GPUコンピューティング（CUDA)講習会

CUDAによるプログラミング基礎

丸山直也

はじめに

- 本講習会では時間の関係上ごくごく基礎的なことのみをとりあげます
- ただし、資料の後半にはメモリアクセスなどに関するチューニングに向けた情報をのせてあります。それらは講習時間内には取り上げません
- チューニングやよりアドバンストな内容の講習会は今後（基礎編の需要が一段落してから）予定しています

目次

1. CUDA概要
2. CUDAプログラム例
3. 実行
4. 並列化
5. 同期
6. 最適化
7. 参考資料

CUDAを実行可能なGPU

- NVIDIAによるG80系アーキテクチャ以降のGPU
  - 例： GeForce 8800 GTX (コアアーキテクチャG80), GeForce 285 GTX （コアアーキテクチャGT200), Tesla S1070 (TSUBAME）
- 以下のURLにCUDA対応GPU全リスト有り
  http://www.nvidia.com/object/cuda_learn_products.html

TSUBAMEのGPUスペック

- telsadebugキューにログイン

```
ssh –t user@login.cc.titech.ac.jp tesladebug
```

- 以下のようにdeviceQueryプログラムを実行

```
nmaruyam@tgg075054:~> /work/gpu/maruyama/deviceQuery
There are 4 devices supporting CUDA

Device 0: "Tesla T10 Processor"
  Major revision number:                         1
  Minor revision number:                         3
  Total amount of global memory:                 4294705152 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.44 GHz
  Concurrent copy and execution:                 Yes
```

GPUによる高速化手法

- BLAS/FFTライブラリを利用
  - CUDAプログラムを書く必要なし→手軽な高速化
  - 本講習の最後にCUBLAS/CUFFTの使い方を説明
- PGIによるGPGPU対応コンパイラを利用
  - 半自動CUDA化コンパイラ（like OpenMP)
  - 手軽、性能そこそこ
- CUDAでプログラミング（←本講習会の目的）
  - CUDA言語を覚える必要あり
  - 自由度最大、効果大

プログラミング言語としてのCUDA

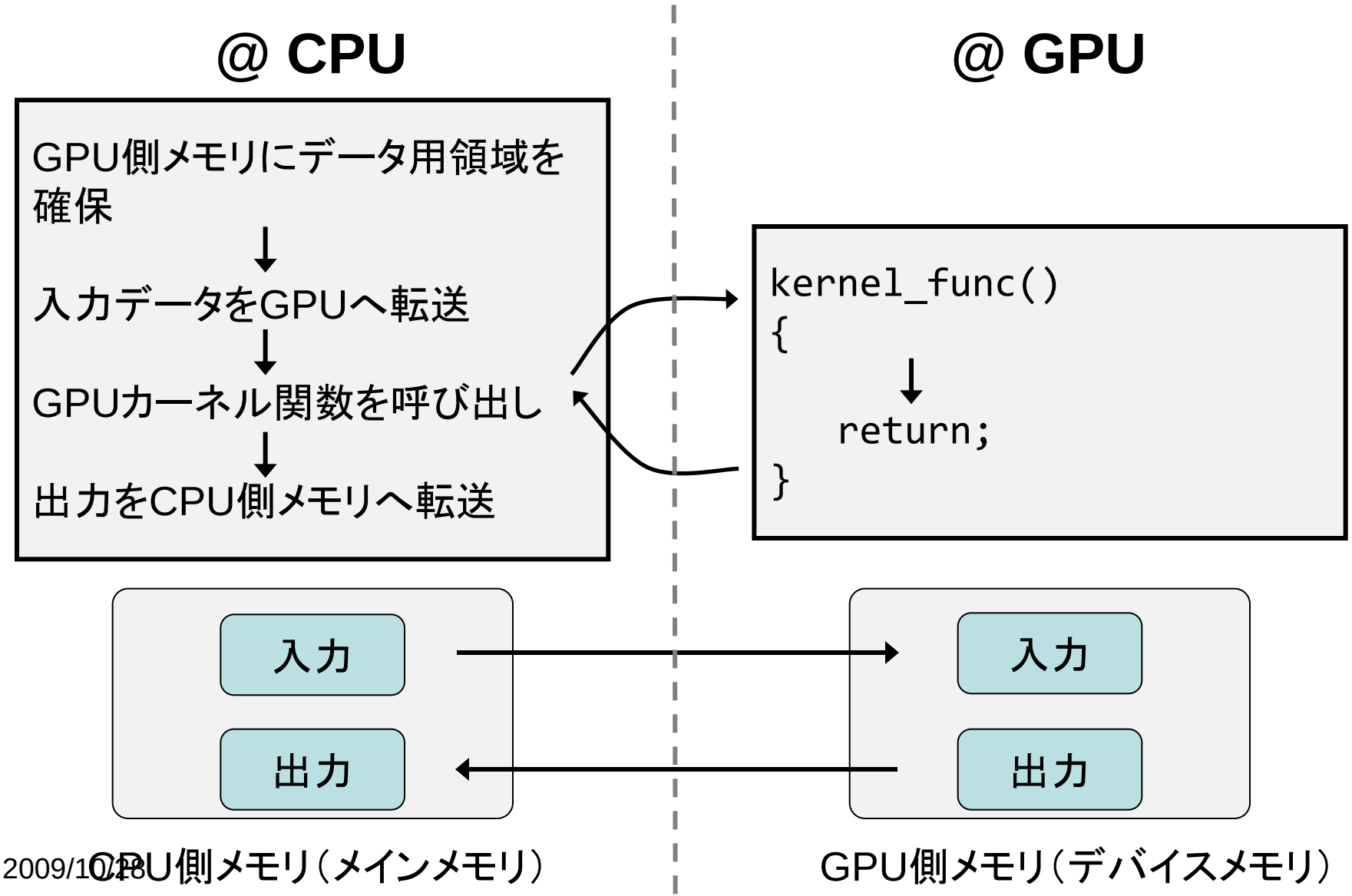
- MPIのようなSPMDプログラミングモデル
  - ただし一部SIMDのような制限有り
- 標準C言語サブセット＋GPGPU用拡張機能
  - 他言語からの利用は通常のCプログラム呼び出し方法により可能
- 2007年2月に最初のリリース、現在v2.3が最新リリース版
  - Tsubameではv2.2が利用可能
  - 今月中にv3.0 betaとの予定
- Windows、Linux, Mac OS X＋CUDA対応NVIDIA GPU の組み合わせで利用可能
- 現状のGPGPUで最も普及
  - Cf. Brook+, OpenCL, RapidMind, etc.

プログラム構成

ホストプログラム ＋ GPUカーネル関数

- ホストプログラム
  - CPU上で実行されるプログラム
  - ほぼ通常のC言語として実装
  - GPUに対してデータ転送、プログラム呼び出しを実行
- （GPU）カーネル関数
  - GPU上で実行されるプログラム
  - ホストプログラムから呼び出されて実行
  - 再帰、関数ポインタは非サポート

典型的な制御とデータの流れ



@CPU: GPU側メモリ領域確保

- `cudaMalloc(void **devptr, size_t count)`
  - GPU側メモリ(*デバイスメモリ*、*グローバルメモリ*と呼ばれる)に領域を確保
  - `devptr`: デバイスメモリアドレスへのポインタ。確保したメモリのアドレスが書き込まれる
  - `count`: 領域のサイズ
- `cudaFree(void *devptr)`
  - 指定領域を開放

例: 長さ1024のintの配列を確保

```
#define N (1024)
int *arrayD;
cudaMalloc((void **)&arrayD, sizeof(int) * N);
// arrayD has the address of allocated device memory
```

@CPU: 入力データ転送

- cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - 先にcudaMallocで確保した領域に指定したCPU側メモリのデータをコピー
  - dst: 転送先デバイスメモリ
  - src: 転送元CPUメモリ
  - kind: 転送タイプを指定する定数。ここではcudaMemcpyHostToDeviceを与える

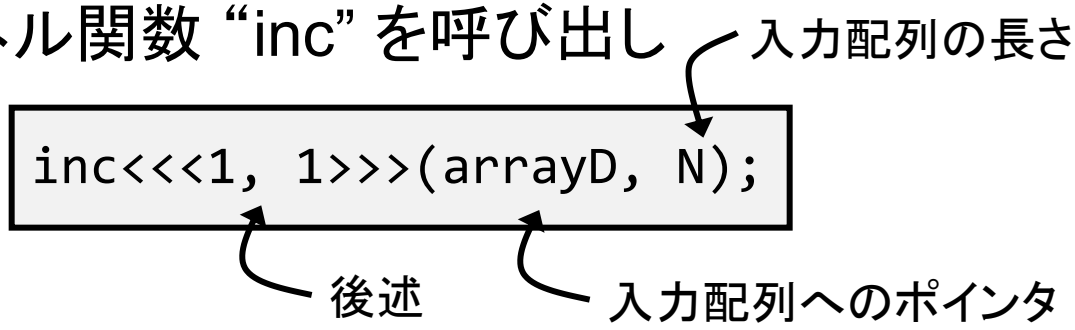
例: 先に確保した領域へCPU上のデータarrayHを転送

```
int arrayH[N];
cudaMemcpy(arrayD, arrayH, sizeof(int)*N,
           cudaMemcpyHostToDevice);
```

@CPU: GPUカーネルの呼び出し

- kernel_func<<<grid_dim, block_dim>>>(kernel_param1, …);
  - kernel_func: カーネル関数名
  - kernel_param: カーネル関数の引数

例： カーネル関数 “inc” を呼び出し



@GPU: カーネル関数

- GPU上で実行される関数
- GPU側メモリのみアクセス可、CPU側メモリはアクセス不可
- 引数利用可能、値の返却は不可

例： int型配列をインクリメントするカーネル関数

```
__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++) array[i]++;
}
```

@CPU: 結果の返却

- 入力転送と同様にcudaMemcpyを用いる
- ただし、転送タイプは cudaMemcpyDeviceToHost を指定

例： インクリメントされた配列をCPU側メモリへ転送

```
cudaMemcpy(arrayH, arrayD, sizeof(int)*N,
           cudaMemcpyDeviceToHost);
```

目次

1. CUDA概要
2. CUDAプログラム例
3. 実行
4. 並列化
5. 同期
6. 最適化
7. 参考資料

プログラム例： inc_seq

int型配列の全要素を1インクリメント

プログラムリスト: inc_seq.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define N  (32)
__global__ void inc(int *array, int len)
{
    int i;
    for (i = 0; i < len; i++)
        array[i]++;
    return;
}
int main(int argc, char *argv[])
{
    int i;
    int arrayH[N];
    int *arrayD;
    size_t array_size;
```

```
    for (i=0; i<N; i++) arrayH[i] = i;
    printf("input: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("¥n");

    array_size = sizeof(int) * N;
    cudaMalloc((void **)&arrayD, array_size);
    cudaMemcpy(arrayD, arrayH, array_size,
            cudaMemcpyHostToDevice);
    inc<<<1, 1>>>(arrayD, N);
    cudaMemcpy(arrayH, arrayD, array_size,
               cudaMemcpyDeviceToHost);
    printf("output: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("¥n");
    return 0;
}
```

プログラム例： 行列積 (1)

プログラムリスト: matmul_seq.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define L (1024)
#define M (1024)
#define N (1024)

__global__ void matmul(float *A, float *B, float *C,
                       int l, int m, int n)
{
   int i, j, k;
   for (i = 0; i < l; i++) {
       for (j = 0; j < n; j++) {
           float sum = 0.0;
           for (k = 0; k < m; k++) {
               sum += A[i * m + k] * B[k * n + j];
           }
           C[i*n+j] = sum;
       }
    }
}
```

