KIẾN TRÚC TESLA VÀ MÔ HÌNH LẬP TRÌNH SONG SONG CUDA

Trần Thanh Hùng
Khoa CNKT Điện tử
Trường Đại học Công nghiệp Hà Nội

Bài viết nhằm đi sâu vào việc khám phá kiến trúc bộ xử lý của card đồ họa NVidia và mô hình lập trình song song CUDA nhằm áp dụng trong các bài toán tính toán hiệu năng cao.

1. KIẾN TRÚC PHẦN CỨNG TESLA CỦA NVIDIA (TESLA ARCHITECTURE)

Hiệu năng tính toán của những bộ vi xử lý đồ hoạ (GPU) ngày càng tăng thậm chí vượt xa so với các bộ xử lý CPU thông thường. Để tận dụng được khả năng tính toán này cho những bài toán phi đồ họa, các nhà sản xuất cũng như nghiên cứu đưa ra một khái niệm mới GPGPU (General-Purpose Computation on GPUs – Tính toán thông dụng trên các bộ xử lý đồ họa) đó là việc sử dụng GPU để làm những công việc trước kia là của CPU để tăng cường hiệu suất làm việc. Làm thế nào để làm được việc đó? Làm thế nào để chương trình lập trình cho GPU có thể tương thích với nhiều bộ xử lý đồ họa khác nhau? để giải quyết vấn đề này, hãng NVIDIA phát triển một ngôn ngữ mới dựa trên C cho phép lập trình viên dễ dàng xây dựng ứng dụng tính toán thông dụng cho các GPU (mà ban đầu là các GPU dòng G80).

Để tiến thêm một bước xa hơn nữa, NVIDIA tung ra dòng sản phẩm có tên gọi Tesla (AMD/ATI cũng có dòng sản phẩm với khái niệm tương tự có tên gọi Stream). Tesla ban đầu dựa trên sức mạnh của GPU GeForce 8800 nhưng không tạo ra tín hiệu Video: chúng được dùng như các bộ xử lý chuyên dụng cho mục đích tính toán. Các chương trình viết cho Tesla phải được dịch bằng CUDA, do đó những người dùng thông thường không được hưởng những lợi ích từ công nghệ này (khi cài đặt Tesla vào máy tính, hiệu suất hệ thống sẽ không tự động tăng lên). Tesla chỉ phù hợp với những ứng dụng nặng về tính toán và những tính toán này có thể song song hóa được. Ví dụ những chương trình trong lĩnh vực: Vật lý, Tài chính, Y tế, Sinh học và Hoá học...

Hình 1. Biểu tượng Tesla

Lập trình viên cũng không nhất thiết phải có Tesla mới có thể sử dụng được CUDA. Họ có thể sử dụng bất kỳ Card màn hình nào thuộc dòng GeForce 8800 trở lên (và GTX200 hiện nay).

2. KIẾN TRÚC TÍNH TOÁN TRÊN GPU TESLA VÀ MÔ HÌNH LẬP TRÌNH SONG SONG CUDA

Trong kỹ thuật xử lý đồ họa 3D, quá trình chuyển đổi một hình ảnh dạng ba chiều sang dạng điểm ảnh đòi hỏi các thao tác xử lý đồ họa điểm và đồ họa véc-tơ. Xử lý đồ họa véc-tơ là các thao tác trên véc-tơ căn bản như các điểm, đường thẳng, và các hình tam giác. Các thao tác này sử dụng các mô hình hình học và việc tô trát trên các mô hình đó để hiển thị đồ họa hay nói cách khác đó là các thao tác truyền tải các hệ thống tọa độ, thiết lập các tham số kết cấu và ánh sáng để hiển thị hình ảnh. Xử lý đồ họa điểm ảnh là thao tác chuyển đổi hình ảnh từ mô hình hình học sang các điểm ảnh và thể hiện chúng trên các lưới điểm ảnh đầu ra, thao tác thường thấy là phủ đầy các điểm ảnh bên trong hình ảnh nguyên thủy.

