Programación masivamente paralela en procesadores gráficos (GPUs)

Informe - Práctico 5

Grupo 26

Nombre	**CI**
Agustín Matías Martínez Acuña	5.074.743-0
Natalie Valentina Alaniz Ferreira	5.209.018-4

Índice

1. Introducción

Este es el informe del Práctico 5 del Grupo 26 del curso de Programación Masivamente Paralela en Procesadores Gráficos (GPUs) de la Facultad de Ingeniería de la Universidad de la República [1]. La propuesta es trabajar con bibliotecas generales de paralelización que brinda CUDA, en particular CUB [2] y Thrust [3].

El primer ejercicio requiere la implementación de la operación de *Scan exclusivo* [8] en CUDA. Luego, se compara el rendimiento sobre el rendimiento de un programa. Finalmente, se realizan una serie de pruebas con acceso desalineado a la memoria y como la distribución de datos en dos líneas de caché afecta el desempeño de un programa.

En el segundo ejercicio se busca optimizar el rendimiento de una función que forma parte de una estrategia de paralelización de resolución de sistemas de ecuaciones triangulares.

1.1. Especificación del entorno de prueba y del programa construido

El proyecto está desarrollado en el framework CUDA (C) y cuenta con un archivo `Makefile` para facilitar la compilación.

Las pruebas se realizan en la plataforma de supercomputación *Cluster.uy* [4]

2. Suma exclusiva

La suma exclusiva toma un operador binario (que tiene que ser asociativo) y un arreglo de n elementos $[x_0, x_1, \ldots, x_{n-1}]$ y devuelve el arreglo $[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})]$. El algoritmo 1 muestra un pseudocódigo para la suma exclusiva, se observa que este enfoque es secuencial.

En esta sección vamos a experimentar con tres implementaciones paralelas de esta operación, entre ellas, una implementación en CUDA "pura", es decir sin uso de bibliotecas generales de paralelización, y otras dos utilizando las bibliotecas THRUST y CUB.

Algorithm 1 Suma exclusiva secuencial

Require: Arreglo in de tamaño n
Ensure: Arreglo out de tamaño n tal que cumple la definición de 2
1: $out[0] \leftarrow in[0]$
2: **for** $j = 1$ **to** $n - 1$ **do**
3: $\quad out[j] \leftarrow out[j-1] + in[j]$
4: **end for**

2.1. Implementación sin bibliotecas

En un principio, se implementa el procedimiento `exclusiveScanKernel`[1], basándose en el Capítulo 39 de GPU Gems 3 [5]. La idea principal de esta implementación consiste en construir un árbol binario balanceado en los datos de entrada y "barrerlo" desde y hacia la raíz para calcular la suma exclusiva. Se recuerda que un árbol binario con n hojas tiene $d = log_2(n)$ niveles, y cada nivel d tiene $2d$ nodos.

El algoritmo se divide en dos partes, la fase de reducción (en la literatura citada se encuentra como *Up-sweep phase* y la fase reversa (*Down-sweep phase*):

2.1.1. Etapa de reducción

En la etapa de reducción, se recorre el árbol desde las hojas hasta la raíz, calculando sumas parciales en los nodos internos del árbol, es decir, en las entradas impares del arreglo.

De esta manera, dado i impar, en la i-ésima entrada del arreglo se almacenará la suma:

$$\sum_{j=0}^{i} x_j$$

La figura 1 ilustra el comportamiento descripto anteriormente, y el algoritmo 2 muestra un pseudocódigo de la implementación realizada.

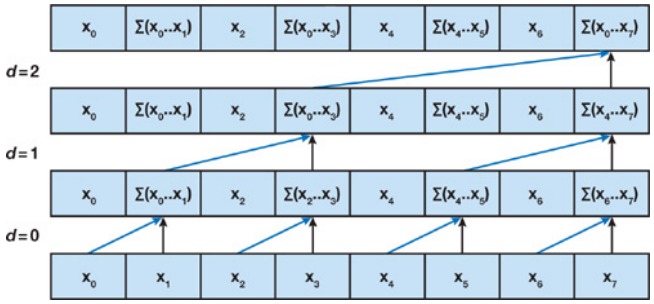


Figura 1: Up-sweep Phase

[1] La implementación se encuentra en el archivo `ej1_a.c`

Algorithm 2 Fase de reducción del Scan Exclusivo

1: **for** $d = 0$ **to** $log_2(n-1)$ **do**
2: **for all** $k = 0$ **to** $n-1$ **by** $2^d + 1$ **paralelamente do**
3: $t = x[k + 2d - 1]$
4: $x[k + 2^d - 1] = x[k + 2^d + 1 - 1]$
5: $x[k + 2^d + 1 - 1] = t + x[k + 2^d + 1 - 1]$
6: **end for all**
7: **end for**

2.1.2. Fase reversa

En la fase reversa, se recorre el árbol hacia abajo desde la raíz, utilizando las sumas parciales de la fase de reducción para construir el resultado. Se comienza asignando 0 a la raíz del árbol y, en cada paso, cada nodo del nivel actual comunica su valor al hijo izquierdo.

La Figura 2 y el Algoritmo 3 ilustran este comportamiento.

