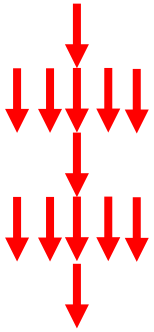
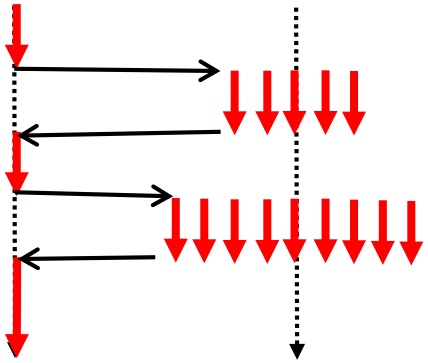
2016年度
実践的並列コンピューティング

GPUプログラミング (2)

遠藤 敏夫
endo@is.titech.ac.jp

CUDAとOpenMPの比較





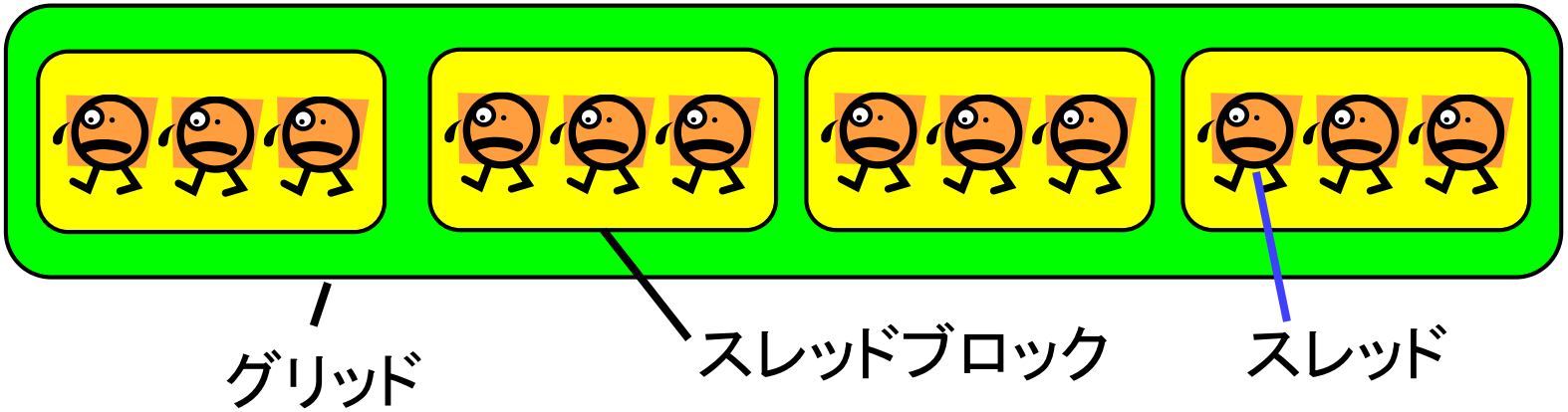
	OpenMP	CUDA
並列実行に使うプロセッサ	CPU	GPU
並列実行箇所の指定	#pragmaをブロックまたは文につける	GPUカーネル関数として切り出す (pthreadの考え方が近い?)
スレッド数の指定	環境変数・通常固定	呼び出しごとに変えられる かつ、最大3次元×2段階
望ましいスレッド数	CPUコア数以下	CUDAコア数以上
メモリの扱い	すべて共有	GPU上スレッド間は共有、CPUとGPU間(または異なるGPU間)では分散

最大3次元×2段階のスレッド指定(1)

例1: func <<< 4, 3 >>> ();

