CUDAプログラミング

東京工業大学学術国際情報センター
丸山直也
第9回GPUコンピューティング講習会
2011年8月3日

目次

1. CUDA概要
2. 最適化
3. マルチGPU計算
4. デバッガー
5. CUDA 4.0

サンプルコードは
/work0/GSIC/seminars/gpu-2011-08-03
からコピー可能です

CUDA概要

プログラミング言語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v4.0が最新版
  - Tsubameではv3.2が利用可能（8月中旬のメンテナンスによりv4へアップデート）
  - v3以降の多くの新機能はFermiのみ対応
- 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;
```

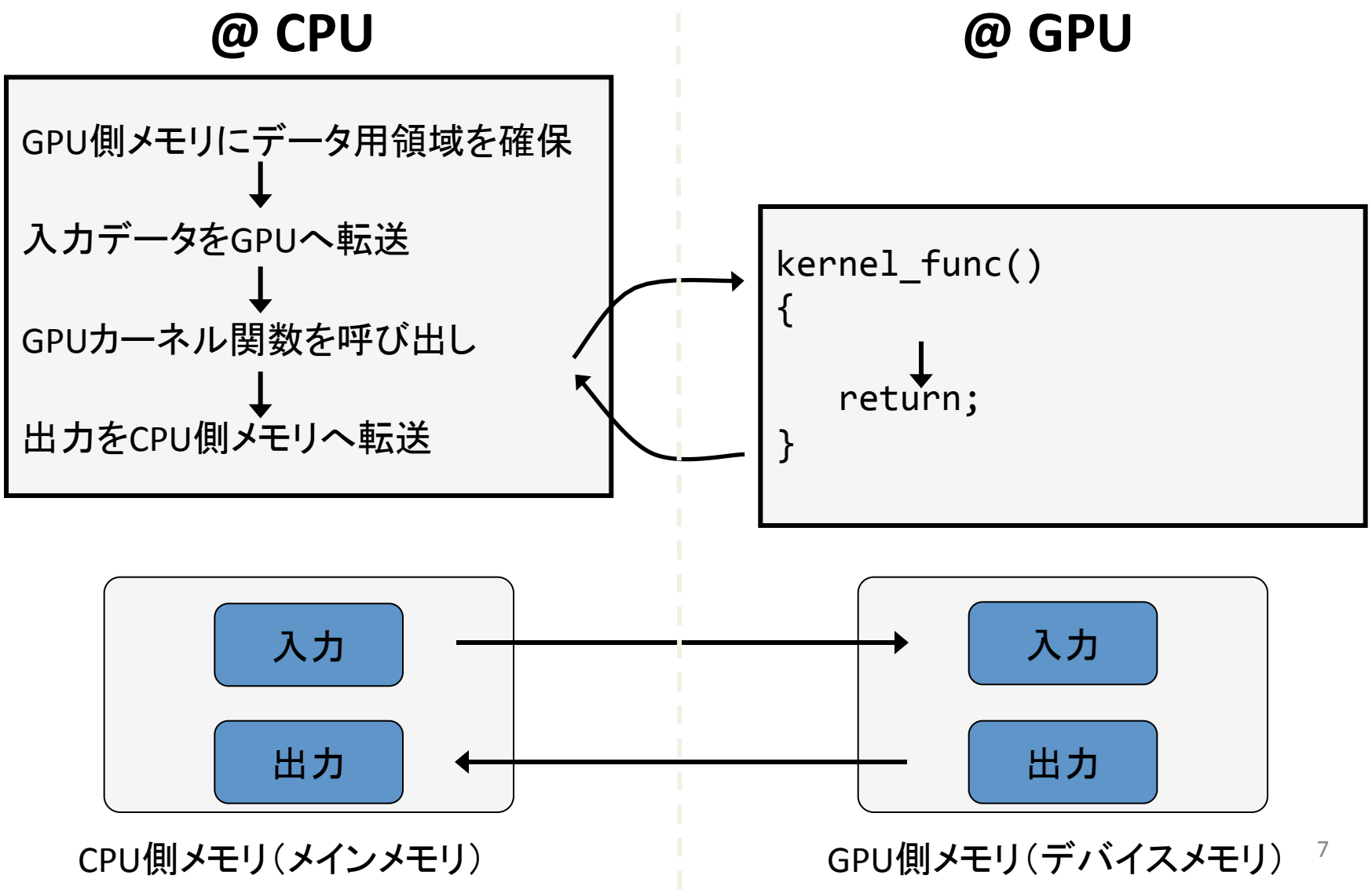
```
    for (i=0; i<N; i++) arrayH[i] = i;
    printf("input: ");
    for (i=0; i<N; i++)
        printf("%d ", arrayH[i]);
    printf("\n");