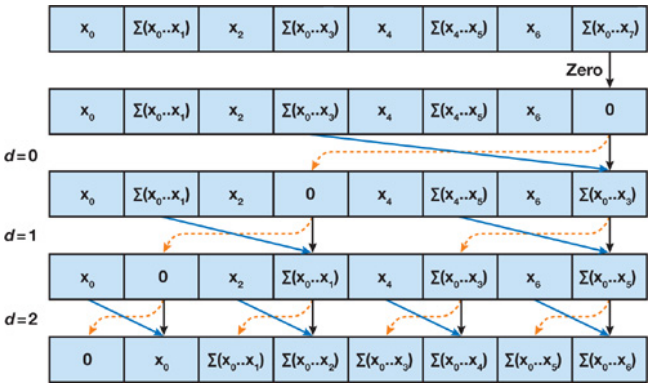


Figura 2: Down-sweep Phase

Algorithm 3 Fase reversa Scan Exclusivo

1: $x[n-1] \leftarrow 0$
2: **for** $d = \log_2(n-1)$ **down to** 0 **do**
3: **for all** $k = 0$ **to** $n-1$ **by** $2^d + 1$ **en paralelo do**
4: $t = x[k + 2^d - 1]$
5: $x[k + 2^d - 1] = x[k + 2^d + 1 - 1]$
6: $x[k + 2^d + 1 - 1] = t + x[k + 2^d + 1 - 1]$
7: **end for**
8: **end for**

2.1.3. Generalización a arreglos de tamaño grande

Dado que cada thread procesa dos elementos, el tamaño de bloque debe ser la mitad del tamaño del arreglo (`threadsPerBlock = n/2`). Esta misma observación tiene como consecuencia que el algoritmo está limitado a un tamaño máximo de arreglo de 2048 elementos. Pues, el algoritmo escanea una arreglo dentro de un solo bloque, y la cantidad máxima de threads en un bloque es 1024. Por lo que la implementación inicial no es capaz de realizar el scan a arreglos de más de 2048 elementos.

Para generalizar la implementación a arreglos de tamaño 2^n, el texto sugiere dividir el arreglo en bloques de tamaño suficientemente pequeños para que se les pueda realizar un scan a cada uno de ellos. Una vez hecho ese paso, se guarda el resultado de cada bloque en un arreglo auxiliar y se aplica un scan al nuevo arreglo. Luego, en la i-ésima entrada del arreglo auxiliar tendremos el valor que hay que sumar al i-ésimo bloque (similar a un offset).

Para lograr esto se identifican 4 etapas, se comienza con el arreglo inicial:

- Etapa 1: Realizar el scan por bloques del arreglo.
- Etapa 2: Guardar la suma final de cada scan en un arreglo auxiliar, el índice en el que se guarda es el identificador del bloque (`aux[blockIdx.x] = sumaParcial`).
- Etapa 3: Realizar scan al arreglo de sumas parciales.
- Etapa 4: Sumar el offset al bloque de scan correspondiente (paralelamente a todos los bloques).

La Figura 3 muestra un comportamiento similar al descripto en el párrafo anterior, la diferencia es que para el paso cuatro se suma al bloque `i` en lugar de `i+1`, ya que el bloque `offsets[0]` contiene el valor 0 (no se suma nada al bloque 0 por neutro de la suma).

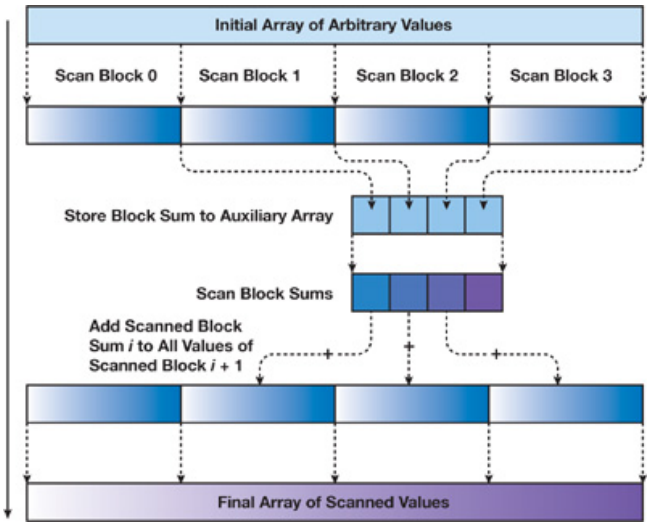


Figura 3: Esquema para generalizar la implementación a arreglos de tamaño 2^n.

Se observa que la etapa 4 es altamente paralelizable, por lo que se escribe un sencillo Kernel (`sumOffsetsKernel`) para realizar esta tarea. La lógica de este kernel es que cada thread de un bloque con identificador i leerá el offset correspondiente y lo aplicará a dos elementos del arreglo.

