

Cách thực thi song song trong CUDA

Trần Trung Kiên
ttkien@fit.hcmus.edu.vn

Cập nhật 03/11/2021



fit@hcmus

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Ôn lại các buổi trước

CUDA cho phép tổ chức grid cũng như block dưới dạng 1D hoặc 2D hoặc 3D

	Compute Capability												
Technical Specifications	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16	128	16	128		
Maximum dimensionality of grid of thread blocks	3												
Maximum x-dimension of a grid of thread blocks	2 ³¹ -1												
Maximum y- or z-dimension of a grid of thread blocks	65535												
Maximum dimensionality of a thread block	3												
Maximum x- or y-dimension of a block	1024												
Maximum z-dimension of a block	64												
Maximum number of threads per block	1024												

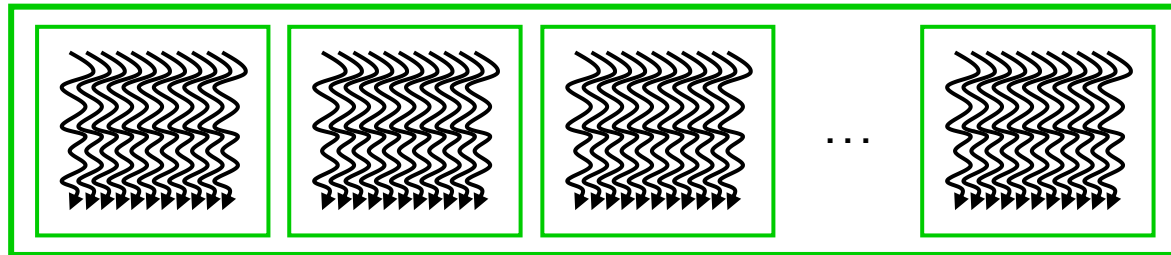
Hôm nay: CUDA level 2

Cách thực thi song song

- ☐ Ở mức các thread
- ☐ Ở mức các SM (Streaming Multiprocessor)
- ☐ Ở mức trong các SM

Cách thực thi song song – mức các thread

Tất cả các thread (được tổ chức thành các block) trong grid cùng thực thi song song hàm kernel



Các thread trong cùng một block có thể hợp tác được với nhau (sẽ được thấy ở các hàm kernel phức tạp hơn trong các buổi tới)

Cách thực thi song song – mức các SM

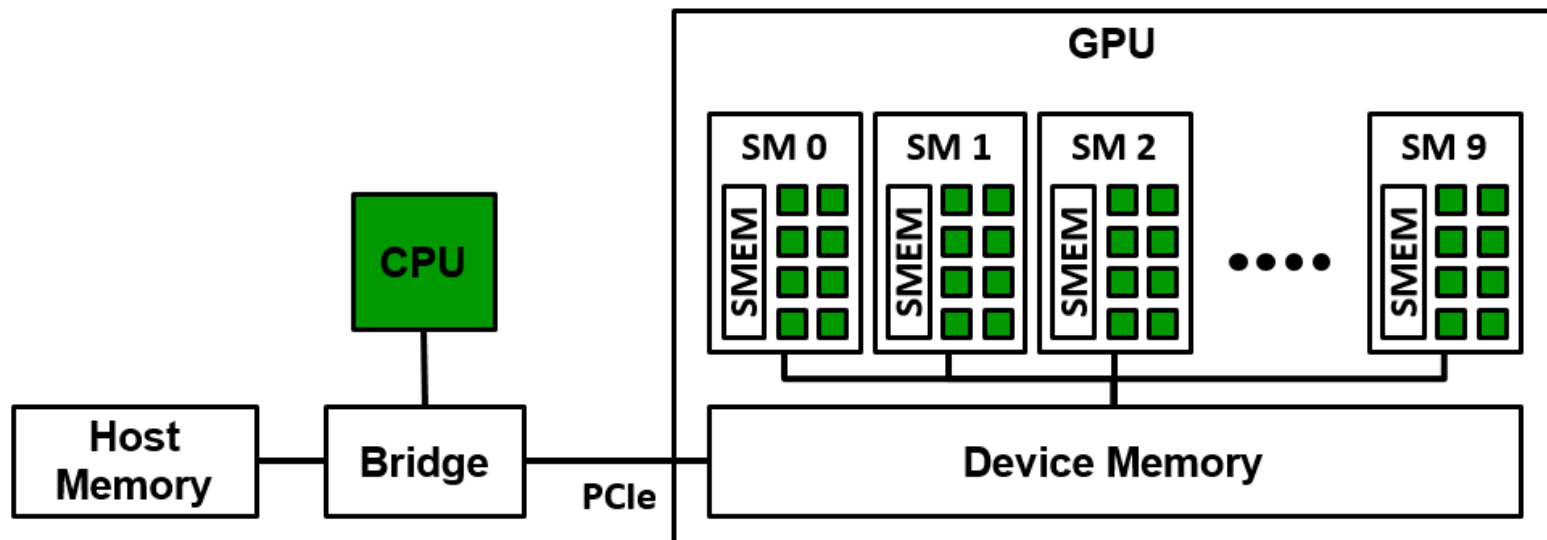
Kiến trúc phần cứng cơ bản của GPU

- GPU bao gồm các **SM (Streaming Multiprocessor)** – bộ xử lý đa luồng

Mỗi SM bao gồm các **SP (Streaming Processor)** – bộ xử lý luồng (còn gọi là CUDA core)