プログラム例： 行列積 (2)

```
void alloc_matrix(float **m_h, float **m_d, int h, int w)
{
    *m_h = (float *)malloc(sizeof(float) * h * w);
    cudaMalloc((void **)m_d, sizeof(float) * h * w);
}
void init_matrix(float *m, int h, int w)
{
    int i, j;
    for (i = 0; i < h; i++)
        for (j = 0; j < w; j++)
            m[i * w + j] = (float)random();
}
```

プログラム例： 行列積 (3)

```
int main(int argc, char *argv[])
{
   float *Ad, *Bd, *Cd;
   float *Ah, *Bh, *Ch;

   // prepare matrix A
   alloc_matrix(&Ah, &Ad, L, M);
   init_matrix(Ah, L, M);
   cudaMemcpy(Ad, Ah, sizeof(float) * L * M,
              cudaMemcpyHostToDevice);
   // do it again for matrix B
   alloc_matrix(&Bh, &Bd, M, N);
   init_matrix(Bh, M, N);
   cudaMemcpy(Bd, Bh, sizeof(float) * M * N,
              cudaMemcpyHostToDevice);
   // allocate spaces for matrix C
   alloc_matrix(&Ch, &Cd, L, N);
```

プログラム例： 行列積 (4)

```
    // still in function main

    // launch matmul kernel
    matmul<<<1, 1>>>(Ad, Bd, Cd, L, M, N);

    // obtain the result
    cudaMemcpy(Ch, Cd, sizeof(float) * L * N,
               cudaMemcpyDeviceToHost);

    return 0;
}
```

開発＆コンパイル方法

- CUDAプログラムは慣例として .cu の拡張子を使用
- コンパイル、リンクにはCUDAツールキット付属の nvcc コマンドを利用
  - ツールキットなどはNVIDIAのCUDAサイトからフリーでダウンロード可能
  - TSUBAMEでは /opt/cuda/bin 以下にあり

```
$ nvcc test.cu -o test
$ ./test
```

- （参考）nvccの内部動作
  1. CUDAプログラムを通常のC++プログラム部とGPUアセンブリ部(PTX)へと分割＆変換
  2. C++コンパイラを呼び出し、C++プログラム部をコンパイルし、CUDAライブラリとリンクして実行ファイルを作成
  3. GPUアセンブリ部をGPUアセンブリ（ptxas）によってGPU機械語へコンパイル

Fortranからの利用

- 方法1：ホストコード＆カーネルをC言語の関数として記述し、Fortranから呼び出し
  - コンパイル方法

```
$ nvcc –c test_fortran.cu
$ gfortran –o test_fortran fortran_main.f
test_fortran.o –L/opt/cuda/lib -lcudart
```

- 方法2： PGIコンパイラ付属のCUDA for Fortran 開発キットを利用
  - すべてFortranで記述可能

TSUBAMEのTesla利用方法:ログイン

1. 端末(iMac)へのログイン
   - 配布した紙に記載されているID, passwordを利用
2. Titech2006もしくは「移動」ユーティリティを選択し、X11.appを起動（xtermの起動）
3. Tsubameへログイン

   > ssh –Y –t login名@login.cc.titech.ac.jp tesladebug

TSUBAMEのTesla利用方法：コンパイルとデバッグマシンでの実行

- 利用の手引き7節「CUDAツールキットによるプログラム作成」参照

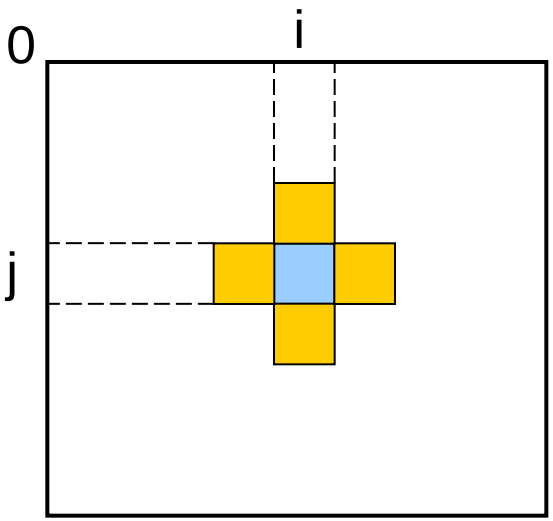
1. Tesla用インタラクティブノードへログイン
   - ssh –t user@login.cc.titech.ac.jp tesladebug
2. nvccを用いてコンパイル
3. コンパイルに成功した実行ファイルは同ノードでそのまま実行可能
   - ただしあくまで開発用ノードなので、長時間に渡るプログラムは実行しないでください
   - 本格的なプログラムは占有キューが利用してください

実習

- 先のサンプルプログラム inc_seq.cu をコンパイル、実行し、出力を確認
  - ソースコードはTSUBAME上の /work/GPU/maruyama 以下に有り
  - 手順
    - cd
    - cp /work/GPU/maruyama/inc/inc_seq.cu .
    - nvcc inc_sec.cu –o inc_sec
    - ./inc_sec
- inc_seq.cu をベースに、1次元配列の足し算を行うプログラムを書け
  - ただし配列サイズはinc_sec.cuと同じく32とする

実習：Point Jacobi

- 2次元Poisson方程式をPoint Jacobi法で計算
  - jacobi.c を参考にGPUを用いて計算するプログラムを作成（簡単のためにサイズ等は固定）



各要素を上下左右の要素を用いて更新

↓

一定回数繰り返したら終了

$$aveX_{i,j} = (oldX_{i+1,j} + oldX_{i-1,j} + oldX_{i,j+1} + oldX_{i,j-1}) / 4$$

$$newX_{i,j} = oldX_{i,j} + (aveX_{i,j} - oldX_{i,j}) \cdot \omega$$

CPU版Jacobiプログラム(jacobi.c)抜粋

```
void jacobi(float *region[2], int nx, int ny, float omega)
{
    int i, j, iter;
    /* buffer id */
    int cur = 0;
    int next = 1;
    for (iter = 0; iter < MAX_ITER; iter++) {
        for (j = 1; j < ny-1; j++) {
            for (i = 1; i < nx-1; i++) {
                float curv = region[cur][j*nx+i];
                /* compute new value from neighbor points */
                float nextv=(region[cur][(j-1)*nx+i]+region[cur][(j+1)*nx+i] +
                     region[cur][j*nx+(i-1)]+region[cur][j*nx+(i+1)]) * 0.25f;
                /* overcorrection */
                region[next][j*nx+i] = curv + (nextv - curv) * omega;
            }
        }
        /* switch buffer */
        cur = 1-cur; next = 1-next;
    }
    return;
}
```

Point JacobiのCUDA実装

- 手順
  1. ホスト上メモリに2次元配列を2つ確保し初期化（CPU版から変更なし）
  2. デバイスメモリに同サイズの2次元配列を2つ確保
  3. ホストメモリからデバイスメモリへデータ転送
  4. GPU上で一定回数（MAX_ITER)計算
  5. デバイスメモリからホストメモリへデータ転送

ここまでのまとめ

- C言語拡張のCUDAの概要
  - SPMDスタイルの並列性
- 典型的なCUDAプログラムのパターン
  - GPU上にデータ領域を確保 (cudaMalloc)
  - 確保したGPU上領域へデータを転送(cudaMemcpy)
  - カーネルを実行
  - 結果をCPU側メモリへ転送 (cudaMemcpy)
- 用語
  - ホスト、カーネル、デバイス、デバイスメモリ
- API（詳細はCUDAリファレンスマニュアルを参照）
  - cudaMalloc
  - cudaMemcpy

目次

1. CUDA概要
2. CUDAプログラム例
3. 実行
4. 並列化
5. 同期
6. 最適化
7. 参考資料

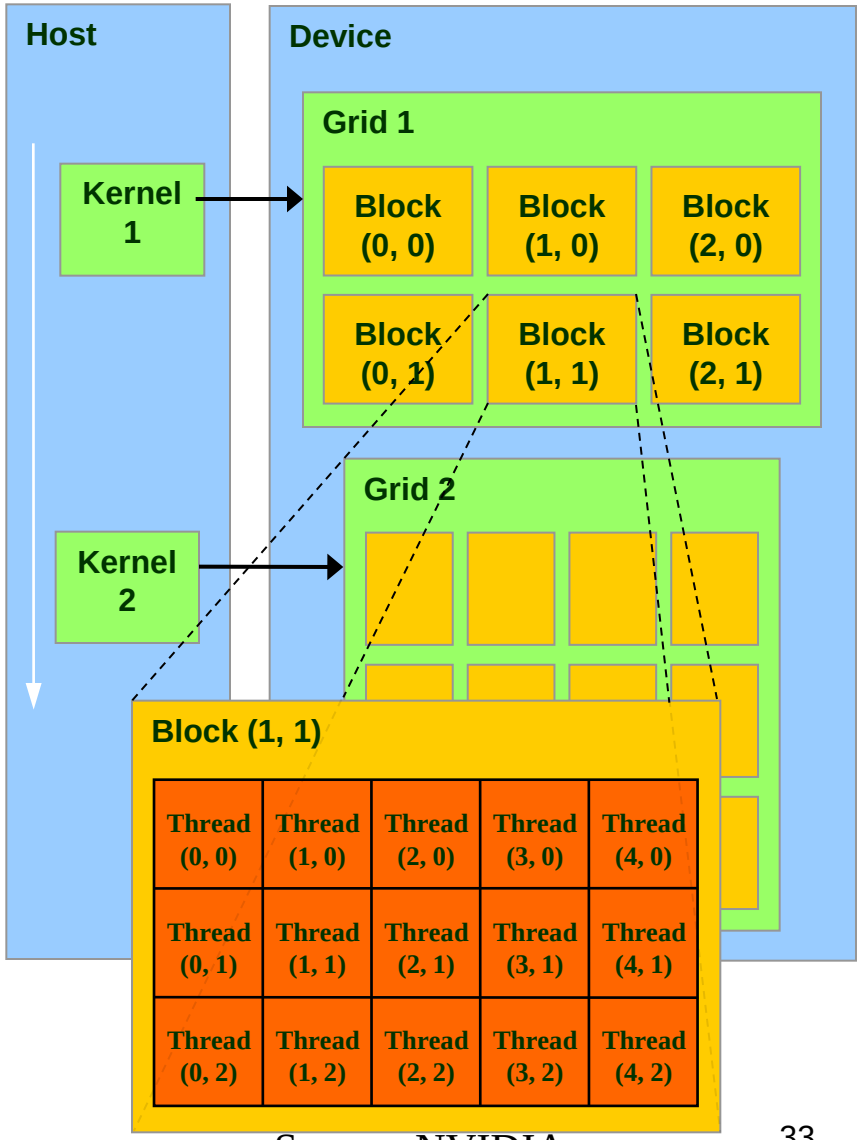
並列化

CUDAにおける並列化

- 軽量スレッドを用いたマルチスレッド並列化
  - 専用ハードウェアにより数千単位のスレッドの生成、スケジューリングを高速実行
  - 先のプログラムinc_sec.cuはGPU上で1スレッドのみで逐次に実行
- データレベル並列性を基にした並列化が一般的
  - 例：大規模配列に対して（ほぼ）同一の処理を適用→部分配列への処理に分割し複数スレッドを用いて並列実行

スレッド管理

- スレッド全体を階層的にまとめて管理
  - *スレッドブロック*
    - 指定した数からなるスレッドの集合
    - 3次元ベクトルでサイズを指定
  - *グリッド*
    - 全スレッドブロックからなる集合
    - 2次元ベクトルでサイズを指定
- スレッドID
  - スレッドのスレッドブロックと位置、スレッドブロックのグリッド内の位置より決定



