TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN ĐIỆN TỬ - VIỄN THÔNG

BÁO CÁO MÔN HỌC

KIẾN TRÚC MÁY TÍNH

Đề tài:

TÌM HIỂU KIẾN TRÚC CỦA BỘ XỬ LÝ ĐỒ HỌA GPU

Sinh viên thực hiện	MSSV	Lớp
Vũ Đức Thái	20172804	ĐTVT.08-K62
Bùi Huy Hoàng	20172568	ĐTVT.07-K62
Nguyễn Văn Hoàng	20172579	ĐTVT.07-K62
Nguyễn Văn Quân	20172769	ĐTVT.06-K62
Nguyễn Hữu Đức	20172478	ĐTVT.06-K62

Giảng viên hướng dẫn: PGS.TS. Nguyễn Đức Minh

Hà Nội, 6-2021

LỜI NÓI ĐẦU

Trong những năm gần đây, với sự phát triển mạnh mẽ của trí tuệ nhân tạo và các ứng dụng xung quanh nó, chúng ta có thể thấy được tầm rất quan trọng của chúng trong tương lai gần. Việc triển khai các mô hình học máy được gia tăng đáng kể nhờ vào việc tính toán song song trên GPU. Một trong những công ty đi đầu về sản xuất GPU là NVIDIA đã cho ra mắt kiến trúc CUDA; năm 2017, NVIDIA phát triển thế hệ nhân mới cho GPU của mình là tensor-core cùng với thế hệ cũ là CUDA-core đã giúp các thuật toán học máy được tăng tốc đáng kể, việc nghiên cứu triển khai các mô hình này xuống kit nhúng cũng đã trở nên rất nổi bật trong những năm trở lại đây.

Trong bài tập lớn kiến trúc máy tính, chúng em có mong muốn được học thêm về kiến trúc của GPU (cụ thể là GPU do NVIDIA phát triển) nên đã quyết định chọn đề tài này để thực hiện.

Trong báo cáo của chúng em, gồm có các nội dung chính như sau:

- **Chương 1 – Lý thuyết về bộ xử lý đồ họa:** Trong chương này sẽ trình bày tổng quan về bộ xử lý đồ họa.
- **Chương 2 - Lập kế hoạch thực hành:** Phân chia công việc thực hiện nghiên cứu đề tài và thực hành.
- **Chương 3 – Kiến trúc CUDA-NVIDIA:** Trình bày chi tiết hơn về kiến trúc phần cứng của bộ xử lý đồ họa CUDA do NVIDIA phát triển.
- **Chương 4 – Lập trình CUDA:** Ngôn ngữ mở rông C/C++ để lập trình trên kiến trúc GPU của NVIDIA, các độ đo đánh giá hiệu năng thuật toán sẽ được áp dụng cho chương 5.
- **Chương 5 – Triển khai các chương trình tính toán trên GPU**: Triển khai chương trình nhân ma trận và bộ lọc trung vị, đo các tham số đánh giá thuật toán.
- **Chương 6 – Kết luận:** Kết luận chung đề tài, vấn đề còn tồn đọng, các kết quả đạt được, phương hướng phát triển.

Kết hợp giữa các kiến thức cơ bản trên lớp và song hành cùng đề tài này đã giúp cho các thành viên trong nhóm bước đầu có được những kiến thức quý giá về lập trình song song, kiến trúc của GPU những kỹ năng làm việc và phân chia công việc

một cách hiệu quả hơn, kỹ năng thuyết trình,… Qua đây chúng em gửi lời cảm ơn chân thành tới thầy Nguyễn Đức Minh đã hỗ trợ chúng em thực hiện đề tài này.

Nhóm xin chân thành cám ơn!

MỤC LỤC

LỜI NÓI ĐẦU ..... **i**

DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT ..... **i**

DANH MỤC HÌNH VẼ ..... **ii**

DANH MỤC BẢNG BIỂU ..... **iii**

CHƯƠNG 1. LÝ THUYẾT VỀ BỘ XỬ LÝ ĐỒ HỌA GPU **1**

1.1 Sự phát triển của bộ xử lý đồ họa ***1***

1.1.1 Lịch sử phát triển của GPU 1

1.1.2 GPU Unifes Graphics and Computing 2

1.1.3 Why CUDA and GPU Computing? 2

1.2 Kiến trúc hệ thống của GPU ***3***

1.2.1 Heterogeneous System 3

1.2.2 Processor Array 3

CHƯƠNG 2. PHÂN CÔNG CÔNG VIỆC **5**

2.1 Công việc chính cần thực hiện ***5***

2.2 Nhân lực làm việc ***5***

2.3 Phân chia công việc ***6***

CHƯƠNG 3. KIẾN TRÚC CUDA – NVIDIA **8**

3.1 Multithreaded Multiprocessor Architecture ***8***

3.1.1 Massive Multithreading 8

3.1.2 Kiến trúc của bộ đa xử lý (Multiprocessor Architecture) 9

3.1.3 Single-Instruction Multiple-Thread (SIMT) 11

3.1.4 Sự phân kỳ và thực thi warp SIMT (SIMT Warp Execution and Divergence) 13

3.1.5 Quản lý luồng và khối luồng 14

3.1.6 Thread Instructions 14

3.1.7 Streaming Processor (SP) 15

3.1.8 Đơn vị chức năng đặc biệt (Special Function Unit - SFU) 15

3.1.9 So sánh với các bộ đa xử lý khác 15

3.1.10 Kết luận đa xử lý đa luồng (Multithreaded Multiprocessor) 16

3.2 Parallel memory system *16*
3.2.1 Introduction 16
3.2.2 CUDA memory types 17
3.2.3 Unified memory 22
CHƯƠNG 4. LẬP TRÌNH CUDA **23**
4.1 Mở rộng ngôn ngữ C++ trong lập trình CUDA *23*
4.1.1 Function Execution Space Specifiers 23
4.1.2 Variable Memory Space Specifiers 25
4.1.3 Built-in Vector Types 28
4.1.4 Memory Fence Functions 29
4.1.5 Synchronization Functions 33
4.2 Các tham số đánh giá hiệu năng của chương trình *34*
4.2.1 Định thời gian 34
4.2.2 Băng thông 36
4.2.3 Thông lượng tính toán 38
CHƯƠNG 5. TRIỂN KHAI CÁC CHƯƠNG TRÌNH TÍNH TOÁN TRÊN GPU – NVIDIA **39**
5.1 Chương trình nhân hai ma trận *39*
5.1.1 Thuật toán nhân hai ma trận vuông sử dụng shared memory 39
5.1.2 Thuật toán nhân hai ma trận vuông sử dụng thư viện cublas 41
5.1.3 Đánh giá benchmark của các chương trình tính 41
5.2 Triển khai bộ lọc trung vị cho ảnh *44*
5.2.1 Thuật toán lọc trung vị không sử dụng shared memory 44
5.2.2 Thuật toán lọc trung vị sử dụng shared memory 45
5.2.3 Đánh giá benchmark của các chương trình 47
CHƯƠNG 6. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN **50**
6.1 Tóm tắt kết quả *50*
6.2 Bài học kinh nghiệm *50*
6.3 Phương hướng phát triển trong tương lai *50*
6.4 Kết luận chung *51*
TÀI LIỆU THAM KHẢO **52**
PHỤ LỤC **53**

DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT

Từ viết tắt	Tiếng anh	Tiếng việt
GPU	Graphics Processing Unit	Bộ xử lý đồ họa
SIMT	Single-Instruction Multiple-Thread	Mô hình thực thi một chỉ dẫn, nhiều luồng thực thi
SP	Streaming Processor	Bộ xử lý luồng
MP	Streaming Multiprocessor	Bộ đa xử lý luồng

DANH MỤC HÌNH VẼ

Hình 1-1 Kiến trúc GPU thống nhất cơ bản 4

Hình 3-1 Bộ xử lý đa luồng với 8 lõi của bộ xử lý vô hướng (Scalar Processor-SP) 10

Hình 3-2 Lập lịch warp đa luồng SIMT 12

Hình 3-3 Cấu trúc hệ thống bộ nhớ trong GPU của NVIDIA 18

Hình 3-4 Minh họa cơ chế Unified Memory 22

Hình 5-1 Ma trận M 39

Hình 5-2 Ma trận M biểu diễn bằng mảng một chiều 39

Hình 5-3 Minh họa nhân hai ma trận 40

Hình 5-4 Chia ma trận thành các khối 40

Hình 5-5 Nhân ma trận sử dụng shared mem 41

Hình 5-6 So sánh thời gian thực hiện giữa cublas, shmem và cpu 42

Hình 5-7 So sánh GFLOP/s giữa cublas, shmem và cpu 43

Hình 5-8 So sánh bandwiwth effective giữa cublas, shmem và cpu 44

Hình 5-9 Ảnh kết quả khi thực hiện lọc ảnh 810x1440 (không sử dụng shared mem) nhiễu muối tiêu với kernel size là 7 45

Hình 5-10 Minh họa về cách thiết kế bộ lọc Median Filter sử dụng shared memory 47

Hình 5-11 Ảnh kết quả khi thực hiện lọc ảnh (sử dụng shared mem) nhiễu muối tiêu với kernel size là 7 47

Hình 5-12 So sánh thời gian thực hiện lọc khi thay đổi kích thước của ảnh và bộ lọc 48

Hình 5-13 So sánh băng thông thực hiện lọc khi thay đổi kích thước của ảnh và bộ lọc 49

DANH MỤC BẢNG BIỂU

Bảng 2-1 Bảng đánh giá nhân lực 5
Bảng 2-2 Bảng phân công công việc.....7

CHƯƠNG 1. LÝ THUYẾT VỀ BỘ XỬ LÝ ĐỒ HỌA GPU

1.1 Sự phát triển của bộ xử lý đồ họa

1.1.1 Lịch sử phát triển của GPU

Vào khoảng 15 năm trước, không có cái gọi là GPU. Đồ họa trên PC được thực hiện bởi bộ điều khiển mảng đồ họa video (VGA). Bộ điều khiển VGA chỉ đơn giản là một bộ điều khiển bộ nhớ và bộ kết nối màn hình, nâng cao hiệu năng được kết nối với một số DRAM. Vào những năm 1990, công nghệ bán dẫn đã phát triển đến mức có thể thêm nhiều chức năng vào bộ điều khiển VGA. Đến năm 1997, bộ điều khiển VGA bắt đầu kết hợp một số chức năng hiển thị ba chiều (3D), bao gồm cả phần cứng để thiết lập và rasterization không gian 3 chiều (lấy một hình ảnh được mô tả ở định dạng đồ họa vector và chuyển đổi nó thành hình ảnh raster) và ánh xạ kết cấu và đổ bóng (áp dụng "decals" hoặc mẫu cho pixel và pha trộn màu).

Vào năm 2000, bộ xử lý đồ họa chip đơn đã kết hợp hầu hết mọi chi tiết của hệ thống đồ họa máy trạm cao cấp truyền thống và do đó, xứng đáng có một cái tên mới ngoài bộ điều khiển VGA. Thuật ngữ GPU được đặt ra để biểu thị rằng thiết bị đồ họa đã trở thành một bộ xử lý.

Theo thời gian, GPU trở nên dễ lập trình hơn, vì bộ vi xử lý có thể lập trình thay thế logic chuyên dụng có chức năng cố định trong khi vẫn duy trì hệ thống đồ họa 3D cơ bản. Ngoài ra, các phép tính trở nên chính xác hơn theo thời gian, tiến triển từ số học được lập chỉ mục, đến số nguyên và điểm cố định, đến dấu phẩy động chính xác đơn và gần đây là dấu chấm động chính xác kép. GPU đã trở thành bộ xử lý lập trình song song hàng loạt với hàng trăm lõi và hàng nghìn luồng.

Sau quãng thời gian dài, các lệnh của bộ vi xử lý và phần cứng bộ nhớ đã được thêm vào để hỗ trợ các ngôn ngữ lập trình mục đích chung và môi trường lập trình đã được tạo ra để cho phép các GPU được lập trình bằng các ngôn ngữ quen thuộc, bao gồm C và C ++. Sự đổi mới này làm cho GPU trở thành bộ xử lý đa điểm, có thể lập trình, đa năng hoàn toàn, mặc dù vẫn có một số hạn chế đặc biệt.

1.1.2 GPU Unifes Graphics and Computing

Với việc bổ sung thêm chức năng tính toán CUDA và GPU vào các khả năng của GPU, giờ đây có thể sử dụng GPU làm cả bộ xử lý đồ họa và bộ xử lý tính toán cùng một lúc và kết hợp những công dụng này trong các ứng dụng tính toán thực tiễn. Kiến trúc bộ xử lý cơ bản của GPU được thể hiện theo hai cách:

- Thứ nhất: khi triển khai các API đồ họa có thể lập trình.
- Thứ hai: nó như một mảng bộ xử lý song song khổng lồ có thể lập trình trong C / C ++ với CUDA.

Mặc dù các bộ xử lý cơ bản của GPU là thống nhất, nhưng không nhất thiết tất cả các chương trình luồng SPMD đều giống nhau. GPU có thể chạy các chương trình Shader đồ họa cho khía cạnh đồ họa của GPU, xử lý hình học, vertices và pixel, đồng thời cũng chạy các chương trình luồng trong CUDA.

Ta có thể kết luận, GPU là một kiến trúc đa xử lý linh hoạt, hỗ trợ nhiều tác vụ xử lý khác nhau. GPU rất phù hợp cho các tác vụ về đồ họa và phân tích hình ảnh vì chúng được thiết kế đặc biệt cho các tác vụ này. GPU cũng rất tương thích trong nhiều ứng dụng thông lượng mục đích chung là “first cousins” của đồ họa, ở chỗ chúng thực hiện nhiều công việc song song, cũng như có nhiều cấu trúc vấn đề thường xuyên. Nói chung, chúng rất phù hợp với các bài toán dữ liệu song song (đã được đề cập đến ở Chương 6), đặc biệt là các bài toán lớn, nhưng ít hơn đối với các bài toán nhỏ hơn.

1.1.3 Why CUDA and GPU Computing?

Bộ xử lý đồng nhất và có thể mở rộng đã gợi ý và tạo nên một mô hình lập trình mới cho GPU. Một lượng lớn tiến trình xử lý dấu phẩy động của mảng bộ xử lý GPU được sử dụng để sử lý các công vệc phức tạp. Với khả năng xử lý song song tối ưu và khả năng mở rộng của mảng bộ xử lý cho các ứng dụng đồ họa, mô hình lập trình cho tính toán tổng quát hơn được thể hiện trực tiếp qua tính song song.

GPU computing là thuật ngữ được đặt ra để sử dụng các GPU để tính toán thông qua ngôn ngữ lập trình song song và APIs, mà không sử dụng đồ hoạ API truyền thống và mô hình đường ống đồ họa. Điều này trái ngược với phương pháp tính toán Mục đích chung trên GPU (GPGPU) trước đó, liên quan đến việc lập trình GPU bằng cách sử dụng API đồ họa và đường ống đồ họa để thực hiện các tác vụ không phải đồ hoạ.

Compute Unifed Device Architecture (CUDA) là một mô hình lập trình song song có thể mở rộng và nền tảng phần mềm dành cho GPU và các bộ xử lý song song khác cho phép lập trình viên bỏ qua đồ hoạ API và giao diện đồ họa của GPU và chỉ cần lập trình bằng C hoặc C ++. Mô hình lập trình CUDA có dạng phần mềm SPMD (một chương trình đa dữ liệu), trong đó lập trình viên viết chương trình cho một luồng được cài đặt sẵn và được thực thi bởi nhiều luồng song song trên nhiều bộ xử lý của GPU. Trên thực tế, CUDA cũng cung cấp một cơ sở để lập trình nhiều lõi CPU, vì vậy CUDA là một môi trường để viết các chương trình song song cho toàn bộ hệ thống máy tính không đồng nhất.

1.2 Kiến trúc hệ thống của GPU

1.2.1 Heterogeneous System

Mặc dù GPU được cho là bộ xử lý song song và mạnh nhất trong một PC thông thường, nhưng chắc chắn nó không phải là bộ xử lý duy nhất. CPU, hiện có đa lõi và sắp trở thành nhiều lõi hơn nữa, là một bộ xử lý nối tiếp bổ sung, chủ yếu kết hợp với GPU nhiều lõi song song khổng lồ. Cùng với nhau, hai loại bộ xử lý này tạo thành một hệ thống đa bộ xử lý không đồng nhất.

