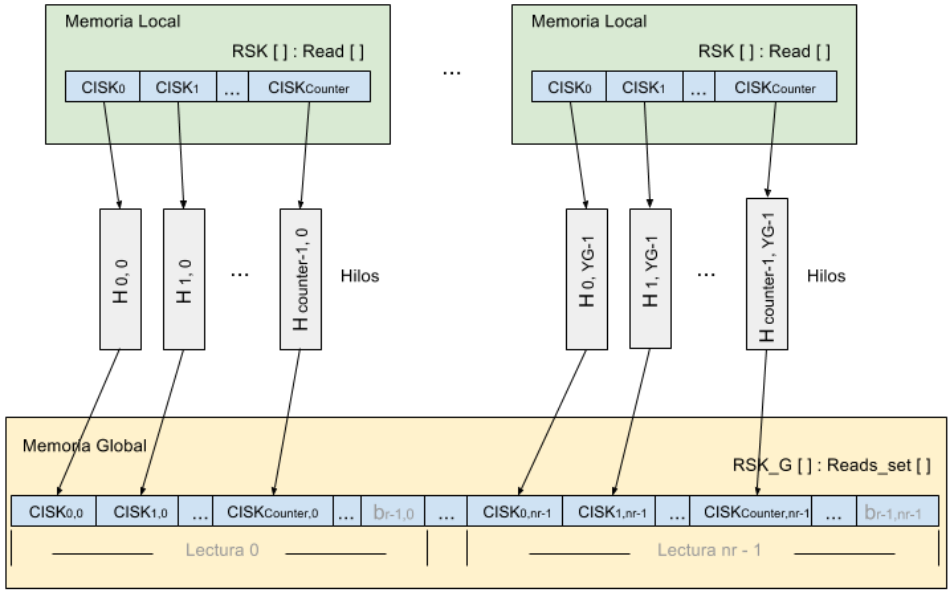


Figura 7.8 Transferencia de super k-mers entre la memoria local y la global. Fuente: Autor.

Por cada una de las lecturas se tiene un valor de conteo que representa el número de super k-mers hallado en dicha lectura. Este valor se encuentra en la memoria local. Un hilo por cada espacio local indexado de procesamiento se encarga de copiar dicho valor y escribirlo en la memoria global para conformar el vector

Counters [ ], el cual contiene el número de super k-mers hallados en cada una de las lecturas. Ver Figura 7.9

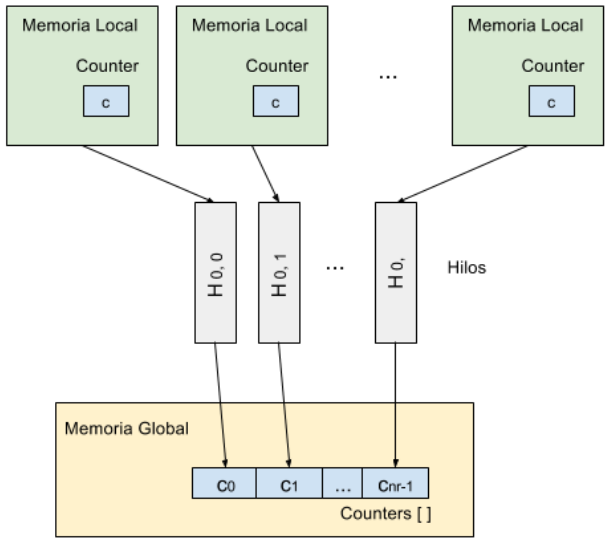


Figura 7.9 Transferencia de conteos entre la memoria global y la local. Fuente: Autor.

Algoritmo de la Etapa 2 – Patrón de paralelización para la obtención de m-mers canónicos (con o sin restricción de signature) mediante granularidad híbrida y tiles.

Esta etapa tiene como objeto hallar los m-mers canónicos de un conjunto de lecturas teniendo o no en cuenta las restricciones de una signature (las cuales se mencionan más adelante). Como se había mencionado con anterioridad, un m-mer canónico es el menor entre el m-mer y su complemento inverso tomando como valor de comparación su peso lexicográfico. La forma convencional para hallar dichos pesos en base 10 (valor decimal) es mediante las ecuaciones convencionales expresadas al inicio del capítulo (ecuaciones 7.1 y 7.2)

En un dispositivo many core ideal (hilos físicos infinitos, memoria ilimitada y no latencia) cada una de estas ecuaciones se podrían paralelizar de forma relativamente sencilla asignando un hilo para computar cada base. Sin embargo en un dispositivo real una implementación de esa forma conllevaría a un sobre-acceso de cada una de las bases de la lectura hasta m - 1 veces, lo que sería catastrófico para la intensidad operacional, adicional a esto se tendría una granularidad demasiado fina que sería ineficiente teniendo en cuenta la forma como se ejecutan los hilos.

Por lo expuesto en el párrafo anterior la metodología propuesta en este modelo para obtener los m-mer canónicos de un conjunto de lecturas no se limita a paralelizar el algoritmo convencional, sino que redefine el algoritmo para

postular un nuevo patrón de paralelización pensado en plataformas many core, basado en tiles con granularidad híbrida, de tal forma que se pueda usar un paralelismo masivo para obtener el primer m-mer de cada tile (mediante las ecuaciones convencionales) y un paralelismo moderado para obtener el resto de m-mers (mediante una estrategia de rollo que permite el reúso de los resultados), evitando así la redundancia en el acceso a cada uno de los elementos del vector de entrada; y se complementa con una estrategia de direccionamiento de los datos de salida que permita el reúso del arreglo donde se almacenaron los datos de la entrada. Ver figura 7.10.

División en Tiles: El algoritmo de paralelización propone la división del conjunto de los m-mers canónicos a hallar de una lectura en sub-conjuntos (tiles), de tal forma que el proceso se divide en subprocesos (que se ejecuten simultáneamente), y cada uno de éstos se encargue de calcular uno de los subconjuntos. El número de subconjuntos se selecciona de tal forma que para cada espacio local de procesamiento haya hilos suficientes para asignarle a cada subproceso una cantidad igual a m (longitud de los m-mers) sin que sobren más de m-1 hilos. En la figura 7.11 se ilustra la división del vector de entrada Read [ ] y del vector de salida CM-mers [ ].

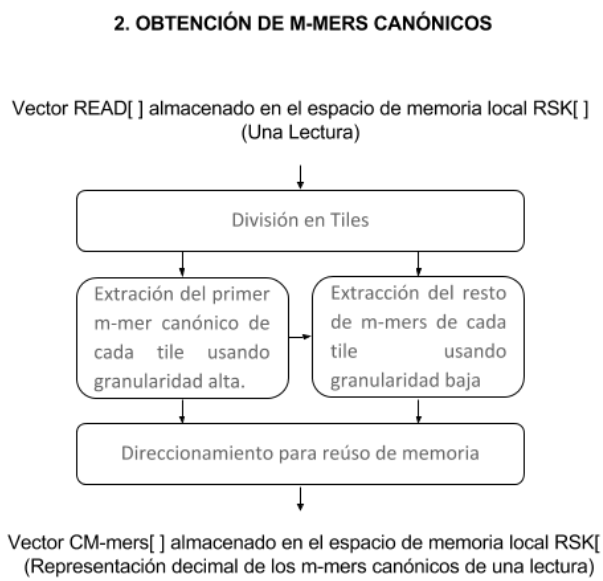
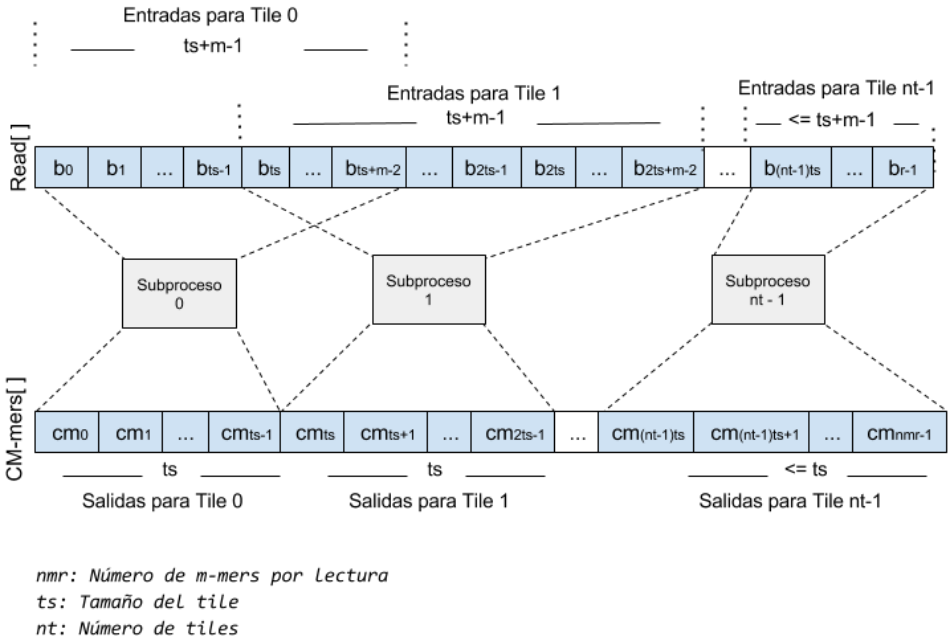


Figura 7.10 Diagrama general de los algoritmos usados en la etapa de obtención de m-mers canónicos. Fuente: Autor.



nmr: Número de m-mers por lectura
ts: Tamaño del tile
nt: Número de tiles

Figura 7.11 División en Tiles de los vectores Read[ ] y CM-mer[ ] para el proceso de obtención de m-mers canónicos (Las notaciones son las mismas usadas en los algoritmos 1 y 2). Fuente: Autor.

Estrategia de direccionamiento de la salida para habilitar el reúso de memoria: Los limitados recursos de memoria de la mayoría de las plataformas many core no solo obligan a usar estructuras de datos eficientes sino a tomar medidas en tiempo de ejecución que minimicen las exigencias de este recurso, como por ejemplo tener estricto control sobre los datos que ya no se requieran para desecharlos o reescribir sobre los espacios que estén ocupando éstos.

El modelo utiliza una estrategia para el reúso del espacio de memoria de la entrada para almacenar la salida: reescribe los m-mers canónicos de una lectura sobre el espacio usado para almacenar dicha lectura. Este proceso se convierte en un reto de direccionamiento teniendo en cuenta que una base puede hacer parte de m m-mers y se puede cometer el error de que un hilo reescriba una base que aún sea requerida por otro hilo para calcular otro m-mer canónico.

Teniendo en cuenta que el modelo utiliza una estrategia de tiles, donde los subprocesos para hallar cada uno de los tiles se ejecutan simultáneamente, pero internamente cada uno de éstos calcula sus m-mers de forma secuencial (ver algoritmo en la siguiente sección) se podría pensar que debido a que la primera base del m-mer calculado en un subproceso ya no hace parte del siguiente m-mer (contiguo a la derecha), esa sería la base candidata para ser sobre-escrita con el valor decimal del m-mer canónico. Sin embargo si se analiza la forma como se emplean las bases para cada uno de los tiles, se puede notar que las primeras m - 1 bases usadas para un tile son las últimas bases requeridas para el tile anterior;

esto derrumba la propuesta de usar la primera base de cada m-mer para sobre-escribir allí el valor decimal de canónico.

De acuerdo a lo anterior pareciera imposible encontrar una estrategia de sobre-escritura que no borrara una base que luego vaya a ser usada. Sin embargo si se analiza el algoritmo empleado internamente en cada tile (en la siguiente sección se explica de forma detallada), se puede notar que éste emplea todas las m bases del m-mer solo para calcular el primer m-mer canónico de cada tile, para el resto de m-mers de cada tile utiliza una función que solo requiere la última base del m-mer a computar y el valor decimal del anterior m-mer (ya calculado) y el de su complemento inverso. Esto se puede interpretar como que a diferencia de las m -1 bases del primer m-mer a computar en cada tile, el resto de bases sólo son empleadas una única vez. Como consecuencia a lo anterior la estrategia de sobre-escritura del actual modelo propone escribir el valor decimal del m-mer canónico en la posición donde se encontraba almacenada la última base de dicho m-mer. En la figura 7.12 se puede observar cómo se emplea un único arreglo llamado RSK [ ], para almacenar tanto el vector Read [ ], como el vector CM-mers [ ].

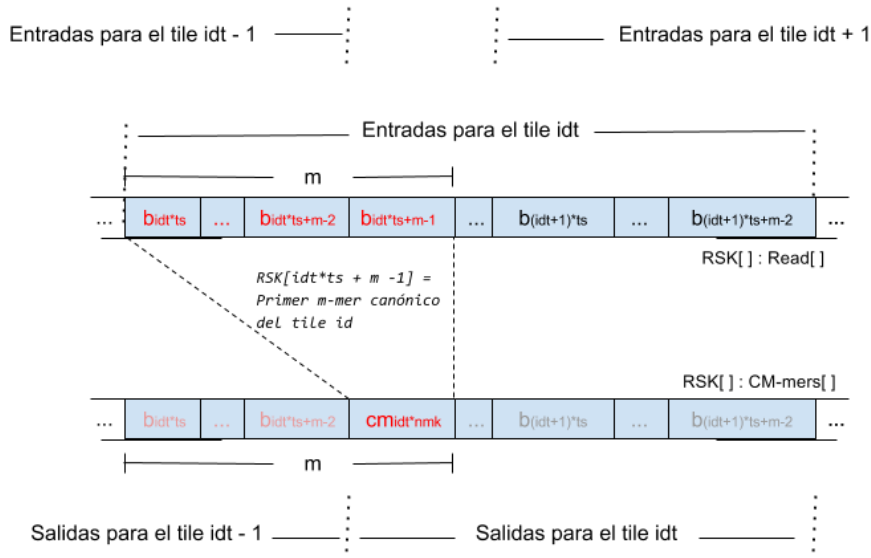


Figura 7.12 Estrategia de direccionamiento de la salida para habilitar el reúso de memoria (Las notaciones son las mismas usadas en los algoritmos 1 y 2). Fuente: Autor.

Obtención del primer m-mer canónico por tile - Granularidad alta (m hilos por tile): Mediante un paralelismo de alta granularidad se obtiene el valor decimal del primer m-mer canónico de cada tile. Por cada cada tile se emplea un grupo de m hilos. Cada uno de los m hilos asignados a cada tile se encarga de leer y procesar (mediante corrimientos, negaciones y operaciones atómicas OR) solo una base (de las primeras m bases) para implementar las ecuaciones

convencionales para hallar el valor decimal de un m-mer y el de su complemento inverso (Ecuaciones 7.1 y 7.2)

A continuación se reasignan los hilos para que haya solo un hilo por tile, el cual lee los dos valores decimales obtenidos anteriormente (m-mer y su complemento inverso) y les aplica la función min o la función sig.

Si este algoritmo se utiliza sólo para hallar los m-mers canónicos de un conjunto de lecturas o si se utiliza para luego hallar los super k-mers de las lecturas bajo el criterio de minimizer canónico, se aplica la función min que simplemente obtiene el valor menor de las dos entradas. Pero si el algoritmo es usado para posteriormente hallar los super k-mers de un conjunto de lecturas bajo el criterio de signatures, se aplica la función sig la cual antes de sacar el mínimo de los valores aplica las restricciones de una signature (que no comience por AAA o ACA o que no tenga AA en cualquier ubicación excepto en su inicio). Si ni el m-mer ni su complemento inverso cumplen con las restricciones de una signature la función devuelve el valor decimal equivalente al mayor valor decimal que puede tener un m-mer con esa longitud, incrementado en uno.

