Master MAIA Parallel Processing System

PROJECT

Comparison of performance histogram computation between CPU and GPU

Supervisor: **Saverio De Vito** *Author:* **Zafar Toshpulatov**

University of Cassino and Southern Latium

July 20, 2018

I. Abstract

Histograms are a commonly used analysis tool in image processing and data mining applications. They show the frequency of occurrence of each data element. Although trivial to compute on the CPU, histograms are traditionally quite difficult to compute efficiently on the GPU. This report presents two methods of parallel histogram computation using CUDA C++.

This project will be covered considering not only theoretical aspects but practical examples will be presented in order to understand how to implement histogram computation. The examples performed for this demonstration begin from CPU implementations and will end up giving a proposal for a medical imaging issue regarding parallel histogram computation using images for 8 bins.

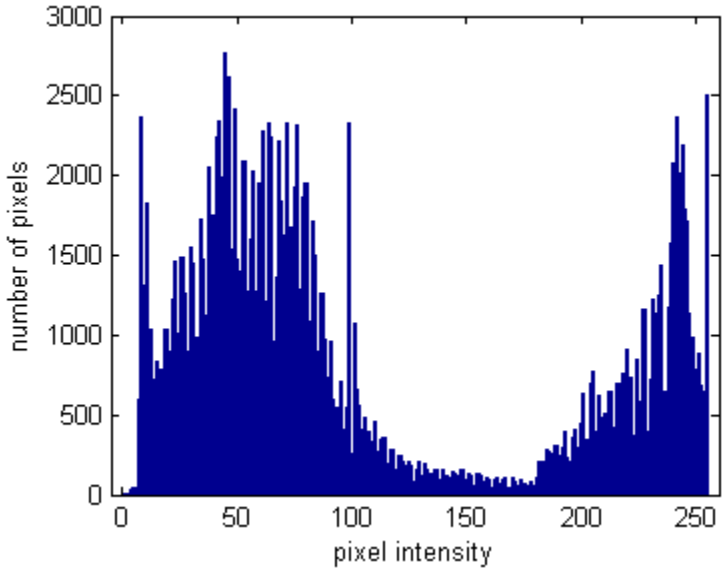
We can use CUDA and the shared memory to efficiently produce histograms, which can then either be read back to the host or kept on the GPU for later use. The CUDA sample demonstrate approaches to efficient histogram computation on GPU using CUDA.

II. Introduction

Histograms are an important data representation with many applications in computer vision, data analytics and medical imaging. A histogram is a graphical representation of the data distribution across predefined bins. The input data set and the number of bins can vary greatly depending on the domain, so let's focus on one of the most common use cases: an image histogram using 8 bins for images (fixed pixel values). Even though we'll use a specific problem setup the same algorithms can benefit other computational domains as well.

A basic serial image histogram computation is relatively simple. For each pixel of the image we find a corresponding integer bin and increment its value. Atomic operations are a natural way of implementing histograms on parallel architectures. Depending on the input distribution, some bins will be used much more than others, so it is necessary to support efficient accumulation of the values across the full memory hierarchy. This is similar to reduction and scan operations, but the main challenge with histograms is that the output location for each element is not known prior to reading its value. Therefore, it is impossible to create a generic parallel accumulation scheme that completely avoids collisions. Histograms are now much

easier to handle on GPU architectures thanks to the improved atomics performance in Kepler and native support of shared memory atomics in Maxwell.