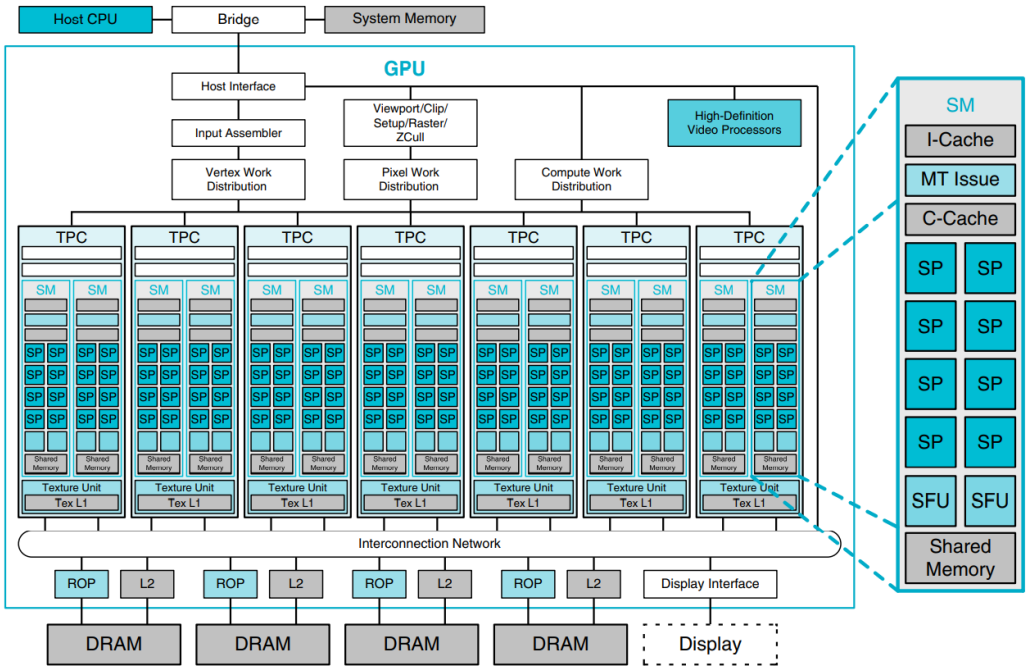
Với hiệu suất gần như tốt nhất cho nhiều ứng dụng đến từ việc sử dụng cả CPU và GPU. Hệ thống này sẽ giúp bạn hiểu cách thức và thời điểm phân chia công việc tốt nhất giữa hai bộ xử lý ngày càng phát triển song song này.

Kiến trúc hệ thống máy tính không đồng nhất sử dụng GPU và CPU có thể được mô tả ở cấp độ cao bằng hai đặc điểm chính: thứ nhất, có bao nhiêu hệ thống con and/or số chip chức năng được sử dụng và công nghệ kết nối và cấu trúc liên kết của chúng là gì; và thứ hai, những hệ thống bộ nhớ nào có sẵn cho các hệ thống con chức năng này.

1.2.2 Processor Array

Một mảng bộ xử lý GPU thống nhất chứa nhiều lõi xử lý, thường được tổ chức thành các bộ xử lý đa luồng. Hình dưới đây cho thấy một GPU với một mảng gồm 112 lõi bộ xử lý phát trực tuyến (SP), được tổ chức thành 14 bộ xử lý đa xử lý phát trực tuyến đa luồng (SM). Mỗi lõi SP có tính đa luồng cao, quản lý 96 luồng đồng thời và trạng thái của chúng trong phần cứng. Các bộ xử lý kết nối với bốn phân vùng DRAM rộng 64 bit thông qua mạng kết nối. Mỗi SM có tám lõi SP, hai đơn vị chức năng đặc biệt (SFU), cấu trúc và bộ nhớ đệm không đổi, một đơn vị cấu trúc đa luồng và một bộ nhớ dùng chung. Đây là cấu trúc Tesla cơ bản được thực hiện bởi

NVIDIA GeForce 8800. Nó có một cấu trúc thống nhất trong đó các chương trình đồ họa truyền thống dành cho đỉnh, hình học và đánh dấu điểm ảnh chạy trên các SM thống nhất và lõi SP của chúng cũng như các chương trình tính toán chạy trên cùng một bộ xử lý.



Hình 1-1 Kiến trúc GPU thống nhất cơ bản

Kiến trúc mảng vi xử lý là khả năng mở rộng để cấu hình GPU nhỏ hơn và lớn hơn bằng cách mở rộng quy mô số lượng multiprocessors và số lượng các phân vùng bộ nhớ. Hình trên cho thấy bảy cụm gồm hai SM chia sẻ một đơn vị cấu trúc và một bộ đệm L1. Đơn vị cấu trúc cung cấp kết quả đã lọc đến SM được cung cấp một tập hợp các tọa độ trong một bản đồ cấu trúc. Vì các vùng được hỗ trợ bởi bộ lọc thường đan xen lẫn nhau cho các yêu cầu kết cấu liên tiếp, một bộ đệm kết cấu trực tuyến nhỏ L1 có hiệu quả để giảm số lượng yêu cầu đến bộ nhớ hệ thống. Mảng bộ xử lý kết nối với bộ xử lý hoạt động raster (ROP), bộ nhớ đệm cấu trúc L2, bộ nhớ DRAM bên ngoài và bộ nhớ hệ thống thông qua mạng kết nối thông GPU. Số lượng bộ xử lý và số lượng bộ nhớ có thể mở rộng để thiết kế hệ thống GPU sao cho cân bằng các phân khúc trị trường và hiệu suất khác nhau.

CHƯƠNG 2. PHÂN CÔNG CÔNG VIỆC

2.1 Công việc chính cần thực hiện

- Tìm hiểu lý thuyết về GPU, tìm hiểu cụ thể hơn về GPU của NVIDIA
- Tìm hiểu về lập trình song song trên GPU của NVIDIA
- Thực hiện triển khai nhân ma trận và đánh giá các chỉ số tính toán
- Thực hiện triển khai bộ lọc trung vị và đánh giá các chỉ số khi thay đổi kích thước bộ lọc và kích thước ảnh
- Kết luận, hướng phát triển trong tương lai

2.2 Nhân lực làm việc

Tên	Điểm mạnh	Điểm yếu
Vũ Đức Thái	Lập trình C/C++, đọc tài liệu, phân chia công việc rõ ràng	Không
Bùi Huy Hoàng	Lập trình, đọc tài liệu tiếng anh	Không
Nguyễn Văn Hoàng	Word, tham khảo tài liệu code.	Lập trình
Nguyễn Văn Quân	Lập trình Python, word, đọc tài liệu tiếng anh, học code nhanh	Không
Nguyễn Hữu Đức	Word, tìm hiểu tài liệu, vẽ minh họa…	Lập trình

Bảng 2-1 Bảng đánh giá nhân lực

2.3 Phân chia công việc

Người thực hiện	Tên công việc	Thời gian thực hiện	Trạng thái
Giai đoạn 1: Tìm hiểu lý thuyết về đề tài			
Cả nhóm	Tìm hiểu đề tài và chọn đề tài.	11/06/2021 – 16/06/2021	Hoàn thành
Nguyễn Hữu Đức	Tìm hiểu chung về bộ xử lý đồ họa (Chương 1).	18/06/2021 – 10/07/2021	Hoàn thành
Nguyễn Văn Hoàng, Bùi Huy Hoàng	Tìm hiểu về kiến trúc phần cứng của bộ xử lý đồ họa CUDA.	20/06/2021 – 05/07/2021	Hoàn thành
Vũ Đức Thái	Tìm hiểu kiến trúc bộ nhớ của CUDA trong lập trình song song; các tham số đánh giá hiệu năng chạy của chương trình.	15/06/2021 – 24/06/2021	Hoàn thành
Nguyễn Văn Quân	Tìm hiểu về ngôn ngữ mở rộng của CUDA.	14/06/2021 – 23/ 06/2021	Hoàn thành
Giai đoạn 2: Triển khai các thuật toán và đánh giá các chương trình			
Vũ Đức Thái, Nguyễn Văn Hoàng	Lập trình triển khai thuật toán nhân ma trận và ghi các tham số đánh giá trên GPU của Jetson Nano	25/06/2021 – 30/06/2021	Hoàn thành
Nguyễn Văn Quân	Vẽ đồ thị đánh giá các tham số của chương trình nhân ma trận	30/06/2021 – 2/07/2021	Hoàn thành
Vũ Đức Thái	Lập trình triển khai thuật toán median filter và ghi các tham số đánh giá trên GPU của Jetson Nano	01/07/2021 – 08/07/2021	Hoàn thành
Bùi Huy Hoàng	Vẽ đồ thị đánh giá các tham số của chương trình median filter	09/07/2021 – 11/07/2021	Hoàn thành

Giai đoạn 3: Viết báo cáo và kết luận (phần triển khai thuật toán)			
Bùi Huy Hoàng, Nguyễn Văn Quân	Phân tích chương trình nhân ma trận, giải thích các độ đo đánh giá	11/07/2021 – 13/07/2021	Hoàn thành
Vũ Đức Thái, Nguyễn Hữu Đức	Phân tích chương trình median filter, giải thích các độ đo đánh giá	09/07/2021 – 14/07/2021	Hoàn thành
Nguyễn Văn Hoàng	Kết luận đề tài, và phương hướng phát triển	14/07/2021 – 16/07/2021	Hoàn thành

Bảng 2-2 Bảng phân công công việc

CHƯƠNG 3. KIẾN TRÚC CUDA – NVIDIA

3.1 Multithreaded Multiprocessor Architecture

Đa xử lý (*multiprocessing*) là việc sử dụng hai hoặc nhiều đơn vị xử lý trung tâm (CPU) trong một hệ thống máy tính. Thuật ngữ này cũng đề cập đến khả năng của một hệ thống hỗ trợ nhiều hơn một bộ xử lý hoặc khả năng phân bổ nhiệm vụ giữa chúng. Có nhiều biến thể về chủ đề cơ bản này và định nghĩa về đa xử lý có thể thay đổi theo ngữ cảnh.

Theo một số từ điển trực tuyến, bộ đa xử lý (*multiprocessor*) là một hệ thống máy tính có hai hoặc nhiều đơn vị xử lý (nhiều bộ xử lý) mỗi bộ chia sẻ bộ nhớ chính và thiết bị ngoại vi, để xử lý đồng thời các chương trình.

Ở cấp độ hệ điều hành, đa xử lý đôi khi được dùng để chỉ việc thực thi nhiều quy trình đồng thời trong một hệ thống, với mỗi quy trình chạy trên một CPU hoặc lõi riêng biệt, trái ngược với một quy trình duy nhất tại bất kỳ thời điểm nào. Khi được sử dụng với định nghĩa này, đa xử lý đôi khi trái ngược với đa nhiệm (*multitasking*), có thể chỉ sử dụng một bộ xử lý duy nhất nhưng chuyển đổi nó theo từng lát thời gian giữa các tác vụ (tức là hệ thống chia sẻ thời gian).

Tại sao lại sử dụng một *multiprocessor*, thay vì nhiều *processor* độc lập? Vì tính song song trong mỗi bộ đa xử lý cung cấp hiệu suất cao và hỗ trợ đa luồng mở rộng cho các mô hình lập trình song song. Các luồng riêng lẻ của một thread block thực thi cùng nhau trong một bộ đa xử lý để chia sẻ dữ liệu.

3.1.1 Massive Multithreading

Bộ xử lý GPU đa luồng (highly multithreaded) để đạt được một số mục tiêu:

- Che đi (*cover*) độ trễ của *memory load* và *texture fetch* từ DRAM
- Hỗ trợ các mô hình lập trình: *fine-grained parallel graphics shader* và *fine-grained parallel computing*
- Ảo hóa bộ xử lý vật lý dưới dạng các luồng và khối luồng để cung cấp khả năng mở rộng minh bạch, rõ ràng.
- Đơn giản hóa mô hình lập trình song song để viết chương trình nối tiếp (*serial program*) cho một luồng

Độ trễ của *memory load và texture fetch* có thể lên đến hàng trăm xung nhịp của bộ xử lý, bởi vì GPU thường có bộ nhớ đệm nhỏ hơn là bộ nhớ đệm của CPU. Một yêu cầu tìm nạp thường yêu cầu độ trễ truy cập DRAM cộng với độ trễ kết nối và bộ đệm. Đa luồng giúp che phủ độ trễ bằng tính toán hữu ích, trong khi một luồng đang đợi tải hoặc tìm nạp hoàn tất, bộ xử lý có thể thực thi một luồng khác.

Chương trình đồ họa *vertex shader* hoặc *pixel shader* là một chương trình cho một luồng xử lý một *vertex* hoặc một *pixel*. Tương tự, một chương trình CUDA là một chương trình C cho một luồng đơn tính toán một kết quả. Các chương trình đồ họa và tính toán khởi tạo nhiều luồng song song để hiển thị các hình ảnh phức tạp và tính toán các mảng kết quả lớn. Để cân bằng động khối lượng công việc của luồng chuyển đổi *vertex* và *pixel shader*, mỗi *multiprocessor* đồng thời thực thi nhiều *thread programs* và nhiều loại *shader program* khác nhau.

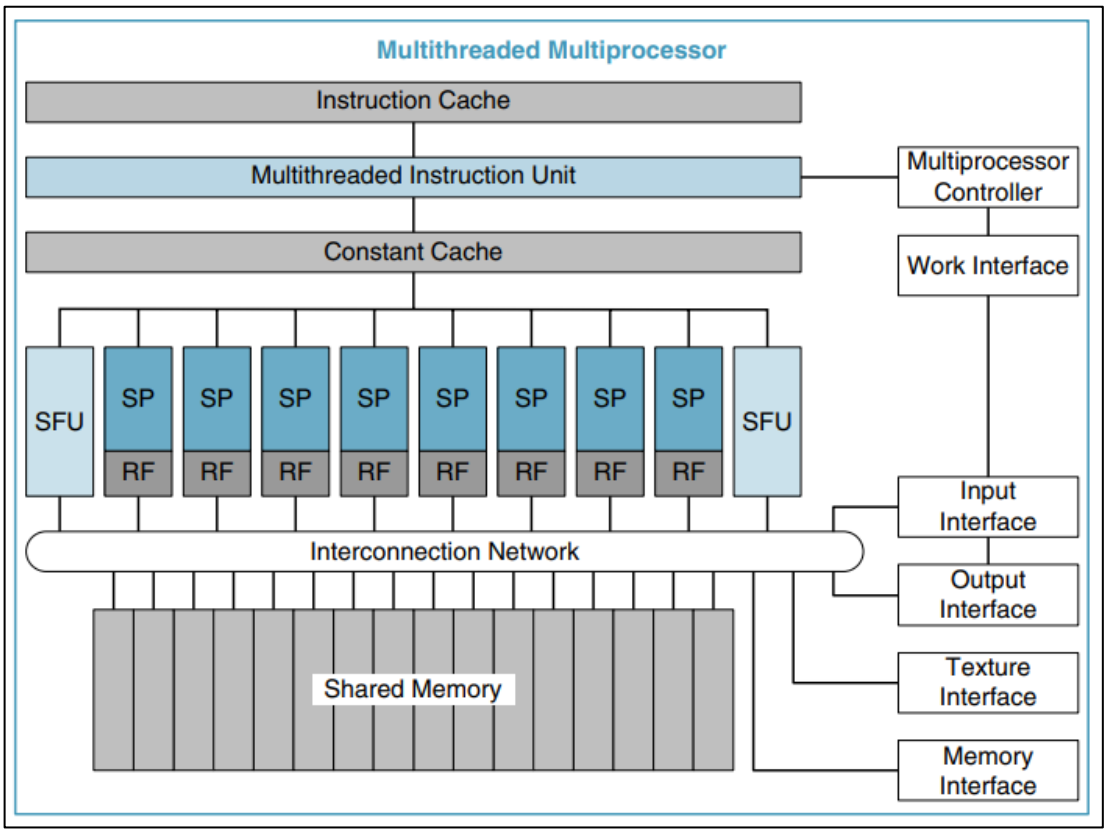
Để hỗ trợ các mô hình lập trình độc lập: *vertex*, *primitive*, và *pixe*l của các *graphics shading language* và mô hình lập trình đơn luồng của CUDA C / C + +, mỗi luồng GPU có các thanh ghi riêng, bộ nhớ riêng cho mỗi luồng, bộ đếm chương trình và luồng trạng thái thực thi và có thể thực thi một đường dẫn mã độc lập. Để thực thi hiệu quả hàng trăm *lightweight threads* đồng thời, bộ xử lý đa xử lý GPU phải là phần cứng đa luồng — nó quản lý và thực thi hàng trăm luồng đồng thời trong phần cứng mà không cần lên lịch chi phí. Các luồng đồng thời trong các khối luồng có thể đồng bộ hóa tại một *barrier* với một lệnh đơn (*single instruction*). Tính năng tạo *lightweight threads*, lập lịch luồng không chi phí và *fast barrier synchronization* hỗ trợ rất hiệu quả *fine-grained parallelism*.

3.1.2 Kiến trúc của bộ đa xử lý (Multiprocessor Architecture)

Một bộ xử lý đa xử lý đồ họa và điện toán thống nhất thực thi các chương trình tạo đỉnh(*vertex*), hình học(*geometry*) và phân mảnh *pixel* cũng như các chương trình tính toán song song. Như *Hình 3.1* cho thấy, bộ đa xử lý bao gồm tám lõi của bộ xử lý vô hướng (Scalar Processor - SP), mỗi lõi có một tệp thanh ghi đa luồng lớn (Register File - RF), hai đơn vị chức năng đặc biệt (Special Function Units - SFU), một đơn vị lệnh đa luồng, một bộ đệm lệnh, một bộ đọc chỉ bộ nhớ cache không đổi và một bộ nhớ được chia sẻ.

