Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Aceleración mediante GPU de un Modelo de Identificación de Anomalías Meteorológicas con Alta Incidencia en la Predicción de Incendios Forestales

Autor: Bruno Burgos Kosmalski

Tutor: José Luis Pedraza Domínguez

Madrid, ENERO - 2026

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Aceleración mediante GPU de un Modelo de Identificación de Anomalías Meteorológicas con Alta Incidencia en la Predicción de Incendios Forestales
ENERO - 2026

Autor: Bruno Burgos Kosmalski
Tutor: José Luis Pedraza Domínguez
DEPARTAMENTO DE ARQUITECTURA Y TECNOLOGÍA DE SISTEMAS INFORMÁTICOS
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

El presente trabajo consiste en el desarrollo de una implementación óptima para la ordenación de múltiples vectores (aproximadamente 200 000), cada uno de ellos de tamaño reducido (en torno a 27 000 elementos por vector), haciendo uso de aceleradores gráficos (GPU). El objetivo principal es optimizar el proceso de ordenación de dichos vectores que se realizaba en el trabajo de fin de carrera de Esteban Aspe Ruiz, titulado “Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática”, orientado al cálculo de la anomalía climática.

Para ello, se han estudiado distintas implementaciones existentes que abordan problemas de características similares, así como las funciones y librerías disponibles que pueden emplearse en este contexto. Sobre esta base, se ha llevado a cabo una serie de pruebas experimentales con el fin de evaluar el rendimiento de cada alternativa. A partir de los resultados obtenidos, se ha realizado una selección de las soluciones más adecuadas para su implantación en el entorno final de cálculo de la anomalía climática, verificando posteriormente el rendimiento alcanzado.

Abstract

This work focuses on the development of an optimal implementation for sorting multiple vectors (approximately 200 000), each of relatively small size (around 27 000 elements per vector), using graphics processing units (GPUs). The main objective is to optimize the sorting of these vectors within the framework of the bachelor's thesis by Esteban Aspe Ruiz, entitled "Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática", aimed at computing climate anomalies.

To this end, existing implementations that address similar problems have been studied, together with the functions and libraries that can be applied to this specific case. Based on this analysis, a series of experimental tests have been carried out to evaluate the performance of each approach. From the results obtained, a selection of the most suitable solutions has been made for their integration into the final environment for the climate anomaly computation, and the final performance achieved has subsequently been evaluated.

Índice

1 Introducción

Este trabajo tiene como objetivo el desarrollo de un algoritmo de ordenación de altas prestaciones que, mediante el uso de técnicas y librerías avanzadas, permita optimizar la ordenación de aproximadamente doscientas mil colecciones indexadas o arrays, cada uno de un tamaño relativamente pequeño (en torno a veintisiete mil elementos). Para lograrlo, se propone aprovechar el potencial de los aceleradores gráficos (GPUs) [1], explorando dos enfoques complementarios: por un lado, el uso de CUDA [2] [3], orientado a las GPUs de la marca NVIDIA [4]; y por otro, la utilización de OpenACC [5] [6], con un carácter más general.

Todo este trabajo se desarrolla entonces con la motivación de contribuir a la predicción de incendios forestales, mediante la reducción de los tiempos de ejecución en el cálculo de la anomalía climática, parámetro fundamental en la estimación del Índice de Propagación Potencial (IPP) [7]. Con este propósito, se busca optimizar y paralelizar el cálculo de la anomalía climática aprovechando los recursos de la GPU, tomando como punto de partida la aplicación desarrollada en el Trabajo de Fin de Grado de Esteban Aspe Ruiz, "Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática" [8]. De esta manera, se pretende mejorar significativamente los tiempos de ejecución del proceso.

1.1 Contexto

Esta propuesta tiene su origen en el Trabajo de Fin de Grado: "Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática" [8], realizado por Esteban Aspe Ruiz. En dicho trabajo se desarrolló una aplicación orientada al cálculo de la anomalía climática en una serie de puntos geográficos, buscando la máxima optimización mediante la utilización de un entorno paralelo con múltiples hilos de ejecución (OpenMP [9] [10]).

Es en este anterior proyecto donde se profundiza en la necesidad de optimizar dicho cálculo, ya que la anomalía climática constituye uno de los parámetros fundamentales para la estimación del Índice de Propagación Potencial (IPP) [7], índice propuesto por Fernando Chico Zamora durante el 8.º Congreso Forestal Español (2022) [11], destinado a la predicción de incendios forestales. Este índice se basa principalmente en las anomalías registradas en la temperatura media y en el nivel de sequía, cuya combinación define el parámetro conocido como anomalía climática.

Para calcular esta anomalía es necesario disponer de un punto de referencia temporal previo, a partir del cual se determinan una serie de percentiles correspondientes a la temperatura media y al grado de sequía. Según lo indicado por Fernando Chico, para que dichos percentiles sean representativos, es imprescindible contar con al menos treinta años de datos diarios. Esta necesidad de un histórico tan extenso implica manejar un volumen de datos considerable, entorno a unos veintisiete mil elementos por punto geográfico, considerando cincuenta años de registros.

El cálculo de los percentiles requiere que los datos estén previamente ordenados, lo que implica incorporar el tiempo de ordenación de las listas dentro del proceso global. En el trabajo anterior se observó que esta fase de ordenación

representaba una parte significativa del tiempo total de ejecución, además de constituir un problema difícil de paralelizar de manera eficiente.

Tras optimizar el cálculo mediante computación paralela con OpenMP, se desarrollaron varias versiones experimentales en las que se intentó delegar la ordenación de los datos a la GPU, utilizando para ello la librería Thrust [12] de CUDA. No obstante, los resultados obtenidos no cumplieron las expectativas, ya que los tiempos de ejecución fueron similares o incluso superiores a los alcanzados en CPU. A ello se sumó la limitación temporal del proyecto, motivo por el cual se decidió descartar el uso de la GPU para esta tarea y no continuar con nuevas pruebas en esa línea.

Como se puede observar en las figuras 1.1, 1.2, 1.3 y 1.4, los tiempos de ejecución en GPU solo resultan competitivos cuando el tamaño de las listas supera el umbral de los 100 000 elementos, una magnitud considerablemente mayor a la de los conjuntos de datos empleados en este estudio. Además, se constató que la ejecución en GPU, que requiere sincronización en determinados puntos, mantiene tiempos prácticamente constantes al aumentar el número de hilos, a diferencia de la ejecución en CPU, donde sí se observa una reducción significativa del tiempo de cálculo con el incremento del paralelismo.

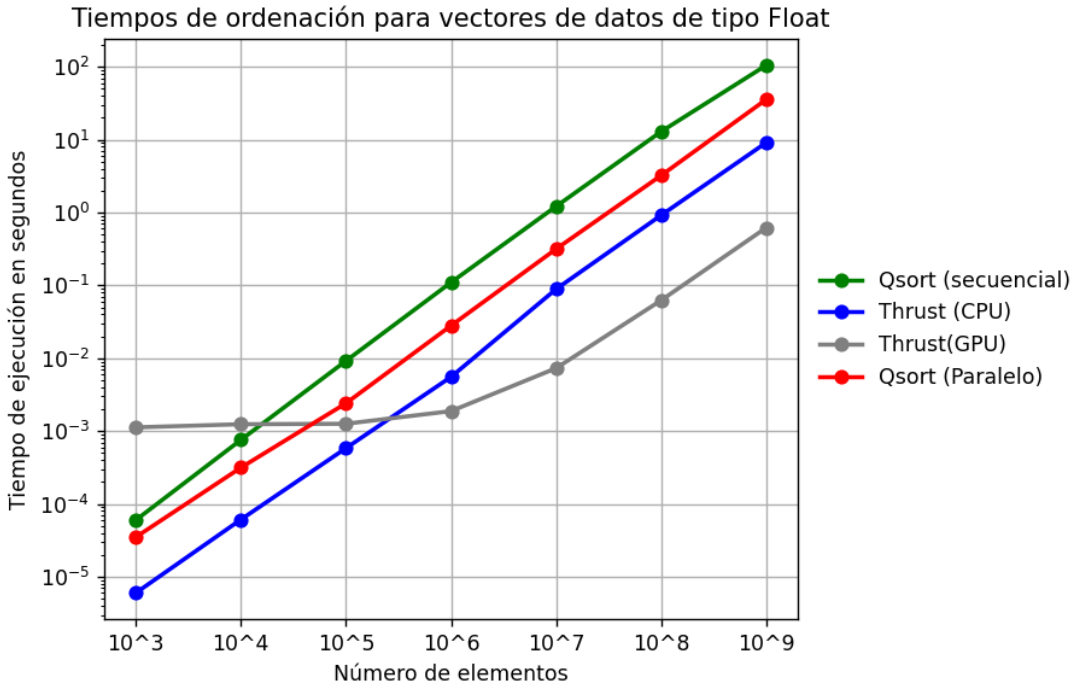

FIGURA 1.1 *GRÁFICA CON LOS TIEMPOS DE EJECUCIÓN PARA DISTINTOS TAMAÑOS DE ARRAYS CON ELEMENTOS DE TIPO FLOAT*

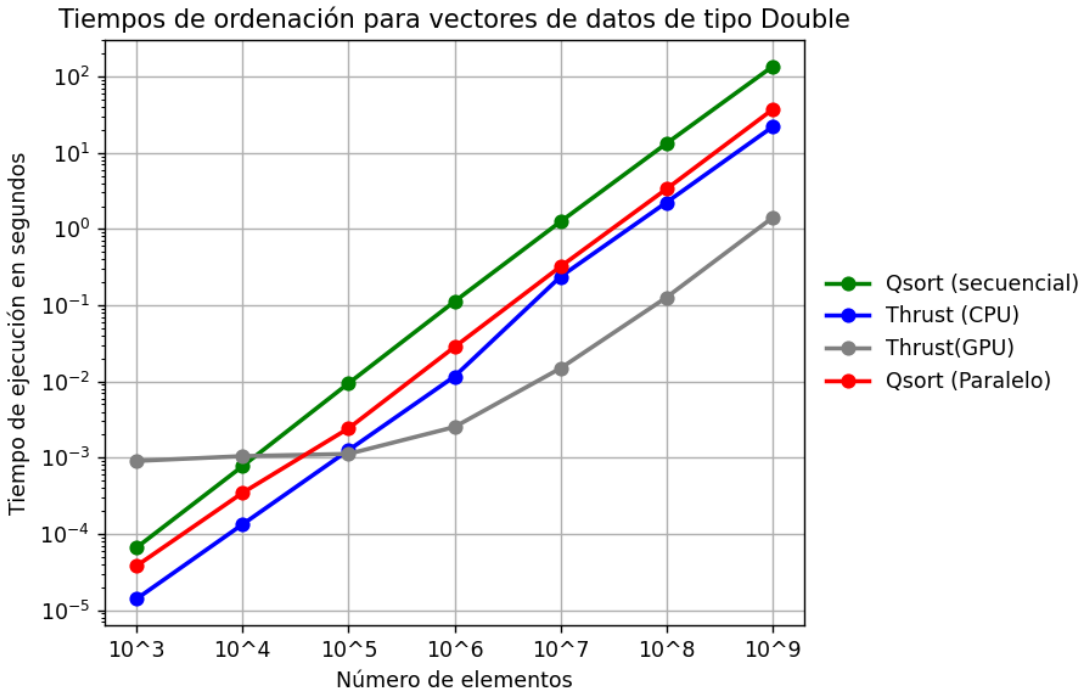

Figura 1.2 *Gráfica con los tiempos de ejecución para distintos tamaños de arrays con elementos de tipo Double*

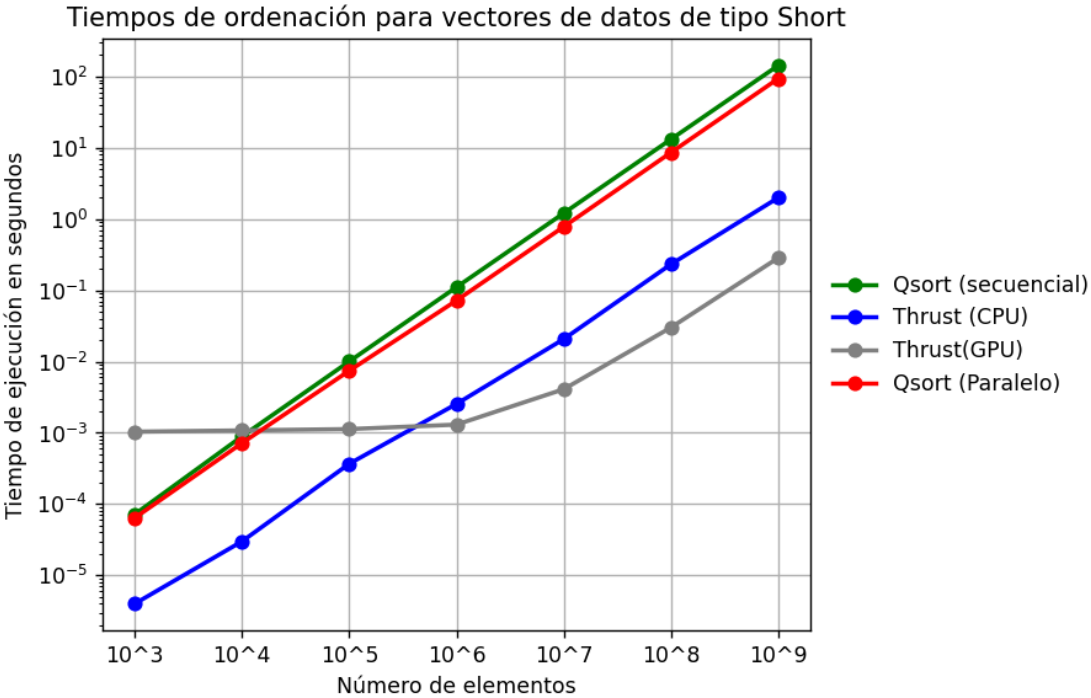

Figura 1.3 *Gráfica con los tiempos de ejecución para distintos tamaños de arrays con elementos de tipo Short*

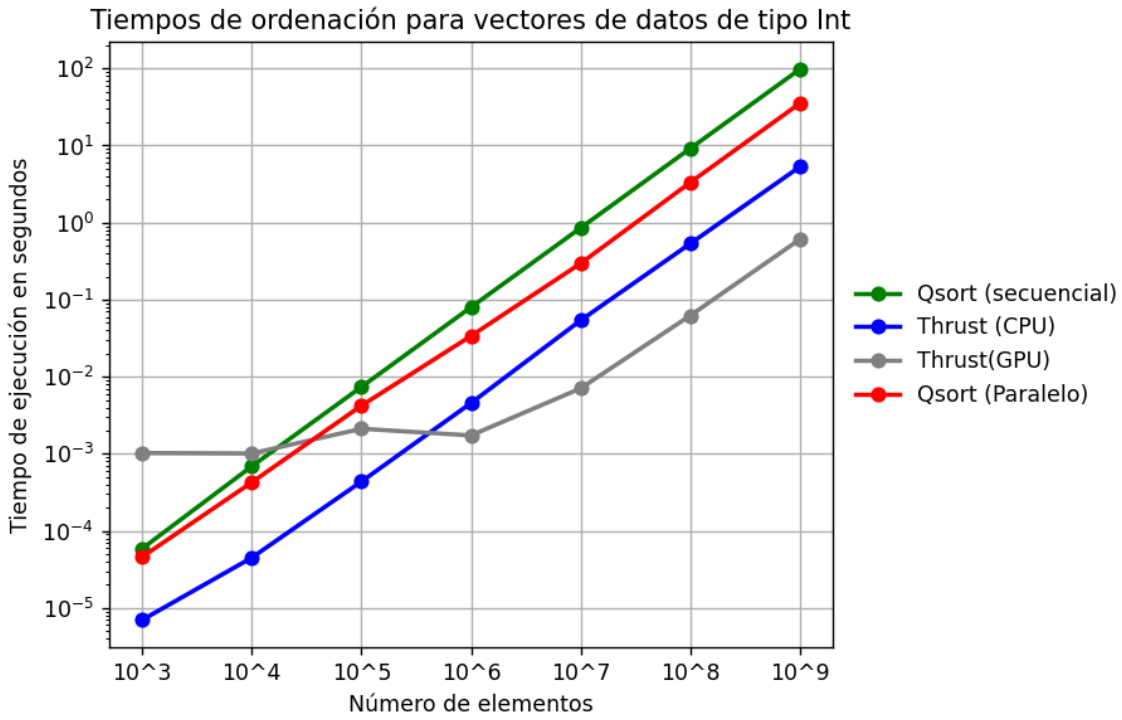

FIGURA 1.4 GRÁFICA CON LOS TIEMPOS DE EJECUCIÓN PARA DISTINTOS TAMAÑOS DE ARRAYS CON ELEMENTOS DE TIPO INT

Es en este punto donde la propuesta actual toma forma, al plantearse retomar el anterior trabajo con el objetivo de maximizar la eficiencia en la ordenación de las listas mediante el uso de GPUs, empleando para ello técnicas y librerías avanzadas. Asimismo, se busca analizar los resultados obtenidos y evaluar la viabilidad de su implementación en el entorno real, orientado al cálculo eficiente de la anomalía climática.

1.2 Objetivos

El objetivo principal de la propuesta es investigar métodos de ordenación en GPU que se puedan adaptar específicamente al problema de ordenación asociado al cálculo de la anomalía climática. A continuación, se evaluarán las prestaciones de dichos métodos de ordenación mediante pruebas sintéticas (vectores arbitrarios, con componentes aleatorios). Seguidamente se aplicaría a un conjunto amplio de vectores reales y finalmente, evaluar las posibles mejoras en el tiempo de ejecución de la aplicación de cálculo de la anomalía climática derivadas del uso del mecanismo de ordenación seleccionado.

- **Investigar métodos de ordenación en GPU** adaptados al problema específico de ordenación vinculado al cálculo de la anomalía climática, considerando la ordenación simultánea de múltiples colecciones indexadas de relativamente pocos elementos (aproximadamente 27000).
- **Diseñar y ejecutar pruebas sintéticas** con vectores arbitrarios para evaluar el rendimiento y la escalabilidad de los distintos algoritmos de ordenación propuestos.
- **Integrar el mecanismo de ordenación seleccionado** en la aplicación existente para el cálculo de la anomalía climática.
- **Evaluar las mejoras obtenidas en el tiempo de ejecución** derivadas del uso del nuevo mecanismo de ordenación.

1.3 Estructura de la Memoria

Esta memoria se estructura en cinco capítulos, cuya organización y contenido se resumen brevemente a continuación:

1. **Introducción:** se presenta el contexto del proyecto, los objetivos principales que guían su desarrollo y la estructura que seguirá la memoria.

2. **Estado del Arte:** se exponen los conceptos teóricos fundamentales para la correcta comprensión del trabajo, así como una revisión de proyectos y estudios relacionados que abordan problemáticas similares.

3. **Desarrollo:** se describe el proceso de implementación del proyecto, incluyendo las dificultades encontradas, los errores detectados y las distintas versiones desarrolladas a lo largo del trabajo.

4. **Conclusiones y Trabajo Futuro:** se exponen las conclusiones finales derivadas del proyecto, así como las posibles líneas de mejora y desarrollo futuro.

5. **Análisis del Impacto:** se expone el impacto personal, social, cultural y medioambiental que ha tenido el desarrollo del proyecto.

2 Estado del Arte

En este apartado se explicarán conceptos teóricos fundamentales para la correcta comprensión del proyecto, así como las diversas tecnologías empleadas durante el desarrollo del mismo. Adicionalmente se hará una breve revisión de proyectos y estudios que abordan problemáticas similares.

2.1 Tecnologías Empleadas

2.1.1 Lenguaje de Programación Principal

Dado que el objetivo principal es portar esta investigación e integrarla en la aplicación principal destinada al cálculo de la anomalía climática, resulta esencial que las pruebas desarrolladas sean plenamente compatibles con el lenguaje de programación C/C++, que constituye la base del proyecto original. Además, este lenguaje ofrece diversas ventajas técnicas, entre ellas su total compatibilidad con CUDA [3] y OpenACC [6], lo que permite aprovechar las capacidades de programación paralela en GPU. Asimismo, C/C++ dispone de amplias bibliotecas especializadas que pueden resultar de gran utilidad para la implementación y evaluación de las pruebas propuestas.

Adicionalmente, este proyecto utiliza las interfaces de OpenMP [10] para la paralelización mediante hilos de ejecución, por lo que las versiones también deberán ser compartibles con esta herramienta

2.1.2 CUDA (Compute Unified Device Architecture)

Compute Unified Device Architecture (CUDA) [2] [3] es una plataforma y modelo de programación desarrollado por NVIDIA que permite aprovechar las unidades de procesamiento gráfico (GPU) [1] para realizar cálculos de propósito general. Gracias a ello, es posible ejecutar tareas altamente paralelas, como simulaciones científicas y algoritmos de aprendizaje profundo (deep learning) [13], entre otras; aprovechando la gran capacidad de paralelización que ofrecen las GPUs por su elevado número de núcleos.

Para resumir, CUDA permite utilizar la GPU como un coprocesador masivamente paralelo, incrementando el rendimiento en aplicaciones que demandan una alta capacidad de cómputo. Sin embargo, este enfoque solo es válido en equipos que cuenten con una tarjeta gráfica NVIDIA, ya que la tecnología es propietaria de dicha marca.

A su vez, este entorno cuenta con diversas herramientas que permiten abordar de manera eficiente el problema de la ordenación de listas. Entre ellas destacan la librería Thrust [12] (empleada en el proyecto anterior), que ofrece funciones optimizadas de ordenación, y la librería CUB, diseñada para proporcionar primitivas de bajo nivel y alto rendimiento en GPU, con especial enfoque en las operaciones de sorting.

Estas librerías permiten aprovechar al máximo la arquitectura paralela de CUDA, simplificando el desarrollo y mejorando significativamente el rendimiento.

2.1.3 OpenACC (Open Accelerators)

OpenACC (Open Accelerators) [5] [6] es un modelo de programación y conjunto de directivas diseñado para facilitar la paralelización de aplicaciones en aceleradores, como GPUs y otras unidades de procesamiento masivamente paralelo. Gracias a OpenACC, se pueden ejecutar tareas altamente paralelas, aprovechando la capacidad de paralelización de estos dispositivos, sin necesidad de gestionar directamente los detalles de bajo nivel del hardware.

