การเพิ่มประสิทธิภาพในการหาขอบภาพโดยใช้ชุดคำสั่ง AVX บนสถาปัตยกรรมมัลติคอร์

Optimization of Edge Detection using AVX Intrinsics on Multi-core Architectures

เฌาฟิก เพ็งโอ

Thaufig Peng-o

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญา
วิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยสงขลานครินทร์

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Engineering in Computer Engineering
Prince of Songkla University

2565

ลิขสิทธิ์ของมหาวิทยาลัยสงขลานครินทร์

การเพิ่มประสิทธิภาพในการหาขอบภาพโดยใช้ชุดคำสั่ง AVX บนสถาปัตยกรรมมัลติคอร์
Optimization of Edge Detection using AVX Intrinsics on Multi-core Architectures

เฒาฟิก เพ็งโอ
Thaufig Peng-o

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญา
วิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยสงขลานครินทร์
A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Engineering in Computer Engineering
Prince of Songkla University
2565
ลิขสิทธิ์ของมหาวิทยาลัยสงขลานครินทร์

ชื่อวิทยานิพนธ์ การเพิ่มประสิทธิภาพในการหาขอบภาพโดยใช้ชุดคำสั่ง AVX บนสถาปัตยกรรมมัลติคอร์

ผู้เขียน นาย เฒาฟิก เพ็งโอ

สาขาวิชา วิศวกรรมคอมพิวเตอร์

อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก

....

(ผู้ช่วยศาสตราจารย์ ดร. ปัญญยศ ไชยกาฬ)

คณะกรรมการสอบ

....ประธานกรรมการ
(รองศาสตราจารย์ ดร. พิชญา ตัณฑัยย์)

....กรรมการ
(ดร. สมชัย หลิมศิโรรัตน์)

....กรรมการ
(ศาสตราจารย์ ดร. จันทนา จันทราพรชัย)

....กรรมการ
(ผู้ช่วยศาสตราจารย์ ดร. ปัญญยศ ไชยกาฬ)

บัณฑิตวิทยาลัย มหาวิทยาลัยสงขลานครินทร์ อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษา ตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์

....

(ผู้ช่วยศาสตราจารย์ ดร. เถกิง วงศ์ศิริโชติ)

รักษาการแทนคณบดีบัณฑิตวิทยาลัย

ขอรับรองว่า ผลงานวิจัยนี้มาจากการศึกษาวิจัยของนักศึกษาเอง และได้แสดงความขอบคุณบุคคลที่มีส่วนช่วยเหลือแล้ว

ลงชื่อ ....
(ผู้ช่วยศาสตราจารย์ ดร. ปัญญยศ ไชยกาฬ)
อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก

ลงชื่อ ....
(นาย เฌาฟิก เพ็งโอ)
นักศึกษา

ข้าพเจ้าขอรับรองว่า ผลงานวิจัยนี้ไม่เคยเป็นส่วนหนึ่งในการอนุมัติปริญญาในระดับใดมาก่อน และไม่ได้ถูกใช้ในการยื่นขออนุมัติปริญญาในขณะนี้

ลงชื่อ ....
(นาย เฒาฟิก เพ็งโอ)
นักศึกษา

ชื่อวิทยานิพนธ์	การเพิ่มประสิทธิภาพในการหาขอบภาพโดยใช้ชุดคำสั่ง AVX บนสถาปัตยกรรมมัลติคอร์
ผู้เขียน	นาย เฌาฟิก เพ็งโอ
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
ปีการศึกษา	2565

บทคัดย่อ

วิทยานิพนธ์นี้นำเสนออัลกอริทึมสำหรับเพิ่มความเร็วในการประมวลผลของการตรวจจับขอบภาพโดยวิธี Sobel และ Canny ด้วยการลดจำนวนการดำเนินการทางคณิตศาสตร์และการโหลดข้อมูลที่จะนำไปประมวลผล เพื่อลดระยะเวลาที่ใช้ในการประมวลผลภาพลง วิธีการที่นำเสนอนั้นใช้แนวทางการปรับปรุงซอฟต์แวร์เพียงอย่างเดียว โดยไม่อาศัยการใช้ฮาร์ดแวร์เร่งความเร็วใด ๆ ในการประมวลผล นอกจากนี้ผู้วิจัยได้นำชุดคำสั่ง AVX และ OpenMP เข้ามาช่วยในการเพิ่มความเร็วในการประมวลผลภาพให้เร็วยิ่งขึ้น การตรวจจับขอบภาพโดยวิธี Sobel ที่นำเสนอนั้นเร็วกว่าฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV โดยเฉลี่ย 28.29 เท่า เมื่อนำวิธีที่นำเสนอไปประยุกต์ใช้กับการตรวจจับขอบภาพของวิธี Canny วิธีการในอัลกอริทึมที่นำเสนอสามารถเพิ่มความเร็วของการตรวจจับขอบภาพได้เพิ่มขึ้นร้อยละ 3.73 เมื่อเปรียบเทียบกับการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV

Thesis Title Optimization of Edge Detection using AVX Intrinsics on Multi-core Architectures
Author Mr. Thaufig Peng-o
Major Program Computer Engineering
Academic Year 2022

Abstract

This thesis presents the algorithm for augmenting the processing speed of Sobel and Canny edge detection. By reducing the number of arithmetic operations and data loads, the processing time is reduced. Our proposed method is purely based on software approach which does not require any accelerated hardware. In addition, the processing speed is further increased by utilizing the AVX intrinsics and OpenMP. Our proposed Sobel edge detection is on average 28.29 times faster than the Sobel function provided by the OpenCV library. When applied with the Canny edge detection, our algorithm can augment the speed of OpenCV's Canny edge detection by 3.73 percent.

กิตติกรรมประกาศ

ขอขอบพระคุณ ผู้ช่วยศาสตราจารย์ ดร.ปัญญยศ ไชยกาฬ อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่ได้เสียสละเวลาในการให้คำปรึกษา แนวคิดในการทำวิจัย รวมถึงการช่วยเหลือแก้ไขปัญหาที่เกี่ยวกับการวิจัย ตลอดจนตรวจสอบและแก้ไขวิทยานิพนธ์ให้ดำเนินไปอย่างลุล่วงสมบูรณ์

ขอขอบพระคุณ รองศาสตราจารย์ ดร.พิชญา ตัณฑัยย์ ที่ได้กรุณาเสียสละเวลาเป็นประธานกรรมการสอบวิทยานิพนธ์ และให้ความช่วยเหลือในงานวิจัย ตลอดจนช่วยตรวจทานแก้ไขวิทยานิพนธ์ให้ดำเนินไปด้วยดี

ขอขอบพระคุณ ศาสตราจารย์ ดร.จันทนา จันทราพรชัย และ ดร.สมชัย หลิมศิโรรัตน์ ที่ได้กรุณาให้คำปรึกษา คำแนะนำ และตรวจทานแก้ไขวิทยานิพนธ์ให้มีความสมบูรณ์

ขอขอบพระคุณ บัณฑิตวิทยาลัย มหาวิทยาลัยสงขลานครินทร์ วิทยาเขตหาดใหญ่ ที่ให้การสนับสนุนทุนในการทำวิจัยและให้ความช่วยเหลือด้านการประสานงานต่าง ๆ

ขอขอบพระคุณ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยสงขลานครินทร์ ที่กรุณาให้ทุนผู้ช่วยวิจัยแก่ข้าพเจ้า

ขอขอบพระคุณ คณาจารย์ บุคลากร และนักศึกษาปริญญาโทภาควิชาวิศวกรรมคอมพิวเตอร์ทุกคนที่ได้ให้คำปรึกษา และกำลังใจในการทำงานเป็นอย่างดีเสมอมา

ขอขอบคุณ เบื้องหลังทุกอณูแห่งกำลังใจจาก คุณบาซีหย๊ะ ภรรยาผู้เป็นที่รักที่คอยให้การช่วยเหลือในการทำงานเป็นอย่างดีเสมอมา

และสุดท้าย ข้าพเจ้าขอน้อมรำลึกถึงพระคุณของมะ ที่คอยให้กำลังใจ เป็นห่วงเรื่องการศึกษาของลูก ขอความดีงามไปถึงปะ (รอหิมะฮูลลอฮฺ) ท่านกลับไปสู่ความเมตตาของพระเจ้าก่อนที่จะได้เห็นลูกเรียนจบปริญญาโท ขอขอบคุณพี่สาวคนโต คุณสาฝีน๊ะ พี่ ๆ หลาน ๆ และทุกคนในครอบครัว ที่เป็นกำลังใจ ส่งเสริมและสนับสนุนน้องชายคนสุดท้องในทุก ๆ เรื่องตลอดมาจนสำเร็จการศึกษา

เฒาฟิก เพ็งโอ

สารบัญ

บทคัดย่อ **(5)**
Abstract **(6)**
กิตติกรรมประกาศ **(7)**
สารบัญ **(8)**
สารบัญ(ต่อ) **(9)**
สารบัญ(ต่อ) **(10)**
รายการตาราง **(11)**
รายการภาพประกอบ **(12)**
รายการภาพประกอบ(ต่อ) **(13)**
รายการภาพประกอบ(ต่อ) **(14)**
สัญลักษณ์และคำย่อ **(15)**
บทที่ 1 บทนำ **1**
1.1 ความสำคัญและที่มาของวิทยานิพนธ์ 1
1.2 งานวิจัยที่เกี่ยวข้อง 2
1.3 วัตถุประสงค์ 4
1.4 ขอบเขตงานวิจัย 4
1.5 ขั้นตอนการวิจัย 5
1.6 ประโยชน์ที่คาดว่าจะได้รับ 5
1.7 ทรัพยากรที่ใช้ในระบบ 5
1.7.1 รายละเอียดของอุปกรณ์ที่ใช้ในการทดสอบ 5
1.7.2 ซอฟต์แวร์ที่ใช้ในการทดสอบ 5
บทที่ 2 ทฤษฎีและหลักการที่เกี่ยวข้อง **6**
2.1 การประมวลผลภาพ 6
2.2 การดำเนินการในการหาขอบภาพ 9
2.2.1 การหาขอบภาพโดยตัวดำเนินการโซเบล 9
2.2.2 การหาขอบภาพโดยวิธีการดำเนินการของพรีวิทท์ 10

สารบัญ (ต่อ)

2.2.3 การหาขอบภาพโดยวิธีการดำเนินการของโรเบิร์ต.......... 11

2.3 การหาขอบภาพโดยวิธีของแคนนี 11

2.4 การใช้ชุดคำสั่ง AVX 13

2.5 การเขียนโปรแกรมบนสถาปัตยกรรมมัลติคอร์ 15

บทที่ 3 แนวทางการวิจัย 18

3.1 การเพิ่มประสิทธิภาพในการคอนโวลูชันสำหรับการหาขอบภาพด้วยวิธีโซเบล 18

3.2 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการประมวลผลข้อมูลพร้อมกันหลายบรรทัด.......... 19

3.3 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการใช้ชุดคำสั่ง AVX.......... 21

3.3.1 การเขียนโปรแกรมโดยวิธีที่นำเสนอ สำหรับ *L*==2 21

3.3.2 การเขียนโปรแกรมโดยวิธีที่นำเสนอ สำหรับ *L*==3 25

3.3.3 การเขียนโปรแกรมโดยวิธีที่นำเสนอ สำหรับ *L*==4 28

3.4 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการใช้ OpenMP 31

3.5 การประยุกต์ใช้อัลกอริทึมที่นำเสนอในการเพิ่มประสิทธิภาพของการหาขอบภาพด้วยวิธีแคนนี.......... 31

บทที่ 4 การทดสอบและการวิเคราะห์ประสิทธิภาพ 32

4.1 ผลการทดสอบประสิทธิภาพในการหาขอบภาพด้วยวิธีโซเบลด้วยวิธีที่นำเสนอ 32

4.1.1 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 800x600 พิกเซล ที่ *L* ขนาดต่าง ๆ 33

4.1.2 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 1024x768 พิกเซล ที่ *L* ขนาดต่าง ๆ.......... 37

4.1.3 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 1280x1024 พิกเซล ที่ *L* ขนาดต่าง ๆ 40

4.1.4 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 1920x1080 พิกเซล ที่ *L* ขนาดต่าง ๆ 43

4.1.5 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 2560x1440 พิกเซล ที่ *L* ขนาดต่าง ๆ 46

สารบัญ (ต่อ)

4.1.6 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 3872x2160 พิกเซล ที่ *L* ขนาดต่าง ๆ 49

4.1.7 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 4928x3264 พิกเซล ที่ *L* ขนาดต่าง ๆ 52

4.1.8 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 7680x4320 พิกเซล ที่ *L* ขนาดต่าง ๆ 55

4.1.9 ผลการทดสอบการหาขอบภาพด้วยวิธีที่นำเสนอของภาพขนาดต่าง ๆ........................ 58

4.2 ผลการทดสอบการประยุกต์ใช้การเพิ่มประสิทธิภาพในการหาขอบภาพสำหรับแคนนี..... 59

4.3 การวิเคราะห์ผลการทดลอง 62

บทที่ 5 สรุปผลการวิจัยและข้อเสนอแนะ 64

5.1 สรุปผลการวิจัย.. 64

5.2 ข้อเสนอแนะ 65

บรรณานุกรม 66

ภาคผนวก 69

ประวัติผู้เขียน 88

รายการตาราง

ตารางที่ 2.1 ประเภทของการดำเนินการของรูปภาพ (Image Operations).................................... 6
ตารางที่ 4.1 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแนวนอน (Gx) และหาค่า Gradient ในแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาดต่าง ๆ ด้วยการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV 33
ตารางที่ 4.2 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (Gx) และหาค่า Gradient ในแกนแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพของภาพขนาด 800x600 พิกเซล ด้วยวิธีการที่นำเสนอ 34
ตารางที่ 4.3 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (Gx) และหาค่า Gradient ในแกนแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพของภาพขนาด 1024x768 พิกเซล ด้วยวิธีการที่นำเสนอ 37
ตารางที่ 4.4 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (Gx) และหาค่า Gradient ในแกนแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพของภาพขนาด 1280x1024 พิกเซล ด้วยวิธีการที่นำเสนอ.. 40
ตารางที่ 4.5 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (Gx) และหาค่า Gradient ในแกนแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพของภาพขนาด 1920x1080 พิกเซล ด้วยวิธีการที่นำเสนอ....................................... 43
ตารางที่ 4.6 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (Gx) และหาค่า Gradient ในแกนแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพของภาพขนาด 2560x1440 พิกเซล ด้วยวิธีการที่นำเสนอ.. 46
ตารางที่ 4.7 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (Gx) และหาค่า Gradient ในแกนแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพของภาพขนาด 3872x2160 พิกเซล ด้วยวิธีการที่นำเสนอ....................................... 49
ตารางที่ 4.8 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (Gx) และหาค่า Gradient ในแกนแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพของภาพขนาด 4928x3264 พิกเซล ด้วยวิธีการที่นำเสนอ 52
ตารางที่ 4.9 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (Gx) และหาค่า Gradient ในแกนแนวตั้ง (Gy) และระยะเวลาในการคำนวณหาขอบภาพของภาพขนาด 7680x4320 พิกเซล ด้วยวิธีการที่นำเสนอ.. 55
ตารางที่ 4.10 ระยะเวลาเฉลี่ย (ms) ที่ใช้ในการประมวลผลเพื่อหาขอบภาพโดยการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV และใช้วิธีการหาขอบภาพด้วยแคนนีที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรมและแบบขนาน 60

รายการภาพประกอบ

รูปที่ 1.1 ความเร็วที่เพิ่มขึ้นเมื่อเปรียบเทียบวิธีการต่าง ๆ 4
รูปที่ 2.1 การดำเนินการในการบวกภาพ 7
รูปที่ 2.2 การดำเนินการในการลบภาพ 7
รูปที่ 2.3 การดำเนินการในการคูณภาพ 8
รูปที่ 2.4 การดำเนินการในการหาค่าเฉลี่ยของภาพ 8
รูปที่ 2.5 ตัวอย่างการทำคอนโวลูชัน 9
รูปที่ 2.6 ตัวอย่างการคอนโวลูชันภาพจริง ของเทมเพลตขนาด 3 x 3 โดยใช้ OpenCV 9
รูปที่ 2.7 เคอร์เนลแนวนอนและแนวตั้งของการคอนโวลูชันของการหาขอบภาพโดยวิธีโซเบล 10
รูปที่ 2.8 ตัวอย่างการหาค่าความแรงของขอบภาพด้วยวิธีโซเบล 10
รูปที่ 2.9 เคอร์เนลแนวนอนและแนวตั้งของการคอนโวลูชันของการหาขอบภาพโดยวิธีพรีวิทท์ 11
รูปที่ 2.10 เคอร์เนลแนวนอนและแนวตั้งของการคอนโวลูชันของการหาขอบภาพโดยวิธีโรเบิร์ต 11
รูปที่ 2.11 ขั้นตอนการหาขอบภาพโดยวิธีแคนนี 11
รูปที่ 2.12 วิธีการแบ่งช่วงย่อยของทิศทางการเปลี่ยนแปลงค่าสี 12
รูปที่ 2.13 ตัวอย่างการดำเนินการกับตัวเลข 64 บิตจำนวน 4 ค่าพร้อม ๆ กันโดยชุดคำสั่ง AVX 13
รูปที่ 2.14 ตัวอย่างการเขียนโปรแกรมบวกตัวเลข 32 บิตจำนวน 8 ค่าพร้อม ๆ กันโดยชุดคำสั่ง AVX 14
รูปที่ 2.15 การแบ่งงานออกเป็นชิ้นเล็กให้แต่ละงานแก่ตัวประมวลผลหลาย ๆ ตัวในเวลาพร้อมกัน 15
รูปที่ 2.16 ลักษณะของเครื่องแบบหลายหน่วยประมวลผลที่ใช้หน่วยความจำร่วมกัน 16
รูปที่ 2.17 ตัวอย่างการเขียนโปรแกรมโดยใช้ OpenMP 17
รูปที่ 3.1 วิธีการทั่วไปในการคอนโวลูชันเพื่อหา Gradient magnitude 18
รูปที่ 3.2 วิธีการที่นำเสนอในการคอนโวลูชันเพื่อหา Gradient ในแนวนอน 19
รูปที่ 3.3 วิธีการที่นำเสนอในการคอนโวลูชันเพื่อหา Gradient ในแนวตั้ง 19
รูปที่ 3.4 การคำนวณหาขนาดของ Gradient magnitude โดยทั่วไป หรือแบบ *L*==1 20
รูปที่ 3.5 วิธีการคำนวณ Gradient ในแนวนอนที่นำเสนอ สำหรับ *L*==4 20
รูปที่ 3.6 วิธีการคำนวณ Gradient ในแนวตั้งที่นำเสนอ สำหรับ *L*==4 20
รูปที่ 3.7 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ภาษาซี สำหรับ *L*==2 21
รูปที่ 3.8 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ชุดคำสั่ง AVX สำหรับ *L*==2 23
รูปที่ 3.9 การแบ่งภาพขนาด 32 พิกเซล จากข้อมูล 8 บิต ให้เป็น 16 บิต เพื่อทำการประมวลผลโดยใช้ชุดคำสั่ง AVX 24
รูปที่ 3.10 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ภาษาซี สำหรับ *L*==3 25
รูปที่ 3.11 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ชุดคำสั่ง AVX สำหรับ *L*==3 26
รูปที่ 3.12 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ภาษาซี สำหรับ *L*==4 28
รูปที่ 3.13 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ชุดคำสั่ง AVX สำหรับ *L*==4 30
รูปที่ 3.14 การใช้ OpenMP สำหรับวิธีการที่นำเสนอ 31

รายการภาพประกอบ (ต่อ)

รูปที่ 4.1 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้งของภาพขนาด 800x600 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 35
รูปที่ 4.2 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 800x600 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 36
รูปที่ 4.3 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 1024x768 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 38
รูปที่ 4.4 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 1024x768 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 39
รูปที่ 4.5 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 1280x1024 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 41
รูปที่ 4.6 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 1280x1024 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 42
รูปที่ 4.7 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 1920x1080 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 44
รูปที่ 4.8 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 1920x1080 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 45
รูปที่ 4.9 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 2560x1440 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 47
รูปที่ 4.10 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 2560x1440 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 48
รูปที่ 4.11 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 3872x2160 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ.......... 50

รายการภาพประกอบ (ต่อ)

รูปที่ 4.12 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 3872x2160 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ 51
รูปที่ 4.13 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 4928x3264 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ 53
รูปที่ 4.14 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 4928x3264 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ 54
รูปที่ 4.15 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 7680x4320 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ 56
รูปที่ 4.16 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 7680x4320 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ 57
รูปที่ 4.17 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพด้วยวิธีการที่นำเสนอเปรียบเทียบกับการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV สำหรับภาพขนาดต่าง ๆ 58
รูปที่ 4.18 ภาพผลลัพธ์การหาขอบภาพโดยใช้วิธีโซเบลของภาพขนาด 800x600 พิกเซล 59
รูปที่ 4.19 ภาพผลลัพธ์การหาขอบภาพโดยใช้วิธีแคนนีของภาพขนาด 800x600 พิกเซล 59
รูปที่ 4.20 a: ภาพต้นฉบับ b: ผลลัพธ์ของการหาขอบภาพด้วยฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV c: ผลลัพธ์ของการหาขอบภาพด้วยวิธีแคนนีที่นำเสนอโดยการใช้ชุดคำสั่ง AVX 60
รูปที่ 4.21 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพด้วยวิธีการที่นำเสนอเปรียบเทียบกับการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV สำหรับภาพขนาดต่าง ๆ 61
รูปที่ 4.22 เปอร์เซ็นต์ของความเร็วที่เพิ่มขึ้นของการหาขอบภาพด้วยวิธีการที่นำเสนอเปรียบเทียบกับการใช้ฟังชัน Canny ในไลบรารีมาตรฐานของ OpenCV สำหรับภาพขนาดต่าง ๆ 62

สัญลักษณ์และคำย่อ

AVX	Advanced Vector Extension instructions
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
G_x	Gradient magnitude ในแนวนอน
G_y	Gradient magnitude ในแนวตั้ง
L	จำนวนบรรทัดของข้อมูลภาพที่ต้องการประมวลผลในแต่ละรอบ
MPI	Message Passing Interface
OpenCV	Open Source Computer Vision
OpenMP	Open Multi-Processing
SIMD	Single Instruction Multiple Data

บทที่ 1
บทนำ

เนื้อหาในบทนี้เป็นส่วนที่ให้ข้อมูลเบื้องต้นเพื่อให้ผู้อ่านเข้าใจถึงที่มาและความสำคัญของวิทยานิพนธ์รวมถึงเอกสารและงานวิจัยที่เกี่ยวข้องเพื่อเป็นแนวทางในการทำวิจัย โดยมีรายละเอียดของเนื้อหาประกอบด้วยหัวข้อย่อยจำนวน 7 หัวข้อ ซึ่งประกอบด้วยหัวข้อดังต่อไปนี้

1.1 ความสำคัญและที่มาของวิทยานิพนธ์
1.2 งานวิจัยที่เกี่ยวข้อง
1.3 วัตถุประสงค์
1.4 ขอบเขตการวิจัย
1.5 ขั้นตอนและวิธีดำเนินการวิจัย
1.6 ประโยชน์ที่คาดว่าจะได้รับ
1.7 ทรัพยากรที่ใช้ในระบบ

1.1 ความสำคัญและที่มาของวิทยานิพนธ์

การประมวลผลภาพ (Image Processing) เป็นงานที่นิยมใช้มากขึ้นในปัจจุบันเนื่องจากความสามารถของคอมพิวเตอร์มีสูงมากขึ้น ซึ่งได้มีการนำไปใช้งานหลายประเภท เช่น ระบบจดจำลายนิ้วมือเพื่อตรวจสอบภาพลายนิ้วมือ ระบบตรวจสอบคุณภาพของผลิตภัณฑ์ในกระบวนการผลิตของโรงงานอุตสาหกรรม ระบบคัดแยกเกรดหรือคุณภาพของพืชผลทางการเกษตร ระบบอ่านรหัสไปรษณีย์อัตโนมัติ เพื่อคัดแยกปลายทางของจดหมายที่มีจำนวนมากในแต่ละวันโดยใช้ภาพถ่ายของรหัสไปรษณีย์ที่อยู่บนซอง ระบบเก็บข้อมูลรถที่เข้าและออกอาคารโดยใช้ภาพถ่ายของป้ายทะเบียนรถเพื่อประโยชน์ในด้านความปลอดภัย ระบบดูแลและตรวจสอบสภาพการจราจร เป็นต้น จะเห็นว่าระบบเหล่านี้จำเป็นต้องใช้การประมวลผลภาพเป็นจำนวนมาก ซึ่งโดยปกติแล้วการประมวลผลภาพเป็นงานที่เสียเวลาและต้องใช้ตัวประมวลผลที่มีความสามารถสูงในการประมวลผลภาพ

การตรวจจับขอบภาพ (Edge detection) เป็นกระบวนการพื้นฐานที่สำคัญของการประมวลผลภาพ ซึ่งต้องใช้เวลาในการประมวลผลนานโดยเฉพาะเมื่อภาพมีขนาดใหญ่ [1] จึงทำให้การประมวลผลภาพแบบเรียลไทม์บางงานใช้การตรวจจับขอบภาพโดยใช้วิธีการทางฮาร์ดแวร์ ตัวอย่างเช่น การใช้ FPGA [2-5] หรือการใช้ GPU [6-8] เพื่อเพิ่มความเร็วในการประมวลผลภาพ แม้ว่าการใช้ฮาร์ดแวร์เร่งความเร็วจะช่วยเพิ่มความเร็วได้ดี แต่วิธีการนี้ต้องใช้ต้นทุนและการใช้พลังงานที่สูงขึ้น ปัจจุบันวิธีการพื้นฐานในการหาขอบภาพสามารถทำได้ในหลายวิธีด้วยกัน เช่น ตัวดำเนินการ Sobel, Prewitt และ Roberts โดยการหาขอบภาพด้วยวิธี Sobel เป็นวิธีที่ใช้กันอย่างแพร่หลาย เนื่องจากมีความซับซ้อนน้อยและใช้เวลาในการประมวลผลภาพน้อย[1] ซึ่งสามารถพบได้ทั้งการใช้งานในฮาร์ดแวร์และซอฟต์แวร์ และสามารถนำไปประยุกต์ใช้กับการตรวจจับขอบที่ซับซ้อนยิ่งขึ้นได้ เช่น การตรวจจับหาขอบภาพของวิธี Canny เนื่องจากมีความผิดพลาดในการหาขอบภาพน้อย และสามารถตรวจจับสัญญาณรบกวนได้ดี แม้ว่าจะใช้เวลานานในการประมวลผลภาพแต่ให้ประสิทธิภาพในการขอบภาพได้ดี [9]

ในสถาปัตยกรรมคอมพิวเตอร์ปัจจุบันมีแนวทางในการเพิ่มประสิทธิภาพในการประมวลผล 2 แนวทางหลักด้วยกัน [10-14] สำหรับแนวทางแรก คือ การใช้สถาปัตยกรรมมัลติคอร์ ซึ่งเป็นการเพิ่มจำนวนของหน่วยประมวลผลทางการคำนวณให้มีหลาย ๆ หน่วยประมวลผล โดยจะกระจายการทำงานไปยังหน่วยประมวลผลต่าง ๆ ในลักษณะการทำงานแบบขนาน ซึ่งเทคโนโลยีที่มักนำมาใช้ในการทำงานแบบขนานอย่างเช่น การใช้ MPI และ OpenMP ส่วนแนวทางที่ 2 คือ การใช้ชุดสั่งประเภท SIMD (Single Instruction Multiple Data) ในการประมวลผล ซึ่งเป็นการดำเนินการครั้งเดียวแล้วสามารถให้ผลได้หลายข้อมูลในเวลาเดียวกัน ซึ่ง Intel ได้นำชุดคำสั่ง AVX (Advanced Vector Extension instructions) มาปรับปรุงประสิทธิภาพบนสถาปัตยกรรมคอมพิวเตอร์ เพื่อช่วยให้คำสั่งสามารถจัดการกับองค์ประกอบข้อมูลหลายองค์ประกอบได้ ทำให้มีการประมวลผลที่รวดเร็วขึ้น ประสิทธิภาพที่สูงขึ้นและการจัดการข้อมูลที่มีประสิทธิภาพมากขึ้นในหลากหลายแอปพลิเคชันตัวอย่างเช่น การประมวลผลภาพ การจำลองทางวิทยาศาสตร์ การวิเคราะห์ทางการเงิน และการสร้างโมเดล 3D และการวิเคราะห์ข้อมูลที่มีขนาดใหญ่

ผู้วิจัยจึงมีความสนใจที่จะศึกษาการเพิ่มประสิทธิภาพของการประมวลผลในการหาขอบภาพโดยวิธีโซเบลด้วยชุดคำสั่ง AVX บนสถาปัตยกรรมมัลติคอร์ โดยมีจุดมุ่งหมายในการวิจัยเพื่อให้ระบบสามารถเพิ่มความเร็วในการประมวลผลภาพโดยไม่ต้องพึ่งพาฮาร์ดแวร์อื่น

1.2 งานวิจัยที่เกี่ยวข้อง

G. N. Chaple [1] ได้ศึกษาเปรียบเทียบข้อแตกต่างระหว่างตัวดำเนินการในการหาขอบภาพวิธี Sobel, Prewitt และ Roberts โดยได้ทำการทดลองประสิทธิภาพของวิธีการต่าง ๆ ด้วยการพัฒนาโปรแกรมโดยใช้ภาษา VHDL บนบอร์ด Field Programmable Gate Array (FPGA) ผลการทดลองพบว่าวิธี Roberts ใช้ฮาร์ดแวร์ในการออกแบบอัลกอริทึมน้อยและจำนวนของตัวดำเนินการในการหาขอบภาพน้อยกว่าวิธีการของ Sobel และ Prewitt เนื่องจากใช้เคอร์เนลในการคอนโวลูชันที่มีขนาดเล็กกว่า จึงให้ความละเอียดในการให้ขอบภาพที่น้อยกว่าวิธีอื่น ส่วนประสิทธิภาพในการตรวจจับหาขอบภาพของวิธี Sobel นั้นดีกว่าวิธีของ Prewitt และ Robert เนื่องจากสามารถตรวจจับหาขอบภาพได้ละเอียดกว่าวิธีอื่น

R. Menaka [2] ได้นำเสนอการออกแบบการตรวจจับหาขอบภาพโดยวิธีโซเบลบนบอร์ด FPGA โดยเลือกใช้ฮาร์ดแวร์ในการตรวจจับขอบภาพโดยวิธีโซเบลเนื่องจากโครงสร้างของอัลกอริทึมการตรวจจับขอบภาพด้วยวิธีโซเบลบนซอฟต์แวร์นั้นมีความคล้ายคลึงกันกับลักษณะของฮาร์ดแวร์และยังสามารถทำให้ลดสัญญารบกวนได้อีกด้วย R. Menaka จึงนำเสนอวิธีการตรวจจับหาขอบภาพโดยวิธีโซเบลบนบอร์ด FPGA ซึ่งผลการทดลองในการใช้ฮาร์ดแวร์ตรวจจับขอบภาพโดยวิธีโซเบลบนบอร์ด Xilinx spartan 6-XC6SLX45CSG324 FPGA สามารถลดการใช้พลังงานและเพิ่มความเร็วในการหาขอบของภาพได้

A. Jose [9] ได้ศึกษาเปรียบเทียบข้อแตกต่างระหว่างตัวดำเนินการในการหาขอบภาพวิธีการต่าง ๆ ได้แก่ Sobel, Prewitt, Roberts และ Canny โดยได้มีการทดลองเปรียบเทียบและพิจารณาบนพื้นฐานของพารามิเตอร์ต่าง ๆ ประกอบด้วย Peak Signal to Noise Ratio (PSNR), Mean Square Error (MSE), Maximum squared Error (MAXERR) และ L2RAT โดยใช้โปรแกรม MATLAB ในการทดลอง พบว่าวิธีการ Canny เหมาะสมที่สุดในการหาขอบภาพ เนื่องจากมีความผิดพลาดในการหาขอบภาพน้อย และสามารถตรวจจับสัญญาณรบกวนได้ดีกว่าวิธีอื่น แม้ว่าจะใช้เวลานานในการประมวลผลภาพแต่ให้ประสิทธิในการขอบภาพได้ดีที่สุด

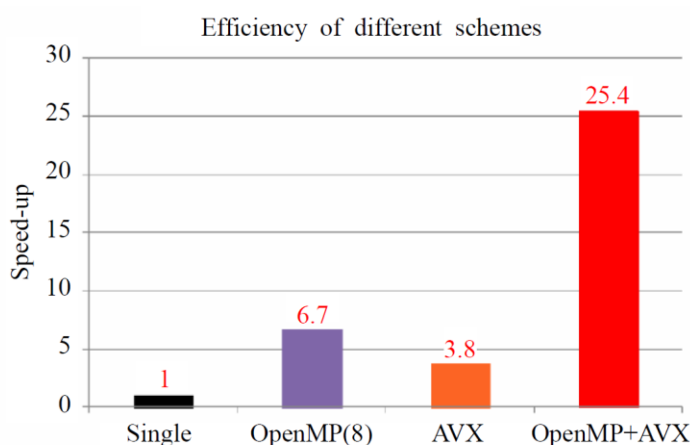
H. Amiri [10] ได้นำเสนอวิธีการคอนโวลูชันด้วยชุดคำสั่ง AVX และใช้ Intrinsic Programming Model (IPM) ในการพัฒนาความเร็วในการประมวลผล โดยได้นำเสนอวิธีในการประมวลผล 2 แบบ คือวิธีการ Broadcasting of coefficients (IPM-BC) และ Repetition of coefficients (IPM-RC) และนำมาเปรียบเทียบประสิทธิภาพความเร็วในการคอนโวลูชันโดยการใช้ Gnu Compiler Collection (GCC), Low Level Virtual Machine (LLVM) และ OpenCV โดยการทดสอบด้วยการใช้เคอร์เนลที่ขนาดแตกต่างกันคือ 3x3, 5x5, 7x7 และ 9x9 ในการคอนโวลูชัน ผลการทดสอบพบว่าวิธีการ IPM-BC และ IPM-RC มีความเร็วในการประมวลเร็วดีกว่าวิธีการต่าง ๆ ที่นำมาเปรียบเทียบ และวิธีการ IPM-BC มีความเร็วมากกว่า IPM-RC

Yonghong Yan [11] ได้นำเสนอการศึกษาประสิทธิภาพและการใช้พลังงานของอัลกอริทึมการคูณเมทริกซ์แบบขนานบนคอมพิวเตอร์มัลติคอร์ โดยใช้ OpenMP เขาได้เน้นถึงประสิทธิภาพด้านความเร็วในประมวลผล และประสิทธิภาพด้านพลังงานที่ส่งผลกระทบต่อหน่วยความจำ โดยได้ทำการทดสอบ 4 อัลกอริทึม ได้แก่ loop chucking, recursive tiling, hybrid tiling และ Strassen’s algorithms แล้วนำมาเปรียบเทียบกับการใช้ไลบรารีมาตรฐานของการคูณเมตริก Intel Math Kernel Library (MKL) โดยผลจากการทดลองอัลกอริทึมการคูณเมทริกซ์แบบขนาน loop chucking, recursive tiling, hybrid tiling , Strassen’s algorithms และ MKL ปรากฏว่า MKL มีประสิทธิภาพที่ดีกว่าอัลกอริทึมอื่น ๆ สำหรับการคูณเมทริกซ์ขนาดใหญ่กว่า 1000x1000 และมีจำนวนเธรดไม่เกิน 32 เธรด แต่หากจำนวนเธรดมากกว่า 32 เธรด อัลกอริทึม hybrid tiling มีความเร็วมากกว่าวิธีการอื่น ๆ ส่วนการเข้าถึงหน่วยความจำวิธีการ Strassen’s algorithms มีอัตราส่วนการเข้าถึงแคชที่ต่ำที่สุดจึงจะทำให้ประหยัดการใช้พลังงานได้ดีกว่า

Hassan [12] ได้นำเสนออัลกอลิทึมเพื่อปรับปรุงประสิทธิภาพของการคูณเมทริกซ์ด้วยวิธีการนำชุดคำสั่ง AVX มาใช้งานร่วมกับการทำ blocking การทำ loop un- rolling การทำ prefetch และได้นำ OpenMP มาใช้ในการกระจายงานให้ทำงานลักษณะแบบขนาด แล้วนำผลการทดสอบมาเปรียบเทียบกับวิธีการคูณเมทริกซ์ด้วยการใช้ไลเบอร์รี MKL ซึ่งเป็นไลเบอร์รีมาตรฐานในการคำนวณทางคณิตศาสตร์ของ Intel ผลการทดลองเปรียบเทียบปรากฏว่าวิธีการที่นำเสนอมีความเร็วที่เพิ่มขึ้นโดยเฉลี่ย 18.20 และ 14.10 เปอร์เซ็น สำหรับการคำนวณสมการ $y=Ax$ และ $y=A^Tx$ ตามลำดับ และในงานวิจัยนี้ได้นำเสนอผลทดลองเปรียบเทียบวิธีการเขียนโปรแกรมด้วยชุดคำสั่ง AVX แบบ Inline Assembly และ Intrinsic Function ผลการทดลองพบว่า วิธีการใช้

Intrinsic Function มีความเร็วมากกว่า Inline Assembly ในการคำนวณสมการ $y=Ax$ และ $y=A^Tx$ โดยเฉลี่ย 20.25 และ 30.15 เปอร์เซ็นต์ ตามลำดับ

Wenge Liu [13] ได้นำเสนอแบบจำลองสมการคลื่นเสียงที่ใช้หน่วยความจำและการคำนวณขั้นสูงแบบขนาน บนซีพียู 8 Core ด้วยการเขียนโปรแกรมในลักษณะที่แตกต่างกันคือ โปรแกรมแบบอนุกรม โปรแกรมแบบขนานโดย OpenMP โปรแกรมที่คำนวณในลักษณะ SIMD ด้วยการใช้ชุดคำสั่ง AVX และโปรแกรมที่ทำงานร่วมกันระหว่าง OpenMP และชุดคำสั่ง AVX ผลการทดลองประสิทธิภาพความเร็วที่เพิ่มขึ้นของวิธีการที่นำมาเขียนโปรแกรมเมื่อเปรียบเทียบกับวิธีการการเขียนโปรแกรมแบบอนุกรม พบว่าวิธีการเขียนโปรแกรมด้วย OpenMP มีความเร็วเพิ่มขึ้น 6.7 เท่า วิธีการเขียนโปรแกรมโดยใช้ชุดคำสั่ง AVX มีความเร็วเพิ่มขึ้น 3.8 เท่า และประสิทธิภาพของความเร็วที่เพิ่มขึ้นจากการประยุกต์ปรับปรุงการใช้งานของการใช้ OpenMP ร่วมกับการใช้ชุดคำสั่ง AVX สามารถเพิ่มประสิทธิภาพความเร็วได้ 25.4 เท่า ดังแสดงในรูปที่ 1.1



รูปที่ 1.1 ความเร็วที่เพิ่มขึ้นเมื่อเปรียบเทียบวิธีการต่าง ๆ ในการทดลองของ Wenge Liu [13]

1.3 วัตถุประสงค์

เพื่อเพิ่มประสิทธิภาพของการประมวลผลในการหาขอบภาพด้วยชุดคำสั่ง AVX ซึ่งทำงานบนสถาปัตยกรรมมัลติคอร์

1.4 ขอบเขตงานวิจัย

พัฒนาวิธีการเพิ่มประสิทธิภาพแอพพลิเคชั่นประมวลผลเพื่อหาขอบภาพ โดยใช้ชุดคำสั่ง AVX บนสถาปัตยกรรมมัลติคอร์

1.5 ขั้นตอนการวิจัย

ขั้นที่ 1: ศึกษาแนวทางและออกแบบวิธีการดำเนินงานวิจัย
ขั้นที่ 2: ศึกษาวิธีการหาขอบภาพ
ขั้นที่ 3: พัฒนาโปรแกรมที่ใช้ในการหาขอบภาพด้วยวิธีโซเบลที่นำเสนอโดยใช้ชุดคำสั่ง AVX
ขั้นที่ 4: ทดสอบการหาขอบภาพด้วยวิธีโซเบลที่นำเสนอเปรียบเทียบกับการใช้ฟังก์ชันโซเบลในไลบรารีมาตรฐานของ Open CV
ขั้นที่ 5: พัฒนาโปรแกรมที่ใช้ในการหาขอบภาพด้วยวิธีแคนนีที่นำเสนอโดยใช้ชุดคำสั่ง AVX
ขั้นที่ 6: ทดสอบการหาขอบภาพด้วยวิธีแคนนีที่นำเสนอเปรียบเทียบกับการใช้ฟังก์ชันแคนนีในไลบรารีมาตรฐานของ Open CV
ขั้นที่ 7: ปรับปรุงและทดสอบระบบทั้งระบบ
ขั้นที่ 8: สรุปผล จัดทำรายงานฉบับสมบูรณ์

