2013年度
実践的並列コンピューティング
第11回

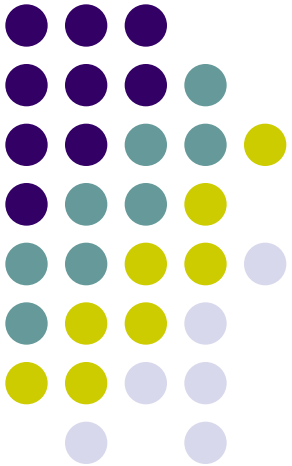
GPUプログラミング (2)

※前回の第10回は休講

遠藤 敏夫

endo@is.titech.ac.jp

2013年6月24日



なぜCUDAではスレッドが二段階か

- ハードウェアの構造に合わせてある

ハードウェア:

1 GPU = 14 SM

1 SM = 32 CUDA core

CUDAのモデル:

1 Grid = 複数thread block

1 thread block = 複数thread

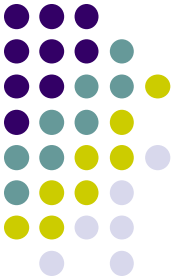
NVIDIA Kepler世代
GPUの構造

1スレッドブロックは、必ず1SM上で動作
(複数スレッドブロックがSMを共有するのはあり)
1スレッドは、必ず1 CUDA coreで動作
(複数スレッドがCUDA coreを共有するのはあり)

スレッド数はどう決めればよい？(1)

- CPUではスレッド数>コア数にしても、効率は上がらないか、むしろ下がる
- グリッドサイズが14以上、かつスレッドブロックサイズが32以上の場合に効率的
  - M2050 GPUでは
    - GPU中のSM数=14
    - SM中のCUDA core数=32 なので
  - ぎりぎりよりも、数倍以上にしたほうが効率的な場合が多い(ベストな点はプログラム依存)
  - 理由は、メモリアクセスのオーバーラップができるから
    - メモリ待ちでプロセッサが待つ代わりに、他のスレッド達を実行できる
    - CPUでもhyperthreadingで同様の効果あるが、せいぜいコアあたり2ハードウェアスレッド

スレッド数はどう決めればよい？(2)



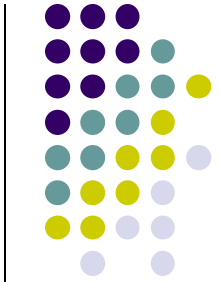
原則的に多いほうが有利

→計算対象の「配列サイズ」を使ってしまうのはあり。ただし、

- スレッドブロック数×スレッド数にうまく割り当てる必要 (例: inc_parサンプル)
- CUDAの定める限界値あり
  - グリッドサイズ x:65535, y:65535, z: 65535
  - スレッドブロックサイズ x: 1024, y: 1024, z: 64, 計1024まで

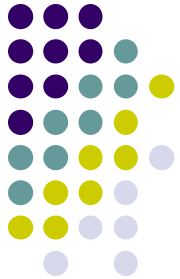
M2050などの場合

- 多すぎて不利になるケースはやっぱりある
  - 共有メモリ(後述)の利用が非効率になるなど
- 多次元配列の場合の割り振りの考慮
  - 例：X, Y方向は並列化して、Z方向は各スレッドに行わせよう



「DIVERGENT分岐」の削減による効率化

GPUでのスレッドの実行のされ方



- スレッドブロック内のブロック達は、(プログラマからは見えないが)32スレッドごとの塊(warp)単位で動作している
- Warpの中の32スレッドは、「常に」足並みをそろえて動いている

→ If文などの分岐があるとどうなる？

  - Warp内のスレッド達の「意見」がそろうか、そろわないかで、動作が異なる

GPU上のif文の実行のされ方

(a) スレッド達の意見がそろう場合

- 全員、xxx>100だとする

```
        :
        :
if (xxx > 100) {
        :
        :
        :
        :
} else {
        :
        :
        :
        :
}
        :
        :
```