16KB *shared memory* chứa bộ đệm dữ liệu đồ họa (*graphics data buffers*) và *shared computing data*. Các biến CUDA được khai báo là __shared__ nằm trong *shared memory.*

Mỗi lõi SP chứa các đơn vị số học dấu phẩy động và số nguyên vô hướng thực hiện hầu hết các lệnh. SP là phần cứng đa luồng, hỗ trợ lên đến 64 luồng. Mỗi lõi SP có một RF lớn gồm 1024 thanh ghi 32-bit mục đích chung, được phân vùng giữa các luồng được chỉ định của nó. Các chương trình khai báo nhu cầu đăng ký của chúng, thường là 16 đến 64 thanh ghi 32-bit vô hướng cho mỗi luồng. SP có thể chạy đồng thời nhiều luồng sử dụng một vài thanh ghi hoặc ít luồng sử dụng nhiều thanh ghi hơn. Trình biên dịch tối ưu hóa việc phân bổ thanh ghi để cân bằng giữa chi phí tràn thanh ghi so với chi phí của ít luồng hơn. Các chương trình *pixel shader* thường sử dụng 16 thanh ghi trở xuống, cho phép mỗi SP chạy tối đa 64 luồng *pixel shader* để *cover* các lần tìm nạp *texture* có độ trễ dài. Các chương trình CUDA đã biên dịch thường cần 32 thanh ghi cho mỗi luồng, giới hạn mỗi SP là 32 luồng, điều này giới hạn một *kernel program* như vậy là 256 luồng cho mỗi khối luồng trên bộ xử lý đa năng.



Hình 3-1 Bộ xử lý đa luồng với 8 lõi của bộ xử lý vô hướng (Scalar Processor-SP)

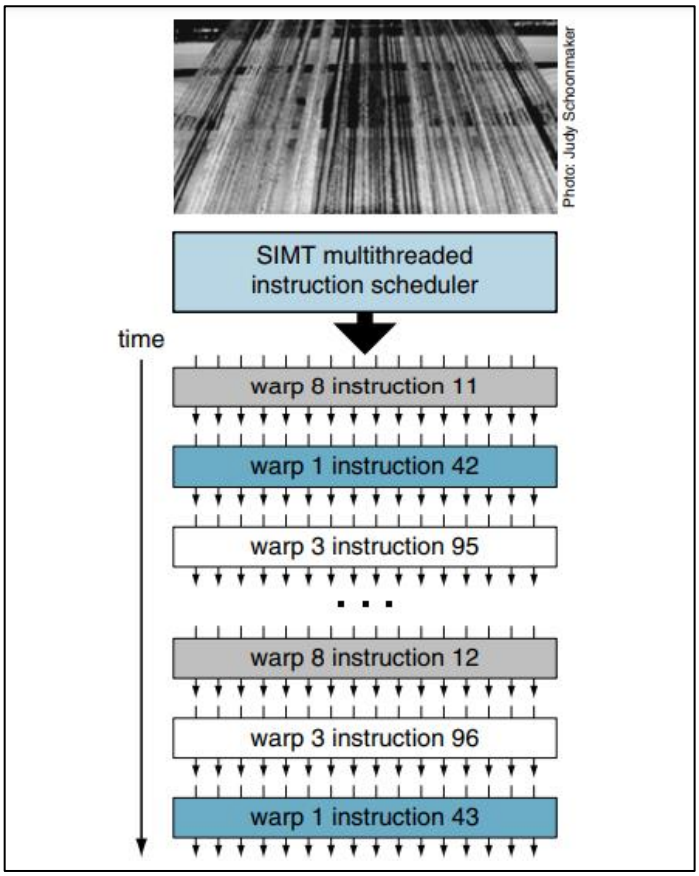
Tám lõi SP mỗi lõi có một tệp đăng ký đa luồng lớn (RF) và chia sẻ một *instruction cache, multithreaded instruction unit,* bộ nhớ đệm không đổi (*constant cache*), hai đơn vị chức năng đặc biệt (SFU), mạng kết nối và *shared memory.*

Các SFU pipelined thực thi các luồng tính toán những hàm đặc biệt và nội suy các thuộc tính pixel từ các thuộc tính *primitive vertex*. Các lệnh này có thể thực hiện đồng thời với các lệnh trên SP.

Bộ đa xử lý thực hiện các lệnh tìm nạp *texture* trên *texture unit* thông qua *texture interface* và sử dụng giao diện bộ nhớ cho các lệnh tải, lưu trữ và *atomic access* bộ nhớ ngoài. Các lệnh này có thể thực hiện đồng thời với các lệnh trên SP. Truy cập *shared memory* sử dụng mạng kết nối có độ trễ thấp giữa bộ xử lý SP và ngân hàng bộ nhớ dùng chung (*shared memory banks*.)

3.1.3 Single-Instruction Multiple-Thread (SIMT)

Để quản lý và thực thi hàng trăm luồng chạy một số chương trình khác nhau một cách hiệu quả, bộ đa xử lý sử dụng kiến trúc đa luồng một lệnh (SIMT). Nó tạo, quản lý, lập lịch và thực thi các luồng đồng thời trong các nhóm luồng song song được gọi là *warps*. Thuật ngữ *warp* bắt nguồn từ dệt, công nghệ sợi song song đầu tiên. Ảnh trong *Hình 3.2* cho thấy một *warp* của các sợi song song xuất hiện từ khung dệt. Mẫu đa xử lý này sử dụng kích thước *warp* SIMT gồm 32 luồng, thực thi bốn luồng trong mỗi tám lõi SP trên bốn xung nhịp. Các khối thread bao gồm một hoặc nhiều *warps.*



Hình 3-2 Lập lịch warp đa luồng SIMT

Bộ đa xử lý SIMT quản lý một nhóm gồm 16 *warps*, tổng số 512 luồng. Các luồng song song riêng lẻ tạo thành một warp là cùng một kiểu và bắt đầu cùng nhau tại cùng một địa chỉ chương trình, chúng có thể tự do phân nhánh và thực thi độc lập. Tại mỗi thời điểm *instruction issue time, the SIMT multithreaded instruction unit* chọn một *warp* đã sẵn sàng để thực hiện lệnh tiếp theo của nó, rồi cấp lệnh đó cho các luồng đang hoạt động của *warp* đó. Trong bộ xử lý đa năng này, mỗi lõi bộ xử lý vô hướng SP thực hiện một lệnh cho bốn luồng riêng lẻ của *warp* sử dụng bốn xung nhịp, phản ánh tỷ lệ 4: 1 của luồng *warp* so với lõi.

Kiến trúc bộ xử lý SIMT giống với thiết kế đa dữ liệu một lệnh (Single Instruction Multiple Data - SIMD), áp dụng một lệnh cho nhiều làn dữ liệu, nhưng khác ở chỗ SIMT áp dụng một lệnh cho nhiều luồng độc lập song song, không chỉ cho nhiều làn dữ liệu. Một lệnh cho bộ xử lý SIMD điều khiển một vectơ gồm nhiều làn dữ liệu cùng nhau, trong khi lệnh cho bộ xử lý SIMT điều khiển một luồng riêng lẻ và đơn vị lệnh SIMT đưa ra lệnh cho một *warp* song song độc lập để đạt hiệu quả.

Bộ xử lý SIMT đạt hiệu quả và hiệu suất lớn nhất khi tất cả các luồng của *warp* có cùng một đường dẫn thực thi. Nếu các luồng của một *warp* phân kỳ qua một nhánh có điều kiện phụ thuộc vào dữ liệu, việc thực thi sẽ được tuần tự hóa, từng đường dẫn nhánh sẽ được thực hiện cho đến khi tất cả các đường dẫn hoàn tất, các luồng sẽ hội tụ đến cùng một đường dẫn thực thi. Đối với các đường dẫn có độ dài bằng nhau, khối mã if-else phân kỳ có hiệu suất 50%. Bộ đa xử lý sử dụng ngăn xếp đồng bộ hóa nhánh để quản lý các luồng độc lập phân kỳ và hội tụ.

Ngược lại với kiến trúc vector SIMD, SIMT cho phép lập trình viên viết mã song song *thread-level* cho các luồng độc lập riêng lẻ, cũng như mã *data-parallel* cho nhiều luồng phối hợp(*coordinated threads*).

3.1.4 Sự phân kỳ và thực thi warp SIMT (SIMT Warp Execution and Divergence)

Phương pháp SIMT lập lịch cho các *warp* độc lập linh hoạt hơn cách lập lịch cho các kiến trúc GPU trước đây. Một *warp* bao gồm các *thread* song song cùng loại: *vertex, geometry, pixel* hoặc *computing*. Đơn vị cơ bản của *pixel fragment shader processing* là bộ tứ *pixel 2 x 2* được triển khai dưới dạng bốn *pixel shader threads*. Bộ điều khiển đa xử lý đóng gói các đơn vị *pixel* thành một *warp*. Tương tự, nó nhóm các *vertex* và *primitive* thành các *warp*, và đóng gói các *computing thread* thành một warp. Một *thread block* bao gồm một hoặc nhiều sợi *warp*. Thiết kế SIMT chia sẻ *instruction fetch* và *issue unit* một cách hiệu quả trên các luồng song song của một *warp,* một *warp* đạt hiệu suất tối đa khi tất cả các luồng của nó đều hoạt động.

Bộ đa xử lý hợp nhất này lên lịch và thực thi đồng thời nhiều loại *warp*, cho phép nó thực hiện đồng thời các *warp (vertex* và *pixel)*. Bộ lập lịch *warp* hoạt động ở mức thấp hơn tốc độ xung nhịp của bộ xử lý, vì có bốn *thread* trên mỗi lõi bộ xử lý. Trong mỗi chu kỳ lập lịch, nó sẽ chọn một *warp* để thực hiện lệnh, như thể hiện trong *Hình 3.2*. Một lệnh được thực thi dưới dạng bốn bộ tám luồng (*four sets of eight threads*) trong bốn chu kỳ thông lượng của bộ xử lý. *Processor pipeline* sử dụng một số *clock of latency* để hoàn thành mỗi lệnh. Nếu số lượng *active warp* nhân với số *clock* trên mỗi *warp* vượt quá độ trễ của đường ống (*pipeline*), lập trình viên có thể bỏ qua độ trễ của đường ống. Đối với bộ đa xử lý này, một *round-robin schedule* của tám *warp* có khoảng thời gian 32 chu kỳ giữa các lệnh liên tiếp cho cùng một *warp*. Nếu chương trình có thể giữ 256 luồng hoạt động cho mỗi bộ đa xử lý, thì độ trễ lệnh lên đến 32 chu kỳ có thể được bỏ qua. Tuy nhiên, với ít

active warp, trễ đường ống của bộ xử lý sẽ không bị bỏ qua và có thể khiến bộ xử lý bị đình trệ.

3.1.5 Quản lý luồng và khối luồng

Bộ điều khiển đa xử lý và *instruction unit* quản lý các luồng và khối luồng. Bộ điều khiển chấp nhận các yêu cầu công việc và dữ liệu đầu vào, phân xử quyền truy cập vào các tài nguyên được chia sẻ, bao gồm đơn vị kết cấu, đường dẫn truy cập bộ nhớ và đường dẫn I/O. Đối với khối công việc đồ họa, nó tạo và quản lý đồng thời ba loại luồng đồ họa: *vertex*, *geometry* và *pixel*. Mỗi loại công việc đồ họa có các đường dẫn đầu vào và đầu ra độc lập. Nó tích lũy và đóng gói từng loại công việc đầu vào này thành SIMT *warp* của các luồng song song thực hiện cùng một chương trình luồng. Nó cấp phát *warp*, cấp phát thanh ghi cho các luồng trong *warp* và bắt đầu thực thi *warp* trong bộ đa xử lý. Mỗi chương trình khai báo nhu cầu đăng ký mỗi luồng của nó; bộ điều khiển chỉ bắt đầu một *warp* khi nó có thể cấp phát số lượng thanh ghi được yêu cầu cho các luồng trong *warp*. Khi tất cả các luồng của *warp* "exit", bộ điều khiển giải nén kết quả và giải phóng các thanh ghi trong *warp* và tài nguyên.

Bộ điều khiển tạo mảng luồng hợp tác (Cooperative Thread Arrays - CTAs) thực hiện các khối luồng CUDA dưới dạng một hoặc nhiều *warp* của các luồng song song. Ngoài các luồng và thanh ghi, CTA yêu cầu phân bổ bộ nhớ dùng chung và các rào cản (*barriers*). Chương trình khai báo các điều kiện cần thiết và bộ điều khiển đợi cho đến khi nó có thể thỏa mãn điều kiện trước khi khởi chạy CTA. Sau đó, nó tạo CTA ở tốc độ lập lịch *warp*, để chương trình CTA bắt đầu thực thi ngay lập tức với hiệu suất tối đa của bộ đa xử lý. Bộ điều khiển giám sát khi tất cả các luồng của CTA đã "*exit*", nó sẽ giải phóng các tài nguyên.

3.1.6 Thread Instructions

Các bộ xử lý luồng SP thực thi các lệnh vô hướng cho các luồng riêng lẻ. Các chương trình CUDA C / C ++ chủ yếu có mã vô hướng trên mỗi luồng. Các GPU trước đây sử dụng tính năng đóng gói vector, nhưng điều đó làm phức tạp phần cứng lập lịch cũng như trình biên dịch. Hướng dẫn vô hướng (*Scalar instructions*) đơn giản hơn và thân thiện với trình biên dịch. Hướng dẫn kết cấu (*Texture instructions*) vẫn dựa trên vector, lấy vector tọa độ nguồn và trả về vector màu đã lọc.

3.1.7 Streaming Processor (SP)

Lõi bộ xử lý luồng đa luồng (SP) là bộ xử lý chính trong bộ đa xử lý. Tệp đăng ký (RF) của nó cung cấp 1024 thanh ghi 32-bit vô hướng cho tối đa 64 luồng. Nó thực thi tất cả các phép toán dấu phẩy động cơ bản, bao gồm *add.f32, mul.f32, mad.f32* (floating multiply-add)), *min. f32, max.f32* và *setp.f32* (floating compare and set predicate). Các phép toán cộng và nhân dấu phẩy động tương thích với tiêu chuẩn IEEE 754 cho các số FP chính xác đơn, bao gồm các giá trị không phải một số (*NaN*) và vô cực. Lõi SP cũng thực hiện tất cả các lệnh số học, so sánh, chuyển đổi và PTX logic 32 bit và 64 bit.

Các phép toán dấu phẩy động *add* và *mul* sử dụng IEEE làm tròn đến gần nhất làm chế độ làm tròn mặc định. Phép toán nhân-cộng dấu phẩy động *mad.f32* thực hiện một phép nhân với phép cắt bớt, theo sau là phép cộng với số làm tròn đến gần nhất.

3.1.8 Đơn vị chức năng đặc biệt (Special Function Unit - SFU)

Một số *thread instruction* nhất định có thể thực thi trên các SFU, đồng thời với các *thread instruction* khác đang thực thi trên SP. SFU thực hiện các lệnh đặc biệt, tính toán các phép xấp xỉ dấu phẩy động 32-bit thành các hàm siêu việt đối ứng, căn bậc hai nghịch đảo và hàm siêu việt khóa. Nó cũng thực hiện nội suy thuộc tính phẳng 32-bit dấu phẩy động cho các *pixel shader,* cung cấp nội suy chính xác các thuộc tính như tọa độ màu, độ sâu và kết cấu.

Mỗi *SFU pipelined* tạo ra một kết quả dấu phẩy động 32 bit cho mỗi chu kỳ. Các SFU cũng thực hiện lệnh nhân *mul.f32* đồng thời với tám SP, tăng tốc độ tính toán cao nhất lên đến 50% cho các luồng có hỗn hợp lệnh phù hợp.

3.1.9 So sánh với các bộ đa xử lý khác

So với các kiến trúc vectơ SIMD như x86 SSE, bộ đa xử lý SIMT có thể thực thi các luồng riêng lẻ một cách độc lập, thay vì luôn thực thi chúng cùng nhau trong các nhóm đồng bộ. Phần cứng SIMT tìm sự song song dữ liệu giữa các luồng độc lập, trong khi phần cứng SIMD yêu cầu phần mềm thể hiện tính song song dữ liệu một cách rõ ràng trong mỗi lệnh vectơ. Một máy SIMT thực hiện đồng bộ 32 luồng khi các luồng có cùng một đường dẫn thực thi, nhưng có thể thực thi từng luồng một cách độc lập khi chúng phân kỳ. Ưu điểm là đáng kể vì các chương trình và câu lệnh SIMT chỉ đơn giản mô tả hoạt động của một luồng độc lập, thay vì một vectơ dữ liệu SIMD của bốn hoặc nhiều làn dữ liệu. Tuy nhiên, bộ đa xử lý SIMT có hiệu suất giống như SIMD, trải rộng diện tích và chi phí của

một đơn vị lệnh trên 32 luồng của một *warp* và trên tám lõi của *streaming processor* (SP). SIMT cung cấp hiệu suất của SIMD cùng với năng suất của đa luồng, tránh phải mã hóa rõ ràng các vectơ SIMD cho các điều kiện cạnh (*edge condition)* và phân kỳ một phần (*partial divergence*).

