Parallelization Methods of the Template Matching Method on Graphics Accelerators

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*—Template matching is a classic technique used in image processing for object detection. It is based on multiple matrix-based calculations, where there are no dependencies on partial results, so parallel solutions could be created. In this article two GPU implemented methods are presented and compared to the CPU-based sequential solution.**

I. Introduction

Graphical processors are generally applied in computationally intensive cases. The architecture of multiple processing units is providing a robust solution for calculations based on multi-dimensional matrix operations, such as computer graphics or image processing.
One of the main challenges in computer vision is to detect a described object in an input image: there are several solutions, mostly based on image keypoints [1], or color information [2]. Keypoint-based solutions [3] are serving the results usually faster than other approaches [4], however there are few occasions when these corner- and edge-based features could fail, especially on noisy and on artificially created smoothless pictures [5].
Template matching is a simple method, with several limitations: it is not able to detect rotated or scaled objects. However, as our former research indicated [6], a robust data-parallel multi-scaled method could be developed. This paper is a study on the parallelization of the general template matching procedure.

II. Template matching

Template matching is a procedure to find the visual representation of a defined template on a reference image [7]. The main idea behind the matching procedure is to calculate the summary of differences of the template image and the reference image on a pixel-level, and select the best value of the results.

The process is usually visualized with a template sized sliding window over the reference image (Figure 1). The sequential algorithm (Algorithm 1) shows the same idea. There are two arrays, which store the pixel values for the template (T) and the reference image (I), and the calculated sum values are stored in array R.
Please note, that while matrices could be programmatically represented by two-dimensional arrays, in the memory these

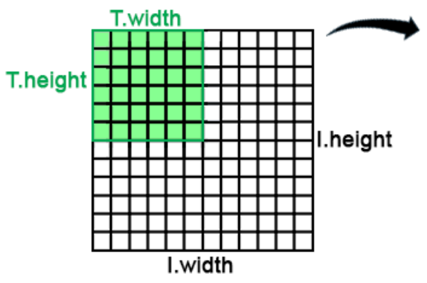

Figure 1. The template matching window (the green area) is moved over the reference image, and in each position the sum of differences is calculated.

indexes only mean a disposition from the starting byte, according to length of the selected data type. In this article the input or output matrices are handled as one-dimensional arrays.

Algorithm 1 Sequential template matching
```
function SeqTemplateMatching($(T, I)$)
    $R \leftarrow new\ array[I.width - T.width + 1 * I.height - T.height + 1]$
    for all $r \in R$ do
        $r \leftarrow$ SumOfDiffs($(T, I, r.id)$)
    end for
    return $R$
end function
```

The sum of differences could be calculated in a number of ways [8]: the most commonly used is the sum of squared pixel-differences. When using this method, the lower value indicates better matching. There are other methods, such as cross-correlation (Figure 2), where the sum of multiplied pixel values is calculated of the input images, and the higher value is the better matching. Because of the missing upper limits, both methods have they own disadvantages: in most cases the normalized versions are used.

III. Data-driven solution

There are no dependency between the values of results R, and reference image I and template image T are both used only as read-only data-structures, this hints that the

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2$$

$$R(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))$$

Figure 2. a) Squared differences, b) Cross correlation. Different methods to calculate the difference of selected pixels: x, and y are representing the location indexes of the upper left pixel of the window, x' and y' are dispositions from the starting position

$SumOfDiffs$ can be calculated parallelly for each element of the result array R.
Since the parallel solution will be implemented using a GPU, the memory transfer times to the device and moving the results back to the host computer should be taken into consideration [9].

A. Naive implementation

A trivial solution would be to calculate each element of R parallelly, by indexing the correct template-sized window on the reference image, and use the template matrix to summarize the differences. However, this solution would use too many threads, and each thread would try to reach out for each pixel data from the global memory of the device [10].
Another method would be to use different blocks to calculate the sum for each window, using different threads on each block to calculate the partial values of the summarization. This way the iteration inside the $SumOfDiffs$ function is parallelized. The backside is that each thread would try to reach out for the global memory of the GPU device, creating a waiting queue, breaching the real parallel execution. This intense memory call can be softened by moving relevant values from the global memory to the block-level shared memory, but this way the granularity of the thread jobs is too fine, and performance suffers from memory transfer and communicational overhead.
A more sophisticated method would be to calculate more than one value in a block, using a larger amount of shared memory in blocks, but fewer blocks and thread particles with medium granularity are used.
A trivial solution is to assign neighbour windows (for example, windows in the same row) to each block, and calculate the results, one after the other (Figure 3).

The kernel function for the multi-windowed template matching algorithm (Algorithm 2) gets template T and reference image I as inputs, and copies the template, and the relevant row of the reference to the shared memory as T_S and I_S. Each block has a unique identifier $blockId$, which is used to separate the rows. A local array is created in the shared memory as V_S to store the partial results from the calculation, and a shared array R_S is created to store the final sums. The outer loop moves the window in the row, the inner parallel loop calculates pixel differences for each position. After the threads finish, the temporary array stores the summary of the differences.

