

เวอร์ชวลคู่มือ : ระบบให้บริการเข้าถึงทรัพยากรดิจิทัลสำหรับ  
เวอร์ชวลแมชีน

โดย

นางสาวสุนทรี บุญมี

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร

วิทยาศาสตรมหาบัณฑิต

สาขาวิชาวิทยาการคอมพิวเตอร์ ภาควิชาวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยธรรมศาสตร์

พ.ศ. 2553

Virtual CUDA: A System Providing Access to GPU Resources for  
Virtual Machines

By

Miss Suntharee Boonmee

A Thesis Submitted in Partial Fulfillment of the Requirements  
for the Degree of Master of Science in Computer Sciences

Department of Computer Science

Faculty of Science

Thammasat University

2010

มหาวิทยาลัยธรรมศาสตร์  
คณะวิทยาศาสตร์และเทคโนโลยี

วิทยานิพนธ์

ของ

นางสาวสุนทรี บุญมี

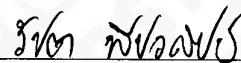
เรื่อง

เวอร์ชวลคู่มือ : ระบบให้บริการเข้าถึงทรัพยากรดิจิทัลสำหรับเวอร์ชวลแมชีน

ได้รับการตรวจสอบและอนุมัติ ให้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร  
วิทยาศาสตรมหาบัณฑิต

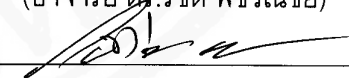
เมื่อ วันที่ 20 เมษายน พ.ศ. 2554

ประธานกรรมการสอบวิทยานิพนธ์



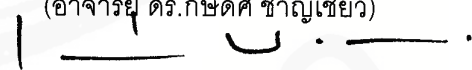
(อาจารย์ ดร.รัชต พิชณิษฐ์)

กรรมการและอาจารย์ที่ปรึกษาวิทยานิพนธ์



(อาจารย์ ดร.กษิตศ ชาญเขียว)

กรรมการสอบวิทยานิพนธ์



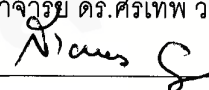
(อาจารย์ ดร.เด่นดวง ประดับสุวรรณ)

กรรมการสอบวิทยานิพนธ์



(อาจารย์ ดร.ศรเทพ วรณรัตน์)

คณบดี



(รองศาสตราจารย์ สายทอง อมริเชษฐ์)

## บทคัดย่อ

เทคโนโลยีทางด้านสถาปัตยกรรมกราฟิกหรือจีพียู (GPU: Graphics Processing Unit) และเทคโนโลยีเวอร์ชวลแมชีน (Virtual Machine หรือ VM) ทั้งสองเทคโนโลยีนี้ ถือได้ว่าเป็นเทคโนโลยีที่มีประสิทธิภาพในการประมวลผลสูง และกำลังเป็นที่นิยมในปัจจุบันทำให้มีการพัฒนาประสิทธิภาพของทั้งสองเทคโนโลยีนี้เพิ่มขึ้นอย่างต่อเนื่อง แต่ด้วยข้อจำกัดของเทคโนโลยีเวอร์ชวลแมชีนที่มีอยู่ในขณะนี้ยังไม่สนับสนุนการประมวลผลแอปพลิเคชันสมรรถนะสูงที่ต้องการใช้งาน หรือเรียกใช้งานจีพียูโดยใช้คิวดำเอพีไอ ดังนั้นวิทยานิพนธ์นี้จึงได้พัฒนาระบบที่เรียกว่า เวอร์ชวลคิวดำ (Virtual CUDA) ซึ่งมีวัตถุประสงค์เพื่อออกแบบให้เวอร์ชวลแมชีนสามารถใช้งาน และเข้าถึงทรัพยากรจีพียูได้ รวมไปถึงการให้บริการและวิธีการจัดสรรทรัพยากรจีพียูให้เพียงพอต่อการใช้งานสำหรับการใช้ทรัพยากรจีพียูร่วมกันของหลายเวอร์ชวลแมชีน กรณีที่ต้องการเข้าถึงเพื่อใช้งานจีพียูพร้อมกัน ได้อย่างมีประสิทธิภาพ ซึ่งจากการทดลองแสดงให้เห็นว่าระบบเวอร์ชวลคิวดำสามารถทำให้เวอร์ชวลแมชีนเข้าถึงและใช้งานจีพียูได้ รวมไปถึงยังสามารถบริหารจัดการ การใช้ทรัพยากรจีพียูร่วมกันของสองเวอร์ชวลแมชีน กรณีเรียกใช้งานจีพียูพร้อมกันได้ นอกจากนี้การใช้เวอร์ชวลคิวดำเพื่อใช้งานจีพียูยังเป็นการเพิ่มประสิทธิภาพในการประมวลผลสำหรับแอปพลิเคชันบนเวอร์ชวลแมชีนที่มีความต้องการในการประมวลผลสูง เช่น การคูณเมตริกซ์ อย่างเห็นได้ชัด ถึงแม้ว่าระบบเวอร์ชวลคิวดำจะมีโอเวอร์เฮดที่เกิดจากการถ่ายโอนข้อมูลระหว่างเวอร์ชวลแมชีนกับโฮสคอมพิวเตอร์ ระบบเวอร์ชวลคิวดำก็สามารถแสดงประสิทธิภาพที่เหนือกว่าการใช้งานเวอร์ชวลแมชีนเพียงอย่างเดียวมากซึ่งเป็นการแสดงให้เห็นศักยภาพของระบบเวอร์ชวลคิวดำในการใช้งานจริง

## Abstract

GPUs and virtual machines have recently becoming popular computing platforms. While GPUs provide enormous computing power, virtual machines provide flexible resource management and utilization for organizations. Despite their advantages, there are only a handful research works that provide accesses to GPU for application programs running on virtual machines. In this research, we present the design and implementation of VirtualCUDA, a library and runtime system that allows accesses to GPU from virtual machines for CUDA applications. The main objectives of are to design and implement 1) a user-level library to pass CUDA command from virtual machines to GPU, and 2) the backend system to handle GPU resources. We have conducted a number of experiments to test our prototypes with the CUDA SDK matrix multiplication program. The first set of experiment evaluates the speed up of the application programs running VirtualCUDA against the serial programs running on virtual machines. We found that the Virtual CUDA program made substantial improvement over the serial one. In the next experiment, we demonstrate that the backend can handle multiple tasks at once; therefore increase resource utilization of the GPU. We have analyzed the experimental results and believe that VirtualCUDA has true practical values.

## กิตติกรรมประกาศ

วิทยานิพนธ์เรื่องเวอร์ชวลคู่มือ : ระบบให้บริการเข้าถึงทรัพยากรจีพียูสำหรับเวอร์ชวลแมชีนฉบับนี้ สามารถสำเร็จลุล่วงไปได้ด้วยดีจากอาจารย์ที่ปรึกษาวิทยานิพนธ์ ดร.กษิตศ ชาญเขียว ที่ให้การสนับสนุนให้คำแนะนำคำปรึกษาทั้งทางด้านวิชาการ และการทำงาน ตรวจสอบแก้ไขข้อบกพร่องต่าง ๆ เพื่อให้วิทยานิพนธ์ฉบับนี้ มีความสมบูรณ์ทุกขั้นตอนตลอดเวลาในการดำเนินการจัดทำ ผู้วิจัยรู้สึกซาบซึ้งในความเมตตากรุณา และขอกราบขอบพระคุณเป็นอย่างสูงไว้ ณ ที่นี้

ขอกราบขอบพระคุณท่านคณะกรรมการสอบวิทยานิพนธ์ที่กรุณาเสียสละเวลา และให้ข้อเสนอแนะในการปรับปรุง แก้ไขวิทยานิพนธ์ให้มีความสมบูรณ์ยิ่งขึ้น

ขอกราบขอบพระคุณคณาจารย์ทุกท่านที่มี ส่วนในการประสิทธิประสาทวิชาให้ แก่ ข้าพเจ้า ขอขอบพระคุณมหาวิทยาลัยธรรมศาสตร์ที่ให้โอกาสข้าพเจ้าได้เข้ามาศึกษา

ท้ายสุดนี้ผู้วิจัยขอกราบขอบพระคุณบิดา มารดา และสมาชิกในครอบครัวทุกคนที่ให้การสนับสนุนเป็นกำลังใจสำคัญ และเป็นแรงผลักดันที่ดีเสมอมาจนสำเร็จการศึกษา

นางสาวสุนทรี บุญมี

มหาวิทยาลัยธรรมศาสตร์

พ.ศ. 2553

## สารบัญ

	หน้า
บทคัดย่อ .....	(2)
กิตติกรรมประกาศ.....	(4)
สารบัญตาราง.....	(7)
สารบัญภาพประกอบ .....	(8)
บทที่	
1. บทนำ .....	1
1.1 ความเป็นมาและความสำคัญของงานวิจัย .....	1
1.2 วัตถุประสงค์ของงานวิจัย .....	3
1.3 ขอบเขตของงานวิจัย.....	3
1.4 ประโยชน์ที่คาดว่าจะได้รับ .....	4
2. งานวิจัยและทฤษฎีที่เกี่ยวข้อง .....	5
2.1 ทฤษฎีที่เกี่ยวข้อง.....	5
2.1.1 Graphic Processing Unit (GPU) .....	5
2.1.2 Compute Unified Device Architecture (CUDA).....	8
2.1.3 เวอร์ชวลแมชีน (Virtual Machine) .....	11
2.1.4 อัลกอริทึมของนายธนาคาร (Banker's Algorithm) .....	15
2.2 งานวิจัยที่เกี่ยวข้อง .....	21
3. วิธีการดำเนินงานวิจัย .....	25

3.1 การออกแบบระบบ .....	26
3.1.1 การทำงานของผู้ให้บริการทรัพยากรจีพียู (Server) และผู้ขอใช้บริการ ทรัพยากรจีพียู (Client) .....	27
3.1.2 การจัดสรรการเข้าถึงทรัพยากรจีพียู .....	30
3.2 วิธีการทดลอง .....	39
3.2.1 เครื่องมือที่ใช้ในการทดลอง .....	39
3.2.2 การออกแบบการทดลอง .....	40
3.2.3 การวิเคราะห์ข้อมูลและการวัดผล .....	43
4. ผลการทดลอง .....	44
4.1 ผลการทดลอง .....	46
4.2 อภิปรายผลการวิจัย .....	61
5. สรุปผลงานวิจัยและข้อเสนอแนะ .....	63
5.1 สรุปผลการวิจัย .....	63
5.2 ข้อเสนอแนะเพิ่มเติม .....	66
รายการอ้างอิง .....	67
ภาคผนวก	
ก. ตัวอย่างโปรแกรม .....	70
ประวัติการศึกษา .....	78



## สารบัญตาราง

ตารางที่		หน้า
3.1	ตัวอย่างตารางการจัดสรรทรัพยากรตามคำสั่งคู่ด้าแอปพลิเคชัน.....	33
4.1	ขนาดของตัวแปรสำหรับใช้ในการประมวลผลการคูณเมตริกซ์.....	45
4.2	จำนวนบล็อกและเทร็ด/บล็อกสำหรับใช้ในการประมวลผลการคูณเมตริกซ์...	45
4.3	ผลการทดลองเปรียบเทียบประสิทธิภาพการทำงานระหว่าง การประมวลผลผ่านซีพียูของเวอร์ชวลแมชีน กับการประมวลผลผ่านเวอร์ชวลคู่ด้า.....	46
4.4	ผลการทดลองเปรียบเทียบการทำงานระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ ตั้งแต่ Initialization ถึง Function Call ด้วยการคูณเมตริกซ์.....	49
4.5	ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูล แบบ Host to Device (จากซีพียูไปซีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ .....	51
4.6	ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูล แบบ Device to Host (จากจีพียูไปซีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ .....	53
4.7	ผลการทดลองเปรียบเทียบการทำงานเข้าถึงจีพียูสำหรับเวอร์ชวลแมชีน แบบครั้งละ 1 เวอร์ชวลแมชีน และพร้อมกัน 2 เวอร์ชวลแมชีน.....	57
4.8	ผลการทดลองเปรียบเทียบการทำงาน ระหว่างการใช้งานจีพียูที่ 1 เวอร์ชวลแมชีน กับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชีน.....	59

## สารบัญภาพประกอบ

ภาพที่		หน้า
1.1	ความแตกต่างของประสิทธิภาพการประมวลผลระหว่าง GPU และ CPU .....	1
2.1	โครงสร้างการทำงานแบบไปป์ไลน์ของจีพียู .....	7
2.2	โครงสร้างการทำงานของคูด้า .....	8
2.3	Compiling CUDA.....	10
2.4	โครงสร้างการแบ่งบล็อกและเทร็ดในคูด้า .....	11
2.5	การแปลง ISA ของซอฟต์แวร์เวอร์ซิลไดซ์ในซีเอสเอ็มเอสเวอร์ซิลแมชชีน.....	13
2.6	ตัวอย่างข้อมูลของระบบที่ใช้งาน.....	20
2.7	ตัวอย่างค่าของ Need := Max – Allocation ของแต่ละโพรเซส .....	20
2.8	ตัวอย่างระบบจำลองที่สร้างเพื่อตรวจสอบภาวะของระบบ.....	21
2.9	ภาพรวมของระบบ GViM และการจัดการพื้นที่หน่วยความจำ.....	22
2.10	แสดงโครงสร้างของระบบวีเอ็มจีแอล (VMGL) .....	23
3.1	Compute Unified Device Architecture Software Stack .....	25
3.2	สถาปัตยกรรมภาพรวมของระบบเวอร์ซิลคูด้า.....	26
3.3	การทำงานระหว่างเครื่องผู้ขอใช้บริการจีพียูกับเครื่องที่ให้บริการจีพียู.....	27
3.4	ภาพรวมการจัดสรรการเข้าถึงทรัพยากรจีพียูของระบบเวอร์ซิลคูด้า สำหรับขั้นตอนการตรวจสอบแต่ละประเภทคำสั่ง.....	35
3.5	ภาพรวมการจัดสรรการเข้าถึงทรัพยากรจีพียูของระบบเวอร์ซิลคูด้า สำหรับขั้นตอนการตรวจสอบพื้นที่บนจีพียูเพื่อประมวลผล .....	35
3.6	ภาพรวมการจัดสรรการเข้าถึงทรัพยากรจีพียูของระบบเวอร์ซิลคูด้า สำหรับขั้นตอนการตรวจสอบคำสั่งปฏิบัติงานในตารางจัดสรร .....	36
3.7	ภาพตรวจสอบคำสั่งการปฏิบัติงาน cuLaunchGrid กรณีเรียกใช้งานซ้ำ ของระบบเวอร์ซิลคูด้า .....	36
3.6	การทำงานของโฮสแมชชีนและสองเวอร์ซิลแมชชีน.....	39
4.1	ผลการทดลองเปรียบเทียบประสิทธิภาพการทำงานระหว่าง การประมวลผลผ่านจีพียูของเวอร์ซิลแมชชีน กับการประมวลผลผ่านเวอร์ซิลคูด้า .....	47

4.2	การวัดผลเรียกใช้งานจีพียูของแบ็คเอนด์ และฟรอนท์เอนด์ผ่านเวอร์ชวลคูด้า	48
4.3	ผลการทดลองเปรียบเทียบการทำงานระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ ตั้งแต่ Initialization ถึง Function Call ด้วยการคูณเมตริกซ์.....	50
4.4	ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูล แบบ Host to Device (จากจีพียูไปจีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ .....	52
4.5	ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูล แบบ Device to Host (จากจีพียูไปซีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ .....	54
4.6	ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลของแบ็คเอนด์ ระหว่าง Device to Host (จากจีพียูไปซีพียู) กับ Host to Device (จากซีพียูไปจีพียู) .....	55
4.7	ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลของฟรอนท์เอนด์ ระหว่าง Device to Host (จากจีพียูไปซีพียู) กับ Host to Device (จากซีพียูไปจีพียู) .....	56
4.8	ผลการทดลองเปรียบเทียบการทำงานเข้าถึงจีพียูสำหรับเวอร์ชวลแมชชีน แบบครั้งละ 1 เวอร์ชวลแมชชีน และพร้อมกัน 2 เวอร์ชวลแมชชีน.....	58
4.9	ผลการทดลองเปรียบเทียบการทำงานระหว่างการใช้งานจีพียูที่ 1 เวอร์ชวลแมชชีน กับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชชีน.....	60

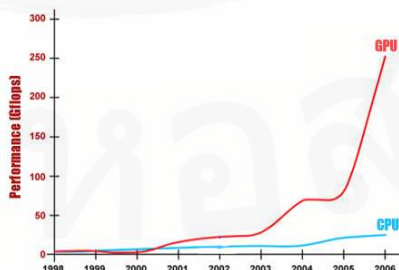
## บทที่ 1

### บทนำ

#### 1.1 ความเป็นมาและความสำคัญของงานวิจัย

ในยุคปัจจุบันเทคโนโลยีทางด้านสถาปัตยกรรมกราฟิกหรือจีพียู (GPU: Graphics Processing Unit) ได้มีความก้าวหน้าเป็นอย่างมาก อีกทั้งมีการพัฒนาความสามารถในด้านการประมวลผลอย่างต่อเนื่องไม่หยุดนิ่ง จึงทำให้ในระยะเวลาไม่กี่ปีที่ผ่านมาประสิทธิภาพในการประมวลผลของจีพียูเพิ่มขึ้นมากอย่างต่อเนื่องเมื่อเปรียบเทียบกับซีพียูดังแสดงในภาพที่ 1.1 ซึ่งแสดงให้เห็นถึงประสิทธิภาพของจีพียู (เส้นบน) ที่เพิ่มขึ้นอย่างมากเมื่อเปรียบเทียบกับซีพียู (เส้นล่าง) ในการประมวลผลแบบ GFLOPS โดยที่แท้จริงแล้วหน้าที่หลักของจีพียูนั้นเริ่มต้นจากการเป็นตัวช่วยในการประมวลผลการทำงานในด้านภาพกราฟิกบนหน้าจอคอมพิวเตอร์ให้มีประสิทธิภาพสูงสุด ซึ่งได้แก่งานประเภทให้ความบันเทิง เช่น เกมส์ หรือวิดีโอ และด้วยโครงสร้างของจีพียูที่มีลักษณะเป็นแบบหลายแกน (Many Core) ที่ทำให้จีพียูมีประสิทธิภาพในการประมวลผลสูง จึงทำให้ในปัจจุบันได้มีการนำความสามารถในการประมวลผลทางด้านการคำนวณของจีพียู มาใช้กับงานที่ไม่ใช่ประเภทการประมวลผลทางด้านกราฟิกเพียงอย่างเดียวแต่นำจีพียูมาประมวลผลแทน หรือช่วยซีพียูในการทำงานที่ซับซ้อนหลาย ๆ อย่างสำหรับงานที่ต้องการความสามารถในการคำนวณสูง (High Performance Computing) เช่น งานในด้านการคำนวณ และงานทางด้านวิทยาศาสตร์ เป็นต้น โดยใช้ชุดคำสั่ง CUDA API และเมื่อเทียบกับซีพียูแล้วการใช้งานจีพียูสามารถเพิ่มประสิทธิภาพของแอปพลิเคชันได้กว่าสิบหรือร้อยเท่า (M. Garland et al, M. Boyer et al, B. Burke, A. Humber)

ภาพที่ 1.1 ความแตกต่างของประสิทธิภาพการประมวลผลระหว่าง GPU และ CPU



ที่มา : “Different cores for different chores” โดย Tony Smith, 2006, จาก

[http://www.reghardware.com/2006/10/26/the\\_story\\_of\\_amds\\_fusion/](http://www.reghardware.com/2006/10/26/the_story_of_amds_fusion/)

เทคโนโลยีเวอร์ชวลแมชีน (Virtual Machine หรือ VM) เป็นเทคโนโลยีการจำลองเครื่องคอมพิวเตอร์โดยซอฟต์แวร์บนเครื่องคอมพิวเตอร์จริง ทำให้สามารถสร้างเวอร์ชวลแมชีนหรือคอมพิวเตอร์เสมือนหลาย ๆ เครื่องให้ทำงานขนานกันไปบนคอมพิวเตอร์จริงเพียงเครื่องเดียว เครื่องคอมพิวเตอร์เสมือนแต่ละเครื่องจะรันระบบปฏิบัติการของตัวเอง เรียกว่าระบบปฏิบัติการเกส (Guest Operating System) การใช้เวอร์ชวลแมชีนนับได้รับความนิยมมากในปัจจุบัน เพราะทำให้เกิดการเพิ่มการใช้ประโยชน์ (Utilization) ของเครื่องคอมพิวเตอร์จริงมากขึ้น

ระบบเวอร์ชวลแมชีนที่มีอยู่ในขณะนี้ไม่สนับสนุนการคำนวณสมรรถนะสูง ในสภาพแวดล้อมจำลองของเวอร์ชวลแมชีนนับเป็นเรื่องยากที่จะรันแอปพลิเคชันสมรรถนะสูงที่ต้องการพลังงาน หรือเรียกใช้งานจีพียูโดยใช้คูด้าเอพีไอผ่านเวอร์ชวลแมชีนเนื่องจากเกสโอเปอเรติงซิสเต็มในเวอร์ชวลแมชีนไม่สามารถเรียกใช้งานอุปกรณ์จีพียูจากไดรเวอร์ของอุปกรณ์จีพียูที่มีอยู่ในระบบปฏิบัติการโฮส (Host Operating System) บนเครื่องคอมพิวเตอร์จริงได้โดยตรง ยิ่งไปกว่านั้นระบบเวอร์ชวลแมชีนที่มีอยู่ยังไม่สนับสนุนการใช้งานจีพียูร่วมกันจากเวอร์ชวลแมชีนหลายเวอร์ชวลแมชีน (ที่อาจอยู่บนเครื่องคอมพิวเตอร์จริงเครื่องเดียวกันกับอุปกรณ์จีพียูหรืออยู่ต่างเครื่องก็ได้) อีกด้วย

วิทยานิพนธ์ฉบับนี้จึงนำเสนอ**ระบบเวอร์ชวลคูด้า** (Virtual CUDA) สำหรับให้บริการ การเข้าถึงทรัพยากรจีพียูได้จากเวอร์ชวลแมชีนโดยผ่านคูด้าเอพีไอที่จำลองขึ้นมา ซึ่งพัฒนาซอฟต์แวร์ต้นแบบ (Prototype) ของระบบเวอร์ชวลคูด้าประกอบไปด้วย ฟรอนต์เอนด์ไลบรารี (Front-End Library) ซึ่งจำลองคูด้าเอพีไอ และแบ็คเอนด์ซอฟต์แวร์ (Back-End Software) โดยการออกแบบระบบเวอร์ชวลคูด้าขึ้นเพื่อให้มีความสามารถในการรองรับการทำงาน ดังนี้ คือ 1) เพื่อให้คูด้าแอปพลิเคชันบนเวอร์ชวลแมชีนหรือเครื่องคอมพิวเตอร์เครื่องอื่นที่ไม่มีอุปกรณ์จีพียูเป็นของตนเอง แต่เชื่อมต่อกับเครื่องคอมพิวเตอร์ที่มีอุปกรณ์จีพียูผ่านเน็ตเวิร์คให้สามารถเรียกใช้งานทรัพยากรจีพียูโดยใช้คูด้าเอพีไอได้เสมือนมีทรัพยากรจีพียูเป็นของตนเอง 2) เพื่อเพิ่มการใช้ประโยชน์ของจีพียูโดยอนุญาตให้เวอร์ชวลแมชีนมากกว่าหนึ่งเครื่องสามารถเข้าใช้งานจีพียูร่วมกันได้ โดยที่ระบบแบ็คเอนด์ของเวอร์ชวลคูด้าจะจัดสรรการเข้าถึงจีพียูของแต่ละผู้ขอรับบริการ เพื่อไม่ให้เกิดปัญหาการเรียกใช้งานถูกยกเลิก กรณีที่ทรัพยากรจีพียูถูกเรียกใช้จนเต็มหรือการรอใช้ทรัพยากรจีพียูจนเกิดปัญหาการติดตาย (Deadlock) และ 3) เพื่อให้สามารถนำไปใช้งานบนเวอร์ชวลแมชีนได้หลายชนิดไม่ขึ้นอยู่กับชนิดใดชนิดหนึ่ง ซอฟต์แวร์ต้นแบบนี้ได้รับการพัฒนาขึ้นบนเควีเอ็ม (KVM) ซึ่งเป็นเวอร์ชวลแมชีนบนระบบลินุกซ์และสามารถนำไปพัฒนาใช้งานบนเวอร์ชวลแมชีนแบบอื่นได้โดยง่าย



## 1.2 วัตถุประสงค์ของงานวิจัย

ในงานวิทยานิพนธ์ฉบับนี้ มีวัตถุประสงค์ในการค้นคว้าวิจัยดังนี้

1. เพื่อศึกษาและพัฒนาวิธีการเข้าถึงอุปกรณ์จีพียูสำหรับคู้ด้าแอปพลิเคชันที่อยู่ในเวอร์ชวลแมชีน
2. เพื่อศึกษาและพัฒนาวิธีการจัดการสรรกรเข้าถึงทรัพยากรจีพียูจากการเรียกใช้งานร่วมกันของเวอร์ชวลแมชีนและหลีกเลี่ยงการเกิดปัญหาการติดตาย (Deadlock)
3. เพื่อให้ผู้ที่ใช้งานเวอร์ชวลแมชีนสามารถเรียกใช้งานจีพียูจากเวอร์ชวลแมชีนได้
4. เพื่อศึกษาประสิทธิภาพของระบบเวอร์ชวลคู้ด้า (Virtual CUDA)
5. เพื่อศึกษาลักษณะของแอปพลิเคชันที่เหมาะสมสำหรับการใช้งานระบบเวอร์ชวลคู้ด้า

## 1.3 ขอบเขตของงานวิจัย

งานวิทยานิพนธ์ฉบับนี้นำเสนอไดร์เวอร์จำลองสำหรับเครื่องเวอร์ชวลแมชีนที่ต้องการเรียกใช้งานจีพียูจากเครื่องหลักของผู้ให้บริการอุปกรณ์จีพียู โดยมีขอบเขตการวิจัยดังนี้

1. ออกแบบและพัฒนาระบบเวอร์ชวลคู้ด้า สำหรับการเรียกใช้งานอุปกรณ์จีพียูจากเครื่องเวอร์ชวลแมชีน
2. ออกแบบและพัฒนาแคเ็นดท์ที่เครื่องคอมพิวเตอร์จริงของผู้ให้บริการจีพียูเพื่อเข้าถึงจีพียูจากการร้องขอของเวอร์ชวลแมชีน
3. ออกแบบและพัฒนากาารจัดการสรรกรเข้าถึงจีพียูจากการร้องขอของเวอร์ชวลแมชีน กรณีมีการขอใช้จีพียูและเข้าถึงจีพียูพร้อมกัน
4. การออกแบบระบบเวอร์ชวลคู้ด้าสำหรับเรียกใช้งานจีพียูกำหนดให้คู้ด้าแอปพลิเคชัน สามารถเรียกใช้งาน CUDA API Programming Interface ดังต่อไปนี้

- |                       |                              |
|-----------------------|------------------------------|
| ➤ cuInit              | ➤ cuMemAlloc                 |
| ➤ cuDeviceGet         | ➤ cuMemcpyHtoD/ cuMemcpyDtoH |
| ➤ cuCtxCreate         | ➤ cuParamSetv                |
| ➤ cuModuleLoad        | ➤ cuParamSetSize             |
| ➤ cuModuleGetFunction | ➤ cuFuncSetBlockShape        |

➤ cuLaunchGrid

➤ cuMemFree

➤ cuCtxDetach

5. การทดลองของงานวิจัยนี้จะใช้การอิมพลีเมนต์บนเควีเอ็มด้วยระบบไฮสโอเอสที่เป็นระบบปฏิบัติการลินุกซ์ และมีระบบปฏิบัติการในเวอร์ชวลแมชีนเป็นระบบปฏิบัติการลินุกซ์เช่นกัน

#### 1.4 ประโยชน์ที่คาดว่าจะได้รับ

1. ระบบเวอร์ชวลคูด้าทำให้สามารถรันคูด้าแอปพลิเคชันจากเวอร์ชวลแมชีนได้
2. ระบบเวอร์ชวลคูด้าทำให้สามารถเรียกใช้งานจีพียูจากเวอร์ชวลแมชีนได้
3. ระบบเวอร์ชวลคูด้าทำให้เกิดการใช้งานจีพียูร่วมกันจากหลายเวอร์ชวลแมชีนพร้อม ๆ กันได้ โดยไม่เกิดการติดตาย
4. ระบบเวอร์ชวลคูด้าทำให้การใช้ประโยชน์ (Utilization) ของจีพียูเพิ่มขึ้น อันเนื่องมาจากการอนุญาตให้มีการใช้งานจีพียูพร้อม ๆ กันจากหลายเวอร์ชวลแมชีน
5. ระบบเวอร์ชวลคูด้าทำให้สามารถรันคูด้าแอปพลิเคชันจากเวอร์ชวลแมชีนที่รันอยู่บนคอมพิวเตอร์ที่ไม่มีอุปกรณ์จีพียูเป็นของตนเอง สามารถเรียกใช้งานจีพียูจากคอมพิวเตอร์เครื่องอื่นที่มีอุปกรณ์จีพียูได้
6. ความรู้จากการศึกษาประสิทธิภาพของระบบเวอร์ชวลคูด้า และลักษณะของคูด้าแอปพลิเคชันที่เหมาะสมกับระบบเวอร์ชวลคูด้า

## บทที่ 2

### งานวิจัยและทฤษฎีที่เกี่ยวข้อง

ในบทนี้จะกล่าวถึงทฤษฎีและงานวิจัยที่ใช้ในการศึกษาประกอบการทำวิทยานิพนธ์ฉบับนี้ โดยส่วนแรกเป็นการอธิบายทฤษฎีที่เกี่ยวข้องที่ใช้ในงานวิจัย และส่วนต่อไปเป็นการนำเสนองานวิจัยที่เกี่ยวข้องและตัวอย่างงานวิจัยที่ผ่านมา

