Scalable Computing: Practice and Experience, ISSN 1895-1767, http://www.scpe.org
© 2022 SCPE. Volume 23, Issues 1, pp. 1–8, DOI 10.12694/scpe.v23i1.1954

CUDA IMPLEMENTATION FOR EYE LOCATION ON INFRARED IMAGES

SORIN VALCAN[*] AND MIHAIL GAIANU[†]

Abstract. Parallel programming using GPUs is a modern solution to reduce computation time for large tasks. This is done by dividing algorithms in smaller parts which can be executed simultaneously. CUDA has many practical applications especially in video processing, medical imaging and machine learning. This paper presents how parallel implementations can speedup a ground truth data generation algorithm for eye location on infrared driver recordings which is executed on a database with more than 2 million frames. Computation time is much shorter compared to a sequential CPU implementation which makes it feasible to run it multiple times if updates are required and even use it in real-time applications.

Key words: CUDA; parallel programming; infrared camera; driver monitoring; eye detection

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction.

Parallel processing is a modern way of accelerating the computation time for algorithms that can be divided into smaller parts, each of which is executed approximately simultaneously. This type of programming is most efficient when it is implemented on GPUs due to their special architecture that allows very fast execution of multiple threads.

This paper is focusing on programming using CUDA API provided by Nvidia for their GPUs. There are several CUDA applications that help researchers improve their computation time in areas such as bioinformatics, video processing, climate analysis, physics, gaming, machine learning and more. Time improvements are significant depending on the specific algorithm and how much it can be parallelized.

There are different forms of parallel computing such as instruction level, data parallelism or task parallelism. In GPU programming emphasis is on data parallelism because different sections of data can be processed in parallel using their multithreading capability.

This paper presents the use of CUDA in a ground truth data generator algorithm for eye location on infrared driver recordings which is presented in [1]. The purpose of this algorithm is to automatically generate good quality ground truth data that will be used for training neural networks in a system that will not require any human manual effort for data labeling.

In recent years, eye detection has become an important research topic in computer vision and pattern recognition ([3] [4]) , because the human eyes locations are essential information for many applications, including military, border control, facial expression recognition, auxiliary driving, and medical diagnosis [5]. For example, half of the face was covered in a cover test for detecting squint eyes [6].

Traditional eye detectors are typically designed according to the geometric characteristics of the eye. These eye detectors can be divided into two subclasses. The first subclass is the geometric model. Valenti and Gevers [7] used the curvature of isophotes to design a voting system for eye and pupil localization. Markuš et al. [8] proposed a method for eye pupil localization based on an ensemble of randomized regression trees. Timm and Barth [9] proposed the use of image gradients and squared dot products to detect the pupils. The second subclass is the template matching. The RANSAC [10] method was used to create an elliptic equation to fit the pupil center.

[*]Department of Computer Science, West University of Timişoara, 300223, Timişoara, Romania; Continental Automotive Romania VNI HMI, 300704, Timişoara, Romania(sorin.valcan96@e-uvt.ro).

[†]Department of Computer Science, West University of Timişoara, 300223, Timişoara, Romania; Continental Automotive Romania VNI HMI, 300704, Timişoara, Romania(mihail.gaianu@e-uvt.ro).

Multiple functionalities of our eye detection algorithm can be implemented using GPU programming and data parallelism concept. This has a big impact on the computation time required to generate ground truth data for recordings in our database and use them for training. It also makes it feasible to make updates for the algorithm and reprocess recordings in order to have better ground truth data without waiting weeks before starting the training process again.

The novelty of the paper is given by the remarkable capacity to reduce the time required for such an algorithm which is made up of multiple specific steps in order to have precise ground truth data.

2. Methods.

This section describes the functionalities that were implemented in parallel using CUDA API and the focus of the discussion is on how it reduces the computational time required for eye detection.

This algorithm is made up of multiple steps because the resulted data must be very precise in order to train a neural network. By bringing a data parallelism approach in specific functionalities for each step the time is reduced significantly.