El mismo hilo realiza la escritura del valor decimal del m-mer canónico obtenido (con o sin restricciones de signtarure) sobre-escribiendo el arreglo donde se encuentran los datos de entrada mediante la estrategia de direccionamiento explicada posteriormente. Ver algoritmo 7.1 y su ilustración en la figura 7.13.

Algoritmo 7.1: Obtención del primer m-mer canónico por tile (Paralelismo de granularidad alta)

---

ts -> Tamaño del tile
m -> Tamaño del m-mer
x -> Índice x del hilo
RSK[] -> Arreglo en memoria local donde las lecturas son almacenadas y los m-mers canónicos son re-escritos.
MT[] -> Arreglo en memoria local donde se almacena el "actual" m-mer de cada tile.
RCMT[] -> Arreglo en memoria local donde se almacena el complemento inverso del "actual" m-mer de cada tile.

Bloque de funciones ejecutado por cada uno de los m hilos de cada tile de cada lectura
{

- ***idt = x/m***

  Mediante una división entera, cada hilo obtiene el índice del tile al cual corresponde

- ***b = RSK [idt*ts + x%m]***

  Cada hilo del tile copia una base (su representación base 4) de la memoria local a la privada.

- ***MT[idt] = OR Atómico (MT[idt] , (b << (m-(x%m)-1)*2))***
  RCMT[idt] = OR Atómico (RCMT[idt] , (((~b) & 3) << ((x%m)*2)))

  Mediante negaciones lógicas, corrimientos y operaciones atómicas OR cada uno de los m hilos del tile computa la base que ha leído para implementar en conjunto las ecuaciones que postulan que dado el m-mer = b0, b1 ,..., bm-1 su valor decimal y el de su complemento inverso será:
  $m\text{-}mer_{10} = b_0x4^{(m-1)} + b_1x4^{(m-2)} +...+ b_{m-1}x4^{(0)}$
  $rcm\text{-}mer_{10} = \sim b_{m-1}x4^{(m-1)} + \ldots + \sim b_1x4^{(1)} + \sim b_0x4^{(0)}$.

}

Sincronización Local

Bloque de funciones ejecutado por un hilo en cada tile de cada lectura
{

- ***RSK[idt*ts + m -1] = min/sig (MT[idt], RCMT[idt])***

  El hilo asignado al tile aplica la función min[1] o sig[2] a los valores decimales del m-mer y el de su complemento inverso y sobreescribe el resultado en la lectura de entrada en la posición que corresponde a la última base del m-mer que se acaba de computar.

  [1] Si este algoritmo se utiliza sólo para hallar los m-mers canónicos de un conjunto de lecturas o si se utiliza para luego hallar los super k-mers de las lecturas bajo el criterio de minimizer canónico, se aplica la función min que simplemente obtiene el valor menor de las dos entradas.
  [2] Si este algoritmo se utiliza para hallar los super k-mers de un conjunto de lecturas bajo el criterio de signatures, se aplica la función sig la cual antes de sacar el mínimo de los valores aplica las restricciones de una sígnature (que no comience por AAA o ACA o que no tenga AA en cualquier ubicación excepto en su inicio). Si ni el m-mer ni su complemento inverso cumplen con las restricciones de una signature la función devuelve el valor decimal equivalente al mayor valor decimal que puede tener un m-mer con esa longitud, incrementado en uno.

}

Sincronización Local

En la figura 7.14 se ilustra un ejemplo del algoritmo usado para hallar el primer m-mer canónico de cada tile (sin restricciones de signature). En el ejemplo se muestra como se halla el primer m-mer (con m = 4) del tile número 6 de una lectura de 90 bases que ha sido divida en 8 tiles (los primeros 7 tiles de 12 bases y el último de 6) de tal forma que usando un espacio local de 32 hilos se podrán asignar m hilos (4 hilos) a cada uno de los 8 tiles.

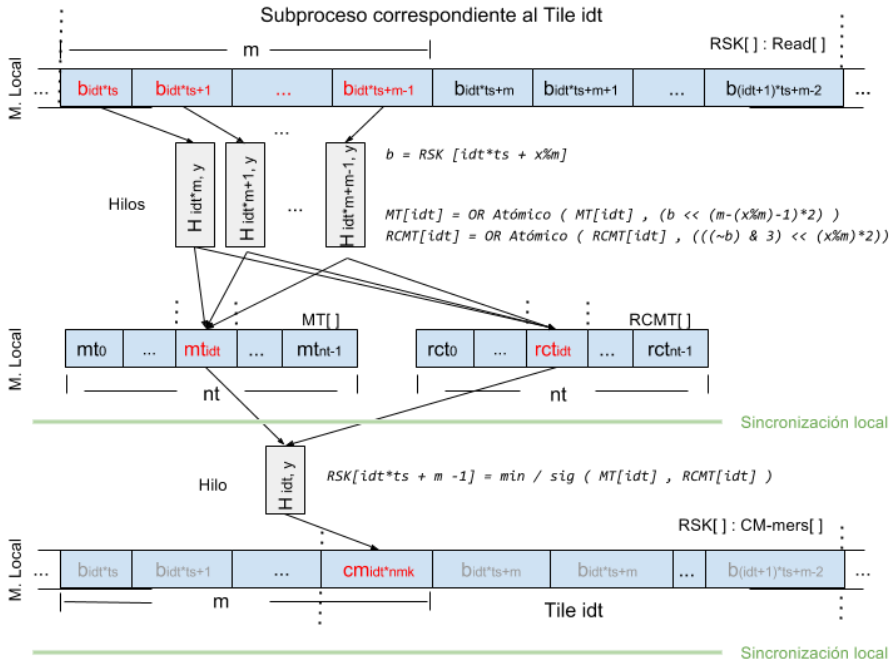


Figura 7.13 Ilustración del algoritmo de obtención del primer m-mer canónico. Las convenciones, siglas y nomenclatura en general son las mismas usadas en el algoritmo 7.1. Fuente: Autor.

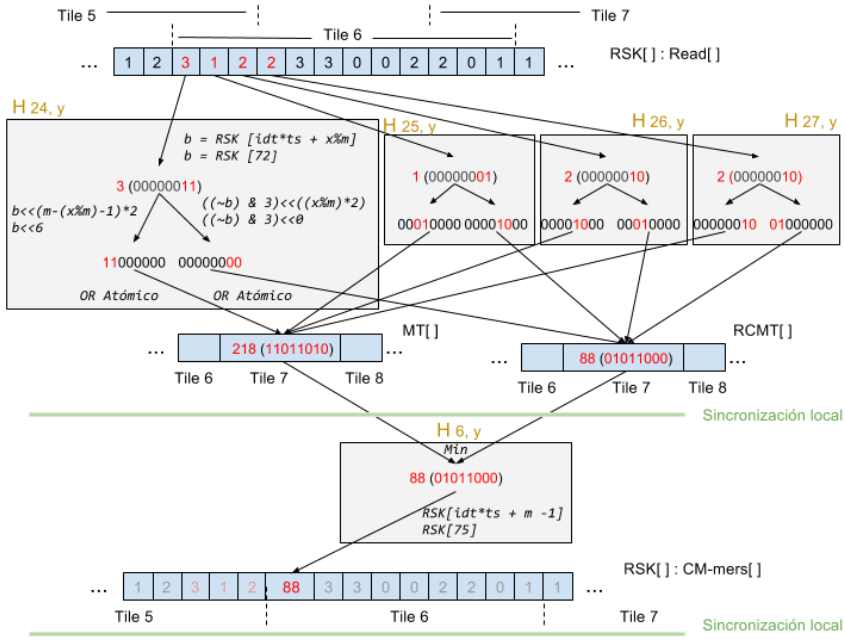


Figura 7.14 Diagrama de un ejemplo de uso del algoritmo de obtención del primer m-mer canónico por tile. Fuente: Autor.

Obtención del resto de m-mers canónicos por tile - Granularidad media (1 hilo por tile): El resto de m-mers canónicos de cada tile se obtienen usando un paralelismo de granularidad media para evitar el sobre-acceso a la memoria Local. Si el resto de m-mers canónicos en cada tile se obtuvieran de igual forma que el primero, cada valor del vector Read [ ] alojado en memoria local sería accedido hasta por m hilos, lo que sería catastrófico para la intensidad operacional. En la estrategia propuesta a continuación cada valor del vector Read [ ] es accedido solo una vez, sin que eso genere un sacrificio en la velocidad de obtención de m-mers. Esto se consigue mediante una estrategia de rollo que re-usa el valor decimal del m-mer previamente calculado y el de su complemento inverso, requiriendo así, únicamente la última base del m-mer a computar tal como se describe a continuación:

Si se tiene el valor decimal de un m-mer (para este caso el antiguo) y el de su complemento inverso,

$$m\text{-}mer\_o_{10} = b\_o_0 \times 4\text{^}(m-1) + b\_o_1 \times 4\text{^}(m-2) + \ldots + b\_o_{m-1} \times 4\text{^}(0)$$
$$rcm\text{-}mer\_o_{10} = \sim b\_o_{m-1} \times 4\text{^}(m-1) + \ldots + \sim b\_o_1 \times 4\text{^}(1) + \sim b\_o_0 \times 4\text{^}(0)$$

Ecuación 7.3a

El valor decimal del nuevo m-mer (m-mer contiguo a la derecha en la lectura) y el de su complemento inverso se puede expresar en función del antiguo m-mer y de la última base del nuevo m-mer (b_n_{m-1}) de la siguiente forma:

$$m\text{-}mer\_n_{10} = (b\_o_0 \times 4\text{^}(m-1)) \times 0 + (b\_o_1 \times 4\text{^}(m-2) + \ldots + b\_o_{m-1} \times 4\text{^}(0)) \times 4 + b\_n_{m-1}$$
$$rcm\text{-}mer\_n_{10} = \sim b\_n_{m-1} + (\sim b\_o_{m-1} \times 4\text{^}(m-1) + \ldots + \sim b\_o_1 \times 4\text{^}(1)) / 4 + (\sim b\_o_0 \times 4\text{^}(0)) \times 0$$

Ecuación 7.3b

Para implementar estas ecuaciones solo basta con que un hilo por cada tile acceda al vector Read [ ] y copie a su memoria privada únicamente la última base del m-mer a calcular, compute el valor decimal del m-mer anterior y el de su complemento inverso de acuerdo a la ecuaciones, adicione la lectura copiada y obtenga los dos valores decimales, el del nuevo m-mer y el de su complemento inverso.

A continuación, el mismo hilo aplica la función min o sig a los valores decimales del m-mer y el de su complemento inverso (de acuerdo a las mismas condiciones expuestas en el algoritmo del primer m-mer por tile) y lo sobre-escribe en el arreglo donde se encuentran los datos de entrada mediante la estrategia de direccionamiento explicada posteriormente. Este procedimiento se repite para el resto de m-mers de cada tile Ver algoritmo 7.2 y su ilustración en la figura 7.15.

Algoritmo 7.2: Obtención del resto de m-mers canónicos por tile (Paralelismo de granularidad media)

idt -> Índice del tile calculado por cada hilo en el algoritmo anterior
ts -> Tamaño del tile
m -> Tamaño del m-mer
x -> Índice x del hilo
RSK[] -> Arreglo en memoria local donde las lecturas son almacenadas y los m-mers canónicos son re-escritos.
MT[] -> Arreglo en memoria local donde se almacena el "actual" m-mer de cada tile.
RCMT[] -> Arreglo en memoria local donde se almacena el complemento inverso del "actual" m-mer de cada tile

Bloque de funciones ejecutado por un hilo en cada tile de cada lectura
{

- ***c = MT[idt]***
  d = RCMT[idt]

  El hilo asignado al tile lee el valor decimal del primer m-mer del tile y el de su complemento inverso (de la memoria local a la privada).

  Bloque de funciones que se repite para las condiciones (z=0; z<(ts-1); z++) y se rompe si se intenta leer una base en una posición fuera de la lectura.
  {

  - ***b = RSK [idt*ts + m + z]***

    El hilo asignado al tile copia de la memoria local a la privada únicamente la última base (su representación base 4) del m-mer a calcular.

  - ***c = ((c & (2^(2m-2)-1)) << 2)|b***
    d = ((d>>2) & (2^(2m-2)-1)) | (((~b) & 3)<<((m-1)*2))

    Utilizando únicamente el valor numérico base 4 de la última base del m-mer a calcular (b_n_{m-1}) y el valor decimal del m-mer anterior y el de su complemento inverso (los cuales ya se encuentran en la memoria privada), el hilo asignado al tile implementa las ecuaciones que postulan que dado el m-mer anterior m-mer_o = b_o_0 ,b_o_1 ,..., b_o_{m-1} el valor decimal del siguiente m-mer y el de su complemento inverso será:
    m-mer_n_{10} = (b_o_0x4^(m-1))x0 + (b_o_1x4^(m-2) +...+ b_o_{m-1}x4^(0))x4 + b_n_{m-1}
    rcm-mer_n_{10} = ~b_n_{m-1} + (~b_o_{m-1}x4^(m-1) +…+ ~b_o_1x4^(1))/4 + (~b_o_0x4^(0))x0.

  - ***RSK[idt*ts + m + z] = min / sig (c, d)***

    El hilo asignado al tile aplica la función min o sig a los valores decimales del m-mer y el de su complemento inverso (de acuerdo a las mismas condiciones expuestas en el algoritmo del primer m-mer por tile) y sobreescribe el resultado en la lectura de entrada en la posición que corresponde a la última base del m-mer que se acaba de computar.

  }

}

Sincronización Local

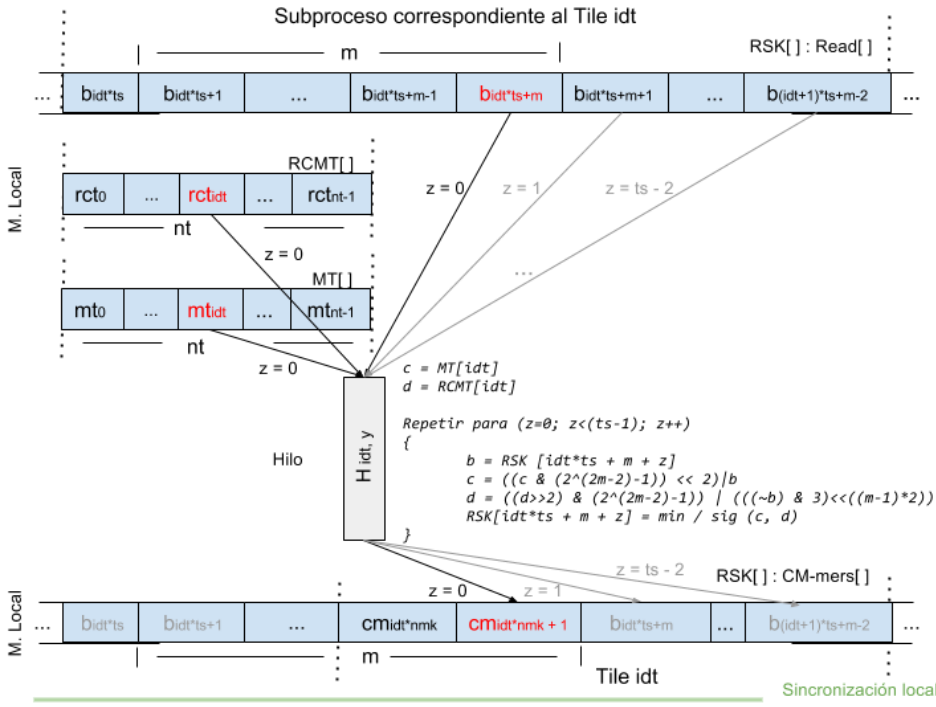


Figura 7.15 Ilustración del algoritmo de obtención del resto de m-mers de cada tile. Las convenciones, siglas y nomenclatura en general son las mismas usadas en el algoritmo 7.2. Fuente: Autor.