Source: NVIDIA

CUDAのマルチスレッド実行

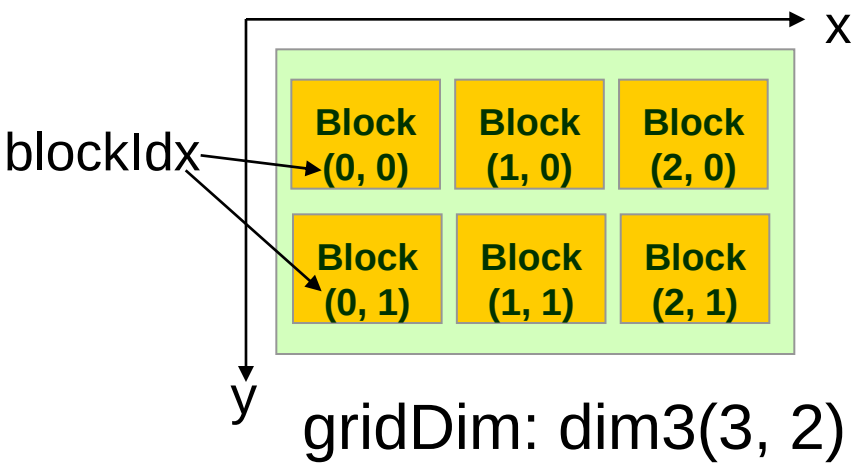
- 実行コンフィグ (Execution Configuration)
  - ホストプログラムからのカーネル呼び出し時に実行スレッド数を指定

```
<<<グリッドサイズ(dim3型またはint型),
                    ブロックサイズ(dim3またはint型)>>>
```

  - inc_sec.cuの“<<<1, 1>>>” ではグリッド、ブロックともにサイズ1を指定
- カーネルが指定されたスレッド数で実行
  - スレッド間同期、排他制御を一部サポート
- スレッドIDより各スレッドが計算する部分を決定

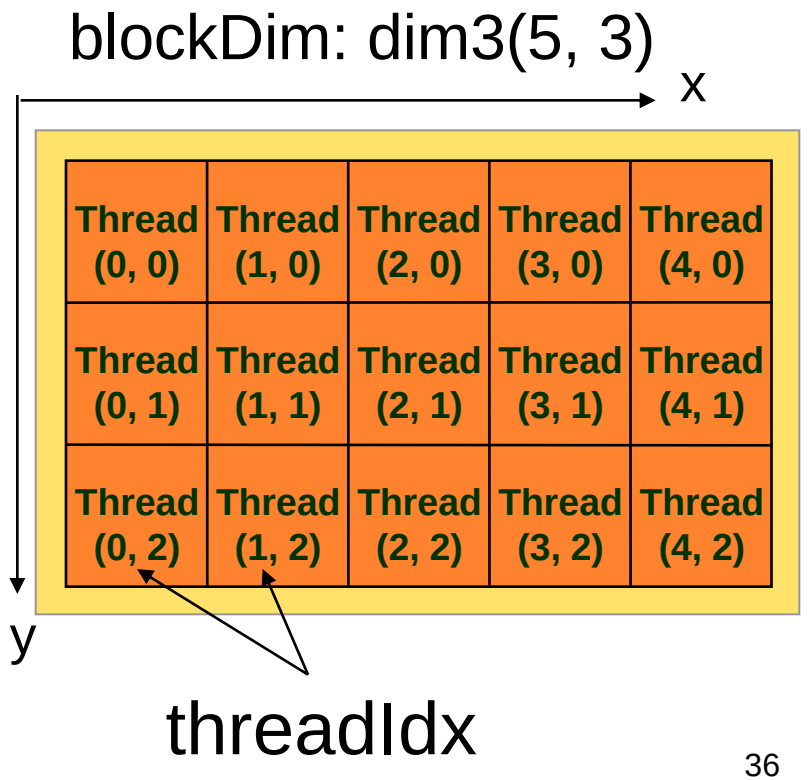
グリッド

- 1次元または2次元でサイズを指定可
- 整数もしくはdim3型を指定（整数の場合は1次元）
  - 以下はすべて等値: n, dim3(n, 1), dim3(n, 1, 1)
- カーネル関数から参照可能な組み込み変数
  - dim3 gridDim
    - グリッドサイズ
  - dim3 blockIdx
    - グリッド内のブロックのインデックス(オフセット）
- 最大サイズ（TSUBAME)
  - 65535 x 65535



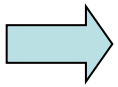
スレッドブロック

- 1次元、2次元、3次元で指定可
- カーネル関数から参照可能な組み込み変数
  - dim3 blockDim
    - ブロックサイズ
  - dim3 threadIdx
    - ブロック内のスレッドのインデックス（オフセット）
- 最大サイズの制限有り
  - TSUBAME では、各次元 512 x 512 x 64
  - 全体で512

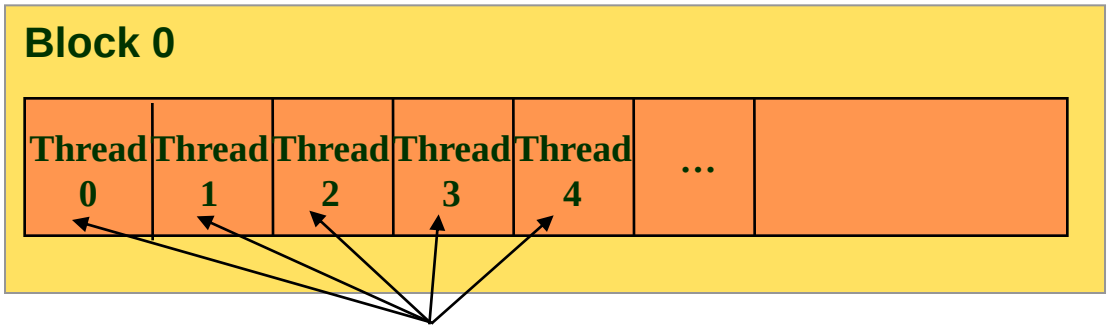


例：Nスレッドを生成

- 1ブロック、nスレッド生成
  - グリッドサイズ→ 1
  - ブロックサイズ → n



```
<<<1, N>>>
```



threadIdx.x

- ただし、ブロックあたりのスレッド数に制限有り
  - 仕様上 & ハードウェアリソース上

incの並列化：バージョン1

- 並列化方針
  - 入力1次元配列をスレッドで分割
  - 簡単化のためにスレッドブロックは1つ
- ホストプログラム
  - カーネル呼び出し時に実行スレッド構成を指定
  - 32スレッドの場合

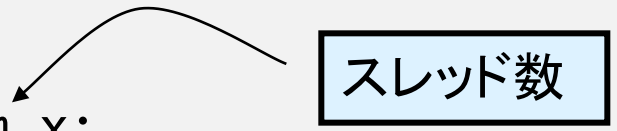
並列版inc (inc_par.cu)

```
inc<<<1, 32>>>(arrayD, N);
```

incの並列化：バージョン1 (2)

- カーネル関数
  - スレッドインデックスを基に各スレッドのパートを決定

```
__global__ void inc(int *array, int len)
{
    int i;
    int tid = threadIdx.x;
    int nthreads = blockDim.x;
    // assumes len is a multiple of nthreads
    int part = len / nthreads;
    for (i = part*tid; i < part*(tid+1); i++)
        array[i]++;
}
```



incの並列化: バージョン2

- バージョン1では単純化のために、スレッドブロックは1つのみ起動
  - 効率はよろしくない
  - TSUBAMEでは最低30（SMの数）起動すべき
- ホストプログラム
  - 30ブロック、32スレッド起動

```
inc<<<30, 32>>>(arrayD, N);
```

incの並列化: バージョン2（2）

- カーネル関数
  - バージョン1と同様にスレッドインデックスを元に各スレッドの担当パートを決定
  - ただし、バージョン1の処理に加えてブロックインデックスを考慮する必要あり

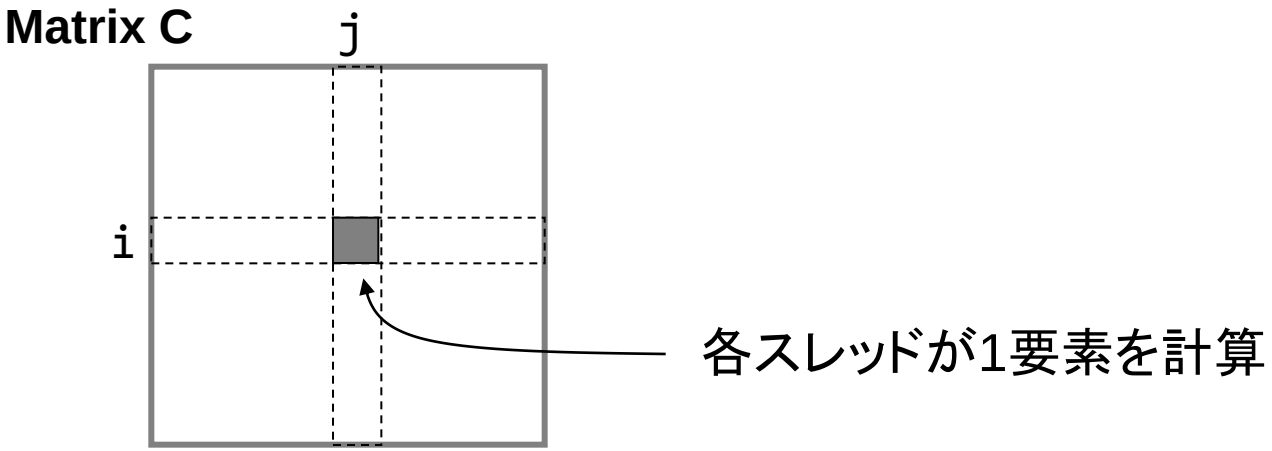
```
__global__ void inc(int *array, int len)
{
    int i;
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    int nthreads = blockDim.x * gridDim.x;
    // assumes len is a multiple of nthreads
    int part = len / nthreads;
    for (i = part*tid; i < part*(tid+1); i++)
        array[i]++;
}
```

実習

- 複数ブロックを用いたincを作成せよ
- 先の実習で作成したベクトルの足し算をカーネルを並列化せよ

並列化その２：行列積

- 方針
  - 結果の行列Cの各要素の計算はデータ並列
    →それぞれ別個のスレッドで計算し並列化
  - 行列は2次元→スレッドを2次元行列にマッピング



並列行列積バージョン1

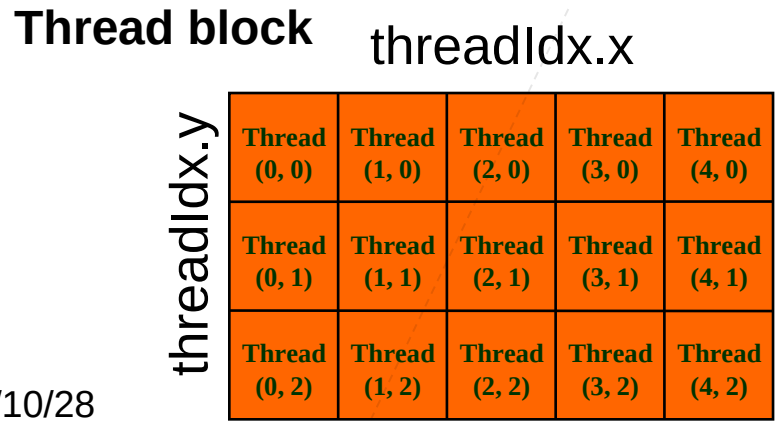
- スレッドの構成
  - 2次元のスレッドブロックにスレッドを割り当て
  - 1スレッドが行列Cの1要素を計算
    - Cの要素 (threadIdx.x, threadIdx.y) を計算
  - 単純化のためにブロックは1つのみ使用(→バージョン2で拡張）
  - 例: l = m = 16の場合

