การพรีเฟ็ตช์ข้อมูลแบบพลวัตสำหรับการคูณเมทริกซ์ด้วยเมทริกซ์

Dynamic Data Prefetching for Matrix-Matrix Multiplication

วรินทร ข้อมงคลอุดม

Varintorn Khomongkonudom

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

การพรีเฟ็ตช์ข้อมูลแบบพลวัตสำหรับการคูณเมทริกซ์ด้วยเมทริกซ์

Dynamic Data Prefetching for Matrix-Matrix Multiplication

วรินทร ข้อมงคลอุดม

Varintorn Khomongkonudom

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

ชื่อวิทยานิพนธ์ การพรีเฟ็ตช์ข้อมูลแบบพลวัตสำหรับการคูณเมทริกซ์ด้วยเมทริกซ์

ผู้เขียน นาย วรินทร ข้อมงคลอุดม

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 .....
(นายวรินทร ข้อมงคลอุดม)
นักศึกษา

ข้าพเจ้าขอรับรองว่า ผลงานวิจัยนี้ไม่เคยเป็นส่วนหนึ่งในการอนุมัติปริญญาในระดับใดมาก่อน และไม่ได้ถูกใช้ในการยื่นขออนุมัติปริญญาในขณะนี้

ลงชื่อ .....

(นายวรินทร ข้อมงคลอุดม)

นักศึกษา

ชื่อวิทยานิพนธ์ การพรีเฟ็ตช์ข้อมูลแบบพลวัตสำหรับการคูณเมทริกซ์ด้วยเมทริกซ์

ผู้เขียน นาย วรินทร ข้อมงคลอุดม

สาขาวิชา วิศวกรรมคอมพิวเตอร์

ปีการศึกษา 2565

บทคัดย่อ

งานวิจัยนี้นำเสนอวิธีการพรีเฟ็ตช์ข้อมูลล่วงหน้าจากหน่วยความจำหลักมายังหน่วยความจำแคช ในการประมวลผลข้อมูลขนาดใหญ่หากมีการใช้คำสั่งพรีเฟ็ตช์ที่เหมาะสมจะสามารถช่วยลดระยะเวลาแฝงในการรอข้อมูลเพื่อใช้ในการประมวลผลลงได้ ในงานวิจัยนี้ได้นำเสนอแบบจำลองในการวิเคราะห์รูปแบบการพรีเฟ็ตช์ที่เหมาะสมที่สุดโดยใช้การคูณเมทริกซ์ด้วยเมทริกซ์เป็นกรณีศึกษา นอกจากนี้ยังได้นำเสนอวิธีการในการหาระยะทางในการพรีเฟ็ตช์ที่เหมาะสมกับทรัพยากรเครื่องที่ใช้เพื่อเพิ่มประสิทธิภาพในการประมวลผลให้แก่เครื่องคอมพิวเตอร์ที่มีทรัพยากรต่างกันได้ ผู้วิจัยได้ทำการทดลองบนเครื่องคอมพิวเตอร์ 2 เครื่อง ผลการทดสอบบนเครื่องที่ใช้หน่วยประมวลผล Core i-5 พบว่าโปรแกรมที่ใช้วิธีการพรีเฟ็ตช์ที่นำเสนอสามารถประมวลผลได้เร็วกว่าโปรแกรมที่ไม่ได้ใช้คำสั่งพรีเฟ็ตช์โดยเฉลี่ยอยู่ร้อยละ 18.86 และเร็วกว่าโปรแกรมการคูณเมทริกซ์ที่มีการแทรกคำสั่งพรีเฟ็ตช์อัตโนมัติโดยคอมไพเลอร์ Intel C++ โดยเฉลี่ยอยู่ร้อยละ 17.54 ผลการทดสอบบนเครื่องที่ใช้หน่วยประมวลผล Intel Core i7 พบว่าโปรแกรมที่ใช้วิธีการพรีเฟ็ตช์สามารถประมวลผลได้เร็วกว่าโปรแกรมที่ไม่ได้ใช้คำสั่งพรีเฟ็ตช์โดยเฉลี่ยอยู่ร้อยละ 8.86 และโปรแกรมการคูณเมทริกซ์ที่มีใส่คำสั่งพรีเฟ็ตช์อัตโนมัติจากคอมไพเลอร์ Intel C++ โดยเฉลี่ยอยู่ร้อยละ 7.73

Thesis Title	Dynamic Data Prefetching for Matrix-Matrix Multiplication
Author	Mr. Varintorn Khomongkonudom
Major Program	Computer Engineering
Academic Year	2022

ABSTRACT

This thesis presents a prefetching method for reading the data from main memory to the cache. When the data size is large, prefetching can reduce the memory waiting time and can reduce the execution time of a program. We propose an analyzing method in order to find the best prefetching pattern to augment the speed of matrix-matrix multiplication. We also propose a method to find the best prefetching distance that can be applied to different computers. Our algorithm was tested on 2 computers. When tested on the Core-i5 machine, our proposed prefetching was 18.86 percent faster than a program that did not use prefetching instructions. Our method was 17.54 percent faster than the automatic prefetching generated by the Intel C++ compiler. When tested on the Core-i7 machine, our prefetching was 8.86 percent faster than without prefetching, and was 7.73 percent faster than automatic prefetching.

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้สมบูรณ์และสำเร็จได้จากความกรุณาและความเมตตาจากอาจารย์ที่ปรึกษาวิทยานิพนธ์ ผู้ช่วยศาสตราจารย์ ดร.ปัญญยศ ไชยกาศ ที่ท่านคอยให้คำปรึกษา มอบความรู้และเทคนิคในการเขียนโปรแกรมรวมถึงทำวิจัยมากมายตั้งแต่ระดับปริญญาตรี อีกทั้งมอบคำแนะนำที่เป็นประโยชน์ ตลอดถึงช่วยตรวจและแก้ไขวิทยานิพนธ์ให้ถูกต้องสมบูรณ์ และมอบแรงบันดาลใจให้แก่ข้าพเจ้าตลอดระยะเวลาที่ข้าพเจ้าศึกษา ข้าพเจ้าขอบพระคุณอาจารย์เป็นอย่างสูง

ขอขอบคุณคณะกรรมการสอบป้องกันวิทยานิพนธ์ และคณาจารย์ทุกท่านที่ช่วยให้คำแนะนำสำหรับปรับปรุง แก้ไขและตรวจสอบความถูกต้องเพื่อให้วิทยานิพนธ์ฉบับนี้มีความสมบูรณ์ อีกทั้งยังมอบความรู้และคำปรึกษาเกี่ยวกับเทคโนโลยีและการทดลองต่าง ๆ

ขอขอบคุณพี่ ๆ และเพื่อน ๆ นักศึกษาระดับปริญญาโท หลักสูตรวิศวกรรมศาสตรมหาบัณฑิต สาขาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ มหาวิทยาลัยสงขลานครินทร์ ที่สนับสนุนการทำวิทยานิพนธ์ฉบับนี้ อีกทั้งคอยให้กำลังใจข้าพเจ้าตลอดระยะเวลาการศึกษา

สุดท้ายนี้ ข้าพเจ้ากราบขอบพระคุณบิดา มารดา และครอบครัวของข้าพเจ้าที่เลี้ยงดูส่งเสริมให้ข้าพเจ้าได้มีโอกาสศึกษา จวบจนข้าพเจ้าประสบความสำเร็จได้ถึงทุกวันนี้

วรินทร ข้อมงคลอุดม

สารบัญ

สารบัญ (ต่อ)

หน้า

รายการตาราง

หน้า

รายการภาพประกอบ

หน้า

บทที่ 1
บทนำ

1.1. ที่มาและความสำคัญของการวิจัย

การประมวลผลข้อมูลจำนวนมากต้องใช้เวลาในการประมวลผลเพิ่มขึ้นไปด้วย ในแต่ละโปรแกรมที่ใช้ในการประมวลผลจะมีรูปแบบการเข้าถึงหน่วยความจำที่แตกต่างกัน ดังนั้น หากเราสามารถอ่านข้อมูลจากหน่วยความจำล่วงหน้าได้อย่างถูกต้อง เวลาในการประมวลผลโดยรวมของโปรแกรมจะลดลงได้

การคูณเมทริกซ์ เป็นการดำเนินการทางคณิตศาสตร์พื้นฐานที่จำเป็นต้องใช้ในงานทางด้านวิทยาศาสตร์และวิศวกรรม หากเมทริกซ์มีขนาดใหญ่จะเสียเวลาในการดำเนินการมาก เราสามารถเพิ่มความเร็วในการทำงานได้หลายวิธี เช่น การเร่งความเร็วด้านฮาร์ดแวร์ [1] การปรับการทำงานของซอฟต์แวร์ และการผสมผสานการเร่งความเร็วด้วยฮาร์ดแวร์และการปรับการทำงานของซอฟต์แวร์ควบคู่กัน ในตัวเลือกดังกล่าวจะเห็นว่าทางเลือกที่เสียค่าใช้จ่ายน้อยที่สุด คือ การปรับการทำงานของซอฟต์แวร์ การปรับการทำงานของซอฟต์แวร์ในการคูณเมทริกซ์ สามารถดำเนินการได้หลายวิธี เช่น การปรับเปลี่ยนอัลกอริทึมการคูณ การลดจำนวนโอเปอเรชันทางคณิตศาสตร์ การประมวลผลแบบขนาน การคลายการวนซ้ำ การใช้ประโยชน์จากชุดคำสั่งที่สามารถประมวลผลได้หลายข้อมูลภายในหนึ่งคำสั่ง (Single Instruction Multiple Data) เป็นต้น [2] นอกจากนี้ยังมีอีกวิธีหนึ่งที่สามารถนำมาใช้เพิ่มความเร็วในการทำงานได้ คือ การพรีเฟ็ตช์ข้อมูลล่วงหน้า (Data Prefetching) [3] โดยในขณะที่หน่วยประมวลผลใช้เวลาในการประมวลผล หากมีกระบวนการอ่านข้อมูลล่วงหน้าจากหน่วยความจำหลักเข้ามาเก็บไว้ในหน่วยความจำแคชก่อน จะส่งผลให้หน่วยประมวลผลไม่จำเป็นต้องเสียเวลาในการรอข้อมูลและทำให้ประมวลผลไปได้อย่างต่อเนื่อง อีกทั้งการประมวลผลข้อมูลที่มีปริมาณมาก มักมีรูปแบบการทำงานที่ซ้ำกันอย่างมีแบบแผน ดังนั้นหากเราสามารถคำนวณค่าตำแหน่งถัดไปในการอ่านข้อมูลล่วงหน้าได้อย่างเหมาะสมและทันเวลา ซึ่งจะสามารถลดระยะเวลาในการประมวลผลในภาพรวมลงได้

การพรีเฟ็ตช์เป็นกระบวนการอ่านข้อมูลล่วงหน้าจากหน่วยความจำหลักมาเก็บไว้ในแต่ละลำดับชั้นของหน่วยความจำแคช ซึ่งกระบวนการนี้สามารถนำไปใช้ได้ทั้งฮาร์ดแวร์และซอฟต์แวร์ การพรีเฟ็ตช์ด้วยฮาร์ดแวร์หรือฮาร์ดแวร์พรีเฟ็ตช์สามารถทำงานได้ดีเมื่อพฤติกรรมการเข้าถึงข้อมูลของโปรแกรมชัดเจนและคาดเดาได้ง่าย [4] โดยจะวิเคราะห์แนวโน้มของการใช้ข้อมูล

จากนั้นคาดการณ์และอ่านข้อมูลล่วงหน้าจากหน่วยความจำหลักไปยังหน่วยความจำแคชเพื่อใช้ในอนาคต การพรีเฟ็ตช์ด้วยซอฟต์แวร์หรือซอฟต์แวร์พรีเฟ็ตช์ทำงานโดยแทรกคำสั่งพิเศษที่เป็นคำสั่งระดับเครื่องลงไปในโปรแกรม คำสั่งที่ใช้ในการพรีเฟ็ตช์จะอ่านข้อมูลจากหน่วยความจำหลักไปใส่ยังหน่วยความจำแคช ซึ่งอนุมานว่าจะถูกใช้ในอนาคตอันใกล้ ซอฟต์แวร์พรีเฟ็ตช์มี 2 แบบ คือ ซอฟต์แวร์พรีเฟ็ตช์อัตโนมัติและซอฟต์แวร์พรีเฟ็ตช์ไม่อัตโนมัติ ซอฟต์แวร์พรีเฟ็ตช์อัตโนมัติจะถูกทำโดยคอมไพเลอร์ คอมไพเลอร์จะวิเคราะห์พฤติกรรมการใช้ข้อมูลของโปรแกรม จากนั้นแทรกคำสั่งเครื่องที่เป็นคำสั่งพรีเฟ็ตช์ภายในโปรแกรมโดยอัตโนมัติ คอมไพเลอร์บางตัวรองรับคุณสมบัตินี้ เช่น คอมไพเลอร์ Intel C เป็นต้น ซอฟต์แวร์พรีเฟ็ตช์ไม่อัตโนมัติจะถูกทำโดยผู้เขียนโปรแกรมเป็นคนวิเคราะห์และใส่คำสั่งพรีเฟ็ตช์ลงในโค้ดที่เป็นภาษาระดับสูงของโปรแกรม หากพรีเฟ็ตช์ในรูปแบบหรือตำแหน่งที่ไม่เหมาะสมอาจทำให้เวลาในการประมวลผลเพิ่มขึ้นได้ จากการค้นคว้าพบว่ามีงานวิจัยที่ศึกษาหาระยะทางเพื่อการอ่านข้อมูลเข้ามาเก็บยังหน่วยความจำแคชล่วงหน้าให้ทันเวลาก่อนการประมวลผล ซึ่งได้นำเสนอสมการคำนวณอย่างง่าย [5] เพื่อปรับระยะทางการพรีเฟ็ตช์ให้เหมาะสมกับทรัพยากรเครื่อง แต่สมการดังกล่าวสามารถใช้งานได้จริงในช่วงแรกเท่านั้น เมื่อมีการแทรกคำสั่งพรีเฟ็ตช์ภายในการวนรอบจะทำให้จำนวน clock cycles เปลี่ยนไปและอาจส่งผลให้การพรีเฟ็ตช์ไม่อยู่ในสถานะตรงเวลาเมื่อรอบที่วนซ้ำมีจำนวนมาก อีกทั้งมีงานวิจัยที่ออกมาแก้ปัญหาดังกล่าวโดยนำเสนอ framework ที่ใช้ในการสร้างเธรดตัวช่วยมาตรวจสอบและคำนวณค่าเพื่อตัดสินใจใช้คำสั่งพรีเฟ็ตช์ในตำแหน่งที่เหมาะสม [6] แต่ framework ดังกล่าวต้องอาศัยฮาร์ดแวร์ตัวช่วยในการสร้างเธรด

ดังนั้นงานวิจัยนี้ได้มีแนวคิดที่จะนำหลักการของซอฟต์แวร์พรีเฟ็ตช์มาช่วยเร่งความเร็วของการคูณเมทริกซ์ด้วยเมทริกซ์ แต่เนื่องจากการพรีเฟ็ตช์ที่มีประสิทธิภาพจะต้องมีรูปแบบการพรีเฟ็ตช์และการระบุค่าระยะทางในการพรีเฟ็ตช์อย่างเหมาะสม ซึ่งมีค่าแตกต่างกันไปสำหรับการทำงานบนแต่ละเครื่องคอมพิวเตอร์ ดังนั้น จึงมีแนวคิดที่จะนำเสนอหลักการหารูปแบบการพรีเฟ็ตช์และการคำนวณหาระยะทางของการพรีเฟ็ตช์ที่เหมาะสมกับทรัพยากรเครื่องที่ใช้ในการประมวลผลอันจะช่วยให้สามารถใช้การพรีเฟ็ตช์ช่วยเพิ่มสมรรถนะการทำงานได้อย่างเหมาะสมบนเครื่องคอมพิวเตอร์ที่มีค่าพารามิเตอร์ของหน่วยความจำรูปแบบต่างกันได้

1.2. วัตถุประสงค์ของการวิจัย

1.2.1 เพื่อนำเสนอแบบจำลองในการวิเคราะห์หาวิธีการพรีเฟ็ตช์

1.2.2 เพื่อนำเสนอแบบจำลองสำหรับการคำนวณค่าตำแหน่งของหน่วยความจำที่ใช้ในการพรีเฟ็ตช์ ซึ่งสามารถทำงานบนหน่วยประมวลผลหรือหน่วยความจำที่มีลักษณะต่างกันได้

1.3. ประโยชน์ที่คาดว่าจะได้รับจากการวิจัย

1.3.1 สามารถลดระยะเวลาในการประมวลผลในภาพรวมลงได้อย่างมีประสิทธิภาพสืบเนื่องจากค่าเวลาที่ตัวประมวลผลจะต้องรอในการอ่านค่าจากหน่วยความจำลดลง

1.3.2 แบบจำลองสามารถปรับค่าระยะทางในการพรีเฟ็ตช์ให้เหมาะสมกับทรัพยากรเครื่องได้

1.4. ขอบเขตของงานวิจัย

1.4.1 ศึกษาและเปรียบเทียบบนเครื่องคอมพิวเตอร์ส่วนบุคคลทั่วไปที่รองรับชุดคำสั่งที่ใช้ในการพรีเฟ็ตช์ของ Intel Intrinsics

1.4.2 ศึกษาและเปรียบเทียบโปรแกรมที่ได้จากการเขียนภาษาซี

1.4.3 โปรแกรมสามารถปรับค่าระยะทางในการพรีเฟ็ตช์ให้เหมาะสมกับทรัพยากรเครื่องได้

1.5. ภาพรวมของงานวิจัย

งานวิจัยฉบับนี้มีจุดประสงค์เพื่อสร้างซอฟต์แวร์พรีเฟ็ตช์สำหรับโปรแกรมการคูณเมทริกซ์ที่สามารถปรับระยะทางในการพรีเฟ็ตช์ได้อย่างเหมาะสมกับทรัพยากรเครื่อง จากนั้นจะถูกนำมาเปรียบเทียบประสิทธิภาพในด้านความเร็วในการประมวลผลกับโปรแกรมการคูณเมทริกซ์ที่มีการพรีเฟ็ตช์อัตโนมัติจาก Intel C++ คอมไพเลอร์ที่ผู้วิจัยใช้เป็นมาตรฐานอ้างอิง ส่วนประกอบงานวิจัยครั้งนี้ประกอบไปด้วยขั้นตอนหลักทั้งสิ้น 4 ขั้นตอนดังต่อไปนี้

1) การปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยการปรับซอฟต์แวร์
2) การทดลองปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยซอฟต์แวร์พรีเฟ็ตช์
3) การหาระยะทางของการพรีเฟ็ตช์ที่เหมาะสม
4) การทดสอบประสิทธิภาพของโปรแกรมการคูณเมทริกซ์ที่นำเสนอบนเครื่องที่มีทรัพยากรต่างกัน

1.6. อุปกรณ์ที่ใช้ในการวิจัย

งานวิจัยฉบับนี้ ผู้วิจัยได้ทำการทดลองทั้งหมดด้วยคอมพิวเตอร์ส่วนตัวของผู้วิจัยเอง คุณสมบัติของคอมพิวเตอร์ที่ใช้ในการวิจัยครั้งนี้มีรายละเอียดดังตารางที่ 1-1

ตารางที่ 1-1 คุณสมบัติของคอมพิวเตอร์ที่ใช้ในการวิจัย

อุปกรณ์	เครื่องที่ 1	เครื่องที่ 2
หน่วยประมวลผลกลาง (CPU)	Intel Core i5-6200U CPU 2.30 GHz	Intel Core i7-9700K CPU 3.60 GHz
หน่วยประมวลผลกราฟิกส์ (GPU)	NVIDIA GeForce 940M	NVIDIA GeForce GTX 1660 SUPER
หน่วยความจำ (RAM)	DDR3 16 GB	DDR4 16 GB
ฮาร์ดดิสก์	SSD 500 GB	SSD 1 TB
หน่วยความจำแคช L1	2x32 Kbytes 8-way Set associative, 64-byte line size	8x32 Kbytes 8-way Set associative, 64-byte line size
หน่วยความจำแคช L2	2x256 Kbytes 4-way Set associative, 64-byte line size	8x256 Kbytes 4-way Set associative, 64-byte line size
หน่วยความจำแคช L3	3 Mbytes 12-way Set associative, 64-byte line size	12 Mbytes 12-way Set associative, 64-byte line size
ระบบปฏิบัติการ	Windows 10 64-bit (Pro)	Windows 10 64-bit (Pro)

บทที่ 2
ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

2.1. ทฤษฎีและหลักการ

การวิจัยและพัฒนาการลดระยะเวลาในการประมวลผลโดยใช้ซอฟต์แวร์พรีเฟ็ตช์เพื่ออ่านข้อมูลจากหน่วยความจำล่วงหน้า จะต้องศึกษาทฤษฎีและหลักการที่เกี่ยวข้อง ดังหัวข้อต่อไปนี้

