|  |
| --- |
| **TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**  **VIỆN TOÁN ỨNG DỤNG VÀ TIN HỌC**  **---------------------------------------**  BLACK AND WHITE  **TÍNH TOÁN SONG SONG  TRÊN GPU VÀ ỨNG DỤNG**  **ĐỒ ÁN TỐT NGHỆP ĐẠI HỌC**  **Chuyên ngành: TOÁN TIN**  **Giáo viên hướng dẫn : ThS. Đoàn Duy Trung**  **Sinh viên thực hiện : Trần Tùng**  **Lớp : Toán tin ứng dụng I**    **HÀ NỘI – 2014** |

**NHẬN XÉT CỦA THẦY HƯỚNG DẪN**

1. **Mục đích và nội dung của đồ án:**
2. **Kết quả đạt được:**
3. **Ý thức làm việc của sinh viên:**

Hà Nội, ngày tháng năm

Thầy hướng dẫn

(Ký và ghi rõ họ tên)

# LỜI CẢM ƠN

Trước tiên tôi xin gửi lời cảm ơn tới toàn thể quý thầy cô và cán bộ của viện Toán Ứng Dụng và Tin Học trường Đại học Bách Khoa Hà Nội đã dạy dỗ và truyền đạt kiến thức quý báu cho tôi trong những năm học tập và rèn luyện ở trường.

Đặc biệt, tôi xin gửi lời cám ơn thầy Th.S Đoàn Duy Trung đã trực tiếp hướng dẫn, chỉ bảo tận tình về các vấn đề chuyên môn giúp tôi hoàn thiện đồ án này.

Và tôi cũng muốn gửi lời cảm ơn tới gia đình, bạn bè, người thân những người luôn ở bên động viên trong suốt quá trình học tập, nghiên cứu cũng như hoàn thành đồ án tốt nghiệp.

Đồ án đạt được một số kết quả nhất định, tuy nhiên trong thời gian thực hiện là có hạn nên không thể tránh khỏi sai sót và hạn chế. Kính mong nhận được sự thông cảm và đóng góp ý kiến từ quý thầy cô và các bạn.

|  |
| --- |
| Hà Nội, tháng 5 năm 2014 |
| Sinh viên |
| Trần Tùng |

# LỜI CAM ĐOAN

Tôi xin cam đoan kết quả đạt được trong khóa luận này là sản phẩm nghiên cứu, tìm hiểu của riêng cá nhân tôi dưới sự hướng dẫn của thầy ThS. Đoàn Duy Trung. Trong toàn bộ nội dung khóa luận, những điều được trình bày hoặc là của cá nhân tôi hoặc là được tổng hợp từ nhiều nguồn tài liệu khác. Tất cả tài liệu tham khảo đều có xuất xứ rõ ràng và trích dẫn hợp pháp. Tôi xin hoàn toàn chịu trách nhiệm và chịu mọi hình thức kỷ luật theo quy định cho lời cam đoan của mình.

|  |
| --- |
| Hà Nội, tháng 5 năm 2014 |
| Sinh viên |
| Trần Tùng |

MỤC LỤC

[LỜI CẢM ƠN 3](#_Toc389132461)

[LỜI CAM ĐOAN 4](#_Toc389132462)

[MỞ ĐẦU 4](#_Toc389132463)

[DANH SÁCH THUẬT NGỮ THƯỜNG SỬ DỤNG 6](#_Toc389132464)

[DANH MỤC HÌNH ẢNH 7](#_Toc389132465)

[DANH MỤC BẢNG BIỂU 8](#_Toc389132466)

[CHƯƠNG 1. GIỚI THIỆU TỔNG QUAN VỀ TÍNH TOÁN SONG SONG 9](#_Toc389132467)

[1.1 Khái niệm về tính toán song song. 9](#_Toc389132468)

[1.2 Phân loại kiến trúc song song. 10](#_Toc389132469)

[1.2.1 Kiến trúc đơn lệnh đa dữ liệu SIMD. 10](#_Toc389132470)

[1.2.2 Kiến trúc đa lệnh đa dữ liệu MIMD. 10](#_Toc389132471)

[1.2.3 Sử dụng bộ nhớ chia sẻ. 11](#_Toc389132472)

[1.2.4 Sử dụng bộ nhớ phân tán. 11](#_Toc389132473)

[1.3 Định luật Amdahl. 12](#_Toc389132474)

[1.4 Mô hình lập trình song song cơ bản. 12](#_Toc389132475)

[1.4.1 Mô hình chia sẻ bộ nhớ chung. 13](#_Toc389132476)

[1.4.2 Mô hình luồng. 13](#_Toc389132477)

[1.4.3 Mô hình truyền thông điệp. 14](#_Toc389132478)

[1.4.4 Mô hình song song dữ liệu. 14](#_Toc389132479)

[1.4.5 Mô hình lai. 14](#_Toc389132480)

[1.4.6 Tắc nghẽn (Deadlock) 14](#_Toc389132481)

[Chương 2. TỔNG QUAN VỀ GPU VÀ TÍNH TOÁN SONG SONG GPU. 16](#_Toc389132482)

[2.1 Tổng quan về GPU. 16](#_Toc389132483)

[2.2 Lịch sử phát triển. 16](#_Toc389132484)

[2.3 Kiến trúc GPU. 17](#_Toc389132485)

[2.3.1 Graphics Pipeline (Đường ống đồ họa ) 17](#_Toc389132486)

[2.3.2 Kiến trúc của GPU hiện đại. 18](#_Toc389132487)

[2.4 Mô hình lập trình tính toán trên GPU. 20](#_Toc389132488)

[2.5 Một số ứng dụng thực tế. 23](#_Toc389132489)

[2.5.1 Game 23](#_Toc389132490)

[2.5.2 Tăng tốc các ứng dụng. 24](#_Toc389132491)

[2.5.3 Sử dụng trong nghiên cứu. 24](#_Toc389132492)

[Chương 3. KIẾN TRÚC TÍNH TOÁN SONG SONG CUDA. 26](#_Toc389132493)

[3.1. Giới thiệu môi trường phát triển ứng dụng 26](#_Toc389132494)

[3.2. Kiến trúc phần cứng 27](#_Toc389132495)

[3.2.1 Multiprocessor. 27](#_Toc389132496)

[3.2.2 Cấu trúc SIMT và cách thức thực thi 29](#_Toc389132497)

[3.3. Kiến trúc lập trình. 29](#_Toc389132498)

[3.3.1 Kiến trúc lập trình kết hợp. 29](#_Toc389132499)

[3.3.2 Hạt nhân chương trình (Kernel). 30](#_Toc389132500)

[3.3.3 Luồng, khối luồng, lưới (thread, block, grid) 31](#_Toc389132501)

[3.3.4 Phân chia bộ nhớ. 31](#_Toc389132502)

[3.4. Phần mở rộng của ngôn ngữ lập trình C 33](#_Toc389132503)

[3.5 Tối ưu. 37](#_Toc389132504)

[Chương 4. CUDA VÀ MỘT SỐ BÀI TOÁN. 39](#_Toc389132505)

[4.1 Nhân hai ma trận. 39](#_Toc389132506)

[4.1.1 Cơ sở lý thuyết. 39](#_Toc389132507)

[4.1.2 Thuật toán tuần tự. 39](#_Toc389132508)

[4.1.3 Song song hóa. 40](#_Toc389132509)

[4.2 Thuật toán phân cụm k-means. 44](#_Toc389132510)

[4.2.1 Cơ sở lý thuyết. 44](#_Toc389132511)

[4.2.2 Thuật toán k-means tuần tự. 44](#_Toc389132512)

[4.2.3 Song song hóa thuật toán. 46](#_Toc389132513)

[4.3 Một số thuật toán trên ảnh. 50](#_Toc389132514)

[4.3.1 Giới thiệu về ảnh số. 50](#_Toc389132515)

[4.3.2 Mức xám của ảnh. 50](#_Toc389132516)

[4.3.3 Chuyển từ ảnh màu sang ảnh xám. 50](#_Toc389132517)

[4.3.4 Image Filtering. 51](#_Toc389132518)

[4.4 Thực nghiệm. 53](#_Toc389132519)

[4.4.1 Nhân hai ma trận. 53](#_Toc389132520)

[4.4.2 Thuật toán k-means. 54](#_Toc389132521)

[4.4.3 Thuật toán trên ảnh. 57](#_Toc389132522)

[KẾT LUẬN. 58](#_Toc389132523)

[TÀI LIỆU THAM KHẢO. 59](#_Toc389132524)

# MỞ ĐẦU

Ngày nay, cùng với sự phát triển mạnh mẽ của khoa học và công nghệ thì chúng ta cũng phải đối mặt với nhiều bài toán phức tạp với dữ liệu lớn, đòi hỏi khả năng xử lý mạnh mẽ theo thời gian thực. Một trong những vấn đề quan trọng của lớp bài toán này là chương trình mất rất nhiều thời gian cho công việc xử lý dữ liệu nếu dùng cách tính toán tuần tự thông thường. Đã có nhiều nghiên cứu nhằm mục đích tăng hiệu suất cho các bài toán trên và một trong những phương pháp hiệu quả được sử dụng nhiều hiện nay là sử dụng tính toán song song sử dụng GPU (Graphic Processing Unit).

GPU thường được biết đến như là một thành phần chuyên xử lý đồ họa cho máy tính, tuy nhiên đến khi GPU NV30 của NVIDIA được ra mắt thì GPU bắt đầu hỗ trợ việc tính toán thông thường. Với khả năng xử lý song song mạnh mẽ thì GPU đã dần thay thế cho CPU trong các hệ thống tính toán hiệu năng cao, và thêm với giá thành của GPU không hề cao thì trong tương lai nó có thể hoàn toàn loại bỏ CPU ra khỏi lĩnh vực này.

Vậy tính toán song song là gì, làm thế nào để có thể sử dụng GPU trong tính toán song song, kiến trúc song song nào đang phổ biến hiện nay? Câu trả lời sẽ được trình bày trong đồ án này “**Tính toán song song trên GPU** **và ứng dụng**”. Đồ án gồm các chương chính sau:

* Chương 1: Giới thiệu tổng quan về tính toán song song.
* Chương 2: Tổng quan về gpu và tính toán song song gpu.
* Chương 3: Kiến trúc tính toán song song CUDA.
* Chương 4: Ứng dụng vào một số bài toán.

**Đối tượng nghiên cứu:**

* Tìm hiểu lý thuyết tính toán song song trên GPU.
* Tìm hiểu về CUDA.
* Tìm hiểu một số thuật toán.
* Xây dựng chương trình cho các thuật toán song song trên CUDA.

**Phạm vi nghiên cứu:** Công nghệ tính toán song song CUDA và ứng dụng vào của nó vào thực tế thông qua chuyển đổi một số bài toán từ tuần tự sang song song để so sánh hiệu năng giữa CPU và GPU.

