Cách thực thi song song trong CUDA

Trần Trung Kiên – Phạm Trọng Nghĩa
ttkien@fit.hcmus.edu.vn
ptnghia@fit.hcmus.edu.vn

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

Ôn lại buổi trước

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

	Compute Capability										
Technical Specifications	**2.x**	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16		4	32				16	128	32	16
Maximum dimensionality of grid of thread blocks	3										
Maximum x-dimension of a grid of thread blocks	65535	$2^{31}-1$									
Maximum y- or z-dimension of a grid of thread blocks	65535										
Maximum dimensionality of 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										

Nguồn: http://docs.nvidia.com/cuda/

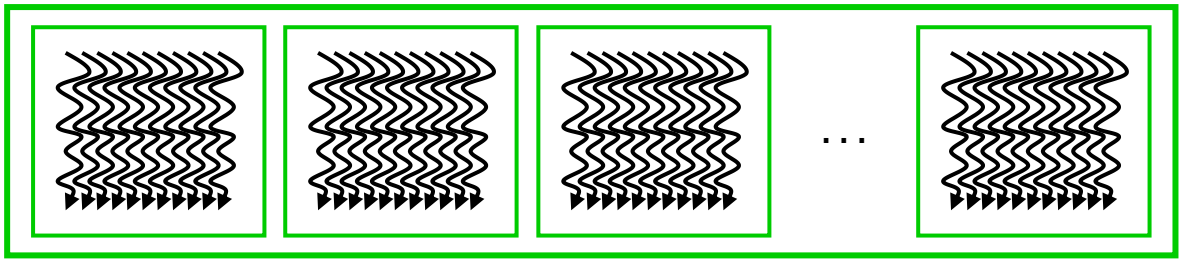
Hôm nay

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)

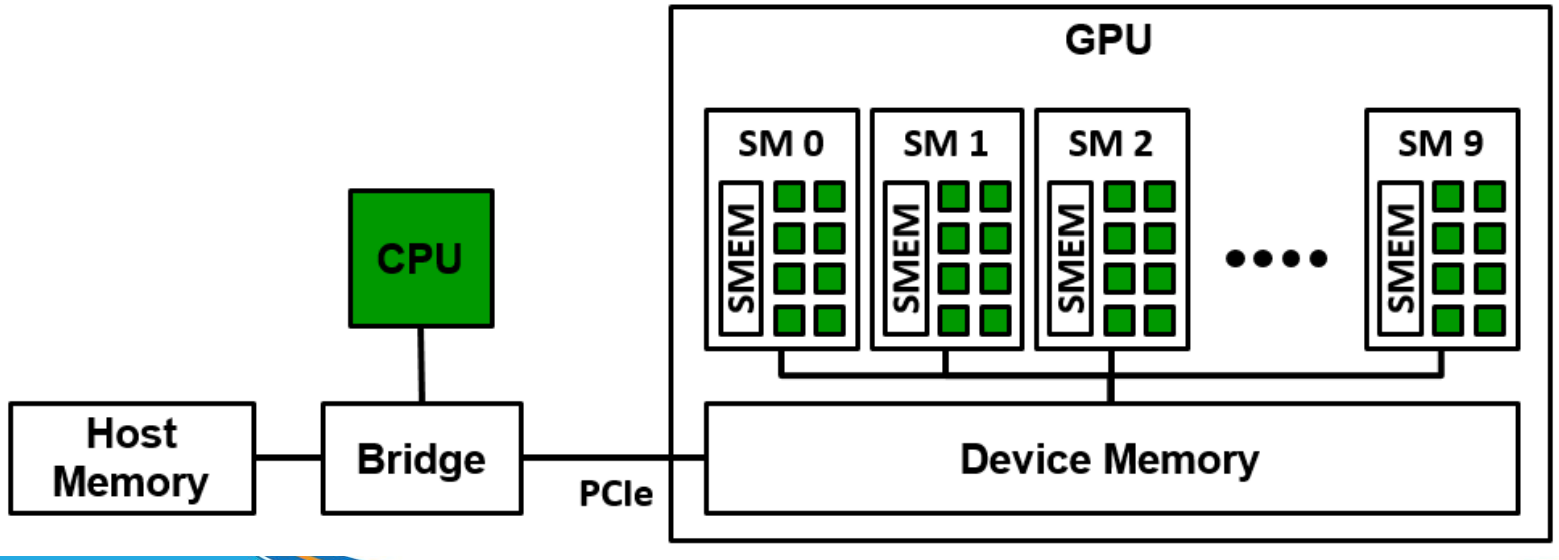
Nguồn ảnh: John Nickolls. Optimizing Parallel GPU Performance

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

  (hiện có: 2.x - Fermi, 3x - Kepler, 5.x - Maxwell, 6.x - Pascal, 7.x - Volta)



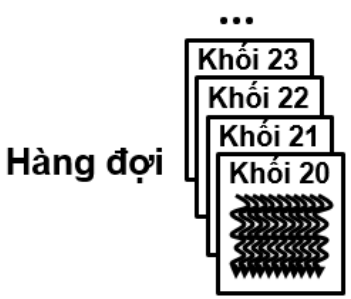
Cách thực thi song song – mức các SM

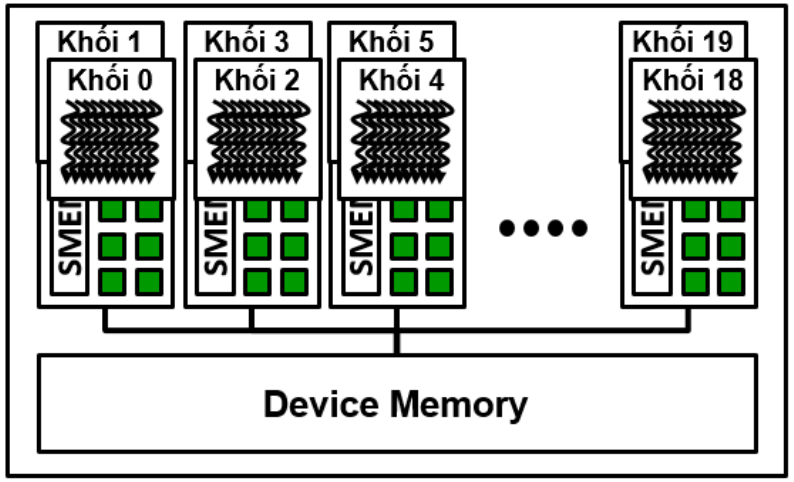
- CUDA "ảo hóa" (virtualize) kiến trúc phần cứng của GPU
  - Block = bộ-xử-lý-đa-luồng "ảo"
  - Thread = bộ-xử-lý-luồng "ảo"
- Khi host gọi hàm kernel, hệ thống sẽ tạo ra một grid gồm các block và mỗi block (bộ xử lý đa luồng "ảo") sẽ được phân vào một SM (bộ xử lý đa luồng thật) để thực thi
  - Mỗi 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

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



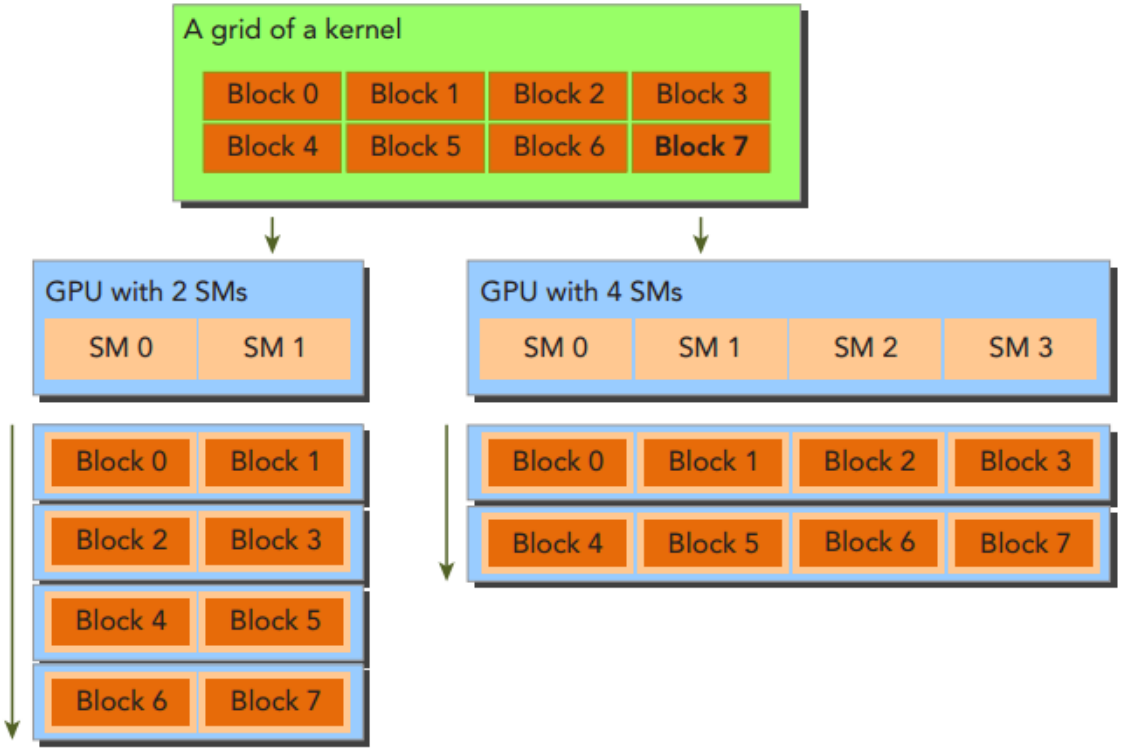


- Lưu ý: hệ thống có thể phân các block vào các SM theo một thứ tự bất kỳ; người lập trình không biết được

Cách thực thi song song – mức các SM

Cách thực thi của CUDA:

- Giúp đạt được tính "scalability" ☺



Nguồn ảnh: John Cheng et al. Professional CUDA C Programming. 2014

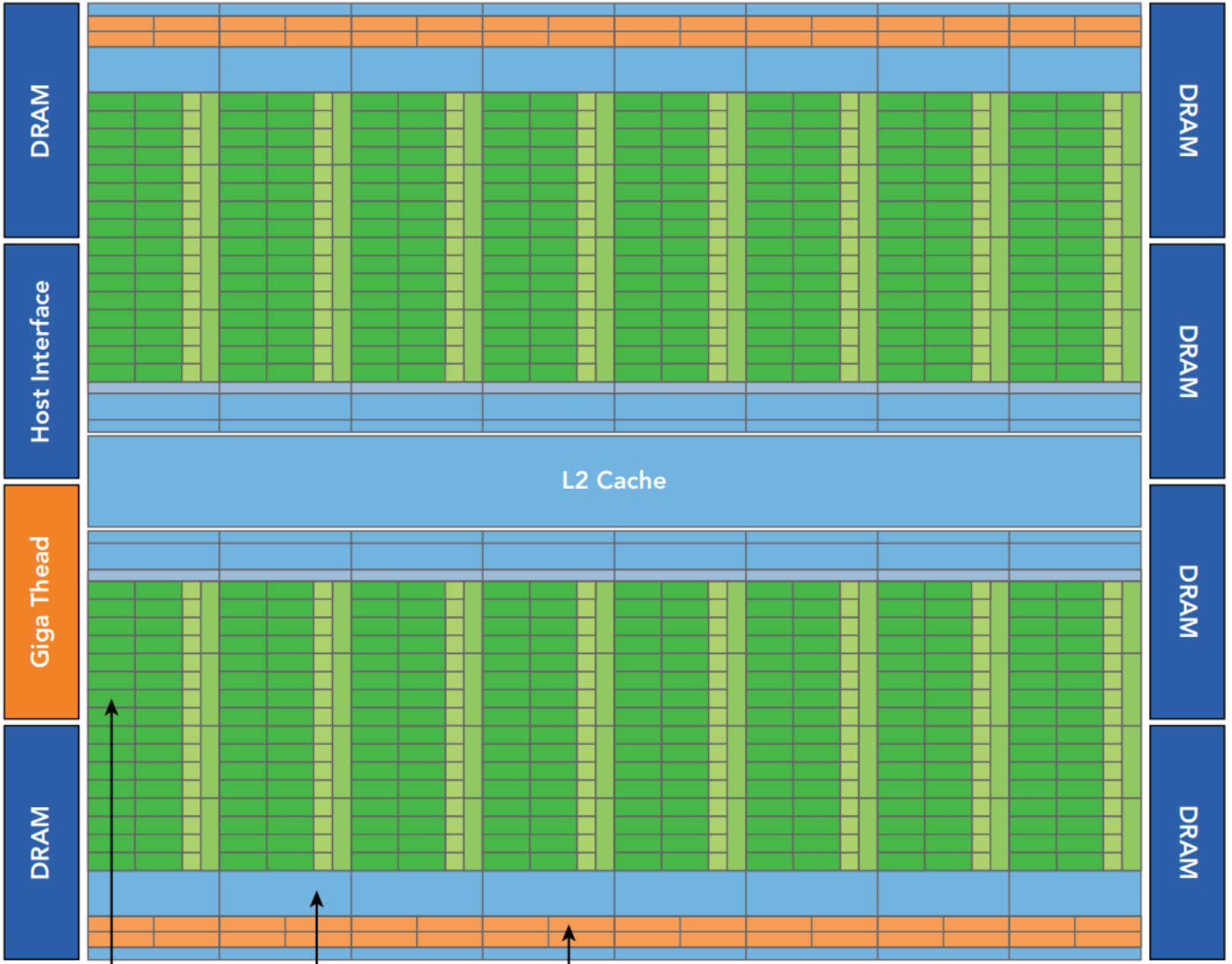
Cách thực thi song song – mức các SM

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

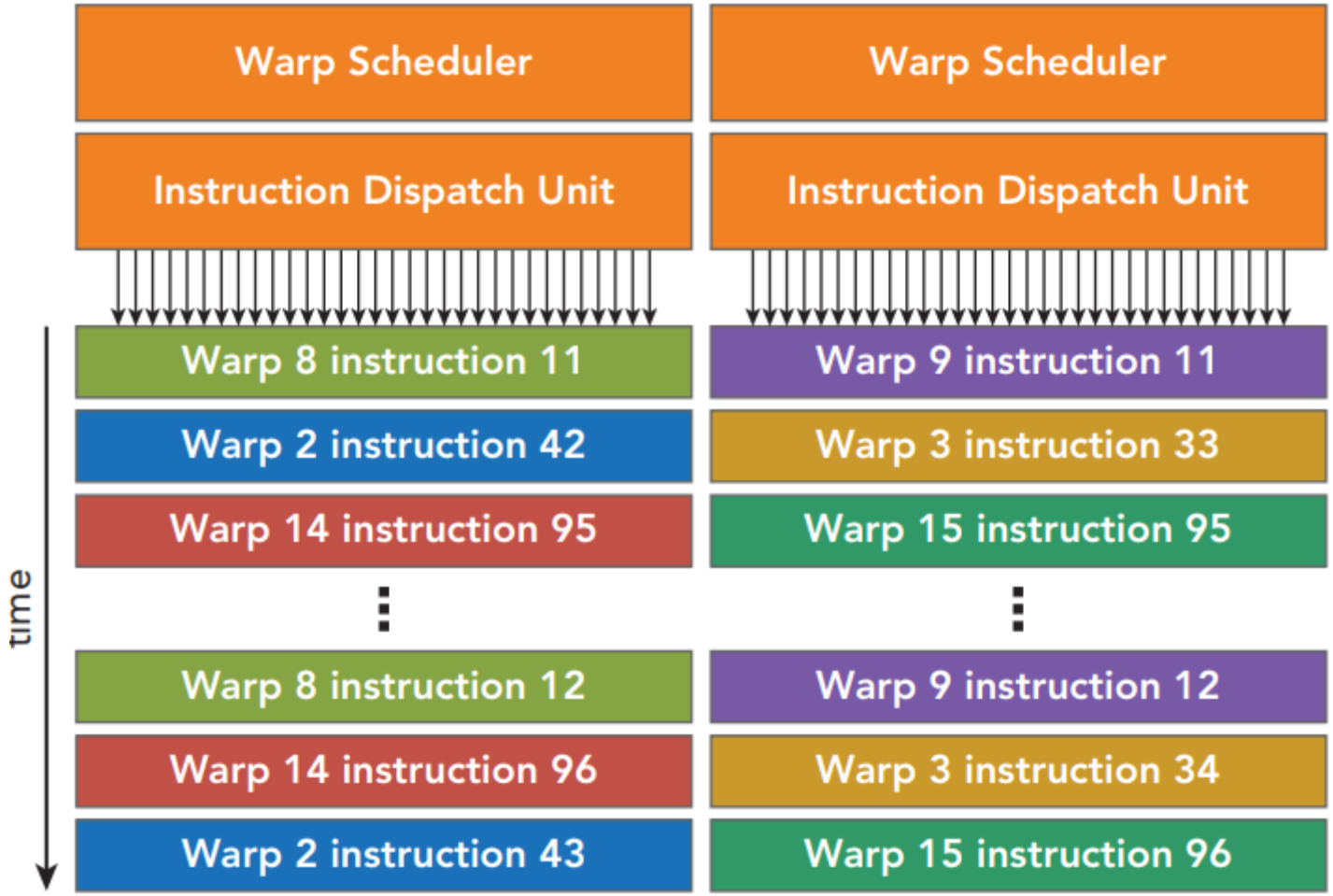
    Nhưng block B chỉ được thực thi khi block A đã thực thi xong ☹
  - Các thread thuộc cùng một block thì có thể hợp tác được với nhau bằng câu lệnh `__syncthreads()`


DRAM
Host Interface
Giga Thead
DRAM
DRAM
DRAM
DRAM
DRAM
DRAM
L2 Cache
CUDA core
Shared memory, register file, and L1 cache
Scheduler and dispatch units

Kiến trúc Fermi

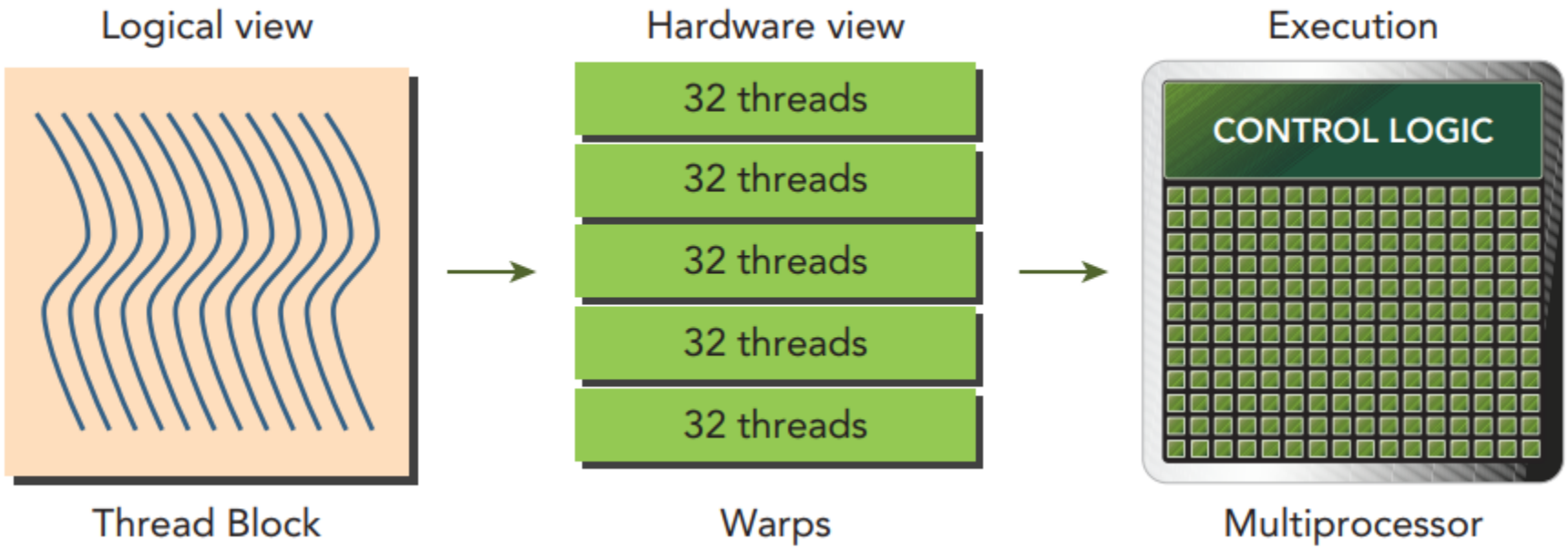
- 512 CUDA core.
- Mỗi core có một Arithmetic logic unit (ALU) và một floating-point unit (FPU)
- Các core được tổ chức thành 16 streaming multiprocessors (SM) ➔Mỗi SM gồm 32 CUDA core
- Gồm 6 DDRAM
- Host interface: kết nối CPU và GPU via PIC Ex
- GigaThread engine: global scheduler để phân phối các block và các SM.