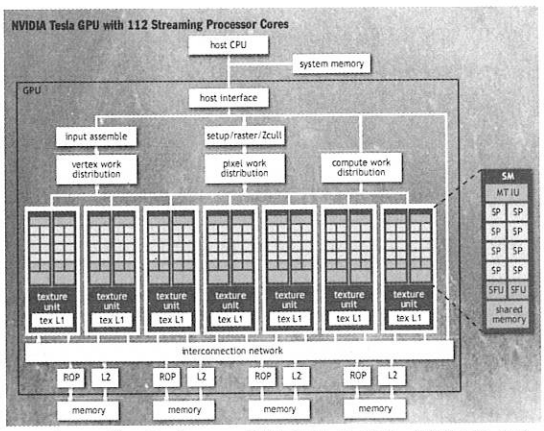
Trong quá trình phát triển, xử lý điểm ảnh và véc-tơ đã phát triển ở hai dạng khác nhau. Xử lý véc-tơ có độ trễ thấp, thao tác toán học có độ chính xác cao và xử lý phức hợp nhiều hơn, chính điều này đã trở thành khả năng có thể lập trình được. Xử lý điểm ảnh có độ trễ cao (vì phải qua quá trình chuyển đổi từ mô hình hình học sang các điểm ảnh, sau đó mới được hiển thị ở đầu ra), bộ lọc kết cấu có độ chính xác thấp hơn...thông thường GPU phải xử lý nhiều điểm ảnh hơn các véc-tơ (với tỷ lệ 3:1). Rồi khả năng

tăng cường tính phổ biến của GPU như tăng cường các thiết kế phức hợp, vùng miền, giá thành của hai cách xử lý riêng biệt. Thêm đó là khả năng tính toán phức hợp (lập trình được), và GPU loại này đang được lựa chọn với tỷ lệ nhiều hơn so với lựa chọn các GPU có các bộ xử lý không lập trình được, chính những điều này là động cơ thúc đẩy của kiến trúc xử lý Tesla.

2.1. Kiến trúc tính toán trên GPU Tesla

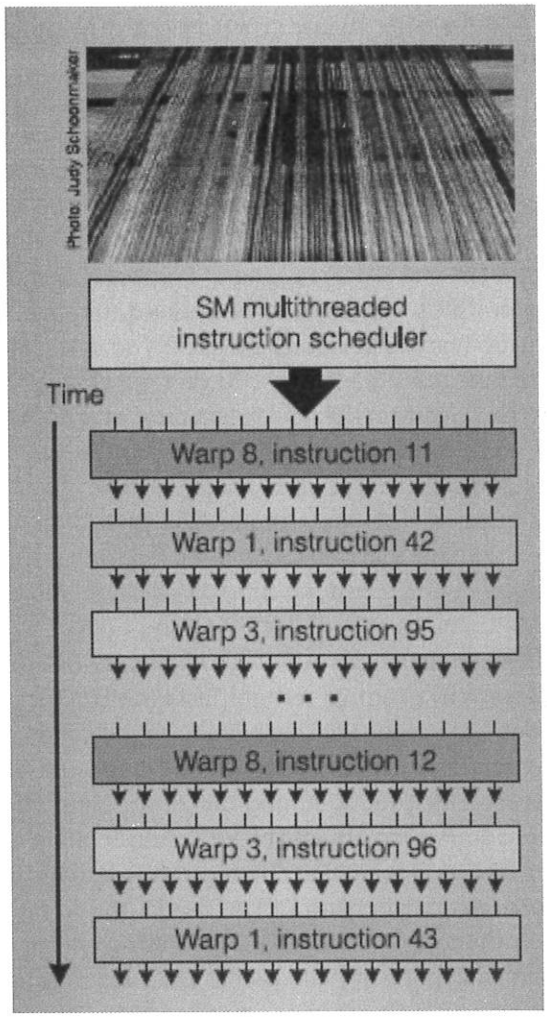
Được NVIDIA giới thiệu vào tháng 11 năm 2006, kiến trúc tính toán và đồ họa hợp nhất Tesla đã phát triển, gia tăng đáng kể về số lượng so với các GPU đồ họa không lập trình được. Khả năng xử lý đa luồng với số lượng luồng thực thi cực lớn khiến kiến trúc này trở nên một nền tảng hiệu quả cho cả những ứng dụng tính toán đồ họa và ứng dụng tính toán song song thông thường trên GPU.