# DANH SÁCH THUẬT NGỮ THƯỜNG SỬ DỤNG

|  |  |  |
| --- | --- | --- |
| 1 | API | Application Program Interface: một API định nghĩa một giao diện chuẩn để triệu gọi một tập các chức năng. |
| 2 | Block | Khối luồng, tập các luồng xử lý |
| 3 | CPU | Bộ xử lý trung tâm |
| 4 | CUDA | Compute Unified Device Architecture (kiến trúc tính toán không đồng nhất ) |
| 5 | Device | Được sử dụng trong CUDA, chỉ GPU |
| 6 | Global memory | Bộ nhớ toàn cục |
| 7 | GPU | Bộ xử lý đồ họa |
| 8 | Grid | Lưới chứa các khối luồng |
| 9 | Host | Được sử dụng trong CUDA, chỉ CPU |
| 10 | Kernel | Hạt nhân |
| 11 | MIMD | Multiple Instruction Multiple Data: đa lệnh đa dữ liệu |
| 12 | Multiprocessor | Bộ đa xử lý: một tập hợp các bộ xử lý |
| 13 | Proccessor | Bộ xử lý |
| 14 | RAM | Bộ nhớ truy cập ngẫu nhiên |
| 15 | Shared memory | Bộ nhớ chia sẻ |
| 16 | SIMD | Single Instruction Multiple Data: đơn lệnh đa dữ liệu |
| 17 | Streaming processor | Bộ xử lý dòng |
| 18 | Thread | Luồng xử lý |
| 19 | Warp | Mỗi khối được tách thành các nhóm SIMD của các luồng |

# DANH MỤC HÌNH ẢNH

[Hình 1: Kiến trúc SIMD. 10](#_Toc389132525)

[Hình 2: Kiến trúc MIMD. 11](#_Toc389132526)

[Hình 3: Kiến trúc GPU Geforce 6 Series. 20](#_Toc389132527)

[Hình 4: Hình ảnh trong game Batman: Arkham Asylum. 23](#_Toc389132528)

[Hình 5: So sánh hiệu quả tăng tốc trên một số ứng dụng. 24](#_Toc389132529)

[Hình 6: Siêu máy tính TITAN sử dụng card đồ họa của NVIDIA. 25](#_Toc389132530)

[Hình 7: Môi trường phát triển ứng dụng của CUDA. 26](#_Toc389132531)

[Hình 8: So sánh cấu trúc CPU và GPU 27](#_Toc389132532)

[Hình 9: Mô tả cấu trúc của multiprocessor. 28](#_Toc389132533)

[Hình 10: Tổ chức luồng trong CUDA 31](#_Toc389132534)

[Hình 11: Tổ chức bộ nhớ trong CUDA. 32](#_Toc389132535)

[Hình 12: Thời gian tồn tại của các biến thuộc từng loại bộ nhớ. 33](#_Toc389132536)

[Hình 13: Thuật toán tuần tự nhân hai ma trận. 40](#_Toc389132537)

[Hình 14: Nhân hai ma trận sử dụng shared memory. 41](#_Toc389132538)

[Hình 15: Thuật toán song song nhân hai ma trận. 43](#_Toc389132539)

[Hình 16: Thuật toán k-means tuần tự. 46](#_Toc389132540)

[Hình 17: Thuật toán song song tìm ma trận khoảng cách. 47](#_Toc389132541)

[Hình 18: Thuật toán song song tìm khoảng cách gần nhất. 47](#_Toc389132542)

[Hình 19: Thuật toán tìm ma trận khoảng cách sử dụng shared memory. 49](#_Toc389132543)

[Hình 20: Thuật toán song song chuyển ảnh màu sau ảnh xám. 51](#_Toc389132544)

[Hình 21: Cách tính chập hai ma trận. 52](#_Toc389132545)

[Hình 22: Thuật toán filter trên ảnh. 53](#_Toc389132546)

[Hình 23: Đồ thị so sánh thời gian chạy chương trình nhân ma trận. 54](#_Toc389132547)

[Hình 24: Đồ thị so sánh thời gian chạy trên tập dữ liệu 1024\*1024 56](#_Toc389132548)

# DANH MỤC BẢNG BIỂU

[Bảng 1: Thời gian chạy chương trình nhân ma trận 54](#_Toc389132549)

[Bảng 2: Kết quả chạy thuật toán K-Means 55](#_Toc389132550)

[Bảng 3: Thời gian thực nghiệm thuật toán. 57](#_Toc389132551)

[Bảng 4: Thời gian chạy thuật toán filter. 57](#_Toc389132552)

# CHƯƠNG 1. GIỚI THIỆU TỔNG QUAN VỀ TÍNH TOÁN SONG SONG

## Khái niệm về tính toán song song.

Các phần mềm phổ biến ngày nay hầu hết đều được viết trên cơ sở của tính toán tuần tự. Các phần mềm này thường được thực hiện trên một máy tính với duy nhất một bộ xử lý. Chương trình giải quyết bài toán thông qua một chuỗi các lệnh được thực hiện tuần tự bởi một bộ xử lý. Tại một thời điểm chỉ có một lệnh được thực thi.

Tính toán song song là sự cải tiến của tính toán tuần tự, được hiểu một cách đơn giản là thực hiện nhiều phép toán trong cùng một thời điểm thay vì thực hiện lần lượt chúng. Tính toán song song được thiết kế dựa trên nguyên tắc chia một bài toán lớn thành những phần nhỏ hơn để giải quyết. Các thuật toán song song thường khó hơn thuật toán tuần tự vì phải xử lý phân chia công việc, xử lý tương tranh giữa các luồng xử lý. Việc quản lý liên kết cũng như đồng bộ giữa các luồng xử lý tốt sẽ tạo nên một chương trình song song hiệu quả.

Hiện nay chúng ta các hệ thống máy tính song song thường được tổ chức dựa trên hai dạng sau:

* Đầu tiên là các máy tính có bộ xử lý đa nhân, các tài nguyên được đồng bộ trong một thiết bị đơn lẻ. Các siêu máy tính cũng tuân theo mô hình này. Các công nghệ được sử dụng trong các máy tính loại này thường có giá thành cao nhưng bù lại thì hiệu năng lại cực kỳ cao.
* Thứ hai là hệ thống song song sử dụng nhiều máy tính đơn lẻ được kết nối với nhau. Trong mô hình này mỗi máy tính cá nhân được coi như 1 nút xử lý của hệ thống, dùng kết nối mạng để liên lạc với nhau. Một máy trong hệ thống sẽ được coi là “chỉ huy” có trách nhiệm phân chia công việc cho các máy con, tổng hợp kết quả nhận được. Mô hình song song này có giá thành rẻ và tính mở rộng cao tuy nhiên thì hiệu năng của nó cũng chỉ thuộc loại trung bình, thường được sử dụng vao những bài toán không yêu cầu xử lý thời gian thực.

## Phân loại kiến trúc song song.

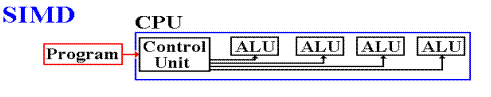
Một trong những phân loại kiến trúc máy tính song song được biết đến nhiều nhất là phân loại của Flynn, được sử dụng từ năm 1966. Michael Flynn dựa vào đặc tính về số lượng bộ xử lý, số chương trình thực hiện, cấu trúc bộ nhớ, v.v… để phân loại kiến trúc song song dựa trên cặp khái niệm: dòng lệnh (instruction stream) và dòng dữ liệu (data stream).

* SIMD - Single Instruction Multiple Data: Đơn chỉ lệnh đa dữ liệu.
* MIMD - Multiple Instruction Multiple Data: Đa chỉ lệnh đa dữ liệu.
* SISD - Single Instruction Single Data: Đơn lệnh đơn dữ liệu.
* MISD - Multiple Instruction Single Data: Đa lệnh đơn dữ liệu.

Hiện nay chỉ có 2 kiến trúc song song được dùng nhiều đó là SIMD và MIMD, còn MISD hiếm khi gặp, SISD thực chất là một chương trình tuần tự.

### 1.2.1 Kiến trúc đơn lệnh đa dữ liệu SIMD.

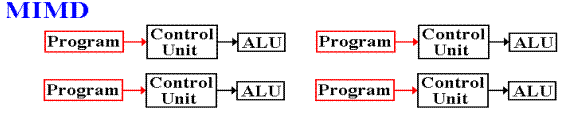
SIMD được hiểu là tất cả các luồng xử lý công việc đều thực hiện 1 lệnh giống nhau nhưng trên các bộ dữ liệu khác nhau. Mô hình này thường được áp dụng hiệu quả trong những bài toán với dữ liệu dạng vector như xử lý ảnh, âm thanh, dữ liệu đa phương tiện…



Hình 1: Kiến trúc SIMD.

### 1.2.2 Kiến trúc đa lệnh đa dữ liệu MIMD.

Đây là mô hình song song trong đó tại bất kỳ thời điểm nào, mỗi một luồng xử lý một lệnh khác nhau với một bộ dữ liệu khác nhau một cách không đồng bộ và độc lập. Các điểm này làm cho MIMD rất linh hoạt trong việc xây dựng các chương trình song song. Tuy nhiên sự linh hoạt của bài toán song song vẫn phụ thuộc phần lớn vào trình độ chuyên môn của lập trình viên. MIMD thường được dùng trong các bài toán mô phỏng, mô hình, chuyển mạch thông tin liên lạc, thiết kế/ hỗ trợ sản xuất máy tính.



Hình 2: Kiến trúc MIMD.

Ngoài kiểu phân loại của Flynn chúng ta còn có một kiểu khác dựa trên cơ chế sử dụng bộ nhớ, tương tác giữa các bộ xử lý.

### 1.2.3 Sử dụng bộ nhớ chia sẻ.

Trong loại kiến trúc này các bộ xử lý dùng chung một bộ nhớ chia sẻ (shared memory). Các bộ xử lý có thể trao đổi thông tin thông qua bộ nhớ chia sẻ. Tuy nhiên có một hạn chế là cần phải xử lý trường hợp nhiều bộ xử lý cùng truy cập đến một vị trí. Máy tính chia sẻ bộ nhớ chung có thuận lợi là giúp cho lập trình người thuận tiện khi viết các chương trình song song. Dữ liệu chia sẻ giữa các nhiệm vụ đảm bảo cả hai tiêu chuẩn nhanh và đồng thời. Tuy nhiên máy tính loại này có một số khó khăn là rất khó mở rộng số lượng các bộ xử lý vì việc thêm các bộ xử lý về phương diện hình học có thể làm tăng các đường kết nối giữa bộ nhớ toàn cục và các bộ xử lý.

### 1.2.4 Sử dụng bộ nhớ phân tán.

Ngược với máy tính chia sẻ bộ nhớ chung là máy tính với bộ nhớ phân tán trong đó không tồn tại bộ nhớ chia sẻ chung mà mỗi bộ xử lý có bộ nhớ cục bộ riêng của chúng. Trong máy tính song song có bộ nhớ phân tán các bộ xử lý liên lạc với nhau bằng các thông điệp (message) qua một mạng liên kết (interconnection network) gồm các liên kết truyền thông trực tiếp giữa một số cặp bộ xử lý. Một trong những lựa chọn quan trọng trong thiết kế lúc đó sẽ là các cặp bộ xử lý nào được nối với nhau.Tốc độ liên lạc là tối ưu khi các bộ xử lý được nối trực tiếp với nhau.Tuy nhiên điều này thường là không khả thi do số lượng các liên kết là quá lớn giẫn đến việc tăng giá thành của hệ thống. Cách thứ hai được sử dụng là các bộ sử lý liên lạc thông qua một kênh chia sẻ (BUS). Ngày nay, các hệ thống máy tính lớn thường kết hợp hai loại trên nhằm tối ưu về hiệu suất cũng như giá thành.

## Định luật Amdahl.

Định luật Amdahl: Gọi P là phần việc có thể thực hiện song song, (1-P) sẽ là phần việc tuần tự, N là số lõi xử lý thì tốc độ tối đa S có thể đạt được được tính theo công thức:



Khi số lõi tiến đến vô cùng thì công thức chỉ còn phụ thuộc vào cách phân chia công việc song song của thuật toán. Luật Amdahl, còn có tên Đối số Amdahl, mang tên của kiến trúc sư máy tính Gene Amdahl. Nó được dùng để tìm ra sự cải thiện tối đa (theo mong đợi) của một hệ thống tổng thể khi chỉ cải thiện một phần của hệ thống. Nó còn được sử dụng trong tính toán song song để dự đoán sự tăng tốc trên lí thuyết khi sử dụng nhiều bộ xử lí (đa xử lí).

Luật Amdahl là một mô hình của mối quan hệ giữa sự tăng tốc mong đợi của việc triển khai song song của một thuật toán với phần tuần tự của thuật toán, với giả thiết rằng kích thước của bài toán không đổi khi tính toán song song. Ví dụ với kích thước bài toán cho trước phần thực hiện song song của thuật toán là 12%, có thể thực thi với tốc độ tuỳ ý, trong khi 88% còn lại không thể thực hiện song song, luật Amdahl cho biết rằng tốc độ tối đa của việc tính toán song song với thuật toán này chỉ nhanh hơn 1/(1 - 0,12) = 1,136 lần so với khi không có tính toán song song.