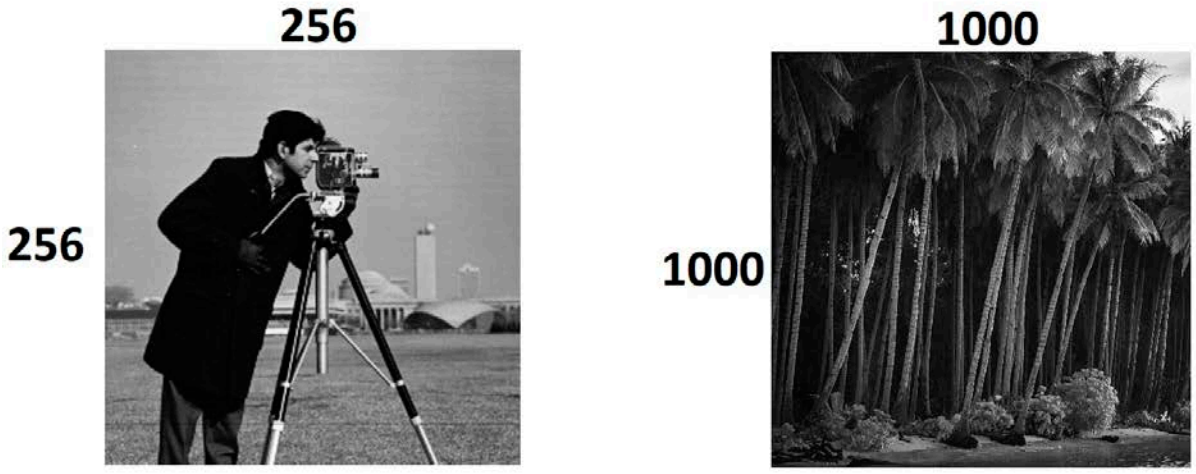
III. Problem description

Computation of image analysis algorithms are slow on CPU, therefore there is a need to parallelize and use GPU. The majority of the computers in the market today have a GPU but most of the image processing applications are still serial. The idea behind this project is to present the possibility to use the CUDA programming in order to launch commonly used in a GPU and improve the overall performance of the histogram computation.

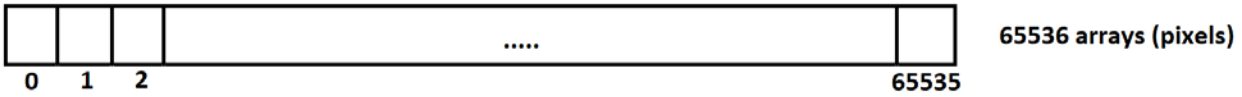
First, I started basic implementation of histogram computation on CPU using two images (256 x 256 and 1000 x 1000 size) fixed arrays for 8 bins (from 0 to 31, 32 to 63 and 224 to 255 and so on). The same histogram computation was implemented on GPU using CUDA C++ programming. Then performance was compared between CPU and GPU and it was determined which was the most optimal implementation.

IV. Parallelization strategy

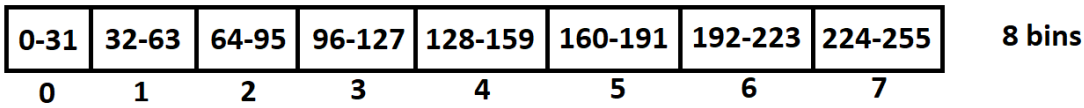
The histogram of two images of pixel size 256x256 and 1000x1000 (65.536 and 1.000.000 pixels) were computed. In order to read image pixel values in Cuda C++, the image pixel values were converted to 2D array in txt file with help of Matlab.



First we look 256x256 pixels image that it has 65.536 pixels values as arrays.

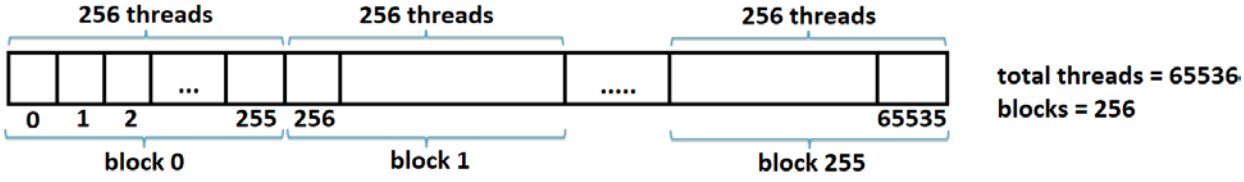


8 bins are considered from 0 to 255 as grayscale image and three different methods are used to implement the parallelization of histogram computation.



