รายงานการวิจัยฉบับสมบูรณ์

การศึกษาการใช้งานการ์ดแสดงผลภาพเพื่อเร่งความเร็วระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ

The Use of Graphics Processing Unit for Acceleration in the Learning Speed of Automatic Construction of Object Recognition Programs

ดร. อุกฤษฏ์ วัชรีฤทัย

ได้รับทุนสนับสนุนงานวิจัยจากเงินรายได้ ประจำปีงบประมาณ 2554
วิทยาลัยนานาชาติ
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ชื่อโครงการ การศึกษาการใช้งานการ์ดแสดงผลภาพเพื่อเร่งความเร็วระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ

แหล่งเงิน เงินรายได้ วิทยาลัยนานาชาติ

ประจำปีงบประมาณ 2554 จำนวนเงินที่ได้รับการสนับสนุน 50,000 บาท

ระยะเวลาทำการวิจัย 1 ปี ตั้งแต่ 1 ต.ค. 2553 ถึง 30 ก.ย. 2554

ชื่อ-สกุล หัวหน้าโครงการ และผู้ร่วมโครงการวิจัย พร้อมระบุ หน่วยงานต้นสังกัดและ อีเมล์

1. หัวหน้าโครงการ ดร. อุกฤษฏ์ วัชรีฤทัย วิทยาลัยนานาชาติ
โทรศัพท์ 02-329-8000 (ต่อ 2116) อีเมล์ kwukrit@kmitl.ac.th

บทคัดย่อ

งานวิจัยชิ้นนี้เกี่ยวกับ การเร่งความเร็วระบบสังเคราะห์โปรแกรมรู้จำวัตถุโดยอัตโนมัติ โดยใช้การ์ดแสดงผลภาพ หรือที่เรียกกันสั้นๆว่า GPU (Graphics Processing Unit) ระบบดังกล่าวทำงานโดยอาศัยคอมพิวเตอร์ โดยที่ผู้ใช้ (ซึ่งไม่จำเป็นต้องเป็นผู้เชี่ยวชาญ) เพียงป้อนฐานข้อมูลภาพที่ใช้การฝึกฝนโปรแกรมรู้จำวัตถุ (Training images) และ กำหนดฟังก์ชันจุดประสงค์ที่จะใช้วัดผลโปรแกรมเท่านั้น ระบบข้างต้นจะทำการสังเคราะห์โปรแกรมหลากหลายรูปแบบและทำการทดสอบ เพื่อหาโปรแกรมที่เหมาะสมในการแก้ปัญหานั้นๆ โดยทั่วไประบบดังกล่าวอาศัยเทคนิคการคำนวณเชิงวิวัฒนาการซึ่งเป็นเทคนิคการค้นและการหาค่าเหมาะที่สุดซึ่งมีแนวคิดมาจากการวิวัฒนาการของสิ่งมีชีวิต อย่างไรก็ตาม ระบบดังกล่าวมีจุดอ่อนที่สำคัญคือ เวลาการประมวลผลที่ยาวนาน เนื่องจากต้องรันโปรแกรมประมวลผลภาพเป็นจำนวนมาก

ผู้วิจัยได้ทดลองพัฒนาระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ ซึ่งรันชุดคำสั่งประมวลผลภาพบนGPU ในปัจจุบันGPU ซึ่งถูกออกแบบเป็นพิเศษเพื่อประมวลผลงานทางด้านคอมพิวเตอร์กราฟิกได้อย่างรวดเร็ว ได้ถูกพัฒนาให้สามารถโปรแกรมได้โดยผู้ใช้เพื่อให้สามารถประมวลผลงานด้านอื่นๆได้ การประมวลผลภาพและทัศนศาสตร์คอมพิวเตอร์ ซึ่งเป็นกระบวนการย้อนกลับของคอมพิวเตอร์กราฟิกก็สามารถที่จะถูกประมวลผลได้อย่างมีประสิทธิภาพบนGPUเช่นกัน จุดประสงค์ของงานวิจัยนี้คือการศึกษาการใช้งาน GPU เพื่องานด้านประมวลผลภาพ เพื่อนำมาเร่งความเร็วระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ ผู้วิจัยได้ทำการศึกษาโครงสร้างของ GPU และภาษา CUDA C ซึ่งเป็นภาษาคอมพิวเตอร์ซึ่งใช้ในการโปรแกรม GPU จากนั้นผู้วิจัยได้พัฒนาชุดคำสั่งประมวลผลภาพพื้นฐานซึ่งจะถูกรันบน GPU และทำการรวมโค้ดของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ กับชุดคำสั่งประมวลผลภาพซึ่งรันบน GPU เข้าด้วยกัน และทำการวัดประสิทธิภาพของระบบ

ผู้วิจัยได้ทำการทดลอง 2 การทดลอง ในการทดลองที่หนึ่งผู้วิจัยเปรียบเทียบเวลาการประมวลผลของชุดคำสั่งประมวลผลภาพแต่ละชุดซึ่งรันบน CPU และ GPU จากผลการทดลองพบว่าอัตราการเร่ง

ความเร็วแตกต่างกันขึ้นกับชุดคำสั่ง แต่อัตราการเร่งความเร็วจะสูงขึ้นเมื่อทำการทดลองกับภาพที่มีขนาดใหญ่ขึ้น เมื่อทำการทดลองกับภาพที่มีขนาด 2560 × 1920 จุดภาพ พบว่าอัตราการเร่งความเร็วแปรผันจากไม่กี่เท่าไปจนถึงแสนเท่าโดยประมาณขึ้นกับชุดคำสั่ง นอกจากนี้ผู้วิจัยได้วัดอัตราการเร่งความเร็วโดยเฉลี่ยของทุกชุดคำสั่งและพบว่าอัตราการเร่งความเร็วเฉลี่ยแปรผันจาก 5.92 เท่าถึง 88.52 เท่า ขึ้นกับขนาดของภาพที่ใช้ทดลอง ในการทดลองที่สอง ผู้วิจัยเปรียบเทียบเวลาการประมวลผลโดยรวมของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติซึ่งใช้และไม่ใช้ GPU โดยขนาดของภาพที่ใช้ในการทำทดลองคือ 160 × 120 จุดภาพ ผลการทดลองแสดงให้เห็นว่า ระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติที่ใช้ GPU สามารถเร่งความเร็วการทำงานได้ถึง 3 เท่า และจะมีค่าสูงกว่านี้หากทำการทดลองกับภาพที่มีขนาดใหญ่ขึ้น

คำสำคัญ: การ์ดแสดงผลภาพ โปรแกรมรู้จำวัตถุ การคำนวณเชิงวิวัฒนาการ การสังเคราะห์โปรแกรมอัตโนมัติ

Research Title: The use of graphics processing unit for acceleration in the learning speed of automatic construction of object recognition programs
Researcher: Dr. Ukrit Watchareeruetai
Faculty: International College **Department:** School of Engineering and Technology

ABSTRACT

This research focuses on the acceleration in the learning speed of an automatic object recognition program construction system by using graphics processing units (GPU). In such a system, a user (need not to be an expert) only inputs an image dataset of the given problem and defines the objective functions used for performance evaluation. Then the system will automatically construct a program that is proper for the given image dataset and the defined objective functions. Many systems proposed so far are based on evolutionary computation techniques, which are powerful search/optimization techniques inspired by biological evolution in nature. In this approach, a numerous number of image programs have been executed and evaluated to find the program proper for the given problem; consequently, it requires much computation time for program construction.

In this work, we have developed a program construction system that performs image processing operations on GPU. Recently, GPU which are specifically designed for computer graphics (CG) tasks such as rendering 3-D scene onto a 2-D screen have been exploited to accelerate computation of general tasks rather than CG. Image processing and computer vision, which are the inverse process of CGs, can be effectively computed on GPUs also. Therefore, if all image processing operations are efficiently computed on GPU, we would greatly reduce computation time of the object recognition program construction system. As the main objective of this work, we aim to study how to program a GPU to effectively perform image processing operations so that we can accelerate the speed of the object recognition program construction systems. So far, we have studied the structure of GPU and CUDA C, which is a programming language that can be program GPU. Then we have developed several basic image processing operations that can be run on GPU, and have combined them with the code of the program construction system which will be run on CPU.

In this research, we have conducted two experiments. In the first experiment, we compared the processing time of each image processing operation that was run on CPU and GPU. The experiment results show that the acceleration rate was different depending on the operation to be executed but it increases when we use the image with larger size. With the largest size image in the experiment (2560 × 1920 pixels), the acceleration rate varies from several times to a hundred thousand times, depending on the operation. In addition, we measured the average acceleration rate of all image processing operations. The results show that the average acceleration rate varies from 5.92 times to 88.52 times, depending on the size of the test image. In the second experiment, we have compared the overall processing time of an automatic object recognition program construction system with and without GPU. The experiment has been conducted by using a dataset of 160 × 120 pixel images. The results indicate that the use of GPU can help accelerate the learning speed of the object recognition program construction system around 3 times, compared with a program construction system without GPU, and the acceleration rate would be increased if we use the images with larger size.

Keywords: GPU, Object recognition program, Evolutionary computation, Automatic program construction

กิตติกรรมประกาศ

ผู้วิจัยขอขอบพระคุณบุคลากรของวิทยาลัยนานาชาติทุกท่านที่ช่วยอำนวยความสะดวกให้กับข้าพเจ้ารวมทั้งความช่วยเหลือต่างๆในการทำวิจัยนี้ การวิจัยครั้งนี้ได้รับทุนสนับสนุนการวิจัยจากสถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง จากแหล่งทุนเงินรายได้วิทยาลัยนานาชาติ ประจำปีงบประมาณ พ.ศ. 2554

ดร. อุกฤษฏ์ วัชรีฤทัย

สารบัญ

สารบัญตาราง

ตารางที่ **หน้า**

สารบัญภาพ

ภาพที่ **หน้า**

บทที่ 1

บทนำ

1.1 ความสำคัญและที่มาของโครงการ

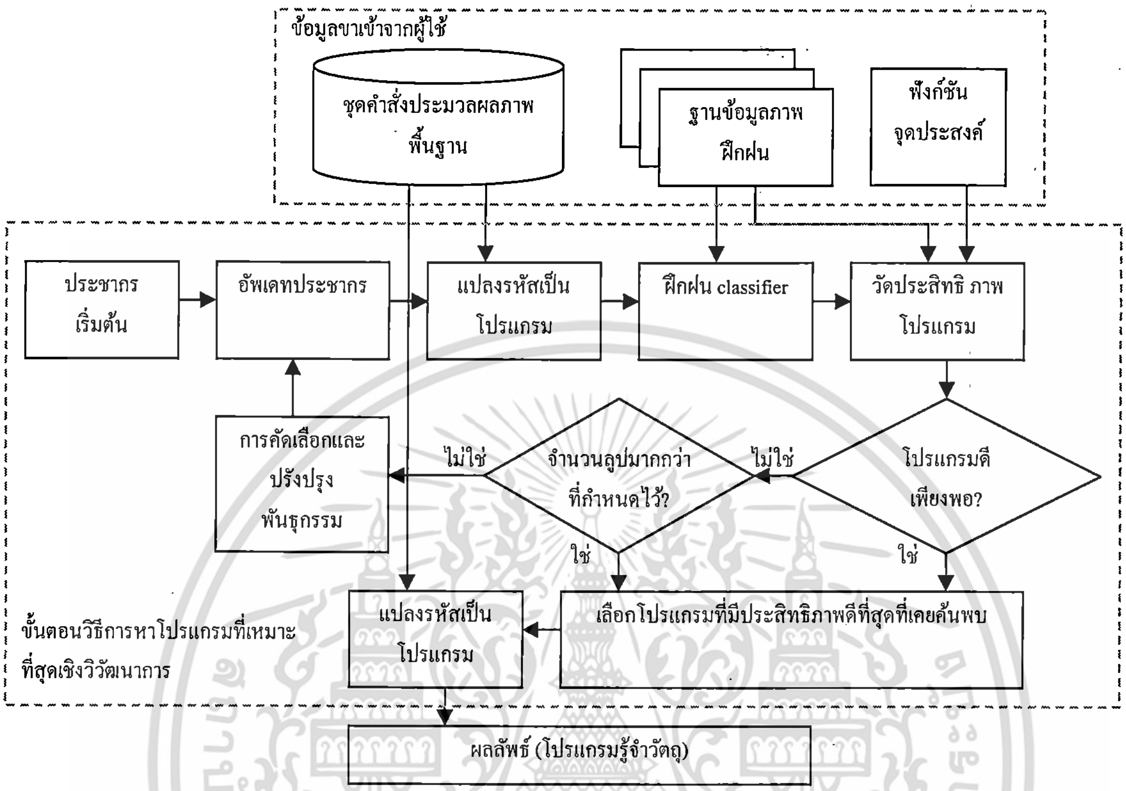
ในโลกปัจจุบัน โปรแกรมการประมวลผลภาพเพื่อรู้จำวัตถุ (Object recognition) เป็นที่ต้องการเพิ่มมากขึ้นทั้งในหลากหลายวงการเช่น อุตสาหกรรม (เพื่อสร้างระบบอัตโนมัติในโรงงาน การตรวจหาจุดบกพร่องในชิ้นงาน) การแพทย์ (ระบบช่วยวินิจฉัยโรคจากภาพถ่ายทางการแพทย์) การเกษตร (เช่นระบบพ่นสารกำจัดวัชพืชอัตโนมัติ) หรือ การรักษาความปลอดภัย (เช่นระบบรู้จำลายนิ้วมือ ลายม่านตา) การออกแบบโปรแกรมรู้จำวัตถุเพื่อแก้ปัญหาต่างๆนั้น ต้องอาศัยผู้เชี่ยวชาญในสาขา การประมวลผลภาพ (Image processing) ทัศนศาสตร์คอมพิวเตอร์ (Computer vision) และ/หรือ การรู้จำรูปแบบ (Pattern recognition) เป็นต้น นอกจากนั้นยังต้องมีทักษะประสบการณ์ในการวิเคราะห์ปัญหาและออกแบบโปรแกรม รวมทั้งอาจจะต้องการความรู้เฉพาะด้านเกี่ยวกับปัญหานั้นๆด้วย จำนวนผู้เชี่ยวชาญเหล่านี้ยังมีน้อยและไม่เพียงพอกับการพัฒนาที่รวดเร็วของโลกปัจจุบัน

นักวิจัยหลายกลุ่มจากทั่วโลก (รวมทั้งตัวผู้วิจัยเอง) ได้นำเสนอระบบเพื่อออกแบบโปรแกรมรู้จำวัตถุโดยอัตโนมัติ [1, 2, 6, 10, 16, 19-25, 27-30] ระบบดังกล่าวทำงานโดยอาศัยคอมพิวเตอร์ โดยที่ผู้ใช้ (ซึ่งไม่จำเป็นต้องเป็นผู้เชี่ยวชาญ) เพียงป้อนฐานข้อมูลภาพที่ใช้การฝึกฝนโปรแกรมรู้จำวัตถุ (Training images) และ กำหนดฟังก์ชันจุดประสงค์ที่จะใช้วัดผลโปรแกรม เท่านั้น (ซึ่งสองสิ่งนี้ เป็นสิ่งที่จำเป็นต้องใช้อยู่แล้วถึงแม้จะเป็นการออกโปรแกรมโดยผู้เชี่ยวชาญก็ตาม) ระบบข้างต้นจะทำการสังเคราะห์โปรแกรมหลากหลายรูปแบบและทำการทดสอบ เพื่อหาโปรแกรมที่เหมาะสมในการแก้ปัญหานั้นๆ

ภาพที่ 1.1 แสดงภาพรวมการทำงานของระบบอัตโนมัติเพื่อการสังเคราะห์โปรแกรมรู้จำวัตถุซึ่งใช้เทคนิคการคำนวณเชิงวิวัฒนาการ (Evolutionary computation) [4] ข้อมูลขาเข้าจากผู้ใช้ได้แก่ ฐานข้อมูลภาพฝึกฝน ฟังก์ชันจุดประสงค์ และชุดคำสั่งประมวลผลภาพพื้นฐาน ระบบจะทำการสร้างประชากรของโปรแกรมจากชุดคำสั่งประมวลผลภาพพื้นฐาน และทำการวิวัฒนาการปรับปรุงโปรแกรมโดยใช้เทคนิคการคำนวณเชิงวิวัฒนาการ เมื่อระบบค้นพบโปรแกรมที่มีประสิทธิภาพดีเพียงพอ หรือจำนวนลูปมากกว่าค่าที่กำหนดไว้ ระบบสังเคราะห์โปรแกรมอัตโนมัติจะหยุดกระบวนการวิวัฒนาการและให้ผลลัพธ์คือโปรแกรมรู้จำวัตถุ ที่มีประสิทธิภาพสูงที่สุดเท่าที่เคยสังเคราะห์มาในกระบวนการวิวัฒนาการ

อย่างไรก็ตามระบบดังกล่าวมีจุดอ่อนที่สำคัญคือ เวลาการประมวลผลที่ยาวนาน เนื่องจากต้องรันโปรแกรมประมวลผลภาพเป็นจำนวนมาก ทั้งนี้ผู้วิจัยเคยนำเสนอขั้นตอนวิธีหลายวิธีเพื่อลดเวลาการประมวลผลของระบบสังเคราะห์โปรแกรมประมวลผลภาพ [23, 24, 27] ซึ่งสามารถลดเวลาการ

ประมวลผลได้ระดับหนึ่ง อย่างไรก็ตามหากปัญหามีความซับซ้อนมาก ระบบอาจจะจำเป็นต้องใช้จำนวนรูปภาพฝึกฝนเป็นจำนวนมาก ซึ่งทำให้จำเป็นต้องใช้เวลามากในการแก้ไขปัญหา



ภาพที่1.1 ผังงานของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติโดยใช้การคำนวณเชิงวิวัฒนาการ

ในปัจจุบัน การ์ดแสดงผลภาพ หรือ ที่เรียกกันสั้นๆว่า GPU (Graphics processing unit) ได้มีการพัฒนาอย่างรวดเร็วจนสามารถประมวลผลได้รวดเร็วกว่า หน่วยประมวลผล (CPU) ในคอมพิวเตอร์ทั่วไป [11] เมื่อไม่นานมานี้ GPU ได้ถูกพัฒนาให้สามารถโปรแกรมได้โดยผู้ใช้เพื่อให้สามารถประมวลผลงานด้านอื่นๆนอกจาก Computer graphics ได้ GPU ชนิดนี้เรียกว่า GPGPU (General purpose GPU) [12] หรือ GPUวัตถุประสงค์ทั่วไป และเมื่อปี 2006 มีการพัฒนาภาษาคอมพิวเตอร์เพื่อใช้ในการโปรแกรม GPGPU ได้แก่ ภาษา CUDA C ซึ่งใช้งานร่วมกับภาษา C ภาษา CUDA C [18] ซึ่งถือได้ว่าเป็นภาษาที่ใหม่มากและยังขาดผู้เชี่ยวชาญการเขียนโปรแกรมภาษานี้ หากมีการศึกษาการใช้งาน GPU อย่างจริงจัง น่าจะเป็นประโยชน์อย่างยิ่งต่อวงการวิจัยของสถาบันฯและประเทศ โครงการวิจัยนี้มุ่งเน้นจะศึกษาการใช้งาน GPGPU เพื่องานด้านประมวลผลภาพ เพื่อนำมาเร่งความเร็วระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ โดยส่วนการประมวลผลภาพจะทำงานบน GPU และการทำงานส่วนที่เหลือเช่นกระบวนการวิวัฒนาการ จะกระทำบน CPU ของคอมพิวเตอร์ตามเดิม

1.2 วัตถุประสงค์และขอบเขตของโครงการ

โครงการวิจัยชิ้นนี้มีวัตถุประสงค์ดังต่อไปนี้

- เพื่อศึกษาการเขียนโปรแกรมประมวลผลภาพเชิงขนานบน GPU โดยใช้ภาษา CUDA C
- เพื่อพัฒนาระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติที่สามารถรันชุดคำสั่งประมวลผลภาพบน GPU ได้
- เพื่อศึกษาและเปรียบเทียบเวลาประมวลผลของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติที่ใช้หรือไม่ใช้ GPU

โครงการวิจัยชิ้นมุ่งเน้นที่จะศึกษาการเขียนโปรแกรมชุดคำสั่งประมวลผลภาพพื้นฐานบน GPU โดยใช้ภาษา CUDA C เพื่อนำไปใช้เร่งความเร็วระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ ไม่ได้มุ่งเน้นที่จะเร่งความเร็วโปรแกรมประมวลผลภาพที่ซับซ้อนหรือโปรแกรมประมวลผลภาพเฉพาะทาง

1.3 วิธีดำเนินการวิจัย

การดำเนินการวิจัยของโครงการนี้ประกอบไปด้วย การศึกษาโครงสร้าง GPU ภาษา CUDA C และการเขียนโปรแกรมเพื่อสั่งงาน GPU เป็นหลัก จากนั้นจะทำการพัฒนาชุดคำสั่งประมวลผลภาพพื้นฐานให้สามารถทำงานบน GPU ได้อย่างมีประสิทธิภาพ และทำการเชื่อมต่อกับระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ สุดท้ายจะทำการวัดประสิทธิภาพของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติที่เชื่อมต่อกับ GPU เปรียบเทียบกับระบบที่ไม่เชื่อมต่อกับGPU และทำการเปรียบเทียบเชิงสถิติ โดยการดำเนินโครงการวิจัยสามารถแบ่งได้เป็น 3 ช่วงหลักๆ ได้แก่

