BM40A1400 GPGPU course project: Color histogram

Due on Tuesday, February 28, 2017

Maria Glukhova & 500503 Ekaterina Nepovinnykh

Contents

1 Introduction

The goal of this project is to create a color histogram for provided image using GPGPU and CUDA [1]. This course project includes the following tasks:

1. GPU implementation of color histogram [2]
2. Multiple GPU support
3. CPU implementation for performance comparison
4. Matlab implementation to confirm that GPU implementation is faster
5. Python wrapper using PyCuda [3]

The work was done in a public repository on GitHub (see [4]).

2 CUDA Implementation

CUDA-based computation of color histogram is performed in two steps:

1. Gather information into partial histograms using atomic counters
2. Accumulate partial histograms into one resulting histogram

Both steps are performed as GPU kernels using one intermediate GPU memory buffer and the result is then copied back into host memory.

2.1 First-pass kernel

Listing 1 shows first-pass histogram kernel. It gathers the information from a part of the image to the partial hisogram. Part of the image is defined by block and grid size ("part of image" is not a continious patch here, rather every n-th pixel). The partial histogram is stored in global memory defined by "out". Every block has a group number (g) assigned to it, and it treats part of global memory (from out + g * NUM_PARTS and every nt bits, where nt is the size (x*y) of the grid) as its own, storing the histogram of its part of the image here.
What happens in the kernel is:
the blocks' own bits of global memory get initialized with zeros.
Every corresponding (assigned to this block) pixel get visited, and:

1. Pixel is decoded into several (ACTIVE_CHANNELS) colors.
2. These values are binned to the desired number of bins (NUM_BINS).
3. Corresponding values in a partial histogram (part of global memory) get updated.

Parts 1 and 2 are implemented in DecodePixel (see listing 2).

2.2 Accumulating partial histograms

Listing 3 shows **histogram_gmem_accum** kernel which gathers partial histograms from global memory (in) into output histogram.
Partial histograms are expected to be stored in a part of global memory of n * ACTIVE_CHANNELS * NUM_BINS * sizeof(uint) size. It is expected that every nth bit belongs to the same partial histogram (the k-th histogram stored in k, n + k, 2 * n + k, ... elements).
Every block running the kernel has a particular bin on the output histogram it is responsible for. It gathers information about this bin from every partial histogram, and sums it up.

2.3 CPU wrapper for kernel execution

Listing 4 shows a function **run_gmem_atomics** which is a CPU wrapper for the histogram computation. It assigns the blocks, allocates the memory for partial histograms, computes them using **histogram_gmem_atomics** (see listing 1) and accumulates with **histogram_gmem_accum** (see listing 3).

2.4 Multiple GPU support

Multiple GPU version is implemented similarly to single GPU one, but with extra parallelization based on color channel. As such, every device has a specific color channel assigned to it. Listings 5 and 6 show implementation of multi-GPU color histogram computation.

3 Matlab implementation

Listing 7 shows a matlab function that computes color histogram of an input image. The function accepts the image itself, bit depth of an image and the number of bins in the resulting histogram. The function performs the necessary computation in exactly $W * H * C$ consecutive steps, where:

W is the width of an image in pixels

H is the height of an image in pixels

C is the number of color channels in an image

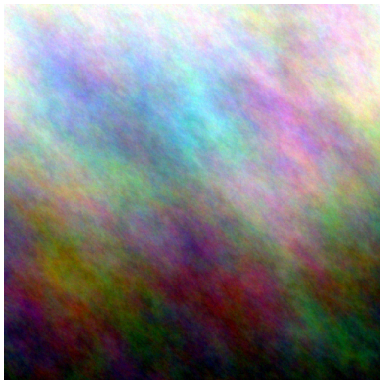
As such, among the images with the same number of color channels and the same color depth this implementation is expected to take linear amount of time relative to the total numer of pixels in an image.

4 Python wrapper

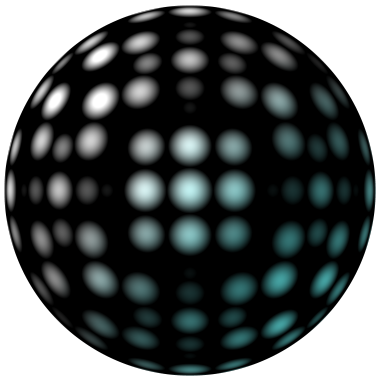
As a part of the task, python wrapper using exising C++-based CUDA code was implemented. The relevant part of the wrapper is presented in listing 8. The algorithm is essentially the same as with C++ version, except Python wrapper requires convertion of native numeric types to their numpy equivalent in order to preserve memory layout for GPU kernels.