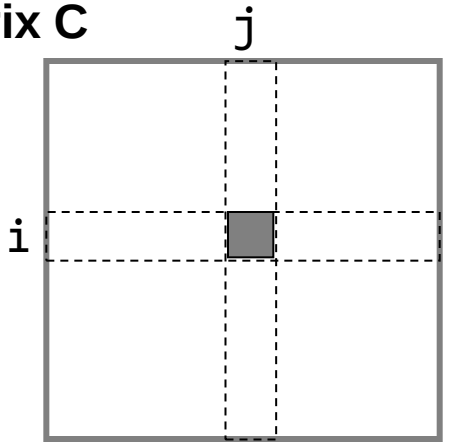
```
matmul<<<1, dim3(16,16)>>>(Ad, Bd, Cd, L, M, N);
```

- カーネルの構成
  - 各カーネルは内積を1回のみ計算

並列行列積：バージョン1

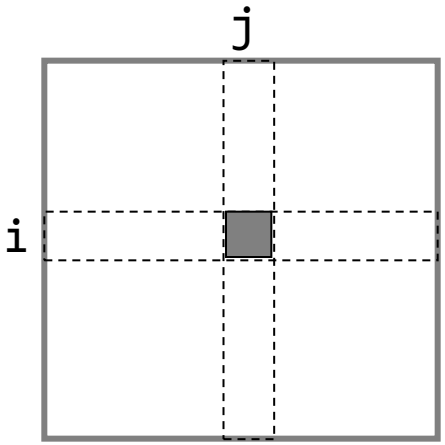
- 逐次版からの変更点
  - カーネル呼び出し（ヒントの通り）
  - カーネル関数
    - 行列CのLxN要素をLxNスレッドで等分割
    - 各スレッドが行列Cの1要素（C[i][j]）のみを計算
    - スレッドの計算対象要素→スレッドブロック内の位置
      - i: threadIdx.y, j: threadIdx.x





並列行列積:バージョン1

プログラムリスト: matmul_par.cu




```
__global__ void matmul(float *A, float *B,
                       float *C, int l,
                       int m, int n)
{
    int i, j, k;
    i = threadIdx.y;
    j = threadIdx.x;
    float sum = 0.0;
    for (k = 0; k < m; k++) {
        sum += A[i*m+k] * B[k*n+j];
    }
    C[i*n+j] = sum;
}
```

並列行列積: バージョン2
より大きなサイズの行列への対応

- 初めの設計
  - 16x16のスレッドブロック1つを立ち上げ
  - 各スレッドが内積を1要素分計算
- 16x16 以上のサイズの行列は？

→ スレッドブロックを大きくすれば良い？

→ No!　1ブロックにつき最大スレッド数は512 (Tesla T10)　(32x32→1024スレッド必要）

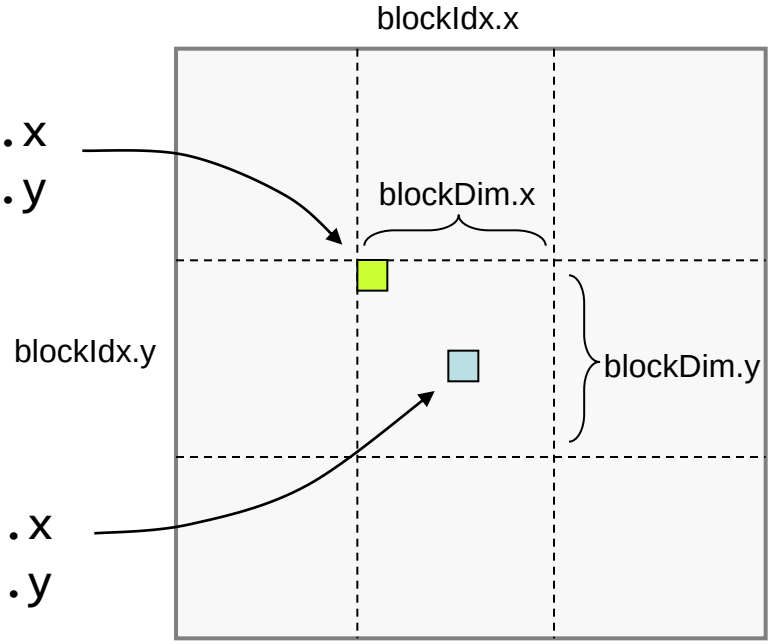
→ 複数ブロックを使うことで対応

並列行列積：バージョン2

- 各スレッドは今回も行列Cの1要素の内積を計算
- 16x16のスレッドブロックを複数立ち上げ
- 各スレッドブロックが行列Cの部分行列を担当

先頭要素からのオフセット

```
x: blockIdx.x * blockDim.x
y: blockIdx.y * blockDim.y
```



先頭要素からのオフセット

```
x: blockIdx.x * blockDim.x + threadIdx.x
y: blockIdx.y * blockDim.y + threadIdx.y
```

並列行列積：バージョン2プログラムリスト（1）

プログラムリスト: matmul_mb.cu より抜粋

```
#define BLOCKSIZE (16)

#define L (BLOCKSIZE * 16)
#define M (BLOCKSIZE * 16)
#define N (BLOCKSIZE * 16)

__global__ void matmul(float *A, float *B, float *C,
                       int l, int m, int n)
{
    int i, j, k;
    float sum;

    i = blockIdx.y * blockDim.y + threadIdx.y;
    j = blockIdx.x * blockDim.x + threadIdx.x;

    sum = 0.0;
    for (k = 0; k < m; k++) {
        sum += A[i * m + k] * B[k * n + j];
    }
    C[i*n+j] = sum;
}
```

16x16スレッド数のブロックを立ち上げ

縦横16倍の行列を計算
→ 16x16ブロック数のグリッドを立ち上げ

複数ブロックへの対応

並列行列積：バージョン2プログラムリスト（2）

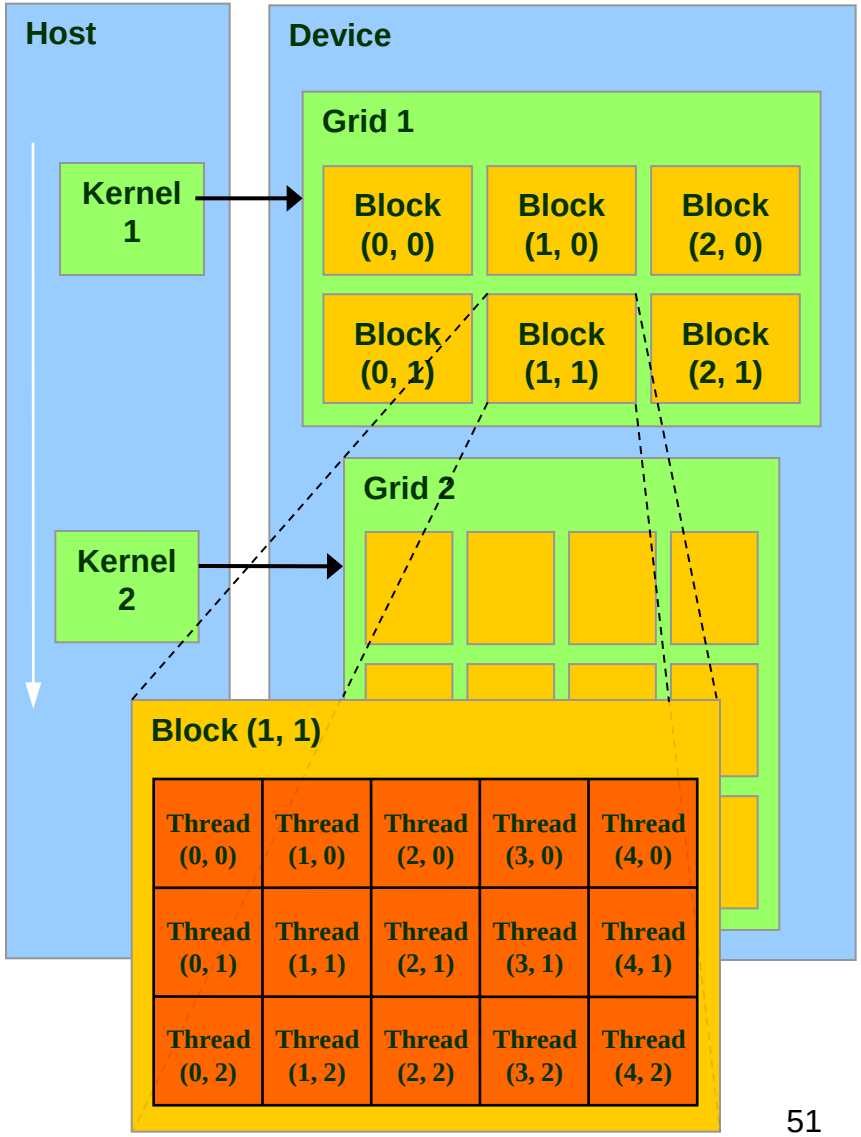
プログラムリスト: matmul_mb.cu より抜粋

```
int main(int argc, char *argv[])
{
    float *Ad, *Bd, *Cd;
    float *Ah, *Bh, *Ch;
    struct timeval t1, t2;

    // prepare matrix A
    alloc_matrix(&Ah, &Ad, L, M);
    init_matrix(Ah, L, M);
    cudaMemcpy(Ad, Ah, sizeof(float) * L * M,
               cudaMemcpyHostToDevice);
    // do it again for matrix B
    alloc_matrix(&Bh, &Bd, M, N);
    init_matrix(Bh, M, N);
    cudaMemcpy(Bd, Bh, sizeof(float) * M * N,
               cudaMemcpyHostToDevice);
    // allocate spaces for matrix C
    alloc_matrix(&Ch, &Cd, L, N);

    // launch matmul kernel
    matmul<<<dim3(N / BLOCKSIZE, L / BLOCKSIZE),
            dim3(BLOCKSIZE, BLOCKSIZE)>>>(Ad, Bd, Cd, L, M, N);

    …
    return 0;
}
```

複数ブロックの立ち上げ

ここまでのまとめ

- 階層化されたスレッド構成を用いたマルチスレッド並列化
  - スレッドブロック
  - グリッド



同期

Point Jacobiの並列化

- 基本方針
  - 各スレッドが行列1要素の更新を担当

```
void jacobi(float *region[2], int nx, int ny, float omega)
{
    /* 省略 */
    i = blockIdx.x * blockDim.x + threadIdx.x;
    j = blockIdx.y * blockDim.y + threadIdx.y;
    for (iter = 0; iter < MAX_ITER; iter++) {
        for (j = 1; j < ny-1; j++) {
            for (i = 1; i < nx-1; i++) {
                float curv = region[cur][j*nx+i];
                /* compute new value from neighbor points */
                /* 省略 */
                /* overcorrection */
                /* 省略 */
            }
        }
        /* switch buffer */
        cur = 1-cur; next = 1-next;
    }
    return;
}
```