- “Compute capability” = SM version

(tính tới 21/10/2020 thì có: 2.x - Fermi, 3.x - Kepler, 5.x - Maxwell, 6.x - Pascal, 7.x - Volta/Turing, 8.x - Ampere)



Cách thực thi song song – mức các SM

- CUDA “ảo hóa” (virtualize) kiến trúc phần cứng của GPU
 - ▣ Block = bộ-xử-lý-đa-luồng “ảo”
 - ▣ Thread = bộ-xử-lý-luồng “ảo”
- Khi host gọi hàm kernel, hệ thống sẽ tạo ra một grid gồm các block và mỗi block (bộ xử lý đa luồng “ảo”) sẽ được phân vào một SM (bộ xử lý đa luồng thật) để thực thi
 - ▣ Mỗi SM có thể chứa nhiều hơn một block để thực thi
 - Tùy vào giới hạn tài nguyên của SM và tài nguyên mà mỗi block cần
 - Vd, SM cần tốn tài nguyên (thanh ghi) để lưu chỉ số block và thread cũng như là tình trạng thực thi của chúng, tài nguyên của SM 2.x chỉ đủ để lưu tối đa 8 block và 1536 thread

Nếu block gồm 512 thread thì SM 2.x chứa được ? block

Nếu block gồm 128 thread thì SM 2.x chứa được ? block

Cách thực thi song song – mức các SM

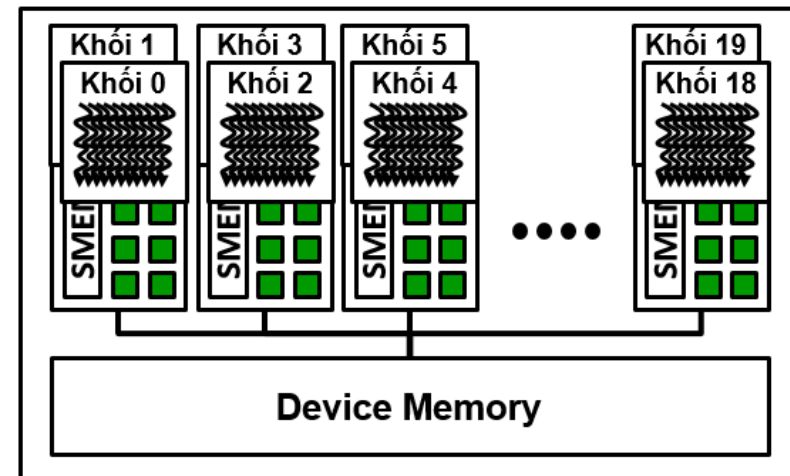
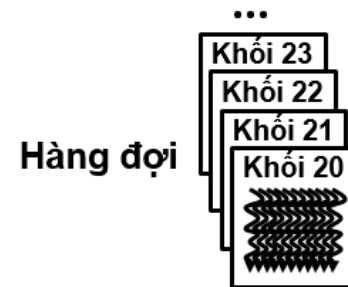
- CUDA “ảo hóa” (virtualize) kiến trúc phần cứng của GPU

- Block = bộ-xử-lý-đa-luồng “ảo”
- Thread = bộ-xử-lý-luồng “ảo”

- Khi host gọi hàm kernel, hệ thống sẽ tạo ra một grid gồm các block và mỗi block (bộ xử lý đa luồng “ảo”) sẽ được phân vào một SM (bộ xử lý đa luồng thật) để thực thi

- Mỗi SM có thể chứa nhiều hơn một block để thực thi
- Các block chưa được thực thi sẽ được đưa vào một hàng đợi
- Khi có một block được thực thi xong, hệ thống sẽ lấy một block chưa được thực thi ở hàng đợi và đưa vào thực thi

