



Implementación y Comparación del Rendimiento de Algoritmos de Similitud Vectorial en CPU/GPU para Escenarios con Alta Cantidad de Comparaciones en Hardware Convencional de Escritorio

Otoño 2025/2026

Autores:

Kirian Roca Moraga

Pau Morente Alcober

Tarjetas Gráficas y Aceleradores

Profesor: Agustín Fernández Jiménez

Índice

1. Introducción.....	4
2. Objetivo.....	5
3. Preparación.....	7
3.1 Modelos utilizados.....	7
3.2 Análisis de los modelos de Embeddings.....	8
4. Algoritmos.....	11
4.1 Cálculo de Similitud Euclidiano.....	11
4.2 Cálculo de Similitud por Coseno.....	12
4.3 Correlación de Pearson.....	12
5. Implementación CPU.....	15
5.1 Estrategias CPU.....	15
5.1.1 Ejecución en serie y Paralelización a nivel de queries.....	15
5.1.2 Paralelización de la métrica mediante reducción.....	17
6. Implementación GPU.....	20
6.1 Estrategias.....	20
6.1.1 Grid por pares + métrica en secuencial.....	21
6.1.2 Grid en grupo pequeño, loop en grupo grande + métrica en secuencial.....	22
6.1.3 Grid en grupo grande, loop en grupo pequeño + métrica en secuencial.....	23
6.1.4 Grid en grupo pequeño + reducción en árbol sobre la métrica.....	24
6.1.5 Grid en grupo grande + reducción en árbol sobre la métrica.....	25
6.1.6 2D Tiled.....	26
7. Comparación.....	27
7.1 Comparación CPU/GPU.....	27
7.2 Comparación Estrategias GPU.....	28
Estrategia 1 (S1): Grid AxB, cada bloque compara un vector de A y uno de B.....	28
Estrategia 2 (S2): Paralelismo sobre el Grupo de Consultas (Queries).....	28
Estrategia 3 (S3): Paralelismo sobre la Base de Datos.....	29
Estrategia 5 (S5): Reducción Paralela con Caché de Queries.....	29
Estrategia 6: Reducción Paralela con Caché de Base de Datos.....	30
Estrategia 7: Tiling con Memoria Compartida.....	31
Resumen de Mejoras (Speedup) respecto a la Estrategia 1 (RTX 3060).....	32
7.3 Comparación Ejecución en diferentes Gráficas.....	32
1. Especificaciones Técnicas del Hardware.....	32
2. RTX 3050 Mobile:.....	33
3. RTX 3060:.....	33
4. RTX 4070 Ti SUPER:.....	34
7.4 Conclusiones.....	34
8. ANEXOS.....	36
8.1 Tablas ejecuciones (mejores resultados).....	36
8.1.1 Ejecución en serie (CPU).....	36
8.1.2 Paralelización a nivel de queries (CPU).....	36

8.1.3 Paralelización de la métrica mediante reducción (CPU).....	36
8.1.4 Grid por pares + métrica en secuencial (GPU).....	37
8.1.4.1 Vectores de 384 dimensiones.....	37
8.1.4.2 Vectores de 768 dimensiones.....	37
8.1.4.3 Vectores de 1024 dimensiones.....	38
8.1.5 Grid en grupo pequeño, loop en grupo grande + métrica en secuencial (GPU)....	39
8.1.5.1 Vectores de 384 dimensiones.....	39
8.1.5.2 Vectores de 768 dimensiones.....	39
8.1.5.3 Vectores de 1024 dimensiones.....	40
8.1.6 Grid en grupo grande, loop en grupo pequeño + métrica en secuencial (GPU)....	41
8.1.6.1 Vectores de 384 dimensiones.....	41
8.1.6.2 Vectores de 768 dimensiones.....	42
8.1.6.3 Vectores de 1024 dimensiones.....	42
8.1.7 Grid en grupo pequeño + reducción en árbol sobre la métrica (GPU).....	43
8.1.7.1 Vectores de 384 dimensiones.....	43
8.1.7.2 Vectores de 768 dimensiones.....	44
8.1.7.3 Vectores de 1024 dimensiones.....	44
8.1.8 Grid en grupo grande + reducción en árbol sobre la métrica (GPU).....	45
8.1.8.1 Vectores de 384 dimensiones.....	45
8.1.8.2 Vectores de 768 dimensiones.....	46
8.1.8.3 Vectores de 1024 dimensiones.....	47
8.1.9 2D Tiled (GPU).....	48
8.1.9.1 Vectores de 384 dimensiones.....	48
8.1.9.2 Vectores de 768 dimensiones.....	48
8.1.9.3 Vectores de 1024 dimensiones.....	49

Todos los códigos en Github: <https://github.com/pmorente/final-TGA-Project>

1. Introducción

Hoy en día cuando hablamos de procesamiento de lenguaje natural (NLP), visión por computadora o sistemas de recomendación es muy frecuente mencionar la palabra embedding. Un embedding es una forma de representar objetos complejos, como textos, imágenes o audio mediante vectores en un espacio multidimensional.

Nosotros utilizaremos los embeddings en una técnica popularizada recientemente con el boom de la IA. La generación aumentada de recuperación (el famoso RAG). El proceso es el siguiente:

- Se dividen los textos en frases y se pasan sobre un modelo específicamente entrenado para generar embeddings. Estos modelos de embeddings pueden tomar una simple frase, extraer su contexto semántico y distribuirlo en un número finito de dimensiones (p.e 384, 768 o 1024 dimensiones)
- Una vez tienes los embeddings generados, se guardan en bases de datos especializadas para este tipo de datos, conocidas como bases de datos vectoriales.
- Después se realizan búsquedas por similitud vectorial y se recuperan los embeddings similares para utilizarlos como contexto para los LLM.

En este proyecto estudiaremos distintos algoritmos para calcular la similitud vectorial entre dos vectores, los implementaremos con versiones propias optimizadas para CPU y GPU y realizaremos un análisis comparativo de rendimiento empleando únicamente hardware convencional de escritorio.

Actualmente la mayoría de estos cálculos sigue realizándose en la nube, la evolución del hardware doméstico y la aparición de nuevas herramientas de software apunta a una creciente capacidad para asumir parte de estas tareas de forma local. Por esta razón optimizar la eficiencia de estos algoritmos resulta crucial para que portátiles y ordenadores convencionales puedan ejecutarlos de manera viable y con un consumo energético contenido.

Embeddings:<https://developers.google.com/machine-learning/crash-course/embeddings/embedding-space>

RAG:<https://learn.microsoft.com/es-es/azure/search/retrieval-augmented-generation-overview>

Bases Vectoriales: <https://www.trychroma.com/>

2. Objetivo

El objetivo de este proyecto es implementar los algoritmos matemáticos que calculan la similitud entre vectores de D dimensiones tanto en CPU como en GPU y analizar el rendimiento. Compararemos el rendimiento entre ambas versiones y, posteriormente, se analizarán y compararán las distintas optimizaciones implementadas en CUDA para maximizar la velocidad de ejecución de los diferentes algoritmos.

Para ejecutar las pruebas de rendimiento utilizaremos el conjunto de vectores (1.000, 10.000) que pertenece al conjunto de datos que hemos creado formado por conjuntos de 500, 1.000, 10.000, 50.000 y 100.000 vectores (embeddings) de tres tamaños distintos: 384, 768 y 1024 dimensiones. Este experimento nos dará un cómputo de 1.000.000 de operaciones sobre las cuales evaluaremos el rendimiento CPU/GPU.

Ejecutaremos el experimento en el siguiente hardware convencional:

CPU

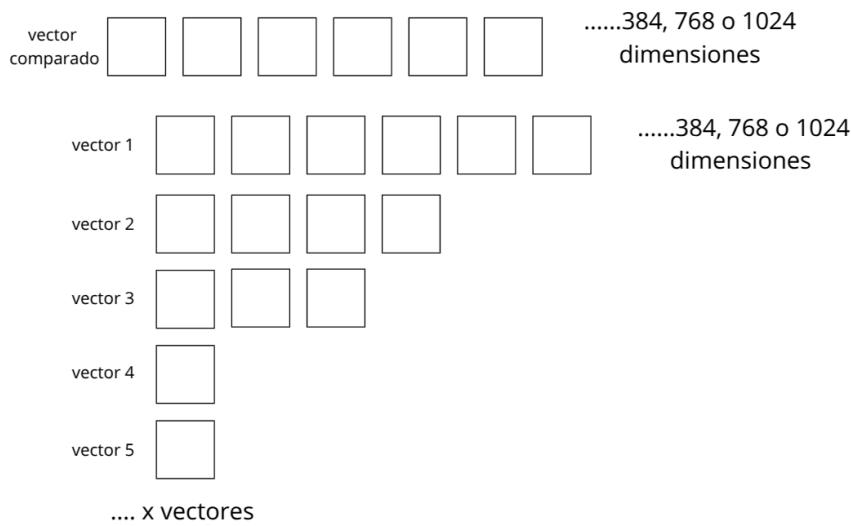
Intel(R) Core(TM) Ultra 7 265K (3.90 GHz) Cores/Hilos: 8 P-cores + 12 E-cores = 20 hilos
--

GPU

GPU	Arquitectura	CUDA Cores	VRAM (tipo)	Bus memoria	Ancho banda	Tensor/RT Cores
NVIDIA GeForce RTX 4070 Ti SUPER	Ada Lovelace	8 448	16 GB GDDR6 X	256-bit	~672 GB/s	264 Tensor, 66 RT
NVIDIA GeForce RTX 3060	Ampere	3 584	12 GB GDDR6	192-bit	~360 GB/s	112 Tensor, 28 RT
NVIDIA GeForce RTX 3050 Mobile	Ampere	2 048	4 GB GDDR6	128-bit	~192 GB/s	64 Tensor, 16 RT

Los algoritmos matemáticos que vamos a implementar para el cálculo de similitud entre dos vectores son los siguientes:

- *Cálculo de Similitud Euclíadiano*
- *Cálculo de Similitud por Coseno*
- *Correlación de Pearson*



El proceso principal será cruzar X' (1.000 vectores) vectores con Y' (10.000 vectores), calculando para cada par de vectores la similitud.

La clave aquí es paralelizar tanto el bucle anidado entre el conjunto pequeño y el conjunto grande de vectores como la propia operación de la métrica. Propondremos distintas estrategias y las compararemos para evaluar el rendimiento.

3. Preparación

Antes de programar los distintos algoritmos de cálculo de similitud es necesario preparar el dataset de vectores. Hemos utilizado el dataset “rojagtap/bookcorpus”, que contiene frases ya separadas de diversos libros. Se han generado los códigos necesarios para crear embeddings con tres modelos: *all-MiniLM-L6-v2* (384 dimensiones), *all-mpnet-base-v2* (768 dimensiones) y *all-roberta-large-v1* (1024 dimensiones).

En la carpeta `tooling_dataset_creation` se incluye el algoritmo empleado para extraer las frases del dataset de HuggingFace, así como los algoritmos paralelizados por lotes mediante la librería NumPy para ejecutar los modelos y convertir las frases en vectores.

3.1 Modelos utilizados

Para generar los embeddings hemos utilizado los siguientes modelos:

- *all-MiniLM-L6-v2*
- *all-mpnet-base-v2*
- *all-roberta-large-v1*

Todos los modelos mencionados provienen de modelos creados por otras empresas (Google, Microsoft, META...) pero han sido ajustados (Fine-Tuning) usando la misma librería: *Sentence Transformers* (SBERT) para generar embeddings (vectores finales que representan oraciones enteras). Este ajuste (fase de especialización del modelo) está basado en una arquitectura de redes siamesas, aprendizaje contrastivo y un pooling final.

- La **red siamesa** recibe dos entradas a la vez, oración A y oración B, esta aplica los mismos criterios (pesos) para ambas oraciones y crea un subvector/token por cada palabra de la oración de las dimensiones pertinentes (384, 768 o 1024). Una vez genera todos los tokens hace un **pooling**, este se encarga de crear el vector final. Normalmente aplica la media (mean pooling), es decir, a cada dimensión del vector final se le asigna la media de los subvectores en esa dimensión). Si A es similar a B el vector final de A debería tener una dirección similar al vector final de B.
- El **aprendizaje contrastivo** se encarga de decirle a la red si ha hecho un buen trabajo o no. Una vez que los ha generado y calculado su similitud, se le dice la relación real que tienen esas dos oraciones. Si no ha acertado, esta ajusta los pesos para modificar los vectores, con el objetivo de alinear o separar.

Que la especialización sea así provoca que los modelos están centrados en optimizar las direcciones de los vectores y no la magnitud de estos. Por lo tanto, puede darse el caso en que frases similares tengan vectores en la misma dirección, pero que los puntos que representan estén lejos. Esta librería ofrece la configuración para que los embeddings resultantes estén normalizados, indagaremos más sobre este tema en el [apartado 4](#).