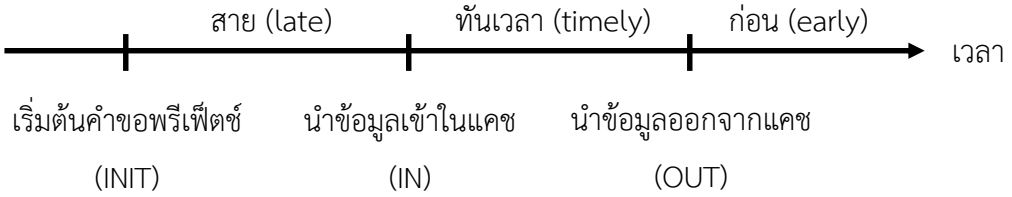
2.1.1 หน่วยความจำแคช

หน่วยความจำแคช [3] คือ หน่วยความจำที่ทำหน้าที่เก็บข้อมูลเพื่อให้หน่วยประมวลผลสามารถเข้าถึงข้อมูลได้อย่างรวดเร็วโดยข้อมูลต่าง ๆ ภายในแคชจะถูกอ่านเข้ามาจากหน่วยความจำหลักที่มีความเร็วในการอ่านหรือเขียนข้อมูลต่ำกว่า การอ่านข้อมูลเข้ามาเก็บภายในแคชจะถูกอ่านมาเป็นบล็อกหรือบรรทัดที่มีขนาดเป็นจำนวนเท่าของ word ซึ่งในแต่ละสถาปัตยกรรมจะพบลำดับชั้นของแคชที่ต่างกัน เช่น สถาปัตยกรรม Skylake ใน Intel จะมีแคชจำนวน 3 ระดับชั้น ได้แก่ L1 L2 และ L3 โดยในแต่ละระดับชั้นจะมีความจุที่ต่างกัน ซึ่งระดับชั้นที่ใกล้หน่วยประมวลผลมากที่สุดจะมีความจุน้อยที่สุดด้วย ในการทำงานของแคชจะประกอบไปด้วย 2 สถานะ ได้แก่ cache hit และ cache miss โดยสถานะ cache hit จะเกิดขึ้นในกรณีที่หน่วยประมวลผลต้องการเข้าถึงข้อมูล ซึ่งมีข้อมูลอยู่ในหน่วยความจำแคช และสถานะ cache miss จะเกิดขึ้นเมื่อหน่วยประมวลผลต้องการเข้าถึงข้อมูลที่ไม่มีอยู่ในหน่วยความจำแคช ส่งผลให้เสียเวลาในการรอข้อมูลที่จะใช้ประมวลผลนั่นเอง

2.1.2 หลักการพื้นฐานของกระบวนการพรีเฟ็ตช์

การพรีเฟ็ตช์ [7] คือ กระบวนการอ่านข้อมูลล่วงหน้าจากหน่วยความจำหลักมาเก็บไว้ในหน่วยความจำแคชแต่ละลำดับชั้น กระบวนการพรีเฟ็ตช์มีทั้งที่เกิดจากฮาร์ดแวร์และซอฟต์แวร์ ซึ่งการพรีเฟ็ตช์ด้วยฮาร์ดแวร์จะทำงานได้ดีในกรณีที่การอ่านข้อมูลที่มีพฤติกรรมชัดเจน ส่วนการพรีเฟ็ตช์ด้วยซอฟต์แวร์จะทำงานได้ดีในการอ่านช่วงข้อมูลที่สั้นและข้อมูลที่เป็น indirect references

ซึ่งต้องคำนึงถึงระยะตำแหน่งของการดึงข้อมูลที่เหมาะสม โดยสถานะของการเกิดกระบวนการพรีเฟ็ตช์ ดังแสดงผังเส้นเวลาในรูปที่ 2-1



รูปที่ 2-1 สถานการณ์เกิดกระบวนการพรีเฟ็ตช์ [7]

จากรูปที่ 2-1 แสดงถึงสถานะของการเกิดกระบวนการพรีเฟ็ตช์ในช่วงเวลาต่าง ๆ ที่หน่วยประมวลผลกลางจะดึงข้อมูลไปใช้ โดยแบ่งออกเป็น 3 สถานะ ได้แก่

1) สถานะสายเกิดขึ้นเมื่อหน่วยประมวลผลต้องการดึงข้อมูลจากหน่วยความจำแคชแต่ข้อมูลที่ต้องการนั้นยังไม่ถูกนำเข้ามาภายในแคช

2) สถานะตรงเวลาเกิดขึ้นเมื่อหน่วยประมวลผลต้องการดึงข้อมูลจากหน่วยความจำแคชซึ่งข้อมูลนั้นมีอยู่ในแคช

3) สถานะก่อนเกิดขึ้นเมื่อหน่วยประมวลผลต้องการดึงข้อมูลจากหน่วยความจำแคชแต่ข้อมูลนั้นถูกนำออกจากแคชไปแล้ว

จะเห็นได้ว่าการเกิดสถานะที่ดีที่สุดในกระบวนการพรีเฟ็ตช์ คือ สถานะทันเวลา ซึ่งจะทำให้หน่วยประมวลผลไม่จำเป็นต้องรอข้อมูลที่ต้องการ และสามารถประมวลผลได้อย่างต่อเนื่อง

2.1.3 คำสั่งที่ใช้ในการพรีเฟ็ตช์

การอ่านข้อมูลเข้ามายังหน่วยความจำแคชหนึ่งบรรทัดจะถูกอ่านมาเป็นบล็อกของข้อมูลจากหน่วยความจำ โดยแต่ละบรรทัดจะมีขนาด *m* ไบต์ ซึ่งใน CPU Intel Core i5 6200U มี *m* เท่ากับ 64 ไบต์ หมายถึงจะสามารถอ่านข้อมูลจากหน่วยความจำหลักเข้ามายังหน่วยความจำแคชทีละ 64 ไบต์

คำสั่งเครื่องในการอ่านข้อมูลจากหน่วยความจำหลักเข้ามายังหน่วยความจำแคชของสถาปัตยกรรมที่ใช้หน่วยประมวลผล Intel [10] ประกอบด้วย 4 คำสั่งตามลักษณะการอ่านเพื่อเก็บข้อมูลในแคชระดับต่าง ๆ ดังนี้

1) PREFETCHT0 เป็นคำสั่งที่ทำหน้าที่อ่านข้อมูลเข้ามาเก็บไว้ยังหน่วยความจำแคชทุกระดับ โดยระบุเป็น hint 1

2) PREFETCHT1 เป็นคำสั่งที่ทำหน้าที่อ่านข้อมูลเข้ามาเก็บไว้ยังหน่วยความจำแคชระดับที่ 2 หรือสูงกว่า คำสั่งนี้เหมาะแก่การใช้กรณีที่มีการใช้ข้อมูลเดิมซ้ำในหน่วยความจำแคชระดับที่ 1 ทำให้สามารถอ่านข้อมูลล่วงหน้าเข้ามาเก็บไว้ในแคชระดับอื่นได้ โดยระบุเป็น hint 2

3) PREFETCHT2 เป็นคำสั่งที่ทำหน้าที่อ่านข้อมูลเข้ามาเก็บไว้ยังหน่วยความจำแคชระดับที่ 3 หรือสูงกว่า คำสั่งนี้เหมาะแก่การใช้กรณีที่มีการใช้ข้อมูลเดิมซ้ำในหน่วยความจำแคชระดับที่ 1 และ 2 ทำให้สามารถอ่านข้อมูลล่วงหน้าเข้ามาเก็บไว้ในแคชระดับอื่นได้ โดยระบุเป็น hint 3

4) PREFETCHNTA เป็นคำสั่งที่ทำหน้าที่อ่านข้อมูลเข้ามาเก็บไว้ยังหน่วยความจำแคชระดับที่ 1 เพียงอย่างเดียว คำสั่งนี้เหมาะแก่การใช้ในกรณีที่ไม่มีการใช้ข้อมูลเดิมซ้ำทำให้ไม่จำเป็นต้องเก็บข้อมูลไว้ในแคชระดับอื่นโดยระบุเป็น hint 0

คำสั่งเครื่องทั้ง 4 คำสั่งนี้สามารถเรียกใช้ได้โดย Intel Intrinsics ซึ่งเป็นเครื่องมืออ้างอิงสำหรับคำสั่งภายในของ Intel ที่มีรูปแบบการเขียนภาษาซี ทำให้ผู้พัฒนาไม่ต้องเขียนภาษาแอสเซมบลีในการเรียกใช้ คำสั่งที่ใช้ในการพรีเฟ็ตช์คือ _mm_prefetch(char const* *p*, int *i*) โดยคำสั่งนี้จะอ่านข้อมูลจากบรรทัดของข้อมูลในหน่วยความจำหลักที่มีตำแหน่ง *p* อยู่และอ่านเข้ามายังแคชแต่ละระดับตามที่ระบุ *i* เช่น _mm_prefetch(*x*, 1) เป็นการพรีเฟ็ตช์ข้อมูลบรรทัดที่มีตำแหน่ง *x* อยู่ โดยอ่านเข้ามายังแคชทุกระดับ เป็นต้น

2.1.4 การคลายการวนซ้ำ

การเพิ่มประสิทธิภาพให้แก่อัลกอริทึมที่มีการวนรอบเกิดขึ้น สามารถทำได้โดยใช้เทคนิคคลายการวนซ้ำ (Loop unrolling) [9] เพื่อลดจำนวนการวนรอบลงแต่เพิ่มจำนวนคำสั่งต่อการวนรอบขึ้น ส่งผลให้ลดค่าโสหุ้ยที่เกิดจากการเปลี่ยนตำแหน่งของ Program Counter ลงได้ การคลายการวนซ้ำมีตัวอย่างการเขียนโปรแกรมดังรูปที่ 2-2

โปรแกรมที่ไม่ใช้เทคนิคการคลายการวนซ้ำ	โปรแกรมที่ใช้เทคนิคคลายการวนซ้ำ

for (int *i* = 0 ; *i* < *M* ; *i* ++) {
result[*i*] = *a*[*i*] + *b*[*i*];
}
	for (int *i* = 0 ; *i* < *M* ; *i* + 4) {
	result[*i*] = *a*[*i*] + *b*[*i*];
	result[*i*+1] = *a*[*i*+1] + *b*[*i*+1];
	result[*i*+2] = *a*[*i*+2] + *b*[*i*+2];
	result[*i*+3] = *a*[*i*+3] + *b*[*i*+3];
	}

รูปที่ 2-2 เปรียบเทียบโปรแกรมที่ใช้ และ ไม่ใช้เทคนิคการคลายการวนซ้ำ

จากรูปที่ 2-2 เป็นการเขียนโปรแกรมการบวกค่าจากอาเรย์ *a* และอาเรย์ *b* ที่ตำแหน่ง *i* โดยเก็บผลลัพธ์ลงในตัวแปร *result* ที่เป็นอาเรย์เช่นกัน จะเห็นได้ว่าในโปรแกรมที่ใช้เทคนิคคลายการวนซ้ำมีคำสั่งเพิ่มขึ้นภายในหนึ่งการวนรอบ แต่จะมีรอบในการวนน้อยลง คิดเป็นจำนวน *M*/4 รอบเนื่องจากตัวแปร *i* มีการเพิ่มค่าทีละ 4 ต่างจากโปรแกรมเดิมที่มีการวน *M* รอบ ทำให้ Program Counter ที่ต้องเปลี่ยนตำแหน่งมีจำนวนที่น้อยลงนั่นเอง

การใช้ซอฟต์แวร์พรีเฟ็ตช์ควบคู่กับการทำ loop unrolling ช่วยลดจำนวนครั้งของคำสั่งในการอ่านข้อมูลล่วงหน้าลงได้ เนื่องจากการพรีเฟ็ตช์หนึ่งครั้งจะสามารถอ่านข้อมูลได้ความจุหนึ่งบรรทัดของหน่วยความจำแคช ซึ่งการประมวลผลบางอัลกอริทึมในหนึ่งการวนรอบใช้ข้อมูลที่อ่านมาไม่เต็มประสิทธิภาพ การทำ loop unrolling ที่มีจำนวนครั้งในการคลายรอบที่เหมาะสมจึงสามารถช่วยให้การวนซ้ำหนึ่งรอบสามารถใช้ข้อมูลที่ได้จากการพรีเฟ็ตช์ได้อย่างมีประสิทธิภาพ อีกทั้งจำนวนรอบการวนซ้ำที่ลดลงจากการทำ loop unrolling ยังส่งผลให้ใช้คำสั่งในการพรีเฟ็ตช์ลดลงอีกด้วย

2.2. งานวิจัยที่เกี่ยวข้อง

ที่ผ่านมามีงานวิจัยต่าง ๆ พยายามเพิ่มความเร็วให้แก่การประมวลผลด้วยวิธีการพรีเฟ็ตช์ ซึ่งมีเทคนิคเพื่อลดปัญหาและเพิ่มประสิทธิภาพที่เกิดจากการใช้ซอฟต์แวร์พรีเฟ็ตช์ ผู้วิจัยจึงได้ศึกษางานวิจัยที่เกี่ยวกับการเขียนซอฟต์แวร์พรีเฟ็ตช์ให้มีประสิทธิภาพดังต่อไปนี้

2.2.1 การใช้ซอฟต์แวร์พรีเฟ็ตช์ควบคู่กับฮาร์ดแวร์พรีเฟ็ตช์

งานวิจัย [7] ได้เสนอวิธีการสำหรับการใช้ซอฟต์แวร์พรีเฟ็ตช์ควบคู่กับฮาร์ดแวร์พรีเฟ็ตช์ให้มีประสิทธิภาพ เพื่อลดอัตราการเกิด cache miss โดยใช้การวิเคราะห์ระดับ source code เพื่อดูพฤติกรรมการทำงานของ compiler และได้วัดผลการทำงานร่วมกันระหว่างซอฟต์แวร์และฮาร์ดแวร์พรีเฟ็ตช์ในแต่ละมาตรฐานอ้างอิง ซึ่งมีผลสรุปดังตารางที่ 2-2

ตารางที่ 2-2 ผลการทดลองการทำงานร่วมกันของซอฟต์แวร์และฮาร์ดแวร์พรีเฟ็ตช์ [7]

Benchmark		Data structure	GHB+T	STR+T	Why SW is better	Why SW is worse	Best
POS	433.milc	Short array	-	-	Short array		SW+GHB
	456.Gems FDTD	Indirect, stride	-	+	Indirect access		SW+STR+T
	429.mcf	Recursive data structures, Array of pointers	-	-	Irregular (Array of pointers)		SW+STR
	470.lbm	Stride	-	-	Reduction in L1 misses		SW+STR+T
	462.libquantum	Stride	-	-	Reduction in L1 misses		SW+STR+T
	403.gcc	Recursive data structures, indirect	-	-	Indirect access		SW+STR
	434.zeusmp	Stride	+	-	Reduction in L1 misses		SW+GHB+T
NEU	434.bzip2	Indirect, hash	+	+			SW+GHB+T
	436.cactusADM	Stride	+	-			SW+GHB+T
	436.soplex	Indirect	-	-			STR
	410.bwaves	Stream	-	-			STR
NEG	482.sphinx3	Stride	+	+		Loop is too short	SW+STR
	347.leslis3d	Stream	-	-		Loop is too short	SW+STR
+: positive, -:negative, Best: best performing case among all eveluated cases							

จากตารางที่ 2-2 เป็นการวิเคราะห์ผลการทดลองวัดผลประสิทธิภาพการทำงานร่วมกันของฮาร์ดแวร์และซอฟต์แวร์พรีเฟ็ตช์ในการดึงข้อมูลล่วงหน้า ในตารางประกอบไปด้วยการทดลองที่นำเอาซอฟต์แวร์พรีเฟ็ตช์ (SW) นำไปใช้ร่วมกับฮาร์ดแวร์พรีเฟ็ตช์ต่างและเทคนิคต่าง ๆ ได้แก่ Global history buffer (GHB) Stride prefetcher (STR) และ training algorithm (T) โดยผลลัพธ์ได้แบ่งเกณฑ์มาตรฐานออกเป็นกลุ่มเชิงบวก เป็นกลางและเชิงลบตามประสิทธิภาพการดึงข้อมูลล่วงหน้า ซึ่งจะบอกถึงข้อดีและข้อเสียที่เกิดจากการใช้ซอฟต์แวร์พรีเฟ็ตช์ร่วมกับฮาร์ดแวร์พรีเฟ็ตช์ การใช้ซอฟต์แวร์พรีเฟ็ตช์ในการวนรอบที่น้อยเกินไปจะทำให้ระยะทางการพรีเฟ็ตช์สั้นเกินไป ส่งผลให้ข้อมูลที่อ่านเข้ามายังหน่วยความจำแคชเกิดสถานะ late และทำให้การพรีเฟ็ตช์ไม่มีประสิทธิภาพส่งผลลบในภาพรวม ดังนั้นระยะของการดึงข้อมูลล่วงหน้าเป็นส่วนสำคัญในการใช้ซอฟต์แวร์พรีเฟ็ตช์ โดยการดึงข้อมูลต้องมีระยะการดึงข้อมูลที่เหมาะสม อีกทั้งการดึงข้อมูลล่วงหน้าโดยใช้ฮาร์ดแวร์พรีเฟ็ตช์ไม่สามารถดึงข้อมูลช่วงสั้น ข้อมูลที่เป็น Indirect references และสายข้อมูลจำนวนมากพร้อมกันได้ ดังนั้นเราต้องใช้ซอฟต์แวร์พรีเฟ็ตช์เข้ามาช่วย ผู้วิจัยจึงได้มีแนวคิดเพื่อเสนอแบบจำลองที่ใช้ซอฟต์แวร์พรีเฟ็ตช์ ซึ่งต้องคำนึงถึงระยะทางในการพรีเฟ็ตช์เป็นสำคัญ

2.2.2 การเพิ่มประสิทธิภาพในการประมวลผล Matrix Vector Multiplication

งานวิจัย [10] ได้เสนอวิธีการเพิ่มประสิทธิภาพในการประมวลผล Matrix Vector Multiplication โดยการทดลองมีการเปรียบเทียบระหว่างการเขียนโปรแกรมแบบ Inline Assembly และ Intrinsic Function ผลการทดลองพบว่าการเขียนโปรแกรมคำนวณสมการ $y=Ax$ และ $y=A^Tx$ ที่ใช้ Intrinsic functions มีประสิทธิภาพมากกว่าการเขียนโปรแกรมแบบ Inline Assembly โดยเฉลี่ยร้อยละ 20.25 และร้อยละ 30.15 ตามลำดับ

นอกจากนั้นในบทความยังเสนอวิธีการเพิ่มประสิทธิภาพในการเขียนโปรแกรมด้วยวิธีการต่าง ๆ ได้แก่ การทำพรีเฟ็ตช์ การทำ loop unrolling การทำ blocking และการใช้ OpenMP การเพิ่มประสิทธิภาพด้วยวิธีการทำพรีเฟ็ตช์ควบคู่กับการทำ loop unrolling ช่วยลดจำนวนครั้งของคำสั่งในการอ่านข้อมูลล่วงหน้าลงเนื่องจากจำนวนการวนรอบที่ลดลง ทำให้หลีกเลี่ยงการอ่านข้อมูลล่วงหน้าซ้ำในตำแหน่งเดิม อีกทั้งการทำ blocking ยังช่วยลดขนาดของเมทริกซ์ลงทำให้อัตราการเกิด cache miss ลดลง ผลการทดลองพบว่าการเขียนโปรแกรมคำนวณสมการ $y=Ax$ และ $y=A^Tx$ ที่ได้มีการเพิ่มประสิทธิภาพด้วยวิธีข้างต้น มีประสิทธิภาพมากกว่าการเขียนโปรแกรมแบบเดิมโดยเฉลี่ย 18.2% และ 14.1% ตามลำดับ ผู้วิจัยจึงได้ออกแบบงานวิจัยให้มีการปรับปรุงประสิทธิภาพโปรแกรมโดยปรับอัลกอริทึมก่อน เพื่อส่งเสริมการใช้ซอฟต์แวร์พรีเฟ็ตช์เข้าไปในการทดลองถัดไป

2.2.3 การหาระยะทางพรีเฟ็ตช์อย่างง่าย

งานวิจัย [5] ได้เสนอวิธีการพรีเฟ็ตช์เพื่อเพิ่มความเร็วในการประมวลผล โดยนำเสนอการคำนวณระยะทางในการพรีเฟ็ตช์อย่างง่าย เพื่อปรับค่าระยะทางในการพรีเฟ็ตช์ที่เหมาะสมกับทรัพยากรเครื่อง ดังในสมการที่ (1)

$$d = L_m / C_{pi} \quad (1)$$

เมื่อ d คือ ระยะทางในการพรีเฟ็ตช์มีหน่วยเป็นไบต์ L_m คือระยะเวลาเฉลี่ยที่ใช้ในการอ่านข้อมูลเข้ามายังหน่วยความจำแคชเมื่อเกิดสถานะ cache miss ขึ้น และ C_{pi} คือค่า clock cycles ของการทำงานต่อหนึ่งการวนรอบ