En la implementación realizada, la etapa 3 (realizar scan del arreglo auxiliar) se realiza de forma secuencial. Esto se hace a pesar que se genere una cota inferior del tiempo de ejecución del programa (por corolario de la ley de Amdahl) y en arreglos enormes pueda llegar a ser un cuello de botella. La justificación consiste en que partir de $n = 2048$, hay un bloque cada 2048 elementos del arreglo, por lo que, la cantidad de bloques (y equivalentemente, la cantidad de entradas del arreglo auxiliar) se rige por la siguiente igualdad (asumimos que el tamaño de la entrada es 1024×2^n):

$$\#Bloques = \frac{n}{\#datosProcesadosPorThread \times \#threadsPorBloque} = \frac{n}{2 \times 1024} = \frac{n}{2048}$$

Creemos que el número de bloques es sustantivamente más bajo que el tamaño del arreglo (en el contexto del práctico), y que es manejable a nivel secuencial. A modo de ejemplo, para un arreglo de tamaño $2^{18} = 262.144$, la cantidad de bloques será $\frac{2^{18}}{2^{11}} = 2^7 = 128$. Sin embargo, cabe aclarar que **si se aumenta exponencialmente el tamaño del arreglo, el tamaño de bloque también aumentará exponencialmente, solo que estará dividido por la constante** $c = 2048$**.**

Queda como oportunidad de mejora paralelizar la fase 3 (no es necesario implementarlo por letra).

2.1.4. Resultados y análisis

Para analizar los resultados de la implementación paralela, es necesario tener una referencia previa, por ejemplo una línea base para comparar y observar si efectivamente hay una mejora de rendimiento. Para esto se implementó una suma exclusiva secuencial[2] y sus resultados de ejecución se pueden observar en la tabla 2 del anexo.

La Figura 4 muestra el tiempo de ejecución en nanosegundos (ns) en función del tamaño de la entrada, las pruebas se realizaron en vectores de tamaño 1024×2^k, para todo $k \in \{0, 1, 2, ..., 15\}$, que consideramos fue un tamaño de muestra suficiente para sacar conclusiones. Si se desea observar los resultados numéricos, consulte la Tabla 2 del Anexo.

Comenzando con lo más obvio, se observa que a medida que crece le tamaño de la entrada, el tiempo de ejecución secuencial alcanza valores extremadamente altos, superando los 80 millones de nanosegundos. Es claro que esta estrategia no escala de manera aceptable.

La operación scan tiene orden $O(n)$ y en la ejecución secuencial es ayudado sustantivamente por el prefetch, dado que la recorrida del arreglo es de izquierda a derecha, sin saltos fijos o aleatorios. Estos factores indican que la tasa de crecimiento es constante (Tiempo lineal). Sin embargo, la línea de tendencia no se asemeja a una función lineal: a medida que crece el tamaño de vector, la pendiente tiende a disminuir.

La naturaleza del experimento puede no estar ayudando a concluir más allá de toda duda, dado que este consiste en aumentar exponencialmente el tamaño del vector, aumentando cada vez más la distancia entre las muestras, esto puede llevar a perder aspectos intermedios vitales para el análisis.

Siguiendo con la Ejecución paralela, se observa una clara tendencia lineal que su contra-parte secuencial (la Figura 5 ilustra mejor la tendencia). La diferencia en tiempo de ejecución está en el orden de decenas de millones de nanosegundos (es decir, decenas de milisegundos), lo cuál muestra las ventajas de paralelizar la suma exclusiva.

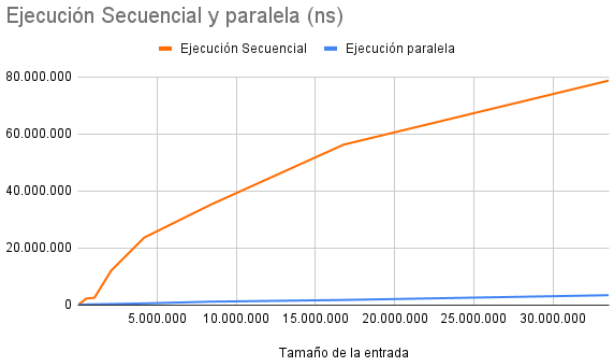


Figura 4: Ejecución Secuencial y Paralela

2.2. Implementaciones utilizando Thrust y Cub

Dado que la suma exclusiva es uno de los patrones más estudiados de computación paralela [6], las bibliotecas generales de paralelización incluyen la operación de suma exclusiva.

En el caso de Thrust, basta con declarar un vector en la memoria del device (llamémosle `d_vec`), a través del tipo de dato `device_vector`, cargando contenido dentro del mismo (puede ser copiando un arreglo en memoria del host) y llamando a la rutina `exclusive_scan(d_vec.begin(), d_vec.end(), d_vec.begin())` [7]

[2] Ver archivo `ej1_secuencial.cu`

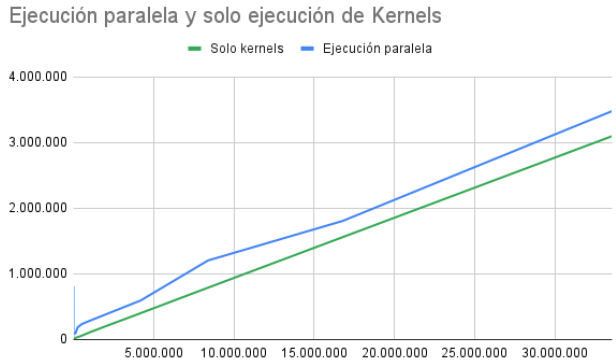


Figura 5: Tiempo de ejecución paralela, en verde se ve el tiempo que toma solamente ambos kernels

Para el caso de Cub es necesario hacer un precálculo del espacio que se necesita en device para hacer la operación, esto se hace a través de una llamada a `cub::ExclusiveSum` pasando por parámetro un vector inicializado en `NULL`. Luego de la primer ejecución, se debe reservar memoria en device para ese vector, y luego sí se puede ejecutar la suma exclusiva con la misma directiva.

