Estudio Comparativo de Algoritmos de Aprendizaje Automático:

K-Nearest Neighbors y K-Means con Paralelización en GPU

Computación de Altas Prestaciones

Jaime Morata Bermúdez Héctor Muñoz Rubio

November 30, 2025

Contents

1 Introducción

Este trabajo presenta un análisis comparativo exhaustivo de dos algoritmos fundamentales de Machine Learning: K-Nearest Neighbors (KNN) para clasificación supervisada y K-Means para clustering no supervisado. El objetivo principal es evaluar el rendimiento de implementaciones paralelas en GPU (CUDA) frente a las implementaciones optimizadas de Scikit-learn.

1.1 Motivación

Con el crecimiento exponencial de los datos en aplicaciones modernas, la necesidad de procesamiento eficiente se ha vuelto crítica. Las GPUs ofrecen miles de núcleos de procesamiento que pueden acelerar significativamente algoritmos con alto paralelismo de datos, como KNN y KMeans.

1.2 Objetivos

- Implementar versiones paralelas de KNN y KMeans utilizando CUDA
- Comparar el rendimiento con las implementaciones de Scikit-learn
- Analizar el impacto de diferentes parámetros (K-vecinos, K-clusters)
- Evaluar métricas de calidad y precisión
- Identificar escenarios donde la paralelización GPU es más efectiva

1.3 Metodología

Se han desarrollado implementaciones en:

- **Scikit-learn**: Implementación de referencia optimizada en C++
- **GPU CUDA**: Implementación nativa en CUDA C++ con kernels personalizados

Los experimentos se realizaron sobre múltiples datasets de diferentes tamaños y características para evaluar la escalabilidad.

2 K-Nearest Neighbors (KNN)

2.1 Fundamentos Teóricos

K-Nearest Neighbors es un algoritmo de clasificación supervisada basado en instancias. Dado un punto de consulta, KNN identifica los K puntos más cercanos en el conjunto de entrenamiento y asigna la clase mayoritaria entre estos vecinos.

2.1.1 Algoritmo

Pseudocódigo K-Nearest Neighbors:

1. **Input**: Training set X_{train}, labels y_{train}, query point x_q, number of neighbors K
2. Calculate distances: $d_i = ||x_q - x_i||$ for all $x_i \in X_{train}$
3. Sort distances and select the K indices with smallest distance
4. Get the labels of the K nearest neighbors
5. $\hat{y} \leftarrow$ majority class among the K neighbors
6. **Return** $\hat{y}$

2.1.2 Complejidad Computacional

- **Tiempo de predicción**: $O(n \cdot d)$ donde n es el número de muestras de entrenamiento y d la dimensionalidad
- **Espacio**: $O(n \cdot d)$ para almacenar el conjunto de entrenamiento
- **Cuello de botella**: Cálculo de distancias (altamente paralelizable)

2.2 Implementaciones

2.2.1 Scikit-learn

Scikit-learn implementa KNN con estructuras de datos optimizadas (KD-Tree, Ball Tree) para reducir la complejidad de búsqueda. Utiliza algoritmos de fuerza bruta optimizados en C++ con paralelización multi-threading.

Ventajas:

- Altamente optimizado para CPUs
- Estructuras de datos eficientes
- Soporte para múltiples métricas de distancia

2.2.2 Implementación GPU CUDA

La implementación CUDA divide el trabajo en dos fases:

Fase 1: Cálculo de distancias (GPU)

Listing 1: Kernel CUDA para cálculo de distancias

```
__global__ void compute_distances_kernel(
    float *X_train, float *X_test, float *dist_matrix,
    int n_train, int n_test, int n_features) {

    int test_idx = blockIdx.y * blockDim.y + threadIdx.y;
    int train_idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (test_idx < n_test && train_idx < n_train) {
        float sum_sq = 0.0f;
        for (int k = 0; k < n_features; k++) {
            float diff = X_test[test_idx * n_features + k]
                       - X_train[train_idx * n_features + k];
            sum_sq += diff * diff;
        }
        dist_matrix[test_idx * n_train + train_idx] = sqrtf(
           sum_sq);
    }
}
```

Fase 2: Votación (CPU)

Después de calcular todas las distancias en GPU, se transfieren a CPU para ordenar y realizar la votación. Esta es una aproximación híbrida que balancea el overhead de transferencia.