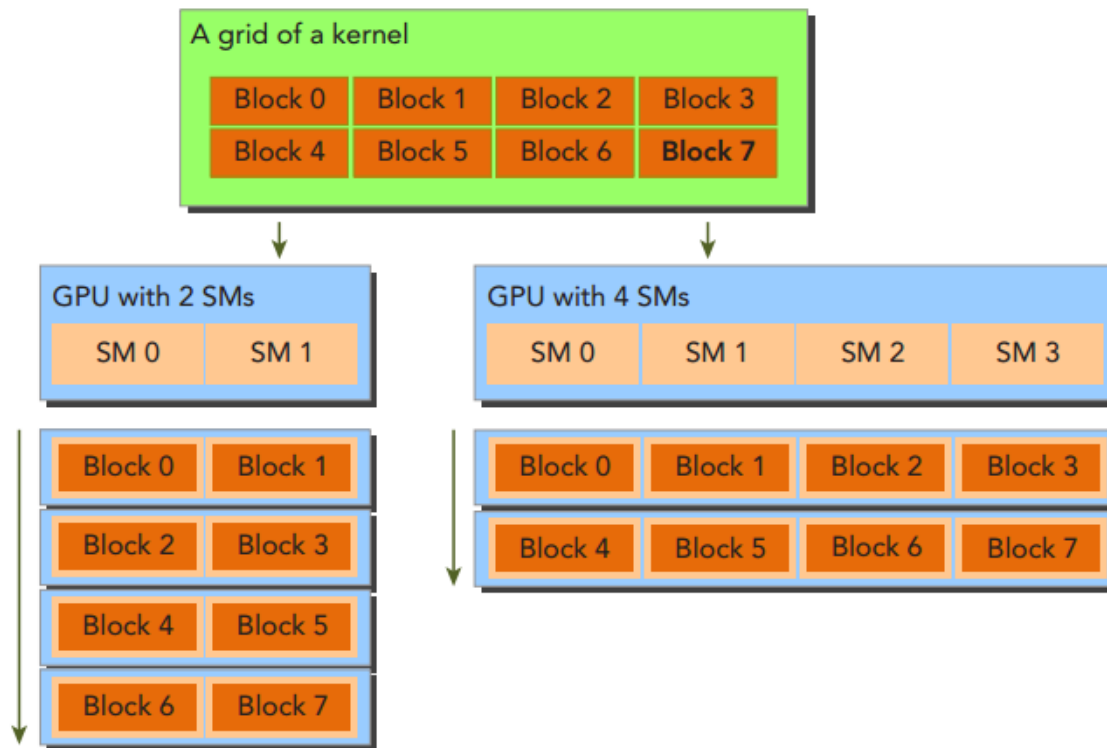
- Lưu ý: hệ thống có thể phân các block vào các SM theo một thứ tự bất kỳ; người lập trình không biết được



Cách thực thi song song – mức các SM

Cách thực thi của CUDA:

- Giúp đạt được tính “scalability” 😊



Cách thực thi song song – mức các SM

Cách thực thi của CUDA:

- ☐ Giúp đạt được tính “scalability” 😊
- ☐ Nhưng đòi hỏi các block phải độc lập với nhau
→ các thread thuộc các block khác nhau không thể hợp tác (đồng bộ hóa) được với nhau 😞
- ☒ Giả sử thread a ở block A muốn sử dụng kết quả của thread b ở block B,
và GPU chỉ đủ phần cứng để thực thi một block tại một thời điểm và đang thực thi block A
Nhưng block B chỉ được thực thi khi block A đã thực thi xong 😞
- ☒ Các thread thuộc cùng một block thì có thể hợp tác được với nhau bằng câu lệnh `__syncthreads()`

Cách thực thi song song – mức trong các SM

- ☐ Trong SM, với mỗi block, hệ thống không quản lý và thực thi riêng lẻ từng thread mà làm theo các **nhóm 32 thread** - gọi là **warp**
- ☐ Cách làm này được gọi là **SIMT (Single Instruction Multiple Thread)** - một câu lệnh được thực thi đồng thời cho tất các thread trong warp (mỗi thread có dữ liệu riêng của mình)
- ☐ Điểm lợi của cách thực thi này?
Giúp đơn hóa về mặt phần cứng

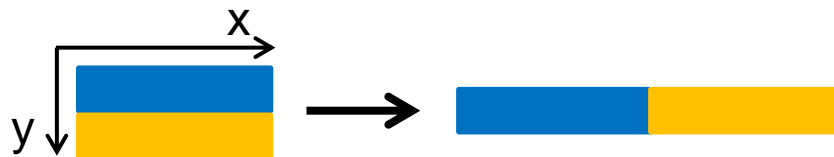
Cách thực thi song song – mức trong các SM

Cách SM chia block ra thành các warp

- **Block 1D:** 32 thread có chỉ số liên tục nhau tạo thành 1 warp (warp 1 gồm các thread có chỉ số 0-31, warp 2 gồm các thread có chỉ số 32-63, ...)

Nếu kích thước block không chia hết cho 32 thì warp cuối vẫn sẽ được thêm vào các thread cho đủ 32, các thread này tuy không làm gì cả nhưng vẫn sẽ chiếm tài nguyên