Else部分は、実行せずに飛ばす

(b) スレッド達の意見が違う場合

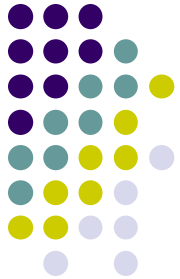
- あるスレッドではyyy>100だが、別スレッドは違う場合

```
        :
        :
if (yyy > 100) {
        :
        :
        :
        :
} else {
        :
        :
        :
        :
}
        :
        :
```

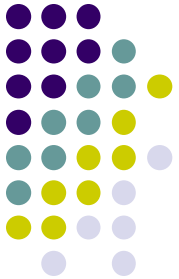
一部スレッドを「眠らせて」
ThenもElseも両方実行

これを**divergent**
分岐と呼ぶ

Divergent分岐はなぜ非効率？



- CPUの常識では、if文はthen部分とelse部分の片方しか実行しないので、片方だけの実行時間がかかる
- Divergent分岐があると、then部分とelse部分の両方の時間がかかってしまう

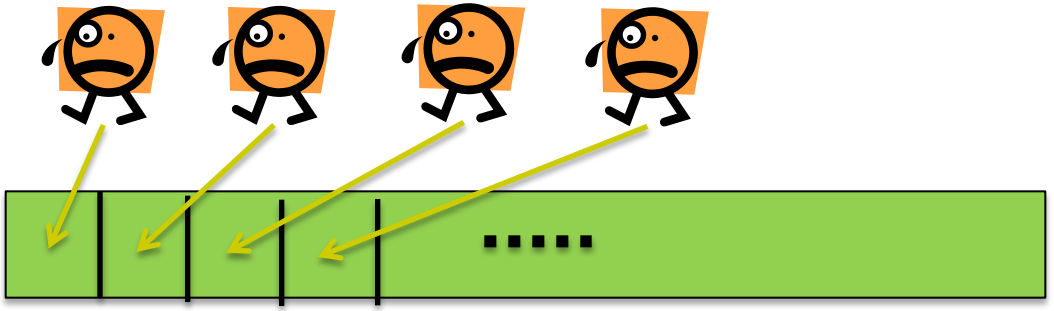


「コアレスド・アクセス」によるメモリアクセス効率化

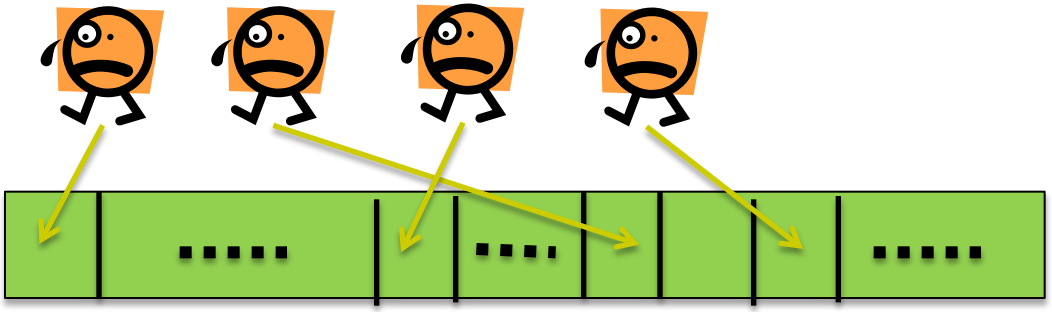
グローバルメモリのアクセスの効率化：コアレスド・アクセス

- メモリの性質上、「近い(たとえば番号が隣りの)スレッドが近いアドレスを同時にアクセスする」のが効率的
  - コアレスド・アクセス (coalesced access)と呼ぶ