- ช่วงการศึกษาการใช้งาน GPU – ศึกษาหาตำรา บทความที่เกี่ยวข้องกับโครงสร้างการทำงานของ GPU การคำนวณแบบขนานบน GPU และการเขียนโปรแกรมสั่งงาน GPU
- ช่วงการพัฒนาชุดคำสั่ง บน GPU – เขียนโปรแกรมชุดคำสั่งประมวลผลภาพพื้นฐานบน GPU
- ช่วงการรวมเข้ากับระบบสังเคราะห์โปรแกรม – เขียนโปรแกรมให้ระบบสังเคราะห์โปรแกรมรู้จำวัตถุสามารถติดต่อกับ GPU ได้ จากนั้นจะทำการศึกษาและเปรียบเทียบเวลาประมวลผลของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติที่ใช้หรือไม่ใช้ GPU

บทที่ 2

การดำเนินการวิจัย

บทนี้จะพูดถึงผลการศึกษาประวัติความเป็นรวมทั้งโครงสร้างของ GPU วิธีเขียนโปรแกรมสั่งงาน GPU การพัฒนาชุดคำสั่งประมวลผลภาพเชิงขนานบน GPU รวมทั้งการรวมชุดคำสั่งประมวลผลภาพที่ได้พัฒนาแล้วเข้ากับระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ

2.1 การศึกษาโครงสร้างของ GPU

2.1.1 ความเป็นมาของ GPU

ประวัติความเป็นมาของ Graphics processing unit หรือ GPU ถือกำเนิดในช่วงปลายยุค 80 โดยเริ่มจาก Graphics accelerator ซึ่งเกิดมาหลังจากระบบปฏิบัติการแบบกราฟิกอย่าง Windows หลังจากนั้นในช่วงกลางถึงปลายยุค 90 เกมส์คอมพิวเตอร์แบบ3มิติ ก็ได้เริ่มเฟื่องฟู พร้อมๆกับ GPU ที่เริ่มพัฒนาเพื่อให้สามารถแสดงข้อมูล3มิติบนจอมอนิเตอร์ได้อย่างรวดเร็ว GPUในยุคแรกๆนี้ถูกพัฒนามาสำหรับงานด้านกราฟิกโดยเฉพาะและไม่อนุญาตให้ผู้ใช้งานเขียนโปรแกรมสั่งงาน GPUได้ GPUได้ถูกพัฒนามาอย่างต่อเนื่องและอาศัยการทำงานเชิงขนานจำนวนมหาศาลพร้อมๆกันจึงสามารถทำงานได้อย่างรวดเร็ว จนกระทั่งในช่วงหลังต้นๆยุคปี 2000 พลังการคำนวณของ GPU เริ่มที่จะสูงกว่า CPU ซึ่งติดปัญหาความร้อนภายในตัว CPU เอง ด้วยเหตุนี้มีผู้มีแนวคิดที่จะนำ GPU มาใช้ในการคำนวณด้านอื่นๆแทน CPU ซึ่งนำไปสู่การพัฒนา GPU ที่อนุญาตให้ผู้ใช้งานเขียนโปรแกรมสั่งงาน GPU ได้

ในช่วงนี้ มีการเริ่มนำ GPU มาใช้ในงานด้านการประมวลผลภาพ (Image processing) และทัศนศาสตร์คอมพิวเตอร์ (Computer vision) ยกตัวอย่างเช่น ในปี 2004 Fung และ Mann [7] ได้ทดลองโปรแกรมตรวจจับการเคลื่อนไหวของกล้องแบบเวลาจริง (Real-time camera motion tracking) บน GPU รุ่น GeForce FX โดยใช้ OpenGLและภาษา Cg โดยสามารถเร่งความเร็วได้ 3.5 เท่าเมื่อเทียบกับ CPU ในปีเดียวกัน Fung และ Mann [8] ยังได้ทดลองใช้ การ์ด GPU หลายๆการ์ดในการทดสอบเพื่อเร่งความเร็วการประมวลผลทัศนศาสตร์คอมพิวเตอร์ โดยทดสอบการรู้จำรูปภาพใน Eigen space โดยใช้ การ์ด GPU 5 ชุดต่อในเครื่องคอมพิวเตอร์หนึ่งเครื่องและพบว่าสามารถเร่งความเร็วได้ 4.5 เท่า ในปี 2006 Durkovic และคณะ [5] ได้ทดลองโปรแกรมคำนวณ Optical flow ของ Lucas และ Kanade [13] บน GPU รุ่น GeForce 6800 Ultra และพบว่าสามารถทำงานได้โดยใช้เวลาน้อยกว่า 1/12 เท่าของเวลาที่ใช้บน CPU ในปี 2006 Marial และคณะ [14] ได้นำเสนอขั้นตอนวิธีการคำนวณ เปรียบเทียบสเตริโอแบบหนาแน่น (Dense stereo matching) แบบเร็วบน GPU ซึ่งใช้สร้างกลับข้อมูลภาพ 3มิติได้ พวกเขาทดสอบกับ GPU รุ่น GeForce 7800 GTX โดยใช้ OpenGL และ ภาษา Cg และพบว่าสามารถเร่งความเร็วการทำงานได้ประมาณ 10 ถึง 15 เท่าเมื่อเทียบกับ CPU

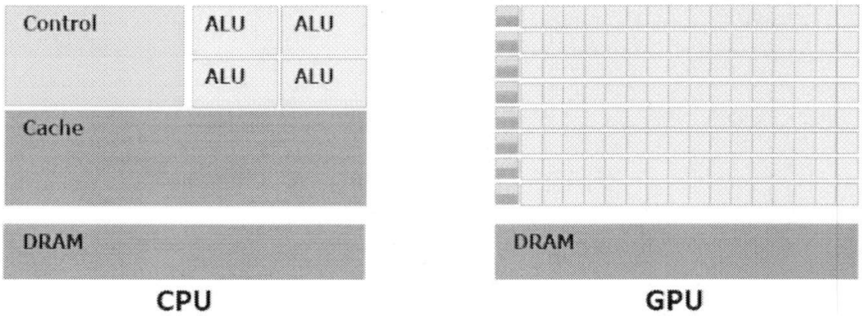
อย่างไรก็ตาม การเขียนโปรแกรมบน GPU ในยุคแรกนี้ มีความยากลำบากเพราะว่า ผู้เขียนโปรแกรมจะต้องสั่งการ GPU ผ่านทาง OpenGL และภาษาสำหรับคอมพิวเตอร์กราฟิกโดยเฉพาะอย่าง Cg ซึ่งหมายความว่า หากนักวิจัยหรือนักพัฒนาต้องการที่จะเร่งความเร็วการคำนวณงานของตน (ซึ่งไม่จำเป็นจะต้องเป็นงานด้านกราฟิก) จะต้องหันมาเรียนรู้ ภาษาคอมพิวเตอร์กราฟิกก่อนจึงจะทำงานได้

ในปี 2006 บริษัท NVIDIA ได้พัฒนา GPU รุ่นแรกซึ่งสนับสนุน Direct X และใช้สถาปัตยกรรมที่ชื่อว่า CUDA [18] รวมทั้งได้พัฒนาภาษาคอมพิวเตอร์เพื่อสั่งการ GPU ที่สนับสนุนสถาปัตยกรรม CUDA (หรือเรียกสั้นๆว่า CUDA GPU) โดยภาษานี้ใช้โครงสร้างดั้งเดิมของภาษา C ซึ่งมักใช้กันเป็นที่แพร่หลายในการคำนวณด้านวิทยาศาสตร์ ภาษาใหม่นี้มีชื่อว่า CUDA C ซึ่งทำให้ผู้ใช้งานไม่จำเป็นจะต้องศึกษาภาษาคอมพิวเตอร์กราฟิก และแก้ไขปัญหาของตัวเองโดยใช้มุมมองของคอมพิวเตอร์กราฟิกอีกต่อไป ภาษา CUDA C คาดว่าจะกลายเป็นมาตรฐานต่อไปสำหรับการเขียนโปรแกรมบน GPU

นอกจากนี้ยังมี GPU รุ่นใหม่ๆซึ่งมีประสิทธิภาพสูงขึ้นเรื่อยๆออกวางตลาด เมื่อไม่นานมานี้ได้มีผู้ประยุกต์ใช้ CUDA GPU เพื่อแก้ปัญหาการคำนวณที่ต้องใช้พลังการคำนวณมาก ยกตัวอย่างเช่น การสร้างภาพ 3มิติเพื่อตรวจหามะเร็งเต้านม การสร้างภาพ3มิตินี้จะต้องใช้ข้อมูลภาพ Ultrasound จำนวนมากซึ่งมีขนาดข้อมูลรวมประมาณ 35 GB (ได้มาจากการสแกนภาพต่อเนื่อง 15 นาที) ด้วยการประมวลผลบน GPU รุ่น Tesla C1060 ซึ่งถือได้ว่าเป็น High-performance computing GPU ตัวหนึ่งในปัจจุบัน แพทย์สามารถที่จะวินิจฉัยมะเร็งเต้านมจากข้อมูล 3มิตินี้ได้ในเวลาไม่เกิน 20 นาที [18] นอกจากนี้ GPU ก็ยังถูกนำไปใช้กับหัวข้อวิจัยด้านอื่นๆเช่น ระบบโครงข่ายประสาทเทียม (Artificial neural networks: ANN) [15] หรือ ในแวดวงวิจัยการโปรแกรมเชิงพันธุกรรม (Genetic programming: GP) ก็ได้มีแนวคิดที่จะเร่งความเร็ว GP โดยใช้ GPU [17] และมักจะมีภาคการประชุมพิเศษสำหรับการประยุกต์ใช้ GPU ในการประชุมวิชาการนานาชาติชั้นนำต่างๆ

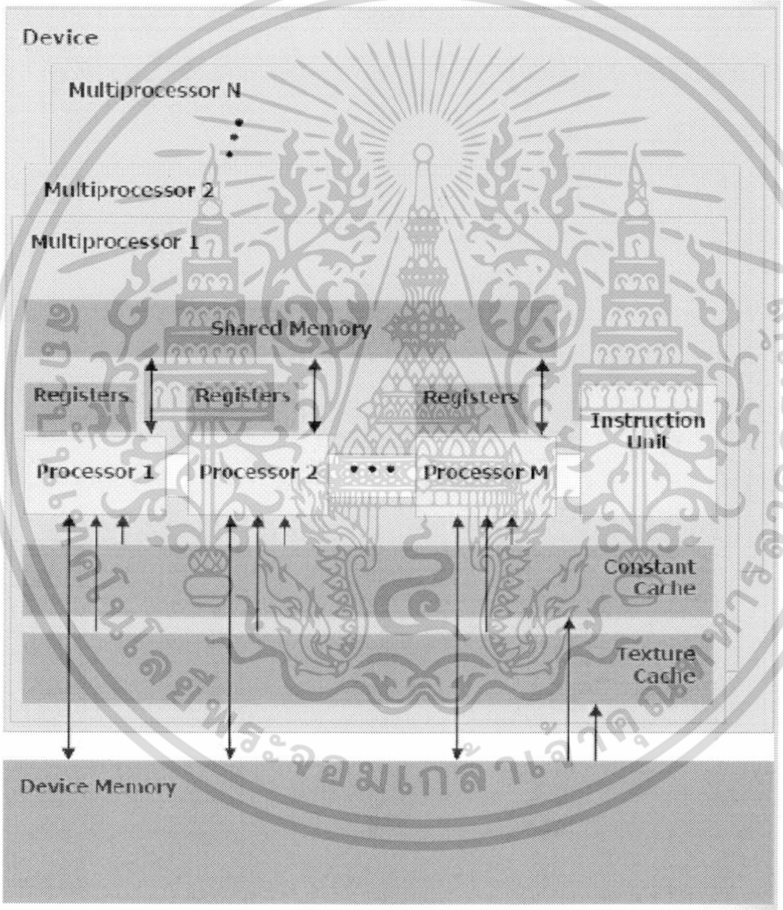
2.1.2 โครงสร้างของGPU

GPU ถูกออกแบบมาเพื่อให้สามารถประมวลผลกราฟิกได้อย่างรวดเร็ว โครงสร้างของ GPU จึงแตกต่างจาก CPU ซึ่งถูกออกแบบมาให้สามารถประมวลงานทั่วไป ภาพที่ 2.1 แสดงให้เห็นถึงความแตกต่างของโครงสร้าง CPU และ GPU จากภาพจะเห็นได้ว่า CPU ในปัจจุบันประกอบไปด้วยหน่วยประมวลผลหรือ core อยู่เพียงไม่กี่หน่วย (ในรูปคือบริเวณสีเขียว) แต่ละ core สามารถทำงานที่ค่อนข้างซับซ้อนได้ ซึ่งเหมาะสมกับการใช้งานทั่วๆไป ในขณะที่ GPU จะประกอบไปด้วย core ขนาดเล็กจำนวนมาก แต่ละ core ถูกออกแบบให้ประมวลผลงานเล็กๆเฉพาะด้าน จึงเหมาะกับงานที่ประมวลผลข้อมูลขนาดใหญ่ (เช่น คอมพิวเตอร์กราฟิก หรือ การประมวลผลภาพ)



ภาพที่ 2.1 โครงสร้างของ CPU และ GPU

ที่มา: A. Lippert, 2009 [12]



ภาพที่ 2.2 โครงสร้างลำดับชั้นของหน่วยความจำ GPU

ที่มา: A. Lippert, 2009 [12]

ภาพที่ 2.2 แสดงรายละเอียดของโครงสร้าง GPU ซึ่งประกอบไปด้วยหน่วยย่อยที่เรียกว่า Streaming multiprocessor หรือ SM หลายๆหน่วย แต่ละหน่วยประกอบไปด้วย core จำนวนมากซึ่งเรียกว่า Stream processor หรือ SP นอกจากนี้ภาพที่ 2.2 ยังแสดงโครงสร้างลำดับชั้นของหน่วยความจำใน GPU อีกด้วย โดยประกอบไปด้วย 1) Device memory ซึ่งเป็นหน่วยความจำแบบ global ใน GPU

สามารถเข้าถึงได้จากทุกๆ SP แต่จะมีเวลาเข้าถึง (Latency time) สูงที่สุด 2) Texture memory ซึ่งเป็นหน่วยความจำแบบ read-only สำหรับเก็บข้อมูลพื้นผิวในงานกราฟิก 3) Constant memory ซึ่งเป็นหน่วยความจำแบบ read-only สำหรับเก็บค่าคงที่ซึ่งไม่สามารถเปลี่ยนแปลงได้ในขณะที่ kernel (ชื่อเรียกฟังก์ชันย่อยซึ่งทำงานบน GPU) กำลังทำงานอยู่ 4) Shared memory ซึ่งเป็นหน่วยความจำที่ใช้ร่วมกันระหว่าง SP ที่อยู่ใน SM เดียวกัน และ 5) Register เป็นหน่วยความจำที่เข้าถึงได้เร็วที่สุดแต่ละ SP จะมี register ของตนเอง

2.2 การศึกษาเขียนโปรแกรมสั่งงาน GPU โดยใช้ CUDA C

2.2.1 การเขียนโปรแกรมสั่งงาน GPU เบื้องต้นโดยใช้ CUDA C

ในปัจจุบัน มีการใช้ประโยชน์จาก GPU เพื่อช่วยเร่งความเร็วในการคำนวณงานต่างๆ โดยโปรแกรมจะถูกแบ่งออกเป็นสองส่วนใหญ่คือ 1) โปรแกรมส่วนที่ถูกประมวลผลโดย CPU และ 2) โปรแกรมส่วนที่ถูกประมวลผลโดย GPU ซึ่งมักจะถูกเขียนเพื่อให้ประมวลผลเชิงขนานกับข้อมูลที่มีขนาดใหญ่ ซึ่งอาศัยหลักการที่เรียกว่า SPMD (Single Program, Multiple Data) ซึ่งจะใช้หน่วยประมวลผล Streaming processor จำนวนมากบน GPU ทำงานชิ้นเดียวกัน (Single program) กับข้อมูลที่ต่างกัน (Multiple data) พร้อมๆกันซึ่งจะช่วยเพิ่มประสิทธิภาพในการคำนวณอย่างมาก ฟังก์ชันที่ถูกเขียนขึ้นมาเพื่อให้สามารถรันบน GPU ได้นั้นจะถูกเรียกว่า Kernel

ภาษา CUDA C [18] ได้ถูกพัฒนาโดยโดยบริษัท NVIDIA เพื่อให้ผู้ใช้สามารถเขียนโปรแกรมสั่งงาน GPU ของ NVDIA ได้โดยใช้ภาษา C และชุดคำสั่งขยายที่เพิ่มขึ้น ผู้ใช้สามารถใช้ภาษา C ปกติในการสั่งงานโปรแกรมส่วนที่ถูกประมวลผลโดย CPU และใช้ชุดคำสั่งขยายใน CUDA เพื่อที่จะควบคุมการรับส่งข้อมูลระหว่าง CPU กับ GPU รวมทั้งสามารถสั่ง GPU ให้ทำการประมวลผลบางอย่างแล้วส่งผลลัพธ์กลับมายัง CPU ได้ ตัวอย่างต่อไปนี้เป็นโค้ด CUDA C เพื่อทำการบวกเวกเตอร์แบบขนานบน GPU (ดัดแปลงมาจากโค้ดตัวอย่างใน [18])

```
#include <iostream>

using namespace std;

__global__ void add_vector( int *a, int *b, int *c, int N ){
    int tid = blockIdx.x;
    if( tid < N )
            c[tid] = a[tid] + b[tid];
}

int main(){
    const int N = 1000;
    int a[N], b[N], c[N];
    for( int i=0; i<N; ++i ){
            a[i] = i;
            b[i] = i + 3;
    }
    int *dev_a, *dev_b, *dev_c;

    cudaMalloc( (void**)&dev_a, N*sizeof(int) );
    cudaMalloc( (void**)&dev_b, N*sizeof(int) );
    cudaMalloc( (void**)&dev_c, N*sizeof(int) );
```

```
23:
24:     cudaMemcpy( dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice );
25:     cudaMemcpy( dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice );
26:
27:     add_vector<<<N,1>>>( dev_a, dev_b, dev_c, N );
28:
29:     cudaMemcpy( &c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost );
30:
31:     for( int i=0; i<N; ++i )
32:             cout << c[i] << " ";
33:
34:     cudaFree( dev_a );
35:     cudaFree( dev_b );
36:     cudaFree( dev_c );
37:
38:     return 0;
39:}
```

โค้ดตัวอย่างสามารถแบ่งได้เป็นสองส่วนหลักๆดังนี้ โค้ดบรรทัดที่ 5-9 เป็นส่วนของ Kernel ที่จะถูกรันบน GPU (__global__ เป็นคำสำคัญที่บ่งบอกว่าเป็นฟังก์ชันที่จะถูกรันบน GPU) และ เมนฟังก์ชันซึ่งจะถูกเริ่มต้นรันบน CPU (บรรทัดที่ 11-39) โดยในเมนฟังก์ชันนี้ จะมีการประกาศอาเรย์ 1 มิติ จำนวน 3 ตัวโดยจะถูกใช้เป็นเวกเตอร์ขาเข้า สองตัว และ เวกเตอร์ผลลัพธ์ 1 เวกเตอร์(บรรทัดที่ 13-17) จากนั้นจะทำการประกาศตัวชี้ (Pointer) และจองหน่วยความจำบน GPU โดยใช้คำสั่งจาก CUDA API ที่ชื่อว่า cudaMalloc (บรรทัดที่ 20-22) และจะทำการก็อปปี้ค่าจากหน่วยความจำบน CPU ลงไปยังหน่วยความจำบน GPU โดยใช้คำสั่ง cudaMemCpy (บรรทัดที่ 24-25) และทำการรัน Kernel (บรรทัดที่ 27) ชื่อว่า add_vector บน GPU โดยเรียกคำสั่ง

add_vector<<<N,1>>>(dev_a, dev_b, dev_c, N);

โดย N และ 1 หมายความว่า จะแบ่งกลุ่มของหน่วยประมวลผลออกเป็น N บล็อก แต่ละบล็อกประกอบไปด้วย 1 เธรด (Thread) ในการประมวลผลคำสั่ง add_vector นี้

ภายใน Kernel (บรรทัดที่ 5-9) จะทำการคำนวณเลขดัชนีในการอ้างอิงถึงแต่ละมูลฐาน (Element) ในเวกเตอร์ โดยในที่นี้จะใช้หมายเลขบล็อกแทนเลขดัชนี (เนื่องจากในกรณี 1 บล็อกมีเพียง 1 เธรด ดังนั้นหมายเลขบล็อกจึงใช้แทนหมายเลขดัชนีได้) จากนั้นจะทำการบวกมูลฐานที่ระบุโดยเลขดัชนี และทำการเก็บค่า kernel นี้จะถูกรันพร้อมๆกันบนหลายๆ SP เพื่อทำการบวกค่าแต่ละมูลฐานพร้อมๆกัน

