Laboratorio del curso GPGPU - Sort

Introducción

Ordenar una secuencia de datos es una tarea clave sobre la cual se construyen numerosos métodos computacionales. La importancia de esta operación ha llevado a su intenso estudio y al desarrollo de técnicas sumamente eficientes para realizarla en distintas plataformas, y en especial en plataformas masivamente paralelas.

En este trabajo deberán implementar dos algoritmos de ordenamiento altamente paralelizables: *radix sort* y *merge sort*.

Como paso previo a la elaboración del *radix sort*, deberá desarrollar una implementación eficiente de la suma prefija o *scan*, una de las operaciones fundamentales de muchos algoritmos paralelos. El patrón de cómputo *scan* se vio en la Clase 8: Patrones de Cómputo II del teórico[1].

La propuesta se basa en [**?**].

Radix Sort

La suma prefija puede ser usada como pieza fundamental de uno de los algoritmos de ordenamiento más eficientes para el caso de claves cortas en procesadores paralelos, el *radix sort*.

El algoritmo comienza considerando el primer bit de cada clave, empezando por el bit menos significativo. Utilizando este bit se particiona el conjunto de claves de forma que todas las claves que tengan un 0 en ese bit se ubiquen antes que las claves que tienen el bit en 1, manteniendo el orden relativo de las claves con mismo valor de bit. El resultado del precedimiento se muestra en la Figura **??**.

Una vez completado este paso se hace lo mismo para cada uno de los bits de la clave hasta completar todos los bits de la misma.

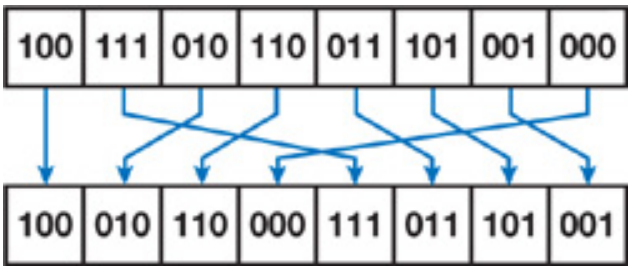

Figura 1: Primera pasada de radix sort para claves de 3 bits. Imagen extraída de [**?**].

Definimos la primitiva *split(array, b)* como la operación que ordena el arreglo de claves *array* de acuerdo al valor de su bit *b* de la forma descrita anteriormente.

Para implementar en GPU dicha primitiva se procederá de la siguiente manera:

[1] `https://eva.fing.edu.uy/pluginfile.php/210752/mod\_resource/content/6/Clase8\%20-\%20Patrones\%202\%20-\%202023.pdf`

- En un arreglo temporal setear en 1 todos los elementos correspondientes a claves falsas (b = 0) y 0 para las verdaderas.
- Computar la suma prefija del arreglo. Ahora cada posición del arreglo contiene la cantidad de claves falsas que hay antes que esa posición. Como en el arreglo de salida las claves falsas se ubican primero, en el caso de las posiciones correspondiente a claves falsas, esa cantidad es igual al índice de dichas claves en el arreglo de salida (al que llamamos f).
- El último elemento del arreglo de salida del scan contiene el total de claves falsas[2]. Almacenamos este valor en la variable `totalFalses`.
- Ahora se computa el índice de las claves verdaderas en el arreglo de salida. Si la clave está en la posición i, este índice será $t = i$ - f + *totalFalses*.
- Una vez obtenidos los índices anteriores se graban las claves en el arreglo de salida en la posición t o f dependiendo de si la clave es falsa o verdadera.

Para implementar el algoritmo de *radix sort* utilizando la primitiva *split* simplemente debe inicializarse una máscara binaria para aislar el bit menos significativo, realizar el *split* del arreglo según ese bit, comprobar si el arreglo ya está ordenado y, si no lo está, hacer un shift a la izquierda de la máscara y volver a iterar. El procedimiento anterior se ejemplifica en la Figura **??**.

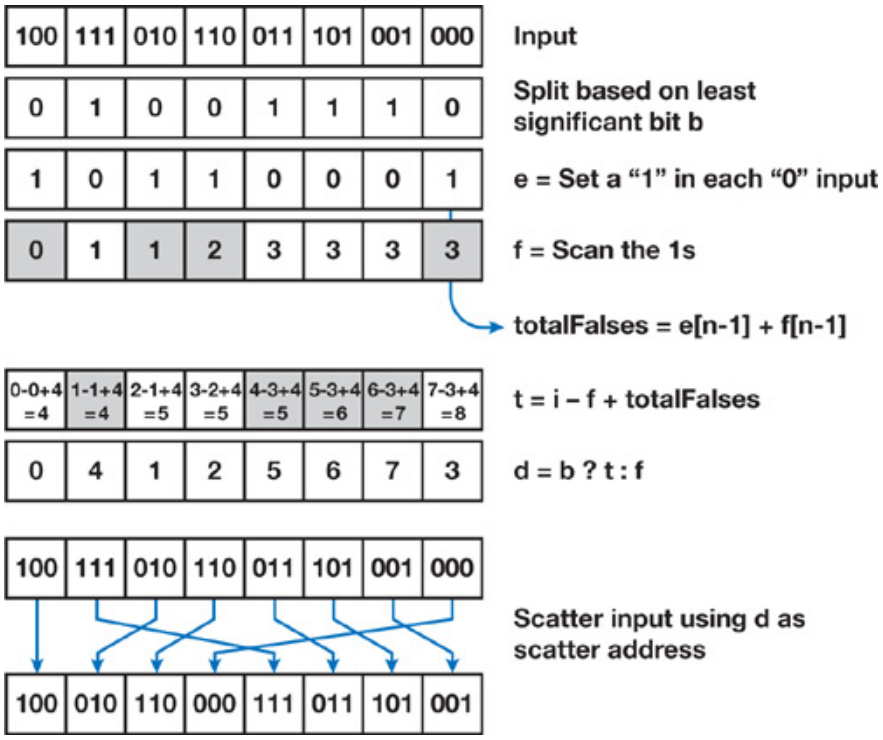

Figura 2: Ejemplo de la primitva split para claves de 3 bits. Imagen extraída de [**?**].

