UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE
INGENIERÍA

Computación de alto desempeño aplicada al procesamiento de grandes volúmenes de datos genómicos

Informe de proyecto de grado presentado por

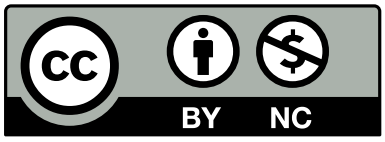
Agustín Núñez, Ignacio Fourcade, Paula Abbona

bajo la supervisión de **Martin Pedemonte** y **Ernesto Dufrechou**, en cumplimiento parcial de los requerimientos para la graduación de la carrera de

Ingeniería en Computación

de *Facultad de Ingeniería de la Universidad de la República.*

Montevideo, 22 de diciembre de 2025



Agradecimientos

Queremos expresar nuestro más sincero agradecimiento a todas las personas que, de una manera u otra, contribuyeron al logro de este proyecto. En especial, queremos agradecer a nuestros tutores de tesis, Ernesto y Martín, quienes nos guiaron y acompañaron en cada etapa de este proceso. Extendemos también nuestro agradecimiento a nuestras familias y amigos, por su apoyo incondicional, paciencia y aliento constante durante todo el recorrido académico. Finalmente, queremos reconocer a Cluster UY (Nesmachnow & Iturriaga, 2019), cuya infraestructura computacional fue fundamental para la realización de los experimentos y el desarrollo de esta investigación. Sin este soporte, gran parte del trabajo habría permanecido en el plano hipotético, donde muchas de las validaciones empíricas, los estudios de escalabilidad y las comparaciones exhaustivas entre implementaciones hubieran sido impracticables.

Resumen

El cómputo de distancias entre vectores es un problema relevante en el ámbito de la bioinformática, utilizado principalmente en diversas etapas en el análisis de datos genómicos. A pesar de que existen implementaciones de alto desempeño, muchas de ellas no explotan características particulares del problema biológico, y, por ello, no aprovechan plenamente los recursos de cómputo al trabajar con conjuntos de datos a gran escala. En este trabajo, diseñamos, implementamos y evaluamos diferentes algoritmos que aprovechan propiedades específicas de los datos genómicos en plataformas CPU y GPU, obteniendo mejoras significativas tanto en el uso de memoria como en la eficiencia computacional respecto a los métodos de referencia del estado del arte. En particular, proponemos un esquema de codificación de los datos que reduce sustancialmente el uso de memoria y permite explotar de manera efectiva la jerarquía de memoria de los recursos de cómputo. A partir de este esquema derivamos un método eficiente para la comparación de secuencias de ADN, capaz de reproducir el resultado equivalente al de la distancia euclidiana, utilizando únicamente una cantidad mínima de operaciones bit a bit. Asimismo, se realizaron optimizaciones sobre el algoritmo R-Kleene que explotan la simetría del problema. Por otra parte, los algoritmos desarrollados pueden adaptarse fácilmente a arquitecturas híbridas y al procesamiento por lotes, gracias a la descomposición natural del problema. Finalmente, realizamos una evaluación experimental para analizar el desempeño computacional de las propuestas, obteniendo mejoras significativas sobre los algoritmos utilizados como línea base.

Keywords: *Bioinformatics, High-Performance Computing, GPU Acceleration, Tensor Cores, Distance Matrix, All-Pairs Shortest Paths, Parallel Algorithms*

Índice General

Capítulo 1

Introducción

Los avances en biología molecular y técnicas de secuenciación masiva han generado cantidades enormes de datos genéticos, cuya interpretación requiere cada vez más del apoyo de herramientas matemáticas y computacionales. La información hereditaria de la mayoría de los seres vivos está contenida en el ácido desoxirribonucleico (ADN) (The 1000 Genomes Project Consortium, 2015), cuya secuencia puede ser leída y analizada a gran escala gracias a estas tecnologías. En particular, el análisis de datos genómicos de poblaciones humanas permite estudiar la diversidad genética y entender relaciones entre grupos a partir del ADN. Sin embargo, el volumen y la complejidad de estos datos presentan grandes desafíos computacionales, especialmente cuando se busca analizar similitudes entre miles de individuos, cada uno representado por millones de posiciones del genoma.

En muchas aplicaciones, resulta útil calcular **distancias entre individuos** a partir de su ADN, ya que parientes cercanos, tendrán más coincidencias entre sí, e individuos más lejanos, menos. En particular, permite detectar estructuras poblacionales debido a aislamiento y pulsos de migraciones (Novembre & Stephens, 2008). Una aplicación posible consiste en construir un grafo que represente las relaciones genéticas entre un conjunto de individuos tras calcular dichas distancias.

Para abordar este trabajo, resulta fundamental comprender cómo se representa esa información y qué propiedades permite analizar. El ADN codifica muchas de las características de los seres vivos. En humanos, ejemplos típicos son la altura, el color de los ojos; en cultivos puede ser la cantidad de toneladas por hectárea, la calidad del grano, o en animales, la cantidad de leche producida o la calidad de carne de una vaca, entre muchas otras. El código del ADN se compone de 4 letras, A (Adenina), C (Citosina), G (Guanina), T (Timina) que se van ordenando de una manera específica.

En general, dado que las mutaciones ocurren por azar y es poco probable que haya dos mutaciones en un mismo sitio debido a las bajas tasas de mutación y largo de los genomas (Wikipedia contributors, 2025), cuando miramos un sitio específico de la cadena de ADN que contiene mutaciones, se encuentran solamente dos variantes, por ejemplo, algunos individuos van a tener una A y otros una G. Estas variaciones específicas, conocidas como polimorfismos de un solo nucleótido o SNPs *(Single Nucleotide Polymorphisms)*, son las formas más comunes de variación genética entre los organismos, permitiendo representar una secuencia de ADN como una tira de 0s y 1s, donde un 0 representa la mutación original (A) y un 1 la otra variante (en este caso G). La Figura 1, ilustra este concepto mostrando una secuencia de ADN en la que una base (por ejemplo, A) puede diferir en algunos individuos, siendo sustituida por otra (por ejemplo, G).

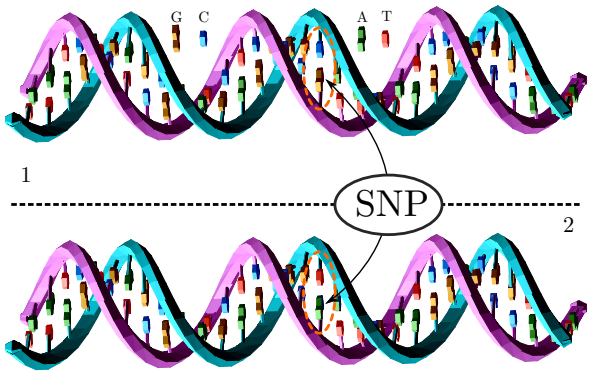


Figura 1: Representación gráfica de un SNP, donde la cadena 1 difiere de la cadena 2 en un solo par de bases. Extraída de Wikipedia contributors (2025).

En organismos diploides, como los humanos, cada individuo posee dos copias de cada cromosoma, una heredada del padre y otra de la madre, lo que significa que tiene dos alelos[1] por posición en el ADN. En el contexto de SNPs, estas combinaciones se pueden representar mediante tiras de 0s, 1s y 2s, representando así al individuo, que resulta de sumar la contribución de ambos alelos en cada posición del ADN. Un valor de 0 indica que ambos alelos son iguales al original, lo que implica la ausencia de un SNP; un valor de 1 señala que uno de los alelos es una variante, mientras que el otro es el original; y un valor de 2 refleja que ambos alelos corresponden a la variante. Esta representación simplificada permite reducir la cantidad de información sin perder las diferencias genéticas relevantes para el cálculo de distancias, optimizando así el análisis y comparación de las secuencias.

Los datos genómicos se suelen organizar en matrices de gran tamaño, donde cada fila representa un individuo y cada columna corresponde a un SNP. En este contexto, el número de columnas —es decir, la cantidad de SNPs considerados— puede llegar a millones, o incluso acercarse a los tres mil millones, que es la cantidad total de pares de bases en el genoma humano, mientras que la cantidad de individuos puede ir desde unos cientos a millones. El tamaño y cantidad de tiras involucradas hacen que calcular las distancias entre un grupo de individuos sea un desafío desde el punto de vista computacional y amerite la aplicación de técnicas de cómputo de alto desempeño.

El procesamiento de una instancia del problema involucra diversos algoritmos intensivos en datos. Sin embargo, no todos estos representan el mismo nivel de desafío desde el punto de vista computacional.

En particular, el cálculo de la matriz de distancias entre individuos representa el principal cuello de botella, debido a la gran cantidad de datos que deben procesarse. La matriz de entrada tiene dimensión `cantidad_de_individuos` $\times$ `cantidad_de_SNPs`. En cambio, en etapas posteriores como la construcción del grafo, se trabaja con matrices de tamaño `cantidad_de_individuos` $\times$ `cantidad_de_individuos`, por lo que, cuando `cantidad_de_individuos` $\ll$ `cantidad_de_SNPs`, este paso es más liviano computacionalmente, aun cuando pueda requerir operaciones costosas.

[1] Un alelo es una variante de un mismo gen que surge por diferencias en la secuencia del ADN.

Volviendo al paso crítico, el cálculo de distancias puede hacerse utilizando diversas métricas, como la Euclideana, Manhattan, Mahalanobis o Minkowski. La complejidad temporal total de esta operación entre n individuos es del orden de $\Theta(C_d n^2)$, siendo C_d el orden de calcular la distancia entre dos elementos con la métrica utilizada. Aprovechando la simetría de la relación de distancia, el número total de comparaciones se reduce a $\frac{n(n-1)}{2}$. Por ejemplo, si consideramos 600.000.000 SNPs y 30.000 individuos, nos da un total de **269.991.000.000.000.000** (269.991 cuatrillones) comparaciones entre SNPs, es decir, si hipotéticamente cada persona en la Tierra realizara una comparación por segundo, completar estas operaciones requeriría del orden de 3.37x 10^7 segundos, es decir, aproximadamente 390 días (1.07 años). Esta magnitud hace que el procesamiento sea extremadamente intensivo, tanto en cómputo como en memoria. En procesadores de propósito general, esta tarea puede tomar varias horas, e incluso días, dependiendo del tamaño de la instancia. Por ello, resulta esencial recurrir a estrategias de cómputo paralelo y hardware especializado.

La tendencia reciente para resolver este problema se ha trasladado al ámbito de las unidades de procesamiento gráfico (GPU). Las GPUs no solo se están popularizando entre las comunidades científicas en diferentes campos de investigación, sino que también se instalan comúnmente en los ordenadores domésticos, estaciones de trabajo, consolas y dispositivos de juego actuales. Actualmente, la computación de propósito general en GPU (GPGPU) ofrece una solución importante a muchos problemas complejos. Sin embargo, la limitación de la memoria del dispositivo en la GPU plantea un nuevo problema de escalabilidad. Por lo tanto, aunque algunos métodos son mucho más rápidos que la versión secuencial, en el mundo real no son ampliamente viables.

Este proyecto de grado se enmarca en la línea de investigación "*Poblaciones y comunidades: Genómica y Evolución*" del Centro Interdisciplinario en Ciencia de Datos y Aprendizaje Automático (CICADA). Dentro de esta línea se desarrolla la tesis de maestría en Ingeniería Matemática de la estudiante Micaela Long, dirigida por María Inés Fariello y Lucía Spangenberg y titulada "*Desarrollo de herramientas de aprendizaje estadístico para el análisis de datos genómicos de la población uruguaya*" (Beca ANII POS_NAC_2024_1_182936). Este trabajo aplica técnicas matemáticas y computacionales al estudio de datos genómicos, con especial énfasis en el análisis de la estructura genética de la población uruguaya. En particular, se emplean métodos de reducción de dimensionalidad basados en grafos, los cuales requieren de la construcción de matrices de adyacencia a partir de medidas de similitud entre individuos.

En este contexto, nuestro proyecto busca desarrollar una solución computacional eficiente para el cálculo de distancias genéticas entre individuos, empleando plataformas de cómputo paralelo. Para ello es necesario revisar y optimizar la representación de los datos en memoria, así como adaptar y extender los algoritmos existentes para aprovechar de manera efectiva la arquitectura de estas plataformas.

Específicamente, el proyecto propone optimizar y escalar estos algoritmos mediante el uso combinado de unidades de procesamiento gráfico (GPU) y unidades de procesamiento central (CPU), junto con una representación compacta de los datos que permita procesar conjuntos genómicos de gran tamaño de manera eficiente.

El documento se estructura de la siguiente manera. En el Capítulo 2 se presentan los fundamentos teóricos y conceptuales que contextualizan el trabajo, incluyendo la descripción del problema, los conceptos de cálculo de distancias y caminos mínimos entre pares de vértices, las arquitecturas de procesamiento utilizadas (CPU y GPU), así como una revisión del trabajo relacionado y de las principales bibliotecas y algoritmos clásicos. En el Capítulo 3 se define la línea base, donde se describe la implementación de referencia actualmente utilizada para el procesamiento de datos genómicos. El Capítulo 4 introduce las propuestas de resolución desarrolladas, abarcando las implementaciones tanto en CPU como en GPU. En el Capítulo 5 se presentan los resultados de la evaluación experimental, junto con el análisis de las principales métricas de desempeño obtenidas. Finalmente, en el Capítulo 6 se exponen las conclusiones del trabajo y se delinean las posibles líneas de trabajo a futuro.

Capítulo 2

Contexto y Fundamento Teórico

En este capítulo se presentan los fundamentos teóricos y conceptuales necesarios para contextualizar el enfoque de este trabajo. En primer lugar, en la Sección 2.1 se describe el problema, se detalla el enfoque actual de la investigación y se desarrollan los conceptos necesarios relacionados con el cálculo de distancias y caminos mínimos entre todos los pares de vértices. En la Sección 2.2 se describen las arquitecturas de procesamiento sobre las que se implementa la propuesta: primero las CPUs, y luego las GPUs y el modelo de programación CUDA, destacando los aspectos más relevantes para la implementación paralela de los algoritmos tratados en este proyecto. Finalmente, la Sección 2.3 se enfoca en el trabajo relacionado y las soluciones existentes que abordan los problemas centrales de este estudio. Abarca una reformulación algebraica integral del problema, que incluye tanto la construcción del grafo como el cálculo de la matriz de distancias. Además, se revisan las principales bibliotecas de cómputo de alto desempeño (HPC) relevantes para estas operaciones y se realiza un recorrido por los algoritmos clásicos para la resolución del problema de caminos mínimos entre todo par de nodos, deteniéndose en particular en la solución basada en la clausura de Kleene mediante el álgebra min-plus.

2.1. Descripción del Problema

El objetivo general que se busca es conseguir una representación de los datos genómicos que refleje, de la manera más fiel posible, la similitud genética entre individuos. Esto implica que, los individuos genéticamente próximos aparezcan cercanos, y que los agrupamientos resultantes tengan sentido desde la perspectiva biológica, mostrando gradientes, mezclas o subestructuras poblacionales (Long, 2024).

En términos prácticos esto se traduce en dos tareas: (i) medir la similitud (o distancia) entre individuos a partir de sus perfiles genómicos; y (ii) construir una estructura (por ejemplo un grafo) que use esas medidas para obtener visualizaciones y agrupamientos útiles para el análisis biológico.

Ahora bien, lo anterior no es una tarea sencilla. Los datos genómicos presentan alta dimensionalidad y distribuciones de densidad no uniformes, de modo que algunas regiones del espacio (subpoblaciones) concentran gran cantidad de puntos y otras muy baja densidad. Además, los intereses biológicos requieren preservar tanto la estructura local (vecindades) como la global (distancias relativas entre grupos).

Aunque métodos clásicos como el Análisis de Componentes Principales (PCA) permiten una reducción lineal de dimensionalidad útil para observar tendencias globales, estos tienden a diluir la información local. Por otro lado, técnicas no lineales como t-SNE o UMAP resaltan la

estructura local pero a menudo distorsionan la organización global y hacen difícil interpretar distancias entre *clusters* en la visualización (Long, 2024).

El enfoque desarrollado por docentes del IMERL, que constituye el punto de partida de este proyecto, plantea en primer lugar, a partir de un conjunto X de individuos, calcular una distancia $d(x_i, x_j)$ para cada pareja de individuos, formando la *matriz de distancias* $\mathcal{D} \in \mathcal{M}_{n\times n}$

$$\mathcal{D} = \begin{bmatrix} d(x_1,x_1) & d(x_1,x_2) & \dots & d(x_1,x_n) \\ d(x_2,x_1) & d(x_2,x_2) & \dots & d(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ d(x_n,x_1) & d(x_n,x_2) & \dots & d(x_n,x_n) \end{bmatrix}$$

Def. 1: Definición de distancia

Sea X un conjunto, donde cada elemento es un *individuo*, entonces la función distancia sobre X es una función d que se define de la siguiente forma:

$$d : X \times X \to \mathbb{R}_{\geq 0}$$

tal que, para cualquier $x, y \in X$, se satisfacen las siguientes propiedades

$$d(x,y) > 0 \text{ si } x \neq y; d(x,x) = 0$$
$$d(x,y) = d(y,x)$$
$$d(x,y) \leq d(x,z) + d(z,y) \ \forall z \in X$$

Cualquier función con estas propiedades es llamada una ***función de distancia*** *o* ***métrica***

Observar que la matriz $\mathcal{D}$ cumple con las siguientes propiedades:

- Su diagonal es el vector nulo $\text{diag}(\mathcal{D}) = \emptyset$.
- Es una matriz simétrica $\mathcal{D} = \mathcal{D}^T$.

Entre las distintas métricas posibles, el enfoque adopta la distancia euclidiana como base para cálculos posteriores.

Def. 2: Distancia Euclidiana

Sean $x = (x_1, x_2, ..., x_n)$ e $y = (y_1, y_2, ..., y_n)$ dos puntos en $\mathbb{R}^n$. La distancia euclidiana entre x e y se define como:

$$d(x,y) = \sqrt{\sum_{k=1}^{n} (x_k - y_k)^2}$$

Este valor corresponde a la longitud del segmento recto que une ambos puntos en el espacio n-dimensional. El conjunto $(\mathbb{R}^n, d)$ constituye un espacio métrico llamado espacio euclidiano n-dimensional (Kolmogorov & Fomin, 1975).

Esta distancia cumple con las propiedades de la definición 1.

En segundo lugar, se utiliza la matriz $\mathcal{D}$ para la construcción de un grafo partiendo del estimador de la distancia de Fermat (Sapienza et al., 2018):

Def. 3: Estimador de la Distancia de Fermat

Sea $\alpha \geq 1$, un conjunto finito $\mathcal{X} \subset \mathbb{R}^n$, y $p, q \in \mathcal{X}$. El *estimador de la distancia de Fermat* ponderado según α entre p y q se define como:

$$\ell_\alpha(p, q, \mathcal{X}) = \min_{\pi=\{x_i\}_{i=1}^{T}} \sum_{i=1}^{T-1} |x_i - x_{i+1}|^\alpha$$

donde π es un camino de puntos de $\mathcal{X} \cup \{p, q\}$ con $x_1 = p$ y $x_T = q$, y $|\cdot|$ la norma euclidiana.

Notar que para $\alpha = 1$. El estimador coincide con la distancia euclidiana.

Una opción natural y común es definir el peso de la arista entre dos individuos mediante la distancia Euclidiana entre sus vectores de SNPs. Aunque sencilla y directa, esta aproximación puede fallar cuando la distribución de los puntos no es uniforme. En particular, la Euclidiana mide la separación directa entre dos puntos, sin tener en cuenta la densidad del entorno. En presencia de regiones densas conectando dos puntos, una secuencia de saltos cortos a través de la región densa puede ser una mejor aproximación a la cercanía genética que un salto directo atravesando una región de baja densidad. Para verlo con un ejemplo numérico muy simple y con el fin de clarificar el procedimiento descrito, incluimos un ejemplo ilustrativo en el Apéndice A.1.

El grafo de Fermat, a diferencia de un grafo simple, pondera sus aristas por la distancia de Fermat, una métrica especializada que captura la geometría y la distribución de los puntos. Se interpreta como un recorrido de mínima distancia ajustado por la densidad de los datos, que tiende a cuantificar qué tan cerca están dos puntos, favoreciendo caminos que atraviesan regiones con mayor densidad de datos.

Def. 4: Grafo de Fermat

Sea $\alpha \in [1, \infty)$ y un conjunto finito de puntos $\mathcal{X} \subset \mathbb{R}^n$, definimos el *grafo de Fermat ponderado según α* como el grafo ponderado

$$G_\alpha = (V, E, w_\alpha)$$

donde el conjunto de vértices es $V = \mathcal{X}$, el conjunto de aristas es $E = \{\{p, q\} \mid p, q \in \mathcal{X}\}$ y $w_\alpha(p, q) = \ell_\alpha(p, q, \mathcal{X})$.

