PRÁCTICA TGA

CUDA 0006

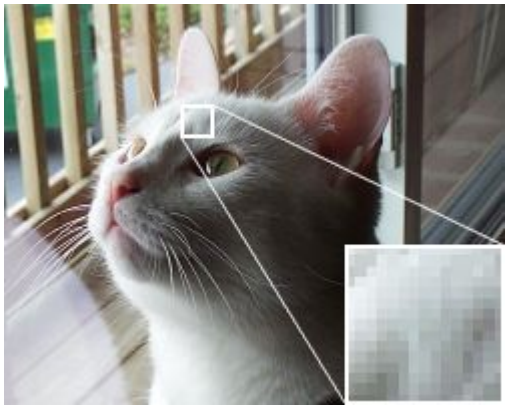
Raül Montoya Pérez
Martí Ferret Vivo

Índice

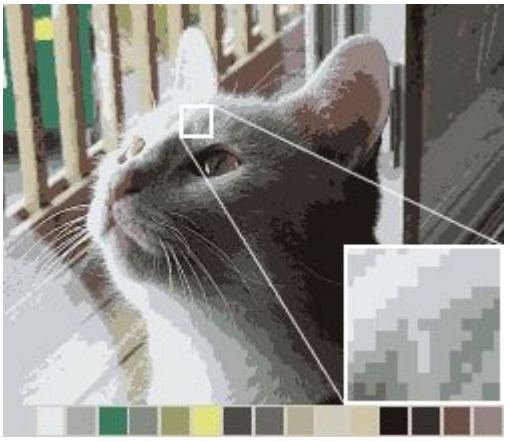
Introducción

El programa que vamos a paralelizar con CUDA es un filtro para imágenes, se llama reducción de colores (color quantization en Inglés). Lo que hace, como su nombre indica, es reducir el número de colores existentes en una imágen a un determinado valor. Su uso es muy común para reducir el tamaño del archivo, aunque para lo que más se usa hoy en día es para obtener una nueva imágen para imprimir con menos colores.

El funcionamiento del filtro es encontrar la paleta de colores (con un número de ellos preestablecido) que más se acerquen a los colores de la imágen original. Como se ve en las imágenes siguientes, se ha reducido el número de colores a 16 con la paleta precalculada.



Imágen original del gato.



Imágen después de aplicar una reducción a 16 colores con la paleta mostrada.

Como se puede apreciar se ha perdido calidad y detalles de la imágen, sin embargo se puede identificar perfectamente el gato y su fondo.

Complejidad

Partiendo de una imágen original (con 255^3 colores posibles) y una paleta precalculada (por ejemplo una de 16 colores) aplicar el filtro es una tarea trivial y paralelizable por cada pixel. Simplemente se interpreta cada píxel como un punto en un espacio 3D donde sus coordenadas son sus 3 colores RGB, entonces se compara con la distancia euclidiana con los 16 colores de la paleta y se pinta el píxel con el color más cercano (distancia más pequeña).

La complejidad real está en calcular los colores finales de la paleta, para ello existen diferentes algoritmos, nosotros hemos optado por implementar el famoso k-means. Lo que hace es calcular las k medias de un conjunto de números o de puntos en un espacio iterando un número fijado de veces.

Identificación zonas paralelas y 1a versión

En el código secuencial hemos podido observar 4 partes diferentes, las 3 primeras forman parte de la ejecución de k-means. La primera es donde se busca para cada color, a cual de los píxeles de la imagen que se han tomado como referencia son asignados. La segunda parte corresponde a, para cada píxel de la imagen cogido como referencia, calcular la suma de los colores de los píxeles asignados a esos píxeles cogidos como referencia y contar cuantos hay. Como tercera parte tenemos la ejecución final de k-means donde se divide la suma total de cada píxel entre el contador de cada uno obteniendo de esta forma la media de los colores de los píxeles asignados a los que se han tomado como referencia. Finalmente, la última parte corresponde a asignar a cada píxel de la imagen, la media calculada del pixel que se ha tomado como referencia asignado a este.

Por lo tanto, en nuestro código CUDA hemos hecho 4 kernels uno para cada una de las partes. Los tres primeros que forman parte de la ejecución de k-means están dentro de un bucle para cada iteración, el último está fuera ya que es el de asignación de colores a la imagen.