Kiến trúc Fermi

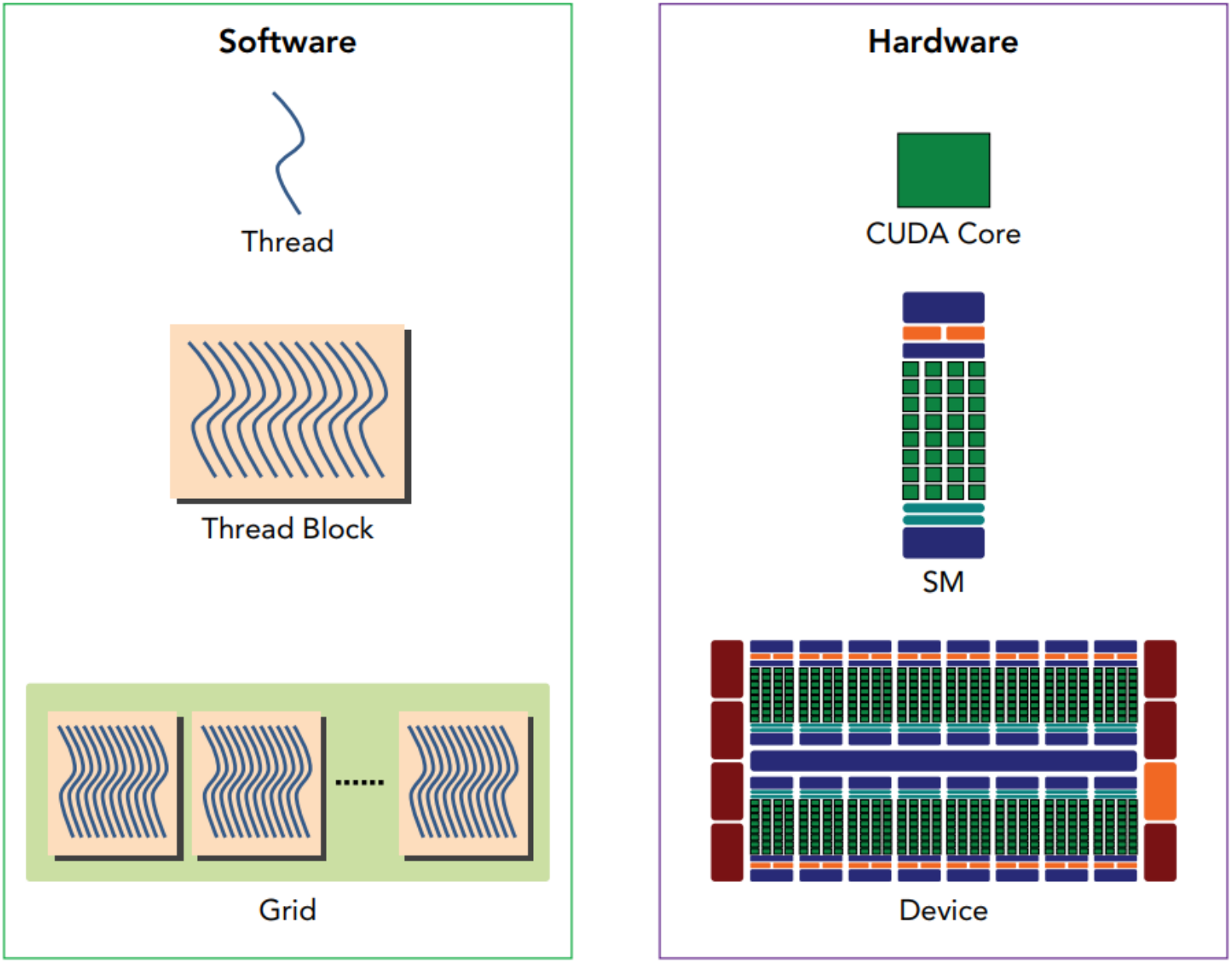


Cách thực thi song song – mức trong các SM

- 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


Logical view
Hardware view
Execution
32 threads
32 threads
32 threads
32 threads
32 threads
CONTROL LOGIC
Thread Block
Warps
Multiprocessor

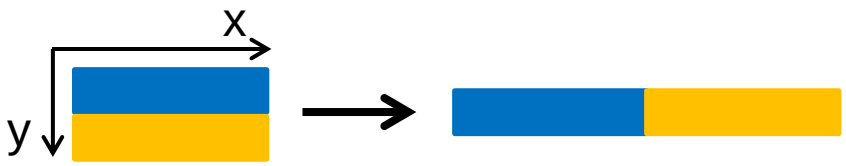
Logical view vs hardware view



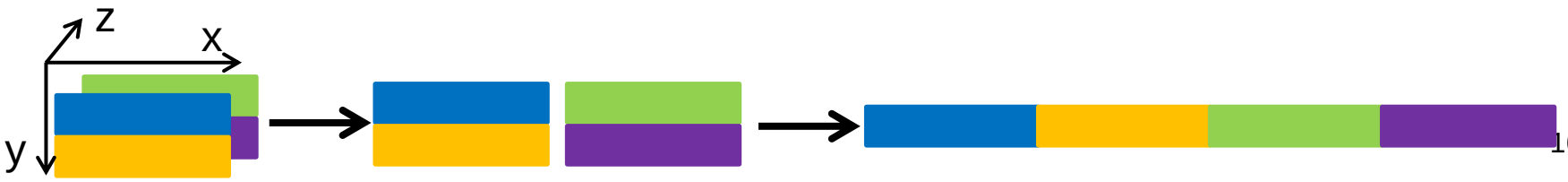
Cách thực thi song song – mức trong các SM

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

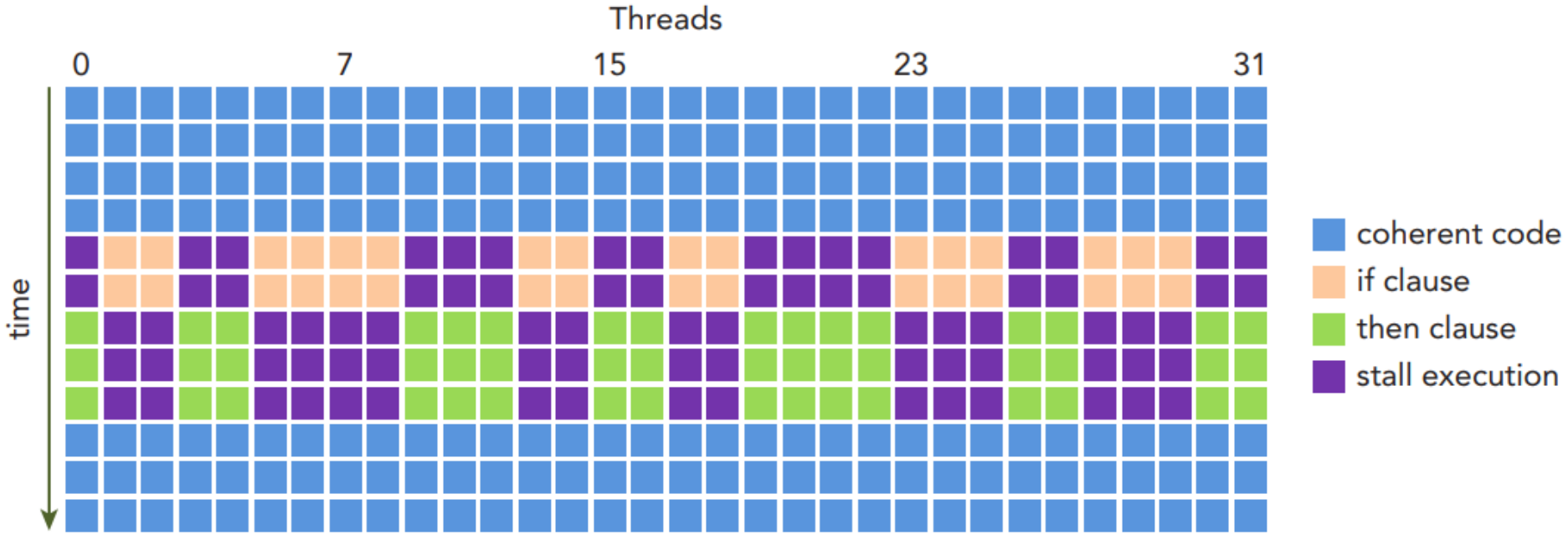


Cách thực thi song song – mức trong các SM

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

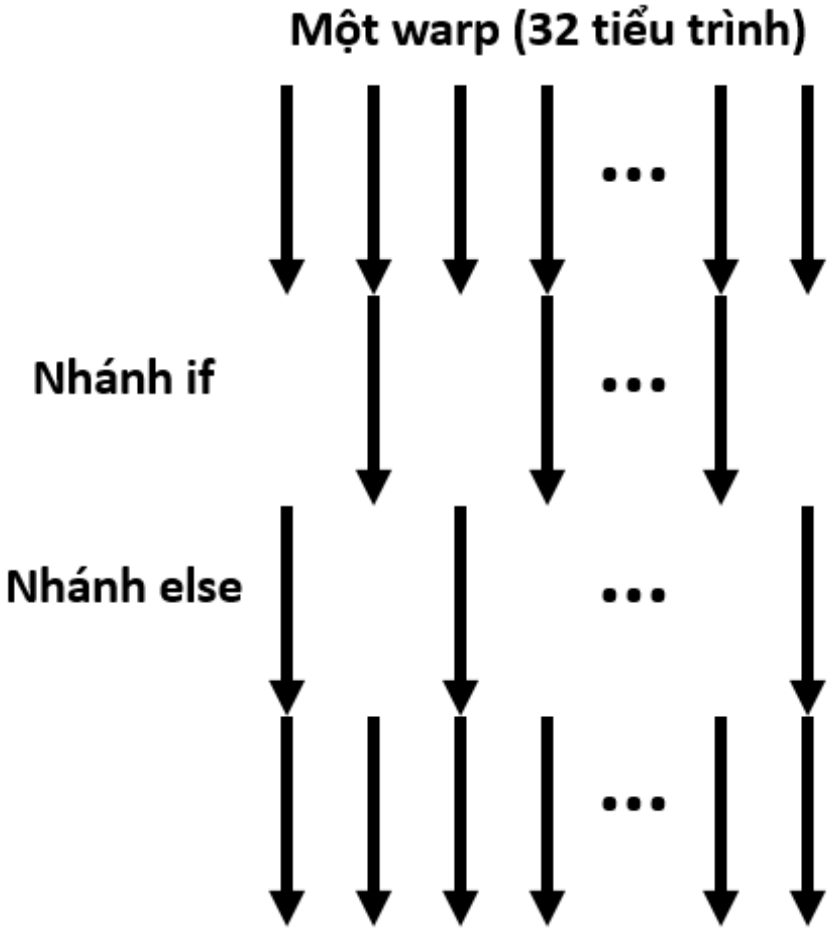
Warp bị phân kỳ (warp divergence)