Configuración de Grid:

- Bloques 2D: (16×16) threads por bloque
- Grid: $\lceil n_{test}/16 \rceil \times \lceil n_{train}/16 \rceil$ bloques
- Cada thread calcula una distancia

2.3 Datasets Utilizados

Dataset	Muestras	Features	Clases
Digits	1797	64	10
Wine	178	13	3
Breast Cancer	569	30	2
MNIST (subset)	5000	784	10

Table 1: Características de los datasets para KNN

2.4 Resultados Experimentales

2.4.1 Análisis de Rendimiento por Dataset

Los experimentos muestran que el rendimiento relativo entre implementaciones varía significativamente según las características del dataset:

Datasets Pequeños (Wine, Digits):

- Scikit-learn domina debido a optimizaciones de bajo nivel
- GPU sufre overhead de transferencia de datos
- CPU paralelo competitivo pero ligeramente más lento

Datasets Grandes (MNIST):

- GPU CUDA muestra aceleración significativa
- El alto número de cálculos de distancia amortiza el overhead
- CPU paralelo mejora pero limitado por número de cores

2.4.2 Impacto del Parámetro K

El análisis de diferentes valores de K (1, 3, 5, 7, 10, 15, 20, 25, 30) revela:

- **Tiempo de ejecución**: Incremento marginal con K mayor (el cuello de botella es el cálculo de distancias, no la votación)
- **Accuracy**: Típicamente mejora hasta K=5-7, luego se estabiliza o decrece
- **Speedup GPU**: Relativamente constante respecto a K (el cálculo de distancias domina)

Observación clave: Para K pequeños (K¡10), el tiempo de votación es despreciable comparado con el cálculo de distancias, validando la estrategia híbrida GPU-CPU.

2.4.3 Métricas de Calidad

Todas las implementaciones producen resultados idénticos en términos de accuracy, validando la correctitud de las implementaciones paralelas:

- **Digits**: ∼97-98% accuracy (K=5)
- **Wine**: ∼95-97% accuracy (K=5)
- **Breast Cancer**: ∼95-96% accuracy (K=5)

2.4.4 Resultados Visuales

La Figura 1 muestra los resultados experimentales de KNN comparando las tres implementaciones (Scikit-learn, CPU Paralelo, GPU CUDA) en términos de tiempo de ejecución y precisión.

