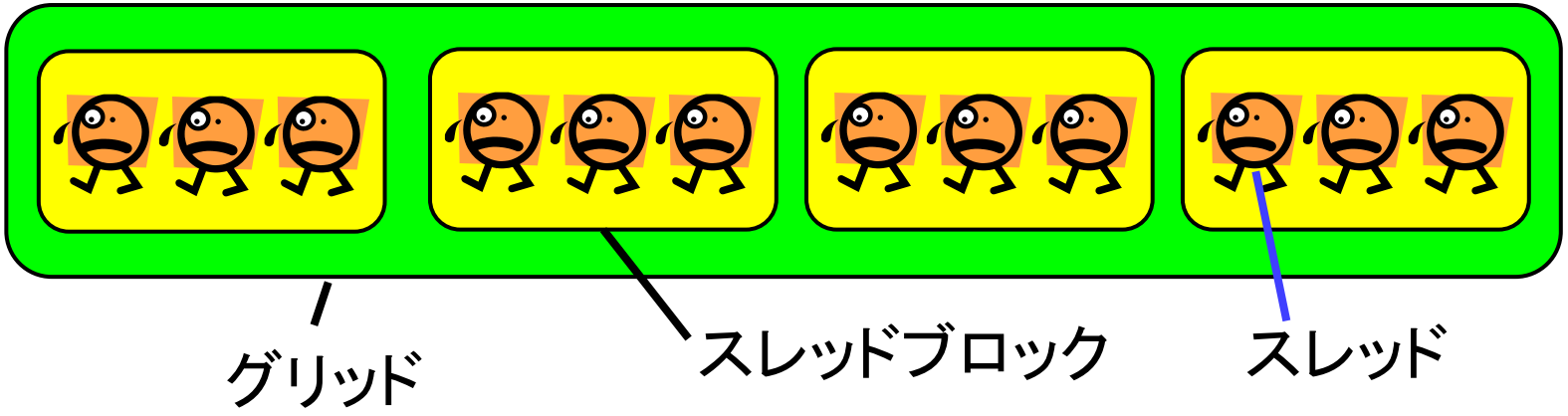
2016年度
実践的並列コンピューティング

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

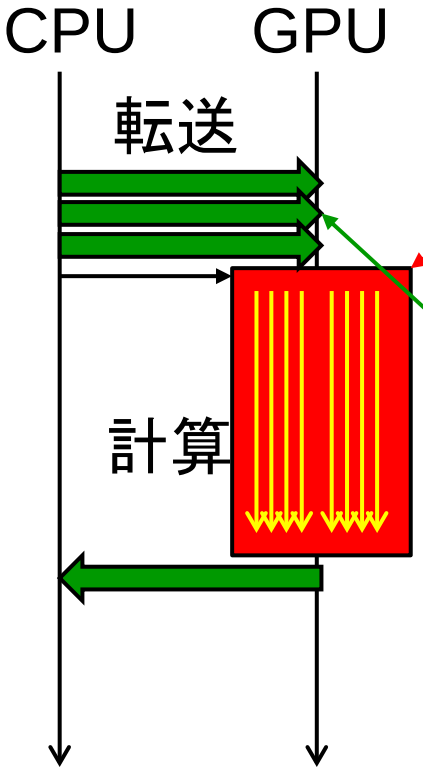
- main関数はCPU上で始まる。GPUで動作させたい(高速化したい)箇所をGPUカーネル関数として記述
  - __global__, __device__つき関数
- GPUカーネル関数はGPU上のメモリ(デバイスメモリ)だけをアクセスできる
  - デバイスメモリの操作にはcudaMalloc, cudaMemcpy
- __global__関数呼び出しの<<<…>>>構文で、スレッドブロック数とスレッド数を指定
  - ハードウェアのコア数の<<<14, 192>>>より多い方が高速な傾向



GPUプログラミングの高速化について

プログラムの並列化ができても、高速化のためにまだまだ気をつける点

- 計算量・転送量を減らせればベストだが、それ以外にも



GPU上の計算時間短縮

- 適切なスレッド数・ブロック数 (前回)
- コアレスドアクセスを増やす
- Divergent分岐を減らす
- (CUDA用語の)共有メモリを利用

転送時間の隠ぺい

- cudaStreamの活用

マルチGPUの活用

～GPU内部の挙動の理解～
もう一つの実行単位:warp

thread < warp < thread block < grid

- Warpは、32個の連続したthreadから成り立つ
  - IDが二次元以上のときは、”xが連続したもの”
- Warp内の32threadは必ず「足並みをそろえて実行」。同時に同じ命令を実行する
  - 一方、違うblock間や、block内の違うwarp間の動作順序は不定

