# A improved GPGPU-accelerated parallelization for rotation invariant thinning algorithm

## Abstract

## Keywords

## 1. Introduction

第一段，细化定义介绍、细化算法的重要性

Thinning is the process of reducing the thickness of patterns to just a single pixel. Thinning is an important pre-processing step for many image analysis operation, such as image processing, character recognition, pattern recognition and so on.

第二段，细化算法的分类。细化算法分为两大类，parallel和sequential。Parallel下又能分为三类。

A comprehensive survey of thinning algorithms is described by Lam et al [8]. There two kinds of the thinning algorithms, the sequential thinning algorithms (STAs) and the parallel thinning algorithms (PTAs). As PTAs scan remove all the contour pixels that should be deleted in one iteration rather than only one pixel like STAs. So PTAs are usually faster than SPAS, but SPAs always produce better results.

第三段，随着图像大小的增大，目前细化算法的问题是运行时间过长，很难实现real-time。比如，Ahmed算法在细化2048\*2048大小图像需要十几秒。Kim在FPGA上实现了2D 细化算法并取得了很好的加速比，但是FPGA很难实现。Hu BingFeng在GPU上实现了3D细化算法，取得了152x的加速比。

The increase of image size, such as satellite image and medical image, result in the longer execution time of thinning algorithms. For example, thinning a 2D image(2048\*2048) takes about 17s on i7 CPU platform by using the A-W algorithm[2]. It is a challenge to implement thinning algorithms in real-time. K.Kim et al[3] implemented a PTA by using FPGA , but the implement is hard for general engineer and it limits the image size. Hu BingFeng et al[9] implement 12- subiteration parallel 3D thinning algorithm on GPU and the speed-up achieved was 152x.

第四段，简单介绍cuda，并根据实验得出并行细化算法在GPU上能够得到很好的加速比。但是A-W算法的结果并不理想，因为该算法内存在很多分支，影响性能。

With the multi-thread parallel processing power, many problems, such as large data sets and intensive computation, can be resolved efficiently on GPGPU. As s modern GPU architecture, CUDA is supported by Nvidia GPUs for general-purpose parallel computing. The parallel thinning algorithms can be well implement on GPU computation plat form owing to its data parallelism. We implement several PTAs on GPU. Figure 1 results show PTAs achieves a good performance on average 60 speedup. But the speedup of A-W algorithm is not ideal. Because there are too many branches which will lead a low warp efficiency.

Figure 1. The results of PTAs on GPU. (AW) Ahmed and Ward’s method. (ZS) Zhang and Suen’s method. (GH) Guo and Hall’s method. (PS) Petrosino and Salvi’s method.

在这篇论文中的主要贡献。1、证明PTAs在GPU上能够取得很好的加速比。2、在A-W的基础上，提出了一种新的并行策略，提高性能。3、我们讨论了性能与block大小、LookUp-Table的内存位置的关系。

In this paper, we manly improve AW algorithm with a novel Templates-to-Lookup Table(**TTL**) strategy. This paper makes the following contributions.

1. We implement several PTAs on GPU and proved that the PATs can obtain a good speedup on GPU (Section 1).

2. We present a novel Templates-to-Lookup Table (TTL) strategy based on AW algorithm on GPU (Section 3). The strategy can decrease branches.

3. We discussed in detail about the relationship between the speedup and image size, block size and lookup-table memory location,. We also talk about the robustness of this strategy. (Section 4)

论文余下部分的组织。

The rest of the paper is organized as follows. Section 2 briefly describe the AW algorithm. Then section 3 illustrates the detailed design of AW algorithm on CUDA. Experimental analysis and results are shown in Section 4 and finally we offer the future work and conclusion in Section5.

## 2. A rotation invariant ruled-based Thinning Algorithm

The AW algorithm is the first thinning algorithm focus on rotation invariant. The AW algorithm is an iteration parallel thinning algorithm. This paper proposed AW algorithm by discuss each condition of 8-neighbors of p and the central lines but disconnected condition logically.

There are three steps in AW algorithm.

