# GPU上基于旋转不变细化算法的并行加速策略

## Abstract

## Keywords

## 1. 引言

随着GPU的发展，GPGPU多线程并行处理能力的增强和CUDA等通用并行编程模型的出现，使其能够解决很多领域内的大型数据密集和计算密集的问题，如图像处理、科学计算等领域。

在图像处理中，图像细化算法为将形状宽度变为1像素宽度的过程。细化算法能够在保持形状骨架的基础上减少冗余信息，因此细化是很多复杂图像算法的一个重要的预处理步骤，如字符识别、模式识别等。Lam 等[8]于1992年调研了上百种图像细化算法，将图像细化算法分为两类：并行细化算法和串行细化算法。并行细化算法通过每次迭代删除边缘冗余点，串行细化算法则一次删除所有冗余点。并行细化算法通常比串行细化算法快，但串行细化算法往往能产生更好的结果。近年来，图像大小逐渐增大，如医学图像和卫星遥感图像，图像细化的处理时间也相应增加。大图像对细化算法的实时性来说是一个极大的挑战。举例说明，细化一个2048\*2048像素大小的灰度图像，在i7 CPU上，Ahmed和Ward（AW）细化算法[2]需要17s之久。

相关研究已经注意到细化算法的实时性问题。Zhang和Sune（ZS）等[7]在1984年提出了一种快速并行细化算法，通过迭代两子步骤删除图像中的冗余点。金汉均等[10]在2014年将ZS算法简单移植到GPU上。将算法简单移植到GPU上仅利用了GPU的硬件优势，往往不能最大化利用GPU的硬件优势。Hu BingFeng等[9]针对D12-PK-C 3D细化算法，根据GPU硬件特性对算法进行改进，取得152倍的加速比。

AW细化算法是首个旋转不变的并行迭代细化算法。如果原始图像旋转了一定角度，那么其细化结果也将旋转相同的角度。该算法很好的解决了断线问题，对噪声点具有一定的抗干扰能力。图1列出了简单移植并行细化算法到GPU上的加速结果，本文实验测试了四个并行细化算法。可见，ZS算法、GH算法和PS算法能够较好的利用GPU硬件优势，获得平均93.7倍加速比。而AW算法的加速比仅为21倍。原因是AW算法的旋转不变特性导致其算法流程比其他细化算法更加复杂，核心步骤中包含有大量条件判断语句。条件分支会降低GPU内核心利用率，对性能产生很大影响。

本文基于AW细化算法，提出了一种新的并行策略，将模板转化为查找表(Templates-to-Lookup Table, **TTL**)，该策略能够减少分支，提高资源利用率，得到更好的加速比。后面章节详细讨论了：

1. 基于AW细化算法提出了一种新的并行策略（TTL），该策略能够很好地减少分支；

2. 从细节上讨论了加速比与图像大小、线程块大小、查找表存储位置的关系，同时还讨论了TTL策略的在其他细化算法上的加速效果。

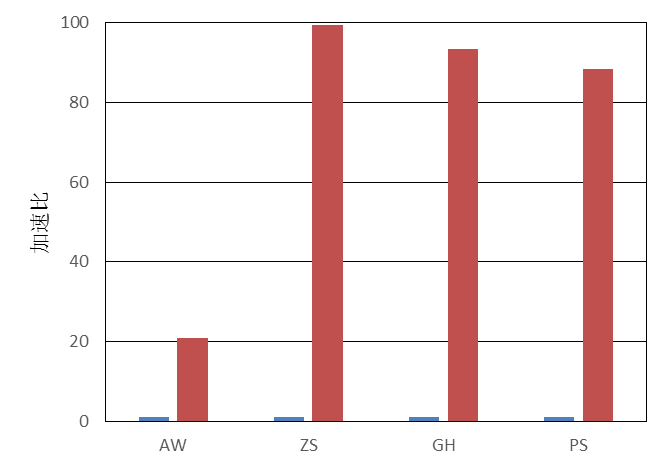


图 1 并行细化算法简单移植到GPU上的加速比。 (AW) Ahmed和Ward算法. (ZS) Zhang和Suen算法. (GH) Guo和Hall算法. (PS) Petrosino和Salvi算法.