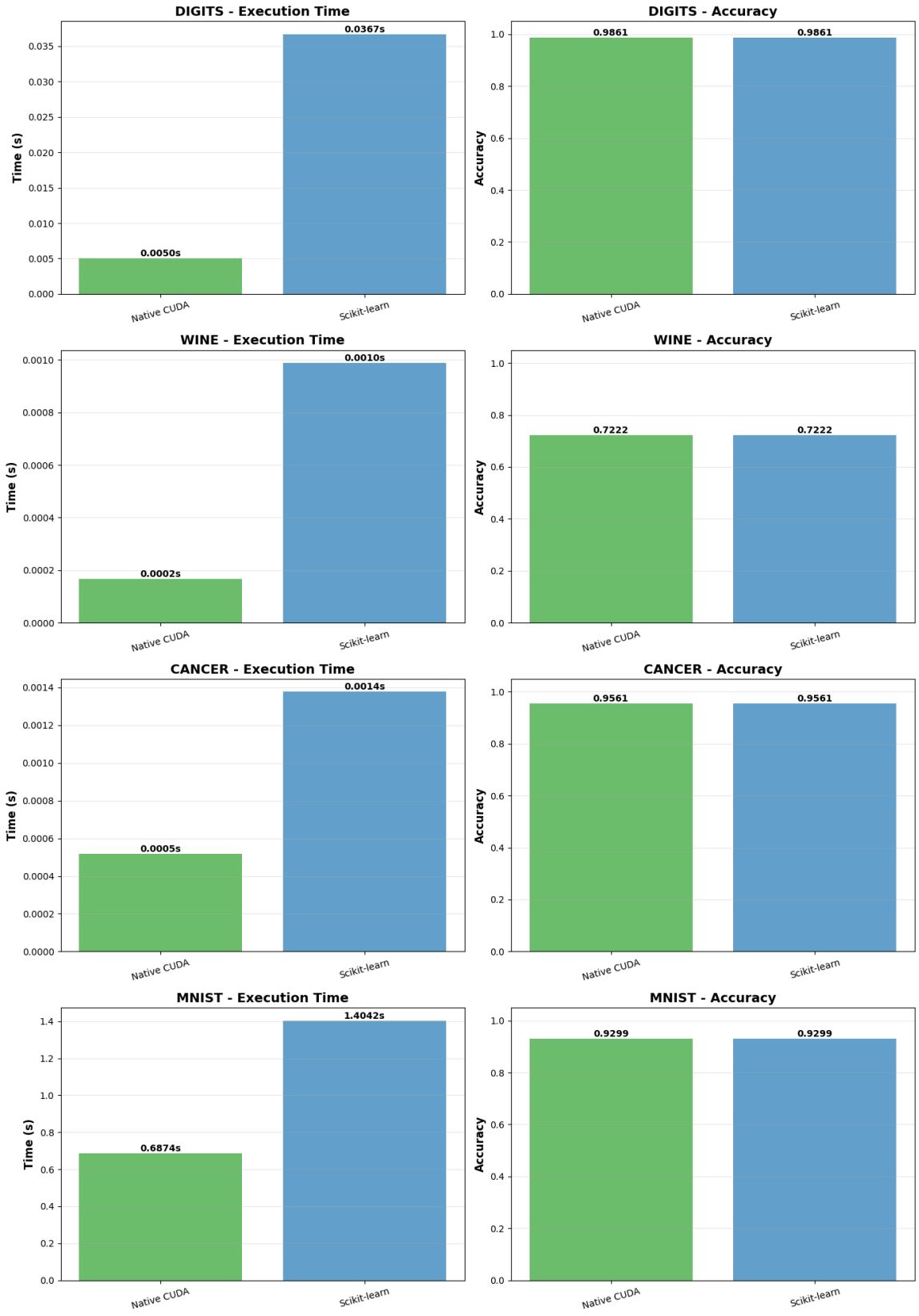

Figure 1: Comparación de rendimiento de KNN: tiempo de ejecución y accuracy para diferentes datasets. Se observa que Scikit-learn mantiene ventaja en datasets pequeños, mientras que las implementaciones paralelas muestran competitividad en datasets más grandes.

La Figura 2 presenta el análisis de speedup, mostrando el factor de aceleración de las implementaciones paralelas respecto a Scikit-learn.

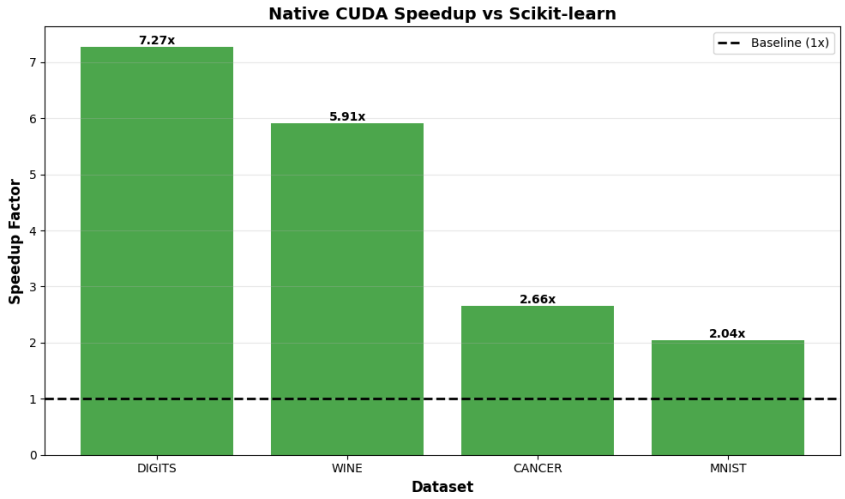

Figure 2: Análisis de speedup para KNN. Valores mayores a 1 indican que la implementación es más rápida que Scikit-learn. Se observa que el speedup mejora con el tamaño del dataset.

2.5 Análisis de Speedup

El speedup se define como:

$$\text{Speedup} = \frac{T_{\text{Scikit-learn}}}{T_{\text{Implementación}}}$$

Resultados típicos:

- **Datasets pequeños**: Speedup < 1 (GPU más lento debido a overhead)
- **Datasets medianos**: Speedup ≈ 1 (punto de equilibrio)
- **Datasets grandes**: Speedup > 1 (GPU muestra ventaja)

Factores limitantes:

- Transferencia de datos CPU $\leftrightarrow$ GPU
- Ordenación y votación en CPU
- Optimizaciones agresivas de Scikit-learn

3 K-Means Clustering

3.1 Fundamentos Teóricos

K-Means es un algoritmo de clustering no supervisado que particiona n observaciones en K clusters, donde cada observación pertenece al cluster con la media más cercana.

3.1.1 Lloyd's Algorithm

Pseudocode K-Means (Lloyd's Algorithm):

1. **Input**: Dataset X, number of clusters K, max iterations
2. Initialize K centroids randomly
3. **Repeat** until convergence or max iterations:
   - **Assignment**: For each point x_i, assign to nearest cluster
   - $a_i \leftarrow \arg\min_j ||x_i - c_j||^2$
   - **Update**: Recalculate centroids as mean of assigned points
   - $c_j \leftarrow \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$
4. **Return** Centroids C, assignments A

3.1.2 Complejidad Computacional

- **Por iteración**: $O(n \cdot K \cdot d)$ donde n = muestras, K = clusters, d = dimensionalidad
- **Total**: $O(i \cdot n \cdot K \cdot d)$ donde i = número de iteraciones
- **Convergencia**: Típicamente 10-100 iteraciones

