CUDA プログラミング入門
2012 年 6 月 1 日版

理化学研究所　情報基盤センター
青山幸也

はじめに

本書は、C 言語のプログラムを CUDA 化して GPU を使用する方法を説明した、初心者向けの講習会テキストです。GPU は、理化学研究所の RICC (Riken Integrated Cluster of Clusters) に導入されている、NVIDIA 社の C1060 を想定し、Compute Capability は 1.3 を想定します。本書を読むための前提として、C 言語の基本的な知識が必要になります。

Fortran のプログラムを CUDA 化する方法、PGI コンパイラの使用方法、GPU の新アーキテクチャー Fermi での使用方法などについては、今後順次加筆する予定です。

なお、本書の記述中に、誤字や脱字、プログラムのバグ、筆者の誤解による記述ミスなどありましたらご容赦ください。また、本書の内容に関する責任は負いかねますので、ご了承願います。

本書は不定期に加筆修正しており、最新版は `http://accc.riken.jp/HPC/training.html` にあります。

2011 年　青山　幸也

目 次

第1章　概要

1-1　GPU/CUDA の概要

GPU/CUDA とは

まず GPU/CUDA が出てきた背景を簡単に説明します。パソコンに装着するビデオカードは、画像処理を行う LSI である GPU (Graphics Processing Unit) と、画像データを保持するビデオメモリ (VRAM) から構成されます。画像処理の計算は、データ量が非常に多く、計算が単純（例えば行列計算）で、並列に処理できるという特徴があります。GPU は、トランジスタの大半を、並列に実行できる多数の演算器に割り当てることにより、画像処理の計算を高速に行います。

このように高速で、かつ価格も安い GPU を、画像処理以外の汎用計算 (例えば HPC 分野) に適用する試みが、2000 年代前半に始まりました。これを GPGPU (General-Purpose computing on Graphics Processing Units) と言います。ところが、当時の GPU のアーキテクチャーは汎用計算に向いておらず、またプログラミングも画像処理用の特殊な方法で行わなければならず、面倒でした。

GPU を汎用計算で容易に扱えるようにするために、GPU メーカーの NVIDIA（エヌビディア）社は、2006 年に 2 つの技術を発表しました。1 つは、画像処理と計算処理を統合した自由度の高いアーキテクチャー G80 です。もう 1 つは、プログラミングを容易にするための統合開発環境 CUDA（クーダ）(Compute Unified Device Architecture) です。CUDA (正式には CUDA Toolkit) は、NVIDIA 社が無償で提供し、コンパイラ、数値計算ライブラリー、デバッガー、プロファイラー、マニュアル、サンプルプログラムなどから構成され、以下の特徴があります。

- C 言語から使用することができます (Fortran からも使用可能ですが、今回の講習では説明しません)。
- 画像処理用の特殊な方法でプログラミングする必要はありません。
- OS は、Windows，Linux，Mac OS X の元で使用することができます。

GPU のアーキテクチャ

GPU のアーキテクチャは、G80 の後、図 1-1-1 のように変遷し、最新は 2009 年に発表された Fermi（フェルミ）です。各アーキテクチャーの詳細は、「CUDA C Programming Guide」(Appendix. G) を参照して下さい。

CUDA のマニュアルなどでは、アーキテクチャー名ではなく、「Compute Capability」という用語を使用し、例えば「〜機能は Compute Capability 1.3 以降でサポートされている」のように記述されていることがあります。理研の RICC (Riken Integrated Cluster of Clusters) に導入されている GPU は、図 1-1-1 の下線に示す「Tesla（テスラ）C1060」で、アーキテクチャーは GT200、Compute Capability は 1.3 です。また、アーキテクチャー、Compute Capability とは別に、CUDA Toolkit にもバージョンがあり、理研の RICC には、2011 年 2 月現在、「CUDA Toolkit 3.2」(以後 CUDA 3.2 と省略します) が導入されています。なお、CUDA Toolkit は、バージョンによって、使用方法や出力結果が、本書で説明する内容と若干異なることがあります。

アーキテクチャー	Compute Capability	代表製品 GeForce	代表製品 Tesla	倍精度演算
G80	1.0	8800 GTX	C870	不可
G80/G92	1.1	9800 GTX		不可
G92	1.2	GT 240		不可
GT200	1.3	GTX 285	C1060	可能
Fermi(GF100)	2.0	GTX 480	C2050	可能

図 1-1-1

GPU の得意な計算と不得意な計算

前述のように、GPU ではトランジスタの大半を、並列に実行できる多数の演算器に割り当てているため、ピーク性能は CPU よりも圧倒的に高速です。このため、単純で、並列に実行可能な計算が得意です。一方 GPU は、キャッシュ処理や `if` 文の分岐予測などの、通常の CPU が行う複雑な処理は行わないので、分岐が多い計算は苦手で、並列度が低い計算も苦手です。

また、GPU は価格が安いという長所があり、家庭用パソコンにも装着することができます。

1-2　RICC の GPU

多目的 PC クラスタ

理研 RICC は、超並列 PC クラスタ、多目的 PC クラスタ、MDGRAPE-3 クラスタ、大容量メモリ計算機から構成されています。このうち多目的 PC クラスタに GPU が導入されています。多目的 PC クラスターは、図 1-2-1 の仕様のマシンが 100 台で構成されています。

	CPU(ﾎｽﾄ) ｲﾝﾃﾙ Xeon X5570	GPU(ﾃﾞﾊﾞｲｽ) NVIDIA Tesla C1060
コア数	8個	単精度240(=8×30)個 倍精度 30(=1×30)個
ピーク性能 (全コアの合計)	93.76GFLops	単精度933.12GFlops 倍精度 77.76GFlops
メモリバンド幅	25.6GByte/s	102.4GByte/s
メモリ容量	24GByte	4GByte

図 1-2-1　マシン 1 台当たりの仕様

図 1-2-1 のマシン (1 台分) を図 1-2-2 に示します。RICC の GPU (NVIDIA 社の Tesla C1060) は、図 1-2-2 に示すように、30 個のストリーミング・マルチプロセッサー (略して SM) から構成されています。各ストリーミング・マルチプロセッサーには、CUDA コアと呼ばれる演算装置が、単精度用に 8 個、倍精度用に 1 個含まれています。なお、CUDA コアは、以前はスカラープロセッサー、あるいはストリーミングプロセッサー (SP) と呼ばれていました。

CUDA では、CPU 側をホスト、GPU 側をデバイスと呼びます。図 1-2-1 の下線部に示すように、デバイス側の単精度のピーク性能（全コアの合計）は、ホスト側のピーク性能の約 10 倍になっています。

メモリはホスト側とデバイス側にそれぞれ存在し、PCI-Express で接続されています。デバイス側のメモリは、低速なオフチップメモリと、高速なオンチップメモリから構成されています。図 1-2-2 に示すように、ホスト側の配列 A は、いったんデバイス側のオフチップメモリ上の配列 dA にコピーしてから、デバイス側で処理を行います（詳細は後述します）。

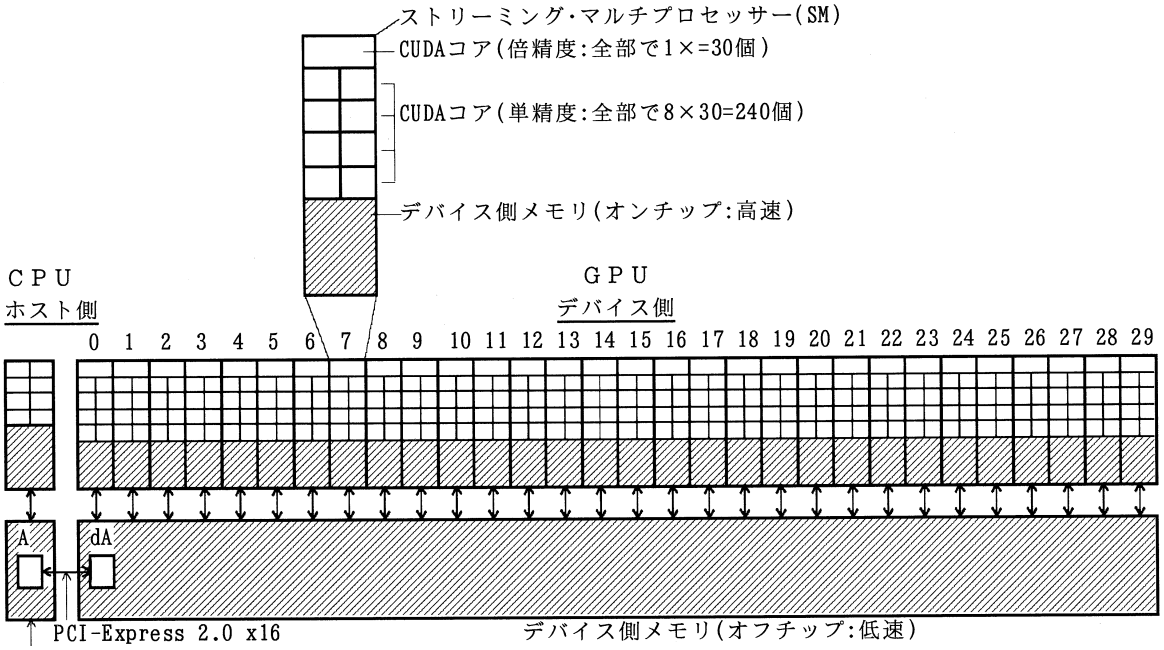


図 1-2-2

Fermi の新機能

参考のため、図 1-1-1 に示した新アーキテクチャ Fermi の主要な機能の概要を、下記に示します（アーキテクチャーの仕様そのものは省略します)。RICC の GPU は GT200 アーキテクチャーなので、下記の機能は使用できないことに注意して下さい。

- GPU は元々画像処理に使用されており、倍精度は不要でしたが、倍精度が要求される HPU などの分野に対応するため、GT200 から、倍精度の機能が付加されました。ただし GT200 では、コア数は単精度の 1/8、ピーク性能は単精度の 1/12 です。Fermi では、ピーク性能が単精度の 1/2 に改善されました。
- デバイス側のメモリに L1、L2 キャッシュが搭載され、プログラムの高速化が容易になります (GT200 では搭載されていません)。
- デバイス側のプログラムで、再帰、関数ポインター、`printf` 文（デバッグに便利）が使用できます (GT200 では使用できません)。
- デバイスメモリ、シェアードメモリ、L1/L2 キャッシュ、レジスターに対して ECC（注）機能が付き、信頼性が向上しました (GT200 では ECC 機能はサポートされていません)。

（注）ECC (Error Correcting Code または Error Check and Correct): 宇宙線がメモリに降りかかると、メモリ内の電子が移動し、メモリーエラーを引き起こすことがあります。ECC 機能が付いたメモリでは、データをメモリに書き出す際、本来のデータの他にエラーチェック用のデータも書き出します。そしてメモリーからデータを読み込んだ時に、エラーチェック用のデータを使用して、メモリーエラーが発生したかどうかをチェックします。64 ビット当たり、1 ビットの誤りは自動的に訂正し、2 ビットの誤りは検出のみ可能です。

GPU は元々画像処理用に使用されており、多少ビットエラーが発生しても画像への影響が少ないため、ECC の機能は付いていませんでしたが、HPC などの信頼性が重視される分野に対応するため、ECC の機能が付きました。RICC の場合は GT200 なので ECC の機能がサポートされていません。メモリーエラーがどの程度の頻度で発生するのか分かりませんが、重要な計算の場合、例えば同じ計算を 2 回実行し、結果が完全に一致しているかをチェックするなどの対策を取った方がよいかもしれません（この方法で回避可能かどうかは不明です)。

本書で取り上げなかった機能

本書では、Compute Capability 1.3 で提供されている主要な機能を説明します。以下の機能の説明は割愛しました。

- CUDA で提供されている関数（例えば、ホスト側のメモリからデバイス側のメモリにデータをコピーする関数）には、以下の 2 種類があります。本書ではこれらを CUDA 関数と呼ぶことにします（正式な用語ではありません)。本書では、(1) の中の主要な関数のみを説明します。(2) は (1) よりも低レベルの関数で、使用方法が複雑なので、本書では説明しません。全 CUDA 関数と、各関数の引数の詳細については「CUDA Reference Manual」（付録参照）を参照してください。
  (1) CUDA ランタイム API (Application Programming Interface)
  (2) CUDA ドライバー API
- テクスチャーメモリーは、デバイス側で使用する、読み込みのみのメモリです。使用方法が特殊なので、本書では説明しません。
- 同期に関係する、`__threadfence()` という CUDA 関数が提供されていますが、使用方法が分からないので説明を省略しました。詳細は「CUDA C Programming Guide」(B.5 節) を参照して下さい。
- CUDA では、右図の `float2` のように、ビルトイン変数と呼ばれる構造体のデータ型が用意されており (`int1`〜`int4`, `float1`〜`float4`, `double1`, `double2` など)、ホスト側とデバイス側のプログラムで使用できますが、本書では説明を省略します。

```
float2 a,b,c;
a.x = 1.0f; a.y = 2.0f;
b.x = 3.0f; b.y = 4.0f;
c = a + b; ← この使い方は不可
```

1-3　簡単は CUDA のプログラム例

本節では、CUDA のプログラミングを概観するため、簡単なプログラムを CUDA 用に変更し (以後 CUDA 化すると言います)、実行する方法を説明します。

元のプログラム

図 1-3-1 (1) のプログラムでは、図 1-3-1 (2) に示すように、大きさ 4 の配列 A に対し、①で値を設定し、②で 1.0 を加え、③で図 1-3-1 (3) の結果を書き出しています。

```
#include <stdio.h>
#include <stdlib.h>
#define N (4)
int main(void){
  int i;
  float A[N];
  for(i=0;i<N;i++){
    A[i] = (float)i;            ①
  }
  for(i=0;i<N;i++){
    A[i] = A[i] + 1.0f;         ②
  }
  for(i=0;i<N;i++){
    printf("%d %f¥n",i,A[i]); ③
  }
  return(0);
}
```

図 1-3-1 (1)

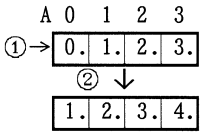


図 1-3-1 (2)

③ 実行結果

```
0 1.0000
1 2.0000
2 3.0000
3 4.0000
```

図 1-3-1 (3)

CUDA 化したプログラム風のホストプログラム

このプログラムを、いきなり CUDA 化すると分かりにくいので、まず、CUDA 化したプログラム風の、ホスト用のプログラムに書き替えます（従って、処理に無駄な部分があります)。これを図 1-3-2 (1) に示し、動作を図 1-3-2 (2) に示します。

図 1-3-1 (1) との相違部分を以下に示します。

- 図 1-3-1 (1) の②のループの計算部分を分離し、⑤の関数 kernel 内で計算するようにします。
- ⑤では、②の配列 A とは異なる配列を使用して計算を行います。そのため、⑧，⑨，⑩で、配列 A と同じ大きさの配列 dA を malloc で確保します (図 1-3-2 (2) 参照)。
- ⑪で配列 A のデータを配列 dA にコピーし、⑬の引数で⑤に渡します。本例では、⑬の配列 dA を、⑤でも同じ名前で使用しますが、異なる名前 (例えば配列 A) で使用しても、もちろん構いません。
- 図 1-3-1 (1) の②のループカウンター i を、⑫に示すように、threadIdx という名前の変数に変更します。ただし関数 kernel 内の⑦では変数 i を使用するため、④で変数 threadIdx をグローバル変数として宣言して threadIdx の値を関数 kernel に渡し、⑥で変数 i に代入します。
- ⑦と②で同じ計算を行い、計算結果が配列 dA に入ります。
- ⑭で、計算結果の入った配列 dA を、配列 A にコピーします。
- ⑮で配列 dA を解放します。

CUDA 化したプログラム

次に、図 1-3-2 (1) を CUDA 化した、図 1-3-3 (1) のプログラムを説明します。また動作を図 1-3-3 (2) と図 1-3-4 に示します。なお、例えば図 1-3-2 (1) の⑩と図 1-3-3 (1) の (10) は対応しています。

- 図 1-3-2 (1) では、図 1-3-2 (2) に示すように、配列 A と dA はともにホスト側のメモリに置きましたが、図 1-3-3 (1) では、図 1-3-3 (2) と図 1-3-4 に示すように、配列 dA はデバイス側のメモリに置きます。デバイス (device) 側メモリ上の配列であることを示すため、配列 dA の先頭を「d」としています。
- 図 1-3-2 (1) の⑧，⑨，⑩に対応する、デバイス側の配列 dA の確保を、図 1-3-3 (1) の (8)，(9)，(10) で、CUDA 関数「cudaMalloc」を使用して行います (各 CUDA 関数の引数の詳細は、「CUDA Reference Manual」（付録参照）を参照して下さい)。
- ホスト側のプログラムから、デバイス側の配列 dA に直接値を代入することはできません。同様に、デバイス側のプログラムから、ホスト側の配列 A に直接値を設定することはできません。そこでまず (11) で、CUDA 関数「cudaMemcpy」を使用して、ホスト側の配列 A の内容をデバイス側の配列 dA にコピーします。
  cudaMemcpy は、以下の〓に示すように、2 つ目の引数の内容を 1 つ目の引数にコピーします。引数 size にはコピーする大きさ（バイト）を指定します。本例では (9) で指定していますが、(11) の引数で直接指定しても、もちろん構いません。また最後の引数には、コピーの方向に応じて下記の下線部を指定します。
  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);（ホスト側の A からデバイス側の dA へコピー）
  〓
  cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);（デバイス側の dA からホスト側の A へコピー）
  cudaMemcpy(dB,dA,size,cudaMemcpyDeviceToDevice);（デバイス側の dA からデバイス側の dB へコピー）
  cudaMemcpy(B,A,size,cudaMemcpyHostToHost);（ホスト側の A からホスト側の B へコピー）
- 図 1-3-2 (1) の⑫のループの N (= 4) 回の反復を、4 つの CUDA コアで配列計算させることにします (つまりループの 1 反復を 1 つの CUDA コアが計算します)。CUDA コアで実行される、図 1-3-3 (1) の (5) の関数をカーネル関数と呼び、(5) の〓に示すように、関数名の前に「__global__」を指定します (global の前後の下線 (_) は各 2 文字です)。
- (13) の <<<...>>> の部分は、英語で「execution configuration」という名称ですが、本書では実行構成と呼びます。実行構成では、カーネル関数をどのような構成で実行するかを指定します（詳細は後述します)。(13) では、CUDA コア上で、(5) のカーネル関数 kernel を N (= 4) 個実行することを指示しています。
  (13) を実行すると、図 1-3-4 に示すように、カーネル関数が、同じストリーミング・マルチプロセッサー内の 4 個の CUDA コアで並列に実行されます (どの CUDA コアで実行されるかは特に意味がありません)。各 CUDA コアで実行される各カーネル関数のことをスレッドと呼びます。従って (13) は「カーネル関数 kernel を CUDA コア上で 4 個のスレッドで実行せよ。」という意味になります。
  なお、CUDA マニュアル「CUDA C Programming Guide」（付録参照）などの図中で、スレッドを「〓」と表現することがあるので、本書でも図 1-3-4 に示すように「〓」を使用します。
- 実行を開始した 4 つのスレッドには、自動的にスレッド ID と呼ばれる識別子が付きます。スレッド数が 4 つの場合、スレッド ID は0, 1, 2, 3のいずれかになります。各スレッドに付けられたスレッド ID は、図 1-3-3 (1) の (4) に示すように、変数 threadIdx.x (正確には構造体 threadIdx のメンバー x) に設定されます。前述の図 1-3-2 (1) の④の変数 threadIdx は、プログラム内でユーザーが指定した変数なので名前は可変ですが、(4) の変数 threadIdx は CUDA が提供しており（従って変数の宣言は不要)、常にこの名前になります。
  threadIdx.x の読み方は「スレッドインデックス. エックス」（エックスの意味は後述します）で、「I」は大文字の「アイ」です。
- 本例では、図 1-3-3 (2) に示すように、配列 dA の要素番号 i (1，2，3，4) と、スレッド ID (0, 1, 2, 3) がちょうど 1 対 1 に対応するので、図 1-3-3 (1) の (6) に示すように、スレッド ID を変数 i に代入します。なお、変数 i は、図 1-3-4 に示すように、スレッドごとに別のレジスター内に置かれます。

- 図 1-3-3 (1) の (7) で、各スレッドは、配列 dA の自分が担当する要素に 1.0f を加算します。加算の際、図 1-3-4 の (7) に示すように、スレッドごとに異なるレジスターが使用されます。なお、図 1-3-3 (1) の (7) は、変数 i を使用せずに、dA[threadIdx.x] = da[threadIdx.x] + 1.0f; としても構いません。
- 図 1-3-3 (1) の (13) の計算が終了したら、(14) で、デバイス側の配列 dA をホスト側の配列 A にコピーします。最後に、(15) の CUDA 関数を使用して配列 dA を解放します。

```
#include <stdio.h>
#include <stdlib.h>
#define N (4)
  int threadIdx;                                  ④
void kernel(float *dA){                           ⑤




  int i = threadIdx;                              ⑥
  dA[i] = dA[i] + 1.0f;                           ⑦
}
int main(void){
  int i;
  float A[N];
  float *dA;                                      ⑧
  for(i=0;i<N;i++){
    A[i] = (float)i;
  }
  size_t size = N*sizeof(float);                  ⑨
  dA = (float*)malloc(size);                      ⑩
  for(i=0;i<N;i++){
    dA[i] = A[i];                                 ⑪
  }
  for(threadIdx=0;threadIdx<N;threadIdx++){       ⑫
    kernel(dA);                  ↰ 4              ⑬
  }
  for(i=0;i<N;i++){
    A[i] = dA[i];                                 ⑭
  }
  free(dA);                                       ⑮
  for(i=0;i<N;i++){
    printf("%d %f¥n",i,A[i]);
  }
  return(0);
}
```

図 1-3-2 (1)

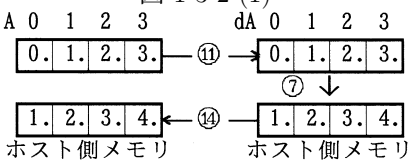


図 1-3-2 (2)

```
#include <stdio.h>
#include <stdlib.h>
#define N (4)
                    ↰カーネル関数
__global__ void kernel(float *dA){ (5) スレッド
  threadIdx.x(ｽﾚｯﾄﾞID)に、
  0,1,2,3のいずれかが                            (4)
  自動的に設定される。
  int i = threadIdx.x;                            (6)
  dA[i] = dA[i] + 1.0f;                           (7)
}
int main(void){
  int i;
  float A[N];
  float *dA;                                      (8)
  for(i=0;i<N;i++){
    A[i] = (float)i;
  }
  size_t size = N*sizeof(float);                  (9)
  cudaMalloc((void**)&dA,size);                   (10)

  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);
                                                  (11)
  kernel<<<1,N>>>(dA);  ｽﾚｯﾄﾞ数N(=4)個で          (13)
                        ｶｰﾈﾙ関数を実行する。

  cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);
                                                  (14)
  cudaFree(dA);                                   (15)
  for(i=0;i<N;i++){
    printf("%d %f¥n",i,A[i]);
  }
  return(0);
}
```

図 1-3-3 (1)

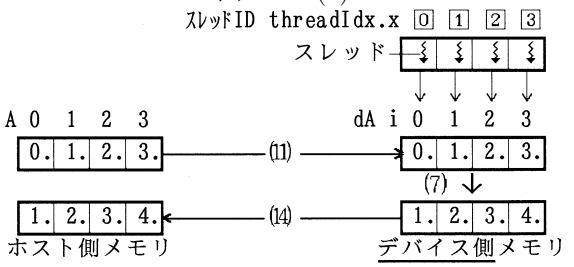


図 1-3-3 (2)

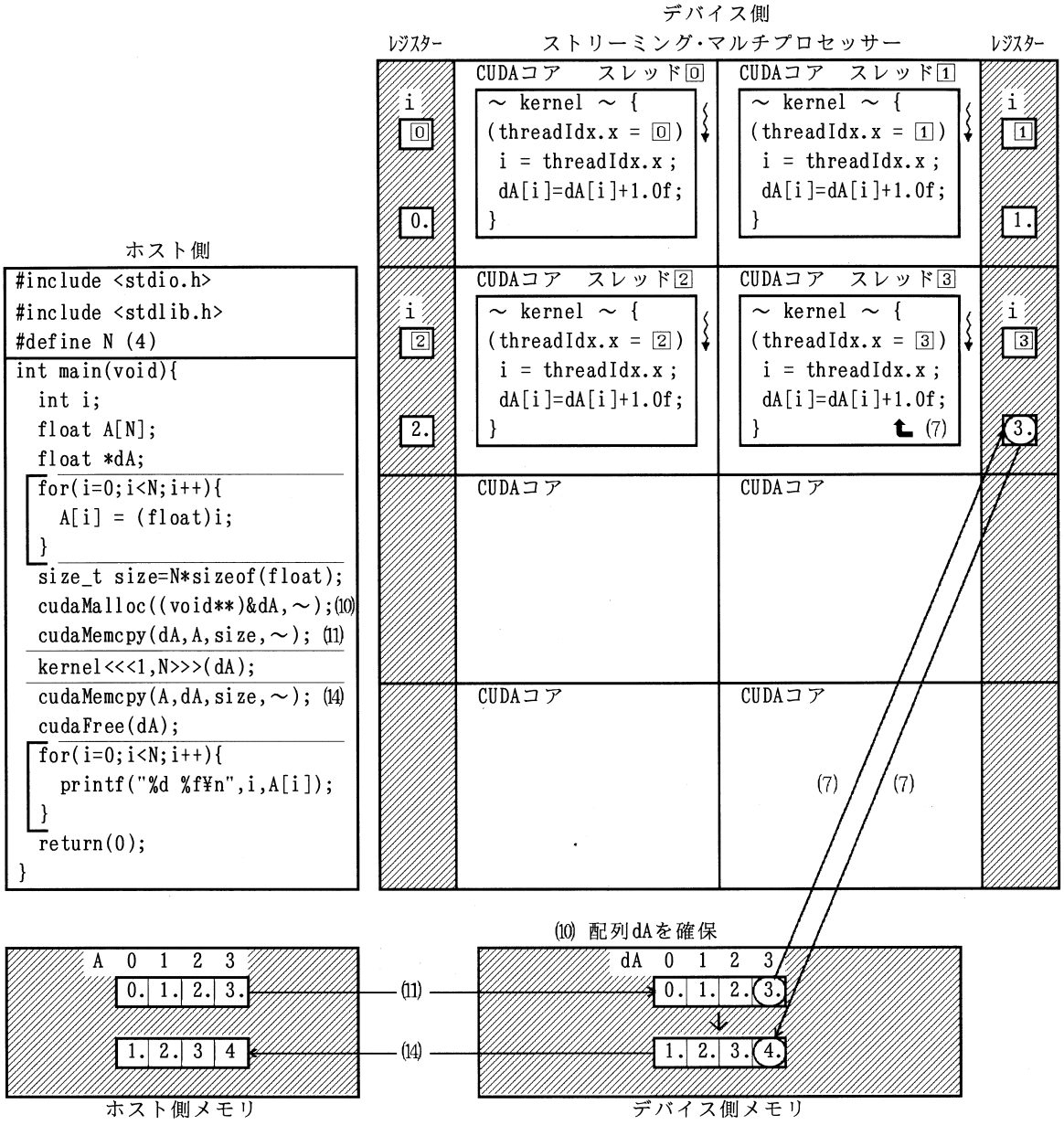

デバイス側
レジスター
ストリーミング・マルチプロセッサー
レジスター
CUDAコア　スレッド0
CUDAコア　スレッド1
CUDAコア　スレッド2
CUDAコア　スレッド3
～ kernel ～ {
(threadIdx.x = 0)
(threadIdx.x = 1)
(threadIdx.x = 2)
(threadIdx.x = 3)
i = threadIdx.x ;
dA[i]=dA[i]+1.0f;
}
(7)
CUDAコア
ホスト側
#include <stdio.h>
#include <stdlib.h>
#define N (4)
int main(void){
int i;
float A[N];
float *dA;
for(i=0;i<N;i++){
A[i] = (float)i;
}
size_t size=N*sizeof(float);
cudaMalloc((void**)&dA,～);(10)
cudaMemcpy(dA,A,size,～); (11)
kernel<<<1,N>>>(dA);
cudaMemcpy(A,dA,size,～); (14)
cudaFree(dA);
for(i=0;i<N;i++){
printf("%d %f¥n",i,A[i]);
}
return(0);
}
(10) 配列dAを確保
A 0 1 2 3
dA 0 1 2 3
(11)
(14)
ホスト側メモリ
デバイス側メモリ

図 1-3-4

CUDA 化したプログラムを RICC でコンパイル／リンク／実行する方法

図 1-3-3 (1) のプログラムを、理研の RICC でコンパイル／リンクして実行する方法を説明します。詳細は「RICC 利用手引書」を参照して下さい。

- まず RICC のログインサーバーにログインします。
- 図 1-3-5 の①に示すように、GPU コンパイルサーバー (accel) にログインします。
- CUDA プログラムのファイル名の末尾を「〜.cu」とします。例えば、図 1-3-3 (1) のプログラムのファイル名を sample.cu とします
- ②に示すように、nvcc コマンドを用いて CUDA プログラムのコンパイル／リンクを行い、ロードモジュール a.out を作成します (C 言語のコンパイラ gcc が内部的に使用されます)。nvcc コマンドの最適化オプションの詳細は 2-7 節で説明します（何も指定しなくても実行は可能です)。
- 図 1-3-6 に示すようなシェルスクリプトを、例えばファイル go.sh に作成し、③でジョブを投入します。なお、以下では GPU コンパイルサーバーからジョブを投入しましたが、ログインサーバーからでもジョブを投入することができます。インタラクティブジョブとして実行することはできません。
- GPU を使用するジョブは、ホスト側のプログラムが MPI 並列やスレッド配列になっていなくても、④に示すように、ノード内の 8 コアを全て占有します。従って、8 コア分の演算時間を消費したことになりますので、注意して下さい。
- ⑤のコマンドで、RICC のコア利用状況を表示することができます。このうち下線部 (upc) が、GPU が導入されている多目的 PC クラスターの利用状況です。

```
$ ssh accel  ↵                          ①
Last login: Sun Jan  1 00:00:00 2010 from ricc2
$ nvcc (最適化ｵﾌﾟｼｮﾝ) sample.cu  ↵   ②
$ qsub go.sh  ↵                         ③
Request 1111111.jms submitted to MJS.
NOTICE: 1111111.jms use node(8 cores) exclusively to consume more resource than default.
$ qstat  ↵                              ④
[A10007] Staff (Advanced center for Computing and Communication)
REQID              NAME                          STAT     ELAPSE START-TIME  CORE
--------------------------------------------------------------------------------
1111111.jms        go.sh                         QUE       --:-- --/-- --:--    8
$ qstat -uc  ↵                          ⑤
The status of CORE use                              RATIO(USED/ALL)
--------------------------------------------------------------------------------
 mpc         ********************--------------------    98.5%(4060/8120)
 upc         ****************************************   100.0%(0800/0800)
 mdg         ----------------------------------------     0.0%(0000/0256)
 ax          ----------------------------------------     0.0%(0000/0032)
```

図 1-3-5

go.sh

```
#!/bin/sh
#MJS: -accel    ←(必須) 多目的PCクラスタでGPUを使用する場合に指定します。
#MJS: -cwd      ←(任意) ジョブが開始すると、ジョブを投入したディレクトリに移動します。
#MJS: -eo       ←(任意) 標準出力と標準エラー出力を合体して出力します。
#MJS: -time 30  ←(任意) ジョブ打切り経過時間(秒)を指定します。
srun a.out              時間を短く指定した方が、一般にジョブが早く開始します。
                        「-time 12:34」だと12分34秒、「-time 12:34:56」だと12時間34分56秒です。
                        指定しない場合のデフォルトは、128コアまでの場合、72時間です。
```

図 1-3-6

1-4　CUDA SDK の導入

CUDA Toolkit に含まれる CUDA SDK (Software Development Kit) には、CUDA プログラムのサンプルコードおよび解説や、ユーティリティーなどが含まれています。本書の以後の説明中で参照することがありますので、導入方法を説明します。なお、以下のサイトに、サンプルコードの簡単な説明があります。

http://www.nvidia.co.jp/object/cuda_get_samples_jp.html

http://www.nvidia.co.jp/object/cuda_sdks_jp.html

- RICC のログインサーバーにログインした後、①で GPU コンパイルサーバー (accel) にログインします。
- 以下では、CUDA SDK を、ホームディレクトリ ($HOME) の下に作成する場合を想定します。②〜⑤を実行すると、$HOME/NVIDIA_GPU_Computing_SDK/ というディレクトリ内に、CUDA SDK が作成されます。
- ⑥，⑦を実行すると、CUDA SDK に含まれるサンプルプログラムのコンパイル／リンクが行われます。
- ⑧は CUDA SDK に含まれているサンプルプログラム（のディレクトリ名）の一覧です。名前から、プログラムの機能がある程度想像できると思います。また、CUDA のある機能の使用方法が分からない場合、これらのプログラム内で、その機能を使用している箇所を検索し、参考にする事ができます。例えば、「__constant__」（配列をコンスタントメモリ上に確保する指定：後述）の使用例を探したい場合、以下のようにします。

```
$ cd $HOME/NVIDIA_GPU_Computing_SDK/C/src
$ fgrep "__constant__" */*.c*
```

```
$ ssh accel ↵                                          ①
Last login: Sun Jan  1 00:00:00 2011 from ricc2
$ cp /usr/local/cuda/gpucomputingsdk_3.2.16_linux.run . ↵  ②
$ sh gpucomputingsdk_3.2.16_linux.run ↵                    ③
(メッセージが出力されます)
Enter install path (default ~/NVIDIA_GPU_Computing_SDK): ↵  ④ (Enterのみ)
(メッセージが出力されます)
Enter CUDA install path (default /usr/local/cuda): ↵       ⑤ (Enterのみ)
(メッセージが出力されます)
$ cd $HOME/NVIDIA_GPU_Computing_SDK/C ↵                    ⑥
$ make ↵                                                   ⑦
(多数のメッセージが表示されます)
$ ls $HOME/NVIDIA_GPU_Computing_SDK/C/src ↵                ⑧
BlackScholes        conjugateGradient     mergeSort              simplePitchLinearTexture
FDTD3d              convolutionFFT2D      nbody                  simplePrintf
FunctionPointers    convolutionSeparable  oceanFFT               simpleStreams
Interval            convolutionTexture    particles              simpleSurfaceWrite
Mandelbrot          cppIntegration        postProcessGL          simpleTemplates
MersenneTwister     dct8x8                ptxjit                 simpleTexture
MonteCarlo          deviceQuery           quasirandomGenerator   simpleTexture3D
MonteCarloCURAND    deviceQueryDrv        radixSort              simpleTextureDrv
MonteCarloMultiGPU  dwtHaar1D             randomFog              simpleVoteIntrinsics
SobelFilter         dxtc                  recursiveGaussian      simpleZeroCopy
SobolQRNG           eigenvalues           reduction              smokeParticles
alignedTypes        fastWalshTransform    scalarProd             sortingNetworks
asyncAPI            fluidsGL              scan                   template
bandwidthTest       histogram             simpleAtomicIntrinsics threadFenceReduction
bicubicTexture      imageDenoising        simpleCUBLAS           threadMigration
bilateralFilter     lineOfSight           simpleCUFFT            transpose
binomialOptions     marchingCubes         simpleGL               vectorAdd
boxFilter           matrixMul             simpleMPI              vectorAddDrv
clock               matrixMulDrv          simpleMultiCopy        volumeRender
concurrentKernels   matrixMulDynlinkJIT   simpleMultiGPU
```

図 1-4-1

図 1-4-1 の下線の「deviceQuery」を例に、サンプルプログラムの実行方法を説明します。

- 図 1-4-1 の⑦の作成されたロードモジュールが入っている、図 1-4-2 の①のディレクトリに移動します。
- 図 1-4-3 のようなシェルスクリプトを作成し、④で図 1-4-1 の任意のプログラム名を指定します。③のジョブ打切り経過時間は、適当に設定します（短い方が、一般にジョブが早く開始します）。
- 「deviceQuery」は、そのシステムに導入されている GPU の仕様を表示するプログラムです。②を実行すると、標準出力に図 1-4-4 が作成され、⑤以下に GPU の仕様が表示されます (⑥以下は無視して下さい)。
- 各仕様は、「CUDA C Programming Guide」の G.1 節にも記載されていますので、参照して下さい。

```
$ cd $HOME/NVIDIA_GPU_Computing_SDK/C/bin/linux/release ↵  ①
$ qsub go.sh ↵                                             ②
```

図 1-4-2

go.sh

```
#!/bin/sh
#MJS: -accel
#MJS: -cwd
#MJS: -eo
#MJS: -time 30    ③
srun deviceQuery ④
```

図 1-4-3

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

There are 2 devices supporting CUDA

Device 0: "Tesla C1060"  ⑤
  CUDA Driver Version:                           3.20          (CUDAﾄﾞﾗｲﾊﾞｰAPIのﾊﾞｰｼﾞｮﾝ)
  CUDA Runtime Version:                          3.20          (CUDAﾗﾝﾀｲﾑAPIのﾊﾞｰｼﾞｮﾝ)
  CUDA Capability Major/Minor version number:    1.3           (Compute Capability 1.3)
  Total amount of global memory:                 4294770688 bytes (ｸﾞﾛｰﾊﾞﾙﾒﾓﾘの大きさは4Gﾊﾞｲﾄ)
  Multiprocessors x Cores/MP = Cores:            30 (MP) x 8 (Cores/MP) = 240 (Cores)
                                    (ｽﾄﾘｰﾐﾝｸﾞﾏﾙﾁﾌﾟﾛｾｯｻｰを30個、CUDAｺｱを合計240(=30×8)個搭載)
  Total amount of constant memory:               65536 bytes   (ｺﾝｽﾀﾝﾄﾒﾓﾘの大きさは64Kﾊﾞｲﾄ)
  Total amount of shared memory per block:       16384 bytes (ﾌﾞﾛｯｸ(注)あたりのｼｪｱｰﾄﾞﾒﾓﾘは16Kﾊﾞｲﾄ)
  Total number of registers available per block: 16384         (ﾌﾞﾛｯｸ(注)あたり利用可能な
                                                               ﾚｼﾞｽﾀｰの個数は16K個)
  Warp size:                                     32            (ﾜｰﾌﾟあたりのｽﾚｯﾄﾞ数は32個)
  Maximum number of threads per block:           512           (ﾌﾞﾛｯｸあたり最大512ｽﾚｯﾄﾞ)
  Maximum sizes of each dimension of a block:    512 x 512 x 64    (ﾌﾞﾛｯｸ内のｽﾚｯﾄﾞの最大の個数)
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1 (ｸﾞﾘｯﾄﾞ内のﾌﾞﾛｯｸの最大の個数)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.30 GHz
  Concurrent copy and execution:                 Yes       (ﾃﾞｰﾀのｺﾋﾟｰとｶｰﾈﾙ関数の同時実行が可能)
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes       (ﾎｽﾄﾍﾟｰｼﾞﾛｯｸﾒﾓﾘﾏｯﾋﾟﾝｸﾞがｻﾎﾟｰﾄされている)
  Compute mode:                                  Default (multiple host threads can use
                                                                 this device simultaneously)
  Concurrent kernel execution:                   No        (複数のｶｰﾈﾙ関数の同時実行はできない)
  Device has ECC support enabled:                No        (ECCﾁｪｯｸは行われない)
  Device is using TCC driver mode                No

Device 1: "Quadro NVS 290"  ⑥
    (以下略)
```

図 1-4-4　（注）「ブロックあたり」を「ストリーミング・マルチプロセッサーあたり」と読み替えても構いません。

1-5　多次元配列のメモリ上での配置

次章から、CUDA の説明に入ります。その前提知識として、CUDA では、メモリ上の各要素をアクセスする順序が速度に影響を与えるので、本節で基本項目をまとめます。

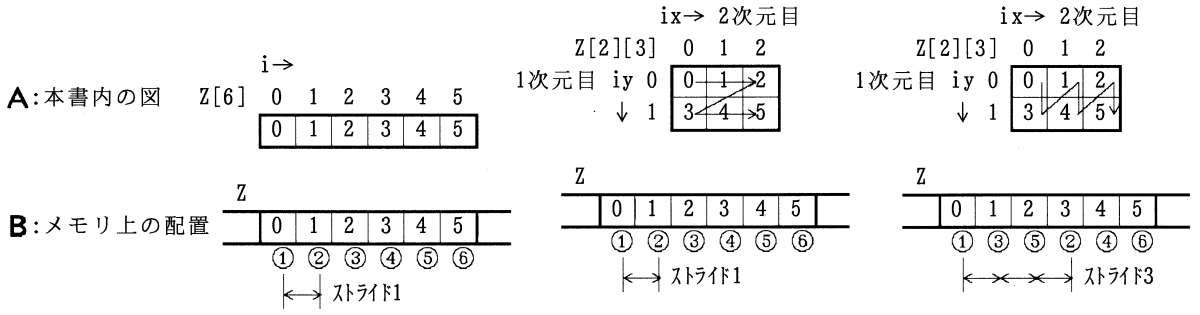
1 次元配列

まず 1 位次元配列 Z[6] のループの例を、図 1-5-1 (1) の C に示します。A は本書内での配列の図、B はメモリ上の配置です。1 次元配列の場合、A と B は同じ順序で並びます。C のループを実行した場合、B の①～⑥の順に要素がアクセスされます。1 つの要素と、次に処理する要素の間隔を、本書ではストライドと呼びます。図 1-5-1 (1) の B ではストライドは 1 です。

2 次元配列

次に 2 次元配列 Z[2][3] について説明します (3 次元以上の場合も同様です)。

- 本書では図 1-5-2 に示すように、配列 Z[2][3] の左側の添字を 1 次元目、右側の添字を 2 次元目と呼びます。
- 図 1-5-1 (2) の A に示すように、本書内の図は、2 次元配列の 1 次元目を縦方向、2 次元目を横方向とします。
- C 言語の場合、2 次元配列は、図 1-5-1 (2) の B に示す順番にメモリ上に並びます。つまり図 1-5-1 (2) の A の矢印の順番にメモリ上に並びます。なお、Fortran の場合は C 言語と並ぶ順番が逆になりますので注意して下さい。図 1-5-1 (2) の C のループでは、B の①～⑥の順にストライドが 1 で要素をアクセスします。
- 図 1-5-1 (2) とループの順番を逆にした図 1-5-1 (3) では、A の矢印の順に処理が行われ、B の①～⑥に示すようにストライドは 3 になります。ストライドの「3」は、配列 Z[2][3] の 2 次元目の大きさです。
  ホスト側のプログラムでは、図 1-5-1 (3) や図 1-5-3 のようにストライドが大きいループは、キャッシュミス (参考文献 [2] 参照) が発生して速度が遅くなります。なるべく図 1-5-1 (1) (2) のように、ストライドが 1 になるようにして下さい。デバイス側のプログラムについては、3 章で説明します。




```
float Z[6];
  :
for(i=0;i<6;i++){
  Z[i] = ～;
}
  :
```

図 1-5-1 (1)　○

```
float Z[2][3];
  :
for(iy=0;iy<2;iy++){
  for(ix=0;ix<3;ix++){
    Z[iy][ix] = ～;
  }
}
  :
```

図 1-5-1 (2)　○

```
float Z[2][3];
  :
for(ix=0;ix<3;ix++){
  for(iy=0;iy<2;iy++){
    Z[iy][ix] = ～;
  }
}
  :
```

図 1-5-1 (3)　×

```
float Z[2][3];
1次元目↓  →2次元目
```

図 1-5-2

```
float Z[6];
  :
for(i=0;i<6;i+=3){
  Z[i] = ～;
}
  :
```

図 1-5-3　×

第2章　基本編（カーネル関数の実行方法）

本章では、カーネル関数を実行するための方法、および実行に入った CUDA プログラムが、GPU 上でどのように動作するかについて説明します。

2-1　デバイス側の関数

デバイス側の関数型修飾子

図 2-1-1 の左はホスト側、右はデバイス側のプログラムです。

- 図 2-1-1 の関数 kernel のように、ホスト側から直接呼ばれる関数をカーネル関数と呼び、__global__ を指定します (前半と後半の「_」は各 2 文字)。__global__ などの〓に示す部分を関数型修飾子と呼びます。
- 図 2-1-1 の関数 sub のように、カーネル関数から呼ばれる関数には __device__ 修飾子を指定します。
- 図 2-1-1 の関数 com のように、ホスト側とデバイス側から同一の関数が呼ばれる場合、その関数には __host__ と __device__ 修飾子を指定します。関数 com をコンパイル／リンクすると、ホスト用とデバイス用の 2 つのロードモジュールが作成されます。例えば、CUDA 化したプログラムをテストするため、ホスト側とデバイス側で同じ計算を (同じ関数 com で) 実行し、計算結果を比較する場合などに使用します。

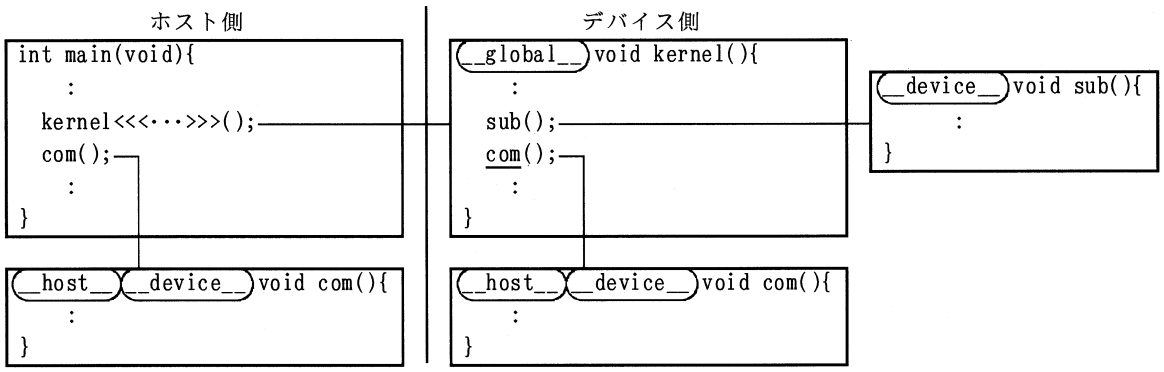


図 2-1-1

以下に、Compute Capability 1.3 の場合の、関数型修飾子を付けた関数の主な制限事項を示します。詳細は「CUDA C Programming Guide」の Appendix. B を参照して下さい。

__global__ または __device__ 修飾子を付けた関数の制限事項

- 関数内で、printf 関数や malloc 関数などを使用することはできません。
- 関数内で、static 変数 (注 1) を宣言することはできません。
- 関数内で、再帰呼び出し (注 2) を行うことはできません。
- 関数の引数に、可変長引数 (注 3) を持つことはできません。

__global__ 修飾子を付けた関数に固有の制限事項

- ホスト側のプログラムからカーネル関数が呼ばれると、制御はすぐに（カーネル関数が実行を開始する前に）ホスト側プログラムに戻ります (詳細は 3-2 節参照)。
- 関数からの戻り値は常に void 型になるので (図 2-1-1 参照)、戻り値を指定することはできません。
- この関数へのポインター (注 4) を利用することはできます。

__device__ 修飾子を付けた関数に固有の制限事項

- 戻り値は void 型でなくても構いません。
- この関数へのポインター (注 4) を利用することはできません。

(注 1) static 変数

図 2-1-2 の①に示す通常の変数は、関数が呼ばれるたびにメモリ上に確保、初期化され、関数が終了すると解放されます。一方②に示す static 関数は、プログラムが開始した時点でメモリ上に固定され、初期化は一度だけ行われます。関数が終了しても変数の値はそのまま保持されます（実行結果参照）。

(注 2) 再帰呼び出し

図 2-1-3 のように、ある関数内でその関数を再度呼び出すことを再帰呼び出しと言います。

```
void func(){
        int  i=0;          ①
  static int si=0;         ②
   i =  i + 1;             ①
  si = si + 1;             ②
  printf("i=%d,si=%d¥n",i,si);
}
```

```
int main(void){
  func();
  func();
  func();
    :
```

実行結果

```
i=1,si=1
i=1,si=2
i=1,si=3
```

図 2-1-2

```
void func(int N){
  printf("%d¥n",N);
  if(N<5) func(N+1);
}
int main(void){
  func(1);
    :
```

実行結果

```
1
2
3
4
5
```

図 2-1-3

(注 3) 可変長引数

例えば printf 関数は、図 2-1-3 に示すように、引数の数は可変です。これを可変長引数と呼びます。可変長引数の関数の例を図 2-1-4 に示します。関数 sum は、①, ②の 1 つ目の引数で、2 つ目以降の引数の数を指定し、2 つ目以降の引数の合計が戻ります。①では引数が 3 個、②では引数が 4 個なので、可変長引数です。

```
printf("%d¥n",m);
printf("%d %d¥n",m,n);
```

図 2-1-3

```
#include <stdarg.h>
int sum(int N, ...){
  int i;
  int result = 0;
  va_list list;
  va_start(list,N);
  for(i=0;i<N;i++){
    result = result
           + va_arg(list,int);
  }
  va_end(list);
  return result;
}
```

```
int main(){
  int N,result;
  N = 2; (加算する個数は2個)
  result = sum(N,10,20);     ①
  printf("N=%d result=%d¥n",N,result);
  N = 3; (加算する個数は3個)
  result = sum(N,10,20,30); ②
  printf("N=%d result=%d¥n",N,result);
    :
```

実行結果

```
N=2 result=30
N=3 result=60
```

図 2-1-4

(注 4) 関数へのポインター

図 2-1-5 (1) で、変数 p_func は関数 func へのポインターです。①でポインターの宣言をし、②で関数 func のアドレスを p_func に代入し、③で p_func を使用して関数 func を呼び出しています。図 2-1-5 (2) のように、CUDA のカーネル関数に対しても、この機能を使うことができます。

```
void func(){
  :
int main(void){
  void (*p_func)(); ①
  p_func = func;    ②
  (*p_func)();      ③
    :
```

図 2-1-5 (1)

```
__global__ void kernel(){
  :
int main(void){
  void (*p_kernel)();
  p_kernel = kernel;
  (*p_kernel)<<<1,1>>>();
    :
```

図 2-1-5 (2)

2-2　CUDA と他の並列化手法の比較

計算機で計算を行なう場合、計算を行う主体はプロセスやスレッド、計算する対象は配列や変数、計算を行う装置は CPU やコアです。この 3 つの要素の関係が、逐次処理、CUDA 以外の並列処理、および CUDA の並列処理でどのようになっているかを検討します。

逐次処理

図 2-2-1 のループを、1 台の CPU を使って逐次処理する場合を図 2-2-2 に示します。なお、1 CPU に複数のコアを含むマルチコアプロセッサーの場合、図中の「CPU」は「コア」に読み替えて下さい。

- A（プロセスと配列の関係）：1 つのプロセスが配列 A の 128 要素を担当します。
- B (プロセスと CPU の関係): 1 つのプロセスが 1 つの CPU で実行します。

MPI 並列

分散メモリ型並列計算機（共有メモリ型並列計算機でも多くの場合可）では、通常、MPI (Message-Passing Interface) を使用して並列化します。MPI 並列では CPU 数が 2 個の場合、動作は図 2-2-3 のようになります。

- A（プロセスと配列の関係）：プロセス（またはランク）0 と 1 が、それぞれ 64 要素ずつ担当します。
- B (プロセスと CPU の関係):（例えば）プロセス 0 は CPU 0 で、プロセス 1 は CPU 1 で実行します。

スレッド並列

共有メモリ型並列計算機では、通常、指示行 OpenMP またはコンパイラによる自動並列（これらをスレッド並列と呼びます）で並列化します。スレッド並列で CPU 数が 2 個の場合、動作は図 2-2-4 のようになります。

- A（プロセスと配列の関係）：スレッド 0 と 1 が、それぞれ 64 要素ずつ担当します。
- B (プロセスと CPU の関係):（例えば）プロセス 0 は CPU 0 で、スレッド 1 は CPU 1 で実行します（途中で入れ替わる場合もあります）。

```
    :
 float A[128];
 for(i=0;i<128;i++){
   A[i] = A[i] + 1.0f;
 }
    :
```

図 2-2-1

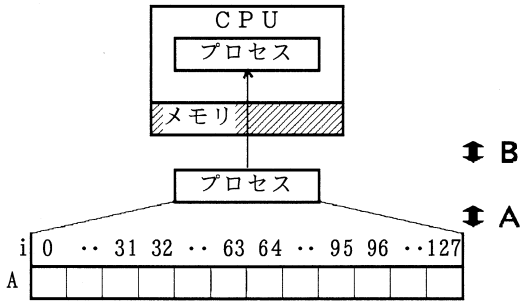


図 2-2-2　逐次処理

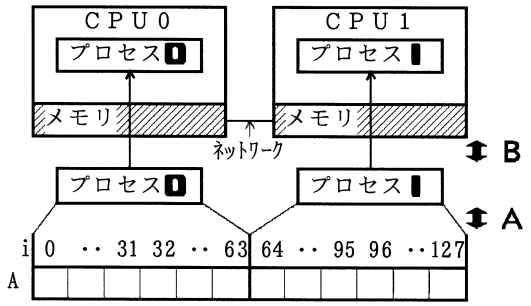


図 2-2-3　MPI 並列

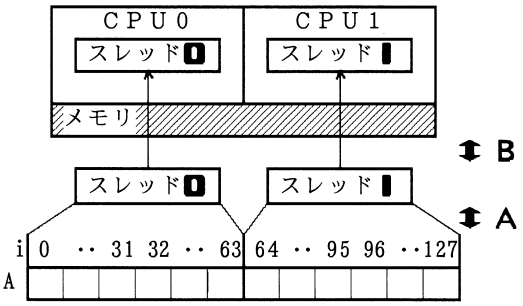


図 2-2-4　スレッド並列

CUDA 並列

図 2-2-1 を CUDA 化し、GPU 上で実行する場合の動作を図 2-2-5 に示します。1 つの GPU に含まれるストリーミング・マルチプロセッサーの数は、(理研 RICC の場合) 30 個ですが、説明を簡単にするため 2 個だとします。

- 前述のように、計算を実行する主体は、MPI 並列ではプロセス、スレッド並列ではスレッドです。CUDA 並列の場合もスレッドですが、少々複雑です。図 2-2-5 に示すように、複数のスレッドの集合をブロック、複数のブロックの集合をグリッドと呼びます。各スレッド、各ブロックには、0，1，2，...の ID（識別子）が自動的に付きます。図 2-2-5 の場合、1 つのグリッドが、(〓に示す) 4 個のブロック (ブロック ID 0，1，2，3) を含み、1 つのブロックが、(〓に示す) 32 個のスレッド (スレッド ID 0～31) を含みます。
- A（スレッドと配列の関係）：図 2-2-5 の ↓ に示すように、（通常）1 つのスレッドが配列 A の 1 つの要素を担当します (1 つのスレッドが複数要素を担当することも可能です)。例えば、ブロック ID 1 のスレッド ID 31は、A[63] の要素を担当します (詳細は 2-3 節参照)。
- B (スレッドと CUDA コアの関係): 図 2-2-5 の ↗ と ↖ に示すように、例えば、ブロック ID 0，1 に含まれる各スレッドが、左のストリーミング・マルチプロセッサー内の CUDA コアで実行し、ブロック ID 2，3 に含まれる各スレッドが、右のストリーミング・マルチプロセッサー内の CUDA コアで実行します。全 CUDA コア数は 16 個で、全スレッド数は 128 個なので、1 つの CUDA コアで合計 8 つのスレッドが実行を行います (詳細は 2-4 節参照)。

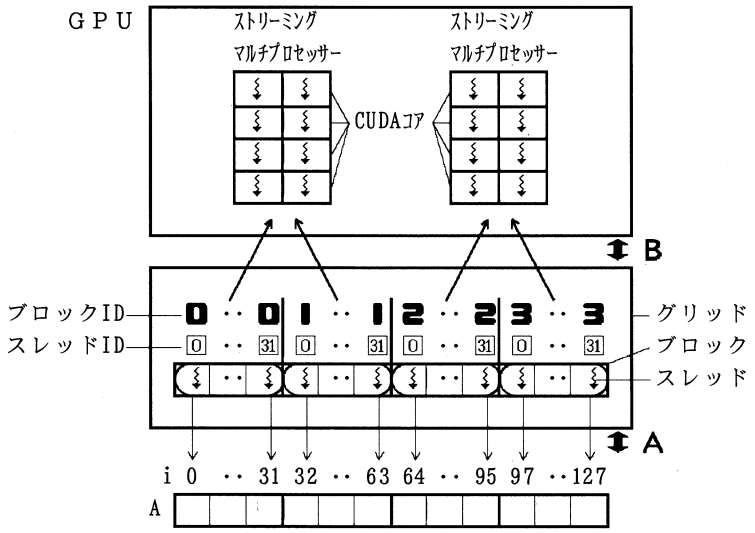


図 2-2-5　CUDA 並列

CUDA 並列の分かりにくい点

CUDA 並列と、MPI 並列やスレッド並列との相違点、および分かりにくい点を以下にまとめ、次節以降で説明します。

- MPI 並列やスレッド並列での CPU が、GPU では、ストリーミング・プロセッサーと CUDA コアという 2 階層に分かれています。
- MPI 並列やスレッド並列でのプロセスやスレッドが、CUDA 並列では、ブロックとスレッドという 2 階層に分かれています。
- A（スレッドと配列の関係）：MPI 並列やスレッド並列の場合は、（通常）1 つのプロセスまたはスレッドが、配列 A の複数の要素を担当しますが、CUDA 並列では、（通常）1 つのスレッドが配列 A の 1 要素を担当します。2 階層になっているスレッドを、どのように配列 A の各要素に対応付けるかが問題になります。
- B (スレッドと CUDA コアの関係): MPI 並列やスレッド並列の場合は、1 つの CPU を 1 つのプロセス（またはスレッド）が使用しますが、CUDA 並列では、1 つの CUDA コアを（通常）複数のスレッドが使用します。2 階層になっている CUDA コアと、2 階層になっているスレッドが、どのように対応付けられるのかが問題になります。

2-3　スレッドと配列の関係

本節では、前節の A（スレッドと配列の関係）について説明します。

実行構成の指定方法

1-3 節の kernel<<<1,N>>>() は、「カーネル関数 kernel を CUDA コア上で N 個のスレッドで実行せよ。」という意味でした。この <<<...>>> の部分（本書では実行構成と呼びます）の指定方法を、図 2-3-1 の例で説明します。この例では、グリッドにブロックが 2 個 (ブロック ID 0，1) 含まれ、各ブロックにスレッドが 4 個 (スレッド ID 0，1，2，3) 含まれます。ブロックとスレッドは、「x 方向 →」に示すように 1 次元ですが、y 方向、z 方向が加わると 2 次元、3 次元になります (2，3 次元については後述します)。

図 2-3-1 の実行構成は、図 2-3-2 の (1)〜(4) のいずれかの方法で指定することができます。

- まず (1) を説明します。①で dim3 型の変数 NBLOCKS と NTHREADS（名前は任意）を宣言します。dim3 型は CUDA で提供された構造体で、x，y，z の 3 つのメンバーを持ちます。
- ②，③，④で、グリッドに含まれる x，y，z 方向のブロック数を、変数 NBLOCKS のメンバー x，y，z に指定します。図 2-3-1 は 1 次元 (x 方向のみ) なので、x，y，z 方向のブロック数 2，1，1 を指定します。NBLOCKS の名前は任意ですが、「Number of BLOCKS」（(グリッド内の）ブロックの数）という意味で、NBLOCKS としました。
- ⑤，⑥，⑦で、ブロックに含まれる x，y，z 方向のスレッド数を、変数 NTHREADS のメンバー x，y，z に指定します。図 2-3-1 は 1 次元 (x 方向のみ) なので、x，y，z 方向のスレッド数 4，1，1 を指定します。NTHREADS の名前は任意ですが、「Number of THREADS」（(ブロック内の）スレッドの数）という意味で NTHREADS としました。
- ②〜⑦は、指定しないとデフォルトで 1 になるので、③，④，⑥，⑦は指定しなくても構いません。
- ⑧の実行構成内の 1 つ目の引数に NBLOCKS を、2 つ目の引数に NTHREADS を指定します。⑧を実行すると、ブロック／スレッドが図 2-3-1 の構成で、カーネル関数が実行を開始します（詳細は後述します）。
- (1) の①〜⑦の部分を (2)〜(4) のようにすることもできます。また①〜⑦と⑧を合体して (5) のようにすることもできます。
- (1)〜(5) の下線部を省略して、[1]〜[5] のようにすることもできます。1-3 節の <<<1,N>>> は、[5] の指定方法でブロック数が 1 の場合です。

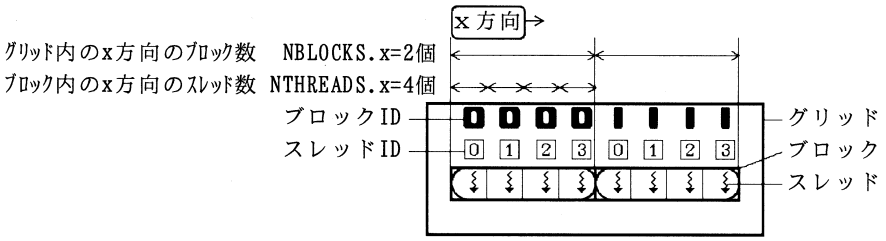


図 2-3-1

(1)
```
dim3 NBLOCKS,NTHREADS; (名前は任意)   ①
NBLOCKS.x = 2;  グリッド内のx方向のブロック数 ②
NBLOCKS.y = 1;  グリッド内のy方向のブロック数 ③
NBLOCKS.z = 1;  グリッド内のz方向のブロック数 ④
NTHREADS.x = 4; ブロック内のx方向のスレッド数 ⑤
NTHREADS.y = 1; ブロック内のy方向のスレッド数 ⑥
NTHREADS.z = 1; ブロック内のz方向のスレッド数 ⑦
kernel<<<NBLOCKS,NTHREADS>>>(dA);      ⑧
```

[1]
```
dim3 NBLOCKS,NTHREADS;
NBLOCKS.x  = 2;
NTHREADS.x = 4;
```

(2)
```
dim3 NBLOCKS(2,1,1),NTHREADS(4,1,1);
```

[2]
```
dim3 NBLOCKS(2),NTHREADS(4);
```

(3)
```
dim3 NBLOCKS,NTHREADS;
NBLOCKS  = dim3(2,1,1);
NTHREADS = dim3(4,1,1);
```

[3]
```
dim3 NBLOCKS,NTHREADS;
NBLOCKS  = 2;
NTHREADS = 4;
```

(4)
```
dim3 NBLOCKS  = dim3(2,1,1);
dim3 NTHREADS = dim3(4,1,1);
```

[4]
```
dim3 NBLOCKS  = 2;
dim3 NTHREADS = 4;
```

(5)
```
kernel<<<dim3(2,1,1),dim3(4,1,1)>>>(dA);
```

[5]
```
kernel<<<2,4>>>(dA);
```

図 2-3-2

スレッドと配列の関係 (1 次元の場合)

次のページの図 2-3-4 (1) (図 2-3-1 と同じ) のスレッドと、図 2-3-4 (2) の配列 dA[7] (8 ではなく 7) の各要素を対応付ける方法を、図 2-3-5 のプログラムで説明します。

- 図 2-3-5 の①，②で、実行構成を「グリッド内の x 方向のブロック数を 2 個、ブロック内の x 方向のスレッド数を 4 個」とし、③でカーネル関数 kernel を実行します。
- カーネル関数内では、4 つの構造体 gridDim，threadDim，blockIdx，threadIdx（構造体名は固定）を、CUDA が自動的に宣言します (各構造体の意味は④〜⑦の説明を参照して下さい)。これらは CUDA で提供されている dim3 型の構造体で、x，y，z のメンバーを持ちます。なお、blockIdx の Idx は、「ID の x」ではなく、「Index」の略です。
- ホスト側のプログラムの①，②で設定した値は、③を経由して、④，⑤のメンバー x に自動的に設定されます。つまり、以下の左と右の変数の値は同じです。ただし両者は別の変数なので注意して下さい。なお、④の gridDim.x は、カーネル関数内で使うことはあまりありません。

適用範囲	ホスト側のプログラム	カーネル関数
変数の宣言	プログラムで宣言	CUDA が自動的に宣言
変数名	任意	固定
値の設定	プログラムで設定	CUDA が自動的に設定
グリッド内の x 方向のブロック数 = 2	NBLOCKS.x　①	gridDim.x　④
ブロック内の x 方向のブロック数 = 4	NTHREADS.x　②	blockDim.x　⑤

- 各スレッドには、そのスレッドが所属する x 方向のブロック ID (本例では 0，1 のいずれか) と、そのスレッドの x 方向のスレッド ID (本例では0，1，2，3のいずれか) が自動的に付けられ、⑥，⑦のメンバー x に自動的に設定されます。
- 図 2-3-4 (1) のブロック ID，スレッド ID と、図 2-3-4 (2) の配列 dA の要素番号 i を対応付けるのが⑧です。⑧のように対応付けた場合、図 2-3-4 (3) のようになります。⑧の具体的な値を⑨に示します。⑧の対応付けは、CUDA プログラムでの定石の書き方です。「ブロック * ブロック + スレッド」と覚えるとよいでしょう。多くの CUDA プログラムでは、⑧のように対応付けを行いますが、例えば図 2-3-3 (1)〜(3) のように、⑧以外の対応付けをすることも可能です (ただしコアレスアクセス (3-2 節参照) の効率が悪くなります)。
- ⑩で各スレッドは、配列 dA の自分が担当する要素 dA[i] の計算を行います。本例では、配列 dA の範囲は dA[0]〜dA[6] なので、dA[7] を計算しないようにするため、下線の if 文を指定します (図 2-3-4 (3) の×参照)。この if 文の指定も CUDA プログラムでの定石です (4-2 節参照)。

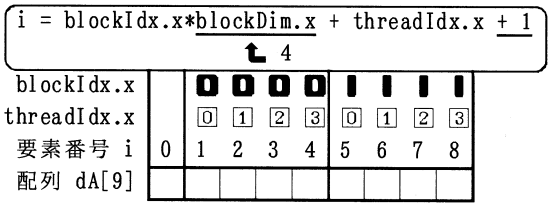


図 2-3-3 (1)　処理する最初の要素が dA[0] でなく dA[1]

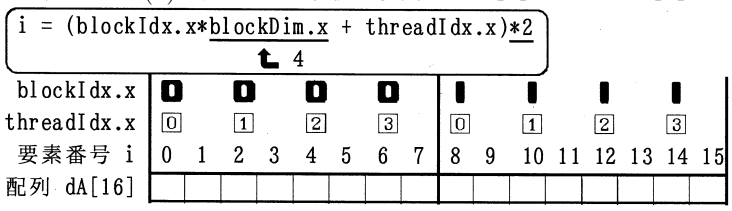


図 2-3-3 (2)　1 要素飛びに処理

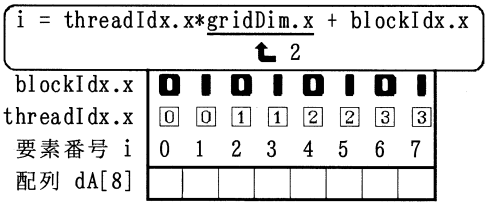


図 2-3-3 (3)　異なるブロックの同じスレッド ID のスレッドが連続した要素を処理

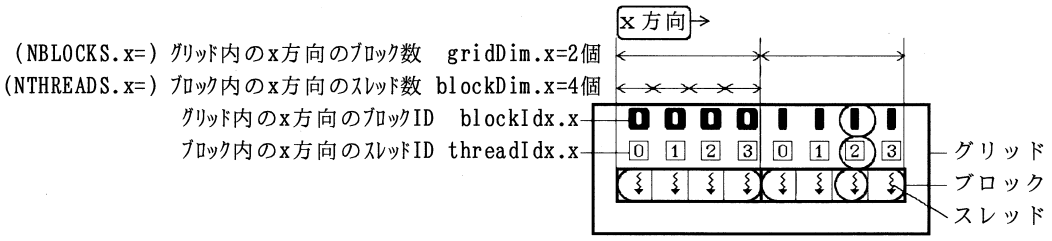


図 2-3-4 (1)

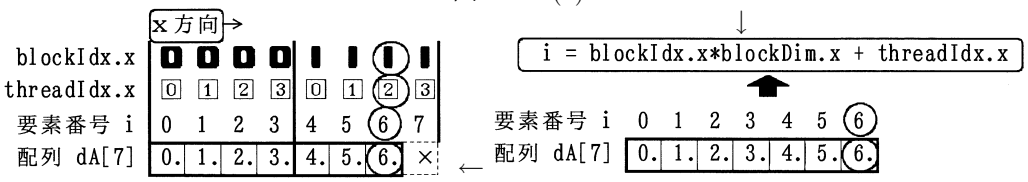


図 2-3-4 (3)　　図 2-3-4 (2)

```
__global__ void kernel(float *dA){
  • gridDim.x (グリッド内のx方向のブロック数)には2が設定されます(以下では未使用)。  ④
  • blockDim.x(ブロック内のx方向のスレッド数)には4が設定されます。                    ⑤
  • blockIdx.x (グリッド内のx方向のブロックID)には■,▌のいずれかが設定されます。      ⑥
  • threadIdx.x(ブロック内のx方向のスレッドID)には0～3のいずれかが設定されます。      ⑦
  int i = blockIdx.x*blockDim.x + threadIdx.x;                              ⑧
      i =  ■  *  4  +  0  ; (= 0)
      i =  ■  *  4  +  1  ; (= 1)
      i =  ■  *  4  +  2  ; (= 2)
      i =  ■  *  4  +  3  ; (= 3)                                           ⑨
      i =  ▌  *  4  +  0  ; (= 4)
      i =  ▌  *  4  +  1  ; (= 5)
      i =  ▌  *  4  +  2  ; (= 6)
      i =  ▌  *  4  +  3  ; (= 7)
  if (i<7) dA[i] = dA[i] + 1.0f;                                            ⑩
}
int main(void){
  dim3 NBLOCKS,NTHREADS;
    :
   NBLOCKS.x = 2;  グリッド内のx方向のブロック数を2個とします。①
  NTHREADS.x = 4;  ブロック内のx方向のスレッド数を4個とします。②
  kernel<<<NBLOCKS,NTHREADS>>>(dA);  ①,②の構成でkernelを実行します。  ③
    :
```

図 2-3-5

スレッドと配列の関係 (2 次元の場合)

次に、計算を行う配列が 2 次元の場合を説明します。この場合、ブロックとスレッドの構成も、通常 2 次元にします。次ページの図 2-3-7 (1) のスレッドと、図 2-3-7 (2) の 2 次元配列 dA[5][7] の各要素を対応付ける方法を、図 2-3-8 のプログラムで説明します。

- 図 2-3-8 の①〜④で、実行構成を、「グリッド内の x，y 方向のブロック数を 2，3 個、ブロック内の x，y 方向のスレッド数を 4，2 個」とし、⑤でカーネル関数 kernel を実行します。ブロックとスレッドの構成を図 2-3-7 (1) に示します。2 次元なので複雑に見えますが、単に前述の 1 次元を 2 方向にしただけです。
- ホスト側のプログラムの①〜④で設定した値は、⑤を経由して、⑥〜⑨のメンバー x，y に自動的に設定されます。つまり、以下の左と右の変数の値は同じです。ただし両者は別の変数なので注意して下さい。

適用範囲	ホスト側のプログラム	カーネル関数
変数の宣言	プログラムで宣言	CUDA が自動的に宣言
変数名	任意	固定
値の設定	プログラムで設定	CUDA が自動的に設定
グリッド内の x 方向のブロック数 = 2	NBLOCKS.x ①	gridDim.x ⑥
グリッド内の y 方向のブロック数 = 3	NBLOCKS.y ②	girdDim.y ⑦
グリッド内の x 方向のスレッド数 = 4	NTHREADS.x ③	blockDim.x ⑧
グリッド内の y 方向のスレッド数 = 2	NTHREADS.y ④	blockDim.y ⑨

- 各スレッドには、そのスレッドが所属する x 方向のブロック ID (本例では 0，1 のいずれか)、y 方向のブロック ID (本例では 0，1，2 のいずれか) と、そのスレッドの x 方向のスレッド ID (本例では0，1，2，3のいずれか)、y 方向のスレッド ID (本例では (0)，(1) のいずれか) が自動的に付けられ、⑩〜⑬のメンバー x，y に自動的に設定されます。以後、本書の図中では、各 ID に 0，1...、0，1...、0，1，...、(0)，(1)... を使用します。
- 図 2-3-7 (1) のブロック ID，スレッド ID と、図 2-3-7 (2) の配列 dA の要素番号 ix，iy を対応付けるのが⑭です。⑭のように対応付けた場合、図 2-3-7 (3) のようになります。例えば図 2-3-7 (1) の〓のスレッドの、要素番号 ix，iy の具体的な値は⑮となり、図 2-3-7 (3) の★に示す dA[4][6] に対応付けられます。多くの CUDA プログラムでは⑭ (図 2-3-6 (1) も同じ) のように対応付けますが、例えば図 2-3-6 (2) のような対応付けも可能です (8-5 節に使用例を示します)。図 2-3-6 (1) (2) の●，▲は同じプロセスの同じスレッドが担当します。

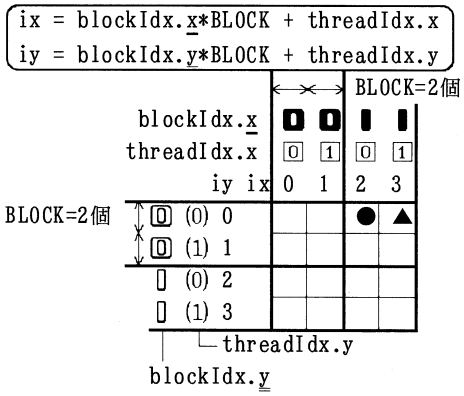


図 2-3-6 (1)　blockIdx.y

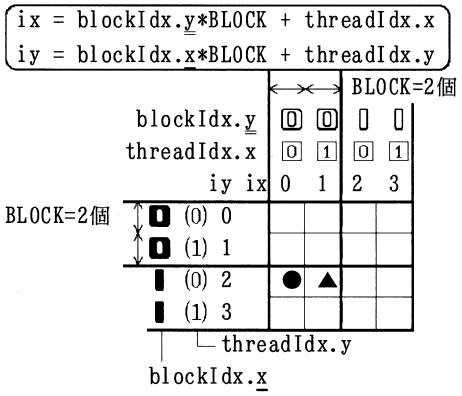


図 2-3-6 (2)　blockIdx.x

- ⑯で各スレッドは、配列 dA の自分が担当する要素 dA[iy][ix] の計算を行います。本例では、配列 dA の範囲は dA[0][0]〜dA[4][6] なので、それ以外の要素を計算しないようにするため、下線の if 文を指定します (図 2-3-7 (3) の×参照)。
- 図 2-3-8 では、カーネル関数内で 2 次元配列 dA[5][7] を使用していますが、CUDA 関数の場合、カーネル関数で 2 次元配列を扱うためには少し考慮が必要となります。これについては 3-3 節と 3-4 節で説明します。

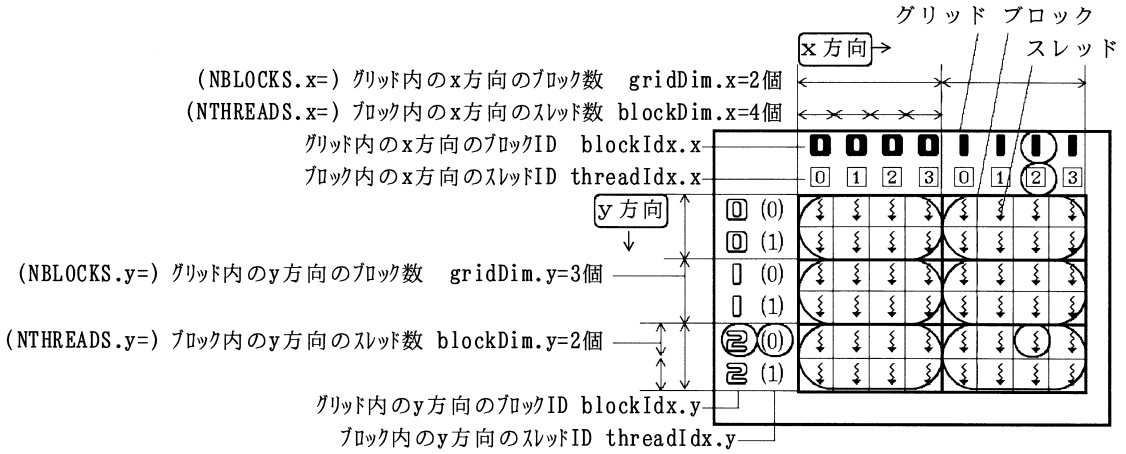


図 2-3-7 (1)

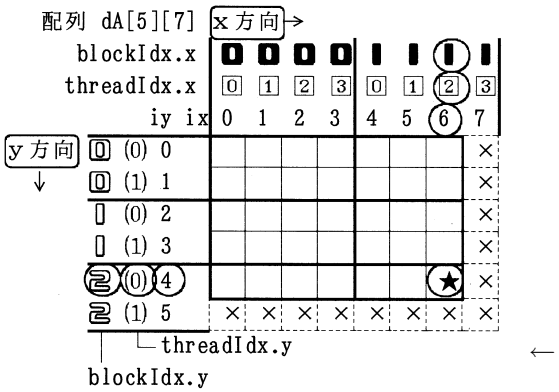


図 2-3-7 (3)　blockIdx.y

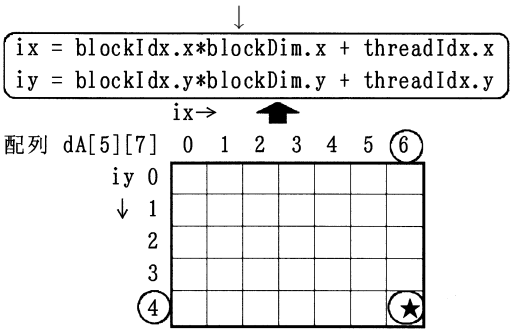


図 2-3-7 (2)