La primera queda de la siguiente manera:

```
//executem k means:
int it;
for (it = 0; it < N_iterations; ++it) {

    //set counts and new_means to 0
    cudaMemset (counts, 0, nBlocks * sizeof (int) * N_colors);
    cudaMemset (new_means, 0, nBlocks * sizeof (Color) * N_colors);

    //for each pixel find the best mean.
    find_best_mean_par<<<dimGrid, dimBlock>>>(means_device, assigns, im_device, Size, N_colors, Size_row);

    cudaDeviceSynchronize();
```

Primero se inicializan a 0 el contador y las medianas que se calcularán después. Se llama al primer kernel para buscar las asignaciones con tamaño de la grilla que corresponde al número de bloques que es (Size + nThreads - 1)/nThreads, donde Size es el tamaño de la imagen (ancho * alto) y nThreads es 1024, y con tamaño de bloque de 1024. Posteriormente se llama a cudaDeviceSynchronize() ya que el siguiente kernel tiene que esperar al primero para poder empezar.

La segunda parte consiste en una reducción por lo que necesitaremos utilizar memoria dinámica compartida, ya que el tamaño de ésta memoria dependerá del número de colores introducido por el usuario. Pero al lanzar el kernel observamos que la memoria compartida sobrepasaba el máximo de memoria permitido por lo que teníamos que reducirla. Como los elementos que se tienen que reducir son un array de colores y un array de contadores, separamos su reducción en 2 kernels, uno para el contador y el otro para los colores, pero a

pesar de haber reducido la memoria compartida seguía sobrepasando el límite. Así que hemos optado por modificar el kernel de los colores para que solo haga la reducción de un color de modo que podemos lanzarlo 3 veces, uno para cada color (r,g,b). De esta forma si que conseguimos que la memoria compartida de forma dinámica no sobrepase el límite aunque entonces el número máximo de colores introducidos por el usuario no puede ser superior de 12 ya que tanto el array de contadores como el de las sumas dependen del número de colores. El tamaño de la memoria compartida es N_colors*1024 * sizeof(unsigned int), ya que contiene un valor por cada thread de cada uno de los píxeles escogidos como referencia. Entonces la invocación de los dos kernels queda de la siguiente manera:

```
matrix_reduction_count<<<dimGrid, dimBlock, shared_memory_size>>>(counts, assigns, im_device, Size_row, Size, N_colors);
matrix_reduction_color<<<dimGrid, dimBlock, shared_memory_size>>>(new_means, assigns, im_device, Size_row, Size, N_colors, 2);
matrix_reduction_color<<<dimGrid, dimBlock, shared_memory_size>>>(new_means, assigns, im_device, Size_row, Size, N_colors, 1);
matrix_reduction_color<<<dimGrid, dimBlock, shared_memory_size>>>(new_means, assigns, im_device, Size_row, Size, N_colors, 0);

cudaDeviceSynchronize();

cudaMemcpy(means_host_red, new_means, nBlocks * N_colors * sizeof(Color), cudaMemcpyDeviceToHost);
cudaMemcpy(counts_host_red, counts, nBlocks * N_colors * sizeof(int), cudaMemcpyDeviceToHost);

memset(counts_host, 0, sizeof (int) * N_colors);
memset(means_host, 0, sizeof (Color) * N_colors);

int i, j;
for (i = 0; i < nBlocks; ++i) {
    for (j = 0; j < N_colors; ++j) {
        counts_host[j] += counts_host_red[i*N_colors + j];
        means_host[j].r += means_host_red[i*N_colors + j].r;
        means_host[j].g += means_host_red[i*N_colors + j].g;
        means_host[j].b += means_host_red[i*N_colors + j].b;
    }
}

cudaMemcpy(new_means, means_host, N_colors * sizeof(Color), cudaMemcpyHostToDevice);
cudaMemcpy(counts, counts_host, N_colors * sizeof(int), cudaMemcpyHostToDevice);
```

Y la implementación del kernel:

```
__global__ void matrix_reduction_count(int counts[], int assigns[], unsigned char *im, int Size_row, int Size, int N_colors) {
    extern __shared__ unsigned int shared[];

    unsigned int tid = threadIdx.x;
    unsigned int id = blockIdx.x*blockDim.x + threadIdx.x;

    //init shared
    for (int j = 0; j < N_colors; ++j) {

        if (j == assigns[id]) {
            shared[tid*N_colors + j] = 1;
        }
        else {
            shared[tid*N_colors + j] = 0;
        }
    }
```

```
    //reduccio
    unsigned int s;
    for(s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            for (int j = 0; j < N_colors; ++j) {
                shared[tid*N_colors + j] += shared[(tid + s)*N_colors + j];
            }
        }
        __syncthreads();
    }

    //copiar valors:
    if (tid == 0) {
       for (int j = 0; j < N_colors; ++j) {
            counts[blockIdx.x*N_colors + j] = shared[j];
        }
    }
}
```