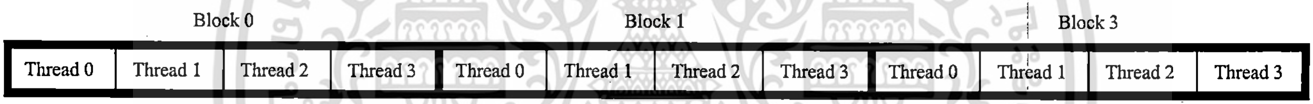
จากนั้นจะทำการอ่านค่าจากหน่วยความจำบน GPU ซึ่งเก็บผลลัพธ์ของการบวกเวกเตอร์ มายังหน่วยความจำบน CPU (บรรทัดที่29) และทำการแสดงผลลัพธ์บนหน้าจอ (บรรทัดที่31-32) ในบรรทัดที่ 34-36 เป็นการคืนหน่วยความจำที่ถูกจองไว้บน GPU

2.2.2 การเขียนโปรแกรมชุดคำสั่งประมวลผลภาพบน GPU โดยใช้ CUDA C

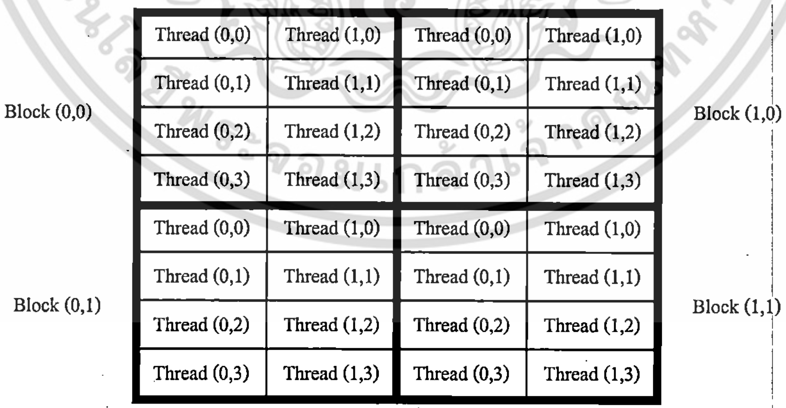
หนึ่งในหลักสำคัญในการเขียนโปรแกรมสั่งงาน GPU ด้วย CUDA C คือ การทำความเข้าใจเกี่ยวกับเธรด (Threads) และ เธรดบล็อก (Thread blocks) ในการประมวลผลด้วย GPU นั้น เธรดเป็น

หน่วยย่อยที่สุดที่ถูกสร้างขึ้นมาเพื่อทำงานที่ถูกโปรแกรมไว้กับข้อมูลหนึ่งชุด (เช่น หนึ่งจุดภาพ) เพื่อที่จะทำงานหลักชิ้นหนึ่ง เช่นแปลงภาพขาเข้าให้เป็นภาพเนกาทีฟ (Negative) กลุ่มของเธรดจำนวนหนึ่งจะถูกสร้างขึ้นมาเพื่อที่จะประมวลผลกับข้อมูลหลายๆชุดพร้อมๆกัน ในการเขียนโปรแกรมด้วย CUDA C เราสามารถที่จะกำหนดกลุ่มของเธรดซึ่งเรียกว่า เธรดบล็อก ขึ้นมาได้ โดยเธรดในเธรดบล็อกเดียวกันจะถูกประมวลผลใน Multiprocessor ตัวเดียวกันซึ่งทำให้สามารถทำงานร่วมกันได้โดยผ่านทาง Shared memory และสามารถรอเข้าจังหวะ (Synchronization) กับเธรดอื่นๆได้ (ในกรณีที่ ต้องรอให้ผลการประมวลจากเธรดอื่นๆเสร็จสิ้นก่อน)

ในการเขียนโปรแกรมด้วย CUDA C เราสามารถกำหนดโครงสร้างของเธรดและเธรดบล็อกได้หลายแบบเช่น เธรดบล็อกแบบ 1 มิติ ดังตัวอย่างในภาพที่ 2.3 และ เธรดบล็อกแบบ 2 มิติ ดังตัวอย่างในภาพที่ 2.4 ซึ่งเธรดบล็อกทั้งสองรูปแบบนี้จะถูกเลือกใช้ตามความเหมาะสม เช่น การแปลงภาพขาเข้าให้เป็นภาพเนกาทีฟนั้น ค่าสีของจุดภาพทุกๆจุดจะถูกประมวลอย่างเดียวกันโดยไม่ขึ้นกับค่าพิกัดของจุดภาพ ในกรณีนี้การใช้เธรดบล็อกแบบ 1 มิติน่าจะเป็นวิธีที่ง่ายที่สุด ในทางกลับกันการประมวลผลภาพบางอย่าง เช่น การกรอง ซึ่งมีการเข้าถึงจุดภาพซึ่งอยู่ใกล้เคียงกันนั้น การใช้เธรดบล็อกแบบ 2 มิติก็จะทำให้โปรแกรมที่ถูกเขียนนั้นเข้าใจง่ายและเป็นธรรมชาติมากกว่าการใช้เธรดบล็อกแบบ 1 มิติ



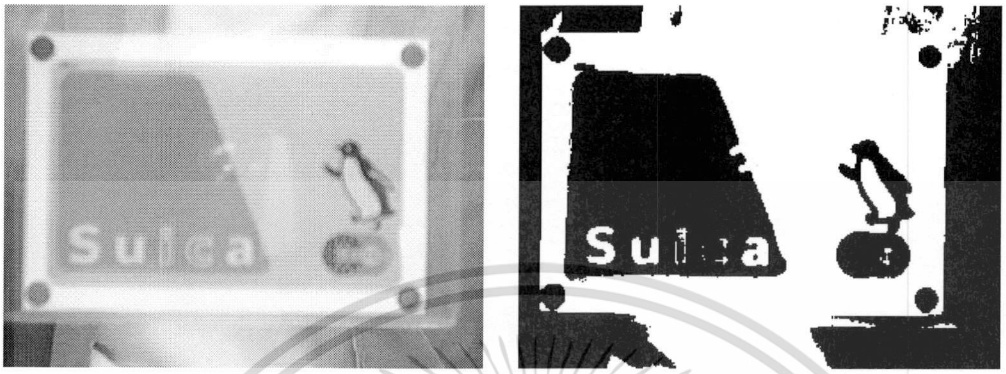
ภาพที่ 2.3 ตัวอย่างเธรดบล็อกแบบ 1 มิติซึ่งประกอบไปด้วย 3 เธรดบล็อก แต่ละเธรดบล็อกมี 4 เธรด



ภาพที่ 2.4 เธรดบล็อกแบบ 2 มิติซึ่งประกอบไปด้วย 4 เธรดบล็อก (2× 2) แต่ละเธรดบล็อกมี 8 เธรด (2× 4)

ผู้วิจัยจะยกตัวอย่างโค้ดโปรแกรมของคำสั่ง การแปลงให้เป็นภาพสองระดับ (Binarization) เพื่ออธิบายการสั่งงานโปรแกรมบน GPU ภาพสองระดับคือ ภาพที่มีค่าสีเพียงสองค่า ซึ่งโดยทั่วไปคือ 0 (สี

ดำ) และ 1 (สีขาว) หรือ ในบางกรณีอาจจะใช้เป็น 0 (สีดำ) และ 255 (สีขาว) ก็ได้ การทำให้เป็นภาพสองระดับมักจะเป็นหนึ่งในกระบวนการที่ถูกใช้แปลงภาพ ก่อนที่จะเริ่มการประมวลภาพบางอย่าง ตัวอย่างผลลัพธ์ของการแปลงให้เป็นภาพสองระดับถูกแสดงไว้ในภาพที่ 2.5



ภาพที่ 2.5 ตัวอย่างภาพที่ได้จากการแปลงให้เป็นภาพสองระดับ โดยใช้ค่าเฉลี่ยเป็นค่าขีดเริ่มเปลี่ยน

ในโค้ดตัวอย่างต่อไปนี้ เป็นการแปลงให้ภาพสองระดับ โดยอาศัยค่าขีดเริ่มเปลี่ยน (Threshold value) ค่าหนึ่ง โดยจุดภาพที่มีค่าระดับสีเทาสูงกว่าหรือเท่ากับค่าขีดเริ่มเปลี่ยนนี้ จะถูกกำหนดค่าสีใหม่ให้เป็นสีขาว (255) ในทางกลับกัน จุดภาพที่มีค่าระดับสีเทาต่ำกว่าค่าขีดเริ่มเปลี่ยน จะถูกกำหนดค่าสีใหม่ให้เป็นสีดำ (0)

```
01:template <typename T>
02:__global__ void kernelThresholding( T *in, T *out, int N, float th ){
03:     //1D thread blocks
04:     int offset = threadIdx.x + blockIdx.x*blockDim.x;
05:
06:     if( offset < N )
07:             if( (float)in[offset] >= th )
08:                     out[offset] = 255;
09:             else
10:                     out[offset] = 0;
11:}
12:template <typename T>
13:void gpuThresholding( T* dev_in, T* dev_out, int width, int height, float th ){
14:     const int N = width*height;
15:     kernelThresholding<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
16:                             ( dev_in, dev_out, N, th );
17:}
```

จากโค้ดข้างต้นประกอบไปด้วยสองฟังก์ชัน 1) ฟังก์ชัน kernelThresholding ซึ่งจะถูกรันบน GPU (บรรทัดที่ 1 - 11) และ 2) ฟังก์ชัน gpuThresholding ซึ่งเป็น C/C++ อินเตอร์เฟสที่จะถูกเรียกใช้โดย CPU (บรรทัดที่ 12 - 17) ฟังก์ชัน gpuThresholding ต้องการพารามิเตอร์ทั้งหมด 5 ตัวได้แก่

1) T* dev_in – ตัวชี้ (pointer) ของข้อมูลภาพขาเข้า
2) T* dev_out – ตัวชี้ของข้อมูลภาพผลลัพธ์
3) int width – ความกว้างของรูปภาพ

4) int height – ความสูงของรูปภาพ

5) float th – ค่าขีดเริ่มเปลี่ยน

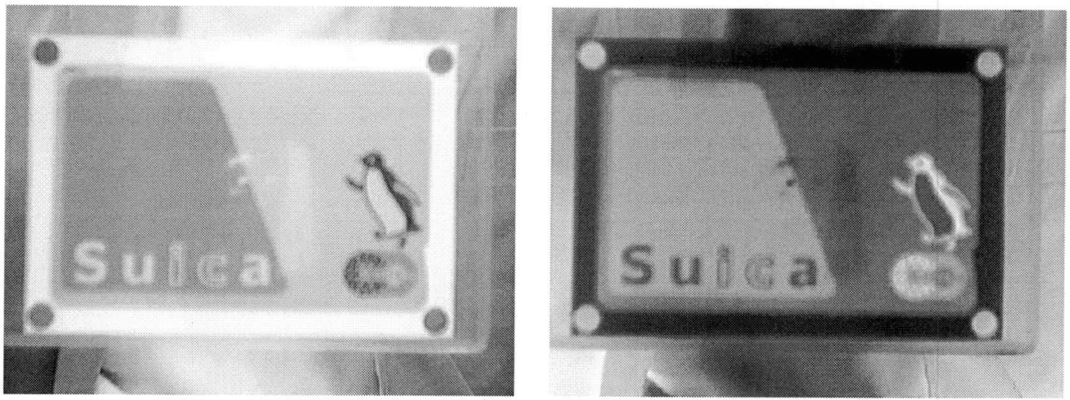
ในบรรทัดที่ 14 จะทำการคำนวณขนาดของข้อมูลภาพในรูปแบบอาเรย์ 1 มิติ โดยนำความกว้างมาคูณกับความสูง จากนั้น ในบรรทัดที่ 15 จะทำการเรียก Kernel ซึ่งจะรันบน GPU โดยส่งพารามิเตอร์เข้าไป 4 ตัว นอกจากนั้น ยังมีพารามิเตอร์ที่ควบคุมโครงสร้างของเธรดบล็อกซึ่งถูกประกาศอยู่ในเครื่องหมาย <<< >>>โดยในกรณีนี้จะสั่งให้ GPU รันโดยใช้ เธรดบล็อกแบบ 1 มิติ ซึ่งประกอบไปด้วยเธรดบล็อกจำนวน (N+threadsPerBlock-1)/threadsPerBlock บล็อก โดยที่แต่ละบล็อกประกอบไปด้วยเธรดจำนวน threadsPerBlock เธรด

ในส่วนการทำงานของ Kernel ฟังก์ชันนั้น จะเริ่มจากบรรทัดที่ 4 ซึ่งทำการคำนวณที่อยู่ (Index) ในอาเรย์ 1 มิติของจุดภาพที่พิกัด (x, y) โดย x = threadIdx.x และ y = threadIdx.y จากนั้นจะทำการตรวจสอบว่า ค่าที่อยู่นี้ อยู่นอกเหนือจากขนาดของอาเรย์ 1 มิติหรือไม่ (บรรทัดที่ 6) ถ้าไม่ ก็จะทำการเปรียบค่าสีของจุดภาพที่พิกัด (x, y) กับค่าขีดเริ่มเปลี่ยน (บรรทัดที่ 7) ถ้าค่าระดับสีเทาสูงกว่าหรือเท่ากับค่าขีดเริ่มเปลี่ยน ก็กำหนดค่า 255 ให้กับจุดภาพที่พิกัด (x, y) ของภาพผลลัพธ์ (บรรทัดที่ 8) มิฉะนั้นจะกำหนดค่า 0 ให้แทน (บรรทัดที่ 10) ในส่วนการทำงานของ Kernel นี้เองที่จะถูกสั่งงานพร้อมๆกันโดยกลุ่มของเธรดจำนวนมาก ซึ่งทำให้การทำงานรวดเร็วกว่าการทำงานแบบเรียงลำดับ (Sequential processing) ใน CPU

ผู้วิจัยได้พัฒนาชุดคำสั่งประมวลผลภาพพื้นฐาน (สามารถดูรายละเอียดของหลายๆคำสั่งได้จาก [9]) ซึ่งสามารถทำงานบน GPU ได้ ซึ่งชุดคำสั่งประมวลผลภาพพื้นฐานนี้จะถูกใช้เป็น Primitive operators ในระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ รายชื่อของชุดคำสั่งที่ได้ทำการพัฒนาไปแล้วถูกแสดงในตารางที่ 2.1 (โค้ดโปรแกรมของชุดคำสั่งดังกล่าวอยู่ในภาคผนวก ก.) นอกจากนี้ภาพที่ 2.6 แสดงตัวอย่างภาพผลลัพธ์ที่ได้ชุดคำสั่งประมวลผลภาพที่ผู้วิจัยได้พัฒนาขึ้น

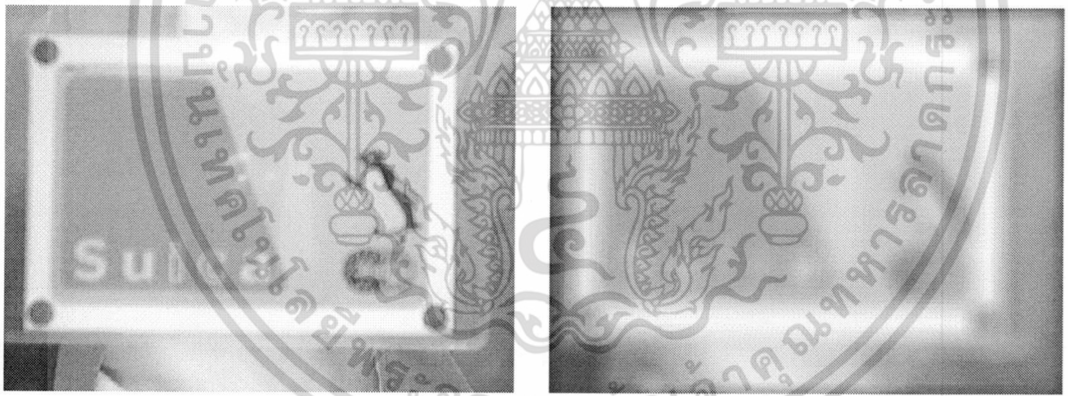
ตารางที่ 2.1 รายชื่อของชุดคำสั่งประมวลผลภาพพื้นฐานที่ได้พัฒนาแล้ว

ชุดคำสั่ง	**การทำงาน**
ชุดคำสั่งการเปลี่ยนค่าระดับสีเทา	
Image negative	แปลงภาพขาเข้าให้เป็นภาพแบบเนกาทีฟ
Histogram equalization	กระจายฮีสโตแกรมของภาพขาเข้าให้สม่ำเสมอ
ชุดคำสั่งการแปลงเป็นภาพสองระดับ	
Thresholding	แปลงภาพขาเข้าให้เป็นภาพสองระดับ โดยใช้ค่าขีดเริ่มเปลี่ยนที่กำหนด
Adaptive thresholding	แปลงภาพขาเข้าให้เป็นภาพสองระดับ โดยใช้ค่าเฉลี่ยเลขคณิตเป็นค่าขีดเริ่มเปลี่ยน
ชุดคำสั่งการหาขอบ	
Sobel edge detection	การหาขอบโดยใช้ตัวดำเนินการ Sobel
ชุดคำสั่งตัวกรอง	
Moving average filter	การกรองด้วยตัวกรองความถี่ต่ำผ่านแบบง่าย (หาค่าเฉลี่ยเลขคณิตของบริเวณท้องถิ่น)
Highpass filter	การกรองด้วยตัวกรองความถี่สูงผ่านแบบง่าย (ตัวกรองขนาด 3 × 3)
Gaussian filter	การกรองด้วยตัวกรองแบบ Gaussian
ชุดคำสั่งตัวดำเนินการ Morphology	
Dilation	ทำการขยายขอบของวัตถุเป้าหมายในภาพ
Erosion	ทำการกร่อนขอบของวัตถุเป้าหมายในภาพ
Opening	ทำการแยกวัตถุเป้าหมายในภาพที่อยู่ติดกันออกจากกัน
Closing	ทำการเชื่อมวัตถุเป้าหมายในภาพที่อยู่ด้วยกันเข้าด้วยกัน
ชุดคำสั่งเลขคณิตและฟังก์ชันของภาพ	
Image addition	นำค่าสีของแต่ละจุดภาพของภาพขาเข้าสองภาพ มาบวกกัน
Image subtraction	นำค่าสีของแต่ละจุดภาพของภาพขาเข้าสองภาพ มาลบกัน
Image multiplication	นำค่าสีของแต่ละจุดภาพของภาพขาเข้าสองภาพ มาคูณกัน
Image division	นำค่าสีของแต่ละจุดภาพของภาพขาเข้าสองภาพ มาหารกัน
Image logarithm	หาค่า logarithm ของค่าสีของแต่ละจุดภาพ
Image exponential	หาค่า exponential ของค่าสีของแต่ละจุดภาพ
Image square root	หารากที่สองของค่าสีของแต่ละจุดภาพ
Image absolute	หาค่าสัมบูรณ์ของค่าสีของแต่ละจุดภาพ
ชุดคำสั่งเชิงสถิติ	
Global mean	หาค่าเฉลี่ยเลขคณิตของระดับค่าสีของทุกจุดภาพ
Global variance	หาค่าความแปรปรวนของระดับค่าสีของทุกจุดภาพ
Global standard deviation	หาค่าเบี่ยงเบนมาตรฐานของระดับค่าสีของทุกจุดภาพ
Global skewness	หาค่า skewness ของระดับค่าสีของทุกจุดภาพ
Global kurtosis	หาค่า kurtosis ของระดับค่าสีของทุกจุดภาพ
Global maximum	หาค่าสูงสุดของระดับค่าสีของทุกจุดภาพ
Global minimum	หาค่าต่ำสุดของระดับค่าสีของทุกจุดภาพ
Global range	หาค่าเรนจ์ (ค่าสูงสุด – ค่าต่ำสุด) ของระดับค่าสีของทุกจุดภาพ

