Targetas Gráficas y Aceleradores - Proyecto final

Ecualización de imágenes a color con CUDA

Albert Ruiz Vives
Xavier Bernat López

Enero de 2024

Índice

1. Definición del problema

En este proyecto, hemos programado un ecualizador de imágenes a color en CUDA. La ecualización de imágenes consiste en la creación de un histograma (o más) a partir de la propia imagen y la redistribución del mismo con el objetivo de que la distribución de los colores sea uniforme, es decir, que el rango de colores pase de ser [min..max] a ser [0..255]. Este proceso se suele hacer en imágenes en blanco y negro y consta de 2 etapas distintas:

La primera etapa es la creación del histograma. Básicamente, se recorren todos los píxeles de la imagen (que, como está en blanco y negro, los 3 canales tienen el mismo valor $v \in [0, 255]$ para cada píxel p_i de la misma) y se crea un histograma que guarde, para cada tonalidad de gris, el número de veces que aparece en la imagen.

Una vez creado el histograma, se procede a la ecualización propiamente dicha. En esta segunda fase, se calcula la probabilidad acumulada de cada color, así como los colores máximo y mínimo de la imagen. A continuación, se modifica el color de cada píxel de la imagen siguiendo la fórmula:

$$Color_i = \frac{F_i - F_{minHisto}}{n_{pixeles} - F_{minHisto}} \times 255 \tag{1}$$

Donde F es la frecuencia acumulada del color, $n_{pixeles}$ es el número de píxeles de la imagen y $minHisto$ es el color más pequeño que hay en el histograma con frecuencia distinta de 0.

Una vez ecualizado el histograma y actualizados los colores, obtenemos la imagen completamente ecualizada. Esta técnica no es nueva, pues lleva existiendo mucho tiempo. Pero siempre se hace con imágenes en blanco y negro, con las limitaciones que ello conlleva.

Qué pasaría, pues, si la imagen estuviera a color? En ese caso, el número de posibilidades aumenta, ya que pasamos de tener un histograma a tener múltiples (si es que así lo deseamos), ya que ahora contamos con 3 canales (R, G y B). En este proyecto, hemos implementado 2 propuestas distintas de ecualizar imágenes a color, para ver si los resultados son realmente convincentes o no. Además, los hemos implementado en CUDA, ya que el procesado de imágnees es un problema que típicamente se puede solucionar mucho más rápido con targetas gráficas.

Lo primero que hemos implementado es la versión en blanco y negro, para habituarnos a la ecualización y al tratado de imágenes en C.

La implementación del Blanco y Negro (*filtrarBW.cu* en los archivos del proyecto) la hemos hecho de la siguiente manera: Primero, recorremos todos los píxeles de la imagen cambiando el color de los mismos a uno en escala de grises

y actualizando el elemento del histograma relativo al color resultante del píxel en cada iteración. Al prinicipio hacíamos esto con la media de R, G y B, pero encontramos otro método que era mucho mejor porque tiene en cuenta la luminosidad de los colores al pasarlos a Blanco y Negro. La fórmula para calcular el color es la siguiente:

$$Color = 0{,}2126 \times R + 0{,}7152 \times G + 0{,}0722 \times B \tag{2}$$

Donde R, G y B son los valores de esos canales (entre 0 y 255 respectivamente).

Una vez hecho el histograma, calculamos la distribución acumulada de frecuencias del histograma y actualizamos los colores de la imagen basándonos en la fórmula anteriormente descrita. Hay que destacar un cambio que hemos realizado a esa fórmula, que ha sido el pasar de aritmética en coma flotante a aritmética de enteros. El objetivo de este cambio ha sido ahorrar casts a float (que son muy costosos) y reducir el coste de las operaciones (pues las operaciones de coma fija son mucho más eficientes que las de coma flotante) a cambio de un poco de pérdida de precisión (que tampoco es crítica en nuestro problema). A continuación ponemos la fórmula para calcular el color de los píxeles, la distribución acumulada para cada color y el color ecualizado de los píxeles:

$$Color = \frac{2126 \times R + 7152 \times G + 0722 \times B}{10000} \tag{3}$$

$$F_i = F_{i-1} + PREC * \frac{Histograma(i)}{n_{pixeles}} \quad \text{donde} \quad 1 \leq i < 256 \tag{4}$$

$$Color_i = \frac{F_{p_i} \times (max(F) - min(F)) + min(F) * PREC}{PREC} \quad \text{donde} \quad 0 \leq i < n_{pixeles} \times 3 \tag{5}$$

Donde PREC es la precisión deseada. A continuación, el código que implementa estas funciones:

```
int min = minHisto(histogramBW);
  int max = maxHisto(histogramBW);
  int maxmin = max - min;
  unsigned int prob[256];
  int length = 256;
  prob[0] = PREC * histogramBW[0] / (width*height);
  //Calculamos los valores de las probabilidades acumuladas
  for(int i = 1; i < length; ++i){
    prob[i] = prob[i-1] + PREC * histogramBW[i]/(width*
   height);
```

```
  }
  //Ecualizamos
  for(int i = 0; i < width * height * 3; i=i+3){
    unsigned char color = (prob[image[i]] * maxmin + min *
    PREC) / PREC;
    image[i] = color;
    image[i+1] = color;
    image[i+2] = color;
  }
}
```

Listing 1: Cálculo de F y ecualización del histograma

Una vez hecha esta versión básica, hemos hecho las otras 2, que ecualizan imágenes a color. Las dos propuestas han sido:

- Crear 3 histogramas (el de R, el de G y el de B) y ecualizarlos por separado.
- Coger los 4 bits más altos de cada canal, unirlos en un sólo número de 12 bits, hacer el histograma en base a esos valores y ecualizarlo, para después actualizar solo los 4 bits más altos de cada color.

