ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

NGUYỄN ĐỨC MINH

NGHIÊN CỨU GiẢI PHÁP CÔNG NGHỆ TÍNH TOÁN HIỆU NĂNG CAO VỚI BỘ XỬ LÝ ĐỒ HỌA GPU VÀ ỨNG DỤNG

LUẬN VĂN THẠC SĨ CÔNG NGHỆ THÔNG TIN

Hà Nội – 2016

ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

NGUYỄN ĐỨC MINH

NGHIÊN CỨU GiẢI PHÁP CÔNG NGHỆ TÍNH TOÁN HIỆU NĂNG CAO VỚI BỘ XỬ LÝ ĐỒ HỌA GPU VÀ ỨNG DỤNG

Ngành: Công nghệ thông tin

Chuyên ngành: Kỹ thuật phần mềm

Mã số: 60480103

LUẬN VĂN THẠC SĨ CÔNG NGHỆ THÔNG TIN

NGƯỜI HƯỚNG DẪN KHOA HỌC: TS. LÊ QUANG MINH

Hà Nội 2016

MỤC LỤC

LỜI CAM ĐOAN

Với mục đích học tập, nghiên cứu để nâng cao kiến thức và trình độ chuyên môn nên tôi đã làm luận văn này một cách nghiêm túc và hoàn toàn trung thực.

Trong luận văn, tôi có sử dụng tài liệu tham khảo của một số tác giả. Tôi đã nêu trong phần tài liệu tham khảo ở cuối luận văn.

Tôi xin cam đoan và chịu trách nhiệm về nội dung và sự trung thực trong luận văn tốt nghiệp Thạc sĩ của mình!

Hà Nội, tháng 05 năm 2016

Học viên

Nguyễn Đức Minh

LỜI CẢM ƠN

Những kiến thức căn bản trong luận văn này là kết quả của ba năm (2013-2016) tôi có may mắn được các thầy cô giáo trong Trường Đại học Công Nghệ - Đại học Quốc Gia Hà Nội, các thầy cô giáo ở các thầy cô giáo ở các Trường Đại học, Viện nghiên cứu trực tiếp giảng dạy, đào tạo và dìu dắt.

Tôi xin bày tỏ lời cảm ơn chân thành tới các thầy cô giáo trong Bộ môn Kỹ Thuật Phần Mềm – Khoa Công nghệ thông tin – Đại học Công Nghệ - ĐHQG Hà Nội, Phòng đào tạo sau đại học – Đại học Công Nghệ - ĐHQG Hà Nội đã tạo điều kiện thuận lợi cho tôi trong thời gian học tập tại trường.

Tôi xin bày tỏ lòng biết ơn chân thành, lời cảm ơn sâu sắc nhất đối với thầy giáo TS. Lê Quang Minh đã trực tiếp hướng dẫn, định hướng cho tôi giải quyết các vấn đề trong luận văn.

Tôi cũng xin cảm ơn các anh chị em đồng nghiệp ở Trường ĐH Công Nghệ Thông Tin & Truyền Thông và Trường ĐH Kỹ Thuật Công Nghiệp đã ủng hộ và giúp đỡ tôi trong quá trình thực hiện luận văn.

Luận văn cũng xin được là lời chia vui với người thân, đồng nghiệp, bạn bè và các bạn đồng môn lớp cao học K20.

Hà Nội, tháng 05 năm 2016

Học viên

Nguyễn Đức Minh

MỞ ĐẦU

Sự bùng nổ của Internet, sự bùng nổ của xu thế mọi thiết bị đều kết nối (Internet of thing - IOT), sự bùng nổ về nhu cầu thưởng các sản phẩm âm thanh đồ họa độ phân giải cao và chất lượng cao, sự bùng nổ của các dịch vụ lưu trữ đám mây, dịch vụ trực tuyến, đã khiến cho khôi lượng dữ liệu mà vi xử lý (CPU) phải tính toán ngày càng lớn và thực sự đã vượt quá nhanh so với sự phát triển tốc độ của CPU. Không những thế con người mặc dù muốn có nhiều thông tin hơn, thông tin phải tốt hơn lại còn muốn tốc độ xử lý phải nhanh hơn, điều này càng làm cho nhu cầu tính toán trong lĩnh vực khoa học, công nghệ đã và đang trở thành một thách thức lớn. Từ đó các giải pháp nhằm tăng tốc độ tính toán đã được ra đời.

Tư năm 2001 đến 2003 tốc độ của Pentium 4 đã tăng gấp đôi từ 1.5GHz lên đến 3GHz. Tuy nhiên hiệu năng của CPU không tăng tương xứng như mức gia tăng xung nhịp của CPU và việc tăng xung nhịp cũng chỉ đạt tới giới hạn công nghệ. Cụ thể tính đến 2005 xung nhịp của Pentium 4 mới chỉ tăng lên được 3.8GHz. Việc tăng xung nhịp của CPU dẫn đến việc tăng nhiệt độ làm việc của CPU. Các công nghệ làm mát có thể không đáp ứng được do bề mặt tiếp xúc của CPU ngày càng nhỏ. Trước tình hình này các nhà nghiên cứu vi xử lý đã chuyển sang hướng phát triển công nghệ đa lõi nhằm song song hóa các quá trình tính toán để tăng tốc độ và tiết kiệm điện năng. Một trong các công nghệ đa lõi xử lý song song ra đời là bộ xử lý đồ họa GPU (Graphic Processing Unit). GPU ban đầu ra đời chỉ phục vụ cho mục đích xử lý đồ họa, và ngành công nghiệp Game. Tuy nhiên ngày nay với công nghệ CUDA được phát triển bởi hãng NVIDIA từ năm 2007 đã cho phép thực hiện các tính toán song song với các phép tính phức tạp như dấu chấm động. Với hiệu xuất cả ngàn lệnh trong một thời điểm. Chính vì vậy một xu hướng nghiên cứu mới đã ra đời đó là phát triển các thuật toán song song thực hiện trên GPU. Với CUDA các lập trình viên có thể nhanh tróng phát triển các ứng dụng tính toán song song cho rất nhiều ứng dụng khác nhau như: Điện toán, sắp xếp, tìm kiếm, xử lý tín hiệu số, ảnh,…

Việc nghiêp cứu áp dụng CUDA để tăng tốc độ tính toán cho các bài toán

mà cần phải xử lý một khối dữ liệu đầu vào khổng lồ hoặc các bài toán yêu cầu tính thời gian thực đã thực sự trở thành một vấn đề cấp thiết trong thực tế. Xuất phát từ như cầu này mà tôi đã chọn đề tài **:** NGHIÊN CỨU GiẢI PHÁP CÔNG NGHỆ TÍNH TOÁN HIỆU NĂNG CAO VỚI BỘ XỬ LÝ ĐỒ HỌA GPU VÀ ỨNG DỤNG.

Luận văn gồm 3 chương chính:

Chương 1: Tổng quan về tính toán song song và GPU, chương này giới thiệu những kiến thức tổng quan về tính toán song song, từ đó tìm hiểu những kiến thức cơ bản về bộ xử lý đồ họa GPU và cách thức ứng dụng tính toán trên đó.

Chương 2: Tính toán song song trên GPU trong CUDA,. Chương này cung cấp các kiến thức về môi trường lập trình, ngôn ngữ lập trình, cách thiết lập chương trình và các chỉ dẫn hiệu năng khi cài đặt ứng dụng tính toán trên GPU.

Chương 3:Tăng tốc độ tính toán một số bài toán sử dụng GPU. Trên cơ cở các kiến thức được trình bày ở các chương trên, tác giả luận văn đã tiến hành cài đặt và thử nghiệm mô phỏng bài toán trên CPU và GPU. Từ đó có những so sánh, nhận xét về năng lực tính toán vượt trội của GPU so với CPU truyền thống. Đồng thời cũng mở ra các hướng cải tiến hiệu năng mới cho bài toán chạy trên GPU.

DANH MỤC THUẬT NGỮ

	Tiếng Anh	**Tiếng Việt**
	API	Application Program Interface: một API định nghĩa một giao diện chuẩn để triệu gọi một tập các chức năng.
	coproccessor	bộ đồng xử lý
	gpgpu	tính toán thông dụng trên GPU
	GPU	Bộ xử lý đồ họa
	kernel	hạt nhân
	MIMD	Multiple Instruction Multiple Data: đa lệnh đa dữ liệu
	primary surface	Bề mặt chính, khái niệm dùng trong kết cấu
	proccessor	Bộ xử lý
	Rasterization	Sự quét mành trên màn hình
	SIMD	Single Instruction Multiple Data: đơn lệnh đa dữ liệu
	stream	Dòng
	streaming processor	Bộ xử lý dòng
	texture	Kết cấu: cấu trúc của đối tượng, nó được xem như mô hình thu nhỏ của đối tượng.
	texture fetches	Hàm đọc kết cấu
	texture reference	Tham chiếu kết cấu
	warp	Mỗi khối được tách thành các nhóm SIMD của các luồng.

DANH MỤC HÌNH VẼ

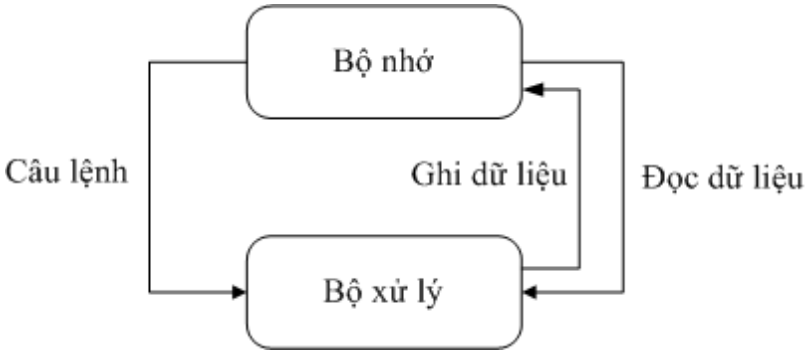
CHƯƠNG 1: TỔNG QUAN VỀ TÍNH TOÁN SONG SONG VÀ GPU

1.1. Tổng quan về tính toán song song

1.1.1. Tổng quan về tính toán song song

1.1.1.1. Lịch sử ra đời tính toán songsong

Trong những thập niên 60, nền tảng để thiết kế máy tính đều dựa trên mô hình của John Von Neumann , với một đơn vị xử lý được nối với một vùng lưu trữ làm bộ nhớ và tại một thời điểm chỉ có một lệnh được thực thi.



Hình 1. Mô tả kiến trúc Von Neumann

Với những bài toán yêu cầu về khả năng tính toán và lưu trữ lớn thì mô hình kiến trúc này còn hạn chế. Để tăng cường sức mạnh tính toán giải quyết các bài toán lớn có độ tính toán cao, người ta đưa ra kiến trúc mới, với ý tưởng kết hợp nhiều bộ xử lý vào trong một máy tính, mà hay gọi là xử lý song song (Multiprocessor) hoặc kết hợp sức mạnh tính toán của nhiều máy tính dựa trên kết nối mạng (máy tính song song-multicomputer).

Kể từ lúc này, để khai thác được sức mạnh tiềm tàng trong mô hình máy tính nhiều bộ xử lý song song, cũng như trong mô hình mạng máy tính xử lý song song thì các giải thuật tuần tự không còn phù hợp nữa cho nên việc xây dựng thiết kế giải thuật song song là điều quan trọng. Giải thuật song song có thể phân rã công việc trên các phần tử xử lý khác nhau.

1.1.1.2. Tại sao phải tính toán song song

Theo xu hướng phát triển của công nghệ thông tin, các bộ xử lý đa nhân, đa lõi (multiple processor) đang dần dần thay thế các bộ xử lý đơn lõi (single processor) tuy nhiên với lối lập trình truyền thống (lập trình tuần tự), các câu lệnh, các quá trình xử lý được thực hiện một cách lần lượt, tuần tự như vậy sẽ không phát huy hết công năng, hiệu năng của bộ vi xử lý đa nhân, đa lõi (multiple processor). Lập trình, tính toán song song ra đời như một lời giải cho yêu cầu, thách thức đặt ra là làm thế nào để phát huy công năng, hiệu năng của bộ đa xử lý (multiple processor).

Trên thực tế, có rất nhiều bài toán với dữ liệu lớn, độ phức tạp tính toán cao mà đòi hỏi thời gian xử lý ngắn, độ chính xác cao. Ví dụ như các bài toán liên quan tới xử lý ảnh, xử lý tín hiệu, dự báo thời tiết, mô phỏng giao thông, mô phỏng sự chuyển động của các phân tử, nguyên tử, mô phỏng bản đồ gen, các bài toán liên quan đến cơ sở dữ liệu và khai thác cơ sở dữ liệu,… với bộ xử lý đơn lõi thì khó có thể thực hiện và cho kết quả như mong muốn được.

Lập trình, tính toán song song là lời giải đáp cho bài toán tăng hiệu năng xử lý đồng thời rút ngắn thời gian xử lý tính toán của người dùng.

1.1.1.3. Một số khái niệm xử lý song song

- **Định nghĩa xử lý song song**

  Xử lý song song là quá trình xử lý gồm nhiều tiến trình được kích hoạt đồng thời và cùng tham gia giải quyết một bài toán. Nói chung, xử lý song song được thực hiện trên những hệ thống đa bộ xử lý.

- **Phân biệt xử lý song song và xử lý tuần tự**

  Trong tính toán tuần tự với một bộ xử lý thì tại mỗi thời điểm chỉ được thực hiện một phép toán. Trong tính toán song song thì nhiều bộ xử lý cùng kết hợp với nhau để giải quyết cùng một bài toán cho nên giảm được thời gian xử lý vì mỗi thời điểm có thể thực hiện đồng thời nhiều phép toán. Dưới đây là bảng so sánh sự khác nhau giữa lập trình tuần tự và lập trình song song.

Bảng 1.1: So sánh sự khác nhau giữa lập trình tuần tự và song song

Lập trình tính toán tuần tự	**Lập trình tính toán song song**
- Chương trình ứng dụng chạy trên bộ xử lý đơn (single processor).	- Chương trình ứng dụng chạy trên hai hoặc nhiều bộ xử lý.
- Các chỉ thị lệnh được bộ xử lý (CPU) thực hiện một cách lần lượt, tuần tự.	- Các chỉ thị lệnh được các bộ vi xử lý thực hiện một cách song song, đồng thời.
- Mỗi chỉ thị lệnh chỉ thực thiện trên duy nhất một thành phần dữ liệu.	- Mỗi chỉ thị lệnh có thể thao tác trên hai hoặc nhiều thành phần dữ liệu khác nhau.
- Lập trình viên chỉ cần đảm bảo viết đúng mã lệnh theo giải thuật chương trình là chương trình có thể dịch, chạy và cho ra kết quả.	- Ngoài việc đảm bảo viết đúng mã lệnh theo giải thuật, lập trình viên còn phải chỉ ra trong chương trình đoạn mã nào được thực hiện song song, đồng thời.
- Thường được áp dụng đối với các bài toán có dữ liệu nhỏ, độ phức tạp bình thường và thời gian cho phép.	- Thường được áp dụng đối với các bài toán có dữ liệu lớn, độ phức tạp cao và thời gian ngắn.

- **Mục đích của xử lý song song**

Thực hiện tính toán nhanh trên cơ sở sử dụng nhiều bộ xử lý đồng thời. Cùng với tốc độ xử lý nhanh, việc xử lý song song cũng sẽ giải được những bài toán phức tạp yêu cầu khối lượng tính toán lớn.

1.1.2. Mô hình máy tính song song

Một hệ thống máy tính song song là một máy tính với nhiều hơn một bộ xử lý cho phép xử lý song song. Định nghĩa này có thể bao quát được tất cả các siêu máy tính với hàng trăm bộ xử lý, các mạng máy tính trạm,… Thậm chí trong mấy năm gần đây các máy tính có vi xử lý áp dụng công nghệ mới multicore cho phép nhiều nhân trong một

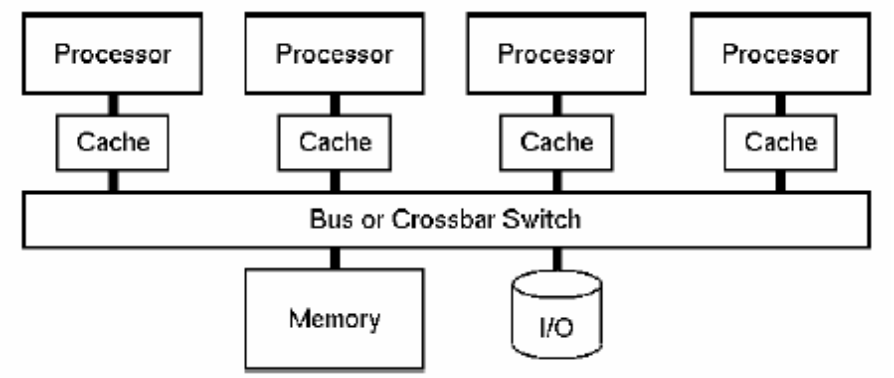
bộ xử lý cũng được xem là hệ thống máy tính song song.

Một trong những phân loại kiến trúc máy tính song song được biết đến nhiều nhất là phân loại của Flynn, được sử dụng từ năm 1966. Michael Flynn dựa vào đặc tính về số lượng bộ xử lý, số chương trình thực hiện, cấu trúc bộ nhớ,… để phân máy tính thành bốn loại dựa trên sự biểu hiện của cặp khái niệm: Dòng lệnh (instruction stream) và dòng dữ liệu (data stream), mỗi loại nằm trong một trong hai trạng thái đơn (single) hoặc đa (multiple). Một dòng dữ liệu là một dãy các dữ liệu được sử sụng để điều khiển các dòng lệnh và dữ liệu có thể được phân ra làm 4 loại như sau :

Bảng 1.2: Mô tả phân loại kiến trúc của Flynn

Dòng lệnh (instruction stream)	**Dòng dữ liệu (data stream)**	**Loại kiến trúc**
Trạng thái đơn (single)	Trạng thái đơn (single)	SISD Single Instruction Single Data
Trạng thái đơn (single)	Trạng thái đa (multiple)	SIMD Single Instruction Multiple
Trạng thái đa (multiple)	Trạng thái đơn (single)	MISD Multiple Instruction Single
Trạng thái đa (multiple)	Trạng thái đa (multiple)	MIMD Multiple Instruction Multiple

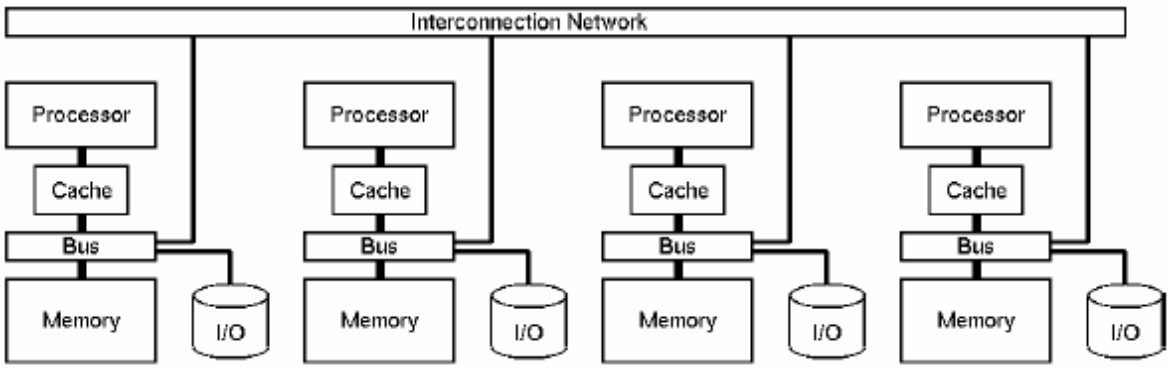
Sự phân chia này được dựa trên kiến trúc bộ nhớ của các máy tính song song. Các máy tính song song có bộ nhớ chia sẻ (shared memory) có nhiều bộ xử lý cùng được truy nhập đến một vùng nhớ tổng thể dùng chung. Tất cả các sự thay đổi nội dung bộ nhớ do một bộ xử lý tạo ra sẽ được nhận biết bởi các bộ xử lý khác.



Hình 2 : Máy tính song song có bộ nhớ chia sẻ

Trong lớp máy tính này có thể phân chia làm 2 lớp nhỏ hơn: Lớp UMA (Uniform Memory Access – Truy cập bộ nhớ đồng nhất) cho phép thời gian truy cập bộ nhớ đối với mỗi bộ xử lý là như nhau. Lớp NUMA (Non-Uniform Memory Access – Truy cập bộ nhớ không đồng nhất) có thời gian truy cập bộ nhớ không phải lúc nào cũng như nhau [3].

Còn lại, các máy tính song song có bộ nhớ phân tán cũng có nhiều bộ xử lý nhưng với mỗi bộ xử lý chỉ có thể truy cập đến bộ nhớ cục bộ của nó, không có một vùng nhớ dùng chung nào cho tất cả các bộ xử lý. Các bộ xử lý hoạt động độc lập với nhau và sự thay đổi trong vùng nhớ cục bộ không làm ảnh hưởng đến vùng nhớ của các bộ xử lý khác.

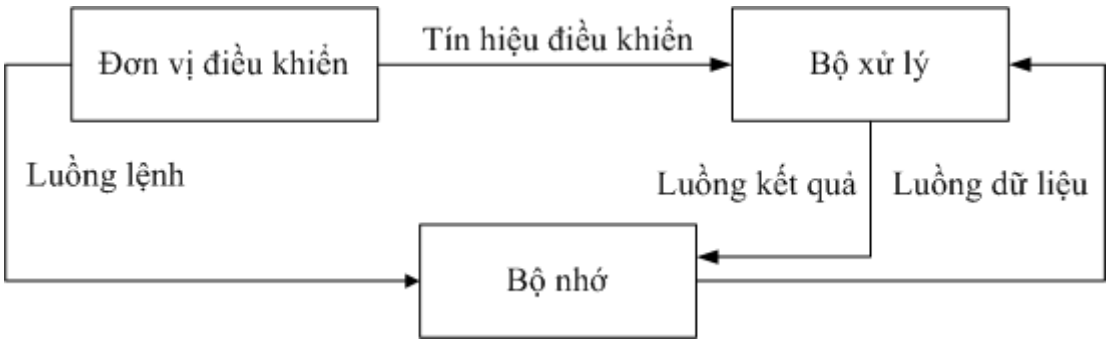


Hình 3 : Máy tính song song có bộ nhớ phân tán

1.1.2.1. Kiến trúc đơn dòng lệnh đơn luồng dữ liệu (SISD)

Máy tính SISD chỉ có một CPU, ở mỗi thời điểm thực hiện một chỉ lệnh và chỉ đọc, ghi một mục dữ liệu. Tất cả các máy tính SISD chỉ có một thanh ghi (register)

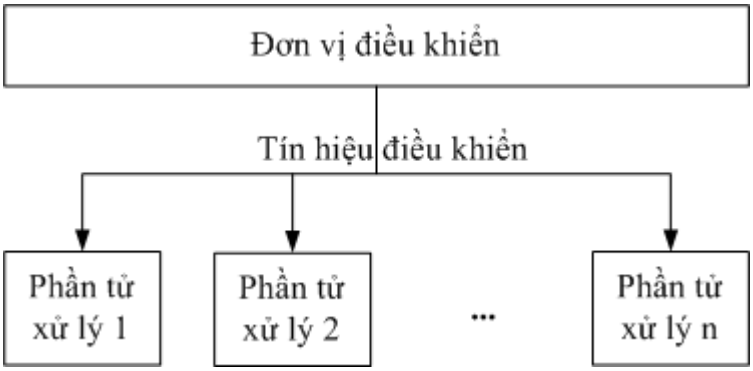
được gọi là bộ đệm chương trình, được sử dụng để nạp địa chỉ của lệnh tiếp theo và kết quả là thực hiện theo một thứ tự xác định của các câu lệnh.



