Histogram algorithm

A histogram algorithm is one of essential tools in processing images. It allows visualizing distribution of color in an image. This further allows to judge certain properties of the image. Associated with this are some common operation on images, such as equalization or saturation.

The algorithm has a *O(n)* (where *n* is the size of the problem, in this case size of input array) time complexity since all individual data points need to be processed and appropriately counted. This also means that for large problems computation of algorithm rises linearly with input. In modern applications, especially image processing or its real time variant, this operation can be significantly time consuming when processed using a single processing node. Also due to the ease of separation and parallelization of data this algorithm is very easy to implement on GPU thanks to its parallel design.

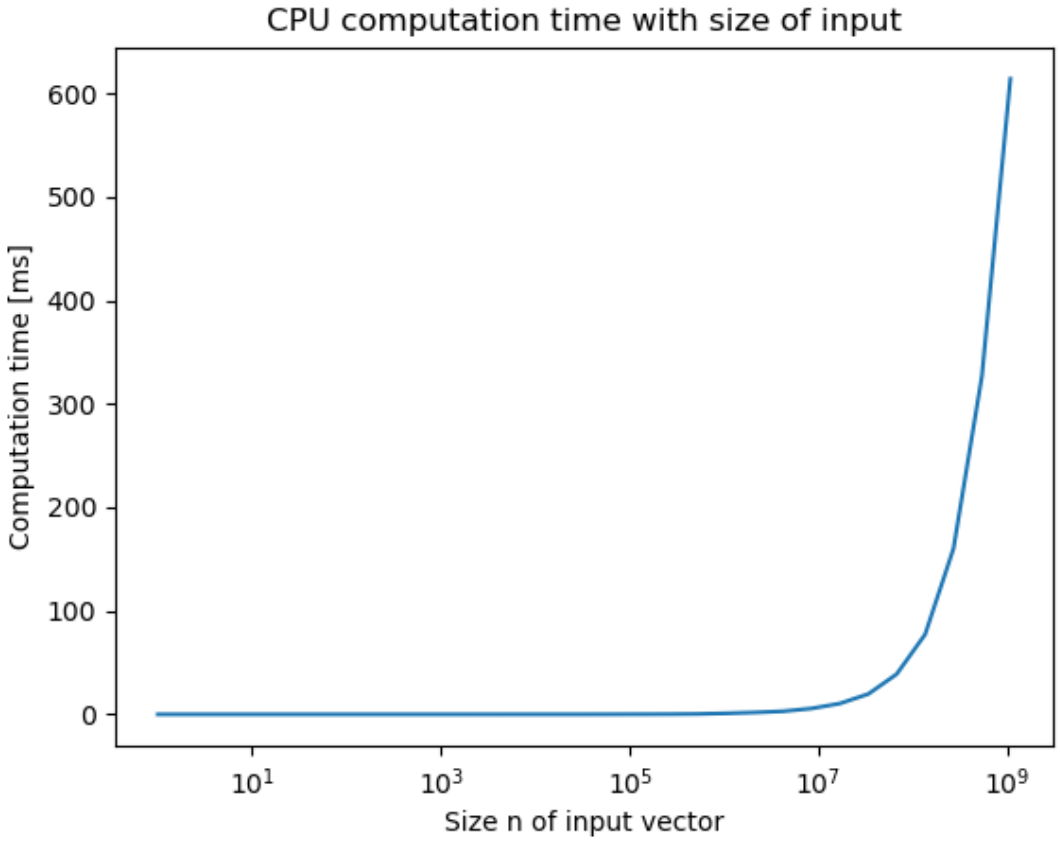

Fig. 1. CPU computation time. Computed for 4096 bins.

As shown in fig. 1 with large data the time to compute a histogram on a single CPU core can take up to even 600 ms. To give some idea how this can affect performance let's consider an image with dimensions 1920x1080 in RGB which gives n values to be processed:

$$n = 1920 \cdot 1080 \cdot 3 = 6\,220\,800$$

Using the same program the computation of that amount of data points takes about 4 ms to complete, which while processing many images can be not insignificant especially considering that GPU offers considerable speedup shown later.

