

GPUプログラミング・応用編

東京工業大学学術国際情報センター

はじめに

基礎編では、以下を説明

- GPUプログラミングの基本
- GPU上のスレッドを使った並列プログラミング

しかし、GPUの特徴を考慮することによって、更に高速化が可能

→ 同じ計算を行うプログラムでも、メモリやスレッドなどの使い方の「最適化」によって、**数倍～数十倍実行速度が違う場合も！**

応用編では、基礎編に入れられなかったテクニックや高速化のポイントを説明

NVIDIA社の資料について

- このスライドは説明を簡略化しているため、正確な情報が必要な場合については、下記のNVIDIA社の公式資料(英語)を参照してください
 - CUDA C Programming Guide
 - CUDA API Reference Manualコンテスト内部Wikiに置いてあります

目次

1. はじめに
2. 多次元配列についてのテクニック
3. CUDAプログラムの時間計測に関する注意
4. 「divergent分岐」の削減による効率化
5. 「コアレスド・アクセス」によるメモリアクセス効率化
6. 「共有メモリ」の有効活用
7. おわりに

2. 多次元配列についてのテクニック

多次元配列についてのテクニック

- CPU上では、大域変数として多次元配列を使える
 - `short int h_array[MAX1][MAX2][MAX3][MAX4];`
のように定義できる
 - サンプルプログラムでも多用している
 - GPUではハードルが高い

GPU上で利用する基本方法は？

- 多次元配列をGPUに確保・コピーして計算に用いるには？

以下が基本の方法：

- cudaMallocで領域を確保し、ポインタ型変数を得る
 - たとえば、`short int *d_array;`
- cudaMemcpyで、`h_array`から`d_array`にコピー
- カーネル関数内で、`d_array`の内容にアクセス可能
しかし一次元配列として使わないといけない

`d_array[i][j][k][l]` → コンパイル時エラーになってしまう 

`d_array[i*MAX2*MAX3*MAX4+j*MAX3*MAX4+k*MAX4+l]`

→ 動くけど、プログラムが結構大変 

次スライドで、少し楽にするテクニックを紹介します

GPU上でも多次元配列!

/home/SC12/gpulec/devarray.cu を参考にしてください

(1) 変数の定義に `__device__` をつけると、GPUメモリ上に確保される

例: `__device__ short int d_array[MAX1][MAX2][MAX3][MAX4];`

(2) `cudaMemcpyToSymbol` (CPU→GPUの場合) や `cudaMemcpyFromSymbol` (GPU→CPU) を使う

例:

```
cudaMemcpyToSymbol("d_array", h_array,  
    sizeof(short int)*MAX1*MAX2*MAX3*MAX4, 0);  
cudaMemcpyFromSymbol(h_array, "d_array",  
    sizeof(short int)*MAX1*MAX2*MAX3*MAX4, 0);
```

(3) これなら、GPUカーネル関数内でも `d_array[i][j][k][l]` のように使えて **ちょっと便利!**

前スライドの基本の方法を使うか、新しい方法を使うかは、おまかせします

特殊なコピー関数の詳細

- `cudaMemcpyToSymbol(char *symbol, const void *src, size_t count, size_t offset)`
CPU側のデータを、GPU上の__device__変数にコピー
 - `symbol`: 転送先の__device__変数の名前を、「文字列で」指定
 - `src`: 転送元CPUメモリ
 - `count`: 転送サイズ(バイト単位)
 - `offset`: `symbol`が表すアドレス+`offset`を、転送先とできる。0でもよい。
- `cudaMemcpyFromSymbol(void *dst, char *symbol, size_t count, size_t offset)`
GPU上の__device__変数のデータを、CPU側にコピー
 - `src`: 転送先CPUメモリ
 - `symbol`: 転送元の__device__変数の名前を、「文字列で」指定
 - `count`: 転送サイズ(バイト単位)
 - `offset`: `symbol`が表すアドレス+`offset`を、転送元とできる。0でもよい。

3. 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、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