#### 2.1 ทฤษฎีที่เกี่ยวข้อง

##### 2.1.1 Graphic Processing Unit (GPU)

Graphic Processing Unit (GPU) นั้นสามารถนำไปใช้งานได้หลายด้านแต่ส่วนมากจะใช้ เพื่อให้การประมวลผลที่เร็วขึ้นมากกว่าปกติที่จากเดิมใช้เพียงซีพียูอย่างเดียว สำหรับการปฏิบัติการด้านคอมพิวเตอร์กราฟิก 2D, 3D รวมทั้ง BitBLT โดยทั่วไปแล้วฮาร์ดแวร์พิเศษที่เรียกว่า Bitter และเป็นตัวดำเนินการในการแสดงผลรูปสี่เหลี่ยม สามเหลี่ยม วงกลม และมุม ซึ่งในปัจจุบันจีพียู สามารถรองรับกราฟิก 3D ในระดับสูง ไม่ว่าจะเป็น Hight Definition Video หรือ เกมส์ต่าง ๆ ที่ล้วนแต่ต้องอาศัยประสิทธิภาพของกราฟิกการ์ดจีพียู

รูปแบบการทำงานของจีพียูเป็นการทำงานแบบ SIMD (Single Instruction Multiple Data) คือการประมวลผลด้วยชุดข้อมูลหลายชุด แต่ทำงานด้วยคำสั่งเดียว ซึ่งเป็นการทำงานแบบคู่ขนาน โดยแต่ละขั้นตอนของการดำเนินงาน อินพุตที่ได้จะมาจากเอาต์พุตของขั้นตอนก่อนหน้า อินพุตดังกล่าวจะเป็นเอาต์พุตที่ถูกส่งไปยังขั้นตอนต่อไป ดังนั้นการคำนวณบนจีพียูจึงคำนวณตามการสั่งของโพสเชสซิงสเตท (Processing stage) หรือที่เรียกว่าไปป์ไลน์ (Pipeline) ซึ่งมีการดำเนินการ 3 ขั้นตอนคือ การประมวลผลจุดยอด (Vertex Processing), การยิงแสงสแกนสาดบนหน้าจอ (Rasterization), การประมวลผลแบบแยกออกเป็นชิ้นๆ (Fragment Processing)

##### 2.1.1.1 โครงสร้างการทำงานของจีพียู

การ์ดแสดงผลมีหน้าที่หลักในการรับข้อมูลดิจิทัลมาแปลงเป็นสัญญาณอนาล็อกเพื่อส่งออกไปแสดงผลยังหน้าจอซึ่งสามารถแบ่งการทำงานของการ์ดแสดงผลออกเป็น 2 โหมดคือ โหมดตัวอักษร (Text Mode) และ โหมดกราฟิก (Graphic Mode) การ์ดแสดงผลในปัจจุบันมี



หน้าที่ในการประมวลผลข้อมูลภาพก่อนที่จะส่งไปแสดงผลยังจอมอนิเตอร์ชิพกราฟิกจึงเทียบเท่ากับสมองของการ์ดแสดงผลซึ่งภาพแต่ละเฟรมที่เห็นผ่านจอมอนิเตอร์ต้องผ่านการทำงานของชิพกราฟิกเกือบทั้งหมด โดยทั่วไปสามารถแบ่งชิพกราฟิกได้เป็น 3 ประเภท ดังต่อไปนี้

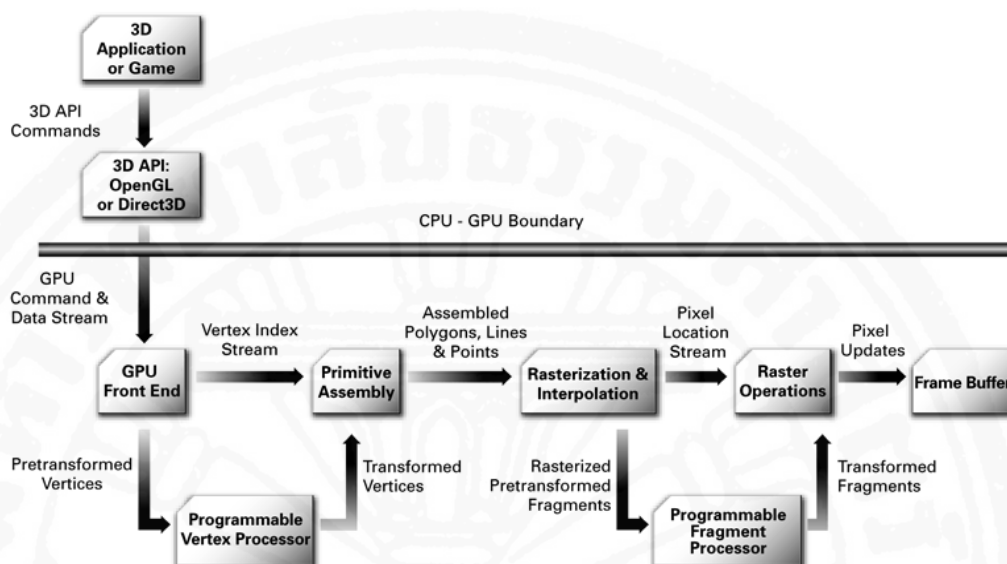
1. เฟรมบัฟเฟอร์ (Frame Buffer) เป็นชิพที่มีการทำงานซับซ้อนน้อยที่สุดเนื่องจากมีหน้าที่เพียงแค่จัดการภาพแต่ละเฟรมที่เก็บไว้ในหน่วยความจำบนการ์ดแล้วส่งข้อมูลไปยังตัวแปลงสัญญาณดิจิทัลให้เป็นอนาล็อก (RAMDAC) เพื่อส่งไปแสดงผลยังหน้าจอมอนิเตอร์ชิพประเภทนี้ไม่ได้มีหน้าที่ช่วยชิพประมวลผลในการสร้างภาพกราฟิกจึงทำให้การประมวลผลด้านกราฟิกอยู่ที่ชิพยูนิตนั้น ส่งผลให้ชิพทำงานมากขึ้น

2. Graphics Accelerator เป็นชิพที่ช่วยเร่งความเร็วให้กับการแสดงผล โดยมีหน้าที่หลัก คือรับคำสั่งจากชิพยูนิตมาทำงานเฉพาะด้าน เช่น การสร้างกรอบ การตีเส้น ซึ่งภายในชิพจะมีชุดคำสั่งเก็บไว้ใช้สำหรับงานที่ต้องการแสดงผลบ่อยจากนั้นชิพยูนิตจะทำหน้าที่ตัดสินใจว่าจะให้ชิพกราฟิกเป็นตัวประมวลผลหรือว่าจะทำการประมวลผลเอง ถึงแม้ว่าชิพตัวนี้จะช่วยลดภาระการทำงานของชิพยูนิตได้ในระดับหนึ่ง แต่ก็มีข้อเสียคือ ชิพยังคงต้องมีการติดต่อกับชิพยูนิตทุกครั้งที่ทำการแสดงผล ประสิทธิภาพความเร็วของชิพกราฟิกประเภทนี้จึงยังไม่สามารถรองรับงานกราฟิกหนักได้ดีเท่ากับชิพประเภท Graphics Co-Processor

3. Graphics Co-Processor หรือที่เรียกว่าจีพียู (GPU: Graphics Processing Unit) เป็นชิพที่มีความสามารถในการจัดการประมวลผลงานทุกอย่างที่เกี่ยวข้องกับการแสดงผล รวมไปถึงการประมวลผลกราฟิก 3 มิติที่ต้องมีการคำนวณเลขทศนิยมที่มีความละเอียดสูง โดยไม่ต้องพึ่งการทำงานของชิพยูนิตทำให้ชิพยูนิตรับภาระด้านการประมวลผลน้อยลง

#### 2.1.1.2 โครงสร้างการทำงานแบบไปป์ไลน์ของจีพียู

ภาพที่ 2.1 โครงสร้างการทำงานแบบไปป์ไลน์ของจีพียู



ที่มา : “User’s Manual A Developer’s Guide to Programmable Graphics  
(www.nvidia.com)” โดย nVidia

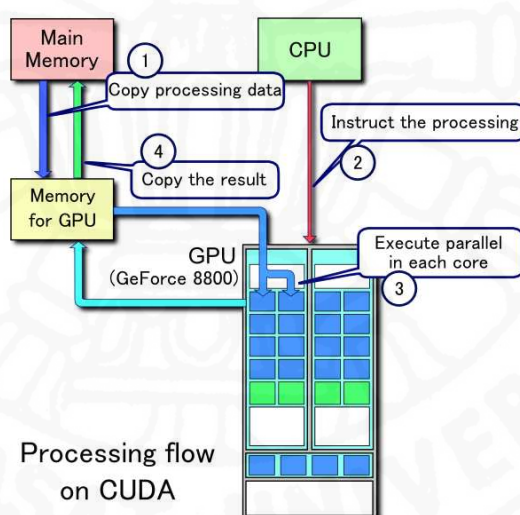
จากภาพที่ 2.1 สามารถอธิบายการทำงานของจีพียูไปป์ไลน์ได้ดังนี้ คือ จีพียูไปป์ไลน์จะทำงานตามลำดับของสเตปไดอะแกรม เริ่มจากส่วนของโปรแกรมเอพีไอ ซึ่งเอพีไอในปัจจุบันจะเป็นพวกโอเพนจีแอล (OpenGL) หรือ ไดเร็กเอ็กซ์ (DirectX) จะทำการส่งข้อมูลมาให้กับหน่วยประมวลผลกราฟิกผ่านทางไดรเวอร์จากจีพียู ที่ส่งผ่านทางบัสเข้ามาที่จีพียูโดยจีพียูฟรอนท์เอนด์จะรับคำสั่งและข้อมูลจากไดรเวอร์ผ่านทาง PCI-Express แล้วจึงเข้าสู่กระบวนการของ Vertex Processing ข้อมูลที่เข้ามา เช่น Position Binormal Tangent Textcoord Color และ Psize นอกจากนี้ในกระบวนการของ Vertex Processing อาจมีการรับข้อมูลจากเท็กซ์เจอร์เข้ามาประมวลผลเซดเดอร์ด้วย เมื่อทำการประมวลผลเสร็จจะได้ตำแหน่งและขนาด 2 มิติ จากนั้นจะส่งต่อไปที่ Primitive Assembly ทำการตรวจสอบจุดและเส้นเหลี่ยม อีกทั้งเชื่อมโยงจุดต่างๆ เข้าด้วยกันจนเป็นรูปสามเหลี่ยมจากนั้นขบวนการ การทำให้เป็นจุดภาพ (Rasterization) จะทำการหาความลึก ความสูงเพื่อที่จะคำนวณภาพ 3 มิติออกมา แล้วจึงเข้าสู่กระบวนการ Fragment Processing ซึ่งสามารถโปรแกรมในส่วนนี้ได้ และอาจมีการรับข้อมูลเท็กซ์เจอร์เข้ามา เพื่อคำนวณเซดเดอร์ทำให้ได้ข้อมูลของสี ความลึก และความสูงของภาพ การตรวจสอบเฟรมบัพเฟอร้นั้น ถ้ามี

ค่าน้อยกว่าแสดงว่าภาพนั้นถูกทับซ้อนอยู่ ด้านหลังขบวนการนี้จะทำการเบลนดิงเพื่อหาสีของเฟรมบัฟเฟอร์นั้น

## 2.1.2 Compute Unified Device Architecture (CUDA)

คูด้าเป็นสถาปัตยกรรมการประมวลผลแบบขนานบนจีพียูที่ถูกพัฒนาขึ้นโดยเอ็นวีเดีย(Nvidia) คูด้ามีความสามารถในการสั่งงานให้จีพียูประมวลผลทั้งในด้านกราฟิกและความสามารถในการคำนวณ โดยมีโครงสร้างการทำงานดังภาพที่ 2.2 และสามารถอธิบายขั้นตอนการทำงานได้ดังนี้

ภาพที่ 2.2 โครงสร้างการทำงานของคูด้า



ที่มา : “en.wikipedia.org/wiki/CUDA” โดย Wikipedia

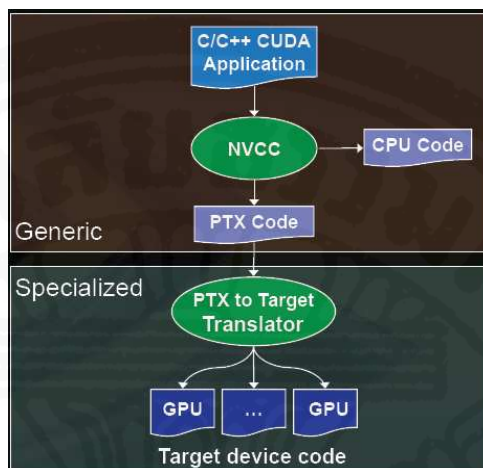
1. เริ่มต้นจากข้อมูลที่อยู่ในหน่วยความจำหลักจะถูกคัดลอกและส่งไปยังหน่วยความจำของจีพียู
2. หลังจากนั้นจีพียูจะส่งคำสั่งสำหรับสั่งงานให้จีพียูประมวลผลไปยังจีพียู
3. เมื่อจีพียูได้รับทั้งข้อมูลและคำสั่งที่ถูกส่งมา จีพียูจะจัดตารางการทำงานและประมวลผลบนจีพียูโดยแบ่งการทำงานไปตามแต่ละแกนหลัก (Core) ที่ยังไม่ได้ถูกเรียกใช้งานซึ่งการประมวลผลการทำงานนี้จะเป็นการประมวลแบบขนาน

4. เมื่อจีพียูประมวลผลจนได้ผลลัพธ์ ผลลัพธ์ดังกล่าวจะถูกคัดลอกกลับไปยังซีพียูเพื่อนำไปใช้งานต่อไป

สถาปัตยกรรมของคุ้ด้าโปรแกรมประกอบด้วย โฮสโปรเซสเซอร์ (Host processor) หน่วยความจำของโฮส (Host memory) และการ์ดจอแสดงผลที่รองรับการทำงานของคุ้ด้าโปรแกรมซึ่งเป็นการ์ดจอของค่ายเอ็นวีเดีย โดยจีพียูที่รองรับการทำงานของคุ้ด้าโปรแกรมจะมีลักษณะการทำงานแบบไปป์ไลน์ โปรเซสเซอร์แรกที่สนับสนุนการทำงานของคุ้ด้าได้แก่ GeForce 8800 ซึ่งเป็นตัวที่ถูกสืบทอดมาจาก GeForce และ Quadro และในปัจจุบันตระกูล Tesla ทั้งหมดก็สนับสนุนสถาปัตยกรรมของคุ้ด้าโปรแกรม ซึ่งคุ้ด้าที่ประมวลผลการทำงานในจีพียูสามารถแบ่งเทรด (Thread) การทำงานเพื่อปฏิบัติงานแบบขนานพร้อมกันได้ครั้งละเป็นพันเทรด และมีลักษณะการทำงานแบบเอสไอเอ็มดี (SIMD) ด้วยหน่วยประมวลผลการคำนวณที่มีเป็นจำนวนมากและฮาร์ดแวร์แบบมัลติเทรดดัง

โปรแกรมภาษาคุ้ด้า นั้นถูกพัฒนาขึ้น ซึ่งถูกต่อขยายมาจากโปรแกรมภาษาซีโดยมีวิธีการทำงานในขั้นตอนการประมวลผลแยกกัน คือกรณีฟังก์ชันการทำงานถูกเรียกใช้สำหรับสั่งงานในส่วนที่เป็นจีพียูตัวแปรและข้อมูลต่าง ๆ จะถูกนำไปประมวลผลบนจีพียูและถ้าฟังก์ชันที่เรียกใช้เป็นคำสั่งสำหรับงานที่อยู่บนซีพียูตัวแปรและข้อมูลต่าง ๆ จะถูกประมวลผลอยู่บนซีพียูเท่านั้น ในการเรียกคุ้ด้าเพื่อใช้งานนั้นจำเป็นต้องเรียกผ่านเอพีไอไลบรารีของคุ้ด้าเพื่อไลบรารีดังกล่าวจะทำหน้าที่เข้าถึงอุปกรณ์จีพียูได้โดยตรง โดยมีขั้นตอนการคอมไพล์ คือเมื่อตัวคุ้ด้าคอมไพเลอร์ เอ็นวีซีซี (nvcc) ทำหน้าที่คอมไพล์ไฟล์ที่เป็นคุ้ด้าไฟล์ เอ็นวีซีซีจะประมวลผลแยกโค้ดของไฟล์ที่นำมาคอมไพล์ออกเป็น 2 ส่วน ดังแสดงในภาพที่ 2.3 คือ โค้ดส่วนที่ประมวลผลบนซีพียูกับโค้ดส่วนที่ประมวลผลบนจีพียูในรูปแบบของไบนารีไฟล์ เรียกว่าคุ้บิน (CUBIN) ไฟล์ สำหรับการเรียกคุ้ด้าเพื่อใช้งานนั้นจำเป็นต้องเรียกผ่านเอพีไอไลบรารีของคุ้ด้า โดยเอพีไอถูกแบ่งเป็น 2 ระดับคือ ระดับต่ำถูกเรียกว่า CUDA driver API และระดับสูงเรียกว่า CUDA runtime API เอพีไอดังกล่าวจะทำหน้าที่ติดต่อสื่อสารและเข้าถึงอุปกรณ์จีพียูได้โดยตรง

ภาพที่ 2.3 Compiling CUDA

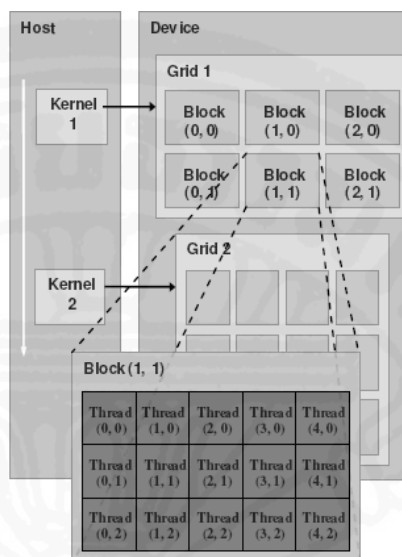


ที่มา : “High Performance Computing with CUDA” โดย Patrick LeGresley,  
2008, iCME Colloquium, น.22

การทำงานของคู่มือสำหรับโปรแกรมเมอร์นั้น คู่มือได้ออกแบบให้มีลักษณะการทำงานแบบขนาน โดยแบ่งการทำงานในส่วนที่สั่งงานจีพียูออกเป็นบล็อกและเทร็ดจากการสั่งงานของโฮส (ซีพียู) ผ่านเคอร์เนล (ฟังก์ชัน) โดยเมื่อสั่งงานไปยังจีพียูในส่วนของจีพียูจะแบ่งการทำงานออกเป็น กริด บล็อก และเทร็ด โดย 1 กริดจะประกอบไปด้วยหลายบล็อกและใน 1 บล็อกจะประกอบไปด้วยหลายเทร็ด ซึ่งกลุ่มของเทร็ดที่อยู่ในบล็อกจะถูกประมวลผลอยู่บนมัลติโพรเซสเซอร์ นอกจากนั้นมัลติบล็อกสามารถถูกสั่งให้ประมวลผลทำงานบนโพรเซสเซอร์เดี่ยวพร้อม ๆ กันได้ในเวลาเดียวกัน ดังแสดงในภาพที่ 2.4



ภาพที่ 2.4 โครงสร้างการแบ่งบล็อกและเทรดในคูด้า



ที่มา : “NVIDIA CUDA Programming Guide Version 2.3” โดย NVIDIA Corporation, developer.nvidia.com, January 7, 2010

### 2.1.3 เวอร์ชวลแมชชีน (Virtual Machine)

ระบบเวอร์ชวลแมชชีนช่วยให้ฮาร์ดแวร์แพลตฟอร์มของโฮสเครื่องหนึ่งสามารถสนับสนุนแกสโอเอสหลายระบบได้ในเวลาเดียวกันด้วยเทคโนโลยีเวอร์ชวลแมชชีน ผู้ใช้สามารถรันระบบปฏิบัติการที่แตกต่างกันได้บนฮาร์ดแวร์เดียวกัน ความสามารถที่สำคัญของเทคโนโลยีซิสเต็มส์เวอร์ชวลแมชชีนคือการแยกออกจากกัน (Isolation) ของระบบต่าง ๆ ที่รันอยู่ในเวลาเดียวกันบนฮาร์ดแวร์แพลตฟอร์มเดียวกัน นั่นคือถ้าแกสโอเอสระบบหนึ่งเกิดความผิดพลาดขึ้นซอฟต์แวร์ที่รันอยู่บนแกสระบบอื่นจะไม่ได้รับผลกระทบไปด้วย โดยหลักในระบบเวอร์ชวลแมชชีน VMM จะแบ่งทรัพยากรฮาร์ดแวร์ระหว่างแกสโอเอสต่าง ๆ เช่น ดิสก์ เวอร์ชวลไลเซชัน โดย VMM จะมีการใช้งานและจัดการทรัพยากรฮาร์ดแวร์ทั้งหมด แกสโอเอสและแอปพลิเคชันโปรเซสจะถูกจัดการภายใต้การควบคุมของ เวอร์ชวลแมชชีนมอนิเตอร์ (VMM : Virtual Machine Monitor) ซึ่งเป็นซอฟต์แวร์เลเยอร์ที่แบ่งฮาร์ดแวร์แพลตฟอร์มออกเป็นเวอร์ชวลแมชชีนหลายเครื่อง เมื่อแกสโอเอสทำคำสั่งพิเศษของระบบหรือทำงานที่ติดต่อกับทรัพยากรฮาร์ดแวร์โดยตรง VMM จะจับการทำงานนั้น ตรวจสอบความถูกต้อง และทำงานนั้นแทนแกสโดยที่ซอฟต์แวร์ของแกสไม่รู้เกี่ยวกับการ

ทำงานนี้ (Smith & Ravi, 2005, p.36) VMs ถูกสร้างขึ้นจากองค์ประกอบที่แตกต่างกัน ดังนี้ (Ruest & Ruest, 2009, pp.30-32)

- คอนฟิกไฟล์ (Configuration File) คือ ไฟล์ที่ประกอบด้วยข้อมูลเกี่ยวกับการตั้งค่าต่าง ๆ สำหรับเวอร์ชวลแมชีน ได้แก่ ขนาด RAM จำนวนโปรเซสเซอร์ จำนวนและประเภทของเน็ตเวิร์คอินเตอร์เฟซการ์ด (NICs) จำนวนและประเภทของเวอร์ชวลดิสก์ โดยแต่ละครั้งที่สร้างเวอร์ชวลแมชีนเครื่องใหม่คอนฟิกไฟล์ของเวอร์ชวลแมชีนเครื่องนั้นจะถูกสร้างขึ้น ซึ่งไฟล์นี้จะบอกเวอร์ชวลไลเซชันซอฟต์แวร์ว่าจะจัดสรรทรัพยากรจริงจากโฮสให้กับเวอร์ชวลแมชีนได้อย่างไร โดยการระบุตำแหน่งที่ฮาร์ดดิสก์ไฟล์อยู่ ขนาด RAM ที่จะใช้วิธีการโต้ตอบกับเน็ตเวิร์คอะแดปเตอร์การ์ดและโปรเซสเซอร์ตัวใดบ้างที่ต้องการใช้งาน

- ฮาร์ดดิสก์ไฟล์ คือไฟล์ที่ประกอบด้วยข้อมูลที่โดยปกติจะมีอยู่ในฮาร์ดดิสก์จริงแต่ครั้งที่สร้างเวอร์ชวลแมชีน เวอร์ชวลไลเซชันซอฟต์แวร์จะสร้างเวอร์ชวลฮาร์ดดิสก์ขึ้นมา นั่นคือไฟล์ที่จะทำงานเหมือนกับดิสก์ที่มีเช็กเตอร์ทั่วไป เมื่อติดตั้งระบบปฏิบัติการบนเวอร์ชวลแมชีน ฮาร์ดดิสก์ไฟล์ จะถูกใส่เข้าไปในไฟล์นี้และเหมือนกับระบบจริง คือแต่ละเวอร์ชวลแมชีนสามารถมีดิสก์ไฟล์ได้หลายไฟล์ เนื่องจากมีจำลองฮาร์ดดิสก์ขึ้นมา ไฟล์นี้จึงสำคัญในด้านขนาด โดยระบบสามารถเริ่มต้นด้วยไฟล์ขนาดเล็ก และค่อย ๆ เพิ่มขนาดขึ้นเมื่อเนื้อหาใหม่ถูกใส่เข้าไปในเวอร์ชวลแมชีน

- ไฟล์สถานะของเวอร์ชวลแมชีน เช่นเดียวกับเครื่องจริงเวอร์ชวลแมชีนสนับสนุนโหมดปฏิบัติการที่คล้ายกับสแตนด์บายหรือไฮเบอร์เนชันในแง่ของเวอร์ชวลไลเซชันหมายถึงการหยุดชั่วคราวหรือค้างการทำงานไว้เหมือนกับการบันทึกสถานะของเครื่องเมื่อเครื่องถูกค้างการทำงานไว้ชั่วคราว สถานะที่ถูกหยุดของมันจะถูกบันทึกลงไปในไฟล์ เนื่องจากมีเพียงสถานะของเครื่องเท่านั้น โดยปกติไฟล์นี้จึงเล็กกว่าฮาร์ดดิสก์ไฟล์

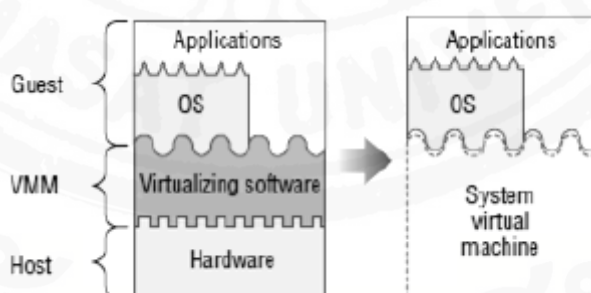
- ไฟล์อื่นๆ คือไฟล์ที่ประกอบด้วยล็อกและข้อมูลที่เกี่ยวข้องกับเวอร์ชวลแมชีนอื่นๆ

#### สถาปัตยกรรมของเวอร์ชวลแมชีน

จากมุมมองของโปรเซสที่เอ็กซิคิวต์ยูเซอร์โปรแกรม แมชีนจะประกอบไปด้วยพื้นที่แอดเดรสของหน่วยความจำที่จัดไว้ให้กับโปรเซสพร้อมกับคำสั่งระดับยูเซอร์และวีจีเอสเตอร์ที่ทำให้สามารถเอ็กซิคิวต์โค้ดที่เป็นของโปรเซสได้ ไอโอของเครื่องจะมองเห็นได้ โดยผ่านทางระบบปฏิบัติการเท่านั้น และวิธีการเดียวที่โปรเซสสามารถโต้ตอบกับระบบไอโอคือผ่านทางซิสเต็มส์คอล จากมุมมองของระบบปฏิบัติการและแอปพลิเคชันของตนเอง ระบบทั้งระบบจะรันบนเครื่องจริง ซึ่ง

ระบบคือสภาพแวดล้อมของการเอ็กซิคิวต์ที่สามารถสนับสนุนโปรเซสจำนวนมากได้ในเวลาเดียวกัน โปรเซสเหล่านี้จะใช้ไฟล์ซิสเต็มส์และทรัพยากรไอโออื่น ๆ ร่วมกัน โดยระบบจะจัดสรรหน่วยความจำจริงและทรัพยากรไอโอให้กับโปรเซสต่าง ๆ และยอมให้โปรเซสสามารถโต้ตอบกับทรัพยากรของตนเองได้ เนื่องจากมีมุมมองของโปรเซสและซิสเต็มส์ที่มีต่อ “แมชชีน” เวอร์ชวลแมชชีนจึงมีสองแบบ คือแบบโปรเซสและซิสเต็มส์ โปรเซสเวอร์ชวลแมชชีน (Process VM) คือเวอร์ชวลแพลตฟอร์มที่เอ็กซิคิวต์โปรเซสใดโปรเซสหนึ่ง โดย VM ประเภทนี้มีอยู่เพื่อช่วยเหลือโปรเซสเท่านั้น โดยจะถูกสร้างขึ้นเมื่อโปรเซสถูกสร้างขึ้นและจะจบการทำงานเมื่อโปรเซสจบการทำงาน (ตัวอย่างเช่น Java Virtual Machine) ในทางตรงข้ามระบบเวอร์ชวลแมชชีน (System VM) จะสร้างสภาพแวดล้อมของระบบที่สมบูรณ์และคงอยู่ตลอด ซึ่งสนับสนุนระบบปฏิบัติการพร้อมกับยูเชอร์โปรเซสจำนวนมากของตนเอง และช่วยให้เกสโอเอส (Guest Operating System) สามารถใช้งานทรัพยากรที่เป็นเวอร์ชวลฮาร์ดแวร์ซึ่งรวมทั้งเน็ตเวิร์คไอโอและอาจรวมถึงกราฟิกยูเชอร์อินเตอร์เฟสพร้อมกับโปรเซสเซอร์ และหน่วยความจำด้วยโปรเซสหรือซิสเต็มส์ที่รันบน VM คือเกส (Guest) และแพลตฟอร์มจริงที่สนับสนุน VM คือโฮส (Host) ซอฟต์แวร์เวอร์ชวลไลซ์ที่อิมพลีเมนต์โปรเซสเวอร์ชวลแมชชีนมักตั้งชื่อว่ารันไทม์ ซึ่งมาจากรันไทม์ซอฟต์แวร์ (Runtime Software) ส่วนซอฟต์แวร์เวอร์ชวลไลซ์ในซิสเต็มส์เวอร์ชวลแมชชีนโดยทั่วไปหมายถึงเวอร์ชวลแมชชีนมอนิเตอร์ (Virtual Machine Monitor หรือ VMM)

ภาพที่ 2.5 การแปลง ISA ของซอฟต์แวร์เวอร์ชวลไลซ์ในซิสเต็มส์เวอร์ชวลแมชชีน



ที่มา: “The architecture of virtual machines”

โดย Smith, J. E., & Ravi, N., 2005, *Computer*, 38(5), น. 34.