The main approach is based on the fact that pixels of a two dimensional image can be processed individually and simultaneously for various functionalities which is much faster than a sequential approach where each step(pixel) waits for the previous one to be done. Using this technique the functionalities discussed in this section are processed much faster.

2.1. CUDA API.

CUDA API provides a general purpose programming model for the NVIDIA GPUs which helps the optimization and acceleration of algorithms that can be split in smaller tasks each of which can be executed in parallel.

2.1.1. Threads in CUDA.

From a software point of view parallel programming is about finding a logical way of executing an algorithm in different threads in order to make it run faster. The CUDA API organizes threads in grids and thread blocks as described in [2].

For our work we used only one dimensional grids with the number of thread blocks computed using the formula:

$$B = \frac{T}{512} + 1 \tag{2.1}$$

where B represents the number of blocks, T is the number of threads required for the current functionality that is executed and 512 is a constant used for the number of threads to be contained within one block. Most of the modern GPUs have a maximum number of threads per block of 512 or 1028.

From a hardware point of view we are interested in understanding how the number of Streaming Multiprocessors (SM) and the warp size influences the way we should organize our algorithms.

A GPU has a specific number of SM and RAM memory which determines its performance. RAM memory is just the amount that can be stored in the GPU memory at a time. A SM is a unit responsible for executing blocks of threads. Blocks are distributed to SM as described in [2].

The warp size is the number of threads from a block that a SM can execute simultaneously from a hardware point of view and its value is 32. The access to threads waiting to be executed is done very fast which makes the entire process very efficient. This is the reason why it is very important to have the number of threads in a block multiple of 32, otherwise for each block there will be a final warp execution where some threads are not used which is a waste of parallel computing power for the entire grid execution.

2.1.2. GPU hardware specifications.

For all computation times presented in our paper we used one GeForce RTX 2070 SUPER. This version has 40 multiprocessors which makes it very efficient because it can process up to 40 thread blocks in parallel. It also has 8 GB of RAM memory which allows it to have a big amount of data available on the GPU at a time but it does not have an influence on our algorithm performance because we use a small amount of memory.

2.2. Image representation in memory.

Our infrared driver recordings contain 2D grayscale images with resolution 1280x800. We use the concept of flatten array to process each pixel individually on GPU because it is much easier to execute one memory allocation of the entire pixel data instead of a separate allocation for each line.

Once we have the flatten array to work with, in case the original indexes of the 2D image are needed we can use the following formulas to compute them:

$$line = \frac{idx}{w} \tag{2.2}$$

$$col = idx - (line * w) \tag{2.3}$$

where idx is the current index in the flatten array and w represents the 2D picture width.

2.3. Time comparison definition. In the following sections we provide for each functionality a time comparison between the sequential CPU and parallel GPU implementation. Usually when GPU time is discussed we need to take in consideration all steps described in Figure 2.1.

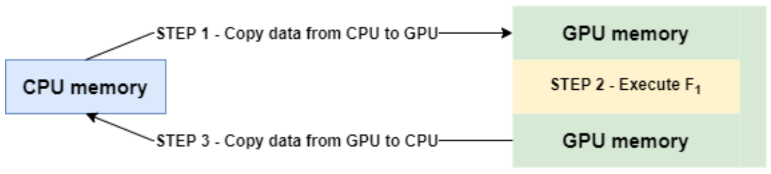
STEP 1 - Copy data from CPU to GPU
GPU memory
CPU memory
STEP 2 - Execute F$_1$
STEP 3 - Copy data from GPU to CPU
GPU memory

FIG. 2.1. *Steps needed for executing single functionality on GPU*

To exemplify this case we will consider a default time for data transfer between memories of 150 microseconds and for three GPU functions F_1, F_2, F_3 a time of 25, 30 and 35 microseconds respectively.

If for each individual function we will perform the three steps from above, we will have a complete execution time of 990 microseconds because the data transfer from CPU to GPU memory and back is executed three times

That is not the case in this paper because our entire algorithm is based on the structure defined in Figure 2.2.