- **Block 2D:** chuyển sang dạng 1D rồi chia warp như 1D



- **Block 3D:** chuyển sang dạng 1D rồi chia warp như 1D



Cách thực thi song song – mức trong các SM

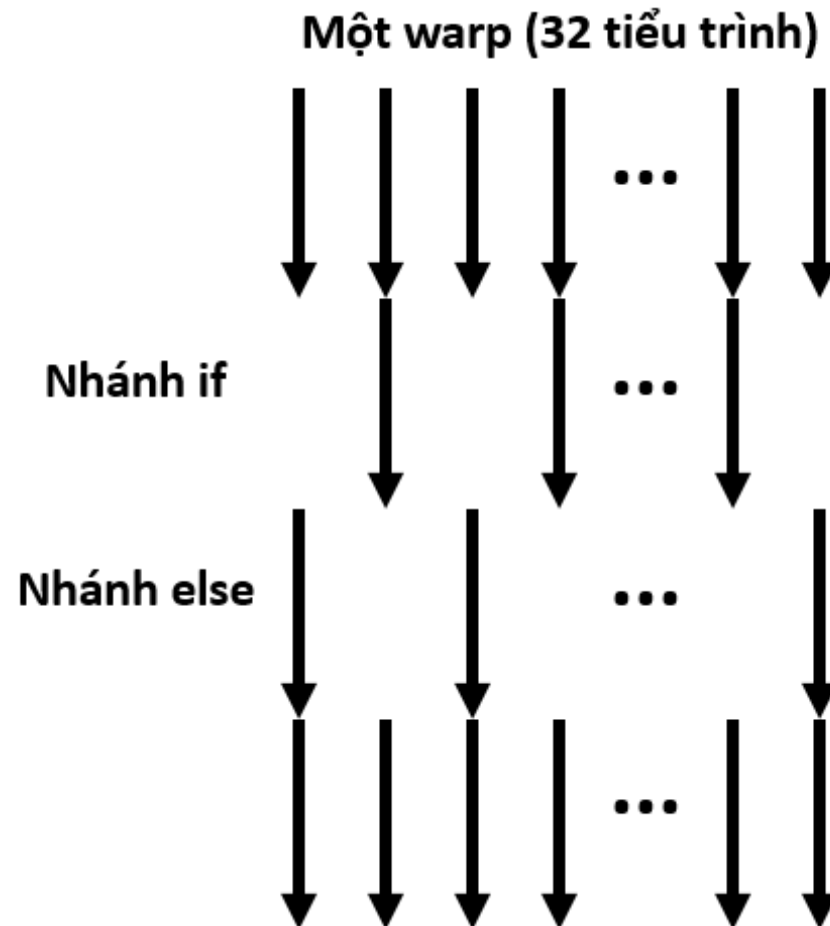
Nếu các thread trong warp không thể thực thi cùng một câu lệnh thì sao?

Warp bị phân kỳ (warp divergence)

- ❑ Tính đúng đắn? OK
- ❑ Tốc độ? Hmm...

Cách thực thi song song – mức trong các SM

Ví dụ về hiện tượng phân kỳ warp: câu lệnh rẽ nhánh

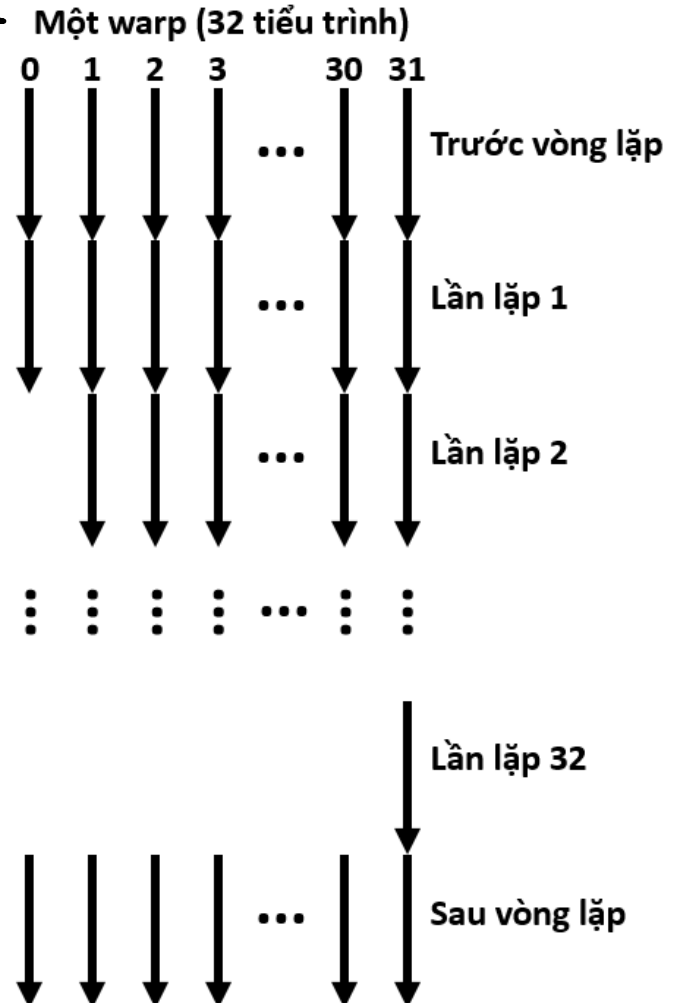


Cách thực thi song song – mức trong các SM

Ví dụ về hiện tượng phân kỳ warp: câu lệnh lặp

Xét warp gồm các thread có chỉ số từ 0-31

```
...
for (int i = 0; i <= threadIdx.x; i++)
{
    ...
}
...
```




Quiz: warp bị phân kỳ

Xét bài toán cộng 2 ma trận

- ☐ Ma trận có kích thước 1000x1000
- ☐ Mỗi thread phụ trách tính một phần tử trong ma trận kết quả
- ☐ Block có kích thước 32x32

Có bao nhiêu warp bị phân kỳ?