A continuación detallamos ambas implementaciones.

1.1. ColorSplit

Esta versión genera 3 histogramas a partir de la imagen, uno para cada canal, y los ecualiza por separado, actuualizando el color de la imagen. Esto se hace, de nuevo, con dos bucles for, uno para la creación de los histogramas (esta vez no modificamos la imagen ya que la queremos a color) y otro que se encarga de aplicar la ecuacion 5 a los 3 canales de cada píxel por separado. Cabe destacar que también calculamos por separado las distribuciones acumuladas para cada canal.

1.2. BITS

Esta versión es un poco más interesante a nivel de implementación. En este caso, en vez de hacer 3 histogramas, creamos uno solo, pero no para un rango de colores normales, sino que concatenamos los 4 bits más altos de cada canal en un solo número y ese número es el que usamos para montar el histograma y ecualizarlo.

Lo primero que necesitamos hacer en este programa es calcular el valor resultante de "mezclar"los 3 canales como hemos descrito arriba. Esto lo hacemos de la siguiente manera:

$$\text{Bits}_{i_c} = \text{Color}_{i_c} \gg 4 \quad \text{donde} \quad i \in [0, \text{width} \times \text{height}] \text{ es el pixel y } c \in \{R, G, B\} \text{ es el canal.} \tag{6}$$

$$\text{Valor}_i = (\text{Bits}_{i_R} \ll 8) | (\text{Bits}_{i_B} \ll 4) | (\text{Bits}_{i_B}) \quad \text{donde} \quad i \in [0, \text{width} \times \text{height}] \text{ es el pixel.} \tag{7}$$

Adjuntamos el código de esta operación:

```
for(int i=0;i<width*height*3;i=i+3){
    unsigned char colorR = image[i];
    unsigned char colorG = image[i+1];
    unsigned char colorB = image[i+2];
    colorR = colorR >> 4;
    colorG = colorG >> 4;
    colorB = colorB >> 4;
    unsigned int index = (colorR << 8) | (colorG << 4) | (
    colorB);
    histogram[index]++;
  }
}
```

Listing 2: Cálculo del histograma y de los valores por píxel

Y con esto, calculamos el histograma (que tiene que ser de 4096 elementos, ya que los valores resultantes de concatenar los bits altos tienen $4 \times 3 = 12$ bits, con lo que el rango de valores posibles, al ser unsigned, es de $2^{12} = 4096$, esto va a ser importante cuando analicemos la versión CUDA de este método). Cabe destacar también que, al tener ahora valores de 12 bits, no podemos almacenarlos como `unsigned char`, como están almacenados los valores de la imagen, pues los `char` tienen un tamaño de 8 bits, con lo que necesitamos guardarlos como `unsigned int`.

Una vez tenemos el histograma ya calculado, podemos calcular la F y ecualizarlo. Hay que recordar que, cuando decidamos el valor ecualizado que debemos asignarle a un píxel, debemos volver a separar los bits de cada canal del valor y sustituir los bits altos de los canales del píxel por los nuevos bits. Esto lo haremos siguiendo las fórmulas a continuación:

$$\text{BitsNuevos}_{i_R} = (\text{Valor'}_i \gg 8) \quad \text{donde} \quad i \in [0, \text{width} \times \text{height}] \text{ es el pixel.} \tag{8}$$

$$\text{BitsNuevos}_{i_G} = (\text{Valor'}_i \gg 4)\,\&\,0b000000001111 \tag{9}$$

$$\text{BitsNuevos}_{i_B} = \text{Valor'}_i\,\&\,0b000000001111 \tag{10}$$

$$\text{ColorFinal}_{i_c} = (\text{BitsNuevos}_{i_c} \ll 4)|(\text{Color}_{i_c}\,\&\,0b00001111) \tag{11}$$

Recordemos que, en la ecuación (11), c representa el canal, que puede ser R, G o B. Adjuntamos el código de esta función:

```
int min = minHisto(histogram);
  int max = maxHisto(histogram);
  int maxmin = max - min;
  unsigned int prob[4096];
  int length = 4096;
  prob[0] = PREC * histogram[0]/(width*height);
  for(int i = 1; i < length; ++i){
    prob[i] = prob[i-1] + PREC * histogram[i]/(width*height)
    ;
  }
  for(int i = 0; i < width * height * 3; i=i+3){
    unsigned char colorR = image[i] >> 4;
    unsigned char colorG = image[i+1] >> 4;
    unsigned char colorB = image[i+2] >> 4;
    unsigned int index = (colorR << 8) | (colorG << 4) | (
    colorB);
    unsigned int color = (prob[index] * maxmin + PREC * min)
    /PREC;

    unsigned char colorRH = color >> 8;
    unsigned char colorGH = (color >> 4) & 0b000000001111;
    unsigned char colorBH = (color) & 0b000000001111 ;

    unsigned char colorRFinal = (colorRH << 4) | (image[i] &
     0b00001111);
    unsigned char colorGFinal = (colorGH << 4) | (image[i+1]
     & 0b00001111);
    unsigned char colorBFinal = (colorBH << 4) | (image[i+2]
     & 0b00001111);

    image[i] = colorRFinal;
    image[i+1] = colorGFinal;
    image[i+2] = colorBFinal;
  }
```

```
29 }
```

Listing 3: Ecualizado del histograma de Bits altos

Los resultados de este método, como ya veremos en un apartado posterior, son bastante extravagantes. De hecho, podemos concluir que este método no ecualiza la imagen, pues los colores de la imagen resultante no tienen nada que ver con los de la real. Eso sí, se le podria considerar como una especie de "filtro"para la imagen.