2.3. Resultados y análisis

Las tres implementaciones se mostraron robustas para vectores de tamaño 2^k, cumpliendo con el objetivo.

Para hacer un análisis correcto de la performance no alcanza con medir los tiempos de ejecución de cada Kernel, ya que dependiendo de la implementación, se pueden hacer operaciones y reservas de memoria adicionales.

Para ser lo más justos posible, se recurre al uso de eventos de CUDA, a grosso modo, un cudaEvent es un mecanismo que permite medir el tiempo entre dos puntos en un programa que utiliza la GPU, así como sincronizar las operaciones en la GPU [9]. En esta tarea se utilizan solamente para medir tiempos para capturar las ejecuciones. Para minimizar el ruido, hay aspectos que son comunes a las tres implementaciones que no serán capturados por los eventos (por ejemplo, generar un arreglo inicial, copiarlo a device, etc...),

Entonces, siguiendo lineamientos similares a los experimentos de la parte anterior, el experimento consiste en ejecutar las tres implementaciones de la Suma Exclusiva con arreglos de tamaño 1024×2^k, $\forall k \in \{0, 1, 2, 3, ..., 15\}$, tomando el tiempo de ejecución para cada una. Para mantener las pruebas lo más parecidas posibles, se generan arreglos de modo que coincida el contenido para las distintas ejecuciones con un mismo k. También se repiten las ejecuciones múltiples veces con el fin de establecer el tiempo promedio.

Los resultados se pueden consultar en la Tabla 3. La Figura 6 muestra la misma serie que la Figura 5 para el tiempo de la implementación, agregando los datos de la ejecución de los algoritmos que utilizan Thrust (en anaranjado) y Cub (en amarillo).

En general se observa una diferencia sustantiva entre el tiempo de ejecución de las implementaciones con bibliotecas que la que utiliza un Kernel propio, esto es esperable porque las bibliotecas suelen ser implementaciones optimizadas. Esta diferencia se visualiza a partir de arreglos con más de 5 millones de entradas ($n > 1024 \times 2^{12}$)

A medida que aumenta k, la diferencia tiende a crecer, por ejemplo, para $k = 15$ ($n = 1024 \times 2^{15}$) la diferencia es de 2 milisegundos, la cuál es muy alta, y la tendencia indicaría a que la brecha siga creciendo.

Los tiempos de los algoritmos que utilizan bibliotecas son alarmantemente similares, tras investigar las implementaciones de cada uno en los repositorios correspondientes, se observa que el método `exlusive_scan`

de Thrust utiliza Cub para funcionar [7], por ejemplo para averiguar la cantidad de almacenamiento temporal (tal como se hizo en la implementación nuestra utilizando Cub).

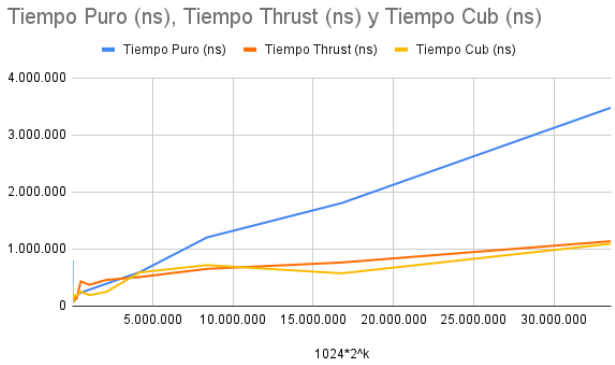


Figura 6: Tiempo Puro (ns), Tiempo Thrust (ns) y Tiempo Cub (ns).png

3. Ejercicio 2

En esta sección se trabaja sobre el problema de resolver un sistema de ecuaciones triangular. Para resolver un sistema triangular, es fundamental abordar las ecuaciones (representadas por filas en la matriz triangular) en un orden específico, ya que cada ecuación depende de las incógnitas de las filas anteriores. [1]

Siguiendo esta lógica, se puede definir una estructura de niveles basada en las dependencias: el nivel 0 corresponde a las ecuaciones que no dependen de ninguna otra, y el nivel $i + 1$ incluye aquellas que dependen de las del nivel i.

Una estrategia para paralelizar la resolución es ejecutar cada nivel en paralelo. No obstante, si se asigna un warp para cada fila, se desperdicia capacidad de cómputo en filas con pocos elementos diferentes de 0. Por eso, una estrategia de optimización es ejecutar filas del mismo nivel y de tamaño similar en el mismo warp (por ejemplo, 16 filas con dos elementos o 8 filas con cuatro). Esta es la estrategia aplicada en el código proporcionado.