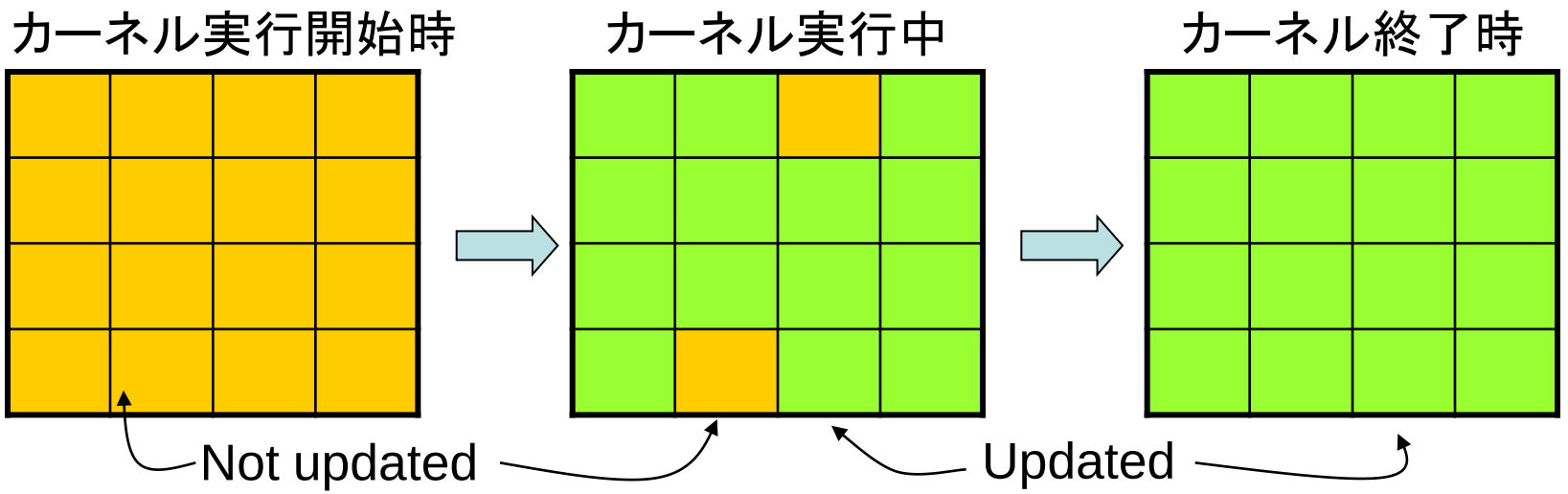
担当要素のインデックス

スレッド並列化によりループは不要

正しくない！

同期

- グローバルメモリを複数スレッドから更新→スレッド間の適切な同期が必要(グローバル同期)
- カーネル実行中のグローバルメモリの同期不可
  - 同期のためにはカーネルを一旦終了させる必要有り
  - （ただし、CUDA2.2から __threadfence 拡張命令が利用可能）



Point Jacobiの並列化

```
__global__ void jacobi(float *region[2], int nx, int ny, float omega)
{
    /* 省略 */
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.j;
    float curv = region[0][j*nx+i];
    /* 省略 */
    region[1][j*nx+i] = onextv;
    return;
}

int main(int argc, char *argv[])
{
    /* 省略 */
    for (iter = 0; iter < max_iter; iter++) {
        jacobi<<<dim3(16, 16), dim3(16, 16)>>>(region, nx, ny, omega);
        float *tmp = region[1];
        region[1] = region[0];
        region[0] = tmp;
    }
    /* 省略 */
}
```

カーネル関数ではイテレーション1回のみ計算

実習

- Point Jacobiの並列化を完成させよ

最適化

最適化基本方針

- メモリアクセス効率化
  - 共有メモリの利用
  - 連続領域へのアクセスによるメモリアクセスの一括処理
  - 共有メモリへのバンクコンフリクトの削減
- 計算処理効率化
  - “diverge”分岐の削除による分岐処理効率化
- ホスト・デバイス間データ転送

- ハードウェアの詳細を（それなりに）知る必要有り
- ただし、最適化による効果も大きい

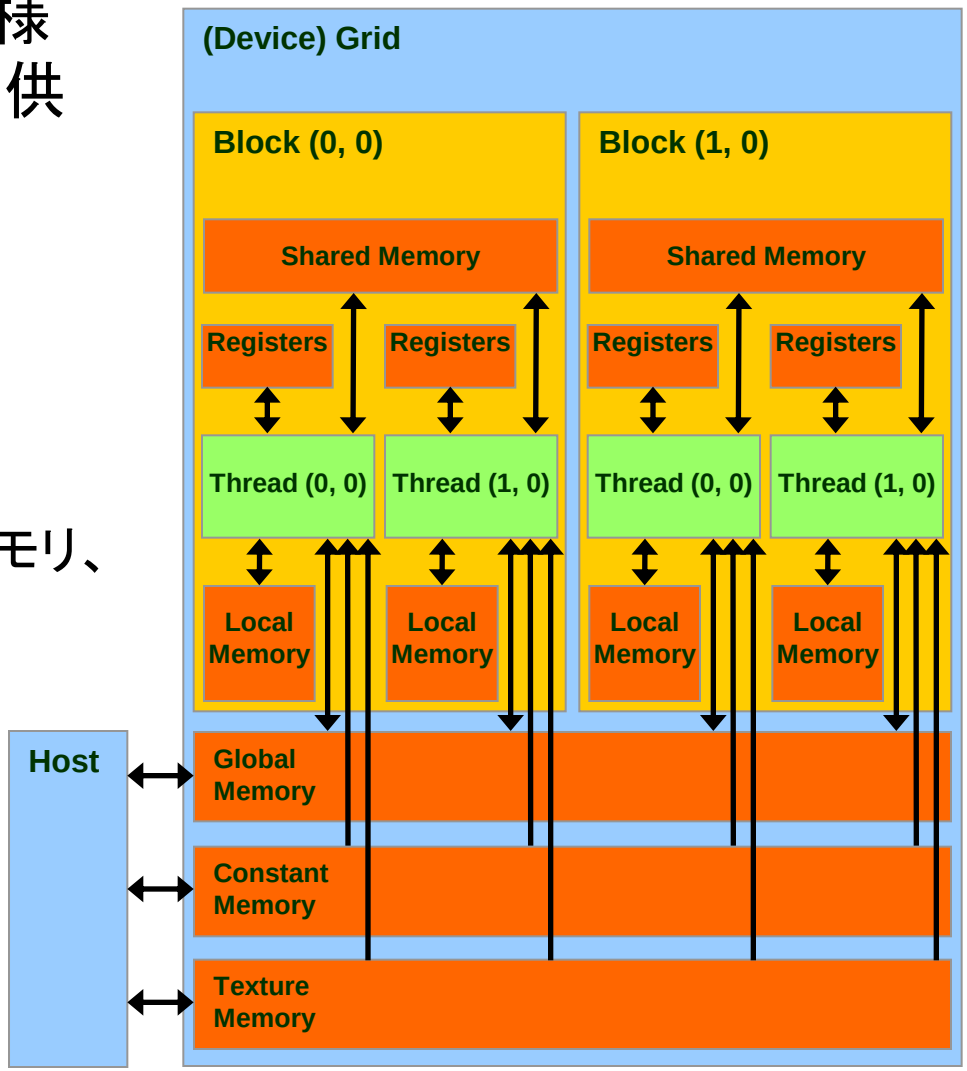
共有メモリの利用

CUDAメモリモデル

階層化スレッドグルーピングと同様に**階層化されたメモリモデル**を提供

- スレッド固有
  - レジスタ、ローカルメモリ
- ブロック内共有
  - 共有メモリ
- グリッド内（全スレッド）共有
  - グローバルメモリ、コンスタントメモリ、テクスチャメモリ
- ないもの
  - スタック

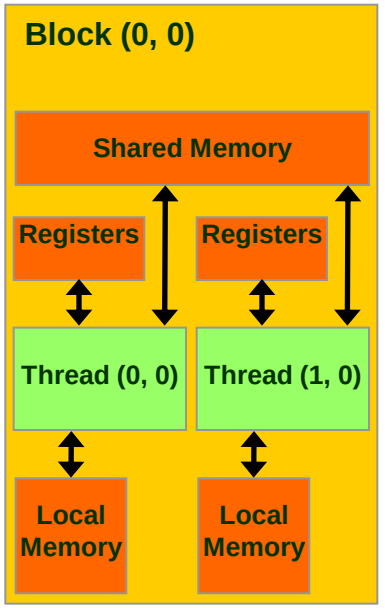
それぞれ速度と容量にトレードオフ有
（高速＆小容量 vs. 低速＆大容量）
→ メモリアクセスの局所性が重要



Source: Kirk and Hwu, ECE 498AL, UIUC

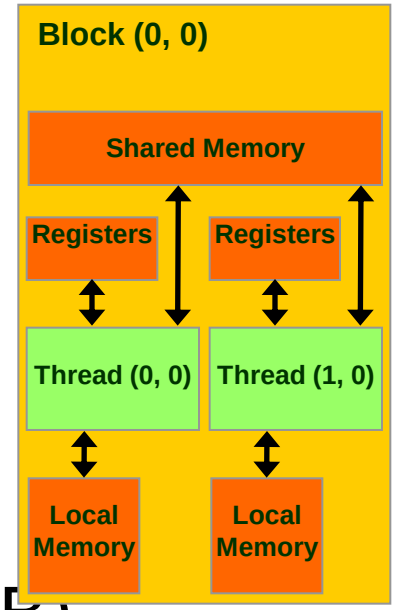
スレッド固有メモリ

- レジスタ
  - GPUチップ内に実装（i.e., オンチップメモリ）
  - カーネル関数のローカル変数を保持
  - 高速（遅延無しで計算ユニットより利用可）
  - T10ではブロックあたり16384本
  - スレッドでレジスタ領域を等分割して利用
- ローカルメモリ
  - GPUチップ外のデバイスメモリに配置（i.e., オフチップメモリ）
  - レジスタへ一度ロードしてからのみ利用可能
  - 主にローカル変数の退避領域として利用
  - 非常に低速（400-600サイクル）



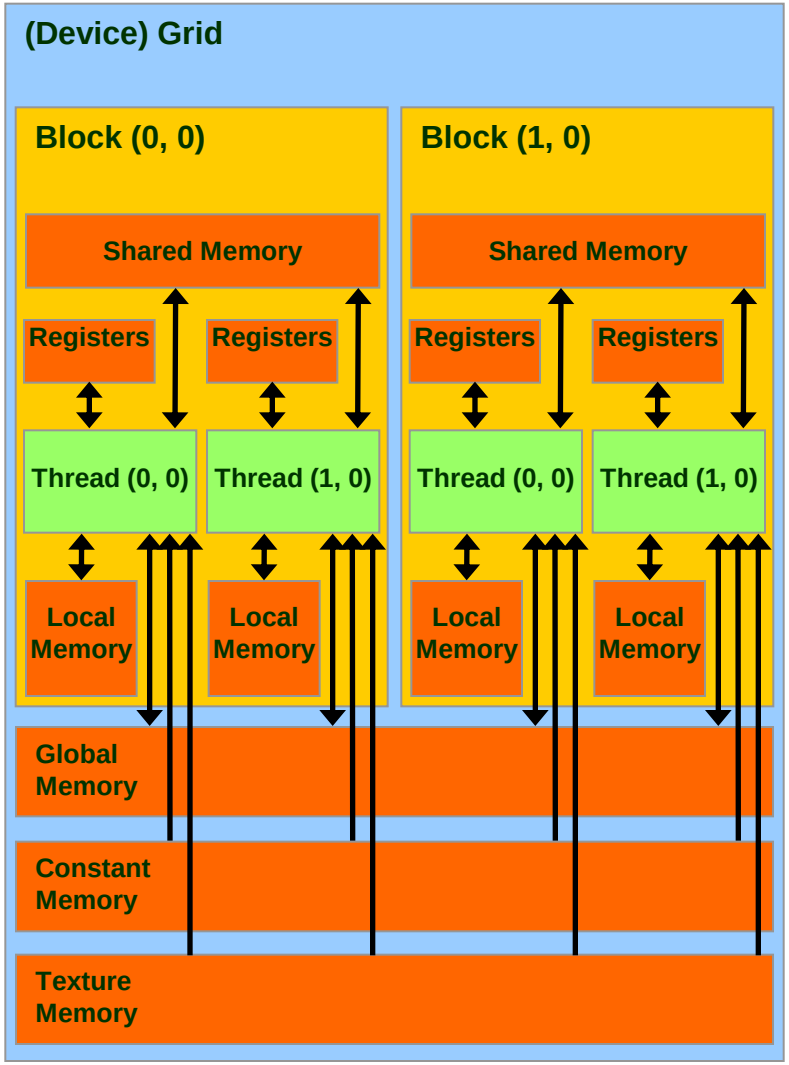
ブロック内共有メモリ

- 共有メモリ（shared memory)
  - ブロック内スレッドのみで「共有」
  - スレッド全体で共有されるわけではない
  - オンチップメモリ
  - レジスタと同様に高速
  - T10ではブロックあたり16KB（計480KB)