2.2.4 การใช้ Trident frame work เพื่อหาระยะทางในการพรีเฟ็ตช์

งานวิจัย [6] กล่าวถึงสมการที่ (1) ว่าสามารถใช้งานได้จริงในช่วงแรกเท่านั้น เมื่อมีการแทรกคำสั่งพรีเฟ็ตช์ภายในการวนรอบจะทำให้จำนวน clock cycles เปลี่ยนไปและอาจส่งผลให้การพรีเฟ็ตช์ไม่อยู่ในสถานะตรงเวลาเมื่อรอบที่วนซ้ำมีจำนวนมาก บทความนี้จึงนำเสนอการพรีเฟ็ตช์โดยอาศัย Trident frame work ซึ่งเป็น event-driven และเป็น multithreaded dynamic optimization framework เข้ามาใช้ในการสร้าง helper thread เพื่อตรวจสอบตาราง Delinquent load หากการพรีเฟ็ตช์ไม่มีประสิทธิภาพเพียงพอจะถูกทำเครื่องหมายเพื่อให้พรีเฟ็ตช์ในจุดอื่นแทน และค่าจะถูกทำนายและแก้ไขในตารางภายหลังเพื่อใช้ในการเพิ่มหรือลดระยะทางในการพรีเฟ็ตช์ในครั้งถัดไป แต่การสร้าง helper thread จำเป็นต้องใช้ทรัพยากรเป็นอย่างมากในการตัดสินใจที่จะพรีเฟ็ตช์และคำนวณระยะทางในการพรีเฟ็ตช์ขณะประมวลผล ผู้วิจัยจึงมีแนวคิดหาระยะทางที่เหมาะสมในการพรีเฟ็ตช์ของเครื่องที่ใช้ในการประมวลผลในขณะ complier โปรแกรม เพื่อลดการใช้ทรัพยากรดังกล่าว อีกทั้งพิจารณาปริมาณการใช้ clock cycles เมื่อแทรกคำสั่งพรีเฟ็ตช์ไปแล้วเพื่อให้การพรีเฟ็ตช์ในการวนรอบอยู่ในสถานะตรงเวลา

บทที่ 3
วิธีการดำเนินการวิจัย

การพรีเฟ็ตช์สามารถนำไปประยุกต์ใช้ได้กับการประมวลผลทั่วไป และสามารถทำงานได้มีประสิทธิภาพยิ่งขึ้นหากงานที่คำนวณมีลักษณะของการทำงานสองประการ คือ ประการที่หนึ่ง ข้อมูลมีขนาดใหญ่มากจนไม่สามารถใส่ในหน่วยความจำแคชของซีพียูได้ทั้งหมดในเวลาเดียวกัน และประการที่สอง คือ มีการใช้ข้อมูลขนาดใหญ่ทั้งหมดนั้นซ้ำหลายครั้ง โดยมีลักษณะการเข้าถึงข้อมูลเป็นแบบแผนที่แน่นอนและสามารถทำการวางแผนการอ่านข้อมูลล่วงหน้าได้ เพื่อให้สามารถทำการพรีเฟ็ตช์ได้อย่างมีประสิทธิภาพ ซึ่งจะเห็นว่าการคูณเมทริกซ์ด้วยเมทริกซ์เป็นการดำเนินการทางคณิตศาสตร์ที่มีคุณลักษณะดังกล่าวอย่างครบถ้วน อีกทั้งเป็นการดำเนินการทางคณิตศาสตร์พื้นฐานที่จำเป็นต้องใช้ในการประมวลผลทางวิทยาศาสตร์และวิศวกรรมอยู่แล้ว [11] ดังนั้นผู้วิจัยจึงได้เลือกใช้กรณีดังกล่าวมาเป็นกรณีศึกษาสำหรับนำเสนอหลักการในการพรีเฟ็ตช์

3.1. การปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยการปรับซอฟต์แวร์

การปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยการปรับซอฟต์แวร์เป็นการใช้เทคนิคในการปรับปรุงประสิทธิภาพของอัลกอริทึมในการคูณเมทริกซ์ก่อนที่จะนำอัลกอริทึมที่ได้รับการปรับปรุงประสิทธิภาพแล้วไปทำการทดลองเพื่อหารูปแบบการพรีเฟ็ตช์ต่อไป ขั้นตอนการปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยการปรับซอฟต์แวร์ประกอบด้วย

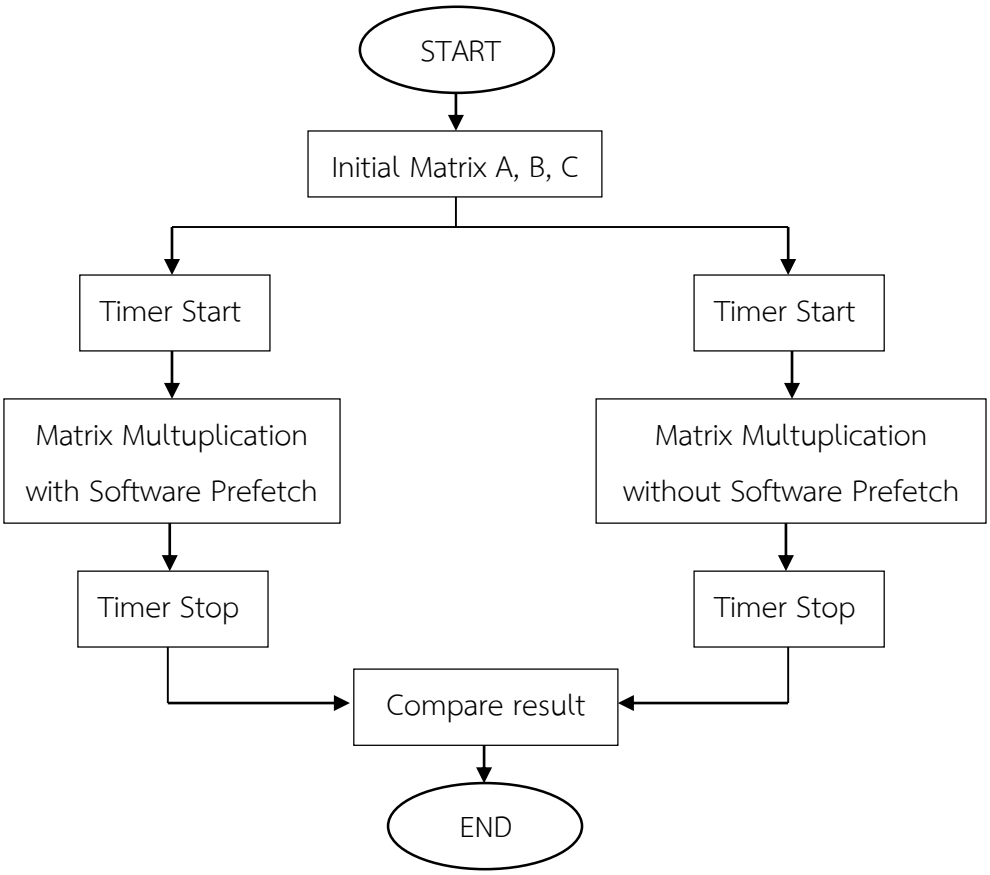
1) การปรับลำดับชั้นการวนรอบ
2) การใช้วิธี blocking
3) การใช้วิธีคลายการวนซ้ำ
4) การใช้ชุดคำสั่ง AVX

การปรับลำดับชั้นการวนรอบเป็นการพิจารณาการใช้ประโยชน์จากลักษณะการใช้ข้อมูลเพื่อประมวลผล ประกอบด้วยการใช้ข้อมูลซ้ำ (temporal locality) และการใช้ข้อมูลในตำแหน่งที่อยู่ติดกันมีโอกาสให้อ่านข้อมูลจากหน่วยความจำน้อยลง (spatial locality) การใช้วิธี blocking เพื่อแบ่งการประมวลผลออกเป็นเมทริกซ์ย่อย การใช้วิธีคลายการวนซ้ำ เพื่อลดจำนวนครั้งในการวนรอบและหลีกเลี่ยงการพรีเฟ็ตช์ข้อมูลในตำแหน่งเดิมซ้ำ การใช้ชุดคำสั่ง AVX

เพื่อประมวลผลชุดข้อมูลหลายตำแหน่งในการใช้คำสั่งเดียว เมื่ออัลกอริทึมได้ถูกปรับปรุงให้สามารถเพิ่มประสิทธิภาพในการพรีเฟ็ตช์และส่งเสริมการใช้คำสั่งพรีเฟ็ตช์แล้ว จึงนำโปรแกรมที่ได้รับการปรับปรุงไปทำการทดลองเพิ่มคำสั่งพรีเฟ็ตช์ต่อไป

3.2. การทดลองปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยซอฟต์แวร์พรีเฟ็ตช์

การทดลองปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยซอฟต์แวร์พรีเฟ็ตช์ มีการใช้คำสั่งในการอ่านข้อมูลล่วงหน้าของสมาชิกเมทริกซ์ที่จะนำไปประมวลผล การอ่านข้อมูลล่วงหน้าของเมทริกซ์สามารถอ่านข้อมูลได้ทั้งแถวและหลักของเมทริกซ์ ดังนั้นจะสามารถมีการพรีเฟ็ตช์ได้ทั้งหมด 6 รูปแบบ ได้แก่ พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง พรีเฟ็ตช์หลักของเมทริกซ์ตัวตั้ง พรีเฟ็ตช์แถวของเมทริกซ์ตัวคูณ พรีเฟ็ตช์หลักของเมทริกซ์ตัวคูณ พรีเฟ็ตช์แถวของผลลัพธ์ และพรีเฟ็ตช์หลักของเมทริกซ์ผลลัพธ์ ผลลัพธ์ของโปรแกรมการคูณเมทริกซ์ที่ใช้คำสั่งพรีเฟ็ตช์จะถูกนำไปเปรียบเทียบกับโปรแกรมการคูณเมทริกซ์ในรูปแบบเดิมก่อนการพรีเฟ็ตช์เพื่อตรวจสอบผลลัพธ์ให้มีความถูกต้องและมีการวัดความเร็วในการประมวลผลเฉพาะส่วนของการคูณเมทริกซ์เพื่อเก็บผลลัพธ์นำไปเปรียบเทียบในการเขียนโปรแกรมที่ต่างกันแสดงดังรูปที่ 3-1

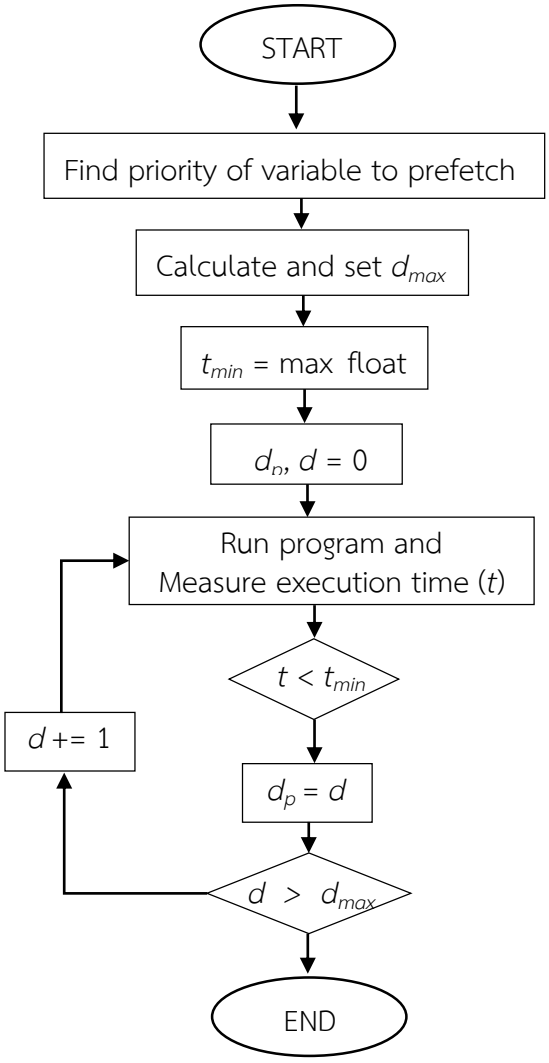


รูปที่ 3-1 ผังการทำงานเพื่อตรวจสอบผลลัพธ์ของโปรแกรมการคูณที่ใช้คำสั่งพรีเฟ็ตช์

หลังจากแจกแจงรูปแบบในการพรีเฟ็ตช์ทั้งหมดที่สามารถเพิ่มคำสั่งพรีเฟ็ตช์ได้แล้ว ผู้วิจัยได้นำรูปแบบที่ได้ทั้งหมดมาแจกแจงเป็นวิธีที่ใช้ในการพรีเฟ็ตช์ทั้งหมด จากนั้นนำวิธีทั้งหมดที่ได้มาวิเคราะห์ปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์ได้และจำนวนการเรียกใช้คำสั่งพรีเฟ็ตช์เพื่อพิจารณาจากปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์มากที่สุดก่อน และพิจารณาจำนวนการเรียกใช้คำสั่งพรีเฟ็ตช์น้อยที่สุดเป็นลำดับถัดไป หากการวิเคราะห์ดังกล่าวไม่สามารถหาวิธีการพรีเฟ็ตช์ที่ดีที่สุดเพียงวิธีเดียวได้ จะต้องทำการทดลองต่อโดยวิเคราะห์รูปแบบพฤติกรรมการใช้ข้อมูลของแต่ละวิธีเพื่อหาวิธีการพรีเฟ็ตช์ที่ดีที่สุดสำหรับอัลกอริทึมนี้เพียงวิธีเดียว การทดลองนี้มีจุดประสงค์ในการเขียนซอฟต์แวร์พรีเฟ็ตช์เพื่อหารูปแบบในการอ่านข้อมูลล่วงหน้าที่เหมาะสมที่ทำให้โปรแกรมการคูณเมทริกซ์มีประสิทธิภาพสูงขึ้น

3.3. การหาระยะทางของการพรีเฟ็ตช์ที่เหมาะสม

การหาระยะทางในการพรีเฟ็ตช์ที่เหมาะสมเริ่มจากการนำเอาโปรแกรมที่มีการพรีเฟ็ตช์รูปแบบที่เหมาะสมที่สุดจากการทดลองปรับปรุงประสิทธิภาพโปรแกรมการคูณ



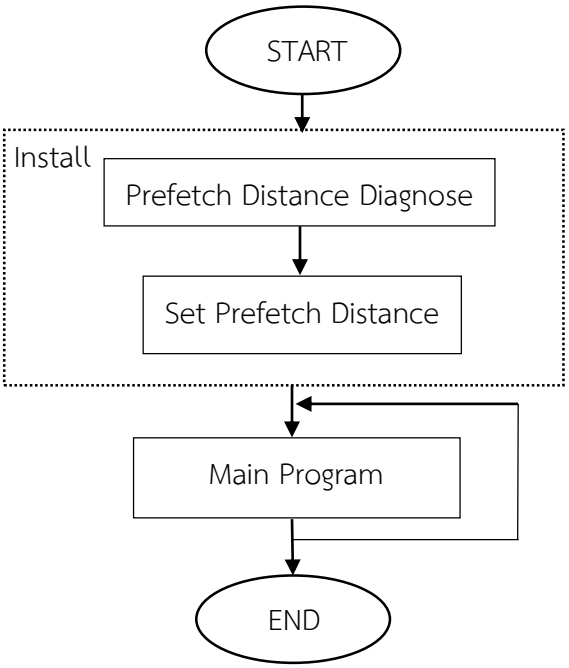
รูปที่ 3-2 แผนผังการทดลองหาระยะทางพรีเฟ็ตช์

มาวิเคราะห์หาลำดับความสำคัญของตัวแปรเมื่อได้รับการพรีเฟ็ตช์ จากนั้นทดลองหาระยะทางที่เหมาะสมของตัวแปรที่ต้องการพรีเฟ็ตช์แต่ละตัวตามลำดับความสำคัญ มีแผนผังการทดลองแสดงดังรูปที่ 3-2

การหาค่าระยะทางพรีเฟ็ตช์ที่เหมาะสมสูงสุดเริ่มจากการหาลำดับความสำคัญของตัวแปรที่ต้องพรีเฟ็ตช์ก่อน และคำนวณหาระยะทางที่มากที่สุดจากการพิจารณาขนาดของหน่วยความจำแคช จากนั้นทำการประมวลผลระยะทางมีค่าเป็น 0 ถึงระยะทางที่มากที่สุด การประมวลในแต่ละระยะทางจะทำการประมวลผลโปรแกรมที่มีระยะทางต่างกันในเมทริกซ์ 7 ขนาด ได้แก่ ขนาด 1024×1024, 2048×2048, 3072×3072, 4096×4096, 5120×5120, 6144×6144 และ 7168×7168 เพื่อหาค่าเฉลี่ยของเวลาในการประมวลผลทั้งหมดในเมทริกซ์แต่ละขนาด การเลือกขนาดเมทริกซ์ที่ใช้ในการทดลองพิจารณาจากขนาดเมทริกซ์ที่หารด้วยขนาด line size แคชของสถาปัตยกรรมที่ใช้ทำการทดลองลงตัว ทำให้สามารถวิเคราะห์หาระยะทางที่เหมาะสมได้ง่าย

3.4. การทดสอบประสิทธิภาพของโปรแกรมการคูณเมทริกซ์ที่นำเสนอบนเครื่องที่มีทรัพยากรต่างกัน

การทดสอบประสิทธิภาพของโปรแกรมการคูณเมทริกซ์ที่นำเสนอจะทดสอบบนเครื่องคอมพิวเตอร์ที่มีทรัพยากรต่างกัน ได้แก่ Intel Core i5 6200U และ Intel Core i7 9700K เพื่อวิเคราะห์และวัดประสิทธิภาพของอัลกอริทึมที่นำเสนอ โดยมีผังการทดลองดังนี้



รูปที่ 3-3 แผนผังภาพรวมโปรแกรมที่นำเสนอ

การทดลองในแต่ละเครื่องคอมพิวเตอร์ที่มีทรัพยากรต่างกัน เริ่มด้วยการหาระยะทางในการพรีเฟ็ตช์ที่เหมาะสมมีขั้นตอนการหาดังรูปที่ 3-2 จากนั้นกำหนดค่าระยะทางการพรีเฟ็ตช์ให้แก่โปรแกรมก่อนที่โปรแกรมจะประมวลผลคูณเมทริกซ์ ซึ่งในขั้นตอนนี้จะถูกเรียกว่าขั้นตอนการติดตั้งค่าระยะทางที่เหมาะสมจะถูกเขียนลงในไฟล์ตั้งค่า การติดตั้งจะทำเพียงครั้งเดียวต่อการประมวลผลคอมพิวเตอร์หนึ่งเครื่อง จากนั้นโปรแกรมการคูณเมทริกซ์จะอ่านค่าจากไฟล์ตั้งค่าดังกล่าวเพื่อนำไปประมวลผลและใช้คำสั่งพรีเฟ็ตช์เพื่อวัดเวลาในการประมวลผลถัดไป

บทที่ 4
ผลการดำเนินงาน

ผลการดำเนินงานแบ่งเป็น 4 การทดลองหลัก ได้แก่ การปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยการปรับซอฟต์แวร์ การทดลองปรับปรุงประสิทธิภาพด้วยซอฟต์แวร์พรีเฟ็ตช์ การหาระยะทางการพรีเฟ็ตช์ที่เหมาะสม และการทดสอบประสิทธิภาพของโปรแกรมการคูณเมทริกซ์ที่นำเสนอบนเครื่องที่มีทรัพยากรต่างกัน อัลกอริทึมที่ที่ได้จากการปรับปรุงประสิทธิภาพด้วยซอฟต์แวร์พรีเฟ็ตช์ที่เป็นการทดลองแรกในจะถูกนำไปใช้ในการทดลองถัดไป จากนั้นอัลกอริทึมหรือผลการทดลองก่อนหน้าจะถูกนำไปใช้ต่อในการทดลองต่อไป ผลการดำเนินงานในแต่ละการทดลองมีรายละเอียดดังต่อไปนี้