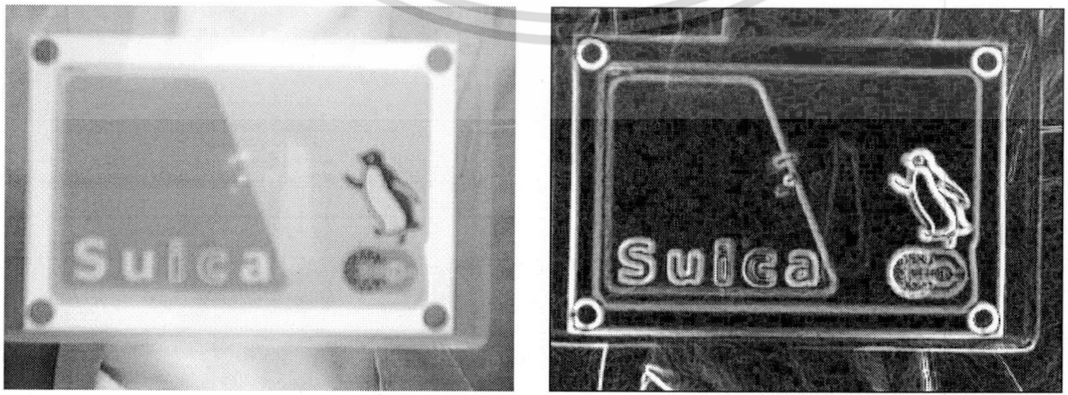


ก) ตัวอย่างภาพที่ได้จากการแปลงภาพ เนกาทีฟ

ข) ตัวอย่างภาพที่ได้จากการทำ Histogram equalization



ค) ตัวอย่างภาพที่ได้จากการแปลงกรองด้วยตัวกรองความถี่ต่ำผ่านอย่างง่าย



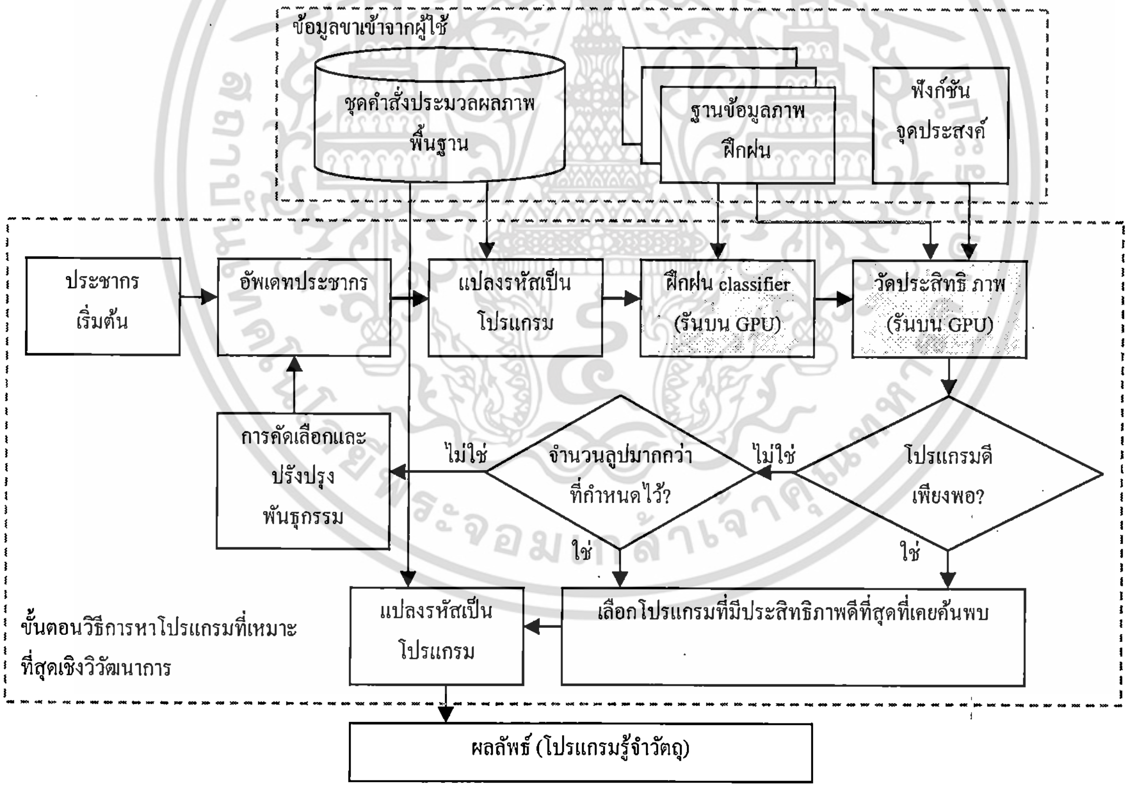
ง) ตัวอย่างภาพที่ได้จากการหาขอบด้วย ตัวดำเนินการ Sobel

ภาพที่ 2.6 ตัวอย่างภาพที่ได้จากคำสั่งประมวลผลภาพพื้นฐานแบบต่างๆซึ่งรันบน GPU

2.3 การพัฒนาระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติโดยใช้ GPU

ผู้วิจัยได้พัฒนาระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ ให้สามารถใช้งาน GPU ในการประมวลผลภาพ โดยภาพรวมของระบบดังกล่าวแสดงไว้ในภาพที่ 2.7 ซึ่งเหมือนกับภาพรวมของระบบในภาพที่ 1.1 ทุกประการ ยกเว้นสองกระบวนการที่ถูกแรเงาไว้เป็นสีเทาได้แก่ 1) การฝึกฝน Classifier และ 2) การวัดประสิทธิภาพโปรแกรม ในสองกระบวนการนี้จะเกี่ยวข้องกับการประมวลผลภาพ โดยในกระบวนการการฝึกฝน Classifier จะทำการประมวลผลภาพชุดฝึกฝนเพื่อนำให้เรียนรู้ Classifier จากนั้นเมื่อ Classifier ได้รับการฝึกฝนเรียบร้อย ระบบจะทำการวัดประสิทธิภาพของโปรแกรมรู้จำวัตถุ โดยทำการประมวลผลภาพชุดทดสอบ (ซึ่งไม่ได้อยู่ในชุดภาพฝึกฝน) และวัดประสิทธิภาพจากฟังก์ชันจุดประสงค์ที่กำหนดไว้ โดยในสองส่วนนี้จะ ระบบจะทำการรันโปรแกรมประมวลผลภาพบน GPU เพื่อลดเวลาการประมวลผล ซึ่งแตกต่างจากระบบเดิมซึ่งทำการประมวลผลภาพบน CPU

ในบทถัดไป ผู้วิจัยจะทำการศึกษาและเปรียบเทียบเวลาการประมวลผลของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติที่ใช้และไม่ใช้ GPU และทำการวิจารณ์ผลการทดลอง



ภาพที่2.7 ผังงานของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติซึ่งใช้ GPU

บทที่ 3

อภิปรายและวิจารณ์ผลการทดลอง

ในบทนี้ผู้วิจัยทำการทดลอง 2 การทดลอง ได้แก่ 1) การทดลองเปรียบเทียบเวลาการประมวลผลของชุดคำสั่งประมวลผลภาพที่รันบน CPU และGPU เพื่อศึกษาผลการเร่งความเร็วของแต่ละชุดคำสั่ง และ 2) การทดลองเปรียบเทียบเวลาการประมวลผลของระบบสังเคราะห์โปรแกรมที่ใช้และไม่ใช้ GPU เพื่อศึกษาผลการเร่งความเร็วของระบบในภาพรวม

3.1 การทดลองเปรียบเทียบเวลาประมวลผลของชุดคำสั่งประมวลผลภาพ

ผู้วิจัยได้ทำการเปรียบเทียบเวลาการประมวลผลของชุดคำสั่งประมวลผลภาพที่แสดงไว้ในตารางที่ 2.1 โดยใช้ภาพทดลอง (ภาพที่ 3.1) ที่มีขนาดต่างๆกันดังนี้ 160 × 120, 320 × 240, 640 × 480, 1280 × 960 และ 2560 × 1920 จุดภาพ ผู้วิจัยทำการวัดเวลาการประมวลผลชุดคำสั่งประมวลผลภาพที่รันบน CPU และ GPU เปรียบเทียบกัน (ใช้เวลาเฉลี่ย) โดยใช้อุปกรณ์และซอฟต์แวร์ดังต่อไปนี้

- คอมพิวเตอร์ PC ที่ใช้ Intel Core i7 CPU ความเร็ว 2.8 GHz. หน่วยความจำ RAM 3 GB.
- การ์ดแสดงผลภาพ GPU รุ่น GeForce GTX 460 ซึ่งมีจำนวน core ทั้งหมด 336 cores หน่วยความจำ GDDR5 ขนาด 1 GB ความเร็วของหน่วยประมวลผล 1,350 MHz.
- ไดรเวอร์ของการ์ดแสดงผลเวอร์ชั่น 8.17.12.6099
- CUDA Toolkit เวอร์ชั่น 3.2
- Visual C++ 2008 Express

ผลการทดลองแสดงไว้ในตารางที่ 3.1

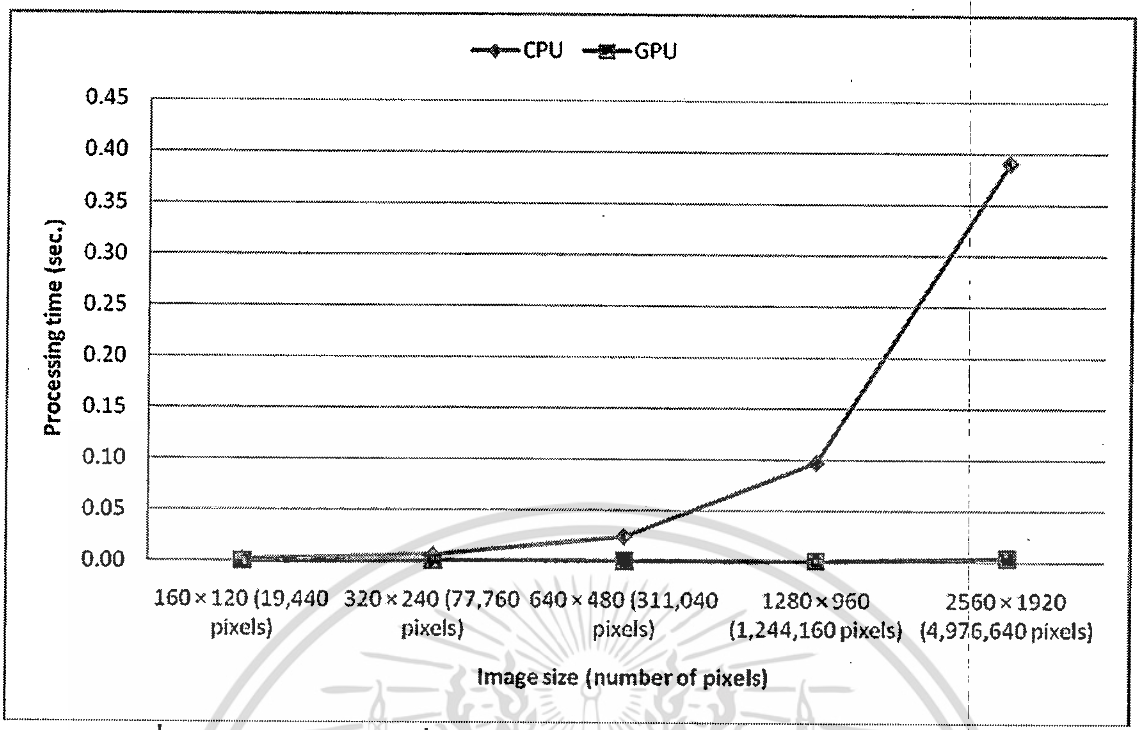
ภาพที่3.1 ตัวอย่างภาพขนาดต่างๆที่ใช้ในการเปรียบเทียบเวลาของชุดคำสั่งประมวลผลภาพ

ตารางที่ 3.1 เปรียบเทียบเวลาประมวลผลของชุดคำสั่งประมวลผลภาพ (วินาที)

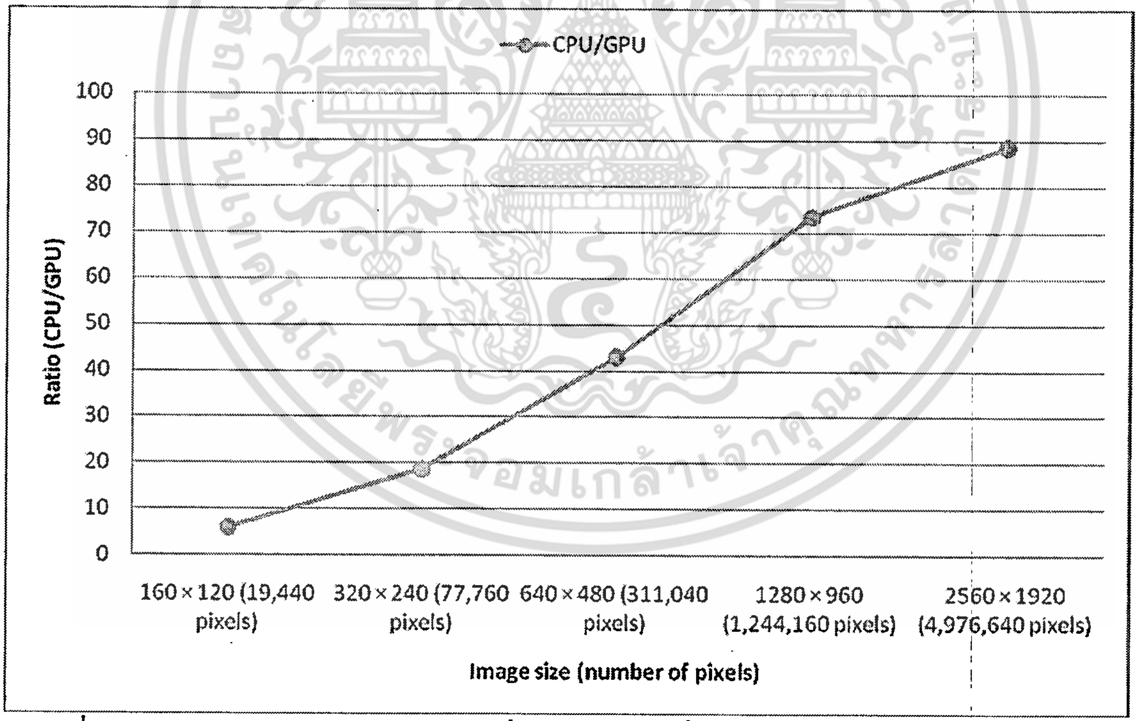
ชุดคำสั่ง	160 × 120			320 × 240			640 × 480			1280 × 960			2560 × 1920		
	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU
Image negative	4.65E-05	7.94E-06	5.86	1.60E-04	7.94E-06	20.14	8.00E-04	7.94E-06	100.72	3.12E-03	7.94E-06	392.80	1.23E-02	7.94E-06	1552.06
Histogram equalization	1.60E-04	5.88E-04	*0.27*	4.00E-04	8.27E-04	*0.48*	1.44E-03	1.55E-03	*0.93*	5.84E-03	4.54E-03	1.29	2.34E-02	1.62E-02	1.45
Thresholding	8.00E-05	7.94E-06	10.07	4.00E-04	7.94E-06	50.36	1.60E-03	7.94E-06	201.44	5.84E-03	7.94E-06	735.24	2.18E-02	7.94E-06	2739.52
Adaptive thresholding	1.60E-04	4.61E-04	*0.35*	5.60E-04	5.32E-04	1.05	2.16E-03	6.51E-04	3.32	8.32E-03	1.19E-03	6.98	3.18E-02	3.27E-03	9.73
Sobel edge detection	6.40E-04	7.94E-06	80.57	2.72E-03	7.94E-06	342.44	1.06E-02	7.94E-06	1339.54	4.25E-02	7.94E-06	5349.11	1.68E-01	7.94E-06	21161.78
Moving average filter (3 × 3)	1.68E-03	4.37E-04	3.85	6.64E-03	5.16E-04	12.86	2.66E-02	6.99E-04	38.11	1.07E-01	1.49E-03	71.74	4.31E-01	4.59E-03	93.95
Moving average filter (11 × 11)	1.77E-02	5.80E-04	30.49	7.18E-02	1.04E-03	68.97	2.89E-01	2.72E-03	106.53	1.16E+0	9.55E-03	121.97	4.66E+00	3.68E-02	126.64
Highpass filter (3 × 3)	1.76E-03	7.94E-06	221.58	7.04E-03	7.94E-06	886.31	2.82E-02	7.94E-06	3545.26	1.13E-01	7.94E-06	14171.98	4.55E-01	7.94E-06	57341.56
Gaussian filter (3 × 3)	1.60E-03	4.37E-04	3.66	6.64E-03	5.16E-04	12.86	2.66E-02	6.91E-04	38.44	1.06E-01	1.49E-03	71.66	4.29E-01	4.59E-03	93.46
Gaussian filter (11 × 11)	1.77E-02	5.80E-04	30.49	7.18E-02	1.03E-03	69.50	2.89E-01	2.72E-03	106.56	1.16E+0	9.55E-03	121.99	4.66E+00	3.68E-02	126.61
Dilation (3 × 3)	2.40E-04	1.59E-05	15.11	8.80E-04	2.38E-05	36.93	3.28E-03	5.56E-05	58.99	1.30E-02	1.99E-04	65.67	5.24E-02	7.86E-04	66.64
Dilation (11 × 11)	4.08E-04	1.59E-05	25.68	1.60E-03	5.56E-05	28.78	5.76E-03	1.35E-04	42.66	2.38E-02	2.46E-04	96.82	1.04E-01	9.21E-04	113.23
Erosion (3 × 3)	2.40E-04	7.94E-06	30.22	8.80E-04	1.59E-05	55.39	3.28E-03	5.56E-05	58.99	1.30E-02	2.07E-04	62.76	5.23E-02	7.94E-04	65.87
Erosion (11 × 11)	2.40E-04	1.59E-05	15.11	9.60E-04	3.18E-05	30.22	3.92E-03	7.15E-05	54.84	1.66E-02	2.14E-04	77.59	7.30E-02	8.18E-04	89.29
Opening (3 × 3)	3.20E-04	1.19E-04	2.69	1.20E-03	1.67E-04	7.19	4.64E-03	6.27E-04	7.39	1.86E-02	1.18E-03	15.86	7.53E-02	3.52E-03	21.39

ตารางที่ 3.1 เปรียบเทียบเวลาประมวลผลของชุดคำสั่งประมวลผลภาพ (วินาที) (ต่อ)

ชุดคำสั่ง	160 × 120			320 × 240			640 × 480			1280 × 960			2560 × 1920		
	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU	CPU	GPU	CPU/GPU
Opening (11 × 11)	3.20E-04	1.27E-04	2.52	1.28E-03	1.83E-04	7.01	5.28E-03	6.67E-04	7.91	2.21E-02	1.19E-03	18.53	9.51E-02	3.57E-03	26.61
Closing (3 × 3)	4.00E-04	1.35E-04	2.96	1.52E-03	1.91E-04	7.97	5.36E-03	6.67E-04	8.03	2.06E-02	1.21E-03	17.04	8.26E-02	3.59E-03	23.00
Closing (11 × 11)	2.40E-03	2.54E-04	9.44	8.80E-03	5.64E-04	15.60	2.17E-02	1.39E-03	15.60	6.20E-02	1.61E-03	38.43	2.56E-01	4.85E-03	52.67
Image addition	8.00E-05	7.94E-06	10.07	4.80E-04	7.94E-06	60.43	1.76E-03	7.94E-06	221.58	7.12E-03	7.94E-06	896.39	2.84E-02	7.94E-06	3575.48
Image subtraction	2.40E-04	7.94E-06	30.22	8.00E-04	7.94E-06	100.72	3.20E-03	7.94E-06	402.87	1.26E-02	7.94E-06	1591.34	5.04E-02	7.94E-06	6345.21
Image multiplication	8.00E-05	7.94E-06	10.07	5.60E-04	7.94E-06	70.50	2.08E-03	7.94E-06	261.87	8.56E-03	7.94E-06	1077.68	3.42E-02	7.94E-06	4310.71
Image division	2.40E-04	7.94E-06	30.22	8.00E-04	7.94E-06	100.72	3.12E-03	7.94E-06	392.80	1.22E-02	7.94E-06	1540.98	4.89E-02	7.94E-06	6154.85
Image logarithm	8.00E-04	7.94E-06	100.72	3.36E-03	7.94E-06	423.01	1.37E-02	7.94E-06	1722.27	5.46E-02	7.94E-06	6880.02	2.18E-01	2.38E-05	9155.95
Image exponential	3.28E-03	7.94E-06	412.94	1.30E-02	7.94E-06	1631.63	5.19E-02	7.94E-06	6537.58	2.07E-01	7.94E-06	26117.08	8.26E-01	7.94E-06	104006.05
Image square root	5.60E-04	7.94E-06	70.50	2.72E-03	7.94E-06	342.44	9.44E-03	7.94E-06	1188.47	3.75E-02	7.94E-06	4723.66	1.53E-01	7.94E-06	19278.36
Image absolute	8.00E-05	7.94E-06	10.07	4.00E-04	7.94E-06	50.36	1.60E-03	7.94E-06	201.44	6.56E-03	7.94E-06	825.88	2.66E-02	7.94E-06	3353.90
Global mean	8.00E-05	4.77E-04	*0.17*	1.60E-04	5.24E-04	*0.31*	6.40E-04	6.12E-04	1.05	2.56E-03	1.06E-03	2.41	1.01E-02	2.78E-03	3.63
Global variance	8.00E-05	5.64E-04	*0.14*	3.20E-04	6.12E-04	*0.52*	1.28E-03	7.15E-04	1.79	5.04E-03	1.22E-03	4.15	2.02E-02	3.12E-03	6.48
Global SD	8.00E-05	5.56E-04	*0.14*	3.20E-04	6.20E-04	*0.52*	1.28E-03	7.15E-04	1.79	5.12E-03	1.22E-03	4.21	2.02E-02	3.11E-03	6.50
Global skewness	1.60E-04	9.69E-04	*0.17*	6.40E-04	1.04E-03	*0.62*	2.64E-03	1.24E-03	2.13	1.05E-02	2.10E-03	5.00	4.20E-02	5.46E-03	7.70
Global kurtosis	2.40E-04	9.29E-04	*0.26*	9.60E-04	1.04E-03	*0.92*	3.60E-03	1.22E-03	2.94	1.44E-02	2.10E-03	6.84	5.75E-02	5.47E-03	10.51
Global maximum	3.10E-05	4.61E-04	0.07	8.00E-05	5.24E-04	*0.15*	4.00E-04	6.12E-04	*0.65*	1.68E-03	1.03E-03	1.63	6.72E-03	2.67E-03	2.52
Global minimum	3.10E-05	4.61E-04	0.07	8.00E-05	5.16E-04	*0.15*	4.00E-04	6.12E-04	*0.65*	1.68E-03	1.03E-03	1.63	6.72E-03	2.66E-03	2.53
Global range	8.00E-05	5.32E-04	*0.15*	1.60E-04	6.12E-04	*0.26*	5.60E-04	6.99E-04	*0.80*	2.08E-03	1.22E-03	1.71	8.41E-03	3.11E-03	2.71
เฉลี่ย	**1.54E-03**	**2.59E-04**	**5.92**	**6.18E-03**	**3.32E-04**	**18.62**	**2.43E-02**	**5.65E-04**	**43.07**	**9.71E-02**	**1.32E-03**	**73.46**	**3.90E-01**	**4.40E-03**	**88.52**