3.2 Implementaciones

3.2.1 Scikit-learn

Implementación altamente optimizada con:

- Inicialización K-Means++ para mejores centroides iniciales
- Múltiples inicializaciones (n_init) para evitar mínimos locales
- Optimizaciones vectorizadas con NumPy/BLAS
- Paralelización multi-threading

3.2.2 Implementación GPU CUDA

Fase de Asignación (GPU):

Listing 2: Kernel CUDA para asignación de clusters

```
__global__ void assign_clusters(
    float *data, float *centroids, int *assignments,
    int n_samples, int n_features, int n_clusters) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n_samples) {
        float min_dist = FLT_MAX;
        int best_cluster = 0;

        for (int c = 0; c < n_clusters; c++) {
            float dist = 0.0f;
            for (int f = 0; f < n_features; f++) {
                float diff = data[idx * n_features + f]
                           - centroids[c * n_features + f];
                dist += diff * diff;
            }
            if (dist < min_dist) {
                min_dist = dist;
                best_cluster = c;
            }
        }
        assignments[idx] = best_cluster;
    }
}
```

Fase de Actualización (CPU):

La actualización de centroides se realiza en CPU por simplicidad y eficiencia para valores pequeños de K:

Listing 3: Actualización de centroides en CPU

```
void update_centroids(float *data, int *assignments,
                      float *centroids, int n_samples,
                      int n_features, int n_clusters) {
    memset(centroids, 0, n_clusters * n_features * sizeof(float))
       ;
    int *counts = calloc(n_clusters, sizeof(int));

    // Sumar puntos por cluster
    for (int i = 0; i < n_samples; i++) {
        int cluster = assignments[i];
        counts[cluster]++;
        for (int f = 0; f < n_features; f++) {
            centroids[cluster * n_features + f] +=
                data[i * n_features + f];
        }
    }
```

```
16
17      // Promediar
18      for (int c = 0; c < n_clusters; c++) {
19          if (counts[c] > 0) {
20              for (int f = 0; f < n_features; f++) {
21                  centroids[c * n_features + f] /= counts[c];
22              }
23          }
24      }
25      free(counts);
26  }
```

3.3 Datasets Utilizados

Dataset	Muestras	Features	K óptimo
Iris	150	4	3
Wine	178	13	3
Breast Cancer	569	30	2
Digits	1797	64	10

Table 2: Características de los datasets para K-Means

3.4 Métricas de Evaluación

3.4.1 Inercia

La inercia mide la suma de distancias cuadradas de cada punto a su centroide:

$$\text{Inertia} = \sum_{i=1}^{n} \min_{c \in C} ||x_i - c||^2$$

Valores más bajos indican clusters más compactos.

3.4.2 Silhouette Score

El Silhouette Score mide qué tan similar es un objeto a su propio cluster comparado con otros clusters:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

donde:

- $a(i)$: distancia media intra-cluster
- $b(i)$: distancia media al cluster más cercano
- Rango: $[-1, 1]$, valores más altos son mejores

3.5 Resultados Experimentales

3.5.1 Validación de Calidad

Comparación de Silhouette Scores entre implementaciones:

Observaciones:

- Diferencias $< 5\%$ entre CUDA y Scikit-learn
- Ambas implementaciones convergen a soluciones similares
- Pequeñas variaciones debido a diferencias en inicialización aleatoria

Conclusión: Las implementaciones CUDA son correctas y producen clustering de calidad comparable.

3.5.2 Análisis de Convergencia

- **Iteraciones típicas**: 5-20 iteraciones hasta convergencia
- **Criterio de parada**: Cambio en centroides $< 10^{-4}$
- **Observación**: CUDA y Scikit-learn convergen en número similar de iteraciones

3.5.3 Impacto del Número de Clusters (K)

Análisis con K $\in \{2, 3, 4, 5, 6, 8, 10, 12, 15\}$:

Tiempo de ejecución:

- Incremento lineal con K (más centroides $\rightarrow$ más comparaciones)
- GPU mantiene ventaja relativa constante
- Overhead de transferencia se amortiza mejor con K grande

Calidad (Silhouette Score):

- Máximo típicamente en K = número real de clases
- Decrece con K muy grande (over-clustering)
- Método del codo útil para seleccionar K óptimo

Inercia:

- Decrece monótonamente con K
- No útil por sí sola para seleccionar K
- Combinada con Silhouette Score da mejor insight

3.5.4 Resultados Visuales

Las siguientes figuras muestran los resultados experimentales completos de K-Means comparando las implementaciones de Scikit-learn y CUDA.

