

PARALELIZACIÓN DE ALGORITMOS DE COMPRESIÓN DE SECUENCIAS GENÓMICAS UTILIZANDO UNIDADES DE PROCESAMIENTO GRÁFICO GPU

Informe Final Practica Académica Presentado Como Requisito Para Optar al
Título de Ingeniero Electrónico

Modalidad Proyecto de Investigación

Jose Fernando Vargas Rodriguez



UNIVERSIDAD DE ANTIOQUIA
1803
FACULTAD DE INGENIERÍA

Prof. Sebastián Isaza Ramírez
Asesor Interno

**UNIVERSIDAD DE ANTIOQUIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA Y
TELECOMUNICACIONES
ENERO DE 2017
MEDELLÍN**

Índice

Resumen	3
1. Introducción	3
2. Objetivos	4
2.1. Objetivo General	4
2.2. Objetivos Específicos	4
3. Marco Teórico	4
3.1. Overlapping Reads Compression with Minimizers (ORCOM)	5
3.1.1. Binning	6
3.1.2. Compression	6
3.2. GPU (Graphics Processing Unit)	6
3.2.1. Arquitectura	7
3.2.2. Modelo de Ejecución	8
3.2.3. Jerarquía de Memoria	8
3.3. CUDA (Compute Unified Device Architecture)	10
3.3.1. Computación Heterogénea	10
3.3.2. Modelo de Programación	10
3.3.3. Kernels	11
3.3.4. Jerarquía de Hilos	11
3.3.5. Jerarquía de Memoria	11
4. Metodología	12
4.1. Análisis	12
4.2. Diseño e Implementación	16
4.3. Pruebas	21
5. Resultados y Análisis	22
5.1. Prueba 1	22
5.2. Prueba 2	25
6. Conclusiones	27
Referencias	28
Anexos	30
A. Códigos	30
B. Tablas de tiempos de ejecución - Prueba 2	33

PARALELIZACIÓN DE ALGORITMOS DE COMPRESIÓN DE SECUENCIAS GENÓMICAS UTILIZANDO UNIDADES DE PROCESAMIENTO GRÁFICO GPU

Resumen

En los últimos años la secuenciación de datos genómicos ha llegado a una etapa en la cual se puede realizar con rapidez y mayor exactitud, esto infiere un crecimiento del tamaño de datos y en sí de la cantidad de archivos fastq resultantes de la secuenciación. Estos archivos no son prácticos de almacenar ni de transportar, por esto existen muchas herramientas programadas para comprimir datos genómicos. En este proyecto se eligió la herramienta de compresión ORCOM y se mejoró el tiempo de ejecución de la etapa de Binning de esta logrando un Speedup de hasta 1.433. Esto se logró por medio de un mapeo en una GPU de algunas funciones que tenían una carga computacional grande en la CPU y por ende consumían demasiado tiempo de ejecución. El algoritmo creado se elaboró por medio de la herramienta CUDA y se usaron Streams para mejorar el envío de datos hacia la GPU.

1. Introducción

El estudio del ADN y el genoma humano ha llevado al mundo a desarrollar nuevas técnicas de almacenamiento de datos, esto dado a que una secuenciación completa del genoma humano puede arrojar datos que tienen un tamaño promedio de cientos de Gigabytes; los cuales se almacenan en archivos fastq para su posterior procesamiento en diversas aplicaciones bioinformáticas.

La cantidad de datos suministrados por el proceso de secuenciación son muy elevados para ser transmitidos o almacenados de forma ágil; por esta razón, muchas investigaciones se han enfocado en encontrar una forma de almacenar eficientemente datos genómicos, puesto que se obtienen más datos de los que se pueden almacenar y/o transportar. Esto se da por un crecimiento exponencial en las bases de datos debido al reciente abaratamiento de costos del proceso de secuenciación.

El proceso de compresión de los datos genómicos puede tardar demasiado tiempo en ejecución en una maquina con pocos recursos. Durante este proyecto se utilizará una GPU para acelerar este proceso mediante el conjunto de herramientas de programación de NVIDIA (CUDA), ya que se puede hacer uso del alto nivel de paralelismo de datos que las GPU presentan en su arquitectura.

Hasta el momento se han desarrollado varias técnicas utilizadas en diferentes algoritmos para que las secuencias genómicas utilicen poco espacio en disco, algunas de estas técnicas son: DSRC2 [7], Quip [8], FQZComp [9], Scale [10], SR-comp [11], ReCoil [12] y BWT-SAP [13], entre otras. Estos algoritmos son seguros

y estables pero su tiempo de ejecución es alto y el archivo resultante después de la compresión aún sigue teniendo un tamaño significativo. Recientemente se ha desarrollado un nuevo algoritmo que mejora el tiempo de ejecución de la compresión, y la tasa de compresión es relativamente buena; dicho algoritmo tiene por nombre "Overlapping Reads Compression with Minimizers" (ORCOM) [2], y está basado en el algoritmo para el conteo de k-mers "KMC 2" [1]. El código implementado en este proyecto se basa en la técnica que utiliza el algoritmo ORCOM, la cual se fundamenta en determinar las redundancias de solapamientos entre los datos genómicos alojados en un archivo fastq y agruparlos en archivos "bin". A partir de estas similitudes y redundancias los archivos "bin" se usan para comprimir los datos y almacenarlos en el menor espacio de disco posible. En este documento se expone la paralelización de algunos módulos en la etapa de Binning de ORCOM, esto programado en CUDA para ser implementado en unidades de procesamiento gráfico GPU. También se presenta la arquitectura, el modelo de ejecución y la jerarquía de memoria de una GPU e igualmente se introducen conceptos de la herramienta de programación CUDA.

Con el algoritmo desarrollado para la GPU se busca reducir el tiempo empleado para la compresión de datos genómicos (Etapa de Binning) y así almacenarlos y transportarlos de una forma más eficaz.

2. Objetivos

2.1. Objetivo General

Desarrollar un algoritmo paralelo basado en la técnica usada en el algoritmo ORCOM utilizando unidades de procesamiento gráfico GPU, para ser usado dentro de un sistema de compresión de archivos que contienen información genómica.

2.2. Objetivos Específicos

- Desarrollar un algoritmo paralelo para GPU capaz de procesar datos genómicos para finalmente agruparlos en archivos "Bin" de acuerdo a redundancias de solapamientos y similitudes.
- Desarrollar pruebas en unidades de procesamiento gráfico GPU donde se compruebe el correcto funcionamiento del algoritmo y su desempeño.

3. Marco Teórico

Desde el punto de vista químico el ADN es una molécula conformada por ácidos nucleótidos ubicada en el núcleo de cada célula, y esta contiene todas las instrucciones genéticas utilizadas para el desarrollo de cada ser viviente. Los estudios que se llevan a cabo analizando la molécula de ADN con el proceso

de secuenciación van desde enfermedades hereditarias, determinar la ascendencia de una persona e inclusive la investigación forense.

El proceso de secuenciación consta de varios métodos bioquímicos y fotográficos que se llevan a cabo con el fin de determinar una sucesión de letras que representan el orden de los nucleótidos bases en una molécula de ADN, estos son: Citosina (C), Adenina (A), Guanina (G) y Timina (T). Cada una de las bases se adhiere simétricamente con su base afín (C-G, A-T), por esta razón durante el proceso de secuenciación solo se toma una de las dos hélices que conforman la molécula de ADN. Estos datos son almacenados en archivos fastq para su posterior análisis.

3.1. Overlapping Reads Compression with Minimizers (ORCOM)

ORCOM es un algoritmo implementado con orientación a objetos (C/C++), encargado de comprimir secuencias de ADN almacenadas en archivos fastq y consta de dos etapas de procesamiento, Binning y Compression. La primera etapa (Binning) se fundamenta en determinar las redundancias de solapamientos entre los datos genómicos alojados en un archivo fastq y agruparlos en archivos "Bin" (Binning). La segunda etapa (Compression) consta de tomar los archivos "Bin" creados, y a partir de estas similitudes y redundancias halladas en la etapa de binning comprimir los datos genómicos. Ver Figura 1.

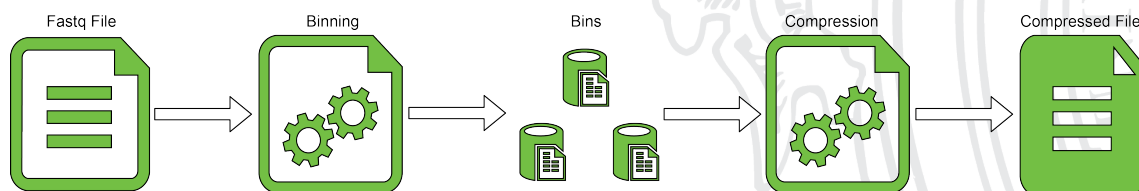


Figura 1: Proceso del algoritmo ORCOM.

A continuación se muestran varias definiciones importantes para comprender este proceso:

- **Read:** Cadena de caracteres de longitud L que representa una pequeña parte de la secuencia genómica, la cual está almacenada en un archivo fastq que contiene muchos Reads.
- **k-mer:** Sub-cadena de caracteres de longitud k contenida en un Read. Representa un fragmento en el Read.
- **Minimizer:** k-mer lexicográficamente más pequeño que se puede encontrar dentro de un Read.
- **Signatura (Minimizer Canónico):** Minimizado lexicográficamente más pequeño hallado entre el Minimizado de un Read y su complemento inverso.
- **Bin:** Archivo en memoria que guarda Reads organizados por similitudes y solapamientos (Signaturas).

- **Complemento inverso de Read:** Es la forma opuesta del Read organizada de atrás hacia adelante y cambiando su base principal por su base afín. Ej.: ACCGTAC ->GTACGGT.

A continuación se exponen con más detalle las etapas del algoritmo ORCOM:

3.1.1. Binning

Durante la primera etapa del método utilizado en el algoritmo ORCOM se hacen análisis estadísticos por segmentos a los datos almacenados en el archivo fastq, los cuales son Reads. Cada Read de longitud L se toma por separado y se procesa para hallar todos sus k-mers de longitud K. Con la ecuación (1) podemos saber el número de k-mers en un Read, donde N es el número de k-mers en el Read:

$$N = L - K + 1 \quad (1)$$

Entre los k-mers encontrados en el Read se calcula el Minimizer. En el algoritmo ORCOM existe la posibilidad de verificar cada Minimizer con el Minimizer del complemento inverso del Read, en este caso el Minimizer lexicográficamente más pequeño será la Signatura, pero en otra situación el Minimizer hallado será la Signatura del Read. En cada Read la Signatura representa un grupo de Reads similares, los cuales se agrupan por conjuntos para su posterior reordenamiento y almacenamiento en los archivos "Bin".

Existen dos tipos de archivos "Bin" resultantes de la etapa de Binning, los archivos "*.bdna" que contienen los conjuntos de Reads codificados y los archivos "*.bmeta" los cuales contienen la meta información de los Reads.

3.1.2. Compression

La segunda etapa en la técnica utilizada por ORCOM es el proceso de compresión de los datos almacenados en los archivos "Bin". Como entrada, toma los archivos producidos por la etapa de Binning: "*.bdna" y "*.bmeta" y genera dos archivos de salida: "*.cdna", que contiene secuencias comprimidas y archivo "*.cmeta", que contiene meta-información del archivo.

No se profundiza en esta etapa ya que el interés del proyecto se centra en la etapa de Binning, pero se puede encontrar mas información en el artículo "Disk-based compression of data from genome sequencing" [2].

3.2. GPU (Graphics Processing Unit)

Una GPU es un coprocesador dedicado al procesamiento de gráficos y a operaciones de punto flotante. Gracias a que cuentan con un alto grado de paralelismo de datos, las GPU se usan para disminuir la carga de procesamiento en el procesador central (CPU) y así acelerar la ejecución de un determinado programa, si su algoritmo lo permite.

Actualmente las GPU no solo se usan para el procesamiento de gráficos, también se usan para aplicaciones de propósito general GPGPU (General Purpose GPU) e igualmente para la computación de alto rendimiento debido a su gran rendimiento, su escalabilidad y su productividad.