## 2. 基于旋转不变的细化算法

对于目标像素点p，AW算法对p的八邻域节点的所有情况进行讨论，选取合理的情况删除p点（步骤一、三）。对于偶数像素宽度可能导致的断线问题，AW算法也给出了很好的解决方案（步骤一、二）。

AW算法共分为三个步骤：

**步骤一**：如果p的邻域符合图2中的(1)(2)(3)(4)(5)(6)情况，则停止计算。如果不符合，进行步骤二。

**步骤二**：如果p的邻域符合图2中(7)(8)情况，则删除p点。如果不符合，进行步骤三。

**步骤三**：将p的八邻域与20个模板进行匹配，如果符合任一模板，则删除p点。

对图像中的每一点迭代上述步骤，直至没有点要删除。

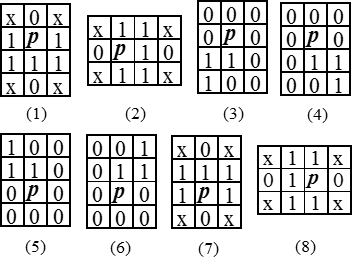


图2 AW算法中的模板

## 3. CUDA环境下AW细化算法思想

CUDA应用程序代码分为两部分，在CPU端运行的host端代码和在GPU上运行的device端代码。本文中AW算法在CUDA环境下的流程结构如图3所示，主机端的主要功能为计算查找表，图像预处理和内存拷贝；设备端端的主要功能为细化，按三个步骤分为三部分。本文将TTL策略应用到步骤三上。

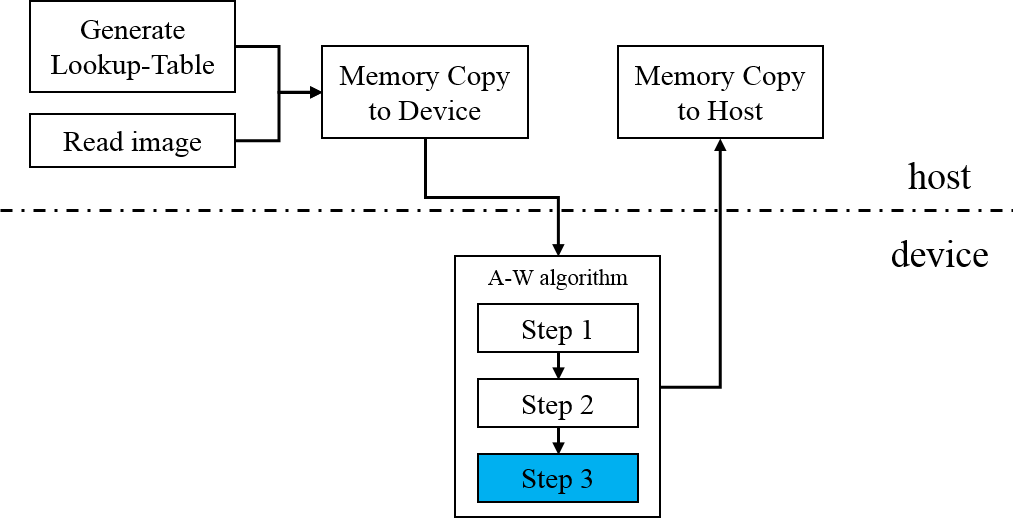


图 3 AW算法在GPU下的流程结构

### 3.1 相关概念定义

对于图像中的每一个目标点p，

**定义1** N(p)是p的邻域像素点的集合。如八邻域有八个像素点，此时N(p)={ p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7]}

**定义2** 权重值[6] WN(p)，计算公式如下



N代表N(p)内像素点数量。对于八邻域，WN(p)的值域为[0,255]

**定义3** 模板（template），是一组固定领域像素点，有3\*3模板，4\*1模板等。在AW算法中定义了20个3\*3模板，图4是其中之一，如果p的八邻域满足任一模板，则删除p点。



图 4 AW算法中的一个模板

### 3.2 模板转变为查找表

在GPU中，每32个线程被称为一个warp。warp是GPU中的执行单元，同一个warp中的所有线程在任何时刻执行相同的指令。CUDA程序中的条件分支对性能有很大的影响。条件分支会损害应用程序的并行性，降低执行单元的利用率，导致性能下降。举例说明，在图5中，假设warp中一半线程的分支条件为true，那么在then子句或else子句，某一时刻只能有一半的线程在运行，而另一半线程则处于等待状态，此时执行单元的利用率只有50%。

