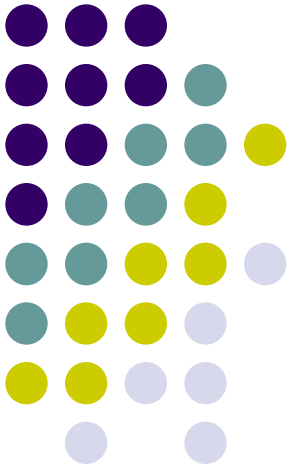
2014年度
実践的並列コンピューティング
第15回

GPUプログラミング (3)



遠藤 敏夫

endo@is.titech.ac.jp

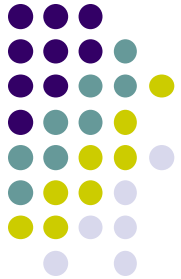
CUDAによるGPUプログラミング

- main関数はCPU上で始まる。GPUで動作させたい(高速化したい)箇所をGPUカーネル関数として記述
  - __global__, __device__つき関数
- GPUカーネル関数はGPU上のメモリ(デバイスメモリ)だけをアクセスできる
  - デバイスメモリの操作にはcudaMalloc, cudaMemcpy
- __global__関数呼び出しの<<<…>>>構文で、スレッドブロック数とスレッド数を指定
  - ハードウェアのコア数の<<<14, 192>>>より多い方が高速な傾向

カーネル関数内でできること・できないこと

- if, for, whileなどの制御構文はok
- GPU側メモリのアクセスはok、CPU側メモリのアクセスは不可
  - inc_seqサンプルで、arrayDと間違ってarrayHをカーネル関数に渡してしまうとバグ!! (何が起こるか分からない)
- ファイルアクセスなどは不可
  - printfは例外的にokなので、デバグに役立つ
- 関数呼び出しは、「__device__つき関数」に対してならok