Notar que G_α es un grafo simple y completo, ya que no presenta lazos ni aristas múltiples, y además todo par de vértices distintos se encuentra conectado por exactamente una arista.

En este contexto, dado un conjunto finito de individuos $\mathcal{X} \subset \mathbb{R}^{N_S}$, donde N_S denota la cantidad de SNPs, se desea encontrar, para cada par de individuos $(p, q) \in \mathcal{X} \times \mathcal{X}$, la *distancia*

mínima desde p hasta q según los pesos w de las aristas, es decir, $\ell_\alpha(p, q)$. De esta manera, el problema puede formularse naturalmente como una instancia del problema clásico de *All-Pairs Shortest Paths* (APSP).

2.2. Computación de Alto Desempeño

Mediante la utilización coordinada de múltiples recursos de cómputo, las técnicas de computación de alto desempeño son capaces de resolver problemas complejos en tiempos de cómputo razonable. Aplicando estrategias de división de datos (descomposición de dominio) o de paralelismo de control (descomposición funcional), las técnicas de computación de alto desempeño permiten alcanzar el poder de cómputo necesario para la resolución eficiente de problemas que involucran complejos modelos matemáticos o grandes volúmenes de datos.

2.2.1. Unidad Central de Procesamiento

Las unidades de procesamiento central (CPUs) constituyen el núcleo tradicional de la computación moderna. Están diseñadas para ejecutar una amplia variedad de instrucciones de manera secuencial, optimizándose para reducir la latencia en la ejecución de hilos individuales y garantizar así un alto rendimiento en aplicaciones de propósito general.

Una característica fundamental de las CPU modernas es su fuerte dependencia de las memorias caché como mecanismo para reducir la latencia de acceso a los datos. La memoria principal (RAM) es varios órdenes de magnitud más lenta que los registros de la CPU; por lo tanto, acceder directamente a ella en cada instrucción resultaría prohibitivo para el rendimiento. Para mitigar este problema, las CPU incorporan múltiples niveles de caché denominados L1, L2 y L3. Una caché es una memoria pequeña y extremadamente rápida que almacena copias de los datos más frecuentemente usados de la memoria principal. Cuando la CPU necesita un dato, primero lo busca en la caché; si está presente (acierto), el acceso es casi instantáneo. Si no está (fallo), se debe acceder a la memoria principal.

El rendimiento de la caché se basa en los siguientes principios:

- Localidad temporal: Si un dato es accedido, es probable que se acceda de nuevo en un futuro cercano.
- Localidad espacial: Si un dato es accedido, es probable que los datos adyacentes en memoria también sean accedidos en un futuro cercano.

Aprovechando estas regularidades en los patrones de acceso, las cachés logran reducir significativamente el tiempo de espera promedio de las operaciones de lectura y escritura de memoria.

2.2.2. Unidad de Procesamiento Gráfico

Las unidades de procesamiento gráfico (GPUs) son dispositivos de hardware diseñados para realizar cálculos en paralelo a gran escala. A diferencia de las CPUs, que se optimizan para operaciones secuenciales y una variedad de tareas generales, las GPUs están construidas con un gran número de núcleos ligeros dedicados exclusivamente al procesamiento simultáneo de múltiples hilos. Esto las hace ideales para aplicaciones que requieren cálculos intensivos, como gráficos, simulaciones científicas, aprendizaje profundo y procesamiento de datos.

En términos de arquitectura, una GPU moderna se compone de cientos a miles de núcleos y una jerarquía de memoria que incluye, por ejemplo, memoria global, memoria compartida, y memoria local para cada hilo.

Este diseño escalable y masivamente paralelo permite a las GPUs sobresalir en tareas donde los datos pueden descomponerse en operaciones independientes. Sin embargo, el rendimiento óptimo depende de una programación cuidadosa que aproveche la jerarquía de memoria y la ejecución paralela.

Uno de los factores mas relevantes para la adopción de la GPU en el contexto del HPC fue el desarrollo de CUDA, por parte del fabricante de GPU NVIDIA.

Arquitectura de Dispositivo Unificado de Computación (CUDA)

CUDA (*Compute Unified Device Architecture*) es una plataforma y modelo de programación desarrollados por NVIDIA que permite a los desarrolladores realizar cálculos paralelos utilizando unidades de procesamiento gráfico (GPUs). Con el soporte para programación en alto nivel, CUDA simplifica el desarrollo de aplicaciones científicas y de ingeniería, haciendo que los cálculos intensivos sean significativamente más rápidos y eficientes. Su popularidad ha consolidado a CUDA como una herramienta ampliamente adoptada y reconocida en el ámbito de la computación de alto rendimiento *(HPC)*.

La programación con CUDA sigue un enfoque heterogéneo, en el que la CPU *(host)* y la GPU *(device)* colaboran para resolver problemas complejos. En este modelo, las tareas secuenciales se ejecutan en la CPU, mientras que las operaciones altamente paralelizables son transferidas a la GPU, maximizando el rendimiento al aprovechar la capacidad masiva de cómputo paralelo de las GPUs.

Estructura de Ejecución y Jerarquía de Memoria en CUDA

En el modelo de programación CUDA, uno de los conceptos fundamentales es el de *kernel*. Un *kernel* es una función que se ejecuta en la GPU. A diferencia de las funciones tradicionales de CPU, los *kernels* están diseñados para ser ejecutados por muchos hilos en paralelo, aprovechando la arquitectura masivamente paralela de las GPUs.

Para facilitar esta ejecución paralela, CUDA organiza los hilos en una jerarquía estructurada que permite escalar el cómputo a distintas arquitecturas y volúmenes de datos. En la base de esta jerarquía se encuentran los *threads* (hilos), que constituyen la unidad de ejecución más pequeña y realizan una parte del cómputo total, ejecutando un *kernel* en paralelo con otros hilos. Varios *threads* se agrupan en *blocks* (bloques), y los hilos dentro de un mismo bloque pueden colaborar entre sí mediante mecanismos como la memoria compartida. Además, los hilos de un bloque se agrupan internamente en unidades llamadas *warps*, cada una compuesta por 32 hilos que se ejecutan en paralelo. Finalmente, los bloques se organizan en una *grid* (grilla), que abarca la ejecución total del *kernel* en la GPU.

En cuanto a la memoria, las GPUs basadas en la arquitectura CUDA disponen de varios tipos, cada una con características específicas de rendimiento y usos recomendados. En el nivel más alto de la jerarquía se encuentra la memoria global. Su alcance y vida útil son globales, lo que significa que es accesible por todos los hilos y su vida útil dura mientras el programa

se ejecuta. Su capacidad es grande, pero presenta alta latencia comparada con otras memorias. Normalmente, el primer paso de un programa en CUDA es transferir los datos necesarios desde el *host* (CPU) a la memoria global de la GPU.

Otro tipo es la memoria de textura, una memoria de solo lectura optimizada para patrones de acceso 2D espacialmente coherentes, como los que se encuentran en el procesamiento de imágenes y texturas. Por su parte, la memoria compartida es una memoria rápida, accesible por todos los hilos dentro de un mismo bloque. Permite que los hilos cooperen y compartan datos eficientemente, reduciendo el número de accesos a la memoria global. Finalmente, existe la memoria constante, una memoria *read-only* con caché, accesible por todos los hilos de la grilla. Es ideal para almacenar datos inmutables que son accedidos frecuentemente por múltiples hilos.

La eficiencia en el uso de la memoria depende, en gran medida, de los patrones de acceso implementados. En el caso de la memoria global, los accesos realizados por un *warp* se agrupan en transacciones de 32 bytes. Cuando los hilos de un *warp* acceden a direcciones contiguas en memoria, las transacciones pueden unirse, lo que minimiza la cantidad de operaciones de memoria necesarias. Por el contrario, accesos dispersos implican múltiples transacciones, reduciendo el *throughput* de instrucciones. En la memoria compartida, los accesos se ven condicionados por su división en bancos. Si los hilos de un *warp* acceden simultáneamente a direcciones en bancos distintos, el acceso se resuelve en un único ciclo. Sin embargo, si varios hilos acceden a posiciones distintas dentro del mismo banco, se generan los denominados conflictos de banco, lo cual introduce esperas adicionales.

La Figura 2 ilustra la jerarquía de ejecución y la Figura 3 los diferentes tipos de memoria disponibles en una GPU con arquitectura CUDA, lo cual resulta útil para comprender cómo se estructuran los hilos y cómo acceden a los datos durante la ejecución.

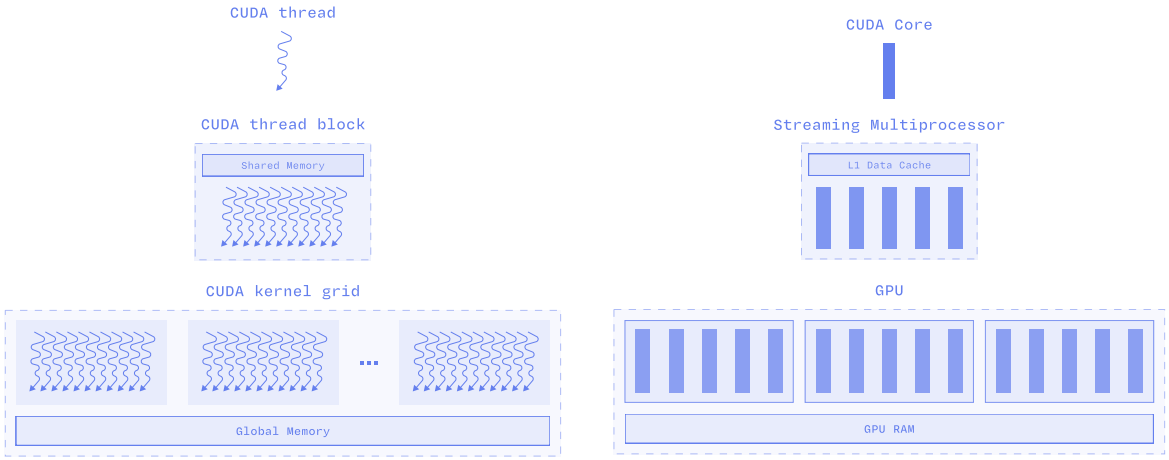


Figura 2: Estructura jerárquica de ejecución en CUDA: organización de hilos, bloques, grillas y su relación con la arquitectura física de la GPU. Extraída de Labs (2025a).

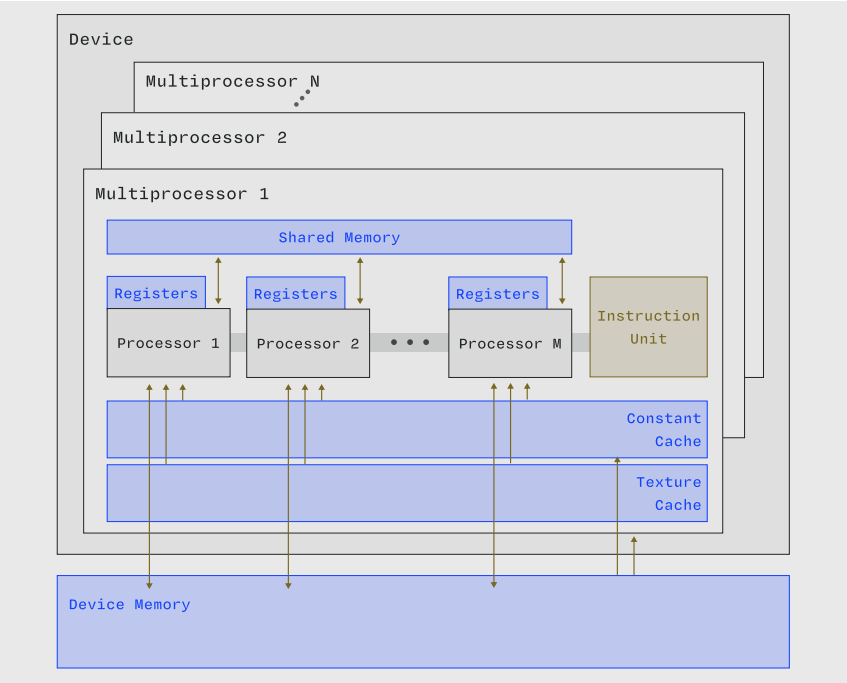


Figura 3: Jerarquía de memoria y organización interna de multiprocesadores en una arquitectura CUDA. Extraída de Labs (2025b).

Para profundizar en los conceptos descriptos en esta sección, se puede consultar Kirk & Hwu (2012).

Tensor Cores

Los Tensor Cores son unidades de procesamiento especializadas introducidas por NVIDIA en la arquitectura de GPU Volta y mejoradas en generaciones posteriores. Están diseñados específicamente para acelerar operaciones de multiplicación de matrices mediante instrucciones *matrix multiply-accumulate* (MMA) que procesan bloques completos en un solo ciclo. Estas instrucciones implementan, de manera eficiente, una operación de multiplicación de matrices del tipo:

$$D = \alpha op_1(A) op_2(B) + \beta C$$

donde A y B son matrices densas de entrada, C y D corresponden a las matrices de acumulación y resultado respectivamente, op denota una transformación lineal aplicada a cada operando (p. ej., la traspuesta), y α y β son escalares.

Esta operación es conocida como GEMM (*General Matrix–Matrix Multiply*), una operación fundamental en álgebra lineal que consiste en calcular una combinación lineal de un producto matricial. Su relevancia en este trabajo radica en que el problema abordado, como se mostrará más adelante, puede expresarse de manera directa como un producto de matrices, lo que permite mapearlo eficientemente a una operación GEMM.

La principal ventaja general de los Tensor Cores es su *throughput* masivo. Como se observa en Tabla 1, la GPU A40 ofrece 299.3 TOPS para operaciones INT8 y 598.7 TOPS (Tera Operaciones Por Segundo) para operandos de tipo INT4, muy superior a los 37.4 TFLOPS de GEMM para el tipo FP32 estándar (sin Tensor Cores). Los valores con asterisco corresponden a

operaciones con *sparsity* estructurada, donde matrices con patrones específicos de ceros permiten duplicar el *throughput* efectivo. Sin embargo, las precisiones INT4 e INT8 presentan ciertas limitaciones. Por un lado, no todas las arquitecturas de GPU soportan operaciones Tensor Core en INT8, ya que esta capacidad requiere que la GPU a utilizar posea *Compute Capability 7.0* (Arquitectura Volta) o superior y no está disponible en arquitecturas previas. Por otro lado, el soporte para INT4 es todavía más restrictivo, ya que el mismo está presente solo en algunas arquitecturas, específicamente Turing, Ampere y Ada Lovelace. Por otro lado el soporte para Tensor Cores INT4 desapareció en arquitecturas más recientes como Hopper y Blackwell. A esto se le suma que estas precisiones suelen requerir formatos de datos y alineamientos muy específicos, lo que incrementa la complejidad de implementación. En consecuencia, la utilización de operaciones basadas en INT4 e INT8 impone requisitos de hardware muy concretos, ya que el usuario debe disponer de una GPU cuya arquitectura soporte explícitamente estas precisiones en Tensor Cores, limitando su uso práctico.

Tipo de Operación	***Throughput (TFLOPS/TOPS)***
FP32 (sin Tensor Cores)	37.4
FP16 Tensor (acumulación FP16)	149.7 / 299.4*
TF32 Tensor	74.8 / 149.6*
BF16 Tensor (acumulación FP32)	149.7 / 299.4*
INT8 Tensor	299.3 / 598.6*
INT4 Tensor	598.7 / 1,197.4*
RT Core	73.1

Tabla 1: Rendimiento de Tensor Cores en GPU NVIDIA A40. *Con sparsity estructurada.

2.2.3. Métricas de Performance

Para evaluar el rendimiento de las distintas rutinas, se utilizaron varias métricas estándar del análisis de algoritmos paralelos:

1. **Tiempo de Ejecución**

El tiempo de ejecución corresponde al tiempo total que tarda un algoritmo en completar su tarea para una instancia determinada del problema. Esta métrica permite evaluar la eficiencia absoluta de un algoritmo y constituye un indicador fundamental del rendimiento.

2. **Escalabilidad**

Una de las más utilizadas es el *speedup* algorítmico, que evalúa cómo varía el desempeño de un mismo algoritmo al aumentar la cantidad de recursos de cómputo disponibles. Se define como:

$$S_n = \frac{T_1}{T_n}$$

donde T_1 es el tiempo de ejecución en un solo procesador (versión secuencial) y T_n el tiempo utilizando n recursos de cómputo.

También consideramos el *speedup* relativo, que se calcula como la razón entre el tiempo de la implementación base y el tiempo del algoritmo evaluado:

$$S_r = \frac{T_{\text{algoritmo base}}}{T_{\text{algoritmo evaluado}}}$$

Si bien estas métricas son estándares ampliamente aceptados en el análisis de algoritmos paralelos, su aplicación directa en GPU presenta ciertas dificultades conceptuales. A diferencia de un sistema clásico con múltiples procesadores, donde es posible definir de manera natural la cantidad de unidades de cómputo empleadas, las GPUs organizan sus recursos en términos de multiprocesadores (SM), *warps* e hilos concurrentes. En consecuencia, nociones como "n procesadores" o un tiempo de referencia T_1 no siempre tienen un análogo claro en este hardware.

3. **Throughput**

El *throughput*, medido en Giga Operaciones por Segundo (GOPS), representa la cantidad de operaciones aritméticas o computacionales que un algoritmo puede realizar por unidad de tiempo. Se calcula como:

$$\text{GOPS} = \frac{\text{nº total de operaciones ejecutadas}}{\text{Tiempo total de ejecución (s)}} \times 10^{-9}$$

Un mayor valor de GOPS indica un mayor rendimiento computacional y una mayor capacidad del sistema para manejar cargas de trabajo intensivas.

2.3. Trabajo Relacionado

En esta sección se presentan los trabajos más relevantes vinculados con los dos componentes centrales del problema que abordamos. La literatura relacionada se divide en tres líneas principales. La primera línea presenta una reformulación de la primera parte del problema, la obtención de la matriz de distancias euclidiana. La segunda corresponde a métodos algebraicos y algorítmicos para resolver APSP, entre ellos las formulaciones basadas en el álgebra min-plus y los algoritmos de clausura de Kleene. La tercera está compuesta por técnicas y bibliotecas optimizadas para el cálculo masivo de distancias, particularmente en GPU.

2.3.1. Construcción de la Matriz de Distancias

La matriz de distancias se obtiene luego de aplicar el Algoritmo 1:

Algoritmo 1: Distancias Euclidianas

$N_I \leftarrow$ *cantidad de individuos*
$N_S \leftarrow$ *cantidad de SNPs*
$X \leftarrow$ *matriz de individuos de tamaño* $N_I \times N_S$
$D \leftarrow$ *matriz de distancias de tamaño* $N_I \times N_I$
for each $i = 1, ..., N_I$ **do**
  for each $j = 1, ..., N_S$ **do**
    $D_{ij} = \sqrt{\sum_{k=1}^{N_S} \left(X_{ik} - X_{jk}\right)^2}$
  end
end

Algoritmo 1: Pseudocódigo para el cálculo de la matriz de distancias euclidianas.

donde la eficiencia permanece limitada por el hecho de que la operación dominante se implementa mediante bucles convencionales poco eficientes. Por esta razón, se exploró una reformulación algebraica del problema que permitiera utilizar rutinas altamente optimizadas.

La matriz de distancias $D^2 \in \mathbb{R}^{n \times n}$ se define como:

$$D_{ij}^2 = \sum_{k=1}^{d} \left(X_{ik} - X_{jk}\right)^2$$

donde $X \in \{0,1,2\}^{n \times d}$ es la matriz de entrada con n individuos y d SNPs. Esta expresión puede reformularse algebraicamente mediante la identidad:

$$d^2(x_i, x_j) = \|x_i\|^2 + \|x_j\|^2 - 2(x_i \cdot x_j)$$

permitiendo expresar la matriz completa como:

$$D^2 = \tilde{N} + \tilde{N}^T - 2XX^T$$

donde $\tilde{N}_{ij} = \|x_i\|^2$ para todo j, y XX^T contiene todos los productos escalares.

Esta reformulación es fundamental para las propuestas presentadas posteriormente, ya que permite aprovechar implementaciones altamente optimizados para multiplicación de matrices.