    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

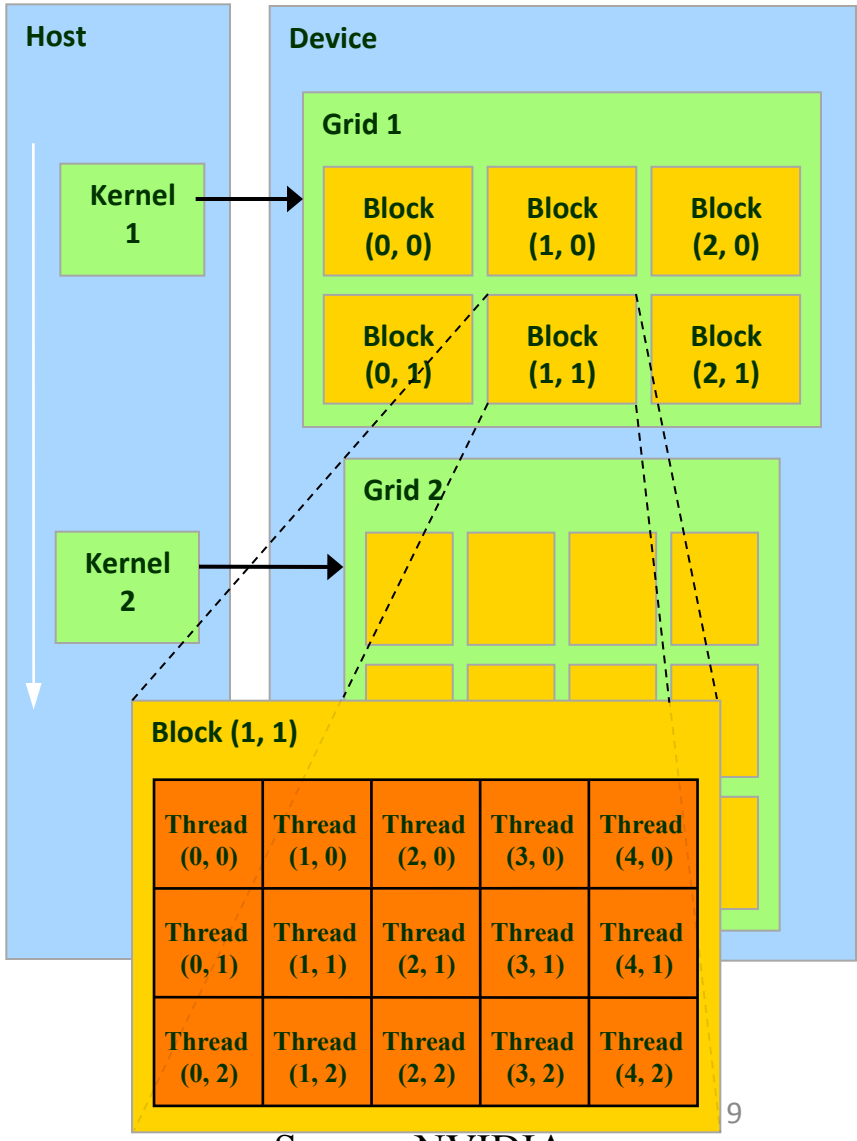


CUDAにおける並列化

- 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

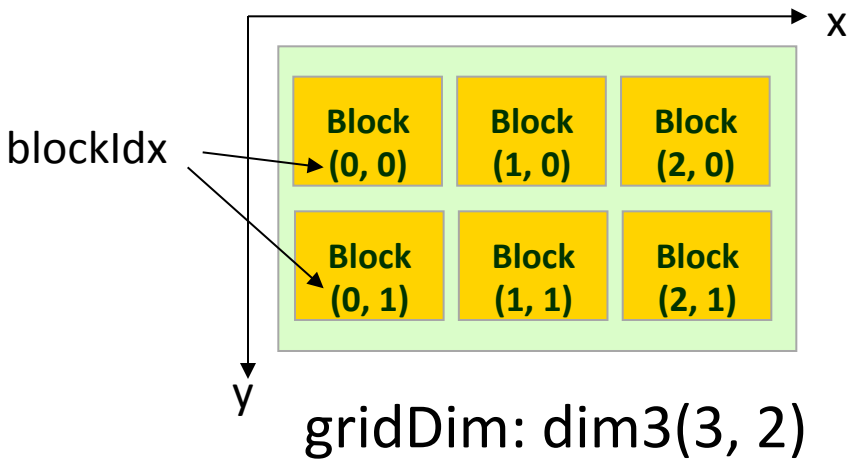
- 実行コンフィグ (Execution Configuration)
  - ホストプログラムからのカーネル呼び出し時に実行スレッド数を指定

```
<<<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

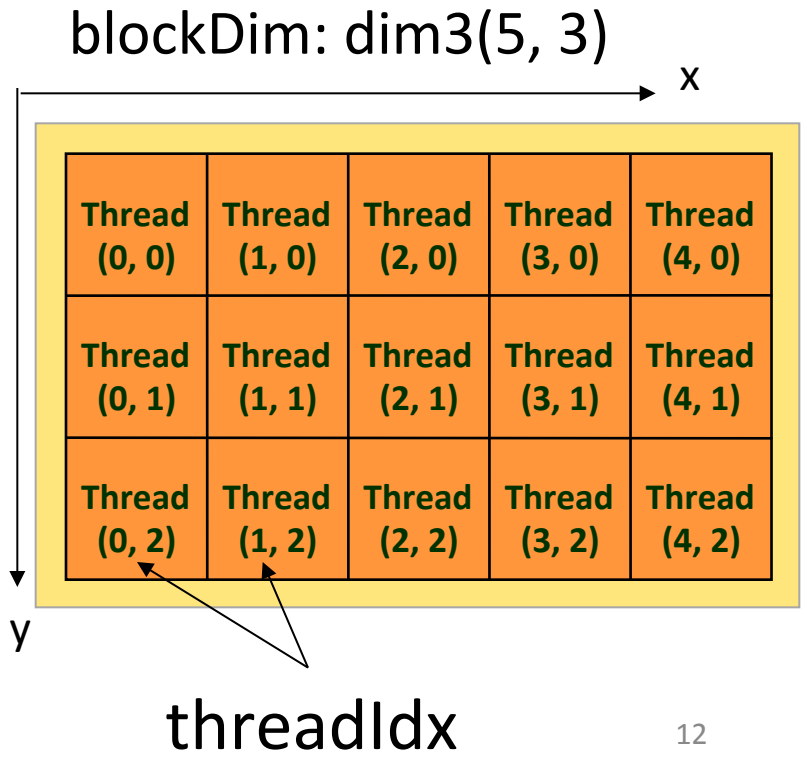
グリッド

- 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



最適化

最適化基本方針

- メモリアクセス効率化
  - オンチップメモリの有効活用
    - 共有メモリ
    - ハードウェアキャッシュ（Fermi以降）
  - 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
  - “divergent”分岐の削除
- ホスト・デバイス間データ転送