5 Experiments

Experiments were conducted on LUT GPGPU cluster in order to determite runtime characteristics of various implementations on the same set of images. The goal is to assert that GPGPU implementation is faster on larger images than both C++ and Matlab implementations. Another goal is to confirm that CPU implementation is linear relative to total number of pixels in an image while GPU is sub-linear.
Fig. 1a, fig. 1c and fig. 1b show examples of images that were used during testing. Several image samples have been prepared with different sizes and color depths in order to mesure performance of all implemented methods.
Fig. 2 shows the plot that compares different implementations total run time relative to size of an input image. The plot clearly shows that CPU-based implementations have linear relative to total number of pixels while GPGPU-based implementations are sub-linear. The best performance was achieved on single-GPU implementation, probably due to syncronization between different devices on multi-GPU version.



(a) plasma0500.png test image (size: 500x500px)



(b) spotted_ball_3500.png test image (size: 3500x3500px)

(c) bstar100.png test image (size: 100x100px)

Figure 1: Test images

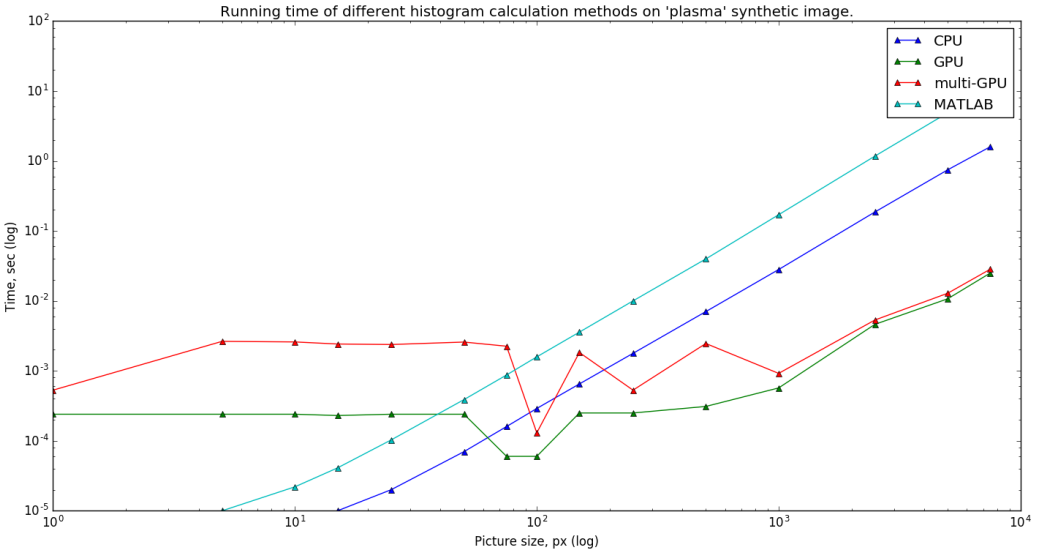


Figure 2: Comparison of performance between different implementations of color histogram

6 Code listings

Listing 1: First-pass histogram kernel

```
__global__ void histogram_gmem_atomics(
        const PixelType *in,
        int width,
        int height,
        unsigned int *out)
{
    /**
    * First-pass histogram kernel (binning into privatized counters)
    * @param in - input image, uchar4 array of continuously placed pixel values.
    * @param width - int, image width in pixels.
    * @param height - int, image height in pixels.
    * @param out (output) - partial histograms.
    */
    // Global position and size.
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int nx = blockDim.x * gridDim.x;
    int ny = blockDim.y * gridDim.y;

    // thread index in workgroup, linear in 0..nt-1
    int t = threadIdx.x + threadIdx.y * blockDim.x;
    int nt = blockDim.x * blockDim.y; // total threads in workgroup

    // Group index in 0..ngroups-1.
    int g = blockIdx.x + blockIdx.y * gridDim.x;

    // Initialize global memory.
    unsigned int *gmem = out + g * NUM_PARTS;
    for (int i = t; i < ACTIVE_CHANNELS * NUM_BINS; i += nt){
        gmem[i] = 0;
    }
    __syncthreads();

    // Process pixels (updates our group's partial histogram in gmem).
    for (int col = x; col < width; col += nx)
    {
        for (int row = y; row < height; row += ny)
        {
            PixelType pixel = in[row * width + col];

            unsigned int bins[ACTIVE_CHANNELS];
            DecodePixel(pixel, bins);

            #pragma unroll
            for (int CHANNEL = 0; CHANNEL < ACTIVE_CHANNELS; ++CHANNEL) {
                atomicAdd(&gmem[(NUM_BINS * CHANNEL) + bins[CHANNEL]], 1);
            }
        }
    }
}
```