ภาพที่3.2 เวลาประมวลผลเฉลี่ยของชุดคำสั่งประมวลผลภาพที่รันบน CPU และ GPU



ภาพที่3.3 อัตราส่วนของเวลาประมวลผลเฉลี่ยของชุดคำสั่งที่รันบน CPU ต่อชุดคำสั่งที่รันบน GPU

ตารางที่ 3.1 แสดงเวลาการประมวลผลชุดคำสั่งประมวลผลภาพที่รันบน CPU และ GPU รวมทั้งอัตราส่วนเวลาประมวลผลที่ได้จากการรันบน CPU ต่อ GPU ซึ่งจะแสดงอัตราการเร่งความเร็ว ถ้าว่าค่านี้มีค่าเกิน 1 หมายความว่า ชุดคำสั่งที่รันบน GPU ทำงานได้รวดเร็วกว่าชุดคำสั่งที่รันบน CPU ถ้าค่า

ยิ่งมากแสดงว่าการใช้ GPU เร่งความชุดคำสั่งนั้นได้ผลมาก แต่ถ้าค่าต่ำกว่า 1 (ดังแสดงไว้เป็นเลขตัวเอียงสีแดง) หมายความว่าชุดคำสั่งนั้นๆทำงานบน CPU ได้รวดเร็วกว่า จากผลการทดลองในตารางที่ 3.1 พบว่าอัตราการเร่งความเร็วการประมวลผลนั้นแตกต่างกันไปตามชุดคำสั่งแต่จะเร่งความเร็วได้สูงขึ้นเมื่อภาพมีขนาดใหญ่ขึ้น จะเห็นว่าเมื่อใช้ภาพที่มีขนาด 160 × 120 จุดภาพ จะมี 8 ชุดคำสั่งที่รันบน CPU ได้รวดเร็วกว่า แต่จำนวนนี้จะลดลงเหลือ 7 และ 4 ชุดคำสั่งเมื่อเพิ่มขนาดภาพเป็น 320 × 240 และ 640 × 480 จุดภาพตามลำดับ และเมื่อใช้ภาพทดลองที่มีขนาด 1280 × 960 และ 2560 × 1920 จุดภาพ พบว่าทุกชุดคำสั่งประมวลผลภาพสามารถรันบน GPU ได้รวดเร็วกว่าทั้งหมด โดยอัตราการเร่งแตกต่างกันไปตามชุดคำสั่งโดยมีค่าตั้งแต่ ไม่กี่เท่าจนถึงแสนเท่า

ภาพที่ 3.2 เปรียบเทียบเวลาประมวลผลเฉลี่ยของชุดคำสั่งที่รันบน CPU และ GPU จากภาพจะพบว่า เวลาการประมวลผลชุดคำสั่งประมวลผลภาพของ CPU นั้นแปรผันตรงกับจำนวนจุดภาพ โดยเมื่อเพิ่มความกว้างและความสูงของภาพเป็น 2 เท่า จำนวนจุดภาพจะเพิ่มขึ้นเป็น 4 เท่า จะพบว่าเวลาการประมวลผลก็จะเพิ่มขึ้น 4 เท่าโดยประมาณเช่นกัน ในขณะที่ชุดคำสั่งที่รันบน GPU จะใช้เวลาการประมวลผลเพิ่มขึ้นเพียงเล็กน้อยเท่านั้น (1.28 เท่า ถึง 3.33 เท่า) ทั้งนี้เป็นเพราะการประมวลผลภาพบน CPU นั้นเป็นแบบเชิงลำดับ (sequential) โดยจะประมวลผลไปทีละจุดภาพๆตามลำดับ ในขณะที่การประมวลผลภาพบน GPU นั้นแบบขนานทำให้สามารถประมวลผลจุดภาพหลายๆจุดได้พร้อมๆกัน (ขึ้นกับจำนวน cores ใน GPU) จึงเป็นสาเหตุให้ GPU สามารถเร่งความเร็วได้สูงขึ้น เมื่อทำการประมวลผลภาพที่มีขนาดใหญ่ขึ้น (5.92 ถึง 88.52 เท่า) ดังแสดงไว้ในภาพที่ 3.3

ตารางที่ 3.2 เวลาที่ใช้ในการจอง คืน อ่านและเขียนข้อมูลบนหน่วยความจำของ GPU (วินาที)

การทำงาน	**ขนาดภาพ**				
	160 × 120	**320 × 240**	**640 × 480**	**1280 × 960**	**2560 × 1920**
การจองและคืนหน่วยความจำภาพ 1 ภาพ	3.97E-05	3.97E-05	3.97E-05	4.05E-04	6.27E-04
การเขียนข้อมูลภาพลงไปบนหน่วยความจำของ GPU	3.97E-05	5.56E-05	1.35E-04	4.21E-04	1.35E-03
การอ่านข้อมูลภาพจากหน่วยความจำของ GPU	4.77E-05	6.35E-05	1.35E-04	4.45E-04	1.57E-03

อย่างไรก็ตามการประมวลผลภาพบน GPU นั้นจะต้องมีการจองคืนหน่วยความจำบน GPU รวมทั้งการอ่าน-เขียนข้อมูลระหว่าง CPU และ GPU อีกด้วย ผู้วิจัยได้ทำการทดลองวัดเวลาประมวลผลเหล่านี้และแสดงไว้ในตารางที่ 3.2 จากผลการทดลองพบว่า เวลาในการจองและคืนหน่วยความจำนั้นมีค่าคงที่ทีขนาดภาพ 160 × 120, 320 × 240, 640 × 480 จุดภาพ (ประมาณ 40 มิลลิวินาที) และเพิ่มขึ้นเป็น 405 และ 627 มิลลิวินาทีเมื่อขนาดภาพเพิ่มขึ้นตามลำดับ ในขณะที่เวลาในการอ่านและเขียนข้อมูลระหว่าง CPU และ GPU นั้นเพิ่มตามขนาดภาพอย่างต่อเนื่อง ซึ่งเวลาในการอ่านเขียนข้อมูลนั้นจะต้องนำมาคิดด้วยในการเปรียบเทียบความเร็วของระบบโดยรวม (ขึ้นกับความถี่ในการอ่านเขียนภาพ กับจำนวนการรันชุดคำสั่งต่อภาพ) ในขณะที่เวลาในการจองและคืนหน่วยความจำนั้นแทบไม่มีผลอะไรต่อระบบสังเคราะห์โปรแกรม เพราะว่าสามารถทำการจองหน่วยความจำในการเก็บภาพครั้งเดียวก่อนการรันระบบสังเคราะห์โปรแกรมก็เพียงพอ

3.2 การทดลองเปรียบเทียบเวลาประมวลผลของระบบสังเคราะห์โปรแกรม

ผู้วิจัยได้ทดลองระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติโดยใช้การโปรแกรมเชิงพันธุกรรมเชิงเส้น (LGP: Linear genetic programming) [3] รวมกับเทคนิคที่เรียกว่า การโปรแกรมเชิงพันธุกรรมแบบโครงสร้างลำดับชั้น หรือ HSGP: Hierarchical structure genetic programming (รายละเอียดอยู่ในภาคผนวก ข.) [24] เพื่อเปรียบเทียบเวลาประมวลผลของระบบที่รันบน CPU อย่างเดียว และระบบที่ใช้ทั้ง CPU และ GPU โดยทำการทดลองระบบแต่ละแบบทั้งหมด 20 ครั้งและทำการวัดเวลาประมวลผลเป็นหน่วยวินาที ภาพฐานข้อมูลฝึกฝนที่ใช้คือ ฐานข้อมูลภาพวัชพืชในสนามหญ้า [26] โดยเป้าหมายของโปรแกรมที่ถูกสังเคราะห์คือ โปรแกรมที่สามารถแยกแยะพื้นที่ที่เป็นวัชพืชออกจากฉากหลังซึ่งเป็นสนามหญ้า ฐานข้อมูลฝึกฝนมีทั้งหมด 20 ภาพ แต่ละภาพมีขนาด 160 × 120 จุดภาพ ผู้วิจัยได้มีการกำหนดตัวแปรในการทดลองดังตารางที่ 3.3

ผลการทดลองเปรียบเทียบเวลาการประมวลผลถูกแสดงไว้ในตารางที่ 3.4 จากผลการทดลองพบว่า ระบบสังเคราะห์โปรแกรมซึ่งใช้ GPU ใช้เวลาประมวลผลประมาณสี่พันวินาทีในขณะที่ระบบสังเคราะห์โปรแกรมซึ่งไม่ใช้ GPU ใช้เวลาประมวลผลประมาณหนึ่งหมื่นสองพันวินาที โดยมีค่าเบี่ยงเบนมาตรฐานของแต่ละระบบอยู่ที่ประมาณ10%ของค่าเฉลี่ยเท่านั้น ค่าสูงสุดและต่ำสุดก็ชี้ให้เห็นว่าเวลาการประมวลผลมีความต่างกันอย่างชัดเจน ผลการทดลองชี้ให้เห็นว่าการใช้ GPU ในการช่วยประมวลผลภาพทำให้สามารถเร่งความเร็วของระบบสังเคราะห์ได้ประมาณ 3 เท่า เมื่อเทียบกับระบบสังเคราะห์โปรแกรมที่รันบน CPU เท่านั้น ซึ่งใกล้เคียงกับอัตราการเร่งความเร็วโดยเฉลี่ยจากการทดลองที่หนึ่ง ซึ่งมีค่า 5.92 เท่า (ดูบทที่ 3.1) โดยผลที่ต่างกันเกิดจากสองสาเหตุได้แก่ 1) ผลจากการทดลองแรกยังไม่ได้นับรวมเวลาในการอ่านและเขียนข้อมูลระหว่าง CPU และ GPU 2) ผลจากการทดลองที่หนึ่งเป็นการหาค่าเฉลี่ยโดยคิดสมมุติโอกาสการรันแต่ละคำสั่งมีค่าเท่าๆกัน ซึ่งต่างกับสิ่งที่เกิดขึ้นจริงในระบบสังเคราะห์โปรแกรมอัตโนมัติโดยใช้การคำนวณเชิงวิวัฒนาการซึ่งโอกาสในการรันบางจะมีค่าแตกต่าง

กันไปแล้วแต่ปัญหาและกลุ่มประชากรที่กำเนิดขึ้นมาในกระบวนการวิวัฒนาการ อย่างไรก็ตามหากทำการทดลองกับภาพที่มีขนาดใหญ่ขึ้น จะสามารถเร่งความได้สูงมากขึ้นอีก (ตามผลจากการทดลองที่หนึ่ง)

ตารางที่ 3.3 การกำหนดตัวแปรในการทดลองเปรียบเทียบเวลาประมวลผลโดยรวม

ตัวแปร	**การตั้งค่า**
จำนวนชั้นของ HSGP	3
จำนวนโหนดเรียนรู้ทั้งหมดใน HSGP	7
ขนาดประชากร (Population size)	50 (ต่อ 1 โหนดเรียนรู้)
จำนวนรุ่นสูงสุด (The maximum number of generations)	50 (ต่อ 1 โหนดเรียนรู้)
ความยาวของชุดคำสั่งในแต่โปรแกรม (Individual length)	10 (คงที่)
จำนวนรีจิสเตอร์ภาพ (Image register)	4
จำนวนรีจิสเตอร์ตัวเลข (Numerical register)	4
จำนวนชุดคำสั่งมูลฐาน (Primitive operators)	35
วิธีการ Crossover	Parameterized uniform crossover
อัตราการ Crossover (Crossover rate)	0.5
วิธีการ Mutation	แก้ไขค่าในแต่ละยีนส์แบบสุ่ม
อัตราการ Mutation (Mutation rate)	0.1
วิธีการคัดเลือก (Selection method)	การเลือกแบบทัวร์นาเมนต์ (Tournament selection)
ขนาดทัวร์นาเมนต์ (Tournament size)	5
เก็บโปรแกรมที่ดีที่สุด (Elitist)	ใช้

ตารางที่ 3.4 เวลาการประมวลผลของระบบสังเคราะห์โปรแกรมรู้จำวัตถุที่ใช้และไม่ใช้ GPU

เวลาการประมวลผล (วินาที)	**ค่าเฉลี่ย**	**ค่าเบี่ยงเบนมาตรฐาน**	**ค่าสูงสุด**	**ค่าต่ำสุด**
ระบบสังเคราะห์โปรแกรมรู้จำวัตถุซึ่งรันบน CPU	12,153	1,341	14,465	9,452
ระบบสังเคราะห์โปรแกรมรู้จำวัตถุซึ่งรันบน CPU และ GPU	4,184	448	5,117	3,403

ผลการทดลองชี้ให้เห็นว่าการนำ GPU มาใช้ในระบบสังเคราะห์โปรแกรมรู้จำวัตถุสามารถช่วยเร็วความเร็วได้จริง ซึ่งจากผลการทดลองข้างต้น GPU สามารถช่วยเร่งความเร็วได้ถึง 3 เท่า (สำหรับขนาดภาพ 160 × 120 จุดภาพ) อย่างไรก็ตามอัตราการเร่งความเร็วนั้นขึ้นอยู่กับปัจจัยหลายอย่าง เช่น วิธีการเขียนโปรแกรมเชิงขนาน ขนาดรูปภาพ ความเร็วของ CPU ความเร็วและจำนวน core ประมวลผลใน GPU

ขนาดรูปภาพ จากอุปกรณ์ที่ใช้ในการทดลองข้างต้นจะเห็นได้ว่า CPU ที่ใช้เป็นรุ่นที่ค่อนข้างจะมีความเร็วสูง (เมื่อเทียบกับ CPU รุ่นอื่นในตลาดในขณะทำการทดลอง) ในขณะที่ GPU ที่ใช้เป็นเพียงรุ่นกลางๆ (ในตลาดขณะนั้น) เท่านั้น ซึ่งหมายความมีความเป็นไปได้ที่จะเร่งความของระบบให้มากขึ้นได้อีก นอกจากนี้การใช้การ์ดแสดงผลภาพ GPU หลายๆการ์ดในคอมพิวเตอร์หนึ่งเครื่องก็เป็นอีกหนึ่งวิธีที่จะเพิ่มความสามารถในการประมวลผลเชิงขนานและจะช่วยเร่งความเร็วของระบบได้มากขึ้น

บทที่ 4

สรุปผลและงานวิจัยในขั้นต่อไป

ในโครงการวิจัยชิ้นนี้ ผู้วิจัยได้ศึกษาโครงสร้างของGPU วิธีการเขียนโปรแกรมสั่งงาน GPU โดยใช้ภาษา CUDA C รวมทั้งวิธีการประมวลผลภาพบน GPU ผู้วิจัยได้พัฒนาชุดคำสั่งประมวลผลภาพพื้นฐานซึ่งสามารถทำงานได้อย่างรวดเร็วบน GPU โดยชุดคำสั่งเหล่านี้ได้ถูกนำมาใช้เป็นชุดคำสั่งมูลฐานของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติ ซึ่งทำให้สามารถลดเวลาที่ใช้ในการสังเคราะห์โปรแกรมลงได้ ผู้วิจัยได้ทำการทดลอง 2 การทดลอง ได้แก่ 1) การทดลองเปรียบเทียบเวลาการประมวลผลชุดคำสั่งประมวลผลภาพที่รันบน CPU และ GPU และ 2) การทดลองเปรียบเทียบเวลาการประมวลผลโดยรวมของระบบสังเคราะห์โปรแกรมรู้จำวัตถุที่ใช้และไม่ใช้ GPU

ในการทดลองที่หนึ่ง ผู้วิจัยได้ทำการเปรียบเทียบเวลาประมวลผลของชุดคำสั่งที่รันบน CPU และ GPU โดยทำการเปลี่ยนแปลงขนาดของภาพทดลอง (160 × 120, 320 × 240 , 640 × 480 , 1280 × 960 , และ 2560 × 1920 จุดภาพ) จากผลการทดลองพบว่า อัตราการเร่งความเร็วของชุดคำสั่งประมวลผลภาพแตกต่างกันไปตามชุดคำสั่ง แต่จะมีค่าเพิ่มขึ้นเมื่อภาพมีขนาดใหญ่ขึ้น เมื่อใช้ภาพที่มีขนาดใหญ่ที่สุดในการทดลอง (2560 × 1920 จุดภาพ) อัตราการเร่งความเร็วมีค่าแปรผันตั้งแต่ ไม่กี่เท่าจนถึงแสนเท่าโดยประมาณ และเมื่อทำการหาอัตราการเร่งเฉลี่ย (สมมุติว่ารันชุดคำสั่งแต่ละคำสั่งอย่างละหนึ่งครั้ง) พบว่าอัตราการเร่งแปรผันตั้งแต่ 5.92 เท่าถึง 88.52 เท่าโดยแปรผันตามขนาดของรูปภาพที่ใช้ทดลอง ผลการทดลองชี้ให้เห็นว่าเวลาในการประมวลผลภาพบน CPU นั้นแปรผันตรงกับจำนวนจุดภาพในภาพ ในขณะที่ เวลาการในการประมวลผลภาพบน GPU นั้นเพิ่มขึ้นเพียงไม่กี่เท่า (1.28-3.33เท่า) เมื่อเพิ่มจำนวนจุดภาพเป็น 4 เท่า ซึ่งเป็นผลของการประมวลผลเชิงขนาน

ในการทดลองที่สอง ผู้วิจัยได้ทำการทดลองวัดเวลาประมวลผลโดยรวมของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติที่ใช้และไม่ใช้ GPU ผู้วิจัยได้กำหนดตัวแปรต่างๆในการทดลองให้เหมือนกันและทำการทดลองเป็นจำนวน 20 ครั้งและดูผลเชิงสถิติ ขนาดภาพที่ใช้ในการทดลองคือ 160 × 120 จุดภาพ ผลการทดลองชี้ให้เห็นว่าการใช้ GPU ในระบบสังเคราะห์โปรแกรมจะช่วยเร่งเวลาในการประมวลผลได้อย่างชัดเจน โดยระบบที่ใช้ GPU ใช้เวลาน้อยกว่าเป็น 3 เท่าของระบบสังเคราะห์โปรแกรมที่ไม่ใช้ GPU ซึ่งชี้ให้เห็นว่าการนำ GPU มาใช้นั้นสามารถช่วยเร่งความเร็วของระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติได้จริง และอัตราการเร่งความเร็วจะสูงขึ้นกว่านี้เมื่อใช้กับภาพทดลองที่มีขนาดใหญ่ขึ้น

ในอนาคตผู้วิจัยมีแผนที่จะพัฒนาระบบสังเคราะห์โปรแกรมรู้จำวัตถุอัตโนมัติซึ่งสามารถรันบน GPU หลายๆตัวได้ ระบบเช่นนี้จะทำให้สามารถประมวลผลเชิงขนานในระดับโปรแกรมได้ ซึ่งหมายถึงระบบสามารถวัดผลโปรแกรมรู้จำวัตถุอัตโนมัติหลายๆตัวในเวลาเดียวกันได้ ซึ่งจะช่วยให้สามารถลดเวลาการสังเคราะห์โปรแกรมลงได้มากขึ้นอีก

เอกสารอ้างอิง

[1] S. Aoki and T. Nagao, "Automatic construction of tree-structural image transformations using genetic programming," Proceedings of ICAIP-99, Venezia, Italy, pp.136-141, 1999.

[2] B. Bhanu, Y. Lin, and K. Krawiec, Evolutionary Synthesis of Pattern Recognition Systems, Springer-Verlag, USA, 2005.

[3] M. Brameier amd W. Banzhaf, Linear Genetic Programming, Springer-Verlag, New York, 2007.

[4] K.A. De Jong, Evolutionary Computation: A Unified Approach, MIT Press, 2006.

[5] M. Durkovic, M. Zwick, F. Obermeier, and K. Diepold, "Performance of optical flow techniques on graphics hardwares," Proc. ICME 2006, pp.241-244, 2006.

[6] M. Ebner, "Evolution of hierarchical translation-invariant feature detectors with an application to character recognition," Proceedings of DAGM-Symposium 1997, Braunschweig, Germany, pp.456-463, 1997.

[7] J. Fung and S. Mann, "Computer vision signal processing on graphics processing units," Proc. International Conference on Acoustic, Speech and Signal Processing (ICASSP'04), vol.5, pp.93-96, 2004.

[8] J. Fung and S. Mann, "Using multiple graphics cards as a general purpose parallel computer: application to computer vision," Proc. 17th International Conference on Pattern Recognition (ICPR'04), pp.805-808, 2004.

[9] R.C. Gonzalez and R.E. Woods, Digital Image Processing (3rd ed.), Pearson Education, 2010.

[10] D. Howard and S.C. Roberts, "A staged genetic programming strategy for image analysis," Proceedings of GECCO-99, Florida, USA, pp.1047-1052, 1999.

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

[12] A. Lippert, "NVIDIA GPU Architecture for General Purpose Computing", Available at: www.cs.wm.edu/~kemper/cs654/slides/nvidia.pdf

[13] D.B. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," Proc. Image Understanding Workshop, pp.121-130, 1981.

[14] J. Mairal, R. Keriven, and A. Chariot, "Fast and efficient dense variational stereo on GPU," Proc. 3rd International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06), pp.97-194, 2006.