1.3. Bits bajos (descartada)

Cabe destacar también una versión que hicimos a modo de prueba, aprovechando que teníamos la versión de bits altos, que fue la de coger los bits bajos en vez de los altos de cada canal. El problema es que la imagen resultante era demasiado parecida a la original, sin apenas cambio aparente (sí que .ªplanabaün poco los colores y prácticamente eliminaba los degradados, pero era algo casi imperceptible). Como el resultado era demasiado parecido a la imagen original y la implemetación de esta versión era prácticamente igual que la de los bits altos solo que cambiando los shifts y poco más, decidimos abandonar esta versión y centrarnos en las otras, descartándola por completo.

2. Implementaciones CUDA

2.1. Implementaciones filtro Black and White

El primer filtro que hemos implementado ha sido el de imágenes en blanco y negro. Como hemos explicado en el apartado anterior, primero hacemos una conversión de la imagen a blanco y negro (para hacer un algoritmo más general) durante la cual leemos los valores de color de los píxeles y los guardamos en un histograma. Luego ecualizamos el histograma mediante la función de distribución acumulativa y aplicamos la nueva escala de colores a la imagen resultante. Para este filtro hemos hecho 2 versiones detalladas a continuación:

2.1.1. Filtro con kernels separados

Para este filtro inicial hemos hecho una versión un tanto "naive" que tiene 2 kernels. El primero paraleliza el proceso de lectura del histograma, así como la conversión de la imagen a blanco y negro. El segundo se encarga simplemente de la ecualización del histograma haciendo uso de un array con la función de distribución acumulada. Esta distribución acumulada se calcula en la CPU en una sección secuencial entre los 2 kernels. A continuación vemos el código:

```
__global__ void HistoK(unsigned int N, unsigned char *image,
    int *h){
  __shared__ int h_private[256];
  int i = 3 * (blockIdx.x*blockDim.x + threadIdx.x);
  int stride = blockDim.x * gridDim.x;
  h_private[threadIdx.x] = 0;
  __syncthreads();
  while (i < N) {
    int colorR = image[i]*2126;
    int colorG = image[i+1]*7152;
    int colorB = image[i+2]*722;
    unsigned char color = (colorR+colorG+colorB)/10000;
    image[i]=color;
    image[i+1]=color;
    image[i+2]=color;
    atomicAdd(&h_private[color], 1);
    i = i + stride;
  }
  __syncthreads();
  i = threadIdx.x;
  atomicAdd(&h[i], h_private[i]);
}
```

Listing 4: Kernel lectura Histograma + pasada a blanco y negro

Vemos que creamos una variable de indexado "i" que es el ID global del thread multiplicado por 3. Esto lo hacemos porque cada 3 posiciones de la matriz ïmageçorresponden a un nuevo píxel (3 posiciones para los 3 canales de color RGB). Haciendo ésto, estamos creando una estrategia donde cada thread se encarga de 1 píxel al completo pero sin dejar de aprovechar la localidad espacial de éstos (a diferencia de como pasaría si pusieramos una condición del estilo threadID % 3 == 0).

También creamos un histograma "privado" en memoria compartida, el cual todos los threads ayudan a inicializar a 0 (nótese que el syncthreads es necesario para asegurar que el histograma está totalmente inicializado). Acto seguido, iremos recorriendo la imagen con todos los bloques de la grid (aplicamos iterativamente un stride del tamaño de los threads totales en la grid hasta llegar al final de la imagen). En el proceso, iremos pasando la imagen de color a blanco y negro aplicando la ecuación de luminosidad vista en la sección anterior y guardando el color resultante en el histograma privado de forma atómica (para evitar data races).

Una vez todos los histogramas privados han sido actualizados con la información relevante sobre las frecuencias de color de la imagen, se actualiza el histograma en memoria global de la misma forma que hemos hecho con los privados (con instrucciones *atomicAdd*). Tenemos el histograma listo para ecualizar.

```
__global__ void Equalize(unsigned int N, unsigned char *
    image, int *minmaxArray, unsigned int *prob){
  __shared__ unsigned int PREC;
  int i = (blockIdx.x*blockDim.x + threadIdx.x);
  int stride = blockDim.x * gridDim.x;
  PREC = 10000;
    __syncthreads();
  i *= 3;
  while (i < N){
    unsigned char color = (prob[image[i]] * minmaxArray[2] +
     minmaxArray[0] * PREC) / PREC;
    image[i] = color;
    image[i+1] = color;
    image[i+2] = color;
    i = i + stride;
  }
}
```

Listing 5: Kernel ecualización histograma

En este kernel volvemos a hacer el truco de multiplicar el globalID de cada thread por 3 para asignar un píxel a cada uno sin dejar threads "idle" de por medio. De la misma forma, usamos el mismo truco de hacer el stride del tamaño de todos los threads en la grid para ir iterando la imagen. Para cada píxel, los

threads calculan el color que correspondería al histograma ecualizado acorde a la equación 5 que se ha explicado anteriormente y asigna este color a todos los canales del píxel (para tener este efecto de escala de grises).

```
int min = minHisto(histogramBW);
int max = maxHisto(histogramBW);
int maxmin = max - min;
unsigned int prob[256];
int length = 256;
prob[0] = PREC * histogramBW[0] / (width*height);
for(int i = 1; i < length; ++i){
  prob[i] = prob[i-1] + PREC * histogramBW[i]/(width*
  height);
}
int auxMinmaxArray[3] = {min, max, maxmin};
```

Listing 6: Código secuencial entre kernels

En el listado anterior vemos el código que hay entre los dos kernels ya mencionados. Este código calcula el mínimo y máximo color que se ha encontrado en la imagen recorriendo el histograma (calculado en HistoK) por delante y por detrás respectivamente. Luego, calcula la función de distribución acumulada de forma secuencial pues cada elemento depende del anterior y finalmente prepara los valores para que éstos sean pasados al kernel de Equalize. Este kernel secuencial es lo que hace que esta versión sea un poco más "naive".