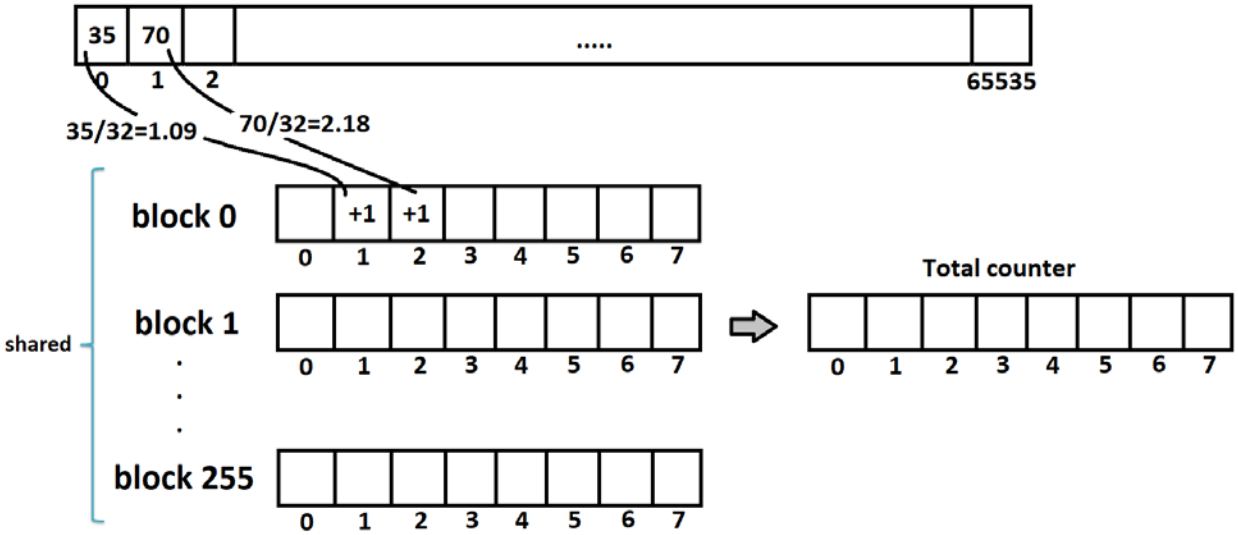
Case 1.

I decided to take the optimal 256 threads per block and each thread take care each element (pixel). The blocks are defined dividing number of arrays by threads per block and taking largest integer.



The partial counter arrays is initialized to zero in shared memory. Once one thread reaches the *__syncthreads( )* call, it will wait until all threads have reached (finishing initialization) it.

Each thread checks own pixel and computes bin index dividing by 32. When we update shared memory, we need to do AtomicAdd. If threads want to update the local histogram, they need to wait and cannot update same time. AtomicAdd will make sure that no one try to access value until that operation finish.