Algorithm 2 Multi-window template matching kernel procedure

```
procedure MultiWindowTemplMatch($T, I, R$)
    $T_S \leftarrow T$
    $I_S \leftarrow I[blockId * I.width \rightarrow blockId * I.width + I.width * T.height]$
    $V_S \leftarrow newarray[T.width * T.height]$
    $R_S \leftarrow newarray[I.width - T.width + 1]$
    for $i := 0 \rightarrow I.width - T.width$ do
        for all $v \in V_S$ do
            $v \leftarrow$ DiffCalc($I_S[(v.id/T.width) * I.width + (v.id\%T.width) + i] - T[v.id]$)
        end for
        $R_S[i] \leftarrow$ Sum($V_S$)
    end for
    $R[blockId * (I.width - T.width + 1) \rightarrow (blockId + 1) * (I.width - T.width + 1)] \leftarrow R_S$
end procedure
```

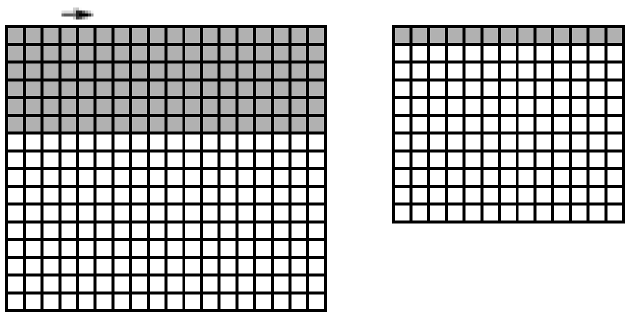

Figure 3. Template matching using a kernel function processing multiple windows. The matrix on the left represents the reference matrix, highlighted the values which are moved to shared memory array I_S of each block. On the right is the result array R, with grey background is the result array R_S of the block.

Finally, the global array R for results gets values from the shared memories of the blocks. This kernel should be loaded on $I.height - T.height + 1$ blocks on the graphics accelerator, $T.width * T.height$ threads on each.
The procedure only generates the values of each window in R, but does not select the best. In a sequential solution, it is trivial to use the maximum or minimum selection, depending on the used differential method. However, in this case parallel reduction [11] could be used to select the best values. This method uses multiple threads to compare values one-by-one. Our version of parallel reduction for maximum (Figure 4) selection starts with selecting values next to each other. The first and second value is compared, and if the second is bigger, than it is copied to the first place. This step is repeated, with the comparison of the best values from the last cycle.

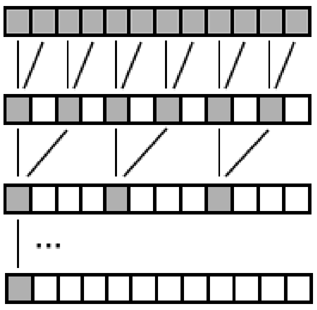
Figure 4. Parallel reduction for best value selection

B. A parallel model

An interesting approach is to assign each template point to a thread, and to calculate the difference with the reference pixel on each window parallelly. After this partial result is calculated, the corresponding values can be summarized, and copied to the global memory.

Algorithm 3 Template matching kernel procedure based on partial summarization

```
procedure PartSumTemplateMatching($(T, I, R, s)$)
    $V_S \leftarrow newarray[(I.widths - T.width + 1) * (I.heights - T.height + 1)]$
    $I_S \leftarrow$ PartOfArray$(I, s, blockId)$
    $t_L \leftarrow T[threadId]$
    $V_L \leftarrow newarray[(I.width/s - T.width + 1) * (I.height/s - T.height + 1)]$
    for all $v \in V_L$ do
        $v \leftarrow$ DiffCalc$(I_S[(v.id/(I.width/s - T.width + 1)) * I.width + (v.id\%(I.width/s - T.width + 1)) + ((threadId/(T.width/s)) * I.width/s) + (threadId\%(T.width/s))], t_L)$
    end for
    for all $v \in V_L$ do
        $V_S[v.id] \leftarrow V_S[v.id] + v$
    end for
    $R \leftarrow V_S$
end procedure
```

This kernel procedure (Algorithm 3) uses three levels of memory: the global memory of the device, accessible by all threads, the shared memory of the block, accessible by the threads in the same block, and the registers of each thread, accessible by only the thread itself. The reference matrix is divided horizontally and vertically by s, so there are s^2 number of pieces, each one assigned to a block.
First of all, the memory allocation should be done, V_S is created in the shared memory to store the final result values for each window, and the correct piece is copied into the shared memory array I_S from I. t_L is a local variable, a template point value assigned to each thread. A new array V_L is created in the local registers to store results of the difference calculations on each window.
After the parallel execution of each the $DiffCalc$s on template points, each thread copies the values stored in their local memory to the block-level shared memory. Please note, that this should be done with care, if multiple threads access the same element in the same time the results could be corrupted. After the results are summarized in V_S, these values can be moved to the global memory array R. After all blocks finish, parallel reduction could be used to select the best values.