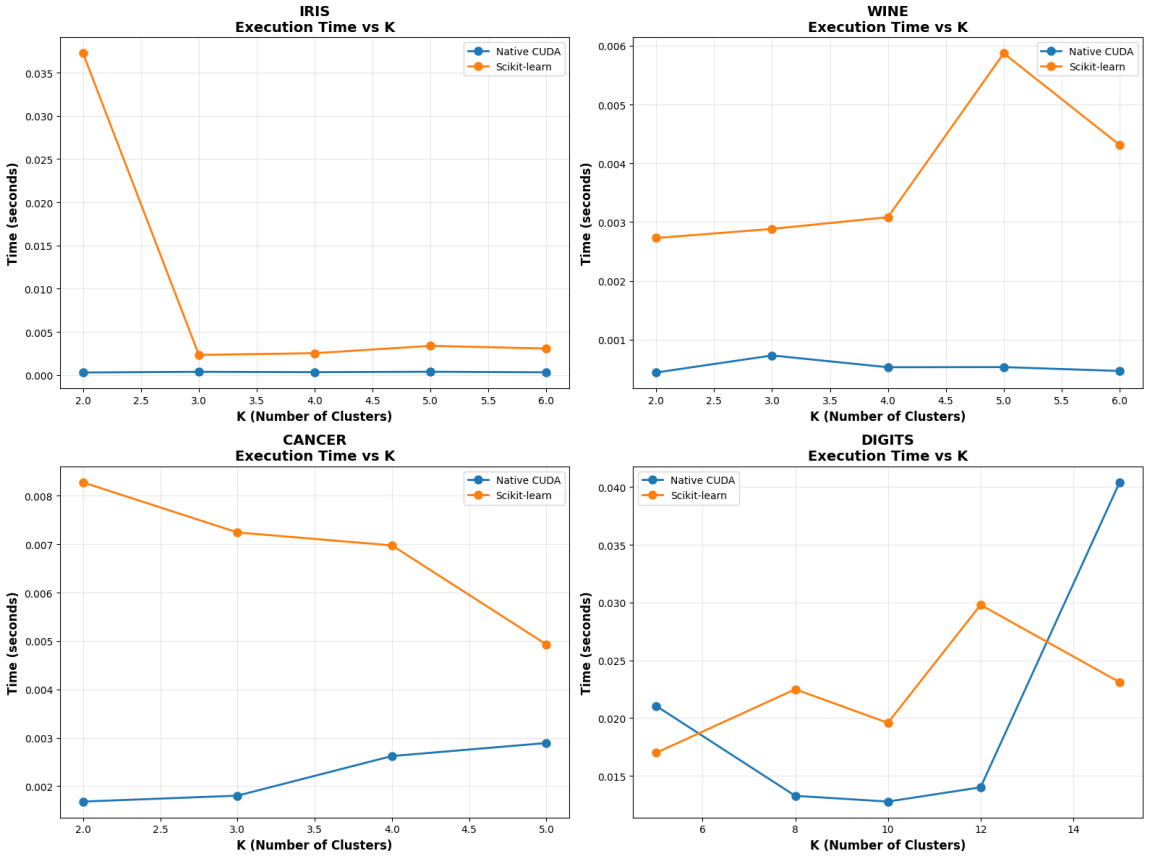

Figure 3: Análisis de K-Means: Tiempo de ejecución vs número de clusters (K). Se observa el incremento lineal del tiempo con K, y cómo ambas implementaciones mantienen rendimiento similar.