Implementing the algorithm on CUDA-enabled GPU is very easy thanks to the before mentioned ease of partition of the problem. That being said newer GPUs offer additional possibility for improving performance. Namely they allow to privatize block-local histograms computed by each block. Additionally this approach requires using atomic operations so as to ensure each committal of data to both global and shared memory maintains order. Also in this exercise the required saturation of bins is achieved using an additional kernel, whereas this operation is done while computing when on using CPU.

Below is a presentation of how long the computations take depending on the approach used and how much data is processed.

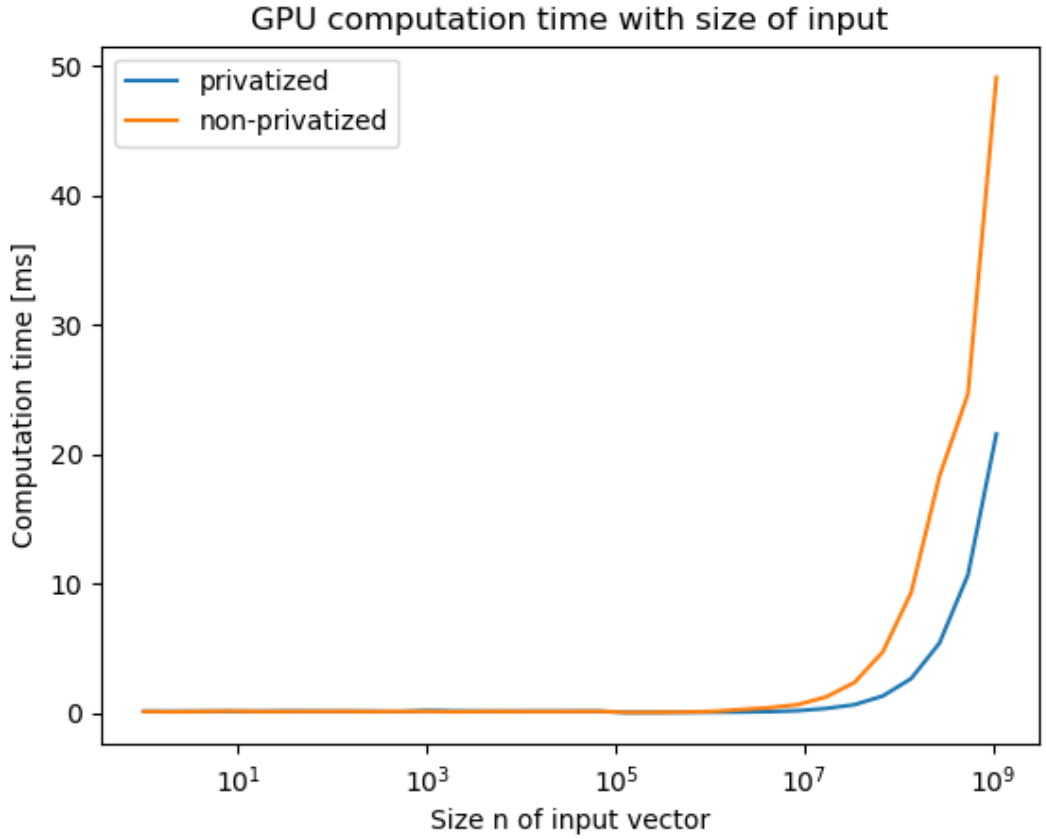

Fig. 2. GPU computation time. Computed for 4096 bins.

