

**ĐẠI HỌC QUỐC GIA HÀ NỘI  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**

**Nguyễn Thị Thùy Linh**

**TÍNH TOÁN HIỆU NĂNG CAO VỚI BỘ XỬ LÝ  
ĐỒ HỌA GPU VÀ ỨNG DỤNG**

**LUẬN VĂN THẠC SĨ**

Hà Nội - 2009

## **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 12 năm 2009  
Học viên

Nguyễn Thị Thùy Linh

## **LỜI CẢM ƠN**

Những kiến thức căn bản trong luận văn này là kết quả của ba năm (2005-2008) 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 trường Đại học, Viện nghiên cứu trong và ngoài nước 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 Hệ thống thông tin – 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. Nguyễn Hải Châu 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 Ngân hàng TMCP Công Thương Việt Nam 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 K12T3.

Hà Nội, tháng 12 năm 2009  
Học viên

Nguyễn Thị Thùy Linh

# MỤC LỤC

LỜI CAM ĐOAN.....	2
LỜI CẢM ƠN.....	3
MỤC LỤC.....	4
MỞ ĐẦU.....	6
DANH MỤC THUẬT NGỮ.....	7
DANH MỤC HÌNH VẼ, BẢNG BIỂU.....	8
Danh mục hình vẽ.....	8
Danh mục bảng biểu.....	8
Chương 1. TỔNG QUAN VỀ TÍNH TOÁN SONG SONG VÀ GPU.....	9
1.1. Tổng quan về tính toán song song.....	9
1.1.1. Các mô hình máy tính song song.....	10
1.1.2. Mô hình lập trình song song.....	12
1.1.3. Sự cần thiết của công cụ phát triển ứng dụng song song.....	16
1.2. Tổng quan về GPU.....	17
1.2.1. Giới thiệu GPU.....	17
1.2.2. Lịch sử phát triển GPU.....	18
1.2.3. Kiến trúc GPU.....	20
1.2.4. Tính toán trên GPU.....	25
1.2.5. Môi trường phần mềm.....	28
1.2.6. Kỹ thuật và ứng dụng.....	31
Chương 2. HỆ THỐNG CHUỖNG TRÌNH DỊCH VÀ NGÔN NGỮ LẬP TRÌNH GPU.....	37
2.1. Giới thiệu về môi trường phát triển CUDA.....	37
2.2. Mô hình lập trình.....	39
2.2.1. Bộ đồng xử lý đa luồng mức cao.....	39
2.2.2. Gom lô các luồng (Thread Batching).....	39
2.2.3. Mô hình bộ nhớ.....	41
2.3. Thiết lập phần cứng.....	42
2.3.1. Tập các bộ đa xử lý SIMD với bộ nhớ dùng chung trên chip.....	42
2.3.2. Mô hình thực thi.....	44
2.3.3. Khả năng tính toán.....	45
2.3.4. Đa thiết bị.....	46
2.3.5. Cơ chế chuyển đổi.....	46
2.4. Giao diện lập trình ứng dụng.....	46
2.4.1. Mở rộng cho ngôn ngữ lập trình C.....	46
2.4.2. Mở rộng ngôn ngữ.....	47
2.4.3. Thành phần chung trong thời gian chạy.....	52
2.4.4. Thành phần thiết bị thời gian chạy.....	55
2.5. Hướng dẫn hiệu năng.....	58
2.5.1. Hiệu năng lệnh.....	58
2.5.2. Số lượng luồng trong một khối.....	64
2.5.3. Truyền dữ liệu giữa Host và device.....	66
2.5.4. Lợi ích của việc tổ chức bộ nhớ.....	66
Chương 3. ỨNG DỤNG GPU VÀO BÀI TOÁN N-BODY VÀ THỬ NGHIỆM CHUỖNG TRÌNH.....	67
3.1. Bài toán mô phỏng N-body.....	67
3.2. Xây dựng bài toán N-body trên CPU.....	69
3.2.1. Thuật toán tích hợp thời gian Verlet.....	69
3.2.2. Công thức tính lực cơ bản và tính tiềm năng.....	69
3.2.3. Thuật toán mô phỏng N-Body.....	70

3.3.	Xây dựng bài toán N-body trên GPU .....	71
3.4.	Thử nghiệm.....	72
3.4.1.	Môi trường thử nghiệm: .....	72
3.4.2.	Kết quả thử nghiệm .....	73
3.5.	Kết luận thử nghiệm .....	76
KẾT LUẬN .....		77
TÀI LIỆU THAM KHẢO .....		78

## MỞ ĐẦU

Các bộ xử lý đồ họa (GPU - Graphic Processing Unit) đã trở thành một phần không thể tách rời của hệ thống máy tính ngày nay. Trong sáu năm vừa qua đã đánh dấu sự gia tăng ấn tượng trong hiệu suất và khả năng của GPU. GPU hiện đại không chỉ là một công cụ xử lý đồ họa mạnh mẽ mà còn là một bộ xử lý hỗ trợ lập trình song song ở mức cao, giúp xử lý các bài toán số học lập trình tính năng xử lý số học phức tạp và băng thông bộ nhớ tăng hơn đáng kể so với CPU cùng loại. Sự tăng tốc nhanh chóng của GPU trong cả khả năng hỗ trợ lập trình và năng lực tính toán của nó đã tạo ra một xu hướng nghiên cứu mới. Một cộng đồng đã nghiên cứu và đã ánh xạ thành công một lượng lớn các vấn đề phức tạp đòi hỏi tính toán lớn vào GPU. Điều này trong nỗ lực chung nhằm mục đích ứng dụng GPU vào giải quyết các bài toán hiệu năng cao của tính toán hiện đại. Tính toán mục đích thông dụng trên GPU (GPGPU) là một thay thế hấp dẫn cho CPU tại trong hệ thống máy tính hiện đại. Trong một tương lai không xa, chúng ta có thể sẽ thấy GPU sẽ đảm nhận thay cho CPU những công việc như xử lý hình ảnh và đồ họa, các tính toán phức tạp thay vì chỉ dừng lại ở những ứng dụng trò chơi 3D.

Với những ý nghĩa thực tiễn đó, luận văn đi vào nghiên cứu tính toán thông dụng trên GPU và thử nghiệm trực tiếp trên bài toán tính toán hiệu năng cao tiêu biểu là n-body.

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: Hệ thống chương trình dịch và ngôn ngữ lập trình GPU**. 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: Ứng dụng GPU vào bài toán n-body và thử nghiệm chương trình**. Trên cơ sở các kiến thức được trình bày ở các chương trên, tác giả luận văn đã tiến hành cài đặt và thử nghiệm mô phỏng n-body 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 n-body chạy trên GPU.

## DANH MỤC THUẬT NGỮ

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

# DANH MỤC HÌNH VẼ, BẢNG BIỂU

## **Danh mục hình vẽ**

Hình 1. Máy tính song song có bộ nhớ chia sẻ.....	10
Hình 2. Máy tính song song có bộ nhớ phân tán.....	11
Hình 3. Hoạt động của hệ thống SIMD.....	11
Hình 4. Hoạt động của hệ thống MIMD.....	12
Hình 5. Mô hình lập trình đa luồng.....	14
Hình 6. Mô hình truyền thông điệp.....	14
Hình 7. Mô hình song song dữ liệu.....	15
Hình 8. Mô hình SPMD.....	16
Hình 9. Mô hình MPMD.....	16
Hình 10: Ảnh chụp 3dfx Voodoo3.....	19
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.....	25
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.....	33
Hình 13: Kiến trúc bộ phần mềm CUDA.....	37
Hình 14: Các thao tác thu hồi và cấp phát bộ nhớ.....	38
Hình 15: Vùng nhớ dùng chung mang dữ liệu gần ALU hơn.....	39
Hình 16: Khối luồng.....	41
Hình 17: Mô hình bộ nhớ.....	42
Hình 18: Mô hình phần cứng.....	44
Hình 19: Hình ảnh mô phỏng N-body [~8].....	68
Hình 20: Biểu đồ so sánh thời gian thực hiện giữa GPU và CPU theo số lượng phần tử trong mô phỏng n-body.....	73
Hình 22: Biểu đồ thể hiện tỷ số tăng tốc CPU/GPU khi số phần tử trong mô phỏng n-body tăng.....	74
Hình 21: Tải tính toán trên CPU khi chạy mô phỏng n-body với số phần tử 256K. 1 CPU luôn ở 100%, đôi khi chiếm thêm tải 100% của các CPU khác.....	75
Hình 23: Biểu đồ hiệu năng trên GPU Geforce 8800 GTX trong mô phỏng n-body khi số phần tử tăng.....	76

## **Danh mục bảng biểu**

Bảng 1: Kết quả thử nghiệm bài toán N-body trên GPU Nvidia GeForce 8800 GTX và CPU Intel(R) Core(TM)2 Quad 2.66GHz.....	73
Bảng 2: Tỷ số tăng tốc giữa CPU và GPU.....	74
Bảng 3: Tốc độ xử lý trên GPU 8800 GTX khi số phần tử tăng.....	76

# Chương 1.

## TỔNG QUAN VỀ TÍNH TOÁN SONG SONG VÀ GPU

### 1.1. Tổng quan về tính toán song song

Khoa học kỹ thuật ngày càng phát triển, đặt ra nhiều bài toán với khối lượng tính toán rất lớn. Trong số đó có những bài toán mà kết quả chỉ có ý nghĩa nếu được hoàn thành trong khoảng thời gian cho phép. Ví dụ như các tính toán trong thời gian thực, mô phỏng các hoạt động ở mức lượng tử, tính quỹ đạo chuyển động của vật thể trong không gian, dự báo thời tiết...

Để giải quyết những bài toán này, người ta đã nghiên cứu tăng tốc độ tính toán bằng hai phương pháp hay kết hợp cả hai:

- *Phương pháp 1:* Cải tiến công nghệ, tăng tốc độ xử lý của máy tính. Công việc này đòi hỏi nhiều thời gian, công sức và tiền của, nhưng tốc độ cũng chỉ đạt được đến một giới hạn nào đó.
- *Phương pháp 2:* Chia bài toán ra thành những công việc nhỏ để có thể chạy song song trên nhiều bộ xử lý.

Việc phát triển công nghệ tính toán theo phương pháp 2 đã cho ra đời công nghệ tính toán song song, đó là việc **sử dụng đồng thời nhiều tài nguyên tính toán để giải quyết một bài toán**. Các tài nguyên tính toán có thể bao gồm một máy tính với nhiều bộ vi xử lý, một tập các máy tính kết nối mạng hay là một sự kết hợp của hai dạng trên. Công nghệ tính toán song song cho phép giảm thời gian thực thi bài toán tùy thuộc cách phân chia và số bộ xử lý thực thi chương trình. Nguyên tắc quan trọng nhất của tính toán song song chính là tính **đồng thời** hay **xử lý nhiều tác vụ cùng một lúc**.

*Trong tính toán song song hiện nay, có hai công nghệ chính:*

Thứ nhất là sử dụng các **siêu máy tính** với rất nhiều bộ xử lý được tích hợp bên trong được thiết kế đồng bộ cả về phần cứng và phần mềm. Các công nghệ được áp dụng trong các siêu máy tính thường là các công nghệ tiên tiến làm cho giá thành của hệ thống siêu máy tính tăng rất cao. Vì thế các siêu máy tính thường được sử dụng trong các lĩnh vực mà vấn đề tính toán phức tạp, nhạy cảm và yêu cầu thời gian thực như mô phỏng thực hiện của các động cơ máy bay, quốc phòng, vũ trụ...

Cách thứ hai là kết nối các máy tính lại với nhau và cùng thực hiện bài toán. Hệ thống các máy tính kết nối này chính là **hệ thống tính toán song song phân cụm**. Hệ thống này có ưu điểm là giá thành rẻ hơn rất nhiều so với siêu máy tính có cùng sức mạnh (do sử dụng các thiết bị thông thường) và tính linh hoạt của hệ thống (số nút, số bộ xử lý, bộ nhớ, thiết bị mạng... đều mang tính tùy biến cao). Sự phát triển mạnh mẽ của mạng máy tính, các công nghệ mạng hiện nay đã lấp đi hạn chế về truyền thông trong hệ thống máy tính song song phân cụm làm cho nó được phát triển rộng rãi. Các lĩnh vực sử dụng hệ thống tính toán song song phân cụm thường yêu cầu tính toán

không quá lớn, không yêu cầu thời gian thực như xử lý ảnh, nhận dạng vân tay, tính toán kết cấu công trình, mô phỏng các thí nghiệm...

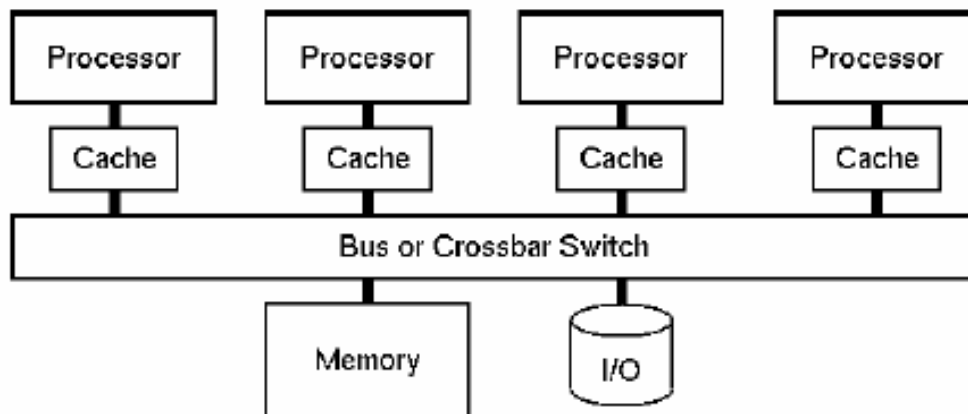
### 1.1.1. Các mô hình máy tính song song

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, hay các hệ thống nhúng ... 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 coi là hệ thống máy tính song song [8].

Dựa vào sự phân biệt ở kết nối giữa các bộ xử lý (hay thành phần xử lý), giữa bộ xử lý và bộ nhớ mà có rất nhiều loại kiến trúc máy tính song song khác nhau. Nhưng theo nguyên tắc phân loại của Flynn thì có hai kiến trúc máy tính song song thông dụng sau [8]:

- **SIMD** - Single Instruction Multiple Data: đơn lệnh đa dữ liệu
- **MIMD** - Multiple Instruction Multiple Data: đa lệnh đa dữ liệu

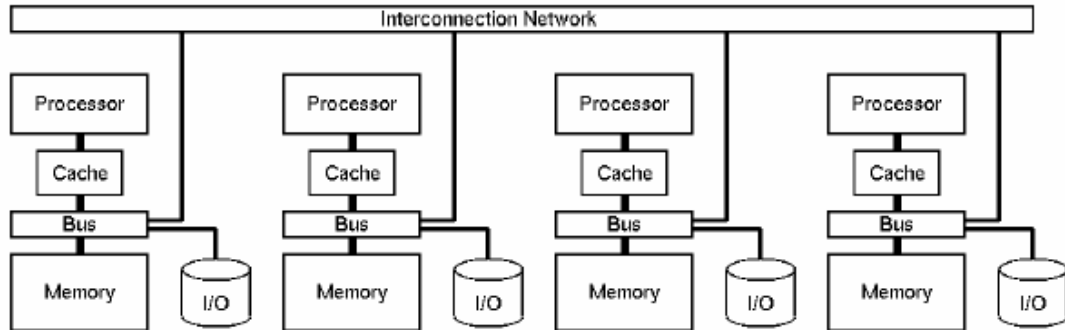
Sự phân chia này được dựa trên kiến trúc bộ nhớ của các máy tính song song. Các **máy tính song song có bộ nhớ chia sẻ** (shared memory) có nhiều bộ xử lý cùng được truy cập đến một vùng nhớ tổng thể dùng chung. Tất cả các sự thay đổi nội dung bộ nhớ do một bộ xử lý tạo ra sẽ được nhận biết bởi các bộ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



Hình 1. Máy tính song song có bộ nhớ chia sẻ

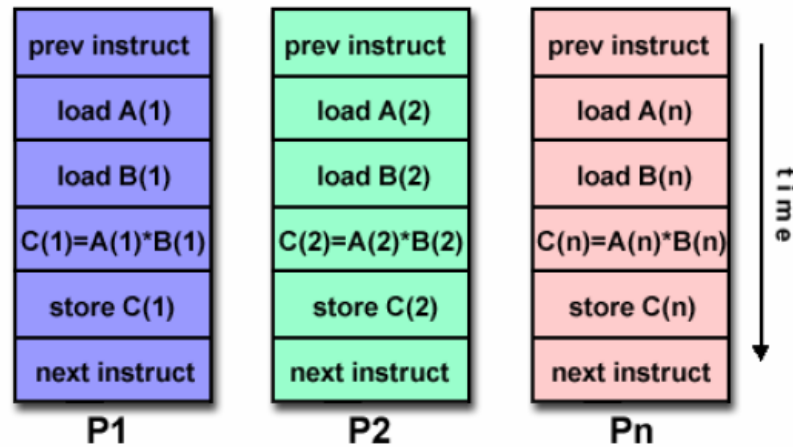
Trong lớp máy tính này có thể phân chia làm 2 lớp nhỏ hơn: Lớp UMA (Uniform Memory Access – Truy cập bộ nhớ đồng nhất) cho phép thời gian truy cập bộ nhớ đối với mỗi bộ xử lý là như nhau; Lớp NUMA (Non-Uniform Memory Access – Truy cập bộ nhớ không đồng nhất) có thời gian truy cập bộ nhớ không phải lúc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

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 2. Máy tính song song có bộ nhớ phân tán

### 1.1.1.1. Mô hình đơn lệnh đa dữ liệu - SIMD



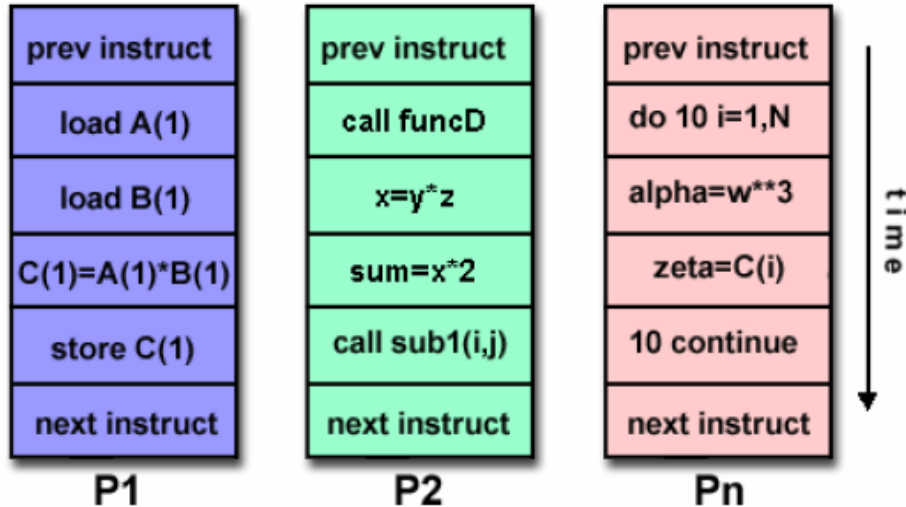
Hình 3. Hoạt động của hệ thống SIMD

SIMD là một kiểu máy tính song song có tất cả các bộ xử lý chỉ thực hiện một lệnh duy nhất. Tuy nhiên lệnh này được thực hiện trên các bộ dữ liệu khác nhau ứng với từng bộ xử lý khác nhau.

Mô hình này có ưu điểm là đơn giản trong phần cứng cũng như phần mềm nhưng chỉ phù hợp để giải quyết các vấn đề tương đối đặc thù có tính cân đối cao trong xử lý như xử lý ảnh ... Các giải thuật cho các đa máy tính thường chạy không hiệu quả trên các máy SIMD.