For every pixel ***p*** do the following:

**Step1**. If p belongs to \*\*\*, stop calculations for this pixel. Otherwise, go to step 2.

**Step2**. If p belongs to \*\*\*, delete p. Otherwise, go to step 3.

**Step3**. Apply the 20 templates to ***p***. If 8-neighbors of p match one of templates, delete p.

By repeating the above steps until no changes occur.

## 3. A-W Thinning Algorithm on CUDA

We divide the AW algorithm into several parts, and design a novel structure on CUDA. The whole algorithm includes host-side (CPU) and device–side (GPU). It can be seen in fig.2 that the host side mainly calculate the lookup-table and preprocess input image, while the main part are parallelized on device-side, including step 1, step 2, step 3. In this paper, we focus on step 3 and apply strategy on step 3.

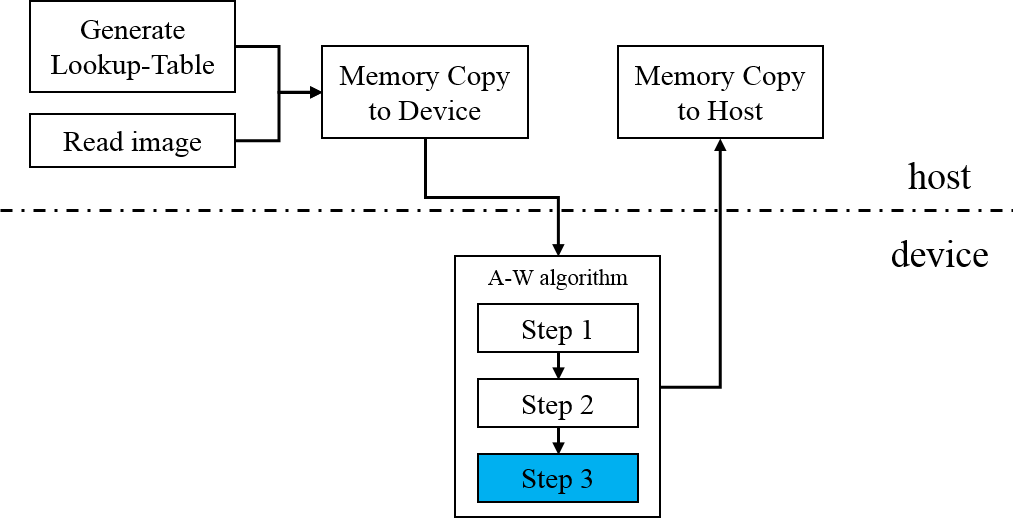


Figure 2. Structure of AW thinning algorithm on GPU

### 3.1 Definition

For every target pixel ***p***,

**Definition 1** N(p) is the set of all neighbors of p. For example 8-neighbors having even numbers(p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7]).

**Definition 2. Weight Number** [6]of p, WN(p), is defined:



For 8-neighbors of p, the weight numbers are among 0 and 255.

**Definition 3.** **Template** is the fixed set of 3\*3 neighbors of target pixel. For example, AW algorithm, have 20 templates. For example, if the 8-neighbors of p match the template in figure 5, then delete p.

|  |  |  |
| --- | --- | --- |
| 1 | X | 0 |
| 1 | p | 0 |
| 1 | 1 | x |

Figure 3. One of templates in AW algorithm.

### 3.2 Template-to-Lookup Table

在GPU内，分支对性能的影响。

In GPU, each 32 threads are grouped into warps. Warp instruction is the execution unit of GPU, which means all the threads in one warp execute the same instruction at any clock. Branch divergence has a significant impact on the performance of GPU programs. Branch divergence can hurt performance due to lower utilization of the execution units, which cannot be compensated for through increased levels of parallelism [1]. For example, in figure 4, where half of the warp threads evaluate the branch condition to true, the utilization of the execution units is only 50%.