3.2.1. Arquitectura

La GPU utilizada en este proyecto es la **Tesla M2090** la cual cuenta con una arquitectura Fermi diseñada por NVIDIA en el año 2010.

Una GPU Fermi contiene 512 CUDA Cores o Streaming Processors (SPs), 16 Streaming Multiprocessor, 768 KB de memoria cache L2, una memoria global y un archivo de registros de 32 bits. En una GPU Fermi cada SM contiene 32 Streaming Processors (SPs), 2 Warp Scheduler, 64 KB de memoria compartida, 48 KB memoria cache L1, una unidad de control SIMD, 4 Special Function Units (SFUs) y 16 unidades Load/Store. A continuación en la Figura 2 se muestra una GPU con arquitectura Fermi.

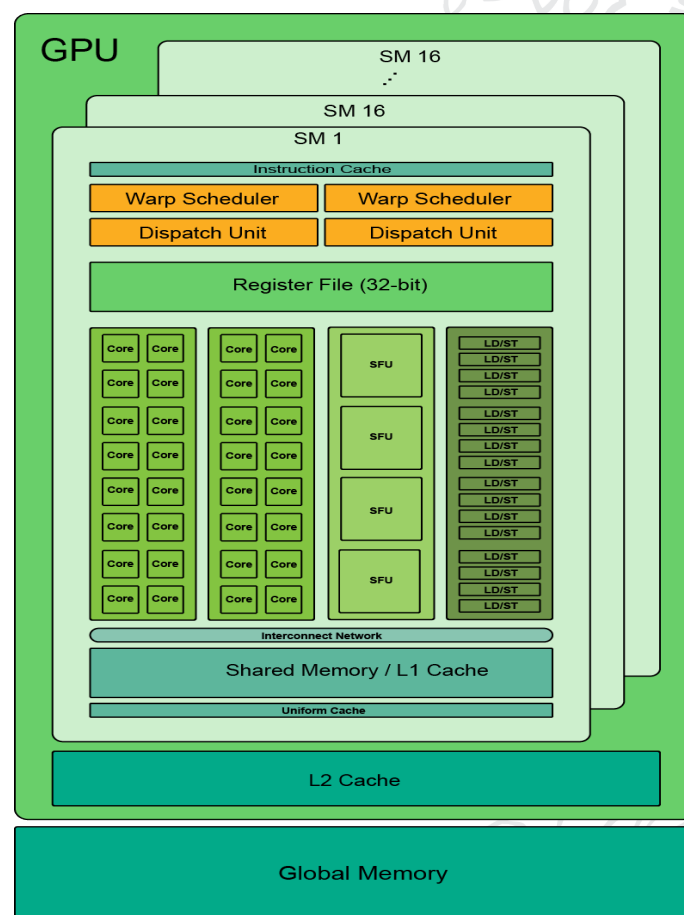


Figura 2: Arquitectura Fermi - GPU.

3.2.2. Modelo de Ejecución

El modelo de ejecución de una GPU se basa en el principio de la técnica SIMT (Single Instruction Multiple Threads), esto en la GPU se traduce en ejecutar una función en múltiples hilos que están contenidos en bloques y que igualmente estos están contenidos en una malla dentro de la GPU. Ver Figura 3.

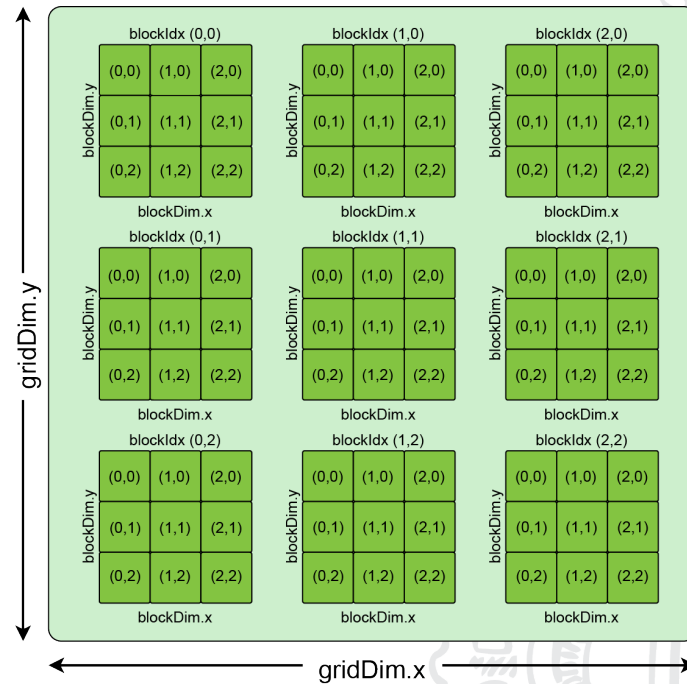


Figura 3: Modelo de ejecución.

Cuando se invoca una función en la GPU (kernel), las propiedades de la malla se describen mediante una configuración de ejecución que tiene una sintaxis especial en CUDA, donde se envían el número de bloques y el número de hilos en cada bloque para ser ejecutados. Los hilos se ejecutan concurrentemente en los bloques, para así formar una malla como la mostrada en la Figura 3. El soporte para el paralelismo dinámico en CUDA amplía la capacidad de configurar, iniciar y sincronizar en las nuevas mallas a los subprocesos que se ejecutan en el dispositivo, por ejemplo funciones llamadas por los kernels.

Los SMs se encargan de crear, programar y ejecutar grupos de 32 Hilos llamados Warps. Todos los Hilos que componen un Warp comienzan en la misma dirección de programa, pero cada uno cuenta con su propio registro de instrucciones y registro de estado, lo que se denomina el contexto del Warp y están gestionados por los Warps Schedulers vistos en la Figura 2.

3.2.3. Jerarquía de Memoria

Una GPU se compone de un conjunto de memorias que tienen una función específica para agilizar el acceso a la memoria de cada hilo y permitir la cooperación de los algoritmos en cada ejecución. El acceso a las memorias

puede ser visto de forma ordenada en la Figura 4.

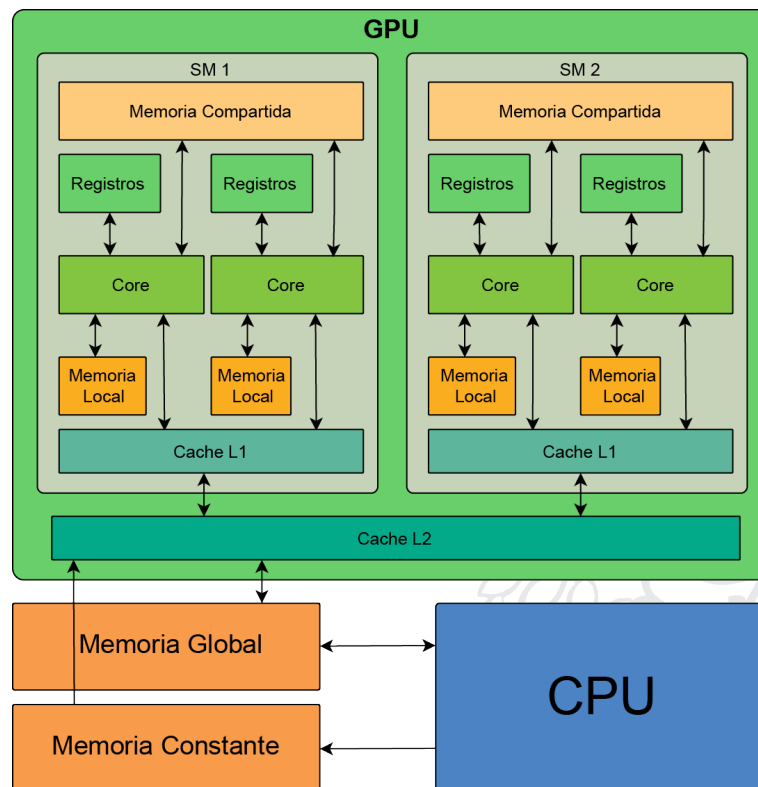


Figura 4: Jerarquía de Memoria

- **Registros:** Es la memoria mas rápida pero la mas pequeña, cada hilo tiene su propia memoria de registros y solo puede ser accedida por este.
- **Memoria Local:** Es una memoria destinada únicamente a la información local de cada hilo. Similar al concepto de memoria virtual, cuando un hilo excede su número de registros accede a esta memoria.
- **Memoria Compartida:** Es una memoria interna en cada SM, por lo tanto puede ser accedida por cualquier hilo mapeado en dicho SM. Es especialmente útil para la comunicación entre hilos, pero su acceso debe ser sincronizado para evitar "Data Hazards".
- **Memoria Global:** Está alojada en la memoria DRAM externa, cumple el papel de unidad de almacenamiento masivo de la GPU. Permite la comunicación CPU-GPU, ya que es la única memoria accesible desde la CPU. Es accesible además por todos los SMs que componen la GPU. Normalmente posee una capacidad de almacenamiento del orden de los GBytes.
- **Memoria Constante:** Es una memoria de sólo lectura para un programa que se ejecuta en un SM y puede ser escrita únicamente desde la CPU. Está alojada en la DRAM externa.

3.3. CUDA (Compute Unified Device Architecture)

Según la empresa NVIDIA: “CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema”. En otras palabras, CUDA es un conjunto de herramientas de programación paralela creadas por la empresa NVIDIA, la cual permite a los desarrolladores implementar variaciones del lenguaje C/C++ para codificar algoritmos en una GPU.

3.3.1. Computación Heterogénea

La computación heterogénea se refiere a sistemas que utilizan más de un tipo de procesador, mayormente en estos sistemas se utilizan procesadores especializados para realizar tareas particulares.

En este proyecto se utiliza un sistema heterogéneo basado en una CPU y una GPU. CUDA establece el nombre de “Host” a la CPU, la cual ejecuta el código secuencialmente; e igualmente proporciona el nombre de “Device” para referirse a la GPU, la cual se utiliza como coprocesador y ejecuta el código en paralelo para realizar tareas particulares y disminuir la carga en el Host.

Una GPU puede en ciertos programas acelerar la ejecución de estos de forma bastante eficiente con solo mapear a la GPU una porción pequeña de código, por ejemplo un código reescrito en CUDA puede ser inferior al 5%, pero consumir más del 50% del tiempo si no se migra a la GPU. Ver Figura 5.

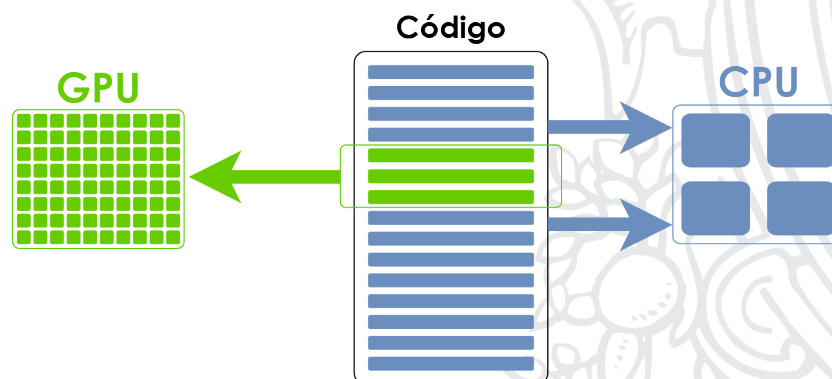


Figura 5: Aceleración en GPU.

3.3.2. Modelo de Programación

El modelo de programación CUDA consta de aprovechar al máximo el paralelismo de la GPU implementando programas transparentes y escalables. Una función, en este caso llamada “kernel” se ejecuta en la GPU de forma transparente pero en varios hilos al tiempo aprovechando la filosofía SIMD (Simple Instruction Multiple Data). Cada hilo se identifica con un ID único y este sirve para repartir el trabajo de procesamiento entre los diferentes hilos.