### 1.1.1.2. Mô hình đa lệnh đa dữ liệu - MIMD.

MIMD là một mô hình kiến trúc máy tính song song thông dụng hiện nay. Với mô hình này thì tất cả các bộ xử lý sẽ thực hiện các lệnh khác nhau với các dữ liệu riêng khác nhau. Sự thực thi các lệnh có thể theo cơ chế đồng bộ hoặc không đồng bộ (synchronous or asynchronous), xác định hay không xác định (deterministic or non-deterministic). Điều này giúp cho mô hình MIMD rất linh hoạt trong việc xử lý song song.



Hình 4. Hoạt động của hệ thống MIMD

Tuy nhiên, cùng với tính linh hoạt của mình, mô hình MIMD cũng mang theo một sự phức tạp nhất định. Việc lập trình được những bài toán song song theo mô hình này đòi hỏi nhiều công sức nghiên cứu, phân tích bài toán để tìm ra một cách phân rã tối ưu. Để lập trình theo mô hình này, lập trình viên cần có trình độ cao trong cả chuyên môn và trong kỹ thuật lập trình song song.

### 1.1.2. 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 nó 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 định vị 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 song song bạn cần phải biết 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:

- **Song song hoá ngầm định** (implicit parallelism): bộ biên dịch hay một vài chương trình khác tự động phân chia các công việc đến các bộ xử lý.
- **Song song hoá bằng tay** (explicit parallelism): người lập trình phải tự phân chia chương trình của anh ta để nó có thể thực thi 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.

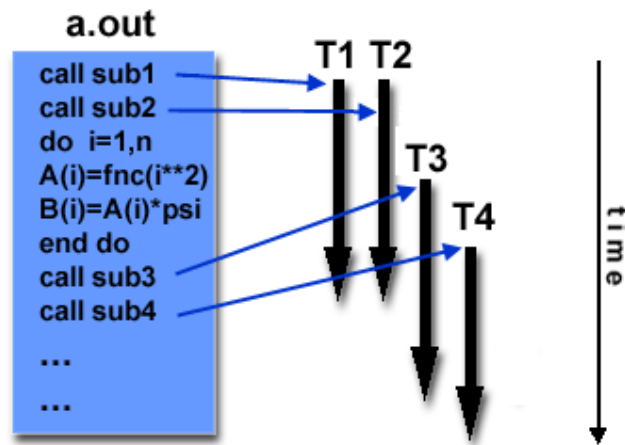
Việc truyền thông giữa các bộ xử lý là một công việc không thể thiếu của lập trình song song. Có hai kỹ thuật truyền thông chủ yếu là: dùng **bộ nhớ chia sẻ** (shared memory) hoặc **truyền thông điệp** (message passing).

Một mô hình lập trình song song là sử dụng một tập các kỹ thuật phần mềm để thể hiện các giải thuật song song và đưa ứng dụng vào thực hiện trong hệ thống song song. Mô hình 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: Đa luồng (Threads), Truyền thông điệp (Message Passing), Song song dữ liệu (Data Parallel), Lai (Hybird) [9].

#### **1.1.2.1. Mô hình đa luồng**

Trong mô hình đa luồng (*Threads*), một luồng có thể có rất nhiều luồng xử lý. Ví dụ, một chương trình chính a.out được đưa vào hệ thống để chạy. Nó sẽ thực hiện một vài công việc tuần tự rồi tạo ra một số luồng con. Mỗi luồng có dữ liệu cục bộ riêng của mình nhưng cũng có thể truy cập đến các tài nguyên chung của chương trình a.out. Mỗi luồng có thể được coi là một chương trình con của chương trình chính và có thể được thực hiện song song với các luồng khác.



Hình 5. Mô hình lập trình đa luồng

Ở khía cạnh lập trình thì mô hình đa luồng có được thể hiện bao gồm:

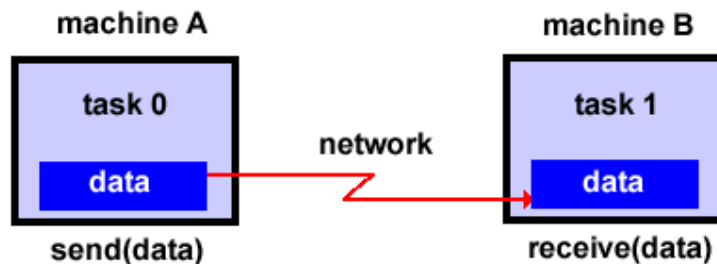
- Một thư viện các hàm được gọi trong mã nguồn chương trình song song.
- Một tập các chỉ dẫn biên dịch trong mã nguồn chương trình tuần tự hay song song.

Hai hệ thư viện lập trình song song cho mô hình này là **POSIX Threads** và **OpenMP**.

#### 1.1.2.2. 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. Nó thường áp dụng cho các hệ thống phân tán. Các đặc trưng của mô hình là:

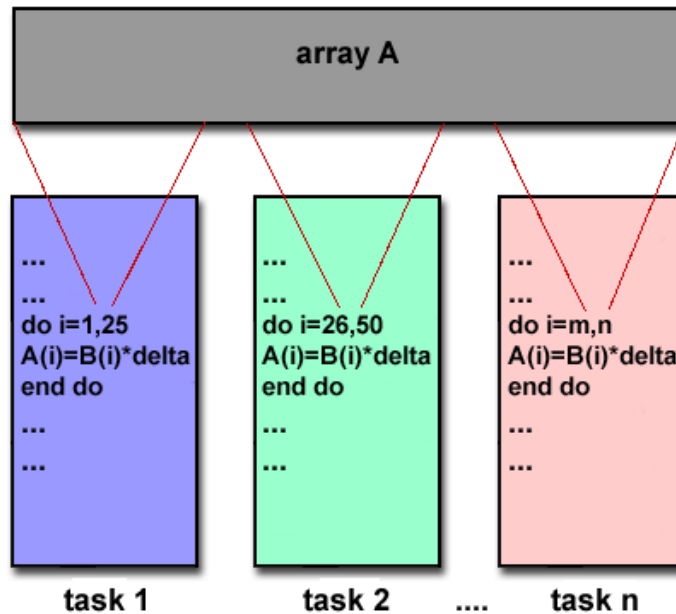
- Một tập các luồng sử dụng vùng nhớ cục bộ riêng của chúng trong suốt quá trình tính toán
- Nhiều luồng có thể cùng sử dụng một tài nguyên vật lý.
- Các luồng trao đổi dữ liệu bằng cách gửi nhận các thông điệp
- Việc truyền dữ liệu thường yêu cầu thao tác điều phối thực hiện bởi mỗi luồng. Ví dụ, một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



Hình 6. Mô hình truyền thông điệp

Về mặt lập trình thì mô hình truyền thông điệp thể hiện bởi việc sử dụng các thủ tục con của hệ thư viện lập trình vào bên trong mã nguồn. Hai hệ thư viện phổ biến nhất hiện nay là *MPI* (Message Passing Interface) và *PVM* (Parallel Virtual Machine).

### 1.1.2.3. Mô hình song song dữ liệu



Hình 7. Mô hình song song dữ liệu

Mô hình song song dữ liệu (*Data Parallel*) nhấn mạnh các thao tác song song trên một tập dữ liệu. Các luồng làm việc chung trên cùng một cấu trúc dữ liệu nhưng ở các phần khác nhau. Với kiến trúc bộ nhớ chia sẻ, tất cả các luồng có thể truy cập cấu trúc dữ liệu chung thông qua vùng nhớ dùng chung. Với kiến trúc bộ nhớ phân tán thì cấu trúc dữ liệu chung được chia ra thành từng phần và định vị trên vùng nhớ cục bộ của mỗi luồng.

Lập trình với mô hình song song dữ liệu thường được thực hiện bởi việc viết chương trình cùng với việc xây dựng song song dữ liệu. Việc làm này có thể thực hiện bởi các hàm thư viện hoặc các chỉ dẫn biên dịch của chương trình biên dịch song song dữ liệu như *Fortran 90* hay *HPF* (High Performance Fortran).

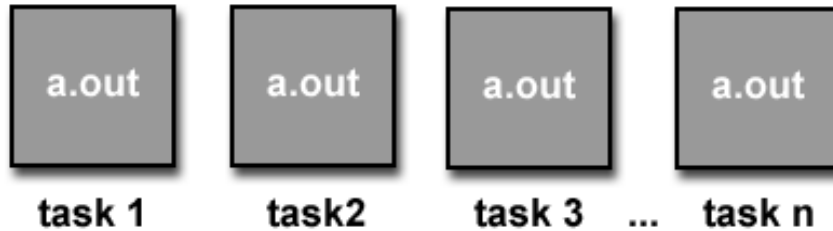
### 1.1.2.4. Các mô hình khác

#### Mô hình lai

Mô hình lai (*hybird*) là sự kết hợp của hai hay nhiều mô hình lập trình song song để tạo ra sự thuận lợi và hiệu quả hơn trong việc tính toán. Một ví dụ hay thấy nhất là sử dụng mô hình truyền thông điệp (MPI) kết hợp với mô hình đa luồng (POSIX Threads hay OpenMP) để tăng sức mạnh tính toán bằng cách sử dụng các máy SMP (Symmetric Multiprocessor).

#### Mô hình đơn chương trình đa dữ liệu

Mô hình đơn chương trình đa dữ liệu (Single Program Multiple Data - SPMD) là một mô hình lập trình ở mức cao mà có thể thực hiện bởi sự kết hợp các mô hình lập trình song song ở trên. Một chương trình được thực thi bởi tất cả các tác vụ cùng một lúc và các tác vụ sử dụng các dữ liệu khác nhau. Trong một thời điểm bất kỳ, các tác vụ có thể thực thi cùng một lệnh hay các lệnh khác nhau trong cùng chương trình.



Hình 8. Mô hình SPMD

### Mô hình đa chương trình đa dữ liệu

Giống như SPMD, mô hình đa chương trình đa dữ liệu (Multiple Program Multiple Data - MPMD) là một mô hình lập trình ở mức cao mà có thể thực hiện bởi sự kết hợp các mô hình lập trình song song ở trên. Mỗi ứng dụng MPMD thường thì có nhiều chương trình được thực thi bởi các tác vụ khác nhau và mỗi tác vụ thì lại sử dụng các dữ liệu khác nhau.



Hình 9. Mô hình MPMD

### 1.1.3. Sự cần thiết của công cụ phát triển ứng dụng song song

Lập trình là một công việc đòi hỏi cần đầu tư nhiều công sức và thời gian. Vì thế các môi trường phát triển tích hợp đã được phát triển từ rất sớm nhằm trợ giúp cho các lập trình viên thuận lợi hơn trong việc lập trình đồng thời làm giảm thời gian lập trình. Hiện nay, các môi trường phát triển tích hợp như Microsoft Visual Studio, Borland Studio, Eclipse, KDevelop, Anjuta ... thực sự đã làm cho việc lập trình trở lên dễ dàng thậm chí đối với cả những người mới bắt đầu học lập trình.

Đối với việc lập trình song song như đã đề cập trong phần 1.4, đòi hỏi cần có một mô hình lập trình song song cụ thể. Các mô hình lập trình song song này thường cung cấp một thư viện lập trình cho phép lập trình song song theo một trong những ngôn ngữ lập trình thông dụng, thường là C/C++ hay Fortran. Nhưng để biên dịch hay chạy chương trình thì cần phải dùng các công cụ ứng với từng mô hình lập trình chứ không

phải sử dụng các trình biên dịch của các ngôn ngữ lập trình. Các công cụ này thường được sử dụng dưới dạng dòng lệnh (console), chẳng hạn như *mpicc*, *mpirun* đối với mô hình lập trình song song truyền thống điệp MPI. Công việc lập trình song song sẽ gặp nhiều khó khăn đối với lập trình viên nhất là khi phải phát triển các ứng dụng lớn.

Mặt khác trong lập trình thì lỗi là điều không thể tránh khỏi, các lỗi trong lập trình song song lại càng phức tạp hơn so với lập trình tuần tự. Có sự trợ giúp của phần mềm gỡ rối trong lập trình song song việc lập trình sẽ trở lên thuận lợi hơn. Ngoài ra, các hệ thống tính toán song song thường có kiến trúc phức tạp khiến cho việc mô hình hoá và lập trình các bài toán đòi hỏi tính chuyên nghiệp và sự hiểu biết sâu về tính toán song song. Do vậy việc xây dựng một công cụ phát triển ứng dụng song song là rất cần thiết tạo cơ sở cho việc ứng dụng tính toán song song trong khoa học kỹ thuật và trong cuộc sống.

Nắm bắt nhu cầu này các công ty, tổ chức, trường đại học trên thế giới cũng đã nghiên cứu xây dựng nhiều công cụ phát triển ứng dụng song song. Các công cụ này đa phần ở mức thử nghiệm nghiên cứu, chưa được sử dụng rộng rãi. Các công cụ có thể kể đến là: Sun HPC ClusterTools [10], PTP-Eclipse [~6], P-GRADE (Parallel Grid Run-time and Application Development Environment) [~29], PADE (Parallel Applications Development Environment) [~4]. Mỗi môi trường phát triển tích hợp này thường chỉ thiết kế cho một mô hình lập trình song song cụ thể và được áp dụng vào một hệ thống cụ thể mà công ty, tổ chức, trường đại học đang có. Chưa có một công cụ nào có thể áp dụng cho mọi mô hình lập trình song song và có thể triển khai trên mọi hệ thống. Mặc dù thế, các công cụ này cũng đã hỗ trợ cho lập trình viên thuận lợi hơn rất nhiều trong việc lập trình giải quyết các bài toán song, làm đơn giản hoá các bước phát triển các ứng dụng song song.

## **1.2. Tổng quan về GPU**

### **1.2.1. Giới thiệu GPU**

Bộ xử lý đồ họa (**Graphics Processing Unit**) 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.

### 1.2.2. Lịch sử phát triển GPU

GPU [~32] là bộ xử lý gắn với card đồ họa, chuyên dùng tính toán các phép toán dấu phẩy 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 phẩy 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 định khả năng của bộ gia tốc đồ họa. Chúng được sử dụng chủ yếu trong các trò chơi 3D, 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. Chip 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 thể 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 bất 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. Bời thời gian này, bộ tăng tốc Windows với đặc tính

cố định chức năng nói chung đặt 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 đáng cho ý nhất cho các chip đồ họa 3D giá thành rẻ là ViRGE S3, ATI Rage, và Matrox Mystique.



Hình 10: Ảnh chụp 3dfx Voodoo3

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. DirectX 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. DirectX 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 sung 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 DirectX 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 phẩy 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, âm đậm, thô ráp, hoặc thậm chí căng mịn hoặc lỗi lổm.

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.

### **1.2.3. 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 trung 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.

### **1.2.3.1. 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 trúc 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 màn hình:**

*Quét màn hình* (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.

### **1.2.3.2. 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 đồ 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 phẩy 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 đồ 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.

### **1.2.3.3. Kiến trúc của GPU hiện đại**

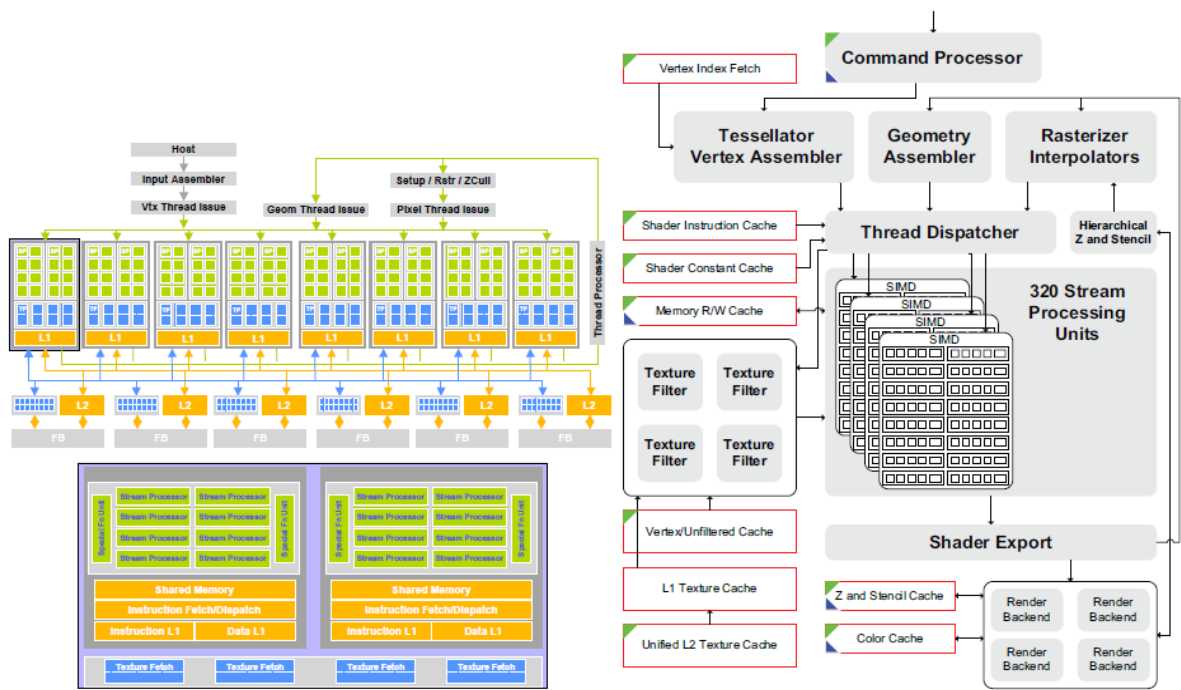
Trong phần giới thiệu, chúng tôi ghi nhận rằng GPU được xây dựng cho các nhu cầu ứng dụng khác nhau so với CPU, đó là các yêu cầu tính toán lớn chạy song song, với trọng tâm là thông lượng hơn là độ trễ. Do đó, các kiến trúc của GPU phát triển theo một hướng khác so với CPU.

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 toán 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 đồ 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 đồ 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 đồ 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 **đồ 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

### 1.2.4. 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.

#### 1.2.4.1. 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 mảng) 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 phẩy 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 đồ 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 nay 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.

#### **1.2.4.2. *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.

##### **1) 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 mảnh 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.

##### **2) 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 mảnh 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 general - 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.

### 3) 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.

### **1.2.5. 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 hãng hái nhất bắt tay vào. Với DirectX 9, lập trình đồ bóng cao cấp đã được thực hiện có thể thông qua ngôn ngữ đồ bóng cấp cao ("high-level shading language" - HLSL), nó được biểu diễn giống như giao diện lập trình C cho lập trình đồ 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 đồ bóng OpenGL (OpenGL Shading Language - GLSL) bây giờ là ngôn ngữ đồ bóng tiêu chuẩn cho OpenGL. Tuy nhiên, vấn đề chính với Cg / HLSL / GLSL cho GPGPU là chúng vốn đã là ngôn ngữ đồ 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 đồ 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 - theo 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 đồ 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 biên 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 đồ 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). Tầng này đưa thêm các cấu trúc (construct) cấp cao hơn, giống như thành phần tương tự trong hệ 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 hiệu 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 hưởng tới tính ổn định và hiệu năng của các ứng dụng GPGPU.