Bộ đa xử lý SIMT đặt ra ít chi phí vì nó là phần cứng đa luồng với đồng bộ hóa rào cản phần cứng. Điều đó cho phép các *graphics shader* và các luồng CUDA thể hiện sự song song rất chi tiết (*very fine-grained*). Các chương trình đồ họa và CUDA sử dụng các luồng để thể hiện tính song song của dữ liệu trong một chương trình, thay vì buộc người lập trình phải thể hiện nó dưới dạng các lệnh vector SIMD. Sẽ đơn giản và hiệu quả hơn khi phát triển mã đơn luồng vô hướng hơn là mã vectơ và bộ đa xử lý SIMT thực thi mã với hiệu quả giống như SIMD.

Ghép tám lõi SP với nhau chặt chẽ thành một bộ đa xử lý và sau đó triển khai một số lượng đa xử lý có thể mở rộng như vậy tạo thành một bộ đa xử lý hai cấp bao gồm nhiều bộ xử lý. Mô hình lập trình CUDA khai thác hệ thống phân cấp hai cấp bằng cách cung cấp các luồng riêng lẻ cho các phép tính song song chi tiết và bằng cách cung cấp lưới (*grid*) các khối luồng cho các hoạt động song song. Cùng một *thread program* có thể cung cấp cả hoạt động chi tiết và thô (*fine-grained and coarse-grained*). Ngược lại, các CPU có lệnh vectơ SIMD phải sử dụng hai mô hình lập trình khác nhau để làm điều này (các luồng song song *coarse-grained* trên các lõi khác nhau và lệnh vectơ SIMD cho song song dữ liệu *fine-grained)*

3.1.10 Kết luận đa xử lý đa luồng (Multithreaded Multiprocessor)

GPU multiprocessor dựa trên kiến trúc Tesla có tính đa luồng cao, thực thi tổng cộng lên đến 512 luồng nhẹ đồng thời để hỗ trợ trình tạo bóng pixel chi tiết (*fine-grained pixel shaders*) và *CUDA threads*. Nó sử dụng một biến thể trên kiến trúc SIMD và đa luồng được gọi là SIMT để phát hiệu quả một lệnh tới một chuỗi 32 luồng song song, đồng thời cho phép mỗi luồng phân nhánh và thực thi độc lập. Mỗi luồng thực thi lệnh của nó trên một trong tám lõi của SP, được xử lý đa luồng lên đến 64 luồng.

3.2 Parallel memory system

3.2.1 Introduction

Trong chương này, chúng ta sẽ phân tích về những ưu và nhược của mỗi loại bộ nhớ trong kiến trúc CUDA của NVIDIA. Global memory có dung lượng địa chỉ lớn, nhưng độ

trễ truy cập vào vùng nhớ này rất cao. Shared memory có băng thông lớn hơn, độ trễ thấp hơn nhưng bù lại thì dung lượng địa chỉ nhỏ hơn so với global memory. Từ những hiểu biết về mỗi loại bộ nhớ của CUDA chúng ta tăng hiệu suất làm việc của chương trình bằng việc tận dụng các loại bộ nhớ khác nhau, và đánh đổi giữa chúng.

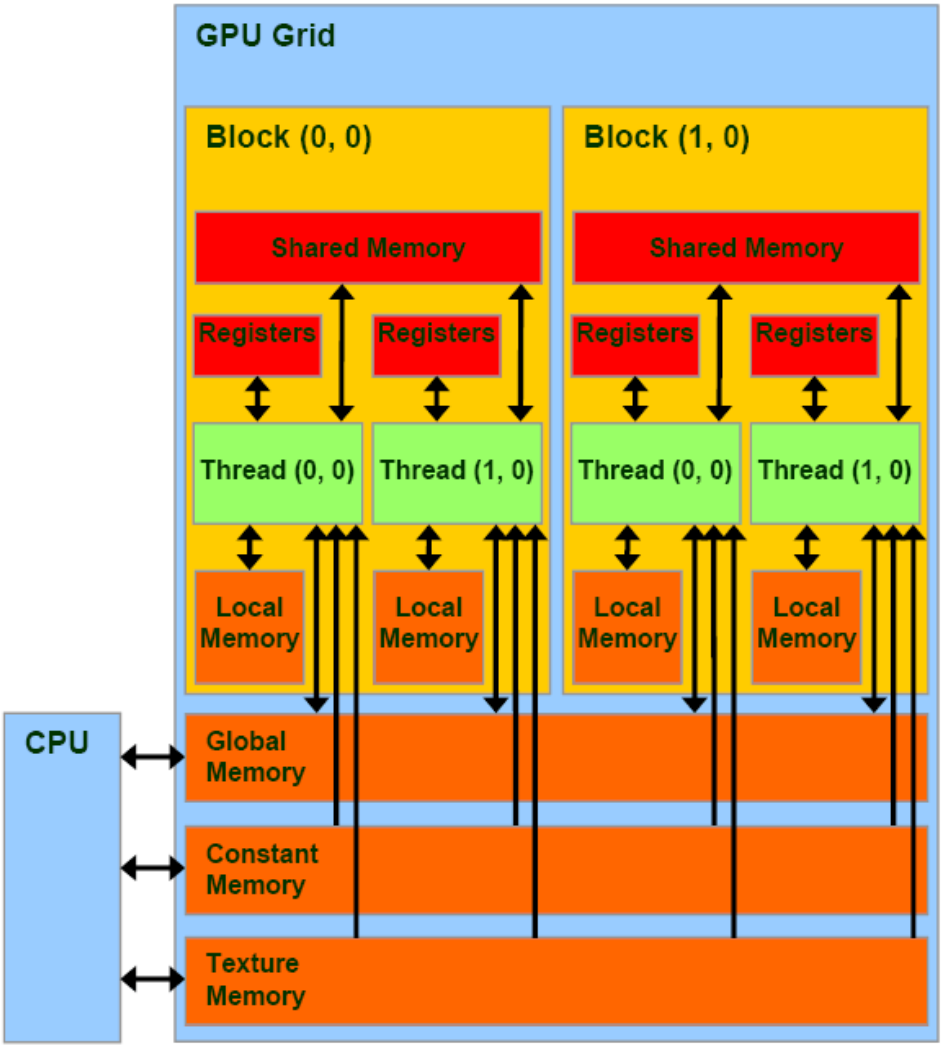
3.2.2 CUDA memory types

Mỗi GPU hỗ trợ CUDA cung cấp một số loại bộ nhớ khác nhau. Mỗi loại bộ nhớ khác nhau này có các thuộc tính khác nhau như độ trễ truy cập (access latency), không gian địa chỉ (address space), phạm vi (scope) và thời gian tồn tại (lifetime).

Các loại bộ nhớ được thể hiện như hình dưới gồm có: register, shared, local, global and constant memory. Trên các thiết bị hỗ trợ CUDA phiên bản 1.x, có hai vị trí mà bộ nhớ có thể có; cache memory, device memory.

Device code:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory



Hình 3-3 Cấu trúc hệ thống bộ nhớ trong GPU của NVIDIA

3.2.2.1 Register

Những biến (scalar variables) được khai báo trong phạm vi của một hàm kernel và không được khai báo với bất kỳ thuộc tính nào được lưu trữ trong register memory theo mặc định. Truy cập register memory rất nhanh, nhưng số lượng register có sẵn trên mỗi block bị hạn chế.

Một mảng khai báo ở trong kernel function cũng có thể được chứa ở trong register nhưng chỉ khi truy cập các phần tử mảng thông qua chỉ số mảng (được xác định trong quá trình biên dịch – compile time). Nó không có khả năng thực thi truy cập ngẫu nhiên vào các register variables.

Các register vairiable thì riêng tư với thread. Các thread trong cùng một block sẽ nhận được các phiên bản riêng của mỗi biến thanh ghi. Các biến thanh ghi chỉ tồn tại miễn là thread chứa nó vẫn đang thực thi. Khi thread kết thúc thực thi, các biến thanh ghi được

giải phóng và không thể truy cập lại. Mỗi lệnh gọi kernel function phải khởi tạo các biến trong thời gian gọi. Điều này có vẻ hiển nhiên vì phạm vi của biến nằm trong kernel function, nhưng điều này không đúng với tất cả các biến được khai báo trong kernel function, chúng ta sẽ đề cập tới shared memory ở phía sau.

Các biến được khai báo trong register memory có thể đọc ghi trong một kernel function. Việc thực hiện đọc ghi này thì không cần được đồng bộ hóa

Chú ý: Tốc độ thực thi trên thanh ghi là cao nhất, cao hơn với việc truy cập vào shared memory vì việc truy cập vào shared memory cũng phải được thực hiện bằng lệnh đọc/ghi thông qua các lệnh trong thanh ghi => Tốn thời gian hơn so với việc đọc trực tiếp từ thanh ghi.

3.2.2.2 Local memory

Bất kỳ biến nào không thể vừa với không gian register memory, kernel sẽ lưu nó vào local memory. Bộ nhớ cục bộ có cùng độ trễ truy cập như bộ nhớ chung (có nghĩa là chậm).

Giống như các register, local memory thì private đối với thread. Mỗi thread phải khởi tạo nội dung của một biến được lưu trữ trong local memory trước khi nó được sử dụng. Chúng ta không thể dựa vào một thread khác (ngay cả trong cùng một block) để khởi tạo local memory vì nó là private đối với mỗi thread.

Các biến trong local memory có thời gian tồn tại cùng với thread, khi kết thúc thời gian thực thi, biến local không còn khả năng truy cập được nữa. Các biến trong local memory có thể được đọc/ghi trong hàm kernel, việc truy cập này không cần thiết phải đồng bộ hóa. Chúng ta không cần phải khai báo với macro nào, compiler sẽ tự động chuyển giá trị của nó vào local memory, với điều kiện:

- Mảng được truy cập bằng chỉ mục trong thời gian chạy. Nghĩa là, trình biên dịch không thể xác định các chỉ số tại thời điểm biên dịch.
- Các cấu trúc hoặc mảng lớn sẽ tiêu tốn quá nhiều không gian đăng ký.
- Bất kỳ biến nào được khai báo vượt quá số lượng thanh ghi cho nhân đó (điều này được gọi là tràn thanh ghi).

Cách duy nhất ta có thể xác định xem trình biên dịch có đặt một số biến phạm vi hàm trong bộ nhớ cục bộ hay không là kiểm tra thủ công mã lắp ráp PTX (có được bằng

cách biên dịch với tùy chọn -ptx hoặc -keep). Các biến local sẽ được khai báo bằng cách sử dụng .local mnemonic và được tải bằng cách sử dụng ld.local mnemonic và được lưu trữ bằng st.local mnemonic.

3.2.2.3 Shared memory

Các biến được khai báo với macro "__shared__" được lưu trữ trong shared memory. Truy cập bộ nhớ dùng chung rất nhanh (nhanh hơn ~ 100 lần so với global memory) mặc dù mỗi SM (Streaming Multiprocessor) có một lượng giới hạn không gian địa chỉ shared memory.

Shared memory phải được khai báo trong phạm vi của kernel function nhưng có thời gian tồn tại cùng với một thread block (trái ngược với register hoặc local memory có thời gian tồn tại cùng với thread). Khi một thread block được thực thi xong, không thể truy cập shared memory đã được cấp phát trong kernel function nữa.

Shared memory có thể được đọc và ghi bên trong kernel function. Việc thay đổi shared memory phải được đồng bộ hóa nếu chúng ta không đảm bảo rằng mỗi luồng sẽ chỉ truy cập vào bộ nhớ sẽ không được đọc từ hoặc ghi vào bởi các thread khác trong cùng một thread block. Đồng bộ hóa khối được thực hiện bằng cách sử dụng hàm __syncthreads() bên trong kernel function.

Vì quyền truy cập vào shared memory nhanh hơn truy cập vào global memory, nên sẽ hiệu quả hơn nếu sao chép một phần từ global memory sang shared memory để sử dụng trong kernel function giúp số lượng truy cập vào bộ nhớ chung có thể giảm trong mỗi thread block.

3.2.2.4 Global memory

Các biến được khai báo với macro là "__device__" và được khai báo trong phạm vi của global memory (bên ngoài phạm vi của kernel function) và được lưu trữ trong global memory. Độ trễ truy cập vào global memory rất cao (chậm hơn ~ 100 lần so với shared memory) nhưng bù lại thì có nhiều không gian lưu trữ hơn so với shared memory (lên đến 6GB nhưng kích thước thực tế khác nhau giữa các card đồ họa ngay cả khi có cùng phiên bản CUDA).

Không giống như register, local memory và shared memory, global memory có thể được đọc và ghi vào sử dụng C-function *cudaMemcpy*. Global memory có thời gian tồn tại

cùng với ứng dụng và có thể truy cập vào tất cả các threads của tất cả các kernel function. Một điểm phải cẩn thận là khi đọc/ghi vào global memory thì việc thực thi luồng không thể được đồng bộ hóa trên các thread block khác nhau. Cách duy nhất để đảm bảo quyền truy cập vào global memory được đồng bộ là bằng cách gọi các lệnh kernel riêng biệt (chia nội dung thành các kernel khác nhau và đồng bộ hóa thông qua host giữa các lệnh gọi kernel function).

Global memory được khai báo trên tiến trình host bằng hàm cudaMalloc và được giải phóng bộ nhớ sử dụng hàm cudaFree. Các con trỏ tới global memory có thể được chuyển tới một kernel function như là các tham số cho kernel.

3.2.2.5 Constant memory

Các biến được khai báo sử dụng macro "__constant__" được lưu trong constant memory. Giống như các global variables, các biến constant variables phải được khai báo trong phạm vi toàn cục (bên ngoài phạm vi của bất kỳ kernel function nào). Các constant variables chia sẻ cùng một vùng lưu trữ giống như global memory (device memory) nhưng không giống như global memory, chỉ có một lượng giới hạn của constant memory có thể được khai báo (64KB trên tất cả các phiên bản CUDA).

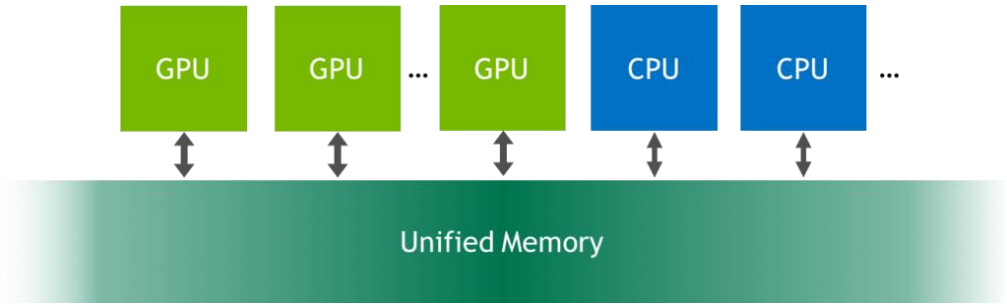
Độ trễ truy cập vào constant memory nhanh hơn đáng kể so với global memory vì constant memory được lưu trong bộ nhớ đệm không giống như global memory, constant memory không thể được ghi vào từ bên trong kernel function. Điều này cho phép bộ nhớ đệm không đổi hoạt động của mình vì chúng ta muốn được đảm bảo rằng các giá trị trong constant memory sẽ không bị thay đổi và do đó sẽ không bị vô hiệu trong quá trình thực thi một kernel function.

Constant memory có thể được ghi vào bởi tiến trình từ phía *host* bằng cách sử dụng hàm *cudaMemcpyToSymbol* và đọc từ constant memory bằng cách sử dụng hàm *cudaMemcpyFromSymbol.* Từ những đặc trưng trên, chúng ta sẽ không thể cấp phát động lưu trữ trong constant memory (kích thước của *constant memory buffers* phải được khai báo tĩnh).

Giống như bộ nhớ toàn cục, bộ nhớ không đổi có thời gian tồn tại cùng với ứng dụng. Nó có thể được truy cập bởi tất cả các threads của tất cả các kernels và giá trị sẽ không thay đổi qua các lệnh gọi kernel, trừ khi có những sửa đổi một cách rõ ràng từ phía *host.*

3.2.3 Unified memory

Trong phần này chúng ta sẽ nói về unified memory , giúp dễ dàng phân bổ và truy cập dữ liệu có thể được sử dụng bằng mã chạy trên bất kỳ bộ xử lý nào trong hệ thống, CPU hoặc GPU.



Hình 3-4 Minh họa cơ chế Unified Memory

3.2.3.1 Khái niệm Unified Memory

Unified memory là một không gian địa chỉ bộ nhớ duy nhất có thể truy cập được từ bất kỳ bộ xử lý nào trong hệ thống. Công nghệ phần cứng/phần mềm này cho phép các ứng dụng phân bổ dữ liệu có thể đọc hoặc ghi từ mã chạy trên CPU hoặc GPU. Phân bổ unified memory đơn giản như việc thay thế các lệnh gọi đến *malloc()* hoặc *new* bằng các lệnh gọi đến *cudaMallocManaged()*, một hàm cấp phát trả về một con trỏ có thể truy cập từ bất kỳ bộ xử lý nào

```
cudaError_t cudaMallocManaged(void** ptr, size_t size);
```

Khi mã chạy trên CPU hoặc GPU truy cập dữ liệu được phân bổ theo cách này (thường được gọi là CUDA *managed* data), phần mềm hệ thống CUDA và/hoặc phần cứng sẽ xử lý việc di chuyển các *memory pages* sang bộ nhớ của bộ xử lý truy cập. Điểm quan trọng ở đây là kiến trúc GPU Pascal là kiến trúc đầu tiên hỗ trợ phần cứng cho *virtual memory page faulting and migration*, thông qua *Page Migration Engine*. Các GPU cũ hơn dựa trên kiến trúc Kepler và Maxwell cũng hỗ trợ nhưng chỉ là một phiên bản giới hạn hơn của Unified Memory.