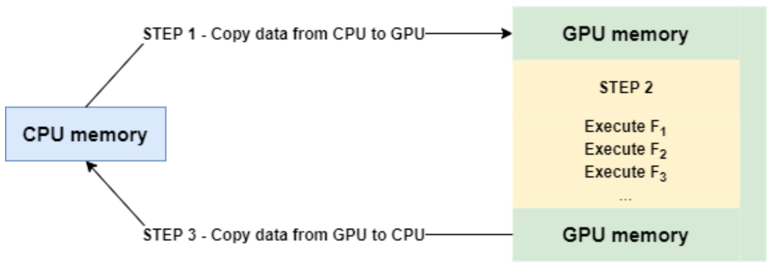

FIG. 2.2. *Steps needed for executing multiple consecutive functionalities on GPU*

When using this structure the time required to copy data between the two RAM memories has a much smaller impact on the total time required by the algorithm because data transfer must be performed only once(from CPU to GPU memory and back). Considering the same transfer and execution times from the previous example and running the GPU functions using the new sequence we will obtain a total execution time of 390 microseconds.

It is clear that the transfer time may vary depending on the size of the data being copied but the fact that we do not need to move data between the two memories multiple times makes the time improvement of each individual function F_n to be the main focus of this paper. This method of multiple function execution between two memory movements is a very efficient way of eliminating the big impact of data transfer time in GPU programming. It is understood that the design of the algorithm must allow such implementation which may not always be possible for various technical reasons. The opposite statement is also true that algorithms design should be thought in a way that allows the use of such an efficient implementation.

The functionalities presented in the following sections represent an individual function F_n contained within a bigger algorithm $\{F_1, ... F_n ..., F_m\}$ which makes the transfer time a different discussion subject. For each functionality we will provide the average time needed on 100 different frames.

2.4. Picture conversion using threshold. The entire algorithm is based on grayscale images obtained from the infrared sensor which are converted to black-gray-white using two thresholds which are dynamically computed for each separate frame or eye patch. This conversion is implemented using the function described in Algorithm 1 which is executed in parallel using one separate thread on GPU for each pixel.

Algorithm 1 Conversion from grayscale to black-gray-white

```
Require: $p$                                   ▷ Flatten array of pixel values
  $h, w$                                       ▷ Image height and width
  $al, ar, au, ad$                             ▷ Area limits where thresholds must be applied(left, right, up, down)
  $t1, t2$                                     ▷ Two thresholds for black-gray-white intervals
  procedure ApplyDoubleThreshold
      $index \leftarrow blockDim.x \times blockIdx.x + threadIdx.x$    ▷ Method to compute index of current thread in the flattened array
      if $index < w \times h$ then
          $line \leftarrow \frac{index}{w}$
          $col \leftarrow index - (line \times w)$
          if $al \leq col \leq ar$ and $au \leq line \leq ad$ then
              if $p[index] < t1$ then
                  $p[index] \leftarrow BLACK\_PIXEL$
              else if $t1 \leq p[index] \leq t2$ then
                  $p[index] \leftarrow GRAY\_PIXEL$
              else
                  $p[index] \leftarrow WHITE\_PIXEL$
              end if
          else
              $p[index] \leftarrow BLACK\_PIXEL$            ▷ If not in area, pixel becomes black
          end if
      end if
  end procedure
```

For our face area selection algorithm we use this functionality with the following boundary parameters:

- width: 1280
- height: 800
- up limit: 0
- down limit: 799
- left limit: 384
- right limit: 896
- t1 and t2 are dynamically computed for each frame in a separate function

This means there are 1,024,000 pixels to be processed. The time required for the sequential CPU implementation is 2.753 microseconds while the parallel GPU implementation requires 31 microseconds which makes it approximately 88 times faster.

2.5. Noise removal. This functionality is used to change the color of specific pixels in black-gray-white images according to the input parameters. One pixel will be marked with change flag if it is contained within one vertical and/or horizontal line with initial color that has pixels count less than an input maximum size. It is used in multiple algorithms like face area selection and eye selection for various logical reasons. The behaviour of the parallel implementation is described in Algorithm 2.