GPU computation follows a certain memory movement plan since data is created/initialized on CPU and thus has to be moved to the device for computation:

1. Initialization and memory allocation on CPU side
2. Copying input data to device global memory
3. Launching first kernel, responsible for doing the initial count without saturation

a) If a privatized approach is chosen: first local helper variables are initialized, then local histogram, held in shared memory, is initialized with zeroes. After that the local histogram is computed and committed to global memory to be saturated
b) If a non-privatized approach is chosen: each partition is immediately computed and committed to global memory using atomic operations

4. Second kernel is launched to saturate the computed histogram by setting a cut-off value, in this exercise 256. This happens strictly using the global memory
5. Computed histogram data is transferred back to host memory and verified against CPU result

It's worth noting that presented solution uses automatic memory transfers between host and device by way of virtual memory (memory allocated with *cudaMallocManaged*) . This means that a necessary step needs to be taken before launching the first kernel which is to prefetch asynchronously the input data to device memory.

Code

```
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
#include <math.h>

#include <cuda_runtime.h>

#define MAX_BINS 4096
#define THREADS_PER_BLOCK 512
#define SATURATION_VALUE 256

#define checkCudaErrors(code) { cudaAssert((code), __FILE__,
__LINE__); }
inline void cudaAssert(cudaError_t code, const char *file, int line,
bool abort = true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"cudaAssert: %s %s %d\n",
cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}

void initInput (unsigned long int size, unsigned int* data, unsigned
int binCount) {
     for(int i = 0; i < size; ++i) {
          data[i] = rand() % binCount;
     }
}

__global__ void cudaHistogramPrivate(unsigned long int dataSize,
unsigned int* dataInput, unsigned int binCount, unsigned int* bins)
{
```

```
     unsigned th = threadIdx.x;
     unsigned bx = blockDim.x;
     unsigned i = blockIdx.x * blockDim.x + th;
     unsigned gx = gridDim.x;
     unsigned stride = gx * bx;
     extern __shared__ unsigned int local_hist[];

     // init prywatnego histo
     for(unsigned int j = th; j < binCount; j += bx) {
          local_hist[j] = 0;
     }
     __syncthreads();

     // zliczanie
     for(unsigned int j = i; j < dataSize; j += stride) {
          atomicAdd(&(local_hist[dataInput[j]]), 1);
     }
     __syncthreads();

     // commit do vramu
     for(unsigned int j = th; j < binCount; j += bx) {
          atomicAdd(&(bins[j]), local_hist[j]);
     }
}

__global__ void cudaHistogramNonPrivate(unsigned long int dataSize,
unsigned int* dataInput, unsigned int binCount, unsigned int* bins)
{
     int th = threadIdx.x;
     int i = blockIdx.x * blockDim.x + th;

     // zliczanie
     for(unsigned int j = i; j < dataSize; j += blockDim.x *
gridDim.x) {
          atomicAdd(&bins[dataInput[j]], 1);
     }
}

__global__ void cudaSaturateHistogram (unsigned int binCount,
unsigned int* histogram) {
     int th = threadIdx.x;

     for(unsigned int j = th; j < binCount; j += blockDim.x) {
          if(histogram[j] > SATURATION_VALUE) {
               histogram[j] = SATURATION_VALUE;
          }
     }
}

void histogramGPU(unsigned long int dataSize, unsigned int*
dataInput, int binCount, unsigned int* bins, int priv) {
     cudaEvent_t start;
     checkCudaErrors(cudaEventCreate(&start));
```

```
	cudaEvent_t stop;
	checkCudaErrors(cudaEventCreate(&stop));