4. 「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

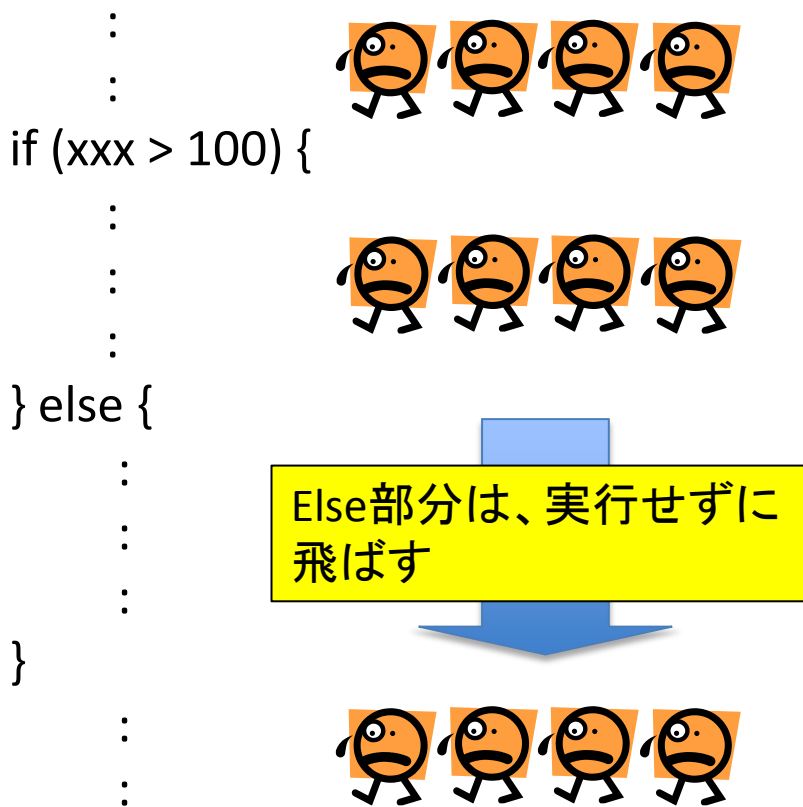
→ If文などの分岐があるとどうなる？

- 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

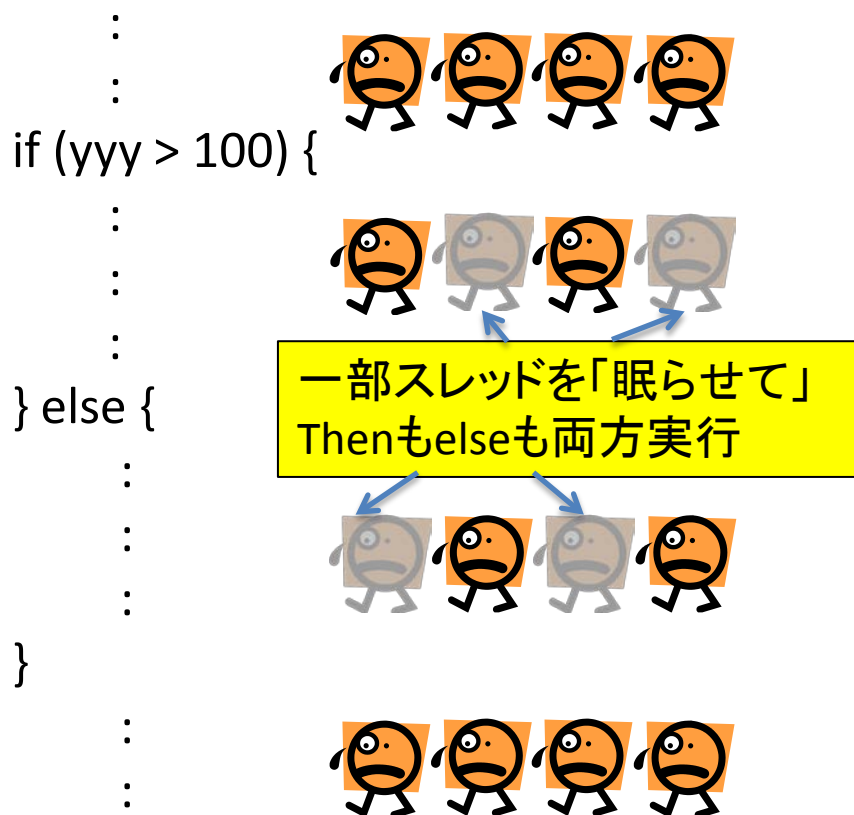
(a) スレッド達の意見がそろった場合

- 全員、 $xxx > 100$ だとする



(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



これを **divergent**
分岐と呼ぶ

Divergent分岐はなぜ非効率?