- A. 0
-  B. 1000
- C. 1024
- D. 2000
- E. Em không biết

Quiz: warp bị phân kỳ

Xét bài toán cộng 2 ma trận

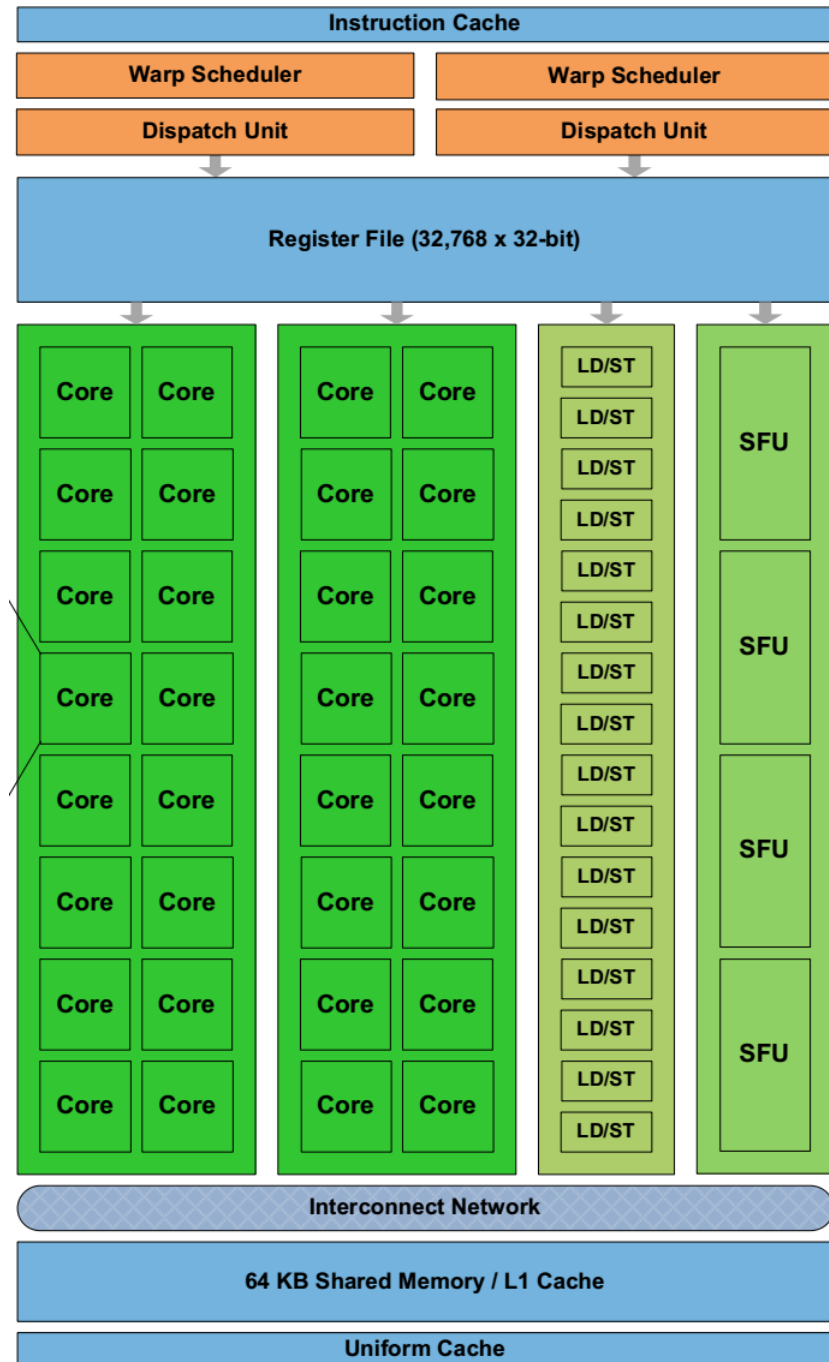
- ☐ Ma trận có kích thước 1000x1000
- ☐ Mỗi thread phụ trách tính một phần tử trong ma trận kết quả
- ☐ Block có kích thước 32x32

Thời gian thực thi của một warp bị phân kỳ so với một warp không bị phân kỳ (không xét warp mà 32 thread đều không tính toán)?