3.3.3. Kernels

Con la extensión del lenguaje C/C++ en CUDA podemos definir funciones (kernels) que se ejecutaran N veces en la GPU de forma paralela en M Streaming Processors. Los kernels se definen usando la declaración “__global__” [5] antes de la definición de su función como se muestra a continuación:

```
__global__ void kernel(int A, int B, int C)
{
    int i = threadIdx;
}
```

Donde “threadIdx” es el identificador único para cada hilo.

3.3.4. Jerarquía de Hilos

CUDA define una jerarquía de hilos, donde la ejecución de un hilo está contenida dentro de un bloque de hilos y este a su vez dentro de una malla de bloques. Se llama un kernel a ejecución de la siguiente forma:

```
kernel<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Donde “numBlocks” es el número de bloques en la malla y “threadsPerBlock” es la cantidad de hilos en cada bloque. Haciendo la multiplicación entre los dos valores podemos saber cuantos hilos en total vamos a ejecutar para ese kernel. A continuación se definen las etapas de la jerarquía de hilos (Ver Figura 3):

- **Malla:** Es un conjunto de bloques del mismo tamaño que ejecutan un mismo kernel paralelamente. La única forma de comunicación entre los bloques es usando la memoria global. Los bloques carecen de sincronización.
- **Bloque:** Es un conjunto de hilos que se ejecutan en un mismo SM. Estos hilos pueden ser sincronizados y pueden comunicarse usando la memoria compartida en cada SM.

3.3.5. Jerarquía de Memoria

Como vimos anteriormente en la sección 3.2.3., cada SP puede leer o guardar en diferentes memorias en la GPU. Ahora veremos que memorias podemos usar dependiendo de la jerarquía de hilos.

- **Malla:** Solo puede acceder a la memoria global y a la memoria de constantes.
- **Bloque:** Puede acceder a la memoria compartida y a las mismas memorias que la malla.
- **Hilo:** Puede acceder a las mismas memorias que los bloques, además de que cada hilo tiene su propio banco de registros y puede acceder a su memoria local.

4. Metodología

4.1. Análisis

■ Revisión de documentos

Se leyeron y estudiaron los artículos relacionados con: conteo de K-mers [1] [3], ORCOM [2], requerimientos para comparar y comprimir secuencias biológicas [4], guía de programación de CUDA en C [5].

■ Análisis estructural de códigos

Partiendo del artículo mencionado en la referencia [2], se analizó el código de ORCOM (Binning) realizando ingeniería inversa para su entendimiento y posterior paralelización en la GPU. El primer paso fue hacer un análisis de muy alto nivel en donde se definió el flujo de datos y la estructura general del programa. Ver Figura 6.

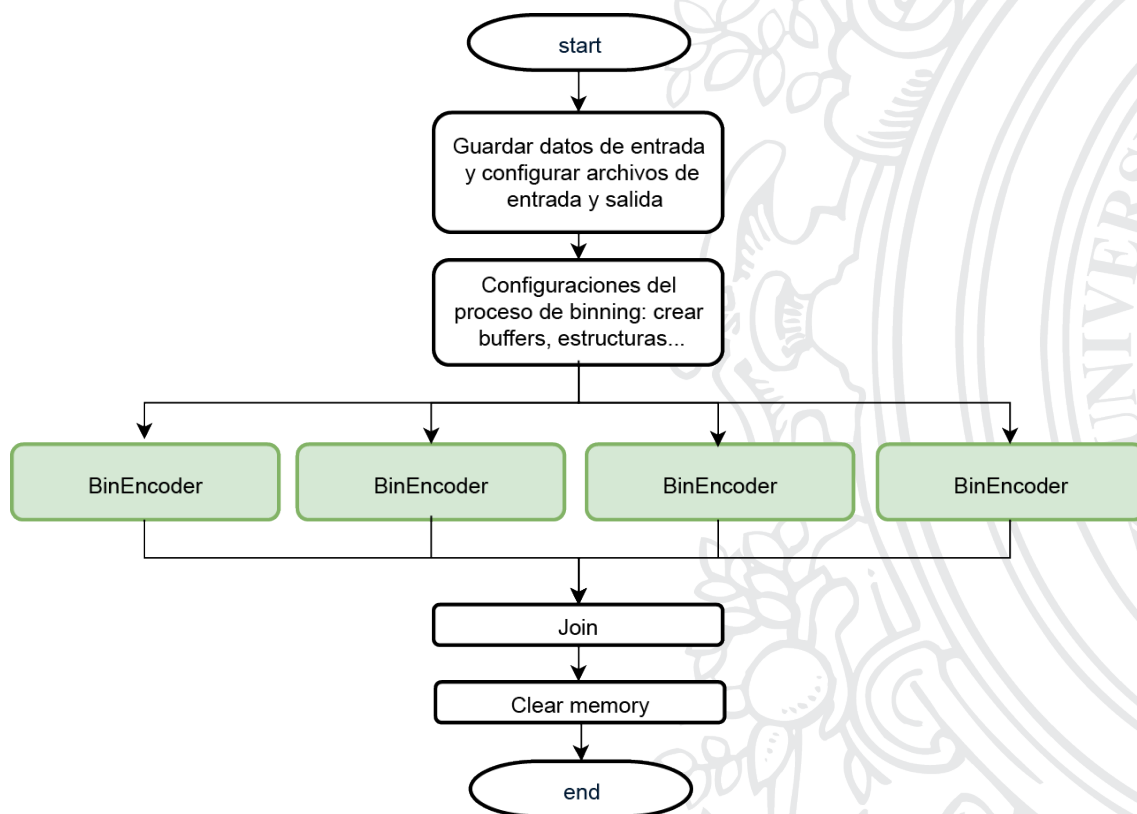


Figura 6: Diagrama de flujo etapa de Binning a Grosso Modo.

El segundo paso fue una revisión más profunda determinando que funciones desarrollaban trabajo de procesamiento de datos y que funciones hacían trabajo de almacenamiento de datos. Se encontró una función principal de procesamiento de datos llamada "BinEncoder", la cual se describe a continuación en la Figura 7.

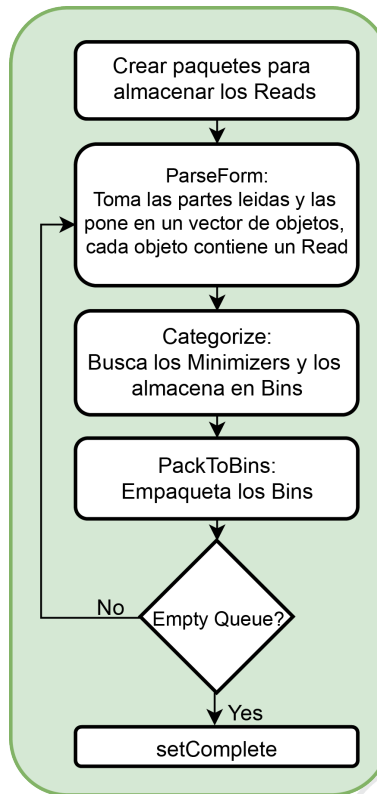


Figura 7: Diagrama de flujo de la función BinEncoder.

Dentro de la función “BinEncoder” se encuentran las funciones de más bajo nivel llamadas “ParseForm”, “Categorize” y “PackToBins”, donde respectivamente organizan los Reads en estructuras, buscan los Minimizers en cada Read y empaquetan los Bins organizados.

Por último se hizo un profiling de ORCOM (Binning) con un archivo de 103 MB para determinar que función tomaba más tiempo de ejecución en el programa. El profiling se hizo mediante tres herramientas: “gprof”, “perf” y “valgrind”.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
44.83	0.52	0.52	818422	0.00	0.00	DnaCategorizer::FindMinimizer(DnaRecord&)
12.93	0.67	0.15				strncmp
7.76	0.76	0.09	1609509	0.00	0.00	DnaParser::SkipLine()
7.76	0.85	0.09	409211	0.00	0.00	DnaPacker::StoreNextRecord(DnaRecord const&)
5.17	0.91	0.06	1	60.00	610.00	DnaCategorizer::DistributeToBins(std::vector<DnaRecord>&)
3.02	0.95	0.04	5052272	0.00	0.00	char* std::__find_if<char*, __gnu_cxx::__normal_iterator<char*, std::vector<DnaRecord>&>, bool (*)>
2.59	0.98	0.03	7957534	0.00	0.00	DnaCategorizer::IsMinimizerValid(unsigned int)
2.59	1.01	0.03	1454909	0.00	0.00	BitMemoryWriter::Put4Bytes(unsigned int)
2.59	1.04	0.03	1	30.00	690.00	DnaCategorizer::Categorize(std::vector<DnaRecord>&)
0.86	1.05	0.01	2354122	0.00	0.00	BitMemoryWriter::PutByte(unsigned char)

Figura 8: Profiling usando la herramienta “gprof”.

Samples: 6K of event 'cycles', Event count (approx.): 4520281494

Overhead	Command	Shared Object	Symbol
33,52%	orcom_bin	orcom_bin	[.] _ZN14DnaCategorizer13FindMinimizerER9DnaRecord
16,12%	orcom_bin	orcom_bin	[.] __mcount_internal
7,30%	orcom_bin	orcom_bin	[.] strcmp
6,74%	orcom_bin	orcom_bin	[.] _ZN9DnaPacker15StoreNextRecordERK9DnaRecordR15BitMem
6,11%	orcom_bin	orcom_bin	[.] _ZN9DnaParser8SkipLineEv
4,34%	orcom_bin	orcom_bin	[.] __mcount
4,27%	orcom_bin	orcom_bin	[.] _ZN14DnaCategorizer16IsMinimizerValidEjj
3,50%	orcom_bin	orcom_bin	[.] _ZN14DnaCategorizer16DistributeToBinsERSt6vectorI9Dn
3,31%	orcom_bin	orcom_bin	[.] _ZSt9_find_ifIPcN9__gnu_cxx5_ops17_Iter_equals_ite
2,72%	orcom_bin	orcom_bin	[.] _ZN14DnaCategorizer10CategorizeERSt6vectorI9DnaRecor
1,34%	orcom_bin	orcom_bin	[.] _ZSt16_introsort_loopIN9__gnu_cxx17__normal_iterato

Figura 9: Profiling usando la herramienta “perf”.

Incl.	Self	Called	Function	Loca
62.64	52.74	1 636 844	DnaCategorizer::FindMinimizer(Dn...	orco
11.18	10.50	3 273 688	DnaParser::SkipLine()	orco
10.74	7.23	818 422	DnaPacker::StoreNextRecord(Dna...	orco
9.66	6.32	15 915 068	DnaCategorizer::IsMinimizerValid(...	orco
5.22	5.22	42 662 775	__mcount_internal	orco
67.48	4.00	2	DnaCategorizer::DistributeToBins(...	orco
8.96	3.74	42 662 775	mcount	orco
2.74	2.74	8 643 468	strcmp	orco
76.10	1.29	2	DnaCategorizer::Categorize(std::ve...	orco
3.04	0.92	10 104 544	char* std::_find_if<>(char*, char*, ...	orco
12.08	0.80	2	DnaParser::ParseFrom(DataChunk ...	orco

Figura 10: Profiling usando la herramienta “valgrind”.

El resultado del profiling en cada una de herramientas fue el mismo, la función que más tiempo de ejecución gasta en ORCOM es FindMinimizer y toma alrededor del 45 % del tiempo de ejecución total. En la Figura 11 se muestra un diagrama Butterfly donde observamos el flujo de llamado de la función “FindMinimizer”.

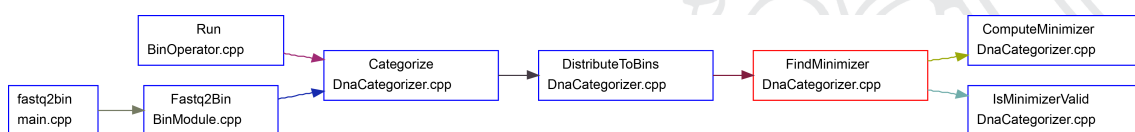


Figura 11: Diagrama Butterfly de la función “FindMinimizer”.

A la función “FindMinimizer” le ingresa una estructura llamada “DnaRecord” cuyos datos más relevantes son: la cadena de caracteres que representa el Read, el tamaño del Read y un booleano que indica si el Read está en su forma de complemento Inverso. Ver Figura 12.

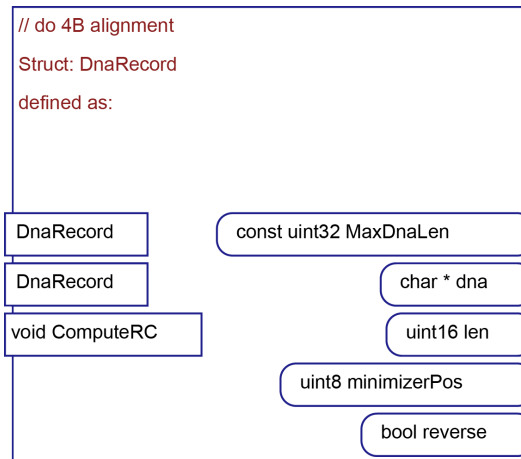


Figura 12: Declaración de la estructura “DnaRecord”.

Podemos observar a continuación el código de la función “FindMinimizer”:

```

uint32 DnaCategorizer::FindMinimizer(DnaRecord &rec_)
{
    uint32 minimizer = maxLongMinimValue;
    ASSERT(rec_.len >= params.signatureLen - params.skipZoneLen + 1);
    const int32 ibeg = 0;
    const int32 iend = rec_.len - params.signatureLen + 1 -
        params.skipZoneLen;

    for (int32 i = ibeg; i < iend; ++i){
        uint32 m = ComputeMinimizer(rec_.dna + i, params.signatureLen);
        if (m < minimizer && IsMinimizerValid(m, params.signatureLen))
            minimizer = m;
    }
    if (minimizer >= maxLongMinimValue)
        return nBinValue;

    return minimizer & (maxShortMinimValue - 1);
}

```

Podemos ver que en la función “FindMinimizer” se invocan dos funciones: “ComputeMinimizer” y “IsMinimizerValid”, las cuales se encargan respectivamente de encontrar el Minimizer de cada K-mer en el Read y verificar si es un Minimizador valido. A continuación podemos ver estas funciones.

```

uint32 DnaCategorizer::ComputeMinimizer(const char* dna_, uint32 mLen_)
{
    uint32 r = 0;
    for (uint32 i = 0; i < mLen_; ++i){
        if (dna_[i] == 'N')
            return nBinValue;

        ASSERT(dna_[i] >= 'A' && dna_[i] <= 'T');
        r <<= 2;
        r |= symbolIdxTable[(uint32)dna_[i]];
    }
    return r;
}

```

```

bool DnaCategorizer::IsMinimizerValid(uint32 minim_, uint32 mLen_)
{
    if (minim_ == 0)
        minim_ = 0;

    const uint32 excludeLen = 3;
    const uint32 excludeMask = 0x3F;
    const uint32 symbolExcludeTable[] = {0x00, 0x15, 0x2A, 0x3F};
    minim_ &= (1 << (2*mLen_)) - 1;
    bool hasInvalidSeq = false;

    for (uint32 i = 0; !hasInvalidSeq && i <= (mLen_ - excludeLen); ++i)
    {
        uint32 x = minim_ & excludeMask;
        for (uint32 j = 0; j < 4; ++j)
            hasInvalidSeq |= (x == symbolExcludeTable[j]);
        minim_ >>= 2;
    }
    return !hasInvalidSeq;
}

```

Las anteriores funciones son pequeñas y se pueden replicar en la GPU fácilmente con un grado de granularidad fino. Sabiendo que funciones se deben mapear a la GPU para conseguir acelerar el tiempo de ejecución en la etapa de Binning, podemos seguir con la siguiente fase del proyecto.

■ **Aprendizaje paralelismo (CUDA)**

Se experimentó con el modelo de programación implementado en CUDA para el lenguaje de programación C/C++ y el entorno de programación paralela para CUDA.

■ **Aprendizaje arquitectura GPU**

Se estudió a detalle la arquitectura de la GPU a utilizar (Fermi) y su importancia para desarrollar un programa que garantice el máximo desempeño de la GPU.

4.2. Diseño e Implementación

Antes de adentrarnos al diseño del algoritmo en CUDA, se explicará brevemente el funcionamiento de la porción de código a mapear en la GPU.

La función a mapear llamada "FindMinimizer" es llamada para encontrar el Minimizer en cada Read del archivo, por esta razón tiene como entrada una estructura que guarda la cadena de caracteres del Read, el tamaño del Read y un booleano que indica si el Read está en su forma de complemento Inverso (Ver Figura 12). Luego de entrar a la función "FindMinimizer" se calcula de la posición donde se comienza a iterar en el Read y número de iteraciones que se realizarán para encontrar el Minimizer, esto depende del número de k-mers en el Read y de la ecuación (1).

Después de calcular el número de iteraciones, se procede a encontrar el Minimizer en el Read en cada iteración con la función "ComputeMinimizer" y lo

convierte en binario con sus respectivos Id para ser comparado fácilmente con el resto de Minimizers, los Id son: A="00", C="01", G="10", T="11". Si en el k-mer se encuentra al menos una letra N, ese Minimizer no será válido.

Finalmente teniendo el Minimizer en cada iteración codificado en binario se verifica si es válido con la función "IsMinimizerValid". Los Minimizers inválidos serán los que tengan tres o más repeticiones del mismo símbolo seguidas, por ejemplo "AAA" = "000000", "CCC" = "010101", "GGG" = "101010" y "TTT" = "111111". De todos los Minimizers válidos, el Minimizador del Read en este caso llamado "Signature" será el lexicográficamente más pequeño de todos y será retornado por la función "FindMinimizer".

Estas explicaciones se ilustran mejor en la Figura 13, donde el Read de entrada a la función "FindMinimizer" es "ATCTAGGGA" y la longitud de los k-mers es 6. Podemos ver que los Minimizers de las iteraciones 3 y 4 no son válidos porque contienen tres repeticiones de la letra G.

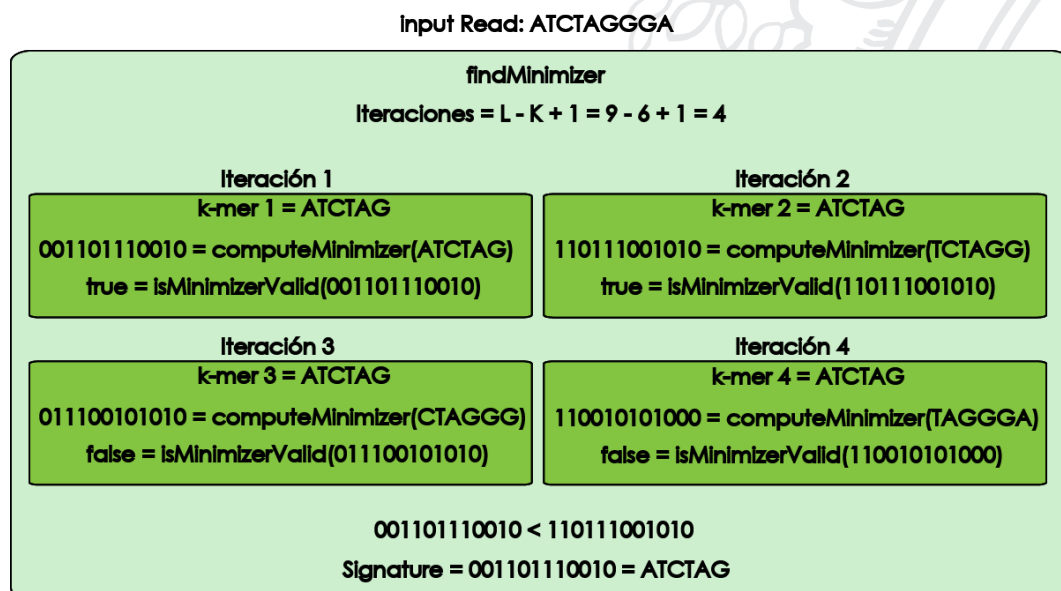


Figura 13: Elección de signature de Read

Teniendo claros los conceptos de programación en CUDA, la arquitectura de la GPU a utilizar y la porción de código para mapear en dicha GPU procedemos a diseñar el algoritmo paralelo.

■ Diseño de algoritmo en paralelo

Una secuencia común de datos genómicos en un archivo fastq puede tener un número de Reads en promedio del orden de 1×10^6 . El número de Reads es igual al número de veces que se llamará a la función "FindMinimizer". Tendiendo en cuenta que ORCOM también encuentra el Minimizer en el complemento inverso de cada Read vemos que la función "FindMinimizer" se llamaría el doble de veces que el número de Reads en el

archivo fastq, esto nos hace entender el por qué de el tiempo de ejecución gastado por esta función.

Como el número de llamados a la función a mapear es bastante alto, si queremos implementar esto en la GPU, se harían el mismo número de llamados a la función "FindMinimizer" como de transmisiones a la GPU, esto nos llevaría a un resultado opuesto al que queremos. Entonces se decidió llevar a la función a mapear a un nivel mas alto, en este caso la función "DistributeToBins" (Ver Figura 11), la cual recibe un arreglo de estructuras que contienen los datos genómicos. La función "DistributeToBins" busca el Minimizer en cada Read con la función "FindMinimizer" y almacena cada Read en el Bin correspondiente dependiendo del Minimizer Canónico hallado en cada uno de estos. Este proceso se puede ver gráficamente en la Figura 14.

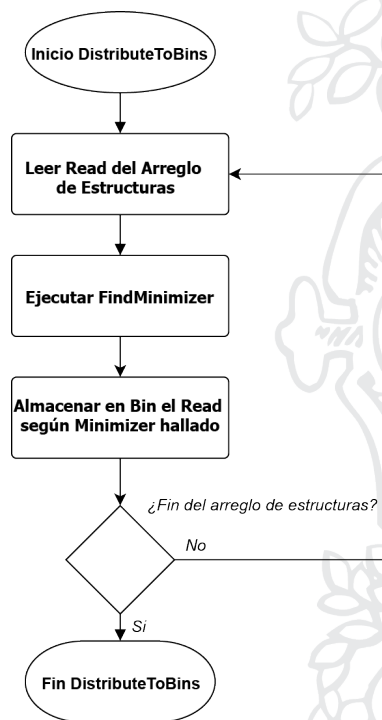


Figura 14: Diagrama de flujo - DistributeToBins

El algoritmo ORCOM lee el archivo fastq cada 256MB y luego llama a la función "DistributeToBins" para que haga su trabajo. Si un archivo fastq tiene como tamaño 1GB, la función a mapear se llamará solo cuatro veces. Esto en comparación al número de llamados de la función "FindMinimizer" es bastante poco.

Se planeó mapear el código con los siguientes requerimientos:

- Mapear tal cual las funciones "FindMinimizer", "ComputeMinimizer" y "IsMinimizerValid" con sus respectivos datos para su correcto funcio-

namiento. Estas funciones estarán alojadas en la memoria de la GPU pero no son un kernel lanzado desde la CPU.

- Concatenar todos los Reads recibidos en cada llamado de la función "DistributeToBins", esto en un arreglo de caracteres para hacer un solo envío a la GPU por cada 256MB. A esto le llamamos preparación de los Reads.
- El kernel lanzado se encargará de tomar el arreglo de Reads y ejecutar un numero de hilos como de Reads en la memoria de la GPU. Cada hilo llamará a la función "FindMinimizer" para encargarse de su respectivo Read y guardará el resultado en un arreglo de enteros que serán enviados a la CPU posteriormente, para que el código secuencial continúe su ejecución.
- Las lecturas y escrituras hechas por los hilos en la memoria de la GPU no pueden tener ningún problema, dado a que no leen ni escriben en la misma posición de memoria en ningún momento.

Durante el proyecto se diseñaron varias versiones de código resaltando los puntos anteriormente mencionados pero con modificaciones pequeñas que los diferencian. El reto de mapear el algoritmo secuencial en CUDA fue la transmisión de los datos leídos por la CPU, ya que estos datos son del orden de los GBytes tardamos mas en guardarlos en la memoria de la GPU que en ejecutar el kernel.

Se hicieron versiones del código desde CUDA1 hasta CUDA6 pero a continuación solo veremos las versiones mas relevantes:

- **CUDA2:** Se toman los Reads y su complemento inverso y se concatenan letra por letra en una cadena de caracteres, esta se envía a la GPU.
- **CUDA3:** Se concatenan los Reads y su complemento inverso en una cadena de caracteres por medio de la función memcpy, esta se envía a la GPU.
- **CUDA4:** La cadena de Reads se toma directamente desde un puntero en el programa y se incluye en la GPU el proceso de obtener el complemento inverso del Read.
- **CUDA6:** Se hace lo mismo que en la version CUDA4 pero esta vez las transmisiones de datos y la ejecución de los kernels se hacen por medio de Streams.

■ Implementación GPU

Las versiones anteriormente propuestas se implementaron tal cual se mencionó y pero la version CUDA6 es diferente a las demás dado a que tiene un pipeline implementado con Streams. A continuación se explicará el procedimiento de la implementación de la version CUDA6 y la forma detallada de la utilidad y el manejo de los Streams en CUDA.

CUDA ofrece una gestión de tareas concurrentes a través de Streams. Un Stream es una secuencia de comandos (posiblemente emitidos por hilos de Hosts diferentes) que se ejecutan en orden. Cada Stream se ejecuta concurrentemente con los demás. Para utilizar los Streams primero debemos crearlos con el siguiente código:

```
cudaStream_t stream[numStreams];
for (int i = 0; i < numStreams; ++i)
    cudaStreamCreate(&stream[i]);
```

Luego pasamos los datos del Device al Host y ejecutamos cada kernel en su respectivo Stream:

```
cudaStream_t stream[numStreams];
for (int i = 0; i < numStreams; ++i){
    cudaMemcpyAsync(inputDevPtr, hostPtr, size,
        cudaMemcpyHostToDevice, stream[i]);

    kernel<<<numBlocks, threadsPerBlock, 0, stream[i]>>>(A, B, C);
}
```

Por ultimo retornamos los datos generados y destruimos los kernels:

```
for (int i = 0; i < numStreams; ++i){
    cudaMemcpyAsync(hostPtr, outputDevPtr, size,
        cudaMemcpyDeviceToHost, stream[i]);

    cudaStreamDestroy(stream[i]);
}
```

En la version CUDA6 tenemos tres tareas que se pueden lanzar de forma concurrente gracias a los Streams, estas son: copiar los datos del Host al Device (memcpyHtoD), lanzar el kernel y retornar los datos del Device al Host (memcpyDtoH). En la Figura 15 podemos observar una comparación para estas tres tareas entre usar un solo Stream (Configuración por defecto) o usar un pipeline suponiendo que lanzamos 4 Streams.

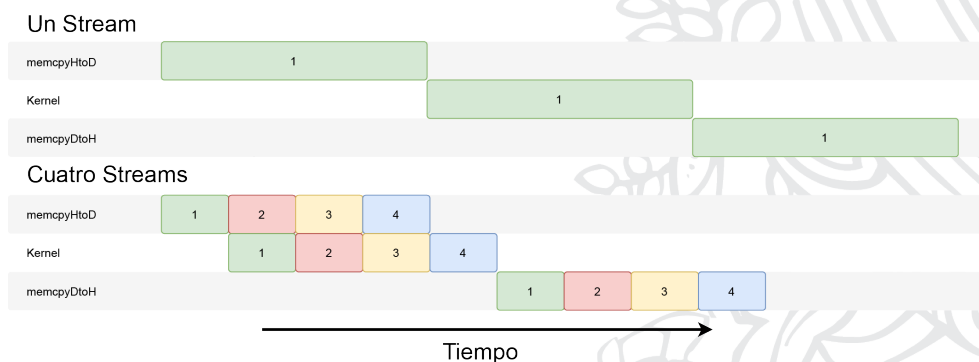


Figura 15: Ejecución de la GPU usando Streams.

En ámbitos generales se reduce el tiempo total de transmisión de datos memcpyHtoD y de ejecución de los kernels. El tiempo para la transmisión memcpyDtoH no se reduce ya que no podemos retornar los datos hasta que estemos seguros que estos han sido creados.

Usando los Streams podemos reducir el tiempo total de ejecución de la GPU por cada 256 MB leídos del archivo fastq, para esto creamos varios Streams (8 Streams para CUDA6) y en cada Stream se envía una pequeña parte del arreglo de Reads. Cada kernel lanzado en un Stream toma esa porción de datos enviados y la procesa. Finalmente se retornan los datos al Host para continuar su ejecución secuencial. Ver Anexo A.

La ejecución del código diseñado e implementado en este proyecto se puede entender mejor en la Figura 16, donde se muestra la línea de tiempo de una ejecución para la etapa de Binning de ORCOM, con las modificaciones hechas en la version CUDA6 y con un archivo de 100MB. Como los Streams no afectan la etapa de memcpyDtoH, se decidió no implementar esta con Streams.

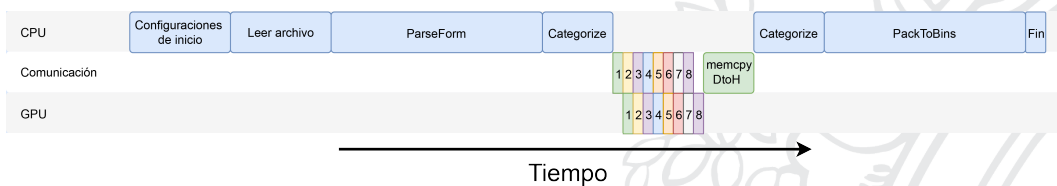


Figura 16: Ejecución de ORCOM modificado.

4.3. Pruebas

Se hicieron dos pruebas por separado, primero en la etapa de diseño e implementación y finalmente una prueba rigurosa del código final con diferentes archivos fastq. Estas pruebas se realizaron en una GPU Tesla M2090 y en un procesador Intel Xeon E5-2620. Las medidas de tiempo tomadas, se hicieron mediante el comando "time" de Linux, el cual retorna el tiempo total que le toma al programa ejecutarse.

- **Prueba 1:** Primero se hicieron pruebas de cada version del código para elegir cual de ellos fue el mejor. En estas pruebas a parte de medir el tiempo de ejecución total, también se midieron tiempos de la GPU (nvprof) y de partes del programa en la CPU con la función de C, gettimeofday() perteneciente a la librería "sys/time.h". Dichas pruebas se hicieron con los archivos mostrados en el Cuadro 1.

Archivo	Tamaño	Número de Reads	Longitud del Read
MicroBRE_3.fastq	103 MB	409.211	101
ERR251006.fastq	1.6 GB	6.478.764	101
ERR160123_1.fastq [15]	8 GB	31.394.858	101
SRR608815.fastq [16]	17 GB	129.029.571	36

Cuadro 1: Archivos de prueba.

Estos archivos de prueba se eligieron por su tamaño y se tomaron aleatoriamente de bases de datos genómicos, las cuales pueden ser fácilmente

encontradas en la web.

- **Prueba 2:** En segundo lugar y habiendo elegido el programa con el mejor tiempo de ejecución, se prepararon pruebas para medir si algunas características de los archivos fastq eran relevantes para el tiempo de ejecución del programa [6]. Dichas pruebas se hicieron con los archivos de los cuadros 2, 3 y 4.

	Archivo	Tamaño	Número de Reads	Long de Read
<15GB	ERR251006.fastq	1.6 GB	6.478.764	101
	ERR160123_1.fastq	8.2 GB	31.394.858	101
15a20GB	DRR001632_2.fastq [22]	20 GB	79.235.506	76
	SRR741411_1.fastq [17]	17 GB	67.021.831	101
>25GB	DRR008465_2.fastq [21]	56 GB	159.933.586	100
	DRR000966_1.fastq [20]	33 GB	123.206.768	111

Cuadro 2: Archivos de prueba por tamaño de archivo.

	Archivo	Tamaño	Número de Reads	Long de Read
Humanos	SRR741412_2.fastq [18]	15 GB	57.106.089	101
	SRR741411_1.fastq [17]	17 GB	67.021.831	101
Plantas	DRR008465_2.fastq [21]	56 GB	159.933.586	100
	DRR001632_2.fastq [22]	20 GB	79.235.506	76
Bacterias	DRR000966_1.fastq [20]	33 GB	123.206.768	111
	DRR000969.fastq [19]	14 GB	41.326.503	152

Cuadro 3: Archivos de prueba por especie.

	Archivo	Tamaño	Número de Reads	Long de Read
<50	SRR608815.fastq [16]	17 GB	129.029.571	36
50a100	DRR001632_2.fastq [22]	20 GB	79.235.506	76
>100	DRR000969.fastq [19]	14 GB	41.326.503	152

Cuadro 4: Archivos de prueba por longitud de Read.

Estos archivos se tomaron de la misma forma que los archivos de la prueba 1 (Bases de datos aleatorias), pero fueron elegidos por las características mencionadas.

5. Resultados y Análisis

5.1. Prueba 1

Para realizar estas pruebas y determinar la mejor version del código implementado tomamos los archivos mostrados en el Cuadro 1, ejecutamos los pro-

gramas y recogimos los mejores resultados.

Primero veamos las mejoras que se presentaron en el tiempo de ejecución de la función "FindMinimizer" mapeada en la GPU. Estas pruebas se realizaron con el archivo de 17 GB - SRR608815.fastq. Ver Cuadro 5.

Función	CPU	CUDA2	CUDA3	CUDA4	CUDA6	CUDA6 real
PrepReads	—	24,50	12,41	0,00	0,38	—
CUDA_memcpyHtoD	—	22,84	22,84	13,32	13,34	—
Kernel	—	4,73	4,79	5,68	9,29	—
CUDA_memcpyDtoH	—	2,46	2,46	2,46	2,48	—
Total	75,90	54,53	42,50	21,46	25,49	17,86

Cuadro 5: Tiempo de ejecución - FindMinimizer(seg).

En la Figura 17 podemos observar que en todas las versiones se mejoró la función mapeada.

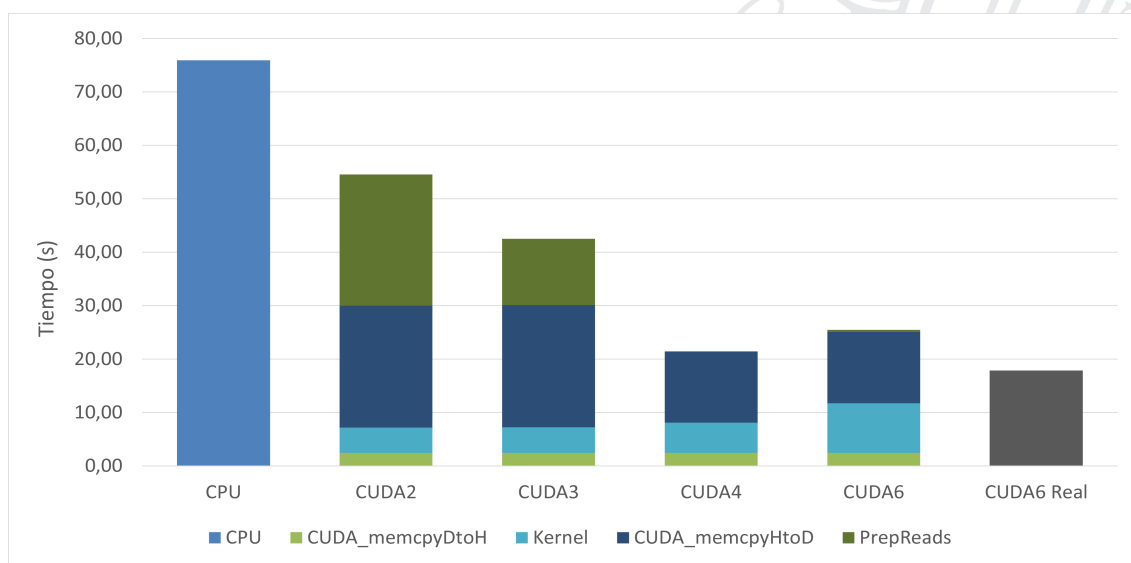


Figura 17: Mejoras de la función FindMinimizer.

Vemos que en la version CUDA2 el tiempo de preparación de los Reads es alto dado a que se concatenan letra por letra los Reads y su complemento inverso para su envío hacia la GPU. En la version CUDA3 el tiempo de preparación de los Reads disminuye gracias a la función memcpy, la cual concatena el Read completo con una sola instrucción. En la version CUDA4 se encuentra un arreglo donde se estaban concatenando los Reads en una función de ORCOM donde se leía del archivo fastq, entonces tomando este arreglo con un puntero se elimina la preparación de los Reads dado a que el coste de esta operación es despreciable para la CPU. En la version CUDA4 también se incorpora a la GPU la función que generaba el complemento inverso de cada Read, esto aumenta el tiempo de ejecución del kernel pero reduce considerablemente

el tiempo de transmisión Host-Device ya que se eliminan la mitad de los datos que se envían.

Observamos que en la version CUDA6 el tiempo de ejecución del kernel se incrementó con respecto a la version anterior como vemos en el Cuadro 5 resaltado en gris, aunque el tiempo de ejecución total del programa fue menor que su version anterior como vemos en la Figura 18. Esto se da por el profiling hecho por las herramientas de CUDA (nvprof). La herramienta nvprof toma los tiempos de cada kernel en cada Stream y en la Figura 18 se observa la sumatoria de todos ellos, pero los kernels se están ejecutando de forma concurrente y esto se refleja en el tiempo de ejecución total. Podemos ver con una barra gris una aproximación del tiempo de ejecución real de CUDA6 en la Figura 17.

En segundo lugar observemos que como consecuencia de la mejora en la función "FindMinimizer" se mejoraron los resultados en el tiempo de ejecución total de todas las versiones. Ver Cuadro 6.

Archivo		CUDA2	CUDA3	CUDA4	CUDA6
MicroBRE_3.fastq	CPU	2,17	2,23	2,19	2,15
	GPU	7,31	6,99	6,89	6,82
	SPEEDUP	0,297	0,319	0,318	0,315
ERR251006.fastq	CPU	26,91	27,05	27,15	26,98
	GPU	32,20	31,05	28,02	26,58
	SPEEDUP	0,836	0,871	0,969	1,015
ERR160123_1.fastq	CPU	130,56	132,05	130,48	130,56
	GPU	134,00	123,39	113,00	110,75
	SPEEDUP	0,974	1,070	1,155	1,179
SRR608815.fastq	CPU	223,23	223,16	223,15	223,20
	GPU	203,00	191,00	171,00	168,13
	SPEEDUP	1,100	1,168	1,305	1,328

Cuadro 6: Tiempo de ejecución total(seg).

Podemos observar mejor estos tiempos en la Figura 18 donde vemos que mientras avanzamos en las versiones, el tiempo de ejecución disminuye debido a las mejoras que se hacen al código.

También podemos observar que el tamaño del archivo es directamente proporcional a la diferencia en los tiempos de ejecución entre la CPU y la GPU. Vemos que en el archivo mas pequeño los tiempos de la GPU son mayores al de la CPU, esto dado que la GPU tarda de 5 a 10 segundos en poner a correr al primer kernel en cada ejecución del programa por reconocimientos de la API de CUDA sobre la GPU. En archivos de gran tamaño el tiempo de reconocimiento de la GPU es despreciable gracias a las mejoras obtenidas en el código en el tiempo de ejecución.

Por ultimo podemos concluir que la version que obtuvo el menor tiempo de

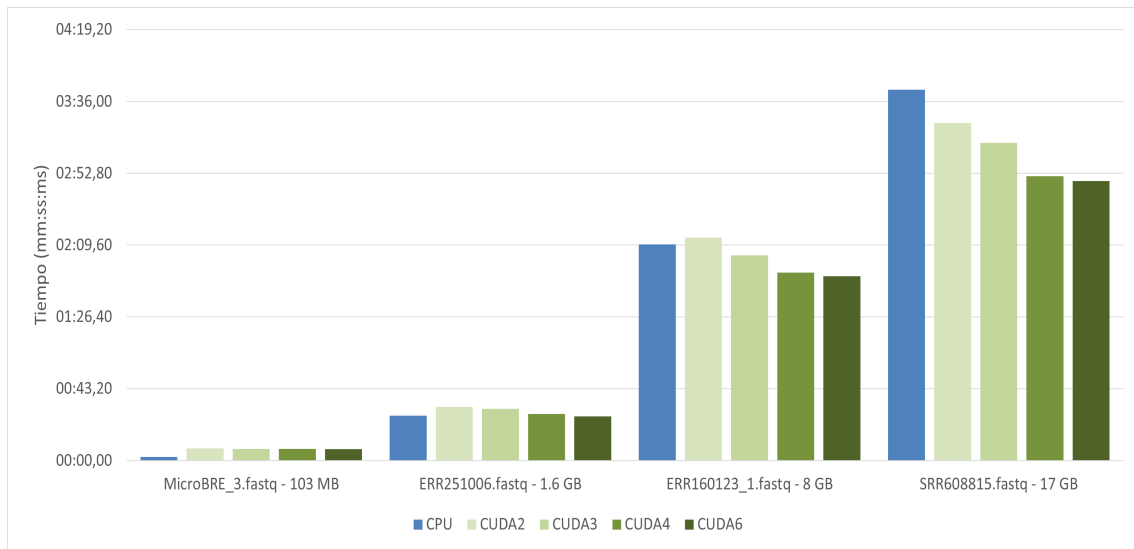