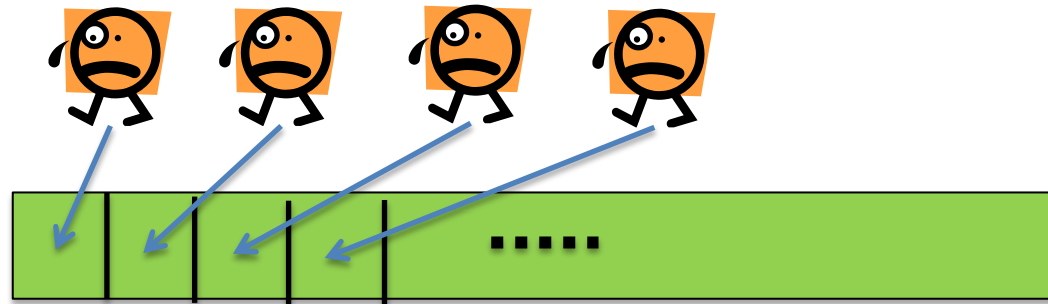
- CPUの常識では、if文はthen部分とelse部分の片方しか実行しないので、片方だけの実行時間がかかる
- Divergent分岐があると、then部分とelse部分の両方の時間がかかってしまう

5. 「コアレスド・アクセス」によるメモリアクセス効率化

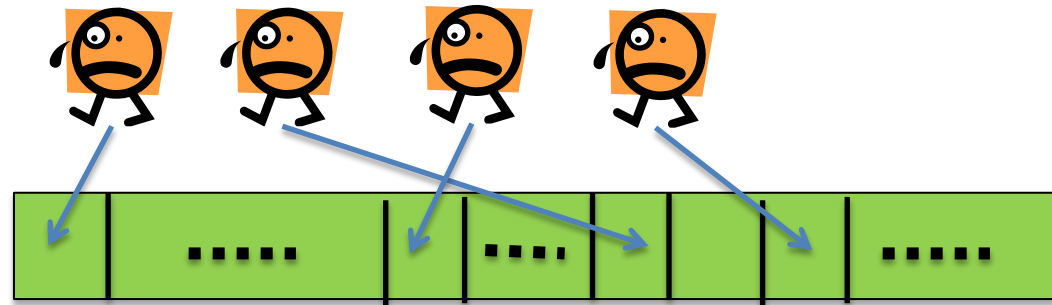
グローバルメモリのアクセスの効率化: コアレスド・アクセス

- 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