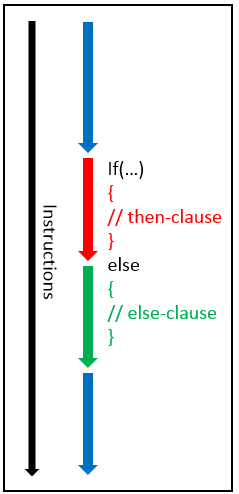


Figure 4. Example of branch

在A-W算法中，step 3理论上产生分支的最大深度为20，这大大降低了性能。

In AW algorithm, there are 20 templates in step 3. It will generate a lot of branches. When executing step 3, in the worst case, the utilization of the execution units is only 5%.

在本章节中，在A-W算法的基础上，提出了一种novel的方法，利用Weight number能够将template 转化为LookUp-Table

In this section, we propose a novel method based on AW algorithm. This method can transform templates into lookup-table. The Fig5 shows the transform code.

// 伪代码我还没怎么学会，在这里先用正常代码表示

Ts**[**20**]:** 20 templates

Np**[**8**]:**8**-**neighbors of p

char lookup**-**table**[**2**^**N**]** **=** **{0};**

**for** **(**int i **=** 0**;** i **<** N**;** **++**i**)** **{**

**if** **(**Template\_Match**(**Np**,** Ts**[**i**]))** **{**

WNp **=** Calculate\_Weight\_Number**(**Np**);**

lookup**-**table**[**WNp**]** **=** **1;**

**}**

**}**

Figure 5. transform templates into lookup-table

那么A-W算法的step 3就可以转变为，如下。

Then we could transform 20 templates in to a lookup-table[256] as follows.

Lookup-table**[**256**]** **=** **{** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 1**,** 1**,** 0**,** 1**,** 0**,** 1**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 0**,** 1**,** 0**,** 1**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 1**,** 0**,** 1**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 1**,** 0**,** 1**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 1**,** 0**,** 1**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 1**,** 0**,** 1**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 0**,** 1**,** 1**,** 0**,** 1**,** 0**,** 0**,** 0**,** 1**,** 1**,** 0**,** 0**,** 0**,** 1**,** 0**,** 1**,** 1**,** 1**,** 1**,** 0**,** 0**,** 1**,** 1**,** 0**,** 0 **};**

Then as shown in figure 6, we can re-design the step 3 of AW algorithm. In origin code of step 3, we have to match 8-neighbors of p with 20 templates in each thread. A lot of branches will lead low performance. Now, we transform 20 templates into one lookup table at host and in TTL code of step 3, we only need calculate the weight-number (WN) of p and then if lookup-table[WN] equal to 1, delete p. There is only one branch in step 3 of each thread.

|  |  |
| --- | --- |
| Origin code of step 3  **if** **(**match**(**Np**,** t**[**0**])** **||** match**(**Np**,** t**[**1**])** **||**  match**(**Np**,** t**[**2**])** **||** match**(**Np**,** t**[**3**])** **||**  match**(**Np**,** t**[**3**])** **||** match**(**Np**,** t**[**5**])** **||**  match**(**Np**,** t**[**4**])** **||** match**(**Np**,** t**[**7**])** **||**  match**(**Np**,** t**[**8**])** **||** match**(**Np**,** t**[**9**])** **||**  match**(**Np**,** t**[**10**])** **||** match**(**Np**,** t**[**11**])** **||**  match**(**Np**,** t**[**12**])** **||** match**(**Np**,** t**[**13**])** **||**  match**(**Np**,** t**[**14**])** **||** match**(**Np**,** t**[**15**])** **||**  match**(**Np**,** t**[**16**])** **||** match**(**Np**,** t**[**17**])** **||**  match**(**Np**,** t**[**18**])** **||** match**(**Np**,** t**[**19**])** **||**  **)** **{**  **delete** p**;**  **}** | TTL code of step 3  int WN **=** **(**x1**)** **\*** 1 **+** **(**x2**)** **\*** 2 **+** **(**x3**)** **\*** 4 **+** **(**x4**)** **\*** 8 **+** **(**x5**)** **\*** 16 **+** **(**x6**)** **\*** 32 **+** **(**x7**)** **\*** 64 **+** **(**x8**)** **\*** 128**;**  **if** **(**LUT**[**WN**]** **==** 1**)** **{**  **delete** p**;**  **}** |