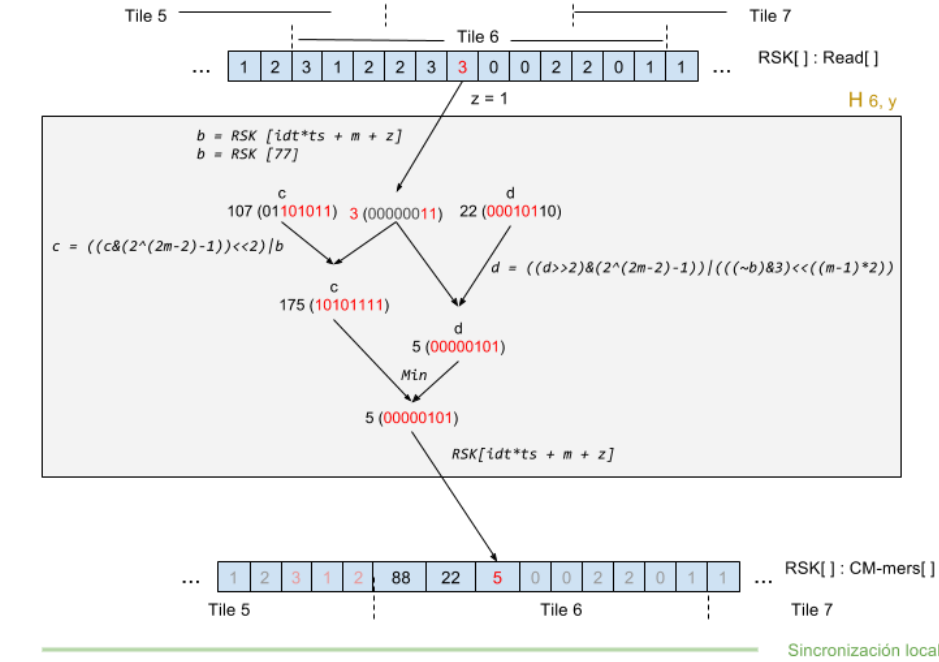


Figura 7.16 Diagrama de un ejemplo de uso del algoritmo de obtención del resto de m-mers canónicos por tile. Fuente: Autor.

En la figura 7.16 se retoma el ejemplo anterior (figura 7.14) y se ilustra cómo se hallaría el tercer m-mer canónico del tile procesado (sin restricciones de signature).

Algoritmo de la etapa 3 – Patrón de paralelización para la búsqueda de super k-mers mediante una ventana de corrimiento por saltos y reducción mixta y adaptativa.

Como se explicó en uno de los capítulos anteriores, un super k-mer en una lectura es la sub-secuencia que contenga todos los k-mers contiguos que presenten la misma semilla. El objeto de esta etapa es identificar cada uno de los super k-mers en todas las lecturas para representarlos y almacenarlos de forma compacta e indexada mediante la estructura de datos CISK.

De acuerdo al concepto definido en el párrafo anterior el algoritmo básico para encontrar los super k-mers de una lectura es evaluar cada uno de los posibles k-mers de la lectura, encontrar su semilla y comparar las semillas entre sí para identificar los k-mers contiguos que presenten la misma semilla. En un dispositivo many core ideal (hilos físicos infinitos, memoria ilimitada y no latencia) este algoritmo se podría implementar asignando un hilo para evaluar cada k-mer o inclusive un hilo por cada uno de los m-mers de los k-mers. Sin embargo en un dispositivo real una implementación de esa forma conllevaría a un sobre-acceso de cada una de las m-mers canónicos de la lectura hasta nmk - 1 (nmk -> número de m-mers por k-mers), lo que sería catastrófico para la intensidad operacional, adicional a esto se tendría una granularidad demasiado fina que sería ineficiente teniendo en cuenta la forma como se ejecutan los hilos y por último se generaría una exigencia elevada de memoria para almacenar temporalmente los k-mers y sus semillas.

De acuerdo a lo anterior la estrategia propuesta por este modelo re-plantea el algoritmo para crear un nuevo patrón de paralelización pensado en plataformas many core. Se postula el uso de una ventana de corrimiento por saltos junto con una estrategia de reasignación de hilos por desplazamiento tipo oruga que evita que los m-mers almacenados en la memoria local sean accedidos más de una vez, y un patrón de reducción mixta y adaptativo. Ver Figura 7.17

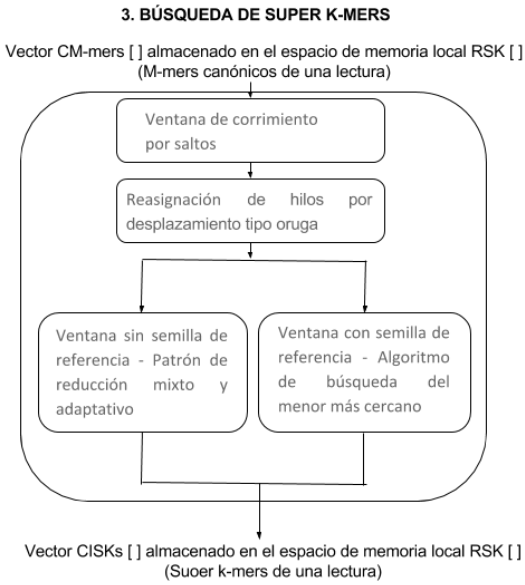


Figura 7.17 Diagrama general de los algoritmos usados en la etapa de búsqueda de super k-mers. Fuente: Autor.

Ventana de corrimiento por saltos. La búsqueda de los super k-mers en cada una de las lecturas se realiza mediante una ventana de procesamiento paralelo de tamaño nmk que se desplaza por saltos, de tal forma que sólo evalúa las zonas que determinan el inicio y/o el final de un super k-mer.

La estrategia de desplazamiento de la ventana define una zona de evaluación inicial fija correspondiente a los m-mers canónicos del primer k-mer de la lectura (que corresponderá al k-mer inicial del primer super k-mer de la lectura); la evaluación de esta zona tendrá como propósito encontrar el minimizer canónico / signature de este k-mer, que a su vez será la semilla del primer super k-mer de la lectura. Ver la figura 7.18.

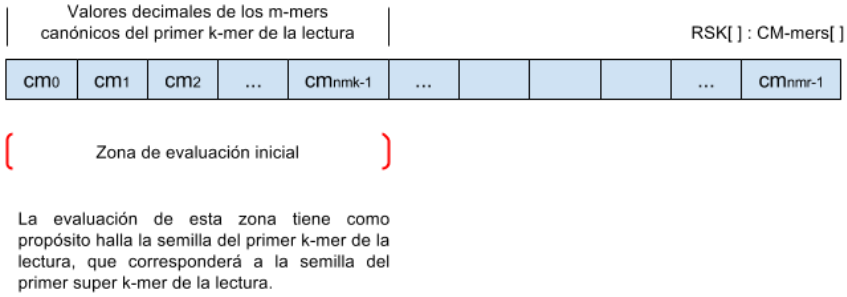


Figura 7.18 Posición inicial de la ventana. Fuente: Autor.

Las siguientes zonas definidas por la ventana tendrán una posición y una finalidad sujeta a la ubicación de la semilla del super k-mer "actual" dentro de la zona anterior. Si la semilla del super k-mer actual se encuentra en una posición diferente a la primera de la zona anteriormente evaluada, la ventana se desplazará a una nueva zona que inicia en dicha semilla (ver figura 7.19). El propósito de esta evaluación será definir el final del actual super k-mer, ya sea porque se encuentra un m-mer canónico menor o porque no se encuentra y entonces esta zona representará el último k-mer del actual super k-mer (debido a que la actual semilla se "sale" del siguiente k-mer). La forma de procesar esta zona corresponde al algoritmo de búsqueda del menor más cercano que se explicará más adelante.

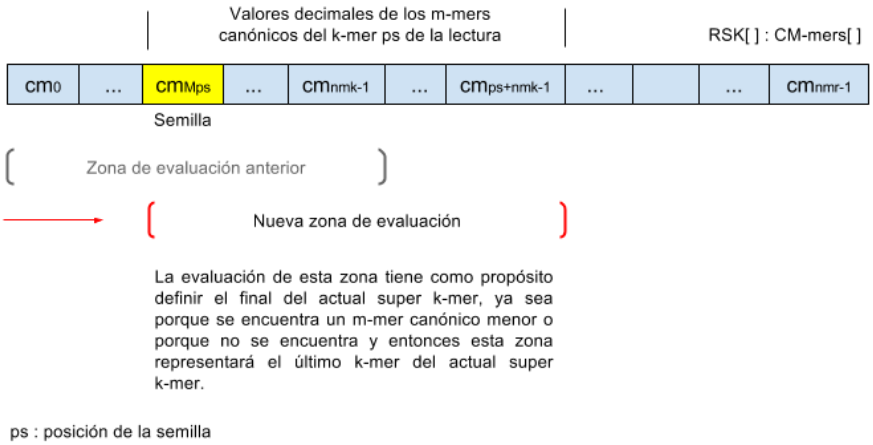


Figura 7.19 Desplazamiento de la ventana cuando la semilla no se encuentra en la primera posición de la anterior zona. Fuente: Autor.

Si la semilla del super k-mer actual se encuentra en la primera posición de la zona evaluada anteriormente, significa que esa zona corresponde al último k-mer del super k-mer actual y por ende el k-mer contiguo a la derecha será el inicio del siguiente super k-mer; por este motivo la nueva zona de evaluación será ese siguiente k-mer (ver figura 7.20) y su propósito será hallar su semilla (que será la misma del nuevo super k-mer). La forma de procesar esta zona corresponde al patrón de reducción mixta árbol / atómica, explicada más adelante.

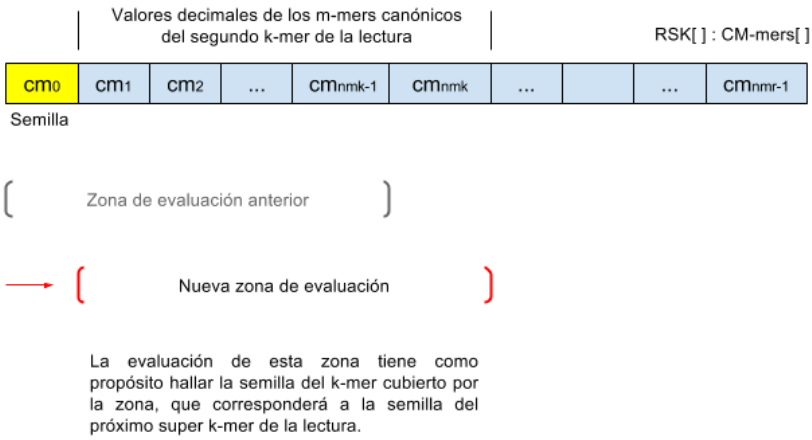


Figura 7.20 Desplazamiento de la ventana cuando la semilla se encuentra en la primera posición de la anterior zona. Fuente: Autor.

El desplazamiento de la ventana se repite (en cualquiera de las dos opciones explicadas dependiendo la zona anterior evaluada), hasta que llegue al final de la lectura. Aunque en las figuras 7.19 y 7.20 se tomó como ejemplo de zona anterior la primera zona de la lectura, se debe tener en cuenta que esto aplica para cualquier zona evaluada.

Estrategia de reasignación de hilos mediante desplazamiento tipo oruga: Cada una de las zonas de evaluación que va seleccionando la ventana se procesa de forma paralela con una granularidad alta generada por la asignación de un hilo por cada m-mer canónico que conforma la zona. A pesar de lo anterior la exigencia de hilos a nivel de toda la lectura no es alta, debido a que estas zonas se van ejecutando una a una de forma secuencial y por este motivo se pueden reutilizar los hilos.

Al analizar la forma como se desplaza la ventana de evaluación se puede notar que las zonas procesadas siempre se solapan. Las posiciones compartidas por dos zonas podrían ser solo una en el caso de que la semilla se encuentre en la última posición de la zona anterior, pero también podría llegar a ser de nmk -1 posiciones en caso de que la semilla se encuentre en la primera posición de la zona anterior.

Las posiciones que se solapan de una zona a otra podrían jugar un papel a favor o en contra de la intensidad operacional dependiendo de la forma como se reasignan los hilos de una zona a la siguiente zona. Si los hilos se reasignan en el mismo orden de zona a zona esto conllevaría a que las posiciones solapadas sean nuevamente accedidas, sin embargo si se utiliza una estrategia donde los hilos que se hayan encargado de posiciones que van a pertenecer a la nueva zona,

conserven esa asignación, se evitaría el sobre acceso a estas posiciones debido a que los hilos conservan en su memoria privada los valores ya leídos.

Para cumplir lo propuesto en el párrafo anterior el modelo propone una estrategia de asignación de hilos basada en el desplazamiento tipo oruga (sistema de desplazamiento común en los tanques de guerra), donde los hilos se reasignan mediante un sistema de rotación tal como se muestra en la figura 7.21.

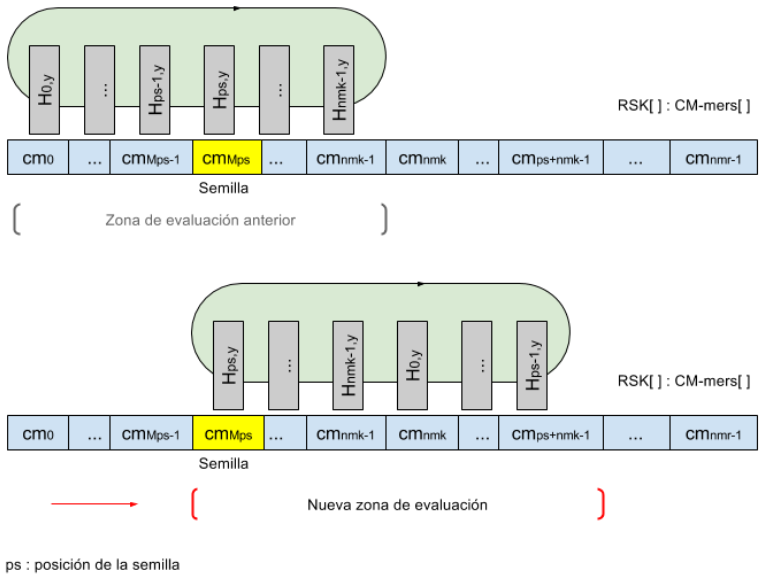


Figura 7.21 Asignación de hilos mediante desplazamiento tipo oruga. Fuente: Autor.