- 上図の矢印の方向にのみ呼び出しできる
  - GPU内からCPU関数は呼べない
- __device__つき関数は、返り値を返せるので便利

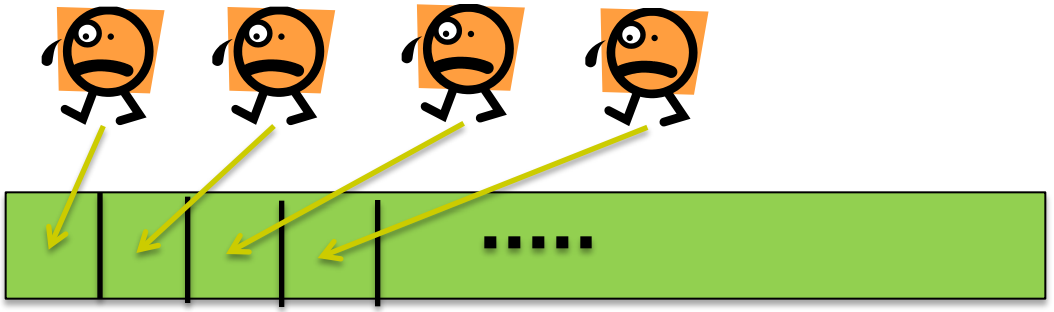


「コアレスド・アクセス」によるメモリアクセス効率化

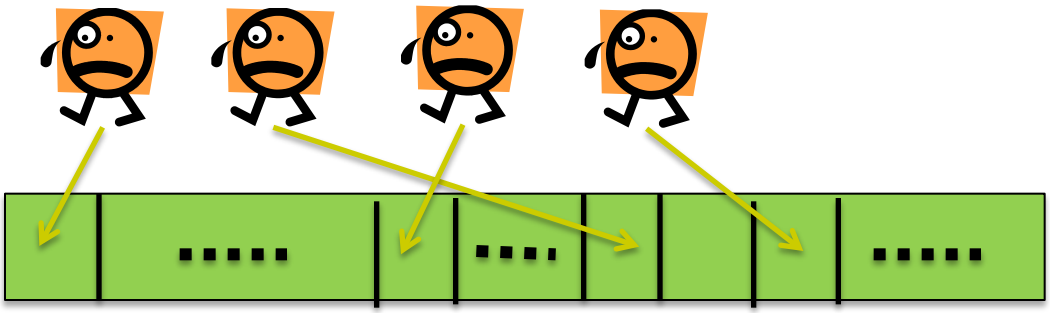
グローバルメモリのアクセスの効率化：コアレスド・アクセス

- 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**



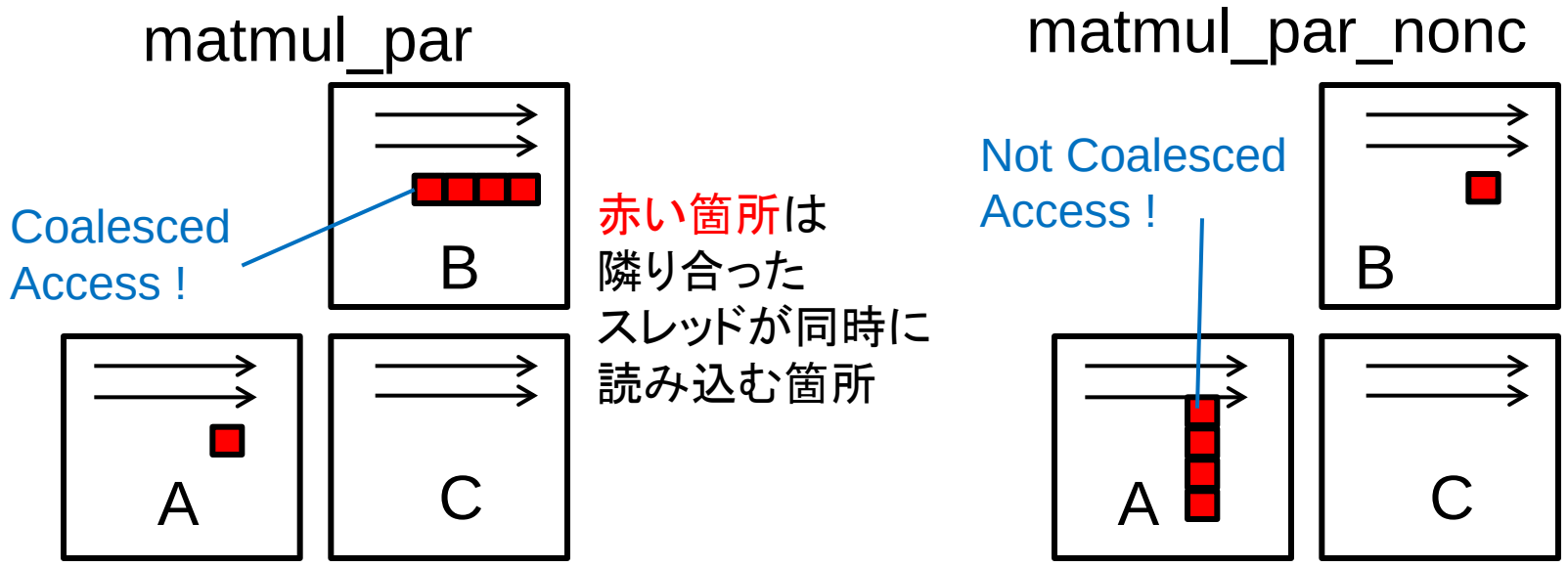
各スレッドがばらばらの要素をアクセス
→ コアレスドアクセスではなく、**低速**



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

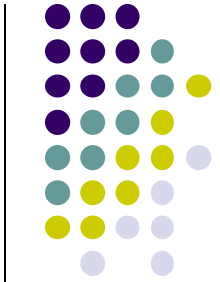
- blockDimが二次元/三次元指定の場合、x方向に並んだスレッドたちのアクセス場所が重要
  - matmul_parサンプルでは、もともとコアレスドアクセスが効いていた
  - このサンプルではデータ並びはrow-major



