Universidad Austral de Chile

Conocimiento y Naturaleza

Sorting Paralelo

INFO188 - PROGRAMACIÓN EN PARADIGMAS FUNCIONAL Y PARALELO

Autores:
Renato Atencio
Handel Venegas
Orlando Contreras
Fabrizio Fresard

Introducción al problema

El ordenamiento es una operación fundamental en muchas aplicaciones informáticas, desde bases de datos hasta simulaciones científicas. En sistemas modernos, la computación paralela ha permitido reducir significativamente los tiempos de ejecución al aprovechar múltiples núcleos de CPU y la masiva paralelización de las GPU. En este informe, se comparan algoritmos de ordenamiento paralelos en CPU y GPU para determinar cuál es el más eficiente en cada plataforma.

Ordenamiento en CPU

Los algoritmos paralelos en **CPU** están diseñados para aprovechar la capacidad de múltiples núcleos, dividiendo el trabajo en subprocesos independientes. Entre los más comunes se encuentran:

- **MergeSort Paralelo:** Este algoritmo es altamente eficiente en CPUs debido a su enfoque de divide y vencerás. Al paralelizar las operaciones de partición y fusión, se logran mejoras significativas en el rendimiento. En implementaciones paralelas utilizando OpenMP, MergeSort puede reducir considerablemente el tiempo de ejecución en arreglos grandes a medida que se incrementa el número de hilos. Sin embargo, el rendimiento final depende del tamaño del arreglo y de la capacidad de paralelización del sistema, ya que en arreglos pequeños o con bajo paralelismo disponible, las mejoras pueden ser limitadas [1][2].
- **QuickSort Paralelo:** Aunque QuickSort es conocido por su rapidez, su paralelización es más compleja. Las divisiones desiguales de los sub-arreglos y la necesidad de sincronización entre hilos afectan negativamente su eficiencia. Algunos estudios sugieren que, aunque QuickSort puede ser competitivo en ciertos casos, MergeSort lo supera en arreglos grandes y sistemas con muchos núcleos debido a su mejor comportamiento de paralelización en estos entornos [3][4].

Ordenamiento en GPU

Las **GPUs**, con miles de núcleos disponibles, permiten un paralelismo masivo que supera ampliamente a las CPUs en ciertas tareas, como el ordenamiento. Entre los algoritmos más utilizados están:

- **RadixSort en GPU:** Este algoritmo es altamente eficiente en GPUs debido a su estructura simple pero paralelizable. Los hilos procesan simultáneamente bits de los enteros, logrando un rendimiento excepcional en arreglos grandes. Según

algunos estudios, RadixSort en GPU puede ser hasta 10 veces más rápido que los algoritmos tradicionales de CPU para volúmenes masivos de datos [5].

- **BitonicSort en GPU:** Aunque menos eficiente en general, BitonicSort es útil para casos específicos, como cuando el número de elementos es una potencia de 2. Su estructura regular permite paralelizar fácilmente, lo que lo hace adecuado para arquitecturas de GPU que priorizan la comunicación predecible entre hilos [6].

Elección del Mejor Algoritmo

- **Para CPU:** Se selecciona **MergeSort Paralelo** debido a su capacidad para dividir eficientemente el trabajo entre núcleos y su rendimiento demostrado en estudios previos [1][2].
- **Para GPU:** Se elige **RadixSort** como el algoritmo más adecuado, dado que maximiza el paralelismo y aprovecha al máximo la arquitectura de las GPUs [5].

Metodología

Implementación de los algoritmos:

Merge Sort en CPU (Paralelo): El algoritmo se implementa de forma paralela utilizando OpenMP para dividir el trabajo entre múltiples hilos. En el Merge Sort paralelo en CPU, la división del arreglo se realiza de forma recursiva, donde el arreglo se divide en dos mitades hasta llegar a subarreglos de un solo elemento. Esta división no se paraleliza. Sin embargo, durante la fase de mezcla, el algoritmo se paraleliza utilizando OpenMP, que distribuye el trabajo entre varios hilos. OpenMP maneja la asignación de tareas, permitiendo que diferentes hilos procesen distintas secciones del arreglo simultáneamente. Esto mejora la eficiencia al fusionar los subarreglos, ya que varios hilos trabajan en paralelo para combinar los resultados.