Adicionalmente, OpenACC es un estándar completamente abierto, a diferencia de CUDA, lo que permite abordar el problema de manera más general, sin depender de una tarjeta gráfica específica de la marca NVIDIA.

2.2 Proyectos Similares

Aunque no existe ningún proyecto que aborde este problema específico más allá del trabajo previo en el que se basa esta propuesta, sí se encuentran varios proyectos que buscan mejorar la eficiencia de la ordenación de arrays en GPUs.

2.2.1 Fast Segmented Sort on GPUs

"FastSegmentedSort" [14]es un algoritmo diseñado para realizar un ordenamiento segmentado (Segmented Sort) de manera eficiente en GPU. El método propuesto se adapta dinámicamente a la longitud de los segmentos, diferenciando entre cortos y largos para evitar desequilibrios de carga y divergencia de hilos. Introduce dos técnicas principales: reg-sort, que permite ordenar datos directamente en registros con comunicación entre hilos, y smem-merge, una fusión optimizada en memoria compartida para segmentos de longitud variable. Las evaluaciones en GPUs modernas muestran mejoras de rendimiento significativas frente a bibliotecas existentes como CUB [15] y ModernGPU [16], llegando en algunos casos a ejecutar hasta 86 y 4 veces más rápido respectivamente.

2.2.2 A Hybrid Sorting Algorithm on Heterogeneous Architectures

"A Hybrid Sorting Algorithm on Heterogeneous Architecture" [17] propone una estrategia de ordenamiento que aprovecha de forma conjunta las capacidades de la CPU y la GPU en sistemas heterogéneos. El algoritmo divide el conjunto de datos inicial en dos bloques que son ordenados en paralelo por la CPU y GPU; la CPU a su misma vez coordina el proceso y realiza la fusión final de los resultados. Esta combinación permite una distribución eficiente de la carga de trabajo y una mejora significativa en el rendimiento, especialmente en conjuntos de datos de gran tamaño. La propuesta se adapta dinámicamente a las características del hardware y del volumen de datos, mostrando ventajas claras frente a enfoques tradicionales que utilizan únicamente CPU o GPU.

2.2.3 GPU-ArraySort: A parallel, in-place algorithm for sorting large number of arrays

"GPU-ArraySort: A Parallel, In-Place Algorithm for Sorting Large Number of Arrays" [18] presenta una solución eficiente para ordenar múltiples arrays pequeños de forma simultánea en GPU. A diferencia de los algoritmos

tradicionales que se enfocan en ordenar un array de gran tamaño, GPU-ArraySort está diseñado para manejar grandes volúmenes de arrays independientes. El algoritmo opera en paralelo y reduce el uso de memoria adicional, mejorando la escalabilidad. Aprovecha la memoria compartida en CUDA para maximizar el rendimiento. Entendiendo que la mayoría de los arrays son lo suficientemente pequeños como para entrar en la memoria compartida de un único bloque. Esta ordenación tiene tres fases de ejecución: Una primera fase donde se gestionan los tamaños del subarray que manejará cada hilo (Splitter Selection); una segunda fase donde se distribuyen los datos (Bucketing); y una fase final de ordenación (Sorting).

3 Desarrollo

En este capítulo se presenta el desarrollo del trabajo, incluyendo la descripción de los algoritmos seleccionados, el diseño y ejecución de las pruebas, el análisis de los tiempos de ejecución obtenidos y las correspondientes implementaciones en el entorno original.

3.1 Aclaración Previa

Todos los datos que se presentan en este documento se han obtenido mediante pruebas de rendimiento basadas en mediciones de tiempo. Dichas pruebas se han ejecutado en un sistema con 16 núcleos físicos (8 de tipo "performance" y otros 8 de tipo "efficiency" 13th Gen Intel® Core™ i7-13700K) y una GPU NVIDIA (NVIDIA Corporation AD103 [GeForce RTX 4080 SUPER], rev. a1).

Adicionalmente, la versión de código para CPU utilizada en las distintas comparaciones es la misma que la empleada en el Trabajo de Fin de Grado de Esteban Aspe Ruiz, "Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática" [8]. En concreto, se utiliza la función sort de la librería Thrust [12], invocada mediante una llamada similar a la siguiente:

```
thrust::sort(thrust::host, vector, vector + N);
```

donde vector representa la colección de datos a ordenar y N el número de elementos. Cabe destacar que la paralelización se realiza exclusivamente a nivel de las iteraciones totales del cálculo, y no sobre el proceso de ordenación en sí mismo, a diferencia de lo que ocurre en las distintas implementaciones y algoritmos ejecutados en la GPU.

Asimismo, todos los códigos intermedios y finales implementados durante el proyecto quedan recogidos en el repositorio público de Github "tfg_ordenacion_vectores_gpu" [19], donde se mantienen accesibles para futuras implementaciones y seguimientos del trabajo.

3.2 Algoritmos para la ordenación de un único vector

La primera selección de algoritmos se basa en la ordenación de un único vector, en la que el único factor de variación es su tamaño. Este enfoque resulta especialmente útil, ya que permite evaluar el potencial de cómputo de la GPU, particularmente a medida que aumenta el número de elementos a procesar. Además, se trata de la aproximación más cercana a la implementación original, y, por tanto, la más sencilla de implementar.

No obstante, este tipo de algoritmos presenta también ciertos inconvenientes. Entre ellos, destaca el incremento de los tiempos de latencia derivado de la elevada cantidad de llamadas necesarias (una por cada vector independiente). Asimismo, en un contexto de ejecución paralelo, un gran número de llamadas puede introducir puntos adicionales de sincronización que penalizan significativamente el rendimiento. Aunque este efecto pueda mitigarse en algunos casos mediante el uso de *CUDA streams* [20].

3.2.1 CUB Device Radix Sort

3.2.1.1 Descripción

CUB Device Radix Sort [21] es una implementación del algoritmo de ordenación radix [22] incluida en la librería CUB de NVIDIA. La ordenación radix [22] es un algoritmo no comparativo que organiza los elementos procesando sus claves dígito a dígito (o grupo de bits a grupo de bits), en lugar de compararlos directamente como hacen otros algoritmos clásicos, como Quicksort [23] o Mergesort [24].

Esta función se invoca directamente desde el programa, y de manera transparente se lanzan los distintos kernels de GPU necesarios para completar la ordenación, manteniendo la asincronía característica de la GPU en la ejecución de los kernels. Asimismo, esta función está altamente optimizada y diseñada específicamente para su ejecución en GPU.

3.2.1.2 Implementación

La implementación y el uso de esta función constituyen una de las aproximaciones más sencillas dentro de la selección de algoritmos considerada. No solo se encuentra integrada en las versiones más recientes de las herramientas de CUDA, sino que además el procedimiento para su utilización resulta relativamente simple. Dicho procedimiento sigue, de forma general, el siguiente esquema:

1. Reservar la memoria necesaria en la GPU, normalmente destinada a un vector de entrada y a un vector de salida.
2. Realizar una llamada de inicialización para las variables temporales, posteriormente reservando la memoria temporal necesaria.
3. Copiar los datos desde el vector residente en la CPU al vector de entrada ubicado en la GPU.
4. Invocar la función con los parámetros correspondientes.
5. Una vez finalizada la ejecución, copiar el vector de salida de la GPU (correspondiente al vector de entrada ya ordenado) de vuelta a la CPU.

Como se ha mencionado anteriormente, esta función requiere la asignación de regiones de memoria temporales en la GPU con el objetivo de optimizar llamadas posteriores. Este mecanismo resulta especialmente útil, por ejemplo, al ordenar vectores del mismo tamaño dentro de un bucle, ya que la inicialización de dichas regiones temporales solo es necesario realizarla una única vez.

Retomando el ejemplo anterior, en cada iteración, en lugar de repetir los pasos del 2 al 5, el paso 2 se ejecutará fuera del bucle, repitiéndose únicamente los pasos del 3 al 5 en cada iteración. De este modo, la memoria temporal se mantiene entre ejecuciones, lo que permite reducir de forma significativa el tiempo asociado a llamadas sucesivas.

Código ejemplo:

```
// Inicialización del vector en CPU

…

// Inicialización de los vectores en GPU
```

```
float* device_vector;
float* device_sorted_vector;
cudaMalloc(&device_vector, N*sizeof(float));
cudaMalloc(&device_sorted_vector, N*sizeof(float));
// Llamada de inicialización
void* d_temp_storage = NULL;
size_t temp_storage_bytes = 0;
cub::DeviceRadixSort::SortKeys(d_temp_storage,
      temp_storage_bytes,
device_vector, device_sorted_vector, N);
cudaMalloc(&d_temp_storage, temp_storage_bytes);
// Ordenación en GPU
cudaMemcpy(device_vector, vector, N*sizeof(float),
      cudaMemcpyHostToDevice);
cub::DeviceRadixSort::SortKeys(d_temp_storage,
      temp_storage_bytes,
device_vector, device_sorted_vector, N);
cudaMemcpy(sorted_vector, device_sorted_vector, N*sizeof(float),
      cudaMemcpyDeviceToHost);
…
```

Asimismo, con el fin de optimizar la ejecución de esta función en un entorno paralelo y evitar sincronizaciones indeseadas, es posible especificar un CUDA stream independiente para cada invocación. De este modo, el hilo correspondiente puede ejecutar la función de manera más fluida y eficiente.

Por defecto, la invocación de la función se realiza sobre el CUDA stream 0. Esto implica que, si varios hilos llaman simultáneamente a la función, las ejecuciones se serializan y se sincronizan siguiendo el orden de las llamadas. Dado que cada invocación se descompone en tres fases principales: copia del vector original a la GPU, ejecución de la función de ordenación y copia del vector ordenado de vuelta a la CPU; pueden producirse situaciones en las que llamadas potencialmente concurrentes permanezcan bloqueadas de forma innecesaria.

El código anterior se adaptaría entonces:

```
// Inicialización del vector en CPU
…
// Inicialización de los vectores en GPU
cudaStream_t stream;
cudaStreamCreate(&stream);
…
cub::DeviceRadixSort::SortKeys(…, stream);
```

```
…
// Ordenación en GPU
cudaMemcpyAsync(…, stream);
cub::DeviceRadixSort::SortKeys(…, stream);
cudaMemcpyAsync(…, stream);
cudaStreamSynchronize(stream);
```

3.2.1.3 Rendimiento

Para evaluar el rendimiento de esta función, se ha llevado a cabo una serie de pruebas de tiempo sobre vectores de tamaño incremental, comparando los resultados obtenidos con los tiempos de ejecución de la ordenación secuencial en CPU.

A partir de dichas pruebas, se han elaborado gráficos de barras que muestran el speedup alcanzado para cada tipo de elemento evaluado: float, double, int y short. Correspondiendo a las figuras 3.1, 3.2, 3.3, 3.4 respectivamente.

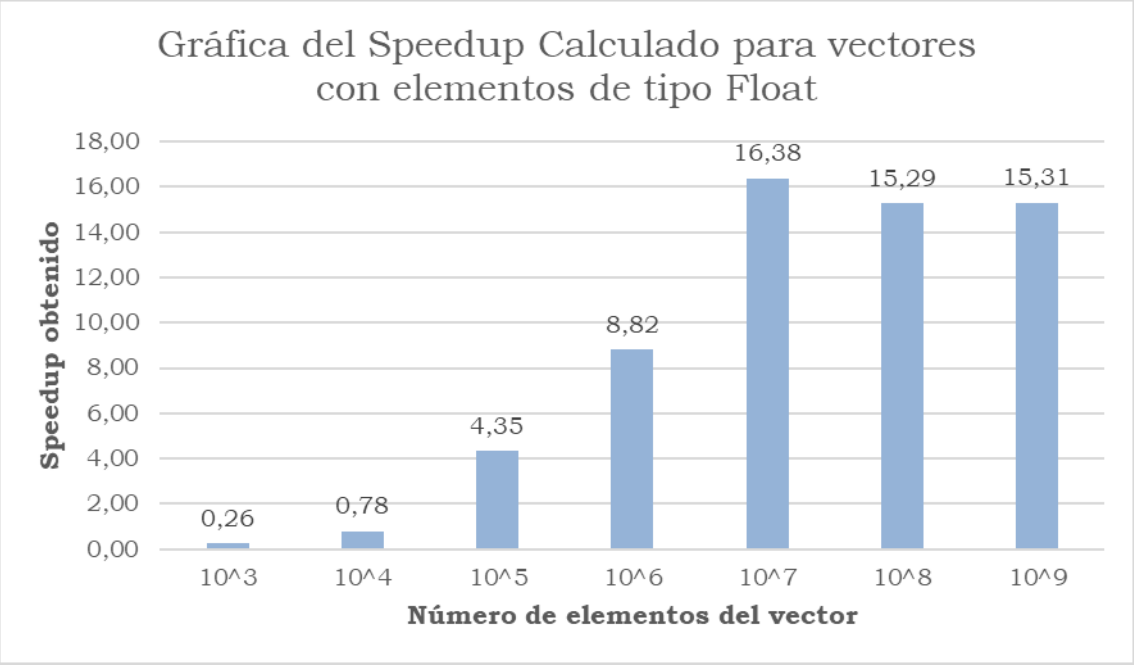

Figura 3.1 *Gráfico de barras con el speedup calculado para los distintos tamaños de vector con elementos de tipo float usando la función device radix sort*

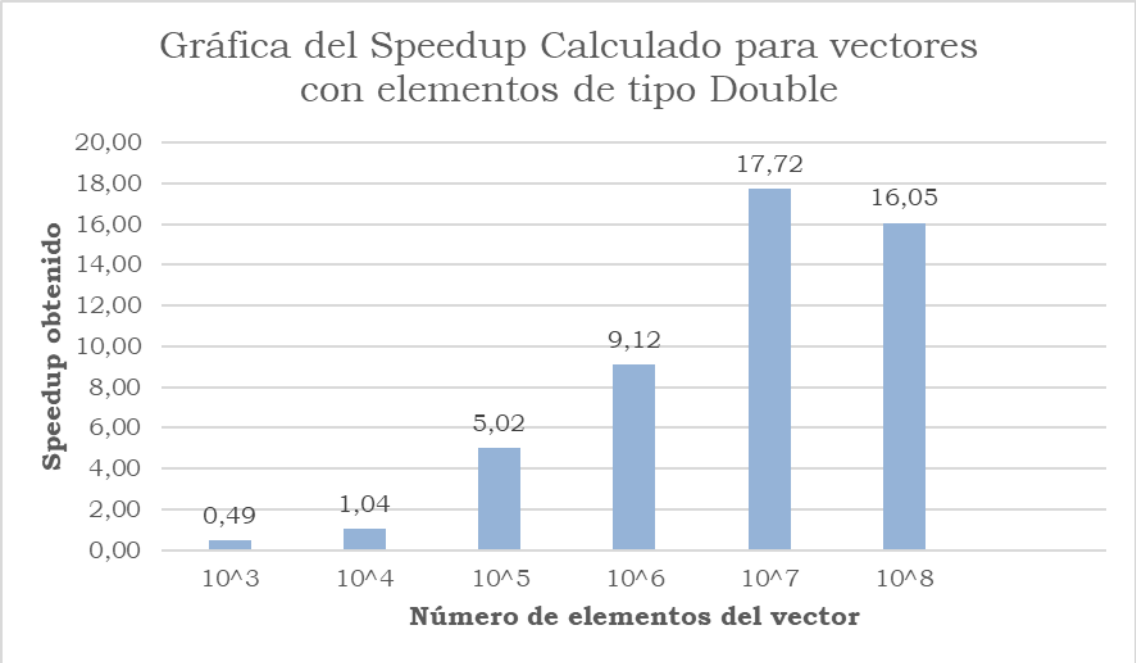

Figura 3.2 *Gráfico de barras con el speedup calculado para los distintos tamaños de vector con elementos de tipo double usando la función device radix sort*

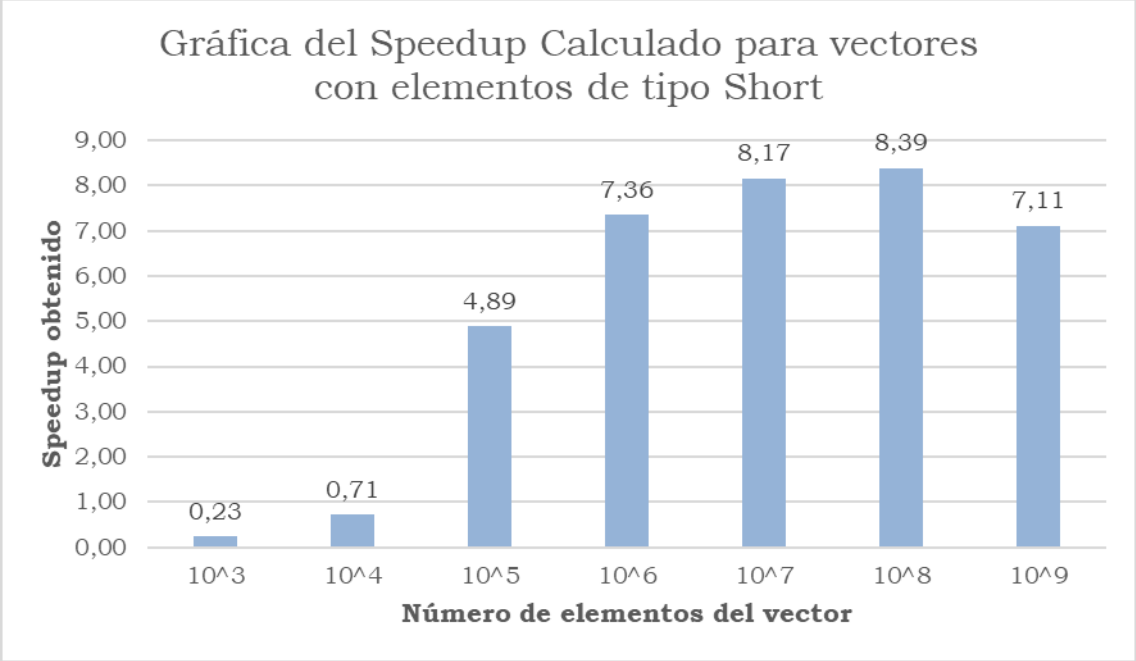

Figura 3.3 *Gráfico de barras con el speedup calculado para los distintos tamaños de vector con elementos de tipo short usando la función device radix sort*

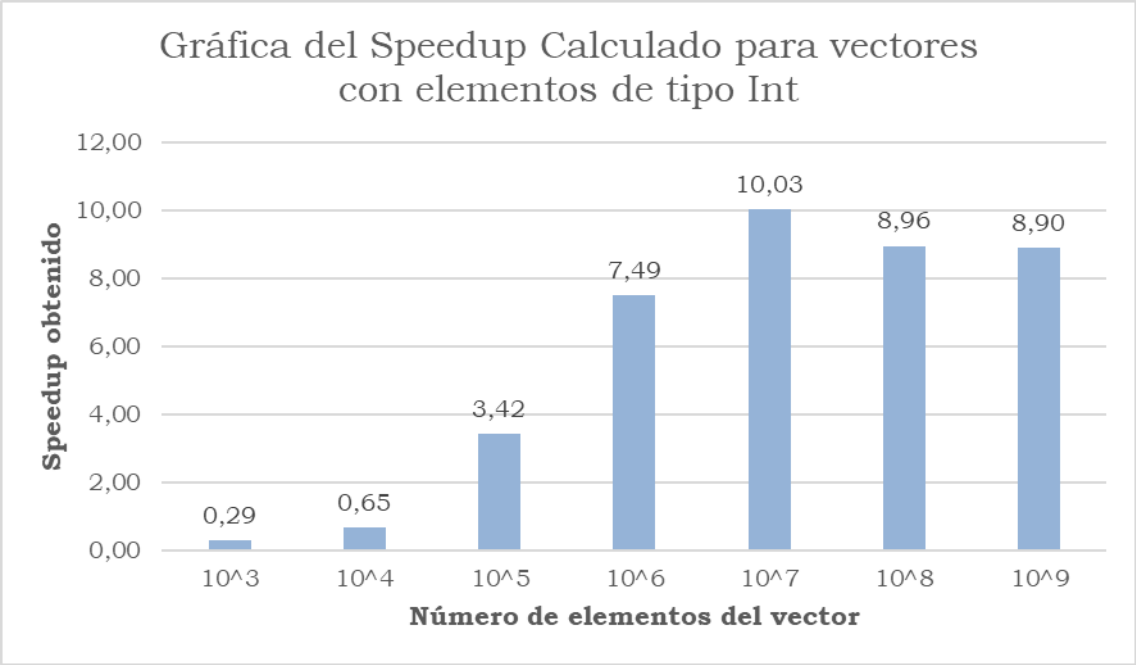

FIGURA 3.4 GRÁFICO DE BARRAS CON EL SPEEDUP CALCULADO PARA LOS DISTINTOS TAMAÑOS DE VECTOR CON ELEMENTOS DE TIPO INT

A partir de los resultados obtenidos se observa que el mejor rendimiento se alcanza al trabajar con vectores de gran tamaño, concretamente a partir de 10^7 elementos por vector. Asimismo, los mejores resultados se obtienen con datos en coma flotante, lo que indica que la GPU ofrece un soporte más eficiente para este tipo de datos en comparación con la CPU.

Este comportamiento se explica porque, al manejar vectores de gran tamaño, es posible aprovechar de forma más efectiva la totalidad de los recursos de la GPU, a diferencia de lo que ocurre con vectores pequeños, donde dichos recursos permanecen infrautilizados. Por otro lado, los resultados muestran que el aumento del tamaño de los vectores no afecta tanto al tiempo de ejecución de la ordenación en sí, como al tiempo dedicado a la copia y transferencia de datos, cuyo impacto dentro del tiempo total incrementa con el tamaño del vector. En este contexto, el coste de transferencia pasa de suponer aproximadamente un 20 % del tiempo total en vectores pequeños a alcanzar hasta un 89 % en los vectores de mayor tamaño, convirtiéndose así en un claro cuello de botella.

Este último aspecto supone una limitación relevante cuando se trabaja con múltiples hilos de ejecución que invocan estas funciones de manera simultánea, ya que los tiempos de espera asociados a las transferencias de datos impiden una reducción significativa del tiempo total de ejecución. Esto se verá en mayor detalle en siguientes apartados.