隣り合ったスレッドが、配列の隣の要素をアクセス
→ コアレスドアクセスになっており、**高速**



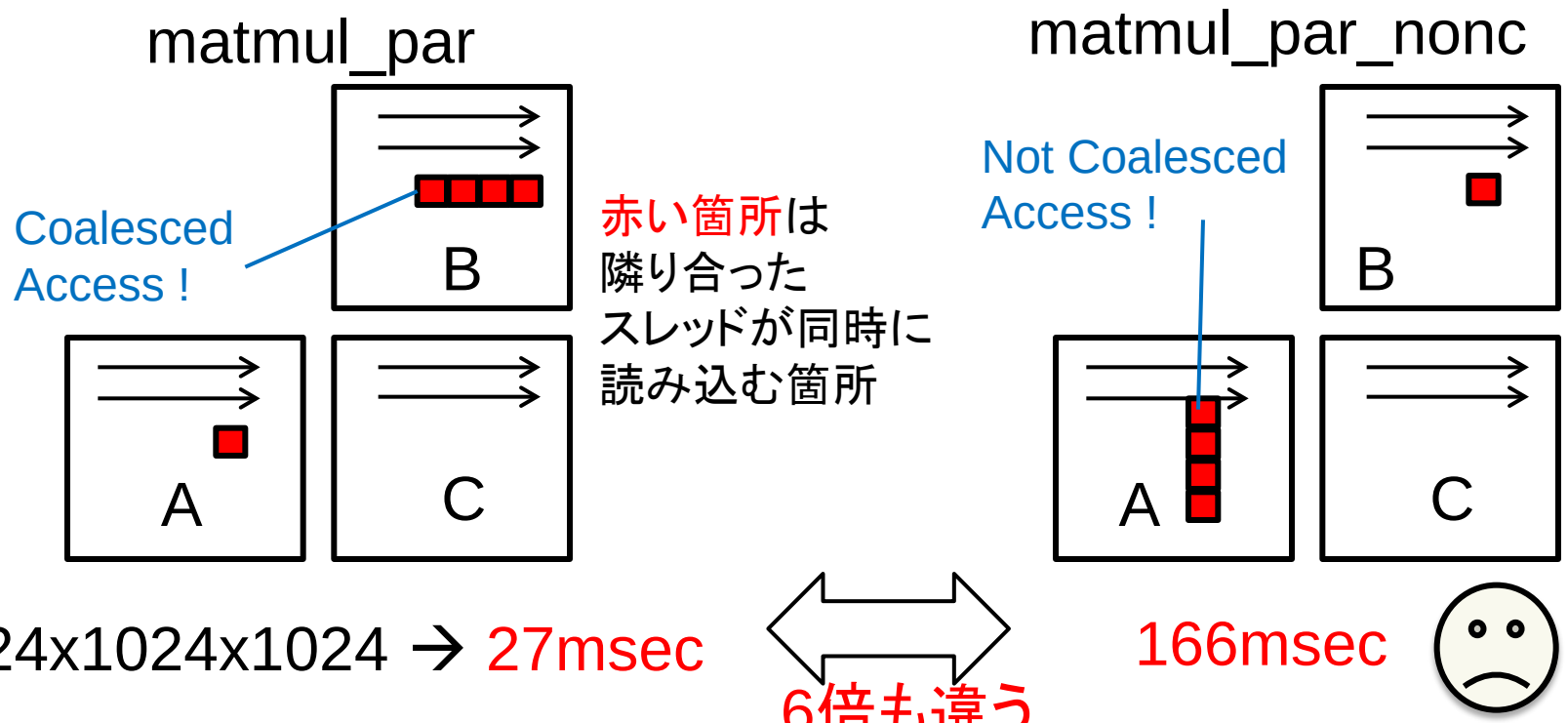
各スレッドがばらばらの要素をアクセス
→ コアレスドアクセスではなく、**低速**



基礎編のinc_parプログラムは、コアレスドアクセスになっていた

コアレスドアクセス有無の影響

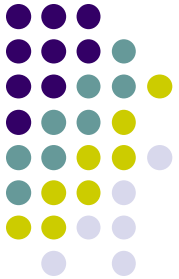
- blockDimが二次元/三次元指定の場合、x方向に並んだスレッドたちのアクセス場所が重要
  - matmul_parサンプルでは、もともとコアレスドアクセスが効いていた