Todos han sido ajustados con el mismo conjunto de datos, conocido como "The All-NLI / 1B Sentence Pairs Dataset", el cual tiene más de 1.000 millones de pares de oraciones. Estas provienen de una mezcla de fuentes diversa que incluye Reddit, Stack Exchange, Yahoo Answers, Google Scholar (títulos de papers), y datasets de preguntas y respuestas (GOOAQ).

Debido a esta mezcla, estos modelos tienen una preferencia semántica generalista. No están sesgados hacia un solo dominio (como el médico o legal), sino que son expertos en entender la intención y el contexto en el lenguaje cotidiano y técnico.

Es decir, entienden que una oración cotidiana como "¿Cómo arreglo mi pantalla?", es semánticamente similar a una técnica como "Guía de reparación de monitores", aunque no comparten muchas palabras.

Estos tres modelos están optimizados por defecto para usar similitud por coseno a causa del ajuste utilizado, ya que la similitud por coseno se centra en comparar direcciones de vectores y no las distancias en el espacio. No obstante, se pueden hacer ajustes para que otros métodos como la distancia euclídea se puedan aplicar adecuadamente.

SBERT: <https://arxiv.org/abs/1908.10084/>

3.2 Análisis de los modelos de Embeddings

all-MiniLM-L6-v2 (MiniLM con SBERT)

Es el producto de un proceso de destilación (obtener un rendimiento similar con menor coste) del modelo de Google BERT-Base. Este es un modelo potente, con muchísimos parámetros (aproximadamente 110 millones) pero con un entrenamiento subóptimo, un tamaño de 768 dimensiones y 12 capas (las capas son las fases de cálculo por las que tiene que pasar cada texto).

BERT utiliza el método MLM (Masked Language Modeling), este se basa en enmascarar palabras al azar de las oraciones haciendo énfasis en entender el contexto para predecir las palabras ocultas. Además, usa también NSP (Next Sentence Prediction) para intentar deducir si la frase B venía después de la frase A.

Esto tiene puntos fuertes como entender muy bien las diferencias según el contexto, crucial en el caso de palabras polisémicas. No obstante, al entrenar con palabras enmascaradas está acostumbrado a un entorno diferente del de la realidad que puede llevar a confusiones, por ejemplo, en casos donde hay más de una máscara yuxtapuesta puede no entender que esas palabras van juntas (como “Nueva York”) e intenta adivinarlas por separado.

Investigadores de Microsoft hicieron una destilación de la cuál surgió MiniLM que conseguía un 99% del rendimiento de BERT pero con la mitad de tiempo de ejecución. Después de utilizar el ajuste (Fine-Tuning) previamente descrito nace all-MiniLM).

Este tiene aproximadamente 22 millones de parámetros, usa 384 dimensiones y 6 capas (menos pasos) por lo que es el que tiene un menor coste en cuanto a memoria y pasos de los tres, por este motivo, también es el más rápido en procesar. No tiene un entendimiento tan bueno como podría tener MPNet pero es más rápido.

BERT: <https://arxiv.org/abs/1810.04805/>

MiniLM: <https://arxiv.org/abs/2002.10957/>

all-mpnet-base-v2 (MPNet con SBERT)

Es el producto de la combinación de XLNet con BERT. XLNet nació para arreglar los problemas de BERT y utiliza el método PLM (Permutation Language Modeling). En lugar de enmascarar palabras las muestra poco a poco y permuta el orden. Es decir, entrena al modelo a “adivinar” que palabra es la que toca a medida que le va mostrando la frase poco a poco en vez de enseñarle toda la frase con la palabra tapada.

Este modelo tiene puntos fuertes como que permite que la máscara no confunda al modelo de lenguaje y que el modelo entienda muy bien la relación entre palabras haciéndolo mejor para textos largos o complejos. El punto débil es que es mucho más complejo y arduo de entrenar que BERT.

En este caso, es fundamental distinguir entre la complejidad dimensional (#dimensiones/capas) y la complejidad algorítmica (naturaleza de la predicción). En el caso de XLNet, su mayor coste computacional frente a BERT no deriva de tener más capas o dimensiones, sino de su mecanismo de Autoatención de Doble Flujo (Two-Stream Self-Attention).

Este mecanismo es el encargado de saber qué palabra se está prediciendo sin que el modelo sepa realmente qué palabra es para evitar contaminar dicha predicción.

MPNet viene de usar **MLM** de BERT y **PLM** de **XLNet** (Masked and Permuted Network) haciendo que tenga un gran entendimiento en el contexto estático y en dependencias complejas entre palabras. Todo ello con un coste equilibrado entre BERT y XLNet. Este

coste equilibrado junto a su entendimiento superior es su mayor punto fuerte, ya que con un coste computacional similar al de BERT-Base (768 dimensiones y 12 capas) consigue aprovechar los puntos fuertes de XLNet al zafarse del Doble Flujo de Autoatención (consigue un entendimiento superior con el mismo coste computacional, por lo que es más eficiente).

Por este motivo, MPNet ha sido referencia a nivel de modelos, siendo muy usado y se le podría considerar el equilibrio perfecto (al menos durante bastante tiempo) entre entendimiento y velocidad (entendimiento a efectos prácticos equivalente al de un modelo *large* de 24 capas con coste de modelo *base* de 12 capas).

MPNet: <https://arxiv.org/abs/2004.09297/>

XLNet: <https://arxiv.org/abs/1906.08237/>

all-roberta-large-v1 (RoBERTa con SBERT)

Google, partiendo de la premisa de que consideraban que BERT estaba entrenado de forma subóptima, decidieron investigar qué elementos confundieron o perjudicaron el aprendizaje.

Se dieron cuenta de que:

- Siempre enmascaraba la misma o las mismas palabras cuando se encontraba con la misma frase.
- Se dieron cuenta que usar “Next Sentence Prediction” (NSP) era perjudicial porque confundía al modelo.
- La cantidad de datos de entrenamiento podría ser mayor.

Por lo tanto, decidieron remediar estos puntos haciendo un nuevo modelo gemelo llamado RoBERTa (Robustly optimized BERT approach) pero que implementa un enmascaramiento dinámico para evitar que el modelo “memorice” las predicciones, elimina el NSP y lo entrenaron con un conjunto de datos 10 veces mayor al de BERT (16GB → 160GB). Lo cual, mejoró significativamente la capacidad de entendimiento del modelo, según algunos benchmarks entre un 9% y un 15%, respecto a BERT-Large. Obteniendo la mayor diferencia en contextos con una gran complejidad de razonamiento y una larga extensión.

El modelo que nosotros utilizaremos es la versión RoBERTa-Large con el ajuste SBERT, por lo tanto, tiene aproximadamente 355 millones de parámetros, 24 capas y 1024 dimensiones. Estas características hacen que sea computacionalmente mucho más costoso, lo cual es su punto débil. Por otra lado, estas características también le aportan puntos fuertes como tener (potencialmente) una capacidad mayor para distinguir relaciones o entender estructuras semánticas mucho más complejas que podrían confundir a modelos más reducidos.

RoBERTa: <https://arxiv.org/abs/1907.11692/>

4. Algoritmos

4.1 Cálculo de Similitud Euclidiano

Es la raíz de la suma al cuadrado de las distancias entre cada componente (cada dimensión) del vector A y el vector B. En otras palabras, mide la distancia de la línea recta que une el punto A con el punto B, siendo las coordenadas de los puntos sus respectivos vectores. De forma que cuanto menor es esta distancia, más cerca están en el espacio (más parecidos son).

Es importante recalcar que los modelos sentence-transformers no generan embeddings normalizados (longitud = 1) por defecto, aunque se puede configurar para que se normalicen fácilmente.

Si los vectores están normalizados, la Distancia Euclíadiana y la Similitud del Coseno son perfectamente correlacionadas.

- Si el Coseno sube (más similitud), la Distancia Euclíadiana baja (menos distancia).
- Matemáticamente: $D_{\text{euclíadiana}}^2 = 2 * (1 - \text{CosineSimilarity})$.

Si los vectores no están normalizados, la distancia Euclíadiana podría dar resultados "raros". Vectores largos similares a cortos podrían parecer lejanos. Esto se debe a que a la hora de crearlos, además del significado de las palabras, también tienen en cuenta la longitud de la oración. Por lo tanto, una frase muy larga que semánticamente es muy similar a una corta (van en la misma dirección) puede estar muy lejos porque el modelo le ha dado una longitud mayor al vector.

Al normalizarlos, todos tendrían la misma longitud y las distancias serían únicamente causadas por la diferencia de dirección.

Dados dos vectores de D dimensiones:

$$d(A, B) = \sqrt{\sum_{i=1}^D (A_i - B_i)^2}$$

Restar componente a componente del vector: $diff_i = A_i - B_i$

Elevar al cuadrado: $squared_i = diff_i^2$

Sumar todos: $sum = \#(i = 1)^D squared_i$

Hacer raíz cuadrada: $d = \sqrt{\text{sum}}$

4.2 Cálculo de Similitud por Coseno

La similitud mide el ángulo entre los vectores. Ignora la magnitud (longitud) y se centra en la orientación. Esta es la métrica nativa. Dará los mejores resultados semánticos (frases parecidas tendrán valores cercanos a 1).

Dados dos vectores de D dimensiones:

$$\cos_{\text{sim}}(A, B) = \frac{\sum_{i=1}^D A_i B_i}{\sqrt{\sum_{i=1}^D A_i^2} \cdot \sqrt{\sum_{i=1}^D B_i^2}}$$

Normalizar cada vector (aunque estos modelos suelen dar ya vectores normalizados):

$$A'_i = \frac{A_i}{\sqrt{\sum_{j=1}^D A_j^2}} \quad B'_i = \frac{B_i}{\sqrt{\sum_{j=1}^D B_j^2}}$$

Multiplicar componente a componente:

$$\text{prod}_i = A'_i \cdot B'_i$$

Sumar todos los productos:

$$\cos_{\text{sim}} = \sum_{i=1}^D \text{prod}_i$$

4.3 Correlación de Pearson

La correlación de Pearson mide la relación lineal entre las variables. En el contexto de embeddings, mide si las dimensiones "suben y bajan juntas" respecto a la media de cada vector. Esencialmente, es una Similitud de Coseno sobre vectores centrados (donde se ha restado la media a cada número del vector).

Es computacionalmente más costosa que el coseno simple (requiere calcular medias y desviaciones estándar para cada vector). En los embeddings de NLP (Procesamiento de Lenguaje Natural), suele dar resultados muy similares al Coseno, pero no es estándar.

$$r(A, B) = \frac{\sum_{i=1}^D (A_i - \bar{A})(B_i - \bar{B})}{\sqrt{\sum_{i=1}^D (A_i - \bar{A})^2} \cdot \sqrt{\sum_{i=1}^D (B_i - \bar{B})^2}}$$

Dados dos vectores de D dimensiones:

Calcular las medias:

$$\bar{A} = \frac{1}{D} \sum_{i=1}^D A_i \quad \bar{B} = \frac{1}{D} \sum_{i=1}^D B_i$$

Restar las medias a cada componente:

$$A'_i = A_i - \bar{A}$$

$$B'_i = B_i - \bar{B}$$

Multiplicar las desviaciones:

$$prod_i = A'_i \cdot B'_i$$

Sumar los productos y dividir por el producto de las desviaciones estándar:

$$r = \frac{\sum_{i=1}^D prod_i}{\sqrt{\sum_{i=1}^D (A'_i)^2} \cdot \sqrt{\sum_{i=1}^D (B'_i)^2}}$$

No obstante, calcular la correlación de Pearson con esta estrategia plantea un problema a la paralelización que a primera vista podría no ser perceptible. La problemática es el uso de las medias de los vectores, esto causa que haya que hacer un recorrido previo para resolver la media del vector antes de calcular los sumatorios. Dependiendo de la estrategia, esto provoca un aumento en el coste de sincronización y bucles extra. Para solventar dicha casuística se explorará una solución basada en la transformación de la fórmula de correlación de Pearson en una versión equivalente que se deshaga de las medias.