Kiến trúc Tesla được xây dựng xung quanh một mảng có thể mở rộng của các SM (streaming multiprocessors – đa xử lý dòng lệnh). Mỗi SM chứa 8 lõi xử lý vô hướng SP. Những GPU theo kiến trúc này có thể thực hiện từ 768 cho tới 12288 luồng thực thi đồng thời. Với sự hỗ trợ của môi trường phát triển ứng dụng CUDA, các GPU này cho phép lập trình viên mở rộng quy mô thực thi song song một cách trong suốt (số luồng không phụ thuộc vào năng lực thực sự của GPU).



Hình 2. Kiến trúc của 1 card NVIDIA Tesla GPU với 112 lõi xử lý

Hình 2 trình bày một GPU với 14 SMs - tương đương với 112 lõi SP (streaming processor), kết nối liên thông với 4 môđun bộ nhớ DRAM bên ngoài. Một chương trình CUDA chia ra hai phần riêng biệt, phần thực thi trên CPU, và phần thực thi trên GPU. Khi một chương trình CUDA trên CPU gọi một kernel (đơn vị chương trình được viết để thực thi trên GPU) kernel này sẽ được thực hiện N lần song song với nhau bởi N luồng CUDA khác nhau. N luồng này được tổ chức thành một lưới bao gồm nhiều khối (block), mỗi khối gồm nhiều luồng. Khi kernel được gọi, đơn vị phân phát công việc tính toán, gọi tắt là CWD (Compute Work Distribution) sẽ liệt kê các khối của lưới và bắt đầu phân phối các khối này tới các SM sẵn sàng thực thi (sẵn sàng chạy). Các luồng trong một khối được thực hiện đồng thời trên một SM. Khi các khối kết thúc, đơn vị CWD sẽ thực thi các khối mới trên các bộ xử lý rảnh rỗi chờ tới khi toàn bộ khối trong lưới được thực thi thành công.



Một SM có chứa 8 lõi xử lý vô hướng SP – scalar processor (SP), 2 đơn vị chức năng đặc biệt – special function units (SFUs), dành cho tính toán siêu

việt, 1 đơn vị điều hướng đa luồng – multithreaded instruction unit (MTIU), và 1 bộ nhớ chia sẻ trên nó (on-chip shared memory). SM đảm nhiệm việc tạo, quản lý, và thực hiện tới 768 luồng đồng thời trên phần cứng mà không tốn kém chi phí lập lịch. Nó có thể thực thi 8 block CUDA nhiều lần trong cùng một thời điểm. SM còn cung cấp rào chắn CUDA __syncthreads(), rào chắn này có tác dụng đồng bộ hóa các luồng bên trong SM với một lệnh đơn. Chính xác hơn, khi một điểm đồng bộ hoá đặc biệt trong kernel gọi __syncthreads(), __syncthreads() hành động như một rào chắn, nó yêu cầu tất cả các luồng trong khối phải đợi trước khi nó được phép xử lý.