```
__global__ void kernel(～){
  • gridDim.x(グリッド内のx方向のブロック数)には2が設定されます(以下では未使用)。  ⑥
  • gridDim.y(グリッド内のy方向のブロック数)には3が設定されます(以下では未使用)。  ⑦
  • blockDim.x(ブロック内のx方向のスレッド数)には4が設定されます。  ⑧
  • blockDim.y(ブロック内のy方向のスレッド数)には2が設定されます。  ⑨
  • blockIdx.x(グリッド内のx方向のブロックID)には0, 1 のいずれかが設定されます。  ⑩
  • blockIdx.y(グリッド内のy方向のブロックID)には0, 1, 2のいずれかが設定されます。  ⑪
  • threadIdx.x(ブロック内のx方向のスレッドID)には0～3のいずれかが設定されます。  ⑫
  • threadIdx.y(ブロック内のy方向のスレッドID)には(0),(1)のいずれかが設定されます。  ⑬
  int ix = blockIdx.x*blockDim.x + threadIdx.x;          ⑭
  int iy = blockIdx.y*blockDim.y + threadIdx.y;          ⑭
     ix =    1    *    4      +    2      ; (=6)        ⑮
     iy =    2    *    2      +    0      ; (=4)        ⑮
  if (ix<7 && iy<5) dA[iy][ix] = dA[iy][ix] + 1.0f;      ⑯
}
int main(void){
  dim3 NBLOCKS,NTHREADS;
    :
   NBLOCKS.x = 2;  グリッド内のx方向のブロック数を2個とします。①
   NBLOCKS.y = 3;  グリッド内のy方向のブロック数を3個とします。②
  NTHREADS.x = 4;  ブロック内のx方向のスレッド数を4個とします。③
  NTHREADS.y = 2;  ブロック内のy方向のスレッド数を2個とします。④
  kernel<<<NBLOCKS,NTHREADS>>>(～);  ①～④の構成でkernelを実行します。  ⑤
    :
```

図 2-3-8

2-4　スレッドと CUDA コアの関係

本節では、B (スレッドと CUDA コアの関係) について説明します。

指定できるブロック／スレッドの最大数

まず、ブロック、スレッドの指定可能な最大数を図 2-4-1 に示します。z 方向は x，y 方向より少なくなります。

- グリッド内で指定できるブロックの最大数は、x 方向が 65535 個、y が 65535 個、z 方向が 1 個です。
- 下の図に示すように、ブロック内で指定できるスレッドの最大数は、x 方向が 512 個、y 方向が 512 個、z 方向が 64 個です。ただし、1 つのブロック内で指定できる全スレッド数は 512 個以下です。従って、例えば (x 方向のスレッド数,y 方向のスレッド数,z 方向のスレッド数) = (8,8,8) などが、1 ブロックの最大スレッド数です。
- 本書では、2 次元の場合、ブロック (0,1)、スレッド ([0],(1)) のように、(x,y) の順に表します。

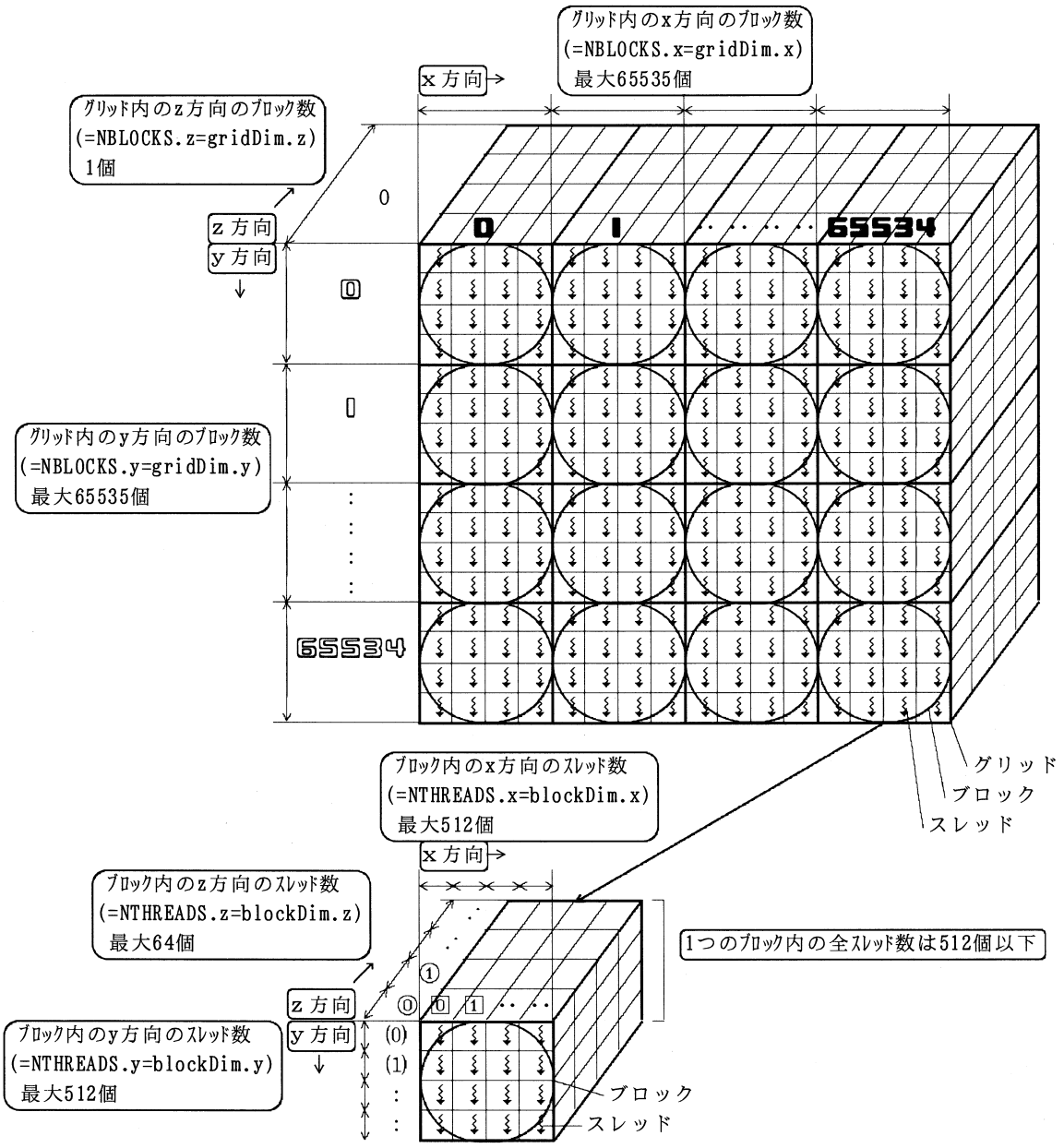


図 2-4-1

ワープ

同一ブロック内にある、連続した（連続の意味は以下で説明します）32 スレッドのことを、ワープ (Warp: 英語の発音はウォープ) と呼びます。CUDA では、ワープを単位として計算が行われます（詳細は後述します）。

例えば 1 つのブロックが図 2-4-2 (1) (2) の場合、矢印が示す順 (まず x 方向が先に動く順、次に y 方向が動く順、最後に z 方向が動く順) に並んだ 32 個のスレッドが、1 つのワープとなります。従って図 2-4-2 (2) では、手前の 4 つの〓と、奥の 4 つの〓（図では見えません）がそれぞれワープとなります。

```
  :
NTHREADS.x = 16;
NTHREADS.y =  8;
NTHREADS.z =  2;
kernel<<<1,NTHREADS>>>();
  :
```

図 2-4-2 (1)

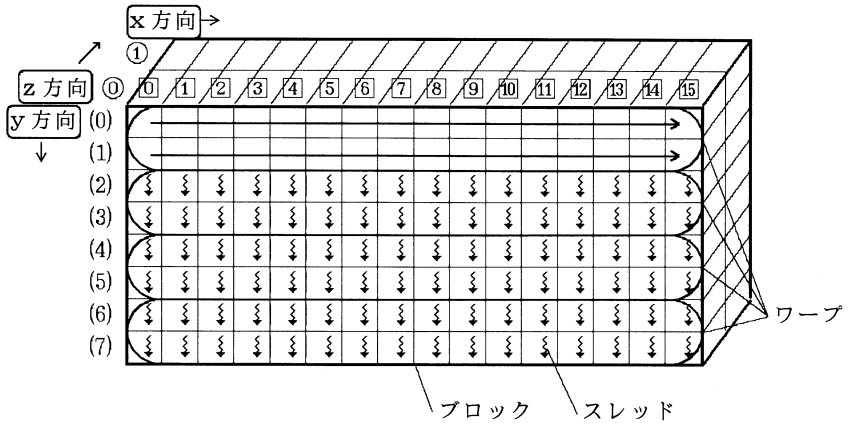


図 2-4-2 (2)

スレッドと CUDA コアの関係

2 階層のブロック／スレッドが、2 階層の GPU 上のストリーミング・マルチプロセッサー／CUDA コアで、どのように処理されるかを説明します。

処理する配列

図 2-4-3 に、処理する 2 次元配列 dA[12][32] (要素数は $12 \times 32 = 384$ 個) を示します。

ブロック／スレッド

図 2-4-3 に示すように、ブロック数は x 方向に 2 個 (ブロック ID $= 0, 1$)、y 方向に 3 個 (ブロック ID $= 0, 1, 2$) の合計 6 個で、各ブロックには、x 方向のスレッド数が 16 個 (スレッド ID $=$ 0〜15)、y 方向のスレッド数が 4 個 (スレッド ID $=$ (0)〜(3)) の合計 64 個のスレッドが含まれています。従って、全スレッド数は $6 \times 64 = 384$ 個 (配列 dA の要素数と同じ) となります。

マシン環境

説明を簡単にするため、実際のマシン環境を以下のように簡単化します。

- 1 つの GPU は、ストリーミング・マルチプロセッサー (以下 SM と略します) が 30 個含まれますが、図 2-4-5 の上部に示すように 2 個だとします。
- 各 SM ごとに、資源（レジスターやシェアードメモリなど）を持っています。ここでは図 2-4-5 の上部に示すように、各 SM の資源は 160 個のレジスターのみだとします（各資源の種類と正確な数は後述します）。
- 各スレッドは、実行時にレジスターを 1 個だけ使用するとします。1 つのブロックには 64 スレッドが含まれるので、1 つのブロックでレジスターを 64 個使用します。

動作

このマシン環境で、図 2-4-3 に示す、$(0,0)$，$(1,0)$，$(0,1)$，$(1,1)$，$(0,2)$，$(1,2)$ の 6 個のブロック内の各スレッドが、どのように実行されるのかを説明します。

- 各 SM には、ブロック単位で割り当てが行われます。本例では、1 つの SM のレジスター数は 160 個、1 つのブロックが使用するレジスター数は 64 個なので、1 つの SM に、同時に 2 個のブロックが存在することができます (存在できるブロック数には上限があります。詳細は 6-1 節参照)。各 SM に、6 個のうちどのブロックを割り当てるかを、図 2-4-5 のスケジューラー 1（正式な名前ではありません）が決定します。
- 図 2-4-5 に示すように、スケジューラー 1 が、例えばブロック (0,0) と (0,1) を左の SM に、ブロック (1,0) と (1,1) を右の SM に割り当てたとします。4 つのブロックが使用するレジスターを図 2-4-5 の一番上の図に示します。残りのブロック (0,2), (1,2) は図の下部の待ち行列に入ります。
- 一度 SM に入ったブロックは、ブロック内の全スレッドがカーネル関数全体の処理を終わるまで、SM 内に存在します。あるブロックの処理が終了したら、そのブロックが使用していた資源（本例ではレジスター）が解放され、スケジューラー 1 は、待ち行列に入っている未処理のブロックを 1 つ選択し、SM に割り当てます。
- 以下、左の SM の動作を説明します。ここからは、ブロックではなく、ワープが処理の単位になります。本例では 1 つのブロックに 2 つのワープが含まれているので、左の SM が担当するのは 4 つのワープ (図のワープ 0，1，2，3) です。4 つのワープのうち、例えば○がついているワープは現在計算を行うことが可能、×がついているワープは不可能（例えばメモリからロード／ストアなどを行っているため）だとします。スケジューラー 2 は、○のワープ (ワープ 0 と 3) から 1 つ (本例ではワープ 0) を選択し、SM に割り当てます。
- GPU では、デコーダ（機械語命令を解読する装置）は、CUDA コアごとではなく、各 SM に 1 つ存在します。図 2-4-5 の上部に示すように、デコーダは機械語命令（例えば加算）を解読すると、各 CUDA コアに伝達し、8 つの CUDA コア内のスレッドは同じ命令を実行します。
- 1 ワープは 32 スレッドで、1 つの SM 内の CUDA コアは 8 個です。図 2-4-5 の①に示すように、ワープ 0 内の、まず 8 スレッドが 8 個の CUDA コアで同じ命令（例えば加算）を実行します。同様に②，③，④の順に、8 スレッドずつ CUDA コアで同じ命令を実行します。タイムチャートを図 2-4-4 に示します。従って、同じワープ内の 32 スレッドは、同じ命令を実行することになります。
- ワープ 0 の 32 スレッドの（例えば）加算が終了すると、ワープ 0 は SM から出ます。すぐに次の命令を実行できる場合は○、ロード／ストアなどを行う場合は×（ロード／ストアが完了したら○）となります。その後、スケジューラー 2 がワープ 0 を再び選択したら、SM で次の命令を実行します (詳細は 6-1 節参照)。
- 左の SM のスケジューラー 2 は、ワープ 0 が出た後、その時点で○のワープ (ワープ 3) を選択し、SM に割り当てます。ワープ 3 内の全スレッドも同じ命令を実行しますが、ワープ 0 のスレッドが実行した命令と、必ずしも同じとは限りません。つまり、どの命令まで実行を完了しているかはワープによって異なります。

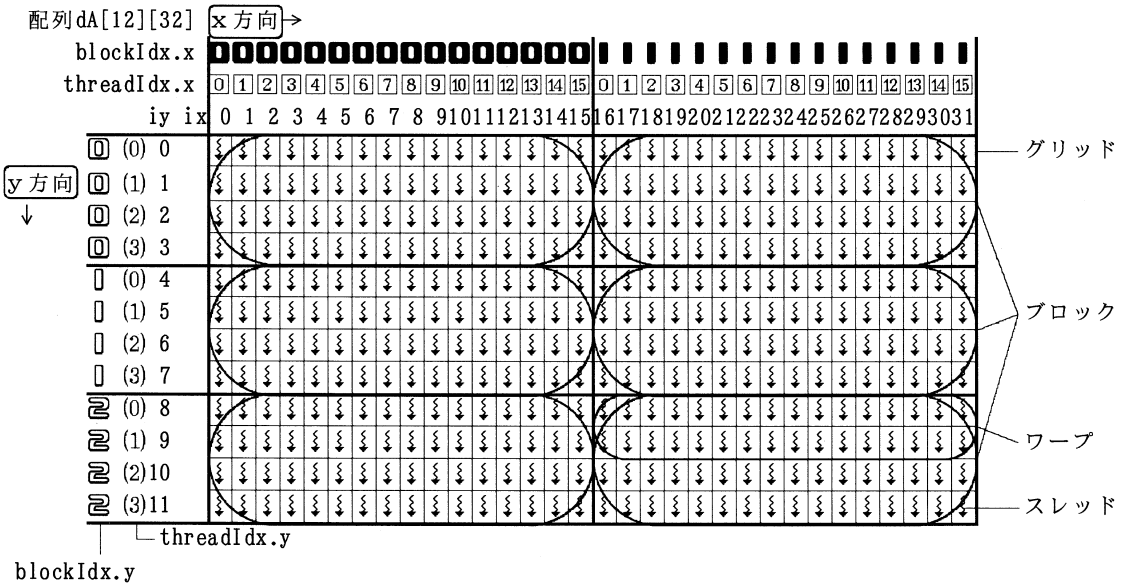


図 2-4-3

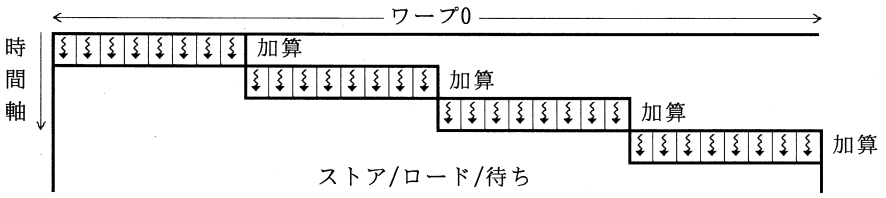


図 2-4-4

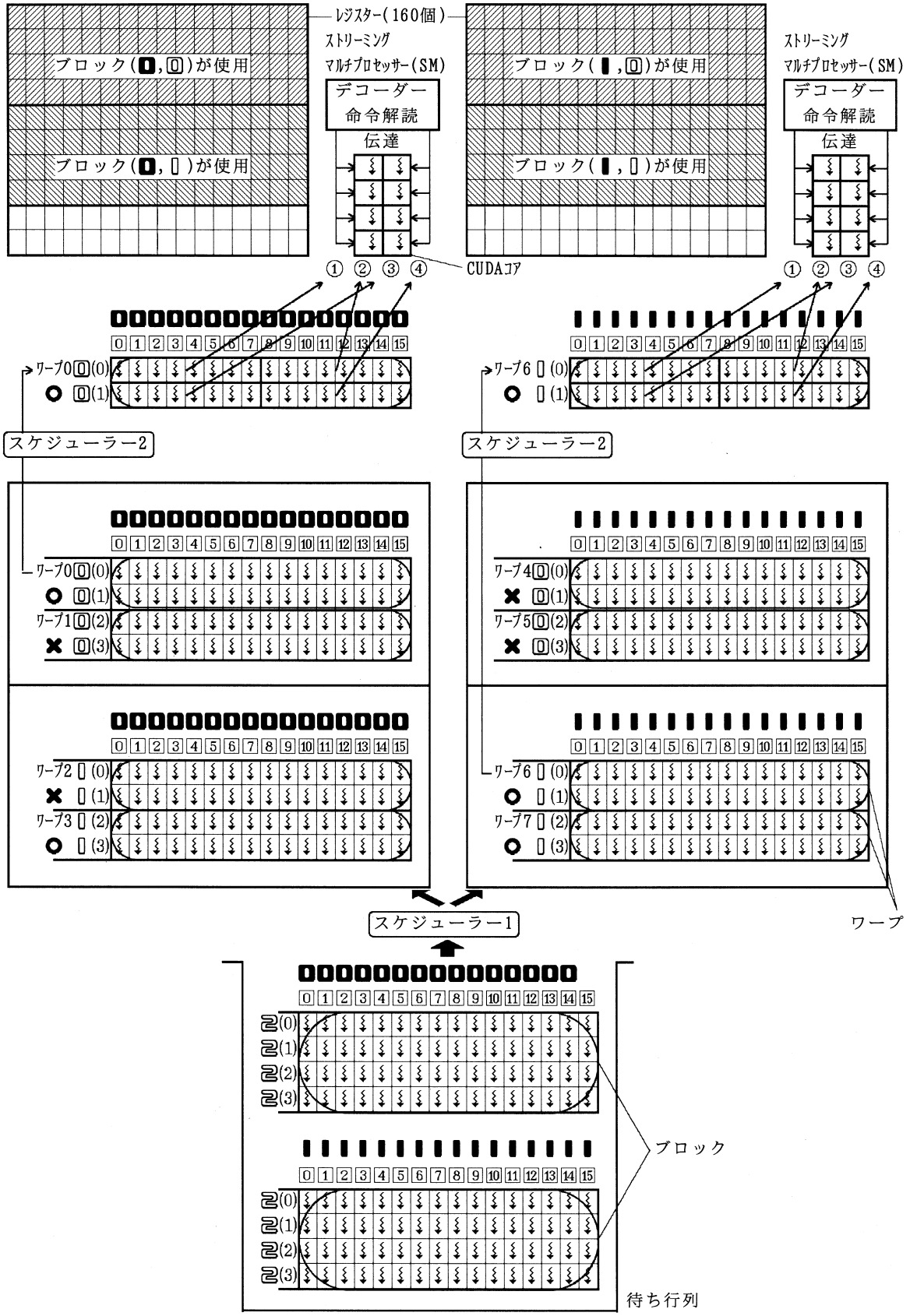

レジスター(160個)
ストリーミング
マルチプロセッサー(SM)
デコーダー
命令解読
伝達
CUDAコア
ブロック(0,0)が使用
ブロック(0,1)が使用
ブロック(1,0)が使用
ブロック(1,1)が使用
スケジューラー2
スケジューラー1
ワープ
ブロック
待ち行列

図 2-4-5

2-5　ブロック数とスレッド数の設定 (1)

本節では、ブロック数とスレッド数の設定方法を説明します (詳細な設定方法は 6-1 節で説明します)。

ブロック数

ストリーミング・マルチプロセッサー (SM) の数は 30 個です。各ブロックが同じ計算量を行う場合、図 2-5-1 (1) (2) に示すように、ブロック数が 30 個でも 1 個でも、計算時間はほぼ同じになります。言いかえると、図 2-5-1 (2) では、29 個のストリーミング・マルチプロセッサーが遊んでいるため、同じ時間で行う計算量は、図 2-5-1 (1) の 1/30 になります。従って、ブロック数は 30 個以上が推奨されます。

また、カーネル関数内で `__syncthreads()` を使用して同期を取っている場合、1 つのストリーミング・マルチプロセッサーあたり複数のブロックが推奨されます (詳細は 6-1 節参照)。

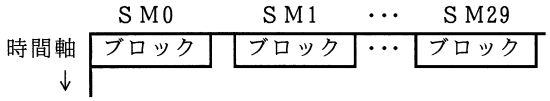


図 2-5-1 (1)

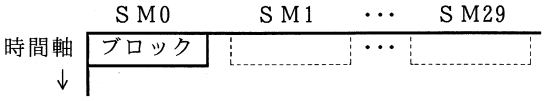


図 2-5-1 (2)

ブロック内のスレッド数

前述のように、スレッドの処理は、ワープ (連続した 32 スレッド) 単位で行われます。1 つのブロック内のスレッド数を例えば 32 個とした場合、タイムチャートは図 2-5-2 (1) のようになります (紙面の都合でスレッドを「↓」で表します)。一方、1 つのブロック内のスレッド数を例えば 25 個とした場合、図 2-5-2 (2) のようになり、計算時間は図 2-5-2 (1) とほぼ同じになります。言いかえると、図 2-5-2 (2) では、右端の空白の部分で、7 個の CUDA コアが動作せずに遊んでいるため、同じ時間で行う計算量は、図 2-5-2 (1) より少なくなります。従って、1 つのブロック内のスレッド数は 32 (= 1 ワープ) の倍数 (上限は 512 個) が推奨されます。

ブロック内のスレッドが 2 次元の場合も同様に、(x 方向のスレッド数) × (y 方向のスレッド数) は、32 の倍数が推奨されます。それに加え、x 方向のスレッド数は 16 の倍数が推奨されます (コアレスアクセスが理由ですが、3-2、3-3 節で説明します)。この条件に該当する x, y 方向のスレッド数を図 2-5-3 に示します。

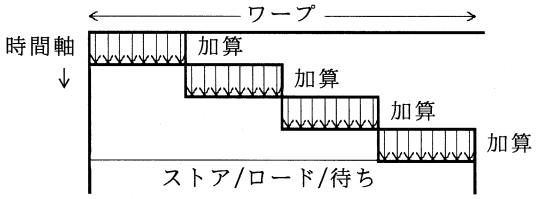


図 2-5-2 (1)

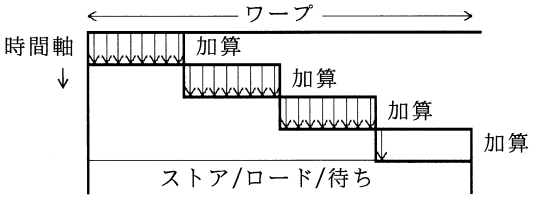


図 2-5-2 (2)

ブロックあたりのスレッド数	32	64	96	128	160	192	224	256
ブロック内の (x方向のスレッド数) ×(y方向のスレッド数)	16×2 32×1	16×4 32×2 64×1	16×6 32×3 48×2 96×1	16×8 32×4 64×2 128×1	16×10 32×5 80×2 160×1	16×12 32×6 48×4 64×3 96×2 192×1	16×14 32×7 112×2 224×1	16×16 32×8 64×4 128×2 256×1
ブロックあたりのスレッド数	288	320	352	384	416	448	480	512
ブロック内の (x方向のスレッド数) ×(y方向のスレッド数)	16×18 32×9 48×6 96×3 144×2 288×1	16×20 32×10 64×5 80×4 160×2 320×1	16×22 32×11 176×2 352×1	16×24 32×12 48×8 64×6 96×4 128×3 192×2 384×1	16×26 32×13 208×2 416×1	16×28 32×14 64×7 112×4 224×2 448×1	16×30 32×15 48×10 80×6 96×5 160×3 240×2 480×1	16×32 32×16 64×8 128×4 256×2 512×1

図 2-5-3

スレッド数を固定してブロック数を求める方法

処理する要素数が決まっている場合、ブロック数と（ブロックあたりの）スレッド数は、一方が決まれば他方は自動的に決まります。通常はスレッド数を固定します。スレッド数を例えば 32 に固定した場合、要素数とスレッド数からブロック数を求める 3 つの方法を、図 2-5-4 の (0), (1), (2) に示します（整数で計算し、除算の計算結果は切り捨てます）。要素数が 0〜96 のときの、(0), (1), (2) で求めたブロック数を図の右欄に示します。

要素数がスレッド数で割りきれる図 2-5-5 (1) の場合、(0), (1), (2) のいずれの方法でも、ブロック数は 2 個（正しい）となります。ところが、要素数がスレッド数で割りきれない図 2-5-5 (2) の場合、(1), (2) ではブロック数は 3 個（正しい）になりますが、(0) では 2 個（誤り）になります。また、要素数が 0 個の場合、(1) ではブロック数が 0 個（正しい）になりますが、(2) では 1 個（誤り）になります。従って (1) を使用するのが無難だと思われます。

図 2-5-4 の (1) 式を使用した場合の、1 次元の場合のプログラム例を図 2-5-6 (1) に、2 次元の場合のプログラム例を図 2-5-6 (2) に示します。

要素数	(0)	1	･･	31	32	33	･･	63	(64)	(65)	･･	95	96
✖ (0) ﾌﾞﾛｯｸ数 = 要素数/ｽﾚｯﾄﾞ数	0	✖0	･･	✖0	1	✖1	･･	✖1	2	✖2	･･	✖2	3
○ (1) ﾌﾞﾛｯｸ数 = (要素数+ｽﾚｯﾄﾞ数-1)/ｽﾚｯﾄﾞ数	0	1	･･	1	1	2	･･	2	2	3	･･	3	3
△ (2) ﾌﾞﾛｯｸ数 = (要素数-1)/ｽﾚｯﾄﾞ数 + 1	✖1	1	･･	1	1	2	･･	2	2	3	･･	3	3

図 2-5-4　スレッド数 = 32 の場合のブロック数（×は誤り）

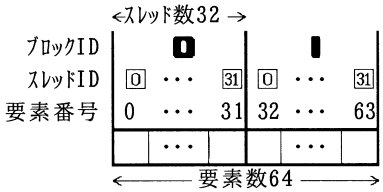


図 2-5-5 (1)

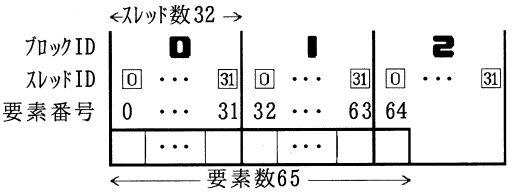


図 2-5-5 (2)

```
#define N (1000) ← 処理する要素数
__device__ float dA[N];
int main(void){
dim3 NBLOCKS,NTHREADS;
  :
NTHREADS.x = 32; ← ﾌﾞﾛｯｸあたりのｽﾚｯﾄﾞ数を32
NBLOCKS.x = (N+NTHREADS.x-1)/NTHREADS.x; ← (1)
kernel<<<NBLOCKS,NTHREADS>>>();
  :
```

図 2-5-6 (1)

```
#define NX (1000) ← 処理するx方向の要素数
#define NY (2000) ← 処理するy方向の要素数
__device__ float dA[NY][NX];
int main(void){
dim3 NBLOCKS,NTHREADS;
  :
NTHREADS.x = 16; ← ﾌﾞﾛｯｸあたりのx方向のｽﾚｯﾄﾞ数
NTHREADS.y = 16; ← ﾌﾞﾛｯｸあたりのy方向のｽﾚｯﾄﾞ数
NBLOCKS.x = (NX+NTHREADS.x-1)/NTHREADS.x; ←(1)
NBLOCKS.y = (NY+NTHREADS.y-1)/NTHREADS.y; ←(1)
kernel<<<NBLOCKS,NTHREADS>>>();
  :
```

図 2-5-6 (2)

処理する要素数が少ない場合／（非常に）多い場合

前述のように、ブロック数は 30 個以上、ブロック内のスレッド数は 32 の倍数 (上限は 512 個) が推奨されます。処理する要素数が 30 × 32 個より少ない場合、例えば 480 個 (= 15 × 32) の場合、以下の (1) のようにブロック数を少なくする方法、(2) のようにスレッド数を少なくする方法、(3) のようにその中間の方法が考えられます。この場合、速度はカーネル関数の処理内容によって変わりますので、試行錯誤で試して下さい。

逆に、処理する要素数（正確には全スレッド数）が 30 × 32 個より（非常に）多い場合、例えば 61440 個 (= 120 × 512) の場合、以下の (4) のようにブロック数を少なくする方法、(5) のようにスレッド数を少なくする方法、(6) のようにその中間の方法が考えられます。これについては 6-1 節で検討します。

	ブロック数	ブロック内のスレッド数
(1)	15	32
(2)	30	16
(3)	20	24

	ブロック数	ブロック内のスレッド数
(4)	120	512
(5)	1920	32
(6)	480	128

全ケースを実測する方法

簡単なプログラムであれば、ブロック数／スレッド数をいろいろ変えて実際に測定し、最も速くなるブロック数／スレッド数を試行錯誤で決定する方法もあります。

配列 1 次元の場合の例を図 2-5-7 (1) に示します。

- ④の実行構成で指定する、ブロック数、ブロック内のスレッド数を設定する変数を、①で宣言します。
- ②で、ブロック内のスレッド数を 32 ずつ (32 の倍数が推奨なので)、最大値の 512 まで変化させます。
- ③で、処理する要素 N とスレッド数から、ブロック数を決定します。
- ④の前後にタイマールーチン (4-4 節参照) を挿入して測定します。

配列が 2 次元の場合の例を図 2-5-7 (2) に示します。

- ⑤で、ブロック内の x 方向のスレッド数を、16 ずつ (2 次元の場合、前述のように x 方向は 16 の倍数が推奨なので)、最大値の 512 まで変化させます。⑥で、ブロック内の y 方向のスレッド数を、1 ずつ、最大値の 512 まで変化させます。
- ⑦で、x 方向と y 方向のスレッド数を掛けて、ブロック内の全スレッド数を求めます。
- ⑧で、ブロック内の全スレッド数が最大値の 512 より大きいか、32 の倍数でない場合は測定を行いません。
- ⑨で、処理する x, y 方向の要素数 NX と NY、および x, y 方向のスレッド数から、x, y 方向のブロック数を決定します。
- ⑩の前後にタイマールーチン (4-4 節参照) を挿入して測定します。

```
#define N (10000)
int main(void){
  double elp1,elp2;
  dim3 NBLOCKS,NTHREADS;                          ①
    :
  for(NTHREADS.x=32;NTHREADS.x<=512;              ②
      NTHREADS.x+=32){                            ②
    NBLOCKS.x = (N+NTHREADS.x-1)/NTHREADS.x;      ③
    cudaThreadSynchronize();
    elp1 = gettimeofday_sec();
    kernel<<<NBLOCKS,NTHREADS>>>(dA);             ④
    cudaThreadSynchronize();
    elp2 = gettimeofday_sec();
    printf("NTHREADS.x = %d ELAPSE = %.6f¥n",
           NTHREADS.x,elp2-elp1);
  }
    :
```

図 2-5-7 (1)

```
#define NX (1000)
#define NY (1000)
int main(void){
  double elp1,elp2;
  dim3 NBLOCKS,NTHREADS;
    :
  for(NTHREADS.x=16;NTHREADS.x<=512;              ⑤
      NTHREADS.x+=16){                            ⑤
    for(NTHREADS.y= 1;NTHREADS.y<=512;            ⑥
        NTHREADS.y++){                            ⑥
      int itemp = NTHREADS.x*NTHREADS.y;          ⑦
      if ((itemp>512)||((itemp%32)!=0))           ⑧
        continue;                                 ⑧
      NBLOCKS.x
           = (NX+NTHREADS.x-1)/NTHREADS.x;        ⑨
      NBLOCKS.y
           = (NY+NTHREADS.y-1)/NTHREADS.y;        ⑨
      cudaThreadSynchronize();
      elp1 = gettimeofday_sec();
      kernel<<<NBLOCKS,NTHREADS>>>();             ⑩
      cudaThreadSynchronize();
      elp2 = gettimeofday_sec();
      printf("NTHREADS.x = %d NTHREADS.y = %d
              ELAPSE = %.6f¥n",NTHREADS.x,
              NTHREADS.y,elp2-elp1);
    }
  }
    :
```

図 2-5-7 (2)

2-6　ブロック数 × スレッド数 より要素数の方が多い場合

2-4 節で説明したように、x 方向の最大ブロック数は 65535、x 方向の最大スレッド数は 512 です。従って図 2-6-1 に示すように、スレッドを、要素数が 65535 × 512 より大きな配列 dA の×の要素に対応させることができません。同様に、z 方向の最大ブロック数は 1、z 方向の最大スレッド数は 64 なので、スレッドを、3 次元配列 dA[100][100][100] の 1 次元目（左の下線部）に対応させることができません。

この場合の対処方法を説明します。説明を簡単にするために、図 2-6-1 の代わりに、図 2-6-2 に示すように、x 方向の最大ブロック数を 3、x 方向の最大スレッド数を 4 とし、スレッドを、要素数が 3 × 4 = 12 より大きな配列 dA[14] に対応させるとします。×に示す dA[12] と dA[13] がスレッドの範囲を越えた要素です。

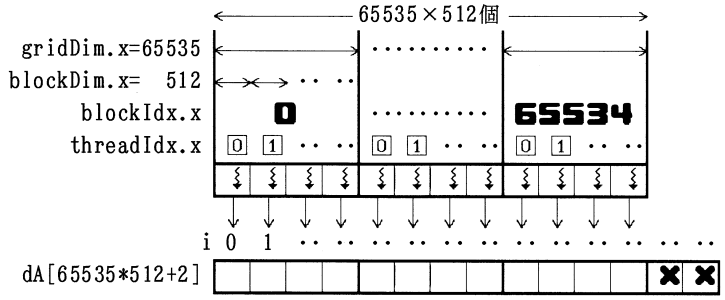


図 2-6-1

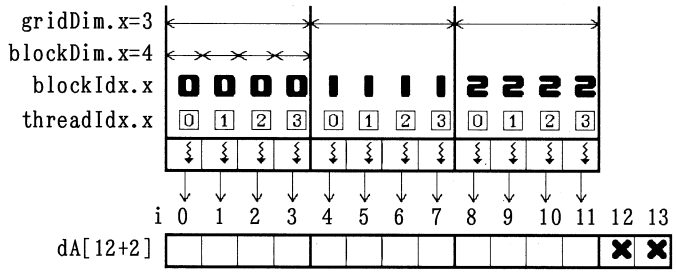


図 2-6-2

【方法 1】図 2-6-3 (2) の①で、通常の方法で配列の要素番号 ista (0〜11) を求めます。②でループを反復させることによって、dA の要素⓪を処理したスレッドは要素⑫も処理し、要素1を処理したスレッドは要素13も処理します。配列 dA の範囲を越えることはないので、後述する方法 2 と 3 で使用する if 文は不要です。

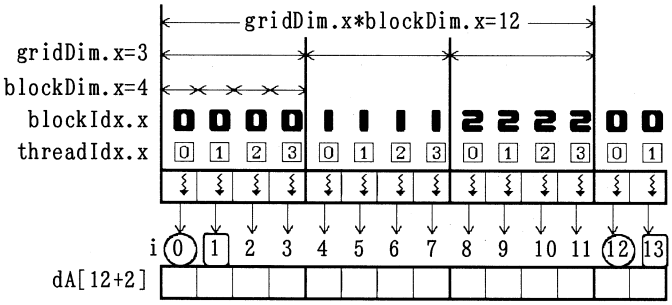


図 2-6-3 (1)

```
#define N (14)
__global__ void kernel(float *dA){
  int ista = blockIdx.x*blockDim.x + threadIdx.x;          ①
  for(int i=ista;i<N;i+=gridDim.x*blockDim.x){             ②
    dA[i] = dA[i] + 1.0f;          ↰ 12
  }
}
int main(void){
  kernel<<<3,4>>>(dA);
    :
```

図 2-6-3 (2)

【方法 2】図 2-6-4 (1) に示すように、x 方向のブロック (ブロック ID 0, 1) の他に、y 方向のブロック (ブロック ID 0, 1) を加え (図 2-6-4 (2) の③参照)、使用できるブロック数を増やします。図 2-6-4 (2) の①で、下線に示す 3 つの ID を使用して要素番号を計算します。①で配列 dA の範囲を越える場合があるので、②の if 文を指定します。

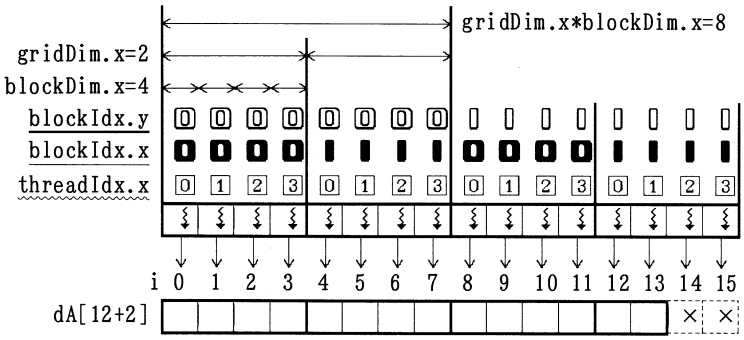


図 2-6-4 (1)

```
#define N (14)
__global__ void kernel(float *dA){
  int i = blockIdx.y*(gridDim.x*blockDim.x) + blockIdx.x*blockDim.x + threadIdx.x;
  if(i<N) dA[i] = dA[i] + 1.0f;  ②  ↳ 8                ↳ 4          ①
}
int main(void){
  kernel<<<dim3(2,2),4>>>(dA);   ③
    :
```

図 2-6-4 (2)

【方法 3】各スレッドは、配列 dA の複数要素を担当します。図 2-6-5 (2) の①で、全要素数 N を全スレッド数で割り、各スレッドが担当する要素数 iwork を求めます (iwork をホスト側で計算して引数で渡す方法もあります)。②で、各スレッドが担当する最初の要素番号 ista (図 2-6-5 (1) の○の要素) を計算し、③で複数要素分だけ反復します。③で配列 dA の範囲を越える場合があるので、④の if 文を指定します。各スレッドが同時に処理する要素がメモリ上でとびとびなので、コアレスアクセス (3-2 節参照) の効率が悪くなります。

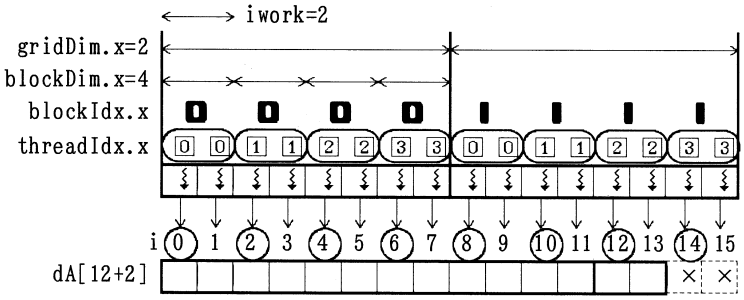


図 2-6-5 (1)

```
#define N (14)
__global__ void kernel(float *dA){
  int iwork = (N-1)/(gridDim.x*blockDim.x) + 1;                    ①
        ↳ 2                  ↳ 8
  int ista = (blockIdx.x*blockDim.x + threadIdx.x)*iwork;          ②
  for(int i=ista;i<ista+iwork;i++){  ↳ 0,1,…,7      ↳ 2           ③
    if(i<N) dA[i] = dA[i] + 1.0f;                                  ④
  }
}
int main(void){
  kernel<<<2,4>>>(dA);
    :
```

図 2-6-5 (2)

2-7　コンパイルオプション

本節では、コンパイルオプションについて説明します。

コンパイル方法

CUDA でのコンパイルの例を図 2-7-1 (1) に示します。

- ソースプログラムのファイル名は「〜.cu」とします (以下の例では test.cu だとします)。
- nvcc コマンドでコンパイルを行ないます。図 2-7-1 (2) に示すように、ホスト側プログラムは内部的に gcc コマンドでコンパイルが行なわれ、デバイス側プログラム（カーネル関数）は内部的に nvopencc コマンドでコンパイルが行なわれます。

```
nvcc (最適化ｵﾌﾟｼｮﾝ) test.cu ↵
```

図 2-7-1 (1)

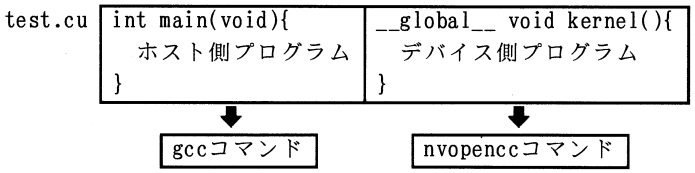


図 2-7-1 (2)

コンパイラのバージョン情報の表示

①で nvcc のバージョン情報が、②で gcc のバージョン情報が表示されます。

```
$ nvcc -V(大文字) ↵  ①
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2010 NVIDIA Corporation
Built on Wed_Nov__3_16:16:57_PDT_2010
Cuda compilation tools, release 3.2, V0.2.1221
```

```
$ gcc -v(小文字) ↵  ②
Using built-in specs.
Target: x86_64-redhat-linux
  :
gcc version 4.1.2 20080704 (Red Hat 4.1.2-44)
```

ヘルプコマンド／マニュアル／書籍

nvcc コマンド

nvcc コマンドで指定可能な各オプションの説明は、以下のコマンドを実行すると表示されます。マニュアルは、「The CUDA Compiler Driver NVCC」（付録参照）です。

```
$ man nvcc ↵
```

```
$ nvcc -h ↵
```

gcc コマンド

gcc コマンドで指定可能な各オプションの説明は、以下のコマンドを実行すると表示されます。

```
$ man gcc ↵
```

```
$ gcc --help ↵
```

gcc コマンドに関して、下記の書籍が出版されています（他にもあるかもしれません)。また Web 上で、gcc コマンドを紹介しているサイトもあります。

– 「実例で学ぶ GCC の本格的活用」（CQ 出版社）
– 「GCC Manual & Reference 増補改訂版」（秀和システム）

nvopencc コマンド

nvopencc コマンドで指定可能なオプションを説明した、ヘルプコマンドやマニュアルは見つかりませんでした。最適化オプションについては後述します。

指定が望ましい最適化オプション

gcc コマンドの最適化オプション

下記の下線部で、gcc コマンドの最適化オプションを指定します。このオプションはホスト側プログラムに対して適用されます。

```
$ nvcc -01(または-02) test.cu ↵
```

最適化オプションには下記の 4 つがあり、(1) は全く最適化が行なわれず、下に行くほど高度な最適化が行なわれます。各オプションの詳細は、前述のヘルプコマンドや参考文献を参照して下さい。

下記に示すように、無指定だとホスト側プログラムの最適化を全く行なわないので、最低 (1) 以上を指定して下さい。一方あまり高度な最適化を指定すると、副作用（元のプログラムと動作が変わる可能性のある最適化を行なう）が発生したり、コンパイラ自体のバグが現れる確率が高くなるので、避けた方が無難です。従って、以下の (1) (または (2)) を指定し、速度がさほど変わらない場合は (1) を指定するのがよいでしょう。

また、以下の (1)〜(3) を指定して、ホスト側プログラムで、コンパイラのバグが疑われる現象が発生したときは、(0) を指定し、同じ問題が発生するかどうかを確認して下さい。

```
最適化を行なわない ┐(0) -00(大文字のｵｰとｾﾛ) または 無指定
                   │(1) -01 または -0
                   │(2) -02
最も高度な最適化を行なう↓(3) -03
```

nvopencc コマンドの最適化オプション

図 2-7-2 の①の「-Xopencc」は、前述の nvopencc コマンドに対するオプションを指定する場合に使用し、デバイス側プログラム（カーネル関数）に対して適用されます。ところが前述のように、nvopencc に関する説明用コマンドやマニュアルが見つからなかったので、オプションの詳細は不明です。

一方、②のオプション（詳細は後述します）を指定すると、test.ptx というファイルにカーネル関数のアセンブラリストが作成され、その中に図 2-7-3 (1) が表示されます。①で例えば「-00（大文字のオーとゼロ）」を指定した場合、③でも「-00」が表示されました。他の最適化オプションについてもテストしたところ、①の指定と③の表示の対応は図 2-7-3 (2) のようになっていました。

nvopencc コマンドの最適化オプションは、通常の①の全体を無指定 (自動的に -03 が指定されます) でよいと思われます。もしデバイス側プログラム（カーネル関数）で、コンパイラのバグが疑われる現象が発生したときは、まず①で -00（大文字のオーとゼロ）を指定し、同じ問題が発生するかどうかを確認して下さい。

```
nvcc -Xopencc -00(大文字のｵｰとｾﾛ) -ptx test.cu ↵
      ①                            ②
```

図 2-7-2

```
    :
//--------------------------------------------------------
// Options:
//--------------------------------------------------------
//  Target:ptx, ISA:sm_10, Endian:little, Pointer Size:64
//  -00 (Optimization level)  ③
//  -g0 (Debug level)
//  -m2 (Report advisories)
//--------------------------------------------------------
    :
```

図 2-7-3 (1)

①の指定	③の表示
無指定	-03
-0	-02
-00	-00
-01	-01
-02	-02
-03	-03

図 2-7-3 (2)

-arch オプション

図 2-7-4 の下線部で、Compute Capability を指定します。

– ①のように何も指定しない場合のデフォルトは②になります。
– アトミック関数 (4-1 節参照) を使用する場合は、③〜⑤のいずれかを指定しないとコンパイルエラーになります。
– カーネル関数で倍精度実数を使用する場合は、⑤を指定しないと誤作動します (4-2 節参照)。

アトミック関数や倍精度実数を使用しない場合、⑤以外でも動きますが、理研 RICC の環境の Compute Capability 1.3 に合わせ、念のため、常に⑤の「-arch=sm_13」を指定することをお勧めします。

```
nvcc              test.cu ↵  ①
nvcc -arch=sm_10  test.cu ↵  ②
nvcc -arch=sm_11  test.cu ↵  ③
nvcc -arch=sm_12  test.cu ↵  ④
nvcc -arch=sm_13  test.cu ↵  ⑤
```

図 2-7-4

指定が望ましい最適化オプション（まとめ）

以上より、理研 RICC の環境では、下記の 2 つのオプションの指定が望ましいと思われます。なお、本書では、紙面の関係で、下記のオプションの指定は省略します。

```
$ nvcc -O1(または-O2) -arch=sm_13 test.cu ↵
```

コンパイルの各ステップの状況

nvcc コマンドでのコンパイルは、10 以上のステップに分かれています。図 2-7-5 (1) の①の「-v」オプションを指定すると、各ステップのコンパイルコマンド、指定されたオプション、使用する作業ファイル（下線部）などが表示されます。コンパイルが終了すると、作業ファイルは削除されます。

作業ファイルの中身を確認するために保管したい場合、図 2-7-5 (2) の②を指定して下さい。

```
$ nvcc (-c) -v test.cu ↵  ①
  :
#$ gcc -D__CUDA_ARCH__=100 -E -x c++ -DCUDA_NO_SM_12_ATOMIC_INTRINSICS
-DCUDA_NO_SM_13_DOUBLE_INTRINSICS   -DCUDA_FLOAT_MATH_FUNCTIONS
-DCUDA_NO_SM_11_ATOMIC_INTRINSICS  "-I/usr/local/cuda/bin/../include"
"-I/usr/local/cuda/bin/../include/cudart"   -I. -D__CUDACC__ -C  -include "cuda_runtime.h"
-m64 -o "/tmp/tmpxft_0000672d_00000000-4_test.cpp1.ii" "test.cu"
  :
#$ nvopencc  -TARG:compute_10 -m64 -CG:ftz=1 -CG:prec_div=0 -CG:prec_sqrt=0
"/tmp/tmpxft_0000672d_00000000-7_test.cpp3.i"  -o "/tmp/tmpxft_0000672d_00000000-2_test.ptx"
  :
```

図 2-7-5 (1)

```
$ nvcc (-c) -keep test.cu ↵  ②
$ ls ↵
a.out          test.cpp4.ii    test.cudafe1.cpp     test.cudafe2.gpu     test.o
test.cpp1.ii   test.cu         test.cudafe1.gpu     test.cudafe2.stub.c  test.ptx
test.cpp2.i    test.cu.cpp     test.cudafe1.stub.c  test.fatbin.c        test.sm_10.cubin
test.cpp3.i    test.cudafe1.c  test.cudafe2.c       test.hash
```

図 2-7-5 (2)

カーネル関数で使用する資源

図 2-7-6 (1) の①の「-Xptxas -v」オプションを指定して、図 2-7-6 (2) の test.cu をコンパイルすると、カーネル関数 (本例では関数 kernel) が使用するメモリ資源の情報が表示されます。

②の表示は、1 スレッドあたり 2 つのレジスターを使用し、1 ブロックあたり 8 + 16 バイトのシェアードメモリを使用することを示します (詳細は 3-5 節〜3-8 節参照)。

③のように、test.cu の全体ではなく、カーネル関数 kernel.cu のみを指定することも可能です。この場合はリンクを行なわないようにするため、「-c」オプションを同時に指定して下さい。

```
$ nvcc -Xptxas -v -arch=sm_13 -c test.cu  ↵  ①
ptxas info    : Compiling entry function '_Z6kernelPf' for 'sm_13'
ptxas info    : Used 2 registers, 8+16 bytes smem  ②
$ nvcc -Xptxas -v -arch=sm_13 -c kernel.cu  ↵  ③
```

図 2-7-6 (1)

```
int main(void){          | __global__ void kernel(){
  :                      |   :
}                        | }
```

test.cu　　kernel.cu

図 2-7-6 (2)

レジスター数の制限

あるプログラムのカーネル関数が、図 2-7-7 の①, ②に示すように、スレッドあたりレジスターを 14 個使用しているとします。何らかの理由で（後述）使用するレジスター数を減らしたい場合、スレッドあたり使用するレジスター数の上限を設定することができます。

③の二重線に示すように、スレッドあたりのレジスター数の上限を 8 個に指定すると、④の二重線に示すように使用するレジスター数は 8 個となり、残りは波線に示すように低速なローカルメモリ (lmem) (3-8 節参照) 上に確保されます。

③の二重線で指定できるスレッドあたりのレジスター数の上限値は、124 個以下の値です。ただし、指定した値によってはコンパイルが終了していないこともあるようなので、その場合は「Cntl」と「C」を同時に押してコンパイルをキャンセルして下さい。

この方法は、使用するレジスターの数を減らすことによって、1 つのストリーミング・マルチプロセッサー上に同時に存在できるワープの数を増やして高速化する場合などに使用しますが (詳細 6-1 節参照)、高速なレジスターの代わりに低速なローカルメモリが使用されるため、却って遅くなる可能性もあります。

```
$ nvcc -Xptxas -v -arch=sm_13 -c test.cu  ↵                          ①
ptxas info    : Compiling entry function '_Z6kernelv' for 'sm_13'
ptxas info    : Used 14 registers, 4 bytes cmem[14]                  ②
$ nvcc -maxrregcount 8 -Xptxas -v -arch=sm_13 -c test.cu  ↵          ③
ptxas info    : Compiling entry function '_Z6kernelv' for 'sm_13'
ptxas info    : Used 8 registers, 64+0 bytes lmem, 4 bytes cmem[14]  ④
```

図 2-7-7

組込関数の高速版の使用

プログラム内で使用している組込関数 (例えば sin(x)) を、高速版の組込関数 (例えば __sinf(x)) に自動的に置き換える場合に指定します (詳細は 6-6 節参照)。ただし、元の組込関数と計算結果が若干変わる可能性があるので注意して下さい。

```
nvcc -use_fast_math test.cu  ↵
```

アセンブラリスト

デバイス側のプログラム（カーネル関数）を、コンパイラがどのように最適化しているかを調べるときに、アセンブラリストが参考になります。以下で、アセンブラリストの見方の概要を説明します。詳細は、「PTX: Parallel Thread Execution ISA Version 2.1」（付録参照）を参照して下さい。

【例 1】基本的な加算

図 2-7-8 の (1) または (2) の下線部を付けて、図 2-7-9 (1) のカーネル関数 kernel.cu をコンパイルすると、図 2-7-9 (3) に示すアセンブラリスト (ファイル名は kernel.ptx) が作成されます。(1) を指定した場合、実線の行と二重線の行が両方表示され、(2) を指定した場合は実線の行は表示されません。

実線の行は、図 2-7-9 (1) の各ステートメントを示し、例えば図 2-7-9 (3) の (3) のステートメントに対応するアセンブラ命令が、後続する①〜⑬に表示されます。また二重線の例えば (4) は、(5) に示す 27 番ファイル (図 2-7-9 (1) の kernel.cu) 内の 3 行目に対応するアセンブラ命令が、後続する①〜⑬に表示されることを示します。

```
nvcc -ptx -Xopencc -LIST:source=on kernel.cu ↵  (1)
nvcc -ptx kernel.cu ↵                           (2)
```

図 2-7-8

図 2-7-9 (3) の (6) は、アセンブラコード内で使用するレジスターを示します。u (u16 など) は符号なし整数、f は実数、s（本例では未使用）は符号付き整数です。また 16，32，64 はビット数です。例えば (7) は、32 ビット (4 バイト) の実数（つまり単精度実数）のレジスター f1〜f5 を示します（ただし、アセンブラ命令内で全部使用されるとは限りません)。

①〜⑬の下線部は、アセンブラ命令の動作を表します。ld はメモリからレジスターへのロード、st はレジスターからメモリへのストア、mov は代入、add は加算、mul は乗算、cvt は型変換を示します。

以下で①〜⑬の動作概要を説明します。図 2-7-9 (3) の右側に、各アセンブラ命令の意味を C 言語風に示します。またデータ動きを図 2-7-9 (2) に示します。

図 2-7-9 (1) で、〓に示す値が設定されているとします。従って、図 2-7-9 (1) の1は i=10 となり、2は dA[11] = dA[2] + 3.4 (= 1.0+3.4 = 4.4) となります。まとめると、図 2-7-9 (1) のプログラムでは、図 2-7-9 (2) の、dA[2] に入っている 1.0 に 3.4 を加算し、その結果の 4.4 を dA[11] に代入します。

- ①で、配列 dA の先頭アドレス (本例では図 2-7-9 (2) の①に示すように例えば 128 バイト目) を、レジスター rd1 にロード (ld) します (図 2-7-9 (2) の①参照、以下同様)。
- ②で、rd1 内の値に 8 バイト (単精度実数 2 要素分) を加えたアドレス (136 バイト目) から開始する要素 (つまり dA[2]) のデータを、レジスター f1 にロード (ld) します。
- ③で、定数 3.4f をレジスター f2 に代入 (mov) します。③の右側に定数の値 (3.4) が表示されます。
- ④で、レジスター f1 と f2 を加算 (add) し、結果をレジスター f3 に代入します。
- ⑤の、tid.x には、threadIdx.x が入っています。これをレジスター r1 に代入します。tid.x は 16 ビット、r1 は 32 ビットなので、代入時に型変換 (cvt) を行ないます。
- ⑥の ctaid.x には、blockIdx.x が入っています。これをレジスター rh1 に代入 (mov) します。
- ⑦の ntid.x には、blockDim.x が入っています。これをレジスター rh2 に代入 (mov) します。
- ⑧で、レジスター rh1 と rh2 を掛けて (mul)、結果をレジスター r2 に代入します (つまり r2 = blockIdx.x*blockDim.x となります)。
- ⑨で、レジスター r1 と r2 を加算 (add) し、結果をレジスター r3 に代入します (つまり r3 = blockIdx.x+blockIdx.x*blockDim.x となります)。
- ⑩で、レジスター r3 をレジスター rd2 に型変換 (cvt) しています。ただし、rd2 をその後使用していないため、⑩の命令は不要だと思われます。
- ⑪で、レジスター r3 に 4 を掛けて (mul)、単位をバイトに変換し、レジスター rd3 に代入します。rd3 (40 バイト) は、配列 dA の先頭アドレスから、要素 dA[10] の先頭アドレスまでの変位（バイト）を表します。
- ⑫で、レジスター rd1 と rd3 を加算 (add) し、結果をレジスター rd4 に代入します。rd4 は、dA[10] の先頭アドレスを示します。
- ⑬で、レジスター f3 の値を、rd4 内の値に 4 バイト (単精度実数 1 要素分) を加えたアドレスから開始する要素 (つまり dA[11]) にストア (st) します。

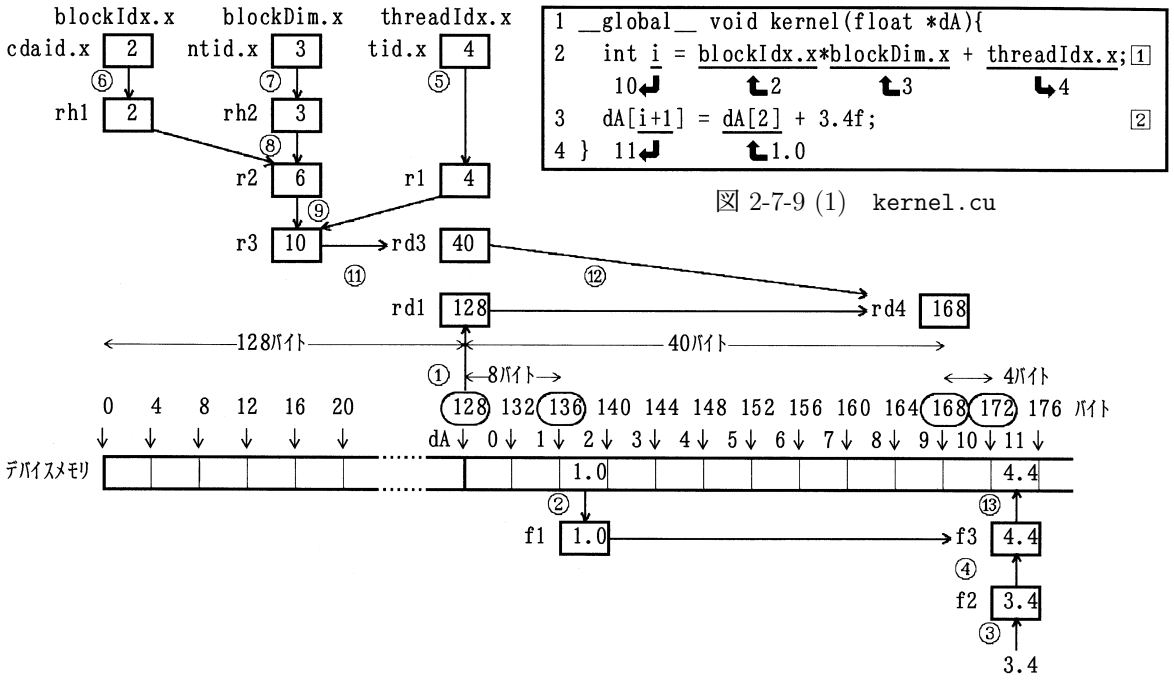


図 2-7-9 (1)　kernel.cu

図 2-7-9 (2)

```
        :
        .file   27      "kernel.cu"                          (5)
        .entry _Z6kernelPf (
                .param .u64 __cudaparm__Z6kernelPf_dA)
        {
        .reg .u16 %rh<4>;
        .reg .u32 %r<5>;                                     (6)
        .reg .u64 %rd<6>;
        .reg .f32 %f<5>;                                     (7)
        .loc    27      1       0
//   1  __global__ void kernel(float *dA){
$LDWbegin__Z6kernelPf:
        .loc    27      3       0                            (4)
//   2     int i = blockIdx.x*blockDim.x + threadIdx.x;       (3)
//   3     dA[i+1] = dA[2] + 3.4f;                            (3)
        ld.param.u64    %rd1, [__cudaparm__Z6kernelPf_dA]; ①
        ld.global.f32   %f1, [%rd1+8];      ②
        mov.f32         %f2, 0f4059999a;    ③    // 3.4
        add.f32         %f3, %f1, %f2;      ④
        cvt.u32.u16     %r1, %tid.x;        ⑤
        mov.u16         %rh1, %ctaid.x;     ⑥
        mov.u16         %rh2, %ntid.x;      ⑦
        mul.wide.u16    %r2, %rh1, %rh2;    ⑧
        add.u32         %r3, %r1, %r2;      ⑨
        cvt.s64.s32     %rd2, %r3;          ⑩
        mul.wide.s32    %rd3, %r3, 4;       ⑪
        add.u64         %rd4, %rd1, %rd3;   ⑫
        st.global.f32   [%rd4+4], %f3;      ⑬
        .loc    27      4       0
//   4  }
        exit;
$LDWend__Z6kernelPf:
        } // _Z6kernelPf
```

【C言語風ｺｰﾄﾞ】

```
f1  = dA[0+2]
f2  = 3.4f
f3  = f1 + f2
r1  = threadIdx.x
rh1 = blockIdx.x
rh2 = blockDim.x
r2  = rh1*rh2
r3  = r1 + r2

rd3 = r3*4
rd4 = rd1 + rd3
dA[10+1] = f3
```

図 2-7-9 (3)

【例 2】各種メモリ

例 2〜例 4 でいくつか補足します。図 2-7-10 (1) のアセンブラリストを図 2-7-10 (2) に示し、右側に、各アセンブラ命令の意味を C 言語風に示します。

- ⓪と①を比較すると分かるように、アセンブラリストでは、配列の大きさは、個数 (10) でなく、バイト ($40 = 10 \times 4$) で表示されます。
- ①，⑨，⑫の下線部に示すように、グローバルメモリからのロード命令と、グローバルメモリへのストア命令には、global が付きます。
- ②，⑦，⑩の下線部に示すように、シェアードメモリからのロード命令と、シェアードメモリへのストア命令には、shared が付きます。
- ③，④の下線部に示すように、コンスタントメモリからのロード命令には、const が付きます。
- ⑤の下線部に示すように、引数 (本例では変数 X) からのロード命令には、param が付きます。
- ⑥，⑪に示すように、減算命令は sub、除算命令は div です。
- ⑧に示すように、__syncthreads() に対応する同期命令は bar.sync です。

```
1   __device__ float dD[10];      ⓪
2   __shared__ float dS[20];
3   __constant__ float dC[30];
4 __global__ void kernel(float X){
5   dS[1] = dC[2] - X;
6   __syncthreads();
7   dD[3] = dD[4]/dS[5];
8 }
```

図 2-7-10 (1)

```
        :
      .global .align 4 .b8 dD[40];     ①
      .shared .align 4 .b8 dS[80];     ②
      .const .align 4 .b8 dC[120];     ③
        :
//   5    dS[1] = dC[2] - X;                                  【C言語風ｺｰﾄﾞ】
      ld.const.f32    %f1, [dC+8];     ④                    f1 = dC[2]
      ld.param.f32    %f2, [__cudaparm__Z6kernelf_X]; ⑤     f2 = X
      sub.f32         %f3, %f1, %f2;   ⑥                    f3 = f1 - f2
      st.shared.f32   [dS+4], %f3;     ⑦                    dS[1] = f3
      .loc    15      6       0
//   6    __syncthreads();
      bar.sync        0;               ⑧                    __syncthreads()
      .loc    15      7       0
//   7    dD[3] = dD[4]/dS[5];
      ld.global.f32   %f4, [dD+16];    ⑨                    f4 = dD[4]
      ld.shared.f32   %f5, [dS+20];    ⑩                    f5 = dS[5]
      div.full.f32    %f6, %f4, %f5;   ⑪                    f6 = f4/f5
      st.global.f32   [dD+12], %f6;    ⑫                    dD[3] = f6
        :
```

図 2-7-10 (2)

【例 3】if 文

図 2-7-11 (1) は if 文が含まれる例です。アセンブラリストを図 2-7-11 (2) に示し、右側に、各アセンブラ命令の意味を C 言語風に示します。

- ①で r1 と r2 を比較し、r1<=r2 (le: less than equal) なら p1 を真に、それ以外なら p1 を偽にします。
- ②で p1 の値が真なら、$Lt_0_1282 に分岐 (bra) します。
- ③で、$Lt_0_1026 に無条件分岐 (bra.uni) します。

```
1   __device__ int dD[1];
2 __global__ void kernel(){
3   if(dD[0]>9){
4     dD[0] = 2;
5   }else{
6     dD[0] = dD[0] + 3;
7   }
8 }
```

図 2-7-11 (1)

```
      :
    ld.global.s32  %r1, [dD+0];
    mov.u32        %r2, 9;
    setp.le.s32    %p1, %r1, %r2;①
    @%p1 bra       $Lt_0_1282;   ②
    .loc    15      4       0
    mov.s32        %r3, 2;
    st.global.s32  [dD+0], %r3;
    bra.uni        $Lt_0_1026;   ③
$Lt_0_1282:
    .loc    15      6       0
    add.s32        %r4, %r1, 3;
    st.global.s32  [dD+0], %r4;
$Lt_0_1026:
    .loc    15      8       0
    exit;
      :
```

```
【C言語風ｺｰﾄﾞ】
r1 = dD[0]
r2 = 9
if(r1<=r2) p1=真 else p1=偽
if(p1==真) goto 1282
r3 = 2
dD[0] = r3
goto 1026
1282:
r4 = r1 + 3
dD[0] = r4
1026:
```

図 2-7-11 (2)

【例 4】for ループ

図 2-7-12 (1) は for ループが含まれる例です。アセンブラリストを図 2-7-12 (2) に示し、右側に、各アセンブラ命令の意味を C 言語風に示します。

- ④で r1 と r3 を比較し、r1 と r3 が等しくない (ne: not equal) なら p1 を真に、それ以外なら p1 を偽にします。
- ⑤で p1 の値が真なら、$Lt_0_1794 に分岐 (bra) します。これは、図 2-7-12 (1) の for ループの反復に相当します。

```
1   __device__ int dD[100];
2 __global__ void kernel(){
3   for(int i=0;i<100;i++){
4     dD[i] = 2.0f;
5   }
6 }
```

図 2-7-12 (1)

```
      :
    mov.u64        %rd1, dD;
    mov.s32        %r1, 0;
$Lt_0_1794:
 //<loop> Loop body line 961,
  nesting depth: 1, iterations: 100
    .loc    15      4       0
    mov.s32        %r2, 2;
    st.global.s32  [%rd1+0], %r2;
    add.s32        %r1, %r1, 1;
    add.u64        %rd1, %rd1, 4;
    mov.u32        %r3, 100;
    setp.ne.s32    %p1, %r1, %r3;④
    @%p1 bra       $Lt_0_1794;   ⑤
    .loc    15      6       0
    exit;
      :
```

```
【C言語風ｺｰﾄﾞ】
(i = 0)
r1 = 0
1794:
r2 = 2
dD[i] = r2
r1 = r1 + 1
rd1 = rd1 + 4 (i = i + 1)
r3 = 100
if(r1!=r3) p1=真 else p1=偽
if(p1==真) goto 1794
```

図 2-7-12 (2)

2-8　CUDA 化の手順

プログラムを CUDA 化する場合、慣れないうちは、「少し修正しては、テストして結果を確認し、…」を繰り返すのが一つの方法です。これを、図 2-8-1 の (2) の部分を CUDA 化する場合で説明します。

- まず、(2) の部分を、図 2-8-2 (1) の①，⑤に示すように関数にします。さらに配列 A とは別の配列 dA を関数側で使用するように、②，③，④，⑥，⑦を追加し、テストします。なお、図 2-8-1 の (2) のループ反復を、図 2-8-2 (1) では関数側の①に入れましたが、図 2-8-2 (2) の①，⑤のようにメインルーチンに残す方法もあります。
- 次に図 2-8-3 (1) の1の下線部を追加し、3〜7を修正し、5に示すように 1 ブロック、1 スレッドでテストします。この段階で、図 2-8-2 (1) と計算結果が異なる場合は、ホスト側とデバイス側の、組込関数 (sinf など) の精度の相違や、コンパイラの最適化の相違が考えられます。図 2-8-2 (2) の場合は図 2-8-3 (2) のように修正します。本例では省略しましたが、この段階でエラーチェックルーチン (4-2 節参照) も付加します。
- 次に、図 2-8-4（紙面右上）の [0] と [1] の下線部を追加し、[5] に示すように（可能であれば）1 ブロック複数スレッドでテストします。最後に [6] に示すように、複数ブロック、複数スレッドでテストします。

```
#define N (100)
int main(void){
  int i;
  float A[N];
  for(i=0;i<N;i++){                    (1)
    A[i] = (float)i;
  }
  for(i=0;i<N;i++){
    A[i] = A[i] + sinf(float(i));      (2)
  }
   :
```

図 2-8-1

```
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;  [0]
  if(i<N) dA[i] = dA[i] + sinf(float(i));       [1]
}
   :
( kernel<<<1,128>>>(dA); )                      [5]
  kernel<<<4,32>>>(dA);                         [6]
   :
```

図 2-8-4

```
void kernel(float *dA){
  for(int i=0;i<N;i++){
    dA[i] = dA[i] + sinf(float(i));    ①
  }
}
 int main(void){
  (1)と同じ
  float *dA;                           ②
  size_t size = N*sizeof(float);       ②
  dA = (float*)malloc(size);           ③
  for(i=0;i<N;i++) dA[i] = A[i];       ④
  kernel(dA);                          ⑤
  for(i=0;i<N;i++) A[i] = dA[i];       ⑥
  free(dA);                            ⑦
    :
```

図 2-8-2 (1)

```
__global__ void kernel(float *dA){
  for(int i=0;i<N;i++){
    dA[i] = dA[i] + sinf(float(i));    [1]
  }
}
 int main(void){
  (1)と同じ
  float *dA;
  size_t size = N*sizeof(float);
  cudaMalloc((void**)&dA,size);                          [3]
  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);          [4]
  kernel<<<1,1>>>(dA);                                   [5]
  cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);          [6]
  cudaFree(dA);                                          [7]
    :
```

図 2-8-3 (1)

```
void kernel(float *dA,int i){
  dA[i] = dA[i] + sinf(float(i));      ①
}
    :
 for(i=0;i<N;i++){
    kernel(dA,i);                      ⑤
  }
    :
```

図 2-8-2 (2)

```
__global__ void kernel(float *dA,int i){
  dA[i] = dA[i] + sinf(float(i));      [1]
}
    :
 for(i=0;i<N;i++){
    kernel<<<1,1>>>(dA,i);             [5]
  }
    :
```

図 2-8-3 (2)

第3章　基本編（メモリ構成）

GPU では、通常の計算機よりもメモリ構成が複雑です。効率の良い CUDA プログラムを作成するためには、各メモリの特性を理解する必要があります。

3-1　メモリ構成の概要

本節では、まず、GPU のメモリ構成の概要を説明し、次節以降で個々のメモリを説明します。

図 3-1-1 で、左側がホスト (CPU)、右側がデバイス (GPU) です。デバイス側の下半分は、オフチップメモリ（大容量：低速）であるデバイスメモリを示し、グローバルメモリ、コンスタントメモリ、ローカルメモリ、テクスチャーメモリ（本書では説明しません）から構成されます。

デバイス側の上半分は、ストリーミング・マルチプロセッサー (図では 2 個ですが、実際には 30 個) を示し、それぞれの中に、レジスター、シェアードメモリ、コンスタントキャッシュなどのオンチップメモリ（小容量：高速）が搭載されています。なお、図の右欄に示す各メモリの容量については、「CUDA C Programming Guide」(Appendix G.) を参照して下さい。

各部の転送速度を A〜D に示します。B の転送速度が最も遅いので、ホストとデバイス間のコピーは最小限にする必要があります。C のデバイスメモリの転送速度は、ホスト側メモリの A に比べると速いですが（ただしデバイス側はホスト側と違ってキャッシュがありません)、D のオンチップメモリと比べると 100 倍以上遅いので、C のロード／ストアもなるべく少なくする必要があります。

以下で、図 3-1-2 のプログラムを例に、各配列／変数の特性を説明します。

各変数／配列が作成される場所

- ③，④で指定した配列 dA、⑭で指定した配列 dD はグローバルメモリ上に、⑮で指定した配列 dC はコンスタントメモリ上に作成されます。ただし dC は、カーネル関数から参照されると、近隣の要素とともにコンスタントキャッシュにコピーされます（詳細は後述します)。
- ⑯または⑱で指定した配列 dS は、ブロックごとにシェアードメモリ上に作成されます。
- ⑰の仮引数で指定した変数 dI は、ブロックごとにシェアードメモリ上に作成されます。
- ⑲で指定したカーネル関数のローカル変数 dL は、スレッドごとにレジスターに作成されます。レジスターを使いきった場合は、ローカルメモリ上に作成されます。

各変数／配列にアクセスできるスレッドの範囲

- 図 3-1-1 の矢印から分かるように、デバイスメモリ上の配列 dA，dD，dC は、全ブロックの全スレッドからアクセス可能です。ただし dC は参照のみで、更新はできません。
- シェアードメモリ上の配列 dS と仮引数 dI は、当該ブロック内の全スレッドからのみアクセス可能です。
- レジスター上のローカル変数 dL は、当該スレッドからのみアクセス可能です。

各変数／配列がメモリ上に存在する期間

- デバイスメモリ上の配列 dA は、図 3-1-2 の④で作成され、⑬で解放されます。
- デバイスメモリ上の上の配列 dD と dC は、プログラムの開始から終了までの間、存在します。
- シェアードメモリ上の配列 dS と仮引数 dI は、当該ブロックがストリーミングマルチプロセッサー上に存在している間、存在します。
- レジスター上のローカル変数 dL は、当該スレッドの開始から終了までの間、存在します。

各変数／配列の値の設定方法

- デバイスメモリ上の配列 dA は、⑥で A から dA にコピーされ、⑩で dA から A にコピーされます。
- デバイスメモリ上の配列 dD は、⑦で D から dD にコピーされ、⑪で dD から D にコピーされます。
- デバイスメモリ上の配列 dC は、⑧で C から dC にコピーされます。配列 dC は参照のみ可能なので、dC から C へのコピーはできません。
- シェアードメモリ上の配列 dS は、⑳で値を設定／参照／更新します。
- カーネル関数の仮引数 dI は、⑨，⑰で I から dI に自動的にコピーされます。
- カーネル関数のローカル変数 dL は、⑳で値を設定／参照／更新します。

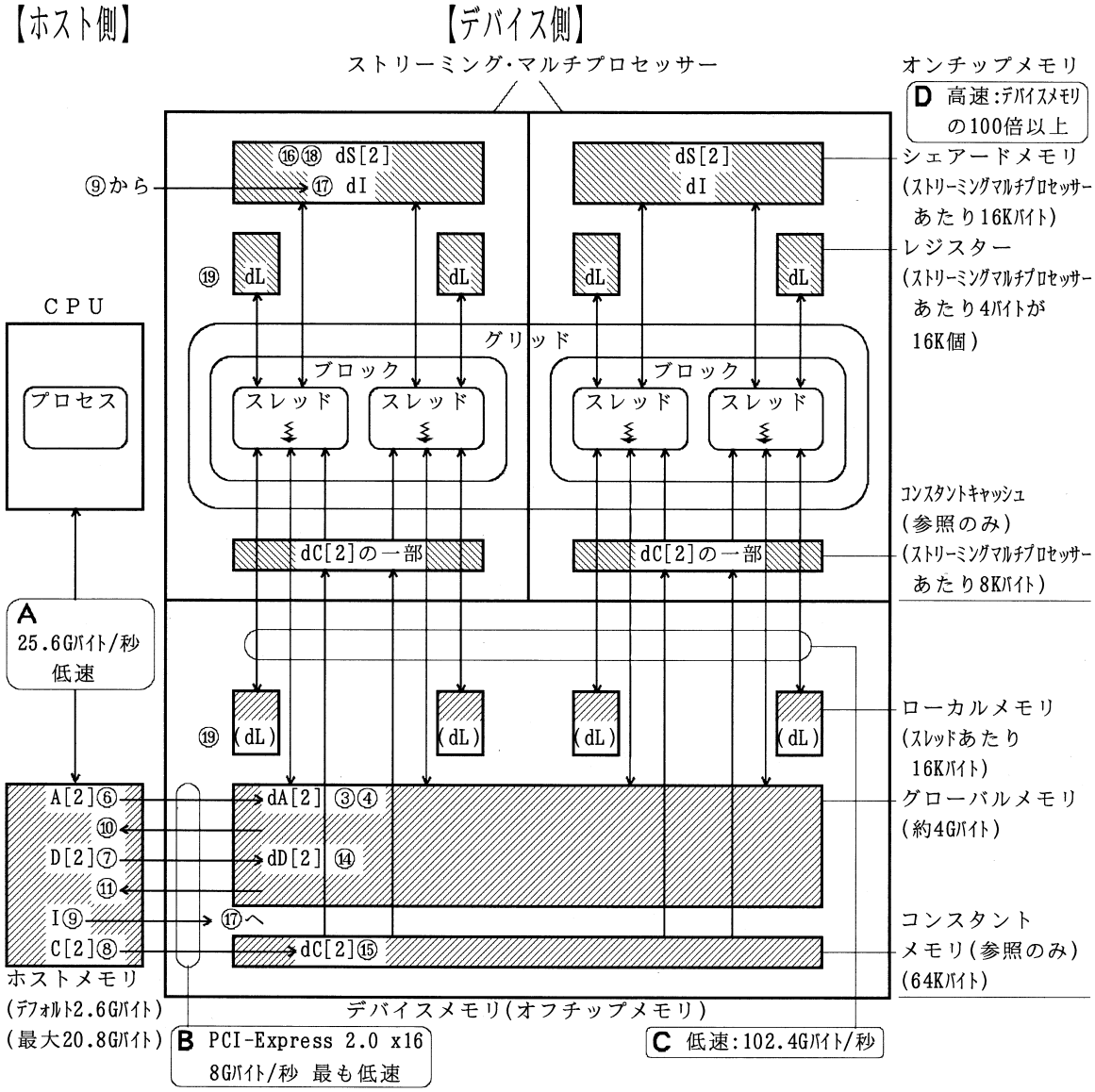


図 3-1-1

ホスト側プログラム

```
int main(void){
  float A[2],D[2],C[2];          ①
  int I;                         ②
  float *dA;                     ③
  cudaMalloc((void**)&dA,～);    ④
  A,D,C,Iに値を設定              ⑤
  cudaMemcpy(dA,A,～);           ⑥
  cudaMemcpyToSymbol(dD,D,～);   ⑦
  cudaMemcpyToSymbol(dC,C,～);   ⑧
  kernel<<<2,2>>>(dA,I);         ⑨
  cudaMemcpy(A,dA,～);           ⑩
  cudaMemcpyFromSymbol(D,dD,～); ⑪
  A,Dを参照                      ⑫
  cudaFree(dA);                  ⑬
}
```

グローバル領域

```
  __device__    float dD[2];     ⑭
  __constant__  float dC[2];     ⑮
 (__shared__    float dS[2];)    ⑯
__global__ void kernel(float *dA,int dI){  ⑰
  __shared__ float dS[2];        ⑱
  int dL;                        ⑲
  dA,dD,dS,dLを参照/更新, dI,dCを参照      ⑳
}
```

デバイス側プログラム(カーネル関数)

図 3-1-2

3-2　グローバルメモリ (cudaMalloc で確保)

本節では、CUDA 関数の cudaMalloc を使用して、グローバルメモリ上に変数／配列を確保する方法について説明します（前節までの説明と重複している部分もあります)。

cudaMalloc で確保した変数／配列の特性を以下に示します (3-1 節参照)。

- 作成される場所：デバイスメモリ内のグローバルメモリ（オフチップ：低速）上に作成されます。
- アクセスできるスレッドの範囲：全ブロックの全スレッドからアクセスすることができます。
- 存在する期間：cudaMalloc で確保してから、cudaFree で解放するまでの間、存在します。
- 容量：約 4 G バイトです (ただし __device__ 修飾子で確保したメモリとの合計です)。

指定方法

図 3-2-1 (2) のように、デバイス側のグローバルメモリ上に大きさ 2 の配列 dA を確保し、ホスト側の大きさ 2 の配列 A との間でコピーを行うプログラムを図 3-2-1 (1) に示します。

- ホスト側の配列 A を①で宣言します。図 3-2-1 (4) の⑨のように malloc で確保しても構いません。
- デバイス側の配列 dA を②で宣言し、③，④でグローバルメモリ上に確保します。プログラムの実行時に確保するので、コンパイル／リンク時に大きさが確定していなくても構いません。図 3-2-1 (4) の⑩のように、③を④の引数に直接指定しても構いません。
- ⑤でホスト側の配列 A からデバイス側の配列 dA にコピーし、⑥でカーネル関数を実行し、⑦で配列 dA から配列 A にコピーします。⑤，⑦では 2 つ目の引数から 1 つ目の引数にコピーします。
- ⑧でデバイス側の配列 dA を解放します。
- 図 3-2-1 (4) の⑪の下線部を指定すると、図 3-2-1 (3) のように、デバイス側の配列 dA から dB（配列の宣言は省略）にコピーします。dA，dB はともにデバイス側の配列ですが、ホスト側プログラムで⑪を実行します。
- 配列でなく、スカラー変数をデバイス上に確保してコピーする場合、図 3-2-2 (1) (2) のようになります。

```
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;                          ⓪
}
int main(void){
  float A[2];                                    ①
  配列Aに値を設定する。
  float *dA;                                     ②
  size_t size = 2*sizeof(float);                 ③
  cudaMalloc((void**)&dA,size);                  ④
  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);
  kernel<<<1,2>>>(dA);                           ⑥  ↰⑤
  cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);
  cudaFree(dA);                                  ⑧  ↰⑦
    :
```

図 3-2-1 (1)　配列の場合

```
__global__ void kernel(float *dX){
  *dX = *dX + 1.0f;                              [0]
}
int main(void){
  float X;
  スカラー変数Xに値を設定する。
  float *dX;
  size_t size = sizeof(float);
  cudaMalloc((void**)&dX,size);
  cudaMemcpy(dX,&X,size,cudaMemcpyHostToDevice);
  kernel<<<1,1>>>(dX);                           ↰[5]
  cudaMemcpy(&X,dX,size,cudaMemcpyDeviceToHost);
  cudaFree(dX);                                  ↰[7]
    :
```

図 3-2-2 (1)　スカラー変数の場合

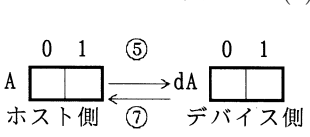


図 3-2-1 (2)

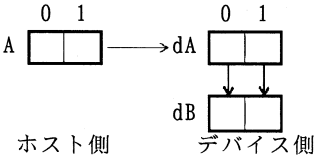


図 3-2-1 (3)

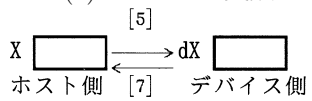


図 3-2-2 (2)