Tras aplicar transformaciones matemáticas, el resultado que obtenemos es la siguiente:

$$r(A, B) = \frac{D \sum (A \cdot B) - (\sum A)(\sum B)}{\sqrt{D \sum A^2 - (\sum A)^2} \sqrt{D \sum B^2 - (\sum B)^2}}$$

Se consigue la expresión equivalente de la siguiente manera:

$$\sum (A_i - \bar{A})(B_i - \bar{B}) = \sum (A_i B_i - A_i \bar{B} - \bar{A} B_i + \bar{A} \bar{B})$$

$$= \sum A_i B_i - \bar{B} \sum A_i - \bar{A} \sum B_i + D(\bar{A} \bar{B})$$

Sustituyendo las medias (\bar{A}, \bar{B}) :

$$= \sum A_i B_i - \frac{\sum B \sum A}{D} - \frac{\sum A \sum B}{D} + D \frac{\sum A \sum B}{D^2}$$

$$= \sum A_i B_i - \frac{\sum A \sum B}{D}$$

Multiplicando por D para eliminar la fracción:

$$\text{Numerador Transformado} = D \sum A_i B_i - (\sum A_i)(\sum B_i)$$

$$\begin{aligned}
\sum (A_i - \bar{A})^2 &= \sum (A_i^2 - 2A_i\bar{A} + \bar{A}^2) \\
&= \sum A_i^2 - 2\bar{A} \sum A_i + D\bar{A}^2 \\
&= \sum A_i^2 - 2\frac{(\sum A)^2}{D} + \frac{(\sum A)^2}{D} \\
&= \sum A_i^2 - \frac{(\sum A)^2}{D}
\end{aligned}$$

Multiplicando por D (dentro de la raíz, equivale a D^2):

$$\text{Denominador Transformado} = \sqrt{D \sum A_i^2 - (\sum A_i)^2}$$

A esta versión la llamaremos más adelante `pearson_opt` y solo la utilizaremos en las versiones de GPU, ya que nos es especialmente útil al paralelizar el cálculo.

5. Implementación CPU

Todos los códigos en Github: <https://github.com/pmorente/final-TGA-Project>

Hemos implementado tres estrategias diferentes, paralelizando en algunos casos con OpenMP. La primera ejecuta el cálculo en serie, la segunda paralleliza el conjunto de queries que se van a comparar (grupo pequeño de 1.000 vectores) y la tercera paralleliza el cómputo de la métrica mediante reducción.

5.1 Estrategias CPU

5.1.1 Ejecución en serie y Paralelización a nivel de queries

El primer código en CPU (compare_one_to_many.cpp) es el secuencial. Este código con un solo subprocesso recorre los bucles anidados en serie, lee ambos archivos, válida las dimensiones y realiza cada comparación dependiendo de la métrica seleccionada, que también se hace secuencialmente.

El segundo código paralelizamos a nivel de los vectores únicamente (parallel_compare_one_to_many.cpp): En este caso utilizamos OpenMP para parallelizar el bucle externo sobre los vectores del grupo pequeño de 1.000 vectores. Cada hilo posee una porción del grupo pequeño y procesa todos sus emparejamientos con B secuencialmente.

Los códigos de las métricas son los mismos para los dos estrategias:

```
// ----- CPU Euclidean similarity -----
float euclidean_distance_cpu(const float* __restrict A, const float* __restrict B, int D) {
    double acc = 0.0;
    int i = 0;

    // Loop unrolling para optimización
    for (; i + 4 <= D; i += 4) {
        double d0 = A[i] - B[i];
        double d1 = A[i+1] - B[i+1];
        double d2 = A[i+2] - B[i+2];
        double d3 = A[i+3] - B[i+3];
        acc += d0*d0 + d1*d1 + d2*d2 + d3*d3;
    }
    for (; i < D; ++i) {
        double d = A[i] - B[i];
        acc += d * d;
    }

    return static_cast<float>(sqrt(acc));
}
```

```

// ----- CPU cosine similarity -----
float cosine_cpu(const std::vector<float> &A, const std::vector<float> &B) {
    if (A.size() != B.size()) throw std::runtime_error("Vector sizes differ");
    double dot = 0.0, normA = 0.0, normB = 0.0;
    int D = A.size();
    for (int i = 0; i < D; ++i) {
        double ai = A[i], bi = B[i];
        dot += ai * bi;
        normA += ai * ai;
        normB += bi * bi;
    }
    double denom = sqrt(normA) * sqrt(normB) + 1e-12;
    return static_cast<float>(dot / denom);
}

```

```

// ----- Pearson Correlation -----
float pearson_corr_cpu(const float* __restrict A, const float* __restrict B, int D) {
    double meanA = 0.0, meanB = 0.0;
    for (int i = 0; i < D; ++i) {
        meanA += A[i];
        meanB += B[i];
    }
    meanA /= D;
    meanB /= D;

    double num = 0.0, denA = 0.0, denB = 0.0;
    int i = 0;
    for (; i + 4 <= D; i += 4) {
        double da0 = A[i] - meanA; double db0 = B[i] - meanB;
        double da1 = A[i+1] - meanA; double db1 = B[i+1] - meanB;
        double da2 = A[i+2] - meanA; double db2 = B[i+2] - meanB;
        double da3 = A[i+3] - meanA; double db3 = B[i+3] - meanB;
        num += da0*db0 + da1*db1 + da2*db2 + da3*db3;
        denA += da0*da0 + da1*da1 + da2*da2 + da3*da3;
        denB += db0*db0 + db1*db1 + db2*db2 + db3*db3;
    }
    for (; i < D; ++i) {
        double da = A[i] - meanA;
        double db = B[i] - meanB;
        num += da * db;
        denA += da * da;
        denB += db * db;
    }

    double denom = sqrt(denA * denB) + 1e-12;

```

```

    return static_cast<float>(num / denom);
}

```

Lo único que cambia entre estas dos estrategias es el bucle que hacemos para ir juntando los pares de vectores:

```

// ----- Ejecución en serie -----
for (size_t i = 0; i < records_a.size(); ++i) {
    for (size_t j = 0; j < records_b.size(); ++j) {
        float result = compute_metric(metric, records_a[i].embedding, records_b[j].embedding);
        (void)result; // Prevent optimization
    }
    ...
}

```

Paralelizamos el bucle exterior con OpenMP.

```

// -----Paralelización a nivel de queries (conjunto pequeño) -----
#pragma omp parallel for schedule(dynamic)
for (size_t i = 0; i < records_a.size(); ++i) {
    for (size_t j = 0; j < records_b.size(); ++j) {
        float result = compute_metric(metric, records_a[i].embedding, records_b[j].embedding);
        (void)result; // Prevent optimization
    }
    ...
}

```

5.1.2 Paralelización de la métrica mediante reducción

En esta estrategia paralelizamos a nivel de métrica (parallel_metric_compare_one_to_many.cpp). En este caso no paralelizamos los bucles, paralelizamos el bucle de dimensión interno dentro de cada métrica (coseno, eucladiana, Pearson) mediante reducciones de OpenMP. Los subprocessos colaboran en un único par de vectores dividiendo sus dimensiones lo que refleja el patrón de reducción.

```

// ----- CPU cosine similarity with parallelized inner loops -----
float cosine_cpu_parallel(const std::vector<float> &A, const std::vector<float> &B) {
    if (A.size() != B.size()) throw std::runtime_error("Vector sizes differ");

    int D = A.size();
    double dot = 0.0, normA = 0.0, normB = 0.0;

    // Parallelize the loop over dimensions using OpenMP reduction
    #pragma omp parallel for reduction(+:dot,normA,normB)
    for (int i = 0; i < D; ++i) {
        double ai = A[i], bi = B[i];
        dot += ai * bi;
        normA += ai * ai;
        normB += bi * bi;
    }

    double denom = sqrt(normA) * sqrt(normB) + 1e-12;
    return static_cast<float>(dot / denom);
}

```

```

// ----- Euclidean Distance with parallelized inner loops -----
float euclidean_distance_cpu_parallel(const float* __restrict A, const float* __restrict B, int D) {
    double acc = 0.0;

    // Parallelize the loop over dimensions using OpenMP reduction
    #pragma omp parallel for reduction(+:acc)
    for (int i = 0; i < D; ++i) {
        double d = A[i] - B[i];
        acc += d * d;
    }

    return static_cast<float>(sqrt(acc));
}

```

En Pearson realizamos dos reducciones siguiendo la fórmula original de Pearson.

```

// ----- Pearson Correlation with parallelized inner loops -----
float pearson_corr_cpu_parallel(const float* __restrict A, const float* __restrict B, int D) {
    double meanA = 0.0, meanB = 0.0;

    // First pass: compute means (parallelized)
    #pragma omp parallel for reduction(+:meanA,meanB)
    for (int i = 0; i < D; ++i) {
        meanA += A[i];
        meanB += B[i];
    }
    meanA /= D;
    meanB /= D;

    // Second pass: compute correlation (parallelized)
    double num = 0.0, denA = 0.0, denB = 0.0;
    #pragma omp parallel for reduction(+:num,denA,denB)
    for (int i = 0; i < D; ++i) {
        double da = A[i] - meanA;
        double db = B[i] - meanB;
        num += da * db;
        denA += da * da;
        denB += db * db;
    }

    double denom = sqrt(denA * denB) + 1e-12;
    return static_cast<float>(num / denom);
}

```

5.3 Tiempos CPU

Hemos ejecutado los códigos de las tres estrategias tres veces para cada fórmula.

Versión en serie, medias del rendimiento por métrica:

Metric	384 dims	768 dims	1024 dims
Cosine	2.043	4.488	6.109
Euclidean	1.332	3.397	4.433
Pearson	3.941	7.782	9.988

Paralelización a nivel de queries, medias del rendimiento por métrica:

Metric	384 dims	768 dims	1024 dims
Cosine	0.425	0.418	0.452
Euclidean	0.411	0.442	0.473
Pearson	0.373	0.463	0.602

Número de threads: 20

Paralelización de la métrica mediante reducción, medias del rendimiento por métrica:

Metric	384 dims	768 dims	1024 dims
Cosine	95.705	148.579	100.935
Euclidean	57.662	52.900	58.108
Pearson	205.797	196.749	194.991

Número de threads: 20 (utilizados para paralelismo a nivel de dimensión)

6. Implementación GPU

Todos los códigos en Github: <https://github.com/pmorente/final-TGA-Project>

6.1 Estrategias

Para abordar la paralelización de la implementación de CPU en CUDA, hemos elaborado las siguientes estrategias para analizar cuál es la más adecuada. Cabe recalcar que para medir el rendimiento de las estrategias hemos tomado las siguientes decisiones:

- Usar un entorno en el que el consumo externo de GPU ronda el 0%.
- Hacer una ejecución de calentamiento o *warm up* para que al obtener los datos de las siguientes ejecuciones la GPU ya esté a máximo rendimiento y no afecte el paso de *sleep mode* a *active mode*.
- Hacer 10 ejecuciones por cada estrategia, calcular la media de tiempo de ejecución y la media de throughput.
- Generar binarios con el resultado para compararlos con el resultado de CPU. De esta forma comprobamos que el resultado es correcto.
- Implementar códigos parametrizables para abordar diferentes configuraciones de la misma estrategia (por ejemplo, la cantidad de dimensiones asignadas a un thread en las estrategias con reducción).

Además, para algunas estrategias, también hemos implementado versiones que en vez de ser completamente configurables (dinámicas) usan templates/plantillas (estáticas con algunos valores definidos) para que el compilador de CUDA pueda optimizar la ejecución del código al saber el valor de las variables previamente a la ejecución.

6.1.1 Grid por pares + métrica en secuencial

Esta estrategia (estrategia1_X.cu) lanza el kernel con un grid bidimensional donde cada bloque corresponde a un par de vectores ($A[i]$, $B[j]$). El grid tiene dimensiones $\text{grid}(\text{num_queries}, \text{num_database})$, de forma que blockIdx.x identifica el índice del vector en el grupo A y blockIdx.y identifica el índice del vector en el grupo B. Cada bloque contiene un único thread ($\text{block}(1)$), y ese thread realiza la comparación completa de forma secuencial, recorriendo todas las dimensiones del vector para calcular la métrica seleccionada. La diferencia principal con estrategias posteriores es que no hay parallelización dentro de la comparación, cada thread procesa todas las dimensiones de forma secuencial lo que la convierte en una estrategia baseline correcta pero lenta.

RTX 3050	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.493911	20.25	1.180315	8.47	1.576215	6.34
Euclidean	0.491580	20.34	0.990433	10.10	1.335154	7.49
Pearson	0.984150	10.16	1.950943	5.13	2.617384	3.82
Pearson Opt	0.499422	20.02	1.185720	8.43	1.584760	6.31

Block size: 32

RTX 3060	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0,27812	35,96	0,667795	14,97	0,882661	11,33
Euclidean	0,279658	35,76	0,558661	17,9	0,73423	13,62
Pearson	0,547638	18,26	1,097405	9,11	1,442059	6,93
Pearson Opt	0,277329	36,06	0,67064	14,91	0,886383	11,28

Block size: 32

RTX 4070 Ti SUPER	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.082283	121.53	0.164514	60.79	0.229004	43.67
Euclidean	0.083431	119.86	0.164626	60.74	0.214961	46.52
Pearson	0.164099	60.94	0.403773	24.77	0.440149	22.72
Pearson Opt	0.082600	121.06	0.166883	59.92	0.21617	46.26

Block size: 128

6.1.2 Grid en grupo pequeño, loop en grupo grande + métrica en secuencial

Esta estrategia (estrategia2_X.cu) lanza el kernel con un grid unidimensional del tamaño del grupo pequeño (A). Cada bloque se asigna a un vector de A y realiza las comparaciones de ese vector con todos los del grupo grande (B). Además, cada bloque tiene un número de threads configurable definido por la variable BLOCK_SIZE de forma que los threads dentro del bloque se reparten las comparaciones con los vectores de B mediante un bucle con stride (cada thread procesa múltiples vectores de B con un incremento igual al número de threads). La diferencia principal con la estrategia 1 es que ahora se paralleliza la distribución de comparaciones entre threads dentro de cada bloque permitiendo que múltiples threads trabajen simultáneamente sobre diferentes vectores de B para el mismo vector de A, aunque cada comparación individual sigue siendo secuencial sobre las dimensiones.