Để quản lý hàng trăm luồng chạy trong nhiều ứng dụng khác nhau, Tesla SM sử dụng một kiến trúc mới gọi là "đơn dòng lệnh đa luồng" – SIMT (Single-Instruction, Multiple-thread). SM ánh xạ mỗi luồng tới một lõi vô hướng SP, và các luồng vô hướng sẽ chạy độc lập với nhau với địa chỉ tập lệnh và trạng thái thanh ghi riêng của nó. Đơn vị SM SIMT tạo, quản lý, sắp xếp và thi hành các luồng trong một nhóm gồm 32 luồng song song được gọi là warps. Các luồng riêng lẻ săp xếp theo trật tự thành một tập hợp warp khởi động cùng nhau tại cùng một địa chỉ chương trình nhưng tự do rẽ nhánh và thực thi một cách độc lập. Mỗi SM quản lý một nhóm gồm 24 warp với 32 luồng trên mỗi warp, và tổng cộng có 768 luồng.

Kiến trúc SIMT gần giống như với cấu trúc véc-tơ SIMD mà trong đó một lệnh điều khiển nhiều quá trình xử lý các phần tử. Một điều then chốt khác là cấu trúc véc-tơ SIMD chỉ ra sự song song hóa ở mức dữ liệu trong chương trình, trong khi các lệnh SIMT định rõ việc thực thi và trạng thái phân nhánh của một luồng đơn. Trái ngược với các véc-tơ SIMD, SIMT cho phép lập trình viên viết đoạn mã lệnh song song phân cấp các luồng một cách độc lập, các luồng vô hướng, cũng như là đoạn mã song song dữ liệu dành cho phối hợp các luồng. Lập trình viên có thể về căn bản bỏ qua động thái SIMT, tuy nhiên, chắc chắn hiệu năng có thể được nhận thấy bởi việc quan sát đoạn đoạn mã ít khi yêu cầu các luồng trong warp phân rẽ. Trong thực tế, nó tương tự như vai trò của các dòng đệm trong các đoạn mã truyền thống: kích thước dòng đệm có thể bỏ qua một cách an toàn khi thiết kế với tính đúng đắn nhưng phải được xem xét trong cấu trúc đoạn mã khi thiết kế cho hiệu năng tối đa. Các kiến trúc véc-tơ, theo một hướng khác, yêu cầu phần mềm kết hợp một khối để load trong các véc-tơ và điều khiển sự phân rẽ một cách thủ công.

Một biến luồng thường lưu trú trong các thanh ghi đang sống. Bộ nhớ chia sẻ 16Kb SM có độ trễ truy cập thấp và dải thông cao tương tự như bộ nhớ đệm L1, nó giữ các biến CUDA __shared__ trên khối để dành cho việc kích hoạt các khối luồng. SM cung cấp việc nạp/lưu trữ các lệnh để truy cập biến CUDA __device__ trên GPU có bộ nhớ DRAM gắn ngoài. Nó kết hợp thành một khối truy cập riêng rẽ của các luồng song song trong cùng một warp và ở trong vài truy cập khối bộ nhớ khi các địa chỉ rơi vào trong cùng một khối và gặp các tiêu chuẩn liên kết. Bởi vì độ trễ của bộ nhớ chung có thể tới hàng trăm xung nhịp xử lý, để tối ưu, các chương trình CUDA thường copy dữ liệu tới bộ nhớ chia sẻ khi dữ liệu này được truy cập nhiều lần bởi một khối luồng. Tesla nạp/lưu trữ các lệnh bộ nhớ sử dụng địa chỉ là các byte số nguyên để thuận tiện cho trình biên dịch chuyển đổi tối ưu các đoạn mã.

Các ứng dụng CUDA thực hiện tốt trên các GPU kiến trúc Tesla bởi vì mô hình song song CUDA, sự đồng bộ, các bộ nhớ chia sẻ, và phân cấp của các nhóm luồng ánh xạ hiệu quả tới các chức năng của kiến trúc GPU.