Figura 18: Tiempo de ejecución por tamaño de archivo.

ejecución y por ende un mejor speedup con respecto al tiempo total de ejecución secuencial en la CPU, fue la version CUDA6 gracias a la implementación de los Streams de CUDA. Esta se encuentra en el url de la referencia [14]

5.2. Prueba 2

Habiendo elegido la version del código mas eficiente, pasamos a la segunda prueba, donde se determina que variables en un archivo fastq influyen en el tiempo de ejecución del programa.

Ejecutando el programa varias veces con cada archivo de los cuadros 2, 3 y 4 se tomó el valor mínimo, la media y la desviación estándar del tiempo de ejecución. Los speedups mostrados en las Figuras 19, 20 y 21 se calcularon con los promedios de los tiempos de ejecución totales del programa. Ver Anexo B para tiempos de ejecución completos.

■ Tamaño de Archivo

En la Figura 19 podemos ver el speedup en tiempo de ejecución de cada tamaño de archivo del Cuadro 2 y se observa que mientras el archivo crece en tamaño el speedup también crece. Se puede concluir que un tamaño de archivo grande favorece el speedup del programa.

En el archivo de tamaño de 33 GB observamos un decremento del speedup con respecto a los demás vistos en la Figura 19, esto se debe a que dicho archivo tiene una longitud de los Reads mayor que todos los archivos de prueba. Se puede inferir que la longitud de Read influye en el speedup pero en la prueba de tamaño de Read podremos analizar esto detalladamente.

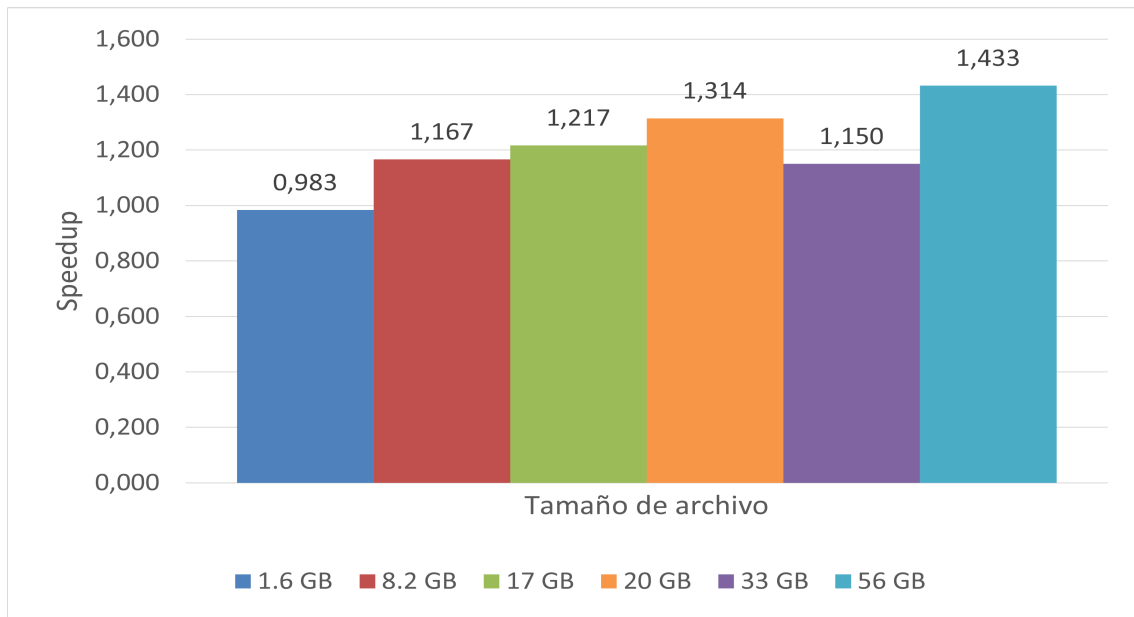


Figura 19: Speedup - Tamaño de archivo

■ Especie

Realizando las pruebas con los archivos del Cuadro 3, podemos observar en la Figura 20 que existen variaciones entre los speedup de las especies, pero estas variaciones reflejan el cambio de las variables: tamaño de archivo y longitud de Read en los archivos fastq. Las redundancias que pueden existir en cada especie no se manifiestan directamente en el speedup, estas se pueden ver en los archivos resultantes; dado a que dependiendo de las redundancias, los Reads se organizan de forma diferente en los Bins en cada especie.

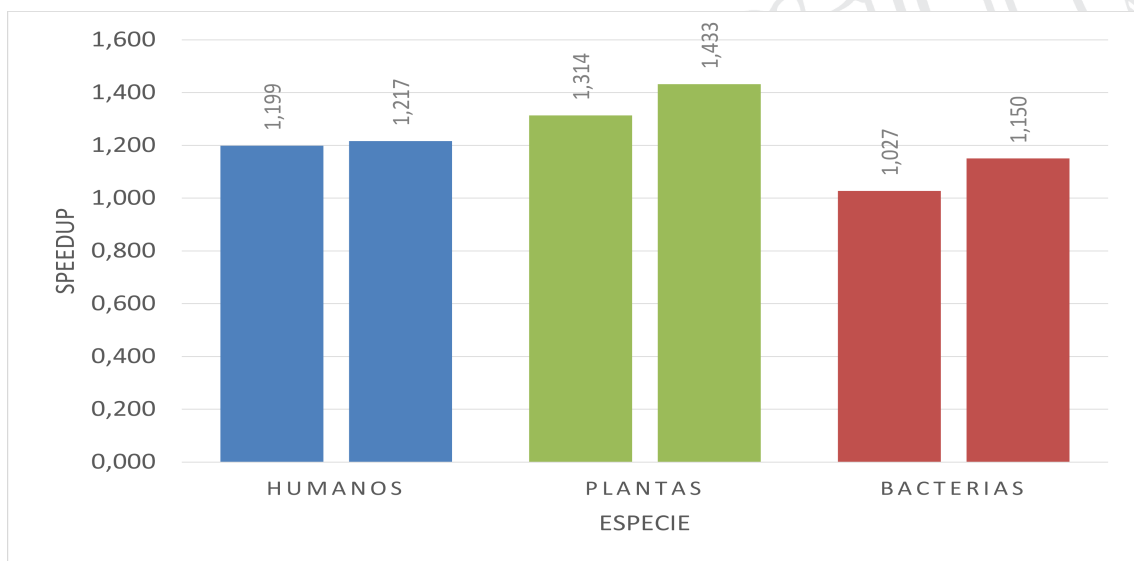


Figura 20: Speedup - Especie

■ Longitud de Read

Tomando los archivos de prueba del Cuadro 4 y observando la Figura 21 se deduce que la longitud del Read es una característica importante en el tiempo de ejecución del programa. Observamos que el archivo con longitud de Read igual a 152 tiene un speedup mas bajo comparado con los otros archivos, los cuales tienen una longitud de Read mas pequeña. También vemos que el tamaño del archivo es un factor que se refleja en la Figura 21 ya que el archivo con el speedup mayor es el archivo mas grande pero no el que tiene una longitud de Read menor.

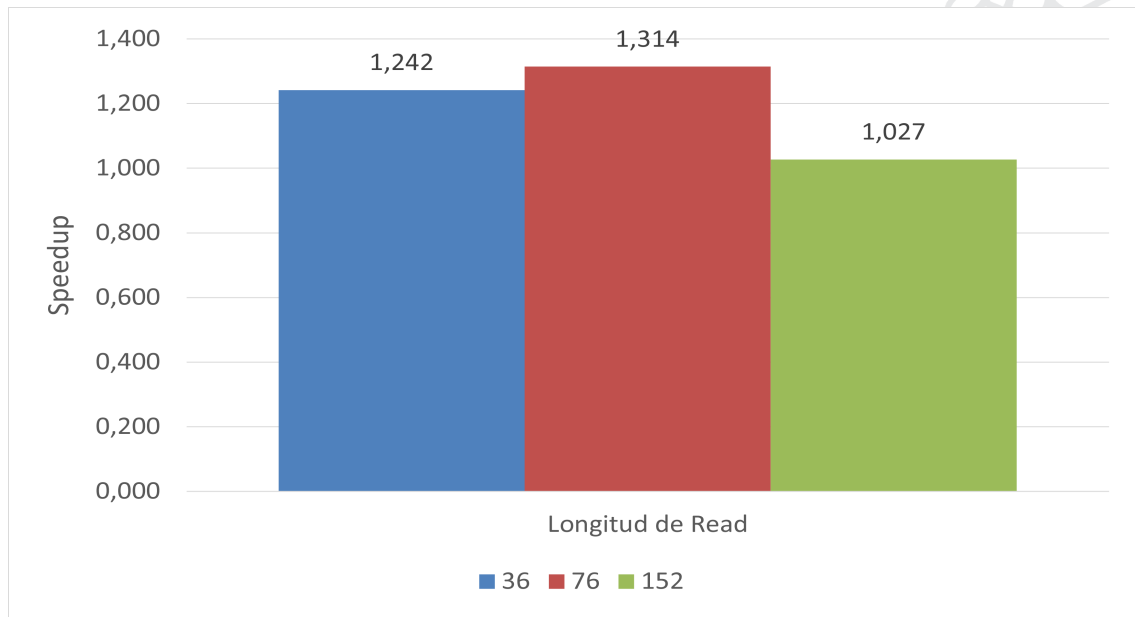


Figura 21: Speedup - Longitud de Read

Comparando los resultados y los análisis de la prueba 2, se concluye que las características analizadas de los archivos fastq mas influyentes en el tiempo de ejecución del programa en su respectivo orden de importancia son: el tamaño del archivo, la longitud de Read y la especie, esta sin tener ningún efecto en el programa.

6. Conclusiones

Durante este proyecto se trabajó en el diseño y la implementación de un algoritmo paralelo para la compresión de datos genómicos en una GPU. Dicho código toma como referencia el algoritmo ORCOM, del cual se paralelizan en la GPU varios módulos que hacen un fuerte trabajo en la CPU y se acelera la etapa de Binning del algoritmo de referencia, haciendo que el tiempo de ejecución disminuya considerablemente dependiendo de algunas características de los archivos fastq que contienen los datos genómicos.

En general el reto del proyecto fue transmitir grandes cantidades de datos al Device desde el Host, ya que estos datos son del orden de los GBytes tardamos

mas en guardarlos en la memoria de la GPU que en ejecutar el kernel del programa. La solución para este problema fue usar los Streams proporcionados por CUDA, ya que haciendo un pipeline con el envío de datos y la ejecución del kernel el speedup mejora considerablemente, esta técnica se conoce como double-buffering. El speedup varía y mejora proporcionalmente con el tamaño del archivo, puesto que al ejecutar la función "DistributeToBins" se acelera un poco el tiempo de ejecución, y al tener un archivo cada vez mas grande se ejecuta esta función un número mayor de veces y el speedup aumenta por cada ejecución de esta función. Para archivos pequeños del orden de los MBytes el speedup disminuye, ya que la GPU tarda de 5 a 10 segundos en reconocer la API de CUDA; para archivos del orden de los GBytes este tiempo es despreciable.

La otra variable que mejora el speedup del programa es la longitud de los Reads. Como en el programa se lee el archivo fastq cada 256 MB, si tomamos una longitud de Read mas pequeño significa tomar mayor cantidad de Reads y a su vez mayor paralelismo, y como el Read es pequeño el kernel tarda menos en ejecutarse. Esto significa que la GPU se aprovecha al máximo por tener mas hilos trabajando concurrentemente y un kernel con menor tiempo de ejecución. Lo contrario ocurre con un archivo de Reads con longitud mayor, donde hay menos trabajo paralelo en la GPU y el kernel tarda mas tiempo en ejecutarse. La variable "especie" en los archivos no determina ningún cambio llamativo en el tiempo de ejecución, dado a que el programa solo procesa cadenas de caracteres y no importa en que orden estén estas cadenas. Las redundancias de la especie de los archivos determinan el orden de los Reads en los Bins.

Aunque el desempeño logrado con la GPU varía según las características de los archivos fastq, se puede decir que se logró el objetivo del proyecto y se aceleró de forma eficiente el proceso de Binning en ORCOM. Esto demuestra que la GPU es una herramienta eficaz a la hora de acelerar programas quitando carga computacional a la CPU, todo gracias a su gran capacidad de paralelismo. También cabe agregar que las GPU son asequibles, puesto que se pueden encontrar en cualquier computador portátil o de escritorio; y económicas en comparación con un sistema multiprocesador de propósito general que se requeriría para lograr speedups similares a los obtenidos.

Referencias

- [1] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, Agnieszka Debudaj-Grabysz, «KMC 2: Fast and resource-frugal k-mer counting», *Bioinformatics*, 31(10), 2015, 1569–1576, doi: 10.1093/bioinformatics/btv022.
- [2] Szymon Grabowski, Sebastian Deorowicz and Łukasz Roguski, «Disk-based compression of data from genome sequencing», *Bioinformatics*, 31(9), 2015, 1389–1395, doi: 10.1093/bioinformatics/btu844.

- [3] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz and Szymon Grabowski, «*Disk-based k-mer counting on a PC*», *Bioinformatics*, 2013, doi: 10.1186/1471-2105-14-160.
- [4] Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount and James A. Yorke, «*Reducing storage requirements for biological sequence comparison*», *Bioinformatics*, 20(18), 2004, 1389–1395, doi: 10.1093/bioinformatics/bth408.
- [5] NVIDIA, «*CUDA C PROGRAMMING GUIDE*», PG-02829-001_v7.5, Sept 2015.
- [6] Ibrahim Numanagić, James K. Bonfield, Faraz Hach, Jan Voges, Jörn Ostermann, Claudio Alberti⁵, Marco Mattavelli, S Cenk Sahinalp, «*Comparison of high-throughput sequencing data compression tools*», *Nature Methods*, 13, 2016, 1005–1008, doi: 10.1038/nmeth.4037.
- [7] Roguski Ł., Deorowicz S., «*DSRC 2—industry-oriented compression of FASTQ files*», *Bioinformatics*, 30, 2014, 2213–2215.
- [8] Jones D. et al, «*Compression of next-generation sequencing reads aided by highly efficient de novo assembly*», *Nucleic Acids Res*, 40, 2012, e171.
- [9] Bonfield J.K., Mahoney M.V., «*Compression of FASTQ and SAM format sequencing data*», *PLoS One*, 8, 2013, e59190.
- [10] Hach F. et al, «*SCALCE: boosting sequence compression algorithms using locally consistent encoding*», *Bioinformatics*, 28, 2012, 3051–3057.
- [11] Selva J.J., Chen X., «*SRComp: Short read sequence compression using burstsort and Elias omega coding*», *PLoS One*, 8, 2013, e81414.
- [12] Yanovsky V., «*Recoil—an algorithm for compression of extremely large datasets of DNA data*», *Algorithms Mol. Biol.*, 6, 2011, 23.
- [13] Cox A.J. et al, «*Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform*», *Bioinformatics*, 28, 2012, 1415–1419.
- [14] https://github.com/josefvargasr/tesis_orcom
- [15] http://ftp.sra.ebi.ac.uk/vol1/fastq/ERR160/ERR160123/ERR160123_1.fastq.gz
- [16] <http://ftp.sra.ebi.ac.uk/vol1/fastq/SRR608/SRR608815/SRR608815.fastq.gz>
- [17] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00099/sequence_read/SRR741411_1.filt.fastq.gz
- [18] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00099/sequence_read/SRR741412_2.filt.fastq.gz
- [19] <ftp://ftp.sra.ebi.ac.uk/vol1/fastq/DRR000/DRR000969/DRR000969.fastq.gz>
- [20] ftp://ftp.sra.ebi.ac.uk/vol1/fastq/DRR000/DRR000966/DRR000966_1.fastq.gz
- [21] ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/DRA000/DRA000971/DRX007651/DRR008465_2.fastq.bz2

Anexos

A. Códigos

A1. Datos constantes mapeados a la GPU