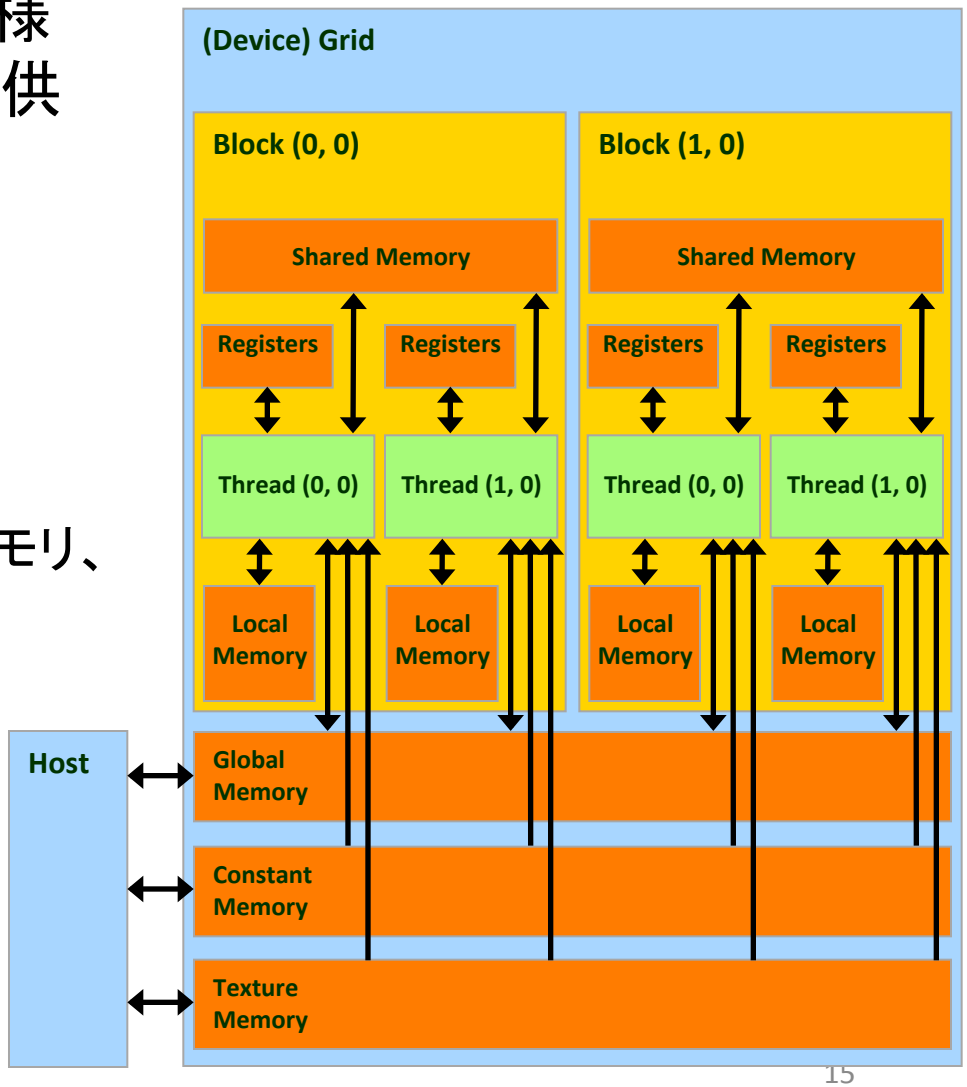
- ハードウェアの詳細を（それなりに）知る必要有り
- ただし、最適化による効果も大きい

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、（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、コンスタントメモリ、テクスチャメモリ、（L2キャッシュ）
- ないもの
  - スタック → CUDA v4 より有り

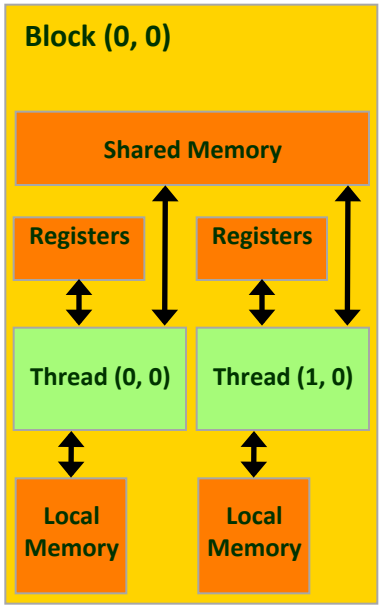
それぞれ速度と容量にトレードオフ有
（高速＆小容量 vs. 低速＆大容量）
→ メモリアクセスの局所性が重要



Source: Kirk and Hwu, ECE 498AL, UIUC

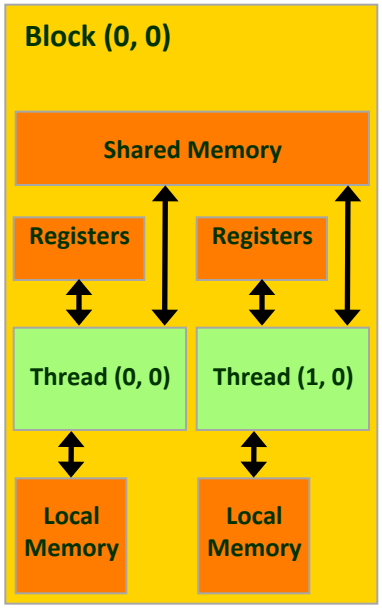
スレッド固有メモリ

- 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）
  - FermiからはL1/L2にキャッシュ



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
  - レジスタに次いで高速
  - SMあたり16KBもしくは48KB（Fermi）