When this functionality is used in face area selection algorithm it has to mark 1,024,000 pixels but only between the limits described in Section 2.4. When it is executed using a sequential CPU implementation it

Algorithm 2 Mark noise pixels to be changed

```
Require: $p$                                   ▷ Flatten array of black-gray-white pixel values
  $h, w$                                       ▷ Image height and width
  $rmPx$                                       ▷ Flatten array of flags to mark color change needed
  $rmSize$                                     ▷ Maximum line size to be considered noise
  $vert, horiz$                                ▷ Boolean flags to search for noise on vertical, horizontal or both
  $searchColor$                                ▷ Color that will be replaced
  $al, ar, au, ad$                             ▷ Area limits where line dimensions are computed(left, right, up, down)
  procedure MarkPixelsToBeReplaced
    $index \leftarrow blockDim.x \times blockIdx.x + threadIdx.x$   ▷ Method to compute index of current thread in the flattened array
    if $index < w \times h$ then
      $line \leftarrow \frac{index}{w}$
      $col \leftarrow index - (line \times w)$
      if $al \leq col \leq ar$ and $au \leq line \leq ad$ then
        if $p[index] = searchColor$ then
          if $vert = true$ then
            $lineSize \leftarrow VerticalCount()$   ▷ Search up/down while pixels have searchColor
            if $lineSize \neq 0$ and $lineSize <= rmSize$ then
              MarkVerticalLineForReplacement(rmPx)   ▷ Same as VerticalCount() but it fills rmPx with replace flag
            end if
          end if
          if $horiz = true$ then
            $lineSize \leftarrow HorizontalCount()$
            if $lineSize \neq 0$ and $lineSize <= rmSize$ then
              MarkHorizontalLineForReplacement(rmPx)
            end if
          end if
        end if
      end if
    end if
  end procedure
```

requires 144.588 microseconds while the parallel GPU implementation requires 996 microseconds which makes it approximately 149 times faster.

When it is used in the eye selection algorithm it has to mark pixels on possible eye patches with resolution 70x70 which means 4.900 pixels. When it is executed using a sequential CPU implementation it requires 540 microseconds while the parallel GPU implementation requires 27 microseconds which makes it approximately 20 times faster.

2.6. Ratio map computation for eye patches. The ratio map is one of the scores we use to check if a possible eye patch actually contains an eye. This functionality marks each pixel with white in case it is contained within a horizontal and a vertical black line with ratio greater than 1.3. The advanced details of the ratio map score are presented in Section 2.4.1 from [1].

We implemented this functionality in parallel where a different thread uses a different pixel from the possible eye patch as a staring point to compute the ratio score. It is described in Algorithm 3.

We use this functionality only with the following parameters:

- width: 70
- height: 70
- searchColor: BLACK_PIXEL
- minRatio: 1.3

Algorithm 3 Compute ratio map for one possible eye patch

```
Require: $p$                                                  ▷ Flatten array of black-gray-white pixel values
  $h, w$                                                      ▷ Image height and width
  $outMap$                                                    ▷ Flatten array of marked pixels
  $searchColor$                                               ▷ Color for area of interest
  $minRatio$                                                  ▷ Minimum ratio between horizontal and vertical
  $al, ar, au, ad$      ▷ Area limits where line dimensions are computed(left, right, up, down)
  procedure ComputeRatioMap
    $index \leftarrow blockDim.x \times blockIdx.x + threadIdx.x$   ▷ Method to compute index of current thread in the flattened array
    if $index < w \times h$ then
      $outMap[index] \leftarrow BLACK\_PIXEL$
      $line \leftarrow \frac{index}{w}$
      $col \leftarrow index - (line \times w)$
      if $al \leq col \leq ar$ and $au \leq line \leq ad$ then
        if $p[index] = searchColor$ then
          $verticalSize \leftarrow VerticalCount()$           ▷ Search up/down while pixels have searchColor
          $horizontalSize \leftarrow HorizontalCount()$       ▷ Search left/right while pixels have searchColor
          if $verticalSize \neq 0$ and $horizontalSize \neq 0$ then
            $ratio \leftarrow \frac{horizontalSize}{verticalSize}$
            if $ratio \geq minRatio$ then
              $outMap[index] \leftarrow WHITE\_PIXEL$
            end if
          end if
        end if
      end if
    end if
  end procedure
```

- down limit: 69
- up limit: 0
- left limit: 0
- right limit: 69

When this functionality is executed using a sequential CPU implementation it requires 452 microseconds while the parallel GPU implementation requires only 33 microseconds which makes it approximately 13 times faster.