จากภาพที่ 2.5 ในซิสเต็มส์เวอร์ชวลแมชชีนซอฟต์แวร์ที่ทำเวอร์ชวลไลซ์จะอยู่ระหว่างฮาร์ดแวร์ของโฮสและซอฟต์แวร์ของเกส VMM จะจำลอง ISA ของฮาร์ดแวร์เพื่อที่ซอฟต์แวร์ของ



เกสจะได้สามารถเอ็กซีคิวต์ ISA ที่แตกต่างจากที่อิมพลีเมนต์อยู่บนโฮสได้ อย่างไรก็ตาม ในหลาย แอปพลิเคชันซิสเต็มส์เวอร์ชวลแมชชีน VMM ไม่ได้ทำการจำลองคำสั่งแต่หน้าที่หลัก คือจัดสรรทรัพยากรฮาร์ดแวร์ที่ถูกเวอร์ชวลไลซ์ให้กับเกส (Smith & Ravi, 2005, p. 34) จากมุมมองของผู้ใช้ซิสเต็มส์เวอร์ชวลแมชชีนส่วนใหญ่จะทำงานได้เหมือนกัน แต่แตกต่างกันในรายละเอียดการอิมพลีเมนต์ วิธีการแบบเวอร์ชวลแมชชีนมอนิเตอร์ซึ่งเป็นการวาง VMM ไว้บนฮาร์ดแวร์จริงและให้เวอร์ชวลแมชชีนอยู่ด้านบน โดย VMM จะรันในโหมดที่มีสิทธิสูงสุด ขณะที่เกสซิสเต็มส์ทั้งหมดจะรันด้วยสิทธิที่ต่ำกว่าเพื่อให้ VMM สามารถแทรกแซงและจำลองการกระทำของเกสโอเอสทั้งหมดที่จะใช้งานหรือจัดการทรัพยากรฮาร์ดแวร์ได้การอิมพลีเมนต์ซิสเต็มส์เวอร์ชวลแมชชีนแบบโฮสเวอร์ชวลแมชชีน (Host Virtual Machine) จะสร้างซอฟต์แวร์เวอร์ชวลไลซ์ด้านบนโฮสโอเอส ข้อดีของโฮสเวอร์ชวลแมชชีน คือผู้ใช้จะติดตั้งเหมือนกับเป็นแอปพลิเคชันโปรแกรมปกติ นอกจากนั้นซอฟต์แวร์เวอร์ชวลไลซ์ยังสามารถอาศัยโฮสโอเอสให้จัดดีไวซ์ไดรเวอร์ และเซอร์วิสระดับล่างอื่น ๆ ให้ แทนที่จะอาศัย VMM ในซิสเต็มส์เวอร์ชวลแมชชีนปกติซิสเต็มส์ซอฟต์แวร์ของทั้งโฮสและเกสรวมทั้งแอปพลิเคชันซอฟต์แวร์จะใช้ ISA เดียวกับฮาร์ดแวร์จริง อย่างไรก็ตามในบางสถานการณ์ระบบโฮสและเกสไม่ได้มี ISA เดียวกัน ดังนั้นเวอร์ชวลแมชชีนแบบทั้งระบบสามารถแก้ปัญหานี้ได้โดยการเวอร์ชวลไลซ์ซอฟต์แวร์ทั้งหมดซึ่งรวมทั้งระบบปฏิบัติการและแอปพลิเคชันด้วย เนื่องจาก ISA แตกต่างกันเวอร์ชวลแมชชีนจึงต้องจำลองโค้ดทั้งแอปพลิเคชัน และระบบปฏิบัติการเวอร์ชวลแมชชีนซอฟต์แวร์จะเอ็กซีคิวต์เหมือนกับเป็นแอปพลิเคชันโปรแกรมที่ได้รับการสนับสนุน โดยโฮสโอเอสและไม่ใช้โอเปอเรชั่น ISA ของระบบเมื่อโฮสแพลตฟอร์มเป็นมัลติโปรเซสเซอร์ขนาดใหญ่ที่ใช้งานหน่วยความจำร่วมกันจุดประสงค์ที่สำคัญคือการแบ่งระบบขนาดใหญ่ออกเป็นระบบมัลติโปรเซสเซอร์ที่เล็กลงหลาย ๆ ระบบโดยการกระจายทรัพยากรฮาร์ดแวร์ของระบบใหญ่ ด้วยการแบ่งแบบฟิสิกส์ทรัพยากรจริงที่เวอร์ชวลซิสเต็มส์หนึ่งใช้จะแยกออกจากทรัพยากรที่ถูกใช้ โดยเวอร์ชวลซิสเต็มส์อื่น ๆ การแบ่งแบบฟิสิกส์นี้ทำให้มีระดับของการแยกออกจากกันสูงดังนั้นปัญหาของซอฟต์แวร์ หรือข้อผิดพลาดของฮาร์ดแวร์บนพื้นที่แบ่งส่วนหนึ่งจะไม่มีผลกระทบต่อโปรแกรมในพื้นที่แบ่งส่วนอื่น ส่วนการแบ่งแบบโลจิคอลทรัพยากรฮาร์ดแวร์จริงจะถูกแบ่งเวลาระหว่างพื้นที่แบ่งต่าง ๆ ซึ่งทำให้การใช้งานทรัพยากรระบบดีขึ้น แต่จะสูญเสียประโยชน์บางอย่างของการแยกออกจากกันของฮาร์ดแวร์ไป โดยปกติเทคนิคการแบ่งพื้นที่ทั้งสองแบบใช้ซอฟต์แวร์ หรือเฟิร์มแวร์พิเศษที่มีการดัดแปลงฮาร์ดแวร์เฉพาะกับพื้นที่แบ่งที่ต้องการการใช้งานฟังก์ชันและการรันบนสถาปัตยกรรมที่แตกต่างกันได้เป็นเป้าหมายของระบบเวอร์ชวลแมชชีนส่วนใหญ่ที่ถูกอิมพลีเมนต์บนฮาร์ดแวร์ที่ถูกพัฒนาขึ้นสำหรับ ISA มาตรฐาน ในทางตรงข้ามเวอร์ชวลแมชชีนแบบโคดีไซน์

(Codesigned Virtual Machine) อิมพลีเมนต์ ISA ใหม่ที่มีเป้าหมายที่การปรับปรุงประสิทธิภาพและการใช้พลังงานให้ดีขึ้น โดยอาจมีการสร้าง ISA ของโฮสใหม่ทั้งหมดหรือเพิ่มขยาย ISA ที่มีอยู่ก็ได้ เวอร์ชวลแมชชีนแบบนี้ไม่มีแอปพลิเคชัน ISA จริง แต่ VMM จะกลายเป็นส่วนหนึ่งของการอิมพลีเมนต์ฮาร์ดแวร์มีหน้าที่อย่างเดียว คือจำลอง ISA ของเกส โดย VMM จะอยู่ในพื้นที่ของหน่วยความจำที่ถูกซ่อนจากซอฟต์แวร์ทั้งหมด ซึ่งรวมทั้งตัวแปลงไบนารีที่เปลี่ยนคำสั่งเกสให้เป็นคำสั่ง ISA ของโฮสและแคชคำสั่งเหล่านั้นเก็บไว้ในพื้นที่ของหน่วยความจำที่ถูกซ่อนไว้ (Smith & Ravi, 2005, pp. 36-37)

#### 2.1.4 อัลกอริทึมของนายธนาคาร (Banker's Algorithm) (Abraham & Peter Baer)

อัลกอริทึมของนายธนาคารเป็นอัลกอริทึมใช้สำหรับหลีกเลี่ยงปัญหาการติดตาย (Deadlock Avoidance) โดยการหลีกเลี่ยงปัญหาการติดตายนี้อาจแตกต่างกับการป้องกันการติดตาย (Deadlock Prevention) คือ การป้องกันการติดตายเป็นการป้องกันเพื่อไม่ให้เกิดการติดตายขึ้น โดยการสร้างข้อกำหนดในการร้องขอทรัพยากร เพื่อให้แน่ใจว่าเงื่อนไขข้อใดข้อหนึ่ง จะไม่เกิดขึ้นอย่างแน่นอน ซึ่งเงื่อนไขดังกล่าวประกอบไปด้วย

1. ห้ามใช้ทรัพยากรร่วมกัน (Mutual Exclusion) เงื่อนไขในข้อนี้ คือ การที่ระบบไม่อนุญาตให้มีการใช้ทรัพยากรร่วมกัน เช่น เครื่องพิมพ์จะไม่สามารถให้กระบวนการ (Process) หลาย ๆ กระบวนการใช้พร้อม ๆ กันได้

2. การถือครองแล้วรอคอย (Hold and Wait) คือ การที่จะไม่ให้เกิดการถือครองแล้วรอคอยขึ้นในระบบ โดยจะต้องกำหนดว่า เมื่อกระบวนการหนึ่งจะร้องขอทรัพยากร กระบวนการนั้นจะต้องไม่ได้ถือครองทรัพยากรใดๆ อยู่ในขณะนั้น ซึ่งอาจทำได้ 2 วิธีการ คือ

- (1) ให้กระบวนการร้องขอทรัพยากรที่ต้องการใช้ทั้งหมด (ตลอดการทำงาน) ก่อนที่จะเริ่มต้นการทำงาน เราอาจดำเนินการตามวิธีนี้ได้ โดยการกำหนดให้การร้องขอทรัพยากรเป็นคำสั่งเรียกระบบ (System call) ที่ต้องทำก่อนการทำงานใด ๆ ของกระบวนการเสมอ

- (2) ยอมให้กระบวนการร้องขอทรัพยากรได้ ก็ต่อเมื่อกระบวนการนั้นมิได้ถือครองทรัพยากรใดไว้เลย ตัวอย่างเช่น กระบวนการหนึ่งอาจร้องขอทรัพยากรบางส่วนและใช้ทรัพยากรนั้นไปก่อน และเมื่อกระบวนการนั้นต้องการทรัพยากรเพิ่มอีก กระบวนการนั้นก็จะต้องคืนทรัพยากรที่ถือครองอยู่กลับสู่ระบบเสียก่อน จึงจะร้องขอใหม่ได้

วิธีการแรกมีข้อเสีย คือ การใช้ทรัพยากรจะมีประสิทธิภาพต่ำมาก เพราะกระบวนการจำเป็นต้องร้องขอและถือครองทรัพยากรไว้ทั้งหมดตลอดช่วงเวลาการทำงาน ทั้ง ๆ ที่การใช้ทรัพยากรแต่ละตัวอาจเป็นเพียงช่วงเวลาสั้น ๆ ก็ตาม

นอกจากนั้นอาจมีปัญหา Starvation อีกด้วย โดยถ้ามีบางกระบวนการต้องการใช้ทรัพยากรหลาย ๆ ตัว อาจต้องรอคอยอย่างไม่มีที่สิ้นสุด เพราะทรัพยากรตัวหนึ่งในจำนวนที่ต้องการอาจมีกระบวนการอื่นใช้อยู่ส่วนวิธีการหลังก็จะมีข้อเสีย คือ ต้องคืนทรัพยากรที่ถือครองอยู่ เพื่อที่จะร้องขอกลับมาใหม่อีกครั้งร่วมกับทรัพยากรตัวใหม่ ทำให้เสียเวลาโดยเปล่าประโยชน์

3. ห้ามแทรกกลางคัน (No Preemption) เราอาจกำหนดกฎเกณฑ์ ดังนี้ ถ้ากระบวนการหนึ่ง (ที่กำลังถือครองทรัพยากรบางส่วนอยู่) และระบบยังไม่สามารถจัดให้ได้ทันที (แสดงว่ากระบวนการที่ร้องขอจะต้องรอ) เราใช้ทรัพยากรทั้งหมดที่กระบวนการนี้ถือครองอยู่ถูกแทรกกลางคัน นั่นคือ ทรัพยากรที่กระบวนการนี้ถือครองอยู่ทั้งหมดจะถูกปล่อยคืนสู่ระบบโดยปริยาย กระบวนการที่ถูกแทรกกลางคันนี้จะต้องรอคอยทรัพยากร ทั้งที่ร้องขอไว้ตั้งแต่แรกและที่ถูกแทรกกลางคันไป ก่อนที่จะสามารถทำงานต่อไปได้

หรืออาจกล่าวได้ว่า ถ้ามีกระบวนการหนึ่งได้ร้องขอทรัพยากรบางส่วนจากระบบ ในตอนแรกระบบจะตรวจสอบว่า ทรัพยากรที่ร้องขอนั้นว่างอยู่หรือไม่ ถ้าว่างอยู่ ระบบก็จะจัดสรรทรัพยากรเหล่านั้นให้แก่กระบวนการ แต่ถ้ากระบวนการที่ถูกร้องขอนั้นไม่ว่าง ระบบจะตรวจสอบก่อนว่าทรัพยากรนั้นไม่ว่างเนื่องจากอะไร ถ้าไม่ว่างเนื่องจากกำลังถูกถือครองโดยกระบวนการอื่น ซึ่งกำลังรอคอยทรัพยากรเพิ่มอยู่ ระบบจะทำการแทรกกลางคันทรัพยากรทั้งหมดของกระบวนการนั้น และจัดสรรทรัพยากรที่ได้มาแก่กระบวนการที่ร้องขอ แต่ถ้ากระบวนการที่ไม่ว่างนั้นไม่ได้ถูกถือครองโดยกระบวนการอื่นที่กำลังรออยู่ ระบบก็จะให้กระบวนการที่ร้องขอทรัพยากรนั้นรอ และขณะที่รอนั้น ทรัพยากรทั้งหมดที่กระบวนการนี้ถือครองอยู่อาจถูกแทรกกลางคันได้ เมื่อมีกระบวนการอื่นร้องขอ และกระบวนการนี้จะสามารถกลับไปทำงานต่อได้ เมื่อได้รับจัดสรรทรัพยากรที่ร้องขอและได้รับทรัพยากรที่อาจถูกแทรกกลางคันไปคืนทั้งหมดเสียก่อน

วิธีการนี้มักใช้กับทรัพยากรที่สามารถเก็บค่าสถานะและติดตั้งค่ากลับคืนมาได้ง่าย เช่น ค่าในรีจิสเตอร์ (ของหน่วยประมวลผลกลาง) เนื้อที่ในหน่วยความจำหลัก เป็นต้น แต่จะไม่สามารถใช้กับทรัพยากรทั่ว ๆ ไปได้ เช่น เครื่องพิมพ์ และหน่วยขับเทป เป็นต้น

4. วงจรรอคอย (Circular Wait) เราอาจป้องกันการเกิดการติดตาย โดยการป้องกันไม่ให้เกิดเงื่อนไขวงจรรอคอย ซึ่งสามารถทำได้โดยการกำหนดลำดับของทรัพยากรทั้งหมดในระบบ และกำหนดให้กระบวนการต้องร้องขอใช้ทรัพยากร เรียงตามเลขลำดับนี้

กำหนดให้  $R = \{ R_1, R_2, \dots, R_m \}$  โดย  $R$  เป็นเซตของทรัพยากรในระบบ และกำหนดให้ทรัพยากรแต่ละประเภทมีค่าลำดับเป็น เลขจำนวนเต็มที่ไม่ซ้ำกัน เขียนแทนด้วย  $F(R_i)$  เพื่อให้เราเปรียบเทียบทรัพยากร 2 ประเภทได้ว่าตัวใดมีลำดับก่อน-หลัง ตัวอย่างเช่น ถ้าเซตของทรัพยากร  $R$  ประกอบด้วย เครื่องขับเทป เครื่องขับจานบันทึก และเครื่องพิมพ์ ดังนั้นค่าเลขลำดับ  $F(R_i)$  อาจถูกกำหนดได้ ดังนี้คือ

$$F(\text{เครื่องขับเทป}) = 1$$

$$F(\text{เครื่องขับดิสก์}) = 5$$

$$F(\text{เครื่องพิมพ์}) = 12$$

และกำหนดวิธีการในการร้องขอทรัพยากรในระบบ ดังนี้

กระบวนการแต่ละตัวสามารถร้องขอทรัพยากรได้ในลำดับที่เพิ่มขึ้นเท่านั้น คือ เริ่มต้นกระบวนการอาจร้องขอทรัพยากรใด ๆ ก็ได้ เช่น ทรัพยากร  $R_i$  แต่ต่อจากนี้กระบวนการจะร้องขอทรัพยากร  $R_j$  ได้ก็ต่อเมื่อ  $F(R_j) > F(R_i)$  ถ้าเป็นการร้องขอทรัพยากรประเภทเดียวกันหลาย ๆ ตัวกระบวนการจะต้องร้องขอทรัพยากรทีละตัว จากตัวอย่าง เลขลำดับทรัพยากรข้างต้น ถ้ากระบวนการหนึ่งต้องการใช้เครื่องขับเทป ( $F(R) = 1$ ) และเครื่องพิมพ์ ( $F(R) = 12$ ) กระบวนการนั้นจะต้องร้องขอเครื่องขับเทปก่อน แล้วจึงร้องขอเครื่องพิมพ์จะขอกลับกันไม่ได้ (ระบบไม่อนุวัติ) ในทางตรงกันข้าม ถ้ากระบวนการร้องขอทรัพยากรประเภท  $R_i$  กระบวนการจะต้องปล่อยทรัพยากร  $R_i$  ซึ่ง  $F(R_i) \geq F(R_j)$  คืนสู่ระบบทุกตัวเสียก่อน เช่น ถัดครอง  $R_5$  อยู่อยู่ยากได้  $R_1$  ต้องคืน  $R_5$  ก่อน  $R_5 \geq R_1$

ตามข้อกำหนดที่กล่าวมา จะเห็นว่าเงื่อนไขวงจรรอคอยจะไม่สามารถเกิดขึ้นได้ สามารถพิสูจน์โดยวิธียกสิ่งตรงข้ามได้ (Proof by Contradiction) ดังนี้

สมมติให้เกิดวงจรรอคอยในระบบ คือ  $\{ P_1, P_2, \dots, P_n \}$  โดยที่กระบวนการ  $P_1$  รอคอยทรัพยากร  $R_1$  ซึ่งกำลังถูกถือครองโดยกระบวนการ  $P_{i+1}$  ถือครองทรัพยากร  $R_i$  อยู่ ขณะที่ร้องขอทรัพยากร  $R_{i+1}$  เราจะได้ว่า  $F(R_i) < F(R_{i+1})$  สำหรับทุก ๆ ค่าของ  $i$  (โดยข้อกำหนดที่ตั้งไว้) ซึ่งหมายความว่า  $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$  ดังนั้น  $F(R_0) < F(R_0)$  เอง ซึ่งเป็นไปไม่ได้สรุปว่าไม่มีวงจรรอคอยในระบบ

เงื่อนไขที่กล่าวมาทั้ง 4 ข้อนี้เป็นการป้องกันการติดตาย แต่สำหรับการหลีกเลี่ยงปัญหาการติดตาย (Deadlock Avoidance) นั้น เราต้องมีข้อมูลเกี่ยวกับการร้องขอทรัพยากรในระบบโดยรวม โดยพิจารณาจากข้อมูลของทรัพยากรที่ถูกร้องขอ เช่น ในระบบที่มีเทป และเครื่องพิมพ์อย่างละตัว ดังนั้นเราอาจจัดสรรให้โปรเซส  $P$  เข้าใช้งานเทปก่อนแล้วจึงใช้งาน



เครื่องพิมพ์ ในขณะที่โพรเซส Q ใช้งานเครื่องพิมพ์ก่อนแล้วค่อยใช้เทป ดังนั้นนอกจากการร้องขอแล้วยังต้องตรวจว่าทรัพยากรว่างหรือไม่ ทรัพยากรนั้นถูกใช้โดยใคร และโพรเซสไหนจะใช้อะไรก่อนหลัง นอกจากนี้อาจมีข้อมูลมากที่สุดที่ขอใช้ทรัพยากรชนิดนั้น รูปแบบอัลกอริทึมของการหลีกเลี่ยงการติดตายจะมีการตรวจสอบสถานะของการใช้ทรัพยากรเพื่อให้แน่ใจว่าจะไม่เกิดการเกิดการรบกวนอย่างสม่ำเสมอตลอดเวลา สถานะของการใช้ทรัพยากรถูกกำหนดด้วยจำนวนของทรัพยากรที่ถูกใช้งานและจำนวนทรัพยากรที่มีอยู่ในระบบ (วรรณรัช สันติอมรทัต)

ในระบบที่ทรัพยากร 1 ตัวสามารถให้บริการได้พร้อมกันหลายโพรเซส อัลกอริทึมของนายธนาคาร ซึ่งเป็นอัลกอริทึมที่ใช้งานได้จริงในระบบธนาคารที่ว่า ธนาคารจะไม่จ่ายเงินที่มีอยู่ให้ตามความต้องการของลูกค้าทั้งหมดได้เป็นเวลานาน (หมายถึงมีคนถอนออกอยากเดียว ธนาคารจะไม่สามารถอยู่ได้) ดังนั้นจึงต้องมีระบบเพื่อกำหนดจำนวนสูงสุดที่จะสามารถให้บริการได้จำนวนนี้อาจไม่จำเป็นต้องเป็นจำนวนทรัพยากรทั้งหมดที่มีอยู่ในระบบ เมื่อผู้ใช้ขอใช้กลุ่มของทรัพยากร ระบบต้องทำการตรวจสอบว่าถ้าให้ใช้แล้วระบบจะยังคงอยู่ในภาวะปลอดภัยหรือไม่ ถ้าไม่ปลอดภัยการให้ใช้ทรัพยากรต้องรอจนกว่าจะมีผู้ใช้รายอื่นทรัพยากรที่ใช้แล้วกลับเข้าสู่ระบบ

โครงสร้างของระบบนายธนาคารมีดังนี้ กำหนดให้มี  $n$  โพรเซส มีทรัพยากรในระบบทั้งหมด  $m$  ตัว

**Available** : เป็นเวกเตอร์ของขนาดที่ใช้ชี้จำนวนของทรัพยากรที่สามารถใช้งานได้  
 $Available[j] = k$  ดังนั้น  $k$  คือจำนวนที่ทรัพยากรชนิด  $j$  จะสามารถให้บริการได้

**Max** : เป็นค่าเมตริกซ์ขนาด  $n \times m$  ที่กำหนดความต้องการสูงสุดของแต่ละ โพรเซส  
 ถ้า  $Max[i,j] = k$  แล้วโพรเซส  $i$  อาจต้องใช้ทรัพยากร  $j$  สูงถึง  $k$  บริการ

**Allocation** เป็นเมตริกซ์ขนาด  $n \times m$  ที่กำหนดจำนวนของทรัพยากรในแต่ละชนิดที่ให้บริการแต่ละโพรเซสอยู่ได้ ถ้า  $Allocation[i,j] = k$  หมายถึงโพรเซส  $i$  กำลังใช้งานทรัพยากรชนิด  $j$  อยู่เป็นจำนวน  $k$  บริการ

**Need** : เมตริกซ์ขนาด  $n \times m$  เพื่อบ่งบอกจำนวนทรัพยากรที่เหลือที่ยังต้องการใช้ของแต่ละโพรเซส เช่น  $Need[i,j] = k$  หมายถึงโพรเซส  $i$  ยังคงต้องการใช้งานทรัพยากร  $j$  อยู่อีก  $k$  บริการ พบว่า  $Need[i,j] = Max[i,j] - Allocation[i,j]$

### อัลกอริทึมที่ปลอดภัย

อัลกอริทึมที่หาได้ว่าระบบปลอดภัยหรือไม่ สามารถทำได้ดังนี้

1. กำหนดให้ Work และ Finish เป็นเวกเตอร์ที่มีขนาด  $n \times m$  โดยเริ่มทำงานที่  
 $Work := available$  และ  $Finish[i]$  เป็นเท็จ โดยที่  $i$  มีค่าตั้งแต่ 1, 2, 3 ... n

2. หาค่า  $i$  ทั้ง 2 พังค์ชัน คือ  $Finish[i] := false$  ,  $Need_i \leq Work$  ถ้าไม่ตามเงื่อนไข ข้ามไปยังขั้นที่ 4
3.  $Work := Work + Allocation$  ,  $Finish[i] := true$  กลับไปยังขั้นที่ 2
4. ถ้า  $Finish[i] = true$  สำหรับทุกค่าของ  $i$  แล้วระบบจะอยู่ในภาวะปลอดภัย เวลาในการทำงานอัลกอริทึมนี้เท่ากับ  $m \times n^2$

### อัลกอริทึมของการขอใช้ทรัพยากร

กำหนดให้  $Request_i$  เป็นเวกเตอร์ของการขอใช้งานสำหรับโพรเซส  $i$  ถ้า  $Request_i[j] = k$  หมายถึง โพรเซส  $i$  ต้องการทำงาน  $k$  บริการจากทรัพยากร  $j$  เมื่อมีการขอใช้งานทรัพยากรโดยโพรเซส  $i$  ก็จะทำให้เกิดการทำงานดังต่อไปนี้

1. ถ้า  $Request_i \leq Need_i$  ไปยังขั้นตอนที่ 2 นอกจากนั้น ให้แสดงข้อความเตือนเนื่องจากโพรเซสมีการใช้ทรัพยากรมากกว่าที่คาดการณ์ไว้
2. ถ้า  $Request_i \leq Available$  ไปยังขั้นตอนที่ 3 นอกจากนั้นโพรเซส  $i$  ต้องรอเนื่องจากทรัพยากรไม่มีให้ใช้งาน
3. มีการตั้งระบบลงเพื่อใช้ทรัพยากรที่โพรเซส  $i$  ขอใช้ โดยการใส่ค่าในตัวแปรต่อไปนี้

$Available := Available - Request_i$ ;

$Allocation_i := Allocation_i + Request_i$  ;

$Need_i := Need_i - Request_i$  ;

ถ้าผลของการกำหนดค่าการให้ใช้ทรัพยากรออกมาพบว่าระบบอยู่ในภาวะปลอดภัย ก็จะจบสิ้นการทำงานแล้วยอมให้โพรเซส  $i$  ใช้งานทรัพยากรนั้นจริงๆ อย่างไรก็ตามถ้าผลออกมาว่าไม่ปลอดภัย โพรเซส  $i$  ต้องรอ  $Request_i$  แล้วก็จะกลับไปสู่ค่าสถานะเก่า

ภาพที่ 2.6 ตัวอย่างข้อมูลของระบบที่ใช้งาน

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	7 5 3	3 3 2
$P_1$	2 0 0	3 2 2	
$P_2$	3 0 2	9 0 2	
$P_3$	2 1 1	2 2 2	
$P_4$	0 0 2	4 3 3	

ที่มา: “Operating System Concepts” โดย Abraham & Peter Baer, 1998, น. 49.

จากภาพที่ 2.6 แสดงข้อมูลการใช้งานของระบบพบว่ามี 5 โพรเซสในการทำงาน  $P_0 - P_4$  และมีทรัพยากรให้ใช้งานได้ 3 ตัว คือ A, B และ C ทรัพยากร A มีให้ใช้ได้ถึง 10 ตัว ทรัพยากร B ใช้ได้ถึง 5 ตัว และทรัพยากร C ใช้ได้ถึง 7 ตัว สมมติว่าที่เวลา  $t_0$  เกิดเหตุการณ์ดังภาพที่ 2.6 ดังนั้นจะสามารถหา  $Need := Max - Allocation$  ดังภาพที่ 2.7 พบว่าระบบอยู่ในภาวะปลอดภัยแน่นอน และลำดับของความปลอดภัยเป็น  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$

ภาพที่ 2.7 ตัวอย่างค่าของ  $Need := Max - Allocation$  ของแต่ละโพรเซส

	<u>Need</u>
	A B C
$P_0$	7 4 3
$P_1$	1 2 2
$P_2$	6 0 0
$P_3$	0 1 1
$P_4$	4 3 1

ที่มา: “Operating System Concepts” โดย Abraham & Peter Baer, 1998, น. 49.

สมมติให้โพรเซส 1 มีการขอใช้งานทรัพยากรเพิ่ม 1 ตัว ของ A และจากทรัพยากร C อีก 2 ตัว ดังนั้น  $Request_1 := (1, 0, 2)$  ซึ่งสามารถตรวจสอบได้จาก  $Request_1 \leq Available$  :  $(1, 0, 2) \leq (3, 3, 2)$  เป็นจริงดังนั้นจึงต้องทำการสร้างระบบจำลองขึ้นมาเพื่อตรวจสอบภาวะของระบบดังภาพที่ 2.8 หลังจากนั้นไปเข้าไปทำอัลกอริทึมที่ปลอดภัยเพื่อหาลำดับความปลอดภัย

และได้ลำดับออกมาดังนี้  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  แต่เราพบว่าถ้าโปรเซส  $P_4$  มีการขอใช้ทรัพยากร (3,3,0) จะไม่มีให้ใช้งานได้ และถ้า  $P_0$  ขอใช้งาน (0,2,0) ก็จะใช้ไม่ได้เช่นกันเนื่องจากจะทำให้อยู่ในภาวะไม่ปลอดภัย

ภาพที่ 2.8 ตัวอย่างระบบจำลองที่สร้างเพื่อตรวจสอบภาวะของระบบ

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
$P_0$	0 1 0	7 4 3	2 3 0
$P_1$	3 0 2	0 2 0	
$P_2$	3 0 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

ที่มา: “Operating System Concepts” โดย Abraham & Peter Baer, 1998, น. 49.

## 2.2 งานวิจัยที่เกี่ยวข้อง

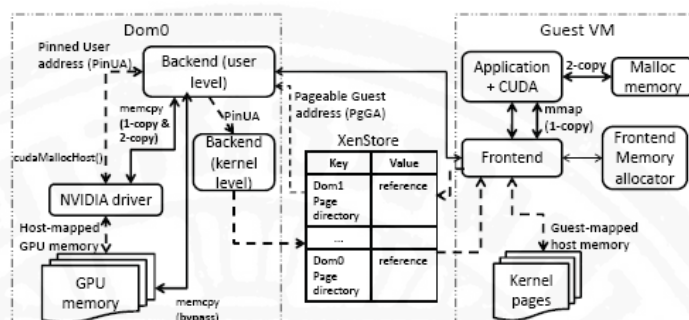
ส่วนนี้กล่าวถึงงานวิจัยที่เกี่ยวข้อง ซึ่งเกี่ยวกับการเข้าถึงทรัพยากรจีพียู และส่งงานจีพียูจากเวอร์ชวลแมชชีนในรูปแบบต่าง ๆ โดยมีรายละเอียดดังนี้

งานวิจัย Vishakha Gupta et al เสนอการออกแบบระบบจีวีเอ็ม (GVIM) เพื่อให้สามารถใช้ทรัพยากรจีพียูร่วมกันสำหรับเวอร์ชวลแมชชีนโดยใช้การจำลองคูด้าเอพีไอ (CUDA API) ภายใต้ระบบเซน (Xen) และใช้เครื่องมือเซนสเตอร์ (XenStore) ในการแชร์ข้อมูลระหว่างเวอร์ชวลแมชชีนกับเครื่องจริง ข้อดีของจีวีเอ็มคือมีค่าโอเวอร์เฮดในการถ่ายโอนข้อมูลระหว่างเวอร์ชวลแมชชีนกับ จีพียูที่น้อยกว่างานอื่น แต่มีข้อจำกัดคือ จีวีเอ็มสนับสนุนการใช้งานจีพียูร่วมกันระหว่างเวอร์ชวล แมชชีนที่อยู่บนระบบเซนที่รันอยู่บนเครื่องคอมพิวเตอร์เครื่องเดียวกันเท่านั้น และไม่รองรับแอปพลิเคชันกราฟิก

วิธีการออกแบบระบบของจีวีเอ็มเพื่อให้สามารถใช้ทรัพยากรจีพียูร่วมกันสำหรับหลายเวอร์ชวลแมชชีนบน Xen-base ได้แบ่งการทำงานสำหรับการเข้าถึงจีพียูออกเป็น 2 ส่วนการทำงานหลัก คือ 1) การเข้าถึงจีพียูระหว่างโดเมนซีโร่ (Dom0) และเกสเวอร์ชวลแมชชีน 2) การจองพื้นที่หน่วยความจำสำหรับประมวลผลคูด้าแอปพลิเคชันโดเมนซีโร่ (Dom0) และเกสเวอร์ชวลแมชชีน



ภาพที่ 2.9 ภาพรวมของระบบ GViM และการจัดการพื้นที่หน่วยความจำ



ที่มา: “GViM: GPU-accelerated virtual machines”

โดย V. Gupta et al, ACM, 2009.

