Procesadores Gráficos y Aplicaciones en Tiempo Real

Tone Mapping en CUDA y Thrust

Álvaro Muñoz Fernández
Iván Velasco González

Índice

1. Introducción

El objetivo de esta práctica consiste en la implementación de varios de los pasos intermedios necesarios para la aplicación de la técnica de *Tone Mapping*. Para ello se realizarán las implementaciones de varios algoritmos paralelos tanto en código CUDA como mediante la librería Thrust.

2. Directivas de compilador

Para facilitar el desarrollo y la realización de pruebas se ha optado por utilizar directivas de preprocesador. Estas están definidas al inicio del fichero con comentarios que describen su comportamiento. Mediante estas directivas puede alternarse entre el uso de la implementación en CUDA o la implementación en Thrust, además de activar o desactivar el volcado de resultados por consola mediante la directiva DEBUG y alternar entre el uso de la implementación de MinMax con memoria compartida o sin ella.

3. Estructura del código

Todo el código escrito para la práctica se encuentra en el fichero *funcHDR.cu* proporcionado como material de apoyo. En este fichero hay definidas varias funciones que han sido completadas para cumplir los requisitos de la entrega y que se explican a continuación:

3.1. calculate_cdf()

Esta es la función principal del código de alumno que es llamada desde la función main() de la aplicación. En ella se realizan las inicializaciones de variables y llamadas a los diferentes kernels a emplear durante la ejecución de la práctica.

El contenido de esta función se ha dividido en dos bloques independientes, siendo compilado sólo uno de ellos en función del valor de la directiva THRUST, que indicará si debe utilizarse la implementación en CUDA o la implementación en Thrust.

Independientemente de la implementación utilizada, cada uno de los dos bloques está además subdividido en varios bloques, cada uno destinado al cálculo de uno de los pasos intermedios a implementar en la práctica. El primero corresponde al cálculo de los valores mínimos y máximos del buffer de luminancias. Después se calcula un histograma a partir de las luminancias y el número indicado de bins, utilizando la fórmula descrita en el enunciado de la práctica. Finalmente se calcula el *exclusive scan*, que es almacenado en la variable de salida *d_cdf*. El flujo de ejecución vuelve entonces a la función principal del programa.

3.2. parallelMinMax()

Este kernel se encarga únicamente de recorrer el buffer de luminancias buscando los valores mínimo y máximo, empleando para ello el método paralelo mediante comparación de pares propuesto en la asignatura, en el que se realizan varias iteraciones, empezando con un número de threads igual a la mitad de los elementos del buffer y reduciendo el número de threads a la mitad en cada iteración, hasta terminar de reducir por completo el buffer a un solo valor.

3.3. parallelMinMaxInit()

Este kernel auxiliar se emplea en la primera iteración para solucionar el caso inicial en el que deben tomarse valores directamente desde el buffer de luminancias y almacenarlos en los buffers auxiliares que se emplearán para realizar las operaciones de reducción. De esta forma no se altera el buffer de luminancias durante la ejecución de las operaciones de reducción, operándose en las futuras iteraciones exclusivamente sobre los arrays auxiliares.

3.4. sharedMinMax()

Kernel alternativo que se ha implementado utilizando memoria compartida para optimizar los tiempos de acceso a memoria, siguiendo un procedimiento similar al empleado en *ParallelMinMax*. En este caso, cada bloque de *Threads* realiza una reducción completa utilizando memoria compartida para almacenar los resultados intermedios, y en ultima instancia copiar el resultado de memoria compartida a la posición correspondiente , la cual es el numero de bloque, en el vector de memoria global.

Ademas, la implementación se ha pensado de tal forma que cada *Thread* carga y opera 2 elementos del vector, procesando, por tanto, el doble de elementos por bloque. Esta implementación tambien evita que existan *Threads* que solamente carguen valores de memoria pero que luego no computen la reducción en si, lo que provacaria una perdida de ocupancia, como sucede en el codigo de ejemplo visto en clase.

3.5. initSharedMinMax()

Kernel alternativo que inicializa y ejecuta la primera iteración del algoritmo paralelo de cálculo de mínimo y máximo utilizando memoria compartida.

3.6. histogram()

Este kernel se ejecutará en un número determinado de threads (se ha optado por utilizar un thread por cada fila de la imagen) que recorrerán una porción del buffer de luminancias (en nuestro caso recorrerán los valores de la fila que les corresponde) y calcularán el bin correspondiente a cada valor de luminancia encontrado, acumulando el contador correspondiente a dicho bin de forma atómica.

3.7. scanReduce()

El primer kernel utilizado el cálculo del *exclusive scan* paralelo, que se encarga de realizar sumas parciales y almacenarlas en el buffer mediante el algoritmo desarrollado por Blelloch. Para ello inicia la ejecución con N/2 threads, siendo N el número de elementos del buffer, y realizando sumas parciales que almacena de nuevo en el buffer. En cada iteración se divide el número de threads a la mitad hasta reducirse a 1, calculándose el *offset* y *stride* adecuados para cada iteración y almacenando los resultados de las sumas parciales de los elementos correspondientes.