3.2.1.4 Puntos destacables y desventajas

Esta función presenta una serie de ventajas claras. En primer lugar, al formar parte de la librería CUB de NVIDIA, no es necesario realizar ninguna acción adicional más allá de incluir la cabecera correspondiente en el código. Es decir, no es preciso clonar repositorios externos ni compilar ejecutables adicionales para poder utilizar la función.

Por otra parte, su integración en el entorno de trabajo resulta muy natural, ya que su uso es muy similar al del entorno original. Esto facilita la adaptación del

código existente y ofrece un elevado potencial de reutilización. A ello se añade la posibilidad de trabajar con distintos CUDA streams, lo que permite evitar sincronizaciones innecesarias en un entorno claramente paralelo, en el que las iteraciones, cada una de ellas asociada a un vector independiente, se distribuyen de manera íntegra entre los distintos hilos de ejecución.

No obstante, como se ha observado en los apartados anteriores, esta función no alcanza su máximo potencial hasta trabajar con vectores de al menos 10^7 elementos. Dado que, en el entorno real de aplicación, se ordenan vectores de un tamaño máximo de 27000 elementos, resulta evidente que no es posible explotar plenamente las capacidades de esta función.

Si bien con vectores de dicho tamaño todavía se puede obtener cierta mejora de rendimiento, esta no resulta lo suficientemente significativa como para generar, en pruebas posteriores, una diferencia notable entre las implementaciones basadas en GPU y aquellas ejecutadas en CPU. Esta situación se ve además acentuada por el hecho de que la división del trabajo entre varios hilos no permite un escalado completo del algoritmo.

3.2.2 CUB Block Radix Sort

3.2.2.1 Descripción

CUB Block Radix Sort [25] es una implementación del algoritmo de ordenación radix [22] incluida en la librería CUB de NVIDIA. Muy parecida en su implementación a la función descrita en el apartado anterior (3.2.1).

No obstante, a diferencia de su homónima (Device Radix Sort), esta variante no se invoca desde el host, sino que se utiliza directamente dentro de bloques que se ejecutan en el device, concretamente en el interior de CUDA kernels. Este enfoque proporciona una mayor flexibilidad a la hora de crear, lanzar y gestionar los kernels, lo que permite particularizar aún más la solución y adaptarla a las necesidades específicas del problema.

Sin embargo, esta mayor flexibilidad conlleva también ciertas desventajas, como la necesidad de un mayor esfuerzo en la implementación, así como un incremento en la complejidad de las pruebas y en el número de pasos necesarios para garantizar un uso correcto y eficiente de la función.

3.2.2.2 Implementación

Para implementar esta función y poder ordenar un vector de elementos es necesario diseñar un CUDA kernel encargado de realizar dicha operación. En el diseño de este kernel deben seguirse, de manera general, los siguientes pasos:

1. Inicializar la memoria compartida necesaria para alojar el vector que se desea ordenar.
2. Cada thread del bloque debe recuperar desde memoria global una porción del vector y almacenarla en registros. Así, si el vector contiene N elementos, el bloque se lanza con T hilos, y asumiendo una distribución uniforme de los datos entre los hilos; cada hilo procesará N/T elementos,.

3. Una vez que los datos han sido cargados desde memoria global, se invoca la función de ordenación Block Radix Sort sobre los elementos almacenados en memoria compartida.
4. Tras finalizar el proceso de ordenación, cada hilo almacena su resultado parcial de vuelta en memoria global.

El código correspondiente al diseño del kernel seguirá entonces un esquema similar al que se muestra a continuación:

```
__global__ void kernel(...)
{
using BlockRadixSort = cub::BlockRadixSort<float, T, N/T>;
// Inicialización de la Memoria compartida
__shared__ typename BlockRadixSort::TempStorage temp_storage;
// Recuperación parcial del vector
float thread_keys[N/T];
...
// Ordenación del vector en memoria compartida
BlockRadixSort(temp_storage).Sort(thread_keys);
// Actualización del vector dentro de la memoria global de la GPU
...
}
```

Como puede observarse, esta aproximación requiere considerar una serie de aspectos adicionales de diseño de los que la función Device Radix Sort no depende. Entre ellos se encuentran la necesidad de diseñar explícitamente un kernel, así como calcular el número adecuado de hilos por bloque y gestionar casos en los que no se dispone de una división perfecta entre el número de hilos lanzados por bloque y el número de elementos del vector.

Por otra parte, esta función parece estar orientada principalmente a escenarios de tamaño reducido que pueden beneficiarse de la ejecución dentro de un único bloque de hilos, a diferencia de casos de mayor escala en los que este enfoque no resulta viable.

3.2.2.3 Descarte

Como se ha mencionado anteriormente, esta función no está diseñada para trabajar con vectores de tamaño relativamente grande. Esto se debe a que el proceso de ordenación se realiza íntegramente en memoria compartida dentro de un único bloque. Por tanto, si el tamaño del vector supera la cantidad de memoria compartida disponible en el bloque, la ejecución se interrumpe abruptamente debido a un error en la GPU.

Tomando como referencia el tamaño máximo de los vectores utilizados en el entorno real de trabajo para el cálculo de la anomalía climática, aproximadamente 27 000 elementos por vector, y suponiendo incluso el tipo de

dato de menor tamaño (short), dicho volumen de datos todavía excede la capacidad de memoria compartida disponible en un único bloque de la GPU empleada en las pruebas, aproximadamente 100 KB de máximo.

Este problema podría abordarse incrementando la complejidad de la implementación, por ejemplo, dividiendo el vector en secciones lo suficientemente pequeñas como para ser procesadas por bloques independientes. Sin embargo, tras la realización de diversas pruebas, se observó que el aumento de complejidad no se veía compensado por la mejora de rendimiento obtenida en comparación con la función Device Radix Sort. Por este motivo, se decidió descartar esta función dentro de la selección final de algoritmos considerados.

3.2.3 CUB Device Merge Sort

3.2.3.1 Descripción

CUB Device Merge Sort [26] es una implementación del algoritmo de ordenación merge sort [24] incluida en la librería CUB de NVIDIA. El algoritmo merge sort [24] opera de manera recursiva sobre el vector, dividiéndolo en subvectores de tamaño progresivamente menor hasta alcanzar elementos individuales, para posteriormente fusionarlos de forma ordenada durante la fase de combinación. Este algoritmo destaca especialmente en escenarios con vectores de tamaño reducido.

Esta función presenta similitudes con la descrita en el apartado 3.2.1, Device Radix Sort, principalmente por tratarse de dos funciones de ordenación pertenecientes a la misma librería. Al igual que en dicho caso, la función se invoca desde el host y, de manera interna y transparente, genera los kernels necesarios para llevar a cabo el proceso de ordenación. No obstante, a diferencia de Device Radix Sort, esta implementación no distingue explícitamente entre un vector de entrada y uno de salida, utilizando un único contenedor para el proceso completo.

3.2.3.2 Implementación

Como se indicaba en el apartado anterior, la implementación de esta función es muy similar a la descrita en el apartado 3.2.1, Device Radix Sort, ya que requiere prácticamente los mismos pasos para su utilización:

1. Reservar la memoria necesaria en la GPU, normalmente destinada a un único vector de datos.
2. Realizar una llamada de inicialización para las variables temporales y, posteriormente, reservar la memoria temporal requerida.
3. Copiar los datos desde el vector residente en la CPU al vector ubicado en la GPU.
4. Invocar la función con los parámetros correspondientes.
5. Una vez finalizada la ejecución, copiar el vector almacenado en la GPU (correspondiente al vector ya ordenado) de vuelta a la CPU.

No obstante, existe una diferencia relevante entre ambas implementaciones: en el caso de Device Merge Sort es necesario definir explícitamente una función de comparación, ya que esta debe pasarse como argumento en la llamada a la

función de ordenación. Esta puede seguir una estructura parecida a la siguiente:

```
struct CustomLess
{
  template <typename DataType>
  __device__ bool operator() (const DataType &lhs,
      const DataType &rhs)
  {
    return lhs < rhs;
  }
};
```

Siguiendo entonces la implementación y, notando que al igual que para la función Device Radix Sort se puede hacer uso de distintos CUDA streams; entonces el código resultante debería seguir la siguiente estructura:

```
// Inicialización del vector en CPU
…
// Inicialización de los vectores en GPU
float* device_vector;
cudaStream_t stream;
cudaMalloc(&device_vector, N*sizeof(float));
cudaStreamCreate(&stream);
// Llamada de inicialización
void* d_temp_storage = NULL;
size_t temp_storage_bytes = 0;
cub::DeviceMergeSort::SortKeys(d_temp_storage,
      temp_storage_bytes, device_vector,
      N, CustomLess(), stream);
cudaMalloc(&d_temp_storage, temp_storage_bytes);
// Ordenación en GPU
cudaMemcpyAsync(device_vector, vector, N*sizeof(float),
      cudaMemcpyHostToDevice);
cub::DeviceMergeSort::SortKeys(d_temp_storage,
      temp_storage_bytes, device_vector, N,
      CustomLess(), stream);
cudaMemcpyAsync(device_vector, device_sorted_vector,
      N*sizeof(float), cudaMemcpyDeviceToHost);
…
cudaStreamSynchronize(stream);
```

3.2.3.3 Rendimiento

Para evaluar el rendimiento de esta función, se han llevado a cabo una serie de pruebas de tiempo sobre vectores de tamaño incremental, comparando los resultados obtenidos con los tiempos de ejecución de la ordenación secuencial en CPU.

A partir de dichas pruebas, se han elaborado gráficos de barras que muestran el speedup alcanzado para cada tipo de elemento evaluado: float, double, int y short. Correspondiendo a las figuras 3.5, 3.6, 3.7, 3.8 respectivamente.

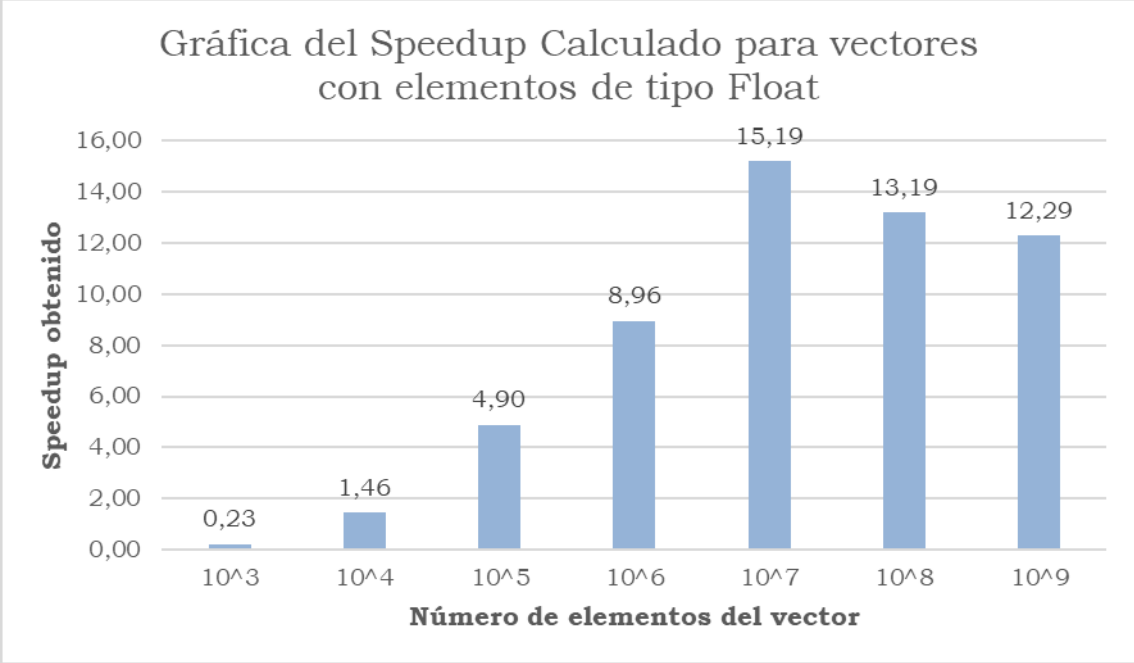

FIGURA 3.5 GRÁFICO DE BARRAS CON EL SPEEDUP CALCULADO PARA LOS DISTINTOS TAMAÑOS DE VECTOR CON ELEMENTOS DE TIPO FLOAT HACIENDO USO DE LA FUNCIÓN DEVICE MERGE SORT

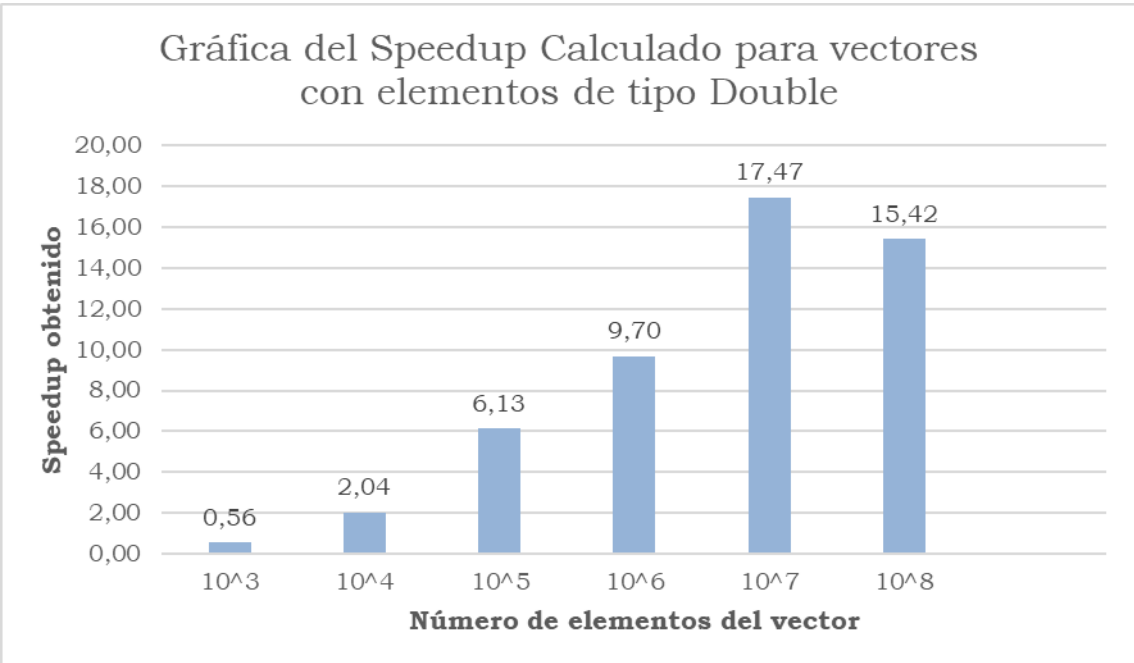

FIGURA 3.6 GRÁFICO DE BARRAS CON EL SPEEDUP CALCULADO PARA LOS DISTINTOS TAMAÑOS DE VECTOR CON ELEMENTOS DE TIPO DOUBLE HACIENDO USO DE LA FUNCIÓN DEVICE MERGE SORT

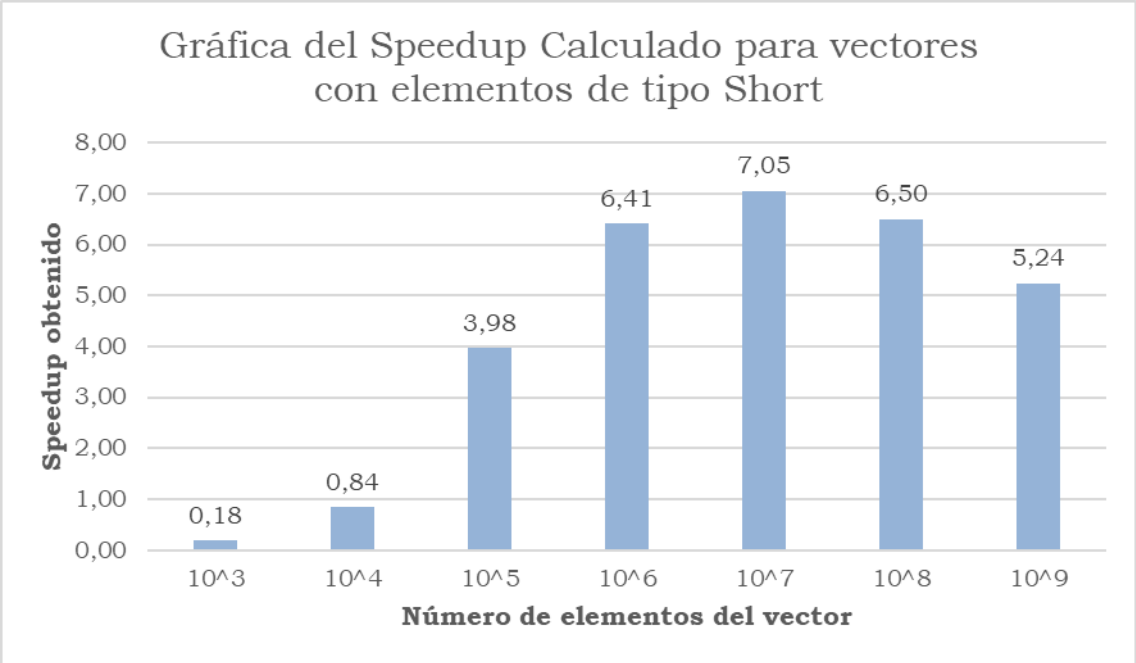

Figura 3.7 *Gráfico de barras con el speedup calculado para los distintos tamaños de vector con elementos de tipo short haciendo uso de la función device merge sort*

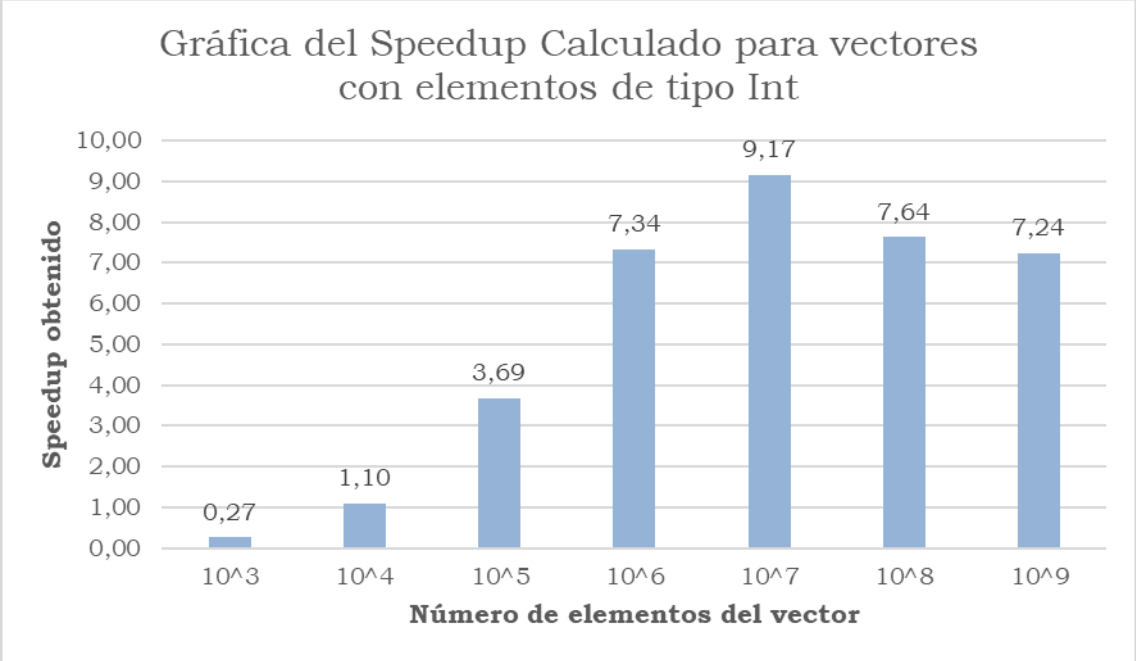

Figura 3.8 *Gráfico de barras con el speedup calculado para los distintos tamaños de vector con elementos de tipo int haciendo uso de la función device merge sort*

Como se puede apreciar en los resultados presentados anteriormente, se repite una situación similar a la observada con la función Device Radix Sort, en la que no se alcanza el mejor rendimiento hasta trabajar con vectores de al menos 10^7 elementos. En este rango, los tiempos de ejecución obtenidos resultan además muy similares a los registrados para Device Radix Sort.

3.2.3.4 Puntos destacables y desventajas

Entre los aspectos más destacados de esta función se encuentran, en gran medida, los mismos ya observados para Device Radix Sort en el apartado 3.2.1. Se trata de una función con un modelo de implementación relativamente sencillo, que no requiere la compilación de librerías externas ni la utilización de repositorios adicionales, lo que simplifica notablemente su integración.

Como característica diferencial, Device Merge Sort ofrece un mayor soporte para tipos de datos complejos, ya que permite especificar explícitamente una función de comparación como argumento de la llamada. Esto amplía su versatilidad frente a la ordenación de estructuras más elaboradas. Asimismo, se observa una ligera mejora en los tiempos de ejecución cuando se trabaja con vectores de tamaño reducido, tendencia que se invierte a medida que aumenta el número de elementos.

Por otro lado, el rendimiento de esta función resulta, en la mayoría de los casos, muy similar al de Device Radix Sort, e incluso inferior en determinadas configuraciones. En consecuencia, no aporta una mejora sustancial ni introduce una ventaja diferencial clara respecto a la función previamente analizada.

3.3 Algoritmos para la ordenación de una colección de vectores

Una vez analizadas las distintas funciones y algoritmos orientados a la ordenación de un único vector en GPU, el siguiente paso consiste en abordar la ordenación de múltiples vectores en una única llamada, es decir, realizar una ordenación batch (conjunta).

Este enfoque presenta una serie de ventajas relevantes, entre las que destaca la reducción de los tiempos de latencia cuando se requiere ordenar un gran número de vectores. Este escenario es especialmente representativo del problema abordado en el cálculo de la anomalía climática, donde se ordenan aproximadamente 200.000 vectores.

No obstante, este enfoque de ejecución batch también presenta una serie de inconvenientes, entre los que destaca el aumento de la complejidad de la implementación. El entorno real para el cálculo de la anomalía climática está diseñado de forma que cada iteración trabaja con un vector independiente, lo que implica que la integración de una estrategia de ordenación conjunta requiera una mayor carga de adaptación.