Para poder lograr el planteo, es necesario generar un orden de ejecución de las filas en función del nivel y su tamaño, que permita agrupar a las del mismo nivel que tienen tamaño similar. La función `ordenar_filas` es la encargada de esta tarea. Esta invoca a un kernel que calcula los niveles llamado `kernel_analysis_L`, dicho kernel recibe los vectores de fila (`row_ptr`), índices de columna (`col_idx`) y un vector auxiliar (`is_solved`) inicializado en 0 y retorna un vector que mantiene en cada posición el nivel de la fila. El retorno de la función es la cantidad de warps necesarios para resolver el sistema triangular dado por esa matriz independientemente de los valores de la matriz.

El objetivo de esta sección es paralelizar el método `ordenar_filas` utilizando Thrust [3].

3.1. Código a paralelizar

En el procedimiento `ordenar_filas` se identifican cinco fragmentos de código potencialmente paralelizable, reconocidos por contener un bucle `for` secuencial, trabajando en CPU. En esta subsección se describe cada uno de esos fragmentos junto con la propuesta de paralelización.

3.1.1. Obtener el máximo nivel

En el primer fragmento se calcula el máximo de un arreglo de enteros, el código secuencial realiza una búsqueda total tal como muestra el Algoritmo 4.

Algorithm 4 Obtener el máximo nivel (secuencial)

```
1: int nLevs = niveles[0];
2: for int i = 1; i < n; ++i) do
3:     nLevs = MAX(nLevs, niveles[i]);
4: end for
```

La propuesta de paralelización es a través de una reducción, con la operación `maximum`, nativa de Thrust. Para esto se usa el tipo `device_ptr<unsigned int>`, que como su nombre lo indica, almacena un puntero a un espacio de memoria del dispositivo.

El código de esta propuesta puede verse en el Algoritmo 5. La directiva en la línea 2 indica hacer una reducción desde el principio hasta el final del arreglo (cargado en la device memory), aplicando la operación maximum.

3.1.2. Contar el número de filas en cada nivel y clase de equivalencia de tamaño

Luego de obtenido el máximo nivel, se cuenta el número de filas en cada nivel, y a que clase de equivalencia de tamaño se corresponde. Esto se puede paralelizar por nivel.

El algoritmo 6 muestra como se logra en la estrategia secuencial, para la paralelizar este fragmento de

Algorithm 5 Obtener el máximo nivel (paralelo)

```
thrust::device_ptr<unsigned int> dev_ptr_d_niveles(d_niveles);
nLevs=thrust::reduce(
    thrust::device,
    dev_ptr_d_niveles,
    dev_ptr_d_niveles + n,
    0,
    thrust::maximum<unsigned int>()
);
```

código, se crea una estructura auxiliar `IncrementarIndice`, que implementa la misma lógica de selección del algoritmo pero para un índice arbitrario `i`, con la diferencia que cuando se incremente la cantidad de filas para un nivel, se tiene el cuidado de hacerlo con `atomic_add`.

Luego, desde Thrust se ejecuta un `for_each` paralelo tal que a cada fila le aplique la operación definida en la estructura auxiliar. Esto puede hacerse porque no hay dependencias entre las filas a contar.

Algorithm 6 Contar el número de filas en cada nivel y clase de equivalencia de tamaño (secuencial)

```
for (int i = 0; i< n; i++ ) do
    int lev = niveles[i]-1;
    int nnz_row = RowPtrL_h[i+1]-RowPtrL_h[i]-1;
    int vect_size;
    select_clase();
    ivects[7*lev+vect_size]++;
end for
```

Algorithm 7 Contar el número de filas en cada nivel y clase de equivalencia de tamaño (paralelo)

```
thrust::device_vector<int> d_niveles_T(niveles, niveles + n);
thrust::device_vector<int> d_RowPtrL_T(RowPtrL_h, RowPtrL_h + n + 1); //se copia el arreglo RowPtrL_h a un device_vector (tamaño n+1)
thrust::device_vector<int> d_ivects_T(7*nLevs, 0);
thrust::for_each(
    thrust::device,
    thrust::counting_iterator<int>(0),
    thrust::counting_iterator<int>(n),
    IncrementIndex{
        thrust::raw_pointer_cast(d_ivects_T.data()),
        thrust::raw_pointer_cast(d_niveles_T.data()),
        thrust::raw_pointer_cast(d_RowPtrL_T.data())
    }
);
```

3.1.3. Suma prefija del vector

Siguiendo la lógica del código, si se hace una suma prefija del vector se obtiene el punto de comienzo de cada par `(tamaño, nivel)` en el vector final ordenado. Esto se puede paralelizar trivialmente llamando a la directiva `thrust::exclusive_scan`.

3.1.4. Generar orden

Para generar un orden el algoritmo secuencial recorre por filas y generar un orden utilizando el nivel (`idepth`) y la clase de tamaño (`vect_size`) como clave. El algoritmo 8 muestra la implementación dada por letra.