Respecto al kernel de ecualización, hay que tener en cuenta que los tamaños de imagen son muy superiores al número de threads totales en la computación. Por lo tanto, es esencial el uso del iterable "while" con un stride del tamaño de la grid, de lo contrario solo se procesan unos pocos píxeles de la imagen, como vemos a continuación. Interesante destacar de esta imagen que parece que cada bloque se encarga de una fila consecutiva de píxeles, resultando en estas bandas formándose.

Figura 1: Ejemplo de ejecución sin el "while" en el kernel Equalize

2.1.2. Filtro con kernels unificados

Vista la anterior versión, varias "mejorías" se hacen aparentes. Para empezar, aunque la forma de calcular los valores mínimo y máximo de los colores de la imagen son bastante eficientes (aprovechando la estructura de histograma ya creada), se podrían paralelizar con una reducción. Adicionalmente, el cálculo de la función de distribución acumulada "prob", a pesar de parecer totalmente secuencial, en verdad se puede paralelizar con un método llamado "scan". Por ello, vamos a crear un par de kernels que paralelizen esta sección secuencial de código. Los otros dos kernels ya vistos se quedan mayoritariamente igual (exceptuando algunos pasos de datos y tal pero la computación es la misma).

```
1  __global__ void reduceMin(int *g_iD, int *g_oD){
2   unsigned int tid, s;
3   __shared__ int sD[256];
4   tid = threadIdx.x;
5   sD[tid] = g_iD[tid];
6   __syncthreads();
7   for(s = 1; s < blockDim.x; s *= 2){
8     if(tid % (2 * s) == 0){
9       int other = sD[tid + s];
10      sD[tid] = (sD[tid]<other) ? sD[tid] : other;
11    }
12    __syncthreads();
13  }
14  if (tid == 0) g_oD[0] = sD[0];
15 }
```

Listing 7: Kernel para calcular el mínimo mediante reducción

El código anterior es un código bastante estándar de reducción. En él, todos los threads de un bloque calculan el mínimo. Para hacerlo, se hace un cómputo "por niveles". Primero, dos threads consecutivos comparan sus valores, el más pequeño se guarda en el primer thread. Este thread "pasará.ªl siguiente nivel, donde la mitad de threads ya dejaran de hacer trabajo. Entonces, dos threads del mismo nivel compararan sus valores y se guardará en el primero... así iterativamente hasta terminar con un solo valor. El kernel para calcular máximos es exactamente igual pero cambiando la comparación de la línea 10 por "sD[tid] > other" para obtener el máximo.

```
1  __global__ void scanCDF(unsigned int N, unsigned int *prob,
       int *h){
2   __shared__ unsigned int prob_priv[256];
3   unsigned int tmp;
4   int i = threadIdx.x;
5   prob_priv[i] = 100000 * h[i] / (N/3);
6   __syncthreads();
```

```
  for(unsigned int s = 1; s < 256; s *= 2){
    if(i >= s){
      tmp = prob_priv[i] + prob_priv[i-s];
    }
    else{
      tmp = prob_priv[i];
    }
    __syncthreads();
    prob_priv[i] = tmp;
    __syncthreads();
  }
  prob[i] = prob_priv[i];
}
```

Listing 8: Kernel para calcular la función de distribución acumulativa mediante Scan

Este código es el encargado de hacer el cálculo de la función de distribución acumulada mediante una operación conocida como scan. Esta operación es muy común en casos donde se tiene que hacer una suma acumulada donde cada elemento se calcula como una suma del actual con el anterior. Para hacerlo, hay una variable iterable "s" que actúa como stride y va duplicándose cada iteración. Entonces, cada thread aportará el resultado de su elemento y uno "s" posiciones anteriores a la variable "'tmp". Finalmente, los resultados totales se guardan en el vector "prob". Tal como está diseñada esta operación, al final de sus iteraciones, cada thread habrá guardado su resultado sumado a todos los anteriores (gracias al "stride" cada vez más grande). Para aplicarla en el caso específico de la función de distribución acumulativa, hemos inicializado cada posición del vector con las frecuencias no acumuladas del histograma, aprovechando el scan posterior para hacer la "acumulación".

Como veremos en siguientes apartados, esta version resultó no brindar más eficiencia, posiblemente debido a overheads causados por la gran cantidad de sincronizaciones necesarias (sobretodo en el scan) para lo pequeño que es este cálculo comparado con la ecualización total de la imagen. Además, parece tener graves problemas de precisión y overflow con la aritmética entera, cosa que nos da resultados más o menos incorrectos dependiendo de la implementación. Viendo cierta pérdida en eficiencia y los problemas de precisión que nos obligarían a cambiar a otras formas de representación (usando long ints o incluso floats directamente, cosa que haría la operación aún más ineficiente), hemos optado por descartar esta versión en los otros filtros.

2.2. Implementaciones filtro Color Split

Este filtro es nuestra primera versión de un filtro capaz de procesar imágenes a color. Tal como hemos explicado en el apartado secuencial, para ésto lo que hacemos es tratar los 3 canales de color con histogramas separados, aplicar la función de distribución acumulada en los 3 y luego ecualizar cada uno por separado. De esta forma, tenemos los 3 canales ecualizados por separado. Para paralelizar este filtro, hemos hecho 3 versiones experimentando un poco con distintos niveles de granularidad y enfoques intuitivos al problema.