1. การจัดการ การเข้าถึงจีพียูระหว่างโดเมนซีโร่ (Dom0) และเกสเวอร์ชวลแมชีน แบ่งการทำงานออกเป็นรายละเอียดดังนี้

- ฟรอนท์เอนด์ไดรเวอร์ ซึ่งถูกติดตั้งอยู่ที่เกสเวอร์ชวลแมชีนทำหน้าที่จัดการสำหรับการติดต่อสื่อสารระหว่างเกสเวอร์ชวลแมชีนและโดเมนซีโร่ โดยใช้ช่องทางในการติดต่อสื่อสารผ่าน Xen-bus และ Xenstore ในการรับแพคเกจข้อมูลจากไลบรารีที่สร้างขึ้นมา สำหรับจัดการแพคเกจข้อมูลและส่งแพคเกจที่ได้จากการแพคเกจไปยังแบคเอนด์ที่โดเมนซีโร่เพื่อทำการประมวลผลและรับข้อมูลกลับจากการประมวลผล

- แบคเอนด์ ถูกติดตั้งอยู่ที่โดเมนซีโร่มีหน้าที่เป็นตัวกลางคอยประสานงานการเข้าถึงจีพียูจากการร้องขอของคูด้า (CUDA) ที่ได้รับมาจากเกสเวอร์ชวลแมชีนผ่านทาง ฟรอนท์เอนด์เพื่อประมวลผลในจีพียูและส่งค่าที่ได้กลับไปยังฟรอนท์เอนด์

2. การจัดการ การจองพื้นที่หน่วยความจำสำหรับประมวลผลคูด้าแอปพลิเคชัน โดเมนซีโร่ (Dom0) และเกสเวอร์ชวลแมชีน

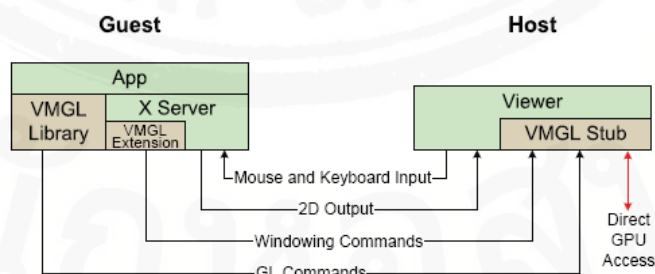
ในการเรียกใช้งานจีพียูจากเกสเวอร์ชวลแมชีนเพื่อส่งข้อมูลสำหรับนำไปให้โดเมนซีโร่ส่งไปประมวลผลในจีพียูนั่นงานวิจัยนี้มีการนำเสนอการส่งข้อมูล 3 วิธี โดยมีรายละเอียดการจัดการพื้นที่หน่วยความจำดังนี้

- 2-copy เกสเวอร์ชวลแมชีนจะจองพื้นที่บนหน่วยความจำตามการร้องขอของคูด้า แอปพลิเคชันบนเกสเวอร์ชวลแมชีน เมื่อต้องการส่งข้อมูลไปให้โดเมนซีโร่ทำงานจะทำการคัดลอกข้อมูลที่มีอยู่บนหน่วยความจำที่จองไว้ส่งไปยังหน่วยความจำบนโฮสจากนั้นจึงคัดลอกจากหน่วยความจำบนโฮสไปยังจีพียู

- 1-copy เป็นการคัดลอกโดยให้ผู้ใช้เรียก mmap() ที่อยู่ในฟรอนท์เอนด์แทนการเรียก malloc เพื่อลดการคัดลอกข้อมูลจากหน่วยความจำของเกสเวอร์ชวลแมชชีนมาที่หน่วยความจำของโฮส
- Bypass เป็นการลดการคัดลอกข้อมูลระหว่างกันทั้งหมดโดยการผสานพื้นที่หน่วยความจำของเกสเวอร์ชวลแมชชีนและจีพียูเข้าด้วยกันเพื่อลดการคัดลอกระหว่างกัน

งานวิจัย H. Andres Lagar-Cavilla et al นำเสนองานวิจัยที่มีชื่อว่าวีเอ็มจีแอล (VMGL) งานวิจัยหลายงานที่สร้างระบบเพื่อให้เข้าถึงทรัพยากรจีพียูจากเวอร์ชวลแมชชีนแต่มีความแตกต่างกับงานวิจัยนี้ ในเรื่องของรูปแบบในการเข้าถึงและลักษณะการจัดสรรทรัพยากรจีพียู งานวิจัยระบบวีเอ็มจีแอล (VMGL) มุ่งเน้นการเข้าถึงจีพียูจากเวอร์ชวลแมชชีนโดยจำลองโอเพนจีแอลเอพีไอ (OpenGL API เป็นเอพีไอมาตรฐานสำหรับการออกคำสั่งให้จีพียูประมวลผลทางด้านกราฟิก) บนเวอร์ชวลแมชชีน ระบบวีเอ็มจีแอลในงานวิจัยนี้ออนุญาตให้แอปพลิเคชันที่ต้องการประมวลผลกราฟิกอยู่บนเวอร์ชวลแมชชีนสามารถเรียกใช้งานทรัพยากรจีพียูได้ แต่งานวิจัยนี้ไม่ได้สนับสนุนการเข้าถึงจีพียูเพื่อการประมวลผลสมรรถนะสูงของแอปพลิเคชันที่ใช้คู่มือเอพีไอแต่จะมุ่งเน้นไปทางด้านการทำโอเพนจีแอลเวอร์ชวลไลเซชัน ซึ่งอนุญาตให้แอปพลิเคชันต่าง ๆ ที่ประมวลผลด้านกราฟิกที่ต้องใช้งานจีพียูและอยู่บนเวอร์ชวลแมชชีนสามารถประมวลผลได้ นอกจากนี้ วีเอ็มจีแอลยังมีความสามารถจัดการการทำงานของแอปพลิเคชันที่ทำงานบนจีพียูต่างค่ายกันได้

ภาพที่ 2.10 แสดงโครงสร้างของระบบวีเอ็มจีแอล (VMGL)



ที่มา: “VMM-independent graphics acceleration”

โดย H. A. Lagar-Cavilla et al., ACM, 2007.

โครงสร้างของวีเอ็มจีแอลประกอบไปด้วย 3 ยูสเซอร์-สเปซ โมเดล คือ

- 1) วีเอ็มจีแอลไลบรารี (VMGL library)
- 2) วีเอ็มจีแอลสตับ (VMGL stub)
- 3) วีเอ็มจีแอลเซิร์ฟเวอร์เอ็กซ์เทนชัน (VMGL X server extension)

จากการทำวิจัยวีเอ็มจีแอลนี้ได้ผลการทดลองว่าวีเอ็มจีแอลมีประสิทธิภาพในการจัดการการแสดงผลแอปพลิเคชันด้านกราฟิกที่ปฏิบัติงานบนเวอร์ชวลแมชีนได้ดี

งานวิจัยของ Dowty และ Sugerman เสนอการเข้าถึงจีพียูโดยใช้การจำลองจีพียูในระดับของฮาร์ดแวร์แทนที่จะเป็นการจำลองคุ้ด้าหรือโอเพนจีแอลเอพีไอดังเช่นในงานวิจัยนี้ หรือสองงานข้างต้น ทำให้สามารถรองรับทั้งแอปพลิเคชันทางด้านกราฟิกและการประมวลผลสมรรถนะสูงที่ใช้คุ้ด้าเอพีไอได้ แต่งานวิจัยนี้ก็มีความซับซ้อนสูงและขึ้นอยู่กับระบบวีเอ็มแวร์ (vmware) และไม่ได้พิจารณาปัญหาการติดตายเมื่อคุ้ด้าแอปพลิเคชันจากเวอร์ชวลแมชีนหลายเครื่องต้องการใช้งานจีพียูพร้อม ๆ กัน

จากงานวิจัยที่ผ่านมา นั้น การเข้าถึงจีพียูเพื่อส่งงาน โดยผ่านคุ้ด้าสำหรับเครื่องที่เป็นเวอร์ชวลแมชีน ส่วนใหญ่จะออกแบบระบบให้มีการทำงานในลักษณะฟรอนท์เอนด์ไลบรารี กับแบ็คเอนด์ แต่ยังไม่ม้งานวิจัยใดพิจารณาเรื่องระบบการจัดการ การเข้าถึงจีพียูพร้อม ๆ กัน ซึ่งอาจทำให้เกิดปัญหาการติดตาย กรณีขอใช้ทรัพยากรจีพียู และการเพิ่มประโยชน์การใช้งานทรัพยากรจีพียูร่วมกัน จึงทำให้มีแนวคิดที่ขอนำเสนอระบบเวอร์ชวลคุ้ด้า เพื่อจัดการ การเข้าถึงจีพียู และการเพิ่มประโยชน์การใช้งานจีพียู ดังจะกล่าวในลำดับถัดไป

### บทที่ 3

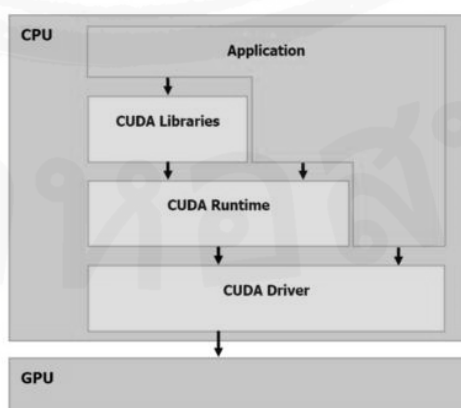
#### วิธีการดำเนินงานวิจัย

วิทยานิพนธ์ฉบับนี้พัฒนาระบบที่เรียกว่า **เวอร์ชวลคูด้า** มีวัตถุประสงค์เพื่อออกแบบการใช้ทรัพยากรจีพียูร่วมกันจากเครื่องที่ให้บริการทรัพยากรจีพียู (Server) ได้อย่างมีประสิทธิภาพ และวิธีการจัดสรรทรัพยากรจีพียูให้เพียงพอต่อการใช้งานสำหรับลูกค้าแอปพลิเคชันโดยไม่จำกัดว่าผู้ที่ขอรับบริการใช้งานจีพียู (Client) นั้น จะเป็นลูกค้าแอปพลิเคชันบนเวอร์ชวลแมชีนหรือลูกค้าแอปพลิเคชันบนเครื่องจริงก็ได้

โดยทั่วไป การเข้าถึงและสั่งงานจีพียูสำหรับการประมวลผลการทำงานของคูด้าแอปพลิเคชันนั้น แอปพลิเคชันจะเรียกใช้งานผ่านคูด้าไดรเวอร์สำหรับการเข้าถึงและสั่งงานไปยังอุปกรณ์จีพียู ไม่ว่าจะเรียกใช้งานในระดับต่ำหรือที่เรียกว่า CUDA driver API คือในส่วนของ CUDA Driver หรือระดับสูงที่เรียกว่า CUDA runtime API คือในส่วนของ CUDA Libraries และ CUDA Runtime ดังภาพที่ 3.1 แต่ทั้งนี้ในระบบคอมพิวเตอร์ที่มีการใช้งานเวอร์ชวลแมชีนนั้น การเรียกใช้งานผ่านคูด้าไดรเวอร์เพื่อเข้าถึงและสั่งงานไปยังจีพียูจะไม่สามารถเรียกได้จากแอปพลิเคชันที่ทำงานอยู่บนเวอร์ชวลแมชีน เนื่องจากเกสโอเปอเรติงซิสเต็มของเวอร์ชวลแมชีนไม่สามารถเข้าถึงจีพียูได้โดยตรง

วิทยานิพนธ์ฉบับนี้จึงได้นำเสนอออกแบบและพัฒนาระบบเวอร์ชวลคูด้าเพื่อสนับสนุนการเข้าถึงและเรียกใช้งานจีพียูขึ้น โดยมีรายละเอียดการออกแบบดังต่อไปนี้

ภาพที่ 3.1 Compute Unified Device Architecture Software Stack

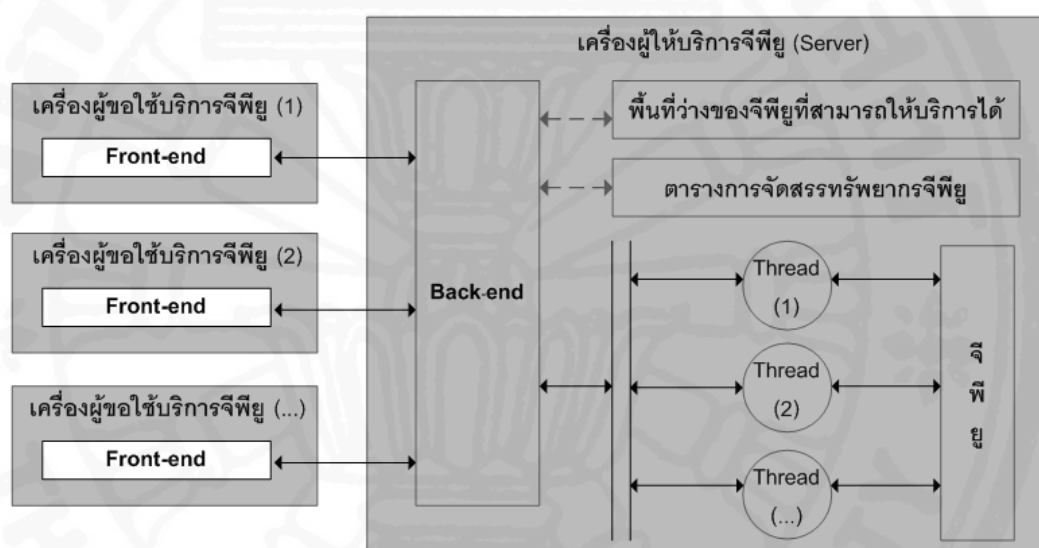


ที่มา : “NVIDIA CUDA Compute Unified Device Architecture” โดย nVidia, 2008, น.12

### 3.1 การออกแบบระบบ

วิทยานิพนธ์ฉบับนี้ได้ออกแบบวิธีการเข้าถึงทรัพยากรจีพียูและวิธีการติดต่อสื่อสารระหว่างเครื่องที่ต้องการใช้ทรัพยากรจีพียูกับเครื่องที่ให้บริการทรัพยากรจีพียูโดยได้ออกแบบสถาปัตยกรรมการทำงานของระบบเป็นภาพโดยรวม ดังภาพที่ 3.2

ภาพที่ 3.2 สถาปัตยกรรมภาพรวมของระบบเวอร์ชวลคู่ค้า



การทำงานของระบบจากภาพที่ 3.2 สามารถอธิบายการลำดับขั้นตอนในการทำงานได้ดังนี้

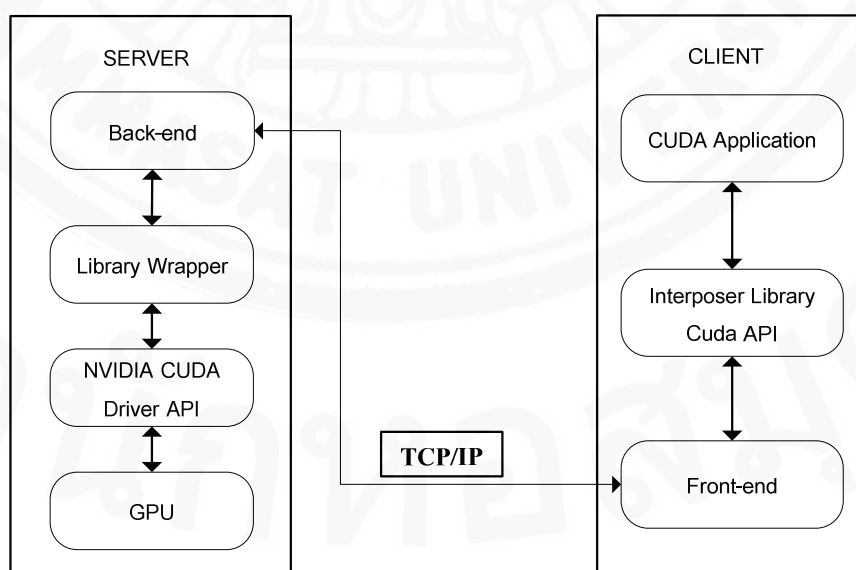
1. คู่ค้าแอปพลิเคชันบนเครื่องผู้ขอรับบริการจีพียูจะเรียกใช้งานจีพียูผ่านทางพรอนต์เอนด์ไอบารี่ (Front-end ดังภาพ) ที่จำลองคู่ค้าเอพีไอ บนเครื่องของตน
2. เมื่อคำร้องขอใช้จีพียูได้ถูกส่งมาถึงเครื่องผู้ให้บริการจีพียู เครื่องของผู้ให้บริการจะมีแบคเอนด์ซอฟต์แวร์ (Back-end ดังภาพ) คอยจัดการคำร้องขอการใช้ทรัพยากรจีพียู
3. แบคเอนด์จะสร้างเธรด (Thread) การทำงานของผู้ขอรับบริการจีพียูโดยจะสร้าง 1 เธรดการทำงานต่อ 1 ผู้ขอรับบริการ เพื่อให้สามารถเรียกใช้งานจีพียูพร้อม ๆ กันได้ครั้งละหลายผู้ขอรับบริการจีพียู

4. ระหว่างการให้บริการจีพียู แบ็คเอนด์จำเป็นต้องคอยตรวจสอบว่าทรัพยากรจีพียูมีหน่วยความจำสำหรับการให้บริการเพียงพอหรือไม่จากตารางการจัดการทรัพยากรที่แบ็คเอนด์สร้างขึ้น

### 3.1.1 การทำงานของผู้ให้บริการทรัพยากรจีพียู (Server) และผู้ใช้บริการทรัพยากรจีพียู (Client)

จากการอธิบายการทำงานภาพรวมสถาปัตยกรรมการทำงานของระบบเวอร์ชวลคูด้าสามารถจำแนกการทำงานของระบบเวอร์ชวลคูด้าออกเป็น 2 ส่วนการทำงานดังแสดงในภาพที่ 3.3 คือ 1) ส่วนการทำงานของผู้ให้บริการทรัพยากรจีพียู (Server) ซึ่งมีส่วนประกอบในการทำงานของ แบ็คเอนด์ (Back-end) และ Library Wrapper 2) ส่วนการทำงานของผู้ใช้ทรัพยากรจีพียู (Client) มีส่วนประกอบในการทำงานของ Interposer Library CUDA API และ ฟรอนท์เอนด์ (Front-end) โดยมีรายละเอียดในการทำงานของ Server และ Client แสดงในภาพที่ 3.3 ดังต่อไปนี้

ภาพที่ 3.3 การทำงานระหว่างเครื่องผู้ใช้บริการจีพียู (Client) กับเครื่องที่ให้บริการจีพียู (Server)





3.1.1.1 การทำงานของผู้ให้บริการทรัพยากรจีพียู (Server) มีรายละเอียดหน้าที่ในการทำงานดังนี้

1. แบคเอนด์ (Back-end) ทำหน้าที่เป็นตัวกลางผสานงานและจัดการ การเข้าถึง จีพียูของแต่ละคำสั่งตามคิวดำแอปพลิเคชันที่ถูกส่งมาจากฟรอนท์เอนด์ของเครื่องผู้ให้บริการ จีพียู โดยนำคำสั่งที่ได้รับไปประมวลผลและเรียกใช้งานจีพียูให้ประมวล หลังจากประมวลผลเสร็จสิ้น แบคเอนด์จะนำผลที่ได้ส่งกลับไปยังฟรอนท์เอนด์ของเครื่องที่ขอใช้บริการจีพียูของแต่ละเครื่อง

สำหรับการเรียกใช้งานคิวดำแอปพลิเคชันของแบคเอนด์นั้นจำเป็นต้องมีการสร้าง แอปพลิเคชันสำหรับเรียกใช้งานไว้ที่ผู้ให้บริการจีพียูสำหรับการเรียกใช้งานบนจีพียู โดยมี รายละเอียดดังต่อไปนี้

ก. เริ่มจากนำโปรแกรมสำหรับประมวลผลการปฏิบัติงานบนจีพียูแล้วนำมา แปลโปรแกรม (Compile) ด้วย NVCC เพื่อให้ได้ Cubin File โดยใช้ตัวเลือกให้ออกมาเป็น PTX Code

ข. นำ Cubin File ดังกล่าวไปวางไว้บนเครื่องของผู้ให้บริการทรัพยากรจีพียู สำหรับแบคเอนด์เรียกใช้งานโดยผ่าน Library Wrapper

2. Library Wrapper ทำหน้าที่แปลงคำสั่งจากฟรอนท์เอนด์ที่ต้องการเรียกใช้งาน จีพียูด้วยคำสั่งของ CUDA Driver API ผ่าน NVIDIA CUDA Library ที่ถูกติดตั้งไว้บนเครื่องผู้ ให้บริการจีพียูให้อยู่ในรูปของคำสั่ง CUDA Driver API ตามปกติ สำหรับใช้ในการติดต่อสื่อสารกับ อุปกรณ์จีพียู เพื่อผสานการทำงานและประมวลผลการทำงานตามแต่ละคำสั่งที่ได้รับจาก ฟรอนท์เอนด์

3.1.1.2 การทำงานของผู้ใช้ทรัพยากรจีพียู (Client) มีรายละเอียดในการทำงาน ดังนี้

1. การทำงานของผู้ใช้ทรัพยากรจีพียูเริ่มต้นเมื่อมีการเรียกใช้งานคิวดำ แอปพลิเคชันโดยส่วนที่เป็นคิวดำโค้ดที่ต้องการเข้าถึงจีพียูเพื่อใช้จีพียูในการประมวลผลการ ทำงาน ซึ่งโดยปกติแล้วคำสั่ง CUDA Driver API จะถูกเรียกใช้งานผ่าน NVIDIA CUDA Library แต่ใน กรณีที่ผู้ใช้ทรัพยากรจีพียูไม่มีอุปกรณ์จีพียูเป็นของตนเองจึงจำเป็นต้องเข้าถึงจีพียู โดยผ่าน Interposer Library CUDA API สำหรับเรียกใช้งานคำสั่ง CUDA Driver API ที่อยู่บนเครื่องของผู้ ให้บริการทรัพยากรจีพียู

2. Interposer Library CUDA API ทำหน้าที่เป็นตัวกลางผสานงานจัดการ การเรียกใช้งาน NVIDIA CUDA Library จากชุดแอปพลิเคชันของผู้ขอรับบริการจีพียู โดยไลบรารีที่ทำหน้าที่เป็นตัวกลางนี้จะถูกนำไปวางไว้บนเครื่องของผู้ขอรับบริการจีพียู และเมื่อไลบรารีถูกเรียกใช้งานจะทำการเตรียมข้อมูลที่ต้องส่งไปให้แบ็คเอนด์ เช่น ตัวแปร ขนาดของข้อมูล และประเภทคำสั่งของชุด นำทุกอย่างอย่างรวมกันเป็นแพ็คเกจแล้วจึงส่งแพ็คเกจนั้น ๆ ต่อให้ฟรอนท์เอนด์ทำงานต่อไป

3. ฟรอนท์เอนด์ (Front-end) ทำหน้าที่จัดการ การติดต่อสื่อสารระหว่างเครื่องผู้ให้บริการจีพียูกับเครื่องผู้ขอรับบริการจีพียู โดยเริ่มจากการสร้างช่องทางการติดต่อด้วยซ็อกเก็ต (Socket) ผ่านโปรโตคอลที่ซีพี/ไอพี (TCP/IP) แล้วจึงจัดส่งแพ็คเกจที่ได้รับจาก Interposer Library CUDA API ส่งไปตามช่องทางที่ถูกสร้างขึ้นไปที่แบ็คเอนด์ของเครื่องผู้ให้บริการทรัพยากรจีพียู ซึ่งการเลือกใช้โปรโตคอลที่ซีพีสำหรับฟรอนท์เอนด์นั้น เนื่องจากที่ซีพีมีกลไกในการตรวจสอบการสูญหายของข้อมูลระหว่างการรับส่งข้อมูลต่างจากยูดีพี (UDP) ทำให้เชื่อมั่นได้ว่าข้อมูลที่ผ่านการรับส่งผ่านที่ซีพีจะไม่เกิดการสูญหายระหว่างทาง

สำหรับฟังก์ชันการทำงานของ CUDA Driver API ที่อยู่ใน Library Wrapper และ Interposer Library CUDA API งานวิจัยนี้ได้รวบรวมฟังก์ชันพื้นฐานที่สำคัญของ CUDA Driver API ที่นำมาใช้สำหรับการประมวลผลปฏิบัติงานของชุดแอปพลิเคชัน โดยส่วนที่เป็นฟังก์ชันสำหรับประมวลผลปฏิบัติงานบนจีพียูนั้นจะถูกแปลโปรแกรม (Compile) ให้อยู่ในรูปแบบของ Cubin File และวางอยู่ในส่วนของผู้ให้บริการทรัพยากรจีพียูสำหรับรอการถูกเรียกใช้งานจากแบ็คเอนด์ ฟังก์ชันพื้นฐานที่สำคัญของ CUDA Driver API ที่นำมาใช้ได้แก่

- |                                 |                       |
|---------------------------------|-----------------------|
| ➤ cuInit                        | ➤ cuParamSetv         |
| ➤ cuDeviceGet                   | ➤ cuParamSetSize      |
| ➤ cuCtxCreate                   | ➤ cuFuncSetBlockShape |
| ➤ cuModuleLoad                  | ➤ cuLaunchGrid        |
| ➤ cuModuleGetFunction           | ➤ cuCtxDetach         |
| ➤ cuMemAlloc                    | ➤ cuMemFree           |
| ➤ cuMemcpyHtoD/<br>cuMemcpyDtoH |                       |



โดยมีเหตุผลในการเลือกใช้ฟังก์ชันพื้นฐานที่สำคัญของ CUDA Driver API ทั้ง 14 ฟังก์ชันนี้ เนื่องจากเป็นฟังก์ชันที่สำคัญพื้นฐานสำหรับการสั่งงานให้จีพียูสามารถประมวลผลการทำงานได้ และเวลาทำงานวิจัยที่มีอยู่อย่างจำกัด

### 3.1.2 การจัดสรรการเข้าถึงทรัพยากรจีพียู

ส่วนนี้กล่าวถึงการออกแบบการจัดสรรการเข้าถึงจีพียูของผู้ให้บริการจีพียูโดยแสดงขั้นตอนการทำงานโดยรวมดังภาพที่ 3.4 , 3.5 และ 3.6 ซึ่งการออกแบบการจัดสรรทรัพยากรจีพียูของระบบเวอร์ชวลคูด้า ได้ออกแบบให้มีตารางสำหรับการจัดสรรทรัพยากรเพื่อหลีกเลี่ยงการเกิดปัญหาการติดตาย อันเนื่องมาจากเหตุการณ์ที่แบ็คเอนด์ทำการจองพื้นที่บนจีพียูทันทีหลังจากได้รับคำสั่งจากฟรอนท์เอนด์ และคำสั่งที่รับมาพร้อม ๆ กันจากหลายเวอร์ชวลแมชีนโดยไม่ใช้คำสั่งสำหรับสั่งให้จีพียูประมวลผลซึ่งอาจจะทำให้เกิดการรอคำสั่งประเภทดังกล่าวจนเกิดการติดตาย หรือเกิดการยกเลิกการประมวลผลของคูด้าแอปพลิเคชันกรณีมีพื้นที่ไม่เพียงพอต่อการประมวลผล อีกทั้งระบบเวอร์ชวลคูด้ายังได้ให้ความสำคัญกับลำดับในการเข้าถึงทรัพยากรจีพียูจากการร้องขอใช้ทรัพยากร และคำนึงถึงการเกิดปัญหาการติดตาย (Deadlock)

วิทยานิพนธ์ฉบับนี้จึงได้ออกแบบอัลกอริทึมสำหรับจัดสรรการเข้าถึงทรัพยากรจีพียูออกเป็น 3 ขั้นตอน คือ 1) การจัดสรรทรัพยากรให้เพียงพอต่อการเรียกใช้งาน 2) การจัดสิทธิในการเข้าถึงทรัพยากรจากลำดับการขอเข้าใช้งานจีพียู 3) ตรวจสอบคำสั่งการปฏิบัติงาน cuLaunchGrid กรณีเรียกใช้งานซ้ำ โดยอธิบายรายละเอียดแต่ละขั้นตอนดังนี้