Algorithm 8 Generar orden (secuencial)

```
1: for i = 0 to n − 1 do
2:    idepth = niveles[i]-1;
3:    nnz_row = RowPtrL_h[i+1]-RowPtrL_h[i]-1;
4:    vect_size;
5:
6:    selectClass(nnz_rows);
7:
8:    iorder[ ivects[ 7*idepth+vect_size ] ] = i;
9:    ivect_size[ ivects[ 7*idepth+vect_size ] ] = ( vect_size == 6)? 0 : pow(2,vect_size);
10:   ivects[ 7*idepth+vect_size ]++;
11: end for
```

Si esta fase se paraleliza sin cuidado (pasando la operación a un struct y llamando `thrust::transform`), se podrían dar condiciones de carrera en las tres últimas asignaciones, donde pueden sobreescribirse valores, o incrementar el índice dos veces seguidas antes de modificar los valores. Los accesos realizados en las tres últimas instrucciones conforman una sección crítica.

Para resolver este problema, para una misma key, un solo hilo debe escribir en esos espacios de memoria. Esto se logra primero realizando el incremento de ivects (instrucción 10 en el Algoritmo 8)), pero este debe realizarse atomicamente (a través de `atomicAdd(&ivects[key], 1);`), para recuperar el índice previo al incremento, basta asignar la función atomicAdd a una variable.

El atomicAdd genera una barrera, pues ahora cualquier hilo que acceda con la misma clave, ya tendrá incrementado el índice y puede reservar su espacio a través de su propio atomicAdd, luego de eso podrá hacer las escrituras de la línea 8 y 9 con la certeza de que ningún otro hilo accederá a esos espacios de memoria.

La lógica descripta anteriormente se engloba en el `struct generarOrden`, y se llama a este a través de `thrust::for_each`, similar a la llamada realizada en el Algoritmo 7.

3.1.5. Asignación a warps

La asignación a warps en el Algoritmo secuencial se da de manera natural, recorriendo el orden dado por `iorder`, y sumando una fila al warp actual si y solo sí se cumplen las siguientes condiciones:

- Coincide el nivel de la fila `ctr` con la fila `ctr-1`.
- Coincide el tamaño de la fila `ctr` con la fila `ctr-1`
- El warp tiene espacio suficiente para incluir la fila:
  - `filas_warp * ivect_size[ctr] < 32`
  - `not(ivect_size[ctr] == 0 && filas_warp == 32)`

La estrategia de paralelización fue utilizar como clave el nivel y el tamaño de fila, ya que un warp contiene solamente filas que estén en el mismo nivel y tengan el mismo tamaño. Una vez generado el conjunto de claves se ejecuta un `reduce_by_key` sobre él, dando como resultado cuantas filas para determinado nivel y tamaño hay. Para contar la cantidad de warps a asignar, se acumula en la variable `ii += (filas_warp * size + 31) / 32`. Sin embargo la estrategia no pareció funcionar, ya que en realidad el resultado es una aproximación a la cantidad de warps necesarios, y puede fallar en el orden de las decenas. Por lo que en el código entregado se encuentra comentado, y se utiliza la estrategia secuencial para el análisis de rendimiento.

3.2. Análisis de rendimiento

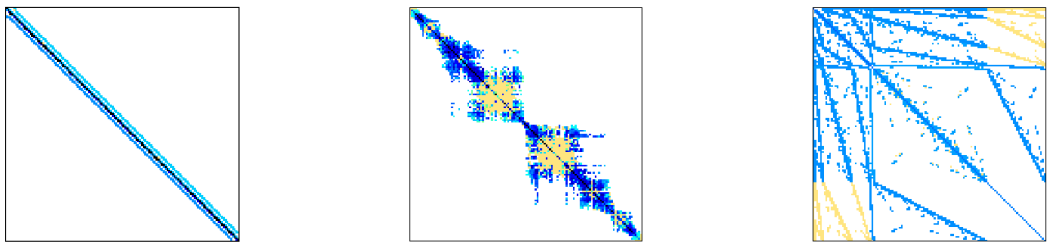
3.2.1. Matrices de prueba

Para realizar las pruebas, se obtiene un conjunto de Matrices Semidefinidas Positivas de la colección de matrices dispersas (SDPs) *SuiteSparse Matrix Collection* [10]. Las propiedades tenidas en cuenta para las matrices son: Número de entradas diferentes de cero (a partir de ahora nnz), dimensión, densidad, estructura). Estas propiedades junto con el tiempo de ejecución de la ejecución secuencial y la paralela se pueden apreciar en la Tabla 1.

Nombre	NNZ	Dimensión	Densidad	Dispersión
Goodwin_040	561.677	17.922 x 17.922	0,002 %	n-banda
gyro_m	340.431	17.361 x 17.361	0,001 %	Dispersa
Dubcova1	253.009	16.129 x 16.129	0,001 %	Muy Dispersa
bcsstk17	428.650	10.974 x 10.974	0,004 %	Mayormente Diagonal
bcsstk13	83.883	2.003 x 2.003	0,021 %	Muy Dispersa
1138_bus	4.054	1.138 x 1.138	0,003 %	Muy Dispersa