Adicionalmente, este enfoque resulta menos compatible con la ejecución en un entorno altamente paralelo, ya que, incluso haciendo uso de distintos CUDA streams, la carga de trabajo tiende a concentrarse de forma más intensa sobre la GPU. Esto puede derivar en una menor flexibilidad en la distribución del trabajo y en posibles cuellos de botella que limitan el escalado del rendimiento.

Por este motivo, a diferencia del apartado anterior, el análisis no se centrará en la ordenación de vectores de tamaño creciente, sino en la ordenación de un número incremental de vectores de tamaño fijo. Este planteamiento permite explotar de forma más eficiente los recursos de la GPU, al mismo tiempo que reduce el impacto total de los costes asociados a la transferencia y copia de datos entre CPU y GPU.

3.3.1 CUB Device Segmented Radix Sort

3.3.1.1 Descripción

CUB Device Segmented Sort [27] es una implementación de una ordenación radix [22] segmentada incluida en la librería CUB de NVIDIA. Una ordenación segmentada es la ordenación de múltiples subvectores o segmentos de un vector único o fuente. Es por esto que esta función con una buena configuración y preparación previa permite la ordenación de múltiples vectores de manera simultánea.

3.3.1.2 Implementación

La implementación de esta función, aunque similar a la de Device Radix Sort descrita en el apartado 3.2.1, requiere cumplir una serie de pasos adicionales para su correcto funcionamiento. No obstante, en muchos aspectos mantiene una estructura análoga, como la inicialización de la memoria temporal y el hecho de no necesitar repositorios externos ni componentes ejecutables adicionales.

Para utilizar correctamente esta función, es necesario seguir los siguientes pasos:

1. Reservar memoria suficiente en la GPU. En este caso, se debe alojar un vector compuesto que contenga todos los vectores a ordenar de forma conjunta. Esto implica que, si se desean ordenar M vectores de N elementos cada uno, será necesario reservar dos vectores en la GPU (uno de entrada y otro de salida) de tamaño N×M elementos.
2. Copiar el vector compuesto desde la CPU a la GPU.
3. Inicializar un vector de desplazamientos (offsets) o índices. Este vector permite a la función identificar el inicio y el final de cada vector independiente o segmento dentro del vector compuesto. Será necesario reservar memoria para este vector y copiarlo a la GPU. En el caso de disponer de M vectores, el vector de offsets deberá contener M+1 elementos.
4. Realizar una llamada de inicialización para las variables temporales, reservando así el espacio auxiliar necesario para las ejecuciones posteriores.
5. Invocar la función de ordenación con los parámetros correspondientes.
6. Una vez finalizado el cómputo, copiar de vuelta a la CPU el vector compuesto, que contendrá todos los segmentos o subvectores ordenados de manera independiente.

Como también es posible declarar un stream distinto al por defecto la estructura de código general que deberá seguirse para implementar esta función es la siguiente:

```
// Inicialización del vector en CPU

…

// Inicialización de los vectores en GPU

float* device_vector;
```

```
float* device_sorted_vector;
cudaMalloc(&device_vector, M*N*sizeof(float));
cudaMalloc(&device_sorted_vector, M*N*sizeof(float));
// inicialización del vector de índices
int h_offsets[M+1];
for(int i = 0; i < M+1; i++) h_offsets[i] = (i*N);
int* d_offsets;
cudaMalloc(&d_offsets, (M+1)*sizeof(int));
cudaMemcpy(d_offsets, h_offsets,
      (M+1)*sizeof(int), cudaMemcpyHostToDevice);
// Inicialización del CUDA stream
cudaStream_t stream;
cudaStreamCreate(&stream);
// Llamada de inicialización
void* d_temp_storage = NULL;
size_t temp_storage_bytes = 0;
cub::DeviceSegmentedRadixSort::SortKeys(d_temp_storage,
      temp_storage_bytes,  device_vector,  device_sorted_vector,
      M*N, M, d_offsets, d_offsets+1, 0, sizeof(float)*8, stream);
cudaMalloc(&d_temp_storage, temp_storage_bytes);
// Ordenación en GPU
cudaMemcpyAsync(device_vector, vector, M*N*sizeof(float),
      cudaMemcpyHostToDevice, stream);
cub::DeviceSegmentedRadixSort::SortKeys(d_temp_storage,
      temp_storage_bytes,  device_vector,  device_sorted_vector,
      M*N, M, d_offsets, d_offsets+1, 0, sizeof(float)*8, stream);
cudaMemcpyAsync(sorted_vector, device_sorted_vector,
      M*N*sizeof(float), cudaMemcpyDeviceToHost, stream);
…
```

3.3.1.3 Rendimiento

Para evaluar el rendimiento de esta función, se han llevado a cabo una serie de pruebas utilizando vectores de 27 400 elementos, una cantidad similar a la que se presenta en el entorno real de implementación para el cálculo de la anomalía climática.

Durante las pruebas, se ha mantenido constante el tamaño de cada vector, variando únicamente el número de vectores a ordenar en una única llamada a la función. Estos resultados se han comparado con una ordenación realizada mediante un bucle, procesando la misma cantidad de vectores de forma

secuencial, con el fin de obtener un tiempo de referencia y calcular así el speedup potencial de la función propuesta.

En las Figuras 3.9 y 3.10 se muestra el cálculo del speedup para dos tipos de datos distintos: elementos de tipo float y elementos de tipo short, respectivamente.

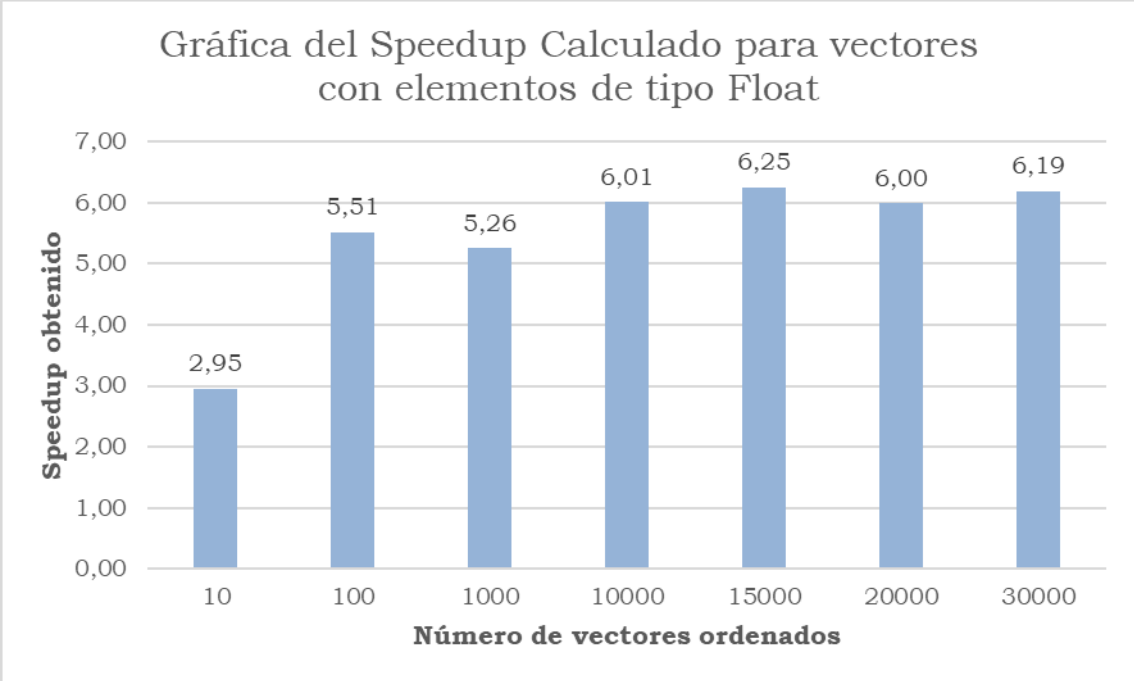

Figura 3.9 *Gráfico de barras con el speedup calculado para un orden incremental de vectores con 27400 elementos de tipo float haciendo uso de la función device segmented radix sort*

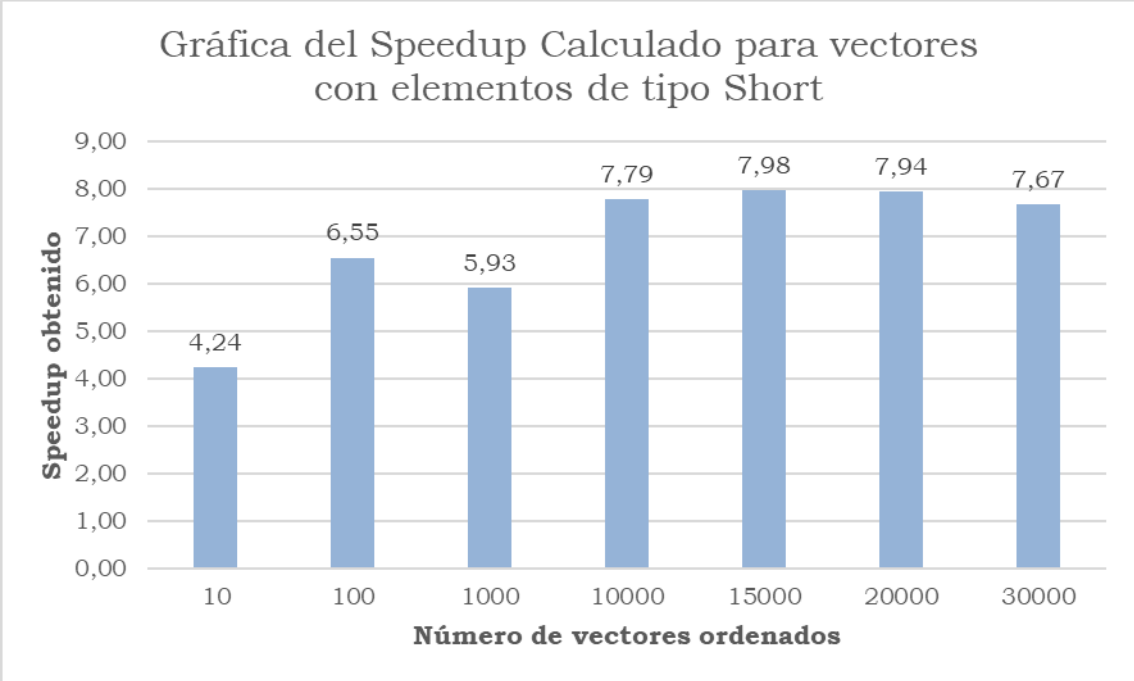

Figura 3.10 *Gráfico de barras con el speedup calculado para un orden incremental de vectores con 27400 elementos de tipo short haciendo uso de la función device segmented radix sort*

Con el objetivo de evaluar con mayor precisión la eficacia del speedup obtenido, se han llevado a cabo una serie de pruebas adicionales comparando la misma configuración utilizando la GPU, pero realizando ordenaciones individuales dentro de un bucle, en lugar de una única llamada. En este caso, la ordenación se efectúa vector a vector en cada iteración, empleando la función CUB Device Radix Sort.

De este modo, se analiza el rendimiento final al comparar el modelo de ordenación en batch con un modelo de ordenaciones independientes, lo que permite una evaluación más detallada del comportamiento y de las ventajas reales del enfoque propuesto.

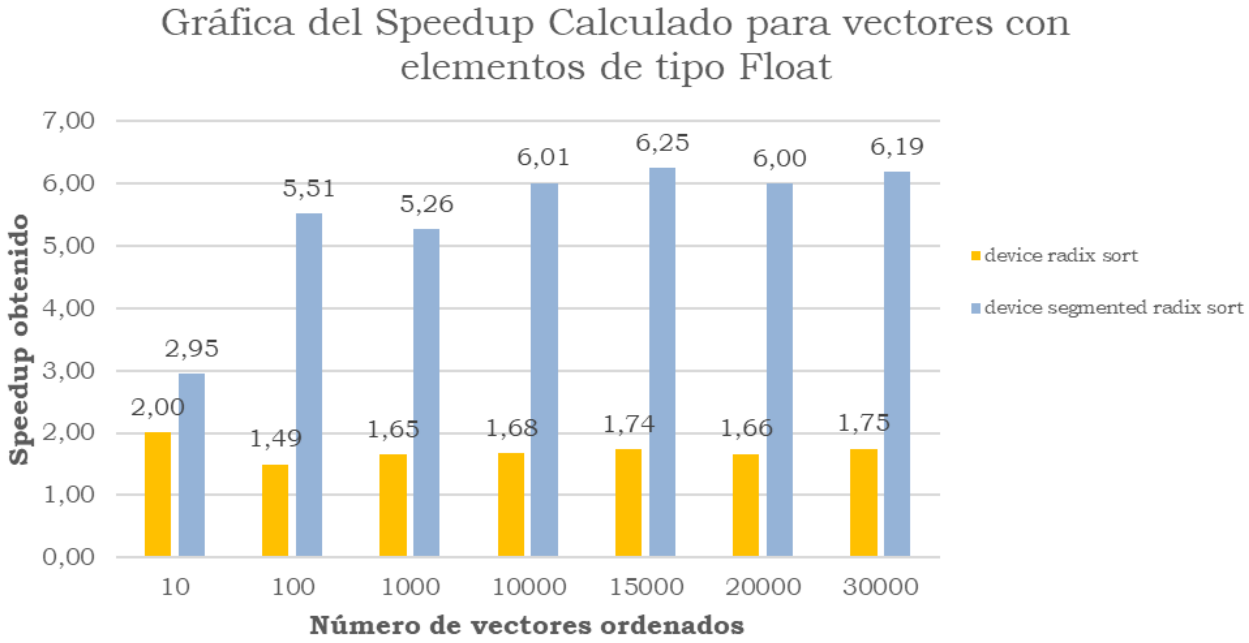

Figura 3.11 *Gráfico de barras que compara el spEedup obtenido en la ordenación de un número incremental de elementos de tipo float usando la función device segmented radix sort contra el uso de llamadas en bucle de la función device radix sort*

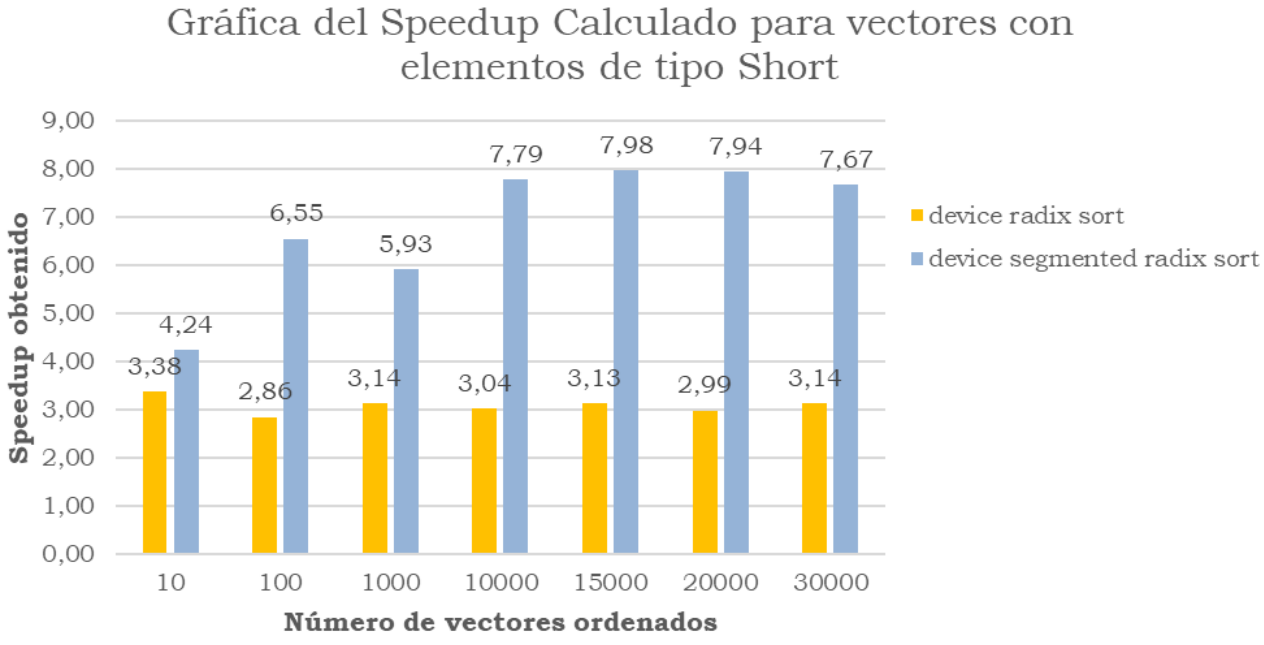

Figura 3.12 *Gráfico de barras que compara el speedup obtenido en la ordenación de un número*

Como puede observarse en los resultados mostrados en las Figuras 3.11 y 3.12, de forma general el enfoque de ordenación en batch presenta un rendimiento superior al de la ordenación vector a vector. Esta diferencia se debe principalmente a dos factores.

En primer lugar, influyen los tiempos de latencia asociados a las operaciones de transferencia y copia de datos. En el caso de la ordenación vector a vector, cada iteración implica una transferencia independiente, lo que conlleva un coste constante por operación. Sin embargo, en el enfoque de ordenación en batch este coste se amortiza, ya que la transferencia de datos se realiza una única vez. Este efecto se refleja también en la diferencia de speedup observada entre los tipos de datos short y float.

Como se analizó en el apartado 3.2.1 con la función Device Radix Sort, incluso para vectores de tamaño reducido los resultados eran más favorables cuando se empleaban datos de tipo float. No obstante, en este experimento los resultados obtenidos son ligeramente mejores al utilizar datos de tipo short. Esto se explica porque, dado el reducido número de elementos por vector, el speedup individual de cada ordenación es limitado, lo que provoca que el impacto relativo de los tiempos de transferencia sea mayor.

En segundo lugar, al ejecutar todas las ordenaciones de manera simultánea, y considerando que el tamaño de los arrays es relativamente pequeño, se logra una mejor ocupación de los recursos de la GPU, facilitando una explotación más eficiente del paralelismo disponible. En contraste, al ordenar un único vector, la ejecución puede llegar a ocupar aproximadamente tres bloques de ejecución, mientras que la GPU está diseñada para gestionar de forma óptima miles de bloques concurrentes.

3.3.1.4 Puntos destacables y desventajas

Esta función presenta una serie de aspectos destacables en su utilización. Al igual que en el apartado 3.2.1, al trabajar con una función perteneciente a la librería CUB, no es necesario añadir ni compilar repositorios externos, lo que simplifica el proceso de desarrollo. Además, se dispone de una cantidad significativa de documentación y soporte, lo que facilita tanto la implementación como la resolución de posibles incidencias. Su integración resulta relativamente sencilla, ya que únicamente requiere la incorporación de unos pocos pasos adicionales respecto a la implementación de la función Device Radix Sort descrita en el apartado 3.2.1, tratándose además de una función optimizada específicamente para este propósito.

Por otro lado, una de las principales limitaciones de este algoritmo, compartida con otras implementaciones de ordenación en batch, es el incremento en el consumo de memoria. Es necesario almacenar de forma simultánea todos los vectores que se desea ordenar tanto en la GPU como en la CPU, lo que puede resultar restrictivo en entornos con recursos limitados.

Adicionalmente, esta función no se adapta de manera eficiente a entornos de ejecución altamente paralelos. En particular, la ejecución simultánea de la misma función con más de dos hilos concurrentes puede llegar a degradar significativamente el rendimiento. Este comportamiento se mantiene incluso al emplear CUDA Streams, debido a que los tiempos de transferencia de datos

adquieren un peso considerable en el tiempo total de ejecución. Esto se verá en más profundidad en los siguientes apartados, cuando se trabaje con la implementación en la aplicación para el cálculo de la anomalía climática.

3.3.2 ModernGPUs Segmented Radix Sort

3.3.2.1 Descripción

La función segmented sort (segsort) de la librería ModernGPU [16], al igual que la descrita en el apartado 3.3.1, realiza una ordenación segmentada sobre un vector. Este proceso consiste en ordenar un conjunto de segmentos o subsecuencias contiguas e independientes, manteniendo intacta la separación entre los distintos segmentos.

A diferencia de enfoques más tradicionales, que asignan una cantidad fija de trabajo por bloque o por kernel, la implementación de segsort en ModernGPU emplea estrategias avanzadas de balanceo dinámico de carga (load balancing), evitando que los recursos de la GPU queden infrautilizados.

Una de las técnicas clave utilizadas es el empleo de colas de trabajo (work queues), que permiten distribuir dinámicamente los segmentos entre los hilos disponibles. En lugar de asignar de forma estática un segmento completo a un bloque de ejecución, los segmentos se dividen en unidades de trabajo más pequeñas, que se procesan conforme los hilos van quedando libres. Este enfoque reduce de forma significativa la divergencia de los hilos (que cada hilo tome rutas alternativas forzando a la GPU y reduciendo el paralelismo [28]) y el desequilibrio entre bloques, problemas habituales en la ordenación de segmentos de tamaño desigual.

Este tipo de implementación presenta una ventaja notable cuando se trabaja con grupos de datos heterogéneos, especialmente en situaciones donde existe una disparidad significativa en el tamaño de los segmentos. Sin embargo, cuando los segmentos son relativamente homogéneos, las diferencias de rendimiento con respecto a otras funciones similares, como Device Segmented Radix Sort de CUB, resultan menos notables.

Por otro lado, la implementación de esta función es considerablemente más compleja, al operar a un nivel de abstracción más bajo. Además, para poder utilizarla es necesario clonar e integrar manualmente el repositorio de ModernGPU, ya que, a diferencia de las funciones incluidas en CUB, esta librería no se encuentra integrada en las versiones actuales de las herramientas oficiales de NVIDIA. En el siguiente apartado se detallará con mayor profundidad el proceso de integración y uso de esta función.

3.3.2.2 Implementación

Para poder hacer uso de esta función, el primer paso consiste en disponer de la librería ModernGPU. Para ello, es necesario seguir los siguientes pasos:

En primer lugar, se debe clonar el repositorio oficial de ModernGPU [29]. Una vez clonado el repositorio, se procede siguiendo las instrucciones proporcionadas por los propios desarrolladores. En concreto, es necesario

compilar los archivos utilizando la herramienta de construcción automática CMake.