- プログラム上はCPUのthreadに近いが、動作は全く異なる
- CUDA “core”と名前はついているが、隣のcoreと同じ命令しか実行できない

→ CPUの世界では、コアというよりAVXやSSE (SIMD)にも少し近い

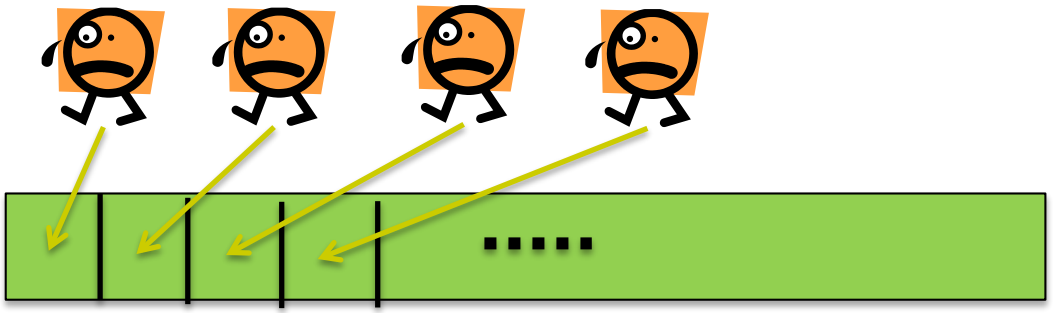
GPU上の計算時間短縮

「コアレスド・アクセス (coalesced access)」によるメモリアクセス効率化

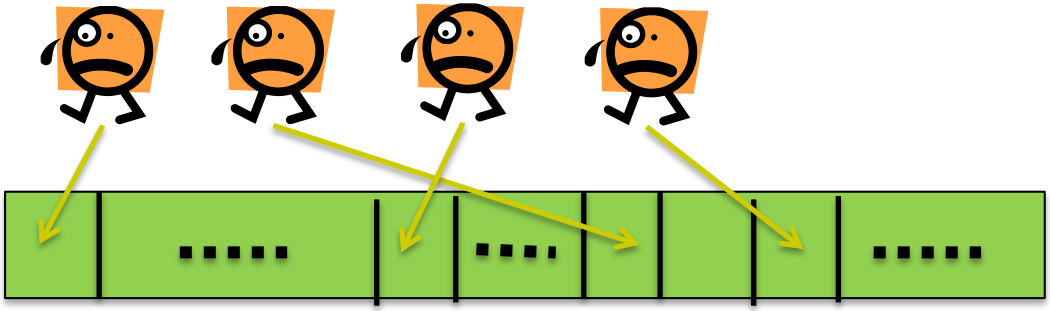
グローバルメモリのアクセスの効率化：コアレスド・アクセス

- 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

より具体的には

あるwarp中のスレッドたちが(同時に)メモリreadを行ったとする。

このとき、メモリからやってくるデータは(alignされた)128バイト単位

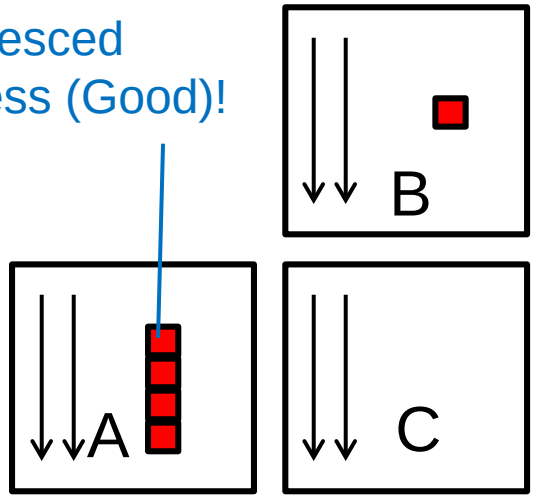
32スレッドのアクセス対象がその単位におさまらない場合は、(128 x n)バイトの転送(1≦n≦32)が必要 → より時間がかかる

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
  - mm-cuda/mm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column-major

mm

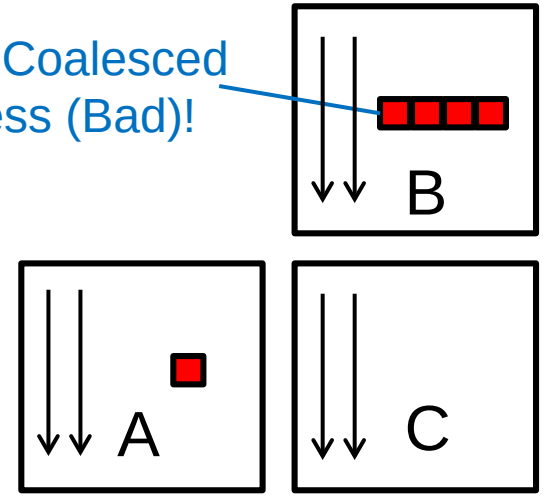
Coalesced Access (Good)!



赤い箇所は隣り合ったスレッドが同時に読み込む箇所

別バージョンmm-nc

Non-Coalesced Access (Bad)!



1024x1024x1024 → 22msec

5倍も性能違う

115msec

GPU上の計算時間短縮

「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

- 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？

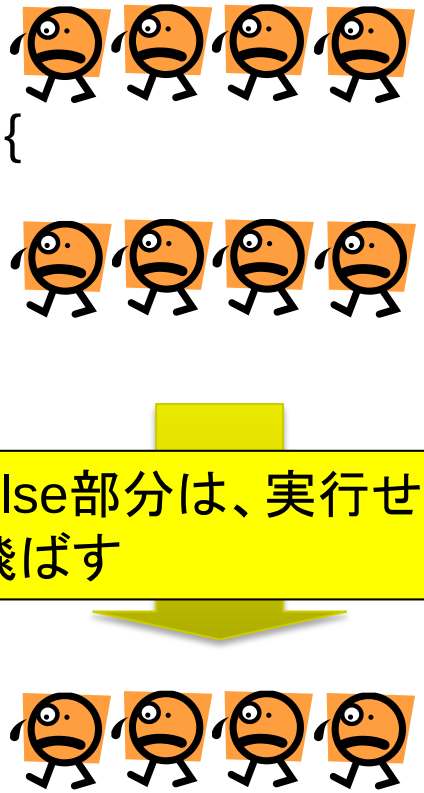
  - ソフトウェア上は別スレッドなので、論理的には足並みがそろわなくなる
  - 実際の動作では、Warp内のスレッド達の「意見」がそろうか、そろわないかで、動作が異なる

