A Novel Method for Robust Multi-Directional Image Projection Computation

Gábor Kertész, Sándor Szénási, Zoltán Vámossy
John von Neumann Faculty of Informatics
Óbuda University
kertesz.gabor@nik.uni-obuda.hu, szenasi.sandor@nik.uni-obuda.hu, vamossy.zoltan@nik.uni-obuda.hu

***Abstract*—This paper introduces a novel method to calculate multi-directional projections of squared images, similar to the Radon transformation. Image projections are often used as object signatures for detection, matching and tracking techniques in computer vision. The Radon transformation provides a fast solution to calculate these pixel intensity sums. The proposed method is based on trigonometric functions and basic coordinate-geometry. The solution is implemented sequentially and the runtimes of a GPU-based implementation are measured and evaluated. The analysis of the results indicate that further research is applicable, new parallel models should be discussed.**

I. Introduction

In computer vision, the detection and tracking of defined objects can be done using several methods, lower-level data such as color or lightness or high-level profiles based on shape, texture or keypoints.

It is well-known, that the most precise methods have the highest computational cost: real-time applications are not available.

A common approach is to use a simple, fast method, and if applicable, multiply the numbers with different scales and angles. As a result, higher precision is obtainable.

A classic approach for object detection is template matching [1], where a sample of the visual representation of the object of interest is available, and this sample is tested for all possible positions. The comparison of the reference sample and the template image is done pixel-by-pixel, so the computational costs are high, however the method is highly parallelizable [1] [2]. Nevertheless, the method itself is very sensitive to rotational or scalable changes, the practical usability of this technique is limited.

An other, low-level object detection method is to compare image projections of the reference and the template images. With additional functions the effects of rotation or scaling can be reduced.

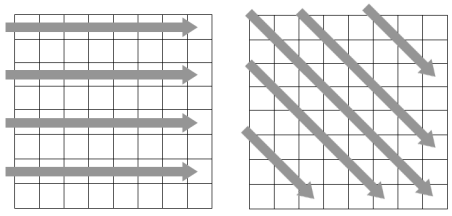

Fig. 1. (a) Horizontal image projection (b) Diagonal image projection

II. Image projections

Horizontal and vertical image projections can be calculated with ease, basically the rows and columns are summarized (Fig. 1(a)):

$$PR_H(i) = \sum_{j=1}^{N} P(i,j), \qquad PR_V(j) = \sum_{i=1}^{N} P(i,j), \tag{1}$$

where P represents the image as an $N \times N$ matrix of values where $\forall i \forall j P(i,j) \in \mathbb{N}$ for each row i and column j.

It is possible to calculate diagonal projections, as seen on Fig. 1(b). Diagonal projections result in a vector with a longer length than the horizontal or vertical, and the number of elements included in the accumulations vary. It is notable that these projections will differ based on the direction of the projection line.

Such image projections are often used to define the visual representation of objects by combining these vectors to so-called object signature. Usually the horizontal-vertical pair is used, however as seen in [3], the usage of diagonal and anti-diagonal signatures provide better results.

The idea of calculating the projections of an image from multiple angles beyond the defined main directions is a classic way to simulate the operations of the CAT (Computer Axial Tomography) scanner. The CAT scanner (or CT) is a device that takes X-ray images of a body from different angles [4], and the results are merged into a 3D model [5].

The method that reconstructs the original object from these measured values is the inverse Radon-transformation. The Radon-transformation [6] itself is a method that defines a set of vectors, the projected data from different angles.

The base of the Radon transform is an integral function, which is defined by a line integral on $L \in R^2$ as

$$Rf(L) = \int_L f(x)|\mathrm{d}x|. \tag{2}$$

Line L can be parametrized as $(x(z), y(z))$ where functions x and y can be turned into a trigonometrical expression based on the rotation matrix