[15] R.J. Meuth and D.C. Wunsch II, "Approximate dynamic programming and neural networks on game hardware," Proc. International Conferece on Neural Networks, pp.852-856, 2007.

[16] T. Nagao and S. Masunaga, "Automatic construction of image transformation processes using genetic algorithm," Proceedings of ICIP-96, vol.3, Lausanne, Switzerland, pp.731-734, 1996.

[17] R. Poli and W.B. Langdon, A Field Guide to Genetic Programming, Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008

[18] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2010.

[19] S. Shirakawa and T. Nagao, "Genetic image network (GIN): automatically construction of image processing," Proceedings of IWAIT-2007, Bangkok, Thailand, pp.643-648, 2007.

[20] A. Song, "Fast video analysis by genetic programming," Proceedings of IEEE Congress on Evolutionary Computation (CEC-08), Hong Kong, China, pp.3237-3243, 2008.

[21] A. Song and V. Ciesielski, "Texture segmentation by genetic programming," Evolutionary Computation, vol.16, no.4, pp.461-481, 2008.

[22] A. Teller and M. Veloso, "PADO: a new learning architecture for object recognition," Symbolic Visual Learning, K. Ikeuchi and M. Veloso (eds.), Oxford Univ. Press, pp.81-116, 1996.

[23] U. Watchareeruetai, "Hierarchical structure genetic programming with generation adaptable learning nodes," Proceedings of the Forth AUN/SEED-Net Regional Conference on Information and Communication Technology, pp.222-228, Ho Chi Minh City, Vietnam, October 18-19, 2011.

[24] U. Watchareeruetai, T. Matsumoto, N. Ohnishi, H. Kudo, and Y. Takeuchi, "Acceleration of genetic programming by hierarchical structure learning: a case study on image recognition program synthesis," IEICE Transactions on Information and Systems, vol.E92-D, no.10, pp.2094-2102, October 2009.

[25] U. Watchareeruetai, T. Matsumoto, Y. Takeuchi, H. Kudo, and N. Ohnishi, "Multi-objective genetic programming with redundancy-regulations for automatic construction of image feature extractors," IEICE Transactions on Information and Systems, vol.E93-D, no.9, pp.2614-2625, September 2010.

[26] U. Watchareeruetai, Y. Takeuchi, T. Matsumoto, H. Kudo, and N. Ohnishi, "Computer vision based methods for detecting weeds in lawns," Machine Vision and Applications, vol.17, no.5, pp. 287-296, October 2006.

[27] U. Watchareeruetai, Y. Takeuchi, T. Matsumoto, H. Kudo, and N. Ohnishi, "Efficient construction of image feature extraction programs by using linear genetic programming with fitness retrieval and intermediate-result caching," in Foundation of Computational Intelligence Volume 4: Bio-inspired Data Mining, A. Abraham et al. (eds), Springer-Verlag, pp.355-375, 2009.

[28] U. Watchareeruetai, Y. Takeuchi, T. Matsumoto, H. Kudo, and N. Ohnishi, "Evaluations of feature extraction programs synthesized by redundancy-removed linear genetic programming: a case study on lawn weed detection," Journal of Information Processing, vol.18, pp.164-174, April 2010.

[29] U. Watchareeruetai, Y. Takeuchi, T. Matsumoto, H. Kudo, and N. Ohnishi, "Redundancies in linear GP, canonical transformation, and its exploitation: a demonstration on image feature synthesis," Genetic Programming and Evolvable Machines, vol.12, no.1, pp.49-77, March 2011.

[30] M. Zhang, V. Ciesialski, and P. Andreae, "A domain-independent window approach to multiclass object detection using genetic programming," EURASIP Journal on Applied Signal Processing, vol.8, pp.841-859, 2003.

ภาคผนวก

ภาคผนวก ก. โค้ดโปรแกรมชุดคำสั่งประมวลผลภาพพื้นฐานบน GPU

1. ชุดคำสั่งการเปลี่ยนค่าระดับสีเทา

1.1 Image negative

```
//-----------------------------------------------------
__global__ void kernelImgNegative( unsigned char *in, unsigned char *out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                out[offset] = 255 - in[offset];
}
//-----------------------------------------------------
void gpuImgNegative( unsigned char* dev_in, unsigned char* dev_out, int width,
                                                                     int height ){
        const int N = width*height;
        kernelImgNegative<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                              ( dev_in, dev_out, N );
}
//-----------------------------------------------------
```

1.2 Histogram equalization

```
//-----------------------------------------------------
__global__ void kernelComputeHist( unsigned char *dev_in, unsigned int *dev_hist, int N){
//Atomic operation on global memory
//Require compute compatability 1.1 or higher

        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
        int stride = blockDim.x*gridDim.x;

        while( offset < N ){
                atomicAdd( &(dev_hist[ dev_in[offset] ]), 1 );

                offset += stride;
        }
}
//-----------------------------------------------------
__global__ void kernelGrayValueMapping( unsigned char *dev_in, unsigned char *dev_out,
                                        unsigned char *dev_tf_function, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
        int stride = blockDim.x*gridDim.x;

        while( offset < N ){
                dev_out[offset] =  dev_tf_function[ dev_in[offset] ];

                offset += stride;
        }
}
//-----------------------------------------------------
void gpuComputeHist( unsigned char* dev_in, unsigned int *dev_hist, int width,
                                                                     int height ){
        const int N = width*height;

        kernelComputeHist<<< 4, 256 >>>( dev_in, dev_hist, N );
}
//-----------------------------------------------------
void gpuHistEQ( unsigned char* dev_in, unsigned char* dev_out, int width, int height ){
        const int N = width*height;

        //Allocate memory for histogram
```

```
        unsigned int *dev_hist;
        cudaMalloc( (void**)&dev_hist, 256*sizeof( unsigned int ) );
        cudaMemset( dev_hist, 0, 256*sizeof( unsigned int ) );

        //Compute Histogram
        kernelComputeHist<<< 4, 256 >>>( dev_in, dev_hist, N );
        unsigned int hist[256];
        cudaMemcpy( hist, dev_hist, 256*sizeof( unsigned int ), cudaMemcpyDeviceToHost );

        //Find Mapping Function (Transformation Function)
        unsigned char tf_function[256];
        unsigned long sum = 0;
        float temp_sum;
        const float _255_N = (float)255.0 / N;

        for( int i=0; i<256; i++ ){
                sum += hist[i];
                temp_sum = sum*_255_N;
                tf_function[i] = (unsigned char)( temp_sum + 0.5 );
        }

        //Mapping the image with the transformation function
        unsigned char *dev_tf_function;
        cudaMalloc( (void**)&dev_tf_function, 256*sizeof( unsigned char ) );
        cudaMemcpy( dev_tf_function, tf_function, 256*sizeof( unsigned char ),
                        cudaMemcpyHostToDevice );

        kernelGrayValueMapping<<< 4, 256 >>>( dev_in, dev_out, dev_tf_function, N );

        //Free memory
        cudaFree( dev_hist );
        cudaFree( dev_tf_function );
}
//-----------------------------------------------------
```

2. ชุดคำสั่งการแปลงเป็นภาพสองระดับ

2.1 Thresholding

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelThresholding( T *in, T *out, int N, float th ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                if( (float)in[offset] >= th )
                        out[offset] = 255;
                else
                        out[offset] = 0;
}
//-----------------------------------------------------
template <typename T>
void gpuThresholding( T* dev_in, T* dev_out, int width, int height, float th ){
        const int N = width*height;
        kernelThresholding<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock
>>>( dev_in,                                  dev_out, N, th );
}
//-----------------------------------------------------
```

2.2 Adaptive thresholding

```
//-----------------------------------------------------
template <typename T>
void gpuMeanThresholding( T* dev_in, T* dev_out, int width, int height ){
        float mean = gpuGetMean( dev_in, width, height );
        gpuThresholding( dev_in, dev_out, width, height, mean );
}
//-----------------------------------------------------
```

3. ชุดคำสั่งการหาขอบ

3.1 Sobel edge detection

```
//-----------------------------------------------------
__global__ void kernelEdgeSobel( unsigned char* dev_in, unsigned char* dev_out,
                                                     const int width, const int height ){
        //2D thread blocks
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        const int grid_line_sizeX = blockDim.x * gridDim.x;
        int offset = x + y * grid_line_sizeX;

        if( x == 0 || x == width-1 || y == 0 || y == height-1){
                dev_out[ offset ] = 0;
        }
        else if( x < width && y < height ){
                short Gx, Gy;
                //finding Gx
                Gx = (short)(  dev_in[ offset - 1 - grid_line_sizeX ]
                               -   dev_in[ offset + 1 - grid_line_sizeX ]
                               + 2*dev_in[ offset - 1 ]

                               - 2*dev_in[ offset + 1 ]
                               +   dev_in[ offset - 1 + grid_line_sizeX ]
                               -   dev_in[ offset + 1 + grid_line_sizeX ] );
                //finding Gy
                Gy = (short)(  dev_in[ offset - grid_line_sizeX - 1 ]
                               + 2*dev_in[ offset - grid_line_sizeX ]
                               + dev_in[ offset - grid_line_sizeX + 1 ]


                               - dev_in[ offset + grid_line_sizeX - 1 ]
                               - 2*dev_in[ offset + grid_line_sizeX ]
                               - dev_in[ offset + grid_line_sizeX + 1 ] );

                short value = (abs(Gx) + abs(Gy));
                if( value >= 255 )
                        dev_out[ offset ] = 255;
                else if( value <= 0 )                   //no such cases
                        dev_out[ offset ] = 0;
                else
                        dev_out[ offset ] = (unsigned char)value;
        }
}
//-----------------------------------------------------
template <typename T>
__global__ void kernelEdgeSobel( T* dev_in, float* dev_out, const int width,
                                                                    const int height ){
        //2D thread blocks
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        const int grid_line_sizeX = blockDim.x * gridDim.x;
        int offset = x + y * grid_line_sizeX;

        if( x == 0 || x == width-1 || y == 0 || y == height-1){
                dev_out[ offset ] = 0;
        }
        else if( x < width && y < height ){
                float Gx, Gy;
                //finding Gx
                Gx =   dev_in[ offset - 1 - grid_line_sizeX ]
                       -   dev_in[ offset + 1 - grid_line_sizeX ]
                       + 2*dev_in[ offset - 1 ]

                       - 2*dev_in[ offset + 1 ]
                       +   dev_in[ offset - 1 + grid_line_sizeX ]
                       -   dev_in[ offset + 1 + grid_line_sizeX ] ;

                //finding Gy
                Gy =   dev_in[ offset - grid_line_sizeX - 1 ]
                       + 2*dev_in[ offset - grid_line_sizeX ]
                       + dev_in[ offset - grid_line_sizeX + 1 ]
```

```
                         - dev_in[ offset + grid_line_sizeX - 1 ]
                       - 2*dev_in[ offset + grid_line_sizeX ]
                       - dev_in[ offset + grid_line_sizeX + 1 ] ;

                dev_out[ offset ] = (T)(abs(Gx) + abs(Gy));
        }
}
//------------------------------------------------------
```

4. ชุดคำสั่งตัวกรอง

4.1 Moving average filter

```
//------------------------------------------------------
template <typename T>
__global__ void kernelNaiveFiltering2D( T* dev_in, T* dev_out, const int width,
                                const int height, float* dev_filter, const int filterSize ){
        //2D thread blocks
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;

        if( x < width && y < height ){
                const int grid_line_sizeX = blockDim.x * gridDim.x;
                int offset = x + y * grid_line_sizeX;

                //convolution
                T value;
                float sum = 0;
                int m, n;
                const int R = (filterSize - 1)/2;       //filterSize must be odd
                for( int i=-R; i<=R; i++ ){
                        m = x + i;
                        if( m > 0 && m < width ){
                                for( int j=-R; j<=R; j++ ){
                                        n = y + j;
                                        if( n > 0 && n < height ){
                                                //get the corresponding pixel value
                                                value = dev_in[ m + n*grid_line_sizeX ];
                                                sum += value*dev_filter[ (i+R) +
                                                                (j+R)*filterSize ];
                                        }
                                }
                        }
                }
                dev_out[ offset ] = (T)(sum);
        }
}
//------------------------------------------------------
template <typename T>
void gpuMovingAverage( T* dev_in, T* dev_out, int width, int height, int filterSize ){
        dim3 blocks( ( width + tileSize - 1 )/tileSize,
                                ( height + tileSize - 1 )/tileSize );
        dim3 threads( tileSize, tileSize );

        const int filterLength1D = filterSize*filterSize;
        float *cpu_filter = (float*)malloc( filterLength1D*sizeof( float ) );
        for( int i=0; i < filterLength1D; i++ )
                cpu_filter[i] = 1.0/filterLength1D;
        float *dev_filter;
        cudaMalloc( (void**)&dev_filter, filterLength1D*sizeof( float ) );
        cudaMemcpy( dev_filter, cpu_filter, filterLength1D*sizeof( float ),
                        cudaMemcpyHostToDevice );

        kernelNaiveFiltering2D<<< blocks, threads >>>( dev_in, dev_out, width, height,
                                                                dev_filter, filterSize );

        cudaFree( dev_filter );
        free( cpu_filter );
}
//------------------------------------------------------
```

4.2 Highpass filter

```
//-----------------------------------------------------
__global__ void kernelHighpass3by3( float* dev_in, float* dev_out, int width, int height ){
//        [ -1 -1 -1 ]
//        [ -1  8 -1 ]
//        [ -1 -1 -1 ]

        //2D thread blocks
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        const int grid_line_sizeX = blockDim.x * gridDim.x;
        int offset = x + y * grid_line_sizeX;

        if( x == 0 || x == width-1 || y == 0 || y == height-1){
                dev_out[ offset ] = dev_in[ offset ];
        }
        else if( x < width && y < height ){
                float value;
                value = (      - dev_in[ offset - 1 - grid_line_sizeX ]
                               - dev_in[ offset - grid_line_sizeX ]
                               - dev_in[ offset + 1 - grid_line_sizeX ]

                               - dev_in[ offset - 1 ]
                               + 8*dev_in[ offset ]
                               - dev_in[ offset + 1 ]

                               - dev_in[ offset - 1 + grid_line_sizeX ]
                               - dev_in[ offset + grid_line_sizeX ]
                               - dev_in[ offset + 1 + grid_line_sizeX ] );

                dev_out[ offset ] = value;
        }
}
//-----------------------------------------------------
template <typename T>
void gpuHighpass3by3( T* dev_in, T* dev_out, int width, int height ){
        dim3 blocks( ( width + tileSize - 1 )/tileSize,
                               ( height + tileSize - 1 )/tileSize );
        dim3 threads( tileSize, tileSize );

        kernelHighpass3by3<<< blocks, threads >>>( dev_in, dev_out, width, height );
}
//-----------------------------------------------------
```

4.3 Gaussian filter

```
//-----------------------------------------------------
template <typename T>
void gpuGaussianFilter( T* dev_in, float* dev_out, int width, int height, int filterSize,
                                                                    const float sigma ){
        dim3 blocks( ( width + tileSize - 1 )/tileSize,
                               ( height + tileSize - 1 )/tileSize );
        dim3 threads( tileSize, tileSize );

        //create filter
        int i, j, x, y;
        const int filterSize_2 = (filterSize - 1)/2;
        const int filterLength1D = filterSize*filterSize;
        float value, sum = 0;
        float *cpu_filter = (float*)malloc( filterLength1D*sizeof( float ) );
        //filter coeffcient
        for( i=0, x=filterSize_2; i < filterSize; i++, x++ )
                for( j=0, y=filterSize_2; j < filterSize; j++, y++ ){
                        value = (float)(exp(-0.5*( sqrtf( x/sigma ) + sqrtf( y/sigma) ) ));
                        cpu_filter[ i + j*filterSize ] = value;
                        sum += value;
                }
        //normalize filter
        for( i=0; i < filterLength1D; i++ )
                cpu_filter[ i ] /= sum;

        //copy filter to GPU memory
        float *dev_filter;
```

```
        cudaMalloc( (void**)&dev_filter, filterLength1D*sizeof( float ) );
        cudaMemcpy( dev_filter, cpu_filter, filterLength1D*sizeof( float ),
                        cudaMemcpyHostToDevice );

        kernelNaiveFiltering2D<<< blocks, threads >>>( dev_in, dev_out, width, height,
                                                              dev_filter, filterSize );

        cudaFree( dev_filter );
        free( cpu_filter );
}
//-----------------------------------------------------
```

5. ชุดคำสั่งตัวดำเนินการ Morphology

5.1 Dilation

```
//-----------------------------------------------------
__global__ void kernelDilationSquare( unsigned char* dev_in, unsigned char* dev_out,
                              const int width, const int height, const int SE_size ){
        //2D thread blocks
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        const int grid_line_sizeX = blockDim.x * gridDim.x;
        int offset = x + y * grid_line_sizeX;

        //copy input image
        if( x < width && y < height ){
                dev_out[ offset ] = dev_in[ offset ];
        }
        __syncthreads();

        const int SE_size_2 = (SE_size - 1)/2;
        const int strX = SE_size_2;
        const int endX = width - SE_size_2;
        const int strY = SE_size_2;
        const int endY = height - SE_size_2;

        if( x >= strX && x <= endX && y >= strY && y <= endY )
                if( dev_in[ offset ] == GPU_OBJECT_VALUE ){
                        int m, n;
                        const int R = SE_size_2;        //SE_Size must be odd
                        for( int i=-R; i<=R; i++ ){
                                m = x + i;
                                for( int j=-R; j<=R; j++ ){
                                        n = y + j;
                                        dev_out[ m + n*grid_line_sizeX ] = GPU_OBJECT_VALUE;
                                }
                        }
                }
}
//-----------------------------------------------------
void gpuDilationSquare( unsigned char* dev_in, unsigned char* dev_out, int width,
                                                        int height, int SE_size ){
        dim3 blocks( ( width + tileSize - 1 )/tileSize,
                        ( height + tileSize - 1 )/tileSize );
        dim3 threads( tileSize, tileSize );

        kernelDilationSquare<<< blocks, threads >>>( dev_in, dev_out, width, height,
                                                                        SE_size );
}
//-----------------------------------------------------
```

5.2 Erosion

```
//-----------------------------------------------------
__global__ void kernelErosionSquare( unsigned char* dev_in, unsigned char* dev_out, const
                                     int width, const int height, const int SE_size ){
//2D thread blocks
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        const int grid_line_sizeX = blockDim.x * gridDim.x;
        int offset = x + y * grid_line_sizeX;
```

```
        //copy input image
        if( x < width && y < height ){
                dev_out[ offset ] = dev_in[ offset ];
        }
        __syncthreads();

        const int SE_size_2 = (SE_size - 1)/2;
        const int strX = SE_size_2;
        const int endX = width - SE_size_2;
        const int strY = SE_size_2;
        const int endY = height - SE_size_2;

        if( x >= strX && x <= endX && y >= strY && y <= endY )
                if( dev_in[ offset ] == GPU_OBJECT_VALUE ){
                        int m, n;
                        const int R = SE_size_2;      //SE_Size must be odd
                        for( int i=-R; i<=R; i++ ){
                                m = x + i;
                                for( int j=-R; j<=R; j++ ){
                                        n = y + j;
                                        if( dev_in[ m + n*grid_line_sizeX ] !=
                                                              GPU_OBJECT_VALUE ){
                                                dev_out[ offset ] = GPU_NON_OBJECT_VALUE;
                                                break;
                                        }
                                }
                        }
                }
}
//-----------------------------------------------------
void gpuErosionSquare( unsigned char* dev_in, unsigned char* dev_out, int width,
                                                            int height, int SE_size ){
        dim3 blocks( ( width + tileSize - 1 )/tileSize,
                                     ( height + tileSize - 1 )/tileSize );
        dim3 threads( tileSize, tileSize );

        kernelErosionSquare<<< blocks, threads >>>( dev_in, dev_out, width,
                                                                   height, SE_size );
}
//-----------------------------------------------------
```

5.3 Opening

```
//-----------------------------------------------------
void gpuOpeningSquare( unsigned char* dev_in, unsigned char* dev_out, int width,
                                                            int height, int SE_size ){
        unsigned char *dev_temp;
        cudaMalloc( (void**)&dev_temp, width*height*sizeof( unsigned char ) );

        gpuErosionSquare( dev_in, dev_temp, width, height, SE_size );
        gpuDilationSquare( dev_temp, dev_out, width, height, SE_size );

        cudaFree( dev_temp );
}
//-----------------------------------------------------
```

5.4 Closing

```
//-----------------------------------------------------
void gpuClosingSquare( unsigned char* dev_in, unsigned char* dev_out, int width,
                                                            int height, int SE_size ){
        unsigned char *dev_temp;
        cudaMalloc( (void**)&dev_temp, width*height*sizeof( unsigned char ) );

        gpuDilationSquare( dev_in, dev_temp, width, height, SE_size );
        gpuErosionSquare( dev_temp, dev_out, width, height, SE_size );

        cudaFree( dev_temp );
}
//-----------------------------------------------------
```

6. ชุดคำสั่งเลขคณิตและฟังก์ชันของภาพ

6.1 Image addition

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelImgAddition( T* in1, T* in2, T* out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                out[offset] = in1[offset] + in2[offset];
}

//-----------------------------------------------------
template <typename T>
void gpuImgAddition( T* dev_in1, T* dev_in2, T* dev_out, int width, int height ){
        const int N = width*height;
        kernelImgAddition<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                        ( dev_in1, dev_in2, dev_out, width*height );
}
//------------------------------------------------------------------
```