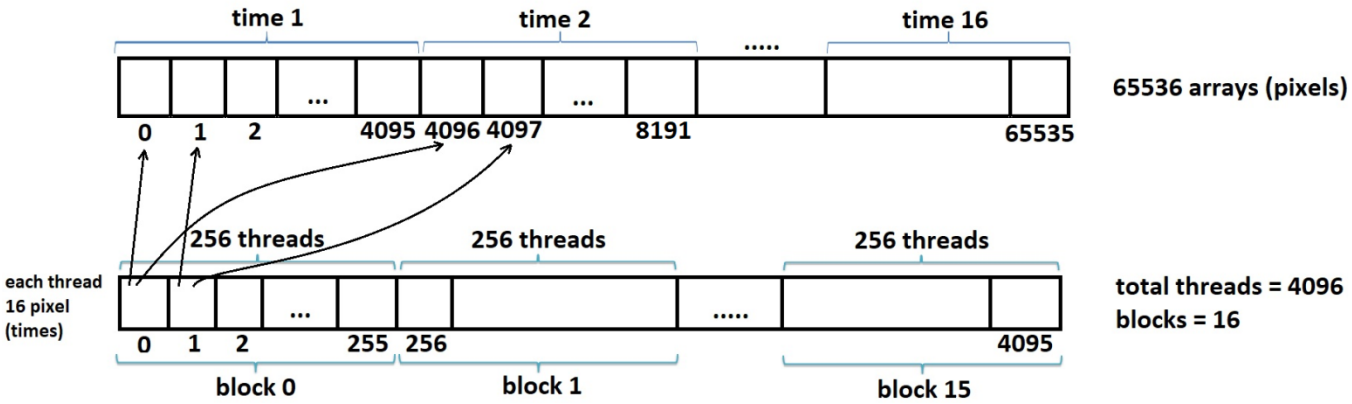
The partial histogram (partial counter) is added to global histogram (total counter) using AtomicAdd. Again we need *__syncthreads( )* that all threads wait for thread0 to finish.

Advantage: For each pixel one thread (256 threads for 256 pixels). Each thread computes only one pixel.

Disadvantage: AtomicAdd is slow.

Case 2.

In this case also 256 threads are used and each thread has own counters and only one update own memory. There is no need atomicAdd and registers are used. Each thread computes pixels 16 times.



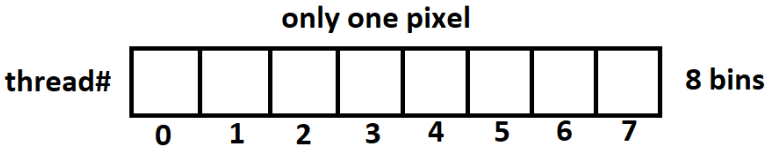
Only one block may (doesn't fit) suffer.

Advantage: Register memory is fastest.

Disadvantage: Few threads (4096) computes 16 times.

Case 3.

In this case I tried to use above two advantages. Each thread computes only one pixel and registers is used.



Disadvantage: In this case also have problem. Each thread checks only one pixel. 8 bins all of them zero except one.

Comparing the three cases, all of them have a negative and positive.

V. Code

Computing on the CPU:

```
#include <stdio.h>
#include <iostream>
#include <cuda.h>
#include <stdlib.h>
#include <ctime>
#include <fstream>

using namespace std;
const int imgH = 256;
const int imgW = 256;
const string filename = "/home/maia/cuda-workspace/Zafar/histogram1/img256.txt";

/* Read file and load to 2D array */
void loadFromFile(std::string filename, int mat[imgH][imgW]);

int main()
{

	int img[imgH][imgW];

	loadFromFile(filename, img);

	const int TotalBins = 8;
	int counter[TotalBins];

	clock_t t;
	t = clock();

	for(int i=0; i<TotalBins; i++){
		counter[i]=0; //initializing bins with 0
	}

	for(int i=0; i<imgH; i++){
		for(int j=0; j<imgW; j++){
			int binIdx = (int)img[i][j]/32;
			counter[binIdx]++;
		}
	}

	t = clock() - t;
	cout << "Ellapsed Time: " << t << " miliseconds" << endl;
	cout << CLOCKS_PER_SEC << " clocks per second" << endl;
	cout << "Ellapsed Time: " << t*1.0/CLOCKS_PER_SEC << " seconds" << endl;


	for(int i=0; i<TotalBins; i++){
		cout<<"Bins "<<i<<" :"<<counter[i]<<endl;
	}
	return 0;
}

void loadFromFile(string filename, int mat[imgH][imgW]) {
```

```
        std::ifstream fin;
        fin.open(filename.c_str());
        if (!fin) { std::cerr << "cannot open file"; }
        for (int i = 0; i<imgH; i++) {
                for (int j = 0; j<imgW; j++) {
                        fin >> mat[i][j];
                }
        }
        fin.close();
}
```