Durante este proceso pueden surgir diversos problemas relacionados con el archivo de configuración CMakeLists.txt, por lo que puede ser necesario corregir manualmente los errores que vayan apareciendo. Por ejemplo, uno de los inconvenientes detectados durante la realización de este trabajo fue la falta de definición explícita de la arquitectura CUDA, lo cual se solucionó añadiendo la siguiente línea al inicio del archivo de configuración:

```
set(CMAKE_CUDA_ARCHITECTURES native)
```

Una vez compilado correctamente el repositorio, la función puede ser referenciada desde un programa externo siempre que se cumpla una serie de requisitos adicionales. En primer lugar, es necesario incluir la ruta a la librería ModernGPU en el proceso de compilación. Un ejemplo de línea de compilación es el siguiente:

```
nvcc -o out programa.cu -I../moderngpu/src -lm
```

Asimismo, dentro del propio archivo de código fuente es necesario incluir los encabezados correspondientes, tanto el fichero que contiene la implementación de la función de ordenación segmentada como el requerido para la inicialización del contexto de ejecución. Esto se realiza mediante las siguientes directivas:

```
#include <moderngpu/kernel_segsort.hxx>
#include <moderngpu/context.hxx>
```

Una vez alcanzado este punto, es importante tener en cuenta que la implementación de esta función requiere definir explícitamente una función de comparación. Este aspecto ha resultado especialmente problemático durante el desarrollo de este trabajo. En versiones anteriores de la librería, se proporcionaba una función de comparación por defecto; sin embargo, en versiones más recientes esta debe ser definida por el usuario.

Además, dicha función de comparación debe declararse mediante extensión de lambdas, lo que incrementa la complejidad del proceso de compilación. Finalmente, la solución adoptada consistió en definir la siguiente estructura de comparación:

```
__global__ void emptyKernel() {}
struct less_float {
    __device__ __host__ bool operator()(const float &a, const
                                        float &b) const {
        return a < b;
    }
};
```

Junto con una línea de compilación adaptada que habilita el soporte de extensión de lambdas:

```
nvcc -o out programa.cu -I../moderngpu/src --extended-lambda
```

Una vez finalizada la configuración previa, se procede a la implementación de la función. Para ello, es necesario seguir los siguientes pasos:

1. Reserva inicial de memoria tanto en el host como en el dispositivo. En este caso, se utiliza un único vector de entrada y salida que contiene todos los segmentos concatenados, junto con un vector de offsets o índices. Este último tiene un tamaño equivalente al número total de segmentos que componen el vector más uno, con el fin de definir correctamente los límites de cada segmento.
2. Inicialización del contexto de ejecución, el cual puede estar asociado a un CUDA stream. En caso de considerarse necesario, es posible utilizar un stream distinto del predeterminado.
3. Inicialización del vector de offsets, que define el comienzo y el final de cada segmento independiente dentro del vector global.
4. Transferencia de los datos desde la CPU a la GPU, copiando tanto el vector de datos como el vector de offsets a la memoria del dispositivo.
5. Ejecución de la operación de ordenación segmentada, invocando la función correspondiente de ModernGPU.
6. Una vez completada la ordenación, recuperación de los datos desde la GPU a la CPU, obteniendo como resultado el vector global compuesto por todos los segmentos ordenados de forma independiente.

Entonces el código debería seguir la siguiente estructura:

```
// Inicialización del vector en CPU
…
// Inicialización de los vectores en GPU
float* device_vector;
cudaMalloc(&device_vector, M*N*sizeof(float));
// inicialización del vector de índices
int h_offsets[M+1];
for(int i = 0; i < M+1; i++) h_offsets[i] = (i*N);
int* d_offsets;
cudaMalloc(&d_offsets, (M+1)*sizeof(int));
cudaMemcpy(d_offsets, h_offsets,
      (M+1)*sizeof(int), cudaMemcpyHostToDevice);
// inicialización del contexto
mgpu::standard_context_t context;
// Transferencia del vector de segmentos de CPU a GPU
cudaMemcpy(device_vector, vector, N*M*sizeof(float),
      cudaMemcpyHostToDevice);
// Ordenación de los segmentos
mgpu::segmented_sort(device_vector, N*M, d_offsets, M,
      less_float(), context);
// Transferencia del vector de segmentos de GPU a CPU
cudaMemcpy(sorted_vector, device_vector,  N*M*sizeof(float),
      cudaMemcpyDeviceToHost);
```

…

3.3.2.3 Rendimiento

Para evaluar el rendimiento de la función, esta se ha sometido a las mismas pruebas que la función CUB Device Segmented Sort, descritas en el apartado 3.3.1. Dichas pruebas consisten en una serie de mediciones de tiempo en las que se ordena simultáneamente un número incremental de vectores de 27.400 elementos mediante esta función, comparando los resultados con un proceso de ordenación secuencial ejecutado en CPU.

Las gráficas 3.13 y 3.14 muestran el speedup alcanzado mediante el uso de esta función para vectores con elementos de tipo float y short, respectivamente.

Como puede observarse en los resultados obtenidos, al igual que en el apartado 3.3.1 con la función Device Segmented Radix Sort, el rendimiento mejora ligeramente cuando los datos presentan un menor tamaño. No obstante, la eficiencia general de esta función no llega a superar a la obtenida con Device Segmented Radix Sort de CUB (apartado 3.3.1), presentando speedups ligeramente inferiores.

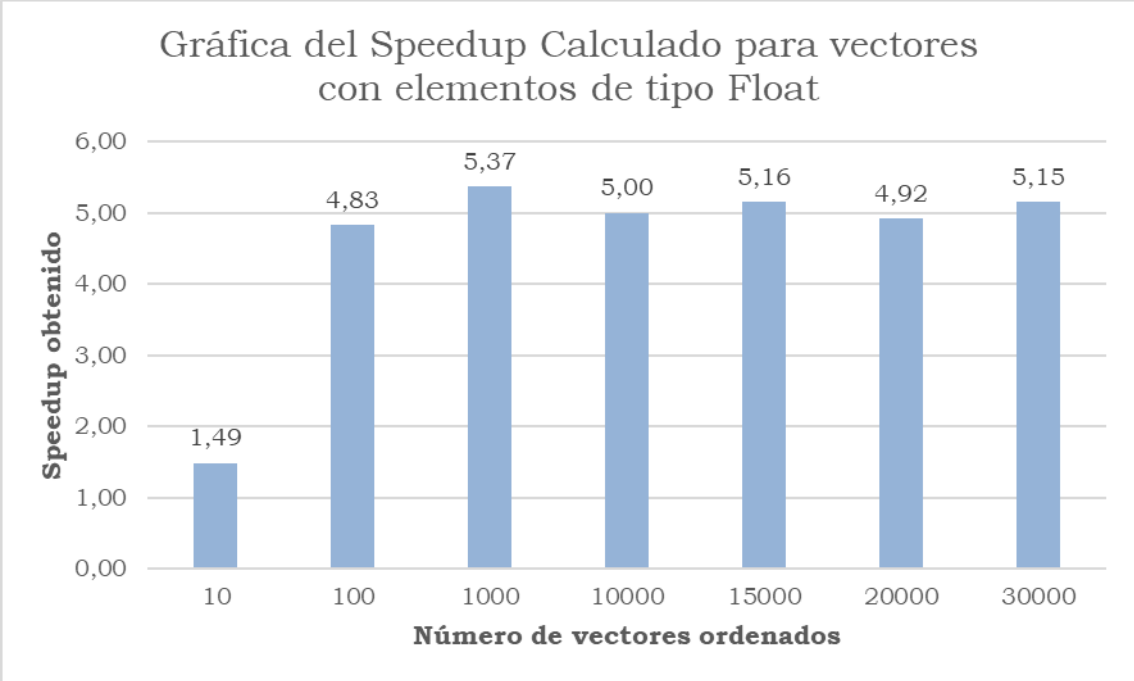

Figura 3.13 *Gráfico de barras que muestra el speedup obtenido en la ordenación de un número incremental de vectores con elementos de tipo float usando la función Segmented Sort de la librería moderngpu*

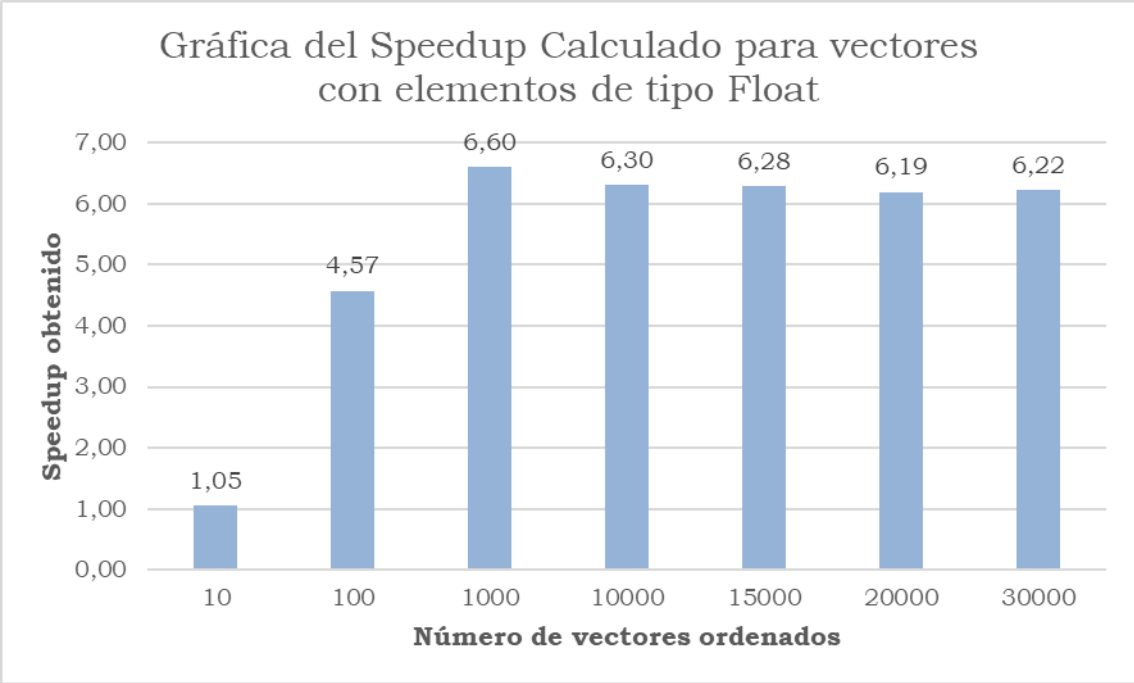

FIGURA 3.14 *GRÁFICO DE BARRAS QUE MUESTRA EL SPEEDUP OBTENIDO EN LA ORDENACIÓN DE UN NÚMERO INCREMENTAL DE VECTORES CON ELEMENTOS DE TIPO SHORT USANDO LA FUNCIÓN SEGMENTED SORT DE LA LIBRERÍA MODERNGPU*

3.3.2.4 Puntos destacables y desventajas

De esta función pueden extraerse varios puntos destacables. En primer lugar, al tratarse de una implementación de más bajo nivel, ofrece una mayor libertad en el diseño y control de la implementación en comparación con la anterior. Asimismo, el uso de un único vector en GPU que actúa tanto como entrada como salida permite reducir los requisitos de memoria dentro del dispositivo.

Sin embargo, tal y como se ha observado en los resultados de rendimiento obtenidos, desde un punto de vista más pragmático y considerando su aplicación al cálculo de la anomalía climática, los resultados no parecen lo suficientemente competitivos. De hecho, el rendimiento se sitúa ligeramente por detrás del alcanzado por la implementación de la función Device Radix Sort de la librería CUB. Esta función parece estar considerablemente mejor optimizada para escenarios en los que existe una mayor heterogeneidad en el tamaño de los segmentos. No obstante, esta característica no puede ser aprovechada en el entorno real de aplicación, ya que para el cálculo de la anomalía climática todos los vectores o segmentos presentan el mismo tamaño.

Por otro lado, para poder ejecutar y utilizar esta función es necesario clonar y compilar el repositorio correspondiente. Aunque este procedimiento es habitual en el ámbito del desarrollo de software, supone la introducción de pasos adicionales y, por tanto, de un mayor margen de error. Adicionalmente, la necesidad de definir explícitamente una función de comparación de elementos, especialmente teniendo en cuenta que en versiones anteriores existía una implementación por defecto, representa un nuevo elemento de complejidad en el proceso de integración.

Finalmente, cabe destacar que el repositorio no ha recibido actualizaciones en al menos cinco años, lo que plantea dudas acerca de la fiabilidad y rapidez del

soporte en comparación con una librería consolidada y activamente mantenida como CUB.

3.3.3 Fast Segmented Sort

3.3.3.1 Descripción

Fast Segmented Sort [14] es otra función de ordenación segmentada, muy similar a la de los dos apartados anteriores (3.3.1 y 3.3.2). Esta función permite ordenar múltiples segmentos o secciones independientes y contiguas dentro de un único vector, si bien introduce algunos matices relevantes en su implementación.

Entre las mejoras propuestas destaca la realización de un análisis y clasificación previa de los segmentos con el objetivo de optimizar su distribución y favorecer un uso más eficiente de los recursos de la GPU, especialmente en escenarios en los que existe una mayor heterogeneidad en el tamaño de los segmentos.

La función presenta características muy similares a la implementación de la función Segmented Sort de la librería ModernGPU. No obstante, los autores del algoritmo afirman obtener resultados superiores a algunas implementaciones tanto de ModernGPU como de CUB. En el presente caso, y a la luz de las pruebas realizadas, estos resultados no parecen confirmarse. Este aspecto se analiza con mayor detalle en los siguientes puntos.

3.3.3.2 Implementación

Al igual que ocurría con la función Segmented Sort de la librería ModernGPU, para poder utilizar esta función es necesario clonar el repositorio [30] en el que se encuentra alojada en el entorno local donde se pretende emplear. Esto, al igual que en el caso anterior, implica una serie de pasos adicionales. No obstante, debido a la forma en que está estructurada la función, no debería ser necesario realizar configuraciones adicionales más allá de referenciarla correctamente dentro del código del programa. Esto se puede hacer con la siguiente línea:

```
#include "../bb_segsort/bb_segsort.h"
```

Por otro lado, el repositorio únicamente proporciona una implementación basada en el esquema de ordenación de estructuras de tipo clave-valor. Es decir, la función ordena de manera conjunta dos vectores: uno de claves y otro de valores, utilizando el vector de claves como criterio de comparación. Esta característica introduce ciertas limitaciones en términos de rendimiento, ya que el consumo de memoria se duplica al requerir dos vectores y, adicionalmente, la ordenación de un segundo vector conlleva un coste computacional no despreciable.

Los pasos necesarios para la implementación de esta función coinciden con los descritos en el apartado 3.3.2 para la función Segmented Sort de ModernGPU:

1. Inicializar el vector de segmentos, que corresponderá al vector de claves.

2. Inicializar un segundo vector de la misma longitud, que actuará como vector de valores. En este caso, al no ser relevante, puede permanecer vacío, aun cuando sí es necesario reservarlo.
3. Inicializar un vector de índices de longitud igual al número de segmentos más uno, que permita indicar el inicio y el final de cada segmento.
4. Reservar memoria suficiente en la GPU para el vector de segmentos y el vector de valores.
5. Transferir desde la CPU a la GPU los datos correspondientes al vector de segmentos y al vector de índices.
6. Invocar la función de ordenación.
7. Recuperar de la GPU, una vez finalizada la ordenación de los segmentos, el resultado de la ejecución, transfiriendo el vector de segmentos ordenados de vuelta a la CPU.

Estos pasos se traducen en el siguiente esquema de código:

```
// N = Tamaño de los segmentos
// M = Número de Segmentos
// Inicialización del vector en CPU
…
// Inicialización de los vectores en GPU
float* device_keys; // vector de segmentos o claves
float* device_values;
cudaMalloc(&device_keys, M*N*sizeof(float));
cudaMalloc(&device_values, M*N*sizeof(float));
// inicialización del vector de índices
int h_offsets[M+1];
for(int i = 0; i < M+1; i++) h_offsets[i] = (i*N);
int* d_offsets;
cudaMalloc(&d_offsets, (M+1)*sizeof(int));
cudaMemcpy(d_offsets, h_offsets,
      (M+1)*sizeof(int), cudaMemcpyHostToDevice);
// Transferencia del vector de segmentos o claves de CPU a GPU
cudaMemcpy(device_keys, vector, N*M*sizeof(float),
      cudaMemcpyHostToDevice);
// Ordenación de los segmentos
bb_segsort<float, float>(device_keys, device_values, M*N,
      d_offsets, M)
// Transferencia del vector de segmentos de GPU a CPU
cudaMemcpy(sorted_vector, device_keys, N*M*sizeof(float),
      cudaMemcpyDeviceToHost);
…
```

3.3.3.3 Rendimiento

Para evaluar el rendimiento de esta función se han realizado las mismas pruebas que en el apartado 3.3.2, las cuales consisten en la ordenación de un número incremental de vectores de tamaño constante (27 400 elementos). De este modo, se ha calculado el speedup obtenido en cada caso.

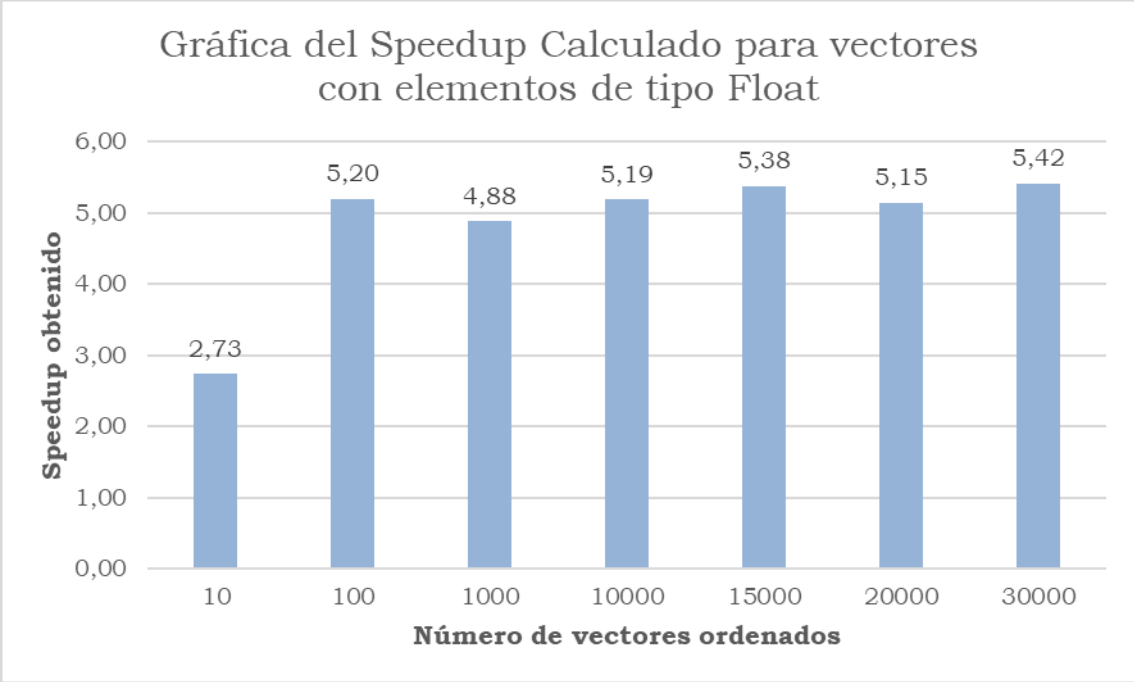

Figura 3.15 *Gráfico de barras que muestra el speedup obtenido en la ordenación de un número incremental de vectores con elementos de tipo float usando la función Fast Segmented Sort*

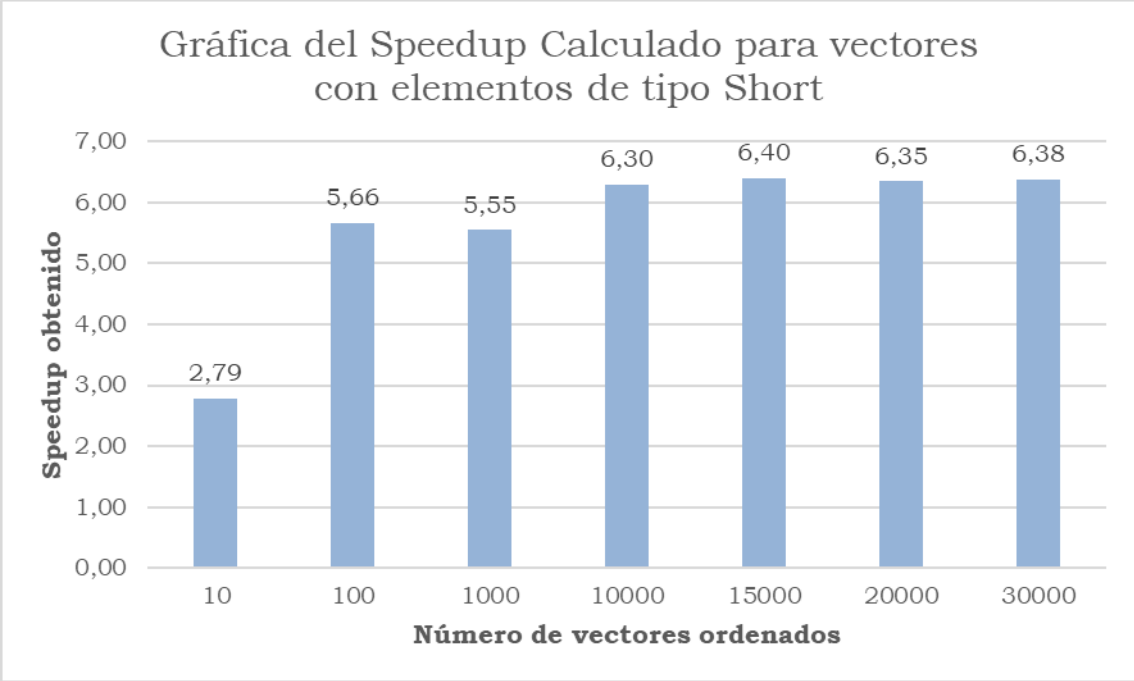

Figura 3.16 *Gráfico de barras que muestra el speedup obtenido en la ordenación de un número incremental de vectores con elementos de tipo short usando la función Fast Segmented Sort*

Como puede observarse en las gráficas 3.15 y 3.16, los resultados obtenidos son muy similares a los calculados para la función Segmented Sort de la librería

ModernGPU, situándose en algunos casos ligeramente por encima y en otros ligeramente por debajo.