978-1-5090-1216-9/16/$31.00 ©2016 IEEE

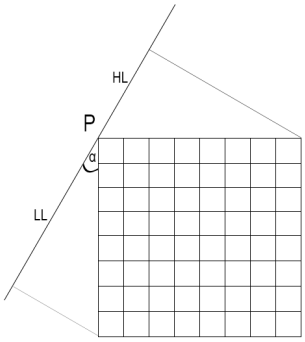

Fig. 2. Calculation of the projection length based on trigonometric expressions

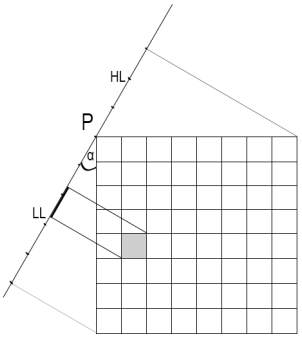

Fig. 3. Defining the projection positions of the pixel

$$R(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}, \tag{3}$$

resulting in

$$\begin{aligned} Rf(\alpha, s) &= \int_{-\infty}^{\infty} f(x(z), y(z))\mathrm{d}z \\ &= \int_{-\infty}^{\infty} f(z\sin\alpha + s\cos\alpha, -z\cos\alpha + s\sin\alpha)\mathrm{d}z, \end{aligned} \tag{4}$$

where s is the distance of L from the origin and α is the angle of the normal vector of L with the x axis. Please note, that the changes of signes cause the counterclockwise rotation change to clockwise.

The implementations of the Radon transformation are well-known [6], even data-parallel implementations exist [7].

III. Methodology

The Radon transform itself defines a great solution based on the rotation transformation, however these high-cost functions are not necessary to calculate a rotated projection itself.

The differing project vector length results in a phenomena called *sinogram*: the distribution and displacement of the values vary like the sinus wave [8]. This analogue approach could be left, since the input is a rasterized image, which elements are pixels.

Since each pixel is technically a small square, and that each pixel is the smallest visualizable element, it is easily concluded, that projections of an image result in a set of vectors where the length of each result vector differs, and for each length $l \in \mathbb{R}$.

For further calculations we realized, that a standardized scale should be used, where the length of each vector is equal, and $l \in \mathbb{Z}$.

To accomplish this, first we need to define the scale. Since the starting point is the horizontal and vertical projections, where the result vector length for both functions are N for an $N \times N$ squared image, we decided to use this length for each rotated projection. To accomplish this, a division of the line segment of the projection is done.

In our method we place a line to the left side of the image (Fig. 2), and rotate it with α degrees around a fixed point P in the upper left corner of the image. $\alpha \in [0, \frac{\pi}{2}]$.

As a result of the rotation we can see that the lowest and highest points of the projection define two line segments around P: each length can be derived using trigonometry functions as

$$\begin{aligned} LL &= \cos\alpha \times N, \\ HL &= \cos\left(\frac{\pi}{2} - \alpha\right) \times N = \\ &= \sin(\alpha) \times N. \end{aligned} \tag{5}$$

The scaling could be done likewise: the total projection segment length $LL + HL$ is divided into N different parts. Finally the $(x; y)$ projection position of each pixel is calculated by

$$\begin{aligned} Start &= \sqrt{(x+1)^2 + y^2} \times \cos(\arctan(y, x+1)), \\ End &= \sqrt{x^2 + (y+1)^2} \times \cos(\arctan(y+1, x)) \end{aligned} \tag{6}$$

as seen on Fig. 3.

Since each pixel is possibly projected into two subsegments on the projections line, each pixel modifies the accumulated value of two values in the result vector. The result value is increased with the ratio of the projection.

Algorithm 1 Method to calculate Multi-Directional Projections of an Image

```
procedure MultiDirectionalProjections(I, N, R)
    for deg := 0 → 90 do
        LL ← cos(deg) * N
        HL ← sin(deg) * N
        res ← (LL + HL)/N
        for all p ∈ I do
            start, end ← PositionOf(p.X, p.Y, deg)
            RationalAcc(R, res, start, end, p)
        end for
    end for
end procedure
```

In general, our method calculates the positions and ratios for each pixel, and increases the accumulator vector for each angle.

In Algorithm 1, results for each angle from 0 degree to 90 degrees are computed for an $N \times N$ sized image I, and stored in result matrix R. Values of the higher and lower projection segments are calculated as LL and HL as defined in Equation 5. The resolution of the segment res is defined by dividing the projection into N pieces. After these variables are set, the calculation of each pixel p in image I is done. First, the position of the projection is calculated using the function called POSITIONOF, which is based on Equation 6. Finally, the correct values of R are increased by the correct ratios of the value of p pixel, represented in function RATIONALACC.