Listing 2: Decode uchar4 pixel into bins

```
__device__ __forceinline__ void DecodePixel(
        uchar4 pixel,
        unsigned int (&bins)[ACTIVE_CHANNELS])
{
    /**
    * Decode uchar4 pixel into bins.
    * @param pixel - uchar4 pixel value.
    * @param bins (output) - Array of ACTIVE_CHANNELS uints representing binned
    *                        channel value.
    */
    unsigned char* samples = reinterpret_cast<unsigned char*>(&pixel);

    #pragma unroll
    for (int CHANNEL = 0; CHANNEL < ACTIVE_CHANNELS; ++CHANNEL) {
        bins[CHANNEL] = (unsigned int) (samples[CHANNEL]) / K_BIN;
    }
}
```

Listing 3: Accumulate partial histograms into global one

```
__global__ void histogram_gmem_accum(
        const unsigned int *in,
        int n,
        unsigned int *out)
{
    /**
    * Accumulate partial histograms into global one.
    * @param in - input partial histograms.
    * @param n - total number of blocks.
    * @param out (output) - global histogram.
    */
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i > ACTIVE_CHANNELS * NUM_BINS) {
        return; // out of range
    }

    unsigned int total = 0;
    for (int j = 0; j < n; j++) {
        total += in[i + NUM_PARTS * j];
    }

    out[i] = total;
}
```

Listing 4: CPU Wrapper for GPU histogram computation

```
void run_gmem_atomics(
        PixelType *d_image,
        int width,
        int height,
        unsigned int *d_hist)
{
    /**
    * Wrapper for GPU histogram computing.
    * @param in - input image, uchar4 array of continuously placed pixel values.
    * @param width - int, image width in pixels.
    * @param height - int, image height in pixels.
    * @param out (output)
    */
    int device_count = 0;
    cudaError_t error_id = cudaGetDeviceCount(&device_count);

    dim3 block(32, 4);
    dim3 grid(16, 16);
    int total_blocks = grid.x * grid.y;

    // Allocate partial histogram.
    unsigned int *d_part_hist;
    cudaMalloc(&d_part_hist, total_blocks * NUM_PARTS * sizeof(unsigned int));

    dim3 block2(128);
    dim3 grid2((ACTIVE_CHANNELS * NUM_BINS + block2.x - 1) / block2.x);

    histogram_gmem_atomics<<<grid, block>>>(
        d_image,
        width,
        height,
        d_part_hist);

    histogram_gmem_accum<<<grid2, block2>>>(
        d_part_hist,
        total_blocks,
        d_hist);

    cudaFree(d_part_hist);
}
```

Listing 5: First pass histogram kernel for multiple GPUs

```
__global__ void histogram_gmem_atomics1(
        const PixelType *in,
        int width,
        int height,
        unsigned int *out,
        int dev_id,
        int dev_count)
{
    // Global position and size.
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int nx = blockDim.x * gridDim.x;
    int ny = blockDim.y * gridDim.y;

    // thread index in workgroup, linear in 0..nt-1
    int t = threadIdx.x + threadIdx.y * blockDim.x;
    int nt = blockDim.x * blockDim.y; // total threads in workgroup

    // Group index in 0..ngroups-1.
    int g = blockIdx.x + blockIdx.y * gridDim.x;

    // Initialize global memory.
    unsigned int *gmem = out + g * NUM_PARTS;
    for (int i = t; i < ACTIVE_CHANNELS * NUM_BINS; i += nt)
    gmem[i] = 0;
    __syncthreads();

    // Process pixels (updates our group's partial histogram in gmem).
    for (int col = x; col < width; col += nx)
    {
        for (int row = y; row < height; row += ny)
        {
            PixelType pixel = in[row * width + col];

            unsigned int bins[ACTIVE_CHANNELS];
            DecodePixel(pixel, bins);

            // Every device process its own channel(s).
            #pragma unroll
            for (int CHANNEL = dev_id; CHANNEL < ACTIVE_CHANNELS; CHANNEL += dev_count) {
                atomicAdd(&gmem[(NUM_BINS * CHANNEL) + bins[CHANNEL]], 1);
            }
        }
    }
}
```