RTX 3050	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.158841	62.96	0.330451	30.26	0.420693	23.77
Euclidean	0.146770	68.13	0.370313	27.00	0.513969	19.46
Pearson	0.357401	27.98	0.759639	13.16	1.018980	9.81
Pearson Opt	0.155962	64.12	0.345070	28.98	0.448331	22.30

Block size: 32

RTX 3060	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0,077195	129,54	0,152939	65,39	0,207912	48,10
Euclidean	0,073462	136,12	0,158650	63,03	0,213725	46,79
Pearson	0,141960	70,44	0,286573	34,90	0,380075	26,31
Pearson Opt	0,078205	127,87	0,154680	64,65	0,214851	46,54

Block size: 32

RTX 4070 Ti SUPER	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.023781	420.50	0.044757	223.43	0.059742	167.39
Euclidean	0.023208	430.89	0.046254	216.20	0.060193	166.13
Pearson	0.044872	222.86	0.089582	111.63	0.119244	83.86
Pearson Opt	0.022558	443.31	0.045047	221.99	0.061661	162.18

Block size: 128

6.1.3 Grid en grupo grande, loop en grupo pequeño + métrica en secuencial

Esta estrategia (estrategia3_X.cu) lanza el kernel con un grid unidimensional del tamaño del grupo grande (B). Cada bloque se asigna a un vector de B y realiza las comparaciones de ese vector con todos los del grupo pequeño (A). Configuramos el número de threads con la variable BLOCK_SIZE de forma que los threads dentro del bloque se reparten las comparaciones con los vectores de A mediante un bucle con stride. A diferencia de la estrategia 2 el grid se organiza sobre el grupo grande en lugar del pequeño lo que puede ser más eficiente cuando el grupo B es significativamente mayor que el grupo A donde se da el caso permitiendo mejor aprovechamiento de los recursos de la GPU al tener más bloques disponibles para la parallelización.

RTX 3050	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.143751	69.56	0.319331	31.32	0.419885	23.82
Euclidean	0.142148	70.35	0.309122	32.35	0.433148	23.09
Pearson	0.284440	35.16	0.712593	14.03	0.964122	10.37
Pearson Opt	0.143132	69.87	0.319192	31.33	0.421576	23.72

Block size: 32

RTX 3060	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0,073609	135,85	0,146234	68,38	0,189069	52,89
Euclidean	0,073143	136,72	0,145057	68,94	0,189171	52,86
Pearson	0,145059	68,94	0,283027	35,33	0,373716	26,76
Pearson Opt	0,074043	135,06	0,144843	69,04	0,190844	52,4

Block size: 32

RTX 4070 Ti SUPER	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.023333	428.58	0.044633	224.05	0.058350	171.38
Euclidean	0.022054	453.43	0.043996	227.29	0.057650	173.46
Pearson	0.042798	233.66	0.085343	117.17	0.113498	88.11
Pearson Opt	0.021758	459.60	0.042988	232.62	0.124732	80.17

Block size: 128

6.1.4 Grid en grupo pequeño + reducción en árbol sobre la métrica

Esta estrategia (estrategia5_X.cu) consiste en lanzar el kernel con un grid de bloques del tamaño del grupo de vectores de menor tamaño (A). Cada bloque tendrá asignado un vector de A y hará las comparaciones de este vector con todos los del grupo grande (B).

Además, cada bloque tendrá un número de threads configurable definido por la variable BLOCK_SIZE de forma que las dimensiones se reparten entre BLOCK_SIZE threads. La diferencia principal con las estrategias anteriores es que ahora aplicaremos una reducción en árbol (o varias, dependiendo del algoritmo) con el resultado obtenido entre dimensiones.

Por lo tanto, ahora la comparación también se hará de forma paralela.

RTX 3050	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.075500	132.45	0.093860	106.54	0.108567	92.11
Euclidean	0.048812	204.87	0.062170	160.85	0.078776	126.94
Pearson	0.093741	106.68	0.126447	79.08	0.148662	67.27
Pearson Opt	0.072007	138.88	0.092927	107.61	0.107249	93.24

Block size: 32

RTX 3060	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0,042672	234,34	0,048752	205,12	0,053001	188,68
Euclidean	0,031818	314,29	0,035036	285,42	0,03719	268,89
Pearson	0,055765	179,32	0,064942	153,98	0,075514	132,43
Pearson Opt	0,045502	219,77	0,048603	205,75	0,053473	187,01

Block size: 32

Block size: 64

RTX 4070 Ti SUPER	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.019577	510.80	0.024671	405.33	0.019959	501.03
Euclidean	0.011632	859.69	0.018681	535.31	0.013743	727.62
Pearson	0.023094	433.01	0.030379	329.17	0.029385	340.31
Pearson Opt	0.018870	529.95	0.023726	421.47	0.019954	501.15

Block size: 128

6.1.5 Grid en grupo grande + reducción en árbol sobre la métrica

Esta estrategia (estrategia6_X.cu) consiste en lanzar el kernel con un grid de bloques del tamaño del grupo de vectores de mayor tamaño (B). Cada bloque tendrá asignado un vector de B y hará las comparaciones de este vector con todos los del grupo pequeño (A). Además, cada bloque tendrá un número de threads configurable definido por la variable BLOCK_SIZE de forma que las dimensiones se reparten entre BLOCK_SIZE threads. La lógica de cálculo es análoga a la estrategia anterior, ya que aplicaremos una reducción en árbol (o varias, dependiendo del algoritmo) con el resultado obtenido entre dimensiones. Por lo tanto, ahora la comparación también se hará de forma paralela.

RTX 3050	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.032512	307.58	0.048692	205.37	0.065785	152.01
Euclidean	0.024949	400.82	0.048081	207.98	0.065523	152.62
Pearson	0.026103	383.10	0.050307	198.78	0.068246	146.53

Block size: 31

RTX 3060	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0,030587	326,93	0,03463	288,77	0,038608	259,02
Euclidean	0,025567	391,13	0,028468	351,27	0,033667	297,02
Pearson	0,044806	223,18	0,056359	177,44	0,064656	154,66
Pearson Opt	0,035647	280,53	0,04282	233,54	0,047623	209,98

Block size: 32

RTX 4070 Ti SUPER	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.012222	818.20	0.014900	671.15	0.015909	628.59
Euclidean	0.007789	1283.92	0.011416	875.96	0.013323	750.56
Pearson	0.017969	556.51	0.023576	424.16	0.027453	364.25
Pearson Opt	0.013210	757.02	0.016737	597.48	0.017807	561.59

Block size: 128

6.1.6 2D Tiled

Esta estrategia (estrategia7_X.cu) consiste en lanzar el kernel con un grid de bloques bidimensional que cubre el tamaño de ambos grupos de vectores (A y B). Cada hilo tendrá asignado un par específico de vectores (uno de A y uno de B) y realizará la comparación completa entre ellos. Además, cada bloque tendrá una dimensión configurable definida por la variable TILE, de forma que los threads se organizan en bloques cuadrados de TILE x TILE. La diferencia principal con las estrategias anteriores es que ahora aplicaremos la técnica de *Tiling*, cargando fragmentos de las dimensiones en memoria compartida para ser reutilizados por todos los threads del bloque. Por lo tanto, ahora la comparación se optimiza maximizando el ancho de banda mediante la reutilización de datos.

RTX 3050	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.031288	319.61	0.051086	195.75	0.068830	145.29
Euclidean	0.025388	393.89	0.050788	196.90	0.068413	146.17
Pearson	0.026454	378.01	0.052722	189.67	0.070679	141.48

Tile: 31

RTX 3060	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0,011756	850,6	0,02143	466,63	0,028301	353,34
Euclidean	0,01024	976,53	0,020432	489,44	0,026351	379,49
Pearson	0,009984	1001,62	0,019892	502,72	0,02675	373,83

Tile: 30

RTX 4070 Ti SUPER	384 dimensions		768 dimensions		1024 dimensions	
	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)	Avg Time (s)	Avg Throughput (M ops/s)
Cosine	0.004377	2284.59	0.008679	1152.22	0.011063	903.89
Euclidean	0.004338	2305.01	0.008404	1189.89	0.011439	874.22
Pearson	0.004456	2244.02	0.008809	1135.16	0.011462	872.45

Tile: 30

7. Comparación

7.1 Comparación CPU/GPU

El primer análisis que vamos a realizar es la comparación entre las ejecuciones en CPU y vamos a compararlas con las realizadas en GPU. Tomando como referencia la ejecución en serie en CPU la paralelización a nivel de queries es la única estrategia que aporta mejoras significativas ya que la versión en CPU que utiliza reducción en la métrica no se paraleliza correctamente, teniendo un speedup entre $0.02\times$ y $0.08\times$ respecto a la versión secuencial. El problema de la reducción en la métrica por CPU es que el overhead introducido por la paralelización es mayor que lo que se puede llegar a ganar repartiendo las sumas del sumatorio del cálculo de la métrica en diferentes threads. Repartiéndolo conjuntos de vectores en diferentes threads para hacer las comparaciones (paralelización a nivel de query) logramos speedups de entre $3\times$ y $13\times$ para las métricas Coseno y Euclídea, y de hasta $\sim 17\times$ en el caso de Pearson para vectores de mayor dimensionalidad. Hemos visto que el speedup no escala de forma proporcional con la dimensionalidad. Aunque el número de operaciones aumenta el rendimiento queda limitado por el número fijo de hilos disponibles en la CPU (20 hilos), el acceso a memoria y la naturaleza secuencial del cálculo interno de la métrica.

Si comparamos la mejor estrategia en CPU (paralelización a nivel de queries) con la mejor estrategia en GPU (2D Tiled) podemos ver speedups de entre $16\times$ y $38\times$ a favor de la GPU, dependiendo de la dimensionalidad del vector. El mayor speedup se obtiene en vectores de menor dimensión donde la GPU consigue tiempos extremadamente bajos mientras que en dimensiones altas la diferencia sigue siendo muy significativa. Si hacemos la comparación frente a la versión de CPU secuencial los speedups alcanzan valores de entre $180\times$ y más de $500\times$ llegando incluso a superar $550\times$ en el caso de la RTX 4070 Ti SUPER. Estos resultados ponen de manifiesto que para escenarios con millones de comparaciones la GPU no solo mejora el rendimiento sino que cambia completamente el orden de magnitud del tiempo de ejecución.

Podemos ver que en este tipo de tareas las optimizaciones en CPU ofrecen mejoras limitadas y rápidamente saturan los recursos disponibles. En cambio, la GPU muestra una escalabilidad muy superior tanto entre estrategias internas como frente a la CPU. La combinación de paralelismo masivo, uso de memoria compartida y técnicas de tiling permite obtener mejoras de uno a dos órdenes de magnitud consolidando a la GPU como la plataforma adecuada para cargas intensivas de similitud vectorial.

7.2 Comparación Estrategias GPU

En este apartado se analiza el rendimiento obtenido con las diferentes estrategias de paralelización en GPU. Para comparar las estrategias, se utilizan como referencia los tiempos de ejecución y las operaciones por segundo (*throughput* en Mops/s) obtenidos en la RTX 3060, este comportamiento se puede asumir equivalente en el resto de tarjetas que hemos estudiado.

A continuación, detallaremos la mejora que introduce cada estrategia respecto a la anterior y las limitaciones que presentan.

Estrategia 1 (S1): Grid AxB, cada bloque compara un vector de A y uno de B

Enfoque: Se asigna un bloque para calcular la similitud de cada par de vectores (un vector de la Query-A contra un vector de la Base de Datos-B). La configuración del kernel utiliza bloques de un solo thread (block(1)).

Limitación: Esta aproximación provoca un derroche crítico de recursos. La arquitectura de NVIDIA planifica la ejecución en grupos de 32 hilos (warps). Al configurar bloques de un solo hilo, el hardware reserva recursos para un warp, pero mantiene 31 threads inactivos en cada ciclo, desperdimando aproximadamente el 97% de la capacidad de procesamiento. Además, ambos vectores se leen desde VRAM, lo cual también resulta limitante.