## 1.4 Mô hình lập trình song song cơ bản.

Việc đưa ra một mô hình máy tính chung cho việc lập trình giúp cho việc thiết kế giải thuật giải thuật trở nên đơn giản hơn. Lập trình song song đưa thêm những khó khăn mới vào mô hình lập trình tuần tự, nếu chương trình được thực hiện ở mức thấp nhất thì không những số lệnh thực hiện là rất lớn mà nó còn phải quản lý trực tiếp quá trình thực hiện song song của hàng nghìn BXL và kết hợp hàng triệu tương tác liên BXL. Bởi vậy việc thiết lập mô hình và tính toán module là các đặc tính rất quan trọng trong lập trình song song. Vậy mô hình nào sẽ phù hợp với lập trình song song. Các mô hình này cần cho phép đánh giá cụ thể về khả năng thực hiện đồng thời cũng như tính cục bộ để cho phép phát triển các chương trình có tính modul và có khả năng mở rộng. Và mô hình đó phải phù hợp với kiến trúc của máy tính song song.

Các mô hình thông dụng bao gồm:

* Mô hình chia sẻ bộ nhớ chung.
* Mô hình luồng.
* Mô hình truyền thông điệp.
* Mô hình song song dữ liệu.
* Mô hình lai.

### 1.4.1 Mô hình chia sẻ bộ nhớ chung.

Trong mô hình chia sẻ bộ nhớ chung các nhiệm vụ cùng chia sẻ một không gian địa chỉ chung có thể được truy cập đọc ghi theo phương thức không đồng bộ. Các cơ chế khác nhau được điều khiển để truy cập đến bộ nhớ toàn cục. Xét theo quan điểm của lập trình viên thì ưu điểm của mô hình này là không có khái niệm sở hữu dữ liệu. Nghĩa là không phải chỉ định rõ ràng quá trình truyền dữ liệu giữa nhiệm vụ gửi và nhiệm vụ nhận dữ liệu. Tính chất này giúp cho phát triển các chương trình đơn giản hơn. Tuy nhiên khi đó việc hiểu và đảm bảo tính cục bộ trở nên khó khăn và cũng được chú ý nhiều nhất trong kiến trúc chia sẻ bộ nhớ chung.

### 1.4.2 Mô hình luồng.

Trong mô hình luồng chương trình chính được chia thành các nhiệm vụ. Mỗi nhiệm vụ được thực hiện bởi các luồng một cách đồng thời. Mỗi một luồng có dữ liệu riêng của nó và chia sẻ dữ liệu toàn cục của chương trình chính. Các nhiệm vụ đưa cho mỗi luồng là các thủ tục con của chương trình chính. Và bất kì luồng nào cũng có thể thực hiện bất kì thủ tục con nào tại cùng thời điểm với các luồng khác. Trong mô hình luồng các luồng kết nối với nhau thông qua bộ nhớ toàn cục với việc kết nối này thì chương trình phải được xây dựng một cách đồng bộ để tránh cùng một lúc có nhiều luồng cùng cập nhập một vị trí trong bộ nhớ.

### 1.4.3 Mô hình truyền thông điệp.

Trong mô hình truyền thông điệp chương trình song song được chia thành các nhiệm vụ, mỗi nhiệm vụ sử dụng bộ nhớ cục bộ của nó. Các nhiệm vụ này có thể được cư trú trên các máy vật lý giống nhau kết nối với nhau qua mạng với số lượng tùy ý. Các nhiệm vụ trao đổi dữ liệu với nhau qua hai phương thức gửi và nhận thông điệp. Xét trên khía cạnh lập trình thì các thông điệp chứa trong một thư viện thông điệp.

### 1.4.4 Mô hình song song dữ liệu.

Mô hình lập trình song song dữ liệu giúp lập trình các chương trình song song được thực hiện trên một tập dữ liệu lớn. Tập dữ liệu ở đây thường được xắp xếp theo một cấu trúc nhất định như là mảng hoặc theo khối. Với mô hình này thì các nhiệm vụ của chương trình làm việc với cùng một cấu trúc dữ liệu. Tuy nhiên mỗi nhiệm vụ sẽ làm việc trên từng phân vùng khác nhau của dữ liệu và các nhiệm vụ phải thưc hiện các thao tác giống nhau. Trong kiến trúc chia sẻ bộ nhớ chung thì tất cả các nhiệm vụ truy cập vào cấu trục dữ liệu thông qua bộ nhớ toàn cục. Còn đối với kiến trúc bộ nhớ phân tán thì dữ liệu được chia ra và lưu trữ trên các bộ nhớ cục bộ của các bộ xử lý.

### 1.4.5 Mô hình lai.

Mô hình lai là sự kết hợp của hai hay nhiều mô hình lập trình song song kết hợp lại với nhau. Hiện nay thì mô hình lai phổ biến nhất là mô hình kết hợp giữa mô hình truyền thông điệp với mô hình luồng hoặc với mô hình chia sẻ bộ nhớ chung. Một mô hình lai khác nữa là sự kết hợp giữa mô hình song song dữ liệu với mô hình truyền thông điệp. Mô hình dạng này rất thuận tiện vì mô hình song song dữ liệu trên kiến trúc bộ nhớ phân tán sử dụng message passing để trao đổi dữ liệu giữa các nhiệm vụ một cách trong suốt đối với lập trình viên song song.

### 1.4.6 Tắc nghẽn (Deadlock)

Tắc nghẽn là tình trạng một tiến trình nắm giữ tài nguyên lâu dài làm cho các tiến trình khác có nhu cầu sử dụng luôn trong tình trạng đợi. Lý do tồn tại sự tắc nghẽn là do nhiều tiến trình cùng sử dụng tài nguyên chung mà không có sự kiểm soát tốt. Sự tắc nghẽn tồn tại trong các hệ điều hành đa nhiệm, cũng như các hệ thống đa bộ xử lý và đa máy tính.

Bốn điều kiện gây nên bế tắc:

* Sự loại trừ lẫn nhau: Mỗi tiến trình có sự độc quyền khi sử dụng tài nguyên của nó.
* Không có sự ưu tiên: Mỗi tiến trình không bao giờ giải phóng tài nguyên mà tiến trình đó đang chiếm giữ.
* Sự chờ đợi tài nguyên: Tiến trình không thực hiện tiếp cho đến khi nhận được tài nguyên.
* Sự chờ đợi giữa các tiến trình: Tiến trình đợi tài nguyên mà tiến trình kế tiếp đang chiếm dữ mà tài nguyên đó không được giải phóng.

Một số phương pháp xử lý:

* Ngăn ngừa, đảm bảo hệ thống không đi vào tình trạng tắc nghẽn.
* Cho phép hệ thống khôi phục lại từ trạng thái tắc nghẽn.
* Bỏ qua trạng thái tắc nghẽn.

# Chương 2. TỔNG QUAN VỀ GPU VÀ TÍNH TOÁN SONG SONG GPU.

## 2.1 Tổng quan về GPU.

Bộ xử lý đồ họa GPU(Graphic Processing Unit) là bộ phận chuyên dụng đảm nhận công việc tăng tốc, xử lý các tác vụ đồ họa trên máy tính. Các GPU hiện đại có hiệu suất cực cao trong xử lý đồ họa, với cấu trúc song song mạnh mẽ thì tính toán trên GPU tỏ ra hiệu quả hơn trên CPU trong nhiều thuật toán phức tạp.

Hiện nay GPU thường xuất hiện trong các hệ thống nhúng, máy tính cá nhân, máy trạm, máy chơi game, điện thoại di động. Trong các hệ thống máy tính GPU có thể được gắn sẵn trên bo mạch chủ hoặc nằm trên card đồ họa gắn ngoài.

## 2.2 Lịch sử phát triển.

* 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.
* Năm 1980, Commodore Amiga là máy tính thương mại đầu tiên có chứa các bộ blit (**B**lock **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.
* Năm 1991, S3 Graphics giới thiệu bộ gia tốc chip 2D đầu tiên, trong suốt những năm 90 bằng vào sự phát triển của công nghệ việc tích hợp chip tăng tốc đồ họa trở nên phổ biến. Thêm vào đó là sự xuất hiện các 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. OpenGL xuất hiện vào thời điểm này như là một giao diện lập trình 3D chuyên nghiệp, và trở thành một thành phần quan trọng trong các ứng dụng đồ họa cho đến nay.
* 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.
* Từ năm 2000 sự ra đời của 2 thương hiệu là NVIDIA và ATI đã đánh dấu bước phát triển mới của GPU.
* Năm 2007 đến nay, NVIDIA ra mắt nền tảng tính toán song song CUDA trên GPU, tạo một bước đột phá mang ý nghĩa to lớn trong lĩnh vực tính toán song song.

## Kiến trúc GPU.

GPU là bộ phận xử lý tập trung vào tính toán, trong những năm gần đây GPU phát triển từ một thành phần chuyên dụng xử lý đồ họa trở thành bộ xử lý cho các mục đích phổ thông. Trong phần này trình bày về kiến trúc của GPU và khía cạnh giúp cho GPU trở thành bộ xử lý song song cho các ứng dụng. Để nắm rõ hoàn toàn được cơ chế, kiến trúc của GPU là rất khó vì vậy trong phần này chỉ trình bày những thành phần cơ bản liên quan đến lập trình song song trên GPU.

### 2.3.1 Graphics Pipeline (Đường ống đồ họa )

Đầu vào của bộ xử lý đồ họa là tập những hình đa giác nguyên thủy, ví dụ như tam giác trong một không gian 3 chiều. Qua một vài bước những hình khối nguyên thủy được xử lý và vẽ lên màn hình theo từng điểm ảnh một.

Các khối 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, thông thường bằng cách tính toán tương tác của chúng với các luồng ánh sáng trong một bối cảnh cụ thể. Với 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 kiều dữ liệu này là rất phù hợp cho phần cứng song song.

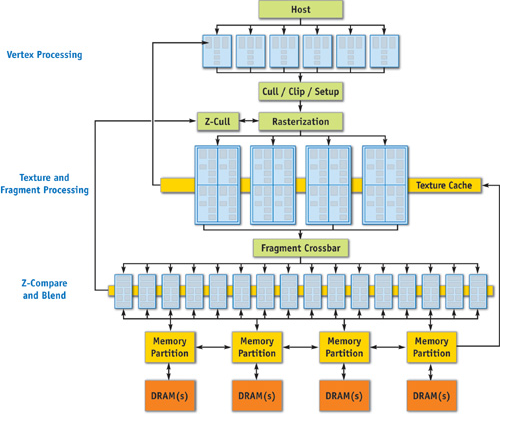
Sự quét mảnh (rasterization) là quá trình chyển đổi các vector thành các điểm ảnh phục vụ cho việc hiển thị hình ảnh lên màn hình, in ấn hoặc lưu trữ. Quét mảnh xác định những vị trí điểm ảnh nào trong không gian màn hình. Mỗi tam giác tạo ra một thành tố nguyên thủy được gọi là "mảnh" tại các vị trí 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. Sử dụng thông tin màu sắc từ vector và lấy dữ liệu bổ sung từ bộ nhớ toàn cục mỗi mảnh được làm bóng mờ (shade) để xác định màu sắc cuối cùng của nó. Cũng như trong xử lý 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. 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.

### Kiến trúc của GPU hiện đại.

Kiến trúc của GPU tập trung vào năng lực xử lý song song do vậy mục tiêu chính của tính toán trên GPU không phải là giảm độ trễ truy cập mà là tăng băng thông. Xem xét quá trình xử lý trên đường ống ta thấy rằng ở hầu hết các giao diện lập trình phải xử lý một lượng lớn các dữ liệu đầu vào. Trong một đường ống dẫn như vậy, đầu ra của mỗi lần xử lý thành công là 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.

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.