1024x1024x1024 → 27msec

6倍も違う

166msec

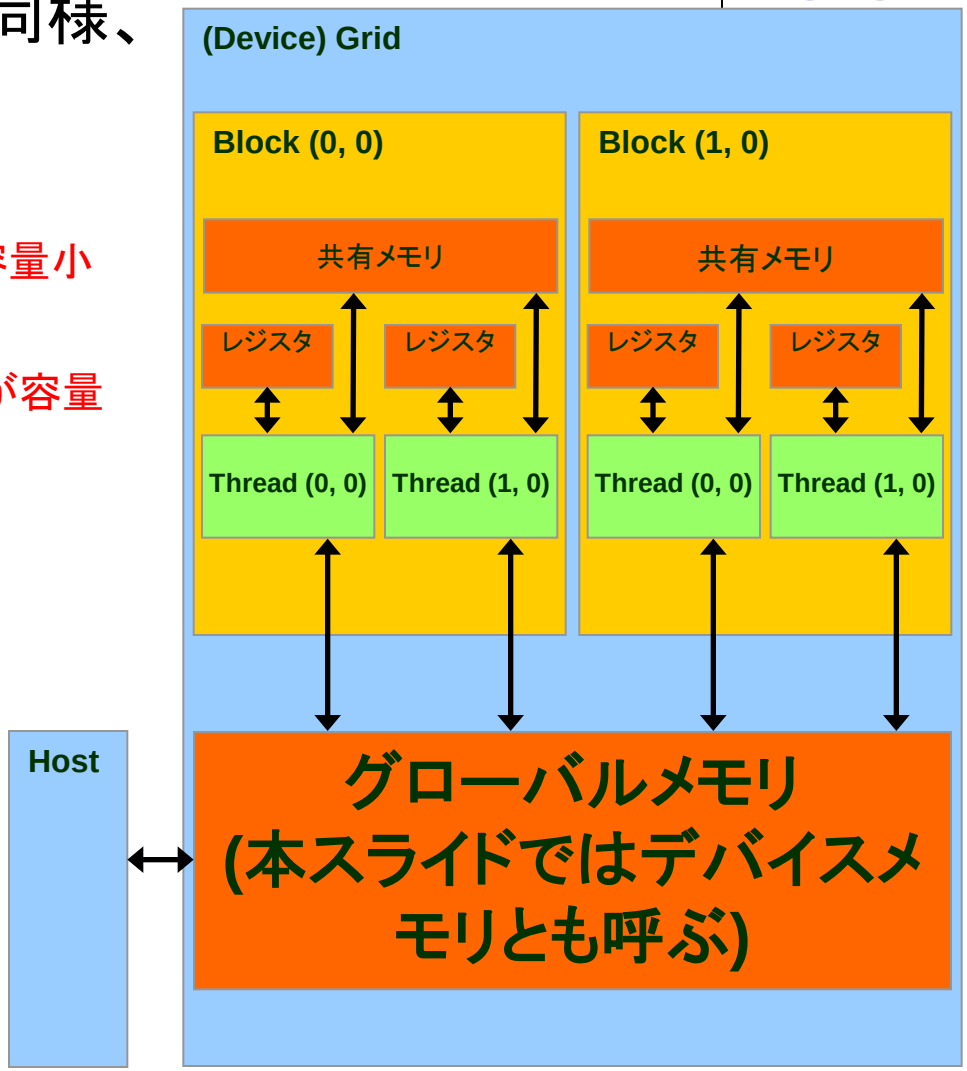


「共有メモリ」の有効活用

CUDAメモリモデル

スレッドが階層化されているのと同様、
メモリも階層化されている

- スレッド固有
  - レジスタ → 局所変数を格納。高速だが容量小
- ブロック内共有
  - 共有メモリ → 本スライドで登場。高速だが容量小
  - （L1キャッシュ）
- グリッド内（全スレッド）共有
  - グローバルメモリ→ __global__変数やcudaMallocで利用。容量大きいが低速
  - （L2キャッシュ）