4.1. การปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยการปรับซอฟต์แวร์

ขั้นตอนการทดลองเริ่มจากเขียนโปรแกรมในการคูณเมทริกซ์ขึ้นมา โปรแกรมการคูณเมทริกซ์จะประกอบไปด้วยการวนรอบซ้อนกัน 3 ชั้น ซึ่งจะเรียกโปรแกรมการคูณรูปแบบนี้ว่า “การคูณแบบ 3-loop” เมื่อให้ N คือจำนวนการวนรอบในแต่ละชั้น สามารถวัดประสิทธิภาพของอัลกอริทึมตาม Big O Notation จะได้เป็น $O(N^3)$ ดังรหัสเทียม (pseudo code) ซึ่งแสดงในรูปที่ 4-1

```
For i = 0 to N-1  // เพื่อเข้าถึงแถวของเมทริกซ์ตัวตั้งและผลลัพธ์
    For j = 0 to N-1  // เพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณและผลลัพธ์
        For k = 0 to N-1  // เพื่อเข้าถึงแถวของเมทริกซ์ตัวคูณและหลักของเมทริกซ์ตัวตั้ง
            C[i][j] = C[i][j] +A[i][k] * B[k][j]
```

รูปที่ 4-1 รหัสเทียมของโปรแกรมคูณเมทริกซ์แบบ 3-loop (อัลกอริทึมที่ 1)

โปรแกรมการคูณเมทริกซ์นี้ประกอบด้วย ชั้นนอกสุดทำการวนรอบเพื่อเข้าถึงแถวของเมทริกซ์ตัวตั้งและเมทริกซ์ผลลัพธ์ ชั้นที่สองทำการวนรอบเพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณและเมทริกซ์ผลลัพธ์ ส่วนชั้นในสุดทำการวนรอบเพื่อเข้าถึงแถวของเมทริกซ์ตัวคูณและหลักของเมทริกซ์

ตัวตั้ง จากนั้นผู้วิจัยได้เปลี่ยนรูปแบบการเข้าถึงเมทริกซ์โดยสลับการวนรอบชั้นที่สองและชั้นนอกสุด ดังแสดงในรูปที่ 4-2

```
For j = 0 to N-1  // เพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณและผลลัพธ์
    For i = 0 to N-1  // เพื่อเข้าถึงแถวของเมทริกซ์ตัวตั้งและผลลัพธ์
        For k = 0 to N-1  // เพื่อเข้าถึงแถวของเมทริกซ์ตัวคูณและหลักของเมทริกซ์ตัวตั้ง
            C[i][j] = C[i][j] +A[i][k] * B[k][j]
```

รูปที่ 4-2 โปรแกรมคูณเมทริกซ์แบบ 3-loop ที่ได้รับการปรับปรุงการทำงานแล้ว (อัลกอริทึมที่ 2)

จากรูปที่ 4-2 เป็นการปรับปรุงโปรแกรมการคูณเมทริกซ์แบบ 3-loop โดยเพิ่มจำนวนการใช้ข้อมูลเดิมซ้ำของสมาชิกเมทริกซ์ตัวคูณ ส่งผลให้ลดระยะเวลาในการอ่านข้อมูลลงได้ การเปรียบเทียบผลลัพธ์ด้านการเข้าถึงหน่วยความจำเมื่อทำการวนรอบเพื่อประมวลผล ดังแสดงในตารางที่ 4-3

ตารางที่ 4-3 การเปรียบเทียบผลลัพธ์ด้านการเข้าถึงหน่วยความจำเมื่อแต่ละตัวแปรที่ใช้วนรอบ

	อัลกอริทึมที่ 1 (อัลกอริทึมเดิม)			อัลกอริทึมที่ 2 (อัลกอริทึมที่ปรับปรุง)		
เมทริกซ์ / การเพิ่มค่า	ตัวตั้ง(*A*)	ตัวคูณ(*B*)	ผลลัพธ์(*C*)	ตัวตั้ง(*A*)	ตัวคูณ(*B*)	ผลลัพธ์(*C*)
รอบในสุด	*S*		*T*	*S*		*T*
รอบกลาง	*T*				*T*	
รอบนอกสุด						

จากตารางที่ 4-3 กำหนดให้ *S* หมายถึงการมีโอกาสในการใช้ข้อมูลตำแหน่งที่ถัดไป (spatial locality) ทำให้ข้อมูลที่อยู่ในตำแหน่งติดกันมีโอกาสถูกอ่านเข้ามายังแคชล่วงหน้า และ *T* หมายถึงการใช้ข้อมูลชุดเดิมซ้ำ (temporal locality) ทำให้จำนวนการอ่านข้อมูลจากหน่วยความจำ พบว่าการเกิดโอกาสในการใช้ข้อมูลตำแหน่งที่ถัดไปของกรณีที่เพิ่มค่ารอบในสุดจะมีมากก็ต่อเมื่อเมทริกซ์ตัวตั้งมีการเปลี่ยนแปลง ซึ่งอัลกอริทึมที่ 2 ใช้การวนรอบกลางเป็นตัวแปรที่เข้าถึงแถวของเมทริกซ์ตัวตั้งทำให้เมทริกซ์ตัวตั้งของอัลกอริทึมที่ 2 เปลี่ยนแปลงมากกว่าเมทริกซ์ตัวตั้งของอัลกอริทึมที่ 1 ที่ตัวแปรเข้าถึงแถวของเมทริกซ์เป็นการวนรอบนอกสุด ส่งผลให้อัลกอริทึมที่ 2

มีโอกาสใช้ข้อมูลตำแหน่งถัดไปมากกว่า จำนวนการวนรอบของตัวแปร *i* ในอัลกอริทึมที่ 1 คิดเป็น *n* รอบ ส่วนอัลกอริทึมที่ 2 คิดเป็น *n*×*n* รอบนั่นเอง จากนั้นผู้วิจัยได้ใช้การทำ blocking เพื่อแบ่งเมทริกซ์ตัวตั้งออกเป็นเมทริกซ์ย่อยเพื่อเพิ่มการใช้ข้อมูลชุดเดิมซ้ำ ตัวอย่างโปรแกรมการทำ blocking ดังแสดงในรูปที่ 4-3

```
for (int j = 0; j < N; j += 1)  // เพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณและผลลัพธ์
    for (int i = 0; i < N ; i += Ti) // แบ่งแถวของเมทริกซ์ตัวตั้งและผลลัพธ์
        for (int k = 0; k < N ; k += Tk) // แบ่งแถวของเมทริกซ์ตัวคูณและหลักของตัวตั้ง
            for (int ii = i; ii < i + Ti; ii++) // เข้าถึงแถวย่อยของเมทริกซ์ตัวตั้งและผลลัพธ์
                for (int kk = k; kk < k + TK; kk++) // เข้าถึงแถวและหลักย่อยของเมทริกซ์
                    C[i][j] = C[i][j] +A[i][k] * B[k][j];
```

รูปที่ 4-3 การคูณเมทริกซ์โดยใช้วิธีการทำ blocking ให้กับเมทริกซ์ตัวตั้ง (อัลกอริทึมที่ 3)

จากรูปที่ 4-3 เป็นการทำ blocking ให้กับเมทริกซ์ตัวตั้ง ซึ่งจะแบ่งตัวแปรที่เข้าถึงแถวและหลักของเมทริกซ์ตัวตั้งออกเป็นแถวและหลักย่อย T_i คือขนาดการแบ่งแถวของเมทริกซ์ตัวตั้งและผลลัพธ์มีค่าเป็น 128 T_k คือขนาดการแบ่งแถวของเมทริกซ์ตัวคูณและหลักของตัวตั้งมีค่าเป็น 64 จากนั้นนำโปรแกรมที่ได้จากรูปที่ 4-3 มาทำใช้เทคนิคคลายการวนซ้ำเพื่อลดจำนวนครั้งในการวนรอบลง ตัวอย่างโปรแกรมคลายการวนซ้ำดังรูปที่ 4-4

```
for (int j = 0; j < N; j += 8)  // เพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณและผลลัพธ์
    for (int i = 0; i < N ; i += Ti) // แบ่งแถวของเมทริกซ์ตัวตั้งและผลลัพธ์
        for (int k = 0; k < N ; k += Tk) // แบ่งแถวของเมทริกซ์ตัวคูณและหลักของตัวตั้ง
            for (int ii = i; ii < i + Ti; ii++) // เข้าถึงแถวย่อยของเมทริกซ์ตัวตั้งและผลลัพธ์
                for (int kk = k; kk < k + Tk; kk++) { // เข้าถึงแถวและหลักย่อยของเมทริกซ์
                    C[i][j] = C[i][j] +A[i][k] * B[k][j];
                    C[i][j+1] = C[i][j+1] +A[i][k] * B[k][j+1];
                    C[i][j+2] = C[i][j+2] +A[i][k] * B[k][j+2];
                    C[i][j+3] = C[i][j+3] +A[i][k] * B[k][j+3];
                    C[i][j+4] = C[i][j+4] +A[i][k] * B[k][j+4];
                    C[i][j+5] = C[i][j+5] +A[i][k] * B[k][j+5];
                    C[i][j+6] = C[i][j+6] +A[i][k] * B[k][j+6];
                    C[i][j+7] = C[i][j+7] +A[i][k] * B[k][j+7];}
```

รูปที่ 4-4 การคลายการวนซ้ำของตัวแปร j (อัลกอริทึมที่ 4)

จากรูปที่ 4-4 เป็นการคลายการวนซ้ำของตัวแปร *j* ลดจำนวนการวนรอบเพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณและเมทริกซ์ผลลัพธ์ลง และเพิ่มคำสั่งการคูณเป็น 8 ครั้งต่อหนึ่งการวนรอบ อีกทั้งการคลายการวนซ้ำสามารถหลีกเลี่ยงการพรีเฟ็ตช์ข้อมูลเดิมซ้ำได้อีกด้วย จากนั้นผู้วิจัยได้นำโปรแกรมที่พัฒนาขึ้นมาทำการใช้ชุดคำสั่ง AVX แทนคำสั่งการคูณทั่วไป ดังแสดงในรูปที่ 4-5 และรูปที่ 4-6

```
for (int j = 0; j < N; j += 64)  // เพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณและผลลัพธ์
    for (int i = 0; i < N ; i += T_i) // แบ่งแถวของเมทริกซ์ตัวตั้งและผลลัพธ์
        for (int k = 0; k < N ; k += T_k) // แบ่งแถวของเมทริกซ์ตัวคูณและหลักของตัวตั้ง
            for (int ii = i; ii < i + T_i; ii++) // เข้าถึงแถวย่อยของเมทริกซ์ตัวตั้งและผลลัพธ์
                r_1= _mm256_setzero_ps();  // ล้างข้อมูลตัวแปรที่ใช้เข้าถึงผลลัพธ์
                r_2 = _mm256_setzero_ps();
                :
                r_8 = _mm256_setzero_ps();
                for (int kk = k; kk < k + T_k; kk++) { // เข้าถึงแถวและหลักย่อยของเมทริกซ์
                    a_1 = _mm256_set1_ps(A[ii][kk]); // อ่านข้อมูลของสมาชิกเมทริกซ์ตัวตั้ง
                    b_1 = _mm256_load_ps(&B[kk][j]); // อ่านข้อมูลของสมาชิกตัวคูณ
                    b_2 = _mm256_load_ps(&B[kk][j + 8]);
                    :
                    b_8 = _mm256_load_ps(&B[kk][j + (8 * 7)]);
                    /* คูณและบวกผลลัพธ์เก็บในตัวแปร r_1 – r_8*/
                    r_1 = _mm256_fmadd_ps(a_1, b_1, r_1);
                    r_2 = _mm256_fmadd_ps(a_1, b_2, r_2);
                     :
                    r_8 = _mm256_fmadd_ps(a_1, b_8, r_8);
                }
                /* อ่านค่าสมาชิกของเมทริกซ์ผลลัพธ์ */
                c_1 = _mm256_load_ps(&C[ii][j]);
                c_2 = _mm256_load_ps(&C[ii][j + 8]);
                  :
                c_8 = _mm256_load_ps(&C[ii][j + (8 * 7)]);
```

รูปที่ 4-5 การใช้ชุดคำสั่ง AVX มาแทนที่คำสั่งการคูณเมทริกซ์ทั่วไป (อัลกอริทึมที่ 5)

```
        /* นำตัวแปรที่เก็บผลลัพธ์จากการคูณเมทริกซ์บวกค่าเดิมของสมาชิกของเมทริกซ์ผลลัพธ์ */
                r1 = _mm256_add_ps(c1, r1);
                r2 = _mm256_add_ps(c2, r2);
                 :
                r8 = _mm256_add_ps(c8, r8);
                /* นำผลลัพธ์ที่ได้บวกสะสมเก็บลงในสมาชิกของเมทริกซ์ผลลัพธ์ */
                _mm256_store_ps(&C[ii][j], r1);
                _mm256_store_ps(&C[ii][j + 8], r2);
                _mm256_store_ps(&C[ii][j + (8 * 2)], r3);
                :
                _mm256_store_ps(&C[ii][j + (8 * 7)], r8);
            }
        }
    }
}
```

รูปที่ 4-6 การใช้ชุดคำสั่ง AVX มาแทนที่คำสั่งการคูณเมทริกซ์ทั่วไป (อัลกอริทึมที่ 5) (ต่อ)

จากอัลกอริทึมที่ 5 เป็นการใช้ชุดคำสั่ง AVX มาแทนที่คำสั่งในการคูณเมทริกซ์ทั่วไป โปรแกรมการคูณเมทริกซ์ที่ถูกเขียนขึ้นใหม่นี้จะสามารถประมวลผลพร้อมกันได้ถึง 256 บิต กล่าวได้ว่าหากใช้ตัวแปรชนิด float จะสามารถประมวลผลพร้อมกันได้ถึง 8 จำนวน อีกทั้งโปรแกรมเดิมที่มีการคลายการวนซ้ำ 8 จำนวน ส่งผลให้สามารถคูณได้ 64 ครั้งต่อหนึ่งการวนรอบนั่นเอง

4.2. การทดลองปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยซอฟต์แวร์พรีเฟ็ตช์

ขั้นตอนการทดลองเริ่มต่อจากการทดลองปรับปรุงประสิทธิภาพโปรแกรมการคูณเมทริกซ์ด้วยการปรับซอฟต์แวร์ ซึ่งได้นำเอาโปรแกรมของผลการทดลองที่ดีที่สุดมาพัฒนาต่อ โดยเพิ่มคำสั่งที่ใช้อ่านข้อมูลล่วงหน้าของสมาชิกเมทริกซ์ที่จะนำไปประมวลผล การใช้คำสั่งพรีเฟ็ตช์ในการคูณเมทริกซ์สามารถทำได้ทั้งหมด 6 รูปแบบ ได้แก่

1) พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง ($A[i+d_A][k]$)
2) พรีเฟ็ตช์หลักของเมทริกซ์ตัวตั้ง ($A[i][k+d_A]$)

3) พรีเฟ็ตช์แถวของเมทริกซ์ตัวคูณ ($B[k+d_B][j]$)

4) พรีเฟ็ตช์หลักของเมทริกซ์ตัวคูณ ($B[k][j+d_B]$)

5) พรีเฟ็ตช์แถวของเมทริกซ์ผลลัพธ์ ($C[i+d_C][j]$)

6) พรีเฟ็ตช์หลักของเมทริกซ์ผลลัพธ์ ($C[i][j+d_C]$)

ผู้วิจัยจึงได้ทดลองเขียนโปรแกรมที่ใช้คำสั่งพรีเฟ็ตช์ต่างกัน 6 รูปแบบ พบว่าการพรีเฟ็ตช์หลักของเมทริกซ์ตัวคูณและหลักของเมทริกซ์ผลลัพธ์ในรูปแบบที่ 4) และ 6) เมื่อวัดเวลาในการประมวลผลจะให้ผลลัพธ์ที่ช้ากว่าการไม่ใช้คำสั่งพรีเฟ็ตช์ เนื่องจากตัวแปรในการเข้าถึงหลักของเมทริกซ์ตัวคูณและเมทริกซ์ผลลัพธ์เป็นการวนรอบที่อยู่ภายนอกสุดดังรูปที่ 4-6 ทำให้ข้อมูลของสมาชิกที่ได้มาจากการพรีเฟ็ตช์มีโอกาสสูงที่จะถูกขับไล่ออกจากแคชทุกระดับก่อนที่จะใช้งาน เช่น ความจุของแคชที่ระดับ L3 มีขนาด 3 เมกะไบต์ แต่การวนรอบนอกสุดในแต่ละรอบของการประมวลผลเมทริกซ์ขนาด 1024x1024 มีการอ่านค่าเข้าไปเก็บยังหน่วยความจำแคชประมาณ 6 เมกะไบต์ ส่งผลข้อมูลที่ได้จากการพรีเฟ็ตช์ในการวนรอบก่อนหน้าถูกนำออกจากแคชก่อนที่จะถูกประมวลผล ดังนั้นการพรีเฟ็ตช์หลักของเมทริกซ์ตัวคูณและหลักของเมทริกซ์ผลลัพธ์เป็นการใช้คำสั่งพรีเฟ็ตช์โดยเปล่าประโยชน์ ผู้วิจัยจึงได้ตัดการพรีเฟ็ตช์ 2 รูปแบบดังกล่าวออกและได้ทดลองนำการพรีเฟ็ตช์อีก 4 รูปแบบมาเขียนโปรแกรมโดยกำหนดให้อ่านข้อมูลล่วงหน้าไปหนึ่งตำแหน่ง ซึ่งสามารถเขียนได้เป็นซอฟต์แวร์พรีเฟ็ตช์จำนวน 15 วิธีดังนี้

1) โปรแกรมที่พรีเฟ็ตช์เฉพาะข้อมูลในแถวของเมทริกซ์ตัวตั้ง

2) โปรแกรมที่พรีเฟ็ตช์เฉพาะข้อมูลในหลักของเมทริกซ์ตัวตั้ง

3) โปรแกรมที่พรีเฟ็ตช์เฉพาะข้อมูลในแถวของเมทริกซ์ตัวคูณ

4) โปรแกรมที่พรีเฟ็ตช์เฉพาะข้อมูลในแถวของเมทริกซ์ผลลัพธ์

5) โปรแกรมที่พรีเฟ็ตช์แถวและหลักของเมทริกซ์ตัวตั้ง

6) โปรแกรมที่พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้งและแถวของเมทริกซ์ตัวคูณ

7) โปรแกรมที่พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้งและแถวของเมทริกซ์ผลลัพธ์

8) โปรแกรมที่พรีเฟ็ตช์หลักของเมทริกซ์ตัวตั้งและแถวของเมทริกซ์ตัวคูณ

9) โปรแกรมที่พรีเฟ็ตช์หลักของเมทริกซ์ตัวตั้งและแถวของเมทริกซ์ผลลัพธ์

10) โปรแกรมที่พรีเฟ็ตช์แถวของเมทริกซ์ตัวคูณและแถวของเมทริกซ์ผลลัพธ์

11) โปรแกรมที่พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง หลักของเมทริกซ์ตัวตั้งและแถวของเมทริกซ์ตัวคูณ

12) โปรแกรมที่พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง หลักของเมทริกซ์ตัวตั้งและแถวของเมทริกซ์ผลลัพธ์

13) โปรแกรมที่พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง แถวของเมทริกซ์ตัวคูณและแถวของเมทริกซ์ผลลัพธ์

14) โปรแกรมที่พรีเฟ็ตช์หลักของเมทริกซ์ตัวตั้ง แถวของเมทริกซ์ตัวคูณและแถวของเมทริกซ์ผลลัพธ์

15) โปรแกรมที่พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง หลักของเมทริกซ์ตัวตั้ง แถวของเมทริกซ์ตัวคูณและแถวของเมทริกซ์ผลลัพธ์

ผู้วิจัยได้นำ 15 วิธีในการพรีเฟ็ตช์มาวิเคราะห์ปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์ได้และจำนวนการเรียกใช้คำสั่งพรีเฟ็ตช์ เพื่อหาวิธีที่ได้ปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์มากที่สุดแต่ใช้จำนวนการเรียกใช้คำสั่งในการพรีเฟ็ตช์น้อยที่สุด เนื่องจากจำนวนการเรียกใช้คำสั่งพรีเฟ็ตช์ส่งผลต่อเวลาในการประมวลผล หากจำนวนการเรียกใช้คำสั่งพรีเฟ็ตช์มีปริมาณมากส่งผลให้ใช้เวลาในการประมวลผลมากขึ้นไปด้วย

ตารางที่ 4-4 วิเคราะห์ปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์ได้และจำนวนการเรียกใช้คำสั่งพรีเฟ็ตช์