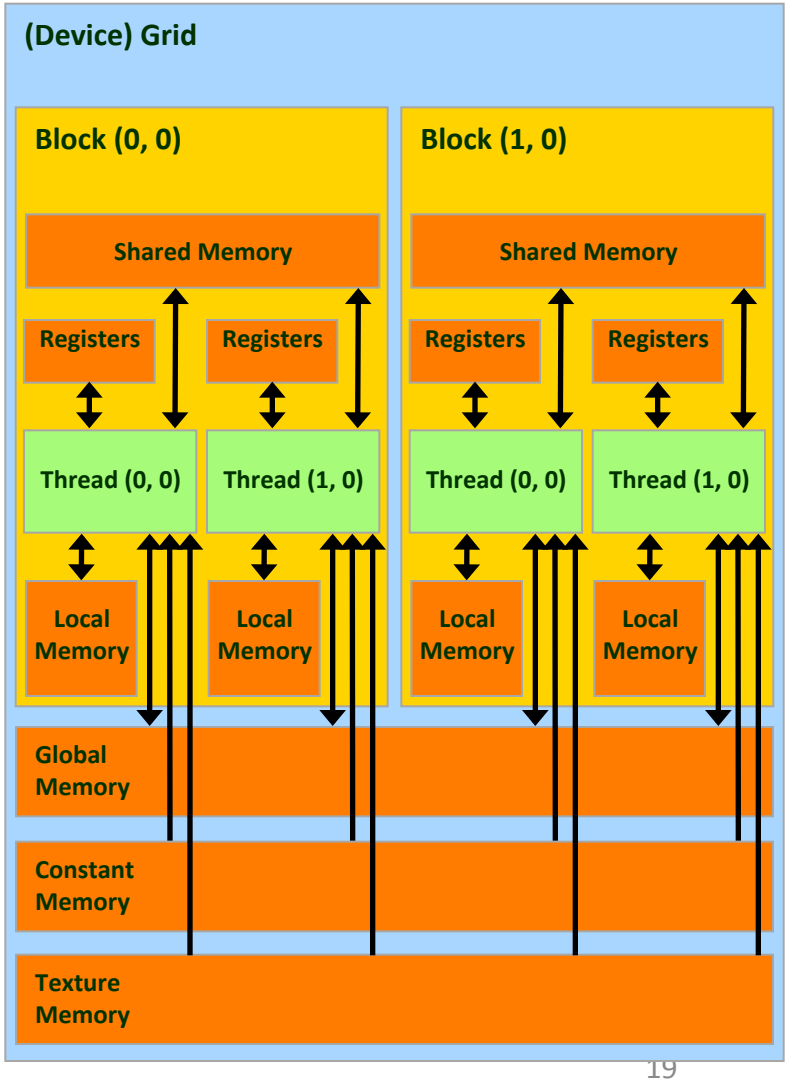


L1キャッシュ

- Fermi GPUより搭載
- 128Bキャッシュライン
- 共有メモリと物理的に同じ領域に存在
- SMあたり16KBもしくは48KB（選択可）
  - cudaFuncSetCacheConfig() 関数により設定
  - 例： cudaFuncSetCacheConfig(inc1, cudaFuncCachePreferL1)

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
  - この講習会では扱わない
- テクスチャメモリ
  - この講習会では扱わない



L2キャッシュ

- Fermiより搭載
- C2050で768KB
- 128Bキャッシュライン
- 全SMより共有
- アトミック操作などの実装にも利用→Fermi以前と比べて性能向上

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

行列積：共有メモリ利用なし

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

行列積： 共有メモリ利用なし（2）

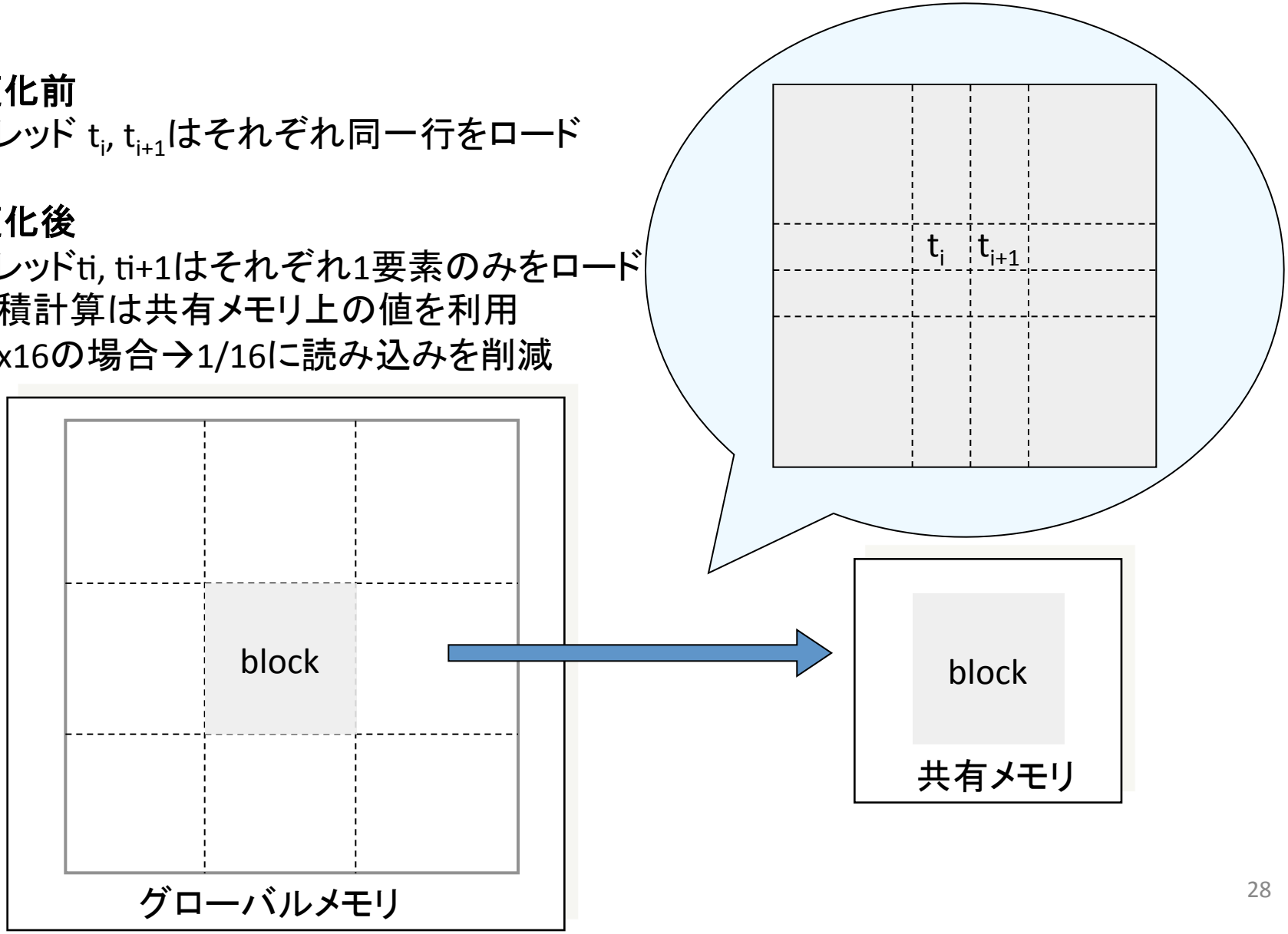
プログラムリスト: matmul_mb.cu より抜粋