IV. PARALLEL IMPLEMENTATION

Since the method uses input matrix itself for the calculations, and the result value for each pixel is not affected by the other pixels, parallel calculations could be done.

The main idea behind GPGPU (General-Purpose computing on Graphical Processing Units) is to use the architecture of a massive number of processing units to solve computationally intense cases. Practically the best-case usage of these devices are in multi-dimensional matrix operations, such as this problem.

Please note, there are techniques to automatically generate code for the GPU application [9], however the object of this paper is to present the results of a native implementation.

The first step to run a process on the GPU is to transfer the input data from the memory of the host computer to the memory of the graphic processor. This memory transfer time, increased with the copying of the results back to the memory of the host should be taken into consideration.

When designing a kernel procedure which is implemented on a graphics processor, two main restrictions should be kept in mind: first of all, the ability to divide the task, based on the input data is necessary: multiple blocks and multiple threads on blocks should be used on parts of the main data to provide a robust solution. Also, the understanding of memory architecture [10] of the board is crucial: the data transferred from the host is located in the global memory of the device. However each block and each thread also have its own memory, and the access of these blocks are performing better than addressing the global memory.

In the proposed method the input image is divided into several smaller blocks. These blocks are transferred into the so-called shared memory of the GPU, which is accessed by the threads assigned to that block. After the transfer, each thread is assigned to one pixel of the input, and the results are calculated for each angle of rotation. The results are stored in the shared memory, and finally these results are accumulated into the global memory of the device.

The proposed method uses three different levels of the memory architecture: the variables stored in the memory of the device are indicated with a small G, those stored in the shared memory of each block are denoted with S, and the local variables of the threads are marked with a letter L.

Algorithm 2 Kernel procedure to calculate the image projection for multiple directions

procedure MULTIDIRECTION_KERNEL(blk, I_G, N, R_G)
  $I_S \leftarrow$ GETBLOCK($I_G, blk.X, blk.Y$)
  $R_S \leftarrow new\ array[]$
  $disp_S \leftarrow new\ array[]$
  for $deg := 0 \rightarrow 90$ **do**
    $LL \leftarrow$ COS(deg) $* N$
    $HL \leftarrow$ SIN(deg) $* N$
    $res \leftarrow (LL + HL)/N$
    $start, end \leftarrow$ POSITIONOF($blk.X, blk.Y + 1, deg$)
    $disp[deg] \leftarrow \lfloor start/res \rfloor$
  end for
  for all $t \in threads$ **do**
    for $deg_L := 0 \rightarrow 90$ **do**
      $LL_L \leftarrow$ COS(deg_L) $* N$
      $HL_L \leftarrow$ SIN(deg_L) $* N$
      $res_L \leftarrow (LL_L + HL_L)/N$
      $st_L, e_L \leftarrow$ GPOS($blk.X, blk.Y, t.X, t.Y, deg_L$)
      RATIONALACC($R_S, res_L, st_L, e_L, I_S[t.X]$)
    end for
  end for
  SUMMARIZATION($R_S, disp_S, R_G$)
end procedure

The algorithm (Algorithm 2) can be divided into four main parts: first, the shared memory is allocated, and initialized, the image part from I_G is transferred to I_S, and the result matrix R_S is created. Also, an array named $disp_S$ is defined: this will store disposition values of the lower left corner of the block calculated for each rotation angle.

After these offsets are computed, each thread t is assigned to each pixel of the local image part I_S, and the positions on the projection line are evaluated. This is necessary, so that the ratios could be selected correctly. With the exact starting position st_L the increasing of the correct result value could be done. The main operation of this function is an addition, which is done as an atomic operation.

Finally, after all threads finished, the values collected in R_S could be summarized to the global memory R_G with an atomic operation.

As mentioned above, the addition of the values are done with an atomic add function, which locks the accessed blocks in the global memory, therefore only one thread could modify them at a time. Using this function, the faulting value problem caused by the race condition [11] is managed.

V. TEST RESULTS

A. Hardware configuration

The defined sequential algorithm was implemented in the Microsoft .NET environment in C# language, and to implement the kernel of the parallel solution we used the nVIDIA CUDA environment in CUDA C [12] and C++.

