Parallel Computing Roman Andoni - 260585085

ECSE 420 Yordan Neshev - 260587938



CUDA Implementation of Lab1 and Lab2

Lab Experiment Report

Prof. Zeljko Zilic

McGill Faculty of Engineering

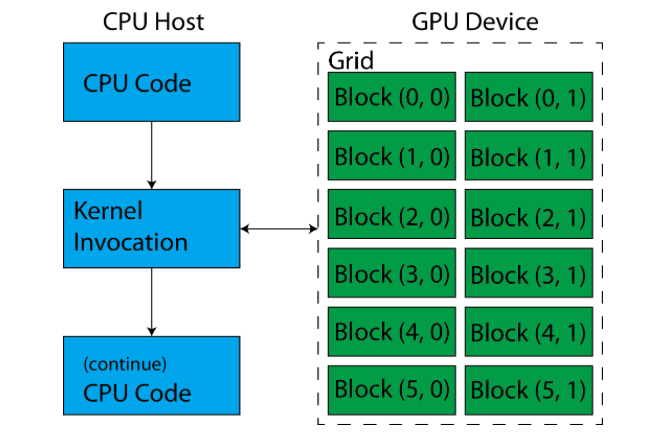
November 8th, 2016

**Abstract:**

In this experiment, it was required to implement the algorithms from the first lab and the second lab using CUDA. Specifically, it was required to write the rectification, pool, convolution and the 4x4, 512x512 drum simulation algorithms using parallelization on GPUs. CUDA is another shared-memory programming model designed by Nvidia for machines using standalone GPUs, although there exist several emulators already that also allow for CUDA execution. This alternative approach allows taking advantage of parallelization on GPU processors rather than the CPU. In fact, CUDA gives direct access to the GPU's virtual instruction set and parallel computational elements. By harnessing the power of GPU processors, it is possible to dramatically increase the performance of parallelizable algorithms and greatly reduce the execution times. In this experiment, the algorithms will be calling the CUDA specific functions to allocate the necessary memory and pass the necessary data for filtering and simulating the drum surface to the GPU. We expect to see a meaningful decrease in run times as the GPU processors are significantly faster than the CPU itself.

**Implementation**

The overall algorithm logic did not change comparing the previous labs, however, several changes have been implemented in order to support CUDA. A given CUDA algorithm consists of two actors: the host and the GPU kernel. In out case, the host is the CPU and the host is the GPU residing on a virtual machine provided by Prof. Zilic. The overall structure of an application that is able to run on a GPU is shown in the figure below:



***Fig.1 (Typical structure of a GPU app)1***

As it can be seen above, the application begins its execution on the CPU host. Once the GPU is needed for parallel execution of the parallelizable part of the application, the host calls a GPU kernel on an available GPU connected to the same machine. The GPU kernel is executed on a grid, called the GPU grid. This grid is composed of several independent group of threads called thread blocks. All threads in such blocks are running in parallel. Such setup allows parallel execution using a large amount of threads. In general, the algorithms that we implemented flow the following way:

* CPU 🡪 Initialize all the variable, load the necessary data (ex: lodepng image data)
* CPU 🡪 Allocate the necessary memory for the GPU processing
  + Using *cudaMalloc()*
* CPU 🡪 Copy the necessary data (image array or array of surface dots) from the the host CPU to the GPU
  + *Using cudaMemcpy()*
* CPU 🡪 Launch GPU kernel, start timer.
* GPU 🡪 Retrieve data from the passed host, compute, flush to output.
* CPU 🡪 Copy the results from the GPU back to the host program (ex: filtered data)
  + *Using cudaMemcpy()*
* CPU 🡪 Continue execution, terminate the program

In terms of language, CUDA implementation was easy due to the fact that it uses the C syntax, thus it was possible to slightly modify the original threading code. For instance, we had to use *\_\_global\_\_* qualifier with *void* return type in order to identify the part of code that the GPU kernel will be executing. For instance, the main rectification function in the rectification program had to implement the following way:

*\_\_global\_\_ void rectify(unsigned char \* cud\_input, unsigned char \* cud\_output)*