Hình 4 : Mô hình kiến trúc máy SISD

1.1.2.2. Kiến trúc đơn dòng lệnh đa luồng dữ liệu (SIMD)

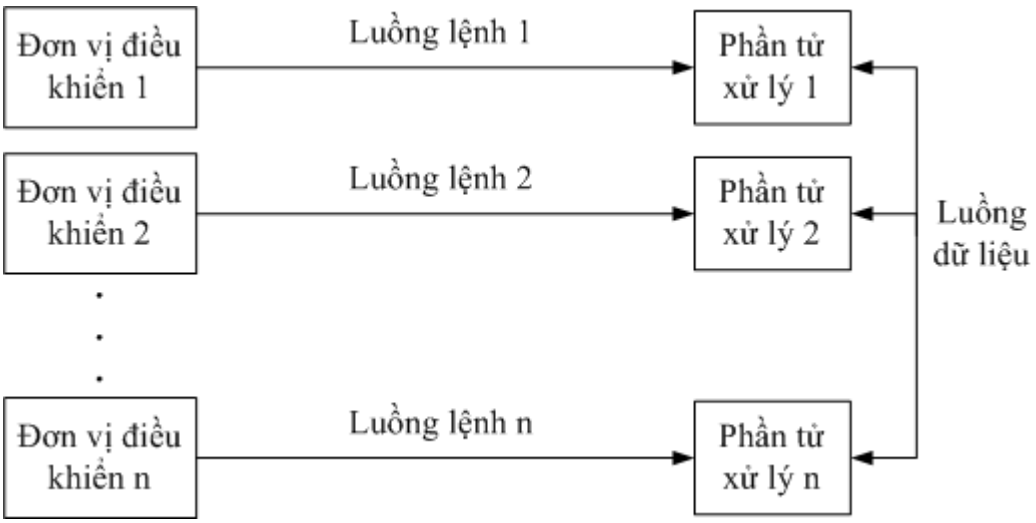
Máy tính SIMD có một đơn vị điều khiển để điều khiển nhiều đơn vị xử lý thực hiện theo một luồng các câu lệnh. CPU phát sinh tín hiệu điều khiển tới tất cả các phần xử lý, những bộ xử lý này cùng thực hiện một phép toán trên các mục dữ liệu khác nhau, nghĩa là mỗi bộ xử lý có luồng dữ liệu riêng. Mô hình SIMD còn được gọi là SPMD, đơn chương trình và đa dữ liệu.



Hình 5 : Mô hình kiến trúc máy SIMD

1.1.2.3. Kiến trúc đa dòng lệnh đơn luồng dữ liệu (MISD)

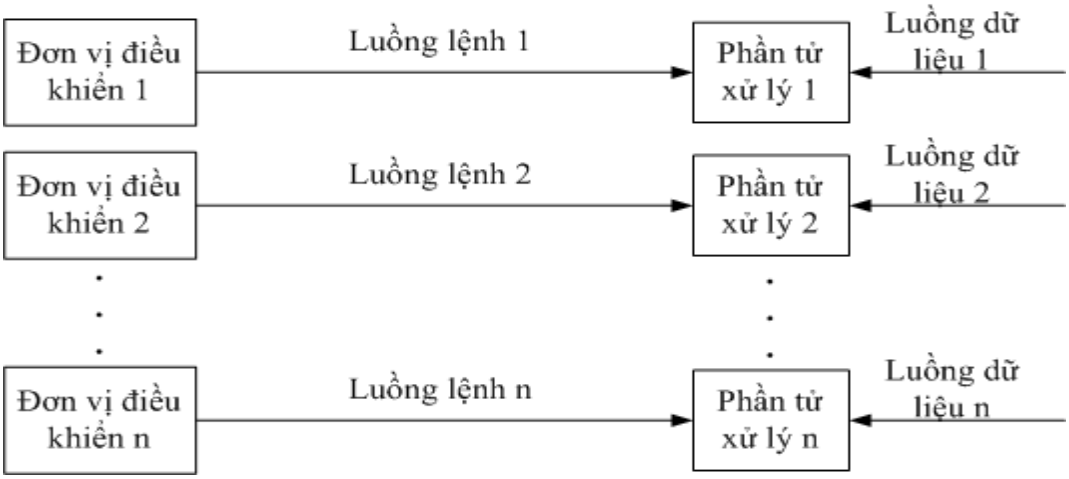
Máy tính loại MISD có thể thực hiện nhiều chương trình (nhiều lệnh) trên cùng một mục dữ liệu (ngược với máy tính loại SIMD).



Hình 6 : Mô hình kiến trúc máy MISD

1.1.2.4. Kiến trúc đa dòng lệnh đa luồng dữ liệu (MIMD)

Máy tính loại MIMD gọi là đa bộ xử lý, trong đó mỗi bộ xử lý có thể thực hiện những luồng lệnh (chương trình) khác nhau trên các luồng dữ liệu riêng. Hầu hết các hệ thống MIMD đều có bộ nhớ riêng và cũng có thể truy cập vào bộ nhớ chung khi cần, do vậy giảm thiểu được thời gian trao đổi dữ liệu giữa các bộ xử lý trong hệ thống. Đây là loại kiến trúc phức tạp nhất, nhưng nó là mô hình hỗ trợ xử lý song song cao nhất và đã có nhiều máy tính được thiết kế theo kiến trúc này, ví dụ: BBN Butterfly, Alliant FX, iSPC của Intel, ... Kiến trúc máy MIMD có mô hình hoạt động theo Hình 1.7 .



Hình 7 : Mô hình kiến trúc máy MIMD

1.1.3. Mô hình lập trình song song

Công việc lập trình song song bao gồm việc thiết kế, lập trình các chương trình máy tính song song sao cho chạy được trên các hệ thống máy tính song song. Hay có nghĩa là song song hoá các chương trình tuần tự nhằm giải quyết một vấn đề lớn hoặc làm giảm thời gian thực thi hoặc cả hai.

Lập trình song song tập trung vào việc phân chia bài toán tổng thể ra thành các công việc con nhỏ hơn rồi phân chia các công việc đó đến từng bộ xử lý (processor) và đồng bộ các công việc để nhận được kết quả cuối cùng. Nguyên tắc quan trọng nhất ở đây chính là tính đồng thời hoặc xử lý nhiều tác vụ cùng một lúc. Do đó, trước khi lập trình cần xác định được rằng bài toán có thể được song song hoá hay không (có thể dựa trên dữ liệu hay chức năng của bài toán). Có hai hướng chính trong việc tiếp cận lập trình song song [3]:

- **Song song hóa ngầm định**: Bộ biên dịch hay một vài chương trình khác tự động phân chia các công việc đến các bộ xử lý.
- **Song song hóa bằng tay**: Người lập trình phải tự phân chia chương trình để nó có thể thực hiện song song.

Ngoài ra trong lập trình song song, người lập trình viên cần phải tính đến yếu tố cân bằng tải (load balancing) trong hệ thống. Phải làm cho các bộ xử lý thực hiện số công việc như nhau, nếu có một bộ xử lý có tải quá lớn thì cần phải di chuyển công việc đến bộ xử lý có tải nhỏ hơn.

Việc truyền thông giữa các bộ xử lý là một công việc không thể thiếu của lập trình song song. Có hai kỹ thuật truyền thông chủ yếu là: dùng bộ nhớ chia sẻ (shared memory) hoặc truyền thông điệp (message passing).

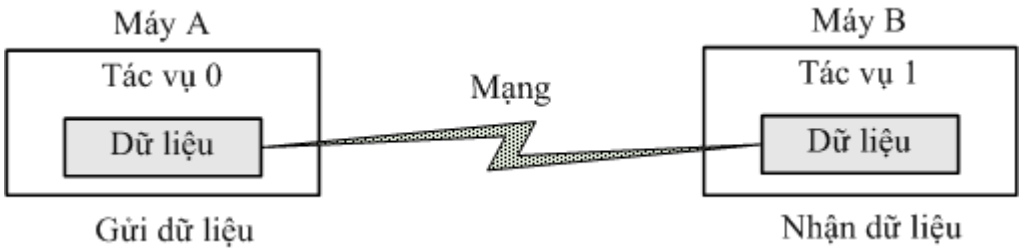
Mô hình lập trình song song bao gồm các ứng dụng, ngôn ngữ, bộ biên dịch, thư viện, hệ thống truyền thông và vào/ra song song. Trong thực tế, chưa có một máy tính song song nào cũng như cách phân chia công việc cho các bộ xử lý nào có thể áp dụng có hiệu quả cho mọi bài toán. Do đó, người lập trình phải lựa chọn chính xác mô hình lập trình song song hoặc pha trộn các mô hình đó để phát triển các ứng dụng song song trên một hệ thống riêng biệt.

Hiện nay có rất nhiều mô hình lập trình song song: Truyền thông điệp (Message Passing), Song song dữ liệu (Data Parallel).

1.1.3.1. Mô hình truyền thông điệp

Truyền thông điệp (Message Passing) là mô hình được sử dụng rộng rãi trong tính toán song song hiện nay, thường áp dụng cho các hệ thống phân tán. Các đặc trưng của mô hình là:

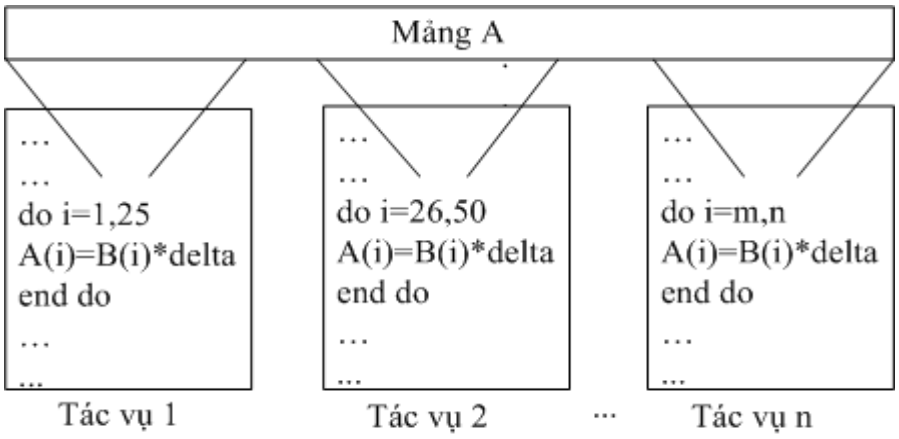
- Một tập các luồng sử dụng vùng nhớ cục bộ riêng của chúng trong suốt quá trình tính toán.
- Nhiều luồng có thể sử dụng một tài nguyên vật lý.
- Các luồng trao đổi dữ liệu bằng cách gửi nhận các thông điệp.
- Việc truyền dữ liệu thường yêu cầu thao tác điều phối thực hiện bởi mỗi luồng. Ví dụ, một thao tác gửi ở một luồng thì phải ứng với một thao tác nhận ở luồng khác.



Hình 8: Mô hình lập trình truyên thông giữa hai tác vụ trên hai máy tính

1.1.3.2. Mô hình song song dữ liệu

Trong mô hình song song dữ liệu được mô tả ở Hình 1.9, hầu hết các công việc song song đều tập trung thực hiện các phép toán trên một tập dữ liệu. Tập dữ liệu này thường được tổ chức trong một cấu trúc dữ liệu thông dụng như mảng hoặc khối. Một tập tác vụ sẽ làm việc trên cùng cấu trúc dữ liệu nhưng mỗi tác vụ sẽ làm việc trên một phần dữ liệu khác nhau với cùng phép toán. Mô hình song song dữ liệu thiết kế chủ yếu dành cho máy tính song song kiểu bộ xử lý mảng.



Hình 9 : Mô hình lập trình song song dữ liệu

1.1.4. Nguyên lý thiết kế giải thuật song song

Khi xử lý song song phải xét đến kiến trúc máy tính và giải thuật song song. Những giải thuật mà trong đó có một số thao tác có thể thực hiện đồng thời được gọi là giải thuật song song.

Khi thiết kế giải thuật song song, cần phải thực hiện:

- Chia bài toán thành những phần nhỏ hơn tương đối độc lập với nhau (phân chia về mặt dữ liệu hay chức năng) và giải quyết chúng một cách song song.
- Chỉ ra cách truy cập và chia sẻ dữ liệu.
- Phân các tác vụ cho các tiến trình (cho bộ xử lý).
- Các tiến trình được đồng bộ ra sao.

1.1.4.1. Nhận thức vấn đề và chương trình có thể song song hóa

Bước đầu tiên trong việc phát triển phần mềm song song là hiểu được vấn đề, hiểu được công việc chúng ta cần giải quyết song song. Nếu chúng ta đang bắt đầu với một chương trình tuần tự, điều này đòi hỏi sự hiểu biết về giải thuật, mã chương trình và cả ngôn ngữ lập trình của chương trình tuần tự đó. Sau đó chúng ta thực hiện các bước sau để kiểm tra xem một bài toán tuần tự có thể song song hóa được hay là không? Các bước như sau:

- Bước đầu tiên chúng ta phải phân tích bài toán, xác định các (điểm nóng) vị trí trong chương trình có thể hoặc không thể song song hóa được.
- Nhận diện các hạn chế xử lý song song, hạn chế phổ biến nhất là sự phụ thuộc dữ liệu, ví dụ như việc tính số hạng của dãy Fibonacci hoặc tính n!.

1.1.4.2. Ví dụ về bài toán có thể song song hóa được

Bài toán cộng hai mảng số nguyên có cùng n phần tử, việc cộng 2 phần tử nào đó là độc lập với các phần tử còn lại. Vấn đề này có thể được giải quyết song song:

A	1	1	1	1	1
+					
B	4	4	4	4	4
=					
C	5	5	5	5	5

1.1.4.3. Ví dụ về bài toán không thể song song hóa được

Tính chuỗi Fibonacci (1, 1, 2, 3, 5, 8, 13, 21,…) bằng cách sử dụng công thức: F(k+2)= F(k+1) + F(k), với n > 1.

Đây là một bài toán không thể song song hóa được bởi vì tính số hạng của dãy Fibonacci theo công thức là phụ thuộc chứ không phải là độc lập. Việc tính giá trị thứ k+2 phải sử dụng giá trị của cả hai giá trị k+1 và k.

1.1.4.4. Phương pháp xây dựng thuật toán song song

Phương pháp tổng quát được áp dụng trong bài nghiên cứu này là phân chia dữ liệu

- Được sử dụng khi vấn đề có liên quan đến các tính toán hay chuyển đổi trên một hay nhiều cấu trúc dữ liệu và cấu trúc này có thể được phân chia và tính toán trên từng phần nhỏ.

Một bài toán ví dụ để dễ dàng thấy việc phân chia dữ liệu trên cùng một tập dữ liệu là bài toán cộng hai mảng số nguyên. Giả sử ta có p bộ xử lý (processor) cùng làm việc để cộng hai mảng A[0…N-1] và B[0…N-1] lưu vào mảng kết quả C[0...N-1], việc phân chia dữ liệu sẽ đặt N/p phần tử của mỗi mảng vào từng quá trình và nó sẽ tính toán N/p phần tử tương ứng của mảng kết quả. Như vậy, với p bộ xử lý (processor) càng nhiều thì thời gian chạy càng nhanh, ngược lại thì chạy càng chậm.

Tổng quan về GPU

1.1.5. Giới thiệu GPU

Bộ xử lý đồ họa (**G**raphics **P**rocessing **U**nit) hay gọi tắt là GPU là bộ xử lý chuyên dụng cho biểu diễn hình ảnh 3D từ bộ vi xử lý của máy tính. Nó được sử dụng trong các hệ thống nhúng, điện thoại di động, máy tính cá nhân, máy trạm, và điều khiển game. Bộ xử lý đồ họa ngày nay rất hiệu quả trong các thao tác đồ họa máy tính, và cấu trúc song song cao cấp làm cho chúng có năng lực xử lý tốt hơn nhiều so với bộ vi xử lý thông thường trong các thuật toán phức tạp. Trong máy tính cá nhân, một GPU được biết tới như một card màn hình (video card) hoặc được tích hợp luôn trên bảng mạch chủ. Hơn 90% các máy tính cá nhân hoặc máy tính xách tay hiện đại đã có tích hợp GPU nhưng thường yếu hơn nhiều so với GPU tích hợp trên các card màn hình chuyên dụng.

1.1.6. Lịch sử phát triển GPU

GPU [~32] là bộ xử lý gắn với card đồ họa, chuyên dùng tính toán các phép toán dấu phảy động. Sự phát triển của card đồ họa kết hợp chặt chẽ với các chip vi xử lý.

Ban đầu GPU là bộ xử lý gắn trên card đồ họa phục vụ việc tính toán cho các phép toán dấu phảy động.

Bộ gia tốc đồ họa kết hợp với các vi mạch siêu nhỏ tùy chọn chứa một số phép toán đặc biệt được sử dụng phổ biến trong biến đổi thành đồ họa ba chiều (graphic rendering). Khả năng của các vi mạch từ đó xác đinhj khả năng của bộ gia tốc đồ họa. Chúng được sử dụng chủ yếu trong các trò chơi 3B, hoặc biến đổi thành đầu ra 3D.

GPU thực thi một số phép toán đồ họa nguyên thủy làm chúng chạy nhanh hơn rất nhiều so với việc vẽ trực tiếp trên màn hình với CPU.

- **Những năm 1970**:

Hãng sản xuất chip ANTIC và CTIA đã đưa ra bộ điều khiển phần cứng cho việc kết hợp đồ họa và chế độ text, tính toán vị trí và hiển thị (theo khuôn dạng phần cứng hỗ trợ) và những hiệu ứng khác trên các máy tính ATARI 8-bit. Chíp ANTIC là một bộ xử lý chuyên biệt cho ánh xạ (theo cách lập trình được) giữa text và dữ liệu đồ họa tới đầu ra video. Nhà thiết kế chip ANTIC, Jay Miner, sau đó đã thiết kế chip đồ họa cho Commodore Amiga.

- **Những năm 1980:**

Commodore Amiga là máy tính thương mại đầu tiên có chứa các bộ blit (**BL**ock **I**mage **T**ransfer là sự chuyển động của một bitmap lớn trong game 2D) trong phần cứng

video của nó, hệ thống đồ họa 8514 của IBM là một trong những card video đầu tiên trên PC có thể thực thi các phép toán 2D nguyên thủy trên phần cứng.

Amiga đã là thiết kế duy nhất, theo thời gian, những tính năng của nó bây giờ được công nhận là bộ gia tốc đồ họa đầy đủ, giảm tải thực tế tất cả các chức năng thế hệ video cho phần cứng, bao gồm vẽ đường thẳng, tô màu vùng, chuyển khối hình ảnh, và bộ đồng xử lý đồ họa với cùng với tập các chỉ thị lệnh nguyên thủy của riêng nó. Trước đó (và sau một thời gian khá dài trên hầu hết hệ thống) CPU sử dụng vào mục đích chung đã phải xử lý mọi khía cạnh của việc vẽ hình ảnh hiển thị.

- **Những năm 1990:**

Năm 1991, S3 Graphics giới thiệu bộ gia tốc chip 2D đầu tiên, các 86C911 S3 (mà nhà thiết kế của nó đặt theo tên của Porsche 911 với ý nghĩa thể hiện dấu hiệu của sự gia tăng hiệu suất như đã cam kết). Các 86C911 sinh ra một máy chủ của các bắt trước: năm 1995, tất cả các nhà sản xuất chip đồ họa máy tính lớn đã thêm vào các hỗ trợ tăng tốc 2D cho chip của họ. Bởi thời gian này, bộ tăng tốc Windows với đặc tính cố định chức năng nói chung đắt tiền đã vượt bộ đồng xử lý đồ họa mục đích chung trong hiệu suất Windows, và các bộ đồng xử lý phai mờ dần trong các thị trường PC.

Trong suốt những năm 1990, 2D GUI tiếp tục tăng tốc phát triển. Từ khả năng sản xuất được cải thiện đã tác động vào các mức độ tích hợp chip đồ họa. Thêm vào đó các giao diện lập trình ứng dụng (API) đem lại một lượng lớn tác vụ, chẳng hạn như thư viện đồ họa của Microsoft WinG cho Windows 3.x, và giao diện sau đó DirectDraw của họ cho tăng tốc phần cứng của game 2D trong Windows 95 và sau đó. Trong đầu và giữa thập niên 1990, với sự hỗ trợ CPU-thời gian thực, đồ họa 3D đã trở nên ngày càng phổ biến trong máy tính và giao diện điều khiển trò chơi, dẫn đến nhu cầu phát triển rộng rãi phần cứng tăng tốc đồ họa 3D.

Ví dụ đầu tiên về loạt trên thị trường phần cứng đồ họa 3D có thể được tìm thấy trong các trò chơi video thế hệ console thứ năm như PlayStation và Nintendo 64. Trong thế giới PC, lần thử đầu tiên không thành công đáng chú ý nhất cho các chip đồ họa 3D giá thành rẻ là ViRGES3

Hình 10: Ảnh chụp 3dfx Voodoo3

, ATI Rage, và Matrox Mystique. Những chip này về cơ bản là bộ gia tốc 2D thế hệ trước bổ sung thêm các tính năng 3D then chốt. Nhiều thành phần được thiết kế tương thích với thế hệ chip trước đó để dễ thực hiện và chi phí tối thiểu. Ban đầu,

hiệu năng đồ họa 3D đã chấp nhận được với bảng mạch rời dành riêng cho các chức năng tăng tốc 3D (thiếu chức năng 2D GUI) như 3dfx Voodoo. Tuy nhiên, như công nghệ sản xuất một lần nữa tiến triển, video, bộ tăng tốc 2D GUI, và chức năng 3D được tích hợp tất cả vào một con chip. chipset Verite của Rendition được là sản phẩm đầu tiên làm điều này và cũng đủ để được lưu ý.