1.6 ประโยชน์ที่คาดว่าจะได้รับ

สามารถเพิ่มประสิทธิภาพของแอพพลิเคชั่นประมวลผลภาพ โดยไม่ต้องใช้ฮาร์ดแวร์เร่งความเร็ว เช่น กราฟฟิกการ์ด ซึ่งใช้พลังงานในการประมวลผลสูง

1.7 ทรัพยากรที่ใช้ในระบบ

งานวิจัยฉบับนี้ ผู้วิจัยได้ทำการทดลองเพื่อเพิ่มประสิทธิภาพในการหาขอบภาพโดยใช้ชุดคำสั่ง AVX บนเครื่องคอมพิวเตอร์สถาปัตยกรรมมัลติคอร์ ซึ่งคุณสมบัติของอุปกรณ์และซอฟต์แวร์ที่ใช้สำหรับการทดสอบมีดังต่อไปนี้

1.7.1 รายละเอียดของอุปกรณ์ที่ใช้ในการทดสอบ

1. หน่วยประมวลผลกลาง (CPU) Intel Core i5-1135G7 2.40GHz
   จำนวน 4 Core และ 8 Thread
   หน่วยความจำแคช L1 จำนวน 4x48 Kbytes
   หน่วยความจำแคช L2 จำนวน 4x1.25 Mbytes
   หน่วยความจำแคช L3 จำนวน 8 Mbytes
2. หน่วยความจำหลัก (RAM) DDR4 16GB
3. หน่วยความจำสำรอง (Hard disk) SSD 1TB

1.7.2 ซอฟต์แวร์ที่ใช้ในการทดสอบ

1. ระบบปฏิบัติการ Windows 10 Pro 64Bit
2. โปรแกรม Visual Studio 2017
3. C++ compiler
4. ชุดคำสั่ง AVX (Advanced Vector Extensions)
5. OpenMP version 2.0
6. ไลบรารี Open Source Computer Vision 4.5.1 (OpenCV)

บทที่ 2
ทฤษฎีและหลักการที่เกี่ยวข้อง

เนื้อหาในบทนี้เป็นการนำเสนอเนื้อหาที่ผู้วิจัยได้ทำการศึกษาและรวบรวมเกี่ยวกับทฤษฎีและหลักการที่เกี่ยวข้องกับงานวิจัยเพื่อเพิ่มประสิทธิภาพในการหาขอบภาพโดยใช้ชุดคำสั่ง AVX บนสถาปัตยกรรมมัลติคอร์ ซึ่งมีการนำเสนอหัวข้อดังนี้

2.1 การประมวลผลภาพ
2.2 การดำเนินการในการหาขอบภาพ
2.3 การหาขอบภาพโดยวิธีของแคนนี
2.4 การใช้ชุดคำสั่ง AVX
2.5 การเขียนโปรแกรมบนสถาปัตยกรรมมัลติคอร์

2.1 การประมวลผลภาพ

การประมวลผลภาพเมื่อระบบได้รับข้อมูลภาพจะคำนวณและส่งออกมาเป็นข้อมูลที่ใช้แทนข้อมูลภาพ การเก็บข้อมูลภาพลงหน่วยความจำของคอมพิวเตอร์เก็บในลักษณะตัวแปรอาร์เรย์ ซึ่งเป็นโครงสร้างของข้อมูลได้หลายมิติ โดยค่าในแต่ละตำแหน่งแสดงถึงคุณสมบัติของจุดภาพ (pixel) หรือค่าความเข้มของสี (intensity) ทำให้ตำแหน่งของช่องอาร์เรย์เป็นตัวกำหนดตำแหน่งของจุดภาพ

ส่วนการจัดเก็บรูปภาพจะจัดเก็บข้อมูลของภาพไว้ในรูปของเมทริกซ์ สมมติว่าภาพมีขนาด *MxN* (แถวxคอลัมน์) ค่าของจุดภาพของภาพที่แทนด้วยเลขจำนวนเต็ม จะสามารถอ้างอิงกับเมทริกซ์ได้ ดังเช่น จุดภาพที่อยู่ ณ ตำแหน่งจุดกำเนิดมีค่า $(x, y) = (0,0)$ จะเท่ากับเมทริกซ์ แถว ที่ 0 คอลัมน์ที่ 0 และพิกัดที่อยู่แถวแรกมีค่า $(x, y) = (0,1)$ จะเท่ากับเมทริกซ์แถวที่ 0 คอลัมน์ ที่ 1 จะแสดงให้เห็นการอ้างอิงจุดภาพของภาพ

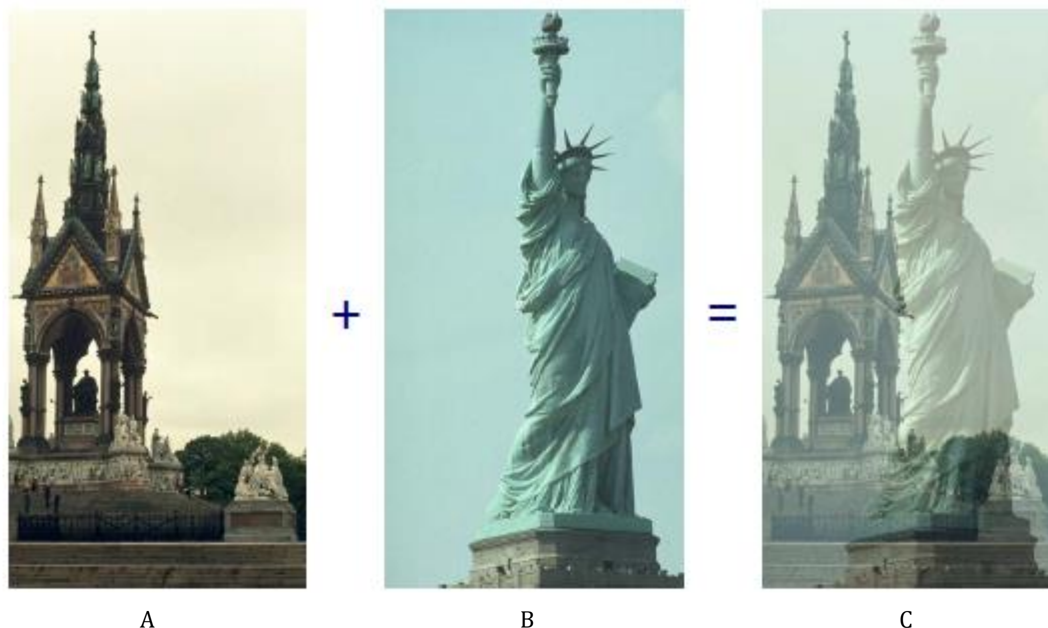
ประเภทของการดำเนินการของรูปภาพ[15] (Image Operations) ที่สามารถนำไปใช้กับภาพดิจิตอลเพื่อเปลี่ยนภาพอินพุต $a[m, n]$ ไปยังภาพเอาพุต $b[m, n]$ สามารถแบ่งออกเป็นสามประเภทดังตารางต่อไปนี้

ตารางที่ 2.1 ประเภทของการดำเนินการของรูปภาพ (Image Operations)

Operations	คุณลักษณะ	ความซับซ้อนทั่วไป / พิกเซล
Point	ค่าเอาต์พุตที่พิกัดเฉพาะนั้น ขึ้นอยู่กับค่าอินพุตที่พิกัดเดียวกันนั้นเท่านั้น	ค่าคงที่
Local	ค่าเอาต์พุตที่พิกัดเฉพาะนั้นขึ้นอยู่กับค่าอินพุตในพื้นที่ใกล้เคียงของพิกัดเดียวกันนั้น โดยกำหนดให้พื้นที่ใกล้เคียงดังกล่าวมีขนาด *PxP* พิกเซล	P^2
Global	ค่าเอาต์พุตที่พิกัดเฉพาะนั้นขึ้นอยู่กับค่าทั้งหมดในภาพอินพุต เมื่อภาพมีขนาด *NxN* พิกเซล	N^2

การดำเนินการของภาพเป็นพื้นฐานในการประมวลผลภาพและเป็นสิ่งที่มีความจำเป็นต่อการประมวลผลภาพดิจิตอล ดังแสดงตัวอย่างวิธีการดำเนินการของภาพ ต่อไปนี้

1. การดำเนินการในการบวกภาพ $C[x, y] = A[x, y] + B[x, y]$ ค่าเอาต์พุตที่พิกัดเฉพาะนั้นขึ้นอยู่กับค่าทั้งหมดในภาพอินพุต ดังแสดงตัวอย่างในรูปที่ 2.1

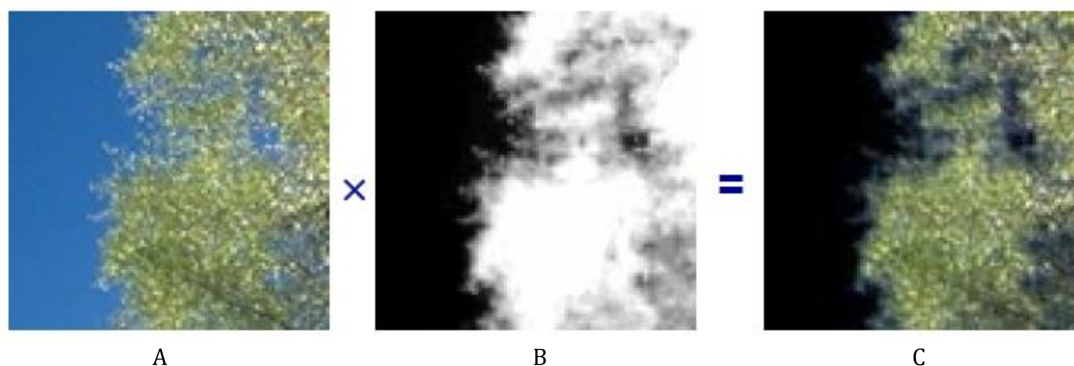


รูปที่ 2.1 การดำเนินการในการบวกภาพ [15]

2. การดำเนินการในการลบภาพ หรือ การหาความแตกต่างของรูปภาพ $C[x, y] = A[x, y] - B[x, y]$ ทำให้สามารถตัดพื้นหลังที่ไม่ต้องการออกได้ ดังแสดงตัวอย่างในรูปที่ 2.2

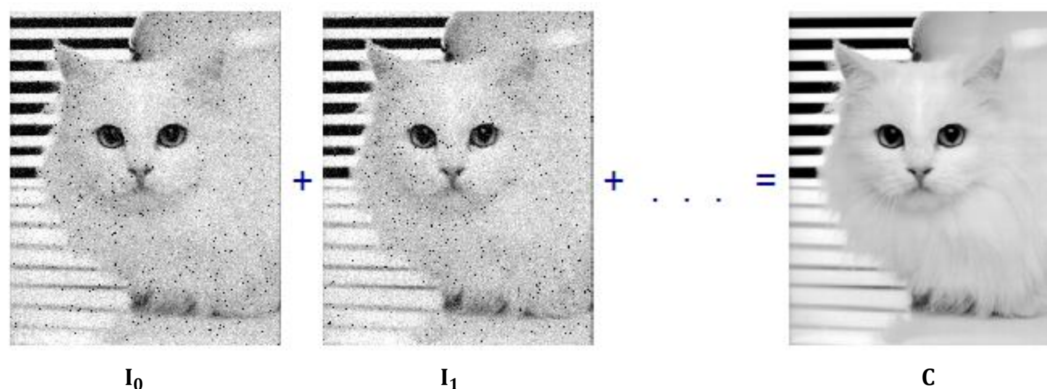
รูปที่ 2.2 การดำเนินการในการลบภาพ [15]

3. การดำเนินการในการคูณภาพ $C[x, y] = A[x, y] \times B[x, y]$ ใช้ประโยชน์สำหรับการ masking และการทำ alpha blending ดังแสดงตัวอย่างในรูปที่ 2.3



รูปที่ 2.3 การดำเนินการในการคูณภาพ [15]

4. การดำเนินการในการหาค่าเฉลี่ยของภาพ โดยการเอาหลาย ๆ ภาพที่เป็นภาพเดียวกันมาหาค่าเฉลี่ยด้วยกัน $C[x,y] = \frac{1}{N}\sum_{n=0}^{N} \mathrm{I_n}\,[x,y]$ ซึ่งมีประโยชน์สำหรับการตัดสัญญานรบกวนภายในภาพ ดังตัวอย่างในรูปที่



รูปที่ 2.4 การดำเนินการในการหาค่าเฉลี่ยของภาพ [15]

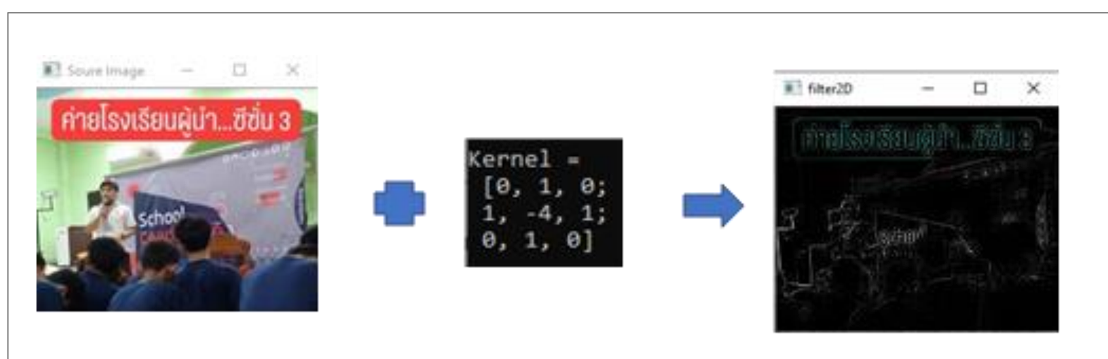
5. การคอนโวลูชัน (Convolution) ใช้สำหรับการกรองรูปภาพ คือกระทำกันระหว่างเคอร์เนล (Kernel) กับเทมเพลต (Template) เคอร์เนล คือ เมตริกซ์ขนาด *mxn* ของชุดตัวเลขที่จะนำไปซ้อนทับภาพที่ตำแหน่งต่าง ๆ เพื่อหาผลลัพธ์ของการคอนโวลูชัน ถ้ากำหนดให้เคอร์เนล $K(m,n)$ และเทมเพลต $T(x,y)$ การคอนโวลูชันระหว่างเทมเพลตกับเคอร์เนล มีสมการ คือ

$$OP(x,y) = \sum_{m=-a}^{a}\sum_{n=-b}^{b} K(m,n) * T(x-m, y-n) \tag{2.1}$$

วิธีการคอนโวลูชันที่ทำให้ภาพผลลัพธ์มีขนาดเท่ากับภาพเริ่มต้น คือการเติมค่าศูนย์บริเวณรอบ ๆ ภาพเริ่มต้น เพื่อทำให้ภาพเริ่มต้นมีขนาดใหญ่ขึ้น หลังจากการคอนโวลูชันจะได้ภาพผลลัพธ์ที่มีขนาดเท่ากับภาพเริ่มต้นเดิมดังในรูปที่ 2.5 และ 2.6

ภาพเริ่มต้น	ภาพเริ่มต้นหลังจากเติมศูนย์		เทมเพลต		ภาพผลลัพธ์
	0 0 0 0 0				
1 2 3	0 1 2 3 0		1 2 3		26 56 54
4 5 6	0 4 5 6 0	*	4 5 6	=	84 165 144
7 8 9	0 7 8 9 0		7 8 9		134 236 186
	0 0 0 0 0				

รูปที่ 2.5 ตัวอย่างการทำคอนโวลูชัน



รูปที่ 2.6 ตัวอย่างการคอนโวลูชันภาพจริง ของเทมเพลตขนาด 3 x 3 โดยใช้ OpenCV

2.2 การดำเนินการในการหาขอบภาพ

การหาขอบภาพเป็นพื้นฐานที่สำคัญของการประมวลผลภาพในหลายๆด้าน เช่น การแบ่งส่วนของภาพ (image segmentation) การกำหนดตำแหน่งของภาพ (image registration) และการรู้จำรูปแบบภาพ (pattern recognition) ซึ่งตัวดำเนินการตรวจจับหาขอบภาพประกอบด้วย

2.2.1 การหาขอบภาพโดยตัวดำเนินการโซเบล

การหาขอบภาพโดยตัวดำเนินการโซเบล (Sobel operator) เป็นเทคนิคที่ใช้กันอย่างแพร่หลายในการหาขอบภาพโดยหลักการพื้นฐานของวิธีนี้คือการคอนโวลูชัน (Convolution) ในแนวนอนและแนวตั้ง [16-18] ซึ่งเป็นวิธีที่ไม่ซับซ้อนและได้ผลการตรวจจับขอบที่ดี [19] การหาขอบภาพโดยใช้เคอร์เนลขนาด 3x3 จำนวนสองเคอร์เนล โดยเคอร์เนลแรกจะใช้เพื่อหาค่าความแตกต่างในแกนแนวนอน (G_x) และเคอร์เนลที่สองจะใช้เพื่อหาค่าความแตกต่างในแกนแนวตั้ง (G_y) ดังนี้

Kernel X

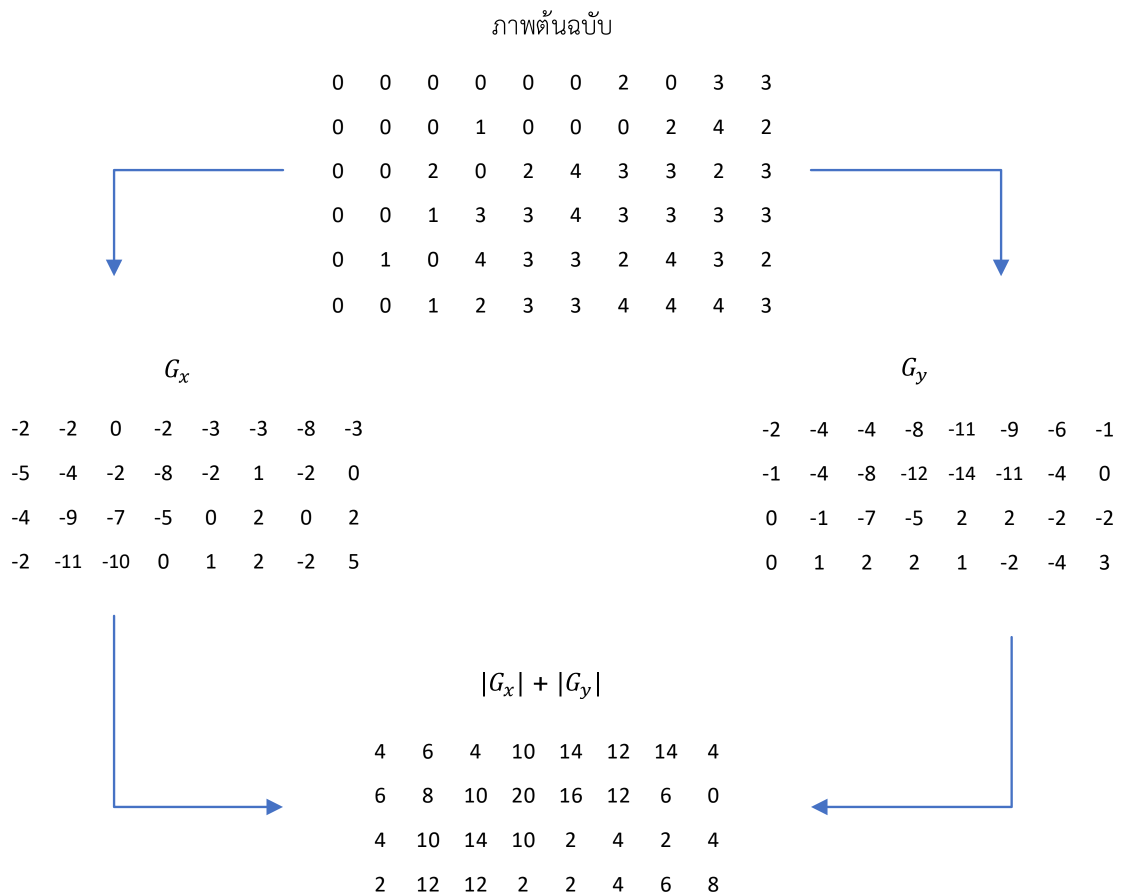
1	0	-1
2	0	-2
1	0	-1

Kernel Y

1	2	1
0	0	0
-1	-2	-1

รูปที่ 2.7 เคอร์เนลแนวนอนและแนวตั้งของการคอนโวลูชันของการหาขอบภาพโดยวิธีโซเบล

วิธีการหาขอบภาพโดยวิธีโซเบลทำได้โดยการนำเคอร์เนลทั้งสองไปทำการคอนโวลูชัน (Convolution) กับภาพ จากนั้นนำค่าสัมบูรณ์ของผลการคอนโวลูชันที่ได้มาบวกด้วยกัน จะได้ค่าความแรงของขอบภาพโดยวิธีโซเบล $|G_x|+|G_y|$ ดังตัวอย่างในรูปที่ 2.8



รูปที่ 2.8 ตัวอย่างการหาค่าความแรงของขอบภาพด้วยวิธีโซเบล

2.2.2 การหาขอบภาพโดยวิธีการดำเนินการของพรีวิทท์

การหาขอบภาพโดยวิธีการดำเนินการของพรีวิทท์ (Prewitt operator) ตัวดำเนินการ Prewitt มีความคล้ายคลึงกับเครื่องตรวจจับขอบของ Sobel โดยการใช้เคอร์เนลขนาด 3x3 มาทำ

การคอนโวลูชัน (Convolution) ในแนวนอนและแนวตั้งเพื่อหา Gradient ในแนวแกน x และแนวแกน y แสดงในรูปที่ 2.9

Kernel X

-1	0	1
-1	0	1
-1	0	1

Kernel Y

1	1	1
0	0	0
-1	-1	-1

รูปที่ 2.9 เคอร์เนลแนวนอนและแนวตั้งของการคอนโวลูชันของการหาขอบภาพโดยวิธีพรีวิทท์

2.2.3 การหาขอบภาพโดยวิธีการดำเนินการของโรเบิร์ต

การหาขอบภาพโดยวิธีการดำเนินการของโรเบิร์ต (Robert operator) เป็นวิธีการที่ง่ายต่อการคำนวณหาขอบภาพ โดยการใช้เคอร์เนลขนาด 2x2 ดังแสดงในรูปที่ 2.10 มาทำการคอนโวลูชัน (Convolution) ในแนวนอนและแนวตั้งเพื่อหา Gradient ในแนวแกน x และแนวแกน y

Kernel X

1	0
0	-1

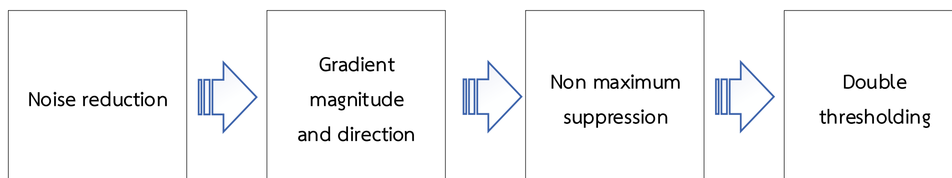
Kernel Y

0	-1
1	0

รูปที่ 2.10 เคอร์เนลแนวนอนและแนวตั้งของการคอนโวลูชันของการหาขอบภาพโดยวิธีโรเบิร์ต

2.3 การหาขอบภาพโดยวิธีของแคนนี

การหาขอบภาพโดยวิธีของแคนนี (Canny Edge Detection) [20-25] เป็นวิธีการตรวจจับขอบที่ซับซ้อน ให้ขอบข้อผิดพลาดต่ำและให้คุณภาพของการแบ่งส่วนภาพสูงมาก แต่ต้องใช้ระยะเวลาในการคำนวณเพิ่มมากขึ้น ต้องใช้กระบวนการ 4 ขั้นตอนหลักประกอบด้วย



รูปที่ 2.11 ขั้นตอนการหาขอบภาพโดยวิธีแคนนี

1. การปรับค่าสีเพื่อลดสัญญาณรบกวน (Noise reduction)

ค่าสีของแต่ละพิกเซลของรูปภาพจะถูกปรับโดยใช้ตัวกรองเกาส์เซียน ซึ่งสร้างหน้ากากเมทริกซ์ย่อย (mask) สำหรับการกรอง โดยใช้สมการเกาส์เซียน

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (2.2)$$

เมื่อ (x, y) เป็นตำแหน่งของสมาชิกในเมทริกซ์ย่อย และ σ เป็นค่าพารามิเตอร์

2. การหาขนาดและทิศทางของการเปลี่ยนแปลงค่าสี (gradient)

คำนวณหาขนาดของการเปลี่ยนแปลงค่าสี (gradient magnitude: M) และทิศทางของการเปลี่ยนแปลงค่าสี (gradient direction: α) โดยใช้สมการดังนี้

$$M(i, j) = \sqrt{g_x^2(i, j) + g_y^2(i, j)} \quad (2.3)$$

$$\alpha(i, j) = arctan\left(\frac{g_y(i, j)}{g_x(i, j)}\right) \quad (2.4)$$

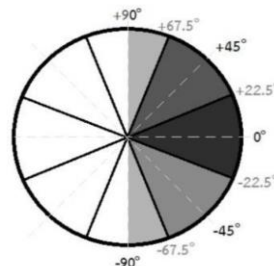
เมื่อ (j, i) คือ ตำแหน่งของพิกเซล g_x และ g_y คือ ขนาดของการเปลี่ยนแปลงค่าสีในแนว x และ y ตามลำดับ ซึ่งค่าของ g_x และ g_y สามารถดำเนินการหาได้หลายวิธี เช่น การใช้หน้ากากของตัวดำเนินการโรเบิร์ต การใช้หน้ากากของตัวดำเนินการพรีวิทท์ หรือการใช้หน้ากากของตัวดำเนินการโซเบล เป็นต้น

3. การพิจารณาขอบที่เป็นไปได้ (non-maxima suppression)

กำหนดพิกเซลที่เป็นไปได้ที่จะเป็นขอบ และกำจัดพิกเซลที่ไม่ใช่ขอบ ตามขั้นตอนต่อไปนี้

3.1 พิจารณาทิศทางของการเปลี่ยนแปลงค่าสีสำหรับแต่ละพิกเซล จากค่า arctanθ ซึ่งอยูในช่วง [-90 ,90] เป็น 4 ช่วงย่อย และทำการปรับค่า ดังนี้

- ถ้า -22.5° < α(i, j) ≤ 22.5°ปรับให้ α(i, j) = 0°
- ถ้า 22.5° < α(i, j) ≤ 67.5°ปรับให้ α(i, j) = +45°
- ถ้า -67.5° < α(i, j) ≤ -22.5°ปรับให้ α(i, j) = -45°
- ถ้า α(i, j) ≤ -67.5° หรือ α(i, j) > 67.5°ปรับให้ α(i, j) = 90°



รูปที่ 2.12 วิธีการแบ่งช่วงย่อยของทิศทางการเปลี่ยนแปลงค่าสี [26]

3.2 เปรียบเทียบขนาดของการเปลี่ยนแปลงค่าสีของพิกเซลที่พิจารณากับพิกเซลข้างเคียงจำนวน 2 พิกเซล ในทิศทางของการเปลี่ยนแปลง โดยกำหนดให้เป็นขอบที่เป็นไปได้ก็ต่อเมื่อ ขนาดของการเปลี่ยนแปลงค่าสีของพิกเซลนั้นมีค่ามากกว่าหรือเท่ากับขนาดของการเปลี่ยนแปลงค่าสีของพิกเซลข้างเคียงทั้ง 2 พิกเซล

4. การใช้ค่าเกณฑ์สองระดับ (double thresholding)

ระบุพิกเซลที่เป็นขอบ โดยกำหนดค่าเกณฑ์สูง (T_h) และค่าเกณฑ์ต่ำ (T_l) และพิจารณาเฉพาะพิกเซลที่เป็นไปได้ที่ได้จากขั้นตอนที่ 3 ดังนี้

- ถ้าขนาดของการเปลี่ยนแปลงค่าสีของพิกเซลมีค่ามากกว่าหรือเท่ากับ T_h จะเรียกพิกเซลนั้นว่า ขอบเข้ม(strong edge) และให้ขอบเข้มเป็นขอบของรูปภาพ

- ถ้าขนาดของการเปลี่ยนแปลงค่าสีของพิกเซลอยู่ระหว่าง T_l และ T_h จะเรียกพิกเซลนั้นว่าขอบไม่เข้ม (weak edge) และขอบไม่เข้มจะเป็นขอบ ถ้ามีพิกเซลข้างเคียงอย่างน้อย T_l พิกเซลเป็นขอบเข้ม

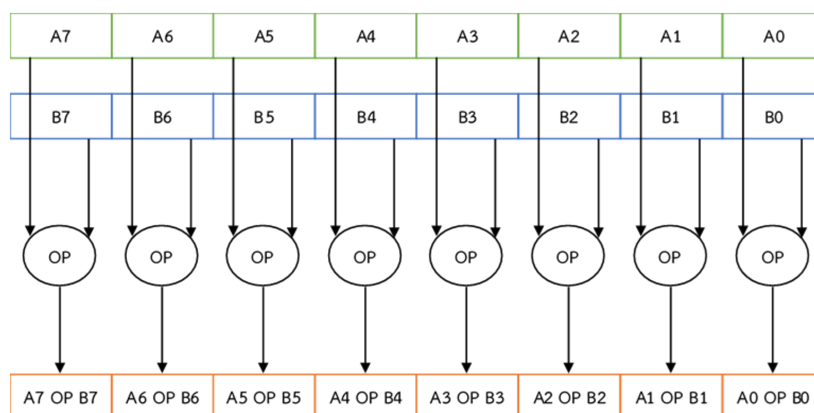
- ถ้าขนาดของการเปลี่ยนแปลงค่าสีของพิกเซลมีค่าน้อยกว่า T_l ให้พิกเซลนั้นไม่เป็นขอบ

2.4 การใช้ชุดคำสั่ง AVX

การใช้ชุดคำสั่ง AVX เป็นการประมวลผลในลักษณะของ SIMD (Single Instruction Multiple Data) ซึ่งเป็นการดำเนินการครั้งเดียวแล้วสามารถให้ผลได้หลายข้อมูลในเวลาเดียวกัน ชุดคำสั่ง AVX จะมีเรจิสเตอร์ใช้งานขนาด 256 บิต จำนวน 16 ตัว ได้แก่ YMM0-YMM15 ซึ่งเรจิสเตอร์แต่ละตัว เก็บค่าข้อมูลชนิดและขนาดต่าง ๆ กันได้ดังนี้

- เก็บตัวเลขจำนวนเต็มขนาด 8 บิต ได้ 32 ตัว
- เก็บตัวเลขจำนวนเต็มขนาด 16 บิต ได้ 16 ตัว
- เก็บตัวเลขจำนวนเต็มขนาด 32 บิต ได้ 8 ตัว
- เก็บตัวเลขจำนวนเต็มขนาด 64 บิต ได้ 4 ตัว
- เก็บตัวเลขทศนิยมขนาด 32 บิต ได้ 8 ตัว
- เก็บตัวเลขทศนิยมขนาด 64 บิต ได้ 4 ตัว

ชุดคำสั่ง AVX สามารถทำงานกับข้อมูลที่เก็บในรูปแบบเลขจำนวนเต็มและทศนิยม โดยสามารถใช้ชุดคำสั่งในการประมวลผลทางคณิตศาสตร์และตรรกะกับข้อมูลหลาย ๆ ข้อมูลโดยการดำเนินการคำสั่งเพียงคำสั่งเดียว ตัวอย่างเช่น



รูปที่ 2.13 ตัวอย่างการดำเนินการกับตัวเลข 32 บิตจำนวน 8 ค่าพร้อม ๆ กันโดยชุดคำสั่ง AVX

การเขียนโปรแกรมเพื่อใช้คำสั่ง AVX สามารถทำได้ 4 วิธี ได้แก่

1. การเขียนโปรแกรมภาษาแอสเซมบลีเพื่อใช้ชุดคำสั่ง AXV โดยตรง
2. การเขียนโปรแกรมภาษาซีและใช้วิธีแทรกโค้ดภาษาแอสเซมบลีในลักษณะ In line
3. การเขียนโปรแกรมภาษาระดับสูงทั่วไปแล้วใช้ความสามารถของคอมไพเลอร์ในการแปลงโค้ดระดับสูงเป็นคำสั่งเครื่อง AVX
4. การเขียนโปรแกรมโดยใช้ Compiler Intrinsic แต่วิธีการที่นิยมใช้ คือ การเขียนโปรแกรมภาษาระดับสูงโดยใช้ Compiler Intrinsics ในบทนี้จะยกตัวอย่างการใช้งานภาษาซีในการเรียกใช้ฟังก์ชัน Intrinsic โดย Intrinsic function ของ AVX จะมีรูปแบบ ดังนี้

__mm256 _mm_<op>_<data type> (__m256 a, __m256 b)
__mm256i _mm_<op>_<data type> (__m256i a, __m256i b)
__mm256d _mm_<op>_<data type> (__m256d a, __m256d b)

จะเห็นว่าชนิดของข้อมูลที่ฟังก์ชัน Intrinsic ของ AVX สนับสนุนจะมีสามชนิด คือ __mm_256, __mm256i, __mm256d สำหรับตัวเลขจำนวนเต็ม ตัวเลขทศนิยม 32 บิต และตัวเลขทศนิยม 64 บิต ตามลำดับ ตัวเลขทศนิยมทั้งสองขนาดจะอิงตามมาตรฐาน IEEE754 ฟังก์ชันตัวอย่างการใช้คำสั่ง AXV แสดงในรูปต่อไปนี้

```
void add_avx(float *a, float *b, float *c) {

    __m256 aa, bb, cc; // กำหนดตัวแปร ขนาด 256 บิต

    aa = _mm256_load_ps(a); // โหลดตัวเลขขนาด 32 บิตจำนวน 8 ตัว เก็บใน aa
    bb = _mm256_load_ps(b); // โหลดตัวเลขขนาด 32 บิตจำนวน 8 ตัว เก็บใน bb

     cc = _mm256_add_ps(aa, bb); //บวกค่า aa และ cc จำนวน 8 ตัวพร้อมกัน เก็บใน cc

     _mm256_store_ps(c, cc); // คืนค่า cc กลับไปยังพอยน์เตอร์ c

}
```

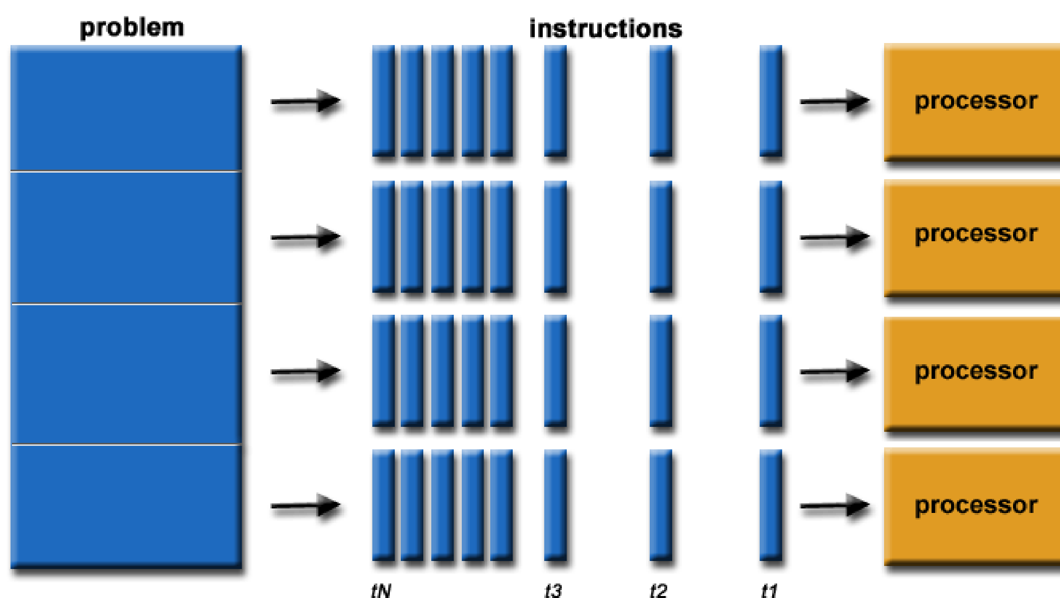
รูปที่ 2.14 ตัวอย่างการเขียนโปรแกรมบวกตัวเลข 32 บิตจำนวน 8 ค่าพร้อม ๆ กันโดยชุดคำสั่ง AVX

รูปที่ 2.14 แสดงตัวอย่างการประยุกต์ใช้ชุดคำสั่ง _mm256_add_ps สำหรับบวกเลขทศนิยม 8 ตัวพร้อม ๆ กัน โดยจะสังเกตเห็นว่าพารามิเตอร์อินพุตเป็นชนิดพอยน์เตอร์ของเลขทศนิยม เนื่องจาก ฟังก์ชัน _mm256_add_ps รับค่าข้อมูลชนิด __m256 ซึ่งเป็นข้อมูลเลขทศนิยม

ชนิด 32 บิต จำนวน 8 ค่า รวมจำนวนบิตทั้งหมด 256 บิต โดยชุดคำสั่งอื่น ๆ สามารถศึกษาเพิ่มเติมได้จาก [27]

2.5 การเขียนโปรแกรมบนสถาปัตยกรรมมัลติคอร์

การเขียนโปรแกรมบนสถาปัตยกรรมมัลติคอร์ (Multicore-programing) เป็นการเขียนโปรแกรมให้สามารถใช้ความสามารถของซีพียูในปัจจุบันซึ่งมีจำนวนคอร์ที่มากขึ้นได้อย่างเต็มที่ เพื่อให้การประมวลผลมีความเร็วเพิ่มขึ้นโดยการเขียนโปรแกรมให้เกิดการทำงานในแบบของการกระจายข้อมูลไปยังหน่วยประมวลผลต่าง ๆ ให้ช่วยกันประมวลผลในลักษณะการคำนวณแบบขนาน (Parallel processing) ซึ่งการประมวลผลแบบขนาน เป็นการแบ่งงานออกเป็นชิ้นเล็กให้แต่ละงานแก่ตัวประมวลผลหลาย ๆ ตัวในเวลาพร้อมกัน ดังรูปที่ 2.15 ประโยชน์ของการใช้วิธีการนี้ คือ แก้ปัญหาของงานขนาดใหญ่ได้ ในเวลาที่เร็วขึ้น



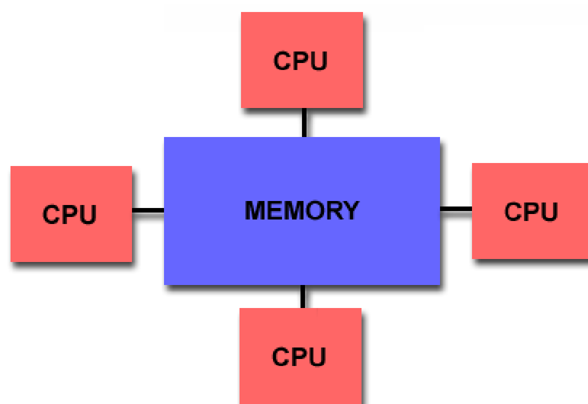
รูปที่ 2.15 การแบ่งงานออกเป็นชิ้นเล็กให้แก่ตัวประมวลผลหลาย ๆ ตัว ทำงานในเวลาพร้อมกัน[28]

เทคโนโลยีในการพัฒนาโปรแกรมแบบขนานมีหลากหลาย เพื่อตอบสนองต่อสถาปัตยกรรมของเครื่องประมวลผลแบบขนานหลายรูปแบบ โดยสามารถแบ่งออกเป็น 3 กลุ่มหลักคือ

1. การใช้คอมไพเลอร์พิเศษที่สามารถแปลงโปรแกรมแบบตามลำดับ (sequential program) ให้เป็นแบบขนาน (parallel program) ได้ เช่น KAP

2. การใช้ภาษาใหม่ที่ออกแบบมาเพื่อการเขียนโปรแกรมแบบขนานโดยเฉพาะ เช่น ชุดคำสั่งสามารถทำงานบนเวกเตอร์หรือเมทริกซ์ของข้อมูลได้ เช่น OCCAM หรือสร้างภาษาโดยขยายผลจากภาษาเก่าที่มีอยู่แล้ว เช่น Fortran 90 และ C