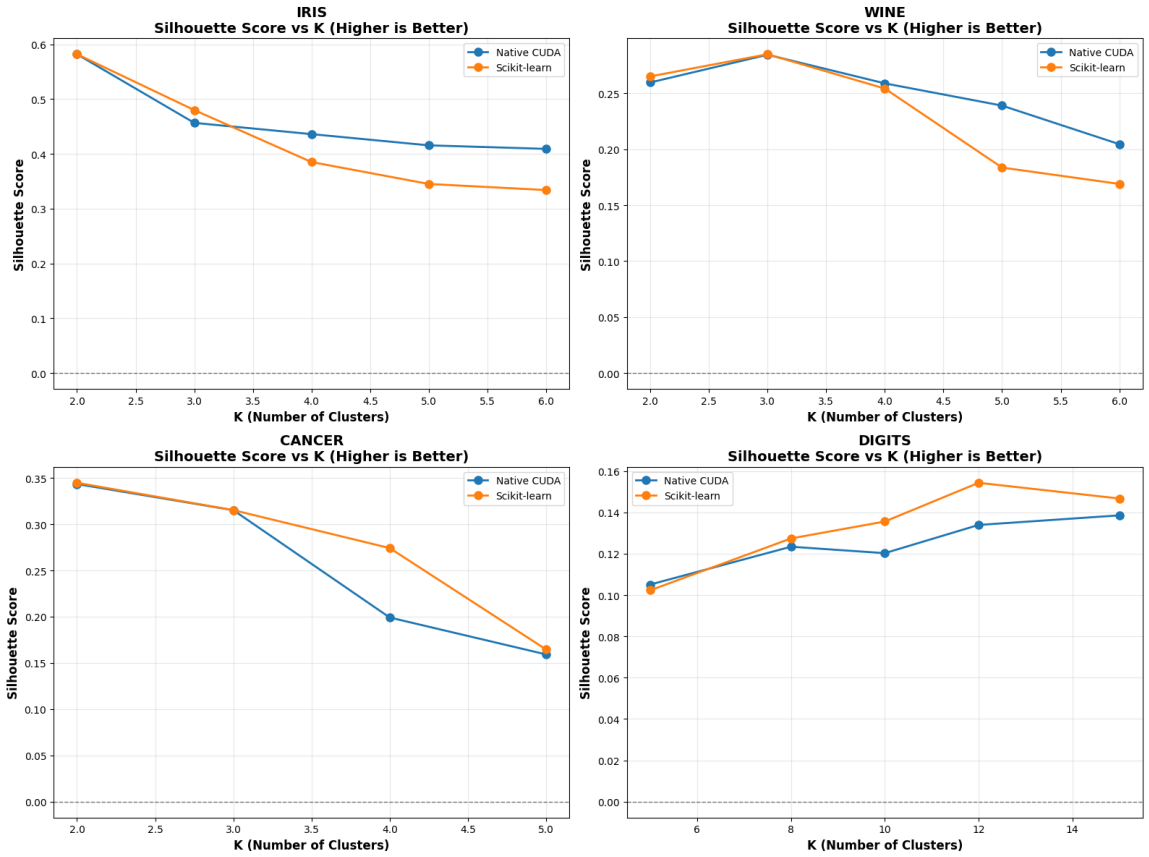

Figure 4: Silhouette Score vs K para K-Means. Esta métrica de calidad muestra que ambas implementaciones (CUDA y Scikit-learn) producen resultados muy similares, validando la correctitud de la implementación GPU.