### **1.2.6. Kỹ thuật và ứng dụng**

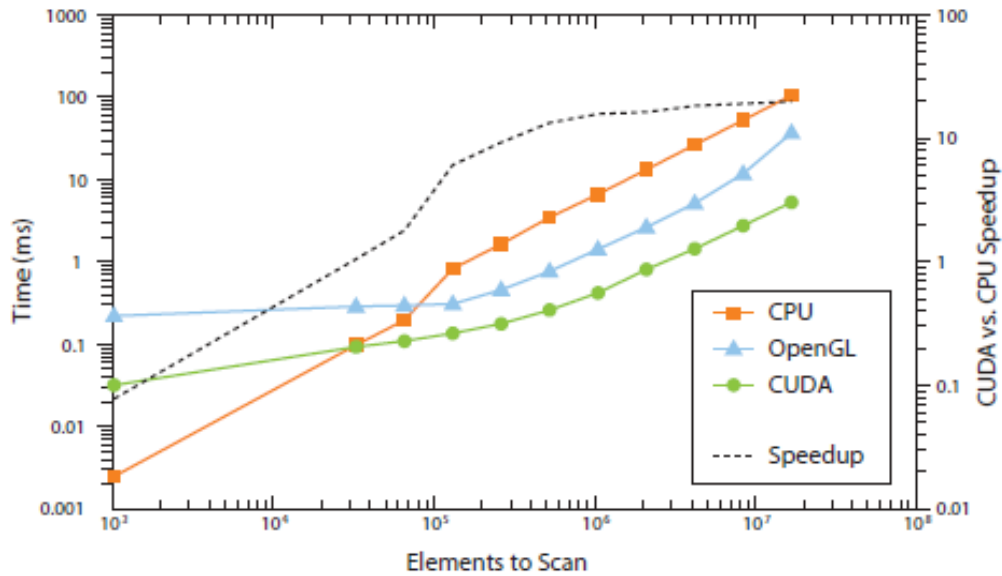
Bây giờ chúng ta khảo sát một số đặc tính tính toán quan trọng, thuật toán, và các ứng dụng tính toán GPU. Chúng tôi lần đầu tiên nêu bật bốn phép toán song song dữ liệu tập trung ở tính toán GPU: thực hiện phép toán tán xạ (scatter) / tập hợp (gather) bộ nhớ, ánh xạ một chức năng vào nhiều yếu tố song 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.

#### **1.2.6.1. 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 phẩy 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 vị 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 đồ 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 đọc 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 thừa số lên đến 20 và OpenGL bởi một thừa số lên đến 7.

### 1.2.6.2. 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 nó 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 Krüger [~15]) và tại các hiệu ứng trực quan cho các chèo trôi 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 trình đại 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 Krüger 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 thừa thớt có nhiều biến đổi và phức tạp hơn so với loại dày đặc đang là một lĩnh vực mở và hướng nghiên cứu tích cực, các nhà nghiên cứu mong có mã nguồn thừa thớt để kiểm chứng lợi ích tương tự hoặc lớn hơn từ tầng tính toán mới GPU.

### 1.2.6.3. **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*):** Như trong phần 2.3 đã nêu, 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 (như CUDA "Warps" sẽ mô tả trong chương 2). 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 đầu phẩy 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  $10 \times$  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 liên 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).

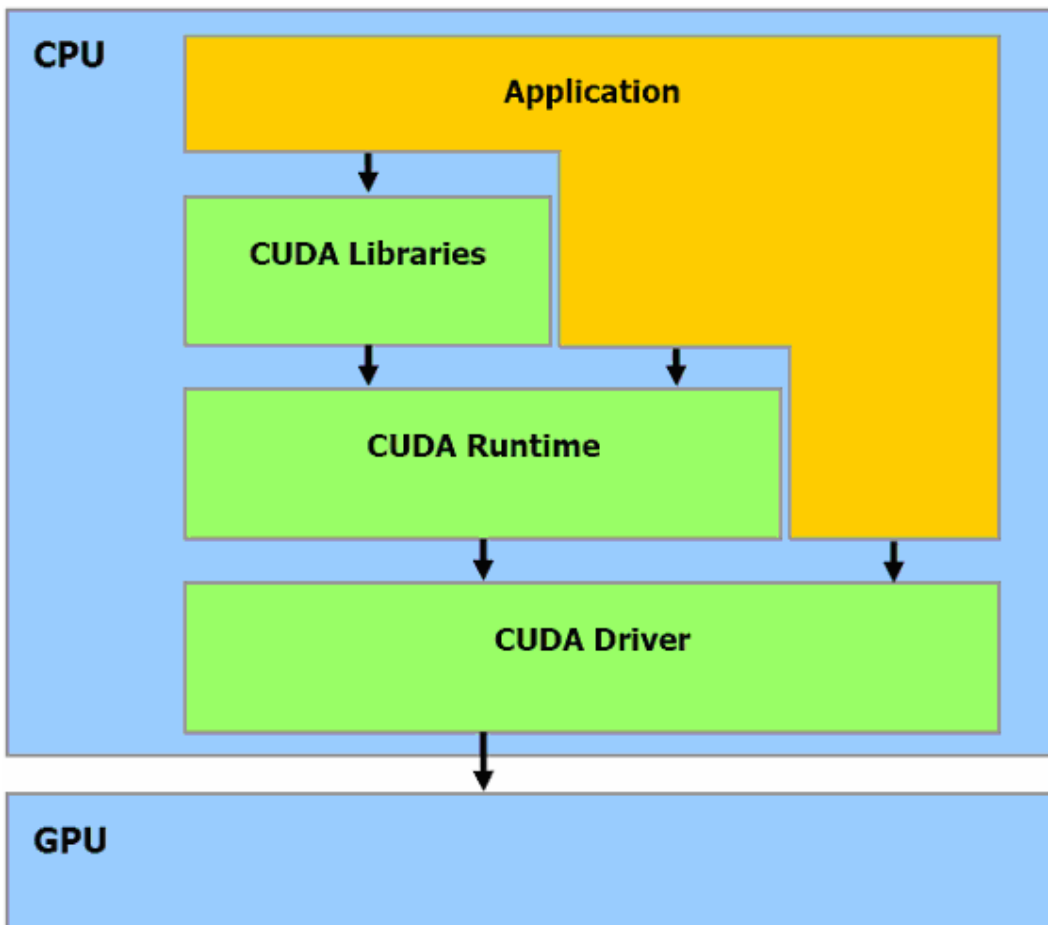
Kinh nghiệm cho thấy rằng khi các thuật toán và ứng dụng có thể làm theo các nguyên tắc thiết kế cho tính toán trên GPU - chẳng hạn như các giải pháp PDE, các gói đại số tuyến tính gói, và hệ thống cơ sở dữ liệu đã nói ở trên, và các trò chơi vật lý và ứng dụng động lực học phân tử có thể đạt được tốc độ gấp 10-100 lần so với các đoạn code CPU hoàn thiện, tối ưu.

## Chương 2. HỆ THỐNG CHƯƠNG TRÌNH DỊCH VÀ NGÔN NGỮ LẬP TRÌNH GPU

### 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.

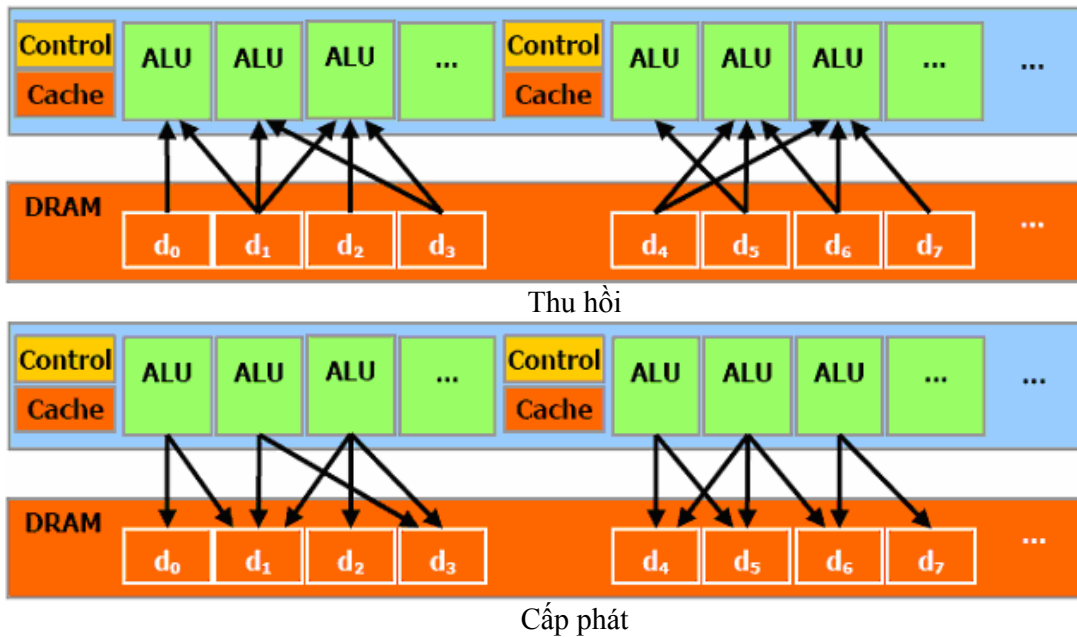
Bộ phần mềm CUDA bao gồm các lớp mô tả trong hình 13: driver cho phần cứng, API lập trình, môi trường thực thi; và hai thư viện toán học mức cao hơn của các hàm thường dùng, CUFFT và CUBLAS. Phần cứng được thiết kế để hỗ trợ driver hạng nhẹ và lớp môi trường thực thi, từ đó cho tốc độ cao.



Hình 13: Kiến trúc bộ phần mềm CUDA

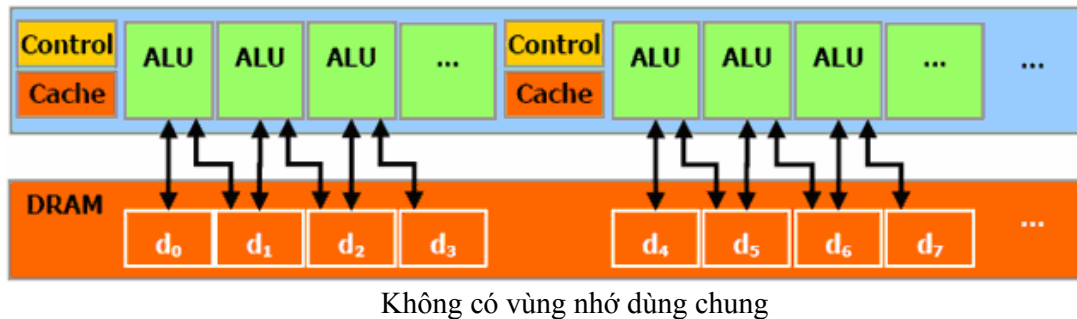
Thư viện lập trình CUDA bao gồm các hàm mở rộng của ngôn ngữ C. CUDA cung cấp cách đánh địa chỉ DRAM thường dùng như mô tả trong hình 14 cho việc lập

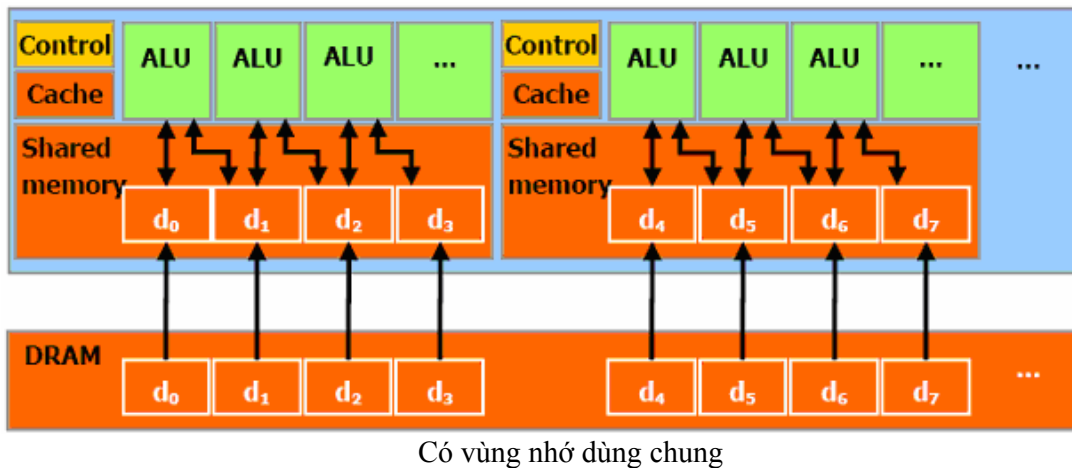
trình linh hoạt hơn, bao gồm cả thao tác cấp phát và thu hồi bộ nhớ. Từ góc độ lập trình, điều đó tương ứng với khả năng đọc và ghi dữ liệu tại bất kỳ địa chỉ nào trong DRAM, giống như CPU.



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

CUDA có đặc tính lưu trữ dữ liệu đệm song song và bộ nhớ chia sẻ 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 bằng thông truyền bộ nhớ DRAM.





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

## 2.2. Mô hình lập trình

### 2.2.1. 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. Nó 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 (off-loaded) 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 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 thiết bị 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.

### 2.2.2. Gom lô các luồng (Thread Batching)

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

#### 2.2.2.1. Khối luồng

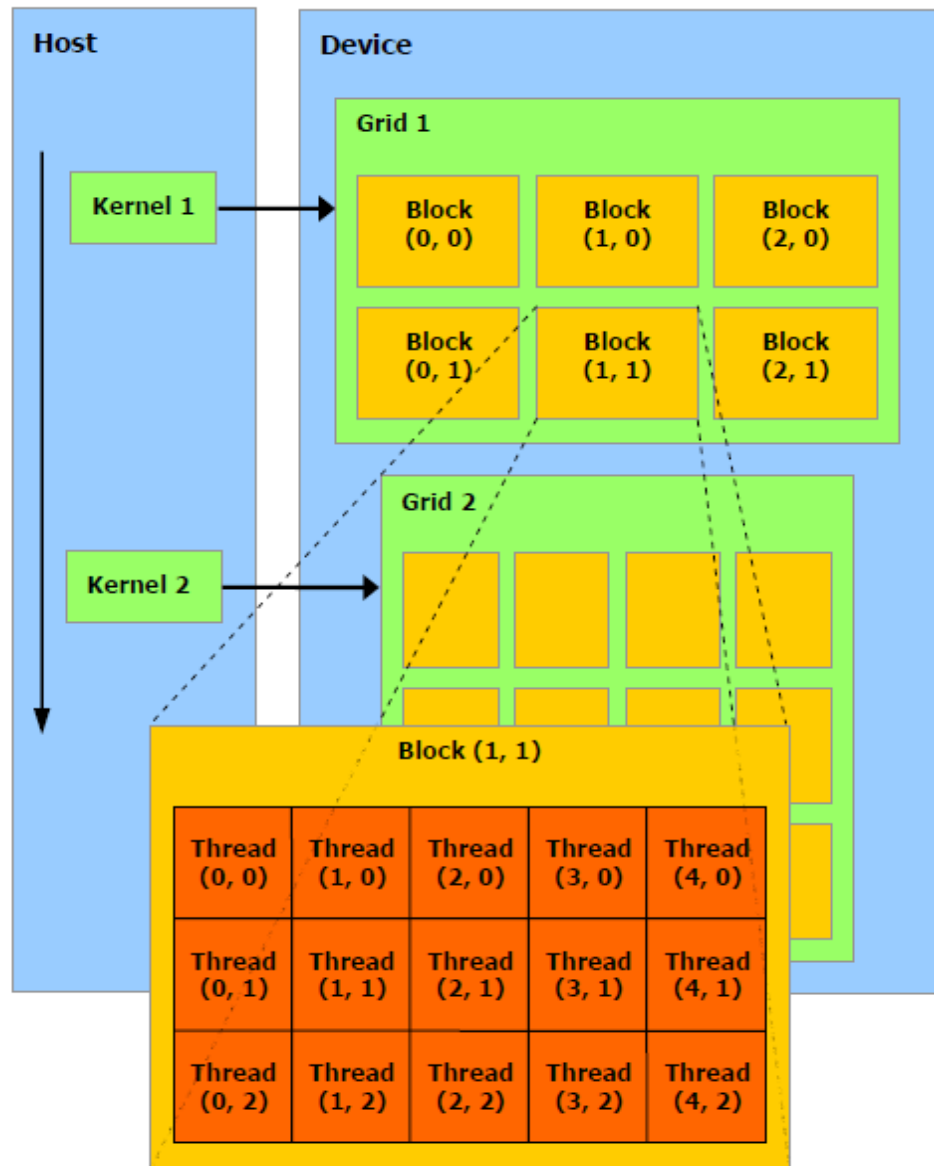
Một *khối luồng* là một tập các luồng, có thể đồng thời xử lý với nhau bằng cách dùng dữ liệu trong bộ nhớ dùng chung và thực thi đồng bộ để phối hợp truy cập bộ nhớ. Chính xác hơn, có thể xác định các điểm đồng bộ trong nhân, nơi các luồng trong khối sẽ dừng cho đến khi tất cả các luồng tới điểm đồng 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 2 hoặc 3 thành phần để thay thế. Đối với các khối kích thước 2 chiều ( $D_x, D_y$ ), thread ID của phần tử có chỉ số  $(x, y)$  là  $(x + y D_x)$  và cho một khối kích thước ba chiều ( $D_x, D_y, D_z$ ), thread ID của phần tử  $(x, y, z)$  là  $(x + y D_x + z D_x D_y)$ .

#### **2.2.2.2. Lưới các khối luồng (Grid of Thread Blocks)**

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 này 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 nó 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 *block ID*, một ứng dụng có thể xác định một lưới như một mảng 2 chiều với kích thước cố định và định danh mỗi khối sử dụng chỉ mục 2 thành phần. Với khối 2 chiều kích thước  $(D_x, D_y)$ , *block ID* của block  $(x,y)$  là  $(x + y D_x)$ .



Hình 16: Khối luồng

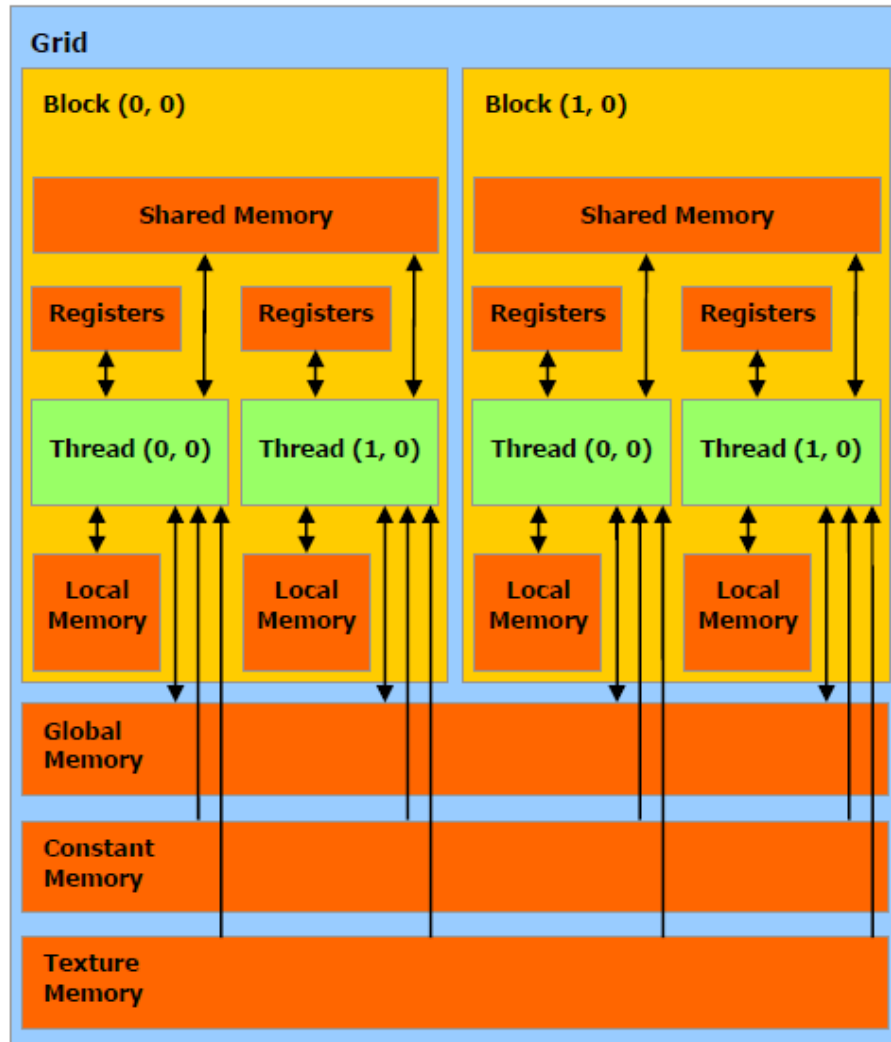
### 2.2.3. Mô hình bộ nhớ

Một luồng thực thi trên thiết bị chỉ truy cập vào DRAM của thiết bị và bộ nhớ trên chip qua các không gian nhớ sau, như mô tả trong hình 17:

- Đọc- ghi trên các thanh ghi của mỗi luồng
- Đọc-ghi bộ nhớ cục bộ mỗi luồng
- Đọc-ghi bộ nhớ dùng chung của mỗi khối.
- Đọc-ghi bộ nhớ toàn cục của mỗi lưới.
- Chỉ đọc bộ nhớ hằng số của mỗi lưới
- Chỉ đọc bộ nhớ kết cấu (texture) của mỗi lưới

Các cùng 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



Hình 17: Mô hình bộ nhớ

## 2.3. Thiết lập phần cứng

### 2.3.1. Tập các bộ đa xử lý SIMD với bộ nhớ dùng chung trên chip

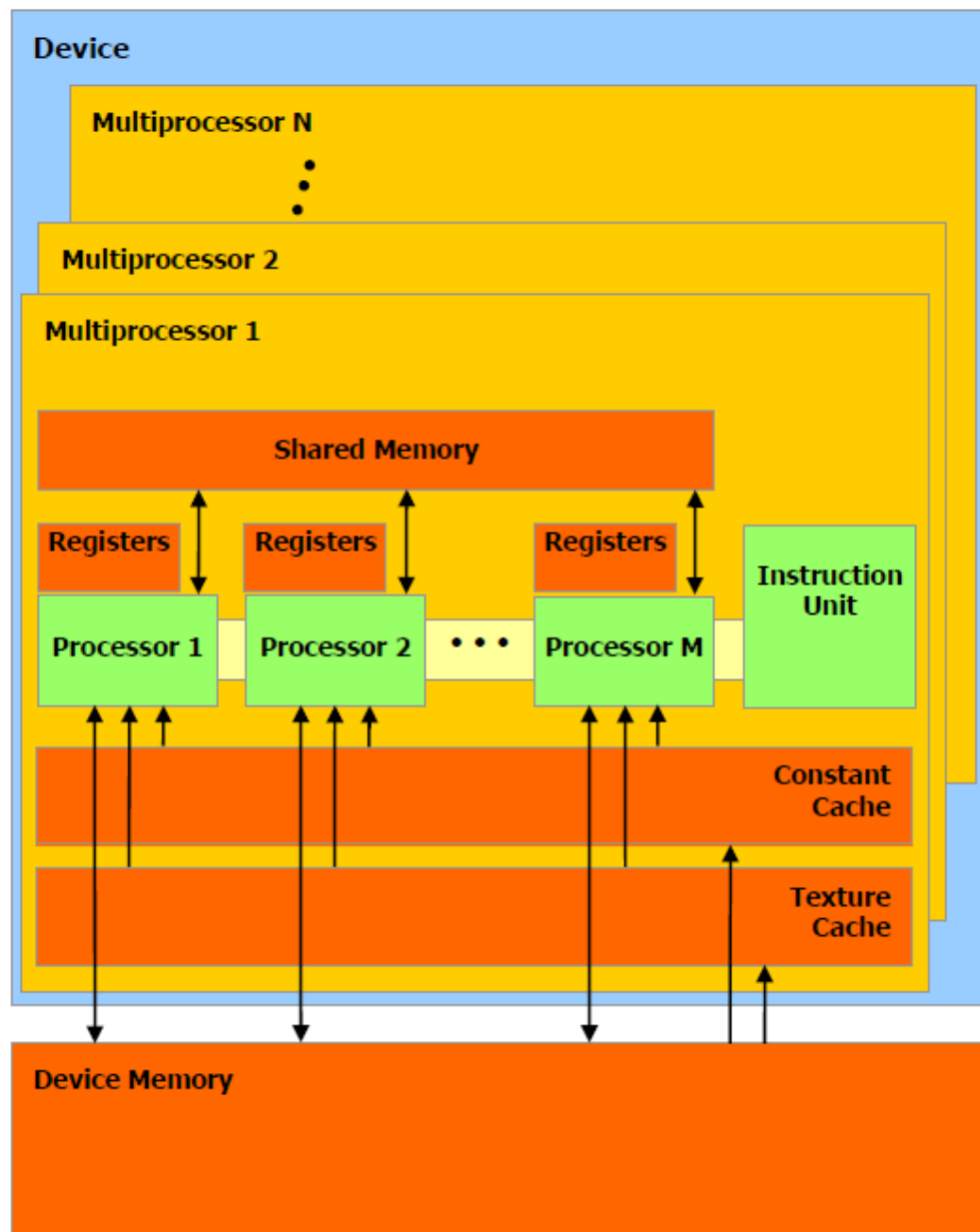
Thiết bị được cấu hình như một tập các bộ đa xử lý như mô tả trong hình 18. Mỗi bộ đa xử lý có một kiến trúc đơn lệnh, đa dữ liệu (SIMD): tại một chu kỳ đồng hồ cho trước, mỗi bộ xử lý của bộ đa xử lý thực thi cùng một lệnh, nhưng với dữ liệu khác nhau.

Mỗi bộ đa xử lý có bộ nhớ trên chip thuộc 4 loại sau:

- Một tập các *thanh ghi* cục bộ 32 bit cho mỗi bộ xử lý.
- Một vùng đệm dữ liệu song song hoặc *vùng nhớ dùng chung* được chia sẻ bởi tất cả các bộ xử lý và cài đặt bởi không gian bộ nhớ dùng chung.
- Một *vùng đệm hằng số* chỉ đọc được dùng chung bởi tất cả bộ xử lý và tăng tốc độ đọc từ không gian bộ nhớ hằng số, được cài đặt như một vùng chỉ đọc của bộ nhớ thiết bị.
- Một *vùng đệm kết cấu* chỉ đọc được dùng bởi tất cả các bộ xử lý và tăng tốc độ đọc từ không gian bộ nhớ kết cấu, được cài đặt như một vùng chỉ đọc của bộ nhớ thiết bị.

Không gian nhớ toàn cục và cục bộ, được cài đặt như một vùng đọc ghi trên bộ nhớ thiết bị và không có bộ đệm.

Mỗi bộ đa xử lý truy cập vùng đệm kết cấu thông qua *đơn vị kết cấu (texture unit)* thực thi nhiều chế độ đánh địa chỉ và lọc dữ liệu trong đã đề cập trong phần 2.2.3.



Hình 18: Mô hình phân cứng

### 2.3.2. Mô hình thực thi

Một lưới các khối luồng được thực thi trên thiết bị bằng cách thực thi một hoặc nhiều khối trên từng bộ đa xử lý sử dụng lát cắt thời gian. Mỗi khối được tách thành các nhóm SIMD của các luồng gọi là *warp*; mỗi warp có cùng số lượng luồng, gọi là kích thước warp, được thực thi bằng bộ đồng xử lý trong mô hình SIMD, *bộ lập lịch luồng* định kỳ chuyển từ warp này sang warp khác để tối đa mức độ sử dụng tài nguyên tính toán của bộ đa xử lý. *Half-warp* là nửa thứ nhất hoặc nửa thứ hai của warp.

Cách tách một khối thành các warp luôn giống nhau, mỗi warp bao gồm các luồng thực hiện liên tục, với id của luồng tăng dần, warp đầu tiên bao gồm thread 0. Phần 2.2.2.1 mô tả mối quan hệ giữa ID của luồng với chỉ số của luồng trong khối.

Một khối luồng được xử lý bằng chỉ một bộ đa xử lý, do vậy không gian nhớ dùng chung trong vùng nhớ dùng chung trên chip dẫn tới tốc độ truy cập bộ nhớ rất nhanh. Các thanh ghi của bộ đa xử lý được cấp phát giữa các luồng trong khối. Nếu số lượng thanh ghi sử dụng cho 1 luồng nhân với số lượng luồng lớn hơn tổng số thanh ghi trên bộ đa xử lý, khối không thể thực thi và nhân tương ứng không thể chạy được.

Một vài khối có thể thực hiện trên cùng một bộ đa xử lý đồng thời bằng cách cấp phát các thanh ghi của bộ đa xử lý và bộ nhớ dùng chung giữa các khối.

Thứ tự các warp trong một block không xác định, nhưng việc thực thi của chúng có thể đồng bộ, như mô tả trong phần 2.2.2.1, để phối hợp đồng thời truy cập bộ nhớ toàn cục và bộ nhớ chia sẻ.

Thứ tự các khối trong một lưới các khối luồng không xác định và không có cơ chế đồng bộ giữa các khối, do vậy luồng ở các khối khác nhau trong lưới không thể giao tiếp với nhau một cách an toàn qua vùng nhớ toàn cục trong quá trình thực thi lưới.

Nếu các lệnh không là nguyên tố thực hiện trong warp ghi vào cùng vị trí trong vùng nhớ toàn cục hoặc vùng nhớ chia sẻ cho nhiều hơn một luồng của warp đó, số lượng và thứ tự thực hiện các phép ghi tuần tự xảy ra tại vị trí đó diễn ra không xác định, nhưng một trong các lệnh ghi được đảm bảo thành công. Nếu lệnh là lệnh nguyên tố (xem phần 2.4.4.6) thực thi bởi warp đọc, thay đổi và ghi tới cùng một vị trí trong vùng nhớ toàn cục cho nhiều luồng của warp, từng thao tác đọc, thay đổi, ghi tới vị trí đó được nối tiếp nhau, nhưng thứ tự chúng diễn ra không xác định.

### **2.3.3. Khả năng tính toán**

Các tính năng của một thiết bị được thể hiện trên số hiệu phiên bản chính và số hiệu phụ đi kèm. Thiết bị với cùng một số phiên bản chính có cùng kiến trúc cốt lõi. GeForce 8 Series, Quadro FX 5600/4600, và Tesla là các giải pháp của năng lực tính toán 1.x (số hiệu phiên bản chính là 1).

Số hiệu phụ tương ứng với một sự cải tiến để gia tăng các kiến trúc lõi, có thể bao gồm cả tính năng mới. Các GeForce 8800 Series, Quadro FX 5600/4600, và Tesla là giải pháp được các tính năng lực 1.0 (nhỏ hiệu phụ là 0) và GeForce 8600 và 8.500 Series có khả năng tính toán 1.1.

Các thông số kỹ thuật của các khả năng tính toán được đưa ra trong Phụ lục A trong [99].

### 2.3.4. Đa thiết bị

Việc sử dụng nhiều GPU như các thiết bị CUDA bởi một ứng dụng chạy trên các hệ thống đa GPU chỉ đảm bảo hoạt động nếu các GPU này cùng loại. Tuy nhiên, nếu hệ thống trong chế độ SLI, chỉ một GPU có thể sử dụng như là thiết bị CUDA do tất cả GPU được giữ ở mức thấp nhất trong *stack driver*. SLI mode cần được tắt trong control panel để CUDA có thể kích hoạt từng GPU như là thiết bị riêng biệt.

### 2.3.5. Cơ chế chuyển đổi

GPU dành cho một số vùng nhớ DRAM cho cái gọi là *bề mặt chính* (*primary surface*), được sử dụng để làm tươi thiết bị hiển thị cho người dùng xem. Khi người dùng khởi tạo chế độ chuyển đổi của màn hình bằng cách thay đổi độ phân giải hoặc số bit của màn hình (sử dụng NVIDIA control panel hoặc Display control panel trên Windows), một lượng bộ nhớ cần cho thay đổi bề mặt chính. Ví dụ nếu người dùng thay đổi độ phân giải từ 1280x1024x32 bit thành 1600x1200x32 bit, hệ thống phải dành ra 7.68 MB hiển thị bề mặt chính thay vì 5.24 MB. (Ứng dụng đồ họa full-screen chạy với chế độ chống răng cưa có thể yêu cầu bộ nhớ hiển thị nhiều hơn nữa cho bề mặt chính). Trên Windows, các sự kiện khác có thể kích hoạt chuyển chế độ hiển thị như chạy ứng dụng DirectX full-screen, nhấn Alt-Tab để task chuyển khỏi ứng dụng DirectX full-screen, hoặc Ctrl+Alt+Del để khóa máy.

Nếu chuyển chế độ tăng dung lượng bộ nhớ cần thiết cho bề mặt chính, hệ thống cần lấy thêm bộ nhớ cung cấp cho ứng dụng CUDA, kết quả là gây đổ vỡ các ứng dụng.

## 2.4. Giao diện lập trình ứng dụng

### 2.4.1. Mở rộng cho 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. Nó gồm có :

- Một thiết lập tối thiểu của các thành phần mở rộng cho ngôn ngữ lập trình C, được miêu tả trong phần 2.4.2, cho phép người lập trình nhắm tới các phân chia mã nguồn chương trình cho việc xử lý trên thiết bị.
- Thư viện chạy được chia thành:
  - + *Thành phần chính* (*host component*), được miêu tả trong 2.4.5, 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 component*), miêu tả trong 2.4.4, được chạy trên các thiết bị và cung cấp các hàm riêng của thiết bị đó.

- + *Một thành phần chung (common component)*, miêu tả trong 2.4.3, cung cấp xây dựng trong kiểu vector và là một tập con thư viện chuẩn của C nó 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 là các chức năng được cung cấp bởi thành phần chạy chung.

## 2.4.2. Mở rộng ngôn ngữ

Mở rộng cho 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 .(Phần 2.4.2.1);
- Từ khóa phạm vi kiểu biến cho phép đặc tả vị trí bộ nhớ trên thiết bị của một biến (phần 2.4.2.2);
- Một chỉ thị mới để xác định cách nhân được thực hiện trên thiết bị từ phía host (phần 2.4.2.3)
- Bốn biến build-in để xác định chiều của lưới và khối, chỉ số khối và luồng (phần 2.3.2.4)

Với mỗi file nguồn chứa các phần mở rộng trên phải được biên dịch với CUDA bằng trình biên dịch **nvcc**, được miêu tả ngắn gọn trong 2.3.2.5. Những miêu tả chi tiết của **nvcc** có thể được tìm thấy trong các tài liệu khác.

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 chúng có thể không được nhận ra.

### 2.4.2.1. Từ khóa phạm vi kiểu hàm

#### 2.4.2.1.1. `__device__`

Khai báo `__device__` định nghĩa một hàm:

- Xử lý trên thiết bị
- Chỉ được gọi từ thiết bị

#### 2.4.2.1.2. `__global__`

Khai báo `__global__` định nghĩa một hàm như là một hạt nhân:

- Xử lý trên thiết bị
- Chỉ có thể triệu gọi được từ host

#### 2.4.2.1.3. `__host__`

Khai báo `__host__` là một hàm:

- Xử lý trên host
- Chỉ có thể triệu gọi được từ host.

Nó tương đương việc khai báo một hàm với chỉ xác định trong host hoặc khai báo nó bên ngoài của host, thiết bị hoặc khai báo toàn cục; trong một số trường hợp khác các hàm được kết hợp với nhau chỉ cho host.

Tuy nhiên việc các hàm hạn định trong host cũng có thể sử dụng kết hợp với các hàm hạn định trong thiết bị, trong một vài trường hợp chức năng kết hợp cho cả host và thiết bị.

#### **2.4.2.1.4. 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.
- Các hàm của `__device__` không thể lấy được địa chỉ của chúng; hàm trỏ tới các hàm `__global__` được hỗ trợ.
- `__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 như trong miêu tả phần 2.3.2.3.
- Gọi tới một hàm `__global__` là không đồng bộ, có nghĩa là nó trả về trước khi thiết bị hoàn thành xong xử lý của nó.
- Tham số của hàm toàn cục hiện đang được truyền qua bộ nhớ dùng chung với thiết bị và giới hạn độ lớn 256 byte.

#### **2.4.2.2. Từ khóa phạm vi kiểu biến**

##### **2.4.2.2.1. `__device__`**

Khai báo `__device__` định nghĩa biến chỉ có giới hạn trên thiết bị đó.

Nhiều nhất là một trong ba kiểu khai báo bên dưới có thể sử dụng cho các thiết bị khác để tiếp tục chỉ định không gian bộ nhớ mà biến thuộc. Nếu không ai trong chúng thể hiện, các biến :

- Tồn tại trong không gian bộ nhớ toàn cục
- Có vòng đời của một ứng dụng
- Truy nhập được từ tất cả các luồng bên trong lưới và host thông qua thư viện runtime

#### 2.4.2.2.2. `__constant__`

Khai báo `__constant__` có thể được dùng với khai báo `__device__` định nghĩa một biến:

- Tồn tại trong không gian bộ nhớ không đổi
- Có lifetime của một ứng dụng
- Truy nhập được từ tất cả các luồng bên trong lưới và host thông qua thư viện runtime

#### 2.4.2.2.3. `__shared__`

Biến chia sẻ lựa chọn sử dụng với các thiết bị khác, miêu tả một biến có :

- Tồn tại trong một không gian bộ nhớ chia sẻ của một luồng
- Có lifetime của một khối.
- Chỉ có thể truy nhập từ tất cả các chủ thể trong khối.

Có đầy đủ trình tự nhất quán của các biến chia sẻ trong phạm vi một luồng. Chỉ sau khi thực hiện một `syncthreads()` (trong phần 4.4.2) làm việc viết từ các luồng khác đảm bảo nhìn thấy được. Trình biên dịch không bị ràng buộc để tối ưu hóa những lần đọc ghi vào bộ nhớ chia sẻ miễn là những câu lệnh trước đó được đáp ứng.

Khi khai báo một biến trong bộ nhớ chia sẻ như một mảng mở rộng như :

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

Kích thước của mảng được xác định tại thời điểm khởi tạo (xem phần 4.2.3). Tất cả các biến nếu khai báo trong thời điểm này , bắt đầu tại cùng một địa chỉ trong bộ nhớ, do đó cách bố trí của các biến trong mảng đó phải được quản lý một cách rõ ràng thông qua offsets. Ví dụ, nếu muốn tương đương với:

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

Trong bộ nhớ chia sẻ động được tạo ta có thể khai báo và khởi tạo các mảng theo cách sau:

```
extern __shared__ char array[];  
__device__ void func() // Hàm chức năng toàn cục hoặc  
thiết bị  
{  
    short* array0 = (short*)array;  
    float* array1 = (float*)&array0[128];  
    int* array2 = (int*)&array1[64];  
}
```

#### 2.4.2.2.4. Các ràng buộc

Những hạn định là không cho phép vào thành phần `struct` và `union`, trên các thông số chính thức và trên các biến cục bộ trong một hàm thực thi trên host.

`__shared__` và `__constant__` không thể sử dụng trong việc kết hợp với các biến khác.

Các biến `__shared__` và `__constant__` ám chỉ lưu trữ tĩnh.

Các biến `__device__` và `__constant__` chỉ cho phép trong phạm vi file

Các biến `__constant__` không thể được gán từ thiết bị, chỉ từ các host lưu trữ.

Các biến `__shared__` không thể có một khởi tạo như bộ phận khai báo.

Một biến tự động khai báo trong mã thiết bị mà không cần phải khai báo trong một đăng ký cư chú chung nào. Tuy nhiên trong một số trường hợp, trình biên dịch có thể chọn để đặt nó trong bộ nhớ cục bộ. Đây thường là trường hợp cho các cấu trúc lớn hoặc mảng đó sẽ tiêu thụ không gian đăng ký quá nhiều, và mảng mà trình biên dịch không thể xác định rằng chúng được lập chỉ mục với số lượng không đổi. Kiểm tra các mã assembly *ptx* (thu được bằng cách biên dịch tùy chọn `-ptx` hoặc `-keep`) sẽ cho biết nếu một biến đã được đặt trong bộ nhớ cục bộ trong lần biên dịch đầu tiên nó sẽ được khai báo sử dụng thuộc tính nhớ `.local` và truy nhập sử dụng `ld.local` và `st.local`. Ngược lại, các lần biên dịch tiếp theo có thể quyết định cách khác nếu chúng tìm thấy nó tiêu thụ quá nhiều không gian thanh ghi cho mục đích cấu trúc.

Con trỏ trong code thực thi trên thiết bị được hỗ trợ miễn là trình biên dịch có thể giải quyết liệu có phải chúng chỉ tới một không gian bộ nhớ dùng chung hay không gian bộ nhớ toàn cục, nếu không chúng phải được hạn chế trỏ tới không gian bộ nhớ đã chỉ định hoặc khai báo trong không gian bộ nhớ toàn cục.