3. การใช้ไลบรารีเพิ่มเติมส่วนการจัดการโปรแกรมแบบขนานร่วมกับภาษาที่มีอยู่ เช่น C หรือ Java ตัวอย่างของไลบรารีที่ถือเป็นมาตรฐานในปัจจุบันคือ MPI (Message Passing Interface) ซึ่งใช้ในการพัฒนาโปรแกรมสำหรับสถาปัตยกรรมแบบหน่วยความจำกระจาย (Distributed Memory) และไลบรารี OpenMP ซึ่งใช้ในการพัฒนาโปรแกรมสำหรับสถาปัตยกรรมแบบหน่วยความจำร่วม (Shared Memory) โดยมีลักษณะดังรูปที่ 2.16



รูปที่ 2.16 ลักษณะของเครื่องแบบหลายหน่วยประมวลผลที่ใช้หน่วยความจำร่วมกัน [28]

2.5.1 การเขียนโปรแกรมแบบขนานแบบตัวแปรร่วมด้วย OpenMP

ไลบรารี OpenMP ใช้ประโยคไดเร็กทีฟ (directive) ในการต่อประสานโปรแกรมประยุกต์ เอื้อให้สามารถพัฒนาโปรแกรมได้สะดวกและง่ายกว่าไลบรารีแบบอื่นๆ และเป็นมาตรฐานที่ใช้ได้กับทั้งภาษา C และ Fortran โดยประโยคไดเร็กทีฟจะเป็นตัวชี้แนะให้คอมไพเลอร์ทราบว่าบรรทัดใดต้องประมวลผลแบบขนาน ประโยคไดเร็กทีฟมีลักษณะโครงสร้างดังนี้

#pragma omp <rest of pragma> <Clause>
ประโยคไดเร็กทีฟประกอบด้วย 3 ส่วน คือ

1. **#pragma omp** เป็นส่วนบอกให้คอมไพเลอร์ทราบว่าเป็นไดเร็กทีฟของ OpenMP
2. *<rest of pragma>* คือ pragmatic information ซึ่งเป็นชื่อเรียกการทำงานแบบขนานลักษณะต่างๆ
3. *<Clause>* เป็นส่วนกำหนดคุณลักษณะ (attribute) ของ pragma โดยแต่ละ pragma มีคลอสที่แตกต่างกัน หากไดเร็กทีฟใดไม่ระบุคลอส คุณลักษณะของ pragma นั้นจะมีค่าเป็นค่าเริ่มต้น (default)

โปรแกรมแบบขนานโดยทั่วไปสามารถแบ่งออกได้เป็น 2 ประเภท ตามลักษณะการมอบหมายงานให้เทรด (thread) หรือ โปรเซส (process) คือการแบ่งงานเชิงข้อมูล (Data/Domain Decomposition) และการแบ่งงานเชิงฟังก์ชัน (Function Decomposition)

ในการทำงานขนานเชิงข้อมูลนั้นเทรดแต่ละตัวทำงานแบบเดียวกัน แต่ใช้ข้อมูลที่นำมาประมวลผลแตกต่างกัน หรืออาจกล่าวได้ว่าแบ่งข้อมูลเป็นส่วนย่อยส่งให้แต่ละหน่วยประมวลผลนำไปคำนวณ โดย pragma ที่ใช้ในการทำงานขนานเชิงข้อมูลตัวอย่างเช่น **pragma parallel for** ดังแสดงการใช้งานดังรูปที่ 2.17

```
#pragma omp parallel for
for(int i = 1; i < 100; ++i)
{
	C[i] = A[i] + B[i]
}
```

รูปที่ 2.17 ตัวอย่างการเขียนโปรแกรมโดยใช้ OpenMP

บทที่ 3
แนวทางการวิจัย

การหาขอบภาพ (Edge Detection) โดยโค้ดมาตรฐานเป็นวิธีการที่ใช้เวลานานได้การคำนวณเพื่อประมวลผลในการหาขอบภาพ ผู้วิจัยจึงได้มีแนวคิดในการปรับปรุงโดยพยายามลดจำนวนครั้งในการคำนวณและการเข้าถึงข้อมูลลงเพื่อเพิ่มความเร็วในการหาขอบภาพ ในงานวิจัยนี้ผู้วิจัยจะแบ่งแนวทางการวิจัยออกเป็นสองส่วนหลักๆ ส่วนแรกคือการหาวิธีเพิ่มความเร็วในการหาขอบภาพของวิธีโซเบล (Sobel Edge Detection) และส่วนที่สองจะเป็นการนำวิธีการในการลดเพิ่มความเร็วของวิธีโซเบลในส่วนแรกมาประยุกต์ใช้เพื่อพัฒนาความเร็วในการหาขอบภาพของแคนนี (Canny Edge Detection) ซึ่งแนวทางในการเพิ่มความเร็วสำหรับการตรวจจับหาขอบภาพจะนำเสนอหัวข้อดังนี้

3.1 การเพิ่มประสิทธิภาพในการคอนโวลูชันสำหรับการหาขอบภาพด้วยวิธีโซเบล
3.2 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการประมวลผลข้อมูลพร้อมกันหลายบรรทัด
3.3 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการใช้ชุดคำสั่ง AVX
3.4 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการใช้ OpenMP
3.5 การประยุกต์ใช้การเพิ่มประสิทธิภาพในการหาขอบภาพสำหรับแคนนี

3.1 การเพิ่มประสิทธิภาพในการคอนโวลูชันสำหรับการหาขอบภาพด้วยวิธีโซเบล

การหาขอบภาพโดยวิธีโซเบลจะใช้วิธีการคอนโวลูชัน (Convolution) ระหว่างรูปภาพกับเคอร์เนลเพื่อนำไปสู่กระบวนการให้ได้ขอบของภาพ ดังนั้นผู้วิจัยจึงได้นำเสนอวิธีการเพิ่มประสิทธิภาพในการคอนโวลูชันภาพเพื่อหาขอบภาพเป็นขั้นตอนแรก โดยการตรวจสอบค่าของเคอร์เนลที่จะคำนวณ หากเคอร์เนลนั้นมีค่าเป็น 0 หรือค่า 1 จะไม่นำค่าในเคอร์เนลตำแหน่งนั้นมาคำนวณเพื่อลดจำนวนครั้งในการดำเนินการของภาพลง ดังแสดงวิธีการในรูปที่ 3.1-3.3

Gradient: G = Kernel * Image

Kernel

1	2	3
4	5	6
7	8	9

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8

G = (1xP0) + (2xP1) + (3xP2) + (4xP3) + (5xP4) + (6xP5) + (7xP6) + (8xP7) + (9xP8)

รูปที่ 3.1 วิธีการทั่วไปในการคอนโวลูชันเพื่อหา Gradient magnitude

Gradient X

Gx = Kernel X * Image

Kernel X

1	0	-1
2	0	-2
1	0	-1

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8

$$Gx = (P0 - P2) + 2(P3 - P5) + (P6 - P8)$$

รูปที่ 3.2 วิธีการที่นำเสนอในการคอนโวลูชันเพื่อหา Gradient ในแนวนอน

Gradient Y

Gy = Kernel Y * Image

Kernel Y

1	2	1
0	0	0
-1	-2	-1

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8

$$Gy = (P0 - P6) + 2(P1 - P7) + (P2 - P8)$$

รูปที่ 3.3 วิธีการที่นำเสนอในการคอนโวลูชันเพื่อหา Gradient ในแนวตั้ง

จากรูปที่ 3.1 จะเห็นว่าวิธีการคอนโวลูชันแบบทั่วไปใช้จำนวนครั้งในการคำนวณ 17 ครั้ง คือ การดำเนินการในการคูณจำนวน 9 ครั้ง และการดำเนินการในการบวกจำนวน 8 ครั้ง สำหรับในรูปที่ 3.2 และ 3.3 การคอนโวลูชันเพื่อหาขอบภาพโดยวิธีที่นำเสนอจะไม่นำค่าในพิกเซลที่คูณกับเคอร์เนลที่เป็น 0 และ 1 มาคำนวณ ซึ่งจะทำให้เหลือจำนวนครั้งในการคำนวณเพียง 6 ครั้ง คือ การดำเนินการในการลบจำนวน 3 ครั้ง การดำเนินการในการบวกจำนวน 2 ครั้ง และการดำเนินการในการคูณจำนวน 1 ครั้ง ดังนั้น จากการคำนวณจำนวนครั้งในการดำเนินการที่ลดลง สามารถทำให้จำนวนครั้งในการเข้าถึงข้อมูลลดลงได้ด้วยเช่นกัน และจะส่งผลให้สามารถลดระยะเวลาและสามารถเพิ่มประสิทธิภาพในการประมวลผลเพื่อหาขอบของภาพได้

3.2 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการประมวลผลข้อมูลพร้อมกันหลายบรรทัด

โดยทั่วไปการใช้งานการตรวจจับขอบภาพโดยวิธีของ Sobel ส่วนใหญ่จะคำนวณหาขนาดของ Gradient magnitude ทีละบรรทัด กระบวนการเริ่มต้นจากพิกเซลซ้ายสุด และพิกเซลที่อยู่ติดกันในบรรทัดเดียวกันจะคำนวณทีละพิกเซลจนกว่าจะได้ขนาดของ Gradient magnitude ในแต่ละบรรทัด กระบวนการนี้ใช้กับรูปภาพทีละบรรทัดจนกว่ารูปภาพจะได้รับการประมวลผลอย่างสมบูรณ์ ผู้วิจัยเรียกวิธีการนี้ว่า การคำนวณแบบ 1 บรรทัด (L==1) ดังแสดงตัวอย่างในรูปที่ 3.4 ต่อมาผู้วิจัยจึงได้นำเสนอวิธีการคำนวณการหาขนาดของ Gradient magnitude โดยการประมวลผลข้อมูลพร้อมกันหลายบรรทัดในแต่ละครั้งของการคำนวณ ซึ่งจะช่วยให้สามารถใช้ผลลัพธ์ของการคำนวณทางคณิตศาสตร์กับไลน์อื่น ๆ ร่วมกันได้ ซึ่งวิธีการนี้จะสามารถลดจำนวนครั้งในการคำนวณได้ ดังแสดงตัวอย่างการใช้ผลลัพธ์ร่วมกันในรูปที่ 3.5 และ 3.6 โดยกำหนดให้ L คือ จำนวนบรรทัดของข้อมูลภาพที่ต้องการประมวลผลในแต่ละรอบ

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8

*

Kernel X

1	0	-1
2	0	-2
1	0	-1

=

Gradients X

Gx = (P0 - P2) + 2(P3-P5) + (P6 – P8)

รูปที่ 3.4 การคำนวณหาขนาดของ Gradient magnitude โดยทั่วไป หรือแบบ L==1

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8
P9	P10	P11
P12	P13	P14
P15	P16	P17

*

Kernel X

1	0	-1
2	0	-2
1	0	-1

=

Gradients X

Gx1 = (P0 - P2) + 2(P3-P5) + (P6 – P8)
Gx2 = (P3 – P5) + 2(P6 – P8) + (P9 – P11)
Gx3 = (P6 – P8) + 2(P9 – P11) + (P12-P14)
Gx4 = (P9 – P11) + 2(P12-P14) + (P15-P17)

รูปที่ 3.5 วิธีการคำนวณ Gradient ในแนวนอนที่นำเสนอ สำหรับ L==4

รูปที่ 3.5 ผลลัพธ์ของการคำนวณค่า P3-P5 จำนวน 1 ครั้ง สามารถนำคำตอบที่ได้ไปใช้ในการคำนวณเพื่อหาค่า Gradient ในแนวนอนได้ 2 ครั้ง ประกอบด้วย Gx1 และ Gx2 และสำหรับการคำนวณ P6-P8 สามารถนำผลลัพธ์ที่ได้จากการคำนวณครั้งเดียวไปใช้ได้ 3 ครั้ง คือใช้ในการคำนวณหา Gx1, Gx2 และ Gx3 และสำหรับการคำนวณ P9-P11 สามารถนำผลลัพธ์ที่ได้จากากรคำนวณครั้งเดียวไปใช้ได้ 3 ครั้งเช่นกัน คือใช้ในการคำนวณหา Gx2, Gx3 และ Gx4 จะเห็นได้ว่าการเพิ่มจำนวนไลน์ในแต่ละลูปของการคำนวณจะสามารถช่วยลดจำนวนครั้งในการคำนวณและการเข้าถึงข้อมูลได้ ซึ่งจะส่งผลให้การประมวลผลเพื่อหาขอบภาพมีความเร็วเพิ่มขึ้น

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8
P9	P10	P11
P12	P13	P14
P15	P16	P17

*

Kernel Y

1	2	1
0	0	0
-1	-2	-1

=

Gradients Y

Gy1 = (P0 + P2) – (P6 + P8) + 2(P1 – P7)
Gy2 = (P3 + P5) – (P9 + P11) + 2(P4 – P10)
Gy3 = (P6 + P8) – (P12 + P14) + 2(P7 – P13)
Gy4 = (P9 + P11) – (P15 + P17) + 2(P10 – P16)

รูปที่ 3.6 วิธีการคำนวณ Gradient ในแนวตั้งที่นำเสนอ สำหรับ L==4

รูปที่ 3.6 ผลบวกของการคำนวณค่า P6+P8 จำนวน 1 ครั้ง สามารถนำคำตอบที่ได้ไปใช้ในการคำนวณเพื่อหาค่า Gradient ในแนวตั้งได้ 2 ครั้ง ประกอบด้วย Gy1 และ Gy3 และสำหรับการคำนวณ P9+P11 สามารถนำผลลัพธ์ที่ได้จากการคำนวณครั้งเดียวไปใช้ได้ 2 ครั้งเช่นกัน คือใช้ในการ

คำนวณหา Gy2 และ Gy4 ดังนั้นจะเห็นได้ว่าการเพิ่มจำนวนไลน์ในแต่ละลูปของการคำนวณจะสามารถช่วยลดจำนวนครั้งในการคำนวณและการเข้าถึงข้อมูลได้ ซึ่งจะส่งผลให้การประมวลผลเพื่อหาขอบภาพมีความเร็วเพิ่มขึ้น

3.3 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการใช้ชุดคำสั่ง AVX

การเขียนโปรแกรมเพื่อเพิ่มความเร็วในการหาขอบภาพ ผู้วิจัยได้นำเสนอวิธีการเขียนโปรแกรมเพื่อเพิ่มความเร็วในการหาขอบภาพโดยใช้ชุดคำสั่ง AVX โดยกำหนดให้ *L* คือ จำนวนบรรทัดของข้อมูลภาพที่ต้องการประมวลผลในแต่ละรอบ ซึ่งจะแสดงตัวอย่างการเขียนโปรแกรมโดยวิธีที่นำเสนอ สำหรับ *L*==2, *L*==3 และ *L*==4 เพื่อให้เห็นตัวอย่างการเขียนโปรแกรมด้วยวิธีที่นำเสนอ ที่จะทำให้สามารถลดจำนวนครั้งในการคำนวณและการเข้าถึงข้อมูลในหน่วยความจำของคอมพิวเตอร์ ทำให้สามารถเพิ่มประสิทธิภาพที่ใช้ในการประมวลผลภาพได้

3.3.1 การเขียนโปรแกรมโดยวิธีที่นำเสนอ สำหรับ *L*==2

ในหัวข้อนี้จะอธิบายวิธีที่ใช้ในการเขียนโปรแกรมด้วยชุดคำสั่ง AVX โดยวิธีการที่นำเสนอเริ่มต้นจากการเขียนโปรแกรมภาษาซีด้วยวิธีที่นำเสนอในรูปที่ 3.7 และปรับปรุงโปรแกรมให้เป็นการเขียนด้วยชุดคำสั่ง AVX โดยใช้วิธีที่นำเสนอในรูปที่ 3.8

```
1   char *pS8 = source.ptr<uchar>(0, 0);
2   char *pD8 = dst.ptr<uchar>(0, 0);
3
4   typedef short s;
5
6   short D0x,D1x,D2x,D3x, S0y,S1y,S2y,S3y;
7   short P0,P1,P2,P3,P4,P5,P6,P7,F8,F9,P10,P11;
8
9   int source_image_width = source.cols;
10  int dst_image_width = dst.cols;
11
12  int step2 = source_image_width;
13  int step3 = 2*source_image_width;
14  int step4 = 3*source_image_width;
15
16  P0 = (s)*pS8;             P1 = (s)*(pS8+1);          P2 = (s)*(pS8+2);
17  P3 = (s)*(pS8+step2);     P4 = (s)*(pS8+step2+1);    P5 = (s)*(pS8+step2+2)
18  P6 = (s)*(pS8+step3);     P7 = (s)*(pS8+step3+1);    P8 = (s)*(pS8+step3+2)
19  P9 = (s)*(pS8+step4);     P10 = (s)*(pS8+step4+1);   P11 = (s)*(pS8+step4+2)
20
21  D0x = P2-P0;              S0y = P2+P0;
22  D1x = P5-P3;              S1y = P5+P3;
23  D2x = P8-P6;              S2y = P8+P6;
24  D3x = P11-P9;             S3y = P11+P9;
25
26  short G0x = D0x + (D1x<<1) +D2x;
27  short G1x = D1x + (D2x<<1) +D3x;
28  short G0y = S0y - S2y + (P1-P7)<<1;
29  short G1y = S1y - S3y + (P4-P10)<<1;
30
31  short R0 = abs(G0x) + abs(G0y);
32  short R1 = abs(G1x) t abs(G1y);
33  *(pD8) = R0;
34  *(pD8 + dst_image_width) = R1;
```

รูปที่ 3.7 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ภาษาซี สำหรับ *L*==2

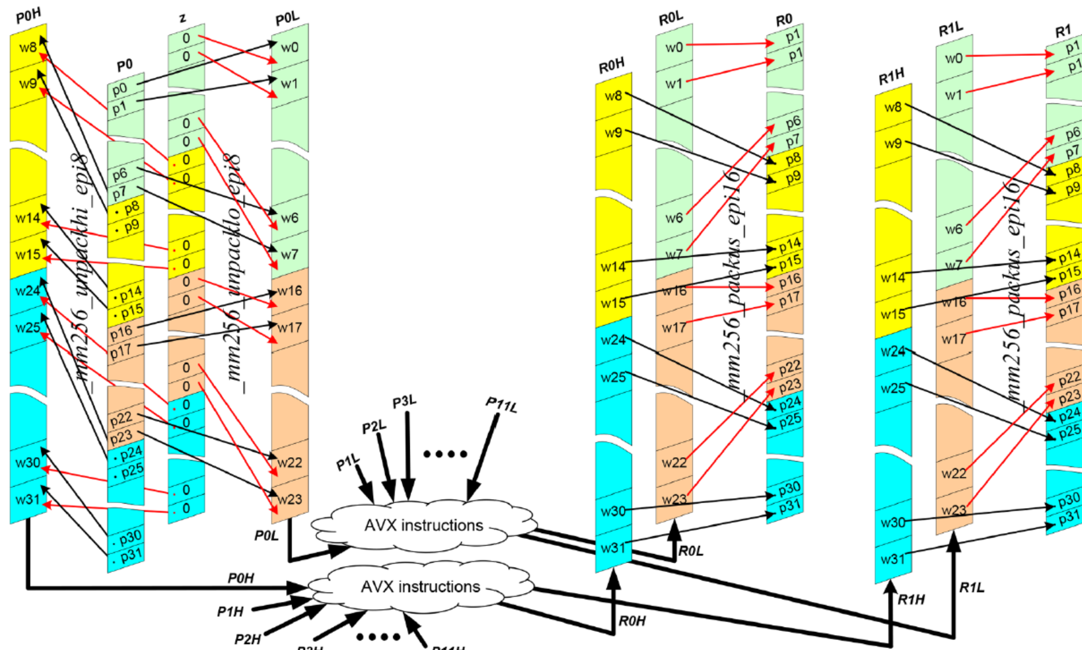
การใช้งานภาษาซีในการเขียนโปรแกรมสำหรับวิธีการที่นำเสนอในการหาขอบภาพแบบ L==2 โดยโปรแกรมในรูปที่ 3.7 บรรทัดที่ 16-19 เป็นการโหลดข้อมูลจากภาพจำนวน 12 พิกเซล และบรรทัดที่ 21-24 เป็นการคำนวณค่าไว้ล่วงหน้าเพื่อเก็บผลลัพธ์ที่สามารถนำมาใช้คำนวณร่วมกันได้ ซึ่งจะนำมาคำนวณหา Gradient magnitudes ในบรรทัดที่ 26-29 จากนั้นจึงนำผลลัพธ์ของ Gradient magnitudes ในแนวแกนนอนและแกนตั้งที่ได้ไปคำนวณหาขอบของภาพในบรรทัดที่ 31-32 จะเห็นได้ว่าวิธีการเขียนโปรแกรมแบบที่นำเสนอสามารถลดจำนวนครั้งของการคำนวณได้โดยการแบ่งปันผลลัพธ์ของการคำนวณร่วมกัน และในหนึ่งรอบของการประมวลผลแบบ L==2 จะได้ผลลัพธ์จำนวน 2 บรรทัดคือ R0 และ R1

```
1   char *pS8 = source.ptr<uchar>(0, 0);
2   char *pD8 = dst.ptr<uchar>(0, 0);
3
4   __m256i P0, P1, ..., P10, P11;
5   __m256i P0L, P1L, ..., P10L, P11L;
6   __m256i P0H, P1H, ..., P10H, P11H;
7   __m256i z = _mm256_setzero_si256();
8   int source_image_width = source.cols;
9   int dst_image_width = dst.cols;
10
11  int step2 = source_image_width;
12  int step3 = 2*source_image_width;
13  int step4 = 3*source_image_width;
14
15  P0 = _mm256_loadu_si256((__m256i*)pS8);
16  P1 = _mm256_loadu_si256((__m256i*)(pS8 + 1));
17  P2 = _mm256_loadu_si256((__m256i*)(pS8 + 2));
18  P3 = _mm256_loadu_si256((__m256i*)(pS8 + step2));
19  P4 = _mm256_loadu_si256((__m256i*)(pS8 + step2 + 1));
20  P5 = _mm256_loadu_si256((__m256i*)(pS8 + step2 + 2));
21  P6 = _mm256_loadu_si256((__m256i*)(pS8 + step3));
22  P7 = _mm256_loadu_si256((__m256i*)(pS8 + step3 + 1));
23  P8 = _mm256_loadu_si256((__m256i*)(pS8 + step3 + 2));
24  P9 = _mm256_loadu_si256((__m256i*)(pS8 + step4));
25  P10 = _mm256_loadu_si256((__m256i*)(pS8 + step4 + 1));
26  P11 = _mm256_loadu_si256((__m256i*)(pS8 + step4 + 2));
27
28  P0L = _mm256_unpacklo_epi8(P0, z);          P0H = _mm256_unpackhi_epi8(P0, z);
29  P1L = _mm256_unpacklo_epi8(P1, z);          P1H = _mm256_unpackhi_epi8(P1, z);
30  ...                                         ...
31  P10L = _mm256_unpacklo_epi8(P10, z);        P10H = _mm256_unpackhi_epi8(P10, z);
32  P11L = _mm256_unpacklo_epi8(P11, z);        P11H = _mm256_unpackhi_epi8(P11, z);
33
34  D0L = _mm256_sub_epi16(P0L, P2L);           D0H = _mm256_sub_epi16(P0H, P2H);
35  D1L = _mm256_sub_epi16(P3L, P5L);           D1H = _mm256_sub_epi16(P3H, P5H);
36  D2L = _mm256_sub_epi16(P6L, P8L);           D2H = _mm256_sub_epi16(P6H, P8H);
37  D3L = _mm256_sub_epi16(P9L, P11L);          D3H = _mm256_sub_epi16(P9H, P11H);
38  S0L = _mm256_add_epi16(P0L, P2L);           S0H = _mm256_add_epi16(P0H, P2H);
39  S1L = _mm256_add_epi16(P6L, P8L);           S1H = _mm256_add_epi16(P6H, P8H);
40  S2L = _mm256_add_epi16(P3L, P5L);           S2H = _mm256_add_epi16(P3H, P5H);
41  S3L = _mm256_add_epi16(P9L, P11L);          S3H = _mm256_add_epi16(P9H, P11H);
42
43  G0xL = _mm256_add_epi16(_mm256_add_epi16(D0L, _mm256_slli_epi16(D1L, 1)), D2L);
44  G1xL = _mm256_add_epi16(_mm256_add_epi16(D1L, _mm256_slli_epi16(D2L, 1)), D3L);
45  D0L = _mm256_sub_epi16(S0L, S1L);
46  D1L = _mm256_sub_epi16(S2L, S3L);
47  G0yL = _mm256_add_epi16(D0L, _mm256_slli_epi16(_mm256_sub_epi16(P1L, P7L), 1));
48  G1yL = _mm256_add_epi16(D1L, _mm256_slli_epi16(_mm256_sub_epi16(P4L, P10L), 1));
49
50  G0xH = _mm256_add_epi16(_mm256_add_epi16(D0H, _mm256_slli_epi16(D1H, 1)), D2H);
51  G1xH = _mm256_add_epi16(_mm256_add_epi16(D1H, _mm256_slli_epi16(D2H, 1)), D3H);
52  D0H = _mm256_sub_epi16(S0H, S1H);
53  D1H = _mm256_sub_epi16(S2H, S3H);
54  G0yH = _mm256_add_epi16(D0H, _mm256_slli_epi16(_mm256_sub_epi16(P1H, P7H), 1));
55  G1yH = _mm256_add_epi16(D1H, _mm256_slli_epi16(_mm256_sub_epi16(P4H, P10H), 1));
56
57  R0L = _mm256_add_epi16(_mm256_abs_epi16(G0xL), _mm256_abs_epi16(G0yL));
58  R1L = _mm256_add_epi16(_mm256_abs_epi16(G1xL), _mm256_abs_epi16(G1yL));
59  R0H = _mm256_add_epi16(_mm256_abs_epi16(G0xH), _mm256_abs_epi16(G0yH));
60  R1H = _mm256_add_epi16(_mm256_abs_epi16(G1xH), _mm256_abs_epi16(G1yH));
61
62  R0 = _mm256_packus_epi16(R0L, R0H);
63  R1 = _mm256_packus_epi16(R1L, R1H);
64  _mm256_store_si256((__m256i*)pD8, R0);
65  _mm256_store_si256((__m256i*)(pD8 + dst_image_width), R1);
```

รูปที่ 3.8 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ชุดคำสั่ง AVX สำหรับ *L*==2

การใช้งานชุดคำสั่ง AVX ในการเขียนโปรแกรมสำหรับวิธีการที่นำเสนอในการหาขอบภาพแบบ *L*==2 โดยโปรแกรมในรูปที่ 3.8 บรรทัดที่ 15-26 เป็นการโหลดข้อมูลจากภาพโดยใช้ฟังก์ชัน _mm256_loadu_si256 ในแต่ละครั้งจะโหลดข้อมูลได้จำนวน 256 บิต ซึ่งจะได้ข้อมูลภาพขนาด

8 บิต จำนวน 32 พิกเซล ต่อมาในบรรทัดที่ 28-32 โปรแกรมได้ขยายข้อมูล 8 บิต ให้เป็นข้อมูล 16 บิต เพื่อนำไปคำนวณในรูปแบบ 16 บิตในการเขียนโปรแกรมด้วยชุดคำสั่ง AVX โดยใช้ฟังก์ชัน _mm256_unpacklo_epi8 สำหรับ P0L-P11L และใช้ฟังก์ชัน _mm256_unpackhi_epi8 สำหรับ P0H-P11H ดังแผนภาพในรูปที่ 3.9 จากนั้นในบรรทัดที่ 34-41 เป็นการคำนวณค่าไว้ล่วงหน้าเพื่อเก็บผลลัพธ์ที่สามารถนำมาใช้คำนวณร่วมกันได้ ซึ่งจะนำมาคำนวณหา Gradient magnitudes ในบรรทัดที่ 43-55 จากนั้นจึงนำผลลัพธ์ของ Gradient magnitudes ในแนวแกนนอนและแกนตั้งที่ได้ไปคำนวณหาขอบของภาพในบรรทัดที่ 57-60 และเมื่อได้ผลลัพธ์ของการคำนวณเรียบร้อยในบรรทัดที่ 62-63 โปรแกรมจะลดขนาดข้อมูลจาก 16 บิต ให้กลับไปอยู่ในลักษณะพิกเซลของภาพ 8 บิต ด้วยฟังก์ชัน _mm256_packus_epi16 ดังแผนภาพในรูปที่ 3.9 และบรรทัดที่ 64-65 เป็นการคืนค่าข้อมูลไปยังภาพผลลัพธ์ที่ต้องการด้วยฟังก์ชัน _mm256_store_si256



รูปที่ 3.9 การแบ่งภาพขนาด 32 พิกเซล จากข้อมูล 8 บิต ให้เป็น 16 บิต
เพื่อทำการประมวลผลโดยใช้ชุดคำสั่ง AVX

จากรูปที่ 3.9 แสดงการแบ่งข้อมูลของภาพขนาด 32 พิกเซล จากข้อมูล 8 บิต ให้เป็น 16 บิต โดนการใช้ฟังก์ชัน *_mm256_unpacklo_epi8* และ *_mm256_unpackhi_epi8* เพื่อนำข้อมูลภาพ 8 บิตมารวมกับค่า 0 อีก 8 บิต เพื่อให้ได้ข้อมูล 16 บิต ไปใช้ในการคำนวณโดยใช้ชุดคำสั่ง AVX และหลังจากเสร็จขั้นตอนการคำนวณของการดำเนินการทางคณิตศาสตร์แล้วจะทำให้รวมข้อมูลผลลัพธ์กลับให้อยู่ในลักษณะ 8 บิตเหมือนเดิมโดยใช้ฟังชัน *_mm256_packus_epi16* เนื่องจากคุณลักษณะของชุดคำสั่ง AVX ถูกออกแบบมาใช้สำหรับงานเข้ารหัสข้อมูลเป็นส่วนๆ เมื่อนำมาปรับใช้กับการประมวลผลภาพ จึงจำเป็นต้องใช้คำสั่ง unpack และ pack ข้อมูล เพื่อให้เกิดผลลัพธ์ของการจัดเรียงข้อมูลปลายทางที่ถูกต้อง อย่างไรก็ดีคำสั่งในการ unpack และ pack มี latency ในการประมวลผลต่ำ จึงไม่ได้มีผลต่อความเร็วในการประมวลผลอย่างมีนัยสำคัญ

3.3.2 การเขียนโปรแกรมโดยวิธีที่นำเสนอ สำหรับ *L*==3

ในหัวข้อนี้จะอธิบายวิธีที่ใช้ในการเขียนโปรแกรมด้วยชุดคำสั่ง AVX โดยวิธีการที่นำเสนอเริ่มต้นจากการเขียนโปรแกรมภาษาซีด้วยวิธีที่นำเสนอในรูปที่ 3.10 และปรับปรุงโปรแกรมให้เป็นการเขียนด้วยชุดคำสั่ง AVX โดยใช้วิธีที่นำเสนอในรูปที่ 3.11

```
1   char *pS8 = source.ptr<uchar>(0, 0);
2   char *pD8 = dst.ptr<uchar>(0, 0);
3
4   typedef short s;
5
6   short D0x, D1x, D2x, D3x, D4x, S0y,S1y,S2y,S3y, S4y;
7   short P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11, P12, P13, P14;
8
9   int source_image_width = source.cols;
10  int dst_image_width = dst.cols;
11
12  int step2 = source_image_width;
13  int step3 = 2*source_image_width;
14  int step4 = 3*source_image_width;
15  int step5 = 4*source_image_width;
16
17  P0 = (s)*pS8;             P1 = (s)*(pS8+1);            P2 = (s)*(pS8+2);
18  P3 = (s)*(pS8+step2);     P4 = (s)*(pS8+step2+1);      P5 = (s)*(pS8+step2+2)
19  P6 = (s)*(pS8+step3);     P7 = (s)*(pS8+step3+1);      P8 = (s)*(pS8+step3+2)
20  P9 = (s)*(pS8+step4);     P10 = (s)*(pS8+step4+1);     P11 = (s)*(pS8+step4+2)
21  P12 = (s)*(pS8+step5);    P13 = (s)*(pS8+step5+1);     P14 = (s)*(pS8+step5+2)
22
23  D0x = P0-P2;              S0y = P0+P2;
24  D1x = P3-P5;              S1y = P3+P5;
25  D2x = P6-P8;              S2y = P6+P8;
26  D3x = P9-P11;             S3y = P9+P11;
27  D4x = P12-P14;            S4y = P12+P14;
28
29  short G0x = D0x + (D1x<<1) +D2x;
30  short G1x = D1x + (D2x<<1) +D3x;
31  short G2x = D2x + (D3x<<1) +D4x;
32  short G0y = S0y - S2y + (P1-P7)<<1;
33  short G1y = S1y - S3y + (P4-P10)<<1;
34  short G2y = S2y - S4y + (P7-P13)<<1;
35
36  short R0 = abs(G0x) + abs(G0y);
37  short R1 = abs(G1x) t abs(G1y);
38  short R3 = abs(G2x) t abs(G2y);
39  *(pD8) = R0;
40  *(pD8 + dst_image_width) = R1;
41  *(pD8 + (2 * dst_image_width)) = R2;
```

รูปที่ 3.10 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ภาษาซี สำหรับ *L*==3

การใช้งานภาษาซีในการเขียนโปรแกรมสำหรับวิธีการที่นำเสนอในการหาขอบภาพแบบ *L*==3 โดยโปรแกรมในรูปที่ 3.10 บรรทัดที่ 17-21 เป็นการโหลดข้อมูลจากภาพจำนวน 15 พิกเซล และบรรทัดที่ 23-27 เป็นการคำนวณค่าไว้ล่วงหน้าเพื่อเก็บผลลัพธ์ที่สามารถนำมาใช้คำนวณร่วมกันได้ ซึ่งจะนำมาคำนวณหา Gradient magnitudes ในบรรทัดที่ 29-34 จากนั้นจึงนำผลลัพธ์ของ Gradient magnitudes ในแนวแกนนอนและแกนตั้งที่ได้ไปคำนวณหาขอบของภาพในบรรทัดที่ 36-38 จะเห็นได้ว่าวิธีการเขียนโปรแกรมแบบที่นำเสนอสามารถลดจำนวนครั้งของการคำนวณได้โดยการแชร์ผลลัพธ์ของการคำนวณร่วมกัน และในหนึ่งรอบของการประมวลผลแบบ *L*==3 จะได้ผลลัพธ์จำนวน 3 บรรทัดคือ R0 , R1 และ R3

```
1   char *pS8 = source.ptr<uchar>(0, 0);
2   char *pD8 = dst.ptr<uchar>(0, 0);
3
4   __m256i P0, P1, ..., P12, P13, P14;
5   __m256i P0L, P1L, ..., P12L, P13L, P14L;
6   __m256i P0H, P1H, ..., P12H, P13H, P14H;
7   __m256i z = _mm256_setzero_si256();
8   int source_image_width = source.cols;
9   int dst_image_width = dst.cols;
10
11  int step2 = source_image_width;
12  int step3 = 2*source_image_width;
13  int step4 = 3*source_image_width;
14  int step5 = 4*source_image_width;
15
16  P0 = _mm256_loadu_si256((__m256i*)pS8);
17  P1 = _mm256_loadu_si256((__m256i*)(pS8 + 1));
18  ...
19  P12 = _mm256_loadu_si256((__m256i*)(pS8 + step5));
20  P13 = _mm256_loadu_si256((__m256i*)(pS8 + step5 + 1));
21  P14 = _mm256_loadu_si256((__m256i*)(pS8 + step5 + 2));
22
23  P0L = _mm256_unpacklo_epi8(P0, z);          P0H = _mm256_unpackhi_epi8(P0, z);
24  P1L = _mm256_unpacklo_epi8(P1, z);          P1H = _mm256_unpackhi_epi8(P1, z);
25  ...                                         ...
26  P13L = _mm256_unpacklo_epi8(P13, z);        P13H = _mm256_unpackhi_epi8(P13, z);
27  P14L = _mm256_unpacklo_epi8(P14, z);        P14H = _mm256_unpackhi_epi8(P14, z);
28
29  D0L = _mm256_sub_epi16(P0L, P2L);           D0H = _mm256_sub_epi16(P0H, P2H);
30  D1L = _mm256_sub_epi16(P3L, P5L);           D1H = _mm256_sub_epi16(P3H, P5H);
31  D2L = _mm256_sub_epi16(P6L, P8L);           D2H = _mm256_sub_epi16(P6H, P8H);
32  D3L = _mm256_sub_epi16(P9L, P11L);          D3H = _mm256_sub_epi16(P9H, P11H);
33  D4L = _mm256_sub_epi16(P12L, P14L);         D4H = _mm256_sub_epi16(P12H, P14H);
34
35  S0L = _mm256_add_epi16(P0L, P2L);           S0H = _mm256_add_epi16(P0H, P2H);
36  S1L = _mm256_add_epi16(P6L, P8L);           S1H = _mm256_add_epi16(P6H, P8H);
37  S2L = _mm256_add_epi16(P3L, P5L);           S2H = _mm256_add_epi16(P3H, P5H);
38  S3L = _mm256_add_epi16(P9L, P11L);          S3H = _mm256_add_epi16(P9H, P11H);
39  S4L = _mm256_add_epi16(P12L, P14L);         S4H = _mm256_add_epi16(P12H, P14H);
40
41  G0xL = _mm256_add_epi16(_mm256_add_epi16(D0L, _mm256_slli_epi16(D1L, 1)), D2L);
42  G1xL = _mm256_add_epi16(_mm256_add_epi16(D1L, _mm256_slli_epi16(D2L, 1)), D3L);
43  G2xL = _mm256_add_epi16(_mm256_add_epi16(D2L, _mm256_slli_epi16(D3L, 1)), D4L);
44  D0L = _mm256_sub_epi16(S0L, S1L);
45  D1L = _mm256_sub_epi16(S2L, S3L);
46  D2L = _mm256_sub_epi16(S1L, S4L);
47  G0yL = _mm256_add_epi16(D0L, _mm256_slli_epi16(_mm256_sub_epi16(P1L, P7L), 1));
48  G1yL = _mm256_add_epi16(D1L, _mm256_slli_epi16(_mm256_sub_epi16(P4L, P10L), 1));
49  G2yL = _mm256_add_epi16(D2L, _mm256_slli_epi16(_mm256_sub_epi16(P7L, P13L), 1));
50
51  G0xH = _mm256_add_epi16(_mm256_add_epi16(D0H, _mm256_slli_epi16(D1H, 1)), D2H);
52  G1xH = _mm256_add_epi16(_mm256_add_epi16(D1H, _mm256_slli_epi16(D2H, 1)), D3H);
53  G2xH = _mm256_add_epi16(_mm256_add_epi16(D2H, _mm256_slli_epi16(D3H, 1)), D4H);
54  D0H = _mm256_sub_epi16(S0H, S1H);
55  D1H = _mm256_sub_epi16(S2H, S3H);
56  D2H = _mm256_sub_epi16(S1H, S4H);
57  G0yH = _mm256_add_epi16(D0H, _mm256_slli_epi16(_mm256_sub_epi16(P1H, P7H), 1));
58  G1yH = _mm256_add_epi16(D1H, _mm256_slli_epi16(_mm256_sub_epi16(P4H, P10H), 1));
59  G2yH = _mm256_add_epi16(D2H, _mm256_slli_epi16(_mm256_sub_epi16(P7H, P13H), 1));
60
61  R0L = _mm256_add_epi16(_mm256_abs_epi16(G0xL), _mm256_abs_epi16(G0yL));
62  R1L = _mm256_add_epi16(_mm256_abs_epi16(G1xL), _mm256_abs_epi16(G1yL));
63  R2L = _mm256_add_epi16(_mm256_abs_epi16(G2xL), _mm256_abs_epi16(G2yL));
64  R0H = _mm256_add_epi16(_mm256_abs_epi16(G0xH), _mm256_abs_epi16(G0yH));
65  R1H = _mm256_add_epi16(_mm256_abs_epi16(G1xH), _mm256_abs_epi16(G1yH));
66  R2H = _mm256_add_epi16(_mm256_abs_epi16(G2xH), _mm256_abs_epi16(G2yH));
67
68  R0 = _mm256_packus_epi16(R0L, R0H);
69  R1 = _mm256_packus_epi16(R1L, R1H);
70  R2 = _mm256_packus_epi16(R2L, R2H);
71  _mm256_store_si256((__m256i*)pD8, R0);
72  _mm256_store_si256((__m256i*)(pD8 + dst_image_width), R1);
73  _mm256_store_si256((__m256i*)(pD8 + (2 * dst_image_width)), R2);
```

รูปที่ 3.11 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ชุดคำสั่ง AVX สำหรับ *L*==3