Computing on the GPU:

Case 1.

```
#include <stdio.h>
#include <iostream>
#include <cuda.h>
#include <stdlib.h>
#include <ctime>
#include <fstream>

using namespace std;

__global__ void histogram_compt(int *array , int *total_counter){

        int Thread_index = threadIdx.x;

        // global index
        int index = Thread_index + blockIdx.x * blockDim.x;

        // step 1: compute local/partial histogram (i.e. counter)

        __shared__ int partial_counter[8];


        if (Thread_index==0) { // only done by thread#0
                for(int i=0; i<8; i++){
                        partial_counter[i] = 0;
                }
        }
        __syncthreads();

        int Bin_index = array[index]/32;
        atomicAdd(&partial_counter[Bin_index], 1);


        // step 2: add partial histogram to global histogram

        if (Thread_index==0) {
                for(int i=0; i<8; i++){
                        atomicAdd(&total_counter[i], partial_counter[i]);

                }
```

```
        }

        __syncthreads();

}


const int imgH = 256;
const int imgW = 256;
const string filename = "/home/maia/cuda-workspace/Zafar/histogram1/img256.txt";

/* Read file and load to 2D array */
void loadFromFile(std::string filename, int mat[imgH][imgW]);

int main()
{
        int array_length = imgH*imgW;
        int img[imgH][imgW]; //input
        loadFromFile(filename, img);
        //int array[] = img;

        const int TotalBins = 8; //bins
        int counter[TotalBins]; //counter

        //Device array
        int *dev_array, *dev_counter;

    //Allocate the memory on the GPU
    cudaMalloc((void **)&dev_array , array_length*sizeof(int) );
    cudaMalloc((void **)&dev_counter , TotalBins*sizeof(int) );

    //Copy Host array to Device array
    cudaMemcpy(dev_array, &img, array_length*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_counter, &counter, TotalBins*sizeof(int), cudaMemcpyHostToDevice);

        int threadsPerBlock = 256; //optimal
    //int blocksPerGrid =(array_length + threadsPerBlock - 1) / threadsPerBlock;
        float blocksPerGrid = array_length/threadsPerBlock;
        int blocks = floor(blocksPerGrid);

    cudaEvent_t start, stop;
        float time;
        cudaEventCreate (&start);
        cudaEventCreate (&stop);
        cudaEventRecord (start, 0);

    //Make a call to GPU kernel, blocks, 256 threads
    histogram_compt <<< blocks, threadsPerBlock >>> (dev_array, dev_counter);

        cudaEventRecord (stop, 0);
        cudaEventSynchronize (stop);
        cudaEventElapsedTime (&time, start, stop);
        cudaEventDestroy (start);
        cudaEventDestroy (stop);
```

```
    //Copy back to Host array from Device array
    cudaMemcpy(&counter, dev_counter, TotalBins*sizeof(int), cudaMemcpyDeviceToHost);

    for(int i=0; i<TotalBins; i++){
                cout<<"Bins "<<i<<" :"<<counter[i]<<endl;
        }

    cout << "Ellapsed Time: " << time << endl;

    cudaFree(dev_array);
    cudaFree(dev_counter);

    return 0;

}

void loadFromFile(string filename, int mat[imgH][imgW]) {
        std::ifstream fin;
        fin.open(filename.c_str());
        if (!fin) { std::cerr << "cannot open file"; }
        for (int i = 0; i<imgH; i++) {
                for (int j = 0; j<imgW; j++) {
                        fin >> mat[i][j];
                }
        }
        fin.close();
}
```