スレッドブロック数
= gridDim

(ブロックあたり)スレッド数
= blockDim

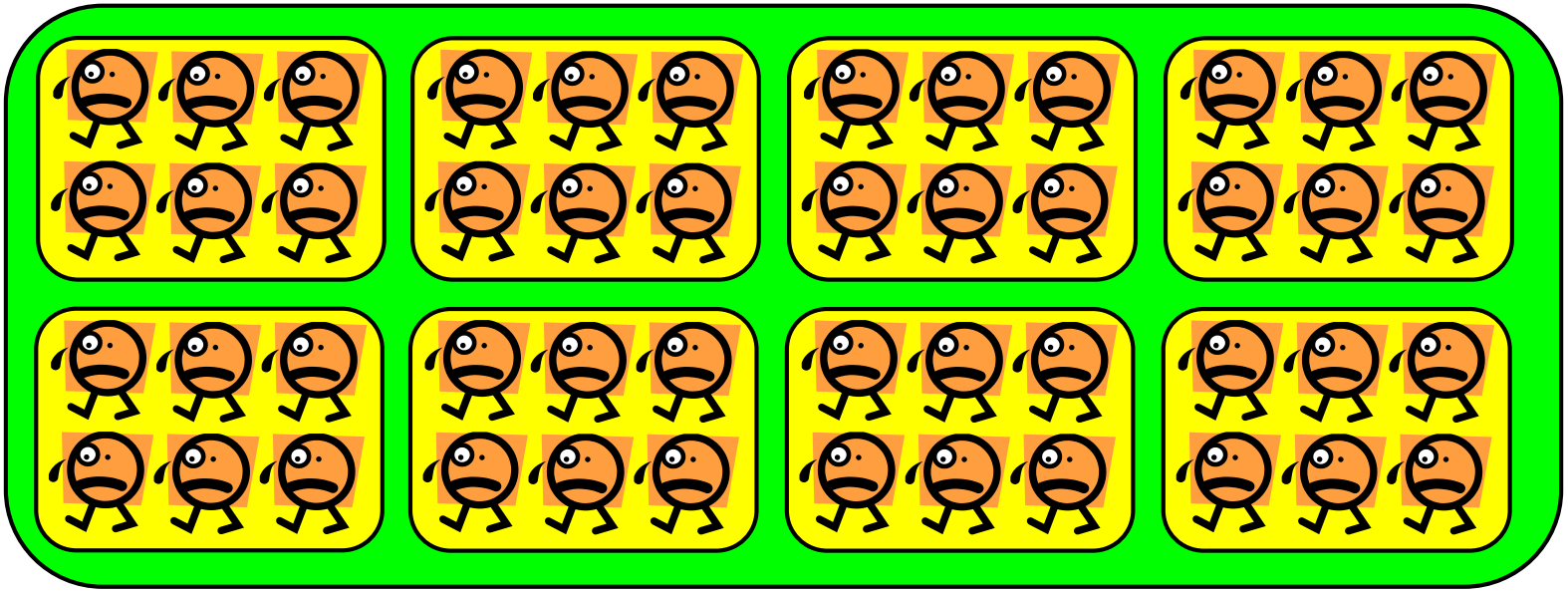


最大3次元×2段階のスレッド指定(2)

スレッドブロック数・スレッド数に指定できるのは

- dim3構造体 (x, y, zの三次元を表す)
- または、int
  - <<<a, b>>>は、<<<dim3(a,1,1), dim3(b,1,1) >>> の略だった

例2: func <<< dim3(4,2,1), dim3(3,2,1) >>> ();

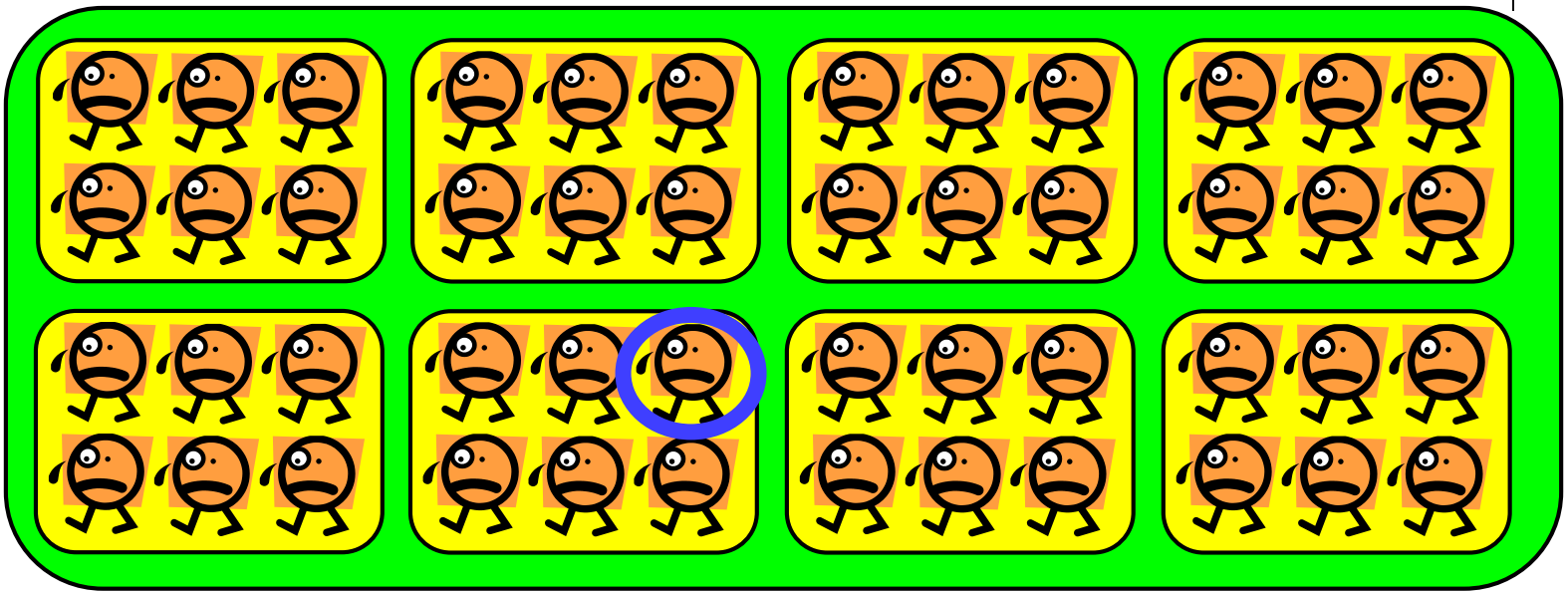


※ これは二次元・二次元の例

※ 略して func <<<dim3(4,2), dim3(3,2)>>>; とも書ける

多次元の場合のスレッドID

func <<< dim3(4,2,1), dim3(3,2,1) >>> (); の場合



- どのスレッドが見ても
  gridDim.x=4, gridDim.y=2, gridDim.z=1
  blockDim.x=3, blockDim.y=2, blockDim.z=1
- 印のスレッドから見ると
  blockIdx.x=1, blockIdx.y=1, blockIdx.z=0
  threadIdx.x=2, threadIdx.y=0, threadIdx.z=0