```
float *A;                                  ⑨ | cudaMalloc((void**)&dA,2*sizeof(float));          ⑩
A = (float*)malloc(size);                  ⑨ | cudaMemcpy(dB,dA,size,cudaMemcpyDeviceToDevice); ⑪
```

図 3-2-1 (4)

同期関数と非同期関数

図 3-2-1 (1) のプログラムを実行した場合のタイムチャートを、図 3-2-3 (1) に示します。

- ④で CUDA 関数をコールすると、ホストプログラムは CUDA 関数の処理が終了するまで待機します。このような関数を本書では同期関数と呼びます (⑤，⑧も同期関数です)。図の右のタイムチャートから分かるように、ホストプログラムと同期関数は同時に実行することができません。
- ⑥でカーネル関数をコールすると、ホストプログラムはただちに（待機せずに）次の⑦の処理（がもしあれば）を実行します。このような関数を、本書では非同期関数と呼びます。カーネル関数の他に、CUDA 関数にも非同期関数があります (cudaMemcpyAsync など：6-3 節参照)。タイムチャートから分かるように、ホストプログラムと非同期関数は同時に実行することができます。
- ホストプログラムは⑧で CUDA 関数をコールしますが、このとき⑥でコールされたカーネル関数がすでに稼働しています。この場合、コールされた CUDA 関数は自動的に待機し、カーネル関数が実行を終了すると、CUDA 関数は自動的に実行を開始します。つまり、CUDA 関数とカーネル関数は、（原則的に）一時点でどちらか 1 つのみが、コールされた順番に実行されます (詳細は 6-3 節参照)。
- 図 3-2-3 (2) の6で、ホストプログラムは非同期関数（本例ではカーネル関数）をコールし、次に7と9の処理を実行します。何らかの理由で、6の処理が完了してから9の処理を開始したい場合、8で同期を取るための CUDA 関数 cudaThreadSynchronize() を実行します。するとホストプログラムは8で待機し、8より前にコールされた全ての CUDA 関数とカーネル関数 (本例では6) が終了したら、9の実行を開始します。これによって、9の実行時点で6が完了していることが保証されます。
- 図 3-2-4 (1) (2) では、タイムステップループが反復するごとに、カーネル関数をコールしています。カーネル関数は非同期関数なので、本例ではカーネル関数が一気に 100 回コールされます (関数自体は 1 つずつ処理されます)。テストしたところ、図 3-2-4 (1) でも正常に動作するようですが、安全のため、図 3-2-5 (1) のように下線部で同期を取る関数を指定し、図 3-2-5 (2) のように処理した方が無難かもしれません。

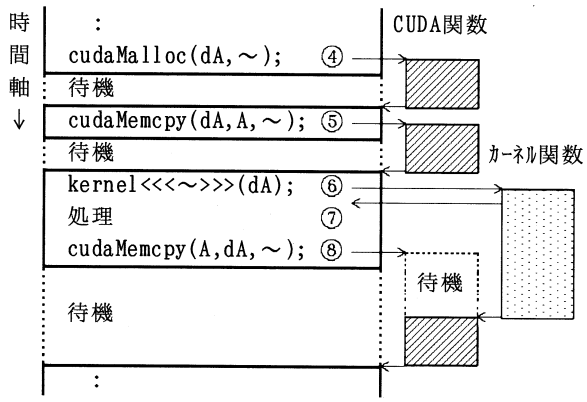


図 3-2-3 (1)

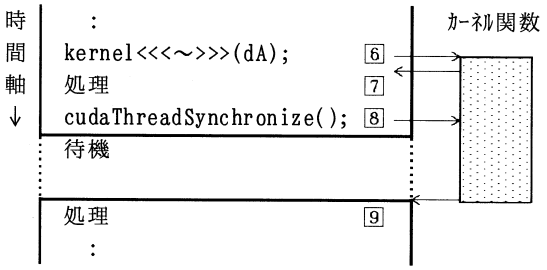


図 3-2-3 (2)

```
    :
for(itime=0;itime<100;itime++){
  kernel<<<～>>>(dA);
}
    :
```

図 3-2-4 (1)

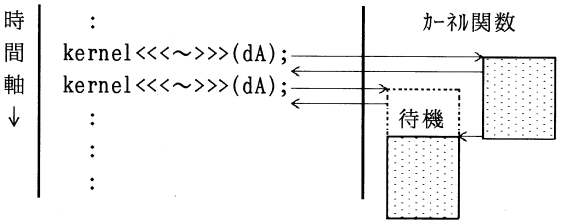


図 3-2-4 (2)

```
    :
for(itime=0;itime<100;itime++){
  kernel<<<～>>>(dA);
  cudaThreadSynchronize();
}
    :
```

図 3-2-5 (1)

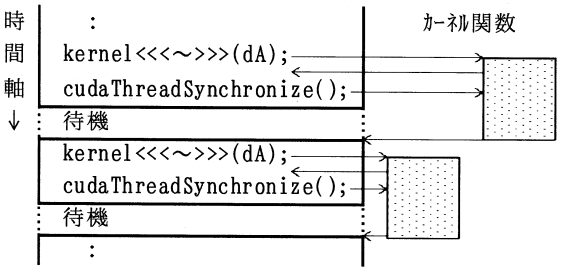


図 3-2-5 (2)

グローバルメモリとトランザクション

図 3-2-6 (1) の下段にグローバルメモリを示します。図の 1 マスは 4 バイト (単精度実数だと 1 要素分) です。グローバルメモリの先頭 (0 バイト目) からの変位が、32 バイト (8 要素) の倍数 (0 バイト，32 バイト，64 バイト，...) になっている「↓」の位置を、32 バイト境界と呼びます。同様に、「↓」は 64 バイト境界、「↓」は 128 バイト境界になります。

GPU では、グローバルメモリとレジスターの間のロード／ストアは、4 バイト (単精度の場合 1 要素) ずつ行うのではなく、図 3-2-6 (2) に示すように、32 バイト (単精度の場合 8 要素)、64 バイト (単精度の場合 16 要素)、128 バイト (単精度の場合 32 要素) のいずれかの単位で行われます（どの単位で行われるかは後述します)。これらの単位を、本書ではトランザクション (元の英語は memory transaction) と呼びます。

トランザクションは、開始する位置が決まっており、図 3-2-6 (1) (2) に示すように、32 バイトトランザクションは 32 バイト境界から開始します。同様に 64，128 バイトトランザクションは 64，128 バイト境界から開始します。

「CUDA C Programming Guide」(付録参照) の 5.3.2.1.1 節によると、「実行時 API (cudaMalloc などの CUDA 関数のこと) を使用して確保した変数（スカラー変数と配列を意味すると思われます）は、グローバルメモリ上の少なくとも 256 バイト境界から開始する。」と記載されています。従って、例えば図 3-2-7 の②，③で確保した単精度の配列 dA[10] と dB[10] は、図 3-2-6 (3) に示すように（少なくとも）256 バイト境界から開始します。

図 3-2-7 の④でブロック数 1、ブロック内のスレッド数 1 でカーネル関数を実行し、①で、例えば要素 dA[3] (図 3-2-6 (3) の「3」の部分) をレジスターにロード、加算し、結果を dA[3] にストアします。この場合、上記で説明したように、dA[3] だけをロード／ストアするのでなく、図 3-2-6 (3) に示すように、dA[3] を含む 32 バイトトランザクションをロード／ストアします。

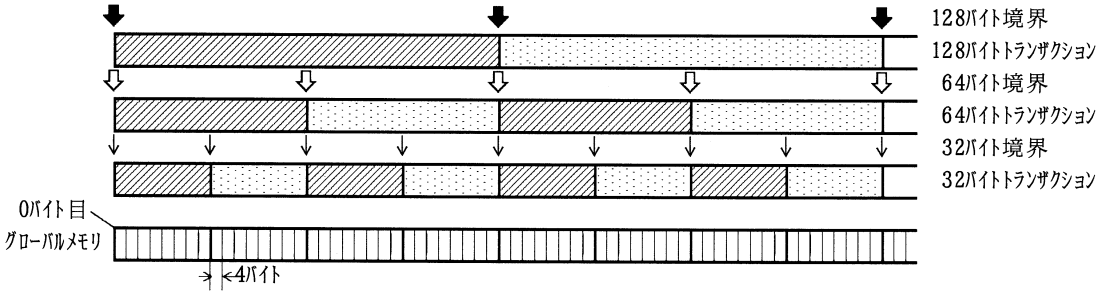


図 3-2-6 (1)

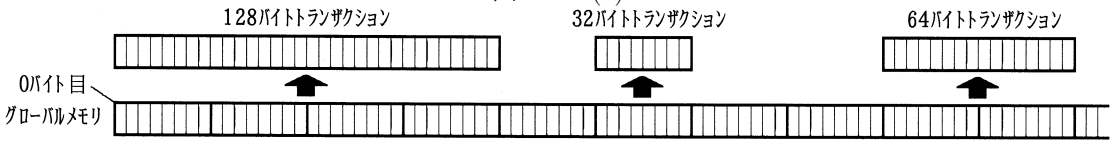


図 3-2-6 (2)

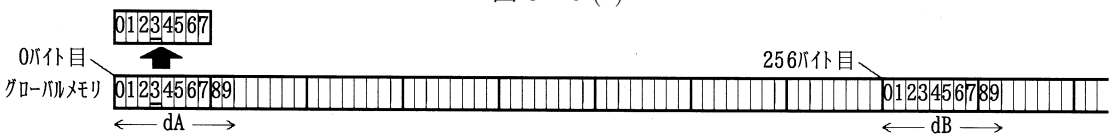


図 3-2-6 (3)

```
__global__ void kernel(float *dA){
  dA[3] = dA[3] + 1.0f;          ①
}
```

```
int main(void){
  float *dA,*dB;
  cudaMalloc((void**)&dA,10*sizeof(float)); ②
  cudaMalloc((void**)&dB,10*sizeof(float)); ③
    :
  kernel<<<1,1>>>(dA);                      ④
    :
```

図 3-2-7

コアレスアクセス

図 3-2-8 は 1 つのブロックを示します。2-4 節で説明したように、ブロック内の連続した 32 スレッドをワープと呼び、計算はワープ単位に行われます。ワープ内の前半 (①の部分) と後半 (②の部分) の 16 スレッドをそれぞれハーフワープと呼びます。以下で説明するように、グローバルメモリからのロード／ストアはハーフワープ単位で行われます。

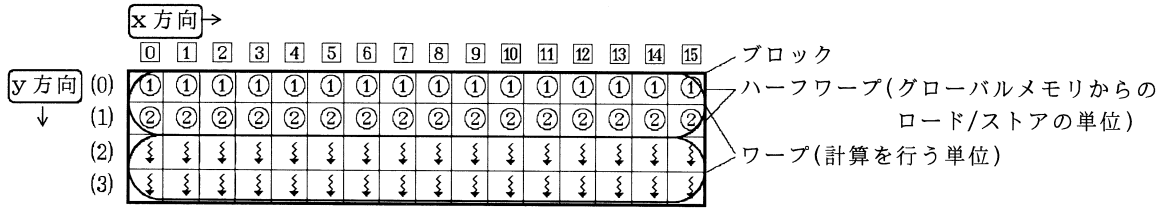


図 3-2-8

図 3-2-9 (1) のプログラムは、②で配列 dA[20] を確保します。前述のように、配列 dA は 256 バイト境界から開始します。③でブロック数 1、ブロック内のスレッド数 16 でカーネル関数を実行し、①で 16 個のスレッドが、配列 dA の自分が担当する要素をロード（またはストア）します。16 個のスレッドが、自分に担当する要素を、図 3-2-6 (3) に示すように 32 バイトのトランザクションでロードしたとすると、図 3-2-9 (2) に示すように、32 バイトのトランザクションを 16 回ロードする必要があります。これでは効率が悪いので、実際には以下のようにロードが行われます。

前述のように、グローバルメモリ上の変数／配列のロード／ストアは、ハーフワープ単位で行われます。図 3-2-9 (1) では、③で指定した 16 スレッドがハーフワープになります。ハーフワープ内の全スレッドがロードする図 3-2-9 (2) の要素を全て合体すると、図 3-2-9 (3) のようになります。これはちょうど図 3-2-6 (2) の 64 バイトトランザクションになるので、ハーフワープ内の 16 スレッドは、単精度の場合 64 バイトトランザクションを 1 回だけロードします (倍精度の場合は 128 バイトトランザクションを 1 回だけロードします)。

このように、ハーフワープに含まれる全スレッドがロード／ストアする各要素を合体してロード／ストアすることを、コアレスアクセス、あるいはコアレッシングと言います。コアレス (coalesce) とは、「合体する」という意味です。

後述するように、プログラムによって、コアレスアクセスが効率よく行われない、またはコアレスアクセスにならない場合があります。トランザクションをロード／ストアする回数が最も少なく、トランザクションの大きさが最も小さい場合に、コアレスアクセスが最も効率よく行われます (5-5 節参照)。

```
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;                          ①
}
```

```
int main(void){
  float *dA;
  cudaMalloc((void**)&dA,20*sizeof(float)); ②
    :
  kernel<<<1,16>>>(dA);                          ③
    :
```

図 3-2-9 (1)

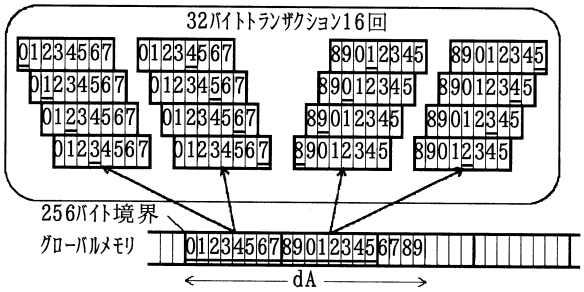


図 3-2-9 (2)

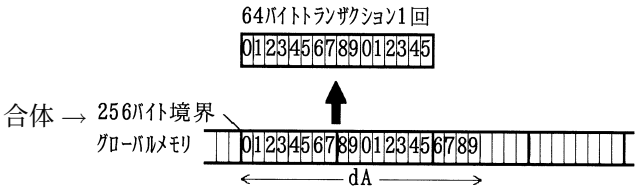


図 3-2-9 (3)

プロファイラー

グローバルメモリのロード／ストアで、32，64，128 バイトのうち、どのトランザクションが使用されたかを、プロファイラーを使用して知ることができます。詳細は 4-5 節を参照して下さい。

コアレスアクセスの注意点 (1) ストライド

ハーフワープ内の隣り合う 2 つのスレッドが担当する要素間の距離（要素数）をストライドと呼ぶことにします。図 3-2-10 の①, ②の変数 stride でストライドを設定します。stride = 1 の場合、図 3-2-11 の (1) に示すように、ハーフワープの各スレッドは、64 バイトトランザクション 1 回でロード／ストアを行います。単精度の場合、64 バイトトランザクション 1 回でのロード／ストアが、ロード／ストアの回数が最も少なく、トランザクションの大きさが最も小さいので、コアレスアクセスが最も効率よく行われた状態です。

stride = 2,3,4,16,32 の場合、前述のプロファイラーを使用して調べたところ、ロード／ストアのトランザクションは、図 3-2-11 の (2)〜(6) のようになりました。図から分かるように、ストライドが長くなると、トランザクションの量や回数が多くなり、コアレスアクセスの効率が悪くなります ((6) は全くコアレスアクセスされていません)。可能であれば、なるべくストライドが短くなるようにして下さい。

```
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x ;
  int stride = 1;                                  ①
  dA[i*stride] = dA[i*stride] + 1.0f;              ②
}
```

```
int main(void){
  float *dA;
    :
  kernel<<<1,16>>>(dA);
    :
```

図 3-2-10

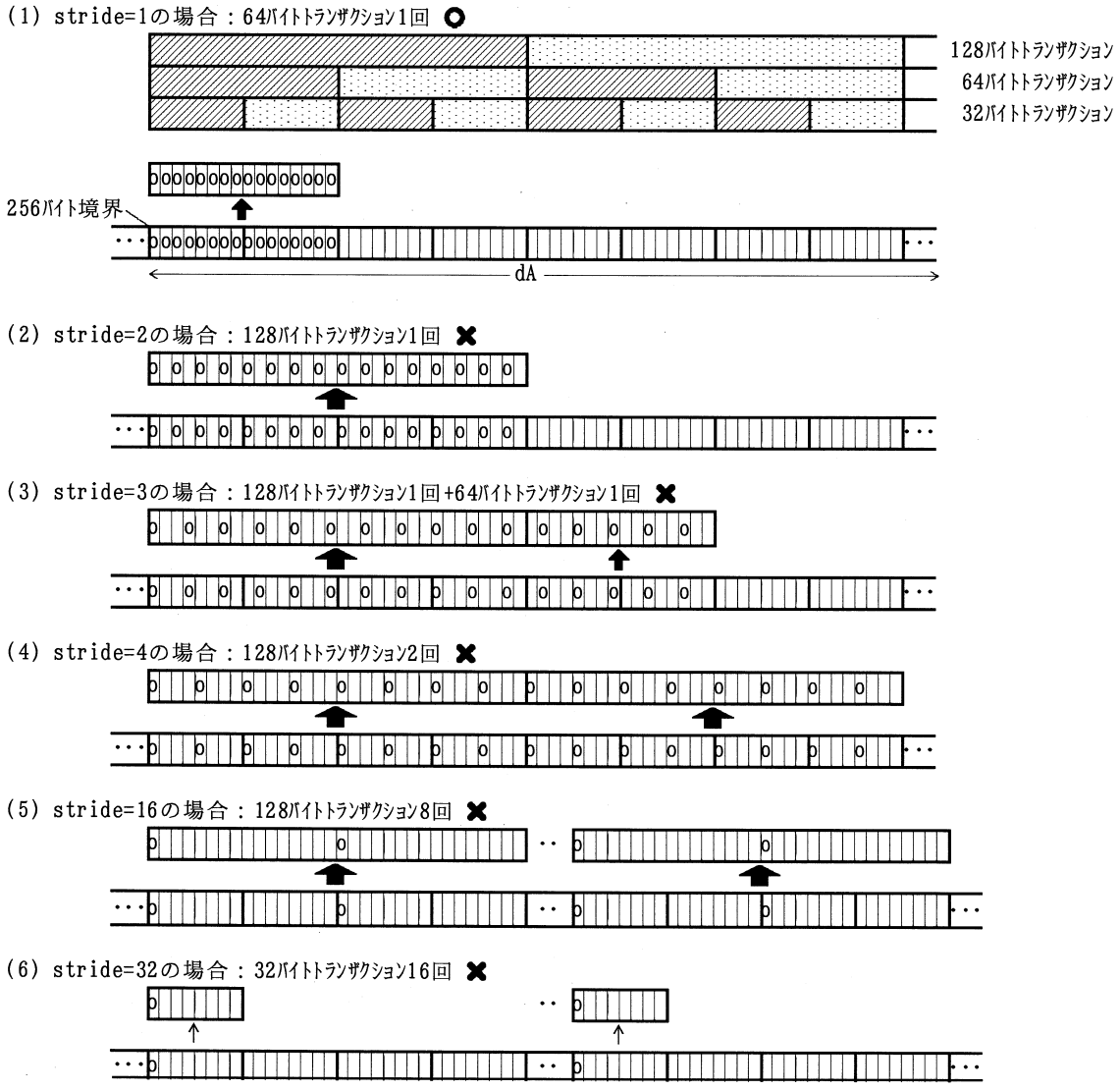


図 3-2-11

コアレスアクセスの注意点 (2) 最初のスレッドが処理する要素の位置

図 3-2-12 の①，②で start = 0 の場合、図 3-2-13 (1) に示すように、64 バイトトランザクション 1 回でロード／ストアが行われるので、前ページで説明したように、コアレスアクセスが最も効率よく行われた状態です。

start = 1, 8, 16, 24 の場合、図 3-2-13 の (2)〜(5) に示すように、最初のスレッドが処理する要素の位置が、配列 dA の先頭である 256 バイト境界から、start で指定した要素分、右にずれます。このうち (4) (64 バイト境界から開始) は (1) と同じなので最も効率がいいですが、(2) と (3) は (1) よりトランザクションの量が多くなり、(5) は (1) よりトランザクションのロード／ストアの回数が多くなり、コアレスアクセスの効率が悪くなります。

以上より、可能であれば、(1) または (4) のように、最初のスレッドが処理する要素の位置が、配列が単精度の場合少なくとも 64 バイト境界に、配列が倍精度の場合少なくとも 128 バイト境界になるようにして下さい。

```
__global__ void kernel(float *dA){
  int start  = 0;                                          ①
  int i = blockIdx.x*blockDim.x + threadIdx.x + start ;    ②
  dA[i] = dA[i] + 1.0f;
}
```

```
int main(void){
  float *dA;
    :
  kernel<<<1,16>>>(dA);
    :
```

図 3-2-12

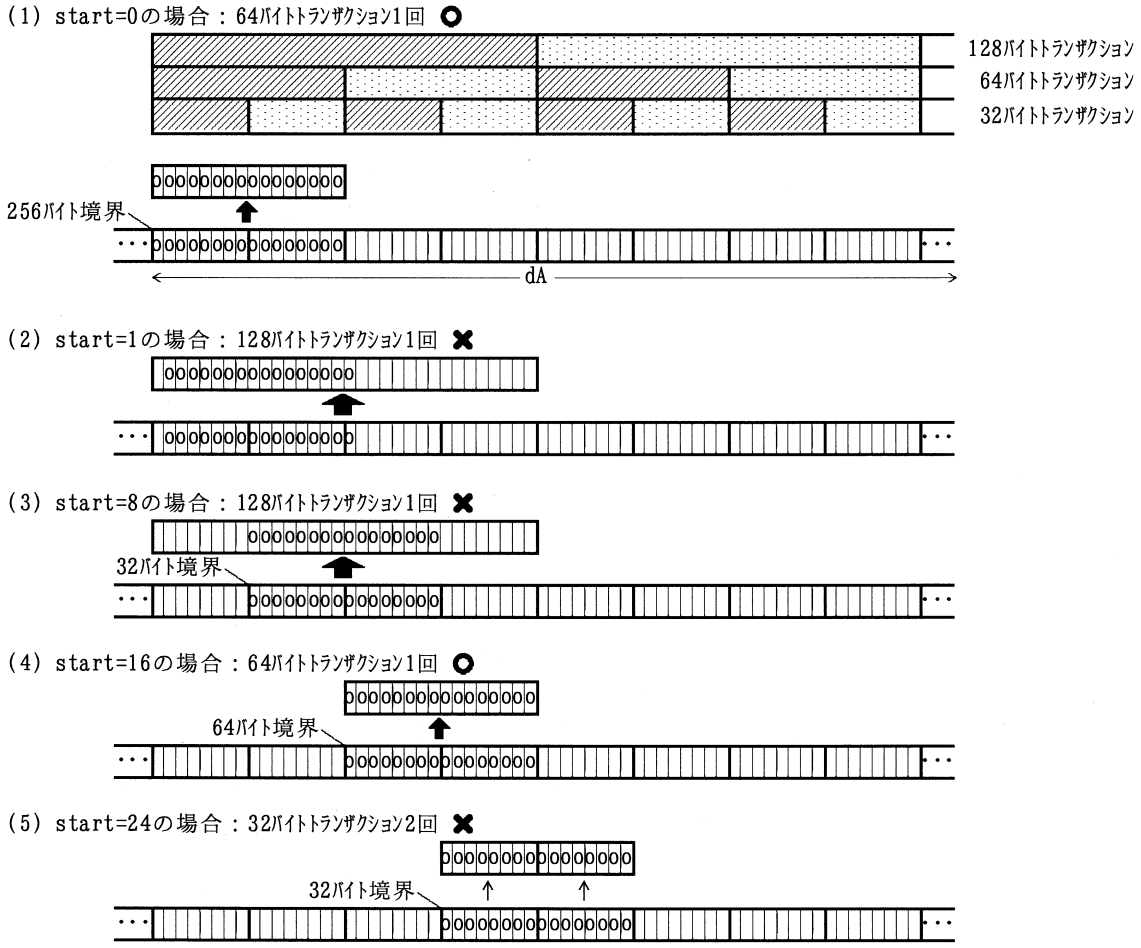


図 3-2-13

コアレスアクセスの注意点 (3) 構造体とコアレスアクセス

構造体を使用しない場合と使用した場合の、コアレスアクセスとコピーの効率を比較します。

- コアレスアクセスの効率：図 3-2-14 (1) では、③で大きさ 16 の配列 A，B，dA，dB を指定し、⑥で 1 ブロック、16 スレッドでカーネル関数を実行します。この場合、図 3-2-15 (1) に示すように、①で、配列 dA を 64 バイトトランザクション 1 回（最も効率のよいコアレスアクセス）でロード／ストアします。②についても同様です。
  一方図 3-2-14 (2) では、⑦で構造体 (KOUZOU) を定義し、⑧，⑨では、⑩で構造体 (KOUZOU) として宣言した配列 dAB のメンバー a，b を使用して計算を行います。この場合、図 3-2-15 (2) に示すように、⑧のロード／ストアは 128 バイトトランザクション 1 回で行い、そのうち「a」の部分のみを計算に使用し、「-」の部分は使用しません。⑧はデータ量が①の 2 倍なので、コアレスアクセスの効率が悪くなります。⑨についても同様です。
  従って、一般に構造体を使用すると、ロード／ストアのデータ量が増えて効率が悪くなります。なお、後述するベクトル型を使用すると、改善できる場合があります。
- コピーの効率：ホスト側からデバイス側へのコピーの効率を比較します。どちらもコピーするデータ量は同じ (本例では計 128 バイト) ですが、図 3-2-14 (1) では④，⑤の 2 回でコピーを行い、図 3-2-14 (2) では⑪の 1 回でコピーを行います。従って構造体を使用した図 3-2-14 (2) の方が、コピー回数が少ないのでコピーの速度は速くなります (5-5 節参照)。なお、構造体を使用しない図 3-2-14 (1) で、④と⑤のコピーを 1 回でまとめて行い、コピー回数を減らす方法もあります (6-2 節参照)。

```
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;                          ①
  dB[i] = dB[i] + 2.0f;                          ②
}
int main(void){
  float A[16],B[16];                             ③
  float *dA  ,*dB;                               ③
    :
  cudaMemcpy(dA,A,16*4,～HostToDevice)           ④
  cudaMemcpy(dB,B,16*4,～HostToDevice)           ⑤
  kernel<<<1,16>>>(dA,dB);                       ⑥
    :
```

図 3-2-14 (1)

```
struct KOUZOU{                                   ↑
  float a;                                       ⑦
  float b;                                       ↓
};
__global__ void kernel(KOUZOU *dAB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dAB[i].a = dAB[i].a + 1.0f;                    ⑧
  dAB[i].b = dAB[i].b + 2.0f;                    ⑨
}
int main(void){
  KOUZOU AB[16];                                 ⑩
  KOUZOU *dAB;                                   ⑩
    :                          ←4バイトのメンバーが2個なので
  cudaMemcpy(dAB,AB,16*8,～HostToDevice)         ⑪
  kernel<<<1,16>>>(dAB);
    :
```

図 3-2-14 (2)

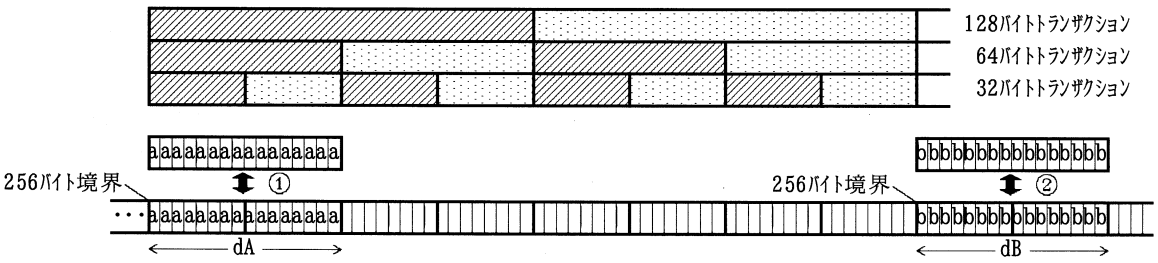


図 3-2-15 (1)

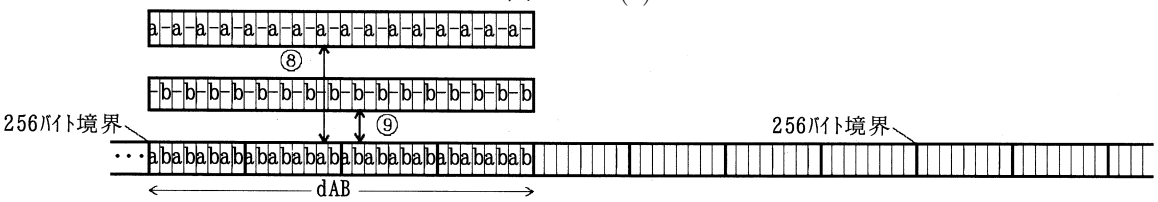


図 3-2-15 (2)

コアレスアクセスの注意点 (4) ベクトル型とコアレスアクセス

CUDA では、図 3-2-16 に示すように、同じ種類の基本的なデータ型 (整数や実数) を複数連結したデータ型 (例えば float を 4 つ連結して float4) が提供されており、これをビルトインベクトル型（以下ベクトル型）と呼びます。ベクトル型は、ホスト側とデバイス側のどちらのプログラムでも使用することができます。

図 3-2-17 (1) では、①で 4 つの単精度実数をメンバーに持つ構造体 (KOUZOU) を宣言し、②で「KOUZOU」型の変数 A を宣言しています。これと同様の指定を、ベクトル型を用いて行う方法を説明します。図 3-2-16 で、4 つの単精度実数を連結したデータ型は「float4」です。そこで図 3-2-17 (2) の③に示すように、「float4」型で変数 A を宣言します。なお、図 3-2-17 (1) では、構造体のメンバー名を任意の名前 (本例では p，q，r，s) にすることができますが、ベクトル型の場合、図 3-2-17 (2) に示すように、メンバー名は決められた名前 (2 次元の場合 x，y、3 次元の場合 x，y，z、4 次元の場合 x，y，z，w) になります。

ベクトル型で定義した変数に対して初期値を設定する場合、図 3-2-17 (3) に示すように、make__xxxx(～) という関数を使用します。xxxx の部分にベクトル型の名前を指定し、カッコ内に初期値を指定します。

- コアレスアクセスの効率：図 3-2-14 (2) と同様のプログラムを、ベクトル型を使って作成すると図 3-2-18 になります。アセンブラリスト (2-7 節参照) とプロファイラー (4-5 節参照) で調べた所、④，⑤では、図 3-2-15 (2) のようにロード／ストアを計 2 回行うのではなく、図 3-2-19 のように 1 回だけ行っているようです。回数が少ないので、(少なくとも float2 の場合は) 構造体を用いた図 3-2-15 (2) より速くなるようです。
  なお、図 3-2-16 以外の、任意のメンバーを持つ構造体の場合、__align__ という修飾子を付けて構造体を定義すると速くなる可能性がありますが、説明は省略します (CUDA マニュアル 5.3.2.1.1 節参照)。
- コピーの効率：⑦で行うコピーのデータ量と回数は、図 3-2-14 (2) の⑪と同じなので、速度も同じです。

1次元	2次元	3次元	4次元	1次元	2次元	3次元	4次元
char1	char2	char3	char4	uchar1	uchar2	uchar3	uchar4
short1	short2	short3	short4	ushort1	ushort2	ushort3	ushort4
int1	int2	int3	int4	uint1	uint2	uint3	uint4
long1	long2	long3	long4	ulong1	ulong2	ulong3	ulong4
longlong1	longlong2			ulonglong1	ulonglong2		
float1	float2	float3	float4				
double1	double2						

図 3-2-16

```
struct KOUZOU{
  float p;
  float q;      ①
  float r;
  float s;
};
```

```
   :
KOUZOU A; ②
A.p = 1.0f;
A.q = 2.0f;
A.r = 3.0f;
A.s = 4.0f;
  :
```

図 3-2-17 (1)

```
   :
float4 A; ③
A.x = 1.0f;
A.y = 2.0f;
A.z = 3.0f;
A.w = 4.0f;
   :
```

図 3-2-17 (2)

```
float4 A = make_float4(1.0f,2.0f,3.0f,4.0f);
```

図 3-2-17 (3)

```
__global__ void kernel(float2 *dAB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dAB[i].x = dAB[i].x + 1.0f;                ④
  dAB[i].y = dAB[i].y + 2.0f;                ⑤
}
int main(void){
  float2 AB[16];                             ⑥
  float2 *dAB;                               ⑥
    :                          ←4バイトのメンバーが2個なので
  cudaMemcpy(dAB,AB,16*8,～HostToDevice)     ⑦
  kernel<<<1,16>>>(dAB);
    :
```

図 3-2-18

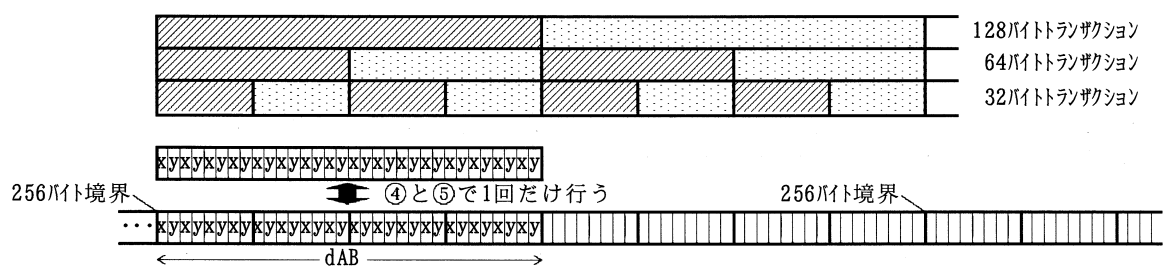


図 3-2-19

3-3　cudaMalloc と多次元配列

cudaMalloc で確保した 1 次元配列を 2 次元配列のように使用

図 3-3-2（通常のプログラム）では、⑦の malloc で確保した大きさ NX × NY の 1 次元配列 A を、⑧で関数 func に渡します。しかし、malloc で確保した 1 次元配列を、関数 func の 2 重ループ内で、③のように 2 次元配列として使用することは（通常）できません。

ところで、図 3-3-1 (1) の 2 次元配列 A[2][3] は、添字 ix と iy を図 3-3-1 (2) のように添字 i に変換すれば、1 次元配列 A[6] となります。この変換を利用すれば、④のように、1 次元配列 A のまま 2 重ループ内で使用することができます。ただし④は添字の意味が分かりにくいという欠点があります。この場合、①のマクロを使用すると、④を⑤のように 2 次元配列風に表すことができ、IND の部分を無視すれば 2 次元配列 A[iy][ix] とほぼ同じになります (⑤はコンパイル時に①によって④に変換されます)。①の IND は INDex の略（名前は任意）です。同様に、②のマクロを指定すると、④を⑥のように 2 次元配列風に表すことができます。なお、①の中では配列名を指定していないため、①を配列 A と同じ大きさの他の配列にも適用することができますが、②の中では配列名 A を指定しているため、②を配列ごとに別々に作成する必要があります。

図 3-3-2 を CUDA 化したプログラムを図 3-3-3 に示します。⑦の cudaMalloc で確保した大きさ NX × NY の 1 次元配列 dA を、⑧でカーネル関数 kernel に渡します。malloc と同様に、cudaMalloc で確保した 1 次元配列も、カーネル関数側で 2 次元配列として使用することは（通常）出来ません。そこで、④, ⑤, ⑥と同様に④, ⑤, ⑥にすれば、1 次元配列 dA を、ix と iy を使用して 2 次元配列風に表すことができます。

なお、本節の以降の説明では、⑤の 1 次元配列 dA[IND(ix,iy)] を、2 次元配列 dA[iy][ix] で表すことにします。

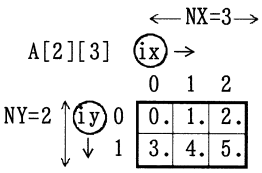


図 3-3-1 (1)

i	=	iy	*	NX	+	ix
0	=	0	*	3	+	0
1	=	0	*	3	+	1
2	=	0	*	3	+	2
3	=	1	*	3	+	0
4	=	1	*	3	+	1
5	=	1	*	3	+	2

図 3-3-1 (2)

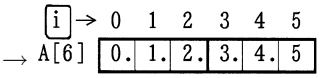


図 3-3-1 (3)

```
#define NX (3)
#define NY (2)
#define IND(iy,ix) ((iy)*NX+(ix))                    ①
#define A(iy,ix) A[(iy)*NX+(ix)]                     ②
void func(float *A){
  for(int iy=0;iy<NY;iy++){
    for(int ix=0;ix<NX;ix++){
     (A[iy][ix]     = 1.0f;) ✖間違い                 ③
      A[iy*NX+ix]   = 1.0f;                          ④
      A[IND(iy,ix)] = 1.0f;                          ⑤
      A(iy,ix)      = 1.0f;                          ⑥
    }        ↖ A[iy][ix]風に表現
  }
}
int main(void){
  float *A;
  size_t size = NX*NY*sizeof(float);
  A = (float*)malloc(size);                          ⑦
  func(A);                                           ⑧
  free(A);
    :
```

図 3-3-2

```
#define NX (3)
#define NY (2)
#define IND(iy,ix) ((iy)*NX+(ix))                    1
#define dA(iy,ix) dA[(iy)*NX+(ix)]                   2
__global__ void kernel(float *dA){
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
 (dA[iy][ix]     = 1.0f;) ✖間違い                    3
  dA[iy*NX+ix]   = 1.0f;                             4
  dA[IND(iy,ix)] = 1.0f;                             5
  dA(iy,ix)      = 1.0f;                             6
}       ↖ dA[iy][ix]風に表現
int main(void){
  float *dA;
  size_t size = NX*NY*sizeof(float);
  cudaMalloc((void**)&dA,size);                      7
  kernel<<<1,dim3(NX,NY)>>>(dA);                     8
  cudaFree(dA);
    :
```

図 3-3-3

2 次元配列の場合のコアレスアクセスの注意点 (1)　配列の添字とスレッド ID の対応

カーネル関数で 2 次元配列を扱う場合の、コアレスアクセスに関する注意点を説明します。図 3-3-4 (1) (2) の③では、図 3-3-5 (1) の 2 次元配列 dA[NY][NX] (実際は図 3-3-6 (1) (2) の 1 次元配列 dA) をロード／ストアします。④でブロック数 1、ブロック内のスレッド数 16 × 16 でカーネル関数を実行した場合、各スレッドは図 3-3-5 (1) (2) の○で囲んだ部分を担当します。ブロック内の 1 つ目と 2 つ目のハーフワープ (16 スレッド) が処理する要素を図の「1...1」と「2...2」に示します。②では、添字 ix を x 方向のスレッド ID (threadIdx.x) に対応付けているため、1 つ目のハーフワープは図 3-3-5 (1) の「1...1」（グローバルメモリ上で連続）を担当し、図 3-3-6 (1) に示すように、最も効率のよい 64 バイトトランザクション 1 回でロード／ストアします。

一方図 3-3-4 (2) の⑤では、添字 iy を x 方向のスレッド ID に対応付けているため、1 つ目のハーフワープは図 3-3-5 (2) の「1...1」（グローバルメモリ上で飛び飛び）を担当し、図 3-3-6 (2) に示すように 32 バイトトランザクション 16 回でロード／ストアするので、コアレスアクセスにならず効率が悪くなります。

以上のことから、カーネル関数内で 2 次元配列を扱う場合、図 3-3-4 (1) の②のように対応付けて下さい。

```
#define NX (32)
#define NY (32)
#define IND(iy,ix) ((iy)*NX+(ix))                    ①
__global__ void kernel(float *dA){
int ix = blockIdx.x*blockDim.x + threadIdx.x;②
int iy = blockIdx.y*blockDim.y + threadIdx.y;②
dA[IND(iy,ix)] = dA[IND(iy,ix)] + 1.0f;      ③
}
   :
kernel<<<1,dim3(16,16)>>>(dA);                       ④
   :
```

図 3-3-4 (1)　○

```
#define NX (32)
#define NY (32)
#define IND(iy,ix) ((iy)*NX+(ix))
__global__ void kernel(float *dA){
int ix = blockIdx.y*blockDim.y + threadIdx.y;⑤
int iy = blockIdx.x*blockDim.x + threadIdx.x;⑤
dA[IND(iy,ix)] = dA[IND(iy,ix)] + 1.0f;      ③
}
   :
kernel<<<1,dim3(16,16)>>>(dA);                       ④
   :
```

図 3-3-4 (2)　×

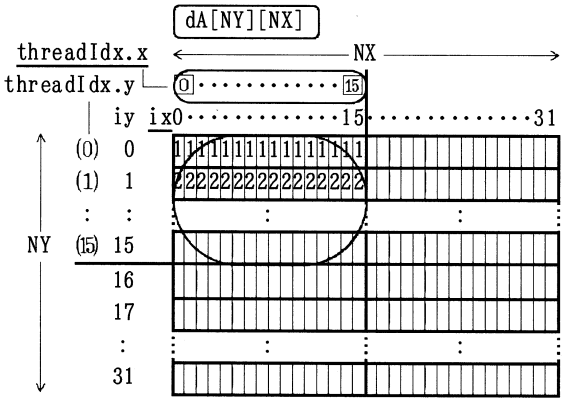


図 3-3-5 (1)

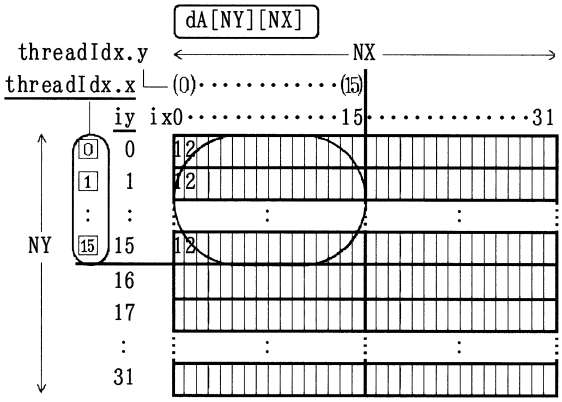


図 3-3-5 (2)

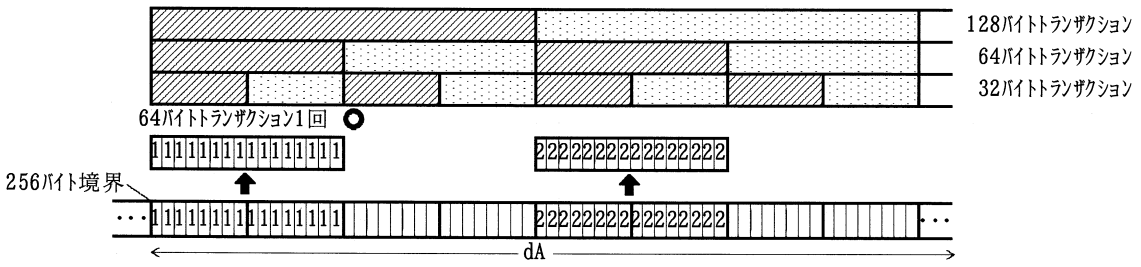


図 3-3-6 (1)

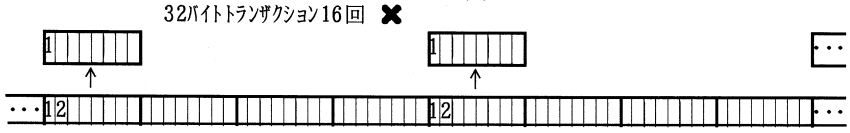


図 3-3-6 (2)

2 次元配列の場合のコアレスアクセスの注意点 (2)　x 方向のスレッド数

図 3-3-4 (1) の④の x 方向のスレッド数を、図 3-3-7 (1) の⑦のように例えば 20 にした場合、ブロック内の 2 つ目のハーフワープが担当する要素「2...2」は、2 次元配列 dA 上では図 3-3-7 (2) に示すように 2 行になり、1 次元配列 dA 上では図 3-3-7 (3) に示すように 2 つに分断されます。このため図 3-3-7 (1) の⑥で、2 つ目のハーフワープは、図 3-3-7 (3) の↑と↑に示すように、32 バイトのトランザクション 1 回と 64 バイトのトランザクション 1 回をロード／ストアし、効率が悪くなります。

以上のことから、カーネル関数内で 2 次元配列を扱う場合、ブロック内の x, y 方向のスレッドのうち、x 方向のスレッド数は 16（ハーフワープ内のスレッド数）の倍数になるようにして下さい。具体的には、図 3-3-7 (1) の⑦の下線部を 16 の倍数に設定して下さい (2-5 節参照)。これに加え、前述のように、ブロック内の全スレッド数は 32 の倍数が望ましく、また上限は 512 という制限があります。

```
#define NX (32)
#define NY (32)
#define IND(iy,ix) ((iy)*NX+(ix))
__global__ void kernel(float *dA){
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  dA[IND(iy,ix)] = dA[IND(iy,ix)] + 1.0f;    ⑥
}
   :
  kernel<<<1,dim3(20,16)>>>(dA);              ⑦
   :
```

図 3-3-7 (1)

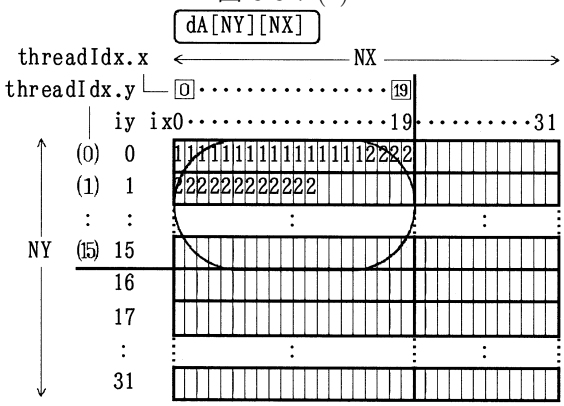


図 3-3-7 (2)

図 3-3-7 (3)

2 次元配列の場合のコアレスアクセスの注意点 (3)　配列の 2 次元目の大きさ

図 3-3-4 (1) の NX が図 3-3-8 (1) の⑧のように 30 の場合、2 次元配列 dA[32][30] は図 3-3-9 (1) となります。このとき 1 次元配列 dA は図 3-3-10 (1) となり、ブロック内の 2 つ目のハーフワープが担当する要素「2...2」の左の 2 要素が、64 バイトトランザクションからはみ出します。このため図 3-3-8 (1) の⑨で、2 つ目のハーフワープは、図 3-3-10 (1) の↑と↑に示すように、32 バイトのトランザクション 1 回と 64 バイトのトランザクション 1 回をロード／ストアし、効率が悪くなります。

この場合、図 3-3-8 (2) の⑪のように、配列 dA の 2 次元目を大きくして dA[32][30+2] にします。図の「P」を付けた部分が追加した要素で、計算には使いません。これを「パディング」すると言います (padding: 詰め物をすること)。これによって、2 つ目のハーフワープは、図 3-3-10 (2) の「2...2」に示すように、最も効率のよい、64 バイトのトランザクション 1 回でロード／ストアします。

以上のことから、カーネル関数内で 2 次元配列を扱う場合、2 次元目 (dA[32][30] の 30) の大きさが 16 要素の倍数になるようにして下さい。なお、⑪の修正に伴い⑩の下線部も修正します。

```
#define NX (30)                                  ⑧
#define NY (32)
#define IND(iy,ix) ((iy)*NX+(ix))
__global__ void kernel(float *dA){
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  dA[IND(iy,ix)] = dA[IND(iy,ix)] + 1.0f;    ⑨
}
  1次元配列dAを大きさNX×NYで確保
  kernel<<<1,dim3(16,16)>>>(dA);
    :
```

図 3-3-8 (1)

```
#define NX (30)
#define NY (32)
#define IND(iy,ix) ((iy)*(NX+2)+(ix))            ⑩
__global__ void kernel(float *dA){
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  dA[IND(iy,ix)] = dA[IND(iy,ix)] + 1.0f;
}
  1次元配列dAを大きさ(NX+2)×NYで確保          ⑪
  kernel<<<1,dim3(16,16)>>>(dA);
    :
```

図 3-3-8 (2)

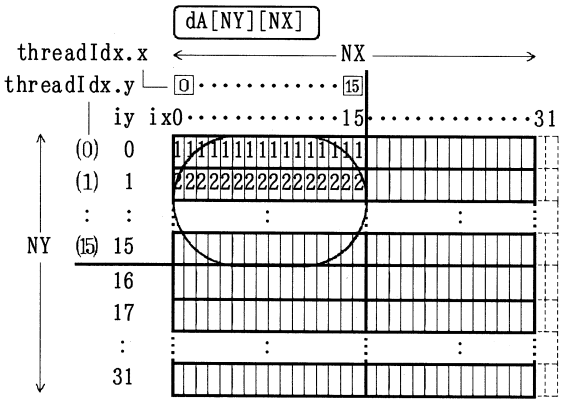


図 3-3-9 (1)

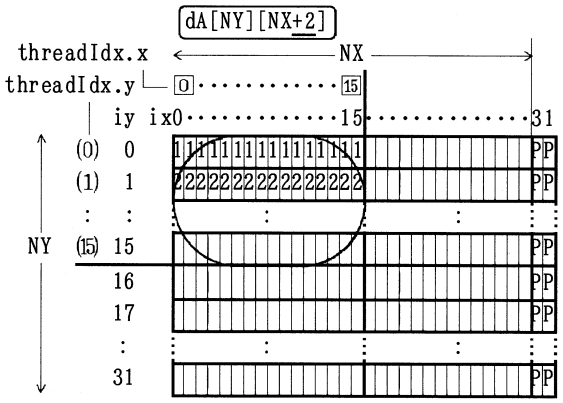


図 3-3-9 (2)

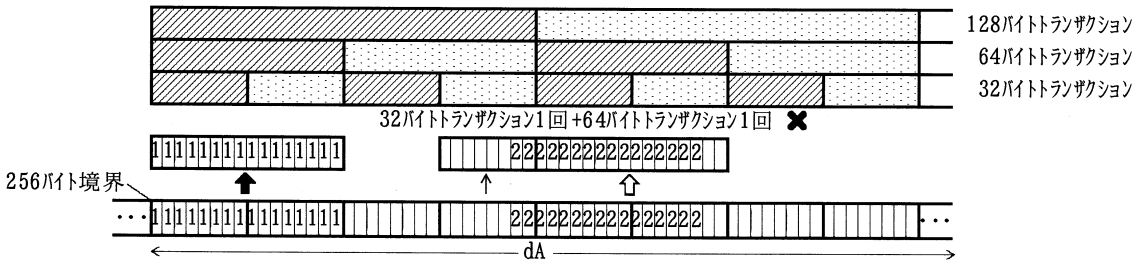


図 3-3-10 (1)

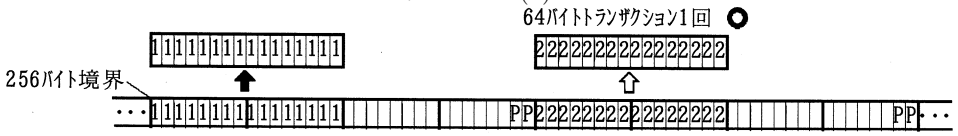


図 3-3-10 (2)

コアレスアクセスを考慮した 2 次元配列のプログラム例

ホスト側の配列 A が図 3-3-11 (1) に示すように A[16][34] で、そのうち実際に計算するのは A[16][30] までの要素（着色した部分）だとします (cudaMemcopy2D の使い方を説明するためにわざと 34 にしていますが、値自体に意味はありません)。このときデバイス側の配列 dA の大きさを、図 3-3-11 (2) のように dA[16][32] にすると、2 次元目の要素数が 16 の倍数になるので、前述のように最も効率よくコアレスアクセスが行われます (図の「P」を付けた部分は計算には使いません)。配列 dA の 2 次元目の大きさを、自動的に 16 の倍数にするプログラムを図 3-3-12 で説明します。

- ④でホスト側の配列 A を確保します。2 次元目が、実際に計算する要素数 (NX = 30) より大きくなっています。
- ⑤でデバイス側の配列 dA を宣言します。
- ⑥で、配列 dA の 2 次元目の大きさ LDdA を求めます (図 3-3-11 (2) 参照)。配列 dA の 2 次元目の実際に計算する部分の要素数 (NX) 以上で、16 要素の倍数の最も小さい値を、変数 LDdA とします (LDdA は Leading Dimension of array dA の略です)。
- ⑦，⑧で、デバイス側の配列 dA をグローバルメモリに確保します。
- ⑨で、ホスト側の配列 A のうち、実際に計算する部分 (図 3-3-11 (1) の着色した部分) を、デバイス側の配列 dA にコピーします。図から分かるように、配列 A，dA はともに、着色した部分がメモリ上で不連続になっています。この場合、CUDA 関数 cudaMemcpy の代わりに cudaMemcpy2D を使用します。⑨の実線がホスト側の配列名 (A) と 2 次元目の大きさ（バイト）、二重線がデバイス側の配列名 (dA) と 2 次元目の大きさ（バイト）、波線が実際にコピーする部分の 2 次元目の大きさ（バイト）、NY はコピーする 1 次元目の要素数です。
- ⑩で x 方向のブロック数を 2、ブロック内のスレッド数を 16 × 16 でカーネル関数を実行します。このとき、配列 dA の 2 次元目の大きさ LDdA をカーネル関数に渡します。
- カーネル関数は②で LDdA を受取り、①のマクロで LDdA を使用し、マクロを使用して③で計算を行います。
- ⑪で、デバイス側の配列 dA から、ホスト側の配列 A にコピーします。このときも、着色した部分がメモリ上で不連続なので、cudaMemcpy の代わりに cudaMemcpy2D を使用します。

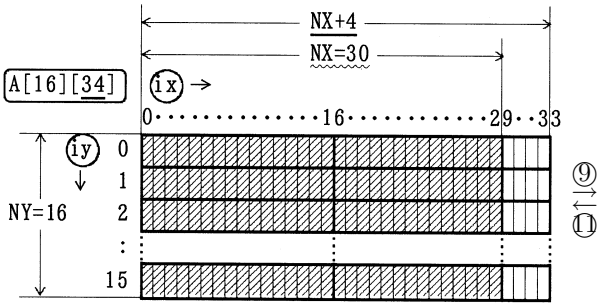


図 3-3-11 (1)

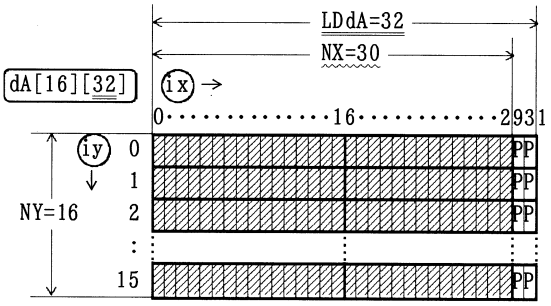


図 3-3-11 (2)

```
#define NX (30)
#define NY (16)
#define IND(iy,ix) ((iy)*LDdA+(ix))                    ①
__global__ void kernel(float *dA,int LDdA){            ②
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  if(ix<NX && iy<NY){
    dA[IND(iy,ix)] = dA[IND(iy,ix)] + 1.0f;            ③
  }
}
```

```
int main(void){
  float A[NY][NX+4];                                   ④
  float *dA;                                           ⑤
  int LDdA = ((NX+15)/16)*16;                          ⑥
  size_t size = LDdA*NY*sizeof(float);                 ⑦
  cudaMalloc((void**)&dA,size);                        ⑧
  配列Aに値を設定します。
  cudaMemcpy2D(dA,LDdA*sizeof(float),
    A,(NX+4)*sizeof(float),NX*sizeof(float),           ⑨
    NY,cudaMemcpyHostToDevice);
  kernel<<<2,dim3(16,16)>>>(dA,LDdA);                  ⑩
  cudaMemcpy2D(A,(NX+4)*sizeof(float),
    dA,LDdA*sizeof(float),NX*sizeof(float),            ⑪
    NY,cudaMemcpyDeviceToHost);
    :
```

図 3-3-12

コアレスアクセスを考慮した 2 次元配列のプログラム例 (cudaMallocPitch を使用)

図 3-3-12 のプログラムでは、⑥で、デバイス側の配列 dA の 2 次元目の大きさが 16 要素の倍数になるように計算しましたが、これを自動的に行う CUDA 関数 cudaMallocPitch が提供されています。ただし、配列 dA の 2 次元目の大きさは、16 要素の倍数にはならず、単精度で 64 要素、倍精度で 32 要素の倍数になるようです。従って図 3-3-11 (2) は、3-3-13 に示すように dA[16][64] となります。

cudaMallocPitch を使用したプログラムを図 3-3-14 に示します。図 3-3-12 の⑥～⑧を⑫～⑭に置き換えた以外は同じなので、相違点のみを説明します。

図 3-3-14 の⑫で変数 pitch (図 3-3-13 参照) を宣言します。⑬の 2 つ目の引数に変数 pitch を指定し、3 つ目の引数に、実際に使用する部分の要素数 (30 × 4)（単位はバイト）を指定して⑬を実行すると、デバイス側の配列 dA[16][64] がグローバルメモリに確保され、2 次元目の要素数 (64 × 4)（単位はバイト）が引数 pitch に戻ります。⑮の引数で LDdA（単位は要素数）をカーネル関数に渡す場合は、⑭でバイトから要素数に変換します。

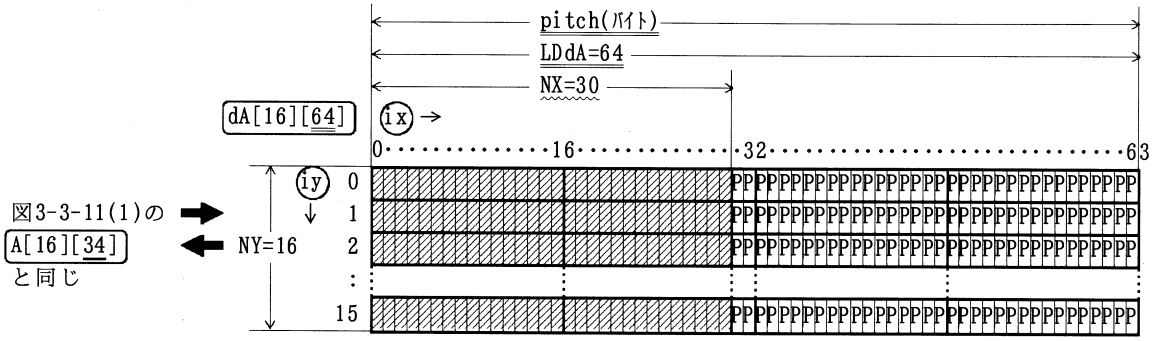


図 3-3-13

```
#define NX (30)
#define NY (16)
#define IND(iy,ix) ((iy)*LDdA+(ix))
__global__ void kernel(float *dA,int LDdA){
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  if(ix<NX && iy<NY){
    dA[IND(iy,ix)] = dA[IND(iy,ix)] + 1.0f;
  }
}
```

```
int main(void){
  float A[NY][NX+4];
  float *dA;
  size_t pitch;                                    ⑫
  cudaMallocPitch((void**)&dA,&pitch,              ⑬
    NX*sizeof(float),NY);                          ⑬
  int LDdA = pitch/sizeof(float);                  ⑭
  配列Aに値を設定します。
  cudaMemcpy2D(dA,LDdA*sizeof(float),
    A,(NX+4)*sizeof(float),NX*sizeof(float),
    NY,cudaMemcpyHostToDevice);
  kernel<<<2,dim3(16,16)>>>(dA,LDdA);              ⑮
  cudaMemcpy2D(A,(NX+4)*sizeof(float),
    dA,LDdA*sizeof(float),NX*sizeof(float),
    NY,cudaMemcpyDeviceToHost);
    :
```

図 3-3-14

cudaMalloc で確保した 1 次元配列を 3 次元配列のように使用

cudaMalloc 関数で確保した 1 次元配列を、図 3-3-16 の③のように、関数 func 内で 3 次元配列として使用することは（通常）できません。3 次元配列風に使用する方法は、前述の 2 次元の場合と同じです。図 3-3-2（通常のプログラム）の 3 次元版を図 3-3-16 に、図 3-3-3 (CUDA 化したプログラム) の 3 次元版を図 3-3-17 に示します。

図 3-3-15 (1) の 3 次元配列 A[2][3][4] は、添字 ix, iy, iz を図 3-3-15 (2) のように添字 i に変換すれば、1 次元配列 A[2*3*4] となります。図 3-3-15 (2) を使用すれば、図 3-3-16 の④に示すように、1 次元配列を 3 重ループ内で使用することができます。④の添字の意味が分かりにくい場合は、①または②のマクロを使用して、⑤または⑥のように 3 次元配列風に表すことができます。図 3-3-16 を CUDA 化すると図 3-3-17 となります。

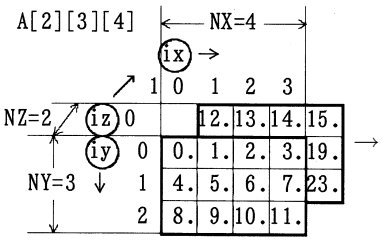


図 3-3-15 (1)

i = iz * NX*NY + iy * NX + ix													i = iz * NX*NY + iy * NX + ix											
0 =	0	*	4*3	+	0	*	4	+	0				12 =	1	*	4*3	+	0	*	4	+	0		
1 =	0	*	4*3	+	0	*	4	+	1				13 =	1	*	4*3	+	0	*	4	+	1		
2 =	0	*	4*3	+	0	*	4	+	2				14 =	1	*	4*3	+	0	*	4	+	2		
3 =	0	*	4*3	+	0	*	4	+	3				15 =	1	*	4*3	+	0	*	4	+	3		
4 =	0	*	4*3	+	1	*	4	+	0				16 =	1	*	4*3	+	1	*	4	+	0		
5 =	0	*	4*3	+	1	*	4	+	1				17 =	1	*	4*3	+	1	*	4	+	1		
6 =	0	*	4*3	+	1	*	4	+	2				18 =	1	*	4*3	+	1	*	4	+	2		
7 =	0	*	4*3	+	1	*	4	+	3				19 =	1	*	4*3	+	1	*	4	+	3		
8 =	0	*	4*3	+	2	*	4	+	0				20 =	1	*	4*3	+	2	*	4	+	0		
9 =	0	*	4*3	+	2	*	4	+	1				21 =	1	*	4*3	+	2	*	4	+	1		
10 =	0	*	4*3	+	2	*	4	+	2				22 =	1	*	4*3	+	2	*	4	+	2		
11 =	0	*	4*3	+	2	*	4	+	3				23 =	1	*	4*3	+	2	*	4	+	3		

図 3-3-15 (2)

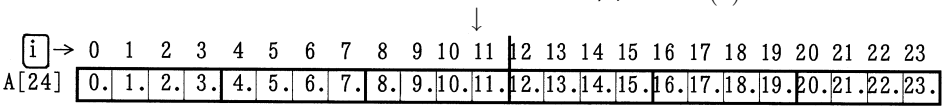


図 3-3-15 (3)

```
#define NX (4)
#define NY (3)
#define NZ (2)                                    ①
#define IND(iz,iy,ix) ((iz)*NX*NY+(iy)*NX+(ix))
#define A(iz,iy,ix) A[(iz)*NX*NY+(iy)*NX+(ix)]
void func(float *A){                              ②
  for(int iz=0;iz<NZ;iz++){
    for(int iy=0;iy<NY;iy++){
      for(int ix=0;ix<NX;ix++){
       (A[iz][iy][ix]    = 1.0f;) ✖ 間違い        ③
        A[iz*NX*NY+iy*NX+ix] = 1.0f;              ④
        A[IND(iz,iy,ix)] = 1.0f;                  ⑤
        A(iz,iy,ix)      = 1.0f;                  ⑥
      }      ← A[iz][iy][ix]風に表現
    }
  }
}
int main(void){
  float *A;
  size_t size = NX*NY*NZ*sizeof(float);
  A = (float*)malloc(size);                       ⑦
  func(A);                                        ⑧
  free(A);
    :
```

図 3-3-16

```
#define NX (4)
#define NY (3)
#define NZ (2)                                    1
#define IND(iz,iy,ix) ((iz)*NX*NY+(iy)*NX+(ix))
#define
   dA(iz,iy,ix) dA[(iz)*NX*NY+(iy)*NX+(ix)]       2
__global__ void kernel(float *dA){
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  int iz = blockIdx.z*blockDim.z + threadIdx.z;
 (dA[iz][iy][ix]    = 1.0f;) ✖ 間違い             3
  dA[iz*NX*NY+iy*NX+ix] = 1.0f;                   4
  dA[IND(iz,iy,ix)] = 1.0f;                       5
  dA(iz,iy,ix)      = 1.0f;                       6
}      ← dA[iz][iy][ix]風に表現
int main(void){
  float *dA;
  size_t size = NX*NY*NZ*sizeof(float);
  cudaMalloc((void**)&dA,size);                   7
  kernel<<<1,dim3(NX,NY,NZ)>>>(dA);               8
  cudaFree(dA);
    :
```

図 3-3-17

コアレスアクセスを考慮した 3 次元配列のプログラム例

前述の図 3-3-12 のプログラムでは、図 3-3-11 (1) に示すホスト側の 2 次元配列 A を、図 3-3-11 (2) に示すデバイス側の 2 次元配列 dA にコピーしました。このとき、ロード／ストアを効率化するため、配列 dA の 2 次元目の大きさが 16 要素の倍数になるようにしました。配列 A と dA が 3 次元の場合も、同じ方法で、配列 dA の 3 次元目（一番右側の添字）の大きさを 16 要素の倍数になるようにすることができます。

図 3-3-12 の 3 次元版のプログラムを図 3-3-19 に、ホスト側の配列 A を図 3-3-18 (1) に、デバイス側の配列 dA を図 3-3-18 (2) に示します。図 3-3-19 のホスト側（右側）のプログラムで、図 3-3-12 と異なる部分を下線部に示します。2 次元が単に 3 次元になっただけで、動作はほとんど同じなので、以下では要点のみ説明します。

図 3-3-19 の⑨, ⑪では、2 次元配列用の cudaMemcpy2D を用いてコピーを行います。つまり、3 次元配列 A[2][16][34]、dA[2][16][32] を、それぞれ 2 次元配列 A[2*16][34]、dA[2*16][32] とみなしてコピーを行います。cudaMemcpy2D の 3 次元版の CUDA 関数 cudaMemcpy3D も提供されていますが、cudaMemcpy2D の方が使いやすいようなので、cudaMemcpy2D を使用しました。

なお、CUDA 関数 cudaMallocPitch を使用した図 3-3-14 のプログラムも、下記と同様に 3 次元化できますが、説明は省略します。

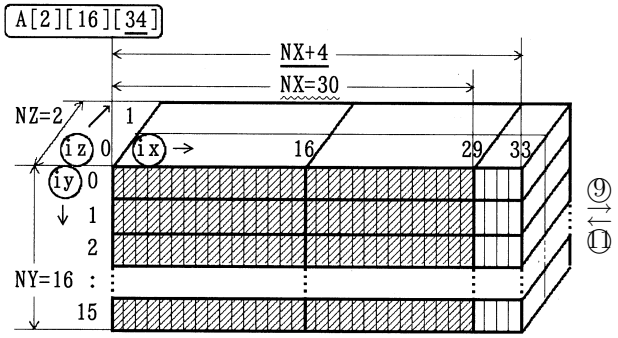


図 3-3-18 (1)

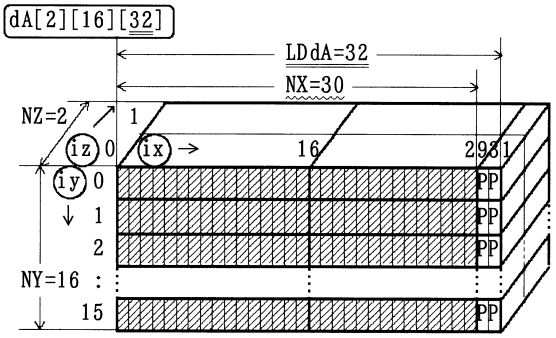


図 3-3-18 (2)

```
#define NX (30)
#define NY (16)
#define NZ (2)
#define
  IND(iz,iy,ix) ((iz)*LDdA*NY+(iy)*LDdA+(ix))①
__global__ void kernel(float *dA,int LDdA){ ②
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  int iz = blockIdx.z*blockDim.z + threadIdx.z;
  if(ix<NX && iy<NY && iz<NZ){
    dA[IND(iz,iy,ix)]                    ③
      = dA[IND(iz,iy,ix)] + 1.0f;        ③
  }
}
```

```
int main(void){
  float A[NZ][NY][NX+4];                 ④
  float *dA;                             ⑤
  int LDdA = ((NX+15)/16)*16;            ⑥
  size_t size = LDdA*NY*NZ*sizeof(float); ⑦
  cudaMalloc((void**)&dA,size);          ⑧
  配列Aに値を設定します。
  cudaMemcpy2D(dA,LDdA*sizeof(float),
    A,(NX+4)*sizeof(float),NX*sizeof(float),⑨
    NY*NZ,cudaMemcpyHostToDevice);
  kernel<<<2,dim3(16,16,2)>>>(dA,LDdA);  ⑩
  cudaMemcpy2D(A,(NX+4)*sizeof(float),
    dA,LDdA*sizeof(float),NX*sizeof(float), ⑪
    NY*NZ,cudaMemcpyDeviceToHost);
    :
```

図 3-3-19

3-4　グローバルメモリ (__device__ 修飾子で確保)

本節では、「__device__」変数型修飾子を使用して、デバイスメモリ上のグローバルメモリに変数／配列を確保する方法について説明します。なお、2-1 節で説明した「__device__」関数型修飾子とは異なります。__device__ 修飾子を指定した変数／配列の特性を以下に示します (3-1 節参照)。

- 作成される場所：デバイスメモリ内のグローバルメモリ（オフチップ：低速）上に作成されます。
- アクセスできるスレッドの範囲：全ブロックの全スレッドからアクセスすることができます。
- 存在する期間：プログラムの開始から終了までの間、存在します。
- 容量：約 4 ギガバイトです (ただし cudaMalloc 関数で確保したメモリとの合計です)。

指定方法

図 3-4-1 (2) のように、デバイスメモリ内のグローバルメモリ上に大きさ 2 の配列 dD を確保し、ホスト側の大きさ 2 の配列 D との間でコピーを行うプログラムを図 3-4-1 (1) に示します。

- ①に示すように、__device__ 修飾子を付けて宣言した配列 dD は、デバイスメモリ上のグローバルメモリに確保されます。①はプログラム内のグローバル領域に記述します。CUDA 関数の cudaMalloc (3-2 節参照) で確保する場合と違い、コンパイル／リンク時に配列の大きさが確定している必要があります。
- __device__ float dD[2][3]; のように、多次元配列も指定することができます。
- ホスト側の配列 D を②で宣言します。
- ホスト側の配列 D から、cudaMalloc で確保した配列 dD へのコピーは、cudaMemcpy で行いましたが、__device__ 修飾子で確保した配列 dD へのコピーは、③に示すように cudaMemcpyToSymbol で行います。4 つ目の引数は、配列 dD の先頭から、コピーする場所の先頭までの変位をバイトで指定します（詳細は後述します)。
- ④でカーネル関数を実行し、⑤で配列 dD から配列 D に cudaMemcpyFromSymbol でコピーします。③, ⑤では、2 つ目の引数から 1 つ目の引数にコピーします。⑤の 4 つ目の引数は、配列 dD の先頭から、コピーするデータの入った場所の先頭までの変位をバイトで指定します。
- 配列の代わりにスカラー変数をデバイス上に確保してコピーする場合、図 3-4-2 (2) のようになります。
- ③, ⑤, 3, 5の引数 dD は、文字列「"dD"」で指定することもできます (「CUDA Reference Manual」参照)。

```
__device__ float dD[2];                                    ①
__global__ void kernel(){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dD[i] = dD[i] + 1.0f;
}
int main(void){
  float D[2];                                              ②
  配列Dに値を設定する。
  size_t size = 2*sizeof(float);
  cudaMemcpyToSymbol(dD,D,size,0,                          ③
                     cudaMemcpyHostToDevice);③
  kernel<<<1,2>>>();                                       ④
  cudaMemcpyFromSymbol(D,dD,size,0,                        ⑤
                       cudaMemcpyDeviceToHost);⑤
    :
```

図 3-4-1 (1)　配列の場合

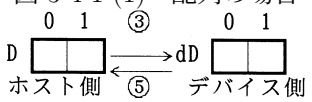


図 3-4-1 (2)

```
__device__ float dD;
__global__ void kernel(){
  dD = dD + 1.0f;
}
int main(void){
  float D;
  スカラー変数Dに値を設定する。
  size_t size = sizeof(float);
  cudaMemcpyToSymbol                                       3
     (dD,&D,size,0,cudaMemcpyHostToDevice);                3
  kernel<<<1,1>>>();
  cudaMemcpyFromSymbol                                     5
     (&D,dD,size,0,cudaMemcpyDeviceToHost);                5
    :
```

図 3-4-2 (1)　スカラー変数の場合

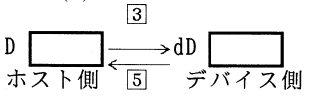


図 3-4-2 (2)

cudaMemcpyTo(From)Symbol の 4 つ目の引数

cudaMemcpyTo(From)Symbol の 4 つ目の引数について補足します。図 3-4-3 (1) の①の下線部は、配列 dD の先頭から、コピーする場所の先頭までの変位が 4 バイト (単精度実数では 1 要素) であることを意味します。同様に②の下線部は、配列 dD の先頭から、コピーするデータの入った場所の先頭までの変位が 4 バイトであることを意味します。①と②でそれぞれ 2 要素をコピーした場合、図 3-4-3 (2) のようにコピーされます。

```
__device__ float dD[3];
int main(void){
  float D[3],E[3];
    :
  size_t size = 2*sizeof(float);
  cudaMemcpyToSymbol                                  ①
     (dD,D,size,4,cudaMemcpyHostToDevice);            ①
    :
  cudaMemcpyFromSymbol                                ②
     (E,dD,size,4,cudaMemcpyDeviceToHost);            ②
    :
```

図 3-4-3 (1)

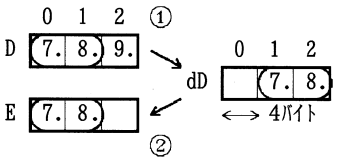


図 3-4-3 (2)

初期値の設定

__device__ 修飾子でグローバルメモリ上に確保した変数／配列は、通常の C 言語で宣言した変数／配列と同様に、初期値を設定することができます。図 3-4-4 (1) のように設定すると、図 3-4-4 (2) のように初期化されます。③では全ての要素が 0 になります。④では 1 つ目の要素が 1 で他の要素が 0 になります。

```
__device__ int dK[2] = {0};   ③
__device__ int dL[2] = {1};   ④
__device__ int dM[2] = {2,3}; ⑤
__device__ int dN    =  4;    ⑥
```

図 3-4-4 (1)

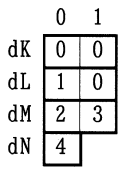


図 3-4-4 (2)

__device__ 修飾子と 2 次元配列

次に、__device__ 修飾子を指定した 2 次元配列について説明します。

通常の (CUDA 化していない) プログラムでは、図 3-4-5 (1) に示すように、②で確保した 2 次元配列を、③と①の引数で指定し、関数 func 側で使用することができます。しかし CUDA では、デバイス側の配列は、cudaMalloc で確保するか、グローバル領域で __device__ 修飾子を使用して宣言する必要があるので、図 3-4-5 (2) の④のように、ホスト側の関数内でデバイス側の配列を宣言することはできません。

```
#define NX (3)
#define NY (2)
void func(float D[NY][NX]){                    ①
  int ix,iy;
  for(iy=0;iy<NY;iy++){
    for(ix=0;ix<NX;ix++){
      D[iy][ix] = 1.0f;
    }
  }
}
int main(void){
  float D[NY][NX];                             ②
  func(D);                                     ③
    :
```

図 3-4-5 (1)

```
#define NX (3)
#define NY (2)
__global__ void kernel(float dD[NY][NX]){
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  if (ix<NX && iy<NY)
    dD[iy][ix] = 1.0f;
}
int main(void){
  float dD[NY][NX];                            ④
  kernel<<<1,dim3(NX,NY)>>>(dD);
    :
```

図 3-4-5 (2)　×　この方法は不可

一方、通常の (CUDA 化していない) プログラムでは、図 3-4-6 (1) の⑤のように、2 次元配列をグローバル領域で指定し、各関数で使用することができます。同様に、CUDA では、図 3-4-6 (2) の⑥のように __device__ 修飾子で宣言すれば、カーネル関数で 2 次元配列を使用することができます。

```
#define NX (3)
#define NY (2)
  float D[NY][NX];                             ⑤
void func(){
  int ix,iy;
  for(iy=0;iy<NY;iy++){
    for(ix=0;ix<NX;ix++){
      D[iy][ix] = 1.0f;
    }
  }
}
int main(void){
  func();
    :
```

図 3-4-6 (1)

```
#define NX (3)
#define NY (2)
  __device__ float dD[NY][NX];                 ⑥
__global__ void kernel(){
  int ix = blockIdx.x*blockDim.x + threadIdx.x;
  int iy = blockIdx.y*blockDim.y + threadIdx.y;
  if (ix<NX && iy<NY)
    dD[iy][ix] = 1.0f;
}
int main(void){
  float D[NY][NX];
  size_t size = NX*NY*sizeof(float);
    :
  cudaMemcpyToSymbol(dD,D,size,0,
                     cudaMemcpyHostToDevice);
  kernel<<<1,dim3(NX,NY)>>>();
  cudaMemcpyFromSymbol(D,dD,size,0,
                     cudaMemcpyDeviceToHost);
   :
```

図 3-4-6 (2)

__device__ 修飾子とコアレスアクセス

__device__ 修飾子で宣言した変数／配列は、グローバルメモリ上に確保されるので、cudaMalloc を使用して確保した場合と、コアレスアクセスに関する注意点はほぼ同じです (1 次元の場合は 3-2 節、2 次元の場合は 3-3 節参照)。相違点についてのみ以下に述べます。

- 「CUDA C Programming Guide」（付録参照）の 5.3.2.1.1 節によると、「グローバルメモリ内に存在する変数（スカラー変数と配列を意味すると思われます）は、グローバルメモリ上の少なくとも 256 バイト境界から開始する。」と記載されています。従って、cudaMalloc で確保した場合と同様に、図 3-4-7 で宣言した単精度の配列 dD[10] と dE[10] は、図 3-4-8 に示すように（少なくとも）256 バイト境界から開始します。

```
__device__ float dD[10];
__device__ float dE[10];
```

図 3-4-7

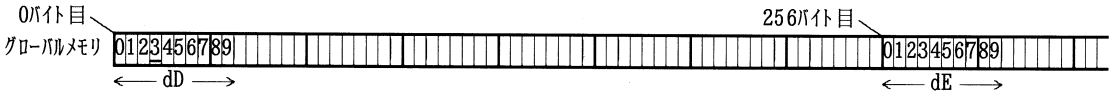


図 3-4-8

- 図 3-4-9 (1) (2) に示すように、ホスト側の配列が例えば D[16][34] で、そのうち実際に処理するのが着色した D[16][30] の場合、デバイス側の配列の 2 次元目の大きさを、図 3-4-10 (1) のように 30 でなく、図 3-4-10 (2) のように 16 の倍数の 32 にした方が、コアレスアクセスが効率よく行われます (3-3 節参照)。この場合、図 3-4-11 (1) を、図 3-4-11 (2) のようにすれば、自動的に 16 の倍数にすることができます (3-3 節参照)。
  なお、cudaMalloc の場合の、cudaMallocPitch に相当する関数は提供されていません。また、cudaMalloc の場合は、下記の着色した部分（メモリ上で不連続）のみをコピーする、CUDA 関数 cudaMemcpy2D が提供されていますが、__device__ 修飾子の場合は同様の関数は提供されていないようです。

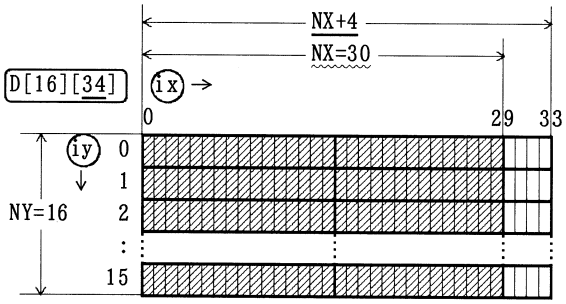


図 3-4-9 (1)

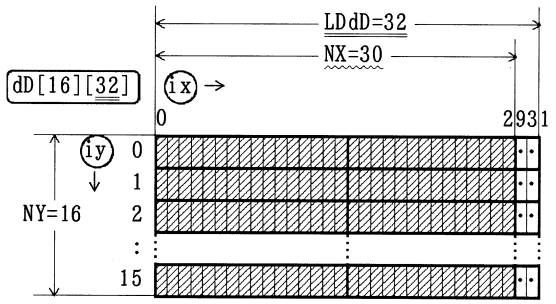


図 3-4-9 (2)