グリッド内（全スレッド）共有メモリ

- GPUチップ外に実装（オフチップ）
- グローバルメモリ
  - T10で4GB
  - 低速（400-600サイクル）
- コンスタントメモリ
  - ホスト側からのみ読み書き可能
  - カーネル側からは読み込みのみ可能
  - この授業では扱わない
- テクスチャメモリ
  - この授業では扱わない



グローバルメモリアクセスの最適化

- グローバルメモリへのアクセス
  - 例： incにおける配列アクセス、matmulにおける行列アクセス
  - 現世代までのGPUではハードウェアキャッシュ無し
  - 次世代GPU（Fermi）からはL1/L2データキャッシュ有り
  - CUDAプログラムにおける最も大きなボトルネックのひとつ
- 最適化： オンチップメモリをキャッシュとして活用 (*Software-managed cache*)
  - プログラムの局所性を特定し、オンチップメモリをプログラマが明示的にキャッシュとして活用
    → グローバルメモリへのアクセスを削減

CUDAにおける局所性

- 時間的局所性
  - 同一スレッドが同一データに複数回アクセス
  - 例： 初回にオンチップ領域に読み込み、オンチップ領域を用いて計算、最後にグローバルメモリへ書き込み
  - レジスタを利用
- スレッド間局所性
  - 異なるスレッド間で同じデータへアクセス
  - 例： あるスレッドが読み込んだデータを他のスレッドからも利用
  - スレッド間で共有可能なオンチップメモリを利用 → 共有メモリ

共有メモリによる最適化

- スレッドブロック内スレッドで共有可能
- 典型的な利用パターン
  1. 各スレッドがグローバルメモリよりデータを読み込み
  2. スレッドブロック内スレッドで同期をとり、読み込みを完了
     - `__syncthreads` 組み込み関数を使用
  3. 各スレッドが自身で読み込んだデータと他のスレッドが読み込んだデータを使って計算

共有メモリの同期

- スレッドブロック内の同期
  - __syncthreads 拡張命令を利用
  - 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

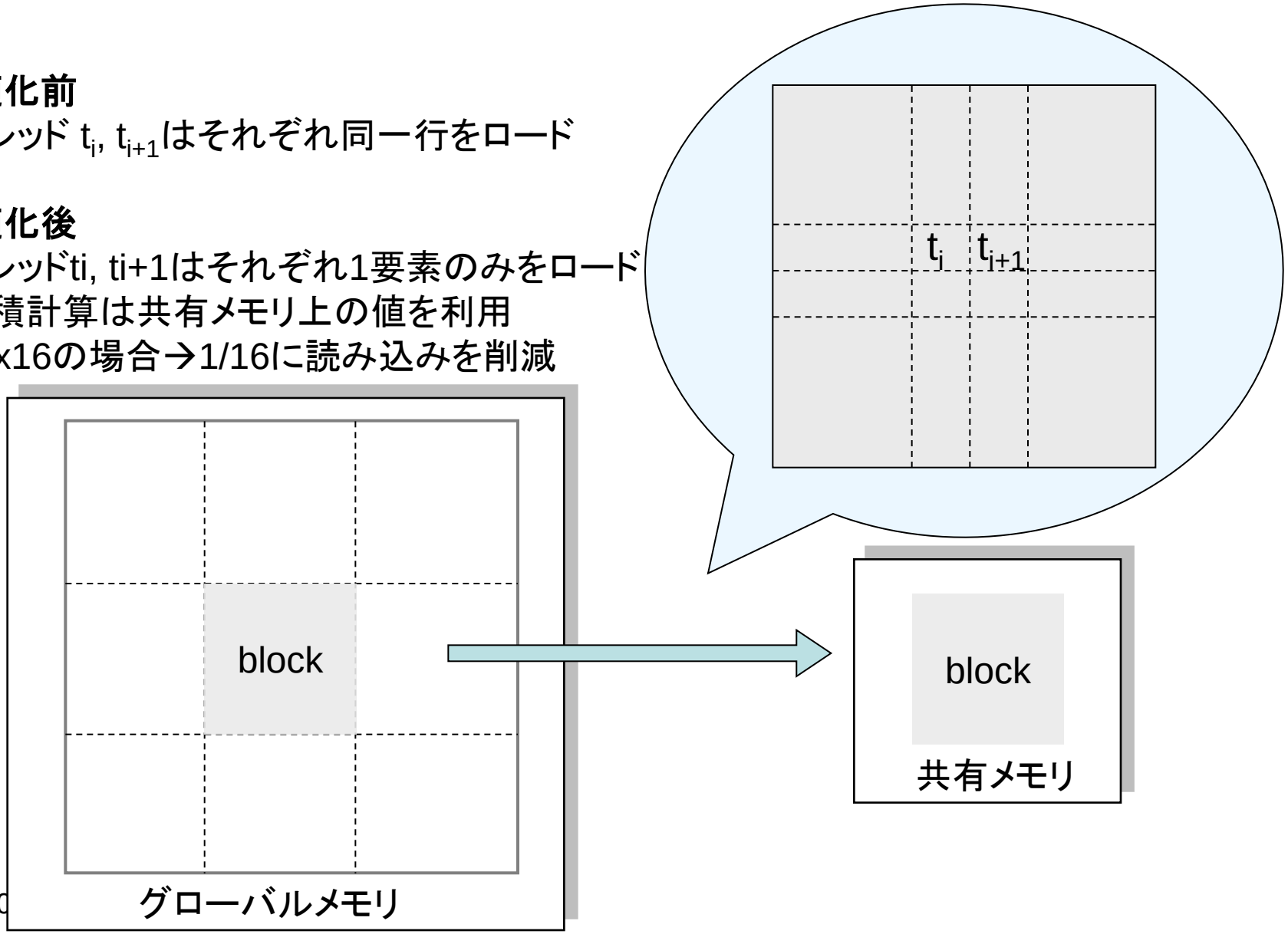
共有メモリを用いた行列積の最適化

- タイリング
  1. 行列A、B共に共有メモリに収まるサイズの部分行列（タイル）を共有メモリに読み込み
  2. 共有メモリを用いて部分行列のかけ算
  3. 次のタイルの積を計算
  4. 繰り返し

ブロックの読み込み

最適化前
- 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
- スレッドti, ti+1はそれぞれ1要素のみをロード
- 内積計算は共有メモリ上の値を利用
- 16x16の場合→1/16に読み込みを削減



行列積(共有メモリ版)

CUDA Programming Guide, Chapter 6より

```
__global__ void Muld(float* A, float* B,
                     int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A
    // processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A
    // processed by the block
    int aEnd = aBegin + wA - 1;
```

行列積(共有メモリ版)

```
// Step size used to iterate through
// the sub-matrices of A
int aStep = BLOCK_SIZE;
// Index of the first sub-matrix of B
// processed by the block
int bBegin = BLOCK_SIZE * bx;
// Step size used to iterate through the
// sub-matrices of B
int bStep = BLOCK_SIZE * wB;
// The element of the block sub-matrix
// that is computed by the thread
float Csub = 0;
```

行列積(共有メモリ版)

```
// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin; a <= aEnd;
     a += aStep, b += bStep) {
   // Shared memory for the sub-matrix of A
   __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
   // Shared memory for the sub-matrix of B
   __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
   // Load the matrices from global memory to
   // shared memory;

   // each thread loads one element of each matrix
   As[ty][tx] = A[a + wA * ty + tx];
   Bs[ty][tx] = B[b + wB * ty + tx];
   // Synchronize to make sure the matrices are loaded
   __syncthreads();
```

行列積(共有メモリ版)

```
// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
   Csub += As[ty][k] * Bs[k][tx];
   // Synchronize to make sure that the preceding
   // computation is done before loading two new
   // sub-matrices of A and B in the next iteration
   __syncthreads();
}

// Write the block sub-matrix to global memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}
```

グローバルメモリアクセスの一括処理（コアレッシング）

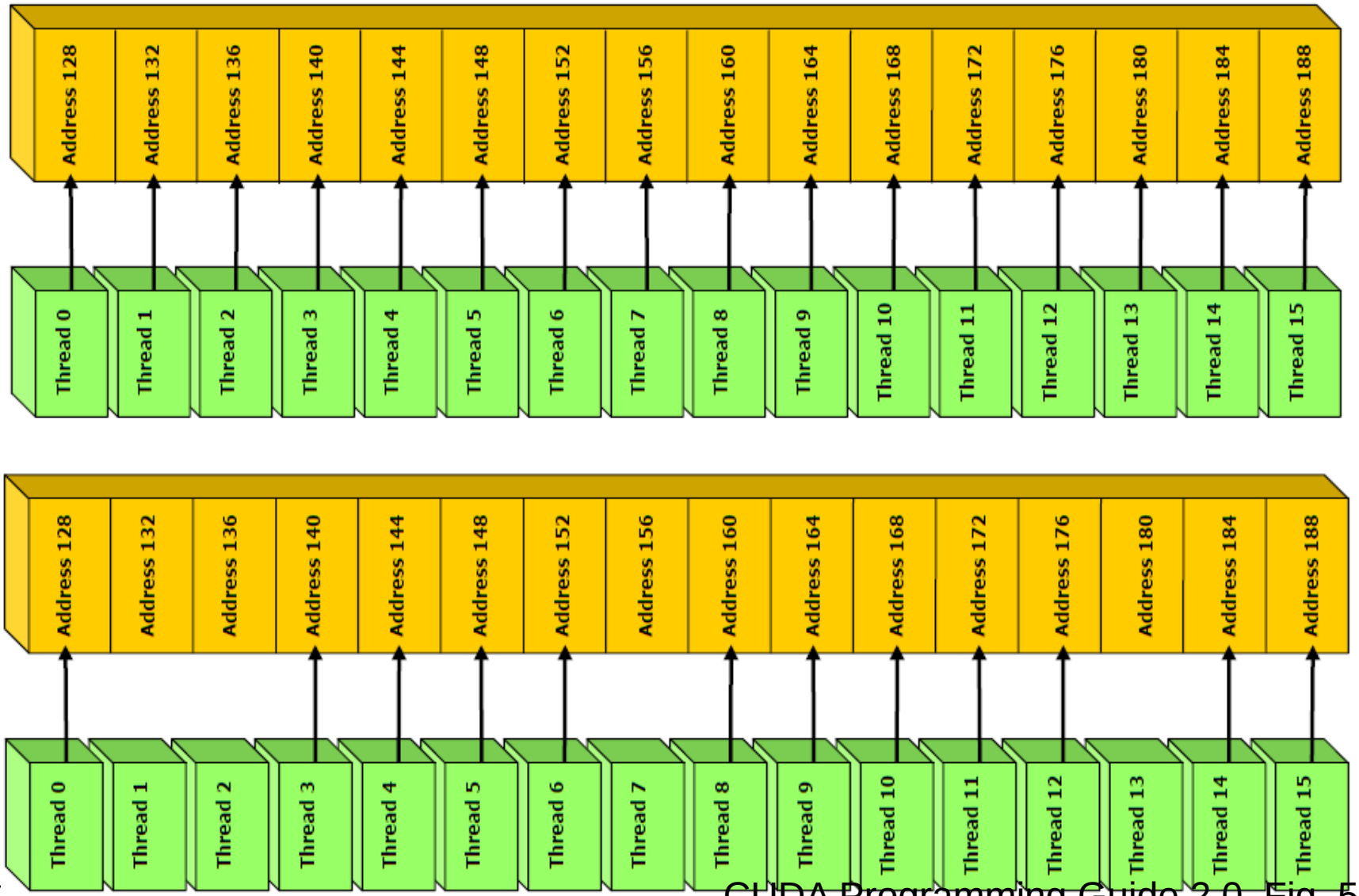
コアレッシング

- グラフィックスメモリは連続アドレスへのバーストアクセスに最適化
  - Tesla C10で理論値100GB/s
  - ランダムアクセスに弱い
- メモリアクセスのコアレッシング (coalescing)
  - 複数スレッドのメモリアクセスを一括（並列）処理
  - CUDAではハーフワープ毎にコアレッシング