ブロック数・スレッド数の制限

ブロック数・スレッド数に指定可能な最大値にも注意

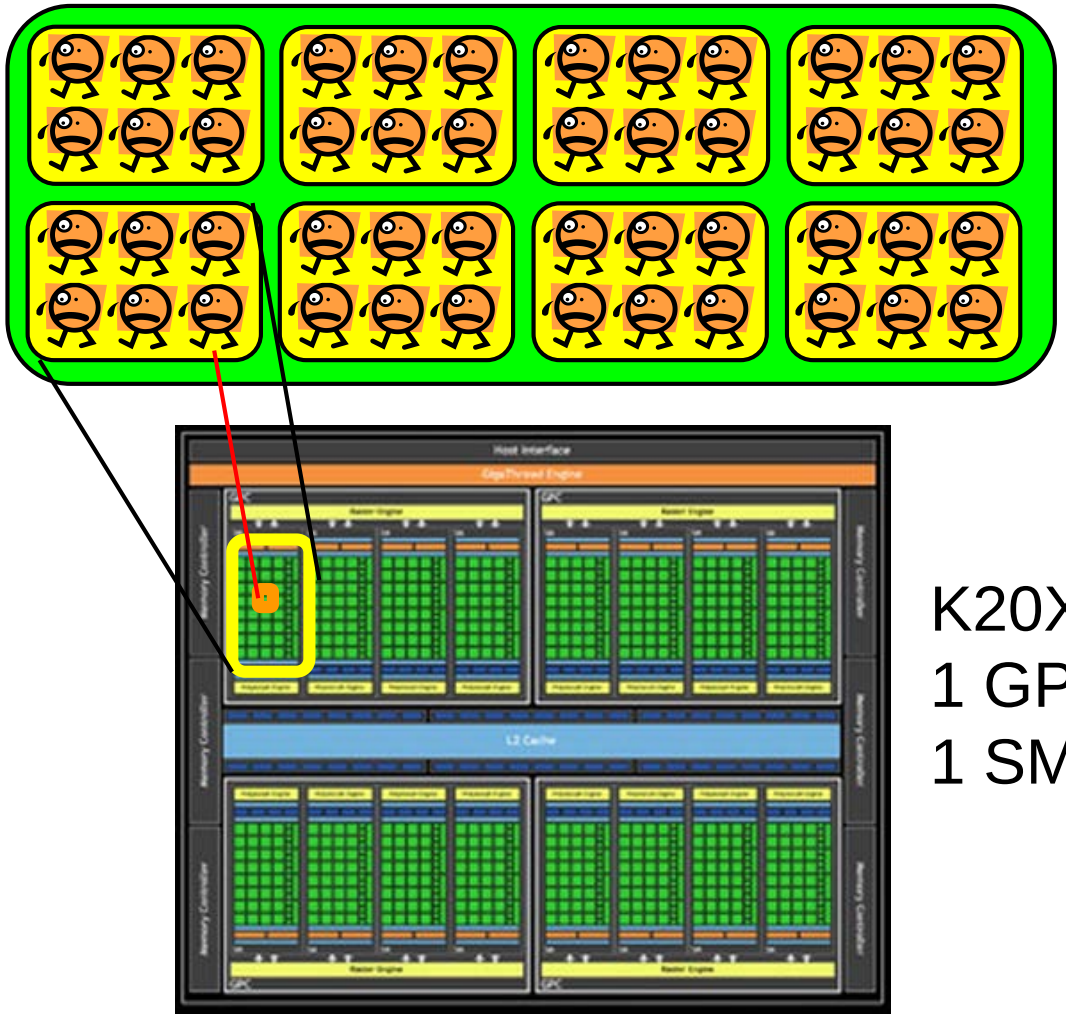
- TSUBAMEのK20Xでは
  - スレッドブロック数：xは2^{31}-1まで、y・zは65535まで
  - ブロック中スレッド数: x, yは1024まで、zは64まで。さらに総数1024まで

    結構ひっかかりやすい。結局、ブロック中スレッド数は固定にして、ブロック数を大きくすることが多い

  - GPUによって違う。CUDA C Programming GuideのAppendix G参照のこと
    - http://docs.nvidia.com/cuda/cuda-c-programming-guide
    - K20Xの「Compute capability」は3.5

なぜCUDAではスレッドが二段階か(1)

- ハードウェアの構造に合わせるため



GPUプロセッサの構造

K20Xの場合
1 GPU = 14 SMX
1 SMX = 192 CUDA core

なぜCUDAではスレッドが二段階か(2)

- 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
  - 隣接32スレッド(=warp)は必ず足並みをそろえて動作

→ 1ブロックが32スレッド未満だとCUDA coreが無駄に

これまでの議論から、(TSUBAMEの)K20X GPUで望ましいのは

- 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(1024以下)

GPU上のスレッド数の考え方が、CPUと違う件 (1)

- CPU: スレッド数≦物理コア数がよい
  - TSUBAMEでは12
- GPU: 総スレッド数>>物理(CUDA)コア数がよい
  - しかも、ぎりぎりよりも、数倍以上多い方が速い傾向
  - TSUBAMEでは、GPUあたり2688CUDAコア