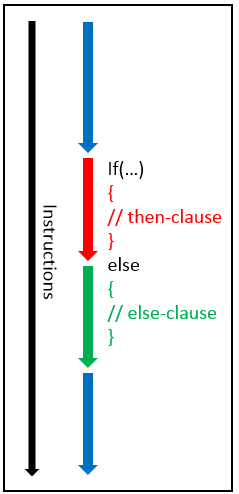


图5 条件分支对执行单元的影响

在A-W算法中，步骤三为将p的八邻域与20个模板进行匹配。则理论最坏情况下条件分支的最大深度为20，此时执行单元的利用率只有5%，大大降低了性能。

本文在AW算法的基础上，提出了一种新的并行策略，利用权重值原理，将模板转化变查找表（TTL）。转化步骤如图6所示。

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;**

**}**

**}**

图6 将模板转化为查找表

根据转化代码，AW算法中的20个模板可以转化为一个大小为256的查找表(Lookup-table**[**256**]**)。

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 **};**

根据得到的查找表，重新设计了AW细化算法中的步骤三。如图7所示，左边为原步骤3，需要将p的八邻域（Np）与20个模板进行匹配，如Np与任一模板匹配，则删除P点，这将会产生大量的分支。右边为TTL策略改进后的步骤三，只需要根据p点的八邻域计算权重值（WN(p)），然后在查找表中查找，如果lookup-table[WN]为1，则删除p点。改进后的步骤三中只有一个分支。

|  |  |
| --- | --- |
| 步骤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策略改进后的步骤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**;**  **}** |

图7 重新设计AW算法的步骤三

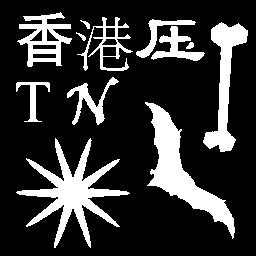
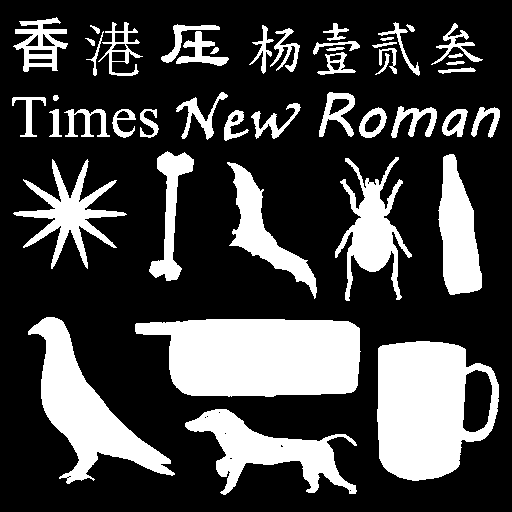
## 4. 实验结果和分析

本文实验共分为七部分，第一部分介绍了测试图像，实验一对比了AW算法在GPU上的普通实现和TTL策略实现的加速比，实验二根据NVVP获得细节数据说明TTL策略加速的原因；实验三讨论了性能和线程块大小之间的关系；实验四探究查找表位于不同内存上对性能的影响；实验五列出了在不同上GPU上AW算法GPU加速和TTL加速的性能；最后讨论了TTL策略在少分支细化算法上的加速效果。

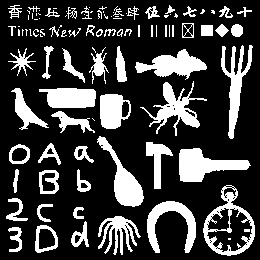
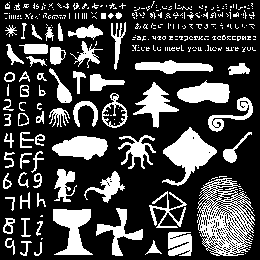
本文所有实验的硬件配置为NVIDIA GTX TITAN、Intel Core(TM) i7-2600 3.40GHz CPU和12G主机内存，软件环境为Ubuntu 14.04.2、CUDA 7.0。为了验证TTL策略的优点，实验中一共使用了四张测试图像，如图8所示。测试图像的详细参数见表1。

表1 测试图像的详细参数

|  |  |  |
| --- | --- | --- |
| 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 |

IA IB

IC ID

F图8 实验中使用的四幅测试图像

图9中列出了对四幅测试图像，AW算法正常GPU加速和TTL策略加速的加速比。TTL策略加速的平均加速比为30倍左右（最高40.5倍），比正常GPU加速的加速比高了176%。随着图像大小的增大，TTL策略的优势更加明显。