Truy nhập vào vùng nhớ mà một con trỏ trỏ tới bộ nhớ toàn cục hoặc bộ nhớ dùng chung trong code đó được thực thi trên host; hoặc tới bộ nhớ của host trong code là thực thi trên thiết bị dẫn tới một hành vi không xác định trước, thường xuyên nhất trong phân đoạn lỗi và khi kết thúc ứng dụng.

### 2.4.2.3. 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** (miêu tả trong 2.4.3.1.2) 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** (miêu tả trong 2.4.3.1.2) và xác định mục đích kích thước của mỗi khối, sao cho **Db.x\*Db.y\*Db.z** bằng số lượng các luồngs 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 như được đề cập đến trong phần 2.3.2.2.3, **Ns** là một đối số tùy chọn mặc định là 0.

Các đối số để cấu hình được ước lượng trước khi thực hiện hàm thực tế.

Một ví dụ cho việc khai báo hàm:

```
__global__ void Func(float* parameter);
```

Phải gọi giống như:

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

#### 2.4.2.4. Các biến Built-in

##### 2.4.2.4.1. gridDim

Đây là biến kiểu **dim3** (xem phần 2.4.3.1.2) và chứa các kích thước của lưới

##### 2.4.2.4.2. blockIdx

Đây là biến thuộc kiểu **uint3** (xem phần 2.4.3.1.1) chứa các chỉ số khối trong lưới

##### 2.4.2.4.3. blockDim

Biến này là loại **dim3** (xem phần 2.4.3.1.2) chứa kích thước của khối.

##### 2.4.2.4.4. threadIdx

Biến này thuộc loại **uint3** (xem phần 2.4.3.1.1) và chứa các chỉ số luồng trong khối

##### 2.4.2.4.5. Các hạn chế

- Nó không cho phép đưa ra địa chỉ của bất kỳ biến built-in nào
- Nó không cho phép gán giá trị cho bất kỳ biến built-in nào

#### 2.4.2.5. Biên dịch với NVCC

**nvcc** là một trình điều khiển trình biên dịch bằng việc đơn giản hóa quá trình biên dịch mã CUDA: Nó 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 code thiết bị từ code host và biên dịch code thiết bị sang dạng nhị phân hoặc các đối tượng *cubin*. Các code host sinh ra là đầu ra có thể là code C để được biên dịch bằng cách sử dụng một tool khác hoặc là code đối tượng trực tiếp bởi việc triệu gọi trình biên dịch host trong giai đoạn biên dịch trước đó.

Ứng dụng hoặc có thể bỏ qua các code host sinh ra và tải đối tượng *cubin* vào thiết bị và khởi động code thiết bị sử dụng trình điều khiển API của CUDA (xem phần 4.5.3), hoặc liên kết tới code 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 miêu tả trong phần 2.3.2.3 thành code cần thiết khởi động trong thời gian chạy CUDA để nạp và khởi động mỗi lần biên dịch hạt nhân (xem phần 2.3.5.2).

*Fron d end* của trình biên dịch xử lý các file 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.

Mô tả chi tiết của `nvcc` có thể được tìm thấy trong các tài liệu riêng biệt.

### 2.4.3. Thành phần chung trong thời gian chạy

Các thành phần chung trong thời gian chạy phổ biến có thể được sử dụng bởi cả hàm của host và thiết bị.

#### 2.4.3.1. Các loại vector built-in

##### 2.4.3.1.1. `char1`, `uchar1`, `char2`, `uchar2`, `char3`, `uchar3`, `char4`, `uchar4`, `short1`, `ushort1`, `short2`, `ushort2`, `short3`, `ushort3`, `short4`, `ushort4`, `int1`, `uint1`, `int2`, `uint2`, `int3`, `uint3`, `int4`, `uint4`, `long1`, `ulong1`, `long2`, `ulong2`, `long3`, `ulong3`, `long4`, `ulong4`, `float1`, `float2`, `float3`, `float4`

Đây là kiểu vector kế thừa từ các kiểu cơ bản là số nguyên và dấu phẩy động. Chúng có cấu trúc và có 1, 2, 3, hoặc 4 thành phần, các thành phần này có thể truy nhập thông qua các trường `x`, `y`, `z` và `w`, theo thứ tự định sẵn. Tất cả chúng có được sau gọi hàm khởi tạo `make_<type name>`; ví dụ:

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

Tạo ra một vector với kiểu `int2` với giá trị `(x,y)`.

##### 2.4.3.1.2. Kiểu `dim3`

Kiểu này là một kiểu vector integer cơ sở dựa trên `uint3` được sử dụng để chỉ định kích thước. Khi xác định một biến của kiểu `dim3` bất kỳ thành phần nào được để lại không chỉ rõ được khởi tạo là 1.

##### 2.4.3.1.3. Hàm toán học

Bảng B-1 trong [99] chứa danh sách đầy đủ thư viện chuẩn của C/C++ thực hiện các hàm toán học hiện đang được hỗ trợ, cùng với các lỗi tương ứng khi xử lý trên thiết bị.

Khi thực thi mã host, một hàm đưa ra sử dụng các cài đặt trong thời gian chạy của C nếu có sẵn.

#### 2.4.3.1.4. Hàm thời gian

```
clock_t clock();
```

Trả về giá trị của một bộ đếm luôn tăng theo mỗi chu kỳ đồng hồ. Lấy mẫu giá trị bộ đếm này ở đầu và cuối của một nhân, lấy sự khác biệt của hai mẫu, và ghi kết quả với từng luồng cung cấp một số đo cho mỗi luồng theo số chu kỳ đồng hồ cần thiết bởi thiết bị để hoàn thành xử lý luồng, nhưng đó không phải là số chu kỳ đồng hồ mà thiết bị thực sự dành ra để xử lý các chỉ thị lệnh trong luồng. Số trước lớn hơn số sau do các luồng có các nhát cắt thời gian.

#### 2.4.3.1.5. Kiểu kết cấu

CUDA hỗ trợ một tập hợp con các phần cứng tạo kết cấu mà GPU sử dụng cho đồ họa để truy cập vào bộ nhớ kết cấu. Đọc dữ liệu từ bộ nhớ kết cấu thay vì từ bộ nhớ toàn cục có thể có một số lợi ích hiệu suất như mô tả trong phần 2.5.4.

Nhân đọc bộ nhớ kết cấu bằng cách dùng các hàm thiết bị gọi là hàm đọc kết cấu (*texture fetches*), miêu tả trong phần 2.3.4.5. Tham số đầu tiên của một hàm đọc kết cấu đặc tả một đối tượng gọi là một tham chiếu kết cấu (*texture reference*).

Tham chiếu kết cấu định nghĩa phần bộ nhớ kết cấu được lấy ra. Nó phải được giới hạn thông qua các hàm runtime trên host (phần 0 và 2.3.5.3.7) cho tới một và vùng trong bộ nhớ, được gọi là kết cấu, trước khi nó có thể sử dụng bởi một nhân. Một vài tham chiếu kết cấu có thể bị ràng buộc vào cùng một kết cấu hoặc những kết cấu đè lên nhau trong bộ nhớ.

Tham chiếu kết cấu có một vài thuộc tính. Một trong số đó là chiều của nó sẽ xác định xem địa chỉ *kết cấu* ở đâu, liệu kết cấu được đánh địa chỉ trong một chiều sử dụng một tọa độ kết cấu, hay trong mảng hai chiều sử dụng hai tọa độ kết cấu. Mỗi phần tử của mảng gọi là *texel*, viết tắt cho “texture elements”.

Những thuộc tính khác định nghĩa kiểu dữ liệu đầu vào và đầu ra của hàm đọc kết cấu, cũng như cách tọa độ đầu vào được phiên dịch và luồng nào cần thực hiện.

#### 2.4.3.1.6. Khai báo tham chiếu kết cấu

Một số những thuộc tính của tham chiếu kết cấu là không thay đổi và phải được biết đến lúc thời gian biên dịch, chúng được xác định khi khai báo các tham chiếu kết cấu. Một tham chiếu kết cấu là khai báo file phạm vi như là một biến của loại kết cấu:

```
texture<Type, Dim, ReadMode> texRef; ở đây:
```

- *Type* xác định kiểu dữ liệu được trả về khi lấy kết cấu; được giới hạn trong số nguyên cơ bản và kiểu dấu phẩy động và vector 1, 2, 4 thành phần được định nghĩa trong phần 2.3.3.1.1;
- *Dim* xác định chiều của các tham chiếu kết cấu và bằng 1 hoặc 2. *Dim* là một đối số tùy chọn mặc định là 1.

- ReadMode bằng cudaReadModeNormalizedFloat hoặc cudaReadModeElementType; nếu nó cudaReadModeNormalizedFloat và loại là kiểu integer 16-bit hoặc 8-bit, giá trị thực trả về như là kiểu floating-point và đầy đủ dải địa chỉ của loại integer được ánh xạ vào
- [0.0, 1.0]; ví dụ, một unsigned 8-bit, phần tử kết cấu với giá trị 0xFF đọc như 1; nếu nó là cudaReadModeElementType, không thực hiện việc chuyển đổi; ReadMode là một đối số tùy chọn mặc định là cudaReadModeElementType.

#### 2.4.3.1.7. Thuộc tính tham chiếu kết cấu trong thời gian chạy

Các thuộc tính khác của một tham chiếu kết cấu có thể thay đổi ở thời gian chạy thông qua host runtime (phần 4.5.2.4 cho thời gian chạy API và phần 4.5.3.7 cho điều khiển API). Chúng xác định rõ tọa độ chuẩn hóa kết cấu hay không, chế độ địa chỉ, và bộ lọc kết cấu, như chi tiết bên dưới.

Theo mặc định, tham chiếu kết cấu sử dụng tọa độ dấu phẩy động trong khoảng  $[0, N]$  trong đó  $N$  là kích thước của kết cấu chiếu tương ứng với tọa độ. Ví dụ, một kết cấu mà là  $64 \times 32$  trong kích thước sẽ được tham chiếu với tọa độ trong khoảng  $[0, 63]$  và  $[0, 31]$  cho  $x$  và kích thước  $y$ , tương ứng. Chuẩn hóa các tọa độ kết cấu tạo ra các tọa độ sẽ được chỉ rõ trong phạm vi  $[0.0, 1.0]$  thay vì  $[0, N]$ , do đó giống như kết cấu  $64 \times 32$  cũng sẽ được xác định bởi tọa độ chuẩn hóa trong khoảng  $[0, 1)$  trong cả hai tọa độ  $x$  và  $y$ .

Chuẩn hóa tọa độ kết cấu là một tọa độ phù hợp với tự nhiên yêu cầu của một số ứng dụng. Nó thích hợp hơn cho tọa độ kết cấu cho việc triển khai của kích cỡ kết cấu.

Các chế độ địa chỉ xác định những cái sẽ xảy ra khi tọa độ trên phạm vi. Khi sử dụng không chuẩn hóa tọa độ kết cấu, tọa độ kết cấu bên ngoài khoảng  $[0, N)$ : các giá trị dưới 0 được đặt là 0 và giá trị lớn hơn hoặc bằng  $N$  được đặt là  $N-1$ . Giữ cố định là cách đánh địa chỉ mặc định chế độ khi sử dụng chuẩn hóa tọa độ kết cấu: Các giá trị dưới 0.0 hoặc cao hơn 1.0 đến phạm vi  $[0.0, 1.0)$ . Đối với chuẩn hóa tọa độ, “wrap” địa chỉ cũng có thể được chỉ định. Wrap địa chỉ được sử dụng khi kết cấu có chứa một chu kỳ tín hiệu. Nó chỉ sử dụng các phần phân đoạn của tọa độ kết cấu; ví dụ, 1.25 tương đương với 0.25 và -1.25 được xem như là 0.75

Tuyến tính lọc kết cấu có thể được thực hiện chỉ cho kết cấu được cấu hình trả về dữ liệu dấu phẩy động. Nó thực hiện phép nội suy chính xác thấp giữ các điểm gần nhau texels. Khi kích hoạt, texels xung quanh một truy cập vị trí kết cấu là đọc và trả về giá trị. Phép nội suy tuyến tính đơn giản được thực hiện cho những kết cấu và phép nội suy song tuyến tính một chiều được thực hiện cho hai chiều.

#### 2.4.3.1.8. Tạo kết cấu từ bộ nhớ tuyến tính so sánh với mảng CUDA

Một kết cấu có thể ở bất kỳ vùng nào trong bộ nhớ tuyến tính hoặc một mảng CUDA (xem phần 2.3.5.1.2). Phân bổ các kết cấu trong bộ nhớ tuyến tính:

- Số chiều chỉ có thể bằng 1
- Không hỗ trợ lọc kết cấu
- Chỉ có thể đánh địa chỉ bằng cách tọa độ kết cấu số nguyên không chuẩn hóa.
- Không hỗ trợ các cơ chế đánh địa chỉ khác nhau: truy cập vào kết cấu ngoài vùng nhớ trả về 0.

Phần cứng áp đặt các yêu cầu liên kết kết cấu dựa trên địa chỉ. Để trừu tượng hóa các yêu cầu liên kết này từ phía người phát triển, những hàm để ràng buộc các tham chiếu kết cấu vào bộ nhớ thiết bị trả về một byte offset phải được áp dụng cho *hàm đọc kết cấu* để đọc từ bộ nhớ mong muốn. Những con trỏ cơ sở trả về từ các thủ tục phân phối của CUDA phải tuân theo các ràng buộc liên kết đó, do đó ứng dụng có thể tránh offsets hoàn toàn bằng cách gửi con trỏ được cấp phát cho `cudaBindTexture() / cuTexRefSetAddress()`.

## 2.4.4. Thành phần thiết bị thời gian chạy

Thành phần thiết bị thời gian thực có thể chỉ được sử dụng trong các hàm thiết bị.

### 2.4.4.1. Các hàm toán học

Đối với số hàm của bảng B-1, một phiên bản ít chính xác hơn, nhưng nhanh hơn phiên bản tồn tại trong thành phần thiết bị thời gian thực; nó cùng tên với tiền tố bắt đầu bằng `__` (như `__sin(x)`). Các hàm này thực chất được liệt kê trong bảng B-2, cùng những ranh giới lỗi tương ứng.

Trình biên dịch này có một tùy chọn (`-use_fast_math`) để bắt buộc mọi hàm tới biên dịch tạo bản sao ít chính xác hơn nếu nó tồn tại.

### 2.4.4.2. Hàm đồng bộ

```
void __syncthreads();
```

Đồng bộ hóa tất cả các luồng trong một khối. Sau khi tất cả các luồng đã đạt đến điểm này, thực hiện tiếp tục lại bình thường.

`__syncthreads()` được sử dụng để phối hợp giao tiếp giữa các luồng của cùng một block. Khi một số luồng bên trong một block truy nhập cùng một địa chỉ được chia sẻ hoặc bộ nhớ toàn cục, có khả năng read-after-write, write-after-read, hoặc write-after-write một số nguy hiểm cho truy nhập bộ nhớ. Những nguy hiểm dữ liệu có thể được tránh bởi việc đồng bộ hóa các luồng giữa các truy cập.

`__syncthreads()` cho phép trong mã có điều kiện nhưng chỉ khi có điều kiện đánh giá giống nhau trên toàn bộ luồng của khối, nếu không thực hiện mã hợp lý có khả năng treo hoặc tạo ra các ngoại lệ không mong muốn.

### 2.4.4.3. Các hàm chuyển đổi kiểu

Hậu tố trong các hàm dưới đây mô tả các kiểu làm tròn được nêu trong IEEE-754:

- Rn là làm tròn tới số chẵn gần nhất (round-to-nearest-even)
- Rz là làm tròn về 0 (round-towards-zero)
- Ru là làm tròn lên (round-up) đến vô cùng dương
- Rd là làm tròn xuống (round-down) đến vô cùng âm

```
int __float2int_[rn,rz,ru,rd](float);
```

Chuyển đổi các tham số dấu phẩy động thành một số nguyên, sử dụng thiết lập chế độ làm tròn.

```
unsigned int __float2uint_[rn,rz,ru,rd](float);
```

Chuyển đổi các tham số dấu phẩy động thành một unsigned integer, sử dụng thiết lập chế độ làm tròn

```
float __int2float_[rn,rz,ru,rd](int);
```

Chuyển đổi các đối số nguyên thành dấu phẩy động, sử dụng thiết lập chế độ làm tròn.

```
float __uint2float_[rn,rz,ru,rd](unsigned int);
```

Chuyển đổi các đối số integer unsigned thành dấu phẩy động, sử dụng thiết lập chế độ làm tròn.

#### **2.4.4.4. Các hàm ép kiểu**

```
float __int_as_float(int);
```

Thực hiện kiểu dấu phẩy động trên các đối số nguyên, để trả về giá trị không thay đổi. Chẳng hạn: `__int_as_float(0xC0000000)` bằng `-2`

```
int __float_as_int(float);
```

Thực hiện một kiểu số nguyên cast trên kiểu dấu phẩy động, trả về giá trị không thay đổi. Chẳng hạn, `__float_as_int(1.0f)` bằng `0x3f800000`.

#### **2.4.4.5. Các hàm kết cấu**

##### **2.4.4.5.1. Tạo kết cấu từ bộ nhớ thiết bị**

Khi tạo kết cấu từ bộ nhớ thiết bị, kết cấu được truy cập với họ các hàm `tex1Dfetch()`, ví dụ:

```
template<class Type>
Type tex1Dfetch (
    texture<Type, 1, cudaReadModeElementType> texRef,
    int x);
```

```
float tex1Dfetch (
```

```
texture<unsigned char, 1, cudaReadModeNormalizedFloat>
texRef, int x);
```

```
float tex1Dfetch (
    texture<signed char, 1, cudaReadModeNormalizedFloat>
texRef, int x);
```

```
float tex1Dfetch (
    texture<unsigned short, 1, cudaReadModeNormalizedFloat>
texRef, int x);
```

```
float tex1Dfetch (
    texture<signed short, 1, cudaReadModeNormalizedFloat>
texRef, int x);
```

Những hàm này lấy các vùng bộ nhớ tuyến tính gắn cho tham chiếu kết cấu là *texRef* sử dụng tọa độ kết cấu *x*. Không có cơ chế lọc kết cấu hay đánh địa chỉ nào được hỗ trợ. Đối với các loại số nguyên, các chức năng này có thể tùy chọn cho số dấu phẩy động 32-bit.

Bên cạnh hàm hiển thị ở trên 2- và 4-tuples được hỗ trợ. Ví dụ:

```
float4 tex1Dfetch (
    texture<uchar4, 1, cudaReadModeNormalizedFloat> texRef,
int x);
```

Lấy bộ nhớ tuyến tính gắn cho tham chiếu kết cấu **texRef** sử dụng tọa độ *x* của kết cấu

#### 2.4.4.5.2. Tạo kết cấu từ mảng CUDA

Khi tạo kết cấu từ mảng CUDA, kết cấu được truy cập bởi `tex1D()` hay `tex2D()`

```
template<class Type, enum cudaTextureReadMode readMode>
Type tex1D(texture<Type, 1, readMode> texRef, float x);
```

```
template<class Type, enum cudaTextureReadMode readMode>
Type tex2D(texture<Type, 2, readMode> texRef, float x,
float y);
```

Các hàm trên lấy ra mảng CUDA gắn vào tham chiếu kết cấu **texRef** bằng cách dùng tọa độ kết cấu *x* và *y*. Tổ hợp của các thuộc tính không biến đổi (thời gian dịch) và biến đổi (thời gian chạy) của tham chiếu kết cấu xác định cách các tọa độ được

phiên dịch, luồng nào sẽ xuất hiện trong quá trình lấy kết cấu, và giá trị trả về được giao cho quá trình lấy kết cấu.

#### **2.4.4.6. Hàm nguyên tố**

Các hàm nguyên tố chỉ có cho các thiết bị phục vụ tính toán. Chúng được liệt kê ở phục lục C của [99] .