```
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;
}
```

共有メモリの利用

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();
```

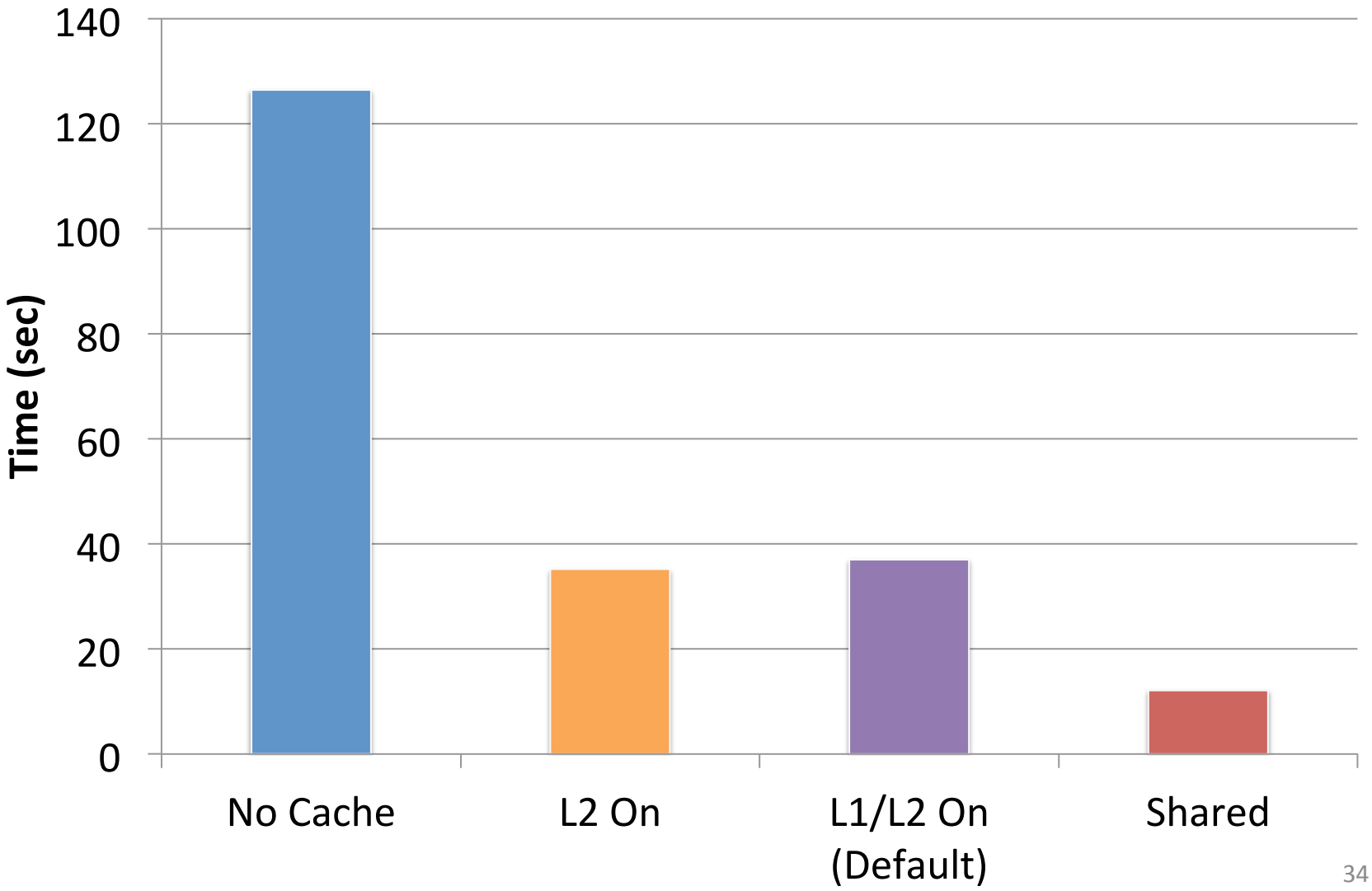
行列積(共有メモリ版)

```
// 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;
}
```

最適化の効果

- キャッシュの有無、共有メモリ最適化による行列積（1024^3）の性能の違いを比較
- ソースコード
  - matmul_mb.cu
    - 共有メモリは使用せず
  - matmul_shared.cu
    - 共有メモリを用いた並列行列積
- キャッシュ利用のオン・オフ
  - nvccコンパイルオプションに “-Xptxas –dlcm=オプション” を与える
  - オプション
    - ca → 全レベルでキャッシュを有効化（デフォルト）
    - cg → L2のみ有効化
    - cs → L1/L2 すべて無効化
- 評価環境はTSUBAMEのM2050一台

最適化の効果


140
120
100
80
60
40
20
0
Time (sec)
No Cache
L2 On
L1/L2 On
(Default)
Shared

マルチGPU計算

マルチGPU計算

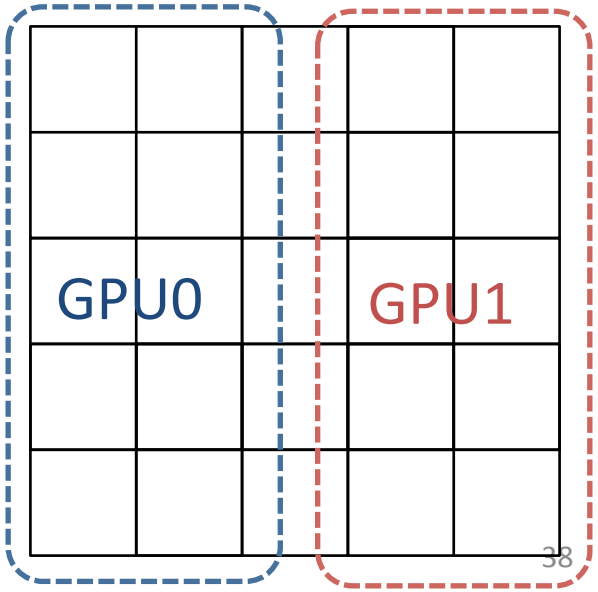
- ここまでは単一のGPUを用いた計算のみ
- より高速に大規模な問題をとくには複数のGPUの利用が効果的
  - TSUBAME2ではノード内にGPU3基、システム全体で4000基
- CUDAに加えて他の並列計算モデルと組み合わせる必要あり
  - CUDA+OpenMP → ノード内の複数GPUを利用
  - CUDA+MPI → ノードをまたがる複数GPUを利用

ノード内マルチGPU計算

- GPU毎に別個のCPUスレッドもしくはプロセスを割り当てる必要有り
  - OpenMP、pthreads等を利用
- 計算対象問題を複数GPUで分割
  - inc → 配列を複数GPUで分割
  - ヤコビ法 → 領域を複数GPUで分割

ヤコビ法のマルチGPU計算

- ノード内に限定
- OpenMPを使用
  - OpenMPのスレッドiがGPU i を管理
- 利用GPU番号の指定
  - cudaSetDevice(番号）関数を利用
  - 他のGPU関連の関数を実行するまえに呼び出す必要あり
- 利用中GPU番号の確認
  - cudaGetDevice 関数