OpenGL xuất hiện vào đầu những năm 90 như là API đồ họa chuyên nghiệp, nhưng đã trở thành một lực lượng chi phối trên máy tính, và là một động lực cho phát triển phần cứng. Triển khai phần mềm của OpenGL đã được phổ biến trong thời gian này mặc dù ảnh hưởng của OpenGL cuối cùng dẫn đến hỗ trợ phần cứng rộng rãi. Theo thời gian một sự lựa chọn nổi lên giữa các tính năng có sẵn bằng phần cứng và những tính năng đó cung cấp tại OpenGL. DirectX đã trở thành phổ biến với các nhà phát triển game Windows trong thời gian cuối những năm 90. Không giống như OpenGL, Microsoft khẳng định nghiêm ngặt về việc cung cấp sự hỗ trợ một-một của phần cứng. Cách tiếp cận đó đã làm DirectX ít phổ biến như là API đồ họa đứng một mình ngay từ đầu trong khi các GPU cung cấp nhiều tính năng đặc biệt của riêng mình, mà hiện đã được ứng dụng OpenGL có thể được hưởng lợi, để lại DirectX thường là một thế hệ sau. Theo thời gian, Microsoft đã bắt đầu làm việc chặt chẽ hơn với các nhà phát triển phần cứng, và bắt đầu nhắm mục tiêu các bản phát hành của DirectX với những phần cứng đồ họa hỗ trợ. Direct3D 5,0 là phiên bản API đầu tiên đang phát triển để đạt được áp dụng rộng rãi trên thị trường chơi game, và nó cạnh tranh trực tiếp với nhiều phần cứng cụ thể hơn, thường là các thư viện đồ họa độc quyền, trong khi OpenGL duy trì điều đó. Direct3D 7,0 hỗ trợ phần cứng tăng tốc biến đổi và ánh sáng (T & L). Bộ tăng tốc 3D biến đổi từ chỉ là bộ quét đường thẳng đơn giản đến có thêm phần cứng quan trọng dùng cho các đường ống dẫn biến đổi 3D. NVIDIA Geforce 256 (còn được gọi là NV10) là sản phẩm đầu tiên trên thị trường với khả năng này. Phần cứng biến đổi và ánh sáng, cả hai đều đã có trong OpenGL, có trong phần cứng những năm 90 và đặt tiền đề cho các phát triển sau đó là các đơn vị đổ bóng điểm ảnh và đổ bóng vector mà với đặc tính linh hoạt hơn và lập trình được.

- **Từ năm 2000 đến nay:**

Với sự ra đời của API OpenGL và các tính năng tương tự trong DirectX, GPU thêm vào tính năng đổ bóng lập trình được. Mỗi điểm ảnh bây giờ có thể được xử lý bởi một chương trình ngắn có thể bao gồm các cấu hình hình ảnh bổ xung là đầu vào, và mỗi vector hình học có thể được xử lý bởi một chương trình ngắn trước khi nó được chiếu lên màn hình. NVIDIA lần đầu tiên được sản xuất một con chip có khả năng lập trình đổ bóng, GeForce 3 (tên mã NV20). Tháng 10 năm 2002, với sự ra đời của ATI Radeon 9.700 (còn gọi là R300), bộ tăng tốc Direct3D 9.0 lần đầu tiên trên thế giới, bộ đổ bóng điểm ảnh và vector có thể thực hiện vòng lặp và các phép toán dấu phảy động dài, và nói chung đã nhanh chóng trở nên linh động như CPU, và đòi hỏi cần có bước

phát triển nhanh hơn cho các phép toán mảng liên quan đến hình ảnh (image-array operations). Đổ bóng điểm ảnh thường được sử dụng cho những thứ như lập bản đồ bump, thêm vào các kết cấu (texture), để làm cho một đối tượng trông bóng, ảm đạm, thô ráp, hoặc thậm chí căng mịn hoặc lỗi lõm.

Khi sức mạnh xử lý của GPU có tăng lên kéo theo nhu cầu nguồn điện cao hơn.GPU hiệu suất cao, thường được tiêu thụ năng lượng nhiều hơn các CPU hiện tại

Ngày nay, GPU song song đã bắt đầu thực hiện xâm nhập máy tính và cạnh tranh với CPU, và theo một nghiên cứu bên lề, gọi là GPGPU cho tính toán chung (General Purpose Computing) trên GPU, đã tìm thấy con đường của mình ứng dụng vào các lĩnh vực khác nhau như thăm dò dầu, xử lý hình ảnh khoa học, đại số tuyến tính, tái tạo 3D và hỗ trợ lựa chọn giá cổ phiếu. Điều này tăng áp lực lên các nhà sản xuất GPU từ "người dùng GPGPU" để cải tiến thiết kế phần cứng, thường tập trung vào việc thêm tính linh hoạt hơn cho mô hình lập trình.

1.1.7. Kiến trúc GPU

GPU luôn luôn là một bộ xử lý với dư thừa tài nguyên tính toán. Tuy nhiên xu hướng quan trọng nhất gần đây đó là trưng bày khả năng tính toán đó cho các lập trình viên. Những năm gần đây, GPU đã phát triển từ một hàm cố định, bộ xử lý chuyên dụng tới bộ xử lý lập trình song song, đầy đủ tính năng độc lập với việc bổ sung thêm các chức năng cố định, và các chức năng chuyên biệt. Hơn bao giờ hết các khía cạnh về khả năng lập trình của bộ xử lý chiếm vị trí trung tâm. Tôi bắt đầu bằng cách ghi chép lại sự tiến triển này, bắt đầu từ cấu trúc của đường ống dẫn đồ họa GPU và làm thế nào GPU trở thành kiến trúc, công cụ giành cho các mục đích thông dụng, sau đó đi xem xét kỹ hơn các kiến trúc của GPU hiện đại.

Đường ống dẫn đồ họa (Graphics Pipeline)

Các đầu vào của GPU là danh sách các hình học nguyên thủy, điển hình là tam giác, trong một thế giới không gian 3 chiều. Qua nhiều bước, những khối hình nguyên thủy đó được làm *bóng mờ* (shade) và được tô vẽ lên màn hình, nơi chúng được lắp ráp để tạo ra một hình ảnh cuối cùng. Đây là kiến thức cơ bản đầu tiên để giải thích các bước cụ thể trong đường ống dẫn kinh điển trước khi cho thấy làm cách nào mà các đường ống đã trở thành lập trình được [~3].

Các phép toán vector:

Các *hình học nguyên thủy* (*primary geometric)* được hình thành từ các vector riêng rẽ. Mỗi vector phải được chuyển thành không gian trên màn hình và có bóng mờ, thường thông bằng cách tính toán tương tác của chúng với các luồng ánh sáng trong một bối cảnh cụ thể. Bởi vì những bối cảnh tiêu biểu có thể có hàng chục đến hàng trăm ngàn vector, và mỗi vector có thể được tính toán độc lập. Do đó kịch bản này là rất phù hợp

cho phần cứng song song.

Thành phần nguyên thủy:

Các vector được lắp ráp vào các hình tam giác, đó chính là phần tử hỗ trợ phần cứng cơ bản trong GPU ngày nay.

Sự quét mành:

Quét mành (rasterization) là quá trình xác định những vị trí điểm ảnh nào trong không gian màn hình được bao chứa bởi mỗi tam giác. Mỗi tam giác tạo ra một thành tố nguyên thủy được gọi là "*mảnh*" tại các vị trí điểm ảnh trong không gian màn hình mà nó bao chứa. Và do nhiều tam giác có thể chồng lên nhau tại một vị trí điểm ảnh bất kỳ nên giá trị màu của mỗi điểm ảnh có thể được tính từ nhiều mảnh.

Thao tác trên mảnh:

Sử dụng thông tin màu sắc từ vector và có thể lấy dữ liệu bổ sung từ bộ nhớ toàn cục trong các hình dạng của sự kết hợp (sự kết hợp là hình ảnh được ánh xạ lên bề mặt), mỗi mảnh được làm bóng mờ để xác định màu sắc cuối cùng của nó. Cũng như trong kịch bản vector, mỗi mảnh có thể được tính toán song song. Giai đoạn này thường là đòi hỏi nhiều tính toán nhất trong đường ống dẫn đồ họa.

Thành phần:

Các mảnh được lắp ráp thành hình ảnh cuối cùng với một màu cho mỗi điểm ảnh, thường là bằng cách giữ lại mảnh gần ống kính nhất cho mỗi vị trí điểm ảnh. Trước đây, các phép toán hiện có tại khung cảnh vector và mảnh đã được cấu hình nhưng không thể lập trình được. Ví dụ, một trong những tính toán chính ở khung cảnh vector là tính toán các màu sắc ở mỗi vector như là một chức năng của thuộc tính vector và các độ sáng trong bối cảnh đó. Trong đường ống chức năng cố định, các lập trình viên có thể kiểm soát được vị trí và màu sắc của các vector và ánh sáng, nhưng không phải là mô hình chiếu sáng mà xác định tương tác giữa chúng.

Tiến hóa của kiến trúc GPU

Các đường ống chức năng cố định thiếu tính tổng quát để có biểu diễn hiệu quả các trường hợp làm bóng mờ phức tạp hơn và các phép toán ánh sáng, mà đó lại là những điều kiện tiên quyết cho các hiệu ứng phức tạp. Bước then chốt trên đã được thay thế bằng các hàm cố định chức năng trên mỗi vector và các phép toán trên mỗi mảnh với chương trình chỉ định người sử dụng chạy trên từng vector và từng mảnh. Trong hơn sáu năm qua, các chương trình vector và chương trình mảnh đã có ngày càng nhiều khả năng, với giới hạn lớn hơn về kích cỡ và tiêu thụ tài nguyên, với bộ chỉ thị (tập lệnh) đầy đủ tính năng, và với các phép toán điều khiển luồng linh hoạt hơn.

Sau nhiều năm của các bộ chỉ thị lệnh riêng rẽ cho các phép toán trên vector và mảnh,

GPU hiện tại hỗ trợ mô hình bóng mờ thống nhất 4.0 (unified Shader Model 4.0) trên cả bóng mờ vector và mảnh [~3]:

- Các phần cứng phải hỗ trợ các chương trình đổ bóng mờ ít nhất là 65 nghìn (65k) chỉ thị tĩnh và chỉ thị động không giới hạn.
- Các tập lệnh, lần đầu tiên, hỗ trợ cả số nguyên 32 bit và số dấu phảy động 32 bit.
- Các phần cứng phải cho phép số lượng tùy ý thao tác đọc trực tiếp và gián tiếp từ bộ nhớ toàn cục (kết cấu - texture).
- Cuối cùng, điều khiển luồng động trong các dạng vòng lặp và rẽ nhánh phải được hỗ trợ.

Khi mô hình đổ bóng ra đời và phát triển mạnh hơn, tất cả các loại ứng dụng GPU đã tăng độ phức tạp chương trình vector và mảnh, kiến trúc GPU ngày càng tập trung vào các bộ phận lập trình được của đường ống dẫn đồ họa. Quả thực, trong khi các thế hệ trước đây của GPU có thể được mô tả chính xác nhất như là phần thêm vào khả năng lập trình được cho đường ống chức năng cố định, GPU ngày nay được khắc họa tốt hơn, như là công cụ lập trình được bao quanh bởi các đơn vị hỗ trợ có chức năng cố định.

Kiến trúc của GPU hiện đại

Trong phần giới thiệu, chúng tôi ghi nhận rằng GPU được xây dựng cho các nhu cầu ứng dụng khác nhau so với CPU, đó là các yêu cầu tính toán lớn chạy song song, với trọng tâm là thông lượng hơn là độ trễ. Do đó, các kiến trúc của GPU phát triển theo một hướng khác so với CPU.

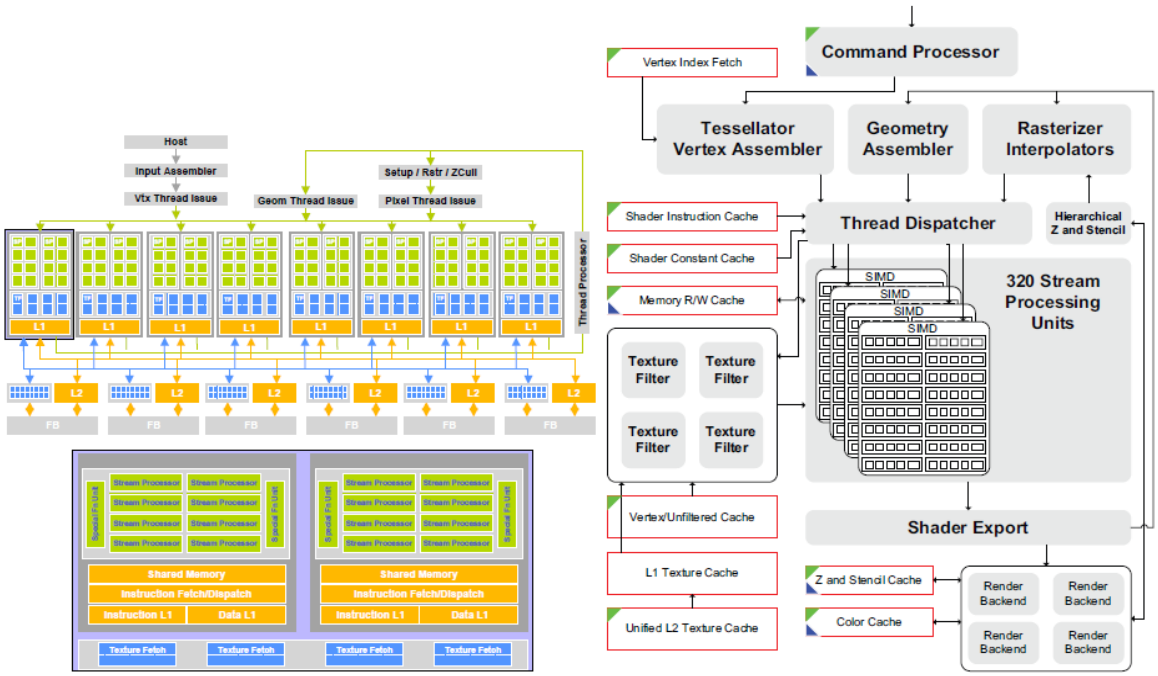
Xem xét một đường ống dẫn của các tác vụ (task), như chúng ta thấy ở hầu hết các giao diện lập trình đồ họa (và như ở nhiều ứng dụng khác) phải xử lý một lượng lớn các yếu tố đầu vào. Trong một đường ống dẫn như vậy, đầu ra của mỗi nhiệm vụ thành công được đưa vào đầu vào của các tác vụ tiếp theo. Đường ống đặt ra cơ chế song song ứng dụng, như là dữ liệu trong nhiều khung cảnh trong đường ống có thể được tính cùng một thời điểm; trong từng khung cảnh, tính toán nhiều hơn một phần tử tại một thời điểm là cơ chế song song dữ liệu. Để thực hiện loại đường ống như vậy, CPU có thể lấy một phần tử đơn (hoặc nhóm các phần tử) và xử lý khung cảnh (stage) đầu tiên trong đường ống, sau đó các khung cảnh tiếp theo cũng làm như vậy. CPU chia đường ống dẫn theo thời gian, áp dụng tất cả các nguồn lực của bộ xử lý vào trong từng khung cảnh khi đến lượt.

GPU có lịch sử lấy một cách tiếp cận khác CPU. GPU phân chia các nguồn lực của bộ xử lý theo các khung cảnh khác nhau, sao cho đường ống được chia theo không gian chứ không phải thời gian. Các phần của bộ vi xử lý làm việc trên một trong những khung cảnh cấp dữ liệu đầu ra trực tiếp vào một phần khác mà sẽ hoạt động

trong giai đoạn tiếp theo. Cơ chế tổ chức này đã rất thành công tại GPU cố định chức năng vì hai lý do. Đầu tiên, phần cứng trong bất kỳ khung cảnh nào có thể khai thác cơ chế song song dữ liệu trong khung cảnh đó, xử lý nhiều phần tử cùng một lúc, và vì nhiều cơ chế song song công việc được chạy bất kỳ lúc nào, GPU có thể đáp ứng nhu cầu tính toán rất lớn của các đường ống dẫn đồ họa. Thứ hai, phần cứng của mỗi khung cảnh có thể được tùy chỉnh với phần cứng chuyên dụng cho công việc đã đưa ra của nó, cho phép tính toán lớn hơn đáng kể và mức độ hiệu quả vượt qua giải pháp cho mục đích chung. Ví dụ, giai đoạn rasterization, cần tính thông tin bao phủ điểm ảnh của từng điểm ảnh tam giác đầu vào, là hiệu quả hơn khi thực hiện trên phần cứng dụng. Theo các khung cảnh lập trình được (chẳng hạn như các chương trình vector và mảnh) thay thế khung cảnh cố định chức năng, các mục đích chuyên dụng, các thành phần cố định chức năng được đơn giản thay thế bằng thành phần lập trình được, nhưng nhiệm vụ tổ chức thực hiện song song không thay đổi.

Kết quả là một đường ống GPU dài, có tính chất feed-forward có nhiều khung cảnh, mỗi khung cảnh thường tăng tốc cho một mục đích đặc biệt, và thích hợp với phần cứng song song. Trong CPU, bất kỳ phép toàn nào cũng có thể mất khoảng 20 chu kỳ hoạt động theo thứ tự tính từ lúc bắt đầu đến khi rời khỏi đường ống CPU. Trên GPU, một phép toán đồ họa cho trước có thể mất hàng ngàn chu kỳ từ khi bắt đầu đến khi kết thúc. Độ trễ của bất kỳ phép toán nào thường là lâu. Tuy nhiên, cơ chế song song tác vụ và dữ liệu từ khung cảnh này tới khung cảnh khác và giữa các khung cảnh tạo ra thông lượng cao. Bất lợi chính của đường ống GPU song song tác vụ là vấn đề cân bằng tải. Giống như bất kỳ đường ống nào, hiệu suất của đường ống GPU phụ thuộc vào khung cảnh chậm nhất của nó. Nếu các chương trình vector rất phức tạp và chương trình mảnh là đơn giản, tổng thể thông qua là phụ thuộc vào hiệu suất của các chương trình vector. Trong những ngày đầu của các khung cảnh lập trình được, tập chỉ thị của các chương trình vector và các chương trình mảnh khá khác nhau, do đó, những khung cảnh này được tách biệt. Tuy nhiên, khi cả hai chương trình vector và chương trình mảnh trở nên đầy đủ tính năng, và tập chỉ thị lệnh hội tụ như nhau, kiến trúc GPU xem xét lại đường ống song song tác vụ nghiêm ngặt trong lợi thế của kiến trúc đổ bóng hợp nhất (unified shader), trong đó tất cả đơn vị lập trình được trong đường ống chia sẻ một đơn vị phần cứng lập trình được duy nhất. Trong khi phần lớn các đường ống vẫn còn là song song tác vụ, các đơn vị lập trình bây giờ phân chia thời gian của nó giữa công việc vector, công việc mảnh, và công việc hình học (với DirectX có bộ đổ bóng 10 loại hình học khác nhau). Các đơn vị này có thể khai thác cả hai cơ chế song song tác vụ và song song dữ liệu. Khi các bộ phận lập trình được của đường ống chịu trách nhiệm tính toán ngày càng nhiều trong các đường ống dẫn đồ họa thì kiến trúc của GPU chuyển từ kiến trúc song song tác vụ trong một đường ống nghiêm ngặt sang kiến trúc được phát triển xung quanh một đơn vị lập trình được theo cơ chế song song dữ liệu thống nhất. AMD giới thiệu các kiến trúc đổ bóng hợp nhất đầu tiên cho

sản phẩm GPU Xenos GPU của nó trong Xbox 360 (2005). Ngày nay, cả GPU của AMD và NVIDIA đều có tính năng đổ bóng hợp nhất (unified shaders) (hình 10). Lợi ích cho người sử dụng GPU là cân bằng tải tốt hơn với chi phí cho phần cứng phức tạp hơn. Lợi ích cho người dùng GPGPU đã rõ ràng: với tất cả nguồn lực lập trình được trong một đơn vị phần cứng duy nhất, lập trình viên GPGPU bây giờ có thể tiếp cận đơn vị lập trình được theo cách trực tiếp, hơn hẳn trước cách tiếp cận trước đây là phân chia công việc trên nhiều đơn vị phần cứng.



Hình 11: Kiến trúc GPU của NVIDIA và AMD có một lượng đồ sộ các đơn vị lập trình được tổ chức song song thống nhất

1.1.8. Tính toán trên GPU

Phần trên chúng ta đã thấy kiến trúc phần cứng của GPU, chúng ta quay sang mô hình lập trình của nó.

1.1.8.1. Mô hình lập trình trên GPU

Các đơn vị lập trình của GPU tuân theo mô hình lập trình SPMD (single program, multiple data): đơn chương trình, đa dữ liệu. Để hiệu quả, GPU xử lý rất nhiều yếu tố (vector hoặc mảnh) song song bằng cách sử dụng nhiều chương trình giống nhau. Mỗi phần tử được độc lập với các phần tử khác, và trong lập trình mô hình cơ sở, các yếu tố không thể giao tiếp với nhau. Tất cả các chương trình GPU phải được

tổ chức theo cách: song song nhiều thành phần, mỗi thành phần được xử lý song song bởi một đơn chương trình. Mỗi thành phần có thể hoạt động trên số nguyên 32-bit hay dữ liệu dấu phảy động với một tập các chỉ thị lệnh vừa đủ dùng cho mục đích thông dụng (general purpose). Các thành phần có thể đọc dữ liệu từ một bộ nhớ chia sẻ toàn cầu (hoạt động "thu thập" (gather)thông tin) và, với GPU mới nhất, cũng ghi trở lại vị trí tùy ý trong bộ nhớ chia sẻ toàn cầu (hoạt động "phát tán" (scatter) thông tin). Đây là mô hình lập trình rất phù hợp với các chương trình làm việc với đường thẳng, như nhiều thành phần có thể được xử lý trong các bước nối tiếp có mã chạy chính xác như nhau. Câu lệnh được viết ra theo cách này được gọi là "SIMD", dùng cho đơn chỉ thị lệnh, đa dữ liệu. Khi chương trình đổ bóng trở nên phức tạp hơn, các lập trình viên thích cho phép các phần tử khác nhau có đường đi khác nhau thông qua chương trình giống nhau, dẫn đến mô hình SPMD tổng quát hơn. Mô hình này được hỗ trợ trên GPU như thế nào?

Một trong những lợi ích của GPU là phần lớn tài nguyên dành cho việc tính toán. Việc cho phép các con đường thực thi khác nhau cho từng phần tử đòi hỏi đáng kể phần cứng điều khiển. Thay vào đó, GPU ngày nay hỗ trợ luồng điều khiển riêng cho từng luồng, nhưng áp đặt một hình phạt nặng cho những phân nhánh tạp nham. Các nhà cung cấp GPU phần lớn thông qua cách tiếp cận này. Các yếu tố được nhóm lại với nhau thành những khối và các khối được xử lý song song. Nếu các yếu tố phân nhánh ra các hướng khác nhau trong một khối, thì phần cứng tính cả hai bên của nhánh cho tất cả các phần tử trong khối. Kích cỡ của khối được giảm với thế hệ GPU gần đây, ngày này đó là thứ tự của 16 phần tử.

Trong khi viết chương trình trên GPU thì rẽ nhánh được phép nhưng không miễn phí. Người lập trình tổ chức mã nguồn của họ sao cho khối có rẽ nhánh mạch lạc sẽ tận dụng phần cứng tốt nhất.

1.1.8.2. Tính toán thông dụng trên GPU (GPGPU)

GPGPU là việc ánh xạ các bài toán tính toán mục đích thông thường lên GPU sử dụng phần cứng đồ họa theo cách giống như bất cứ ứng dụng đồ họa chuẩn nào. Bởi vì sự tương tự này, nó vừa dễ dàng hơn và cũng khó khăn hơn trong việc giải thích quá trình hoạt động. Một mặt, các hoạt động thực tế là như nhau và rất dễ làm theo. Mặt khác, thuật ngữ này có điểm khác nhau giữa đồ họa và sử dụng cho mục đích thông thường. Harris cung cấp một mô tả tuyệt vời của quá trình ánh xạ này [3]. Chúng tôi bắt đầu bằng cách mô tả lập trình trên GPU sử dụng các thuật ngữ đồ họa, sau đó cho thấy cách các bước tương tự được sử dụng theo cách thông thường để tạo ra ứng dụng GPGPU, và cuối cùng là sử dụng các bước tương tự để thể hiện đơn giản hơn và trực tiếp hơn về cách ngày nay các ứng dụng tính toán trên GPU được viết như thế nào.