Hàm nguyên tố thực thi thao tác nguyên tố đọc- thay đổi- ghi (read-modify-write) trên các từ 32-bit có trong bộ nhớ toàn cục. Ví dụ `atomicAdd()` đọc một từ 32-bit trong một vùng nhớ của bộ nhớ toàn cục, cộng thêm 1 số nguyên cho nó, và ghi kết quả trả về vào cùng địa chỉ đọc ra. Thao tác trên là nguyên tố trong ngữ cảnh nó được đảm bảo để thực thi mà không có sự can thiệp từ luồng khác. Nói cách khác, không luồng nào khác có thể truy cập vào địa chỉ đó cho tới khi thao tác trên được hoàn thành.

Các hàm nguyên tố chỉ làm việc với số nguyên 32-bit có dấu và không dấu.

## **2.5. Hướng dẫn hiệu năng**

### **2.5.1. Hiệu năng lệnh**

Để xử lý một lệnh cho một warp các luồng, bộ đa xử lý cần thực hiện:

- Đọc toán hạng lệnh cho mỗi luồng của warp,
- Thực hiện lệnh
- Ghi kết quả của mỗi luồng

Do vậy, thông lượng xử lý lệnh phụ thuộc vào thông lượng lệnh thuần túy, cộng với độ trễ và băng thông của bộ nhớ. Nó được tối đa nếu:

- Tối thiểu việc sử dụng lệnh với thông lượng thấp
- Tối đa việc sử dụng băng thông của tất cả các loại bộ nhớ
- Cho phép các bộ lập lịch luồng có thể chùng các thao tác bộ nhớ với các thao tác tính toán toán học tối đa có thể, điều này yêu cầu:
  - + Chương trình được thực hiện bởi các luồng có cường độ số học cao, có nghĩa là số lượng lớn phép toán số học trên phép toán bộ nhớ;
  - + Có nhiều luồng có thể chạy đồng thời.

#### **2.5.1.1. Thông lượng lệnh**

##### **2.5.1.1.1. Các lệnh số học**

Để phát ra một lệnh của warp, bộ đa xử lý mất:

- 4 chu kỳ đồng hồ cho phép toán cộng, nhân, cộng-nhân dấu phẩy động, cộng số nguyên, phép dịch bit, so sánh, lớn nhất, nhỏ nhất, ép kiểu.

- 16 chu kỳ đồng hồ cho đối ứng, căn bậc hai, `__log(x)` (xem Bảng B-2 của [99]).

Phép nhân số nguyên 32 bit hết 16 chu kỳ đồng hồ, nhưng `__mul24` và `__umul24` (phụ lục B của [99]) cung cấp phép nhân có dấu và không dấu số nguyên 24 bit trong 4 chu kỳ đồng hồ. Tuy nhiên trong kiến trúc tương lai, `__[u]mul24` sẽ chậm hơn phép nhân số nguyên 32 bit, do đó nên cung cấp hai nhân, một sử dụng `__[u]mul24` và một sử dụng phép nhân số nguyên 32 bit, được gọi một cách thích hợp bởi ứng dụng. Phép chia số nguyên và phép lấy số dư chiếm nhiều thời gian và nên tránh nếu có thể thay thế bởi toán tử dịch bit. Nếu  $n$  là lũy thừa của 2,  $(i/n)$  tương đương với  $(i >> \log_2(n))$  và  $(i \% n)$  tương đương với  $(i \& (n-1))$ ; chương trình dịch sẽ thực hiện các chuyển đổi này nếu  $n$  là chữ.

Các chức năng khác chiếm nhiều chu kỳ đồng hồ hơn, và chúng được thực hiện bằng cách thực hiện nhiều lệnh.

Phép căn bậc hai dấu phẩy động được cài đặt bằng phép lấy căn bậc 2 đối ứng, do đó mất ít nhất 32 chu kỳ đồng hồ cho warp.

Phép chia dấu phẩy động mất 36 chu kỳ đồng hồ, nhưng `__fdivdef(x, y)` cung cấp một bản nhanh hơn với 20 chu kỳ đồng hồ.

`__sin(x)`, `__cos(x)`, `__exp(x)` mất 32 chu kỳ đồng hồ.

Nhiều khi chương trình dịch phải thêm lệnh đổi kiểu, làm tăng một số chu kỳ đồng hồ:

- Các phép toán trên `char` or `short` mà các toán hạng cần đổi kiểu về `int`.
- Các hằng số dấu phẩy động độ chính xác kép `double` được sử dụng như đầu vào của các phép toán dấu phẩy động độ chính xác đơn.
- Các biến dấu phẩy động độ chính xác đơn sử dụng các tham số đầu vào như là độ chính xác đơn của các hàm toán học.
- Hai trường hợp cuối có thể tránh bằng cách:
  - các hằng số dấu phẩy động độ chính xác đơn, xác định bởi biến có hậu tố `f` như `3.141592653589793f`, `1.0f`, `0.5f`.
  - Phiên bản toán học độ chính xác đơn, với hậu tố `f` ở đầu như `sinf()`, `logf()`, `expf()`.

Cho code có độ chính xác đơn, nên sử dụng các loại biến `float` và các hàm toán học độ chính xác đơn. Khi dịch cho các thiết bị mà không có hỗ trợ các phép toán dấu phẩy động độ chính xác đôi `double`, chẳng hạn như các thiết bị tính toán thể hệ `1.x`, biến kiểu `double` bị ép kiểu thành `float` như mặc định và các hàm toán học độ chính

xác đôi `double` được ánh xạ tới các phép toán độ chính xác đơn tương đương. Tuy nhiên các thiết bị tương lai sẽ hỗ trợ độ chính xác đôi `double`, các hàm này sẽ ánh xạ tới việc thực hiện độ chính xác đôi `double`.

#### 2.5.1.1.2. Các lệnh điều khiển

Các lệnh điều khiển (`if`, `switch`, `do`, `for`, `while`) có thể ảnh hưởng lớn đến thông lượng lệnh bởi nó làm các luồng trong cùng warp phân ra, có nghĩa là theo các đường thực hiện (execution path) khác nhau. Nếu điều này xảy ra, các đường thực hiện khác nhau được thực hiện nối tiếp, tăng tổng số lệnh thực hiện cho warp. Khi tất cả các cách thực hiện hoàn thành, các luồng hội tụ lại về cùng 1 đường thực hiện.

Để đạt được hiệu quả tốt nhất trong các trường hợp luồng điều khiển phụ thuộc vào `thread ID`, điều kiện điều khiển phải được viết sao cho tối thiểu số lượng phân nhánh warp. Điều này hoàn toàn có thể bởi việc phân tán các warp trên các khối được xác định trong phần 3.2 của chương này. Một ví dụ nhỏ là khi điều kiện điều khiển chỉ phụ thuộc vào (`threadIdx / WSIZE`) với `WSIZE` là kích thước warp. Trong trường hợp này không warp nào phân nhánh do đó điều kiện điều khiển là hoàn toàn liên kết với warp.

Đôi khi trình biên dịch có thể unroll vòng lặp hoặc nó tối ưu hóa nếu toán tử `if` hoặc `switch` bằng cách sử dụng dự đoán nhánh thay thế, như mô tả chi tiết phía dưới. Trong các trường hợp này, không warp nào có lệch. Khi sử dụng dự đoán nhánh không lệnh nào mà sự thực thi của lệnh phụ thuộc vào điều kiện điều khiển bị bỏ qua. Thay vào đó, mỗi lệnh liên kết với một mã điều kiện mỗi luồng hoặc chắc chắn được thiết lập là `true` hoặc `false` dựa trên điều kiện điều khiển và mặc dù mỗi lệnh điều có lập lịch để thực thi, chỉ các lệnh với predicate là `true` mới được thực hiện thực sự. Các lệnh với predicate là `false` không ghi kết quả, và không định địa chỉ hoặc đọc toán hạng.

Trình biên dịch sẽ thay thế một lệnh nhánh với một lệnh predicated chỉ nếu số lượng lệnh điều khiển bởi điều kiện nhánh nhỏ hơn hoặc bằng một ngưỡng nào đó: Nếu trình biên dịch xác định rằng điều kiện có khả năng sinh nhiều phân nhánh warp, ngưỡng là 7, ngược lại là 4.

#### 2.5.1.1.3. Các lệnh bộ nhớ

Các lệnh bộ nhớ bao gồm các lệnh đọc và ghi tới vùng nhớ chia sẻ hoặc vùng nhớ toàn cục. Bộ đa xử lý mất 4 chu kỳ đồng hồ để đưa ra một lệnh bộ nhớ cho warp. Khi truy cập bộ nhớ toàn cục, thêm vào đó sẽ mất độ trễ là 400 tới 600 chu kỳ đồng hồ.

Ví dụ phép gán trong đoạn code sau:

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

Phải mất 4 chu kỳ đồng hồ để đưa ra một lệnh đọc từ vùng nhớ toàn cục, 4 chu kỳ đồng hồ để đưa ra một lệnh viết vào bộ nhớ dùng chung, nhưng trên 400 tới 600 chu kỳ đồng hồ để đọc một biến float từ bộ nhớ toàn cục.

Độ trễ bộ nhớ toàn cục nhiều đến mức có thể ảnh hưởng bởi bộ lập lịch luồng nếu có các lệnh số học không phụ thuộc có thể ban hành trong khi chờ truy cập bộ nhớ kết thúc.

#### 2.5.1.1.4. Lệnh đồng bộ

`__syncthreads` mất 4 chu kỳ đồng hồ để gán cho một warp nếu không luồng nào phải đợi luồng nào.

#### 2.5.1.2. Bảng thông bộ nhớ

##### 2.5.1.2.1. Bộ nhớ toàn cục

Không gian nhớ toàn cục không được lưu vào bộ nhớ đệm, vì thế điều quan trọng là truy xuất đúng cách để có được băng thông tối đa, đặc biệt là chi phí cho việc truy cập bộ nhớ thiết bị.

Đầu tiên, thiết bị có thể đọc cùng lúc 32, 64 hoặc 128 bit cùng lúc từ bộ nhớ toàn cục vào thanh ghi với 1 câu lệnh. Ví dụ:

```
__device__ type device[32];
type data = device[tid];
```

Biên dịch đoạn trên thành lệnh máy, `type` phải có giá trị bằng biểu thức `sizeof(type)`, thường bằng 4,8,16 và các biến kiểu `type` phải căn 2,8,16 bytes (vì có 2,3 hoặc 4 bit có nghĩa tối thiểu của địa chỉ bằng zero)

Việc xếp bộ nhớ của các biến như vậy được làm tự động cho các kiểu có sẵn có trong phần 4.3.1.1 như **float2** hoặc **float4**.

Với các kiểu cấu trúc, kích thước và xếp bộ nhớ có thể được thi hành bởi trình biên dịch sử dụng những chỉ thị cụ thể như `__align__(8)` hoặc `__align__(16)`, ví dụ:

```
struct __align__(8) {
    float a;
    float b;
};
```

Hoặc

```
struct __align__(16) {
    float a;
    float b;
    float c;
    float d;
};
```

Với những cấu trúc lớn hơn 16 bytes, trình biên dịch tạo ra vài lệnh “ nạp”. Để đảm bảo số câu lệnh được tạo ra là ít nhất, các cấu trúc nên được định nghĩa với chỉ thị `__align__(16)`, ví dụ:

```
struct __align__(16) {
    float a;
    float b;
    float c;
    float d;
    float e;
};
```

Cấu trúc trên sẽ được biên dịch thành 2 lệnh máy nạp có độ dài 128 bit thay vì 5 lệnh máy nạp dài 32 bit

Thứ 2, địa chỉ bộ nhớ toàn cục được truy xuất đồng thời bởi từng luồng trong suốt việc thi hành của 1 lệnh máy đọc hoặc ghi nên được sắp xếp để việc truy cập bộ nhớ có thể kết hợp thành việc truy xuất 1 vùng nhớ liên tục duy nhất

Chính xác hơn, trong mỗi half-warp, luồng số N trong half-warp nên truy cập vào địa chỉ

```
HalfWarpBaseAddress + N
```

Với `HalfWarpBaseAddress` là kiểu con trỏ `type*` tuân theo cách dàn bộ nhớ như đã thảo luận ở trên. Hơn nữa, `HalfWarpBaseAddress` nên được cấp vùng nhớ theo cách `16*sizeof(type)` byte; nói cách khác, nó nên có số bit có nghĩa tối thiểu `log2(16*sizeof(type))` bằng zero. Bất kỳ địa chỉ `BaseAddress` của 1 biến thường trú trong bộ nhớ toàn cục hoặc được trả lại bằng 1 trong các cách cấp phát bộ nhớ được nhắc đến trong D.3 hoặc E.6 luôn được đưa vào vùng nhớ ít nhất 256 bytes, vì thế để thỏa mãn rằng buộc dàn xếp bộ nhớ, `HalfWarpBaseAddress` nên là bội của `16*sizeof(type)`.

Chú ý rằng nếu 1 half-warp thỏa mãn tất cả yêu cầu bên trên, các truy xuất bộ nhớ của từng luồng luôn liên tục với nhau mặc dù 1 vài luồng của half-warp không thực sự truy xuất bộ nhớ

Nên tuân thủ các yêu cầu về gắn kết của toàn bộ warp hơn chỉ với các half-warp riêng rẽ vì các thiết bị trong tương lai sẽ cần điều đó cho việc kết tập

1 cách truy xuất bộ nhớ toàn cục là khi mỗi luồng của luồng có ID là `tid` truy cập 1 phần tử của mảng được cấp phát tại địa chỉ `BaseAddress` của kiểu `type*` sử dụng địa chỉ sau:

```
BaseAddress + tid
```

Để có được việc truy xuất kết tập, type phải tuân theo kích thước và yêu cầu cấp phát bộ nhớ như đã thảo luận ở trên. Đặc biệt, điều đó nghĩa là nếu type là 1 cấu trúc lớn hơn 16 byte, nó nên được chia nhỏ thành vài cấu trúc khác phù hợp với các yêu cầu đó và dữ liệu nên được phân chia trong bộ nhớ thành danh sách của vài mảng của cấu trúc đó thay vì 1 mảng duy nhất của kiểu **type\***

Một cách truy cập bộ nhớ toàn cục phổ biến khác là khi mỗi luồng có chỉ số (tx,ty) truy cập 1 phần tử của mảng 2 chiều đặt tại địa chỉ BaseAddress của kiểu **type\*** và chiều rộng width sử dụng địa chỉ sau:

```
BaseAddress + width * ty + tx
```

Trong trường hợp đó, việc truy xuất bộ nhớ có thể kết tập cho tất cả half-warp của khối luồng nếu:

- + chiều rộng của khối luồng là bội số của kích thước của warp.
- + chiều rộng phải là bội số của 16

Đặc biệt, điều đó có nghĩa 1 mảng có chiều rộng không phải là bội số của 16 sẽ được truy xuất hiệu quả hơn nếu nó thực tế được cấp phát với chiều rộng được làm tròn lên thành bội số của 16 và các hàng của nó cũng được xếp như vậy. Các hàm `cuMemAllocPitch()` và `cudaMallocPitch()` và các hàm sao chép bộ nhớ có liên quan được mô tả trong D.3 và E.6 cho phép người phát triển viết các dòng lệnh không phụ thuộc vào phần cứng để cấp phát các mảng thỏa mãn các điều kiện đó

#### **2.5.1.2.2. Bộ nhớ hằng số**

Không gian bộ nhớ hằng số được lưu vùng đệm, vì vậy việc đọc từ một bộ nhớ hằng số mất thời gian bằng một lần đọc từ thiết bị nhớ chỉ trong trường hợp không có trong cache, còn trường hợp còn lại chỉ bằng một lần đọc trong vùng đệm hằng số.

Đối với tất cả luồng của half-warp, việc đọc từ vùng đệm hằng số nhanh như là việc đọc từ thanh ghi miễn là tất cả các luồng đọc cùng địa chỉ. Giá của việc đọc từ vùng nhớ hằng số gần như tỷ lệ với số địa chỉ khác nhau được đọc bởi các luồng. Tất cả các luồng của toàn bộ mạch đọc cùng địa chỉ đối lập với trường hợp tất cả luồng nằm trong một nửa của mạch.

#### **2.5.1.2.3. Bộ nhớ kết cấu**

Không gian vùng nhớ kết cấu được lưu vào vùng đệm, vì vậy việc đọc kết cấu mất một lần đọc từ thiết bị nhớ chỉ trong trường hợp không có trong cache, ngược lại nó chỉ mất một lần đọc từ vùng đệm kết cấu. Vùng đệm kết cấu được tối ưu cho không gian 2D, các luồng của cùng warp đọc địa chỉ kết cấu gần nhau hơn sẽ đạt được hiệu năng tối đa.

Đọc bộ nhớ thiết bị qua việc lấy kết cấu có thể là một lựa chọn nâng cao để đọc bộ nhớ thiết bị từ bộ nhớ toàn cục hoặc bộ nhớ hằng số như mô tả chi tiết trong phần V.4.

#### **2.5.1.2.4. Bộ nhớ dùng chung**

Vì bộ nhớ dùng chung gắn trên chip, nên không gian bộ nhớ dùng chung nhanh hơn nhiều so với các không gian bộ nhớ cục bộ và bộ nhớ toàn cục. Trong thực tế, để tất cả các luồng của một warp, truy cập vào bộ nhớ dùng chung sẽ nhanh như truy cập vào một thanh ghi miễn là không có bất kỳ sự xung đột dải nhớ (bank) giữa các luồng, như chi tiết dưới đây.

Để có được băng thông bộ nhớ cao, bộ nhớ dùng chung được chia thành các module bộ nhớ có kích thước bằng nhau, được gọi là các dải nhớ, mà có thể được truy cập cùng một lúc. Vì vậy, bất kỳ bộ nhớ đọc hoặc ghi yêu cầu thực hiện của các  $n$  địa chỉ nằm trong  $n$  dải nhớ riêng biệt thì có thể được phục vụ đồng thời, hiệu suất của một băng thông hiệu quả cao hơn  $n$  lần băng thông của một module đơn lẻ.

Tuy nhiên, nếu hai địa chỉ của một yêu cầu bộ nhớ rơi cùng vào một dải nhớ, đó là một xung đột dải nhớ và việc truy cập vào các dải nhớ phải được nối tiếp. Phần cứng, khi cần thiết, thực hiện việc chia tách một yêu cầu vùng nhớ với các xung đột dải nhớ thành nhiều các yêu cầu không bị tranh chấp riêng biệt, làm giảm băng thông hiệu quả do một yếu tố bằng với số yêu cầu của bộ nhớ riêng biệt. Nếu số lượng yêu cầu bộ nhớ riêng biệt là  $n$ , yêu cầu vùng nhớ khởi tạo ban đầu sẽ gây ra xung đột bank theo  $n$  cách.

Để có được hiệu suất tối đa, quan trọng đó là hiểu được các địa chỉ vùng nhớ được ánh xạ với các dải nhớ như thế nào từ đó để lập ra lịch trình các yêu cầu vùng nhớ và để giảm thiểu sự xung đột giữa dải nhớ. Trong trường hợp của không gian bộ nhớ dùng chung, các bank được tổ chức như liên tiếp các từ 32-bit và được gán cho liên tiếp các dải nhớ và mỗi bank có một băng thông 32 bit trong mỗi hai chu kỳ đồng hồ.

Đối với các thiết bị khả năng tính toán 1.x, kích thước warp là 32 và số lượng của các bank nhớ là 16 (xem phần V.1); một yêu cầu bộ nhớ dùng chung cho một warp được chia thành một yêu cầu cho nửa đầu của warp và yêu cầu một cho nửa sau của warp. Như một hệ quả, có thể không có xung đột dải nhớ giữa một luồng thuộc một nửa đầu tiên một warp và một một luồng thuộc nửa sau cùng warp đó

#### **2.5.2. Số lượng luồng trong một khối**

Khái niệm số lượng luồng trên lưới, số luồng trên khối lệnh (block) hoặc số khối lệnh nên được lựa chọn nhằm tối ưu hóa tài nguyên sẵn dụng của bộ xử lý. Điều này có nghĩa là, với việc xử lý nhiều tập lệnh khác nhau nên cần nhiều bộ xử lý trên mỗi thiết bị phân cứng.