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

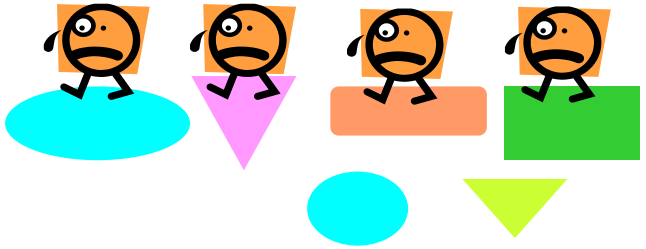
これを**divergent**
分岐と呼ぶ

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

ここまでを振り返って:
GPUに向いていない計算とは？

- Warpの存在→メモリアクセス・命令実行とも、定型的な場合が得意
- 逆に言うと、非定型的な計算は不得意
  - CPUよりも遅い場合も



タスク並列とか・・・

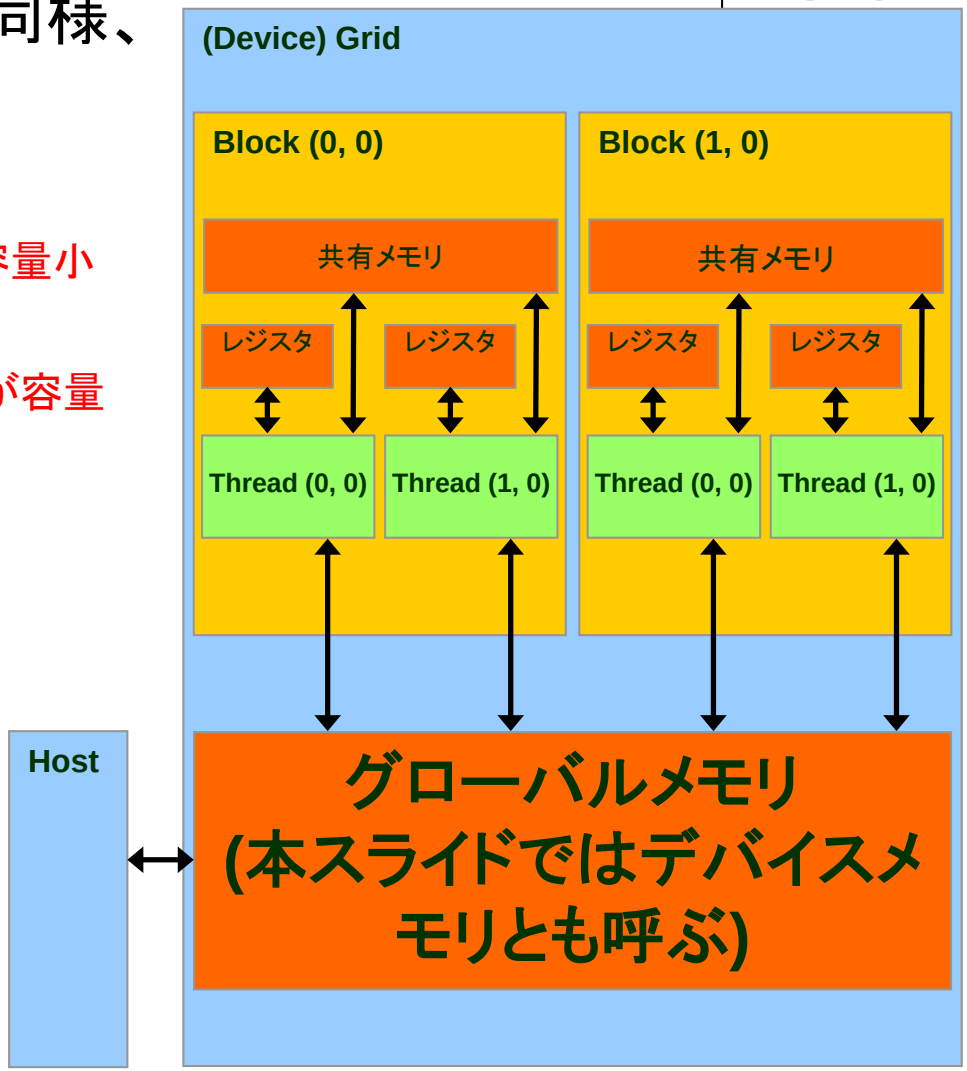
- さらに残念なお知らせ:
  - スレッド間の排他制御なし
    - Atomic Operationを使って、近いことは一応可能
  - バリア同期は限定的
    - スレッドブロック内なら、__syncThreads();
    - スレッドブロックをまたぎたいなら、いちどCPUに戻る必要

