Informe - Practico 4

Mateo Quiller, Leonardo Pombo, Joaquín Mezquita

Grupo 8

30 de noviembre de 2025

Índice

1. Ejercicio 1 - Implementación de Exclusive Scan

1.1. Implementación sin librerías

1.1.1. Descripción del Algoritmo

Para la implementación de esta solución nos basamos en el algoritmo de Blelloch que se basa en las siguientes etapas:

1. **Up-sweep (Reducción):** Construye un árbol binario de sumas parciales, donde cada nivel del árbol reduce la cantidad de elementos activos a la mitad.
2. **Down-sweep (Distribución):** Distribuye las sumas parciales hacia abajo para obtener el exclusive scan final.

1.1.2. Implementación CUDA

La implementación se divide en tres kernels principales:

Kernel `partial_inclusive_scan`: Este kernel implementa el algoritmo de Blelloch sobre cada bloque individualmente:

- Utiliza memoria compartida (`__shared__`) para minimizar accesos a memoria global.
- Cada thread procesa dos elementos del arreglo para maximizar la utilización de recursos.
- Implementa las fases up-sweep y down-sweep del algoritmo de Blelloch.
- Almacena el resultado parcial de cada bloque en `offsets_array`.
- Tiene la limitante de que puede procesar como máximo 2048 elementos del array.

Kernel `scan_adjust`: Este kernel combina los resultados parciales de todos los bloques:

- Suma los offsets acumulados de bloques anteriores.
- Ajusta los valores del pre-scan para obtener el exclusive scan final.
- Maneja correctamente el valor inicial del exclusive scan.

Kernel `exclusive_scan`: Coordina la ejecución de todo el algoritmo invocando a los otros 2 kernels, para lo que se requiere disponer de paralelismo dinámico:

- Gestiona la memoria temporal necesaria (prescan_array, offsets_array).
- Ejecuta el scan parcial en paralelo para todos los bloques dado que cada bloque puede procesar únicamente 2048 elementos.

- Calcula el scan de los offsets de bloques utilizando nuevamente el kernel partial_-inclusive_scan para determinar cuál es el ajuste que hay que realizar sobre cada bloque.
- Aplica los ajustes finales para obtener el resultado correcto.

1.1.3. Características de la Implementación

- **Tamaño de bloque:** 1024 threads por bloque.
- **Elementos por thread:** 2 elementos para maximizar ocupación.
- **Uso de memoria compartida:** Reduce significativamente la latencia de acceso a memoria.
- **Sincronización:** Uso adecuado de `__syncthreads()` para coordinar threads dentro de cada bloque.

1.1.4. Análisis de Resultados

Para vectores de tamaño 1024×2^N con $N = [1, 2, ..., 10]$, se observaron los siguientes patrones:

- **Escalabilidad:** El tiempo no crece linealmente con N, pero aún así se nota un leve crecimiento en los tiempos de ejecución.
- **Rendimiento óptimo:** Hasta N=7 Se puede observar un crecimiento moderado en el tiempo promedio y tiempos similares de Desviación estándar, pero con N >7 se observa un salto considerable en tiempo promedio pero una baja en la desviación estándar, lo que nos indica que el kernel es más predecible en vectores grandes.

Se puede ver en detalle los tiempos para cada N en la tabla 1.

1.2. Uso de librerías

La librería CUB de NVIDIA proporciona primitivas altamente optimizadas para operaciones paralelas.

Thrust proporciona una interfaz de alto nivel similar a STL.

Por último NVTX fué utilizada para agregar marcadores en el código para posteriormente poder medir los tiempos de ejecución con la herramienta nsys.

La implementación en detalle de ambas puede verse en 1 y 2.