Una vez calculados los arrays de sumas y contadores parciales, para calcular el resultado final de los arrays hay que hacer la suma final de cada bloque para cada uno de los píxeles cogidos como referencia. Lo haremos en la CPU en esta versión. Para ello copiamos los datos parciales del device al host, calculamos, y luego los volvemos a copiar al device, tal como muestra la primera imagen.

La tercera parte consiste en calcular las medias de las sumas calculadas anteriormente. Esto es simple y se puede hacer con un solo kernel:

```
//Divide sums by counts to get new centroids.
divide_sums_by_counts_par<<<dimGridMeans, dimBlock>>>(means_device, N_colors, new_means, counts);

cudaDeviceSynchronize();
```

Aquí lo único que cambia es la dimensión de la grilla ya que ahora no tendremos todos los píxeles de la imagen sino que solo tendremos los píxeles cogidos como referencia, por lo tanto, nBlocks = (N_colors + nThreads - 1)/nThreads;

Finalmente, la última parte también es sencilla y solo tenemos que copiar los datos a la imagen:

```
//assignem colors:
assign_colors_par<<<dimGrid, dimBlock>>>(means_device, assigns, im_device, Size_row, Size);

//copy to host:
cudaMemcpy(im_host, im_device, infoHeader.imgsize * sizeof(unsigned char), cudaMemcpyDeviceToHost);
```

Ahora si que la dimensión de la grilla depende de todos los píxeles y será como en las dos primeras partes. Después de llamar al kernel, copiamos los datos del device al host para mostrar la imagen y sus propiedades.

2a versión

En esta versión lo que queremos hacer es antes de hacer la reducción de las sumas y contadores parciales en la CPU, llamar a un kernel para que lo haga de manera que la CPU no haga tanto trabajo. Para ello hemos hecho dos nuevos kernels uno para el array contadores y el otro para las sumas (para un color que podrá ser llamado por los tres colores) y hemos añadido las llamadas de la siguiente manera:

```
//volmemos a hacer otra reduccion
matrix_reduction_count_2<<<nBlocks/nThreads, dimBlock, shared_memory_size>>>(counts_2, counts, Size_row, Size, N_colors);
matrix_reduction_color_2<<<nBlocks/nThreads, dimBlock, shared_memory_size>>>(new_means_2, new_means, Size_row, Size, N_colors, 2);
matrix_reduction_color_2<<<nBlocks/nThreads, dimBlock, shared_memory_size>>>(new_means_2, new_means, Size_row, Size, N_colors, 1);
matrix_reduction_color_2<<<nBlocks/nThreads, dimBlock, shared_memory_size>>>(new_means_2, new_means, Size_row, Size, N_colors, 0);

cudaDeviceSynchronize();
```

Lo único que cambia respecto al anterior es que se añaden estas nuevas llamadas con nBlocks/Threads y estos kernels difieren de los anteriores solo en la inicialización de la memoria compartida, solo tenemos que hacer lo siguiente para inicializar:

```
//init shared
for (int j = 0; j < N_colors; ++j) {

    shared[tid*N_colors + j] = counts[id*N_colors + j];
}
```

De esta forma pasamos a tener un bucle de reducción en la CPU de nBlocks a nBlocks/nThreads.

3a versión

En esta versión queremos mejorar el patrón de acceso a los datos porque los kernels que hemos implementado en la versión anterior utilizan los threads de forma muy ineficiente, ya que primero trabajan todos los threads, luego los Threads pares, luego los threads múltiplos de 4… Esto hace que se muy ineficiente ya que CUDA ejecuta los threads por warps (en grupos de 32) y a partir de la tercera iteración en cada warp se ejecutan pocos threads.

Así pues lo que hacemos es cambiar el patrón de acceso para que los datos sean accedidos en posiciones consecutivas de memoria. Para ello solo hace falta cambiar la parte de reducción de cada kernel por la siguiente:

```
//reduccio
unsigned int s;
for(s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        for (int j = 0; j < N_colors; ++j) {
            shared[tid*N_colors + j] += shared[(tid + s)*N_colors + j];
        }
    }
    __syncthreads();
}
```

Con este cambio hemos tenido buenas ganancias respecto a las versiones anteriores, se comentará en la parte de resultados.