- 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

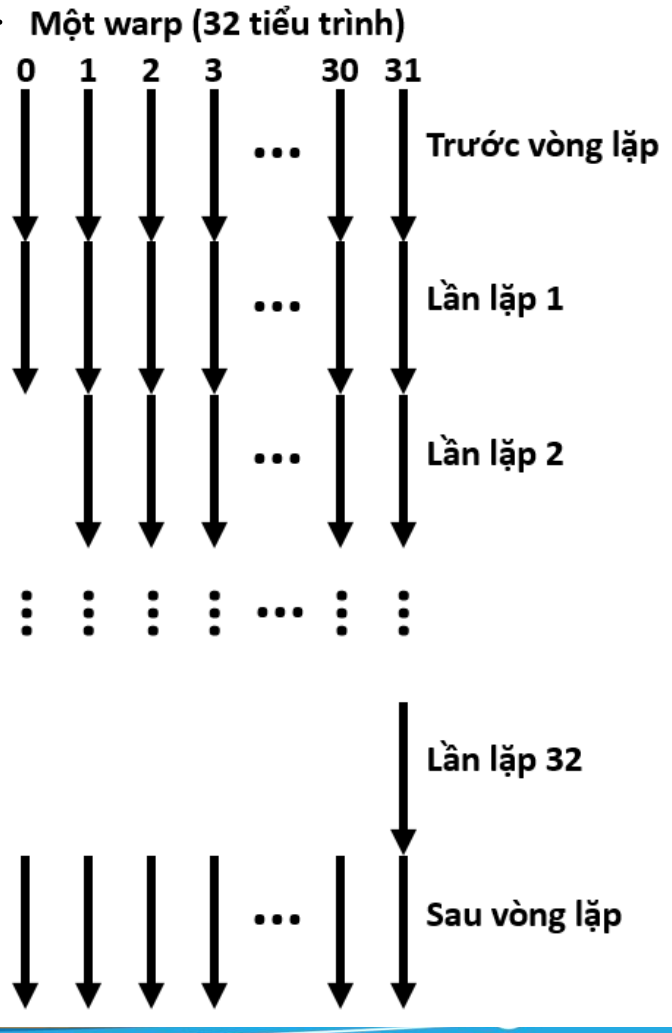


Cách thực thi song song – mức trong các SM

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

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

Thời gian thực thi của một warp bị phân kỳ so với một warp không bị phân kỳ?

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

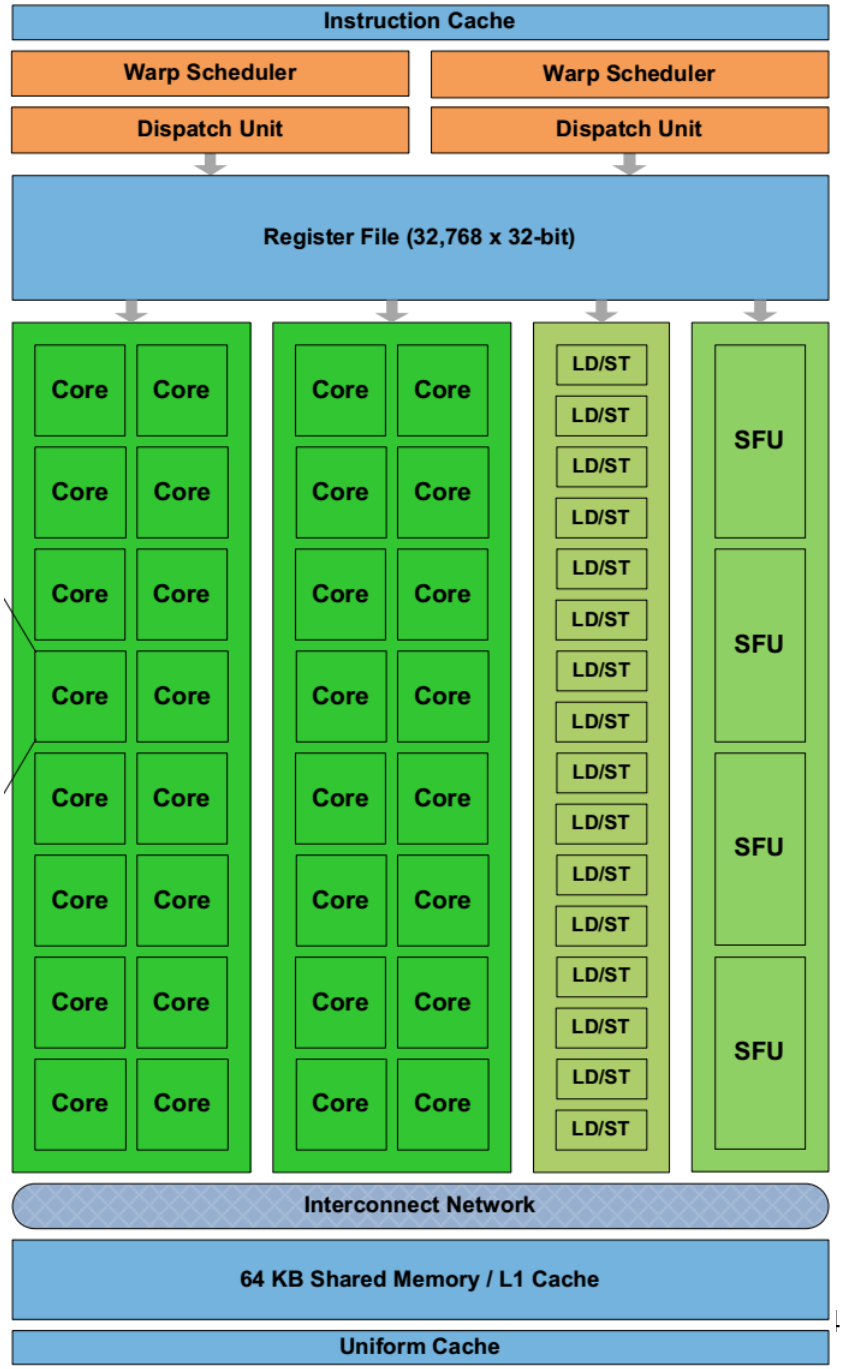
Ví dụ về phân kỳ

```
__global__ void mathKernel1(float *c) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float a, b;
    a = b = 0.0f;
    if (tid % 2 == 0) {
        a = 100.0f;
    } else {
        b = 200.0f;
    }
    c[tid] = a + b;
}
```

Ví dụ về phân kỳ

```
__global__ void mathKernel1(float *c) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float a, b;
    a = b = 0.0f;
    if ((tid / warpSize) % 2 == 0) {
        a = 100.0f;
    } else {
        b = 200.0f;
    }
    c[tid] = a + b;
}
```

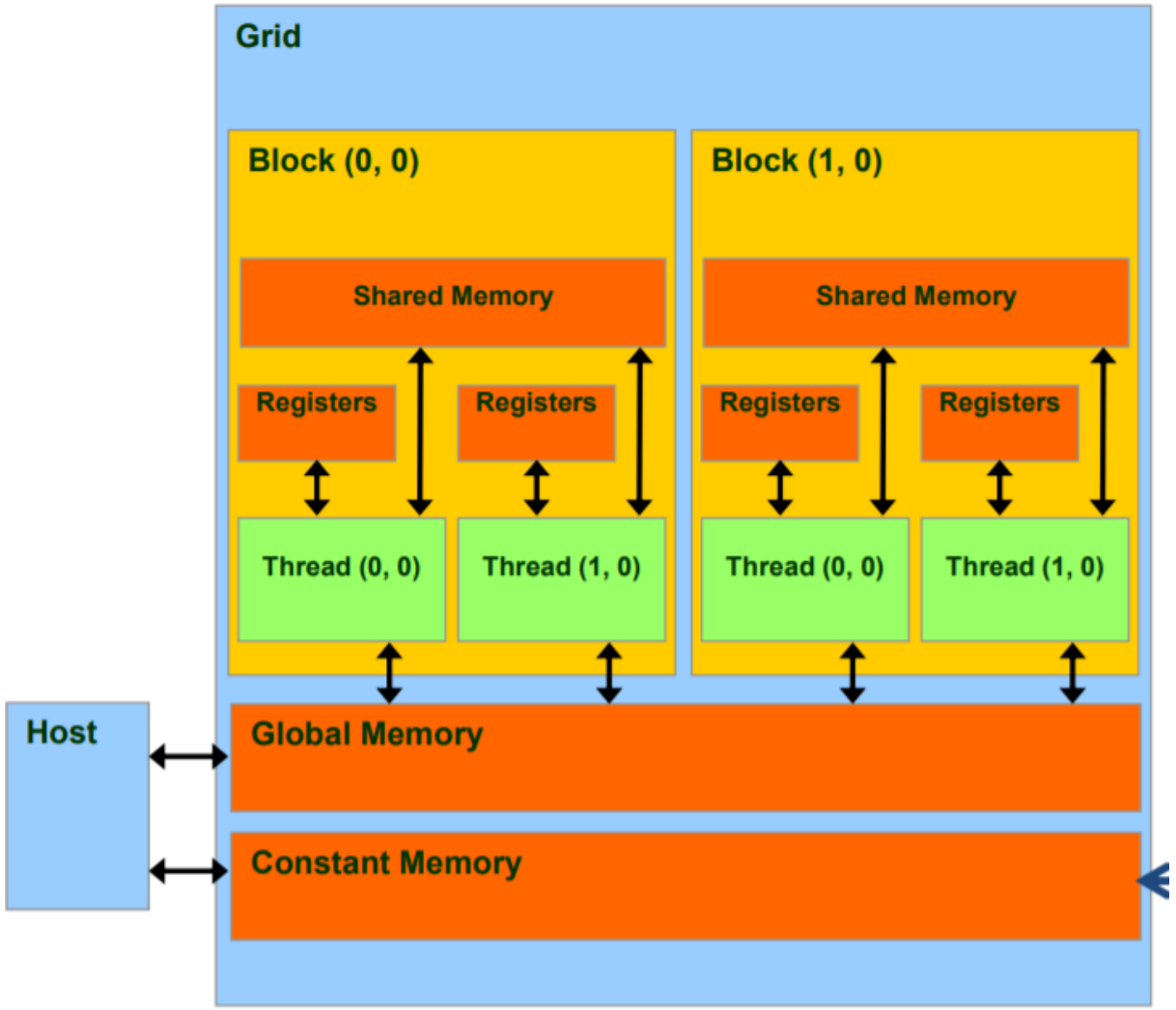
- Có phải trong SM các warp sẽ được thực thi song song với nhau?
  - Không hẳn là vậy. Vd, Fermi SM (2.x) có thể chứa tối đa 48 warp (1536 thread), nhưng chỉ có 32 core
- Vậy trong SM các warp được thực thi như thế nào?
- Tại sao SM lại chứa nhiều warp/thread trong khi tài nguyên thực thi (số core) lại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