コアレッシングされる条件（Tesla T10)

- ハーフワープの各スレッドが同一データサイズにアクセスする場合
  - 8ビット、16ビット、32ビット、64ビット
- かつ、それぞれアクセスする先が一定サイズのセグメント内に収まる場合
  - 8ビット→32バイト, 16ビット→64バイト、32ビット→128バイト、64ビット→128バイト
- その他アラインメントの制限もあり
- 古い世代のGPUではさらに制限あり
- 詳細は、CUDA Programming Guide 2.0, Section 5.1.2.1を参照
  - Compute capability 1.2の場合を参照

コアレッシング例その1



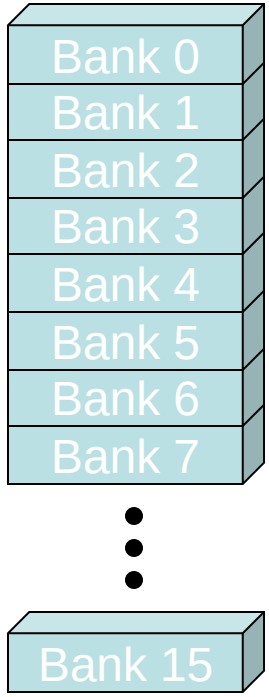
CUDA Programming Guide 2.0, Fig. 5.1

共有メモリのバンクと
バンクコンフリクト

注:　一部の図、文はUIUC ECE498より抜粋
(http://courses.ece.uiuc.edu/ece498/al/Syllabus.html)

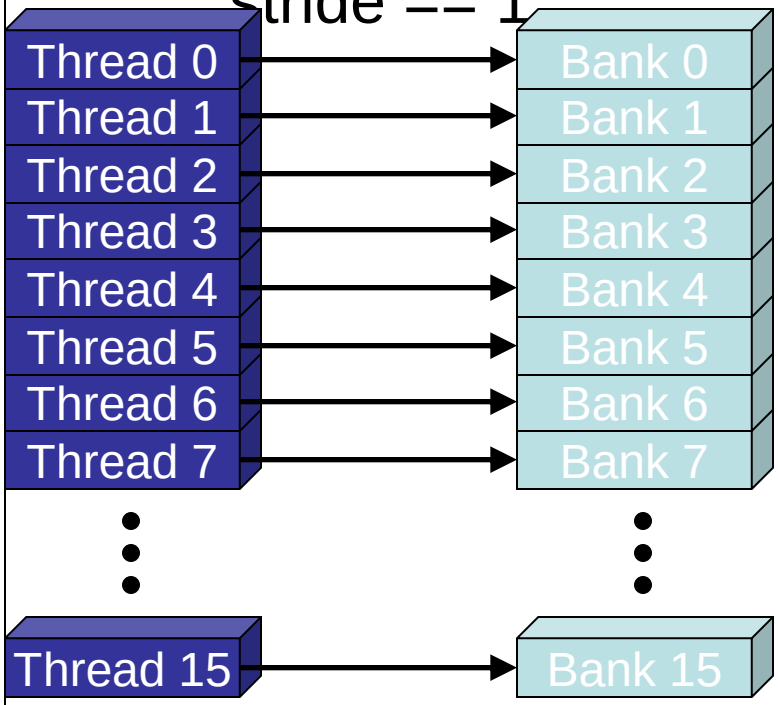
メモリバンク

- GPUのようなマルチスレッドアーキテクチャでは複数スレッドが同時にメモリにアクセス
  - メモリが一度に1アクセスしか処理できない場合、逐次処理に→ボトルネックになりがち
- CUDA共有メモリではメモリを16バンクに分割
  - 各バンクは連続したアドレスに対応
  - 16スレッドが別個のアドレスにアクセス→16バンクすべてを使うことにより並列処理
  - 複数スレッドが同一アドレスにアクセス→アクセス先バンクの衝突（バンクコンフリクト）

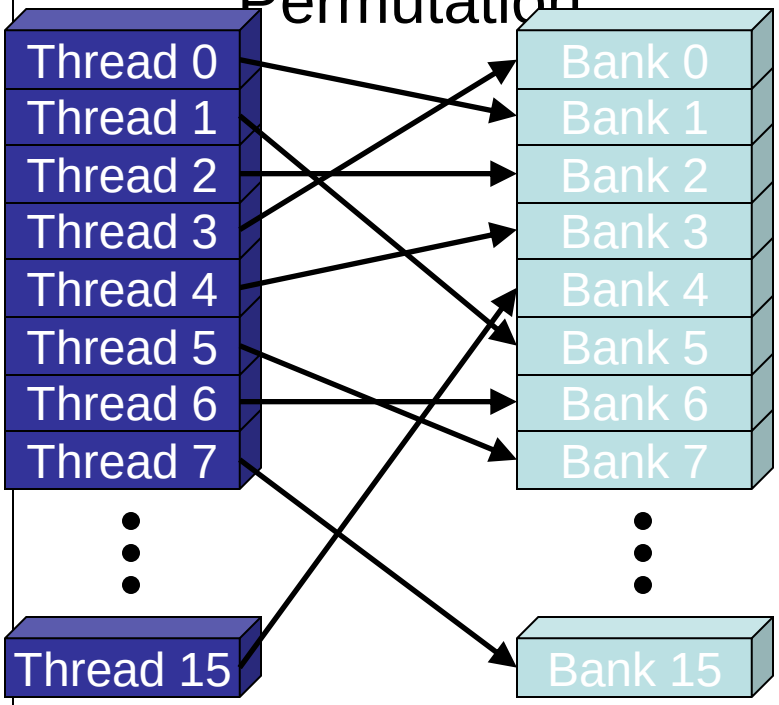


バンクコンフリクトが起きない例

- No Bank Conflicts
  - Linear addressing stride == 1

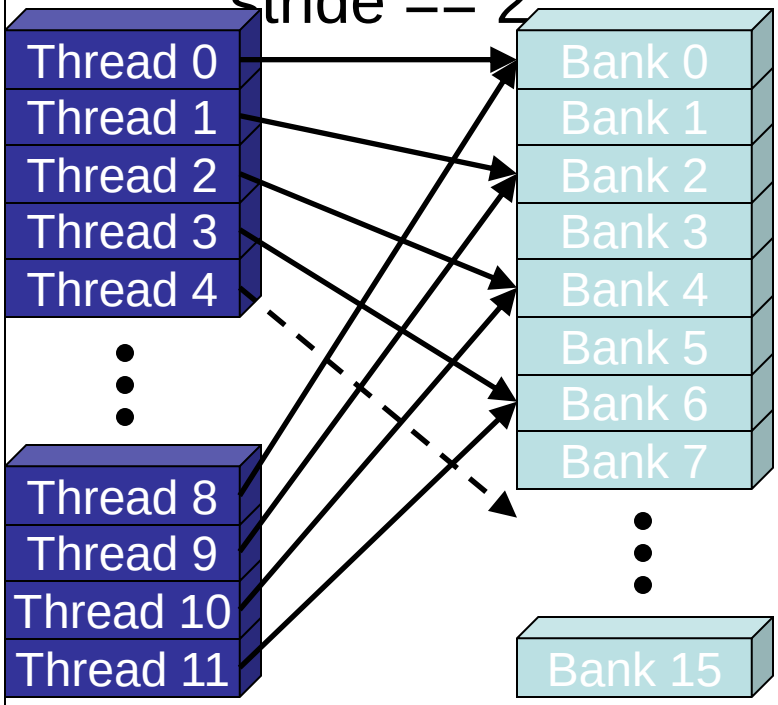


- No Bank Conflicts
  - Random 1:1 Permutation

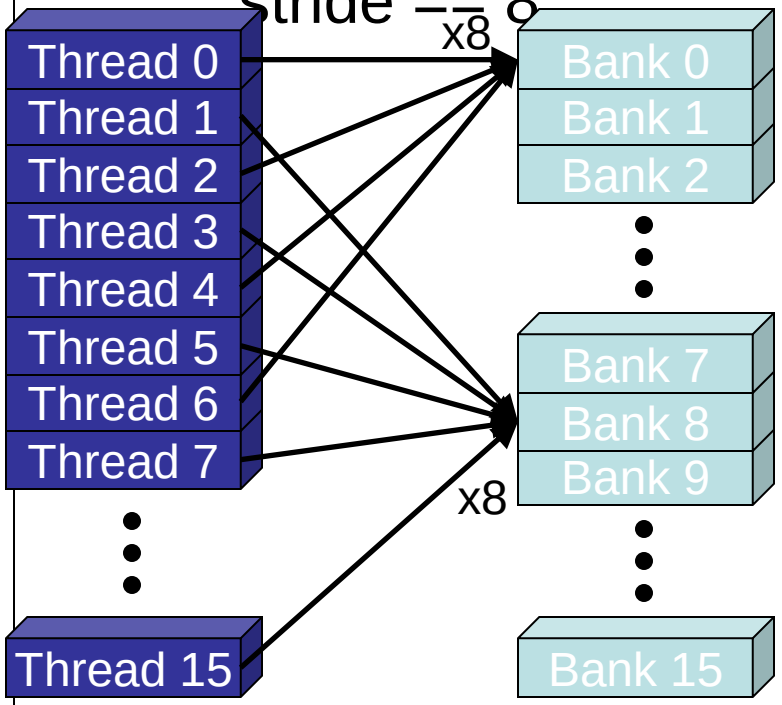


バンクコンフリクトが起きる例

- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



ここまでのまとめ

- メモリアクセス効率化
  - 共有メモリの利用
  - 連続領域へのアクセスによるメモリアクセスの一括処理
  - 共有メモリへのバンクコンフリクトの削減

ライブラリ等

CUDA SDKの利用方法

- サンプルコード、補助ライブラリなどを含む
- http://www.nvidia.com/object/cuda_get.html より最新版はダウンロード可能
- TSUBAMEではバージョンは2.2を利用
  - TSUBAMEの本講習用ディレクトリ以下にある NVIDIA_CUDA_SDK_*.run という名前のファイル
- ファイルの展開
  - sh <ダウンロードしたファイル名>
  - Enter連打
- コンパイル
  - 展開されたディレクトリへ移動
  - make