- **Lập trình GPU cho đồ họa:**

Chúng tôi bắt đầu với cùng một đường ống dẫn GPU mà chúng ta đã mô tả ở trên và tập trung vào các khía cạnh lập trình được của đường ống này.

- Lập trình viên xác định dạng hình học sẽ bao phủ một khu vực trên màn hình. Quá trình quét mành trên màn hình tạo ra một mảnh ở mỗi vị trí điểm ảnh được bao phủ bởi hình học đó.
- Mỗi mảnh được làm bóng mờ của chương trình mảnh.
- Các chương trình mảnh tính giá trị của các mảnh bằng cách kết hợp của phép toán toán học và bộ nhớ toàn cục đọc từ bộ nhớ kết cấu toàn cục.
- Các hình ảnh kết quả sau đó có thể được sử dụng như là kết cấu trong tương lai

  đi qua các đường ống dẫn đồ họa.

- **Lập trình GPU cho các chương trình mục đích thông dụng (cũ):**

Đồng lựa chọn đường ống dẫn này để thực hiện tính toán general-purpose liên quan đến cùng các bước cụ thể giống nhau, nhưng ký hiệu khác nhau.

Một ví dụ tích cực là một mô phỏng tính chất lỏng được tính toán trên lưới: tại mỗi bước, chúng tôi tính toán trạng thái tiếp theo của chất lỏng cho mỗi điểm lưới từ tình trạng hiện tại trên lưới của nó và trạng thái các điểm hàng xóm của nó trên lưới.

- Lập trình viên chỉ rõ một hình nguyên thủy bao gồm một miền tính toán ưa thích. Các chương trình quét mành tạo ra một mảnh (fragment) ở mỗi vị trí điểm ảnh trong hình đó. (Trong ví dụ của chúng tôi, màu gốc phải bao phủ một mạng lưới các mảnh bằng với kích thước của chất lỏng mô phỏng.)
- Mỗi mảnh được làm bóng mờ bởi chương trình gerenal - purpose SPMD. (Mỗi

  điểm lưới chạy cùng một chương trình để cập nhật tình trạng chất lỏng của nó).

- Các chương trình mảnh (fragment program) tính giá trị của mảnh bằng cách kết hợp các phép toán toán học và các truy cập "thu thập" từ bộ nhớ toàn cục. Mỗi điểm lưới có thể truy cập trạng thái của các láng giềng của nó ở bước tính toán trước đó trong khi tính toán giá trị hiện tại của nó.
- Các bộ nhớ đệm chứa kết quả trong bộ nhớ toàn cục sau đó có thể được sử dụng như là một đầu cho các chu kỳ tiếp theo trong tương lai. Các trạng thái hiện tại của chất lỏng sẽ được sử dụng trên các bước tiếp theo.

- **Lập trình GPU cho chương trình mục đích thông dụng (mới):**

Một trong những khó khăn trong lịch sử lập trình ứng dụng GPGPU đó là mặc dù

các tác vụ general-purpose của chúng không có liên quan gì tới đồ họa, các ứng dụng vẫn phải được lập trình bằng cách sử dụng các API đồ họa. Ngoài ra, chương trình đã được cấu trúc trong điều kiện của đường ống đồ họa, với các đơn vị lập trình được chỉ có thể truy cập được như một bước trung gian trong đường ống, trong khi các lập trình viên chắc chắn muốn truy cập vào các đơn vị lập trình được trực tiếp.

Các môi trường lập trình chúng tôi mô tả chi tiết trong Mục Môi trường phần mềm, được giải quyết khó khăn này bằng cách cung cấp một giao diện tự nhiên hơn, trực tiếp hơn, không có giao diện đồ họa cho phần cứng và đặc biệt là các đơn vị lập trình được. Ngày nay, ứng dụng tính toán GPU được tổ chức theo cách sau:

1) Các lập trình viên trực tiếp xác định tên miền tính toán ưa thích như một lưới cấu trúc của các *luồng (thread).*

2) Chương trình general-purpose SPMD tính giá trị của từng luồng.

3) Các giá trị cho mỗi luồng được tính bằng cách kết hợp các phép toán toán học và cả truy cập "thu thập" (đọc) và "scatter" (ghi) bộ nhớ toàn cục. Không giống như hai phương pháp trước đó, cùng một bộ đệm có thể được dùng cho cả đọc và ghi, cho phép thêm các thuật toán mềm dẻo hơn (ví dụ, các thuật toán sử dụng ít bộ nhớ).

4) Các vùng đệm chứa kết quả trong bộ nhớ toàn cục sau đó có thể được sử dụng như là một đầu vào của tính toán sau đó.

Mô hình lập trình này mạnh vì một số lý do sau. Đầu tiên, nó cho phép các phần cứng khai thác triệt để cơ chế song song dữ liệu của các ứng dụng bằng cách xác định rõ ràng cơ chế song song trong chương trình. Tiếp theo, nó gây ấn tượng bằng việc tạo ra sự cân bằng vững chắc giữa tính phổ biến (một thủ tục hoàn toàn có thể lập trình tại mỗi phần tử) và sự hạn chế để đảm bảo hiệu năng tốt (mô hình SPMD, có các hạn chế về phân nhánh cho hiệu quả, có hạn chế về dữ liệu giao tiếp giữa các thành phần và giữa hạt nhân /chu kỳ, v.v..). Cuối cùng, khả năng truy cập trực tiếp đến các đơn vị lập trình được đã loại bỏ nhiều thách thức phức tạp của các lập trình viên GPGPU trước đây trong việc đồng thời chọn giao diện đồ họa cho lập trình mục đích thông dụng.

Kết quả là các chương trình thường được thể hiện bằng ngôn ngữ lập trình quen thuộc (chẳng hạn như ngôn ngữ lập trình của NVIDIA giống như cú pháp của C thể hiện trong môi trường lập trình CUDA của họ) và đơn giản hơn và dễ dàng hơn để xây dựng và gỡ lỗi (và đang ngày càng hoàn thiện như là các công cụ lập trình độc lập). Điều đó tạo nên một mô hình lập trình cho phép người dùng của mình tận dụng đầy đủ các sức mạnh phần cứng của GPU nhưng cũng cho phép mô hình lập trình mức cao ngày càng tăng giúp sản xuất của các ứng dụng phức tạp.

1.1.9. Môi trường phần mềm

Trong quá khứ, phần lớn các chương trình GPGPU được thực hiện trực tiếp thông qua các API đồ họa. Mặc dù nhiều nhà nghiên cứu đã thành công làm cho các ứng dụng làm việc thông qua các API đồ họa nhưng có một điều không phù hợp cơ bản giữa mô hình lập trình truyền thống mà mọi người đang dùng và các mục tiêu của các API đồ họa. Ban đầu, người ta sử dụng các hàm cố định, các đơn vị đồ họa cụ thể (ví dụ như các bộ lọc kết cấu (texture filter), trộn (blending), và các phép toán tạo mẫu tô đệm để thực hiện các thao tác GPGPU. Điều này nhanh chóng tốt hơn với phần cứng là bộ xử lý các mảnh hoàn toàn lập trình được với ngôn ngữ assembly mã giả, nhưng cách này vẫn khó tiếp cận cho dù đã có tất cả các nhà nghiên cứu những hăng hái nhất bắt tay vào. Với DirectX 9, lập trình đổ bóng cao cấp đã được thực hiện có thể thông qua ngôn ngữ đổ bóng cấp cao ("high-level shading language" - HLSL), nó được biểu diễn giống như giao diện lập trình C cho lập trình đổ bóng. NVIDIA Cg cung cấp các tính năng tương tự như HLSL, nhưng đã có thể biên dịch ra nhiều đích và cung cấp ngôn ngữ lập trình cấp cao đầu tiên cho OpenGL. Ngôn ngữ đổ bóng OpenGL (OpenGL Shading Language - GLSL) bây giờ là ngôn ngữ đổ bóng tiêu chuẩn cho OpenGL. Tuy nhiên, vấn đề chính với Cg / HLSL / GLSL cho GPGPU là chúng vốn đã là ngôn ngữ đổ bóng. Tính toán vẫn phải được thể hiện bằng các thuật ngữ đồ họa như vector, kết cấu (texture), mảnh (fragment), và pha trộn (blending). Vì vậy, mặc dù bạn có thể làm tính toán thông dụng hơn với đồ họa API và ngôn ngữ đổ bóng, chúng vẫn phần lớn không tiếp cận được bởi các lập trình viên thông thường.

Những gì các nhà phát triển thực sự muốn là có được một ngôn ngữ cấp cao hơn được thiết kế để tính toán một cách rõ ràng và trừu tượng hóa tất cả các cơ chế đồ họa của GPU. BrookGPU [~9] và Sh [~25] là hai đầu dự án nghiên cứu đầu tiên với mục tiêu trừu tượng GPU như là bộ xử lý dòng (streaming processor). Mô hình lập trình dòng tổ chức chương trình để thực hiện song song và cho phép giao tiếp hiệu quả và truyền dữ liệu đồng thời phù hợp với các nguồn lực xử lý song song và hệ thống bộ nhớ có sẵn trên GPU. Một chương trình dòng bao gồm một tập các dòng (stream), các tập được sắp xếp dữ liệu, và hạt nhân (kernel), các hàm chức năng được thiết lập với từng phần tử trong tập các dòng tạo ra một hay nhiều dòng đầu ra.

Brook đi theo cách tiếp cận trừu tượng tính toán dòng đơn giản, để biểu diễn dữ liệu như là các dòng và tính toán như là các hạt nhân. Không có khái niệm về kết cấu vector, mảnh, hoặc trộn (blending) trong Brook. Hạt nhân là các tính toán được viết trong một tập hợp con giới hạn của C, đặc biệt là không có con trỏ và scatter (sự tán xạ- thao tác ghi bộ nhớ), với đầu vào, đầu ra định nghĩa trước, và trùm các dòng được sử dụng trong hạt nhân như một phần của định nghĩa của nó. Brook chứa các chức năng truy cập dòng như: lặp lại và thoát khỏi vòng lặp, rút gọn các dòng, và khả năng xác định tên miền, tập con các dòng để sử dụng như đầu vào và đầu ra. Những hạt nhân

được chạy cho mỗi phần tử trong miền các dòng đầu ra. Hạt nhân của người dùng được ánh xạ tới đoạn code đổ bóng cho mảnh và đến các dòng liên quan tới kết cấu.

Dữ liệu tải lên và tải về GPU được thực hiện thông qua các lời gọi đọc / ghi rõ ràng được phiên dịch thao tác cập nhật kết cấu và cập nhật vào bộ đệm phản hồi. Cuối cùng, tính toán được thực hiện bởi một biến đổi vào không gian 3 chiều vùng các điểm ảnh trong miền đầu ra.

Dự án Microsoft's Accelerator (bộ gia tốc của Microsoft) [6] có mục tiêu tương tự như Brook ở chỗ tập trung vào khía cạnh tính toán, nhưng thay vì sử dụng biên dịch offline, bộ gia tốc dựa vào biên dịch tức thời (just-in-time) của các phép toán dữ liệu song song cho bộ đổ bóng mảnh. Không giống như mô hình của Brook và Sh được phần lớn các phần mở rộng từ C, bộ gia tốc là ngôn ngữ dựa trên mảng (array-base language) phát triển từ ngôn ngữ C #, và tất cả các tính toán được thực hiện thông qua các phép toán trên các mảng. Không giống như Brook, nhưng tương tự như Sh, mô hình đánh giá độ trễ cho biên dịch tức thời tích cực hơn dẫn đến khả năng chuyên biệt hơn và tối ưu code tạo ra để thực hiện trên GPU.

Trong năm qua, đã có những thay đổi lớn trong môi trường phần mềm cho phép phát triển các ứng dụng GPGPU dễ dàng hơn nhiều cũng như tạo ra các hệ thống phát triển mạnh mẽ hơn, chất lượng thương mại hơn. RapidMind [~24] thương mại hóa Sh và bây giờ đặt mục tiêu nhiều platform trong một GPU, các STI Cell Broadband Engine, và CPU đa-lõi, và hệ thống mới tập trung nhiều hơn nữa vào tính toán so với SH trong việc bao gồm nhiều phép toán đồ họa trung tâm.

Tương tự như bộ gia tốc của Microsoft, RapidMind sử dụng ước lượng độ trễ và biên dịch online để chụp lại và tối ưu hóa mã nguồn ứng dụng của người dùng cùng với các phép toán và mở rộng kiểu của C ++ để tạo ra những hỗ trợ trực tiếp cho mảng. PeakStream [8] là hệ thống mới, sáng tạo từ Brook, được thiết kế xoay quanh các phép toán trên mảng. Tương tự như RapidMind và bộ gia tốc, PeakStream chỉ sử dụng trong biên dịch tức thời, nhưng linh hoạt hơn nhiều trong việc vector hóa code của người dùng nhằm đạt hiệu suất cao nhất trên kiến trúc SIMD. PeakStream cũng là platform đầu tiên cung cấp hỗ trợ profiling và gỡ lỗi, là các khía cạnh mà sau đó tiếp tục là một vấn đề hóc búa trong phát triển GPGPU. Cả hai nỗ lực này giúp cho các nhà cung cấp của bên thứ ba tạo các hệ thống với sự hỗ trợ từ các nhà cung cấp GPU. Trong một buổi giới thiệu quảng cáo về các điều lý thú xung quanh GPGPU và sự thành công của phương pháp này cho tính toán song song, Google mua PeakStream trong năm 2007.

Cả AMD và NVIDIA bây giờ cũng có riêng hệ thống lập trình GPGPU. AMD công bố và phát hành hệ thống của họ cho các nhà nghiên cứu vào cuối năm 2006. CTM, hay "Close To The Metal", cung cấp mức trừu tượng phần cứng ở cấp thấp (HAL) cho dòng R5XX và dòng R6XX của GPU ATI. CTM-HAL cung cấp truy cập

mức assembly thô cho động cơ mảnh (bộ xử lý dòng - stream processor) cùng với bộ lắp ráp và bộ đệm lệnh để điều khiển thực thi trên phần cứng. Không tính năng đồ họa cụ thể nào được xuất qua các giao diện này. Tính toán được thực hiện bằng cách ràng buộc bộ nhớ như là đầu vào và đầu ra các bộ vi xử lý dòng, tải mã nhị phân ELF, và định nghĩa một miền các kết quả đầu ra mà trên đó để thực thi nhị phân. AMD cũng đưa ra tầng trừu tượng tính toán - Compute Abstraction Layer (CAL) . Tần này đưa thêm các cấu trúc (construct) cấp cao hơn, giống như thành phần tương tự trong hệt thống chạy của Brook, và hỗ trợ biên dịch GPU ISA cho GLSL, HLSL, và mã giả Assembly như Pixel Shader 3.0. Đối với lập trình cấp cao hơn, AMD hỗ trợ biên dịch các chương trình Brook trực tiếp đến phần cứng R6XX, cung cấp một mức lập trình trừu tượng cao hơn so với CAL hoặc HAL. NVIDIA CUDA là một giao diện cấp cao hơn HAL và CAL của AMD. Tương tự như Brook, CUDA cung cấp một cú pháp giống C để thực hiện trên GPU và biên dịch offline.

Tuy nhiên, không giống như Brook chỉ khai thác một hướng xử lý song song là song song dữ liệu thông qua cơ chế dòng, CUDA khai thác hai cấp xử lý song song là song song dữ liệu và đa luồng. CUDA cũng khai thác các nguồn tài nguyên phần cứng nhiều hơn Brook, làm lộ nhiều cấp độ của bộ nhớ hệ thống phân cấp; các thanh ghi theo từng luồng, bộ nhớ chia sẻ nhanh chóng giữa các luồng trong một khối, bộ nhớ bo mạch, và bộ nhớ máy chủ. Các hạt nhân trong CUDA cũng linh hoạt hơn trong Brook bằng cách cho phép sử dụng con trỏ (mặc dù dữ liệu phải ở trên bo mạch), việc lấy ra/lưu trữ thông thường vào bộ nhớ cho phép người sử dụng tán xạ (scatter) dữ liệu từ bên trong một hạt nhân, và đồng bộ giữa các luồng trong một khối luồng. Tuy nhiên, tất cả sự linh hoạt này và hiệu quả tiềm năng đạt được đi kèm với cái giá đòi hỏi người sử dụng phải hiểu nhiều hơn các chi tiết ở cấp thấp của phần cứng, đặc biệt là sử dụng thanh ghi, luồng và lập lịch cho khối luồng, và các hành vi của các mẫu truy cập bộ nhớ.

Tất cả các hệ thống này cho phép người phát triển xây dựng các ứng dụng lớn dễ dàng hơn. Ví dụ, Folding@Home GPU client và ứng dụng mô phỏng chất lỏng lớn được viết bằng BrookGPU, NAMD và VMD hỗ trợ thực thi trên GPU thông qua CUDA, RapidMind đã thử nghiệm mô phỏng chùm tia và sự hội tụ, và PeakStream đã biểu diễn dầu và khí đốt và các ứng dụng tính toán tài chính. CUDA cung cấp điều chỉnh và tối ưu hóa thư viện Blas và FFT để sử dụng như xây dựng khối cho các ứng dụng lớn. Truy cập cấp thấp vào phần cứng, như là cung cấp bởi CTM, hoặc hệ thống GPGPU cụ thể như CUDA, cho phép các người phát triển vượt qua một cách có hiệu quả các trình điều khiển đồ họa và duy trì ổn định hiệu năng và tính đúng đắn. Sự phát triển và tối ưu hóa trình điều khiển (driver) của các nhà cung cấp trong các API đồ họa có xu hướng chỉ để kiểm thử trên các trò chơi mới nhất và phổ biến nhất. Việc tối ưu được thực hiện để tối ưu hóa cho hiệu năng game có thể ảnh hướng tới tính ổn định và hiệu năng của các ứng dụng GPGPU.

1.1.10.Kỹ thuật và ứng dụng

Bây giờ chúng ta khảo sát một số đặc tính tính toán quan trọng, thuật toán, và các ứng dụng tính toán GPU. Chúng tôi lần đầu tiên nêu bật bốn phép toán song song dữ liệu tập trung ở tính toán GPU: thực hiện phép toán tán xạ (scatter) / tập hợp (gather) bộ nhớ, ánh xạ một chức năng vào nhiều yếu tố song song, giảm một bộ sưu tập các yếu tố thành một yếu tố hoặc một giá trị, và tính toán rút gọn cho trước một mảng song song. Chúng tôi nghiên cứu kỹ tính toán nguyên thủy cốt lõi ở một số chi tiết trước khi chuyển đến một cách nhìn tổng quan mức cao về các vấn đề thuật toán mà các nhà nghiên cứu đã nghiên cứu trên GPU: quét, sắp xếp, tìm kiếm, truy vấn dữ liệu, phương trình vi phân, và đại số tuyến tính. Các thuật toán cho phép một loạt các ứng dụng khác nhau, từ cơ sở dữ liệu, khai phá dữ liệu, đến các mô phỏng khoa học, như là động lực học và chuyển động nhiệt của chất lỏng (chúng ta sẽ xem kỹ hơn trong Phần VI và VII), chuyển động vật lý trong trò chơi và động lực học phân tử.

Tính toán nguyên thủy:

Các kiến trúc song song dữ liệu của GPU đòi hỏi thuật ngữ lập trình quen thuộc từ lâu với người sử dụng siêu máy tính song song, nhưng thường là mới với các lập trình viên ngày nay trưởng thành từ máy móc tuần tự hoặc cụm máy tính kết nối lỏng lẻo. Chúng ta thảo luận ngắn gọn về bốn các thành ngữ quan trọng: tán xạ / tập hợp (scatter/gather), ánh xạ, rút gọn, và quét. Chúng tôi mô tả những tính toán nguyên thủy này trong bối cảnh cả "Cũ" (dựa trên đồ họa) và "mới" (tính toán trực tiếp) trên tính toán GPU để nhấn mạnh sự đơn giản và tính linh hoạt của cách tiếp cận tính toán trực tiếp.

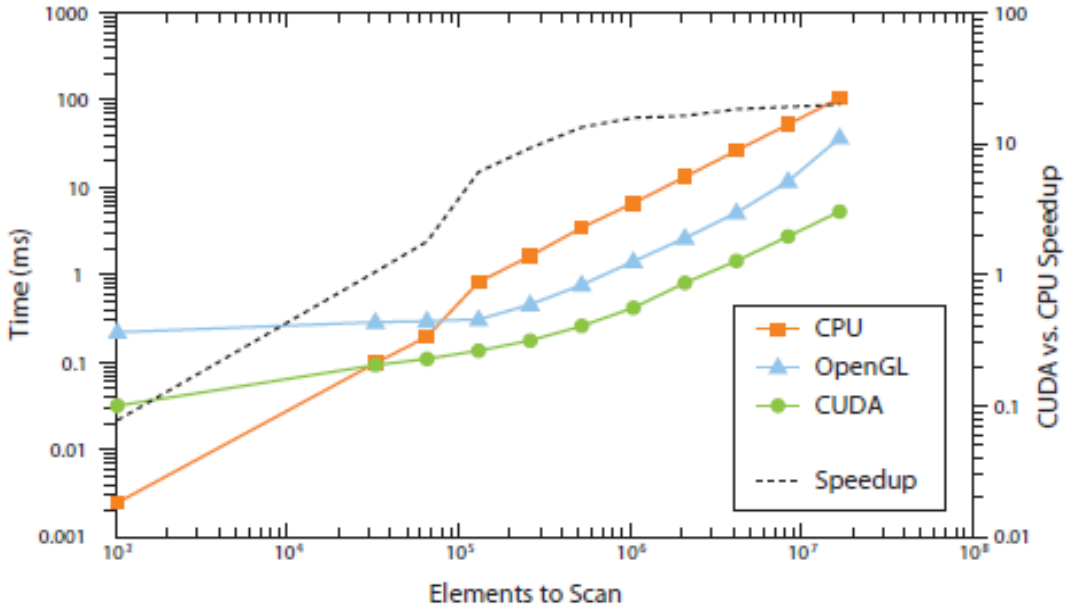
Tán xạ/tập hợp (scatter/gather) :

Viết vào hoặc đọc ra một vị trí được tính toán trong bộ nhớ. Tính toán GPU dựa trên đồ họa cho phép tập hợp hiệu quả bằng cách sử dụng các hệ thống con về kết cấu, lưu trữ dữ liệu như hình ảnh kết và đánh địa chỉ dữ liệu bằng cách tính toán tọa độ hình ảnh tương ứng và thực hiện phép nạp kết cấu. Tuy nhiên, hạn chế về kết cấu làm cho khó phát triển rộng rãi: hạn chế kích thước kết cấu đòi hỏi các mảng chứa trên 4.096 phần tử thành nhiều dòng của một kết cấu 2D, bổ sung thêm phép toán đánh địa chỉ, và phép nạp kết cấu đơn chỉ có thể lấy 4 giá trị dấu phảy động 32bit, hạn chế bộ nhớ lưu trữ mỗi phần tử. Phép tán xạ trong tính toán GPU dựa trên đồ họa khó khăn và đòi hỏi phải tái liên kết dữ liệu để thực thi như là các vector, hoặc sử dụng phép nạp kết cấu đỉnh hoặc render-to-vertex-buffer. Ngược lại lớp trực tiếp tính toán cho phép đọc và ghi không giới hạn đến các địa điểm tùy ý trong bộ nhớ. CUDA của NVIDIA cho phép người dùng truy cập vào bộ nhớ bằng cách sử dụng các cấu trúc C chuẩn (mảng,

con trỏ, biến); CTM của AMT cũng gần linh hoạt được như vậy, nhưng sử dụng địa chỉ 2D.