2.2.1. Filtro con thread por píxel

Esta versión es básicamente una versión a color de la primera versión del filtro blanco y negro que hemos visto anteriormente. Cada thread se ocupa de un píxel entero y va llenando y ecualizando 3 histogramas, uno para cada canal de color. Analizemos los cambios en los kernels:

```
__global__ void HistoK(unsigned int N, unsigned char *image,
    int *hR, int *hG, int *hB){
  __shared__ int hR_private[256];
  ...
  __syncthreads();
  while (i < N) {
    unsigned char colorR = image[i];
    unsigned char colorG = image[i+1];
    unsigned char colorB = image[i+2];
    atomicAdd(&hR_private[colorR], 1);
    atomicAdd(&hG_private[colorG], 1);
    atomicAdd(&hB_private[colorB], 1);
    i = i + stride;
  }
  __syncthreads();
  i = threadIdx.x;
  atomicAdd(&hR[i], hR_private[i]);
  ...
}

__global__ void Equalize(unsigned int N, unsigned char *
   image, int *minmaxArray, unsigned int *probR, unsigned
   int *probG, unsigned int *probB){
  ...
  while (i < N){
    unsigned char colorR = (probR[image[i]] * minmaxArray[2]
    + PREC * minmaxArray[0])/PREC;
    unsigned char colorG = (probG[image[i+1]] * minmaxArray
   [5] + PREC * minmaxArray[3])/PREC;
```

```
25      unsigned char colorB = (probB[image[i+2]] * minmaxArray
        [8] + PREC * minmaxArray[6])/PREC;
26      image[i] = colorR;
27      image[i+1] = colorG;
28      image[i+2] = colorB;
29      i += stride;
30    }
31  }
```

Listing 9: Kernels para el método thread por píxel:

En HistoK los cambios son bastante sencillos. Hemos eliminado la parte del código que unifica los 3 colores en una escala de grises acorde a la ecuación de la luminosidad (ecuación 3) y ahora para cada canal de color de la imagen hemos creado un histograma. Éstos se cargan a memoria compartida, inicializan a 0 como hacíamos antes y se van añadiendo los valores encontrados en cada canal en la posición correspondiente. Finalmente, se vuelven a pasar a los histogramas globales de forma atómica. No hay mucho interesante a comentar excepto la ausencia del paso a blanco y negro y la repetición de trabajo en 3 histogramas.

Para el kernel de Equalize, así como la sección de código secuencial vista anteriormente, simplemente hay repetición de trabajo para los 3 histogramas. La ecuación 5 de ecualización se aplica por separado y así obtenemos el color final de cada canal. Lo único relevante a destacar aquí es que ahora el paso del mínimo y máximo se hace en un array de 9 en vez de 3, puesto que contiene los valores de máximo, mínimo y (máximo - mínimo) para los 3 canales (3*3 valores = 9).

2.2.2. Filtro con thread por canal

Viendo la implementación anterior, identificamos una posible alternativa. En vez de designar que cada thread se encargue de un píxel entero, podemos hacer que cada uno se encargue de una posición de la matriz image, es decir, un canal de un píxel. Esto nos da una granularidad más fina, cosa que reduce la carga de trabajo en un solo thread pero puede incrementar los overheads por sincronización.

```
1  __global__ void HistoK(unsigned int N, unsigned char *image,
       int *h){
2    __shared__ int h_private[768];
3    int i = (blockIdx.x*blockDim.x + threadIdx.x);
4    int stride = blockDim.x * gridDim.x;
5    int j = threadIdx.x;
6    int k = j;
7    while(j < 768){
```

```
    h_private[j] = 0;
    j += 256;
  }
  __syncthreads();
  while (i < N) {
    unsigned char color = image[i];
    atomicAdd(&h_private[color + 256*(i%3)], 1);
    i = i + stride;
  }
  __syncthreads();
  while(k < 768){
    atomicAdd(&h[k], h_private[k]);
    k += 256;
  }
}

__global__ void Equalize(unsigned int N, unsigned char *
    image, int *minmaxArray, unsigned int *prob){
  __shared__ unsigned int PREC;
  int i = (blockIdx.x*blockDim.x + threadIdx.x);
  int stride = blockDim.x * gridDim.x;
  PREC = 10000;
  while (i < N){
    unsigned char color = (prob[image[i] + 256*(i%3)] *
    minmaxArray[2 + 3*(i%3)] + PREC * minmaxArray[0 + 3*(i%3)
    ])/PREC;
    image[i] = color;
    i += stride;
  }
}
```

Listing 10: Kernels para el método thread por canal:

Esta implementación es un poco más interesante pues permite hacer varias mejoras sobre la versión base o "naive". La implementación más ingenua de esta variante sería dejar de multiplicar el ID global del thread por 3 (que se hacía para offsetear el número de canales en cada píxel) y luego comprobar la congruencia del ID del thread con 3 mediante una cadena de "if elses". Esto nos daría 3 posibles casos, donde dependiendo del módulo del ID con 3 (valores posibles 0, 1 y 2) el thread se encargaría de actualizar el canal R, G y B respectivamente. Sin embargo, la documentación de Nvidia nos dice que cuando hay un "if" en un kernel, sólo los threads que cumplen la condición se ejecutan a la vez, y luego los que no. Esto haría que sólo se ejecutaran los threads encargados del canal R, luego los del G y finalmente los de B. De forma que cualquier paralelismo que pudieramos explotar con esta granularidad se anula.