Hơn nữa, việc xử lý một khối lệnh trên bộ xử lý đa nhân tương đương với việc làm tăng thời gian nghỉ của bộ xử lý trong suốt quá trình đồng bộ và truy cập bộ nhớ nếu không đủ luồng trên một khối lệnh. Điều này vẫn tốt hơn cho việc xử lý hai hoặc nhiều tập lệnh thực hiện trên cùng một bộ xử lý đa nhân, các khối lệnh vẫn phải đợi thực hiện theo hàng. Đối với vấn đề này, việc xử lý nhiều khối lệnh đòi hỏi phải có nhiều bộ xử lý trên cùng một thiết bị, mà còn liên quan tới việc phân chia vùng nhớ chia sẻ cho mỗi khối lệnh nên được dành phần lớn số lượng bộ nhớ sẵn dùng đối với mỗi bộ đa xử lý. Rất nhiều luồng xử lý theo luồng thông qua thiết bị và được thực thi dần dần tuần tự.

Với những khối lệnh có số lượng lớn, số lượng luồng thực thi trên mỗi khối lệnh nên được lựa chọn xử lý với nhiều kích cỡ khác nhau nhằm tránh phải chờ đợi tài nguyên tính toán sai lệch, để xử lý tốt hơn, tham khảo toán tử 64 bit được nêu trong mục V.1.2.5. Việc tổ chức nhiều luồng xử lý trên khối lệnh được cho hiệu quả tốt hơn khi chi nhỏ theo thời gian, nhưng nhiều luồng xử lý trên một bộ lệnh tương ứng với việc mất nhiều chi phí đăng ký cho mỗi luồng. Điều này sẽ ngăn chặn cách gọi hàm nhân xử lý tuần tự nếu hàm nhân thực hiện xử lý nhiều đăng ký hơn là việc được phép bởi cấu hình thực hiện từ trước.

Đối với các thiết bị tính toán gấp 1.x, số lượng đăng ký sẵn sàng trên một luồng được tính theo công thức:

$$\frac{R}{B \times \text{ceil}(T, 32)}$$

Trong đó:

R – tổng số đăng ký trên bộ đa xử lý (phụ lục A, B chính là số lượng khối lệnh hiện thời)

T – số luồng xử lý trên một khối lệnh

$\text{ceil}(T, 32)$  – là phép tính làm tròn lên số bội số của 32.

64 luồng trên một khối lệnh là tối thiểu và tạo ra cảm giác có rất nhiều khối lệnh đang thực hiện đồng thời. Với các giá trị 192 hoặc 256 luồng trên khối lệnh thường hợp lý hơn và cho phép đăng ký vừa đủ để xử lý.

Số lượng khối lệnh trên lưới nên để tối thiểu là 100 nếu muốn các khối lệnh này phân chia thành các thiết bị tương lai; 1000 khối lệnh sẽ phân ngang cấp thành một số phát sinh khác.

Tỉ lệ số lượng sai lệch khi thực thi xử lý đồng thời trên bộ đa xử lý so với số lượng tối đa được gọi là thời gian chiếm giữ bộ xử lý (*occupancy*).

### 2.5.3. Truyền dữ liệu giữa Host và device

Băng thông trao đổi dữ liệu giữa thiết bị và bộ nhớ thường cao hơn nhiều so với băng thông giữa bộ nhớ và bộ nhớ của máy chủ. Vì vậy, nên phân đầu tối thiểu hóa việc truyền tải dữ liệu giữa máy chủ và thiết bị. Ví dụ cấu trúc dữ liệu tức thời có thể được tạo trong bộ nhớ thiết bị, tính toán bởi thiết bị và quá trình hủy bỏ không phụ thuộc vào máy chủ / bộ nhớ trên máy chủ.

Đối với tổng chi phí trên đường truyền, việc xử lý theo lô các gói nhỏ trên đường truyền lớn sẽ mang lại hiệu quả tốt hơn so với việc truyền tải một lần với khối lượng lớn.

### 2.5.4. Lợi ích của việc tổ chức bộ nhớ

Việc tổ chức bộ nhớ theo cấu trúc trên các thiết bị nhớ sẽ mang lại nhiều lợi ích hơn so với việc truy cập từ bộ nhớ toàn cục (global) hoặc hằng số (constant):

- Dữ liệu được lưu vào vùng đệm, khả năng truy xuất dữ liệu nhanh hơn rất nhiều
- Chúng không được bị ràng buộc vào việc truy xuất các phần tử bộ nhớ
- Quá trình tính toán địa chỉ được giảm thiểu hơn, cải thiện hiệu năng cho ứng dụng truy xuất dữ liệu ngẫu nhiên.
- Dữ liệu đóng gói có thể được phân chia thành các biến tách biệt trong toán tử đơn.
- Dữ liệu đầu vào số nguyên 8 và 16 bit có thể được chuyển đổi thành 32 bit dấu phẩy động thuộc với các giá trị nằm trong dải  $[0, 1]$ .

Nếu việc tổ chức theo mảng CUDA (tham khảo mục 2.3.3.4.2), phần cứng sẽ đáp ứng nhiều điểm khả năng khác, mang lại hiệu quả cho nhiều ứng dụng khác nhau, đặc biệt trong công nghệ xử lý ảnh:

Đặc điểm	Phù hợp với nội dung	Giới hạn
Bộ lọc	Nhanh, độ chính xác thấp nội suy giữa các texel	Chỉ có giá trị nếu tổ chức tham chiếu trả dữ liệu số thực
Chuẩn hóa cấu trúc tọa độ	Độ phân giải đánh mã độc lập	
Cơ chế đánh địa chỉ	Tự động nhận biết các giá trị biên	Có thể sử dụng để chuẩn hóa cấu trúc tọa độ

## Chương 3.

# ỨNG DỤNG GPU VÀO BÀI TOÁN N-BODY VÀ THỬ NGHIỆM CHƯƠNG TRÌNH

### 3.1. Bài toán mô phỏng N-body

N-body là bài toán tiêu biểu cho tính toán hiệu năng cao, được ứng dụng rộng rãi trong các mô phỏng vật lý, hóa học, thiên văn học với khối lượng tính toán rất lớn.

Mô phỏng n-body là mô phỏng số lượng rất lớn các hạt dưới ảnh hưởng của các lực vật lý, thường là lực hấp dẫn. Mô phỏng này thường được sử dụng trong vũ trụ học để nghiên cứu các quá trình dữ liệu cấu trúc phi tuyến tính như cơ cấu hình thành các dải thiên hà và các ngôi sao từ hố đen trong thiên văn học. Mô phỏng n-body trực tiếp được dùng trong nghiên cứu vụ nổ của các cụm sao.

Trong nhiều trường hợp, kích thước của mô phỏng thiên văn học N-body bị giới hạn bởi các tài nguyên tính toán hiện có. Mô phỏng cho hệ thống N-body hấp dẫn thuần khiết là một ví dụ điển hình. Vì lực hấp dẫn là sự tương tác ở khoảng cách dài (long-range), độ phức tạp tính toán cho sự tương tác giữa tất cả các phần tử là  $O(N^2)$  cho từng bước tính toán của mô hình đơn giản nhất, với N là số lượng phần tử trong hệ thống. Chúng ta có thể giảm độ phức tạp tính toán từ  $O(N^2)$  còn  $O(N \log N)$  bằng cách sử dụng một vài thuật toán xấp xỉ, như thuật toán cây Barnes-Hut [~11], nhưng hệ số tỉ lệ (scaling coefficient) thực sự lớn. Do vậy, tính toán sự tương tác giữa các phần tử thường là phần "đắt" nhất trong toàn bộ việc tính toán, và do đó giới hạn số lượng các phần tử chúng ta có thể xử lý. Smoothed Particle Hydrodynamics (SPH) [[~3][~22] trong đó các phần tử biểu diễn phân tử chất lỏng (khí) là một ví dụ khác. Trong các tính toán SPH, phương trình tính toán thủy động học được biểu diễn bởi sự tương tác giữa các phần tử ở khoảng cách ngắn (short-range). Độ phức tạp tính toán của SPH tương đối cao bởi số lượng phần tử trung bình tương tác với 1 phần tử thực sự lớn, thường dao động ở 50, và tính toán tương tác giữa từng cặp 2 phần tử thì phức tạp hơn một chút so với tương tác hấp dẫn.

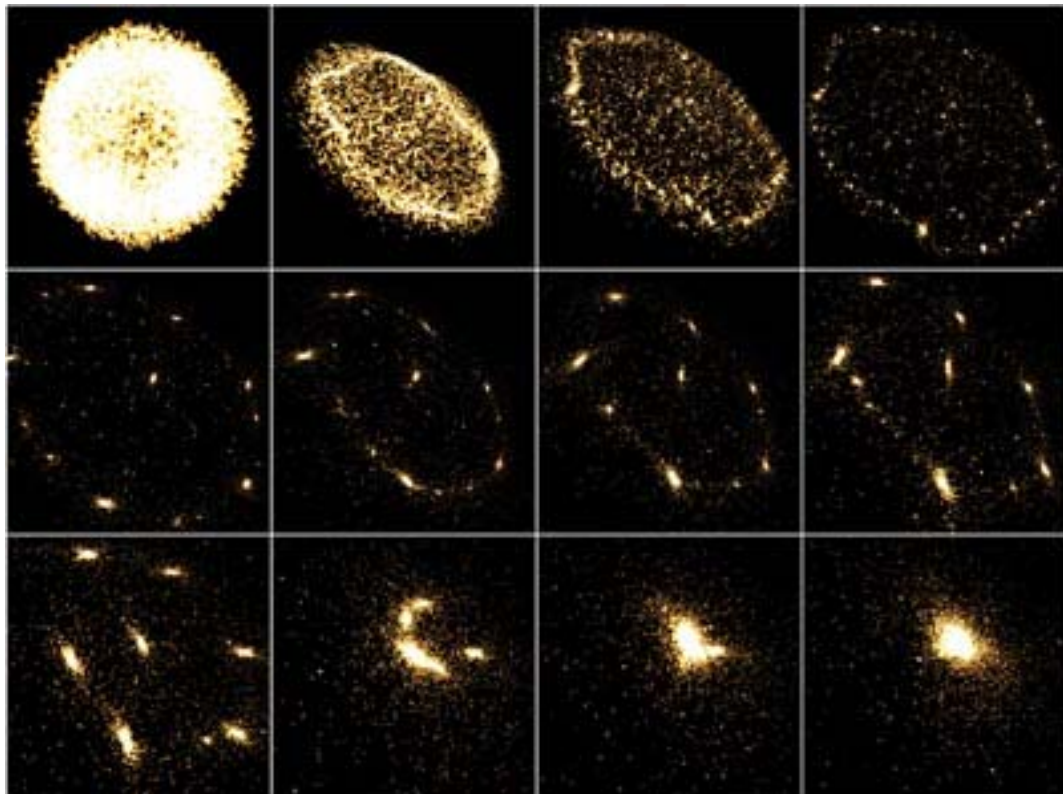
Trong mô phỏng thiên văn học N-body, tương tác quan trọng nhất là lực hấp dẫn. Sử dụng tài máy tính, chúng ta tính được lực hấp dẫn của phần tử thứ i từ j phần tử theo công thức sau:

Thiên văn học không phải là ứng dụng duy nhất của mô phỏng N-body. Mô phỏng chuyển động phân tử (MD - Molecular dynamics) và phương pháp khoanh vùng thành phần (BEM - Boundary element method) là những ví dụ của phương pháp số học trong đó từng thành phần trong hệ thống nằm trong những tương tác cơ bản với các thành phần còn lại của hệ thống. Trong cả 2 trường hợp, cách tiếp cận giống với thuật toán cây Barnes-Hut hoặc FMM [~21] giúp giảm được độ chi phí tính toán, tuy nhiên tính toán sự tương tác vẫn ảnh hưởng lớn tới tổng chi phí tính toán.

Một cách tiếp cận tiêu biểu làm tăng tốc mô phỏng N-body là xây dựng 1 máy tính chuyên biệt (special-purpose) cho việc tính toán tương tác. Hai đặc trưng của tính toán tương tác làm chúng rất phù hợp với cách tiếp cận này, đó là:

Thứ nhất, tính toán tương tác trên từng cặp đôi là tương đối đơn giản. Trong trường hợp tương tác hấp dẫn, tổng số phép toán dấu phẩy động (đếm toàn bộ phép toán bao gồm cả phép nhân bậc hai và phép chia) cũng chỉ dao động quanh 20. Do vậy không khó hiểu khi thiết kế 1 bộ xử lý có đường ống dẫn đầy đủ, gắn cứng và được kiểm soát bằng vi mạch điện tử phục vụ việc tính toán tương tác hấp dẫn. Với những ứng dụng khác như SPH, hay chuyển động phân tử học thì tính toán tương tác phức tạp hơn, tuy nhiên cách tiếp cận về phần cứng vẫn khả thi.

Đặc trưng thứ hai, sự tương tác theo cách đơn giản nhất của nó tác động tất cả lên tất cả (all-to-all). Nói cách khác, từng phần tử trong hệ thống tác động lên tất cả các phần tử còn lại. Do đó có rất nhiều cách thức song song áp dụng được. Nói cụ thể, có thể thiết kế một phần cứng để tính toán lực tác động tự 1 phần tử lên nhiều phần tử một cách song song. Theo cách này chúng ta có thể giảm yêu cầu về băng thông (bandwidth) bộ nhớ. Tất nhiên là nếu sự tương tác thuộc vào loại khoảng cách ngắn (short-range), thì nó có thể thực hiện một số cách thông minh để làm giảm chi phí tính toán từ  $O(N^2)$  thành  $O(N)$ , và việc làm giảm băng thông bộ nhớ không hiệu quả bằng trường hợp tính toán  $O(N^2)$



Hình 19: Hình ảnh mô phỏng N-body [~8]

### 3.2. Xây dựng bài toán N-body trên CPU

Mô phỏng n-body có 2 loại là mô phỏng trực tiếp và mô phỏng tương đối. Trong phần tiếp theo mô tả mô phỏng trực tiếp sử dụng phương pháp đơn giản nhất là: phương pháp hạt-hạt (particle-particle (PP)). Tác giả luận văn đã cài đặt mô phỏng N-body chạy trên CPU theo phương pháp trên. Đồng thời có các đánh giá thử nghiệm trên mô phỏng này so sánh với mô phỏng trên GPU. Chi tiết được trình bày trong phần kết quả thử nghiệm.

Phương pháp hạt-hạt dựa trên thuật toán tích hợp thời gian và thuật toán tính lực. Thuật toán Verlet(1967) mô tả dưới đây là thuật toán tích hợp thời gian phổ biến nhất và đơn giản nhất. Những thuật toán tích hợp thời gian khác như: leap-frog, Beeman(1976), Tuckerman là các thuật toán có nhiều bước thời gian.

#### 3.2.1. Thuật toán tích hợp thời gian Verlet:

Giả sử  $\vec{r}(t)$ ,  $\vec{v}(t)$  và  $\vec{a}(t)$  tương ứng là vị trí, vận tốc và gia tốc của hạt P tại thời điểm  $t$ . Thuật toán Verlet được biểu diễn dưới dạng:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \Delta t \cdot \vec{v}(t + \Delta t / 2) \quad (1)$$

$$\vec{v}(t + \Delta t / 2) = \vec{v}(t - \Delta t / 2) + \Delta t \cdot \vec{a}(t) \quad (2)$$

#### 3.2.2. Công thức tính lực cơ bản và tính tiềm năng

Để công thức đơn giản, ta giả sử  $N=2$ , gọi  $\vec{r}_1, \vec{v}_1, m_1$  và  $\vec{r}_2, \vec{v}_2, m_2$  là vị trí, vận tốc và gia tốc của một số lượng lớn hai hạt  $P_1$  và  $P_2$ , trong hệ thống 2-body. Lực  $F_1$  đưa vào hạt  $P_1$  phụ thuộc vào  $P_2$ :

$$\vec{F}_1 = -G \frac{m_1 m_2}{|\vec{r}_1 - \vec{r}_2|^2} \hat{r}_{12} \quad (3)$$

Công thức tương tự cho lực  $F_2$ . Sử dụng định luật 2 Newton  $\vec{F} = m\vec{a}$  chúng ta có gia tốc của hạt  $P_1$ .

$$\vec{a}_1 = -G \frac{m_2}{|\vec{r}_1 - \vec{r}_2|^2} \hat{r}_{12} \quad (4)$$

Trong đó  $G$  là hằng số hấp dẫn và  $\hat{r}_{12}$  là vector đơn vị

$$\hat{r}_{12} = \frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|}$$

Tiềm năng  $\Phi_1$  tại vị trí  $\vec{r}_1$  phụ thuộc vào  $P_2$ :

$$\Phi_1 = -G \frac{m_2}{|\vec{r}_1 - \vec{r}_2|} \quad (5)$$

Trong mô tả sản xuất, các nhà nghiên cứu thường sử dụng các tham số mềm để tránh các lực và gia tốc rất lớn gây ra do khoảng cách rất gần giữa các hạt. Trong hệ thống N-body, các tham số mềm thường được định nghĩa là :

$$\varepsilon = \frac{1}{\sqrt[3]{N}} \quad (6)$$

Sử dụng các tham số mềm, biểu thức (4), (5) trở thành:

$$\vec{a}_1 = -G \frac{m_2}{|\vec{r}_1 - \vec{r}_2|^2 + \varepsilon^2} \cdot \frac{(\vec{r}_1 - \vec{r}_2)}{\sqrt{|\vec{r}_1 - \vec{r}_2|^2 + \varepsilon^2}} \quad (7)$$

Và:

$$\Phi_1 = -G \frac{m_2}{\sqrt{|\vec{r}_1 - \vec{r}_2|^2 + \varepsilon^2}} \quad (8)$$

Động năng  $K_i$  của các hạt  $P_i$  với  $i=1,2$  là

$$K_i = \frac{1}{2} m_i v_i^2 \quad (9)$$

Và tiềm năng năng lượng  $W_i$  của hạt  $P_i$  là

$$W_i = \frac{1}{2} m_i \Phi_i \quad (10)$$

Động năng  $K_i$  và năng lượng tiềm năng  $W_i$  hệ thống 2-body là:

$$K = \sum_{i=1}^2 K_i \quad (11)$$

$$W = \sum_{i=1}^2 W_i \quad (12)$$

Và năng lượng tổng cộng của hệ thống là:

$$E = K + W \quad (13)$$

### 3.2.3. Thuật toán mô phỏng N-Body

1. Đánh giá gia tốc ban đầu của mỗi hạt bằng cách sử dụng biểu thức (7).
2. Đánh giá tổng năng lượng ban đầu  $E_0$  của hệ thống bằng cách sử dụng biểu thức (2), (8) - (13).
3. Xác định các tham số mềm  $\varepsilon$  bằng cách sử dụng biểu thức (6). Set  $\Delta t = \varepsilon / 2$ , và  $t=0$ .
4. Xác định số lượng bước thời gian và giới hạn trên  $t_{max}$  của  $t$ .
5. for // lặp cho mỗi bước theo thời gian
6. Tính vận tốc của mỗi hạt tại thời điểm  $t + \Delta t / 2$  sử dụng biểu thức (2).
7. Tính vị trí mới của mỗi hạt tại thời điểm  $t + \Delta t$  bằng biểu thức (1).
8. Tính gia tốc của mỗi hạt tại thời điểm  $t + \Delta t$  sử dụng biểu thức (7).

9. Tính tốc độ của mỗi hạt tại thời điểm  $t + \Delta t$  sử dụng biểu thức (2).
10. (Tùy chọn) Tính tổng năng lượng  $E_t$  của hệ thống sử dụng biểu thức (8)-(13).
11. (Tùy chọn) Tính sai số tương đối  $(E_t - E_0)/E_0$  nếu bước 8 đã thực hiện
12. (Tùy chọn) Tính  $2K/|W|$  để giám sát sự cân bằng của hệ thống (equilibrium).
13.  $t = t + \Delta t / 2$
14. nếu  $t > t_u$  hoặc số bước thời gian đạt tới một số lượng định trước, break thoát khỏi vòng lặp.
15. endfor //kết thúc lặp

### 3.3. Xây dựng bài toán N-body trên GPU

Bài toán thử nghiệm N-body trên GPU được tham khảo từ [~8]. Tác giả luận văn đã tìm hiểu, nghiên cứu mã nguồn ví dụ, điều chỉnh các tham số chương trình và cài đặt trên môi trường thử nghiệm.