```
__device__ float dD[16][30];
```

図 3-4-10 (1)

```
__device__ float dD[16][32];
```

図 3-4-10 (2)

```
#define NX (30)
#define NY (32)
__device__ float dD[NY][NX];
```

図 3-4-11 (1)

```
#define NX (30)
#define NY (32)
#define LDdD (((NX+15)/16)*16)
__device__ float dD[NY][LDdD];
```

図 3-4-11 (2)

3-5　コンスタントメモリ

本節では、「__constant__」変数型修飾子を使用して、デバイスメモリ上のコンスタントメモリに変数／配列を確保する方法について説明します。

__constant__ 修飾子を指定した変数／配列の特性を以下に示します (3-1 節参照)。

- 作成される場所：デバイスメモリ内のコンスタントメモリ（オフチップ：低速）上に作成されます。ただし、後述するように、データがコンスタントキャッシュ (ストリーミング・マルチプロセッサーあたり 8K バイト) (オンチップ：高速) に入った場合は、高速にアクセスされます。
- アクセスできるスレッドの範囲：全ブロックの全スレッドからアクセスすることができます。ただし参照のみが可能で、値を更新することはできません。
- 存在する期間：プログラムの開始から終了までの間、存在します。
- 容量：65536 バイト (単精度だと 16384 個) です。

指定方法

図 3-5-2 に示すように、デバイス側の、(デバイスメモリ内の) コンスタントメモリに、大きさ 128 の配列 dC を確保し、ホスト側の配列 C のデータをコピーするプログラムを図 3-5-1 に示します。

- ①に示すように __constant__ 修飾子を付けて宣言した配列 dC は、デバイスメモリ内のグローバルメモリに確保されます。なお、①はプログラム内のグローバル領域に記述します。
- ④で配列 C を配列 dC にコピーします。コピーは、cudaMemcpy ではなく、__device__ 修飾子の場合と同様に cudaMemcpyToSymbol を使用します (cudaMemcpyToSymbol の使用方法は 3-4 節を参照して下さい)。
- ⑤でカーネル関数を実行し、②で、コンスタントメモリとして指定した配列 dC の要素 dC[0] を参照します。前述のように、配列 dC は、カーネル関数内で更新することはできず、参照のみ行うことができます。
- ②で要素 dC[0] を参照する部分の動作を説明します。配列 dC が cudaMalloc で確保した通常の配列の場合は、図 3-5-2 の⑧に示すように、グローバルメモリから直接レジスターにロードされます (低速)。一方コンスタントメモリの配列の場合は、⑥に示すように、dC[0] だけでなく、近隣の要素 (例えば dC[0]～dC[31]) がまとめてコンスタントキャッシュ (高速のオンチップメモリ) にロードされ (低速)、次に⑦に示すように、要素 dC[0] がレジスターにロードされます (高速)。②の処理が終了した後で、要素 dC[0]～dC[31] を参照した場合は、コンスタントキャッシュ上に dC[0]～dC[31] が存在するので、ロードは高速に行われます。本例では②の後に要素 dC[0]～dC[31] を参照していないので、コンスタントメモリを使用したメリットはありません。
- シェアードメモリ (3-6 節参照) の使用を節約するために、カーネル関数で参照するだけの変数／配列を、シェアードメモリの代用として、コンスタントメモリを使用する用途もあります。
- コンスタントキャッシュの構造については、マニュアルに記述がないので不明です。

```
__constant__ float dC[100];                    ①
__device__   float dD;

__global__ void kernel(){
  dD = dC[0] + 1.0f;                           ②
}
```

```
int main(void){
  float C[100];                                ③
  配列Cに値を設定します。
  size_t size = 100*sizeof(float);
  cudaMemcpyToSymbol                           ④
     (dC,C,size,0,cudaMemcpyHostToDevice);     ④
  kernel<<<1,1>>>();                           ⑤
    :
```

図 3-5-1

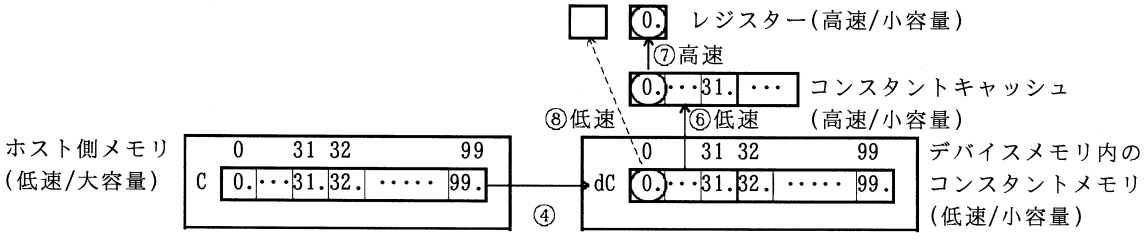


図 3-5-2

コンスタントメモリの最大容量

コンスタントメモリの最大容量は、図 3-5-3 の①に示すように、65536 バイト (単精度実数だと 16384 個) です。このプログラムを、図 3-5-4 の②の下線部を指定してコンパイルすると、④の下線部に示すように、コンスタントメモリ (cmem) が 65536 バイト使用されたことが表示されます。波線の部分の意味は不明です。

①で 16384 より大きな値を指定した場合は、⑤が表示されます。

```
__constant__ float dC[16384];          ①
__device__   float dD;
__global__ void kernel(){
  dD = dC[0] + 1.0f;
}
```

図 3-5-3

```
$ nvcc -Xptxas -v -arch=sm_13 -c test.cu ↵                                    ②
ptxas info    : Compiling entry function '_Z6kernelv' for 'sm_13'              ③
ptxas info    : Used 2 registers, 65536 bytes cmem[0], 4 bytes cmem[14]        ④
/tmp/tmpxft_00003d52_00000000-7_test.cpp3.i(0): Error: Const space overflowed ⑤
```

図 3-5-4

初期値の設定

__constant__ 修飾子でグローバルメモリ上に確保した変数／配列は、通常の C 言語や __device__ 修飾子で宣言した変数／配列と同様に、初期値を設定することができます。図 3-5-5 (1) のように設定すると、図 3-5-5 (2) のように初期化されます。③では全ての要素が 0 になります。④では 1 つ目の要素が 1 で他の要素が 0 になります。

```
__constant__ int dK[2] = {0};   ③
__constant__ int dL[2] = {1};   ④
__constant__ int dM[2] = {2,3}; ⑤
__constant__ int dN    =  4;    ⑥
```

図 3-5-5 (1)

	0	1
dK	0	0
dL	1	0
dM	2	3
dN	4	

図 3-5-5 (2)

コンスタントメモリを使用する際の注意点

コンスタントメモリでは、前述のように、コンスタントキャッシュが使用されます。このため、通常の CPU のキャッシュと同様の注意が必要となります。以下では、キャッシュに関連する現象と対処方法のみを説明し、理由については、キャッシュの構造から説明する必要があるので割愛します。興味のある方は、付録の「■　書籍／雑誌」の [2] の第 4 章などを参照して下さい。

なお、プロファイラーのマニュアル（付録参照）によると、シェアードメモリと同様に、コンスタントメモリでも、バンクコンフリクトが発生するようなので、シェアードメモリを使用する場合と同様の注意 (3-6 節参照) も必要だと思われます。

以下は、各スレッドが、コンスタントメモリ上の配列の、1 要素のみを参照するのではなく、複数の要素を参照する場合の例です。

- 図 3-5-6 (1) の①で、配列 dC をコンスタントメモリで指定し、③でカーネル関数を実行します。②では、配列 dA を、図 3-5-7 (1) の (1), (2), (3), ... の順にストライド 10 で (10 要素ずつとびとびに) アクセスしています。キャッシュを使用した計算機の場合、このようにメモリ上で大きなストライドでアクセスすると、キャッシュミスという現象が発生して速度が遅くなります。図 3-5-6 (2) もストライドが 10 なので、同様に遅くなります。
  一方、図 3-5-6 (3) では、図 3-5-7 (2) に示すようにストライド 1 でアクセスしており、キャッシュミスはあまり発生しません。可能性であれば、図 3-5-6 (3) のように、ストライド 1 でアクセスして下さい。

```
#define N (10000)
  __constant__ float dC[N];                    ①
  __device__   float dD[30*32];
__global__ void kernel(){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for(int j=0;j<10000;j+=10){
    sum = sum + dC[j];                         ②
  }
  dD[i] = sum;
}
int main(void){
    :
  kernel<<<30,32>>>();                         ③
    :
```

図 3-5-6 (1)　×

```
    :
for(int j=0;j<1000;j++){
  sum = sum + dC[j*10];
}
    :
```

図 3-5-6 (2)　×

```
    :
for(int j=0;j<1000;j++){
  sum = sum + dC[j];
}
    :
```

図 3-5-6 (3)　○

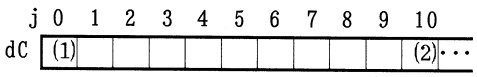


図 3-5-7 (1)　×　ストライド 10

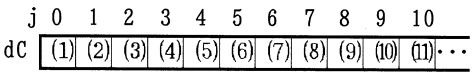


図 3-5-7 (2)　○　ストライド 1

- 図 3-5-8 (1) の①で、2 次元配列 dC をコンスタントメモリで指定し、③でカーネル関数を実行します。C 言語の場合、2 次元配列の要素は、図 3-5-9 (1) (2) の矢印に示すように、メモリ上で dA[0][0], dA[0][1], ... の順に、「右側の添字が先に動く順番に」並びます (Fortran では「左側の添字が先に動く順番に」並びます)。
  図 3-5-8 (1) の②の 2 重ループでは、図 3-5-9 (1) の (1), (2), (3), ... の順に、ストライド 100 でアクセスされるため、キャッシュミスが発生して速度が低下します。一方 2 重ループの順番を逆にした図 3-5-8 (2) では、図 3-5-9 (2) に示すように、ストライド 1 でアクセスされるので、キャッシュミスはあまり発生しません。従って C 言語の場合、可能であれば、図 3-5-8 (2) に示すように、配列 dC の右側の添字 (ix) を内側のループで反復させるようにして下さい (Fortran の場合は左側の添字を内側のループで反復)。

```
#define NX (100)
#define NY (90)
  __constant__ float dC[NY][NX];          ①
  __device__   float dD[30*32];
__global__ void kernel(){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for(int ix=0;ix<NX;ix++){               ②
    for(int iy=0;iy<NY;iy++){             ②
      sum = sum + dC[iy][ix];
    }
  }
  dD[i] = sum;
}
int main(void){
    :
  kernel<<<30,32>>>();                    ③
    :
```

図 3-5-8 (1)　×

```
    :
for(int iy=0;iy<NY;iy++){
  for(int ix=0;ix<NX;ix++){
    sum = sum + dC[iy][ix];
  }
}
    :
```

図 3-5-8 (2)　○

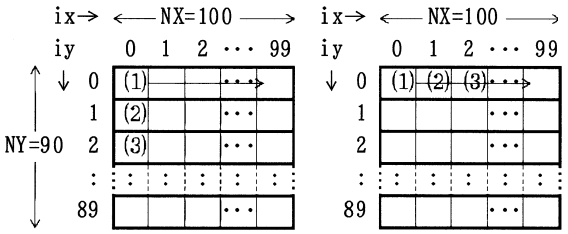


図 3-5-9 (1)　×　　図 3-5-9 (2)　○

- 図 3-5-10 (1) の②で、配列 dC～dG をコンスタントメモリで指定し、④でカーネル関数を実行します。③では、配列 dA～dE の要素をストライド 1 でアクセスしており、通常ならばキャッシュミスはあまり発生しません。ところが①に示すように、大きさが 2 のベキ乗 (1024, 2048 など) の配列をある程度以上の個数使用すると、ストライドが 1 でもキャッシュミスが発生することがあります (理由は付録の文献 [2] を参照)。
  このような場合、試しに図 3-5-10 (2) に示すように、配列の大きさを 2 のベキ乗より少し大きくし、もし速度が速くなるようなら、試行錯誤で配列の大きさを調整して下さい。

```
#define N (1024)                          ①
  __constant__ float
  dC1[N],dC2[N],dC3[N],dC4[N],dC5[N];     ②
  __device__ float dD[30*32];
__global__ void kernel(){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for(int j=0;j<1024;j++){
    sum = sum + dC1[j] + dC2[j] + dC3[j]  ③
  }           + dC4[j] + dC5[j];          ③
  dD[i] = sum;
}
```

```
#define N (1024)
  __constant__ float
  dC1[N+5],dC2[N+5],dC3[N+5],dC4[N+5],dC5[N+5];
    :
```

図 3-5-10 (2)　○

```
int main(void){
    :
  kernel<<<30,32>>>();                    ④
    :
```

図 3-5-10 (1)　×

3-6　シェアードメモリ

CUDA プログラミングでは、高速なシェアードメモリの有効利用が、速度向上のポイントとなります。本節では、「__shared__」変数型修飾子を使用して、シェアードメモリに変数／配列を確保する方法について説明します。__shared__ 修飾子を指定した変数／配列の特性を以下に示します (3-1 節参照)。

- 作成される場所：ブロックごとに、ストリーミング・マルチプロセッサー上のシェアードメモリ（オンチップ：高速）上に作成されます。
- アクセスできるスレッドの範囲：当該ブロック内の全スレッドからのみアクセスすることができます。
- 存在する期間：当該ブロックがストリーミング・マルチプロセッサーに配置されてから、終了するまでの間（つまり、当該ブロック内の全スレッドが処理を終了するまでの間)、存在します。
- 容量：ストリーミングマルチプロセッサーあたり 16384 バイト (単精度だと 4096 個) です。

指定方法 1（配列を静的に確保）

変数／配列をシェアードメモリ上に確保する方法は 2 通りあります。

図 3-6-1 に示すように、__shared__ 修飾子を付けた変数や配列は、シェアードメモリに置かれます。ホスト側プログラムからシェアードメモリの変数／配列に直接データをコピーすることはできません。ホストからグローバルメモリ内の配列 (例えば図 3-6-1 の配列 dD) にデータをコピーした後、②に示すようにグローバルメモリからシェアードメモリにデータをコピーします。シェアードメモリの変数／配列からホスト側プログラムへのコピーも、③に示すように、グローバルメモリの変数／配列を介して行います。

__shared__ 修飾子は、図 3-6-2 の④のようにグローバル領域内、⑤のようにカーネル関数内、⑥のようにカーネル関数から呼ばれる関数内で指定することができます。④は (1) と (3) から、⑤は (2) から、⑥は (4) から使用することができます。

```
#define N (1)
  __device__ float dD[N];
__global__ void kernel(){
  __shared__ float dS[N]; ①
  int i;
  for(i=0;i<N;i++){
    dS[i] = dD[i];          ②
    dD[i] = dS[i] + 1.0f; ③
  }
}
```

図 3-6-1

```
  __shared__ float dS;     ④
__global__ void kernel(){
  __shared__ float dT;     ⑤
  dS = ～;                 (1)
  dT = ～;                 (2)
  kernel2();
}
```

```
__device__ void kernel2(){
  __shared__ float dU;     ⑥
  dS = ～;                 (3)
  dU = ～;                 (4)
}
```

図 3-6-2

図 3-6-1 で N ＝ 1 の場合、図 3-6-3 の①に示すように、4 バイトの領域がシェアードメモリ上に確保されます。なお、__shared__ 修飾子で指定したシェアードメモリは、ブロック内の全スレッドが共有するので、この値は 1 スレッドではなく 1 ブロックでの値です。

図 3-6-1 のプログラムの場合、N ＝ 4092 までは、②に示すようにシェアードメモリ上に確保できますが、N ＝ 4093 以上ではシェアーメモリが確保できず、③に示すようにコンパイルエラーになります。

```
$ nvcc -Xptxas -v -arch=sm_13 -c test.cu ↵
ptxas info    : Compiling entry function '_Z6kernelv' for 'sm_13'
ptxas info    : Used 3 registers, 4+16 bytes smem, 4 bytes cmem[14]    ①   ← 図3-6-1でN=1
ptxas info    : Used 4 registers, 16368+16 bytes smem, 4 bytes cmem[1],②   ← 図3-6-1でN=4092
                4 bytes cmem[14]                                       ②
ptxas info    : Used 4 registers, 16372+16 bytes smem, 4 bytes cmem[1],    ← 図3-6-1でN=4093
                4 bytes cmem[14]
ptxas error   : Entry function '_Z6kernelv' uses too much shared data  ③
                (0x3ff4 bytes + 0x10 bytes system, 0x4000 max)         ③
```

図 3-6-3

指定方法 2 (1) (1 個の配列を動的に確保)

シェアードメモリに確保したい配列の大きさが、コンパイル時に確定せず、実行時に確定する場合、前述の方法で配列を指定することはできません。このような配列を実行時に動的にシェアードメモリ上に確保する方法を図 3-6-4 に示します。

まず①で、シェアードメモリに確保したい配列の要素数を設定し、②でバイト数に変換し、それを③で実行構成の 3 つ目の引数に指定します。3 番目の引数は今までの例では指定しませんでしたが、指定しない場合は、デフォルトで 0 になります。

カーネル関数内で④を指定すると、図 3-6-5 に示すように、要素数 10 の配列 dS がシェアードメモリ上に動的に確保されます。なお、④は⑤のようにグローバル領域に記述しても構いません。

この方法で確保した配列 S は 1 次元なので、多次元配列風に取り扱いたい場合は、3-3 節で説明したマクロを使用して下さい。

図 3-6-6 の⑥のシェアードメモリの容量 smem の値 (16 + 16) は、コンパイル時に確定している量が表示されます。従って④で確保した配列 S の容量は smem には現れず、①の lenS の値を変えても⑥の smem の値は変わりません。

本例では、lenS = 4089 以上だとシェアードメモリが足りなくなり、プログラムを実行すると③のカーネルカーネル関数の実行が失敗します。しかしデフォルトでは何もメッセージが表示されないので、一見プログラムが正常終了したように見えます。図 3-6-7 の⑨〜⑫のエラーチェックルーチン (4-2 節参照) を付加すると、プログラムの実行中に図 3-6-6 の⑦のエラーメッセージが表示されます。lenS = 4097 以上だと⑧が表示されます。

```
  __device__ float dD[10000];
 (extern __shared__ float dS[];)  ⑤
__global__ void kernel(int lenS){
  extern __shared__ float dS[];   ④
  int i;
  for (i=0;i<lenS;i++) {
    dS[i] = dD[i];
    dD[i] = dS[i] + 1.0f;
  }
}
int main(void){
    :
  int lenS = 10;                  ①
  int size = lenS*sizeof(float); ②
  kernel<<<1,1,size>>>(lenS);     ③
    :
```

図 3-6-4

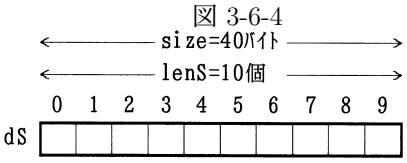


図 3-6-5

```
#include <cutil.h>                  ⑨
  __device__ float dD[10000];
void CUDA_ERROR_CHECK(char *msg){   ⑩
  :
}
__global__ void kernel(int lenS){
  extern __shared__ float dS[];
  int i;
  for (i=0;i<lenS;i++) {
    dS[i] = dD[i];
    dD[i] = dS[i] + 1.0f;
  }
}
int main(void){
    :
  int lenS = 4089;
  int size = lenS*sizeof(float);
  kernel<<<1,1,size>>>(lenS);
  CUDA_SAFE_CALL(cudaThreadSynchronize()); ⑪
  CUDA_ERROR_CHECK("kernel");       ⑫
    :
```

図 3-6-7

```
$ nvcc -Xptxas -v -arch=sm_13 -c test.cu ↵
ptxas info    : Compiling entry function '_Z6kerneli' for 'sm_13'
ptxas info    : Used 4 registers, 16+16 bytes smem, 4 bytes cmem[14] ⑥
                CUDA error: kernel: invalid argument.                ⑦
                CUDA error: kernel: invalid configuration argument.  ⑧
```

図3-6-4で
← lenS=10
← lenS=4089
← lenS=4097

図 3-6-6

指定方法 2 (2)（複数の配列を動的に確保）

図 3-6-8 のように複数の配列 (float S1[2], int S2[3], float S3[4]) をシェアードメモリに動的に確保する方法を図 3-6-9 に示します。まず①で、配列 S1，S2，S3 の要素数を設定し、②で S1，S2，S3 の合計の大きさをバイト数に変換し、それを③で実行構成の 3 つ目の引数に指定します。

カーネル関数内で④を指定すると、図 3-6-8 に示すように、配列 S_ALL がシェアードメモリに動的に確保されます。この配列 S_ALL は、配列 S1，S2，S3 が配置される仮の領域で、プログラム内では使用しないので、④で指定する名前 (S_ALL) および型 (float) は任意で構いません。

⑤で配列 S_ALL を配列 S1，S2，S3 に切り分けます (図 3-6-8 参照)。⑤の太線で S1、二重線で S2、波線で S3 の属性を指定することに注意して下さい。

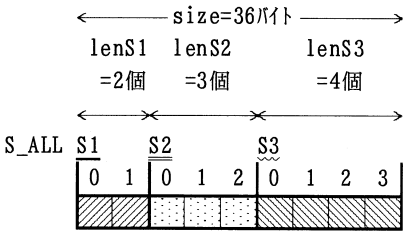


図 3-6-8

```
__global__ void kernel(int lenS1,int lenS2,int lenS3){
  extern __shared__ float S_ALL[];                          ④
  int i;
  float *S1 = (float*)S_ALL;                                ↑
    int *S2 =   (int*)&S1[lenS1];                           ⑤
  float *S3 = (float*)&S2[lenS2];                           ↓
  for (i=0;i<lenS1;i++) S1[i] = ～;
  for (i=0;i<lenS2;i++) S2[i] = ～;
  for (i=0;i<lenS3;i++) S3[i] = ～;
    :
int main(void){
  int lenS1,lenS2,lenS3;
  lenS1=2;lenS2=3;lenS3=4;                                  ①
  int size = (lenS1+lenS3)*sizeof(float)                    ②
           + lenS2*sizeof(int);
  kernel<<<1,1,size>>>(lenS1,lenS2,lenS3);                  ③
    :
```

図 3-6-9

データ型（の大きさ）が異なる配列を上記の方法で確保する場合、配列の順序に注意する必要があります。図 3-6-10 の 1 マスは 1 バイトを表します。図 3-6-10 では、short 型 (2 バイト) の配列 S1[2] と float 型 (4 バイト) の配列 S2[2] を上記の方法で確保しています。この場合、各配列の先頭は以下の位置に置く必要があります。

配列 S1 のように 1 要素が 2 バイト (short) の配列の先頭は、2 バイトの倍数 $(0, 2, 4, \ldots)$ に置く必要があります。同様に、配列 S2 のように 1 要素が 4 バイト (float) の配列の先頭は、4 バイトの倍数 $(0, 4, 8, \ldots)$ に置く必要があります。

図 3-6-10 では、S1，S2 がともに上記の条件を満たしており、問題ありません。S1 の要素数が 1 つ増えた図 3-6-11 では、S2 の先頭が 4 の倍数でない 6 バイトで開始しているので、上記の条件を満足しておらず、誤動作します。図 3-6-11 の S1 と S2 を逆にした図 3-6-12 では、S1，S2 が共に上記の条件を満たしており、問題ありません。図 3-6-12 のように、1 要素のバイト数が大きい順に配列を指定すれば、常にこの条件を満足します。

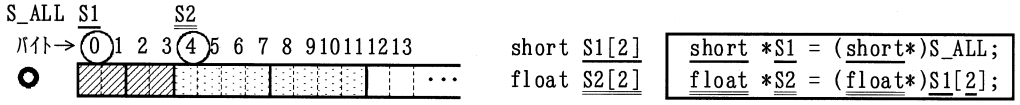


図 3-6-10

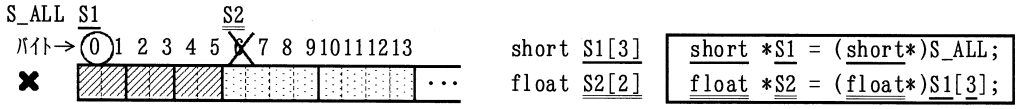


図 3-6-11

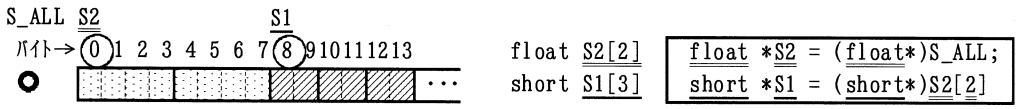


図 3-6-12

シェアードメモリを使う局面 (1)（同一スレッドが同じデータを何度も使用する場合）

図 3-6-13 (1) を、ブロック数 2、ブロック内のスレッド数 2 で実行した場合の動作を図 3-6-14 (1) に示します。図 3-6-13 (1) は、グローバルメモリ上の同じ要素 dX[i] を 3 回ロードしているので、時間がかかります。この場合、図 3-6-13 (2) のように一時変数 temp を導入すると、dX[i] のロードは 1 回となり、速度が向上します。一時配列 temp 自体は、図 3-6-14 (2) に示すように、スレッドごとにレジスター（高速）に置かれます (3-8 節参照)。なお、ホスト側のコンパイラでは、通常、図 3-6-13 (1) を 3-6-13 (2) に自動的に置き換えると思われますが、CUDA では、アセンブラリスト (2-7 節参照) で調べたところ、置き換えないようです。

例えばレジスターの数が足りないような場合、一時変数 temp の代わりにシェアードメモリを使用することができます。プログラムを図 3-6-13 (3) に示します。シェアードメモリの配列 dS は、図 3-6-14 (3) に示すように、各ブロックごとに確保されるので、ブロック内のスレッド数と同じ要素数 (本例では 2) にし、スレッド ID (本例では0, 1) を添字に使用してアクセスします。なお、本例では、図 3-6-13 (4) (図 3-6-13 (2) と似ています) のように、シェアードメモリとしてスカラー変数 dS を使用するのは間違いです。シェアードメモリ上の変数 dS は、ブロック内の全スレッドからアクセスできるので、図 3-6-14 (4) に示すように、ブロック内の全スレッドが、配列 dX の自分が担当する要素を、同じ変数 dS にロードしてしまい、結果が不定になります。

この例のように、同一スレッドがグローバルメモリ上の同じデータを何度も使用する場合、そのデータを一度シェアードメモリにロードしてから使用すると、速度が速くなる場合があります。

```
__global__ void kernel(～)
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dX[i];
  dB[i] = dX[i];
  dC[i] = dX[i];
}
```

図 3-6-13 (1)

```
   :
float temp = dX[i];
dA[i] = temp;
dB[i] = temp;
dC[i] = temp;
   :
```

図 3-6-13 (2)

```
   :
__shared__ float dS[2];
dS[threadIdx.x] = dX[i];
dA[i] = dS[threadIdx.x];
dB[i] = dS[threadIdx.x];
dC[i] = dS[threadIdx.x];
   :
```

図 3-6-13 (3)

```
   :
__shared__ float dS;
dS = dX[i];
dA[i] = dS;
dB[i] = dS;
dC[i] = dS;
   :
```

図 3-6-13 (4)　×　間違い

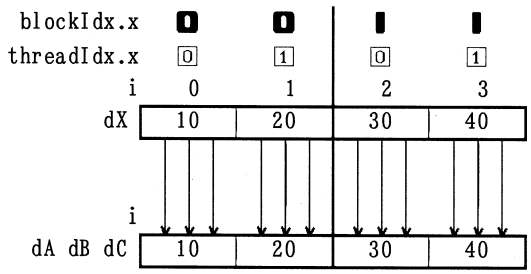


図 3-6-14 (1)

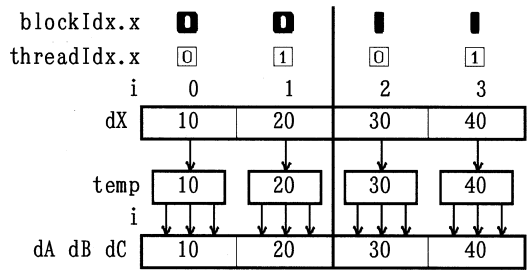


図 3-6-14 (2)

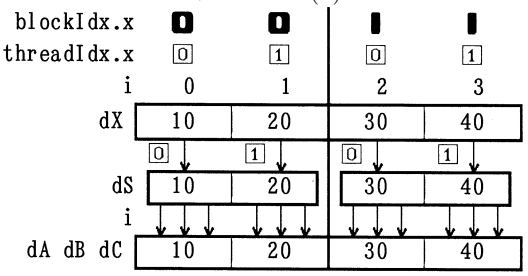


図 3-6-14 (3)

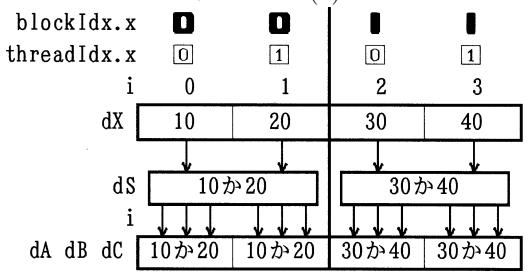


図 3-6-14 (4)　×　間違い

シェアードメモリを使う局面 (2)（複数のスレッドが同じデータを使う場合）

以下の例は、N 体問題 (8-1 節参照) や行列乗算 (8-4 節参照) でシェアードメモリを使う方法の基礎となります。

図 3-6-18 (1) (2) では、②でブロック数が 2、ブロック内のスレッド数が 2 で実行を行い、①で各スレッドは、配列 dA[0]～dA[3] の合計を、配列 dB の自分が担当する要素に代入します（全スレッドが全く同じ計算を行っているので、計算自体は意味がありません）。このプログラムを、シェアードメモリを使用して高速化する方法を説明します。

まず簡単な修正を行います。①では、1 スレッドあたり、グローバルメモリ上の配列 dA のロードを 4 回 (N = 4 なので)、配列 dB のロード／ストアを各 4 回行っており、時間がかかります。そこで図 3-6-19 (1) の③に示すようにスカラー変数 sum を導入し、④, ⑤のように変更します。変数 sum は、3-6-19 (2) に示すように、各スレッドごとに、高速なレジスターに置かれます (3-8 節参照)。

この修正により、1 スレッドあたり、④で行う配列 dA のロード 4 回は変わりませんが、配列 dB は⑤でロード 0 回、ストア 1 回に減少します。

```
#define N (4)
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dB[i] = 0.0f;
  for(int j=0;j<N;j++)
    dB[i] = dB[i] + dA[j];     ①
  }
}
int main(void){
    :
  kernel<<<2,2>>>(dA,dB);      ②
    :
```

図 3-6-18 (1)

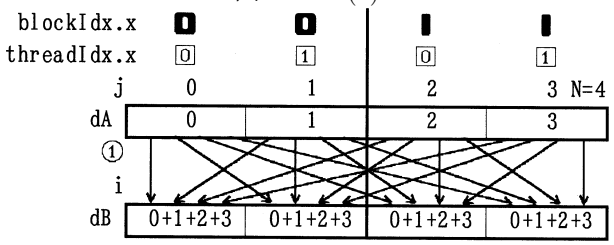


図 3-6-18 (2)

```
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0.0f;            ③
  for(int j=0;j<N;j++)
    sum = sum + dA[j];         ④
  }
  dB[i] = sum;                 ⑤
}
```

図 3-6-19 (1)

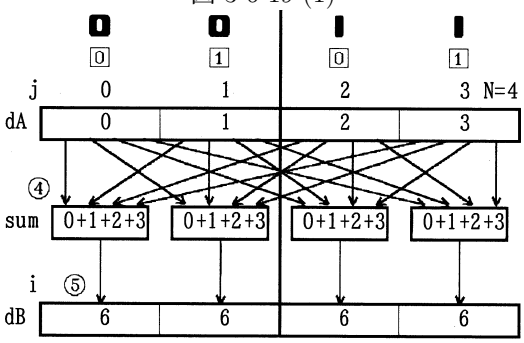


図 3-6-19 (2)

次に、シェアードメモリを用いて、図 3-6-19 (1) の④での、配列 dA のロード 4 回を減少させる方法を説明します。シェアードメモリを導入したプログラムを図 3-6-20 (1) に示します。

まず全体の動作を説明します。図 3-6-20 (2) に示すように、まず dA[0], dA[1] を各ブロックのシェアードメモリ dS[0], dS[1] にコピーし、加算を行います。次に図 3-6-20 (3) に示すように、dA[2], dA[3] を各ブロックのシェアードメモリ dS[0], dS[1] にコピーし、加算を行います。

以下で図 3-6-20 (1) の説明をします。

- 図 3-6-20 (1) の⑥で、シェアードメモリ上に配列 dS[2] を確保します (配列 dS はブロックごとに確保され、ブロック内のスレッド数が 2 なので、配列の大きさは 2 となります)。
- ⑦の j は、ブロック内のスレッド数 (本例では 2) ずつ反復します (j = 0, 2)。j = 0 のときの動作が図 3-6-20 (2)、j = 2 のときの動作が図 3-6-20 (3) です。
- ⑧で各スレッドは、配列 dA のうち、自分が担当する要素を配列 dS にコピーします。
- ⑨の __syncthreads() は、同一ブロック内の全スレッド間の同期を取る命令です。あるスレッドが⑨に到達すると、そのスレッドは、そのスレッドが所属するブロック内の全スレッドが⑨に到達するまで、⑨で待機します。全スレッドが到達したら、(全スレッドは) ⑩に進みます。本例では、全スレッドが⑧のロードを終了してからでないと⑩の加算を開始できないため、⑨で同期を取ります (同期の詳細は 4-1 節参照)。なお、図 3-6-20 (2) (3) の⑨の横線は、同期を取っていることを表します。
- ⑩と⑪で、各スレッドは配列 dS 内の全要素を加算します。
- あるスレッドが⑩, ⑪を計算しているときに、そのスレッドが所属するブロック内の他のスレッド (2 つのスレッドは別のワープに所属) が、⑩, ⑪を先に終了し、⑦の次の反復で⑧を実行してしまうのを防ぐため、⑫で再びブロック内の全スレッドの同期を取ります (同期の詳細は 4-1 節参照)。
- ⑦で j = 2 となり、図 3-6-20 (3) の処理を同様に行います。
- 最後に⑬で、合計を配列 dB の自分が担当する要素に書き込みます。

図 3-6-20 (1) では、1 スレッドあたり、配列 dA のロードが 2 回 (⑦のループ反復が 2 回で、1 反復あたり⑧でロードを 1 回なので) に減少します。なお、⑬の配列 dB のストア 1 回は、図 3-6-19 と同じで変わりません。

この例のように、複数のスレッドがグローバルメモリ上の同じデータを使用する場合、そのデータをシェアードメモリにロードしてから使用すると、速度が速くなることがあります。

割り切れない場合の処理

図 3-6-20 (1) では、全要素数 N (= 4) がブロックあたりの全スレッド数 (= 2) で割り切れましたが、一般には割り切れません。割り切れない場合 (例えば要素数 N = 3) の処理を図 3-6-21 (1)〜(3) に示します。図 3-6-21 (2) (3) の点線で示した部分の処理を行わないようにするため、図 3-6-21 (1) の⑭〜⑲の下線部を追加しています。なお、図 3-6-21 (2) (3) の「*」は、加算しない要素（値は不定）を意味します。

- 配列 dA の範囲が dA[0]〜dA[2] なので、dA[3] からのロードを行わないように、⑭の下線部を指定します。
- 図 3-6-21 (2) (3) の一番右のスレッドが⑱の加算を行わないように、⑰の if 文を指定します。
- ⑮と⑯で設定する変数 jjsize は、配列 dS 内で加算を行う要素の数（図中の〓内の要素数）を示します。図 3-6-21 (3) で、加算の対象外である dS[1] の値を加算しないように、⑱の下線部で変数 jjsize を使用します。
- 配列 dB の範囲が dB[0]〜dB[2] なので、dB[3] へのストアを行わないように、⑲の下線部を指定します。
- 同期を取る __syncthreads() は、同一ブロック内の全スレッドが実行する必要があります。上記で if 文を追加したことによって、実行しないスレッドが発生しないように注意して下さい (4-1 節参照)。

```
__global__ void kernel(float *dA,float *dB){
  __shared__ float dS[2];                          ⑥
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for(int j=0;j<N;j+=blockDim.x){                  ⑦
    dS[threadIdx.x] = dA[j+threadIdx.x];           ⑧
    __syncthreads();                               ⑨
    for(int jj=0;jj<blockDim.x;jj++){              ⑩
      sum = sum + dS[jj];                          ⑪
    }
    __syncthreads();                               ⑫
  }
  dB[i] = sum;                                     ⑬
}
```

図 3-6-20 (1)

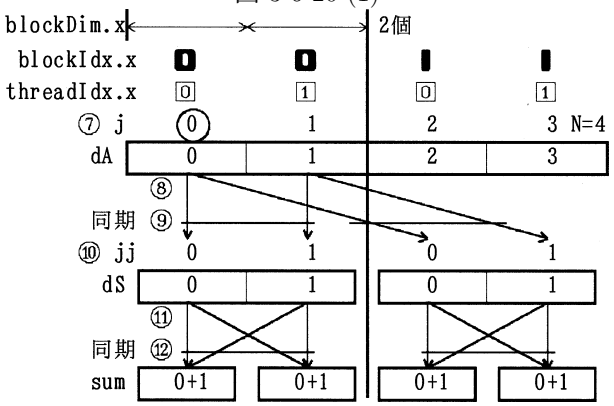


図 3-6-20 (2)　j＝0 のとき

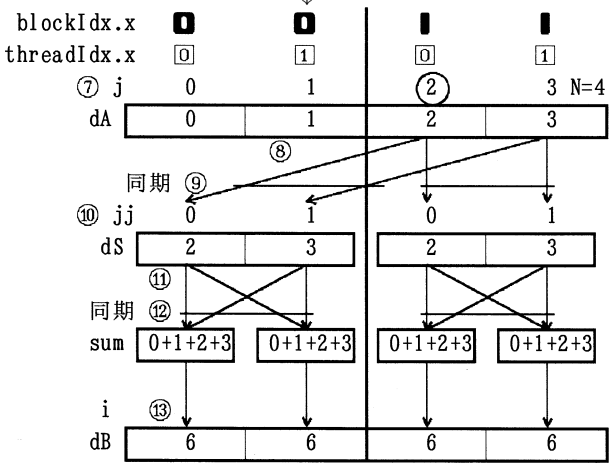


図 3-6-20 (3)　j＝2 のとき

```
__global__ void kernel(float *dA,float *dB){
  __shared__ float dS[2];
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0.0f;
  for(int j=0;j<N;j+=blockDim.x){
    if(j+threadIdx.x<N)                            ⑭
      dS[threadIdx.x] = dA[j+threadIdx.x];         ⑭
    __syncthreads();
    int jjsize = blockDim.x;                       ⑮
    if(N-j<blockDim.x) jjsize = N-j;               ⑯
    if(i<N){                                       ⑰
      for(int jj=0;jj<jjsize;jj++){                ⑱
        sum = sum + dS[jj];
      }
    }
    __syncthreads();
  }
  if(i<N) dB[i] = sum;                             ⑲
}
```

図 3-6-21 (1)

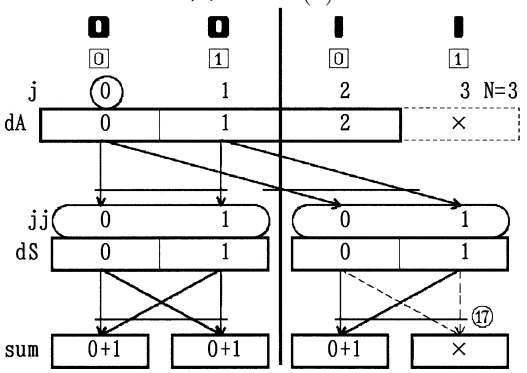


図 3-6-21 (2)　j＝0 のとき

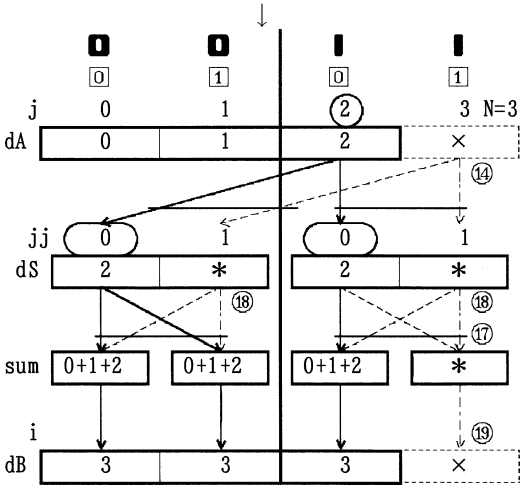


図 3-6-21 (3)　j＝2 のとき

バンクコンフリクト (1 次元配列)

シェアードメモリは、図 3-6-23 (1) に示すように、16 個の区画（バンクと呼びます）に分かれています。各バンクは同時に（並列に）ロード／ストアすることができます（詳細は後述します）。1 つのバンクの大きさは 4 バイトなので、int 型 (4 バイト) や float 型 (4 バイト) は 1 バンクに 1 要素が入り、double 型 (8 バイト) は連続した 2 バンクに 1 要素が入ります。以下では float 型 (4 バイト) を想定して説明します。

図 3-6-22 では、①で 1 次元配列 S[1000] をシェアードメモリに確保し、②で参照（ロード）しています。配列の要素 S[0]，S[1]，...は、図 3-6-23 (1) に示すように、バンク 0，バンク 1，...の順に配置されます。

1 つのスレッドが担当する要素と、隣のスレッドが担当する要素の間隔のことをストライド（単位は要素数）と呼ぶことにします。図 3-6-22 の変数 stride で、ストライドの値を設定します。

前述のように、メモリ上のデータのロード／ストアは、ハーフワープ (ワープ内の前半、または後半の 16 スレッド) 単位に行われます。図 3-6-22 をブロック数 1、ブロック内のスレッド数 16 (スレッド ID = 0, 1, ..., 15) で実行した場合、②の配列 S のロードでは、各バンクが同時に（並列に）動作するので、図 3-6-23 (1) の〓に示す 16 個の↑が一度にロードされ、高速です。

stride = 2 の場合、図 3-6-23 (2) に示すように、ハーフワープ内の各スレッドがロードする要素が、同じバンク内に 2 個あります (例えば S[0] と S[16])。この場合、まず S[0]，S[2]，...，S[14] の要素が一度にロードされ、次に S[16]，S[18]，...，S[30] の要素が一度にロードされます。ロードが 2 回行われるため、図 3-6-23 (1) よりも速度が低下します。

このように、同一バンク上に、同一ハーフワープ内の複数スレッドの要素が存在し、ロード／ストアを 2 回以上で行うことをバンクコンフリクト（コンフリクトは衝突という意味）と呼びます。ロード／ストアが 2 回の場合を 2 ウェイのバンクコンフリクトと呼びます。

```
__shared__ float S[1000];                          ①

__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int stride = 1;
  dA[i] = S[stride*threadIdx.x] + 1.0f;   ②
}                          ↑ スレッドID=0,1,・・・,15

int main(void){
    :
  kernel<<<1,16>>>(dA);
    :
}
```

図 3-6-22

- stride = 1: バンクコンフリクトは発生しません。

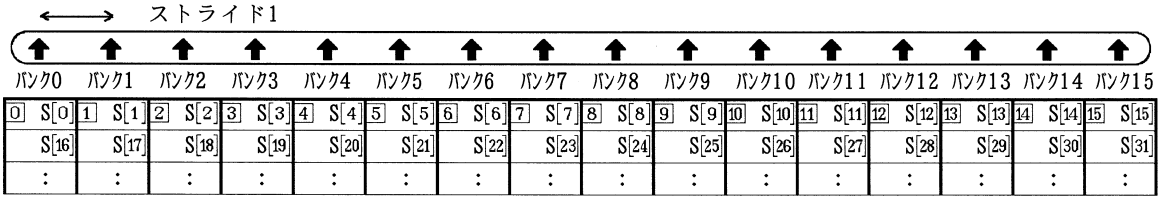


図 3-6-23 (1)

- stride = 2: 2 ウェイ・バンクコンフリクトになります。

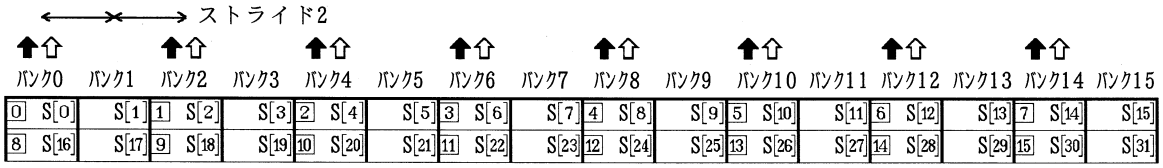


図 3-6-23 (2)

バンクコンフリクトは、ストライドが偶数の場合にのみ発生します。奇数の場合、図 3-6-24 (1) に示すように、各要素は全バンクに散らばるので、バンクコンフリクトは発生しません。

stride = 4, 6, 8, 16（いずれも偶数）とした場合のバンクコンフリクトの様子を図 3-6-24 (2)〜(5) に示します。ストライドが 16 の場合、すべての要素がバンク 0 に集中しており、最も遅い 16 ウェイ・バンクコンフリクトになります。

- stride = 3（または奇数）：ストライドが奇数の場合、バンクコンフリクトは発生しません。

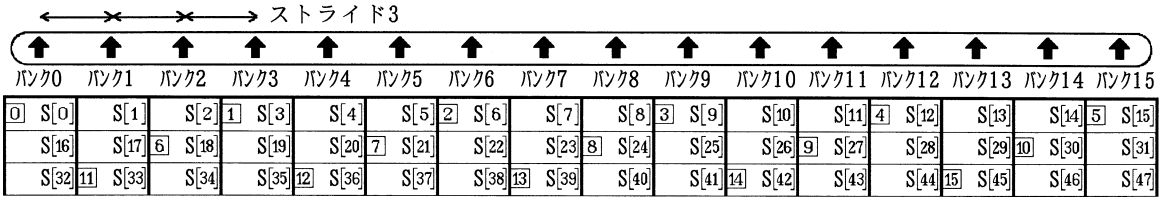


図 3-6-24 (1)

- stride = 4: 4 ウェイ・バンクコンフリクトになります。

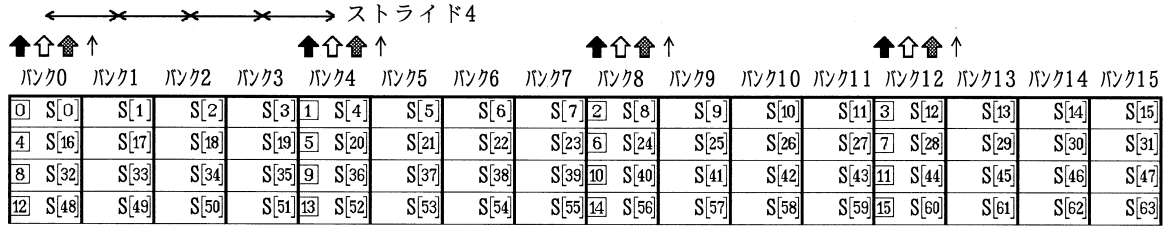


図 3-6-24 (2)

- stride = 6: 6 ウェイでなく、2 ウェイ・バンクコンフリクトになります。

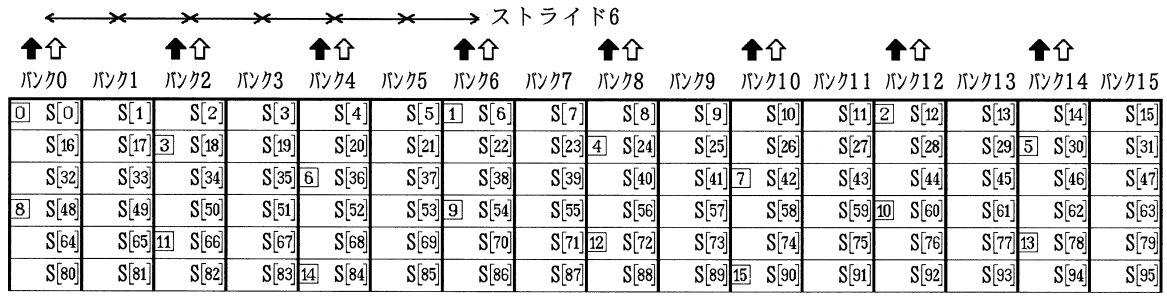


図 3-6-24 (3)

- stride = 8: 8 ウェイ・バンクコンフリクトになります。

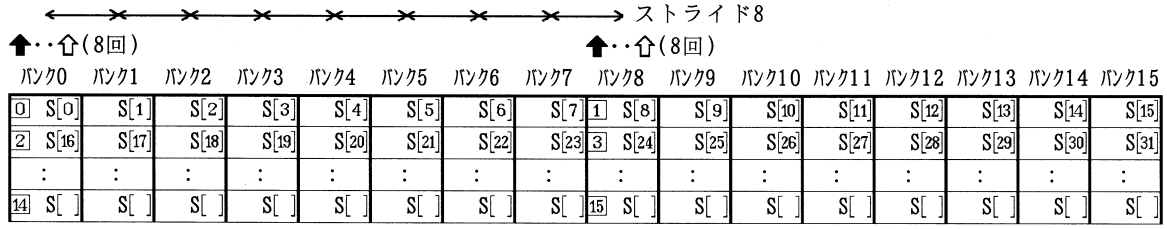


図 3-6-24 (4)

- stride = 16: 16 ウェイ・バンクコンフリクトになります。

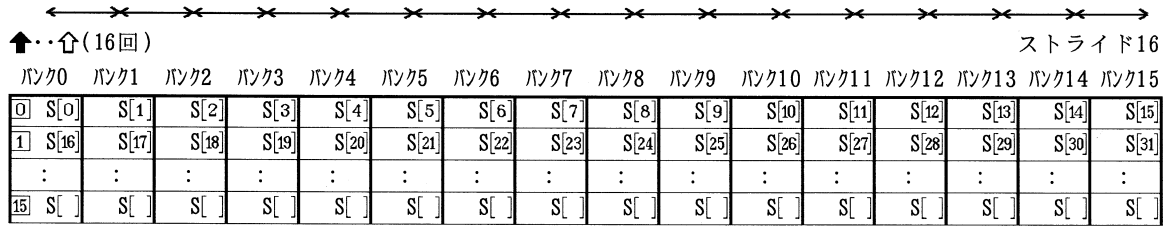


図 3-6-24 (5)

図 3-6-25 (1) に示すように、全スレッドが同一要素 (S[0]) をロードする場合、バンクコンフリクトは起きず、ロードは 1 回で行われます (ストアでは 16 ウェイ・バンクコンフリクトになり、結果が不定になります)。

倍精度 (8 バイト) の要素は、単精度 (4 バイト) の、連続する 2 個の要素で構成されます。図 3-6-25 (2) に示すように、倍精度でストライドが 1 の場合、各要素の左の 4 バイトのロードと、右の 4 バイトのロードのそれぞれで、2 ウェイ・バンクコンフリクトが発生します。

図 3-6-25 (3) に示すように、ストライドが 18 (16 より大きい) の場合は、図 3-6-23 (2) のストライドが 2 の場合と同じになります。つまり、ストライドが 1〜16 のケースと、ストライドが 17〜32、33〜48、...のケースは、同じバンク・コンフリクトが発生します。まとめると、ストライドの値とバンク・コンフリクトのウェイの数の関係は、図 3-6-26 の①, ②のようになります（単精度の場合)。例えば図 3-6-25 (3) はストライドが 18 で、16 で割った余りは 2 なので、図 3-6-26 の〇に示すように、2 ウェイ・バンクコンフリクトになります。

- stride = 0: バンクコンフリクトは発生しません。

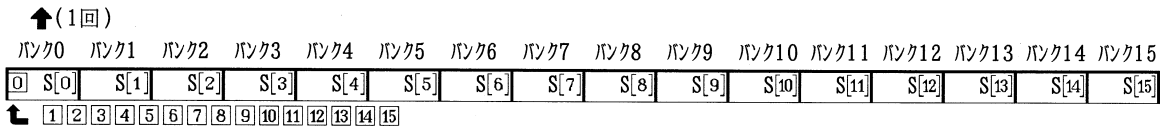


図 3-6-25 (1)

- 倍精度 (8 バイト) で stride = 1: 2 ウェイ・バンクコンフリクトが 2 回発生します。

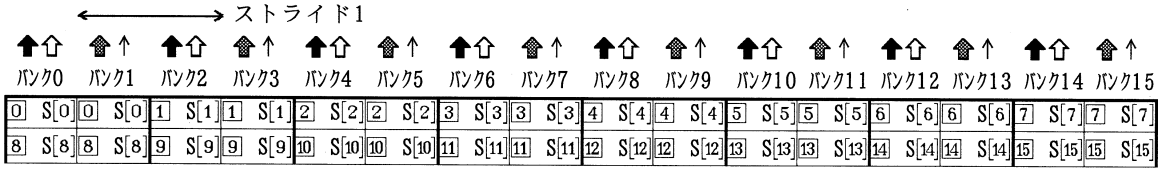


図 3-6-25 (2)

- stride = 18: 2 ウェイ・バンクコンフリクトになります。

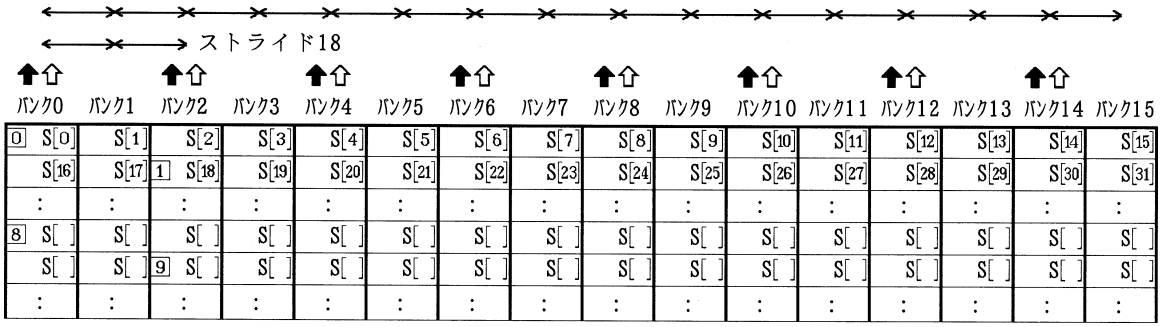


図 3-6-25 (3)

①	ストライドを16で割った余り	1	(2)	3	4	5	6	7	8	9	10	11	12	13	14	15	(注)0
②	nウェイ・バンクコンフリクト	なし	(2)	なし	4	なし	2	なし	8	なし	2	なし	4	なし	2	なし	16
③	warp_serializeの値	0	8	0	24	0	8	0	56	0	8	0	24	0	8	0	120

図 3-6-26 （注）図 3-6-25 (1) の場合は除きます。

プロファイラー (4-5 節参照) を使って、バンクコンフリクトの発生状況を知ることもできます。図 3-6-27 (2) に示すように、ファイル config（名前は任意）に「warp_serialize」を設定し、図 3-6-27 (1) の環境変数を設定してジョブを実行すると、図 3-6-27 (3) に示すファイル log（名前は任意）が作成され、バンクコンフリクトの回数が表示されます。図 3-6-22 のプログラムでテストしたところ、図 3-6-26 の③のように、②と異なる値となりました。②と③の値を比較すると下記の関係になるようですが、真偽は不明です。

$$\text{③ warp\_serialize の値} = (\underline{\text{② } n \text{ ウェイ・バンクコンフリクト}} - 1) \times 8$$

```
export CUDA_PROFILE=1
export CUDA_PROFILE_CONFIG=config
export CUDA_PROFILE_LOG=log
```

図 3-6-27 (1)

```
config  warp_serialize
```

図 3-6-27 (2)

```
log     ～    warp_serialize=[ 16 ]
```

図 3-6-27 (3)

バンクコンフリクト (2 次元配列)

シェアードメモリ上に確保する配列 S が、2 次元の場合について説明します。ブロック内のスレッド数が例えば 2 次元で 16 × 16 の場合、2 次元配列 S は S[16][16] となり、プログラムは図 3-6-28 (1) または図 3-6-29 (1) のようになります。図 3-6-28 (1) ではバンクコンフリクトは発生しませんが、図 3-6-29 (1) では 16 ウェイのバンクコンフリクトが発生し、速度が低下します。この理由と回避方法について説明します。

まずブロック内の 16 × 16 のスレッドを図 3-6-30 に示します。図の横方向が x 方向のスレッド ID (tx=threadIdx.x)、縦方向が y 方向のスレッド ID (ty=threadIdx.y) を表します。スレッドは x 方向の順に並びます (2-4 節参照)。一方メモリ上のデータのロード／ストアは、ハーフワープ (ワープ内の前半、または後半の 16 スレッド) 単位に行われます。従って図 3-6-30 の〓内のスレッドが、1 つ目のハーフワープになります。

図 3-6-28 (2) は行列 S[16][16] を表します。本書では、図の横方向が S[16][16] の下線部の次元を、図の縦方向が S[16][16] の下線部の次元を表します (1-5 節参照)。図 3-6-28 (1) の①より、図 3-6-28 (2) の横方向が tx、縦方向が ty になるので、図 3-6-30 の〓のハーフワープ内の各スレッドは、図 3-6-28 (2) の〓内の各要素をアクセスします。一方図 3-6-29 (2) では、図 3-6-29 (1) の②に示すように tx と ty が①と反対なので、図 3-6-30 の〓のハーフワープ内の各スレッドは、図 3-6-29 (2) の〓内の各要素をアクセスします。

C 言語では、2 次元配列はメモリ上で S[0][0], S[0][1], ... の順 (3-6-28 (2) の矢印の順) に配置されます (1-5 節参照)。従って配列 S は、バンク上では図 3-6-28 (3) のように配置されます (図中の例えば [0][1] は S[0][1] を示します)。図 3-6-28 (2) の〓は、図 3-6-28 (3) の0〜15に示すようにストライド 1 なので、図 3-6-23 (1) と同様にバンクコンフリクトは発生しません。一方図 3-6-29 (2) の〓は、図 3-6-29 (3) の0〜15に示すようにストライド 16 なので、図 3-6-24 (5) と同様に 16 ウェイバンクコンフリクトが発生し、速度は低下します。

```
   :
__shared__ float S[16][16];
int tx = threadIdx.x;
int ty = threadIdx.y;
～ = S[ty][tx] + 1.0f;    ①
   :
```

図 3-6-28 (1)　○

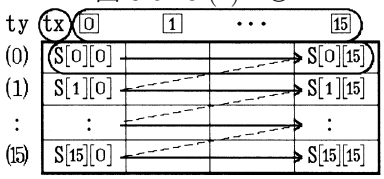


図 3-6-28 (2)　シェアードメモリ S[16][16]

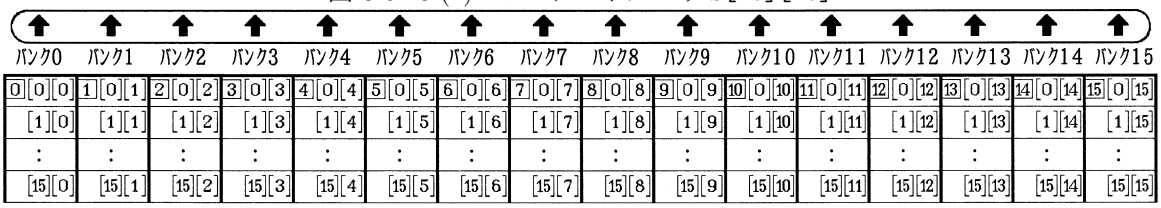


図 3-6-28 (3)

```
  :
__shared__ float S[16][16];
int tx = threadIdx.x;
int ty = threadIdx.y;
～ = S[tx][ty] + 1.0f ;  ②
  :
```

図 3-6-29 (1)　×

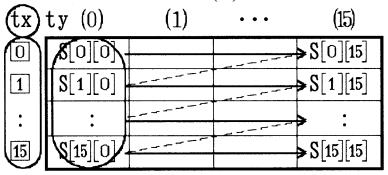


図 3-6-29 (2)　シェアードメモリ S[16][16]

⬆・・⇧(16回)

バンク0	バンク1	バンク2	バンク3	バンク4	バンク5	バンク6	バンク7	バンク8	バンク9	バンク10	バンク11	バンク12	バンク13	バンク14	バンク15
0[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]	[0][8]	[0][9]	[0][10]	[0][11]	[0][12]	[0][13]	[0][14]	[0][15]
1[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[1][6]	[1][7]	[1][8]	[1][9]	[1][10]	[1][11]	[1][12]	[1][13]	[1][14]	[1][15]
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:
15[15][0]	[15][1]	[15][2]	[15][3]	[15][4]	[15][5]	[15][6]	[15][7]	[15][8]	[15][9]	[15][10]	[15][11]	[15][12]	[15][13]	[15][14]	[15][15]

図 3-6-29 (3)

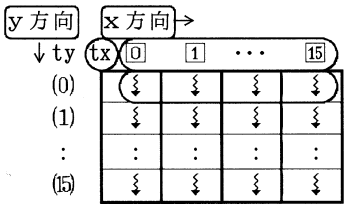


図 3-6-30　スレッド

前ページの図 3-6-29 (1) のバンクコンフリクトの回避方法を説明します。図 3-6-30 (1) (2) に示すように、配列 S の 2 次元目を大きくして奇数にすると、配列 S はバンク内で図 3-6-30 (3) のように配置されます。これによって、ハーフワープ内のスレッド0〜15が担当する要素が、異なるバンクに分散するため、バンクコンフリクトは発生しません。このように、余分な配列要素 (図中の P) を追加することを「パディング」する」(padding: 詰め物をすること) と言います。

```
   :
__shared__ float S[16][17];
int tx = threadIdx.x;
int ty = threadIdx.y;
S[tx][ty] = ～ ;
   :
```

図 3-6-30 (1)

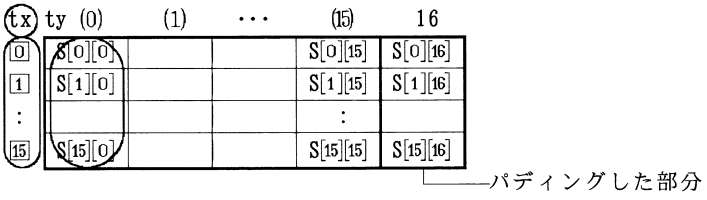


図 3-6-30 (2)　シェアードメモリ S[16][17]

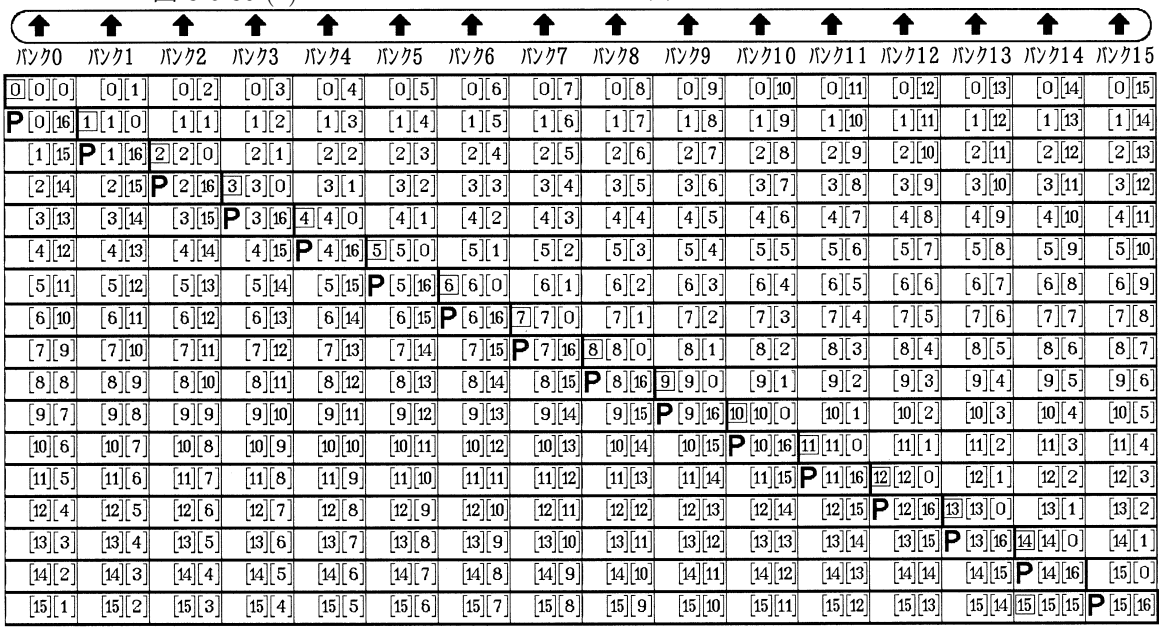


図 3-6-30 (3)

行列の乗算での使用例

行列乗算 $C = AB$ をシェアードメモリを使って計算する例 (8-4 節) の一部を簡単化したプログラムを、図 3-6-31 に示します。①で 16×16 の 2 次元小行列 sA と sB をシェアードメモリに確保し、④で sA と sB の乗算を行います。図 3-6-29 (1) と同様に、行列 sA と sB の 2 次元目の大きさが偶数 (16) で、かつ⑤のループを反復させると、⑥の計算で、配列 sA は 2 次元目が、sB は 1 次元目が反復するので、どちらかの配列でバンクコンフリクトが発生しそうに思われます。ところが実際には、以下で説明するようにバンクコンフリクトは発生しないので、図 3-6-30 (1) のように sA[16][17]，sB[16][17] にする必要はありません。

図 3-6-31 の②，③の指定により、ブロック内の 16×16 のスレッドの ID は図 3-6-32 のようになります。ブロック内の最初のハーフワープに含まれる 16 個のスレッドは、〓に示す ([0], (0))〜([15], (0)) です (以後スレッド[0]〜[15]と略記します)。

図 3-6-31 の⑥で、行列 sA の 2 次元目と sB の 1 次元目の添字にスレッド ID を使用しているので、sA と sB は図 3-6-33 のようになります。⑤のループが反復すると、スレッド[0]〜[15]は、図 3-6-33 の → の行と ↓ の各列の内積を計算します。⑤で k = 0, 1 のときの、⑥でスレッド[0]と[15]がロードする行列 sA と sB 内の要素を、図 3-6-34 (1) に示します (図中の例えば[0]は、スレッド[0]がロードする要素を表します)。

ハーフワープ内の全スレッド ([0]〜[15]) がロードする要素をまとめると、図 3-6-34 (2) になります。まず行列 sA では、ハーフワープ内の全スレッドが●に示す同一要素をロードするため、図 3-6-25 (1) で示したようにバンクコンフリクトは発生しません。また行列 sB では、ハーフワープ内の各スレッドが、〓に示すようにストライド 1 でロードするので、図 3-6-28 (3) と同様にバンクコンフリクトは発生しません。

```
__global__ void kernel(){
  __shared__ float sA[16][16],sB[16][16]; ①
  int j = threadIdx.x;                    ②
  int i = threadIdx.y;                    ③
    :
  float sum = 0.0f;
  for (int k=0;k<16;k++){                 ⑤
    sum = sum + sA[i][k]*sB[k][j];        ⑥   ④
  }
    :
```

図 3-6-31

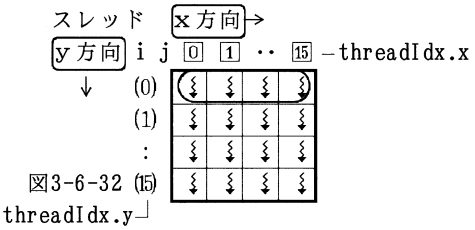


図 3-6-32

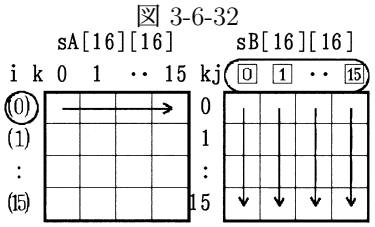


図 3-6-33

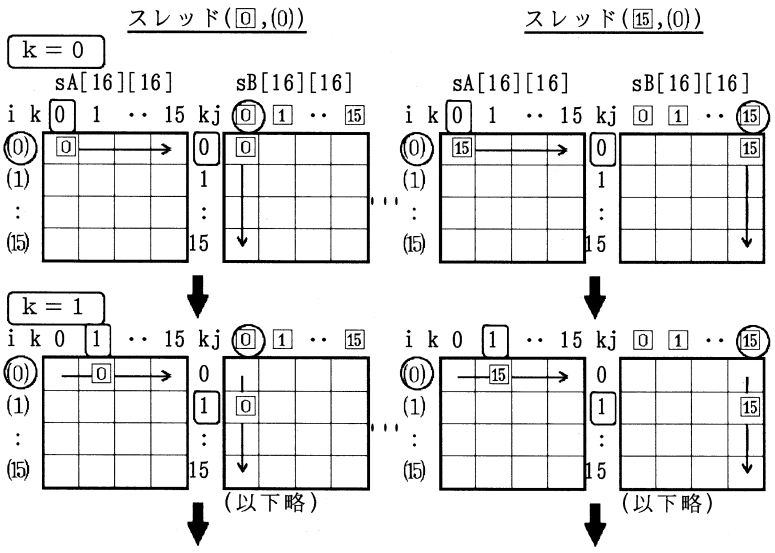


図 3-6-34 (1)

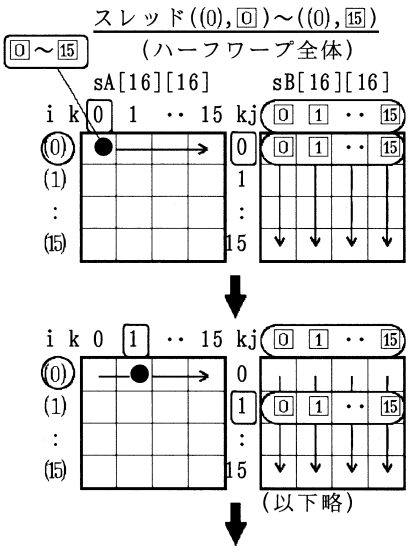


図 3-6-34 (2)

シェアードメモリ上の変数／配列の存在する期間

図 3-6-35 (1) では、①でグローバルメモリ上に変数 dD を確保します。3-4 節で説明したように、変数 dD は、プログラムの開始時にグローバルメモリ上に確保され、プログラムが終了するまで存在します。

図 3-6-35 (1) を実行すると、④でカーネル関数 kernel が呼ばれ、②で変数 dD に値を設定します。次に⑤でカーネル関数 kernel2 が呼ばれ、③で、②で設定した dD の値を参照します。この様子を図 3-6-36 (1) に示します。このように、④と⑤のカーネル関数間で、グローバルメモリ上の変数 dD の値を受け渡すことができます。

一方図 3-6-35 (2) では、1でシェアードメモリ上に変数 dS を確保します。本節で説明したように、変数 dS は、ブロックがストリーミングマルチプロセッサー上に配置された時点でシェアードメモリ上に確保され、そのブロック内の全スレッドが処理を終了したら解放（消滅）されます。

図 3-6-35 (2) を実行すると、4で (ブロック数が 1 で) カーネル関数 kernel が呼ばれ、2で変数 dS に値を設定します。前述のように、この変数 dS は、4が終了した時点で消滅しています。従って、次に5で (ブロック数が 1 で) カーネル関数 kernel2 が呼ばれ、3で参照する dS には、2で設定した値は入っていません。この様子を図 3-6-36 (2) に示します。

このように、複数のカーネル関数間で、シェアードメモリ上の変数 dS の値を受け渡すことはできないので注意して下さい（同一カーネル関数内の異なるブロック間でも、受け渡すことはできません）。図 3-6-36 (3) に示すように、同一のカーネルを複数実行した場合も同様に、変数 dS の値を受け渡すことはできません。

```
__device__ float dD;                    ①
__global__ void kernel(){
  dD = 99.0f;                           ②
}
__global__ void kernel2(float *dA){
  dA[0] = dD;                           ③
}
int main(void){
    :
  kernel <<<1,1>>>();                   ④
  kernel2<<<1,1>>>(dA);                 ⑤
    :
```

図 3-6-35 (1)

```
__shared__ float dS;                    1
__global__ void kernel(){
  dS = 99.0f;                           2
}
__global__ void kernel2(float *dA){
  dA[0] = dS;                           3
}
int main(void){
    :
  kernel <<<1,1>>>();                   4
  kernel2<<<1,1>>>(dA);                 5
    :
```

図 3-6-35 (2)

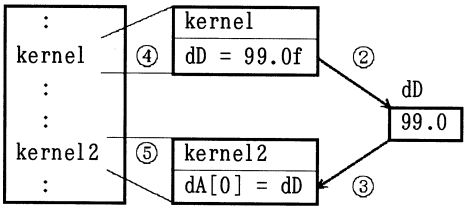


図 3-6-36 (1)

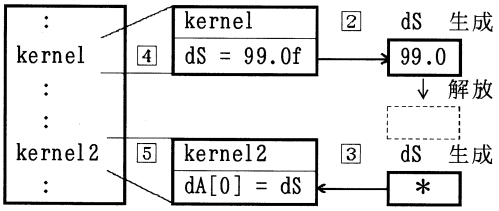


図 3-6-36 (2)

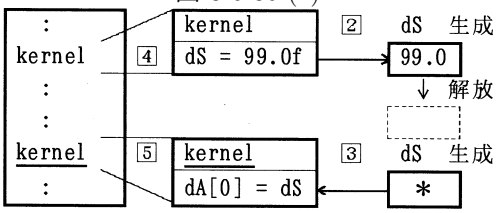


図 3-6-36 (3)

3-7　カーネル関数の仮引数

関数を呼び出す側の引数を実引数、関数側の引数を仮引数と呼びます。C 言語では、実引数に（配列、ポインター以外の）変数を指定した場合、データの受渡しは値渡しです (Fortran では参照渡しです)。値渡しの場合、図 3-7-1 に示すように、実引数と仮引数はメモリ上の別の領域に存在し、関数が呼ばれたときに、実引数の内容が仮引数にコピーされます（実引数と仮引数は同一名でも構いません）。

CUDA で、ホスト側のプログラムからカーネル関数を呼び出す場合も同じで、図 3-7-2 に示すように、カーネル関数が呼ばれたときに、ホスト側のメモリ上にある実引数の内容が、デバイス側のメモリ上にある仮引数にコピーされます。したがって、cudaMemcpy を使って明示的にコピーする必要はありません。

仮引数は、図 3-7-2 の dI に示すように、ブロックごとに 1 つ、シェアードメモリ上に存在します。カーネル関数内で仮引数の dI を更新した場合は、dI とは別に、各スレッドごとに dI が、レジスターに確保されます。

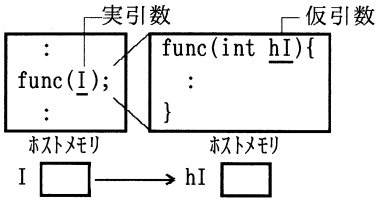


図 3-7-1

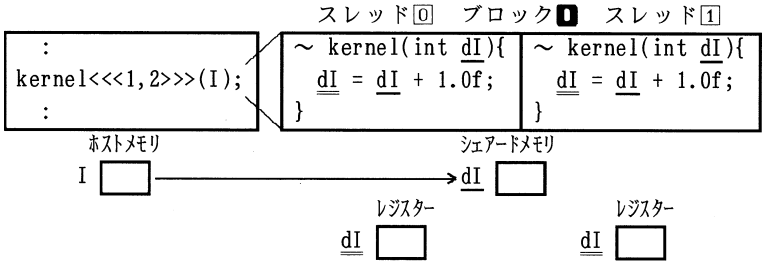


図 3-7-2

カーネル関数の仮引数で指定した変数の特性を以下に示します (3-1 節参照)。

- 作成される場所：ブロックごとに、ストリーミング・マルチプロセッサー上のシェアードメモリ（オンチップ：高速）上に作成されます。
- アクセスできるスレッドの範囲：当該ブロック内の全スレッドからアクセス（参照のみ）可能です。
- 存在する期間：当該ブロックがストリーミングマルチプロセッサー上に存在している間、存在します。
- 容量：ブロックあたり 256 バイト (単精度だと 64 個) です (「CUDA C Programming Guide」の B.1.4 節参照)。

図 3-7-3 の①の下線部のオプションをつけてコンパイルすると、カーネル関数が使用するメモリの種類と容量が表示されます。smem はシェアードメモリを示し、1 ブロックあたりの容量が表示されます。図 3-7-4 は仮引数が整数 1 個の場合で、②に示すように 4 バイトの領域がシェアードメモリに確保されます。「+16」の部分は他の用途で使用されていると思われますが、詳細は不明です。

図 3-7-5 は仮引数が整数 2 個の場合で、③に示すように 8 バイトの領域がシェアードメモリに確保されます。

図 3-7-6 は変数を整数 64 個の構造体にした場合で、④に示すように $64 \times 4 = 256$ バイトの領域がシェアードメモリに確保されます。図 3-7-6 で X[65] 以上にすると、前述の 256 バイトの制限を越えるので、⑤に示すようにコンパイルエラーが表示されます。

```
$ nvcc -Xptxas -v -arch=sm_13 -c test.cu ↵                          ①
ptxas info    : Compiling entry function '_Z6kerneli' for 'sm_13'
ptxas info    : Used 0 registers, 4+16 bytes smem                   ②  ← 図3-7-4
ptxas info    : Used 0 registers, 8+16 bytes smem                   ③  ← 図3-7-5
ptxas info    : Used 0 registers, 256+16 bytes smem                 ④  ← 図3-7-6でX[64]
/tmp/tmpxft_000028ee_00000000-7_test.cpp3.i(0):                     ⑤  ← 図3-7-6でX[65]
Error: Formal parameter space overflowed in function _Z6kernel4data ⑤
```

図 3-7-3

```
__global__ void kernel(int dI){
}
```

図 3-7-4

```
__global__ void kernel(int dI,int dJ){
}
```

図 3-7-5

```
typedef struct data{
  int X[64];
} DATA;
__global__ void kernel(DATA dX){
}
```

図 3-7-6

3-8　カーネル関数内で宣言したローカル変数

カーネル関数内で宣言した、ローカル変数／配列の特性を以下に示します (3-1 節参照)。

- 作成される場所：スレッドごとに、ストリーミング・マルチプロセッサー上のレジスター（オンチップ：高速）に作成されます。レジスターを使いきった場合は、デバイスメモリ内のローカルメモリ（オフチップ：低速）上に作成されます。
- アクセスできるスレッドの範囲：当該スレッドのみからアクセスすることができます。
- 存在する期間：当該スレッドが実行を開始してから終了するまでの間、存在します。
- 容量：ストリーミングマルチプロセッサーあたり利用可能なレジスター数は、16384 個 (1 つのレジスターは 4 バイト) です。1 スレッドで使用できるレジスター数は不明ですが、下記テストでは 50 個程度でした。

図 3-8-1 のプログラムを、図 3-8-3 の①の下線部のオプションをつけてコンパイルすると、②に示すように、スレッドあたりの使用レジスター数が 4 個と表示されます。このうち 1 個は、図 3-8-1 の下線部に示すローカル変数 dL で、他の 3 つは、内部的に作業域として使用されるレジスターです (詳細はアセンブラリスト (2-7 節) を参照して下さい)。

カーネル関数内で複数のローカル変数を使用している場合、各ローカル変数にレジスターが 1 つずつ固定される訳ではなく、ある変数の使用がプログラム内で終了したら、その変数に割り当てられたレジスターは他の変数に割り当てられ、使い回しされます。

図 3-8-2 では、ローカル変数として (変数 i と) 配列 dL[N] (N = 40) を使用しています。この場合、図 3-8-3 の③に示すように、使用レジスター数は全部で 50 個になりました。次に N = 41 にしたところ、レジスターを使いきり、④に示すように、デバイスメモリ上のローカルメモリ (lmem) が使用されました。このような場合、ローカルメモリは低速なので、可能であれば、使用するレジスター数を減らすようにプログラムを修正し、ローカル変数の使用はなるべく避けるようにして下さい。

N をさらに増やしたところ、⑤に示すように、N = 4096 まではローカルメモリが使用されますが、N = 4097 以上では、⑥に示すようにコンパイルエラーとなりました。これは、スレッドあたりのローカルメモリの上限である 16 KB の制限 (「CUDA C Programming Guide」の G.1 節参照) を越えたためだと思われます。

```
__device__ int dD;
__global__ void kernel(){
  int dL;
  dL = dD + 1;
  dD = dD*dL;
}
```

図 3-8-1

```
#define N (40)
__device__ int dD[N];
__global__ void kernel(){
  int i,dL[N];
  for(i=0;i<N;i++){
    dL[i] = dD[i] + i;
  }
  for(i=0;i<N;i++){
    dD[i] = dL[i];
  }
}
```

図 3-8-2