Zona sin semilla de referencia - Patrón de reducción mixta y adaptativa: Si después de haber evaluado una zona, la semilla se encuentra al inicio de esta, la ventana se desplaza una posición a la derecha de tal forma que inicia en la posición siguiente a la semilla. En este caso la nueva zona de evaluación no cuenta con una semilla de referencia y representa el primer k-mer de un nuevo super k-mer. El procesamiento que se le debe realizar a esta nueva zona es obtener el mínimo dentro de los valores decimales de los m-mers canónicos que la conforman, el cual representará la semilla del k-mer evaluado y del super k-mer que inicia ahí. Teniendo en cuenta que el tamaño de la ventana se define por el número de m-mers en un k-mer, el cual corresponde al tamaño del k-mer menos la longitud del m-mer más 1 (nmk = k - m + 1) y que los valores comunes de k varían entre 21 y 127, el patrón de reducción usado debe tener un buen comportamiento tanto para muy pocos elementos (alrededor de 21) como para una cantidad moderada de éstos (alrededor de 127).

De acuerdo a lo anterior el modelo propone un patrón de reducción mixto y adaptativo. Es mixto porque usa una estructura de árbol de tres niveles, pero sus ramas convergen mediante operaciones atómicas, y es adaptativo porque el número de ramas del segundo nivel (a las que convergen los elementos iniciales)

se adapta de acuerdo al tamaño de la zona, de tal forma que el número de elementos por bloque a reducir en cada uno de los niveles permanezca en una rango no tan variado y con niveles bajos (alrededor de 8), para que así las operaciones atómicas tengan un rendimiento alto y homogéneo en cada uno de los niveles del árbol (Lars Nyland & Stephen Jones, 2013). Es posible que varios elementos de la zona tengan el mismo valor mínimo, en ese caso se selecciona el que presente una mayor posición, con el fin de favorecer la construcción de super k-mers más extensos. Ver algoritmo 7.3 y figura 7.22.

Algoritmo 7.3: Evaluación de zona sin semilla de referencia (Patrón de reducción mixto y adaptativo)

---

bs -> Tamaño del bloque.
m -> Tamaño del m-mer.
cm -> Valos decimal de un m-mer canónico
nmk -> Número de m-mers por k-mer.
x -> índice x del hilo.
p -> Posición del elemento del vector Cm-mer[] que el hilo está procesando.
Mp -> Posición de la "actual" semilla.
RSK[] -> Arreglo en memoria local donde se almacenan los m-mers canónicos.
MT[] -> Arreglo en memoria local donde se almacena el mínimo de cada bloque.
nb -> Número de bloques (si k <= 36 -> nb = 6; si k <= 64 -> nb = 8; si k <= 100 -> nb = 10; si k > 100 -> nb = 12)
Minimizer -> Valor decimal de un minimizer

Bloque de funciones ejecutado por nmk hilos por lectura
{
*Bloque de funciones que se ejecuta si p < sz (*Aquellos hilos cuyo elemento procesado en la zona anterior está en una posición menor al inicio de la actual zona - *desplazamiento tipo oruga)*
{

- ***p = p + nmk***

  Se actualiza la *posición del elemento del vector Cm-mer[] que cada hilo va a leer y procesar (desplazamiento tipo oruga)*

- ***cm = RSK[p + m - 1]***

  Cada hilo lee su nuevo elemento a procesar del vector Cm-mer[] (el offset m-1 se debe a que el vector Cm-mer[] se encuentra almacenado en el arreglo RSK[] pero corrido m-1 posiciones debido a la estrategia de sobre-escritura)

}

- ***MT[(x%bs] = Min atómico (MT[x%bs], cm)***

Mediante una operación modular se hace que los hilos que pertenezcan al mismo bloque realicen una operación atómica "mínimo" con la misma ubicación del vector MT[]. De esta forma se reduce cada bloque a su elemento menor.

}

Sincronización Local

Bloque de funciones ejecutado por nb hilos por lectura
{

- ***Minimizer = Min atómico (MT[x], Minimizer)***

  Mediante una operación atómica se reduce cada uno de los bloques a su elemento menor

}

Sincronización Local

Bloque de funciones ejecutado por nmk hilos por lectura
{

Aquellos hilos cuyo valor cm sea igual a la semilla ejecutan la siguiente función

{

- ***Mp = Max atómico (Mp, p)***

  Por último se obtiene la posición del mínimo. Si en la zona evaluada se hallaron más de un mínimo con el mismo valor, se escoge como semilla aquel que tenga una mayor posición para favorecer la construcción de super k-mers más extensos.

}

}

Sincronización Local

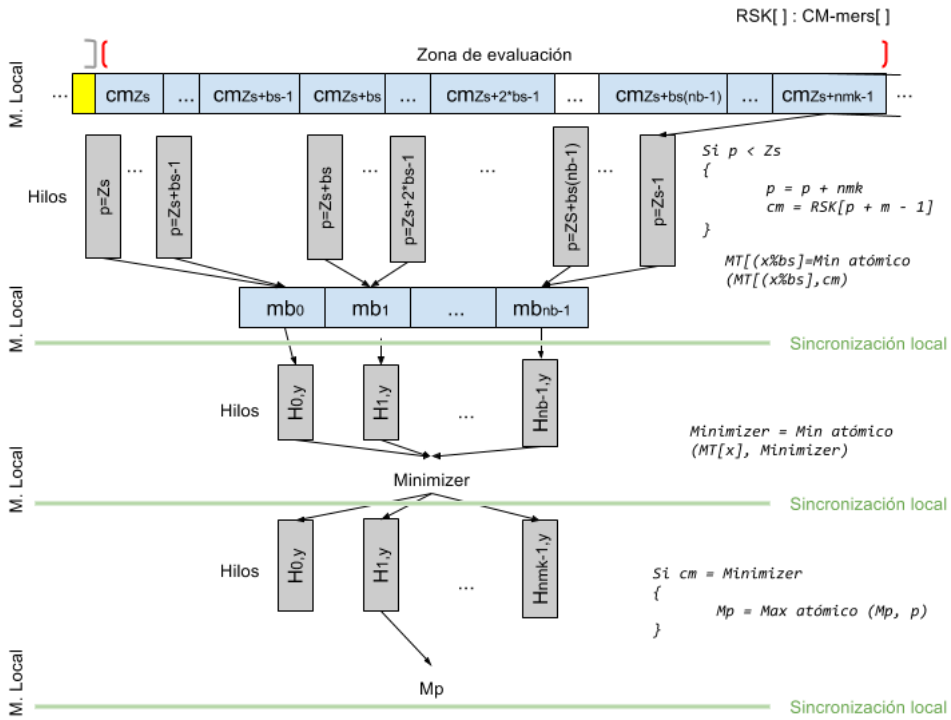


Figura 7.22 Ilustración del algoritmo de evaluación de la zona sin semilla de referencia. Las convenciones, siglas y nomenclatura en general son las mismas usadas en el algoritmo 7.3. Fuente: Autor.

Zona con semilla de referencia - Algoritmo de búsqueda del menor más cercano: Si después de haber procesado una zona, la semilla se encuentra en una posición diferente al inicio de la zona, la ventana se corre de tal forma que la siguiente zona a evaluar inicia en la ubicación de la semilla. En este caso, la nueva zona si cuenta con una semilla de referencia y el procesamiento realizado tiene como propósito encontrar los valores que sean menores a la semilla, y de éstos seleccionar el más cercano (este será la semilla del nuevo super k-mer).

Para conseguir esto, cada hilo se encarga de comparar el valor de su m-mer canónico con la semilla, y solo aquellos que presenten un valor menor prosiguen a hacer una operación atómica con su valor de p (posición de su m-mer canónico dentro del vector Cm-mer [ ]) para seleccionar el de menor posición. Ver algoritmo 7.4 y figura 7.23.

Algoritmo 7.4: Evaluación de zona con semilla de referencia (Algoritmo de búsqueda del menor más cercano)

```
m -> tamaño del m-mer.
cm -> Valor decimal de un m-mer canónico.
nmk -> Número de m-mers por k-mer.
p -> Posición del elemento del vector Cm-mer[] que el hilo está procesando.
Mp -> Posición de la "actual" semilla.
NMp -> Posición de la "nueva" semilla.
RSK[] -> Arreglo en memoria local donde se almacenan los m-mers canónicos.
Minimizer -> Valor decimal de un minimizer.

Bloque de funciones ejecutado por nmk hilos por lectura
{
        Bloque de funciones que se ejecuta si p < sz (Aquellos hilos cuyo
        elemento procesado en la zona anterior está en una posición menor al
        inicio de la actual zona - desplazamiento tipo oruga)
        {
            -   p = p + nmk

                Se actualiza la posición del elemento del vector Cm-mer[] que
cada
                hilo va a
                leer y procesar (desplazamiento tipo oruga)

            -   cm = RSK[p + m - 1]

                Cada hilo lee su nuevo elemento a procesar del vector Cm-mer[]
                (el offset m-1 se debe a que el vector Cm-mer[] se encuentra
                almacenado en el arreglo RSK[] pero corrido m-1 posiciones
                debido a la estrategia de sobre-escritura)
        }

        Aquellos hilos cuyo valor cm sea menor que la semilla ejecutan la
siguiente
        función
        {
            -   NMp = Min atómico (NMp, p)

                Se obtiene la posición del elemento más cerca cuyo valor haya
                sido menor a la semilla.
```

```
        }
}
```

Sincronización Local

Bloque de funciones ejecutado por 1 hilo por lectura sólo si NMp es diferente a Mp (solo si se encontró una nueva semilla)

{

- ***Mp = NMp***
  Minimizer = RSK [Mp + m -1]

  Se actualiza la posición y el valor de la semilla encontrada.

}

Sincronización Local

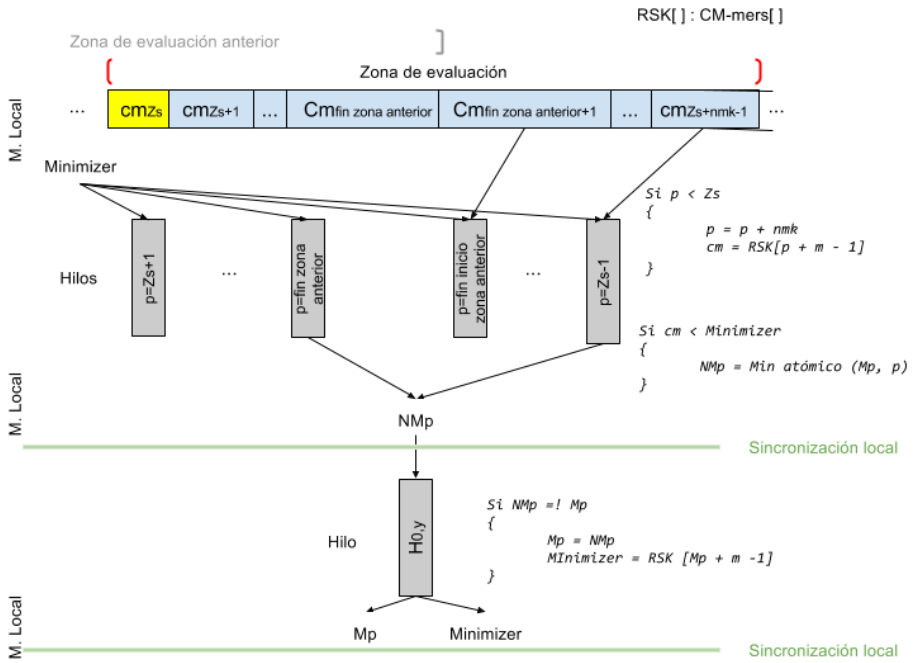


Figura 7.23 Ilustración del algoritmo de evaluación de la zona con semilla de referencia. Las convenciones, siglas y nomenclatura en general son las mismas usadas en el algoritmo 7.4. Fuente: Autor.

8. IMPLEMENTACIÓN Y EVALUACIÓN

8.1. Implementación

La implementación se realizó mediante una programación heterogénea conformada por un código de CPU (host) y dos códigos de dispositivo many core (kernels). Todo el código está disponible en el repositorio: https://github.com/BioinfUD/K-mersCL

Código host

El código host se desarrolló en Python haciendo uso PyOpenCL (Klöckner et. al., 2012), el cual es un paquete de código abierto que permite el fácil acceso al API OpenCL desde Python. El código host se ocupa de cargar las lecturas desde el disco, hacer la conversión a representación numérica y llamar a cualquiera de los kernels a los cuales les pasa las lecturas en su representación numérica. Posterior, toma el resultado de los kernels, que es la estructura de datos CISK y hace la extracción explicita y distribución de los super k-mers. El código host cuenta con unas plantillas que permiten la asignación dinámica de valores dentro de los kernels.

Kernels

Se desarrollaron dos kernels usando OpenCL. Ambos permiten la obtención de los super k-mers y sus semillas de un conjunto de lectura, la diferencia radica en el tipo de semilla empleada, uno de ellos utiliza minimizers canónicos y el otro signatures. Los dos Kernels reciben un conjunto de lecturas representadas en forma numérica (base 4) y retornan los super k-mers y sus semillas en formato CISK para cada una de las lecturas del conjunto.

8.2. Evaluación

Metodología de la evaluación

Herramientas / algoritmos a evaluar: El objetivo de esta sección es evaluar el modelo diseñado mediante la medición del desempeño de los dos kernels implementados en OpenCL: Obtención de super k-mers basados en minimizers

canónicos y obtención de super k-mers basados en signature. Para obtener un factor de aceleración real, estos dos kernel se comparan contra el de desempeño presentado específicamente por los procesos similares en las dos herramientas contadoras de k-mers más usadas que usan dichos estrategias (MSPKmerCounter - Minimizer y KMC2 - Signature) paralelizadas en CPU.

Conjuntos de datos: A partir de lecturas cortas provenientes de la secuenciación del cromosoma 23th del homo sapiens se han generado 11 conjuntos de datos variando el tamaño de la lectura para cubrir un valor típico (100 bases), un valor alto (180 bases) y un valor muy alto con el objeto de poner a prueba la cobertura del modelo (300 bases). También se varió el tamaño del conjunto de acuerdo a la máquina de cómputo sobre la cual se realizó la evaluación: para una máquina de escritorio se usaron conjuntos de 1 y 2 Millones de lecturas, y para una máquina de alto desempeño se usaron conjuntos de 3, 9 y 15 Millones de lecturas.

- Tamaño del archivo: 1M lecturas, Tamaño de la lectura: 180 bases
- Tamaño del archivo: 1M lecturas, Tamaño de la lectura: 300 bases
- Tamaño del archivo: 2M lecturas, Tamaño de la lectura: 100 bases
- Tamaño del archivo: 2M lecturas, Tamaño de la lectura: 180 bases
- Tamaño del archivo: 3M lecturas, Tamaño de la lectura: 100 bases
- Tamaño del archivo: 3M lecturas, Tamaño de la lectura: 180 bases
- Tamaño del archivo: 3M lecturas, Tamaño de la lectura: 300 bases
- Tamaño del archivo: 9M lecturas, Tamaño de la lectura: 180 bases
- Tamaño del archivo: 9M lecturas, Tamaño de la lectura: 300 bases
- Tamaño del archivo: 15M lecturas, Tamaño de la lectura: 100 bases
- Tamaño del archivo: 15M lecturas, Tamaño de la lectura: 180 bases

Equipos de cómputo: La evaluación se llevó acabo en 2 equipos con el fin de medir el desempeño del modelo bajo las limitaciones de un equipo de escritorio y bajo las ventajas de un equipo de alto desempeño:

Computador personal (Thinkpad):
Tarjeta gráfica: GeForce 940M
Disco duro: SSD 512GB
Procesador: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
Memoria: 8GB DDR3
Sistema Operativo: Ubuntu 16.04

Equipo de alto desempeño (CECAD):
Tarjeta gráfica: Nvidia k80
Disco duro: SSD 480GB
Procesador: Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
Memoria: 128GB DDR3
Sistema Operativo: CentOS 7.3

Configuraciones: Tanto los dos kernels como los procesos en las herramientas contadoras de k-mers se evaluaron para longitudes típicas de k-mers y de m-mers. En el caso de minimizers canónicos se usaron longitudes de k-mers iguales a k=31 y k=71, en el caso de signatures se usaron k=51 y K=81. Para ambos casos (minimizers canónicos y signatures) se usaron longitudes de m-mers iguales a m=4, m=5 y m=7.