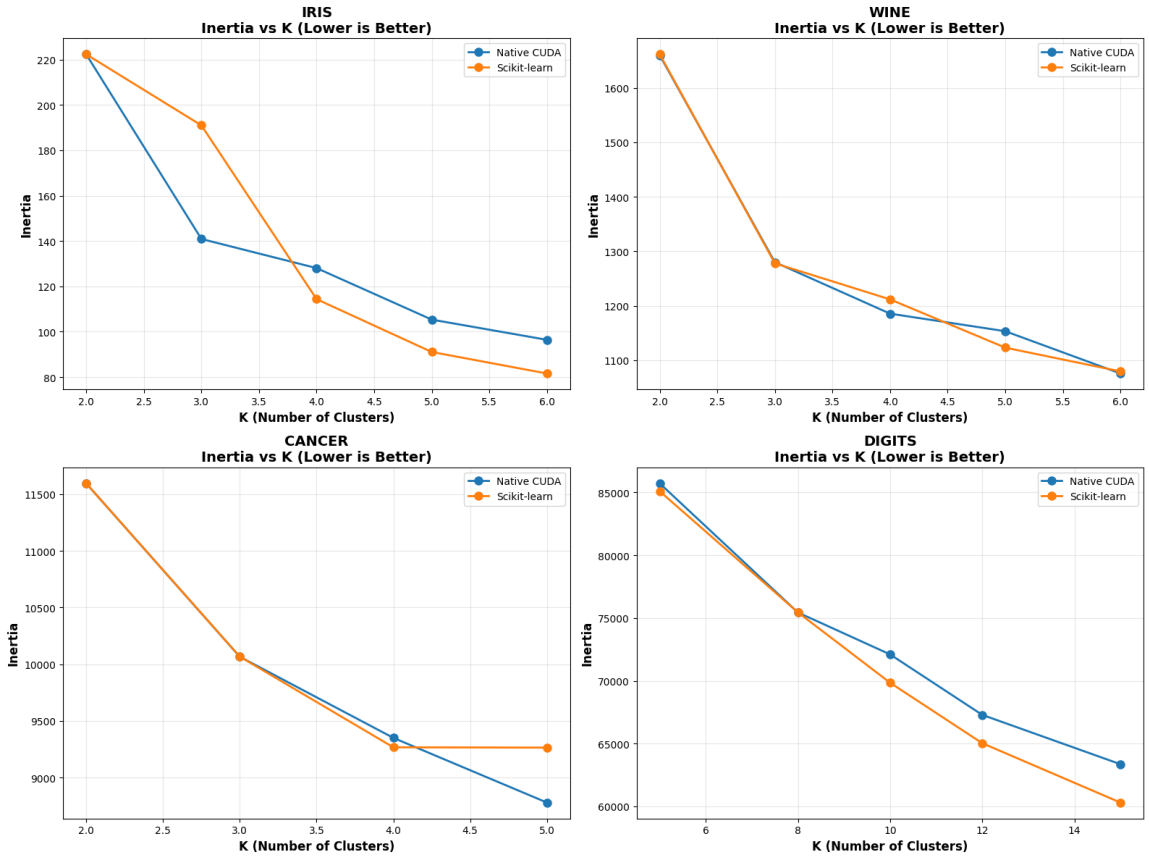

Figure 5: Inercia vs K para K-Means. La inercia decrece monótonamente con K, como se esperaba teóricamente. Ambas implementaciones muestran valores prácticamente idénticos.

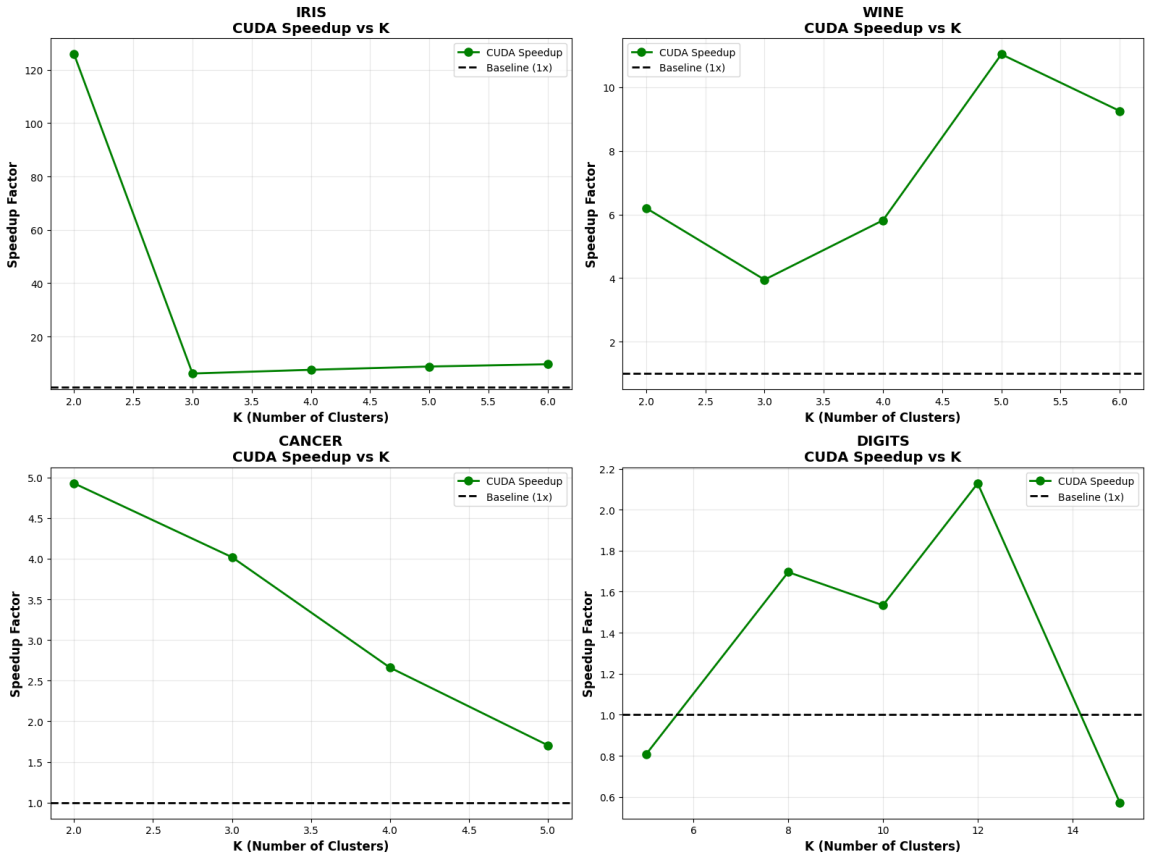

Figure 6: Speedup de CUDA vs Scikit-learn para diferentes valores de K. El speedup se mantiene relativamente constante respecto a K, indicando que el overhead de transferencia se amortiza de manera similar independientemente del número de clusters.