Trong CPU, bất kỳ phép toàn nào cũng có thể mất khoảng 20 chu kỳ hoạt động theo thứ tự tính từ lúc bắt đầu đến khi rời khỏi đường ống CPU. Trên GPU, một phép toán đồ họa cho trước có thể mất hàng ngàn chu kỳ từ khi bắt đầu đến khi kết thúc. Độ trễ của các phép toán trên GPU thường là lâu. Tuy nhiên, cơ chế song song tác vụ và dữ liệu của GPU tạo ra thông lượng cao bù lại cho độ trễ xử lý lớn. 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 tác vụ chậm nhất của nó. Nếu các chương trình vector rất phức tạp và chương trình quét mảnh là đơn giản thì tổng thể hiệu suất thông qua là phụ thuộc vào hiệu suất của các chương trình xử lý vector. Ngày đầu các tác vụ vector và quét mảnh tách biệt thì tập chỉ thị của hai phần này khá là khác nhau. Đến nay, khi chương trình cho hai tác vụ này trở nên hoàn thiện thì tập chỉ thị dần có những cải thiện, kiến trúc GPU được xem xét thiết kế lại lại. Các đơn vị lập trình trong đường ống được chia sẻ một đơn vị xử lý duy nhất, các đơn vị phân chia thời gian thực hiện tác vụ xử lý vector, quét mảnh và các công việc hình học. Cơ chế song song tác vụ và song song dữ liệu được khai thác một cách triệt để.



Hình 3: Kiến trúc GPU Geforce 6 Series.

## 2.4 Mô hình lập trình tính toán trên GPU.

Lập trình trên GPU tuân theo mô hình SPMD (single program, multiple data): đơn chương trình, đa dữ liệu. Để đạt hiệu quả tối ưu, GPU xử lý nhiều yếu tố song song bằng cách sử dụng nhiều chương trình giống nhau. Mỗi phần tử là độc lập và không thể giao tiếp với nhau. Tất cả các chương trình trên GPU được tổ chức theo cách: chia nhỏ ra thành nhiều phần, mỗi thành phần được xử lý như một chương trình độc lập.

Một trong những ưu điểm của GPU là phần lớn tài nguyên phần cứng được dành cho việc tính toán. GPU cho phép các phần tử thực thi với các con đường khác nhau nhưng đòi hỏi tài nguyên khá đáng kể. Trong khi thiết kế chương trình trên GPU thì rẽ nhánh được phép nhưng phải trả một giá nhất định. Người lập trình pahir tổ chức mã nguồn của họ sao cho khối có rẽ nhánh mạch lạc sẽ tận dụng phần cứng tốt nhất. Để tìm hiểu mô hình lập trình tính toán trên GPU ta xem xét từng vấn đề sau:

#### 2.4.1.1 Lập trình đồ họa trên GPU

* Lập trình viên xác định dạng hình học sẽ hiển thị trên màn hình, quá trình quét mảnh trên màn hình tạo ra một mảnh ở vị trí được bao phủ bởi hình đó.
* Mỗi mảnh được xử lý shade, các chương trình quét mảnh tính giá trị của các mảnh đó.
* Các hình ảnh kết quả sau đó được đưa ra màn hình.

#### 2.4.1.2 Lập trình GPGPU(tính toán thông thường trên GPU)

Một trong những khó khăn trong lịch sử lập trình ứng dụng GPGPU đó là mặc dù các tác vụ thông thường không có liên quan gì tới đồ họa, các ứng dụng vẫn phải được lập trình bằng cách sử dụng các hàm giao tiếp đồ họa. Ngoài ra, chương trình được cấu trúc trong điều kiện của đường ống đồ họa, với các đơn vị lập trình 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. Ngày nay, các ứng dụng tính toán trên GPU được tổ chức theo cách:

* Chia nhỏ thành các chương trình con độc lập thành một lưới các luồng.
* Các chương trình con được xử lý song song bởi các luồng.
* Giá trị mà mỗi luồng tính toán được là kết quả của các phép tính toán học và dữ liệu được lưu trữ trên bộ nhớ toàn cục.
* Các vùng đệm có thể lưu trữ kết quả tạm thời cho các tính toán sau đó.

Mô hình này tỏ ra khá mạnh mẽ và linh hoạt giúp giảm thời gian phát triển ứng dụng của lập trình viên. 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. Hơn nữa 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. 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.

## 2.5 Một số ứng dụng thực tế.

### 2.5.1 Game trên máy tính.

Một trong những ứng dụng phổ biến của GPU là trong nền công nghiệp game. Với GPU việc xử lý hình ảnh trong game trở nên nhanh và hiệu quả hơn CPU rất nhiều.

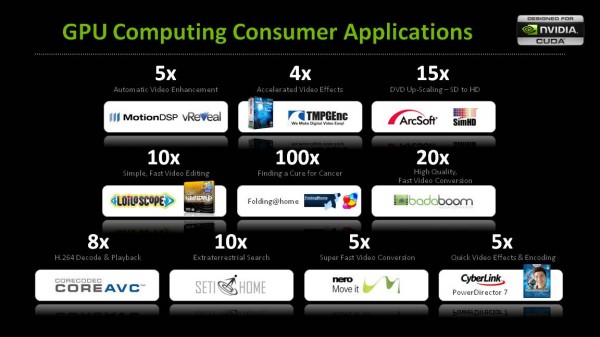


Hình 4: Hình ảnh trong game Batman: Arkham Asylum.

Hình trên là game “Batman: Arkham Asylum”, đây là game trên máy tính được thế giới đánh giá cao với chất lượng đồ họa tuyệt vời. Game sử dụng bộ công cụ PhysX của NVIDIA chuyên tính toán hiệu ứng vật lý như khói, lửa, vụ nổ, kính vỡ… PhysX hỗ trợ hầu hết các nền tảng máy chơi game phổ biến hiện nay, và tất nhiên hỗ trợ NVIDIA CUDA. Với tốc độ phát triển của phần mềm lẫn phần cứng như hiện nay thì chẳng bao lâu nữa thì đồ họa game càng ngày gần với thực tế sẽ trở thành hiện thực.

### 2.5.2 Tăng tốc các ứng dụng.

Công nghệ CUDA còn được đưa vào trong các phần mềm đòi hỏi khối lượng tính toán lớn như xử lý ảnh, xử lý video, âm thanh.



Hình 5: So sánh hiệu quả tăng tốc trên một số ứng dụng.

Bên trên là 1 vài những ứng dụng được so sánh. Con số ghi bên trên mỗi ứng dụng là tỉ lệ tăng tốc giữa CUDA chạy trên card NVIDIA GTX285, còn của CPU là chip Intel Core 2 Duo E8200. Ta có thể thấy được sự khác biệt sự tăng tốc rõ rệt giữa CPU và GPU.

### 2.5.3 Sử dụng trong nghiên cứu.

Với khả năng tăng tốc độ tính toán một cách đáng kể của GPU còn được sử dụng phục vụ cho tính toán khoa học, mô phỏng. Những hệ thống tính toán nổi tiếng sử dụng GPU phải kể đến các siêu máy tính



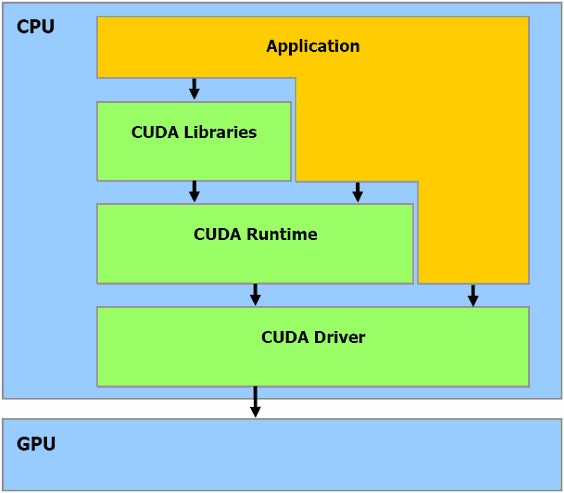
Hình 6: Siêu máy tính TITAN sử dụng card đồ họa của NVIDIA.

Ví dụ như siêu máy tính đứng thứ 2 trên thế giới vào thời điểm hiện tại TITAN của Mỹ được tạo nên từ 18 688 node mỗi node gồm 1 CPU AMD Opteron 6274 16 core với 32 GiB RAM, 1 GPU Nvidia Tesla K20X 6GiB. TITAN là siêu máy tính đầu tiên được xây dựng trên GPU mà hiệu năng vượt qua 17 petaFLOPS(17 triệu tỉ phép tính trên giây). Siêu máy tính TITAN được dùng trong nhiều lĩnh vực như y sinh, vật lý, thiên văn, mô phỏng thiên tai, dự báo thời tiết …

# Chương 3. KIẾN TRÚC TÍNH TOÁN SONG SONG CUDA.

## 3.1. Giới thiệu môi trường phát triển ứng dụng

CUDA- viết tắt của Compute Unified Device Architecture (kiến trúc tính toán không đồng nhất ), 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 sử dụng ngôn ngữ C, không cần quan tâm đến các hàm xử lý đồ họa của GPU.



Hình 7: Môi trường phát triển ứng dụng của CUDA.

Môi trường phát triển ứng dụng của CUDA cung cấp driver cho card màn hình, API, bộ biên dịch, các thư viện hỗ trợ giúp lập trình viên dễ dàng hơn trong thiết kế chương trình song song.

## 3.2. Kiến trúc phần cứng

Kiến trúc của GPU CUDA được xây dựng với mục đích tối ưu hóa các tác vụ song song, vì thế số “lõi” của GPU nhiều hơn đáng kể so với CPU. Thực chất card đồ họa chính là một máy tính thu nhỏ, có một vài thành phần tương tự nhau giũa chúng.



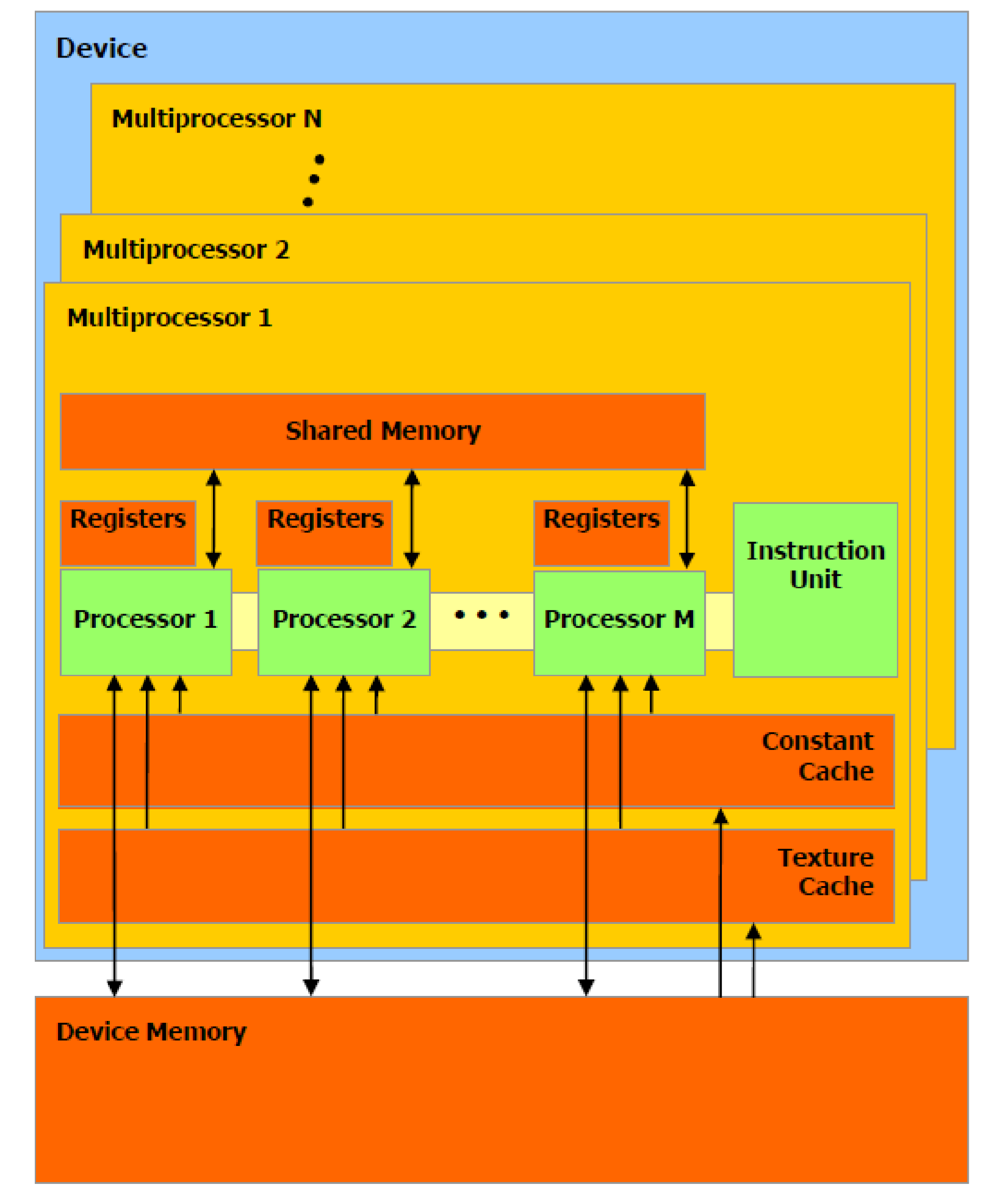
Hình 8: So sánh cấu trúc CPU và GPU

Hình 8 mô tả cấu trúc đơn giản của CPU và GPU, chúng ta có thể thấy được các thành phần tương đương, đều có RAM, cache, bộ điều khiển, bộ xử lý tính toán. Với CPU các nhân xử lý, bộ điều khiển và cache được thiết kế tối ưu cho các công việc xử lý nhanh gọn một vài luồng, còn GPU lại tập trung vào hướng xử lý song song nhiều luồng trong cùng một thời điểm. CPU sử dụng cache để giảm độ trễ truy cập dữ liệu còn GPU sử dụng cache để tăng băng thông của bộ nhớ. Trong vận hành, CPU hiện tại có thể chạy song song 4 – 8 luồng cùng lúc, trong khi đó mỗi “lõi” của GPU có thể chạy 1024 luồng(1 GPU thường có 8 – 16 lõi như vậy). Các lõi của GPU có bộ điều khiển, cache và các đơn vị xử lý độc lập với nhau.