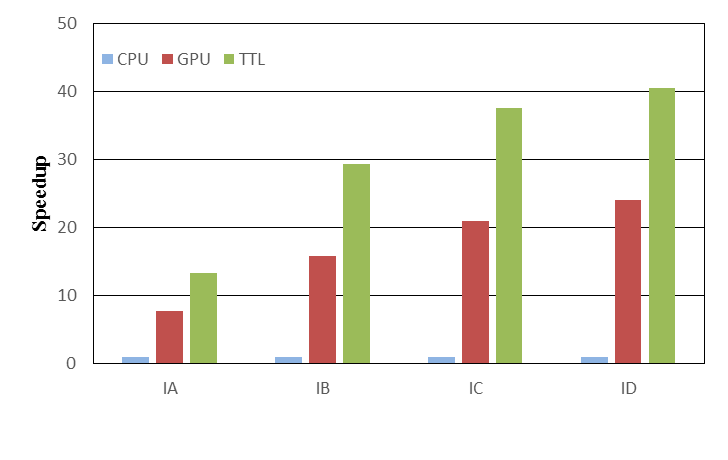


图9 AW算法在GPU上的普通实现和TTL策略实现的加速比

NVVP是NVIDIA研发的一个跨平台的性能分析工具，开发者可以根据NVVP得到的运行时的详细数据来优化CUDA程序。表2列出了NVVP获得的核函数执行过程中的细节数据。本文中选取了四个性能评价指标，核心占有率（Achieved Occupancy），warp执行效率（Warp Execution Efficiency,），控制流指令数（Control-Flow Instructions）和 时钟周期内执行指令数（IPC）。可见，TTL策略能够大幅度降低控制流指令数（465%），即降低了核函数内的条件分支数量，提高核心占有率、warp执行效率，获得更好的性能。

表2 核函数运行时的部分详细数据

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

在同一个线程块内的线程可以分享一定数量的寄存器和共享内存，线程块大小的设计不合理会影响核心内线程的执行数量，导致核心内资源利用率低。性能和线程块大小之间的关系如图10所示。可见，随着线程块大小的变化，TTL策略加速和GPU加速的性能走势基本相同。最佳线程块大小也相差不大，GPU版本的最佳线程块大小是32\*8，TTL版本的最佳线程块大小是32\*4。

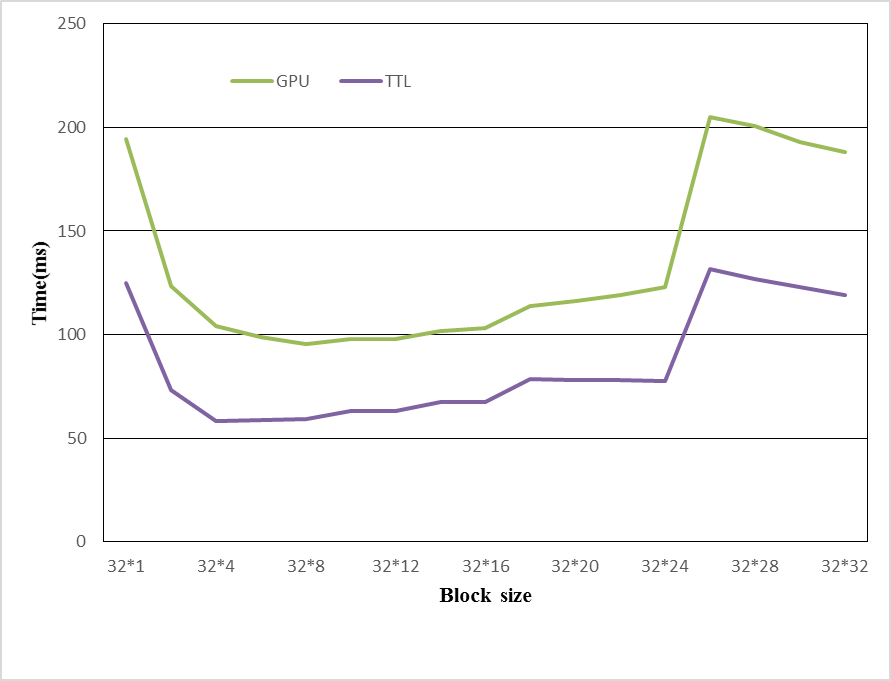


图10 性能和线程块大小之间的关系

在GPU中，全局内存（Global memory）属于片外内存，共享内存（Shared memory）和常量内存（Constant memory）是片上内存，在访问速度上共享内存和常量内存比全局内存快很多。在TTL策略中，每个有效线程需要访问一次查找表，所以核函数执行时间内，查找表会被频繁访问。为了探究查找表位于不同内存上对性能的影响，设计实验将查找表分别存放在全局内存、共享内存和常量内存中。实验结果见表3，查找表位于全局内存和常量内存的加速比基本相同，最佳线程块大小也相同。这是因为常量内存每一个时钟周期内的最大吞吐量为4B，如果一个warp内的所有线程所需数据大小大于4B，则需要多次读取。查找表位于共享内存的加速比较差，最佳线程块的大小为32\*16。这表明将查找表从全局内存拷贝到共享内存的过程对核函数内逻辑和资源分配有了较大的影响。在TTL策略中，每一个线程只需要读取一次查找表，所以查找表并不适合存储在共享内存中。