3.8. scanReverse()

El segundo kernel del cálculo de *exclusive scan* recibirá los resultados obtenidos tras la ejecución del kernel previo, asignará un valor de 0 al último elemento del array e iniciará un proceso similar al del kernel *scanReduce()*, pero de forma inversa. Esto quiere decir que en lugar de empezar con N/2 threads empezará con uno, e irá duplicándolos hasta llegar a N/2 threads, usando los valores de *offset* y *stride* adecuados. Adicionalmente, tras calcular la suma parcial de cada pareja de elementos y almacenarlos en la posición

del último de los dos elementos, se almacenará el valor inicial del segundo elemento en la posición del primero. Al terminar este proceso se obtendrán los resultados acumulados del histograma.

4. Mejoras implementadas

4.1. Minimización de reserva de memoria

Para evitar overhead durante la ejecución se ha reducido al mínimo la reserva de memoria para las operaciones de los algoritmos. Sólo se ha reservado memoria para los buffers auxiliares de cálculo de mínimos y máximos, siendo en cada caso necesario un buffer de la mitad de elementos que el buffer de luminancias original. Para el resto de operaciones se utilizan los buffers ya existentes, operándose el histograma y el *exclusive scan* directamente sobre el buffer de salida, *c_cdf*.

4.2. Uso de memoria compartida

Se ha añadido una implementación alternativa de cálculo de mínimos y máximos que hace uso de memoria compartida. Con esta implementación se han visto mejoras sustanciales respecto a la implementación original, reduciéndose el tiempo de ejecución para la imagen *waterfall_bg.exr* de unos 2.45ms a 1.62ms.

4.3. Uso de la librería Thrust

Se ha realizado una implementación alternativa usando la librería Thrust. No ha sido posible implementar el cálculo del histograma mediante la librería ya que, a pesar de haber buscado ejemplos en la web, el único que parecía factible se ha implementado pero no ha funcionado correctamente, a pesar de haber utilizado el código de ejemplo disponible en el repositorio de CUDA. El código adaptado del ejemplo se ha mantenido en el fichero de código fuente, aunque se ha comentado y se ha añadido en su lugar el kernel de CUDA desarrollado para la práctica.

El resto de apartados se han implementado mediante las funciones disponibles en Thrust, procurando no realizar copias innecesarias de memoria haciendo una conversión de los punteros de CUDA a los *wrappers* definidos por la librería Thrust.

Adicionalmente se han realizado 10 ejecuciones consecutivas de la aplicación sobre la imagen *waterfall_bg.exr* tanto con la implementación en CUDA como en Thrust, obteniéndose como resultado una media de tiempo de ejecución de 1.22ms utilizando Thrust y una media de 1.62ms con nuestra implementación en CUDA. Estos resultados indican que la implementación en CUDA de cálculo de *min/max* y de *exclusive scan* tiene todavía margen de optimización.

4.4. Tamaño de bloque

Después de una breve investigación en cuanto a implementaciones de este tipo de algoritmos, la conclusión a la que llegamos es que utilizaban bloques de una sola dimensión, por lo que así se han implementado en la práctica. Los bloques tendrán un número de threads múltiplo del tamaño de *warp*, por tanto serán múltiplo de 32. Hemos visto una ligera mejora, aunque casi imperceptible, en los tiempos de ejecución utilizando bloques de 132 threads respecto a otros valores, por lo que nos hemos decidido por ese valor.

5. Pruebas locales

Además de las pruebas descritas en el apartado de la librería Thrust, se han realizado pruebas sobre las imágenes de ejemplo para comprobar el correcto resultado de la ejecución. La coincidencia con las imágenes de ejemplo ha sido perfecta, a excepción de la imagen *memorial_raw_large*, que tras ser procesada se almacena con unas dimensiones incorrectas. El problema parece provenir del propio código de apoyo, por lo que hemos tomado por válida nuestra implementación en función al resultado de las otras imágenes de ejemplo.

6. Valoración personal y dificultades encontradas

La práctica ha sido intuitiva y no ha tenido una carga de trabajo alta, pero sí que han surgido algunos problemas durante la implementación. El primero de ellos, que en el enunciado se pedía una implementacion de *exclusive scan* pero se proporcionaba como ejemplo el resultado de un *inclusive scan*, lo que ha dado lugar a confusiones que sólo se han disuelto al terminar la práctica y poder comprobar los resultados con las imágenes de referencia.

Además el cálculo de bins no especifica cómo tratar el último elemento, que si se aplica la fórmula tal cual se define en el enunciado queda excluido de cualquier bin, ya que tomará un valor de bin de $1 * numBins$, siendo este un índice fuera del buffer del histograma. Para solucionarlo hemos tomado como valor de bin el $min(bin, numBins - 1)$.

En cuanto a la implementación del método de Blelloch, las diapositivas, aunque daban la información necesaria, no eran del todo descriptivas. Las imágenes de ejemplo podían dar lugar a confusiones ya que había que entender cómo interpretar el orden de cada iteración, además de que en el paso de *reverse* no queda suficientemente claro que los valores del segundo elemento de cada par deben copiarse después en la posición del primer elemento.