4a versión

En la versión anterior el trabajo que hacen los threads no es mucho ya que la mitad de los threads en la 1a iteración ya no hacen nada. Lo que vamos a hacer es ejecutar la mitad de thread blocks y que cada block procese 2 * nThreads elementos. Todos los threads harán por lo menos 1 suma parcial. Para esta versión lo vamos a hacer para la parte pequeña de la reducción, es decir, los kernels que añadimos en la segunda versión, ya que es bastante complicado hacerlo para todos los kernels en nuestro caso.

Así pues lo que hacemos es llamar al kernel con la mitad de thread blocks y cambiar la inicialización de la siguiente manera:

```
extern __shared__ unsigned int shared[];

unsigned int tid = threadIdx.x;
unsigned int id = blockIdx.x*(blockDim.x * 2) + threadIdx.x;

//init shared
for (int j = 0; j < N_colors; ++j) {

    shared[tid*N_colors + j] = counts[id*N_colors + j] + counts[((id + blockDim.x) * N_colors) + j];
}
```

Para el kernel de la suma de colores es realizar lo mismo que en el de contadores.

```
extern __shared__ unsigned int shared[];

unsigned int tid = threadIdx.x;
unsigned int id = blockIdx.x*(blockDim.x * 2) + threadIdx.x;

//init shared
for (int j = 0; j < N_colors; ++j) {

    if (offset == 2)        shared[tid*N_colors + j] = new_means[id*N_colors + j].r + new_means[(id + blockDim.x) *N_colors + j].r;
    else if (offset == 1)   shared[tid*N_colors + j] = new_means[id*N_colors + j].g + new_means[(id + blockDim.x) * N_colors + j].g;
    else                    shared[tid*N_colors + j] = new_means[id*N_colors + j].b + new_means[(id + blockDim.x) *N_colors + j].b;
}
```

5a versión

En esta versión hemos intentado aplicar lo mismo que en la versión 4 pero para los kernels “grandes”. Hemos tenido muchos problemas con esta versión y al final no hemos conseguido proyectar la imagen de forma correcta ya que sale negra, pero nos ha servido para ver el tiempo que podríamos haber ganado respecto la versión anterior ya que la ejecución la hace bien pero coje valores raros en las sumas y contadores durante la ejecución..

Versión 0

Esta versión la hemos llamado 0 ya que no parte de las versiones anteriores sino que parte del código inicial. Lo que hemos hecho es en lugar de utilizar memoria dinámica compartida, hemos utilizado “AtomicAdd” para hacer las reducciones. Hemos juntado las dos primeras partes en un solo kernel que busca las asignaciones y lleva las sumas y contadores también. Así pues, el código de las llamas kernel queda de la forma siguiente:

```
//set counts and new_means to 0
cudaMemset (counts, 0, nBlocks * sizeof (int) * N_colors);
cudaMemset (new_means, 0, nBlocks * sizeof (Color) * N_colors);

//for each pixel find the best mean and sum up.
findandsum<<<dimGrid, dimBlock>>>(means_device,new_means, assigns, im_device, counts, Size_row, Size, N_colors);
cudaDeviceSynchronize();

//Divide sums by counts to get new centroids.
divide_sums_by_counts_par<<<dimGridMeans, dimBlock>>>(means_device, N_colors, new_means, counts);

cudaDeviceSynchronize();
```

En la siguiente página se muestra el código del kernel, con los respectivos AtomicAdd para cada array.

```
__global__ void findandsum(Color means[],Color new_means[], int assigns[], unsigned char *im, int counts[],
    int Size_row, int Size, int ncolors) {
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < Size) {
                int j;
        int index = (id*3/Size_row) * Size_row + ((id*3)%Size_row);
        int dist_min = -1;
        int dist_act, assign;
        for (j = 0; j < ncolors; ++j) {
            dist_act = (im[index+2] - means[j].r)*(im[index+2] - means[j].r) + (im[index+1] - means[j].g)*(im[index+1] - mean
            if (dist_min == -1 || dist_act < dist_min) {
                dist_min = dist_act;
                assign = j;
            }
        }
        assigns[id] = assign;

        atomicAdd(&new_means[assign].r, im[index+2]);
        atomicAdd(&new_means[assign].g, im[index+1]);
        atomicAdd(&new_means[assign].b, im[index]);
        atomicAdd(&counts[assign], 1);
    }

}
```

Esta posible solución la consideramos al principio y a resultado funcionar bastante bien. En el apartado siguiente se analizarán los resultados obtenidos.