CHƯƠNG 4. LẬP TRÌNH CUDA

4.1 Mở rộng ngôn ngữ C++ trong lập trình CUDA

4.1.1 Function Execution Space Specifiers

Function Execution Space Specifiers biểu thị liệu một chức năng có thực thi trên máy chủ hay trên thiết bị hay không và liệu nó có thể được gọi từ máy chủ hay từ thiết bị hay không.

4.1.1.1 __global__

Để thực thi __global__ ta khai báo một hàm như là một kernel. Một chức năng như vậy được:

‣ Thực hiện trên thiết bị,

‣ Có thể gọi từ máy chủ,

‣ Có thể gọi từ thiết bị cho các thiết bị có khả năng tính toán 3.2 trở lên (xem Song song động CUDA để biết thêm chi tiết).

Hàm __global__ phải có kiểu trả về void và không được là thành viên của một lớp.

Bất kỳ lệnh gọi nào đến hàm __global__ phải chỉ định cấu hình thực thi của nó như được mô tả trong Cấu hình thực thi.

Lệnh gọi hàm __global__ là không đồng bộ, có nghĩa là nó trả về trước khi thiết bị hoàn thành việc thực thi

4.1.1.2 __device__

__device__ execution space specifier khai báo 1 hàm:

‣ Thực hiện trên thiết bị,

‣ Chỉ có thể gọi từ thiết bị.

__global__ and __device__ execution space specifiers không thể được sử dụng cùng nhau

4.1.1.3 __host__

__host__ execution space specifier khai báo 1 hàm:

‣ Thực hiện trên máy chủ,

‣ Chỉ có thể gọi từ máy chủ.

Nó tương đương với việc khai báo một hàm chỉ có __host__ execution space specifier hoặc khai báo nó mà không có bất kỳ __host__, __device__ hoặc __global__ execution space specifier nào; trong cả hai trường hợp, chức năng chỉ được biên dịch cho máy chủ lưu trữ. Các __global__ và __host__ execution space specifier không thể được sử dụng cùng nhau. Tuy nhiên, các __device__ và __host__ execution space specifier có thể được sử dụng cùng nhau, trong trường hợp đó, hàm được biên dịch cho cả máy chủ và thiết bị. Macro __CUDA_ARCH__ được giới thiệu trong Application Compatibility có thể được sử dụng để phân biệt các đường dẫn mã giữa máy chủ và thiết bị:

```
__host__ __device__ func()
{
#if __CUDA_ARCH__ >= 800
    // Device code path for compute capability 8.x
#elif __CUDA_ARCH__ >= 700
    // Device code path for compute capability 7.x
#elif __CUDA_ARCH__ >= 600
    // Device code path for compute capability 6.x
#elif __CUDA_ARCH__ >= 500
    // Device code path for compute capability 5.x
#elif __CUDA_ARCH__ >= 300
    // Device code path for compute capability 3.x
#elif !defined(__CUDA_ARCH__)
    // Host code path
#endif
}
```

4.1.1.4 Hành vi không xác định

'cross-execution space' call có hành vi không xác định khi:

‣ __CUDA_ARCH__ được defined, gọi __host__ function từ __global__, __device__ hoặc __host__ __device__ function.

‣ __CUDA_ARCH__ được undefined, __device__ function được gọi từ __host__ function .

4.1.1.5 __noinline__ and __forceinline__

Trình biên dịch nội tuyến bất kỳ hàm __device__ nào khi được cho là phù hợp

Bộ định tính hàm __inline__ có thể được sử dụng như một gợi ý để trình biên dịch không nội tuyến hàm nếu có thể.

4.1.2 Variable Memory Space Specifiers

Các biến chỉ định không gian bộ nhớ biểu thị vị trí bộ nhớ trên vùng nhớ của một biến. Một biến tự động được khai báo trong mã vùng nhớ mà không có bất kỳ ký hiệu __device__, __shared__ 5 và __constant__ không gian bộ nhớ nào được mô tả trong phần này thường nằm trong một thanh ghi. Tuy nhiên, trong một số trường hợp, trình biên dịch có thể chọn đặt nó vào bộ nhớ cục bộ, điều này có thể gây ra những hậu quả bất lợi về hiệu suất như được trình bày chi tiết trong Quyền truy cập bộ nhớ thiết bị.

4.1.2.1 __device__

chỉ định không gian bộ nhớ __device__ khai báo một biến nằm trên thiết bị.

Một biến tự động được khai báo trong mã thiết bị mà không có bất kỳ ký hiệu xác định không gian bộ nhớ __device__, __shared__ và __constant__ được mô tả trong phần này thường nằm trong aregister. Tuy nhiên, trong một số trường hợp, trình biên dịch có thể chọn đặt nó trong bộ nhớ cục bộ, điều này có thể gây ra những hậu quả bất lợi về hiệu suất như được trình bày chi tiết trong phần Quyền truy cập bộ nhớ thiết bị.

4.1.2.2 __shared__

Bộ chỉ định không gian bộ nhớ __shared__, được tùy chọn sử dụng cùng với __device__, khai báo

một biến:

- Nằm trong không gian bộ nhớ dùng chung của khối luồng,
- Có tuổi thọ của khối,
- Có một đối tượng riêng biệt trên mỗi khối,
- Chỉ có thể truy cập từ tất cả các chuỗi trong khối,
- Không có địa chỉ cố định.

Khi khai báo một biến trong bộ nhớ dùng chung dưới dạng một mảng bên ngoài, chẳng hạn như

```
extern __shared__ float shared[];
```

kích thước của mảng được xác định tại thời điểm khởi chạy (xem Cấu hình thực thi). Tất cả các biến được khai báo theo cách này, bắt đầu tại cùng một địa chỉ trong bộ nhớ, để bố cục của các biến trong mảng phải được quản lý rõ ràng thông qua các hiệu số.

```
short array0[128];
float array1[64];
int   array2[256];
```

trong bộ nhớ chia sẻ được cấp phát động, người ta có thể khai báo và khởi tạo các mảng theo cách sau:

```
extern __shared__ float array[];
__device__ void func()      // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[128];
    int*   array2 =   (int*)&array1[64];
}
```

Lưu ý rằng các con trỏ cần được căn chỉnh theo kiểu mà chúng trỏ tới, vì vậy, ví dụ: đoạn mã sau không hoạt động vì array1 không được căn chỉnh thành 4 byte.

```
extern __shared__ float array[];
__device__ void func()      // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[127];
}
```

4.1.2.3 __restrict__

nvcc hỗ trợ các con trỏ hạn chế thông qua từ khóa __restrict__. Các con trỏ hạn chế đã được giới thiệu trong C99 để giảm bớt vấn đề răng cưa tồn tại trong ngôn ngữ Ctype và ngăn chặn tất cả các loại tối ưu hóa từ sắp xếp lại mã cho đến loại bỏ biểu thức phụ thông thường.

Dưới đây là một ví dụ về vấn đề răng cưa, trong đó việc sử dụng con trỏ hạn chế có thể giúp trình biên dịch giảm số lượng lệnh:

```
void foo(const float* a,
         const float* b,
         float* c)
{
    c[0] = a[0] * b[0];
    c[1] = a[0] * b[0];
    c[2] = a[0] * b[0] * a[1];
    c[3] = a[0] * a[1];
    c[4] = a[0] * b[0];
    c[5] = b[0];
    ...
}
```

Trong các ngôn ngữ kiểu C, các con trỏ a, b và c có thể được đặt bí danh, vì vậy bất kỳ cách viết nào thông qua c đều có thể sửa đổi các phần tử của a hoặc b. Điều này có nghĩa là để đảm bảo tính đúng đắn của chức năng, trình biên dịch

không thể tải a [0] và b [0] vào các thanh ghi, nhân chúng và lưu trữ kết quả vào cả c [0] và c [1], vì kết quả sẽ khác với mô hình thực thi trừu tượng nếu, chẳng hạn, a [0] thực sự là cùng một vị trí với c [0]. Vì vậy trình biên dịch không thể tận dụng được biểu thức con chung. Tương tự như vậy, trình biên dịch không thể chỉ sắp xếp lại thứ tự tính toán của c [4] thành sự gần gũi của phép tính của c [0] và c [1] vì việc ghi trước đó vào c [3] có thể thay đổi đầu vào thành phép tính của c [4]. Bằng cách tạo các con trỏ bị giới hạn a, b và c, lập trình viên khẳng định với trình biên dịch rằng các con trỏ trên thực tế không phải là bí danh, trong trường hợp này có nghĩa là việc ghi qua c sẽ không bao giờ ghi đè lên các phần tử của a hoặc b. Điều này thay đổi nguyên mẫu hàm như sau:

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c);
```

Lưu ý rằng tất cả các đối số con trỏ cần được thực hiện hạn chế để trình tối ưu hóa trình biên dịch thu được bất kỳ lợi ích nào. Với các từ khóa __restrict__ được thêm vào, trình biên dịch hiện có thể sắp xếp lại thứ tự và thực hiện loại bỏ biểu thức con phổ biến theo ý muốn, trong khi vẫn giữ lại chức năng giống hệt vớimô hình thực thi trừu tượng:

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c)
{
    float t0 = a[0];
    float t1 = b[0];
    float t2 = t0 * t1;
    float t3 = a[1];
    c[0] = t2;
    c[1] = t2;
    c[4] = t2;
    c[2] = t2 * t3;
    c[3] = t0 * t3;
    c[5] = t1;
    ...
}
```

Các tác động ở đây là giảm số lần truy cập bộ nhớ và giảm số lần tính toán. Điều này được cân bằng bởi sự gia tăng áp suất thanh ghi do tải "được lưu trong bộ nhớ cache" và các biểu thức con chung.

Vì thanh ghi là một vấn đề quan trọng trong nhiều mã CUDA, việc sử dụng các con trỏ bị hạn chế có thể có tác động tiêu cực đến hiệu suất đối với mã CUDA, do giảm tỷ lệ sử dụng.

4.1.3 Built-in Vector Types

4.1.3.1 char, short, int, long, longlong, float, double

Đây là các kiểu vectơ có nguồn gốc từ các kiểu số nguyên và dấu phẩy động cơ bản. Chúng là các cấu trúc và các thành phần thứ nhất, thứ 2, thứ 3 và thứ 4 có thể truy cập được thông qua các trường x, y, z và w tương ứng. Tất cả chúng đều đi kèm với một hàm khởi tạo có dạng make_ <type name>; ví dụ,

```
int2 make_int2(int x, int y);
```

Các yêu cầu về căn chỉnh của các loại vectơ được trình bày chi tiết trong Bảng 4.

Type	Alignment
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16
long1, ulong1	4 if sizeof(long) is equal to sizeof(int) 8, otherwise
long2, ulong2	8 if sizeof(long) is equal to sizeof(int), 16, otherwise
long3, ulong3	4 if sizeof(long) is equal to sizeof(int), 8, otherwise
long4, ulong4	16
longlong1, ulonglong1	8

Type	Alignment
longlong2, ulonglong2	16
longlong3, ulonglong3	8
longlong4, ulonglong4	16
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16
double3	8
double4	16

4.1.4 Memory Fence Functions

Mô hình lập trình CUDA giả định một thiết bị có mô hình bộ nhớ có thứ tự yếu, đó là thứ tự trong đó luồng CUDA ghi dữ liệu vào bộ nhớ dùng chung, bộ nhớ chung, bộ nhớ máy chủ được phân trang hoặc bộ nhớ của thiết bị ngang hàng không nhất thiết phải theo thứ tự mà dữ liệu được quan sát được ghi bởi CUDA hoặc luồng máy chủ khác. Đó là hành vi không xác định đối với hai luồng đọc hoặc ghi vào cùng một vị trí bộ nhớ mà không đồng bộ hóa. Trong ví dụ sau, luồng 1 thực thi writeXY (), trong khi luồng 2 thực thi readXY ().

```
__device__ int X = 1, Y = 2;

__device__ void writeXY()
{
    X = 10;
    Y = 20;
}

__device__ void readXY()
{
    int B = Y;
    int A = X;
}
```

Hai luồng đọc và ghi từ cùng một vị trí bộ nhớ X và Y đồng thời. Bất kỳ cuộc đua dữ liệu nào đều là hành vi không xác định và không có ngữ nghĩa xác định. Các giá trị kết quả cho A và B có thể là bất kỳ giá trị nào. Các chức năng hàng rào bộ nhớ có thể được sử dụng để thực thi một số thứ tự trên các truy cập bộ nhớ. Các chức năng của hàng rào bộ nhớ khác nhau về phạm vi mà các lệnh được thực thi nhưng chúng độc lập với không gian bộ nhớ

được truy cập (bộ nhớ dùng chung, bộ nhớ chung, bộ nhớ máy chủ bị khóa trang và bộ nhớ của thiết bị ngang hàng).

```
void __threadfence_block();
```

đảm bảo rằng:

‣ Tất cả ghi vào tất cả bộ nhớ được thực hiện bởi chuỗi gọi trước khi thực hiện cuộc gọi tới __threadfence_block () được quan sát bởi tất cả các luồng trong khối của luồng đang gọi xảy ra trước khi tất cả ghi vào tất cả bộ nhớ được thực hiện bởi luồng đang gọi sau khi gọi tới __threadfence_block ();

‣ Tất cả các lần đọc từ tất cả bộ nhớ được thực hiện bởi chuỗi gọi trước khi lệnh gọi tới __threadfence_block () được sắp xếp trước khi tất cả các lần đọc từ tất cả bộ nhớ được thực hiện bởi chuỗi gọi sau cuộc gọi tới __threadfence_block ().

```
void __threadfence();
```

hoạt động như __threadfence_block () cho tất cả các luồng trong khối của luồng đang gọi và cũng đảm bảo rằng không ghi vào tất cả bộ nhớ được thực hiện bởi luồng đang gọi sau cuộc gọi tới __threadfence () được quan sát bởi bất kỳ luồng nào trong thiết bị như xảy ra trước khi ghi vào tất cả bộ nhớ được tạo bởi chuỗi đang gọi trước khi gọi tới __threadfence (). Lưu ý rằng để đảm bảo thứ tự này là đúng, các luồng quan sát phải thực sự quan sát bộ nhớ chứ không phải các phiên bản được lưu trong bộ nhớ cache của nó; điều này được đảm bảo bằng cách sử dụng từ khóa dễ bay hơi như được trình bày chi tiết trong Volatile Qualifier.

```
void __threadfence_system();
```

hoạt động như __threadfence_block () cho tất cả các luồng trong khối của luồng đang gọi và cũng đảm bảo rằng tất cả các lần ghi vào bộ nhớ được thực hiện bởi luồng đang gọi trước khi lệnh gọi tới __threadfence_system () được quan sát bởi tất cả các luồng trong thiết bị, luồng máy chủ và tất cả các luồng trong các thiết bị ngang hàng như xảy ra trước khi tất cả ghi vào tất cả bộ nhớ được thực hiện bởi luồng đang gọi sau khi gọi tới __threadfence_system ().

__threadfence_system () chỉ được hỗ trợ bởi các thiết bị có khả năng tính toán 2.x trở lên.

Trong ví dụ trước, chúng ta có thể chèn hàng rào trong các mã như sau:

```
__device__ int X = 1, Y = 2;

__device__ void writeXY()
{
    X = 10;
    __threadfence();
    Y = 20;
}

__device__ void readXY()
{
    int B = Y;
    __threadfence();
    int A = X;
}
```

Đối với mã này, các kết quả sau có thể được quan sát thấy:

- A bằng 1 và B bằng 2,
- A bằng 10 và B bằng 2,
- A bằng 10 và B bằng 20.

Kết quả thứ tư là không thể, bởi vì chữ viết frist phải được nhìn thấy trước khi viết thứ hai. Nếu luồng 1 và 2 thuộc cùng một khối, chỉ cần sử dụng __threadfence_block () là đủ. Nếu luồng 1 và 2 không thuộc cùng một khối, __threadfence () phải được sử dụng nếu chúng là luồng CUDA từ cùng một thiết bị và __threadfence_system () phải được sử dụng nếu chúng là luồng CUDA từ hai thiết bị khác nhau.