#### 3.1.2.1 การจัดสรรทรัพยากรให้เพียงพอต่อการเรียกใช้งาน

การจัดสรรทรัพยากรจีพียูสำหรับผู้ขอใช้บริการทรัพยากรจีพียูนั้น วิทยานิพนธ์นี้ได้นำอัลกอริทึมของนายธนาคาร (Banker's Algorithm) เข้ามาประยุกต์ในการจัดสรรทรัพยากรเพื่อใช้ในการตรวจสอบทรัพยากรว่ามีเพียงพอต่อการให้บริการหรือไม่ อีกทั้งยังสามารถหลีกเลี่ยงปัญหาการติดตาย (Deadlock) ซึ่งการเลือกอัลกอริทึมของนายธนาคารมาใช้เนื่องจาก อัลกอริทึมของนายธนาคาร เป็นอัลกอริทึมที่ใช้หลีกเลี่ยงการเกิดปัญหาการติดตายแต่ไม่ใช่อัลกอริทึมเพื่อแก้ปัญหาการติดตายเหมือนกับ Detection Algorithm จึงเหมาะสมกับระบบเวอร์ชวลคูด้าที่จัดทำขึ้น โดยมีขั้นตอนการจัดสรรทรัพยากรดังนี้

1. เมื่อแบ็คเอนด์รับคำสั่งคูด้าแอปพลิเคชันมาจากเครื่องผู้ขอรับบริการแบ็คเอนด์ จะทำหน้าที่ตรวจสอบก่อนว่าคำสั่งคูด้า นั้นมีหน้าที่ใด กรณีเป็นคำสั่งเพื่อจองพื้นที่บนจีพียู

(cuMemAlloc) แบริคเ็นดจะนำตัวแปร ขนาดของตัวแปร และหมายเลขเครื่องผู้ขอรับบริการจีพียู ไปเก็บไว้ยังตารางการจัดสรรทรัพยากร (ดังตารางที่ 3.1) เพื่อบรรณและนำไปใช้งานต่อไป อีกทั้งนำ ข้อมูลที่ถูกส่งมาจากพรอนท์เอนดด้วยคำสั่ง cuMemcpyHtoD พักไว้บนแบริคเ็นดสำหรับรอ เรียกใช้งานในการถ่ายโอนข้อมูลไปยังจีพียูเมื่อมีการสั่งให้ปฏิบัติงานด้วยคำสั่ง cuLaunchGrid เกิดขึ้นจากการสั่งงานของพรอนท์เอนด การนำทุกคำสั่งคู่ด้าแอฟฟลิเคชันจากพรอนท์เอนดมาเก็บ ในตารางการจัดสรรทรัพยากร เนื่องจากการทำงานด้วยวิธีนี้สามารถป้องกันการเกิดปัญหาติดตาย และแอฟฟลิเคชันที่ประมวลผลไม่สำเร็จเนื่องจากพื้นที่สำหรับการใช้ประมวลผลไม่เพียงพอต่อการ ใช้งาน

2. กรณีที่แบริคเ็นดรับคำสั่งคู่ด้าแอฟฟลิเคชัน ซึ่งส่งมาจากเครื่องผู้ขอรับบริการ และเป็นคำสั่งเพื่อเรียกใช้งานในฟังก์ชันของคู่ด้าแอฟฟลิเคชัน ซึ่งคำสั่งดังกล่าวคือคำสั่ง cuLaunchGrid ในส่วนการทำงานของแบริคเ็นดจะทำการตรวจสอบก่อนว่าตัวแปรที่ฟังก์ชันต้อง นำไปใช้นั้นมีอะไรบ้างและเมื่อทราบแล้ว แบริคเ็นดจะรวมพื้นที่ทั้งหมดของตัวแปรดังกล่าวและ นำไปตรวจสอบกับพื้นที่ของจีพียูที่ยังเหลืออยู่ว่ามีขนาดเพียงพอกับเนื้อที่ทั้งหมดของตัวแปรที่จะ นำไปใช้หรือไม่ หากเพียงพอแบริคเ็นดจะจัดการเรียกการทำงานผ่าน NVIDIA CUDA Library เพื่อจองพื้นที่ให้กับตัวแปรทั้งหมดพร้อมกันทำสัญลักษณ์ให้กับตัวแปรในตารางการจัดสรร ทรัพยากรว่าตัวแปรดังกล่าวได้ถูกจองพื้นที่ไว้แล้วในหน่วยความจำของจีพียู การทำสัญลักษณ์ว่า มีการจองพื้นที่ให้กับตัวแปรไปแล้วในตารางการจัดสรรทรัพยากรนี้มีวัตถุประสงค์เพื่อ เมื่อมีการ เรียกใช้งานตัวแปรเดิมซ้ำแบริคเ็นดจะไม่นำตัวแปรดังกล่าวไปจองพื้นที่ในหน่วยความจำจีพียูซึ่ง ตัวแปรทุกตัวจะถูกยกเลิกออกจากตารางการจัดสรรทรัพยากรและหน่วยความจำจีพียูด้วยคำสั่ง cuMemFree

### 3.1.2.2 การจัดสิทธิในการเข้าถึงทรัพยากรจากลำดับการขอเข้าใช้งานจีพียู

ส่วนนี้กล่าวถึงการออกแบบการจัดการการเข้าถึงจีพียูของผู้ให้บริการจีพียู โดยการ จัดสิทธิสำหรับการเข้าถึงจีพียูจากลำดับการขอเข้าใช้งานจีพียูซึ่งการจัดการดังกล่าวนี้ จะจัดการ ในระดับคำสั่งของคู่ด้า ตามการเรียกใช้ของผู้ขอรับบริการจีพียู ด้วยวิธีมาก่อนจะได้รับสิทธิในการ ประมวลผลก่อน (FIFO) ซึ่งการจัดลำดับความสำคัญนี้จะให้ความสำคัญที่คำสั่ง cuLaunchGrid โดยมีรายละเอียดตามขั้นตอนดังนี้

1. เมื่อผู้ให้บริการจีพียูได้รับคำสั่ง cuLaunchGrid ในส่วนการทำงานของ แบริคเ็นดจะทำหน้าที่ให้ลำดับความสำคัญในการประมวลผลและหมายเลขเครื่องผู้ขอรับบริการจีพียู

2. หลังจากให้ลำดับความสำคัญในการประมวลผลแล้ว แบ็คเอนด์จะทำการสำรวจคำสั่งการให้ปฏิบัติงานของ cuLaunchGrid จากตารางการจัดสรรทรัพยากรและสั่งให้ cuLaunchGrid ที่มีลำดับการปฏิบัติงานสูงที่สุดปฏิบัติงาน โดยต้องผ่านการตรวจสอบใน 3.1.2.1) เป็นที่เรียบร้อยแล้ว

3. เมื่อการประมวลผลตามคำสั่งการปฏิบัติงานของ cuLaunchGrid เสร็จสิ้นหรือแบ็คเอนด์ได้รับคำสั่งให้ยกเลิกการจองพื้นที่ของตัวแปรในจีพียู แบ็คเอนด์จำเป็นต้องสำรวจตารางการจัดสรรทรัพยากรโดยทำตาม ขั้นตอนที่ 2) สำหรับการจัดสิทธิในการเข้าถึงทรัพยากรจากลำดับการขอเข้าใช้งานจีพียู จนกระทั่งไม่มีคำสั่ง cuLaunchGrid เหลืออยู่ในตารางจัดสรรทรัพยากร

### 3.1.2.3 ตรวจสอบคำสั่งการปฏิบัติงาน cuLaunchGrid กรณีเรียกใช้งานซ้ำ

ส่วนนี้กล่าวถึงการตรวจสอบคำสั่งปฏิบัติงาน cuLaunchGrid กรณีเรียกใช้งานซ้ำจากผู้ขอรับบริการจีพียู ซึ่งส่งงานผ่านมาทางฟรอนท์เอนด์ และเมื่อคำสั่งดังกล่าวมาถึงแบ็คเอนด์ แบ็คเอนด์มีหน้าที่ต้องตรวจสอบ ตามรายละเอียดดังนี้

1. ตรวจสอบข้อมูลตัวแปรแต่ละตัวที่คำสั่ง cuLaunchGrid ต้องการเรียกใช้งานเพื่อนำไปประมวลผลว่ามีการจองพื้นที่แล้วบนจีพียูแล้วหรือไม่ กรณีที่มีการจองพื้นที่แล้วแบ็คเอนด์จะไม่ทำการจองพื้นที่ซ้ำอีกครั้ง

2. ตรวจสอบข้อมูลตัวแปรที่เป็น Input แต่ละตัวที่คำสั่ง cuLaunchGrid ต้องการเรียกใช้งานว่ามีการคัดลอกข้อมูลไปยังจีพียูแล้วหรือไม่ กรณีที่มีการคัดลอกแล้ว แบ็คเอนด์จะไม่ทำการคัดลอกซ้ำลงไปอีก

เมื่อทำการตรวจสอบตามข้อ 1. และ 2. เสร็จเรียบร้อยแล้วและผลปรากฏว่าเป็นไปตามเงื่อนไข คือตัวแปรแต่ละตัวได้ถูกจองพื้นที่บนจีพียูแล้ว และตัวแปรที่เป็น Input ได้ถูกคัดลอกข้อมูลไปยังจีพียูแล้ว นั้นแสดงว่า cuLaunchGrid เคยถูกเรียกใช้งานมาก่อนและข้อมูลต่าง ๆ ที่อยู่บนจีพียูยังไม่เคยถูกล้างออกไป ซึ่งจะเป็นเหตุผลให้แบ็คเอนด์เรียกใช้งาน cuLaunchGrid ได้ทันทีโดยไม่ต้องไปเริ่มทำคำสั่ง cuInit ถึง cuMemcpyHtoD ใหม่อีกครั้ง จึงเป็นการลดเวลาในการทำงานในขั้นตอนดังกล่าวสำหรับแบ็คเอนด์ หรือแม้แต่ฟรอนท์เอนด์หากต้องการเรียกใช้งาน cuLaunchGrid เดิมซ้ำ ก็ไม่จำเป็นต้องส่งงาน cuInit ถึง cuMemcpyHtoD มายังแบ็คเอนด์อีก ซึ่งจะเป็นการลดเวลาในการถ่ายโอนข้อมูลมายังแบ็คเอนด์เป็นอย่างมาก

### 3.1.2.4 ตัวอย่างการจัดสรรการเข้าถึงทรัพยากรจีพียู

ตารางที่ 3.1 ตัวอย่างตารางการจัดสรรทรัพยากรตามคำสั่งคู่ด้าแอปพลิเคชัน

คำสั่ง	ชื่อตัวแปร	ขนาดข้อมูล	เครื่องผู้ขอใช้ จีพียู	ลำดับการ ทำงาน	พื้นที่ถูก จอง
cuMemcpyHtoD	A	20MB	1	-	ไม่
cuMemcpyHtoD	A	30MB	2	-	ใช่
cuMemcpyHtoD	B	20MB	2	-	ใช่
cuMemcpyHtoD	B	50MB	1	-	ไม่
cuLaunchGrid	-	0	1	1	-
cuLaunchGrid	-	0	2	2	-

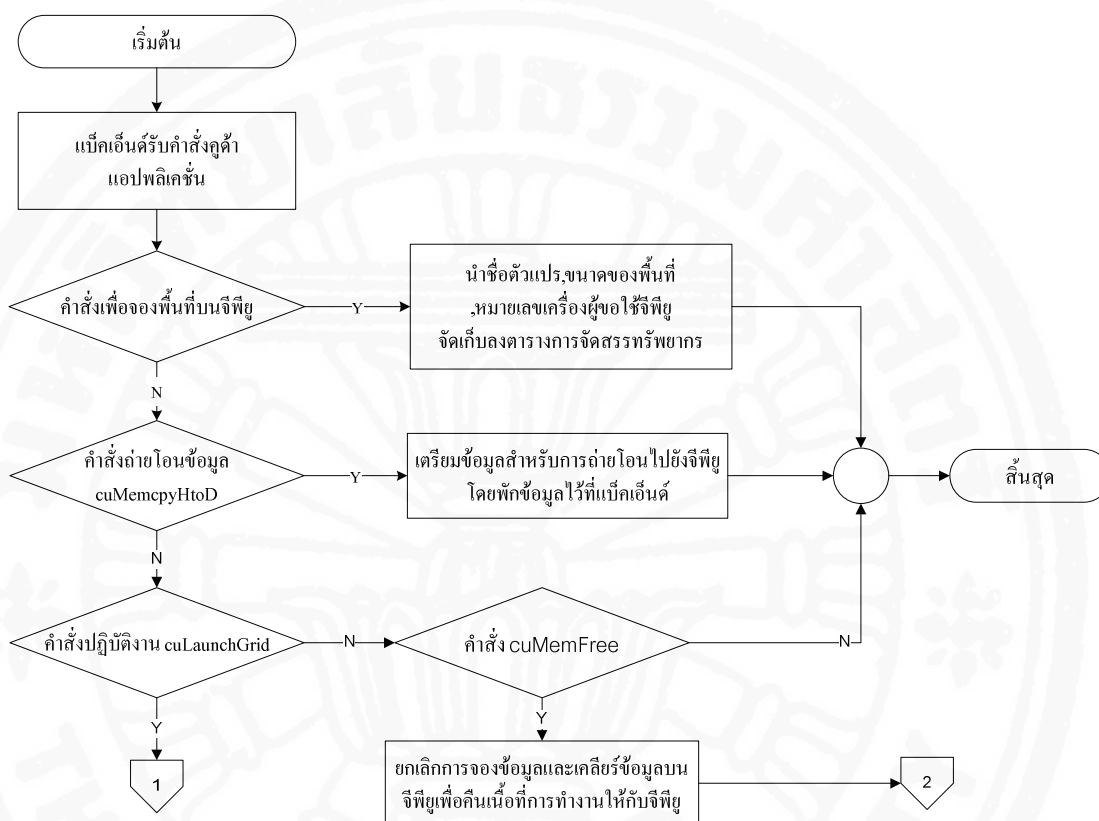
จากตารางที่ 3.1 สามารถอธิบายตัวอย่างการทำงานจัดสรรการเข้าถึงทรัพยากรจีพียูได้ดังนี้ คือตารางที่ 3.1 เป็นตารางการจัดสรรทรัพยากรจีพียูที่ฝังอยู่บนแบคเอนด์ของเครื่องผู้ให้บริการจีพียู โดยมีเหตุการณ์สมมุติดังนี้ เมื่อมีการเรียกใช้งานคำสั่งคู่ด้าฟังก์ชันในการจองพื้นที่สำหรับตัวแปร A จากเครื่องหมายเลข 1 ขนาดของข้อมูล 20 MB และมีประเภทการจองพื้นที่เป็น cuMemcpyHtoD คือขอจองพื้นที่เพื่อคัดลอกข้อมูลจากจีพียูไปยังจีพียู จากฟรอนท์เอนด์ของเครื่องเวอร์ชวลแมชีน แบคเอนด์จะทำการตรวจสอบคำสั่งที่ส่งมาก่อนว่าเป็นคำสั่งปฏิบัติงาน cuLaunchGrid หรือไม่ กรณีตรวจสอบแล้วไม่ใช่จะนำคำสั่ง cuMemcpyHtoD ชื่อตัวแปร A หมายเลขเครื่องและขนาดของข้อมูลไปเก็บไว้ในตารางการจัดสรร ดังแสดงในตารางที่ 3.1 และทำแบบเดียวกันนี้ไปเรื่อย ๆ จนกว่าคำสั่งที่ส่งมาจากฟรอนท์เอนด์จะเป็นคำสั่ง cuLaunchGrid ซึ่งในตารางที่ 3.1 ได้เก็บไว้ 4 คำคือ A,B ของเครื่องที่ 1 และ 2 กรณีตรวจสอบแล้วเป็นคำสั่ง cuLaunchGrid จริง แบคเอนด์จะยังไม่ทำการรันคู่ด้าแอปพลิเคชันแต่จะตรวจสอบพื้นที่ก่อนว่ามีเพียงพอสำหรับการประมวลผลหรือไม่ โดยจะรวมค่าตัวแปรทั้งหมดที่ต้องใช้สำหรับ cuLaunchGrid นั้น ๆ กรณีมีพื้นที่เพียงพอแบคเอนด์จะทำการประมวลผลคู่ด้าแอปพลิเคชันนั้น ๆ ทันที แต่ถ้าหากมีพื้นที่ไม่เพียงพอจะนำคำสั่ง cuLaunchGrid ไปเก็บไว้ในตารางการจัดสรรทรัพยากรจีพียู และให้ลำดับในการรอประมวลผลจนกว่ามีพื้นที่ว่างพอให้ประมวลผล จากตัวอย่างกำหนดให้พื้นที่ในการประมวลผลจีพียูเหลือเพียง 50 MB เพราะฉะนั้นจะเห็นได้ว่าเมื่อมีคำสั่ง

cuLaunchGrid ที่ส่งมาจากฟรอนท์เอนด์ จากตารางที่ 3.1 แถวที่ 5 ซึ่งเป็นของเครื่องหมายเลขที่ 1 พบว่าเมื่อแบคเอนด์ทำการตรวจสอบพื้นที่ต้องใช้ในการประมวลผลสำหรับเครื่องหมายเลข 1 นั้น พื้นที่ของตัวแปร A บวกกับพื้นที่ของ ตัวแปร B สำหรับใช้ในการประมวลผลคู่ด้าแอฟพลีเคชัน สำหรับคำสั่ง cuLaunchGrid นั้นมีพื้นที่มากเกินไปกว่าที่จีพียูจะสามารถให้บริการได้ แบคเอนด์จึง นำคำสั่ง cuLaunchGrid ของเครื่องหมายเลข 1 ไปเก็บไว้ในตารางการจัดสรรทรัพยากรจีพียู พร้อมกับบรรทัดของสถิติที่ได้รับสำหรับการทำงานเมื่อจีพียูมีพื้นที่เพียงพอ และต่อมา แบคเอนด์ได้รับคำสั่ง cuLaunchGrid ของเครื่องหมายเลข 2 ปรากฏว่าจากการตรวจสอบพบว่า มีพื้นที่เหลือเพียงพอในการประมวลผลจึงปล่อยให้ cuLaunchGrid ของเครื่องหมายเลข 2 ทำงาน และเมื่อมีพื้นที่ว่างจากการที่คู่ด้าแอฟพลีเคชันก่อนหน้านี้ประมวลผลเสร็จ cuLaunchGrid ของ เครื่องหมายเลข 1 จะถูกทำงานทันทีเช่นกัน (ดังภาพที่ 3.4 , 3.5 และ 3.6 )

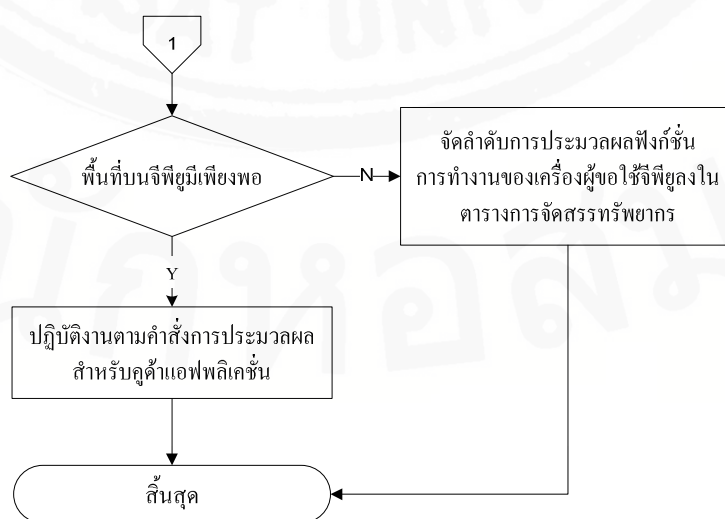
จากตารางที่ 3.1 สามารถอธิบายตัวอย่างตรวจสอบคำสั่งการปฏิบัติงาน cuLaunchGrid กรณีเรียกใช้งานซ้ำได้ดังนี้ คือ กรณีที่เครื่องผู้ขอใช้จีพียูหมายเลข 2 เข้าไปส่งงาน จีพียูตามคำสั่ง cuLaunchGrid แสดงว่ามีพื้นที่บนจีพียูเพียงพอสำหรับให้บริการ ทำให้ตัวแปร A และ B สามารถจองพื้นที่ และคัดลอกข้อมูลลงไปที่จีพียูได้ ขณะที่ตัวแปร A และ B จองพื้นที่ และ คัดลอกข้อมูลลงไปที่จีพียูได้นั้น แบคเอนด์จะทำสัญลักษณ์การจองพื้นที่ไว้บนตารางการจัดสรรว่า ตัวแปรดังกล่าวเคยจองพื้นที่ และคัดลอกข้อมูลลงไปที่จีพียูแล้ว โดยจะถูกยกเลิกข้อมูลก็ต่อเมื่อ พบคำสั่ง cuMemFree ทั้งนี้ทำให้เมื่อเครื่องผู้ขอใช้จีพียูหมายเลข 2 สั่ง cuLaunchGrid อีกครั้ง แบคเอนด์จะไปตรวจสอบข้อมูลตัวแปร A และ B ที่ตารางการจัดสรร ซึ่งจะพบว่าข้อมูลทั้งสองตัว แปรเคยถูกจองพื้นที่และมีข้อมูลอยู่บนจีพียูแล้ว ซึ่งจะทำให้แบคเอนด์สามารถสั่ง cuLaunchGrid ได้ทันทีโดยไม่ต้องจองพื้นที่และคัดลอกข้อมูลลงไปที่จีพียูซ้ำอีกครั้ง (ดังภาพที่ 3.7)



ภาพที่ 3.4 ภาพรวมการจัดการจัดการเข้าถึงทรัพยากรจีพียูของระบบเวอร์ชวลคู่สำหรับขั้นตอน  
การตรวจสอบแต่ละประเภทคำสั่ง

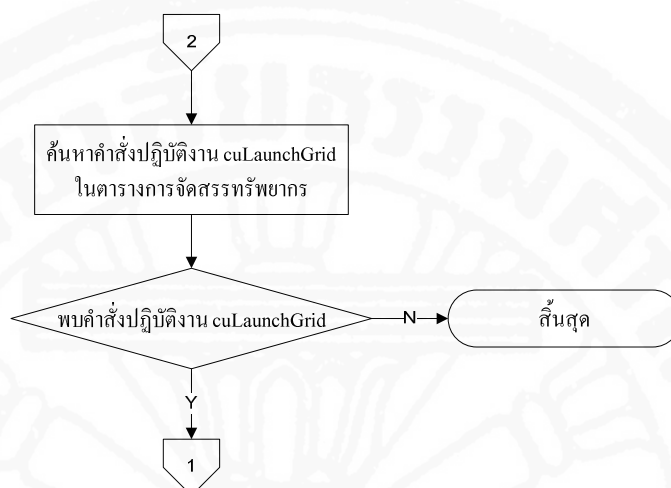


ภาพที่ 3.5 ภาพรวมการจัดการจัดการเข้าถึงทรัพยากรจีพียูของระบบเวอร์ชวลคู่สำหรับขั้นตอน  
การตรวจสอบพื้นที่บนจีพียูเพื่อประมวลผล

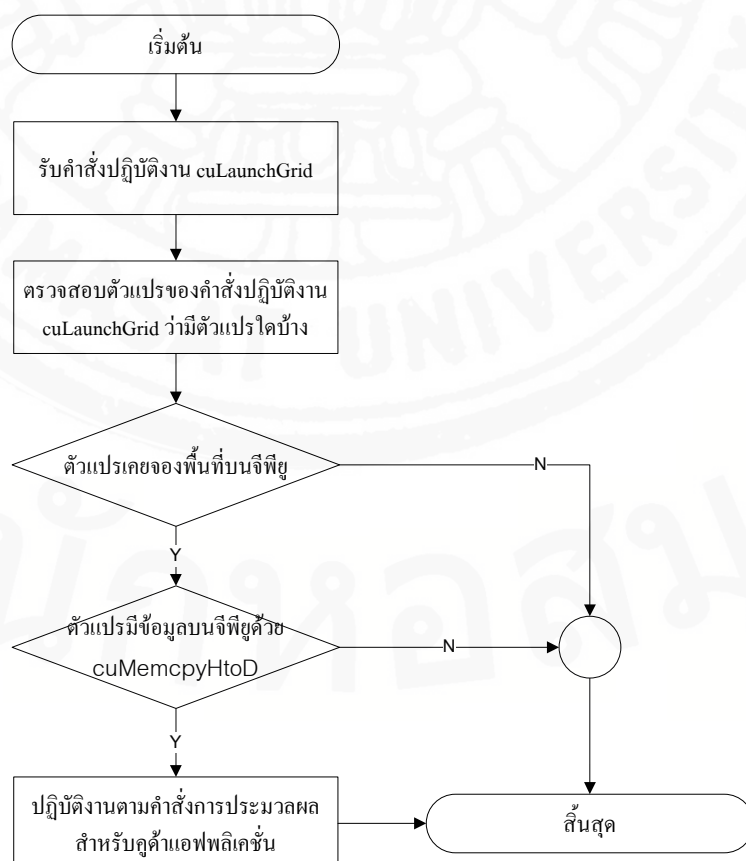




ภาพที่ 3.6 ภาพรวมการจัดการเข้าถึงทรัพยากรจีพียูของระบบเวอร์ชวลคูด้า สำหรับขั้นตอน  
การตรวจสอบคำสั่งปฏิบัติงานในตารางจัดสรร



ภาพที่ 3.7 ภาพตรวจสอบคำสั่งการปฏิบัติงาน cuLaunchGrid กรณีเรียกใช้งานซ้ำ  
ของระบบเวอร์ชวลคูด้า



### 3.1.3 ตัวอย่างโปรแกรมสำหรับเรียกใช้งานจีพียูด้วยคูด้าแอปพลิเคชันโดยผ่านระบบเวอร์ชวลคูด้า

ระบบเวอร์ชวลคูด้าที่ได้พัฒนาขึ้นมานั้น จะทำงานโดยเข้าถึงจีพียูด้วยคำสั่งของคูด้าแบบ CUDA driver API โดยจะใช้ฟังก์ชันพื้นฐานที่ได้กล่าวไว้ในข้อ 3.1.1.2 โดยแสดงการเรียกใช้งาน ดังนี้

```
#include "functioncuda_client.h"

void matrixMul(int argc, char** argv)
{
    int sockfd=connectServer();

    // set seed for rand()
    srand(2006);

    unsigned int uiWA, uiHA, uiWB, uiHB, uiWC, uiHC;
    int iSizeMultiple = 1;
    uiWA = WA * iSizeMultiple;
    uiHA = HA * iSizeMultiple;
    uiWB = WB * iSizeMultiple;
    uiHB = HB * iSizeMultiple;
    uiWC = WC * iSizeMultiple;
    uiHC = HC * iSizeMultiple;

    unsigned int size_A = uiWA * uiHA;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float* h_A = (float*)malloc(mem_size_A);
    unsigned int size_B = uiWB * uiHB;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float* h_B = (float*)malloc(mem_size_B);

    // initialize host memory
    randomInit(h_A, size_A);
```

```
randomInit(h_B, size_B);

cuInitialize(sockfd);
cuLoadFunction("matrixMul",sockfd);

unsigned int size_C = uiWC * uiHC;
unsigned int mem_size_C = sizeof(float) * size_C;

cudaMalloc((void**) &d_C, mem_size_C);*/
cuMemAllocDevice(mem_size_C,"d_C",sockfd);
cuMemAllocDevice(mem_size_A,"d_A",sockfd);
cuMemAllocDevice(mem_size_B,"d_B",sockfd);

cuMemcpy("d_A",h_A,mem_size_A,HtoD,sockfd);
cuMemcpy("d_B",h_B,mem_size_B,HtoD,sockfd);

cudaCallFunction(sockfd,BLOCK_SIZE,BLOCK_SIZE,1,GridX,GridY);
cuMemcpy("d_C",h_C,mem_size_C,DtoH,sockfd));

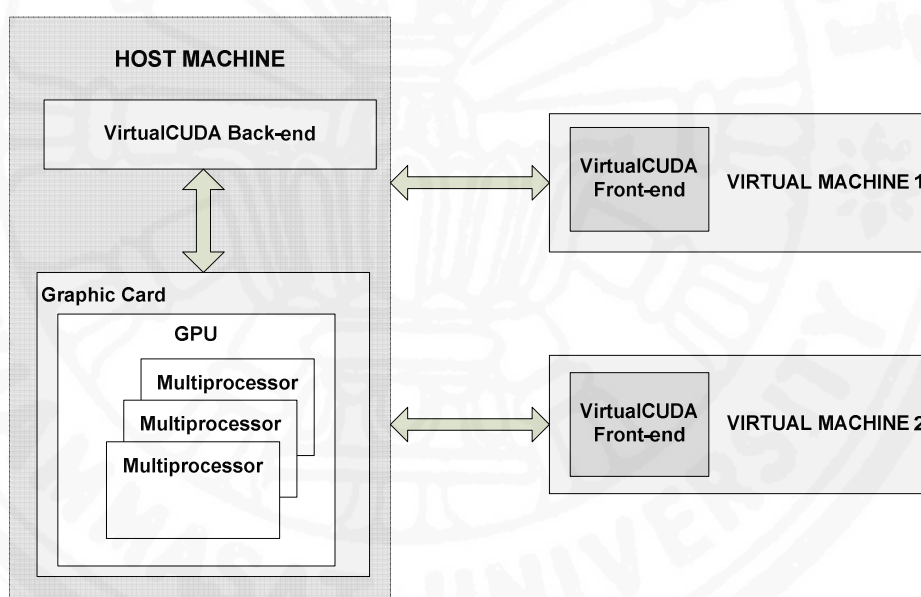
free(h_A);
free(h_B);
free(h_C);
cudaMemFree("d_A",sockfd);
cudaMemFree("d_B",sockfd);
cudaMemFree("d_C",sockfd);
}
```

### 3.2 วิธีการทดลอง

#### 3.2.1 เครื่องมือที่ใช้ในการทดลอง

ในการทดลองนี้เรากำหนดให้มีเวอร์ชวลแมชชีนสองเครื่องสามารถเรียกใช้งานทรัพยากรจ็ฟิบบนเครื่องของโฮสแมชชีน โดยผ่านระบบเวอร์ชวลคูด้า ดังภาพที่ 3.8 และสองเวอร์ชวลแมชชีน ดังกล่าวจะถูกรันอยู่บนเครื่องโฮสเครื่องเดียวกัน ซึ่งแสดงรายละเอียดของ โฮสแมชชีน เวอร์ชวลแมชชีน และ Graphic Card ดังต่อไปนี้

ภาพที่ 3.8 การทำงานของโฮสแมชชีนและสองเวอร์ชวลแมชชีน



1. โฮสแมชชีนสำหรับแบ็คเอนด์ มีรายละเอียดดังนี้
  - Architecture : x86\_64
  - Processor : Intel(R) Xeon(R) CPU E5530 @ 2.40GHz
  - CPU : 4
  - Memory capacity : 8GB
  - OS : Ubuntu 10.10
  - Kernel : 2.6.35-24-server
2. เวอร์ชวลแมชชีนแบบ KVM เครื่องที่ 1 มีรายละเอียดดังนี้