GPU上の計算時間短縮
「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）

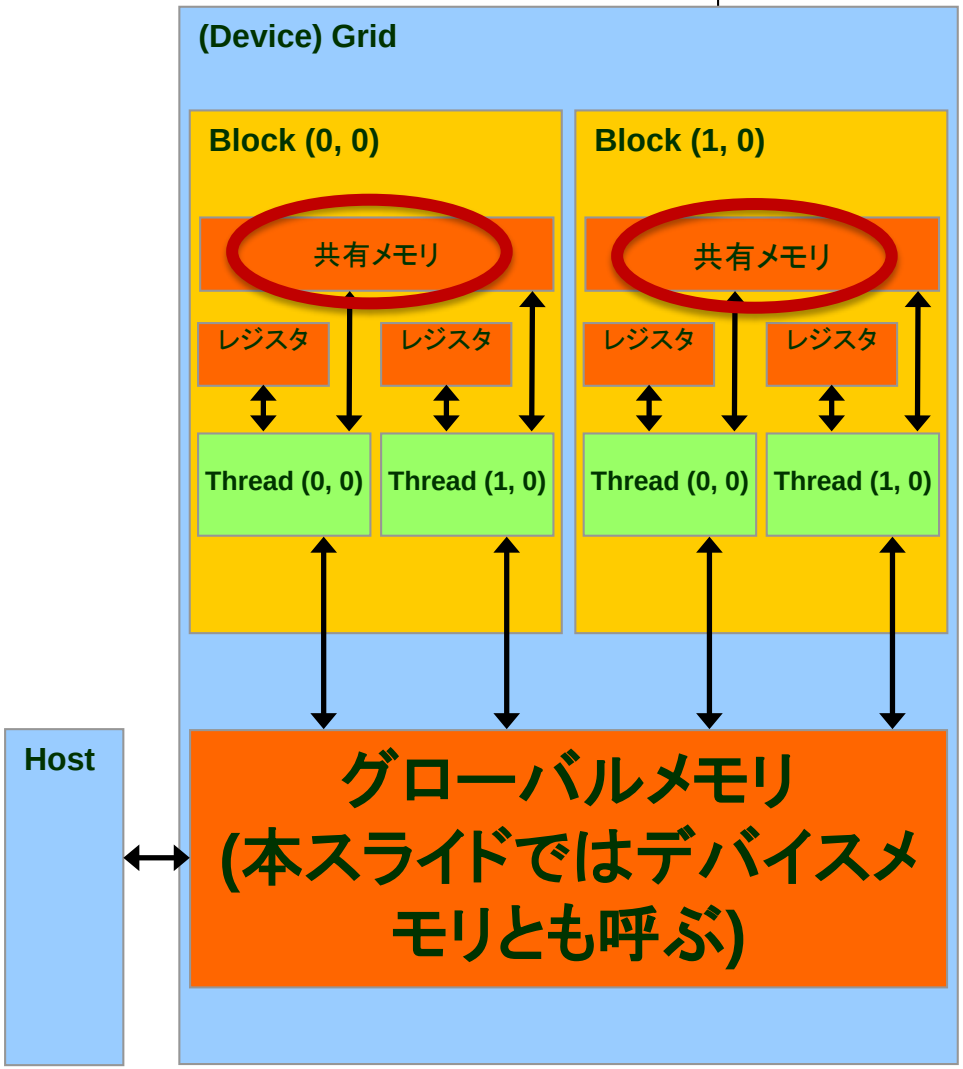


それぞれ速度と容量にトレードオフ有
（高速＆小容量 vs. 低速＆大容量）
→ メモリアクセスの局所性が重要

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

- これまでの知識で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)
  - 用語としては誤解を招く･･･あくまでNVIDIAがつけた名前
- __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

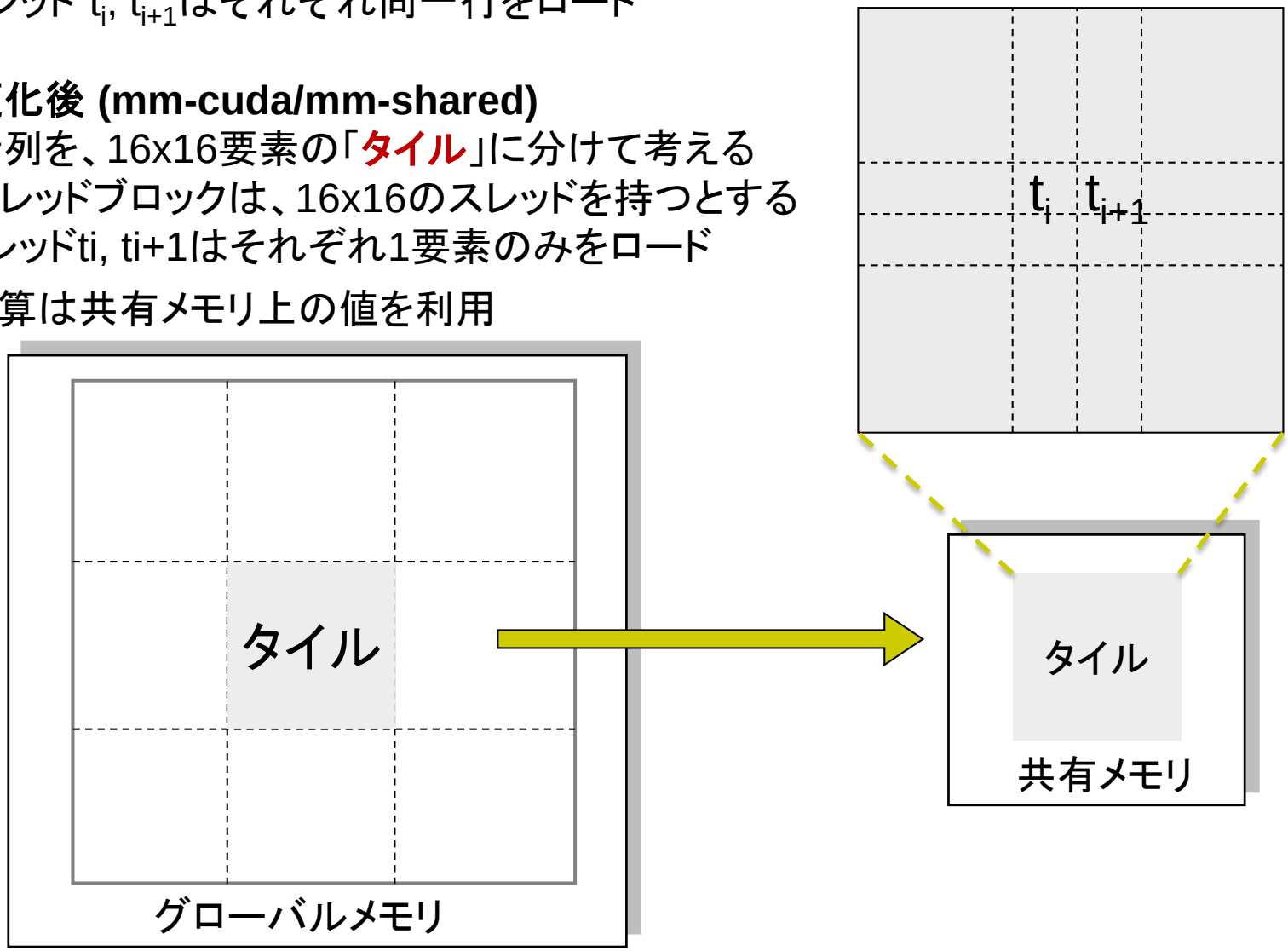
最適化前 (mm-cuda/mm)