Case 2.

```
#include <stdio.h>
#include <iostream>
#include <cuda.h>
#include <stdlib.h>
#include <ctime>
#include <fstream>

using namespace std;

const string filename = "/home/maia/cuda-workspace/Zafar/histogram1/img256.txt";
const int imgH = 256;
const int imgW = 256;
int array_length = imgH*imgW;

int threadsPerBlock = 256; //optimal
int pixels_per_thread = 16;
float blocksPerGrid = array_length/threadsPerBlock; //256*256/256=256
int blocks = floor(blocksPerGrid)/pixels_per_thread; //256/16=16
int total_threads = blocks*threadsPerBlock; //16*256=4096


__global__ void histogram_compt(int *array, int *total_counter, int
pixels_per_thread, int total_threads){

        // global index
```

```
        int index = threadIdx.x + blockIdx.x * blockDim.x;

        // step 1: compute local/partial histogram (i.e. counter)

        int partial_counter[8];

        for(int i=0; i<8; i++){
                partial_counter[i] = 0;
        }

        for(int i=0; i<pixels_per_thread; i++){
                int Bin_index = array[index + (total_threads * i)]/32;
                partial_counter[Bin_index]++;
        }

        // step 2: add partial histogram to global histogram

        for(int i=0; i<8; i++){
                atomicAdd(&total_counter[i], partial_counter[i]);
        }

}


/* Read file and load to 2D array */
void loadFromFile(std::string filename, int mat[imgH][imgW]);

int main()
{
     int img[imgH][imgW]; //input
     loadFromFile(filename, img);

     const int TotalBins = 8; //bins
     int counter[TotalBins]; //counter

     //Device array
     int *dev_array, *dev_counter;

    //Allocate the memory on the GPU
    cudaMalloc((void **)&dev_array , array_length*sizeof(int) );
    cudaMalloc((void **)&dev_counter , TotalBins*sizeof(int) );

    //Copy Host array to Device array
    cudaMemcpy(dev_array, &img, array_length*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_counter, &counter, TotalBins*sizeof(int), cudaMemcpyHostToDevice);

    cudaEvent_t start, stop;
     float time;
     cudaEventCreate (&start);
     cudaEventCreate (&stop);
     cudaEventRecord (start, 0);

    //Make a call to GPU kernel, blocks, 256 threads
    histogram_compt <<< blocks, threadsPerBlock >>> (dev_array, dev_counter,
pixels_per_thread, total_threads);

     cudaEventRecord (stop, 0);
```

```
    cudaEventSynchronize (stop);
    cudaEventElapsedTime (&time, start, stop);
    cudaEventDestroy (start);
    cudaEventDestroy (stop);


    //Copy back to Host array from Device array
    cudaMemcpy(&counter, dev_counter, TotalBins*sizeof(int), cudaMemcpyDeviceToHost);

    for(int i=0; i<TotalBins; i++){
            cout<<"Bins "<<i<<" :"<<counter[i]<<endl;
        }

    cout << "Ellapsed Time: " << time << endl;

    cudaFree(dev_array);
    cudaFree(dev_counter);

    return 0;

}

void loadFromFile(string filename, int mat[imgH][imgW]) {
        std::ifstream fin;
        fin.open(filename.c_str());
        if (!fin) { std::cerr << "cannot open file"; }
        for (int i = 0; i<imgH; i++) {
                for (int j = 0; j<imgW; j++) {
                        fin >> mat[i][j];
                }
        }
        fin.close();
}
```