- A. Nhanh hơn
- B. Chậm hơn
-  C. Bằng nhau
- D. Em không biết

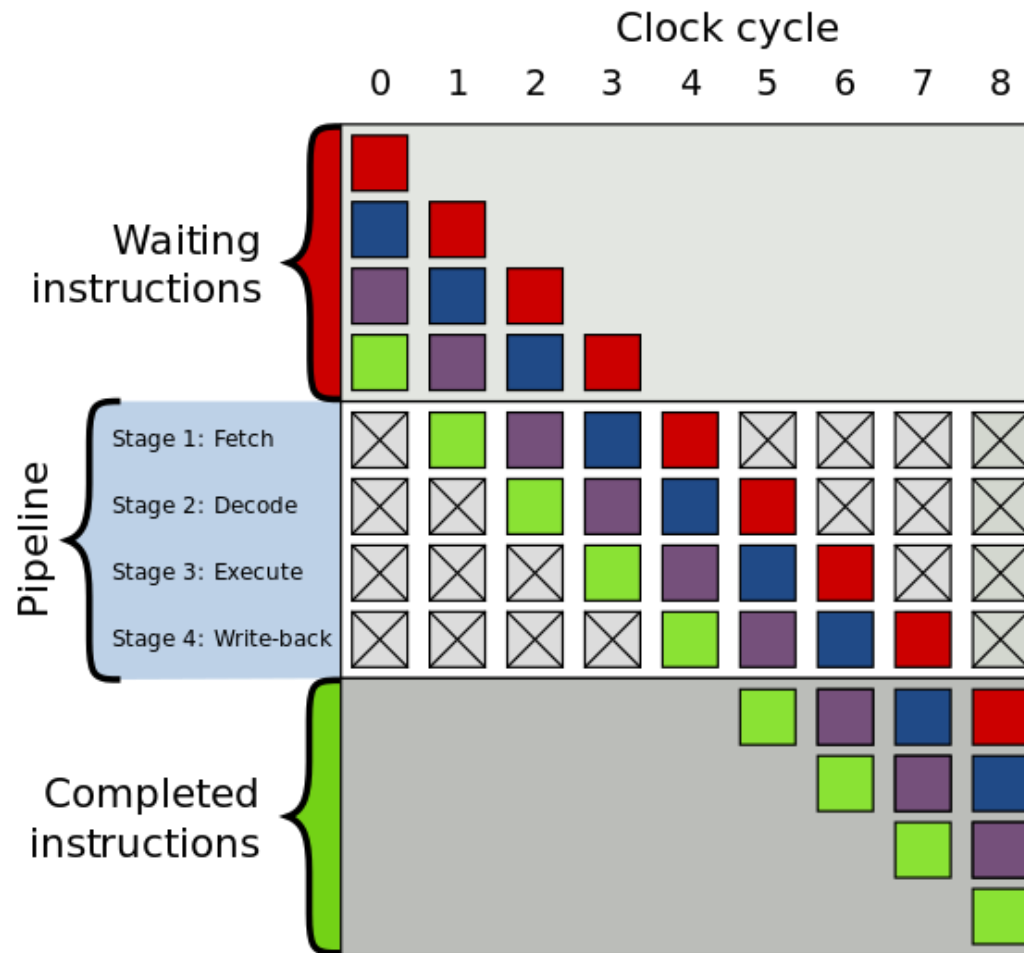
- ☐ Có phải trong SM các warp sẽ được thực thi song song với nhau?
 - ☐ Không hẳn là vậy. Vd, Fermi SM (2.x) có thể chứa tối đa 48 warp (1536 thread), nhưng chỉ có 32 core
- ☐ Vậy trong SM các warp được thực thi như thế nào?
- ☐ Tại sao SM lại chứa nhiều warp/thread trong khi tài nguyên thực thi (số core) lại rất ít (so với số lượng thread)?

Nguồn ảnh: NVIDIA. Fermi white paper



Fermi Streaming Multiprocessor (SM)

Trước khi đi tiếp, nhắc lại về “instruction pipeline”



Cách thực thi song song – mức trong các SM

- Trong SM, các warp được phân cho các **warp scheduler** (Fermi SM có 2 scheduler, Kepler SM có 4)
- Scheduler sẽ lần lượt gửi từng lệnh của một warp xuống các core (mỗi core là một pipeline; lý tưởng: 32 core cho 1 warp)
- Nếu sau khi gửi một câu lệnh của warp xuống mà *câu lệnh kế của warp đã sẵn sàng (độc lập với câu lệnh trước)* thì scheduler có thể gửi ngay câu lệnh kế xuống → giữ cho các pipeline luôn đầy

Scheduler trong Kepler SM có thể gửi một lúc hai câu lệnh liên tiếp của một warp xuống nếu hai câu lệnh này đều đã sẵn sàng (và các tài nguyên thực thi sẵn sàng)

- Nhưng nếu gặp phải một *câu lệnh chưa sẵn sàng để gửi xuống (do phụ thuộc vào một câu lệnh trước mà câu lệnh đó chưa thực thi xong)* thì warp sẽ bị đứng lại → scheduler đứng nhìn các pipeline bị lãng phí ☹, phải làm sao?