1024x1024x1024 → 22msec

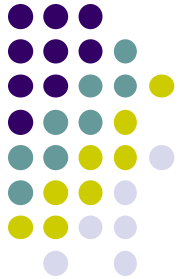
5倍も性能違う

115msec



「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

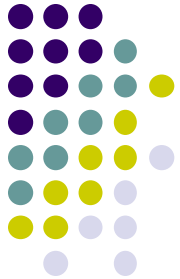
- あるスレッドではyyy>100だが、別スレッドは違う場合

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

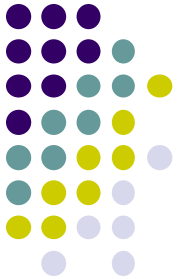
一部スレッドを「眠らせて」
ThenもElseも両方実行

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

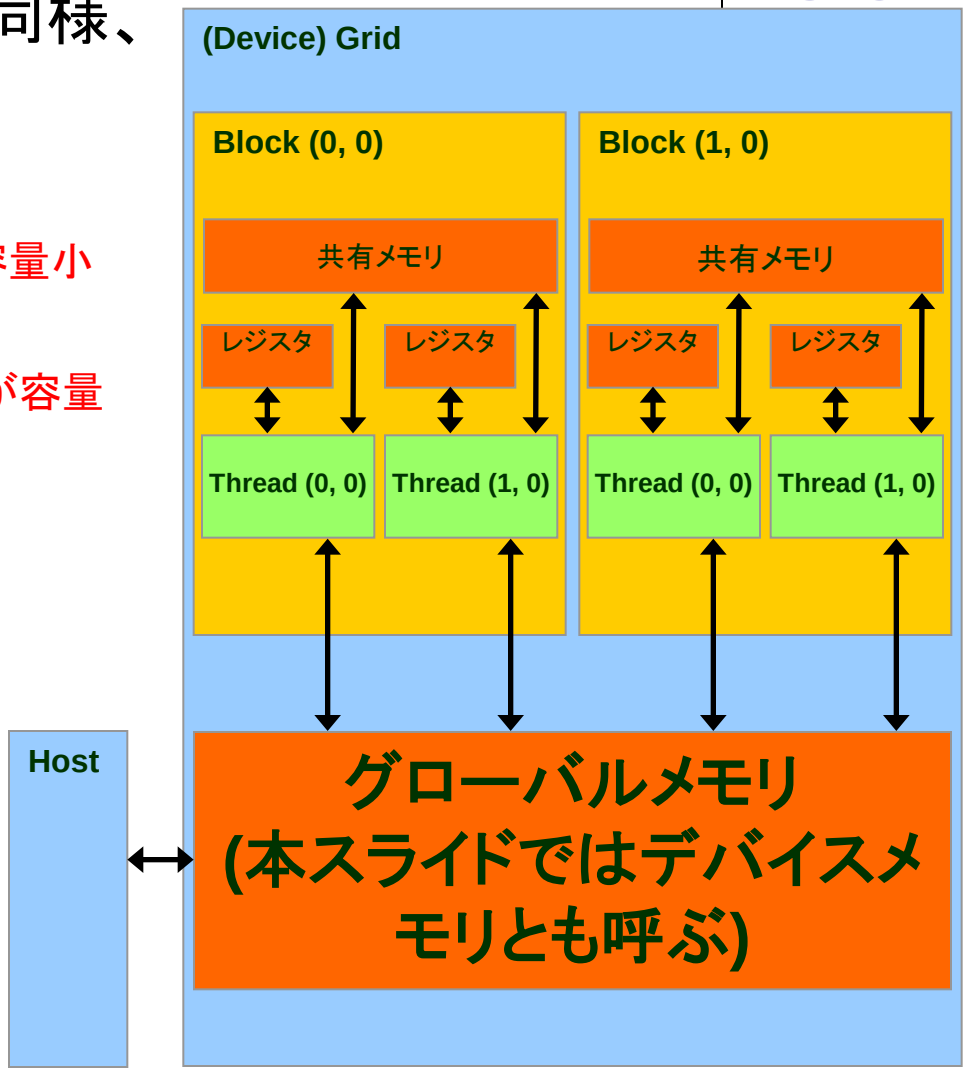


「共有メモリ」の有効活用

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）

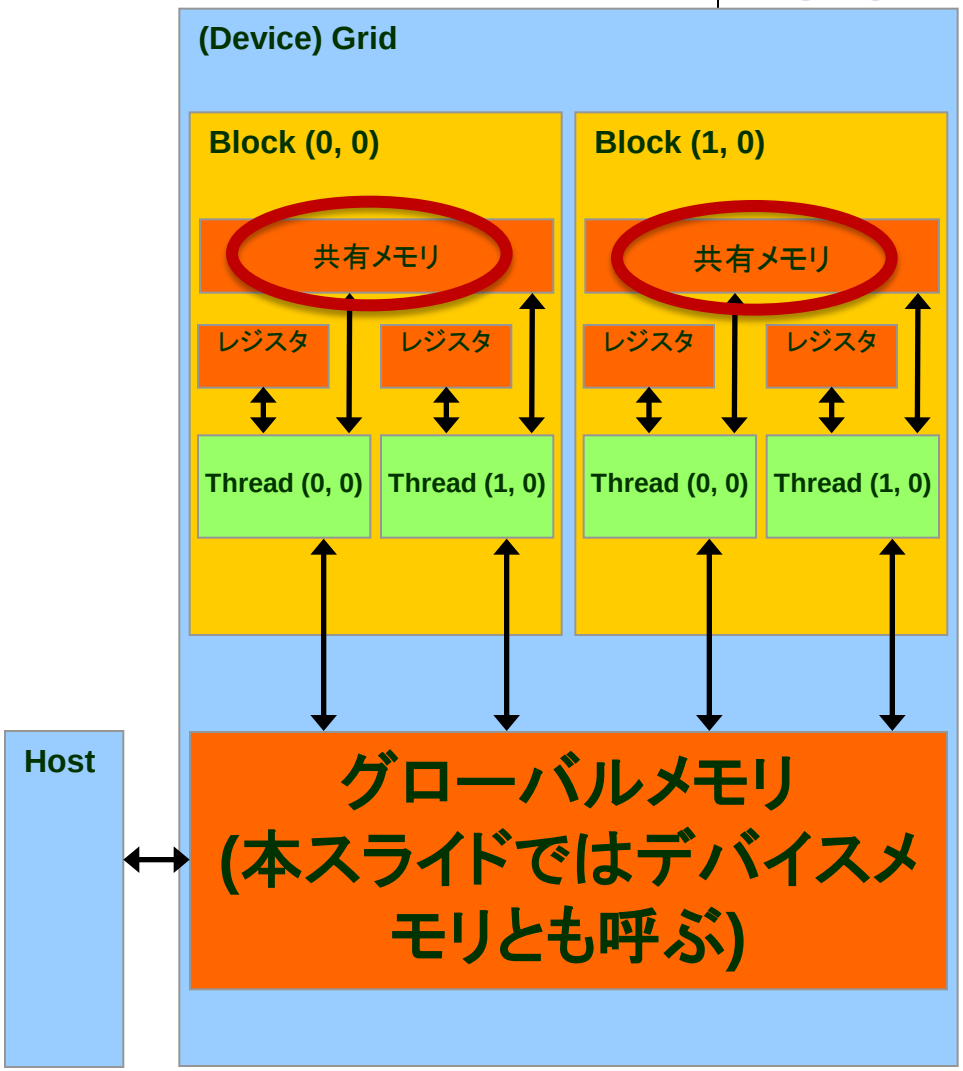


それぞれ速度と容量にトレードオフ有
（高速＆小容量 vs. 低速＆大容量）
→ メモリアクセスの局所性が重要

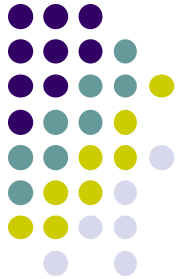
Source: Kirk and Hwu, ECE 498AL, UIUC

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

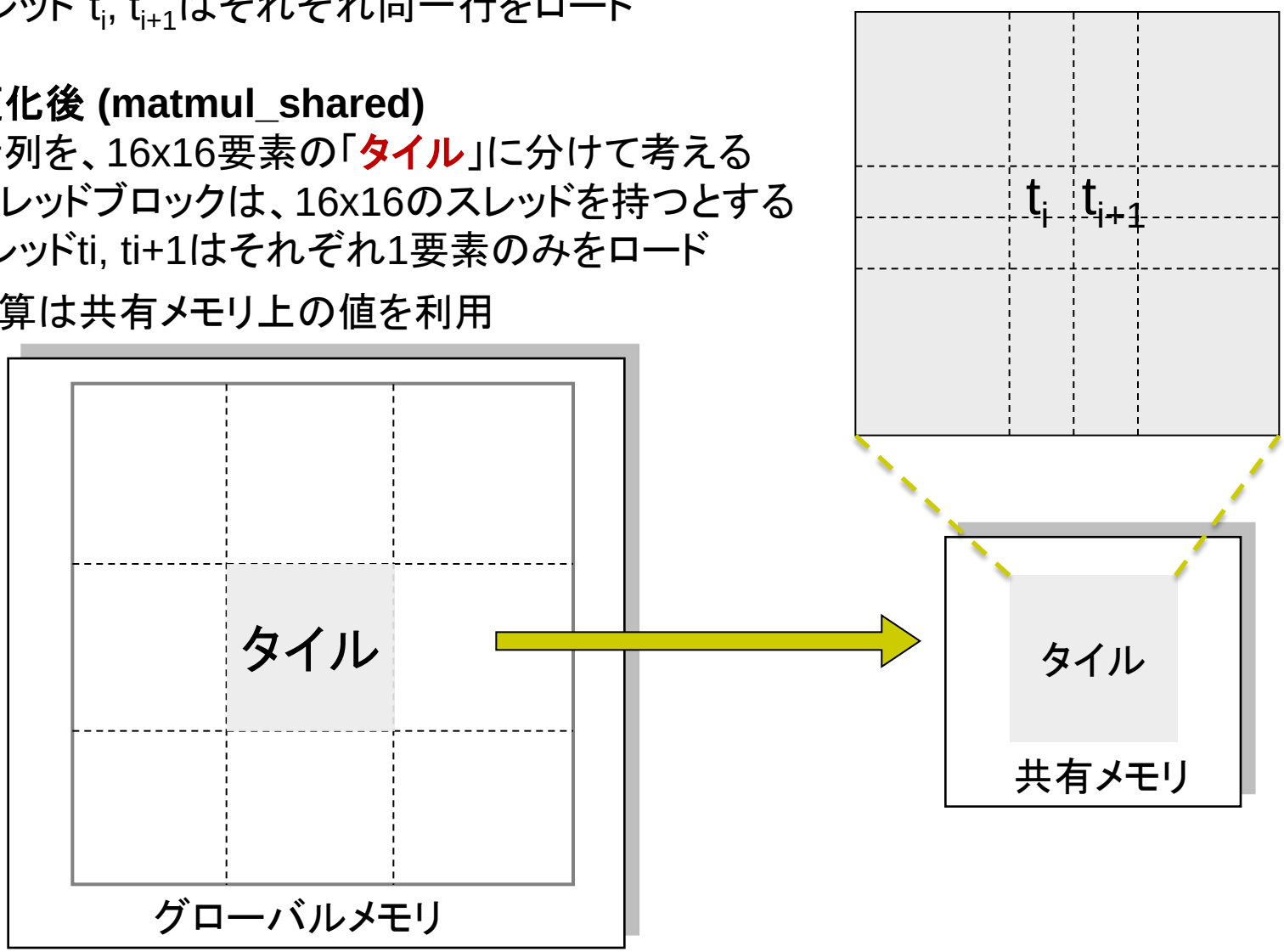
最適化前 (matmul_par)