```
$ nvcc -Xptxas -v -arch=sm_13 -c test.cu ↵                              ①
ptxas info    : Compiling entry function '_Z6kernelv' for 'sm_13'
ptxas info    : Used 4 registers, 4 bytes cmem[14]                      ②   ← 図3-8-1
ptxas info    : Used 50 registers, 4 bytes cmem[14]                     ③   ← 図3-8-2でN=40
ptxas info    : Used 7 registers, 164+0 bytes lmem, 4 bytes cmem[1],    ④   ← 図3-8-2でN=41
                4 bytes cmem[14]
ptxas info    : Used 3 registers, 16384+0 bytes lmem, 4 bytes cmem[1],  ⑤   ← 図3-8-2でN=4096
                4 bytes cmem[14]
ptxas info    : Used 3 registers, 16388+0 bytes lmem, 4 bytes cmem[1],
                4 bytes cmem[14]
ptxas error   : Entry function '_Z6kernelv' uses too much local data    ⑥   ← 図3-8-2でN=4097
                (0x4004 bytes, 0x4000 max)
```

図 3-8-3

第4章　基本編（その他）

本章では、CUDA プログラミングの基本項目のうち、前章までで説明していない項目を説明します。

4-1　並列化と同期

本節では、同期に関連する項目をまとめます。CUDA 化したプログラムで、同期を取るべき箇所で同期を取るのを忘れた場合、例えば、同一プログラムを 100 回実行し、99 回は計算結果が正しく、1 回だけ計算結果がおかしくなる、などの現象が起こる可能性があります (99 回計算結果が正しくても、プログラムとしては間違いなのは言うまでもありません)。このため、同期の指定については十分に注意する必要があります。なお、本節で紹介する方法以外に、__threadfence() という CUDA 関数が提供されていますが、使用方法がよく分からないので、説明は省略します。

同期の取り忘れとプログラムの正当性の保証

図 4-1-1 (1) のプログラムを、間違えて図 4-1-1 (2) にしてしまった場合のように、通常のプログラムのバグは、実行すると必ず計算結果がおかしくなり、バグに気が付きます。

```
m = 2;
printf("%d¥n",m);
```

図 4-1-1 (1)　○

```
m = 1;
printf("%d¥n",m);
```

図 4-1-1 (2)　×

CUDA 化した図 4-1-2 (1) のプログラムで、カーネル関数を 2 スレッドで実行した場合、__syncthreads() で同期を取っているため、①と②が両方終了した後に③が実行され、sum = 1.0 + 2.0 = 3.0 となります。一方、__syncthreads() を指定し忘れた図 4-1-2 (2) の場合、①, ②, ③の順番に実行された場合は計算結果はおかしくなりますが (sum = 1.0 + 未定義)、図 4-1-2 (3) の①, ②, ③の順番に実行された場合は、図 4-1-2 (1) と同じ実行順序なので、計算結果は（たまたま）正しくなります。

このように、同期を取り忘れた場合、実行しても計算結果がおかしくならず、バグに気付かないことがあります。同期を取り忘れたのに計算結果がおかしくならない確率は、プログラムのロジックや、データの規模によって異なります（例えばテスト用の小規模データだと結果が（たまたま）正しかったのが、本番用の大規模データで実行したら結果がおかしくなるなど)。

計算結果がおかしくならない確率が 1% 程度であれば、テストするとほぼ計算結果がおかしくなり、バグに気付きますが、計算結果がおかしくならない確率が 99% だと、テストしても 100 回のうち 99 回は計算結果が正しくなるので、バグに気付かない可能性が高くなります。言いかえると、テストを何万回行っても、同期を取り忘れたバグが絶対に含まれていないとは断言できません。

従って、CUDA 化したプログラムのどこで同期を取る必要があるのかを、ユーザーが自分の責任で、プログラムのロジックから慎重に判断する必要があります。

一方、同期は時間がかかるので、同期を取る必要がない部分でむやみに同期を取ると、遅くなります。

スレッド0	スレッド1
① dS[0] = 1.0; __syncthreads(); ③ sum = dS[0] + dS[1]; 3.0　1.0　2.0	② dS[1] = 2.0; __syncthreads();

図 4-1-2 (1)　バグなし；結果が正しい

スレッド0	スレッド1
① dS[0] = 1.0; ② sum = dS[0] + dS[1]; ?　1.0　未定義	③ dS[1] = 2.0;

図 4-1-2 (2)　バグあり；結果がおかしい

スレッド0	スレッド1
① dS[0] = 1.0; ③ sum = dS[0] + dS[1]; 3.0　1.0　2.0	② dS[1] = 2.0;

図 4-1-2 (3)　バグあり；(たまたま) 結果が正しい

同一ワープ内の全スレッド (32 スレッド) の同期

まず同一ワープ内の全スレッド (32 スレッド) の同期について説明します。図 4-1-3 (1) に示すように、ブロック数 1、ブロック内のスレッド数 32 (= ワープ数 1) でカーネル関数を実行し、同一ワープ内の 32 スレッド全てが①のステートメントを完了した後、②のステートメントを実行したいとします。

図 4-1-3 (1) のプログラムを実行すると、2-4 節で説明したように、図 4-1-3 (2) の①を 8 スレッドずつ連続に実行し、その後②を 8 スレッドすつ連続に実行します (③で一度中断する場合もあります)。従って①を全スレッドが終了した後、②の実行に入るので、上記の下線部のように動作します。まとまると、同一ワープ内の連続した 32 スレッド内では、①と②の間で、__syncthreads() を使用して明示的に同期を取らなくても、自動的に同期が取られます (「CUDA C Programming Guide」5.4.3 節参照)。

この性質は、以下のような場合に適用することができます。

- 同期を使用する計算で、ブロックあたり 32 スレッド (1 ワープ) で実行すれば、__syncthreads() で同期を取る必要がなくなり、同期のオーバーヘッドを減らすことができます。ただしブロックあたり 32 スレッドの場合、6-1 節で説明するように、1 つのストリーミング・マルチプロセッサーあたりに同時に存在できるワープ数が少なくなり、グローバルメモリーのロード／ストアの時間が増加する可能性があります。
- 同期を使用する計算で、計算を担当するスレッド数が次第に減少する場合 (例えば 8-3 節の合計を求める計算など)、ブロックあたりのスレッド数が 32 以下になったら __syncthreads() で同期を取るのをやめるようにすれば、同期のオーバーヘッドを減らすことができます。

逆に、①と②の間で明示的に同期を取るべきなのに取り忘れたプログラムを、ブロックあたり 32 スレッド以下でテストしたため自動的に同期が取られ、計算結果が（たまたま）合っていたというケースも考えられます。この場合は、スレッド数を多くすると計算結果がおかしくなり、バグが表に現れます。従って、同期を使用するプログラムのテストは、ある程度たくさんのブロック数、スレッド数で行うようにして下さい。

なお、次ページの説明では、図 4-1-3 (2) を図 4-1-3 (3) のように簡略化して表します。

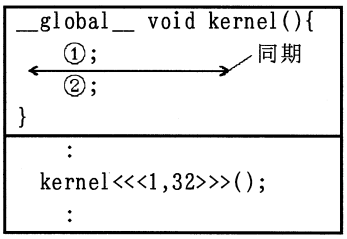


図 4-1-3 (1)

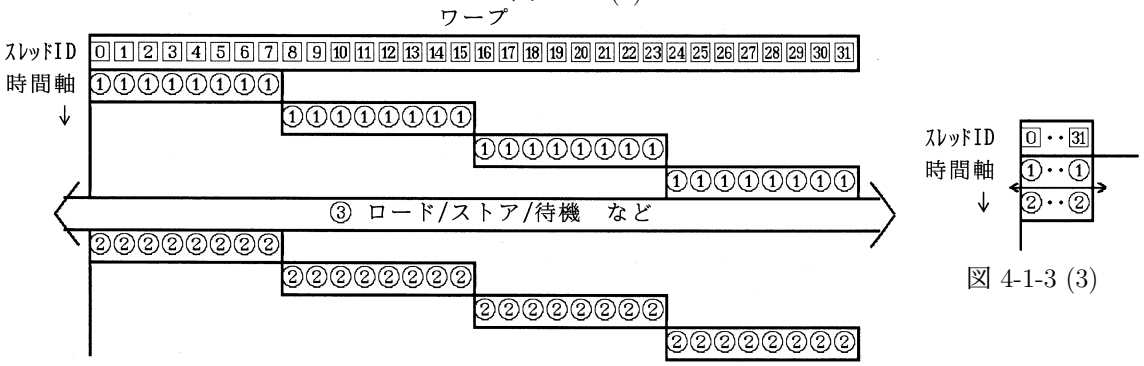


図 4-1-3 (2)

図 4-1-3 (3)

同一ブロック内の全スレッドの同期

図 4-1-4 (1) に示すように、ブロック数 2、ブロック内のスレッド数 64 (= ワープ数 2) でカーネル関数を実行し、各ブロックで、同一ブロック内の 64 スレッド全てが①のステートメントを完了した後、②のステートメントを実行したいとします (シェアードメモリ (3-6 節参照) を使用する場合にこの状況が発生します)。

図 4-1-4 (1) では明示的に同期を取っていません。この場合の動作例を図 4-1-5 (1) に示します。ブロック ID = 0 では、0...63の全スレッドが①を実行した後、0...63のスレッドが②を実行しており、（たまたま）上記のように動作しています。一方ブロック ID = 1 では、32...63のスレッドが②を実行した後、0...31のスレッドが①を実行しており、上記のように動作していません。

この場合、図 4-1-4 (2) に示すように、①と②の間に __syncthreads() を挿入すると、図 4-1-5 (2) に示すように、各ブロックで、0...63の全スレッドが①を実行した後②を実行するので、上記のように動作します。

なお、__syncthreads() を使用した場合、1 つのストリーミング・マルチプロセッサーあたり、複数のブロック数が推奨されます (詳細は 6-1 節参照)。

全ブロック内の全スレッドの同期

図 4-1-5 (3) に示すように、全ブロック内の全スレッドが①のステートメントを完了した後、②のステートメントを実行したい場合、図 4-1-4 (3) に示すように、①と②を別のカーネル関数にする必要があります。kernel1 が終了した後、kernel2 の実行が開始するので、同期が取られたのと同じ動作になります。

```
__global__ void kernel(){
    ①;
    ②;
}
    :
kernel<<<2,64>>>();
    :
```

同期

図 4-1-4 (1)

```
__global__ void kernel(){
    ①;
  __syncthreads();
    ②;
}
    :
kernel<<<2,64>>>();
    :
```

図 4-1-4 (2)

```
__global__ void kernel1(){
    ①;
}
__global__ void kernel2(){
    ②;
}
    :
kernel1<<<2,64>>>();
kernel2<<<2,64>>>();
    :
```

図 4-1-4 (3)

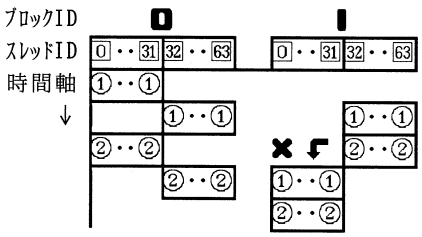


図 4-1-5 (1)

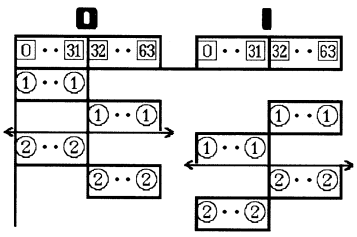


図 4-1-5 (2)

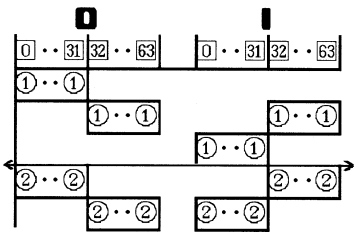


図 4-1-5 (3)

__syncthreads() に関する補足 (1)

- 図 4-1-6 (1) では、②で指定した 1 ブロック 32 スレッドのうち、ID が 0〜19 のスレッドのみが①で同期を取っていますが、このような使用方法は誤りです (「CUDA C Programming Guide」B.6 節参照)。同一ブロック内の全スレッド (本例ではスレッド ID 0〜31) が、同一の __syncthreads() で同期を取る必要があります。
- 図 4-1-6 (2) では、同一ブロック内の各スレッドが、③と④のいずれかの __syncthreads() で同期を取っていますが、このような使用方法は誤りです。図 4-1-6 (3) の⑤に示すように、同一ブロック内の各スレッドが、同一の __syncthreads() で同期を取るようにして下さい。

```
__global__ void kernel(){
  if(threadIdx.x<20){
    :
    __syncthreads();  ①
  }
    :
    :
kernel<<<1,32>>>(); ②
    :
```

図 4-1-6 (1)　×　誤り

```
    :
  if(～){
    :
    __syncthreads(); ③
  }else{
    :
    __syncthreads(); ④
  }
    :
```

図 4-1-6 (2)　×　誤り

```
    :
  if(～){
    :
  }else{
    :
  }
  __syncthreads(); ⑤
    :
```

図 4-1-6 (3)　○　正しい

__syncthreads() に関する補足 (2)

図 4-1-7 のプログラムを、⑩に示すように 1 ブロック、ブロックあたり 4 スレッドで実行するとします。実際には 32 スレッドより多い場合を想定しますが (32 スレッド以下だと、図 4-1-3 (1) (2) に示したように自動的に同期が取られるため)、ここでは説明を簡単にするため、4 スレッドとします。

- ①のループは 2 反復します。1 反復目では、図 4-1-8 (1) の②に示すように、各スレッドは配列 dA の自分が担当する■，●の要素を、シェアードメモリ上の配列 dS にロードします。
- 全スレッドが②のロードを完了してから、④を実行するようにするため、③でブロック内の全スレッドが同期を取ります。
- ④で各スレッドは、dS[0]～dS[3] 内の■，●を使用して計算を行います。
- 全スレッドが④を完了してから、ループの 2 反復目の⑥を実行する必要があるため、⑤でブロック内の全スレッドが同期を取ります（詳細は後述します）。
- ①のループが 2 反復目になり、図 4-1-8 (2) に示すように、⑥，⑦，⑧，⑨で☆，△の処理が同様に行われます。

③の同期が必要なのは明らかです。ここでは⑤の同期の必要性について説明します。1 つのワープには連続する 32 スレッドが含まれますが、説明を簡単にするため、1 ワープは 2 スレッドで、1 つ目のワープにはスレッド ID 0，1が含まれ、2 つ目のワープにはスレッド ID 2，3が含まれるとします (図中の〓が 1 つのワープを表します)。

⑤の同期を指定しなかった場合、どうなるかを説明します。図 4-1-8 (1) で、例えばスレッド ID 0，1の所属するワープが先に④を終了し、スレッド ID 2，3はまだ④を実行しているとします。図 4-1-8 (3) に示すように、スレッド ID 0，1はループの 2 反復目の⑥で☆を dS[0]，dS[1] にロードします。このためスレッド ID 2，3は④で、■でなく☆を使って計算してしまい、結果がおかしくなります。

以上より、図 4-1-7 のように、カーネル関数内にループがあり、その中で __syncthreads() を使用して同期を取る場合、1 回でなく 2 回同期を取る必要がある場合があるので注意して下さい。

```
__global__ void kernel(*dA){
    :
  __shared__ float dS[4];
 for(j=0;j<8;j+=4){                    ①
   dS[threadIdx.x] = dA[j+threadIdx.x];  ②⑥
   __syncthreads();                    ③⑦
   各スレッドはdS[0]～dS[3]を使用して計算 ④⑧
   __syncthreads();                    ⑤⑨
 }
    :
```

```
    :
kernel<<<1,4>>>();                     ⑩
    :
```

図 4-1-7

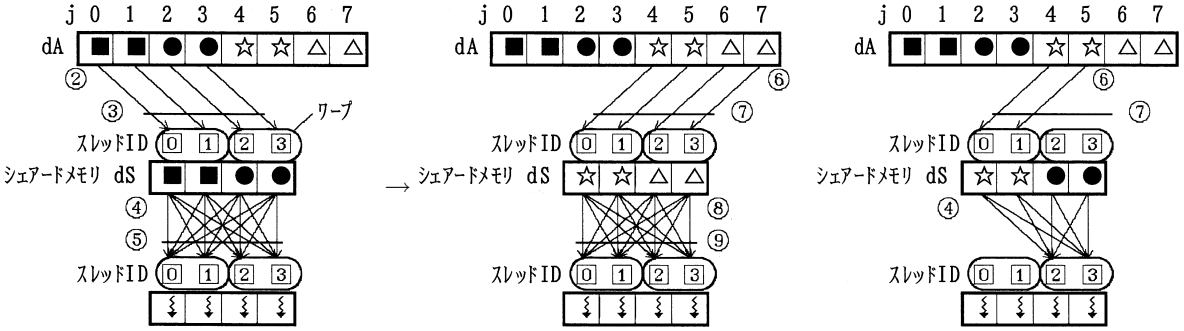


図 4-1-8 (1)　図 4-1-8 (2)　図 4-1-8 (3)

cudaThreadSynchronize()

3-2 節で説明したように、図 4-1-9 (1) の①のカーネル関数は非同期関数なので、①をコールすると、ホスト側のプログラムは、ただちに（待機せずに）次の②の処理を実行します。そして、（もし必要であれば）③で同期を取るための CUDA 関数 cudaThreadsSynchronize() を実行すると、ホストプログラムは③で待機し、③より前にコールされた全ての CUDA 関数とカーネル関数 (本例では①) が終了したら、④の実行を開始します。③の同期は、カーネル関数内でエラーが発生したかどうかをチェックする場合 (4-2 節参照) や、経過時間の測定 (4-4 節参照) などに用いられます。

図 4-1-9 (2) の1のように、非同期関数の CUDA 関数 (例えば cudaMemcpyAsync: 6-3-1 節参照) の場合も、（もし必要であれば）3で同期を取ります。3の同期は、ホスト側プログラムの計算と、ホストとデバイス間のコピーをオーバーラップさせて、実行時間を短縮する場合などに用いられます (6-3-1 節参照)。

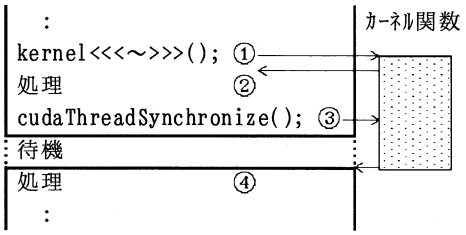


図 4-1-9 (1)

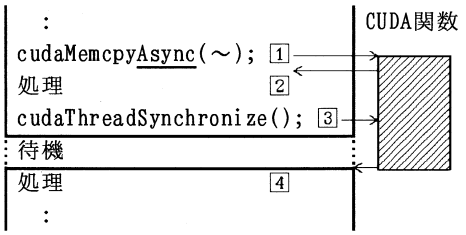


図 4-1-9 (2)

連続した 2 つのループの依存関係

図 4-1-10 (1) の 2 つのループはお互いに依存関係がないので、図 4-1-10 (2) のようにループを合体することができます。図 4-1-10 (1) の状態で CUDA 化すると図 4-1-11 (1) となり、図 4-1-10 (2) の状態で CUDA 化すると図 4-1-11 (2) となります。図 4-1-11 (2) の方がカーネル関数を呼び出す回数が少ないため、オーバーヘッドも少なくなります。

```
    :
for(i=0;i<8;i++){
  A[i] = A[i] + 1.0f;
}
for(i=0;i<8;i++){
  B[i] = B[i] + 2.0f;
}
    :
```

図 4-1-10 (1)

```
    :
for(i=0;i<8;i++){
  A[i] = A[i] + 1.0f;
  B[i] = B[i] + 2.0f;
}
    :
```

図 4-1-10 (2)

```
__global__ void kernel1(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;
}
__global__ void kernel2(float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dB[i] = dB[i] + 2.0f;
}
    :
  kernel1<<<2,4>>>(dA);
  kernel2<<<2,4>>>(dB);
    :
```

図 4-1-11 (1)

```
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;
  dB[i] = dB[i] + 2.0f;
}
    :
  kernel<<<2,4>>>(dA,dB);
    :
```

図 4-1-11 (2)

次に、図 4-1-12 (1) の 2 つのループについて検討します (配列 A は元々ゼロクリアされているとします)。このループを実行すると、図 4-1-13 (1) に示すように、①の設定が全て終了した後、②が実行されます。一方、2 つのループを合体した図 4-1-12 (2)（間違い）を実行すると、図 4-1-13 (2) の (1), (2), (3)...の順に処理が行われ、最終的に図 4-1-13 (3) となり、図 4-1-13 (1) と計算結果が変わってしまいます。つまり、図 4-1-12 (1) のように、2 つのループ間に下線に示す依存関係がある場合、図 4-1-12 (2) のように合体することはできません。

図 4-1-12 (1) を CUDA 化した図 4-1-14 (1) の動作を図 4-1-15 (1) に示します。図 4-1-14 (1) のようにカーネル関数を 2 回呼び出した場合、前述の「■　全ブロック内の全スレッドの同期」で説明したように、全ブロック内の全スレッドで同期が取られます。従って、ブロック ID 0 の全スレッドが図 4-1-15 (1) の [1] を、ブロック ID 1 の全スレッドが [2] を終了した後、[3], [4] が処理され、図 4-1-13 (1) と同じ（正しい）結果が得られます。

一方、図 4-1-14 (2)（間違い）の状態で CUDA 化した図 4-1-14 (2) を実行した場合、たまたま図 4-1-15 (1) の順番で実行された場合は結果が正しくなり、図 4-1-15 (2) のようにブロック 0 が [1], [2] を終了した後、ブロック 1 が [3], [4] を実行すると，×の部分で結果がおかしくなります (配列 dA は元々ゼロクリアされているとします)。

以上より、図 4-1-12 (1) (2) のように、依存関係のある 2 つのループを 1 つにして CUDA 化した場合、偶然結果が正しくなり（ただしプログラムは間違い）、バグに気付かない可能性があるので注意して下さい。

```
    :
for(i=0;i<8;i++){
  A[i] = (float)i;          ①
}
for(i=1;i<7;i++){
  B[i] = A[i-1] + A[i+1]; ②
}
    :
```

図 4-1-12 (1)

```
    :
for(i=0;i<8;i++){
                  A[i] = (float)i;          (1)(2)(4)
  if(1<=i && i<7) B[i] = A[i-1] + A[i+1];   (3)(5)
}
    :
```

図 4-1-12 (2)　×　間違い

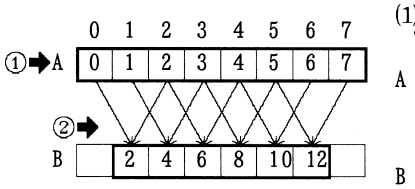


図 4-1-13 (1)

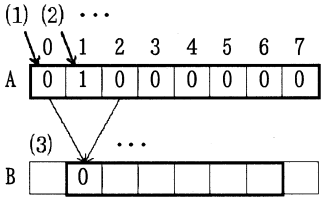


図 4-1-13 (2)　×

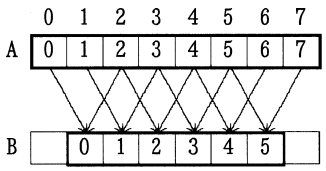


図 4-1-13 (3)　×

```
__global__ void kernel1(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
                  dA[i] = (float)i;
}
__global__ void kernel2(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if(1<=i && i<7) dB[i] = dA[i-1] + dA[i+1];
}
    :
  kernel1<<<2,4>>>(dA);
  kernel2<<<2,4>>>(dA,dB);
    :
```

図 4-1-14 (1)

```
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
                  dA[i] = (float)i;
  if(1<=i && i<7) dB[i] = dA[i-1] + dA[i+1];
}
    :
  kernel<<<2,4>>>(dA,dB);
    :
```

図 4-1-14 (2)　×　間違い

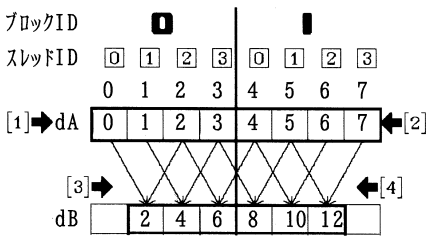


図 4-1-15 (1)

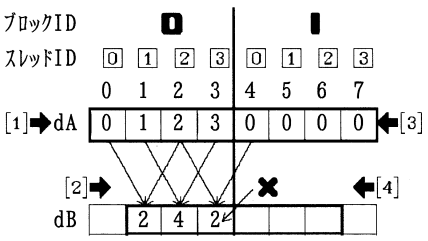


図 4-1-15 (2)

アトミック関数

図 4-1-16 (1) に示すように、グローバルメモリ上の関数 icount に「0」が入っていて、スレッド0と1がそれぞれ 1 を加算し、icount を最終的に「2」にしたいとします。ところが図 4-1-16 (1) (2) の①〜⑥の順に動作した場合、icount は「1」になってしまいます。このように、並列計算では、複数のスレッドがほぼ同時に同じ変数に加算を行うと、タイミングによって結果がおかしくなることがあります (5-2 節参照)。

CUDA では、排他制御を行うアトミック関数が提供されています。図 4-1-17 (1) のように加算のアトミック関数 atomicAdd を使用した場合、図 4-1-17 (2) に示すように、スレッド0が atomicAdd の実行を開始した直後にスレッド1が atomicAdd を実行すると、スレッド1は待機状態になります。そしてスレッド0が加算を終了したら、スレッド1の待機状態が解除され、加算を開始します。

プログラム例を図 4-1-18 に示します。(1) でグローバルメモリ上の変数 icount に初期値「0」を設定します。(2) を実行すると、加算前の icount の値が変数 i に代入された後、icount に「1」が加算されます。試しに、(3), (4) で、i (= 0, 1, 2, ...) 番目の加算を行ったブロック ID とスレッド ID を記録して (6) で書き出すようにし、(5) に示すように 10 ブロック、1 ブロックあたり 4 スレッドで実行したところ、図 4-1-19 の⓪, ①, ... の順番に (2) が実行されていました（実行順序は毎回変わります)。なお、atomicAdd に指定する icount は、シェアードメモリ上の変数も指定可能で、6-2 節の「方法 3」の cudaHostAllocMapped で指定した変数は指定できません。また Compute Capability 1.3 では、icount には整数のみが指定可能です。

アトミック関数は、処理をどこまで終了したかを記録するためのカウンターなどに使用することができます。ただし、図 4-1-18 のように全スレッドが (2) を実行するのではなく、可能であれば各ブロックの代表スレッド (例えばスレッド0) のみが実行した方が、アトミック関数のオーバーヘッドが少なくなります。

提供されているアトミック関数を以下に示します (「CUDA C Programming Guide」(B.10) 参照)。

`atomicAdd()`, `atomicSub()`, `atomicExch()`, `atomicMin()`, `atomicMax()`,
`atomicInc()`, `atomicDec()`, `atomicCAS()`, `atomicAnd()`, `atomicOr`, `atomicXor()`

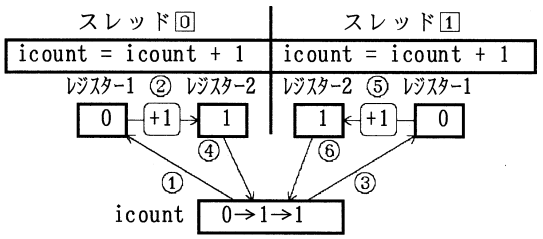


図 4-1-16 (1)　×

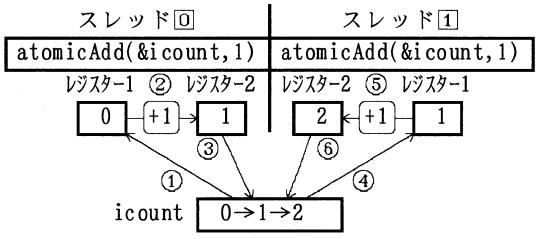


図 4-1-17 (1)

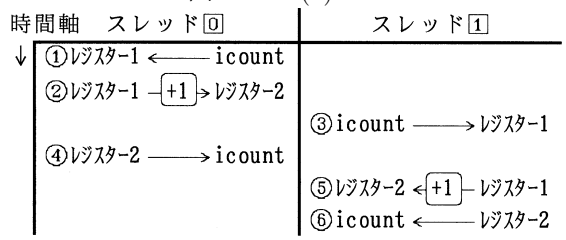


図 4-1-16 (2)　×

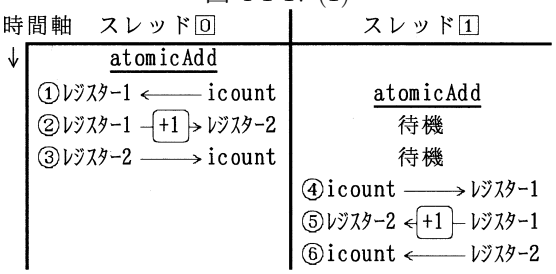


図 4-1-17 (2)

```
  __device__ int icount = 0;                    (1)
__global__ void kernel(int *dBID,int *dTID){
  int i = atomicAdd(&icount,1);                 (2)
  dBID[i] =  blockIdx.x;                        (3)
  dTID[i] = threadIdx.x;                        (4)
}
```

```
   :
kernel<<<10,4>>>(dBID,dTID)                     (5)
dBIDをBIDに、dTIDをTIDにコピーします。
for(i=0;i<40;i++){
  printf("%d %d %d¥n",i,BID[i],TID[i]);        (6)
}
   :
```

図 4-1-18

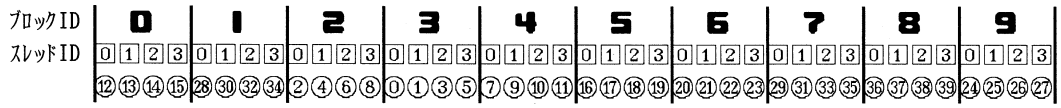


図 4-1-19

4-2　エラーチェック

本節では、CUDA 化したプログラムのエラーチェックに関する説明をします。

CUDA ランタイム API とカーネル関数のエラーチェック

図 4-2-1 のプログラムにはエラーが 2 つあります。まず、①で指定しているデバイス側の配列 dA の大きさが大き過ぎます。また、③で指定しているブロックあたりのスレッド数が上限 (512) を越えています。ところがこのプログラムを実行すると、図 4-2-4 の⑮に示すように⑤が表示され、一見正常に動作しているように見えます。しかし実際には①, ③のエラーのため、カーネル関数は動作していません。このように、CUDA のプログラムでは、特に何も指定しない場合、エラーが発生してもメッセージは出ず、異常終了もしません。

②, ④などの CUDA 関数で発生したエラーをチェックしたい場合、⑧, ⑫のように CUDA 関数を「CUDA_SAFE_CALL(～)」で囲みます。そして⑥を指定し、⑯の下線部 (1-4 節で導入した CUDA SDK) を指定してコンパイル／リンクします。プログラムを実行し、⑧でエラーが発生すると、異常終了して⑰に示すようにエラー発生箇所とエラー内容が表示されます。なお、「CUDA_SAFE_CALL」の代わりに、図 4-2-3 の⑭の「cutilSafeCall」を使用することもできます。この場合、⑥の代わりに⑬を指定し、出力メッセージは⑱となります。

③のカーネル関数で発生したエラーをチェックしたい場合、⑦の関数（関数名は任意）を使用します。そして⑨でカーネル関数を呼び出した直後の⑩で同期を取り、⑪を指定します。すると、実行中にカーネル関数内でエラーが発生した場合、カーネル関数終了後の⑪でエラーが検出され、⑲が表示されます (⑲の下線部には、⑪のカッコ内で指定したカーネル関数名が表示されます)。なお、⑨のカーネル関数内で発生したエラーは、種類によっては⑩の CUDA_SAFE_CALL で検出され、メッセージが表示される場合もあります。

本書では、紙面の都合でエラーチェックルーチンは省略しますが、必ず指定するようにして下さい。

```
__global__ void kernel(float *dA){
  :
}
int main(void){
  float *dA;
  size_t size = 10000000000*sizeof(float); ①
  cudaMalloc((void**)&dA,size);            ②
  kernel<<<1,513>>>(dA);                   ③
  cudaFree(dA);                            ④
  printf("OK¥n");                          ⑤
    :
```

図 4-2-1

```
#include <cutil.h>                                    ⑥
void CUDA_ERROR_CHECK(char *msg){
  cudaError_t status = cudaGetLastError();
  if (status != cudaSuccess){
    printf("CUDA error in %s: %s.¥n",msg,             ⑦
           cudaGetErrorString(status));
    exit(-1);
  }
}
__global__ void kernel(float *dA){
    :
int main(void){
  float *dA;
  size_t size = 10000000000*sizeof(float);
  CUDA_SAFE_CALL(cudaMalloc((void**)&dA,size));
  kernel<<<1,513>>>(dA);                   ⑨  ↰ ⑧
  CUDA_SAFE_CALL(cudaThreadSynchronize());  ⑩
  CUDA_ERROR_CHECK("kernel");               ⑪
  CUDA_SAFE_CALL(cudaFree(dA));             ⑫
  printf("OK¥n");
    :
```

図 4-2-2

```
#include <cutil_inline.h>                          ⑬
    :
  cutilSafeCall(cudaMalloc((void**)&dA,size));
    :                                          ↰ ⑭
```

図 4-2-3

```
$ nvcc (最適化ｵﾌﾟｼｮﾝ) test.cu ↵
 OK  (図4-2-1の実行結果)  ⑮
$ nvcc (最適化ｵﾌﾟｼｮﾝ) test.cu -I$HOME/NVIDIA_GPU_Computing_SDK/C/common/inc ↵  ⑯
Cuda error in file 'test.cu' in line 17 : out of memory.   (図4-2-2の実行結果)  ⑰
                                  ↰ 17行目;本例では図4-2-2の⑧の行
test.cu(17) : cudaSafeCall() Runtime API error : out of memory. (図4-2-3の実行結果)  ⑱
        ↰ 17行目;本例では図4-2-2の⑧の行
CUDA error in kernel: invalid configuration argument. (図4-2-2の実行結果)        ⑲
```

図 4-2-4

セグメンテーション・フォールトのチェック

図 4-2-5 (1) では、①に示すように大きさ 30 の配列 dA を使用します。③で、1 ブロック、ブロック内のスレッド数 32 で実行した場合、図 4-2-6 (1) に示すように、②でスレッド ID 30と31は、配列 dA の範囲を越えた部分にアクセスします。これをセグメンテーション・フォールトと呼び、メモリの内容が破壊されてプログラムが誤作動する可能性があります。図 4-2-5 (1) の場合、以下のいずれかの方法で対処する必要があります。

(1) 図 4-2-5 (2) の⑤の if 文を付け、配列 dA の範囲を越えたスレッドは、配列 dA をアクセスしないようにします。本例より複雑な例を、3-6 節の「■　割り切れない場合の処理」に示します。本書では、紙面の関係で、if 文によるチェックは基本的に省略しますが、実際のプログラムでは、適切に処理を行って下さい。

(2) 図 4-2-6 (2) に示すように、配列 dA の大きさを、全スレッド数と同じ 32 (実際に計算するのは 30 要素) にします (⑤の if 文は不要となります)。本来計算を行わないスレッド ID 30と31が計算を行うので、配列 dA[30], dA[31] を適当な値で初期化しないと、演算例外（オーバーフローやゼロ除算）を生じる可能性があります。

図 4-2-5 (1) を通常に実行した場合、発生したセグメンテーション・フォールトは検出されず、プログラムは異常終了せず、メッセージも表示されません。図 4-2-2 のチェックルーチンを付加しても同様です。

CUDA で提供されている「cuda-memcheck」コマンドを使うと、セグメンテーションフォールトを検出することができます (詳細は check-memecheck のマニュアル（付録参照）を参照して下さい)。図 4-2-7 の⑥の下線部を付けて実行すると (実際はシェルを使用して実行します)、⑦の下線部に示すように、カーネル関数 kernel 内で、読み込み時にセグメンテーションフォールトが発生したことが表示されます。

⑥の場合は、エラーが 1 回表示されたら終了しますが、⑧の下線部を付けて実行すると、⑨, ⑩に示すように、すべてのエラー (本例では③, ④に対するエラー) が表示されます (図 4-2-5 (1) で関数 kernel2 は省略しています)。

セグメンテーション・フォールトが発生しなかった場合は⑪が表示されます。cuda-memcheck は、デバッガー (CUDA-GDB) 内で使用することもできます。なお、cuda-memcheck を付けて実行すると速度が遅くなるので、デバッグのときのみ使用して下さい。

```
#define N (30)                                          ①
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;                                 ②
}
int main(void){
  float *dA;
  size_t size = N*sizeof(float);                        ①
  cudaMalloc((void**)&dA,size);
    :
  kernel <<<1,32>>>(dA);                                ③
  kernel2<<<1,32>>>(dA);                                ④
    :
```

図 4-2-5 (1)

```
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if(i<N) dA[i] = dA[i] + 1.0f;                         ⑤
}
```

図 4-2-5 (2)

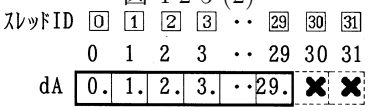


図 4-2-6 (1)

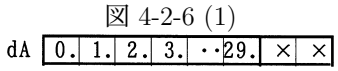


図 4-2-6 (2)

```
$ cuda-memcheck a.out ↵                                 ⑥
====== CUDA-MEMCHECK
====== Invalid read of size 4
======     at 0x00000028 in kernel
======     by thread (30,0,0) in block (0,0)            ⑦
====== Address 0x00100078 is out of bounds
======
====== ERROR SUMMARY: 1 error
$ cuda-memcheck --continue a.out ↵                      ⑧
====== CUDA-MEMCHECK
====== Invalid read of size 4
======     at 0x00000028 in kernel                      ⑨
======     by thread (30,0,0) in block (0,0)
====== Address 0x00100078 is out of bounds
======
====== Invalid read of size 4
======     at 0x00000028 in kernel2                     ⑩
======     by thread (30,0,0) in block (0,0)
====== Address 0x00100078 is out of bounds
======
====== ERROR SUMMARY: 2 errors
$ cuda-memcheck a.out ↵
====== CUDA-MEMCHECK
====== ERROR SUMMARY: 0 errors                          ⑪
```

図 4-2-7

グローバルメモリのクリア

図 4-2-10 は単純な加算のプログラムで、図 4-2-8 (1) に示すように、以下の処理を行います。

- ②でホスト側の配列 A[3]，B[3] を確保し、③で配列 A に値を設定し、④でデバイス側の配列 dA にコピーします。
- ⑤で 1 ブロック、ブロックあたり 3 スレッドでカーネル関数を実行します (⑤の横線の意味は後述します)。
- 各スレッドは、①で配列 dA に 10.0 を加算し、配列 dB に代入します。
- ⑥で、配列 dB をホスト側の配列 B にコピーし、⑦で書き出し、図 4-2-9 の (1) が表示されます。

図 4-2-10 のプログラムを実行した直後、誤って、例えば図 4-2-10 の⑤の部分を削除してしまったとします。ところがこのプログラムをコンパイルし、前と同じ計算機で実行すると、図 4-2-9 の (1) と同じ（正しい）結果が表示されてしまいます。この原因を説明します。最初の実行時の配列 dB の値 (図 4-2-8 (1) 参照) が、グローバルメモリにそのまま残っていたため、⑤を削除した図 4-2-10 を実行し、図 4-2-8 (2) の ↓ が実行されなくても、残っていた配列 dB の結果が、⑦でホスト側の配列 B にコピーされ、正しい結果が表示されました。

このような場合、図 4-2-11 のように、配列 dB をゼロクリアしてから計算を行えば、図 4-2-8 (3) のようになり、実行すると図 4-2-9 の (2) が表示されてプログラムのバグに気が付くことができます。また、図 4-2-12 の⑩のように、CUDA 関数 cudaMemset (詳細は「CUDA Reference Manual」を参照) を使用しても、配列 dB をゼロクリアできます (配列 dB を cudaMallocPitch で確保した場合は、cudaMemset2D でゼロクリアします)。

このように、プログラムを CUDA 化する際、修正を間違えたのに、メモリ上に残っていた正しい計算結果が表示されていたため、修正間違いにずっと気が付かなかったということがよくありますので、注意して下さい。また、セキュリティ上の理由から、グローバルメモリ上のデータを消去したい場合も、cudaMemset などで明示的に消去して下さい。

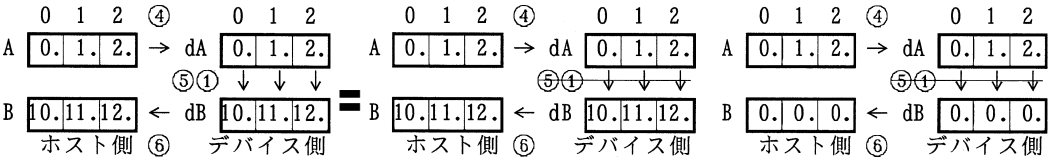


図 4-2-8 (1)

図 4-2-8 (2)

図 4-2-8 (3)

```
#define N (3)
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dB[i] = dA[i] + 10.0f;                          ①
}
int main(void){
  float A[N],B[N];                                ②
  float *dA,*dB;
  for(i=0;i<N;i++){                               ③
    A[i] = (float)i;
  }
  size_t size = N*sizeof(float);
  cudaMalloc((void**)&dA,size);
  cudaMalloc((void**)&dB,size);
  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);
  kernel<<<1,3>>>(dA,dB);                         ⑤ ↩④
  cudaMemcpy(B,dB,size,cudaMemcpyDeviceToHost);
  printf("%f %f %f¥n",B[0],B[1],B[2]);            ⑦ ↩⑥
    :
```

図 4-2-10

```
10.000000 11.000000 12.000000 (1)
 0.000000  0.000000  0.000000 (2)
```

図 4-2-9

```
    :
  for(i=0;i<N;i++){                               ⑧
    B[i] = 0.0f;
  }
  cudaMemcpy(dB,B,size,cudaMemcpyHostToDevice);
  kernel<<<1,3>>>(dA,dB);                         ⑤ ↩⑨
    :
```

図 4-2-11

```
    :
  size_t size = N*sizeof(float);
  cudaMemset(dB,0,size);                          ⑩
  kernel<<<1,3>>>(dA,dB);                         ⑤
    :
```

図 4-2-12

倍精度プログラムと `-arch=sm_13`

図 4-2-13 は倍精度のプログラムです。図 4-2-14 (1) に示すように、ホスト側の②で配列 `A[0]` に `1.0` を設定し、③でデバイス側の配列 `dA[0]` にコピーし、デバイス側では①で配列 `dA[0]` に `2.0` を再設定し、④でホスト側の配列 `A[0]` にコピーし、⑤で出力しています。

このプログラムのように、カーネル関数で倍精度の変数を使用する場合、図 4-2-15 の⑥の下線部のオプションを付けてコンパイルする必要があります (2-7 節参照)。プログラムを実行すると、①で設定した⑦が表示されます。

一方、⑥の下線部を指定し忘れて⑧でコンパイルした場合、⑨の警告メッセージが表示されますが、コンパイル／リンクは成功します。ところがプログラムを実行すると、⑩のように結果がおかしくなります。出力結果から、図 4-2-14 (2) に示すように、①が実行されず、②で設定した値がそのまま⑩で表示されたのではないかと思われます。前述のエラーチェックルーチンを指定した場合も、同じ結果となります。

このように、⑥の下線部を指定し忘れた場合、実行時にエラーメッセージが出ず、異常終了もしないため、エラーが起こったことに気付かない可能性があります。カーネル関数内で倍精度の変数を使用する場合はもちろん、使用しない場合も、理研 RICC の環境の Compute Capability 1.3 に合わせ、念のため⑥の下線部を常に指定してコンパイル／リンクすることをお勧めします (2-7 節参照)。

```
__global__ void kernel(double *dA){
  dA[0] = 2.0;                                          ①
}
int main(void){
  double A[1];
  double *dA;
  size_t size = sizeof(double);
  A[0] = 1.0;                                           ②
  cudaMalloc((void**)&dA,size);
  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);         ③
  kernel<<<1,1>>>(dA);
  cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);         ④
  printf("%f¥n",A[0]);                                  ⑤
    :
```

図 4-2-13

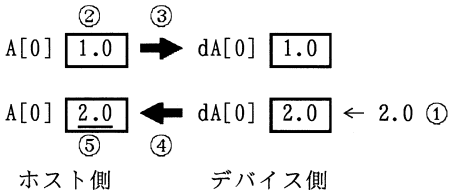


図 4-2-14 (1)

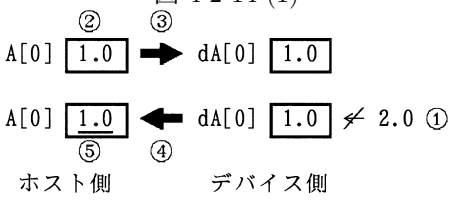


図 4-2-14 (2)

```
nvcc (最適化ｵﾌﾟｼｮﾝ) -arch=sm_13 test.cu ↵  ⑥
2.000000                                    ⑦
nvcc (最適化ｵﾌﾟｼｮﾝ) ___________ test.cu ↵  ⑧
ptxas /tmp/tmpxft_00007a3f_00000000-2_zzz.ptx, line 56; warning : Double is not supported. ⑨
   Demoting to float                                                                       ⑨
1.000000                                    ⑩
```

図 4-2-15

ホストプログラムと CUDA 化したプログラムの結果の相違

ホストで動作しているプログラムを CUDA 化した場合、両方の計算結果を比較し、修正が正しいことを確認する必要があります。ところが、特に単精度のプログラムの場合、結果が若干変わることがあります。コンパイラが行う最適化の方法や、数値計算の丸めの方法 (「CUDA C Programming Guide」C.1 節参照) の相違が原因だと思われます。

図 4-2-16 のホストプログラムと、図 4-2-17 の CUDA 化したプログラムで、単精度の「10 ÷ 3」を実行した結果を図 4-2-18 の②, ③に示します。このように簡単な計算でも、単精度だと若干変わることがあります。なお、コンパイラによる最適化の影響を避けるため、①の下線部で、ホスト側とカーネル側の最適化オプションを「-O0（オーゼロ）」にしました (2-7 節参照)。

本例のように、CUDA 化した部分が簡単な場合は、結果の相違がプログラムの修正ミスなのかどうかを判断できますが、複雑な場合、判断が難しくなります。このような場合、図 4-2-16 と 4-2-17 の下線部を double にし、単精度で実行すると、本プログラムの場合、⑤, ⑥に示すように計算結果は一致します。このように、単精度のホストプログラムと CUDA 化したプログラムで計算結果が若干異なる場合、倍精度にしてテストするのも 1 つの方法です。

なお、カーネル関数内で倍精度の変数を使用する場合は、前述のように、④の下線部のコンパイルオプションを指定する必要があります。

```
int main(void){
  float A[1];
  A[0] = 10.0f;
  A[0] = A[0]/3.0f;
  printf("%.20f¥n",A[0]);
}
```

図 4-2-16

```
__global__ void kernel(float *dA){
  dA[0] = dA[0]/3.0f;
}
int main(void){
  float A[1];
  float *dA;
  size_t size = sizeof(float);
  cudaMalloc((void**)&dA,size);
  A[0] = 10.0f;
  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);
  kernel<<<1,1>>>(dA);
  cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);
  printf("%.20f¥n",A[0]);
}
```

図 4-2-17

```
$ nvcc -O0(オーゼロ) -Xopencc -O0(オーゼロ) -test.cu ↵                    ①
(ホスト　：単精度) 3.33333325386047363281  ②
(カーネル：単精度) 3.33333349227905273438  ③
$ nvcc -O0(オーゼロ) -Xopencc -O0(オーゼロ) -arch=sm_13 test.cu ↵  ④
(ホスト　：倍精度) 3.33333333333333348136  ⑤
(カーネル：倍精度) 3.33333333333333348136  ⑥
```

図 4-2-18

4-3　デバッガー

Compute Capability 1.3 では、カーネル関数内で printf 文を使えないので、デバッガーでデバッグします。GCC コンパイラでは、デバッガー GDB が提供されています。一方 CUDA のカーネル関数用には、デバッガー CUDA-GDB が提供されており、使用方法は GDB とほぼ同じです。まず、GDB の基本的な使用方法を説明します。

- GDB は会話形式で実行しますが、理研 RICC の多目的 PC クラスタでは、会話形式の実行ができないので、⓪でインタラクティブ・バッチモードに入って実行して下さい。図 4-3-1 (1) のプログラム（左の数字は行番号）を、①で「-g」を付けてコンパイル／リンクします。最適化を抑止する「-O0（オーゼロ）」も指定した方がよいでしょう。続いて②でデバッガー GDB を起動します。なお、②の代わりに「gdb -tui a.out ↵」とすると、画面の上半分に、ソースプログラムと、デバッガーが現在停止している位置が表示されます。
- ③で、main 関数の最初の実行命令 (図 4-3-1 (1) の 3 行目) をブレークポイント（デバッガーが停止する地点）として設定します。④でプログラムを実行すると、⑤に示すように、デバッガーはブレークポイントの命令の直前まで進み、停止します。「3 i=1;」の「3」はプログラムの行番号です。
- ⑥で、この地点での変数 i の値を表示します。プログラムの 3 行目がまだ実行されていないので、i = 0（正確には、メモリ上に残っているゴミ）が表示されます。
- ⑦の「step」を実行すると、プログラムは 1 行進み、3 行目が実行されます。⑧で変数 i を表示すると 1 になります。⑦は「next」でも構いません (「step」と「next」の相違は後述します)。
- ⑨を実行すると、プログラムは 1 行進み、4 行目が実行されます。⑩で変数 i を表示すると 2 になります。
- 現在の（実行する直前の）命令が何なのかを知りたい場合、⑪を実行します。また現在の命令の前後の命令を表示したい場合、⑫を実行します。
- デバッグ作業が終了したので、プログラムを最後まで実行して終了したい場合、⑬を実行します。
- GDB 自体を終了する場合は⑭を実行します。⑭の代わりに⑮を実行した場合、GDB は終了せず。プログラムは再び最初から実行します。③で設定したブレークポイントはそのまま設定されています。
- プログラムの実行中に、プログラムを再び最初から実行する場合、⑯, ⑰のようにします。
- プログラムの実行中に、プログラムをキャンセルし、GDB も終了する場合、⑱, ⑲のようにします。

```
1 int main(void){
2   int i;
3   i=1;
4   i=2;
5 }
```

図 4-3-1 (1)　test1.c

```
$ qsub -i -accel ↵                    ⓪
$ gcc -g -O0(オーゼロ) test1.c ↵      ①
$ gdb a.out ↵                         ②
 (メッセージが表示されます)
(gdb) break main ↵                    ③
Breakpoint 1 at 0x40044c: file test1.c, line 3.
(gdb) run ↵                           ④
Starting program: /home/user/a.out
Breakpoint 1, main () at test1.c:3    ⑤
3         i=1;  (の直前)              ⑤
(gdb) print i ↵                       ⑥
$1 = 0
(gdb) step ↵                          ⑦
4         i=2;  (の直前)
(gdb) print i ↵                       ⑧
$2 = 1
(gdb) step ↵                          ⑨
5       }       (の直前)
(gdb) print i ↵                       ⑩
$3 = 2
```

```
(gdb) where ↵                         ⑪
#0  main () at test1.c:5  (5行目の直前)
(gdb) list ↵                          ⑫
1       int main(void){
2         int i;
3         i=1;
4         i=2;
5       }
(gdb) continue ↵                      ⑬
Continuing.
Program exited with code 040.
(gdb) quit ↵                          ⑭
(gdb) run                             ⑮ (⑭の代わりに)
Starting program: /home/user/a.out
Breakpoint 1, main () at test1.c:3    ⑤と同じ
3         i=1;  (の直前)              ⑤と同じ
(gdb) run ↵                           ⑯
The program being debugged has been
  started already.
Start it from the beginning? (y or n)y ↵ ⑰
Starting program: /home/user/a.out
Breakpoint 1, main () at test1.c:3    ⑤と同じ
3         i=1;  (の直前)              ⑤と同じ
(gdb) quit ↵                          ⑱
The program is running.
  Exit anyway? (y or n)y ↵            ⑲
```

図 4-3-1 (2)

図 4-3-2 (1) のように関数を含む場合、1 行ずつ進めるとどうなるかを説明します。図 4-3-2 (2) の①で、図 4-3-2 (1) の 6 行目をブレークポイントに設定して②でプログラムを実行し、6 行目の直前まで進みます。

- ③で「next」を指定した場合、関数 func を一気に実行し、main ルーチンの 7 行目の直前に戻ります。
- ③の代わりに④で「step」を指定した場合、関数 func 内に入り、⑤，⑥で関数内を 1 行ずつ進みます。
- ③の代わりに⑦で「step」を指定し、関数 func 内で（もう作業が終了したので）⑧を指定した場合、関数 func の残りの部分を一気に実行し、main ルーチンの 7 行目の直前に戻ります。

```
1 void func(){
2   int k=2;
3 }
```

```
4 int main(void){
5   int i;
6   func();
7   i=1;
8 }
```

図 4-3-2 (1)　test2.c

```
(gdb) break 6 ↵  ①
Breakpoint 1 at 0x40045d: file test2.c, line 6.
(gdb) run ↵      ②
Starting program: /home/user/a.out
Breakpoint 1, main () at test2.c:6
6         func();
(gdb) next ↵     ③
7         i=1;
```

```
(gdb) step ↵   ④ (③の代わり)
func () at test2.c:2
2         int k=2;
(gdb) step ↵   ⑤
3       }
(gdb) step     ⑥
main () at test2.c:7
7         i=1;
(gdb) step ↵   ⑦ (③の代わり)
func () at test2.c:2
2         int k=2;
(gdb) finish ↵ ⑧
Run till exit from #0  func ()
                  at test2.c:2
main () at test2.c:7
7         i=1;
```

図 4-3-2 (2)

図 4-3-3 (1) を例に、ブレークポイントの設定について説明します。図 4-3-3 (2) の①で、プログラムの 6 行目に、②で関数 func の最初の実行命令に，③で test3.c (図 4-3-3 (1)) の 8 行目に、それぞれブレークポイントを設定します。なお、プログラムの入ったファイルが 1 つの場合は、③のファイル名は指定しなくても構いません。

④で、現在設定されているブレークポイントを表示します。設定した、例えば「3」のブレークポイントを解除したい場合、⑤を実行します。⑥でプログラムを実行すると、最初のブレークポイントに到達し、（作業終了後に）⑦を実行すると次のブレークポイントに到達し、（作業終了後に）⑧を実行すると、ブレークポイントが (⑤で 1 つ削除したため) 残っていないので、プログラムの最後まで実行を行い、終了します。

```
1 void func(){
2   int k=1;     ②
3 }
```

```
4 int main(void){
5   int i;
6   i=1;         ①
7   func();
8   i=2;         ③
9 }
```

図 4-3-3 (1)　test3.c

```
(gdb) break 6 ↵          ①
Breakpoint 1 at 0x40045d: file test3.c, line 6.
(gdb) break func ↵       ②
Breakpoint 2 at 0x40044c: file test3.c, line 2.
(gdb) break test3.c:8 ↵  ③
Breakpoint 3 at 0x40046e: file test3.c, line 8.
```

```
(gdb) info breakpoints ↵  ④
Num  Type       (省略) What
1    breakpoint (省略) in main at test3.c:6
2    breakpoint (省略) in func at test3.c:2
3    breakpoint (省略) in main at test3.c:8
(gdb) delete 3 ↵  ⑤
(gdb) run ↵       ⑥
Starting program: /home/user/a.out
Breakpoint 1, main () at test3.c:6
6         i=1;
(gdb) continue ↵  ⑦
Continuing.
Breakpoint 2, func () at test3.c:2
2         int k=1;
(gdb) continue ↵  ⑧
Continuing.
Program exited normally.
```

図 4-3-3 (2)

次に、変数の表示を簡単に行う方法について説明します。図 4-3-4 (1) の 7 行目をブレークポイントに設定してプログラムを実行すると、図 4-3-4 (2) の①になります。②を実行すると配列 A の全要素が表示されます。

以後、プログラムの 7, 8 行目を実行したときの変数 sum の値の変化を調べるため、③～⑦を実行しました。このように、「step」で 1 行進んだ後、「print sum ⏎」を毎回実行するのは少々面倒です。このような場合、③の代わりに⑧を実行します。すると以後、⑨, ⑩に示すように、プログラムが停止するたびに、変数 sum の値が自動的に表示されます（⏎参照）。

現在 display で設定されている変数を調べる場合、⑪を実行します。設定を解除する場合、⑫を実行します。

```
1 int main(void){
2   int i;
3   float A[2],sum;
4   A[0] = 1.0f;
5   A[1] = 2.0f;
6   sum = 0.0f;
7   sum = sum + A[0];
8   sum = sum + A[1];
9 }
```

図 4-3-4 (1)　test4.c

```
7         sum = sum + A[0]; ①
(gdb) print A ⏎      ②
$1 = {1, 2}
(gdb) print sum ⏎    ③
$2 = 0
(gdb) step ⏎         ④
8         sum = sum + A[1];
(gdb) print sum ⏎    ⑤
$3 = 1
(gdb) step ⏎         ⑥
9       }
(gdb) print sum ⏎    ⑦
$4 = 3
```

```
7         sum = sum + A[0];
(gdb) display sum ⏎        ⑧ (③の代わり)
1: sum = 0
(gdb) step ⏎               ⑨
8         sum = sum + A[1];
1: sum = 1
(gdb) step ⏎               ⑩
9       }
1: sum = 3
(gdb) info display ⏎       ⑪
Auto-display expressions now in effect:
Num Enb Expression
1:   y  sum
(gdb) delete display 1 ⏎   ⑫
```

図 4-3-4 (2)

GDB についていくつか補足します。

- ⏎キーを押すと、それ以前にキーインしたコマンドが過去に向かって順に表示されます。逆に、⏎キーを押すと、現在に向かって順に表示されます。
- 「GDB を使った実践的デバッグ手法」(CQ 出版社) という書籍が出版されています。
- Web で検索すると、GDB の使用方法を解説したサイトが多数見つかります。以下は一例です。
  http://rat.cis.k.hosei.ac.jp/article/devel/index.html
- GDB の主なコマンドの短縮型を下記に示します。例えば「run ⏎」は「r ⏎」で実行できます。

run	r	プログラムを最初から実行します
continue	c	プログラムを再開します。
step	s	次の命令に進みます。関数内の命令も 1 行ずつ進みます。
next	n	次の命令に進みます。ただし関数内の命令は一気に実行します。
finish	fin	現在の関数の最後まで一気に実行し、呼び出し側に戻ります。
quit	q	GDB または CUDA-GDB を終了します。
list	l	現在停止している位置の前後のプログラムを表示します。
where	whe	現在停止している位置を表示します。
break	b	ブレークポイントを設定します。
info breakpoints	i b	break で設定されているブレークポイントを表示します。
delete 番号	d	指定した番号の、ブレークポイントの設定を解除します。
print	p	変数や配列の値を表示します。
display 変数名	disp	プログラムが停止するたびに、指定した変数の値を自動的に表示します。
info display	i di	display で設定されている変数／配列を表示します。
delete display 番号	d d	指定した番号の、display の設定を解除します。

CUDA のカーネル関数内のデバッグには、CUDA-GDB を使用します。図 4-3-5 (1) のプログラムで使用方法を説明します。CUDA-GDB の詳細は、「CUDA-GDB User Manual」（付録参照）を参照して下さい。

- CUDA-GDB は会話形式で実行しますが、理研 RICC の多目的 PC クラスタでは、会話形式の実行ができないので、⓪でインタラクティブ・バッチモードに入って実行して下さい。
- 図 4-3-5 (2) の①の「`-g -G`」を付けてコンパイル／リンクし、②の「`cuda-gdb`」で CUDA-GDB を開始します。
- ③でカーネル関数 `kernel` をブレークポイントに設定し、④で、図 4-3-5 (1) に示すように、ブロック数 2 (0, 1)、ブロック内のスレッド数 64 (0〜63) でプログラムを実行します。CUDA-GDB は、開始時点では⑤に示すようにブロック 0, スレッド 0 として動作し、⑥に示すようにプログラムの 3 行目の直前で停止します。
- 例えばブロック 1 のスレッド 63 (最後のブロックの最後のスレッド）で動作したい場合、⑦を実行します。
- ⑧, ⑨に示すように、`gridDim`, `blockDim`, `blockIdx`, `threadIdx` などの値を表示させることができます。
- ⑩で変数 `i` を表示すると、まだプログラムの 3 行目が実行されていないので、⑪に示すように、メモリ上のゴミ (本例では 63) とウォーニングメッセージが表示されます。
- ⑫でプログラムが 1 行進みます。この場合、CUDA-GDB が実行しているスレッドと同じワープ内の全スレッド (ブロック 1 の 32〜63) が⑫を実行し、他のスレッドは⑫を実行せず、3 行目の直前で停止しています。
- ⑬で再び変数 `i` を表示すると、CUDA-GDB が動作しているスレッドの変数 `i` の値 127（下記）が表示されます。

$$\texttt{i} = \texttt{blockIdx.x}\ (=1) * \texttt{blockDim.x}\ (=64) + \texttt{threadIdx.x}\ (=63) = 127$$

- ⑭でプログラムが 1 行進みます。この場合も、ブロック 1 のスレッド 32〜63 のみが⑭を実行します。
- ⑭が終了した時点で、ブロック 1 のスレッド 32〜63 は、4 行目まで実行が終了しています。⑮で配列 `dM` を表示すると、ブロック 1 の 32〜63 が担当した `dM[96]`〜`dM[127]` のみ、4 行目で設定した値が表示されます。
- ⑯で、各スレッドが現在停止している位置が表示されます。ブロック 0 の全スレッド (0〜63) と、ブロック 1 のスレッド 0〜31 は、プログラムの 3 行目の直前で停止し、ブロック 1 のスレッド 32〜63 は、(⑫, ⑭のコマンドによって) プログラムの 5 行目の直前で停止しています。
- 現在 CUDA-GDB が動作しているスレッドの、ブロック ID (1) とスレッド ID (63) を知りたい場合、⑰を実行します。

```
1  __device__ int dM[128];
2 __global__ void kernel(){
3   int i = blockIdx.x*blockDim.x + threadIdx.x;
4   dM[i] = i;
5 }
6 int main(void){
7    kernel<<<2,64>>>();
8 }
```

図 4-3-5 (1)　test5.cu

デバイス側の最適化オプションを念のため0に設定します。

```
$ qsub -i -accel ↵                    ⓪
$ nvcc -g -G -O0(オーゼロ) -Xopencc -O0(オーゼロ)
        test5.cu ↵                    ①
$ cuda-gdb a.out ↵                    ②
(メッセージが表示されます)
(cuda-gdb) break kernel ↵             ③
Breakpoint 1 at 0x400d9c:file test5.cu, line 2.
(cuda-gdb) run ↵                      ④
Starting program: /home/user/a.out
(メッセージが表示されます)
[Switching to CUDA Kernel 0                ⑤
  (<<<(0,0),(0,0,0)>>>)]                   ⑤
Breakpoint 1, kernel <<<(2,1),(64,1,1)>>> ()
  at test5.cu:3                            ⑥
3   int i = blockIdx.x*blockDim.x + threadIdx.x;
```

```
(cuda-gdb) cuda block (1) thread (63) ↵    ⑦
[Switching to CUDA Kernel 0 (device 0, sm 3,
  warp 1, lane 31, grid 2, block (1,0),
  thread (63,0,0))]
(cuda-gdb) print threadIdx.x ↵             ⑧
$1 = 63
(cuda-gdb) print threadIdx ↵               ⑨
$2 = {x = 63, y = 0, z = 0}
(cuda-gdb) print i ↵                       ⑩
warning: Variable is not live at this point.
  Returning garbage value.                 ⑪
$3 = 63
(cuda-gdb) step ↵                          ⑫
4         dM[i] = i;
(cuda-gdb) print i ↵                       ⑬
$4 = 127
(cuda-gdb) step ↵                          ⑭
5       }
(cuda-gdb) print dM ↵                      ⑮
$5 = {0 <repeats 96 times>, 96, 97 ··· , 127}
(cuda-gdb) info cuda threads ↵             ⑯
<<<(0,0),(0,0,0)>>> ... <<<(1,0),(31,0,0)>>>
  kernel <<<(2,1),(64,1,1)>>> () at test5.cu:3
<<<(1,0),(32,0,0)>>> ... <<<(1,0),(63,0,0)>>>
  kernel <<<(2,1),(64,1,1)>>> () at test5.cu:5
(cuda-gdb) cuda kernel ↵                   ⑰
[Current CUDA kernel 0 (device 0, sm 3, warp 1,
lane 31, grid 2, block (1,0), thread (63,0,0))]
```

図 4-3-5 (2)

4-4　タイマールーチン

ホスト側での測定

C 言語で経過時間 (Elapsed Time) を測定する方法を、カーネル関数の測定にも使うことができます。図 4-4-1 の⑦のカーネル関数の経過時間を測定する場合、①，②，③を指定し、⑦の前後の⑥と⑨で測定を行い、⑩で経過時間（単位は秒）を表示します。このとき以下で説明するように、⑧と⑤で同期を取る必要があります (4-1 節参照)。なお、経過時間は、他のジョブが流れていない占有状態で測定して下さい。

⑧を指定しないと、図 4-4-2 (1) に示すように、⑦を呼び出すとホストプログラムにすぐに制御が戻るので (3-2 節参照)、測定時間 (↕ の部分) がほぼゼロになってしまいます。⑧を指定すると、図 4-4-2 (2) に示すように、カーネル関数が終了するまでホストプログラムは待機するので、正しく測定することができます。

本例では、④で他のカーネル関数 (other) が呼ばれています。このような場合、⑤を指定しないと、図 4-4-3 (1) に示すように、⑥の時点で関数 other がまだ動いていて、その時間が測定時間 (↕ の部分) に含まれてしまう可能性があります (この場合、other が終了した後、自動的に kernel が開始します)。⑤を指定すると、図 4-4-3 (2) に示すように、関数 other が終了した後に⑥を実行するので、関数 other の時間は測定時間に含まれません。

```
#include <sys/time.h>                                   ①
double gettimeofday_sec() {                             ②
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + (double)tv.tv_usec*1e-6;
}
int main(void){
  double elp1,elp2; ← 変数名は任意                      ③
    :
```

```
other<<<1,1>>>();                                       ④
cudaThreadSynchronize();                                ⑤
elp1 = gettimeofday_sec();                              ⑥
kernel<<<1,1>>>();  ↕ 測定したい部分                    ⑦
cudaThreadSynchronize();                                ⑧
elp2 = gettimeofday_sec();                              ⑨
printf("ELAPSE = %.6f¥n",elp2-elp1);                    ⑩
  :
```

図 4-4-1

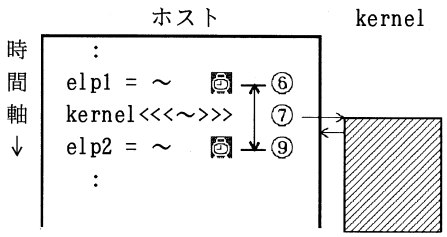


図 4-4-2 (1)　×

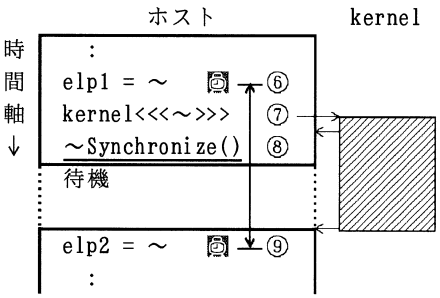


図 4-4-2 (2)

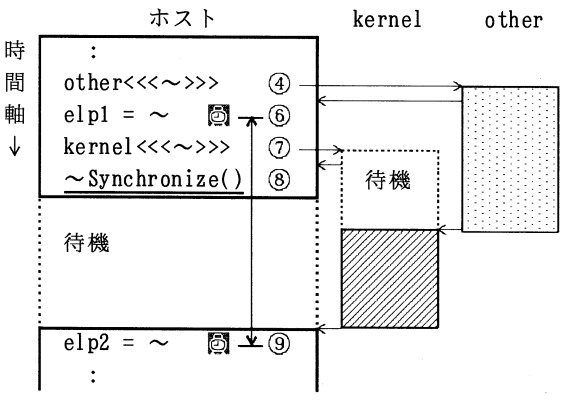


図 4-4-3 (1)　×

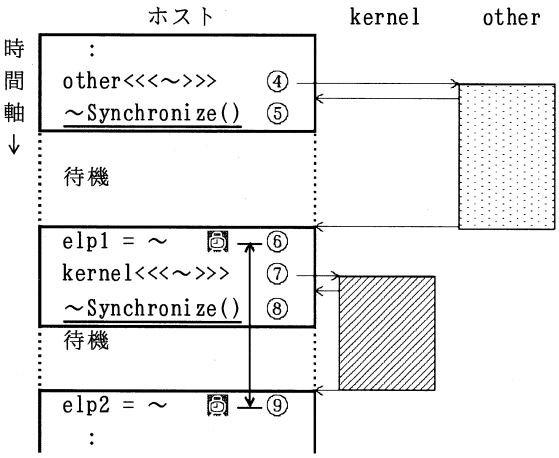


図 4-4-3 (2)　○

カーネル関数内での測定

CUDA で提供されている組込関数 clock() を使用して、カーネル関数内の各部の経過時間を、スレッドごとに測定することができます。詳細は「CUDA C Programming Guide」（付録参照）の B.9 節を参照して下さい。なお、CUDA で提供されている clock() は、C 言語で提供されている clock() とは別の関数です。

- 測定したい図 4-4-4 (1) の③の前後に②と④を挿入し、⑤で差を取ります。この値が、②と④の間の経過時間（単位はクロック数）となります。経過時間なので、コアが実際に動いた時間以外の時間も含まれます。また、実行するたびに値は多少変動します。
- 測定値はスレッドごとに求まります。そこで、①でスレッド数分の大きさの配列 dELAPSE をグローバルメモリに確保し、⑤で配列 dELAPSE にスレッドごとに測定値を保存します。
- ⑥でホスト側の配列 ELAPSE にコピーします。
- ワープ内の 32 スレッドは同じ測定値になるようです。そのため、⑦，⑧で、32 スレッドごとに書き出しを行います。出力例を図 4-4-4 (2) に示します。
- 単位はクロック数なので、この値を秒に変換して使うのではなく、以下のように、速度を相対的に比較する場合に用いるのがよいでしょう。
  - カーネル関数内の、ある部分と他の部分の速度の比較（どの部分が時間がかかっているかを調べるため）
  - カーネル関数内のある部分の、チューニング前とチューニング後の速度の比較（チューニングの効果を調べるため）
  - 異なるワープあるいはブロック間の比較（ワープ間あるいはブロック間のロードバランスが均等かどうかを調べるため）

```
#define N (30*256)
  __device__ int dELAPSE[N];   ①

__global__ void kernel(float *A){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  clock_t ista = clock();      ②
  A[i] = A[i] + 1.0f;          ③ 測定したい部分
  clock_t iend = clock();      ④
  dELAPSE[i] = iend - ista;    ⑤
}
```

```
int main(void){
  int ELAPSE[N];
    :
  kernel<<<30,256>>>(dA);
  cudaMemcpyFromSymbol(ELAPSE,dELAPSE,
    N*sizeof(int),0,cudaMemcpyDeviceToHost);⑥
  for(i=0;i<N;i+=32){                        ⑦
    printf("THREAD = %d                      ⑧
            ELAPSE = %d¥n",i,ELAPSE[i]);
  }
    :
```

図 4-4-4 (1)

```
THREAD = 0 ELAPSE = 1134
THREAD = 32 ELAPSE = 1088
THREAD = 64 ELAPSE = 1160
THREAD = 96 ELAPSE = 1124
    :
```

図 4-4-4 (2)

4-5　プロファイラー

一般にプロファイラーは、プログラムをチューニングする前に、プログラムの中で計算時間がかかっている部分を調べるために使用します。そしてその部分に対して、チューニングや並列化を行います。ホスト側では、通常 prof や gprof などのプロファイラーが提供されており、理研の RICC では、富士通のプロファイラーが提供されています。

これらのプロファイラーで時間のかかっている部分を調べ、その部分を CUDA 化したとします。CUDA 化した部分に、速度上の問題がないかを調べるために、Compute Profiler という、CUDA 用のプロファイラーが提供されています。本節では、Cuda Profiler の簡単な使用方法を説明します。詳細はマニュアル「Compute_Profiler.txt」（付録参照）を参照して下さい。

また、Compute Profiler の結果をグラフィカルに表示し、解析しやすくした、Compute Visual Profiler というツールが提供されています（本書では説明しません）。詳細はマニュアル「COMPUTE VISUAL PROFILER」(http://www.nvidia.com/object/cuda_get.html の「Linux」の「Visual Profiler User Guide」) を参照して下さい。なお、Compute Visual Profiler の使用方法の簡単な説明が、http://gpu.fixstars.com/index.php/ の「CUDA プログラミング TIPS」の「CUDA_Visucal_Profiler を使う」、および参考文献 [1] にあります。

Cuda Profiler の基本的な使用方法

図 4-5-1 のプログラムを例に、Compute Profiler (以下プロファイラー) の基本的な使用方法を説明します。コンパイル／リンクは nvcc で普通に行います。実行時に、図 4-5-2 の①の環境変数 (C シェルおよび tcsh では「setenv COMPUTE_PROFILE 1 ↵」) を指定してプログラムを実行すると、図 4-5-3 に示すように、「cuda_profile_0.log」というファイルが作成されます。

⑤は、この 4 つのパラメータ（デフォルト）に関するプロファイルが行われたことを示します。⑥, ⑧の下線部は、「ホストからデバイスへのコピー」と「デバイスからホストへのコピー」を意味し、図 4-5-1 の (6), (8) に対応します。⑦の下線部はカーネル関数名「kernel」を意味し、図 4-5-1 の (7) に対応します。

「COMPUTE VISUAL PROFILER」（付録参照）の「PROFILER OUTPUT TABLE」の「GPU Time」と「CPU Time」によると、⑥〜⑧の「gputime」は、GPU が動作した時間 (単位はマイクロ秒 (10^{-6} 秒)) を表します。

一方「cputime」は、非同期関数（カーネル関数など）では、CPU 側でかかった（関数呼び出しなどの）オーバーヘッドの時間のみを表し、同期関数 (cudaMemcpy など) では、CPU 側でかかった（関数呼び出しなどの）オーバーヘッドの時間と GPU が動作した時間の合計を表します。ただし後述する「■　指定できる数が 4 個以下のパラメータ」を指定した場合、非同期関数のカーネル関数は上記の同期関数と同じ扱いになるようです。

```
#define N (30*2*128)                                    (1)
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if(i<N) dA[i] = dA[i] + 1.0f;                         (2)
}
int main(void){
  size_t size = N*sizeof(float);                        (3)
  float A[N];
  float *dA;
  配列Aに値を設定します。
  cudaMalloc((void**)&dA,size);                         (6)
  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);
  kernel<<<30*2,128>>>(dA);                             (7)
  cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);
    :                                                   (8)
```

図 4-5-1

```
export COMPUTE_PROFILE=1 ↵                              ①
プログラムを実行
export COMPUTE_PROFILE_LOG=xxx ↵                        ②
export COMPUTE_PROFILE_CSV=1 ↵                          ③
export COMPUTE_PROFILE_CONFIG=yyy ↵                     ④
```

図 4-5-2

```
method,gputime,cputime,occupancy
memcpyHtoD,16.704,17.000
_Z6kernelPf,3.904,11.000,1.0000
memcpyDtoH,9.344,43.000
```

図 4-5-4

```
method,gputime,cputime,occupancy                                                    ⑤
method=[ memcpyHtoD ] gputime=[ 16.704 ] cputime=[ 17.000 ]                         ⑥
method=[ _Z6kernelPf ] gputime=[ 3.904 ] cputime=[ 11.000 ] occupancy=[ 1.000 ]     ⑦
method=[ memcpyDtoH ] gputime=[ 9.344 ] cputime=[ 43.000 ]                          ⑧
```

図 4-5-3　cuda_profile_0.log (または xxx)

⑦の「ocupancy」は、ワープの専有率 (6-1 節参照) が 100% であることを意味します。

②の環境変数で xxx にファイル名を指定すると、図 4-5-3 がファイル xxx に作成されます。③の環境変数を指定すると、図 4-5-3 は図 4-5-4 のように表示されます。

⑤（デフォルトのパラメータ）以外に、各種のパラメータをプロファイルすることができます。④の環境変数で yyy にファイル名を指定し、そのファイル名に、以下のようにプロファイルしたいパラメータを設定します。

yyy
```
timestamp
  :
```

パラメータは、上記ファイル内に指定できる数に制限がないパラメータと、指定できる数が 4 個以内のパラメータに分類されます。まず指定できる数に制限がないパラメータについて説明します。以下の説明中の「～」部分がパラメータを示します。

指定できる数に制限がないパラメータ

(1)　タイムスタンプ

- 「timestamp」：カーネル実行とコピーのタイムスタンプが表示されます。
- 「gpustarttimestamp」：カーネルが実行を開始した時点のタイムスタンプが表示されます（ただし数字の意味は不明です）。
- 「gpuendtimestamp」：カーネルが実行を終了した時点のタイムスタンプが表示されます。

【出力例】上記のパラメータを指定して、図 4-5-1 のプログラムを実行した場合の出力例を示します。各パラメータの結果は、図 4-5-3 の⑥～⑧の該当部分に付加されます。

```
timestamp=[ 2585.000 ] gpustarttimestamp=[ 11ee47e53e267480 ]
                         gpuendtimestamp=[ 11ee47e53e26b5c0 ] method=[ memcpyHtoD ] ～
timestamp=[ 2626.000 ] gpustarttimestamp=[ 11ee47e53e270a40 ]
                         gpuendtimestamp=[ 11ee47e53e2719a0 ] method=[ _Z6kernelPf ] ～
timestamp=[ 2640.000 ] gpustarttimestamp=[ 11ee47e53e2754c0 ]
                         gpuendtimestamp=[ 11ee47e53e277940 ] method=[ memcpyDtoH ] ～
```

(2)　実行構成

- 「gridsize」：図 4-5-1 の (7) の実行構成で指定した、x，y 方向のブロック数が表示されます。
- 「threadblocksize」：実行構成で指定した、x，y，z 方向のスレッド数が表示されます。

【出力例】図 4-5-1 の (7) の実行構成で指定した値が表示されます。

```
method=[ _Z6kernelPf ] gridsize=[ 60, 1 ] threadblocksize=[ 128, 1, 1 ] ～
```

(3)　ストリーム ID

- 「streamid」：CUDA 関数、およびカーネル関数に付けられたストリーム ID (6-3 節参照) が表示されます。

【出力例】本例では、デフォルトのストリーム ID ゼロが表示されます。

```
method=[ memcpyHtoD ] streamid=[ 0 ] ～
method=[ _Z6kernelPf ] streamid=[ 0 ] ～
method=[ memcpyDtoH ] streamid=[ 0 ] ～
```

(4)　ホストとデバイス間のコピー

- 「memtransfersize」：ホスト〓デバイス間でコピーする量がバイトで表示されます。
- 「memtransferdir」：コピーする方向が表示されます。ホストからデバイスへのコピーは「1」（注）、デバイスからホストへのコピーは「2」（注）が表示されます。
- 「memtransferhostmemtype」：コピーするホスト側のメモリの種類を指定します。通常は「0」、cudaHostAlloc (6-2 節参照) で確保した変数／配列の場合は「1」が表示されます。

（注）「Compute_Profiler.txt」では、前者が「0」、後者が「1」になっていますが、【出力例】のように表示されました。

【出力例】図 4-5-1 の (1), (3) に示すように、(6), (8) のコピーの量は $30 \times 2 \times 128 \times 4$（バイト）$= 30720$（バイト）です。

```
method=[ memcpyHtoD ] memtransfersize=[ 30720 ] memtransferdir=[ 1 ]
                      memtransferhostmemtype=[ 0 ] ～
method=[ memcpyDtoH ] memtransfersize=[ 30720 ] memtransferdir=[ 2 ]
                      memtransferhostmemtype=[ 0 ] ～
```

(5)　シェアードメモリ

- 「dynsmemperblock」：1 つのブロックが使用する、「extern __shared__」変数型修飾子 (3-6 節参照) で動的に配置されたシェアードメモリの量（バイト）が表示されます。この情報は、コンパイルオプション「-Xptxas=-v」(2-7 節参照) を付けてコンパイルした場合は表示されません。
- 「stasmemperblock」：1 つのブロックが使用する、「__shared__」変数型修飾子で、静的に配置されたシェ」」アードメモリの量（バイト）が表示されます。この情報は、コンパイルオプション「-Xptxas=-v」(2-7 節参照) を付けてコンパイルした場合も表示されます。

【出力例】図 4-5-1 では、明示的にシェアードメモリは確保していませんが、カーネル関数の引数などで、24 バイトのシェアードメモリが内部的に使用されています。