Fermi Streaming Multiprocessor (SM)

Nguồn ảnh: NVIDIA. Fermi white paper

Sự phân chia tài nguyên



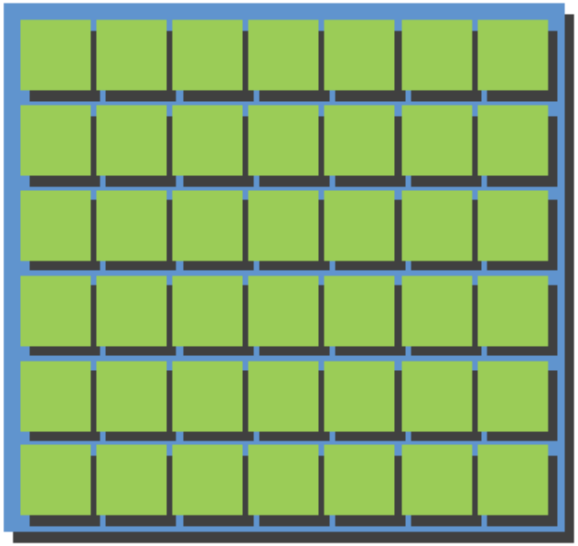
Sự phân chia tài nguyên

- Từng SM có:
  - 32-bit register: chia cho các thread
  - Shared memory: chia cho các block.
- Số lượng block và wrap tối đa cùng tồn tại trong một SM phụ thuộc vào 2 loại tài nguyên trên
- Giảm số lượng register kernel cần ➔ More warp
- Giảm số lượng shared memory kernel cần ➔ More block.

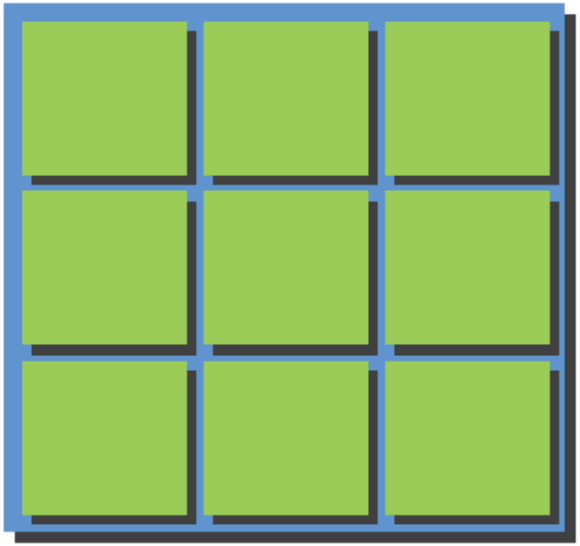
Registers per SM

Kepler: 64K

Fermi: 32K



More threads with fewer registers per thread

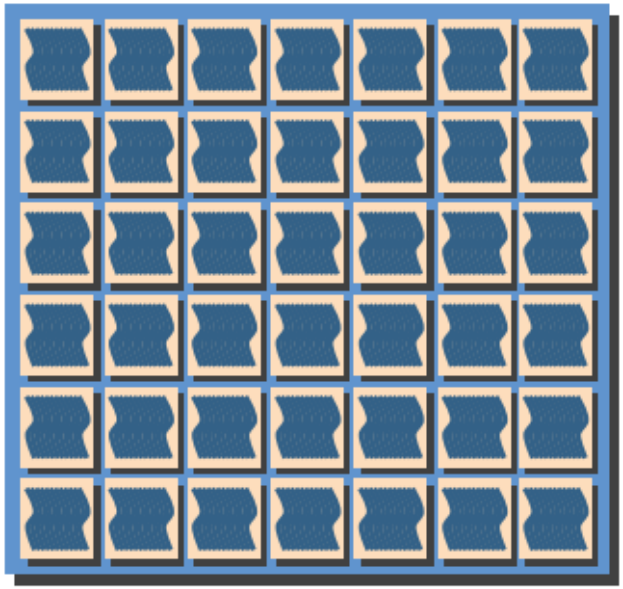


Fewer threads with more registers per thread

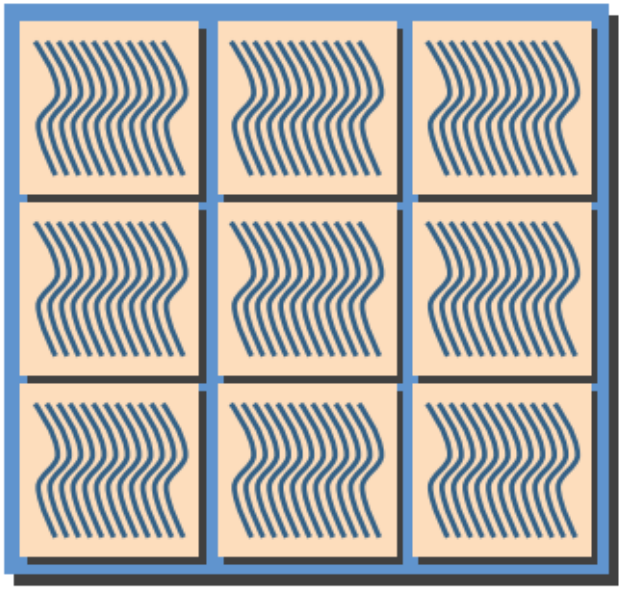
Shared Memory per SM

Kepler: up to 48K

Fermi: up to 48K



More blocks with less shared memory per block



Fewer blocks with more shared memory per block

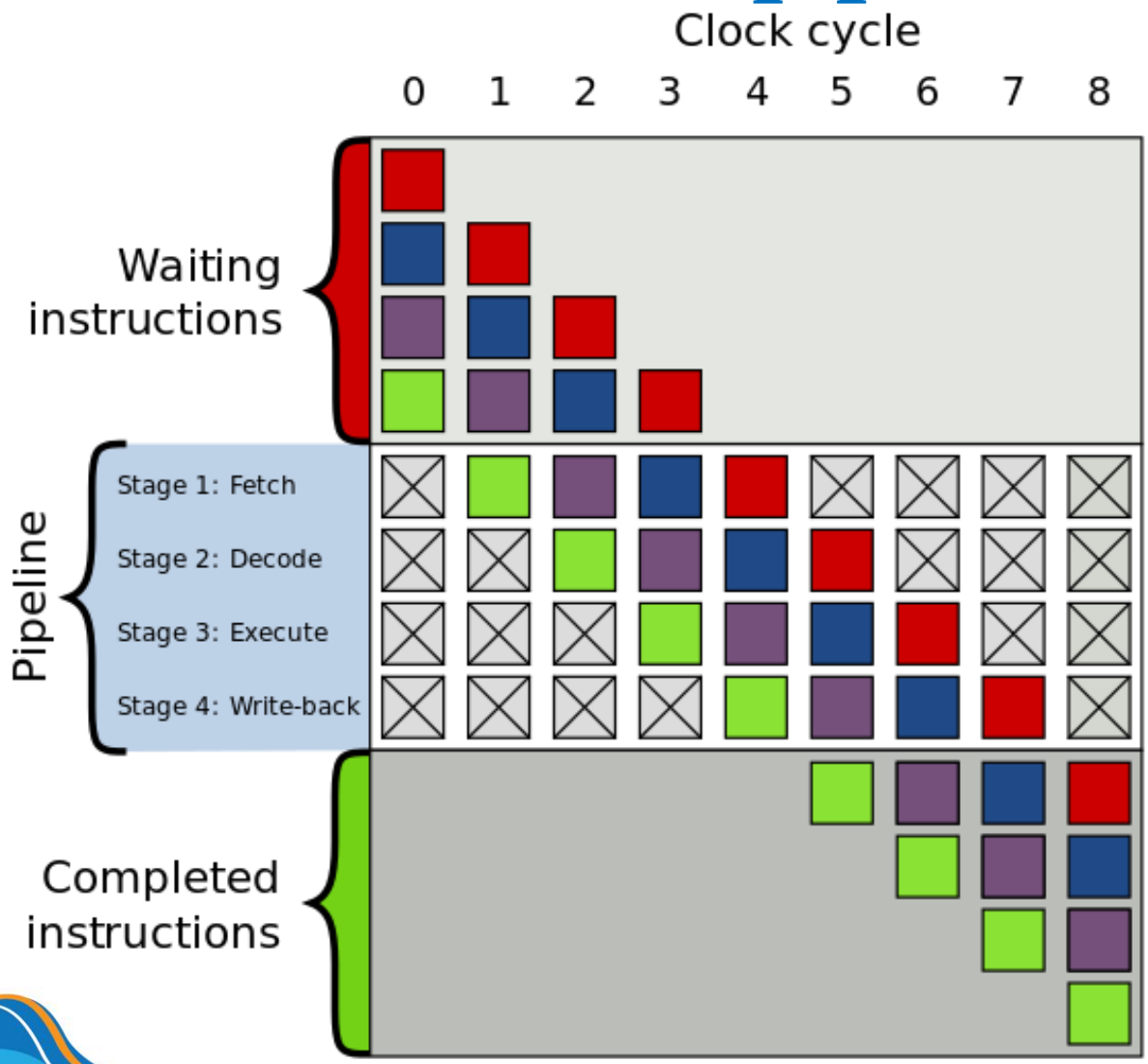
Giới hạn tài nguyên

TECHNICAL SPECIFICATIONS	COMPUTE CAPABILITY			
	2.0	2.1	3.0	3.5
Maximum number of threads per block	1,024			
Maximum number of concurrent blocks per multiprocessor	8		16	
Maximum number of concurrent warps per multiprocessor	48		64	
Maximum number of concurrent threads per multiprocessor	1,536		2,048	
Number of 32-bit registers per multiprocessor	32 K		64 K	
Maximum number of 32-bit registers per thread	63		255	
Maximum amount of shared memory per multiprocessor	48 K			