- スレッド t_i, t_{i+1}はそれぞれ同一行をロード

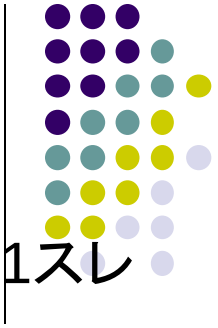
最適化後 (matmul_shared)

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



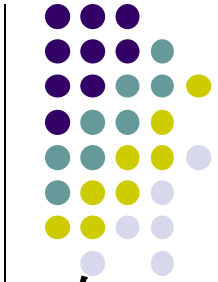
このプログラムでは、1スレッドブロックがCの1タイル分を計算。1スレッドがCの1要素を計算。

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

CUDAにはブロックをまたいだ全スレッドのバリア同期がない
→ 一度GPUカーネル関数を抜ければその効果

共有メモリを使った高速化の結果

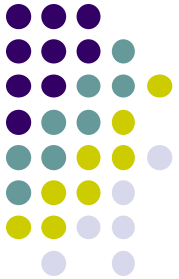


サイズ1024x1024の行列A, B, Cがあるとき、C=A×Bを計算する

- 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();
```

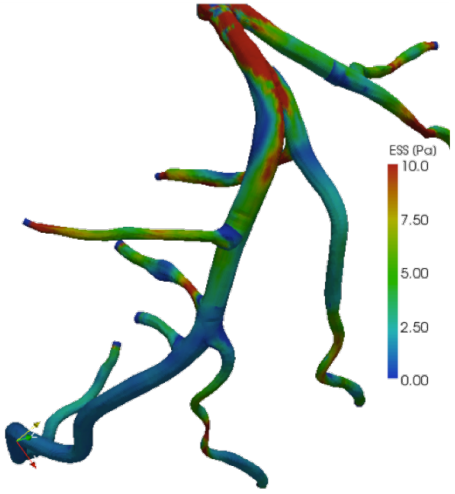
- t1とt2の差分が、cudaMemcpy (ホストからデバイス)の時間
- t2とt3の差分が、カーネル関数実行にかかった時間
- t3とt4の差分が、cudaMemcpy (デバイスからホスト)の時間

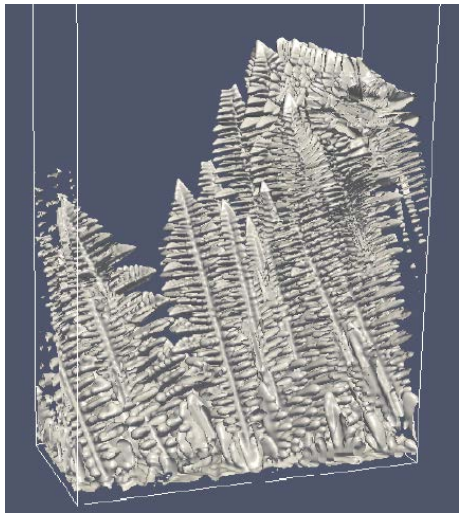
マルチGPUの利用

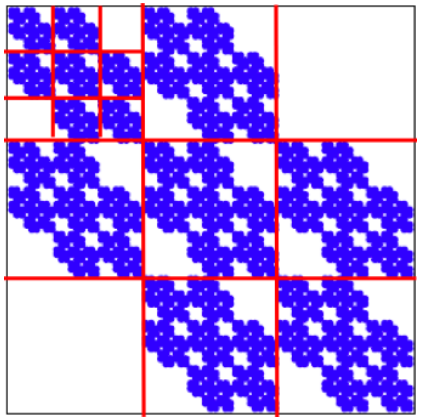
- GPU内の多数CUDA coreを用いてプログラムの高速化可能だが、限界がある

⇒ 多数GPUを用いて限界突破を狙う

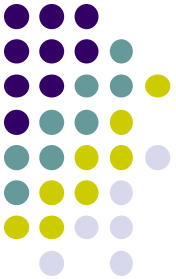
  - TSUBAME2には1ノードあたり3GPU、システム全体で4200GPU







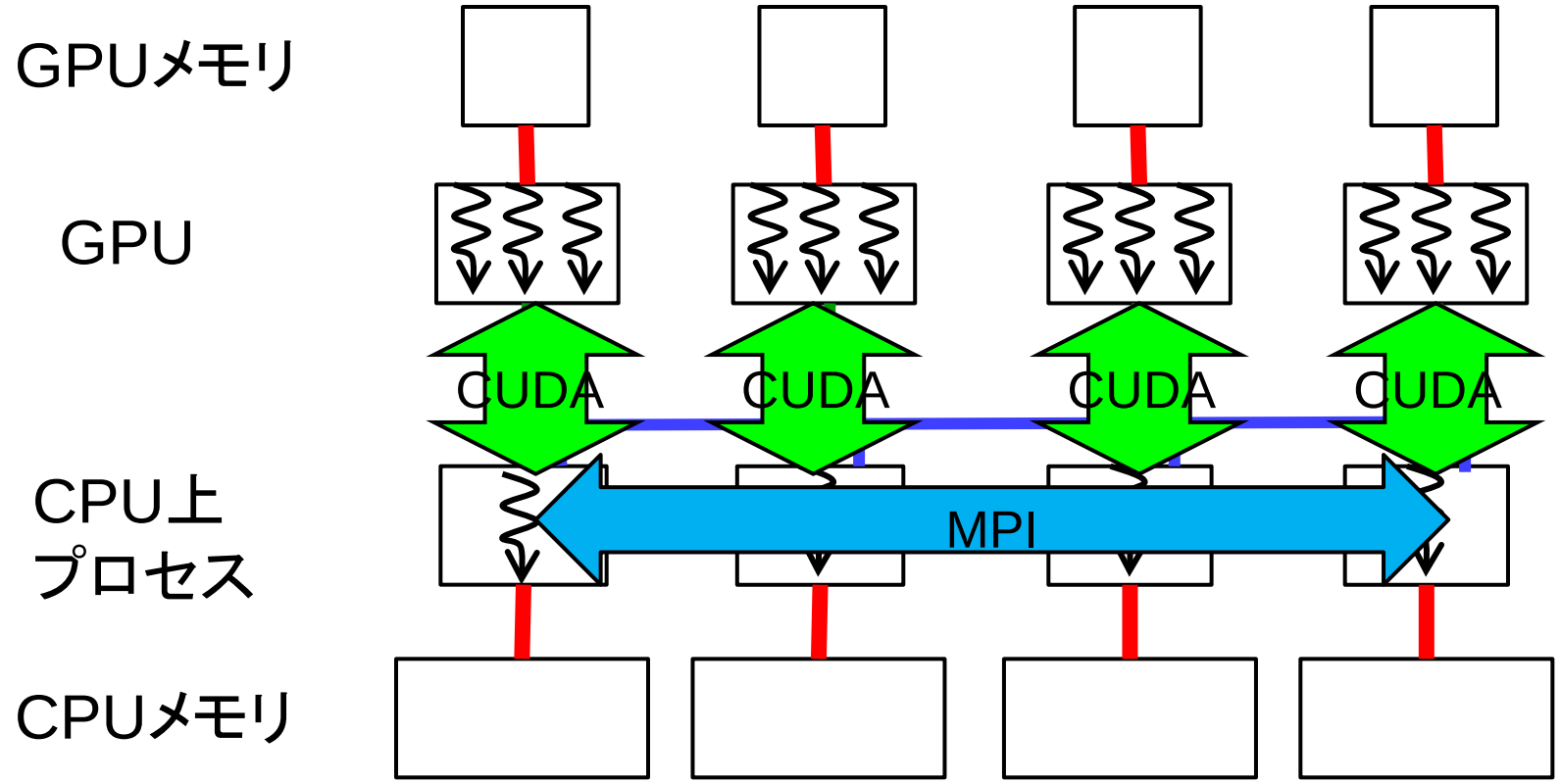
マルチGPUの利用



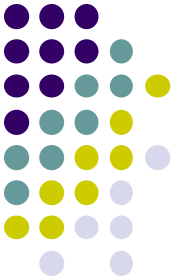
いくつかの基本方針:

- MPI+CUDA: 1プロセスが1GPUを担当　←これを少しだけ解説
- OpenMP+CUDA: 1スレッドが1GPUを担当
  - 欠点：1ノード3GPUまで
- 最近のCUDAでは、複数プロセスが1GPU利用したり、1プロセスが複数GPU利用できる(今回は略)。