Resultados y conclusiones

Para obtener los resultados hemos ejecutado todas las versiones varias veces cambiando el número de colores entre 10 y 12. Los resultados obtenidos para cada una de las versiones y en cada escenario contemplado son los siguientes:

Tiempo de ejecución (ms)		
Versión \ N_colores	**10 colores**	**12 colores**
Versión Secuencial	16000 ms	18500 ms
Versión inicial	761 ms	1087 ms
Versión 2	760 ms	1083 ms
Versión 3	591 ms	823 ms
Versión 4	588 ms	824 ms
Versión 5	137 ms	140 ms
Versión 0	386 ms	351 ms

Así pues, vemos claramente que las versiones han ido mejorando en tiempo respecto la versión secuencial. Desde el principio hemos pensado que ganaríamos bastante tiempo si paralizábamos el código ya que detectamos las partes explicadas en los apartados anteriores.

Se puede ver también que las versiones 2 y 4 no aportan mejoras respecto las versiones inicial y 3. Esto es debido a que en la versión 2 el patrón de acceso es el mismo que en la inicial, y ya hemos visto que no hace un uso eficiente de los threads. La única mejora que se le añade es la de calcular una parte de la reducción en la GPU en lugar de la CPU, pero el efecto que tiene en relación a la ineficiencia del patrón de acceso es casi negligible.
En el caso de la versión 4, aplicamos la mejora solo en la reducción que hemos pasado a hacer desde la CPU a la GPU en la versión 2, por lo tanto, es una pequeña parte solo y no se nota esa ganancia. En cambió al aplicarla con toda la reducción si que se ve la ganancia respecto las versiones anteriores.

Hay que recordar que la versión 5 no nos funciona correctamente en el sentido de que imprime la imagen negra debido a que se accede mal en algún momento en algunas posiciones de memoria pero el tiempo obtenido nos da una referencia sobre el tiempo que podríamos haber obtenido en el caso de que funcionara bien.

Por otro lado, la versión 0, aún siendo la versión donde menos cambios se han hecho, ofrece unos muy buenos resultados, aunque se ha visto superada por la versión 5. Esto es porque el uso de AtomicAdd da un buen rendimiento pero se hacen muchos locks y por lo tanto, dificulta el paralelismo. Por esa razón, a medida que hemos ido mejorando las versiones hemos ido ganando tiempo y al final superando el tiempo obtenido en la versión 5.

A continuación se muestra un resumen de los Speedups obtenidos respecto la versión secuencial:

Speedup		
Versión \ N_colores	**10 colores**	**12 colores**
Versión Secuencial	16000 ms	18500 ms
Versión inicial	21	17
Versión 2	21	17
Versión 3	27	22.5
Versión 4	27.2	22.5
Versión 5	116.8	132.1
Versión 0	41.5	52.7

Por lo tanto, hemos conseguido un speedup de 116.8 respecto la versión secuencial con 10 colores, y un speedup de 132.1 con 12 colores. Repetimos que la versión 5 no nos funciona correctamente pero sí que representa el tiempo que podríamos haber obtenido. De esta forma vemos claramente el potencial del paralelismo y el saber como cómo organizar un programa y como acceder a los datos de forma eficiente.

Cabe destacar que seguramente podríamos haber obtenido un speedup mayor ya que a partir de la versión 5 se podían haber aplicado algunas mejoras como mejorar el rendimiento ejecutando el último warp sin hacer una sincronización ya que no hace falta esperar a nadie más si ya es el último warp.

Para acabar, tenemos que decir que adaptar el código de c++ a CUDA nos ha sido un poco difícil al principio pero cada vez hemos ido cojiendo práctica a lo largo de las versiones. Nos ha sabido mal también el hecho de que no funcionara la versión 5 ya que ofrece una gran mejora respecto a las versiones anteriores pero no sabemos donde estamos accediendo mal, la reducción que se hace en este algoritmo es un poco más difícil de lo que hemos hecho en clase y junto con pasar el código de c++ a CUDA han sido las partes donde hemos tenido más problemas y más difíciles nos han parecido.

Bibliografía

Color quantization - Wikipedia
https://en.wikipedia.org/wiki/Color_quantization

Shared memory CUDA:
https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

Stackoverflow:
https://stackoverflow.com/

Transparencias de la asignatura