```
__device__ char d_symbolIdxTable[128];
__device__ uint32 d_nBinValue;
__device__ uint32 d_maxLongMinimValue;
__device__ uint32 d_maxShortMinimValue;
__device__ uint8 d_params_signatureLen;
__device__ uint8 d_params_skipZoneLen;

// Alojjar variables globales en GPU
uint32* dd_maxLongMinimValue;
HANDLE_ERROR(cudaGetSymbolAddress((void**)&dd_maxLongMinimValue,
d_maxLongMinimValue));
HANDLE_ERROR(cudaMemcpy(dd_maxLongMinimValue, &maxLongMinimValue,
sizeof(uint32), cudaMemcpyHostToDevice));
uint32* dd_maxShortMinimValue;
HANDLE_ERROR(cudaGetSymbolAddress((void**)&dd_maxShortMinimValue,
d_maxShortMinimValue));
HANDLE_ERROR(cudaMemcpy(dd_maxShortMinimValue, &maxShortMinimValue,
sizeof(uint32), cudaMemcpyHostToDevice));
uint8* dd_params_signatureLen;
HANDLE_ERROR(cudaGetSymbolAddress((void**)&dd_params_signatureLen,
d_params_signatureLen));
HANDLE_ERROR(cudaMemcpy(dd_params_signatureLen, &params.signatureLen,
sizeof(uint8), cudaMemcpyHostToDevice));
uint8* dd_params_skipZoneLen;
HANDLE_ERROR(cudaGetSymbolAddress((void**)&dd_params_skipZoneLen,
d_params_skipZoneLen));
HANDLE_ERROR(cudaMemcpy(dd_params_skipZoneLen, &params.skipZoneLen,
sizeof(uint8), cudaMemcpyHostToDevice));
char* dd_symbolIdxTable;
HANDLE_ERROR(cudaGetSymbolAddress((void**)&dd_symbolIdxTable,
d_symbolIdxTable));
HANDLE_ERROR(cudaMemcpy(dd_symbolIdxTable, symbolIdxTable, 128 *
sizeof(char), cudaMemcpyHostToDevice));
uint32* dd_nBinValue;
HANDLE_ERROR(cudaGetSymbolAddress((void**)&dd_nBinValue, d_nBinValue));
HANDLE_ERROR(cudaMemcpy(dd_nBinValue, &nBinValue, sizeof(uint32),
cudaMemcpyHostToDevice));
```

A2. Funciones en la memoria de la GPU