- 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 (mm-cuda/mm-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

- CPUで計算

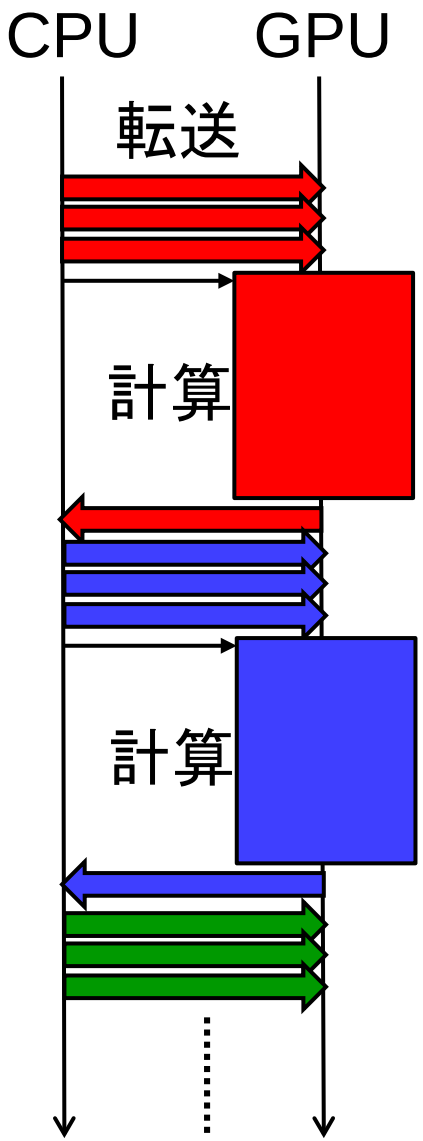
→ **約8.3秒** (gcc –O2でコンパイルした場合)

- GPUの1スレッドで計算 → **約200秒**
- GPUの複数スレッドで計算 → **約0.027秒**
- GPUの複数スレッドで計算し、共有メモリも利用

→ **約0.012秒(!)**

転送時間の隠ぺい
cudaStreamの利用

転送コストを無視できない場合がある



- 連続して、複数の行列積を行うとする。入力データはCPU上にある。
  - C1 = A1 × B1
  - C2 = A2 × B2

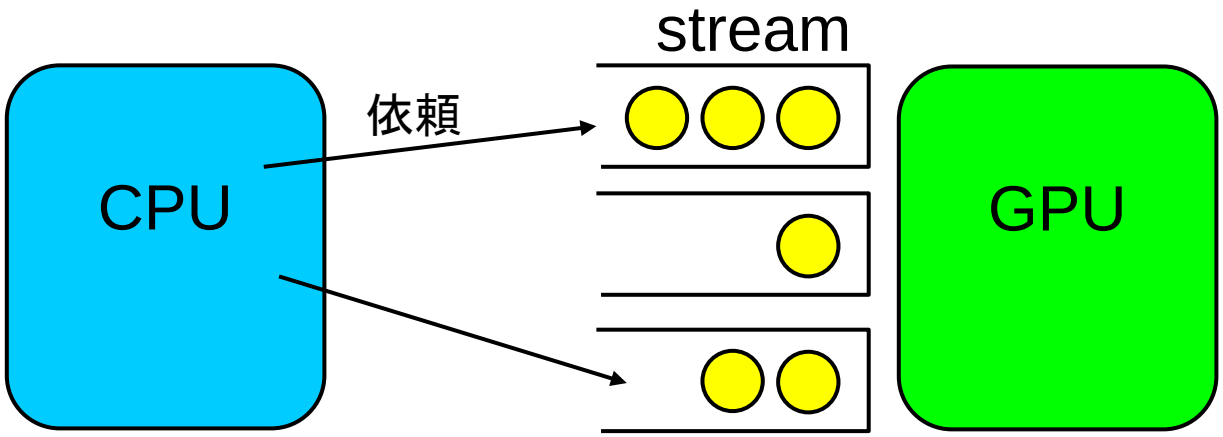
    ….
  - Cn = An × Bn
- データ転送の間、GPUが何もしないのはもったいない

→ 転送時間の隠ぺいにcudaStreamが役に立つ

cudaStreamによる非同期実行 (1)

Streamとは何か？

- GPUが、CPUからの仕事依頼を受ける窓口のようなもの
  - 各streamには、複数の仕事がたまる場合がある
- CPUからGPUに頼む仕事とは、
  - データ転送 (ホスト → デバイス向き)
  - カーネル関数呼び出し
  - データ転送 (デバイス →デバイス向き)