Este comportamiento era esperable, especialmente si se tienen en cuenta las similitudes existentes entre ambas implementaciones. No obstante, los resultados siguen siendo discretos en relación con las mejoras de rendimiento prometidas por los autores del algoritmo.

3.3.3.4 Puntos a destacables y desventajas

Esta implementación presenta características muy similares a las estudiadas en el apartado 3.3.2 para la función Segmented Sort de la librería ModernGPU, lo que puede observarse claramente en los resultados de rendimiento obtenidos en ambos casos.

Por un lado, puede afirmarse que el uso de esta función resulta más accesible y sencillo de incorporar al programa, sin necesidad de realizar un número elevado de pasos adicionales. Sin embargo, ofrece muchas menos opciones de implementación y un menor grado de flexibilidad en comparación con librerías más consolidadas como ModernGPU y CUB, que proporcionan un mayor soporte para distintos casos de uso. En este caso, el repositorio únicamente incluye esta función, sin variantes alternativas. Esto se pone especialmente en evidencia con la necesidad de inicializar un vector de valores vacío, requerido por la función, al no existir una implementación específica para la ordenación de un único vector de claves.

Adicionalmente, los resultados obtenidos son muy similares a los alcanzados con la función Segmented Sort de ModernGPU y, en consecuencia, no superan el rendimiento de la función Device Segmented Sort de CUB. Todo ello implica que esta función deba ser descartada, ya que su utilización conlleva tanto un aumento de la complejidad de la implementación como una ligera reducción del rendimiento, de forma similar a lo observado en el apartado 3.3.2.

3.3.4 Back To Back

3.3.4.1 Descripción

Esta implementación surge a partir de una de las ideas originales descartadas en el Trabajo Fin de Carrera de Esteban Aspe Ruiz “Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática” [8], en la que se plantea, de forma análoga a la ordenación segmentada, la posibilidad de realizar múltiples ordenaciones de manera simultánea.

El enfoque propuesto se basa en el uso de dos vectores: un vector de claves y un vector de valores. El vector de claves se construye mediante la concatenación de todos los vectores que se desean ordenar, mientras que el vector de valores contiene el mismo número de elementos y actúa como identificador del vector original al que pertenece cada clave. De este modo, si se consideran, por ejemplo, seis vectores de tres elementos cada uno, los tres primeros valores del vector de identificadores serán iguales a 0 (correspondientes al primer vector), los tres siguientes iguales a 1, y así sucesivamente.

Para ilustrar este procedimiento, supóngase el siguiente conjunto de seis vectores de tres elementos:

Vectores:

```
[7, 9, 14], [41, 32, 76], [28, 2, 13], [29, 96, 88], [74, 19, 99], [5, 31, 62]
```

A partir de ellos se generan los siguientes vectores:

Vector de claves:

```
[7, 9, 14, 41, 32, 76, 28, 2, 13, 29, 96, 88, 74, 19, 99, 5, 31, 62]
```

Vector de valores:

```
[0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5]
```

Una vez definidos estos vectores, se realiza una ordenación clave-valor, emparejando cada elemento del vector de claves con su correspondiente elemento en el vector de valores, y utilizando las claves como criterio de comparación. Tras esta primera ordenación, los vectores quedan del siguiente modo:

Vector de claves:

```
[2, 5, 7, 9, 13, 14, 19, 28, 29, 31, 32, 41, 62, 74, 76, 88, 96, 99]
```

Vector de valores:

```
[2, 5, 0, 0, 2, 0, 4, 2, 3, 5, 1, 1, 5, 4, 1, 3, 3, 4]
```

A continuación, se aplica una segunda ordenación, esta vez utilizando el vector de valores como criterio de comparación. Esta operación reagrupa los elementos según su vector original, preservando el orden relativo obtenido en la ordenación previa. Como resultado, cada vector queda ordenado de forma independiente:

Vector de claves (resultado):

```
[7, 9, 14, 32, 41, 76, 2, 13, 28, 29, 88, 96, 19, 74, 99, 5, 31, 62]
```

Vector de valores:

```
[0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5]
```

De esta manera se obtiene el resultado deseado: todos los vectores originales quedan ordenados de forma independiente mediante dos ordenaciones globales.

Para la implementación de este algoritmo se ha utilizado la función Device Radix Sort Pairs de la librería CUB. Por su sencillez en la implementación y por los resultados que promete. Esta función realiza una ordenación Radix sobre un vector de claves que tiene asociado un segundo vector de valores. Lo que encaja perfectamente con el algoritmo a implementar.

3.3.4.2 Implementación

Para implementar este algoritmo, tal y como se describía en el apartado anterior, se ha hecho uso de la función CUB Device Radix Sort Pairs, muy similar a la función CUB Device Radix Sort analizada en el apartado 3.2.1, aunque con algunas pequeñas modificaciones. Además, al tratarse de una función incluida dentro de la librería CUB, no es necesario descargar ni compilar ningún archivo adicional.

1. Los pasos necesarios para la implementación del algoritmo son los siguientes:
2. Inicializar en la CPU los vectores de entrada: el vector de segmentos (claves) y el vector de identificadores.
3. Reservar memoria suficiente en la GPU para los vectores de entrada y salida, al menos dos vectores de entrada y dos de salida: uno de entrada y uno de salida para el vector de segmentos, y otro par para el vector de identificadores.
4. Transferir a la GPU los vectores alojados en CPU.
5. Invocar la función de ordenación utilizando el vector de segmentos como vector de claves y el vector de identificadores como vector de valores.
6. Una vez finalizada la primera ordenación, volver a invocar la función, intercambiando los roles de los vectores: el vector de identificadores actúa como vector de claves y el vector de segmentos como vector de valores.
7. Finalizada la segunda llamada, recuperar de la GPU los datos correspondientes al vector de segmentos, que contendrá los segmentos o subvectores ordenados de forma independiente.

Una vez descritos los pasos necesarios para la implementación, la estructura del código debería ser similar a la siguiente:

```
// N = Tamaño de los segmentos
// M = Número de Segmentos
// Inicialización del vector de segmentos en CPU
float vector[N*M]; // vector de entrada
float sorted_vector[N*M]; // vector de salida
...
// Inicialización del vector de identificadores o índices
float identificadores[N*M];
for(int i = 0; i < M; i++)
      for(int j = 0; j < N; j++)
            identificadores[(i*N)+j] = (float) i;
// Inicialización de los vectores en GPU
float* device_keys_in; // entrada
float* device_values_in; //entrada
float* device_keys_out; // salida
float* device_values_out; // salida
cudaMalloc(&device_keys_in, M*N*sizeof(float));
cudaMalloc(&device_values_in, M*N*sizeof(float));
cudaMalloc(&device_keys_out, M*N*sizeof(float));
cudaMalloc(&device_values_out, M*N*sizeof(float));
// Llamada de inicialización
cudaStream_t stream; // definición del cuda stream
cudaStreamCreate(&stream);
```

```
void* d_temp_storage = NULL; // almacenamiento temporal
size_t temp_storage_bytes = 0;
cub::DeviceRadixSort::SortPairs(d_temp_storage,
      temp_storage_bytes, device_keys_in, device_keys_out,
      device_values_in, device_values_out, M*N, stream);
cudaMalloc(&d_temp_storage, temp_storage_bytes);
// Transferencia de los vectores de CPU a GPU
cudaMemcpyAsync(device_keys_in, vector, N*M*sizeof(float),
      cudaMemcpyHostToDevice, stream);
cudaMemcpyAsync(device_values_in, identificadores,
      N*M*sizeof(float), cudaMemcpyHostToDevice, stream);
// Primera ordenación
cub::DeviceRadixSort::SortPairs(d_temp_storage,
      temp_storage_bytes, device_keys_in, device_keys_out,
      device_values_in, device_values_out, M*N, stream);
// Segunda ordenación
cub::DeviceRadixSort::SortPairs(d_temp_storage,
      temp_storage_bytes,  device_values_out,  device_values_in,
      device_keys_out, device_keys_in, M*N, stream);
// Transferencia del vector de segmentos de GPU a CPU
cudaMemcpyAsync(sorted_vector, device_keys_in, N*M*sizeof(float),
      cudaMemcpyDeviceToHost, stream);
cudaStreamSynchronize(stream);
...
```

3.3.4.3 Rendimiento

Para evaluar el rendimiento de esta implementación, se ha sometido a las mismas pruebas descritas en los apartados anteriores. En concreto, se ha inicializado un vector con valores aleatorios, compuesto por segmentos independientes que deben ser ordenados. Del mismo modo que en las pruebas previas, el speedup se ha calculado comparando los tiempos de ejecución obtenidos con los de una ejecución secuencial en CPU.

Los valores resultantes pueden observarse en las gráficas 3.17 y 3.18, correspondientes al caso de vectores con datos de tipo float y short, respectivamente.

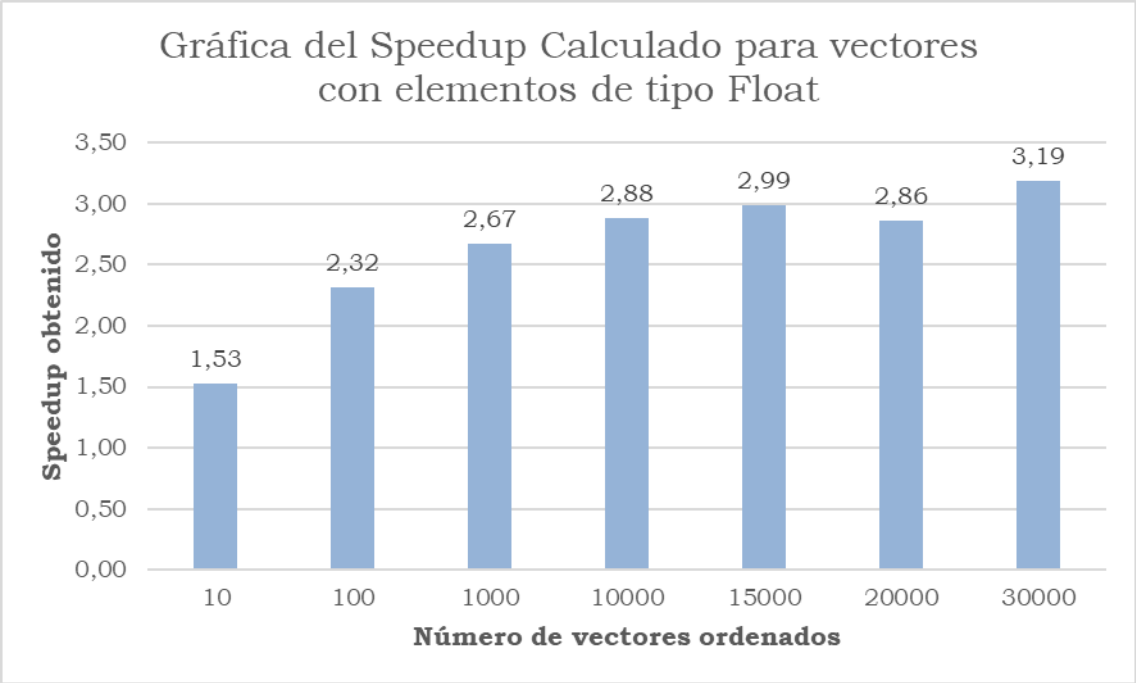

FIGURA 3.17 *GRÁFICO DE BARRAS QUE MUESTRA EL SPEEDUP OBTENIDO EN LA ORDENACIÓN DE UN NÚMERO INCREMENTAL DE VECTORES CON ELEMENTOS DE TIPO FLOAT OBTENIDOS CON UNA IMPLEMENTACIÓN BACK TO BACK USANDO LA FUNCIÓN DEVICE RADIX SORT PAIRS*

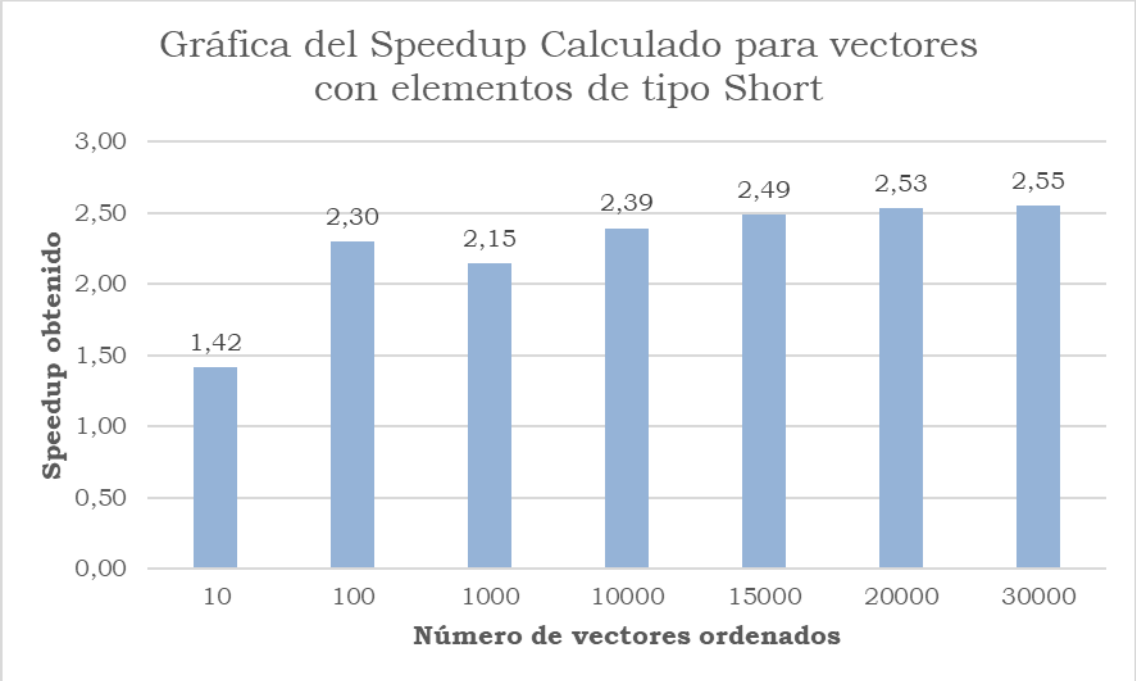

FIGURA 3.18 *GRÁFICO DE BARRAS QUE MUESTRA EL SPEEDUP OBTENIDO EN LA ORDENACIÓN DE UN NÚMERO INCREMENTAL DE VECTORES CON ELEMENTOS DE TIPO SHORT OBTENIDOS CON UNA IMPLEMENTACIÓN BACK TO BACK USANDO LA FUNCIÓN DEVICE RADIX SORT PAIRS*

Como puede observarse a partir de los datos obtenidos, el rendimiento de esta implementación es sensiblemente inferior al de las otras implementaciones analizadas. Este comportamiento se debe, de manera general, a dos factores principales.

En primer lugar, la implementación requiere el uso de un vector adicional para la identificación de los elementos pertenecientes a cada segmento. Esto conlleva una fase de inicialización previa adicional y un incremento en la cantidad de datos que deben transferirse desde la CPU a la GPU, lo que introduce un tiempo de cómputo no despreciable.

En segundo lugar, para completar correctamente esta implementación es necesario realizar dos llamadas a la GPU en lugar de una única llamada, lo que incrementa los tiempos de ejecución debido a las sincronizaciones requeridas.

3.3.4.4 Puntos destacables y desventajas

Como se mencionaba en el apartado anterior, se ha podido observar que esta implementación, a pesar de su carácter original, no supone una mejora respecto a otras implementaciones de características similares, sino que presenta un rendimiento significativamente inferior. Además, la necesidad de inicializar vectores adicionales incrementa la complejidad de la solución sin aportar beneficios proporcionales.

Por todo ello, se descarta esta propuesta al no resultar una implementación óptima para el caso de estudio abordado en este trabajo de fin de carrera.

3.3.5 Variante Back To Back

3.3.5.1 Descripción

Siguiendo la línea del apartado anterior (3.3.4), en esta sección se estudia una implementación derivada de la versión back-to-back previamente analizada. Dado que los resultados obtenidos no fueron suficientemente atractivos y considerando la posibilidad de optimización, se propuso el análisis de esta nueva implementación.

En esta propuesta se plantea la unión de los dos vectores empleados anteriormente, el vector de valores y el vector de identificadores, en una única estructura. Para ello, se aprovecha el uso del tipo de datos short, lo que permite realizar esta operación de forma sencilla, mientras que con datos en coma flotante el proceso resultaría considerablemente más complejo.

Al combinar ambos vectores en uno solo, es suficiente realizar una única ordenación, a diferencia de la implementación anterior, que requería dos pasadas independientes.

Aunque los detalles de esta implementación se describirán en mayor profundidad en los siguientes apartados, a continuación, se presenta una breve descripción general:

La implementación está diseñada para trabajar con datos de tipo short, es decir, enteros de 16 bits, en este caso todos positivos. Para ello, se inicializa un vector de tipo int (enteros de 32 bits) del mismo tamaño que el vector original de valores. Los 16 bits menos significativos de cada elemento almacenan el valor original, mientras que los 16 bits más significativos contienen el identificador del segmento al que pertenece dicho elemento.

De esta forma, al ordenar el vector resultante, los segmentos quedan ordenados de manera independiente sin necesidad de una segunda pasada de ordenación.

Para llevar a cabo este proceso se ha empleado la función device radix sort de la librería CUB, de la misma manera que se planteaba en el apartado 3.2.1.

3.3.5.2 Implementación

Para esta implementación, se seguirán los mismos pasos utilizados en la implementación de la función Device Radix Sort presentada en el apartado 3.2.1, introduciendo únicamente algunas modificaciones menores, como la inicialización adicional del vector compuesto.

1. En consecuencia, los pasos a seguir son los siguientes:
2. Inicialización del vector compuesto a partir del vector original de segmentos.
3. Reserva de memoria en el dispositivo (GPU), que en este caso comprende un vector de entrada y un vector de salida correspondientes al vector compuesto.
4. Realización de una llamada de inicialización para las variables internas y de la memoria temporal requerida.
5. Transferencia de los datos almacenados en la CPU a la GPU.
6. Ordenación del vector compuesto.
7. Una vez finalizado el proceso de ordenación, transferencia del vector resultante desde la GPU a la CPU, que corresponderá al vector ordenado por segmentos independientes.
8. Finalmente, recuperación de los elementos del vector compuesto para reconstruir el vector original de segmentos ya ordenado.

Siguiendo estos pasos, debería resultar una estructura de código parecida a la siguiente:

```
// N = Tamaño de los segmentos
// M = Número de Segmentos
// Inicialización del vector de compuesto en CPU
int* merged_vector = (int*) malloc(N*M*sizeof(int));
for(int i = 0; i < M; i++)
      for(int j = 0; j < N; j++)
            merged_vector[(i*N)+j] = i<<16 + vector[(i*N)+j];
// Inicialización de los vectores en GPU
int* device_vector;
int* device_sorted_vector;
cudaMalloc(&device_vector, M*N*sizeof(int));
cudaMalloc(&device_sorted_vector, M*N*sizeof(int));
// Llamada de prueba
void* d_temp_storage = NULL;
size_t temp_storage_bytes = 0;
cub::DeviceRadixSort::SortKeys(d_temp_storage,
```

```
      temp_storage_bytes, device_vector,
      device_sorted_vector, M*N);
cudaMalloc(&d_temp_storage, temp_storage_bytes);
// Ordenación en GPU
cudaMemcpy(device_vector, merged_vector, M*N*sizeof(int),
      cudaMemcpyHostToDevice);
cub::DeviceRadixSort::SortKeys(d_temp_storage,
      temp_storage_bytes, device_vector,
      device_sorted_vector, M*N);
cudaMemcpy(merged_vector, device_sorted_vector, M*N*sizeof(int),
      cudaMemcpyDeviceToHost);
// reconstrucción del vector de segmentos ordenado
for(int i = 0; i < M*N; i++)
      sorted_vector[i] = (short) (merged_vector[i] & 65535);
…
```

3.3.5.3 Rendimiento

Para evaluar el rendimiento de esta implementación, al igual que en apartados anteriores, se ha llevado a cabo una serie de pruebas en las que se establece un conjunto incremental de vectores del mismo tamaño, los cuales se ordenan de manera independiente. No obstante, en este caso los experimentos se han realizado únicamente con un tipo de dato, concretamente elementos de tipo short, esto por las limitaciones mencionadas anteriormente.

Una vez recopilados los resultados, se ha calculado el speedup correspondiente a esta implementación. Los valores obtenidos se muestran en la gráfica 3.19.

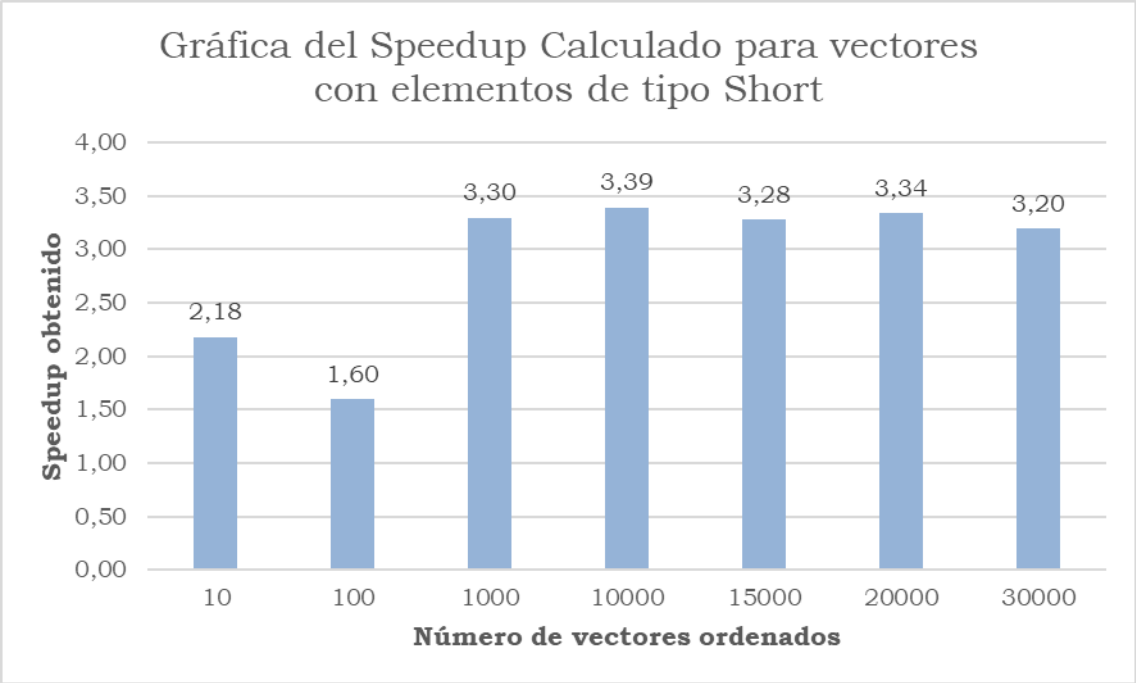

FIGURA 3.19 *GRÁFICO DE BARRAS QUE MUESTRA EL SPEEDUP OBTENIDO EN LA ORDENACIÓN DE UN NÚMERO INCREMENTAL DE VECTORES CON ELEMENTOS DE TIPO SHORT OBTENIDOS CON UNA IMPLEMENTACIÓN DERIVADA DE BACK TO BACK USANDO LA FUNCIÓN DEVICE RADIX SORT.*

Como puede observarse en los resultados obtenidos, el rendimiento alcanzado es ligeramente superior al de la implementación back-to-back analizada en el apartado 3.3.4. No obstante, estos resultados continúan estando por debajo de los obtenidos mediante implementaciones basadas en funciones específicas para la ordenación segmentada, aquellas estudiadas en los apartados 3.3.1, 3.3.2 y 3.3.3.