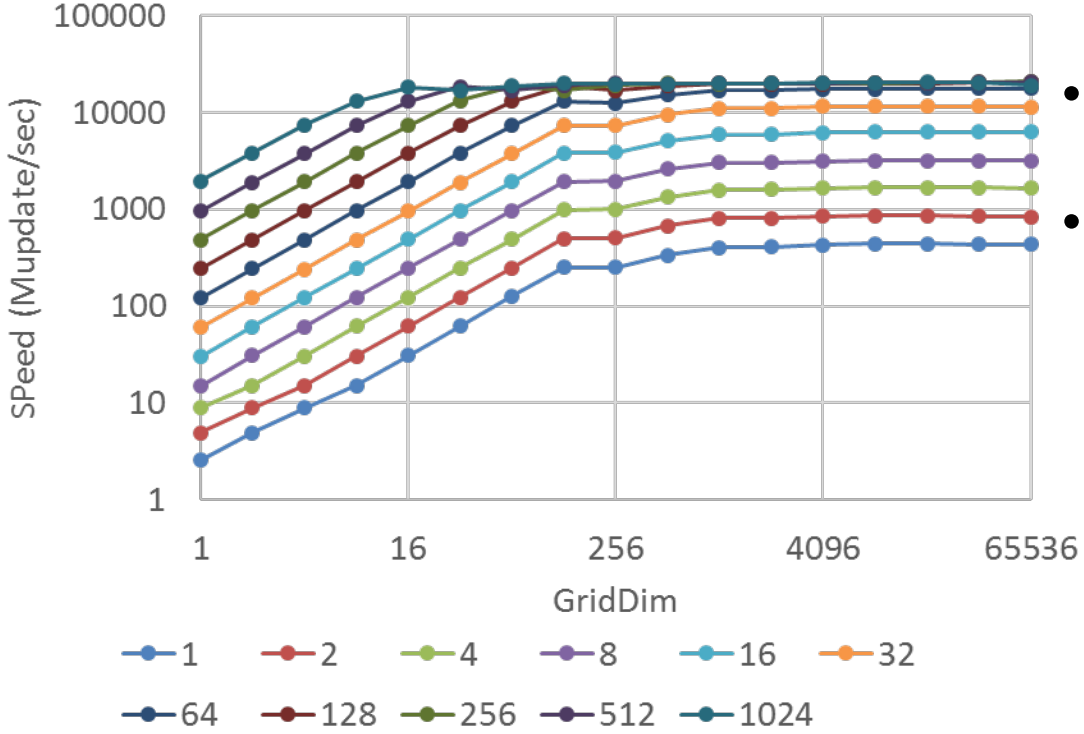
  ⇒　総スレッド数は10,000以上が良く、百万などもok

この議論に基づき、inc_parでは総スレッド数=配列サイズとした

一方、dim3で指定する次元はあまり性能に関係ない
この点は、プログラムの書きやすさ優先でよい

スレッド数による性能変化

- inc_parの改造版inc_rrの性能
  - スレッド数可変、cyclic分割
  - (更新した配列サイズ / 実行時間)を測定。cudaMemcpy時間除く
  - TSUBAME2.5のK20X GPU利用

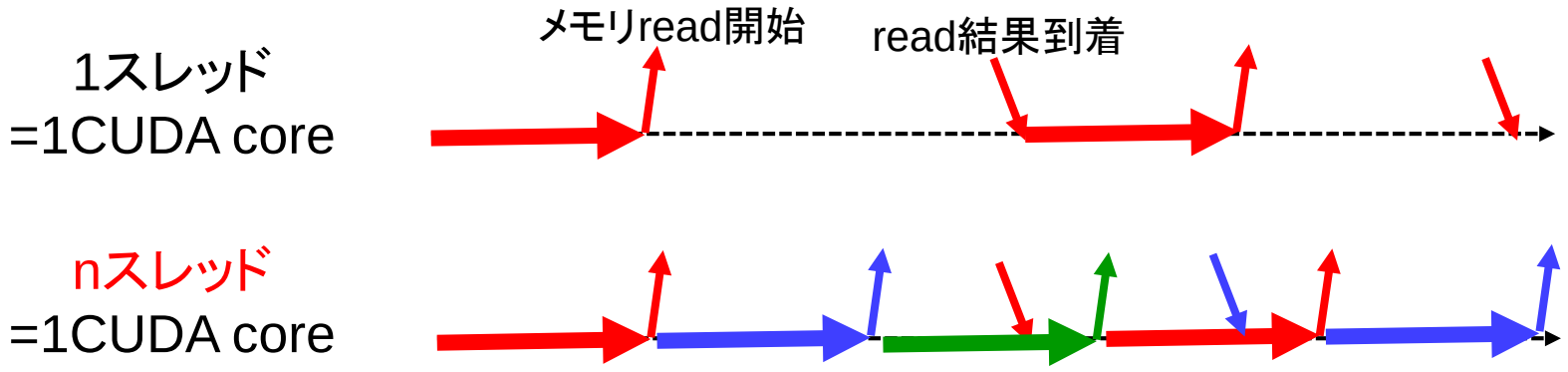


様々なBlockDim (ブロックあたりスレッド数)

- GridDim, BlockDimとも、大きいほうが有利
- (このプログラムの場合) 性能が飽和するのは GridDim × BlockDim≧16384 かつBlockDim ≧64
  - スレッド数2688ではまだ足りない