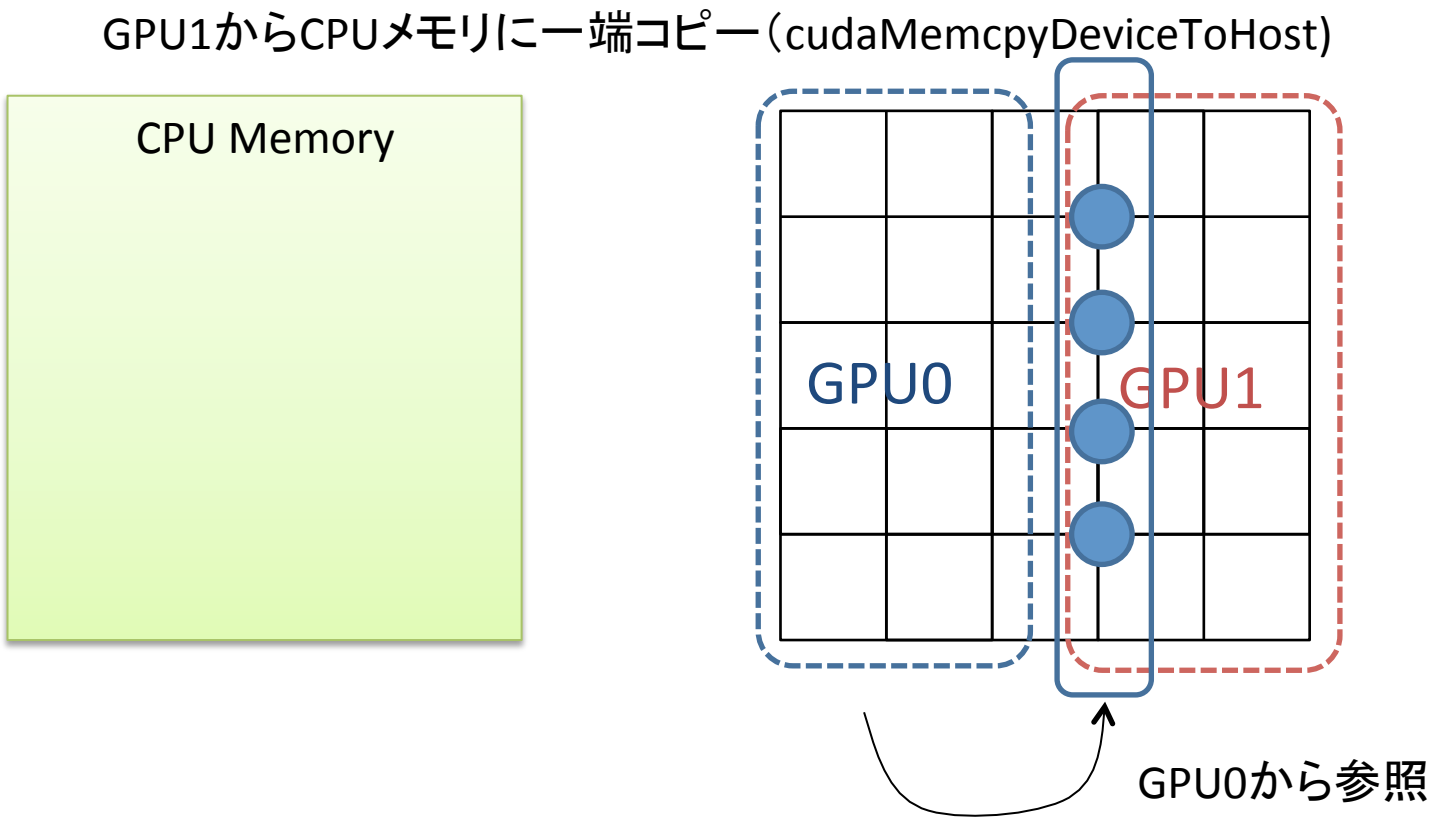


処理の大まかな流れ

```
// 利用GPU数の指定(TSUBAMEでは最大3)
omp_set_num_threads(3);
#pragma omp parallel
{
   // このブロック内はomp_set_num_threadsで指定されたスレッド数で並列実行

   //  利用GPU番号をOpenMPスレッド番号から設定
   cudaSetDevice(omp_get_thread_num());
   // メモリ割り当て等初期化
   cudaMalloc(…); …;
   for (int i = 0; i < MAX_ITER; i++) {
     // Y方向に3GPUで分割
     jacobi_kernel<<<dim3(NX/16, NY/16/3), dim3(16,16)>>>(…);
     // バッファの切り替え
   }
}
```