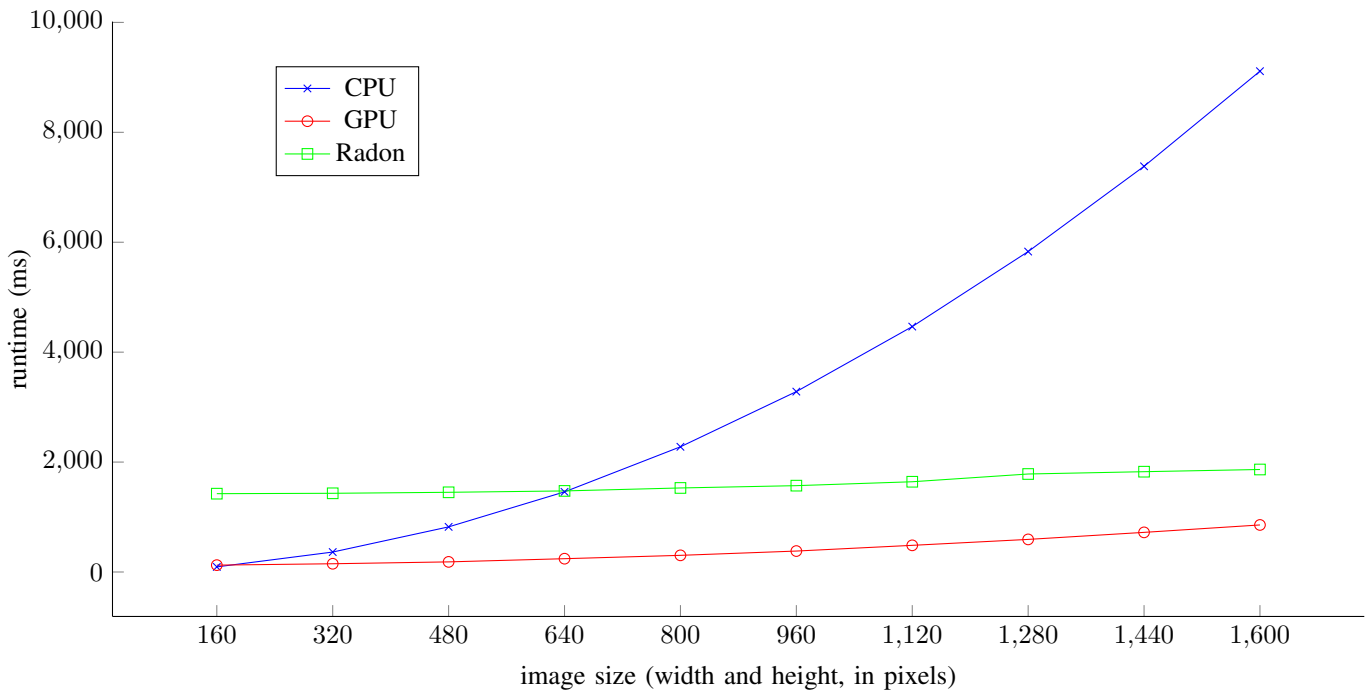

Fig. 4. Processing of the projection calculation methods on different image sizes

The following hardware configuration has been used during the tests:

- Processor: Intel Core i7-2600
- Architecture: Sandy Bridge
- Number of cores: 4
- Memory: 16 GB DDR2

During the testing of the GPU-accelerated implementation, we used the *nVIDIA GeForce GTX Titan X* videocard:

- Number of CUDA cores: 3072
- Memory: 12 GB GDDR5
- Architecture: Maxwell

The host computer of the graphics accelerator was the same computer mentioned above.

During the tests, we used different image sizes from 160×160 to 1600×1600. In the parallel solution, the local image size was 16×16, resulting with a number of 256 parallel threads for each block. The rotation of the line was done by degrees, starting from 0 to 90, resulting in a set of vectors starting with the horizontal, and ending with the vertical projection.

As the results in Table I show, the difference of processing times for small images (160×160, 320×320) not significant. However, the runtimes of the CPU implemented solution increase exponentially, while the runtime of the parallel method on the graphic processor increase linearly (Fig. 4).

The similar process times of small inputs are caused by the significant time wasted during memory transfer during initialization. When increasing the image size, these losses of time are overruled by the time profited by the multiprocessing of the blocks.