Para los procesos en las herramientas contadoras de k-mers paralelizadas en CPU se configuró una cantidad de hilos de 4 para el equipo de escritorio y 8 para el equipo de alto desempeño. Para los kernels se configuraron espacios indexados de procesamiento de la siguiente forma:

- Espacio global: Espacio bidimensional con un número de filas igual a las lecturas que conforman el conjunto y un número de columnas igual al menor múltiplo de 32 hilos que cumpla las siguientes condiciones: que sea mayor o igual a 64 y al número de m-mers por k-mer.
- Espacio local: Espacio unidimensional (1 fila) con un número de columnas igual a las asignadas al espacio global.

Métricas: El principal objetivo de esta evaluación es obtener el speed-up alcanzado por cada uno de los kernels sobre los procesos paralelizados en CPU de las herramientas contadoras de k-mers; por este motivo se miden los tiempos de ejecución para cada uno de los kernels y procesos evaluados. Se debe hacer la salvedad de que las herramientas contadoras (de referencia) evaluadas realizan el procesamiento por tandas para disminuir la exigencia de memoria (tanto la lectura de los datos de entrada como la escritura de los de salida se realiza por tandas mediante unos buffer de tamaños adecuados a la memoria), el tiempo de lectura y escritura para cada tanda no se han considerado en la evaluación, se mide a partir de que los datos de entrada ya están en la memoria y hasta que los de salida se ubiquen en la memoria (buffer temporal de salida), de esta forma se asume que las herramientas de referencia se ejecutan en una sola tanda (el mejor de los casos para éstas) al igual que los kernels. Con esto se consigue que la evaluación sea independiente de las estrategias de lectura y escritura y que se enfoque solo en los algoritmos de búsqueda de super k-mers y sus semillas, además que el speed-up se obtenga en el peor de los casos (es decir para el mejor de los casos en las herramientas de referencia).

Como se mencionó en el párrafo anterior, el uso de la memoria no es un problema para las herramientas de referencia debido a que realizan la lectura y escritura en disco por tandas usando buffer temporales de tamaños adecuados a la memoria. Sin embargo para los kernels si puede resultar un gran problema teniendo en cuenta que se realiza todo el procesamiento del conjunto de datos en una sola llamada del kernel. Con el fin de medir que tanto se eleva la exigencia de memoria por parte de los kernels y cuanto se puede mitigar con las estructuras de datos propuestas se realiza la medición del pico de memoria

exigida tanto para los procesos de las herramientas de referencia como para los kernels.

Resultados

Kernel Minimizer en computador personal: A continuación se exponen los resultados para la evaluación del kernel que utiliza minimizers canónicos (Kmercl-Minimizer) con respecto a los procesos similares en la herramienta contadora de k-mers MSPKmercounter usando un computador personal (Tabla 8.1, figuras 8.1, 8.2 y 8.3).

			k=31				**K=71**			
Tool	**NR**	**RS**	**m=4**		**m=7**		**m=4**		**m=7**	
			M(MB)	T(s)	M(MB)	T(s)	M(MB)	T(s)	M(MB)	T(s)
Kmercl-Minimizer	2M	100	630	0.887	630	0.979	630	0.911	630	0.894
MSPKmercounter	2M	100	430	3.38	410	3.55	407	2.53	400	2.59
Speed-up (x)				**3.811**		**3.626**		**2.777**		**2.897**
Kmercl-Minimizer	2M	180	1120	1.591	1120	1.763	1120	1.625	1120	1.77
MSPKmercounter	2M	180	470	6.31	507	6.6	550	5.2	450	5.3
Speed-up (x)				**3.966**		**3.744**		**3.200**		**2.994**

NR = Número de lecturas, RS = Tamaño de la lectura.

Tabla 8.1 Resultados para la evaluación del kernel que utiliza minimizers canónicos con respecto al proceso similar en la herramienta MSPKmercounter.usando un computador personal

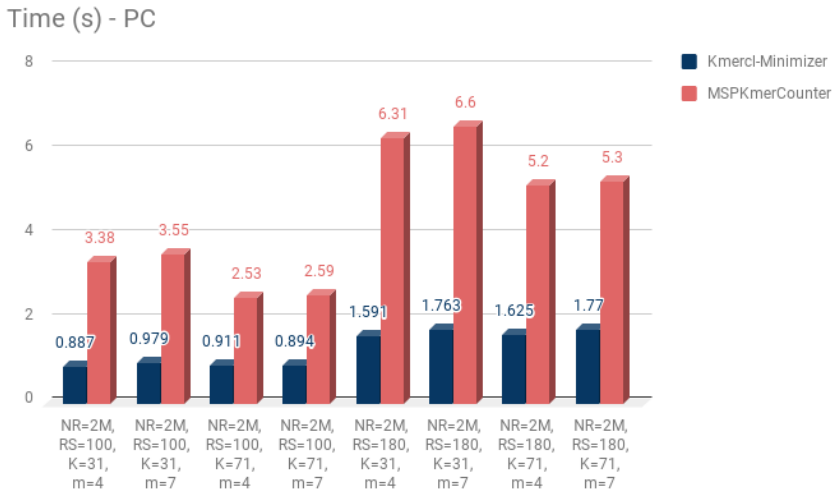


Figura 8.1 Tiempos de ejecución del kernel que utiliza minimizers canónicos y del proceso similar en la herramienta MSPKmercounter usando un computador personal. Fuente: Autor.

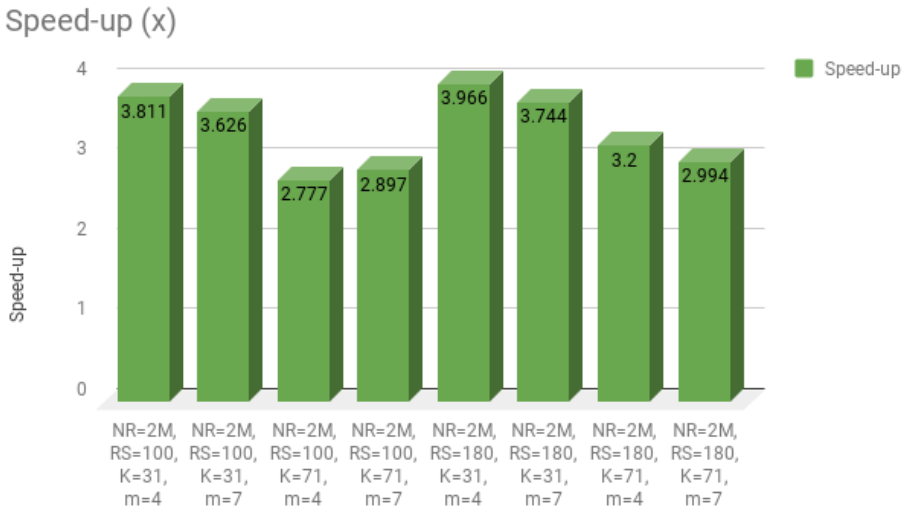


Figura 8.2 Speed-up del kernel que utiliza minimizers canónicos con respecto al proceso similar en la herramienta MSPKmercounter usando un computador personal. Fuente: Autor.

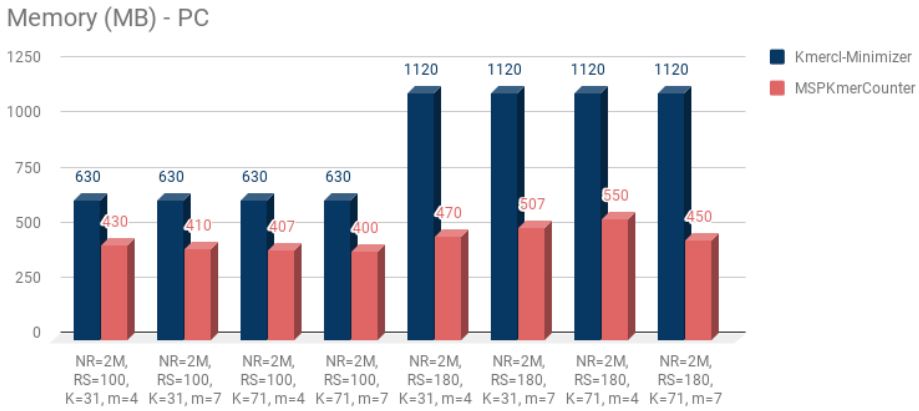


Figura 8.3 Máxima memoria exigida por el kernel que utiliza minimizers canónico y por el proceso similar en la herramienta MSPKmercounter usando un computador personal. Fuente: Autor.

Kernel Minimizer en computador de alto desempeño: A continuación se exponen los resultados para la evaluación del kernel que utiliza minimizers canónicos (Kmercl-Minimizer) con respecto a los procesos similares en la herramienta contadora de k-mers MSPKmercounter usando un computador de alto desempeño (Tabla 8.2, figuras 8.4, 8.5 y 8.6).

			k=31				K=71			
Tool	**NR**	**RS**	**M=4**		**M=7**		**M=4**		**M=7**	
			M(MB)	T(s)	M(MB)	T(s)	M(MB)	T(s)	M(MB)	T(s)
Kmercl-Minimizer	3M	100	1110	0.622	1110	0.772	1110	0.422	1110	0.456
MSPKmercounter	3M	100	844	1.868	1084	2.131	867	1.49	846	1.59
Speed-up (x)				**3.003**		**2.760**		**3.531**		**3.487**
Kmercl-Minimizer	15M	100	5843	2.564	4650	3.146	4650	1.742	4650	1.9
MSPKmercounter	15M	100	2600	7.85	2502	8.851	2580	6.441	2494	6.115
Speed-up (x)				**3.062**		**2.813**		**3.697**		**3.218**
Kmercl-Minimizer	3M	180	2135	0.998	1815	1.144	1815	0.702	1815	0.76
MSPKmercounter	3M	180	1218	3.6	1187	3.375	1217	2.9	1110	2.751
Speed-up (x)				**3.607**		**2.950**		**4.131**		**3.620**

Kmercl-Minimizer	15M	180	10421	4.621	8600	5.418	8600	3.109	8600	3.428
MSPKmercounter	15M	180	3720	14.08	3710	15.85	3699	13.4	3787	12.2
Speed-up (x)				**3.047**		**2.925**		**4.310**		**3.559**

NR = Número de lecturas, RS = Tamaño de la lectura.

Tabla 8.2 Resultados para la evaluación del kernel que utiliza minimizers canónicos con respecto al proceso similar en la herramienta MSPKmercounter usando un computador de alto desempeño

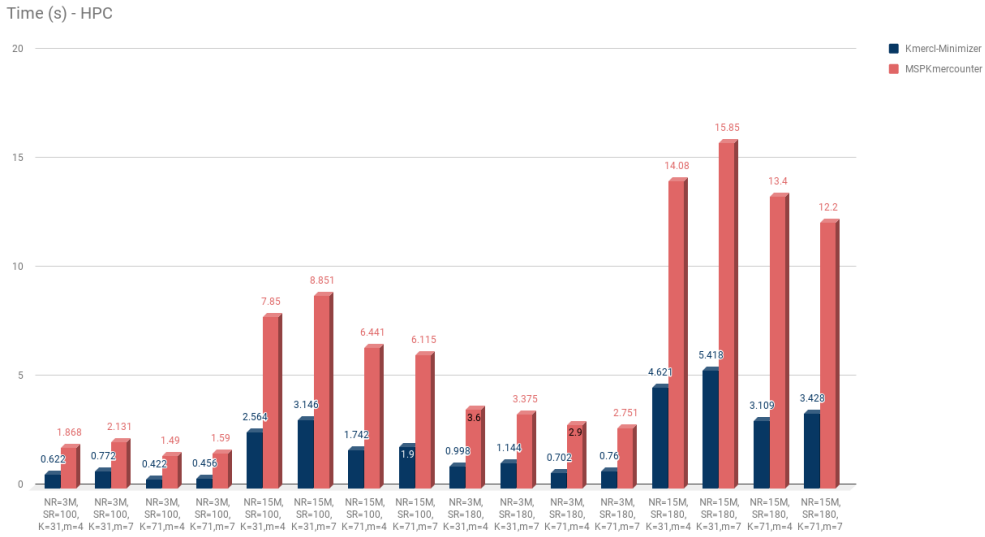


Figura 8.4 Tiempos de ejecución del kernel que utiliza minimizers canónicos y del proceso similar en la herramienta MSPKmercounter usando un computador de alto desempeño. Fuente: Autor.

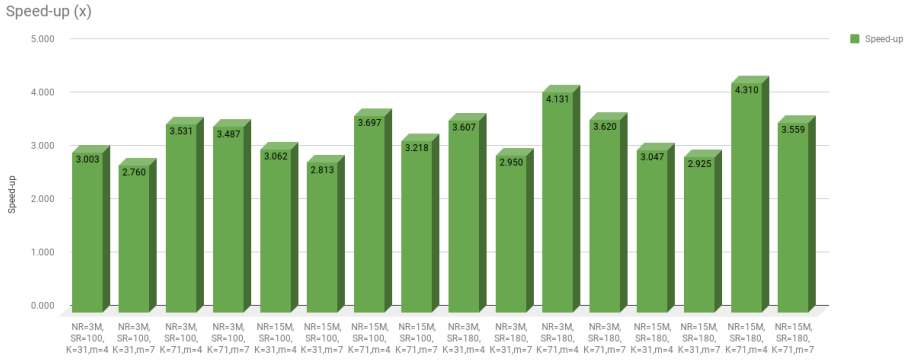


Figura 8.5 Speed-up del kernel que utiliza minimizers canónicos con respecto al proceso similar en la herramienta MSPKmercounter usando un computador de alto desempeño. Fuente: Autor.

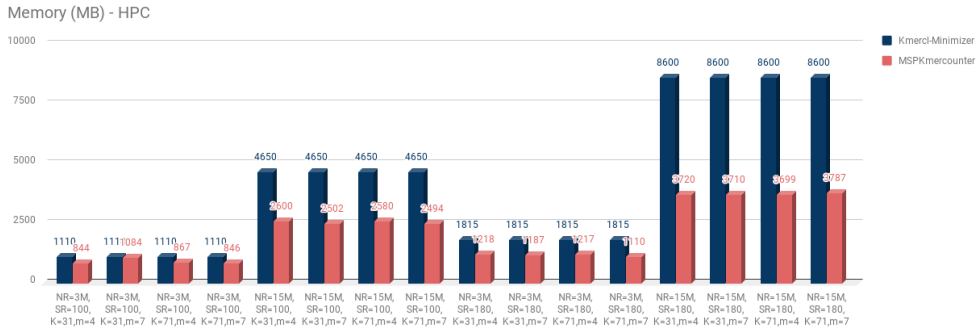


Figura 8.6 Máxima memoria exigida por el kernel que utiliza minimizers canónicos y por el proceso similar en la herramienta MSPKmercounter usando un computador de alto desempeño. Fuente: Autor.