Radix Sort en GPU: Se implementa usando CUDA, donde el algoritmo realiza la máscara de bits, el escaneo exclusivo (scan) y el reordenamiento en paralelo. En el algoritmo de Radix Sort implementado en CUDA, el proceso se divide en varias fases paralelizadas, utilizando kernels específicos para cada tarea. La estructura de los kernels en el código es la siguiente:

> **Máscara de Bits:** Se crea un kernel que aplica una máscara de bits a los elementos del arreglo, aislando las posiciones de los dígitos relevantes para cada paso del algoritmo (basado en los bits de cada número). Este paso es fundamental para identificar las posiciones de los dígitos que se ordenarán.
>
> **Escaneo Exclusivo (Scan)**: Otro kernel realiza el escaneo exclusivo (scan), que es una operación acumulativa en la que se calcula, para cada elemento, el número de elementos menores o iguales a dicho elemento en el arreglo. Esto permite clasificar los elementos de manera eficiente. Se utiliza una técnica de paralelización para acelerar este proceso, de modo que se realice en bloques y luego se combine el resultado.
>
> **Reordenamiento**: Un tercer kernel maneja el reordenamiento de los elementos. Utilizando los resultados del escaneo, los elementos del arreglo se colocan en las posiciones correctas, asegurando que los elementos sean ordenados según el dígito o bit que se está procesando en cada iteración del algoritmo.
>
> Cada uno de estos kernels se ejecuta en paralelo, distribuyendo el trabajo entre los hilos de la GPU para aprovechar al máximo la capacidad de procesamiento en paralelo, lo que mejora significativamente el rendimiento en comparación con

una implementación secuencial tradicional. El uso de CUDA permite que estas operaciones se realicen de manera eficiente en grandes volúmenes de datos.

std::sort en CPU: Se utiliza la función estándar de ordenación de C++ como un punto de comparación de rendimiento Esta librería se usa exclusivamente para un benchmark..

Generación de Datos: Los arreglos de entrada se inicializan con números aleatorios utilizando la función **mt19937** de la librería estándar C++, junto con la distribución uniforme **uniform_int_distribution<int>**. Este generador de números pseudo-aleatorios se inicializa con una semilla basada en el tiempo actual **(time(0))**, lo que permite generar una secuencia de números aleatorios en el rango de 0 a 999999. Esta estrategia busca simular diversas situaciones de datos para probar los algoritmos de ordenamiento en distintas condiciones.

Configuración del experimento:

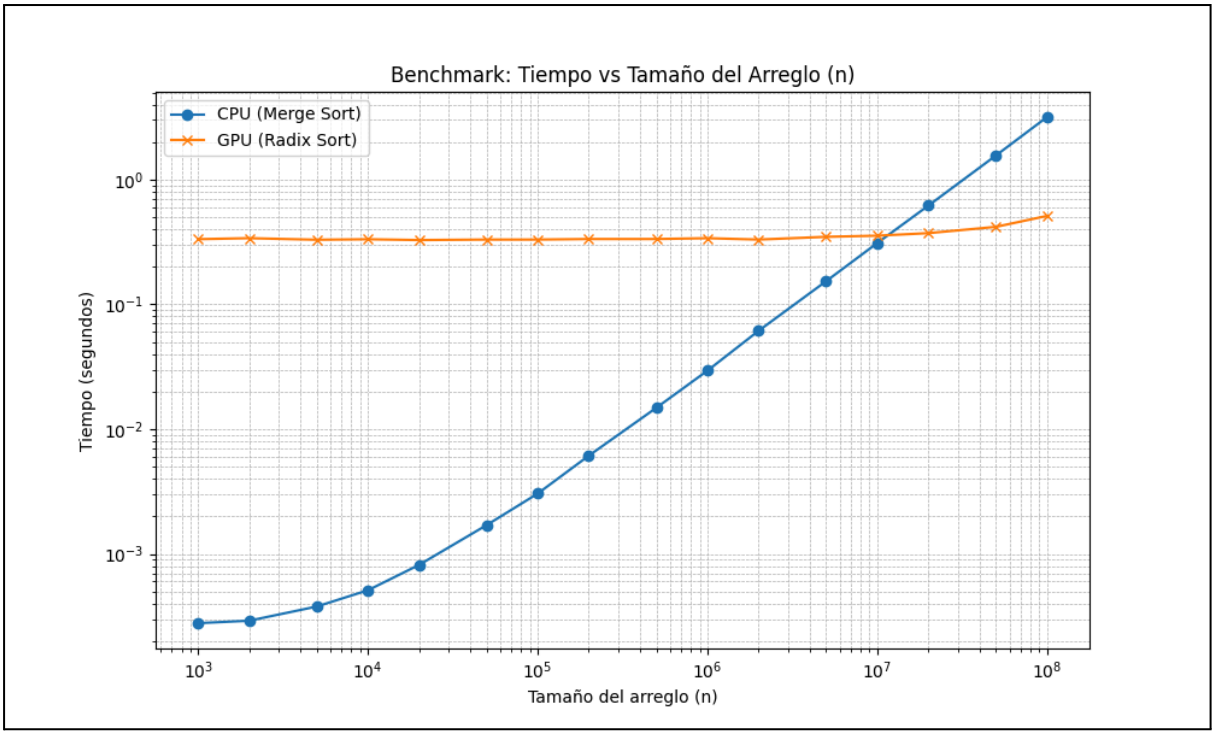
Parámetros de entrada: Se evalúan diferentes tamaños de arreglo **n** y números de hilos (para el Merge Sort en CPU) o gridsize (para el Radix Sort en GPU) en las implementaciones paralelas. El número de hilos o bloques se pasa como parámetro a través de la línea de comandos, permitiendo ajustar la configuración para cada ejecución.