การใช้งานชุดคำสั่ง AVX ในการเขียนโปรแกรมสำหรับวิธีการที่นำเสนอในการหาขอบภาพแบบ *L*==3 โดยโปรแกรมในรูปที่ 3.11 บรรทัดที่ 16-21 เป็นการโหลดข้อมูลจากภาพโดยใช้ฟังก์ชัน _mm256_loadu_si256 ในแต่ละครั้งจะโหลดข้อมูลได้จำนวน 256 บิต ซึ่งจะได้ข้อมูลภาพขนาด 8 บิต จำนวน 32 พิกเซล ต่อมาในบรรทัดที่ 23-27 โปรแกรมได้ขยายข้อมูล 8 บิต ให้เป็นข้อมูล 16 บิต เพื่อนำไปคำนวณในการเขียนโปรแกรมด้วยชุดคำสั่ง AVX โดยใช้ฟังก์ชัน _mm256_unpacklo_epi8 สำหรับ P0L-P14L และใช้ฟังก์ชัน _mm256_unpackhi_epi8 สำหรับ P0H-P14H ดังแผนภาพในรูปที่ 3.9 จากนั้นในบรรทัดที่ 29-39 เป็นการคำนวณค่าไว้ล่วงหน้าเพื่อเก็บผลลัพธ์ที่สามารถนำมาใช้คำนวณร่วมกันได้ ซึ่งจะนำมาคำนวณหา Gradient magnitudes ในบรรทัดที่ 41-59 จากนั้นจึงนำผลลัพธ์ของ Gradient magnitudes ในแนวแกนนอนและแกนตั้งที่ได้ไปคำนวณหาขอบของภาพในบรรทัดที่ 61-66 และเมื่อได้ผลลัพธ์ของการคำนวณเรียบร้อยในบรรทัดที่ 68-70 โปรแกรมจะลดขนาดข้อมูลจาก 16 บิต ให้กลับไปอยู่ในลักษณะพิกเซลของภาพ 8 บิต ด้วยฟังก์ชัน _mm256_packus_epi16 ดังแผนภาพในรูปที่ 3.9 และบรรทัดที่ 71-73 เป็นการคืนค่าข้อมูลไปยังภาพผลลัพธ์ที่ต้องการด้วยฟังก์ชัน _mm256_store_si256

3.3.3 การเขียนโปรแกรมโดยวิธีที่นำเสนอ สำหรับ *L*==4

ในหัวข้อนี้จะอธิบายวิธีที่ใช้ในการเขียนโปรแกรมด้วยชุดคำสั่ง AVX โดยวิธีการที่นำเสนอเริ่มต้นจากการเขียนโปรแกรมภาษาซีด้วยวิธีที่นำเสนอในรูปที่ 3.12 และปรับปรุงโปรแกรมให้เป็นการเขียนด้วยชุดคำสั่ง AVX โดยใช้วิธีที่นำเสนอในรูปที่ 3.13

```
1   char *pS8 = source.ptr<uchar>(0, 0);
2   char *pD8 = dst.ptr<uchar>(0, 0);
3
4   typedef short s;
5
6   short D0x, D1x, D2x, D3x, D4x, D5x, S0y,S1y,S2y,S3y, S4y, S5y;
7   short P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11, P12, P13, P14, P15, P16, P17;
8
9   int source_image_width = source.cols;
10  int dst_image_width = dst.cols;
11
12  int step2 = source_image_width;
13  int step3 = 2*source_image_width;
14  int step4 = 3*source_image_width;
15  int step5 = 4*source_image_width;
16  int step6 = 5*source_image_width;
17
18  P0 = (s)*pS8;             P1 = (s)*(pS8+1);          P2 = (s)*(pS8+2);
19  P3 = (s)*(pS8+step2);     P4 = (s)*(pS8+step2+1);    P5 = (s)*(pS8+step2+2)
20  P6 = (s)*(pS8+step3);     P7 = (s)*(pS8+step3+1);    P8 = (s)*(pS8+step3+2)
21  P9 = (s)*(pS8+step4);     P10 = (s)*(pS8+step4+1);   P11 = (s)*(pS8+step4+2)
22  P12 = (s)*(pS8+step5);    P13 = (s)*(pS8+step5+1);   P14 = (s)*(pS8+step5+2)
23  P15 = (s)*(pS8+step6);    P16 = (s)*(pS8+step6+1);   P17 = (s)*(pS8+step6+2)
24
25  D0x = P0-P2;              S0y = P0+P2;
26  D1x = P3-P5;              S1y = P3+P5;
27  D2x = P6-P8;              S2y = P6+P8;
28  D3x = P9-P11;             S3y = P9+P11;
29  D4x = P12-P14;            S4y = P12+P14;
30  D5x = P15-P17;            S5y = P15+P17;
31
32  short G0x = D0x + (D1x<<1) +D2x;
33  short G1x = D1x + (D2x<<1) +D3x;
34  short G2x = D2x + (D3x<<1) +D4x;
35  short G3x = D3x + (D4x<<1) +D5x;
36  short G0y = S0y - S2y + (P1-P7)<<1;
37  short G1y = S1y - S3y + (P4-P10)<<1;
38  short G2y = S2y - S4y + (P7-P13)<<1;
39  short G3y = S3y - S5y + (P10-P16)<<1;
40
41  short R0 = abs(G0x) + abs(G0y);
42  short R1 = abs(G1x) t abs(G1y);
43  short R3 = abs(G2x) t abs(G2y);
44  short R4 = abs(G3x) t abs(G3y);
45  *(pD8) = R0;
46  *(pD8 + dst_image_width) = R1;
47  *(pD8 + (2 * dst_image_width)) = R2;
48  *(pD8 + (3 * dst_image_width)) = R3;
```

รูปที่ 3.12 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ภาษาซี สำหรับ *L*==4

การใช้งานภาษาซีในการเขียนโปรแกรมสำหรับวิธีการที่นำเสนอในการหาขอบภาพแบบ *L*==4 โดยโปรแกรมในรูปที่ 3.12 บรรทัดที่ 18-23 เป็นการโหลดข้อมูลจากภาพจำนวน 18 พิกเซล และบรรทัดที่ 25-30 เป็นการคำนวณค่าไว้ล่วงหน้าเพื่อเก็บผลลัพธ์ที่สามารถนำมาใช้คำนวณร่วมกันได้ ซึ่งจะนำมาคำนวณหา Gradient magnitudes ในบรรทัดที่ 32-39 จากนั้นจึงนำผลลัพธ์ของ Gradient magnitudes ในแนวแกนนอนและแกนตั้งที่ได้ไปคำนวณหาขอบของภาพในบรรทัด

ที่ 41-44 จะเห็นได้ว่าวิธีการเขียนโปรแกรมแบบที่นำเสนอสามารถลดจำนวนครั้งของการคำนวณได้โดยการแชร์ผลลัพธ์ของการคำนวณร่วมกัน และในหนึ่งรอบของการประมวลผลแบบ L==3 จะได้ผลลัพธ์จำนวน 4 บรรทัดคือ R0 , R1, R3 และ R4

```
1   char *pS8 = source.ptr<uchar>(0, 0);
2   char *pD8 = dst.ptr<uchar>(0, 0);
3
4   __m256i P0, P1, ..., P15, P16, P17;
5   __m256i P0L, P1L, ..., P15L, P16L, P17L;
6   __m256i P0H, P1H, ..., P15H, P16H, P17H;
7
8   __m256i z = _mm256_setzero_si256();
9   int source_image_width = source.cols;
10  int dst_image_width = dst.cols;
11
12  int step2 = source_image_width;
13  int step3 = 2*source_image_width;
14  int step4 = 3*source_image_width;
15  int step5 = 4*source_image_width;
16  int step6 = 5*source_image_width;
17
18  P0 = _mm256_loadu_si256((__m256i*)pS8);
19  P1 = _mm256_loadu_si256((__m256i*)(pS8 + 1));
20  P2 = _mm256_loadu_si256((__m256i*)(pS8 + 2));
21  ...
22  P15 = _mm256_loadu_si256((__m256i*)(pS8 + step6));
23  P16 = _mm256_loadu_si256((__m256i*)(pS8 + step6 + 1));
24  P17 = _mm256_loadu_si256((__m256i*)(pS8 + step6 + 2));
25
26  P0L = _mm256_unpacklo_epi8(P0, z);          P0H = _mm256_unpackhi_epi8(P0, z);
27  P1L = _mm256_unpacklo_epi8(P1, z);          P1H = _mm256_unpackhi_epi8(P1, z);
28  P2L = _mm256_unpacklo_epi8(P2, z);          P2H = _mm256_unpackhi_epi8(P2, z);
29  ...
30  P15L = _mm256_unpacklo_epi8(P15, z);        P15H = _mm256_unpackhi_epi8(P15,
    z);                        ...
31  P16L = _mm256_unpacklo_epi8(P16, z);        P16H = _mm256_unpackhi_epi8(P16, z);
32  P17L = _mm256_unpacklo_epi8(P17, z);        P17H = _mm256_unpackhi_epi8(P17, z);
33
34  D0L = _mm256_sub_epi16(P0L, P2L);           D0H = _mm256_sub_epi16(P0H, P2H);
35  D1L = _mm256_sub_epi16(P3L, P5L);           D1H = _mm256_sub_epi16(P3H, P5H);
36  D2L = _mm256_sub_epi16(P6L, P8L);           D2H = _mm256_sub_epi16(P6H, P8H);
37  D3L = _mm256_sub_epi16(P9L, P11L);          D3H = _mm256_sub_epi16(P9H, P11H);
38  D4L = _mm256_sub_epi16(P12L, P14L);         D4H = _mm256_sub_epi16(P12H, P14H);
39  D5L = _mm256_sub_epi16(P15L, P17L);         D5H = _mm256_sub_epi16(P15H, P17H);
40
41  S0L = _mm256_add_epi16(P0L, P2L);           S0H = _mm256_add_epi16(P0H, P2H);
42  S1L = _mm256_add_epi16(P6L, P8L);           S1H = _mm256_add_epi16(P6H, P8H);
43  S2L = _mm256_add_epi16(P3L, P5L);           S2H = _mm256_add_epi16(P3H, P5H);
44  S3L = _mm256_add_epi16(P9L, P11L);          S3H = _mm256_add_epi16(P9H, P11H);
45  S4L = _mm256_add_epi16(P12L, P14L);         S4H = _mm256_add_epi16(P12H, P14H);
46  S5L = _mm256_add_epi16(P15L, P17L);         S5H = _mm256_add_epi16(P15H, P17H);
47
48  G0xL = _mm256_add_epi16(_mm256_add_epi16(D0L, _mm256_slli_epi16(D1L, 1)), D2L);
49  G1xL = _mm256_add_epi16(_mm256_add_epi16(D1L, _mm256_slli_epi16(D2L, 1)), D3L);
50  G2xL = _mm256_add_epi16(_mm256_add_epi16(D2L, _mm256_slli_epi16(D3L, 1)), D4L);
51  G3xL = _mm256_add_epi16(_mm256_add_epi16(D3L, _mm256_slli_epi16(D4L, 1)), D5L);
52  D0L = _mm256_sub_epi16(S0L, S1L);
53  D1L = _mm256_sub_epi16(S2L, S3L);
54  D2L = _mm256_sub_epi16(S1L, S4L);
55  D3L = _mm256_sub_epi16(S3L, S5L);
56  G0yL = _mm256_add_epi16(D0L, _mm256_slli_epi16(_mm256_sub_epi16(P1L, P7L), 1));
57  G1yL = _mm256_add_epi16(D1L, _mm256_slli_epi16(_mm256_sub_epi16(P4L, P10L), 1));
58  G2yL = _mm256_add_epi16(D2L, _mm256_slli_epi16(_mm256_sub_epi16(P7L, P13L), 1));
59  G3yL = _mm256_add_epi16(D3L, _mm256_slli_epi16(_mm256_sub_epi16(P10L, P16L), 1));
60
61  G0xH = _mm256_add_epi16(_mm256_add_epi16(D0H, _mm256_slli_epi16(D1H, 1)), D2H);
62  G1xH = _mm256_add_epi16(_mm256_add_epi16(D1H, _mm256_slli_epi16(D2H, 1)), D3H);
63  G2xH = _mm256_add_epi16(_mm256_add_epi16(D2H, _mm256_slli_epi16(D3H, 1)), D4H);
64  G3xH = _mm256_add_epi16(_mm256_add_epi16(D3H, _mm256_slli_epi16(D4H, 1)), D5H);
65  D0H = _mm256_sub_epi16(S0H, S1H);
66  D1H = _mm256_sub_epi16(S2H, S3H);
```

```
66    D1H = _mm256_sub_epi16(S2H, S3H);
67    D2H = _mm256_sub_epi16(S1H, S4H);
68    D3H = _mm256_sub_epi16(S3H, S5H);
69    G0yH = _mm256_add_epi16(D0H, _mm256_slli_epi16(_mm256_sub_epi16(P1H, P7H), 1));
70    G1yH = _mm256_add_epi16(D1H, _mm256_slli_epi16(_mm256_sub_epi16(P4H, P10H), 1));
71    G2yH = _mm256_add_epi16(D2H, _mm256_slli_epi16(_mm256_sub_epi16(P7H, P13H), 1));
72    G3yH = _mm256_add_epi16(D3H, _mm256_slli_epi16(_mm256_sub_epi16(P10H, P16H), 1));
73    R0L = _mm256_add_epi16(_mm256_abs_epi16(G0xL), _mm256_abs_epi16(G0yL));
74    R1L = _mm256_add_epi16(_mm256_abs_epi16(G1xL), _mm256_abs_epi16(G1yL));
75    R2L = _mm256_add_epi16(_mm256_abs_epi16(G2xL), _mm256_abs_epi16(G2yL));
76    R3L = _mm256_add_epi16(_mm256_abs_epi16(G3xL), _mm256_abs_epi16(G3yL));
77    R0H = _mm256_add_epi16(_mm256_abs_epi16(G0xH), _mm256_abs_epi16(G0yH));
78    R1H = _mm256_add_epi16(_mm256_abs_epi16(G1xH), _mm256_abs_epi16(G1yH));
79    R2H = _mm256_add_epi16(_mm256_abs_epi16(G2xH), _mm256_abs_epi16(G2yH));
80    R3H = _mm256_add_epi16(_mm256_abs_epi16(G3xH), _mm256_abs_epi16(G3yH));
81
82    R0 = _mm256_packus_epi16(R0L, R0H);
83    R1 = _mm256_packus_epi16(R1L, R1H);
84    R2 = _mm256_packus_epi16(R2L, R2H);
85    R3 = _mm256_packus_epi16(R3L, R3H);
86    _mm256_store_si256((__m256i*)pD8, R0);
87    _mm256_store_si256((__m256i*)(pD8 + dst_image_width), R1);
88    _mm256_store_si256((__m256i*)(pD8 + (2 * dst_image_width)), R2);
89    _mm256_store_si256((__m256i*)(pD8 + (3 * dst_image_width)), R3);
```

รูปที่ 3.13 การเขียนโปรแกรมด้วยวิธีที่นำเสนอโดยใช้ชุดคำสั่ง AVX สำหรับ *L*==4

การใช้งานชุดคำสั่ง AVX ในการเขียนโปรแกรมสำหรับวิธีการที่นำเสนอในการหาขอบภาพแบบ *L*==4 โดยโปรแกรมในรูปที่ 3.13 บรรทัดที่ 18-24 เป็นการโหลดข้อมูลจากภาพโดยใช้ฟังก์ชัน _mm256_loadu_si256 ในแต่ละครั้งจะโหลดข้อมูลได้จำนวน 256 บิต ซึ่งจะได้ข้อมูลภาพขนาด 8 บิต จำนวน 32 พิกเซล ต่อมาในบรรทัดที่ 26-32 โปรแกรมได้ขยายข้อมูล 8 บิต ให้เป็นข้อมูล 16 บิต เพื่อนำไปคำนวณโดยใช้คำสั่ง AVX ในภายหลังซึ่งจะประมวลผลข้อมูลละ 16 บิตจำนวน 16 ข้อมูล การขยายข้อมูลในขั้นตอนนี้ทำโดยใช้ฟังก์ชัน _mm256_unpacklo_epi8 สำหรับ P0L-P17L และใช้ฟังก์ชัน _mm256_unpackhi_epi8 สำหรับ P0H-P17H ดังแผนภาพในรูปที่ 3.9 จากนั้นในบรรทัดที่ 34-46 เป็นการคำนวณค่าไว้ล่วงหน้าเพื่อเก็บผลลัพธ์ที่สามารถนำมาใช้คำนวณร่วมกันได้ ซึ่งจะนำมาคำนวณหา Gradient magnitudes ในบรรทัดที่ 48-72 จากนั้นจึงนำผลลัพธ์ของ Gradient magnitudes ในแนวแกนนอนและแกนตั้งที่ได้ไปคำนวณหาขอบของภาพในบรรทัดที่ 73-80 และเมื่อได้ผลลัพธ์ของการคำนวณเรียบร้อยในบรรทัดที่ 82-85 โปรแกรมจะลดขนาดข้อมูลจาก 16 บิต ให้กลับไปอยู่ในลักษณะพิกเซลของภาพ 8 บิต ด้วยฟังก์ชัน _mm256_packus_epi16 ดังแผนภาพในรูปที่ 3.9 และบรรทัดที่ 86-89 เป็นการคืนค่าข้อมูลไปยังภาพผลลัพธ์ที่ต้องการด้วยฟังก์ชัน _mm256_store_si256 จะเห็นได้ว่าเมื่อ *L* มีขนาดเพิ่มมากขึ้นจำนวนบรรทัดของการเขียนโปรแกรมก็จะเพิ่มมากขึ้น ส่งผลให้ขนาดของโปรแกรมใหญ่ขึ้น ดังนั้นค่าประสิทธิภาพที่ได้จากการลดจำนวนครั้งในการคำนวณที่คณิตศาสตร์และขนาดของโปรแกรมที่ใหญ่ขึ้นนี้จึงเป็นสิ่งที่จะต้องมีการหาจุดที่ลงตัวให้ได้ว่าค่า *L* ที่เหมาะสมควรเป็นค่าใด จึงจะทำให้ประสิทธิภาพโดยรวมออกมาดีที่สุด

3.4 การเพิ่มประสิทธิภาพในการหาขอบภาพโดยการใช้ OpenMP

ในขั้นตอนนี้เป็นการกระจายข้อมูลที่จะนำมาคำนวณไปยังหน่วยประมวลผลต่าง ๆ ให้ช่วยกันทำงานในลักษณะของการทำงานแบบขนาด โดยใช้ OpenMP เข้ามาช่วยการจัดสรรการทำงานร่วมกันในลักษณะของการทำงานแบบมัลติคอร์ ดังแสดงการเขียนโปรแกรมภาษาซีดังต่อไปนี้

```
#pragma omp parallel
{
    int step = image_height / omp_get_num_threads();
    for (int x = (omp_get_thread_num() * step); x < ((omp_get_thread_num() + 1) * step);
                x += num_line)
   {
        for (int y = 0; y < image_width; y++)
        {  //place the proposed method in this loop-body
            ....
            ....
        }
    }
}
```

รูปที่ 3.14 การใช้ OpenMP สำหรับวิธีการที่นำเสนอ

รูปที่ 3.14 การใช้ **#pragma omp parallel** เป็นการสั่งเริ่มต้นการทำงานแบบนานทั้งหมดที่อยู่ในใต้คำสั่งนี้ โดยในโปรแกรมจะใช้อัลกอริทึมในการแบ่งส่วนของงานออกเป็นส่วนๆเท่ากับจำนวน Thread ทั้งหมดที่มีในหน่วยประมวลผลเพื่อให้สามารถประมวลผลได้เร็วขึ้นและเป็นการทำงานที่ไม่ซ้อนกัน

3.5 การประยุกต์ใช้อัลกอริทึมที่นำเสนอในการเพิ่มประสิทธิภาพของการหาขอบภาพด้วยวิธีแคนนี

เป็นการนำเทคนิควิธีการเพิ่มความเร็วในการหาของภาพโดยวิธีโซเบลที่นำเสนอผ่านมาทั้งหมดในหัวข้อที่ 3.1-3.4 มาประยุกต์ใช้กับวิธีการหาขอบภาพของแคนนี ซึ่งวิธีแคนนีเป็นวิธีที่มีผลลัพธ์ในการหาขอบภาพที่ดี แต่มีความซับซ้อนมากกว่าและใช้เวลาในการหาขอบภาพที่นานขึ้นเช่นกัน เพื่อให้ความเร็วของการหาขอบภาพของวิธีแคนนีประมวลผลได้เร็วขึ้น ผู้วิจัยจึงได้ทำการทดสอบว่าหากแทนกลไกการหา Gradient magnitude ในอัลกอริทึมของแคนนีด้วยวิธีที่นำเสนอแล้ว จะช่วยให้ผลการหาขอบภาพด้วยอัลกอริทึมของแคนนีมีประสิทธิภาพสูงขึ้นหรือไม่ อย่างไร โดยผลการทดลองในส่วนนี้จะอยู่ในหัวข้อ 4.2 ของบทที่ 4

บทที่ 4
การทดสอบและการวิเคราะห์ประสิทธิภาพ

ผู้วิจัยได้ทดสอบวิธีการหาขอบของภาพด้วยวิธีที่นำเสนอ เพื่อตรวจสอบประสิทธิภาพของวิธีการดังกล่าว แล้วนำมาเปรียบเทียบกับการใช้คำสั่งมาตรฐานของไลบรารี OpenCV ในการประมวลผลภาพชนิด 8 บิต โดยทดลองกับภาพขนาดต่าง ๆ ได้แก่ 800x600, 1024x768, 1280x1024, 1920×1080, 2560×1440, 3872x2160, 4928x3264 และ 7680x4320 พิกเซล ซึ่งเป็นขนาดของภาพที่เป็นที่นิยมใช้งานโดยทั่วไป ด้วยการทดสอบบนซีพียู Intel Core i5-1135G7 ของสถาปัตยกรรม Tiger Lake โดยการทดสอบหลัก แบ่งออกเป็น 2 ส่วน ได้แก่ การทดสอบเพื่อหาประสิทธิภาพของการหาขอบด้วยวิธีที่เสนอเทียบกับวิธีการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐาน OpenCV และ การทดสอบประสิทธิภาพในการหาขอบภาพของวิธีที่เสนอเมื่อเปรียบเทียบกับการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV ซึ่งนำเสนอหัวข้อดังนี้

4.1 ผลการทดสอบประสิทธิภาพในการหาขอบภาพด้วยวิธีโซเบลด้วยวิธีที่นำเสนอ

4.2 ผลการทดสอบการประยุกต์ใช้การเพิ่มประสิทธิภาพในการหาขอบภาพสำหรับแคนนี

4.3 การวิเคราะห์ผลการทดลอง

4.1 ผลการทดสอบประสิทธิภาพในการหาขอบภาพด้วยวิธีโซเบลด้วยวิธีที่นำเสนอ

ในหัวข้อนี้ผู้วิจัยได้ทดสอบการหา Gradient ในแนวนอน (Horizontal gradient) และ Gradient ในแนวตั้ง (Vertical gradient) และการตรวจจับขอบของภาพ (Edge detection) ด้วยวิธีการที่เสนอเปรียบเทียบกับวิธี Sobel ในไลบรารีมาตรฐานของ OpenCV สำหรับการทดสอบหาขนาดของค่า L ที่ดีที่สุด จาก 2 ถึง 16 ของรูปภาพภาพขนาดต่าง ๆ โดยกำหนดให้ L คือ จำนวนบรรทัดของข้อมูลภาพที่ต้องการประมวลผลในแต่ล่ะรอบ โดยการทดสอบด้วยการประมวลผลโปรแกรมจำนวน 100 ครั้ง จากนั้นจึงเฉลี่ยเวลาที่ใช้ เพื่อคำนวณ Gradient ของทั้งสองแกนและการตรวจจับขอบ และแสดงเวลาที่ใช้ในการคำนวณ และเปรียบเทียบความเร็วในการหา Gradient ในแนวนอน (G_x) ความเร็วในการหา Gradient ในแนวตั้ง (G_y) และความเร็วการตรวจจับขอบภาพเปรียบเทียบกับวิธีการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ซึ่งฟังก์ชันและพารามิเตอร์ที่ใช้ใน OpenCV ที่นำเปรียบเทียบกับวิธีที่นำเสนอได้แก่

1. หา Gradient ในแนวนอน (G_x) ด้วยการใช้ฟังก์ชันและพารามิเตอร์ดังนี้

   ```
   Sobel(img, grad_x, CV_16S, 1, 0, 3, -1, 0, 0);
   ```

2. หา Gradient ในแนวตั้ง (G_y) ด้วยการใช้ฟังก์ชันและพารามิเตอร์ดังนี้

   ```
   Sobel(img, grad_y, CV_16S, 0, 1, 3, -1, 0, 0);
   ```

3. หาขอบภาพ (Edge detection) โดยการใช้ฟังก์ชันและพารามิเตอร์ดังนี้

```
Sobel(img, grad_x, CV_16S, 1, 0, 3, -1, 0, 0);
Sobel(img, grad_y, CV_16S, 0, 1, 3, -1, 0, 0);
convertScaleAbs(grad_x, abs_grad_x);
convertScaleAbs(grad_y, abs_grad_y);
addWeighted(abs_grad_x, 1, abs_grad_y, 1, 0, filter_opencv);
```

ตารางที่ 4.1 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแนวนอน (G_x) และหาค่า Gradient ในแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาดต่าง ๆ ด้วยการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV

Image size (pixel)	G_x (ms)	G_y (ms)	$\|G_x\|+\|G_y\|$ (ms)
800x600	0.1822	0.1823	0.4668
1024x768	0.2789	0.2879	0.7856
1280x1024	0.4369	0.4607	1.5056
1920x1080	0.6494	0.7108	2.6612
2560x1440	1.1811	1.3019	4.9224
3872x2160	3.3637	3.2107	11.2992
4928x3264	5.6138	5.9413	21.8250
7680x4320	13.6540	14.1082	44.7104

จากตารางที่ 4.1 แสดงระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแนวนอน (G_x) และหาค่า Gradient ในแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาดต่าง ๆ ด้วยการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV เพื่อนำไปเปรียบเทียบความเร็วกับผลการทดลองของวิธีการที่นำเสนอดังต่อไปนี้

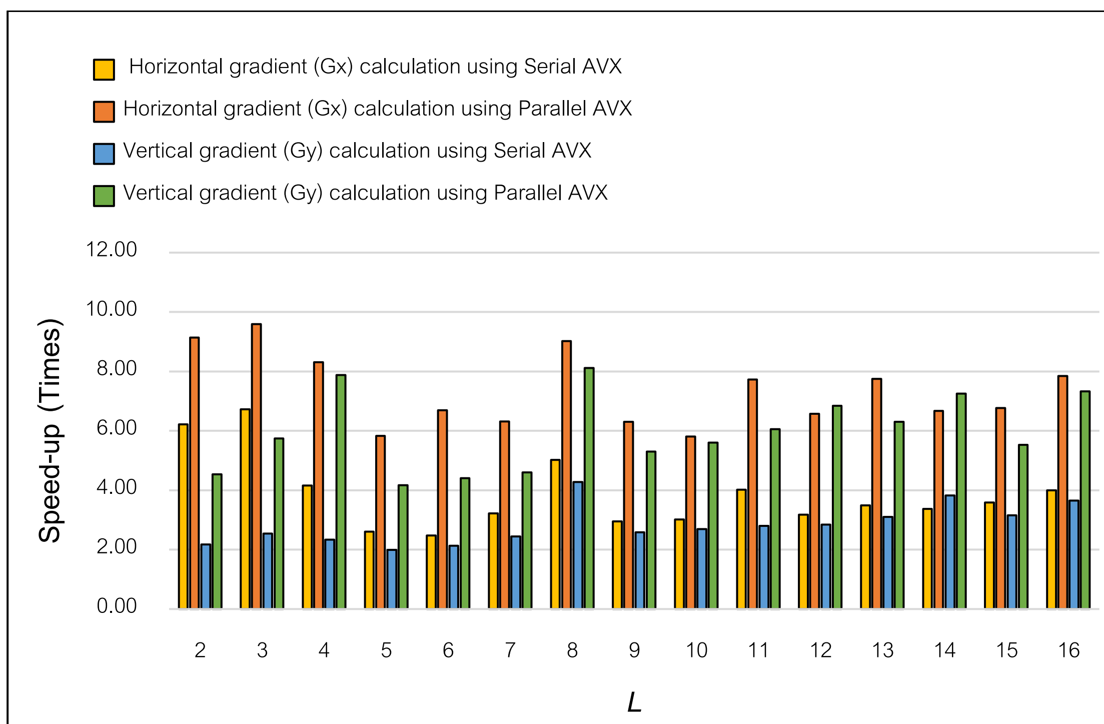
4.1.1 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 800x600 พิกเซล ที่ L ขนาดต่าง ๆ

ผลการทดสอบหาระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (G_x) และหาค่า Gradient ในแกนแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 800x600 พิกเซล ด้วยวิธีการที่นำเสนอ ดังแสดงในตารางที่ 4.2

ตารางที่ 4.2 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน ($\boldsymbol{G_x}$) และหาค่า Gradient ในแกนแนวตั้ง ($\boldsymbol{G_y}$) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 800x600 พิกเซล ด้วยวิธีการที่นำเสนอ

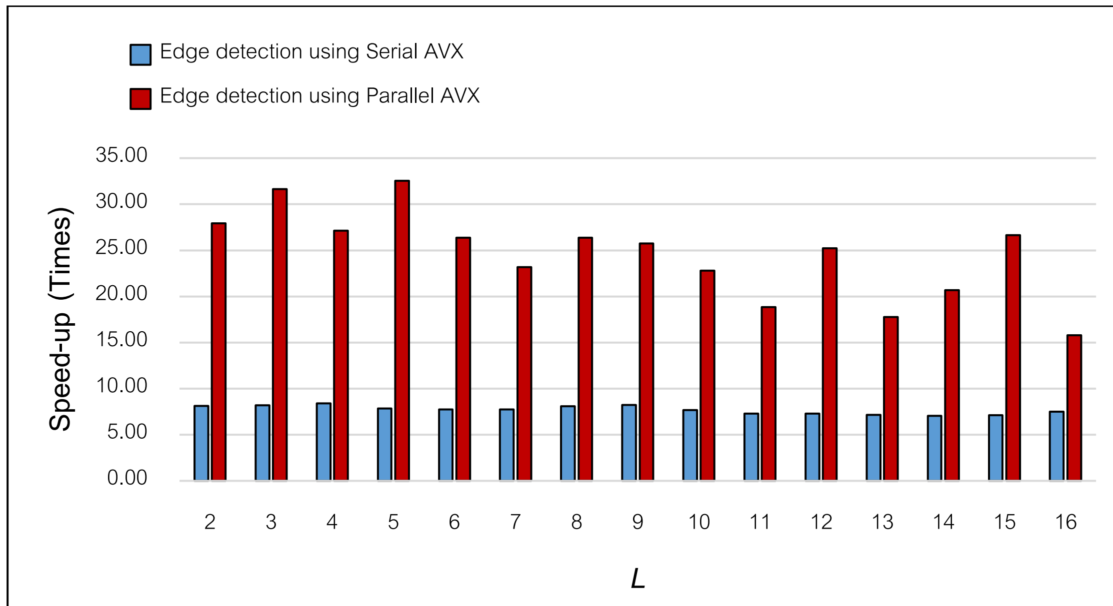
L	G_x (ms)		G_y (ms)		$\|G_x\|+\|G_y\|$ (ms)	
	serial AVX	parallel AVX	serial AVX	parallel AVX	serial AVX	parallel AVX
2	0.0293	0.0199	0.0839	0.0402	0.0575	0.0167
3	**0.0271**	**0.0190**	0.0717	0.0317	0.0570	0.0148
4	0.0438	0.0219	0.0781	0.0231	**0.0556**	0.0172
5	0.0699	0.0312	0.0915	0.0437	0.0594	**0.0144**
6	0.0736	0.0272	0.0856	0.0414	0.0602	0.0177
7	0.0565	0.0289	0.0747	0.0396	0.0603	0.0201
8	0.0363	0.0202	**0.0426**	**0.0225**	0.0578	0.0177
9	0.0617	0.0289	0.0706	0.0344	0.0568	0.0181
10	0.0603	0.0314	0.0678	0.0325	0.0607	0.0205
11	0.0453	0.0236	0.0652	0.0301	0.0639	0.0248
12	0.0573	0.0277	0.0641	0.0266	0.0639	0.0185
13	0.0522	0.0235	0.0587	0.0289	0.0653	0.0263
14	0.0541	0.0273	0.0476	0.0251	0.0661	0.0226
15	0.0509	0.0269	0.0577	0.0330	0.0656	0.0175
16	0.0456	0.0232	0.0499	0.0249	0.0622	0.0296

จากระยะเวลาการประมวลผลในตารางที่ 4.2 จะเห็นได้ว่า การหา Gradient ในแนวนอน (G_x) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า *L* ที่ดีที่สุดคือ *L*==3 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า *L* ที่ดีที่สุดคือ *L*==3 และสำหรับการหา Gradient ในแนวตั้ง (G_y) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า *L* ที่ดีที่สุดคือ *L*==8 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า *L* ที่ดีที่สุดคือ *L*==8 สำหรับการตรวจจับหาขอบภาพนั้น วิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรม ค่า *L* ที่ดีที่สุดคือ *L*==4 และวิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบขนานค่า *L* ที่ดีที่สุด คือ *L*==5



รูปที่ 4.1 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 800x600 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.1 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนที่ดีที่สุดโดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 6.73 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 9.60 เท่า โดยค่าเฉลี่ยของการหา Gradient ในแนวนอนแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 3.87 และ 7.36 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวนอนด้วยไลบรารีมาตรฐานของ OpenCV และสำหรับการหา Gradient ในแนวตั้งที่ดีที่สุดด้วยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานมีความเร็วเพิ่มขึ้น 4.28 และ 8.12 เท่าตามลำดับ โดยค่าเฉลี่ยอยู่ที่ 2.84 และ 5.98 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวตั้งด้วยไลบรารี OpenCV



รูปที่ 4.2 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 800x600 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.2 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการตรวจจับขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 8.40 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 32.53 เท่า โดยค่าเฉลี่ยของการหาขอบภาพแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 7.70 และ 24.58 เท่าตามลำดับ เมื่อเทียบกับการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐานของ OpenCV

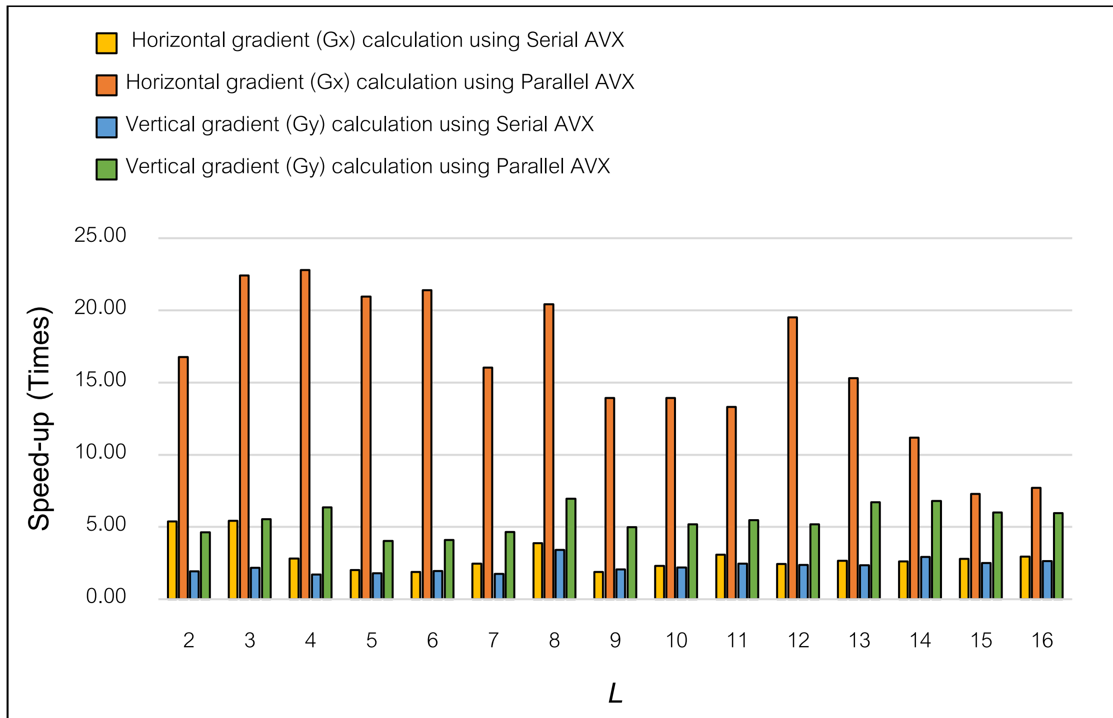
4.1.2 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 1024x768 พิกเซล ที่ L ขนาดต่าง ๆ

ผลการทดสอบหาระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (G_x) และหาค่า Gradient ในแกนแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 1024x768 พิกเซล ด้วยวิธีการที่นำเสนอ ดังแสดงในตารางที่ 4.3

ตารางที่ 4.3 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน ($\boldsymbol{G_x}$) และหาค่า Gradient ในแกนแนวตั้ง ($\boldsymbol{G_y}$) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 1024x768 พิกเซล ด้วยวิธีการที่นำเสนอ

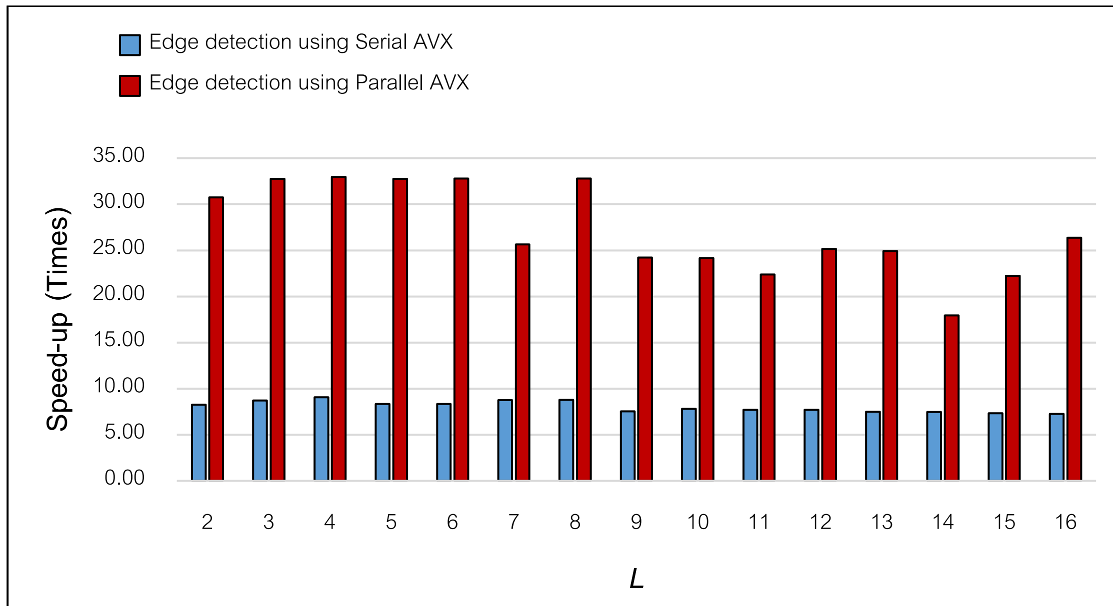
L	G_x (ms)		G_y (ms)		$\lvert G_x \rvert + \lvert G_y \rvert$ (ms)	
	serial AVX	parallel AVX	serial AVX	parallel AVX	serial AVX	parallel AVX
2	0.0518	0.0166	0.1492	0.0622	0.0951	0.0256
3	**0.0514**	0.0124	0.1329	0.0519	0.0899	0.0240
4	0.0987	**0.0122**	0.1677	0.0452	**0.0867**	**0.0238**
5	0.1375	0.0133	0.1597	0.0714	0.0942	0.0240
6	0.1478	0.0130	0.1476	0.0702	0.0943	0.0240
7	0.1128	0.0174	0.1636	0.0619	0.0896	0.0306
8	0.0717	0.0137	**0.0842**	**0.0414**	0.0895	0.0240
9	0.1485	0.0200	0.1393	0.0577	0.1041	0.0324
10	0.1207	0.0200	0.1306	0.0554	0.1006	0.0325
11	0.0907	0.0210	0.1174	0.0527	0.1018	0.0351
12	0.1142	0.0143	0.1210	0.0554	0.1017	0.0312
13	0.1045	0.0182	0.1225	0.0428	0.1049	0.0315
14	0.1065	0.0249	0.0985	0.0424	0.1053	0.0438
15	0.0995	0.0383	0.1144	0.0479	0.1070	0.0353
16	0.0942	0.0362	0.1095	0.0483	0.1081	0.0298