Figure 6. re-design the step 3 of AW algorithm

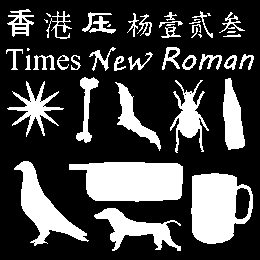
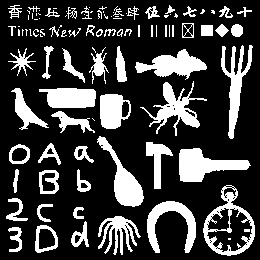
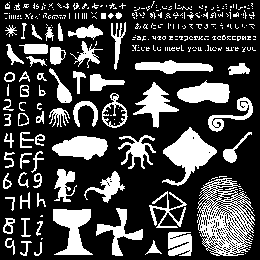
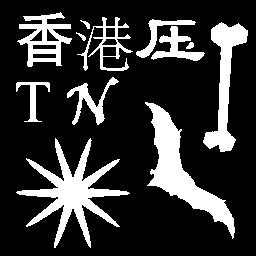
## 4. Experimental Results and Analysis

介绍实验图像和实验平台。

To evaluate our approach, we have chosen an Intel Core(TM) i7-2600 3.40GHz CPU and a NVIDIA GTX TITAN graphics card, using CUDA version 7.0. Four images are used in our experiment. Fig 7 show the images and table 1 show the detail of the images.

Table 1. The detail of images

|  |  |  |
| --- | --- | --- |
| 图像ID | 图像大小 | 图像内点数 |
| IA | 256\*256 | 11105 |
| IB | 512\*512 | 63150 |
| IC | 1024\*1024 | 215767 |
| ID | 2048\*2048 | 944177 |



fFigure 7. Four images are used in experiment

实验一，正常GPU加速和look up table方法加速的加速比。随着图像大小的增大，加速比越来越高。

Firstly, we compare the performance of the origin AW algorithm and TTL AW algorithm. As shown in fig 8, the average speedup of TTL achieve 30x(maximum 40.5x improvement) and better than origin on average 176%. With the increase of image size, the speedup get higher.

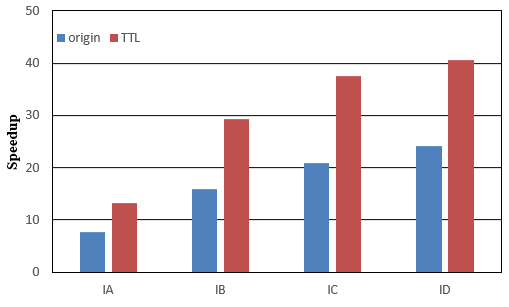


Figure 8. Speedup comparison of four images between origin and TTL.

实验二：应用TTL策略与直接GPU加速的加速比，并对比nvvp统计的数据，进行分析

We get the detail statistical data when executing the kernel using the NVIDIA Visual Profiler(NVVP). NVVP is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. We selected four criteria, Achieved Occupancy, Warp Execution Efficiency, Control-Flow Instructions, IPC. As shown in table 2, the TTL could decrease Control-Flow Instructions 465% which lead the higher Achieved Occupancy, higher Warp Execution Efficiency and higher IPC.

Table 2. The detail statistical data from NVVP

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Achieved Occupancy | Warp Execution Efficiency | **Control-Flow Instructions** | IPC |
| Origin | 0.534 | 37.3% | **3921276** | 1.859 |
| TTL | 0.607 | 59% | **842756** | 2.164 |

实验三：分析加速比和block大小的关系。TTL并没有占用太多的系统资源，对最佳线程块的影响是微不足道的。

Then, we analysis the relationship between performance and block size. Threads within a block can consume a fixed size registers and shared memory. The block size will impact the occupancy and the unsuitable block size will lead low utilization. The result is showed in fig 9. It shows the TTL not take up much system resource. The influence of best block size is insignificant. Origin AW algorithm’s best block size is 32\*8 and to TTL, it is 32\*4 on GeForce GTX TITAN.