Métricas: El tiempo de ejecución de cada algoritmo (Merge Sort en CPU, Radix Sort en GPU, y std::sort) se mide utilizando omp_get_wtime() para la CPU y se captura mediante temporizadores en ambos modos. Los resultados se almacenan en un archivo CSV, incluyendo el tamaño del arreglo, el modo de ejecución (CPU o GPU), el número de hilos/bloques, y el tiempo de ejecución correspondiente. Esto facilita el análisis comparativo del rendimiento entre las implementaciones.

Resultados y Análisis

En esta sección se presentan y analizan los resultados obtenidos de un total de **717** experimentos realizados para evaluar el rendimiento de tres algoritmos de ordenamiento: Merge Sort paralelo en CPU, Radix Sort en GPU y std::sort de la STL. Los experimentos se llevaron a cabo variando el tamaño del arreglo n, el número de hilos en CPU y el número de bloques en GPU, con el objetivo de medir y comparar el rendimiento en distintas configuraciones. A través de una serie de gráficos, se analiza cómo los algoritmos se comportan en función de estos parámetros y se discuten las diferencias en términos de tiempo de ejecución, escalabilidad y eficiencia, tanto para la CPU como para la GPU. A continuación se presentan los gráficos y sus respectivos análisis:

Gráfico 1: Tiempo vs Tamaño del Array (n)



Análisis datos:

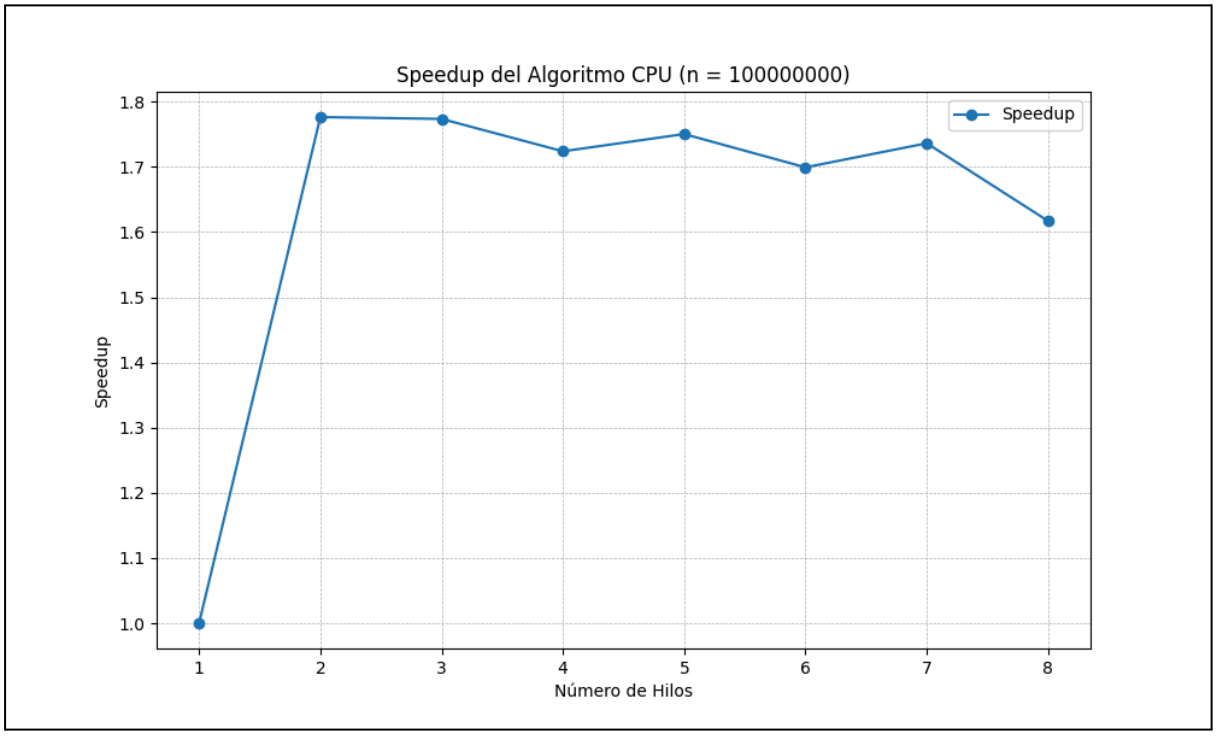
- **Tiempos CPU**:
  - Para n pequeños (1000, 2000, etc.), los tiempos de ejecución son bajos y consistentes.
  - A medida que n crece, los tiempos aumentan de manera progresiva, especialmente para n mayores a 1,000,000, alcanzando hasta más de 3 segundos para n = 100,000,000.
  - A partir de 20.000.000 de elementos, los tiempos sobrepasan el segundo,, con máximos de hasta 3.8 segundos para n = 100,000,000.
- **Tiempos GPU**:
  - Los tiempos iniciales para n pequeños (1000, 2000) son mucho mayores que los de la CPU (aproximadamente 0.3-0.4 segundos).
  - Sin embargo, a medida que **n** aumenta, la GPU muestra una ventaja clara, con tiempos mucho menores que los de la CPU, ya que si bien también pasados los 10.000.000, lo hace de manera más lenta y estable, en comparación a la CPU que crece de más rápido..
  - Para n relativamente “grandes” (10.000.000), los tiempos se estabilizan entre 0.3 y 0.5 segundos, mucho más eficientes que la CPU.
- **Comparación**:
  - **CPU**: Es más rápida en arreglos pequeños pero se vuelve menos eficiente conforme aumentan.