วิธีการพรีเฟ็ตช์	จำนวนการเรียกใช้ คำสั่งพรีเฟ็ตช์	ปริมาณข้อมูลที่ได้จาก การพรีเฟ็ตช์ (Byte)
1)	$\frac{N^3}{E * T_k}$	$D * \frac{N^3}{E * T_k}$
2)	$\frac{N^3}{E * T_k}$	$D * \frac{N^3}{E * T_k}$
3)	$\frac{N^3}{E}$	$D * \frac{N^3}{E}$
4)	$\frac{N^3}{E * T_k}$	$D * \frac{N^3}{E * T_k}$
5)	$\frac{2N^3}{E * T_k}$	$D * \frac{N^3}{E * T_k}$
6)	$\frac{N^3}{E * T_k} + \frac{N^3}{E}$	$D * \left(\frac{N^3}{E * T_k} + \frac{N^3}{E}\right)$
7)	$\frac{2N^3}{E * T_k}$	$D * \frac{2N^3}{E * T_k}$
8)	$\frac{N^3}{E * T_k} + \frac{N^3}{E}$	$D * \left(\frac{N^3}{E * T_k} + \frac{N^3}{E}\right)$
9)	$\frac{2N^3}{E * T_k}$	$D * \frac{2N^3}{E * T_k}$

วิธีการพรีเฟ็ตช์	จำนวนการเรียกใช้ คำสั่งพรีเฟ็ตช์	ปริมาณข้อมูลที่ได้จาก การพรีเฟ็ตช์ (Byte)
10)	$\frac{N^3}{E*T_k}+\frac{N^3}{E}$	$D*\left(\frac{N^3}{E*T_k}+\frac{N^3}{E}\right)$
11)	$\frac{2N^3}{E*T_k}+\frac{N^3}{E}$	$D*\left(\frac{N^3}{E*T_k}+\frac{N^3}{E}\right)$
12)	$\frac{3N^3}{E*T_k}$	$D*\frac{2N^3}{E*T_k}$
13)	$\frac{2N^3}{E*T_k}+\frac{N^3}{E}$	$D*\left(\frac{2N^3}{E*T_k}+\frac{N^3}{E}\right)$
14)	$\frac{2N^3}{E*T_k}+\frac{N^3}{E}$	$D*\left(\frac{2N^3}{E*T_k}+\frac{N^3}{E}\right)$
15)	$\frac{3N^3}{E*T_k}+\frac{N^3}{E}$	$D*\left(\frac{2N^3}{E*T_k}+\frac{N^3}{E}\right)$

กำหนดให้ *D* หมายถึงปริมาณข้อมูลที่ทำการพรีเฟ็ตช์ต่อหนึ่งคำสั่ง *N* หมายถึงขนาดของเมทริกซ์ที่ใช้ในการประมวลผล *E* หมายถึงจำนวน element ของเมทริกซ์ต่อหนึ่งคำสั่งของชุดคำสั่ง AVX และ T_k หมายถึงจำนวนแถวของเมทริกซ์ย่อย

จากตารางที่ 4-4 พิจารณาจากปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์พบว่าวิธีที่ 13 14 และ 15 มีปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์มากที่สุด แต่เมื่อพิจารณาถึงจำนวนการเรียกใช้คำสั่งพรีเฟ็ตช์พบว่า วิธีที่ 13 และ 14 มีจำนวนครั้งในการเรียกใช้น้อยกว่าวิธีที่ 15 เนื่องจากวิธีที่ 15 มีการใช้คำสั่งพรีเฟ็ตช์เพื่ออ่านค่าข้อมูลในตำแหน่งที่ทับกัน ดังนั้นวิธีที่ 13 และ 14 เป็นวิธีที่ดีที่สุดเมื่อวิเคราะห์จากปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์ได้และจำนวนการเรียกใช้คำสั่งพรีเฟ็ตช์

หลังจากนั้นผู้วิจัยได้นำการพรีเฟ็ตช์วิธีที่ 13 และ 14 มาวิเคราะห์รูปแบบการทำงานเพื่อเปรียบเทียบปริมาณการใช้หน่วยความจำ โดยสมมติให้การพรีเฟ็ตช์มีสถานะตรงเวลา กำหนดให้ *BPL* หมายถึงปริมาณข้อมูลต่อหนึ่งการวนรอบ N_L หมายถึงจำนวนการวนรอบ *D* หมายถึงปริมาณข้อมูลต่อหนึ่งคำสั่งของการพรีเฟ็ตช์ d_B หมายถึงระยะทางการพรีเฟ็ตช์ของเมทริกซ์ตัวคูณ d_A หมายถึงระยะทางการพรีเฟ็ตช์ของเมทริกซ์ตัวตั้ง T_i หมายถึงจำนวนหลักของเมทริกซ์ย่อย T_k หมายถึงจำนวนแถวของเมทริกซ์ย่อย M_x หมายถึงปริมาณข้อมูลต่อการวนรอบ *x* ซึ่ง *x* หมายถึงลำดับชั้นการวนรอบ ได้แก่ *k*, *kk*, *i*, *ii* และ *j* tM_x หมายถึงปริมาณข้อมูลรวมของการวนรอบ *x* ทั้งหมด *E* หมายถึงจำนวน element ของเมทริกซ์ต่อหนึ่งคำสั่งของชุดคำสั่ง AVX *uF* หมายถึงจำนวนการคลายการวนซ้ำ และ *n* หมายถึงขนาดของเมทริกซ์ ผลการวิเคราะห์แสดงดังตารางที่ 4-5 และตารางที่ 4-6

ตารางที่ 4-5 รูปแบบการใช้ข้อมูลของการพรีเฟ็ตช์วิธีที่ 13

การวนรอบ	รูปแบบที่ 1		รูปแบบที่ 2		รูปแบบที่ 3	
	BPL (Byte)	N_L	*BPL* (Byte)	N_L	*BPL* (Byte)	N_L
kk	M_{kk}	d_B	M_{kk}-D	T_k-d_B	-	-
ii	tM_{kk}+M_{ii}	d_C+1	tM_{kk}+M_{ii}-(uF/ D)*E*D*T_k	d_A-(d_C+1)	tM_{kk}+M_{ii}-(uF/D) *E*D*T_k - D	T_i-d_A-(d_C+1)
k	tM_{ii}	n/T_k	-	-	-	-
i	tM_k	n/T_i	-	-	-	-
j	tM_i	$n/(E*uF)$	-	-	-	-
รวม	(((((((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii})*(d_C+1))+((((M_{kk}*d_B)+((M_{kk}-D)*(T_k-d_B))+M_{ii}-(uF/D)*E *D*T_k)*(d_A-(d_C+1)))+((((M_{kk}*d_B)+((M_{kk}-D)*(T_k-d_B))+M_{ii}-(uF/D)*E*D*T_k-D)*(T_i-d_A-(d_C+1))))*n/T_K)*n/T_i)*$n/(E*uF)$					

ตารางที่ 4-6 รูปแบบการใช้ข้อมูลของการพรีเฟ็ตช์วิธีที่ 14

การวนรอบ	รูปแบบที่ 1		รูปแบบที่ 2		รูปแบบที่ 3	
	BPL (Byte)	N_L	*BPL* (Byte)	N_L	*BPL* (Byte)	N_L
kk	M_{kk}	d_B	M_{kk}-D	T_k-d_B	M_{kk}	d_B
ii	tM_{kk}+M_{ii}	d_C+1	tM_{kk}+M_{ii}-(uF/D) *E *D* T_k	T_i-(d_C+1)	tM_{kk}+M_{ii}	d_C+1
k	tM_{ii}	d_A	tM_{ii}-D	(n/T_k)-d_A	tM_{ii}	d_A
i	tM_k	n/T_i	-	-	tM_k	n/T_i
j	tM_i	$n/(E*uF)$	-	-	tM_i	$n/(E*uF)$
รวม	(((((((((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii})*(d_C+1))+(((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii}-(uF/D)*E *D* T_k)*(T_i-(d_C+1)))*d_A)+ (((((((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii})*(d_C+1)) +((((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii}-(uF/D)*E *D* T_k)*(T_i-(d_C+1))-D)*((n/T_k)-d_A))) *n/T_i)*$n/(E*uF)$					

จากตารางที่ 4-5 และตารางที่ 4-**6** แสดงรูปแบบพฤติกรรมการใช้ข้อมูลที่ต่างกันภายในการวนรอบต่าง ๆ เช่น ในรูปแบบที่ 1 ของตารางที่ 4-5 มีปริมาณการอ่านข้อมูลเท่ากับ M_{kk} ซึ่งสามารถแทนค่าด้วย 384 เนื่องจากสถาปัตยกรรมเครื่องของ Intel Core i5 ที่ใช้ทดลองและตัวแปรชนิด float ที่ใช้ในการเขียนโปรแกรม เมื่อพิจารณาในรูปแบบที่ 1 การวนรอบ *kk* มาจากคำสั่งที่ใช้ภายในการวนรอบ *kk* เป็นการใช้คำสั่งพรีเฟ็ตช์เพื่ออ่านข้อมูลขนาด 64 ไบต์ มีการอ่านข้อมูลเมทริกซ์ตัวตั้งขนาด 64 ไบต์ และมีการอ่านข้อมูลเมทริกซ์ตัวคูณขนาด 256 ไบต์ เมื่อนำปริมาณการอ่านข้อมูลมารวมกันจะได้ 384 ไบต์ เมื่อวิเคราะห์แต่ละการวนรอบในลักษณะนี้จะได้สมการคำนวณปริมาณหน่วยความจำรวมที่ใช้ในการทำงานของโปรแกรมที่ใช้การพรีเฟ็ตช์วิธีที่ 13 และ 14 จากนั้นแทนค่าตัวแปรต่าง ๆ ยังสมการรวมทั้ง 2 กำหนดให้ M_{kk} แทนค่าด้วย 384 *D* แทนค่าด้วย 64 เนื่องจากการอ่านข้อมูลเข้ามายังแคชของสถาปัตยกรรมเครื่องที่ใช้ในการทดลองจะถูกอ่านมาทีละ 64 ไบต์ จากการปรับปรุงอัลกอริทึมในการทดลองก่อนหน้า T_k แทนค่าด้วย 128 T_i แทนค่าด้วย 64 และ *uF* แทนค่าด้วย 8 เมื่อแทนค่าต่าง ๆ ลงในขนาดเมทริกซ์ (*n*) ที่ต่างกันแสดงผลดัง

ตารางที่ 4-7 เปรียบเทียบปริมาณหน่วยความจำที่เข้าถึงระหว่างวิธีการพรีเฟ็ตช์ที่ 13 และ 14

Matrix Size (*N*×*N*)	Total amount of memory access (GB)	
	Pattern 13	Pattern 14
1024×1024	6.14	6.17
2048×2048	47.42	47.54
3072×3072	165.82	166.22
4096×4096	393.06	393.99
5120×5120	767.70	769.52
6144×6144	1326.59	1329.76
7168×7168	2106.57	2111.55
Average	687.614	689.25

จากตารางที่ 4-7 พบว่า โปรแกรมที่ใช้การพรีเฟ็ตช์วิธีการที่ 13 ใช้หน่วยความจำน้อยกว่าโปรแกรมที่ใช้วิธีการพรีเฟ็ตช์ที่ 14 โดยเฉลี่ยจากการทดลองทั้ง 7 ขนาดเมทริกซ์อยู่ 1.636 GB เนื่องจากตัวแปรที่เข้าถึงแถวของตัวตั้งอยู่การวนรอบในกว่าตัวแปรที่เข้าถึงหลักของตัวตั้งทำให้การพรีเฟ็ตช์แถวของตัวตั้งจะส่งผลมากกว่าการพรีเฟ็ตช์หลักของตัวตั้งนั่นเอง จากการวิเคราะห์

นี้จึงสรุปได้ว่าการพรีเฟ็ตช์ที่มีประสิทธิภาพมากที่สุดสำหรับอัลกอริทึมที่เหลือใช้คือโปรแกรมที่พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง แถวของเมทริกซ์ตัวคูณและแถวของเมทริกซ์ผลลัพธ์

```
for (int j = 0; j < N; j += 64)  // เพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณและผลลัพธ์
    for (int i = 0; i < N ; i += T_i) // แบ่งแถวของเมทริกซ์ตัวตั้งและผลลัพธ์
        for (int k = 0; k < N ; k += T_k) // แบ่งแถวของเมทริกซ์ตัวคูณและหลักของตัวตั้ง
            for (int ii = i; ii < i + T_i; ii++) // เข้าถึงแถวย่อยของเมทริกซ์ตัวตั้งและผลลัพธ์
                /* รีเซ็ตตัวแปรที่ใช้เก็บผลลัพธ์ */
                r_1= _mm256_setzero_ps();
                r_2 = _mm256_setzero_ps();
                :
                r_8 = _mm256_setzero_ps();
                // พรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง

                _mm_prefetch(&A[ii+d_A][k], HINT)
                // พรีเฟ็ตช์แถวของเมทริกซ์ผลลัพธ์
                _mm_prefetch(&C[ii+d_C][j], HINT)
                for (int kk = k; kk < k + T_k; kk++) { // เข้าถึงแถวและหลักย่อยของเมทริกซ์
                    a_1 = _mm256_set1_ps(A[ii][kk]); // อ่านข้อมูลของสมาชิกเมทริกซ์ตัวตั้ง
                    // พรีเฟ็ตช์แถวของเมทริกซ์ตัวคูณ
                    _mm_prefetch(&B[kk+d_B][j], HINT)
                    b_1 = _mm256_load_ps(&B[kk][j]); // อ่านข้อมูลของสมาชิกตัวคูณ
                    b_2 = _mm256_load_ps(&B[kk][j + 8]);
                    :
                    b_8 = _mm256_load_ps(&B[kk][j + (8 * 7)]);
```

รูปที่ 4-7 การใช้ชุดคำสั่ง AVX มาแทนที่คำสั่งการคูณเมทริกซ์ทั่วไป (อัลกอริทึมที่ 6)

```
/* คูณและบวกผลลัพธ์เก็บในตัวแปร r1 – r8*/
                r1 = _mm256_fmadd_ps(a1, b1, r1);
                r2 = _mm256_fmadd_ps(a1, b2, r2);
                 :
                r8 = _mm256_fmadd_ps(a1, b8, r8);
            }
/* อ่านค่าสมาชิกของเมทริกซ์ผลลัพธ์ */
            c1 = _mm256_load_ps(&C[ii][j]);
            c2 = _mm256_load_ps(&C[ii][j + 8]);
              :
            c8 = _mm256_load_ps(&C[ii][j + (8 * 7)]);
      /* นำตัวแปรที่เก็บผลลัพธ์จากการคูณเมทริกซ์บวกค่าเดิมของสมาชิกของเมทริกซ์ผลลัพธ์ */
            r1 = _mm256_add_ps(c1, r1);
            r2 = _mm256_add_ps(c2, r2);
              :
            r8 = _mm256_add_ps(c8, r8);
            /* นำผลลัพธ์ที่ได้บวกสะสมเก็บลงในสมาชิกของเมทริกซ์ผลลัพธ์ */
            _mm256_store_ps(&C[ii][j], r1);
            _mm256_store_ps(&C[ii][j + 8], r2);
            _mm256_store_ps(&C[ii][j + (8 * 2)], r3);
            :
            _mm256_store_ps(&C[ii][j + (8 * 7)], r8);
          }
       }
   }
}
```

รูปที่ 4-8 การใช้ชุดคำสั่ง AVX มาแทนที่คำสั่งการคูณเมทริกซ์ทั่วไป (อัลกอริทึมที่ 6) (ต่อ)

จากรูปที่ 4-7 และรูปที่ 4-8 เป็นอัลกอริทึมการคูณเมทริกซ์ที่ใช้อัลกอริทึมที่ 5 และเพิ่มการพรีเฟ็ตช์วิธีที่ 13 เข้าไป ซึ่งอัลกอริทึมที่ 6 นี้ประกอบด้วย การเพิ่มคำสั่งพรีเฟ็ตช์แถวของ

ตัวคูณเข้าไปที่การวนรอบชั้นในสุด การเพิ่มคำสั่งพรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้งและเมทริกซ์ผลลัพธ์เข้าไปที่การวนรอบชั้นรองในสุด การพรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้งและเมทริกซ์ผลลัพธ์ไม่สามารถเพิ่มเข้าไปที่การวนรอบชั้นในสุดเหมือนการพรีเฟ็ตช์แถวของเมทริกซ์ตัวคูณ เนื่องจากการวนรอบชั้นในสุดเป็นการวนรอบเพื่อเข้าถึงหลักของเมทริกซ์ตัวตั้งเพียงเท่านั้น ทำให้หากเพิ่มการพรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้งเข้าไปจะเป็นการพรีเฟ็ตช์ข้อมูลเดิมซ้ำและเกิดค่าโสหุ้ยมากกว่าระยะเวลาที่ลดลงของการประมวลผล ส่วนการใช้ข้อมูลของเมทริกซ์ผลลัพธ์เกิดหลังการวนรอบชั้นในสุด ดังนั้นหากเพิ่มการพรีเฟ็ตช์แถวของเมทริกซ์ผลลัพธ์เข้าไปจะเป็นการพรีเฟ็ตช์ข้อมูลเดิมซ้ำและเกิดค่าโสหุ้ยมากกว่าระยะเวลาที่ลดลงของการประมวลผลเช่นกัน

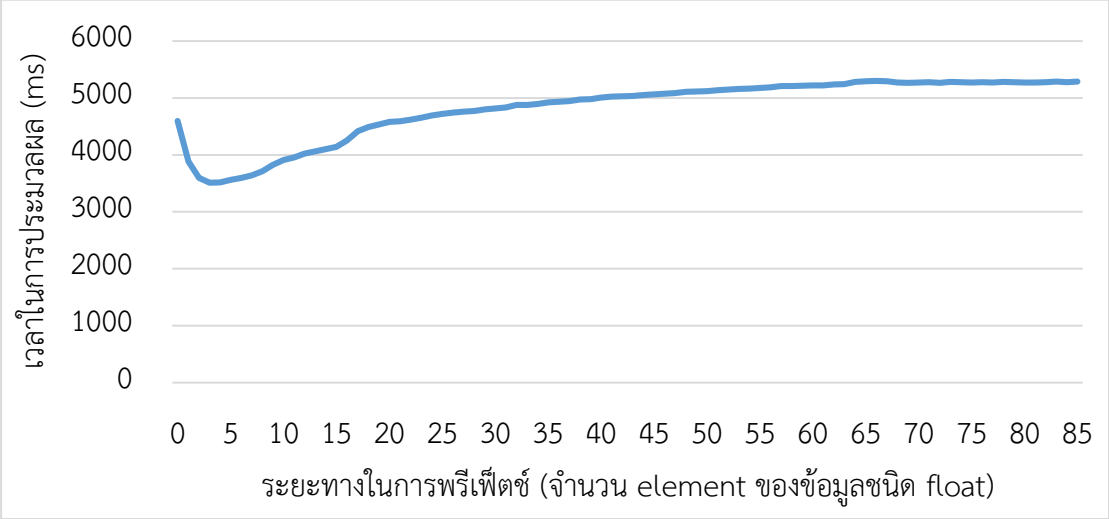
4.3. การหาระยะทางของการพรีเฟ็ตช์ที่เหมาะสม

เมื่อได้วิธีที่การพรีเฟ็ตช์ให้มีประสิทธิภาพมากที่สุดแล้ว ผู้วิจัยได้ทำการวิเคราะห์ตัวแปรที่สำคัญในการพรีเฟ็ตช์ จากอัลกอริทึมที่ 6 พบว่าการพรีเฟ็ตช์ตัวแปรที่เข้าถึงแถวของเมทริกซ์ตัวคูณเป็นตัวแปรที่สำคัญที่สุด เนื่องจากอัลกอริทึมที่ใช้การวนรอบในสุดเป็นการวนรอบเพื่อเข้าถึงแถวของเมทริกซ์ตัวคูณ ทำให้จำนวนครั้งในการอ่านข้อมูลของตัวแปร *kk* มีจำนวนมากที่สุด นอกจากนั้นตัวแปรที่เข้าถึงแถวของเมทริกซ์ตัวคูณยังเหมาะแก่การใช้เทคนิคคลายการวนซ้ำที่สามารถหลีกเลี่ยงการพรีเฟ็ตช์ข้อมูลเดิมซ้ำได้ ดังนั้นการพรีเฟ็ตช์ตัวแปรที่มีจำนวนครั้งในการอ่านข้อมูลมากที่สุดจึงสำคัญที่สุดนั่นเอง การทดลองหาระยะทางในการพรีเฟ็ตช์ต่างกันจึงให้ลำดับความสำคัญตัวแปรที่เข้าถึงแถวของเมทริกซ์ตัวคูณเป็นอันดับแรก ผู้วิจัยได้กำหนดระยะทางที่ทำการทดลองไว้ตั้งแต่ 0-d_{max} โดย d_{max} คำนวณได้จากสมการที่ (2)

$$d_{max} = M_{L1} / M_{kk} \quad (2)$$

d_{max} คือ ระยะทางในการพรีเฟ็ตช์ที่มากที่สุดก่อนจะทำให้หน่วยความจำแคช L1 เต็มส่งผลให้เมื่อหน่วยประมวลผลต้องการใช้ข้อมูลที่พรีเฟ็ตช์แต่ข้อมูลนั้นถูกนำออกจากแคช L1 ไปแล้วจะเกิดสถานะ cache miss ที่ L1 นั่นเอง M_{L1} คือขนาดของหน่วยความจำแคช L1 และ M_{kk} คือปริมาณข้อมูลต่อการวนรอบ *kk* ซึ่งเป็นรอบชั้นในสุดของอัลกอริทึมสามารถแทนค่าด้วย 384 ไบต์ และขนาดของหน่วยความจำแคช L1 ของเครื่องที่ใช้ทำการทดลองมีขนาด 32 KB ดังนั้นระยะทางในการพรีเฟ็ตช์ที่มากที่สุดก่อนจะทำให้หน่วยความจำแคช L1 เต็ม มีค่าเป็น 85.33 ผู้วิจัยจึงทำการทดลองประมวลผลโปรแกรมพรีเฟ็ตช์เพื่อวัดเวลาของการประมวลผลโดยใช้ฟังก์ชัน timer ซึ่งอยู่ภายใน timer.h เพื่อจับเวลาการทำงานของการคูณเมทริกซ์ ที่มีระยะห่างตั้งแต่ 0-85 ทั้งหมด 86 โปรแกรม โปรแกรมละ 30 ครั้งในเมทริกซ์แต่ละขนาด เพื่อหาค่าเฉลี่ยในเมทริกซ์ 7 ขนาด ได้แก่