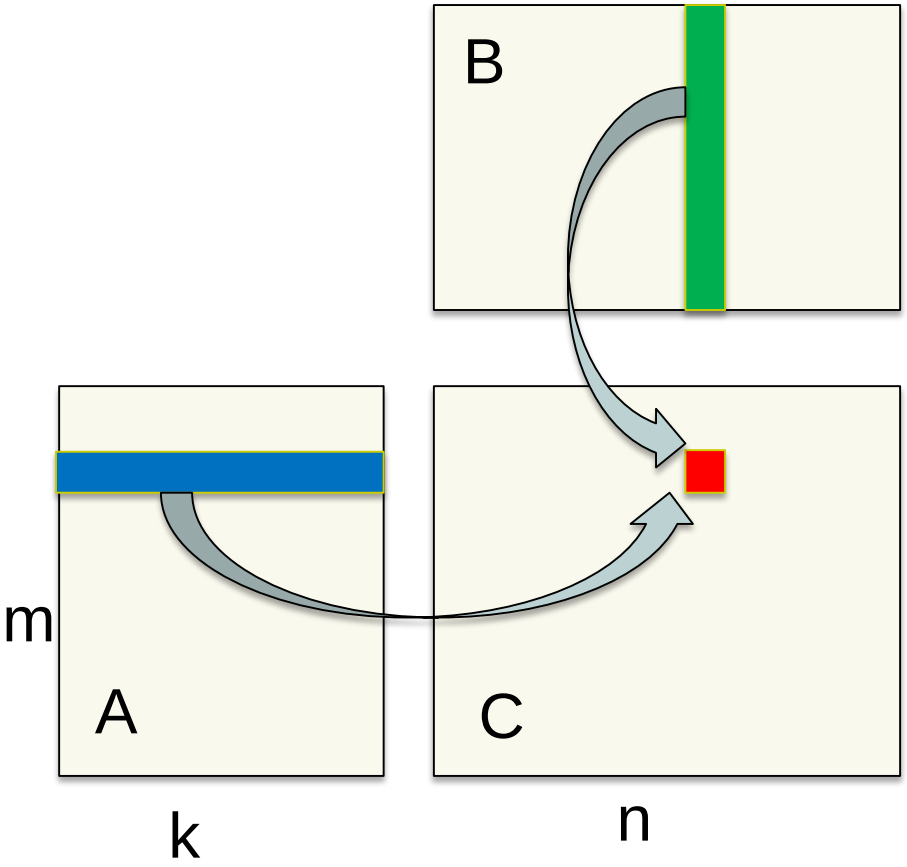
GPU上のスレッド数の考え方が、CPUと違う件 (2)

- CPUとの違いの理由：GPUではコンテクストスイッチが非常に軽い
  - CPUではレジスタ・スタックの退避などをOSがソフトウェアで行う(遅い)
  - GPUではハードウェアによりほぼゼロクロック(速い)
- コア数ちょうどより、大幅に多い方が速い理由：
  - メモリアクセスによるひまな時間(ストール)を、他のスレッドが埋めることができる
  - Intel CPUではHyperthreadに相当するが、こちらは物理コア×2まで



mm-cuda:
行列積サンプルCUDA版

C = A x B



行列Cの要素$C_{i,j}$を求めるには

- Aの第i行全体
- Bの第j列全体

の内積計算を行う

計算量は

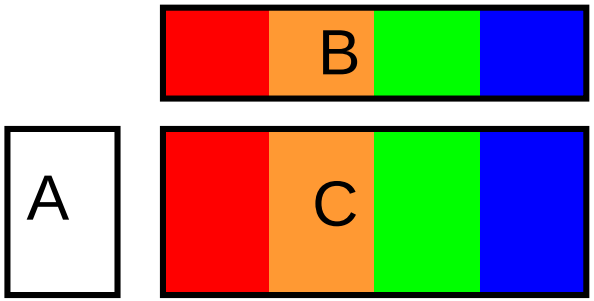
一要素あたりO(k)

全体でO(mnk)

以下、行列配置はcolumn-majorとする

OpenMP版ではどうだった?

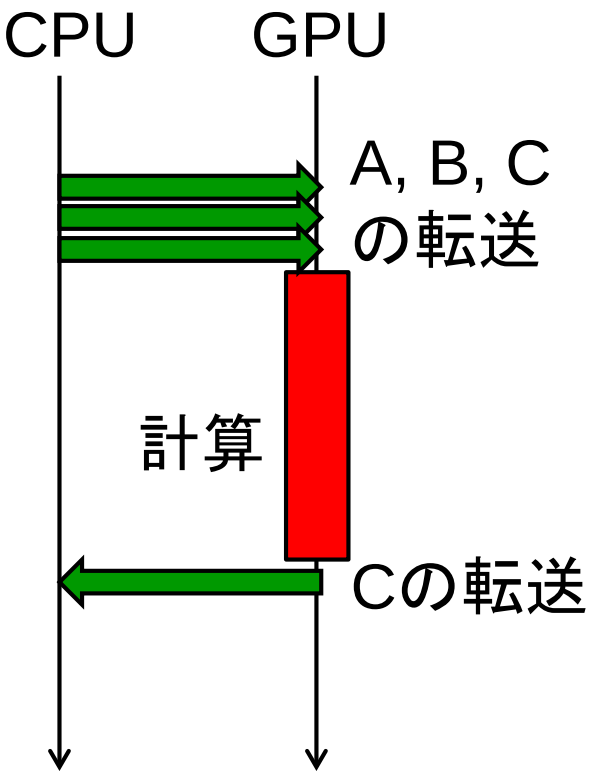
- 三重ループで記述
  - Cの行ループ
  - Cの列ループ
  - 内積のためのループ
- Cをスレッド間で分割し、担当領域を計算
  - Cの列ループ(または行ループ)に#pragma omp for