Scheduler sẽ chuyển sang gửi câu lệnh của một warp khác mà đã sẵn sàng

Che độ trễ (latency hiding)

Nếu một câu lệnh có độ trễ n chu kỳ đồng hồ thì scheduler sẽ cần $\sim n$ câu lệnh sẵn sàng (có thể đến từ cùng warp hoặc các warp khác) để “che” độ trễ, giữ cho các pipeline luôn đầy

- Độ trễ của các câu lệnh tính toán: 10-20 chu kỳ đồng hồ
- Độ trễ của các câu lệnh bộ nhớ: 400-800 chu kỳ đồng hồ

Ví dụ về số lượng warp cần cho Kepler SM

- Mỗi SM có 4 warp scheduler

→ cần 4+ warp / SM

Thực tế cần hơn 4 rất nhiều để có thể che độ trễ của câu lệnh (Kepler SM có thể chứa tối đa 64 warp)

- Với chương trình mà thời gian chủ yếu nằm ở các câu lệnh tính toán (độ trễ 10+ chu kỳ)

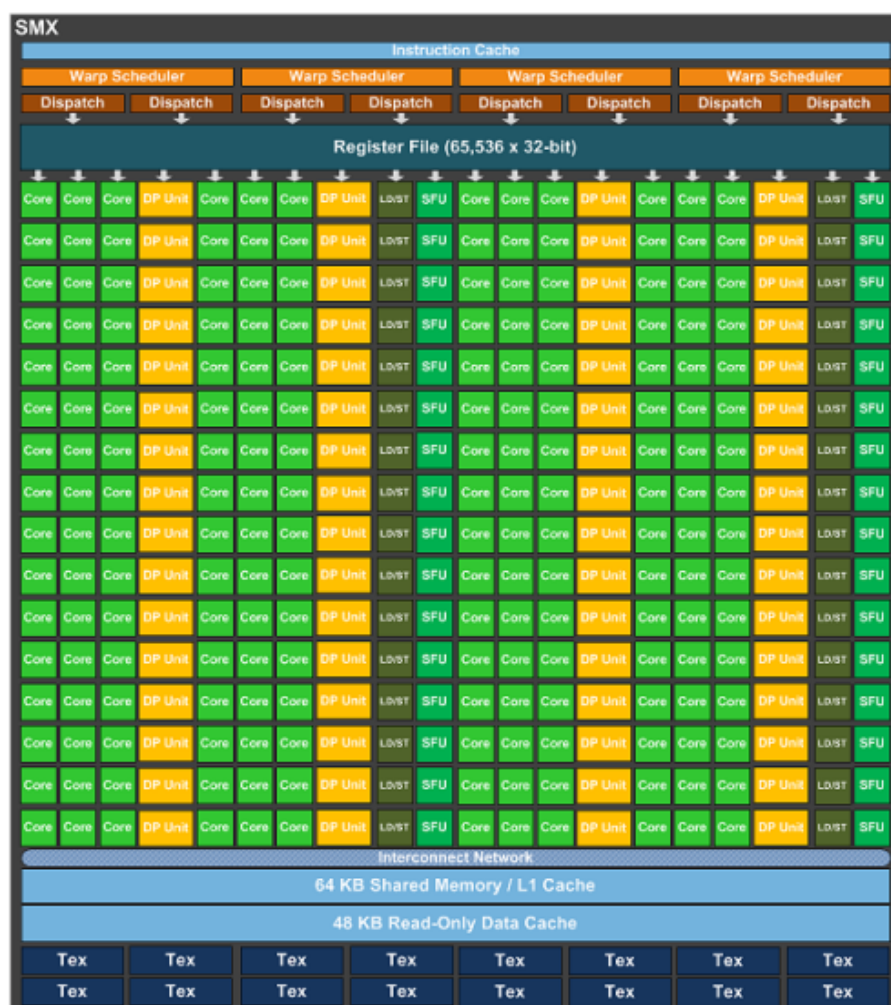
- Không có **ILP (Instruction Level Parallelism** – các câu lệnh độc lập liên tiếp nhau trong một warp):