3.3.5.4 Análisis

Esta implementación, aunque cumple con el objetivo de mejorar el rendimiento de la versión back-to-back, todavía no alcanza los resultados esperados. Esto se debe en gran medida, al igual que ocurría con la implementación back-to-back, a la necesidad de inicializar el vector compuesto, lo que introduce un tiempo de ejecución adicional no despreciable.

Adicionalmente, esta propuesta presenta una limitación importante en cuanto a los tipos de datos que puede manejar, ya que actualmente se restringe a enteros de 16 y 32 bits, sin permitir el uso eficiente de datos en coma flotante.

Teniendo en cuenta tanto el rendimiento insuficiente como el aumento de la complejidad que introduce esta implementación, se ha decidido, al igual que en el caso de la versión back-to-back, descartarla para su utilización en el entorno final.

3.3.6 Implementación Híbrida

Un aspecto relevante a tener en cuenta es que, en este caso, se está comparando el rendimiento con una implementación basada en la paralelización a nivel de

hilos en CPU. En particular, utilizando los 16 hilos físicos disponibles en el sistema.

Sometiendo esta implementación a las mismas pruebas que en los apartados anteriores, consistentes en la ordenación paralela de un número incremental de vectores independientes, se ha calculado el speedup correspondiente. Los resultados obtenidos se muestran en las gráficas 3.20 y 3.21, donde se presentan los casos de vectores con datos de tipo float y short, respectivamente.

Como puede observarse a partir de estos resultados, las implementaciones basadas en una ordenación puramente en GPU quedan significativamente por detrás en términos de rendimiento. No obstante, un punto a favor del uso de la GPU es que, incluso trabajando con múltiples hilos en CPU, esta puede ejecutar de forma simultánea al cómputo en GPU, lo que abre la posibilidad de optimizar los tiempos de ejecución globales.

En este contexto, se planteó una implementación híbrida que repartiera la carga de trabajo entre CPU y GPU, permitiendo su ejecución concurrente y la obtención potencial de ciertos beneficios en rendimiento. Sin embargo, tras la realización de diversas pruebas con distintas variantes de esta aproximación, se hicieron notables una serie de inconvenientes:

En primer lugar, esta solución resultaba altamente dependiente del entorno de ejecución, de modo que en servidores con diferentes configuraciones de CPU o GPU los resultados podrían no ser óptimos. En segundo lugar, la estrategia de reparto del trabajo entre CPU y GPU introducía una complejidad excesiva en el código.

Por último, la implementación híbrida no comenzaba a mostrar mejoras apreciables hasta que la carga asignada a la GPU se reducía por debajo del 20 % del cómputo total, lo que provocaba que, incluso en los mejores casos, los resultados fueran muy similares a los obtenidos sin emplear la GPU.

Por estos motivos, esta implementación fue finalmente descartada, optándose por enfoques más generales y prácticos frente a soluciones excesivamente específicas y complejas.

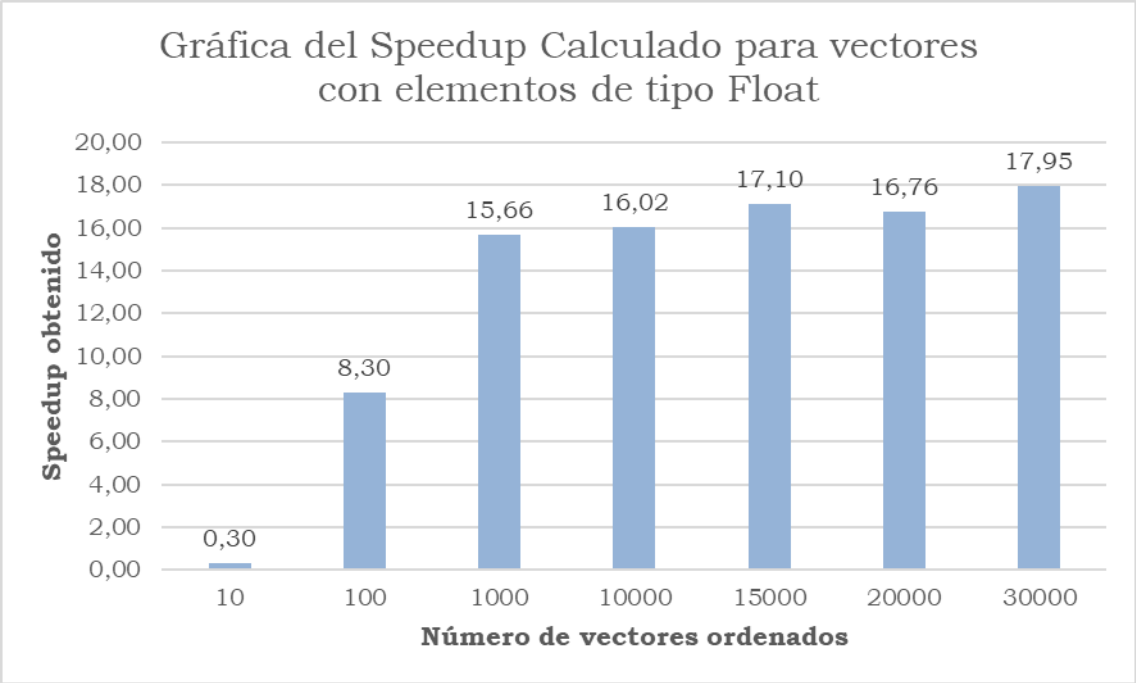

Figura 3.20 *Gráfico de barras que muestra el speedup obtenido en la ordenación de un número incremental de vectores con elementos de tipo float obtenidos con una ordenación paralela en CPU usando OpenMP*

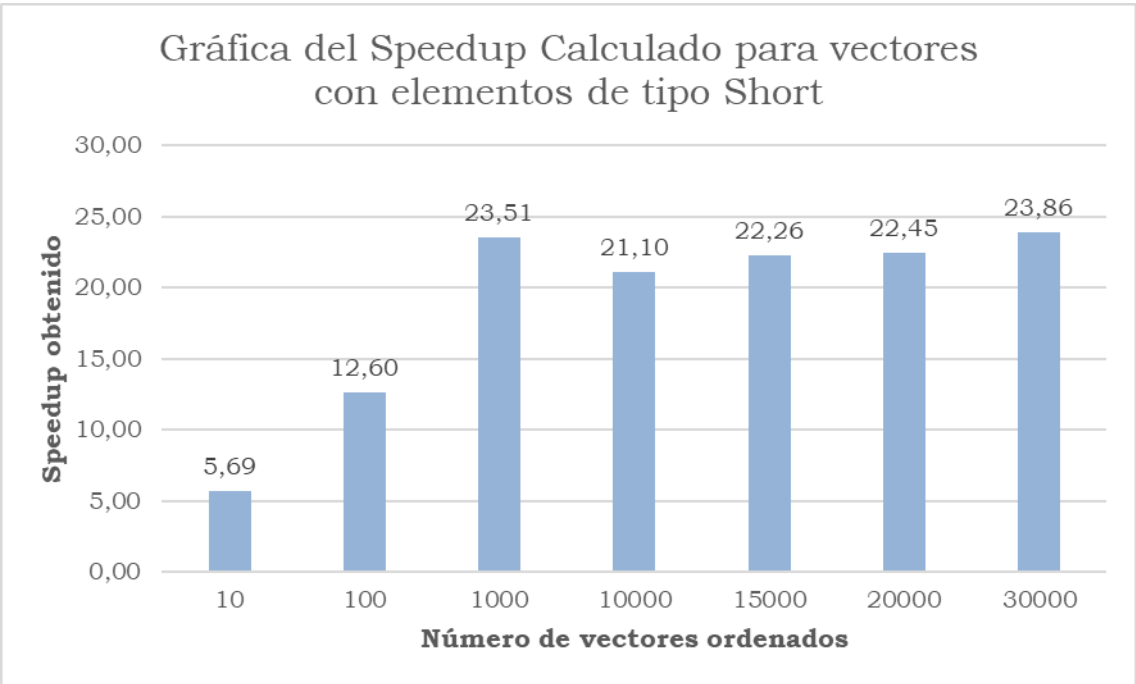

Figura 3.21 *Gráfico de barras que muestra el speedup obtenido en la ordenación de un número incremental de vectores con elementos de tipo short obtenidos con una ordenación paralela en CPU usando OpenMP*

3.4 Implantación en el entorno Real

Una vez finalizadas las pruebas y analizados los distintos algoritmos de ordenación en GPU, se han seleccionado dos candidatos finales para su implementación en el entorno definitivo destinado al cálculo de la anomalía climática. Atendiendo tanto a la simplicidad de implementación como a los resultados obtenidos, se han elegido las implementaciones proporcionadas por la librería CUB: Device Segmented Sort y Device Radix Sort.

La primera versión se basa en una ejecución de vectores independientes por iteración, en la que los hilos en ejecución se reparten dinámicamente el número de iteraciones, delegando en cada caso la ordenación del vector a la GPU. Esto implica la realización de llamadas concurrentes a la GPU, siendo esta implementación la más similar a la usada en el entorno final para el cálculo de la anomalía climática.

Por su parte, la segunda versión se fundamenta en una ejecución por lotes, en la que cada iteración está compuesta por varias ejecuciones, por lo que en vez de dividir la totalidad de las iteraciones entre el número de hilos, se paraleliza cada iteración, delegando mientras tanto la ordenación del lote de vectores en la GPU.

El funcionamiento y las particularidades de ambas aproximaciones se describirán con mayor detalle en los apartados siguientes.

3.4.1 Ordenación de un único vector por iteración

Esta implementación se basa en la distribución de las iteraciones entre el número de hilos disponibles. Dado que cada iteración corresponde al cálculo de la anomalía climática para un punto cardinal específico (longitud, latitud), y que cada uno de estos puntos dispone de un vector de datos históricos asociado, cada hilo accede a una región distinta de los datos. Este enfoque es muy similar al empleado en la implementación actual del programa para el cálculo de la anomalía climática, con la diferencia de que una de las ordenaciones se delega a la GPU.

En lugar de delegar la ordenación de ambos vectores a la GPU, únicamente uno de ellos se procesa en dicho dispositivo, con el objetivo de explotar la paralelización entre CPU y GPU. De este modo, se evita que el hilo permanezca inactivo mientras espera a la finalización del cómputo en GPU.

Los pasos que sigue esta implementación son los siguientes:

1. Dentro de la región paralela, cada hilo reserva en la GPU la memoria necesaria para albergar un vector de entrada y uno de salida, e inicializa un CUDA stream independiente.
2. A continuación, cada hilo realiza una llamada de prueba para la reserva de la memoria temporal requerida por la ordenación.
3. Una vez dentro del bucle principal, cada iteración se desarrolla del siguiente modo:
   3.1. Se leen los datos correspondientes al vector asociado a la iteración.
   3.2. Se procesan y eliminan los datos inválidos del vector. En caso de que el porcentaje de datos inválidos supere un umbral determinado, la celda se considera inválida y no se realiza el cálculo.
   3.3. Si la celda es válida, se procede a la ordenación de los vectores.

3.4. En primer lugar, se transfiere a la GPU el vector que será ordenado en dicho dispositivo y se invoca la función de ordenación correspondiente.
3.5. Aprovechando la ejecución concurrente entre CPU y GPU, mientras la ordenación en GPU está en curso, se ordena en CPU el segundo vector y se calcula a partir de este el valor de estrés correspondiente.
3.6. Una vez finalizada la ordenación en GPU, se recupera el vector ordenado y se calcula en CPU el estrés asociado al mismo.
3.7. Con ambos valores, estrés hídrico y estrés térmico, se procede al cálculo de la anomalía climática.
3.8. Finalmente, el resultado obtenido se almacena en la región de memoria correspondiente mediante una operación de escritura en el fichero.

Siguiendo este modelo de ejecución simple, se han recogido los resultados correspondientes, los cuales, como cabía esperar, han sido muy similares a los obtenidos mediante una ejecución que utiliza exclusivamente los recursos de la CPU. Esto se debe tanto a las limitaciones existentes en la paralelización efectiva entre CPU y GPU como a la sobrecarga introducida por las llamadas concurrentes a la GPU.

En la gráfica 3.22 se muestra el speedup obtenido al comparar ambas versiones: por un lado, la implementación que delega una de las ordenaciones en la GPU y, por otro, la implementación original basada únicamente en la paralelización en CPU mediante OpenMP.

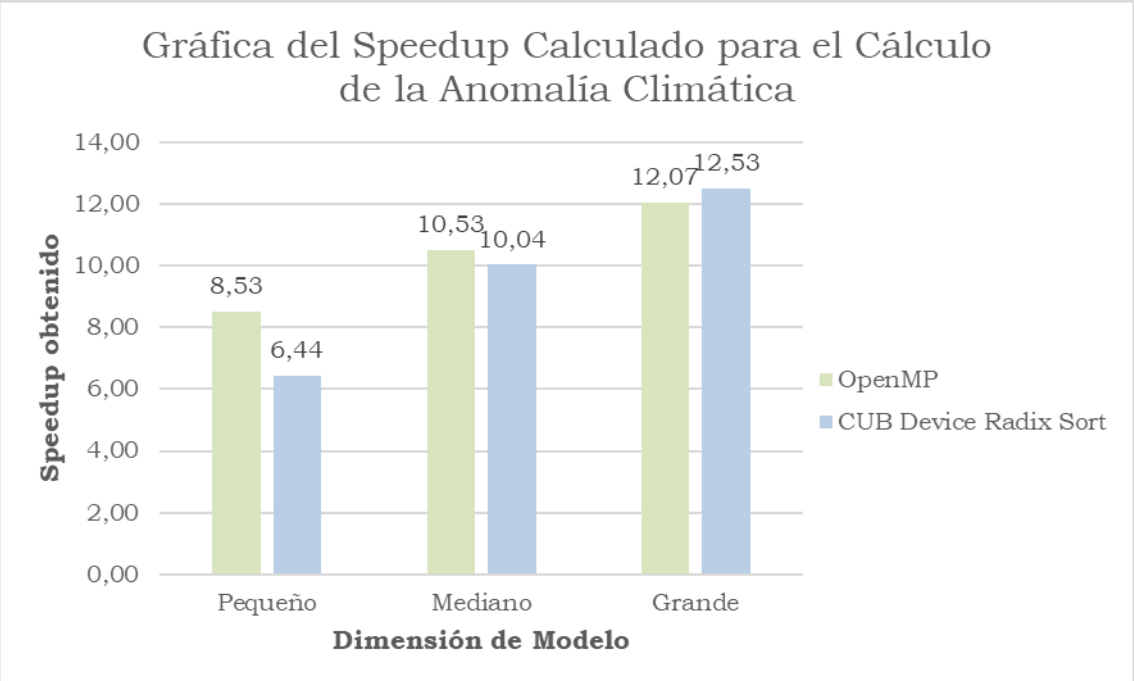

Figura 3.22 *Gráfico de barras que muestra el speedup obtenido en el cálculo de la anomalía climática dados tres modelos distintos, uno pequeño, uno mediano y uno grande. Donde se calcula para dos implementaciones distintas, una de paralelización solo en CPU con OpenMP y una segunda implementación muy similar a la primera en la que se delega la ordenación de la mitad de los vectores a la GPU mediante la función device radix sort de la librería CUB*

Como puede observarse a partir de los pasos descritos para esta implementación, el margen de paralelización efectiva entre la CPU y la GPU es limitado, ya que únicamente existen unas pocas operaciones que pueden ejecutarse en la CPU mientras la GPU está trabajando. No obstante, con el

objetivo de mejorar dicha paralelización, se ha propuesto una serie de modificaciones sobre esta implementación.

En primer lugar, se adelanta en una iteración el proceso de limpieza y preparación de los vectores, de modo que la ordenación en GPU pueda ejecutarse simultáneamente con la lectura y depuración del vector correspondiente a la siguiente iteración. Asimismo, se pospone la escritura de los resultados, realizándose esta operación mientras la GPU procesa la ordenación de los datos de la iteración previa.

Aunque esta versión introduce una complejidad considerablemente mayor respecto a la implementación inicial, permite incrementar el grado de paralelización entre CPU y GPU.

A partir de esta implementación se han obtenido los resultados correspondientes y se ha calculado el speedup resultante, el cual se muestra en la gráfica 3.23.

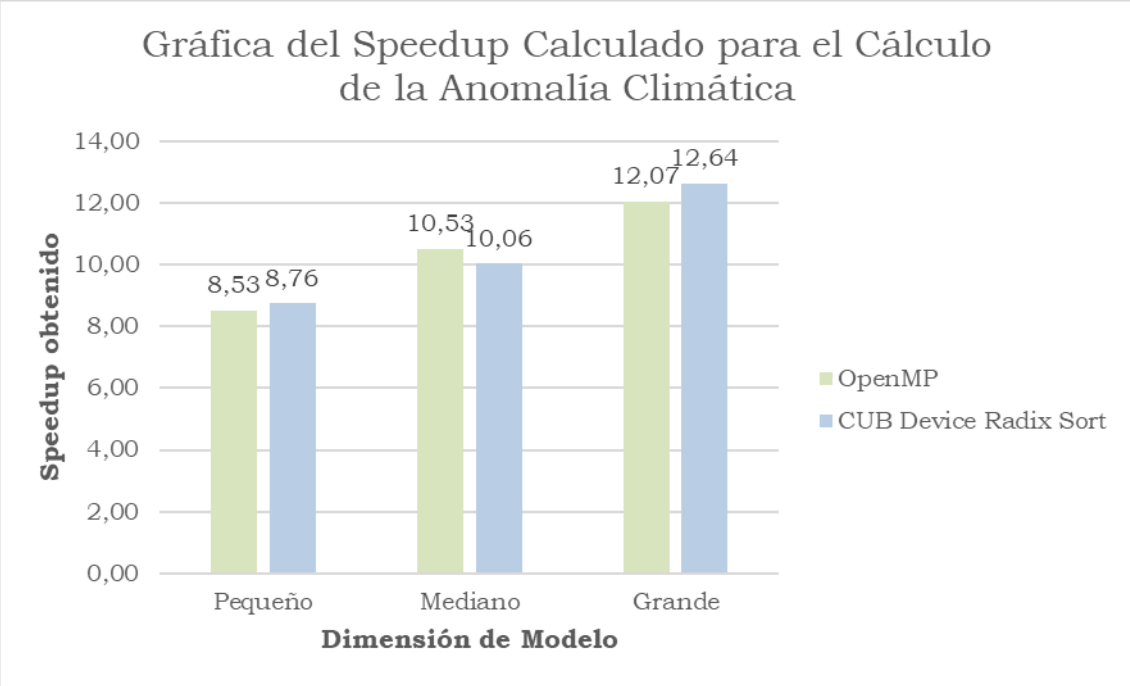

FIGURA 3.23 *GRÁFICO DE BARRAS QUE MUESTRA EL SPEEDUP OBTENIDO EN EL CÁLCULO DE LA ANOMALÍA CLIMÁTICA DADOS TRES MODELOS DISTINTOS, UNO PEQUEÑO, UNO MEDIANO Y UNO GRANDE. DONDE SE CALCULA PARA DOS IMPLEMENTACIONES DISTINTAS, UNA DE PARALELIZACIÓN SOLO EN CPU CON OPENMP Y UNA SEGUNDA IMPLEMENTACIÓN MUY SIMILAR A LA PRIMERA EN LA QUE SE DELEGA LA ORDENACIÓN DE LA MITAD DE LOS VECTORES A LA GPU MEDIANTE LA FUNCIÓN DEVICE RADIX SORT DE LA LIBRERÍA CUB*

Como puede observarse en los datos representados en la gráfica 3.23, esta implementación, aunque logra una ligera mejora respecto a los resultados anteriores, continúa ofreciendo tiempos de ejecución muy similares a los de la versión original. Sin embargo, a pesar de que las diferencias en rendimiento son prácticamente inapreciables, ambas implementaciones difieren de forma considerable en términos de complejidad.

Mientras que la primera implementación mantiene un diseño relativamente simple, la versión optimizada requiere una reestructuración sustancial del código, con la introducción de nuevas variables y un aumento en el uso de memoria. Por ejemplo, para adelantar la lectura de los datos es necesario disponer de un buffer adicional en el que almacenar la información correspondiente mientras se procesa la iteración previa.

Por otro lado, aun cuando se consigue incrementar ligeramente el grado de paralelización entre CPU y GPU, la GPU continúa actuando en muchos casos como un recurso sincronizante, al que es necesario esperar, lo que limita las ventajas que podría aportar en este escenario concreto.

3.4.2 Ordenación de múltiples vectores por iteración

En este apartado se describe una implementación basada en la ejecución por lotes, tal y como se introducía en el apartado 3.3. Esta implementación hace uso de la función Device Segmented Radix Sort de la librería CUB, al tratarse de la alternativa más sencilla de implementar de forma general y atendiendo al rendimiento previamente obtenido con la misma.