Ánh xạ (Map): áp dụng một phép toán mọi phần tử trong bộ sưu tập. Mô tả điển hình là vòng lặp *for* trong chương trình tuần tự (như là một luồng trên một CPU đơn lõi), một thực thi song song có thể giảm thời gian cần thiết bằng cách áp dụng phép toán đó đến nhiều phần tử song song. Tính toán GPU dựa trên đồ họa thực hiện phép ánh xạ như là chương trình mảnh được gọi từ bộ sưu tập điểm ảnh (một điểm ảnh cho mỗi phần tử). Từng chương trình mảnh của điểm ảnh đọc (fetch) dữ liệu từ kết cấu tại một ví trí tương ứng với vị trí của điểm ảnh trong hình ảnh đã biến đổi (render), thực thi phép toán đó, sau đó lưu trữ các kết quả tại điểm ảnh đầu ra. Tương tự, CTM và CUDA thường sinh ra một chương trình luồng để thực hiện phép toán đó trong nhiều luồng, với mỗi luồng nạp vào một phần tử, thực hiện tính toán, và lưu trữ kết quả. Lưu ý rằng vì vòng lặp hỗ trợ mỗi luồng có thể cũng lặp nhiều lần trên nhiều phần tử.

Rút gọn (Reduce): liên tục áp dụng một phép toán kết hợp nhị phân để rút gọn một tập hợp các phần tử thành một phần tử duy nhất hoặc một giá trị duy nhất. Ví dụ bao gồm việc tìm kiếm tổng (trung bình, tối thiểu, tối đa, phương sai, vv...) của một tập các giá trị. Một thực thi tuần tự trên CPU truyền thống sẽ lặp trên một mảng, tính tổng từng phần tử bằng cách chạy phép cộng tất cả các phần tử hiện có. Ngược lại, một rút gọn tổng theo cơ chế song song thực hiện nhiều lần phép cộng song song trên một tập thu hẹp các phần tử. Tính toán GPU dựa trên đồ họa thực hiện rút gọn dựa trên biến đổi (rendering) tập giảm dần các điểm ảnh. Trong từng biến đổi từng vượt qua chương trình mảnh đọc nhiều giá trị từ một kết cấu (thực thi khoảng 4 hoặc 8 lần đọc kết cấu), tính tổng đó, và ghi giá trị đó vào điểm ảnh đầu ra trong kết cấu khác (nhỏ hơn 4 hoặc 8 lần), mà sau đó sẽ bị ràng buộc như là đầu vào cho bộ đổ bóng mảnh tương tự và quá trình lặp đi lặp lại cho đến khi đầu ra là một điểm ảnh đơn chứa kết quả cuối cùng của quá trình rút gọn. CTM và CUDA cùng cho ra cùng một quá trình trực tiếp hơn, ví dụ bằng cách tạo ra một tập các luồng, mỗi luồng dọc 2 phần tử và ghi tổng của chúng vào một phần tử đơn. Một nửa số luồng lặp lại quá trình trên, sau đó là nửa còn lại, cứ như vậy cho đến khi còn lại một luồng sống sót sẽ ghi kết quả cuối cùng ra bộ nhớ.



Hình 12:Hiệu năng quét trên CPU, và GPU dựa trên đồ họa (sử dụng OpenGL), và GPU tính toán trực tiếp (sử dụng CUDA). Kết quả thực hiện trên GeForce 8800 GTX GPU và Intel Core2Duo Extreme 2.93 GHz CPU. Hình vẽ được lấy H. Nguyen (ed), GPU Gems 3, copyright (c) 2008 NVIDIA Corporation, published by Addison-Wesley Professional.

Quét (Scan):

Đôi khi được gọi là tổng tiền tố song song, quét lấy một mảng A các phần tử và trả về một mảng B có cùng chiều dài, trong đó mỗi phần tử B [i] đại diện cho một phép rút gọn mảng con A[1...i]. Quét là công cụ xây dựng khối dữ cực kỳ hữu ích cho thuật toán song song dữ liệu; Blelloch mô tả nhiều ứng dụng tiềm năng của quét từ Sắp xếp nhanh (quicksort) tới các phép toán ma trận thưa thớt[9]. Harris và đồng nghiệp[10] đã giới thiệu một thực thi của quét hiệu quả bằng cách sử dụng CUDA (hình 12); kết quả của họ minh họa cho những lợi thế của tính toán trực tiếp hơn là tính toán GPU dựa trên đồ họa. CUDA thực hiện nhanh hơn so với CPU bởi một một thừa số lên đến 20 và OpenGL bởi một thừa số lên đến 7.

1.1.11.Giải thuật và ứng dụng

Khi xây dựng phần lớn vào các phép toán nguyên thủy ở trên, các nhà nghiên cứu biểu diễn nhiều thuật toán mức cao và các ứng dụng khai thác các thế mạnh tính toán của GPU. Các thăm dò về các thuật toán tính toán GPU và các miền ứng dụng của có thể tham khảo ở [~13].

Sắp xếp (Sort): GPU đã có những cải thiện đáng kể trong sắp xếp từ khi cộng đồng tính toán trên GPU đã nghiên cứu lại, áp dụng, và cải thiện các

thuật toán sắp xếp, đáng chú ý là sắp xếp *bitonic merge* [~6]. Thuật toán "sorting network" này về bản chất là song song và mù, có nghĩa là các bước tương tự được thực hiện bất kể đầu vào. Govindaraju và các đồng nghiệp đã giành giải hiệu năng "PennySort" trong cuộc thi "TeraSort" năm 2005 [~29] bằng việc sử dụng hệ thống thiết kế cẩn thận và sự kết hợp của cải tiến nhiều thuật toán.

Tìm kiếm và truy vấn cơ sở dữ liệu (Search & database queries): Các nhà nghiên cứu cũng đã triển khai thực hiện một số hình thức tìm kiếm trên GPU, như tìm kiếm nhị phân (ví dụ: Horn [~4]) và tìm kiếm láng giềng gần nhất [~2], cũng như các thao tác cơ sở dữ liệu được xây dựng trên phần cứng đồ họa mục đích đặc biệt (gọi là bộ đệm độ sâu stencil) và các thuật toán sắp xếp nhanh ở trên [~28], [~27].

Phương trình vi phân (Differential equations): Những nỗ lực sớm nhất để sử dụng GPU cho tính toán phi đồ họa tập trung vào giải quyết các tập lớn phương trình vi phân. Phép tìm đạo hàm là một ứng dụng GPU phổ biến cho phương trình vi phân thường (ODEs), được sử dụng rất nhiều trong mô phỏng khoa học (ví dụ, hệ thống thăm dò lưu lượng của Kr¨uger [~15]) và tại các hiệu ứng trực quan cho các chò trơi trên máy tính. GPU đã được sử dụng nhiều để giải quyết các vấn đề trong phương trình vi phân riêng (PDEs) như phương trình Navier- Stokes cho dòng chảy tự do. ứng dụng đặc biệt thành công mà GPU PDE đã giải quyết bao gồm các động lực chất lỏng (ví dụ như Bolz [~12]) và phương trình thiết lập phân chia âm thanh [~1].

Đại số tuyến tính (Linear algebra): chương trìnhđại số tuyến tính là các khối tạo dựng cốt lõi cho một rất lớn các thuật toán số học, bao gồm giải pháp PDE đề cập ở trên. Ứng dụng chứa mô phỏng các hiệu ứng vật lý như: chất lỏng, nhiệt, và bức xạ, hiệu ứng quang học như lĩnh vực độ sâu [~23], và tương tự, theo đó chủ đề của đại số tuyến tính trên GPU đã nhận được nhiều sự chú ý. Một ví dụ điển hình là sản phẩm của Kr ¨uger và Westermann [~14] giải quyết một lớp rộng của các vấn đề đại số tuyến tính bằng cách tập trung vào biểu diễn ma trận và vectơ trong tính toán trên GPU dựa trên đồ họa (ví dụ như đóng gói các vector dày đặc (dense) và thưa thớt (sparse) vào các kết cấu, bộ đệm vector, v.v..). Một sản phẩm đáng chú ý khác là các phân tích về phép nhân ma trận dày đặc của Fatahalian và đồng nghiệp [~19] và giải pháp cho các hệ thống tuyến tính dày đặc của Gallapo và đồng nghiệp [~26], tác giả cho thấy có khả năng tốt hơn thậm chí các triển khai ATLAS tối ưu hoá mức cao. Ứng dụng của các tầng trực tiếp tính toán như CUDA và CTM vừa đơn giản hoá đồng thời cải thiện hiệu suất của đại số tuyến tính trên GPU. Ví dụ,

NVIDIA cung cấp uBLAS, một gói đại số tuyến tính dày đặc thực thi trong CUDA và sau đó là các quy ước BLAS phổ biến. Các thuật toán đại số tuyến tính thưa thớt có nhiều biến đổi và phức tạp hơn so với loại dày đặc đang là một lĩnh mở và hướng nghiên cứu tích cực, các nhà nghiên cứu mong có mã nguồn thưa thớt để kiểm chứng lợi ích tương tự hoặc lớn hơn từ tầng tính toán mới GPU.

Tổng kết

Một số chủ đề định kỳ nổi lên khắp các thuật toán và khám phá các ứng dụng trong tính toán GPU cho đến nay. Xem xét chủ đề này cho phép chúng tôi mô tả lại GPU làm tốt những gì. Ứng dụng tính toán GPU thành công có các đặc tính sau:

- ***Nhấn mạnh xử lý song song (Emphasize parallelism):*** GPU là về cơ bản máy song song và việc sử dụng hiệu quả nó phụ thuộc vào mức độ xử lý song song trong khối lượng công việc. Ví dụ, NVIDIA CUDA thích để chạy hàng ngàn luồng chạy tại một thời điểm, tối đa hóa cơ hội che dấu độ trễ bộ nhớ bằng cách sử dụng đa luồng. Nhấn mạnh xử lý song song đòi hỏi lựa chọn các thuật toán mà chia miền tính toán thành càng nhiều mảnh độc lập càng tốt. Để tối đa hóa số lượng luồng chạy đồng thời, GPU lập trình cũng nên tìm cách giảm thiểu việc sử dụng thread chia sẻ tài nguyên (như dùng các thanh ghi cục bộ và bộ nhớ dùng chung CUDA), và nên sự đồng bộ giữa các luồng là ít đi.
- ***Giảm thiểu sự phân kỳ SIMD (Minimize SIMD divergence)*:** GPU cung cấp một mô hình lập trình SPMD: nhiều luồng chạy cùng một chương trình tương tự, nhưng truy cập dữ liệu khác nhau và do đó có thể có sự khác nhau trong thực thi của chúng. Tuy nhiên, trong một số trường hợp đặc biệt, GPU thực thi chế độ SIMD cho các lô các luồng Nếu luồng trong một lô trệch ra, toàn bộ lô sẽ thực thi cùng các đường code cho đến khi các luồng hội tụ lại. Tính toán hiệu năng cao GPU đòi hỏi cơ cấu code sao cho giảm thiểu sự phân kỳ trong lô.
- ***Tăng tối đa cường độ số học (Maximize arithmetic intensity):*** Trong khung cảnh tính toán ngày nay, các tính toán thực tế là tương đối rẻ nhưng băng thông là quý giá. Điều này thật sự rất đúng với GPU nơi có nhiều sức mạnh dấu phảy động rất phong phú. Để tận dụng tối đa sức mạnh đó cần cấu trúc thuật toán để tối đa hóa cường độ số học, hoặc số lượng các tính toán trên số thực hiện trong mỗi thao tác với bộ nhớ. Truy cập dữ liệu mạch lạc bằng các luồng trợ giúp riêng biệt bởi vì các thao tác này có thể kết hợp để làm giảm tổng số

thao tác bộ nhớ. Sử dụng bộ nhớ dùng chung CUDA trên GPU NVIDIA cũng giúp giảm overfetch (do các luồng có thể giao tiếp) và cho phép các chiến lược "blocking" việc tính toán trên bộ nhớ của chip.

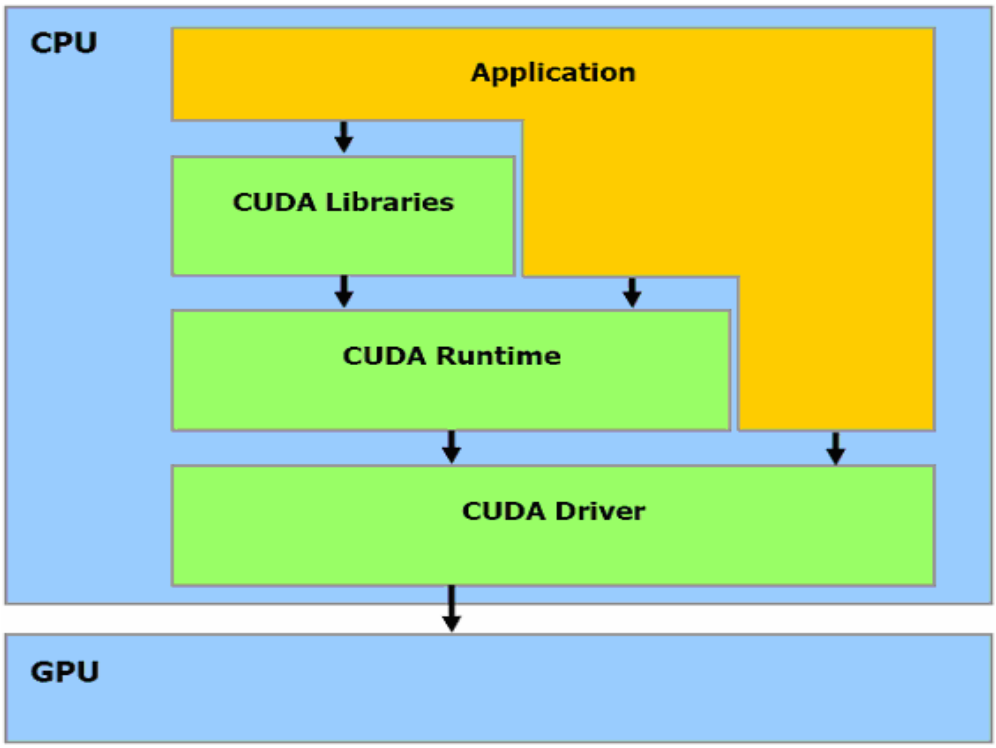
- ***Khai thác băng thông dòng (Exploit streaming bandwidth):*** Mặc dù có tầm quan trọng của cường độ số học, nó là cần lưu ý rằng GPU có băng thông rất ít (very high peak) trên bộ nhớ đi kèm, trên thứ tự của 10 × CPU - băng thông bộ nhớ thông dụng trên nền máy PC. Đây là lý do tại sao GPU có thể thực thi tốt hơn CPU ở các tác vụ như sắp xếp, trong đó có tỷ lệ tính toán/băng thông thấp. Để đạt được hiệu năng cao trên các ứng dụng như thế đòi hỏi các mẫu truy cập bộ nhớ dòng (streaming) trong đó các luồng đọc và ghi vào các khối lớn liền mạch (tối đa hóa băng thông cho mỗi giao dịch) nằm trong các khu vực riêng biệt của bộ nhớ (tránh các rủi ro dữ liệu).

Chương 2.
TÍNH TOÁN SONG SONG TRÊN GPU TRONG CUDA

2.1. Giới thiệu về môi trường phát triển CUDA

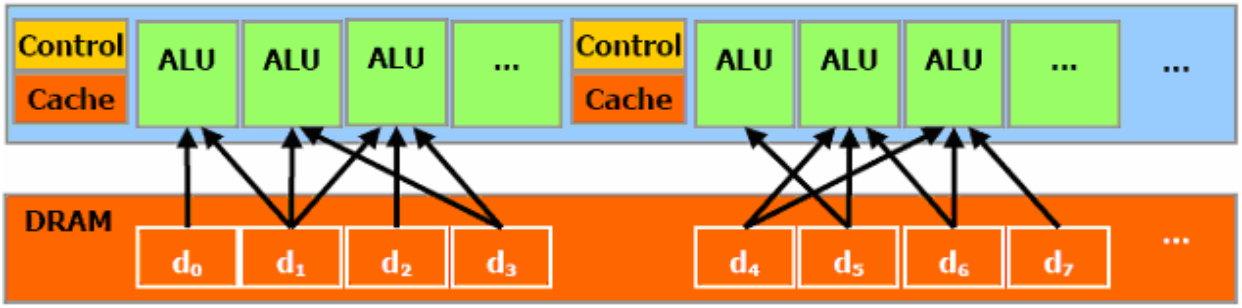
CUDA- viết tắt của Compute Unified Device Architecture, là kiến trúc mới bao gồm cả phần cứng và phần mềm để phát triển và quản lý việc tính toán trên GPU như một thiết bị tính toán song song mà không cần ánh xạ vào các hàm lập trình đồ họa. Kiến trúc này có trong giải pháp của GeForce 8 Series, Quadro FX 5600/4600, và Tesla của NVIDIA. Cơ chế đa nhiệm của hệ điều hành chịu trách nhiệm cho việc quản lý truy cập tới GPU bởi các ứng dụng CUDA và ứng dụng đồ họa chạy song song.

Bộ phần mềm CUDA bao gồm các lớp mô tả trong hình 13: dirver cho phần cứng, API lập trình, môi trường thực thi; và hai thư viện toán học mức cao hơn của các hàm thường dùng, CUFFT và CUBLAS. Phần cứng được thiết kế để hỗ trợ dirver hạng nhẹ và lớp môi trường thực thi, từ đó cho tốc độ cao.

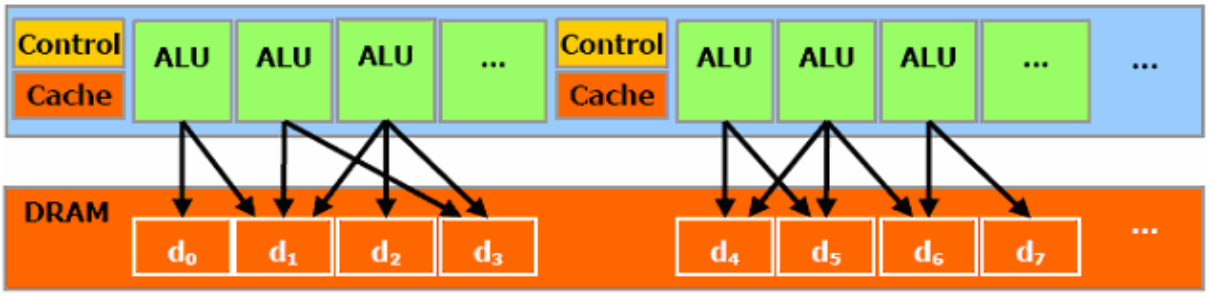


Hình 13: Kiến trúc bộ phần mềm CUDA

Thư viện lập trình CUDA bao gồm các hàm mở rộng của ngôn ngữ C. CUDA cung cấp cách đánh địa chỉ DRAM thường dùng như mô tả trong hình 14 cho việc lập trình linh hoạt hơn, bao gồm cả thao tác cấp phát và thu hồi bộ nhớ. Từ góc độ lập trình, điều đó tương ứng với khả năng đọc và ghi dữ liệu tại bất kỳ địa chỉ nào trong DRAM, giống như CPU.



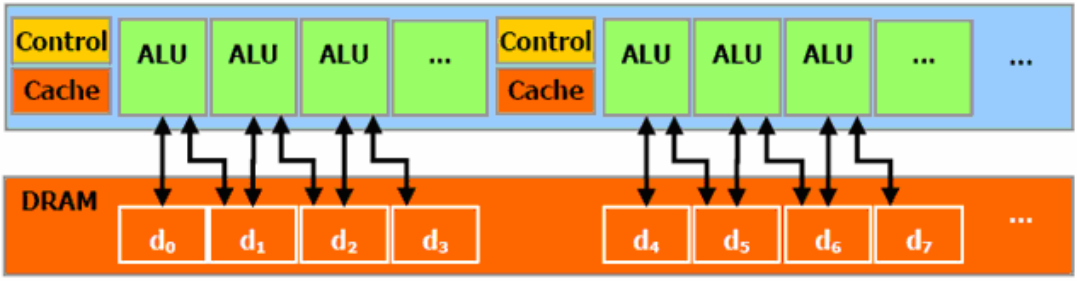
Thu hồi



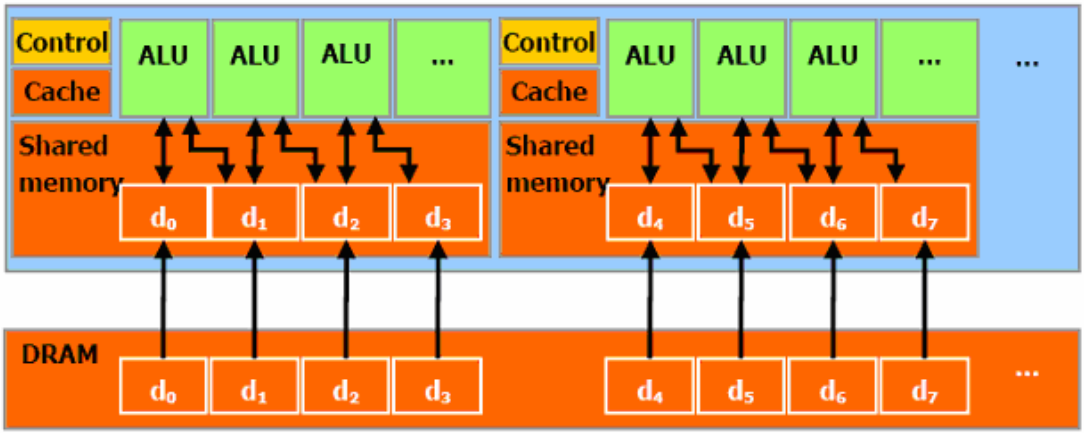
Cấp phát

Hình 14: Các thao tác thu hồi và cấp phát bộ nhớ

CUDA có đặc tính lưu dữ liệu đệm song song và và bộ nhớ chia sẽ trên chip với tốc độ đọc ghi rất cao, các luồng dùng bộ nhớ này để chia sẻ dữ liệu với nhau. Như mô tả trong hình 15, ứng dụng có thể đạt kết quả tốt với việc tối thiểu việc lấy/trả dữ liệu từ DRAM, từ đó trở giảm phụ thuộc băng thông truyền bộ nhớ DRAM.



Không có vùng nhớ dùng chung



Có vùng nhớ dùng chung

Hình 15: Vùng nhớ dùng chung mang dữ liệu gần ALU hơn

2.1 Môi trường lập trình và cơ chế hoạt động của chương trình CUDA

2.2.1 Môi trường lập trình

Để chương trình CUDA hoạt động được trong môi trường windows hoặc linux, cần phải có các thư viện hỗ trợ. Các thư viện này do NVIDIA cung cấp gồm có các phần sau: Trình điều khiển thiết bị đồ họa cho GPU của NIVIDA, bộ công cụ phát triển CUDA (gọi là CUDA Toolkit) và bộ CUDA SDK.