Một trường hợp sử dụng phổ biến là khi các luồng sử dụng một số dữ liệu do các luồng khác tạo ra như được minh họa bằng mẫu mã sau đây của một hạt nhân tính tổng của một mảng N số trong một lần gọi. Đầu tiên mỗi khối tính tổng một tập hợp con của mảng và lưu trữ kết quả trong bộ nhớ chung. Khi tất cả các khối được thực hiện xong, khối cuối cùng được thực hiện đọc từng tổng từng phần này từ bộ nhớ chung và tính tổng chúng để thu được kết quả cuối cùng. Để xác định khối nào được hoàn thành cuối cùng, mỗi khối tăng nguyên tử một bộ đếm để báo hiệu rằng khối đó được hoàn thành với tính toán và lưu trữ tổng từng phần của nó (xem Hàm nguyên tử về các hàm nguyên tử). Khối cuối cùng là khối nhận giá trị bộ đếm bằng gridDim.x-1. Nếu không có hàng rào nào được đặt giữa việc lưu trữ tổng từng phần và tăng dần bộ đếm, bộ đếm có thể tăng lên trước khi tổng từng phần được lưu trữ và do đó, có thể đạt tới gridDim.x-1 và để khối cuối cùng bắt đầu đọc tổng từng phần trước khi chúng thực sự được cập nhật trong trí nhớ. Các chức năng hàng rào bộ nhớ chỉ ảnh hưởng đến thứ tự của các hoạt động bộ nhớ theo một luồng; họ làm

không đảm bảo rằng các hoạt động bộ nhớ này được hiển thị cho các luồng khác (như __syncthreads () đối với các luồng trong một khối (xem Chức năng đồng bộ hóa)). Trong mẫu mã dưới đây, khả năng hiển thị của các hoạt động bộ nhớ trên biến kết quả được đảm bảo bằng cách khai báo nó là biến đổi (xem Volatile Qualifier).

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
                    volatile float* result)
{
    // Each block sums a subset of the input array.
    float partialSum = calculatePartialSum(array, N);

    if (threadIdx.x == 0) {

        // Thread 0 of each block stores the partial sum
        // to global memory. The compiler will use
        // a store operation that bypasses the L1 cache
        // since the "result" variable is declared as
        // volatile. This ensures that the threads of
        // the last block will read the correct partial
        // sums computed by all other blocks.
        result[blockIdx.x] = partialSum;

        // Thread 0 makes sure that the incrementation
        // of the "count" variable is only performed after
        // the partial sum has been written to global memory.
        __threadfence();

        // Thread 0 signals that it is done.
        unsigned int value = atomicInc(&count, gridDim.x);

        // Thread 0 determines if its block is the last
        // block to be done.
        isLastBlockDone = (value == (gridDim.x - 1));
    }

    // Synchronize to make sure that each thread reads
    // the correct value of isLastBlockDone.
    __syncthreads();

    if (isLastBlockDone) {

        // The last block sums the partial sums
        // stored in result[0 .. gridDim.x-1]
        float totalSum = calculateTotalSum(result);

        if (threadIdx.x == 0) {

            // Thread 0 of last block stores the total sum
            // to global memory and resets the count
            // varialble, so that the next kernel call
            // works properly.
            result[0] = totalSum;
            count = 0;
```

4.1.5 Synchronization Functions

```
void __syncthreads();
```

đợi cho đến khi tất cả các luồng trong khối luồng đạt đến điểm này và tất cả các truy cập toàn cục và bộ nhớ chia sẻ được thực hiện bởi các luồng này trước __syncthreads () sẽ hiển thị cho tất cả các luồng trong khối.

__syncthreads () được sử dụng để điều phối giao tiếp giữa các luồng của cùng một khối. Khi một số luồng trong một khối truy cập cùng địa chỉ trong bộ nhớ chung hoặc bộ nhớ chung, sẽ có những nguy cơ tiềm ẩn là đọc sau khi ghi, ghi sau khi đọc hoặc ghi sau khi ghi đối với một số truy cập bộ nhớ này. Có thể tránh được những nguy cơ dữ liệu này bằng cách đồng bộ hóa các chuỗi giữa các lần truy cập này.

_syncthreads () được cho phép trong mã có điều kiện nhưng chỉ khi điều kiện đánh giá giống nhau trên toàn bộ khối luồng, nếu không việc thực thi mã có thể bị treo hoặc tạo ra các tác dụng phụ không mong muốn.

Các thiết bị có khả năng tính toán 2.x trở lên hỗ trợ ba biến thể của __syncthreads () được mô tả bên dưới.

```
int __syncthreads_count(int predicate);
```

giống với __syncthreads () với tính năng bổ sung là nó đánh giá vị từ cho tất cả các luồng của khối và trả về số luồng mà vị từ đánh giá là khác.

```
int __syncthreads_and(int predicate);
```

giống với __syncthreads () với tính năng bổ sung là nó đánh giá vị từ cho tất cả các luồng của khối và trả về khác 0 nếu và chỉ khi vị từ đánh giá khác 0 cho tất cả chúng.

```
int __syncthreads_or(int predicate);
```

giống với __syncthreads () với tính năng bổ sung là nó đánh giá vị từ cho tất cả các luồng của khối và trả về khác 0 nếu và chỉ khi vị từ đánh giá khác 0 đối với bất kỳ luồng nào trong số chúng.

```
void __syncwarp(unsigned mask=0xffffffff);
```

sẽ khiến luồng đang thực thi đợi cho đến khi tất cả các đường dọc có tên trong mặt nạ đã thực thi một __syncwarp () (với cùng một mặt nạ) trước khi tiếp tục thực thi. Tất cả các luồng không thoát có tên trong mặt nạ phải thực thi một __syncwarp () tương ứng với cùng một mặt nạ, nếu không kết quả là không xác định.

Việc thực thi __syncwarp () đảm bảo thứ tự bộ nhớ giữa các luồng tham gia vào rào cản. Do đó, các luồng bên trong một sợi dọc muốn giao tiếp qua bộ nhớ có thể lưu vào bộ nhớ, thực thi __syncwarp (), và sau đó đọc các giá trị được lưu trữ bởi các luồng khác trong sợi dọc một cách an toàn.

4.2 Các tham số đánh giá hiệu năng của chương trình

Khi cố gắng tối ưu hóa mã CUDA, chúng ta phải biết cách đo lường hiệu suất một cách chính xác và hiểu được vai trò của băng thông trong việc đo lường hiệu suất. Trong phần này chúng ta sẽ thảo luận về cách đo hiệu suất một cách chính xác bằng cách sử dụng bộ định thời CPU và CUDA events. Nó thể hiện cách băng thông ảnh hưởng đến các chỉ số hiệu suất và cách giảm thiểu một số thách thức mà nó đặt ra.

4.2.1 Định thời gian

4.2.1.1 Sử dụng bộ định thời CPU

Bất kỳ bộ đếm thời gian CPU nào cũng có thể được sử dụng để đo thời gian đã trôi qua của một lời gọi CUDA hoặc một lời gọi kernel.

Khi sử dụng CPU timers, một điều quan trọng cần nhớ là các hàm CUDA API đều là không đồng bộ, do đó chúng sẽ trả lại quyền thực thi cho luồng chính trước khi các hàm CUDA API hoàn thành nhiệm vụ.

Tất cả các hàm khởi tạo kernel đều là các hàm không đồng bộ, cũng như các hàm truyền dữ liệu giữa các memory của device và host có đuôi *"Async"* ở cuối tên hàm. Do đó để đo được chính xác thời gian cụ thể của những lời gọi CUDA một cách riêng biệt, chúng ta cần đồng bộ hóa một CUDA call với CPU thread hiện tại bằng cách gọi hàm *"cudaDeviceSynchronize()"* trước khi bắt đầu và kết thúc các hàm đo định thời của CPUs. *"cudaDeviceSynchronize()"* ngăn chặn các lệnh điều khiển tiếp theo của CPU thread cho đến khi tất cả các lời gọi CUDA đã được gọi từ trước hoàn thành.

Đoạn code dưới đây minh họa về cách sử dụng bộ định thời CPU để đo tham số của một lời gọi kernel:

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

t1 = myCPUTimer();
saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
cudaDeviceSynchronize();
t2 = myCPUTimer();

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

4.2.1.2 Sử dụng bộ định thời GPU

CUDA event API cung cấp các lệnh gọi tạo và hủy events, ghi lại các event (bao gồm cả timestamp) và chuyển đổi sự khác biệt về dấu thời gian thành giá trị dấu phẩy động được tính bằng mili giây. Cách tính thời gian mã bằng các CUDA events được minh họa trong đoạn code dưới đây.

```
cudaEvent_t start, stop;

float time;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord( start, 0 );

kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y,NUM_REPS);

cudaEventRecord( stop, 0 );

cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );

cudaEventDestroy( start );

cudaEventDestroy( stop );
```

4.2.2 Băng thông

Băng thông - tốc độ truyền dữ liệu - là một trong những yếu tố quan trọng nhất đối với hiệu suất. Hầu hết tất cả các thay đổi đối với mã phải được thực hiện trong bối cảnh chúng ảnh hưởng đến băng thông như thế nào. Băng thông có thể bị ảnh hưởng đáng kể bởi việc lựa chọn bộ nhớ trong đó dữ liệu được lưu trữ, cách dữ liệu được bố trí và thứ tự truy cập dữ liệu, cũng như các yếu tố khác. Để đo hiệu suất một cách chính xác, rất hữu ích khi tính toán băng thông hiệu quả và lý thuyết. Khi cái sau thấp hơn nhiều so với cái trước, các chi tiết thiết kế hoặc triển khai có khả năng làm giảm băng thông và đó phải là mục tiêu chính của các nỗ lực tối ưu hóa tiếp theo để tăng băng thông. Một lưu ý ở đây là, mức độ ưu tiên cao: Sử dụng băng thông hiệu dụng trong chương trình tính toán của chúng ta làm thước đo khi đo lường hiệu suất và lợi ích tối ưu hóa.

4.2.2.1 Tính băng thông theo lý thuyết

Băng thông lý thuyết có thể được tính bằng cách sử dụng các thông số kỹ thuật phần cứng có sẵn trong tài liệu về sản phẩm. Ví dụ, GPU NVIDIA Tesla M2050 sử dụng RAM DDR (tốc độ dữ liệu gấp đôi) với memory clock rate là 1.546 MHz và bit wide memory interface 384-bit. Sử dụng các mục dữ liệu này, băng thông bộ nhớ lý thuyết cao nhất của NVIDIA Tesla M2050 là 148 GB/s, được tính như sau.

$$BW\ (Theoretical)\ =\ \frac{1546\ *\ 10^6\ *\ 384\ *\ 2}{8\ *\ 10^9}\ =\ 148\ GB/s$$

Trong phép tính này, chúng ta chuyển đổi tốc độ đồng hồ bộ nhớ thành Hz, nhân nó với chiều rộng giao diện (chia cho 8, để chuyển đổi bit thành byte) và nhân với 2 do tốc độ dữ liệu gấp đôi. Cuối cùng, chúng tôi chia cho 109 để chuyển đổi kết quả thành GB / s.

4.2.2.2 Tính băng thông hiệu dụng

Chúng ta tính toán băng thông hiệu quả bằng cách tính thời gian cho các hoạt động trong chương trình cụ thể và phân tích cách chương trình của chúng ta truy cập dữ liệu. Ở đây chúng ta sẽ sử dụng công thức sau đây:

$$BW\ (Effective)\ =\ \frac{R_B\ +\ W_B}{t\ *\ 10^9}$$

Ở đây, $BW_{Effective}$ là băng thông hiệu dụng tính bằng đơn vị GB/s, R_B là số byte được đọc trên mỗi kernel, W_B là số byte được ghi trên mỗi kernel và t là thời gian trôi qua tính

bằng giây. Chúng tôi có thể sửa đổi ví dụ SAXPY để tính toán băng thông hiệu dụng trong đoạn mã dưới đây:

```
__global__
void saxpy(int n, float a, float *x, float *y){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
int main(void){
  int N = 20 * (1 << 20);
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaEventRecord(start);
  // Perform SAXPY on 1M elements
  saxpy<<<(N+511)/512, 512>>>(N, 2.0f, d_x, d_y);
  cudaEventRecord(stop);
  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
  cudaEventSynchronize(stop);
  float milliseconds = 0;
  cudaEventElapsedTime(&milliseconds, start, stop);
  float maxError = 0.0f;
  for (int i = 0; i < N; i++) {
    maxError = max(maxError, abs(y[i]-4.0f));
  }
  printf("Max error: %fn", maxError);
  printf("Effective Bandwidth (GB/s): %fn", N*4*3/milliseconds/1e6);
}
```

4.2.3 Thông lượng tính toán

Trong phần trên, chúng ta đã trình bày cách đo băng thông hiệu dụng của một chương trình, là thước đo thông lượng dữ liệu. Một số liệu khác rất quan trọng đối với hiệu suất là *thông lượng tính toán*. Một thước đo phổ biến của thông lượng tính toán là *GFLOP/s*, viết tắt của *"Giga-FLoating-point OPerations per second"*. Đối với chương trình tính toán SAXPY trong ví dụ trên, việc đo lường thông lượng hiệu quả rất đơn giản: mỗi phần tử SAXPY thực hiện một phép nhân và cộng, thường được đo lường như hai FLOP, vì vậy chúng ta sẽ có:

$$\text{GFLOP/s (Effective)} = \frac{2N}{t * 10^9}$$

N là số toán tử trong chương trình *SAXPY* của chúng ta, *t* là thời gian trôi qua tính bằng giây. Giống như cách tính băng thông lý thuyết, GFLOP/s lý thuyết có thể được thu thập từ tài liệu sản phẩm (nhưng tính toán nó có thể hơi phức tạp vì nó rất phụ thuộc vào từng kiến trúc). Ví dụ: GPU Tesla M2050 có thông lượng dấu chấm động chính xác đơn cao nhất trên lý thuyết là 1030 GFLOP/s và thông lượng chính xác kép đỉnh lý thuyết là 515 GFLOP/s.

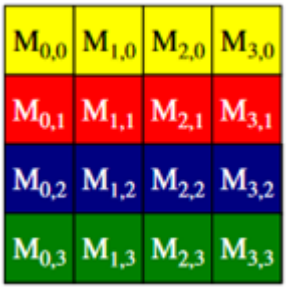
Trong các chương trình tính toán phức tạp hơn, việc đo lường hiệu suất ở cấp độ FLOP có thể rất khó khăn. Do đó, việc sử dụng các công cụ để biết liệu thông lượng tính toán có phải là một điểm nghẽn hay không. Các công cụ thường cung cấp các số liệu thông lượng theo vấn đề cụ thể (thay vì cụ thể về kiến trúc) và do đó hữu ích hơn cho người dùng.

CHƯƠNG 5. TRIỂN KHAI CÁC CHƯƠNG TRÌNH TÍNH TOÁN TRÊN GPU – NVIDIA

5.1 Chương trình nhân hai ma trận

5.1.1 Thuật toán nhân hai ma trận vuông sử dụng shared memory

Giả sử ta muốn nhân hai ma trận vuông M và N và kết quả là ma trận P. Trước tiên ta sẽ cần biểu diễn chúng trong máy dưới dạng mảng một chiều như ở hình vẽ. Có hai cách để thực hiện đó là ưu tiên theo hàng và ưu tiên theo cột.

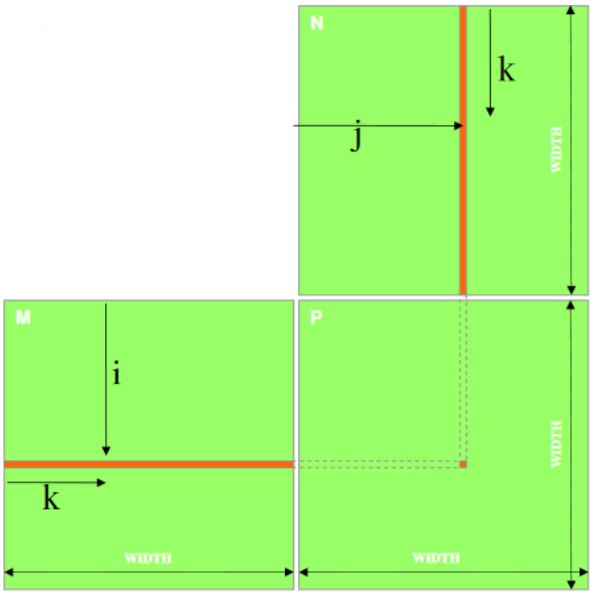


Hình 5-1 Ma trận M

Dưới đây là ưu tiên theo hàng.

Hình 5-2 Ma trận M biểu diễn bằng mảng một chiều

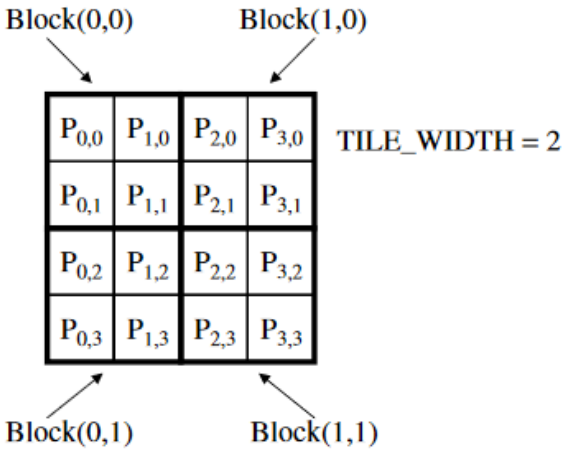
Sau khi đã biểu diễn được thì thuật toán nhân hai ma trận có thể được diễn giải như sau. Với mỗi hàng của ma trận M, với mỗi hàng của ma trận N, ta nhân các phần tử tương ứng với nhau và cộng tổng lại sẽ thu được kết quả là phần tử của ma trận P. Minh họa như hình bên dưới.



Hình 5-3 Minh họa nhân hai ma trận

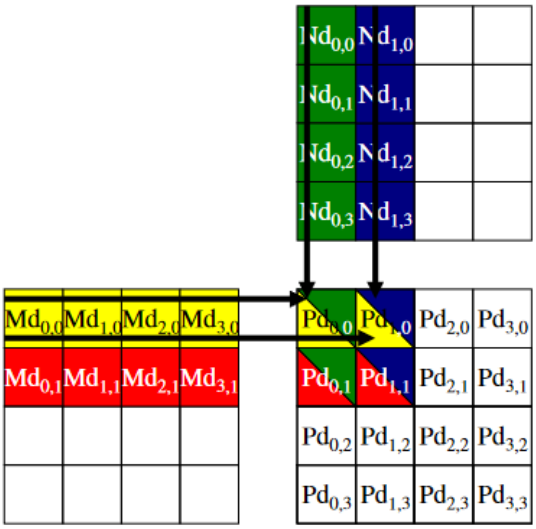
Bây giờ thay vì thực hiện các phép toán một cách tuần tự, ta sẽ gán cho mỗi phần tử tương ứng ở ma trận P là một thread. Ta sẽ khai báo một block có số thread tương đương với số lượng phần tử ở ma trận P mà chúng ta cần tính. Nhưng gặp phải vấn đề đó là nếu sử dụng cách này thì ta sẽ bị giới hạn kích thước ma trận do số thread có trong một khối bị giới hạn.

Để sử dụng nhiều khối thì ta sẽ sử dụng sharedmem. Cách thực hiện đó là ta sẽ chia nhỏ ma trận cần tính của chúng ta sẽ cũng thành các block như ở hình vẽ.



Hình 5-4 Chia ma trận thành các khối

Ta sẽ cần share mem để lưu ma trận con của A và ma trận con của B và sau đó việc tính toán diễn ra bình thường như ở nhân ma trận trong một block.



Hình 5-5 Nhân ma trận sử dụng shared mem

5.1.2 Thuật toán nhân hai ma trận vuông sử dụng thư viện cublas

Cublas là implementation của BLAS (Basic Linear Algebra Subprograms – Chương trình bên dưới thực hiện các phép đại số tuyến tính cơ bản).

BLAS là một bản mô tả tập hợp các cách thức để thực hiện các phép toán trong đại số tuyến tính như cộng vector, nhân với hằng số, phép nhân vô hướng, có hướng, nhân ma trận và tổ hợp tuyến tính. Các tính năng của BLAS được chia ra thành 3 tập lớn được gọi là các "level" theo thứ tự. Level 1 là những phép toán có độ phức tạp về thời gian là O(n). Tương tự Level 2 là những phép toán có độ phức tạp là n bình phương và level 3 là n lập phương. Bài toán của ta giải quyết là nhân ma trận có độ phức tạp tính toán là n lập phương vì thế nó thuộc level 3.

Về cơ bản phương thức nhân ma trận của CuBlas không khác gì so với cách ta đã nêu ở share mem. Việc tối ưu được thực hiện bằng các thủ thuật lập trình và đặc biệt là tận dụng tối đa bộ nhớ cache. Điều này khiến cho tốc độ của Cublas tăng lên đáng kể,

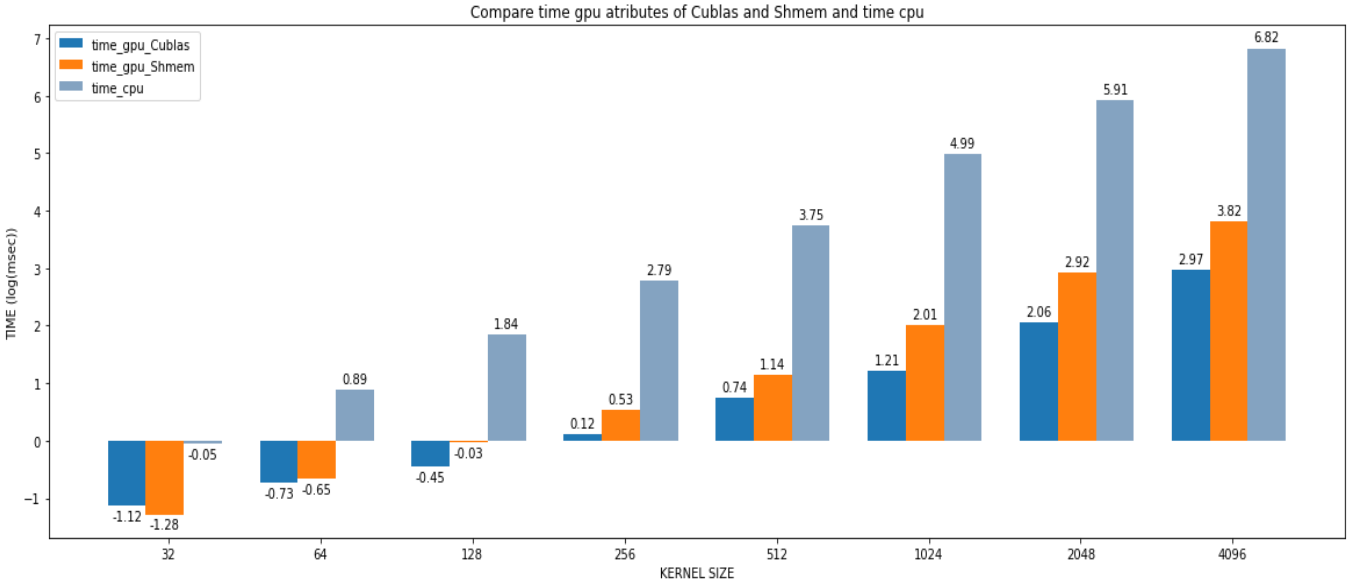
5.1.3 Đánh giá benchmark của các chương trình tính

Benchmark của chương trình được đánh giá trên GPU của Jetson Nano với những điều kiện như sau:

- Thẻ nhớ lưu trữ: Sandisk Ultra 128GB class 10 (tốc độ đọc 100MB/s, tốc độ ghi 10~15MB/s).

- Jetson Nano KIT: đang cài đặt ở chế độ 5W (yêu cầu về nguồn điện không đủ nên chỉ chạy kit ở chế độ 5W).
- Các thông số về ảnh đánh giá: ảnh định dạng *.JPG sử dụng OpenCV để đọc ảnh (có thể đọc được bất kỳ định dạng ảnh khác).

5.1.3.1 Thời gian thực thi



Hình 5-6 So sánh thời gian thực hiện giữa cublas, shmem và cpu

Vì giá trị thời gian ở mốc kernel size 32 và 4096 chênh lệch nhau rất lớn nên khi vẽ biểu đồ chúng em dùng thêm hàm $\log_{10}()$ cho toàn bộ giá trị để đưa về thang đo nhỏ hơn, tiện cho việc so sánh các thuộc tính giữa GPU và CPU, tương tự cho 2 biểu đồ bên dưới là Gflop/s và bandwidth effective.

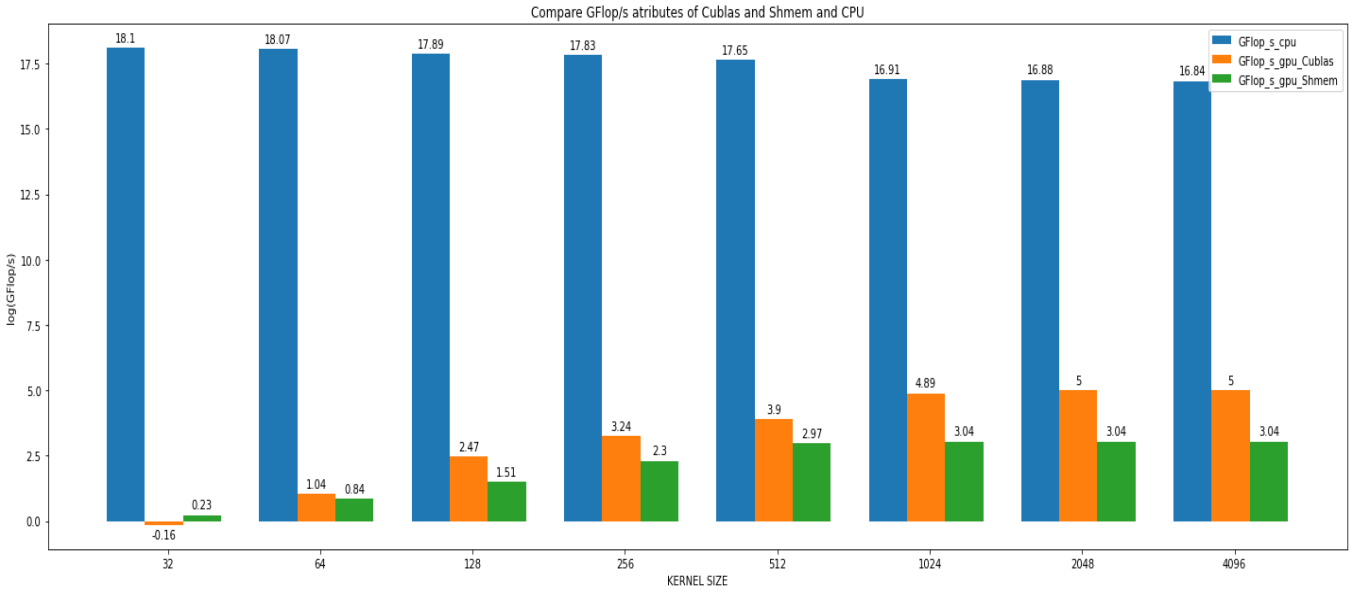
Hàm $\log_{10}()$ là hàm đồng biến, khi phân tích đồ thị ta có thể so sánh được giá trị hiện tại nào nhỏ hơn thì có nghĩa là thời gian thực thi nhỏ hơn, để so sánh đúng hơn về mặt định lượng ta chỉ cần lấy hai giá trị cần so sánh trừ đi cho nhau và lấy lũy thừa cơ số 10 của giá trị đó.

Dựa vào đồ thị trên Hình 5-6 chúng ta có một vài nhận xét như sau:

- Các chương trình nhân ma trận thực thi trên CPU có thời gian xử lý lâu hơn đáng kể so với các chương trình thực thi trên GPU (cụ thể, ở phép nhân hai ma trận vuông kích thước 4096 thời gian xử lý của chương trình tính toán trên GPU với CUBLAS nhanh hơn gấp 7079.45 lần so với thực thi trên CPU).

- Nhìn chung ở những phép nhân hai ma trận vuông với kích thước nhỏ (32 và 64) thì thời gian thực thi trên GPU của hai chương trình shmem (nhân ma trận sử dụng tối ưu với shared memory) và cublas (nhân ma trận sử dụng thư viện CUBLAS của NVIDIA) có thời gian thực thi gần bằng nhau (không đáng kể).
- Khi tăng kích thước ma trận lên càng lớn hơn (từ 128 đến 4096) chương trình sử dụng thư viện CUBLAS tỏ ra hiệu quả hơn so với chương trình nhân sử dụng tối ưu shared memory (cụ thể ở kích thước ma trận 4096 gấp khoảng hơn 7 lần thời gian thực thi).

5.1.3.2 GFLOPs của chương trình tính



Hình 5-7 So sánh GFLOP/s giữa cublas, shmem và cpu

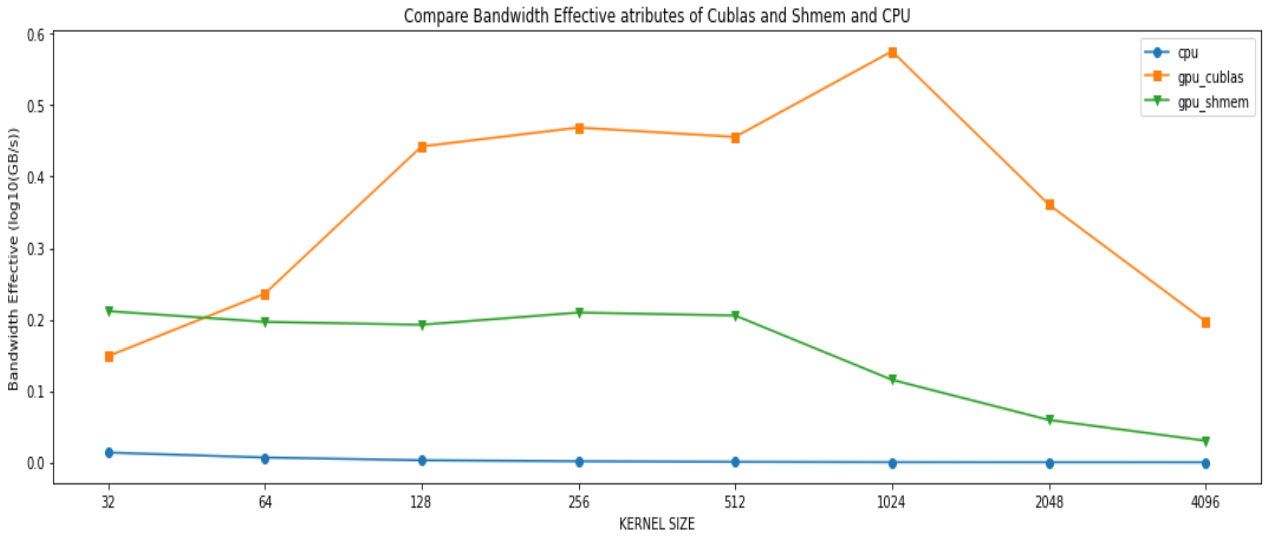
Công thức tính:

$$GFlop/s = \frac{Operators}{Time}$$

Trong đó : Ops : số phép toán thực hiện

time : thời gian thực hiện toàn bộ phép toán

5.1.3.3 Bandwidth của chương trình



Hình 5-8 So sánh bandwiwth effective giữa cublas, shmem và cpu

Công thức tính Bandwidth Effective:

$$BW\ (\text{Effective}) = \frac{3 * kernel\_size^2 * 4\ Byte}{t * 10^9}$$

Trong đó : GB không phải đơn vị lưu trữ nên không lấy 2^{30} mà là chia cho 10^9

T : thời gian thực hiện (s)

4 Byte : kích thước kiểu float

Tất cả giá trị đã được đưa qua hàm $\log_{10}()$ để về thang đo nhỏ hơn.

5.2 Triển khai bộ lọc trung vị cho ảnh

5.2.1 Thuật toán lọc trung vị không sử dụng shared memory

Chương trình được thiết kế với đầu vào là ảnh đã chèn padding zero (ứng với từng kích thước bộ lọc khác nhau thì kích thước chèn sẽ khác nhau), con trỏ chứa ảnh đầu ra (là ảnh gốc không chèn padding), độ phân giải của ảnh gốc ban đầu (chưa chèn padding).

Ý tưởng thiết kế thuật toán như sau:

- Ban đầu chúng ta tạo ra các thread block (hai chiều) chứa toàn bộ các thread, mỗi thread ứng với vị trí của từng điểm ảnh trong output.
- Kiểm tra điều kiện của mỗi thread, nếu chỉ số của thread hiện tại đang nằm ngoài ảnh output thì không thực hiện xử lý tiếp.

- Mỗi thread sẽ thực hiện: khởi tạo mảng một chiều có số lượng các phần tử là số lượng các phần tử trong một kernel; ứng với mỗi thread, ta sẽ trích xuất các phần tử trong kernel ứng với từng vị trí của ảnh input (đã thực hiện padding zero); sắp xếp các phần tử (bubble sort) theo chiều tăng dần và kiểm tra điều kiện lặp, trích xuất phần tử ở giữa mảng; thay thế giá trị đó vào vị trí thread hiện tại ứng với mỗi phần tử trong output.

Hình 5-9 Ảnh kết quả khi thực hiện lọc ảnh 810x1440 (không sử dụng shared mem) nhiễu muối tiêu với kernel size là 7