```
method=[ _Z6kernelPf ] dynsmemperblock=[ 0 ] stasmemperblock=[ 24 ] ～
```

(6)　レジスター

- 「regperthread」：1 つのスレッドが使用するレジスターの個数が表示されます。なお、この情報は、コンパイルオプション「-Xptxas=-v」(2-7 節参照) を付けてコンパイルした場合も表示されます。

【出力例】図 4-5-1 では、スレッドあたりレジスターが 2 個使用されます。

```
method=[ _Z6kernelPf ] regperthread=[ 2 ] ～
```

指定できる数が 4 個以下のパラメータの注意点

以下で説明する、「指定できる数が 4 個以下のパラメータ」では、例えば「実行命令数」のように、ジョブを実行したときの、プロファイル対象が発生した回数を調べます。これらのパラメータを使用する場合の注意点を以下に説明します。詳細は「COMPUTE VISUAL PROFILER」の「INTERPRETING COUNTER VALUES」を参照して下さい。

理研 RICC の GPU (C1060) には、図 4-5-5 の [0]〜[29] に示すように、30 個のストリーミング・マルチプロセッサー (本節では SM と省略) が搭載されています。各 SM は、Texture Processing Cluster (本節では TPC と省略) という装置に含まれており、Compute Capability 1.3 では、図 4-5-5 の (0)〜(9) に示すように、1 つの TPC に 3 つの SM が含まれています。

図 4-5-1 の (7) では、実行構成を `<<<30*2,128>>>` と指定したので、全ブロック数は 30 × 2 個、1 つの SM あたりのブロック数は 2 個、1 つのブロック内のスレッド数は 128 個 (= 4 ワープ) となります。これを図で表すと図 4-5-5 のようになります。ブロックを縦長の長方形、ワープを●, ○, ○で表します。1 つのワープには、連続した 32 スレッドが入ります。

以下では、図 4-5-5 を用いて、「指定できる数が 4 個以下のパラメータ」の注意点を説明します。

- 回数を測定する範囲は全ての SM ではなく、パラメータによって以下のどちらかになります。
  (A) `SM[0]` での回数：図 4-5-5 の `SM[0]` で実行した、全ブロック内の全ワープ (図 4-5-5 の●の部分) での回数。
  (B) `TPC(0)` での回数：図 4-5-5 の `TPC(0)` で実行した、全ブロック内の全ワープ (図 4-5-5 の●と○の部分) での回数。
- 回数の数え方について説明します。1 つのループ内の全スレッド (32 スレッド) は、同じ命令を（ほぼ）同時に実行します。1 つのワープ内の各スレッドがある命令を実行した場合、以下の説明では、ワープの実行命令回数は 1 回であると言います。
  また、グローバルメモリからの要素のロード（ストアも同様）は、ハーフワープ (16 スレッド) ごとに行われます。1 つのハーフワープ内のスレッドが 1 要素をロードした場合、以下の説明では、ハーフワープのロード回数は 1 回であると言います。
  マニュアルによると、上記の (A) の場合はワープの回数になっていますが、(B) の場合は、「ワープの回数ではない」という記述しかありません（恐らくハーフワープの回数だと思われます）。どちらになっているかについては、後述する各パラメータごとに説明します。

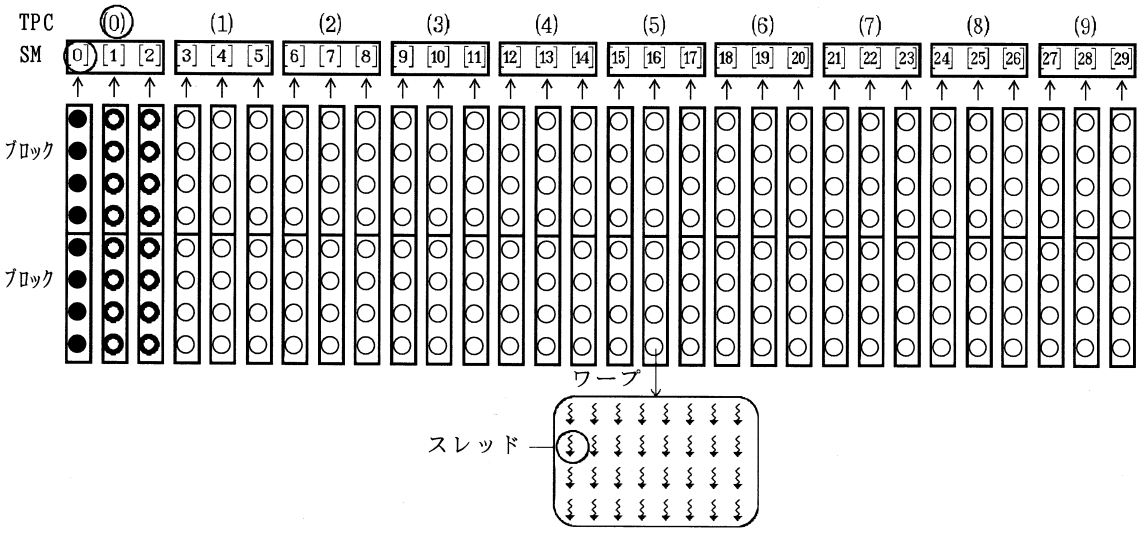


図 4-5-5　●, ○, ○は 1 ワープ (32 スレッド) を表します。

- （参考）下記の動作は、後述するパラメータ「sm_cta_launched」と「cta_launched」でのテスト結果からの推定で、マニュアルには記載されていません。
  図 4-5-6 (1) (2) (3) に示すように、ブロック数を 1，10，11 で実行した場合、各ブロック (ID = 0〜10) は、図 4-5-7 (1) (2) (3) の各 SM で動作するようです。このことから、プロファイルの結果は次のようになります。
  – 「(A) SM[0] での合計」のパラメータをプロファイルした場合、図 4-5-6 (1) (2) (3) は同じ結果になります (SM[0] 内のブロック数が同じ (= 1) なので)、
  – 「(B) TPC(0) での合計」のパラメータをプロファイルした場合、図 4-5-6 (1) (2) は同じ結果になり (TPC(0) 内のブロック数が同じ (= 1) なので)、図 4-5-6 (3) は値が 2 倍になります (TPC(0) 内のブロック数が 2 なので)。
  – 図 4-5-6 (4) のように、2 つのカーネル関数をブロック数 1 で連続に実行した場合、kernel0 の実行が終了してから kernel1 の実行が開始します (3-2 節参照)。この場合、kernel0 のブロック 0 と kernel1 のブロック 0 は、図 4-5-7 (4) の各 SM で動作するようです。そして、「(A) SM[0] での合計」と「(B) TPC(0) での合計」のパラメータはいずれも、kernel0 では値が現れ、kernel1 では値がゼロとなります (kernel0 のブロックは SM[0] と TPC(0) 上に存在し、kernel1 のブロックは SM[0] と TPC(0) 上に存在しないので)。

```
   :
kernel<<<1,32>>>();
   :
```

図 4-5-6 (1)

```
   :
kernel<<<10,32>>>();
   :
```

図 4-5-6 (2)

```
   :
kernel<<<11,32>>>();
   :
```

図 4-5-6 (3)

```
   :
kernel0<<<1,32>>>();
kernel1<<<1,32>>>();
   :
```

図 4-5-6 (4)

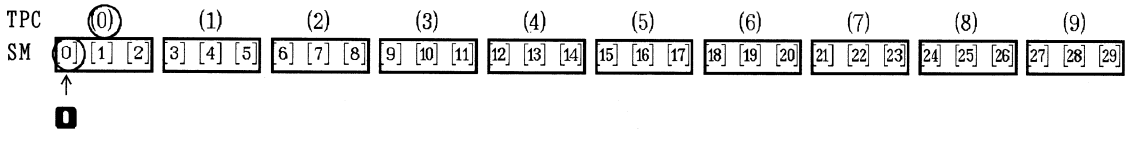


図 4-5-7 (1)

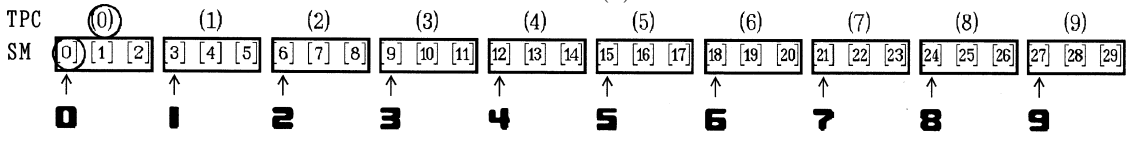


図 4-5-7 (2)

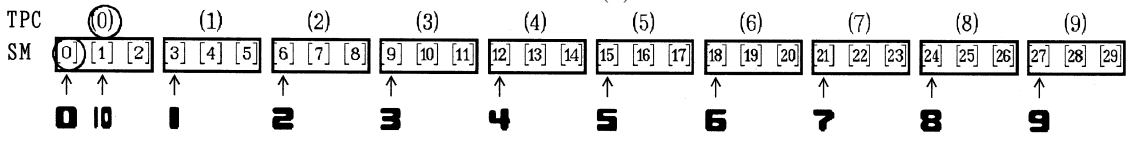


図 4-5-7 (3)

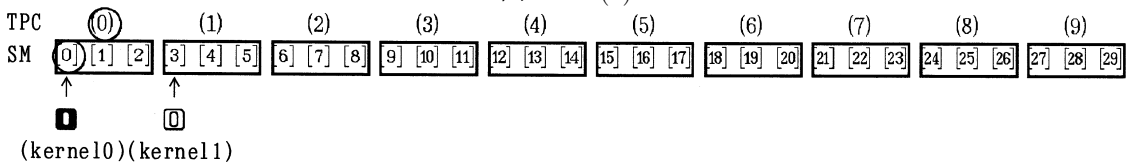


図 4-5-7 (4)

- 「COMPUTE VISUAL PROFILER」の「INTERPRETING COUNTER VALUES」では、プロファイラーを実行する場合のブロックの数は、ストリーミング・マルチプロセッサの数 (理研 RICC では 30 個) と、少なくとも同じか、倍数 (例えば 60，90 個) が望ましいと記載されています。

指定できる数が 4 個以下のパラメータ

以下で説明するパラメータは、図 4-5-2 の④のファイルに 4 個まで指定することができます。5 個以上指定した場合は、図 4-5-3 の一番上に以下のメッセージが表示されます。5 個以上の情報を取得したい場合は、パラメータを 1 回に 4 個まで指定し、ジョブを複数回実行してください。

```
NV_Warning: Signal パラメータ名 can not be profiled in this run.
```

なお、以下のパラメータのうち、チューニングで注目すべきパラメータについて、【注目点】の表示を加えました。

(7)　ブロックの数

- 「sm_cta_launched」：SM[0] で実行したブロックの数が表示されます。
- 「cta_launched」：TPC(0) で実行したブロックの数が表示されます。

【出力例】図 4-5-1 では、図 4-5-5 に示すように、SM[0] で実行したブロック数は 2 個、TPC(0) で実行したブロック数は 6 個となります。

```
method=[ _Z6kernelPf ] sm_cta_launched=[ 2 ] cta_launched=[ 6 ] ～
```

(8)　実行命令数

- 「instructions」：[ワープの実行命令数] × [SM[0] で実行したワープ数] が表示されます。なお、「gld_request」または「gst_request」と同時に指定すると、「instructions」の値が実際より増えるので、同時に指定しないで下さい。

【出力例】図 4-5-1 のプログラムでは、下記のように 88 命令となりました。図 4-5-5 に示すように、SM[0] で実行したワープ数は 8 個なので、ワープあたり 11 (= 88/8) 命令を実行したと思われます。

```
method=[ _Z6kernelPf ] instructions=[ 88 ] ～
```

(9)　分岐命令数

- 「branch」：[ワープの（実際に実行した）分岐命令数 (if 文など)] × [SM[0] で実行したワープ数] が表示されます。同期を取る「__syncthreads()」も分岐命令としてカウントされます。なお、「gld_request」または「gst_request」と同時に指定すると、「branch」の値が実際より増えるので、同時に指定しないで下さい。
- 「divergent_branch」：[ワープのワープ・ダイバージェント (6-5 節参照) した分岐命令数] × [SM[0] で実行したワープ数] が表示されます。

【出力例】図 4-5-1 の (2) に if 文が 1 つあり、図 4-5-5 に示すように SM[0] のワープ数は 8 個なので、ワープの分岐命令数は 8 個になります。またワープ内の全スレッドで if 文の判定が同じ（真）なので、ワープ・ダイバージェントは発生せず、0 個になります。

```
method=[ _Z6kernelPf ] branch=[ 8 ] divergent_branch=[ 0 ] ～
```

【注目点】ワープ・ダイバージェントが 0 個以外のときは、可能であれば 0 個になるようにして下さい (6-5 節参照)。

(10)　グローバルメモリのロード／ストア命令の要求

- 「gld_request」：[ワープの（グローバルメモリからの）ロード命令要求回数] × [SM[0] (注 2) で実行したワープ数] が表示されます。
- 「gst_request」：[ワープの（グローバルメモリへの）ストア命令要求回数] × [SM[0] (注 2) で実行したワープ数] が表示されます。

(注 1) 上記の 2 つのパラメータは、「branch」または「instructions」と同時に指定すると、「branch」、「instructions」の値が実際より増えるので、同時に指定しないで下さい。

(注 2)「Compute_Profiler.txt」では「SM[0] で実行した値」、「COMPUTE VISUAL PROFILER」では「TPC(0) で実行した値」と記載されていますが、下記の例では、「SM[0] で実行した値」になっていました。

【出力例】図 4-5-5 に示すように、SM[0] で実行したワープ数は 8 個で、図 4-5-1 の (2) で、ワープがロードとストアの要求を 1 回行なうので、8 回になります。

```
method=[ _Z6kernelPf ] gld_request=[ 8 ] gst_request=[ 8 ] ～
```

(11)　グローバルメモリの 32/64/128 バイトトランザクションのロード／ストア

- 「gld_32b」「gld_64b」「gld_128b」：[ハーフワープ (注 2) の (グローバルメモリからの 32 バイト，64 バイト，128 バイトトランザクションの) ロード回数] × [TPC(0) で実行したハーフワープ数] が表示されます。
- 「gst_32b」「gst_64b」「gst_128b」：[ハーフワープ (注 2) の (グローバルメモリへの 32 バイト，64 バイト，128 バイトトランザクションの) ストア回数 (注 3)] × [TPC(0) で実行したハーフワープ数] が表示されます。

(注 1) 前述の「gld_request」「gst_request」との違いについては、次ページで説明します。

(注 2) 2 冊のマニュアルには記載されていませんが、下記の例から「ハーフワープごと」だと思われます。

(注 3)「COMPUTE VISUCAL PROFILER」では、ストアの方は、32，64，128 バイトトランザクションの 1 回のストアを、それぞれ 2 回、4 回、8 回にカウントすると記載されていますが、下記の例では、ロードとストアの回数は同じなので、1 回にカウントされているようです。

【出力例】図 4-5-1 の配列 dA は、64 バイトトランザクションのロード／ストアになります (3-2 節参照)。図 4-5-5 に示すように、TPC(0) で実行したハーフワープ数は 48 (= 8 × 3 × 2) 個で、図 4-5-1 の (2) で、ハーフワープが 64 バイトトランザクションのロード／ストアを 1 回行うので、48 回になります。

```
method=[ _Z6kernelPf ] gld_32b=[ 0 ] gld_64b=[ 48 ] gld_128b=[ 0 ] ～
method=[ _Z6kernelPf ] gst_32b=[ 0 ] gst_64b=[ 48 ] gst_128b=[ 0 ] ～
```

【注目点】前述の「gld_request」「gst_request」と「gld_32b」「gld_64b」「gld_128b」「gst_32b」「gst_64b」「gst_128b」を使用して、グローバルメモリに対するロード／ストアが、最も効率よくコアレスアクセスされているか (3-2 節参照) を知ることができます。これを以下で説明します。

図 4-5-1 の (2) の配列 dA に対するロードの場合で説明します（ストアの場合も同じです)。図 4-5-1 の (2) では、ワープのロード命令の要求回数は 1 回です。コアレスアクセスで最も効率よくロードする場合、ハーフワープが（単精度なので) 64 バイトトランザクションのロードを 1 回行います (3-2 節参照)。本例で、ロード回数が 2 回以上だったり、32 バイトトランザクションや 128 バイトトランザクションが含まれている場合は、コアレスアクセスの効率が悪くなります。

まとめると、以下のようになっていれば、コアレスアクセスが最も効率よく行われています。

> A：(10)[ワープのロード命令要求回数] ＝ (11)[ハーフワープの(トランザクションの)ロード回数] になっている。
> 　　(10)[ワープのロード命令要求回数] ＜ (11)[ハーフワープの(トランザクションの)ロード回数] では効率が悪くなります。
> B：単精度(倍精度)の場合、64(128)バイトトランザクションのみでロード/ストアを行っている。

前ページのプロファイル例の結果を使用して、上記について検討します。説明中で、図 4-5-5 の一部を取り出した図 4-5-8 を参照します。

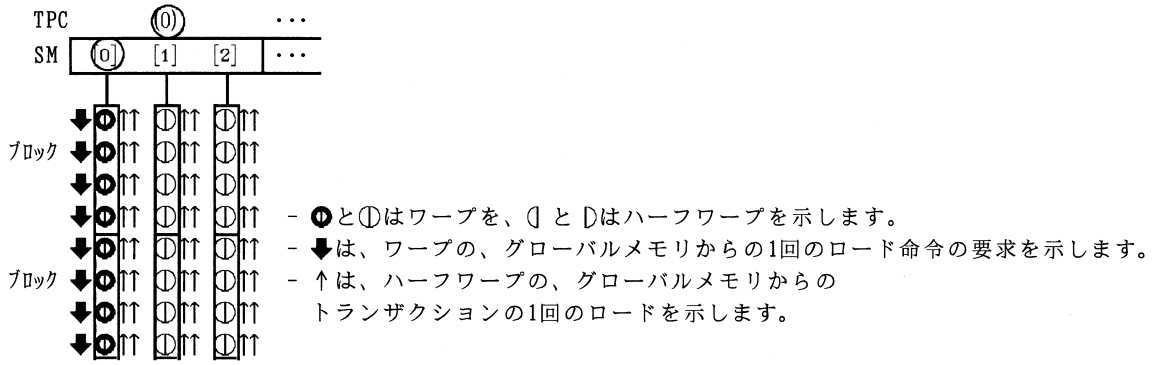


図 4-5-8

まず、以下の 3 つの値を使用して、(10)［ワープのロード命令要求回数］(1 つの〓あたりの ↓ の数) を求めます。

- 「gld_request」：SM[0] で実行した全てのワープの、グローバルメモリからのロード命令の要求回数は 8 回です (図 4-5-8 の ↓ の数)。
- 「sm_cta_launched」：SM[0] で実行したブロックの数は 2 個です (図 4-5-8 の左端の長方形の数)。
- 「1 ブロック内のワープ数」：図 4-5-1 の (7) で指定した、1 ブロック内のスレッド数 (本例では 128) を 32 で割り、128 ÷ 32 ＝ 4 個 となります (図 4-5-8 の 1 つの長方形内の〓の数)。

(10)[ワープのロード命令要求回数]（1つの❶あたりの⬇の数）
＝ [ワープのロード命令要求回数]×[SM[0]内で実行したワープ数] ÷ [SM[0]内で実行したワープ数]
＝ 「gld_request」 ÷ (「sm_cta_launched」×「1ブロック内のワープ数」)
＝ 8回 ÷ (2個 × 4個)
＝ 1回

次に、以下の 3 つの値を使用して、(11)［ハーフワープの（トランザクションの）ロード回数］(1 つの〓または〓あたりの ↑ の数) を求めます。

- 「gld_64b」：TPC(0) で実行した全てのハーフワープの、グローバルメモリからの 64 バイトトランザクションのロード回数は 48 回です (図 4-5-8 の ↑ の数)。
- 「cta_launched」：TPC(0) で実行したブロックの数は 6 個です (図 4-5-8 の全ての長方形の数)。
- 「1 ブロック内のハーフワープ数」：図 4-5-1 の (7) で指定した、1 ブロック内のスレッド数 (本例では 128) を 16 で割り、128 ÷ 16 ＝ 8 個 となります (図 4-5-8 の 1 つの長方形内の〓と〓の数)。

(11)[ハーフワープの(トランザクションの)ロード回数]（1つの〔 または 〕あたりの↑の数）
＝ [ハーフワープの(トランザクションの)ロード回数]×[TPC(0)内で実行したハーフワープ数] ÷ [TPC(0)内で実行したハーフワープ数]
＝ 「gld_64b」 ÷ (「cta_launched」×「1ブロック内のハーフワープ数」)
＝ 48回 ÷ (6個 × 8個)
＝ 1回

以上より、A:［ワープのロード命令要求回数］＝［ハーフワープの（トランザクションの）ロード回数］(＝ 1 回) を満足します。また図 4-5-1 の配列 dA は単精度で、ロードでは「gld_64」しか使われていないので ((11) の【出力例】参照) B も満足します。従って図 4-5-1 の (2) の配列 dA のロードでは、コアレスアクセスが最も効率よく行われています。

(12)　グローバルメモリのロードとストアのコアレスアクセス

- 「gld_incoherent」「gld_coherent」「gst_incoherent」「gst_coherent」：これらのパラメータは、「COMPUTE VISUAL PROFILER」によると、Compute Capability 1.2 以降では無効なパラメータになっているので、説明は省略します。

(13)　ローカルメモリのロードとストア

- 「local_load」：[ハーフワープ (注 1) のローカルメモリからのロード回数] × [TPC(0) (注 2) で実行したハーフワープ数] が表示されます。
- 「local_store」：[ハーフワープ (注 1) のローカルメモリへのストア回数] × [TPC(0) (注 2) で実行したハーフワープ数] が表示されます。このとき、32，64，128 バイトのトランザクションを、1 回でなく、それぞれ 2 回、4 回、8 回にカウントします。

(注 1) 2 冊のマニュアルには記載されていませんが、テストしたところ、「ハーフワープごと」だと思われます。

(注 2)「Compute_Profiler.txt」では「SM[0] で実行した値」、「COMPUTE VISUAL PROFILER」では「TPC(0) で実行した値」と記載されていますが、テストしたところ、「TPC(0) で実行した値」になっていました。

【出力例】図 4-5-1 では、ローカルメモリを使用していないので、ゼロが表示されます。

```
method=[ _Z6kernelPf ] local_load=[ 0 ] local_store=[ 0 ] ～
```

【注目点】カーネル関数内で宣言したローカル変数は、通常レジスターに置かれますが、数が多いとローカルメモリに置かれ、速度が低下します (3-8 節参照)。上記の値がゼロ以外になっている場合は注意して下さい。

(14)　シェアードメモリのバンクコンフリクト

- 「warp_serialize」：[ワープのシェアードメモリ (注 1) のバンクコンフリクトの回数 (注 2)] × [SM[0] で実行したワープ数] が表示されます。

(注 1) 2 冊のマニュアルによるとコンスタントメモリに対しても適用されるとのことですが、コンスタントメモリがどのようなバンク構成になっているかの記述はないようです。

(注 2) テストしたところ、何を「1 回」と数えるのか不明でした (3-6 節参照)。

【出力例】図 4-5-1 では、明示的にシェアードメモリを使用していないので、ゼロが表示されます。

```
method=[ _Z6kernelPf ]  warp_serialize=[ 0 ] ～
```

【注目点】この値がゼロ以外のときは、原因を調べ、可能であればゼロになるようにして下さい (3-6 節参照)。

第5章　高速化編(高速化の3要素)

5-1　速度向上率を上げるためには

5，6 章では、CUDA 化したプログラムを高速化する方法について説明します。まず本章では、GPU に特化せず、一般の並列計算に共通する高速化の方法を説明します。従って、GPU での並列化に必ずしも関連しない項目も含まれます。まず並列計算機と並列計算方法について概観します。

- 分散メモリ型並列計算機：図 5-1-1 (1) に示すように、（最も基本的な構成の場合）1 つのノードは、1 つの CPU、1 つの OS (Linux，Windows など)、1 つのメモリから構成され、ノード間はネットワークで結合されます。
- 共有メモリ型並列計算機：図 5-1-1 (2) に示すように、複数の CPU、1 つの OS、1 つのメモリから構成されます (マルチコアプロセッサーの場合は、図 5-1-1 (2) 全体が CPU、図中の CPU がコアとなります)。最近は、図 5-1-1 (1) の各ノードが図 5-1-1 (2) になっている、共有／分散メモリの複合型の並列計算機が一般的です。
- MPI 並列のプログラム：図 5-1-1 (1) に示す、分散メモリ型並列計算機用の並列プログラムです (ただしマシン環境によっては図 5-1-1 (2) でも実行可能です)。①に示す MPI (Message-Passing Interface) の通信ルーチンを使用して、②のネットワークを経由してデータの通信を行います。
- スレッド並列のプログラム：図 5-1-1 (2) に示す、共有メモリ型並列計算機用の並列プログラムです。コンパイラに自動的に並列化させる方法や、③に示す OpenMP の指示行をループに指定する方法があります。

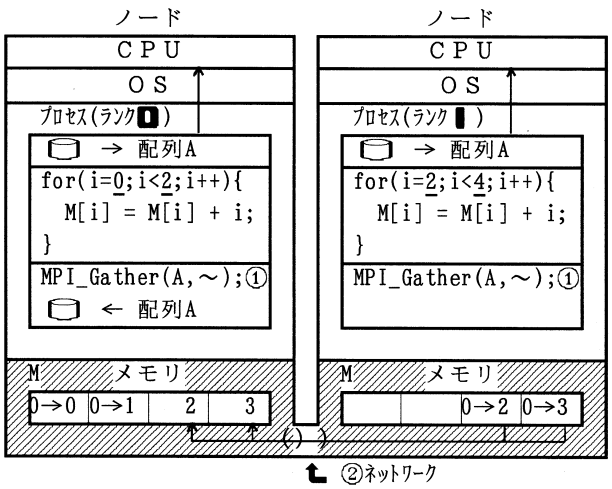


図 5-1-1 (1)

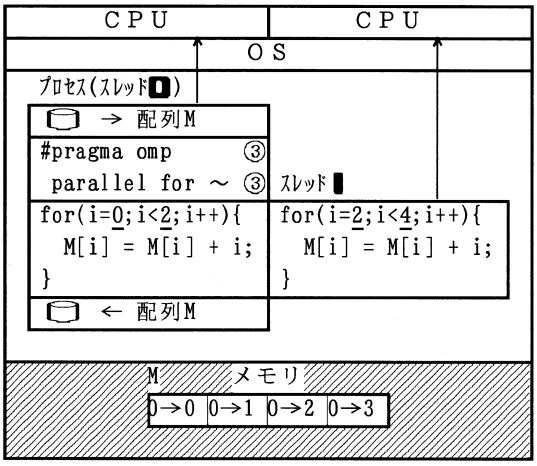


図 5-1-1 (2)

アムダールの法則

アムダールの法則を簡単に説明します。図 5-1-2 に示すように、1 台の CPU で計算時間が 100 分かかるプログラムを並列化し、計算時間の 80% の部分が並列化されたとします (これを並列化率 80% と言います)。80% もの部分が並列化できるので、一見、並列化の効果が高いように思われます。

しかしこのプログラムを、たとえ 100 万台の CPU で並列に計算しても、図 5-1-2 に示すように、100 分 → 20 分（理想的な場合）にしかなりません。つまり、計算時間の 80% しか並列化できないプログラムでは、CPU をいくら増やしても、速度向上率は 5 倍 (= 100 分/20 分) が限界です。例えば CPU 100 台で 80 倍のような高い速度向上率を得るためには、並列化率が 80% 程度では論外で、少なくとも 99% 以上である必要があります。

ただし、これは CPU の台数が多い場合の話です。CPU の台数が少ない場合、並列化率が 80% ならば、CPU 2 台で 1.67 倍、4 台で 2.5 倍と、それなりの効果が得られます。

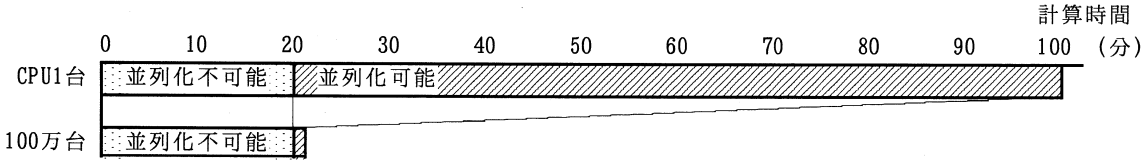


図 5-1-2

速度向上率を上げるためには

CPU 1 台で 100 分かかっているプログラムは、理想的には、CPU 4 台で 25 分 (速度向上率は 4 倍) になります。ところが図 5-1-3 の場合、50 分 (速度向上率は 2 倍) にしかなりません。その原因を以下に示します。

(1) 並列化率が 80% しかなく、並列に実行しても、前述のように並列化できない 20% の部分が残っています。
(2) 並列化可能な 80% の部分が、1/4 の 20% にならず、各 CPU で計算時間がバラついています。このような場合、一番遅い CPU (本例では CPU 3) に足をひっぱられてしまいます。言い換えると、処理が早く終了して遊んでいる CPU (CPU 0 など) があるため、速度が遅くなります。
(3) プログラムを並列に実行すると、各種のオーバーヘッドが発生します。例えば、スレッド生成／同期のオーバーヘッド（スレッド並列の場合)、データの通信時間 (MPI 並列の場合) などがあります。

以上より、速度向上率を上げるためには、以下の点に考慮する必要があります。これらについて、次節以降で説明します。

(1) (CPU の台数が多い場合) 並列化率を高くする。
(2) CPU 間のロードバランスを均等にする。
(3) 並列化に伴うオーバーヘッドを少なくする。

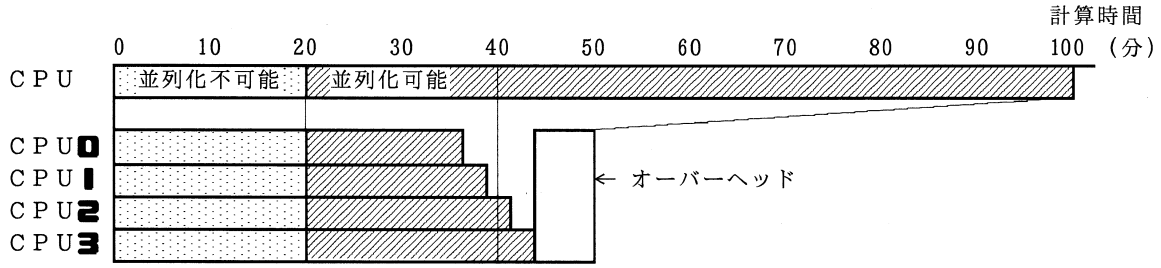


図 5-1-3

CUDA 化して並列化の効果が出るかどうかを見積もる方法

図 5-1-3 を GPU での並列計算に適用してみます。CUDA 化を検討している (CUDA 化する前の) プログラムの各部の時間が、例えば①〜③のようになっていたとすると、図 5-1-3 は図 5-1-4 のようになります。

① CPU で、ジョブ全体の計算時間は 100 分でした。
② CPU で、GPU で並列化可能な部分のみの計算時間を測定したところ、60 分でした。GPU は CPU より圧倒的に計算時間が速いので、この部分は CUDA 化すると近似的にゼロになるとみなすことができます。
③ CUDA 化した場合に、CPU と GPU の間でコピーが必要になります。そのコピー量と同じ量のコピーだけを行う簡単な CUDA プログラムを作成し、コピー時間を測定したところ、20 分かかりました。

図 5-1-4 から、このプログラムを CUDA 化した場合、（理想的な場合で）40% 程度速くなると予想されます。このように、プログラムによっては、CUDA 化して効果が出そうかどうかを、CUDA 化する前にある程度見積もることができます。

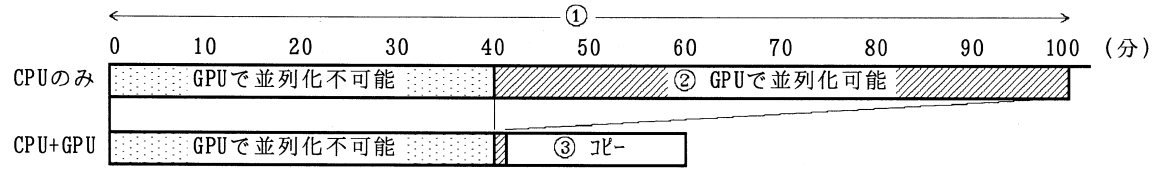


図 5-1-4

5-2　並列性とは何か

5-1 節の「(1) 速度向上率を上げるために、並列化率を高くする」方法を説明する前提として、本節では「並列性とは何か」について説明します。並列化するのは並列性のあるループです。CUDA で並列化する場合、並列化したいループに並列性があるかどうかは、ユーザーが自分で判断する必要があります。並列性のないループを誤って並列化した場合、通常は結果がおかしくなるので間違いに気付きますが、ループによっては、100 回に 99 回は結果が（たまたま）正しく、1 回だけおかしくなるという可能性もあるので、並列性の判断には注意が必要です。

並列性の定義

並列化する部分の多くはループなので、以下ではループの並列化を想定します。反復回数が 2 回のループを逐次に処理した場合、図 5-2-1 (1) に示すように、ループの 1 反復目、2 反復目 (以後、反復 1、反復 2 と略記します) の順に処理されます。このループを並列化して並列に実行した場合、図 5-2-1 (2) のように実行が行われます。なお、図 5-2-1 (1) (2) の「CPU」とは、その CPU で稼働するスレッドまたはプロセスを意味します。

並列化された反復 1 と反復 2 は、どちらが先に実行されるか、あるいは全く同時に実行されるか分かりません。反復 1 と反復 2 がどの順に実行された場合でも、図 5-2-1 (1) (2) の計算結果が同じになるループが、並列性のあるループ、あるいは並列化できるループです。

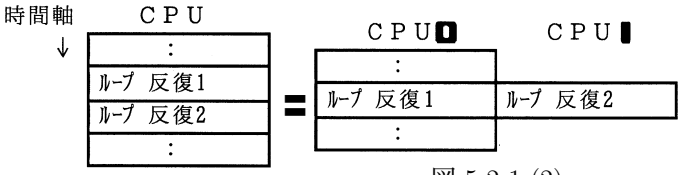


図 5-2-1 (1)　　図 5-2-1 (2)

ループ反復間の依存関係

どのようなときに、図 5-2-1 (1) (2) の計算結果が同じになるか (= 並列性があるか) を検討します。

- 図 5-2-2 (1) のように、反復 1 と反復 2 が、異なる変数や配列しか使用しない場合、計算結果は同じになります。
- 図 5-2-2 (2) のように、反復 1 と反復 2 が、同じ変数 A を参照のみする場合、計算結果は同じになります。
- 図 5-2-2 (3) では、反復 1 で変数 A を更新し、反復 2 で同じ変数 A を参照しています。これは後方依存性と言います。この場合、2 つの反復には依存関係があり、必ず反復 1、反復 2 の順に実行する必要があります。このループを並列に実行した場合、反復 1 と反復 2 のどちらが先に実行されるかは、タイミングによって異なります。反復 1 が先に実行された場合、逐次処理と実行順序が同じなので、たまたま逐次処理と同じ結果になります。一方反復 2 が先に実行された場合、反復 1 が更新する前の変数 A の値を反復 2 が参照してしまうため、逐次処理と計算結果が変わってしまいます。このように、後方依存性のあるループは、タイミングによって逐次処理と計算結果が変わってしまう可能性があるため、並列性はありません。
- 図 5-2-2 (4) では、反復 1 で変数 A を参照し、反復 2 で同じ変数 A を更新します。これを前方依存性と言います。並列に実行し、反復 2 が先に実行された場合、反復 2 が A を更新した後で反復 1 が A を参照し、逐次処理と結果が変わってしまうので、並列性はありません。
- 図 5-2-2 (5) では、反復 1 と反復 2 が同じ変数 A を更新します。これを出力依存性と言います。並列に実行し、反復 2 が先に実行された場合、反復 2 が A を更新した後で反復 1 が A を更新し、逐次処理と結果が変わってしまうので、並列性はありません。

以上のように、ループの各反復間に依存関係（後方依存性、前方依存性、出力依存性）があるループは、並列性がなく、並列化できません（後述するように、ループによっては、ロジックを変更して並列化できる場合もあります）。依存関係があって並列性のないループを、回帰参照のあるループと呼ぶこともあります。

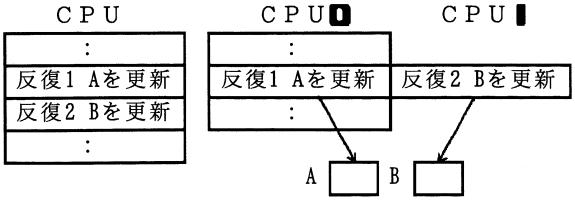


図 5-2-2 (1)　並列性あり○

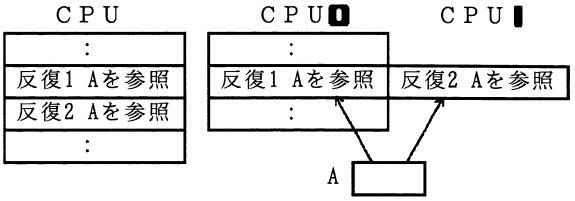


図 5-2-2 (2)　並列性あり○

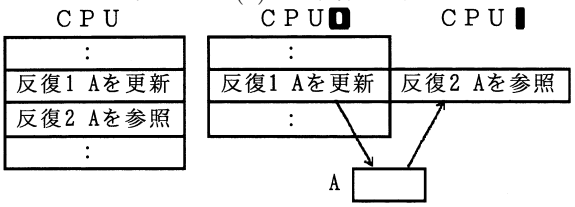


図 5-2-2 (3)　後方依存性×

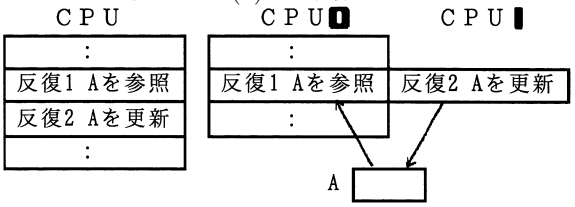


図 5-2-2 (4)　前方依存性×

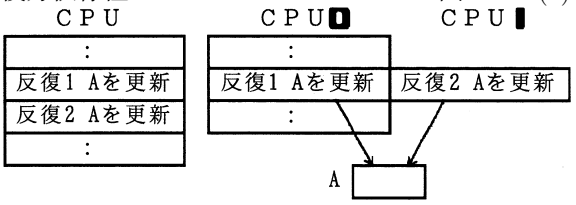


図 5-2-2 (5)　出力依存性×

並列性のあるループの例

図 5-2-3 (1) のループを展開した図 5-2-3 (2) の、①と②のステートメント間には全く依存関係がありません。言うまでもないですが、このようなループは並列性があります。逐次処理で実行した場合の動作を図 5-2-3 (3) に示します。図の例えば「0 → 1」は、実行前の「0」が、実行後「1」に変化したことを示します。このループを並列に実行した場合、図 5-2-4 (1) (2) に示すように、CPU 0 の処理 (①) と CPU 1 の処理 (1) が同時に実行されても、①が先に、あるいは1が先に実行されても、図 5-2-3 (3) と同じ結果になります。

```
    :
for(i=0;i<2;i++){
  A[i] = B[i] + C;
}
    :
```

図 5-2-3 (1)　逐次

```
A[0] = B[0] + C; ①

A[1] = B[1] + C; ②
```

図 5-2-3 (2)　逐次

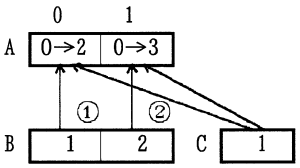


図 5-2-3 (3)　逐次

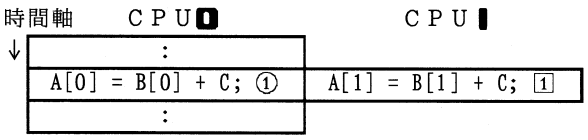


図 5-2-4 (1)　並列

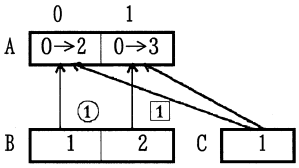


図 5-2-4 (2)　並列

後方依存性のあるループの例

後方依存性のあるループの具体例を図 5-2-5 (1) に示します。ループを展開した図 5-2-5 (2) の ↘ に示すように、①の計算結果を②で使用するため、①と②のステートメント間には、①が終了しないと②を実行できないという依存関係があります。

まず逐次処理で実行した場合、図 5-2-5 (3) のようになります。このループを並列に実行し、図 5-2-6 (1) (2) に示すように、CPU 0 の処理 (①) が先に実行された場合は、図 5-2-5 (3) の結果と一致します。しかし図 5-2-7 (1) (2) に示すように、CPU 1 の処理 (1) が先に実行された場合は、図 5-2-7 (2) の下線部の計算結果が図 5-2-5 (3) と変わります。このため、このループには並列性がありません。なお、図 5-2-5 (1) のループの場合、ロジックを変更すればある程度の並列化ができます。この方法については次節で説明します。

```
   :
for(i=1;i<3;i++){
  A[i] = A[i-1] + B[i];
}
   :
```

図 5-2-5 (1)　逐次

```
A[1] = A[0] + B[1]; ①
A[2] = A[1] + B[2]; ②
```

図 5-2-5 (2)　逐次

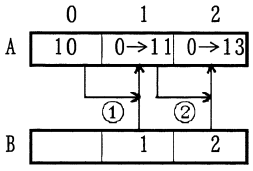


図 5-2-5 (3)　逐次

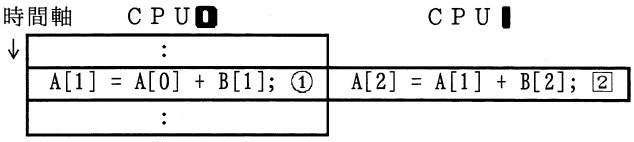


図 5-2-6 (1)　並列 (①が先に実行)

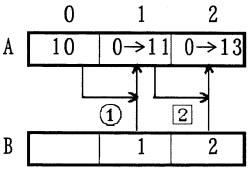


図 5-2-6 (2)　並列 (①が先に実行)

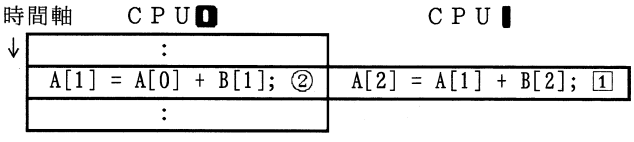


図 5-2-7 (1)　並列 (1が先に実行)

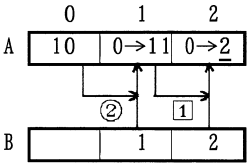


図 5-2-7 (2)　並列 (1が先に実行)

前方依存性のあるループの例

前方依存性のあるループの例を図 5-2-8 (1) に示します。ループを展開した図 5-2-8 (2) から分かるように、ステートメント間に ／ 方向の、(逐次処理では存在しない) 依存関係があります。

まず逐次処理で実行した場合、図 5-2-8 (3) のようになります。このループを並列に実行した場合、図 5-2-9 (1) (2) に示すように、CPU 0 の処理 (①) が先に実行された場合は、図 5-2-8 (3) の結果と一致します。しかし図 5-2-10 (1) (2) に示すように、CPU 1 の処理 (1) が先に実行された場合は、図 5-2-10 (2) の下線部の計算結果が図 5-2-8 (3) と変わります。このため、このループには並列性がありません。

なお、図 5-2-8 (1) のループの場合、ロジックを少し変更すれば並列化ができます。この方法については次節で説明します。

```
   :
for(i=0;i<2;i++){
  A[i] = A[i+1] + B[i];
}
   :
```

図 5-2-8 (1)　逐次

```
A[0] = A[1] + B[0]; ①
A[1] = A[2] + B[1]; ②
```

図 5-2-8 (2)　逐次

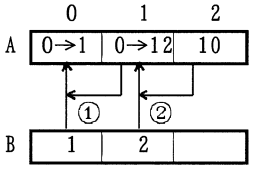


図 5-2-8 (3)　逐次

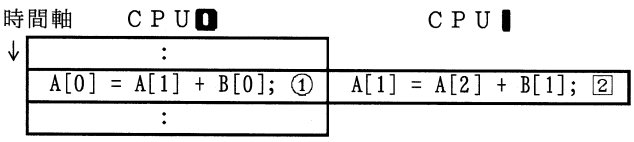


図 5-2-9 (1)　並列 (①が先に実行)

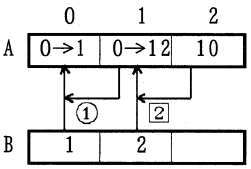


図 5-2-9 (2)　並列 (①が先に実行)

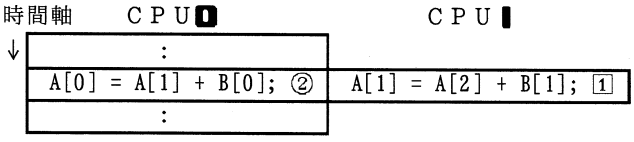


図 5-2-10 (1)　並列 (1が先に実行)

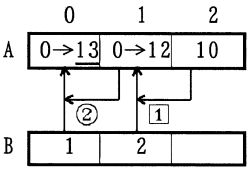


図 5-2-10 (2)　並列 (1が先に実行)

出力依存性のあるループの例

図 5-2-11 (1) では、左辺がリストベクトル (配列 INDEX) を使用した間接アドレスになっています。ループを展開した図 5-2-11 (2) の〓から分かるように、①と③で同じ要素 A[1] を更新しており、出力依存性があります。

まず逐次処理で実行した場合、図 5-2-11 (3) のように処理が行われ、A[1] は最終的に「3」になります。このループを、各 CPU が 2 反復ずつ担当して並列に実行し、図 5-2-12 (1) (2) に示すように、CPU 0 の処理 (①) が先に実行された場合は、図 5-2-11 (3) の結果と一致します。しかし図 5-2-13 (1) (2) に示すように、CPU 1 の処理 (1) が先に実行された場合は、図 5-2-13 (2) の下線部の計算結果が図 5-2-11 (3) と変わります。このため、このループには並列性がありません。

なお、図 5-2-14 (1) のように、下線部の値が全部異なる場合は、図 5-2-14 (2) に示すように、配列 A の同じ要素を更新しておらず、図 5-2-14 (3) に示すように並列化が可能です。

```
int INDEX[4]={1,3,1,2};
for(i=0;i<4;i++){
  A[INDEX[i]] = B[i];
}
  :
```

図 5-2-11 (1)　逐次

```
A[1] = B[0]; ①
A[3] = B[1]; ②
A[1] = B[2]; ③
A[2] = B[3]; ④
```

図 5-2-11 (2)　逐次

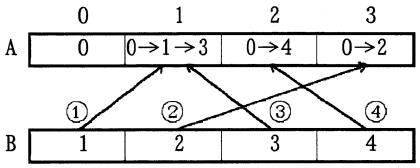


図 5-2-11 (3)　逐次

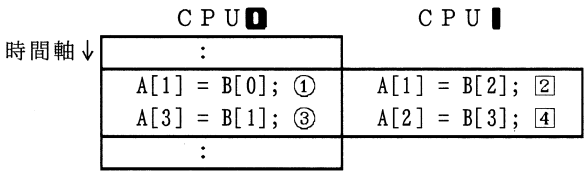


図 5-2-12 (1)　並列 (①が先に実行)

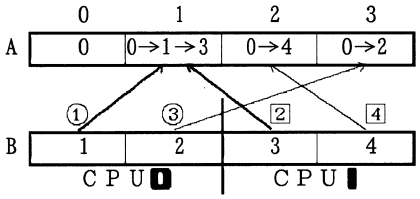


図 5-2-12 (2)　並列 (①が先に実行)

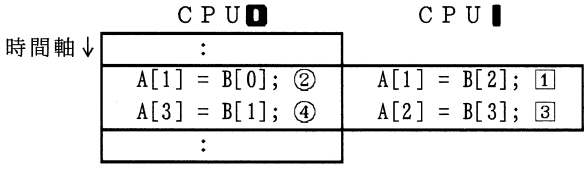


図 5-2-13 (1)　並列 (1が先に実行)

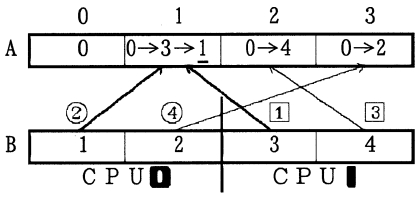


図 5-2-13 (2)　並列 (1が先に実行)

```
int INDEX[4]={1,3,0,2};
for(i=0;i<4;i++){
  A[INDEX[i]] = B[i];
}
  :
```

図 5-2-14 (1)　逐次

```
A[1] = B[0]; ①
A[3] = B[1]; ②
A[0] = B[2]; ③
A[2] = B[3]; ④
```

図 5-2-14 (2)　逐次

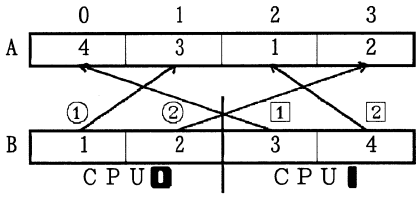


図 5-2-14 (3)　並列

一時変数と並列性

図 5-2-15 のように、計算の中間結果を保管するため、一時変数 temp を使用するループの並列性について検討します。説明を簡単にするため、図 5-2-16 (1) の例で説明します。

図 5-2-16 (1) のループを展開すると図 5-2-16 (2) になります。ループの 2 反復間には、後方依存性 (↘)、前方依存性 (↗)、出力依存性（〓）があるため、並列性がありません。このように、ループの各反復で同じ一時変数を更新するループを並列化した場合、複数の CPU が同じ一時変数をほぼ同時に更新／参照する可能性があるため、タイミングによって結果が正しかったり間違ったりする現象が発生します（同期を指定し忘れた場合と同様の現象です）。

一方、一時変数 temp を配列にした図 5-2-17 (1) の場合、図 5-2-17 (2) のようにループの 2 反復間の依存性がなくなり、並列性があります。このループを並列化した場合、複数の CPU が異なる一時変数を更新／参照するため、タイミングによって結果が変わることはなく、常に正しい結果となります。

これを CUDA 化したプログラムに当てはめると、図 5-2-18 (1) (2) では、各スレッドが同じ一時変数を更新／参照するため、図 5-2-16 (1) と同様に並列性はなく、タイミングによっては誤動作します。一方図 5-2-18 (3) (4) では、各スレッドが異なる一時変数を更新／参照するため、図 5-2-17 (1) と同様に並列性があり、正常に動作します。

```
    :
for(i=0;i<2;i++){
  temp = A[i] + 1.0f;
  B[i] = temp*2.0f;
  C[i] = temp*3.0f;
}
    :
```

図 5-2-15

```
    :
for(i=0;i<2;i++){
  temp = A[i];
  B[i] = temp;
}
    :
```

図 5-2-16 (1)

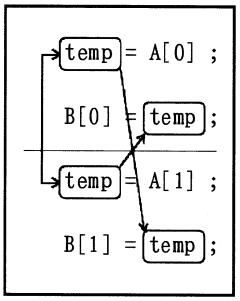


図 5-2-16 (2)

```
    :
for(i=0;i<2;i++){
  temp[i] = A[i];
  B[i] = temp[i];
}
    :
```

図 5-2-17 (1)

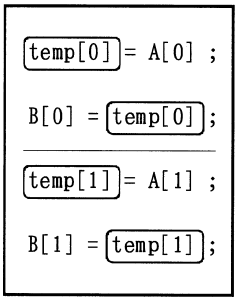


図 5-2-17 (2)

```
__device__ float temp;
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  temp = dA[i];   tempをcudaMallocで確保した
  dB[i] = temp;   場合も同様に間違いです。
}
```

図 5-2-18 (1)　×　間違い

```
__global__ void kernel(float *dA,float *dB){
  float temp;
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  temp = dA[i];   tempはｽﾚｯﾄﾞごとに
  dB[i] = temp;   異なるﾚｼﾞｽﾀｰに確保されます。
}
```

図 5-2-18 (3)

```
__global__ void kernel(float *dA,float *dB){
  __shared__ float temp;
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  temp = dA[i];
  dB[i] = temp;
}
```

図 5-2-18 (2)　×　間違い

```
__global__ void kernel(float *dA,float *dB){
  __shared__ float temp[512];
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  temp[i] = dA[i];   ﾌﾞﾛｯｸ内のｽﾚｯﾄﾞ数が
  dB[i] = temp[i];   512の場合の例です。
}
```

図 5-2-18 (4)

縮約演算と並列性

図 5-2-19 のように、配列の複数の要素 (A[0]～A[5]) の値から、1 つの結果 (sum) を求める演算を、本書では縮約演算 (reduction operation) と呼びます。合計、内積、最大／最小などが代表的な縮約演算です。

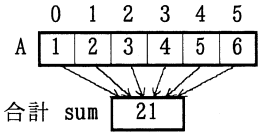


図 5-2-19

縮約演算を行うグループの並列性について検討します。図 5-2-20 (1) に示す、合計を求めるループを展開した図 5-2-20 (2) の (1) の部分の動作を、図 5-2-21 (1) の①～④に示します。図中の〓はレジスターを示します。まず①で、変数 sum の現在の値 (0) をレジスターにロードし、②で、加算する A[0] をレジスターにロードし、③で 2 つのレジスターを加算し、④で変数 sum にストアします。図 5-2-20 (2) の (1) が終了し、次に (2) で、図 5-2-21 (1) の⑤～⑧を同様に実行します。最終的に、変数 sum の値は「3」(= 1 + 2) になります。

このループを図 5-2-20 (3) のように並列化し、例えば CPU 0 と CPU 1 が図 5-2-21 (2) の数字の順に動作したとします。すると CPU 0 が⑦で、「1」を変数 sum にストアし、その後 CPU 1 が⑧で、「2」を変数にストアするため、最終的に変数 sum の値は「2」になり、図 5-2-21 (1) の結果と変わってしまいます。これは、異なる CPU が同じ変数に、ほぼ同時に値を加算したのが原因です。このように、縮約演算は、そのままでは並列化することはできません。

図 5-2-22 に示すように、A で各 CPU が、メモリ上の異なる変数に値を加算し、それを最後に B で、図 5-2-21 (2) の問題が発生しないように加算すれば (注 1)、縮約演算の並列化が可能です。CUDA での具体的な縮約演算の方法については 8-3 節で説明します。

(注 1) 図 5-2-22 で、CPU 0 が代表して B の加算を行う方法や、ロック機能 (ある CPU が B の加算を実行中に、他の CPU は B の加算ができないようにする機能) を使用する方法などがあります。

```
    :
sum = 0.0f;
for(i=0;i<2;i++){
  sum = sum + A[i];
}
    :
```

図 5-2-20 (1)

```
    :
sum = 0.0f;
sum = sum + A[0]; (1)
sum = sum + A[1]; (2)
    :
```

図 5-2-20 (2)

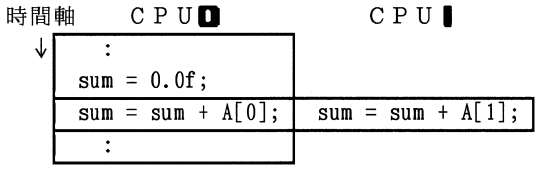


図 5-2-20 (3)

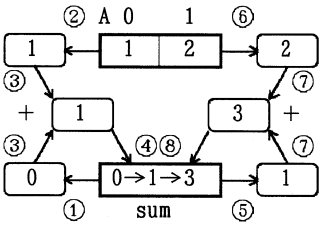


図 5-2-21 (1)

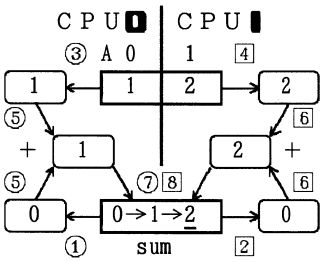


図 5-2-21 (2)

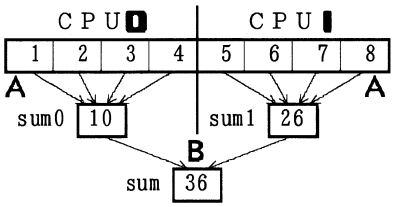


図 5-2-22

縮約演算を並列化する場合の注意点

縮約演算のうち、実数の合計や内積を求めるループを並列化する場合の注意点を説明します。a, b, c が実数のとき、数学の世界では下記の (1) 式が成立しますが、計算機の世界では、扱える桁数が有限なので、丸め誤差や桁落ちの影響で、厳密には (2) 式となります。

$$(1)\ (a+b)+c=a+(b+c) \quad (2)\ (a+b)+c \neq a+(b+c)$$

実数の合計や内積を求めるループを並列化した場合、以下のように、CPU 数によって加算の順序が変わるため、上記の理由により、CPU 数によって計算結果が若干変わる可能性があります。また例えば 4 CPU で、⊕の加算が「早い者順」に行われる場合は、同じ 4 CPU でも実行するたびに結果が若干変わることがあります。

- 逐次処理：`A[0] + A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7]`
- 2 CPU: (`A[0] + A[1] + A[2] + A[3]`) + (`A[4] + A[5] + A[6] + a[7]`)
- 4 CPU: (`A[0] + A[1]`) ⊕ (`A[2] + A[3]`) ⊕ (`A[4] + A[5]`) ⊕ (`A[6] + A[7]`)

例えば連立一次方程式の反復解法 (CG 法など) では、答えが収束するまで（計算結果がある条件を満足するまで）何度も計算を繰り返します。この解法で現れる、合計や内積のループを並列化すると、上記の理由で、CPU 数によって、あるいは同じ CPU 数でも実行するたびに、収束回数が変わる可能性があります。

複雑な計算を行うループの並列性

本節の例のように構造が簡単なループは、並列性があるかどうかを判断するのは容易ですが、図 5-2-23 (1) のように構造が複雑な場合、判断が難しくなります。これを容易に、（ある程度）判断できる方法を説明します。

図 5-2-24 (1) のループは並列性があります。このループを展開すると図 5-2-25 (1) になります。各行の順番を変えた全てのパターンを図 5-2-25 (2)〜図 5-2-25 (6) に示します。なお、図 5-2-24 (1) の②のループ反復の順番を逆にした図 5-2-24 (2) を展開したのが図 5-2-25 (6) です。

図 5-2-24 (1) のように並列性のあるループの場合、展開した図 5-2-25 (1)〜図 5-2-25 (6) は全て、図 5-2-24 (1) と同じ計算結果になります（並列に実行したとき、どの順に実行が行われても正しくなる必要があるので）。従って、図 5-2-24 (1) が並列性のあるループであれば、図 5-2-24 (2) も同じ計算結果になります。

以上のことから、図 5-2-23 (1) の場合も、①のループ反復の順番を逆にした図 5-2-23 (2) が図 5-2-23 (1) と異なる計算結果ならば、①のループに並列性はないと判断することができます。一方同じ計算結果の場合、①のループに並列性のある可能性は高いですが、断言することはできません。

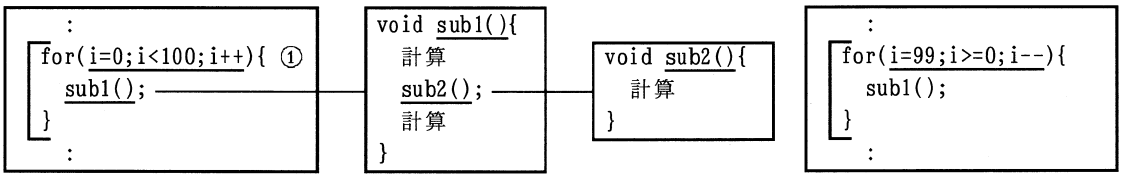


図 5-2-23 (1)　　図 5-2-23 (2)

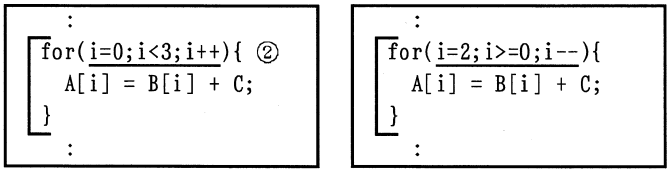


図 5-2-24 (1)　　図 5-2-24 (2)

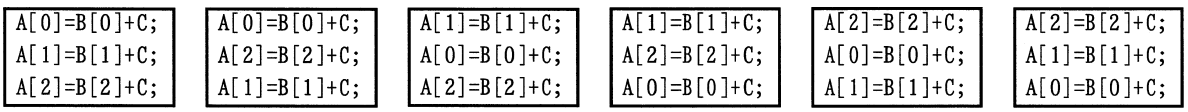


図 5-2-25 (1)　図 5-2-25 (2)　図 5-2-25 (3)　図 5-2-25 (4)　図 5-2-25 (5)　図 5-2-25 (6)

5-3　並列化率を高くする方法

本節では、5-1 節で述べた「(1) 速度向上率を上げるために、並列化率を高くする」方法を説明します。並列性のないプログラムを並列化することは不可能ですが、別の解法に変更して並列化できる場合もあります。

並列化できない部分の分離

図 5-3-1 (1) では、1 つのループの中に、並列性のある部分と並列性のない部分が混在しています。両者の間に依存関係がない場合、図 5-3-1 (2) のようにループを 2 つに分割すれば、並列性のある方のループを並列化することができます。図 5-3-1 (3) のように I/O 命令が含まれる場合も同様です。

```
   :
for(i=1;i<5;i++){
  X[i] = Y[i];    並列性あり
  A[i] = A[i-1]; 並列性なし
}
   :
```

図 5-3-1 (1)

```
   :
for(i=1;i<5;i++){
  X[i] = Y[i];     並列性あり
}
for(i=1;i<5;i++){
  A[i] = A[i-1]; 並列性なし
}
   :
```

図 5-3-1 (2)

```
   :
for(i=1;i<5;i++){
  X[i] = Y[i];      並列性あり
  printf(～,X[i]); 並列性なし
}
   :
```

図 5-3-1 (3)

ループを分割して並列化

図 5-3-3 (1) のループを逐次で実行した場合、図 5-3-4 (1) の①～⑧の順に実行が行われます。図 5-3-3 (1) を展開すると、図 5-3-2 に示すように後方依存性があるため、並列性はありません。もし図 5-3-3 (2) のように（無理に）並列に実行すると、図 5-3-4 (2) で CPU 1 の[2]が CPU 0 の③より先に実行された場合、下線部の結果が図 5-3-4 (1) と変わってしまいます。

この場合、図 5-3-3 (3) のようにループを 2 つに分割すると、図 5-3-4 (3) のように正しく実行が行われます。

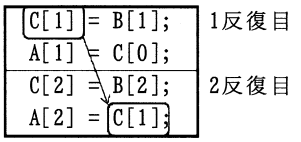


図 5-3-2

```
   :
for(i=1;i<5;i++){ 逐次
  C[i] = B[i];
  A[i] = C[i-1];
}
   :
```

図 5-3-3 (1)

```
   :
for(i=1;i<5;i++){ 並列
  C[i] = B[i];
  A[i] = C[i-1];
}
   :
```

図 5-3-3 (2)　×

```
   :
for(i=1;i<5;i++){ 並列
  C[i] = B[i];
}
for(i=1;i<5;i++){ 並列
  A[i] = C[i-1];
}
   :
```

図 5-3-3 (3)　○

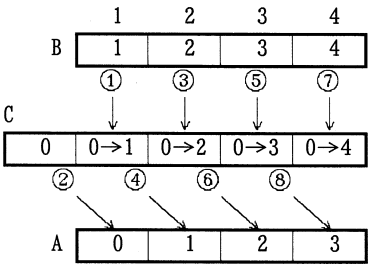


図 5-3-4 (1)

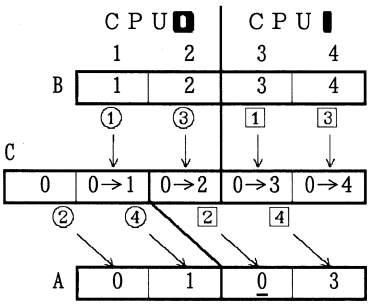


図 5-3-4 (2)

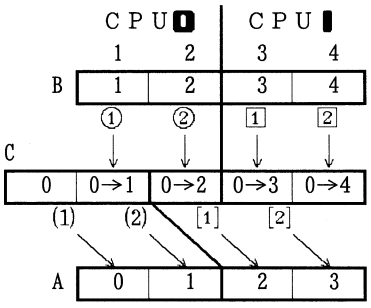


図 5-3-4 (3)

反復の途中で離脱するループ

図 5-3-5 (1) で、配列 INDEX には「0」か「1」が設定されています。ループが反復し、INDEX の値が「0」の間②の計算をし、「1」になったら①でループで離脱します。逐次で実行した場合の実行結果を図 5-3-6 (1) に示します。

このループを図 5-3-5 (2) のように並列化し、2 台の CPU で実行した場合、図 5-3-6 (2) のように下線部の計算結果が逐次の場合と変わってしまうので、並列化できません。ただし図 5-3-6 (3) に示すように、配列 INDEX の最初の「1」以降の要素を全て「1」に設定することができれば、図 5-3-5 (2) でも並列化できます。

別の並列化方法を図 5-3-5 (3) に示します。元のループを 2 つに分割し、最初のループでは、配列 INDEX の値が「1」になるループ反復 iend (本例では 3) を逐次処理で調べます。2 つ目のループでは、i = 0〜(iend-1) の範囲のみを並列に実行します。

```
   :
for(i=0;i<8;i++){ 逐次
  if(INDEX[i]==1) break;①
  M[i] = 9;              ②
}
   :
```

図 5-3-5 (1)

```
   :
for(i=0;i<8;i++){ 並列
  if(INDEX[i]==1) break;
  M[i] = 9;
}
   :
```

図 5-3-5 (2)　×（○）

```
   :
for(i=0;i<8;i++){ 逐次
  if(INDEX[i]==1) break;
}
iend = i;
for(i=0;i<iend;i++){ 並列
  M[i] = 9;
}
   :
```

図 5-3-5 (3)　○

	0	1	2	3	4	5	6	7
INDEX	0	0	0	1	0	0	0	0
M	9	9	9					

図 5-3-6 (1)

	CPU0				CPU1			
	0	1	2	3	4	5	6	7
INDEX	0	0	0	1	0	0	0	0
M	9	9	9		9	9	9	9

図 5-3-6 (2)　×

	CPU0				CPU1			
	0	1	2	3	4	5	6	7
INDEX	0	0	0	1	1	1	1	1
M	9	9	9					

図 5-3-6 (3)　○

CPU0　iend

	0	1	2	3	4	5	6	7
INDEX	0	0	0	1	0	0	0	0

iend

	0	1	2	3	4	5	6	7
M	9	9	9					
	CPU0		CPU1					

図 5-3-6 (4)

配列の添字を加算で決定するループ

図 5-3-7 (1) のループは、下線部が縮約演算なので、5-2 節で説明したように、このままでは並列化できません。図 5-3-7 (2) または図 5-3-7 (3) のようにすれば、縮約演算がなくなるので、並列化することができます。

```
   :
icount = 0;
for(i=0;i<6;i++){
  M[icount] = 1;
  icount = icount + 2;
}
   :
```

図 5-3-7 (1)　×

```
   :
for(i=0;i<6;i+=2){
  M[i] = 1;
}
   :
```

図 5-3-7 (2)　○

```
   :
for(i=0;i<6;i++){
  M[i*2] = 1;
}
   :
```

図 5-3-7 (3)　○

	0	1	2	3	4	5
M	0→1	0	0→1	0	0→1	0

	CPU0			CPU1		
	0	1	2	3	4	5
M	0→1	0	0→1	0	0→1	0

後方依存性のあるループ

図 5-3-8 (1) は、5-2 節で説明した後方依存性のあるループなので、このままでは並列化はできません。しかし不完全な並列化であれば可能です。図 5-3-8 (1) を逐次処理で計算した場合、図 5-3-8 (2) の①，②，... の順に計算が行われます。

これを例えば 3 台の CPU で、図 5-3-8 (3) に示すように並列に計算します。

- まず各 CPU は、①，(1)，1（太線）を並列に計算（加算）します。
- 計算が完全に終了したら、②，(2)，2（点線）を、② → (2) → 2 の順に逐次処理で計算（累積）します。
- 計算が完全に終了したら、各 CPU は③，(3)，3（細線）を、並列に計算（累積）します。

このうち②，(2)，2は逐次処理ですが、他の処理より計算量が少ないので、処理する要素数が多いときは無視することができます。各要素の加算を、①，(1)，1と、③，(3)，3の 2 回行っているので、②，(2)，2は無視した場合、並列化による速度向上率は 4 CPU で 2 倍、8 CPU で 4 倍となり、不完全な並列化となります (完全な並列化の場合は 4 CPU で 4 倍、8 CPU で 8 倍となります)。

```
   :
for(i=1;i<10;i++){
  A[i] = A[i-1] + B[i];
}
   :
```

図 5-3-8 (1)

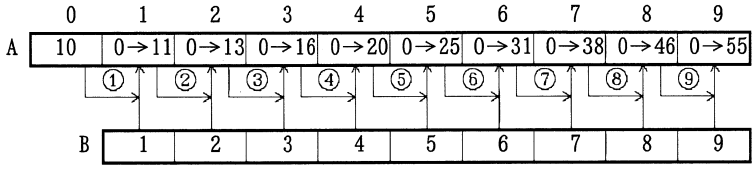


図 5-3-8 (2)

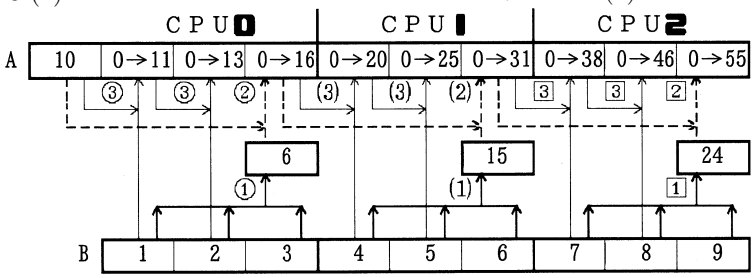


図 5-3-8 (3)

前方依存性のあるループ

図 5-3-9 (1) は、5-2 節で説明した前方依存性のあるループなので、このままでは並列化はできません。しかし、ロジックを少し変更すれば並列化が可能です。図 5-3-9 (1) を逐次処理で計算した場合、図 5-3-9 (2) の①，②，... の順に計算が行われます。

これを 3 台の CPU で図 5-3-9 (3) のように並列に計算します。まず各 CPU は、①，(1)，1で、`A[3]`，`A[6]`，`A[9]` の値を一時変数 `temp` にコピー（退避）します (太い ↓)。次に各 CPU は、②，(2)，2を並列に計算します。このとき `A[3]`，`A[6]`，`A[9]` の参照の代わりに、一時変数 `temp` の値を使用します (太い ←)。なお、`A[9]` は一時変数 `temp` にコピーしなくてもよいですが、全 CPU が同じ動作をした方がプログラムが簡単なので、コピーしています。

```
   :
for(i=0;i<9;i++){
  A[i] = A[i+1] + B[i];
}
   :
```

図 5-3-9 (1)

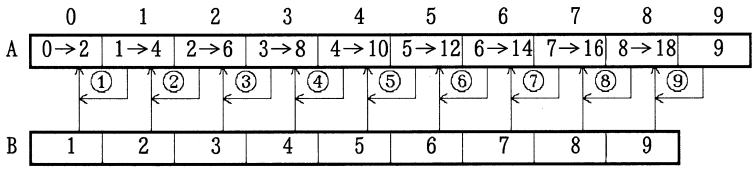


図 5-3-9 (2)

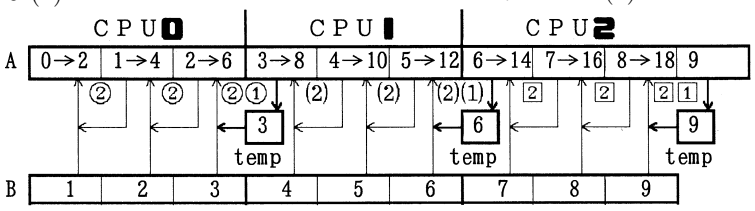


図 5-3-9 (3)

並列性のある解法への変更

図 5-3-10 (1) は、ヤコビ法を 1 次元化したループです。ヤコビ法は反復解法で、収束するまで図 5-3-10 (1) のループを何度も実行します (配列 B が次の反復での配列 A になります)。動作を図 5-3-10 (3) に示します。配列 A[1]～[8] の各要素の、ある反復での計算前の値を (1)～(8)、計算後の値を①～⑧で示します。また「境」は固定境界の値を示します。矢印は各要素の依存関係を示します。ヤコビ法は、図 5-3-10 (1) のループを展開した図 5-3-10 (2) から分かるように、依存関係がなく、並列化が可能です。

図 5-3-11 (1) は、ヤコビ法の改良版であるガウスザイデル法 (または SOR 法) を簡単化したループです。動作を図 5-3-11 (3) に示します。前述のヤコビ法では、図 5-3-10 (3) の〓に示すように、②を計算するのに、計算前の値である (1) と (3) を使用しました。ガウスザイデル法では、図 5-3-11 (3) の〓に示すように、計算後の①と計算前の (3) を使用します。このため、ヤコビ法よりも収束が速くなります。

ガウスザイデル法は、図 5-3-11 (1) を展開した図 5-3-11 (2) の矢印に示すように、後方依存性 (↘) と前方依存性があり (↙)、並列性がありません。これを図 5-3-11 (3) で確認してみます。CPU 2 台で並列に実行し、CPU 0 が④を計算する前に CPU 1 が⑤を計算した場合、⑤を、計算後の④でなく計算前の (4) を使用して計算し、計算結果が逐次処理の場合と変わってしまうため、並列性がありません。

図 5-3-12 (1) は、ガウスザイデル法 (または SOR 法) を、マルチカラー法（またはレッドブラック法）という方法に変更したループです。動作を図 5-3-12 (3) に示します。1のループの 1 反復目 (k = 0) に、〓に示す①，③，⑤，⑦を計算し、2 反復目 (k = 1) に②，④，⑥，⑧を計算します。ガウスザイデル法 (または SOR 法) と計算順序が異なっており、計算結果も同じにはなりませんが、収束計算なので問題ありません。

マルチカラー法は、図 5-3-12 (1) を展開した図 5-3-12 (2) に示すように、1のループの 1 反復目内、および 2 反復目内の各ステートメント間に依存関係はありません。従って2のループは並列に実行することができます。ただし、1のループの 1 反復目と 2 反復目間に、図 5-3-12 (2) の〓に示す依存関係があるので、1 反復目が完全に終了してから 2 反復目を実行するように、1 反復目と 2 反復目の間で同期を取る必要があります。

以上をまとめます。ガウスザイデル法のように並列性のない解法を、並列性のある解法に変更できる場合があります。例えばヤコビ法は、収束が遅いので逐次処理では使われませんが、並列性があります。またマルチカラー法は、ガウスザイデル法と収束性はそれほど変わらず、並列性があります。

```
    :
for(i=1;i<9;i++){
  B[i] = A[i-1] + A[i+1];
}
    :
```

図 5-3-10 (1)　○

```
B[1] = A[0] + A[2]

B[2] = A[1] + A[3]
```

図 5-3-10 (2)

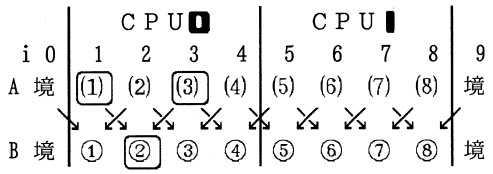


図 5-3-10 (3)

```
    :
for(i=1;i<9;i++){
  A[i] = A[i-1] + A[i+1];
}
    :
```

図 5-3-11 (1)　×

```
A[1] = A[0] + A[2]

A[2] = A[1] + A[3]
```

図 5-3-11 (2)

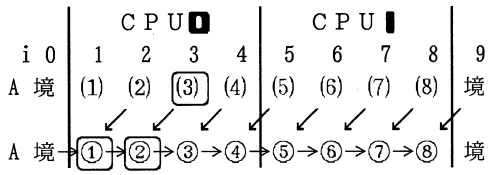


図 5-3-11 (3)

```
    :
for(k=0;k<=1;k++){        1
  for(i=1+k;i<9;i+=2){    2
    A[i] = A[i-1] + A[i+1];
  }
}
    :
```

図 5-3-12 (1)　○

```
A[1] = A[0] + A[2]
A[3] = A[2] + A[4]
A[5] = A[4] + A[6]
A[7] = A[6] + A[8]
同期を取ります。
A[2] = A[1] + A[3];
A[4] = A[3] + A[5];
A[6] = A[5] + A[7];
A[8] = A[7] + A[9];
同期を取ります。
```

図 5-3-12 (2)

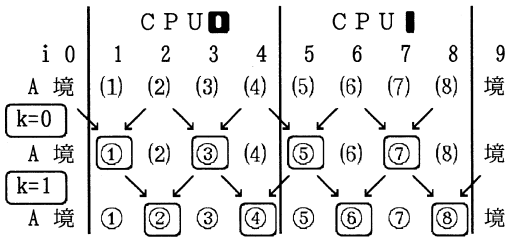


図 5-3-12 (3)

5-4　ロードバランスを均等にする方法

本節では、5-1 節で述べた「(2) 速度向上率を上げるために、CPU 間のロードバランスを均等にする」方法を説明します。

図 5-4-1 (1) のループを 2 CPU で並列に計算する場合、通常のループであれば、図 5-4-1 (2) (3) に示すように各 CPU が配列 A の要素を 1/2 ずつ処理すれば、各 CPU のロードバランスは均等になります。ところが以下で説明するように、均等にならないループもあります。

なお、並列化するため、図 5-4-2 (2) では「ループを 1/2 に分割」しており、図 5-4-2 (3) では「配列を 1/2 に分割」しています。通常、両者は同じ意味なので、本節では主に配列の図を使用して説明します。

```
    :
for(i=0;i<6){
  A[i] = A[i] + 1.0f;
}
    :
```

図 5-4-1 (1)