2.2. Mô hình lập trình song song CUDA

CUDA mở rộng từ ngôn ngữ lập trình C, cho phép các nhà lập trình định nghĩa các hàm trên C, được gọi là các kernel, chúng được thực thi song song tại cùng thời điểm bởi N tham chiếu tới các luồng CUDA, giống như các hàm C thông thường. Một kernel được định nghĩa sử dụng một tham số mô tả _global_ và số các luồng CUDA cho mỗi lời gọi chúng được chỉ rõ bằng việc sử dụng một cú pháp mới: <<<...>>>

```
// Định nghĩa một Kernel
__global__ void vecAdd(float* A, float* B, float* C){...}
int main(){
// Lời gọi Kernel
vecAdd<<<1, N>>>(A, B, C);}
```

Mỗi luồng thực thi được gán với một chỉ số luồng duy nhất thông qua biến threadIdx. Ví dụ, đoạn mã sau cộng 2 vector A và B có kích thước N và lưu kết quả vào trong vector C:

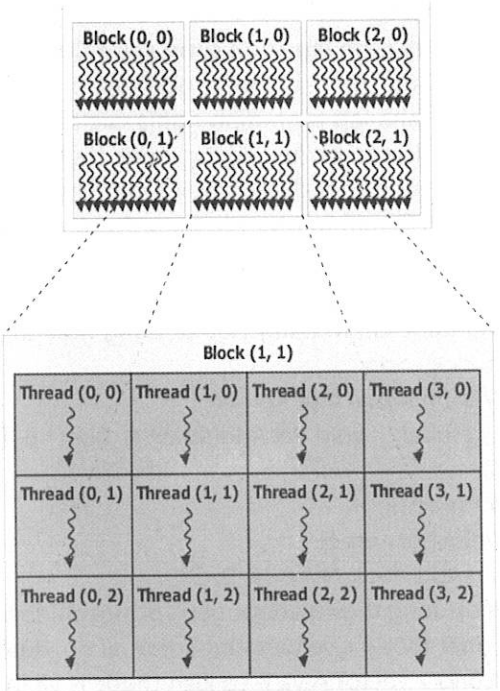
```
__global__ void vecAdd(float*A,float*B,float* C){
int i = threadIdx.x;
C[i] = A[i] + B[i];}
int main(){
// Lời gọi Kernel
vecAdd<<<1, N>>>(A, B, C);}
```

Mỗi luồng thực thi hàm vecAdd() sẽ tính một phép cộng hai phần tử tại vị trí I tương ứng với chỉ số luồng threadIdx.

- Sự phân cấp luồng

Để thuận lợi, CUDA thiết kế threadIdx như là một vector 3 thành phần (.x,.y,.z), do đó người lập trình có thể quyết định xây dựng các luồng trong một khối như là các mảng 1 chiều, hai chiều hay ba chiều. Trong ví dụ sau, đoạn mã thực hiện phép cộng 2 ma trận A và B có kích thước NxN và kết quả được lưu vào ma trận C:

```
__global__ void matAdd(float A[N][N], float B[N]
[N], float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j]; }
int main() {
    // Lời gọi Kernel
    dim3 dimBlock(N, N);
    matAdd<<<1, dimBlock>>>(A, B, C); }
```



Hình 3. Lưới các Luồng Blocks

Chỉ số của một luồng và threadId của nó có mối quan hệ tương ứng với nhau: với block một chiều, chúng giống nhau, với block 2 chiều kích thước (Dx,Dy), threadID của một luồng có chỉ số (x,y) là (x+yDx), với một block 3 chiều kích thước (Dx,Dy,Dz), threadID của một luồng có chỉ số (x,y,z) là (x+yDx+zDxDy). Các luồng thuộc về một khối có thể cùng hợp tác tính toán bằng xử lý trên vùng dữ liệu chia sẻ tương ứng với chỉ số luồng, và khi cần truy đồng bộ hóa dữ liệu chúng có thể sử dụng hàm __syncthreads() để đảm bảo các tác vụ trước rào chắn phải được hoàn tất trước khi tiếp tục xử lý.