### 3.2.1 Multiprocessor.

Thiết bị GPU được thiết kế dựa trên một tập các multiprocessor như trong hình 3, mỗi multiprocessor được coi như một “lõi” của GPU. Các bộ phận cấu thành multiprocessor gồm có:

* Các nhân tính toán
* Một thanh ghi 32 bit
* Một vùng nhớ đệm chia sẻ (shared memory) được tất cả các bộ xử lý dùng chung
* Vùng nhớ đệm hằng số (constant memory) và texture memory chỉ đọc.



Hình 9: Mô tả cấu trúc của multiprocessor.

### 3.2.2 Cấu trúc SIMT và cách thức thực thi

Để quản lý hàng nghìn thread trong mỗi multiprocessor, CUDA sử dụng một kiến trúc gọi là SIMT(Single Instruction Multi Thread) đơn lệnh đa luồng. Kiến trúc SIMT khác với SIMD ở cách xử lý về cấp phát luồng và bộ nhớ để xử lý dữ liệu còn về bản chất vẫn giống nhau.

Trong CUDA, các luồng được gom vào các nhóm, mỗi nhóm có 32 luồng được gọi là wrap. Các thread trong một wrap được thực thi cùng nhau trong mỗi lần gọi, tuy nhiên các thread vẫn được chạy độc lập với nhau. Khi multiprocessor thực thi một thread block thì nó chia các thread trong block ra thành các wrap rồi đưa các wrap vào bộ sắp xếp lịch cho wrap(wrap scheduler). Cách phân chia block trong mọi trường hợp là giống nhau, mỗi wrap chứa các thread được đánh số liên tiếp nhau, wrap đầu tiên sẽ có thread chỉ số 0.

## 3.3. Kiến trúc lập trình.

### 3.3.1 Kiến trúc lập trình kết hợp.

Trong kiến trúc lập trình của CUDA được chia là 2 phần lớn. Phần host gồm thành phần trên máy tính CPU, RAM. Phần device là card đồ họa dùng để tính toán. Trên host chủ yếu xử lý nhập, điều khiển các thiết bị mở rộng. Phần tính toán được đưa vào device để xử lý, kết quả sẽ được đưa trở lại host. Phần host code được trình dịch của CPU thực hiện bình thường còn phần device code sẽ được trình biên dịch của CUDA phụ trách. Phần host code được trình dịch của hệ điều hành chạy bình thường còn device code được trình dịch riêng của CUDA xử lý.

Để tiện cho việc diễn đạt trong các phần sau ta sử dụng một đoạn code mẫu:

|  |
| --- |
| // Device code  // Kernel definition  \_\_global\_\_ voidVecAdd(float\* A, float\* B, float\* C)  {  inti = threadIdx.x;  C[i] = A[i] + B[i];  }  //host code  Int main()  {  ...  // Kernel invocation with N threads  VecAdd<<<1, N>>>(A, B, C);  ...  } |

### 3.3.2 Hạt nhân chương trình (Kernel).

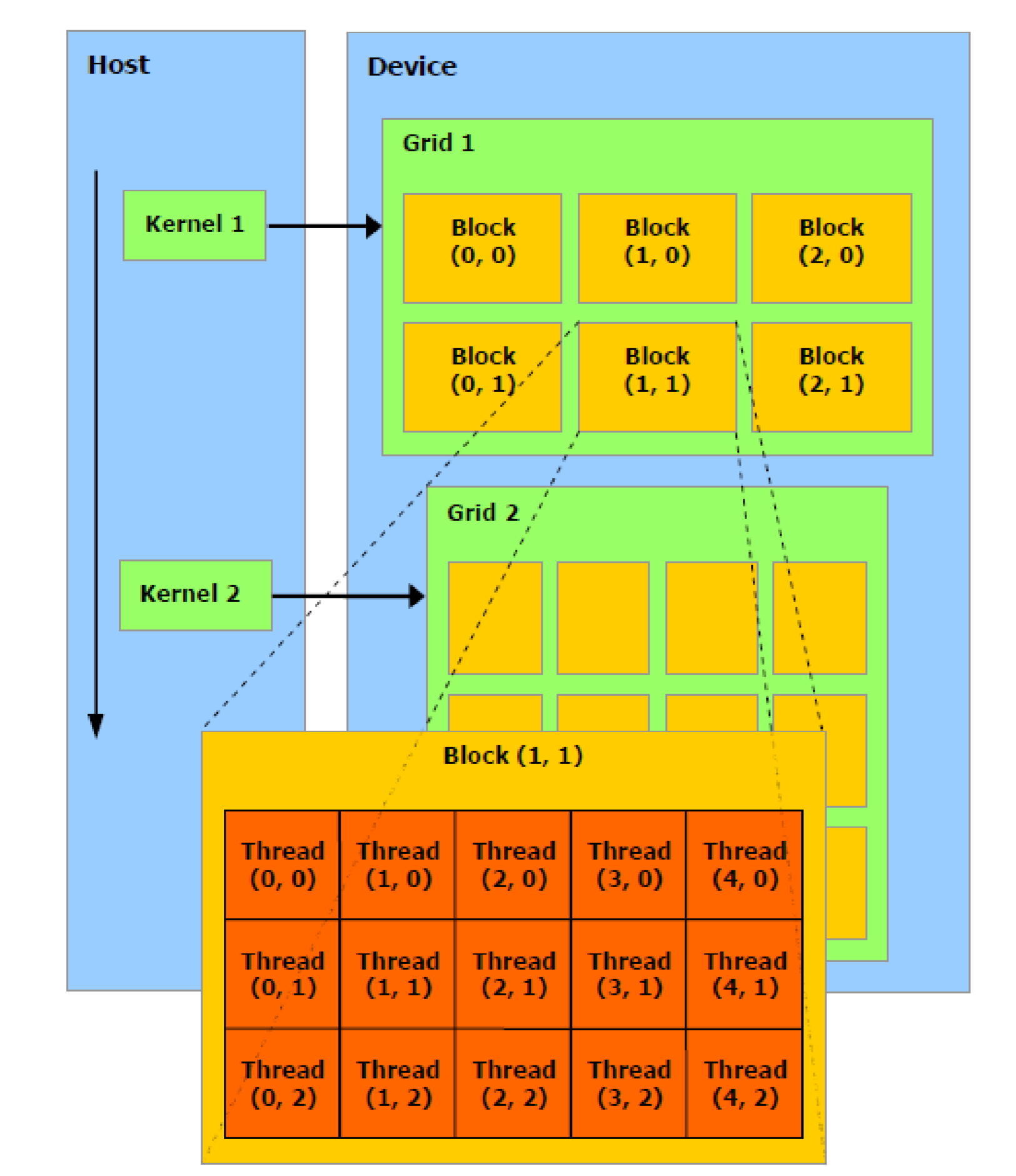
Trong chương trình CUDA, đoạn code song song được thực thi trên device được gọi là kernel. Mỗi kernel được trên device nhiều lần và tại một thời điểm chỉ có 1 kernel được gọi. Trong ví dụ hình 4 thì đoạn code dưới đây là một kernel.

|  |
| --- |
| \_\_global\_\_ voidVecAdd(float\* A, float\* B, float\* C)  {  inti = threadIdx.x;  C[i] = A[i] + B[i];  } |

Từ khóa \_\_global\_\_ và threadIdx được định nghĩa trong 3.4 là những thành phần của CUDA.

### 3.3.3 Luồng, khối luồng, lưới (thread, block, grid)

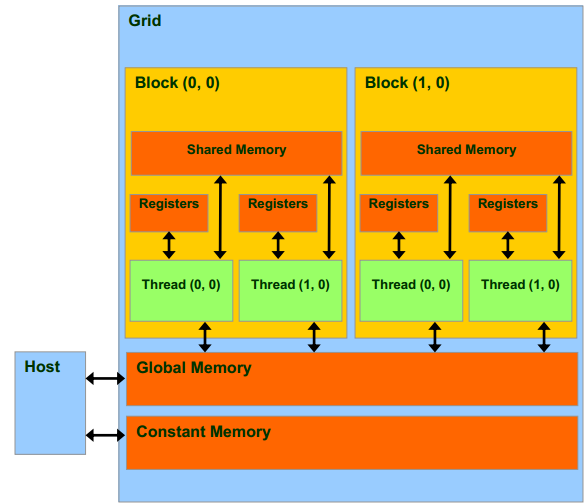
Một luồng trên CUDA là một phần tử cơ bản để xử lý dữ liệu. Mỗi luồng của CUDA được xác định bởi một định danh threadId. Các luồng được nhóm với nhau thành block và các block được nhóm thành grid. Block cũng có định danh là blockId và mỗi lần thực thi chỉ có 1 grid được gọi.



Hình 10: Tổ chức luồng trong CUDA

### 3.3.4 Phân chia bộ nhớ.

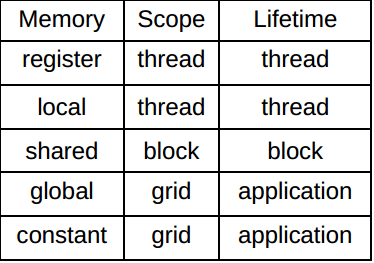
Trong mô hình lập trình thì các xử lý liên quan đến bộ nhớ có tầm quan không khá lớn. Khi dữ liệu chuyển từ host sang device mặc định được đưa vào RAM của device. Mức độ truy cập được miêu ta như hình 11.



Hình 11: Tổ chức bộ nhớ trong CUDA.

Các thread trong một chương trình có thể thực hiện các hành vi sau:

* Đọc và ghi dữ liệu trên global memory.
* Đọc và ghi dữ liệu trên register.
* Đọc và ghi dữ liệu trên shared memory.
* Chỉ đọc dữ liệu trên constant memory.



Hình 12: Thời gian tồn tại của các biến thuộc từng loại bộ nhớ.

Ngoài thời gian tồn tại khác nhau thì tốc độ truy cập trên từng loại bộ nhớ cũng có sự phân biệt. Các bộ nhớ register, shared memory có tốc độ nhanh nhất nhưng dung lượng lại bé nhất còn các bộ nhớ còn lại thì constant memory có tốc độ và dung lượng trung bình còn global memory có tốc độ chậm nhất nhưng dung lượng lớn nhất.

## 3.4. Phần mở rộng của ngôn ngữ lập trình C

Trong phần này sẽ trình bày các thành phần của CUDA được thêm vào ngôn ngữ lập trình C. Các từ khóa mở rộng của CUDA được chi làm các loại chính sau:

* + Từ khóa hàm: dùng để xác định phạm vi của hàm thực thi.
  + Từ khóa biến: xác định kiểu biến, loại bộ nhớ mà biến sử dụng.
  + Ký hiệu xác định cấu hình: xác định số thread, block cần dùng.
  + Các kiểu biến nguyên thủy trong CUDA.