Para evitar esto, hemos encontrado una solución alternativa. Primero unificamos los tres histogramas en uno sólo de 768 elementos. La posición 0 a 255 pertenecen al canal R, la 256 a 511 a la G y la 512 a 767 al canal B. Lo mismo con el vector "prob" que contiene la distribución acumulada. Luego, podemos hacer que cada thread acceda a su canal correspondiente mediante la ecuación *256*(i %3)* donde "i" es el ID global del thread. Esto aplicará un offset que hará que según la congruencia del ID a 3, cada thread se encargue del canal que le corresponde en todo caso. nótese que los "while" añadidos en HistoK para la inicialización y copia final de los histogramas son necesarios para el funcionamiento correcto. Pues como en la invocación del kernel sólo le pasamos 256 threads, si no hacemos ésto, solo inicializamos el canal R a 0, dejando los canales G y B en valores muy altos, lo que da este peculiar resultado:

Figura 2: Ejemplo de mala inicialización del histograma unificado

Como vemos, ya que los valores de R seran de 0 a 255 y los de G y B arbitrariamente altos, la imagen se nos tira hacia un color cian rojizo, pues en en CMY el cian seri 255, 0, 0. Que en RGB sería 0, 255, 255. Explicando porqué nos sale ésta imagen curiosa pero errónea.

2.2.3. Filtro con múltiples kernels

Otra alternativa curiosa sería crear un kernel para cada canal tanto en HistoK como en Equalize. Vemos la implementación a continuación:

```
__global__ void HistoR(unsigned int N, unsigned char *image,
    int *h){
  __shared__ int h_private[256];
  int i = 3*(blockIdx.x*blockDim.x + threadIdx.x);
  int stride = blockDim.x * gridDim.x;
  int j = threadIdx.x;
  h_private[j] = 0;
  __syncthreads();
  while (i < N) {
    unsigned char color = image[i];
```

```
    atomicAdd(&h_private[color], 1);
    i = i + stride;
  }
  __syncthreads();
  atomicAdd(&h[j], h_private[j]);
}

__global__ void EqualizeR(unsigned int N, unsigned char *
    image, int *minmaxArray, unsigned int *prob){
  __shared__ unsigned int PREC;
  int i = 3*(blockIdx.x*blockDim.x + threadIdx.x);
  int stride = blockDim.x * gridDim.x;
  PREC = 10000;
  while (i < N){
    unsigned char color = (prob[image[i]] * minmaxArray[2] +
     PREC * minmaxArray[0])/PREC;
    image[i] = color;
    i += stride;
  }
}
```

Listing 11: Kernels para el método multikernel:

Esta versión tampoco tiene demasiado adicional a comentar. Simplemente hemos separado eliminado las repeticiones de los histogramas en ambos kernels y triplicados los 2. De esta forma tenemos dos kernels por canal (aquí vemos los del canal R). Cada uno tiene su propio histograma y variable de indexado (que dependerá de la congruencia). Cosa que nos evita tener que jugar con los índices de un histograma más grande, entre otras cosas. También es interesante para ver como se compara la eficiencia entre estos 2 enfoques tan similares (lo veremos en el siguiente apartado).

2.3. Implementaciones filtro BITS

Finalmente, implementamos la versión paralela del filtro que unifica los bits de mayor peso de los 3 canales de color y luego los ecualiza juntos. Vemos los kernels implementados a continuación:

```
__global__ void HistoK(unsigned int N, unsigned char *image,
    int *h){
  __shared__ int h_private[4096];
  int i = 3 * (blockIdx.x*blockDim.x + threadIdx.x);
  int stride = blockDim.x * gridDim.x;
  int j = threadIdx.x;
  int k = j;
  while (j < 4096){
```

```
    h_private[j] = 0;
    j += 256;
  }
  __syncthreads();
  while (i < N) {
    unsigned char colorR = image[i];
    unsigned char colorG = image[i+1];
    unsigned char colorB = image[i+2];
    colorR = colorR >> 4;
    colorG = colorG >> 4;
    colorB = colorB >> 4;
        unsigned int index = (colorR << 8) | (colorG << 4) |
     (colorB);
    atomicAdd(&h_private[index], 1);
    i = i + stride;
  }
  __syncthreads();
  while(k < 4096){
    atomicAdd(&h[k], h_private[k]);
    k += 256;
  }
}

__global__ void Equalize(unsigned int N, unsigned char *
    image, int *minmaxArray, unsigned int *prob){
  ...
  while (i < N){
    unsigned char colorR = image[i] >> 4;
    unsigned char colorG = image[i+1] >> 4;
    unsigned char colorB = image[i+2] >> 4;
    unsigned int index = (colorR << 8) | (colorG << 4) | (
    colorB);
    unsigned int color = (prob[index] * minmaxArray[2] +
    PREC * minmaxArray[0])/PREC;

    unsigned char colorRH = color >> 8;
    unsigned char colorGH = (color >> 4) & 0b000000001111;
    unsigned char colorBH = (color) & 0b000000001111 ;

    unsigned char colorRFinal = (colorRH << 4) | (image[i] &
     0b00001111);
    unsigned char colorGFinal = (colorGH << 4) | (image[i+1]
     & 0b00001111);
    unsigned char colorBFinal = (colorBH << 4) | (image[i+2]
     & 0b00001111);

    image[i] = colorRFinal;
```

```
48      image[i+1] = colorGFinal;
49      image[i+2] = colorBFinal;
50      i = i + stride;
51    }
52  }
```

Listing 12: Kernels para el filtro por bits:

Los Kernels que vemos en el código anterior son muy similares a los de la primera versión del filtro en blanco y negro que hemos visto. Pues solo hay un histograma y un kernel para cada proceso. Los principales cambios son en los cálculos en sí, los cuales son bastante similares a la versión secuencial. Para la lectura del histograma, cogemos los bits de mayor peso de cada canal y los concatenamos para decidir la posición a la que se le sumará la frecuencia. Es notable destacar que como ahora estamos trabajando con 12 bits de "color", el histograma ya no será de 256 posibles valores, sino 4096 (2 elevado a 12). Esto nos fuerza a trabajar a veces con tipos de datos con mayor rango (usar ints en vez de chars) y a usar los mismos "while" que hemos visto en el filtro de color por canal para asegurar que inicializamos correctamente el histograma. Además, hay que actualizar la cantidad de memoria reservada para el histograma acorde al nuevo tamaño de éste. De lo contrario, resultando en artefactos bastante curiosos como el que hay a continuación. En este caso, el error es debido a sólo trabajar con los valores más bajos del histograma, que corresponden a valores R y G nulos, solo considerando los bits de B absolutos. Por lo tanto, este curioso error nos muestra una especie de "filtro de azules"de la imagen.

Figura 3: Ejemplo de mala inicialización del histograma de bits

Para la parte de ecualización, no mucho cambia excepto el juego de bits que hay que hacer para usar parte de los bits del histograma ecualizado a la vez que usamos los bits bajos originales de los colores de la imagen. También es notorio el juego con los rangos de tipos que hay que hacer para no salirnos de rango y hacer overflow. Sobretodo si tenemos en cuenta la aritmética de enteros de la que hacemos uso y que aumenta bastante el rango.

3. Experimentación y Resultados

En esta sección detallaremos el proceso de experimentación con las varias implementaciones CUDA de los distintos métodos comparándolas entre ellas y también con el código secuencial. Primero vamos a poner los resultados de cada método de ecualización (los de las versiones paralelas, concretamente).

Figura 4: Imagen Original

Figura 5: Blanco y negro

Figura 6: Color Split

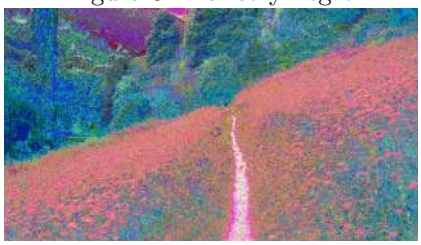

Figura 7: BITS

Figura 8: Resultados de aplicar cada método de ecualización a la imagen IMG03.jpg

Observamos que el blanco y negro ecualiza como debería la imagen (la imagen en blanco y negro sin ecualizar tiene mucho menos contraste). El Color Split sí que detalla más la imagen, en ciertos aspectos (se ven más detalles en la montaña de fondo o entre los árboles, que se pueden distinguir mucho mejor unos de otros) aunque el aspecto de la imagen es mucho más artificial con estos colores. Para acabar, el BITS nos da una imagen muy extraña, en la que se puede distinguir que la imagen sigue ahí pero los colores son completamente distintos a los de la imagen inicial. No añade especialmente detalles, de hecho, tiene menos detalle la versión .ecualizada"que la original. Por lo tanto, podemos concluir que, de las versiones a color, la que mejor ecualiza es la Color Split.

No vamos a adjuntar imágenes de las versiones secuenciales porque son iguales (lo hemos comprobado haciendo diff).

A continuación, y sabiendo que el resultado que dan las versiones paralelas y las secuenciales es igual (a excepción de Blanco y Negro con kernels unificados),

podemos comparar tiempos de ejecución y anchos de banda. Para ello hemos cogido la misma imagen (IMG03.jpg) y hemos realizado 5 ejecuciones de cada versión de los métodos de ecualización. Hemos realizado luego las medias tanto del tiempo global (es decir, el tiempo total del programa, incluyendo lecturas y escrituras de imagen), el tiempo de ecualización (es decir, sin tener en cuenta la carga de la imagen en memoria y la escritura de la imagen resultante) y el ancho de banda (que es el resultado de dividir el tamaño de la imagen, $n_{pixeles} \times 3$, entre el tiempo de ecualización. A continuación mostramos los distintos resultados:

	BW Secuencial	**BW Unificado**	**BW separado**
Tiempo Global	710.6124146	734.078418	689.5185912
Tiempo Ecualización	31.8245986	5.1097664	5.1124544
Ancho de Banda	500.4494375	3116.710878	3115.026351

Cuadro 1: Tiempos (en ms) y Ancho de banda (en MB/s) de las versiones en negro y blanco

	CSSeq	**CSThByP**	**CSThByCh**	**CSMulti**
Tiempo Global	1279.079394	1250.79043	1243.085181	1246.083569
Tiempo Ecualización	39.4822006	5.1369152	5.1226624	6.8238592
Ancho de Banda	403.3622663	3100.173219	3108.80978	2333.763757

Cuadro 2: Tiempos (en ms) y Ancho de banda (en MB/s) de las versiones del Color Split (Secuencial, Thread by Pixel, Thread by Channel y Multikernel en ese orden)

	BITS Secuencial	**BITS Paralelo**
Tiempo Global	1310.986377	1270.797803
Tiempo Ecualización	30.7897994	5.1392832
Ancho de Banda	517.2362773	3098.771596

Cuadro 3: Tiempos (en ms) y Ancho de banda (en MB/s) de las versiones de BITS

Para ver las tablas enteras, se puede ver el archivo tablas.pdf.

Observando la tabla, nos podemos dar cuenta de que las versiones paralelas son claramente más rápidas que las secuenciales, especialmente en cuanto a tiempo de ecualización y, por tanto, en ancho de banda. Si nos ponemos a comparar versiones paralelas entre ellas, la cosa ya no está tan clara.