จากระยะเวลาการประมวลผลในตารางที่ 4.3 จะเห็นได้ว่า การหา Gradient ในแนวนอน (G_x) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==3 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==4 และสำหรับการหา Gradient ในแนวตั้ง (G_y) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==8 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==8 เช่นเดียวกัน สำหรับการตรวจจับหาขอบภาพนั้น วิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุด คือ L==4 และวิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบขนานค่า L ที่ดีที่สุด คือ L==4



รูปที่ 4.3 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 1024x768 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.3 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนที่ดีที่สุดโดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 5.43 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 22.80 เท่า โดยค่าเฉลี่ยของการหา Gradient ในแนวนอนแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 2.98 และ 16.20 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวนอนด้วยไลบรารีมาตรฐานของ OpenCV และสำหรับการหา Gradient ในแนวตั้งที่ดีที่สุดด้วยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานมีความเร็วเพิ่มขึ้น 3.42 และ 6.96 เท่าตามลำดับ โดยค่าเฉลี่ยอยู่ที่ 2.28 และ 5.51 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวตั้งด้วยไลบรารี OpenCV



รูปที่ 4.4 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 1024x768 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.4 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการตรวจจับขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 9.06 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 32.97 เท่า โดยค่าเฉลี่ยของการหาขอบภาพแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 8.04 และ 27.18 เท่าตามลำดับ เมื่อเทียบกับการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐานของ OpenCV

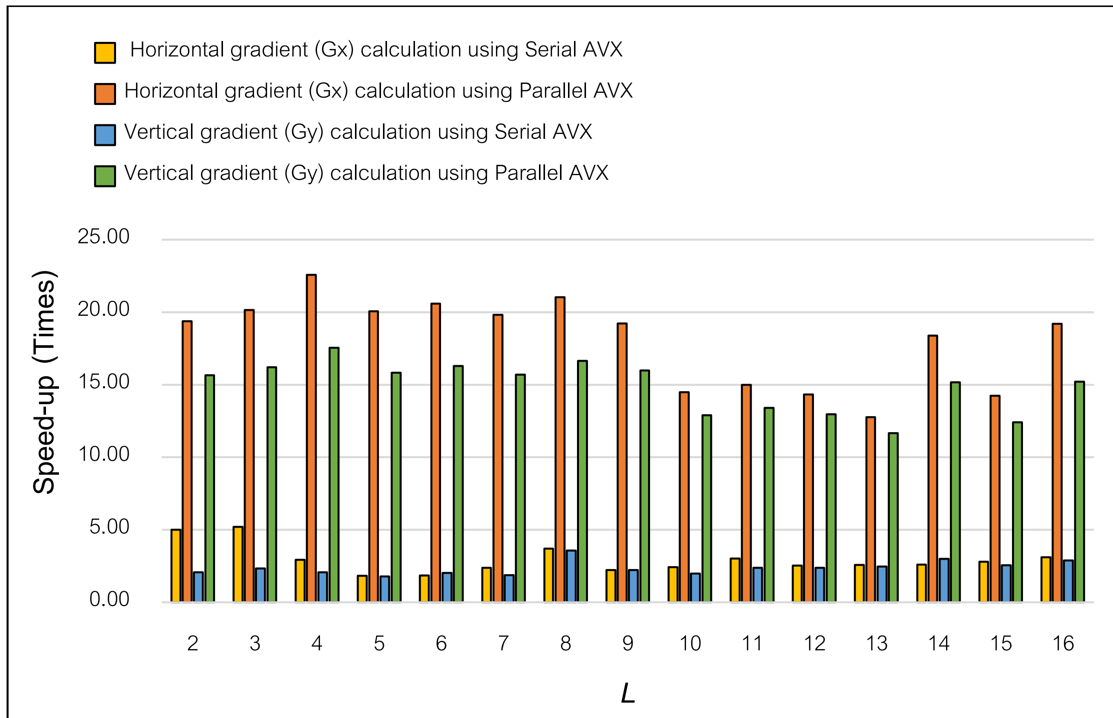
4.1.3 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 1280x1024 พิกเซล ที่ L ขนาดต่าง ๆ

ผลการทดสอบหาระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (G_x) และหาค่า Gradient ในแกนแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 1280x1024 พิกเซล ด้วยวิธีการที่นำเสนอ ดังแสดงในตารางที่ 4.4

ตารางที่ 4.4 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน ($\boldsymbol{G_x}$) และหาค่า Gradient ในแกนแนวตั้ง ($\boldsymbol{G_y}$) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 1280x1024 พิกเซล ด้วยวิธีการที่นำเสนอ

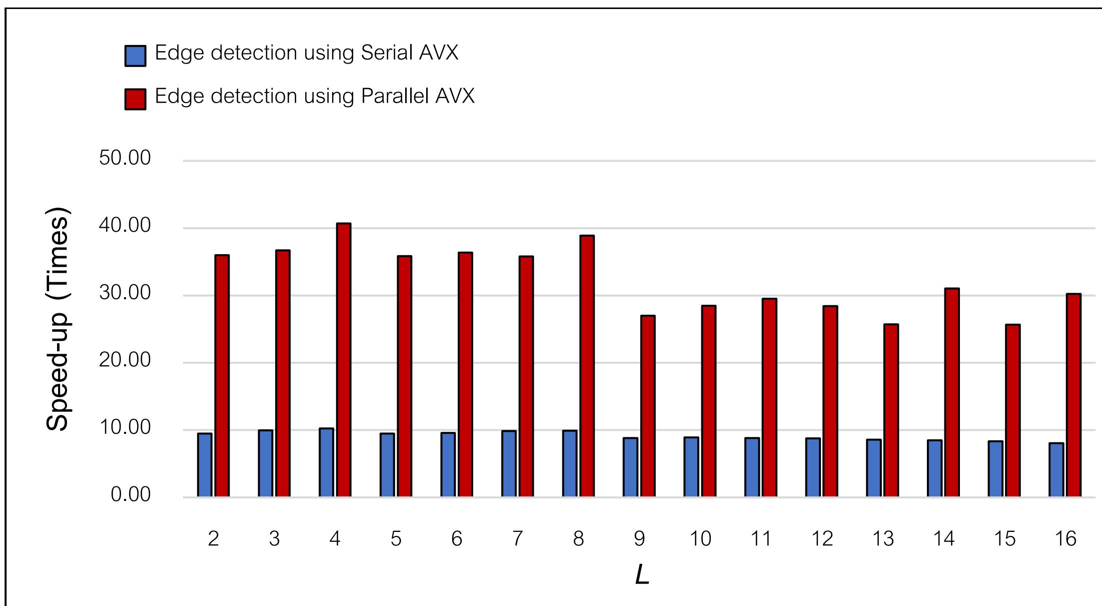
L	G_x (ms)		G_y (ms)		$\|G_x\|$+$\|G_y\|$ (ms)	
	serial AVX	parallel AVX	serial AVX	parallel AVX	serial AVX	parallel AVX
2	0.0875	0.0225	0.2227	0.0294	0.1587	0.0419
3	**0.0840**	0.0217	0.1970	0.0284	0.1511	0.0410
4	0.1488	**0.0194**	0.2220	**0.0263**	**0.1467**	**0.0370**
5	0.2401	0.0218	0.2585	0.0291	0.1589	0.0420
6	0.2352	0.0212	0.2279	0.0283	0.1575	0.0414
7	0.1841	0.0221	0.2464	0.0293	0.1524	0.0420
8	0.1180	0.0208	**0.1293**	0.0277	0.1517	0.0387
9	0.1975	0.0227	0.2073	0.0288	0.1704	0.0558
10	0.1811	0.0302	0.2314	0.0358	0.1690	0.0529
11	0.1451	0.0291	0.1936	0.0344	0.1710	0.0510
12	0.1722	0.0305	0.1943	0.0355	0.1720	0.0530
13	0.1695	0.0343	0.1864	0.0395	0.1757	0.0586
14	0.1680	0.0238	0.1535	0.0304	0.1773	0.0485
15	0.1560	0.0307	0.1800	0.0372	0.1809	0.0587
16	0.1410	0.0228	0.1598	0.0303	0.1867	0.0498

จากระยะเวลาการประมวลผลในตารางที่ 4.4 จะเห็นได้ว่า การหา Gradient ในแนวนอน (G_x) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==3 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==4 และสำหรับการหา Gradient ในแนวตั้ง (G_y) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==8 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==4 เช่นเดียวกัน สำหรับการตรวจจับหาขอบภาพนั้น วิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==4 และวิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบขนานค่า L ที่ดีที่สุด คือ L==4



รูปที่ 4.5 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 1280x1024 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.5 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนที่ดีที่สุดโดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 5.20 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 22.57 เท่า โดยค่าเฉลี่ยของการหา Gradient ในแนวนอนแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 2.94 และ 18.08 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวนอนด้วยไลบรารีมาตรฐานของ OpenCV และสำหรับการหา Gradient ในแนวตั้งที่ดีที่สุดด้วยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานมีความเร็วเพิ่มขึ้น 3.56 และ 17.54 เท่าตามลำดับ โดยค่าเฉลี่ยอยู่ที่ 14.90 และ 9.15 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวตั้งด้วยไลบรารี OpenCV



รูปที่ 4.6 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 1280x1024 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.6 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการตรวจจับขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 10.26 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 40.70 เท่า โดยค่าเฉลี่ยของการหาขอบภาพแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 9.15 และ 32.42 เท่าตามลำดับ เมื่อเทียบกับการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐานของ OpenCV

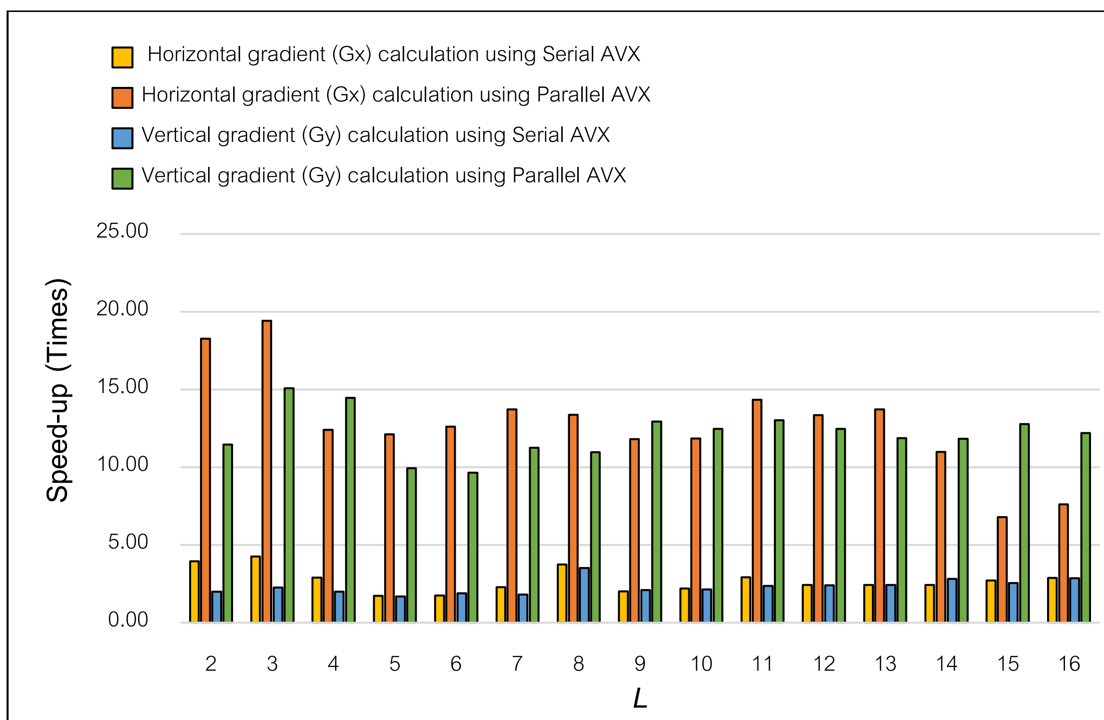
4.1.4 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 1920x1080 พิกเซล ที่ L ขนาดต่าง ๆ

ผลการทดสอบหาระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (G_x) และหาค่า Gradient ในแกนแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 1920x1080 พิกเซล ด้วยวิธีการที่นำเสนอ ดังแสดงในตารางที่ 4.5

ตารางที่ 4.5 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน ($\boldsymbol{G_x}$) และหาค่า Gradient ในแกนแนวตั้ง ($\boldsymbol{G_y}$) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 1920x1080 พิกเซล ด้วยวิธีการที่นำเสนอ

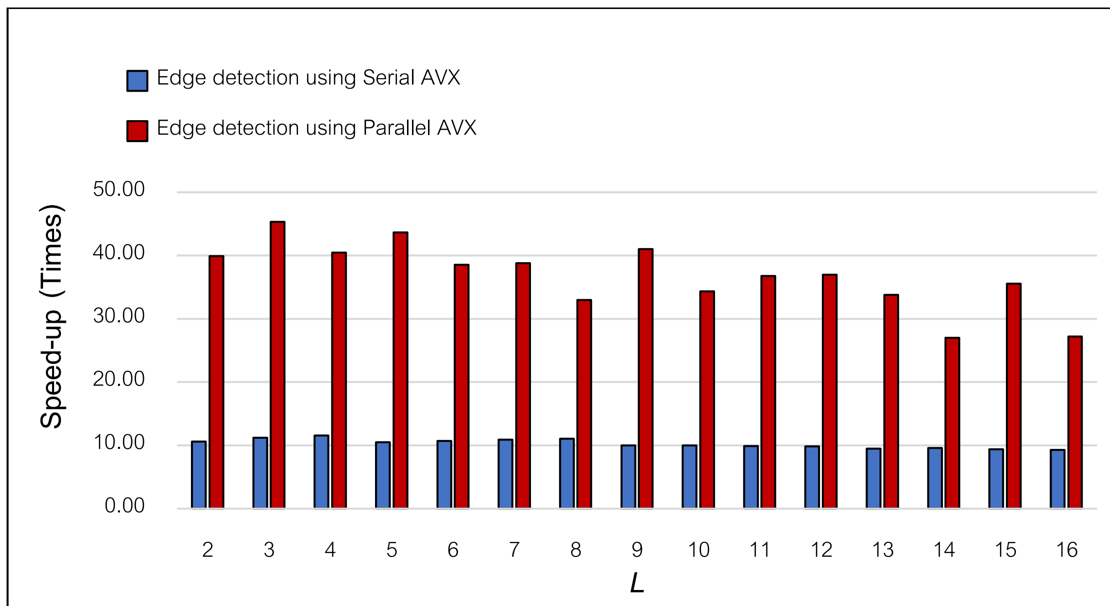
L	G_x (ms)		G_y (ms)		$\|G_x\|$+$\|G_y\|$ (ms)	
	serial AVX	parallel AVX	serial AVX	parallel AVX	serial AVX	parallel AVX
2	0.1643	0.0355	0.3570	0.0620	0.2505	0.0667
3	**0.1523**	**0.0334**	0.3134	**0.0471**	0.2371	**0.0587**
4	0.2239	0.0523	0.3556	0.0492	**0.2300**	0.0658
5	0.3741	0.0536	0.4195	0.0715	0.2536	0.0609
6	0.3726	0.0515	0.3760	0.0737	0.2492	0.0691
7	0.2850	0.0473	0.3926	0.0632	0.2436	0.0686
8	0.1731	0.0486	**0.2026**	0.0649	0.2409	0.0807
9	0.3224	0.0550	0.3375	0.0549	0.2668	0.0649
10	0.2944	0.0548	0.3329	0.0570	0.2662	0.0775
11	0.2221	0.0453	0.2993	0.0546	0.2685	0.0724
12	0.2670	0.0486	0.2955	0.0570	0.2706	0.0720
13	0.2675	0.0474	0.2935	0.0599	0.2802	0.0787
14	0.2670	0.0591	0.2514	0.0601	0.2777	0.0986
15	0.2389	0.0957	0.2795	0.0556	0.2831	0.0749
16	0.2263	0.0854	0.2485	0.0583	0.2867	0.0978

จากระยะเวลาการประมวลผลในตารางที่ 4.5 จะเห็นได้ว่า การหา Gradient ในแนวนอน (G_x) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==3 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==3 และสำหรับการหา Gradient ในแนวตั้ง (G_y) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==8 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==3 เช่นเดียวกันสำหรับการตรวจจับหาขอบภาพนั้น วิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==4 และวิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบขนานค่า L ที่ดีที่สุด คือ L==3



รูปที่ 4.7 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 1920x1080 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.7 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนที่ดีที่สุดโดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 4.26 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 19.42 เท่า โดยค่าเฉลี่ยของการหา Gradient ในแนวนอนแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 2.71 และ 12.82 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวนอนด้วยไลบรารีมาตรฐานของ OpenCV และสำหรับการหา Gradient ในแนวตั้งที่ดีที่สุดด้วยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานมีความเร็วเพิ่มขึ้น 3.51 และ 15.08 เท่าตามลำดับ โดยค่าเฉลี่ยอยู่ที่ 2.32 และ 12.16 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวตั้งด้วยไลบรารี OpenCV



รูปที่ 4.8 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 1920x1080 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.8 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการตรวจจับขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 11.57 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 45.34 เท่า โดยค่าเฉลี่ยของการหาขอบภาพแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 10.27 และ 36.83 เท่าตามลำดับ เมื่อเทียบกับการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐานของ OpenCV

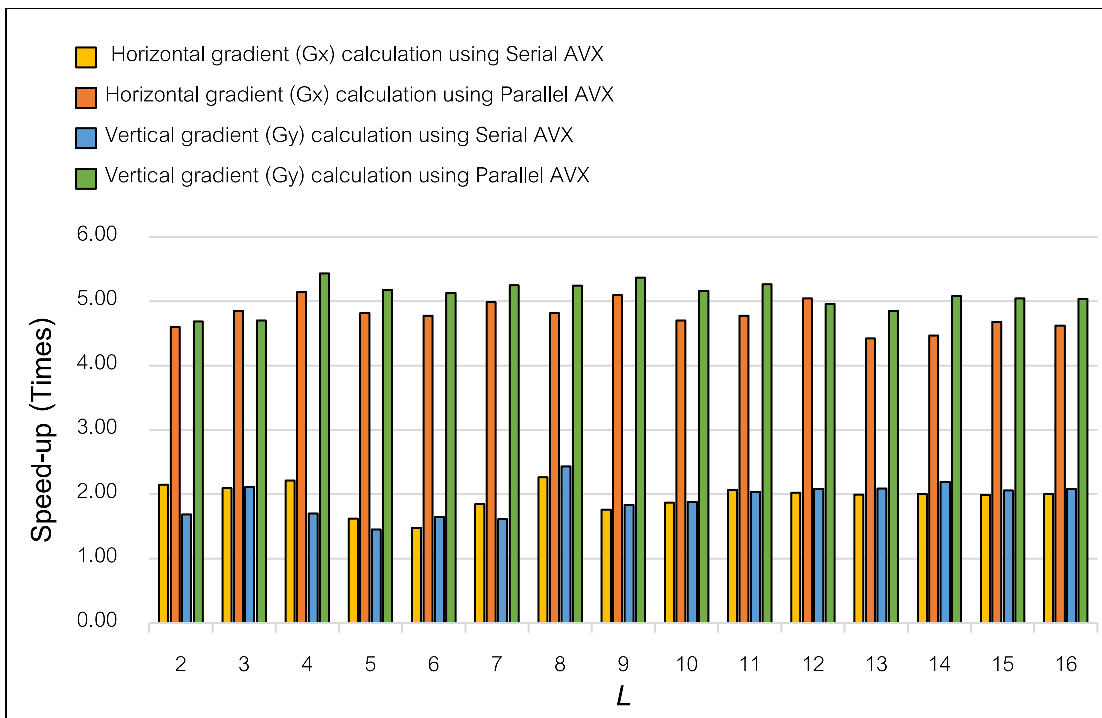
4.1.5 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 2560x1440 พิกเซล ที่ L ขนาดต่าง ๆ

ผลการทดสอบหาระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (G_x) และหาค่า Gradient ในแกนแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 2560x1440 พิกเซล ด้วยวิธีการที่นำเสนอ ดังแสดงในตารางที่ 4.6

ตารางที่ 4.6 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน ($\boldsymbol{G_x}$) และหาค่า Gradient ในแกนแนวตั้ง ($\boldsymbol{G_y}$) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 2560x1440 พิกเซล ด้วยวิธีการที่นำเสนอ

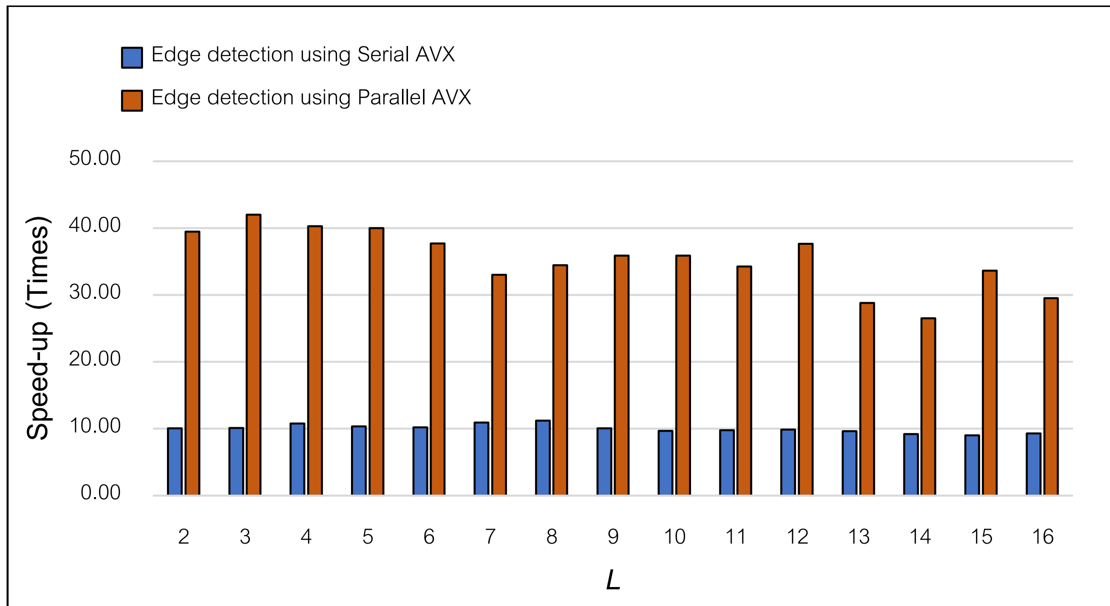
L	G_x (ms)		G_y (ms)		$\lvert G_x \rvert$+$\lvert G_y \rvert$ (ms)	
	serial AVX	parallel AVX	serial AVX	parallel AVX	serial AVX	parallel AVX
2	0.5495	0.2565	0.7710	0.2779	0.4888	0.1247
3	0.5635	0.2434	0.6152	0.2769	0.4861	**0.1172**
4	0.5329	**0.2296**	0.7641	**0.2397**	0.4573	0.1222
5	0.7269	0.2453	0.8956	0.2514	0.4765	0.1230
6	0.7998	0.2474	0.7906	0.2539	0.4832	0.1305
7	0.6392	0.2370	0.8079	0.2480	0.4504	0.1491
8	**0.5210**	0.2451	**0.5347**	0.2483	**0.4388**	0.1428
9	0.6711	0.2319	0.7079	0.2425	0.4881	0.1371
10	0.6306	0.2513	0.6920	0.2524	0.5071	0.1371
11	0.5711	0.2472	0.6373	0.2474	0.5024	0.1438
12	0.5832	0.2341	0.6241	0.2623	0.4988	0.1307
13	0.5917	0.2669	0.6234	0.2685	0.5117	0.1709
14	0.5890	0.2644	0.5938	0.2563	0.5337	0.1857
15	0.5933	0.2523	0.6316	0.2581	0.5474	0.1463
16	0.5881	0.2555	0.6262	0.2584	0.5282	0.1667

จากระยะเวลาการประมวลผลในตารางที่ 4.6 จะเห็นได้ว่า การหา Gradient ในแนวนอน (G_x) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==8 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==4 และสำหรับการหา Gradient ในแนวตั้ง (G_y) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==8 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==4 เช่นเดียวกัน สำหรับการตรวจจับหาขอบภาพนั้น วิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==8 และวิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบขนานค่า L ที่ดีที่สุด คือ L==3



รูปที่ 4.9 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 2560x1440 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.9 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนที่ดีที่สุดโดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 2.27 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 5.15 เท่า โดยค่าเฉลี่ยของการหา Gradient ในแนวนอนแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 1.96 และ 4.79 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวนอนด้วยไลบรารีมาตรฐานของ OpenCV และสำหรับการหา Gradient ในแนวตั้งที่ดีที่สุดด้วยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานมีความเร็วเพิ่มขึ้น 2.43 และ 5.43 เท่าตามลำดับ โดยค่าเฉลี่ยอยู่ที่ 1.93 และ 5.02 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวตั้งด้วยไลบรารี OpenCV



รูปที่ 4.10 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 2560x1440 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.10 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการตรวจจับขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 11.22 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 42.01 เท่า โดยค่าเฉลี่ยของการหาขอบภาพแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 10.02 และ 35.28 เท่าตามลำดับ เมื่อเทียบกับการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐานของ OpenCV

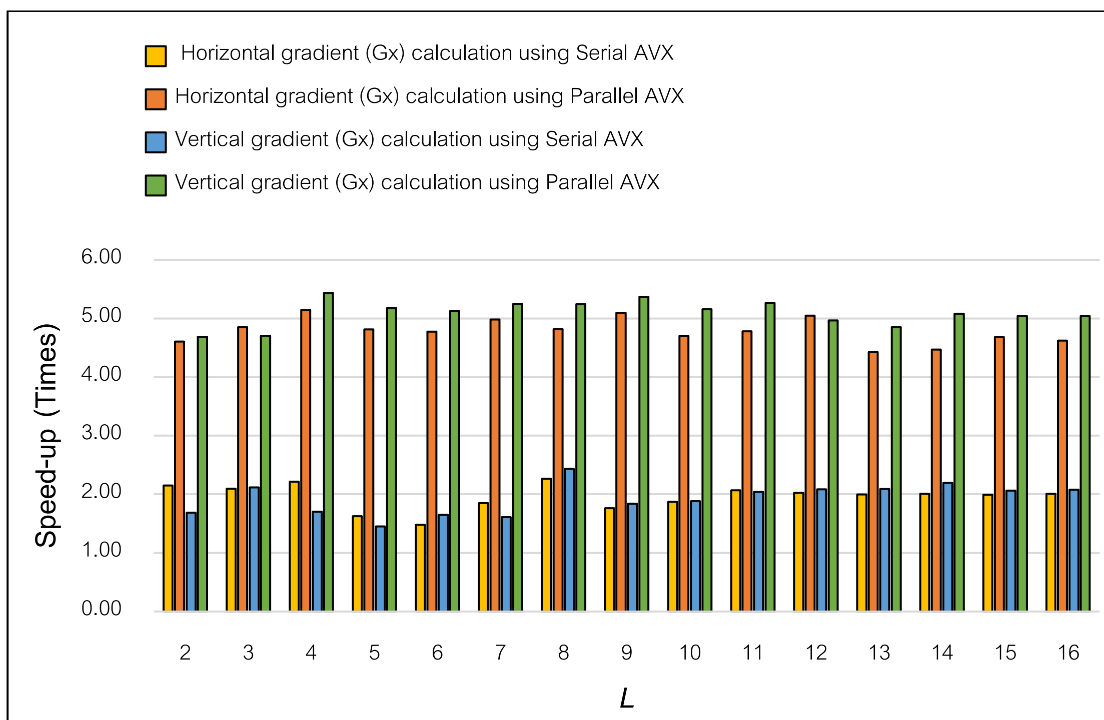
4.1.6 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 3872x2160 พิกเซล ที่ L ขนาดต่าง ๆ

ผลการทดสอบหาระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (G_x) และหาค่า Gradient ในแกนแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 3872x2160 พิกเซล ด้วยวิธีการที่นำเสนอ ดังแสดงในตารางที่ 4.7

ตารางที่ 4.7 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน ($\boldsymbol{G_x}$) และหาค่า Gradient ในแกนแนวตั้ง ($\boldsymbol{G_y}$) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 3872x2160 พิกเซล ด้วยวิธีการที่นำเสนอ

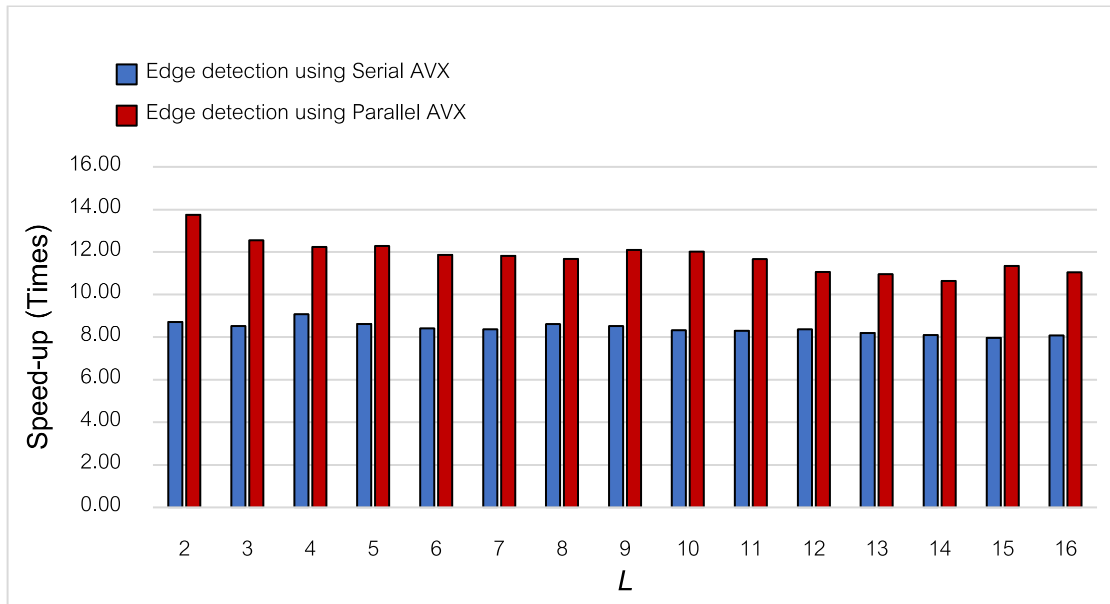
L	G_x (ms)		G_y (ms)		$\lvert G_x \rvert$+$\lvert G_y \rvert$ (ms)	
	serial AVX	parallel AVX	serial AVX	parallel AVX	serial AVX	parallel AVX
2	1.8332	**1.9360**	2.4208	**1.7358**	1.2975	**0.8213**
3	1.8029	2.0841	2.3072	1.9285	1.3259	0.9001
4	**1.7694**	2.0594	2.1483	2.0573	**1.2464**	0.9234
5	1.9313	2.0408	2.1757	1.9235	1.3107	0.9200
6	2.0134	2.0547	2.0537	2.0583	1.3445	0.9521
7	1.8319	2.1791	2.0736	2.1261	1.3509	0.9550
8	1.7957	2.2654	1.8865	2.3116	1.3128	0.9672
9	1.8964	2.1524	1.9373	2.1856	1.3271	0.9346
10	1.8221	2.2220	1.9286	2.2511	1.3571	0.9394
11	1.8295	2.3132	1.8790	2.2658	1.3601	0.9696
12	1.8508	2.2598	1.8768	2.3495	1.3510	1.0211
13	1.8432	2.3590	**1.8660**	2.3372	1.3775	1.0310
14	1.8262	2.3326	1.8905	2.4200	1.3957	1.0628
15	1.8508	2.3388	1.8661	2.4231	1.4178	0.9964
16	1.9200	2.4299	1.9017	2.4126	1.3975	1.0231

จากระยะเวลาการประมวลผลในตารางที่ 4.7 จะเห็นได้ว่า การหา Gradient ในแนวนอน (G_x) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==4 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==2 และสำหรับการหา Gradient ในแนวตั้ง (G_y) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==13 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==2 เช่นเดียวกันสำหรับการตรวจจับหาขอบภาพนั้น วิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==4 และวิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบขนานค่า L ที่ดีที่สุด คือ L==2



รูปที่ 4.11 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 3872x2160 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.11 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนที่ดีที่สุดโดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 1.90 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 1.74 เท่า โดยค่าเฉลี่ยของการหา Gradient ในแนวนอนแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 1.82 และ 1.53 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวนอนด้วยไลบรารีมาตรฐานของ OpenCV และสำหรับการหา Gradient ในแนวตั้งที่ดีที่สุดด้วยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานมีความเร็วเพิ่มขึ้น 1.72 และ 1.85 เท่าตามลำดับ โดยค่าเฉลี่ยอยู่ที่ 1.60 และ 1.48 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวตั้งด้วยไลบรารี OpenCV



รูปที่ 4.12 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 3872x2160 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.12 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการตรวจจับขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 9.07 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 13.76 เท่า โดยค่าเฉลี่ยของการหาขอบภาพแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 8.41 และ 11.80 เท่าตามลำดับ เมื่อเทียบกับการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐานของ OpenCV

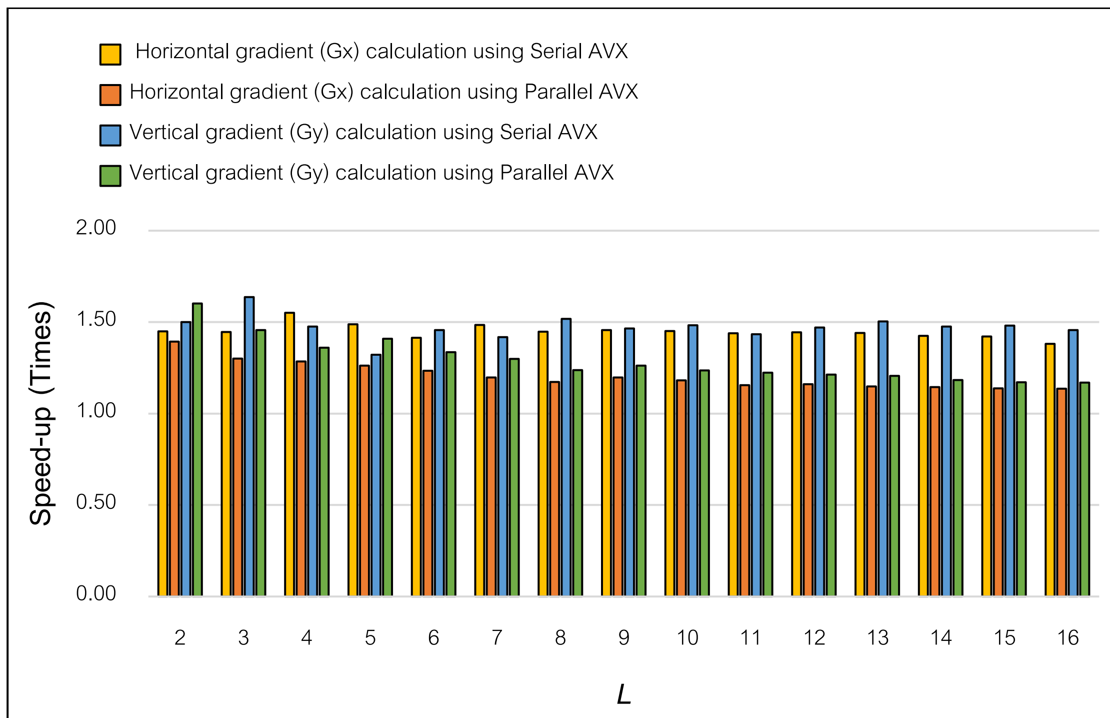
4.1.7 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 4928x3264 พิกเซล ที่ L ขนาดต่าง ๆ

ผลการทดสอบหาระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (G_x) และหาค่า Gradient ในแกนแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 4928x3264 พิกเซล ด้วยวิธีการที่นำเสนอ ดังแสดงในตารางที่ 4.8

ตารางที่ 4.8 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน ($\boldsymbol{G_x}$) และหาค่า Gradient ในแกนแนวตั้ง ($\boldsymbol{G_y}$) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 4928x3264 พิกเซล ด้วยวิธีการที่นำเสนอ

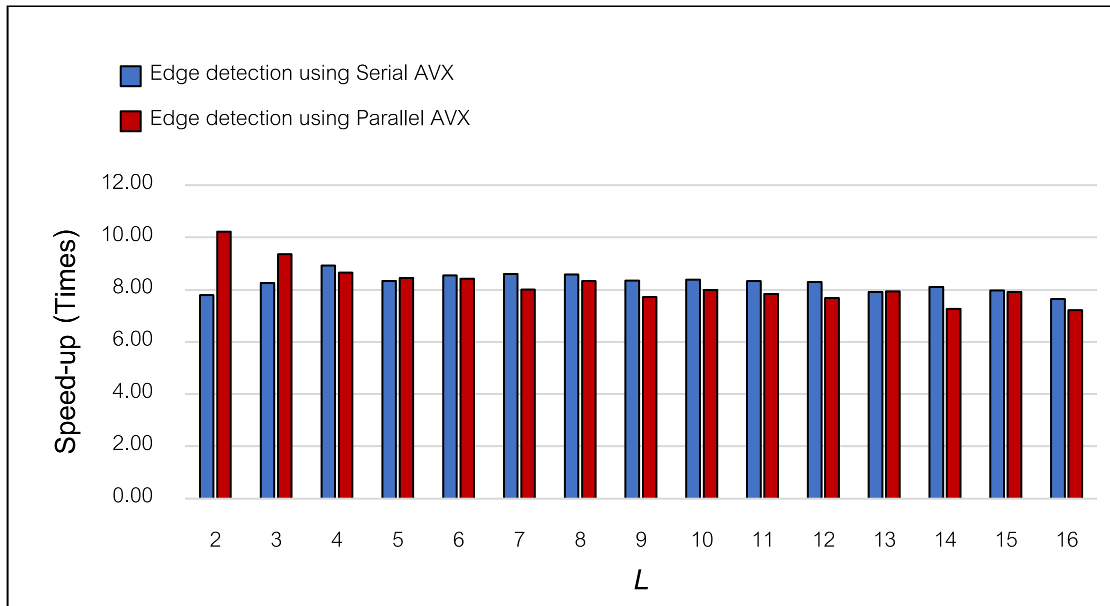
L	G_x (ms)		G_y (ms)		$\lvert G_x\rvert$+$\lvert G_y\rvert$ (ms)	
	serial AVX	parallel AVX	serial AVX	parallel AVX	serial AVX	parallel AVX
2	3.8709	**4.0276**	3.9592	**3.7092**	2.8012	**2.1360**
3	3.8815	4.3129	**3.6304**	4.0779	2.6445	2.3326
4	**3.6190**	4.3705	4.0248	4.3695	**2.4444**	2.5222
5	3.7714	4.4497	4.4957	4.2160	2.6171	2.5853
6	3.9699	4.5449	4.0791	4.4473	2.5538	2.5920
7	3.7800	4.6874	4.1890	4.5703	2.5348	2.7264
8	3.8759	4.7847	3.9138	4.8006	2.5422	2.6219
9	3.8552	4.6910	4.0530	4.7069	2.6136	2.8303
10	3.8670	4.7531	4.0052	4.8071	2.6010	2.7306
11	3.9024	4.8611	4.1456	4.8523	2.6198	2.7840
12	3.8886	4.8354	4.0376	4.8934	2.6347	2.8449
13	3.8951	4.8914	3.9496	4.9267	2.7593	2.7494
14	3.9375	4.8996	4.0256	5.0192	2.6937	3.0009
15	3.9503	4.9321	4.0133	5.0759	2.7404	2.7615
16	4.0625	4.9415	4.0807	5.0794	2.8566	3.0291

จากระยะเวลาการประมวลผลในตารางที่ 4.8 จะเห็นได้ว่า การหา Gradient ในแนวนอน (G_x) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==4 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==2 และสำหรับการหา Gradient ในแนวตั้ง (G_y) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==3 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==2 เช่นเดียวกันสำหรับการตรวจจับหาขอบภาพนั้น วิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==4 และวิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบขนานค่า L ที่ดีที่สุด คือ L==2



รูปที่ 4.13 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 4928x3264 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.13 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนที่ดีที่สุดโดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 1.55 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 1.39 เท่า โดยค่าเฉลี่ยของการหา Gradient ในแนวนอนแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 1.45 และ 1.21 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวนอนด้วยไลบรารีมาตรฐานของ OpenCV และสำหรับการหา Gradient ในแนวตั้งที่ดีที่สุดด้วยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานมีความเร็วเพิ่มขึ้น 1.64 และ 1.60 เท่าตามลำดับ โดยค่าเฉลี่ยอยู่ที่ 1.47 และ 1.29 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวตั้งด้วยไลบรารี OpenCV