Un trabajo especialmente relevante para nuestro estudio es el presentado por Ltaief et al. (2024). Los autores se enfocan en acelerar cómputos fundamentales para estudios de asociación genómica (GWAS) a gran escala mediante técnicas de álgebra lineal adaptativas y de precisión mixta, explotando de forma explícita el alto rendimiento de los tensor cores de GPU NVIDIA (Ampere y Hopper). En relación directa con nuestro trabajo, en el artículo dedican una sección a rediseñar el cálculo de distancias euclidianas entre individuos como un problema de álgebra lineal, implementando esta operación mediante Tensor Cores INT8, explotando que los SNPs toman valores en {0,1,2}. Este estudio se realiza sobre un conjunto de datos compuesto por 305880 individuos y 43333 SNPs, obtenido del UK Biobank, y para experimentos mayores, utilizan datos sintéticos con ~13M de individuos y ~20M de SNPs, evaluando la *performance* del algoritmo en diferentes *clusters*: Summit (**18.432 V100 GPUs**), Leonardo (**4.096 A100 GPUs**), Frontier (**36.100 MI250X GPUs**), y Alps (**8.100 GH200 Superchips**).

A partir de este punto, consideraremos la matriz de distancias euclidianas al cuadrado D^2, sin aplicar la raíz cuadrada. Como se mencionó anteriormente, en la etapa de construcción del grafo de Fermat cada elemento de la matriz se eleva a una potencia α. Dado que calcular D_{ij}^{α} o calcular $\left(D_{ij}^2\right)^{\frac{\alpha}{2}}$ produce el mismo resultado, calcular la raíz cuadrada en esta etapa no aporta ningún beneficio y puede omitirse, simplificando el cálculo y mejorando la eficiencia elevando posteriormente la matriz a la potencia $\frac{\alpha}{2}$.

2.3.2. Resolución de Caminos Mínimos

En esta sección se analizan los algoritmos clásicos para resolver el problema de caminos mínimos entre todos los pares de vértices (*All-Pairs Shortest Paths*, APSP) (Kozen, 1991). Para cada par de vértices u y v en un grafo dirigido ponderado, se busca calcular dos valores fundamentales: $\text{dist}(u, v)$, que representa la longitud del camino más corto de u a v, y $\text{pred}(u, v)$, el penúltimo vértice en dicho camino más corto.

El problema APSP es fundamental en teoría de grafos, con aplicaciones que abarcan desde *routing* de redes y análisis de redes sociales hasta bioinformática y optimización de transporte. A diferencia del problema de camino mínimo desde una única fuente (SSSP), donde se calcula la distancia desde un vértice específico hacia todos los demás, APSP requiere calcular las distancias entre todos los pares posibles de vértices, resultando en una matriz completa de dimensión $|V| \times |V|$ de distancias.

Los algoritmos clásicos presentados se diferencian fundamentalmente en su enfoque: algunos reducen APSP a múltiples ejecuciones de algoritmos SSSP (BFS, Dijkstra, Bellman-Ford), mientras que otros emplean programación dinámica directa (Floyd-Warshall). El algoritmo de Johnson constituye un caso especial que combina ambos enfoques mediante una transformación ingeniosa de pesos.

Enfoque Basado en SSSP

La solución más directa al problema APSP consiste en ejecutar un algoritmo de camino mínimo desde una única fuente (SSSP) para cada vértice del grafo, llenando así cada fila de la matriz de distancias mediante una ejecución independiente. Esta estrategia, aunque conceptualmente simple, permite aprovechar la sofisticación de algoritmos SSSP ya optimizados y resulta particularmente efectiva en ciertos contextos.

Breadth-First Search (BFS)

Para grafos no ponderados o con pesos unitarios, el algoritmo BFS proporciona la solución más simple y eficiente. Al ejecutar BFS desde cada vértice, como se muestra en Algoritmo 2, se obtiene una complejidad total de $\mathcal{O}(VE)$, que se reduce a $\mathcal{O}(V^3)$ para grafos densos. El algoritmo explora el grafo por niveles, garantizando que cuando un vértice es visitado por primera vez, se ha encontrado el camino más corto hacia él.

Algoritmo 2: APSP-BFS

```
1  Input: V, E (grafo no ponderado)
2  Output: matriz dist[V × V]
3  for each vértice s ∈ V do
4  |  Inicializar cola Q ← {s}
5  |  dist[s, s] ← 0
6  |  for each vértice v ≠ s do
7  |  |  dist[s, v] ← ∞
8  |  end
9  |  while Q ≠ ∅ do
10 |  |  u ← Q.dequeue()
11 |  |  for each vecino v de u do
12 |  |  |  if dist[s, v] = ∞ then
13 |  |  |  |  dist[s, v] ← dist[s, u] + 1
14 |  |  |  |  Q.enqueue(v)
15 |  |  |  end
16 |  |  end
17 |  end
18 end
```

Algoritmo 2: APSP usando BFS para grafos no ponderados

Algoritmo de Dijkstra

Para grafos con pesos no-negativos, el algoritmo de Dijkstra (1959) ofrece una solución eficiente. La complejidad de ejecutar Dijkstra desde cada vértice, como se muestra en Algoritmo 3, depende de la estructura de datos utilizada para la cola de prioridad: con *heap* binario se obtiene $\mathcal{O}(VE \log V) = \mathcal{O}(V^3 \log V)$ para grafos densos, mientras que con *heap* de Fibonacci se puede alcanzar $\mathcal{O}(VE + V^2 \log V) = \mathcal{O}(V^3)$ en el caso denso, aunque esta última implementación raramente se utiliza en la práctica debido a su *overhead* constante.

El algoritmo mantiene un conjunto S de vértices cuyas distancias definitivas han sido determinadas, expandiendo iterativamente este conjunto seleccionando el vértice no incluido con menor distancia estimada. Esta estrategia garantiza la optimalidad gracias a la ausencia de pesos negativos.

Algoritmo 3: APSP-Dijkstra

```
1  Input: V, E, w (pesos no-negativos)
2  Output: matriz dist[V × V]
3  for each vértice s ∈ V do
4      for each vértice v ∈ V do
5          dist[s, v] ← ∞
6      end
7      dist[s, s] ← 0
8      Inicializar cola de prioridad Q con todos los vértices
9      while Q ≠ ∅ do
10         u ← Q.extractMin()
11         for each vecino v de u do
12             if dist[s, v] > dist[s, u] + w(uv) then
13                 dist[s, v] ← dist[s, u] + w(uv)
14                 Q.decreaseKey(v, dist[s, v])
15             end
16         end
17     end
18 end
```

Algoritmo 3: APSP usando Dijkstra para grafos con pesos no-negativos

Algoritmo de Bellman-Ford

El algoritmo de Bellman (1958) y Ford (1956), a diferencia de Dijkstra, puede manejar grafos con pesos negativos, aunque con mayor costo computacional. La idea fundamental se basa en la observación de que cualquier camino más corto simple en un grafo de $|V|$ vértices contiene a lo sumo $|V| - 1$ aristas. El algoritmo relaja iterativamente todas las aristas del grafo, garantizando que después de ℓ iteraciones, se han encontrado los caminos más cortos que utilizan a lo sumo ℓ aristas.

Para el problema APSP, se puede ejecutar Bellman-Ford desde cada vértice, como se muestra en Algoritmo 4, resultando en una complejidad de $\mathcal{O}(V^2E)$, que se reduce a $\mathcal{O}(V^4)$ para grafos densos. Alternativamente, se puede formular directamente como un algoritmo APSP que procesa simultáneamente todas las fuentes, manteniendo una matriz de distancias que se actualiza iterativamente.

Algoritmo 4: APSP-Bellman-Ford

```
1  Input: V, E, w
2  Output: matriz dist[V × V]
3  for all vértices u do
4  |  for all vértices v do
5  |  |  if u = v then
6  |  |  |  dist[u, v] ← 0
7  |  |  else
8  |  |  |  dist[u, v] ← ∞
9  |  |  end
10 |  end
11 end
12 for ℓ ← 1 to |V| − 1 do
13 |  for all vértices u do
14 |  |  for all aristas xv ∈ E do
15 |  |  |  if dist[u, v] > dist[u, x] + w(xv) then
16 |  |  |  |  dist[u, v] ← dist[u, x] + w(xv)
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

Algoritmo 4: All-Pairs Bellman-Ford

La ventaja principal de Bellman-Ford es su capacidad para detectar ciclos negativos: si después de $|V| - 1$ iteraciones aún es posible reducir alguna distancia, el grafo contiene un ciclo negativo. El uso de memoria es $\mathcal{O}(V^2)$ para almacenar la matriz de distancias. A pesar de su complejidad temporal superior, Bellman-Ford es esencial cuando se trabaja con pesos negativos y no se puede aplicar la optimización de Johnson.

Algoritmo de Johnson

El algoritmo de Johnson (1977) resuelve APSP en grafos con pesos negativos mediante una técnica de reponderación que permite usar Dijkstra en lugar de Bellman-Ford. El algoritmo transforma los pesos de las aristas asignando un «precio» $\pi(v)$ a cada vértice y redefiniendo $w'(uv) = \pi(u) + w(uv) - \pi(v)$. Esta transformación preserva los caminos más cortos porque para cualquier camino de u a v, el cambio neto de longitud es $\pi(u) - \pi(v)$, independiente del camino tomado.

Para garantizar que $w'(uv) \geq 0$ para toda arista, Johnson agrega una fuente artificial s con aristas de peso cero hacia todos los vértices, ejecuta Bellman-Ford desde s, y define $\pi(v) = \text{dist}(s, v)$.

El algoritmo, como se muestra en Algoritmo 5, alcanza complejidad $\mathcal{O}(VE \log V)$ con *heap* binario, siendo $\mathcal{O}(V^2)$ en espacio. Para grafos dispersos con pesos negativos es significativamente

más eficiente que Bellman-Ford múltiple, que requeriría $\mathcal{O}(V^4)$, aunque para grafos densos no mejora asintóticamente sobre Floyd-Warshall.

Algoritmo 5: Johnson-APSP

```
Input: V, E, w
Output: matriz dist[V × V]
// Fase 1: Agregar fuente artificial
Agregar nuevo vértice s
for each vértice v ∈ V do
    Agregar arista sv con w(sv) ← 0
end
// Fase 2: Calcular costos de vértices
Ejecutar Bellman-Ford desde s para calcular dist[s, ·]
if Bellman-Ford detectó ciclo negativo then
    return «El grafo contiene un ciclo negativo»
end
for each vértice v ∈ V do
    π(v) ← dist[s, v]
end
// Fase 3: Reponderar aristas
for each arista uv ∈ E do
    w'(uv) ← π(u) + w(uv) − π(v)
end
// Fase 4: Ejecutar Dijkstra desde cada vértice
for each vértice u ∈ V do
    Ejecutar Dijkstra desde u con pesos w' para obtener dist'[u, ·]
end
// Fase 5: Recuperar distancias originales
for each vértice u ∈ V do
    for each vértice v ∈ V do
        dist[u, v] ← dist'[u, v] − π(u) + π(v)
    end
end
return dist
```

Algoritmo 5: Algoritmo de Johnson para APSP

Algoritmo de Floyd-Warshall

El algoritmo de Floyd & Warshall (1962), cuyas raíces se remontan a trabajos de Kleene (1956), Roy (1959) y Warshall (1962), introduce una parametrización fundamentalmente diferente del subproblema de programación dinámica.

La idea general es, como se muestra en Algoritmo 6, a lo largo de $|V|$ iteraciones, seleccionar secuencialmente todos los vértices del grafo como posibles nodos intermedios en los caminos más

cortos. Para cada par de nodos *(u,v)*, se verifica si pasar por el nodo intermedio r proporciona una distancia menor a la previamente conocida, y en caso afirmativo, se actualiza la matriz de distancias.

Algoritmo 6: Floyd-Warshall