Listing 6: CPU Wrapper for multiple GPUs histogram computation

```
void run_multigpu(
        PixelType *d_image,
        int width,
        int height,
        unsigned int *d_hist,
        int device_id,
        int device_count)
{
    dim3 block(32, 4);
    dim3 grid(16, 16);
    int total_blocks = grid.x * grid.y;

    // Allocate partial histogram.
    // Actually, we need less memory (only the channels assigned to the device),
    // but for the sake of simplicity, let us have the "full" histogram for
    // every device (still counting only relevant bits).
    // TODO: Memory-efficient way.
    unsigned int *d_part_hist;
    cudaMalloc(&d_part_hist, total_blocks * NUM_PARTS * sizeof(unsigned int));

    dim3 block2(128);
    dim3 grid2((ACTIVE_CHANNELS * NUM_BINS + block2.x - 1) / block2.x);

    histogram_gmem_atomics1<<<grid, block>>>(
        d_image,
        width,
        height,
        d_part_hist,
        device_id,
        device_count
        );

    histogram_gmem_accum<<<grid2, block2>>>(
        d_part_hist,
        total_blocks,
        d_hist);


    cudaFree(d_part_hist);
}
```

Listing 7: Matlab function for computing color histogram of an image

```
function hist = histogram( img, bit_depth, num_bins )
%HISTOGRAM Computes c-dimensional color histogram of an image
%   c is number of channels in an image
%   arguments:
%      img - input image matrix (w*h*c)
%      bit_depth - input image bit depth
%      num_bins - number of bins in the histogram
%   return hist - num_bins*c color histogram

[w,h,c] = size(img);
channel_depth = bit_depth / c;
hist = zeros(num_bins, c);

% total number of possible values for each channel is assumed to be 256
bin_size = floor(2^channel_depth / num_bins);

for x=1:w
    for y=1:h
        for z=1:c
            % compute the bin that the value in (x, y, z) belongs to
            bin = floor(double(img(x,y,z)) / bin_size) + 1;
            % increment the bin
            hist(bin, z) = hist(bin, z) + 1;
        end;
    end;
end;
end
```

Listing 8: Python wrapper that runs CUDA-based color histogram calculation

```
def histogram(image_path, num_bins):
    image = misc.imread(image_path)
    bin_size = 256 / num_bins

    # calculate image dimensions
    (w, h, c) = image.shape

    image = image.view(numpy.uint32) # reinterpret image with 4-byte type
    dest = numpy.zeros((c, bin_size), numpy.uint32)
    parts = num_bins * c
    block1 = (32, 4, 1); grid1 = (16, 16, 1)
    partial = numpy.zeros(grid1[0] * grid1[1] * parts, numpy.uint32)
    block2 = (128,1, 1); grid2 = ((c * num_bins + block2[0] - 1) / block2[0], 1, 1)
    W = numpy.dtype(numpy.int32).type(w); H = numpy.dtype(numpy.int32).type(h)

    # run CUDA kernels
    histogram_atomics(drv.In(image), W, H, drv.Out(partial), block=block1, grid=grid1)
    sz = numpy.dtype(numpy.int32).type(grid1[0] * grid1[1])
    histogram_accum(drv.In(partial), sz, drv.Out(dest), block=block2, grid=grid2)

    return dest
```

References

[1] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[2] Gpu pro tip: Fast histograms using shared atomics on maxwell. https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/. Accessed: 2017-02-20.

[3] Pycuda. https://mathema.tician.de/software/pycuda/. Accessed: 2017-02-20.

[4] kwadraterry/gpgpu-lut: Practical assignment for gpgpu course in lut. https://github.com/kwadraterry/GPGPU-LUT. Accessed: 2017-02-27.