Kernel Signature en computador personal: A continuación se exponen los resultados para la evaluación del kernel que utiliza signatures (Kmercl-Signature) con respecto a los procesos similares en la herramienta contadora de k-mers KMC 2 usando un computador personal (Tabla 8.3, figuras 8.7, 8.8 y 8.9).

			k=51				k=81			
Tool	**NR**	**RS**	**m=5**		**m=7**		**m=5**		**m=7**	
			M(MB)	T(s)	M(MB)	T(s)	M(MB)	T(s)	M(MB)	T(s)
Kmercl-Signature	1M	180	711	1.092	711	1.285	707	0.787	711	0.855
KMC 2	1M	180	872	2.009	632	2.063	1068	1.697	624	1.782
Speed-up (x)				**1.840**		**1.605**		**2.157**		**2.084**
Kmercl-Signature	1M	300	1169	1.777	1169	2.096	1169	1.261	1169	1.385
KMC 2	1M	300	1117	3.101	1121	3.183	1466	2.788	1043	2.814
Speed-up (x)				**1.745**		**1.518**		**2.210**		**2.031**

NR = Número de lecturas, RS = Tamaño de la lectura.

Tabla 8.3 Resultados para la evaluación del kernel que utiliza signatures con respecto al proceso similar en la herramienta KMC 2 usando un computador personal

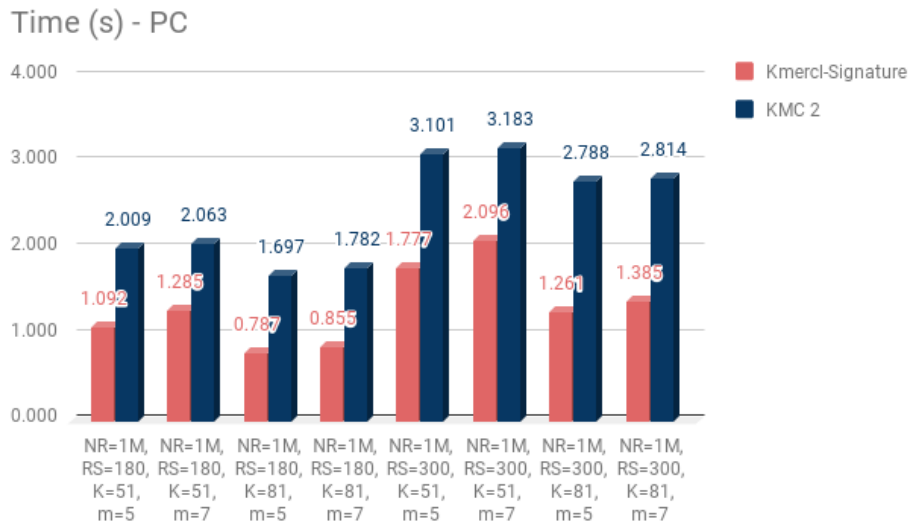


Figura 8.7 Tiempos de ejecución del kernel que utiliza signatures y del proceso similar en la herramienta KMC 2 usando un computador personal. Fuente: Autor.

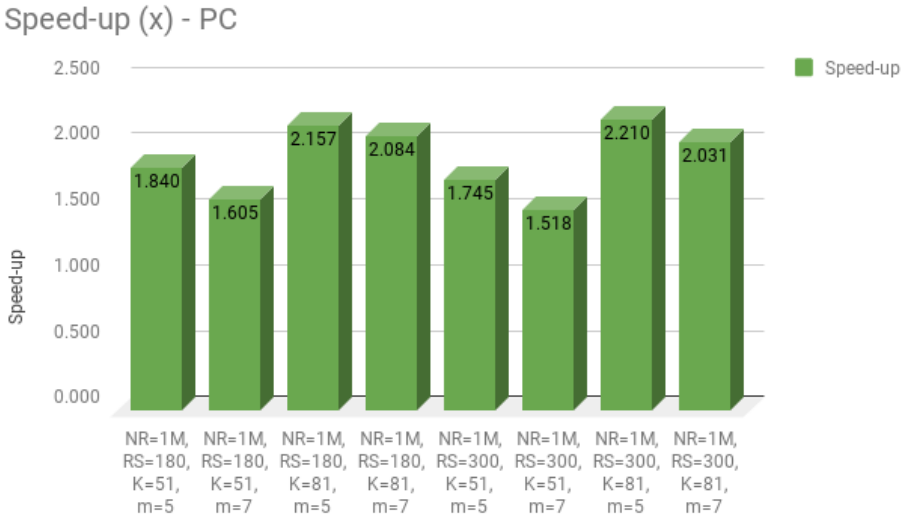


Figura 8.8 Speed-up del kernel que utiliza signatures con respecto al proceso similar en la herramienta KMC 2 usando un computador personal. Fuente: Autor.

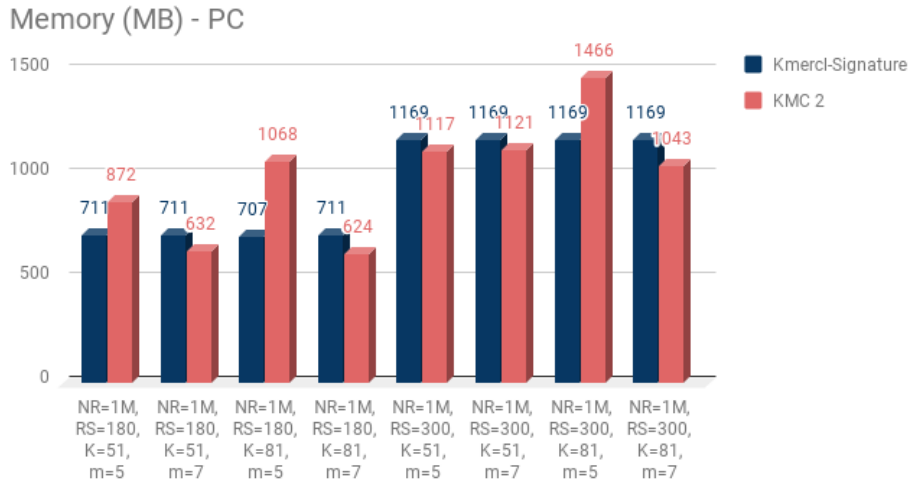


Figura 8.9 Máxima memoria exigida por el kernel que utiliza signatures y por el proceso similar en la herramienta KMC 2 usando un computador personal. Fuente: Autor.

Kernel Signature en computador de alto desempeño: A continuación se exponen los resultados para la evaluación del kernel que utiliza signatures (Kmercl-Signature) con respecto a los procesos similares en la herramienta contadora de k-mers KMC 2 usando un computador de alto desempeño (Tabla 8.4, figuras 8.10, 8.11 y 8.12).

			k=51				k=81			
Tool	**NR**	**RS**	**m=5**		**m=7**		**m=5**		**m=7**	
			M(MB)	T(s)	M(MB)	T(s)	M(MB)	T(s)	M(MB)	T(s)
Kmercl-Signature	3M	180	2142	0.760	2142	0.907	2130	0.640	2142	0.714
KMC 2	3M	180	3642	4.415	2321	4.453	4488	4.278	3064	4.347
Seed-up (x)				**5.806**		**4.910**		**6.685**		**6.085**
Kmercl-Signature	9M	180	6285	1.976	6285	2.387	6285	1.684	6285	1.929
KMC 2	9M	180	5616	9.887	4657	10.543	3767	9.623	4445	10.671
Seed-up (x)				**5.003**		**4.417**		**5.715**		**5.532**
Kmercl-Signature	3M	300	3515	1.116	3515	1.298	3515	1.286	3515	1.069
KMC 2	3M	300	4584	5.733	3543	6.043	5387	5.531	2794	5.810
Seed-up (x)				**5.138**		**4.657**		**4.302**		**5.434**
Kmercl-Signature	9M	300	10406	3.169	10406	3.787	10406	2.768	10406	3.083

KMC 2	9M	300	8999	15.429	6293	15.984	8471	15.229	6477	16.250
Seed-up (x)				**4.869**		**4.221**		**5.501**		**5.270**

NR = Número de lecturas, RS = Tamaño de la lectura.

Tabla 8.4 Resultados para la evaluación del kernel que utiliza signatures con respecto al proceso similar en la herramienta KMC 2 usando un computador de alto desempeño

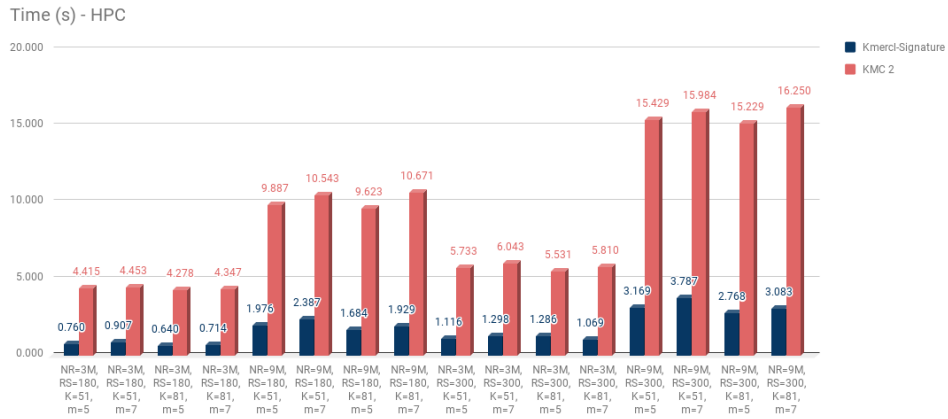


Figura 8.10 Tiempos de ejecución del kernel que utiliza signatures y del proceso similar en la herramienta KMC 2 usando un computador de alto desempeño. Fuente: Autor.

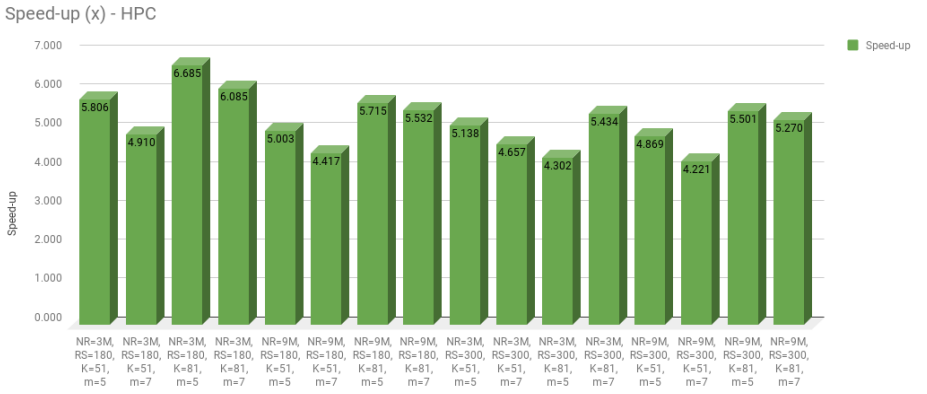


Figura 8.11 Speed-up del kernel que utiliza signatures con respecto al proceso similar en la herramienta KMC 2 usando un computador de alto desempeño. Fuente: Autor.

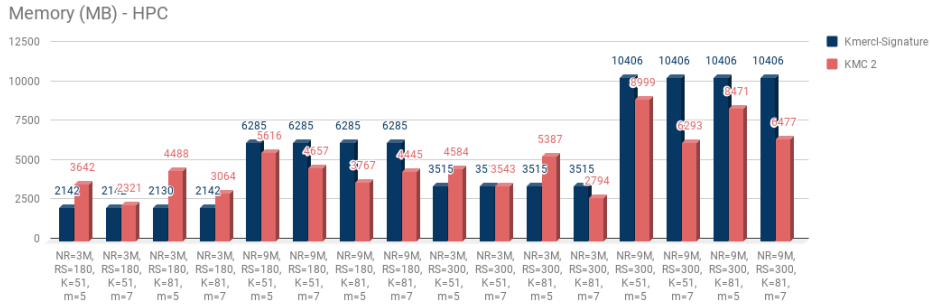


Figura 8.12 Máxima memoria exigida por el kernel que utiliza signatures y por el proceso similar en la herramienta KMC 2 usando un computador de alto desempeño. Fuente: Autor.

Análisis de resultados

Speed-up:

- En promedio el kernel basado en minimizers canónico bajo ambas condiciones computacionales presentó un factor de aceleración de 3.364x
- En promedio el kernel basado en signatures bajo ambas condiciones computacionales presentó un factor de aceleración de 4.114x.
- El factor de aceleración para el mismo kernel bajo las mismas condiciones computacionales se mantiene en un rango determinado sin picos (mínimos o máximos) considerables a pesar de las variaciones en el tamaño del conjunto de datos, en la longitud del k-mer y del m-mer.
- El cambio de condiciones computacionales del computador personal al computador de alto desempeño generó una mejora en el factor de aceleración, no tan considerable para el kernel basado en minimizer canónicos y muy considerable para el kernel basado en signatures.
- Los factores de aceleración más bajos se experimentaron para el kernel basado en signatures bajo las condiciones del computador personal (un rango cercano a 2x) y los más altos se midieron para el mismo kernel pero ejecutado en el computador de alto desempeño (un rango cercano a 5x)

Memoria:

- A pesar de que los kernels procesan todo el conjunto de datos en la GPU en un solo llamado, éstos presentan requerimiento de memoria en rangos similares a los exigidos por los procesos de las herramientas de referencia las cuales procesan por tandas.
- La exigencia de memoria por parte de los kernels va sujeta al tamaño del conjunto de datos de entrada y no a los parámetros que pueden hacer variar los super k-mers de salida. Esto significa que el conjunto de datos de salida no impacta sobre los requerimientos de memoria gracias a la estructura de datos CISK.

9. CONCLUSIONES, PRINCIPALES APORTES, DIVULGACIÓN, RECOMENDACIONES Y FUTUROS TRABAJOS

9.1. Conclusiones

A lo largo de cada una de las fases de esta investigación e inclusive en los estudios preliminares se ha llegado a un listado de conclusiones que se presentan a continuación:

- Un flujo de trabajo típico de un conjunto de lecturas cortas genómicas secuenciadas mediante tecnología NGS (Next Generation Sequencing) se conforma por 4 etapas: Pre-procesamiento, ensamblaje, anotación y mapeo. De estas etapas la que presenta mayor exigencia computacional es el ensamblaje, especialmente cuando no se cuenta con un genoma de

referencia (de-novo). A pesar de que en la última década se ha simplificado y facilitado dicho proceso mediante métodos basados en grafos de De Bruijn, aún sigue siendo la etapa de mayor complejidad y costo computacional debido a la dificultad de procesamiento de las sub-secuencias que representan los nodos y vértices del grafo (k-mers).

- Las dificultades que surgen al procesar computacionalmente k-mers se fundamentan principalmente en dos razones: - El enorme volumen de datos generado al extraer los k-mers, debido a la redundancia (dos k-mers contiguos presentan una redundancia de k-1 bases) y – la alta dificultad del procesamiento particionado: la mayoría de tareas con k-mers exigen la carga en memoria de todo el conjunto de datos debido a la imposibilidad del procesamiento seccionado o la necesidad de tareas de unión pos-procesado con muy alta complejidad.