```
1  Input: V, E, w
2  Output: matriz dist[V × V]
3  // Inicialización
4  for all vértices u ∈ V do
5  |  for all vértices v ∈ V do
6  |  |  if u = v then
7  |  |  |  dist[u, v] ← 0
8  |  |  else if uv ∈ E then
9  |  |  |  dist[u, v] ← w(uv)
10 |  |  else
11 |  |  |  dist[u, v] ← ∞
12 |  |  end
13 |  end
14 end
15 // Relajación progresiva
16 for all vértices r ∈ V do
17 |  for all vértices u ∈ V do
18 |  |  for all vértices v ∈ V do
19 |  |  |  if dist[u, v] > dist[u, r] + dist[r, v] then
20 |  |  |  |  dist[u, v] ← dist[u, r] + dist[r, v]
21 |  |  |  end
22 |  |  end
23 |  end
24 end
25 return dist
```

Algoritmo 6: Floyd-Warshall para APSP

El algoritmo alcanza una complejidad temporal de $\mathcal{O}(V^3)$, óptima para grafos densos, y requiere espacio $\mathcal{O}(V^2)$ para almacenar la matriz de distancias. Esta simplicidad, combinada con la complejidad óptima y el uso eficiente de memoria, ha convertido a Floyd-Warshall en el algoritmo de elección para APSP en la mayoría de las aplicaciones prácticas con grafos densos.

Clausura de Kleene

Este problema puede formularse de manera natural usando la matriz de adyacencia W de un grafo ponderado. Cuando no existen aristas con pesos negativos, el cálculo de distancias mínimas puede interpretarse como la obtención de la **clausura de Kleene** de W en el **álgebra min-plus** (Asplund, 2014; D'alberto & Nicolau, 2007).

En este álgebra, la operación suma usual se reemplaza por el mínimo, y el producto por la suma:

$$x \oplus y := \min(x, y), \quad x \otimes y := x + y$$

De este modo, la clausura de Kleene de W se define como:

$$W^* = I \oplus W^{\otimes 2} \oplus W^{\otimes 3} \oplus ...,$$

donde $W^{\otimes k}$ representa las longitudes mínimas de todos los caminos con exactamente k aristas, y la suma infinita $\oplus$ acumula el mínimo para todos los posibles largos de camino. El resultado W^* es, precisamente, la matriz de distancias mínimas entre todos los pares.

Aunque la definición $W^* = I \oplus W^{\otimes 2} \oplus W^{\otimes 3} \oplus ...$ es una suma formal sobre términos infinitos, en grafos finitos sin ciclos de peso negativo esa suma se puede truncar, ya que basta considerar potencias hasta $W^{\otimes(n-1)}$, donde n es el número de vértices. Por tanto la clausura de Kleene es computable en el sentido práctico siempre que el grafo no tenga ciclos negativos.

Aun así, calcular explícitamente todas las potencias hasta $n-1$ resulta, en general, muy costoso. Un resultado clave, mostrado en Asplund (2014), es que esta clausura de Kleene puede calcularse por bloques, dividiendo W en submatrices:

$$W = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

En este marco, se demuestra que la clausura W^* puede escribirse como:

$$W^* = \begin{bmatrix} F^* & F^*BD^* \\ D^*CF^* & D^* + D^*CF^*BD^* \end{bmatrix}, \quad \text{donde } F = A + BD^*C.$$

Esta forma no surge arbitrariamente: proviene de modelar el problema como un autómata con dos estados (uno por cada bloque diagonal) y analizar todas las rutas posibles entre ellos, como se muestra en la Figura 4.

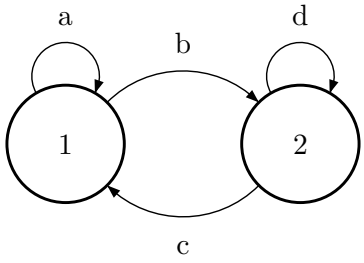


Figura 4: Autómata con dos estados sobre el alfabeto $\sum = \{a, b, c, d\}$.

Las expresiones:

$$1 \to 1 : (a + bd^*c)^*$$

$$1 \to 2 : (a + bd^*c)^*bd^*$$

$$2 \to 1 : d^*c(a + bd^*c)^*$$

$$2 \to 2 : d^* + d^*c(a + bd^*c)^*bd^*$$

describen formalmente todos los caminos que empiezan y terminan en cada subconjunto de nodos, permitiendo ciclos internos en cada bloque. Al traducir estas expresiones a notación matricial, se obtienen las fórmulas anteriores.

Formalmente, la multiplicación de matrices $E += FG$ (con $E, F, G \in \mathbb{R}^{n \times n}$) es simplemente $e_{ij} += \sum_{k=0}^{n-1} f_{ik} * g_{kj}$; donde la suma en realidad es el mínimo y la multiplicación es la suma. A esta operación le llamaremos ***minplus***, por tanto, la expresión anterior se puede escribir como $E = \min(E, \text{minplus}(F, G))$.

Método R-Kleene

Entre los diversos algoritmos que existen para resolver este problema, nuestra implementación final se basa en el algoritmo R-Kleene, presentado en D'alberto & Nicolau (2007), y posteriormente adaptado para arquitecturas GPU en Anjary (2023). Este algoritmo no depende de $|E|$, sino que depende únicamente de $|V|$, teniendo un orden de complejidad algorítmica de $\mathcal{O}(|V|^3)$, lo que en nuestro problema es $\mathcal{O}(n^3)$.

La diferencia clave con Floyd–Warshall, que realiza actualizaciones elemento a elemento y presenta dependencias secuenciales entre iteraciones (lo cual dificulta la paralelización), es que implementa de forma recursiva este cálculo de la clausura de Kleene por bloques.

Como se observa en Algoritmo 7, el algoritmo se llama recursivamente a sí mismo sobre el bloque A, para calcular los caminos mínimos que no salen de la primera mitad. Luego, actualiza los bloques B, C y D usando operaciones matriciales en el álgebra *minplus* para incorporar los nuevos caminos que atraviesan A. Posteriormente, se realiza otra llamada recursiva sobre D, y se actualizan nuevamente los bloques restantes para reflejar los caminos más cortos que atraviesan D. Este enfoque *divide-and-conquer* permite que las operaciones sobre bloques se procesen de forma paralela y eficiente.

R-Kleene(W)

/* $W = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ */

1 A = R-Kleene(A);
2 $B += A \times B$; ▷ $B += minplus(A, B)$
3 $C += C \times A$; ▷ $C += minplus(C, A)$
4 $D += C \times B$; ▷ $D += minplus(C, B)$

5 D = R-Kleene(D);
6 $B += B \times D$; ▷ $B += minplus(B, D)$
7 $C += D \times C$; ▷ $C += minplus(D, C)$
8 $A += B \times C$; ▷ $A += minplus(B, C)$

Algoritmo 7: Pseudocódigo del algoritmo R-Kleene. Extraído de D'alberto & Nicolau (2007).

Aunque su complejidad teórica sigue siendo $\mathcal{O}(n^3)$, R-Kleene presenta tiempos de ejecución significativamente menores en plataformas paralelas como GPU, gracias a su estructura de

bloque que favorece un mejor aprovechamiento del paralelismo y de la jerarquía de memoria. En implementaciones concretas se ha observado que, a gran escala, supera a Floyd–Warshall incluso cuando este último se ejecuta en GPU (D'alberto & Nicolau, 2007).

En la práctica, el trabajo Anjary (2023), reporta que este diseño recursivo puede mostrar un crecimiento casi logarítmico del tiempo de ejecución respecto al tamaño del problema, en sentido empírico, más que en el teórico, debido a la mejora en la localidad de datos y a la optimización del uso del ancho de banda en arquitecturas donde la latencia y el ancho de banda son los principales limitantes, más que el número total de operaciones aritméticas.

En el Apéndice A.2 se presenta un ejemplo detallado de ejecución del algoritmo sobre una matriz de distancias pequeña, ilustrando de manera explícita la división en bloques, las actualizaciones recursivas y la aplicación del álgebra *minplus*.

2.3.3. Bibliotecas de Alto Desempeño

Las bibliotecas de alto desempeño permiten acelerar las partes más costosas de nuestro problema al ofrecer rutinas ya optimizadas para operaciones numéricas, paralelización y manejo eficiente de datos. En lugar de implementar desde cero algoritmos complejos o estrategias de distribución de trabajo, estas herramientas proporcionan soluciones probadas que aprovechan al máximo el hardware disponible. En esta sección se introducen bibliotecas que evaluaremos para la implementación de nuestros algoritmos.

OpenBLAS

Esta biblioteca resulta especialmente útil cuando se reformula el cálculo de la matriz de distancias en términos de expresiones básicas de álgebra lineal. Aprovecha esta reformulación para utilizar rutinas BLAS (*Basic Linear Algebra Subprograms*) (BLAS Technical Forum, 2025) altamente optimizadas. BLAS es un estándar de facto para operaciones de álgebra lineal, con implementaciones que explotan instrucciones vectoriales (SIMD), múltiples núcleos de CPU y jerarquías de caché complejas. En particular, OpenBLAS proporciona excelente rendimiento en arquitecturas modernas.

RAPIDS

Existen bibliotecas GPU establecidas para cálculos de distancias, particularmente cuML (Team, 2025a) y cuVS (Team, 2025b) de NVIDIA RAPIDS. Estas bibliotecas proporcionan implementaciones optimizadas de métricas de distancia comunes, incluyendo distancia euclidiana, y están diseñadas para integración con flujos de trabajo de aprendizaje automático.

No obstante, sus implementaciones son genéricas, ya que están orientadas a comparar dos conjuntos arbitrarios de vectores y, por lo tanto, no pueden explotar las particularidades estructurales de nuestros datos.

Además, en nuestra evaluación experimental estas soluciones no lograron escalar a conjuntos de datos de gran tamaño. Los problemas observados incluyen limitaciones de memoria para

conjunto de datos con decenas de miles de individuos y cientos de miles de SNPs, y tiempos de ejecución no competitivos comparados con nuestras implementaciones especializadas.

Finalmente, cabe mencionar cuGraph (Team, 2025c), la biblioteca de análisis de grafos del mismo ecosistema. Si bien ofrece algoritmos acelerados por GPU para variantes de SSSP (*Single Source Shortest Path*), actualmente no incluye una implementación directa para el problema APSP.

CUTLASS

CUTLASS (*CUDA Templates for Linear Algebra Subroutines and Solvers*) (NVIDIA Corporation, s. f.-a) es una biblioteca de NVIDIA que proporciona plantillas de alto rendimiento para operaciones de álgebra lineal en GPU, construidas sobre CUDA. Su objetivo es permitir que desarrolladores e investigadores compongan *kernels* especializados para problemas de cómputo intensivo como GEMM, aprovechando al máximo la jerarquía de memoria de la GPU y las capacidades de paralelismo masivo del hardware moderno. A diferencia de bibliotecas como cuBLAS, CUTLASS ofrece una infraestructura modular y altamente configurable, permitiendo ajustar *layouts*, tamaños de bloques, *pipelines* y niveles de precisión para adaptarse de manera óptima a la arquitectura subyacente incluyendo Tensor Cores.

Los Tensor Cores son ideales para el problema de distancias euclidianas, ya que el cálculo de XX^T es una operación GEMM pura que domina el tiempo de ejecución.

En particular, CUTLASS soporta dos *layouts* principales:

- *row-major*: Los elementos de cada fila se almacenan contiguos en memoria. Para una matriz de tamaño $M \times N$, el elemento m_{ij} está en la posición $i \times N + j$.
- *column-major*: Los elementos de cada columna se almacenan contiguos en memoria. Para una matriz de tamaño $M \times N$, el elemento m_{ij} está en la posición $j \times M + i$.

cuASR

cuASR (*CUDA Algebra for Semiring-based Reasoning*) (hpcgarage, 2025), es una biblioteca desarrollada sobre CUTLASS cuyo propósito es generalizar el modelo de cómputo más allá del álgebra lineal clásica, permitiendo ejecutar operaciones GEMM sobre semianillos arbitrarios directamente en GPU. Mientras que CUTLASS se centra en acelerar multiplicaciones de matrices estándar bajo el álgebra convencional $(+, x)$, cuASR expone la misma infraestructura de alto rendimiento pero permitiendo redefinir las operaciones fundamentales del GEMM:

- el producto del semianillo (por defecto la multiplicación),
- la suma del semianillo (por defecto la suma),
- y los elementos identidad correspondientes.

Esta flexibilidad hace posible expresar una amplia variedad de algoritmos como una sola llamada GEMM optimizada, incluyendo problemas de caminos mínimos, propagación de distancias, grafos dirigidos, análisis de redes, y otros patrones computacionales que pueden formularse en términos de álgebra *min-plus*, *max-plus*, lógico-booleano o incluso semianillos definidos por el usuario.

graph-tool

La biblioteca graph-tool (graph-tool developers, s. f.) es un paquete de alto rendimiento para el análisis y manipulación de grafos, diseñado en C++ y expuesto a través de una interfaz Python para combinar eficiencia y facilidad de uso. Su arquitectura, basada en plantillas y estructuras de datos optimizadas, permite ejecutar algoritmos de teoría de grafos a velocidades significativamente superiores a las de otras bibliotecas puramente en Python. En particular, la función `shortest_distance` se destaca por su implementación altamente optimizada de diversos algoritmos de caminos mínimos, como Dijkstra, Bellman–Ford, Breadth-First Search (BFS) para grafos no ponderados, y Johnson para grafos con pesos no negativos o dispersos. La selección automática del algoritmo apropiado según la estructura y propiedades del grafo permite obtener resultados de manera eficiente incluso en grafos muy grandes. Gracias a estas optimizaciones, graph-tool se convierte en una herramienta poderosa para el análisis intensivo de redes y para aplicaciones en las que el rendimiento es un factor crítico.

SciPy

La biblioteca SciPy (SciPy Community, s. f.) es uno de los pilares del ecosistema científico de Python, ya que proporciona un amplio conjunto de herramientas numéricas optimizadas para análisis, álgebra lineal, optimización y procesamiento de datos. Dentro de sus módulos, la función `pdist` (del submódulo `scipy.spatial.distance`) permite calcular de manera eficiente todas las distancias por pares entre elementos de un conjunto, utilizando implementaciones vectorizadas y altamente optimizadas. Esto resulta especialmente útil en problemas donde se requiere construir matrices de distancias densas a partir de datos de alta dimensión. Por otro lado, SciPy también ofrece una implementación del algoritmo de Floyd–Warshall en `scipy.sparse.csgraph`, que permite obtener las distancias mínimas entre todos los pares de nodos de un grafo, incluso cuando este se representa mediante matrices dispersas.

Capítulo 3

Definición de la Línea Base

Este capítulo presenta el estudio detallado de las estrategias de cómputo actualmente empleadas para el procesamiento de datos genómicos por las docentes del IMERL, con el objetivo de establecer una implementación de referencia (línea base) frente a la cual evaluar posteriores optimizaciones.

3.1. Análisis de la Implementación Actual

En esta sección se analiza cómo se representan internamente los datos genéticos en forma matricial, cómo se calcula la matriz de distancias entre individuos, y cómo a partir de dicha matriz se construye el grafo de Fermat junto con la aplicación algoritmos de caminos mínimos.

3.1.1. Representación Interna

Actualmente, se representa el conjunto $N_I \times N_S$ como una matriz X haciendo uso de la biblioteca `numpy.ndarray` (Developers, 2025a), donde X_{ij} representa el *j-ésimo* SNP del *i--ésimo* individuo, siendo este SNP representado como un entero de 4 bytes. Esto implica que, actualmente, la representación de la matriz X requiere al menos $4N_I N_S$ bytes de memoria.

Para comprender mejor la naturaleza de los datos, se realizó un análisis de la frecuencia de ocurrencia de los valores observados en los SNPs. La Figura 5 muestra el histograma correspondiente, construido a partir del archivo real `19mil_sin_mex.vcf` suministrado por las docentes del IMERL, el cual contiene datos genómicos humanos. Si bien se trata de datos reales y completos del archivo, su representatividad está limitada al grupo específico de individuos incluidos; es decir, los patrones observados en este histograma describen fielmente este conjunto, pero no necesariamente se extienden a la población humana general. La matriz utilizada tiene dimensiones 2312×19916, y presenta una distribución de valores en los SNPs donde los códigos 0 y 1 predominan, mientras que el valor 2 aparece con menor frecuencia pero aún de manera significativa.

Este comportamiento llevó a descartar ciertos enfoques preliminares que asumían una matriz binaria (con solo dos posibles valores), o estrategias basadas en representar la matriz como dispersa, hipótesis que habrían sido razonables si, por ejemplo el 0, hubiera sido abrumadoramente dominante. Sin embargo, la distribución observada revela una representación lo bastante equilibrada de los tres valores como para justificar el tratamiento completo de los tres valores.

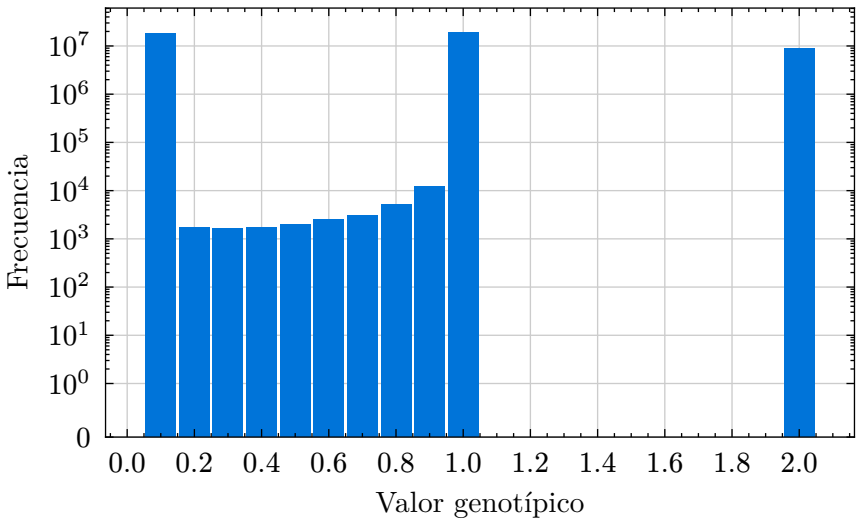


Figura 5: Frecuencia de los valores de SNPs sobre el conjunto de datos `19mil_sin_mex.vcf`.

La Figura 5 presenta la información utilizando una escala logarítmica en el eje vertical. Esta transformación permite distinguir con mayor claridad la presencia de valores no enteros, valores atípicos resultantes del proceso de imputación, que quedan visualmente ocultos en la escala lineal debido a la gran disparidad entre las frecuencias de los valores principales (0, 1 y 2) y la baja frecuencia de estos valores intermedios. Estos valores se deben, según se identificó, a una estrategia de imputación utilizada durante la preparación de los datos: en ciertos casos, se reemplazaron valores faltantes con la media del SNP correspondiente.

3.1.2. Construcción del Grafo de Fermat

El proceso de construcción del grafo de Fermat comienza con el cálculo de la matriz de distancias euclidianas, donde la solución actual utiliza la implementación `scipy.spatial.distance.pdist` (Developers, 2025a) que retorna el triángulo superior de dicha matriz. Luego se crea la matriz de distancias euclidianas D usando el resultado obtenido junto a `scipy.squareform` (Developers, 2025b). Este cómputo tiene una complejidad espacial de $\mathcal{O}(N_I^2)$ y un complejidad computacional de $\mathcal{O}(N_I^2 N_S)$.

Finalmente, esta matriz es utilizada para generar el grafo de Fermat a partir del estimador definido en Capítulo 2. Primero cada componente de la matriz obtenida en la etapa anterior, es elevado a cierta potencia α mediante la función `numpy.power` (Developers, 2025b). Luego, se aplica un algoritmo de búsqueda para encontrar el camino más corto entre cada par de nodos, que, en este caso, se utiliza el algoritmo de Floyd-Warshall, siguiendo la implementación de `scipy.sparse.csgraph.floyd_warshall` (Developers, 2025c).

Por este motivo, la construcción del grafo de Fermat, en la implementación actual, tiene un orden de complejidad $\mathcal{O}\big(N_I^2 N_S + N_I^3\big)$.

Las implementaciones de pdist y floyd_warshall mencionadas anteriormente se basan en los pseudocódigos del Algoritmo 1 y Algoritmo 6 respectivamente, con breves modificaciones que explotan características de arquitecturas CPU.

3.2. Análisis del Problema

En esta sección se examinan las características fundamentales del problema, sus restricciones, los datos involucrados y los factores que influyen en su complejidad. Este análisis permite identificar los aspectos críticos que deben ser considerados en el diseño de la solución, así como las limitaciones que condicionan su implementación. A partir de esta revisión detallada, se establece una base sólida para justificar las decisiones metodológicas y las técnicas empleadas en los capítulos posteriores.

3.2.1. Descomposición del Problema

En aplicaciones genómicas reales, las instancias del problema pueden alcanzar escalas masivas que involucran cientos de miles de individuos y millones de SNPs. Estas dimensiones presentan dos desafíos inmediatos. Por un lado, la matriz de entrada puede exceder la capacidad de memoria de un único dispositivo de cómputo. Por otro, el tiempo de ejecución en una sola máquina resulta prohibitivo.

Afortunadamente, la estructura matemática del problema admite una descomposición natural. La distancia euclidiana se define como una suma sobre todas las dimensiones (SNPs), y esta suma puede dividirse en subsumas independientes. Esto permite particionar la matriz de entrada $X_{N_I \times N_S}$ en bloques verticales $\tilde{X}_1, \tilde{X}_2, ..., \tilde{X}_P$, donde cada bloque contiene el conjunto completo de individuos pero solo un subconjunto de SNPs.

Cada bloque puede procesarse de manera completamente independiente en un dispositivo distinto, produciendo una matriz de distancias parciales. Una vez completados todos los cálculos parciales, las matrices resultantes se agregan mediante suma elemento a elemento para reconstruir la matriz de distancias completa. Esta estrategia es equivalente a procesar la matriz original de forma monolítica, pero distribuye la carga entre múltiples recursos de cómputo. El esquema general de este proceso se presenta en el Algoritmo 8.

En entornos distribuidos (clústeres con múltiples nodos), esta distribución se implementa típicamente mediante interfaces de paso de mensajes como MPI (Message Passing Interface Forum, 2021). El proceso maestro distribuye los bloques a los procesos trabajadores, donde cada uno ejecuta el cálculo sobre su fragmento asignado, y finalmente las contribuciones parciales se combinan mediante una operación de reducción colectiva.

Esta aproximación no solo resuelve las limitaciones de memoria, sino que además permite escalabilidad horizontal. Instancias arbitrariamente grandes pueden abordarse añadiendo más nodos al clúster. El único requisito es que cada bloque individual quepa en la memoria del dispositivo asignado, lo cual puede ajustarse eligiendo un número adecuado de particiones.

Algoritmo 8: Distancias por pares en entorno distribuido

Input: $X_{N_I \times N_S}$, P *(número de dispositivos)*
Output: $D_{N_I \times N_I}$
Particionar X en P bloques: $\tilde{X}_1, ..., \tilde{X}_P$
for $p = 1$ **to** P **do**
  $D_p \leftarrow \text{Compute}(\tilde{X}_p)$
end for
$D \leftarrow \sum_{p=1}^{P} D_p$

Algoritmo 8: Esquema general de procesamiento distribuido.

3.2.2. Simetría del Problema

En primer lugar, es fácil observar que el problema es simétrico, ya que el grafo de Fermat a calcular es un grafo no dirigido, es decir, para toda pareja de individuos x, y se cumple $w_{\alpha(x,y)} = w_{\alpha(y,x)}$. Además su matriz de adyacencia presenta la diagonal nula, dado que $w_{\alpha(x,x)} = 0$ para todo x. Estas propiedades permiten reducir el esfuerzo computacional, ya que es suficiente calcular únicamente la mitad de las combinaciones posibles para resolver el problema planteado.

3.2.3. Características de los Datos

Para mantener la coherencia con la representación de los SNPs establecida en el Capítulo 1, donde cada genotipo se codifica mediante valores discretos 0, 1 y 2, resulta conveniente emplear técnicas de imputación que preserven dicha estructura, como la imputación mediante el valor más frecuente del SNP. Además, resulta interesante explotar esta característica para reducir significativamente el uso de memoria al momento de representar los individuos en el dispositivo.

En algunos casos podría resultar deseable trabajar con datos normalizados por individuo, de modo que cada fila x_i se re-escale por un factor $s_i = \sum_{k=1}^{m} X_{ik}$. La normalización explícita produciría una matriz $X_{\texttt{norm}}$ donde cada elemento es $\frac{X(i,k)}{s_i}$, pero construirla implica un costo adicional significativo. Sin embargo, gracias a la formulación utilizada para D^2, es posible incorporar la normalización sin modificar la matriz de entrada. Basta con pre-calcular para cada individuo el valor s_i, y aplicar los factores de escala directamente sobre los términos ya computados:

$$D^2_{norm}(i,j) = \frac{\|x_i\|^2}{s_i^2} + \frac{\|x_j\|^2}{s_j^2} - 2\frac{(XX^T)_{ij}}{s_i s_j}$$

De este modo, se evita normalizar cada elemento de forma explícita, reduciendo el costo computacional y manteniendo la compatibilidad con las implementaciones que presentaremos en capítulos siguientes.

3.3. Conclusiones

El análisis realizado evidencia que el principal desafío computacional en el procesamiento de datos genómicos radica en el cálculo de la matriz de distancias entre individuos. Esta etapa, debido a la enorme cantidad de pares a comparar y al volumen de información asociado a cada

individuo, constituye el cuello de botella dominante, tanto en términos de cómputo como de consumo de memoria.

En contraste, la etapa de la construcción del grafo de Fermat, presenta una complejidad teórica costosa, pero opera sobre matrices de dimensión sustancialmente menor, lo que las vuelve comparativamente menos exigentes. En esta etapa, se adopta como línea base la implementación actual de `scipy.sparse.csgraph.floyd_warshall`, basada en el algoritmo de Floyd–Warshall, ya que resulta la opción más adecuada para las características del grafo considerado. En particular, BFS no es aplicable por tratarse de un grafo ponderado; Dijkstra, aunque válido para pesos positivos, resulta poco eficiente en grafos densos; Bellman–Ford es innecesariamente costoso dado que no existen pesos negativos; y Johnson pierde sus ventajas estructurales al tratarse de un grafo denso donde su complejidad se aproxima a la de Floyd–Warshall.

Por otro lado, la reformulación algebraica del cálculo de distancias habilita el uso de rutinas BLAS altamente optimizadas, y en particular de OpenBLAS, que ofrece un excelente aprovechamiento de instrucciones vectoriales, múltiples núcleos de CPU y jerarquías de memoria modernas. El algoritmo consta de tres etapas. Primero, se construye la matriz X a partir de los datos (valores SNP en $\{0, 1, 2\}$), representando cada valor como FP32 para compatibilidad con BLAS en precisión simple. Luego, se calcula $G = XX^T$ mediante SYRK (*symmetric rank-k updates*), donde la diagonal de G contiene automáticamente las normas cuadradas: $G_{ii} = \|x_i\|^2$. Esto elimina la necesidad de calcular las normas por separado. Finalmente, se aplica la fórmula de distancia:

$$D_{ij}^2 = G_{ii} + G_{jj} - 2G_{ij}$$

La principal ventaja de esta aproximación es que aprovecha décadas de optimizaciones en BLAS, resultando en código portable entre arquitecturas CPU con buen balance entre simplicidad de implementación y rendimiento. La complejidad temporal de esta implementación es $\mathcal{O}(N_I^2 N_S)$, dominada por el producto matricial SYRK que es aproximadamente x2 más rápido que GEMM completo gracias a la explotación de simetría.

La limitación principal es la desventaja de usar punto flotante de precisión simple. Pues nos vemos obligados a utilizar 32 bits para la representación de 3 valores discretos, degradando el uso de memoria e introduciendo errores de redondeo en la fórmula $\|x_i\|^2 + \|x_j\|^2 - 2(x_i \cdot x_j)$.

Por estas razones, la implementación basada en SYRK de OpenBLAS se adopta como la línea base para la primera etapa del problema, constituyendo la alternativa más eficiente disponible sobre CPU.

Aun así, para manejar conjuntos de datos genómicos de escala creciente resulta necesario recurrir a plataformas de cómputo de mayor paralelismo. En particular, el uso de GPUs se presenta como una estrategia prometedora para acelerar significativamente el cálculo de distancias y superar las limitaciones inherentes a las arquitecturas de CPU. Sin embargo, cabe señalar, que la comparación directa entre implementaciones en CPU y GPU tiene un valor interpretativo limitado, pues las arquitecturas y los modelos de ejecución difieren de manera fundamental. La referencia en CPU sirve, únicamente como un punto de contraste para cuantificar la magnitud de la aceleración obtenida, y no como una comparación estricta entre plataformas equivalentes.

Para realizar esta operación en la GPU usamos la biblioteca CUTLASS. Para el cálculo de la matriz $G = XX^T$, donde X es de tamaño $N_I \times N_s$ y se almacena en formato *row-major*, se adopta una estrategia de organización de datos que permite evitar la transposición explícita. En particular, se define el operando A en formato *row-major* y debemos "engañar" a CUTLASS indicando que el operando B se encuentra en *column-major* (equivalente a trasponer), aprovechando la organización de datos para mantener los accesos *coalesced* y maximizando el ancho de banda efectivo.

El siguiente capítulo estará dedicado al diseño e implementación de una solución personalizada tanto en CPU como en GPU, cuyo desempeño será evaluado en relación con las líneas bases establecidas en este capítulo.

Capítulo 4

Propuesta de Resolución

«... I seem, then, in just this little thing to be wiser than this man at any rate, that what I do not know I do not think I know either.»

SÓCRATES
from the Henry Cary literal translation of 1897.

Este capítulo presenta las propuestas de resolución implementadas para el problema planteado. El desafío central consiste en calcular eficientemente la matriz de distancias euclidianas al cuadrado entre todos los pares de individuos en un conjunto de datos genómicos, seguido de la aplicación de algoritmos de caminos mínimos para finalmente formar el grafo de Fermat.

Se han desarrollado implementaciones tanto para CPU como para GPU, cada una diseñada para aprovechar diferentes características del hardware moderno y la naturaleza específica de los datos genómicos. Las soluciones propuestas explotan el uso desde bibliotecas optimizadas de álgebra lineal hasta hardware especializado como los Tensor Cores, pasando por técnicas personalizadas de cómputo a nivel de bits.

4.1. Cálculo de la Matriz de Distancias

Las GPUs modernas ofrecen un paralelismo masivo que puede acelerar dramáticamente el cálculo de matrices de distancias. Sin embargo, para aprovechar este potencial, debemos diseñar cuidadosamente tanto la representación de datos como los patrones de acceso a memoria. Esta sección presenta implementaciones en GPU y CPU, que combinan compresión de datos, operaciones bit a bit, y una arquitectura jerárquica de memoria inspirada en técnicas modernas de multiplicación de matrices.

4.1.1. Esquema de Codificación de SNPs

El primer paso hacia una implementación eficiente es reconocer una propiedad fundamental de nuestros datos: los SNPs toman solo tres valores posibles (0, 1, 2). Esta observación nos permite usar una representación mucho más compacta que el byte tradicional.

Dado que solo necesitamos distinguir entre tres estados diferentes, una codificación de 2 bits es suficiente:

Valor SNP	Genotipo	Codificación 2-bit
0	AA (homocigoto ref.)	00
1	AG (heterocigoto)	01
2	GG (homocigoto alt.)	10

Tabla 2: Codificación binaria de 2 bits para valores de SNPs.

Esta codificación tiene una estructura particular: el valor del SNP se mapea directamente a su representación binaria. Esta elección no es arbitraria, está diseñada para habilitar una serie de optimizaciones algebraicas que se presentarán más adelante.

Con esta codificación es posible empaquetar múltiples SNPs en un solo entero de 32 bits. Dado que cada SNP ocupa 2 bits, un entero de tipo `u32` puede almacenar exactamente 16 SNPs (16×2 = 32 bits). Así, el conjunto de datos original, de dimensiones $N_I \times N_S$, puede representarse en memoria como una matriz *comprimida* $\tilde{X}$ de tamaño $m \times k$, donde cada entrada es un entero de 32 bits y:

$$m = N_I;\ k = \left\lceil \frac{N_s}{16} \right\rceil$$

Para un *dataset* típico con $N_S = 50.000$ SNPs, se obtiene $k = 3.125$ palabras por individuo. El empaquetamiento se realiza concatenando los 2 bits de cada SNP consecutivos.

4.1.2. Uso Eficiente de la Jerarquía de Memoria

Para paralelizar el cómputo de la matriz D en la GPU, la matriz D se divide en submatrices disjuntas $\tilde{D}$ de dimensiones $m_s \times n_s$, y las submatrices se procesan en paralelo con un bloque de hilos que calcula una submatriz $\tilde{D}$ de forma independiente de los demás bloques de hilos. Para calcular $\tilde{D}$, se itera sobre la dimensión K. En cada iteración, una submatriz A de tamaño $m_s \times k_s$ y una submatriz B de tamaño $n_s \times k_s$ se transfieren desde la memoria global del dispositivo a la memoria compartida como se muestra en la Figura 6.

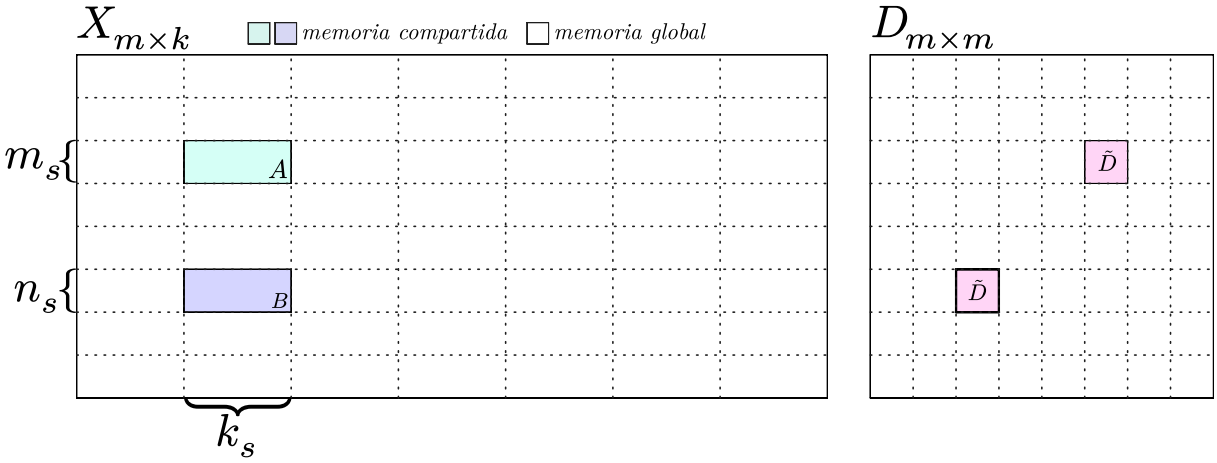


Figura 6: Procesamiento realizado por un bloque de hilos para actualizar su submatriz $\tilde{D}_{m_s \times n_s}$.

Se calcula la matriz de distancia euclidiana al cuadrado entre estas submatrices y el resultado se utiliza para actualizar $\tilde{D}$. Las submatrices A, B y $\tilde{D}$ se denominan a menudo bloques. En total hay $\frac{K}{k_s}$ iteraciones (en el caso más simple, donde K es divisible por k_s). La capacidad limitada de la memoria compartida es la razón por la que la dimensión K se divide en bloques más pequeños k_s. Los bloques completos $m_s \times K$, $K \times n_s$ simplemente no cabrían en la memoria compartida disponible. Por ahora, no se explicará por qué las matrices se cargan en la memoria compartida ni cómo se computa $\tilde{D}$ a partir de A y B; esto será discutido en la próxima sección. Centrémonos en la transferencia eficiente de datos de la memoria global a la memoria compartida como primer paso.

c

Pseudocódigo del algoritmo, desde la perspectiva de un bloque de hilos