Số luồng mỗi khối bị hạn chế bởi giới hạn nguồn tài nguyên bộ nhớ của lõi bộ vi xử lý. Với kiến trúc Tesla của NDIVIA, một khối có thể chứa tới 512 luồng. Tuy nhiên, một kernel có thể được thực thi nhiều hơn số lượng luồng trong 1 khối bằng cách tổ chức các luồng thành nhiều khối. Các khối có thể được tổ chức vào trong lưới một chiều hoặc 2 chiều của một lưới, được minh hoạ như hình 3.

Chiều của lưới được chỉ rõ bởi tham số đầu tiên của ký pháp <<<...>>>. Vị trí mỗi khối trong một lưới được quy định bởi chỉ số một chiều hoặc hai chiều blockIdx. Chiều của block có thể truy cập thông qua biến blockDim. Đoạn mã ví dụ trước đây trở thành:

```
__global__ void matAdd(float A[N][N], float B[N]
[N], float C[N][N]) {
    int i=blockIdx.x * blockDim.x + threadIdx.x;
    int j=blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
      C[i][j] = A[i][j] + B[i][j]; }
int main() {
    // Lời gọi Kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N+dimBlock.x-1) / dimBlock.x,
(N+dimBlock.y-1)/dimBlock.y);
    matAdd<<<dimGrid, dimBlock>>>(A,B,C); }
```

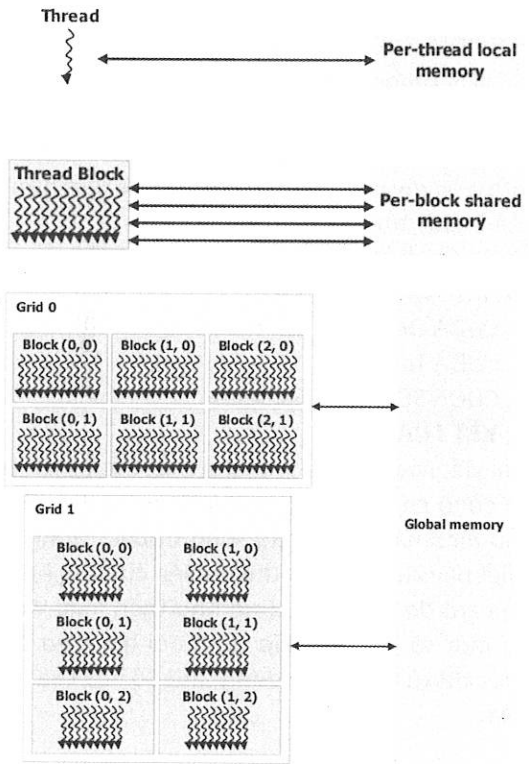
Kích thước một block là 16x16 = 256 luồng tương ứng với một ma trận phần tử. Lưới được tạo ra với kích thước vừa đủ với số ma trận phần tử.

Các khối được yêu cầu thực thi một cách độc lập với nhau: các khối có thể được thực thi song song hoặc tuần tự. Tính độc lập này cho phép các khối có thể dễ dàng lập lịch trên bất kỳ GPU nào, không phụ thuộc số lượng SM. Điều này cũng cho phép lập trình viên dễ dàng viết các mã khả chuyển giữa

các họ GPU khác nhau. Số các khối trong một lưới thường phụ thuộc kích thước của dữ liệu được xử lý hơn là số các bộ vi xử lý trong hệ thống.

• Phân cấp bộ nhớ

Các luồng CUDA có thể truy cập dữ liệu trên nhiều không gian nhớ khác nhau trong suốt quá trình thực thi như được minh hoạ trong hình 4, mỗi luồng có một bộ nhớ địa phương riêng. Mỗi khối có một bộ nhớ dùng chung có thể truy nhập được từ tất cả các luồng trong khối. Cuối cùng, tất cả các luồng có thể cùng truy cập tới cùng bộ nhớ tổng thể.



Hình 4. Kiến trúc bộ nhớ

Ngoài các không gian nhớ kể trên, các GPU Tesla cũng có thêm hai dạng không gian nhớ chỉ đọc, có thể truy nhập được từ mọi luồng: bộ nhớ texture, và bộ nhớ hằng số.