    - **GPU**: Es menos eficiente en arreglos pequeños, pero supera a la CPU en arreglos grandes debido a su capacidad de paralelización.
- **Tendencia General**:
    - **CPU**: Adecuada para n pequeños (hasta 10,000,000).
    - **GPU**: A partir de **n** de aproximadamente 50,000,000, la GPU es mucho más rápida que la CPU, mostrando su capacidad para manejar grandes volúmenes de datos eficientemente.
- **Impacto de los valores constantes**:
    - **GridSize** y **Threads** constantes (256 y 8 respectivamente) no parecen afectar la tendencia observada, ya que los tiempos siguen la misma tendencia de aumento en función del tamaño de n.

Conclusión

Para este gráfico se realizaron 320 experimentos en total (120 tanto para la GPU como la CPU). Se puede concluir que la CPU es preferible para pequeñas cantidades de datos, mientras que la GPU sobresale cuando los arreglos de datos son grandes, gracias a su mayor capacidad de paralelización.

Gráfico 2: Speedup vs Número de Threads (CPU)



Análisis de los datos:

- **Datos de entrada**:
    - Tamaño del arreglo (n=100,000,000) es fijo.
    - Se incluyeron exclusivamente datos para el modo CPU (mode=0).

- Se mide el tiempo de ejecución para diferentes números de hilos (desde threads = 1 hasta threads = 8).

- **Observaciones iniciales**:
  - Para 1 hilo, el tiempo promedio de ejecución es alrededor de 5.12 segundos.
  - A medida que se incrementan los hilos, los tiempos disminuyen inicialmente, alcanzando un mínimo cercano a 2.83 segundos para 2 o 3 hilos, y luego comienzan a oscilar para valores mayores de threads.
  - El speedup se calcula respecto al tiempo promedio con 1 hilo.

Cálculo del Speedup

El speedup se define como: Tiempo con un Hilo/Tiempo promedio con x Hilos, de los datos que se generaron en el csv y posteriormente se graficaron, se obtuvo lo siguiente:

Hilos	Tiempo Promedio (s)	Speedup
1	5.12	1.00
2	2.88	1.78
3	2.89	1.77
4	2.95	1.73
5	2.93	1.75
6	3.02	1.69
7	2.95	1.74
8	3.05	1.68

Observaciones del Speedup

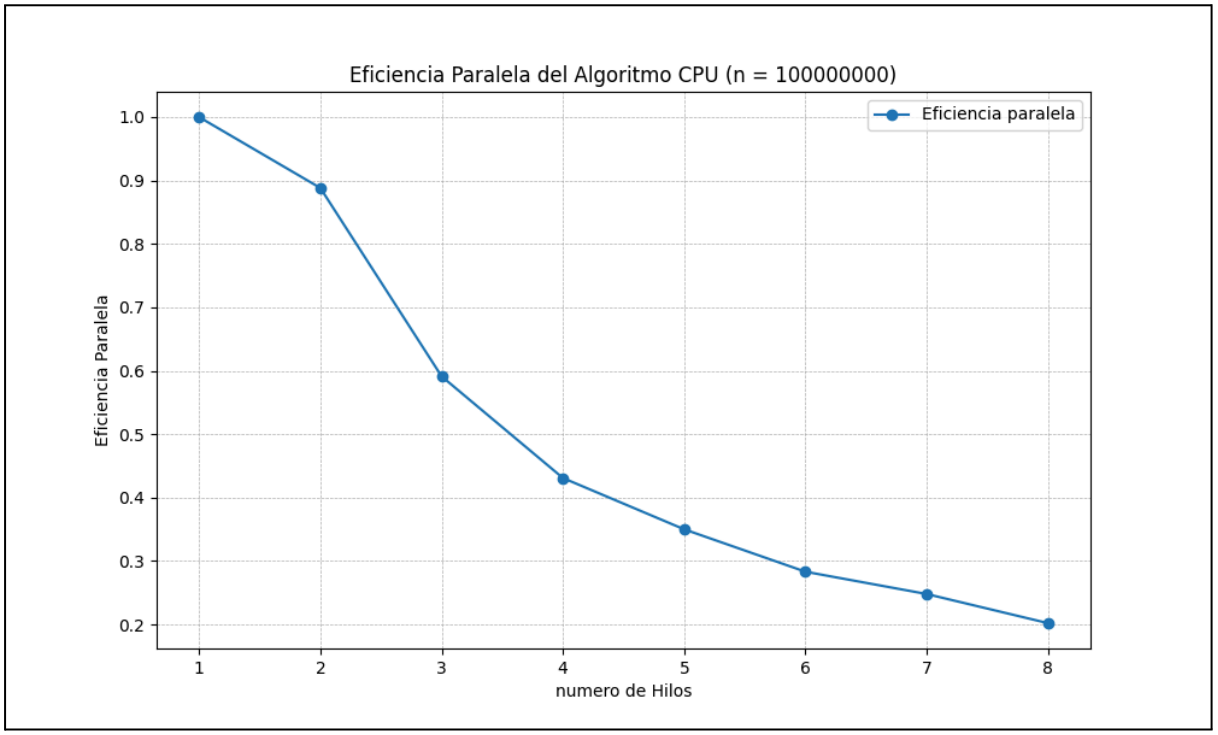
- **Comportamiento inicial**:
  - El speedup crece al usar más hilos, alcanzando un valor cercano a 1.78 para threads=2.
  - A partir de threads=2, el speedup se estabiliza y decrece ligeramente para más hilos.
- **Punto de saturación**:
  - Más allá de threads=2, el tiempo de ejecución no mejora significativamente.
  - El incremento en hilos introduce sobrecarga en la gestión de los mismos, afectando el rendimiento.