```
        CPU0                          CPU1
    :
for(i=0;i<3){                 for(i=3;i<6){
  A[i] = A[i] + 1.0f;           A[i] = A[i] + 1.0f;
}                             }
    :
```

図 5-4-1 (2)

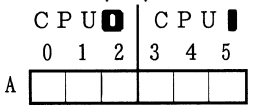


図 5-4-1 (3)

ループ／配列の分割方法

ループあるいは配列を分割する方法として、以下の方法があります。

(1)　ブロック分割

配列が 1 次元の場合、図 5-4-2 のように分割する方法をブロック分割と呼びます。並列化する場合、ブロック分割が最も多く用いられます。

配列が 2 次元の場合、図 5-4-3 (1) (2) に示すように 1 つの次元で分割する方法と、図 5-4-4 (1) (2) に示すように 2 つの次元で分割する方法があります。

1 つの次元で分割した場合、図 5-4-3 (2) (3) から分かるように、CPU の台数を増やしても、1 つの CPU が担当する領域の境界の長さ（二重線）は変わりません。一方 2 つの次元で分割した場合、図 5-4-4 (2) (3) から分かるように、CPU の台数を増やすと、1 つの CPU が担当する領域の境界の長さ（二重線）は短くなります。

従って、MPI 並列で、境界の部分で他の CPU との通信が発生し、CPU の台数が多い場合、図 5-4-3 (3) よりも図 5-4-4 (3) の方が、（境界の長さが短いので）通信量が少なくなり、通信時間も短くなります。ただし図 5-4-3 (3) よりも図 5-4-4 (3) の方が境界の数が 2 倍になるので、通信回数は 2 倍になり、通信の立上り時間のオーバーヘッド (5-5 節参照) が増えます。

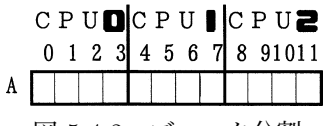


図 5-4-2　ブロック分割

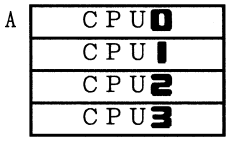


図 5-4-3 (1)

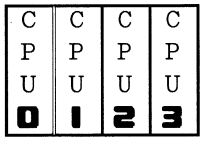


図 5-4-3 (2)

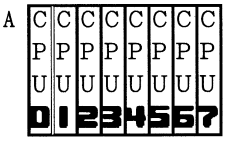


図 5-4-3 (3)

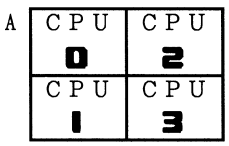


図 5-4-4 (1)

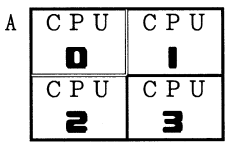


図 5-4-4 (2)

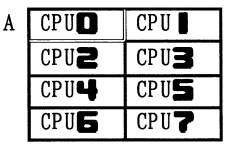


図 5-4-4 (3)

(2)　サイクリック分割／ブロックサイクリック分割

CPU 0, 1, 2 が、1 次元配列の各要素を、図 5-4-6 (1) のように担当するとします。このように、1 要素ずつ、異なる CPU が順繰りに担当する分割方法をサイクリック分割と呼びます。ブロック分割では何らかの理由で各 CPU のロードバランスが不均等になる場合、サイクリック分割で均等化できることがあります（例は後述します）。

図 5-4-7 (1) の〓に示すように、複数要素（同じ数）ずつ、異なる CPU が順繰りに担当する分割方法をブロックサイクリック分割と呼びます。ブロックサイクリック分割は、本当はサイクリック分割にしたいけれども、サイクリック分割にすると別の問題が発生する場合、サイクリック性を多少犠牲にする代わりに、別の問題を低減する目的で用いられます。例を 2 つ示します。

図 5-4-6 (1) の場合、CPU 0 は図 5-4-6 (2) の①，②，③，④の順に要素を処理します。例えば矢印に示すように 3 つ飛び以上のストライド（間隔）で要素を処理した場合、キャッシュミスが発生して速度が低下するとします。図 5-4-6 (2) では矢印が 3 本なので、3 回キャッシュミスが発生します。一方図 5-4-7 (2) では、矢印は 1 本なので、キャッシュミスは 1 回に減少します（ただしサイクリック性が若干失われます）。

他の例として、MPI 並列で、他の CPU が担当する要素との境界で、通信が必要になるとします。例えば CPU 0 が担当する要素の境界 (↔) の数は、図 5-4-6 (3) では多い (7 個) ですが、図 5-4-7 (3) では少なく (3 個) なるので、通信によるオーバーヘッドが減少します（ただしサイクリック性が若干失われます）。

サイクリック分割、ブロックサイクリック分割は、2 次元配列でも可能です。図 5-4-8 (1) は、縦、横ともサイクリック分割、図 5-4-8 (2) は、縦はブロック、横はサイクリック分割、図 5-4-8 (3) は、縦、横ともブロックサイクリック分割の例です。

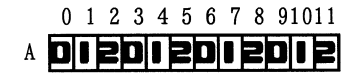


図 5-4-6 (1)　サイクリック分割

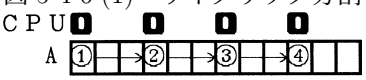


図 5-4-6 (2)　サイクリック分割

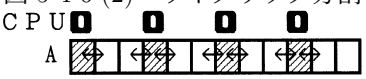


図 5-4-6 (3)　サイクリック分割

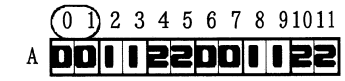


図 5-4-7 (1)　ブロックサイクリック分割

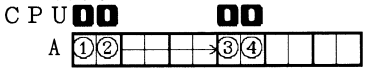


図 5-4-7 (2)　ブロックサイクリック分割

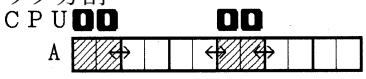


図 5-4-7 (3)　ブロックサイクリック分割

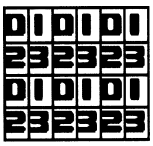


図 5-4-8 (1)

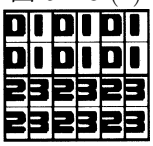


図 5-4-8 (2)

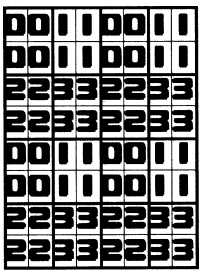


図 5-4-8 (3)

ブロック分割ではロードバランスが均等にならない例

普通のループであれば、ブロック分割にすれば、CPU 間のロードバランスは、ほぼ均等になります。以下では、ブロック分割でロードバランスが均等にならない例を紹介します。

(1)　場所によって計算量の分布に偏りがある場合 (1)

図 5-4-9 (1) の 2 次元配列の中で、実際に計算するのは■の要素のみだとします（例えば湖の計算など）。図 5-4-9 (2) に示すように、配列の大きさで単純にブロック分割すると、各 CPU が担当する要素数が不均等になります。実際に計算する要素数 (本例では 27 個) が既知の場合、図 5-4-9 (3) に示すように、各 CPU が担当する要素数がほぼ均等になるようにブロック分割すれば、CPU 間のロードバランスはほぼ均等になります。

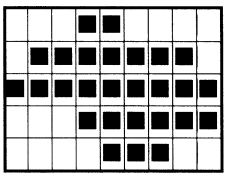


図 5-4-9 (1)

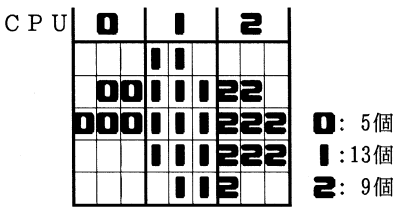


図 5-4-9 (2)　×

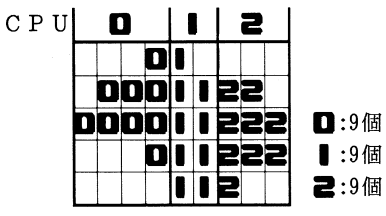


図 5-4-9 (3)　○

(2)　場所によって計算量の分布に偏りがある場合 (2)

図 5-4-10 (1) の配列内で、■の部分は□の部分よりも計算量が多いとします。また、■の部分がどのように分布するかが、計算前には分からないとします。図 5-4-10 (2) に示すように、配列の大きさで単純にブロック分割すると、各 CPU が担当する要素数が不均等になる可能性があります。

この場合、図 5-4-10 (3) のようにサイクリック分割にすれば、各列の計算量はばらついていても、各列の計算量を合計すると均等化されるので、多くの場合、CPU 間のロードバランスは、ほぼ均等になります。ただし境界の数が増えるので、MPI 並列で、境界間の通信が必要な場合はオーバーヘッドが多くなります。

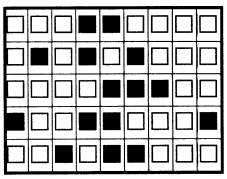


図 5-4-10 (1)

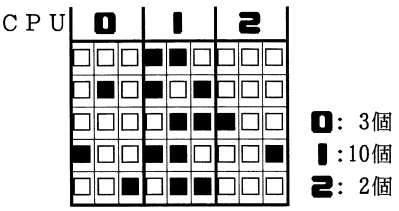


図 5-4-10 (2)　×

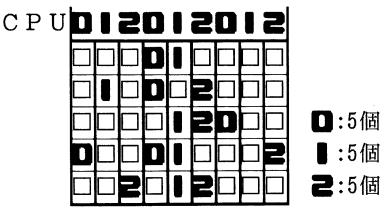


図 5-4-10 (3)　○

(3)　場所によって計算量の分布に偏りがある場合 (3)

図 5-4-11 (1) では箱の中に粒子が入っており、各粒子の計算量は、上下左右の隣接するマス内の粒子数に比例するとします（マスは実際には存在しません)。例えば②は、この粒子の計算量が 2 (②の粒子に隣接する粒子が 2 個) であることを示します。本例では中央付近に粒子が多いので、中央付近の粒子は計算量が多くなります。

図 5-4-11 (2) に示すように、箱の左から順に粒子数でブロック分割すると、中央付近の粒子を担当する CPU 1 の計算量が多くなり、CPU 間のロードバランスが不均等になります。この場合、図 5-4-11 (3) に示すように、箱の左から順に粒子をサイクリック分割にすれば、多くの場合、CPU 間のロードバランスは、ほぼ均等になります。

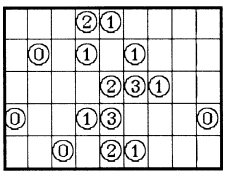


図 5-4-11 (1)

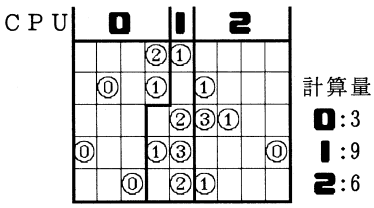


図 5-4-11 (2)　×

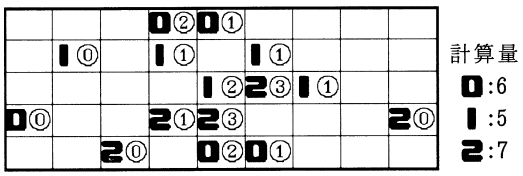


図 5-4-11 (3)　○

(4)　計算が進むにつれて計算量の分布が変化する場合

図 5-4-12 (1) では、計算が進むにつれて、計算領域が次第に小さくなります (例えば連立一次方程式の LU 分解など)。図 5-4-12 (1) のようにブロック分割にした場合、計算が進むと CPU 0 の計算部分が終了し、さらに計算が進むと CPU 1 の計算部分も終了するので、次第に CPU 間のロードバランスが不均等になります。

一方サイクリック分割にした図 5-4-12 (2) では、計算が進んでも、常にほぼロードバランスが均等になります。このように、計算が進むにつれて、計算領域の範囲が変化する場合、サイクリック分割にすると、CPU 間のロードバランスをほぼ均等できる場合があります。

図 5-4-12 (3) の例では、計算が進むにつれて、計算領域の大きさは変化しませんが、計算量の多い部分（■）と少ない部分（□）の分布が変化します。この場合もロードバランスを均等にするためにはサイクリック分割が有効です。

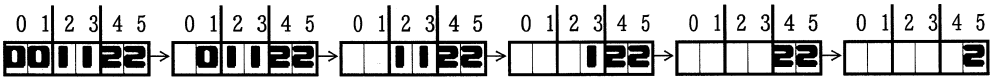


図 5-4-12 (1)　×

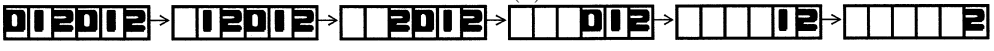


図 5-4-12 (2)　○

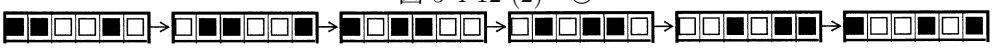


図 5-4-12 (3)

マススレーブ（マスターワーカー）方式

前述のブロック分割やサイクリック分割では、並列部分の計算を開始する前に、どの CPU がどの要素を担当するかが決まっていました。以下で説明するマススレーブ（マスターワーカー）方式では、並列部分の実行中に、どの CPU がどの要素を担当するかが動的に決まります。

マススレーブ方式では、1 台のマスター CPU（係長に相当）と、複数台のスレーブ CPU（平社員に相当）を使用します。マスター CPU はスレーブ CPU に仕事を与え、スレーブ CPU は仕事を行い、終了したらマスター CPU に報告します。マスター CPU はそのスレーブ CPU に、次の仕事を与えます。

図 5-4-13 (1) では、ループの各反復 (⓪〜⑧) での計算量が不規則です。そして、前述のブロック分割でもサイクリック分割でも、CPU 間のロードバランスが均等になりにくいとします。このような場合、マススレーブ方式で、ロードバランスを均等にできる場合があります。

図 5-4-13 (1) をマススレーブ方式で実行した場合のタイムチャートを、図 5-4-13 (2) に示します。

- 図 5-4-13 (1) で、マスター CPU は、まずスレーブ CPU 0，1，2 に、⓪，①，②反復目の処理を指示します。
- 例えば CPU 0 は、図 5-4-13 (2) の ↓ の時点で、⓪反復目の処理が完了します。
- CPU 0 の以後の動作を、図 5-4-13 (3) で説明します。まず (1) に示すように、CPU 0 はマスター CPU に、⓪反復目が終了したことを報告します。
- 図 5-4-13 (3) の左下の表は、各ループ反復の状態（未処理、処理中、処理済み）を示します。マスター CPU は (2) で、⓪反復目を「処理中」から「処理済」に変更します。
- マスター CPU は (3) で、未処理の⑤反復目を選択し、(4) で、⑤反復目の処理を CPU 0 に指示します。
- CPU 0 は (5) で、⑤反復目の処理を開始します。

このように、各スレーブ CPU には「終わったら次の処理、終わったら次の処理」と絶え間なく未処理の反復が与えられるので、プログラムのほぼ最後まで、遊んでいる CPU はなくなり、各 CPU のロードバランスはほぼ均等になります。

図 5-4-13 (1) (2) では、ループの各反復の計算量が異なっていて、各 CPU の処理能力は同じでした。逆に図 5-4-14 (1) (2) のように、ループの各反復の計算量は同じで、各 CPU の処理能力が異なっている場合 (速い CPU と遅い CPU が混在、あるいは、すいている CPU と混んでいる CPU が混在) でも同様に、マススレーブ方式でロードバランスを均等にすることができる場合があります。

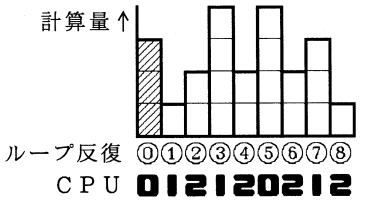


図 5-4-13 (1)

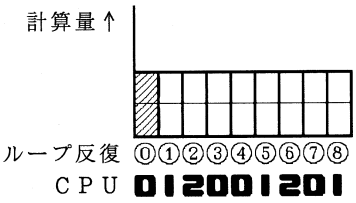


図 5-4-14 (1)

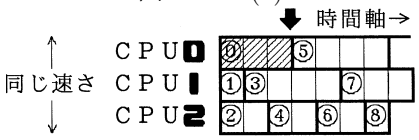


図 5-4-13 (2)

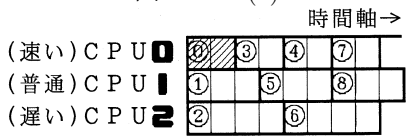


図 5-4-14 (2)

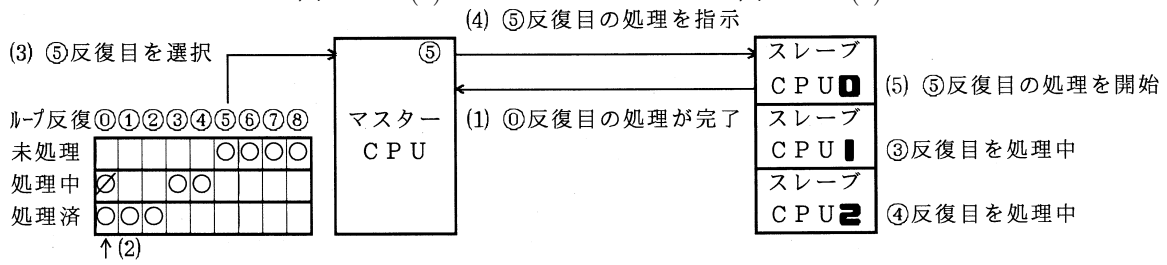


図 5-4-13 (3)

5-5　オーバーヘッドを低減する方法

本節では、5-1 節で述べた「(3) 速度向上率を上げるために、並列化に伴うオーバーヘッドを少なくする」方法を説明します。並列計算の種類によって、発生するオーバーヘッドが異なるので、以下では GPU に関係するオーバーヘッドを取り上げます。

カーネル関数の呼び出しのオーバーヘッド

GPU では、カーネル関数を 1 回呼ぶたびにオーバーヘッドが発生します。5-1 節で紹介したスレッド並列でも同様のオーバーヘッドが発生するので、以下では図で説明しやすいスレッド並列で説明します。

図 5-5-1 (2) の 2 重ループで、内側のループ (①) でも外側のループ (②) でも並列性があるとします。例えば 2 CPU で並列に実行した場合、①で並列化すると、図 5-5-2 (1) に示すように、内側のループが 2 つの CPU で実行されます。一方②で並列化すると、図 5-5-2 (2) に示すように、外側のループが 2 CPU で実行されます。

このとき、図の「↙↘」の部分でオーバーヘッドが発生します (「↘↙」の部分でも若干発生します)。「↙↘」は、図 5-5-2 (1) では 100 回実行され、図 5-5-2 (2) では 1 回実行されるので、図 5-5-2 (1) の方がオーバーヘッドの回数が多く、時間がかかります。従って、多重ループはなるべく外側の②のループで並列化して下さい。なお、もし図 5-5-1 (1) の③で並列化できれば、「↙↘」の回数が最も少なくなります。

別の例を示します。図 5-5-3 (1) では、2 つのループを別々に並列化しているので、「↙↘」のオーバーヘッドが 2 回かかります。もし 2 つのループを、図 5-5-3 (2) のように合体することができれば、オーバーヘッドは 1 回になります。ただし、2 つのループが合体できるのは、ループ間に依存関係がない場合のみです。

GPU の場合も同様に、可能であればカーネル関数の呼び出し回数がなるべく少なくなるようにして下さい。

```
void main(){
    :
  for(k=0;k<1000;k++){  ← ③
      :
    func();
      :
  }
    :
```

=

```
void func(){
    :
  for(iy=0;iy<100;iy++){  ← ②
    for(ix=0;ix<100;ix++){  ← ①
      A[iy][ix] = ～;
    }
  }
    :
```

図 5-5-1 (1)　　図 5-5-1 (2)

```
    :
for(iy=0;iy<100;iy++){ ②
        CPU❶                    CPU❷
  for(ix=0;ix<50;ix++){ ①   for(ix=50;ix<100;ix++){①
    A[iy][ix] = ～;            A[iy][ix] = ～;
  }                          }
}
    :
```

図 5-5-2 (1)　①で並列化　×

```
    :
  CPU❶                      CPU❷
for(iy=0;iy<50;iy++){   ②  for(iy=50;iy<100;iy++){ ②
  for(ix=0;ix<100;ix++){ ①   for(ix=0;ix<100;ix++){ ①
    A[iy][ix] = ～;            A[iy][ix] = ～;
  }                          }
}
    :
```

図 5-5-2 (2)　②で並列化　○

```
    :
for(i=0;i<100;i++){  ← 並列
  A[i] = A[i] + 1.0f;
}
for(i=0;i<100;i++){  ← 並列
  B[i] = B[i] + 1.0f;
}
    :
```

図 5-5-3 (1)　×

```
    :
for(i=0;i<100;i++){  ← 並列
  A[i] = A[i] + 1.0f;
  B[i] = B[i] + 1.0f;
}
    :
```

図 5-5-3 (2)　○

コピー／トランザクションのロード・ストアのオーバーヘッド

GPU では、ホストとデバイス間のデータのコピー (CUDA 関数 cudaMemcpy など) にオーバーヘッドがかかり、通常この部分が GPU の一番のボトルネックになります。図 5-5-4 に示すように、コピー時間は以下のようになります。

全コピー時間 = (1) 立上り時間 + (2) コピー自体の時間

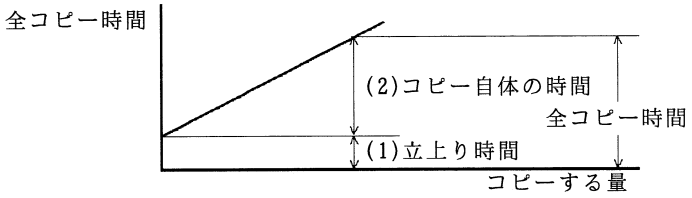


図 5-5-4

(1) 一回コピーを行うごとに、一定の立上り時間 (latency: レイテンシー) がかかります。たとえ 1 要素しかコピーしなくても立上り時間がかかります。従って、コピーする量が同じならば、コピーする回数はなるべく少なくなるようにして下さい。

(2) コピー自体の時間はコピーする量に比例します。つまり、コピーする量が 2 倍なら、コピー自体の時間も 2 倍になります。従って、計算に必要な最小限のデータのみをコピーするようにして下さい。

コピーの他に、3-2 節で説明したグローバルメモリからのトランザクションのロード・ストアの場合も同様のオーバーヘッドが発生します。従って、トランザクションをロード／ストアする回数が最も少なく、トランザクションの大きさが最も小さい場合に、コアレスアクセスが最も効率よく行われます。

(1) (2) を少なくするための方法を以下で説明します。

(1)　コピーする量が同じならばコピーする回数を少なくする

- 図 5-5-5 (1) や図 5-5-5 (2) に示すように、メモリ上で飛び飛びのデータを 1 要素ずつコピーすると、コピーするたびに立上り時間 (→ の部分) がかかってしまいます。図 5-5-5 (3) のように、コピーしたい要素のみを 1 箇所に集め、1 回でコピーすれば、立上り時間は 1 回で済みます。
  なお、飛び飛びの要素のまま 1 回でコピーする CUDA 関数 (cudaMemcpy2D など：3-3 節参照) も提供されており、これを使用しても立上り時間は 1 回で済みます。

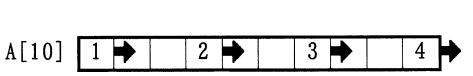


図 5-5-5 (1)　×

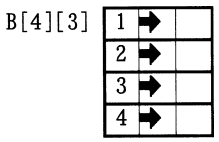


図 5-5-5 (2)　×

＝

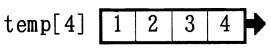


図 5-5-5 (3)　○

- 図 5-5-6 (1) のように、コピーをループ内で何度も行うと立上り時間がかかるので、図 5-5-6 (2) のように、コピーをループの外で 1 回だけ行うようにします。

```
   :
for(i=0;i<N;i++){
  X[i] = ～;
  cudaMemcpy(～,&X[i],sizeof(float),～);
}
   :
```

図 5-5-6 (1)　×

```
   :
for(i=0;i<N;i++){
  X[i] = ～;
}
cudaMemcpy(～,X,N*sizeof(float),～);
   :
```

図 5-5-6 (2)　○

(2)　必要最小限のデータのみをコピーする

- 図 5-5-7 (1) を CUDA 化する場合、図 5-5-7 (2) のようにカーネル関数呼び出しの前後で毎回配列 A のコピーを行うと、コピーする量（以下コピー量）と計算量のオーダーが共に N^2 なので、あまり効果は出ません。
  図 5-5-8 (1) (2) の行列乗算の場合は、コピー量のオーダーが N^2、計算量のオーダーが N^3 で、コピー量の方が次数が少ないので、CUDA 化すると（一般に）効果が出ます。また、行列のサイズが大きくなればなるほど、以下に示すように、計算に対するコピーの比率が少なくなり、効果が高くなります。

  行列のサイズが $N \times N$ の場合：(コピー量のオーダー)/(計算量のオーダー) $= N^2/N^3 = 1/n$
  行列のサイズが $(2N) \times (2N)$ の場合：(コピー量のオーダー)/(計算量のオーダー) $= (2N)^2/(2N)^3 = 1/(2n)$

```
float A[N][N];
  :
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    A[i][j] = A[i][j] + 1.0f;
  }
}
  :
```

図 5-5-7 (1)　×

```
  :
cudaMemcpy(dA,A,～);  ｺﾋﾟｰ量:N²のｵｰﾀﾞｰ
kernel<<<～>>>(dA);   計算量:N²のｵｰﾀﾞｰ
cudaMemcpy(A,dA,～);  ｺﾋﾟｰ量:N²のｵｰﾀﾞｰ
  :
```

図 5-5-7 (2)　×

```
float A[N][N],B[N][N],C[N][N];
  :
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    float sum = 0.0f;
    for(k=0;k<N;k++){
      sum = sum + A[i][k]*B[k][j];
    }
    C[i][j] = sum;
  }
}
  :
```

図 5-5-8 (1)　○

```
  :
cudaMemcpy(dA,A,～);          ｺﾋﾟｰ量:N²のｵｰﾀﾞｰ
cudaMemcpy(dB,B,～);          ｺﾋﾟｰ量:N²のｵｰﾀﾞｰ
kernel<<<～>>>(dA,dB,dC); 計算量:N³のｵｰﾀﾞｰ
cudaMemcpy(C,dC,～);          ｺﾋﾟｰ量:N²のｵｰﾀﾞｰ
  :
```

図 5-5-8 (2)　○

- 図 5-5-9 (1) のようにタイムステップループごとにコピーするのではなく、可能であれば図 5-5-9 (2) のように、一度デバイス側にコピーしたデータをデバイス側で何度も使用し、最後にホスト側にコピーします。

```
  :
for(ﾀｲﾑｽﾃｯﾌﾟﾙｰﾌﾟ){
  cudaMemcpy(dA,A,～);
  kernel<<<～>>>();
  cudaMemcpy(A,dA,～);
}
  :
```

図 5-5-9 (1)　×

```
  :
cudaMemcpy(dA,A,～);
for(ﾀｲﾑｽﾃｯﾌﾟﾙｰﾌﾟ){
  kernel<<<～>>>();
}
cudaMemcpy(A,dA,～);
  :
```

図 5-5-9 (2)　○

- 図 5-5-10 (1) では、CPU 側で計算した結果をデバイス側にコピーしていますが、図 5-5-10 (2) のように、GPU 側で CPU 側と同じ計算を行い (GPU は高速なので)、コピーをなくす方法もあります。

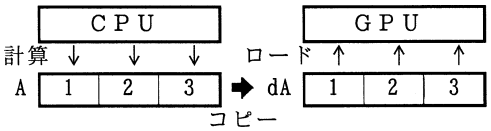


図 5-5-10 (1)　×

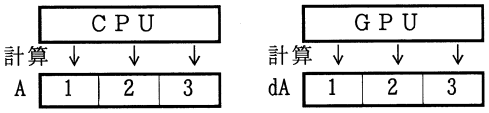


図 5-5-10 (2)　○

第6章　高速化編（各種高速化技法）

本章では、CUDA プログラムの速度を向上させるための各種方法について説明します。

6-1　ブロック数とスレッド数の設定 (2)（占有率計算機）

2-5 節で、ブロック数とブロック内のスレッド数の設定方法を説明しました。本節では、ストリーミング・マルチプロセッサー上に同時に存在するワープ数を考慮して、これらの数を決定する方法を説明します。

計算とロード／ストアのオーバーラップによるロード／ストア時間の隠蔽

まず、グローバルメモリ内の変数や配列の、ロード／ストア時間の隠蔽について説明します。

通常の CPU（またはコア）で並列プログラムを実行する場合、「使用する CPU 数 = プロセス数」または、「使用するコア数 = スレッド数」となります。また、プロセス（またはスレッド）の切り換えに時間がかかるため（注)、図 6-1-1 (1) (2) に示すように、ある CPU で実行に入ったプロセスは、途中で他の CPU に移動しないのが一般的です（スレッドの場合は多少移動する場合もあります)。

一方、GPU で CUDA 化したプログラムを実行する場合、一般に「使用する CUDA コア数 < スレッド数」となります。また GPU では、スレッドの切り換えに時間がかからないため（注)、図 6-1-1 (3) に示すように、1 つの CUDA コアで動作するスレッド（正確にはワープ）を頻繁に切り換えても問題ありません。GPU では、この性質を利用して、以下で説明するように速度を速くすることができます。

（注）CPU やコアでは、プロセスやスレッドの切り換えで、レジスターの内容をメモリに保存／復元します。GPU ではレジスターが多く、メモリへの保存や復元が必要ないので、切り換えに時間がかかりません。

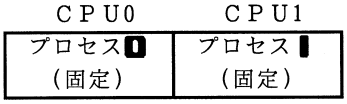


図 6-1-1 (1)　MPI 並列

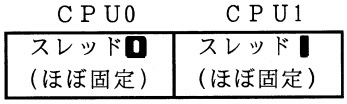


図 6-1-1 (2)　スレッド並列

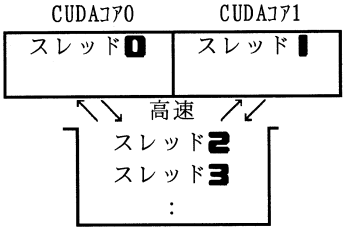


図 6-1-1 (3)

図 6-1-2 (1) の CUDA プログラムを、⓪に示すように、1 ブロック、ブロック内のスレッド数 32 (= 1 ワープ) で実行した場合の、そのワープが動作するストリーミング・マルチプロセッサーのタイムチャートの例を、図 6-1-2 (2) に示します (2-4 節参照)。図中の「↓」はスレッドを表します。

図 6-1-2 (1) の①で、各スレッドは、グローバルメモリ上にある配列 dA の、自分が担当する要素 dA[i] をロード、加算、ストアします。これを図 6-1-2 (1) の (1), 1, [1] に示します。以後、図 6-1-2 (1) の②, ③, ④で同様の処理が行われます。

図 6-1-2 (2) の点線の部分 (例えば [1], (2)) では、ロード／ストアはハーフワープ単位に行われ (3-2 節参照)、この間、CUDA コアは動作していません。ロード／ストアの時間は、図では短いように見えますが、実際には加算と比べてはるかに時間がかかります。従って、点線の部分で CUDA コアを動作させることができれば、速度が向上します。

図 6-1-3 (1) では、⑤に示すように、1 ブロック、ブロック内のスレッド数 64 (= 2 ワープ) で実行します。2 つのワープは同一ブロック上に含まれるので、同じストリーミング・マルチプロセッサー上で実行されます。図 6-1-3 (1) で、ワープ 0 が⑥の実行を開始した直後にワープ 1 が⑥を開始した場合のタイムチャートの例を、図 6-1-3 (2) に示します。

図から分かるように、例えばワープ 0 の [6], (7) のストア／ロードと、ワープ 1 の6の加算がオーバーラップ（同時に動くこと）しています。これによって、時間のかかるグローバルメモリのロード／ストアの時間 ([6], (7)) を隠蔽することができ、（本例の場合）CUCA コアが常に動作します。

このように、1 つのストリーミング・マルチプロセッサー上に同時に存在できるワープ数が多いと、CUDA コアが動作する時間が長くなって速度が向上する可能性が高くなります。

```
__global__ void kernel(){
    :
  dA[i] = dA[i] + 1.0f  ①
  dB[i] = dB[i] + 2.0f  ②
  dC[i] = dC[i] + 3.0f  ③
  dD[i] = dD[i] + 4.0f  ④
    :
int main(void){
    :
  kernel<<<1,32>>>();   ⓪
    :
```

図 6-1-2 (1)

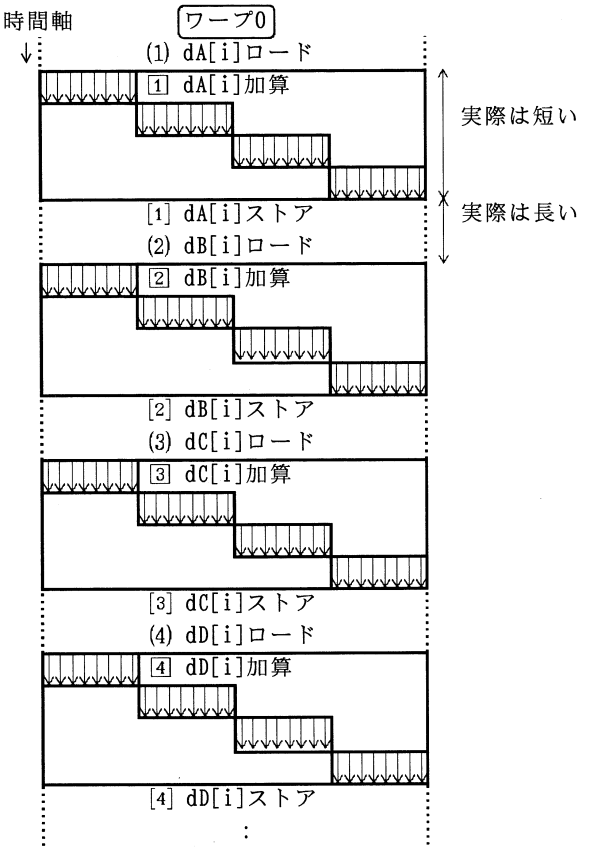


図 6-1-2 (2)

```
__global__ void kernel(){
    :
  dA[i] = dA[i] + 1.0f  ⑥
  dB[i] = dB[i] + 2.0f  ⑦
  dC[i] = dC[i] + 3.0f  ⑧
  dD[i] = dD[i] + 4.0f  ⑨
    :
int main(void){
    :
  kernel<<<1,64>>>();   ⑤
    :
```

図 6-1-3 (1)

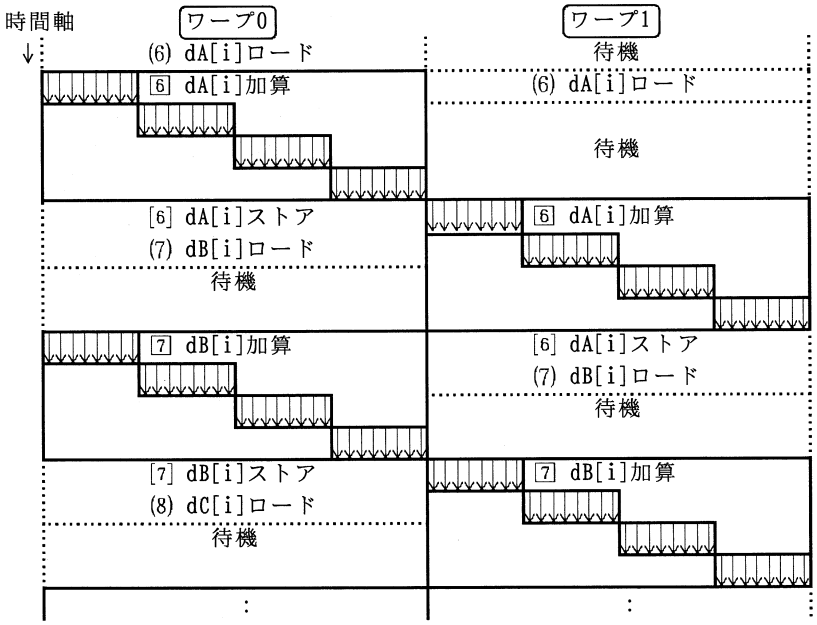


図 6-1-3 (2)

1 つのストリーミング・マルチプロセッサー上に同時に存在できるワープ数の制限

以下で、1 つのストリーミング・マルチプロセッサー上に同時に存在できるワープ数を多くする方法について説明します。「CUDA C Programming Guide」G.1 節によると、CUDA (Compute Capability 1.3) では以下の制限があります。

【制限 1】1 つのブロック内の最大スレッド数は 512 個です。つまり、1 つのブロックに入る最大ワープ数は 16 (= 512 ÷ 32) 個です。

【制限 2】1 つのストリーミング・マルチプロセッサー上に同時に存在できる最大ブロック数は 8 個です。

【制限 3】1 つのストリーミング・マルチプロセッサー上に同時に存在できる最大ワープ数は 32 個です。

図 6-1-4 に示すように、ブロックを□、ワープを「o」で表した場合、上記の 3 つの制限は以下のように言い換えることができます。

【制限 1】1 つの□の中に「o」が最大 16 個入ります。

【制限 2】1 つのストリーミング・マルチプロセッサー上に同時に存在できる最大の□の数は 8 個です (なお、各□での「o」の数は同一です)。

【制限 3】1 つのストリーミング・マルチプロセッサー上に同時に存在できる最大の「o」の数は 32 個です。

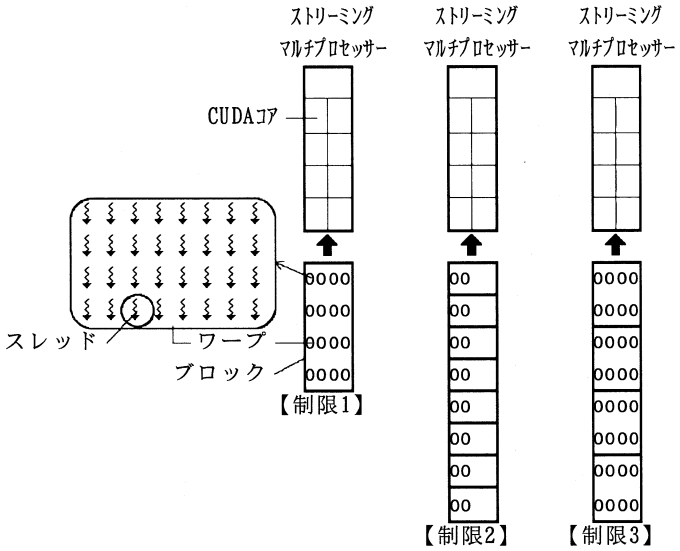


図 6-1-4

上記の 3 つの制限を満足する、1 つのストリーミング・マルチプロセッサーあたりに同時に存在できるブロック、ワープの状態を、図 6-1-5 の [1]〜[16] に示します。

- 図の左から順に、1 つのブロック内のワープ数が、1，2，...，16 個の場合を示します。
- 例えば [1] の上側の図はブロック数が最大 (8 個) の場合、下側の図はブロック数が 1 個の場合を示します。

図 6-1-5 が、上記 3 つの制限を満足することを確認します。

- 各ケースとも、1 つの□内の「o」の数が 16 個以下なので、上記の【制限 1】を満足します。
- 各ケースとも、□の数は 8 個以下なので、上記の【制限 2】を満足します。
- 各ケースとも、全ての□内の「o」の合計は 32 個以下なので、上記の【制限 3】を満足します。

図 6-1-5 をまとめると図 6-1-6 になります。図 6-1-5 の各ケースでの、1 つのストリーミング・マルチプロセッサーあたりに同時に存在できる全ワープ数を、図 6-1-6 の○内の数字で表します。例えば図 6-1-5 の★のケースでは、ブロック数は 2 個、ブロック内のワープ数は [11] 個なので、図 6-1-6 の★は㉒ ($= 2 \times [11]$) 個となります。

図 6-1-6 から、例えば 1 つのブロックあたりのスレッド数を 512 に設定した場合、ケース [16] の〓に示すように、1 つのストリーミング・マルチプロセッサーあたりに同時に存在できるブロック数は 1 か 2 のいずれか、ワープ数は⑯か㉜のいずれかになることが分かります。

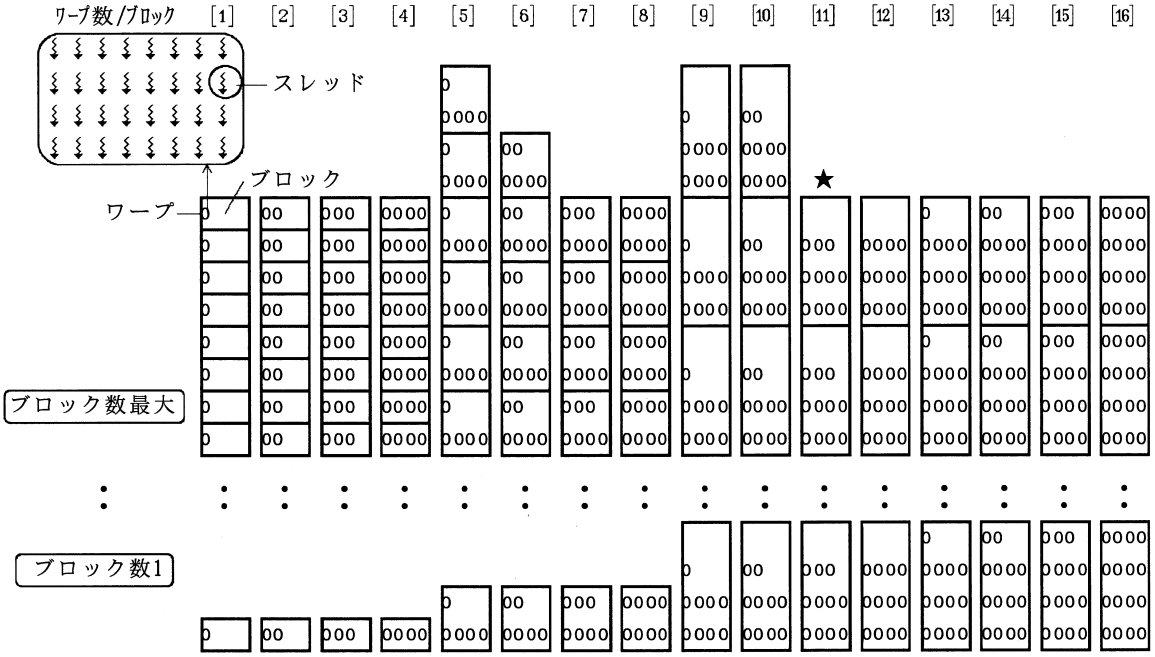


図 6-1-5

ワープ数/ブロック	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
スレッド数/ブロック	32	64	96	128	160	192	224	256	288	320	352	384	416	448	480	512
8	⑧	⑯	㉔	㉜												
1つの 7	⑦	⑭	㉑	㉘												
ストリーミング 6	⑥	⑫	⑱	㉔	㉚											
マルチプロセッサー 5	⑤	⑩	⑮	⑳	㉕	㉚										
あたりの 4	④	⑧	⑫	⑯	⑳	㉔	㉘	㉜								
ブロック数 3	③	⑥	⑨	⑫	⑮	⑱	㉑	㉔	㉗	㉚						
2	②	④	⑥	⑧	⑩	⑫	⑭	⑯	⑱	⑳	★㉒	㉔	㉖	㉘	㉚	㉜
1	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫	⑬	⑭	⑮	⑯

図 6-1-6　例えば㉒は、1 つのストリーミングマルチプロセッサーあたりに存在するワープ数が 22 個であることを示します。

処理する要素数が少ない場合

処理する要素数が少ない場合、図 6-1-6 の範囲は狭くなります。例えば要素数が 2000 個で、1 要素を 1 スレッドが担当する場合、全ワープ数は $2000 \div 32 = 62.5$ (切り上げて 63 個) です。

63 個のワープを 30 個のストリーミング・マルチプロセッサーで処理するので、1 つのストリーミング・マルチプロセッサーで処理するワープ数は、平均 $63 \div 30 = 2.1$ 個となります。

例えば図 6-1-6 のケース [2] (1 ブロックのワープ数が 2 個) の場合、ワープ数が 2.1 に近い、⑥，④，②個の場合の状態を図示すると、図 6-1-7 (1) (2) (3) のようになります。図中の[0], [1], ... は、ストリーミング・マルチプロセッサーを示します。なお、ケース [2] では、1 つのブロックに 2 ワープ入るので、全ワープ数は 63 個でなく 64 個になっています。

(1)　ケース [2] で 1 つのストリーミングマルチプロセッサーあたりのワープ数が⑥個の場合

図から分かるように、各ストリーミング・マルチプロセッサーが処理するブロック数が不均等なので、実際にはこの状態にはならず、図 6-1-7 (2) になります。

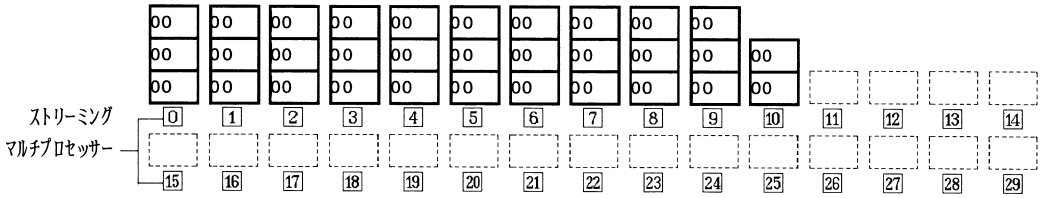


図 6-1-7 (1)

(2)　ケース [2] で 1 つのストリーミングマルチプロセッサーあたりのワープ数が④個の場合

通常はこの状態になります。なお、★に示すブロックは、実際には3に入るようです (4-5 節参照)。

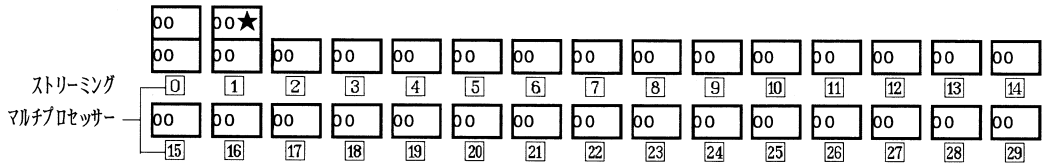


図 6-1-7 (2)

(3)　ケース [2] で 1 つのストリーミングマルチプロセッサーあたりのワープ数が②個の場合

図 6-1-7 (2) のケースで、後述するように、資源不足のため、1 つのストリーミング・マルチプロセッサーに 2 個以上のブロックが入れない場合、図 6-1-7 (3) に示すように 1 個だけ入り、残りは待ち行列に入ります。

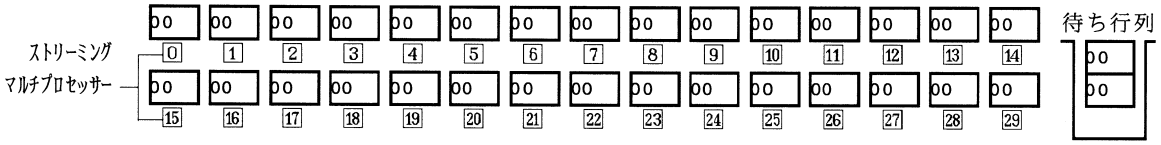


図 6-1-7 (3)

本例では、1 つのストリーミング・マルチプロセッサーで処理するワープ数は、平均 $63 \div 30 = \underline{2.1}$ 個です。従って、図 6-1-6 のケース [2] の場合、②，④，…，⑯のうち、2.1 個以上で最も小さい個数 (図 6-1-7 (2) に示す④個) か、それ以下の個数 (図 6-1-7 (3) に示す②個) が、可能なワープ数の範囲となります。

図 6-1-8 (図 6-1-6 と同じ) の範囲を求める手順を、要素数が 13000 個の場合で再度説明します。式中の「32」はワープ内のスレッド数、「30」はストリーミング・マルチプロセッサーの数です。

［全ワープ数］$= (13000 + 32 - 1) \div 32 = \underline{407}$　（注）割り算の結果は切り捨てます。

［全ワープ数］$\div 30 = \underline{407} \div 30 = \underline{13.6}$ 個　(1 つのストリーミング・マルチプロセッサーで処理する平均ワープ数)

図 6-1-8 の [1]〜[16] の各ケースで、13.6 個以上の値のうち、一番小さい値（○で囲んだ部分）か、それ以下の値が、可能なワープ数の範囲となります（図の太線より下の部分)。なお、ケース [1] では 13.6 個以上の値が存在しないので、一番大きな⑧以下となります。

ワープ数/ブロック	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
スレッド数/ブロック	32	64	96	128	160	192	224	256	288	320	352	384	416	448	480	512
8	⑧	⑯	㉔	㉜												
ストリーミング 7	⑦	⑭	㉑	㉘												
マルチプロセッサー 6	⑥	⑫	⑱	㉔	㉚											
あたりの 5	⑤	⑩	⑮	⑳	㉕	㉚										
ブロック数 4	④	⑧	⑫	⑯	⑳	㉔	㉘	㉜								
3	③	⑥	⑨	⑫	⑮	⑱	㉑	㉔	㉗	㉚						
2	②	④	⑥	⑧	⑩	⑫	⑭	⑯	⑱	⑳	㉒	㉔	㉖	㉘	㉚	㉜
1	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	⑪	⑫	⑬	⑭	⑮	⑯

図 6-1-8　(図 6-1-6 と同じ)

レジスター／シェアードメモリに関する制限

前述の 3 つの制限には、カーネル関数の特性（使用するレジスター数とシェアードメモリの容量）は加味されていませんでした。実際には、下記の 2 つの制限が加わります (Compute Capability 1.3 の場合)。

【制限 4】1 つのストリーミング・マルチプロセッサーに搭載されているレジスター数は 16384 個 (1 個あたり 4 バイト) です。1 つのブロックに含まれる全スレッドが使用するレジスター数の合計が r（個）だとすると、1 つのストリーミング・マルチプロセッサー内に同時に存在できるブロック数は、$16384/r$（個）以下となります。

【制限 5】1 つのストリーミング・マルチプロセッサーに搭載されているシェアードメモリの容量は 16384 バイトです。1 つのブロックに含まれる全スレッドが使用するシェアードメモリの合計が s（バイト）だとすると、1 つのストリーミング・マルチプロセッサー内に同時に存在できるブロック数は、$16384/s$（個）以下となります。

【制限 4】について具体例を示します (【制限 5】も考え方は同様です)。図 6-1-9 のプログラムでは、ブロックあたりのスレッド数が 512 個なので、ブロックあたりのワープ数は $512 \div 32 = 16$ 個 です。この場合、【制限 1】～【制限 3】から、1 つのストリーミング・マルチプロセッサー上のブロックの状態は、図 6-1-5 の [16] の下段か上段のいずれかになります。これを図 6-1-10 (1) (2) に再掲します。

以下の説明を簡単にするため、レジスター 512 個をまとめて「R」1 個で表します。1 つのストリーミング・マルチプロセッサーには、レジスターが 16384 個 (1 個あたり 4 バイト) 搭載されているので、「R」に換算すると、図 6-1-10 (1) の右図に示すように、32 (= 16384 ÷ 512) 個となります。

図 6-1-9 のプログラムで、例えば 1 つのスレッドがレジスターを 16 個使用するとします。1 つのワープ内の全スレッドは 512 (= 16 × 32) 個、すなわち「R」1 個分のレジスターを使用します。したがって、図 6-1-10 (1) に示すように、1 つの「o」が 1 つの「R」を使用するので、2 ブロック (合計 32 ワープ) が、同時に 1 つのストリーミング・マルチプロセッサー上に存在することができます。これが図 6-1-6 のケース [16] の㉜の状態です。

一方、1 つのワープ内の全スレッドが「R」2 個分のレジスターを使用する場合、【制限 4】により、図 6-1-10 (2) に示すように、1 ブロック (合計 16 ワープ) しか、ストリーミング・マルチプロセッサー上に存在することができません。これが図 6-1-6 のケース [16] の⑯の状態です。

なお、1 つのワープ内の全スレッドが「R」2 個分より多い (例えば「R」4 個分) レジスターを使用する場合、図 6-1-10 (3) に示すように、8 ワープまででレジスターを使いきるため、1 ブロック内の 16 ワープが 1 つのストリーミング・マルチプロセッサー上に存在できず、(図 6-1-9 のようにブロックあたり 512 スレッドでは) プログラムの実行は不可能になります。この場合の対処方法は、後述する「■　補足」を参照して下さい。

```
   :
kernel<<<100,512>>>();
   :
```

図 6-1-9

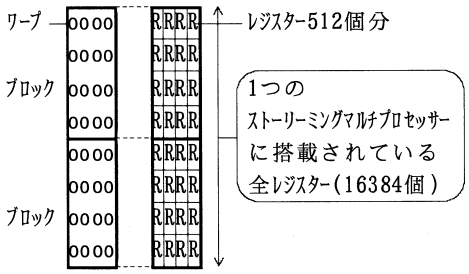


図 6-1-10 (1)

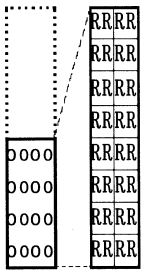


図 6-1-10 (2)

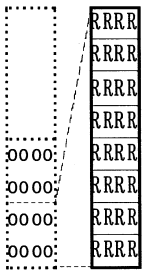


図 6-1-10 (3)

占有率計算機

CUDA では、上記で説明した内容を自動的に計算する、「Cuda Occupancy Calculator」（以後占有率計算機）というツールが提供されています。占有率計算機は、パソコン上の Excel を使用して動作します。

占有率計算機は、http://www.nvidia.com/object/cuda_get.html の「Linux」の「GPU Computing SDK code samples」の「CUDA Occupancy Calculator」をクリックし、パソコンにダウンロードします。

占有率計算機の使用方法

図 6-1-11 のプログラムを例に、占有率計算機の使用方法の概要を説明します。

占有率計算機では、カーネル関数が使用する資源の量を入力で設定するので、まずこの量を調べます。図 6-1-11 のプログラム（カーネル関数部分のみでも可）を、6-1-12 の②の下線部 (2-7 節参照) を指定してコンパイルすると、③が表示されます。③の下線部（太線）は、スレッドあたりレジスターを 2 個使用することを示します。下線部（二重線）は、ブロックあたりシェアードメモリ (smem) を 24 (= 8 + 16) バイト使用することを示します。

図 6-1-11 では、①に示すように、ブロックあたりのスレッド数が 480 個なので、ブロックあたりのワープ数は 480 ÷ 32 = 15 個です。従って、前述の【制限 1】～【制限 3】により、1 つのストリーミング・マルチプロセッサー上のブロックは、図 6-1-5 の [15] の上段、下段、実行不能のいずれかになります。これを図 6-1-13 (1)～(3) に示します。図 6-1-10 (1)～(3) で説明したように、使用する資源が少ないと図 6-1-13 (1) に、多いと図 6-1-13 (2) に、非常に多いと図 6-1-13 (3) になります。

ブロックあたりのスレッド数 (480 個)、使用する資源 (レジスター (2 個)、シェアードメモリ (24 バイト)) を設定し、占有率計算機を実行すると、図 6-1-13 (1)～(3) のどの状態になるかが表示されます。なお本例では、後述するように図 6-1-13 (1) になります。

```
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;
}
```

```
int main(void){
    :
  kernel<<<100,480>>>(dA);                    ①
    :
}
```

図 6-1-11　test.cu

```
$ nvcc -Xptxas -v -c test.cu ↵          ②
  ～ Used 2 registers , 8+16 bytes smem    ③
```

図 6-1-12

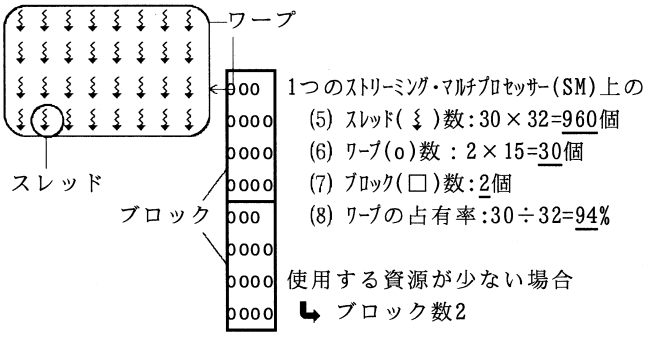


図 6-1-13 (1)

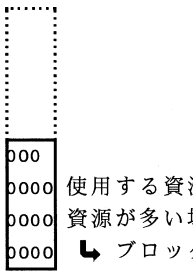


図 6-1-13 (2)

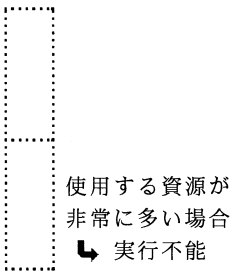


図 6-1-13 (3)

設定する項目

パソコンにダウンロードした占有率計算機のアイコンをクリックすると、占有率計算機が起動し、図 6-1-14 と図 6-1-16 (1)〜(3) の画面がパソコンに表示されます。なお、使用方法の詳細は、画面左下の「help」をクリックして下さい。

- 図 6-1-14 の (1) では、理研の RICC の場合、1.3 を選択します。
- (2) では、ブロックあたりのスレッド数を設定します。本例では、図 6-1-11 の①の 480 を設定します。
- (3) では、スレッドあたりに使用するレジスターの個数 (16384 バイト個以下) を設定します。本例では、図 6-1-12 の③で表示された 2 個を設定します。
- (4) では、ブロックあたりに使用するシェアードメモリの容量（バイト）(16384 バイト以下) を設定します。本例では、図 6-1-12 の③で表示された 24 (= 8 + 16)（バイト）を設定します。

表示される項目

設定後、カーソルを適当な場所に置いてマウスをクリックすると、図 6-1-14 の (5)〜(11) の各値と、図 6-1-16 (1)〜(3) のグラフが表示されます。これらは、図 6-1-14 の (1)〜(4) の条件の場合、図 6-1-13 (1)〜(3) のうち、図 6-1-13 (1) の状態になることを示しています。なお、以下の説明では、ストリーミング・マルチプロセッサーを SM と略します。

- (5) では、1 つの SM 上に同時に存在することができるスレッド数が表示されます。本例では、図 6-1-13 (1) に示すように、960 個が表示されます。
- (6) では、1 つの SM 上に同時に存在することができるワープ数が表示されます。本例では、図 6-1-13 (1) に示すように、30 個が表示されます。
- (7) では、1 つの SM 上に同時に存在することができるブロック数が表示されます。図 6-1-13 (1) に示すように、2 個が表示されます。
- (8) では、ワープの占有率が表示されます。前述の【制限 3】で説明したように、1 つの SM 上に同時に存在できる最大のワープ数は 32 個です。32 個に対する (6) のワープ数 (単位は %) を占有率 (Occupancy) と呼びます。本例では、図 6-1-13 (1) に示すように、94% が表示されます。
- (9)〜(11) には参考データが表示されます。(9) では、ブロックあたりのワープ数が表示されます。本例では、(2) で指定した 480 から、480 ÷ 32 = 15 個が表示されます (図 6-1-13 (1) 参照)。
- (10) では、ブロックあたりに割り当てられるレジスター数が表示されます。本来ならば (10) = (2) × (3) で 960 個となるはずですが、1024 個が表示されます。これは、実際に使用するレジスター数が 960 個で、割り当てられるレジスター数が 1024 個という意味です。占有率計算機では、1 つの SM 上に同時に存在することができるワープ数の計算には、1024 個の方を用います (後述する (11) の場合も同様です)。
  (2) から (10) を求める方法を以下に示します (注 1)。1 つ目の式は、(2)（ブロックあたりのスレッド数）が例えば 1〜64 個のときは 64 個、65〜128 個のときは 128 個を、変数 `temp` に代入するという意味です。
  2 つ目の式は、1 つ目の式で求めた `temp`（ブロックあたりのスレッド数）に (3)（スレッドあたりのレジスター数）を掛けてブロックあたりのレジスター数を求め、この値が例えば 1〜512 個のときは 512 個、513〜1024 個のときは 1024 個を、(10)（ブロックあたりに割り当てられるレジスター数）にするという意味です。

$$\texttt{temp} = \{((2) + 63)/64\} * 64 \leftarrow \text{割り算の結果は切り捨てます。}$$
$$(10) = \{(\texttt{temp} * (3) + 511)/512\} * 512 \leftarrow \text{割り算の結果は切り捨てます。}$$

- (11) では、ブロックあたりに割り当てられるシェアードメモリの容量（バイト）が表示されます。本来ならば (11) = (4) で 24（バイト）となるはずですが、512（バイト）が表示されます。これは、実際に使用するシェアードメモリの容量が 24（バイト）で、割り当てられるシェアードメモリの容量が 512（バイト）であるという意味です。
  計算方法を以下に示します (注 1)。この式は、(4)（ブロックあたりのシェアードメモリの容量）が例えば 1〜512（バイト）のときは 512（バイト）、513〜1024（バイト）のときは 1024（バイト）を、(11)（ブロックあたりに割り当てられるシェアードメモリの容量）にするという意味です。

$$(11) = \{((4) + 511)/512\} * 512 \leftarrow \text{割り算の結果は切り捨てます。}$$

(注 1) 計算方法は Compute Capabilitiy によって異なります (上記は 1.3 の場合です)。計算方法の詳細は「CUDA C Programming Guide」（付録参照）の 4.2 節を参照して下さい。

表示されるグラフ

- 図 6-1-16 (1) のグラフは、図 6-1-14 の (3)，(4) の設定はそのままにした場合の、(2) と (6) の関係を示します。▲は、現在 (2) で設定し、(6) で表示されている値（〓の部分）を示します。
  グラフから分かるように、(2)（ブロックあたりのスレッド数）が 480 個だと (6) は 30 個ですが、(2) を例えば 256 個にすると、△に示すように、(6) が 32 個に増えることが分かります。このグラフを使用して、(6) がなるべく大きくなる (2) の値を探し、図 6-1-11 の①で設定します。
  なお、本例では使用する資源が少ないので、図 6-1-16 の [1]〜[16] の各ケースはいずれも、一番上の○のワープ数 (⑧，⑯，…，㉜) になります。この値が、図 6-1-16 (1) のグラフと一致します。
- 図 6-1-16 (2) のグラフは、図 6-1-14 の (2)，(4) の設定はそのままにした場合の、(3) と (6) の関係を示します。このグラフから、(3)（スレッドあたりのレジスター数）が 16 個以下だと、(6) は 30 個 (図 6-1-13 (1) の状態) ですが、17 個〜32 個だと (6) は 15 個 (図 6-1-13 (2) の状態) になり、33 個以上だと実行不能 (図 6-1-13 (3) の状態) になることが分かります。
  このグラフは、例えばあるプログラムで (3) が 20 個だとすると、△に示すように (6) は 15 個になりますが、(3) が 16 個ならば (6) は 30 個に増えるので、カーネル関数で使用するレジスターの数をあと 4 個減らして 20 個 → 16 個とし（方法は「■　補足」を参照)、(6) を増やすことができないかを検討する、という場合に使用します。
- 図 6-1-16 (3) のグラフは、図 6-1-14 の (2)，(3) の設定はそのままにした場合の、(4) と (6) の関係を示します。図 6-1-16 (2) と考え方は同じなので、説明は省略します。

補足

- 占有率計算機では、前述の「処理する要素数が少ない場合、図 6-1-6 の範囲が狭くなる」は考慮されていないので注意して下さい（処理する要素数を指定する欄がないので)。
- 使用するレジスター数が多すぎて実行が不可能な場合や、レジスター数を減らして SM 上に同時に存在できるワープ数を増やしたい場合、使用するレジスター数を減らす方法として、カーネル関数内で使用している一時変数をできるだけ使い回すようにプログラムを修正する方法と、コンパイルオプションでスレッドあたり使用できるレジスター数の上限を設定する方法 (2-7 節参照) があります。ただし後者は、高速なレジスターの代わりに低速なローカルメモリが使用されるので、却って遅くなる可能性があります。
- 「CUDA C Best Practices Guide」（付録参照）の 4.4 節によると、カーネル関数内で `__syncthreads()` を使用して同期を取っている場合、1 つのストリーミング・マルチプロセッサーあたり、複数のブロック数が望ましいと記載されています。これについて説明します。
  占有率計算機で調べた結果、1 つのストリーミング・マルチプロセッサーあたり、同時に存在できるワープ数が最も多いのは、図 6-1-15 (1) と図 6-1-15 (2) の場合だとします。
  図 6-1-15 (3) のように、カーネル関数内の②で、`__syncthreads()` を使用してブロック内の全スレッドの同期を取っている場合、図 6-1-15 (2) では、先に①を終了して②に到達したワープは、ブロック 0 内の 16 個の全ワープが②に到達するまで③に進むことができません。
  一方図 6-1-15 (1) では、先に②に到達したワープ (ブロック 0 に所属するとします) は、ブロック 0 の 8 個の全ワープが②に到達すれば、ブロック 1 のワープが②に到達していなくても③に進むことができます。
  従って、同じワープ数なら、図 6-1-15 (1) のようにブロック数が多い方が、CUDA コアの稼働率が高くなるため速度が速くなる可能性があります。

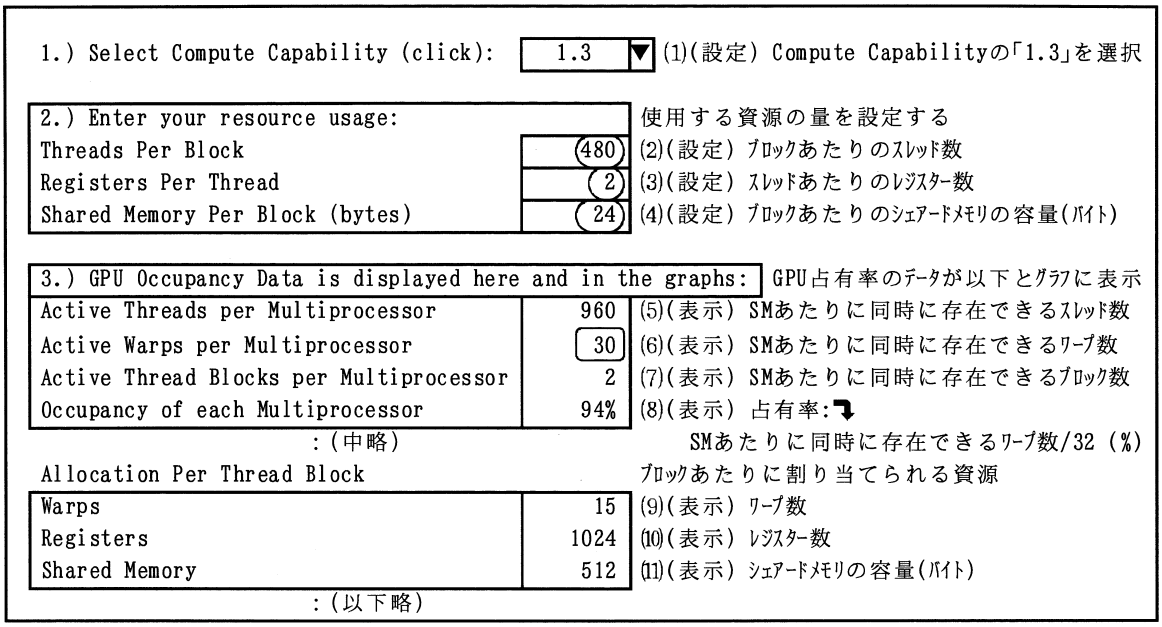


図 6-1-14　ストリーミング・マルチプロセッサーを SM と省略しています。

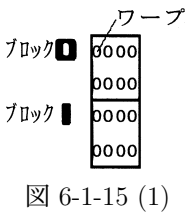


図 6-1-15 (1)

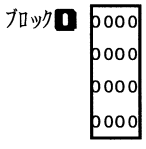


図 6-1-15 (2)

```
__global__ void kernel(){
    処理                    ①
  __syncthreads();          ②
    処理                    ③
}
    :
  kernel<<<60, 8*32>>>()      ← 図6-1-15(1)の場合
  kernel<<<30,16*32>>>()      ← 図6-1-15(2)の場合
    :
```

図 6-1-15 (3)

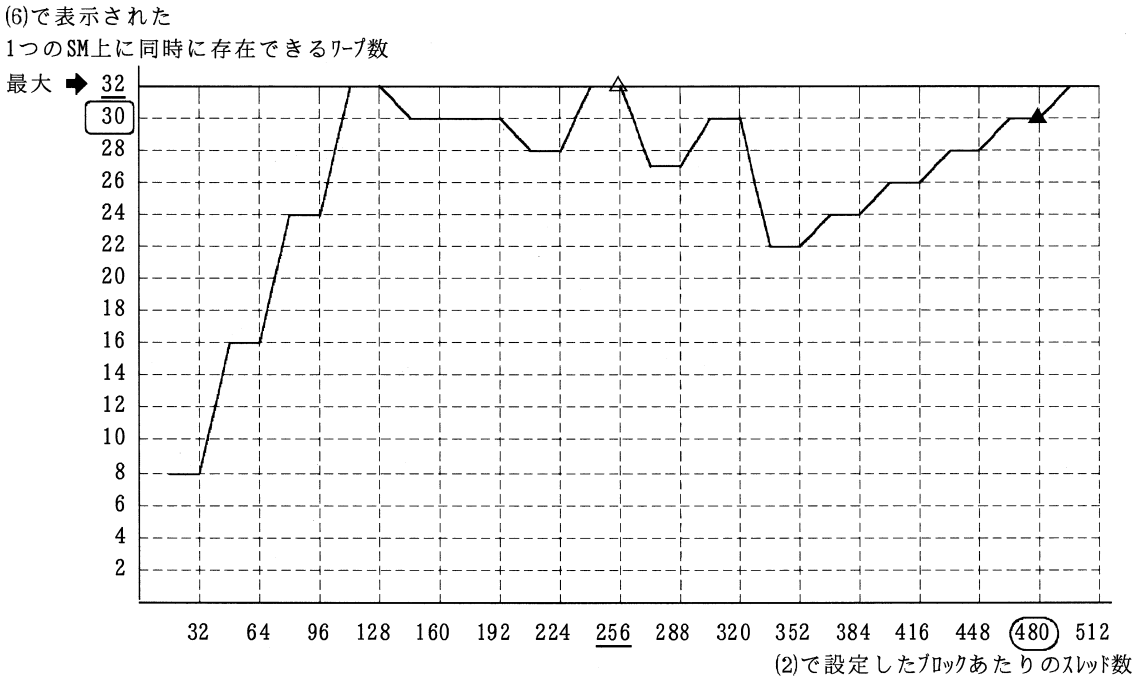


図 6-1-16 (1)

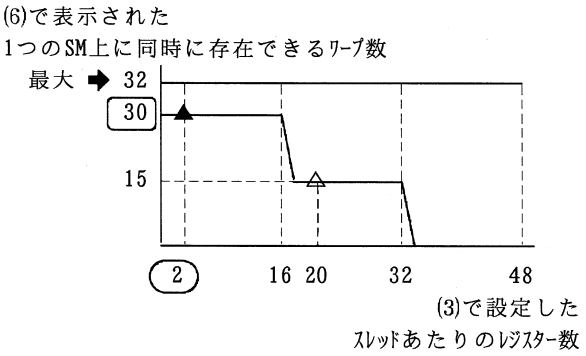


図 6-1-16 (2)

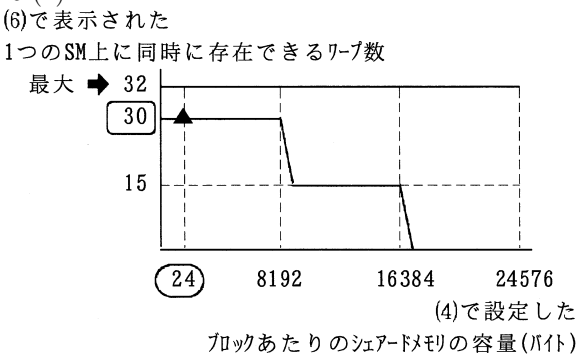


図 6-1-16 (3)

6-2　ホストとデバイス間のコピーの高速化

ホストとデバイスの間のコピー (cudaMemcpy など) の速度を速くする 3 つの方法を説明します。いずれの方法も、ホスト側の配列を、Page-Locked (または pinned) ホストメモリ（注）として指定します。この指定によって、なぜコピーの速度が速くなるかについては、「CUDA Reference Manual」（付録参照）にもあまり記載されていないので、以下では説明せず、具体的な指定方法のみを説明します。

なお、本節の方法を、大きな配列に適用したり、多くの配列に適用すると、システム全体の効率が低下して、ジョブ全体の経過時間が却って遅くなってしまいます。修正後に経過時間を測定し、速くなった場合のみ使用して下さい。

（注）メモリ上にある普通の配列は、メモリが足りなくなると、ページングによって、ディスク上のページングファイルに追い出される可能性がありますが、Page-Locked ホストメモリ上の配列は、メモリが足りなくなっても追い出されません。

方法 1

図 6-2-1 (2) に、修正前のプログラムを示します (図 6-2-1 (1) 参照)。②，④でホスト側の配列 A を malloc を使用して確保し、⑥でデバイス側の配列 dA にコピーし、⑦，①で配列 dA を処理します。処理が終了したら⑧で配列 dA をホスト側の配列 A にコピーし、⑨で配列 A を解放します。

方法 1 では、図 6-2-2 に示すように、④を修正して (4) に、⑨を修正して (9) にします。なお、(4) は CUDA 関数です。CUDA 関数で（デバイス側でなく）ホスト側の配列 A を確保していることに注意して下さい。

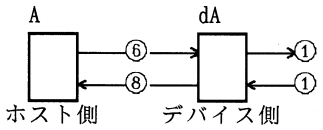


図 6-2-1 (1)

方法 2

図 6-2-3 に示すように、④を修正して [4] に、⑨を修正して [9] にします。この方法の場合、[11] のようにホストプログラム内で配列 A を更新する場合は問題ないですが、[12] のように配列 A を参照する場合、速度がかなり低下します。従ってこの方法は、ホスト側で参照を行わない配列に対して適用して下さい。

```
#define N (10000)
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;                          ①
}
  int main(void){
  float *A;                                      ②
  float *dA;                                     ③
  size_t size = N*sizeof(float);
  A = (float*)malloc(size);                      ④
  cudaMalloc((void**)&dA,size);                  ⑤
    :
  cudaMemcpy(dA,A,size,                          ⑥
                  cudaMemcpyHostToDevice);       ⑥
  kernel<<<1,1>>>(dA);                           ⑦
  cudaMemcpy(A,dA,size,                          ⑧
                  cudaMemcpyDeviceToHost);       ⑧
    :
  free(A);                                       ⑨
  cudaFree(dA);                                  ⑩
    :
```

図 6-2-1 (2)

```
    :
  cudaHostAlloc((void**)&A,size,                 (4)
                cudaHostAllocDefault);           (4)
    :
  cudaFreeHost(A);                               (9)
    :
```

図 6-2-2　方法 1

```
    :
  cudaHostAlloc((void**)&A,size,                 [4]
                cudaHostAllocWriteCombined);     [4]
  for(i=0;i<N;i++){
    A[i] = ～ ;  ○                               [11]
    ～ = A[i];  ×                                [12]
  }
    :
  cudaFreeHost(A);                               [9]
    :
```

図 6-2-3　方法 2

方法 3

通常のプログラムでは、図 6-2-4 (1) (図 6-2-1 (1) と同じ) に示すように、ホスト側の配列 A とデバイス側の配列 dA がそれぞれ存在し、⑥と⑧で明示的にデータを転送します。

方法 3 では、図 6-2-4 (2) に示すように、ホスト側の配列 A のみしか存在せず、デバイス側プログラムから配列 A を直接アクセスします (ただし、ホスト側の名前である配列 A でアクセスすることはできず、異なる名前 (本例では配列 dA) でアクセスします)。従って、図 6-2-4 (1) の⑥と⑧のような転送を、明示的に行う必要はありません。

方法 3 のプログラムを図 6-2-5 に示します。2と3でホスト側の配列 A（のポインター）とデバイス側の配列 dA（のポインター）を宣言します。配列 A と dA は物理的には同一です。4, 5を指定すると、配列 A がホスト側のメモリ上に確保されます。

カーネル関数内の1では、A という名で配列 A を参照／更新することはできません。6で A と dA を対応付け、7の引数に dA を指定し、1では配列 dA という名で配列 A を参照／更新します。なお、6の最後の引数は、CUDA の現在のバージョンでは「0」を指定します。

以下に注意点を述べます。

- 3-2 節で説明したように、7でカーネル関数をコールすると、すぐにホストプログラムに制御が戻ります。従って、7の後の8では、ホストプログラムとカーネル関数が同時に動作するため、もしホストプログラムが8で配列 A を更新すると、1の途中でカーネル関数の配列 dA (配列 A と同一) が更新されてしまい、計算結果がおかしくなる可能性があります。従って8で配列 A を更新してはいけません。
- 図 6-2-1 (2) では、⑧のコピーの後であれば、①でカーネル関数が計算した結果がホストプログラムの配列 A に入っています。一方図 6-2-5 では、図 6-2-1 (1) の⑧に相当するコピーがないため、ホストプログラムが8で配列 A を参照しても、1でカーネル関数が計算した結果が配列 A に入っているとは限りません。従って8で配列 A を参照してはいけません。
- 以上より、9で同期を取り、カーネル関数が終了した後、10で配列 A を更新あるいは参照して下さい。
- 方法 3 では、配列 dA は実際にはホスト側の配列なので、カーネル関数内で配列 dA をアトミック関数 (4-1 節参照) で処理することはできません。

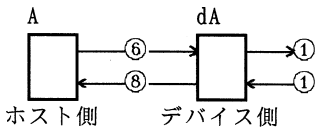


図 6-2-4 (1)

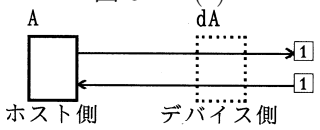


図 6-2-4 (2)

```
#define N (10000)
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;                                   1
}
  int main(void){
  float *A;                                               2
  float *dA;                                              3
  size_t size = N*sizeof(float);
  cudaSetDeviceFlags(cudaDeviceMapHost);                  4
  cudaHostAlloc((void**)&A,size,cudaHostAllocMapped);     5
  cudaHostGetDevicePointer((void**)&dA,(void*)A,0);       6
    :
  kernel<<<1,1>>>(dA);                                    7
  A[0] = ～; ✖    ～ = A[0]; ✖                           8
  cudaThreadSynchronize();                                9
  A[0] = ～; ○    ～ = A[0]; ○                           10
  cudaFreeHost(A);                                        11
    :
```

図 6-2-5　方法 3

ホストとデバイス間のコピー回数を減らす方法

5-5 節で説明したように、コピーするデータ量が同じならば、コピーする回数が少ない方が、オーバーヘッドが少ないため速度は速くなります。

図 6-2-6 (1) では、図 6-2-7 (1) に示すように、⑤と⑥で配列 A，B を配列 dA，dB にコピーしており、コピー回数は 2 回です。このコピー回数を 1 回に減らしたプログラムを図 6-2-6 (2) に示します。

- ③の配列 A，B，dA，dB を、⑩のように合体して配列 AB，dAB とします。配列 AB の前半を元の配列 A、後半を元の配列 B として使用します。
- これに伴い、元の④の処理を⑪のように修正する必要があり、プログラムはやや分かりにくくなります。
- ⑫で、図 6-2-7 (2) に示すように、ホスト側の配列 AB をデバイス側の配列 dAB にコピーします。⑫は、⑤，⑥とコピーするデータ量は同じですが、コピー回数が 1 回で済むため、速度が速くなります。
- ⑬で、実引数の 1 つ目に配列 dAB の前半の先頭アドレスを、2 つ目に後半の先頭アドレスを指定し、カーネル関数をコールします。
- ⑧の仮引数の 1 つ目に配列 dA を、2 つ目に配列 dB を指定します (①と同じです)。これによって、図 6-2-7 (2) に示すように、カーネル関数内では、配列 dAB の前半を配列 dA、後半が配列 dB として取り扱うことができます。従って⑨は②と同一で、修正する必要はありません。

```
                                                    ①
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;                             ②
  dB[i] = dB[i] + 2.0f;                             ②
}
int main(void){
  float A[16],B[16];                                ③
  float *dA  ,*dB;                                  ③
    :
  for(i=0;i<16;i++){
    A[i] = ～;                                      ④
    B[i] = ～;                                      ④
  }
  cudaMemcpy(dA,A,16*4,～HostToDevice)              ⑤
  cudaMemcpy(dB,B,16*4,～HostToDevice)              ⑥
  kernel<<<1,16>>>(dA,dB);                          ⑦
    :
```

図 6-2-6 (1)