- Architecture : x86\_64
  - Processor : QEMU Virtual CPU version 0.12.5
  - Memory capacity : 1GB
  - OS : Ubuntu 9.04
  - Kernel: 2.6.28-13-generic
3. เวอร์ชวลแมชีนแบบ KVM เครื่องที่ 2 มีรายละเอียดดังนี้
- Architecture : x86\_64
  - Processor : QEMU Virtual CPU version 0.12.5
  - Memory capacity : 1GB
  - OS : Ubuntu 9.04
  - Kernel : 2.6.28-13-generic
4. ไฮเปอร์ไวเซอร์ kvm-88
5. Graphic Card
- รุ่น : NVIDIA GeForce 8800GTS
  - CUDA Driver Version : 3.2
  - CUDA Cores : 96
  - Core Clock (MHz) : 500
  - Shader Clock (MHz) : 1200
  - Memory Clock (MHz) : 800
  - Memory Amount : 640MB
  - Memory Interface : 320-bit
  - Memory Bandwidth (GB/sec) : 64

### 3.2.2 การออกแบบการทดลอง

ในส่วนนี้อธิบายถึงการออกแบบการทดลองสำหรับการขอรับบริการใช้งานทรัพยากรจีพียูของเวอร์ชวลแมชีนโดยแบ่งการทดลองออกเป็น 3 แบบคือ 1) การทดลองประมวลผลผ่านซีพียูของเวอร์ชวลแมชีนเปรียบเทียบกับประมวลผลผ่านเวอร์ชวลคูด้า 2) การทดลองเพื่อวัด

ระยะเวลาที่เรียกใช้งานจีพียูผ่านระบบเวอร์ชวลคูด้า 3) การทดลองการใช้งานจีพียูร่วมกันของเวอร์ชวลแมชชีนผ่านระบบเวอร์ชวลคูด้า โดยมีรายละเอียดในการทดลองแต่ละแบบดังต่อไปนี้

3.2.2.1 แบบที่ 1 การทดลองประมวลผลผ่านจีพียูของเวอร์ชวลแมชชีนเปรียบเทียบกับ การประมวลผลผ่านเวอร์ชวลคูด้า มีรายละเอียดดังต่อไปนี้

• วิธีการทดลอง

1. ทำการรันแอปพลิเคชันบนเครื่องเวอร์ชวลแมชชีนโดยที่แอปพลิเคชันดังกล่าวประมวลผลบนจีพียู จากนั้นทำการจับเวลาทั้งหมดที่ใช้ไปสำหรับการประมวลผล
2. นำเอาแอปพลิเคชันเดียวกันที่เคยประมวลผลบนจีพียูมาดัดแปลงให้เป็นคูด้าแอปพลิเคชันเพื่อให้สามารถประมวลผลบนจีพียูได้ โดยการประมวลผลของคูด้าแอปพลิเคชันดังกล่าวจำเป็นต้องประมวลผลผ่านระบบเวอร์ชวลคูด้า จากนั้นทำการจับเวลาทั้งหมดที่ใช้ไปสำหรับการประมวลผล

3. นำเวลาที่ได้จากการประมวลผลทั้งในข้อ 1 และข้อ 2 มาเปรียบเทียบกัน

• เป้าหมายการทดลอง

1. เพื่อวัดประสิทธิภาพในการประมวลผลและการใช้ประโยชน์ (Utilization) ที่เพิ่มมากขึ้น เมื่อนำจีพียูเข้ามาช่วยในการประมวลผล
2. เพื่อแสดงให้เห็นความสามารถในการทำงานสำหรับการเข้าถึงจีพียูของระบบเวอร์ชวลคูด้าจากเวอร์ชวลแมชชีน

• สมมติฐานการทดลอง

การประมวลผลผ่านคูด้าแอปพลิเคชันเพื่อเรียกใช้งานจีพียูผ่านเวอร์ชวลคูด้า การประมวลผลมีประสิทธิภาพที่ดีกว่าการประมวลผลผ่านจีพียูของเวอร์ชวลแมชชีน

3.2.2.2 แบบที่ 2 การทดลองเพื่อวัดระยะเวลาที่เรียกใช้งานจีพียูผ่านระบบเวอร์ชวลคูด้า มีรายละเอียดดังต่อไปนี้

• วิธีการทดลอง

1. รันคูด้าแอปพลิเคชันสำหรับเรียกใช้งานจีพียูบนเวอร์ชวลแมชชีนผ่านระบบเวอร์ชวลคูด้า



2. วัดระยะเวลาการทำงานของแบ็คเอนด์ และฟรอนท์เอนด์ ตั้งแต่ Initialization (คำสั่ง cuInit) ถึง Function Call (คำสั่ง cuLaunchGrid) จากนั้นทำการเปรียบเทียบระยะเวลาที่ได้ดังกล่าวระหว่างแบ็คเอนด์ กับฟรอนท์เอนด์

3. วัดระยะเวลาการทำงานของฟังก์ชันการคัดลอกข้อมูลแบบ Host to Device (จากจีพียูไปจีพียู) ของแบ็คเอนด์ และฟรอนท์เอนด์ จากนั้นทำการเปรียบเทียบระยะเวลาที่ได้ดังกล่าวระหว่างแบ็คเอนด์ กับฟรอนท์เอนด์

4. วัดระยะเวลาการทำงานของฟังก์ชันการคัดลอกข้อมูลแบบ Device to Host (จากจีพียูไปซีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ จากนั้นทำการเปรียบเทียบระยะเวลาที่ได้ดังกล่าวระหว่างแบ็คเอนด์ กับฟรอนท์เอนด์

#### • เป้าหมายการทดลอง

1. เพื่อวัดผลและวิเคราะห์โอเวอร์เฮดการประมวลผลของชุดแอปพลิเคชันสำหรับแบ็คเอนด์ และฟรอนท์เอนด์กรณีเรียกใช้งานจีพียูผ่านระบบ เวอร์ชวลคูด้า

2. เพื่อวัดผลและวิเคราะห์โอเวอร์เฮดการถ่ายโอนข้อมูลระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ กรณีเรียกใช้งานจีพียูผ่านระบบเวอร์ชวลคูด้า

#### • สมมติฐานการทดลอง

1. ระยะเวลาการประมวลผลของแบ็คเอนด์ให้ผลที่ดีกว่าการประมวลผลของฟรอนท์เอนด์

2. การถ่ายโอนข้อมูลไปมาระหว่างแบ็คเอนด์ และฟรอนท์เอนด์มีผลทำให้ระยะเวลาในการประมวลผลของชุดแอปพลิเคชันบนเวอร์ชวลแมชชีนเพิ่มมากขึ้นตามขนาดของข้อมูลที่ถูกถ่ายโอน

3.2.2.3 แบบที่ 3 การทดลองการใช้งานจีพียูร่วมกันของเวอร์ชวลแมชชีนผ่านระบบเวอร์ชวลคูด้า มีรายละเอียดดังต่อไปนี้

#### • วิธีการทดลอง

1. ทำการรันชุดแอปพลิเคชันจากเครื่องเวอร์ชวลแมชชีนผ่านระบบเวอร์ชวลคูด้า เพื่อร้องขอใช้บริการจีพียูมายังเครื่องผู้ให้บริการ กรณีเวอร์ชวลแมชชีนมากกว่า 1 เครื่อง แต่แบ็คเอนด์สามารถขอใช้บริการจีพียูได้เพียงครั้งละ 1 เวอร์ชวลแมชชีน โดยที่แบ็คเอนด์ไม่แบ่งเทรดการทำงานให้กับเวอร์ชวลแมชชีนแต่ละเครื่องที่เข้ามาขอใช้บริการจีพียูทำให้เวอร์ชวลแมชชีนต้องต่อคิวเพื่อเข้าใช้งานจีพียู

2. เพิ่มประสิทธิภาพแกระบบเวอร์ชวลคู่ด้าในการให้บริการจีพียู สำหรับการรันคู่ด้าแอปพลิเคชันบนเครื่องเวอร์ชวลแมชชีน ด้วยการเพิ่มความสามารถให้ กับแบ็คเอนด์ในฝั่งของผู้ให้บริการจีพียูให้สามารถรองรับการทำงานของเวอร์ชวลแมชชีน มากกว่า 1 เครื่อง กรณีขอเข้าใช้งานจีพียูพร้อมกันเพื่อประโยชน์ในการใช้ทรัพยากรจีพียูแบบเต็มประสิทธิภาพมากที่สุด

3. กำหนดให้คู่ด้าแอปพลิเคชันที่ใช้ในการประมวลผลทั้งในข้อ 1 และข้อ 2 เป็นคู่ด้าแอปพลิเคชันเดียวกัน

• เป้าหมายการทดลอง

1. เพื่อวัดประสิทธิภาพและเปรียบเทียบการทำงานในข้อ 1 และข้อ 2 ว่ามีการใช้ประโยชน์ (Utilization) สำหรับการเรียกใช้งานจีพียูกรณีใช้งานร่วมกันเพิ่มขึ้นหรือไม่

2. เพื่อวัดผลและวิเคราะห์โอเวอร์เฮดสำหรับเวลาที่ใช้ไปในการประมวลผลของคู่ด้าแอปพลิเคชัน กรณีที่ขอใช้ทรัพยากรจีพียูร่วม

• สมมติฐานการทดลอง

1. เวอร์ชวลคู่ด้าสามารถรองรับเวอร์ชวลแมชชีนกรณีเรียกใช้งานจีพียูพร้อมกันสองเครื่องได้

2. การเข้าใช้งานจีพียูพร้อมกันของสองเวอร์ชวลแมชชีนได้โดยผ่านเวอร์ชวลคู่ด้ามีประสิทธิภาพในการทำงานที่ดีกว่าการเข้าใช้งานจีพียูโดยผ่านเวอร์ชวลคู่ด้าครั้งละหนึ่งเวอร์ชวลแมชชีน

### 3.2.3 การวิเคราะห์ข้อมูลและการวัดผล

1. วิเคราะห์ประสิทธิภาพเปรียบเทียบระหว่างการประมวลผลบนจีพียูและบนจีพียูผ่านเวอร์ชวลคู่ด้าของเวอร์ชวลแมชชีน

2. วัดผลและวิเคราะห์โอเวอร์เฮดของการใช้งานคู่ด้าแอปพลิเคชัน สำหรับแบ็คเอนด์ และฟรอนท์เอนด์

3. วิเคราะห์ประสิทธิภาพเปรียบเทียบระหว่างการใช้งานจีพียูที่แบ็คเอนด์อนุญาตให้เข้าถึงได้เพียงครั้งละ 1 เวอร์ชวลแมชชีนกับการใช้งานจีพียูพร้อมกันครั้งละมากกว่า 1 เวอร์ชวลแมชชีน

4. วัดผลและวิเคราะห์โอเวอร์เฮดระหว่างการใช้งานจีพียูที่ 1 เวอร์ชวลแมชชีนกับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชชีน

## บทที่ 4

### ผลการทดลอง

ผลการทดลองของงานวิจัยนี้ ได้ทำทดลองกับการประมวลผลการคูณเมตริกซ์ แบบ Single Precision Floating Point ขนาด  $M * N$  โปรแกรมการคูณเมตริกซ์นี้ได้นำมาจาก CUDA SDK 3.2 โดยกำหนดขนาดของเมตริกซ์ และจำนวนบล็อกที่นำมาทดลองดังนี้

กำหนดขนาดของเมตริกซ์  $M * N$

เมื่อ  $M = 5 * (5 * n) * 16$  โดยที่  $n$  มีค่าตั้งแต่ 2 ถึง 11

$N = 10 * (5 * n) * 16$  โดยที่  $n$  มีค่าตั้งแต่ 2 ถึง 11

กำหนดขนาดของจำนวนบล็อก  $X * Y$

เมื่อ  $X = 5 * (5 * k)$  โดยที่  $k$  มีค่าตั้งแต่ 2 ถึง 11

$Y = 10 * (5 * k)$  โดยที่  $k$  มีค่าตั้งแต่ 2 ถึง 11

ผลการคูณเมตริกซ์ดังกล่าวได้จากสมการ  $C = A * B$  ซึ่งขนาดตัวแปร A B และ C นั้น ได้อธิบายขนาดไว้ตามตารางที่ 4.1 รวมไปถึงจำนวนบล็อกและเทร็ด / บล็อก สำหรับใช้ในการคูณเมตริกซ์แต่ละขนาด ดังตารางที่ 4.2 (ตัวอย่างบางส่วนของโปรแกรม sequential และ parallel อยู่ในภาคผนวก ก) และผลการทดลองที่ได้ในแต่ละการทดลองจะเป็นค่าเฉลี่ยจากการรัน 10 ครั้ง โดยแบ่งผลการทดลองออกเป็น 3 ลักษณะ ได้แก่ 1) ผลการทดลองประมวลผลผ่านซีพียูของเวอร์ชวลแมชชีนเปรียบเทียบกับประมวลผลด้วยจีพียูผ่านเวอร์ชวลคูด้าของเวอร์ชวลแมชชีน 2) ผลการทดลองแสดงรายละเอียดระยะเวลาที่เรียกใช้งานจีพียูผ่านเวอร์ชวลคูด้า 3) ผลการทดลองการใช้งานจีพียูร่วมกันของเวอร์ชวลแมชชีนผ่านเวอร์ชวลคูด้า โดยแต่ละหัวข้อมีรายละเอียดดังต่อไปนี้

ตารางที่ 4.1

ขนาดของตัวแปรสำหรับการประมวลผลการคูณเมตริกซ์

การคูณ เมตริกซ์	ตัวแปร A		ตัวแปร B		ตัวแปร C	
	ขนาดเมตริกซ์	ขนาดข้อมูล (Byte)	ขนาดเมตริกซ์	ขนาดข้อมูล (Byte)	ขนาดเมตริกซ์	ขนาดข้อมูล (Byte)
800 x 1600	800 x 1600	5,120,000	800 x 800	2,560,000	800 x 1600	5,120,000
1200 x 2400	1200 x 2400	11,520,000	1200 x 1200	5,760,000	1200 x 2400	11,520,000
1600 x 3200	1600 x 3200	20,480,000	1600 x 1600	10,240,000	1600 x 3200	20,480,000
2000 x 4000	2000 x 4000	32,000,000	2000 x 2000	16,000,000	2000 x 4000	32,000,000
2400 x 4800	2400 x 4800	46,080,000	2400 x 2400	23,040,000	2400 x 4800	46,080,000
2800 x 5600	2800 x 5600	62,720,000	2800 x 2800	31,360,000	2800 x 5600	62,720,000
3200 x 6400	3200 x 6400	81,920,000	3200 x 3200	40,960,000	3200 x 6400	81,920,000
3600 x 7200	3600 x 7200	103,680,000	3600 x 3600	51,840,000	3600 x 7200	103,680,000
4000 x 8000	4000 x 8000	128,000,000	4000 x 4000	64,000,000	4000 x 8000	128,000,000
4400 x 8800	4400 x 8800	154,880,000	4400 x 4400	77,440,000	4400 x 8800	154,880,000

ตารางที่ 4.2

จำนวนบล็อกและเทร็ด / บล็อกสำหรับการประมวลผลการคูณเมตริกซ์

การคูณเมตริกซ์	จำนวน เทร็ด / บล็อก	จำนวน บล็อก
800 x 1600	16 x 16	50 x 100
1200 x 2400	16 x 16	75 x 150
1600 x 3200	16 x 16	100 x 200
2000 x 4000	16 x 16	125 x 250
2400 x 4800	16 x 16	150 x 300
2800 x 5600	16 x 16	175 x 350
3200 x 6400	16 x 16	200 x 400
3600 x 7200	16 x 16	225 x 450
4000 x 8000	16 x 16	250 x 500
4400 x 8800	16 x 16	275 x 550

#### 4.1 ผลการทดลอง

##### 4.1.1 ผลการทดลองประมวลผลผ่านซีพียูของเวอร์ชวลแมชีนเปรียบเทียบกับ การประมวลผล ด้วยจีพียูผ่านเวอร์ชวลคูด้าของเวอร์ชวลแมชีน

การทดลองในส่วนนี้จะวัดผลเพื่อเปรียบเทียบการทำงานของเวอร์ชวลแมชีน กรณีใช้งานซีพียูประมวลผลกับสามารถใช้จีพียูประมวลผลได้โดยผ่านเวอร์ชวลคูด้า โดยวัดผลจากการจับเวลาการประมวลผลการคูณเมตริกซ์ขนาดต่าง ๆ ซึ่งการทดลองนี้จะแสดงผลการทดลองเป็นค่า Speedup เพื่อแสดงให้เห็นถึงประสิทธิภาพในการประมวลผลที่เพิ่มมากขึ้นเมื่อใช้จีพียูเข้ามาช่วยประมวลผลในการทำงาน โดยแสดงผลการทดลองดังตารางที่ 4.1

$$\text{ผลการทดลองได้จาก Speedup} = \frac{\text{Execution Time}_{\text{VirtualMachine}}}{\text{Execution Time}_{\text{VirtualCUDA}}}$$

เมื่อ  $\text{Execution Time}_{\text{VirtualMachine}}$  คือ เวลาที่ใช้ในการประมวลผลการคูณเมตริกซ์ผ่านซีพียู ของเวอร์ชวลแมชีน

$\text{Execution Time}_{\text{VirtualCUDA}}$  คือ เวลาที่ใช้ในการประมวลผลการคูณเมตริกซ์ของเวอร์ชวลแมชีนด้วยจีพียูโดยผ่านเวอร์ชวลคูด้า

#### ตารางที่ 4.3

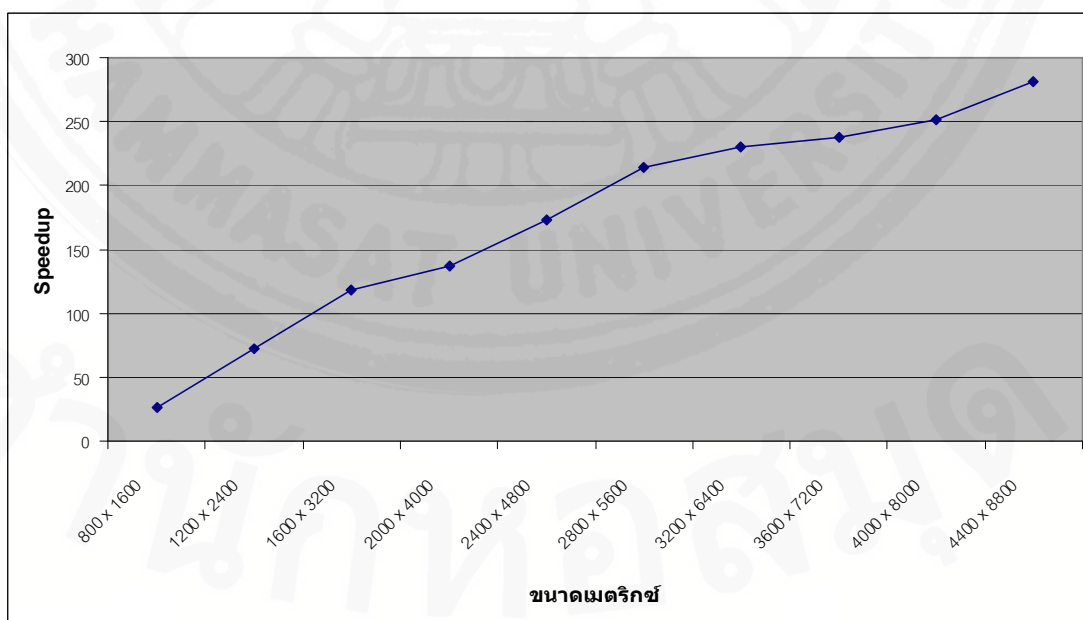
ผลการทดลองเปรียบเทียบประสิทธิภาพการทำงานระหว่าง  
การประมวลผลผ่านซีพียูของเวอร์ชวลแมชีนกับการประมวลผลผ่านเวอร์ชวลคูด้า

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B+C (Byte)	VirtualMachine (วินาที)	VirtualCUDA (วินาที)	Speedup
800 x 1600	12,800,000	12.10	0.46	26.30
1200 x 2400	28,800,000	50.26	0.69	72.84
1600 x 3200	51,200,000	131.04	1.12	117.00

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B+C (Byte)	VirtualMachine (วินาที)	VirtualCUDA (วินาที)	Speedup
2000 x 4000	80,000,000	256.08	1.86	137.68
2400 x 4800	115,200,000	481.21	2.77	173.72
2800 x 5600	156,800,000	818.67	3.84	213.20
3200 x 6400	204,800,000	1185.20	5.15	230.14
3600 x 7200	259,200,000	1583.93	6.66	237.83
4000 x 8000	320,000,000	2166.46	8.62	251.33
4400 x 8800	387,200,000	3040.83	10.81	281.30

ภาพที่ 4.1

ผลการทดลองเปรียบเทียบประสิทธิภาพการทำงานระหว่าง  
การประมวลผลผ่านซีพียูของเวอร์ชวลแมชีนกับการประมวลผลผ่านเวอร์ชวลคูด้า



จากการทดลองพบว่าเวลาที่ใช้ในการประมวลผลการคูณเมตริกซ์ในแต่ละขนาดเมตริกซ์ของทั้งซีพียูบนเวอร์ชวลแมชีน และบนจีพียูที่เรียกใช้งานผ่านเวอร์ชวลคูด้า เวลาที่ใช้ในการ

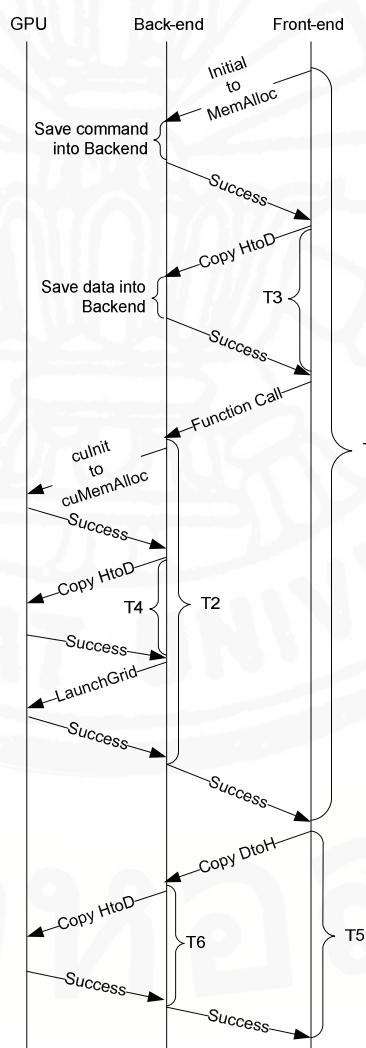


ทำงานของทั้งสองแบบจะใช้เวลาเพิ่มมากขึ้นตามลำดับของขนาดข้อมูลที่ใช้ในการประมวลผล และพบว่าเวลาที่ใช้ในการประมวลผลด้วยซีพียูบนเวอร์ชวลแมชีนจะใช้เวลามากกว่าเวลาที่ใช้ประมวลผลบนจีพียูโดยผ่านเวอร์ชวลคูด้า ซึ่งแสดงให้เห็นค่า Speedup การทำงานของจีพียูที่เพิ่มมากขึ้นเมื่อเปรียบเทียบกับซีพียู กรณีที่ขนาดของเมตริกซ์มีขนาดเพิ่มมากขึ้น ตามตารางที่ 4.3

#### 4.1.2 ผลการทดลองแสดงรายละเอียดระยะเวลาที่เรียกใช้งานจีพียูผ่านเวอร์ชวลคูด้า

ภาพที่ 4.2

การวัดผลเรียกใช้งานจีพียูของแบ็คเอนด์ และฟรอนท์เอนด์ผ่านเวอร์ชวลคูด้า



ผลการทดลองทดลองในส่วนนี้จะแสดงออกเป็น 3 ผลการทดลองคือ 1) ผลการเปรียบเทียบการทำงานระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ ตั้งแต่ Initialization (คำสั่ง `culnit`) ถึง Function Call (คำสั่ง `cuLaunchGrid`) ด้วยการคูณเมตริกซ์ 2) ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Host to Device (จากซีพียูไปจีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ 3) ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Device to Host (จากจีพียูไปซีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ โดยมีรายละเอียดผลการทดลองดังนี้

1. ผลการเปรียบเทียบการทำงานระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ ตั้งแต่ Initialization (คำสั่ง `culnit`) ถึง Function Call (คำสั่ง `cuLaunchGrid`) ด้วยการคูณเมตริกซ์ ซึ่งผลการทดลองที่ได้เกิดจากการจับเวลาการประมวลผลตั้งแต่เริ่ม Initialization การจองพื้นที่สำหรับค่าตัวแปรที่ใช้ในการประมวลผล การคัดลอกข้อมูลตัวแปร A และตัวแปร B ไปวางไว้ยังแบ็คเอนด์เพื่อรอทำการประมวลผล และการทำ Function Call ซึ่งผลที่ได้นี้จะไม่รวมการคัดลอกข้อมูลกลับมายัง ฟรอนท์เอนด์ และเวลาที่ใช้ในการยกเลิกการจองพื้นที่ในจีพียู ดังภาพที่ 4.2 โดยการทดลองนี้จะเปรียบเทียบระยะเวลาที่  $T1$  กับ  $T2$  ผลการทดลองนี้ได้แสดงไว้ในตารางที่ 4.4

เมื่อ  $T1$  คือ ระยะเวลาการทำงานของฟรอนท์เอนด์ ตั้งแต่ Initialization (คำสั่ง `culnit`) ถึง Function Call (คำสั่ง `cuLaunchGrid`)

$T2$  คือ ระยะเวลาการทำงานของแบ็คเอนด์ ตั้งแต่ Initialization (คำสั่ง `culnit`) ถึง Function Call (คำสั่ง `cuLaunchGrid`)

ตารางที่ 4.4

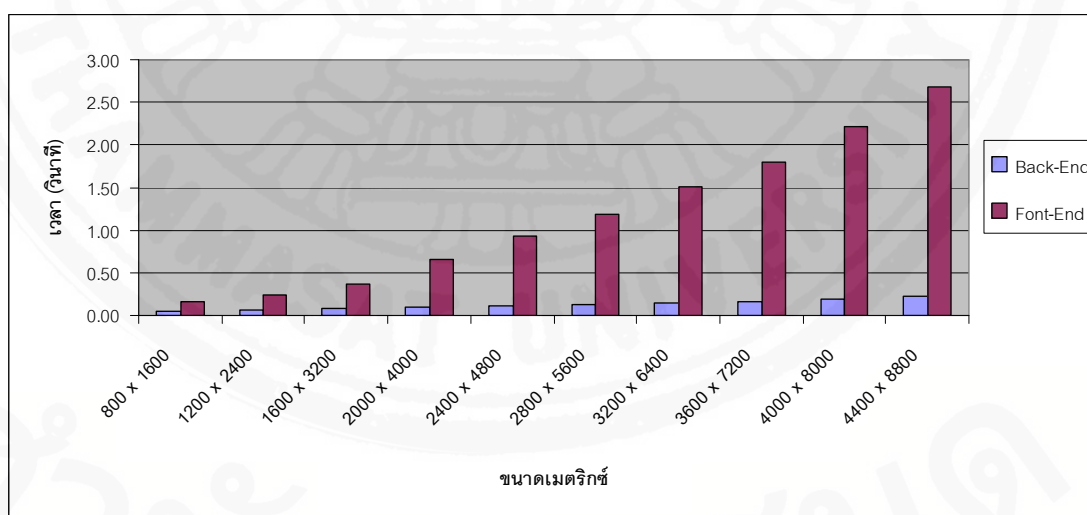
ผลการทดลองเปรียบเทียบการทำงานระหว่างแบ็คเอนด์ และฟรอนท์เอนด์  
ตั้งแต่ Initialization ถึง Function Call ด้วยการคูณเมตริกซ์

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B (Byte)	เวลา (วินาที)	
		แบ็คเอนด์ ( $T2$ )	ฟรอนท์เอนด์ ( $T1$ )
800 x 1600	7,680,000	0.055	0.159
1200 x 2400	17,280,000	0.061	0.233
1600 x 3200	30,720,000	0.077	0.370

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B (Byte)	เวลา (วินาที)	
		แบ็คเอนด์ (T2)	ฟรอนท์เอนด์ (T1)
2000 x 4000	48,000,000	0.094	0.657
2400 x 4800	69,120,000	0.110	0.929
2800 x 5600	94,080,000	0.124	1.207
3200 x 6400	122,880,000	0.148	1.522
3600 x 7200	155,520,000	0.165	1.799
4000 x 8000	192,000,000	0.191	2.213
4400 x 8800	232,320,000	0.218	2.699

ภาพที่ 4.3

ผลการทดลองเปรียบเทียบการทำงานระหว่างแบ็คเอนด์ และฟรอนท์เอนด์  
ตั้งแต่ Initialization ถึง Function Call ด้วยการคูณเมตริกซ์



ผลการทดลองจากตารางที่ 4.4 พบว่าเวลาที่ใช้ในการประมวลผลของฟรอนท์เอนด์ จะใช้เวลามากกว่าการประมวลผลของแบ็คเอนด์ คือ 0.104, 0.172, 0.293, 0.563, 0.819, 1.083, 1.374, 1.634, 2.022 และ 2.481 วินาที ตามลำดับของขนาดข้อมูลเมตริกซ์ที่เพิ่มมากขึ้น ซึ่งโอเวอร์เฮดดังกล่าวเกิดจากเวลาในการคัดลอกข้อมูลของ ฟรอนท์เอนด์ผ่านเน็ตเวิร์คไปยัง

แบ็คเอนด์บวกกับเวลาที่เสียไปสำหรับการเข้าใช้งานจีพียูของแบ็คเอนด์ เพราะการทำงานของพรอนท์เอนด์นั้นจำเป็นต้องส่งข้อมูลที่ต้องใช้ในการประมวลผลไปยังแบ็คเอนด์เพื่อใช้ในการประมวลผลและรอให้แบ็คเอนด์ประมวลผลการทำงานและเข้าถึงจีพียูทำให้เวลาในการประมวลผลของพรอนท์เอนด์มากขึ้นตามลำดับ

2. ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Host to Device (จากจีพียูไปจีพียู) ระหว่างแบ็คเอนด์ และพรอนท์เอนด์ ซึ่งผลการทดลองที่ได้เกิดจากการวัดเวลาที่ใช้ไปสำหรับฟังก์ชันคัดลอกข้อมูลตัวแปร A และตัวแปร B ของทั้งแบ็คเอนด์ และพรอนท์เอนด์ ดังภาพที่ 4.2 โดยการทดลองนี้จะเปรียบเทียบระยะเวลาที่  $T3$  กับ  $T4$  โดยผลการทดลองนี้ได้แสดงไว้ในตารางที่ 4.5

เมื่อ  $T3$  คือ ระยะเวลาที่ใช้ในการคัดลอกข้อมูลแบบ Host to Device ของพรอนท์เอนด์  
 $T4$  คือ ระยะเวลาที่ใช้ในการคัดลอกข้อมูลแบบ Host to Device ของแบ็คเอนด์

ตารางที่ 4.5

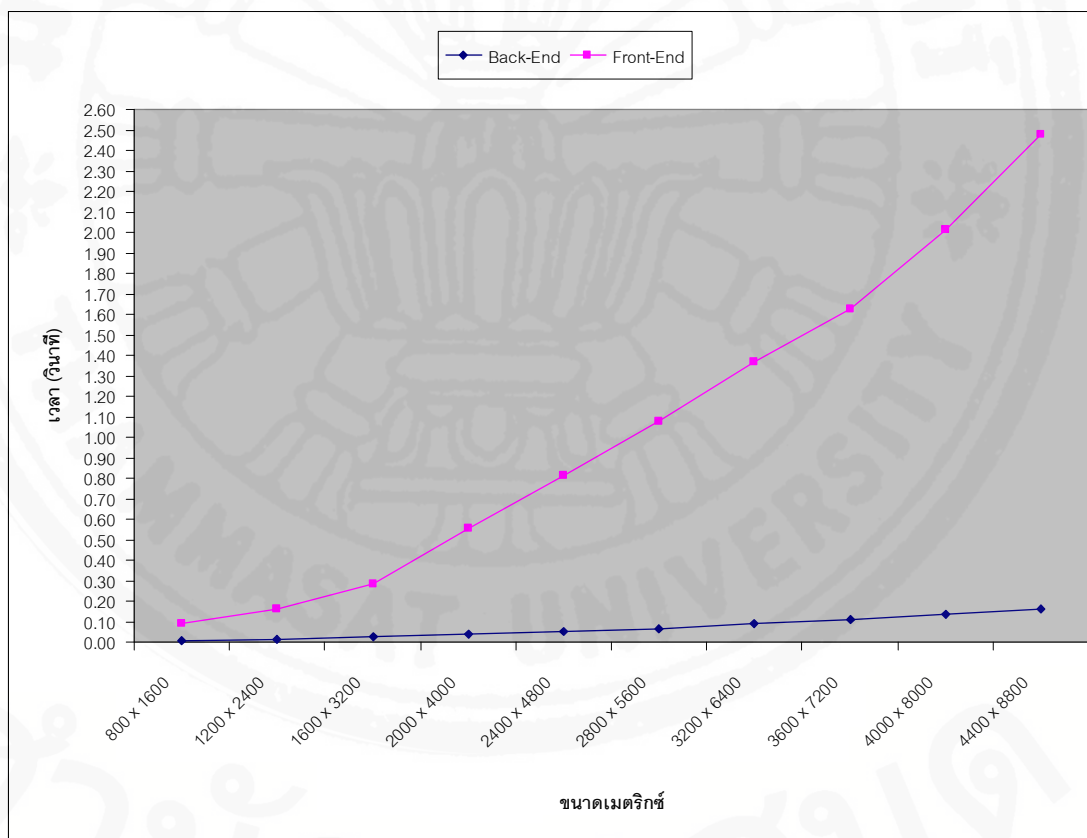
ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Host to Device (จากจีพียูไปจีพียู)  
 ระหว่างแบ็คเอนด์ และพรอนท์เอนด์

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B (Byte)	เวลา (วินาที)	
		แบ็คเอนด์ ( $T4$ )	พรอนท์เอนด์ ( $T3$ )
800 x 1600	7,680,000	0.006	0.092
1200 x 2400	17,280,000	0.013	0.163
1600 x 3200	30,720,000	0.023	0.283
2000 x 4000	48,000,000	0.036	0.553
2400 x 4800	69,120,000	0.052	0.811
2800 x 5600	94,080,000	0.067	1.075
3200 x 6400	122,880,000	0.090	1.366
3600 x 7200	155,520,000	0.108	1.626
4000 x 8000	192,000,000	0.134	2.013

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B (Byte)	เวลา (วินาที)	
		แบ็คเอนด์ ( $T_4$ )	ฟรอนท์เอนด์ ( $T_3$ )
4400 x 8800	232,320,000	0.160	2.476

ภาพที่ 4.4

ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Host to Device (จากซีพียูไปจีพียู)  
ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์



ผลการทดลองจากตารางที่ 4.5 พบว่าเวลาที่ใช้ในคัดลอกข้อมูลแบบ Host to Device ของฟรอนท์เอนด์จะใช้เวลามากกว่าคัดลอกข้อมูลแบบ Host to Device ของแบ็คเอนด์ คือ 0.086, 0.150, 0.260, 0.517, 0.759, 1.008, 1.276, 1.518, 1.879 และ 2.316 วินาที ตามลำดับของขนาดข้อมูลเมตริกซ์ที่เพิ่มมากขึ้น ซึ่งโอเวอร์เฮดดังกล่าวเกิดจากเวลาในการคัดลอกข้อมูลของ

พรอนท์เอนด์ผ่านเน็ตเวิร์คไปยังแบ็คเอนด์ แต่สำหรับการคัดลอกข้อมูลในส่วนของแบ็คเอนด์นั้นไม่จำเป็นต้องเสียเวลาในการคัดลอกข้อมูลผ่านเน็ตเวิร์ค เพราะข้อมูลได้ถูกนำมาวางไว้ที่แบ็คเอนด์เรียบร้อยแล้ว จึงทำให้เวลาที่เสียไปสำหรับการคัดลอกข้อมูลแบบ Host to Device ของแบ็คเอนด์ใช้เวลาน้อยกว่าพรอนท์เอนด์

3. ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Device to Host (จากจีพียูไปซีพียู) ระหว่างแบ็คเอนด์ และพรอนท์เอนด์ ซึ่งผลการทดลองที่ได้เกิดจากการวัดเวลาที่ใช้ไปสำหรับฟังก์ชันคัดลอกข้อมูลตัวแปร C ของทั้งแบ็คเอนด์ และพรอนท์เอนด์ ดังภาพที่ 4.2 โดยการทดลองนี้จะเปรียบเทียบระยะเวลาที่  $T5$  กับ  $T6$  โดยผลการทดลองนี้ได้แสดงไว้ในตารางที่ 4.6

เมื่อ  $T5$  คือ ระยะเวลาที่ใช้ในการคัดลอกข้อมูลแบบ Device to Host ของพรอนท์เอนด์

$T6$  คือ ระยะเวลาที่ใช้ในการคัดลอกข้อมูลแบบ Device to Host ของแบ็คเอนด์

ตารางที่ 4.6

ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Device to Host (จากจีพียูไปซีพียู)  
ระหว่างแบ็คเอนด์ และพรอนท์เอนด์

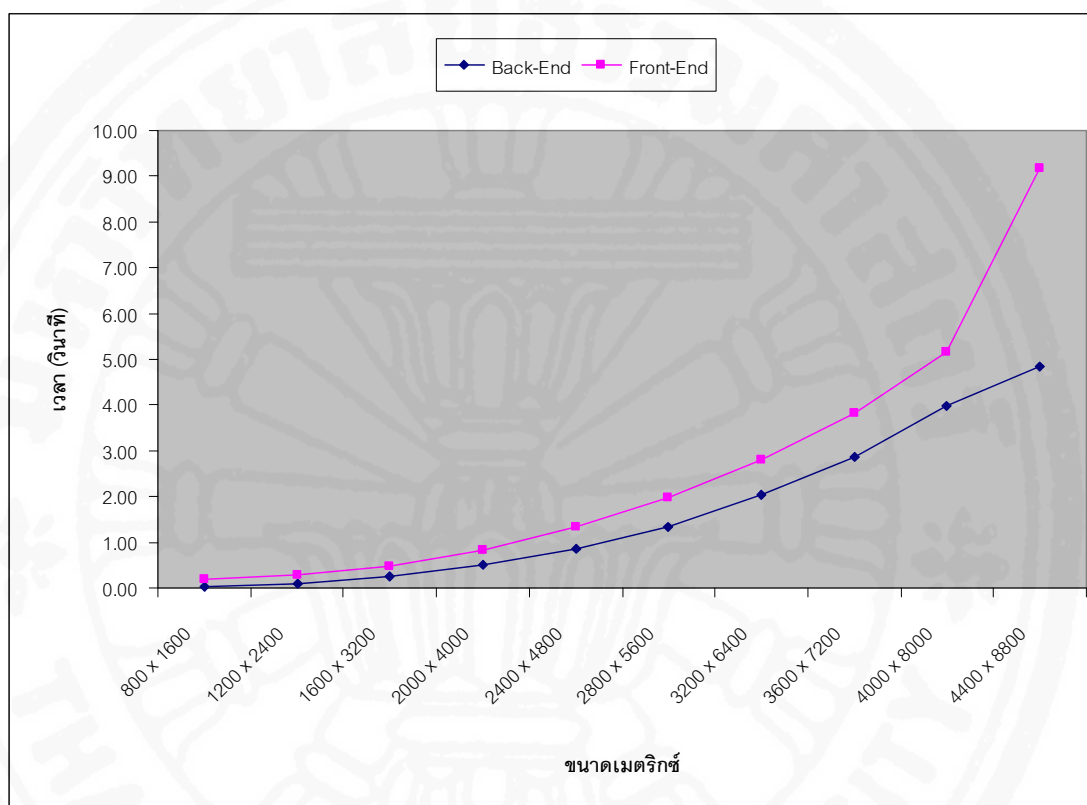
ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า C (Byte)	เวลา (วินาที)	
		แบ็คเอนด์ ( $T6$ )	พรอนท์เอนด์ ( $T5$ )
800 x 1600	5,120,000	0.034	0.184
1200 x 2400	11,520,000	0.111	0.276
1600 x 3200	20,480,000	0.255	0.469
2000 x 4000	32,000,000	0.495	0.822
2400 x 4800	46,080,000	0.863	1.344
2800 x 5600	62,720,000	1.346	1.966
3200 x 6400	81,920,000	2.030	2.813
3600 x 7200	103,680,000	2.850	3.833
4000 x 8000	128,000,000	3.974	5.175
4400 x 8800	154,880,000	4.842	9.180



ภาพที่ 4.5

ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Device to Host (จากจีพียูไปซีพียู)

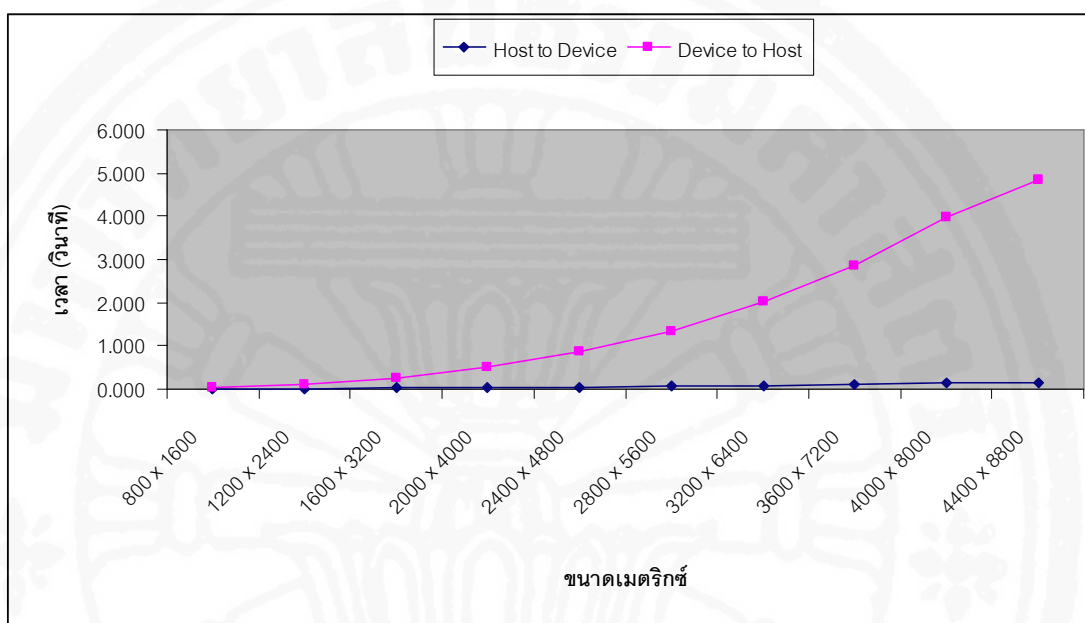
ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์



ผลการทดลองจากตารางที่ 4.6 พบว่าเวลาที่ใช้ในคัดลอกข้อมูลแบบ Device to Host ของฟรอนท์เอนด์จะใช้เวลามากกว่าคัดลอกข้อมูลแบบ Device to Host ของแบ็คเอนด์ คือ 0.150, 0.165, 0.214, 0.327, 0.481, 0.620, 0.783, 0.983, 1.201 และ 4.338 วินาที ตามลำดับของขนาดข้อมูลเมตริกซ์ที่เพิ่มมากขึ้น ซึ่งโอเวอร์เฮดดังกล่าวเกิดจากเวลาที่ใช้ในการคัดลอกข้อมูลจากยังแบ็คเอนด์ผ่านเน็ตเวิร์คกลับมาที่ฟรอนท์เอนด์บวกกับเวลาที่แบ็คเอนด์คัดลอกข้อมูลจากจีพียูมาวางไว้ยังซีพียูเพื่อเตรียมที่จะส่งกลับไปยังฟรอนท์เอนด์ แต่สำหรับการคัดลอกข้อมูลในส่วน of แบ็คเอนด์นั้นไม่จำเป็นต้องเสียเวลาในการคัดลอกข้อมูลผ่านเน็ตเวิร์ค แต่ทำแค่เพียงคัดลอกมาวางไว้ยังซีพียูเท่านั้น จึงทำให้เวลาที่เสียไปสำหรับการคัดลอกข้อมูลแบบ Device to Host ของแบ็คเอนด์ใช้เวลาน้อยกว่าฟรอนท์เอนด์

ภาพที่ 4.6

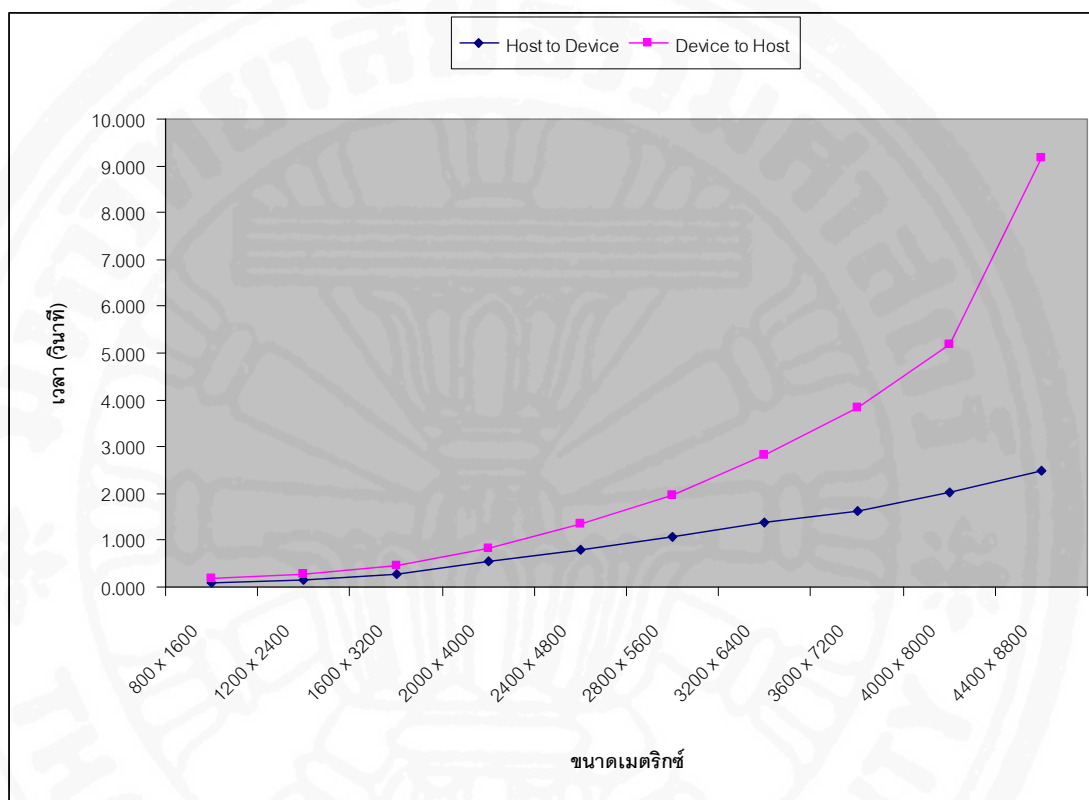
ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลของแบ็คเอนด์  
ระหว่าง Device to Host (จากจีพียูไปซีพียู) กับ Host to Device (จากซีพียูไปจีพียู)



จากผลการทดลองตามตารางที่ 4.5 และ ตารางที่ 4.6 พบว่าเวลาที่ใช้คัดลอกข้อมูลของแบ็คเอนด์แบบ Host to Device จะใช้เวลาในการคัดลอกข้อมูลน้อยกว่าเวลาที่ใช้สำหรับคัดลอกข้อมูลแบบ Device to Host เนื่องจากการคัดลอกแบบ Host to Device เป็นการทำ asynchronous function call แต่สำหรับการคัดลอกแบบ Device to Host เป็นการทำแบบ synchronous

ภาพที่ 4.7

ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลของพรอนท์เอนด์  
ระหว่าง Device to Host (จากจีพียูไปซีพียู) กับ Host to Device (จากซีพียูไปจีพียู)



จากผลการทดลองตามตารางที่ 4.5 และ ตารางที่ 4.6 พบว่าเวลาที่ใช้คัดลอกข้อมูลของพรอนท์เอนด์แบบ Host to Device จะใช้เวลาในการคัดลอกข้อมูลน้อยกว่าเวลาที่ใช้สำหรับคัดลอกข้อมูลแบบ Device to Host เนื่องจากการคัดลอกแบบ Host to Device ของพรอนท์เอนด์จะต้องเสียโอเวอร์เฮดในส่วนของเวลาที่ใช้ไปสำหรับการคัดลอกข้อมูลผ่านเน็ตเวิร์คจากพรอนท์เอนด์มาเก็บไว้ยังแบ็คเอนด์เท่านั้น แต่สำหรับการคัดลอกข้อมูลแบบ Device to Host จะต้องเสียโอเวอร์เฮดในส่วนของเวลาที่ใช้ไปสำหรับการคัดลอกข้อมูลจากจีพียูมาที่แบ็คเอนด์บวกกับเวลาที่ต้องถ่ายโอนข้อมูลผ่านเน็ตเวิร์คจากแบ็คเอนด์ไปยังพรอนท์เอนด์ทำให้เวลาในการคัดลอกข้อมูลแบบ Device to Host ใช้เวลามากกว่าการคัดลอกข้อมูลแบบ Host to Device

#### 4.1.3 ผลการทดลองการใช้งานจีพียูร่วมกันของเวอร์ชวลแมชีน

การทดลองในส่วนนี้จะวัดผลเพื่อแสดงถึงประโยชน์และความสามารถของเวอร์ชวลคู่มือในการจัดสรรทรัพยากรจีพียูสำหรับการใช้งานทรัพยากรจีพียูร่วมกันของเวอร์ชวลแมชีน โดยแสดงผลการทดลองออกเป็น 2 แบบ คือ 1) ผลการทดลองเปรียบเทียบการทำงานระหว่างการใช้งานจีพียูที่แบ็คเอนด์อนุญาตให้เข้าถึงได้เพียงครั้งละ 1 เวอร์ชวลแมชีนกับการใช้งานจีพียูพร้อมกันครั้งละมากกว่า 1 เวอร์ชวลแมชีน 2) ผลการทดลองเปรียบเทียบการทำงานระหว่างการใช้งานจีพียูที่ 1 เวอร์ชวลแมชีนกับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชีน โดยแสดงรายละเอียดผลการทดลองดังต่อไปนี้

1. ผลการทดลองเปรียบเทียบการทำงานระหว่างการใช้งานจีพียูที่แบ็คเอนด์อนุญาตให้เข้าถึงได้เพียงครั้งละ 1 เวอร์ชวลแมชีนกับการใช้งานจีพียูที่แบ็คเอนด์อนุญาตให้เข้าถึงได้พร้อมกันครั้งละมากกว่า 1 เวอร์ชวลแมชีน ซึ่งวัดผลด้วยการจับเวลาจาก 2 เวอร์ชวลแมชีนที่ต้องการเรียกใช้งานจีพียูผ่านเวอร์ชวลคู่มือพร้อมกันโดยประมวลผลการคูณเมตริกซ์ขนาดต่าง ๆ ผลที่ได้แสดงดังตารางที่ 4.7

ตารางที่ 4.7

ผลการทดลองเปรียบเทียบการทำงานเข้าถึงจีพียูสำหรับเวอร์ชวลแมชีน

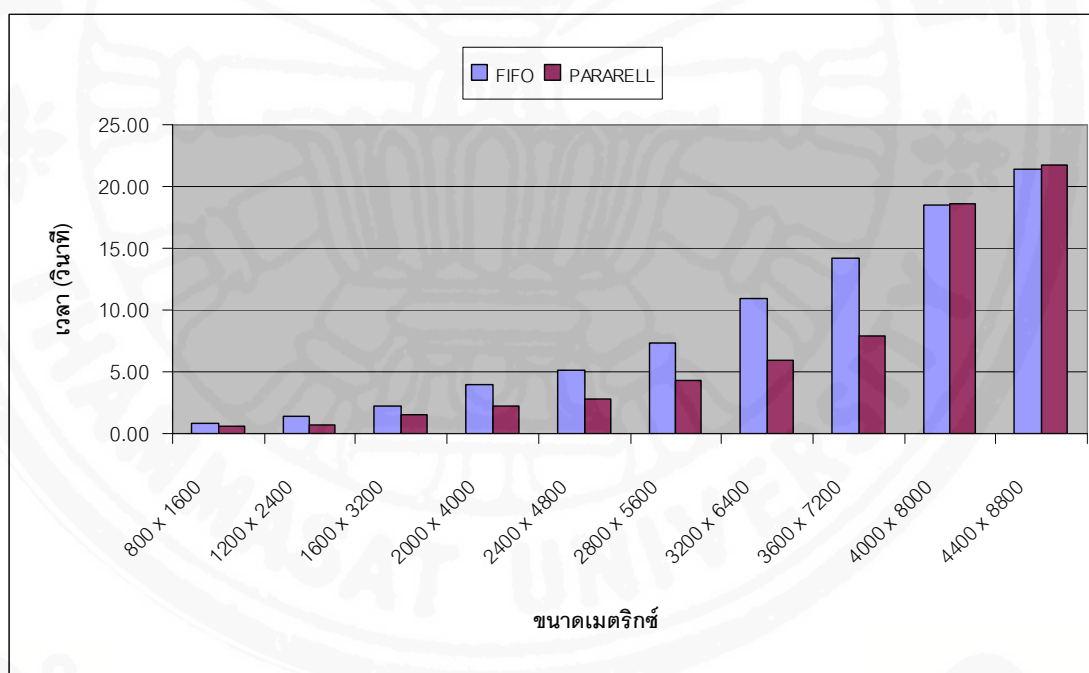
แบบครั้งละ 1 เวอร์ชวลแมชีน และพร้อมกัน 2 เวอร์ชวลแมชีน

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B+C (Byte)	เวลา (วินาที)	
		FIFO	PARARELL
800 x 1600	12,800,000	0.83	0.57
1200 x 2400	28,800,000	1.36	0.71
1600 x 3200	51,200,000	2.26	1.51
2000 x 4000	80,000,000	3.98	2.26
2400 x 4800	115,200,000	5.06	2.85
2800 x 5600	156,800,000	7.31	4.36
3200 x 6400	204,800,000	10.92	5.95

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B+C (Byte)	เวลา (วินาที)	
		FIFO	PARARELL
3600 x 7200	259,200,000	14.14	7.85
4000 x 8000	320,000,000	18.54	18.63
4400 x 8800	387,200,000	21.39	21.74

ภาพที่ 4.8

ผลการทดลองเปรียบเทียบการทำงานเข้าถึงจีพียูสำหรับเวอร์ชวลแมชีน  
แบบครั้งละ 1 เวอร์ชวลแมชีน และพร้อมกัน 2 เวอร์ชวลแมชีน



ผลการทดลองจากตารางที่ 4.7 พบว่าเวลาที่ใช้ในการประมวลผลการคูณเมตริกซ์ โดยเรียกใช้งานจีพียูผ่านเวอร์ชวลคู่มือกรณีแบ็คเอนด์อนุญาตให้เข้าถึงจีพียูได้เพียงครั้งละ 1 เวอร์ชวลแมชีนจะใช้เวลาในการประมวลผลมากกว่ากรณีแบ็คเอนด์อนุญาตให้ใช้งานจีพียูพร้อมกันครั้งละ 2 เวอร์ชวลแมชีน ที่เมตริกซ์มีขนาด 800 x 1600, 1200 x 2400, 1600 x 3200, 2000 x 4000, 2400 x 4800, 2800 x 5600, 3200 x 6400 และ 3600 x 7200 ด้วยผลต่างเวลา 0.26, 0.65, 1.72, 2.21, 2.95, 4.97 และ 6.29 วินาที ตามขนาดของเมตริกซ์ที่เพิ่มมากขึ้น แต่ขณะที่

เมตริกซ์มีขนาด  $4000 \times 8000$  และ  $4400 \times 8800$  นั้นการประมวลผลการคูณเมตริกซ์โดยเรียกใช้งานจีพียูผ่านเวอร์ชวลคูด้ากรณีแบ็คเอนด์อนุญาตให้เข้าถึงจีพียูได้เพียงครั้งละ 1 เวอร์ชวลแมชชีน จะใช้เวลาในการประมวลผลน้อยกว่ากรณีแบ็คเอนด์อนุญาตให้ใช้งานจีพียูพร้อมกันครั้งละ 2 เวอร์ชวลแมชชีน ที่ 0.09 และ 0.35 วินาที เนื่องจากเมตริกซ์ขนาด  $4000 \times 8000$  และ  $4400 \times 8800$  ต้องใช้พื้นที่สำหรับข้อมูลที่ต้องนำไปประมวลผลเกินกว่าพื้นที่ที่จีพียูมีให้บริการ ดังนั้นแบ็คเอนด์ของเวอร์ชวลคูด้าจึงทำการจัดสรรทรัพยากรจีพียูโดยให้เวอร์ชวลแมชชีนที่เข้ามาขอใช้จีพียูรอต่อคิวเพื่อใช้งานจีพียูจนกว่าเวอร์ชวลแมชชีนเครื่องแรกจะประมวลผลเสร็จและมีพื้นที่เหลือเพียงพอที่จะให้เวอร์ชวลแมชชีนเครื่องต่อไปทำงาน จึงเป็นเหตุให้เวลาที่ได้จากการประมวลผลสำหรับ 2 เวอร์ชวลแมชชีนที่สามารถเข้าใช้งานจีพียูได้พร้อมกันมีเวลามากกว่าการประมวลผลสำหรับ 2 เวอร์ชวลแมชชีนที่เข้าใช้งานจีพียูได้เพียงครั้งละ 1 เวอร์ชวลแมชชีน

2. ผลการทดลองเปรียบเทียบการทำงานระหว่างการใช้งานจีพียูที่ 1 เวอร์ชวลแมชชีน กับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชชีน โดยทำการจับเวลาการประมวลผลการคูณเมตริกซ์ขนาดต่าง ๆ ซึ่งผลที่ได้แสดงดังตารางที่ 4.8 ดังนี้

ตารางที่ 4.8

ผลการทดลองเปรียบเทียบการทำงาน

ระหว่างการใช้งานจีพียูที่ 1 เวอร์ชวลแมชชีน กับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชชีน

ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า $A+B+C$ (Byte)	เวลา (วินาที)	
		1 VM	2 VM (PARARELL)
$800 \times 1600$	12,800,000	0.46	0.57
$1200 \times 2400$	28,800,000	0.69	0.71
$1600 \times 3200$	51,200,000	1.12	1.51
$2000 \times 4000$	80,000,000	1.86	2.26
$2400 \times 4800$	115,200,000	2.77	2.85
$2800 \times 5600$	156,800,000	3.84	4.36
$3200 \times 6400$	204,800,000	5.15	5.95
$3600 \times 7200$	259,200,000	6.66	7.85