#### Từ khóa hàm

Trong chương trình chúng ta thường gặp các loại từ khóa cho hàm sau:

**1.\_\_global\_\_**

Khai báo \_\_global\_\_ định nghĩa một hàm như là một hạt nhân:

* Chỉ được thực thi trên device
* Chỉ có thể triệu gọi được từ host
* Hàm \_\_global\_\_ là một thủ tục.

**2.\_\_device\_\_**

Khai báo \_\_device\_\_ định nghĩa một hàm:

* Chỉ được thực thi trên device .
* Chỉ được gọi từ device.

**3.\_\_host\_\_**

Khai báo \_\_host\_\_ là một hàm:

* Chỉ được thực thi trên host.
* Chỉ có thể gọi được từ host.
* Có thể kết hợp \_\_host\_\_ và \_\_device\_\_ đề tạo nên 1 hàm được gọi và thực thi trên cả host và device.

**4.\_\_noinline\_\_ và \_\_forceinline\_\_**

Khai báo \_\_ noinline\_\_ và \_\_forceinline\_\_ báo cho bộ biên dịch biết hàm đó có là loại inline hay không.

Mỗi từ khóa hàm được đặt trước định nghĩa hàm đó. Ví dụ:

|  |
| --- |
| \_\_global\_\_ voidVecAdd(float\* A, float\* B, float\* C){…} |

Khi sử dụng tù khóa hàm ta cần lưu ý một số nguyên tắc sau:

* Các hàm của \_\_device\_\_ là hàm đóng (inlined).
* Các hàm của \_\_device\_\_và \_\_global\_\_không hỗ trợ đệ quy.
* Các hàm của \_\_device\_\_và \_\_global\_\_không thể khai báo các biến static trong thân hàm.
* Các hàm của \_\_device\_\_không thể lấy được địa chỉ của chúng; hàm trỏ tới các hàm \_\_global\_\_ thì có thể.
* \_\_global\_\_ và \_\_host\_\_ không thể sử dụng đồng thời. \_\_global\_\_ phải có kiểu trả về là kiểu void.
* Hàm \_\_global\_\_ 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.

#### Từ khóa cho biến.

Tương tự như hàm thì các biến cũng có những từ khóa:

**1.\_\_device\_\_**

Khai báo \_\_device\_\_ định nghĩa biến trên host và biến chỉ được truy cập trên device. Khi ta khai báo từ khóa \_\_device\_\_ thì biến mặc định sẽ có những đặc điểm sau:

* Được tạo ra trên bộ nhớ toàn cục.
* Thời gian tồn tại bằng vòng đời của ứng dụng.
* Có thể truy cập từ tất cả các thread trên device và truy cập từ host bằng   
  thư viện runtime.

**2.\_\_constant\_\_**

Khai báo \_\_constant\_\_ có thể được dùng với khai báo \_\_device\_\_ định nghĩa một biến:

* Tồn tại trong constant memory.
* Thời gian tồn tại bằng vòng đời của ứng dụng.
* Chỉ có thể truy cập bởi các thread trên device.

**3.\_\_shared\_\_**

Khai báo \_\_share\_\_ định nghĩa biến:

* Tồn tại trong shared memory.
* Thời gian tồn tại trong một block.
* Có hai cách tạo biến kiểu shared:

**Tạo trực tiếp với kích thước rõ ràng:**

|  |
| --- |
| \_\_shared\_\_ float shared[256]; |

**Tạo với kích thước không cố định:**

Kích thước của mảng được xác định khi gọi kerne trong phần 3.4.3.

|  |
| --- |
| extern \_\_shared\_\_ float shared[];  \_\_device\_\_ void func(){  short\* array0 = (short\*)array; |

#### 3.4.3 Cấu hình thực thi kernel.

Bất kỳ lời kernel nào từ host đều cần phải có tham số cấu hình đi cùng. Cấu hình lời gọi được đặt trong <<<Dg,Db,Ns,S>>> trong đó:

* Dg: là biến kiểu dim3, xác định kích thước lưới, Dg.z\*Dg.y\*Dg.z bằng số block của lưới.
* Db: biến kiểu dim3, xác định số thread trong một block.
* Ns: biến kiểu int xác định số byte cấp cho một block trên shared memory.
* S: kiểu cudaStrem\_t, xác định số stream kiên kết.

Ví dụ:

|  |
| --- |
| \_\_global\_\_ void Func(float\* parameter);  …  Func<<< Dg, Db, Ns >>>(parameter); |

Lời gọi hàm Func ngoài tham số parameter như bình thường thì còn 3 tham số xác định cấu hình thực thi trên device.

#### 3.4.4. Các dạng vector

* 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 dữ liệu thừa kế từ các kiểu dữ liệu cơ bản trong ngôn ngữ C, chúng có từ 1 đến 4 thành phần và được truy cập qua các trường x, y, z, w. Các vector thuộc dạng này có thể được tạo bằng hàm make\_<type\_name>.

Ví dụ: int2 make\_int2(int x, int y);

* tạo ra một vector với hai giá trị x, y kiểu int.
* dim3: là một vector kiểu dữ liệu int dựa trên uint3 thường được dùng để tạo nhanh một kích thước 3 chiều, trong khởi tạo nếu chỉ có 1 hoặc 2 giá trị thì giá trị chiều còn lại mặc định là 1.

#### 3.4.4. Các biến lưu chỉ số cho thread, block, grid.

Trong quá trình thực thi các đoạn mã trên thiết bị, CUDA tự động đánh số cho các thread dựa trên cấu hình mà ta đưa vào. Các biến này gồm có:

* gridDim: Đây là biến kiểu dim3 và chứa các kích thước của lưới
* blockIdx: Đây là biến thuộc kiểu uint3 chứa các chỉ số khối trong lưới
* blockDim: Biến này là loại dim3 chứa kích thước của khối.
* threadIdx: Biến này thuộc loại uint3 và chứa các chỉ số luồng trong khối

## 3.5 Tối ưu.

Trong phần này sẽ trình bầy một số bước cơ bản của việc tối ưu về phần cách sử dụng bộ nhớ cũng như tổ chức chương trình. Các chiến thuật tối ưu trình phụ thuộc vào nhiều yếu tố như thuật toán, dữ liệu, phần cứng hệ thống…

#### 3.5.1 Tối ưu phần sử dụng trên GPU( Maximize Utilization)

Tối ưu phần sử dụng trên GPU được hiểu đơn giản là chúng ta thiết kế chương trình song song một cách tối đa nhất có thể, tận dụng tối đa tài nguyên trên GPU. Ở cấp độ cao ứng dụng, việc tối ưu chủ yếu nhắm vào các giao tiếp giữa host và device, đồng bộ các lời gọi hàm. Ở cấp độ thấp hơn là trên device, tối ưu thực hiện trên từng kernel. Trong khi thực hiện kernel trên device, sẽ có những điểm mà sự song song bị phá vỡ bởi một số luồng yêu cầu đồng bộ dữ liệu với các luồng khác. Tại đây sẽ có hai trường hợp: Thứ nhất các luồng nằm trong cùng 1 block, lúc này ta chỉ cần sử dụng shared memory để các luồng đều có thể truy cập đến. Trường hợp thứ hai là khác block thì ta cần sử dụng global memory làm trung gian. Như vậy khi thiết kế cấu trúc của các block cần tối ưu sao cho sự đồng bộ các luồng nằm trên cùng 1 block. Thấp hơn là cấp độ trên các Multiprocessor, ta cần chú ý đến đặc điểm phần cứng cũng như bản chất của cấu trúc chương trình. Trong phần này cần chú ý thuật ngữ “latency” chỉ số chu kỳ đồng hồ mà một wrap chuẩn bị để thực hiện chỉ thị tiếp theo.Thực hiện tối ưu trong mức độ này, chúng ta cần giữ cho các wrap luôn luôn các chỉ thị sẵn sàng để chạy với mỗi chu kỳ đồng hồ.

#### 3.5.2 Tối ưu băng thông bộ nhớ

Việc tối ưu bộ nhớ được bắt đầu từ việc trao đổi dữ liệu giữa host và device. Tiếp theo là tối việc truy cập dữ liệu trên global memory bằng cách sử dụng các bộ nhớ on-chip: shared memory, cache. Việc truy cập bộ nhớ trên GPU cần chú ý đến hiện tượng chồng chéo khi có 2 luồng cùng truy cập(đọc, ghi) lên cùng một chỗ, khi đó ta cần sử dụng đến hàm atomic của cuda, mục đích của loại hàm này là khóa bộ nhớ đang được dùng bởi 1 luồng cho đến khi nó được thực hiện xong.

#### 3.5.3 Tối ưu chỉ thị.

Trong CUDA, cần chú ý đến một khái niệm là thông lượng của chỉ thị. Để xử lý một lệnh cho một wrap, GPU cần thực hiện các bước sau:

* Đọc chỉ lệnh cho mỗi luồng của warp.
* Thực hiện lệnh.
* Ghi lại kết quả của mỗi luồng.

Thông lượng xử lý lệnh phụ thuộc vào thông lượng lệnh cần xử lý, độ trễ, băng thông bộ nhớ, chỉ số này được tối ưu khi:

* Giảm thiểu sử dụng lệnh với thông lượng thấp.
* Sử dụng tối đa băng thông của các loại bộ nhớ.

# Chương 4. CUDA VÀ MỘT SỐ BÀI TOÁN.

## 4.1 Nhân hai ma trận.

### 4.1.1 Cơ sở lý thuyết.

Nhân hai ma trận là bài toán cơ bản trong trong toán học nói chung cũng như tính toán song song nói riêng. Phần dưới sẽ trình bày thuật toán song song trên CUDA để nhân hai ma trận.

Đầu vào là hai ma trận A kích thước n\*m, B có kích thước m\*k:

Ma trận kết quả C có kích thước n\*k, trong đó phần tử ở hàng thứ i, cột j được xác định bởi:

Với i = 0…n, j = 0…k.

### 4.1.2 Thuật toán tuần tự.

|  |
| --- |
| Thuật toán tuần tự nhân 2 ma trận.  Input: Ma trận a,b. Kích thước rA,cA,rB,cB của hai ma trận  Output: Ma trận kết quả c. |
| sum 🡨 0;  for i 🡨 0 to rA  for j 🡨 0 to cB  for k 🡨 0 to rB  sum 🡨 sum + a[i\*cA + k]\*b[k\*cB + j];  end for;  c[i\*cB +j] 🡨 sum; sum 🡨 0;  end for;  end for; |

Hình 13: Thuật toán tuần tự nhân hai ma trận.

Chú ý: Trong bài toán nhân ma trận này và các bài toán trình bày ở phần sau, các ma trận sẽ được lưu dưới dạng mảng 1 chiều.

### 4.1.3 Song song hóa.

Ý tưởng: mỗi thread sẽ tính một phần tử của ma trận kết quả

Nhận xét: mỗi phần tử c[i,j] được tính bằng công thức:



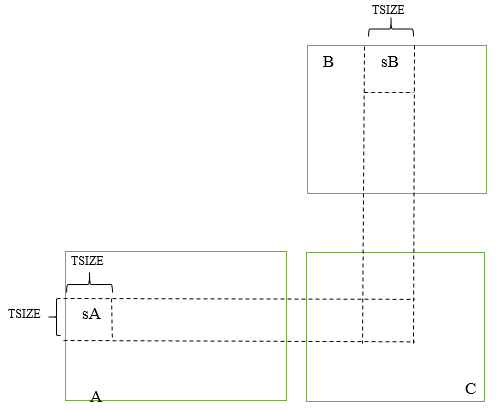
Với ma trận kết quả có kích thước n\*k, ta cần n\*k thread để tính toán. Công việc cũng khá đơn giản mỗi thread load một hàng của ma trận A, một cột của ma trận B dựa trên chỉ số định danh threadIdx của thread đó rồi đưa ra kết quả.