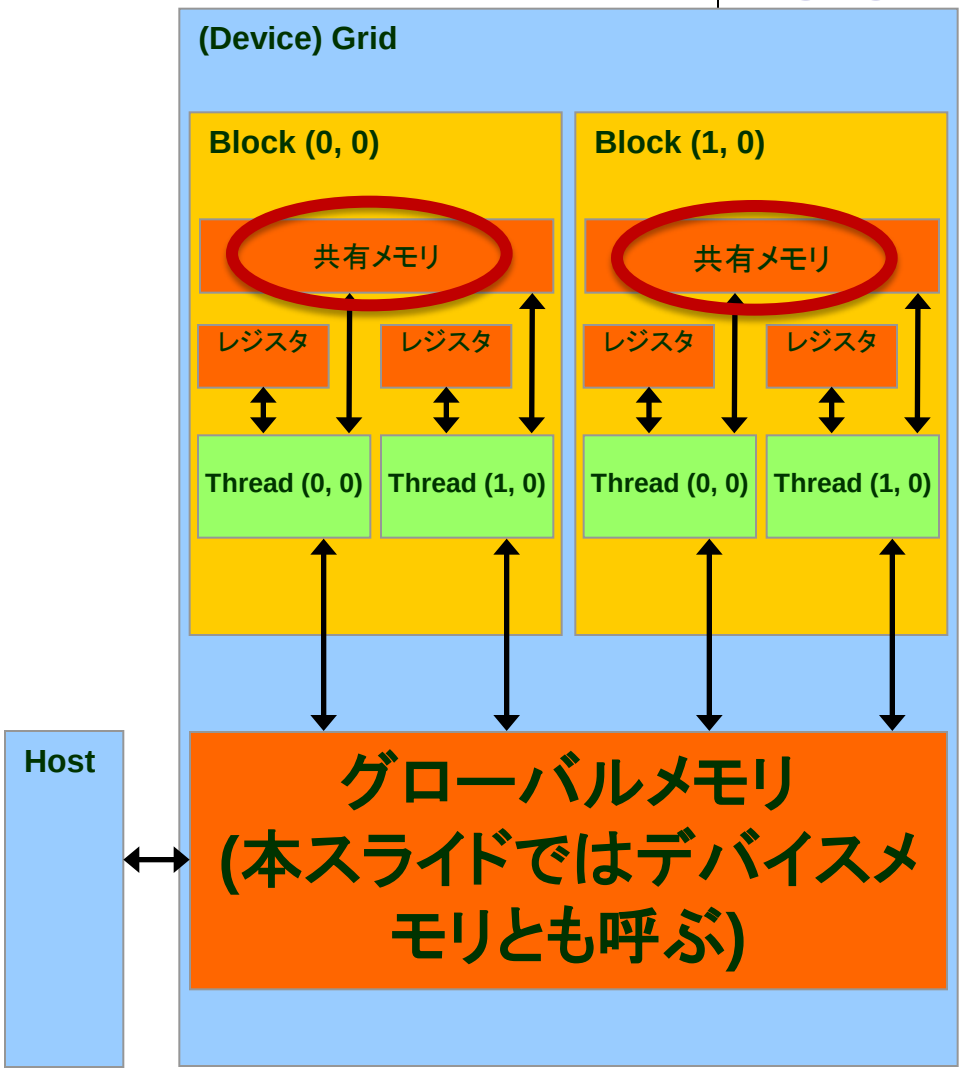


それぞれ速度と容量にトレードオフ有
（高速＆小容量 vs. 低速＆大容量）
→ メモリアクセスの局所性が重要

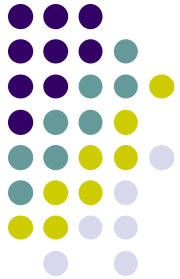
Source: Kirk and Hwu, ECE 498AL, UIUC

共有メモリの利用による プログラム効率化

- 基礎編のようにプログラムを書くと、通常はレジスタとグローバルメモリのみを利用
- 共有メモリとは:
  - ブロック内のスレッド達で共有されるメモリ領域
  - 高速
  - 容量は小さい(ブロックあたり16KB以下)
- __shared__ int a[16]; のように書くと、共有メモリ上に置かれる



共有メモリをどういう時に使うと効果的？



- 一般的には、グローバルメモリの同じ場所を、ブロック内の別スレッドが使いまわす場合に効率的
  - たとえばmatmul_parプログラムでは、A, Bの要素は複数スレッドによって読み込まれる

- 一度グローバルメモリから共有メモリに明示的にコピーしてから、使いまわすと有利
  - カーネル関数の書き換えが必要
  - ただし、GPUにはキャッシュもあるため、共有メモリで本当に高速化するか?は場合による