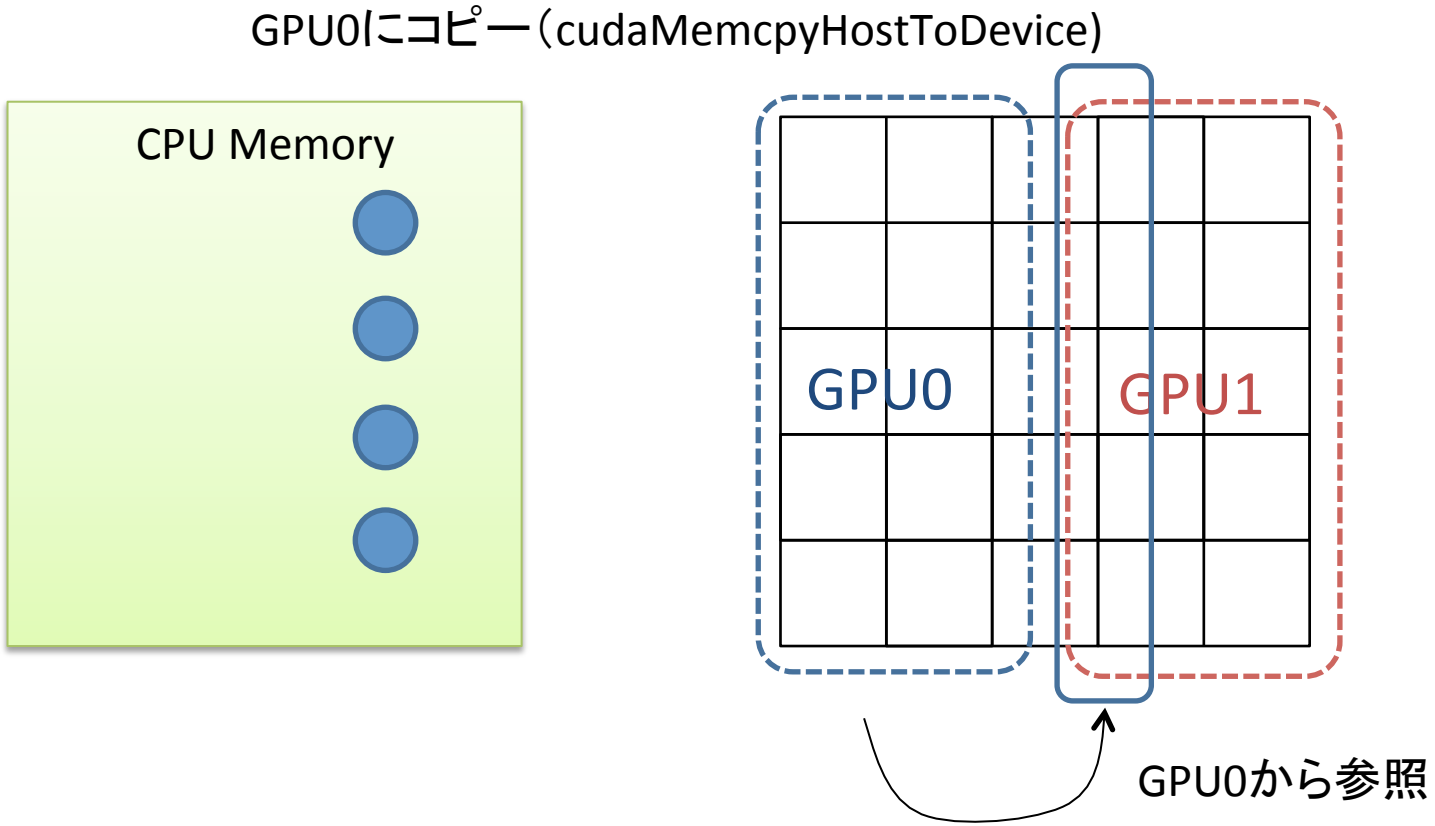
GPU間隣接領域の交換

- GPUはそれぞれ別個のメモリを利用
- 他のGPU上のデータの直接の参照は不可
  →CUDA 4.0 & Fermi Tesla GPUでは可能に



GPU間隣接領域の交換

- GPUはそれぞれ別個のメモリを利用
- 他のGPU上のデータの直接の参照は不可
  →CUDA 4.0 & Fermi Tesla GPUでは可能に



領域交換処理の流れ

```
for (int i = 0; i < MAX_ITER; i++) {
  jacobi_kernel<<<dim3(NX/16, NY/16/3), dim3(16,16)>>>(…);
  // Y軸方向のみの分割
  GPU(i-1)番へ渡す列をホストにコピー
  GPU(i+1)番へ渡す列をホストにコピー
  // 全スレッドがホストにコピーするまで待つ
#pragma omp barrier
  GPU(i+1)番のデータをGPU i番のメモリにコピー
  GPU(i-1)番のデータをGPU i番のメモリにコピー
  バッファの切り替え
}
```

コンパイル方法

- nvccにOpenMPを使うための指示が必要

```
$ nvcc –c test.cu –Xcompiler –fopenmp
$ nvcc test.o test -lgomp
```

CUDA 4.0によるマルチGPUプログラミング

- 特定の条件ではノード内GPU間で相互にデータコピー可能
  - CPUに一旦コピーする必要なし
  - 性能向上（の可能性）
- 条件
  - Fermi世代のTesla GPU
    - M/C/S/X 2050, 2070, 2090
    - GeForce系は不可（？）
  - 同一チップセットに接続
    - TSUBAME2.0では二つのチップセットにより3つのGPUを接続
      → 同一チップセットに接続された2つのGPU間では相互データアクセス可能

デバッガー

CUDA-GDB

- GDB
  - Linux標準のデバッガ
  - 標準的なデバッガの機能を搭載
    - シングルステップ実行、ブレイクポイント、など
- CUDA-GDB
  - GDBをベースにGPU上のプログラム実行のデバッグをサポート
  - ホストコードは通常のGDBと同様にデバッグ可能
  - カーネル関数のシングルステップ実行やブレイクポイントの設定が可能
- 両者ともコマンドラインインターフェイスのみ
  - TotalViewなどはより使いやすいGUIを提供

CUDA-GDBの使い方

- NVIDIA CUDA-GDB Manualより
- デモ
  - サンプルプログラム: bitreverse.cu

```
__global__ void bitreverse(unsigned *data) {
  extern __shared__ int array[];
  array[threadIdx.x] = data[threadIdx.x];
  array[threadIdx.x] = ((0xf0f0f0f0 & array[threadIdx.x]) >> 4) |
      ((0x0f0f0f0f & array[threadIdx.x]) << 4);
  array[threadIdx.x] = ((0xcccccccc & array[threadIdx.x]) >> 2) |
      ((0x33333333 & array[threadIdx.x]) << 2);
  array[threadIdx.x] = ((0xaaaaaaaa & array[threadIdx.x]) >> 1) |
      ((0x55555555 & array[threadIdx.x]) << 1);
  data[threadIdx.x] = array[threadIdx.x];
}
```

ステップ1：再コンパイル

- nvcc に –g –G オプションが必要
  - -g → ホストコードにデバッグ用情報を付加
  - -G → カーネルコードにデバッグ用情報を付加＆最適化を抑制

```
$ nvcc –g –G bitreverse.cu –o bitreverse
```

- 注意
  - 上記オプションにより生成されるコードは最適化が抑制されるため通常のコンパイル時とは異なる→特定の最適化によってのみ発生するバグがデバッグ時には発生しない可能性もあり

ステップ2：cuda-gdbコマンド

- 実行プログラムを cuda-gdb コマンドにより起動
  - 通常の gdb のように実行中のプログラムにアタッチすることは未サポート

```
$ cuda-gdb ./bitreverse
NVIDIA (R) CUDA Debugger
3.1 beta release
Portions Copyright (C) 2008,2009,2010 NVIDIA Corporation
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
. . .
This GDB was configured as "x86_64-unknown-linux-gnu"...
Using host libthread_db library "/lib64/libthread_db.so.1".
(cuda-gdb)
```