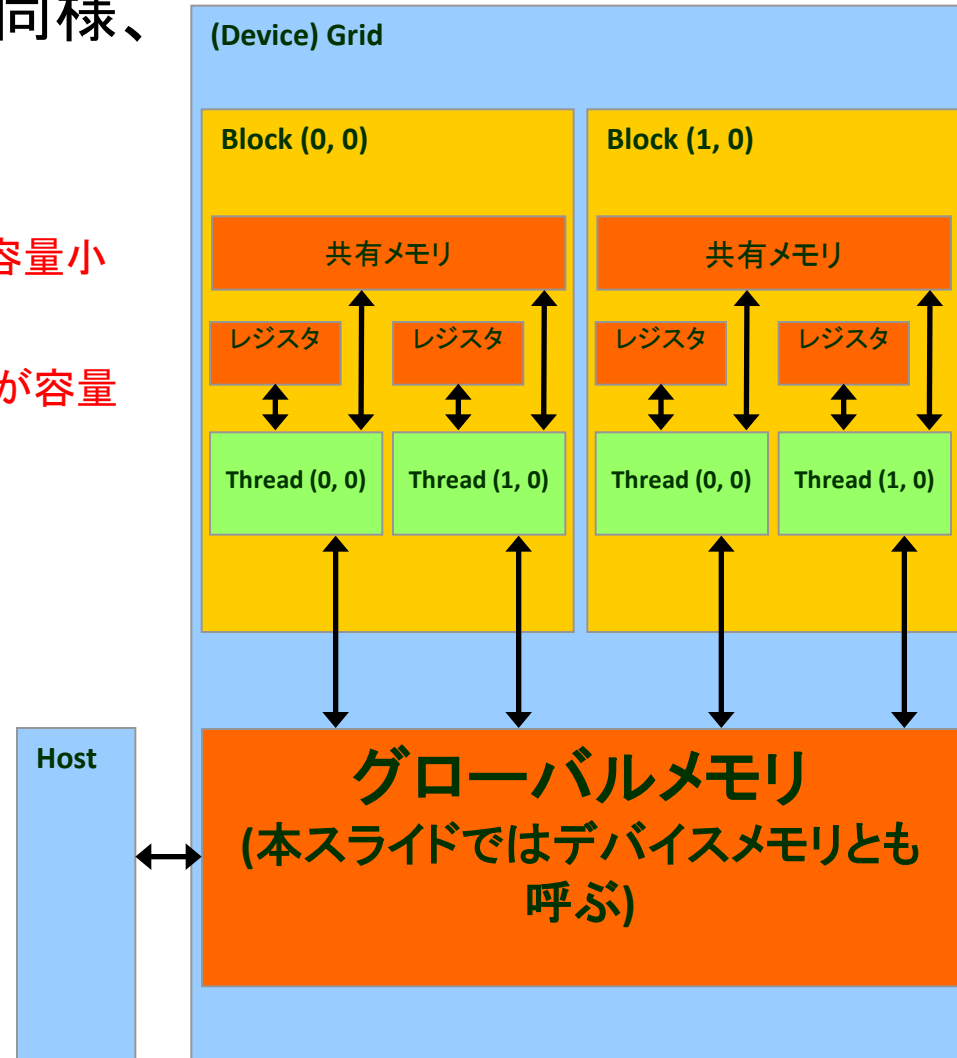
6. 「共有メモリ」の有効活用

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**

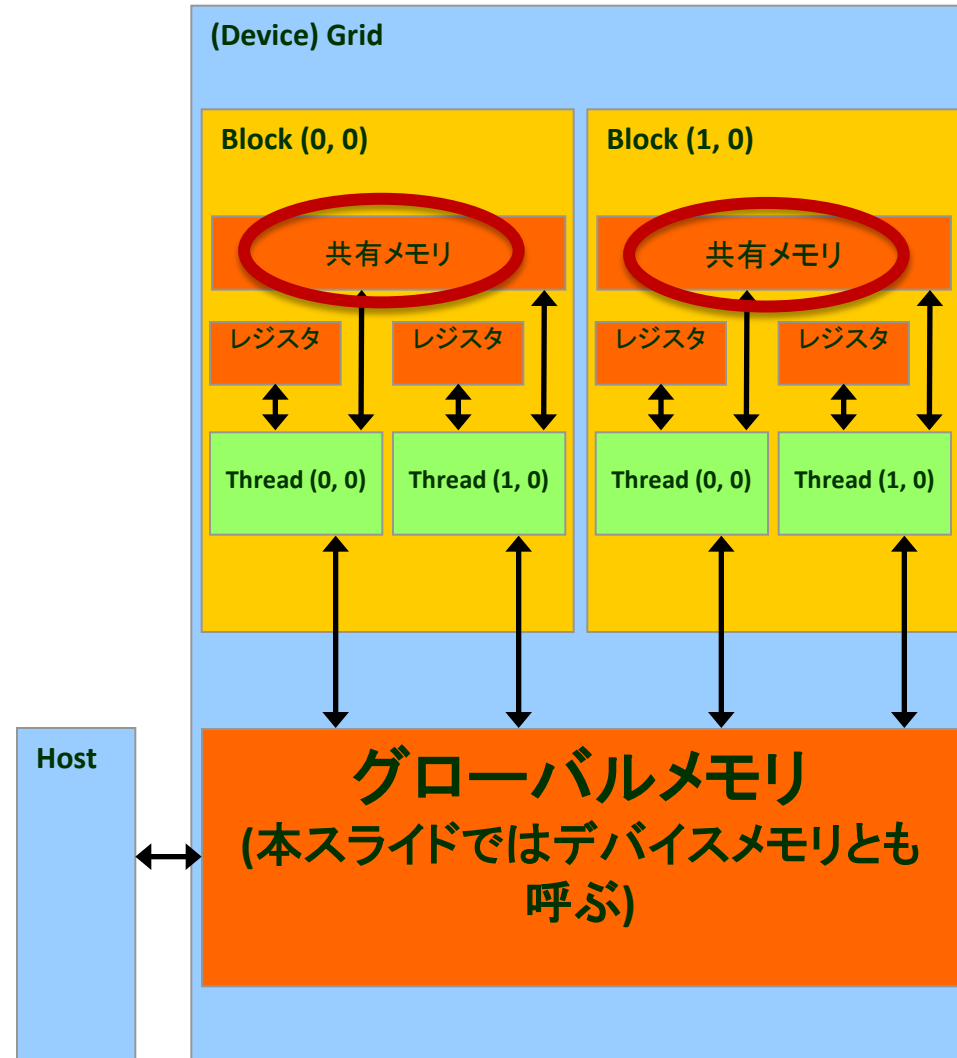
- スレッド固有
 - レジスタ → 局所変数を格納。高速だが容量小
- ブロック内共有
 - 共有メモリ → 本スライドで登場。高速だが容量小
 - (L1キャッシュ)
- グリッド内(全スレッド)共有
 - グローバルメモリ → `__global__` 変数や `cudaMalloc` で利用。容量大きい低速
 - (L2キャッシュ)