Dentro de la aplicación para el cálculo de la anomalía climática se distinguen principalmente tres ejes: latitud, longitud y tiempo. En este contexto, el eje temporal define la dimensión de los vectores a ordenar, mientras que la latitud y la longitud identifican cada uno de dichos vectores. En la versión original, el código se estructura mediante dos bucles anidados, donde el bucle externo recorre la latitud y el interno la longitud, repartiéndose el total de iteraciones entre el conjunto de hilos en ejecución.

Para poder aplicar una ordenación por lotes, se ha llevado a cabo una reconfiguración de esta estructura. En lugar de mantener ambos bucles anidados, se conserva el bucle externo, mientras que el interno se transforma en una única iteración paralela entre los distintos hilos. De este modo, se realiza una única lectura y escritura de los datos, y el cómputo se ejecuta de manera simultánea en los distintos hilos. Concurrentemente y una vez finalizada la fase de depuración de los datos, la ordenación de la mitad de los vectores se delega a la GPU.

Esta estrategia permite, en principio, un mejor aprovechamiento de los recursos de la GPU, al tiempo que se reducen las latencias y los tiempos de espera asociados a la transferencia de datos. Sin embargo, tal y como se muestra en los resultados representados en la gráfica 3.24, esta implementación presenta un rendimiento significativamente inferior en comparación con las aproximaciones anteriores.

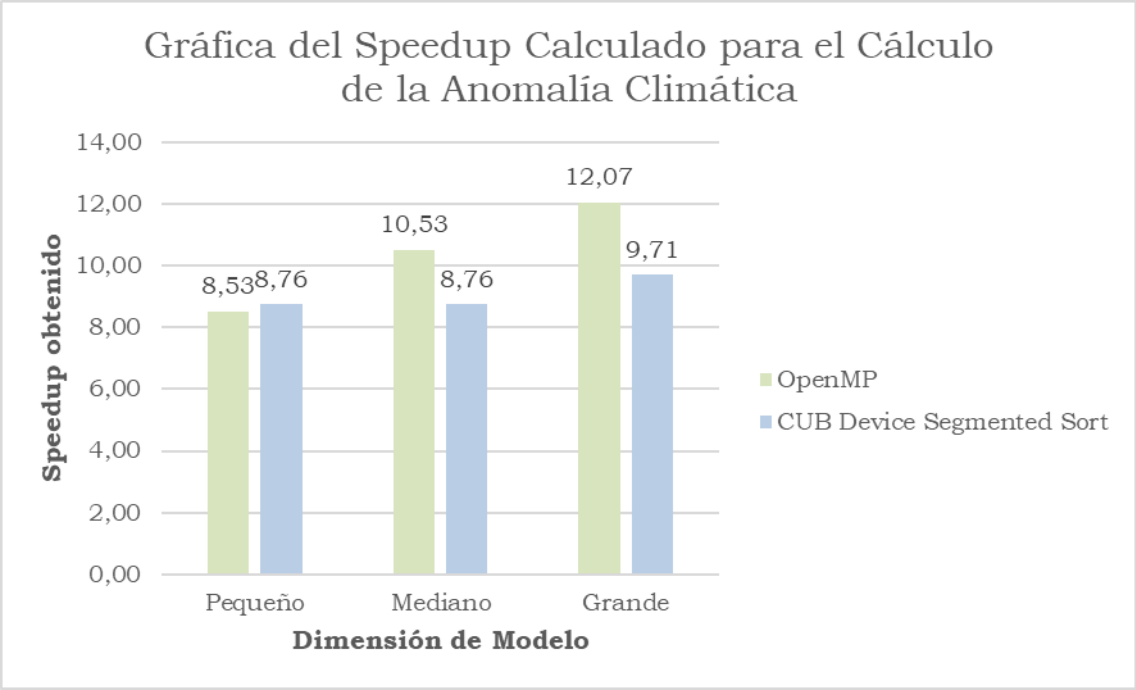

FIGURA 3.24 *GRÁFICO DE BARRAS QUE MUESTRA EL SPEEDUP OBTENIDO EN EL CÁLCULO DE LA ANOMALÍA CLIMÁTICA DADOS TRES MODELOS DISTINTOS, UNO PEQUEÑO, UNO MEDIANO Y UNO GRANDE. DONDE SE CALCULA PARA DOS IMPLEMENTACIONES DISTINTAS, UNA DE PARALELIZACIÓN SOLO EN CPU CON OPENMP Y UNA SEGUNDA IMPLEMENTACIÓN POR LOTES DONDE SE DELEGA LA ORDENACIÓN DE LA MITAD DE LOS VECTORES A LA GPU CON LA FUNCIÓN DEVICE SEGMENTED RADIX SORT DE LA LIBRERÍA CUB*

Este comportamiento se explica, en primer lugar, por los problemas ya mencionados anteriormente: la división de las ordenaciones de los vectores entre CPU y GPU puede llegar a ser, en este caso, al menos el doble de rápida cuando se realiza únicamente en CPU, mientras que la ejecución en GPU no resulta lo suficientemente eficiente para compensar dicha diferencia. A esto se añade una mayor necesidad de sincronización entre todos los hilos del sistema, ya que, frente a la ejecución independiente, en esta implementación es frecuente que algunos hilos deban esperar a la finalización del cómputo en GPU, incrementando así los tiempos de inactividad.

Adicionalmente, se repite el problema detectado en el apartado anterior: la estructura base de la aplicación dificulta la explotación efectiva de la paralelización entre CPU y GPU, debido al reducido número de operaciones que pueden ejecutarse de forma simultánea. Por este motivo, al igual que en el apartado previo, se ha desarrollado una segunda versión de esta implementación en la que se incrementa la paralelización entre ambos dispositivos, adelantando la lectura de los datos y posponiendo la escritura de los resultados, de modo que estas operaciones puedan solaparse con la ordenación en GPU.

Los resultados obtenidos de la segunda implementación son los que se muestran en la gráfica 3.25, donde se ve la diferencia entre los speedups calculados.

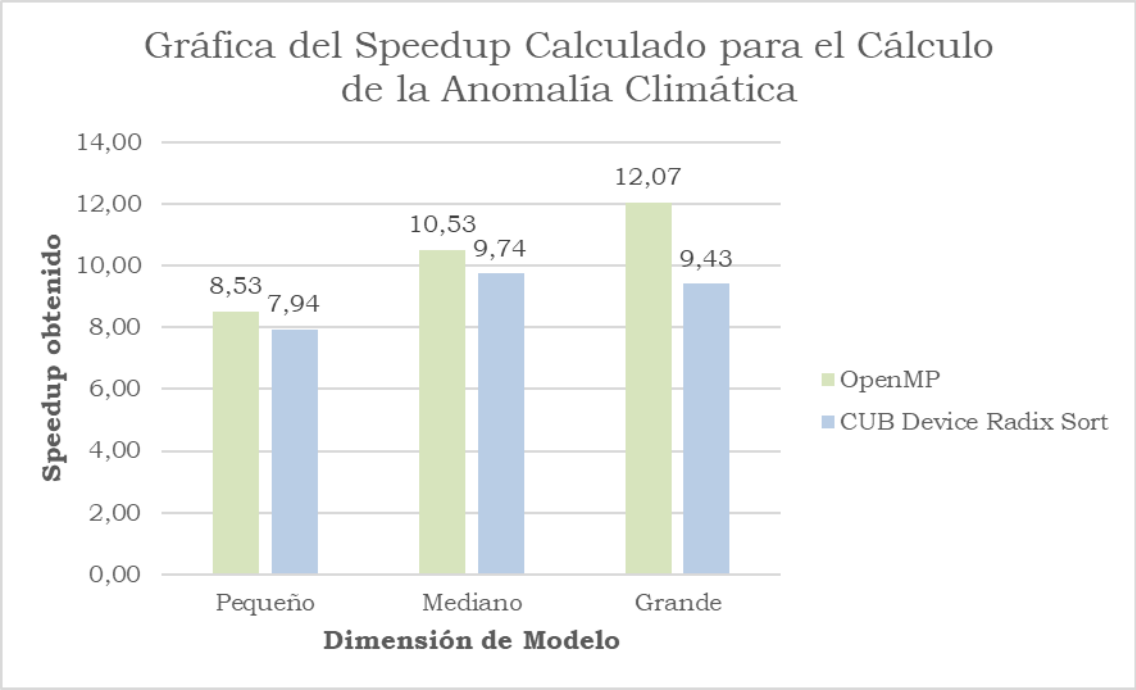

FIGURA 3.25 *GRÁFICO DE BARRAS QUE MUESTRA EL SPEEDUP OBTENIDO EN EL CÁLCULO DE LA ANOMALÍA CLIMÁTICA DADOS TRES MODELOS DISTINTOS, UNO PEQUEÑO, UNO MEDIANO Y UNO GRANDE. DONDE SE CALCULA PARA DOS IMPLEMENTACIONES DISTINTAS, UNA DE PARALELIZACIÓN SOLO EN CPU CON OPENMP Y UNA SEGUNDA IMPLEMENTACIÓN POR LOTES DONDE SE DELEGA LA ORDENACIÓN DE LA MITAD DE LOS VECTORES A LA GPU CON LA FUNCIÓN DEVICE SEGMENTED RADIX SORT DE LA LIBRERÍA CUB*

Como puede observarse en la gráfica 3.25, los resultados experimentales son muy parecidos a la implementación más simple. Ciertamente, el rendimiento obtenido sigue siendo claramente inferior al alcanzado por la implementación basada exclusivamente en CPU, diferencia que en este caso se vuelve aún más significativa a medida que aumenta la dimensión de los vectores.

Este comportamiento se debe principalmente al incremento en los tiempos de transferencia de datos, lo que provoca una mayor inactividad de los hilos que deben esperar a que dichas transferencias finalicen. Asimismo, apenas se aprecia una diferencia con respecto a la implementación original, ya que los tiempos asociados a la ordenación en GPU superan a los tiempos de ejecución de las operaciones realizadas por los hilos en CPU. En consecuencia, incluso aumentando el grado de paralelización entre CPU y GPU, no se consigue una mejora significativa en el rendimiento global.

4 Conclusiones y Trabajo Futuro

Una vez concluidas las pruebas y las distintas implementaciones sobre la ordenación de vectores de tamaño relativamente reducido (aproximadamente 27 000 elementos por vector) mediante el uso de la GPU, se han podido identificar una serie de comportamientos y conclusiones comunes.

En el caso de los algoritmos de ordenación, resulta complejo aprovechar el alto grado de paralelismo que ofrece la GPU, ya que se trata de algoritmos que requieren una elevada sincronización. Si se compara el speedup obtenido con el alcanzable mediante algoritmos trivialmente paralelizables, como puede ser la suma de vectores, se observa una diferencia significativa en el rendimiento alcanzado. A pesar del gran potencial de paralelización de la GPU, en este tipo de algoritmos la reducción del tiempo de ejecución es limitada, especialmente cuando se trabaja con conjuntos de datos relativamente pequeños, como en el caso estudiado.

Tomando como ejemplo la función Device Radix Sort de la librería CUB, estudiada en el apartado 3.2.1, se ha comprobado que el rendimiento óptimo no se alcanza hasta manejar volúmenes de datos del orden de 10^7 elementos. Incluso en ese escenario, el speedup obtenido se sitúa en torno a un factor seis, mientras que otros algoritmos con mayor grado de paralelización pueden alcanzar aceleraciones de cientos o incluso miles de veces respecto a la ejecución secuencial. En este contexto, los tiempos de transferencia de datos entre CPU y GPU reducen de forma significativa el beneficio potencial, hasta el punto de no compensar su uso en ejecuciones reales cuando se dispone de más de ocho hilos de CPU aproximadamente.

Aun así, uno de los principales puntos fuertes de la GPU es la posibilidad de utilizarla de forma concurrente junto con la CPU. En el caso práctico del cálculo de la anomalía climática, la CPU no se limita únicamente a realizar operaciones de ordenación, lo que permite que delegar dicha tarea a la GPU pueda traducirse en una mejora global del rendimiento. Sin embargo, como se ha observado en los experimentos realizados, este enfoque presenta ciertas limitaciones cuando se ejecutan múltiples llamadas simultáneas para la transferencia de datos, ya que dichas transferencias no son fácilmente paralelizables y generan sincronizaciones que reducen el tiempo de cómputo efectivo.

Aunque de forma aislada la ordenación en GPU puede llegar a ser aproximadamente el doble de rápida, con vectores de 27000 elementos, que su equivalente en CPU, cuando se ejecuta con varios hilos concurrentes, en este caso, a partir de seis hilos, el beneficio comienza a diluirse debido a los costes de sincronización. Como resultado, se alcanza un punto en el que una ejecución paralela basada únicamente en CPU puede llegar incluso a superar a la implementación en GPU.

Incluso en el caso de las implementaciones de ordenación por lotes, en las que se reducen los tiempos de latencia y de transferencia de datos, el rendimiento obtenido sigue viéndose ampliamente superado por el de la ejecución paralela en CPU, llegando a ser en algunos casos más del doble de rápido.

Como consecuencia, al integrar este tipo de funciones dentro del entorno final para el cálculo de la anomalía climática apenas se logra un rendimiento

comparable al alcanzado sin el uso de la GPU, e incluso puede resultar significativamente inferior en determinados escenarios. Esto se debe, principalmente, a la necesidad de sincronización entre los hilos concurrentes, que incrementa los tiempos de inactividad. Este efecto es especialmente notable en la implementación por lotes, donde la sincronización entre hilos es más frecuente y tiene un impacto negativo considerable sobre los tiempos de ejecución.

A modo de evaluación global, los resultados obtenidos con las funciones de ordenación estudiadas muestran que, incluso en el escenario más favorable, el speedup no supera aproximadamente un factor diez. No obstante, como se ha observado, en el caso práctico del cálculo de la anomalía climática esta mejora apenas llega a percibirse a nivel global.

Cabe destacar que incluso las implementaciones más simples basadas en GPU implican un incremento no trivial en la complejidad del código. Si bien este aumento suele estar justificado cuando va acompañado de mejoras sustanciales en el rendimiento, al no ser este el caso, resulta razonable considerar la opción de descartar el uso de la GPU para esta aplicación concreta.

Como se ha podido notar, debido a las características del problema concreto que se ha querido abordar, no se han podido obtener mejoras significativas con el uso de estas librerías. Sin embargo, el estudio realizado, la evaluación y comparación de los distintos métodos que se han realizado durante el trabajo, queda a disposición de quien quiera continuar un trabajo de búsqueda (o investigación) de otras posibles soluciones en este campo a partir de los resultados que se dejan por escrito en este documento.

Concluyendo así este trabajo, como un análisis del rendimiento de los algoritmos de ordenación en GPU de las principales librerías CUDA cuando se aplican al proceso práctico de la ordenación de vectores dentro de la aplicación para el cálculo de la anomalía climática.

4.1 Trabajo Futuro

Por limitaciones de tiempo, algunos aspectos de este proyecto han quedado pendientes de abordar. En particular, resultaría de interés estudiar e implementar los algoritmos considerados en un contexto más específico, fuera del uso directo de librerías estándar, mediante el desarrollo de implementaciones propias basadas en kernels optimizados para este problema concreto. De este modo, sería posible analizar en mayor detalle las diferencias de rendimiento entre las implementaciones de librerías ampliamente utilizadas en el ámbito científico y aquellas diseñadas y optimizadas específicamente para este proceso.

Durante el desarrollo del proyecto se diseñó, además, una versión preliminar basada en OpenACC. No obstante, no se llegó a profundizar en la evaluación de sus resultados debido a las dificultades encontradas para obtener un rendimiento comparable con el de las versiones “nativas”. En consecuencia, se consideró prioritario continuar profundizando en el desarrollo y la optimización de estas últimas. Por tanto, otra posible línea de trabajo futuro consiste en el diseño, desarrollo y evaluación exhaustiva de una versión basada en OpenACC específicamente optimizada para este propósito.

5 Análisis del Impacto

Tras la finalización de este proyecto se han obtenido resultados que contribuyen a la optimización del cálculo de la anomalía climática. Si bien dichos resultados no han cumplido plenamente con todos los objetivos propuestos, sí han permitido cerrar o acotar líneas de posible optimización que habían quedado abiertas tras la finalización del trabajo de fin de carrera de Esteban Aspe Ruiz, titulado “Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática”.

Estas aportaciones resultan entonces relevantes en el ámbito del cálculo de la anomalía climática, el cual mantiene una relación directa con la predicción de incendios forestales a través del IPP (Índice de Propagación Potencial). En este contexto, y en consonancia con los Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030 [31], el proyecto contribuye especialmente al ODS 13 (Acción por el clima) y al ODS 15 (Vida de ecosistemas terrestres), al favorecer la mejora de los sistemas de prevención, mitigación y gestión del riesgo asociado a los incendios forestales.

Desde un punto de vista estrictamente académico, la realización de este proyecto ha supuesto un incremento significativo en la profundidad de los conocimientos adquiridos sobre CUDA y el funcionamiento de las GPU. Dicho aprendizaje no se ha limitado únicamente a la ordenación de vectores, sino que ha abarcado los fundamentos y principios generales de la computación en GPU. Tratándose, además, de un entorno tecnológico con una elevada demanda en la actualidad, por lo que la adquisición de estas competencias resulta especialmente relevante.

Por último, desde una perspectiva personal, este proyecto ha contribuido de manera notable al desarrollo de mi madurez personal. La realización del mismo ha supuesto un desafío exigente, que ha requerido un alto nivel de dedicación y esfuerzo continuo, permitiendo así el desarrollo de capacidades como la constancia, la autonomía y la superación personal.

6 Bibliografía

[1] M. Flinders, S. Susnjara and I. Smalley, "What is a graphics processing unit (GPU)?," [Online]. Available: https://www.ibm.com/think/topics/gpu.

[2] NVIDIA, "What is CUDA?," [Online]. Available: https://nvidia.custhelp.com/app/answers/detail/a_id/2132/~/what-is-cuda.

[3] NVIDIA, «CUDA Toolkit,» [En línea]. Available: https://developer.nvidia.com/cuda-toolkit.

[4] NVIDIA, «About Us NVIDIA,» [En línea]. Available: https://www.nvidia.com/en-us/about-nvidia.

[5] OpenACC, «About OpenACC,» [En línea]. Available: https://www.openacc.org/about.

[6] OpenACC, «OpenACC Specification,» [En línea]. Available: https://www.openacc.org/specification.

[7] Consejería de Desarrollo Sostenible Castilla-La Mancha, «Índice de Propagación Potencial de incendios forestales,» [En línea]. Available: https://infocam.castillalamancha.es/indice-de-propagacion-potencial.

[8] E. Aspe Ruiz, «Implementación y evaluación de un algoritmo de altas prestaciones para el cálculo de la anomalía climática,» Boadilla del Monte, 2024.

[9] Cornell University, «What Is OpenMP?,» [En línea]. Available: https://cvw.cac.cornell.edu/openmp/intro/what-is-openmp.

[10] OpenMP, «OpenMP Specifications,» [En línea]. Available: https://www.openmp.org/specifications/.

[11] F. C. Zamora, «8º Congreso Forestal Español (2022),» [En línea]. Available: https://8cfe.congresoforestal.es/es/content/el-indice-de-propagacion-potencial-ipp-de-castilla-la-mancha-herramienta-para-la-prediccion.

[12] «CUDA Core Compute Libraries - Thrust,» [En línea]. Available: https://nvidia.github.io/cccl/thrust/.

[13] T. Talaei Khoei, H. Ould Slimane y N. Kaabouch, «Deep learning: systematic review, models, challenges, and research directions,» [En línea]. Available: https://link.springer.com/article/10.1007/s00521-023-08957-4.

[14] K. Hou, W. Liu, H. Wang y W.-c. Feng, «Fast Segmented Sort on GPUs,» [En línea]. Available: https://www.ssslab.cn/assets/papers/2017-hou-fastsegsort.pdf.

[15] NVIDIA, «CUDA Toolkit Documentation CUB,» [En línea]. Available: https://docs.nvidia.com/cuda/archive/12.9.0/cub/index.htm.

[16] S. Baxter, «moderngpu 2.0,» [En línea]. Available: https://github.com/moderngpu/moderngpu/wiki.

[17] M. Xu, X. Xu, F. Zheng, Y. Yang and M. Yin, "A Hybrid Sorting Algorithm on Heterogeneous Architectures," [Online]. Available: https://www.proquest.com/docview/1805463805.

[18] M. Gul Awan y F. Saeed, «GPU-ArraySort: A parallel, in-place algorithm for sorting large number of arrays,» [En línea]. Available: https://scholarworks.wmich.edu/pcds_reports/5.

[19] B. B. Kosmalski, «tfg_ordenacion_vectores_gpu,» [En línea]. Available: https://github.com/BrunoBKos/tfg_ordenacion_vectores_gpu.

[20] NVIDIA, «CUDA C++ Programming Guide,» [En línea]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#asynchronous-concurrent-execution.

[21] NVIDIA, «NVIDIA CUB API Reference cub::DeviceRadixSort,» [En línea]. Available: https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceRadixSort.html.

[22] Mentores Tech, «Radix Sort,» [En línea]. Available: https://www.mentorestech.com/resource-algorithms-radix-sort.php.

[23] geeksforgeeks, «Quick Sort,» [En línea]. Available: https://www.geeksforgeeks.org/dsa/quick-sort-algorithm/.

[24] geeksforgeeks, «Merge Sort,» [En línea]. Available: https://www.geeksforgeeks.org/dsa/merge-sort/.

[25] NVIDIA, «CUDA Core Compute Libraries - Block Radix Sort,» [En línea]. Available: https://nvidia.github.io/cccl/cub/api/classcub_1_1BlockRadixSort.html.

[26] NVIDIA, «CUDA Compute Core Libraries - Device Merge Sort,» [En línea]. Available: https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceMergeSort.html.

[27] NVIDIA, «CUDA Compute Core Libraries - Device Segmented Sort,» [En línea]. Available: https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceSegmentedSort.html.

[28] D. Spuler, «CUDA Thread Divergence,» [En línea]. Available: https://www.aussieai.com/blog/cuda-thread-divergence.

[29] «moderngpu,» [En línea]. Available: https://github.com/moderngpu/moderngpu/tree/master.

[30] K. a. L. W. a. W. H. a. F. W.-c. Hou, «bb_segsort GitHub,» [En línea]. Available: https://github.com/vtsynergy/bb_segsort/tree/master.

[31] Naciones Unidas, «Objetivos del Desarrollo Sortenible,» [En línea]. Available: https://www.un.org/sustainabledevelopment/es/.