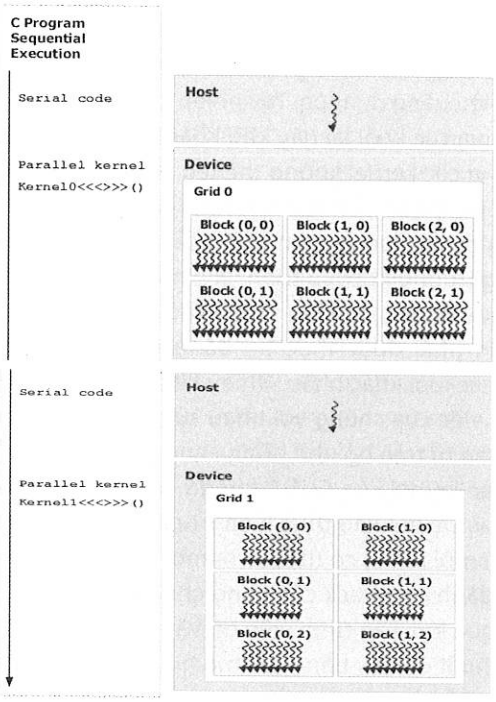
• Host và Device

Như được minh hoạ trong hình 5, CUDA giả thiết rằng các luồng CUDA được thực thi trên một thiết bị vật lý riêng biệt (device) song hành với chương trình chạy trên bộ xử lý trung tâm (CPU). Trong ví dụ, khi các kernel thực thi trên một GPU thì phần còn lại của chương trình C chạy trên CPU.

CUDA cũng giả thiết rằng cả host và device có bộ nhớ DRAM riêng, được tham chiếu tới như bộ nhớ host và bộ nhớ device. Các kernel chỉ được phép truy nhập trong vùng bộ nhớ device trong khi các đoạn mã thực thi trên CPU chỉ truy nhập được vùng bộ nhớ host. CUDA cung cấp các phương tiện để chuyển dữ liệu qua lại giữa hai vùng bộ nhớ này.

• Xếp chồng phần mềm (Software Stack)

Các lớp phần mềm của CUDA được cấu thành từ một số thành phần như được minh hoạ bởi hình 6: một device driver, một giao diện lập trình ứng dụng (API) và thực thi, và các thư viện toán học mức cao dùng chung như CUFFT và CUBLAS.

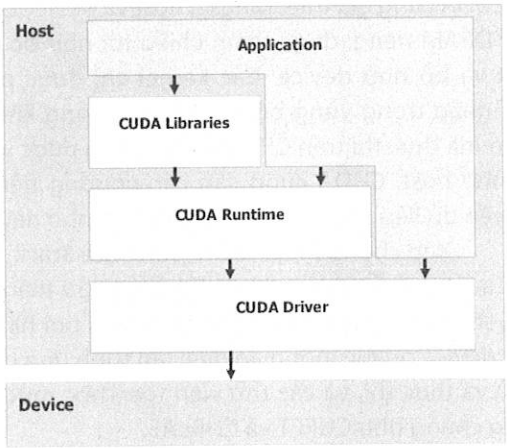


Hình 5. Chương trình không đồng nhất: mã thực thi trên host tuần tự trong khi mã thực thi trên device song song.

2.3. Những mặt hạn chế của CUDA

Một vấn đề quan trọng đối với các lập trình viên CUDA là phải nắm vững được những hạn chế của kiến trúc này để có thể tối ưu hóa được ứng dụng.