Conclusión:

Para obtener los datos de este gráfico, se hizo un total de 95 experimentos , de los cuales se puede concluir lo siguiente:

- **Limitación en la escalabilidad**:
    - El rendimiento no escala linealmente con el número de hilos debido a:
        - Costos de sincronización y sobrecarga de hilos.
        - Limitaciones en la arquitectura de la CPU o el algoritmo.
    - El speedup máximo observado es 1.78, lo cual sugiere que la eficiencia de paralelización es limitada.
- **Máxima eficiencia alcanzada con 2-3 hilo**s:
    - Más hilos no necesariamente mejoran el rendimiento y pueden incluso empeorar levemente debido a sobrecarga.

Gráfico 3: Eficiencia Paralela vs Número de Threads (CPU)



Análisis de los datos:

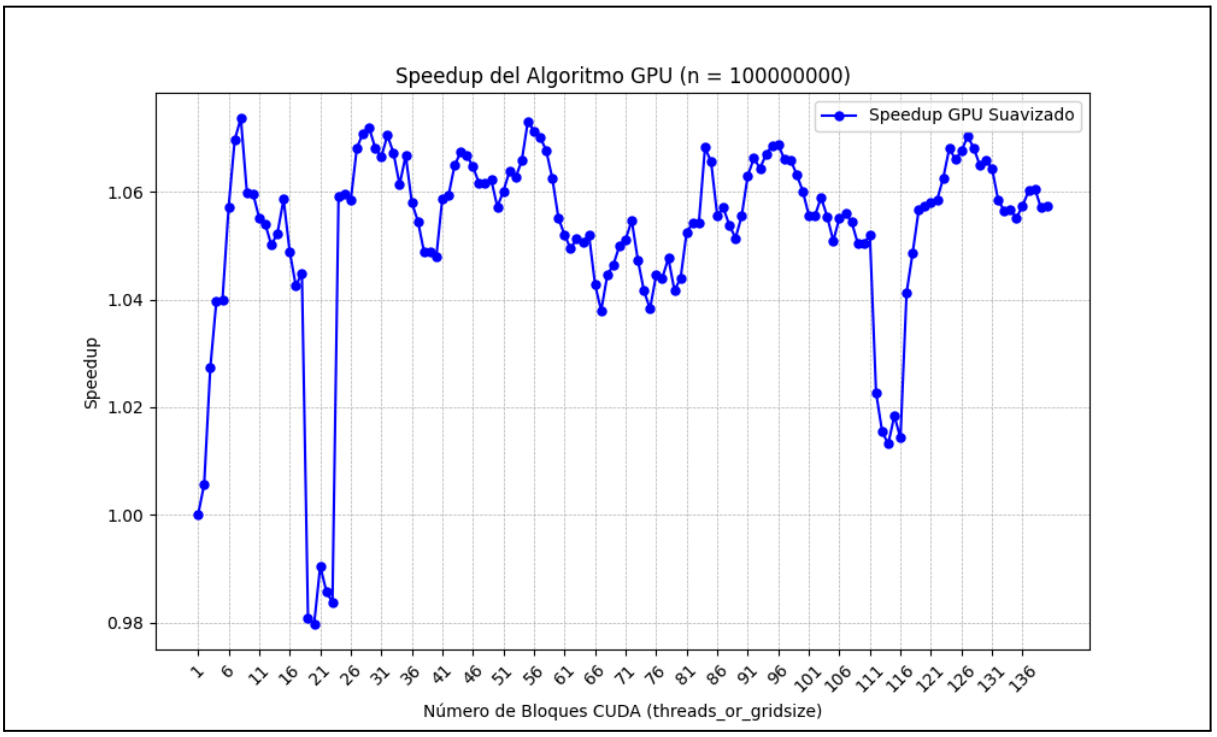
- **Tiempo base (1 hilo)**:
    - Tiempo promedio: 5.12 segundos.
    - Este valor se usa como referencia para calcular speedup y eficiencia.
- **Tendencia de tiempo**s:
    - Los tiempos disminuyen al aumentar los hilos hasta 6, pero los beneficios marginales disminuyen significativamente después de 6 hilos.
- **Eficiencia paralela**:

    - Alta eficiencia inicial (80-90%) con 2 a 4 hilos.
    - La eficiencia cae drásticamente (por debajo de 30%) con 6 o más hilos, indicando overheads y saturación.
- **Punto de saturación**:
    - Entre 6 y 8 hilos, los tiempos se estabilizan, mostrando que agregar más hilos no aporta mejoras significativas.
- **Causas de la pérdida de eficiencia**:
    - Overheads de sincronización y comunicación entre hilos.
    - Tamaño del problema insuficiente para amortiguar los costos adicionales de paralelización.
- **Observaciones**:
    - Usar entre 4 y 6 hilos para este tamaño de problema (n=100,000,000).
    - Para usar más hilos eficientemente, aumentar el tamaño del problema o mejorar la estrategia de paralelización.

Conclusión:

Para este experimento se usaron los datos generados del benchmark anterior. El algoritmo muestra una buena escalabilidad inicial, pero su eficiencia decrece significativamente con más hilos, debido a limitaciones típicas de paralelismo en CPU multicore.

Gráfico 4: Speedup vs Número de Bloques (GPU).



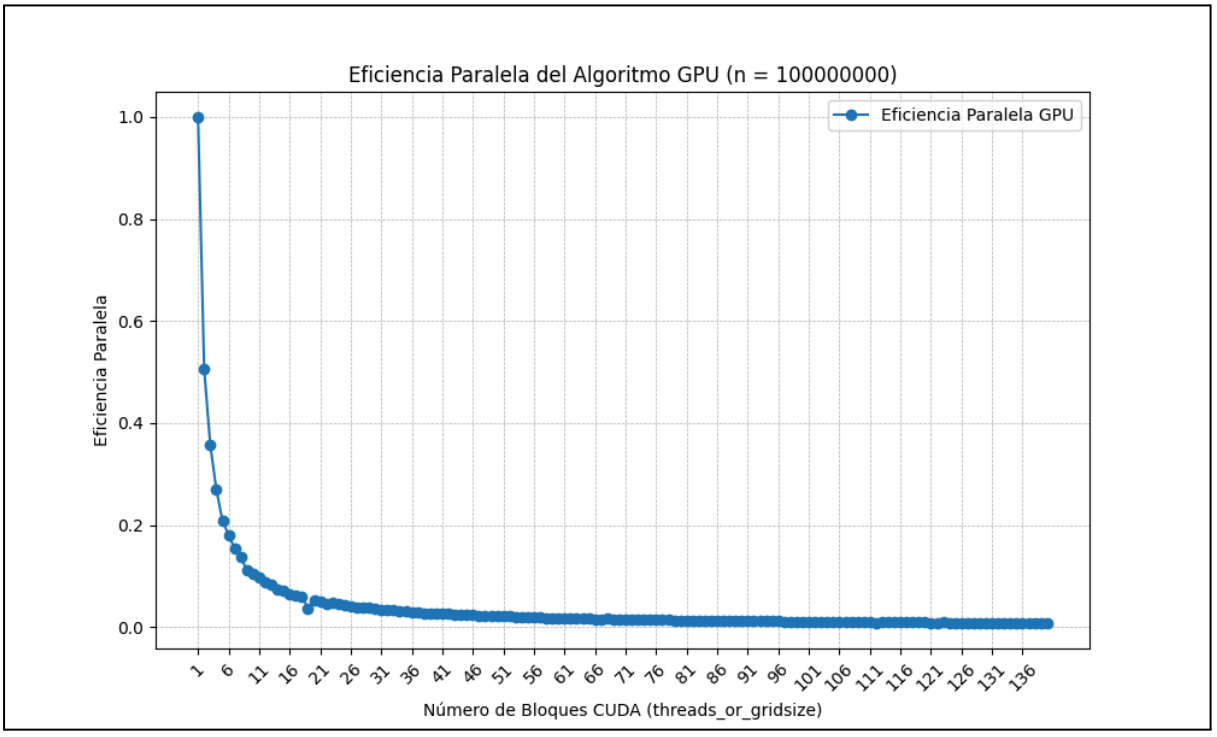
Análisis de los datos:

- **Crecimiento Inicial:** El *speedup* mejora rápidamente al aumentar los bloques hasta 20, debido a una mayor distribución del trabajo entre los núcleos CUDA.
- **Caídas bruscas:** Hay caídas en el *speedup* (por ejemplo, en gridsizes 19-20 y 113), probablemente por problemas de sincronización, saturación o acceso a memoria.
- **Estabilización:** A partir de 60 bloques, el *speedup* se estabiliza alrededor de 1.06, indicando que la GPU ha alcanzado su límite de paralelismo efectivo.
- **Variación de eficiencia:** Oscilaciones: Después del crecimiento inicial, el *speedup* oscila alrededor de un valor cercano a 1.06. Esto indica que la GPU alcanza un límite en su capacidad de paralelismo. Incrementar el número de bloques más allá de este punto no genera mejoras significativas, y las pequeñas variaciones pueden atribuirse a fluctuaciones en la asignación de recursos

Conclusiones:

- **Beneficio Inicia**l: Incrementar los bloques mejora el speedup hasta que los recursos de la GPU estén completamente utilizados (de 40 a 60 bloques).
- **Límite de Escalabilidad**: Más bloques no mejoran significativamente el rendimiento tras la saturación.
- **Anomalías**: Las caídas en el speedup deben investigarse, ya que pueden ser optimizables en la implementación o ajustes del kernel.
- **Rango Óptimo**: Entre 40 y 60 bloques se logra un balance ideal de rendimiento y uso de recursos.

Gráfico 5: Eficiencia Paralela vs Número de Bloques (GPU)

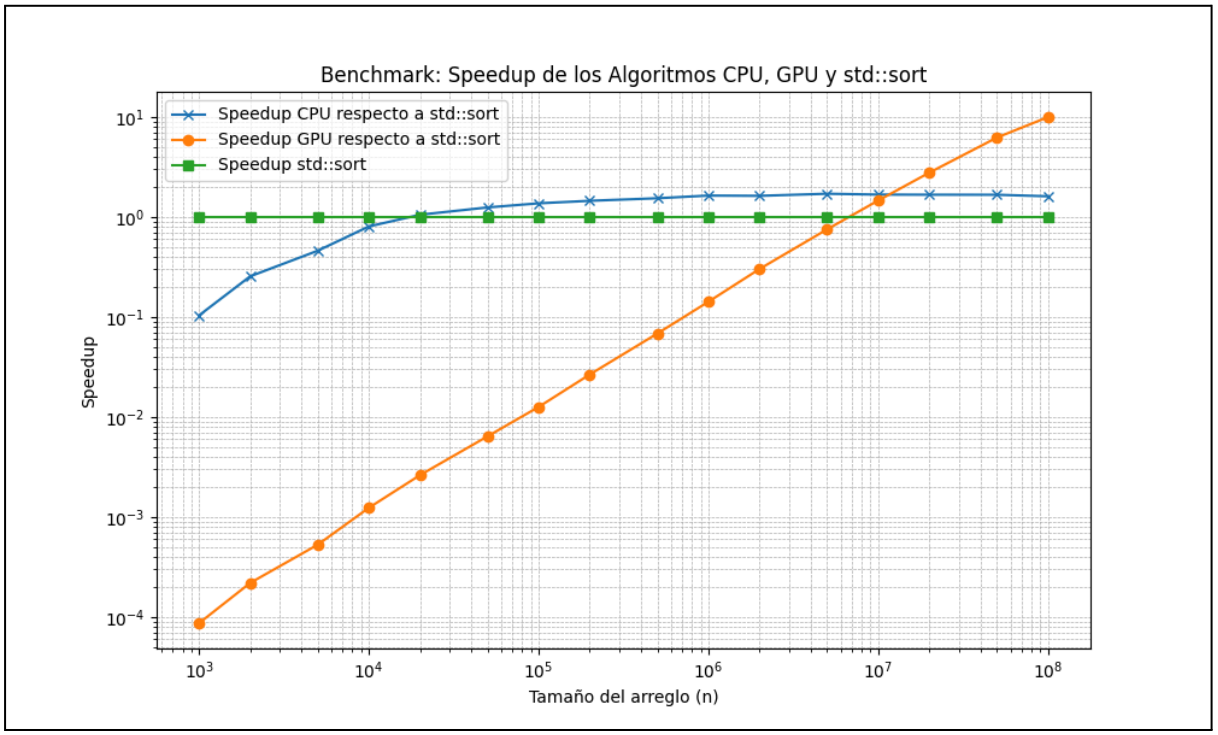


- **Distribución Asimétrica**:
  Las caídas abruptas o pequeñas fluctuaciones en la eficiencia también reflejan problemas de escalabilidad, como contención de recursos o asignación desigual del trabajo entre los bloques.