Sự phân chia tài nguyên

- Active block: một block khi đã được cấp phát tài nguyên.
- Active warp: các warp trong active block. Gồm 3 loại:
  - Selected warp: Đang thực thi
  - Stalled warp:
  - Eligible warp: sẵn sàng, nhưng chưa thực thi
- Kepler SM:
  - Active wrap: <= 64
  - Selected warp: <= 4

Trước khi đi tiếp, nhắc lại về "instruction pipeline"



Nguồn ảnh: https://en.wikipedia.org/wiki/Instruction_pipelining

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)
- Scheduler sẽ lần lượt gửi từng lệnh của một warp xuống các core (mỗi core là một pipeline)
- 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 độc lập với nhau (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 → lãng phí các pipeline, phải làm sao?
  - Scheduler sẽ chuyển sang gửi câu lệnh của một warp khác mà đã sẵn sàng

Che độ trễ (latency hiding)

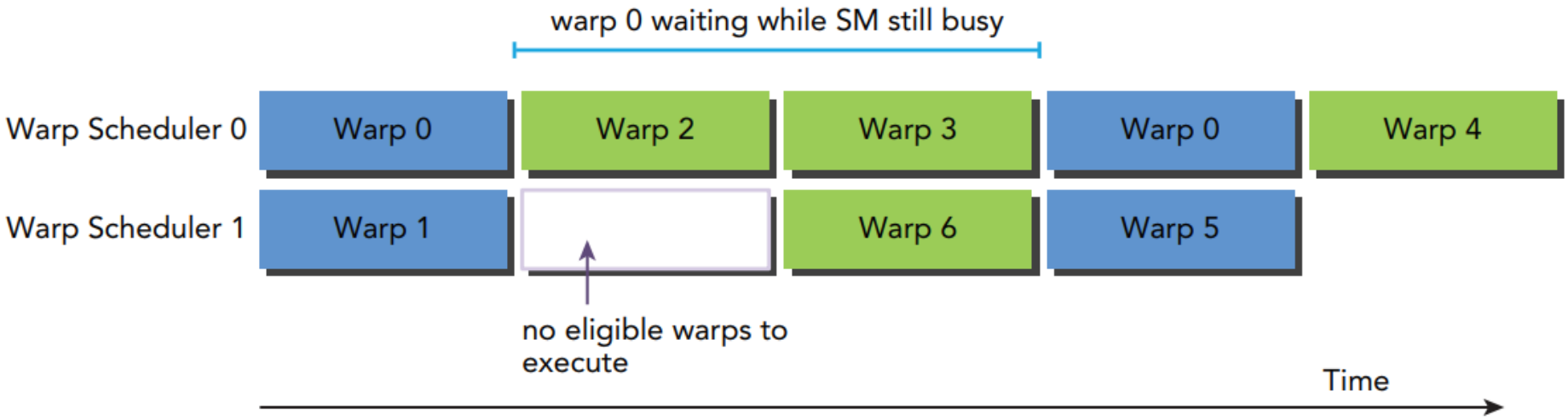
- Latency: số clock cycle cần thiết để xử lý một lệnh.
- Chương trình tận dụng tối đa tài nguyên:
  - Tại mỗi cycle luôn có eligible warp.
  - Cần maximize throughput
- Có hai loại instruction:
  - Arithmetic instructions
  - Memory instructions

Che độ trễ (latency hiding)

Nếu một câu lệnh có độ trễ **n** chu kỳ đồng hồ thì scheduler sẽ cần ~**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 toán học: 10-20 chu kỳ đồng hồ
- Độ trễ của các câu lệnh truy xuất global memory: 400-800 chu kỳ đồng hồ

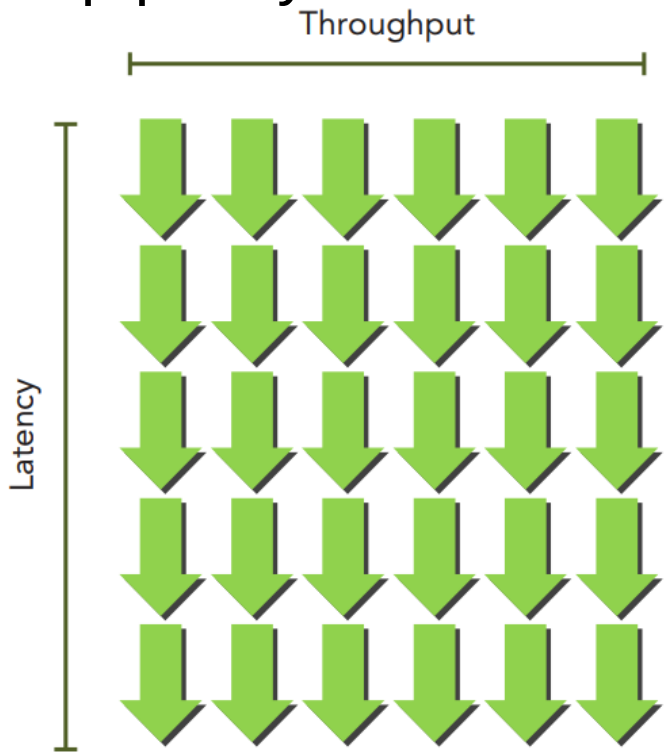
Execution pipeline



Little's Law

$$Number\ of\ Required\ Warps = Latency \times Throughput$$

- Một câu lệnh:
  - Latency: 5 cycle
  - Cần giữa throughput 6 warp per cycle
  - Cần bao nhiêu warp?



Parallelism cho phép toán số học

GPU MODEL	INSTRUCTION LATENCY (CYCLES)	THROUGHPUT (OPERATIONS/CYCLE)	PARALLELISM (OPERATIONS)
Fermi	20	32	640
Kepler	20	192	3,840

- Độ song song cần thiết cho phép toán số học biểu diễn bằng số lượng phép tính hay số warp.
- Hai cách để tăng tính song song:
  - Instruction-level parallelism (ILP): thêm các câu lệnh độc lập nhau trong một thread
  - Thread-level parallelism (TLP): cần thêm eligible threads.

Ví dụ về số lượng warp cần cho Kepler SM

- Mỗi SM có 4 warp scheduler → cần 4+ warp / SM
  - Thực tế cần nhiều hơn 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

Parallelism cho truy cập bộ nhớ

GPU MODEL	INSTRUCTION LATENCY (CYCLES)	BANDWIDTH (GB/SEC)	BANDWIDTH (B/CYCLE)	PARALLELISM (KB)
Fermi	800	144	92	74
Kepler	800	250	96	77

- Cần đọc một số thực 4 bytes. Cần bao nhiêu thread/warp để che độ trễ. (1.566 Ghz)

144 GB/Sec ÷ 1.566 GHz ≅ 92 Bytes/Cycle

74 KB ÷ 4 bytes/thread ≅ 18,500 threads

18,500 threads ÷ 32 threads/warp ≅ 579 warps

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

Nguồn ảnh: Paulius Micikevicius. Performance Optimization, GTC2013

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**

Khái niệm "core" của NVIDIA chỉ ám chỉ loại đơn vị thực thi số thực và số nguyên 32 bit; ngoài ra, còn có các loại đơn vị thực thi khác

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

Nguồn ảnh: Paulius Micikevicius. Performance Optimization, GTC2013

Hướng dẫn chọn kích thước block

- Số lượng thread chia hết cho 32 (kích thước warp)
- SM có đủ warp và ILP trong mỗi 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
    - 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
    - …

$$occupancy = \frac{active\ warps}{maximum\ warps}$$

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	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