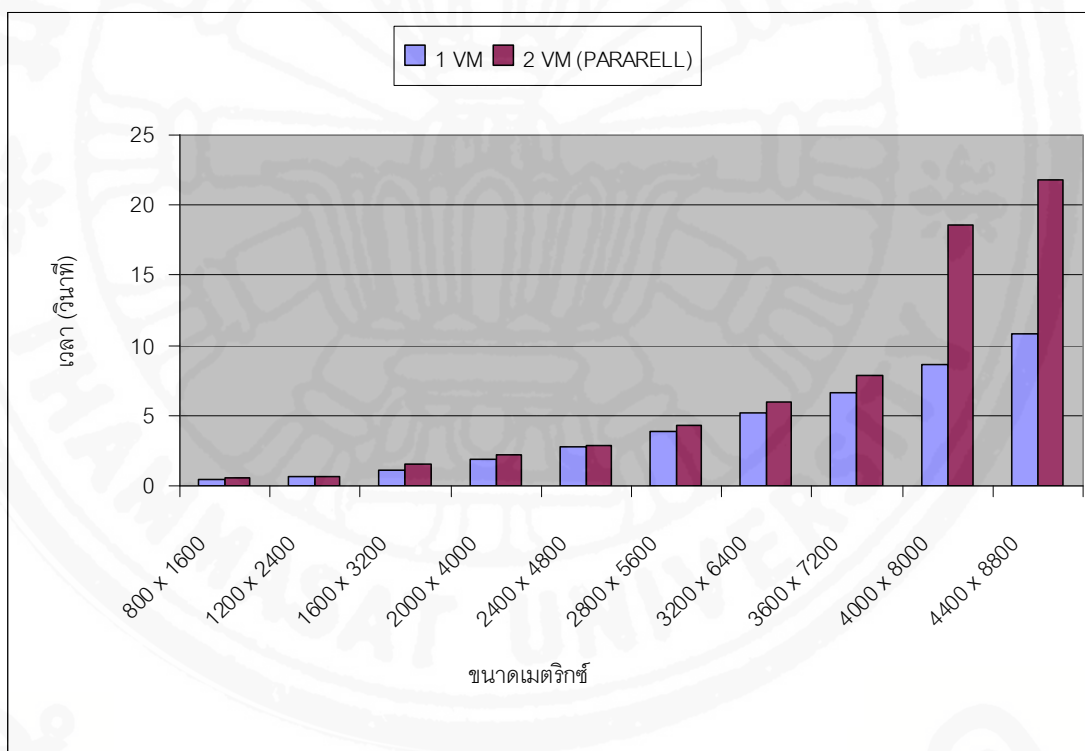


ขนาดเมตริกซ์	ขนาดเมตริกซ์ ตามค่า A+B+C (Byte)	เวลา (วินาที)	
		1 VM	2 VM (PARARELL)
4000 x 8000	320,000,000	8.62	18.63
4400 x 8800	387,200,000	10.81	21.74

ภาพที่ 4.9

ผลการทดลองเปรียบเทียบการทำงานระหว่าง

การใช้งานจีพียูที่ 1 เวอร์ชวลแมชีน กับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชีน



ผลการทดลองจากตารางที่ 4.8 พบว่า การประมวลผลการคูณเมตริกซ์ สำหรับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชีน ที่เมตริกซ์มีขนาด 800 x 1600, 1200 x 2400, 1600 x 3200, 2000 x 4000, 2400 x 4800, 2800 x 5600, 3200 x 6400 และ 3600 x 7200 เวลาที่ใช้ในการประมวลผลไม่มากเท่ากับ 2 เท่าของการประมวลผลการคูณเมตริกซ์สำหรับการใช้งานจีพียูที่ 1 เวอร์ชวลแมชีน แต่ใช้เวลามากกว่าเพียง 0.11, 0.02, 0.39, 0.40, 0.08, 0.52, 0.80

และ 1.19 วินาที ตามลำดับ แตกต่างกับที่เมตริกซ์มีขนาด  $4000 \times 8000$  และ  $4400 \times 8800$  คือ การประมวลผลการคูณเมตริกซ์ สำหรับการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชชีนจะใช้เวลา มากกว่า การประมวลผลการคูณเมตริกซ์สำหรับการใช้งานจีพียูที่ 1 เวอร์ชวลแมชชีน 2 เท่า เนื่องจากเมตริกซ์ขนาด  $4000 \times 8000$  และ  $4400 \times 8800$  ต้องใช้พื้นที่สำหรับข้อมูลที่ต้องนำไป ประมวลผลเกินกว่าพื้นที่ ที่จีพียูมีให้บริการ ดังนั้นแบ็คเอนด์ของเวอร์ชวลคูด้าจึงทำการจัดสรร ทรัพยากรจีพียูโดยให้เวอร์ชวลแมชชีนที่เข้ามาขอใช้จีพียูรอต่อคิวเพื่อใช้งานจีพียูจนกว่าเวอร์ชวล แมชชีนเครื่องแรกจะประมวลผลเสร็จและมีพื้นที่เหลือเพียงพอที่จะให้เวอร์ชวลแมชชีนเครื่องต่อไป ทำงาน จึงเป็นเหตุให้เวลาที่ได้จากการประมวลผลการใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชชีน มี เวลามากกว่าการใช้งานจีพียูที่ 1 เวอร์ชวลแมชชีนเป็น 2 เท่า

## 4.2 อภิปรายผลการวิจัย

### 4.2.1 อภิปรายผลการประมวลผลผ่านซีพียูและจีพียูของเวอร์ชวลแมชชีน

จากการผลทดลองแสดงให้เห็นได้ว่านอกจากเวอร์ชวลคูด้าจะทำให้เวอร์ชวลแมชชีน สามารถเรียกใช้งานจีพียูได้แล้ว เวอร์ชวลคูด้ายังเป็นผู้เพิ่มประสิทธิภาพในการประมวลผลของ แอปพลิเคชันบางประเภท เช่น การคูณเมตริกซ์ ซึ่งสังเกตได้จากผลการทดลองในกรณีที่เวอร์ชวล แมชชีนไม่สามารถเรียกใช้งานจีพียูให้ช่วยประมวลผล เวอร์ชวลแมชชีนจำเป็นต้องประมวลผลบน ซีพียูเท่านั้น ซึ่งเวลาที่ได้จากการใช้ซีพียูประมวลผลนั้นจะใช้เวลามากเมื่อเปรียบเทียบกับเมื่อใช้ จีพียูเข้ามาช่วยประมวลผล

### 4.2.2 อภิปรายผลของระยะเวลาที่เรียกใช้งานจีพียูผ่านเวอร์ชวลคูด้า

จากการผลทดลองในส่วนนี้ทั้ง 3 ผลการทดลองคือ 1) ผลการเปรียบเทียบการทำงาน ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ ตั้งแต่ Initialization ถึง Call Function ด้วยการคูณเมตริกซ์ 2) ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Host to Device (จากซีพียูไปจีพียู) ระหว่าง แบ็คเอนด์ และฟรอนท์เอนด์ 3) ผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Device to Host (จากจีพียูไปซีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ เห็นได้ชัดเจนว่าเวลาที่เสียไปในการ เรียกใช้งานจีพียูของเวอร์ชวลแมชชีนส่วนมากจะเป็นเวลาที่ใช้ในการถ่ายโอนข้อมูลจากฟรอนท์ เอนด์ไปยังแบ็คเอนด์ และการรอรับค่าจากแบ็คเอนด์มายังจากฟรอนท์เอนด์

#### 4.2.3 อภิปรายผลการใช้งานจีพียูร่วมกันของเวอร์ชวลแมชีน

จากการทดลองในส่วนนี้แสดงให้เห็นถึงประโยชน์และความสามารถของเวอร์ชวลคู่มือในการจัดสรรทรัพยากรจีพียูสำหรับการทำงานของทรัพยากรจีพียูร่วมกันของเวอร์ชวลแมชีน ซึ่งทำให้เวอร์ชวลแมชีนสามารถเข้าถึงจีพียูและประมวลผลคู่มือแอปพลิเคชันได้พร้อม ๆ กัน กรณีที่ทรัพยากรจีพียูมีพื้นที่เพียงพอต่อการเตรียมข้อมูลสำหรับนำไปประมวลผล อีกทั้งยังสามารถจัดคิวการทำงานให้กับเวอร์ชวลแมชีนที่ต้องการเข้าถึงจีพียูแต่พื้นที่สำหรับให้บริการบนจีพียูไม่เพียงพอให้สามารถเข้าใช้งานได้สำเร็จในคิวถัดไป



## บทที่ 5

### สรุปผลงานวิจัยและข้อเสนอแนะ

วิทยานิพนธ์ฉบับนี้มีวัตถุประสงค์เพื่อออกแบบระบบที่ชื่อว่า *เวอร์ชวลคูด้า* เพื่อจัดการ การใช้ทรัพยากรจีพียูร่วมกันสำหรับเวอร์ชวลแมชชีน จากเครื่องที่ให้บริการทรัพยากรจีพียูได้อย่างมีประสิทธิภาพ และสามารถจัดสรรทรัพยากรจีพียูให้เพียงพอต่อการใช้งานสำหรับคูด้า แอปพลิเคชันจากเวอร์ชวลแมชชีนที่ขอเข้าใช้งานจีพียูในเวลาเดียวกัน ดังนั้นในบทนี้เสนอ การสรุปผลการวิจัย และข้อเสนอแนะเพิ่มเติม รายละเอียดดังต่อไปนี้

#### 5.1 สรุปผลการวิจัย

##### 5.1.1 สรุปผลการทดลองประมวลผลผ่านจีพียูของเวอร์ชวลแมชชีนเปรียบเทียบกับ การประมวลผลด้วยจีพียูผ่านเวอร์ชวลคูด้าของเวอร์ชวลแมชชีน

จากผลการทดลองตามรูปที่ 4.3 ซึ่งแสดงค่า Speedup เปรียบเทียบประสิทธิภาพการทำงานระหว่างจีพียูผ่านเวอร์ชวลคูด้ากับจีพียูบนเวอร์ชวลแมชชีน จะเห็นได้ว่าการใช้จีพียูเข้ามาช่วยประมวลผลสำหรับแอปพลิเคชันบางประเภทบนเวอร์ชวลแมชชีน เช่น การคูณเมตริกซ์ จะทำให้แอปพลิเคชันดังกล่าวใช้เวลาในการประมวลผลน้อยกว่านำไปประมวลบนจีพียู และจากภาพที่ 4.1 แสดงให้เห็นว่าแนวโน้มจะเพิ่มขึ้นตามขนาดของข้อมูลที่ใช้ในการประมวลผลที่มากขึ้น แสดงว่าหากข้อมูลที่ต้องนำไปประมวลผลมีมากจีพียูจะต้องใช้เวลาในการประมวลผลมากกว่าจีพียูตามข้อมูลที่เพิ่มขึ้นตามลำดับ

ดังนั้นสามารถสรุปได้ว่าประสิทธิภาพในการใช้งานจีพียูผ่านเวอร์ชวลคูด้า นั้นดีกว่าประสิทธิภาพในการใช้งานจีพียูบนเวอร์ชวลแมชชีนมาก และเป็นการพิสูจน์ว่าถึงแม้ว่าเวอร์ชวลคูด้าจะต้องทำงานผ่านไลบรารี อีกทั้งต้องมีการถ่ายโอนข้อมูลไปมาระหว่างเวอร์ชวลแมชชีน กับโฮสโดยผ่านเน็ตเวิร์ค เวอร์ชวลคูด้าก็ยังทำงานได้ดีกว่าการไม่มีมันและผู้ใช้งานเวอร์ชวลแมชชีนต้องทำงานบนจีพียูของเวอร์ชวลแมชชีนเพียงอย่างเดียว

### 5.1.2 สรุปผลการทดลองของระยะเวลาที่เรียกใช้งานจีพียูผ่านเวอร์ชวลคูด้า

5.1.2.1 สรุปผลการเปรียบเทียบการทำงานระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ ตั้งแต่ Initialization (คำสั่ง `cuInit`) ถึง Function Call (คำสั่ง `cuLaunchGrid`) ด้วยการคูณเมตริกซ์ ด้วยวิธีการวัดผลตามภาพที่ 4.2 เปรียบเทียบระยะเวลาที่  $T1$  กับ  $T2$  จากผลการทดลองตามภาพที่ 4.3 สรุปได้ว่า การประมวลผลของฟรอนท์เอนด์จะใช้เวลาในการประมวลผลมากกว่าการประมวลผลของแบ็คเอนด์ ซึ่งโอเวอร์เฮดดังกล่าวเกิดจากเวลาในการคัดลอกข้อมูลของฟรอนท์เอนด์ผ่านเน็ตเวิร์คไปยังแบ็คเอนด์บวกกับเวลาที่เสียไปสำหรับการเข้าใช้งานจีพียูของแบ็คเอนด์ ก่อนที่จะส่งผลกลับมายังฟรอนท์เอนด์ว่าประมวลผลสำเร็จ

5.1.2.2 สรุปผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Host to Device (จากซีพียูไปจีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ ด้วยวิธีการวัดผลตามภาพที่ 4.2 เปรียบเทียบระยะเวลาที่  $T3$  กับ  $T4$  จากผลการทดลองตามภาพที่ 4.4 สรุปได้ว่า เวลาที่ใช้ในการคัดลอกข้อมูลแบบ Host to Device ของฟรอนท์เอนด์จะใช้เวลามากกว่าคัดลอกข้อมูลแบบ Host to Device ของแบ็คเอนด์ ซึ่งโอเวอร์เฮดดังกล่าวเกิดจากเวลาในการคัดลอกข้อมูลของฟรอนท์เอนด์ผ่านเน็ตเวิร์คไปยังแบ็คเอนด์ แต่สำหรับการคัดลอกข้อมูลในส่วนของแบ็คเอนด์นั้นไม่จำเป็นต้องเสียเวลาในการคัดลอกข้อมูลผ่านเน็ตเวิร์ค อีกทั้งเวลาในการคัดลอกข้อมูลนี้จะเพิ่มตามขนาดของข้อมูลที่ใช้ในการถ่ายโอนที่เพิ่มมากขึ้นตามลำดับ

5.1.2.3 สรุปผลการเปรียบเทียบฟังก์ชันการคัดลอกข้อมูลแบบ Device to Host (จากจีพียูไปซีพียู) ระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ ด้วยวิธีการวัดผลตามภาพที่ 4.2 เปรียบเทียบระยะเวลาที่  $T5$  กับ  $T6$  จากผลการทดลองตามภาพที่ 4.5 สรุปได้ว่า เวลาที่ใช้ในการคัดลอกข้อมูลแบบ Device to Host ของฟรอนท์เอนด์จะใช้เวลามากกว่าคัดลอกข้อมูลแบบ Device to Host ของแบ็คเอนด์ ซึ่งโอเวอร์เฮดดังกล่าวเกิดจากเวลาที่ใช้ในการคัดลอกข้อมูลจากยังแบ็คเอนด์ผ่านเน็ตเวิร์คกลับมาที่ฟรอนท์เอนด์บวกกับเวลาที่แบ็คเอนด์คัดลอกข้อมูลจากจีพียูมาวางไว้ยังซีพียูเพื่อเตรียมที่จะส่งกลับไปยังฟรอนท์เอนด์ แต่สำหรับการคัดลอกข้อมูลในส่วนของแบ็คเอนด์นั้นไม่จำเป็นต้องเสียเวลาในการคัดลอกข้อมูลผ่านเน็ตเวิร์ค

จากสรุปผลการทดลองทั้งหมดใน 5.1.2 นี้จะแสดงให้เห็นว่าโอเวอร์เฮดส่วนใหญ่ของการใช้งานผ่านเวอร์ชวลคูด้าจะเสียไปกับการถ่ายโอนข้อมูลไปมา แต่หากพิจารณาจากสรุปผลการทดลองที่ 5.1.1 ซึ่งแสดงให้เห็นว่าการใช้งานจีพียูผ่านเวอร์ชวลคูด้า นั้น ช่วยทำให้การประมวลผลเสร็จเร็วกว่าใช้เพียงซีพียูประมวลผลเพียงอย่างเดียว ถึงแม้จะเสียโอเวอร์เฮดในการถ่ายโอนข้อมูล



ไปมาก็ตาม การใช้งานผ่านเวอร์ชวลคูด้าก็ยังคงให้ประสิทธิภาพสำหรับทำงานที่ดีกว่าการใช้เพียงซีพียูประมวลผลเพียงอย่างเดียว หรือการที่ไม่มีเวอร์ชวลคูด้า

### 5.1.3 สรุปผลการใช้งานจีพียูร่วมกันของเวอร์ชวลแมชชีน

จากผลการทดลองตามภาพที่ 4.8 และ 4.9 แสดงให้เห็นว่าเวอร์ชวลคูด้าสามารถให้บริการกับ 2 เวอร์ชวลแมชชีนที่ต้องการเข้าใช้งานจีพียูพร้อมกันได้จริง เพราะขณะที่เมตริกซ์มีขนาด  $800 \times 1600$ ,  $1200 \times 2400$ ,  $1600 \times 3200$ ,  $2000 \times 4000$ ,  $2400 \times 4800$ ,  $2800 \times 5600$ ,  $3200 \times 6400$  และ  $3600 \times 7200$  จะใช้เวลาในการประมวลผลน้อยกว่า 2 เวอร์ชวลแมชชีนที่เข้าใช้งานจีพียูได้ครั้งละ 1 เวอร์ชวลแมชชีนตามรูปที่ 4.6 และใช้เวลาไม่เท่ากับ 2 เท่าเมื่อเปรียบเทียบ 1 เวอร์ชวลแมชชีนที่เข้าใช้งานจีพียูตามรูปที่ 4.7 แต่ขณะที่เมตริกซ์มีขนาด  $4000 \times 8000$  และ  $4400 \times 8800$  นั้น เวลาที่ใช้ประมวลผลจะใกล้เคียงกันกับการใช้งานจีพียูได้ครั้งละ 1 เวอร์ชวลแมชชีนกรณีเข้าใช้งานจีพียูพร้อมกัน 2 เวอร์ชวลแมชชีน เนื่องจากการเข้าใช้งานจีพียูพร้อมกันของ 2 เวอร์ชวลแมชชีนต้องใช้พื้นที่สำหรับข้อมูลที่ต้องนำไปประมวลผลเกินกว่าพื้นที่ ที่จีพียูมีให้บริการ จึงทำให้เกิดการรอที่จะประมวลผลของเวอร์ชวลแมชชีนตามที่แบ็คเอนด์ของเวอร์ชวลคูด้าจัดสรร

การทดลองนี้ได้กำหนดให้พื้นที่ ที่จีพียูสามารถให้บริการได้อยู่ที่ 640000000 Bytes หรือ 610.3515625 MB โดยประมาณ เนื่องจากพื้นที่ Memory ของ NVIDIA GeForce 8800GTS นี้มีพื้นที่ใช้งานอยู่ที่ 640 MB แต่ขณะที่ได้ทำการทดลองนั้น ผู้วิจัยได้กำหนดให้พื้นที่สำหรับให้บริการสำหรับจีพียูอยู่ที่ 640 MB พอดี ปรากฏว่าระบบเวอร์ชวลคูด้าไม่สามารถให้บริการพร้อมกัน 2 เวอร์ชวลแมชชีนได้และมีค่าผิดพลาด (Error) คืนกลับมาว่า Memory ไม่เพียงพอ ซึ่งแสดงว่าจีพียูไม่สามารถให้บริการพื้นที่ Memory ที่ 640 MB ได้ ผู้วิจัยจึงทำการลดขนาดของพื้นที่ ที่ให้บริการลงเหลือ 610.3515625 MB ทั้งนี้เพื่อให้เข้ากับขนาดของเมตริกซ์ที่ได้นำมาทดลองดังตารางที่ 4.1 ซึ่งเป็นค่าสูงสุดที่สามารถให้บริการได้ และเหมาะสมกับการทดลองดังกล่าว

#### 5.1.4 แอปพลิเคชันที่เหมาะสมสำหรับระบบเวอร์ชวลคูด้า

จากการงานวิจัยนี้ ได้ทำการทดลองคูด้าแอปพลิเคชันการคูณเมตริกซ์สำหรับระบบเวอร์ชวลคูด้าที่พัฒนาขึ้น จึงสรุปได้ว่าแอปพลิเคชันที่เหมาะสมสำหรับระบบเวอร์ชวลคูด้า คือ แอปพลิเคชันที่ต้องใช้เวลามากในการประมวลผลบนซีพียู จึงนำแอปพลิเคชันดังกล่าวมาให้จีพียูช่วยในการประมวลผล โดยเรียกใช้งานผ่านระบบเวอร์ชวลคูด้า ซึ่งเวลาที่แอปพลิเคชันประมวลผล



บนจีพียูจะต้องมากกว่าเวลาที่ใช้ในการถ่ายโอนข้อมูลไปมาระหว่างเวอร์ชวลแมชีนและเครื่องโฮสต์ หรือแอปพลิเคชันที่ประมวลผลบนจีพียูโดยผ่านระบบเวอร์ชวลคูด้าแล้วใช้เวลาน้อยกว่าเมื่อนำไปประมวลผลบนจีพียู

## 5.2 ข้อเสนอแนะเพิ่มเติม

ในส่วนนี้จะกล่าวถึงข้อเสนอแนะเพิ่มเติมของงานวิจัย รายละเอียดดังต่อไปนี้

1. จากผลการทดลองทั้งหมดแสดงให้เห็นได้ว่าการทำงานของระบบเวอร์ชวลคูด้าจะเสียเวลาไปกับการถ่ายโอนข้อมูลไปมาระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ เพื่อคัดลอกข้อมูลจากจีพียูของเวอร์ชวลแมชีนไปยังจีพียูและจากจีพียูกลับมายังจีพียูของเวอร์ชวลแมชีน หากลดเวลาในการถ่ายโอนข้อมูลไปมาระหว่างแบ็คเอนด์ และฟรอนท์เอนด์ของระบบเวอร์ชวลคูด้าได้ จะทำให้เวลาในการประมวลผลโดยใช้จีพียูผ่านเวอร์ชวลคูด้าใช้เวลาในการประมวลผลด้น้อยลง

2. ระบบเวอร์ชวลคูด้ารองรับเพียงฟังก์ชันพื้นฐานที่สำคัญของ CUDA Driver API ได้แก่

- |                                 |                       |
|---------------------------------|-----------------------|
| ➤ cuInit                        | ➤ cuParamSetv         |
| ➤ cuDeviceGet                   | ➤ cuParamSetSize      |
| ➤ cuCtxCreate                   | ➤ cuFuncSetBlockShape |
| ➤ cuModuleLoad                  | ➤ cuLaunchGrid        |
| ➤ cuModuleGetFunction           | ➤ cuCtxDetach         |
| ➤ cuMemAlloc                    | ➤ cuMemFree           |
| ➤ cuMemcpyHtoD/<br>cuMemcpyDtoH |                       |

แต่ยังไม่รองรับฟังก์ชันอื่น ๆ ของ CUDA Driver API หากระบบเวอร์ชวลคูด้าสามารถรองรับฟังก์ชันอื่น ๆ ได้จะทำให้สามารถรองรับคูด้าแอปพลิเคชันได้หลากหลายมากขึ้น

## รายการอ้างอิง

- John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell, "A Survey of General-Purpose Computation on Graphics", Computer Graphics Forum, vol. 26, 2007
- M. Garland et al., "Parallel Computing Experiences with CUDA", IEEE Micro, vol. 28, 2008
- David Kirk/NVIDIA and Wen-mei Hwu, "CUDA Programming Model", 2008
- Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron, "Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors", IPDPS, 2009
- Brian Burke, "NVIDIA CUDA Technology Dramatically Advances The Pace of Scientific Research", [http://www.nvidia.com/object/io\\_1229516081227.html](http://www.nvidia.com/object/io_1229516081227.html)
- Andrew Humber, "NVIDIA Achieves Monumental Folding@Home Milestone With Cuda", [http://www.nvidia.com/object/io\\_1219747545128.html](http://www.nvidia.com/object/io_1219747545128.html)
- James E. Smith and Ravi Nair, "The architecture of virtual machines", Computer, vol. 38, 2005
- Keith Adams and Ole Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization", in Proc. of the 12th Intl. Conf. on Architectural support for programming lang. and OS, ACM, 2006
- KVM, <http://www.linux-kvm.org>
- M. Satyanarayanan and Eyal de Lara, "VMM-independent graphics acceleration", in Proceedings of the 3rd international conference on Virtual execution environments, ACM, 2007
- OpenGL - The Industry Standard for High Performance Graphics, <http://www.opengl.org>
- Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar and Parthasarathy Ranganathan, "GViM: GPU-accelerated Virtual Machines", in Proceedings of the 3rd ACM Workshop on System level Virtualization for High Performance Computing, ACM, 2009

Xen, <http://www.xen.org>

Micah Dowty and Jeremy Sugerman, "GPU virtualization on VMware's hosted I/O Architecture", SIGOPS Oper. Syst. Rev., vol. 43, 2009

NVIDIA, "NVIDIA CUDA C Programming Guide version 3.2", <http://www.nvidia.com>, 2010

NVIDIA, <http://forums.nvidia.com/index.php?showtopic=190535>





ภาคผนวก

เจ้าหน้าที่หอสมุด

## ผนวก ก

## ตัวอย่างโปรแกรม

## การคูณเมตริกซ์แบบ Sequential

```

void
matrixMul(float* C, const float* A, const float* B, unsigned int hA, unsigned int wA,
unsigned int wB)
{
    for (unsigned int i = 0; i < hA; ++i)
        for (unsigned int j = 0; j < wB; ++j) {
            float sum = 0;
            for (unsigned int k = 0; k < wA; ++k) {
                float a = A[i * wA + k];
                float b = B[k * wB + j];
                sum += a * b;
            }
            C[i * wB + j] = (float)sum;
        }
}

```

## การคูณเมตริกซ์แบบ Parallel

การคูณเมตริกซ์แบบ Parallel นี้เป็นตัวอย่างการคูณโดยใช้ CUDA

```

__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

```

```
// Thread index
int tx = threadIdx.x;
int ty = threadIdx.y;

// Index of the first sub-matrix of A processed by the block
int aBegin = wA * BLOCK_SIZE * by;

// Index of the last sub-matrix of A processed by the block
int aEnd = aBegin + wA - 1;

// Step size used to iterate through the sub-matrices of A
int aStep = BLOCK_SIZE;

// Index of the first sub-matrix of B processed by the block
int bBegin = BLOCK_SIZE * bx;

// Step size used to iterate through the sub-matrices of B
int bStep = BLOCK_SIZE * wB;

// Csub is used to store the element of the block sub-matrix
// that is computed by the thread
float Csub = 0;

// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
    a <= aEnd;
    a += aStep, b += bStep) {
```



```

// Declaration of the shared memory array As used to
// store the sub-matrix of A
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

// Declaration of the shared memory array Bs used to
// store the sub-matrix of B
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

// Load the matrices from device memory
// to shared memory; each thread loads
// one element of each matrix
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];

// Synchronize to make sure the matrices are loaded
__syncthreads();

// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += AS(ty, k) * BS(k, tx);

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
}

```

```

// Write the block sub-matrix to device memory;

// each thread writes one element

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;

C[c + wB * ty + tx] = Csub;
}

```

### ตัวอย่างการเข้าถึงและสั่งงานจีพียูด้วย CUDA Programming

การเข้าถึงและสั่งงานจีพียูด้วย CUDA Programming โดยปกติ สามารถสั่งงานจีพียูผ่านคำสั่งของคูด้าได้ 2 ลักษณะคือ CUDA runtime API และ CUDA driver API ดังตัวอย่าง ซึ่งการทำงานของแบ็คเอนด์ในระบบเวอร์ชวลคูด้าที่ได้พัฒนาขึ้นมานั้น จะทำงานโดยเข้าถึงจีพียูด้วยคำสั่งของคูด้าแบบ CUDA driver API โดยจะใช้ฟังก์ชันพื้นฐานที่ได้กล่าวไว้ในข้อ 3.1.1.2

#### ตัวอย่าง CUDA Programming แบบ CUDA runtime API

```

// Device code

__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code

int main()
{
    // Allocate vectors in device memory

    size_t size = N * sizeof(float);

    float* d_A;

```

```

cudaMalloc((void**)&d_A, size);
float* d_B;
cudaMalloc((void**)&d_B, size);
float* d_C;
cudaMalloc((void**)&d_C, size);
// Copy vectors from host memory to device memory
// h_A and h_B are input vectors stored in host memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>>(d_A, d_B, d_C);
// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
}

```

ที่มา : “CUDA Programming Guide Version 2.3” โดย NVIDIA CUDA™ จาก

<http://www.nvidia.com>

#### ตัวอย่าง CUDA Programming แบบ CUDA driver API

```

// Host code
int main()
{
// Initialize

```

```
if (cuInit(0) != CUDA_SUCCESS)
    exit (0);

// Get number of devices supporting CUDA
int deviceCount = 0;
cuDeviceGetCount(&deviceCount);
if (deviceCount == 0) {
    printf("There is no device supporting CUDA.\n");
    exit (0);
}

// Get handle for device 0
CUdevice cuDevice = 0;
cuDeviceGet(&cuDevice, 0);

// Create context
CUcontext cuContext;
cuCtxCreate(&cuContext, 0, cuDevice);

// Create module from binary file
CUmodule cuModule;
cuModuleLoad(&cuModule, "VecAdd.ptx");

// Get function handle from module
CUfunction vecAdd;
cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");

// Allocate vectors in device memory
size_t size = N * sizeof(float);
CUdeviceptr d_A;
cuMemAlloc(&d_A, size);
CUdeviceptr d_B;
cuMemAlloc(&d_B, size);
CUdeviceptr d_C;
cuMemAlloc(&d_C, size);
```

```

// Copy vectors from host memory to device memory
// h_A and h_B are input vectors stored in host memory
cuMemcpyHtoD(d_A, h_A, size);
cuMemcpyHtoD(d_B, h_B, size);

// Invoke kernel
#define ALIGN_UP(offset, alignment) \
    (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)

int offset = 0;
void* ptr;
ptr = (void*)(size_t)d_A;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);
ptr = (void*)(size_t)d_B;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);
ptr = (void*)(size_t)d_C;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);
cuParamSetSize(VecAdd, offset);

int threadsPerBlock = 256;
int blocksPerGrid =(N + threadsPerBlock - 1) / threadsPerBlock;
cuFuncSetBlockShape(vecAdd, threadsPerBlock, 1, 1);
cuLaunchGrid(VecAdd, blocksPerGrid, 1);

// Copy result from device memory to host memory
// h_C contains the result in host memory
cuMemcpyDtoH(h_C, d_C, size);

```

```
// Free device memory  
cuMemFree(d_A); cuMemFree(d_B); cuMemFree(d_C);  
}
```

ที่มา : “CUDA Programming Guide Version 2.3” โดย NVIDIA CUDA™ จาก

<http://www.nvidia.com>





## ประวัติการศึกษา

ชื่อ	นางสาวสุนทรี บุญมี
วันเดือนปีเกิด	22 กุมภาพันธ์ 2524
วุฒิการศึกษา	ปริญญาตรี สาขาวิทยาการคอมพิวเตอร์ สถาบันเทคโนโลยีพระจอมเกล้า พระนครเหนือ
ผลงานทางวิชาการ	เวอร์ชวลคู่มือ : ระบบให้บริการเข้าถึงทรัพยากรดิจิทัล สำหรับเวอร์ชวลแมชีน
ประสบการณ์ทำงาน	ปี 2547- ปัจจุบัน โปรแกรมเมอร์ ธนาคารกรุงศรีอยุธยา จำกัด มหาชน

ชำนาญการพิเศษ