Rendimiento: Es la estrategia de menor rendimiento y la usaremos como base para la comparación.

Estrategia 2 (S2): Paralelismo sobre el Grupo de Consultas (Queries)

Mejora sobre S1: Se corrige el problema de ocupación de los warps. Se asigna un bloque a cada vector del conjunto Queries (A) con una cantidad de hilos ≥ 32 . Cada hilo del bloque itera sobre la Base de Datos (B) para calcular la similitud con todos los vectores de B de forma secuencial. Esta estrategia, al lanzar el kernel con warps activos al 100%, aprovecha la capacidad SIMT (Single Instruction Multiple Threads) de la GPU.

Limitación: El cuello de botella se desplaza del malgasto de recursos a la memoria. Cada hilo lee de forma independiente los vectores (su vector de A y todos los de B) de la VRAM para cada cálculo, sin reutilización de datos, saturando rápidamente el ancho de banda de memoria.

Rendimiento: Esto resulta en una mejora de rendimiento de aproximadamente 3.8x respecto a S1.

Estrategia 3 (S3): Paralelismo sobre la Base de Datos

Mejora sobre S2: Inversamente a la Estrategia 2, se asigna un bloque de hilos a cada vector de la Base de Datos (B), cada hilo compara ese vector con las distintas Queries (A). En este caso la mejora es mínima porque tenemos el mismo problema de saturación de memoria. No obstante, al B ser un conjunto con más vectores (en el caso estudiado 10 veces más) se explota mejor el paralelismo al lanzar más bloques y reducir el tamaño del

bucle secuencial que lanza cada uno. Esta estrategia solo mejora S2 si hay una diferencia de tamaño significativa entre A y B, siempre que B sea más grande.

Limitación: Misma limitación que S2.

Rendimiento: En el caso estudiado el rendimiento es solo un poco superior a S2, llegando a aproximadamente a una mejora de 4.6x respecto a S1.

Estrategia 5 (S5): Reducción Paralela con Caché de Queries

Mejora sobre S2: Esta estrategia es análoga a S2, pero introduce el uso de shared memory. Se asigna un bloque a cada Query (A), como en S2, pero se aplica lo siguiente:

Caching: Cada bloque carga el vector Query en la Memoria Compartida (Shared Memory) una sola vez. Cada hilo del bloque participa en cargar un fragmento del vector en la memoria compartida. Una vez el vector está cargado, todos los hilos se sincronizan y leen este vector desde la caché rápida mientras iteran sobre la Base de Datos reduciendo significativamente el tráfico hacia VRAM.

Reducción: El algoritmo de comparación no lo ejecuta un solo hilo, sino que se distribuye entre todos los hilos del bloque mediante un algoritmo de reducción en árbol en memoria compartida. Este cambio disminuye la presión sobre el bus de memoria y permite que la parte computacional se paralelice, esto es especialmente interesante para vectores de muchas dimensiones.

Limitación: Al mapear un bloque por Query, si el número de consultas es bajo (en nuestras pruebas 1.000), se generan pocos bloques respecto a la capacidad real de la gráfica, lo que puede resultar en una baja ocupación de los multiprocesadores en GPUs de gama alta como las que hemos utilizado. Además, la S5 trae los vectores de B, no los tiene en shared memory. Este conjunto, al ser relativamente grande, no cabe en la caché de la gráfica (al menos en la RTX 3060, que tiene una caché L2 de entre 3-4 MB) por lo tanto, al querer traer los vectores de B está teniendo que ir a por ellos a VRAM constantemente porque no puede tenerlos todos en caché. Esto provoca que no se pueda aprovechar al máximo un tamaño más óptimo como 32, ya que entre la “poca” cantidad de bloques y los fallos de caché (leer de VRAM) se ve muy penalizado por las esperas y no se mitiga bien la latencia de memoria. Con los vectores de 384 dimensiones llega a ser mejor 32 que 64, pero en cuanto vamos subiendo las dimensiones tenemos que recurrir a 64 para encontrar el mejor rendimiento. Esto se debe a que logra mitigar mejor las esperas que se sufren al tener que traer B, ya que cada bloque puede lanzar 2 warps.

Estrategia 6: Reducción Paralela con Caché de Base de Datos

Mejora sobre S5: Mantiene la lógica (Caché + Reducción) de la S5, pero asigna un bloque a cada vector de la Base de Datos (B), análoga a S3. Además, lo que se carga en shared memory son los vectores de B por lo que se traen menos vectores de VRAM en comparación a S5. Al ser la Base de Datos mucho mayor que el conjunto de Queries (A), se generan más bloques (en el caso estudiado 10.000), esto hace que la GPU esté más saturada de trabajo. Además, esta estrategia permite la optimización de hardware que la S5 no lograba explotar (al menos en la RTX 3060): usar un tamaño de bloque de 32 hilos, el tamaño físico de un warp. Esto se debe a que A tiene muchos menos vectores, por lo tanto, incluso el tamaño más grande de dimensiones ejecutado cabe entero o en su gran mayoría en la L2 de caché. Lo cual provoca que se mitiguen mucho las esperas originadas por memoria y que usar 32 hilos por bloque tenga un rendimiento mucho más cercano al ideal.

Al usar 32 hilos se permite la ejecución síncrona a nivel de hardware (lock-step), lo cual hace que “se salte” las sincronizaciones porque se garantiza que todos los hilos (del 0 al 31) del warp empiezan y acaban la instrucción lanzada al mismo tiempo.

Limitación: A pesar de que mitiga muchos de los problemas que presenta la S5, sigue teniendo la carga de que en memoria compartida realmente solo tiene el vector de B asignado a ese bloque. Realmente cada bloque está leyendo los vectores de A, incluso aunque los lean de la caché L2, sigue habiendo un límite de ancho de banda. Al fin y al cabo, aunque es una solución más inteligente, ya que se adapta mejor al contexto (grupo pequeño y grupo grande), sigue estando limitada por la memoria. Además, hay otra limitación que afecta a S5 también que habría que comentar, la reducción. Esta reducción permite parallelizar el algoritmo, pero esto no quita que en el momento en el que se llega a la reducción cada vez hay menos hilos trabajando y se añaden esperas. Por lo tanto, se dedican muchos recursos para obtener un solo dato (esto es particularmente importante para el cambio de paradigma de la S7).

Rendimiento: Se consigue un rendimiento muy superior en vectores de dimensión pequeña/media, alcanzando mejoras de entre 10x y 22x respecto a la Estrategia 1.

Estrategia 7: Tiling con Memoria Compartida

Mejora sobre S6: Esta estrategia cambia el paradigma sobre el que habíamos aplicado las anteriores estrategias, ya que no se basa en tener uno de los conjuntos (ya sea en caché, aplicando reducción o ninguna) y recorrer el otro. En lugar de cargar vectores completos, los bloques cargan submatrices (tiles) tanto de Queries como de la Base de Datos en la Memoria Compartida. Una vez en caché, se calculan todas las interacciones posibles entre esos fragmentos. Por lo tanto, cambiamos de un cálculo vector-vector a uno que es matriz-matriz. Ahora los hilos no se encargan de comparar un vector con otro y pasar al siguiente, sino que se encargan de calcular los valores de un tile de la matriz resultado. Concretamente, cada bloque tiene dos matrices en memoria compartida, una corresponde a valores de A y la otra a valores de B (de tamaño Tile x Tile). Cada hilo se encarga de una posición de estas dos matrices, hace el cálculo y guarda el resultado en la matriz resultado C. Además, como cada hilo accede a posiciones a las que no acceden el resto, estos pueden guardar los datos necesarios en sus propios registros y guardar el resultado final. Por lo tanto, en la fase de cálculo, no tienen por qué escribir en memoria compartida ni esperar a otro hilo (latencia 0).

Limitación: Este planteamiento desplaza el cuello de botella de la memoria a la capacidad de computación de la gráfica, que en este caso lanza muchos más bloques. Ahora cada hilo tiene que mantener los valores en sus registros, lo que limita cuantos warps puede lanzar realmente la GPU. Esta estrategia se beneficiará (escalará) de gráficas que tengan la capacidad de poder lanzar más warps al mismo tiempo, ya que ahí reside la presión ahora (tal y como se puede observar en la RTX 4070 ti super). Podemos observar que para los recursos de la RTX 3060 un tamaño de tile de 30 ha sido el que mejor rendimiento ha obtenido. Esto se debe a que es un punto dulce entre cantidad de hilos en ejecución, ocupación de memoria compartida y la dirección de bancos en memoria compartida (que se calcula sobre 32). Por este motivo, si ponemos 32 el rendimiento cae significativamente, ya que nuestro algoritmo recorre para una dimensión (columna) el valor en las diferentes filas, si ponemos 32 las columnas de las diferentes filas caen en el mismo banco de memoria compartida y la lectura se vuelve secuencial. Además, 32 lanzaría bloques de 1024 hilos, que es el límite de la RTX 3060. Por otro lado el tamaño 31 sería el mejor para solucionar el

problema de los bancos de memoria, no obstante se queda en un punto en el que la gráfica está demasiado saturada. En cambio, con tamaño 30 se lanzan bloques de 900 hilos, lo cual deja más libertad a la GPU para aliviar la presión en los registros.

Rendimiento Final: Es la estrategia más eficiente, logrando mejoras de 27x a 55x respecto a S1. La mejora es muy significativa, sobre todo en el algoritmo de correlación de Pearson, donde la fusión de cálculos en los tiles evita la repetición de lecturas sobre los mismos datos.

Resumen de Mejoras (Speedup) respecto a la Estrategia 1 (RTX 3060)

Estrategia	Dim 384	Dim 768	Dim 1024	Observación Principal
Estrategia 2/3	~3.8x	~4.5x	~4.6x	Uso eficiente de SIMT, pero saturación de VRAM.
Estrategia 5	~6.5x	~13.7x	~16.6x	Reduce tráfico VRAM, paraleliza algoritmo de comparación.
Estrategia 6	~9.1x	~19.3x	~22.9x	Alta ocupación y eficiencia de Warp (Block 32).
Estrategia 7	~23.6x	~31.2x	~31.2x	Máximo rendimiento por reutilización intensiva de datos (Tiling).

7.3 Comparación Ejecución en diferentes Gráficas

En este apartado analizamos el rendimiento de las estrategias propuestas al variar la GPU sobre las que se ejecutan. El objetivo es identificar qué cuellos de botella limitan a cada estrategia según los recursos disponibles.

1. Especificaciones Técnicas del Hardware

Es fundamental saber las diferencias físicas entre las GPUs, ya que factores como el tamaño de la caché L2 o el número de Multiprocesadores (SMs) son determinantes para explicar las variaciones de rendimiento observadas.

Característica	RTX 3050 Mobile	RTX 3060 Desktop	RTX 4070 Ti SUPER
Arquitectura	Ampere (GA107)	Ampere (GA106)	Ada Lovelace (AD103)
Streaming Multiprocessors (SMs)	16 SMs	28 SMs	66 SMs
CUDA Cores	2048	3584	8448
Ancho de Banda Memoria	192 GB/s (128-bit)	360 GB/s (192-bit)	672 GB/s (256-bit)
Caché L2	2 MB	3 MB	48 MB

Contexto de Datos: Los conjuntos de prueba constan de 10^4 vectores (*Database*) y 10^3 vectores (*Queries*) respectivamente.

- Tamaño Database (B) (768 dim, float): **30.7 MB**.

- Tamaño Queries (A) (768 dim, float): **3.07 MB**.

A continuación, diseccionamos el comportamiento de las estrategias S6 y S7 en función de las limitaciones físicas de cada GPU.

2. RTX 3050 Mobile:

Es la gráfica con menos recursos computacionales y de memoria, podemos observar que la Estrategia 7 (Tiling) no logra superar claramente a la Estrategia 6, llegando a rendir por debajo en ciertas configuraciones. Hemos deducido los siguientes motivos:

- **Limitación de caché L2 (2 MB)**: El tamaño de la caché L2 es insuficiente incluso para alojar el conjunto de Queries (3 MB). Esto provoca que la Estrategia 6, que depende de leer el conjunto B repetidamente, sufra fallos de caché constantes, teniendo que acudir a la VRAM.
- **Falta de recursos para mantener registros en S7**: La Estrategia 7 mitiga el problema del ancho de banda mediante *Tiling*, pero lo hace a costa de un alto consumo de registros por hilo para mantener las variables. Con solo 16 SMs disponibles, la alta demanda de registros limita el número de *Warps* que se pueden lanzar al mismo tiempo. Sin suficientes Warps en ejecución para ocultar la latencia de memoria, los núcleos terminan en estado de espera, impidiendo que la S7 obtenga el rendimiento esperado.