3. Obtained Results. Until now we presented time improvements obtained for separate functionalities by implementing them in parallel using GPU and CUDA API. In this section we will present an overall time improvement for our ground truth data generator.

All these functionalities that we presented are individual functions which work more efficient in a parallel implementation compared to a sequential CPU version. They are all part of a big eye detection algorithm presented in [1].

For our overall time improvement computation we ran a version of the algorithm where only the three functionalities presented above are implemented in a sequential CPU way. The rest of the algorithm may still contain GPU implementations which are more efficient. In Table 3.1 we present the time improvements obtained only with the functionalities described in this paper.

We selected 10 random recordings with different number of frames and different number of ground truth data frames computed. We can see on the time improvement column that on average the GPU parallel implementation is 15.88 times faster than the sequential CPU implementation. For recordings where a bigger number of ground truth frames is generated like Rec 4 we can see the best improvements because the algorithm is searching in a smaller area when the eye locations for the previous frame is available which makes it more

TABLE 3.1
Time improvements for GPU implementations compared to sequential CPU

Recording name	Number of frames	Ground truth frames generated	CPU time(s)	GPU time(s)	X times faster	GPU average fps
Rec 1	5422	363	2418	162	14.92	33.46
Rec 2	3194	674	1433	95	15.08	33.62
Rec 3	1421	327	599	35	17.11	40.60
Rec 4	3103	2047	1429	80	17.86	38.78
Rec 5	5879	751	2509	173	14.50	33.98
Rec 6	1995	415	965	59	16.35	33.81
Rec 7	5934	321	2726	171	15.94	34.70
Rec 8	885	70	378	24	15.75	36.87
Rec 9	6320	714	2724	178	15.30	35.50
Rec 10	1737	609	736	46	16.00	37.76

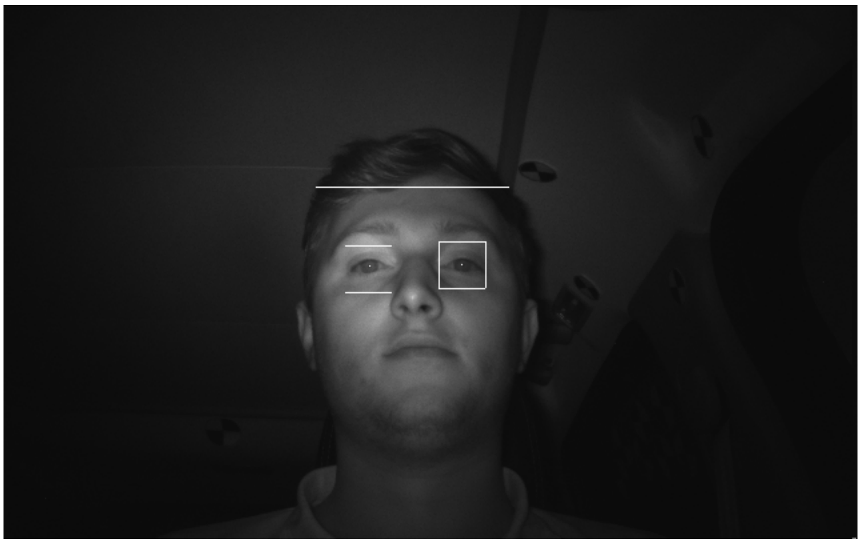
FIG. 3.1. *Example of eye labels ground truth data generated for one frame*

efficient.

This results have huge implications when we want to generate ground truth data on 2 millions frames (see example of eye labels in Figure 3.1). Just by taking the worst case of Rec 5 we can estimate the time required on GPU would take less than 17 hours compared to the CPU version of the algorithm which would take almost 10 days. This makes the entire process of ground truth data generation to be more feasible and more prone to improvements because it can be ran multiple times in a short period of time.