ステップ3:ブレイクポイントのセット

- ブレイクポイント
  - デバッグ中プログラムの特定の場所(行、関数など)
  - プログラムの実行がブレイクポイントに到達するとその時点で一旦実行を停止
  - breakコマンドを利用
- ウォッチポイントは未サポート

```
(cuda-gdb) break main
Breakpoint 1 at 0x400db0: file bitreverse.cu, line 25.
(cuda-gdb) break bitreverse
Breakpoint 2 at 0x40204f: file bitreverse.cu, line 8.
(cuda-gdb) break 29
Breakpoint 3 at 0x40205b: file bitreverse.cu, line 29.
```

ステップ4：実行開始

- run コマンドにより実行開始

```
(cuda-gdb) run
Starting program: /home0/usr0/maruyama-n-aa/gpgpu-lecture/
code/bitreverse/bitreverse
[Thread debugging using libthread_db enabled]
[New process 2740]
[New Thread 47701781626496 (LWP 2740)]
[Switching to Thread 47701781626496 (LWP 2740)]

Breakpoint 1, main () at bitreverse.cu:21
21          unsigned *d = NULL; int i;
(cuda-gdb)
```

ステップ5：GPUへ制御が移るまで実行

```
(cuda-gdb) c
Continuing.
[Launch of CUDA Kernel 0 on Device 0]
[Switching to CUDA Kernel 0 (<<<(0,0),(0,0,0)>>>)]

Breakpoint 2, bitreverse <<<(1,1),(256,1,1)>>>
(data=0x5100000)
    at bitreverse.cu:10
10          array[threadIdx.x] = data[threadIdx.x];
(cuda-gdb)
```

ステップ6:状態の確認

```
(cuda-gdb) info cuda threads
<<<(0,0),(0,0,0)>>> ... <<<(0,0),(255,0,0)>>> bitreverse
    <<<(1,1),(256,1,1)>>> (data=0x5100000) at bitreverse.cu:10
(cuda-gdb) bt
#0  bitreverse <<<(1,1),(256,1,1)>>> (data=0x5100000) at bitreverse.cu:10
(cuda-gdb) thread
[Current thread is 2 (Thread 47984931774080 (LWP 3031))]
(cuda-gdb) thread 2
[Switching to thread 2 (Thread 47984931774080 (LWP 3031))]#0
0x0000000000402289 in main () at bitreverse.cu:30
30        bitreverse<<<1, N, N*sizeof(int)>>>(d);
(cuda-gdb) bt
#0  0x0000000000402289 in main () at bitreverse.cu:30
(cuda-gdb) info cuda kernels
* 0 Device 0 bitreverse <<<(1,1),(256,1,1)>>> (data=0x5100000)
    at bitreverse.cu:10
(cuda-gdb) cuda kernel 0
[Switching to CUDA Kernel 0 (<<<(0,0),(0,0,0)>>>)]#0  bitreverse <<<(1,1),
(256,1,1)>>> (data=0x5100000) at bitreverse.cu:10
10        array[threadIdx.x] = data[threadIdx.x];
(cuda-gdb) bt
#0  bitreverse <<<(1,1),(256,1,1)>>> (data=0x5100000) at bitreverse.cu:10
(cuda-gdb)
```

CUDA-GDBその他コマンド

- print コマンド
  - 変数やメモリの値の表示
  - 共有メモリも表示可能
- info コマンド
  - info cuda system → ノード全体の情報を表示
  - info cuda device → デバイスの情報を表示
  - info cuda sm → SMで実行中のワープの情報を表示
  - info cuda warp → スレッドダイバージェンスなどを表示
  - その他も。CUDA GDB Manualを参照

CUDA-GDBその他コマンド

- set cuda memcheck on
  - グローバルメモリに関するアクセス違反を検出
  - cuda-memcheck コマンドも利用可能

```
$ vi bitreverse.cu // insert an invalid memory access bug
$ nvcc –g –G bitreverse.cu –o bitreverse
$ cuda-memcheck ./bitreverse
. . .

========= Invalid read of size 4
=========     at 0x000000a0 in bitreverse.cu:8:bitreverse
=========     by thread (0,0,0) in block (0,0)
========= Address 0x4050ffffc is out of bounds
=========
========= ERROR SUMMARY: 1 error
```

CUDA 4.0

CUDA 4.0新機能

- 複数GPUの単一CPUスレッドからの利用
  - これまでは1GPUあたり1スレッド必要
- 単一GPUコンテキストを複数スレッドから共有
- C++ new/delete
- インラインPTXアセンブラ
- Thurstライブラリの統合
- 統合アドレス空間
- GPUDirect v2.0

Thurst

- 配列データを簡便に操作するためのC++用ライブラリ
- C++ STLのvector型の似たGPU用 vector 型を提供
- リダクション、ソート等が可能
- CUDA 4.0に付属（ヘッダーファイルのみ）

サンプルコード

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <iostream>
int main(void) {
   // H has storage for 4 integers
   thrust::host_vector<int> H(4);
   // initialize individual elements
   H[0] = 14;  H[1] = 20;  H[2] = 38;  H[3] = 46;
   // H.size() returns the size of vector H
   std::cout << "H has size " << H.size() << std::endl;
  for(int i = 0; i < H.size(); i++)
      std::cout << "H[" << i << "] = " << H[i] << std::endl;
   // resize H
   H.resize(2);
   std::cout << "H now has size " << H.size() << std::endl;
   // Copy host_vector H to device_vector D
   thrust::device_vector<int> D = H;
   // elements of D can be modified
   D[0] = 99;
   for(int i = 0; i < D.size(); i++)
      std::cout << "D[" << i << "] = " << D[i] << std::endl;
   // H and D are automatically deleted when the function returns
}
```

http://code.google.com/p/thrust/wiki/QuickStartGuide

GPUDirect v2.0

- ノード内の複数GPU間で直接 cudaMemcpy が可能
  - CPUメモリにコピーする必要なし
  - プログラミングが簡便に
  - 性能も多少改善
- ただし、PCI Expressのスイッチを複数介する場合は動作せず（？）
  - TsubameではノードGPU3GPUを2つのPCIeスイッチで接続
  - 3GPU間ではGPUDirect動作せず（？）
- MPI等のライブラリがGPU対応になる可能性も
  - MPI_SendによってGPU上のデータを直接転送指定
  - MPIライブラリが適切にコピー（GPUDirectが利用可能な場合は直接コピー、そうではない場合は一旦CPUにコピー）
  - MVAPICH、OpenMPI等が開発中（？）
- 異なるノードのGPU間の直接コピーは不可
  - 将来サポートされる見込み