Trước tiên, các luồng và các khối được tạo bởi lời gọi tới một kernel duy nhất, do đó song song hóa dữ liệu là mô hình thiết kế hiệu quả các ứng dụng trên GPU. Các công việc được nhóm thành lưới của các khối luồng, và các khối luồng này độc lập với nhau. Sự độc lập này cho phép CUDA xây dựng



Hình 6. Kiến trúc phần mềm của CUDA

một bộ lập lịch đơn giản mà không tạo ra các chi phí hiệu năng dư thừa. Tuy nhiên cũng chính sự độc lập giữa các khối lại gây khó khăn đối với người lập trình vì các kernel không thể tạo ra được những rào chắn vượt qua biên giới của khối (ví dụ không đồng bộ được các luồng nằm trong các khối khác nhau). Trong trường hợp lập trình viên muốn tổng hợp các kết quả được tạo ra bởi nhiều khối, các khối này nói chung phải được thực thi bằng cách chạy kernel trên các lưới khác nhau. Nhiều khối có thể phối hợp công việc của chúng với nhau sử dụng các thao tác nguyên tử trên bộ nhớ dùng chung.

Các kernel của CUDA không hỗ trợ lời gọi hàm đệ qui. Đệ qui khó thực hiện trong nhân song song bởi vì mỗi luồng có thể tạo ra một stack riêng và bộ nhớ dành cho stack của hàng chục nghìn luồng trở nên qua lớn. Lập trình viên do vậy phải thay thế các giải thuật đệ quy bằng các mô hình song song lồng nhau. Để hỗ trợ một kiến trúc không đồng nhất bao gồm một CPU và một GPU, với mỗi hệ thống bộ nhớ riêng của chúng, các chương trình CUDA phải được copy dữ liệu và các kết quả giữa bộ nhớ host (CPU) và bộ nhớ thiết bị (GPU).

2.4. Thiết lập môi trường tính toán CUDA

2.4.1. Môi trường phần cứng

Vì các ứng dụng CUDA được viết để chạy trên các GPU của Nvidia, nên điều kiện đầu tiên là phải có một GPU mà tính năng CUDA trên đó được thiết lập sẵn. Hiện náy các dòng GPU thiết lập sẵn CUDA bao gồm các dòng:

• GeForce 8, 9, 100, 200. Bộ nhớ nhỏ nhất trên các GPU này là 256Mb

• Các GPU Tesla như S870, D870, C870, C1060, C1070,...

• Các GPU Quadro FX,...

2.4.2. Bộ công cụ phát triển phần mềm

Bộ công cụ phát triển CUDA của NVIDA gồm có 3 thành phần bao gồm:

1. Driver CUDA mới nhất.

2. Một bộ Toolkit CUDA bao gồm:

- nvcc C compiler
- CUDA FFT and BLAS libraries for the GPU
- Profiler
- gdb debugger for the GPU
- CUDA runtime driver (also available in the standard NVIDIA GPU driver)
- CUDA programming manual

3. Các đoạn mã ví dụ CUDA SDK

Bộ công cụ này có thể Download tại:

http://www.nvidia.com/object/cuda_get.htm

2.4.3. Quy trình cài đặt

Chúng ta tiến hành cài đặt bộ phát triển công cụ CUDA theo thứ tự như sau:

1. CUDA Driver
2. CUDA Toolkit
3. CUDA SDK Code Samples

3. KẾT LUẬN

Bài viết này đã cho chúng ta thấy được kiến trúc phần cứng của bộ Vi xử lý NVidia có hỗ trợ CUDA, và mô hình lập trình song song CUDA. Thông qua bài viết này chúng ta có thể nghiên cứu sâu hơn về dòng card đồ họa này, cũng như triển khai và xây dựng một số bài toán lập trình đòi hỏi hiệu năng cao, tốc độ xử lý lớn trên dòng card đồ họa của hãng NVidia.

TÀI LIỆU THAM KHẢO

[1]. NVIDIA CUDA Programming Guide 2.1 – NVIDIA Corporation

[2]. Scalable Parallel Programming with CUDA - Ohn Nickolls, Ian Buck, and Michael Garlan, NVIDIA, Kenvin Skadron University of Virginia

[3]. http://fastgpu.wordpress.com Xử lý song song (CPU+GPU).