1.2.1. Análisis y comparación de rendimiento

Los resultados experimentales muestran que:

- **CUB** ofrece el mejor rendimiento, especialmente para vectores grandes, debido a optimizaciones específicas como el uso de múltiples algoritmos según el tamaño de entrada

- **Thrust** proporciona un rendimiento similar a CUB hasta N=7 y tiene la gran ventaja de ser extremadamente simple de utilizar
- **Implementación manual** resulta muy complejo resolver los conflictos de banco, creemos que para algoritmos que son genéricos el uso de una librería es lo esperable

La implementación manual demuestra la complejidad en la optimización de algoritmos paralelos, mientras que las librerías como CUB y Thrust abstraen esta complejidad proporcionando soluciones robustas y eficientes para la mayoría de casos de uso. Los resultados con detalle puede verse en la tabla 2 y 3.

2. Ejercicio 2 - Uso de "building blocks" para paralelizar rutinas complejas

2.1. Paralelización

La función `ordenar_filas()` implementa un algoritmo secuencial de tres etapas para determinar la cantidad de warps necesarios para resolver un sistema matricial triangular inferior teniendo en cuenta las dependencias entre filas.

2.1.1. Algoritmo Original

El algoritmo secuencial consta de las siguientes etapas:

1. **Cálculo de niveles**: Utiliza el kernel kernel_analysis_L para determinar el nivel de dependencia de cada fila.
2. **Clasificación por tamaño**: Categoriza las filas según su número de elementos no-cero y según el nivel generando una clave única.
3. **Ordenamiento**: A partir de la clave generada se procesan los elementos y se genera el vector iorder.
4. **Simulación de warps**: Cuenta el número total de warps necesarios considerando las restricciones de nivel y tamaño.

2.1.2. Estrategia de Paralelización

Para la paralelización se utilizó la biblioteca **Thrust**, aprovechando los siguientes métodos:

- `thrust::transform`: En nuestro caso fue usado con functores personalizados para el cálculo de claves combinadas en paralelo, siguiendo la misma idea que se usa en el código secuencial.
- `thrust::stable_sort_by_key`: Usado para agrupar las filas que tienen la misma clave para que luego el reduce by key funcione correctamente ya que reduce by key solo agrupa las claves si están contiguas en el array.
- `thrust::reduce_by_key`: Para conteo de filas con la misma clave.

2.1.3. Functores Implementados

Se desarrollaron dos functores especializados:

- «calculate_keys_functor» Permite paralelizar el cálculo de las claves combinadas.
- «calculate_warps_per_group» permite paralelizar el cálculo de la cantidad de warps que serán necesarios para una determinada clave.

Se puede ver en mas detalle el código de cada uno en 3 y 4.

2.1.4. Flujo de la Versión Paralela

La implementación paralela sigue este flujo:

1. **Cálculo de niveles**: Reutiliza el kernel original.
2. **Generación de claves**: Usa `thrust::transform` con el functor calculate_keys_functor para generar una clave única según la cantidad de elementos distintos de cero en la fila (usando las clases de equivalencia explicadas anteriormente) y el nivel.
3. **Ordenamiento**: Aplica `thrust::stable_sort_by_key` para ordenar las claves manteniendo el orden respecto a los valores es decir: Si tenemos keys = (1,2,3,2) y values (4,3,2,1) reordena las claves en orden ascendente (1,2,2,3) y los valores asociados son reordenados manteniendo la asociación clave,valor (4,3,1,2) (permite construir iorder).
4. **Conteo de filas con igual clave**: Utiliza `thrust::reduce_by_key` con un iterador constante de 1s `constant_iterator` por lo que como resultado obtenemos para cada clave la cantidad de filas que tienen esa clave.
5. **Cálculo de warps**: Transforma cada grupo ya reducido a la cantidad de warps requeridos según la cantidad de filas que tienen esa clave.
6. **Reducción final**: Suma total de warps con `thrust::reduce`.

2.2. Comparación de desempeño

Para el análisis se utilizaron distintas matrices de la colección SuiteSparse con características variadas como se especifica en la letra, se puede ver con mas detalle las características de cada una en la tabla 4.

2.2.1. Análisis de Tiempos

Los tiempos se midieron considerando dos componentes:

- **Tiempo de paralelización**: Solo la parte reemplazada (ordenamiento y conteo).
- **Tiempo total**: Incluye cálculo de niveles y transferencias de memoria.