รูปที่ 4.14 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 4928x3264 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.14 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการตรวจจับขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 8.93 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 10.22 เท่า โดยค่าเฉลี่ยของการหาขอบภาพแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 8.27 และ 8.20 เท่าตามลำดับ เมื่อเทียบกับการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐานของ OpenCV

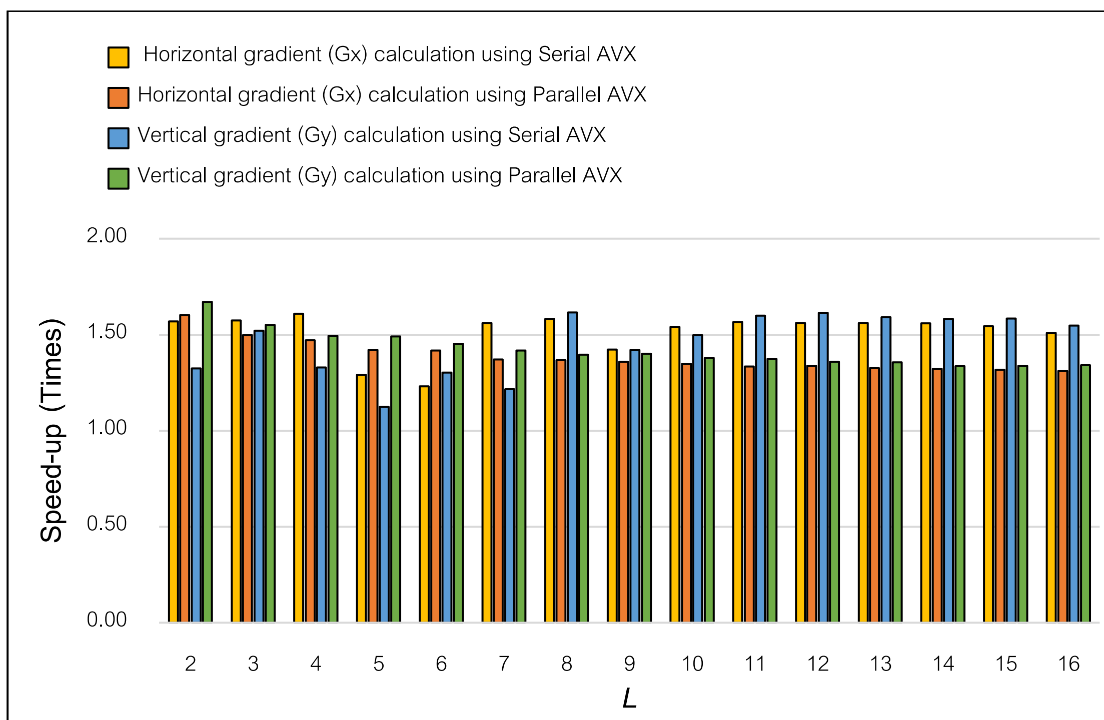
4.1.8 ผลการทดสอบหา Gradient Magnitude และตรวจจับหาขอบภาพของรูปภาพขนาด 7680x4320 พิกเซล ที่ L ขนาดต่าง ๆ

ผลการทดสอบหาระยะเวลาในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน (G_x) และหาค่า Gradient ในแกนแนวตั้ง (G_y) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 7680x4320 พิกเซล ด้วยวิธีการที่นำเสนอ ดังแสดงในตารางที่ 4.9

ตารางที่ 4.9 ระยะเวลาเฉลี่ย (ms) ในการคำนวณเพื่อการหาค่า Gradient ในแกนแนวนอน ($\boldsymbol{G_x}$) และหาค่า Gradient ในแกนแนวตั้ง ($\boldsymbol{G_y}$) และระยะเวลาในการคำนวณหาขอบภาพ ของภาพขนาด 7680x4320 พิกเซล ด้วยวิธีการที่นำเสนอ

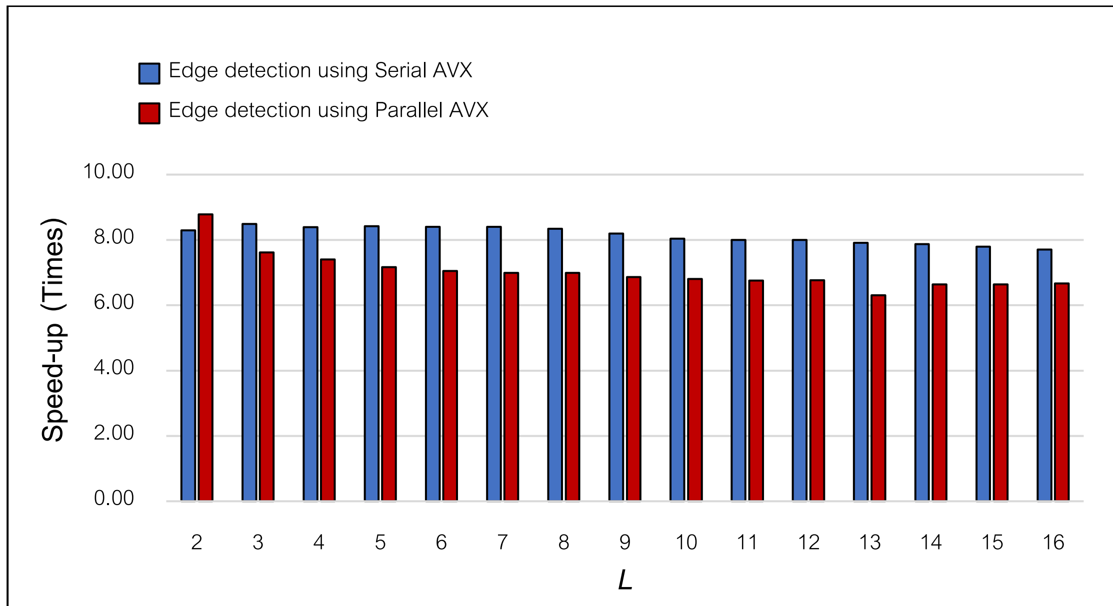
L	G_x (ms)		G_y (ms)		$\|G_x\|+\|G_y\|$ (ms)	
	serial AVX	parallel AVX	serial AVX	parallel AVX	serial AVX	parallel AVX
2	8.6993	**8.5192**	10.6542	**8.4430**	5.3920	**5.0881**
3	8.6720	9.1184	9.2745	9.0982	**5.2682**	5.8661
4	**8.4829**	9.2820	10.6192	9.4430	5.3311	6.0432
5	10.5729	9.6140	12.5414	9.4675	5.3130	6.2378
6	11.0979	9.6331	10.8294	9.7148	5.3215	6.3402
7	8.7458	9.9572	11.6056	9.9469	5.3220	6.3975
8	8.6259	9.9806	**8.7302**	10.1056	5.3618	6.3984
9	9.5956	10.0498	9.9268	10.0731	5.4563	6.5141
10	8.8573	10.1375	9.4248	10.2259	5.5589	6.5662
11	8.7170	10.2350	8.8181	10.2630	5.5885	6.6134
12	8.7516	10.2028	8.7379	10.3773	5.5914	6.6053
13	8.7499	10.2969	8.8706	10.4000	5.6495	7.0901
14	8.7541	10.3292	8.9160	10.5660	5.6816	6.7366
15	8.8466	10.3656	8.9018	10.5519	5.7349	6.7348
16	9.0444	10.4212	9.1164	10.5207	5.8042	6.7086

จากระยะเวลาการประมวลผลในตารางที่ 4.9 จะเห็นได้ว่า การหา Gradient ในแนวนอน (G_x) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==4 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==2 และสำหรับการหา Gradient ในแนวตั้ง (G_y) โดยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==8 และวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน ค่า L ที่ดีที่สุดคือ L==2 เช่นเดียวกันสำหรับการตรวจจับหาขอบภาพนั้น วิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรม ค่า L ที่ดีที่สุดคือ L==3 และวิธีการที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบขนานค่า L ที่ดีที่สุด คือ L==2



รูปที่ 4.15 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนและ Gradient ในแนวตั้ง ของภาพขนาด 7680x4320 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.15 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหา Gradient ในแนวนอนที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 1.61 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 1.60 เท่า โดยค่าเฉลี่ยของการหา Gradient ในแนวนอนแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 1.51 และ 1.39 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวนอนด้วยไลบรารีมาตรฐานของ OpenCV และสำหรับการหา Gradient ในแนวตั้งที่ดีที่สุดด้วยวิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานมีความเร็วเพิ่มขึ้น 1.62 และ 1.67 เท่าตามลำดับ โดยค่าเฉลี่ยอยู่ที่ 1.46 และ 1.42 เท่าตามลำดับ เมื่อเทียบกับการหา Gradient ในแนวตั้งด้วยไลบรารี OpenCV

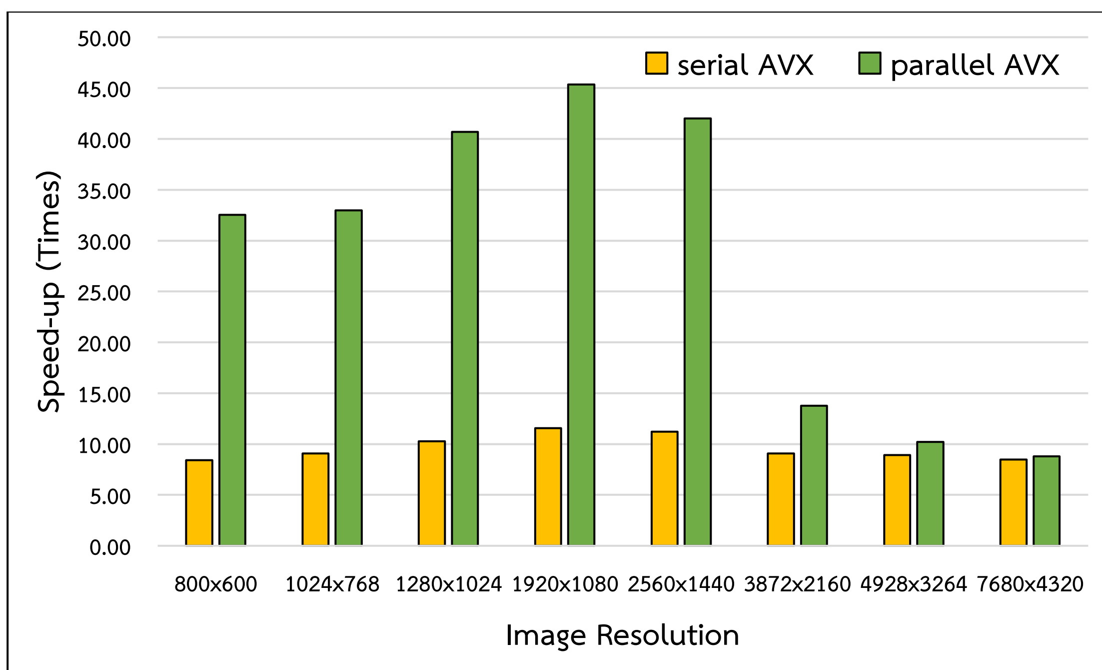


รูปที่ 4.16 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพ ของภาพขนาด 7680x4320 พิกเซล ด้วยวิธีการที่นำเสนอเปรียบเทียบการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV ที่ *L* ขนาดต่าง ๆ

จากรูปที่ 4.16 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการตรวจจับขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม มีความเร็วเพิ่มขึ้น 8.49 เท่า และวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน มีความเร็วเพิ่มขึ้น 8.79 เท่า โดยค่าเฉลี่ยของการหาขอบภาพแบบอนุกรม และแบบขนานด้วยวิธีที่นำเสนอ มีความเร็วเพิ่มขึ้น 8.15 และ 7.03 เท่าตามลำดับ เมื่อเทียบกับการหาขอบภาพโดยใช้วิธี Sobel ในไลบรารีมาตรฐานของ OpenCV

4.1.9 ผลการทดสอบการหาขอบภาพด้วยวิธีที่นำเสนอของภาพขนาดต่าง ๆ

ประสิทธิภาพในการหาขอบภาพโดยใช้วิธีการที่เสนอด้วยชุดคำสั่ง AVXเปรียบเทียบกับการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV กับรูปภาพขนาดต่าง ๆ



รูปที่ 4.17 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพด้วยวิธีการที่นำเสนอเปรียบเทียบกับการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV สำหรับภาพขนาดต่าง ๆ

จากรูปที่ 4.17 ประสิทธิภาพในการหาขอบภาพโดยใช้วิธีการที่เสนอเปรียบเทียบกับวิธี Sobel โดยใช้ไลบรารีมาตรฐานของ OpenCV กับรูปภาพขนาดต่าง ๆ โดยรูปภาพขนาด 1920x1080 พิกเซล วิธีการที่เสนอมีประสิทธิภาพในการหาขอบภาพได้ดีที่สุด ซึ่งสามารถลดระยะเวลาในการหาขอบภาพได้ 11.57 เท่า เมื่อใช้วิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม และ 45.34 เท่า สำหรับการใช้ชุดคำ AVX แบบขนาน ซึ่งจะเห็นได้ว่าผลการทดสอบความเร็วของภาพช่วงขนาดระหว่าง 800x600 พิกเซล ถึง 2560x1440 พิกเซล มีความเร็วของระยะเวลาการหาขอบภาพเพิ่มขึ้นโดยเฉลี่ย10.10 เท่า เมื่อประมวลผลด้วยวิธีโซเบลที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม และมีความเร็วของระยะเวลาการหาขอบภาพเพิ่มขึ้นโดยเฉลี่ย 38.71 เท่า เมื่อประมวลผลด้วยวิธีโซเบลที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน เปรียบเทียบกับการหาขอบภาพโดยใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV

และสำหรับภาพขนาด 7680×4320 พิกเซล การคำนวณขอบในวิธีที่เสนอสามารถลดระยะเวลาในการคำนวณหาขอบภาพได้น้อยที่สุดคือ 8.49 และ 8.79 เท่า เมื่อใช้วิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานตามลำดับ เปรียบเทียบกับการใช้ฟังก์ชัน Sobel ใน

ไลบรารีมาตรฐานของ OpenCV โดยผลจากทดสอบความเร็วของการประมวลผลภาพช่วงขนาด ระหว่าง 3872x2160 พิกเซล ถึง 7680x4320 พิกเซล มีความเร็วเฉลี่ยที่เพิ่มขึ้นของการหาขอบภาพ ด้วยวิธีโซเบลที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม 8.83 เท่า และแบบขนาน 10.92 เท่า เมื่อ เทียบกับการหาขอบภาพโดยใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV

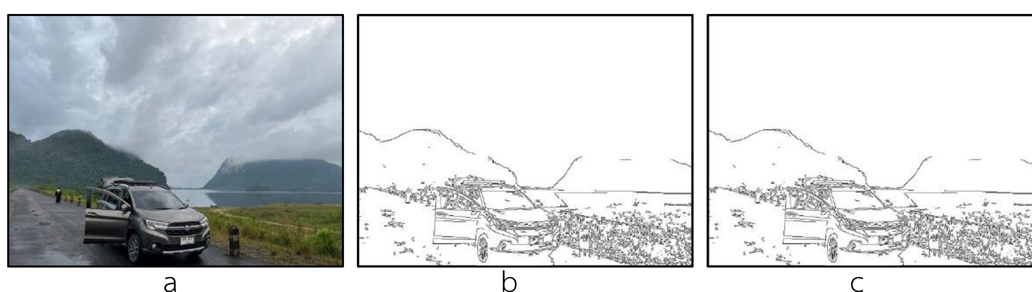
4.2 ผลการทดสอบการประยุกต์ใช้การเพิ่มประสิทธิภาพในการหาขอบภาพสำหรับแคนนี

ผู้วิจัยได้นำวิธีการหาขอบภาพโดยวิธีโซเบลที่นำเสนอด้วยชุดคำสั่ง AVX ไปประยุกต์ใช้กับ วิธีการหาขอบภาพด้วยวิธีแคนนีเพื่อเพิ่มประสิทธิภาพในการหาขอบภาพให้ดียิ่งขึ้น เนื่องจากวิธีการ หาขอบภาพด้วยวิธีแคนนีนั้นสามารถลดสัญญานรบได้ดีกว่าวิธีโซเบล [1] ดังแสดงเปรียบเทียบในภาพ ที่ 4.18 และ 4.19 เป็นการแสดงตัวอย่างผลภาพผลลัพธ์การหาขอบภาพโดยใช้วิธีโซเบล และแสดงผล ภาพผลลัพธ์การหาขอบภาพโดยใช้วิธีแคนนีของภาพขนาด 800x600 พิกเซล

รูปที่ 4.18 ภาพผลลัพธ์การหาขอบภาพโดยใช้วิธีโซเบลของภาพขนาด 800x600 พิกเซล

รูปที่ 4.19 ภาพผลลัพธ์การหาขอบภาพโดยใช้วิธีแคนนีของภาพขนาด 800x600 พิกเซล

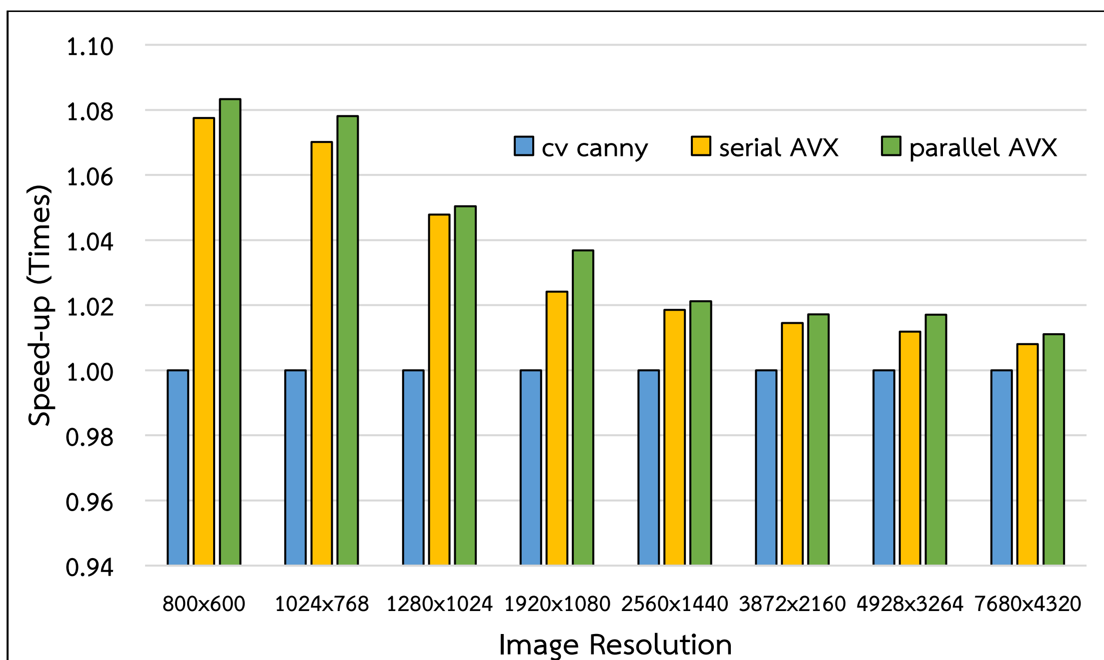
สำหรับในหัวข้อนี้ผู้วิจัยได้ทำการทดสอบหาประสิทธิภาพที่ใช้ในการประมวลผลเพื่อหาขอบภาพด้วยวิธีแคนนีที่นำเสนอโดยการใช้ชุดคำสั่ง AVX มาเปรียบเทียบกับการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV โดยฟังก์ชันและพารามิเตอร์ที่ใช้ใน OpenCV มีรูปแบบคือ Canny(img, canny_output, thresh, thresh * 2, 3); โดยผลลัพธ์ของการตรวจจับหาขอบภาพของวิธีการทั้งสองมีผลลัพธ์ตรงกันทุกตำแหน่งดังแสดงในรูปที่ 4.20 และผลการทดสอบคำนวณระยะเวลาที่ใช้ในการประมวลผลเพื่อหาขอบภาพมาจากการทดสอบด้วยโปรแกรมจำนวน 100 ครั้ง และนำมาคำนวณหาค่าเฉลี่ย ซึ่งผลการทดลองโดยการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV และผลทดลองของการหาขอบภาพด้วยวิธีแคนนีที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรมและแบบขนาน ดังแสดงในตารางที่ 4.10



รูปที่ 4.20 a: ภาพต้นฉบับ b: ผลลัพธ์ของการหาขอบภาพด้วยฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV c: ผลลัพธ์ของการหาขอบภาพด้วยวิธีแคนนีที่นำเสนอโดยการใช้ชุดคำสั่ง AVX

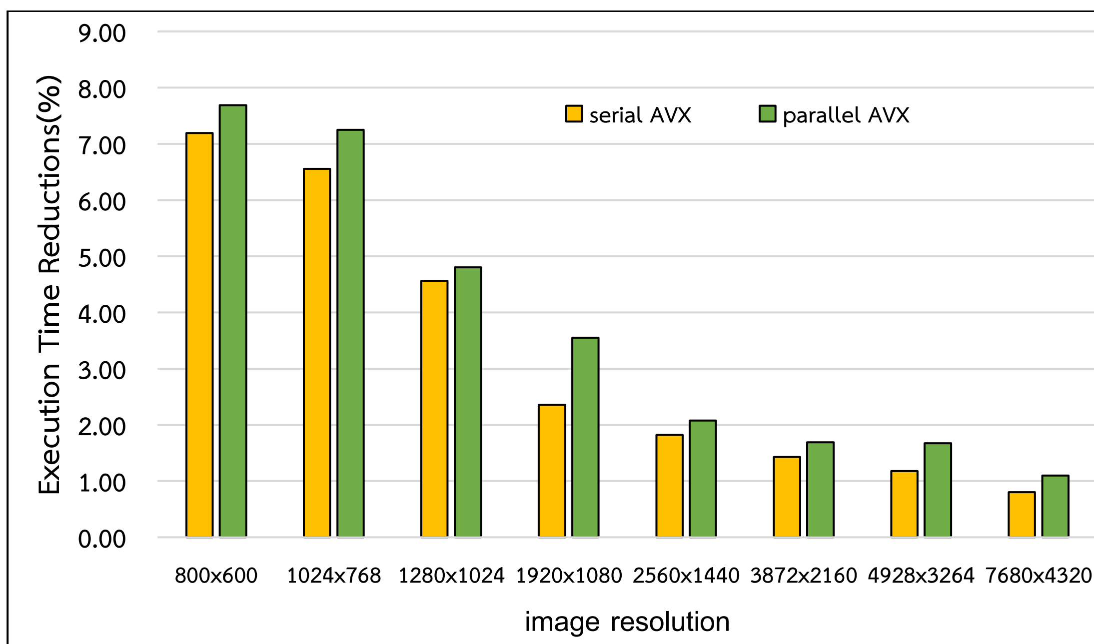
ตารางที่ 4.10 ระยะเวลาเฉลี่ย (ms) ที่ใช้ในการประมวลผลเพื่อหาขอบภาพโดยการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV และใช้วิธีการหาขอบภาพด้วยแคนนีที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรมและแบบขนาน

Image size (pixel)	CV Canny (ms)	Serial AVX (ms)	Parallel AVX (ms)
800x600	1.4925	1.3851	1.3777
1024x768	2.1153	1.9767	1.9620
1280x1024	3.4365	3.2796	3.2715
1920x1080	5.9958	5.8544	5.7827
2560x1440	10.6229	10.4296	10.4021
3872x2160	21.8441	21.5319	21.4746
4928x3264	36.1292	35.7046	35.5237
7680x4320	63.9060	63.3935	63.2033



รูปที่ 4.21 ความเร็วที่เพิ่มขึ้น (Speed-up) ของการหาขอบภาพด้วยวิธีการที่นำเสนอเปรียบเทียบกับการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐานของ OpenCV สำหรับภาพขนาดต่าง ๆ

รูปที่ 4.21 แสดงให้เห็นว่าการตรวจจับหาขอบภาพด้วยวิธีการ Canny เวอร์ชันที่ผู้วิจัยปรับปรุงนั้นเร็วกว่าเวอร์ชันดั้งเดิมที่อยู่ในไลบรารีมาตรฐานของ OpenCV เนื่องจากการตรวจจับขอบภาพของ Canny มีหลายขั้นตอน อย่างไรก็ตามประสิทธิภาพของการเพิ่มความเร็วของโค้ด Canny ที่แก้ไขแล้วจะไม่มากเท่ากับในกรณีของการตรวจจับหา Gradient magnitude เพียงอย่างเดียว จากภาพจะเห็นได้ว่าเมื่อขนาดภาพเป็น 800x600 จะมีความเร็วที่เพิ่มขึ้น (Speed-up) ที่ดีที่สุดโดยวิธีการหาขอบภาพด้วยแคนนีที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรมแบบขนานสามารถเพิ่มความเร็วในการประมวลผลได้ 1.07 และ 1.08 เท่าตามลำดับ และเมื่อขนาดภาพใหญ่ขึ้น เปอร์เซ็นต์ของการลดเวลาดำเนินการหาขอบภาพจะลดลง การตรวจจับขอบภาพด้วยวิธี Canny โดยใช้อัลกอริทึมที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมของเรานั้นเร็วกว่าไลบรารี OpenCV โดยเฉลี่ย 1.03 เท่า และแบบขนานของเราเร็วกว่าแบบอนุกรมเล็กน้อย และเร็วกว่าเวอร์ชันดั้งเดิมของไลบรารีมาตรฐานของ OpenCV โดยเฉลี่ย 1.04 เท่า



รูปที่ 4.22 เปอร์เซ็นต์ของความเร็วที่เพิ่มขึ้นของการหาขอบภาพด้วยวิธีการที่นำเสนอเปรียบเทียบกับการใช้ฟังชัน Canny ในไลบรารีมาตรฐานของ OpenCV สำหรับภาพขนาดต่าง ๆ

รูปที่ 4.22 แสดงเปอร์เซ็นต์ของความเร็วที่เพิ่มขึ้นของการหาขอบภาพด้วยวิธีการที่นำเสนอเปรียบเทียบกับการใช้ฟังชัน Canny ในไลบรารีมาตรฐานของ OpenCV สำหรับภาพขนาดต่าง ๆ จากภาพจะเห็นได้ว่าเมื่อขนาดภาพเป็น 800x600 จะได้อัตราเปอร์เซ็นต์ของการลดเวลาที่ใช้ในการหาขอบภาพที่ดีที่สุด โดยวิธีที่นำเสนอที่ทำการแก้ไข Canny โดยใช้ชุดคำสั่ง AVX แบบอนุกรมแบบขนานสามารถลดเวลาในการดำเนินการได้ 7.19 และ 7.69 เปอร์เซ็นต์ตามลำดับ และเมื่อขนาดภาพใหญ่ขึ้น เปอร์เซ็นต์ของการลดเวลาดำเนินการหาขอบภาพจะลดลง การตรวจจับขอบภาพด้วยวิธี Canny โดยใช้อัลกอริทึมที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมของเรานั้นเร็วกว่าไลบรารี OpenCV โดยเฉลี่ย 3.24 เปอร์เซ็นต์ และแบบขนานของเราเร็วกว่าแบบอนุกรมเล็กน้อย และเร็วกว่าเวอร์ชันดั้งเดิมของไลบรารีมาตรฐานของ OpenCV โดยเฉลี่ย 3.73 เปอร์เซ็นต์

4.3 การวิเคราะห์ผลการทดลอง

จากผลการทดลองการหา Gradient โดยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนาน สำหรับขนาด *L*==2 ถึง *L*==16 ประสิทธิภาพจะเพิ่มขึ้นตามขนาด *L* และเมื่อ *L* เพิ่มขึ้นจนถึงค่าที่เหมาะสมที่สุด ประสิทธิภาพจะลดลง จะเห็นได้ว่าเมื่อค่า *L* มีค่าน้อย จำนวนการลดลงของการดำเนินการทางคณิตศาสตร์จะลดลงได้น้อยและการแชร์ข้อมูลก็จะมีน้อย ซึ่งจะสามารถช่วยลดเวลาที่ใช้ในการประมวลผลได้น้อย เมื่อเปรียบเทียบกับไลบรารีมาตรฐานของ OpenCV และเมื่อถึงค่า *L* เพิ่มขึ้นถึงค่าที่เหมาะสม ประสิทธิภาพการลดระยะเวลาในการประมวลผลจะเหมาะสมที่สุด และการใช้ชุดคำสั่ง AVX ยังช่วยลดจำนวนครั้งในการเข้าถึงข้อมูลได้อีกด้วย ทำให้สามารถลดเวลาที่ใช้ในการหาขอบของภาพได้ แต่เนื่องจากข้อจำกัดของจำนวนรีจิสเตอร์ เมื่อถึงค่า *L* ที่เหมาะสม การ

ทำงานที่ดีที่สุดจึงเกิดขึ้น จากนั้นเมื่อเพิ่มจำนวน L ประสิทธิภาพการตรวจจับขอบภาพก็จะไม่สามารถทำได้อีก ซึ่งค่า L ที่เหมาะสมที่สุด สำหรับการทดลองบนทรัพยากรที่ใช้ในระบบนี้คือ L==4

วิธีการตรวจหาขอบภาพที่นำเสนอด้วยชุดคำสั่ง AVX ทำงานได้ดีกว่ากับภาพที่เล็กกว่า ตัวอย่างเช่น รูปภาพขนาด 800×600 พิกเซล มีความเร็วเพิ่มขึ้น 32.53 เท่า เมื่อทดสอบด้วยวิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนาน เมื่อเปรียบเทียบกับการตรวจจับขอบภาพด้วยวิธี Sobel ในไลบรารีมาตรฐานของ OpenCV และรูปภาพขนาดใหญ่ถึง 7680×4320 พิกเซล มีความเร็วเพิ่มขึ้นเพียง 8.79 เท่า สาเหตุหลักมาจากปัญหาคอขวดและข้อจำกัดในการเข้าถึงข้อมูลในหน่วยความจำ ซึ่งจะเห็นได้ว่าเมื่อภาพมีขนาดตั้งแต่ 3872x2160 พิกเซลขึ้นไป จะพบว่าขนาดของหน่วยความจำที่ใช้เก็บจะมีขนาดใหญ่กว่าขนาดของ cache ระดับ 3 ส่งผลให้ประสิทธิภาพลดลง ยกตัวอย่างเช่น เมื่อภาพมีขนาด 3872x2160 พิกเซล จำนวนความจำต้องต้องใช้เพื่อเก็บภาพ Gray scale จะเท่ากับ 8,363,520 ไบต์ ซึ่งจะเห็นว่ามีขนาดใหญ่กว่าขนาดของ cache ระดับ 3 ของซีพียูที่ใช้ ส่งผลให้ประสิทธิ์ภาพตกลง เนื่องจากต้องมีการสลับข้อมูลระหว่างหน่วยความจำหลักและ cache ระดับ 3 เพิ่มมากขึ้น โดยวิธีการตรวจจับขอบภาพด้วยวิธีโซเบลที่นำเสนอโดยใช้ชุดคำสั่ง AVX แบบอนุกรมและแบบขนานสามารถเพิ่มความเร็ว (Speed-up) ในการหาขอบภาพของภาพขนาดต่าง ๆ เพิ่มขึ้นโดยเฉลี่ย 9.62 และ 28.29 เท่าตามลำดับ เมื่อเทียบกับการใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV และช่วงของขนาดภาพที่มีความเร็วเพิ่มขึ้นที่ดีที่สุดคือ ช่วงของภาพขนาด 800x600 พิกเซล ถึง 2560x1440 พิกเซล มีความเร็วของระยะเวลาการหาขอบภาพเพิ่มขึ้นโดยเฉลี่ย 10.10 และ 38.71 เท่า เมื่อประมวลผลด้วยวิธีโซเบลที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนาด เปรียบเทียบกับการหาขอบภาพโดยใช้ฟังก์ชัน Sobel ในไลบรารีมาตรฐานของ OpenCV

สำหรับการตรวจจับขอบภาพโดยใช้วิธีการที่เสนอเมื่อเปรียบเทียบกับการใช้ฟังก์ชัน Canny ในไลบรารีมาตรฐาน OpenCV จะเห็นได้ว่าประสิทธิภาพการลดเวลาในการคำนวณบนภาพขนาดเล็กมีเปอร์เซ็นต์ดีกว่าภาพขนาดใหญ่ โดยเมื่อใช้วิธีที่นำเสนอด้วยชุดคำสั่ง AVX แบบอนุกรม รูปภาพ 800 × 600 พิกเซล จะเร็วขึ้น 7.19% และภาพ 7680 × 4320 พิกเซล เร็วขึ้น 0.08% และเมื่อใช้วิธีการที่นำเสนอด้วยชุดคำสั่ง AVX แบบขนานนั้น เปอร์เซ็นการลดลงของระยะเวลาในการประมวลผลเพื่อหาขอบภาพของรูปภาพขนาด 800 × 600 พิกเซล จะมีความเร็วเพิ่มขึ้น 7.69% และภาพขนาด 7680 × 4320 พิกเซลจะมีความเร็วเพิ่มขึ้นเพียง 1.10% สาเหตุหลักมาจากปัญหาคอขวดและข้อจำกัดการเข้าถึงข้อมูลในหน่วยความจำเช่นกัน

บทที่ 5
สรุปผลการวิจัยและข้อเสนอแนะ

ในบทนี้ได้กล่าวถึงการสรุปผลของการวิจัยที่ได้ดำเนินการสำหรับวิทยานิพนธ์ฉบับนี้ และข้อเสนอแนะต่าง ๆ เพื่อเป็นประโยชน์ต่อการทำวิจัยด้านการเพิ่มประสิทธิภาพในการหาขอบภาพ โดยได้นำเสนอหัวข้อดังนี้

5.1 สรุปผลการวิจัย

5.2 ข้อเสนอแนะ

5.1 สรุปผลการวิจัย

การตรวจจับขอบภาพเป็นพื้นฐานที่สำคัญของการการประมวลผลภาพ ซึ่งเป็นงานที่เสียเวลาในการประมวลผลมากเมื่อภาพมีขนาดใหญ่ ในปัจจุบันการแก้ปัญหาส่วนใหญ่มักนำฮาร์ดแวร์เข้ามาเป็นตัวช่วยในการเร่งเสริมในการเพิ่มความเร็วของการประมวลผลภาพ แต่เป็นวิธีการที่จะทำให้เกิดค่าใช้จ่ายเพิ่มขึ้นและสิ้นเปลืองพลังงานในการประมวลผล

ผู้วิจัยได้นำเสนอวิธีการเพิ่มความเร็วในการประมวลผลของการตรวจจับขอบภาพของวิธี Sobel โดยการตรวจสอบค่าของเคอร์เนลที่เป็น 0 หรือ 1 จะไม่นำค่าในเคอร์เนลตำแหน่งนั้นมาคำนวณ เพื่อการลงจำนวนครั้งในการดำเนินการลง และได้นำวิธีการคำนวณการหาขนาดของ Gradient magnitude โดยการประมวลผลข้อมูลภาพพร้อมกันหลายบรรทัดในแต่ละครั้งของการคำนวณ ซึ่งจะช่วยให้สามารถใช้ผลลัพธ์ของการคำนวณทางคณิตศาสตร์ของบรรทัดปัจจุบันร่วมกับบรรทัดอื่น ๆ ได้ และได้มีการนำชุดคำสั่ง AVX มาช่วยในการเพิ่มประสิทธิภาพในการประมวลผล ส่งผลให้การดำเนินการหนึ่งครั้งจะได้ผลลัพธ์ของข้อมูลที่ต้องการหลายจำนวน และได้นำเอา OpenMP มาช่วยในการกระจายข้อมูลที่จะนำมาคำนวณไปยังหน่วยประมวลผลต่าง ๆ ให้ช่วยกันทำงานในลักษณะของการทำงานแบบขนานกัน ประสิทธิภาพของวิธีการปรับปรุงคุณภาพของวิธีของโซเบลที่นำเสนอ เปรียบเทียบกับการใช้ฟังก์ชันโซเบลมาตรฐานในไลบรารีของ OpenCV มีความเร็วเพิ่มขึ้น (Speed Up) เฉลี่ย 28.29 เท่า

นอกจากนี้ผู้วิจัยยังได้นำเทคนิควิธีการเพิ่มความเร็วในการหาของภาพโดยวิธีโซเบลที่นำเสนอมาประยุกต์ใช้กับวิธีการหาขอบภาพด้วยวิธีแคนนี โดยนำอัลกอริทึมที่นำเสนอไปใช้ในขั้นตอนการหา Gradient magnitude ของการหาขอบภาพด้วยวิธีแคนนี เพื่อให้ความเร็วในภาพรวมของการหาขอบภาพของวิธีแคนนีสามารถทำได้เร็วยิ่งขึ้น จากผลการทดสอบพบว่าเมื่อทำงานร่วมกับอัลกอริทึมที่ผู้วิจัยเสนอแล้ว ประสิทธิภาพของการหาขอบภาพด้วยวิธีแคนนีมีความเร็วเฉลี่ยเพิ่มขึ้น 3.37 เปอร์เซ็นต์

จากผลการทดลอง สามารถสรุปได้ว่าการตรวจจับขอบภาพด้วยชุดคำสั่ง AVX แบบอนุกรมและแบบขนานที่นำเสนอนั้นเร็วกว่าฟังก์ชัน Sobel ที่จัดทำโดยไลบรารีมาตรฐานของ OpenCV และการใช้อัลกอริทึมที่นำเสนอในกระบวนการตรวจจับขอบภาพวิธี Canny ที่นำเสนอสามารถลดเวลาในการประมวลผลได้ เนื่องจากอัลกอริทึมที่นำเสนอจะลดปริมาณการโหลดของข้อมูลและการคำนวณ

ทางคณิตศาสตร์ได้ แนวทางที่นำเสนอนั้นคุ้มค่าเพราะต้องการเพียงการดัดแปลงซอฟต์แวร์โดยไม่ต้องใช้ตัวเร่งฮาร์ดแวร์ใด ๆ

5.2 ข้อเสนอแนะ

5.2.1 เนื่องจากวิธีโซเบลที่นำเสนอสามารถนำไปใช้ในขั้นตอนของการหา Gradient magnitude ของการหาขอบภาพ ของงานวิจัยอื่น ๆ ที่เกี่ยวข้องกับการประมวลผลภาพ โดยสามารถนำวิธีการที่นำเสนอไปประยุกต์ใช้เพื่อเพิ่มความเร็วในการประมวลผลภาพได้

5.2.2 แม้ว่าการทดลองที่ดำเนินการไปแล้วในวิทยานิพนธ์จะพบว่าค่า L ที่ดีที่สุดบนเครื่อง Core i5 มีค่าเท่ากับ 4 อย่างไรก็ดียังไม่มีการศึกษาว่าขนาดของ L ควรมีความสัมพันธ์อย่างไรกับตัวประมวลหรือคุณสมบัติของเครื่องที่ใช้ จึงควรมีการศึกษาเพิ่มเติมถึงวิธีที่นำมาศึกษาค่า L เพื่อให้ผู้ที่นำอัลกอริทึมที่นำเสนอไปใช้สามารถปรับค่าพารามิเตอร์ได้ง่ายยิ่งขึ้น

5.2.3 ในการทดลองได้เลือกขนาดของภาพเป็นขนาดที่เป็นที่นิยมใช้งานโดยทั่วไป อย่างไรก็ดียังมิได้มีการศึกษาว่า หากขนาดของภาพหรือสัดส่วนของภาพ มิได้เป็นขนาดดังกล่าวจะมีผลกระทบอย่างไรต่อประสิทธิภาพของการประมวลผล จึงควรมีการศึกษาผลกระทบจากกรณีดังกล่าวเพิ่มเติม

5.2.4 ในสภาพแวดล้อมของ OpenMP มี Directives ที่สามารถสั่งให้คอมไพเลอร์สร้างคำสั่งเครื่องแบบเวกเตอร์ได้ อย่างไรก็ดีในวิทยานิพนธ์ฉบับนี้ยังมิได้ทดสอบด้วยทางเลือกดังกล่าวว่ามีประสิทธิภาพมากน้อยเพียงใดเมื่อใช้กับอัลกอริทึมที่นำเสนอ จึงควรมีการศึกษาเพิ่มเติมในส่วนนี้

บรรณานุกรม

[1] G. N. Chaple, R. D. Daruwala and M. S. Gofane, "Comparisions of Robert, Prewitt, Sobel operator based edge detection methods for real time uses on FPGA," 2015 International Conference on Technologies for Sustainable Development (ICTSD), 2015, pp. 1-4, doi: 10.1109/ICTSD.2015.7095920.

[2] R. Menaka, R. Janarthanan, K. Deeba, FPGA implementation of low power and high speed image edge detection algorithm, Microproc. Microsyst. 75 (2020), 103053, https://doi.org/10.1016/j.micpro.2020.103053.

[3] S. Abed, Implementation of an edge detection algorithm using FPGA reconfigurable hardware, J. Eng. Res. 8 (1) (2020) 179–197.