2.2.1 Cơ chế hoạt động một chương trình CUDA

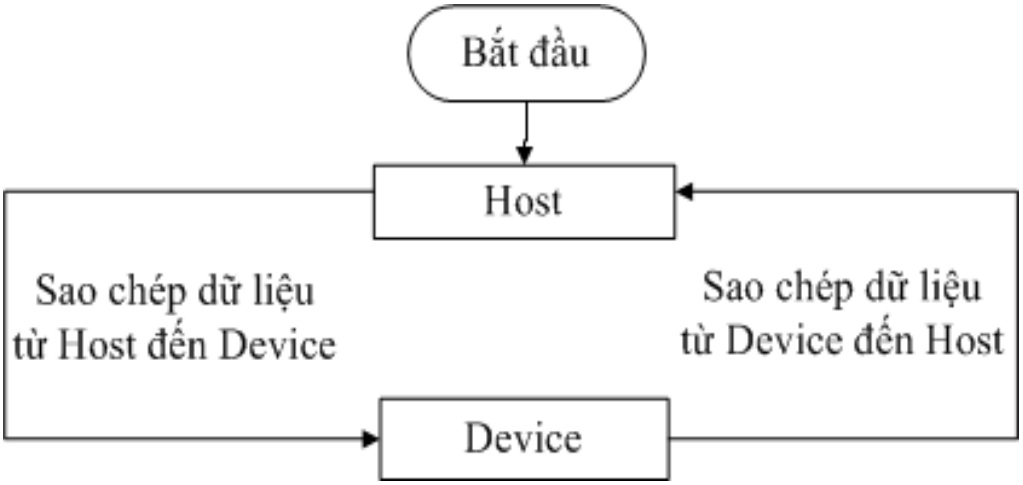
Sử dụng CUDA vì mong muốn chương trình chạy nhanh hơn nhờ khả năng xử lý song song. Vì thế tốt hơn hết cần loại bỏ các ảnh hưởng làm một chương trình chạy chậm đi. Một chương trình CUDA hoạt động theo mô hình SIMD (single instruction multiple data) do vậy ảnh hưởng chính đến tốc độ của chương trình là sự không thống nhất và tranh chấp vùng nhớ trong quá trình đọc và lưu dữ liệu. Điều này buộc trình biên dịch phải chọn giải pháp an toàn trong truy cập dữ liệu. Điều này biến một chương trình song song theo mô hình SIMD thành mô hình nối tiếp.

Kích thước của kiểu dữ liệu rất quan trọng trong việc truy cập dữ liệu một

cách thống nhất (coalescing) kích thước dữ liệu phải bằng 4, 8, 16 bytes. Ngoài ra nếu số lệnh tính toán lớn thì nên sao chép dữ liệu từ bộ nhớ chung (global memory) vào bộ nhớ chia sẻ (shared memory) để hạn chế việc truy cập thường xuyên vào bộ nhớ chung làm chậm chương trình (do việc truy cập vào bộ nhớ chung mất rất nhiều thời gian hơn truy cập vào bộ nhớ chia sẻ) [3].

Cấu trúc của một chương trình CUDA thường sử dụng hai hàm: Một hàm dành cho việc truy cập dữ liệu và hàm còn lại gọi là hàm kernel dùng cho việc xử lý dữ liệu.

Để hiểu cách hoạt động một chương trình CUDA (xem Hình 2.4), cần thống nhất một số các khái niệm sau:



Hình 16: Sơ đồ hoạt động truyền dữ liệu giữa Host và Device

- Host: Là những tác vụ và cấu trúc phần cứng, phần mềm được xử lý từ CPU.
- Device: Là những tác vụ và cấu trúc phần cứng, phần mềm được xử lý từ GPU.

Cách hoạt động được mô tả như sau:

- Dữ liệu cần tính toán luôn ở trên bộ nhớ của Host, vì vậy trước khi muốn thực hiện trên Device bước đầu tiên là sao chép dữ liệu cần tính toán từ bộ nhớ Host sang bộ nhớ Device.

- Sau đó Device sẽ thực hiện việc tính toán trên dữ liệu đó (gọi các hàm riêng của Device để tính toán).
- Sau khi tính toán xong, dữ liệu cần được sao chép lại từ bộ nhớ Device sang bộ nhớ Host.

Mô hình lập trình

Bộ đồng xử lý đa luồng mức cao

Trong lập trình CUDA, GPU được xem như là một thiết bị tính toán có khả năng thực hiện một số lượng rất lớn các luồng song song. GPU hoạt động như là một bộ đồng xử lý với CPU chính. Nói cách khác, dữ liệu song song, phần tính toán chuyên dụng của các ứng dụng chạy trên host được tách rời khỏi thiết bị.

Chính xác hơn, một phần của một ứng dụng được thực hiện nhiều lần, nhưng độc lập về mặt dữ liệu, có thể nhóm thành một chức năng được thực hiện trên thiết bị như nhiều luồng khác nhau. Để có điều đó, một chức năng được biên dịch thành các tập lệnh của thiết bị và tạo ra chương trình, gọi là nhân (kernel), được tải vào thiết bị.

Cả hai Host và Device (thiết bị) duy trì DRAM riêng của nó, được gọi là bộ nhớ host và bộ nhớ thiết bị. Có thể sao chép dữ liệu giữa DRAM của Host và Device thông qua API đã tối ưu hóa có sử dụng cơ chế truy cập bộ nhớ trực tiếp tốc độ cao (DMA) của thiết bị.

Gom lô các luồng

Lô các luồng thực hiện được nhân (kernel) tổ chức thành một lưới các khối luồng được miêu tả trong phần khối luồng và lưới các khối luồng dưới đây.

Khối luồng

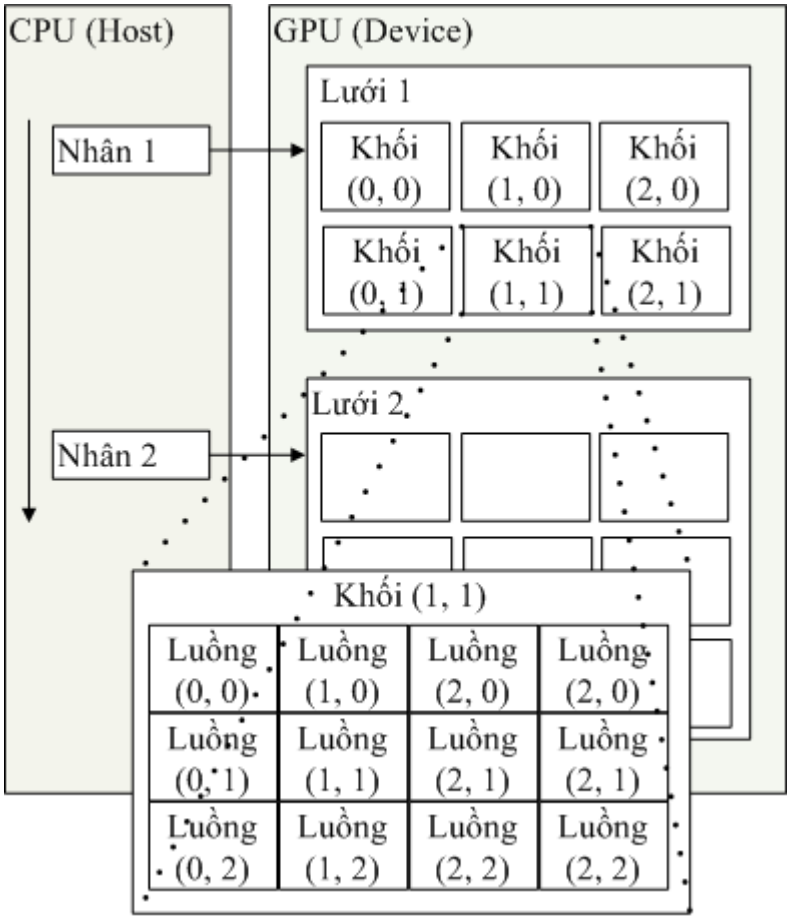
Một khối luồng là một tập các luồng, có thể đồng thời xử lý với nhau bằng cách dùng dữ liệu trong bộ nhớ dùng chung và thực thi đồng bộ để phối hợp truy cập

bộ nhớ.

Chính xác hơn, có thể xác định các điểm đồng bộ trong nhân, nơi các luồng trong khối sẽ dừng cho đến khi tất cả các luồng tới điểm đồng bộ.

Mỗi luồng được xác định bởi ID, đó là số hiệu của luồng trong khối. Để hỗ trợ việc định địa chỉ phức tạp dựa trên ID luồng, một ứng dụng cũng có thể chỉ định một khối như một mảng hai hoặc ba chiều có kích thước tùy ý và xác định từng luồng bằng cách sử dụng chỉ số hai hoặc ba thành phần để thay thế. Đối với các khối kích thước hai chiều (Dx, Dy), ID luồng của phần tử có chỉ số (x, y) là (x + y Dx) và cho một khối kích thước ba chiều (Dx, Dy, Dz), ID luồng của phần tử (x, y, z) là (x + yDx + z Dx Dy) [3].

Lưới các khối luồng (Grid of Thread Blocks)



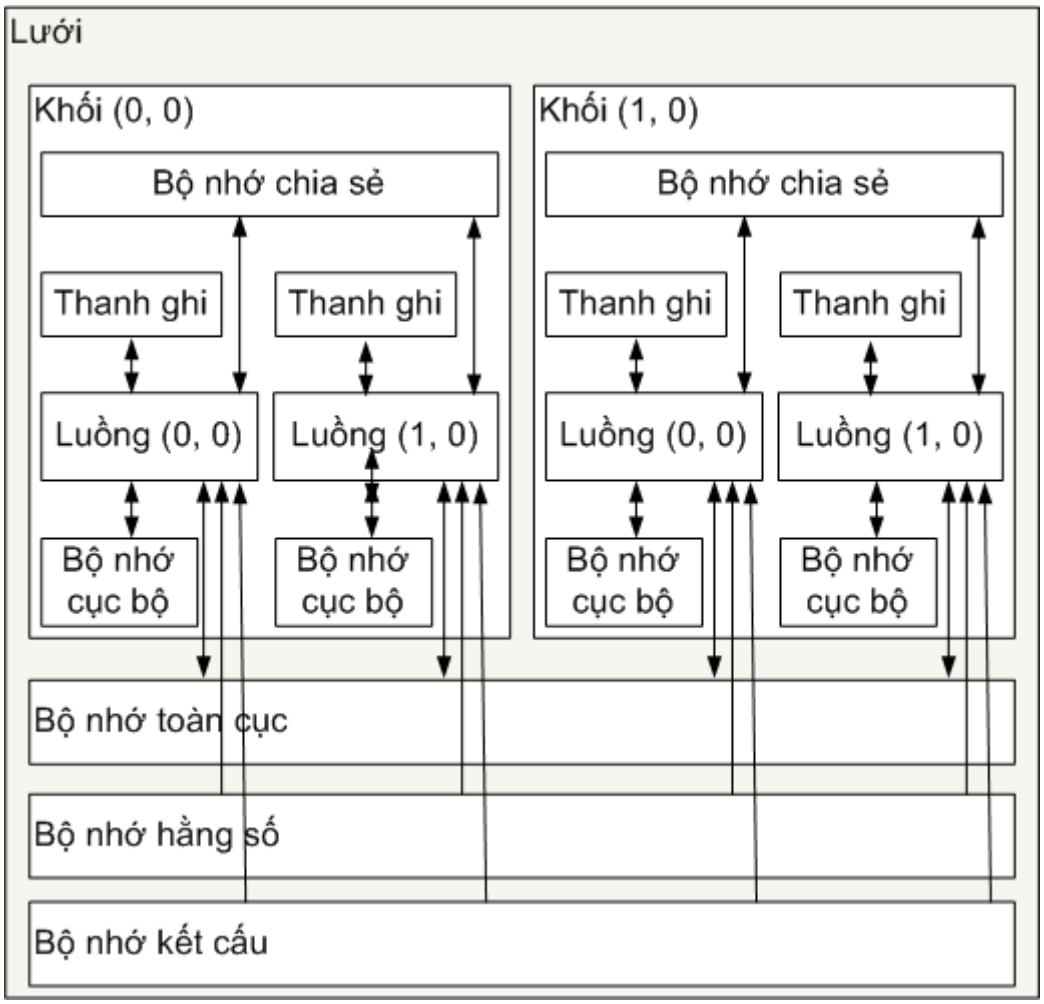
Hình 17: Khối luồng

Số lượng luồng tối đa trong một khối có giới hạn. Tuy nhiên, các khối cùng số chiều và kích thước thực thi trên cùng nhân có thể nhóm với nhau thành lưới các khối, do vậy tổng số luồng chạy trên một nhân là lớn hơn nhiều. Điều này xuất phát tại các chi phí

hợp tác giữa các luồng giảm, vì các luồng trong các lô khác nhau trong lưới không thể trao đổi và đồng bộ với nhau. Mô hình mô tả ở Hình 2.5, cho phép các nhân chạy hiệu quả mà không phải dịch lại trên các loại thiết bị khác nhau với khả năng chạy song song khác nhau: Một thiết bị có thể chạy trên tất cả khối của lưới một cách tuần tự nếu thiết bị đó có rất ít khả năng chạy song song hoặc chạy song song nếu nó có khả năng chạy song song nhiều hoặc kết hợp cả hai.

Mỗi khối được xác định bởi ID của nó, đó là số khối trong lưới. Để hỗ trợ việc định địa chỉ phức tạp dựa trên khối ID (block ID), một ứng dụng có thể xác định một lưới như một mảng hai chiều với kích thước cố định và định danh mỗi khối sử dụng chỉ mục hai thành phần. Với khối hai chiều kích thước (Dx, Dy), ID của khối (x,y) là (x + y Dx).

Mô hình bộ nhớ



Hình 18: Mô hình bộ nhớ trên GPU

Một luồng thực thi trên thiết bị chỉ truy cập vào DRAM của thiết bị và bộ nhớ trên bộ vi xử lý qua các không gian nhớ như mô tả trong Hình 2.6 :

- Đọc và ghi trên các thanh ghi (Registers) của mỗi luồng.
- Đọc và ghi bộ nhớ cục bộ (Local Memory) của mỗi luồng.
- Đọc và ghi bộ nhớ dùng chung (Shared Memory) của mỗi khối.
- Đọc và ghi bộ nhớ toàn cục (Global Memory) của mỗi lưới.
- Chỉ đọc bộ nhớ hằng số (Constant Memory) của mỗi lưới.
- Chỉ đọc bộ nhớ kết cấu (Texture Memory) của mỗi lưới.

Các vùng nhớ toàn cục, hằng số và kết cấu có thể đọc hoặc ghi bởi Host và liên

tục giữa các lần thực thi nhân bởi cùng một ứng dụng.

Các vùng nhớ toàn cục, hằng số và kết cấu được tối ưu hóa cho các cách sử dụng bộ nhớ khác nhau. Vùng nhớ kết cấu cũng đưa ra các cơ chế đánh địa chỉ khác, cũng như lọc dữ liệu cho một số loại dữ liệu đặc biệt.

2.3.Lập trình ứng dụng với CUDA

2.3.1. CUDA là mở rộng của ngôn ngữ lập trình C

Mục tiêu của giao diện lập trình CUDA là cung cấp cách tiếp cận khá đơn giản cho những người sử dụng quen với ngôn ngữ lập trình C, có thể dễ dàng viết chương trình cho việc xử lý bằng các thiết bị. Lập trình CUDA gồm có:

- Một thiết lập tối thiểu của các thành phần mở rộng cho ngôn ngữ lập trình C được miêu tả trong phần 2.1.6.2. , cho phép người lập trình nhắm tới cách phân chia mã nguồn chương trình cho việc xử lý trên thiết bị.
- Thư viện chạy được chia thành:
  - ✓ Thành phần chính (host componet): Chạy trên Host và cung cấp các chức năng cho việc điều khiển và truy nhập một hoặc nhiều thiết bị khác từ Host.
  - ✓ Các thiết bị thành phần (device componet): Được chạy trên các thiết bị và cung cấp các hàm riêng của thiết bị đó.
  - ✓ Một thành phần chung (commom componet): Cung cấp xây dựng trong kiểu vector và là một tập con thư viện chuẩn của C. Thành phần chung hỗ trợ cho cả Host và các thiết bị thành phần.

Cần nhấn mạnh rằng chỉ có hàm từ thư viện chuẩn của C là được hỗ trợ cho việc chạy trên các thiết bị có các chức năng được cung cấp bởi thành phần chạy chung [3].

2.3.2. Những mở rộng của CUDA so với ngôn ngữ lập trình C

Ngôn ngữ lập trình CUDA là mở rộng của ngôn ngữ lập trình C ở bốn khía cạnh

- Từ khóa phạm vi kiểu hàm cho phép xác định liệu một hàm thực hiện trên host hay trên thiết bị và liệu nó có thể được triệu gọi từ host hoặc từ thiết bị.
- Từ khóa phạm vi kiểu biến cho phép đặc tả vị trí bộ nhớ trên thiết bị của một biến.
- Bốn biến build-in để xác định chiều của lưới và khối, chỉ số khối và luồng.
- Một chỉ thị mới để xác định cách nhân (kernel) được thực hiện trên thiết bị từ phía host.

Với mỗi tập tin nguồn chứa các phần mở rộng trên phải được biên dịch với CUDA bằng trình biên dịch NVCC, được miêu tả ngắn gọn trong mục 2.1.6.7. Những miêu tả chi tiết của NVCC có thể được tìm thấy trong các tài liệu khác [3].

Mỗi phần mở rộng đi kèm với một số hạn chế được mô tả trong phần dưới, NVCC sẽ đưa ra lỗi hoặc thông điệp cảnh báo một số xung đột của các phần hạn chế trên, nhưng một số xung đột có thể không được nhận ra.

Từ khóa phạm vi kiểu hàm

Dùng để khai báo một hàm có phạm vi hoạt động ở trên Host hay trên Device, và được gọi từ Host hay từ Device:

- Từ khóa device_:

✓ Khai báo__device__định nghĩa một hàm chỉ xử lý trên thiết bị (Device)..

✓ Chỉ được gọi từ thiết bị.

✓ Ví dụ : device_void HamXuLyTaiDevice(parameter,…) {…}

- Từ khóa global_:

✓ Khai báo global_định nghĩa một hàm như là một hạt nhân (kernel), xử lý trên thiết bị.

✓ Chỉ có thể triệu gọi được từ Host.

✓ Ví dụ : global_void HamKernelXuLy(parameter,…){…}

- Từ khóa host_:

✓ Khai báo__host__là định nghĩa một hàm xử lý trên Host.

✓ Chỉ có thể triệu gọi được từ Host.

Các hạn chế:

- Các hàm của device____là hàm đóng (inlined).
- Các hàm của device_____và global___không hỗ trợ sự đệ quy.
- Các hàm của device_____và global_không thể khai báo các biến static trong thân hàm.
- Các hàm của device_____và global___không thể có số biến của thay đổi.
- global_____và host_____không thể sử dụng đồng thời. global_____phải có kiểu trả về là kiểu void.
- Lời gọi hàm global_____phải chỉ rõ cấu hình thực hiện nó (xem mục 2.1.6.5).
- Gọi tới một hàm __global_____là không đồng bộ, có nghĩa là hàm global_____trả về trước khi thiết bị hoàn thành xong xử lý [3].

Từ khóa phạm vi kiểu biến

Cho phép đặc tả vị trí bộ nhớ trên thiết bị của một biến:

- __device__:

✓ Tồn tại trong không gian bộ nhớ toàn cục (có bộ nhớ lớn, độ trễ cao).

✓ Được cấp phát với cudaMalloc.

✓ Có vòng đời (lifetime) của một ứng dụng.

✓ Truy nhập được từ tất cả các luồng bên trong lưới

- __shared__:

✓ Tồn tại trong không gian bộ nhớ chia sẻ của một luồng (bộ nhớ

nhỏ,độ trễ thấp).

- ✓ Được cấp phát khi thực hiện việc cấu hình, hay khi biên dịch chương trình.
- ✓ Có vòng đời của một khối.
- ✓ Chỉ có thể truy cập từ tất cả các luồng bên trong một khối (các luồng thuộc khối khác không thể truy cập).

Thực hiện cấu hình

Bất kỳ lời gọi tới hàm toàn cục (global) phải xác định cấu hình thực hiện cho lời gọi

Cấu hình xử lý xác định kích thước lưới và khối mà sẽ được sử dụng thực hiện chức năng trên thiết bị. Nó được xác định bằng cách chèn một biểu thức mẫu dạng <<< Dg, Db, Ns >>> giữa tên hàm và danh sách tham số được để trong ngoặc đơn, ở đây:

- Dg là kiểu dim3 và xác định mục đích và kích thước của lưới, sao cho Dg.x*Dg.y bằng với số khối được đưa ra.
- Db là kiểu dim3 và xác định mục đích kích thước của mỗi khối, sao cho Db.x*Db.y*Db.z bằng số lượng các luồng trên khối.
- Ns là một kiểu size_t và xác định số byte trong bộ nhớ chia sẻ, nó cho phép khai báo động trên mỗi khối cho lời gọi ngoài việc cấp phát bộ nhớ tĩnh. Việc cấp phát bộ nhớ động sử dụng bởi bất kỳ biến khai báo như là một mảng mở rộng, Ns là một đối số tùy chọn mặc định là 0 [3].

Một ví dụ cho việc khai báo hàm:

*__global__ void Func(int *parameter);*

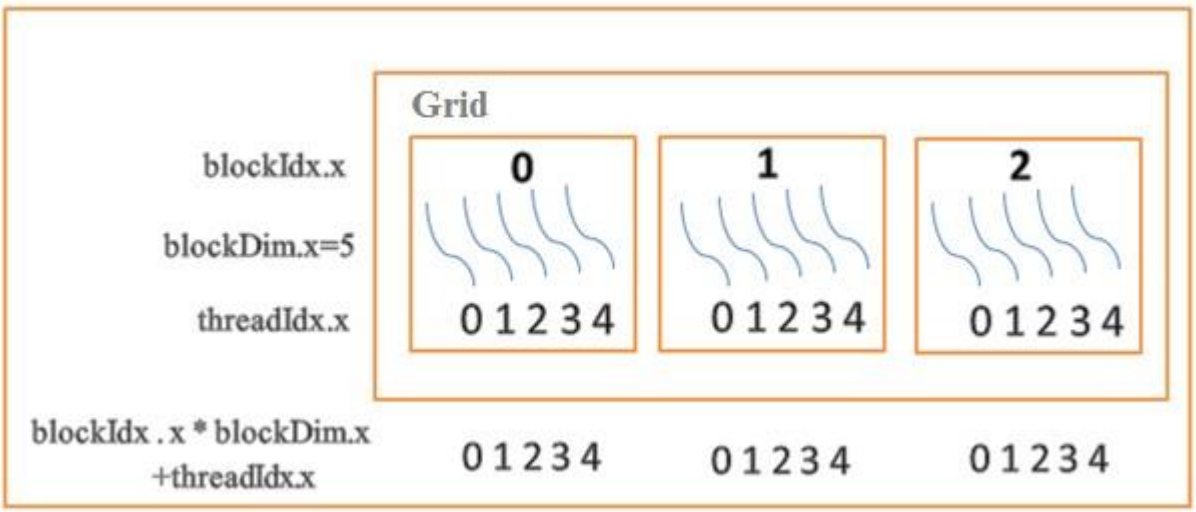
Phải gọi hàm từ Host giống như sau :

Func<<<Dg, Db, Ns>>>(parameter);

Các biến Built-in

Biến build-in để xác định chiều của lưới và khối, chỉ số khối và luồng :

- gridDim là biến kiểu dim3 và chứa các kích thước của lưới.
- blockIdx là biến thuộc kiểu unit3 và chứa các chỉ số khối trong lưới.
- blockDim là biến kiểu dim3 và chứa kích thước của một khối.
- threadIdx là biến kiểu unit3 và chứa các chỉ số luồng trong khối.