3. RTX 3060:

Como esta GPU tiene más recursos, tanto de memoria como a nivel computacional, la Estrategia 7 comienza a distanciarse a tener una mejor notoría respecto a la S6 (1.6x más rápida que S6 en 768 dim), aunque la S6 mantiene una eficiencia notable. Estas mejoras deducimos que se deben principalmente a:

- **Queries en L2 (3 MB)**: La caché L2 de la RTX 3060 coincide casi exactamente con el tamaño del conjunto de Queries. Esto beneficia a la Estrategia 6, ya que una vez cargadas las queries, los accesos a los vectores de A se hacen casi íntegramente desde la L2, reduciendo el cuello de botella de memoria.
- **Más recursos para S7**: A pesar de la mejora de la S6, la S7 gana porque reduce el tráfico de memoria para ambas matrices (A y B) mediante el uso de *Shared Memory*. Además, con 28 SMs, la tarjeta tiene suficientes recursos para lanzar más warps y gestionar la presión de registros de la S7 mitigando las esperas.

4. RTX 4070 Ti SUPER:

Esta GPU es de una gama superior, lo podemos ver por el cambio de arquitectura a Ada Lovelace, la cual aporta muchos recursos en comparación a las anteriores (más de 10x de tamaño de caché L2 y más de 2x de SMs entre otros factores). El rendimiento de la Estrategia 7 se dispara y alcanza mejoras de hasta 5.6x respecto a la RTX 3050 y superando a la S6 un 70% en algunos algoritmos. Estas mejoras deducimos que principalmente son causadas por:

- **Tamaño de la L2 (48 MB):** La caché L2 es mayor que toda la Base de Datos (30.7 MB) y las Queries (3.07 MB) juntas, lo cual permite eliminar los fallos de caché y las lecturas desde VRAM. Tras la primera iteración del kernel, todos los vectores (Queries y Database) están ya cargados en la caché L2. Este hecho provoca que la S6 se vea solamente limitada por el ancho de banda de la caché (y la reducción), aunque esta GPU tiene un ancho de banda mucho mayor que la RTX 3060/3050.
- **Capacidad superior de computación:** La Estrategia 7, que no tiene sincronizaciones en la fase de cálculo y almacena la información en registros, es la única capaz de saturar la gran cantidad de cores que tiene la RTX 4070 ti super. Esta GPU con sus 66 SMs y sus 8848 CUDA cores consigue lanzar muchos más warps simultáneamente, lo cual consigue un rendimiento muy superior al resto.

7.4 Conclusiones

La comparativa evidencia que la Estrategia 7 es con diferencia la solución que mejor escala y la que obtiene una velocidad mayor, no obstante, en gráficas que tienen menos recursos computacionales puede tener una limitación demasiado grande. Esta limitación sería una presión demasiado grande en los registros que impediría lanzar muchos warps a la vez. Por lo tanto, se puede ver superada por estrategias que planteen la mitigación de latencia de una forma distinta (como lanzar bloques con múltiples warps, caso de S5 o S6 con bloques de 64 hilos).

Concluimos que la **estrategia 7** es la mejor estrategia si los recursos de la gráfica cumplen un mínimo, en gráficas con menos recursos la mejor sería S6 siempre y cuando se cumpla la premisa de que $|B| >> |A|$. La estrategia 6 ofrece un rendimiento más robusto debido a su menor presión sobre los registros y su diseño orientado a reducir la latencia, su rendimiento será mayor en los casos en los que A quepa en la caché L2 de la gráfica.

Una vez determinado qué estrategia es la mejor, podemos analizar qué algoritmo en concreto ha conseguido el mejor rendimiento. Viendo los datos obtenidos, podemos asumir que el algoritmo que mejor rendimiento obtiene es **euclidean**. Observamos que es

precisamente porque es el algoritmo que menos operaciones y variables necesita guardar, esto alivia los registros en el caso de la S7 y el tiempo de computación de todas las estrategias en general. Además, en las estrategias que usan reducción esto es especialmente significativo, ya que el menor número de variables a sincronizar y con las que operar hace que esta sea mucho más rápida en comparación a las demás. Esto es a nivel general, es cierto que hay algunos casos concretos en los que en una gráfica o en un tamaño concreto otro algoritmo consigue un mejor rendimiento (Pearson en la S7-RTX 3060 para 384 y 768 o Cosine en S7-RTX 4070 Ti super para 1024). Debido a que en estos casos la mejora no es tan significativa y no es son más bien inusuales, asumiremos que Euclidean es la mejor. Nuestra hipótesis es que a nivel de compilación puede haber una planificación de instrucciones que para un número de operaciones aparentemente menor implique unas dependencias entre instrucciones no triviales. Por lo tanto, esto implicaría que este bloque que a primera vista supondríamos menor se ejecute de forma más lenta que otro conjunto de operaciones aparentemente mayor. Esto implicaría que en casos concretos de tamaño del vector o cantidad de recursos de la gráfica, otro algoritmo consiguiera un poco más de rendimiento sobre euclidean.

8. ANEXOS

8.1 Tablas ejecuciones (mejores resultados)

8.1.1 Ejecución en serie (CPU)

	384 dimensions			768 dimensions			1024 dimensions		
	run 1	run 2	run 3	run 1	run 2	run 3	run 1	run 2	run 3
Cosine	2.070784	2.027428	2.030493	4.534554	4.407653	4.521011	6.496704	5.892849	5.938289
Euclidean	1.348588	1.358950	1.287532	3.750274	3.214152	3.226560	4.442035	4.421221	4.436736
Pearson	3.712166	4.515661	3.595848	7.439849	8.391079	7.516142	9.986390	9.941816	10.034884

8.1.2 Paralelización a nivel de queries (CPU)

	384 dimensions			768 dimensions			1024 dimensions		
	run 1	run 2	run 3	run 1	run 2	run 3	run 1	run 2	run 3
Cosine	0.387241	0.504047	0.383932	0.331423	0.396379	0.527182	0.451674	0.445323	0.459898
Euclidean	0.333101	0.516454	0.382255	0.416350	0.567491	0.342122	0.421607	0.426791	0.571726
Pearson	0.317016	0.408022	0.392632	0.435515	0.480646	0.471618	0.605098	0.614719	0.587234

8.1.3 Paralelización de la métrica mediante reducción (CPU)

	384 dimensions			768 dimensions			1024 dimensions		
	run 1	run 2	run 3	run 1	run 2	run 3	run 1	run 2	run 3
Cosine	92.7842 48	98.729043	95.602206	94.464777	93.383552	257.88773 4	100.22115 6	102.45163 6	100.13302 8
Euclidean	57.7646 18	62.973702	52.247347	49.688144	56.913825	52.098213	57.230386	57.901266	59.191103
Pearson	186.594 263	204.08343 6	226.71456 2	198.67399 6	194.86812 3	196.70464 5	191.37908 6	194.94740 8	198.64735 1

8.1.4 Grid por pares + métrica en secuencial (GPU)

8.1.4.1 Vectores de 384 dimensiones

Dimensión del vector: 384 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.172187	0.156245	0.153763	0.154134	0.157181	0.156997	0.158895	0.158015	0.158498	0.162498
Euclidean	0.147617	0.146334	0.147467	0.145948	0.145518	0.149810	0.146641	0.145032	0.146610	0.146722
Pearson	0.348336	0.373457	0.350453	0.362279	0.363935	0.357926	0.350609	0.356937	0.358588	0.351491
Pearson Opt	0.159513	0.156092	0.153785	0.162996	0.157121	0.160453	0.156711	0.150816	0.149780	0.152358

Dimensión del vector: 384 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Euclidean	0,295067	0,294617	0,276726	0,273682	0,273442	0,273439	0,273432	0,273670	0,273677	0,273445
Cosine	0,280223	0,274075	0,289266	0,294915	0,293012	0,274487	0,272762	0,272768	0,272541	0,272532
Pearson	0,582975	0,54516	0,546393	0,547678	0,545806	0,545148	0,543794	0,539644	0,539884	0,539895
Pearson Opt	0,292276	0,292442	0,283117	0,27291	0,27238	0,27259	0,271783	0,272016	0,272006	0,271775

Dimensión del vector: 384 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.08308	0.08036	0.08122	0.08045	0.08041	0.08047	0.08131	0.08121	0.08208	0.082124
Euclidean	0.08465	0.08264	0.08213	0.08176	0.08181	0.08290	0.08172	0.08176	0.08169	0.083863
Pearson	0.16394	0.16370	0.16370	0.16368	0.16435	0.16386	0.16377	0.16387	0.16368	0.163929
Pearson Opt	0.08299	0.08235	0.08250	0.08258	0.08243	0.08235	0.08275	0.08244	0.08268	0.082611

8.1.4.2 Vectores de 768 dimensiones

Dimensión del vector: 768 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.328162	0.338388	0.326463	0.325327	0.333351	0.328065	0.326648	0.330506	0.329272	0.338333
Euclidean	0.355133	0.381842	0.366403	0.372712	0.377099	0.362713	0.361617	0.380068	0.375863	0.369677
Pearson	0.774550	0.761394	0.760149	0.750579	0.752408	0.762342	0.748255	0.761169	0.763907	0.761638
Pearson Opt	0.340211	0.365890	0.336950	0.335473	0.347537	0.349457	0.337290	0.347426	0.344766	0.345703

Dimensión del vector: 768 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,694695	0,665712	0,664953	0,664669	0,664484	0,664239	0,664684	0,664899	0,664281	0,665330

<i>Euclidean</i>	0,567504	0,597924	0,552707	0,553049	0,553937	0,552187	0,551946	0,552191	0,553001	0,552165
<i>Pearson</i>	1,108917	1,095283	1,096173	1,09713	1,095537	1,095927	1,096879	1,096852	1,095885	1,095466
<i>Pearson Opt</i>	0,695931	0,667681	0,668971	0,668047	0,668051	0,667199	0,668022	0,667824	0,667208	0,667471

Dimensión del vector: 768 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.163483	0.163828	0.163645	0.163620	0.163747	0.164100	0.163880	0.163734	0.163669	0.163781
Euclidean	0.163172	0.164142	0.163501	0.163477	0.163624	0.163552	0.163708	0.163875	0.163537	0.163459
Pearson	0.326981	0.326994	0.326795	0.326926	0.326782	0.326858	0.327012	0.326933	0.326710	0.327067
Pearson Opt	0.165340	0.165306	0.165867	0.165712	0.165418	0.165421	0.165449	0.165440	0.165446	0.165661

8.1.4.3 Vectores de 1024 dimensiones

Dimensión del vector: 1024 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.418569	0.426486	0.415656	0.427860	0.416942	0.419995	0.416672	0.418609	0.426796	0.419348
Euclidean	0.533570	0.501333	0.512390	0.498864	0.521577	0.512432	0.513211	0.515164	0.514166	0.516981
Pearson	1.022974	1.008415	1.021606	1.017724	1.023405	1.029702	1.019123	0.995605	1.024427	1.026822
Pearson Opt	0.457441	0.448702	0.449311	0.447136	0.456920	0.447998	0.436215	0.450654	0.451112	0.437819

Dimensión del vector: 1024 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,883027	0,882019	0,883009	0,882604	0,883248	0,882251	0,883419	0,882231	0,882800	0,882005
Euclidean	0,773181	0,740572	0,727786	0,728017	0,728231	0,729229	0,728109	0,728021	0,72857	0,730583
Pearson	1,444868	1,44109	1,44138	1,441687	1,442509	1,442043	1,441262	1,441998	1,441558	1,442201
Pearson Opt	0,887023	0,886361	0,886433	0,887018	0,886143	0,886365	0,886393	0,886222	0,885788	0,886086

Dimensión del vector: 1024 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.219414	0.219743	0.219717	0.219418	0.219450	0.219452	0.219772	0.219708	0.219431	0.219520
Euclidean	0.219048	0.219345	0.219468	0.219354	0.219617	0.219380	0.218620	0.218769	0.218845	0.218846
Pearson	0.437272	0.436978	0.437173	0.437217	0.437162	0.437326	0.436957	0.437494	0.437161	0.437220
Pearson Opt	0.219212	0.219025	0.219140	0.219024	0.219115	0.219206	0.219486	0.219085	0.219268	0.219068

8.1.5 Grid en grupo pequeño, loop en grupo grande + métrica en secuencial (GPU)

8.1.5.1 Vectores de 384 dimensiones

Dimensión del vector: 384 dim

RTX 3050	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10
Cosine	0.171557	0.133790	0.134546	0.136171	0.136095	0.135539	0.135281	0.138755	0.135510	0.135453
Euclidean	0.154561	0.137543	0.179473	0.145584	0.146804	0.163298	0.147627	0.152824	0.159804	0.138941
Pearson	0.288999	0.328355	0.339958	0.269219	0.266926	0.269463	0.333111	0.332785	0.265702	0.288521
Pearson Opt	0.132650	0.132665	0.132666	0.132713	0.132644	0.132699	0.132785	0.132701	0.133555	0.133561