```
D = 0
__shared__ u32 A[ms][ks]
__shared__ u32 B[ns][ks]
for (i=0; i<K/ks; i++) {
    A = cargar i-ésimo bloque A // desde memoria global a compartida
    B = cargar i-ésimo bloque B // desde memoria global a compartida
    D += distance(A, B)
    > Calcular distancia entre A_{m_s×k_s} y B_{n_s×k_s} y acumular en D̃_{m_s×n_s}
}
store(D)
> Transferir el resultado D̃_{m_s×n_s} a memoria global
```

Las transferencias de datos desde la memoria global a la memoria compartida presentan una latencia significativamente mayor en comparación con las operaciones aritméticas. Durante este tiempo, los hilos se ven obligados a detenerse, esperando inactivamente los datos necesarios para calcular la distancia entre los bloques A y B. Una forma de mitigar esta latencia es superponiendo las transferencias de datos con los cálculos, aprovechando el paralelismo a nivel de instrucción (ILP). En las implementaciones de GEMM, se suele utilizar una técnica conocida como *doble búfer* para superponer cómputo y transferencias de datos, una estrategia ampliamente empleada en *kernels* de SGEMM optimizados para GPU, como se describe en Salykov (2025).

Pseudocódigo del algoritmo, desde la perspectiva de un bloque de hilos

```
D = 0
__shared__ u32 A[2][ms][ks]
__shared__ u32 B[2][ns][ks]
> Definir doble búfer en memoria compartida para los bloques A y B
A[0] = cargar primer bloque A
B[0] = cargar primer bloque B

for (i=0; i<(K/ks-1); i++) {
    idx = i%2
    prefetch_idx = (i+1)%2
    // prefetch de los próximos bloques
    A[prefetch_idx] = cargar próximo bloque A
    B[prefetch_idx] = cargar próximo bloque B
    D += distance(A[idx], B[idx])
    > Calcular distancia entre los bloques A_{m_s×k_s} y B_{n_s×k_s} cargados en la iteración anterior y acumular en D̃_{m_s×n_s}
}
D += distance(A[prefetch_idx], B[prefetch_idx])
> Calcular distancia entre los bloques últimos bloques A_{m_s×k_s} y B_{n_s×k_s} y acumular en D̃_{m_s×n_s}
store_to_global_memory(D)
> Transferir el resultado D̃_{m_s×n_s} a memoria global
```

Cabe destacar, que la operación `D += distance(A[idx], B[idx])` no depende de los bloques respecto a los índices `prefetch_idx`, lo que permite que las instrucciones de cómputo de distancia entre bloques se ejecuten en paralelo con las de transferencia de datos. Sin embargo, esto implica

duplicar el uso de memoria compartida, ya que se necesitan almacenar dos bloques en lugar de uno. Afortunadamente, las GPU modernas cuentan con suficiente memoria compartida para admitir el doble búfer.

Ya hemos introducido varios parámetros, como el tamaño de los bloques (m_s, k_s, n_s) y el número de hilos por bloque. La elección de estos parámetros depende en gran medida de la forma de X, así como de la arquitectura de la GPU subyacente. Por ejemplo, cuBLAS implementa varios *kernels* GEMM optimizados para diversas formas de matriz y arquitecturas de GPU. En tiempo de ejecución, selecciona el *kernel* más apropiado mediante un enfoque heurístico. El tamaño de los bloques (m_s, k_s, n_s) afecta no solo a cómo se obtienen los datos de la memoria global, sino también a cómo se organiza el trabajo en todos los pasos posteriores (cargas de memoria compartida, operaciones aritméticas, almacenamientos en memoria global) entre los hilos para lograr el mejor rendimiento posible. La elección del tamaño de los bloques y del número de hilos por bloque también influye en el uso de la memoria compartida y los registros, lo que puede provocar una disminución del rendimiento si no se tiene en cuenta. Como es de esperar, identificar los valores óptimos de los parámetros requiere un excelente conocimiento del hardware y una amplia experimentación.

A modo de ejemplo, comenzaremos con la implementación de un *kernel* de `128x128x8`. La elección final de estas dimensiones se ajustará posteriormente durante la etapa de evaluación experimental, donde se realizarán distintas pruebas para determinar la configuración que ofrezca el mejor rendimiento.

Ahora que conocemos las dimensiones del bloque y el número de hilos por bloque, analicemos cómo organizar eficientemente la carga de datos desde la memoria global y el almacenamiento a la memoria compartida. Primero, necesitamos cargar la submatriz A de `128x8` usando `256` hilos. Esto resulta en que cada hilo cargue `128*8/256 = 4` elementos `u32` de la memoria global. Existen varias maneras de organizar la carga del bloque. Para lecturas o escrituras de la memoria global, siempre se busca que los accesos sean contiguos o *coalesced*, de modo que 32 hilos en un bucle accedan a 32 elementos `u32` consecutivos en la memoria. Si un acceso a memoria es *coalesced*, se utilizará el número mínimo de transacciones de memoria. Cuando un *warp* ejecuta una instrucción que accede a la memoria global, fusiona los accesos a la memoria de los hilos dentro del *warp* en una o más de estas transacciones de memoria dependiendo del tamaño de la palabra a la que accede cada hilo y la distribución de las direcciones de memoria entre los hilos.

Sin embargo, esto no es posible en el caso del bloque A, ya que cada fila del bloque contiene solo 8 elementos consecutivos. No obstante, incluso en tales casos, es preferible que hilos consecutivos en un bucle accedan a elementos consecutivos en la memoria, lo que generalmente resulta en un mejor rendimiento.

16 bytes

r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}	r_{16}	r_{17}	r_{18}	r_{19}	r_{20}	r_{21}	r_{22}	r_{23}	r_{24}	r_{26}	r_{25}	r_{27}	r_{28}	r_{29}	r_{30}	r_{31}
t_0				t_1				t_2				t_3				t_4				t_5				t_6				t_7			
t_8				t_9				t_{10}				t_{11}				t_{12}				t_{13}				t_{14}				t_{15}			
t_{16}				t_{17}				t_{18}				t_{19}				t_{20}				t_{21}				t_{22}				t_{23}			
t_{24}				t_{25}				t_{26}				t_{27}				t_{28}				t_{29}				t_{30}				t_{31}			

Figura 7: Transferencia del bloque A a memoria compartida, donde cada hilo t_i carga 16 bytes consecutivos desde memoria global.

En la Figura 7 se muestra cómo se implementa la carga del bloque A. En este caso, almacenamos 16 bytes (equivalentes a 128 bits) por hilo mediante instrucciones vectoriales. Esto supone un total de 512 bytes por solicitud a nivel de *warp*. La GPU divide la solicitud en 4 transacciones (los hilos `t0` a `t7` realizan una transacción, los hilos `t8` a `t15` otra y así sucesivamente), cada una de 128 bytes.

Ahora bien, si almacenamos los datos según nuestro esquema, cada hilo dentro de los hilos `t0-t7` accedería a bancos de memoria distintos, lo que resultaría en el acceso a 32 bancos de memoria por transacción de memoria. Lo mismo se aplica a las transacciones de memoria restantes, es decir, `t8-t15`, `t16-t23`, etc.

El mismo procedimiento se realiza para cargar y almacenar la submatriz $B_{n_s \times k_s}$.

Con los bloques A y B ahora en memoria compartida, veamos cómo cargar eficientemente desde la memoria compartida y calcular el bloque $\tilde{D}$. Para ello, profundizaremos en nuestra estrategia de paralelización y describiremos el algoritmo desde la perspectiva de un *warp*.

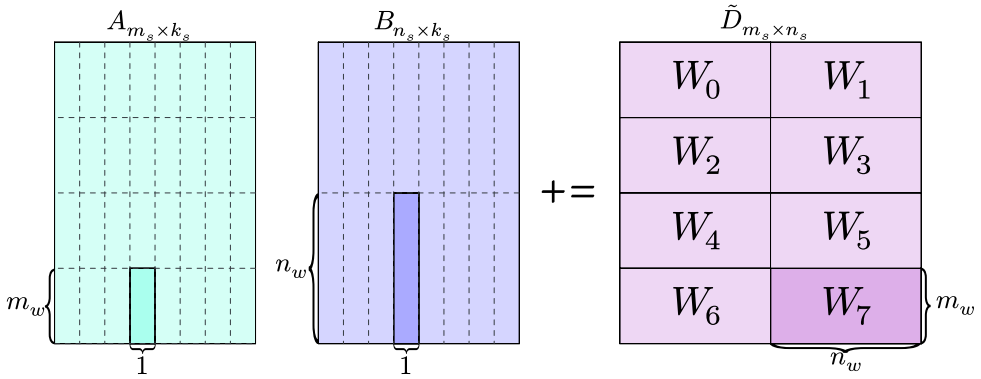


Figura 8: División del bloque $\tilde{D}$ en 8 regiones y su asignación a *warps*.

El bloque de hilos lanzado consta de 256 hilos, lo que corresponde a 256/32=8 *warps*. El bloque $\tilde{D}$, con dimensiones 128×128 se divide, por lo tanto, en 8 regiones $\tilde{D}_w$ etiquetadas como $W_0, ..., W_7$ como se aprecia en la Figura 8.

Cada región $\tilde{D}_w$ tiene dimensiones $m_W \times n_W$ y se calcula mediante un único *warp*: W_0 se calcula mediante los hilos `t0-t31`, W_1 se calcula mediante los hilos `t32-t63`, y así sucesivamente, hasta que W_7 se calcula mediante los hilos `t224-t255`.

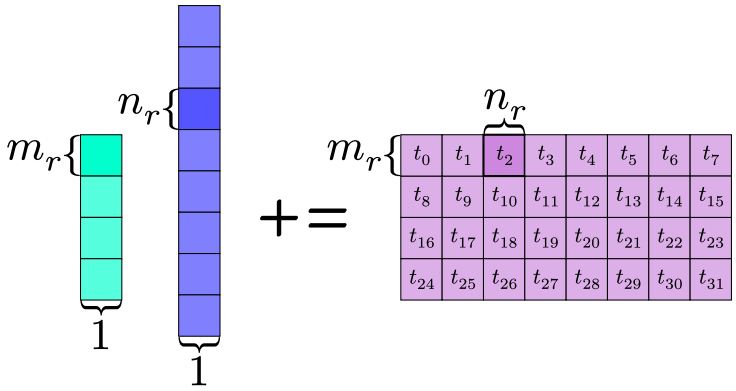


Figura 9: Distribución del trabajo de cálculo de una subregión (W_7) entre los 32 hilos de un único warp.

La Figura 9 utiliza W_7 como ejemplo para demostrar cómo se calcula una única región $\tilde{D}_W$. Iteramos sobre la dimensión K y en cada iteración, cada hilo carga una columna (*fragmento a*) $m_W \times 1$ de A y una columna (*fragmento b*) $1 \times n_W$ de B, y el *warp* procede a computar su submatriz de $\tilde{D}$ de tamaño $m_W \times n_W$.

Cada hilo en un *warp* calcula un acumulador de $m_r \times n_r$ dentro de $\tilde{D}$. Para ello, cada hilo carga m_r elementos del *fragmento a*, y n_r elementos del *fragmento b* en sus registros (como se ilustra para el hilo `t2` en la Figura 10), calculando la distancia entre los fragmentos para actualizar el acumulador.

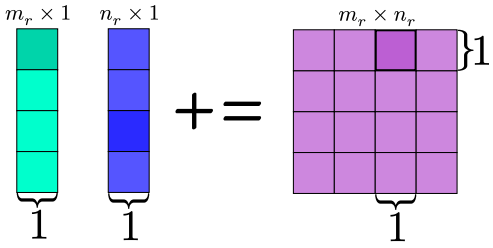


Figura 10: Cálculo del acumulador de $m_r \times n_r$ a nivel de hilo, a partir de los fragmentos $m_r \times 1$ de A y $1 \times n_r$ de B.

Para evitar conflictos de bancos durante la carga de fragmentos desde memoria compartida, se implementa una técnica de *swizzling* en la dimensión k_s. Cada hilo lee 16 bytes contiguos, abarcando 4 bancos consecutivos de memoria compartida. Sin *swizzling*, múltiples hilos de un *warp* accediendo a la misma columna k pero diferentes filas producirán conflictos de bancos de memoria. La solución implementada utiliza la fórmula `k_swizzled = (warp_k + lane_id) % ks`, donde cada hilo accede a un índice $\tilde{k}$ rotado según su índice dentro del *warp*. Esto distribuye los accesos de hilos consecutivos a través de diferentes posiciones en la dimensión k_s, dispersando las solicitudes de memoria entre distintos bancos y eliminado conflictos sin necesidad de *padding* adicional en memoria compartida.

La implementación culmina con cada hilo realizando la escritura de su acumulador de dimensión $m_r \times n_r$ sobre memoria global de forma *coalesced*. Para maximizar el ancho de banda de escritura, cada hilo realiza escrituras vectorizadas, que permite almacenar 128 bits en una única transacción de memoria.

4.1.3. Cómputo de Distancia

Hasta ahora solo hemos hablado de como *acceder* a nuestros datos. Sin embargo, todavía no hemos mencionado como *calcular* la distancia entre individuos.

En principio, la distancia entre dos SNPs podría definirse directamente como

$$\hat{d} : \{0,1,2\}^2 \to \{0,1,4\},\ \hat{d}(a,b) = (a-b)^2$$

No obstante, al compactar nuestros valores en 2 bits, no podemos utilizar de forma directa la fórmula $(a-b)^2$. Para esto, es necesario iterar SNP por SNP, extraer sus 2 bits y luego computar su contribución:

```
s0 = subsecuencia de snps del individuo 0
s1 = subsecuencia de snps del individuo 1
snps = cantidad de snps de la subsecuencias
acc = 0
for (int i = 0; i < snps; i++) {
  a = s0 & 0x3 // extraer snp de s0
  b = s1 & 0x3 // extraer snp de s1

  diff = a - b
  acc += diff * diff

  s0 >>= 2; // shift s0 dos bits a la derecha
  s1 >>= 2; // shift s1 dos bits a la derecha
}
```

Sin embargo, el punto clave es que no necesitamos calcular explícitamente $(a-b)^2$ para cada par de SNPs, sino únicamente detectar el **patrón de diferencia** entre ellos. En lugar de calcular directamente la norma euclidiana al cuadrado, podemos explotar nuestro sistema de codificación $S = \{00_2, 01_2, 10_2\}$. Así, basta con identificar el patrón de bits en el que difiere un par de SNPs y mapear dicho patrón al valor correspondiente de la distancia.

De este modo, la diferencia entre dos SNPs puede determinarse mediante una simple operación a nivel de hardware: el `XOR` bit a bit. Dados dos SNPs $a, b \in S$, se define

$$z = a \oplus b,\ z \in \{00_2, 01_2, 10_2, 11_2\}.$$

A partir del patrón de bits de z, es posible construir una función equivalente a $\hat{d}$ que denominaremos $\tilde{d}$:

$$\tilde{d}(a,b) := \begin{cases} 0 \text{ si } a \oplus b = 00_2 \\ 1 \text{ si } a \oplus b \in \{01_2, 11_2\} \\ 4 \text{ si } a \oplus b = 10_2 \end{cases}$$

De esta manera, se evita explícitamente la operación de resta y el cálculo de cuadrados, reemplazándolos por una comparación sobre patrones de bits. Para eso definimos una *lookup table (LUT)* para mapear cada patrón a su contribución de distancia:

```
lut[4] = {
    0,  // Patrón 00: iguales → distancia 0
    1,  // Patrón 01: diferencia de 1 → distancia 1
```

```
4     4,  // Patrón 10: diferencia de 2 → distancia 4
5     1   // Patrón 11: diferencia de 1 → distancia 1
6 };
```

```
1 acc = 0
2 xor_result = s0 xor s1
3 for (int i = 0; i < snps; i++) {
4   pattern = xor_result & 0x3 // extraer patrón de diferencia
5   acc += lut[pattern]        // usar distancia precomputada
6   xor_result >>= 2;          // shift resultado xor dos bits a la derecha
7 }
```

Esto elimina las restas y multiplicaciones, reemplazándolas por una operación XOR y *lookups* de tabla. Sin embargo, una gran desventaja de esta solución es la gran cantidad de iteraciones que se realizan para comparar dos subcadenas. Al estar cada cadena representada por 32 bits, se realizan 16 iteraciones para comparar dos subsecuencias de 16 SNPs.

Una observación importante es que no necesitamos procesar los SNPs uno por uno, sino que podemos contar cuántos patrones se encuentran en el resultado z y luego multiplicar cada patrón por su contribución. Afortunadamente, existe una instrucción de hardware, `POPC` (*population count*), que nos indica cuántos bits están en 1 en una palabra, que es perfecta para esto.

El razonamiento anterior puede extenderse a secuencias de SNPs completas como se muestra en la Figura 11 posterior. Dadas dos subsecuencias de individuos x e y, su patrón de diferencia se obtiene aplicando el operador `XOR`:

$$z = x \oplus y$$

El resultado z contiene, para cada SNP, el patrón de diferencia bit a bit introducido anteriormente.

La distancia total entre x e y puede expresarse como

$$d(x, y) = \texttt{popc}(p_{01}) + 4 * \texttt{popc}(p_{10}) + \texttt{popc}(p_{11})$$

donde p_{01}, p_{10}, p_{11} son secuencias de $0s$ y $1s$, en las que la aparición de $1s$ indica la presencia del patrón correspondiente al sufijo. Para conseguir estas secuencias se definen tres máscaras auxiliares, s_p, s_i y s_m, donde s_m es una máscara de selección binaria con $1s$ en las posiciones pares (0x555..) y, s_p y s_i, contienen los bits pares e impares de la secuencia z respectivamente, a partir de s_m.

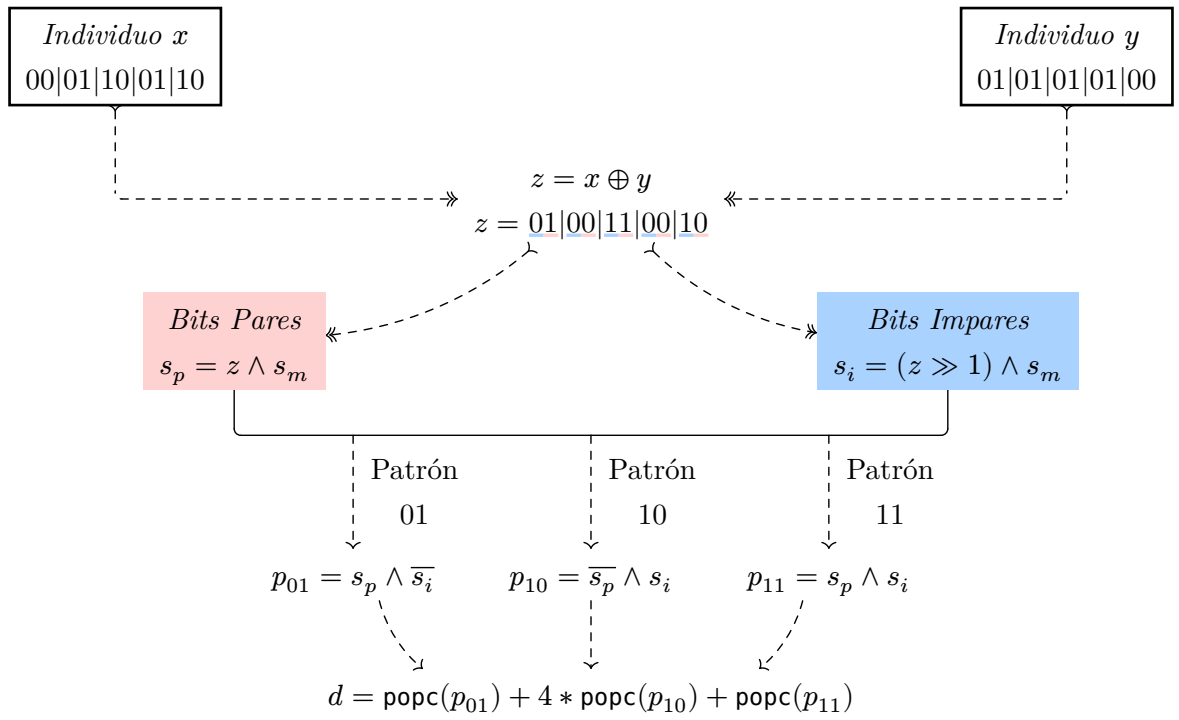


Figura 11: Ejemplo ilustrativo para dos subsecuencias de individuos x e y.

Esta técnica reduce significativamente la cantidad de instrucciones necesarias para realizar el cómputo de distancia entre dos subsecuencias de individuos.

```
1 u32 x = s0 ^ s1                          // resultado xor
2 u32 even_bits  = x & 0x55555555u         // máscara con bits pares
3 u32 odd_bits   = (x >> 1) & 0x55555555u  // máscara con bits impares
4 u32 pattern_01 = even_bits & (~odd_bits) // extraer patrón 01
5 u32 pattern_10 = (~even_bits) & odd_bits // extraer patrón 10
6 u32 pattern_11 = even_bits & odd_bits    // extraer patrón 11
7
8 // acumular diferencia
9 acc += popc(pattern_01) + 4 * popc(pattern_10) + popc(pattern_11)
```

Podemos simplificar la fórmula observando que los patrones son disjuntos y que, además, los patrones p_{01} y p_{11} contribuyen con el mismo peso en la distancia. Esto permite evitar el conteo independiente de bits para cada uno. En su lugar, basta con aplicar la operación `POPC` sobre la expresión $p_{01} \vee p_{11}$ y luego multiplicar el resultado por el peso correspondiente. De esta manera, se cumple que $\texttt{popc}(p_{01}) + \texttt{popc}(p_{11}) = \texttt{popc}(p_{01} \vee p_{11})$.

Si expandimos la expresión

$$p_{01} \vee p_{11} = \left(s_p \wedge \overline{s_i}\right) \vee \left(s_p \wedge s_i\right) = s_p \wedge \left(\overline{s_i} \vee s_i\right) = s_p,$$

vemos que para contabilizar los patrones cuya diferencia generacional es 1 basta con contar los bits activos en las posiciones pares del resultado de la operación `XOR`.

Con esta observación, la función de distancia puede reescribirse como

$$\tilde{d}(x, y) = \texttt{popc}(s_p) + 4 * \texttt{popc}(p_{10}),$$

lo cual reduce la cantidad de operaciones.

```
1 u32 x = s0 ^ s1                          // resultado xor
2 u32 even_bits  = x & 0x55555555u         // máscara con bits pares
3 u32 odd_bits   = (x >> 1) & 0x55555555u  // máscara con bits impares
4 u32 pattern_10 = (~even_bits) & odd_bits // extraer patrón 10
5
6 // acumular diferencia
7 acc += 4 * popc(pattern_10) + popc(even_bits)
```

Es interesante destacar que contamos con estas instrucciones para operandos de 64 bits, por lo tanto, podemos reemplazar `u32` por `u64`, permitiéndonos comparar 32 SNPs con solo 4 operaciones lógicas y 2 instrucciones `POPC64`.

Por último, es posible obtener otra optimización al observar la estructura de los patrones antes de realizar el conteo: los bits pares de cada patrón podrán valer 0 o 1, mientras que los bits impares *siempre* valdrán 0. Esto ocurre al aplicar la máscara de bits pares sobre una palabra, obteniendo una palabra de la forma `0X0X...0X`.

Esto implica que al realizar el conteo de bits en 1 mediante la instrucción `POPC64`, estamos desperdiciando la mitad del potencial de esta instrucción, ya que se sabe de antemano que la mitad de los bits se encuentran en 0. Para explotar al máximo la instrucción, en lugar de procesar solo 32 SNPs con una instrucción `POPC64`, podemos procesar 64 SNPs sin incrementar el número de llamadas a `POPC64`. La idea es que, si consideramos también la siguiente subsecuencia a comparar y aplicamos el mismo enmascaramiento de bits pares, obtenemos otra palabra de la forma `0Y0Y...0Y`. Entonces, al desplazar la primer palabra una posición hacia la izquierda y realizar una operación `OR` con la segunda, cada par de bits contiguos en esta nueva palabra `YXYX...YX` contiene la información combinada de ambas palabras.

```
posición         9   8   7   6   5   4   3   2   1   0
               +---+---+---+---+---+---+---+---+---+---+
bits_lo        | 0 | x | 0 | x | 0 | x | 0 | x | 0 | x |
               +---+---+---+---+---+---+---+---+---+---+

               +---+---+---+---+---+---+---+---+---+---+
bits_hi        | 0 | y | 0 | y | 0 | y | 0 | y | 0 | y |
               +---+---+---+---+---+---+---+---+---+---+
                                   |
                           (bits_hi << 1) | bits_lo
                                   v
               +---+---+---+---+---+---+---+---+---+---+
compactado     | y | x | y | x | y | x | y | x | y | x |
               +---+---+---+---+---+---+---+---+---+---+
```

Esta estrategia intercala ambos patrones, permitiéndonos realizar el mismo cómputo pero utilizando la mitad de las instrucciones `POPC`. Lo que aumenta el *throughput* del cómputo sin incrementar el número de operaciones de conteo de bits.

```
1 u128 s0 = subsecuencia de snps del individuo 0
2 u128 s1 = subsecuencia de snps del individuo 1
```

```
 3
 4 // resultados xor
 5 u64 x_lo = s0.lo ^ s1.lo
 6 u64 x_hi = s0.hi ^ s1.hi
 7
 8 // máscaras de bits pares
 9 u64 even_bits_lo = x_lo & 0x55...5
10 u64 even_bits_hi = x_hi & 0x55...5
11
12 // patrón X1
13 u64 pX1 = even_bits_lo | (even_bits_hi << 1)
14
15 // máscara de bits impares
16 u64 odd_bits_lo = (x_lo >> 1) & 0x55...5;
17 u64 odd_bits_hi = (x_hi >> 1) & 0x55...5;
18
19 // patrón 10
20 u64 p10 = ((~even_bits_lo) & odd_bits_lo) | ((~even_bits_hi) & odd_bits_hi)
21
22 // acumular diferencia
23 acc += 4 * popc(p10) + popc(pX1);
```

Este estrategia es usada por cada hilo al momento de actualizar el acumulador, su matriz $\tilde{D}_{m_r \times n_r}$, a partir de los fragmentos `a` y `b` que se encuentran en sus registros.