IV. Test results

A. Hardware configuration

The defined algorithms was implemented in C++ language, the sources of both parallel methods are using the nVIDIA CUDA environment.
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 Tesla K40 Active videocard:

- Number of CUDA cores: 2880
- Memory: 6144 MB GDDR5
- Architecture: Kepler GK110

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

During template matching, squared differences was used to compare pixels, so when selecting the best match, we had to search for the minimum values.
When running the tests, first the sequential template matching algorithm was applied on the CPU and after it finished, processing times was also measured on the GPU accelerated versions.

Results show, that the multiple window based solution (Table I) is approximately finishes in 10-times less time as the sequential solution. However, it is very sensitive to reference and template matrix sizes, times increase noticeably in effect of increasing the size of the matrices.
As we expected, the method based on partial summarization (Table II) is only sensitive to the reference size, since blocks are processing the input data parallelly. However, it is notable, that if the size of the template matrix exceeds a sum of 1024 pixels, the runtime will start to decrease. The environment has a limit of a total of 1024 threads per block, so beyond this number, the template points will not have a dedicated thread.

TABLE I
PROCESSING TIMES OF THE MULTI-WINDOW TEMPLATE MATCHING METHOD ON DIFFERENT MATRIX SIZES

Reference size	Template size	CPU runtime (ms)	GPU runtime (ms)
80*80	8*8	24.96	12.4
160*160	8*8	110.29	10.9
240*240	8*8	255.53	15.6
320*320	8*8	460.83	15.6
80*80	16*16	78.78	10.9
160*160	16*16	394.06	14.1
240*240	16*16	949.73	18.7
320*320	16*16	1742.69	25.0
80*80	32*32	180.18	10.9
160*160	32*32	1251.13	20.3
240*240	32*32	3276.95	32.7
320*320	32*32	6260.79	48.4

TABLE II
PROCESSING TIMES OF THE TEMPLATE MATCHING METHOD BASED ON PARTIAL SUMMARIZATION

Reference size	Template size	CPU runtime (ms)	GPU runtime (ms)
80*80	8*8	24.96	10.9
160*160	8*8	110.29	14.0
240*240	8*8	255.53	21.8
320*320	8*8	460.83	45.2
80*80	16*16	78.78	10.9
160*160	16*16	394.06	9.4
240*240	16*16	949.73	21.9
320*320	16*16	1742.69	31.2
80*80	32*32	180.18	10.9
160*160	32*32	1251.13	17.2
240*240	32*32	3276.95	29.6
320*320	32*32	6260.79	57.8

V. CONCLUSION

We have implemented a multi-windowed algorithm, with less fine granularity than the naive solution. As our results show, the runtimes are mostly 10-times lower than the sequential method. The parallel method based on partial summarizations is also showing results with significantly lower processing times, with small sensitivity to the size of reference matrix.

As our measurements show, the optimalization of the memory transfer and communication operations would increase the performance. Our future plan is to optimize the code of the GPU implementation to maximize the usage of shared and local memory, and based on these new results further research on multi-scaled template matching could be reasonable.

Our future plans also include the research of object detection and matching on multiple smart cameras. There are similar micro-controller based solutions [12], where the results of the image processing could be done parallelly [13].

A. Acknowledgements

The authors would like to thank NVIDIA Corporation for providing graphics hardware for the GPU benchmarks through the CUDA Teaching Center program. The authors also wish to thank the members of the CUDA Teaching Center at Óbuda University for their constructive comments and suggestions.

REFERENCES

[1] J. C. Dunn, "Group averaged linear transforms that detect corners and edges," *Computers, IEEE Transactions on*, vol. C-24, pp. 1191–1201, 1975.
[2] A. E. Abdel-Hakim and A. A. Farag, "Color segmentation using an eigen color representation," *Information Fusion, 2005 8th International Conference on*, vol. 2, 2005.
[3] D. Stojcsics, "Autonomous waypoint-based guidance methods for small size unmanned aerial vehicles," *Acta Polytechnica Hungarica*, vol. 11, pp. 215–233, 2014.
[4] A. Rövid, "Machine vision-based measurement system for vehicle body inspection," *Acta Polytechnica Hungarica*, vol. 10, pp. 145–158, 2013.
[5] T. Hashimoto, T. Suzuki, H. Aoshima, and A. Rövid, "Multi-camera-based high precision measurement approach for surface acquisition," *Acta Polytechnica Hungarica*, vol. 10, pp. 139–152, 2013.
[6] 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.
[7] R. Brunelli, *Template Matching Techniques in Computer Vision: Theory and Practice*. Wiley, 2009.
[8] I. Corporation and Itseez. (2015) Opencv 3.0 online documentation. [Online]. Available: http://docs.opencv.org
[9] NVIDIA, *CUDA C Programming Guide*. NVIDIA Corporation, 2014.
[10] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on approach*. Morgan Kaufmann, 2010.
[11] M. Harris, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, 2007.
[12] G. Györök, "A special case of electronic power control of induction heating equipment," *Acta Polytechnica Hungarica*, vol. 11, pp. 235–246, 2014.
[13] D. Almási, C. Imreh, T. Kovács, and J. Tick, "Heuristic algorithms for the robust pns problem," *Acta Polytechnica Hungarica*, vol. 11, pp. 169–181, 2014.