これまでのプログラムでは、”default stream”を使っていた

cudaStreamによる非同期実行 (2)

streamの作成

```
cudaStream_t str;
cudaStreamCreate(&str); // stream strを一つ作成
```

streamを指定して、データ転送

```
cudaMemcpyAsync(dst, src, size, type, str);
```

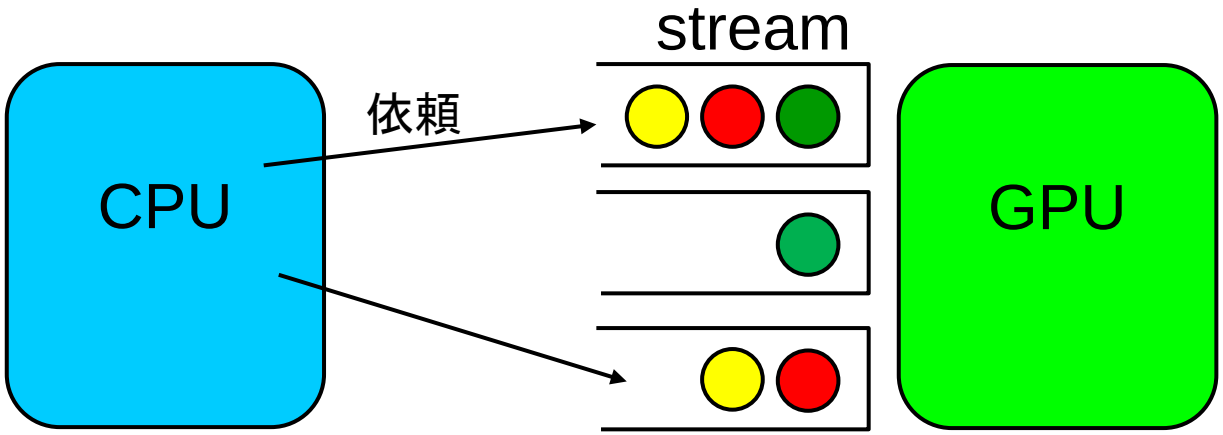
streamを指定して、GPUカーネル関数呼び出し

```
func<<<gs, bs, 0, str>>>( … ); // 第3引数は共有メモリ関係
```

stream上にある仕事が終了するまで待つ

```
cudaStreamSynchronize(str);
```

GPUが仕事をこなすルール



- 同一stream上の仕事は、早く積まれた順に逐次に行う
  - 異なるstream間では順不同
- 異なるstream上、かつ種類(H→D, カーネル、D→H)が異なる仕事どうしは、同時に実行できる
  - stream1の「H→D転送」と、stream2の「カーネル呼び出し」は、同時に実行できる

計算と転送の同時実行(オーバラップ)による性能向上

下記の独立した計算を、別々のstream(n個)に割り当て、GPUへ次々に依頼

- C1 = A1 × B1
- C2 = A2 × B2

  ….

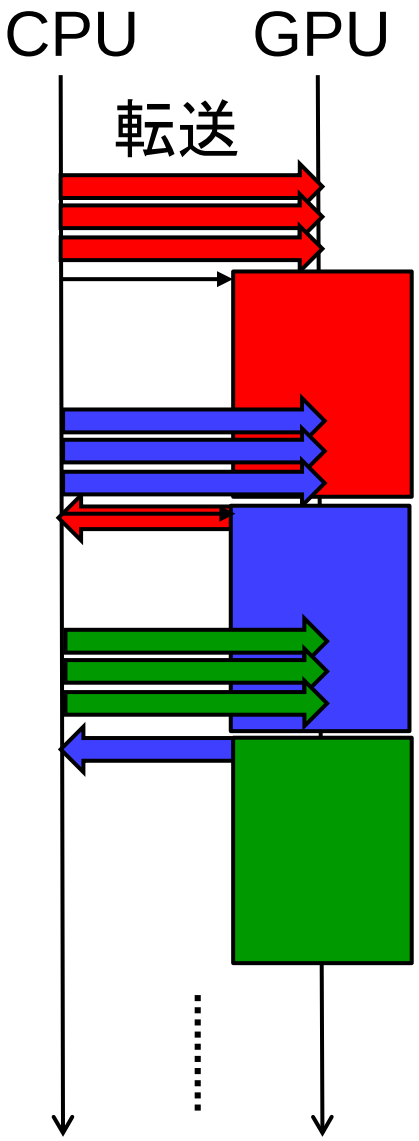
- Cn = An × Bn

→計算と転送が同時実行され、性能が向上

ただしあくまでも、

(総計算時間 + 総転送時間)が

max(総計算時間, 総転送時間)になる程度



なお、上の方法はstream n個およびデバイスメモリ上の配列を3n個用いる点が非効率的である。
nにかかわらず、2個のstream・3x2個の配列を使いまわして、同様の効果を得ることができる (ダブルバッファリング)