cần 4 scheduler * 10+ chu kỳ = 40+ warp / SM

- Có ILP: có thể cần ít warp hơn

Nói thêm về các core

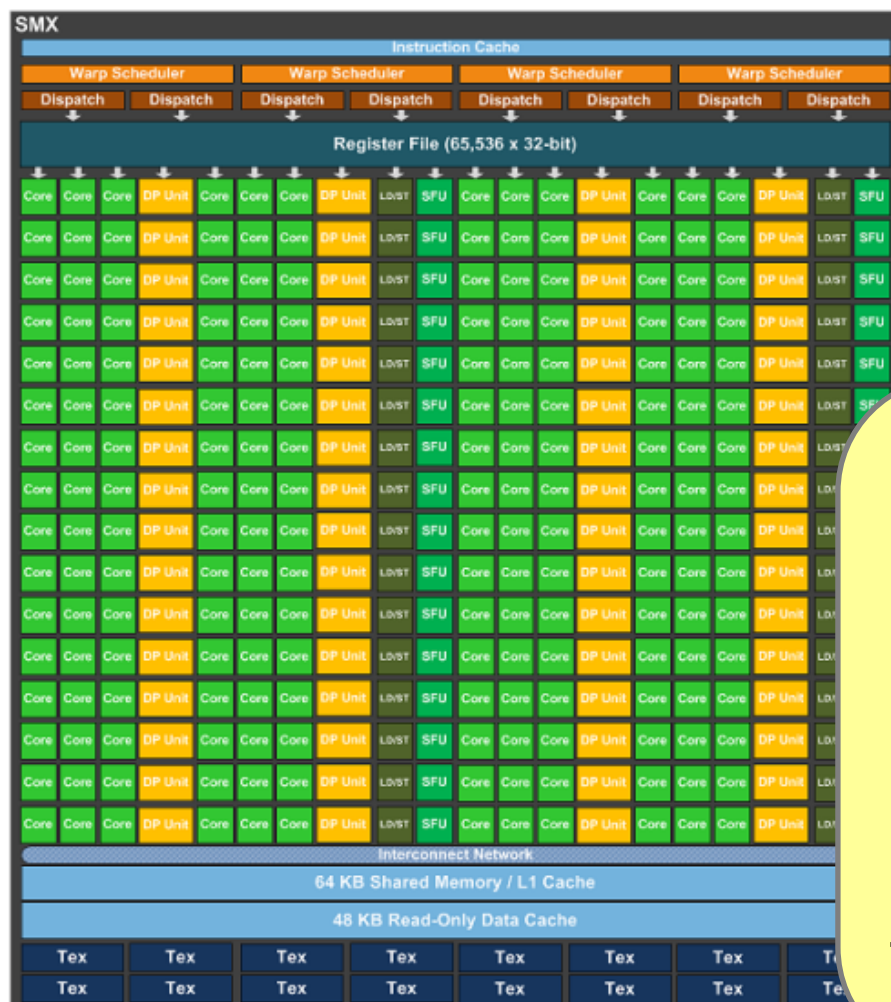
Ví dụ: Kepler SM



- **192 fp32 lanes (cores)**
 - fp32 math
 - Simple int32 math (add,min,etc.)
- **64 fp64 lanes**
- **32 SFU lanes**
 - Int32 multiplies, etc.
 - Transcendentals
- **32 LD/ST lanes**
 - GMEM, SMEM, LMEM accesses
- **16 TEX lanes**
 - Texture access
 - Read-only GMEM access

Nói thêm về các core

Ví dụ: Kepler SM



- **192 fp32 lanes (cores)**
 - fp32 math
 - Simple int32 math (add,min,etc.)
- **64 fp64 lanes**

Khái niệm “core” của NVIDIA chỉ ám chỉ loại core thực thi số thực và số nguyên 32 bit

Số lượng core thực thi số thực 32 bit thường sẽ nhiều hơn 64 bit
→ Nếu được thì nên dùng số thực 32 bit

Đố: chỉ có 4 scheduler, nhưng sao lại có tới 192 core số thực 32 bit?

Hướng dẫn chọn kích thước block

- Kích thước block nên chia hết cho 32 (kích thước warp)
- Kích thước block nên được chọn sao cho SM có đủ warp để che độ trễ và tận dụng hết tài nguyên
- Độ đo **occupancy**: tỉ lệ giữa số lượng warp có trong SM với số lượng warp tối đa mà SM có thể chứa
 - Nên chọn kích thước block sao cho occupancy lớn
 - Ví dụ: giả sử SM chỉ chứa được tối đa 8 block và 1536 thread (48 warp); nên chọn block có kích thước nào: 64, 256, 1024
- Không nhất thiết: 100% occupancy = max performance
 - Chỉ cần đủ warp để che độ trễ và tận dụng hết tài nguyên
 - Nếu trong warp có ILP thì có thể sẽ cần ít warp hơn
 - ...

Thí nghiệm

- ☐ Kích thước ma trận: $(2^{13} + 1) \times (2^{13} + 1)$
- ☐ Phát sinh ngẫu nhiên giá trị của các ma trận đầu vào trong $[0, 1]$
- ☐ So sánh thời gian chạy của `addMatOnDevice2D` với các kích thước block khác nhau
- ☐ GPU: GeForce GTX 560 Ti (CC 2.1, 8 SM)
Mỗi SM có thể chứa tối đa 8 block và 1536 thread (48 warp)

Kết quả thí nghiệm

Block size	Grid size	Occupancy (%)	Time (ms)
64 x 1	129 x 8193	33%	13.765
256 x 1	33 x 8193	100%	7.958
1024 x 1	9 x 8193	67%	12.685
16 x 16	513 x 513		

Kết quả thí nghiệm

Block size	Grid size	Occupancy (%)	Time (ms)
64 x 1	129 x 8193	33%	13.765
256 x 1	33 x 8193	100%	7.958
1024 x 1	9 x 8193	67%	12.685
16 x 16	513 x 513	100%	

Kết quả thí nghiệm

Block size	Grid size	Occupancy (%)	Time (ms)
64 x 1	129 x 8193	33%	13.765
256 x 1	33 x 8193	100%	7.958
1024 x 1	9 x 8193	67%	12.685
16 x 16	513 x 513	100%	10.904