***Fig.2 (Rectification kernel fucntion)***

Where *cud\_input* is the input array that the kernel will be receiving and *cud\_output* is the output array into which the kernel will be flushing the processed results. Another important aspect to consider was the kernel invocation itself. For invocation, CUDA uses Chevron Syntax, allowing to invoke the kernel as a simple function call. Specifically, we used the “<<< >>>” brackets that contain the necessary parameters for degree of parallelism of our program. The parameters in the brackets indicated how many threads do we want to use per block, mentioned above, and how many blocks will be there in the grid. A kernel invocation in our pool algorithm can be observed as:

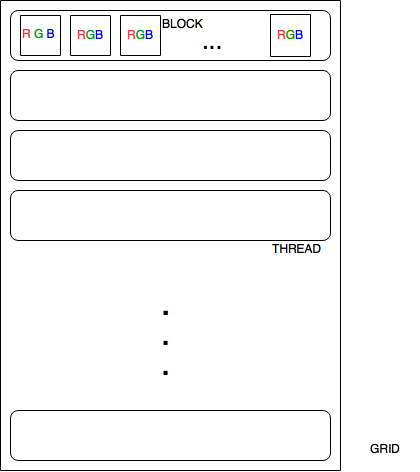
*rectify<<<dimGrid, dimBlock>>>(cud\_input, cud\_output)*

***Fig.3 (Rectification kernel invocation)***

It is important to note that the *dimGrid* and *dimBlock* parameters are three dimentional vector types (*dim3*) that represent the grid and the block dimensions. For instance, *dimBlock* in our case is equal to the input image width multiplied by the size of char. In other words, it is a vector of form *{ input\_image\_width \* sizeof(unsigned char), 1, 1}*. With those dimensions, each thread will know its position in the grid and the block, which will allow to access the necessary (x,y) value in the array to be processed.

In the case of Lab 1 algorithms, the general structure was simple. At first, the host started by loading image data into the allocated array and getting the input image dimensions. Then, the host allocated the necessary GPU memory for input and output arrays by calling the *cudaMalloc* function with the respected array size and the *cudaMemcpyHostToDevice* parameter indicating the direction of the input flow. Following, the algorithms initialized the block and grid dimensional vectors described above. Given that we can issue a large amount of threads, in case of rectification, for example, we used *input\_image\_width \* sizeof(unsigned char) \* 1 \*1* threads per block and *input\_image\_height \* 1 \* 1* blocks per grid. Following, the host invokes the kernel with the input and output arrays previously allocated for the GPU as well as the dimensional vectors in the <<<>>> brackets. Once the kernel finalizes its execution, the host CPU continues the main program execution by copying back the results into the output allocated memory by calling the *cudaMemcpy* CUDA function, but now with the *cudaMemcpyDeviceToHost* parameter, to indicate that the data flows from the GPU slave to the host CPU. Finally the image is written back to the PNG using the lode library.

The kernel functions themselves, denoted with the \_\_global\_\_ qualifier conserved the algorithm logic (pool, convolution and rectification specific). However, this time, in order to identify the location of the data to process in the array, a thread within a block uses CUDA specific identifiers, such as, blockIdx.x, threadIdx.x and blockDim, which are the block index within the grid, the thread index within the block and the dimension of the block itself. In a nutshell, we break down the input image array in such way so that we have the number of threads in a block equal to the number of pixels in the image, which is 994. This number MUST NOT surpass 1024 as no current GPU supports more than 1024 threads per block. Furthermore, we set up the number of blocks per grid equal to the height of the image, which is equal to 998. This number, however, can be considerably large. Same analogy applies for the surface simulation algorithms, where the number of threads per block is either 4 or 512. One can visualize the array data distribution within a grid in the following diagram:



***Fig.4 (Rectification kernel invocation)***

The algorithms in the second lab follow the similar structure as those in the first lab, however, slight differences were introduced. For instance, it was required to allocate several arrays for the GPU in order to be able to pass the neighboring data from one block to another using the shared memory. Moreover, the kernel function contains the general update algorithm, that on its turn, calls the boundary, corner and the center update algorithms that are denoted with the *\_\_device\_\_*

qualifier. This qualifier allows the update functions to be called only and only from the device (GPU) and not from the cpu. Thus, the update functions are be executed on the GPUs. Finally, the work division was done similarly to the algorithms in the first lab. In the case of 4x4 grid algorithm, we divided the surface to be distributed on one grid, with one thread per block and 16 blocks per grid. Similarly, the 512x512 grid was divided on one grid, with 512 blocks and 512 threads within a thread.