Conclusiones:

- **Limitaciones de Escalabilidad:**
  La eficiencia paralela disminuye drásticamente al incrementar los bloques porque la sobrecarga del paralelismo supera las ganancias de velocidad.
- **Rango Óptimo de Bloques:**
  Un rango bajo de bloques (menor a 10) es más eficiente, mientras que utilizar más bloques solo introduce desperdicio de recursos.
- **Sobrecarga del Sistema**:
  A partir de cierto punto, la cantidad de bloques no mejora el rendimiento y únicamente incrementa la sobrecarga de coordinación y acceso a memoria.
- **Recomendación:**
  Optimizar el número de bloques es clave para maximizar tanto el *speedup* como la eficiencia, manteniéndose dentro de un rango donde la GPU no se sature ni se desperdicie capacidad.

Gráfico 6: Speedup vs tamaño del array(n) comparado con std::sort



Análisis del gráfico:

- **CPU (Merge Sort Paralelo)**:
    - Inicialmente tiene un speedup bajo (menor a 1) en comparación con std::sort.
    - A medida que el tamaño del arreglo crece, se acerca e incluso supera ligeramente el rendimiento de std::sort en arreglos grandes.
- **GPU (Radix Sort)**:
    - Es significativamente más lento que std::sort para tamaños pequeños (speedup muy bajo).
    - Exhibe un aumento exponencial en el speedup, aunque todavía está por debajo de std::sort en este rango de tamaños.
- **std::sort**:
    - Sirve como referencia constante, con un speedup de 1 en todos los tamaños.

Conclusiones:

- **CPU**: Merge Sort paralelo en CPU es competitivo con std::sort para tamaños grandes, pero no aprovecha al máximo arreglos pequeños debido a la sobrecarga de paralelismo.
- **GPU**: Radix Sort en GPU muestra su potencial para tamaños muy grandes, pero su inicialización y transferencia de datos lo penalizan significativamente en arreglos pequeños y medianos.
- **std::sort**: Sigue siendo el método más equilibrado y eficiente en términos generales, especialmente para tamaños pequeños a medianos.

Conclusión General sobre Algoritmos de Ordenamiento Paralelo en CPU y GPU

En este informe comparativo de algoritmos de ordenamiento paralelo se pueden concluir características distintivas de dos enfoques fundamentales:

En CPU (Merge Sort Paralelo):

- **Características principales**: Algoritmo de divide y vencerás, implementado con OpenMP
- **Fortalezas**:
    - Eficiente para arreglos pequeños y medianos
    - Buena distribución del trabajo entre núcleos
    - Escalabilidad inicial alta
- **Limitaciones**:
    - Speedup máximo limitado a 1.78
    - Pérdida de eficiencia con más de 3-4 hilos
    - Overhead de sincronización reduce el rendimiento

En GPU (Radix Sort):

- **Características principales**: Implementación en CUDA con kernels especializados
- **Fortalezas**:
    - Rendimiento excepcional en arreglos muy grandes (>50 millones de elementos)
    - Procesamiento masivamente paralelo
    - Capacidad de manejar grandes volúmenes de datos eficientemente
- **Limitaciones**:
    - Rendimiento bajo en arreglos pequeños
    - Sobrecarga inicial de transferencia y configuración
    - Eficiencia decreciente al aumentar bloques de procesamiento

Comparación Global:

- Tamaño de Datos:
    - Hasta 10 millones: CPU (Merge Sort) más eficiente
    - Sobre 50 millones: GPU (Radix Sort) significativamente superior
    - Rango intermedio: std::sort mantiene un rendimiento equilibrado
- Escalabilidad:
    - CPU: Mejora inicial con 2-3 hilos, luego estancamiento
    - GPU: Beneficios hasta 40-60 bloques, después saturación

Conclusiones Principales:

- No existe un algoritmo “globalmente” óptimo
- La elección depende del contexto y requerimientos:
    - Volumen de datos
    - Arquitectura del sistema
    - Requisitos específicos de la aplicación

Observaciones y conclusión final:

- Considerar Merge Sort en CPU para datos pequeños/medianos
- Migrar a Radix Sort en GPU para grandes volúmenes
- Optimizar la configuración de hilos/bloques según el hardware

Referencias

1. https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/T.pdf
2. https://rachitvasudeva.medium.com/parallel-merge-sort-algorithm-e8175ab60e7
3. https://medium.com/@sj.jainsahil1005/parallel-quick-sort-algorithm-ff8b4cb09bad
4. https://www.uio.no/studier%2Femner%2Fmatnat%2Fifi%2FINF3380%2Fv10%2Fundervisningsmateriale%2Finf3380-week12.pdf%2F
5. https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21572-a-faster-radix-sort-implementation.pdf
6. https://www.irjet.net/archives/V8/i7/IRJET-V8I7714.pdf

Link al repositorio de github:

https://github.com/TomasCB18/Tarea2_INFO188