Dimensión del vector: 384 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Euclidean	0,077926	0,076167	0,079963	0,076289	0,079647	0,076136	0,076297	0,075674	0,077720	0,076136
Cosine	0,073622	0,073541	0,073629	0,073166	0,073401	0,073552	0,072940	0,073944	0,073165	0,073662
Pearson	0,143441	0,141754	0,141904	0,141673	0,141761	0,141786	0,141777	0,141752	0,141741	0,142011
Pearson Opt	0,085349	0,079992	0,078515	0,076608	0,076838	0,076500	0,076843	0,076762	0,079373	0,075277

Dimensión del vector: 384 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.023819	0.023725	0.024090	0.023713	0.024036	0.023736	0.023619	0.023740	0.023690	0.023645
Euclidean	0.023200	0.023161	0.023130	0.023173	0.023138	0.023140	0.023142	0.023497	0.023286	0.023211
Pearson	0.044917	0.044756	0.044913	0.044809	0.044792	0.044981	0.044925	0.044981	0.044918	0.044728
Pearson Opt	0.022495	0.022467	0.022616	0.022663	0.022593	0.022490	0.022503	0.022630	0.022518	0.022603

8.1.5.2 Vectores de 768 dimensiones

Dimensión del vector: 768 dim

RTX 3050	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10
Cosine	0.385373	0.360908	0.437650	0.271409	0.367886	0.290242	0.272347	0.382601	0.339567	0.336321
Euclidean	0.561795	0.576738	0.568870	0.509531	0.598069	0.573984	0.571939	0.559848	0.510074	0.571619
Pearson	1.002717	0.997829	0.958084	1.039486	0.962653	1.010377	0.968141	1.021996	1.019170	1.057767

Pearson Opt	0.326435	0.325969	0.266492	0.324013	0.264009	0.287343	0.280461	0.264007	0.345533	0.263940
-------------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

Dimensión del vector: 768 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,158636	0,154579	0,150516	0,153881	0,150537	0,150361	0,150246	0,154432	0,154267	0,151936
Euclidean	0,150552	0,154018	0,159885	0,145689	0,153915	0,165655	0,166103	0,159627	0,166148	0,164910
Pearson	0,287295	0,288296	0,287333	0,286205	0,285618	0,286013	0,286037	0,286112	0,286607	0,286212
Pearson Opt	0,153954	0,153425	0,153390	0,153734	0,153284	0,160977	0,153641	0,153451	0,156977	0,153967

Dimensión del vector: 768 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.044706	0.044770	0.044665	0.044723	0.045028	0.045851	0.045528	0.045543	0.044844	0.044864
Euclidean	0.046540	0.046341	0.046436	0.046203	0.046262	0.046209	0.046162	0.046260	0.046224	0.046188
Pearson	0.090462	0.089928	0.089971	0.089906	0.089926	0.091559	0.092020	0.090298	0.089796	0.089869
Pearson Opt	0.046101	0.046275	0.046159	0.047503	0.048187	0.046825	0.046463	0.046023	0.046238	0.045984

8.1.5.3 Vectores de 1024 dimensiones

Dimensión del vector: 1024 dim

RTX 3050	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10
Cosine	0.732005	0.741553	0.735706	0.756891	0.775447	0.728624	0.689349	0.750699	0.748940	0.734554
Euclidean	0.832684	0.848493	0.869146	0.862099	0.878979	0.868005	0.854068	0.834019	0.902218	0.876570
Pearson	1.391821	1.533862	1.504786	1.572337	1.450900	1.628884	1.519826	1.572595	1.502745	1.502452
Pearson Opt	0.539353	0.572197	0.573620	0.572079	0.568773	0.529268	0.553486	0.549794	0.581698	0.578893

Dimensión del vector: 1024 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,201435	0,202082	0,20635	0,207381	0,208184	0,21305	0,212165	0,208169	0,212299	0,208009
Euclidean	0,208682	0,212414	0,208023	0,219012	0,223113	0,2173	0,221057	0,207884	0,208108	0,211652
Pearson	0,379319	0,378166	0,379111	0,380947	0,382409	0,384481	0,378745	0,380644	0,377657	0,379269
Pearson Opt	0,214439	0,214788	0,214968	0,215701	0,214675	0,214294	0,214815	0,215397	0,21484	0,214594

Dimensión del vector: 1024 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.059598	0.059654	0.059604	0.059784	0.059679	0.059685	0.059926	0.059787	0.059821	0.059884
Euclidean	0.060231	0.060136	0.060253	0.060192	0.060058	0.060277	0.060204	0.060168	0.060176	0.060237
Pearson	0.118928	0.119330	0.119279	0.119299	0.119149	0.119298	0.119368	0.119369	0.119223	0.119200
Pearson Opt	0.061001	0.061256	0.061345	0.061389	0.061201	0.061625	0.062347	0.061578	0.063092	0.061778

8.1.6 Grid en grupo grande, loop en grupo pequeño + métrica en secuencial (GPU)

8.1.6.1 Vectores de 384 dimensiones

Dimensión del vector: 384 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.170740	0.139307	0.139736	0.139931	0.139571	0.139577	0.140551	0.146969	0.141169	0.139962
Euclidean	0.142074	0.140040	0.141717	0.142378	0.141524	0.144533	0.141437	0.143503	0.142629	0.141647
Pearson	0.283364	0.288746	0.286652	0.279808	0.284075	0.288248	0.285115	0.283359	0.283525	0.281505
Pearson Opt	0.142836	0.142556	0.142509	0.142059	0.142027	0.147437	0.144130	0.142333	0.142841	0.142590

Dimensión del vector: 384 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Euclidean	0,073565	0,073583	0,073571	0,073850	0,073580	0,073609	0,073572	0,073602	0,073579	0,073581
Cosine	0,072970	0,072967	0,072949	0,073220	0,072964	0,072955	0,072962	0,073359	0,073717	0,073365
Pearson	0,145223	0,144915	0,145004	0,144982	0,145049	0,145026	0,144991	0,145483	0,145003	0,144914
Pearson Opt	0,073988	0,074020	0,074267	0,074012	0,074043	0,073996	0,074042	0,073969	0,074015	0,074080

Dimensión del vector: 384 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.023612	0.023586	0.023632	0.022877	0.022382	0.022237	0.022285	0.022431	0.022266	0.022434
Euclidean	0.021982	0.022151	0.022043	0.021962	0.021962	0.022044	0.021976	0.022096	0.022026	0.021977
Pearson	0.042697	0.042632	0.042677	0.042790	0.042862	0.042589	0.042925	0.042675	0.042653	0.042796
Pearson Opt	0.021595	0.022354	0.021701	0.021613	0.021748	0.021656	0.021672	0.021645	0.021636	0.021597

8.1.6.2 Vectores de 768 dimensiones

Dimensión del vector: 768 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.313750	0.334603	0.317153	0.314446	0.320662	0.318080	0.318850	0.320802	0.317621	0.317341
Euclidean	0.314296	0.310100	0.314814	0.303253	0.303026	0.310833	0.305878	0.297852	0.310280	0.320887
Pearson	0.721323	0.715431	0.702936	0.709077	0.718745	0.712031	0.706897	0.713636	0.713912	0.711940
Pearson Opt	0.317753	0.334230	0.314625	0.311099	0.321361	0.322632	0.314070	0.314446	0.324531	0.317173

Dimensión del vector: 768 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,146154	0,146030	0,146623	0,146376	0,146240	0,146026	0,146058	0,146056	0,146656	0,146117
Euclidean	0,144440	0,145431	0,145624	0,145479	0,145083	0,145421	0,145533	0,144639	0,144589	0,144329
Pearson	0,282712	0,282958	0,282700	0,282976	0,282993	0,282990	0,283370	0,282947	0,283287	0,283332
Pearson Opt	0,144150	0,144785	0,144506	0,144914	0,144728	0,145388	0,145408	0,145310	0,144306	0,144933

Dimensión del vector: 768 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8*	Run 9	Run 10
Cosine	0.044539	0.044836	0.044558	0.044519	0.044490	0.044506	0.044523	0.687199	0.045784	0.045428
Euclidean	0.046361	0.046310	0.045172	0.044600	0.045097	0.044593	0.046716	0.045812	0.046219	0.046132
Pearson	0.086814	0.085777	0.085195	0.085311	0.085104	0.085144	0.085124	0.085140	0.085234	0.085128
Pearson Opt	0.042860	0.042940	0.042852	0.043005	0.043136	0.042947	0.042795	0.042911	0.043124	0.043170

Outlier en run 8.

8.1.6.3 Vectores de 1024 dimensiones

Dimensión del vector: 1024 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.416380	0.423222	0.410911	0.425368	0.421710	0.418638	0.423227	0.418770	0.423128	0.417494
Euclidean	0.451281	0.415138	0.429456	0.452763	0.435323	0.432465	0.428601	0.415941	0.449304	0.421204
Pearson	0.962707	0.950873	0.961899	0.959246	0.970466	0.969948	0.959199	0.968867	0.967651	0.970361
Pearson Opt	0.417031	0.430302	0.419652	0.426501	0.417492	0.423612	0.426340	0.415106	0.422539	0.417184

Dimensión del vector: 1024 dim

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,188735	0,189036	0,188811	0,189543	0,188922	0,189043	0,188587	0,189125	0,19003	0,188856
Euclidean	0,188981	0,188624	0,188804	0,18908	0,189775	0,189229	0,189061	0,189402	0,189591	0,189165
Pearson	0,374202	0,373646	0,373035	0,374781	0,373279	0,373029	0,374568	0,373058	0,373892	0,373667
Pearson Opt	0,190733	0,190321	0,192063	0,191469	0,190634	0,190683	0,190059	0,191354	0,191123	0,190002

Dimensión del vector: 1024 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.058215	0.058218	0.058271	0.058275	0.058970	0.058311	0.058196	0.058326	0.058600	0.058549
Euclidean	0.057484	0.057469	0.057513	0.057437	0.057558	0.057699	0.057485	0.057622	0.057831	0.057787
Pearson	0.113370	0.113625	0.113677	0.113950	0.113637	0.113645	0.113639	0.113658	0.113772	0.113727
Pearson Opt	0.057072	0.057066	0.057316	0.057162	0.057149	0.057189	0.057234	0.057164	0.057295	0.057276

8.1.7 Grid en grupo pequeño + reducción en árbol sobre la métrica (GPU)

8.1.7.1 Vectores de 384 dimensiones

Dimensión del vector: 384 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.084986	0.080301	0.080250	0.074272	0.072573	0.072382	0.072502	0.072581	0.072622	0.072527
Euclidean	0.048819	0.048771	0.048867	0.048856	0.048816	0.048758	0.048775	0.048829	0.048801	0.048825
Pearson	0.092626	0.092743	0.092791	0.092697	0.092717	0.092715	0.092691	0.094159	0.100133	0.094133
Pearson Opt	0.071891	0.071865	0.071910	0.071888	0.071862	0.071909	0.071915	0.071908	0.073120	0.071798

Dimensión del vector: 384 dim / Block size: 32

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,058173	0,057579	0,058561	0,058267	0,057912	0,058146	0,058423	0,058343	0,060293	0,061245
Euclidean	0,031884	0,031866	0,031920	0,032192	0,031852	0,031955	0,031959	0,030632	0,030021	0,030031
Pearson	0,055455	0,058158	0,057353	0,053275	0,053555	0,053240	0,053117	0,053069	0,053108	0,054766
Pearson Opt	0,045287	0,045370	0,045499	0,045546	0,045205	0,045449	0,044479	0,042213	0,042162	0,042255

Dimensión del vector: 384 dim

RTX 4070 Ti	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10

SUPER										
Cosine	0.019013	0.019220	0.019066	0.021568	0.019041	0.019046	0.019862	0.019011	0.019116	0.020831
Euclidean	0.011567	0.011649	0.011585	0.011571	0.011548	0.011598	0.011811	0.011750	0.011564	0.011678
Pearson	0.023076	0.023242	0.024035	0.022823	0.022951	0.022946	0.022997	0.022835	0.023087	0.022948
Pearson Opt	0.018575	0.018650	0.020448	0.018608	0.018613	0.018728	0.018976	0.018621	0.018780	0.018697

8.1.7.2 Vectores de 768 dimensiones

Dimensión del vector: 768 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.094154	0.093221	0.093299	0.093321	0.093244	0.093204	0.093185	0.094027	0.096908	0.094038
Euclidean	0.061777	0.062206	0.062227	0.062181	0.061995	0.062434	0.062425	0.062169	0.062147	0.062143
Pearson	0.125950	0.125403	0.124829	0.124125	0.125447	0.134001	0.128630	0.125765	0.124606	0.125714
Pearson Opt	0.091731	0.091812	0.091731	0.091878	0.091739	0.091730	0.091729	0.092636	0.101856	0.092428