6.2 Image subtraction

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelImgSubtraction( T* in1, T* in2, T* out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                out[offset] = in1[offset] - in2[offset];
}
//-----------------------------------------------------
template <typename T>
void gpuImgSubtraction( T* dev_in1, T* dev_in2, T* dev_out, int width, int height ){
        const int N = width*height;
        kernelImgSubtraction<<< (N+threadsPerBlock-1)/threadsPerBlock,
                        threadsPerBlock >>>( dev_in1, dev_in2, dev_out, width*height );
}
//-----------------------------------------------------
```

6.3 Image multiplication

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelImgMultiplication( T* in1, T* in2, T* out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                out[offset] = in1[offset] * in2[offset];
}
//-----------------------------------------------------
template <typename T>
void gpuImgMultiplication( T* dev_in1, T* dev_in2, T* dev_out, int width, int height ){
        const int N = width*height;
        kernelImgMultiplication<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock
                                >>>( dev_in1, dev_in2, dev_out, width*height );
}
//-----------------------------------------------------
```

6.4 Image division

```
//-----------------------------------------------------
__global__ void kernelImgDivision( unsigned char* in1, unsigned char* in2,
                                        unsigned char* out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
```

```
        if( offset < N )
                if( in2[offset] != 0 ) //denominator is 0
                        out[offset] = in1[offset] / in2[offset];
                else
                        out[offset] = 255;      //maximum value
                //      out[offset] = 0;        //as in the previous version
}
//----------------------------------------------------
template <typename T>
void gpuImgDivision( T* dev_in1, T* dev_in2, T* dev_out, int width, int height ){
        const int N = width*height;
        kernelImgDivision<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                        ( dev_in1, dev_in2, dev_out, width*height );
}
//----------------------------------------------------
```

6.5 Image logarithm

```
//----------------------------------------------------
template <typename T>
__global__ void kernelImgLogarithm( T* in, float* out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                if( in[offset] > 0 )
                        out[offset] = (float)log( (float)in[offset] );
                else
                        out[offset] = -FLT_MAX;         //minimum value (need to include
limits.h)
}
//----------------------------------------------------
template <typename T>
void gpuImgLogarithm( T* dev_in, float* dev_out, int width, int height ){
        const int N = width*height;
        kernelImgLogarithm<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                                ( dev_in, dev_out, width*height );
}
//----------------------------------------------------
```

6.6 Image exponential

```
//----------------------------------------------------
template <typename T>
__global__ void kernelImgExponential( T* in, float* out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                if( in[offset] > 0 )
                        out[offset] = (float)exp( (float)in[offset] );
}
//----------------------------------------------------
template <typename T>
void gpuImgExponential( T* dev_in, float* dev_out, int width, int height ){
        const int N = width*height;
        kernelImgExponential<<< (N+threadsPerBlock-1)/threadsPerBlock,
                                        threadsPerBlock >>>( dev_in, dev_out, width*height );
}
//----------------------------------------------------
```

6.7 Image square root

```
//----------------------------------------------------
template <typename T>
__global__ void kernelImgSquareRoot( T* in, float* out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                if( in[offset] >= 0 )
                        out[offset] = (float)sqrt( (float)in[offset] );
```

```
            else
                    out[offset] = (float)sqrt( -(float)in[offset] );     //PROTECTED!!!
}
//-----------------------------------------------------
template <typename T>
void gpuImgSquareRoot( T* dev_in, float* dev_out, int width, int height ){
        const int N = width*height;
        kernelImgSquareRoot<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                                    ( dev_in, dev_out, width*height );
}
//-----------------------------------------------------
```

6.8 Image absolute

```
//-----------------------------------------------------
__global__ void kernelImgAbsolute( float* in, float* out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;

        if( offset < N )
                if( in[offset] >= 0 )
                        out[offset] = in[offset] ;
                else
                        out[offset] = -in[offset] ;
}
//-----------------------------------------------------
void gpuImgAbsolute( float* dev_in, float* dev_out, int width, int height ){
        const int N = width*height;
        kernelImgAbsolute<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                                    ( dev_in, dev_out, width*height );
}
//-----------------------------------------------------
```

7. ชุดคำสั่งเชิงสถิติ

7.1 Global mean

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelGetMean( T* dev_in, float* dev_partial_out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
        int cacheIndex = threadIdx.x;

        __shared__ float cache[threadsPerBlock];

        //First step: in case of a very long array
        cache[cacheIndex] = 0;
        while(offset < N){
                cache[cacheIndex] += dev_in[offset];

                offset += blockDim.x*gridDim.x;
        }
        __syncthreads();

        //Second step: reduction inside a block
        //Note: threadsPerBlock must be the power of 2
        int i = blockDim.x/2;
        while( i != 0 ){
                if( cacheIndex < i )
                        cache[cacheIndex] += cache[cacheIndex + i];
                __syncthreads();
                i /= 2;
        }

        //Third step: return the maximum value of the block
        //(need further reduction in CPU side)
        if( cacheIndex == 0 )
                dev_partial_out[blockIdx.x] = cache[0];
}
//-----------------------------------------------------
template <typename T>
float gpuGetMean( T* dev_in, int width, int height ){
```

```
        const int N = width*height;
        int blocksPerGrid = (N+threadsPerBlock-1)/threadsPerBlock;

        //First step: Allocate an array to store the partial result
        float *dev_partial_out;
        cudaMalloc( (void**)&dev_partial_out, blocksPerGrid*sizeof( float ) );

        //Second step: call kernel to obtain the maximum value of in each block
        kernelGetMean<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                        ( dev_in, dev_partial_out, width*height );

        //Third step: copy the partial result onto CPU memory
        float *partial_sum = (float*)malloc( blocksPerGrid*sizeof( float ) );
        cudaMemcpy( partial_sum, dev_partial_out , blocksPerGrid*sizeof( float ),
                                                        cudaMemcpyDeviceToHost );

        //Forth step: find the maximum among the maximum value of all blocks
        float sum = partial_sum[0];
        for( int i=1; i<blocksPerGrid; i++ )
                        sum += partial_sum[i];

        //Free memory
        cudaFree( dev_partial_out );
        free(partial_sum);

        return (float)(sum/N);
}
//-----------------------------------------------------
```

7.2 Global variance

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelGetVariance( T* dev_in, float* dev_partial_out_sum, float*
                                        dev_partial_out_sum2, int N ){
        //VAR[X] = E[(X-E(X))^2]  (central moment order 3)
        //       = E[X^2]-E[X]^2

        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
        int cacheIndex = threadIdx.x;

        __shared__ float cache_sum[threadsPerBlock];
        __shared__ float cache_sum2[threadsPerBlock];

        //First step: in case of a very long array
        cache_sum[cacheIndex]  = 0;
        cache_sum2[cacheIndex] = 0;
        while(offset < N){
                cache_sum[cacheIndex]  += dev_in[offset];
                cache_sum2[cacheIndex] += dev_in[offset]*dev_in[offset];

                offset += blockDim.x*gridDim.x;
        }
        __syncthreads();

        //Second step: reduction inside a block
        //Note: threadsPerBlock must be the power of 2
        int i = blockDim.x/2;
        while( i != 0 ){
                if( cacheIndex < i ){
                        cache_sum[cacheIndex]  += cache_sum[cacheIndex + i];
                        cache_sum2[cacheIndex] += cache_sum2[cacheIndex + i];
                }
                __syncthreads();
                i /= 2;
        }

        //Third step: return the maximum value of the block
        //(need further reduction in CPU side)
        if( cacheIndex == 0 ){
                dev_partial_out_sum[blockIdx.x]  = cache_sum[0];
                dev_partial_out_sum2[blockIdx.x] = cache_sum2[0];
        }
}
```

```
//-----------------------------------------------------
template <typename T>
float gpuGetVariance( T* dev_in, int width, int height ){
       //VAR[X] = E(X-E(X)^2)          (central moment order 3)
       //              = E[X^2]-E[X]^2
       const int N = width*height;
       int blocksPerGrid = (N+threadsPerBlock-1)/threadsPerBlock;

       //First step: Allocate an array to store the partial result
       float *dev_partial_out_sum;
       cudaMalloc( (void**)&dev_partial_out_sum, blocksPerGrid*sizeof( float ) );
       float *dev_partial_out_sum2;
       cudaMalloc( (void**)&dev_partial_out_sum2, blocksPerGrid*sizeof( float ) );

       //Second step: call kernel to obtain the maximum value of in each block
       kernelGetVariance<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                      ( dev_in, dev_partial_out_sum, dev_partial_out_sum2, width*height );

       //Third step: copy the partial result onto CPU memory
       float *partial_sum = (float*)malloc( blocksPerGrid*sizeof( float ) );
       cudaMemcpy( partial_sum, dev_partial_out_sum , blocksPerGrid*sizeof( float ),
                      cudaMemcpyDeviceToHost );
       float *partial_sum2 = (float*)malloc( blocksPerGrid*sizeof( float ) );
       cudaMemcpy( partial_sum2, dev_partial_out_sum2 , blocksPerGrid*sizeof( float ),
                      cudaMemcpyDeviceToHost );

       //Forth step: find the maximum among the maximum value of all blocks
       double sum = partial_sum[0];
       double sum2 = partial_sum2[0];
       for( int i=1; i<blocksPerGrid; i++ ){
                      sum  += partial_sum[i];
                      sum2 += partial_sum2[i];
       }

       //Free memory
       cudaFree( dev_partial_out_sum );
       cudaFree( dev_partial_out_sum2 );
       free(partial_sum);
       free(partial_sum2);

       //VAR[X] = E(X-E(X)^2)          (central moment order 3)
       //              = E[X^2]-E[X]^2
       sum = sum/N;
       sum2 = sum2/N;
       return (float)( sum2 - sum*sum );
}
//-----------------------------------------------------
```

7.3 Global standard deviation

```
//-----------------------------------------------------
template <typename T>
float gpuGetSTD( T* dev_in, int width, int height ){
       double var = gpuGetVariance( dev_in, width, height );

       return (float)sqrt(var);
}
//-----------------------------------------------------
```

7.4 Global skewness

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelGetSkewness( T* dev_in, float* dev_partial_out, int N,
                                                       const float mean ){
       //Skewness = E[(X-E(X))^3]      (central moment order 3)

       //1D thread blocks
       int offset = threadIdx.x + blockIdx.x*blockDim.x;
       int cacheIndex = threadIdx.x;

       __shared__ float cache[threadsPerBlock];
```

```
        //First step: in case of a very long array
        float dif;
        cache[cacheIndex] = 0;
        while(offset < N){
                dif = dev_in[offset] - mean;
                cache[cacheIndex] += dif*dif*dif;

                offset += blockDim.x*gridDim.x;
        }
        __syncthreads();

        //Second step: reduction inside a block
        //Note: threadsPerBlock must be the power of 2
        int i = blockDim.x/2;
        while( i != 0 ){
                if( cacheIndex < i )
                        cache[cacheIndex] += cache[cacheIndex + i];
                __syncthreads();
                i /= 2;
        }

        //Third step: return the maximum value of the block
        //(need further reduction in CPU side)
        if( cacheIndex == 0 )
                dev_partial_out[blockIdx.x] = cache[0];
}
//-----------------------------------------------------
template <typename T>
float gpuGetSkewness( T* dev_in, int width, int height ){
        //Skewness = E[(X-E(X))^3]     (central moment order 3)

        const int N = width*height;
        int blocksPerGrid = (N+threadsPerBlock-1)/threadsPerBlock;

        //First step: Calculate the mean E[X]
        float mean = gpuGetMean( dev_in, width, height );

        //Second step: Allocate an array to store the partial result
        float *dev_partial_out;
        cudaMalloc( (void**)&dev_partial_out, blocksPerGrid*sizeof( float ) );

        //Third step: call kernel to obtain the maximum value of in each block
        kernelGetSkewness<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                ( dev_in, dev_partial_out, width*height, mean );

        //Forth step: copy the partial result onto CPU memory
        float *partial_sum = (float*)malloc( blocksPerGrid*sizeof( float ) );
        cudaMemcpy( partial_sum, dev_partial_out , blocksPerGrid*sizeof( float ),
                        cudaMemcpyDeviceToHost );

        //Fifth step: find the maximum among the maximum value of all blocks
        double sum = partial_sum[0];
        for( int i=1; i<blocksPerGrid; i++ )
                        sum += partial_sum[i];

        //Free memory
        cudaFree( dev_partial_out );
        free(partial_sum);

        return (float)(sum/N);
}
//-----------------------------------------------------
```

7.5 Global kurtosis

```
//------------------------------------------------------------
template <typename T>
__global__ void kernelGetKurtosis( T* dev_in, float* dev_partial_out, int N,
                                                        const float mean ){
        //Kurtosis = E[(X-E(X))^4]     (central moment order 4)

        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
        int cacheIndex = threadIdx.x;
```

```
        __shared__ float cache[threadsPerBlock];

        //First step: in case of a very long array
        float dif;
        cache[cacheIndex] = 0;
        while(offset < N){
                dif = dev_in[offset] - mean;
                dif = dif*dif;
                dif = dif*dif;
                cache[cacheIndex] += dif;

                offset += blockDim.x*gridDim.x;
        }
        __syncthreads();

        //Second step: reduction inside a block
        //Note: threadsPerBlock must be the power of 2
        int i = blockDim.x/2;
        while( i != 0 ){
                if( cacheIndex < i )
                        cache[cacheIndex] += cache[cacheIndex + i];
                __syncthreads();
                i /= 2;
        }

        //Third step: return the maximum value of the block
        //(need further reduction in CPU side)
        if( cacheIndex == 0 )
                dev_partial_out[blockIdx.x] = cache[0];
}
//-----------------------------------------------------
template <typename T>
float gpuGetKurtosis( T* dev_in, int width, int height ){
        //Kurtosis = E[(X-E(X))^4]     (central moment order 4)

        const int N = width*height;
        int blocksPerGrid = (N+threadsPerBlock-1)/threadsPerBlock;

        //First step: Calculate the mean E[X]
        float mean = gpuGetMean( dev_in, width, height );

        //Second step: Allocate an array to store the partial result
        float *dev_partial_out;
        cudaMalloc( (void**)&dev_partial_out, blocksPerGrid*sizeof( float ) );

        //Third step: call kernel to obtain the maximum value of in each block
        kernelGetKurtosis<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                        ( dev_in, dev_partial_out, width*height, mean );

        //Forth step: copy the partial result onto CPU memory
        float *partial_sum = (float*)malloc( blocksPerGrid*sizeof( float ) );
        cudaMemcpy( partial_sum, dev_partial_out , blocksPerGrid*sizeof( float ),
                        cudaMemcpyDeviceToHost );

        //Fifth step: find the maximum among the maximum value of all blocks
        double sum = partial_sum[0];
        for( int i=1; i<blocksPerGrid; i++ )
                        sum += partial_sum[i];

        //Free memory
        cudaFree( dev_partial_out );
        free(partial_sum);

        return (float)(sum/N);
}
//-----------------------------------------------------
```

7.6 Global maximum

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelGetMax( T* dev_in, T* dev_partial_out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
```

```
        int cacheIndex = threadIdx.x;

        __shared__ T cache[threadsPerBlock];

        //First step: in case of a very long array
        cache[cacheIndex] = dev_in[offset];
        int temp_offset = offset + blockDim.x*gridDim.x;
        while(temp_offset < N){
                if( cache[cacheIndex] < dev_in[temp_offset] )
                        cache[cacheIndex] = dev_in[temp_offset];

                temp_offset += blockDim.x*gridDim.x;
        }
        __syncthreads();

        //Second step: reduction inside a block
        //Note: threadsPerBlock must be the power of 2
        int i = blockDim.x/2;
        while( i != 0 ){
                if( cacheIndex < i )
                        if( cache[cacheIndex] < cache[cacheIndex + i])
                                cache[cacheIndex] = cache[cacheIndex + i];
                __syncthreads();
                i /= 2;
        }

        //Third step: return the maximum value of the block
        //(need further reduction in CPU side)
        if( cacheIndex == 0 )
                dev_partial_out[blockIdx.x] = cache[0];
}
//-----------------------------------------------------
template <typename T>
T gpuGetMax( T* dev_in, int width, int height ){
        const int N = width*height;
        int blocksPerGrid = (N+threadsPerBlock-1)/threadsPerBlock;

        //First step: Allocate an array to store the partial result
        T *dev_partial_out;
        cudaMalloc( (void**)&dev_partial_out, blocksPerGrid*sizeof( T ) );

        //Second step: call kernel to obtain the maximum value of in each block
        kernelGetMax<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                             ( dev_in, dev_partial_out, width*height );

        //Third step: copy the partial result onto CPU memory
        T *partial_max = (T*)malloc( blocksPerGrid*sizeof( T ) );
        cudaMemcpy( partial_max, dev_partial_out , blocksPerGrid*sizeof( T ),
                      cudaMemcpyDeviceToHost );

        //Forth step: find the maximum among the maximum value of all blocks
        T max = partial_max[0];
        for( int i=1; i<blocksPerGrid; i++ )
                if( max < partial_max[i] )
                        max = partial_max[i];

        //Free memory
        cudaFree( dev_partial_out );
        free(partial_max);

        return max;
}
//-----------------------------------------------------
```

7.7 Global minimum

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelGetMin( T* dev_in, T* dev_partial_out, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
        int cacheIndex = threadIdx.x;

        __shared__ T cache[threadsPerBlock];
```

```
        //First step: in case of a very long array
        cache[cacheIndex] = dev_in[offset];
        int temp_offset = offset + blockDim.x*gridDim.x;
        while(temp_offset < N){
                if( cache[cacheIndex] > dev_in[temp_offset] )
                        cache[cacheIndex] = dev_in[temp_offset];

                temp_offset += blockDim.x*gridDim.x;
        }
        __syncthreads();

        //Second step: reduction inside a block
        //Note: threadsPerBlock must be the power of 2
        int i = blockDim.x/2;
        while( i != 0 ){
                if( cacheIndex < i )
                        if( cache[cacheIndex] > cache[cacheIndex + i])
                                cache[cacheIndex] = cache[cacheIndex + i];
                __syncthreads();
                i /= 2;
        }

        //Third step: return the minimum value of the block
        //(need further reduction in CPU side)
        if( cacheIndex == 0 )
                dev_partial_out[blockIdx.x] = cache[0];
}
//-----------------------------------------------------
template <typename T>
T gpuGetMin( T* dev_in, int width, int height ){
        const int N = width*height;
        int blocksPerGrid = (N+threadsPerBlock-1)/threadsPerBlock;

        //First step: Allocate an array to store the partial result
        T *dev_partial_out;
        cudaMalloc( (void**)&dev_partial_out, blocksPerGrid*sizeof( T ) );

        //Second step: call kernel to obtain the minimum value of in each block
        kernelGetMin<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                                        ( dev_in, dev_partial_out, width*height );

        //Third step: copy the partial result onto CPU memory
        T *partial_min = (T*)malloc( blocksPerGrid*sizeof( T ) );
        cudaMemcpy( partial_min, dev_partial_out , blocksPerGrid*sizeof( T ),
                        cudaMemcpyDeviceToHost );

        //Forth step: find the minimum among the minimum value of all blocks
        T min = partial_min[0];
        for( int i=1; i<blocksPerGrid; i++ )
                if( min > partial_min[i] )
                        min = partial_min[i];

        //Free memory
        cudaFree( dev_partial_out );
        free(partial_min);

        return min;
}
//-----------------------------------------------------
```

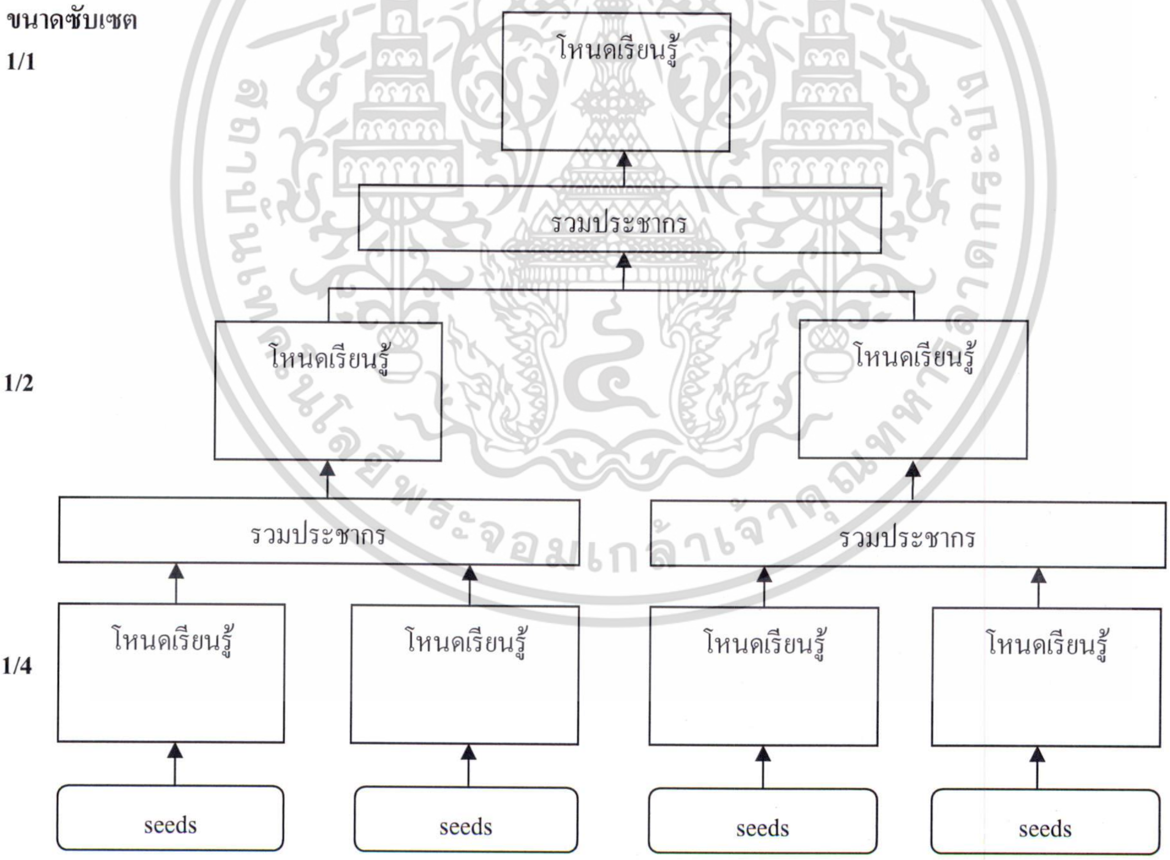
7.8 Global range