4スレッドの場合の分担

CUDA版ではどう考えるか (1)

- データの置き場:
  - 少なくとも計算の時点では、A, B, Cともデバイスメモリ上にあるべき
  - 初期化はCPU, GPUどちらでやるか?
  - → 今回はCPUで初期化し、その後cudaMemcpyするようにした
  - → 計算後に、結果CをやはりcudaMemcpyでCPU側へ



CUDA版ではどう考えるか (2)

- どこを並列化するか
  - OpenMPと同様、別スレッドはCの別の部分を計算させる
    - 内積ループの並列化は、race conditionのため困難
  - OpenMPと異なり、

総スレッド数は10,000以上が良く、百万などもok

⇒ 今回は「スレッド数 = Cの要素数」としてしまう

⇒ 1スレッドは一つのC_{ij}だけを計算

⇒ 内積の一重ループのみで済む

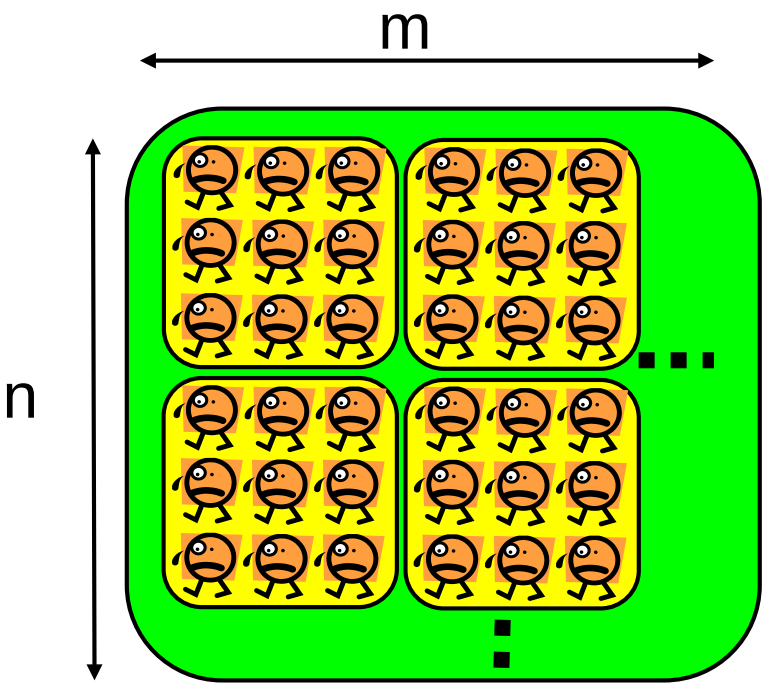
- これはOpenMPではありえない
- 一方、CUDAでもスレッド数 < Cの要素数というプログラムを書くことはもちろん可能

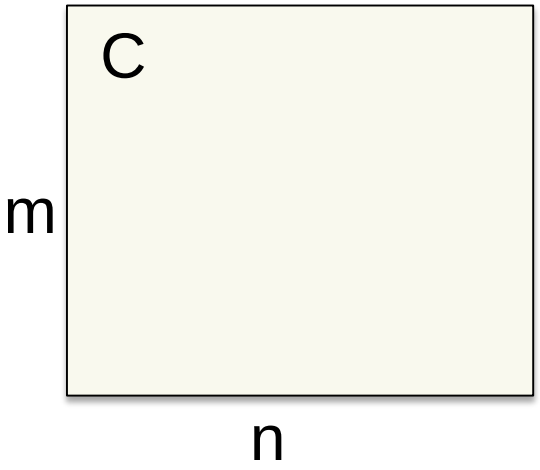
CUDA版ではどう考えるか(3)

- 総スレッド数はmn (行列サイズ依存)と決まった。では、gridDim, blockDimをどう決めるか?
- 前提: blockDimはCUDAの制約によりあまり大きくできない

→ 今回は、blockDimを小さめの固定数(16x16=256)、gridDimを可変に

→ プログラムを楽にするため、二次元指定を用いる





※なぜxを行に対応させたかは、次回説明

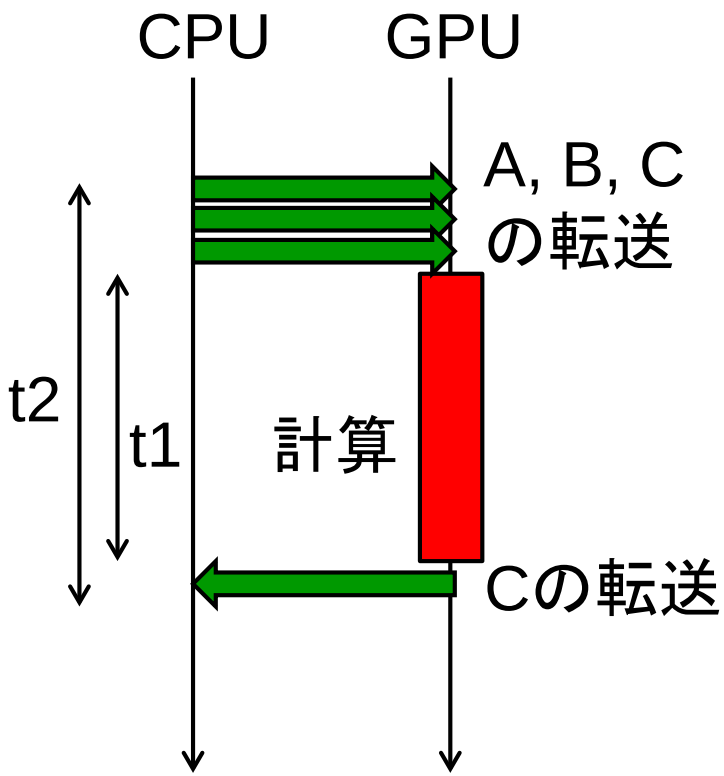
実際の呼び出し･スレッド番号取得

```
matmul_kernel<<<dim3(m / BS, n / BS), dim3(BS, BS)>>>
        (DA, DB, DC, m, n, k);
```