[4] S. Taslimi, R. Faraji, A. Aghasi, H.R. Naji, Adaptive edge detection technique implemented on FPGA, Iran. J.Sci.Technol.-Trans. Electri. Eng. (2020), https://doi.org/10.1007/s40998-020-00333-5.

[5] N. Nausheen, A. Seal, P. Khanna, S. Halder, A FPGA based implementation of Sobel edge detection, Microprocess. Microsyst. 56 (2018) 84–91, https://doi.org/10.1016/j.micpro.2017.10.011.

[6] Jiang, B. (2018). Real-time multi-resolution edge detection with pattern analysis on graphics processing unit. Journal of Real-Time Image Processing, 14(2), 293-321, doi: 10.1007/ s11554-014-0450-x

[7] S. Abed, M.H. Ali, M Al-Shayeji, Enhanced GPU-based anti-noise hybrid edge detection method, Comput. Syst. Sci. Eng. 35 (1) (2020) 21–37, https://doi.org/10.32604/csse.2020.35.021.

[8] Bettaieb, N. Filali, T. Filali, H.B. Aissia, GPU acceleration of edge detection algorithm based on local variance and integral image: application to air bubbles boundaries extraction, Comput. Optics 43 (3) (2019) 446–454, https://doi.org/10.18287/2412-6179-2019-43-3-446-454.

[9] A. Jose, K. Deepa Merlin Dixon, N. Joseph, E. S. George and V. Anjitha, "Performance study of edge detection operators," 2014 International Conference on Embedded Systems (ICES), 2014, pp. 7-11, doi: 10.1109/EmbeddedSys.2014.6953040.

[10] H. Amiri and A. Shahbahrami, "High performance implementation of 2D convolution using Intel's advanced vector extensions," 2017 Artificial Intelligence and Signal Processing Conference (AISP), 2017, pp. 25-30, doi: 10.1109/AISP.2017.8324097.

[11] Y. Yan, J. Kemp, X. Tian, A. M. Malik and B. Chapman, "Performance and Power Characteristics of Matrix Multiplication Algorithms on Multicore and Shared Memory Machines," 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, 2012, pp. 626-632, doi: 10.1109/SC. Companion.2012.87.

[12] Hassan, S.A., Mahmoud, M.M., Hemeida, A.M., & Saber, M.A. (2018). Effective Implementation of Matrix-Vector Multiplication on Intel's AVX Multicore Processor. Computer Languages Systems & Structures, 51, 158-175, doi: 10.1016/j.cl.2017.06.003

[13] Liu, W.E., Wang, F., & Zhou, H.W. (2019). Parallel Seismic Modeling Based on OpenMP plus AVX and Optimization Strategy. Journal of Earth Science, 30(4), 843-848, doi: 10.1007/s12583-018-0831-3

[14] H. Amiri and A. Shahbahrami, (2020). SIMD programming using Intel vector extensions. Journal of Parallel and Distributed Computing, 135, 83-100, doi:10.1016/j.jpdc.2019.09.012

[15] Kenneth R. Castleman, (1996), “Digital Image Processing”, Englewood Cliffs, N.J. : Prentice Hall.

[16] W. Gao, X. Zhang, L. Yang, and H. Liu, "An improved Sobel edge detection," in Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on, 2010, pp. 67-71.

[17] S. Beeran Kutty, S. Saaidin, P. N. A. Megat Yunus and S. Abu Hassan, "Evaluation of canny and sobel operator for logo edge detection," 2014 International Symposium on Technology Management and Emerging Technologies, 2014, pp. 153-156, doi: 10.1109/ISTMET.2014.6936497.

[18] Chunxi Ma, Wenshuo Gao, Lei Yang and Zhonghui Liu, "An improved Sobel algorithm based on median filter," 2010 2nd International Conference on Mechanical and Electronics Engineering, 2010, pp. V1-88-V1-92, doi: 10.1109/ICMEE.2010.5558590.

[19] R. Maini and H. Aggarwal, "Study and comparison of various image edge detection techniques," International Journal of Image Processing (IJIP), vol. 3, pp. 1-11, 2009.

[20] J. Canny, "A Computational Approach to Edge Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, pp. 679-698, Nov. 1986, doi: 10.1109/TPAMI.1986.4767851.

[21] Canny, "A Computational Approach to Edge Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, pp.679-698, Nov. 1986, doi: 10.1109/TPAMI.1986. 4767851.

[22] M. A. Rahman, M. F. I. Amin and M. Hamada, "Edge Detection Technique by Histogram Processing with Canny Edge Detector," 2020 3rd IEEE International Conference on Knowledge Innovation and Invention (ICKII), 2020, pp. 128-131, doi: 10.1109/ ICKII 50300.2020. 9318922.

[23] X. Wang and J. -Q. Jin, "An Edge Detection Algorithm Based on Improved CANNY Operator," Seventh International Conference on Intelligent Systems Design and Applications (ISDA 2007), 2007, pp. 623-628, doi: 10.1109/ISDA.2007.6.

[24] L. Er-sen, Z. Shu-long, Z. Bao-shan, Z. Yong, X. Chao-gui and S. Li-hua, "An Adaptive Edge-Detection Method Based on the Canny Operator," 2009 International Conference on Environmental Science and Information Application Technology, 2009, pp. 465-469, doi: 10.1109/ESIAT.2009.49.

[25] M. Juneja and P. Sandhu, “Performance evaluation of edge detection Techniques for images in spatial domain” International Journal of Computer theory and Engineering, Vol. 1, No. 5, December, 2009 1793- 8201.

[26] N. Tepkasetkul and J. Wetweerapong, "Modifications of Canny Method for Image Edge Detection," in 20th National Graduate Research Conference (NGRC) at Khon Kaen University, 2019, pp. 299-308.

[27] Intel, Intel Intrinsics Guide, Retrived April 7, 2022, from https://www.intel.com/ content/www/us/en/docs/intrinsics-guide/index.html

[28] Blaise Barney, Introduction to Parallel Computing Tutorial, Retrived April 26, 2022, from https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

ภาคผนวก

ภาคผนวก ก.
ผลงานตีพิมพ์เผยแพร่จากวิทยานิพนธ์

1. T. Peng-o and P. Chaikan, "Optimization of Edge Detection using AVX Intrinsics on Multi-core Architectures," 2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), 2022, pp. 364-367, doi: 10.1109/ITC-CSCC55581.2022.9894947.
2. T. Peng-o and P. Chaikan, "High performance and energy efficient sobel edge detection", Microprocessors and Microsystems, Volume 87, 2021, 104368, ISSN 0141-9331, https://doi.org/10.1016/j.micpro.2021.104368.

1st CALL FOR PAPERS

ITC-CSCC 2022

The 37th International Technical Conference on Circuits /Systems, Computers and Communications (ITC-CSCC 2022)

July 5-8, 2022, Duangjitt Resort & Spa, Patong, Phuket, Thailand

With the great success of the International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC) as the world leading conference devoted to the advancement of high technologies in Circuits, Systems, Computers, and Communications, we would like to invite all the scholars and experts around the world to attend the 37th ITC-CSCC 2022 to be hosted in Phuket, Thailand.

Topics

The conference is open to researchers from all regions of the world. Participation from Asia Pacific region is particularly encouraged. Proposals for special sessions are welcome. Papers with original work in all aspects of Circuits, Systems, Computers, and Communications are invited. Topics include, but not limited to, the followings

Circuit and systems
- Analog Circuits
- Computer Aided Design
- Intelligent Transportation Systems & Technology
- Linear / Nonlinear Systems
- Medical Electronics & Circuits
- Modern Control
- Neural Networks
- Power Electronics & Circuits
- RF Circuits
- Semiconductor Devices & Technology
- Sensors & Related Circuits
- Verification & Testing
- VLSI Design

Communications
- Antenna & Wave Propagation
- Audio / Speech Signal Processing
- Circuits & Components for Communications
- IP Networks & QoS
- MIMO & Space-Time Codes
- Multimedia Communications
- Mobile & Wireless Communications
- Network Management & Design
- Optical Communications & Components
- Radar / Remote Sensing
- Communication Signal Processing
- Ubiquitous Networks
- UWB
- Visual Communications
- Wireless Sensor Networks
- Underwater Communications

Computers
- Artificial Intelligence
- Biocomputing
- Computer Systems & Applications
- Computer Vision
- Face Detection & Recognition
- Image Coding & Analysis
- Image Processing
- Internet Technology & Applications
- Motion Analysis
- Multimedia Service & Technology
- Object Extraction & Technology
- Security
- Watermarking
- Blockchain
- Data Analytics
- Internet of Things
- Virtual Reality

Submission of Papers

Prospective authors are invited to submit an original paper with 2 – 4 pages in length of PDF format written in English. Paper submission procedures are available at https://itc-cscc2022.org

Proceedings and Publications

All registered participants are provided with online conference proceedings. Upon requested, accepted papers will be published in IEEE Xplore. Moreover, authors of the accepted papers are encouraged to submit full-length manuscripts to IEIE JSTS (Korea), IEIE SPC (Korea), IEICE Transactions (Japan), or ECTI Transactions (Thailand). All the submissions need to follow the standard procedure of their publication and be published on regular issues.

Contact: secretary@itc-cscc2022.org , http://www.itc-cscc2022.org

Important Dates

Deadline of Manuscript Submission : April 1, 2022
Notification of Acceptance : May 14, 2022
Submission of Camera-Ready Paper : June 6, 2022

ECTI Association

IEIE
The Institute of Electronics and Information Engineers
ECTI
Association
EiC
Certificate of Presentation
This is to certify that
Thaufig Peng-o
presented a paper titled
Optimization of Edge Detection using AVX Intrinsics
on Multi-core Architectures
at the
ITC-CSCC 2022
The 37th International Technical Conference on Circuits/Systems, Computers, and Communications
July 5-8, 2022
Phuket, Thailand
Chanon Warisarn
King Mongut's Institute of Technology
Ladkrabang TPC Chair
OASYS
DATA ANALYTICS
TCEB

Optimization of Edge Detection using AVX Intrinsics on Multi-core Architectures

Thaufig Peng-o
Computer Engineering
Prince of Songkla University
Hadyai, Songkhla
6210120056@email.psu.ac.th

Panyayot Chaikan
Computer Engineering
Prince of Songkla University
Hadyai, Songkhla
panyayot.c@psu.ac.th

2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC) | 978-1-6654-8559-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ITC-CSCC55581.2022.9894947

***Abstract*—This paper presents the algorithm for augmenting the processing speed of Sobel and Canny edge detection. By reducing the number of arithmetic operations and data loads, the processing time is reduced. Our proposed method is purely based on software approach which does not require any accelerated hardware. In addition, the processing speed is further increased by utilizing the AVX intrinsics and OpenMP. Our proposed Sobel edge detection is on average 28.29 times faster than the Sobel function provided by the OpenCV library. When applied with the Canny edge detection, our algorithm can augment the speed of OpenCV's Canny edge detection by 3.73 percent.**

Keywords*— *Sobel, Canny, edge detection, AVX, OpenMP, multi-core, image processing.

I. Introduction

Edge detection is an important fundamental process of image processing and pattern recognition [1]. It is a time-consuming operation especially when the image size is large. Because of this reason, some real-time application implemented edge detection using hardware approach. For example, the utilization of FPGA [2-5], or the use of GPU[6-8] to augment the image processing speed. Although using this kind of accelerated hardware does help the speed augmentation, this approach requires higher cost and energy consumption.

There are many ways to obtain edges from the image, for example, the Sobel, Prewitt, and Roberts operators. Sobel is more utilized than the others. It can be found in both hardware and software implementations. It can be utilized in the more sophisticated edge detection such as Canny edge detection. In this paper, we propose an optimization method to increase a processing speed of Sobel edge detection by reducing the number of arithmetic operations and the amounts of memory loads. Our approach is based on software modification alone without the need of any accelerated hardware. The processing speed is further increased by utilizing the AVX intrinsics and OpenMP directive calls. We have tested our algorithm versus the functions provided by the OpenCV library. We found that our algorithm outperforms both the Sobel and Canny functions of the OpenCV library.

This paper is organized as follows: section II details background and related work, section III describes our proposed method, section IV presents the experimental results. The results are discussed and concluded in section V.

II. Background and Related Work

Sobel edge detection is a widely used technique for locating the edges of images. Its noise reduction is better than that of both Prewitt and Roberts operators [9-10]. Its convolution kernels are shown in Figure 1 [11-13].

The gradient in the horizontal and vertical directions, denoted Gx and Gy respectively, are obtained by convolving the image with the horizontal and vertical masks of Figure 1. The gradient magnitude, denoted ∇f, is approximated by

$$\nabla f = |Gx| + |Gy| \quad (1)$$

M_x

1	0	-1
2	0	-2
1	0	-1

M_y

1	2	1
0	0	0
-1	-2	-1

Figure 1. Convolution Masks for the Sobel operator: Mx and My are its horizontal and vertical kernels respectively.

Canny is a more sophisticated edge detection. It provides low error edges and achieves very high quality of image segmentation [14]. It requires multi-step process as shown in Figure 2 [15]. These four main steps include: 1) noise reduction; 2) gradient magnitude and direction detection; 3) non maximum suppression; and 4) tracing edges through the image hysteresis thresholding.

Figure 2. Steps of Canny edge detection.

III. Proposed Method

Most Sobel edge implementations are based on convolution method between the image and the Sobel kernel, as shown in Figure 3. This approach requires 17 arithmetic calculations: 9 multiplication operations and 8 addition operations. It is not efficient because some elements of the Sobel kernel are zeros, so it is a waste of time to read them and multiply them with the image pixels. Figures 4 and 5 eliminate this drawback away by reducing the number of image pixel read down to 6 pixels for each gradient detection. As a result, the number of multiplications and additions are reduced to 1 and 8 respectively. This method not only reduce the amounts of arithmetic operations but also reduce the number of data load from main memory. We further improve the efficiency of image gradient calculation by calculating the gradient more than one line at a time, as shown in Figure 6.

Authorized licensed use limited to: Prince of Songkla University provided by UniNet. Downloaded on October 16,2022 at 15:53:36 UTC from IEEE Xplore. Restrictions apply.

Gradient G = Kernel * Image

Kernel

1	2	3
4	5	6
7	8	9

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8

G = (1xP0) + (2xP1) + (3xP2) + (4xP3) + (5xP4) + (6xP5) + (7xP6) + (8xP7) + (9xP8)

Figure 3. Traditional method for estimating the gradient magnitude of an image.

Gradient X Gx = Kernel X * Image

Kernel X

1	0	-1
2	0	-2
1	0	-1

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8

Gx = (P0 – P2) + 2(P3 – P5) + (P6 – P8)

Figure 4. Proposed method for obtaining the horizontal gradient of an image.

Gradient Y Gy = Kernel Y * Image

Kernel Y

1	2	1
0	0	0
-1	-2	-1

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8

Gy = (P0 – P6) + 2(P1 – P7) + (P2 – P8)

Figure 5. Proposed method for obtaining the vertical gradient of an image.

Figure 6 shows our proposed algorithm. It calculates the gradient magnitude of the image more than one line at a time. As a result, some arithmetic operation between different gradient lines can be shared together. For example, if the number of lines is 4, then the calculation of P6-P8 which is required by the Gx1, Gx2, and Gx3 derivations can be performed only once. In addition, the derivation of GX2, Gx3, and Gx4 can share the calculation of P9-P11. The derivation of Gx1 and Gx2 can share the calculation of P3-P5. The derivation of Gx3 and Gx4 can share the calculation of P12-P14. The derivation of Gy1 and Gy3 can share the calculation of P6+P8. The derivation of Gy2 and Gy4 can share the calculation of P9+P11.

As a result, the number of arithmetic operations is reduced and the processing speed is increased. Let L be the number of gradient magnitudes to be computed at a time. In theory, the more L values tends to decrease the number of arithmetic operations. However, when this algorithm is implemented on a multicore-architecture, too large value of L leads to the contention of memory between the processing cores and degrades the speed of execution. The best value of L must empirically be determined.

Image

P0	P1	P2
P3	P4	P5
P6	P7	P8
P9	P10	P11
P12	P13	P14
P15	P16	P17

Horizontal Gradients (Gx):

Gx1 = (P0 - P2) + 2(P3-P5) + (P6 – P8)
Gx2 = (P3 – P5) + 2(P6 – P8) + (P9 – P11)
Gx3 = (P6 – P8) + 2(P9 – P11) + (P12-P14)
Gx4 = (P9 – P11) + 2(P12-P14) + (P15-P17)

Vertical Gradients (Gy):

Gy1 = (P0 + P2) – (P6 + P8) + 2(P1 – P7)
Gy2 = (P3 + P5) – (P9 + P11) + 2(P4 – P10)
Gy3 = (P6 + P8) – (P12 + P14) + 2(P7 – P13)
Gy4 = (P9 + P11) – (P15 + P17) + 2(P10 – P16)

Figure 6. Example of our proposed method with the parameter of L==4.

We implemented our proposed algorithm on the shared memory multi-core architecture. To enable parallelization, the OpenMP directive is called in the C code [16-17], as shown in Figure 7. The processing speed is further improved by utilizing the AVX intrinsics. They are built-in functions that can be mapped to the AVX instruction which supports SIMD operations [18-20].

```
#pragma omp parallel
{
  int step = image_height / omp_get_num_threads();
  for (int x = (omp_get_thread_num() * step);
          x < ((omp_get_thread_num() + 1) * step);
          x += num_line)
  {
    for (int y = 0; y < image_width; y++)
    {
      //place the proposed method in this loop-body
      ...
      ....
    }
  }
}
```

Figure 7. An OpenMP directive call and our proposed algorithm.

IV. Experimental Results

We tested the proposed algorithm on the Intel Core i5-1135G7 machine. At first, we tried to figure out which value of L would produce the best results. Six values of L, starting from 2 to 7 were carried out on the image of size 2560×1440. Each configuration was tested 100 times and the average execution time was calculated. Figure 8 shows the execution time for horizontal gradient calculation and vertical gradient calculation of our algorithm over different sizes of L. The computation time for horizontal gradient is lower than vertical gradient because it can share more arithmetic calculations between lines. The best execution time of horizontal and vertical gradient calculation was found when the L==4.

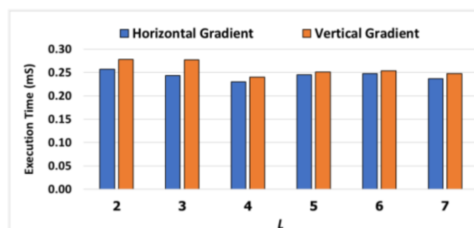


Figure 8. Execution time for horizontal and vertical gradient calculations of our algorithm over different sizes of L.

Authorized licensed use limited to: Prince of Songkla University provided by UniNet. Downloaded on October 16,2022 at 15:53:36 UTC from IEEE Xplore. Restrictions apply.

We then compared the execution time of our algorithm with the execution time of Sobel function provided by the OpenCV library (version 4.5.2). Figure 9 shows the speed-up of our algorithm over different sizes of L. Although the best execution time of horizontal and vertical gradient calculation was found with L==4, the optimum execution time of edge detection computation was reached at L==3. When the image size is 2560×1440 and the L value is 3, our edge detection algorithm is 42.01 times faster than the Sobel function of the OpenCV library.

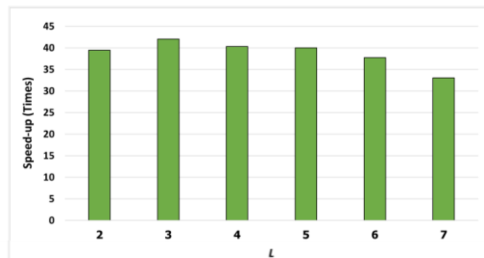


Figure 9. Speed-up of our proposed edge detections compared to the OpenCV's Sobel edge detection as a baseline, tested on the image of size 2560×1440.

We then compare the execution time of our algorithm with the OpenCV library. This test was carried out on eight image resolutions, as shown in Figure 10. In addition, we tested our algorithm over 2 different configurations: the serial AVX and the parallel AVX. The serial AVX differs from the parallel version in that the OpenMP directives were removed from the C source code. Figure 10 shows that our serial AVX version is faster than the OpenCV in every image resolution even though it runs on only single core. It is on average 9.62 times faster than the OpenCV. Our parallel AVX version is on average 28.29 times faster than the OpenCV.

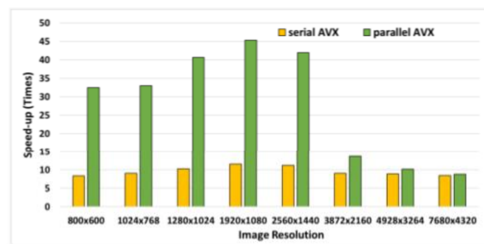


Figure 10. Speed-up of our proposed edge detections compared to the OpenCV's Sobel edge detection as a baseline, tested on 8 image resolutions.

We then applied our Sobel implementation to the Canny edge detection of the OpenCV. The source code of the cv.Canny() function was modified by removing its original gradient magnitude and direction calculation away, and be replaced by our proposed gradient direction detection. The modified Canny using our proposed algorithm was compared with its original precompiled version, and the execution time of both configurations were evaluated. As before, we had two versions of our modified Canny: the serial AVX and the parallel AVX.

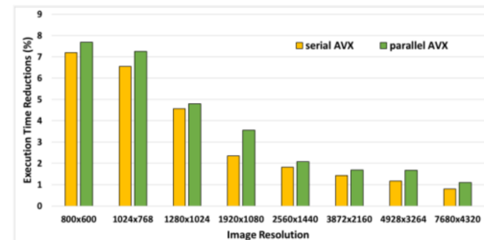


Figure 11. Execution time reduction rate of two implementations of our Canny edge detection compared to the OpenCV's Canny edge function as a baseline.

Figure 11 shows that our modified versions of the Canny edge detection are faster than the original version provided by the OpenCV library. Since the Canny edge detection comprises of many steps, so the speed-up of our modified Canny code is not as much as in the case of gradient detection alone. When the image size is 800x600, the best reduction rate is obtained. The modified Canny utilizing our serial AVX and parallel AVX can reduce the execution time by 7.19 and 7.69 percent respectively. When the image size is larger, the percentage of execution time reduction decreases. The Canny edge detection utilizing our serial AVX algorithm is on average 3.24 percent faster than that of the OpenCV library. Our parallel AVX Canny is a little bit faster than the serial version and is on average 3.73 percent faster than the original version of the OpenCV library.

V. Discussions and Conclusions

We have proposed an efficient method to augment the processing speed of Sobel and Canny edge detection by reducing the amounts of arithmetic calculations and data loading. We have demonstrated that our algorithm effectively reduces the execution time of Sobel edge detection. Our approach is cost effective since it requires only software modification without the need of any hardware accelerator. Our single thread and multi-thread implementations are both faster than the standard Sobel function provided by the OpenCV library. We have also demonstrated that the utilization of our algorithm in the Canny edge detection process can reduce its execution time. Since our algorithm reduces the amount of data loads and arithmetic operations. It possibly reduces the energy required for program execution. Our future work will focus on this issue.

References

[1] A. Jose, K. Deepa Merlin Dixon, N. Joseph, E. S. George and V. Anjitha, "Performance study of edge detection operators," 2014 International Conference on Embedded Systems (ICES), 2014, pp. 7-11, doi: 10.1109/EmbeddedSys.2014.6953040.

[2] R. Menaka, R. Janarthanan, K. Deeba, FPGA implementation of low power and high speed image edge detection algorithm, Microproc. Microsyst. 75 (2020), 103053, https://doi.org/10.1016/j.micpro.2020.103053.

[3] S. Abed, Implementation of an edge detection algorithm using FPGA reconfigurable hardware, J. Eng. Res. 8 (1) (2020) 179–197.

[4] S. Taslimi, R. Faraji, A. Aghasi, H.R. Naji, Adaptive edge detection technique implemented on FPGA, Iran. J.Sci.Technol.-Trans. Electri. Eng. (2020), https://doi.org/10.1007/s40998-020-00333-5.

Authorized licensed use limited to: Prince of Songkla University provided by UniNet. Downloaded on October 16,2022 at 15:53:36 UTC from IEEE Xplore. Restrictions apply.

[5] N. Nausheen, A. Seal, P. Khanna, S. Halder, A FPGA based implementation of Sobel edge detection, Microprocess. Microsyst. 56 (2018) 84–91, https://doi.org/10.1016/j.micpro.2017.10.011.

[6] Jiang, B. (2018). Real-time multi-resolution edge detection with pattern analysis on graphics processing unit. Journal of Real-Time Image Processing, 14(2), 293-321, doi: 10.1007/ s11554-014-0450-x

[7] S. Abed, M.H. Ali, M Al-Shayeji, Enhanced GPU-based anti-noise hybrid edge detection method, Comput. Syst. Sci. Eng. 35 (1) (2020) 21–37, https://doi.org/10.32604/csse.2020.35.021.

[8] A. Bettaieb, N. Filali, T. Filali, H.B. Aissia, GPU acceleration of edge detection algorithm based on local variance and integral image: application to air bubbles boundaries extraction, Comput. Optics 43 (3) (2019) 446–454, https://doi.org/10.18287/2412-6179-2019-43-3-446-454.

[9] J. Canny, "A Computational Approach to Edge Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, pp.679-698, Nov. 1986, doi: 10.1109/TPAMI.1986.4767851.

[10] M. A. Rahman, M. F. I. Amin and M. Hamada, "Edge Detection Technique by Histogram Processing with Canny Edge Detector," 2020 3rd IEEE International Conference on Knowledge Innovation and Invention (ICKII), 2020, pp. 128-131, doi: 10.1109/ ICKII 50300.2020.9318922.

[11] R. Maini and H. Aggarwal, "Study and comparison of various image edge detection techniques," International Journal of Image Processing (IJIP), vol. 3, pp. 1-11, 2009.

[12] W. Gao, X. Zhang, L. Yang, and H. Liu, "An improved Sobel edge detection," in Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on, 2010, pp. 67-71.

[13] Chunxi Ma, Wenshuo Gao, Lei Yang and Zhonghui Liu, "An improved Sobel algorithm based on median filter," 2010 2nd International Conference on Mechanical and Electronics Engineering, 2010, pp. V1-88-V1-92, doi: 10.1109/ICMEE.2010.5558590.

[14] M. Mekideche and Y. Ferdi, "A new edge detector based on fractional integration," 2014 International Conference on Multimedia Computing and Systems (ICMCS), 2014, pp. 223-228, doi: 10.1109/ICMCS.2014.6911409.

[15] P. M. Daigavane and P. R. Bajaj, "Road Lane Detection with Improved Canny Edges Using Ant Colony Optimization," 2010 3rd International Conference on Emerging Trends in Engineering and Technology, 2010, pp. 76-80, doi: 10.1109/ICETET. 2010.128.

[16] W.E. Liu, F. Wang, H.W. Zhou, Parallel seismic modeling based on OpenMP plus AVX and optimization strategy, J. Earth Sci. 30 (4) (2019) 843–848, https://doi.org/10.1007/s12583-018-0831-3.

[17] Y. Yan, J. Kemp, X. Tian, A. M. Malik and B. Chapman, "Performance and Power Characteristics of Matrix Multiplication Algorithms on Multicore and Shared Memory Machines," 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, 2012, pp. 626-632, doi: 10.1109/SC. Companion.2012.87.

[18] Hassan, S.A., Mahmoud, M.M., Hemeida, A.M., & Saber, M.A. (2018). Effective Implementation of Matrix-Vector Multiplication on Intel's AVX Multicore Processor. Computer Languages Systems & Structures, 51, 158-175, doi: 10.1016/j.cl.2017.06.003.

[19] H. Amiri, A. Shahbahrami, SIMD programming using Intel vector extensions, J. Parallel Distrib. Comput. 135 (2020) 83–100, https://doi.org/10.1016/j.jpdc.2019.09.012.

[20] H. Amiri and A. Shahbahrami, "High performance implementation of 2D convolution using Intel's advanced vector extensions," 2017 Artificial Intelligence and Signal Processing Conference (AISP), 2017, pp. 25-30, doi: 10.1109/AISP.2017.8324097.

Authorized licensed use limited to: Prince of Songkla University provided by UniNet. Downloaded on October 16,2022 at 15:53:36 UTC from IEEE Xplore. Restrictions apply.

Embedded
Systems Design

EMBEDDED HARDWARE DESIGN

MICPRO

MICROPROCESSORS AND MICROSYSTEMS

Microprocessors and Microsystems 87 (2021) 104368

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

High performance and energy efficient sobel edge detection

Thaufig Peng-o, Panyayot Chaikan *

Department of Computer Engineering, Faculty of Engineering, Prince of Songkla University, 15 Karnjanavanich Rd., Hadyai, Songkhla 90110, Thailand

ARTICLE INFO

Keywords:
Sobel
Edge detection
AVX
OpenMP
Image processing
Multi-core

ABSTRACT

Sobel edge detection is widely used in computer vision and image processing but its processing time becomes a serious problem in real-time environments, especially when an image is very large. Instead of utilizing a hardware-accelerated approach, we propose a purely software-based method which is simpler and cheaper. Our algorithm reduces the number of arithmetic operations and data loads, so that processing speed is increased and energy consumption reduced. The processing time is further reduced by the use of AVX intrinsics and OpenMP directives which distribute the workload among the AVX engines in a multi-core architecture. Our algorithm reduces the number of arithmetic operations by 22.73% compared to that of the state-of-the-art Sobel (SOAS) algorithm, while the number of data loads are reduced by 43.75% compared to SOAS. Performance and energy consumption comparisons between our algorithm and SOAS, as well as with the Sobel functions offered by the OpenCV and IPP libraries are investigated, and the results demonstrate that a multi-core version of our algorithm, implemented by AVX intrinsics, is on average 3.20, 9.34, and 13.99 times faster than IPP, SOAS, and OpenCV respectively. Also, it consumes an average of 2.91, 8.43, and 11.21 times less energy than IPP, SOAS, and OpenCV. Our algorithm, utilizing software modifications alone, benefits from both shorter development time and reduced cost compared to hardware approaches relying on an FPGA, ASIC, or GPU, making it more suitable for resource-constrained environments.

1. Introduction

Edge detection is a fundamental operation in computer vision, image processing, and pattern recognition. However, its processing time becomes a serious problem in a real-time environment when the image is very large, or there are multiple images in a very large database. This problem can be addressed in several ways, for example, Joshi et al. uses an application-specific integrated circuit (ASIC) [1], while other researchers utilize FPGA [2–6], or employ the GPU to augment processing speeds [7–9]. ASIC delivers the highest performance and consumes less energy than the other approaches, but comes with a very high development cost. FPGA is cheaper than ASIC but its price is still high compared to using a GPU. Although GPU performance is satisfactory, it consumes more energy than the FPGA and ASIC approaches [10]. We propose a solution to these issues by modifying software alone, which is both simpler and cheaper than hardware approaches.

The Sobel operator is one of the most popular edge detection methods, offering better performance than the Prewitt or the Robert operators in terms of noise tolerance [11], and ease of implementation in hardware such as ASICs or FPGAs [1, 2]. It can also be used as the basis for more sophisticated edge detection schemes, such as the Canny operator [12, 13].

We propose a novel approach to augmenting the processing speed of the Sobel edge detection. Our algorithm reduces the number of arithmetic operations by 22.73% compared to that of the state-of-the-art Sobel implementations, while the number of data loads are reduced by 43.75%.

Our contributions are as follows. First, we propose a method to augment the processing speed of Sobel edge detection by reducing the number of arithmetic operations and data loads. Second, different implementations of our algorithm are investigated by utilizing standard C and AVX intrinsics, and their performance are compared. Third, the processing speed and energy consumption of the Sobel edge functions provided by two well known image processing libraries are compared with our algorithm.

Our paper is organized as follows: Section 2 details background and related work, and Section 3 describes our algorithm. Section 4 outlines our implementation utilizing AVX intrinsics, and Section 5 presents experimental results. The results are discussed in Section 6, and Section 7 concludes the paper.

* Corresponding author.
E-mail address: panyayot.c@psu.ac.th (P. Chaikan).

https://doi.org/10.1016/j.micpro.2021.104368
Received 2 June 2021; Received in revised form 16 September 2021; Accepted 12 October 2021
Available online 27 October 2021
0141-9331/© 2021 Elsevier B.V. All rights reserved.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

(a) (b)

Fig. 1. Two convolution masks for the Sobel Operator:(a) a horizontal mask and (b) a vertical mask.

```
for(j=v-1;j≤v+L;j++)
{
        dx(u,j)= P(u+1,j) - P(u-1,j);
        sy(u,j)= P(u+1,j) + P(u-1,j);
}
for(k=v; k≤v+L-1; k++)
{
        Gx(u,k) = dx(u,k-1) + 2dx(u,k) + dx(u,k+1);
        Gy(u,k) = sy(u,k+1) - sy(u,k-1) + 2[P(u,k+1) - P(u,k-1)]
        ∇f(u,k) = |Gx(u,k)| + |Gy(u,k)|
}
```

Fig. 2. Calculating the gradient magnitude using the *L* parameter.

2. Background and related work

Sobel edge detection searches for an edge by means of a gradient method [14, 15]. For example, a gradient in the horizontal direction (G_x) is obtained by convolving the horizontal mask in Fig. 1(a) with an image. A vertical gradient (G_y) is derived by applying the vertical mask in Fig. 1(b) in a similar way. The gradient magnitude, defined as ∇f, is obtained by using:

$$\nabla f = \sqrt{G_x^2 + G_y^2}. \tag{1}$$

Applying square root and squaring operations over all the pixels of the image is computational expensive, so most implementations reduce the burden by using the approximation [2, 4, 5, 15] which is defined as:

$$\nabla f \approx |G_x| + |G_y| \tag{2}$$

The simplicity of convolution masks means that most state-of-the-art Sobel (SOAS) implementations [2, 5] try to avoid direct convolution between the masks and the image by utilizing addition, subtraction, and left-shift operations. There is no need to perform multiplications between all the zeros and ones of the masks and image pixels. Also, multiplication by two can be replaced by a left-shift which is simpler and requires less energy [2,5]. If $P(i, j)$ is the source image pixel located at (i, j), then the horizontal gradient at location (u, v) can be determined by:

$$G_x(u,v) = [P(u+1,v-1) - P(u-1,v-1)] + 2[P(u+1,v) - P(u-1,v)] + [P(u+1,v+1) - P(u-1,v+1)] \tag{3}$$

and the vertical gradient is:

$$G_y(u,v) = [P(u+1,v+1) + P(u-1,v+1)] - [P(u+1,v-1) + P(u-1,v-1)] + 2[P(u,v+1) - P(u,v-1)] \tag{4}$$

By applying Eqs. (2) – (4), to the input image, SOAS effectively reduces the number of arithmetic operations required by 59.46% compared to convolution. However, our approach further reduces the amount of SOAS computation, thereby increasing the processing speed and reducing energy consumption.

3. Proposed method

Most Sobel edge detection implementations, including SOAS, calculate the gradient magnitude one line at a time. The process starts from the left-most pixel, and the adjacent pixels in the same line are calculated one by one until the gradient magnitude of the line is obtained. This process is applied to the image line by line until the image have been completely processed.

We calculate the gradient magnitude by utilizing more than one line at a time, which allows some arithmetic operations to be shared among lines, so the total number of calculations is reduced.

If L is the number of lines used to calculate the gradient magnitude at a time, then $\nabla f(u, v)$, $\nabla f(u, v + 1)$, …, $\nabla f(u, v + L - 1)$ can be calculated as shown in Fig. 2.

The first for-loop in Fig. 2 generates partial differences (d_x), and partial sums (s_y) of the image pixels in the horizontal and vertical directions, respectively. The second loop uses these values to calculate the vertical and horizontal gradient before the gradient magnitude is derived.

For example, if $L == 4$, then six partial differences in the x-direction, denoted d_x, are calculated:

$d_x(u, v - 1) = P(u + 1, v - 1) - P(u - 1, v - 1)$;
$d_x(u, v) = P(u + 1, v) - P(u - 1, v)$;
$d_x(u, v + 1) = P(u + 1, v + 1) - P(u - 1, v + 1)$;
$d_x(u, v + 2) = P(u + 1, v + 2) - P(u - 1, v + 2)$;
$d_x(u, v + 3) = P(u + 1, v + 3) - P(u - 1, v + 3)$;
$d_x(u, v + 4) = P(u + 1, v + 4) - P(u - 1, v + 4)$.

The partial sums in the y-direction, denoted s_y, are also obtained:

$s_y(u, v - 1) = P(u + 1, v - 1) + P(u - 1, v - 1)$;
$s_y(u, v) = P(u + 1, v) + P(u - 1, v)$;
$s_y(u, v + 1) = P(u + 1, v + 1) + P(u - 1, v + 1)$;
$s_y(u, v + 2) = P(u + 1, v + 2) + P(u - 1, v + 2)$;
$s_y(u, v + 3) = P(u + 1, v + 3) + P(u - 1, v + 3)$;
$s_y(u, v + 4) = (u + 1, v + 4) + P(u - 1, v + 4)$.

The four horizontal gradients are obtained using:

$G_x(u, v) = d_x(u, v - 1) + 2d_x(u, v) + d_x(u, v + 1)$;
$G_x(u, v + 1) = d_x(u, v) + 2d_x(u, v + 1) + d_x(u, v + 2)$;
$G_x(u, v + 2) = d_x(u, v + 1) + 2d_x(u, v + 2) + d_x(u, v + 3)$;
$G_x(u, v + 3) = d_x(u, v + 2) + 2d_x(u, v + 3) + d_x(u, v + 4)$.

The four vertical gradients are:

$G_y(u, v) = s_y(u, v + 1) - s_y(u, v - 1) + 2[P(u, v + 1) - P(u, v - 1)]$;
$G_y(u, v + 1) = s_y(u, v + 2) - s_y(u, v) + 2[P(u, v + 2) - P(u, v)]$;
$G_y(u, v + 2) = s_y(u, v + 3) - s_y(u, v + 1) + 2[P(u, v + 3) - P(u, v + 1)]$;
$G_y(u, v + 3) = s_y(u, v + 4) - s_y(u, v + 2) + 2[P(u, v + 4) - P(u, v + 2)]$.

The four gradient magnitudes, which are vertically aligned, are determined using:

$\nabla f(u, v) = |G_x(u, v)| + |G_y(u, v)|$;
$\nabla f(u, v + 1) = |G_x(u, v + 1)| + |G_y(u, v + 1)|$;
$\nabla f(u, v + 2) = |G_x(u, v + 2)| + |G_y(u, v + 2)|$;
$\nabla f(u, v + 3) = |G_x(u, v + 3)| + |G_y(u, v + 3)|$.

The calculation of $d_x(u, v + 1)$ is performed only once, but its result is used to calculate three horizontal gradients – $G_x(u, v)$, $G_x(u, v + 1)$, and $G_x(u, v + 2)$ – thereby reducing the number of subtractions. In a similar way, $d_x(u, v)$ is employed by both $G_x(u, v)$ and $G_x(u, v + 1)$, and $d_x(u, v + 2)$ is used by $G_x(u, v + 1)$ and $G_x(u, v + 2)$. In addition, $d_x(u, v + 3)$ is utilized by $G_x(u, v + 2)$ and $G_x(u, v + 3)$.

The vertical gradients also share partial sums, but to a lesser degree. For example, $s_y(u, v + 1)$ is used by $G_y(u, v)$ and $G_y(u, v + 2)$, and $s_y(u, v + 2)$ is utilized in $G_y(u, v + 1)$ and $G_y(u, v + 3)$.

The first for-loop in Fig. 2 needs to be completely unrolled so that all of the d_x and d_y values are retained in the CPU's register or cache for faster access by the subsequent gradient calculations. When this algorithm is implemented, an optimized value for L needs to be determined which produces best performance.

Our algorithm requires the same number of absolute and left-shift operations as SOAS. However, our method requires less additions and subtractions because they are shared between lines. For the previous example, our algorithm requires 36 additions and subtractions to produce four vertically aligned gradient magnitudes, denoted $\nabla f(u, v)$, $\nabla f(u, v + 1)$, $\nabla f(u, v + 2)$, and $\nabla f(u, v + 3)$. On the other hand, SOAS requires 11 additions and subtractions for each gradient magnitude based on Eqs. (2) – (4). This means that SOAS requires 44 additions and subtractions to produce the same number of gradient magnitudes.