ขนาด 1024×1024, 2048×2048, 3072×3072, 4096×4096, 5120×5120, 6144×6144 และ 7168×7168 จากนั้นนำผลการทดลองแต่ละขนาดมาเฉลี่ยกันมีผลการทดลองดังนี้



รูปที่ 4-9 การหาระยะทางการพรีเฟ็ตช์ของตัวแปรที่เข้าถึงแถวเมทริกซ์ตัวคูณของหน่วยประมวลผล Intel Core i5

จากรูปที่ 4-9 พบว่าระยะทางในการพรีเฟ็ตช์ที่เหมาะสมของตัวแปรที่เข้าถึงแถวของเมทริกซ์ตัวคูณคือ 3 element หลังจากนั้นผู้วิจัยได้ทำการวิเคราะห์และลำดับความสำคัญของตัวแปรที่ใช้ในการพรีเฟ็ตช์รองลงมา จากรูปที่ 4-6 พบว่าได้ว่าการพรีเฟ็ตช์ตัวแปรที่เข้าถึงแถวของเมทริกซ์ตัวตั้งมีความสำคัญรองลงมา เนื่องจากอัลกอริทึมที่ใช้การวนรอบในสุดเป็นการวนรอบเพื่อเข้าถึงหลักของเมทริกซ์ตัวคูณ ทำให้จำนวนครั้งในการอ่านข้อมูลของตัวแปร *kk* มีจำนวนมากที่สุด แต่ส่งผลเพียงเล็กน้อยเท่านั้นเนื่องจากตำแหน่งที่ใช้คำสั่งพรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้งอยู่ภายในการวนรอบชั้นรองลงมา แต่ตัวแปรที่เข้าถึงแถวของเมทริกซ์ตัวตั้งอยู่ภายในการวนรอบชั้นในสุด ทำให้การพรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้งแต่ละครั้งอ่านข้อมูลล่วงหน้าได้เพียง 1 line size ก่อนจะทำการเข้าถึงข้อมูลตัวตั้งของเมทริกซ์เท่านั้น ระยะทางในการพรีเฟ็ตช์ที่มากที่สุดก่อนจะทำให้หน่วยความจำแคช L1 เต็มจะเปลี่ยนไปจากสมการที่ (2) เนื่องจากตำแหน่งของการพรีเฟ็ตช์และข้อมูลที่ต้องใช้เปลี่ยนไป ผู้วิจัยได้กำหนดระยะห่างที่ทำการทดลองไว้ตั้งแต่ 0-d_{max} โดย d_{max} เพื่อพรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้งคำนวณได้จากสมการที่ (3)

$$d_{max} = M_{L1} / (tM_{kk}+M_{ii}) \quad (3)$$

M_{ii} คือปริมาณข้อมูลต่อการวนรอบ *ii* และ tM_{kk} คือปริมาณข้อมูลทั้งหมดของการวนรอบ *kk* หรือปริมาณข้อมูลต่อการวนรอบ *kk* คูณจำนวนรอบของการวนรอบ *kk* ดังสมการที่ (4)

$$tM_{kk} = M_{kk} \times N_{kk} \quad (4)$$

จากสมการที่ (4) สามารถแทนค่า M_{kk} เป็น 384 ไบต์และจำนวนการวนรอบ *kk* เท่ากับ T_k แทนค่าด้วย 64 ดังนั้นปริมาณข้อมูลทั้งหมดของการวนรอบ *kk* คือ 24576 ไบต์ จากสมการที่ (3) เมื่อแทนค่าขนาดของหน่วยความจำแคช L1 ของเครื่องที่ใช้ทำการทดลองมีขนาด 32 KB จะได้ระยะทางในการพรีเฟ็ตช์ที่มากที่สุดก่อนจะทำให้หน่วยความจำแคช L1 เต็มเป็น 1.31 ผู้วิจัยจึงทดลองประมวลผลโปรแกรมพรีเฟ็ตช์เพื่อวัดเวลาของการประมวลผลโดยใช้ฟังก์ชัน timer ซึ่งอยู่ภายใน timer.h เพื่อจับเวลาการทำงานของการคูณเมทริกซ์ ที่มีระยะห่างตั้งแต่ 0-1 ทั้งหมด 2 โปรแกรม โปรแกรมละ 30 ครั้งในเมทริกซ์แต่ละขนาด เพื่อหาค่าเฉลี่ยในเมทริกซ์ 7 ขนาด ได้แก่ ขนาด 1024×1024, 2048×2048, 3072×3072, 4096×4096, 5120×5120, 6144×6144 และ 7168×7168 มีผลการทดลองดังนี้

ตารางที่ 4-8 เวลาการประมวลผลของระยะทางที่ 0 และ 1 ของการพรีเฟ็ตช์แถวของเมทริกซ์ตัวตั้ง

ขนาด / ระยะทาง	1024	2048	3072	4096	5120	6144	7168	เฉลี่ย
0	56.93	431.85	1405.5	3435.38	6441.95	11256.89	17941.68	5852.89
1	58.86	427.80	1394.47	3425.14	6416.81	11202.49	17931.74	5836.76

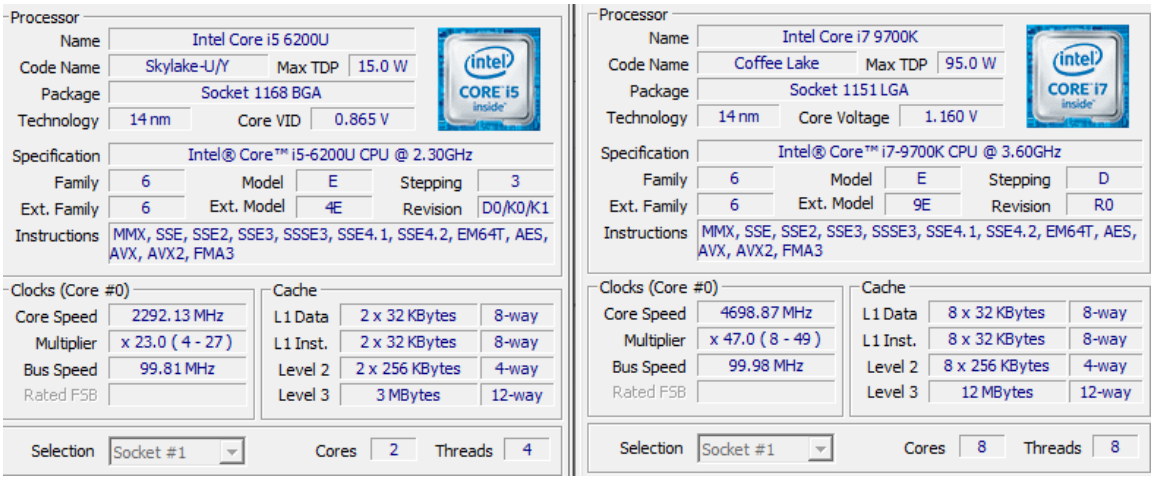
จากตารางที่ 4-8 พบว่าระยะทางในการพรีเฟ็ตช์ที่เหมาะสมของตัวแปรที่เข้าถึงแถวของเมทริกซ์ตัวตั้งคือ 1 element เมื่อวิเคราะห์จากการลำดับคำสั่ง หากระยะทางเป็น 0 การอ่านข้อมูลจากการพรีเฟ็ตช์จะช้าเกินกว่าที่ข้อมูลจะใช้งาน เนื่องจากคำสั่งในการพรีเฟ็ตช์อยู่ห่างจากคำสั่งที่ต้องอ่านข้อมูลแถวของเมทริกซ์ตัวตั้งเพียง 2 คำสั่ง ซึ่งหากระยะทางมากกว่า 1 จะทำให้ข้อมูลที่ต้องการใช้ถูกนำออกจากหน่วยความจำแคช L1 ได้ หลังจากนั้นผู้วิจัยได้วิเคราะห์ระยะทางในการพรีเฟ็ตช์ของตัวแปรที่เข้าถึงแถวของเมทริกซ์ผลลัพธ์ ผู้วิจัยได้กำหนดระยะห่างที่ทำการทดลองไว้ตั้งแต่ 0-d_{max} โดย d_{max} เพื่อพรีเฟ็ตช์แถวของเมทริกซ์ผลลัพธ์คำนวณได้จากสมการที่ (5)

$$d_{max} = M_{L1} / (tM_{kk}+M_{ii}) - 1 \quad (5)$$

เมื่อแทนค่าสมการพบว่าระยะทางที่เหมาะสมคือ 0 เนื่องจากคำสั่งที่ใช้อ่านเมทริกซ์ผลลัพธ์อยู่ถัดจากการวนรอบในสุดดังรูปที่ 4-6 ทำให้มีการอ่านข้อมูลเข้าสู่หน่วยความจำแคช L1 จำนวน 24576 ไบต์ก่อนที่จะถูกใช้ข้อมูลที่พรีเฟ็ตช์มา เหตุผลดังกล่าวจึงทำให้ในสมการที่ (5) มีการลบ 1 จากสมการที่ (4)

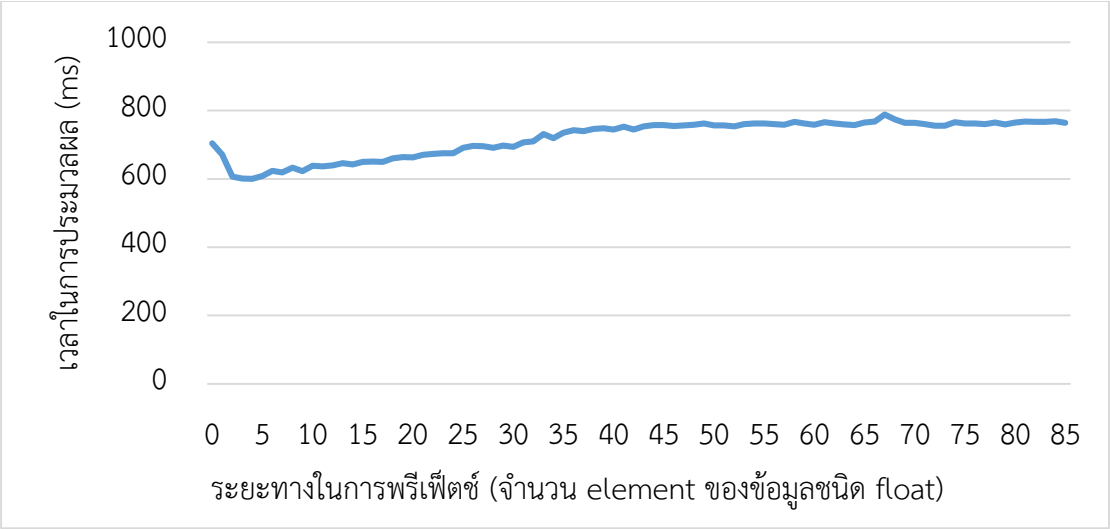
4.4. การทดสอบประสิทธิภาพของโปรแกรมการคูณเมทริกซ์ที่นำเสนอบนเครื่องที่มีทรัพยากรต่างกัน

การทดสอบประสิทธิภาพของโปรแกรมการคูณเมทริกซ์ที่นำเสนอบนเครื่องที่มีทรัพยากรต่างกันหาระยะทางในการพรีเฟ็ตช์ต่างกัน ผู้วิจัยได้ทำการทดลองกับคอมพิวเตอร์ที่ต่างกัน 2 เครื่องดังรูปที่ 4-10



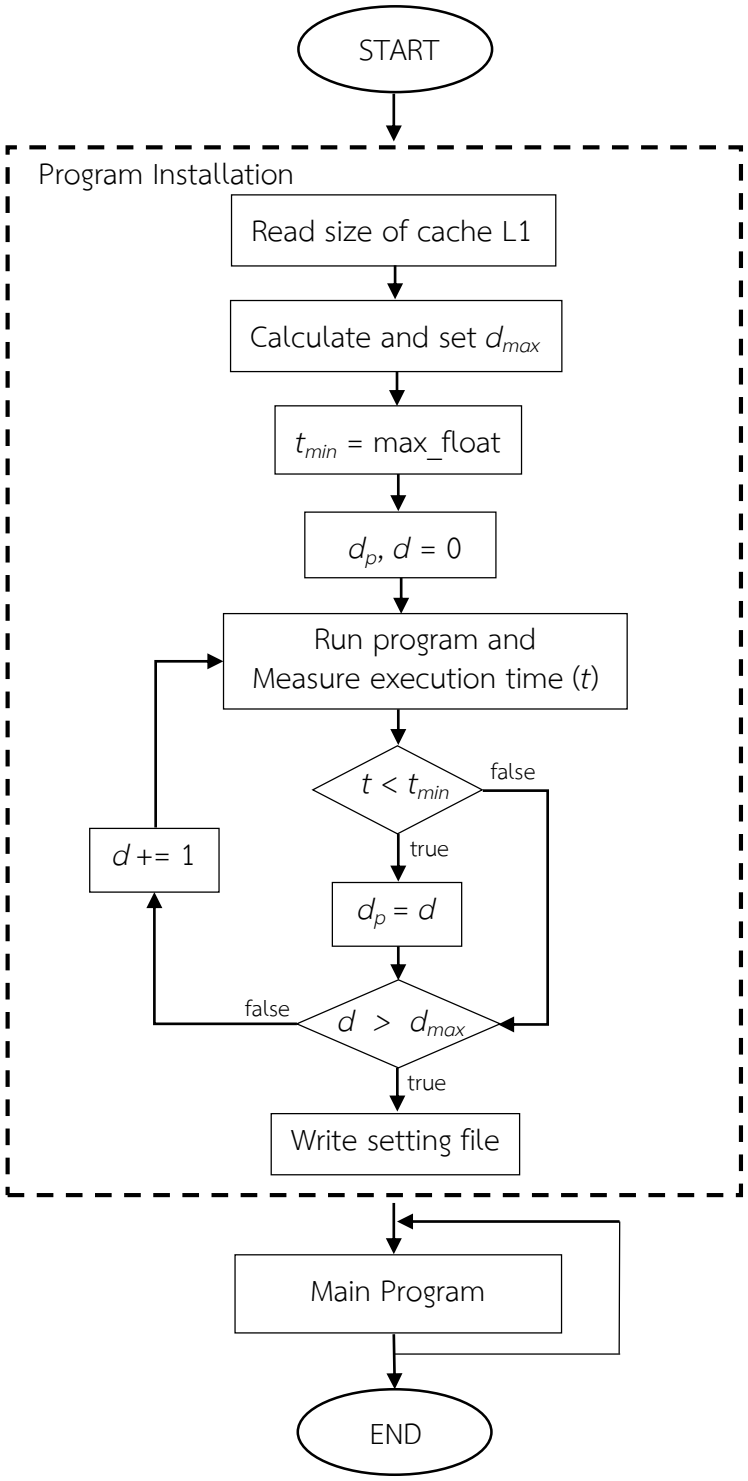
รูปที่ 4-10 ข้อมูลคอมพิวเตอร์ที่ใช้ในการทดลอง

ผู้วิจัยได้ทำตามวิธีการดำเนินงานดังรูปที่ 3-3 โดยมีการหาค่าระยะทางการพรีเฟ็ตช์ของเมทริกซ์ตัวคูณทั้ง 2 เครื่อง คอมพิวเตอร์ที่ใช้หน่วยประมวลผล Intel Core i5 และ Intel Core i7 มีระยะทางการพรีเฟ็ตช์แถวของเมทริกซ์ตัวคูณดังรูปที่ 4-9 และ ดังรูปที่ 4-11 ตามลำดับ



รูปที่ 4-11 การหาระยะทางการพรีเฟ็ตช์ของตัวแปรที่เข้าถึงแถวเมทริกซ์ตัวคูณของหน่วยประมวลผล Intel Core i7

ระยะทางการพรีเฟ็ตช์ที่ดีที่สุดในการพรีเฟ็ตช์แถวเมทริกซ์ตัวคูณของหน่วยประมวลผล Intel Core i5 คือ 3 element ของข้อมูลชนิด float และระยะทางการพรีเฟ็ตช์ที่ดีที่สุดในการพรีเฟ็ตช์แถวเมทริกซ์ตัวคูณของหน่วยประมวลผล Intel Core i7 คือ 4 element ของข้อมูลชนิด float ผลการทดลองที่ต่างกันเนื่องจากขนาดของแคชเป็นปัจจัยหลักในการเก็บข้อมูลที่รอประมวลผล จากผลการทดลองในการหาระยะทางการพรีเฟ็ตช์แถวของเมทริกซ์ตัวคูณระยะทางที่ดีที่สุดคือจุดที่ต่ำสุดของกราฟ ผู้จัดทำจึงมีแนวคิดในการหาระยะทางพรีเฟ็ตช์ที่เหมาะสมทั้งหมดในขั้นตอนการติดตั้งโปรแกรม ซึ่งในขั้นตอนการติดตั้งโปรแกรมนี้จะมีการเรียกใช้ส่วนของโปรแกรมที่จะทำการวัดระยะทางพรีเฟ็ตช์ที่ดีที่สุด การวัดระยะทางดังกล่าวจะถูกเรียกใช้เพียงครั้งเดียวและมีการบันทึกค่าระยะทางที่ดีที่สุดเอาไว้ในไฟล์ และทุกครั้งที่โปรแกรมถูกเรียกขึ้นมาใช้งานจะมีการอ่านค่าระยะทางที่บันทึกไว้นี้ในการสั่งการโปรแกรมเพื่อให้โปรแกรมได้ค่าที่ดีที่สุดไปใช้งานบนเครื่องคอมพิวเตอร์เครื่องนั้น ๆ รูปที่ 4-12 แสดงการทำงานของโปรแกรมดังกล่าว



รูปที่ 4-12 แผนผังแบบจำลองที่นำเสนอ

จากรูปที่ 4-12 แผนผังแบบจำลองเริ่มจากการติดตั้งโปรแกรม ซึ่งภายในขั้นตอนการติดตั้งประกอบไปด้วยการอ่านข้อมูลทรัพยากรเครื่องหรืออ่านค่าขนาดของหน่วยความจำแคช L1 เพื่อใช้ในการหาค่าระยะทางที่เหมาะสมจากแบบจำลองที่นำเสนอ จากนั้นตั้งค่าระยะทางที่เหมาะสมให้แก่โปรแกรม โดยกำหนดให้ t คือเวลาที่ใช้ประมวลผล t_{min} คือเวลาที่ใช้ประมวลผลน้อยที่สุด d_p คือระยะทางในการพรีเฟ็ตช์ d คือระยะทางพรีเฟ็ตช์ที่ทดลอง จากนั้นรันเพื่อหาค่าระยะทางพรีเฟ็ตช์ที่ใช้เวลาในการประมวลผลน้อยที่สุดเพื่อกำหนดระยะทางที่เหมาะสมในการพรีเฟ็ตช์ลงในไฟล์ตั้งค่า การติดตั้งโปรแกรมทำเพียงครั้งเดียวต่อหนึ่งเครื่องคอมพิวเตอร์ที่ใช้ประมวลผล หากเปลี่ยนคอมพิวเตอร์ในการประมวลผลต้องติดตั้งโปรแกรมใหม่เพื่อหาระยะทางพรีเฟ็ตช์ที่เหมาะสมกับเครื่องคอมพิวเตอร์ที่ใช้ประมวลผลนั้น เมื่อติดตั้งโปรแกรมเสร็จจะสามารถดำเนินงานโปรแกรมได้โดยเริ่มจากโปรแกรมจะอ่านค่าจากไฟล์ตั้งค่าและทำการประมวลผลต่อไป

