GPUコンピューティング No.7

thread の実行方式とWarp

東京工業大学　学術国際情報センター

青木　尊之

同時実行可能なthread数

thread の 使用する Register 数: Nr

Shared Memory 量: Ns [byte]

$ nvcc sample12.cu –Xptxas –v –arch sm_20

```
ptxas info    : Compiling entry function '_Z4copyPdS_' for 'sm_20'
ptxas info    : Function properties for _Z4copyPdS
ptxas info    : Used 10 registers, 48 bytes cmem[0]
```

block 当たりの thread 数: Db

Db < 1024（block当たりの最大thread数の制限）

Ns [byte] < 49,152/16,384 [byte]

（block当たりのShared Memoryの制限）

Db × Nr < 32,768 (32-bit register)

（block当たりの総Register数の制限）

thread実行の詳細（Warp）

block 内の thread は Streaming Multiprocessor によって 32 thread 毎に並列実行される

Warp:

block 内の32 threadのかたまり

例：block 内に 256 thread = Warp 8 個

■ プログラム上には現れない

考慮しなくても正しいプログラムを書くことは可能。ただし、実行性能を引き出すためには考慮する必要がある。

Latencyを含んだ命令の実行

1命令(Instruction) は 1 サイクル(Clock) だけでは済まない。

Latency:

- 簡単な命令でも、数サイクル～20サイクル以上
- Global メモリへのアクセスは数100サイクル

CPU: Pre-fetch, Out-of-order

Warp Scheduling

Warp 内の32 thread は同一命令を実行
SIMD (Single Instruction Multiple Data)