BSは前もって適当に決めた数(16)
実際には、割り切れないケースのため切り上げ

```
matmul_kernel関数内:
          :
   j = blockIdx.y * blockDim.y + threadIdx.y;
   i = blockIdx.x * blockDim.x + threadIdx.x;
          :   以降、自分はCijを計算
```

データ転送時間を性能測定に含めるべきか否か？



- 一概にはどちらが正しいとは言えない。実用的なプログラムでは、前後の文脈によるため
- サンプルプログラムでは、t1とt2両方表示
- t1 ≒ cmnk
- t2 ≒ t1 + d(mk+kn+2mn)
  - c,dはアーキテクチャから決まる定数

行列積性能の簡単な比較

全行列とも1024x1024のとき、いくつかのバージョンを比較:

- mm
  - CPU上の1スレッドで計算 → **約1.1秒**
- mm-omp
  - CPU上の複数スレッドで計算 → **約0.14秒** (12スレッド)
- mm-cuda1t
  - GPUの1スレッドで計算 → **約400秒**

- mm-cuda
  - GPUの複数スレッドで計算
  - → **約0.036秒 (cudaMemcpy除くと0.025秒)**

※ 課題[G1]では網羅的に評価すること！

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」

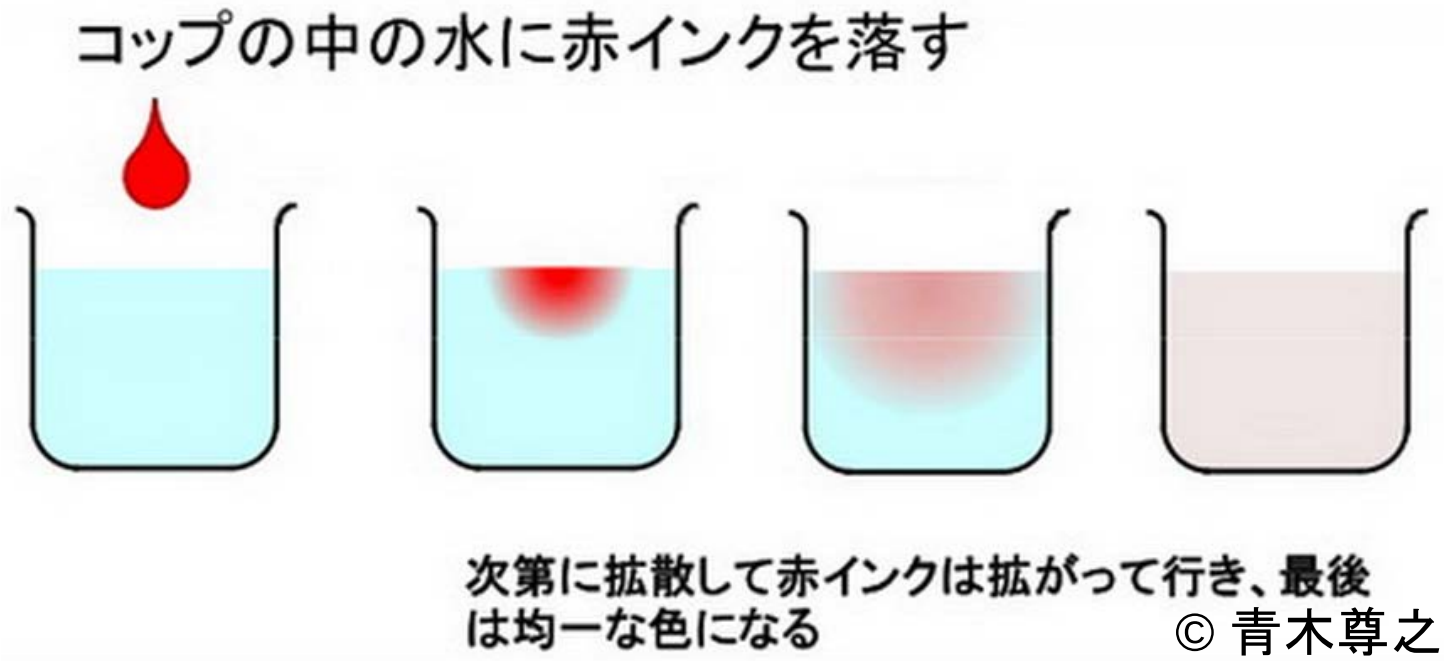
各部分ごとの時間計測を行うには

```
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

再掲
サンプルプログラム: diffusion

拡散現象

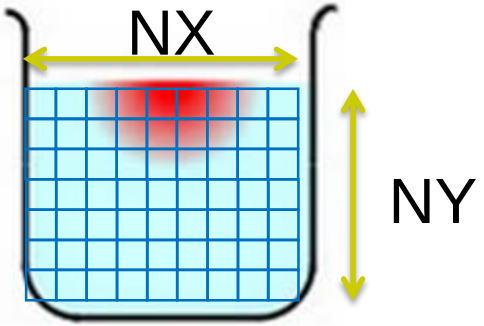


© 青木尊之