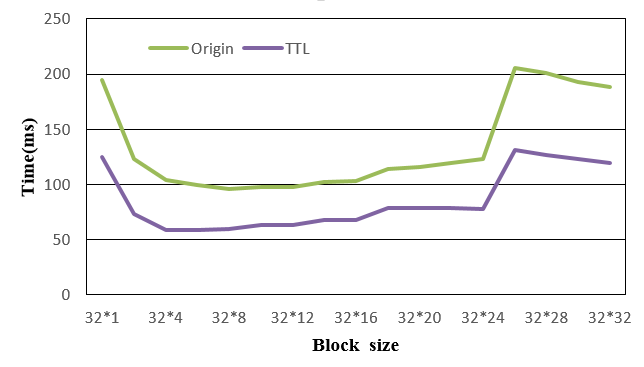


Figure 9. The relationship between block size and speedup

实验四：对比LookUp Table在Constant、Global、Shared内的速度

In GPU, global memory is off-chip. Shared memory and constant memory are on-chip and faster than global memory. We analysis the speedup when LUT located in global memory, shared memory and constant memory. As table 3 shown, the speedup in global is almost equal the speedup in constant and the best block size of both is 32\*4. That is because the throughput of constant memory is 4B per clock per SM, so unless the entire warp reads the same address, replays are need. The speedup in shared memory is worst and the best block is 32\*16. In TTL, it need copy LUT from global memory to shared memory which will have significant impact on logical complexity of the kernel. For another, each thread only read LUT once at step 3, LUT is not fit to locate in shared memory.

Table 3. The speedup when LUT located in global, shared and constant.

|  |  |  |
| --- | --- | --- |
|  | Best Performance | Block size |
| Global | 58.1382 | 32\*4 |
| Shared | 69.291 | 32\*16 |
| Constant | 58.48 | 32\*4 |

实验五：对比K40和TITAN上的加速比。K40的默认频率为745MHz，TITAN的默认频率是876MHz，并且K40有EEC部分来保证7\*24工作。所以K40的性能比TITAN低一些。

In addition, we executed the AW algorithm on diffident GPUs, Tesla K40c and GeForce GTX TITAN. Figure 10 shows the results. K40 get the lower speedup than TITAN. Because the default clock of K40 (745 MHz) is lower than TITAN (876MHz). Besides, the K40 have ECC part to guarantee 7\*24 working.

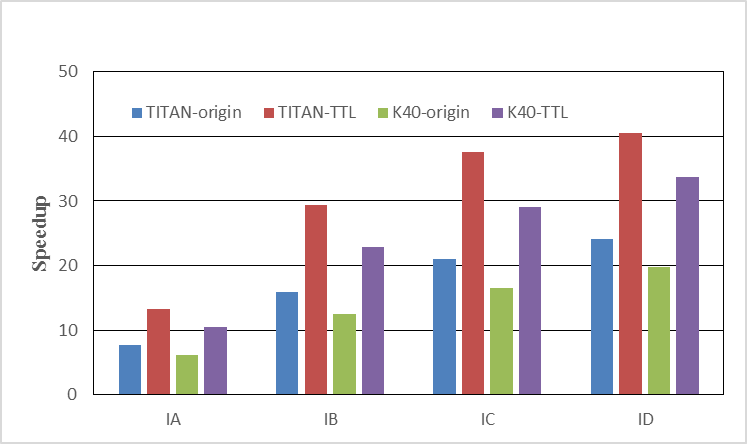


Figure 10. the speedups on K40 and TITAN

实验六：证明TTL的鲁棒性。TTL对于多分支算法具有显著的性能提升，并且TTL的消耗（内存拷贝和权值计算），对其他的少分支细化算法的影响微不足道。

Finally, we will discuss the robustness of TTL. TTL can improve performance of AW algorithm on GPU. We apply TTL to several others thinning algorithms with less branch. TTL have consuming in memory copy (copy the LUT from host to device) and calculate the WN(p) of p, but consuming and improvement can canceled out on other thinning algorithms. As figure 11 shown, TTL have insignificant influence on these algorithms.