Hình 19: Chiều của lưới và khối với chỉ số khối và luồng

2.3.3. Biên dịch với NVCC

NVCC là một trình điều khiển trình biên dịch bằng việc đơn giản hóa quá trình biên dịch mã CUDA. NVCC cung cấp các tùy chọn dòng lệnh đơn giản và quen thuộc thực hiện chúng bằng cách gọi tập hợp của các công cụ thực hiện các công đoạn biên dịch khác nhau.

NVCC bao gồm luồng công việc cơ bản trong việc tách mã thiết bị từ mã Host và biên dịch mã thiết bị sang dạng nhị phân hoặc các đối tượng cubin. Các mã Host sinh ra là đầu ra có thể là mã C để được biên dịch bằng cách sử dụng một công cụ khác hoặc mã đối tượng trực tiếp bởi việc triệu gọi trình biên dịch Host trong giai đoạn biên dịch trước đó.

Ứng dụng có thể bỏ qua các mã Host sinh ra, tải đối tượng cubin vào thiết bị và khởi động mã thiết bị sử dụng trình điều khiểu API của CUDA hoặc

liên kết tới mã Host sinh ra, trong đó bao gồm các đối tượng cubin được xem như mảng dữ liệu khởi tạo toàn cục và chứa một bản dịch các cú pháp thực thi cấu hình thành mã cần thiết khởi động trong thời gian chạy CUDA để nạp và khởi động mỗi lần biên dịch hạt nhân.

Frond end của trình biên dịch xử lý các tập tin nguồn CUDA theo cú pháp quy định C++. Tuy nhiên, chỉ có các tập con C của C++ được hỗ trợ. Điều này có nghĩa là những đặc tính đặc trưng của C++ như các lớp (classes), sự kế thừa hoặc việc khai báo các biến trong khối cơ bản là không được hỗ trợ. Như một hệ quả của việc sử dụng cú pháp C++, con trỏ void (ví dụ như trả lại malloc()) không thể được gán tới những con trỏ non-void mà không có ép kiểu [3].

2.4. Ví dụ tính toán song song bằng CUDA

- **Cộng hai số nguyên:** Ví dụ này cho thấy cách thức viết một hàm chạy trên thiết bị (device) và được triệu gọi ra sao:

Cộng hai số nguyên a và b kết quả được đưa vào số nguyên kết quả c. Chú ý là dùng kiểu con trỏ cho các biến.

Code tuần tự

```
void CongHaiSoNguyen(int *a,int *b, int *c)
{
*c=*a+*b;
}
void main()
{
int *a,*b,*c;
CongHaiSoNguyen(a,b,c);
}
```

Code CUDA

```
__global__void KernelCongHaiSoNguyen(int *a,int *b,int *c)
{
*c=*a+*b;
}
void main()
{
int *a,*b,*c;
*a=1; *b=5;
int *deva,*devb,*devc;
cudaMalloc((void**)&deva,
sizeof(int) );
cudaMalloc((void**)&devb,
sizeof(int) );
cudaMalloc((void**)&devc,
sizeof(int) );
cudaMemcpy(deva, a, sizeof(int),
cudaMemcpyHostToDevice); cudaMemcpy(devb, b,
sizeof(int), cudaMemcpyHostToDevice);
KernelCongHaiSoNguyen<<<1,1>>>(deva, devb, devc);
cudaMemcpy(c, devc, sizeof(int),
cudaMemcpyDeviceToHost);
}
```

Trên đây ta thấy gọi hàm KernelCongHaiSoNguyen khá đặc biệt, ta chỉ cấp 1 luồng để xử lý việc cộng 2 số a và b và kết quả lưu vào c. Ta chưa thấy được việc chạy song song trên thiết bị. Ví dụ này cho ta thấy cách viết một hàm thiết bị và gọi nó như thế nào. Ví dụ cộng hai mảng số nguyên phía sau đây sẽ thực hiện song song trên thiết bị.

❖ **Cộng hai mảng số nguyên**: Ví dụ này cho thấy được việc song song hóa trên thiết bị (device).Cộng hai mảng số nguyên a[n] và b[n], kết quả được lưu vào mảng c[n]. Làm cách nào để chúng ta chạy song song trên thiết bị?

- Cách giải quyết thứ nhất là thay cấu hình gọi hàm <<<1,1>>> bằng <<<n,1>>> có nghĩa là cấp n block (mỗi block chỉ có một thread) để thực hiện việc cộng từng phần tử của 2 mảng a và b lưu vào mảng c. Như vậy code song song của chúng ta sẽ là:

```
__global___void KernelAdd(int *a, int *b, int *c)
{
c[blockIdx.x]= a[blockIdx.x] + b[blockIdx.x];
}
```

Trên thiết bị, mỗi block sẽ thực hiện song song:

- Block 0 thực hiện: c[0]= a[0] + b[0];
- Block 1 thực hiện: c[1]= a[1] + b[1];
- Block 2 thực hiện: c[2]= a[2] + b[2];
- Block 3 thực hiện: c[3]= a[3] + b[3];
- Block n-1 thực hiện: c[n-1]= a[n-1] + b[n-1];

- Cách giải quyết thứ hai là thay vì ta dùng block để song song , ta có thể dùng luồng (threads) để song song, cấu hình gọi hàm sẽ là <<<1,n>>> (một block với nhiều luồng):

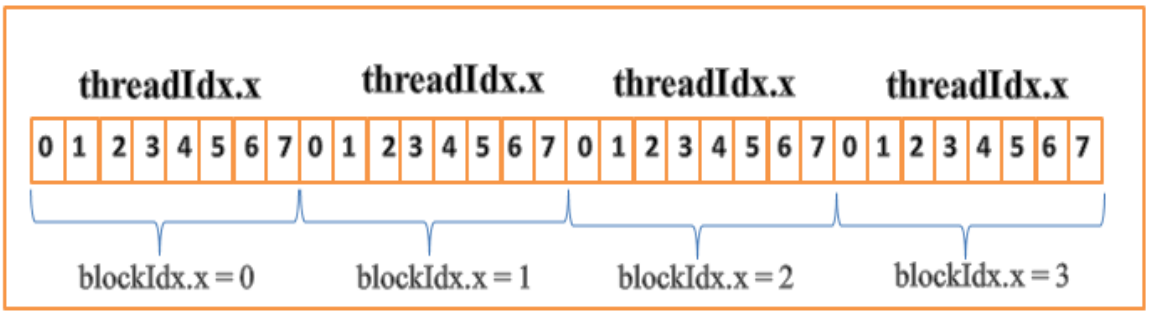
Code song song của chúng ta sẽ là :

```
__global___void KernelAdd(int *a, int *b, int *c)
{
c[threadIdx.x]= a[threadIdx.x] + b[threadIdx.x];
}
```

Như vậy chúng ta đã thấy việc song song dùng :

- Nhiều block với một thread cho mỗi block.
- Một block với nhiều luồng.

- Cách giải quyết thứ ba là kết hợp cả block và thread : Không còn đơn giản như việc dùng blockIdx.x và threadIdx.x nữa, ta hãy xem cách đánh chỉ số của một mảng với một phần tử của mảng cho mỗi thread (8thread/block) trong Hình 2.8.



Hình 20: Phương pháp đánh chỉ số luồng.

- Với M thread/block, một chỉ số duy nhất cho mỗi thread sẽ là:

*int index = threadIdx.x + blockIdx.x * M;*

- Dùng biến built-in blockDim.x (tương ứng với số lượng thread trong một block) thay cho M ta được :

*int index = threadIdx.x + blockIdx.x * blockDim.x;*

- Vậy code song song của chúng ta sẽ là:

```
__global___void KernelAdd(int *a, int *b, int *c)
{
int index=  threadIdx.x + blockIdx.x * blockDim.x;

c[index]= a[index] +  b[index];
}
void main()
{
……;
KernelAdd<<<n/ThreadperBlock, ThreadperBlock >>>(deva,devb,devc);
……;
```

}

2.5. Ứng dụng của CUDA trong lĩnh vực công nghệ

CUDA cho ngành công nghiệp trò chơi

Một trong những ứng dụng về sự thành công của công nghệ CUDA là trong ngành công nghiệp giải trí với lĩnh vực trò chơi. Hình ảnh trong trò chơi như thật là nhờ bộ công cụ PhysX SDK và khung hình làm việc có khả năng mở rộng động trên nhiều nền tảng có liên quan với nó gọi là APEX, cả hai đều do NVIDIA cung cấp. Đây là những công cụ đầy sức mạnh trong bộ các engine AXE, giành riêng cho vật lý trong trò chơi, hay nói cách khác, được thiết kế để xử lý các di chuyển phát sinh động và tương tác của các đối tượng trong từng cảnh của trò chơi.

Vật lý trong trò chơi khiển cho tính năng đồ họa của một trò chơi trở nên sống động và chẳng bao lâu nữa chuyện hiển thị cảnh như phim với thời gian thực trong trò chơi sẽ trở thành hiện thực với sự hỗ trợ của PhysX và APEX.

Bộ công cụ PhysX SDK hiện nay đã có trên hầu hết các nền tảng máy trò chơi thông dụng, từ XBOX 360 sang PlayStation 3 sang Wii rồi đến NVIDIA GPU, với hơn 150 tựa trò chơi mới trên thị trường.

CUDA cho các ứng dụng video số

Có thể nói CUDA rất thành công trong với xử lý video. Rất nhiều ứng dụng video số hóa dựa trên CUDA, chẳng hạn như cải tiến chất lượng hình ảnh video với phần mềm vReveal của MotionDSP, mở rộng độ phân giải DVD với SimHD của ArcSoft. Một vài ví dụ trong số các ứng dụng hay này là vReveal đến từ MotionDSP là phần mềm cải thiện chất lượng hình ảnh như: Làm rõ nét, điều chỉnh độ tương phản và ổn định hóa (xóa run) các video. vReveal thường cần đến các hệ thống CPU đa bộ vi xử lý đắt tiền để hiển thị video một cách chậm chạp. Nhưng giờ đây với CUDA GPU đã có thể thực hiện nó theo thời gian thực đến khoảng năm lần nhanh hơn so với CPU. MotionDSP còn cung cấp một cung cấp một phiên bản cao cấp hơn, gọi là Ikenna, cho lĩnh vực tình báo và điều tra pháp luật.

Trong thời gian gần đây, sự phát triển của những thiết bị di động có khả năng thu dữ liệu hình ảnh, video với chất lượng cao đã khiến con người thỏa mái hơn trong

việc thưởng thức âm nhạc, phim, hình chụp cá nhân ở mọi lúc, mọi nơi. Tuy nhiên, phong cách giải trí mới trong cuộc sống hàng ngày sẽ không thể có được nếu không có những nỗ lực của riêng mình. Chẳng hạn như phải tốn nhiều thời gian để chuyển đổi nhạc, phim trong máy để bàn của mình sang chiếc iPod Touch yêu quý và ngược lại. Quá trình chuyển đổi đó hoàn toàn không đơn giản, nếu như chỉ là một người sử dụng máy tính bình thường. Trong trường hợp đó, phần mềm Badaboom của Elemental Technologies có thể giúp ích rất nhiều. Đó là bộ chuyển đổi media nhanh nhất và được thiết kế đầu tiên trên thế giới để chạy tối ưu với GPU và CUDA của NVIDIA. Khi so sánh bộ chuyển định dạng cuariTunes, Badaboom có thể nhanh hơn đến 20 lần hoặc tối thiểu cũng nhanh hơn hai đến ba lần ngay khi sử dụng CPU nhanh nhất và đắt tiền Core i7 của Intel.

Chương 3: TĂNG TỐC ĐỘ TÍNH TOÁN MỘT SỐ BÀI TOÁN SỬ DỤNG GPU

3.1. Giới thiệu một số bài toán cơ bản

Trong ngành công nghiệp giải trí hiện này thì một nhu cầu cấp thiết đó là các công nghệ phải cung cấp các hình ảnh và âm thanh chất lượng cao cùng với kich thước lớn.Vì thế nghiên cứu song song hóa các thuật toán xử lý tín hiệu số hoặc xử ảnh là một xu thế tất yếu và đã được nhiều nhà nghiên cứu cũng như các công tư công nghệ thực hiện.Chính vì thế trong luận văn này tôi lựa chọn hai bài toán tiêu biểu để phát triển chạy song song trên nên tảng GPU. Việc chọn hai bài toán này ngoài mục tiêu ứng ụng GPU để tăng tốc độ tính toán, luận văn còn muốn chỉ ra phạm vi các bài toán có thể song song hóa được trên GPU, và phướng pháp đơn giản để chuyển các tính toán trên CPU xuống GPU sử dụng Matlab.

3.2. Biến đổi FFT trên GPU

Các phép phân tích và biến đổi Fourier là một trong các bước tiền xử lý quan trọng trong xử lý tín hiệu số. Do tín hiệu thực trong cuộc sống là tín hiệu tương tự liên tục theo thời gian nên không tương thích khó áp dụng các thuật toán trên máy tính. Vì vậy trước khi có thể áp dụng các phương pháp số như lọc băng tần, khử nhiễu, tăng cường, … thì tin hiệu luôn được biến đổi sang khong gian khác gọi là miền tần số. Vì vậy phân tích và biến đổi Fourier gần như xuất hiện trong mọi hệ thống xử lý tín hiệu số.

3.2.1 Phân tích Fourier

Theo Fourier thì mội tín hiệu đều có thể biểu diễn lại được bằng 1 chuỗi Fourier có dạng như công thức sau:

$$f(t) = a_0 + \sum_{n=1}^{\infty}\left[a_n \cos\frac{2n\pi}{T}t + b_n \sin\frac{2n\pi}{T}t \right] \quad (3.1)$$

Trong đó:

- a_0 là thành phần không đảo chiều (DC)
- a_n và b_n là biên độ của thành tần số thứ n.

Từ công thức trên cho thấy phép phân tích Fourier sử dụng các hàm cosine và sine nên số liệu được sử dụng trong quá trình tính toán sẽ là số thực dấu phẩy

động.Các phép toán trên loại dữ liệu này tối khá nhiều hiệu năng của CPU. Cũng từ công thức trên cho thấy thời để phân tích một tín hiệu Fourier se là:

$$Time=T*N*\beta \qquad (3.2)$$

Trong đó: T là chiều dài mẫu, N là số thành phần tần số, và βlà thời gian để thực hiện phép tính sine/cosine.

3.1.1. Phép biến đổi Fourier

Ý nghĩa của phép biến đổi Fourier (FFT) rời rạc là biến đổi tín hiệu ở miền thời gian sang miền tần số theo công thức sau:

$$X_k = \sum_{j=0}^{N-1} x_j \, e^{-2\pi ijk/N} \qquad (3.3)$$

Trong đó:

- k là thành phần tuần hoàn thứ k, k=1..K
- N là kích thước của khung dữ liệu tính toán.
- $e^{i\phi} = \cos\phi + i\sin\phi,$

Từ công thức này cho thấy phép biến đổi FFT vẫn thực hiện các phép toán trên dữ liệu dấu chấm động.

3.1.2. Phân tích và biến đổi FFT trên GPU

Từ các phân tích ở mục 1.2 và 1.3 cho thấy phép phân tích và biến đổi FFT đều thực hiện các phép toán trên số thực dấu chấm động.Việc này sẽ tiêu tốn hiệu năng cho CPU.Đối với các tín hiệu có kích thước lớn (T rất lớn trong công thức (3.1), N rất lớn trong công thức (3.3)) thì chi phí thời gian để tính toán sẽ là rất lớn. Tuy nhiên giá trị *f(t)* trong (3.1) và *X(k)* trong công thức 3.3 hoàn toàn độc lập với t ± 1, k ± 1, điều này dẫn đến là f(t), f(t+1), f(t-1) hoặc X(k), X(k+1), X(k-1) có thể được thực hiện song song. Nếu một GPU có L cores thì ta có thể tính L phần tử f(t) hoặc X(k) cùng một thời điểm. Xuất pháp từ lý do này luận văn tiếp hành xây dựng chương trình tính toán biến đổi fourier trên GPU theo nguyên tắc sau đây:

1. Khai báo một mảng gồm T (số mẫu trong công thức (3.1)) hoặc K (số thành phần tuần hoàn với tần số k) phần tử. Mỗi phần tử của mảng này sẽ được sử dụng để lưu trữ một giá trị f(t) hoặc X(k). Nghĩa là các giá trị f(t) và X(K) sẽ không được tính toán một cách tuần tự, nếu số cores của GPU là L thì bằng cách khai báo các phần tử như trên thì ta có thể tính toán song song một lúc L giá trị trong mảng (nghĩa là có thể tính toán được L giá trị f(t) hoặc X(K) một lúc)

2. Copy toàn bộ mảng đã khai bao vào bộ nhớ RAM của GPU
3. Sử dựng thư viện hàm CUDA thực hiện các phép tính (*,/,sin,cos) trong công thức.
4. Copy giá trị ngược lại về bộ nhớ RAM máy tính = > thu được kết quả

3.1.3. Chương trình thử nghiệm

Chương trình sau đây gồm các bước

1. Tạo ra 1 tín hiệu với T mẫu (T=numSamples) là tổ hợp của n thành phần tuần hoàn (n=freq) có tần số là i*20 với i=1..n
2. Thực hiện phép biến đổi FFT trên tín hiệu ở bước 1

Mã nguồn trên Matlab

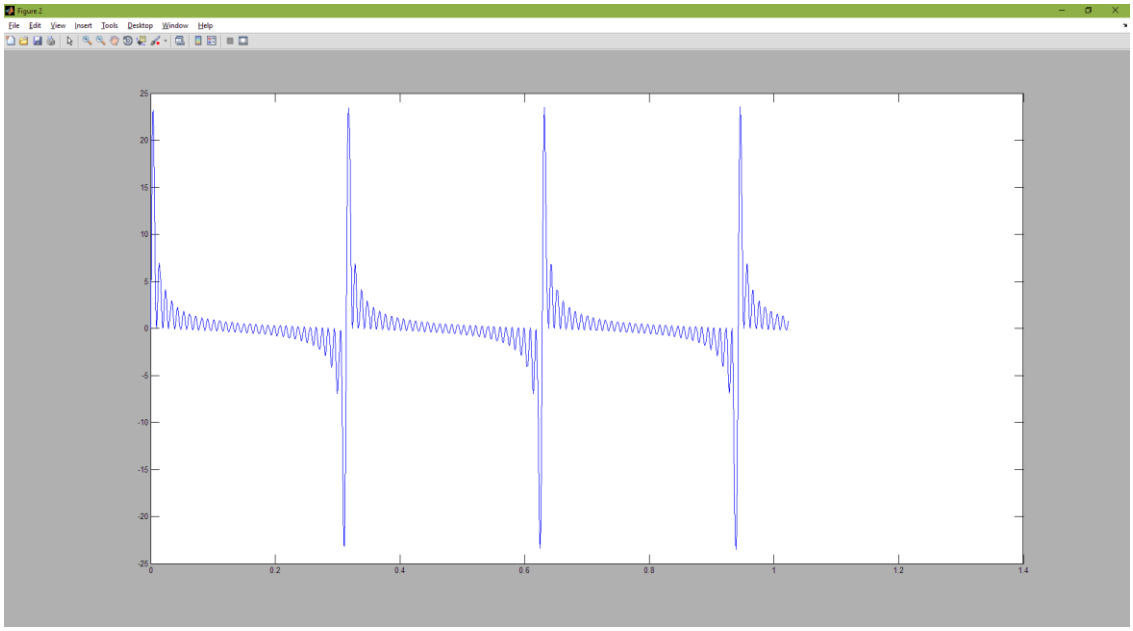
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clearall
sampleFreq = 1000;
sampleTime = 1/sampleFreq;
numSamples = 2^23;
timeVec    = gpuArray( (0:numSamples-1) * sampleTime );
freq=128
tic
signal=0;
for i=1:freq
signal=signal + sin(i*20 .* timeVec);
end
transformedSignal = fft( signal );
powerSpectrum = transformedSignal .* conj(transformedSignal) ./ numSamples;
frequencyVector = sampleFreq/2 * linspace( 0, 1, numSamples/2 + 1 );
plot( frequencyVector, real(powerSpectrum(1:numSamples/2+1)) );
toc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Chương trình sử dụng CPU:

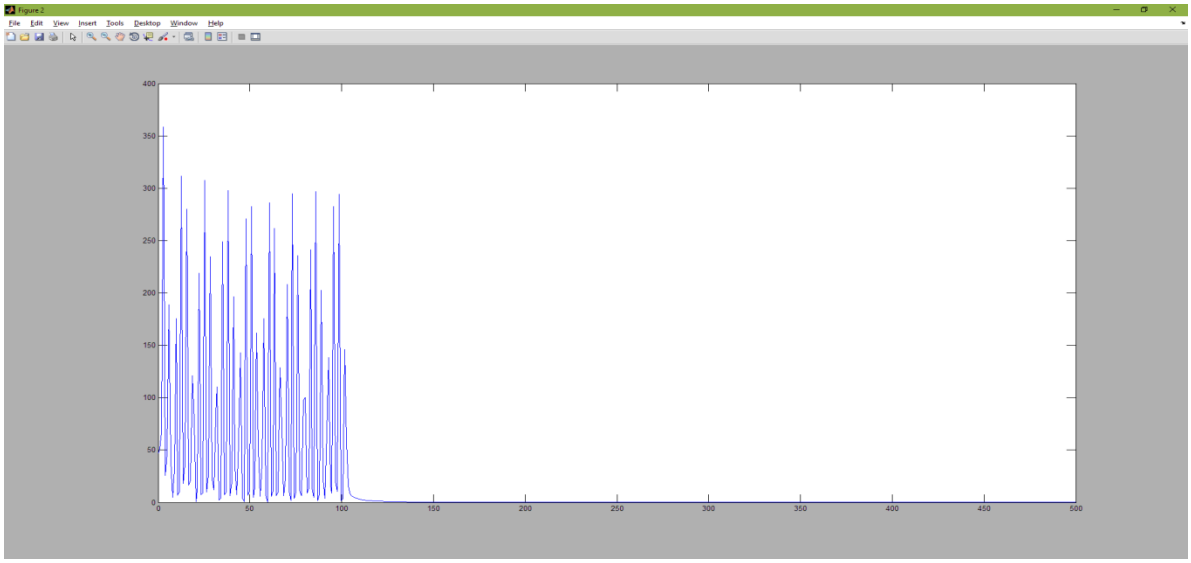
```
clear all
sampleFreq = 1000;
sampleTime = 1/sampleFreq;
numSamples = 2^10;
timeVec    = (0:numSamples-1) * sampleTime ;
freq1      = 2 * pi * 50;
freq2      = 2 * pi * 120;
tic
signal=0;
for i=1:32
signal=signal + sin(i*20 .* timeVec);
end
figure;plot(timeVec,signal);
transformedSignal = fft( signal );
transformedSignal = fft( signal );
powerSpectrum = transformedSignal .* conj(transformedSignal) ./ numSamples;
frequencyVector = sampleFreq/2 * linspace( 0, 1, numSamples/2 + 1 );
figure;plot( frequencyVector, real(powerSpectrum(1:numSamples/2+1)) );
toc
```

3.1.4. Kết quả thử nghiệm

3.1.4.1. Dữ liệu



Mẫu tín hiệu vào với 32 thành phần songhài



Phổ tín hiệu sau khi áp dụng phép biến đổi FFT

3.1.4.2. Đánh giá hiệu suất tính toán

Tần số lấy mẫu (Hz)	Số mẫu	số thành phần tần số	thời gian tính toán CPU Core-i5 (4 cores), đơn vị giây	thời gian tính toán GPU (Quadro 600, 99 cores), đơn vị giây
1000	2^23	2	0.61	0.34
		4	0.72	0.62
		8	0.97	0.52
		16	3.15	0.66
		32	9.3	0.94
		64	22.1	2.51
		128	47.2	3.7

3.2. Phát hiện biên ảnh

3.2.1. Phương pháp phát hiện biên

Xử lý ảnh là một lĩnh vực đã được nghiên cứu từ rất lâu, với nhiều kỹ thuật thuật đã được áp dụng trong thực tế.Ngày này với sự phát triển vượt bậc về mặt công nghệ, các camera cho độ nét và phân giải cao hơn rất nhiều.Điều đó đồng nghĩa với việc các bộ xử lý sẽ phải tiêu tốn nhiều hiệu suất hơn để xử lý các bức ảnh có kích thước lớn.Việc này có thể dẫn tình trạng làm giảm tốc độ xử lý, có thể khiến cho các hệ thống không thể đáp ứng được các yêu cầu về thời gian thưc. Ví dụ như một trong các nhiệm vụ của máy bay trinh thám là chụp ảnh và phát hiện đối tượng khả nghi trong hành trình bay của nó. Máy bay sẽ phải liên tục chụp ảnh với kích thước và độ phân giải cao, xử lý nhận dạng và báo cáo thông tin về trung tâm điều khiển. Do máy bay thực hiện việc chụp ảnh và xử lý ngay trong quá trình bay nên việc tăng tốc độ tính toán để đảm bảo tính thừoi gian thực là một vấn đề cấp thiết. Vì vậy việc nghiên cứu phát triển các phương pháp nâng cao tốc độ tính toán trong xử ảnh luôn được các nhà khoa học không ngừng nghiên cứu.Trong quá trình nhận ảnh, nhận dạng và tìm kiếm đối tượng trong ảnh thì việc phân tích xác định biên của các đối tượng trong ảnh là một khâu tiền xử lý quan trọng ban

đầu được thực hiện để khoảng vùng đói tượng. Trong phạm vi luận văn này sẽ trình bày cách thức đã thuật toán phát hiện biên ảnh chạy trên GPU để tăng tốc độ tính toán.