CUDA SDKについて

- CUTILライブラリ
  - 各種補助関数、マクロを提供
  - 例
    - CUDA_SAFE_CALL(call) → callを実行後、同期＆エラーチェック
  - SDK_DIR/common以下にプログラムファイル有り
  - projects以下のサンプルコードで使用
  - CUTILを利用したサンプルコードをベースにプログラムを構成する場合は、CUTIL関連のファイルへの依存性に注意
    - ヘッダーファイルの場所の指定 -ISDK_DIR/common/inc
    - ライブラリの指定 –LSDK_DIR/common/lib -lcutil

CUBLAS

- 単精度: Level 1, 2, 3すべて
- 倍精度
  - Level 1: DASUM, DAXPY, DCOPY, DDOT, DNRM2, DROT, DROTM, DSCAL, DSWAP, ISAMAX, IDAMIN
  - Level 2: DGEMV, DGER, DSYR, DTRSV
  - Level 3: ZGEMM, DGEMM, DTRSM, DTRMM, DSYMM, DSYRK, DSYR2K
- 行列のデータ順 → Column major (BLASと同じ)

CUBLAS利用法

cublas.hのインクルード

simpleCUBLAS.cより抜粋

cublasの初期化

GPUメモリに配列を確保

配列に入力データをセット

```
#include “cublas.h”

…

int main(int argc, char *argv[])
{

    status = cublasInit();
    if (status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "!!!! CUBLAS initialization error¥n");
        return EXIT_FAILURE;
    }
    status = cublasAlloc(n2, sizeof(d_A[0]), (void**)&d_A);
    if (status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "!!!! device memory allocation error (A)¥n");
        return EXIT_FAILURE;
    }
    …
    /* Initialize the device matrices with the host matrices */
    status = cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);
    if (status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "!!!! device access error (write A)¥n");
        return EXIT_FAILURE;
    }
```

CUBLAS利用法

simpleCUBLA.cより抜粋

BLASルーチンを呼び出し

結果をCPU側メモリへ転送

```
    /* Performs operation using cublas */
    cublasSgemm('n', 'n', N, N, N, alpha, d_A, N, d_B, N, beta, d_C, N);
    status = cublasGetError();
    if (status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "!!!! kernel execution error.¥n");
        return EXIT_FAILURE;
    }

    h_C = (float*)malloc(n2 * sizeof(h_C[0]));
    if (h_C == 0) {
        fprintf (stderr, "!!!! host memory allocation error (C)¥n");
        return EXIT_FAILURE;
    }

    /* Read the result back */
    status = cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);
    if (status != CUBLAS_STATUS_SUCCESS) {
        fprintf (stderr, "!!!! device access error (read C)¥n");
        return EXIT_FAILURE;
    }

    …
}
```

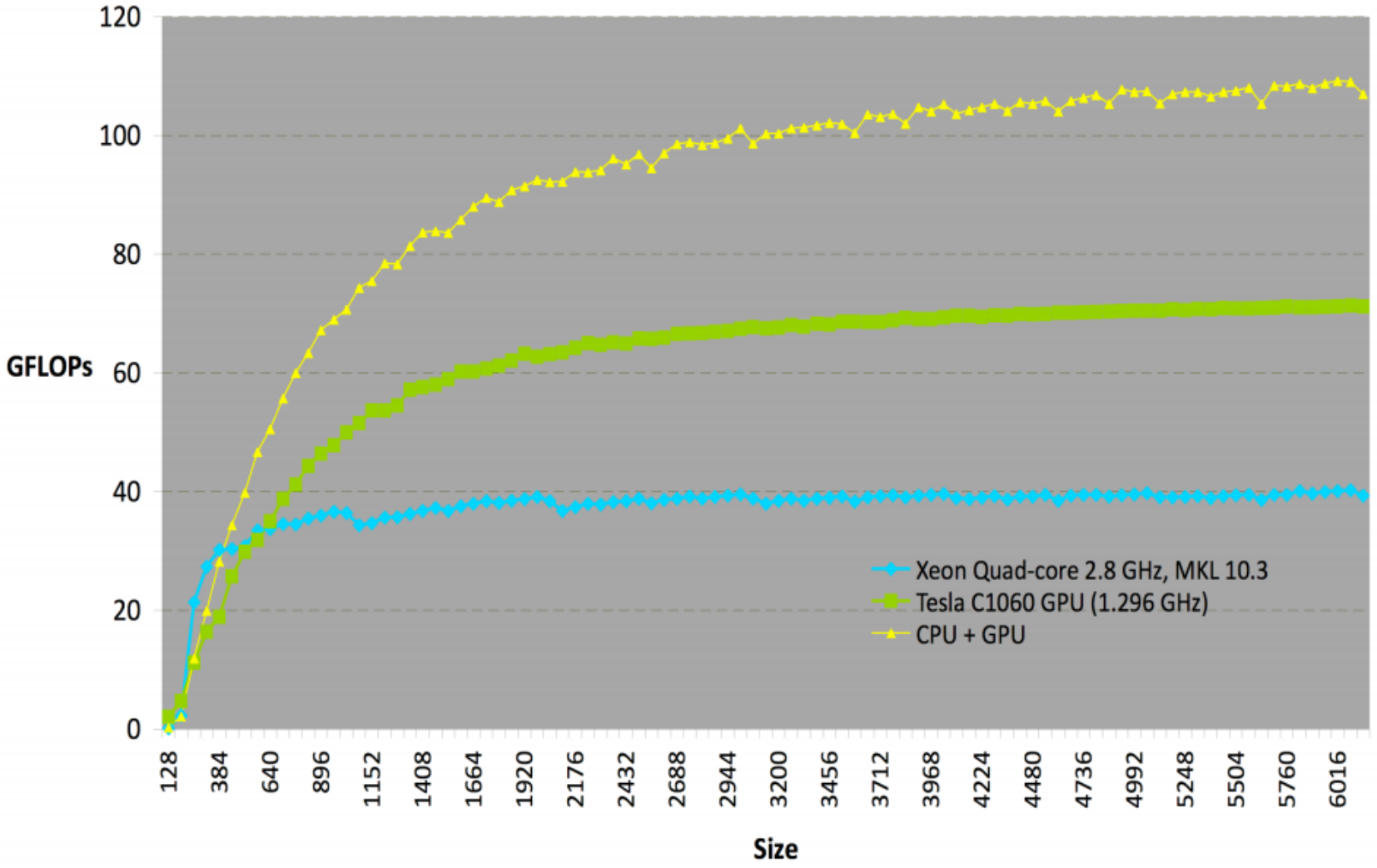
コンパイル

```
$ gcc simpleCUBLAS.c –I/opt/cuda/include -L/opt/cuda/lib -lcublas
```

CUBLASのFortranからの利用法

- 方法1： C言語版CUDAでGPUプログラムを書き、Fortranから呼び出し
  - GSIC Tesla利用の手引き5章を参照してください
    - http://www.gsic.titech.ac.jp/~ccwww/tebiki/tesla/tesla5.html
- 方法2： すべてFortranで記述
  - PGI社のコンパイラにCUDA　for Fotranの開発キットが付属

CUBLASの性能(DGEMM)



Source: Massimilaino Fatica, “CUDA Toolkit,” CUDA Tutorial at SC’08.

CUFFT

- FFTWをモデルに構成
  1. はじめにプランを作成しデータサイズ、GPUに最適化するためのデータを作成
  2. プランを用いて（複数回）FFTを実行
- 実数＆複素数の1D, 2D, 3D FFTをサポート
- 2Dと3Dでは配列内データ配置は row-major
  - Fortranから使う場合は転置する必要有り

CUFFTサンプルコード

```
#include “cufft.h”
#define NX 256
#define NY 128
cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
…
 /* Create a 1D FFT plan. */
 cufftPlan2d(&plan, NX,NY, CUFFT_C2C);
 /* Use the CUFFT plan to transform the signal out of
place. */
 cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);
 /* Inverse transform the signal in place. */
 cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);
 /* Note:
    Different pointers to input and output arrays
implies out of place   transformation
 */
 /* Destroy the CUFFT plan. */
  cufftDestroy(plan);
  cudaFree(idata), cudaFree(odata);
```


Source: Massimilaino Fatica, “CUDA Toolkit,” CUDA Tutorial at SC’08.

LAPACK

- CULA by CULAtools
  - http://www.culatools.com/
  - 無料版と有料版があり
    - 無料版は単精度の一部ルーチンのみ
    - 有料版はほぼすべてルーチンをカバー（倍精度含む）
  - 9月28日現在無料版のみダウンロード可
  - CUDA 2.3に依存するためTSUBAMEでは現在のところ利用不可
- MAGMA by テネシー大
  - http://icl.cs.utk.edu/magma/
  - フリー

補足

デバイス情報の参照

- SDK付属のdeviceQueryを利用
  - projects/deviceQuery 以下にソース
  - bin/linux/release/deviceQueryが実行バイナリ
  - /work/GPU/maruyama以下にもあり

例

```
tgg075055:~$ /work/GPU/maruyama/deviceQuery
There are 4 devices supporting CUDA

Device 0: "Tesla T10 Processor"
  Major revision number:                         1
  Minor revision number:                         3
  Total amount of global memory:                 4294705152 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.44 GHz
  Concurrent copy and execution:                 Yes
```

最適化効果の測定

- カーネル実行時間を計測
  - カーネル実行前後にgettimeofday 関数呼び出しを挿入
  - 但し、適切に同期させる必要あり
    - カーネル実行は非同期、メモリ転送も非同期可
- cudaThreadSynchronize
  - 呼び出し時点までに呼び出したデバイス関連の実行の終了をすべて待つ

時間計測例

プログラムリスト: matmul_mb.cu より抜粋

```
cudaThreadSynchronize();

gettimeofday(&t1, NULL);

// launch matmul kernel
matmul<<<dim3(N / BLOCKSIZE, L / BLOCKSIZE),
             dim3(BLOCKSIZE, BLOCKSIZE)>>>(Ad, Bd, Cd,
L, M, N);

cudaThreadSynchronize();

gettimeofday(&t2, NULL);
printf("Elapsed time: %f¥n", get_elapsed_time(&t1,
&t2));
```

非同期実行の処理の完了を待ち

非同期実行の処理の完了を待ち

デバッグ

- エミュレーション

  $ nvcc –emu

  – CPU実行用バイナリを生成
  – 通常のマルチスレッドCPUプログラムとしてデバッグ可
  – valgrind等のメモリチェッカも利用可
- CUDA_SAFE_CALL
  – CUDA API呼び出しのエラーチェック用マクロ
  – SDKのCUTILに定義
  – SDK内サンプルプログラムで利用
- CUDA Memcheck
  – CUDA 3.0に付属予定のメモリアクセスエラーチェッカー

その他取り上げられなかった事項

- メモリ
  - テクスチャメモリ、ローカルメモリ、コンスタントメモリ
  - 動的にサイズが決まる共有メモリの割り当て
  - アトミック操作
- ホストとGPU間のデータ転送の最適化
- エラー処理
- デバッガ
  - Linux 64-bit用にはハードウェアデバッガ有り（TSUBAMEでは未提供）
  - Windows向けにはVisualStudioプラグインとしてNexusが提供予定
- プロファイラ
  - 環境変数の設定によりGPU上のパフォーマンスカウンタ値をファイルへ保存
- CUDPP
  - http://gpgpu.org/developer/cudpp
  - CUDA SDKに付属
  - scan, reductionなどのプリミティブをライブラリとして提供

参考資料

- NVIDIA CUDAサイト
  - http://www.nvidia.com/object/cuda_develop.html
- CUDA Reference Manual & Programming Guide
  - 上記サイトよりダウンロード可能
- GSIC, Tesla利用の手引き
- プログラム例のソースコード
  - TSUBAME上の /work/GPU/maruyama に有り