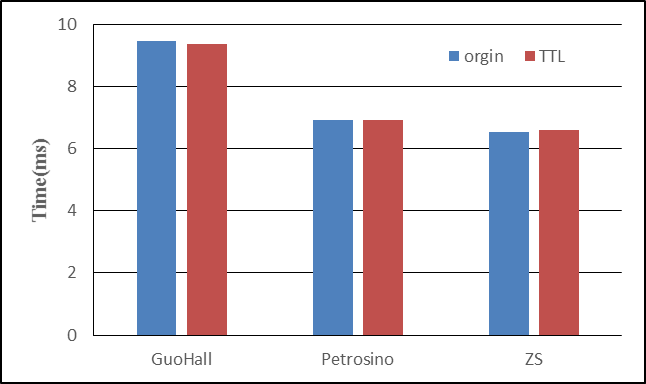


Figure 11. TTL on other thinning algorithms

## 5. Conclusion

Future work

该方法不仅适用于细化算法，在图像处理算法领域，只要满足根据领域特征进行复杂判断的图像算法都可以用该方法进行GPU加速

We call A-W thinning algorithm “branches-algorithms” which always need the feature of p’s neighbors to make decision. In the field of image processing, “branches-algorithms” is common.

In these algorithms, TTL will get a good speedup on GPU.

In this paper, firstly, we proved that thinning algorithms are suitable to implement on GPU. Second, we propose a novel parallel strategy based on A-W thinning algorithm using CUDA. This strategy can decrease the branch in each thread by transform Templates into Lookup Table. Then we analysis the relationship between speedup with block size. For data allocation, we compare the speedup of data storage in global, constant and shared memory. The experiment result shows the speedup of the proposed strategy can reach 40.5x . Moreover, we analyze the consumption of TTL and we conclude that our approach is not only more suitable with A-W algorithm on larger image, but also other “branchs-algorithms”.

## 7. Reference

[1] Han T D, Abdelrahman T S. Reducing branch divergence in GPU programs[C]//Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. ACM, 2011: 3.

[2] Ahmed, M. & Ward, R. A rotation invariant rule-based thinning algorithm for character recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence 24, 1672-1678 (2002).

[3] Kim, K.H.; Pham Cong Thien; Seung Hun Jin; Kim, D.K.; Jae Wook Jeon "Dedicated parallel thinning architecture based on FPGA", Multisensor Fusion and Integration for Intelligent Systems, 2008. MFI 2008. IEEE International Conference on, On page(s): 208 – 213

[4] Guo, Z., & Hall, R. W. (1989). Parallel thinning with two-subiteration algorithms. Communications of the Acm, 32(32), 359-373.

[5] Petrosino, A. ,., & Salvi, G. ,. (2000). A two-subcycle thinning algorithm and its parallel implementation on simd machines.. IEEE Transactions on Image Processing A Publication of the IEEE Signal Processing Society, 9(2), 277 - 283.

[6] Wang, P. S. P., & Zhang, Y. Y. (1996). A Parallel Thinning Algorithm with Two-Subiteration that Generates One-Pixel-Wide Skeletons. Pattern Recognition, International Conference on(Vol.4, pp.457). IEEE Computer Society.

[7] Zhang, T. Y., & Suen, C. Y. (1984). A fast parallel algorithm for thinning digital patterns. Communications of the Acm, 27(3), 236-239.

[8] Lam, L., Lee, S. W., & Suen, C. Y. (1992). Thinning methodologies-a comprehensive survey. IEEE Transactions on Pattern Analysis & Machine Intelligence*,* *14*(9), 869-885.

[9] Hu, B., & Yang, X. (2013). GPU-Accelerated Parallel 3D Image Thinning. High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC\_EUC), 2013 IEEE 10th International Conference on(pp.149 - 152). IEEE.