	checkCudaErrors(cudaEventRecord(start, NULL));
	// odpalanie kerneli
	if(priv == 1) {
		cudaHistogramPrivate<<<120, THREADS_PER_BLOCK, binCount *
sizeof(unsigned int)>>>(dataSize, dataInput, binCount, bins);
	}
	else {
		cudaHistogramNonPrivate<<<120,
THREADS_PER_BLOCK>>>(dataSize, dataInput, binCount, bins);
	}

	cudaDeviceSynchronize();

	checkCudaErrors(cudaGetLastError());

	cudaSaturateHistogram<<<8, 512>>>(binCount, bins);

	checkCudaErrors(cudaGetLastError());

	checkCudaErrors(cudaEventRecord(stop, NULL));

	checkCudaErrors(cudaEventSynchronize(stop));
	float msecTotal = 0.0f;
	checkCudaErrors(cudaEventElapsedTime(&msecTotal, start, stop));

	printf("GPU code run time: %f ms\n", msecTotal);

	cudaEventDestroy(start);
	cudaEventDestroy(stop);
}

void histogramCPU(unsigned long int dataSize, unsigned int*
dataInput, int binCount, unsigned int* bins) {
	timespec start, end;

	clock_gettime(CLOCK_MONOTONIC_RAW, &start);

	for(unsigned long int i = 0; i < dataSize; ++i) {
		if(bins[dataInput[i]] < SATURATION_VALUE) {
			bins[dataInput[i]]++;
		}
	}

	clock_gettime(CLOCK_MONOTONIC_RAW, &end);

	float msecTotal = (float)((end.tv_sec - start.tv_sec) * 1000000
+ (end.tv_nsec - start.tv_nsec) / 1000) / 1000;
	printf("CPU code run time: %f ms\n", msecTotal);
}
```

```
void checkBins(unsigned int binCount, unsigned int* cpuBins,
unsigned int* gpuBins) {
     for(int i = 0; i < binCount; ++i) {
          if(cpuBins[i] != gpuBins[i]) {
               printf("BLAD!!!!!!! i = %d: %d != %d\n", i,
cpuBins[i], gpuBins[i]);
               return;
          }
     }
}

int main(int argc, char** argv) {
     unsigned int priv, binCount;
     unsigned long int dataSize;
     char* endptr;

     if(argc != 4) {
          exit(-1);
     }

     dataSize = strtoul(argv[1], &endptr, 0);
     binCount = atoi(argv[2]);
     priv = atoi(argv[3]);

     if(binCount > MAX_BINS)
          binCount = MAX_BINS;

     cudaDeviceProp prop;
     checkCudaErrors(cudaSetDevice(0));
     checkCudaErrors(cudaGetDeviceProperties(&prop, 0));

     printf("running on: %s\ndata size: %lu bin count: %d private:
%s\n", prop.name, dataSize, binCount, priv == 1 ? "true" : "false");

     srand(time(0));

     unsigned int* dataInput;
     checkCudaErrors(cudaMallocManaged(&dataInput, dataSize *
sizeof(unsigned int)));

     initInput(dataSize, dataInput, binCount);

     unsigned int* bins;
     checkCudaErrors(cudaMallocManaged(&bins, binCount *
sizeof(unsigned int)));

     for(unsigned int i = 0; i < binCount; ++i)
          bins[i] = 0;

     unsigned int cpuBins[binCount];

     for(unsigned int i = 0; i < binCount; ++i){
          cpuBins[i] = 0;
     }
```

```
     histogramCPU(dataSize, dataInput, binCount, cpuBins);

     checkCudaErrors(cudaMemPrefetchAsync(dataInput, dataSize *
sizeof(unsigned int), 0));
     checkCudaErrors(cudaMemPrefetchAsync(bins, binCount *
sizeof(unsigned int), 0));
     histogramGPU(dataSize, dataInput, binCount, bins, priv);

     checkBins(binCount, cpuBins, bins);

     checkCudaErrors(cudaFree(dataInput));
     checkCudaErrors(cudaFree(bins));

     printf("end\n");

     return EXIT_SUCCESS;
}
```

Naturally the code needs to be compiled using Nvidia's dedicated compiler *nvcc* after which example usage can look like this:

./executable_name size_of_input number_of_bins private

Where:

- size_of_input – size of input data (number n)
- number_of_bins – size of output histogram, number between 1 and 4096
- private – 1 for privatized GPU approach or 0 for non-privatized

In the current version of the program the input data is initiated with 1s so as to make validation of output easier.