Some recordings have better improvements compared to others because there are less frames with face area detected. This leaves no room for time improvements in the eye detection functionalities because they never get to be executed. We can observe in cases like Rec 1 and Rec 5 that a small number of ground truth frames are generated which leads to a smaller time improvement. There are also cases like Rec 7 where we have a small number of ground truth frames but the time improvement is better. It means that face area was detected, the eye detection algorithm was executed but eyes were not found very often.

The GPU average fps column is computed by diving the number of frames to the GPU time required to process the recording. We can observe the minimum fps is 33 which makes the algorithm feasible to be run in real-time scenarios. This was not the main purpose of our parallel implementations but the performance one can obtain using parallel programming is remarkable.

4. Future Work. In future publications we will present GPU implementations we are using in the processing of possible eye patches. This requires the generation of the possible eye patches for a given area in a parallel way and storing it in a single continuous location of memory on the GPU. Once the eye patches are available we can use this memory location to perform steps for eye selection by processing each possible patch in a separate thread where this is possible.

In the future development of the ground truth data generator we want to include nostrils and mouth detection in order to have all face features available for training. Those algorithms will use similar parallel functionalities and we expect the time improvement of the entire detection(eyes, nostrils, mouth) to become even more significant compared to sequential CPU implementations.

5. Conclusions. This paper presented three parallel implementations which are used for eye detection on infrared driver recordings. The time improvement is significant and has a big impact on the entire process of ground truth data generation on a big number of frames. It also makes the algorithm feasible to be updated and re-executed in a short amount of time and it can be also used for real-time detection due to good fps performance.

Such parallel implementation methods can be used in several areas to reduce the time required for specific computations and have a major impact for big data processing where efficient algorithms help us save weeks of waiting time.

REFERENCES

[1] Valcan, S. and Gaianu, M., *Ground Truth Data Generator for Eye Location on Infrared Driver Recordings*, in Journal of Imaging 7, 162, doi: 10.3390/jimaging7090162, 2021.

[2] *CUDA C++ Programming Guide*, Available at https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] H. Fu, Y. Wei, F. Camastra, P. Arico, and H. Sheng, *Advances in Eye Tracking Technology: Theory, Algorithms, and Applicationst*, Computational Intelligence and Neuroscience, vol. 2016, Article ID 7831469, 2016.

[4] L. Zhang, Y. Cao, F. Yang, and Q. Zhao, *Machine Learning and Visual Computing*, Applied Computational Intelligence and Soft Computing, vol. 2017, Article ID 7571043, 2017.

[5] H. Mosa, M. Ali, and K. Kyamakya, LU-*A computerized method to diagnose strabismus based on a novel method for pupil segmentation*, in Proceedings of the International Symposium on Theoretical Electrical Engineering, 2013.

[6] Lorenz, Birgit and Moore, Anthony. (eds.) *Pediatric Ophthalmology, Neuro-Ophthalmology, Genetics*, doi: 10.1007/3-540-31220-X, 2006.

[7] R. Valenti and T. Gevers, *Accurate eye center location through invariant isocentric patterns*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 34, no. 9, pp. 1785–1798, 2012.

[8] N. Markuš, M. Frljak, I. S. Pandžić, J. Ahlberg, and R. Forchheimer, *Eye pupil localization with an ensemble of randomized trees*, Pattern Recognition, vol. 47, no. 2, pp. 578–587, 2014.

[9] F. Timm and E. Barth, *Accurate eye centre localisation by means of gradients*, in Proceedings of the 6th International Conference on Computer Vision Theory and Applications, VISAPP 11, pp. 125–130, 2011.

[10] L. Świrski, A. Bulling, and N. Dodgson, *Robust real-time pupil tracking in highly off-axis images*, in Proceedings of the 7th Eye Tracking Research and Applications Symposium, ETRA 2012, pp. 173–176, 2012.

Edited by: Marian Vajtersic
Received: Dec 17, 2021
Accepted: Apr 6, 2022