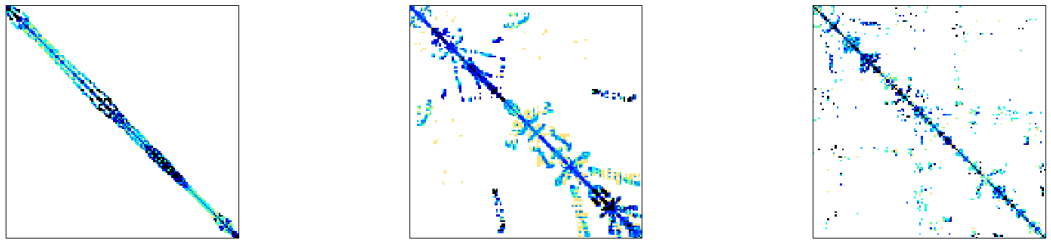
Cuadro 1: Matrices

En las figuras 7 y 8 se pueden observar las estructuras de las matrices:



(a) Goodwin_40 (b) gyro_m (c) Dubcova1

Figura 7: Estructura matrices [1]



(a) bcsstk17 (b) bcsstk13 (c) 1138_bus

Figura 8: Estructura matrices [2]

3.3. Resultados

Los resultados numéricos pueden consultarse en la Tabla 4 del Anexo. Las Figuras 9a y 9b, muestran el tiempo de ejecución sobre el conjunto de matrices para la función original y paralela respectivamente, así como el tiempo de ejecución solo del fragmento de código a paralelizar. Las matrices están ordenadas de mayor a menor dimensión y NNZ. También, la Figura 10 aporta la visualización de las diferencias de performance en cada matriz, si la diferencia es positiva, entonces el algoritmo paralelo es superior que el algoritmo secuencial.

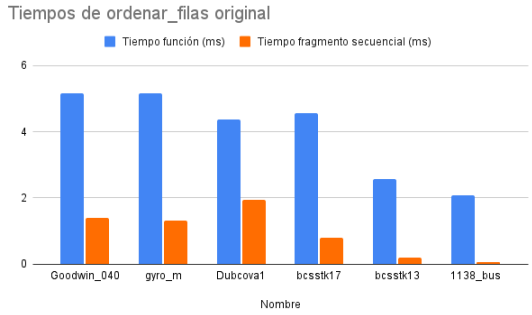
En primera instancia, los tiempos del fragmento paralelo descienden en su mayoría con respecto a su contraparte secuencial, pero el tiempo de ejecución de la función no acompaña esta tendencia. El tiempo secuencial sigue una tendencia intuitiva, a mayor dimensión y cantidad de NNZ, mayor tiempo, mientras que la función paralela tiene mayor tiempo para matrices medianas, esto puede ser por varios factores, por ejemplo la estructura de las matrices.

Se observa que las matrices Goodwin_040 y gyro_m muestran mejoras significativas tanto en la función completa como en el fragmento paralelizado (un poco menos de un 1ms). Esto indica que la paralelización ha sido efectiva en estos casos (notar que estas matrices son las más grandes). Sin embargo, en el caso de la matriz Dubcova1, se observa una disminución sustantiva en el rendimiento de la función completa, a pesar de que el fragmento paralelizado muestra una pequeña mejora. Este comportamiento indica que, aunque la paralelización del fragmento específico fue exitosa, otros factores influyeron negativamente en el rendimiento general. Es probable que la estructura compleja de Dubcova1 sea un obstáculo para la paralelización.

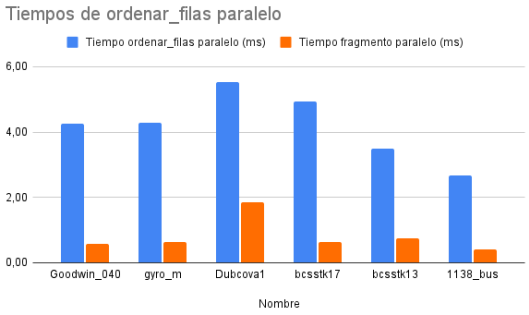
Las matrices bcsstk17 muestran un ligero decaimiento en la función completa y una mejora menor en el fragmento paralelizado, sugiriendo una eficiencia moderada de la paralelización. Por otro lado, las matrices bcsstk13 y 1138_bus, que son las más pequeñas de la selección, presentan disminuciones en el rendimiento de la función completa y en el fragmento paralelo. Se sospecha que **el tamaño de estas matrices no justifican el uso de paraleización**.

Generalizando la noción del párrafo anterior, es claro que para matrices pequeñas, hay un overhead dominante en la reserva de memoria y ejecución de las técnicas de paralelización que enlentece el cálculo. Cuando los tamaños de matrices crecen, el overhead tiende a ser despreciable, siendo más conveniente la utilización de la implementación paralela.

Como mejora a futuro, se puede experimentar con matrices más grandes.



(a) Tiempos de `ordenar_filas` original



(b) Tiempos de `ordenar_filas` paralelo

Figura 9: Comparación de tiempos de ordenar_filas

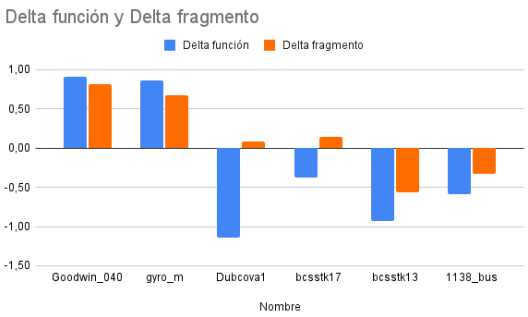


Figura 10: Delta función y Delta fragmento

4. Conclusiones

En este informe se ha realizado un análisis detallado de la suma exclusiva (scan) en CUDA utilizando tres enfoques diferentes: una implementación manual en CUDA "pura", y dos implementaciones basadas en las bibliotecas Thrust y Cub. Además, se ha paralelizado la función `ordenar_filas` utilizando Thrust, y se ha evaluado su rendimiento en comparación con la versión secuencial.