共有メモリを使った行列積プログラム: matmul_shared

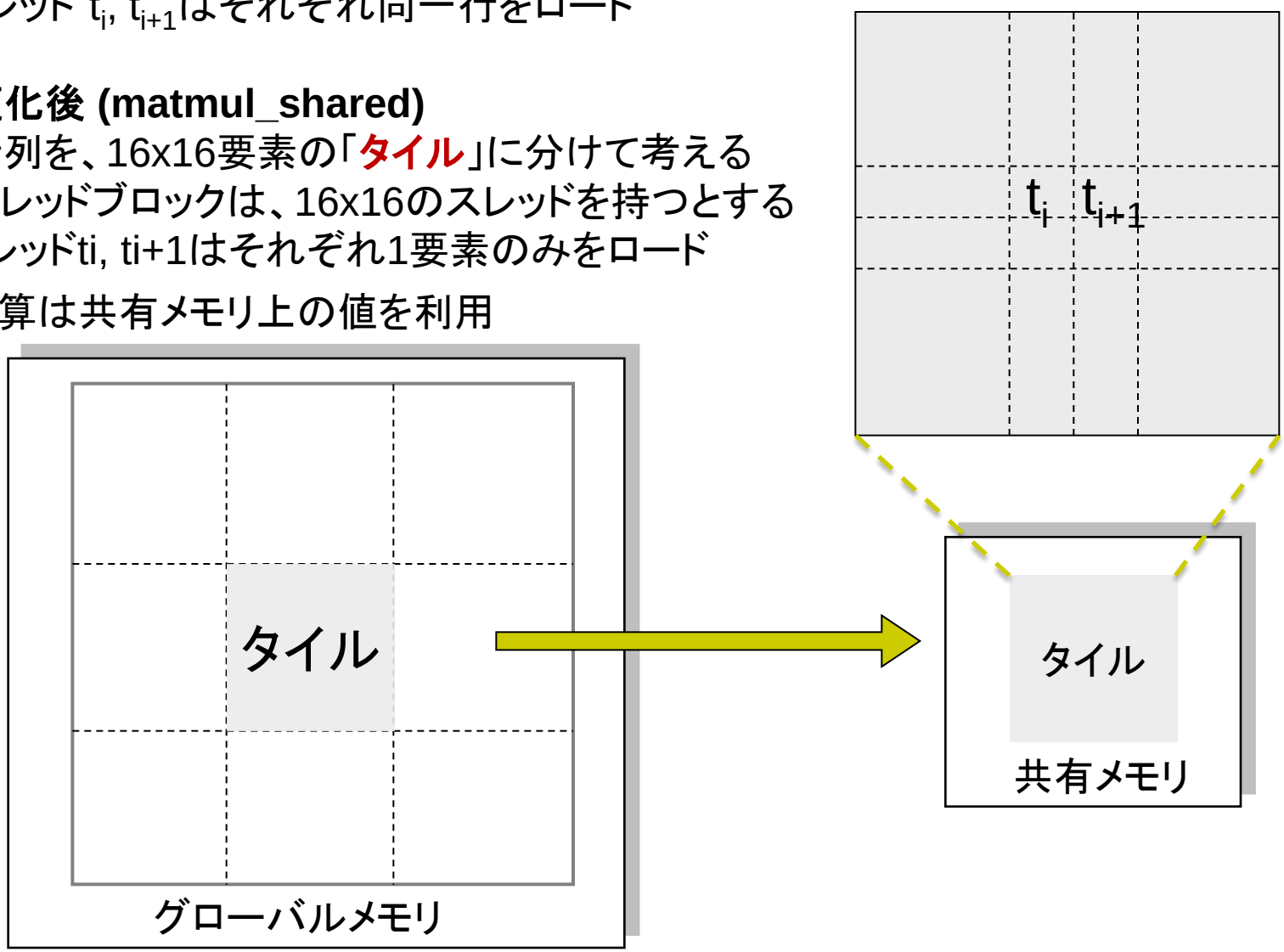
最適化前 (matmul_par)

- スレッド t_i, t_{i+1}はそれぞれ同一行をロード

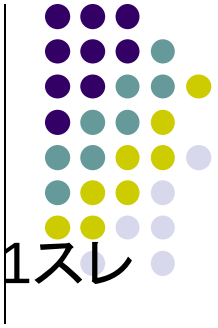
最適化後 (matmul_shared)