BW tiene dos versiones paralelas. La que solo paraleliza la creación del histograma y su ecualización y la que lo paraleliza todo. Si bien es cierto que la versión ünificada"tiene un rendimiento ligeramente superior a la otra, no es una mejora sustancial (motivo por el cual, como ya hemos mencionado, acabamos por

abandonarla). Esto se puede deber a que el rendimiento que ganaríamos de no tener que calcular la distribución acumulada secuencialmente, lo perdemos con los `reduceMin`, `reduceMax` y `scanCDF`, que tienen a muchos threads sin trabajar por la forma en la que están distribuidas las tareas.

Color Split cuenta con 3 versiones paralelas distintas. Una en la que cada pixel es tratado por un thread, otra en la que cada píxel es tratado por 3 threads (uno por canal) y otra en la que los 3 canales se tratan en Kernels distintos. La tabla nos muestra que la peor versión paralela es claramente esta última. Seguramente esto se deba al overhead que supone tener que llamar a 3 kernels distintos tanto para crear los histogramas como para ecualizar los canales, resultando en un rendimiento peor. En cuanto a las otras dos versiones, vemos que la versión Thread By Channel es ligeramente mejor que la de Thread By Pixel, pero por muy poco. Esto se puede deber a que cada thread tiene que hacer menos trabajo por iteración de los bucles de los kernels.

De BITS solo hemos hecho una versión, pues realmente no teníamos ninguna otra manera que no hubiéramos probado ya de paralelizar esta ecualización.

4. Conclusiones

Este trabajo nos ha permitido observar y probar varias técnicas de ecualización de imágenes. Aunque no se ha podido explicar en tanto detalle debido a la gran cantidad de resultados finales y versiones, hemos ido experimentando sobre varios aspectos de la ecualización, por ejemplo con el paso de color a blanco y negro hemos podido experimentar con diversas fórmulas (de más ingenuas a más realistas), hemos observado varias técnicas de ecualización de imágenes de color (tanto usar los canales por separado como modificar bits específicos de un color) e implementado diversas versiones paralelas de estos algoritmos. Pese a que son *embarassingly parallel* (ya que no hay dependencias entre píxeles) y por lo tanto no hay mucho que explorar en cuanto a técnicas de recorrido de matrices, aunque sí que hemos podido jugar con granularidades y niveles de paralelización. En general, creemos haber aplicado una progresión iterativa de mejora y experimentación que ha dado información bastante interesante en el proceso. Un método que nos gustaría haber explorado es el de thread por bloque de píxeles, aunque eso hubiera complicado mucho el tratamiento de imágenes de cualquier dimensión. También se podría intentar dividir la imagen en 4 subimágenes y enviar cada una a una gráfica distinta, para intentar minimizar aún más el tiempo de ecualización.

En cuanto a los filtros implementados y sus versiones, creemos que el mejor es el filtro color split, concretamente la versión de "thread por canal" al ser la más eficiente. El filtro de blanco y negro da buenos resultados pero nos fuerza a obtener solo imágenes sin color (puede ser interesante para ecualizar y ajustar la luminosidad de las imágenes pero es bastante limitado en este sentido). El filtro Bits, por contra, hace cambios o muy leves e imperceptibles (cambiando los bits bajos) o cambios que alteran totalmente la estructura de la imagen (cambiando los bits altos). Los resultados son interesantes y este filtro acentúa ciertas propiedades de las imágenes como los focos de luz (como sería el sol o una lámpara) o gradientes de colores (por luz difusa o similar) tal y como vemos en la figura 11. Por contra, el filtro color split da imágenes que, a pesar de tener colores bastante surreales, permiten sacar la mayor información de éstas. Permitiendo distinguir detalles muy concretos como distintas especies de árboles y plantas por su diferente tonalidad de verde (acentúando algunos hacia tonos azulados y otros a tonos más amarillentos o rojizos), vislumbrar detalles en zonas oscuras o demasiado claras, etc. El hecho que no destruya la imagen como el filtro por bits o trabaje con colores limitados como el filtro blanco y negro, realmente ayuda a vislumbrar detalles tal y como es el propósito de la ecualización.

En un futuro, se podrían plantear posibles mejoras como jugar con tamaños de bloque y de grid para ver qué versiones son más eficientes (por ejemplo en la versión color split de thread por canal, vimos que con 768 threads por

bloque, saturamos el histograma y aumentábamos el ancho de banda por unos 400MB/s), experimentar con otras formas de ecualizar imágenes a color (al igual que hemos hecho con el filtro por bits) o aumentar la fracción paralela del código paralelizando también el proceso de lectura y escritura de la imagen.

Figura 9: Imagen Original

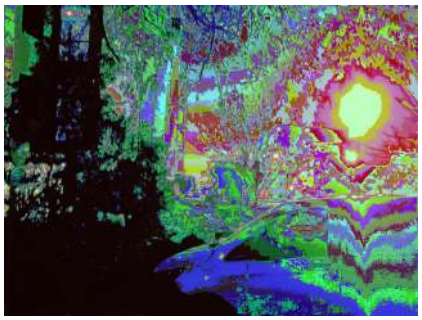

Figura 10: Versión BITS

Figura 11: Comparación entre una imagen con un foco de luz y su versión tras aplicarle BITS

Referencias

[1] *Color Theory*. https://en.wikipedia.org/wiki/Relative_luminance.

[2] *Fundamentos de Procesamiento de Imagenes - Domingo Mery*. https://github.com/domingomery/imagenes.

[3] *Histogram Equalization - Wikipedia*. https://en.wikipedia.org/wiki/Histogram_equalization.

[4] *Transparencias TGA - CUDA - Agustín Fernández - UPC - FIB*.