5.2.2 Thuật toán lọc trung vị sử dụng shared memory

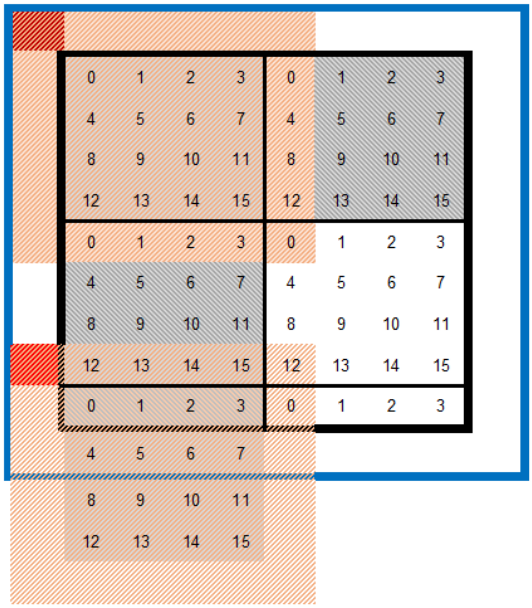
Chương trình được thiết kế hướng tới việc có khả năng lọc ảnh có kích thước bất kỳ, có thể thay đổi kích thước bộ lọc ảnh, thay đổi kích thước block thread, và thực hiện trên ảnh xám. Chương trình được thiết kế với đầu vào là ảnh đã chèn padding zero (ứng với từng kích thước bộ lọc khác nhau thì kích thước chèn sẽ khác nhau), con trỏ chứa ảnh đầu ra (là ảnh gốc không chèn padding), độ phân giải của ảnh gốc ban đầu (chưa chèn padding).

Ý tưởng thiết kế thuật toán được mô tả theo hình dưới đây (lấy ví dụ ảnh input có kích thước 11x10 đã được chèn padding – nằm trong khung viền xanh lam; ảnh output có kích thước 9x8 – nằm trong khung viền đen; kích thước kernel lọc là 3, thread block là 4x4; vùng shared memory block ứng với mỗi thread block có kích thước 6x6 – nằm trong vùng kẻ sọc màu đỏ, điểm đỏ đậm trên mỗi thread block được lấy giống như điểm tham chiếu tọa độ từ mỗi shared memory block so với vị trí tọa độ tại ảnh input) được chia thành hai phần chính như sau:

1. Thực hiện load data từ input vào mỗi shared memory block của mỗi thread block:

- Khởi tạo shared memory của mỗi block với kích thước phụ thuộc vào kích thước cửa sổ lọc và kích thước của mỗi thread block (như trong ví dụ này thì kích thước cửa sổ lọc là 3x3, kích thước của mỗi thread block là 4x4, do đó kích thước của shared memory là 6x6), khởi tạo điểm ban đầu ứng với mỗi thread block (sọc đỏ tía).
- Tính toán ***n*** vòng lặp xử lý công việc (nạp dữ liệu từ input vào shared memory ứng với từng phần tử) lặp lại của của các phần tử mang chỉ số tương ứng với chỉ số trong mỗi shared memory (các chỉ số ở đây được minh họa như trong hình là các chỉ số của các thread trong mỗi thread block, chúng ta cũng sẽ tính toán tương tự để suy ra chỉ số tương ứng trong shared memory).
- Mỗi thread trong thread block sẽ thực hiện ***n*** lần vòng lặp để nạp data từ vị trí hiện tại của input vào shared memory (để định được vị trí của input hiện tại trong ảnh, chúng ta tham chiều tọa độ của từng vị trí trong shared mem so với điểm tham chiếu ở đầu (điểm sọc đỏ tía)).

2. Thực hiện xử lý tính toán trên mỗi kernel size:
    - Kiểm tra điều kiện của mỗi thread, nếu chỉ số của thread hiện tại đang nằm ngoài ảnh output thì không thực hiện xử lý tiếp.
    - Mỗi thread sẽ thực hiện: khởi tạo mảng một chiều có số lượng các phần tử là số lượng các phần tử trong một kernel; ứng với mỗi thread, ta sẽ trích xuất các phần tử trong kernel ứng với từng vị trí trong shared memory (vừa thực hiện nạp dữ liệu ở phần trên); sắp xếp các phần tử (bubble sort) theo chiều tăng dần và kiểm tra điều kiện lặp, trích xuất phần tử ở giữa mảng; thay thế giá trị đó vào vị trí thread hiện tại ứng với mỗi phần tử trong output.



Hình 5-10 Minh họa về cách thiết kế bộ lọc Median Filter sử dụng shared memory

Kết quả của ảnh đầu ra được thể hiện như sau (giống với chương trình lọc ở trên), để kiểm tra chi tiết hơn về kết quả đầu ra của hai ảnh trong hai chương trình, ta có thể tính toán các chỉ số sau đây trong xử lý ảnh số: SSIM – đánh giá sự tương đồng giữa hai ảnh (trải dài từ -1 -> +1, giá trị bằng 1 thể hiện hai ảnh y hệt nhau), PSNR – đánh giá chất lượng lọc nhiễu sau khi lọc ảnh (tính bằng dB, giá trị càng lớn thì càng thể hiện chất lượng ảnh cải thiện nhiều).

Hình 5-11 Ảnh kết quả khi thực hiện lọc ảnh (sử dụng shared mem) nhiễu muối tiêu với kernel size là 7

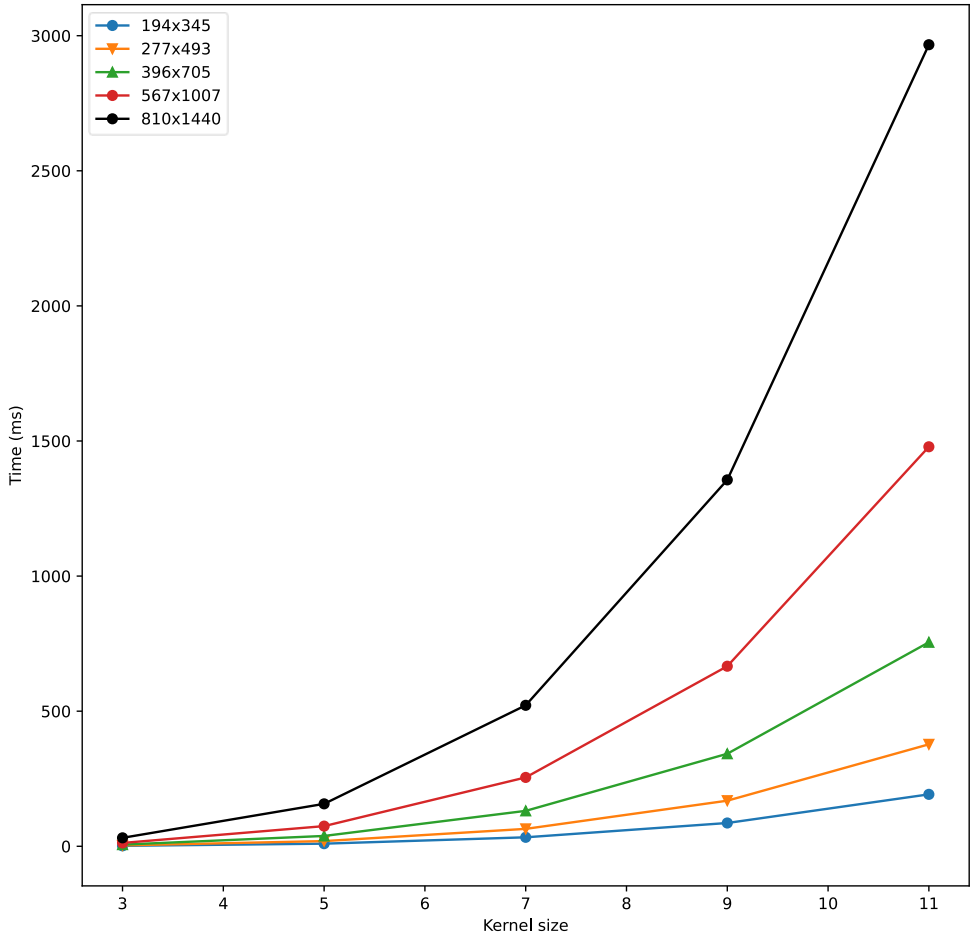
5.2.3 Đánh giá benchmark của các chương trình

Benchmark của chương trình được đánh giá trên GPU của Jetson Nano với những điều kiện như sau:

- Thẻ nhớ lưu trữ: Sandisk Ultra 128GB class 10 (tốc độ đọc 100MB/s, tốc độ ghi 10~15MB/s).
- Jetson Nano KIT: đang cài đặt ở chế độ 5W (yêu cầu về nguồn điện không đủ nên chỉ chạy kit ở chế độ 5W).
- Các thông số về ảnh đánh giá: ảnh định dạng *.JPG sử dụng OpenCV để đọc ảnh (có thể đọc được bất kỳ định dạng ảnh khác). Ảnh khảo sát có các độ phân giải ảnh là: 194x345, 277x493, 396x705, 567x1007, 810x1440; thay đổi với các kích thước kernel là: 3, 5, 7, 9, 11.

5.2.3.1 So sánh thời gian thực hiện lọc khi thay đổi kích thước của ảnh và bộ lọc

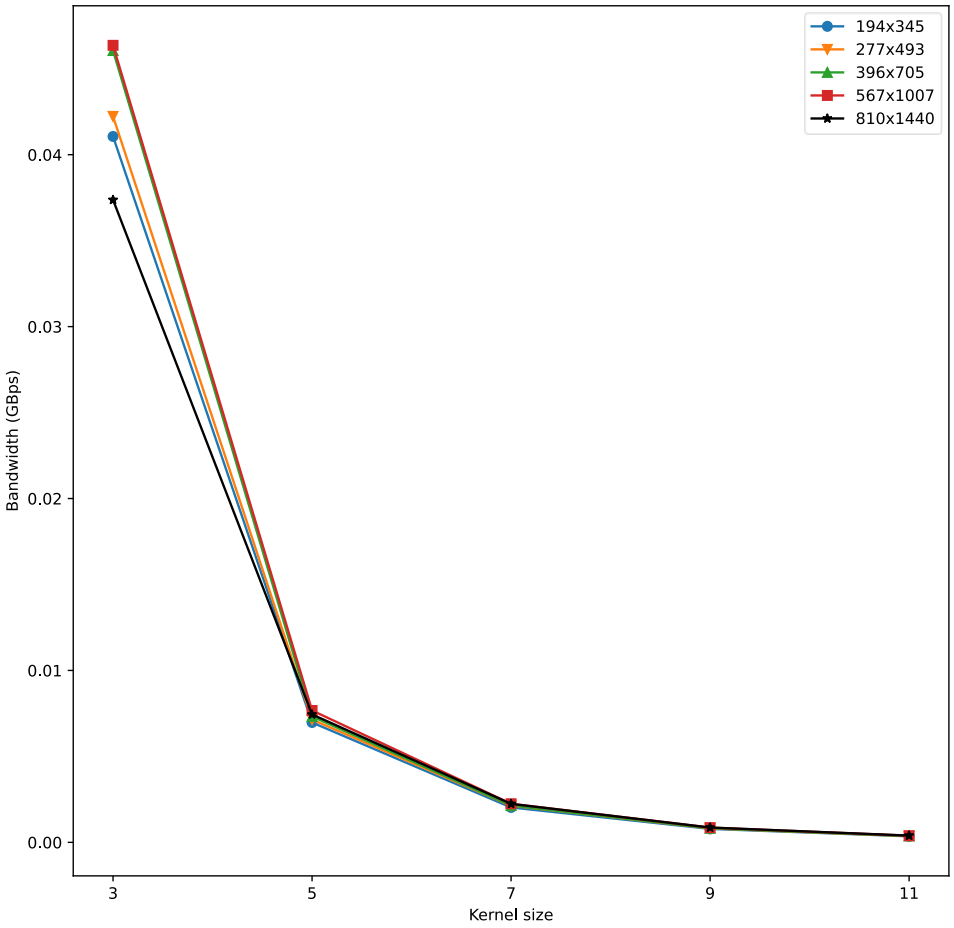
Hình vẽ dưới đây thể hiện sự thay đổi kích thước của ảnh và kích thước bộ lọc (độ phân giải ảnh vẽ theo các màu đường, kích thước kernel là trục hoành), ứng với mỗi một kích thước ảnh và kích thước bộ lọc ta có thời gian thực thi trên GPU tính bằng mili giây (trục tung).



Hình 5-12 So sánh thời gian thực hiện lọc khi thay đổi kích thước của ảnh và bộ lọc

5.2.3.2 So sánh băng thông thực thi lọc khi thay đổi kích thước của ảnh và bộ lọc

Băng thông của chương trình nhìn chung rất thấp so với những tính toán băng thông trên lý thuyết của Jetson Nano (trên spec của Jetson Nano thì băng thông tính được là 25.6 GB/s – xem ở phần phụ lục). Điều này xảy ra có thể do nhiều yếu tố, có thể do Jetson trong thời điểm đánh giá chỉ chạy mode 5W nên chưa phát huy được tối đa công suất tính toán, công thức tính băng thông chỉ đo trong phạm vi đọc ghi từ global memory (chưa tính các yếu tố đọc ghi khác như các vùng shared memory, local memory, register, …); do đó nếu cần có một định lượng chính xác về băng thông thì phải dùng tool của NVIDIA để đo.



Hình 5-13 So sánh băng thông thực hiện lọc khi thay đổi kích thước của ảnh và bộ lọc

CHƯƠNG 6. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

6.1 Tóm tắt kết quả

❖ Kết quả đạt được:

- Cả nhóm hoàn thành đúng tiến độ công việc và hoàn thiện được các đánh giá như yêu cầu.
- Các thành viên thảo luận sôi nổi, giúp đỡ lẫn nhau trong quá trình học tập.
- Hiểu biết hơn về lập trình song song, nâng cao tư duy lập trình kiến trúc hệ thống song song (cụ thể là kiến trúc CUDA của NVIDIA).

❖ Các vấn đề còn tồn đọng:

- Vẫn chưa hiểu được các cơ chế sâu, tổ chức hoạt động dưới phần cứng của CUDA.
- Chưa tối ưu hóa tốt phần triển khai bộ lọc trung vị sử dụng tối ưu shared memory (điển hình là thời gian chạy được đo trên GPU của chương trình tối ưu chỉ nhanh hơn một chút so với chương trình chính, chỉ cải thiện được vài nano giây).

6.2 Bài học kinh nghiệm

- Tư duy trong lập trình song song rất khác biệt so với tư duy lập trình thông thường (chạy trên CPU).
- Muốn tối ưu được chương trình, chúng ta cần phải hiểu rõ hơn về cách tổ chức hoạt động của GPU, phân tích tài nguyên, khái niệm về stream trong CUDA, … và phân tích được các pha quan trọng ảnh hưởng đến tốc độ xử lý.
- Trong code cần viết giải thích rõ ràng để người đọc có thể hiểu được, nhất là đối với chương trình tính toán trên GPU.

6.3 Phương hướng phát triển trong tương lai

- Phần triển khai bộ lọc trung vị có thể được áp dụng trong các quá trình tiền xử lý ảnh. Median filter là một bộ lọc thông thấp đặc biệt giúp chúng ta loại

bỏ được nhiễu muối tiêu hiệu quả; do thuật toán triển khai đơn giản, không phức tạp nên thời gian chạy nhanh, vẫn cho được kết quả mong muốn.

6.4 Kết luận chung

Sau quá trình thực hiện đề tài này nhóm chúng em đã học thêm được rất nhiều điều, nhóm đã có một cái nhìn tổng quan về lập trình song song, từ đó nhận biết được sự đa dạng, mạnh mẽ cũng như sự phát triển nhanh chóng của chúng.

Các loại kĩ năng mềm như lập kế hoạch, phân chia công việc, tìm tài liệu, thuyết trình… cũng được nhóm sử dụng rất nhiều trong quá trình thực thi nên khi kết thúc đề tài, những kĩ năng này của các thành viên trong nhóm tăng lên rất nhiều.

Chúng em rất cảm ơn thầy Nguyễn Đức Minh đã giúp chúng em hiểu hơn về kiến trúc máy tính trong phần lý thuyết ở trên lớp, để chúng em có cái nhìn rõ nét về những thành phần cơ bản trong kiến trúc máy tính .

Chúng em xin chân thành cảm ơn!

TÀI LIỆU THAM KHẢO

[1] Computer Organization and Design RISC-V Edition_v2

[2] cplusplus, "cplusplus," cplusplus, 18 06 2020. [Online]. Available: https://www.cplusplus.com/reference/fstream/fstream/?kw=fstream.

[3] NVIDIA, "https://developer.nvidia.com," 28 06 2021. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.

[4] NVIDIA, "https://docs.nvidia.com/," 14 07 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[5] NVIDIA, "https://docs.nvidia.com," 18 06 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations.

PHỤ LỤC

❖ Technical spectifications của Jetson Nano Developer Kit:

TECHNICAL SPECIFICATIONS	
GPU	NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores
CPU	Quad-core ARM Cortex-A57 MPCore processor
Memory	4 GB 64-bit LPDDR4, 1600MHz 25.6 GB/s
Storage	16 GB eMMC 5.1
Video Encode	250MP/sec 1x 4K @ 30 (HEVC) 2x 1080p @ 60 (HEVC) 4x 1080p @ 30 (HEVC) 4x 720p @ 60 (HEVC) 9x 720p @ 30 (HEVC)
Video Decode	500MP/sec 1x 4K @ 60 (HEVC) 2x 4K @ 30 (HEVC) 4x 1080p @ 60 (HEVC) 8x 1080p @ 30 (HEVC) 9x 720p @ 60 (HEVC)
Camera	12 lanes (3x4 or 4x2) MIPI CSI-2 D-PHY 1.1 (1.5 Gb/s per pair)
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI 2.0 and eDP 1.4
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I^2C, I^2S, SPI, UART
Mechanical	69.6 mm x 45 mm 260-pin edge connector