Tuy nhiên ta nhận thấy có 2 vấn đề chính sau:

* Thứ nhất trong tính toán song song các thread chạy đồng thời nên có thời điểm có hơn một thread cùng đọc vào dữ liệu một hàng hoặc một cột của ma trân A, B. Với những ma trân nhỏ thì không có vấn đề nhưng khi số chiều của ma trận tăng lên thì đây là một trong những yếu tố chính làm giảm hiệu suất của chương trình.
* Thứ hai dữ liệu của hai ma trận A và B đều nằm trên global memory nên tốc độ truy cập rất chậm.

Để giải quyết hai vấn đề trên ta có một giải pháp khá tốt đó là dùng shared memory. Bộ nhớ chia sẻ là một thành phần trong block có tốc độ truy cập nhanh hơn global memory khoảng 100 lần, tuy nhiên dung lượng của shared memory rất nhỏ ~48Kb nên cần xử lý mềm dẻo.

Ta sẽ dùng hai mảng sA và sB trên shared memory có kích thước TSIZE\*TSIZE để lưu trữ các giá trị load từ global memory. Mỗi block có tương ứng TSIZE\*TSIZE thread xử lý, các thread sử dụng chung hai mảng sA và sB để tính toán. Mỗi lần load dữ liệu vào shared memory với mỗi thread trong block ta sẽ tính được một giá trị tạm thời, sau khi block duyệt hết dải dữ liệu thì ta thu được một khối kết quả TSIZE\*TSIZE. Cách làm này có ưu điểm là chỉ cần đọc dữ liệu từ global memory 1 lần duy nhất đối với mỗi lần tính toán nên giảm thời gian trễ của chương trình khi chạy một cách đáng kể. Mô tả cách thức hoạt động được trình bày trong hình 14.



Hình 14: Nhân hai ma trận sử dụng shared memory.

Mô tả chi tiết của thuật toán:

|  |
| --- |
| Thuật toán song song cho bài toán nhân ma trận |
| Input: Ma trận a,b. Kích thước rA,cA,rB,cB của hai ma trận, kích thước mảnh TSIZE  Output: Ma trận kết quả c. |
| \_\_shared\_\_ sa[TSIZE][TSIZE];  \_\_shared\_\_ sb[TSIZE][TSIZE];  tx 🡨 threadIdx.x;  ty 🡨 threadIdx.y;  row 🡨 blockIdx.x\*TSIZE + threadIdx.x;  col 🡨 blockIdx.y\*TSIZE + threadIdx.y;  sum 🡨 0;  maxloop 🡨 cA/TSIZE;  if (cA%TSIZE != 0) maxloop 🡨 maxloop + 1;  end if;  if (cA<TSIZE) maxloop 🡨 1;  end if;  For i = 0 to maxloop - 1  {  If ((row < rA) && ((i\*TSIZE+ty)<cA))  sa[tx][ty] 🡨 a[row\*cA + (i\*TSIZE+ty)];  end if;  else  sa[tx][ty] 🡨 0;  end else;  if ((col < cB) && ((i\*TSIZE + tx)<rB))  sb[tx][ty] 🡨 b[(i\*TSIZE + tx)\*cB + col];  end if;  else  sb[tx][ty] 🡨 0;  end else;  \_\_syncthreads();  For k 🡨 0 to TSIZE -1  sum 🡨 sum + sa[tx][k] \* sb[k][ty];  end for;  \_\_syncthreads();  }  if(row<rA && col<cB)  c[row\*cB + col] 🡨 sum;  end if; |

Hình 15: Thuật toán song song nhân hai ma trận.

## 4.2 Thuật toán phân cụm k-means.

### 4.2.1 Cơ sở lý thuyết.

Thuật toán K-Means, cho phép sắp xếp các điểm dữ liệu vào những nhóm cho trước dựa trên sự “gần gũi” của chúng. Mỗi điểm chỉ thuộc về 1 nhóm, tâm của nhóm được xác định bằng trung bình của tất cả các điểm thuộc nhóm đó. Thuật toán K-Means được ứng dụng trong nhiều lĩnh vực khác nhau như phân nhóm khách hàng trong kinh doanh, phân nhóm động/thực vật trong sinh học, phân loại tài liệu, dự đoán, phát hiện gian lận tài chính…

Trong bài này ta sử dụng công thức Euclidean làm tiêu chuẩn để xác định khoảng cách giữa hai điểm. Công thức Euclidean:

Đầu vào của bài toán là số nhóm k và ma trận A:

### 4.2.2 Thuật toán k-means tuần tự.

Đầu tiên chúng ta đi vào thuật toán tuần tự. Thuật toán k-means truyền thống gồm các bước chính:

* B1: Khởi tạo K tâm ngẫu nhiên.
* B2: Tính toán khoảng cách giữa phần tử và tâm. Khoảng cách với tâm nào nhỏ nhất thì phân vào tâm đó.
* B3: Tính toán lại tâm.
* B4: Nếu không có phần tử nào thay đổi tâm thì đến B5, nếu có quay lại B2.
* B5: Kết thúc, đưa ra kết quả.

Khi khởi tạo tâm, ta thường chọn ngẫu nhiên k điểm trong tập dữ liệu. Nhưng chọn ngẫu nhiên các tâm sẽ làm kết quả không kiểm soát được, vì thế để thuận lợi cho việc so sánh hiệu năng ta chọn cố định k điểm đầu tiên làm tâm. Thuật toán k-means:

|  |
| --- |
| Thuật toán K-Means tuần tự. |
| **Do**  Change  **For** i 🡨 0 **to** n - 1 **do**  min\_dis 🡨 ();  Index 🡨 0;  **For** j 🡨 1 **to** k-1 **do**  t\_dis 🡨 ()  **if**(t\_dis < min\_dis)  index 🡨 j;  min\_dis 🡨 t\_dis;  **end** **if**;  if    Change  **End** **for**;  **End** **for**;  **For** i 🡨 0 **to** k – 1 **do**  **End for**  **While** change still true; |

Hình 16: Thuật toán k-means tuần tự.

Trong thuật toán ta có C là ma trận tâm của các nhóm kích thước k\*n, A là tập dữ liệu còn m chứa vị trí nhóm của các phần tử.

### 4.2.3 Song song hóa thuật toán.

Nhận xét: trong bước 2 của thuật toán k-means với mỗi điểm ta thực hiện tính toán khoảng cách với k tâm rồi từ đó tìm ra tâm gần nó nhất, như vậy tất cả các điểm đều thực hiện một công việc giống nhau chỉ khác ở bộ dữ liệu đầu vào.

Ý tưởng song song thuật toán k-means trong bước này là sử dụng n\*k thread để tính toán ra một ma trận khoảng cách D kích thước n\*k và với D[i,j] là khoảng cách từ điểm i đến tâm j. Sau khi có ma trân khoảng cách D duyệt song song các hàng để tìm ra tâm gần điểm tương ứng với hàng đó nhất. Sau cùng tính toán lại tâm theo các điểm mới thêm vào nhóm.

Thuật toán song song tìm ma trận khoảng cách:

|  |
| --- |
| Tìm ma trận khoảng cách D |
| Input : dataset A, centroid C.  Output : distance D.  //blockDim.x : kích thước của block phương x.  // blockIdx.x : định danh của block theo phương x.  // threadIdx.x : định danh của thread theo phương x. |
| objIdx 🡨 blockDim.x \* blockIdx.x + threadIdx.x;  clusIdx 🡨 blockDim.y \* blockIdx.y + threadIdx.y;  D[objIdx][clusIdx] 🡨 (); |

Hình 17: Thuật toán song song tìm ma trận khoảng cách.

Thuật toán song song tìm tâm gần nhất với M là mảng chứa chỉ số nhóm của các phần tử. Ví dụ M[i] = m nghĩa là phần tử vị trí thứ i thuộc vào nhóm m.

|  |
| --- |
| Thuật toán tìm khoảng cách gần nhất |
| Input: distance D.  Output: member M. |
| int objectId 🡨 blockDim.x \* blockIdx.x + threadIdx.x;  min\_dist 🡨 D[objectId][0];  index 🡨 0;  **for** i 🡨 1 **to** k – 1  dist = D[objectId][i];  **if** (dist < min\_dist)  min\_dist 🡨 dist;  index 🡨 i;  **end if**  **end for**  M[objectId] 🡨 index; |

Hình 18: Thuật toán song song tìm khoảng cách gần nhất.

Xét thấy : ở bước thứ 2 ta có hai ma trận đầu vào là ma trận dữ liệu A và ma trân tâm C, ta nhận thấy bước này có phần tương tự với bài toán nhân ma trận ở phần 4.1 ở điểm dữ liệu các tâm sẽ bị truy cập nhiều lần và nhiều luồng cùng đọc giá trị của một tâm. Áp dụng ý tưởng chia nhỏ dữ liệu tính toán thành từng phần giống bài toán nhân ma trận ta viết lại thuật toán tìm ma trận khoảng cách như sau:

* Load từng phần của dữ liệu từ global memory vài shared memory
* Tính toán trên mảnh dữ liệu vừa load vào
* Tiếp tục load phần tiếp theo và lặp lại đến khi hết hàng dữ liệu đó.

|  |
| --- |
| Thuật toán tính khoảng cách sử dụng shared memory. |
| Input : dataset A, centroid C.  Output : distance D.  //\_\_syncthreads(); đồng bộ các thread  //\_\_shared\_\_ khai báo biến trong shared memory.  // BLOCK\_SIZE: kích thước mảng trung gian. |
| tx = threadIdx.x;  ty = threadIdx.y;  bx = blockIdx.x;  by = blockIdx.y;  ROW 🡨 BLOCK\_SIZE \* bx + tx;  COL 🡨 BLOCK\_SIZE \* by + ty;  //Tạo hai ”mảnh” trên shared memory  \_\_shared\_\_ sA[BLOCK\_SIZE][BLOCK\_SIZE];  \_\_shared\_\_ sB[BLOCK\_SIZE][BLOCK\_SIZE];  sum 🡨 0; //biến lưu kết quả tam thời  tmp 🡨 0;  maxLoop = n/BLOCK\_SIZE + 1;  **for** m 🡨 0 **to** maxLoop – 1  //load dữ liệu  sA[tx][ty] 🡨 A[ROW][BLOCK\_SIZE\*m + ty];  sB[tx][ty] 🡨 C[COL][ BLOCK\_SIZE\*m + tx];  \_\_syncthreads();  **For** k 🡨 0 to BLOCK\_SIZE - 1 ;  tmp 🡨 sA[tx][k] - sB[k][ty];  sum 🡨 sum + tmp\*tmp;  **end** **for;**  \_\_syncthreads();  **End for;**  D[index] = sum; |

Hình 19: Thuật toán tìm ma trận khoảng cách sử dụng shared memory.

## 4.3 Một số thuật toán trên ảnh.

### 4.3.1 Giới thiệu về ảnh số.

Ảnh tự nhiên là ảnh liên tục về không gian và độ sáng. Để xử lý các ảnh này bằng máy tính thì các ảnh này cần được số hóa. Số hóa ảnh là sự biến đổi gần đúng một ảnh liên tục thành các điểm ảnh thể hiện màu sắc, độ sáng tương ứng với ảnh thật. Mỗi điểm như vậy được gọi là 1 pixel, mỗi pixel trong một bức ảnh hai chiều sẽ có hai thành phần tọa độ tương ứng là (x,y). Kích thước và khoảng cách của các pixel khá bé làm cho mắt con người cảm nhận được sự liên tục về không gian, độ sáng, màu sắc của ảnh số gần như ảnh thật.

### 4.3.2 Mức xám của ảnh.

Một điểm ảnh có hai đặc trưng cơ bản là vị trí (x,y) và độ xám. Mức xám của điểm ảnh là cường độ ánh sáng của nó được gán bằng giá trị số tại điểm đó. Các thang giá trị mức xám thông thường : 16, 32, 64, 128, 256 (mức 256 được dùng phỏ biến nhất). Hiện nay ảnh số có 3 loại chính sau:

* **Ảnh đen trắng**: là ảnh chỉ có hai màu đen và trắng, người ta phân mức đen trắng của ảnh ra làm L mức. Nếu điểm ảnh được biểu diễn bằng 8 bit thì L = = 256 mức. Nếu L = 2 thì ảnh này được gọi là ảnh nhị phân. Mức 1 ứng với màu sáng nhất, mức 0 ứng với màu tối nhất.
* **Ảnh màu**: theo lý thuyết của Thomas thì ảnh màu được tổ hợp từ 3 màu cơ bản : đỏ, lục, lam. Với ảnh màu thì mỗi điểm ảnh cần 8 bit để biểu diễn cho 1 màu, vậy chúng ta cần 24 bit cho mỗi điểm ảnh. Mỗi màu cũng được phân ra L = 256 cấp khác nhau biểu diễn độ sáng của màu đó tại điểm ảnh đang xét.