Các bước thực hiện:

1. Cài đặt bộ tool kit của NVIDIA phiên bản 1.0 trở lên. Có thể download tại <http://developer.nvidia.com/cuda>
2. Cấu hình GPU cần thiết: NVIDIA 8-Series hoặc mới hơn.
3. Biên dịch chương trình trên Linux:

Make file biên dịch chương trình:

```
#####
#####
#
# Build script for project
#
#####
#####

# Add source files here
EXECUTABLE := nbody
# Cuda source files (compiled with cudacc)
CUFILES := bodysystemcuda.cu
# C/C++ source files (compiled with gcc / c++)
CCFILES := \
    nbody_gold.cpp bodysystemcpu.cpp bodysystemcuda.cpp nbody.cpp \
    render_particles.cpp \

USEGLLIB := 1
USEPARAMGL := 1
USEGLUT := 1

#####
#####
# Rules and targets
```

```
include ../../common/common.mk
```

Ở thư mục gốc, gõ lệnh:

```
Make; Make dbg=1  
"Make -f Makefile_paramgl; Make -f Makefile_paramgl dbg=1"
```

Vào thư mục: projects/nbody, gõ lệnh:

```
Make; Make dbg=1; Make emu=1; Make emu=1 dbg=1
```

Chương trình được chạy trong thư mục:

```
bin/linux/release/nbody
```

Chương trình có 3 chế độ chạy: interactive (đồ họa) , benchmark và test.

*Chế độ interactive*: chạy đồ họa, cho phép người dùng có thể nhìn thấy mô phỏng n-body, các hạt chuyển động.

*Chế độ test*: mô phỏng được chạy trên cả CPU và GPU. Nếu mô phỏng trên GPU nằm trong sự cho phép của mô phỏng trên CPU thì kết quả hiển thị "Test PASSED", ngược lại hiển thị "Test FAILED"

*Chế độ benchmark*: chỉ chạy tính toán các tương tác, không có chuyển đổi sang đồ họa 3D và không có thời gian chờ. Báo cáo mô phỏng gồm: tổng thời gian, thời gian trung bình, trung bình tương tác giữa các phần tử trong 1 giây, tỷ số GFLOP/s. Tác giả luận văn lựa chọn chế độ này để chạy thử nghiệm hiệu năng GPU so sánh với CPU. Lệnh chạy như sau:

```
nohup echo "8388608 start" >> linh_result.txt && date >> linh_result.txt && nbody -  
benchmark -n=8388608 >> linh_result.txt && date >> linh_result.txt && echo "8388608  
end" >> linh_result.txt &
```

Trong đó:

**linh\_result.txt** là file chứa kết quả chạy chương trình

**nohup**: lệnh chạy mô phỏng trên server, không cần tương tác với client

**date**: lệnh ghi thời gian hệ thống khi bắt đầu và kết thúc chương trình

**nbody -benchmark -n=8388608**: chạy mô phỏng ở chế độ benchmark, với số phần tử n= 8388608 (8192K)

### 3.4. Thử nghiệm

#### 3.4.1. Môi trường thử nghiệm:

Hai mô phỏng được miêu tả ở phần 3.2 và 3.3 được cài đặt thử nghiệm trên môi trường máy PC có cấu hình như sau:

```
CPU: Intel(R) Core(TM)2 Quad CPU @ 2.66GHz
```

Cache: 4096 KB

RAM: 2 GB

GPU: Nvidia GeForce 8800 GTX

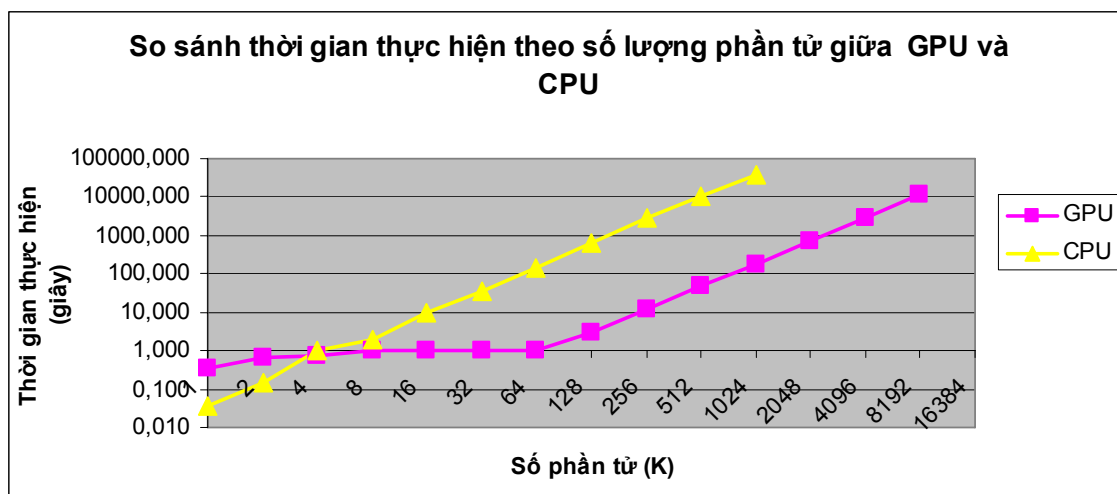
Hệ điều hành Linux

### 3.4.2. Kết quả thử nghiệm

#### 3.4.2.1.1. So sánh thời gian thực hiện trên GPU và CPU

Số phần tử (K)	Thời gian chạy trên GPU	Thời gian chạy trên CPU
1	0.348	0.035
2	0.687	141
4	0.7	1
8	1	2
16	1	10
32	1	36
64	1	148
128	3	608
256	12	2940
512	48	10009
1024	179	38922
2048	719	Thời gian tính toán rất lớn cỡ vài ngày
4096	2820	
8192	11348	

Bảng 1: Kết quả thử nghiệm bài toán N-body trên GPU Nvidia GeForce 8800 GTX và CPU Intel(R) Core(TM)2 Quad 2.66GHz



Hình 20: Biểu đồ so sánh thời gian thực hiện giữa GPU và CPU theo số lượng phần tử trong mô phỏng n-body

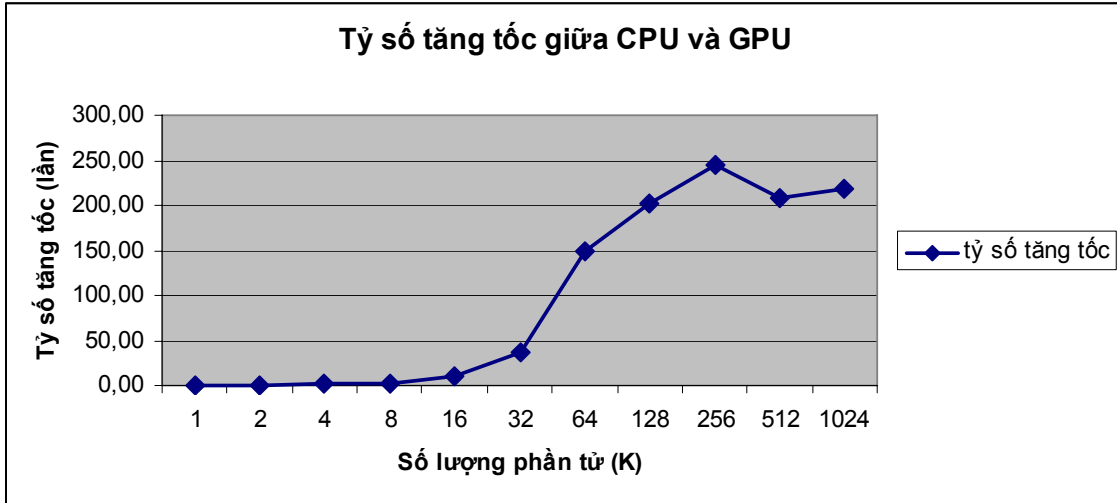
**Nhận xét:** Với số lượng phần tử nhỏ  $< 8K$  thì tính toán trên CPU và GPU đều rất nhanh, cỡ dưới 1 giây, và CPU có phần nhanh hơn. Nhưng khi số lượng phần tử trong mô phỏng tăng lên theo cấp số 2 thì tính toán trên GPU nhanh hơn CPU rất nhiều lần. Trên đồ thị thời gian thực hiện có thể chứng minh độ phức tạp thuật toán bằng  $O(N^2)$

với N là số phần tử. Khi số phần tử tăng lên gấp 2 thì thời gian thực hiện tăng lên gấp 4 lần trên cả CPU và GPU.

### 3.4.2.1.2. Tỷ số tăng tốc giữa CPU và GPU

Số phần tử	1	2	4	8	16	32	64	128	256	512	1024
Tỷ số tăng tốc	0,10	0,21	1,43	2,00	10,00	36,00	148,00	202,67	245,00	208,52	217,44

Bảng 2: Tỷ số tăng tốc giữa CPU và GPU



Hình 21: Biểu đồ thể hiện tỷ số tăng tốc CPU/GPU khi số phần tử trong mô phỏng n-body tăng

**Nhận xét:** Thực nghiệm trên khảo sát tỷ lệ tăng tốc tốc độ giữa CPU và GPU. Ở những mô phỏng số lượng phần tử thấp thì tốc độ của CPU và GPU là tương đương nhau. Nhưng khi số lượng phần tử cao từ 64K trở nên thì tốc độ chênh lệch đáng kể duy trì ở mức trên 200 lần. Điều này cho thấy sức mạnh tính toán của GPU.

### 3.4.2.1.3. Hiệu năng tính toán trên CPU

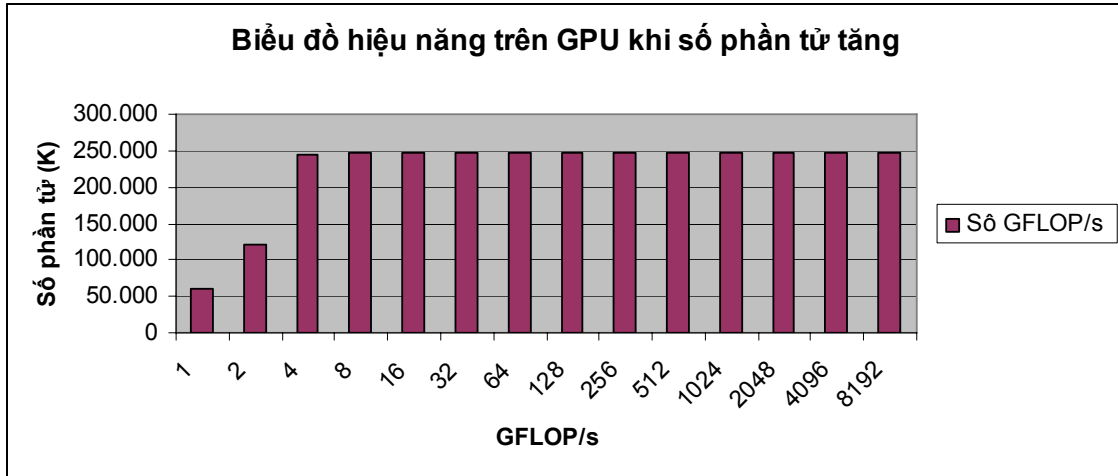
```
top - 10:01:54 up 8 days, 16:45, 1 user, load average: 2.38, 1.15, 0.68
Tasks: 117 total, 4 running, 113 sleeping, 0 stopped, 0 zombie
Cpu(s): 75.0%us, 0.1%sy, 0.0%ni, 24.9%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 2057248k total, 936592k used, 1120656k free, 167984k buffers
Swap: 2031608k total, 0k used, 2031608k free, 533932k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
21656	chau	25	0	51288	40m	12m	R	100.2	2.0	2:20.33	nbody
21675	chau	25	0	28760	20m	6520	R	100.2	1.0	0:48.93	nbody
21655	chau	25	0	28760	20m	6296	R	99.9	1.0	2:20.29	nbody
1	root	15	0	10344	680	568	S	0.0	0.0	0:02.17	init
2	root	RT	-5	0	0	0	S	0.0	0.0	0:00.01	migration/0
3	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/0
4	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/0
5	root	RT	-5	0	0	0	S	0.0	0.0	0:00.07	migration/1
6	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/1
7	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/1
8	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	migration/2
9	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/2
10	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/2
11	root	RT	-5	0	0	0	S	0.0	0.0	0:00.04	migration/3
12	root	34	19	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/3
13	root	RT	-5	0	0	0	S	0.0	0.0	0:00.00	watchdog/3
14	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	events/0
15	root	10	-5	0	0	0	S	0.0	0.0	0:00.00	events/1

Hình 22: Tải tính toán trên CPU khi chạy mô phỏng n-body với số phần tử 256K. 1 CPU luôn ở 100%, đôi khi chiếm thêm tải 100% của các CPU khác

Khi thực hiện mô phỏng trên CPU thì tải CPU luôn đạt mức cao nhất 100% trong suốt thời gian chạy. Chứng tỏ sự tiêu tốn tài nguyên khi tính toán trên CPU.

### 3.4.2.1.4. Hiệu năng tính toán trên GPU



Hình 23: Biểu đồ hiệu năng trên GPU Geforce 8800 GTX trong mô phỏng n-body khi số phần tử tăng

Số phần tử (K)	Tốc độ xử lý (GFLOP/s)
1	60.191
2	122.183
4	245.350
8	247.043
16	247.676
32	247.999
64	248.091
128	248.119
256	248.137
512	248.146
1024	248.144
2048	248.145
4096	248.142
8192	248.139

Bảng 3: Tốc độ xử lý trên GPU 8800 GTX khi số phần tử tăng

**Nhận xét:** Khi thực hiện mô phỏng N-body trên GPU 8899 GTX, tài nguyên được sử dụng mở mức tối đa khi số phần tử từ 4K trở nên. Và duy trì ở mức 248GFLOP/s.

### 3.5. Kết luận thử nghiệm

Các kết quả thử nghiệm trong luận văn đã cho thấy năng lực tính toán vượt trội của GPU so với CPU trong bài toán tính toán song song mô phỏng n-body. Kết quả cho thấy card đồ họa Geforce 8800 GTX có năng lực xử lý trong bài toán song song gấp khoảng hơn 200 lần so với chip Intel Quad core 2.66GHz. Một kết quả ấn tượng!

## 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 n-body trên GPU, giảm độ phức tạp tính toán từ  $O(N^2)$  xuống còn  $O(n \log n)$ . Mong rằng các kết quả nghiên cứu trong tương lai của luận văn sẽ đạt được điều đó.

## TÀI LIỆU THAM KHẢO

- [1] E. Lefohn, “*A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces*,” Master’s thesis, University of Utah, Dec. 2003.
- [2] Bustos, O. Deussen, S. Hiller, and D. Keim, “*A graphics hardware accelerated algorithm for nearest neighbor search*,” in Proceedings of the 6th International Conference on Computational Science, ser. Lecture Notes in Computer Science. Springer, May 2006, vol. 3994, pp. 196–199.
- [3] Blythe, “*The Direct3D 10 system*,” ACM Transactions on Graphics, vol. 25, no. 3, pp. 724–734, Aug. 2006.
- [4] Horn, “*Stream reduction operations for GPGPU applications*,” in GPU Gems 2, M. Pharr, Ed. Addison Wesley, Mar. 2005, ch. 36, pp. 573–589.
- [5] Tarditi, S. Puri, and J. Oglesby, “*Accelerator: Using data-parallelism to program GPUs for general-purpose uses*,” in Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2006, pp. 325–335.
- [6] Eclipse Parallel Tools Platform, <http://www.eclipse.org/ptp/>
- [7] Gingold and J. J. Monaghan, “*Smoothed particle hydrodynamics - theory and application to non-spherical stars*,” MNRAS, vol. 181, pp. 375–389, 1977.
- [8] GPU Gems 3, Chapter 31. Fast N-Body Simulation with CUDA [http://http.developer.nvidia.com/GPUGems3/gpugems3\\_ch31.html](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html)
- [9] I. Buck, T. Foley, D. Horn, J. Sugerma, K. Fatahalian, M. Houston, and P. Hanrahan, “*Brook for GPUs: Stream computing on graphics hardware*,” ACM Transactions on Graphics, vol. 23, no. 3, pp. 777–786, Aug. 2004.
- [10] *Introduction to Parallel Computing*, [http://www.llnl.gov/computing/tutorials/parallel\\_comp/](http://www.llnl.gov/computing/tutorials/parallel_comp/)
- [11] J. Barnes and P. Hut, “*A Hierarchical  $O(N \log N)$  Force-Calculation Algorithm*,” Nature, vol. 324, pp. 446–449, Dec. 1986.
- [12] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, “*Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*,” ACM Transactions on Graphics, vol. 22, no. 3, pp. 917–924, Jul. 2003.
- [13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, “*A survey of general-purpose computation on graphics hardware*,” Computer Graphics Forum, vol. 26, no. 1, pp. 80–113, 2007.
- [14] J. Krüger and R. Westermann, “*Linear algebra operators for GPU implementation of numerical algorithms*,” ACM Transactions on Graphics, vol. 22, no. 3, pp. 908–916, Jul. 2003.
- [15] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann, “*A particle system for interactive visualization of 3D flows*,” IEEE Transactions on Visualization and Computer Graphics, vol. 11, no. 6, pp. 744–756, Nov./ Dec. 2005.

- [16] J. Postel, J. Reynolds, <http://www.ietf.org/rfc/rfc0959.txt> , RFC File Transfer Protocol, 1985
- [17] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, "GPU Computing", PROCEEDINGS OF THE IEEE, VOL. 96, NO. 5, MAY 2008
- [18] K. E. Batcher, "Sorting networks and their applications," in Proceedings of the AFIPS Spring Joint Computing Conference, vol. 32, Apr. 1968, pp. 307–314.
- [19] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in Graphics Hardware 2004, Aug. 2004, pp. 133–138.
- [20] L. B. Lucy, "A numerical approach to the testing of the fission hypothesis," Astronomical Journal, vol. 82, pp. 1013–1024, Dec. 1977
- [21] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," Journal of Computational Physics, vol. 73, pp. 325–348, Dec. 1987
- [22] M. Harris, "Mapping computational concepts to GPUs," in GPU Gems 2, M. Pharr, Ed. Addison Wesley, Mar. 2005, ch. 31, pp. 493–508.
- [23] M. Kass, A. Lefohn, and J. Owens, "Interactive depth of field using simulated diffusion on a GPU," Pixar Animation Studios, Tech. Rep. #06-01, Jan. 2006, <http://graphics.pixar.com/DepthOfField/>.
- [24] M. McCool, "Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform," in GSPx Multicore Applications Conference, Oct./Nov. 2006.
- [25] "M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," ACM Transactions on Graphics, vol. 23, no. 3, pp. 787–795, Aug. 2000"
- [26] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LUGPU: Efficient algorithms for solving dense linear systems on graphics hardware," in Proceedings of the ACM/IEEE Conference on Supercomputing, Nov. 2005, p. 3.
- [27] N. K. Govindaraju and D. Manocha, "Efficient relational database management using graphics processors," in ACM SIGMOD Workshop on Data Management on New Hardware, Jun. 2005, pp. 29–34.
- [28] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Jun. 2004, pp. 215–226.
- [29] "N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha, "Interactive visibility ordering of geometric primitives in complex environments," in Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games, Apr. 2005, pp. 49–56."
- [30] PADE, <http://math.nist.gov/mcsd/savg/pade/>
- [31] P-GRADE, <http://www.lpds.sztaki.hu/pgrade/>
- [32] wikipedia [http://en.wikipedia.org/wiki/Graphics\\_processing\\_unit](http://en.wikipedia.org/wiki/Graphics_processing_unit)