Los resultados pueden verse en la tabla 5.

2.2.2. Análisis de Resultados

La paralelización de la lógica de ordenamiento y conteo muestra speedups significativos (de hasta 35.63 %), con mejor rendimiento en matrices grandes debido a:

- Menor impacto del overhead generado por el lanzamiento de threads y copias a memoria de la GPU.
- Uso más eficiente de los recursos de GPU.

En contraposición, cuando se trabaja con matrices de baja dimensión (en el contexto de las matrices usadas), se detecta una baja de rendimiento por parte del algoritmo paralelo, en particular hasta un -59 % de perdida de eficiencia.

Podemos concluir que el algoritmo paralelo planteado es altamente efectivo cuando la dimensión de la matriz es enorme, y no le afecta las demás características puntuales (simetría y/o cantidad de ceros).

Como nota, por lo visto en las conclusiones del ejercicio 1, es esperable que los tiempos del algoritmo paralelo mejoren aun mas si se utilizara la librería de CUB.

3. Anexo

Cuadro 1: Tiempos de ejecución para la implementación manual de Exclusive Scan

N	Tamaño del Vector (L)	Tiempo Promedio (ns)	Desviación estándar (ns)
1	$1024 \times 2^1 = 2048$	81,947	7,127.9
2	$1024 \times 2^2 = 4096$	82,610	7,061.5
3	$1024 \times 2^3 = 8192$	93,451	7,804.1
4	$1024 \times 2^4 = 16384$	83,525	7,710.8
5	$1024 \times 2^5 = 32768$	89,545	7,433.9
6	$1024 \times 2^6 = 65536$	93,529	8,386.1
7	$1024 \times 2^7 = 131072$	96,428	7,505.8
8	$1024 \times 2^8 = 262144$	120,627	4,750.1
9	$1024 \times 2^9 = 524288$	127,878	4,484.1
10	$1024 \times 2^{10} = 1048576$	112,377	4,573.6

Cuadro 2: Comparación tiempos de ejecución de las distintas implementaciones

N	Tiempo manual (ns)	Tiempo CUB (ns)	Tiempo Thrust (ns)
1	81,947	31,194.0	31,843.3
2	82,610	30,776.4	31,678.5
3	93,451	23,055.3	29,269.6
4	83,525	32,208.4	32,180.7
5	89,545	23,277.0	30,386.1
6	93,529	33,106.8	35,219.8
7	96,428	26,391.8	35,907.6
8	120,627	43,366.2	183,217.8
9	127,878	61,707.9	217,202.4
10	112,377	90,744.8	229,803.6

Cuadro 3: Comparación entre diferentes enfoques de implementación

Aspecto	Implementación Manual	CUB	Thrust
Líneas de código (Aprox.)	150	20	10
Dificultad de desarrollo	Alta	Media	Baja
Rendimiento	Regular	Excelente	Bueno (hasta N=7)

Cuadro 4: Características de las matrices utilizadas

Matriz	**n**	**no zeros**	**Tipo**
A_matrix	24	81	No Simétrica
bcspwr02	59	167	Simétrica
cant	62.451	4.007.383	Simétrica
exdata_1	6.001	2.269.500	Simétrica
TSOPF_RS_b300_c2	28.338	2.943.887	No Simétrica
case39	40.216	1.042.160	Simétrica

Cuadro 5: Comparación de tiempos de ejecución

Matriz	**Serial (ms)**	**Paralelo (ms)**	**Speedup**	**Eficiencia**
Tiempo de parte paralelizada				
A_matrix	0.062	0.272	0.228x	-338.71 %
bcspwr02	0.062	0.275	0.225x	-343.55 %
cant	2.01	0.652	3.083x	67.54 %
exdata_1	0.208	0.476	0.437x	-128.85 %
TSOPF_RS_b300_c2_600	0.766	0.507	1.511x	33.82 %
case39	1.110	0.749	1.482x	32.61 %
Tiempo total de función				
A_matrix	0.203	0.307	0.66×	-51.23 %
bcspwr02	0.198	0.315	0.63×	-59.09 %
cant	6.308	5.061	1.25×	19.74 %
exdata_1	3.308	3.516	0.94×	-6.29 %
TSOPF_RS_b300_c2	1.458	1.105	1.31×	24.23 %
case39	1.431	0.921	1.55×	35.63 %