Case 3.

```
#include <stdio.h>
#include <iostream>
#include <cuda.h>
#include <stdlib.h>
#include <ctime>
#include <fstream>

using namespace std;

__global__ void histogram_compt(int *array , int *total_counter){

        // global index
        int index = threadIdx.x + blockIdx.x * blockDim.x;

        // step 1: compute local/partial histogram (i.e. counter)

        int partial_counter[8];
```

```
        for(int i=0; i<8; i++){
                partial_counter[i] = 0;
        }

        int Bin_index = array[index]/32;
        partial_counter[Bin_index]++;


        // step 2: add partial histogram to global histogram

        for(int i=0; i<8; i++){
                atomicAdd(&total_counter[i], partial_counter[i]);
        }

}


const int imgH = 256;
const int imgW = 256;
const string filename = "/home/maia/cuda-workspace/Zafar/histogram1/img256.txt";

/* Read file and load to 2D array */
void loadFromFile(std::string filename, int mat[imgH][imgW]);

int main()
{
        int array_length = imgH*imgW;
        int img[imgH][imgW]; //input
        loadFromFile(filename, img);

        const int TotalBins = 8; //bins
        int counter[TotalBins]; //counter

        //Device array
        int *dev_array, *dev_counter;

    //Allocate the memory on the GPU
    cudaMalloc((void **)&dev_array , array_length*sizeof(int) );
    cudaMalloc((void **)&dev_counter , TotalBins*sizeof(int) );

    //Copy Host array to Device array
    cudaMemcpy(dev_array, &array, array_length*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_counter, &counter, TotalBins*sizeof(int), cudaMemcpyHostToDevice);

        int threadsPerBlock = 256; //optimal
        float blocksPerGrid = (array_length)/threadsPerBlock;
        int blocks = floor(blocksPerGrid);

    cudaEvent_t start, stop;
     float time;
     cudaEventCreate (&start);
     cudaEventCreate (&stop);
     cudaEventRecord (start, 0);

    //Make a call to GPU kernel, blocks, 256 threads
    histogram_compt <<< blocks, threadsPerBlock >>> (dev_array, dev_counter);
```

```
        cudaEventRecord (stop, 0);
        cudaEventSynchronize (stop);
        cudaEventElapsedTime (&time, start, stop);
        cudaEventDestroy (start);
        cudaEventDestroy (stop);


    //Copy back to Host array from Device array
    cudaMemcpy(&counter, dev_counter, TotalBins*sizeof(int), cudaMemcpyDeviceToHost);

    for(int i=0; i<TotalBins; i++){
                cout<<"Bins "<<i<<" :"<<counter[i]<<endl;
        }

    cout << "Ellapsed Time: " << time << endl;

    cudaFree(dev_array);
    cudaFree(dev_counter);

    return 0;

}

void loadFromFile(string filename, int mat[imgH][imgW]) {
        std::ifstream fin;
        fin.open(filename.c_str());
        if (!fin) { std::cerr << "cannot open file"; }
        for (int i = 0; i<imgH; i++) {
                for (int j = 0; j<imgW; j++) {
                        fin >> mat[i][j];
                }
        }
        fin.close();
}
```

VI. Environment

All experiments were carried out on the NVIDIA TitanXp machine, the following tables show most important setup properties.

PROPERTY	MACHINE
CPU	1, 4 cores
RAM	32 GB
GPU	NVIDIA TitanXp, 12GB onboard
CUDA version	9.0.176
CUDNN version	7005
Linux version	4.13.0-41-generic, amd64

VII. Results

Comparison of performance histogram computation between CPU and GPU.

Example	Image size, pixels (arrays)	Method	Elapsed time (execution time)	Average
CPU	256x256	CPU	2.06 sec	2.61 sec
	1000x1000		3.16 sec	
GPU	256x256	Case 1	0.48 sec	sec
	256x256	Case 2	sec	
	256x256	Case 3	0.94 sec	
	1000x1000	Case 1	0.57 sec	sec
	1000x1000	Case 2	sec	
	1000x1000	Case 3	2.54 sec	

VIII. References

1. Victor Podlozhnyuk, Histogram calculation in CUDA, November 2017.
2. Introduction to CUDA C – Nvidia, www.nvidia.com/content/GTC-2010/pdfs/2131_GTC2010.pdf