各行列を、16x16要素の「**タイル**」に分けて考える
各スレッドブロックは、16x16のスレッドを持つとする

- スレッドti, ti+1はそれぞれ1要素のみをロード
- 計算は共有メモリ上の値を利用



matmul_sharedの流れ

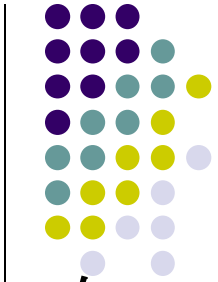


このプログラムでは、1スレッドブロックがCの1タイル分を計算。1スレッドがCの1要素を計算。

1. 行列A、B共に、その一部のタイルをグローバルメモリから共有メモリにコピー
2. __syncthreads() により同期
3. 共有メモリを用いてタイルとタイルのかけ算。
4. 次のタイルのために、1へ戻る
5. 各スレッドは、自分が計算した$C_{i,j}$をグローバルメモリに書き込む

- 2.の__syncthreads() とは？
  - スレッドブロック内の全スレッドの「足並みをそろえる(同期)」
  - この命令を呼ぶまでは、共有メモリに書いた値が必ずしも他のスレッドへ反映されない

共有メモリを使った高速化の結果

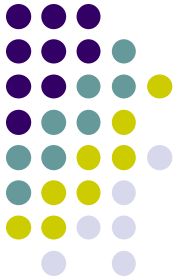


サイズ1024x1024の行列A, B, Cがあるとき、C=A×Bを計算する

- matmul_cpu.c
  - CPUで計算

  → **約8.3秒**（gcc –O2でコンパイルした場合）
- matmul_seq.cu
  - GPUの1スレッドで計算　→ **約200秒**
- matmul_par.cu
  - GPUの複数スレッドで計算　→ **約0.027秒**
- matmul_shared.cu
  - GPUの複数スレッドで計算し、共有メモリも利用

  → **約0.012秒(!)**



CUDAプログラムの時間計測に関する注意

時間計測に関する注意

- プログラム中の各部分にかかる時間を測るために、clock(), gettimeofday()関数を使うことはよくある
- CUDAプログラムで以下を測るとき注意が必要

  (a) cudaMemcpy(ホスト→デバイス方向)

  (b) カーネル関数呼び出し
- 本当の時間よりもはるかに短く見えてしまう
  - 実際には、上記(a)(b)を実行すると、「仕事を依頼しただけ」の状態で、実行が帰ってきてしまう(非同期呼び出し)

  → 時刻測定前にcudaDeviceSynchronize()を行っておくこと
  - cudaDeviceSynchronize()の意味:「現在までにGPUに依頼した仕事が、全部終了するまで待つ」

各部分ごとの時間計測を行うには

```
clock_t t1, t2, t3, t4

cudaDeviceSynchronize(); t1 = clock();
cudaMemcpy(･･･, cudaMemcpyHostToDevice);

cudaDeviceSynchronize(); t2 = clock();
my_kernel<<<･･･, ･･･>>>(･･･);

cudaDeviceSynchronize(); t3 = clock();
cudaMemcpy(･･･, cudaMemcpyDeviceToHost);

cudaDeviceSynchronize(); t4 = clock();
```

- t1とt2の差分が、cudaMemcpy (ホストからデバイス)の時間
- t2とt3の差分が、カーネル関数実行にかかった時間
- t3とt4の差分が、cudaMemcpy (デバイスからホスト)の時間

About Account

- TSUBAME2のアカウントができたら，連絡してください．授業用のTSUBAMEグループへ登録します．

Subject: TSUBAME2 ppcomp account

To: endo@is.titech.ac.jp

- 専攻・研究室
- 学年
- 氏名
- アカウント名

次回/Next Lecture

- 7/1(月)
  - CUDAによるGPUプログラミング (3)
  - GPU編の課題について

- スケジュールについてはOCW pageも参照
  - http://www.el.gsic.titech.ac.jp/~endo/

  → 2013年度前期情報(OCW) → 講義ノート