```
 1 #pragma unroll
 2 for (int i = 0; i < mr; ++i) {
 3     #pragma unroll
 4     for (int j = 0; j < nr; ++j) {
 5         u128 s0 = fragment_a[i];
 6         u128 s1 = fragment_b[j];
 7
 8         u64 x_lo = s0.x ^ s1.x;
 9         u64 x_hi = s0.y ^ s1.y;
10
11         u64 even_bits_lo = x_lo & MASK;
12         u64 even_bits_hi = x_hi & MASK;
13         u64 pX1 = even_bits_lo | (even_bits_hi << 1);
14         accumulator[i][j] += __popcll(pX1);
15
16         u64 odd_bits_lo = (x_lo >> 1) & MASK;
17         u64 odd_bits_hi = (x_hi >> 1) & MASK;
18         u64 p10 = ((~even_bits_lo) & odd_bits_lo) | ((~even_bits_hi) &
   odd_bits_hi);
19         accumulator[i][j] += 4 * __popcll(p10);
20     }
21 }
```

Esta versión final procesa dos palabras de 64 bits (128 bits totales = 64 SNPs) con una cantidad escasa de operaciones lógicas y únicamente dos instrucciones `POPC`.

Comparado con la versión anterior, que requería 64 extracciones, 64 restas, 64 multiplicaciones y 64 sumas, la reducción es dramática, logrando transformar operaciones aritméticas

complejas en simples manipulaciones de bits que las GPUs (y CPUs modernas) pueden ejecutar extremadamente rápido.

4.1.4. Implementación para CPU

Esta estrategia de resolución puede ser aplicada de la misma forma sobre arquitecturas CPU, utilizando técnicas avanzadas de computación de alto desempeño inspiradas en el diseño de bibliotecas BLAS modernas como OpenBLAS. La implementación explota exhaustivamente la jerarquía de memoria mediante bloques de caché multinivel con cinco bucles anidados. La Figura 12, muestra cómo el bucle más externo itera sobre la dimensión m, particionando la matriz $D_{m\times m}$ en bloques columna D_j de dimensiones $m \times n_c$ y la matriz $X_{m\times k}$ en bloques B_j de dimensiones $n_c \times k$.

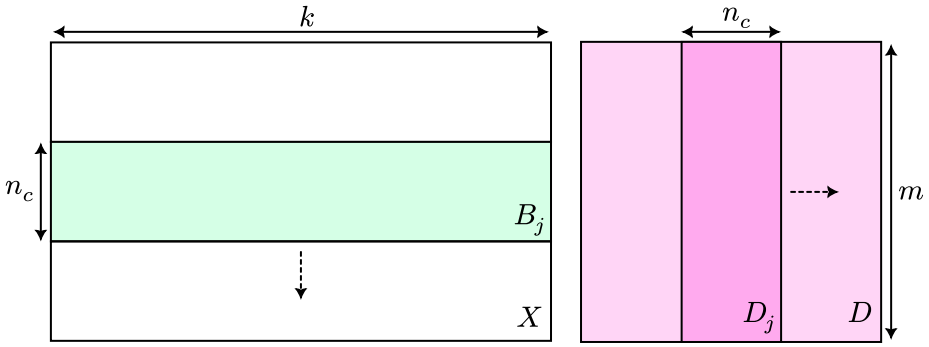


Figura 12: Bucle 5, división de las matrices X y D en bloques B_j y D_j.

En la Figura 13, se observa el cuarto nivel, iterando sobre la dimensión k, particionando B_j en bloques B_p de dimensiones $n_c \times k_c$ de un tamaño adecuado para residir en caché L3.

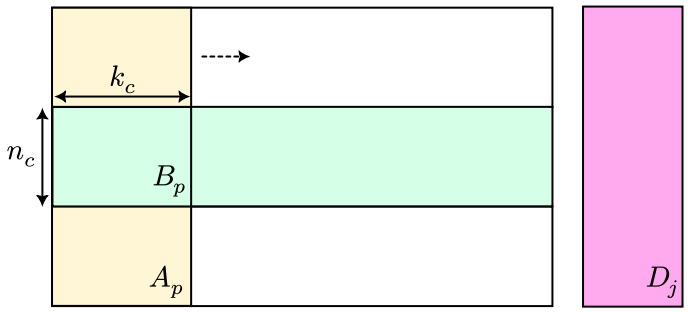


Figura 13: Bucle 4, división del bloque B_j en bloques B_p y la matriz X en bloques A_p.

El bloque B_p se reorganiza en un bloque $\tilde{B}_p$ en formato *column-major* que reside completamente en memoria caché L3, lo que permite eliminar *strides*, permitiendo acceso secuencial a memoria en las etapas posteriores.

En la Figura 14, se observa el tercer nivel, iterando sobre la dimensión m, particionando los bloques D_j y A_p en bloques D_i de dimensiones $m_c \times n_c$ y A_i de dimensiones $m_c \times k_c$ respectivamente, ambos diseñados para residir en caché L2.

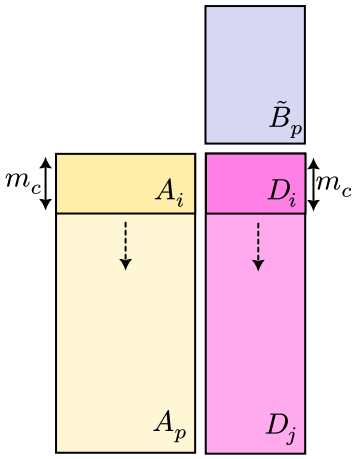


Figura 14: Bucle 3, división de los bloques D_j y A_p en bloques D_i y A_i respectivamente junto con la reorganización de B_p.

Al igual que B_p, el bloque A_i se reorganiza en un bloque $\tilde{A}_i$ en formato *column-major*. En este caso, el bloque reside completamente en memoria caché L2, lo que permite eliminar *strides*, permitiendo acceso secuencial a memoria en las etapas posteriores.

En la Figura 15, se itera sobre la dimensión n_c de $\tilde{B}_p$, cargando un panel de dimensiones $n_r \times k_c$ en memoria caché L1, siendo utilizado para actualizar el acumulador de D de dimensión $m_c \times n_r$ a partir de la matriz $\tilde{A}_i$.

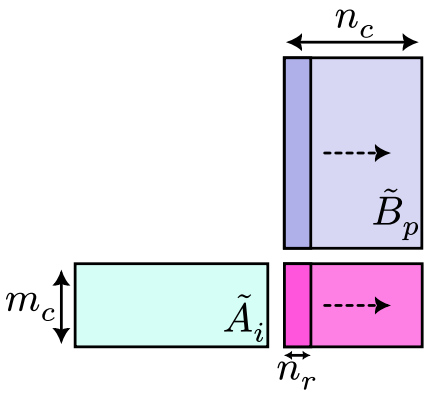


Figura 15: Bucle 2, división de $\tilde{B}_p$ en paneles de dimensión $n_r \times k_c$.

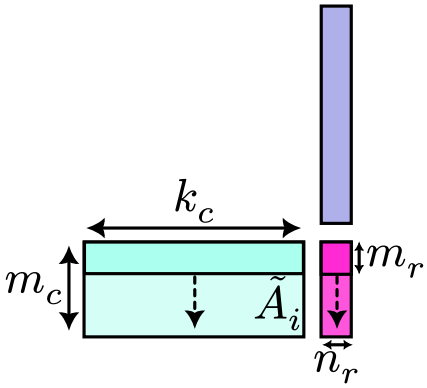


Figura 16: Bucle 1, división de $\tilde{A}_i$ en paneles de dimensión $m_r \times k_c$.

En la Figura 16, se itera sobre la dimensión m_c de $\tilde{A}_i$, cargando un panel de dimensiones $m_r \times k_c$, para calcular la distancia entre el panel de $\tilde{B}_p$ cargado anteriormente. Este cálculo se realiza mediante un *micro-kernel* el cual se detalla a continuación, siendo este el núcleo computacional de la implementación.

En la Figura 17, se observa el comportamiento del *micro-kernel*, que a partir de los paneles de $\tilde{B}_p$ y $\tilde{A}_i$ de dimensiones $n_r \times k_c$ y $m_r \times k_c$, se computa la matriz de distancia entre ambos paneles, residiendo completamente en registros. Para esto, se realizan k_c iteraciones, en cada iteración, se carga una columna de cada panel y se procede a actualizar el acumulador. Al reorganizar ambos paneles en *column-major*, cada iteración accede a datos contiguos en memoria, aprovechando al máximo la línea de caché de la arquitectura y reduciendo la latencia durante la actualización del acumulador.

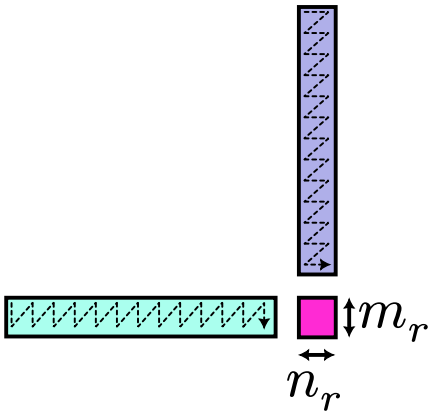


Figura 17: Trabajo realizado por el *micro-kernel* para el cómputo de la matriz de distancia entre dos columnas de dimensiones m_r y n_r.

En arquitecturas *multicore*, podemos utilizar múltiples hilos para paralelizar los bucles introducidos. En especial, realizamos una paralelización sobre los bucles 1 y 2 que iteran alrededor del *micro-kernel* mediante la directiva `#pragma omp for collapse(2)` de OpenMP. Esto nos permite realizar una paralelización del problema, sin entrar en conflictos entre hilos al momento de realizar la escritura de sus acumuladores en memoria principal. Además, se agrega la directiva `schedule(dynamic)`, puesto que la carga de trabajo por hilo varía en algunas ocasiones, por ejemplo, cuando se calculan acumuladores triangulares. Por otra parte, esta metodología de paralelismo nos permite paralelizar la reorganización de los bloques $\tilde{B}_p$ y $\tilde{A}_i$, siendo el bloque $\tilde{B}_p$ compartido entre los hilos mediante la memoria caché L3, y el bloque $\tilde{A}_i$ alojado en la memoria caché L1 de cada hilo.

Por último, cabe destacar que las operaciones bit a bit suelen ser vectorizadas por el compilador mediante SIMD en la gran mayoría de las CPU modernas que soportan la ISA[2] correspondiente. Por otra parte, la operación *population count* suele estar disponible en forma vectorizada en algunos conjuntos de instrucciones, como AVX-512 mediante `VPOPCNTB`, lo que permite operar con vectores de hasta 512 bits.

4.2. Resolución del Problema de Caminos Mínimos

El problema que queremos resolver es el de caminos mínimos entre todos los pares de nodos *(All-Pairs Shortest Paths, APSP)*. En este caso, al contar con una matriz densa, resulta conveniente utilizar un algoritmo cuya complejidad no dependa del número de aristas del grafo, sino únicamente de la cantidad de nodos.

Por esta razón, y teniendo en cuenta las ventajas prácticas reportadas en Anjary (2023) respecto del algoritmo R-Kleene, adoptamos dicho enfoque desarrollando una versión optimizada para GPU. Nuestra contribución combina dos aspectos fundamentales: (i) una reformulación del algoritmo original, que reduce operaciones redundantes y explota simetrías estructurales, y (ii) el uso de la biblioteca cuASR para acelerar productos de matrices en el semianillo *min-plus*, evitando la necesidad de programar *kernels* manuales en CUDA.

[2]ISA (*Instruction set architecture*): interfaz abstracta entre el hardware y el software que define el conjunto de instrucciones que un procesador puede ejecutar.

El algoritmo original de R-Kleene actualiza los bloques de la matriz de distancias de acuerdo con las siguientes ecuaciones, donde el producto y la suma deben interpretarse en el semianillo *min-plus*:

$$B += A \times B \tag{1}$$

$$C += C \times A \tag{2}$$

$$D += C \times B \tag{3}$$

$$B += B \times D \tag{4}$$

$$C += D \times C \tag{5}$$

$$A += B \times C \tag{6}$$

En nuestra implementación realizamos tres modificaciones clave:

1. En un grafo de distancias donde la diagonal de la matriz es nula, en la actualización $B += A \times B$, el término B ya está implícitamente contenido en el producto $A \times B$. Por lo tanto, el operador "+ =" no agrega información adicional y puede reemplazarse por una asignación directa "=".

   A continuación se presenta una prueba formal. Recordando que $B += A \times B$ se obtiene a partir de $b_{ij} += \sum_{k=0}^{n-1} a_{ik} * b_{kj}$, si consideramos el caso particular $i = k$, en un momento la expresión $b_{ij} = \min\left(b_{ij}, \sum_{k=0}^{n-1} a_{ik} * b_{kj}\right)$ valdrá $b_{ij} = \min(b_{ij}, a_{ii} + b_{ij}, ...)$. Puesto que A es una matriz con diagonal nula, $a_{ii} = 0$, demostrando así que el término $\sum_{k=0}^{n-1} a_{ik} * b_{kj} \leq b_{ij}$. *Observación: lo mismo ocurre para los casos* $C += C \times A$, $B += B \times D$ *y* $C += D \times C$.

2. Debido a que los bloques B y C son transpuestos entre sí $(C = B^T)$, es suficiente con calcular uno de ellos y derivar el otro.

3. Reescribimos las actualizaciones en términos de productos del tipo XY^T, que puede aprovechar directamente los *kernels* optimizados de cuASR, logrando eficiencia en la ejecución sobre GPU como se explora en Su et al. (2025).

Para implementar los productos de matrices en el semianillo *min-plus*, recurrimos a cuASR, biblioteca basada en CUTLASS que implementa productos de matrices sobre semianillos algebraicos SRGEMM (Semiring General Matrix Multiplication), incluyendo el tropical semianillo $(\min, +)$.

Gracias a esta integración, la implementación mantiene la corrección matemática del algoritmo, evitando la necesidad de programar *kernels* manualmente y simplificando el mantenimiento del código. Cada producto matricial de R-Kleene se realiza ahora mediante cuASR, lo que garantiza que las distancias mínimas entre todos los pares de nodos se calculen de forma eficiente y paralela, aprovechando la simetría y las optimizaciones introducidas en la reformulación algebraica del algoritmo.

En la Figura 9 se presenta el pseudocódigo de nuestra implementación final de R-Kleene en GPU. La figura ilustra la estructura recursiva del algoritmo, la aplicación de los productos *min-plus* sobre los bloques de la matriz de distancias y la forma en que la simetría y la reorganización de los productos se traducen en operaciones eficientes sobre GPU.

R-Kleene+OPT(W)

/* $W = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ */

1 A = R-Kleene+OPT(A)
2 B = minplus-cuASR(A, C^T)
3 $C = B^T$
4 D += minplus-cuASR(C, C^T)

5 D = R-Kleene+OPT(D)
6 B = minplus-cuASR(B, D)
7 $C = B^T$
8 A += minplus-cuASR(B, B^T)

Figura 9: Pseudocódigo del algoritmo R-Kleene+OPT.

Capítulo 5

Evaluación Experimental

En este capítulo se presentan los resultados de la evaluación experimental de nuestro trabajo, realizada con el objetivo de validar la eficacia y eficiencia de las estrategias implementadas. Durante esta evaluación se calcularán y analizarán métricas tales como tiempo de ejecución, *speedup*, escalabilidad y *throughput*.

Todas las pruebas se llevaron a cabo en ClusterUY (Nesmachnow & Iturriaga, 2019), infraestructura de cómputo de alto rendimiento que proporciona los recursos necesarios para ejecutar experimentos a gran escala.

Para las pruebas se empleó un servidor equipado con un procesador AMD EPYC 7763 que cuenta con 128 hilos, 256 GB de RAM y GPUs NVIDIA A40 de 48 GB.

5.1. Configuración Paramétrica

Los algoritmos desarrollados incorporan diversos parámetros configurables. Estos fueron ajustados para aprovechar al máximo las características del hardware empleado en las evaluaciones experimentales. En particular, se seleccionaron tamaños de bloque que permiten explotar la jerarquía de memoria, evitar conflictos entre bancos y optimizar el acceso a los datos. Además, estos parámetros fueron fijados de forma explícita en el código, lo que facilita que el compilador aplique optimizaciones agresivas, como *loop unrolling* y otras transformaciones que requieren valores conocidos en tiempo de compilación.

Para determinar la mejor configuración paramétrica en GPU se realizó un test de Friedman sobre 30 configuraciones distintas, con el objetivo de evaluar si las diferencias observadas entre las configuraciones son estadísticamente significativas. La Tabla 3 presenta la mejor configuración en GPU, y será la utilizada en las evaluaciones a continuación.

m_s	n_s	k_s	n_w	m_w	m_r	n_r
64	32	8	32	16	4	4

Tabla 3: Mejor configuración paramétrica en GPU.

Asimismo, se realiza la configuración paramétrica del algoritmo en CPU sobre 30 configuraciones distintas. La Tabla 4 presenta la mejor configuración en CPU obtenida en todas las instancias de calibración, y será la utilizada en las evaluaciones a continuación.

m_r	n_r	k_c	m_c	n_c
16	8	2K	1K	64K

Tabla 4: Mejor configuración paramétrica en CPU.

5.2. Generación de Instancias

Como se ha mencionado a lo largo del trabajo, las instancias del problema dependen del número de individuos y de SNPs. Por esta razón, para la evaluación se generarán distintas instancias de manera aleatoria. La generación aleatoria de datos no representa un riesgo para la validez de los experimentos, ya que nuestros algoritmos no dependen de valores específicos de genotipos. Además, esta aproximación se justifica por la ausencia de conjuntos de datos reales con dimensiones tan grandes como las que consideramos, lo que hace imprescindible recurrir a instancias sintéticas para poder realizar pruebas representativas.

Instancia	**Cantidad de Individuos (N_I)**	**Cantidad de SNPs (N_S)**
Sintética 1	4.000	80.000.000
Sintética 2	30.000	600.000.000
Sintética 3	2.000	3.000.000.000

Tabla 5: Instancias sintéticas del problema a evaluar.

Las Figuras 18, 19 y 20 muestran la distribución porcentual del tiempo de ejecución entre las distintas etapas del cálculo de la matriz de distancias para las tres instancias sintéticas consideradas.

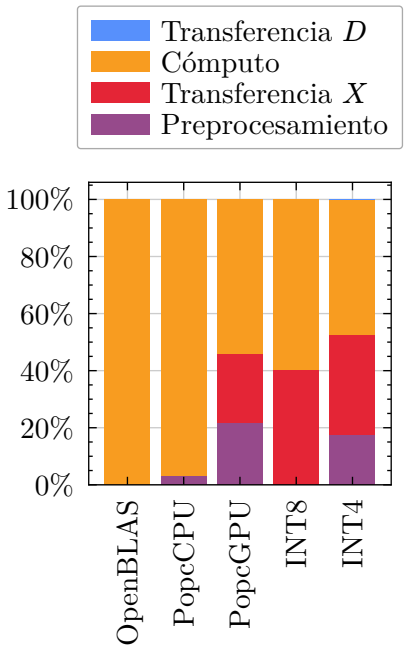


Figura 18: Sintética 1.

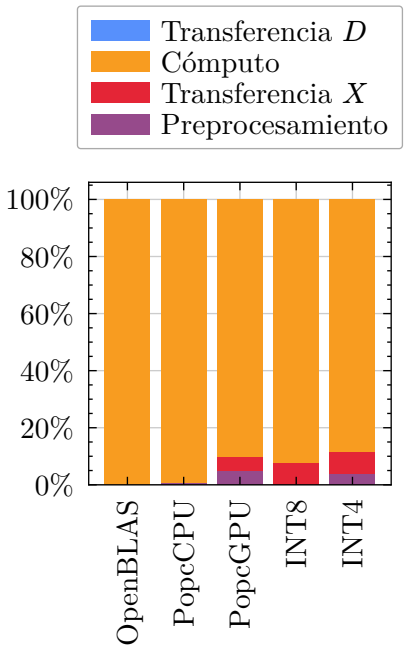


Figura 19: Sintética 2.

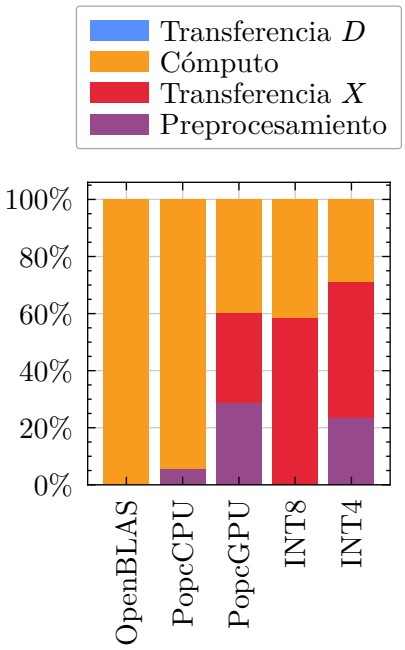


Figura 20: Sintética 3.

El gráfico de barras descompone el tiempo en transferencia de datos, preprocesamiento y cómputo, representando la proporción relativa de cada una de estas etapas para facilitar la comparación. A partir de estos datos, se observa cómo varía el peso relativo de cada etapa según la implementación y el tamaño del problema. Para instancias con poca cantidad de individuos, se observa que en las variantes de GPU, la transferencia de datos hacia la GPU empieza a tener un peso importante, limitando parcialmente el desempeño de la implementación, mientras que el preprocesamiento es despreciable para matrices relativamente grandes.

5.3. Escalabilidad

Por lo discutido en el Capítulo 2, el problema presenta un grado de paralelismo muy alto y las necesidades de sincronización entre hilos son prácticamente despreciables. Sobre esta base, se llevó a cabo una evaluación de la escalabilidad al incrementar los recursos de cómputo, en particular, el número de hilos para las implementaciones sobre CPU. La Figura 21 muestra estos resultados para las tres instancias, resaltando con colores más claros la desviación estándar, y en color más oscuro, las líneas delgadas ilustran la media.

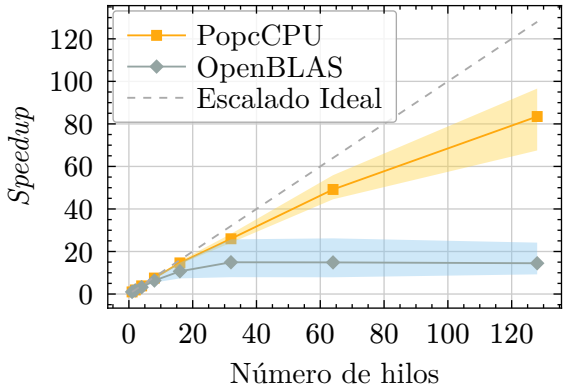


Figura 21: *Speedup* algorítmico obtenido al aumentar la cantidad de hilos en implementaciones sobre CPU.

Tal como se observa, las dos implementaciones sobre CPU presentan comportamientos de escalabilidad claramente diferentes. En el caso de OpenBLAS, el *speedup* crece en las primeras decenas de hilos, pero luego se estabiliza rápidamente, alcanzando un valor cercano a $\times 10$, lo que indica una pérdida progresiva de eficiencia al aumentar el paralelismo. En contraste, la implementación POPC muestra un escalado casi lineal en un amplio rango de hilos, aproximándose al comportamiento ideal, lo que refleja un aprovechamiento mucho más efectivo de los recursos de cómputo disponibles.

Para evaluar el rendimiento de las implementaciones desarrolladas a lo largo de un espacio exhaustivo de parámetros, se ejecutó una batería de experimentos sobre instancias sintéticas variando tanto el número de individuos N_I como el número de marcadores SNP. Cada combinación de parámetros fue evaluada múltiples veces para garantizar la estabilidad de las mediciones, registrando el tiempo total de ejecución y calculando la cantidad de comparaciones de SNPs por segundo de cada implementación. Dado el volumen de datos generados y la naturaleza bidimensional del espacio de parámetros, se optó por utilizar mapas de calor como herramienta de visualización, lo que permite identificar rápidamente las regiones del espacio donde cada implementación alcanza su máxima eficiencia, así como detectar patrones de comportamiento y limitaciones inherentes a cada enfoque. Las Figuras 22 a 26 presentan estos mapas de calor, donde cada píxel representa una instancia del problema caracterizada por sus dimensiones (N_I, N_S), y el color indica el rendimiento en GOPS obtenido en una escala logarítmica.

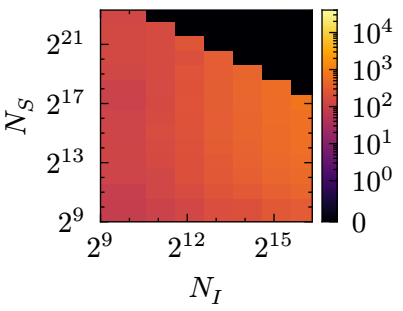


Figura 22: OpenBLAS.

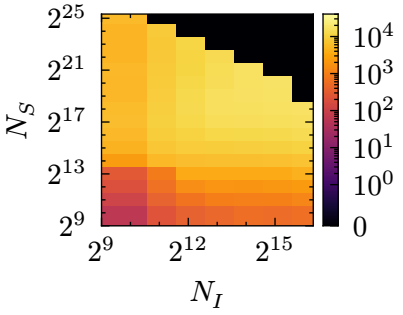


Figura 23: INT8.

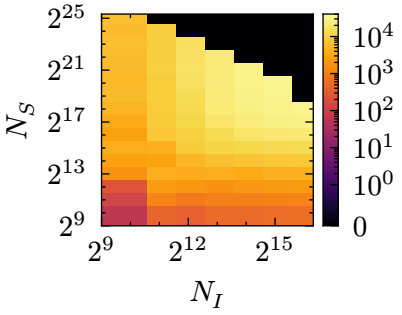


Figura 24: INT4.

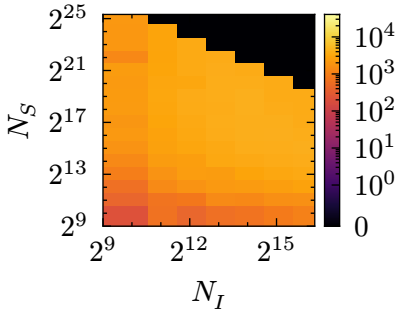


Figura 25: PopcCPU.

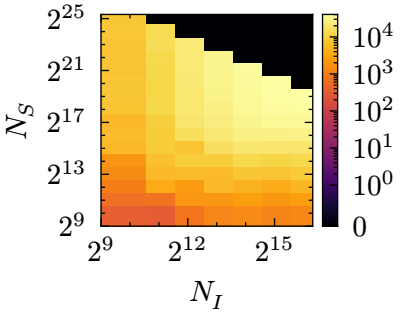


Figura 26: PopcGPU.

La escala de colores, desde tonos oscuros (bajo rendimiento) hasta amarillos brillantes (alto rendimiento), revela patrones distintivos para cada enfoque. OpenBLAS (Figura 22) muestra el rendimiento más limitado, con regiones oscuras dominando el mapa y mejoras graduales solo en problemas de gran escala. PopcCPU (Figura 25) presenta una mejora sustancial gracias a las operaciones bit a bit, exhibiendo regiones más iluminadas y estables, *throughput* significativamente mejor OpenBLAS en cualquier dimensión del problema.

Se observa claramente que, de las implementaciones basadas en Tensor Cores, la implementación con tensores INT4 (Figura 24) es superior a la de tensores INT8 (Figura 23), mostrando esta última un mejor rendimiento debido al mayor *throughput* teórico. PopcGPU (Figura 26) presenta un rendimiento comparable a la de Tensor Cores INT4 en la mayoría del espacio de parámetros, demostrando la eficacia de nuestra implementación y la compresión de datos.

Un patrón notable presente en la mayoría de las implementaciones es la degradación del *throughput* para valores pequeños de N_I, observable como regiones más oscuras en el mapa de calor. Este fenómeno se atribuye a la subutilización del hardware. Cuando el número de individuos es reducido, no se generan suficientes bloques de trabajo para saturar completamente los recursos de cómputo disponibles, dejando recursos ociosos. Para mitigar este problema en escenarios con N_I pequeños pero N_S grande, se pueden aplicar técnicas de paralelización avanzadas como *split-k* (NVIDIA Corporation, s. f.-b), que descomponen la reducción a lo largo de la dimensión N_S en múltiples bloques independientes, permitiendo distribuir el trabajo no solo sobre el eje de individuos N_I, sino también sobre el eje de SNPs N_S, mejorando la utilización del hardware disponible.