Podemos concluir que las bibliotecas de paralelización no solo son herramientas útiles y prácticas para el desarrollo de aplicaciones paralelas, sino que tienen mucha mejor performance y escalabilidad que las implementaciones propias.

La implementación manual de la suma exclusiva en CUDA, aunque funcional y mucho más rápida que la versión secuencial, mostró tiempos de ejecución sustantivamente mayores en comparación con Thrust y Cub. Las bibliotecas optimizadas manejan eficientemente arreglos de gran tamaño, teniendo una clara ventaja en términos de rendimiento y escalabilidad.

Sobre el ejercicio 2, la función `ordenar_filas` paralelizada mostró mejoras en la mayoría de los casos. Sin embargo, hubieron casos donde la performance empeoró, una posible razón es que el tamaño de las matrices no fue lo suficientemente grande como para justificar el uso de la función altamente paralelizada. De todas formas, se puede estudiar por qué la performance empeora para esos casos.

5. Referencias

Referencias

[1] Letra del Práctico 5 | GPGPU 2024 | FING | Udelar <eva.fing.edu.uy/course/view.php?id=1076>.

[2] CUB <docs.nvidia.com/cuda/cub/index.html>.

[3] Thrust <nvidia.github.io/cccl/thrust>.

[4] Página web del Centro Nacional de Supercomputación (Uruguay) <cluster.uy>.

[5] GPU Gems 3 | Chapter 39 | Parallel Prefix Sum (Scan) with CUDA <developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda>.

[6] Patrones de Cómputo 2 | GPGPU 2024 | FING | Udelar <https://eva.fing.edu.uy/pluginfile.php/504457/mod_label/intro/Clase8%20-%20Patrones%202.pdf>

[7] Documentación de Thrust - `exclusive_scan` <https://nvidia.github.io/cccl/thrust/api/function_group__prefixsums_1ga8dbe92b545e14800f567c69624238d17.html>

[8] Diapositivas del Curso | GPGPU 2024 | FING | Udelar <eva.fing.edu.uy/pluginfile.php/504457/mod_label/intro/Clase8%20-%20Patrones%202.pdf>.

[9] CUDA Runtime API - 6.6. Event Management <https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html>.

[10] SuiteSparse Matrix Collection, <sparse.tamu.edu>.

6. Anexo

6.1. Tablas del Ejercicio 1

Cuadro 2: Comparación de tiempos de ejecución

n	1024×2^n	Tiempo secuencial (ns)	Tiempo paralelo (ns)
0	1.024	1.780	810.976
1	2.048	5.629	84.927
2	4.096	15.377	94.944
3	8.192	27.772	89.472
4	16.384	56.150	100.928
5	32.768	118.635	93.216
6	65.536	266.384	95.711
7	131.072	583.347	95.200
8	262.144	1.090.196	184.256
9	524.288	2.293.296	235.328
10	1.048.576	2.584.903	288.320
11	2.097.152	12.065.493	390.751
12	4.194.304	23.682.811	595.167
13	8.388.608	35.208.017	1.204.480
14	16.777.216	56.249.337	1.806.751
15	33.554.432	78.751.640	3.479.552

Cuadro 3: Comparación de tiempos de ejecución

n	**Tamaño de la entrada**	**Tiempo Paralelo (ns)**	**Tiempo Thrust (ns)**	**Tiempo Cub (ns)**
0	1.024	810.976	149.215	350.784
1	2.048	84.927	127.360	99.104
2	4.096	94.944	110.944	95.040
3	8.192	89.472	126.527	100.703
4	16.384	100.928	110.431	101.824
5	32.768	93.216	118.047	99.775
6	65.536	95.711	129.536	88.863
7	131.072	95.200	120.863	181.856
8	262.144	184.256	131.456	181.311
9	524.288	235.328	433.952	254.079
10	1.048.576	288.320	373.984	192.959
11	2.097.152	390.751	458.047	248.032
12	4.194.304	595.167	509.887	590.623
13	8.388.608	1.204.480	653.504	718.591
14	16.777.216	1.806.751	764.671	574.912
15	33.554.432	3.479.552	1.138.368	1.094.336

6.2. Tablas del Ejercicio 2

Nombre	Total sec (ms)	Frag sec (ms)	Total paral (ms)	Frag paral (ms)	Delta Total	Delta Frag
Goodwin_040	5.18	1.39	4.27	0.58	0.91	0.81
gyro_m	5.16	1.30	4.30	0.62	0.86	0.68
Dubcova1	4.38	1.93	5.52	1.84	-1.14	0.09
bcsstk17	4.57	0.78	4.93	0.63	-0.37	0.15
bcsstk13	2.56	0.18	3.49	0.74	-0.93	-0.56
1138_bus	2.08	0.07	2.66	0.40	-0.59	-0.33

Cuadro 4: Tabla de tiempos y deltas