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**



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

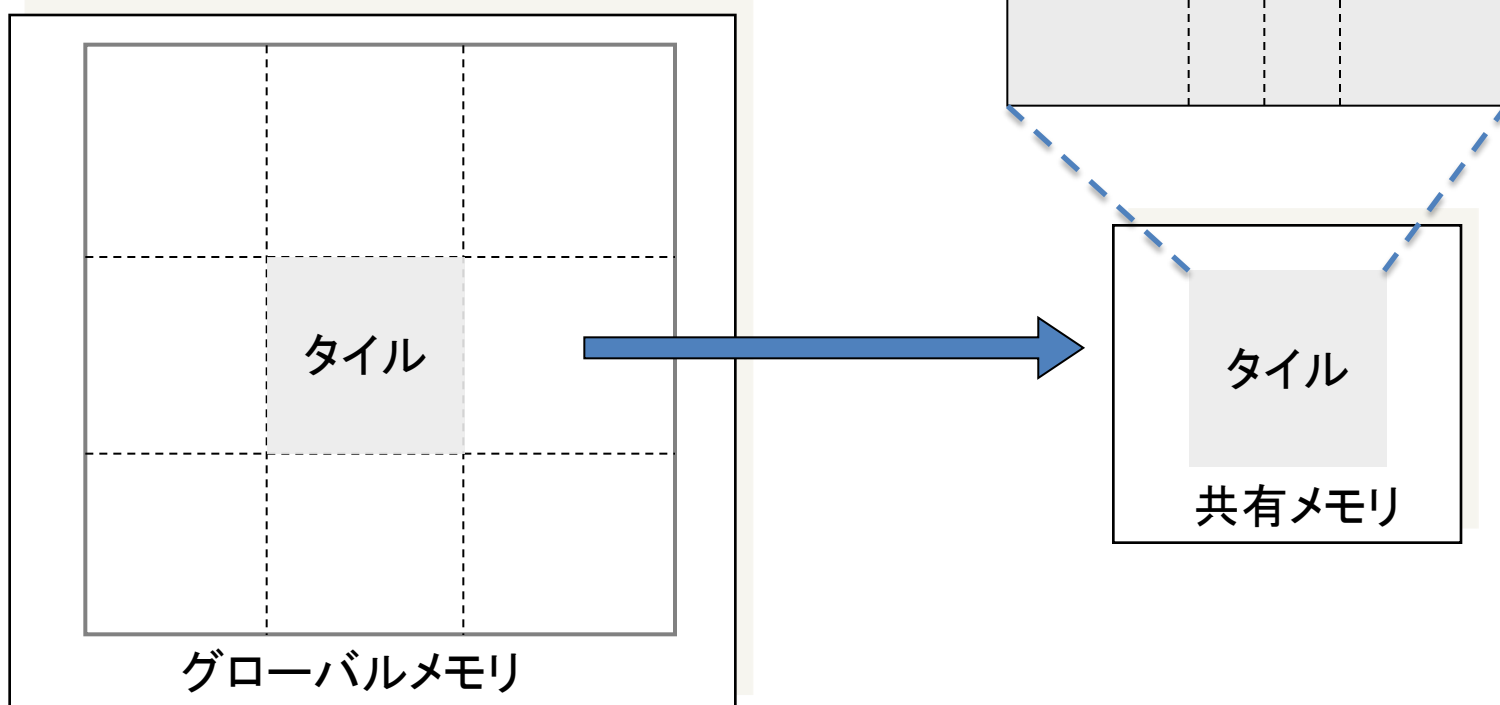
最適化前 (matmul_par)

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

- スレッド t_i, t_{i+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_{ij} 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、 $C=A \times B$ 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(!)

おわりに

GPUプログラムにおいて

- 基礎編の知識だけではちょっと不便な点の改良
- 何が起こってしまうと非効率的になってしまうか
- 何をできるだけ避けるべきか

について、いくつかポイントを説明した