Thuật toán phát hiện biên sử dụng Laplaciantrong ảnh có thể được phát biểu tổng quan như sau: Mỗi điểm ảnh mới L(x,y) được xác định như công thức (3.4)

$$L(x,y) = \frac{\delta^2 I}{\delta x^2} + \frac{\delta^2 I}{\delta y^2} \quad (3.4)$$

3.2.2. Thực hiện thuật toán phát hiện biên ảnh trên GPU

Từ công thức (3.4) cho thấy các giá trị của điểm ảnh trong ảnh kết quả được xác định mà không phụ thuộc vào giá trị của các điểm lân cận trong ảnh kết quả. Do việc việc tính toán này có thể được thực một cách song song trên GPU bằng các chuyen toàn bộ ảnh đầu vào và ảnh tính toán L(x,y) sang RAM của GPU. Sau đó mỗi một phần tử của mảng (x,y) sẽ gọi hàm tính toán L(x,y) với một lõi CUDA cảu GPU.

Chương trình tính toán phát hiện biên trên GPU như sau:

```
clear all
cdata=imread('2000px.jpg');
tic
if length(size(cdata))>2 % anh mau, can chuyen ve anh da cap xam
cdata=gpuArray(rgb2gray(cdata));
end
cdata=double(cdata)/255; % chuyen sang kieu double
gaussSigma = 0.1;
edgePhobia=0.1;
dx = cdata(2:end-1,3:end) - cdata(2:end-1,1:end-2);
dy = cdata(3:end,2:end-1) - cdata(1:end-2,2:end-1);
dx2 = dx.*dx;
dy2 = dy.*dy;
dxy = dx.*dy;
gaussHalfWidth = max( 1, ceil( 2*gaussSigma ) );
ssq = gaussSigma^2;
t = -gaussHalfWidth : gaussHalfWidth;
```

```
gaussianKernel1D = exp(-(t.*t)/(2*ssq))/(2*pi*ssq); % The Gaussian 1D filter
gaussianKernel1D = gaussianKernel1D / sum(gaussianKernel1D);
smooth_dx2 = conv2( gaussianKernel1D, gaussianKernel1D, dx2, 'valid' );
smooth_dy2 = conv2( gaussianKernel1D, gaussianKernel1D, dy2, 'valid' );
smooth_dxy = conv2( gaussianKernel1D, gaussianKernel1D, dxy, 'valid' );
det = smooth_dx2 .* smooth_dy2 - smooth_dxy .* smooth_dxy;
trace = smooth_dx2 + smooth_dy2;
score = det - 0.25*edgePhobia*(trace.*trace);
toc
figure;imshow(trace);
```

Chương trình sử dụng CPU:

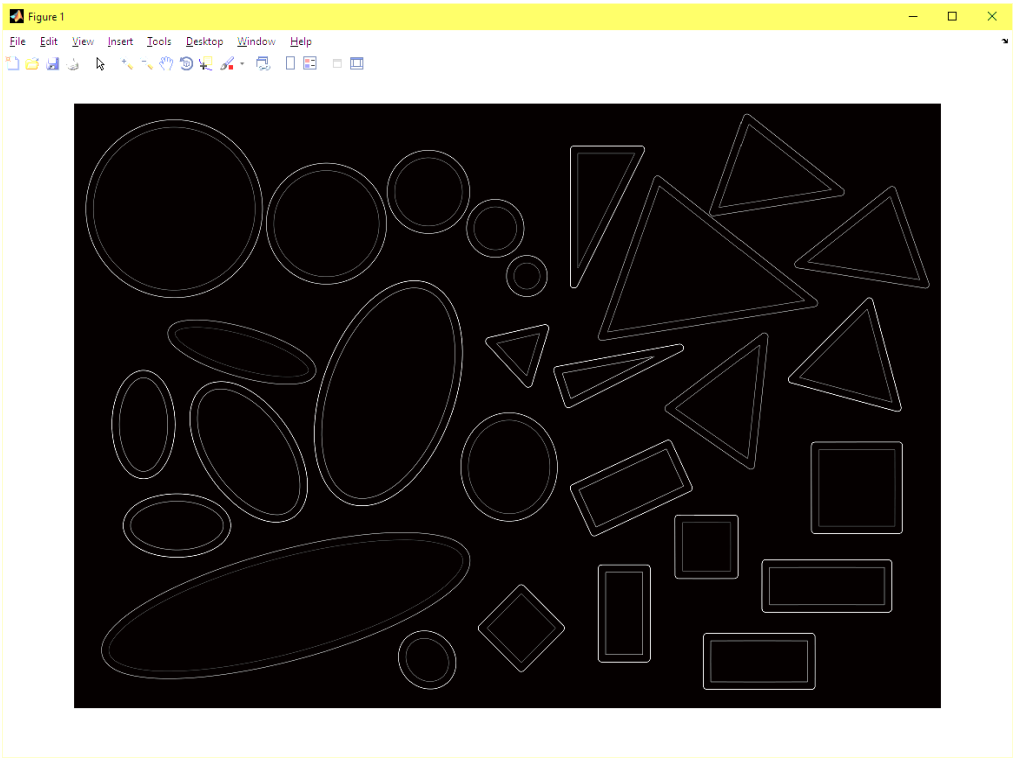
```
cdata=imread('2000px.jpg');
tic
if length(size(cdata))>2 % anh mau, can chuyen ve anh da cap xam
cdata=rgb2gray(cdata);
end
cdata=double(cdata)/255; % chuyen sang kieu double
gaussSigma = 0.5;
edgePhobia=0.5;
dx = cdata(2:end-1,3:end) - cdata(2:end-1,1:end-2);
dy = cdata(3:end,2:end-1) - cdata(1:end-2,2:end-1);
dx2 = dx.*dx;
dy2 = dy.*dy;
dxy = dx.*dy;
gaussHalfWidth = max( 1, ceil( 2*gaussSigma ) );
ssq = gaussSigma^2;
t = -gaussHalfWidth : gaussHalfWidth;
gaussianKernel1D = exp(-(t.*t)/(2*ssq))/(2*pi*ssq); % The Gaussian 1D filter
```

```
gaussianKernel1D = gaussianKernel1D / sum(gaussianKernel1D);
smooth_dx2 = conv2( gaussianKernel1D, gaussianKernel1D, dx2, 'valid' );
smooth_dy2 = conv2( gaussianKernel1D, gaussianKernel1D, dy2, 'valid' );
smooth_dxy = conv2( gaussianKernel1D, gaussianKernel1D, dxy, 'valid' );
det = smooth_dx2 .* smooth_dy2 - smooth_dxy .* smooth_dxy;
trace = smooth_dx2 + smooth_dy2;
score = det - 0.25*edgePhobia*(trace.*trace);
toc
figure;imshow(trace);
```

3.2.3. Kết quả thử nghiệm

Ảnh đầu vào:

Ảnh đầu ra sau khi phát hiện biên:



3.2.4. Đánh giá hiệu xuất tính toán

Luận van tiến hành thử nghiệm với ảnh có nội dung như ở phần 3.3.3, kích thước của ảnh được thay đổi ở mỗi được thử nghiệm sau đó chạy thuật toán trên GPU và CPU của cùng một máy tính để đo thời gian tính toán. Kết quả thử nghiệm như sau:

Kích thước ảnh	thời gian tính toán CPU (4 cores)	thời gian tính toán GPU (99 cores)
2000x1385	0,169	0,087
4000x1385	1,369	0,949
8000x2770	3,078	1,901

3.3. Tạo ảnh sơn mài

3.3.1. Cài đặt thuật toán tạo ảnh sơn mài trên GPU

Thử nghiệm tiếp theo luận văn tiến hành cài đặt thuật toán tạo ảnh sơn mài từ ảnh chụp. Đây là một thuật toán biến đổi ảnh với phép tính nhận chập ma trận là chính. Phép nhận chập là một kỹ thuật cơ bản được sử dung hầu hết trong các kỹ thuật xử lý ảnh số. Chương trình tạo ảnh sơn mài như sau:

Chương trình sử dụng GPU

```
function out = canvasEffect(im)
% Filter the image with a Gaussian kernel.
h = fspecial('gaussian');
imf = imfilter(im,h);
% Increase image contrast for each color channel.
ima = cat( 3, imadjust(imf(:,:,1)), imadjust(imf(:,:,2)), imadjust(im(:,:,3)) );
% Perform a morphological closing on the image with a 11x11 structuring
% element.
se = strel('disk',9);
out = imopen(ima,se);
clear all
anhvao=imread('city2.jpg');
figure;imshow(anhvao);
tic
tmp=gpuArray(anhvao);
anhra=canvasEffect(tmp);
figure;imshow(anhra);
toc
```

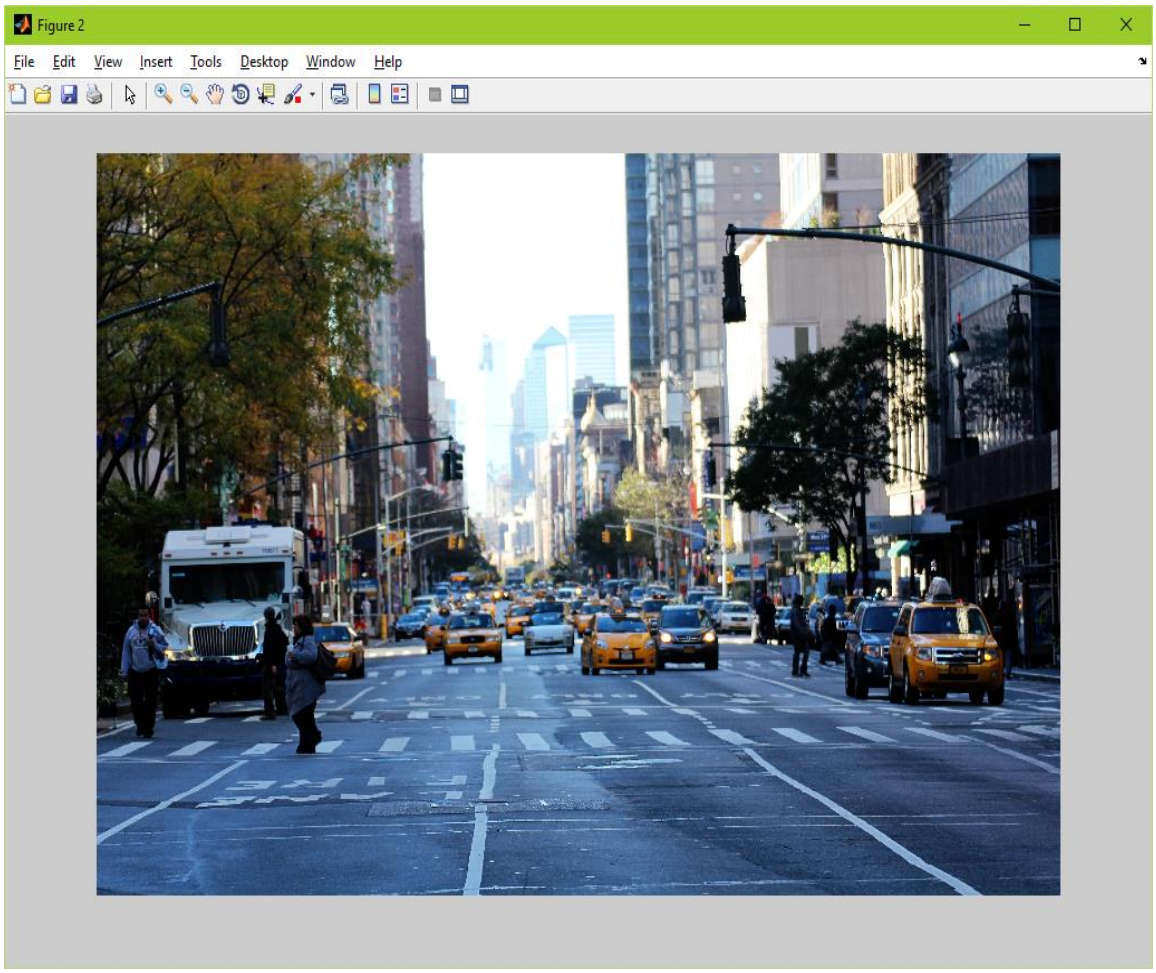
Chương trình sử dụng CPU

```
function out = canvasEffect(im)
% Filter the image with a Gaussian kernel.
h = fspecial('gaussian');
imf = imfilter(im,h);
% Increase image contrast for each color channel.
ima = cat( 3, imadjust(imf(:,:,1)), imadjust(imf(:,:,2)), imadjust(im(:,:,3)) );
% Perform a morphological closing on the image with a 11x11 structuring
% element.
se = strel('disk',9);
```

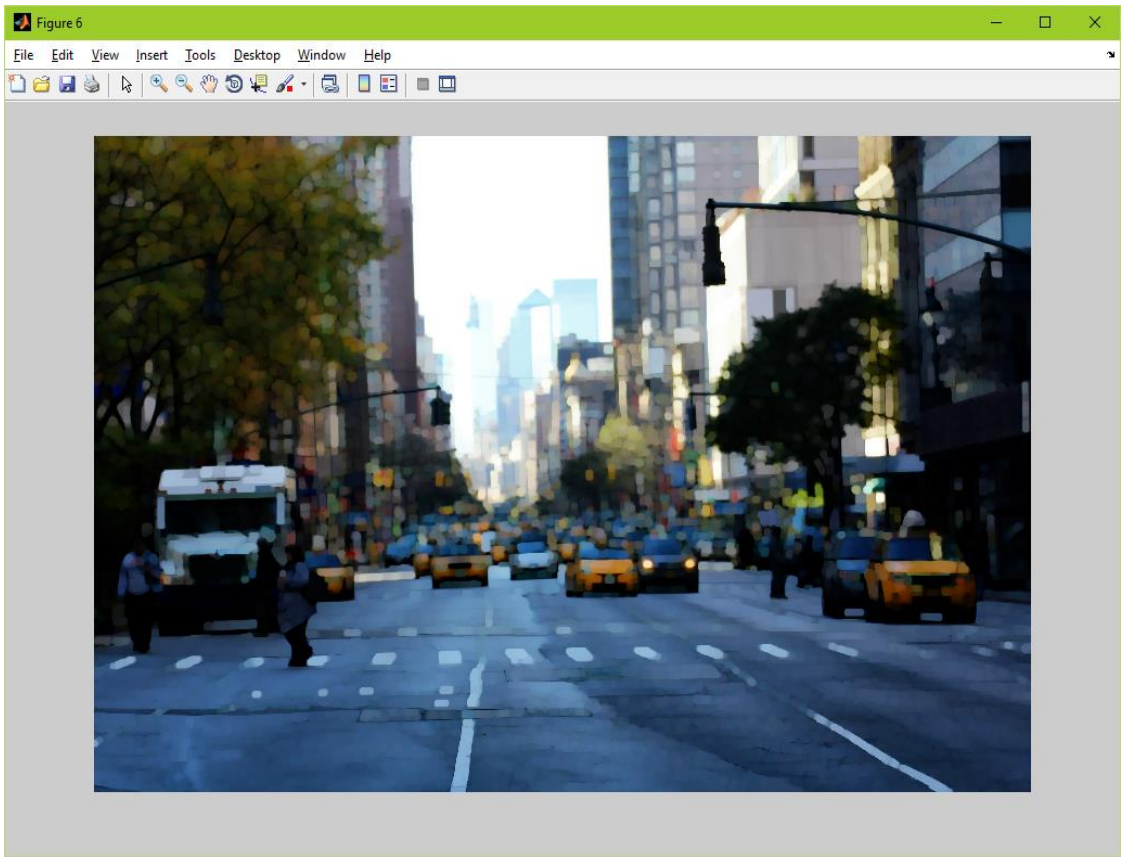
```
out = imopen(ima,se);
clear all
anhvao=imread('city2.jpg');
figure;imshow(anhvao);
tic
anhra=canvasEffect(anhvao);
figure;imshow(anhra);
toc
```

3.3.2. Kết quả thử nghiệm

Ảnh vào:



Ảnh ra:



3.3.3. Đánh giá hiệu xuất tính toán

Để đánh giá hiệu thuật tính toán trên GPU và CPU, luận van tiến hành thay đổi kích thước của ảnh đầu vào và chạy chương trình trên GPU và CPU của cùng một máy tính. Kết quả thời gian tính toán như sau:

Kích thước ảnh	thời gian tính toán CPU (4 cores)	thời gian tính toán GPU (99 cores)
5184x3456	8,17	2,69
2592x1728	2,16	0,9
1296x864	0,76	0,48

3.4. Hướng phát triển

Qua các thử nghiệm ở trên cho thấy đối với các bài toán dữ liệu lớn, yêu cầu các phép toán số dấu phẩy động thì việc thực hiện trên GPU cho tốc

độ tính toán nhanh hơn nhiều lần so với thực hiện trên CPU.Ngày nay với sự bùng nổ của mạng internet và đặc biệt các dữ liệu online thì kích thước dữ liệu lại ngày càng trở nên khổng lồ, và thực sự quả tải nếu chỉ sử dụng 1 CPU.Việc nghiên cứu cài đặt các thuật toán trên 1 hoặc nhiều GPU là một như cầu cần thiết và cấp bách. Trong thực tế hiện nay các công ty lớn như Google hay Amazon cũng đã công bố các công cụ huấn luyện và cài đặt mạng nơron học sâu (Deep Learning NN) chạy trên GPU. Các nghiên cứu và công bố này đã thay đổi và tạo ra hướng đi mới trong việc sử dụng mạng Nơron trong các lĩnh vực xử lý, nhận dạng, học máy.Trong trương lại tác giả sẽ tiếp tục nghiên cứu phát triển cài đặt các thuật toán, các phương pháp xử lý tín hiếu số, ảnh áp dụng mạng Nơron trên nền tảng GPU.

KẾT LUẬN

Luận văn đã nghiên cứu tổng quan về tính toán song song, đó là điều kiện cần để phát triển ứng dụng GPU cho mục đích thông dụng. Tác giả luận văn cũng đã tìm hiểu về cơ chế hoạt động của GPU, các kiến trúc bên trong nó, mô hình lập trình trên GPU. Trong chương 2, luận văn đã tìm hiểu công cụ lập trình GPU phổ biến nhất hiện nay là CUDA. Tác giả luận văn cũng trình bày chi tiết các mô hình lập trình, thiết lập phần cứng trên card đồ họa của Nvidia, giao diện lập trình cũng như các chỉ dẫn hiệu năng khi chạy ứng dụng trên card đồ họa.Từ các hiệu biết trên, tác giả đã thực hiện thử nghiệm năng lực tính toán của GPU so sánh với CPU để kiểm chứng những điều mà lý thuyết đã nói. Các kết quả thử nghiệm được trình bày chi tiết trong chương 3 của luận văn.

Với các kết quả đạt được, tác giả mong muốn có các nghiên cứu thêm về cải tiến hiệu năng bài toán mô phỏng tiếp tục nghiên cứu phát triển cài đặt các thuật toán, các phương pháp xử lý tín hiếu số, ảnh áp dụng mạng Nơron trên nền tảng GPU Mong rằng các kết quả nghiên cứu trong tương lai của luận văn sẽ đạt được điều đó.

TÀI LIỆU THAM KHẢO

Tài liệu tiếng việt

[1] Trương Văn Hiệu (2011), "*Nghiên cứu các giải thuật song song trên hệ thống xử lý đồ họa GPU đa lõi*", luận văn thạc sĩ, trường Đại học Đà Nẵng.

[2] Nguyễn Việt Đức – Nguyễn Nam Giang (2012), "Xây dựng thuật toán song song tìm đường đi ngắn nhất với CUDA", luận văn thạc sỹ, trường Đại học Công nghệ Hồ Chí Minh.

[3] Nguyễn Thị Thùy Linh (2009), "*Tính toán hiệu năng cao với bộ xử lý đồ họa GPU và ứng dụng*", luận văn thạc sĩ, trường Đại học Công nghệ Hà Nội.

Tài liệu tiếng anh

[4] Jason Sanders, Edward Kandrot, "*CUDA by example*", an introduction to General-Purpose GPU programming.

[5] Maciej Matyka, "*GPGPU programming on example of CUDA*", Institute of Theoretical Physics University of Wroclaw.

[6] NVIDIA, "*High performance computing with CUDA*", Users Group Conference San Diego, CA June 15, 2009.