### 4.3.3 Chuyển từ ảnh màu sang ảnh xám.

Trong xử lý ảnh thì đây là một thuật toán cơ bản, thuật toán này chuyển ảnh màu kiểu RGB sang ảnh xám. Với ảnh xám mỗi điểm ảnh không phải tổng hợp của 3 màu đỏ, xanh lá và xanh dương mà chỉ có một giá trị duy nhất. Giá trị của các điểm ảnh thường nằm trong khoảng [0,255].

Ta có hai công thức chuyển đổi hay dùng:

* Y = 0.299R +0.587G + 0.114B (1)
* Y = 0.2126R + 0.7152G + 0.0772B (2)

Công thức (1) thường được dùng cho các hệ TV PAL, SECAM, NTSC, còn công thức (2) dùng cho các màn hình độ phân giải cao HDTV.

Giải thuật song song: Với ảnh có kích thước w\*h, ta cho mỗi thread tính toán một giá trị Y.

|  |
| --- |
| Thuật toán song song chuyển ảnh màu sang ảnh xám |
| Input:Mảng giá trị màu R,G,B  Output:Mảng giá trị xám Y |
| Tx 🡨 blockDim.x \* blockIdx.x + threadIdx.x;  ty 🡨 blockDim.y \* blockIdx.y + threadIdx.y;  Y[tx\*h+ty]🡨 0.299R[tx\*h+ty] +0.587G[tx\*h+ty] + 0.114B[tx\*h+ty] |

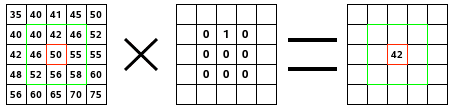
Hình 20: Thuật toán song song chuyển ảnh màu sau ảnh xám.

### 4.3.4 **Image Filtering.**

#### 4.3.2.1 Tích chập hai ma trận

Trong toán học tích chập là 1 phép toán thực hiện với hai hàm số f và g, kết quả của phép toán này cho ra một hàm số mới. Trong phần này sẽ không đi sâu vào phần toán học của tích chập mà chỉ đề cập đến tích chập cho hai ma trận.

* Trong tích chập ma trận, chúng ma có hai ma trận đầu vào:
* Thứ nhất là ma trận gốc chứa dữ liệu của ảnh.
* Thứ hai là ma trận trọng số.
* Cách tính chập của hai ma trận



Hình 21: Cách tính chập hai ma trận.

Với ma trận 3\*3 và tâm có giá trị là 50 thì tâm của ma trận kết quả có giá trị tương ứng là (40\*0 + 42\*0 + 46\*0 + 46 \* 0 + 50 \* 0 + …+ 58 \* 0) = 42. Với những ô ở biên, các giá trị nằm ngoài vùng dữ liệu ta cho mặc định là 0.

#### 4.3.2.2 Áp dụng

|  |
| --- |
| Thuật toán xử lý ảnh sử dụng chập ma trận |
| Input: ma trận dữ liệu ảnh data kích thước w\*h, ma trận trọng số filter kích thước m\*m  Output:Ma trận kết quả result |
| for i 🡨 0 to w do  for j 🡨 0 to h do  red 🡨 0; green 🡨 0; blue 🡨 0;  for ix 🡨 0 to m - 1  for iy 🡨 0 to m - 1  x 🡨 (i - filterSize/2 + ix + w) % w;  y 🡨 (j - filterSize/2 + iy + h) % h;  red += data.r[x\*h+y]\* filter [ix\*m + iy];  green += data.g[x\*h+y]\* filter [ix\*m + iy];  blue += data.b[x\*h+y]\* filter [ix\*m + iy];  End for;  Result.r[i\*h + j] 🡨 red;  Result.g[i\*h + j] 🡨 green;  Result.b[i\*h + j] 🡨 blue;  End for;  End for;  End for; |

Hình 22: Thuật toán filter trên ảnh.

#### 4.3.2.3 Song song hóa.

Ý tưởng cũng tương tự như phần chuyển đổi ảnh màu sang ảnh xám, với bức ảnh w\*h thì chúng ta sẽ có w\*h thread xử lý song song.Trong phép chập ma trận với ma trận trọng số có kích thước m\*m mỗi lần thực hiện ta cần đọc một ma trận kích thước m\*m như vậy 2 ô gần nhau sẽ sử dụng chung (m\*m – m). Kết quả là thời gian tính toán không thực sự được cải thiện nhiểu nếu chúng ta sử dụng global memory, thay vào đó chúng ta sẽ sử dụng shared memory để lưu trữ dữ liệu tính toán.

## 4.4 Thực nghiệm.

Trong phần này ta sẽ chạy các chương trình trên cả CPU lẫn GPU CUDA nhằm so sánh hiệu năng giữa chúng. Cấu hình thử nghiệm:

* CPU: Core i3 2230M 2.2Ghz, RAM 6 GB 1333Mhz.
* GPU: nVidia Geforce GT 540M 672MHz, Ram 1GB 900Mhz.

### 4.4.1 Nhân hai ma trận.

Chương trình thực hiện nhân hai ma trận Ta thực hiện và so sánh kết quả khi chạy bằng CPU và khi dùng GPU. Ma trận kết quả:

với

Bảng kết quả thử nghiệm:

Bảng 1: Thời gian chạy chương trình nhân ma trận

|  |  |  |  |
| --- | --- | --- | --- |
| Size | CPU | GPU | Speed up |
| 512x512 | 915 ms | 71 ms | 12.8 |
| 1024x1024 | 28128 ms | 579 ms | 48.5 |
| 2048x2048 | 192567 ms | 4632 ms | 45.0 |

Hình 23: Đồ thị so sánh thời gian chạy chương trình nhân ma trận.

### 4.4.2 Thuật toán k-means.

Với 2 đề xuất thực hiện song song hóa thuật toán K-Means với CUDA bên trên, ta tiến hành thử nghiệm thuật toán song song K-Means trên nền tảng phần cứng với thông số như sau: CPU Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz (4 CPUs), ~2.2GHz và GPU Nvidia Geforce GT540M và thu được kết quả như liệt kê tại Bảng 6 ở đó CUDA0 là ký hiệu khi thực hiện thuật toán song song K-Means cải tiến và CUDA1 là ký hiệu khi thực hiện thuật toán song song K-Means chưa cải tiến.

Bảng kết quả chạy thử nghiệm chương trình trên nhiều tập dữ liệu khác nhau:

Bảng 2: Kết quả chạy thuật toán K-Means

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Object | Dim | Cluster | CPU | CUDA0 | CUDA1 | Speed up |
| 1024 | 1024 | 16 | 35.5 | 7.8 | 23.39 | 4.5 |
| 2048 | 1024 | 16 | 75 | 16 | 43.122 | 4.6 |
| 4096 | 1024 | 16 | 137.5 | 29.6 | 97.76 | 4.6 |
| 1024 | 1024 | 32 | 116.45 | 17.125 | 58.4 | 6.8 |
| 2048 | 1024 | 32 | 236.433 | 34.69 | 119.23 | 6.8 |
| 1024 | 1024 | 64 | 187.8 | 24.12 | 115.2 | 7.8 |
| 1024 | 1024 | 128 | 394 | 47 | 186.37 | 8.3 |
| 512 | 512 | 128 | 39.56 | 4.58 | 14.7 | 8.6 |
| 1024 | 512 | 128 | 73.3 | 8.6 | 28.1 | 8.55 |
| 2048 | 512 | 128 | 187.6 | 21.8 | 65.2 | 8.5 |
| 1024 | 512 | 512 | 218.1 | 23.6 | 69.77 | 9.2 |
| 2048 | 512 | 512 | 638.5 | 68.3 | 177.63 | 9.35 |
| 1024 | 1024 | 512 | 1019.9 | 108.5 | 478.5 | 9.4 |

Hình 24: Đồ thị so sánh thời gian chạy trên tập dữ liệu 1024\*1024

Qua bảng kết quả thử nghiệm và biểu đồ (với tập dữ liệu là 1024\*1024) thể hiện thời gian thực hiện phụ thuộc số lượng Clusters thì ta thấy thuật toán song song K-Means cải tiến (CUDA0) có thời gian thực hiện tăng lên đáng kể so với thời gian thực hiện thuật toán song song K-Means chưa cải tiến (CUDA1) và so với thời gian thực hiện thuật toán K-Means trên CPU.

### 4.4.3 Thuật toán trên ảnh.

Trong phần này ta thực hiện với hai bức ảnh có kích thước là 1920x1200 và 3008x2000, kích thước ma trận thu được kết quả như sau:

Bảng kết quả thời gian chạy thuật toán chuyển ảnh màu sang ảnh xám:

Bảng 3: Thời gian thực nghiệm thuật toán.

|  |  |  |  |
| --- | --- | --- | --- |
| Image size | CPU | GPU | Speed up |
| 1920x1200 | 202 ms | 14 ms | 14.4 |
| 3008x2000 | 492 ms | 37 ms | 13.29 |

Bảng kết quả thời gian chạy thuật toán filter:

Bảng 4: Thời gian chạy thuật toán filter.

|  |  |  |  |
| --- | --- | --- | --- |
| Image size | CPU | GPU | Speed up |
| 1920x1200 | 1487 ms | 237 ms | 6.2 |
| 3008x2000 | 3870 ms | 741 ms | 5.22 |

Qua hai thuật toán trên ảnh ta có thể thấy được sự khác biệt khá lớn giữa tính toán trên CPU và GPU. Hai thuật toán chạy thử nghiệm chưa được tối ưu hoàn toàn, nếu có thời gian phát triển thì kết quả chắc chắn sẽ còn ấn tượng hơn rất nhiều.

# KẾT LUẬN.

Trong đồ án này đã trình bày tổng quát về tính toán song song và cấu trúc cơ bản của mô hình tính toán song song trên GPU. Trên cơ sở đó trong chương thứ 3 của đồ án đã đi sâu vào nghiên cứu kiến trúc lập trình song song trên GPU CUDA với các khía cạnh như mô hình lập trình, kiến trúc phần cứng, các hướng dẫn tăng hiệu năng chương trình. Từ những kiến thức trong quá trình nghiên cứu thì chương 4 của đồ án là phần thử nghiệm năng lực tính toán của GPU so với CPU trên một số bài toán.

Với những hạn chế về trình độ cũng như thời gian thì đồ án lần này mới chỉ dừng lại ở mức nghiên cứu công nghệ tính toán song song cũng như thực hành trên một số bài toán mang tính chất thử nghiệm. Với cơ sở hiện tại thì hướng phát triển trong tương lai của đồ án là nghiên cứu sâu về tính toán song song trên GPU CUDA và đưa ra một ứng dụng hoàn thiện.

# TÀI LIỆU THAM KHẢO.

**Tài liệu tiếng anh:**

1. Blaise Barney,"Introduction to Parallel Computing".

2. C.F. Olson, "Parallel algorithms for hierarchical clustering"

3. CUDA C PROGRAMMING GUIDE, Nvidia.

4. Mark Harris, "Optimizing Parallel Reduction in CUDA".

5. MD Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary, "Scalable Parallel OPTICS Data Clustering Using Graph Algorithmic.

6. Mohamed Zahran, "Graphics Processing Units (GPUs): Architecture and Programming (aka Z)", mzahran@cs.nyu.edu, <http://www.mzahran.com>.

7. Wei-keng Liao, "Parallel K-Means Data Cluster".

**Các tài liệu khác:**

8. Convolution Matrix http://docs.gimp.org/2.8/en/pluginconvmatrix.html.

9. wiki, "Image processing".

10. wiki, "K-means clustering ".

11. wiki, "Parallel computing ".