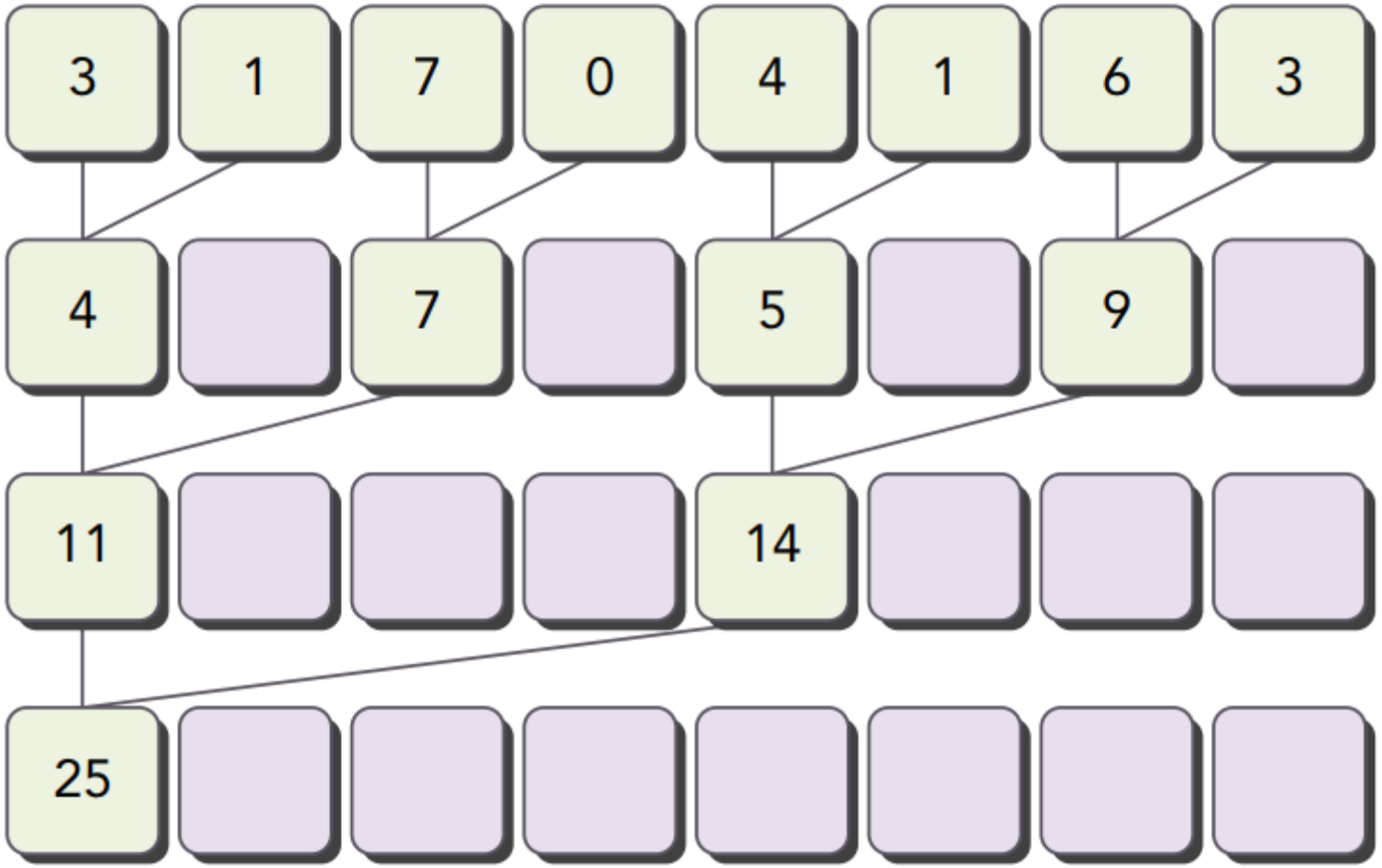
Bài toán áp dụng

- Tính tổng các phần tử trong một mảng

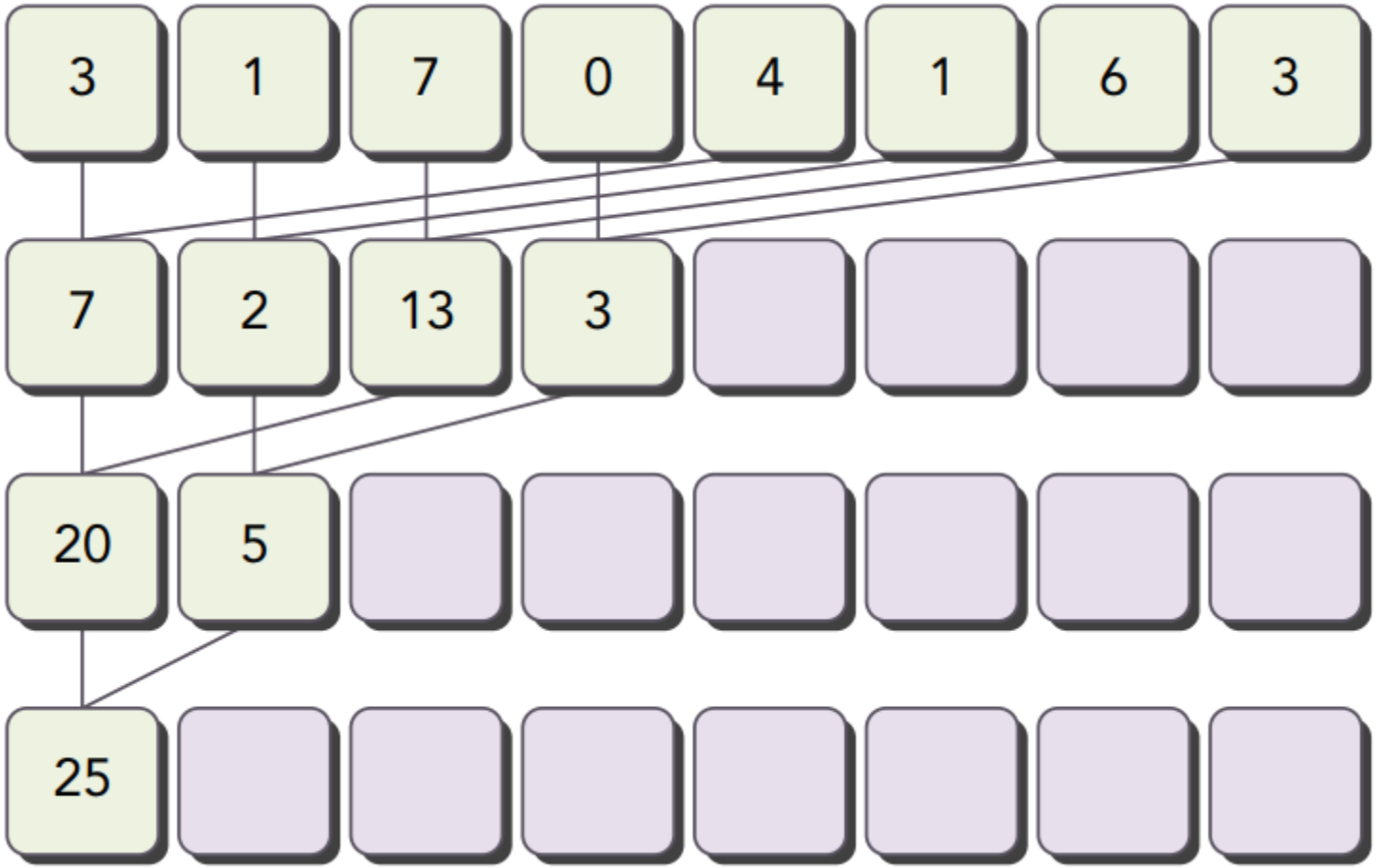
```
int sum = 0;
for (int i = 0; i < N; i++)
   sum += array[i];
```

- Làm thế nào để song song hoá?

Song song hoá



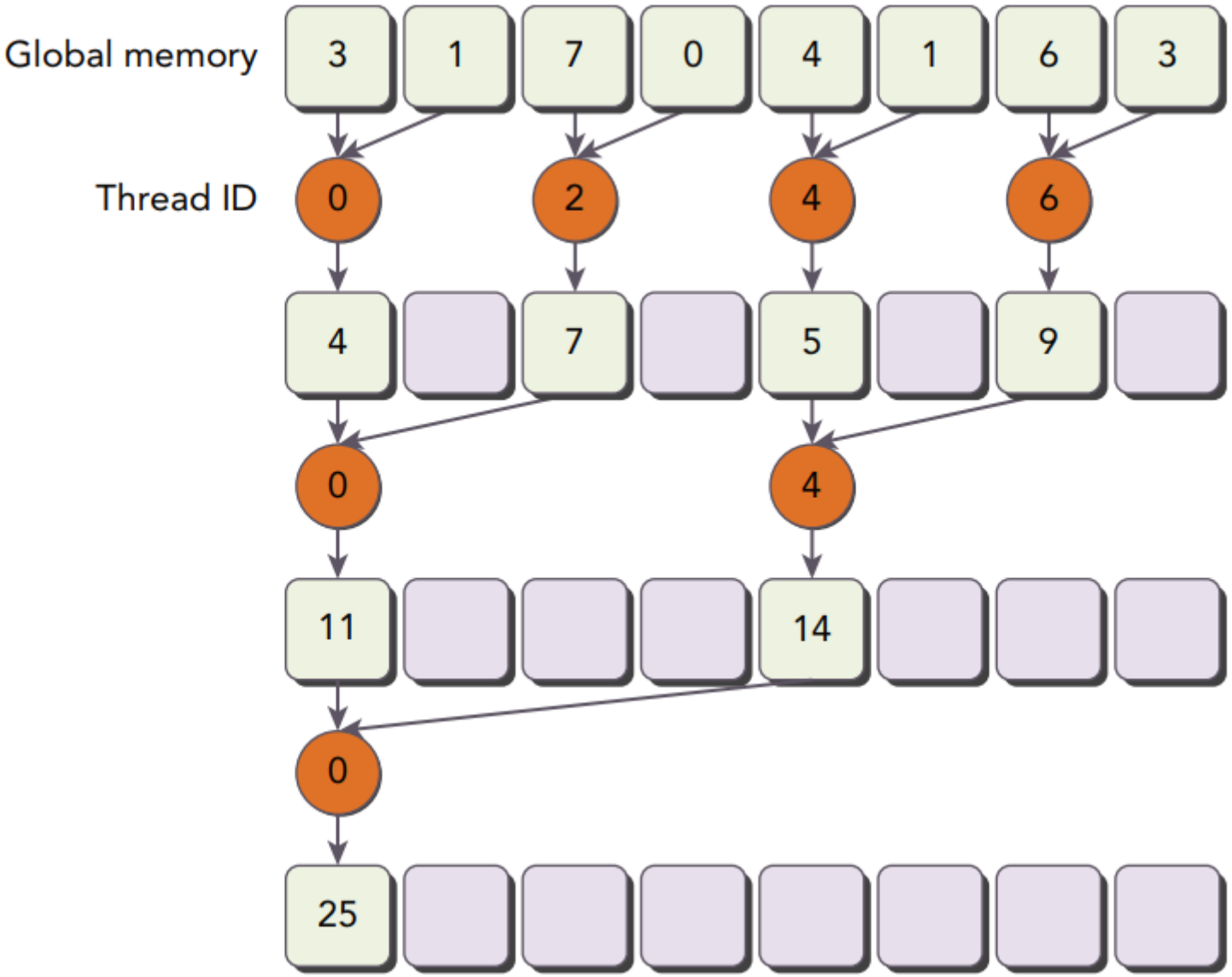
Song song hoá



Song song hoá

```
int recursiveReduce(int *data, int const size) {
    // terminate check
    if (size == 1) return data[0];
    // renew the stride
    int const stride = size / 2;
    // in-place reduction
    for (int i = 0; i < stride; i++) {
        data[i] += data[i + stride];
    }
    // call recursively
    return recursiveReduce(data, stride);
}
```