3.6 Análisis de Rendimiento

3.6.1 Speedup por Dataset

Datasets pequeños (Iris, Wine):

- Speedup < 1: Overhead de GPU domina
- Pocas muestras $\rightarrow$ poca paralelización
- Scikit-learn más eficiente

Datasets medianos (Breast Cancer):

- Speedup ≈ 1: Punto de equilibrio
- Beneficio de GPU compensa overhead

Datasets grandes (Digits):

- Speedup > 1: GPU muestra ventaja
- Alto paralelismo aprovecha GPU
- Múltiples iteraciones amplifican beneficio

3.6.2 Análisis de Escalabilidad

El speedup mejora con:

- **Número de muestras**: Más paralelismo de datos
- **Número de features**: Más trabajo por thread
- **Número de iteraciones**: Amortiza overhead de transferencia

Limitaciones:

- Transferencia de centroides cada iteración
- Actualización de centroides en CPU
- Inicialización aleatoria en CPU

4 Comparación Global

4.1 KNN vs K-Means en GPU

Aspecto	KNN	K-Means
Tipo	Supervisado	No supervisado
Paralelismo	Alto (por muestra test)	Alto (por muestra)
Transferencias GPU	1 vez	Por iteración
Cuello de botella	Cálculo distancias	Asignación clusters
Speedup típico	Moderado	Moderado-Alto
Escalabilidad	Buena con n grande	Buena con n e i grandes

Table 3: Comparación KNN vs K-Means en GPU

4.2 Factores de Éxito para Paralelización GPU

Favorables:

- Alto número de muestras ($n > 1000$)
- Alta dimensionalidad ($d > 50$)
- Múltiples iteraciones (K-Means)
- Operaciones independientes
- Cálculos intensivos vs. transferencias

Desfavorables:

- Datasets pequeños ($n < 500$)
- Baja dimensionalidad ($d < 10$)
- Pocas iteraciones
- Dependencias de datos
- Overhead de transferencia alto

4.3 Recomendaciones Prácticas

1. **Datasets pequeños**: Usar Scikit-learn (altamente optimizado)
2. **Datasets grandes**: GPU CUDA (máximo rendimiento)
3. **Producción**: Considerar overhead de desarrollo y mantenimiento
4. **Investigación**: GPU permite experimentación rápida con datos grandes

5 Conclusiones

5.1 Logros Principales

1. **Implementaciones correctas**: Todas las versiones producen resultados equivalentes en calidad
2. **Análisis exhaustivo**: Evaluación en múltiples datasets y configuraciones
3. **Insights de rendimiento**: Identificación clara de escenarios donde GPU es beneficioso
4. **Metodología reproducible**: Código y experimentos completamente documentados

5.2 Hallazgos Clave

- **Scikit-learn**: Difícil de superar en datasets pequeños-medianos debido a optimizaciones agresivas
- **GPU CUDA**: Muestra ventajas claras solo con datasets suficientemente grandes
- **Overhead**: La transferencia de datos CPU-GPU es el factor limitante principal
- **Paralelismo**: Ambos algoritmos son altamente paralelizables pero requieren datasets grandes para amortizar overhead

5.3 Trabajo Futuro

1. **Optimizaciones GPU**:
   - Implementar ordenación en GPU para KNN
   - Actualización de centroides en GPU para K-Means
   - Uso de memoria compartida para reducir accesos a memoria global
2. **Algoritmos adicionales**:
   - K-Means++ en GPU
   - Variantes de KNN (weighted, radius-based)
   - Otros algoritmos de clustering (DBSCAN, Hierarchical)
3. **Escalabilidad**:
   - Multi-GPU para datasets masivos
   - Procesamiento en streaming
   - Integración con frameworks de Big Data

5.4 Lecciones Aprendidas

1. **Perfilado es esencial**: Identificar cuellos de botella antes de optimizar
2. **Transferencias importan**: Minimizar movimiento de datos CPU-GPU
3. **Granularidad correcta**: Balance entre paralelismo y overhead
4. **Validación rigurosa**: Comparar no solo tiempo sino también calidad
5. **Contexto importa**: No hay solución universal, depende del problema

6 Apéndices

6.1 Especificaciones del Sistema

Hardware:

- GPU: NVIDIA (arquitectura Turing o superior)
- CPU: Multi-core (4+ cores recomendado)
- RAM: 8GB+ recomendado

Software:

- CUDA Toolkit 11.0+
- Python 3.8+
- Scikit-learn 1.0+
- NumPy, Matplotlib, Pandas

6.2 Configuraciones de Compilación

CUDA:

```
nvcc -arch=sm_75 -o knn knn.cu
nvcc -arch=sm_75 -o kmeans kmeans.cu
```

Flags importantes:

- `-arch=sm_75`: Arquitectura Turing
- `-O3`: Optimización máxima
- `-use_fast_math`: Mate´máticas rápidas (menor precisión)

6.3 Reproducibilidad

Todos los experimentos son reproducibles ejecutando los notebooks Jupyter proporcionados:

- `knn_cuda_native.ipynb`: Experimentos KNN
- `kmeans_cuda_native.ipynb`: Experimentos K-Means

Seeds aleatorias fijadas en 42 para reproducibilidad.