表3 查找表位于全局内存、共享内存核常量内存的最大加速比和对应的线程块大小

|  |  |  |
| --- | --- | --- |
|  | 最大加速比 | 线程块大小 |
| 全局内存 | 58.1382 | 32\*4 |
| 共享内存 | 69.291 | 32\*16 |
| 常量内存 | 58.48 | 32\*4 |

此外，本文探究了在不同上GPU上AW算法GPU加速和TTL加速的性能。本实验使用的GPU为Tesla K40c 和GeForce GTX TITAN。实验结果见图11，可以看出AW算法在TITAN上的性能更好。这是因为K40的默认核心频率是745MHz，TITAN的默认核心频率是876MHz，并且K40必须维持ECC功能以保证7\*24工作。

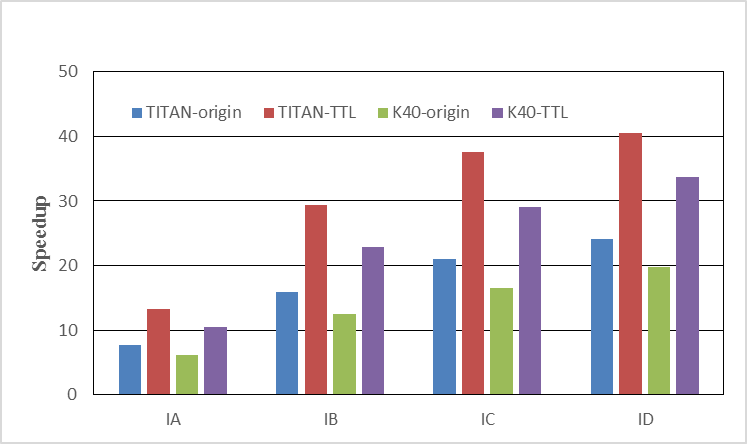


图11 AW算法在K40和TITAN上的加速比

AW细化算法属于“多分支算法”，这类算法的特点是需要根据目标点的邻域特征来决定下一步的操作，这类算法往往存在大量的条件分支，TTL加速策略能够有效地降低条件分支数量，取得很好的加速效果。为探究TTL策略在分支少细化算法上的加速效果，本实验选取三个少条件分支的细化算法，比较GPU加速和TTL加速的性能。如图12所示，两者性能几乎相等。这是因为TTL策略在查找表内存拷贝和计算WN(p)的消耗与性能提升相抵消。本实验证明TTL策略能够很好的加速多分支细化算法，而对少分支细化算法的性能几乎没有影响。在图像领域中，多分支算法非常常见，如高斯平滑、角点检测等。我们会在未来探讨TTL策略在这些算法上的加速效果。

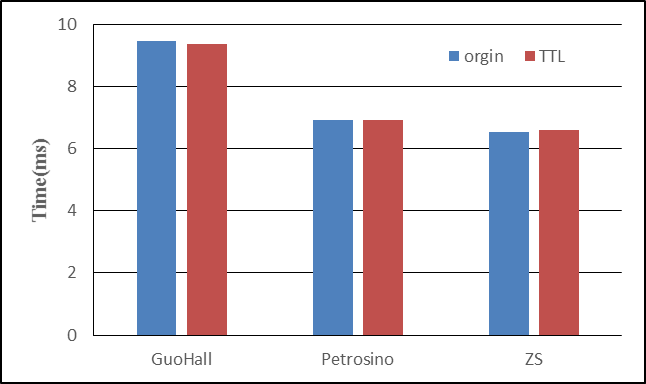


图12 在其他细化算法上应用TTL策略的加速比

## 5. Conclusion

随着信息时代的发展，图像大小也随之增长，这导致很难保证图像算法的实时性，利用GPGPU加速是解决这一问题的有效方法。在对AW细化算法进行深入研究的基础上，本文提出了TTL加速策略。该策略能够很好的降低算法中原有的条件分支，提高核心内资源利用率，获得平均30倍的加速效果。本文在实验部分继续深入讨论线程块大小、查找表存储位置与性能的关系以及TTL策略在分支少细化算法上的加速效果。

## 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.

[10] 金汉均, 梅洪洋. 基于CUDA的图像快速并行细化算法的研究与实现[J]. 电子测量技术, 2014, 08期(8):75-79.