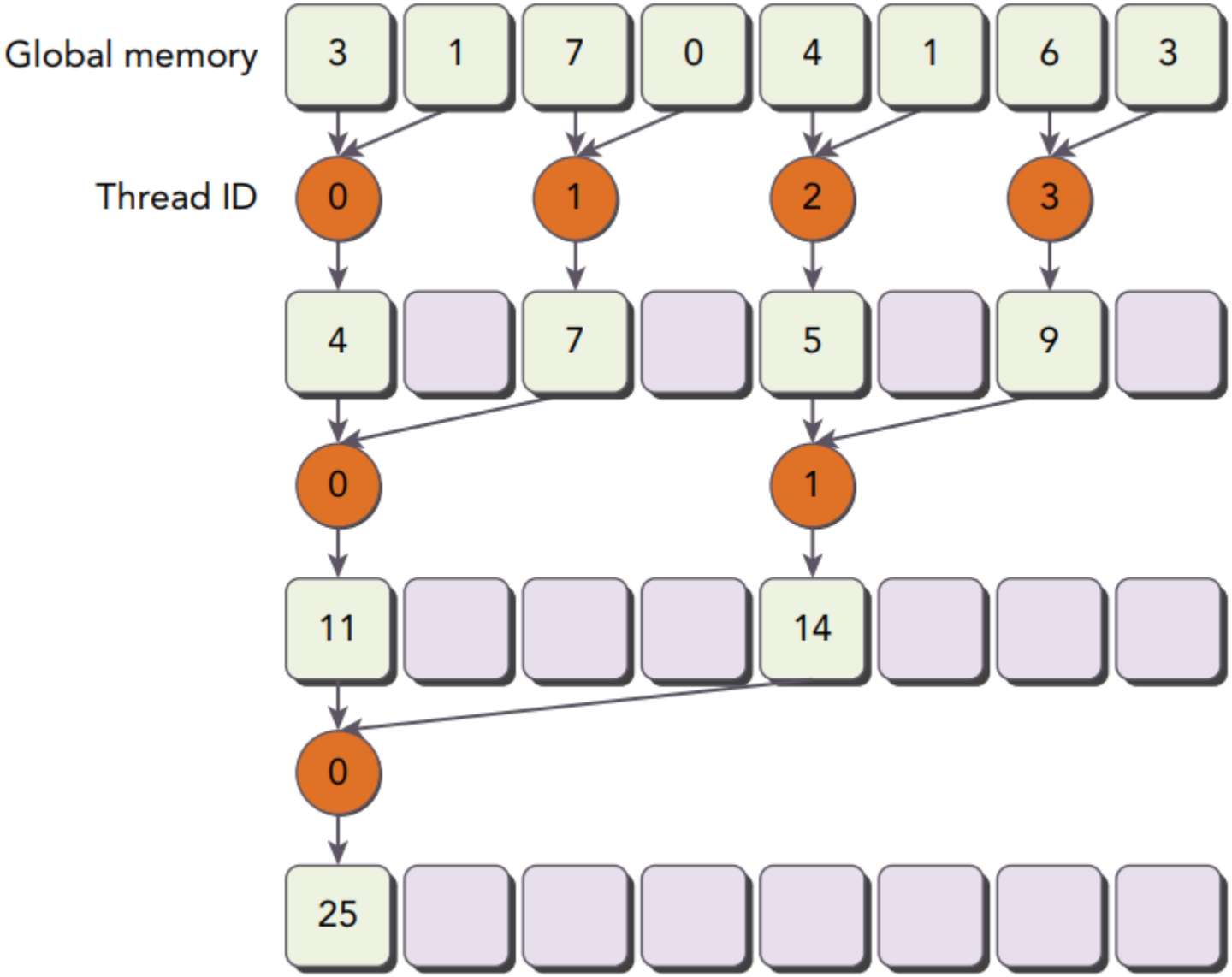
Phiên bản GPU - Divergence



Phiên bản GPU - Divergence

```
__global__ void reduceNeighbored(int *g_idata, int *g_odata, unsigned int n) {
   // set thread ID
   unsigned int tid = threadIdx.x;

   // convert global data pointer to the local pointer of this block
   int *idata = g_idata + blockIdx.x * blockDim.x;

   // boundary check
   if (idx >= n) return;

   // in-place reduction in global memory
   for (int stride = 1; stride < blockDim.x; stride *= 2) {
      if ((tid % (2 * stride)) == 0) {
         idata[tid] += idata[tid + stride];
      }

      // synchronize within block
      __syncthreads();
   }

   // write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

Phiên bản GPU – Không Divergence



Phiên bản GPU - Không Divergence

```
__global__ void reduceNeighboredLess (int *g_idata, int *g_odata, unsigned int n) {
   // set thread ID
   unsigned int tid = threadIdx.x;
   unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

   // convert global data pointer to the local pointer of this block
   int *idata = g_idata + blockIdx.x*blockDim.x;

   // boundary check
   if(idx >= n) return;

   // in-place reduction in global memory
   for (int stride = 1; stride < blockDim.x; stride *= 2) {
        // convert tid into local array index
      int index = 2 * stride * tid;
      if (index < blockDim.x) {
         idata[index] += idata[index + stride];
      }

      // synchronize within threadblock
      __syncthreads();
   }

   // write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

Reducing with Interleaved Pairs

```
__global__ void reduceInterleaved (int *g_idata, int *g_odata, unsigned int n)
{
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x;
    // boundary check
    if(idx >= n) return;
    // in-place reduction in global memory
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1)
    {
        if (tid < stride)
        {
            idata[tid] += idata[tid + stride];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

Reducing with Unrolling

```
__global__ void reduceUnrolling2 (int *g_idata, int *g_odata, unsigned int n)
{
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;
    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x * 2;
    // unrolling 2
    if (idx + blockDim.x < n) g_idata[idx] += g_idata[idx + blockDim.x];
    __syncthreads();
    // in-place reduction in global memory
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1)
    {
        if (tid < stride)
        {
            idata[tid] += idata[tid + stride];
        }
        // synchronize within threadblock
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

Reducing with Unrolled Warps

```
// unrolling warp
if (tid < 32)
{
    volatile int *vmem = idata;
    vmem[tid] += vmem[tid + 32];
    vmem[tid] += vmem[tid + 16];
    vmem[tid] += vmem[tid +  8];
    vmem[tid] += vmem[tid +  4];
    vmem[tid] += vmem[tid +  2];
    vmem[tid] += vmem[tid +  1];
}
```

Reducing with Unrolled Warps

```
__global__ void reduceUnrollWarps8 (int *g_idata, int *g_odata, unsigned int n)
{
    // set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x * blockDim.x * 8 + threadIdx.x;

    // convert global data pointer to the local pointer of this block
    int *idata = g_idata + blockIdx.x * blockDim.x * 8;

    // unrolling 8
    if (idx + 7 * blockDim.x < n)
    {
        int a1 = g_idata[idx];
        int a2 = g_idata[idx + blockDim.x];
        int a3 = g_idata[idx + 2 * blockDim.x];
        int a4 = g_idata[idx + 3 * blockDim.x];
        int b1 = g_idata[idx + 4 * blockDim.x];
        int b2 = g_idata[idx + 5 * blockDim.x];
        int b3 = g_idata[idx + 6 * blockDim.x];
        int b4 = g_idata[idx + 7 * blockDim.x];
        g_idata[idx] = a1 + a2 + a3 + a4 + b1 + b2 + b3 + b4;
    }
```

```
    __syncthreads();
    // in-place reduction in global memory
    for (int stride = blockDim.x / 2; stride > 32; stride >>= 1)
    {
        if (tid < stride)
        {
            idata[tid] += idata[tid + stride];
        }
        // synchronize within threadblock
        __syncthreads();
    }
    // unrolling warp
    if (tid < 32)
    {
        volatile int *vmem = idata;
        vmem[tid] += vmem[tid + 32];
        vmem[tid] += vmem[tid + 16];
        vmem[tid] += vmem[tid +  8];
        vmem[tid] += vmem[tid +  4];
        vmem[tid] += vmem[tid +  2];
        vmem[tid] += vmem[tid +  1];
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = idata[0];
}
```

Reducing with Complete Unrolling

```
// in-place reduction and complete unroll
if (blockDim.x >= 1024 && tid < 512) idata[tid] += idata[tid + 512];
__syncthreads();

if (blockDim.x >= 512 && tid < 256) idata[tid] += idata[tid + 256];
__syncthreads();

if (blockDim.x >= 256 && tid < 128) idata[tid] += idata[tid + 128];
__syncthreads();

if (blockDim.x >= 128 && tid < 64) idata[tid] += idata[tid + 64];
__syncthreads();
```

Reducing with Template Functions

```
template <unsigned int iBlockSize>
__global__ void reduceCompleteUnroll(int *g_idata, int *g_odata,
                                     unsigned int n)
{
    //....

    __syncthreads();
    // in-place reduction and complete unroll
    if (iBlockSize >= 1024 && tid < 512) idata[tid] += idata[tid + 512];
    __syncthreads();

    if (iBlockSize >= 512 && tid < 256)  idata[tid] += idata[tid + 256];
    __syncthreads();

    if (iBlockSize >= 256 && tid < 128)  idata[tid] += idata[tid + 128];
    __syncthreads();

    if (iBlockSize >= 128 && tid < 64)   idata[tid] += idata[tid + 64];
    __syncthreads();
```

Reducing with Template Functions

```
switch (blocksize) {
   case 1024:
      reduceCompleteUnroll<1024><<<grid.x/8, block>>>(d_idata, d_odata, size);
      break;
   case 512:
      reduceCompleteUnroll<512><<<grid.x/8, block>>>(d_idata, d_odata, size);
      break;
   case 256:
      reduceCompleteUnroll<256><<<grid.x/8, block>>>(d_idata, d_odata, size);
      break;
   case 128:
      reduceCompleteUnroll<128><<<grid.x/8, block>>>(d_idata, d_odata, size);
      break;
   case 64:
      reduceCompleteUnroll<64><<<grid.x/8, block>>>(d_idata, d_odata, size);
      break;
}
```

Test result: Tesla M2070

KERNEL	TIME (S)	STEP SPEEDUP	CUMULATIVE SPEEDUP
Neighbored (divergence)	0.011722		
Neighbored (no divergence)	0.009321	1.26	1.26
Interleaved	0.006967	1.34	1.68
Unroll 8 blocks	0.001422	4.90	8.24
Unroll 8 blocks + last warp	0.001355	1.05	8.65
Unroll 8 blocks + loop + last warp	0.001280	1.06	9.16
Templatized kernel	0.001253	1.02	9.35