MPI+CUDAの考え方



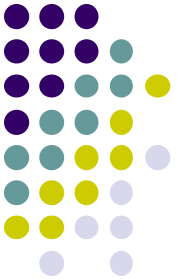
MPI+CUDAの考え方



- CPU-GPU間の通信はcuda(cudaMemcpyなど)で、CPU-CPU間の通信はMPIで
  - GPU-GPU間の直接通信は原則ない

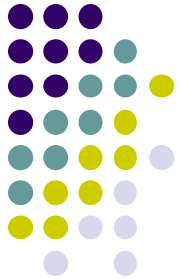
    ⇒ ただし、GPU Directで一部可能な場合も。
    MVAPICH 1.8以降で対応開始

MPI+CUDAの注意



- 1ノードに複数GPUが搭載されている場合、デバイス番号の指定の必要
  - 各プロセスは最初にcudaSetDevice(デバイス番号); を呼ぶ
  - TSUBAME2は3つ。デバイス番号は0~2
  - 1ノードに3MPIプロセスずつ起動し、cudaSetDevice(rank % 3); など

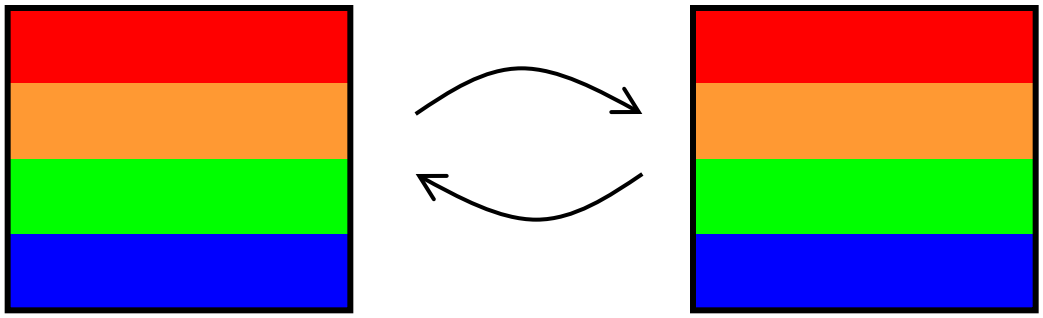
MPI+CUDAの注意(2)



- .cuファイルをnvccでコンパイルした後、MPIライブラリ、CUDAライブラリをリンクする必要
  - ~endo-t-ac/ppcomp/14/mpicudatest/Makefile を参照
  - (このプログラムは速くない例ですが)
- OpenMPIでは、mpirunのオプションに以下が必要
  - -mca mpi_leave_pinned 0 または

    -mca btl_openib_flags 1
  - MPIとCUDAの両方がDMAを用いることによる問題を避ける

Diffusion on MPI+CUDAの方針



- 格子を複数プロセスに空間分割するのは基礎編と同じ
  - プロセス数が多いなら二次元・三次元分割したい
- 毎ステップで境界領域を隣接プロセス間で交換(MPI通信)するのも同じ
  - しかし、計算中の格子データはGPUメモリ上にある。どうするか

MPI+CUDA版Diffusionの流れの想定

MPI_Init

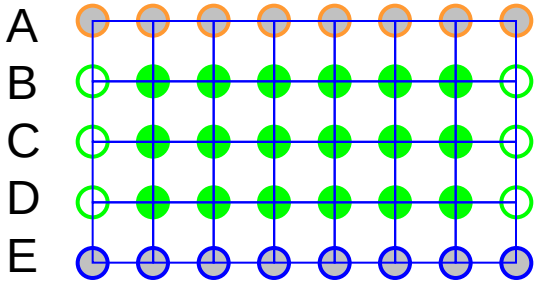
初期の「部分」二次元格子データをCPUからGPUへ

for (jt = 0; jt < nt; jt++) //時間ループ

- 行B, DをGPUからCPUへコピー
- B, DをMPIで送信, A, Eを受信
- 行A, EをCPUからGPUへコピー
- 担当の格子点をGPUで計算
- 二つのバッファを交換

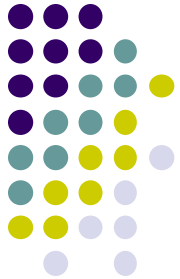