La región triangular negra en todas las figuras representa combinaciones que exceden las restricciones de memoria en el dispositivo, siendo más restrictiva para las implementaciones sin compresión, como es el caso de OpenBLAS, donde se utilizan 4 bytes para representar cada

SNPs, lo que evidencia el beneficio de las estrategias de compresión al momento de resolver grandes instancias del problema.

Por otra parte, la Figura 27, en escala logarítmica, compara los tiempos de ejecución de cinco estrategias distintas para resolver el problema de caminos mínimos tanto en CPU como en GPU. Las implementaciones en CPU, denominadas SciPy y graph-tool, corresponden, respectivamente, a la línea base establecida en el Capítulo 3 y a la implementación de `shortest_distance` de graph-tool (*Graph-Tool documentation*, 2025), la cual se basa en el algoritmo de Floyd-Warshall. Por otro lado, las implementaciones evaluadas en GPU fueron FW-GPU, el algoritmo R-Kleene introducido en el Capítulo 2, y la versión optimizada de R-Kleene propuesta en el Capítulo 4, RK+OPT. Estas últimas implementaciones hacen uso de las primitivas ofrecidas por la biblioteca cuASR. La implementación FW-GPU de Floyd–Warshall incorpora ciertas modificaciones respecto al algoritmo original. En particular, realiza múltiples productos *min–plus* sucesivos hasta que converge al resultado, tal como se propone en Anjary (2023).

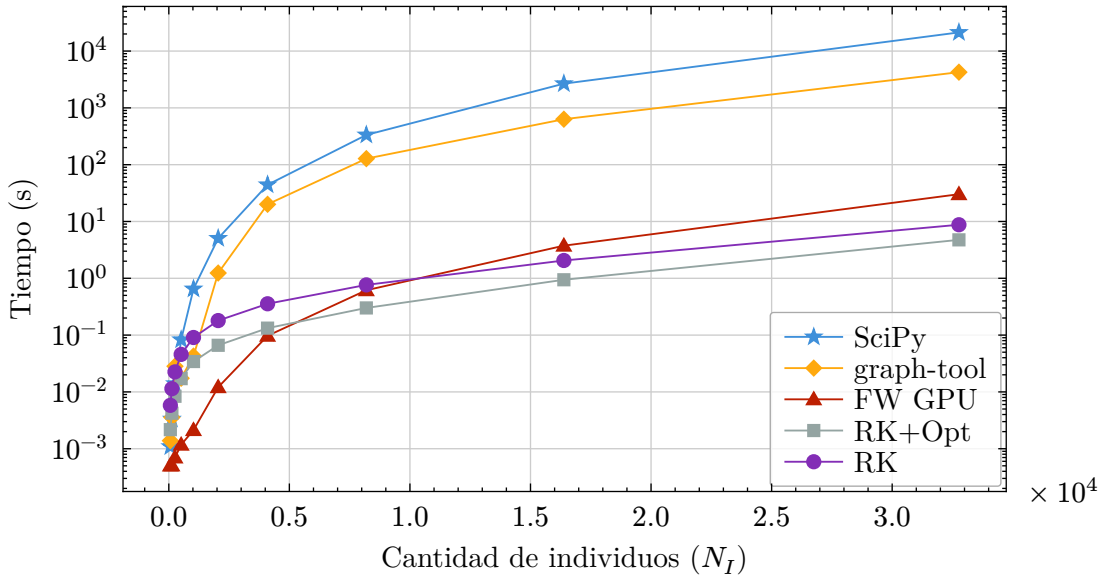


Figura 27: Tiempos de ejecución para los algoritmos APSP en función de la cantidad de individuos.

En la gráfica puede observarse que ambas implementaciones en CPU evidencian el comportamiento cúbico característico del algoritmo Floyd–Warshall. En el caso de graph-tool, los tiempos se mantienen competitivos para tamaños pequeños y medianos, siendo insignificantes para matrices con menos de 1024 individuos. No obstante, a partir de 2048 el crecimiento se vuelve abrupto, donde el tiempo se eleva de 1.2s a 20.1s en 4096, alcanzando 127.8s en 8192.

La implementación de SciPy, muestra un incremento aún más pronunciado, donde para 4096 individuos alcanza aproximadamente el doble que graph-tool, y para 8192 individuos asciende hasta 333.9s. Aunque se trata de una implementación ampliamente utilizada, su rendimiento queda claramente por detrás de las alternativas en GPU, incluso en tamaños moderados.

Por su parte, Floyd-Warshall en GPU es competente para matrices pequeñas y medianas, superando en algunos casos a las implementaciones de R-Kleene. Sin embargo, a medida que aumenta el tamaño de la matriz, el tiempo comienza a crecer pronunciadamente, lo que refleja

la desventaja mencionada en el Capítulo 2, donde la operación se repite múltiples veces hasta lograr convergencia, pudiendo requerir más pasos a medida que el tamaño aumenta.

En contraste, las implementaciones de R-Kleene muestran un crecimiento temporal moderado, donde la versión optimizada consigue mejoras significativas.

La Tabla 6 resume los tiempos de ejecución en formato `hh:mm:ss`, obtenidos para cada algoritmo evaluado, junto con su *speedup* relativo respecto de la línea base correspondiente.

Algoritmo	**Instancia 1**		**Instancia 2**		**Instancia 3**	
	Tiempo	Speedup	Tiempo	Speedup	Tiempo	Speedup
OpenBLAS	0:46:59	×1.00	163:03:35	×1.00	11:15:32	×1.00
PopcCPU	0:03:23	×13.88	26:38:47	×6.11	0:37:00	×18.25
PopcGPU	0:00:26	×105.94	2:05:35	×77.90	0:06:19	×106.82
INT8	0:01:02	×45.20	5:28:36	×29.77	0:13:31	×49.96
INT4	0:00:38	×72.63	2:48:25	×58.08	0:09:32	×70.77
SciPy	0:00:44	×1.00	5:54:58	×1.00	0:00:05	×1.00
graph-tool	0:00:20	×2.20	1:10:50	×5.01	0:00:01	×4.07
FW-GPU	0:00:00	×455.27	0:00:30	×704.68	0:00:00	×417.30
RK	0:00:00	×124.36	0:00:08	×2431.04	0:00:00	×28.02
RK+Opt	0:00:00	×333.81	0:00:04	×4482.53	0:00:00	×76.33

Tabla 6: *Speedup* de las distintas implementaciones sobre su respectiva linea base.

En el caso del cálculo de la matriz de distancias, todas las implementaciones presentan mejoras notables respecto de la línea base OpenBLAS. Por otra parte, la performance del algoritmo PopcCPU se degrada en la segunda instancia, que presenta una gran cantidad de individuos. Por otro lado, el *speedup* obtenido en las implementaciones de Tensor Cores coincide con lo esperado: la implementación INT4 obtiene aproximadamente el doble de *speedup* en comparación a INT8 en todas las instancias. Sin embargo, nuestro algoritmo en GPU supera ampliamente cualquiera de las implementaciones de Tensor Cores en todas las instancias evaluadas.

En la segunda etapa del problema, la implementación FW-GPU constituye la contribución más destacada en las instancias con menor cantidad de individuos, reflejando que FW posee un gran potencial y puede ser una alternativa viable cuando el tamaño poblacional es reducido. Por otro lado, la implementación RK+Opt supera al resto de las implementaciones en la instancia del problema más grande, logrando un *speedup* de ×4482.53 frente al *speedup* ×704.64 de la implementación FW-GPU. Además, es evidente que las optimizaciones aplicadas en la implementación RK+Opt son notorias al compararse frente a la implementación clásica RK, logrando duplicar la performance del algoritmo en todas las instancias. Por otra parte, en las implementaciones de CPU, la biblioteca graph-tool supera ampliamente a la biblioteca SciPy en su implementación del algoritmo Floyd-Warshall.

En todos los casos, las versiones en GPU superan notoriamente a las versiones en CPU, lo cual era esperable.

Capítulo 6

Conclusiones

En este trabajo se desarrollaron soluciones que superan a los algoritmos disponibles para el cálculo de distancias y la resolución del problema de caminos mínimos en grafos. Las implementaciones propuestas explotan de manera explícita las propiedades algebraicas y estructurales del problema, permitiendo formular operaciones más eficientes y compatibles con rutinas altamente optimizadas en CPU y GPU. Este enfoque se ve potenciado por la aplicación de técnicas de programación paralela, que habilitan un aprovechamiento profundo del paralelismo inherente a las arquitecturas modernas y permiten mantener un rendimiento elevado incluso frente a instancias de gran escala.

La combinación de operaciones bit a bit junto a la operación *population count* para el cálculo de distancias, y el algoritmo R-Kleene junto a las optimizaciones aplicadas para la etapa de resolución de caminos mínimos, conforma un proceso optimizado de extremo a extremo. Este diseño no solo mejora la eficiencia tanto a nivel de cómputo como de memoria, sino que también permite la ejecución de este proceso en una amplia gama de dispositivos, a diferencia del uso de Tensor Cores que se encuentra disponible en un conjunto reducido de GPUs.

Los resultados experimentales confirman aceleraciones de varios órdenes de magnitud respecto de las líneas base tradicionales, validando la efectividad del enfoque.

Finalmente, este trabajo abre nuevas oportunidades en dominios donde los problemas de similitud y las distancias basadas en grafos son esenciales. En particular, las técnicas aquí desarrolladas resultan especialmente relevantes para líneas de investigación aplicadas a bioinformática, como aquellas que actualmente llevan adelante las docentes del IMERL, donde la manipulación de grandes volúmenes de datos biológicos y el análisis de estructuras complejas demandan métodos altamente optimizados y escalables. Estas contribuciones sientan así una base sólida para futuras extensiones, tanto hacia variantes híbridas CPU–GPU como hacia aplicaciones específicas.

6.1. Trabajo Futuro

El primer paso a realizar es el desarrollo de una biblioteca de alto nivel que exponga los algoritmos desarrollados a lo largo del trabajo. Esto permitiría a la comunidad científica utilizar estas implementaciones en futuros trabajos de manera más amigable.

A pesar de los buenos resultados conseguidos en arquitecturas GPU, la transferencia de la matriz de individuos suele degradar considerablemente el tiempo de ejecución total del algoritmo. Para mejorar estas transferencias, que son aún más notables en procesamiento por lotes, es interesante explorar técnicas como *pinned memory*, esto es, en lugar de usar memoria paginable al representar a los individuos, utilizar no paginable, mejorando así los tiempos de

transferencia. Por otra parte, explorar tecnologías como NVLink (NVIDIA Corporation, s. f.-c) para interconectar los diferentes dispositivos a emplear, podría aumentar la eficiencia de las transferencias al trabajar con grandes volúmenes de datos.

Aunque nuestro algoritmo para el cálculo de distancias euclidianas es prometedor, presenta inconvenientes al trabajar con conjuntos de datos donde la cantidad de individuos es reducida, debido a la forma en que se descompone el problema en pequeños subproblemas, llegando a casos donde el dispositivo no se sature por completo. Entre las opciones exploradas, se destaca la utilización de heurísticas para saber en qué dimensiones realizar la descomposición del problema. Por ejemplo, en caso de que la cantidad de individuos sea reducida, se podría tener implementaciones alternativas que paralelicen el eje de SNPs. Además, se podrían implementar diferentes algoritmos que exploten las características de cada arquitectura de hardware, siguiendo los principios de bibliotecas de alto desempeño como cuBLAS y OpenBLAS, donde a partir de heurísticas, seleccionan el algoritmo óptimo para atacar el problema.

Por último, la implementación de estos algoritmos sobre arquitecturas híbridas puede realizarse naturalmente mediante bibliotecas de pasaje de mensajes como es el caso de MPI, pudiendo explotar todos los recursos de cómputo disponibles por el usuario.

Apéndice

A.1 Ejemplo ilustrativo del Problema

Consideremos un ejemplo sencillo con un conjunto de tres individuos, cada uno representado por dos SNPs:

$$X = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 1 \end{pmatrix}$$

1. Cálculo de distancias

La distancia entre dos individuos se define en general como una función métrica que mide la disimilitud entre ambos. Para este ejemplo usaremos una distancia habitual en espacios vectoriales, lo que nos permite obtener la siguiente matriz de distancias aplicada al conjunto X:

$$D = \begin{pmatrix} 0 & 1 & 2.24 \\ 1 & 0 & 1.41 \\ 2.24 & 1.41 & 0 \end{pmatrix}$$

2. Construcción del grafo de Fermat con $\alpha = 2$

De acuerdo a la definición del estimador de la distancia de Fermat, los pesos de las aristas se obtienen elevando las distancias a la potencia α. Para $\alpha = 2$:

$$w_2(x_1, x_2) = (1)^2 = 1, \quad w_2(x_2, x_3) = (1.41)^2 \approx 2, \quad w_2(x_1, x_3) = (2.24)^2 \approx 5$$

De esta forma, la matriz de pesos del grafo es:

$$W_2 = \begin{pmatrix} 0 & 1 & 5 \\ 1 & 0 & 2 \\ 5 & 2 & 0 \end{pmatrix}$$

2.1 Caminos mínimos (APSP)

Al aplicar un algoritmo de caminos mínimos sobre W_2, se observa que:

- El costo directo entre x_1 y x_3 es 5.
- El costo alternativo a través de x_2 es $w_2(x_1, x_2) + w_2(x_2, x_3) = 1 + 2 = 3$.

En este caso, el camino indirecto resulta ser más corto. El grafo de Fermat con $\alpha = 2$ queda entonces:

$$G_2 = \begin{pmatrix} 0 & 1 & 3 \\ 1 & 0 & 2 \\ 3 & 2 & 0 \end{pmatrix}$$

A.2 Ejemplo de ejecución del algoritmo R-Kleene

Consideremos la matriz de distancias obtenida en el paso 2 de Apéndice A.1 entre 3 nodos:

$$H = \begin{pmatrix} 0 & 1 & 5 \\ 1 & 0 & 2 \\ 5 & 2 & 0 \end{pmatrix}$$

1. División de la matriz

Se divide la matriz por la mitad, utilizando división entera:

$$H = \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} 0 \end{pmatrix} & \begin{pmatrix} 1 & 5 \end{pmatrix} \\ \begin{pmatrix} 1 \\ 5 \end{pmatrix} & \begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix} \end{pmatrix}$$

2. Recursión sobre A

Llamamos recursivamente a R-Kleene sobre el bloque A, y resolvemos los caminos mínimos entre el nodo 0. Puesto que A es de dimensión 1, entramos a un caso base donde no hacemos nada.

3. Actualización de B y C

Calculamos:

$$B = \min(B, \text{minplus}(A, B))$$

$$C = \min(C, \text{minplus}(C, A))$$

Aplicando la operación *minplus*:

$$B_{i,j} = \min_k(B_{ij}, A_{i,k} + B_{k,j}), \quad C_{i,j} = \min_k(C_{ij}, C_{i,k} + A_{k,j})$$

Obteniendo:

$$B = (\min(1, 0+1) \;\; \min(5, 0+5)) = (1 \;\; 5)$$

$$C = \begin{pmatrix} \min(1, 0+1) \\ \min(5, 0+5) \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \end{pmatrix}$$

sin ningun cambio.

4. Actualización de D

$$D \leftarrow \min(D, \text{minplus}(C, B))$$

Computamos $\text{minplus}(C, B) = \begin{pmatrix} \min(1+1) & \min(1+5) \\ \min(5+1) & \min(1+1) \end{pmatrix} = \begin{pmatrix} 2 & 6 \\ 6 & 10 \end{pmatrix}$

Ahora tomamos el mínimo elemento a elemento con D:

$$D = \min\left(\begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix}, \begin{pmatrix} 2 & 6 \\ 6 & 10 \end{pmatrix}\right) = \begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix}$$

sin cambios en D.

5. Recursión sobre D

Como D es de dimensión 2, se entra a un caso base y por tanto no cambia.

6. Actualizar B, C y A usando D

$$B = \min(B, \text{minplus}(B, D))$$
$$\text{minplus}(B, D) = (\min(0+1, 5+2) \;\; \min(1+2, 5+0)) = (1 \;\; 3)$$
$$B = \min(B, (1 \;\; 3)) = (1 \;\; 3)$$

$$C = \min(C, \text{minplus}(D, C))$$
$$\text{minplus}(D, C) = \begin{pmatrix} \min(0+1, 2+5) \\ \min(2+1, 0+5) \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$
$$C = \min\left(C, \begin{pmatrix} 1 \\ 3 \end{pmatrix}\right) = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

$$A = \min(A, \text{minplus}(B, C))$$
$$\text{minplus}(B, C) = (\min(1+1, 3+3)) = (2)$$
$$A = \min(A, (2)) = (0)$$

7. Finalmente, la matriz resultado con los bloques actualizados es:

$$H^* = \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 0 & 1 & 3 \\ 1 & 0 & 2 \\ 3 & 2 & 0 \end{pmatrix}$$

Esta matriz representa las distancias mínimas entre todos los nodos, coincidiendo con la matriz hallada en el paso 3 del Apéndice A.1.

Bibliografía

The 1000 Genomes Project Consortium. (2015). A global reference for human genetic variation. *Nature*, *526*, 68-74. https://doi.org/10.1038/nature15393

Anjary, T. (2023,). *The Floyd-Warshall Algorithm Re-implemented Using 3D-Tensors and Hardware Acceleration*. https://arxiv.org/abs/2310.03983

Asplund, T. (2014,). *Formalizing the Kleene Star for Square Matrices* (Número 14002). IT.

Bellman, R. (1958). ON A ROUTING PROBLEM. *Quarterly of Applied Mathematics*, *16*, 87-90. https://api.semanticscholar.org/CorpusID:123639971

BLAS Technical Forum. (2025,). *Basic Linear Algebra Subprograms (BLAS)*.

D'alberto, P., & Nicolau, A. (2007). R-Kleene: A High-Performance Divide-and-Conquer Algorithm for the All-Pair Shortest Path for Densely Connected Networks. *Algorithmica*, *47*(2), 203-213.

Developers, N. (2025a,). *numpy.ndarray — NumPy Manual*. https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html

Developers, N. (2025b,). *numpy.power — NumPy Manual*. https://numpy.org/doc/stable/reference/generated/numpy.power.html

Developers, S. (2025c,). *scipy.sparse.csgraph.floyd_warshall — SciPy Reference*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.floyd_warshall.html

Developers, S. (2025a,). *scipy.spatial.distance.pdist — SciPy v1.XX Reference Guide*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html

Developers, S. (2025b,). *scipy.spatial.distance.squareform — SciPy v1.XX Reference Guide*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.squareform.html

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, *1*, 269-271. http://eudml.org/doc/131436

Floyd, & Warshall. (1962). Algorithm 97: Shortest path. *Commun. ACM*, *5*(6), 345. https://doi.org/10.1145/367766.368168

Ford, L. R. (1956). *Network Flow Theory* (technical report No. P-923).

graph-tool developers. (s. f.). *graph-tool: Python library for graph analysis*.

Graph-Tool documentation. (2025,).

hpcgarage. (2025,). *cuASR: CUDA Algebra for Semirings*.

Johnson, D. B. (1977). Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM*, *24*(1), 1-13. https://doi.org/10.1145/321992.321993

Kirk, D. B., & Hwu, W.-m. W. (2012). *Programming Massively Parallel Processors: A Hands-on Approach* (2.ª ed.). Morgan Kaufmann Publishers Inc.

Kolmogorov, A., & Fomin, S. (1975). *Introductory Real Analysis*. Dover Publications. https://books.google.com.uy/books?id=z8IaHgZ9PwQC

Kozen, D. C. (1991). *The Design and Analysis of Algorithms*. Springer-Verlag.

Labs, M. (2025b,). *What is Parallel Thread Execution?*. https://modal.com/gpu-glossary/device-software/parallel-thread-execution

Labs, M. (2025a,). *What is the CUDA Programming Model?*. https://modal.com/gpu-glossary/device-software/cuda-programming-model

Long, M. (2024). *Aplicación de la distancia de Fermat en la visualización de datos genéticos*. https://drive.google.com/file/d/1bxtYki6bGjbF2yyWwAiqr89y2gGxyX3x/view

Ltaief, H., Alomairy, R., Cao, Q., Ren, J., Slim, L., Kurth, T., Dorschner, B., Bougouffa, S., Abdelkhalak, R., & Keyes, D. E. (2024,). *Toward Capturing Genetic Epistasis From Multivariate Genome-Wide Association Studies Using Mixed-Precision Kernel Ridge Regression*. https://arxiv.org/abs/2409.01712

Message Passing Interface Forum. (2021). *MPI: A Message-Passing Interface Standard Version 4.0*. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

Nesmachnow, S., & Iturriaga, S. (2019). Cluster-UY: Collaborative Scientific High Performance Computing in Uruguay. En M. Torres & J. Klapp (eds.), *Supercomputing: Supercomputing*.

Novembre, J., & Stephens, M. (2008). Interpreting principal component analyses of spatial population genetic variation. *Nature Genetics*, *40*(5), 646-649. https://doi.org/10.1038/ng.139

NVIDIA Corporation. (s. f.-a). *CUTLASS — CUDA Templates for Linear Algebra Subroutines and Solvers*.

NVIDIA Corporation. (s. f.-b). *Efficient GEMM using CUTLASS*.

NVIDIA Corporation. (s. f.-c). *NVLink High-Speed GPU Interconnect – NVLink Bridges*.

Salykov, A. (2025, enero). *Advanced Matrix Multiplication Optimization on NVIDIA GPUs*.

Sapienza, F., Jonckheere, M., & Groisman, P. (2018). *Weighted Geodesic Distance Following Fermat's Principle* (p.) [Doctoral dissertation].

SciPy Community. (s. f.). *SciPy — Open-source scientific tools for Python*.

Su, S., Sun, X., Li, X., Wang, A., Li, J., & Shum, C. (2025,). *CUDA-L2: Surpassing cuBLAS Performance for Matrix Multiplication through Reinforcement Learning*. https://arxiv.org/abs/2512.02551

Team, R. D. (2025c,). *cuGraph — RAPIDS Graph Analytics Library*.

Team, R. D. (2025a,). *RAPIDS cuML Documentation: API Reference*. https://docs.rapids.ai/api/cuml/stable/api/

Team, R. D. (2025b,). *RAPIDS cuVS Legacy Python API: Distance*. https://docs.rapids.ai/api/cuvs/legacy/python_api/distance/

Wikipedia contributors. (2025,). *Infinite alleles model — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Infinite_alleles_model&oldid=1295839006

Wikipedia contributors. (2025,). *Single-nucleotide polymorphism — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Single-nucleotide_polymorphism&oldid=1300696560