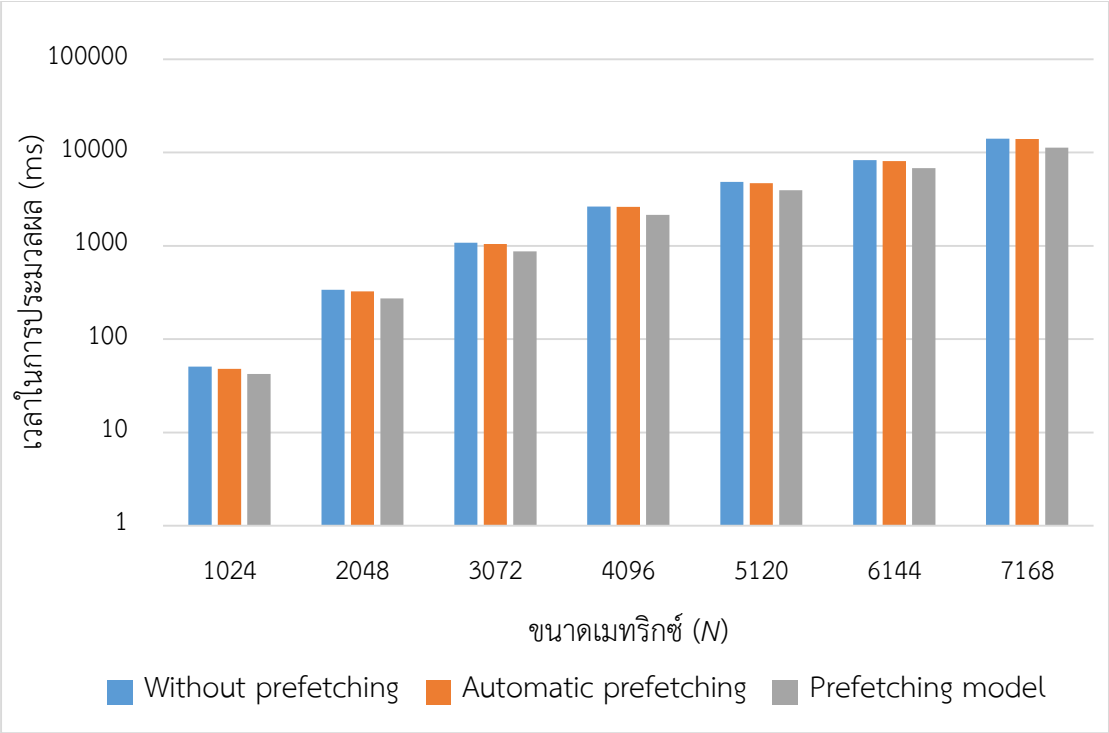
การกำหนดระยะทางในการพรีเฟ็ตช์ที่เข้าถึงแถวของเมทริกซ์ตัวตั้ง ตัวคูณและผลลัพธ์ในแต่ละเครื่องคอมพิวเตอร์สามารถสรุปได้ดังตารางที่ 4-9

ตารางที่ 4-9 ระยะทางการพรีเฟ็ตช์ตัวแปรแต่ละตัวบนคอมพิวเตอร์ที่ต่างกัน

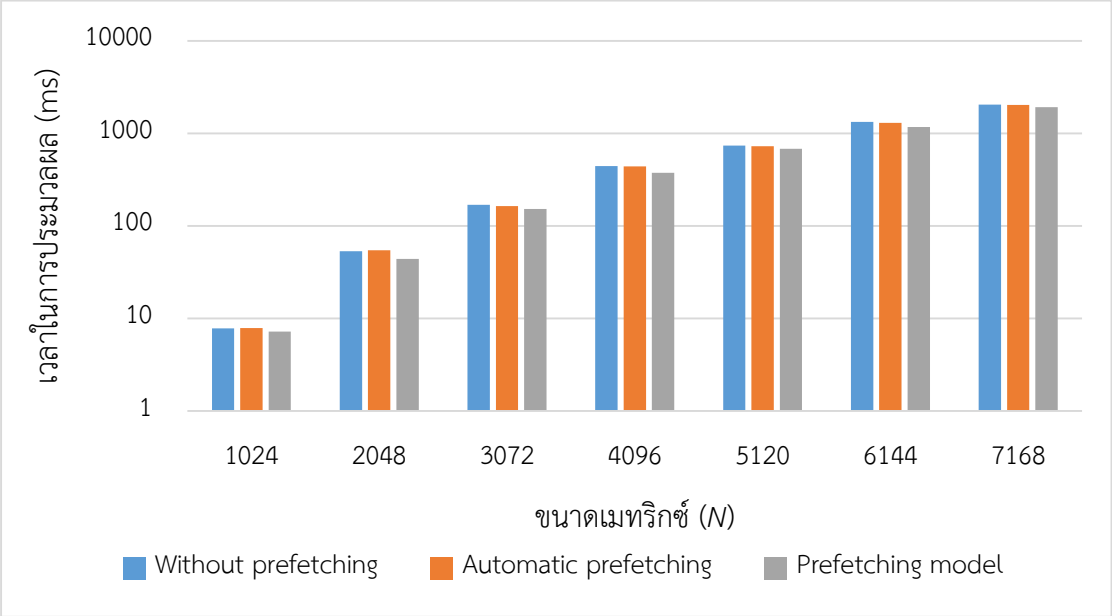
การพรีเฟ็ตช์ / หน่วยประมวลผล	แถวเมทริกซ์ตัวตั้ง	แถวเมทริกซ์ตัวคูณ	แถวเมทริกซ์ผลลัพธ์
Intel Core i5	1	3	0
Intel Core i7	1	4	0

ระยะทางในการพรีเฟ็ตช์ที่เหมาะสมของแถวเมทริกซ์ตัวตั้งและเมทริกซ์ผลลัพธ์ของคอมพิวเตอร์ทั้งสองเครื่องเท่ากันเนื่องจากมีหน่วยความจำแคช L1 ที่เท่ากันในหนึ่ง Core ทำให้เมื่อแทนค่าลงในสมการที่ (3) และ (5) ได้เท่ากัน

เมื่อนำเวลาที่ใช้ประมวลผลของโปรแกรมการคูณเมทริกซ์ที่ใช้การพรีเฟ็ตช์วิธีการที่นำเสนอเปรียบเทียบกับโปรแกรมการคูณเมทริกซ์ที่ไม่มีการพรีเฟ็ตช์ และโปรแกรมการคูณเมทริกซ์ที่มีการพรีเฟ็ตช์อัตโนมัติจาก Intel C++ คอมไพเลอร์เวอร์ชัน 19.2 มีผลการทดลองดังรูปที่ 4-13 และรูปที่ 4-14



รูปที่ 4-13 เวลาในการประมวลผลบนเครื่อง Intel Core i5 ของโปรแกรม 3 ตัว เมื่อเมทริกซ์มีขนาดแตกต่างกัน



รูปที่ 4-14 เปรียบเทียบเวลาในการประมวลผลของ 3 โปรแกรมในเมทริกซ์ที่ขนาดต่างกันของหน่วยประมวลผล Intel Core i7

ผลการทดลองเปรียบเทียบเวลาในการประมวลผลของ 3 โปรแกรมในเมทริกซ์ที่ขนาดต่างกันของหน่วยประมวลผล Intel Core i5 และ Intel Core i7 พบว่าโปรแกรมที่พรีเฟ็ตช์โดย

ใช้วิธีการที่นำเสนอใช้เวลาในการประมวลผลน้อยที่สุดในทุกขนาดของเมทริกซ์ ในคอมพิวเตอร์ที่ใช้หน่วยประมวลผล Intel Core i5 โปรแกรมที่ใช้วิธีการพรีเฟ็ตช์ที่นำเสนอสามารถประมวลผลได้เร็วกว่าโปรแกรมที่ไม่ได้ใช้คำสั่งพรีเฟ็ตช์โดยเฉลี่ยอยู่ร้อยละ 18.86 และโปรแกรมการคูณเมทริกซ์ที่มีการพรีเฟ็ตช์อัตโนมัติจาก Intel C++ คอมไพเลอร์โดยเฉลี่ยอยู่ร้อยละ 17.54 ในคอมพิวเตอร์ที่ใช้หน่วยประมวลผล Intel Core i7 โปรแกรมที่ใช้วิธีการพรีเฟ็ตช์ที่นำเสนอสามารถประมวลผลได้เร็วกว่าโปรแกรมที่ไม่ได้ใช้คำสั่งพรีเฟ็ตช์โดยเฉลี่ยอยู่ร้อยละ 8.86 และโปรแกรมการคูณเมทริกซ์ที่มีการพรีเฟ็ตช์อัตโนมัติจาก Intel C++ คอมไพเลอร์โดยเฉลี่ยอยู่ร้อยละ 7.73

บทที่ 5
สรุปผลวิจัยและข้อเสนอแนะ

5.1. สรุปผลการวิจัย

การดึงข้อมูลล่วงหน้าเป็นเทคนิคที่สำคัญในการลดระยะเวลาในการอ่านข้อมูล โดยเฉพาะเมื่อมีการประมวลผลข้อมูลจำนวนมาก การพรีเฟ็ตช์ที่มีประสิทธิภาพต้องกำหนดวิธีการและระยะทางในการพรีเฟ็ตช์ที่เหมาะสมกับอัลกอริทึมที่ใช้ ในการวิจัยนี้ได้นำเสนอการวิเคราะห์เพื่อเลือกวิธีการพรีเฟ็ตช์ที่ดีที่สุด โดยมีการกำหนดวิธีทั้งหมดที่สามารถเกิดขึ้นได้ จากนั้นวิเคราะห์ในแต่ละวิธีจากการคำนวณจำนวนคำสั่งที่ใช้ในการพรีเฟ็ตช์และปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์ การเลือกวิธีที่ดีที่สุดจะเลือกวิธีที่มีจำนวนครั้งในใช้คำสั่งพรีเฟ็ตช์น้อยที่สุด แต่ปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์มากที่สุด เมื่อเลือกวิธีการพรีเฟ็ตช์ที่ดีที่สุดแล้วผู้วิจัยได้ทำการทดลองเพื่อหาระยะทางที่เหมาะสมในการพรีเฟ็ตช์ จากการวิเคราะห์ขนาดหน่วยความจำของแคชและคำนวณปริมาณการใช้หน่วยความจำในอัลกอริทึมที่ใช้ เพื่อหาระยะทางในการพรีเฟ็ตช์ที่มากที่สุดก่อนจะทำให้หน่วยความจำแคช L1 เกิดสถานะ cache miss เนื่องจากหน่วยความจำเต็มและนำข้อมูลออกก่อนจะเกิดการเรียกใช้ จากนั้นดำเนินงานโปรแกรมที่ระยะทางต่างกันเพื่อหาหาระยะทางที่ดีที่สุดในการพรีเฟ็ตช์ โดยโปรแกรมพรีเฟ็ตช์ที่นำเสนอสามารถประมวลผลได้เร็วกว่าโปรแกรมที่ไม่ได้รับการพรีเฟ็ตช์และโปรแกรมการคูณเมทริกซ์ที่มีการพรีเฟ็ตช์อัตโนมัติจาก Intel C++ คอมไพเลอร์

5.2. ปัญหาและอุปสรรค

การหาวิธีการพรีเฟ็ตช์ต้องใช้ความเข้าใจในอัลกอริทึมที่ประมวลผลว่ามีการดึงข้อมูลในปริมาณเท่าใดในตำแหน่งไหนของโปรแกรมบ้าง เพื่อวิเคราะห์จำนวนคำสั่งที่ใช้ในการพรีเฟ็ตช์และปริมาณข้อมูลที่ได้จากการพรีเฟ็ตช์ ซึ่งต้องหลีกเลี่ยงกรณีที่เกิดการพรีเฟ็ตช์ซ้ำในข้อมูลตำแหน่งเดิมเพราะจะทำให้เกิดค่าโสหุ้ยได้ นอกจากนั้นในการทดลองเพื่อหาระยะทางที่ดีที่สุดในการพรีเฟ็ตช์มีจำนวนครั้งที่มากในการประมวลผลโปรแกรมในระยะทางต่าง ๆ จึงเกิดแนวคิดให้ทำการติดตั้งโปรแกรมเพื่อหาระยะทางที่ดีที่สุดเพียงครั้งเดียวต่อหนึ่งเครื่องคอมพิวเตอร์ก่อนการดำเนินงานโปรแกรมต่อไป

5.3. ข้อเสนอแนะ

การวิเคราะห์โปรแกรมที่ต้องการเพิ่มคำสั่งพรีเฟ็ตช์สามารถทำได้ดีในโปรแกรมที่มีอัลกอริทึมเรียกใช้งานที่มีรูปแบบการทำงานที่ซ้ำกันอย่างมีแบบแผน สามารถทำการวางแผนการอ่านข้อมูลล่วงหน้าได้ การพรีเฟ็ตช์โปรแกรมที่มีการเรียกใช้ปริมาณข้อมูลน้อยไม่เหมาะสมแก่การเพิ่มคำสั่งพรีเฟ็ตช์เพราะทำให้เกิดค่าโสหุ้ยได้ ดังนั้นโปรแกรมที่สามารถเพิ่มคำสั่งพรีเฟ็ตช์ที่ดีคือโปรแกรมที่มีการประมวลผลข้อมูลในปริมาณมากและมีรูปแบบซ้ำกันอย่างมีแบบแผนในการใช้ข้อมูลที่สามารถทำการวางแผนการอ่านข้อมูลล่วงหน้าได้

งานในอนาคตสามารถนำแบบจำลองที่นำเสนอไปใช้กับโปรแกรมอื่นที่มีรูปแบบโปรแกรมที่ซับซ้อนมากขึ้นได้ เช่น คำสั่งภายในการวนรอบมีการอ่านข้อมูลเพื่อใช้ในการประมวลผลมากขึ้น การอ่านข้อมูลที่เป็นอาเรย์ช่วงสั้นอาจทำให้มีปัจจัยเพิ่มเติมเพื่อหาตำแหน่งระยะทางพรีเฟ็ตช์ที่เหมาะสม และในกรณีที่ความจุของหน่วยความจำแคชมีน้อย จำนวนคำสั่งภายในการวนรอบมีมากขึ้นหรือเป็นการพรีเฟ็ตช์ชนิดตัวแปรที่มีขนาดใหญ่ขึ้น จะส่งผลให้จำนวนครั้งการวนรอบเพื่อหาระยะทางที่เหมาะสมลดลง แต่ไม่ได้ส่งผลให้ระยะเวลาในการติดตั้งโปรแกรมน้อยลง หากโปรแกรมมีขนาดใหญ่ในการรันโปรแกรมเพื่อหาระยะทางที่เหมาะสมยังต้องใช้เวลาเพื่อติดตั้งโปรแกรม

เอกสารอ้างอิง

[1] Ahmad and M. A. Pasha, “Optimizing Hardware Accelerated General Matrix-Matrix Multiplication for CNNs on FPGAs,” in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 67, no. 11, pp. 2692-2696, Nov. 2020, doi: 10.1109/TCSII.2020.2965154

[2] H. Amiri and A. Shahbahrami, “High performance implementation of 2D convolution using Intel's advanced vector extensions,” Artificial Intelligence and Signal Processing Conference (AISP), 25-27 Oct. 2017.

[3] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes SIGARCH Comput,” Archit. *News 22,* Apr. 1994*,* pp. 223–232*.* doi: *https://doi.org/10.1145/192007.192030*

[4] D. Choi, K. Kim, and E.y. Chung, “Asymmetric Prefetching Architecture for Multicore Processor,” International SoC Design Conference (ISOCC), 21-24 Oct. 2020.

[5] J. Lu, H. Chen, P. C. Yew, and W. C. Hsu, “Design and implementation of a lightweight dynamic optimization system,” J. Instr. Parallelism, vol. 6, pp. 1–24, 2004.

[6] W. Zhang, B. Calder, and D. M. Tullsen, “A self-repairing prefetcher in an event-driven dynamic optimization framework,” Proc. CGO 2006 - 4th Int. Symp. Code Gener. Optim., no. Cgo, pp. 12–64, 2006, doi: 10.1109/CGO.2006.4.

[7] J. Lee, H. Kim, and R. Vuduc, “When prefetching works, when it doesn’t, and why,” *Trans. Archit. Code Optim.*, vol. 9, no. 1, 2012, doi: 10.1145/2133382.2133384.

[8] “Intel® 64 and IA-32 Architectures Software Developer’s Manual,” 2016. Accessed: Sep. 2020. [Online]. Available: ห http://www.intel.com/design/literature.htm.

[9] N. Anchev, M. Gusev, S. Ristov and B. Atanasovski, "Some optimization techniques of the matrix multiplication algorithm," Proceedings of the ITI 2013

35th International Conference on Information Technology Interfaces, Cavtat, 2013, pp. 71-76, doi: 10.2498/iti.2013.0572.

[10] S. A. Hassan, M. M. M. Mahmoud, A. M. Hemeida, and M. A. Saber, “Effective Implementation of Matrix–Vector Multiplication on Intel’s AVX multicore Processor,” Comput. Lang. Syst. Struct., vol. 51, 2018, pp. 1339–1351, doi: 10.1016/j.cl.2017.06.003.

[11] N. Eiron, M. Rodeh, and T. Steinwarts, “Matrix Multiplication: A Case Study of Enhanced Data Cache Utilization,” ACM J. Exp. Algorithmics, vol. 4, no. 212, 1999, p. 3, doi: 10.1145/347792.347806.

ภาคผนวก

ภาคผนวก ก

ผลงานตีพิมพ์และเผยแพร่

Varintorn Khomongkonudom; Panyayot Chaikan

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


IEIE
The Institute of Electronics and Information Engineers
ECTI Association
EIC
Certificate of Presentation
This is to certify that
Varintorn Khomongkonudom
presented a paper titled
Optimum Prefetching Patterns Searching:
A Case Study of Matrix-Matrix Multiplication
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

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

IEIE The Institute of Electronics and Information Engineers

EIC ECTI Association

Optimum Prefetching Patterns Searching: A Case Study of Matrix-Matrix Multiplication

Varintorn Khomongkonudom
Department of Computer Engineering,
Prince of Songkla University
varintorn.k12m@gmail.com

Panyayot Chaikarn
Department of Computer Engineering,
Prince of Songkla University
panyayot@coe.psu.ac.th

***Abstract*—Prefetching reduces data fetch latency and augments the speed of program execution. This paper presents an analysis model for selecting the optimum prefetching pattern for matrix-matrix multiplication. By calculating the number of prefetch instructions and the amount of data from prefetching, numbers of prefetching candidates are selected. Then the best prefetching pattern is obtained by selecting the candidate with lowest value of memory access. The prefetching pattern obtained from this algorithm is on average 18.86 percent faster than without prefetching. It is on average 17.54 percent faster than the automatic prefetching feature provided by the Intel C compiler.**

Keywords—software prefetching, matrix-matrix multiplications, memory access, algorithm

I. INTRODUCTION

Processing large amounts of data takes a lot of execution time. Each application program has a specific memory access pattern, so if its data is properly read from main memory, then the program execution time can be reduced. Prefetching is a process of data pre-reading from main memory and then storing it in the cache hierarchy. This process can be implemented in both hardware and software approach. Hardware prefetching works well when the data access behavior of the program is clear and easy to predict [1]. It analyzes the trend of data usage and then predicts and reads the data in advance from main memory to the cache for future use. Software prefetching works by inserting the special machine level instruction inside the application to read the data before it is acually needed. The prefetching instruction will put the data inside the cache with a hope that they will be accessed in the near future. There are two ways of software prefetching: 1) automatic software prefetching and 2) manual software prefetching. Automatic software prefetching is done by the compiler. The compiler will analyze the behavior of an application, then automatically insert the prefetching machine instruction inside the object code. Some compilers support this feature e.g. Intel C compiler. Manual software prefetching is done by the programmers. The compiler's intrinsic prefetching functions are manually inserted inside the high-level language source code of the program. Software prefetching might slow down the program execution if not used properly.

II. BACKGROUND AND RELATED WORK

When data is prefetched in the program, there are 3 states of prefetching result that might happened in the system including: 1) Early state which occurs when the data required by processor has not yet arrived because it is still in the prefetching process; 2) Timely state which occurs when the desired data has already been inside the cache, and 3) late state which occurs when the processor needs the data but the prefetched data has already been removed from the cache.

The best state in the prefetching is the timely state, which eliminates the need for the processor to wait for the required data. The x86 architecture has 4 types of prefetching instruction: 1) PREFETCHT0 which reads data into the cache at all levels; 2) PREFETCHT1 which reads data into the cache level 2 or higher. This instruction should be used when the same data will be reused in level 1 cache; 3) PREFETCHT2 which reads the data into the cache level 3 or higher; 4) PREFETCHNTA which reads data into the level 1 cache memory. This instruction should be used when the same data is required again in the future, eliminating the need to store data in a different cache level. These 4 instructions can called by using intrinsic instructions in C programming.