- Los métodos de procesamiento de k-mers por particionamiento en disco basados en semillas de tipo minimizer por peso lexicográfico representan una muy buena solución a la alta exigencia de memoria, tanto por la distribución del conjunto de datos en archivos pequeños que se pueden procesar por separado, como por la reducción drástica de redundancia gracias al uso de estructuras llamadas super k-mers. Sin embargo este gran beneficio en el uso eficiente de la memoria se ve un poco opacado por el incremento de la necesidad de cómputo previo referente a la búsqueda de semillas en cada uno de los k-mers de cada una de las lecturas.

- Las intensivas tareas de búsquedas de minimizers y construcción de super k-mers en los métodos de particionamiento de disco basados en semilla se pueden resolver de forma eficiente mediante un modelo de procesamiento heterogéneo y una estructura de datos basada en índices, de tal forma que la búsqueda de semillas y la detección de super k-mers se procesan en la plataforma many core sin necesidad de hacer explícita la extracción de los k-mers ni de los super k-mers, y las tareas de extracción explícita y distribución se realicen en la plataforma multi core donde no hay tanta capacidad de cómputo pero si de memoria.

- La paralelización masiva del proceso de obtención de m-mers canónicos y la detección de super k-mers por minimizer de un conjunto de lecturas se puede resolver de forma eficiente mediante patrones enfocados a reducir el uso de memoria y a maximizar la intensidad operacional disminuyendo el acceso a los tipos de memoria con latencia alta tales como la global y la local (OpenCL) / compartida (CUDA) y favoreciendo el uso de memorias rápidas como la privada. El patrón de paralelización para la obtención de m-mers canónicos se consigue mediante un algoritmo de granularidad híbrida por tiles y el de la detección de super k-mers por minimizers se obtiene mediante una ventana de corrimiento por saltos con un patrón de reducción mixto y

adaptativo. En esta investigación se desarrollaron y evaluaron dos kernels basados en dichos patrones, los cuales alcanzaron factores de aceleración de hasta 6x con respecto a procesos similares en herramientas contadores de k-mers muy reconocidas y recientes.

- Un modelo de programación heterogénea se puede representar mediante una metodología simplificada usando solo 3 componentes: - Las estructuras de datos, - la distribución de tareas entre plataformas multi core y many core y - el modelo de procesamiento masivo sobre la(s) plataforma(s) many core. Este último modelo a su vez se representa mediante el uso de la estructura jerárquica de la memoria, los espacios indexados de procesamiento y mediante los algoritmos de paralelización masiva definidos a través de funciones indexadas con respecto a los espacios de procesamiento.

9.2. Principales aportes y divulgación

FASE	APORTE	DIVULGACIÓN
Propuesta	**Marco de referencia** de requerimientos computacionales, potenciales ventajas y desventajas, cuellos de botella y retos para: - Principales etapas de análisis genómico y transcriptómico de secuenciación de nueva generación	**Libro:** Libro Entorno de trabajo bioinformático para RNA-Seq. Editorial UD. (2015) **Artículos publicados:** **A2**. Automation of functional annotation of genomes and transcriptomes. Revista Tecnura, 18, p 90-96 (2014) **A2**. The immunotranscriptome of the Caribbean reef-building coral Pseudodiploria strigosa. Immunogenetics. 67(9), 515-530 (2015) **A2**. Performance assessment for main stages in genomic and transcriptomic data processing based upon reads from illumina sequencing technologies. International

		Journal of Applied Engineering Research, 10(14), p 34670-34674 (2015) **Ponencias realizadas:** Massive Automatic Functional Annotation MAFA. International Work-Conference on Bioinformatics and Biomedical Engineering – IWBBIO. Granada España 2014 Galaxy RNA-Seq UD: Una plataforma bioinformática para el análisis RNA-Seq. Conferencia Ibérica de Sistemas e Tecnologías de Información – CISTI. Aveiro Portugal 2015.
Diseño	**Marco de referencia** de requerimientos computacionales, potenciales ventajas y desventajas, cuellos de botella y retos para: - Ensambladores genómicos de-novo basados en grafos de De Bruijn - Técnicas de particionamiento en disco usadas en el ensamblaje genómico de-novo - Técnicas de procesamiento de k-mers - Minimizers usados en el conteo de K-mers **Metodología simplificada** para modelar sistemas de computación heterogénea incluyendo la programación paralela masiva sobre plataformas many core. **Modelo de computación heterogénea** para el particionamiento en disco de un conjunto de lecturas genómicas usando super k-mers por semillas tipo minimizer. **CISK, una estructura de datos** para representar super k-mers y sus semillas de forma indexada y compacta. **Patrón de paralelización masiva** para: - La obtención de los m-mers canónicos de un conjunto de lecturas sobre plataformas many	**Artículos publicados:** **A1**. Computational Performance Assessment of k-mers counting algorithms. Journal of Computational Biology. 23(4), 248-255 (2016). **A2**. Performance Assessment by Stages of Main Genomic De-Novo Assemblers Based Upon De Bruijn Graphs. Life Science Journal, 12(12), 13-21 (2015) **C**. Computational Performance Assessment Of Minimizers Uses In Counting K-Mers. Australian journal of basic and applied sciences. 10(8), 71-78 (2016). **Ponencias realizadas:** Disk Partition Techniques assessment and analysis applied to genomic assemblers based on bruijn graphs. 2nd International Conference on Bioinformatics and Computer Engineering – ICBCE. Los Ángeles, USA 2016.

	core. - La búsqueda de super k-mers por semilla tipo minimizer sobre plataformas many core. La generalización de este proceso generó un nuevo patrón: - La búsqueda de min/max en ventanas corredizas de series de datos.	
Implementación	**Código host y kernels en OpenCL** para el particionamiento en disco de un conjunto de lecturas genómicas usando super k-mers por semillas tipo minimizer sobre computación heterogénea CPU/GPU	**Libro:** Manual Práctico OpenCL (Libro aprobado para la colección especial del doctorado de la Editorial UD) **Repositorio:** https://github.com/BioinfUD/K-mersCL
Evaluación	**Marco de referencia** sobre el desempeño del modelo de programación heterogénea para el particionamiento en disco de un conjunto de lecturas genómicas usando super k-mers por semillas tipo minimizer y los dos nuevos patrones de paralelización propuestos, para obtener los m-mers canónico y para buscar los super k-mers.	**Artículos publicables:** **A1**. Heterogeneous computing model to the partition on disk of short reads data sets using super k-mers based on minimizers or signatures. (En evaluación) **A1**. Parallel Programing Pattern to obtain the canonical m-mers of a reads set on many core platforms. (En evaluación) **A1**. Parallel Programing Pattern to search super k-mers based on minimizers in a reads set on many core platforms. (En evaluación) **Ponencias realizadas:** Basic K-Mer Operations Using Massive Parallel Processing On Heterogeneus Architectures. 7th IEEE International Conference on Software Engineering and Service Science – ICSESS. Beijing, China 2016 Heterogeneous parallel computing to accelerate k-mers processing in genome assembly processes. International Conference on Cloud and Big Data Computing – ICCBDC. Londres, Inglaterra 2017.
Proyección	**Banco de Proyectos** de los cuales 3 ya han sido aprobados para trabajos de grado y 1 se encuentra en evaluación en una convocatoria del CIDC:	

	Proyecto de Maestría: - Modelo de procesamiento paralelo en arquitecturas heterogéneas para regresiones lineales multivariables. (Aprobado) **Proyectos de Pregrado:** - Sistema modular de adquisición, gestión y visualización web de métricas de desempeño para centros de cómputo de alto rendimiento. Caso de estudio: CECAD. (Aprobado) - Diseño e implementación de un entorno de trabajo Big Data para el CECAD. (Aprobado) **Proyectos Institucionales:** - Diseño, implementación y evaluación de modelos de procesamiento para patrones fundamentales de paralelismo enfocados a ciencia de datos de volúmenes grandes. (En evaluación)	

Tabla 9.1 Principales aportes y divulgación

9.3. Recomendaciones y futuros trabajos

A través del desarrollo de cada una de las fases de esta investigación se vislumbraron potenciales proyectos de investigación:

- Diseño de patrones fundamentales de paralelismo enfocados a ciencia de datos de volúmenes grandes. (Actualmente se encuentra en evaluación en una convocatoria del CIDC)

- Uso de computación heterogenea para el modelado de procesos estocásticos. (Actualmente es una propuesta aprobada de tesis de maestría)

- Diseño de funciones HASH pensadas naturalmente para ser ejecutadas en paralelo sobre plataforma many core.

- Diseño de Filtros Bloom basados en funciones HASH en paralelo.

- Diseño de metodologías de tratamientos de k-mers en paralelo basadas en la unión de técnicas de particionamiento en disco por semillas tipo minimizers y filtros Bloom.

- Desarrollo de un ensamblador genómico de-novo completo usando computación heterogénea.

- Diseño de patrones de paralelización enfocados a procesos de Deep Learning sobre plataformas many core.

- Diseño de patrones de paralelización en cualquier área para ser ejecutados en entornos heterogéneos con dispositivos many core reconfigurable (FPGAs).

- Diseño e implementación de entornos de desarrollo y monitoreo para centros de cómputo de alto desempeño enfocados a las ciencias de datos y a Big Data. (Actualmente hay 2 propuestas de proyectos de pregrados aprobados)

REFERENCIA BIBLIOGRÁFICAS

Abouelhoda, M. I., Kurtz, S., & Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms, 2(1), 53-86. doi:10.1016/s1570-8667(03)00065-0

Alba, E. (2005). Parallel metaheuristics: a new class of algorithms (Vol. 47). John Wiley & Sons.

Altschul, S. (1997). Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research, 25(17), 3389-3402. doi:10.1093/nar/25.17.3389

Amar Shan. (2006). Heterogeneous processing: a strategy for augmenting moore's law. Linux J. 2006, 142, 1-7

Arora, N. S., Blumofe, R. D., & Plaxton, C. G. (2001). Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems, 34(2), 115-144.

Audano, P., & Vannberg, F. (2014). KAnalyze: a fast versatile pipelined K-mer toolkit. Bioinformatics; 30(14), 2070-2072.

Bao, S., Jiang, R., Kwan, W., Wang, B., Ma, X., & Song, Y. Q. (2011). Evaluation of next-generation sequencing software in mapping and assembly. Journal of human genetics, 56(6), 406-414.

Bastide, M., & McCombie, W. R. (2007). Assembling genomic DNA sequences with PHRAP. Current Protocols in Bioinformatics, 11-4.

Batzoglou, S., Jaffe, D. B., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., ... & Lander, E. S. (2002). ARACHNE: a whole-genome shotgun assembler.Genome research, 12(1), 177-189.

Birol, I., Jackman, S. D., Nielsen, C. B., Qian, J. Q., Varhol, R., Stazyk, G., ... & Jones, S. J. (2009). De novo transcriptome assembly with ABySS.Bioinformatics, 25(21), 2872-2877.

Blelloch, G. E. (1990). Vector models for data-parallel computing (Vol. 356). Cambridge: MIT press.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM; 13(7), 422-426.

Bowe, A., Onodera, T., Sadakane, K., & Shibuya, T. (2012). Succinct de Bruijn Graphs. Lecture Notes in Computer Science Algorithms in Bioinformatics, 225-235. doi:10.1007/978-3-642-33122-0_18

Bryant, D. W., Wong, W. K., & Mockler, T. C. (2009). QSRA–a quality-value guided de novo short read assembler. BMC bioinformatics, 10(1), 69.

Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I. A., Belmonte, M. K., Lander, E. S., ... & Jaffe, D. B. (2008). ALLPATHS: de novo assembly of whole-genome shotgun microreads. Genome research, 18(5), 810-820.

Buttari, A., Langou, J., Kurzak, J., & Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing, 35(1), 38-53.

Castillo, J. N. P., Parra, N. E. V., & Ramirez, L. M. G. (2014). Optimización del pre-procesamiento de lecturas de secuenciación de nueva generación. REDES DE INGENIERÍA, 5(2).

Chaisson, Mark J, and Pavel A Pevzner. "Short read fragment assembly of bacterial genomes." Genome research 18.2 (2008): 324-330.

Chaisson, M. J., Brinza, D., & Pevzner, P. A. (2009). De novo fragment assembly with short mate-paired reads: Does the read length matter?. Genome research, 19(2), 336-346.

Chang, L., & Gómez-Luna, J. (2017). Parallel patterns: prefix sum. Programming Massively Parallel Processors, 175-197. doi:10.1016/b978-0-12-811986-0.00008-x

Chandra, R. (Ed.). (2001). Parallel programming in OpenMP. Morgan Kaufmann.

Cheung, V. G., & Spielman, R. S. (2009). Genetics of human gene expression: mapping DNA variants that influence gene expression. Nature Reviews Genetics, 10(9), 595-604.

Chikhi, R., Limasset, A., Jackman, S., Simpson, J. T., & Medvedev, P. (2014). On the Representation of de Bruijn Graphs. Lecture Notes in Computer Science Research in Computational Molecular Biology, 35-55. doi:10.1007/978-3-319-05269-4_4

Chikhi, R., Limasset, A., & Medvedev, P. (2016). Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinformatics, 32(12), I201-I208. doi:10.1093/bioinformatics/btw279
Chikhi, R., & Medvedev, P. (2013). Informed and automated k-mer size selection for genome assembly. Bioinformatics, 30(1), 31-37. doi:10.1093/bioinformatics/btt310

Chikhi, R., & Rizk, G. (2012). Space-Efficient and Exact de Bruijn Graph Representation Based on a Bloom Filter. Lecture Notes in Computer Science Algorithms in Bioinformatics, 236-248. doi:10.1007/978-3-642-33122-0_19

Cock, P. J., Fields, C. J., Goto, N., Heuer, M. L., & Rice, P. M. (2010). The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. Nucleic acids research, 38(6), 1767-1771.

Compeau, P. E., Pevzner, P. A., & Tesler, G. (2011). How to apply de Bruijn graphs to genome assembly. Nature biotechnology, 29(11), 987-991.

Compeau, P. E., & Pevzner, P. A. (2010). Genome Reconstruction: A Puzzle with a Billion Pieces.

Cormode, G., & Muthukrishnan, S. (2005). An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms; 55(1), 58-75.

CUDA C Best Practices Guide. Retrieved September 22, 2017, from http://docs.nvidia.com/cuda/cuda-c-best-practices-guide

CUDA C Programming Guide. Retrieved September 22, 2017, from http://docs.nvidia.com/cuda/cuda-c-programming-guide

De Antonio, M., & Marina, L. (2005). Computación paralela y entornos heterogéneos.

Deorowicz, S., Kokot, M., Grabowski, S., & Debudaj-Grabysz, A. (2015). KMC 2: fast and resource-frugal k-mer counting. Bioinformatics, 31(10), 1569-1576. doi:10.1093/bioinformatics/btv022

DiGuistini, S., Liao, N. Y., Platt, D., Robertson, G., Seidel, M., Chan, S. K., ... & Jones, S. J. (2009). De novo genome sequence assembly of a filamentous fungus using Sanger, 454 and Illumina sequence data. Genome Biol, 10(9), R94.

Dohm, J. C., Lottaz, C., Borodina, T., & Himmelbauer, H. (2007). SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. Genome research, 17(11), 1697-1706.

El-Metwally, S., Hamza, T., Zakaria, M., & Helmy, M. (2013). Next-generation sequence assembly: four stages of data processing and computational challenges. PLoS computational biology, 9(12), e1003345.

El-Metwally, S. (2014). Next Generation Sequencing Technologies and Challenges in Sequence Assembly (Vol. 7). Springer Science & Business.