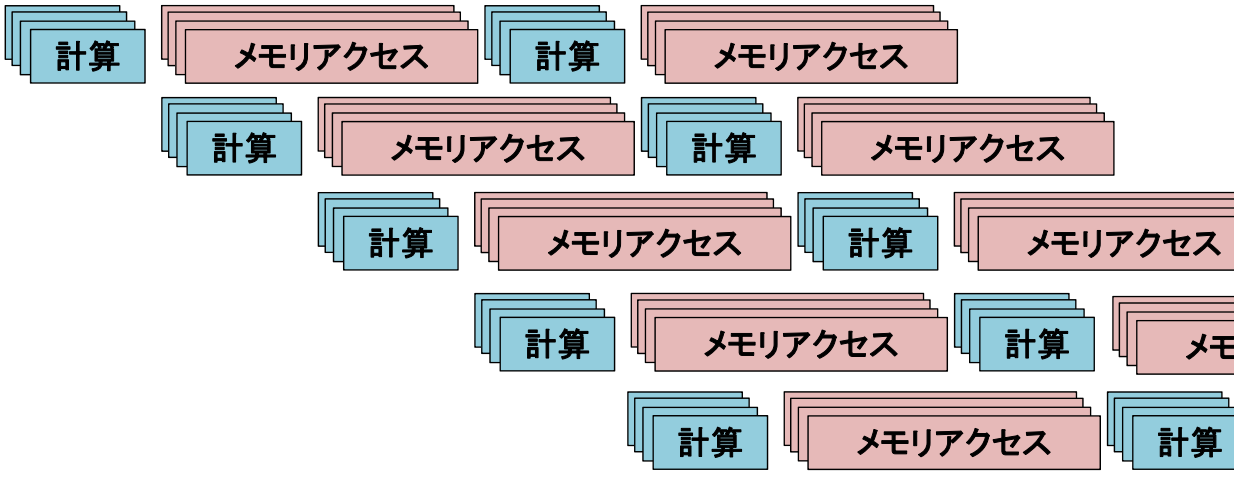
ハードウェア・スケジューラが実行可能な Warp を適宜選択

ある Warp がメモリの読み込み待ちの間に、別の実行可能な Warp を実行することでメモリ・レイテンシーの隠蔽できる。

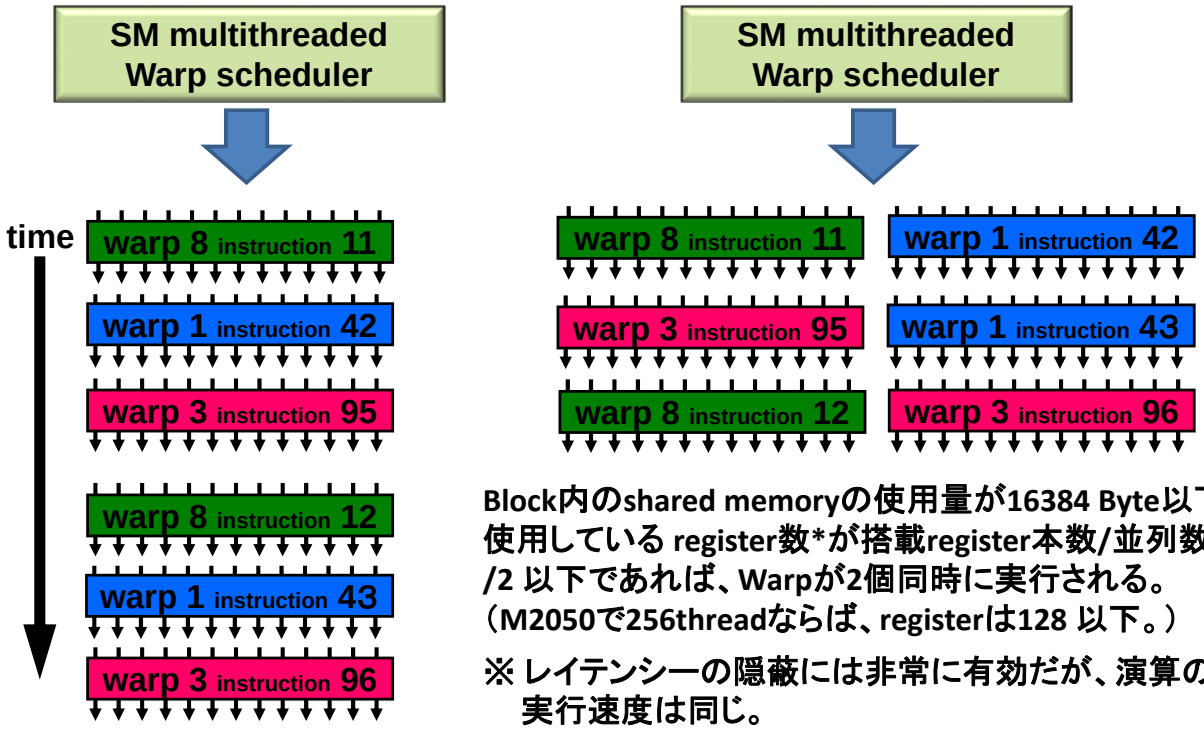
Global Memory へのデータ転送レートが上がる。

パイプライン実行

Latency を隠ぺいするパイプラインが実装
※ ベクトル型スパコンのベクトル・パイプラインとは違う
※ グラフィクス・パイプラインとは少し違う意味



Warp Scheduling



Block内のshared memoryの使用量が16384 Byte以下、使用している register数*が搭載register本数/並列数/2 以下であれば、Warpが2個同時に実行される。
（M2050で256threadならば、registerは128 以下。）

※ レイテンシーの隠蔽には非常に有効だが、演算の実行速度は同じ。

Occupancy

$$\text{Occupancy} = \frac{\text{Active Threads per Multiprocessor}}{\text{Maximum Threads per Multiprocessor}}$$

同じblock 内の Warp でも、異なる block の Warp でも、同じように Latency の隠ぺいができる。

同時に実行される Warp 数（Active Warp）が多いほど、Latency を隠ぺいできる。

Occupancy が大きい方が良いが、Active block の方が重要。

Occupancy Calculator

/opt/cuda/5.0/tools/CUDA_Occupancy_Calculator.xls

CUDA_Occupancy_calculator.xls [読み取り専用] [互換モード]

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	2.0	(Help)
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	8	
Shared Memory Per Block (bytes)	1024	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1536	(Help)
Active Warps per Multiprocessor	48	
Active Thread Blocks per Multiprocessor	6	
Occupancy of each Multiprocessor	100%	
Physical Limits for GPU Compute Capability:	2.0	
Threads per Warp	32	
Warps per Multiprocessor	48	
Threads per Multiprocessor	1536	
Thread Blocks per Multiprocessor	8	
Total # of 32-bit registers per Multiprocessor	32768	
Register allocation unit size	64	
Register allocation granularity	warp	
Shared Memory per Multiprocessor (bytes)	49152	
Shared Memory Allocation unit size	128	
Warp allocation granularity (for register allocation)	0	

CUDA_Occupancy_calculator.xls [読み取り専用] [互換モード]

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1.3	(Help)
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	8	
Shared Memory Per Block (bytes)	1024	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100%	
Physical Limits for GPU Compute Capability:	1.3	
Threads per Warp	32	
Warps per Multiprocessor	32	
Threads per Multiprocessor	1024	
Thread Blocks per Multiprocessor	8	
Total # of 32-bit registers per Multiprocessor	16384	
Register allocation unit size	512	
Register allocation granularity	block	
Shared Memory per Multiprocessor (bytes)	16384	
Shared Memory Allocation unit size	512	
Warp allocation granularity (for register allocation)	2	

同時実行可能なthreadとWarp数

thread の 使用する Register 数: Nr
Shared Memory 量: Ns [byte]

block 当たりの thread 数: Db

Warp per SM = Db/32

Active block
= Min(8,
48/(Db/32),
49152/Ns,
32768/(Db*Nr))

（block当たりの最大48 Warp）
（Shared Memory の制限）
（Register の制限）

Warp の中のthreadの実行

Warp 内の全 thread は同一命令を実行（SIMD モデル）にも関わらず、プログラマから見た場合、各スレッドは別個の命令を実行可能（SPMDモデル）

例えば以下は正しく実行される。

```
if (threadIdx.x == 0) do_something();
else                  do_otherthing();
```

threadIdx.x = 0 ～ 31 の Warp に対して、threadIdx.x = 0 だけが do_something(); を実行し、残りの 31 thread は do_otherting(); を実行することができるか。

threadの中の条件分岐

プログラム上では任意の分岐を記述可能。

ハードウェア上での分岐命令の処理 Warp内全threadが同一パスに分岐する場合 は 全threadが分岐先（のみ）を実行する。

Warp内のthreadが異なるパスに分岐する場合は全スレッドが両方の命令を実行 (diverged branch)し、最後に適合する方だけを採用する。性能低下の原因の一つ。

threadの中の条件分岐

diverge する場合：

```
if (threadIdx.x == 0) do_something();
else                  do_otherthing();
```

diverge しない場合：

```
if (threadIdx.x /32 == 0) do_something();
else                      do_otherthing();
```

異なるワープ間では diverge しない。

ハードウェアでの処理の問題なので、プログラマは意識しなくても正しいプログラムを記述可能。

分岐の低減

- 条件分岐の入れ子は、それぞれに diverge する：

```
if (threadIdx.x == 0) {
    if(j < 0) do_something0();
    else      do_something1();
}
else {
    if(j < 0) do_something0();
    else      do_something1();
}
```

- アルゴリズムによる改善
- ループも条件分岐の一つ

ループアンローリング → # pragma unroll