Merge Sort

Merge sort es un algoritmo de ordenamiento del estilo *divide-and-conquer* que divide el arreglo de claves en sub-arreglos que son ordenados independientemente. Una vez ordenados, estos sub-arreglos son las hojas

[2]Hay que sumar 1 a este valor si la última clave fue falsa.

de un "árbol de ordenamiento", que son unidos y ordenados dos a dos, iterativamente, hasta obtener un único arreglo ordenado.

Las fusiones dos a dos de dichos arreglos pueden ser realizadas en paralelo, por lo que este método de ordenamiento es adecuado para arquitecturas paralelas. Más aún, la posición de cada elemento de los sub-arreglos en el arreglo de salida en la fusión de dos sub-arreglos se puede computar eficientemente como se describe a continuación.

Dadas las secuencias ordenadas A y B, se desea obtener C = merge(A, B). Suponemos que las secuencias son lo suficientemente pequeñas (por ejemplo de largo $t = 256$) para ser unidas por un único bloque de $2t$ threads. Llamamos $r(x, X)$ (rango de x en X) a la posición que tendría x si se insertara ordenado en el arreglo X.

Para un elemento $a_i \in A$, $r(a_i, C)$ (la posición de un elemento de A en el arreglo ordenado C) es simplemente su posición en el arreglo A más la cantidad de elementos de B menores que A, es decir, $r(a_i, C) = i + r(a_i, B)$. Como B está ordenado la cantidad de elementos de B menores que a_i puede implementarse eficientemente como una búsqueda por bipartición. Lo mismo que fue descrito para los elementos en el arreglo A vale para los elementos del arreglo B.

Para unir arreglos más grandes que t, realizamos un paso previo para dividir ambos arreglos en trozos de tamaño menor o igual a t que pueden ser unidos independientemente.

Para lograr esto, primero construimos dos secuencias de "separadores", S_A y S_B, tomando los elementos correspondientes a los índices t,$2t$,$3t$,...,mt de A y B respectivamente. Una vez obtenidos los dos conjuntos de separadores los unimos utilizando la propia rutina merge para arreglos cortos, de forma que obtenemos $S = merge(S_A, S_B)$. Dado que la separación entre elementos de S_A y S_B es exactamente t, la separación entre elementos de S es a lo sumo $2t - 1$. De esta forma, S divide al arreglo de salida en particiones (de distinto largo) de a lo sumo $2t - 1$ elementos. Cada una de estas particiones será el merge de una partición de A con una partición de B de, a lo sumo, t elementos.

Resta saber qué elementos de los arreglos A y B involucran cada partición. Para ello debemos hallar, para cada elemento s_i de S, $r(s_i, A)$ y $r(s_i, B)$.

Sea $s \in S$. Suponiendo que $s \in A$ calcular el rango en A es trivial (es su posición en el arreglo A) así que resta calcular $r(s, B)$. Como B está ordenado, podemos obtener dicho rango realizando una búsqueda por bipartición nuevamente. Si tenemos en cuenta que $r(s, S_B) = r(s, S) - r(s, S_A)$, podemos realizar la búsqueda binaria restringida al *tile* de B delimitado por los elementos de S_B que "encierran" a s.

Una vez que obtenemos $rs_i^A = r(s_i, A)$ y $rs_i^B = r(s_i, B)$ para cada $s_i \in S$ sabemos que el segmento del arreglo C comprendido entre los índices $r(s_i, S)$ y $r(s_{i+1}, S)$ es el merge del segmento de A comprendido entre rs_i^A y rs_{i+1}^A y del segmento de B comprendido entre rs_i^B y rs_{i+1}^B, y que cada uno de estos segmentos es de largo inferior a t, por lo que pueden ser unidos usando la rutina para *tiles* de largo menor que t.

Consigna

El objetivo del laboratorio es realizar un programa que ordene un arreglo en GPU.
El ordenamiento consta de dos etapas:

1. Se divide el arreglo en *tiles* de 32 elementos y se ordenan los mismos de forma independiente utilizando radix sort y un warp para cada tile.

2. Una vez ordenados, se unen los *tiles* utilizando merge sort.

Para simplificar el código, puede asumirse que el arreglo siempre tiene largo igual a una potencia de 2, múltiplo de 32.

Parte a)

Implementar la suma prefija sobre un *tile* de datos de 32 elementos usando operaciones de warp shuffle y warp voting (any, all, ballot).

Parte b)

Utilizando el código de la parte anterior implemente un kernel que particione el arreglo de entrada en *tiles* de largo 32 y ordene cada tile de forma concurrente utilizando el radix sort y operaciones de warp shuffle y warp voting.

Parte c)

Implementar una función en GPU que realice el merge sort de dos arreglos de tamaño pequeño t utilizando un solo bloque y la memoria compartida.

Parte d)

Implemente el merge sort en GPU según el procedimiento que se describe en la letra.

Parte e)

Realizar un análisis del desempeño comparándolo con la función sort() de Thrust.

Entregar

1. Código fuente de la solución.
2. Un informe que contenga una explicación detallada de la solución, incluyendo optimizaciones que haya hecho y la discusión de los resultados experimentales.

Referencias

[1] Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[2] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.