```
__device__ bool d_IsMinimizerValid(uint32_t minim_, uint32_t mLen_)
{
    if(minim_ == 0)
        minim_ = 0;

    const uint32_t excludeLen = 3;
    const uint32_t excludeMask = 0x3F;
```

```

const uint32_t symbolExcludeTable[] = {0x00, 0x15, 0x2A, 0x3F};

minim_ &= (1 << (2 * mLen_)) - 1;
bool hasInvalidSeq = false;
for (uint32_t i = 0; !hasInvalidSeq && i <= (mLen_ - excludeLen); ++i){
    uint32_t x = minim_ & excludeMask;

    for(uint32_t j = 0; j < 4; ++j){
        hasInvalidSeq |= (x == symbolExcludeTable[j]);
    }

    minim_ >>= 2;
}

return !hasInvalidSeq;
}

__device__ uint32 d_ComputeMinimizer(const char* dna_, uint32_t mLen_)
{
    uint32_t r = 0;

    for(uint32_t i = 0; i < mLen_; ++i){
        if(dna_[i] == 'N')
            return d_nBinValue;

        r <= 2;
        r |= d_symbolIdxTable[(uint32_t)dna_[i]];
    }

    return r;
}

__device__ uint32 d_FindMinimizer(char * d_dna_, uint16 d_len_)
{
    uint32 minimizer = d_maxLongMinimValue;
    const int32 ibeg = 0;
    const int32 iend = d_len_ - d_params_signatureLen + 1 - d_params_skipZoneLen;

    for (int32 i = ibeg; i < iend; ++i){
        uint32 m = d_ComputeMinimizer(d_dna_ + i, d_params_signatureLen);
        if (m < minimizer && d_IsMinimizerValid(m, d_params_signatureLen))
            minimizer = m;
    }

    if (minimizer >= d_maxLongMinimValue)
        return d_nBinValue;

    return minimizer & (d_maxShortMinimValue - 1);
}

__device__ char* d_ComputeRC(char* d_reads_, char* d_readsRC_, uint16 d_len_)
{
    const char rcCodes[24] = {-1, 'T', -1, 'G', -1, -1, -1, 'C',
                              -1, -1, -1, -1, -1, -1, 'N', -1,
                              -1, -1, -1, -1, 'A', -1, -1, -1,
                              };
};

```

```

        for(uint32 i = 0; i < d_len_; ++i){
            d_readsRC_[d_len_-1-i] = rcCodes[(int32)d_reads_[i] - 64];
        }

        return d_readsRC_;
    }

```

A3. Kernel

```

__global__ void d_DistributeToBins(uint32 n_reads_, uint32* d_arr_minim_,
uint16* d_lenReads_, uint32* d_posReads_, char* d_reads, char* d_readsRC,
uint32 offset, uint32 sizeOffset2)
{
    uint32 tid = offset + threadIdx.x + (blockDim.x * blockIdx.x);
    uint32 tid2 = threadIdx.x + (blockDim.x * blockIdx.x);
    if(tid < n_reads_ && tid2 < sizeOffset2 )
    {
        d_arr_minim_[tid] = d_FindMinimizer(&d_reads[d_posReads_[tid]],
d_lenReads_[tid]);
        d_arr_minim_[tid + n_reads_] = d_FindMinimizer(
d_ComputeRC(&d_reads[d_posReads_[tid]],
&d_readsRC[d_posReads_[tid]], d_lenReads_[tid]), d_lenReads_[tid]);
    }
}

```

A4. Lanzamiento de Streams

```

const int n = (n_reads) + (N_THREADS-1);
const int nStreams = numStream;
const int streamSize = (n)/nStreams ;

// Crear Streams
cudaStream_t stream[numStream];
for (uint32 i = 0; i < nStreams; ++i)
    cudaStreamCreate(&stream[i]);

// Copiar Datos a la GPU y Lanzar Kernel por cada Stream
for (uint32 i = 0; i < nStreams; ++i)
{
    uint32 offset1 = i * (h_dnaSize/nStreams);
    uint32 offset2 = i * (recordsCount_/nStreams);
    uint32 sizeOffset1;
    uint32 sizeOffset2;
    uint32 bSize = (((n)/N_THREADS)/nStreams) + 1;

    if(i == nStreams - 1){
        sizeOffset1 = h_dnaSize/nStreams;
        sizeOffset1 = sizeOffset1 + (h_dnaSize -
(sizeOffset1 * nStreams));
        sizeOffset2 = recordsCount_/nStreams;
        sizeOffset2 = sizeOffset2 + (recordsCount_ -
(sizeOffset2 * nStreams));
    }else{
        sizeOffset1 = h_dnaSize/nStreams;
        sizeOffset2 = recordsCount_/nStreams;
    }

    HANDLE_ERROR(cudaMemcpyAsync(&d_reads[offset1], &h_reads[offset1],

```



```

        sizeOffset1 * sizeof(char), cudaMemcpyHostToDevice, stream[i]));
HANDLE_ERROR(cudaMemcpyAsync(&d_lenReads[offset2], &h_lenReads[offset2],
        sizeOffset2 * sizeof(uint16), cudaMemcpyHostToDevice, stream[i]));
HANDLE_ERROR(cudaMemcpyAsync(&d_posReads[offset2], &h_posReads[offset2],
        sizeOffset2 * sizeof(uint32), cudaMemcpyHostToDevice, stream[i]));
d_DistributeToBins<<< bSize, N_THREADS, 0, stream[i]>>>(n_reads,
        d_arr_minim, d_lenReads, d_posReads, d_reads, d_readsRC,
        offset2, sizeOffset2);
}

for (uint32 i = 0; i < nStreams; ++i)
    HANDLE_ERROR(cudaStreamSynchronize(stream[i]));

// Copiar Datos a la CPU
for (uint32 i = 0; i < nStreams; ++i)
{
    uint32 sizeOffset2;
    uint32 offset2 = i * 2 * (recordsCount_/nStreams);

    if(i == nStreams - 1){
        sizeOffset2 = recordsCount_/nStreams;
        sizeOffset2 = sizeOffset2 + (recordsCount_ -
            (sizeOffset2 * nStreams));
    }else{
        sizeOffset2 = recordsCount_/nStreams;
    }

    HANDLE_ERROR(cudaMemcpyAsync(&h_arr_minim[offset2], &d_arr_minim[offset2],
        2 * sizeOffset2 * sizeof(uint32), cudaMemcpyDeviceToHost, stream[i]));
}

// Destruir Streams
for (uint32 i = 0; i < nStreams; ++i)
    cudaStreamDestroy(stream[i]);

```

B. Tablas de tiempos de ejecución - Prueba 2

Tamaño		Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4
1.6 GB	CPU	26,94	26,92	26,88	26,62
	GPU	26,61	27,64	27,57	27,37
8.2 GB	CPU	136,76	136,53	136,76	136,88
	GPU	113,70	119,43	113,29	123,00
17 GB	CPU	291,92	291,28	292,24	276,50
	GPU	238,25	234,16	237,28	237,16
20 GB	CPU	275,88	266,08	266,52	259,16
	GPU	199,36	201,20	210,41	201,91
33 GB	CPU	577,12	581,57	589,45	580,43
	GPU	491,19	510,05	510,88	512,69
56 GB	CPU	845,21	854,99	858,08	788,60
	GPU	605,37	583,55	567,87	580,85

Cuadro 7: Tiempo de ejecución - Tamaño de archivo (seg).

Tamaño		Mínimo	Media	Desv Estándar
1.6 GB	CPU	26,62	26,84	00,15
	GPU	26,61	27,30	00,47
	SPEEDUP	1,000	0,983	
8.2 GB	CPU	136,53	136,73	00,15
	GPU	113,29	117,35	04,69
	SPEEDUP	1,205	1,167	
17 GB	CPU	276,50	287,98	07,67
	GPU	234,16	236,71	01,77
	SPEEDUP	1,181	1,217	
20 GB	CPU	259,16	266,91	06,86
	GPU	199,36	203,22	04,91
	SPEEDUP	1,300	1,314	
33 GB	CPU	577,12	582,14	05,22
	GPU	491,19	506,20	10,07
	SPEEDUP	1,175	1,150	
56 GB	CPU	788,60	836,72	32,55
	GPU	567,87	584,41	15,56
	SPEEDUP	1,389	1,433	

Cuadro 8: Promedios de tiempo de ejecución - Tamaño de archivo (seg).

Especie		Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4
Humano	CPU	236,15	235,97	235,36	235,84
	GPU	198,55	195,71	193,19	199,60
Humano	CPU	291,92	291,28	292,24	276,50
	GPU	238,25	234,16	237,28	237,16
Planta	CPU	275,88	266,08	266,52	259,16
	GPU	199,36	201,20	210,41	201,91
Planta	CPU	845,21	854,99	858,08	788,60
	GPU	605,37	583,55	567,87	580,85
Bacteria	CPU	247,41	246,49	245,67	247,97
	GPU	243,79	242,29	238,84	236,63
Bacteria	CPU	577,12	581,57	589,45	580,43
	GPU	491,19	510,05	510,88	512,69

Cuadro 9: Tiempo de ejecución - Especie (seg).

Especie		Mínimo	Media	Desv Estándar
Humano	CPU	235,36	235,83	00,34
	GPU	193,19	196,76	02,89
	SPEEDUP	1,218	1,199	
Humano	CPU	276,50	287,98	07,67
	GPU	234,16	236,71	01,77
	SPEEDUP	1,181	1,217	
Planta	CPU	259,16	266,91	06,86
	GPU	199,36	203,22	04,91
	SPEEDUP	1,300	1,314	
Planta	CPU	788,60	836,72	32,55
	GPU	567,87	584,41	15,56
	SPEEDUP	1,389	1,433	
Bacteria	CPU	245,67	246,89	01,01
	GPU	236,63	240,39	03,25
	SPEEDUP	1,038	1,027	
Bacteria	CPU	577,12	582,14	05,22
	GPU	491,19	506,20	10,07
	SPEEDUP	1,175	1,150	

Cuadro 10: Promedios de tiempo de ejecución - Especie (seg).

Long Read		Ejecución 1	Ejecución 2	Ejecución 3	Ejecución 4
36	CPU	226,23	225,51	227,09	228,84
	GPU	185,33	178,97	174,16	193,47
76	CPU	275,88	266,08	266,52	269,16
	GPU	199,36	201,20	210,41	201,91
152	CPU	247,41	246,49	245,67	247,97
	GPU	243,79	242,29	238,84	236,63

Cuadro 11: Tiempo de ejecución - Longitud de Read (seg).

Long Read		Mínimo	Media	Desv Estándar
36	CPU	225,51	226,92	01,44
	GPU	174,16	182,98	08,36
	SPEEDUP	1,295	1,242	
76	CPU	259,16	266,91	06,86
	GPU	199,36	203,22	04,91
	SPEEDUP	1,300	1,314	
152	CPU	245,67	246,89	01,01
	GPU	236,63	240,39	03,25
	SPEEDUP	1,038	1,027	

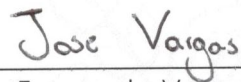
Cuadro 12: Promedios de tiempo de ejecución - Longitud de Read (seg).

Revisión

Se hace constar que el informe final del proyecto de investigación **“PARALELIZACIÓN DE ALGORITMOS DE COMPRESIÓN DE SECUENCIAS GENÓMICAS UTILIZANDO UNIDADES DE PROCESAMIENTO GRÁFICO GPU”** vinculado al grupo de investigación SISTEMIC de la Universidad de Antioquia, está completo y revisado por el Prof. Sebastián Isaza Ramírez, asesor interno.



Prof. Sebastian Isaza Ramirez
C.C.: 98669951
Asesor Interno



Jose Fernando Vargas Rodriguez
C.C.: 1017207056
Estudiante de Ingeniería Electrónica