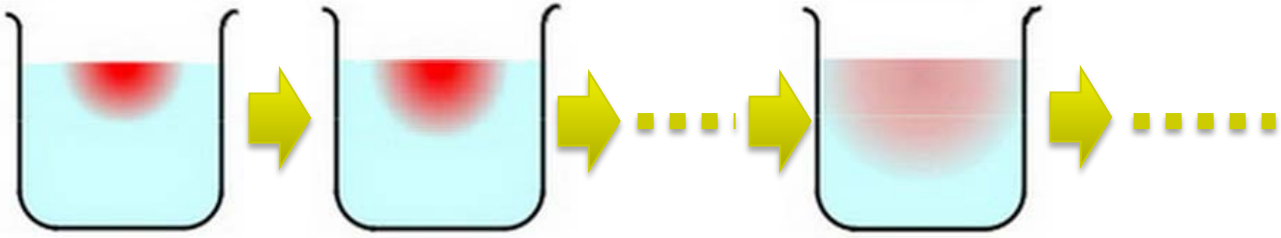
- 各点のインク濃度は、時間がたつと変わっていく
  → その様子を計算機で計算
  - 天気予報などにも含まれる計算
  - GPUで並列化するには？？

再掲:
diffusionのデータ構造

- シミュレーションしたい空間をマス目で区切り、配列で表す(本プログラムでは二次元配列)



- 時間を少しずつ、パラパラ漫画のように進めながら計算する

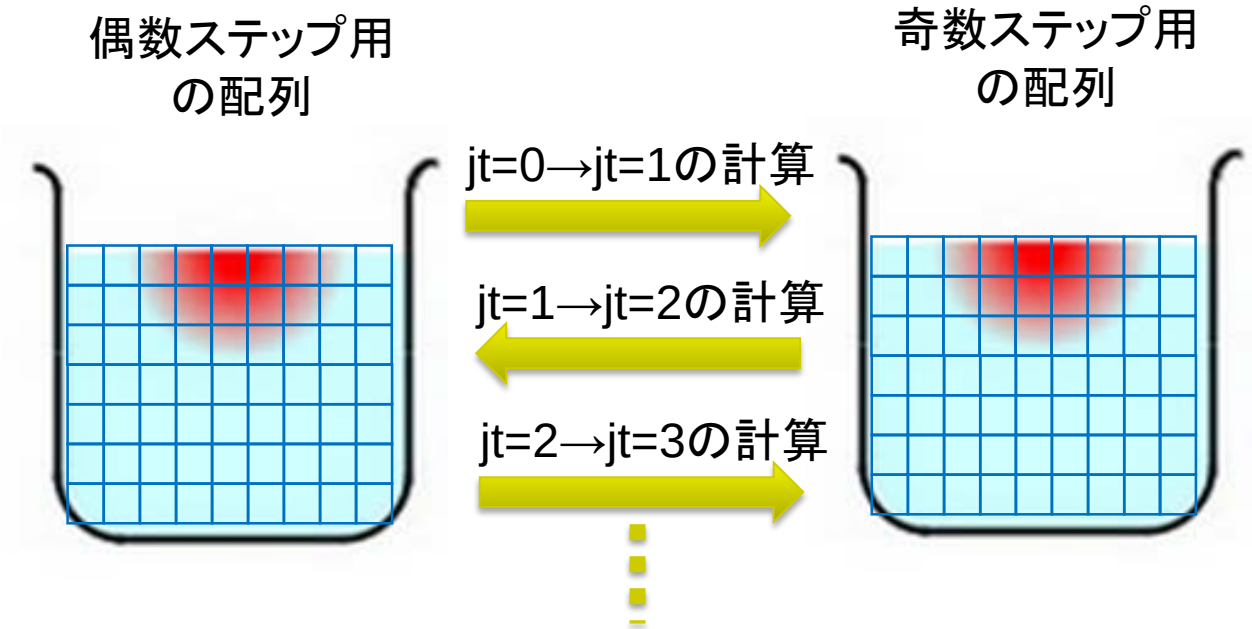


時間ステップ jt=0　　jt=1　　jt=20

再掲:
ダブルバッファリング技術

- 全時間ステップの配列を覚えておくとメモリ容量を食い過ぎる

→ 二ステップ分だけ覚えておき、二つの配列(ダブルバッファ)を使いまわす



※ サンプルプログラムでは、大域変数
float data[2][NY][NX];
で表現

想定されるGPU版diffusionの流れ

CPU上で初期条件作成

cudaMallocでGPUメモリ上の領域確保(配列二枚分)

初期条件の二次元格子データをCPUからGPUへ(cudaMemcpy)

For (jt = 0; jt < nt; jt++) //時間ループ

　　GPUカーネル関数を呼出し、その中で全格子点を計算

　　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(cudaMemcpy)

※ 時間ループの中に(格子全体の)cudaMemcpyを置くと非常に遅い (各自試してみましょう)

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)
  - OpenMPパート(5/19)、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

- GPUプログラミング(3)
  - CUDAプログラムの高速化について