**Results**

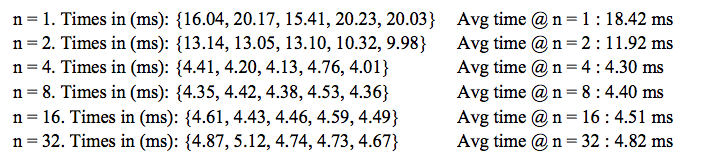
All algorithms were executed and tested on the VM connected to a GPU provided by Prof. Zilic.

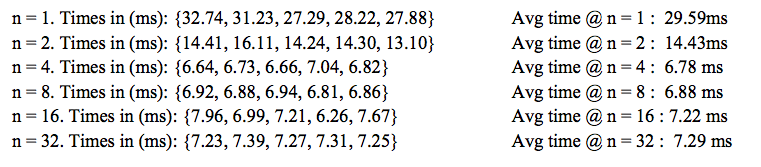
The following run time results were obtained:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 |
| Rectification | 2.146ms | 2.151ms | 2.154ms | 2.134ms | 2.300ms |
| Pool | 0.772ms | 0.711ms | 0.808ms | 0.790ms | 0.784ms |
| Convolution | 3.568ms | 3.533ms | 3.652ms | 3.560ms | 4.498ms |
| 4x4 Grid |  |  |  |  |  |
| 512x512 Grid |  |  |  |  |  |

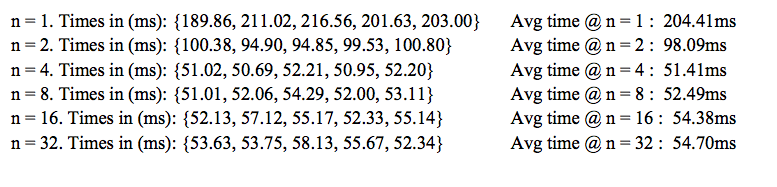
***Fig.5 (CUDA results)***

Previous lab results:

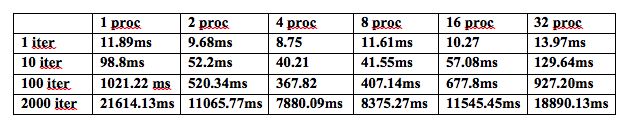
 ***Fig.6 (Lab 1 rectification)***

****

***Fig.7 (Lab 1 pooling)***

****

***Fig.8 (Lab 1 convolution)***

****

***Fig.10 (Lab 2 execution results)***

As it can be observed, the results obtained with CUDA allow smaller run times, compared to multithreading on CPU. For the rectification algorithm, with 32 threads, the algorithm took 4.82ms vs 2.221 on average with CUDA. This results a speedup of around 2.21. With the pooling algorithm, the speedup observed is around 9.289 and finally the convolution algorithm with CUDA resulted a speedup of around 11.42. Similar results are observed with the drum algorithms, where 4\_4 algorithm resulted runtimes of 0.534ms, 1.0361ms, 10.972ms and 200.53 ms, resulting much quicker times for 1,10,100 and 2000 iterations.

**Conclusion**

To conclude, our assumptions were confirmed. Due to the GPU’s ability to execute operations in parallel using a large pool of threads, it is possible to speed up the processing and decrease run times by using CUDA. While GPUs core clock frequency can be smaller than the one of CPU, the GPU has much more cores compared to CPU, thus allowing a more efficient workload distribution. The tests confirmed that by using CUDA, we are able to run the parallelizable part of our algorithms on the provided GPU, thus decreasing the overall computational and run time and resulting significant speedups.