Dimensión del vector: 768 dim / Block size: 64

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,051669	0,050879	0,051179	0,051523	0,051565	0,050824	0,050964	0,051110	0,051214	0,050838
Euclidean	0,036238	0,036398	0,035590	0,035760	0,036215	0,034174	0,033839	0,034208	0,033990	0,033782
Pearson	0,062218	0,063607	0,062300	0,062775	0,062644	0,062645	0,062647	0,063113	0,062853	0,062949
Pearson Opt	0,047289	0,047633	0,048173	0,047401	0,048131	0,048055	0,048144	0,047824	0,047787	0,047800

Dimensión del vector: 768 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.024558	0.024871	0.024437	0.024474	0.024280	0.024559	0.024988	0.024232	0.025349	0.024964
Euclidean	0.018005	0.018854	0.019762	0.019007	0.018953	0.018180	0.018782	0.019106	0.018210	0.017950
Pearson	0.030163	0.031179	0.029856	0.030693	0.030485	0.030730	0.030316	0.029950	0.030278	0.030143
Pearson Opt	0.023666	0.024526	0.024720	0.023221	0.023402	0.023590	0.023456	0.023440	0.023632	0.023607

8.1.7.3 Vectores de 1024 dimensiones

Dimensión del vector: 1024 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.107776	0.107437	0.107344	0.107225	0.107636	0.107705	0.111227	0.113732	0.107861	0.107732
Euclidean	0.078569	0.078488	0.078633	0.078892	0.078198	0.078757	0.078538	0.078374	0.078444	0.080863
Pearson	0.147449	0.147182	0.147732	0.147656	0.155772	0.149941	0.147686	0.147523	0.147687	0.147989
Pearson Opt	0.105418	0.105003	0.105264	0.105215	0.105897	0.105552	0.110108	0.119656	0.105036	0.105345

Dimensión del vector: 1024 dim / Block size: 64

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,054634	0,056267	0,05805	0,056212	0,055257	0,055752	0,05664	0,053508	0,056952	0,056715
Euclidean	0,038843	0,038846	0,039007	0,036609	0,036377	0,036581	0,036902	0,036972	0,036751	0,037647
Pearson	0,073400	0,071247	0,071922	0,071929	0,072681	0,072445	0,071928	0,072966	0,073470	0,072418
Pearson Opt	0,052363	0,052277	0,052369	0,052532	0,052603	0,052681	0,052970	0,052823	0,052776	0,052837

Dimensión del vector: 1024 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.019921	0.019932	0.019946	0.019895	0.019850	0.019995	0.020567	0.019723	0.019856	0.019902
Euclidean	0.013500	0.013549	0.013716	0.013856	0.013538	0.013826	0.013853	0.013951	0.013870	0.013775
Pearson	0.028800	0.029080	0.029755	0.029279	0.029563	0.030048	0.029321	0.029327	0.029506	0.029174
Pearson Opt	0.019955	0.019900	0.019856	0.020066	0.019858	0.019573	0.020188	0.020041	0.020051	0.020055

8.1.8 Grid en grupo grande + reducción en árbol sobre la métrica (GPU)

8.1.8.1 Vectores de 384 dimensiones

Dimensión del vector: 384 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.073360	0.069635	0.068852	0.067838	0.056057	0.056060	0.056079	0.056072	0.056055	0.056091
Euclidean	0.044641	0.044643	0.044650	0.044649	0.044640	0.044644	0.044646	0.044634	0.044646	0.044634

Pearson	0.081784	0.081778	0.081775	0.081789	0.081797	0.081806	0.081794	0.081804	0.081817	0.081798
Pearson Opt	0.064565	0.064568	0.064588	0.064585	0.064565	0.064577	0.064595	0.064566	0.064581	0.064589

Dimensión del vector: 384 dim / Block size: 32

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,032872	0,032853	0,032862	0,032867	0,032991	0,032883	0,032875	0,032978	0,032866	0,032850
Euclidean	0,025865	0,025880	0,025607	0,024321	0,024306	0,024338	0,024309	0,024339	0,024311	0,024320
Pearson	0,044621	0,044621	0,044642	0,044624	0,044626	0,044624	0,044623	0,044997	0,045011	0,045189
Pearson Opt	0,035467	0,035453	0,035418	0,035443	0,035479	0,035411	0,035429	0,035407	0,035422	0,035440

Dimensión del vector: 384 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.012165	0.012139	0.012109	0.013275	0.012083	0.012077	0.012111	0.012210	0.012241	0.011809
Euclidean	0.007756	0.007964	0.007762	0.007731	0.007750	0.007913	0.007799	0.007732	0.007705	0.007775
Pearson	0.017930	0.018177	0.017944	0.017900	0.017910	0.017901	0.018017	0.018061	0.017907	0.017944
Pearson Opt	0.013267	0.013176	0.013168	0.013261	0.013120	0.013287	0.013314	0.013193	0.013157	0.013154

8.1.8.2 Vectores de 768 dimensiones

Dimensión del vector: 768 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.061757	0.061762	0.061787	0.061777	0.061828	0.061756	0.061764	0.061806	0.061973	0.062231
Euclidean	0.049714	0.049719	0.049727	0.049718	0.049737	0.049738	0.049727	0.049732	0.049719	0.049760
Pearson	0.096422	0.096449	0.096327	0.096476	0.096472	0.096854	0.096775	0.097287	0.097081	0.096642
Pearson Opt	0.071818	0.071842	0.071750	0.071826	0.071826	0.071822	0.071895	0.071801	0.071873	0.071569

Dimensión del vector: 768 dim / Block size: 32

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,037435	0,037241	0,036787	0,037225	0,037235	0,037390	0,036816	0,037295	0,037268	0,037471

Euclidean	0,029738	0,029764	0,029746	0,029749	0,029741	0,027650	0,027658	0,027621	0,027603	0,027592
Pearson	0,057365	0,062872	0,062319	0,062967	0,062927	0,062254	0,061339	0,061752	0,061682	0,061742
Pearson Opt	0,046715	0,047128	0,045748	0,042695	0,042729	0,042702	0,042940	0,042688	0,042708	0,042678

Dimensión del vector: 768 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.014755	0.014793	0.014941	0.014939	0.014876	0.014795	0.014934	0.015122	0.014996	0.014846
Euclidean	0.011091	0.011071	0.011315	0.011373	0.011403	0.011428	0.011406	0.011345	0.011344	0.012385
Pearson	0.023576	0.023434	0.023623	0.023571	0.023545	0.023917	0.023578	0.023498	0.023528	0.023491
Pearson Opt	0.016536	0.016736	0.016703	0.016833	0.016706	0.016739	0.016737	0.016809	0.016862	0.016709

8.1.8.3 Vectores de 1024 dimensiones

Dimensión del vector: 1024 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.070102	0.070007	0.070087	0.070148	0.070327	0.070161	0.070102	0.070099	0.070336	0.070254
Euclidean	0.058330	0.058068	0.058050	0.058090	0.058127	0.058172	0.058203	0.058082	0.058021	0.058091
Pearson	0.111412	0.111047	0.111112	0.111131	0.111189	0.111222	0.112567	0.114189	0.112058	0.111467
Pearson Opt	0.082254	0.081897	0.082122	0.082199	0.082110	0.082184	0.082175	0.082639	0.082079	0.083787

Dimensión del vector: 1024 dim / Block size: 32

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,041496	0,038437	0,038446	0,038406	0,038491	0,038382	0,038492	0,038373	0,038632	0,038414
Euclidean	0,034544	0,034526	0,03452	0,034514	0,034481	0,034502	0,034507	0,032001	0,031504	0,03197
Pearson	0,071454	0,066187	0,06471	0,064722	0,064934	0,064751	0,065317	0,071395	0,070971	0,069612
Pearson Opt	0,051524	0,052171	0,052806	0,052566	0,050371	0,04794	0,047655	0,047654	0,047661	0,047677

Dimensión del vector: 1024 dim

RTX 4070 Ti SUPER	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10

Cosine	0.015489	0.015651	0.015802	0.015901	0.016066	0.015794	0.016062	0.015879	0.016616	0.015827
Euclidean	0.012792	0.013099	0.013476	0.013607	0.013445	0.013450	0.013294	0.013337	0.013348	0.013389
Pearson	0.026876	0.027625	0.027635	0.027583	0.027404	0.027544	0.027542	0.027285	0.027582	0.027460
Pearson Opt	0.017445	0.017603	0.018109	0.017759	0.017726	0.017808	0.017971	0.017793	0.017921	0.017934

8.1.9 2D Tiled (GPU)

8.1.9.1 Vectores de 384 dimensiones

Dimensión del vector: 384 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.033067	0.034935	0.034927	0.033626	0.032058	0.032060	0.032053	0.032048	0.032047	0.028301
Euclidean	0.024850	0.024869	0.024852	0.024948	0.024992	0.024997	0.024994	0.024994	0.025001	0.024995
Pearson	0.026163	0.026096	0.026099	0.026098	0.026094	0.026089	0.026096	0.026101	0.026095	0.026101

Dimensión del vector: 384 dim / Tile size: 30

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Euclidean	0,010953	0,010944	0,010328	0,010953	0,010955	0,010323	0,010950	0,010955	0,010320	0,010946
Cosine	0,010254	0,010641	0,010265	0,010261	0,010252	0,010250	0,010250	0,010250	0,010253	0,010251
Pearson	0,010588	0,010928	0,010845	0,010437	0,010925	0,01092	0,010433	0,010911	0,010925	0,010438

Dimensión del vector: 384 dim

RTX 4070 Ti SUPER	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10
Cosine	0.004391	0.004382	0.004360	0.004372	0.004378	0.004379	0.004381	0.004375	0.004382	0.004371
Euclidean	0.004370	0.004342	0.004321	0.004328	0.004318	0.004335	0.004336	0.004371	0.004325	0.004338
Pearson	0.004455	0.004427	0.004479	0.004474	0.004452	0.004454	0.004428	0.004460	0.004439	0.004495

8.1.9.2 Vectores de 768 dimensiones

Dimensión del vector: 768 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
----------	-------	-------	-------	-------	-------	-------	-------	-------	-------	--------

Cosine	0.048812	0.048766	0.048671	0.048660	0.048660	0.048669	0.048669	0.048674	0.048658	0.048684
Euclidean	0.048239	0.048081	0.048084	0.048062	0.048058	0.048066	0.048057	0.048047	0.048043	0.048068
Pearson	0.050334	0.050239	0.050245	0.050254	0.050244	0.050250	0.050273	0.050422	0.050361	0.050443

Dimensión del vector: 768 dim / Tile size: 30

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,020788	0,021241	0,020771	0,020773	0,020922	0,020859	0,020768	0,020910	0,020792	0,020899
Euclidean	0,020435	0,020434	0,020435	0,020448	0,020438	0,020448	0,020457	0,020447	0,020443	0,020437
Pearson	0,021686	0,021955	0,021865	0,021698	0,022312	0,021684	0,022297	0,021677	0,021721	0,022256

Dimensión del vector: 768 dim

RTX 4070 Ti SUPER	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10
Cosine	0.008702	0.008809	0.008833	0.008736	0.008702	0.008672	0.008724	0.008653	0.008502	0.008456
Euclidean	0.008310	0.008340	0.008459	0.008438	0.008474	0.008457	0.008413	0.008344	0.008461	0.008346
Pearson	0.008744	0.008764	0.008844	0.008917	0.008814	0.008809	0.008766	0.008757	0.008762	0.008916

8.1.9.3 Vectores de 1024 dimensiones

Dimensión del vector: 1024 dim

RTX 3050	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0.065936	0.065685	0.065765	0.065822	0.065817	0.065746	0.065658	0.065786	0.065818	0.065813
Euclidean	0.065520	0.065530	0.065529	0.065517	0.065520	0.065513	0.065521	0.065529	0.065522	0.065524
Pearson	0.068408	0.068239	0.068300	0.068167	0.068326	0.068306	0.068305	0.068231	0.068093	0.068086

Dimensión del vector: 1024 dim / Tile size: 30

RTX 3060	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Cosine	0,025974	0,025984	0,025979	0,025979	0,02598	0,026002	0,02599	0,025985	0,025988	0,026108
Euclidean	0,027555	0,027544	0,027507	0,027512	0,027509	0,02752	0,027502	0,025237	0,026561	0,027659
Pearson	0,029771	0,029269	0,026607	0,026571	0,026572	0,026607	0,026741	0,02674	0,026731	0,02674

Dimensión del vector: 1024 dim

RTX 4070 Ti SUPER	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10
Cosine	0.011040	0.011033	0.011024	0.011102	0.011043	0.011007	0.011023	0.011059	0.011201	0.011103
Euclidean	0.011611	0.011528	0.011631	0.011571	0.011519	0.011572	0.011164	0.011160	0.011320	0.011311
Pearson	0.011492	0.011590	0.011732	0.011482	0.011438	0.011610	0.011475	0.011276	0.011262	0.011263