結果の二次元格子データをGPUからCPUへ

MPI_Finalize



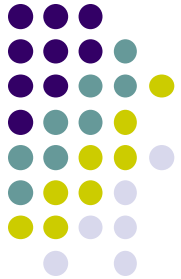
デッドロックに注意。
MPIの回参照

その他、取り上げられなかった話題



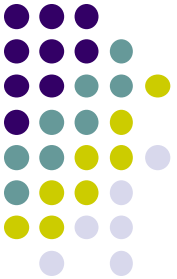
- CUDA Stream
  - 複数stream利用により、GPU計算、CPU→GPU通信、GPU→CPU通信をオーバラップ可能に
- CUDA Profiler
  - GPUカーネルの性能解析
- より詳細なアーキテクチャによる影響
  - Shared memory のbank conflict
  - スレッドブロック数とレジスタ数・shared memory容量の関係
- CUDA 5.5やKepler世代GPU特有の機能
  - Unified virtual memory, HyperQ
- OpenACC言語: CUDAよりも気軽なGPUプログラミング
  - http://tsubame.gsic.titech.ac.jp → 各種利用の手引き → OpenACC利用の手引き

本授業のレポートについて



- 各パートで課題を出す。2つ以上のパートのレポート提出を必須とする
  - OpenMPパート
    - ノード内のCPUコアを使う並列プログラミング
  - MPIパート
    - 複数ノードを使う並列プログラミング
  - GPU(CUDA)パート
    - 1GPU内の数百コアを使う並列プログラミング

GPUパート課題説明 (1)

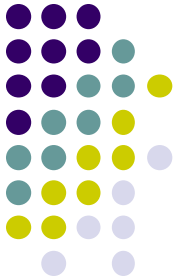


以下のG1, G2, G3の、いずれかについてレポートを提出してください.

[G1] 行列積プログラムの性能を、行列サイズを変化させながら性能評価してください。CPU(OpenMP)版とも比較してください。

- データ転送コストを速度計算に入れる場合・入れない場合それぞれについて測定
  - 転送コストが相対的に大きくなるのはどういう場合か。計算量オーダー、転送量オーダーにも触れて議論すること
- GPU版とCPU版の性能比についても調べること。差が大きいとき、小さいときはどういう場合か
- プログラムを改良してもok

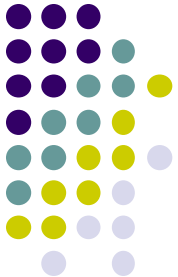
GPUパート課題説明 (2)



[G2] diffusionサンプルプログラムをGPUを用いて並列化し、性能評価してください。

- CPU1コア版は
  ~endo-t-ac/ppcomp/14/diffusion/