Prefetch distance must be set to an appropriate value to achieve the best performance [2]. It is also utilized in conjunction with loop unrolling and blocking to improve the speed of matrix-vector multiplication [3]. When applied together with loop unrolling, prefetching reduces reading overhead of the data from the same address and increases the spatial locality of reference of the data. The prefetch distance can be calculated by using [4]

$$d = \frac{L_{miss}}{CPI} \quad (1)$$

where d is the distance in bytes, which indicates how far from the current position to the desired reading location. The L_{miss} stands for the average miss latency which is the amount of time to read the data into the cache when cache miss occurs, and the *CPI* stands for the number of cycles required per loop iteration. However, this equation applicable only with the single loop operation, but when the loop is nested, the number of clock cycles will change and may results in a prefetch time out. Trident framework was invented to solve this problem [5]. It is an event-driven and multithreaded dynamic optimization framework, and is used to create a helper thread to check the Delinquent load table. If the prefetch is not effective enough, it will be marked to be prefetched elsewhere. The values are predicted and corrected in the table later to be used to increase or decrease the distance in the next prefetch. Using this framework requires machine resources to create a helper thread. However, this method works well if the amount of work to be processed is large enough due to its overhead.

Matrix multiplication is a basic mathematical operation required in science and engineering [6]. Large matrix takes a long time to process. Software tuning is the cheapest way to

augment the execution speed of a program. Software tuning of matrix multiplication can be done in several ways such as modifying the algorithm [7], reducing the number of mathematical operations, parallel programming, loop unrolling [8], etc. Prefetching is another way to increase processing speed.

In this paper, we propose a new method to find an optimized prefetch location and its distance based on the cache access analysis. We use a matrix-matrix multiplication as our case study because it is a time consuming operation and is required by many scientific and engineering applications.

III. PROPOSED METHOD

Many optimization approaches to augment the processing speed have been proposed e.g. loop unrolling [3], code blocking [3], parallelization, and the utilization of SIMD instructions [9]. Prefetching is another one technique that can improve the performance of matrix-matrix multiplication. Prefetch distance must be set to an appropriate value to achieve the best performance [2]. It is also utilized in conjunction with loop unrolling and blocking to improve the speed of matrix-vector multiplication [3]. When applied together with loop unrolling, prefetching reduces reading overhead of main memory and the execution speed of a program is increased.

Matrix-matrix multiplication is a time-consuming operation and is widely utilized in scientific applications. When the matrix size is large, it can take a long time to process. In this paper, we propose the utilization of prefetching to augment the execution speed of matrix-matrix multiplication. Figure 1 shows the multiplication algorithm with 4 different prefetch locations.

```
for (j=0; j<N; j += AVX_SZ*uF)
  for(i=0; i<N; i +=T_i)
    for(k=0; k<N; k +=T_k)
      for(ii=i; ii<i+T_i; ii++)
        prefetch A[ii+d_A][k]      // location a. prefetch a row of matrix A
        prefetch A[ii][k+d_A]      // location b. prefetch a column of matrix A
        prefetch C[ii +d_C][j]     // location d. prefetch a row of matrix C
        for(kk=k; kk<k+T_k; kk++)
          read A[ii][k] into AVX registers
          prefetch B[kk+d_B][j]    // location c. prefetch a row of matrix B
          read B[kk][j] into AVX registers
          read C[ii][j] into AVX registers
          // AVX code for add and multiply member of matrix
          C[ii][j+AVX_SZ*0] += A[ii][kk] * B[kk][j+AVX_SZ*0]
          C[ii][j+AVX_SZ*1] += A[ii][kk] * B[kk][j+AVX_SZ*1]
          …
          C[ii][j+AVX_SZ* (uF-1)] += A[ii][kk] * B[kk][j+AVX_SZ*(uF-1)]
```

Figure 1. The algorithm of matrix-matrix multiplication.

Let A, B, and C be square matrices of size $N \times N$. The product of matrix-matrix multiplication can be defined as $C = A \times B$. Figures 1 shows pseudo-code of the matrix-matrix multiplication using loop unrolling and loop tiling as the optimization techniques. This algorithm utilizes a fused-multiply-accumulate (FMA) operation of the AVX intrinsics to implement the multiplication. The variable AVX_{sz} is the amount of single-precision floating-point data that can be processed simultaneously using the AVX engine of the processor. The uF stands for the unrolling *factor* which can be set between 2-8. The T_k and T_i stand for the size of the data tile. D_A, D_B, and D_C stand for the prefetch distances of matrix A, B, and C, respectively. The innermost loop accesses the columns of matrix A columns and the rows of matrix B. The second level loop accesses the rows of matrix A and the columns of matrix B. The third level loop access the columns of matrix A and the rows of matrix B. This loop serves to move the tile position of the A matrix and B matrix. The fourth level loop access the rows of both matrix A and matrix C. The outermost loop accesses the columns both matrix B and matrix C.

Prefetching columns of matrix B and columns of matrix C is useless because the outermost loop variable is utilized to store the data, and they will be expelled from the cache memory before the processor can use them. For this reason, there are only 4 locations to insert the prefetch instruction. There are 15 possible prefetching combinations, as shown in Figure 2.

Prefetching locations
a) a row of matrix A
b) a column of matrix A
c) a row of matrix B
d) a row of matrix C

Possible Combinations	
1) a	9) b+d
2) b	10) c+d
3) c	11) a+b+c
4) d	12) a+b+d
5) a+b	13) a+c+d
6) a+c	14) b+c+d
7) a+d	15) a+b+c+d
8) b+c	

Figure 2. Possible prefetching patterns.

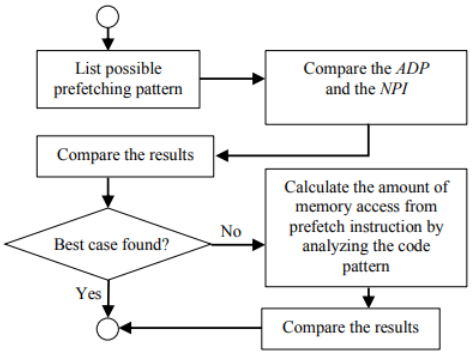


Figure 3. Flowchart of finding the best prefetching pattern.

Since there are 15 possible prefetching patterns, we need to find out which one will produce the best result. Figure 3 shows the process of finding the best prefetching pattern. The flowchart starts from calculating number of prefetching instructions (*NPI*) of each pattern. This value depends on the size of matrix (N), the column size of the tile (T_k), and the size of data elements per AVX instruction (E). The next step is to calculate the amount of data from prefetching (*ADP*). Most *ADPs* are equal to *NPI* multiplied by the data per prefetch instruction (D) unless the data is prefetched at the same position. The best pattern requires high ADP and low NPI values. If a single best pattern is not found, then it is necessary to calculate the amount of memory access from a prefetch

instruction by analyzing the code pattern. This analyzing process considers each hierarchy of the loop and assumes that the prefetch status is timely. Each loop hierarchy considers different patterns of the amount of memory used by prefetch instruction and the number of times to execute prefetch instruction. Let d_A, d_B, and n_u stand for the prefetch distance of matrix $A(d_A)$, the prefetch distance of matrix B, and the number of loop-unrolling respectively. T_k and T_i are the tile's column and row size respectively. M_{kk} is the memory access of loop *kk*. M_{ii} is the memory access of the loop *ii*. tM_{kk} is total memory accesses of the loop *kk*. tM_{ii} is total memory access of the loop *ii*. tM_k is total memory access in the loop *k*. tM_i is total memory access in the loop *i*. L_s is a cache line size.

Table I. THE AMOUNT OF DATA FROM PREFETCHING VERSUS THE NUMBER OF PREFETCH INSTRUCTIONS

Pattern	Number of prefetch instructions (*NPI*)	Amount of data from prefetching (*ADP*)	*ADP* per *NPI*
1) a	$N^3/(E*T_k)$	$D*N^3/(E*T_k)$	D
2) b	$N^3/(E*T_k)$	$D*N^3/(E*T_k)$	D
3) c	N^3/E	$D*N^3/E$	D
4) d	$N^3/(E*T_k)$	$D*N^3/(E*T_k)$	D
5) a+b	$2N^3/(E*T_k)$	$D*2N^3/(E*T_k)$	D
6) a+c	$N^3/(E*T_k)+N^3/E$	$D*(N^3/(E*T_k)+N^3/E)$	D
7) a+d	$2N^3/(E*T_k)$	$D*2N^3/(E*T_k)$	D
8) b+c	$N^3/(E*T_k)+N^3/E$	$D*(N^3/(E*T_k)+N^3/E)$	D
9) b+d	$2N^3/(E*T_k)$	$D*2N^3/(E*T_k)$	D
10) c+d	$N^3/(E*T_k)+N^3/E$	$D*(N^3/(E*T_k)+N^3/E)$	D
11) a+b+c	$2N^3/(E*T_k)+N^3/E$	$D*(2N^3/(E*T_k)+N^3/E)$	D
12) a+b+d	$3N^3/(E*T_k)$	$D*3N^3/(E*T_k)$	D
13) a+c+d	$2N^3/(E*T_k)+N^3/E$	$D*(2N^3/(E*T_k)+N^3/E)$	D
14) b+c+d	$2N^3/(E*T_k)+N^3/E$	$D*(2N^3/(E*T_k)+N^3/E)$	D
15) a+b+c +d	$3N^3/(E*T_k)+N^3/E$	$D*(2N^3/(E*T_k)+N^3/E)$	$D*(2+E)$ $/(3+E)$

Table II. THE AMOUNT OF MEMERY ACCESS BY PREFETCHING PATTERN 13

loop	Type 1		Type 2		Type 3	
	Memory access (Byte)	Number of loops	Memory access (Bytes)	Number of loops	Memory access (Byte)	Number of loops
kk	M_{kk}	d_B	M_{kk}-D	T_k-d_B	-	-
ii	tM_{kk}+M_{ii}	d_C+1	tM_{kk}+ M_{ii}-(uF/L_s) *E *D*T_k	d_A-(d_C+1)	tM_{kk}+ M_{ii}-(uF/L_s) *E*D* T_k-D	T_i-d_A-(d_C+1)
k	tM_{ii}	n/T_k	-	-	-	-
i	tM_k	n/T_i	-	-	-	-
j	tM_i	$n/(E*uF$)	-	-	-	-
Total	(((((((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii})*(d_C+1))+((((M_{kk}*d_B)+((M_{kk}-D)*(T_k-d_B))+M_{ii}-(uF/L_s)*E *D*T_k)*(d_A-(d_C+1)))+((((M_{kk}*d_B)+((M_{kk}-D)*(T_k-d_B))+M_{ii}-(uF/L_s)*E *D*T_k-D)*(T_i-d_A-(d_C+1))))*n/T_K)*n/T_i)* $n/(E*uF)$					

Table 2 shows how prefetching pattern 13 accesses the data. There are 5 levels of nested loops. The innermost loop uses *kk* as a loop counter, which comprises of 2 types of data access. The first is the data access for the row of Matrix *B* before prefetching. The second is the prefetched data. The next level uses *ii* as a loop counter. It comprises of 3 types of data access. The first is the prefetched data from rows of matrix C. The second is the data before prefetching from rows of Matrix A. The third type is the data accessed from rows of matrix C and A.

Table III. THE AMOUNT OF MEMORY ACCESS OF PREFETCHING PATTERN 14

loop	Type 1		Type 2	
	Memory access (Byte)	Number of loops	Memory access (Byte)	Number of loops
kk	M_{kk}	d_B	M_{kk}-D	T_k-d_B
ii	tM_{kk}+M_{ii}	d_C+1	tM_{kk}+M_{ii}-(uF/L_s)*E *D*T_k	T_i-(d_C+1)
k	tM_{ii}	d_A	tM_{ii}-D	(n/T_k)-d_A
i	tM_k	n/T_i	-	-
j	tM_i	$n/(E*uF)$	-	-
Total	(((((((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii})*(d_C+1))+((((M_{kk}*d_B)+((M_{kk}-D)*(T_k-d_B))+M_{ii}-(uF/L_s)*E *D* T_k)*(T_i-(d_C+1)))*d_A)+ (((((((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii})*(d_C+1))+((((M_{kk}*d_B)+(M_{kk}-D)*(T_k-d_B))+M_{ii}-(uF/L_s)*E *D* T_k)*(T_i-(d_C+1))-D)*((n/T_k)-d_A)))*n/T_i)*$n/(E*uF)$			

IV. EXPERIMENTAL RESULTS

We implemented the matrix-matrix multiplication on the Core i5-6200U machine with 16 GB main memory. Firstly, fifteen prefetching patterns were inserted in the matrix-matrix multiplication algorithm, as shown in Figure 1, and the execution time of each configuration was evaluated. These 15 prefetching patterns were compiled on Microsoft Visual Studio 2017. Initially, we tested all prefetching patterns with three different matrix sizes: 1024, 4096, and 7168. We found that prefetching pattern 13 gave the best execution time for all matrix sizes, as shown in Figure 4. We then wanted to know why it gave better execution time than the others. The *NPI* and *ADP* values of each prefetching pattern were calculated, as shown in Table 1. We found that prefetching patterns 13, 14, and 15 had the same *ADP* values. However, the *NPI* value of pattern 15 was more than those of both patterns 13 and 14. So we calculated the amount of memory access of patterns 13 and 14 utilizing the equations in Tables 2 and 3. We found that total amount of memory access of prefetching pattern 13 was less than that of pattern 14, as shown in Table 4. This gave us the reason why pattern 13 was better than pattern 14, as shown in Figure 4.

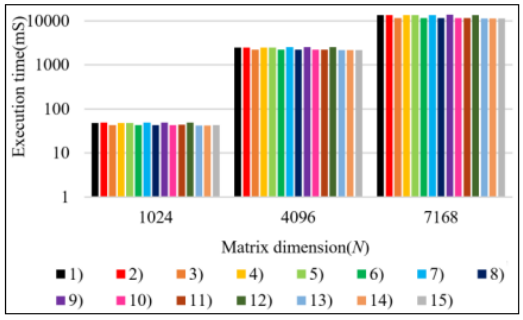


Figure 4. Execution time comparison of each prefetching pattern.

We then compared the execution time of matrix-matrix multiplication with and without prefetching. We compiled these two configurations utilizing Microsoft Visual Studio 2017. In the first configuration, prefetching pattern 13 was

inserted in the matrix-matrix multiplication. In the second configuration, all the prefetching intrinsic calls were deleted from the C code. We found that when prefetching was removed from the C code, MS Visual Studio did not add any prefetching machine instruction to the object code.

We further compiled the code without manual prefetching utilizing Intel C compiler (version 14.16.27012). We found that the object code was automatically inserted with the prefetching machine instructions. This auto-prefetching feature is only available in the Intel C compiler, but not available in Microsoft Visual Studio.

Table IV. COMPARISON OF MEMORY ACCESS OF PREFETCHING PATTERNS 13 AND 14

Matrix Size ($N \times N$)	Total amount of memory access (GB)	
	Pattern 13	Pattern 14
1024×1024	6.14	6.17
2048×2048	47.42	47.54
3072×3072	165.82	166.22
4096×4096	393.06	393.99
5120×5120	767.70	769.52
6144×6144	1326.59	1329.76
7168×7168	2106.57	2111.55
Average	687.614	689.25

Figure 5 shows that auto-prefetching of the Intel compiler gives better execution time than no-prefetching. However, manual prefetching using pattern 13 gives better result than auto-prefetching.

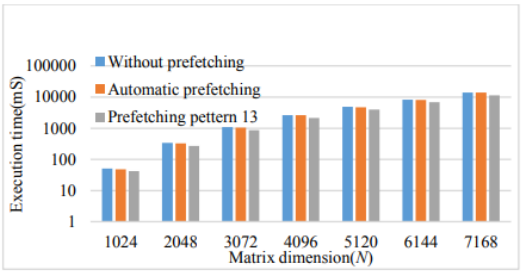


Figure 5. Execution time of 3 configurations over different matrix sizes.

V. DISCUSSION

Our code analysis model consists of two steps. The first step calculates the amount of data from prefetching versus the number of prefetch instructions. The second step calculates the amount of memory access. The first step is quick and easy but is not able to differentiate some prefetching patterns. For example, the result of the first step shows that the prefetching patterns 13 and 14 give the same result. After the second step is applied, the best prefetching pattern is found.

For matrix-matrix multiplication, the best prefetching pattern 13 is the best because variable access of rows of matrix A are more than that of the column. For this reason, prefetching a row of matrix A has a greater effect than prefetching a column of the same matrix. From the experiment, it can be concluded that prefetching the row of matrix A, rows of matrix B, and the rows of matrix C is the best prefetch pattern in this matrix multiplication algorithm.

Our proposed prefetching pattern 13 is on average 18.86 percent faster than without prefetching, and is on average 17.54 percent faster than the automatic prefetching feature provided by the Intel C compiler.

VI. CONCLUSIONS

Prefetching is an important technique to hide memory latencies, especially when processing with large amount data. To obtain the best result, suitable prefetching pattern must be determined. In this paper, we have proposed an analysis method for selecting the best prefetching pattern. By calculating the number of prefetch instructions and the amount of data from prefetching, numbers of prefetching candidates are selected. Then the best prefetching pattern is obtained by selecting the candidate with lowest value of memory access. The prefetching pattern obtained from the proposed analysis method is faster than without prefetching. In addition, the obtained pattern is also faster than the automatic prefetching feature available in the Intel C compiler.

REFERENCES

[1] D. Choi, K. Kim, and E.y. Chung, "Asymmetric Prefetching Architecture for Multicore Processor," International SoC Design Conference (ISOCC), 21-24 Oct. 2020.

[2] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *Trans. Archit. Code Optim.*, vol. 9, no. 1, 2012, doi: 10.1145/2133382.2133384.

[3] S. A. Hassan, M. M. M. Mahmoud, A. M. Hemeida, and M. A. Saber, "Effective Implementation of Matrix–Vector Multiplication on Intel's AVX multicore Processor," *Comput. Lang. Syst. Struct.*, vol. 51, pp. 1339–1351, 2018, doi: 10.1016/j.cl.2017.06.003.

[4] J. Lu, H. Chen, P. C. Yew, and W. C. Hsu, "Design and implementation of a lightweight dynamic optimization system," *J. Instr. Parallelism*, vol. 6, pp. 1–24, 2004.

[5] W. Zhang, B. Calder, and D. M. Tullsen, "A self-repairing prefetcher in an event-driven dynamic optimization framework," *Proc. CGO 2006 - 4th Int. Symp. Code Gener. Optim.*, no. Cgo, pp. 12–64, 2006, doi: 10.1109/CGO.2006.4.

[6] N. Eiron, M. Rodeh, and T. Steinwarts, "Matrix Multiplication: A Case Study of Enhanced Data Cache Utilization," *ACM J. Exp. Algorithmics*, vol. 4, no. 212, p. 3, 1999, doi: 10.1145/347792.347806.

[7] T. F. Chen and J. L. Baer, "Performance study of software and hardware data prefetching schemes," *Conf. Proc. - Annu. Int. Symp. Comput. Archit. ISCA*, pp. 223–232, 1994, doi: 10.1145/192007.192030.

[8] N. Anchev, M. Gusev, S. Ristov, and B. Atanasovski, "Some optimization techniques of the matrix multiplication algorithm," *Proc. Int. Conf. Inf. Technol. Interfaces, ITI*, pp. 71–76, 2013, doi: 10.2498/iti.2013.0572.

[9] H. Amiri and A. Shahbahrami, "High performance implementation of 2D convolution using Intel's advanced vector extensions," Artificial Intelligence and Signal Processing Conference (AISP), 25-27 Oct. 2017.

ประวัติผู้เขียน

ชื่อ สกุล นาย วรินทร ข้อมงคลอุดม

รหัสประจำตัวนักศึกษา 6310120050

วุฒิการศึกษา

วุฒิ	ชื่อสถาบัน	ปีที่สำเร็จการศึกษา
วิศวกรรมศาสตรบัณฑิต (วิศวกรรมคอมพิวเตอร์)	มหาวิทยาลัยสงขลานครินทร์	2562

ตำแหน่งและสถานที่ทำงาน

กรรมการบริษัท อิมพอร์ตสตาร์ จำกัด อำเภอหาดใหญ่ จังหวัดสงขลา

การตีพิมพ์เผยแพร่ผลงาน

- V. Khomongkonudom and P. Chaikarn, "Optimum Prefetching Patterns Searching: A Case Study of Matrix-Matrix Multiplication," *2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, 2022, pp. 349-352, doi: 10.1109/ITC-CSCC55581.2022.9894990.