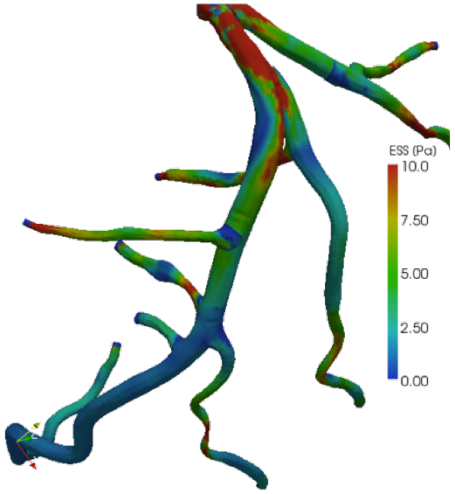
マルチGPUの利用について

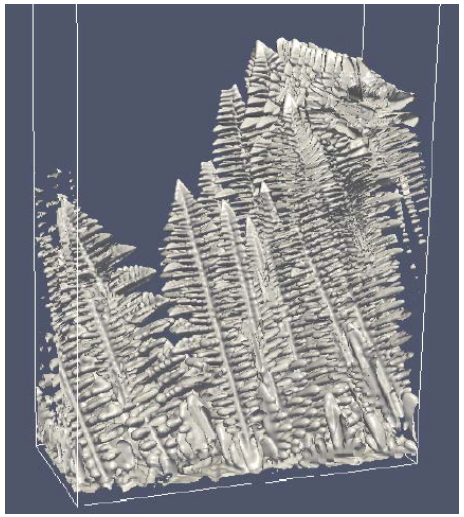
マルチGPUの利用

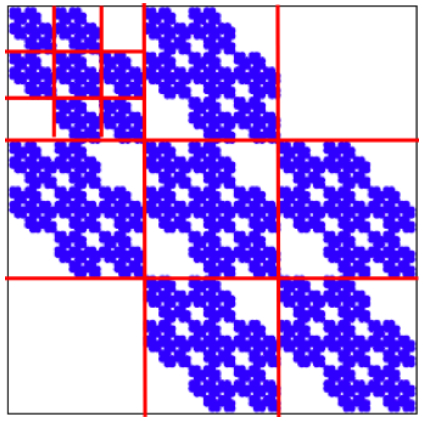
- GPU内の多数CUDA coreを用いてプログラムの高速化可能だが、限界がある

⇒ 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





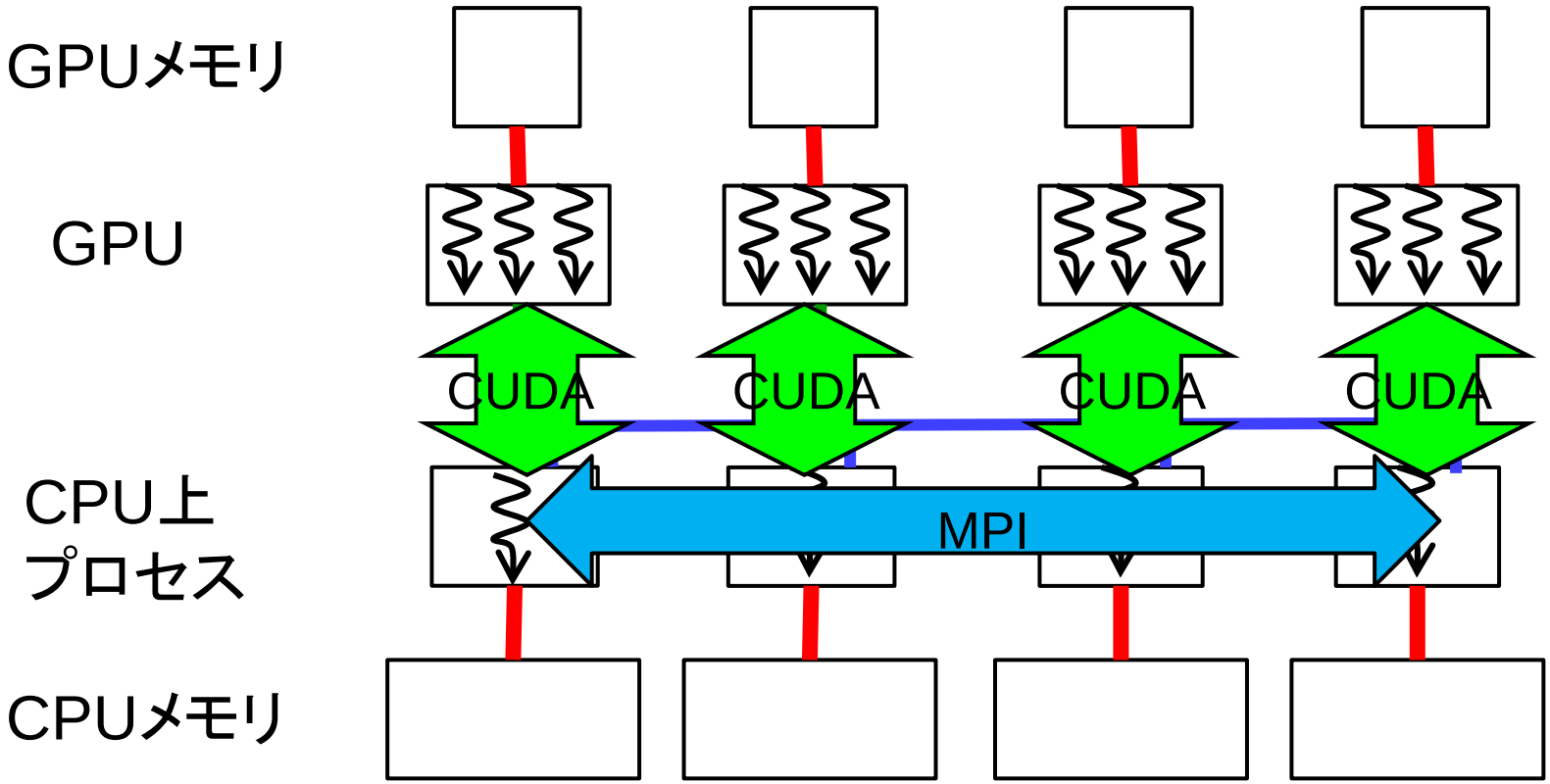


マルチGPUの利用

基本方針を決める:

- MPI+CUDA: 1プロセスが1GPUを担当
  - 以降、このケースを少しだけ解説
  - mpicudatestサンプル
    - ほとんどMPI通信しない、意味の薄いものですが
- OpenMP+CUDA: 1スレッドが1GPUを担当
  - 欠点:1ノード3GPUまで