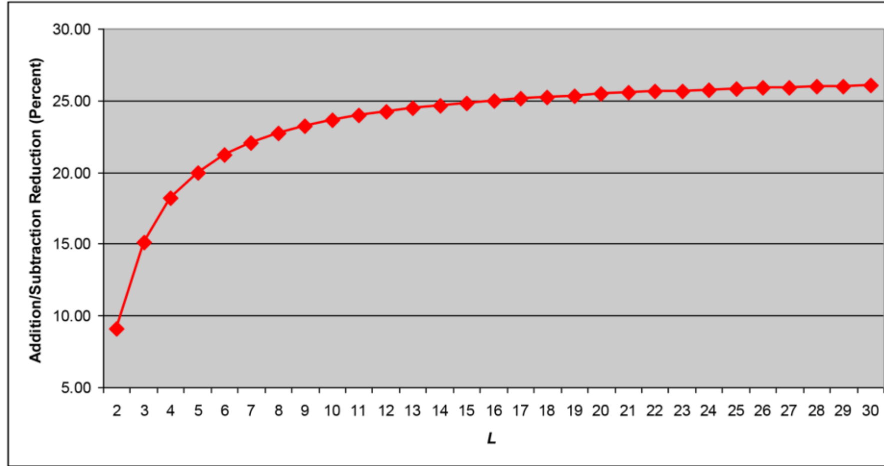


Fig. 3. The percentage of addition/subtraction reductions in our algorithm compared to SOAS.

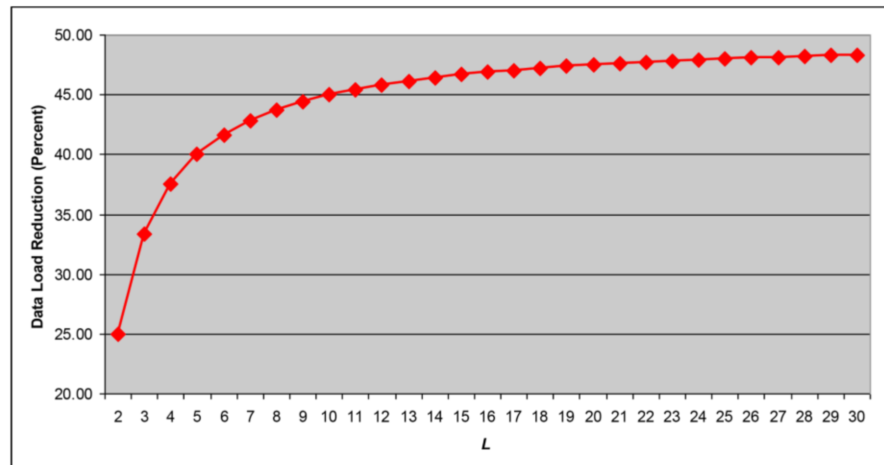


Fig. 4. The percentage of data load reductions in our algorithm compared to SOAS.

Therefore, our algorithm reduces the number of additions and subtractions compared to SOAS by 18.18 percent in this case.

Let ∇f be the gradient magnitude for an image of size $w \times h$. The number of additions and subtractions is denoted as OP_{AS}, which in our algorithm with the L parameter is determined by:

$$OP_{AS} = \frac{h^* w^* (8L + 4)}{L}. \tag{5}$$

However, the OP_{AS} equation for SOAS is:

$$OP_{AS} = 11^* h^* w. \tag{6}$$

Fig. 3 shows the percentage of addition/subtraction reductions for our algorithm compared to SOAS, with L ranging from 2 to 30. When $L < 10$, the reduction rate increases dramatically with increasing L. However, when $L \geq 10$, the reduction rate still increases with increasing L, but at a lower rate.

Our algorithm not only reduces the number of add/subtract operations, but also the number of data loads. When processing an image of size $w \times h$, the number of data loads, denoted as $LOAD_{NUM}$, of our algorithm is:

$$LOAD_{NUM} = \frac{(4L + 4)^* h^* w}{L}. \tag{7}$$

However, the $LOAD_{NUM}$ for SOAS is:

$$LOAD_{NUM} = 8^* h^* w. \tag{8}$$

Fig. 4 shows the data load reduction percentage for our algorithm compared to SOAS when L ranges from 2 to 30. It follows the same pattern as Fig. 3, with the load reduction rate high for smaller values of L, but becoming more less significant with larger L.

Figs. 3 and 4 illustrate how the L value affects the number of arithmetic and data load reductions, and in particular that after a certain value of L is reached, the improvement becomes almost constant. Clearly, for the best performance, this value of L must be determined.

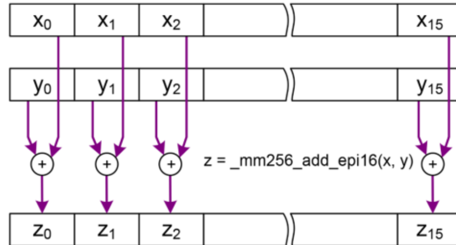


Fig. 5. An example of using the AVX intrinsic function *_mm256_add_epi16*.

4. Implementation utilizing conventional C and AVX intrinsics

Modern computer architectures come equipped with powerful SIMD instructions, such as those found in the NEON [16] extension to the ARM architecture, or the Altivec [17] instructions in the PowerPC platform. For the x86 architecture, the Advanced Vector extension (AVX) provides 256-bit SIMD registers supporting both floating-point and integer operations [18]. These allow eight 32-bit or sixteen 16-bit integer values to be processed within one clock cycle, and for floating-point data, they permit eight single precision or four double precision values to be processed simultaneously.

Several researchers have looked at using AVX to augment processing speeds. For example, Lasch et al. employed it to enhance strong string dictionary compression [19], and Zhang et al. utilized it to accelerate the determination of 3D rotations for aligning two objects or sets of points [20]. Fortin et al. used AVX to increase the efficiency of polynomial factorization and polynomial greatest common divisor computation [21], Hemeida et al. optimized matrix-matrix multiplication speed on a multi-core architecture [22], and Andre et al. utilized AVX for efficient nearest neighbor search in high-dimensional space [23].

There are three ways to take advantage of AVX instructions in C [24]: 1) by using AVX inline assembly; 2) by employing the compiler's automatic vectorization features; or 3) by utilizing compiler intrinsics.

Unfortunately, some current C compilers only support inline assembly in 32-bit applications [25], so we decided to implement our algorithm using two different approaches: 1) by utilizing conventional C with the compiler's automatic vectorization left to generate the machine language for using the AVX instructions; and 2) by coding with AVX intrinsics directly, which allows us to vectorize our algorithm ourselves, and so obtain maximum performance. Fig. 5 shows an example of how to use the intrinsic function *_mm256_add_epi16*, which is translated into be the machine instruction "*vpaddw ymm, ymm, ymm*", and allows sixteen 16-bit integer operands to be added during one machine cycle.

Fig. 6 shows the C implementation of our algorithm when L==2. It generates two vertically aligned gradient magnitudes by reading 12 source image pixels. It requires only 20 additions/subtractions, while SOAS needs 22 operations to produce the same result.

Manual vectorization using intrinsic functions can easily outperform a compiler's automatic vectorization, as confirmed in work by Amiri and Shahbahrami [26]. However, for our algorithm to fully utilize the CPU's AVX capabilities, the AVX intrinsic functions must be called from C [27]. Rather than employing AVX floating-point instructions, we decided to implement our algorithm by utilizing AVX's integer instructions for two reasons. The first is that they allow sixteen 16-bit signed integer values to be processed simultaneously, twice the number for single precision floating-point operations [18]. The second is that they are faster than floating-point arithmetic and require less energy.

Every image pixel load shown in Fig. 6 is replaced by the AVX integer load function *_mm256_loadu_si256*, as shown in Fig. 7 (lines 12–16). As an AVX register is 256 bits large, one load operation can read 32 8-bit image pixels into the AVX register. These pixels must be processed in two halves. The first half, named *P0L*, and comprising *p0–p7* and *p16–p23*, are extracted and interleaved with zeros kept in a *z* variable before storing the sixteen 16-bit signed integers in the AVX register. This 8-bit to 16-bit zero extension process is carried out by the *_mm256_unpacklo_epi8* function, as shown on lines 18–22. The second half, named *P0H*, and comprising *p8-p15* and *p24-p31*, are zero extended in a similar way but by means of the *_mm256_unpackhi_epi8* function. This zero extension process is required for every image pixel load, producing lower and higher half variables with the suffix 'L' and 'H' respectively, (e.g. variables *P0L–P11L* and *P0H–P11H* in Figs. 7 and 8). Each half is

```
 1  char *pS8;//pS8 is a pointer to the source image
 2  char *pD8;//pD8 is a pointer to the destination image
 3  typedef short s;
 4  short D0x,D1x,D2x,D3x, S0y,S1y,S2y,S3y;
 5  short P0,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11;
 6
 7  int step2 = image_width;
 8  int step3 = 2*image_width;
 9  int step4 = 3*image_width;
10
11  P0=(s)*pS8;            P1=(s)*(pS8+1);           P2=(s)*(pS8+2);
12  P3=(s)*(pS8+step2);    P4=(s)*(pS8+step2+1);     P5=(s)*(pS8+step2+2);
13  P6=(s)*(pS8+step3);    P7=(s)*(pS8+step3+1);     P8=(s)*(pS8+step3+2);
14  P9=(s)*(pS8+step4);    P10=(s)*(pS8+step4+1);    P11=(s)*(pS8+step4+2);
15
16  D0x = P2-P0;           S0y = P2+P0;
17  D1x = P5-P3;           S1y = P5+P3;
18  D2x = P8-P6;           S2y = P8+P6;
19  D3x = P11-P9;          S3y = P11+P9;
20
21  short G0x = D0x + (D1x<<1) +D2x;
22  short G1x = D1x + (D2x<<1) +D3x;
23  short G0y = S0y - S2y + (P1-P7)<<1;
24  short G1y = S1y - S3y + (P4-P10)<<1;
25
26  short R0 = abs(G0x) + abs(G0y);
27  short R1 = abs(G1x) + abs(G1y);
28  *(pD8+step2) = R0;
29  *(pD8+step3) = R1;
```

Fig. 6. The key parts of our algorithm implemented in C with L == 2.

```
1  char *pS8;//pS8 is a pointer to the source image
2  char *pD8;//pD8 is a pointer to the destination image
3  __m256i P0,P1,...,P11, P0L,P1L,...,P11L, P0H,P1H,...,P11H;
4  __m256i D0L,D1L,D2L,D3L, D0H,D1H,D2H,D3H;
5  __m256i S0L,S1L,S2L,S3L, S0H,S1H,S2H,S3H;
6  __m256i z = _mm256_setzero_si256();
7
8  int step2 = image_width;
9  int step3 = 2*image_width;
10 int step4 = 3*image_width;
11
12 P0 = _mm256_loadu_si256((__m256i*)pS8);
13 P1 = _mm256_loadu_si256((__m256i*)(pS8 + 1));
14 ...
15 P10 = _mm256_loadu_si256((__m256i*)(pS8 + step4 + 1));
16 P11 = _mm256_loadu_si256((__m256i*)(pS8 + step4 + 2));
17
18 P0L = _mm256_unpacklo_epi8(P0,z);  P0H = _mm256_unpackhi_epi8(P0, z);
19 P1L = _mm256_unpacklo_epi8(P1,z);  P1H = _mm256_unpackhi_epi8(P1, z);
20 ...
21 P10L = _mm256_unpacklo_epi8(P10,z); P10H = _mm256_unpackhi_epi8(P10, z);
22 P11L = _mm256_unpacklo_epi8(P11,z); P11H = _mm256_unpackhi_epi8(P11, z);
23
24 D0L = _mm256_sub_epi16(P2L, P0L);  D0H = _mm256_sub_epi16(P2H, P0H);
25 D1L = _mm256_sub_epi16(P5L, P3L);  D1H = _mm256_sub_epi16(P5H, P3H);
26 D2L = _mm256_sub_epi16(P8L, P6L);  D2H = _mm256_sub_epi16(P8H, P6H);
27 D3L = _mm256_sub_epi16(P11L, P9L);  D3H = _mm256_sub_epi16(P11H, P9H);
28 S0L = _mm256_add_epi16(P2L, P0L);  S0H = _mm256_add_epi16(P2H, P0H);
29 S1L = _mm256_add_epi16(P5L, P3L);  S1H = _mm256_add_epi16(P5H, P3H);
30 S2L = _mm256_add_epi16(P8L, P6L);  S2H = _mm256_add_epi16(P8H, P6H);
31 S3L = _mm256_add_epi16(P11L, P9L);  S3H = _mm256_add_epi16(P11H, P9H);
32
33 G0xL = _mm256_add_epi16(_mm256_add_epi16(D0L, _mm256_slli_epi16(D1L, 1)), D2L);
34 G1xL = _mm256_add_epi16(_mm256_add_epi16(D1L, _mm256_slli_epi16(D2L, 1)), D3L);
35 S0L = _mm256_sub_epi16(S0L, S2L);
36 S1L = _mm256_sub_epi16(S1L, S3L);
37 G0yL = _mm256_add_epi16(S0L, _mm256_slli_epi16(_mm256_sub_epi16(P1L, P7L), 1));
38 G1yL = _mm256_add_epi16(S1L, _mm256_slli_epi16(_mm256_sub_epi16(P4L, P10L),
39 1));
40
41 G0xH = _mm256_add_epi16(_mm256_add_epi16(D0H, _mm256_slli_epi16(D1H, 1)), D2H);
42 G1xH = _mm256_add_epi16(_mm256_add_epi16(D1H, _mm256_slli_epi16(D2H, 1)), D3H);
43 S0H = _mm256_sub_epi16(S0H, S2H);
44 S1H = _mm256_sub_epi16(S1H, S3H);
45 G0yH = _mm256_add_epi16(S0H, _mm256_slli_epi16(_mm256_sub_epi16(P1H, P7H), 1));
46 G1yH = _mm256_add_epi16(S1H, _mm256_slli_epi16(_mm256_sub_epi16(P4H, P10H),
47 1));
48
49 R0L = _mm256_add_epi16(_mm256_abs_epi16(G0xL), _mm256_abs_epi16(G0yL));
50 R1L = _mm256_add_epi16(_mm256_abs_epi16(G1xL), _mm256_abs_epi16(G1yL));
51
52 R0 = _mm256_packus_epi16(R0L, R0H);
53 R1 = _mm256_packus_epi16(R1L, R1H);
54 _mm256_store_si256((__m256i*) (pD8 + step2), R0);
55 _mm256_store_si256((__m256i*) (pD8 + step3), R1);
```

Fig. 7. The key parts of our algorithm utilizing AVX with $L == 2$.

processed separately using the AVX arithmetic and logic instructions. The add, the subtract, the absolute, and the left shift operations which are utilized in the C code in Fig. 6 are replaced by the functions *_mm256_add_epi16*, *_mm256_sub_epi16*, *_mm256_abs_epi16*, and *_mm256_slli_epi16* respectively, as in Fig. 7. Utilizing these functions allows sixteen image pixels to be processed simultaneously.

After both the lower and higher halves of the results are obtained, each consisting of sixteen 16-bit signed integers, they are converted back to 8-bit integers using unsigned saturation by calling *_mm256_packus_epi16*, as illustrated in Figs. 7 and 8.

Our algorithm to distributes its workload among the core's AVX engines by calling an OpenMP directive in the C code, as shown in Fig. 9. It lets the runtime environment create multiple threads on different cores, and also splits the data between the cores to enhance data parallelism [24, 28].

5. Experimental results

We tested our algorithm on a 2.5 GHz Core i7–4710HQ (Haswell microarchitecture) with four processing cores and Hyperthreading turned on, coding in Microsoft Visual C++ 2019 and OpenMP. We implemented our algorithm using two approaches: 1) with the arithmetic operations provided in standard C, as in the example in Fig. 6, and 2) with the AVX load and arithmetic intrinsic functions, as in the example in Fig. 7.

Each approach resulted in two implementations: one serial, the other

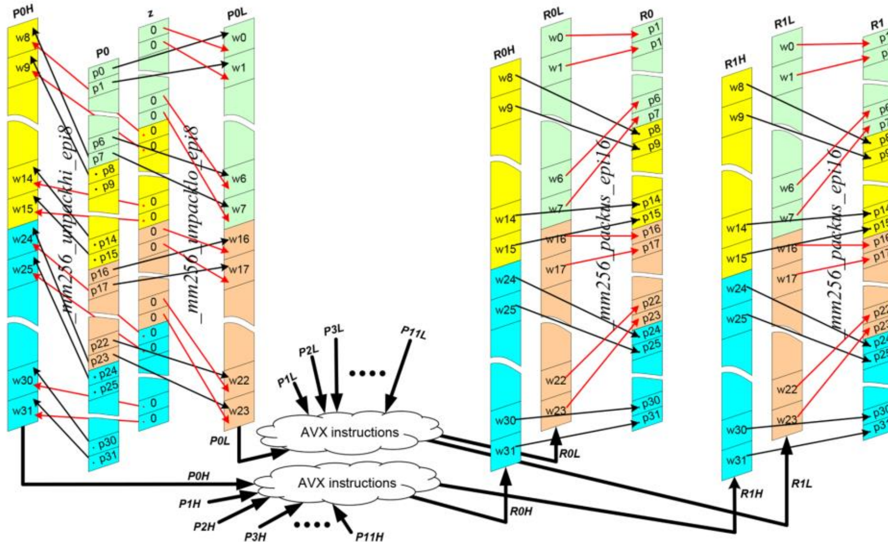


Fig. 8. Separating 32 image pixels into two packs of sixteen 16-bit data, before and after processing them with AVX's 16-bit-integer arithmetic instructions.

```
#pragma omp parallel for
for (int y = 0; y < image_rows; y++)
  for (int x = 0; x < image_columns; x++)
  {
    //place the Sobel algorithm in this loop-body
    ..
    ...
    ....
  }
```

Fig. 9. An OpenMP directive used with Sobel edge detection.

parallel. As a result, there are four possible configurations for our algorithm: 1) a serial version utilizing standard C, called "serial non-AVX" in Fig. 10, 2) a parallel version utilizing OpenMP ("parallel non-AVX"), 3) parallel version using AVX ("parallel AVX"), and 4) a serial version employing AVX ("serial AVX").

Ten values for L, ranging from 2 to 15, were tested to find the most appropriate number of image lines to process at a time. An image of size 3840 $\times$ 2160 was passed to each implementation and the execution times measured. Each configuration was run 20 times, and the average execution times are shown in Figs. 10 and 11.

Figs. 10 and 11 show that when L is small, performance increases with its size. However, performance drops when L reaches a certain value. For the non-AVX code, the maximum performance for the serial and parallel implementations is obtained when L is 10 and 9. For the AVX code, peak performance for the serial and parallel implementations occurs when L is 7 and 8. All subsequent tests used these settings to obtain maximum performance.

In summary, when the source image is of size 3840 $\times$ 2160, parallel AVX is the most effective implementation, on average 1.40 times faster than the serial utilizing AVX alone. Also, this implementation is on average 10.75 and 4.06 times faster than the serial and parallel implementations of the non-AVX code.

We also compared the performance of our four implementations against the SOAS and the Sobel functions provided by Intel's Integrated Performance Primitives (IPP) and the OpenCV libraries. The kernel size of the filters in these libraries were set to 3 $\times$ 3, and their gradient

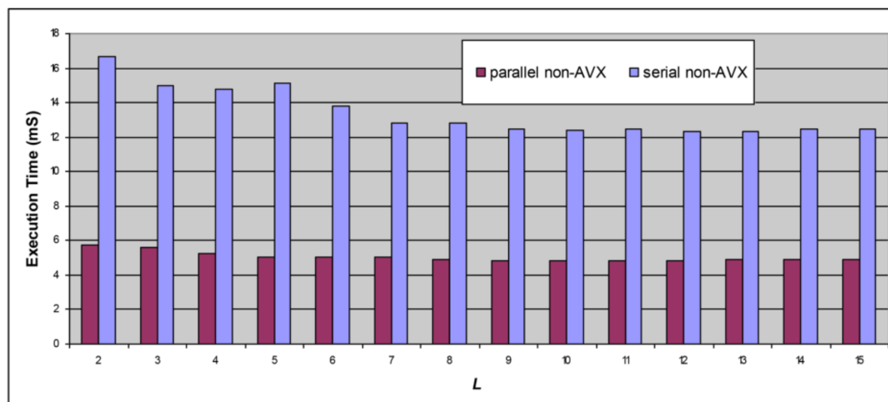


Fig. 10. Performance of the non-AVX code with different L parameters.

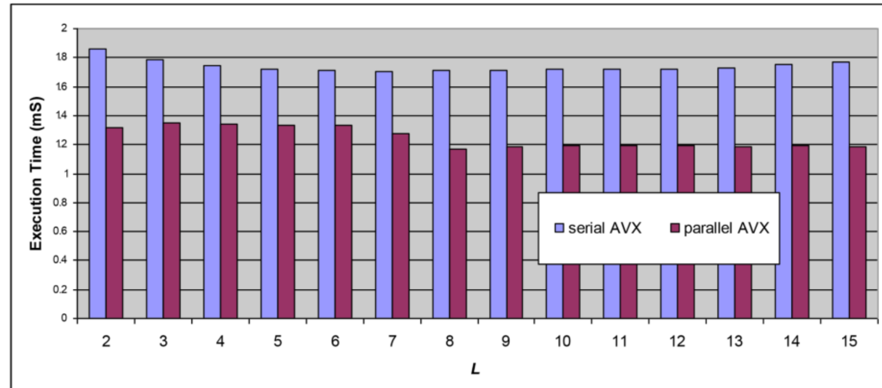


Fig. 11. Performance of the AVX code with different L parameters.

Table 1
Performance of the four implementations of our algorithm compared to the IPP, OpenCV, and SOAS libraries.

Image Resolution	Execution Time (mS) Our Algorithm SerialNon-AVX	ParallelNon-AVX	Serial AVX	ParallelAVX	IPP	OpenCV	SOAS
1920 × 1080	3.009	1.159	0.329	0.107	0.553	2.628	1.831
2560 × 1440	5.480	2.178	0.624	0.228	1.082	4.871	3.429
3840 × 2160	12.425	4.830	1.706	1.171	2.658	10.887	6.823
4896 × 3264	24.535	9.998	3.295	2.808	5.308	21.033	13.716
7680 × 4320	55.618	22.885	7.136	6.584	12.813	47.662	30.394

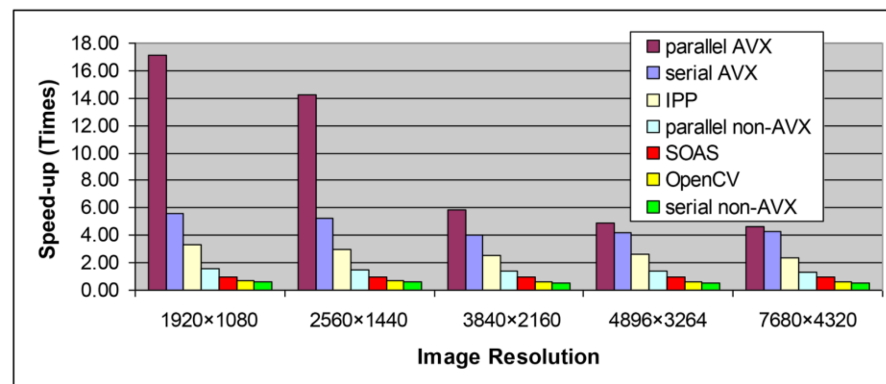


Fig. 12. Speed-ups of the four implementations of our algorithm, IPP, and OpenCV compared to SOAS as a baseline.

magnitude derivation options set to utilize the absolute operation instead of square root. We tested five image resolutions: 1920 × 1080, 2560 × 1440, 3840 × 2160, 4896 × 3264, and 7680 × 4320, and for each configuration, the image was processed 20 times and the average execution times evaluated. Table 1 compares the performance of the image edge detection of our algorithm with the IPP, OpenCV, and SOAS libraries.

Fig. 12 shows the speed-ups of the implementations of our algorithm, OpenCV, and the IPP libraries, by utilizing execution time of SOAS as a baseline. OpenCV and our serial non-AVX code are on average 34.39 and 42.88 percent slower than SOAS. However, SOAS is on average 1.44, 2.77, 4.64, and 9.34 times slower than our parallel non-AVX code, IPP, the serial AVX code, and the parallel AVX code respectively. In other words, IPP is faster than our parallel non-AVX code but slower than the AVX implementations of our algorithm.

By utilizing the Intel Power Gadget tool [29], we also compared the energy consumption of image edge detection utilizing SOAS, OpenCV, IPP, and the four implementations of our algorithm using the same five image resolutions as before. For each configuration, the image was processed 20 times and the average energy consumption (in Joules) are shown in Table 2.

Fig. 13 shows the energy reductions of the implementations of our algorithm, OpenCV, and IPP libraries by utilizing the energy consumption of SOAS as a baseline. On average, SOAS consumes 25.46 and 29.21

Table 2
Energy consumption of the four implementations of our algorithm compared to the IPP, OpenCV, and SOAS libraries.

Image Resolution	Energy consumption (mJ) Our Algorithm SerialNon-AVX	ParallelNon-AVX	Serial AVX	ParallelAVX	IPP	OpenCV	SOAS
1920 × 1080	85.369	41.452	9.962	4.447	19.407	83.444	64.374
2560 × 1440	156.179	76.351	20.034	9.435	38.449	155.280	115.827
3840 × 2160	358.326	180.665	50.276	41.668	94.415	309.669	246.484
4896 × 3264	703.775	353.004	104.491	95.205	187.415	672.031	498.318
7680 × 4320	1595.083	814.020	233.626	242.317	449.753	1537.783	1033.760

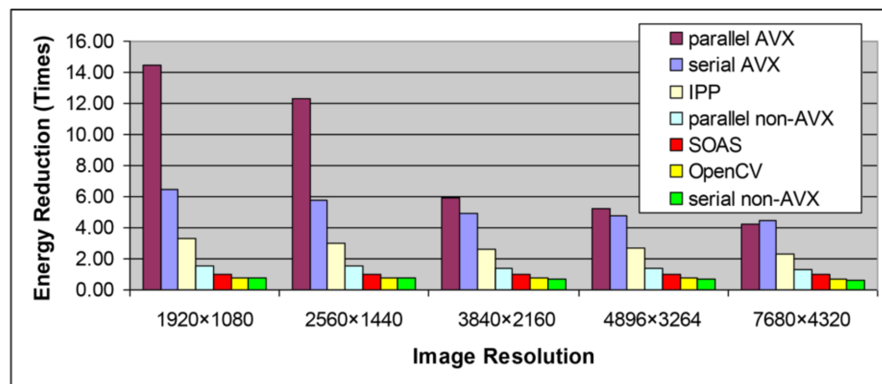


Fig. 13. Energy reductions of the four implementations of our algorithm, IPP, and OpenCV compared to SOAS as a baseline.

percent less energy than OpenCV and our serial non-AVX code. However, SOAS consumes on average 1.42, 2.78, 5.27, and 8.43 times more energy than our parallel non-AVX code, IPP, the serial AVX, and the parallel AVX implementations respectively.

6. Discussion

As reported in the previous section, the most appropriate *L* values for the serial non-AVX, the parallel non-AVX, the serial AVX, and the parallel AVX implementations were 10, 9, 7, and 8 respectively. For smaller sizes of *L*, the performance was lower because less arithmetic operation reduction was possible. With bigger *L*'s, the increased arithmetic operation reduction was offset by the increase in the number of variables in the code. In particular, the limited number of the AVX registers caused register spilling to increase with more variable. Register spilling increases memory accesses, which is much slower than accessing registers.

Another question is why the optimum values for the non-AVX implementations (10 and 9) are higher than those for the AVX implementations (7 and 8). The AVX load instruction reads pixels from memory 32 bytes at a time. However, the simpler instruction in the non-AVX implementations writes to a C "short" variable which can only hold two bytes. This means that AVX code accesses memory at a much higher rate than non-AVX instructions. This means that the *L* value for the non-AVX code can be set higher than that for the AVX code before register spilling causes extra memory accesses.

The previous section showed that our serial non-AVX code is slower than SOAS. This is because the serial code is unable to distribute its workload among cores unlike SOAS, even though it requires less arithmetic operations. However, our parallel non-AVX implementation is on average 1.44 times faster than SOAS.

OpenCV is faster than our serial non-AVX code because it utilizes AVX instructions (if the CPU supports them) and a multi-core architecture [30]. However, OpenCV's Sobel function is slower than the one in SOAS because it uses convolution which requires a larger number of arithmetic operations.

IPP is faster than SOAS, OpenCV, and our parallel non-AVX code because it is fully optimized for Intel processors and effectively utilizes all of the AVX engines in all of the cores [31]. However, our AVX implementations are faster even though IPP also supports AVX and a multi-core architecture. Our AVX-implementation is faster because it uses less data loads and less arithmetic operations.

The next question is why the speed-up of our algorithm compared to SOAS is high when the image size is small, but dramatically drops as the size increases? Much of the speed when the image size is small is caused by the cache effect. For example, the speed-up of the parallel AVX code versus SOAS for images of size 1920 × 1080 and 7680 × 4320 is 17.11 and 4.62 times. Profiling revealed that when the image is small, all the data can be stored inside the CPU cache, so the AVX engines can process them without delay. However, when the image is larger than the cache, the speed-up decreases due to memory bottlenecks because each core has to wait for data coming from main memory. This phenomenon occurs not only with our AVX code, but also with IPP and OpenCV, as shown in Fig. 12.

Our parallel AVX code effectively distributes the workload among all the processing cores only when the image size is small. For example, it is 3.07 times faster than the serial AVX code when the image size is 1920 × 1080. However, it is only 1.08 times faster when the image size is 7680 × 4320. The main reason for this are memory bottlenecks and increasing memory contention among the cores. On average, our parallel AVX code is 3.20, 9.34, and 13.99 times faster than IPP, SOAS, and OpenCV. However, when processing very large images, the parallel AVX code speed-ups drop to 1.95, 4.62, and 7.24 times faster than IPP, SOAS, and OpenCV.

Compared to OpenCV and IPP, the energy consumption of our code follows the same pattern as its speed-ups. The energy reduction rate is high when the image is small, but drops as the image gets bigger. There are two main reasons why our algorithm is more energy efficient than IPP and OpenCV: it requires less arithmetic operations and less data

loads. On average, our parallel AVX code's energy consumption is 2.91, 8.43, and 11.21 times less than IPP, SOAS, and OpenCV. However, for very large images, the energy consumption is only 1.86, 4.27, and 6.35 times less than IPP, SOAS, and OpenCV.

7. Conclusions

We have proposed a new algorithm for Sobel edge detection that is both faster and consumes less energy. By processing multiple rows of an image at once, the arithmetic operations for different rows can be shared so that the number of data loads and arithmetic calculations are reduced. Four different configurations of our algorithm were implemented, and their performance evaluated against the state-of-the-art Sobel (SOAS) algorithm, and the Sobel functions of the IPP and OpenCV image processing libraries.

Peak performance is obtained from our algorithm when it is implemented using AVX intrinsics and parallelized using OpenMP directives with $L == 8$. This configuration reduces the number of arithmetic operations and data loads by 22.73 and 43.75 percent compared to SOAS. The experimental results shows that our parallel AVX implementation is on average 3.20, 9.34, and 13.99 times faster than IPP, SOAS, and OpenCV respectively. Also, it consumes an average of 2.91, 8.43, and 11.21 times less energy than IPP, SOAS, and OpenCV.

Our algorithm adds another choice to the range of approaches for augmenting processing speed but without requiring accelerated hardware such as an FPGA, ASIC, or GPU. Our algorithm, based on software modifications alone, offers both shorter development time and reduced cost compared to these other methods, making it more suitable for use in resource-constrained environments.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The authors are grateful to Dr. Andrew Davison for his kind help in polishing the language of this paper.

References

[1] Joshi, R., Zaman, M.A., & Katkoori, S. (2020). Novel bit-sliced near-memory computing based vlsi architecture for fast sobel edge detection in IoT edge devices, in: 2020 IEEE international symposium on smart electronic systems (iSES), Chennai, India, 291–296, doi: 10.1109/iSES50453.2020.00071.

[2] R. Menaka, R. Janarthanan, K. Deeba, FPGA implementation of low power and high speed image edge detection algorithm, Microproc. Microsyst. 75 (2020), 103053, https://doi.org/10.1016/j.micpro.2020.103053. Article Number.

[3] S. Taslimi, R. Faraji, A. Aghasi, H.R. Naji, Adaptive edge detection technique implemented on FPGA, Iran. J.Sci.Technol.-Trans. Electri. Eng. (2020), https://doi.org/10.1007/s40998-020-00333-5.

[4] N. Nausheen, A. Seal, P. Khanna, S. Halder, A FPGA based implementation of Sobel edge detection, Microprocess. Microsyst. 56 (2018) 84–91, https://doi.org/10.1016/j.micpro.2017.10.011.

[5] S. Abed, Implementation of an edge detection algorithm using FPGA reconfigurable hardware, J. Eng. Res. 8 (1) (2020) 179–197.

[6] P. Sikka, A.R. Asati, C. Shekhar, High-Speed and area-efficient sobel edge detector on field-programmable gate array for artificial intelligence and machine learning applications, Comput. Intell. (2020) 1–12, https://doi.org/10.1111/coin.12334.

[7] Jiang, B. (2018). Real-time multi-resolution edge detection with pattern analysis on graphics processing unit. *J. Real-Time Image Process.*14(2), 293–321, doi: 10.1007/s11554-014-0450-x.

[8] S. Abed, M.H. Ali, M Al-Shayeji, Enhanced GPU-based anti-noise hybrid edge detection method, Comput. Syst. Sci. Eng. 35 (1) (2020) 21–37, https://doi.org/10.32604/csse.2020.35.021.

[9] A. Bettaieb, N. Filali, T. Filali, H.B. Aissia, GPU acceleration of edge detection algorithm based on local variance and integral image: application to air bubbles boundaries extraction, Comput. Optics 43 (3) (2019) 446–454, https://doi.org/10.18287/2412-6179-2019-43-3-446-454.

[10] Jin, Z., & Finkel, H. (2019). Exploration of OpenCL 2D Convolution Kernels On Intel FPGA, CPU, and GPU Platforms, in: 2019 IEEE International Conference on Big Data (IEEE Big Data 2019), Los Angeles, USA, 4460–4465, doi:10.1109/BigData47090. 2019.9006494.

[11] S.S. Abdullah, M.P. Rajasekaran, Modified Sobel Mask to Locate Knee Joint Boundaries, 3C TECNOLOGIA (2020) 195–204. SI10.17993/3ctecno.2020.specialissue4.195-205.

[12] D.H.E. Lee, P.Y. Chen, F.H. Yang, W.T. Weng, High-efficient low-cost VLSI implementation for canny edge detection, J. Inf. Sci. Eng. 36 (3) (2020) 535–546, https://doi.org/10.6688/JISE.202005_36(3).0004.

[13] Liu, Z., Jing, F., Fan, J., & Wang Z. (2019). Implementation of a FPGA-ARM-based canny edge detection system, in: 2019 Chinese Control Conference (CCC), Guangzhou, China, 3468–3472, doi: 10.23919/ChiCC.2019.8865695.

[14] S. Abasi, M.A. Tehran, M.D. Fairchild, Colour metrics for image edge detection, Color Res. Appl. 45 (4) (2021) 632–643, https://doi.org/10.1002/col.22494.

[15] R. Tian, G.L. Sun, X.C. Liu, B.W. Zheng, Sobel edge detection based on weighted nuclear norm minimization image denoising, Electronics (Basel) 10 (6) (2021) 655, https://doi.org/10.3390/electronics10060655. Article Number.

[16] ARM Limited, Neon Intrinsics: Getting Started On Android User Guide (Online), 2020. Retrieved March 20, 2021, from https://developer.arm.com/solutions/os/android/developer-guides/neonintrinsics getting-started-on-android.

[17] IBM Corporation, Power ISA: Version 2.07B (Online), 2018. Retrieved March 23, 2021, from https://ibm.ent.box.com/s/jd5w15gz301s5b5dt375mshpq9c3lh4u.

[18] Intel Corporation, Intel® Architecture Instruction Set Extensions and Future Features (Online), 2021. Retrieved August 30, 2021, from, https://software.intel.com/content/dam/develop/external/us/en/documents-tps/architecture-instruction-set-extensions-programming-reference.pdf.

[19] R. Lasch, I. Oukid, R. Dementiev, N. May, S.S. Demirsoy, K.U. Sattler, Faster & strong: string dictionary compression using sampling and fast vectorized decompression, VLDB J 29 (2020) 1263–1285, https://doi.org/10.1007/s00778-020-00620-x.

[20] J.E. Zhang, A. Jacobson, M. Alexa, Fast updates for least-squares rotational alignment, Comput. Graph. Forum 40 (2) (2021) 13–22, https://doi.org/10.1111/cgf. 142611.

[21] P. Fortin, A. Fleury, F. Lemaire, M. Monagan, High-performance SIMD modular arithmetic for polynomial evaluation, Concurr. Comput. 33 (16) (2021), https://doi.org/10.1002/cpe.6270. Article number e6270.

[22] A.M. Hemeida, S.A. Hassan, S. Alkhalaf, M.M.M. Mahmoud, M.A. Saber, A. M. Bahaa Eldin, T. Senjyu, A.H. Alayed, Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor, Ain Shams Eng. J. 11 (4) (2020) 1179–1190, https://doi.org/10.1016/j.asej.2020.01.003.

[23] F. Andre, A.-.M. Kermarrec, N. Le Scouarnec, Quicker ADC: unlocking the hidden potential of product quantization with SIMD, IEEE Trans. Pattern Anal. Mach. Intell. 43 (5) (2021) 1666–16771, 10.1109/TPAMI.2019. 2952606.

[24] Hassan, S.A., Mahmoud, M.M., Hemeida, A.M., & Saber, M.A. (2018). Effective implementation of matrix-vector multiplication on intel's AVX multicore processor. *Comput. Lang. Syst. Struct.*, 51, 158–175, doi: 10.1016/j.cl.2017.06.003.

[25] Microsoft Corporation, Inline Assembler (Online), 2018. Retrieved August 28, 2021, from, https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler?view=. msvc-160.

[26] Amiri, H., & Shahbahrami, A. (2017). High performance implementation of 2D convolution using intel's advanced vector extensions, in: 2017 international symposium on artificial intelligence and signal processing (AISP2017), Shiraz, Iran, 25–30, doi:10.1109/AISP.2017.8324097.

[27] H. Amiri, A. Shahbahrami, SIMD programming using Intel vector extensions, J. Parallel Distrib. Comput. 135 (2020) 83–100, https://doi.org/10.1016/j.jpdc.2019.09.012.

[28] W.E. Liu, F. Wang, H.W. Zhou, Parallel seismic modeling based on OpenMP plus AVX and optimization strategy, J. Earth Sci. 30 (4) (2019) 843–848, https://doi.org/10.1007/s12583-018-0831-3.

[29] Intel Corporation, Intel Power Gadget, 2021. Retrieved from https://software.intel.com/content/www/us/en/develop/articles/intel-power-gadget.html.

[30] A. Alekhin, OpenCV Change Logs, 2020. Retrieved from, https://github.com/opencv/opencv/wiki/ChangeLog.

[31] Intel Corporation, Intel Integrated Performance Primitives, 2020, 2020, Retrieved from, https://software.intel.com/content/dam/develop/external/us/en/documents/ipp-userguide.pdf.

Thaufig Peng-o, was born in Thailand in 1991. He received his B.Eng. degree in computer engineering from faculty of engineering, Prince of Songkla university, Songkhla, Thailand, in 2013. He is currently a M.Eng. student in the Department of computer engineering, Prince of Songkla university. His main area of research is parallel processing. Email: 6210120056@psu.ac.th

Panyayot Chaikan, was born in Thailand in 1976. He received his B.Eng. and M.Eng, in computer engineering and electrical engineering from Faculty of Engineering, King Mongkhut's Institute of Technology Ladkrabang, Thailand, in 1999 and 2002 respectively. He obtained his Ph.D. degree in computer engineering in 2010 from Prince of Songkla University, Thailand. Currently, he has an assistant professor position in the Department of Computer Engineering, Faculty of Engineering. Prince of Songkla University, Thailand. His research interests include parallel processing, embedded systems, image processing and pattern recognition. Email: panyayot.c@psu.ac.th

ประวัติผู้เขียน

ชื่อ สกุล	นาย เฒาฟิก เพ็งโอ
รหัสประจำตัวนักศึกษา	6210120056

วุฒิการศึกษา

วุฒิ	ชื่อสถาบัน	ปีที่สำเร็จการศึกษา
วิศวกรรมศาสตรบัณฑิต (วิศวกรรมคอมพิวเตอร์)	มหาวิทยาลัยสงขลานครินทร์	2556

การตีพิมพ์เผยแพร่ผลงาน

1. T. Peng-o and P. Chaikan, "Optimization of Edge Detection using AVX Intrinsics on Multi-core Architectures," 2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), 2022, pp. 364-367, doi: 10.1109/ITC-CSCC55581.2022.9894947.
2. T. Peng-o and P. Chaikan, “High performance and energy efficient sobel edge detection”, *Microprocessors and Microsystems*, Volume 87, 2021, 104368, ISSN 0141-9331, https://doi.org/10.1016/j.micpro.2021.104368.