Listing 1: Implementacion de Exclusive Scan con CUB

```
#include <cub/cub.cuh>

// Determinar el tamanio de memoria temporal requerido
size_t temp_storage_bytes = 0;
cub::DeviceScan::ExclusiveScan( d_temp_storage, temp_storage_bytes,
     d_array, scan_array, cub::Sum(), INITIAL_VALUE, ARRAY_SIZE);

// Asignar memoria temporal
void *d_temp_storage = nullptr;
cudaMalloc(&d_temp_storage, temp_storage_bytes);

// Ejecutar exclusive scan
cub::DeviceScan::ExclusiveScan( d_temp_storage, temp_storage_bytes,
     d_array, scan_array, cub::Sum(), INITIAL_VALUE, ARRAY_SIZE);
```

Listing 2: Implementacion de Exclusive Scan con Thrust

```
#include <thrust/scan.h>
#include <thrust/device_vector.h>
```

```
thrust::device_vector<int> d_input(h_input.begin(), h_input.end());
thrust::device_vector<int> d_output(num_items + 1);

thrust::exclusive_scan(d_input.begin(), d_input.end(), d_output.begin(), 0);
```

Functor calculate_keys_functor:

Listing 3: Functor para cálculo de claves

```
struct calculate_keys_functor {
    const int* niveles;
    const int* RowPtrL;

    // Constructor para inicializar los punteros
    calculate_keys_functor(const int* _niveles, const int* _RowPtrL)
        : niveles(_niveles), RowPtrL(_RowPtrL) {}

    __host__ __device__
    int operator()(int idx) const {
        int level = niveles[idx] - 1; // igual al secuencial
        int nnz_row = RowPtrL[idx + 1] - RowPtrL[idx] - 1; // Elementos no
            cero en la parte triangular inferior
        int vect_size;

        // Clasificacion de la fila segun su numero de elementos no cero
            (nnz_row)
        if (nnz_row == 0)     // 0 elementos no cero
            vect_size = 6;
        else if (nnz_row == 1) // 1 elemento no cero
            vect_size = 0;
        else if (nnz_row <= 2) // Hasta 2 elementos no cero
            vect_size = 1;
        else if (nnz_row <= 4) // Hasta 4 elementos no cero
            vect_size = 2;    //
        else if (nnz_row <= 8) // Hasta 8 elementos no cero
            vect_size = 3;
        else if (nnz_row <= 16) // Hasta 16 elementos no cero
            vect_size = 4;
        else                  // Mas de 16 elementos no cero
            vect_size = 5;

        // Clave combinada: Se multiplica el nivel por 7 (ya que vect_size
            tiene 7 valores posibles, 0-6) y se suma vect_size para obtener
            una clave unica para cada combinacion de (nivel, tipo de fila).
        return level * 7 + vect_size;
    }
};
```

Functor calculate_warps_per_group:

Listing 4: Functor para cálculo de warps por grupo

```
struct calculate_warps_per_group {
    __host__ __device__
    int operator()(const thrust::tuple<int, int>& t) const {
        int combined_key = thrust::get<0>(t);
        int count = thrust::get<1>(t);
        int vect_size = combined_key % 7;
        int threads_per_row = (vect_size == 6) ? 0 : (1 << vect_size);
        int threads_this_group = count * threads_per_row;

        return (vect_size == 6) ?
            (count + 31) / 32 : // Cuando solo hay ceros en la fila asigno un
                warp
            (threads_this_group + 31) / 32; // Si hay elementos distintos de 0
                asigno ceil(threads_this_group/32)
    }
};
```