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
  - MPI_Send, MPI_Recvなどに指定できるのは、「原則」ホストメモリ上のポインタ
  - GPUから別のGPUにデータを移動するのは手間がかかる
    - 送り側プロセス：cudaMemcpy(D→H), MPI_Send
    - 受け側プロセス：MPI_Recv, cudaMemcpy(H→D)

⇒ ただし、最近のMPI (GPU direct対応のMVAPICH 1.8以降など)だと、デバイスメモリポインタを通信に使える

MPI+CUDAの注意(1): プログラミング

- 1ノードに複数GPUが搭載されている場合、デバイス番号を指定する必要がある
  - 各プロセスが「最初」にcudaSetDevice(デバイス番号); を呼ぶのがよい
    - MPI_Init()後、cuda関連の関数(cudaMalloc等)を呼ぶ前に
    - そうしないと、3プロセスともGPU0番を使ってしまって非効率
  - TSUBAME2ではノードあたり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): コンパイル

?? MPIプログラムのコンパイルにはmpicc/mpic++, CUDAプログラムにはnvcc。どうすればよい ??

おすすめの方法は、

(1) コンパイル : .cuファイルをnvccでコンパイル。MPI関連のヘッダの設定が必要

(2) リンク: .oファイルをmpic++でリンク。CUDA関連ライブラリのリンクが必要

- ~endo-t-ac/ppcomp/16/mpicudatest/Makefile を参照
- Makefileの中身を、利用するMPI, CUDAに応じて書き換えてください
  - which mpicc, which nvcc で確認

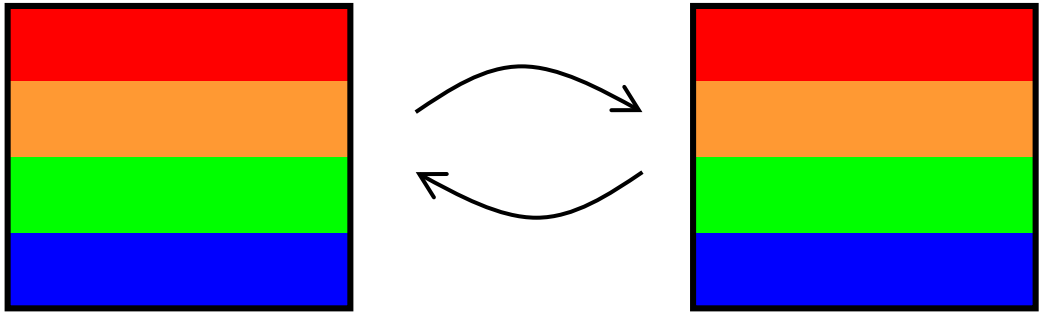
MPI+CUDAの注意(3): プログラム実行

- MPIプログラムと思ってmpirunなどで実行する
- ノードあたりのプロセス数に注意。ノードあたり3GPU使うのであれば、それに合わせる
  - t2sub … -l select=10:mpiprocs=3 … など

[参考]OpenMPIの場合、バージョンによっては、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
- これがないと、MPI通信関数かcudaMemcpy等が失敗することがある
- ↑ MPIとCUDAの両方がDMAを用いるため

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デバイスメモリ上にあることに注意

MPI+CUDA版Diffusionの流れの例

MPI_Init

cudaSetDevice(rank % 3); // 必要ならば

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

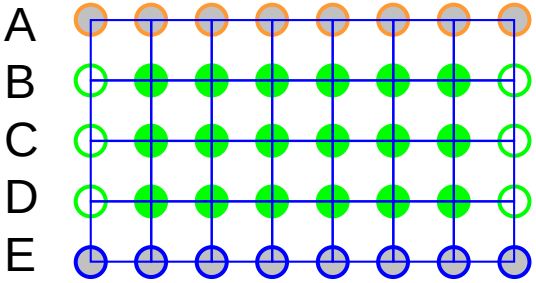
行A, EをCPUからGPUへコピー

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、GPUパートで取り上げられなかった話題

- CUDA Profiler
  - GPUカーネルの性能解析
- Atomic operation, shuffle operation
- 最新のCUDAやGPU特有の機能
  - Unified memory, HyperQ
- 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
- 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

  予定パート:

  - OpenMPパート
  - MPIパート
  - GPUパート

- サンプルプログラムについては、TSUBAMEの

  ~endo-t-ac/ppcomp/16/ 以下から各自コピーすること

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] 行列積サンプルmm-cuda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)版の性能とも比較してください。
- データ転送コストを考慮に入れる場合・入れない場合それぞれについて速度を示すこと
  - 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
- プログラムを改良してもok
  - 各スレッドの担当領域の変更、共有メモリによる高速化、
  - 小行列の組み合わせ(ブロッキング)、などなど

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。

- 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、行列積をOpenACCで記述したときの性能は？
- たとえば、Reduction処理を含むpiサンプルをGPU化できるか？
- 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
    - プロセッサ(コア)数を様々に変化させたとき
    - 問題サイズを様々に変化させたとき(可能な問題なら)
    - 「XXコア以上で」「問題サイズXXX以上で」発生する問題に触れているとなお良い
  - 高性能化・機能追加など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good
  - 作成したプログラムも提出
    - zipなどで圧縮してOCW-iに提出
    - 困難な場合は，TSUBAME2の自分のホームディレクトリに置き，置き場所を連絡(パーミッションに注意)

課題の提出について

- GPUパート提出期限
  - 6/16(木)
  - MPIパート(6/13)にも注意
- 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（最終回）

- ソフトウェアを高速化したいプログラマのためのメモリアーキテクチャ (予定)