- 参考プログラム: …/advection-cuda/
- 改良してもok。たとえば
  - Divergent分岐の影響の削減
  - Shared memoryの利用による高速化
  - マルチGPUの利用
  - ほか

GPUパート課題説明 (3)



[G3] 自由課題：任意のプログラムを，GPU を用いて並列化し、性能評価してください

- たとえば，過去のSuperConの本選問題

  http://www.gsic.titech.ac.jp/supercon/

  たんぱく質類似度(2003)，N体問題(2001)・・・

  入力データは自分で作る必要あり
- たとえば，自分が研究している問題

課題の注意

- いずれの課題の場合も，レポートに以下を含むこと
  - 計算・データの割り当て手法の説明
  - TSUBAME2などで実行したときの性能
    - スレッド数、スレッドブロック数を様々に変化させたときの変化に触れているとなお良い
    - 問題サイズを様々に変化させたとき(可能な問題なら)
  - 高性能化のための工夫が含まれているとなお良い
    - 「XXXのためにXXXをしてみたが高速にならなかった」のような失敗でも可
  - プログラムについては，zipなどで圧縮して添付
    - 困難な場合，TSUBAME2の自分のホームディレクトリに置き，置き場所を連絡

課題の提出について

- 今日のMPIパート〆切にも注意
- GPUパート提出期限
  - 8/4 (月) 23:50
  - 8/5 からTSUBAME2はメンテで止まる予定
- OCW-i ウェブページから下記ファイルを提出のこと
- レポート形式
  - 本文：PDF, Word, テキストファイルのいずれか
  - プログラム：zip形式に圧縮するのがのぞましい

- OCW-iからの提出が困難な場合、メールでもok
  - 送り先： ppcomp@el.gsic.titech.ac.jp
  - メール題名： ppcomp report

次回/補講

- 8/4(月) 補講 10:45～12:15
  - TSUBAME見学会、出欠は成績に影響しません
  - 集合はいつもどおり西8号館W832で、そこからTSUBAMEのある建物へ移動します

ただし、**IT特別教育研究コース関連の方**には、**アンケートをお願い**したいので、出席のご協力お願いします

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

→ 2014年度前期情報(OCW) → 講義ノート