Fialka, O., & Cadik, M. (2006). FFT and Convolution Performance in Image Filtering on GPU. Tenth International Conference on Information Visualisation (IV06). doi:10.1109/iv.2006.53

Garg, A., Jain, A., & Paul, K. (2013). GGAKE: GPU Based Genome Assembly Using K-Mer Extension. 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing. doi:10.1109/hpcc.and.euc.2013.156

Erbert, M., Rechner, S., & Müller-Hannemann, M. (2017). Gerbil: A Fast and Memory-Efficient k-mer Counter with GPU-Support. Algorithms for Molecular Biology, 12-9. doi:10.1186/s13015-017-0097-9

Georganas, E., Buluc, A., Chapman, J., Oliker, L., Rokhsar, D., & Yelick, K. (2014). Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly. SC14: International Conference for High Performance Computing, Networking, Storage and Analysis. doi:10.1109/sc.2014.41

Grabherr, M. G., Haas, B. J., Yassour, M., Levin, J. Z., Thompson, D. A., Amit, I., ... & Regev, A. (2011). Full-length transcriptome assembly from RNA-Seq data without a reference genome. Nature biotechnology, 29(7), 644-652.

Gonnella, G., & Kurtz, S. (2012). Readjoiner: a fast and memory efficient string graph-based sequence assembler. BMC bioinformatics, 13(1), 82.

Gupta, S., Chaudhury, S., & Panda, B. (2014, February). MUSIC: A hybrid computing environment for burrows-wheeler alignment for massive amount of short read sequence data. In Biomedical Engineering (MECBME), 2014 Middle East Conference on (pp. 188-191). IEEE.

Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S. H., Grajewski, M., & Turek, S. (2007). Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. Parallel Computing, 33(10), 685-699.

Han, T. D., & Abdelrahman, T. S. (2011, March). Reducing branch divergence in GPU programs. In Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (p. 3). ACM.

Harish, P., & Narayanan, P. J. (2007). Accelerating large graph algorithms on the GPU using CUDA. In High performance computing–HiPC 2007 (pp. 197-208). Springer Berlin Heidelberg.

Hart, A. (2012). The OpenACC programming model. Cray Exascale Research Initiative Europe, Tech. Rep.

Henschel, R., Lieber, M., Wu, L., Nista, P. M., Haas, B. J., & LeDuc, R. D. (2012). Trinity RNA-Seq assembler performance optimization. Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond. ACM.

Heo, Y., Wu, X. L., Chen, D., Ma, J., & Hwu, W. M. (2014). BLESS: Bloom filter-based error correction solution for high-throughput sequencing reads. Bioinformatics, btu030.

Hernandez, D., François, P., Farinelli, L., Østerås, M., & Schrenzel, J. (2008). De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. Genome research, 18(5), 802-809.

Hill, M. D. (2008). Amdahl's Law in the multicore era. 2008 IEEE 14th International Symposium on High Performance Computer Architecture. doi:10.1109/hpca.2008.4658638

Hossain, M. S., Azimi, N., & Skiena, S. (2009). Crystallizing short-read assemblies around seeds. BMC bioinformatics, 10(Suppl 1), S16.

Huang, X., & Madan, A. (1999). CAP3: A DNA sequence assembly program.Genome research, 9(9), 868-877.

Huson, D. H., Reinert, K., & Myers, E. W. (2002). The greedy path-merging algorithm for contig scaffolding. Journal of the ACM (JACM), 49(5), 603-615.

I. Maccallum, D. Przybylski, S. Gnerre, J. Burton, I. Shlyakhter, A. Gnirke, J. Malek, K. McKernan, S. Ranade, T. P. Shea, et al., Allpaths 2: Small genomes assembled accurately and with high continuity from short paired reads, Genome Biol., vol. 10, no. 10, p. R103, 2009.

Ilie, L., Haider, B., Molnar, M., & Solis-Oba, R. (2014). SAGE: String-overlap Assembly of GEnomes. BMC bioinformatics, 15(1), 302.

Idury, R. M., & Waterman, M. S. (1995). A new algorithm for DNA sequence assembly. Journal of computational biology, 2(2), 291-306.

Jain, A., Garg, A., & Paul, K. (2013). GAGM: Genome assembly on GPU using mate pairs. High Performance Computing (HiPC), 2013 20th International Conference on. IEEE.

Jeck, W. R., Reinhardt, J. A., Baltrus, D. A., Hickenbotham, M. T., Magrini, V., Mardis, E. R., ... & Jones, C. D. (2007). Extending assembly of short DNA sequences to handle error. Bioinformatics, 23(21), 2942-2944.

Karp, A. H., & Flatt, H. P. (1990). Measuring parallel processor performance. Communications of the ACM, 33(5), 539-543. doi:10.1145/78607.78614

Katz, G. J., & Kider Jr, J. T. (2008, June). All-pairs shortest-paths for large graphs on the GPU. In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (pp. 47-55). Eurographics Association.

Kirk, D. B., & Wen-mei, W. H. (2012). Programming massively parallel processors: a hands-on approach. Newnes.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2012). PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code

generation. Parallel Computing, 38(3), 157-174. doi:10.1016/j.parco.2011.09.001

Kokot, M., Długosz, M., & Deorowicz, S. (2017). KMC 3: counting and manipulating k-mer statistics. Bioinformatics, 33(17), 2759-2761. doi:10.1093/bioinformatics/btx304

Kurtz, S., Narechania, A., Stein, J. C., & Ware, D. (2008). A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes. BMC genomics, 9(1), 517.

Lars Nyland & Stephen Jones, (2013). Understanding and Using Atomic Memory Operations, GTC-2013 NVIDIA.

Ladan-Mozes, E., & Shavit, N. (2004). An optimistic approach to lock-free FIFO queues. Springer Berlin Heidelberg; 117-131.

Li, D., Liu, C., Luo, R., Sadakane, K., & Lam, T. (2015). MEGAHIT: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. Bioinformatics, btv033.

Li & Ruiqiang (2010) "De novo assembly of human genomes with massively parallel short read sequencing." Genome research 202() 265-272.

Li, Y. (2015). MSPKmerCounter: a fast and memory efficient approach for k-mer counting. arXiv preprint arXiv;1505.06550.

Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., & Suri, S. (2013). Memory efficient minimum substring partitioning. Proceedings of the VLDB Endowment, 6(3), 169-180. doi:10.14778/2535569.2448951

Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., ... & Fan, W. (2012). Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. Briefings in functional genomics, 11(1), 25-37

Lipman, D., & Pearson, W. (1985). Rapid and sensitive protein similarity searches. Science, 227(4693), 1435-1441. doi:10.1126/science.2983426

Lu, M., Luo, Q., Wang, B., Wu, J., & Zhao, J. (2013). GPU-accelerated bidirected De Bruijn graph construction for genome assembly. In Web Technologies and Applications (pp. 51-62). Springer Berlin Heidelberg.

Luo & Ruibang "SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler." Gigascience 1.1 (2012): 18.

Luo, J., Wang, J., Li, W., Zhang, Z., Wu, F., Li, M., & Pan, Y. (2015). EPGA2: memory-efficient de novo assembler. Bioinformatics. doi:10.1093/bioinformatics/btv487

Mahmood, S. F., & Rangwala, H. (2011). Gpu-euler: Sequence assembly using gpgpu. High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on. IEEE.

Marçais, G., & Kingsford, C. A. (2011). fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics; 27(6), 764-770.
Martin, J. A., & Wang, Z. (2011). Next-generation transcriptome assembly.Nature Reviews Genetics, 12(10), 671-682.

Melsted, P., & Pritchard, J. K. (2011). Efficient counting of k-mers in DNA sequences using a bloom filter. BMC bioinformatics, 12(1), 333.

Meng, J., Wang, B., Wei, Y., Feng, S., & Balaji, P. (2014). SWAP-Assembler: scalable and efficient genome assembly towards thousands of cores. BMC Bioinformatics, 15(Suppl 9). doi:10.1186/1471-2105-15-s9-s2

Merrill, D., Garland, M., & Grimshaw, A. (2012, February). Scalable GPU graph traversal. In ACM SIGPLAN Notices (Vol. 47, No. 8, pp. 117-128). ACM.

Metzker, M. L. (2010). Sequencing technologies—the next generation. Nature Reviews Genetics, 11(1), 31-46.

Miller, J. R., Delcher, A. L., Koren, S., Venter, E., Walenz, B. P., Brownley, A., ... & Sutton, G. (2008). Aggressive assembly of pyrosequencing reads with mates. Bioinformatics, 24(24), 2818-2824.

Miller, J. R., Koren, S., & Sutton, G. (2010). Assembly algorithms for next-generation sequencing data. Genomics, 95(6), 315-327.

Muggli, M. D., Bowe, A., Noyes, N. R., Morley, P. S., Belk, K. E., Raymond, R., . . . Boucher, C. (2017). Succinct colored de Bruijn graphs. Bioinformatics, 33(20), 3181-3187. doi:10.1093/bioinformatics/btx067

Mullikin, J. C., & Ning, Z. (2003). The phusion assembler. Genome research,13(1), 81-90.

Munshi, A. (2009). The opencl specification. Khronos OpenCL Working Group, 1, l1-15.

Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., ... & Venter, J. C. (2000). A whole-genome assembly of Drosophila. Science, 287(5461), 2196-2204.

Nickolls, J., & Dally, W. J. (2010). The GPU computing era. IEEE micro, 30(2), 56-69.

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. Queue, 6(2), 40-53.

NVIDIA_OpenCL_BestPracticesGuide. Retrieved May 05, 2016, from https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

OpenCL 2.2 API Specification. Retrieved May 12, 2017, from https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.html

OpenCL Optimization Guide. Retrieved May 05, 2016, from http://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/

OpenMP, A. R. B. (2008). OpenMP application program interface, v. 3.0. OpenMP Architecture Review Board.

Pearson, W. R., & Lipman, D. J. (1988). Improved tools for biological sequence comparison. Proceedings of the National Academy of Sciences, 85(8), 2444-2448. doi:10.1073/pnas.85.8.2444

Peng, Y., Leung, H. C., Yiu, S. M., & Chin, F. Y. (2010, January). IDBA–a practical iterative de Bruijn graph de novo assembler. In Research in Computational Molecular Biology (pp. 426-440). Springer Berlin Heidelberg.

Pérez, N., Gutierrez, M., & Vera, N. (2016). Computational Performance Assessment of k-mer Counting Algorithms. Journal of Computational Biology, 23(4), 248-255. doi:10.1089/cmb.2015.0199

Pevzner, P. A., Tang, H., & Waterman, M. S. (2001). An Eulerian path approach to DNA fragment assembly. Proceedings of the National Academy of Sciences, 98(17), 9748-9753.

Pop, M., Phillippy, A., Delcher, A. L., & Salzberg, S. L. (2004). Comparative genome assembly. Briefings in bioinformatics, 5(3), 237-248.

Pop, M. (2009). Genome assembly reborn: recent computational challenges. Briefings in bioinformatics, 10(4), 354-366.

Pop, M., Salzberg, S. L., & Shumway, M. (2002). Genome sequence assembly: Algorithms and issues. Computer, 35(7), 47-54.

Ramtin Shams and R. A. Kennedy., (2007). Efficient histogram algorithms for NVIDIA CUDA compatible devices, Semantic Schoolar, 1-7.

Rizk, G., Lavenier, D., & Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. Bioinformatics, 29(5), 652-653. doi:10.1093/bioinformatics/btt020

Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M., & Yorke, J. A. (2004). Reducing storage requirements for biological sequence comparison.Bioinformatics, 20(18), 3363-3369.

Roy, R. S., Bhattacharya, D., & Schliep, A. (2014). Turtle: Identifying frequent k-mers with cache-efficient algorithms. Bioinformatics; btu132.

Rudy, G. (2010). A hitchhiker's guide to next-generation sequencing. xiii, 13, 14

Salikhov, K., Sacomoto, G., & Kucherov, G. (2013). Using Cascading Bloom Filters to Improve the Memory Usage for de Brujin Graphs. Lecture Notes in Computer Science Algorithms in Bioinformatics, 364-376. doi:10.1007/978-3-642-40453-5_28

Schatz, M. C., Trapnell, C., Delcher, A. L., & Varshney, A. (2007). High-throughput sequence alignment using Graphics Processing Units. BMC bioinformatics, 8(1), 474.

Shi, H., Schmidt, B., Liu, W., & Müller-Wittig, W. (2010). A parallel algorithm for error correction in high-throughput short-read data on CUDA-enabled graphics hardware. Journal of Computational Biology, 17(4), 603-615.

Simpson, J. T., & Durbin, R. (2012). Efficient de novo assembly of large genomes using compressed data structures. Genome research, 22(3), 549-556.

Snir, M. (Ed.). (1998). MPI--the Complete Reference: The MPI core (Vol. 1). MIT press.

Tarditi, D., Puri, S., & Oglesby, J. (2006, October). Accelerator: using data parallelism to program GPUs for general-purpose uses. In ACM SIGARCH Computer Architecture News (Vol. 34, No. 5, pp. 325-335). ACM.

Trapnell, C., & Schatz, M. C. (2009). Optimizing data intensive GPGPU computations for DNA sequence alignment. Parallel computing, 35(8), 429-440.

Vera N., Gutierrez M. & Perez J. (2016). Computational performance assessment of minimizers uses in counting k-mers. Australian Journal of Basic and Applied Sciences. 10(8), 71-78.

Vera-Parra, Pérez-Castillo & Rojas-Qunitero. (2015) Performance assessment for main stages in genomic and transcriptomic data processing based upon reads from illumina sequencing technologies. International Journal of Applied Engineering Research, 10(14), 34670-34674.

Vera-Parra, Pérez-Castillo & Rojas-Qunitero. (2015) Performance Assessment by Stages of Main Genomic De-Novo Assemblers Based Upon De Bruijn Graphs, Life Science Journal, 12(12), 13-21.

Warnke-Sommer, J. D., & Ali, H. H. (2017). Parallel NGS Assembly Using Distributed Assembly Graphs Enriched with Biological Knowledge. 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). doi:10.1109/ipdpsw.2017.143

Warren, R. L., Sutton, G. G., Jones, S. J., & Holt, R. A. (2007). Assembling millions of short DNA sequences using SSAKE. Bioinformatics, 23(4), 500-501.

Wen-Mei, W. H. (2011). GPU Computing Gems Emerald Edition. Elsevier.

Wick R., (2016). Effect of kmer size. April 29, 2016, from https://github.com/rrwick/Bandage/wiki/Effect-of-kmer-size

Williams, S., Waterman, A., & Patterson, D. (2009). Roofline. Communications of the ACM, 52(4), 65. doi:10.1145/1498765.1498785

Wolfe, M. (2013). The openacc application programming interface, version 2.0.

Yu, H., & Rauchwerger, L. (2014). Adaptive reduction parallelization techniques. 25th Anniversary International Conference on Súpercomputing Anniversary Volume -. doi:10.1145/2591635.2667180

Yang X, Chockalingam SP, Aluru S (2013) A survey of error-correction methods for next-generation sequencing. Brief Bioinform 14: 56–66.

Zerbino, D. R., & Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome research, 18(5), 821-829.

Zhang, Q., Pell, J., Canino-Koning, R., Howe, A. C., & Brown, C. T. (2014). These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. PLoS One; vol. 9, no 7.

Zhou, J., Liu, X., Stones, D. S., Xie, Q., & Wang, G. (2011). MrBayes on a graphics processing unit. Bioinformatics, 27(9), 1255-1261.