```
//-----------------------------------------------------
template <typename T>
__global__ void kernelGetMaxMin( T* dev_in, T* dev_partial_out_max,
                                                T* dev_partial_out_min, int N ){
        //1D thread blocks
        int offset = threadIdx.x + blockIdx.x*blockDim.x;
        int cacheIndex = threadIdx.x;

        __shared__ T cache_max[threadsPerBlock];
        __shared__ T cache_min[threadsPerBlock];

        //First step: in case of a very long array
```

```
        cache_max[cacheIndex] = dev_in[offset];
        cache_min[cacheIndex] = dev_in[offset];
        int temp_offset = offset + blockDim.x*gridDim.x;
        while(temp_offset < N){
                if( cache_max[cacheIndex] < dev_in[temp_offset] )
                        cache_max[cacheIndex] = dev_in[temp_offset];
                if( cache_min[cacheIndex] > dev_in[temp_offset] )
                        cache_min[cacheIndex] = dev_in[temp_offset];

                temp_offset += blockDim.x*gridDim.x;
        }
        __syncthreads();

        //Second step: reduction inside a block
        //Note: threadsPerBlock must be the power of 2
        int i = blockDim.x/2;
        while( i != 0 ){
                if( cacheIndex < i ){
                        if( cache_max[cacheIndex] < cache_max[cacheIndex + i])
                                cache_max[cacheIndex] = cache_max[cacheIndex + i];
                        if( cache_min[cacheIndex] > cache_min[cacheIndex + i])
                                cache_min[cacheIndex] = cache_min[cacheIndex + i];
                }
                __syncthreads();
                i /= 2;
        }

        //Third step: return the maximum value of the block
        //(need further reduction in CPU side)
        if( cacheIndex == 0 ){
                dev_partial_out_max[blockIdx.x] = cache_max[0];
                dev_partial_out_min[blockIdx.x] = cache_min[0];
        }
}
//-------------------------------------------------------
template <typename T>
T gpuGetRange( T* dev_in, int width, int height ){
        const int N = width*height;
        int blocksPerGrid = (N+threadsPerBlock-1)/threadsPerBlock;
        //First step: Allocate an array to store the partial result
        T *dev_partial_out_max;
        T *dev_partial_out_min;
        cudaMalloc( (void**)&dev_partial_out_max, blocksPerGrid*sizeof( T ) );
        cudaMalloc( (void**)&dev_partial_out_min, blocksPerGrid*sizeof( T ) );

        //Second step: call kernel to obtain the minimum value of in each block
        kernelGetMaxMin<<< (N+threadsPerBlock-1)/threadsPerBlock, threadsPerBlock >>>
                        ( dev_in, dev_partial_out_max, dev_partial_out_min, width*height );

        //Third step: copy the partial result onto CPU memory
        T *partial_max = (T*)malloc( blocksPerGrid*sizeof( T ) );
        T *partial_min = (T*)malloc( blocksPerGrid*sizeof( T ) );
        cudaMemcpy( partial_max, dev_partial_out_max , blocksPerGrid*sizeof( T ),
                        cudaMemcpyDeviceToHost );
        cudaMemcpy( partial_min, dev_partial_out_min , blocksPerGrid*sizeof( T ),
                        cudaMemcpyDeviceToHost );

        //Forth step: find the minimum among the minimum value of all blocks
        T max = partial_max[0];
        T min = partial_min[0];
        for( int i=1; i<blocksPerGrid; i++ ){
                if( max < partial_max[i] )
                        max = partial_max[i];
                if( min > partial_min[i] )
                        min = partial_min[i];
        }

        //Free memory
        cudaFree( dev_partial_out_max );
        cudaFree( dev_partial_out_min );
        free(partial_max);
        free(partial_min);

        return ( max - min );
}
//-------------------------------------------------------
```

ภาคผนวก ข. การโปรแกรมเชิงพันธุกรรมแบบโครงสร้างลำดับชั้น (HSGP)

การโปรแกรมเชิงพันธุกรรมแบบโครงสร้างลำดับชั้น หรือ HSGP (Hierarchical structure genetic programming) [24] เป็นเทคนิคที่ผู้วิจัยและคณะได้นำเสนอเมื่อปี ค.ศ. 2009 โดยมีจุดประสงค์เพื่อเร่งความเร็วของการโปรแกรมเชิงพันธุกรรม เทคนิคนี้จะอาศัยโหนดเรียนรู้ (learning node) ซึ่งต่อกันแบบโครงสร้างลำดับชั้นรูปแบบหนึ่ง เช่น โครงสร้างต้นไม้ทวิภาค (ภาพที่ ข.1) แต่ละโหนดเรียนรู้จะรันการโปรแกรมเชิงพันธุกรรมหนึ่งกระบวนการแล้วส่ง ประชากรที่วิวัฒนาการแล้ว ขึ้นไปยัง โหนดเรียนรู้ที่อยู่ติดกันด้านบน เพื่อผ่านกระบวนการวิวัฒนาการต่อไป แนวคิดหลักของเทคนิคก็คือ การลดจำนวน fitness cases ในช่วงต้นของกระบวนการวิวัฒนาการ (ซึ่งมักจะเกิดโปรแกรมที่ไม่สมเหตุสมผลเป็นจำนวนมาก) ระบบไม่จำเป็นจะต้องวัดผลโปรแกรมเหล่านี้โดยใช้ fitness cases ทุกรูปแบบ แต่ระบบเพียงใช้ซับเซตของ fitness cases ทั้งหมดในการประมาณค่าความเหมาะสมของโปรแกรมก็เพียงพอ



ภาพที่ ข.1 ตัวอย่างโครงสร้างลำดับชั้นทวิภาค 3 ชั้นของ *HSGP*

จากตัวอย่าง HSGP ในภาพที่ ข.1 สมมุติให้ **M** เป็นจำนวน fitness cases ในเซตฝึกฝนทั้งหมด N เป็นขนาดประชากร และ G เป็นจำนวนรุ่นที่จะถูกวิวัฒนาการ (N และ G เป็นค่าคงที่สำหรับทุกโหนดเรียนรู้) แต่โหนดเรียนรู้ในชั้นล่างสุดจะใช้ซับเซตซึ่งมี fitness cases ทั้งหมดเป็น M/4 (แต่ละซับเซตไม่ทับซ้อนกัน) ดังนั้นจำนวนครั้งของการรันโปรแกรมในแต่ละโหนดเรียนรู้ในชั้นล่างสุดจะเท่ากับ N × G × M/4 เช่นเดียวกันในโหนดเรียนรู้ของชั้นถัดไปจะใช้ซับเซตที่มีขนาดเป็น M/2 และจะรันโปรแกรมจำนวน N × G × M/2 ครั้ง ในขณะที่โหนดเรียนรู้ชั้นบนสุดจะใช้เซตของ fitness cases ทั้งหมดและจะรันโปรแกรมจำนวน N × G × M ครั้ง ซึ่งผลรวมจำนวนครั้งของการรันโปรแกรมทั้งหมดในเทคนิค HSGP ข้างต้นจะเป็น 3(N × G × M) ครั้ง โดยมีจำนวนรุ่นของการวิวัฒนาการรวมทั้งหมด 7G รุ่น (G รุ่นสำหรับแต่ละโหนดเรียนรู้) เมื่อเปรียบเทียบกับการโปรแกรมเชิงพันธุกรรมทั่วไปซึ่งใช้จำนวนครั้งในการรันโปรแกรมเท่ากันคือ 3(N × G × M) และมีจำนวน fitness cases เป็น M และขนาดประชากรเป็น G นั้น หมายความจำนวนรุ่นของการวิวัฒนาการจะมีเพียง 3G รุ่นเท่านั้น

จากผลการทดลองใน [24] พบว่าเมื่อใช้จำนวนในการรันโปรแกรมเท่ากัน HSGP จะสามารถสังเคราะห์โปรแกรมที่มีประสิทธิภาพสูงการโปรแกรมเชิงพันธุกรรมแบบปกติ ในอีกมุมหนึ่งถ้าหากเราต้องการสังเคราะห์โปรแกรมที่มีประสิทธิภาพสูงพอๆกัน เทคนิค HSGP จะใช้จำนวนครั้งของการรันโปรแกรมเพียง 60-70% ของการโปรแกรมเชิงพันธุกรรมแบบปกติเท่านั้น

ข้อมูลประวัติคณะผู้วิจัย

ประวัติส่วนตัว

ชื่อ-สกุล ดร. อุกฤษฏ์ วัชรีฤทัย

เพศ ■ ชาย ☐หญิง วันเดือนปีเกิด 28 ธันวาคม 2522 อายุ 32 ปี

สถานภาพ ■ โสด ☐ สมรส

ตำแหน่งปัจจุบัน อาจารย์ สาขาวิชาวิศวกรรมและเทคโนโลยี วิทยาลัยนานาชาติ

ประวัติการศึกษา

ชื่อย่อปริญญา	สาขา	สถาบันที่จบ	ปีที่จบ
D. Eng	Media Science	Graduate School of Information Science, Nagoya University	2553
MS. of Information Science	Media Science	Graduate School of Information Science, Nagoya University	2550
วศ.บ. (เกียรตินิยมอันดับหนึ่ง)	วิศวกรรมไฟฟ้า	คณะวิศวกรรมศาสตร์ มหาวิทยาลัยเกษตรศาสตร์	2545

สาขาวิจัยที่มีความชำนาญพิเศษ: การประมวลผลและรู้จำภาพ (Image Processing and Pattern Recognition), การโปรแกรมเชิงพันธุกรรม (Genetic Programming), ทัศนศาสตร์คอมพิวเตอร์เชิงวิวัฒนาการ (Evolutionary Computer Vision), ไบโอเมตริกส์ (Biometrics)

รางวัลด้านวิชาการ/ด้านวิจัย/งานสร้างสรรค์ (ด้านศิลปะ หรืออื่นๆ) ที่ได้รับ

ปี พ.ศ.	ชื่อรางวัล	สถาบันที่ให้
2554	IPSJ Digital Courier Funai Young Researcher Encouragement Award	Funai Foundation for Information Technology (FFIT)
2551	Presentation award (The 72nd Technical Meeting of IPSJ SIG-MPS (IPSJ MPS-72))	The Information Processing Society of Japan (IPSJ)
2549	Finalist for The Best Paper Award (2006 IEEE International Conference on Cybernetics and Intelligent Systems)	Institute of Electrical and Electronics Engineers (IEEE)

ทุนการศึกษาและทุนวิจัยที่เคยได้รับ

ปี พ.ศ.	ทุนการศึกษาและทุนวิจัย	สถาบันที่ให้
2553	ทุนสร้างสรรค์นวัตกรรมภาครัฐ	สำนักงานคณะกรรมการข้าราชการและพลเรือน (ก.พ.)
2551	ทุนสนับสนุนการวิจัยในหัวข้อ Automatic Extraction of Image Features Based on Evolutionary Computation	The Hori Information Science Promotion Foundation
2547	ทุนรัฐบาลญี่ปุ่น	The Ministry of Education, Culture, Sports, Science and Technology of Japan (MEXT)
2541	ทุนการศึกษา (สำหรับผู้ที่ได้คะแนนสอบเอ็นทรานซ์ลำดับที่ 1-6 ของคณะ)	คณะวิศวกรรมศาสตร์ มหาวิทยาลัยเกษตรศาสตร์

ผลงานวิจัย/งานสร้างสรรค์

ผลงานวิจัย/งานสร้างสรรค์ที่ตีพิมพ์เผยแพร่ (ระดับชาติและนานาชาติ)

1. Ukrit Watchareeruetai, Akisato Kimura, Robert Cheng Bao, Takahito Kawanishi, and Kunio Kashino, "Interest point detection based on stochastically derived stability," IPSJ Transactions on Computer Vision and Applications, vol.3, pp.186-197, December 2011.
2. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Redundancies in linear GP, canonical transformation, and its exploitation: a demonstration on image feature synthesis," Genetic Programming and Evolvable Machines, vol.12, no.1, pp.49-77, March 2011.
3. Ukrit Watchareeruetai and Noboru Ohnishi, "A new color-based lawn weed detection method and its integration with texture-based methods: a hybrid approach," IEEJ Transactions on Electronics, Information and Systems, vol.131, no.2, pp.355-366, February, 2011.
4. Ukrit Watchareeruetai, Tetsuya Matsumoto, Yoshinori Takeuchi, Hiroaki Kudo, and Noboru Ohnishi, "Multi-objective genetic programming with redundancy-regulations for automatic construction of image feature extractors," IEICE Transactions on Information and Systems, vol.E93-D, no.9, pp.2614-2625, September 2010.
5. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Evaluations of feature extraction programs synthesized by redundancy-removed linear

genetic programming: a case study on lawn weed detection," Journal of Information Processing, vol.18, pp.164-174, April 2010. (Reprinted in: Information and Media Technologies, vol.5, no.2, pp.566-576, June 2010) ***(IPSJ Digital Courier Funai Young Researcher Encouragement Award)***

6. Ukrit Watchareeruetai, Tetsuya Matsumoto, Noboru Ohnishi, Hiroaki Kudo, and Yoshinori Takeuchi, "Acceleration of genetic programming by hierarchical structure learning: a case study on image recognition program synthesis," IEICE Transactions on Information and Systems, vol.E92-D, no.10, pp.2094-2102, October 2009.
7. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Efficient construction of image feature extraction programs by using linear genetic programming with fitness retrieval and intermediate-result caching," in Foundation of Computational Intelligence Volume 4: Bio-inspired Data Mining, A. Abraham et al. (eds.), Springer-Verlag, pp.355-375, 2009. (ISBN 978-3-642-01087-3)
8. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Computer vision based methods for detecting weeds in lawns," Machine Vision and Applications, vol.17, no.5, pp. 287-296, October 2006.
9. อุกฤษฏ์ วัชรีฤทัย, "การสังเคราะห์เชิงวิวัฒนาการของโปรแกรมประมวลผลภาพเพื่อตรวจแยกพื้นม่านตา," NECTEC Technical Journal, ฉบับที่ 10 (22), หน้า 173-179, 2010.

การเสนอผลงานวิชาการ

1. Ukrit Watchareeruetai, "Hierarchical structure genetic programming with generation adaptable learning nodes," Proceedings of the Forth AUN/SEED-Net Regional Conference on Information and Communication Technology, pp.222-228, Ho Chi Minh City, Vietnam, October 18-19, 2011.
2. Ukrit Watchareeruetai, "A research on automatic construction of image processing programs," Proceedings of the Third AUN/SEED-Net Regional Conference in Electrical and Electronics Engineering, pp.122-126, Manila, Philippines, September 8-9, 2010.
3. Ukrit Watchareeruetai, Akisato Kimura, Robert Cheng Bao, Takahito Kawanisi, and Kunio Kashino, "StochasticSIFT: Interest point detection based on stochastically-derived stability," Proceedings of Meeting on Image Recognition and Understanding 2010 (MIRU2010), Kushiro, Japan, July 27-29, 2010.
4. Ukrit Watchareeruetai, Tetsuya Matsumoto, Yoshinori Takeuchi, Hiroaki Kudo, and Noboru Ohnishi, "Construction of image feature extractors based on multi-objective genetic programming

with redundancy regulations," Proceedings of 2009 IEEE Conference on Systems, Man, and Cybernetics (SMC2009), pp.1365-1370, Texas, USA, October 11-14, 2009.

5. Ukrit Watchareeruetai, Tetsuya Matsumoto, Noboru Ohnishi, Hiroaki Kudo, and Yoshinori Takeuchi, "Acceleration of genetic programming by hierarchical structure learning: a study on image recognition program synthesis," Learning and Intelligent Optimization Conference (LION3), Trento, Italy, January 14-18, 2009.
6. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Linear GP with redundancy-removed recombination for synthesis of image feature extraction programs," IPSJ SIG Technical Report (2008-MPS-72), vol.2008, no.126, pp.33-36, Osaka, Japan, December 17-18, 2008. (***Presentation Award***)
7. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Improving search performance of linear genetic programming based image recognition program synthesis by redundancy-removed recombination," Proceedings of 2008 IEEE Conference on Soft Computing in Industrial Applications (SMCia/08), pp.393-398, Muroran, Japan, June 25-27, 2008.
8. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Transformation of redundant representations of linear genetic programming into canonical forms for efficient extraction of image features," Proceedings of 2008 IEEE Congress on Evolutionary Computation (CEC-08), IEEE World Congress on Computational Intelligence, pp.1996-2003, Hong Kong, China, June 1-6, 2008.
9. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Modified lawn weed detection: utilization of edge-color based SVM and grass-model based blob inspection filterbank," Proceedings of 14th International Conference on Neural Information Processing (ICONIP-07) Part II, LNCS 4985, Springer-Verlag Berlin Heidelberg, pp.30-39, Kitakyushu, Japan, November 13-16, 2007.
10. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "A lawn weed detection in winter season based on color information," Proceedings of IAPR Conference on Machine Vision Applications (MVA 2007), pp.524-527, Tokyo, Japan, May 16-18, 2007.
11. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Lawn weeds detection methods based on image processing techniques," IEICE Technical Report of PRMU 2006, vol.106, no.301, pp.65-70, Tokyo, Japan, October 19-20, 2006.

12. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Computer vision based methods for detecting weeds in lawns," Proceedings of 2006 IEEE International Conference on Cybernetics and Intelligent Systems (CIS 2006), pp.323-328, Bangkok, Thailand, June 7-9, 2006 ***(Finalist for the Best Paper Award)***.
13. Vutipong Areekul,Teesid Leelasawassuk, Kittiwat Suppasriwasuseth, Suksan Jirachawang, and Ukrit Watchareeruetai, "Progress research on fingerprint verification algorithm in Thailand," Proceedings of 2006 Electrical Engineering/Electronics, Computer, Telecommunications, and Information Technology (ECTI) International Conference (ECTI-CON 2006), Ubon Ratchathani, Thailand, May 10-13, 2006.
14. Ukrit Watchareeruetai, Yoshinori Takeuchi, Tetsuya Matsumoto, Hiroaki Kudo, and Noboru Ohnishi, "Image processing based weed detection in lawn," 2005 Tokai-Section Joint Conference of the 8 Institutes of Electrical and Related Engineers, Nagoya, Japan, September 15-16, 2005.
15. Vutipong Areekul, Ukrit Watchareeruetai, Kittiwat Suppasriwasuseth, and Sawasd Tantaratana, "Separable Gabor filter realization for fast fingerprint enhancement," Proceedings of IEEE International Conference on Image Processing (ICIP 2005), vol. 3, pp.253-256, Genoa, Italy, September 11-14, 2005.
16. Vutipong Areekul, Ukrit Watchareeruetai, and Sawasd Tantaratana, "Fast separable Gabor filter for fingerprint enhancement," Proceedings of First International Conference on Biometrics Authentication (ICBA 2004), LNCS 3072, Springer-Verlag Berlin Heidelberg, pp.403-409, Hong Kong, China, July 17-19, 2004.
17. อุกฤษฏ์ วัชรีฤทัย, "วิธีการตรวจจับวัชพืชในสนามหญ้าฤดูหนาวโดยใช้กฎเกณฑ์ฟัซซี่," การประชุมทางวิชาการวิศวกรรมไฟฟ้าครั้งที่33 (EECON-33), เล่มที่ 2, หน้า 1237-1240, เชียงใหม่, ประเทศไทย, 1-3 ธันวาคม 2010
18. ธีรภัทร จำปีรัตน์ อุกฤษฏ์ วัชรีฤทัย กิตติวัฒน์ ศุภศรีวสุเศรษฐ์ และ วุฒิพงศ์ อารีกุล, "การเปรียบเทียบลายนิ้วมือโดยใช้คุณลักษณะต่างๆของมินูเทียร์ร่วมกับการอ้างอิงจุดโฟกัส," การประชุมทางวิชาการวิศวกรรมไฟฟ้าครั้งที่27 (EECON-27), เล่ม 2, หน้า 185-188, ขอนแก่น, ประเทศไทย, 11-12 พฤศจิกายน 2004
19. อุกฤษฏ์ วัชรีฤทัย และ วุฒิพงศ์ อารีกุล, "การตรวจสอบลายนิ้วมือโดยใช้การเปรียบเทียบการสุ่มเส้น," การประชุมทางวิชาการวิศวกรรมไฟฟ้าครั้งที่25 (EECON-25), หน้า 25-29, สงขลา, ประเทศไทย, 21-22 พฤศจิกายน 2002