An interesting evaluation of the proposed method is by benchmarking the Radon transform function of *Matlab* and comparing it with the results of the GPU implementation of the method introduced in this paper (Table II). The test inputs and parameters were the same as in the cases above.

We assume, that the Matlab function is well-defined and multiple CPU-level parallelization methods are implemented to reach the highest available computational performance. Notable, that the Radon function results in more data than the method provided here, nevertheless the runtimes are slightly different.

TABLE I
PROCESSING OF THE PROJECTION CALCULATION METHODS ON DIFFERENT IMAGE SIZES

Reference size	CPU runtime (ms)	GPU runtime (ms)
160*160	93	125
320*320	363	150
480*480	821	184
640*640	1458	242
800*800	2278	303
960*960	3282	381
1120*1120	4465	485
1280*1280	5830	593
1440*1440	7381	721
1600*1600	9112	856

TABLE II
RUNTIME OF THE PROPOSED METHOD COMPARED TO THE RADON TRANSFORM FUNCTION IN MATLAB

Reference size	GPU (ms)	Radon transform (ms)
160*160	125	1425
320*320	150	1432
480*480	184	1450
640*640	242	1475
800*800	303	1529
960*960	381	1571
1120*1120	485	1642
1280*1280	593	1783
1440*1440	721	1825
1600*1600	856	1865

VI. CONCLUSION

In this paper we introduced a novel method to calculate the projections of an image from multiple directions. The implementation was done sequentially and data-parallel way, in a multiprocessor environment provided by the graphical processors.

The tests indicate, that the solution based on the GPU provides remarkable performance, however, real-time applications are off the table.

Our further plans include the design of a method which defines image projections for every angle from 0 degree to 180 degrees, based on the current results.

Furthermore object detection applications based on this technique could be developed. There are several procedures to compare projections of two object, however a GPU-based implementation is reasonable because of the importance of performance. In addition, the cost of memory transfer could be handled, if the projection profiles of known objects could be stored in the memory of the graphical processor.

A. Acknowledgements

The authors would like to thank the members of the GPU Education Center (formerly known CUDA Teaching Center) at Óbuda University for their constructive comments and suggestions.

The authors would like to say thanks for the help of the Hungarian National Talent Program (NTP-HHTDK-0015-0036).

REFERENCES

[1] G. Kertész, S. Szénási, and Z. Vámossy, "Performance measurement of a general multi-scale template matching method," *INES 2015 - 19th IEEE International Conference on Intelligent Engineering Systems*, 2015.

[2] ——, "Parallelization methods of the template matching method on graphics accelerators," *CINTI 2015 - 16th IEEE International Symposium on Computational Intelligence and Informatics*, 2015.

[3] V. Jelača, A. Pižurica, J. O. Niño-Castañeda, A. Frías-Velázquez, and W. Philips, "Vehicle matching in smart camera networks using image projection profiles at multiple instances," *Image and Vision Computing*, vol. 31, pp. 673–685, 2013.

[4] Á. Takács, D. Á. Nagy, I. J. Rudas, and T. Haidegger, "Origins of surgical robotics: From space to the operating room," *Acta Polytechnica Hungarica*, vol. 13, pp. 13–30, 2016.

[5] T. M. Lehmann, C. Gonner, and K. Spitzer, "Survey: interpolation methods in medical image processing," *IEEE Transactions on Medical Imaging*, vol. 18, pp. 1049–1075, 1999.

[6] S. R. Deans, *The Radon Transform and Some of Its Applications*, 1983.

[7] C. B. Mendl, "Real-Time Radon Transform via the GPU Graphics Pipeline," 2010. [Online]. Available: http://christian.mendl.net/software/radon_gpu_manuscript.pdf

[8] P. R. Edholm and G. T. Herman, "Linograms in Image Reconstruction from Projections," *IEEE Transactions on Medical Imaging*, vol. 6, pp. 301–307, 1987.

[9] R. M. Aciu and H. Ciocarlie, "Runtime translation of the java bytecode to opencl and gpu execution of the resulted code," *Acta Polytechnica Hungarica*, vol. 13, pp. 25–44, 2016.

[10] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on approach*. Morgan Kaufmann, 2010.

[11] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, p. 569, 1965.

[12] NVIDIA, *CUDA C Programming Guide*. NVIDIA Corporation, 2014.