```
                                                    ⑧
__global__ void kernel(float *dA,float *dB){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;                             ⑨
  dB[i] = dB[i] + 2.0f;                             ⑨
}
int main(void){
  float AB[32];                                     ⑩
  float *dAB;                                       ⑩
    :
  for(i=0;i<16;i++){
    AB[i]    = ～;                                  ⑪
    AB[i+16] = ～;                                  ⑪
  }
  cudaMemcpy(dAB,AB,32*4,～HostToDevice);           ⑫
  kernel<<<1,16>>>(&dAB[0],&dAB[16]);               ⑬
    :
```

図 6-2-6 (2)

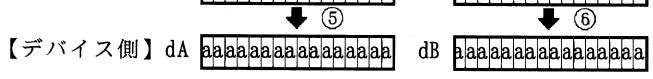


図 6-2-7 (1)

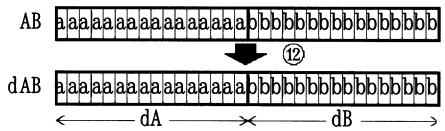


図 6-2-7 (2)

6-3　ストリーム

本節では、ストリームという機能を使用して、ホスト側とデバイス側の処理をオーバーラップ（同時に実行すること）させたり、あるいはデバイス側のコピーとカーネル関数の処理をオーバーラップさせることによって、実行時間を短縮する方法を説明します。

なお、本節では、Compute Capability 1.3 (RICC のマシン環境) の場合の動作を説明します。他の Compute Capability では、動作が異なる可能性がありますので、注意して下さい。

6-3-1　非同期関数

非同期関数の種類

3-2 節でも説明しましたが、図 6-3-1 の①のように、関数をコールしてから関数の処理が終了するまで、ホスト側のプログラムが待機する関数を同期関数、②のように、待機しないでただちに次の処理を実行する関数を非同期関数と言います。

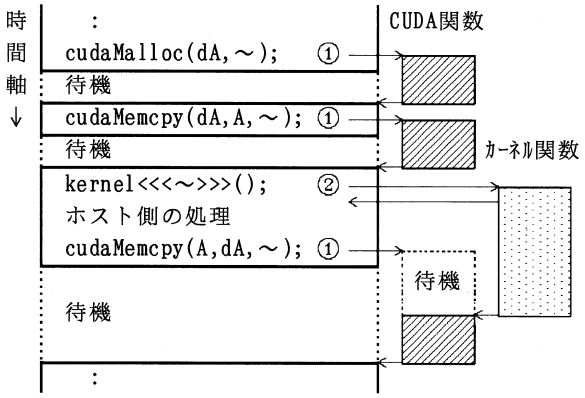


図 6-3-1

CUDA で提供されている非同期関数を以下に示します (「CUDA C Programming Guide」の stream の節を参照)。

(1) カーネル関数。

(2) コピーを行う CUDA 関数のうち、名前の最後に Async がついている以下の関数 (以後 cudaMemcpy～Async 型の CUDA 関数と呼びます) を使用して、ホスト〓デバイスのコピーを行った場合。なお、cudaMemcpy～Async 型の CUDA 関数は、次ページで説明するように、ホスト側の配列を、Page-Locked (または pinned) ホストメモリとして確保する必要があります。

1 次元用	2 次元用	3 次元用
cudaMemcpyAsync	cudaMemcpy2DAsync	cudaMemcpy3DAsync
cudaMemcpyFromArrayAsync	cudaMemcpy2DFromArrayAsync	
cudaMemcpyToArrayAsync	cudaMemcpy2DToArrayAsync	
cudaMemcpyFromSymbolAsync		
cudaMemcpyToSymbolAsync		

(3) コピーを行う CUDA 関数 (cudaMemcpy, cudaMemcpyAsync など) で、デバイス〓デバイスのコピーを行った場合。(例) cudaMemcpy(dB,dA,size,cudaMemcpyDeviceToDevice)

(4) (2) 以外の、名前の最後に Async がついている CUDA 関数 (cudaMemsetAsync, cudaMemcpyPeerAsync など)。

(5) メモリにデータを設定する CUDA 関数 (cudaMemset (4-2 節参照) など)。

cudaMemcpy〜Async 型の CUDA 関数の使い方

ストリームの説明に入る前に、前ページ (2) の cudaMemcpy〜Async 型の CUDA 関数の使用方法を説明します。図 6-3-2 (1) の②で使用している通常の cudaMemcpy（同期関数）を、cudaMemcpy〜Async 型の CUDA 関数 cudaMemcpyAsync（非同期関数）に置き換えたプログラムを図 6-3-2 (2) に示します。変更箇所は次の通りです。

- 2で cudaMemcpyAsync を使用します。最後の引数に（本例では）「0」を指定します（この引数の意味は後述します）。
- cudaMemcpy〜Async 型の CUDA 関数の場合、2で使用するホスト側の配列 A は、①のように通常の方法で確保せず、1のように Page-Locked (または pinned) ホストメモリ (6-2 節参照) として確保する必要があります。それに伴い、ホスト側の配列 A を解放する⑥を6に変更します。
- 本例では、2でデバイス側からホスト側にコピーした配列 A を、⑤で出力します。2が非同期関数なので、5の時点で2のコピーが完了していることを保証するため、4で同期を取ります (3-2 節参照)。

2は非同期関数なので、図 6-3-2 (2) の着色した部分でホストプログラム側の3の処理とコピーがオーバーラップし、コピーの時間が隠蔽されるため、（一般に）実行時間が短縮します（詳細は次節で説明します）。

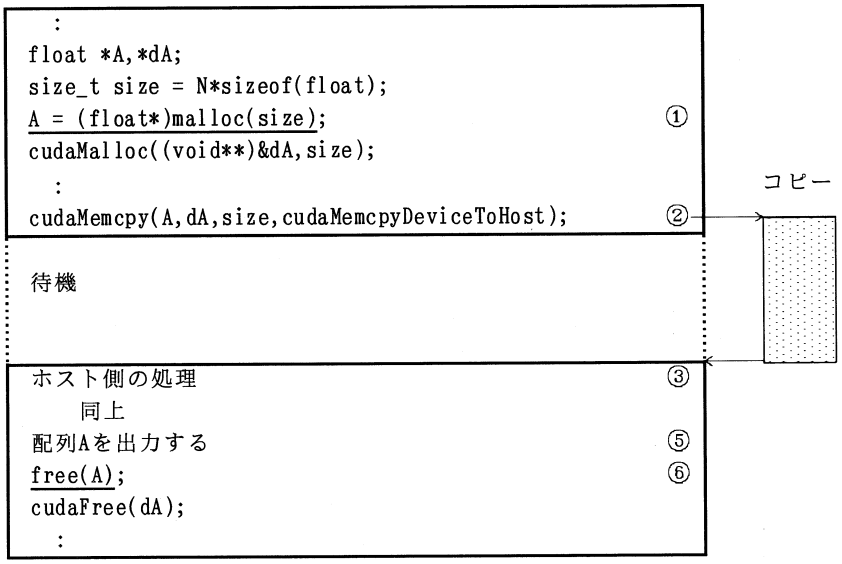


図 6-3-2 (1)

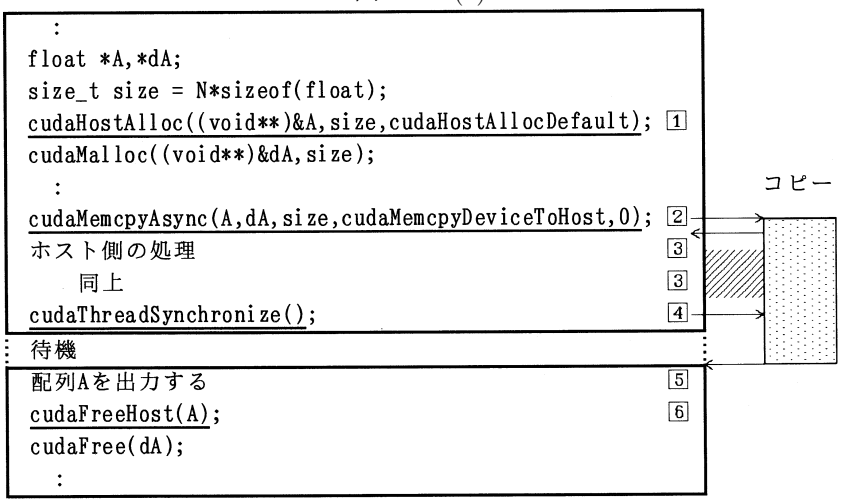


図 6-3-2 (2)

6-3-2　ホスト側とデバイス側の処理のオーバーラップ

本節では、前節で簡単に取り上げた、ホスト側とデバイス側の処理をオーバーラップ（同時に実行）させてデバイス側の処理時間を隠蔽し、実行時間を短縮する方法を説明します。

ストリーム ID の指定方法

まずストリーム ID について説明します。カーネル関数と、cudaMemcpy〜Async 型の CUDA 関数には、ストリーム ID という識別子を（暗黙または明示的に）指定します。ストリーム ID にはゼロ (以下0と表します) と、0以外の 2 種類があります（それぞれの意味は後述します)。

(1)　ストリーム ID 0の設定方法

カーネル関数や cudaMemcpy〜Async 型の CUDA 関数に、ストリーム ID 0を指定する方法を説明します。

- 図 6-3-3 の (1) に示すように、今までは、カーネル関数の実行構成には 2 つの引数を指定しましたが、実際には (2) に示すように、全部で 4 つの引数があります。3 番目の引数には、動的に確保したいシェアードメモリの量 (3-6 節参照) を指定し、4 番目の引数にストリーム ID 0を指定します。(1) のように、3，4 番目の引数を指定しなかった場合、(2) のようにデフォルト値の0が設定されます。
- cudaMemcpy〜Async 型の CUDA 関数の場合、(3) に示すように、最後の引数にストリーム ID 0を指定します。
- (4) の cudaMemcpy のような同期関数には、ストリーム ID を指定する引数がありません。この場合、デフォルト値の0が設定されていると考えて下さい。

```
kernel<<<30,32>>>();                                          (1)
kernel<<<30,32,0,0)>>>();                                     (2)
cudaMemcpyAsync(A,dA,size,cudaMemcpyDeviceToHost,0);          (3)
     cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);            (4)
```

図 6-3-3

(2)　0以外のストリーム ID の設定方法

カーネル関数や cudaMemcpy〜Async 型の CUDA 関数に、0以外のストリーム ID を指定する方法を説明します。

- 図 6-3-4 の [1] で、ストリーム ID を保管する変数または配列（名前は任意）を宣言し、[2] を実行すると、変数 (本例では stream) にストリーム ID が戻ります（戻る具体的な値について関知する必要はありません)。
- 作成したストリーム ID をカーネル関数に指定する場合、[3] のようにします。cudaMemcpy〜Async 型の CUDA 関数に指定する場合、[4] のようにします。
- 作成したストリーム ID を解放する場合、[5] を実行します。

```
cudaStream_t stream;                                          [1]
cudaStreamCreate(&stream);                                    [2]
kernel<<<30,32,0,stream)>>>();                                [3]
cudaMemcpyAsync(A,dA,size,cudaMemcpyDeviceToHost,stream);     [4]
cudaStreamDestroy(stream);                                    [5]
```

図 6-3-4

ストリームの対象となる関数

これから説明するストリームという機能の対象となるのは、図 6-3-5 (1) に示すように、非同期関数 (カーネル関数と、cudaMemcpy〜Async 型の CUDA 関数でホスト〓デバイスのコピーを行った場合) です。以下の説明で、図 6-3-5 (1) の各非同期関数を、図 6-3-5 (2) のように略記します (→，← はコピーの方向を表します)。また、0以外のストリーム ID を、①，②のように表します。

```
cudaMemcpyAsync(〜,cudaMemcpyHostToDevice,0);
kernel<<<〜,0>>>();
cudaMemcpyAsync(〜,cudaMemcpyDeviceToHost,0);
kernel<<<〜,stream1>>>();
kernel<<<〜,stream2>>>();
```

図 6-3-5 (1)

＝

```
(→)ｺﾋﾟｰ0
ｶｰﾈﾙ関数0
(←)ｺﾋﾟｰ0
ｶｰﾈﾙ関数①
ｶｰﾈﾙ関数②
```

図 6-3-5 (2)

ストリームの例

図 6-3-6 (1) を実行したときのデータの動きを図 6-3-6 (2) に示します (配列 A と dA の大きさは 2 とします)。図から分かるように、この 3 つの非同期関数は、(1)，(2)，(3) の順に実行しなければなりません。このように、コールした順番 1 つずつに実行しなければならない非同期関数の一続きを、ストリームと言います。つまり、同一ストリーム内の各非同期関数間には、実行順序に依存関係があります。そして、同一ストリームに属する各非同期関数には、同じストリーム ID (本例では0) を（暗黙または明示的に）指定する必要があります。

```
    :
(1)  (→)ｺﾋﾟｰ0 (配列AをdAにｺﾋﾟｰ)
(2)  ｶｰﾈﾙ関数0 (配列dAに1を加算)
(3)  (←)ｺﾋﾟｰ0 (配列dAをAにｺﾋﾟｰ)
    :
```

図 6-3-6 (1)

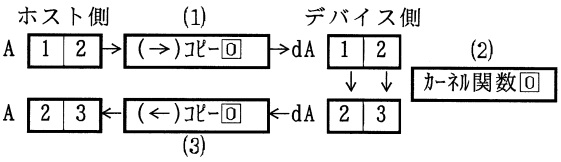


図 6-3-6 (2)

ホスト側とデバイス側の処理のオーバーラップ

図 6-3-7 (1) で、(1) と (3) は同期関数、(2) は非同期関数です。(1) と (3) も非同期関数に変更したプログラムを図 6-3-7 (2) に示します。図 6-3-6 (1) (2) で説明したように、[1]，[2]，[3] はこの順序で実行する必要があるので、同一ストリーム (本例ではストリーム ID 0) にする必要があります。また、6-3-1 節で説明したように、[1] と [3] で使用するホスト側の配列 A は、Page-Locked (または pinned) ホストメモリとして確保する必要があります。

図 6-3-7 (1) (2) で、ホスト側の処理とデバイス側の処理をオーバーラップできる可能性があるのは ↕ の部分です。図 6-3-7 (2) では、[1]，[3] を非同期関数にしたため、オーバーラップできる部分が長くなっています。

オーバーラップ時に行うホスト側の処理としては、その時点でホストプログラム側に存在する計算結果をファイルに書き出すなどの処理が考えられます。

```
   :
cudaMemcpy(dA,A,〜); (1)
kernel<<<〜>>>(dA);  (2)
ホスト側の処理
    同上
    同上
    同上
cudaMemcpy(A,dA,〜); (3)
   :
```

図 6-3-7 (1)

```
   :
cudaMemcpyAsync(dA,A,〜,0); [1]
kernel<<<〜>>>(dA);         [2]
cudaMemcpyAsync(A,dA,〜,0); [3]
ホスト側の処理
    同上
    同上
    同上
cudaThreadSynchronize();
   :
```

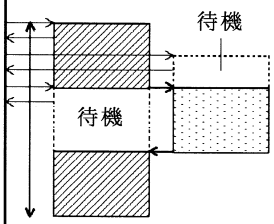


図 6-3-7 (2)

非同期関数の終了チェック

図 6-3-8 (1) では、(1) で非同期関数 (カーネル関数またはホスト〓デバイスのコピーを行う cudaMemcpy〜Async 型の CUDA 関数) をコールしており、↕の部分でホスト側の処理とオーバーラップできる可能性があります。本例のようにデバイス側の処理が先に終了した場合、[3]，[4] ではホスト側の処理のみが行われます。

図 6-3-8 (2) のように、デバイス側の処理が終了したら、ホスト側の処理を [2] で中断し ([3] で再開)、(2) で別の処理（例えばカーネル関数をコール）を実行させる方法を説明します。

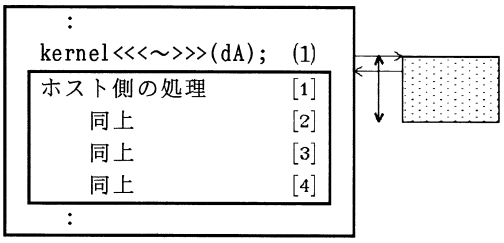


図 6-3-8 (1)

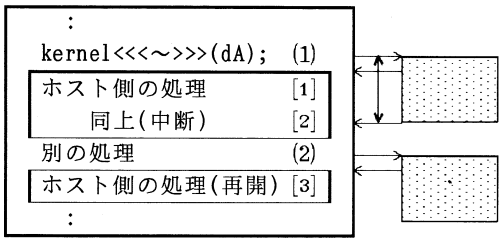


図 6-3-8 (2)

プログラム例を図 6-3-9 (1) に、タイムチャートを図 6-3-9 (2) に示します。

- 図 6-3-9 (1) の (1) で非同期関数をコールすると、すぐにホスト側のプログラムに制御が戻ります。なお、本例では (1) のストリーム ID は stream です。
- (2) の下線部の関数は、この時点までにコールされている、カッコ内に指定したストリーム ID (本例では stream) のタスクが全て終了しているかどうかをチェックします。全て終了している場合は「cudaSuccess」が戻り、終了していない場合は「cudaErrorNotReady」が戻ります。
- 図 6-3-9 (2) の [1] の処理が終了していない場合、(2) で「cudaErrorNotReady」が戻るので、(3) で「ホスト側の処理」を実行します。図 6-3-9 (2) の着色部分に示すように、[1] と「ホスト側の処理」はオーバーラップして同時に動きます。
- 以後、(2)，(3) が反復し、[1] の処理が終了しているかどうかのチェックと、「ホスト側の処理」を交互に実行します。
- [1] の処理が終了した後、再び (2) の下線部の関数を実行すると、「cudaSuccess」が戻るため、「ホスト側の処理」を中断して (2) のループから抜け、(4) で「別の処理」を行います。

```
   :
cudaStream_t stream;
cudaStreamCreate(&stream);
   :
kernel<<<〜,stream>>>();                    (1)
while(cudaStreamQuery(stream)!=cudaSuccess) (2)
  ホスト側の処理                            (3)
}
別の処理                                    (4)
   :
```

図 6-3-9 (1)

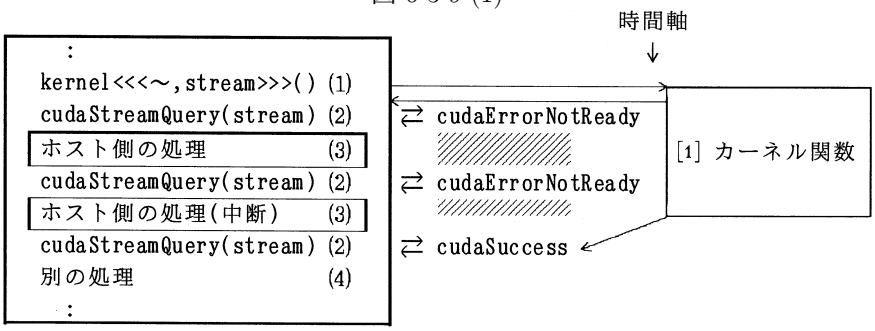


図 6-3-9 (2)

6-3-3　コピーとカーネル関数の処理のオーバーラップ

本節では、コピーとカーネル関数の処理をオーバーラップ（同時に実行）させて、時間のかかるコピーの時間を隠蔽し、実行時間を短縮する方法を説明します。

複数のストリームの例

コピーとカーネル関数のオーバーラップを行うためには、複数のストリームが必要となります。その場合、異なるストリームに所属する関数の間に依存関係があってはいけません。以下に例を示します。

【例 1】前節で説明したように、図 6-3-10 (1) で、配列 A と dA を処理する 3 つの非同期関数は、実行順序に依存関係があるため、同一ストリームにする必要があります。同様に、図 6-3-10 (2) で、配列 B と dB を処理する 3 つの非同期関数も、同一ストリームにする必要があります。一方、図 6-3-10 (1) の各関数と図 6-3-10 (2) の各関数の間には依存関係がありません。この場合、図 6-3-10 (1) の 3 つの関数をストリーム ID ①、図 6-3-10 (2) の 3 つの関数をストリーム ID ②のように、複数のストリームにすることができます (6 つの関数を同一ストリームにすることもできますが、その場合、オーバーラップはできません)。

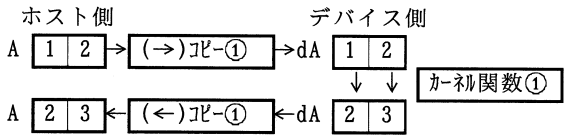


図 6-3-10 (1)

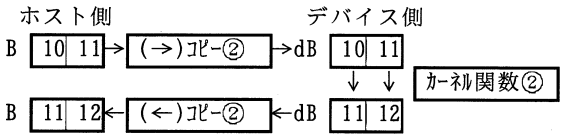


図 6-3-10 (2)

【例 2】図 6-3-11 (1) の 3 つの非同期関数は配列 A と dA の 1 つ目の要素を処理し、図 6-3-11 (2) の 3 つの非同期関数は 2 つ目の要素を処理するので、図 6-3-11 (1) の各関数と図 6-3-11 (2) の各関数の間には依存関係がありません。この場合、図 6-3-11 (1) の 3 つの関数をストリーム ID ①、図 6-3-11 (2) の 3 つの関数をストリーム ID ②のように、複数のストリームにすることができます。

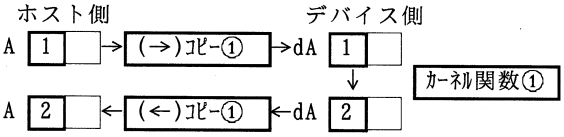


図 6-3-11 (1)

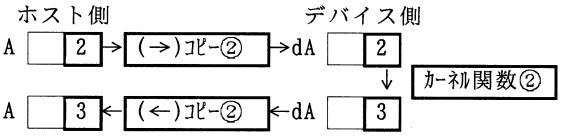


図 6-3-11 (2)

複数のストリームが存在する場合の動作概要

図 6-3-12 の左図のプログラムには、複数のストリーム (①と②) が含まれています。各関数は全て非同期関数なので一気にコールされ、コールされた順番に待ち行列に入ります。待ち行列に入った各関数を、以後タスクと呼ぶことにします。

スケジューラーは、次のページで説明する方法で、待ち行列から、実行するタスクを選択します。スケジューラーが、図 6-3-12 のように、コピーとカーネル関数のタスクを同時に選択した場合、2 つのタスクはオーバーラップして同時に実行します。その結果、時間のかかるコピーの時間が隠蔽され、実行時間が短縮します。

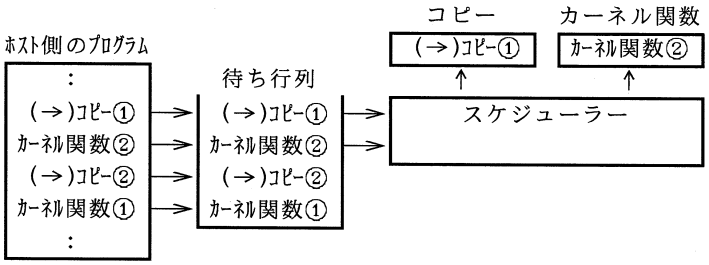
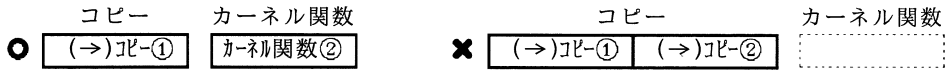


図 6-3-12

複数のタスクが同時に実行できる条件

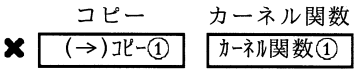
複数のタスクが同時に実行できる条件について説明します。

[条件 1] コピーとカーネル関数のタスクは、それぞれ 1 時点で最大 1 つ、実行することができます。

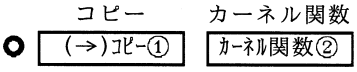


[条件 2] ストリーム ID [0]のタスクは、他のタスクと同時に実行することはできません。

[条件 3] ストリーム ID が同じタスクは、同時に実行することができません。



以上をまとめると、一方がコピー、他方がカーネル関数で、ストリーム ID が異なる (ただし[0]以外) タスクが、同時に実行できる可能性があります。



スケジューラーが待ち行列からタスクを選択する方法

待ち行列内に多くのタスクが存在する場合、スケジューラーは上記の条件に加え、下記の条件に従って実行するタスクを選択します。

現在、図 6-3-13 (1) に示すように「(→) コピー①」のタスクが実行中だとします (「カーネル関数①」のタスクが実行中の場合は、以下の説明中のコピーとカーネル関数が逆になります)。上記 [条件 1] より、本例で同時に実行可能なのは、1 つのカーネル関数のタスクです。

待ち行列の例を図 6-3-13 (2)〜(5) に示します。以下の説明で、待ち行列から選択され、実行を開始したタスクは、待ち行列内にそのまま存在し（図中の「実」)、実行が終了したら待ち行列から除去されるとします。

[条件 4] 図 6-3-13 (2) のように、待ち行列内にカーネル関数のタスクが複数ある場合、カーネル関数で一番上の「カーネル関数①」のみが候補になります。「カーネル関数①」が他の条件 (本例では [条件 6]) を満足せずに選択されなかった場合、その下の「カーネル関数②」は他の条件を満足したとしても候補になることはできません。

[条件 5] 待ち行列が図 6-3-13 (3) の場合、カーネル関数のタスクの一番上にある「カーネル関数②」が候補になります。本例のように、候補のタスクより上に、ストリーム ID [0]のタスクが存在する場合、候補のタスクは選択されません。

[条件 6] 待ち行列が図 6-3-13 (4) の場合、カーネル関数のタスクの一番上にある「カーネル関数②」が候補になります。本例のように、候補のタスクより上に、候補のタスクと同じストリーム ID を持つタスクが存在する場合、候補のタスクは選択されません。

待ち行列が図 6-3-13 (5) の場合、カーネル関数のタスクの一番上にある「カーネル関数③」は、[条件 4]、[条件 5]、[条件 6] を全て満足するので選択されます。

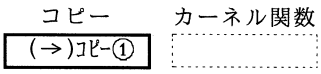


図 6-3-13 (1)

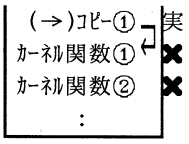


図 6-3-13 (2)

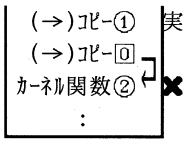


図 6-3-13 (3)

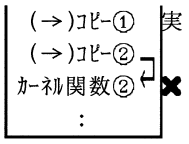


図 6-3-13 (4)

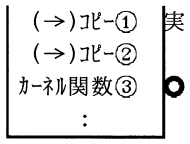


図 6-3-13 (5)

ストリームの例

図 6-3-14 (1) の左図のプログラムで、コピーとカーネル関数の実行時間が同じだとします。以下で説明するように、このプログラムを実行したときのタイムチャートは、図 6-3-14 (2) のようになります (T1，T2，...は各時刻を表します)。また、各時刻における待ち行列の状態は、図 6-3-14 (3) のように変化します。

(1) 図 6-3-14 (1) を実行すると、6 つの非同期関数が一気にコールされ、待ち行列は図 6-3-14 (3) の T1 の状態になります。↓は同一ストリーム内の各タスク間の依存関係を示します。なお、本例ではストリーム ID 0のタスクは存在しないので、ストリーム ID 0に関する依存関係はありません。

(2) T1 の待ち行列の中で一番上にある「(→) コピー①」が選択されます（選択されたタスクを○で示します)。

(3) コピーと同時に実行できるカーネル関数のタスクのうち、待ち行列内の一番上にある「カーネル関数①」が候補になります。しかし「カーネル関数①」は、上から↓が来ている (ストリーム ID ①の依存関係がある) ため選択されません。その結果、時刻 T1 では、図 6-3-14 (2) に示すように「(→) コピー①」のみが動きます。

(4)「(→) コピー①」の実行が終了したら、図 6-3-14 (3) の T2 に示すように、「(→) コピー①」と、そこから下に出ている↓を、待ち行列から除去します。

(5) 時刻 T2 の待ち行列の中で、一番上にある「(→) コピー②」が選択されます。

(6) コピーと同時に実行できるカーネル関数のタスクのうち、待ち行列内の一番上にある「カーネル関数①」が候補になり、上から↓が来ていない (ストリーム ID ①の依存関係がない) ため選択されます。その結果、時刻 T2 では、図 6-3-14 (2) に示すように「(→) コピー②」と「カーネル関数①」がオーバーラップして同時に動きます。

(7) 同様の処理を、待ち行列からタスクがなくなるまで繰り返します。その結果、タイムチャートは図 6-3-14 (2) となり、着色した部分でコピーとカーネル関数がオーバーラップし、実行時間が短縮します。

図 6-3-14 (1) の、関数をコールする順番を変えた図 6-3-15 (1) では、図 6-3-15 (2) (3) に示すように、オーバーラップは発生せず、図 6-3-14 (1) よりも速度が（一般に）遅くなります。このように、非同期関数をコールする順番によって、オーバーラップの効率が変わることがあります。

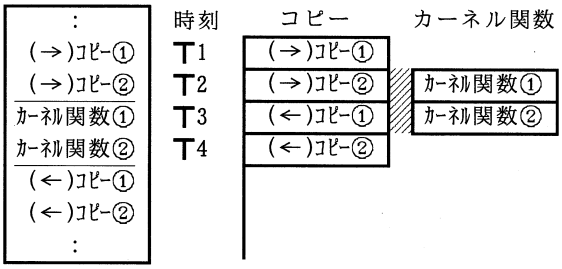


図 6-3-14 (1)　　図 6-3-14 (2)

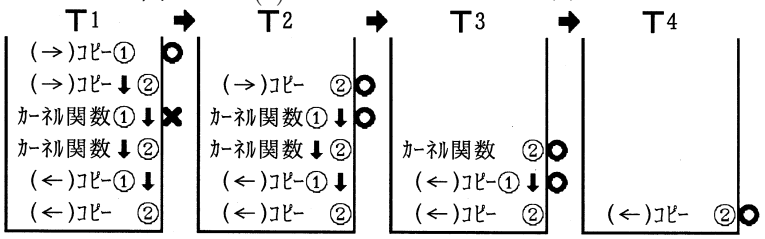


図 6-3-14 (3)

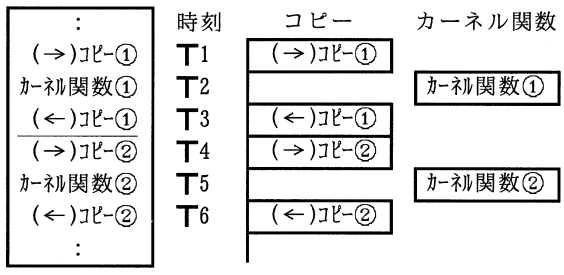


図 6-3-15 (1)　　図 6-3-15 (2)

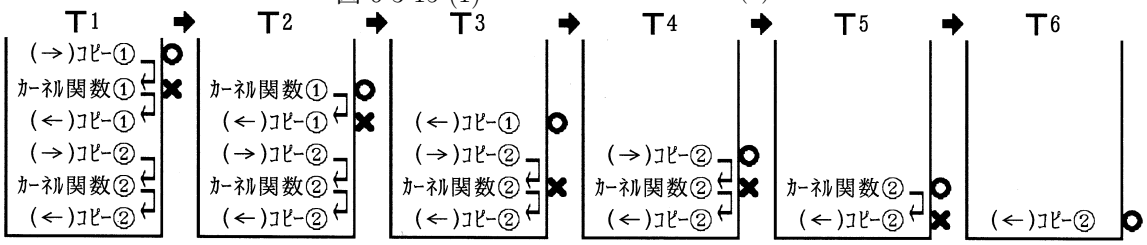


図 6-3-15 (3)

ストリームのプログラム例

図 6-3-14 (1) (2) の実際のプログラム例を図 6-3-16 に示します。このプログラムは、前述の図 6-3-11 (1) (2) と同様の処理を行います。図 6-3-17 に示すように、配列 A の 24 個の要素を処理し、前半の 12 個をストリーム①が、後半の 12 個をストリーム②が担当します。各ストリームでは、ブロック数を 3 (0，1，2)、ブロック内のスレッド数を 4 (0, 1, 2, 3) でカーネル関数を実行します。

以下で図 6-3-16 の説明をします。なお、説明を簡単にするため、要素数 (24) は「ストリーム数 (2) × ブロック数 (3)」で割り切れるとし、割り切れない場合のエラーチェックは省略します。

- (1) で全要素数を N (本例では 24) に、(2) でストリーム数を NSTREAMS (本例では 2) に設定します。
- (4) でストリームあたりの要素数 NS (本例では 12) を求めます。
- (5) で 2 つのストリームの ID を保管する配列 stream[2] を宣言します。(6) を実行すると、配列 stream に、2 つのストリームの ID が設定されます。以下ではストリーム ID を①，②として説明しますが、実際には違う値が設定されます。
- (9) で cudaMemcpyAsync を使用するため、ホスト側の配列 A を、通常の配列宣言や malloc ではなく、(7) で宣言します (6-3-1 節参照)。
- (8) でデバイス側の配列 dA を確保します。
- (9) で、非同期関数 cudaMemcpyAsync を使用して、ホスト側の配列 A からデバイス側の配列 dA に、データをコピーします。一番最後の引数にストリーム ID を指定します。図 6-3-17 の (9) に示すように、ストリーム①では A[0] からの 12 要素を dA[0] からの 12 要素にコピーし、ストリーム②では A[12] からの 12 要素を dA[12] からの 12 要素にコピーします。各ストリームでの A と dA の最初の要素を (9) の二重線で指定します。
- (10) でストリーム①と②がそれぞれカーネル関数を実行します。3 番目の引数（動的に確保するシェアードメモリの大きさ）(3-6 節参照) はゼロとし、4 番目の引数にストリーム ID を指定します。
  (10) と (3) の二重線に示すように、ストリーム①では、実際の dA[0] がカーネル関数内の dA[0] に対応し、ストリーム②では、実際の dA[12] がカーネル関数内の dA[0] に対応します (図 6-3-17 参照)。
  本例では全ブロック数が 6 でストリーム数が 2 なので、(10) の波線では、各ストリームでのブロック数を 6/2 (= 3) としています。
- (11) で、非同期関数 cudaMemcpyAsync を使用して、デバイス側の配列 dA からホスト側の配列 A に、データをコピーします。一番最後の引数にストリーム ID を指定します。
- ストリーム①，②がそれぞれ (9)，(10)，(11) を実行するので、合計 6 個の非同期関数が一気にコールされます。従って、（本例では）計算結果を参照する (13) より前の (12) で、全てのタスクの同期を取る必要があります (4-1 節参照)。なお (12) で、全てのタスクではなく、特定のストリーム (例えば stream[0]) に所属するタスクのみの同期を取りたい場合は、cudaStreamSynchronize(stream[0]) とします。
- (14) でストリームを解放します。
- (15) で配列 A を、(16) で配列 dA を解放します。

補足

- ストリームを使用した場合、プログラムのロジックやデータ量によって、効果が出る場合と、却って遅くなる場合があります。
- タスク (コピーを行う CUDA 関数またはカーネル関数) は、1 回実行するごとにオーバーヘッドがかかります。ストリーム数を多くすると、コピーを行う CUDA 関数とカーネル関数が同時に動く可能性は高くなりますが、タスクの実行回数が増えるので、オーバーヘッドが増加します。ストリーム数は、試行錯誤で最適な数に設定して下さい。
- 4-5 節で説明したタイムスタンプを使用すると、各タスクがどの順番に、（逐次または同時に）処理されたかを、ある程度知ることができます。

```
#define N (24)                                          (1)
#define NSTREAMS (2)                                    (2)
__global__ void kernel(float *dA){                      (3)
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;
}
int main(void){
  float *A;
  float *dA;
  int NS = N/NSTREAMS;                                  (4)
  size_t sizeN  =  N*sizeof(float);
  size_t sizeNS = NS*sizeof(float);
  cudaStream_t stream[NSTREAMS];                        (5)
  for(i=0;i<NSTREAMS;i++){
    cudaStreamCreate(&stream[i]);                       (6)
  }
  cudaHostAlloc((void**)&A,sizeN,                       (7)
                cudaHostAllocDefault);                  (7)
  配列Aにデータを設定します。
  cudaMalloc((void**)&dA,sizeN);                        (8)
for(i=0;i<NSTREAMS;i++){
  cudaMemcpyAsync(&dA[i*NS],&A[i*NS],                   (9)
  sizeNS,cudaMemcpyHostToDevice,stream[i]);
}
for(i=0;i<NSTREAMS;i++){
  kernel<<<6/NSTREAMS,4,0,stream[i]>>>                  (10)
                                  (&dA[i*NS]);
}
for(i=0;i<NSTREAMS;i++){
  cudaMemcpyAsync(&A[i*NS],&dA[i*NS],                   (11)
  sizeNS,cudaMemcpyDeviceToHost,stream[i]);
}
cudaThreadSynchronize();                                (12)
配列Aを参照します。                                     (13)
for(i=0;i<NSTREAMS;i++){
  cudaStreamDestroy(stream[i]);                         (14)
}
cudaFreeHost(A);                                        (15)
cudaFree(dA);                                           (16)
  :
```

図 6-3-16

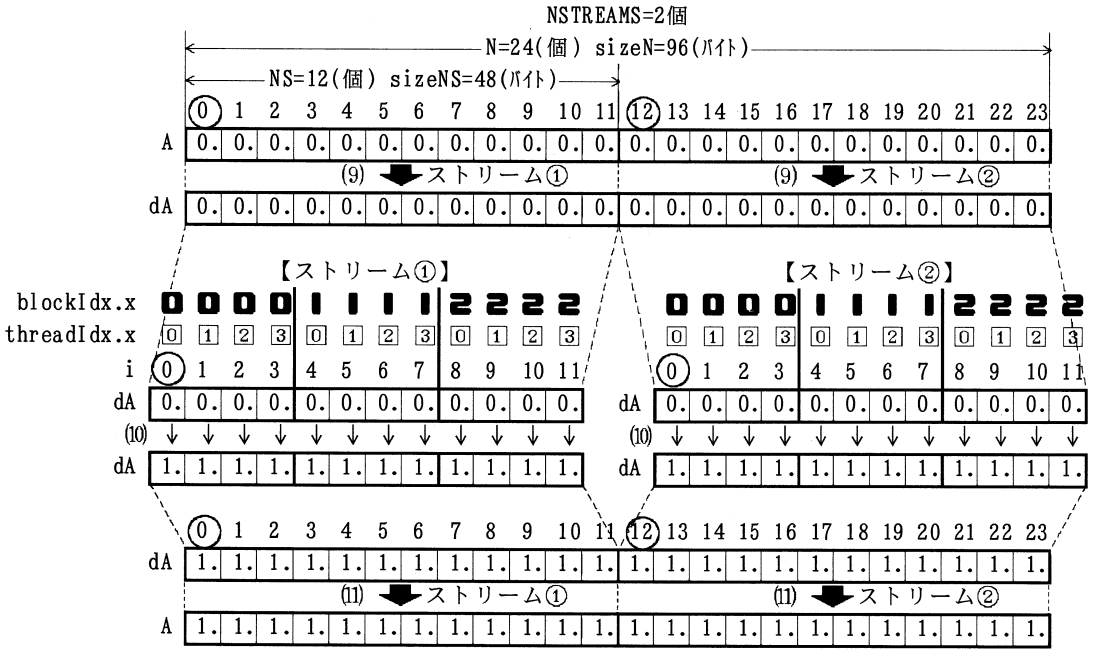


図 6-3-17

6-4　イベント

本節では、6-3 節で説明したストリームと関係の深い、イベントについて説明します。以下ではイベントを、経過時間を測定するタイマールーチン (4-4 節参照) として使用する方法を説明します。

- 図 6-4-1 の (3) のカーネル関数の経過時間を測定するとします。
- (0) で、(3) の実行直前と直後の時刻を保管する変数 start，stop（名前は任意）を宣言します。なお、実際には、変数 start，stop に時刻の値そのものが保管される訳ではありません。
- (1) で変数 start，stop をイベントとして登録します。
- 測定対象の (3) の直前に (2) を、直後に (4) をコールします。(2)，(4) は、カーネル関数、cudaMemcpyAsync などと同様に非同期関数で、コールするとすぐにホスト側に制御が戻ります。(2)，(3)，(4) が同一ストリーム (6-3 節参照) になるように、(2)，(4) の下線部にストリーム ID を指定します。本例では、(3) のストリーム ID が0（デフォルト値）なので、(2)，(4) の下線部も0とします。なお、(3) が同期関数 (例えば cudaMemcpy) でストリーム値を持たない場合も、(2)，(4) の下線部を0とします。
- (2)，(3)，(4) が同一ストリームなので、図の右側に示すように、[2]，[3]，[4] の順に 1 つずつ実行が行われます。[2] が実行されると、その時点の時刻を変数 start に保管します。同様に、[4] が実行されると、その時点の時刻を変数 stop に保管します。なお、図では [2]，[4] の処理時間が長く見えますが、実際は一瞬で終了します。
- (5) の前に (6) を説明します。引数に start と stop を指定して (6) を実行すると、start と stop の間にかかった経過時間、つまり [3] の経過時間が、ミリ秒 (1/1000 秒) 単位で、単精度の実数 elapse（名前は任意）に戻ります。
- 次に (5) を説明します。本例では (2)，(3)，(4) が非同期関数なので一気にコールが終了します。(4) をコールした後すぐに (6) を実行すると、[2]，[3]，[4] がまだ完了しておらず、経過時間を正しく測定できません。そこで (6) の前に (5) を実行します。すると、(5) の引数 (stop) に指定した [4] のイベントが完了するまで、ホスト側のプログラムは (5) で待機します。従って、(5) の後で (6) を実行すれば、[2]，[3]，[4] が完了していることが保証されるので、経過時間を正しく測定することができます。
- (7) で経過時間を秒に変換して出力し、(8) でイベントを解放します。
- なお、イベントで指定したタスクが完了しているかどうかを照会する、cudaEventQuery という CUDA 関数 (6-3 節で説明した cudaStreamQuery に相当) が提供されています。

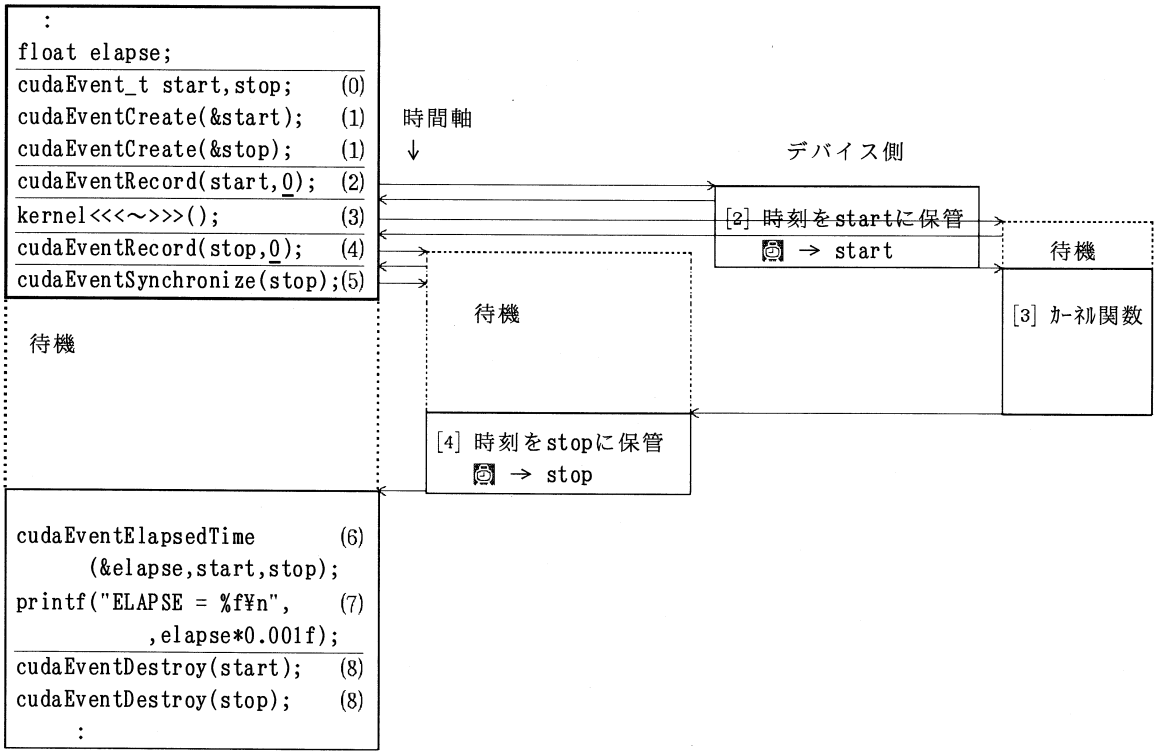


図 6-4-1

6-5　ワープ・ダイバージェント

カーネル関数内に if 文が含まれる場合の動作について、図 6-5-1 のプログラムで説明します。

- ③に示すように 1 ブロック、ブロックあたり 32 スレッド (= 1 ワープ) でカーネル関数を実行するとします。前述のように、連続する 32 スレッドは同一のワープに属し、ほぼ同時に同じ動作を行います。
- 図 6-5-3 (1) に示すように、配列 dINDEX の全要素に「1」が設定されているとします。
- 図 6-5-1 の①で、配列 dINDEX の各要素について、if 文の条件式が真 (o) か偽 (x) かが判定されます。図 6-5-3 (1) では全ての要素が真なので、結果は図 6-5-3 (1) の①のようになります。本書では、結果が入る二重線の部分をマスクベクトルと呼びます (CUDA の正式な用語ではありません)。
- 図 6-5-1 の②で、各スレッドは、マスクベクトル内の自分が担当する要素が「o」なら処理を行い、「x」なら処理を行いません。図 6-5-3 (1) では、②の着色した部分に示すように、全要素が処理を行います。

図 6-5-3 (2) では、配列 dINDEX の全要素に「0」が設定されています。この場合、図 6-5-1 の②では、図 6-5-3 (2) の一番下に示すように、全要素が処理を行わず、処理時間はかかりません。

図 6-5-3 (3) では、配列 dINDEX の左端の要素のみ「1」で、他は全て「0」が設定されています。この場合、図 6-5-1 の②では、図 6-5-3 (3) の一番下に示すように、左端の要素が処理を行います。このとき、図 6-5-3 (1) のように全要素を処理したのと、ほぼ同じ処理時間がかかるので注意して下さい。

図 6-5-2 のプログラムでは、if 文が真のとき⑤の、偽のとき⑥の処理を行います。配列 dINDEX の値によって、図 6-5-4 (1)〜(3) のように処理が行われます。図 6-5-4 (1) は⑤のみ、図 6-5-4 (2) は⑥のみを実行しますが、図 6-5-4 (3) は⑤と⑥を両方実行するので、図 6-5-4 (1) (2) と比べて約 2 倍の処理時間がかかります。

図 6-5-3 (3)、図 6-5-4 (3) のように、同一ワープ (32 スレッド) 内に、if 文の条件式が真と偽のスレッドが存在することを、ワープ・ダイバージェント (divergent: 分岐する) と呼び、計算時間がかかります。できれば事前に要素の並べ変えを行ない、同一ワープ内の全スレッドが、if 文で同じ判定になるようにして下さい。

```
__global__ void kernel(float *dA, int *dINDEX){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if(dINDEX[i]==1){                ①
    dA[i] = dA[i] + 1.0;           ②
  }
}
```

```
    :
  kernel<<<1,32>>>(dA,dINDEX);     ③
    :
```

図 6-5-1

```
__global__ void kernel(float *dA, int *dINDEX){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if(dINDEX[i]==1){                ④
    dA[i] = dA[i] + 1.0;           ⑤
  }else{
    dA[i] = dA[i] - 1.0;           ⑥
  }
}
```

図 6-5-2

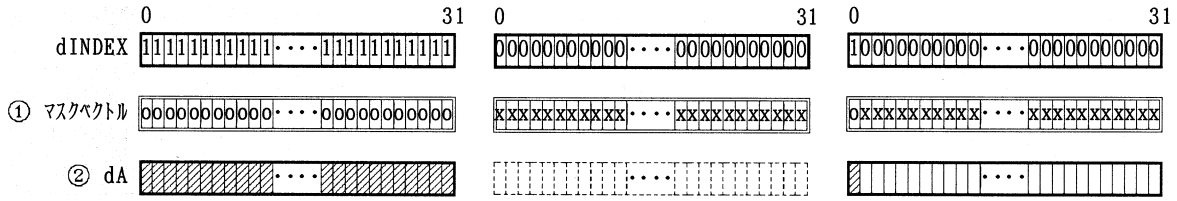


図 6-5-3 (1)

図 6-5-3 (2)

図 6-5-3 (3)

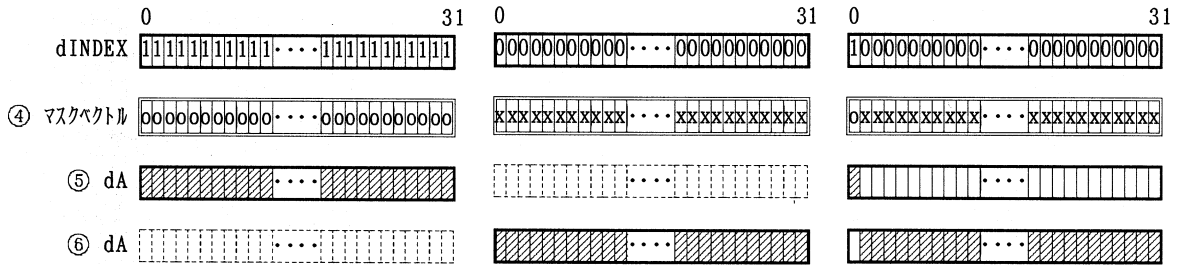


図 6-5-4 (1)

図 6-5-4 (2)

図 6-5-4 (3)

ワープ・ダイバージェントが発生しているかどうかは、CUDA のプロファイラーで調べることができます。図 6-5-5 (1) の①の環境変数を設定し、ファイル config に②を設定してプログラムを実行すると、③のファイル log が作成されます。下線部は、ワープあたり、if 文でワープ・ダイバージェントが発生した回数を示し、図 6-5-3 (1) (2)，図 6-5-4 (1) (2) では 0 回、図 6-5-3 (3)，図 6-5-4 (3) では 1 回となります (詳細は 4-5 節参照)。

次に、図 6-5-6 のプログラムで説明します。ブロック数 1、ブロック内のスレッド数 128 でカーネル関数を実行した場合、ワープの数は 4 (ワープ 0〜3) となります。また配列 dINDEX に 0〜7 の値が図 6-5-7 のように設定されているとします。④のループの k を 0，1，2，...，7 と反復させると、⑤の if 文により、⑥は、dINDEX[i] の値が 0 以上，1 以上，2 以上，...，7 以上の場合に実行が行われます。各 k で、⑥の処理が行われる要素を図 6-5-7 の着色部分に示します。⑥の処理が行われる回数は、実線で囲んだ長方形が示すように 12 個となります。

一方、配列 dINDEX に 0〜7 の値が図 6-5-8 のように設定されているとします。④で k = 0 のとき、⑥で全要素が処理され、k = 1, 2, 3, 4, ... のとき、⑥で 1，2，4，8，...飛びに要素が処理されます。⑥の処理が行われる回数は、実線で囲んだ長方形が示すように 27 個で、図 6-5-7 より多いので、速度は遅くなります。

本例では、図中の実線で囲んだ長方形内に、着色した要素と着色していない要素が含まれている場合、ワープ・ダイバージェントが発生します。図 6-5-7 では 5 回 (k = 3〜7 の部分)、図 6-5-8 では 23 回 (k = 1〜7 の部分) となり、この値が図 6-5-5 (2) の③の下線部に表示されます。

本例のように、計算が進むにつれて処理する要素数が半分ずつ減少する演算は、合計、内積、最大（小）値を求める計算などに現れます (8-3 節参照)。可能であれば、図 6-5-7 のように処理した方が速くなります。

```
export CUDA_PROFILE=1                    ①
export CUDA_PROFILE_CONFIG=config        ①
export CUDA_PROFILE_LOG=log              ①
```

図 6-5-5 (1)

```
config  divergent_branch                 ②
   log  ~ divergent_branch=[ 1 ]         ③
```

図 6-5-5 (2)

```
__global__ void kernel(float *dA, int *dINDEX){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  for(int k=0;k<8;k++){                  ④
    if(k<=dINDEX[i])                     ⑤
      dA[i] = dA[i] + 1.0f;              ⑥
  }
}
```

図 6-5-6

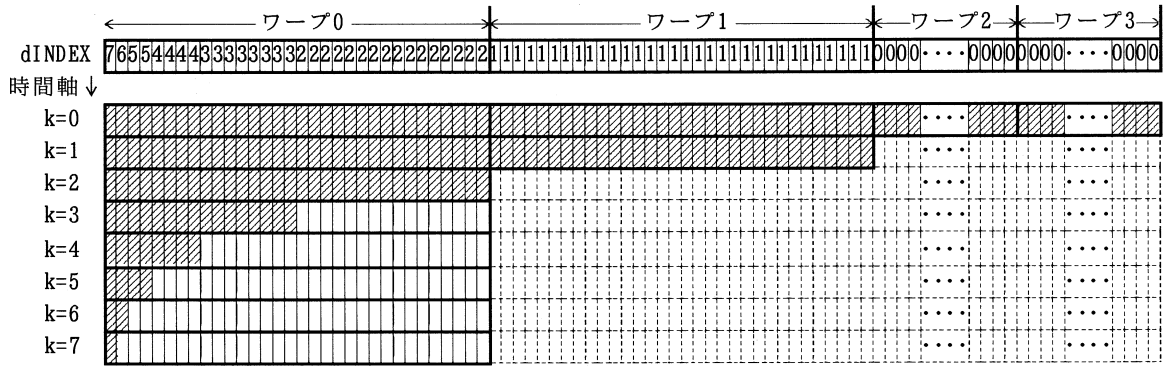


図 6-5-7

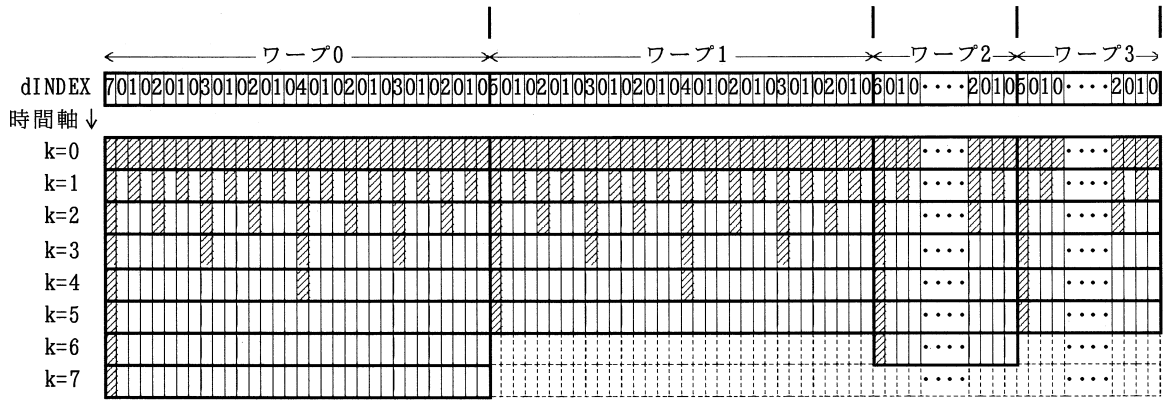


図 6-5-8

6-6　カーネル関数のチューニング

本節では、カーネル関数の、プログラムレベルでのチューニングについて説明します。

- 一般にチューニングを行うと、桁落ちや丸めによる誤差によって、計算結果が若干変わることがありますので、計算結果に影響がない場合のみ採用して下さい。
- 本節のチューニングは、プログラムのロジックによっては、コンパイラが自動的に行う場合もあります。チューニング前とチューニング後で、計算時間が速くなったかどうかを確認し、明らかに速くなった場合のみ採用して下さい。
- 一般の計算機（またはホスト側のプログラム）のチューニング方法については、参考文献 [2]（付録参照）を参照して下さい。このチューニング方法は、カーネル関数にも適用できる場合もあります。

Best Practices Guide の例

「CUDA C Best Practices Guide」（付録参照）の 5.1 節に掲載されている例を説明します。なお、下記のうち、組込関数については、単精度版の例を示します。倍精度版、および各関数の使用方法の詳細については、「CUDA C Programming Guide」の Appendix C.（付録参照）を参照して下さい。

単精度の定数

下記の①は、倍精度定数 1.0 から単精度定数 1.0f への型変換が必要となります。②は型変換が不要なので、②の方が速くなります。

```
float a;
a = 1.0;  ①
a = 1.0f; ②
```

2 のベキ乗の除算

下記の①の下線（二重線）が 1 以上の整数で、下線（実線）が 2 のベキ乗 (2, 4, 8, ...) の場合、②のようにビットの右シフト演算にした方が速くなります。②の「>>3」は、ビットの右方向へのシフトを 3 回行うという意味です。例えば 10 進数の 10 は 2 進数の 00001010 で、ビットの右シフトを 3 回行うと、以下のように 10 進数の 1 になります。「>>」は四則演算より演算の優先順位が低いので、本例では②にカッコを付けて下さい。

なお、下記の①のように、下線（実線）に明示的に値（数字）が指定されている場合、コンパイラが自動的に①を②に置き換える場合があります。

00001010 → 00000101 → 00000010 → 00000001

10進数の10　　10進数の1

```
int i;
i = 10/8 + 1;     ①
i = (10>>3) + 1;  ②
```

下線部に 8=2³ の3を指定します。

2 のベキ乗の剰余

下記の①の下線（二重線）が 1 以上の整数で、下線（実線）が 2 のベキ乗 (2, 4, 8, ...) の場合、②のようにビットの AND 演算にした方が速くなります。②の「&」は、以下のように、2 進数の各ビットの AND を取ることを意味します。「&」は四則演算より演算の優先順位が低いので、本例では②にカッコを付けて下さい。

なお、下記の①のように、下線（実線）に明示的に値（数字）が指定されている場合、コンパイラが自動的に①を②に置き換える場合があります。

00001010 & 00000111 = 00000010

10進数の10　10進数の7　10進数の2

```
int i;
i = 10%8 + 1;     ①
i = (10&7) + 1;   ②
```

下線部に 8-1=7 を指定します。

分母の平方根

下記の①のように、分母に平方根がある場合は、②の組込関数に変えた方が速くなります。①の除算は②では乗算になります。

```
float a,x;
a = 1.23f/sqrtf(x);  ①
a = 1.23f*rsqrtf(x); ②
```

同じ値の sin と cos を両方求める場合

下記の①のように、同じ値 x の sin と cos を両方求める場合、②の組込関数に変えた方が速くなります。

```
float a,b,x;
a = sinf(x);          ①
b = cosf(x);          ①
sincosf(x,&a,&b);     ②
```

高速版の組込関数

CUDA では、sinf などの組込関数の、高速版の組込関数が提供されています。下記の①は純正の組込関数です。一方②は高速版の組込関数で、SFU (Super Function Unit) という、ストリーミングマルチプロセッサー上にある専用の装置で計算を行うため、①より高速になりますが、若干精度が悪くなります。高速版の組込関数は、カーネル関数内のみで使用可能です。高速版の組込関数の一覧は、「CUDA C Programming Guide」Appendix C.2（付録参照）を参照して下さい。

一方、③の下線部を指定してコンパイル／リンクすると、プログラム内の純正の組込関数（高速版が提供されている関数のみ）が、自動的に高速版の組込関数に変換されます。変換される関数の一覧は「CUDA C Programming Guide」Appendix B.7（付録参照）を参照して下さい。

```
float a,x;
a =   sinf(x);  ①
a = __sinf(x);  ②
```

```
nvcc（最適化ｵﾌﾟｼｮﾝ）-use_fast_math test.cu ↵  ③
```

高速版の組込関数を使用する場合、精度の相違が計算結果に影響を及ぼす可能性があるので注意して下さい。参考のため、下記に、各組込関数の計算結果を表示します。純正の sinf(x) の方が、高速版の __sinf(x) よりも、倍精度の sin(x) の値に近いようです。

x	単精度（純正） sinf(x)	単精度（高速版） __sinf(x)	倍精度 sin(x)
1.0	0.8414710164	0.8414708972	0.8414709848
2.0	0.9092974663	0.9092973471	0.9092974268
3.0	0.1411200017	0.1411199421	0.1411200081
4.0	-0.7568024993	-0.7568023801	-0.7568024953
5.0	-0.9589242935	-0.9589242935	-0.9589242747
6.0	-0.2794154882	-0.2794155180	-0.2794154982
7.0	0.6569865942	0.6569864154	0.6569865987
8.0	0.9893582463	0.9893582463	0.9893582466
9.0	0.4121184945	0.4121187925	0.4121184852
10.0	-0.5440210700	-0.5440207720	-0.5440211109

最適化

以下に示す例は、通常の計算機のコンパイラでは自動的に最適化する可能性が高いですが、カーネル関数側のコンパイラは自動的に最適化せず、手作業でチューニングすると速くなりました。以下の例を見る限り、カーネル関数側のコンパイラの最適化能力は、（現時点では）通常の計算機のコンパイラよりも低いようです。

コンパイラがどのように最適化を行ったかについては、アセンブラリストで確認することができます (2-7 節参照)。

なお、チューニング方法によっては、一時変数を使用するため、使用するレジスターの数が増えてしまい、6-1 節で述べた、ストリーミング・マルチプロセッサー上に同時に存在できるワープの数が減り、遅くなる可能性があります。

グローバルメモリからのロードの低減

グローバルメモリからのロード（またはストア）は時間がかかります。

図 6-6-1 (1) では、下線部に示すように、グローバルメモリ上の同じ要素 dX[0] を 3 回ロードしています。これを図 6-6-1 (2) のように変更すると、dX[0] を一度だけロードするため速くなります (変数 temp はレジスターに置かれるので、グローバルメモリからのロードは不要です)。このように、(A) 同じ計算を複数回行っている場合、一回だけ実行するようにすると、速くなる場合があります。

一方、図 6-6-2 (1) では、ループが反復するたびに、下線部で毎回ロードを行っています。これを図 6-6-2 (2) のように変更すると、dX[0] を一度だけロードするため速くなります。このように、(B) ループ反復とは関係のない計算をループの外に出して一回だけ実行するようにすると、速くなる場合があります。

次ページの例も、(A) と (B) の両方の形式に適用できますが、(B) の形式で説明します。

```
__global__ void kernel
  (float *dA,float *dB,float *dC,float *dX){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + dX[0];
  dB[i] = dB[i] + dX[0];
  dC[i] = dC[i] + dX[0];
}
```

図 6-6-1 (1)　×

```
__global__ void kernel
  (float *dA,float *dB,float *dC,float *dX){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float temp = dX[0];
  dA[i] = dA[i] + temp;
  dB[i] = dB[i] + temp;
  dC[i] = dC[i] + temp;
}
```

図 6-6-1 (2)　○

```
__global__ void kernel(float *dA,float *dX){
  for(int k=0;k<N;k++){
    dA[k] = dA[k] + dX[0];
  }
}
```

```
      :
    kernel<<<1,1>>>(dA,dX);
      :
```

図 6-6-2 (1)　×

```
__global__ void kernel(float *dA,float *dX){
  float temp = dX[0];
  for(int k=0;k<N;k++){
    dA[k] = dA[k] + temp;
  }
}
```

```
      :
    kernel<<<1,1>>>(dA,dX);
      :
```

図 6-6-2 (2)　○

組込関数の削減

sinf などの組込関数は計算時間がかかります。図 6-6-3 (1) の、ループ反復とは関係のない sinf を、図 6-6-3 (2) のようにループの外に出して一度だけ実行するようにすると、速くなります。

```
__global__ void kernel(float *dA,float x){
  for(int k=0;k<N;k++){
    dA[k] = dA[k] + sinf(x);
  }
}
```

図 6-6-3 (1)　×

```
__global__ void kernel(float *dA,float x){
  float temp = sinf(x);
  for(int k=0;k<N;k++){
    dA[k] = dA[k] + temp;
  }
}
```

図 6-6-3 (2)　○

除算の乗算化

四則演算 (+，−，×，÷) では除算が一番計算時間がかかります。図 6-6-4 (1) の下線部の除算を図 6-6-4 (2) のように乗算に変換すると、速くなります。

また、図 6-6-5 (1) の、ループ反復とは関係のない除算を、図 6-6-5 (2) のようにループの外に出して一度だけ実行するようにすると、速くなります。

```
__global__ void kernel(float *dA){
  for(int k=0;k<N;k++){
    dA[k] = dA[k]/2.5f;
  }
}
```

図 6-6-4 (1)　×

```
__global__ void kernel(float *dA){
  for(int k=0;k<N;k++){
    dA[k] = dA[k]*0.4f;
  }
}
```

図 6-6-4 (2)　○

```
__global__ void kernel(float *dA,float x){
  for(int k=0;k<N;k++){
    dA[k] = dA[k]/x;
  }
}
```

図 6-6-5 (1)　×

```
__global__ void kernel(float *dA,float x){
  float temp = 1.0f/x;
  for(int k=0;k<N;k++){
    dA[k] = dA[k]*temp;
  }
}
```

図 6-6-5 (2)　○

並列化できない／並列化しにくい部分のデバイス側での実行

カーネル関数で実行するのは、必ずしも並列化された部分だけではありません。一連の計算の一部に、並列化できない部分や並列化しにくい部分（例えば合計などの縮約演算）が含まれている場合、通常はその部分をホスト側のプログラムで計算しますが、以下のように、デバイス側で、例えばブロック ID = 0、スレッド ID = 0 のスレッドが代表して計算する方法もあります。

```
__global__ void kernel(〜){
    :
  if(blockIdx.x==0 && threadIDx.x==0) {
    並列化できない(しにくい)部分
  }
    :
}
```

6-7　ループアンローリング

ループアンローリングとは

ループの反復回数を減らし、その代わりにループ内のステートメント数を増やすことをループアンローリングと呼びます。例えば図 6-7-1 (1) の for ループを 2 段にアンローリングすると、図 6-7-1 (2) のようになります。ループ反復を 2 反復ごとに行い、その代わりにステートメントを 2 行とし、配列の添字を「i」、「i+1」とします。同様に 3 段にアンローリングすると図 6-7-1 (3) のようになります。6 段にアンローリングした場合は、元のループの反復回数が 6 回なので、図 6-7-1 (4) のようにループが展開されて消滅します。

```
   :
 float A[6],B[6];
 for(i=0;i<6;i++){
   A[i] = B[i];
 }
   :
```

図 6-7-1 (1)　アンローリングなし

```
   :
 for(i=0;i<6;i+=2){
   A[i]   = B[i];
   A[i+1] = B[i+1];
 }
   :
```

図 6-7-1 (2)　2 段にアンローリング

```
   :
 for(i=0;i<6;i+=3){
   A[i]   = B[i];
   A[i+1] = B[i+1];
   A[i+2] = B[i+2];
 }
   :
```

図 6-7-1 (3)　3 段にアンローリング

```
   :
 A[i]   = B[i];
 A[i+1] = B[i+1];
 A[i+2] = B[i+2];
 A[i+3] = B[i+3];
 A[i+4] = B[i+4];
 A[i+5] = B[i+5];
   :
```

図 6-7-1 (4)　ループの展開

（カーネル関数内のループに対して）ループアンローリングを行うと、ループ反復のオーバーヘッドが若干減少します。さらに、プログラムによっては以下のような効果があります。本書では (1) を目的とします。

(1) 1 重ループ、または多重ループの内側のループのアンローリングによって、コンパイラによる最適化（ソフトウェア・パイプライニングなど) が促進される可能性があります。

(2) 多重ループの外側のループのアンローリングによって、ロードとストアが削減される可能性があります。

手作業で行う方法

ループアンローリングは、手作業で行う方法とコンパイラに自動的に行なわせる方法があります。まず手作業で行う方法の概要を説明します。アンローリング前のプログラムを図 6-7-2 (1) に示します。一般の場合にも適用できるように、ループ反復の範囲は変数 (imin〜imax) になっています。

図 6-7-2 (1) のループを n (= 3) 段にアンローリングしたプログラムを図 6-7-2 (2) に示し、動作を図 6-7-2 (3) に示します。⑥はアンローリングして計算する主力のループで、3 段なので 3 つ飛び (本例では i = 0,3) に反復する代わりに、各反復で①，②，③を実行します。⑨は、割り切れない残った反復 (本例では i = 6,7) をアンローリングせずに計算するループです。

⑤でアンローリングの段数 n を 3 に設定し、⑥で⑧のループの上限 itemp (本例では 5) を求めます。3 段なので⑦の下線部でループ反復を 3 つ飛びにし、①を 2 行コピーして②，③とし、下線部を追加します。

```
#define N (8)
   :
 float A[N],B[N]
 int imin = 0;
 int imax = N-1;
 for(i=imin;i<imax+1;i++){
   B[i] = A[i];
 }
   :
```

図 6-7-2 (1)

```
   :
 int n = 3;          ⑤          ⑥
 int itemp = imax - (imax-imin+1)%n;
 for(i=imin;i<itemp+1;i+=n){   ⑦
   B[i]   = A[i];    ① ← 1段目
   B[i+1] = A[i+1];  ② ← 2段目    ⑧
   B[i+2] = A[i+2];  ③ ← n(=3)段目
 }
 for(i=itemp+1;i<imax+1;i++){
   B[i] = A[i];      ④            ⑨
 }
   :
```

図 6-7-2 (2)

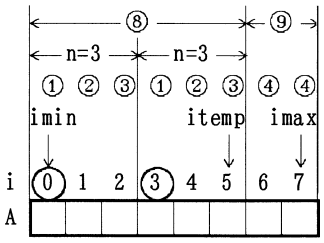


図 6-7-2 (3)

コンパイラに自動的に行わせる方法

次に、コンパイラに自動的にループアンローリングを行わせる方法について説明します。コンパイラに対して、指示行を使用してループアンローリングに関する指示を行うことができます。

指示行「#pragma unroll n」(n は無指定または 1 以上の整数) は、for 文の直前に指定し、その for 文のみに適用されます。図 6-7-3 と図 6-7-4 を例に、指示行の使用方法を説明します。図 6-7-3 はループの反復回数が下線に示すように（コンパイラにとって）既知の場合、図 6-7-4 は未知の場合です。

なお、ループによっては、以下の結果と異なる結果になるかもしれません。

```
#define N (10)
__global__ void kernel(float *dA){
  #pragma unroll n (nは無指定か1以上の整数)
  for(int k=0;k<N;k++){
    dA[k] = dA[k] + 1.0f;
  }
}
```

図 6-7-3　ループ反復回数が既知

```
__global__ void kernel(float *dA,int m){
  #pragma unroll n (nは無指定か1以上の整数)
  for(int k=0;k<m;k++){
    dA[k] = dA[k] + 1.0f;
  }
}
```

図 6-7-4　ループ反復回数が未知

「#pragma unroll」を指定しない場合

「#pragma unroll」を指定しない場合、図 6-7-3 では for 文は展開 (図 6-7-1 (4) 参照) されました。N = 40 までは同様に展開され、N = 41 以上では展開されませんでした。一方図 6-7-4 では展開されませんでした。

「#pragma unroll 1」を指定した場合

「#pragma unroll 1」は、直後の for 文のアンローリングを行わないことをコンパイラに指示します。図 6-7-3 と図 6-7-4 はどちらもアンローリングされませんでした。

「#pragma unroll n」(n は 2 以上の整数) を指定した場合

「#pragma unroll n」(n は 2 以上の整数) は、n 段にアンローリングすることをコンパイラに指示します。図 6-7-3 で、例えば「#pragma unroll 2」を指定した場合、2 段にアンローリングされます。N の値を大きくしてテストしたところ、「#pragma unroll n」の「n」が最大 5625 まではアンローリングされ、それ以上の値を指定すると、下記のメッセージが表示されてアンローリングされませんでした。

図 6-7-4 は図 6-7-3 と同じ結果になりました。

```
./test.cu(24): Advisory: Loop was not unrolled, too much code expansion
          ↳ファイルtest.cu内の、for文の行番号
```

「#pragma unroll」を指定した場合

「#pragma unroll」は、直後の for 文を展開 (図 6-7-1 (4) 参照) することをコンパイラに指示します。図 6-7-3 では N が最大 5625 までは展開され、それ以上では上記のメッセージが表示され、展開されませんでした。図 6-7-4 では反復回数が不明なので、展開されませんでした。

デバイス側のコンパイルオプション

前ページの結果は全て、デバイス側のコンパイラの最適化オプションがデフォルト値「-O3」の場合です (2-7 節参照)。下記の下線部のように「-O2」以下を指定した場合、上記のいずれの場合もアンローリングは行われませんでした。なお、「-O2」を指定してコンパイルした場合、①のメッセージが表示されます。

```
nvcc (最適化ｵﾌﾟｼｮﾝ) -Xopencc -O2 test.cu ↵
./test.cu(24): Advisory: Loop was not unrolled, cannot deduce loop trip count  ①
```

ループアンローリングが行われたかどうかを知る方法

コンパイラに自動的にアンローリングを行わせる場合、実際にアンローリングが行われたかどうかを知る方法を説明します。図 6-7-5 (1) の下線部を付けてコンパイルすると (test.cu はカーネル関数のみでも可)、アセンブラリストが（下記の例では）test.ptx というファイルに出力されます。

図 6-7-3 のプログラムで、ループがアンローリングされなかった場合は、図 6-7-5 (2) のように表示され、例えば 3 段にアンローリングされた場合は、図 6-7-5 (3) のように同じ命令パターンが 3 回表示されます。アセンブラリストの見方については、2-7 節を参照して下さい。

```
nvcc (最適化ｵﾌﾟｼｮﾝ) -ptx test.cu ↵
```

図 6-7-5 (1)

```
  :
ld.global.f32     %f1, [%rd1+0];
mov.f32     %f2, 0f3f800000;        // 1
add.f32     %f3, %f1, %f2;
st.global.f32     [%rd1+0], %f3;
  :
```

図 6-7-5 (2)

```
  :
ld.global.f32     %f1, [%rd1+0];
mov.f32     %f2, 0f3f800000;        // 1
add.f32     %f3, %f1, %f2;
st.global.f32     [%rd1+0], %f3;
ld.global.f32     %f4, [%rd1+4];
mov.f32     %f5, 0f3f800000;        // 1
add.f32     %f6, %f4, %f5;
st.global.f32     [%rd1+4], %f6;
ld.global.f32     %f7, [%rd1+8];
mov.f32     %f8, 0f3f800000;        // 1
add.f32     %f9, %f7, %f8;
st.global.f32     [%rd1+8], %f9;
  :
```

図 6-7-5 (3)

何段のアンローリングがよいか

アンローリングで最も速度が速くなる段数は、当然ながらプログラムによって異なるので、試行錯誤で決定する必要があります。図 6-7-3 の簡単なプログラムでテストした限りでは、数段程度でほぼピークになり、後は段数を増やしてもあまり変わりせんでした。

手作業と自動の比較

アンローリングを手作業で行うか、コンパイラに自動的に行わせるかは好みによります。手作業で行う場合、当然ながら修正作業に手間がかかり、またプログラムが分かりにくくなります。

一方コンパイラに自動的に行わせる場合ですが、コンパイラと言えども、人間が作成したソフトウェアなので、バグを含んでいる可能性があります。一般に、コンパイラに複雑な処理（例えば最適化オプションを一番高くして最も高度な最適化を行わせるなど）を行わせると、コンパイラ自体のバグに遭遇する確率が高くなります。コンパイラにループアンローリングを自動的に行わせる場合も、同様の問題が発生する可能性があります。

第7章　数値計算ライブラリー

本章では、GPU 用に並列化された数値計算ライブラリーを紹介します。

7-1　CUBLAS

絶対値の合計

CUBLAS は、線形計算の数値計算ライブラリー BLAS (Basic Linear Algebra Subprograms) の CUDA 版です。配列の絶対値の合計を求める「cublasSasum」ルーチンの使用方法を、図 7-1-1 (1) で説明します（単精度用のルーチンで説明します)。CUBLAS の詳細は、「CUDA CUBLAS Library」（付録参照）を参照して下さい。

- CUBLAS のルーチン (図 7-1-1 (1) の「cublasxxxx」) を使用する場合、⓪を指定します。
- ②で、図 7-1-1 (2) に示すように、ホスト側の配列 X[5] に適当な値を設定します。
- ③で、CUBLAS ライブラリーを初期化します。③は、すべての CUBLAS ルーチンの一番最初に実行します。
- ④ (2 箇所) で、デバイス側の配列 dX[5] を確保します (後述するように CUDA 関数を使用しても構いません)。
- ⑤で、図 7-1-1 (2) に示すように、ホスト側の配列 X をデバイス側の配列 dX にコピーします。⑤の左の下線部は、配列 X の、コピーする要素間の間隔です。例えば図 7-1-1 (3) では「2」となります。同様に右の下線部は、コピー先の配列 dX 内の、コピーされた要素間の間隔です。
- ⑥は、BLAS のルーチン SASUM（単精度）の CUBLAS 版です。配列 dX 内の N 個の要素の絶対値の合計を求め、結果が①で宣言した変数 sumX に戻ります。⑥の右の下線部は、配列 dX 内の、計算に使用する要素間の間隔です。
- 計算が終了したら、⑦でデバイス側の配列 dX を解放し、⑧で CUBLAS ライブラリーが使用した CPU 側の資源を解放します。GPU 側の資源は、プログラムの終了時に解放されます。
- コンパイル／リンク時に図 7-1-2 の下線部を指定し、実行します。⑥は、配列 X の各要素の絶対値の合計を求めるルーチンなので、計算結果は⑩のようになります。

いくつか補足します。

- 本例では、ホスト側の配列 X をデバイス側の配列 dX にコピーして計算しましたが、配列 dX が既にデバイス側に存在しているときは、それをそのまま使用することができます。
- ④，⑤，⑦の代わりに、図 7-1-3 の4，5，7のように CUDA 関数を使用しても構いません。
- 図 7-1-1 (1) のように CUDA 関数を 1 つも使用しない場合、「gcc ～ test.c 〓」でもコンパイル可能です。

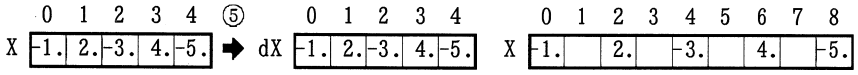


図 7-1-1 (2)　　図 7-1-1 (3)

```
#include "cublas.h"                                    ⓪
#define N (5)
int main(void){
  float X[N];
  float *dX;                                           ④
  float sumX;                                          ①
  配列Xに値を設定する。                                ②
  cublasInit();                                        ③
  cublasAlloc(N,sizeof(float),(void**)&dX);            ④
  cublasSetVector(N,sizeof(float),X,1,dX,1);           ⑤
  sumX = cublasSasum(N,dX,1);                          ⑥
  cublasFree(dX);                                      ⑦
  cublasShutdown();                                    ⑧
  printf("sumX = %f¥n",sumX);                          ⑨
    :
```

図 7-1-1 (1)

```
$ nvcc (最適化ｵﾌﾟｼｮﾝ) -lcublas test.cu ↵
sumX = 15.000000                                       ⑩
```

図 7-1-2

```
size_t size = N*sizeof(float);
cudaMalloc((void**)&dX,size);                          4
cudaMemcpy(dX,X,size,cudaMemcpyHostToDevice);          5
cudaFree(dX);                                          7
```

図 7-1-3

エラーチェック

図 7-1-1 (1) では説明を簡単にするため、CUBLAS ルーチン内で発生したエラーのチェック処理は省略しました。以下でエラーチェック方法について説明します。なお、エラーチェックを行わない場合、CUBLAS ルーチンでエラーが発生しても、メッセージは何も表示されず異常終了もしないので、必ずエラーチェックを行うようにして下さい。

図 7-1-1 (1) の③～⑧の CUBLAS ルーチンのうち、⑥は実際に計算を行うルーチンで、それ以外は CUBLAS の補助ルーチンです。例えば③の補助ルーチンのエラーチェック方法を説明します（他の補助ルーチンの場合も同じです）。図 7-1-4 (1) の⑯に示すように、補助ルーチンを実行すると戻り値が戻ります。これを、⑮で宣言した適当な名前の変数 (本例では status) に代入し、⑰で⑪の関数（関数名は任意）をコールします。関数内の⑫で戻り値をチェックし、正常値でなければ⑬で図 7-1-4 (2) のメッセージを表示し、⑭で強制終了します。図 7-1-4 (2) の「xx」には、⑯の戻り値が表示されます。

表示される値とその意味を図 7-1-4 (3) に示します。どの補助ルーチンでエラーが発生したのかを知るため、⑰の波線に⑯の補助ルーチン名を記述し、この補助ルーチン名が㉑の波線に表示されます。

一方、CUBLAS のルーチンのうち、⑱のように実際に計算を行うルーチンには、戻り値がありません。この場合、計算終了後、⑲で戻り値を取得し、⑳で⑰と同様に⑪の関数をコールします。

```
#include "cublas.h"
void CUBLAS_ERROR_CHECK(char *msg,                    ⑪
                        cublasStatus status){
  if (status != CUBLAS_STATUS_SUCCESS){               ⑫
    printf("CUBLAS error in %s.
            Error Code = %d¥n",msg,status);           ⑬
    exit(-1);                                         ⑭
  }
}
int main(void){
  cublasStatus status;                                ⑮
    :
  status = cublasInit();                              ⑯
  CUBLAS_ERROR_CHECK("cublasInit",status);            ⑰
    :
  sumX = cublasSasum(N,dX,1);                         ⑱
  status = cublasGetError();                          ⑲
  CUBLAS_ERROR_CHECK("cublasSasum",status);           ⑳
    :
```

図 7-1-4 (1)

```
CUBLAS error in cublasInit. Error Code = xx ㉑
```

図 7-1-4 (2)

```
 0：演算が正常終了しました。
 1：CUBLASライブラリーが初期化されていません。
 3：資源の割り当てに失敗しました。
 7：サポートされていない値が関数に渡されました。
 8：関数が、デバイスのアーキテクチャーに存在
    しないアーキテクチャーの機能を要求しました。
11：GPUのメモリ領域へのアクセスが失敗しました。
13：GPUプログラムの実行が失敗しました。
14：CUBLASの内部エラーです。
```

図 7-1-4 (3)

絶対値の最大（最小）値

配列 X 内の絶対値の最大値を求める場合は、図 7-1-1 (1) の①，⑥，⑨を、図 7-1-5 (1) のように変更します。⑥を実行すると、変数 ipos には、配列 X の最初の要素の位置を 1 としたときの、絶対値の最大値が入っている位置が戻ります。図 7-1-5 (2) の例では、絶対値の最大値は X[2] = -9.0 なので、ipos には「3」が戻ります。⑨を実行すると、図 7-1-5 (3) が表示されます。

最小値を求める CUBLAS ルーチンは cublasIsamin です。

```
  float maxX;                                         ①
  int ipos;                                           ①
  ipos = cublasIsamax(N,dX,1);                        ⑥
  maxX = X[ipos-1]);                                  ⑨
  printf("index=%d max=%f¥n",ipos-1,maxX);            ⑨
```

図 7-1-5 (1)

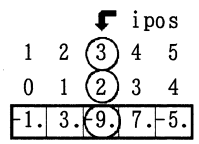


図 7-1-5 (2)

```
index=2 max=-9.000000
```

図 7-1-5 (3)

正負を考慮した合計／最大（最小）値

正負の値が混在した配列 X に対し、正負を考慮して合計／最大（最小）値を求める方法を説明します。

【方法 1】図 7-1-6 の (1) の配列 Y，Z をゼロクリアし、配列 X 内の正の要素を配列 Y に、負の要素を配列 Z に分けて入れ、(2) で cublasSasum を使用して絶対値の合計 sumY と sumZ を求め、(3) で「sumX = sumY-sumZ」とします。最大値を求める場合は、(2) で cublasIsamax を使用して配列 Y の絶対値の最大値を求めます。最小値を求める場合は、(2) で配列 Z の絶対値の最大値を求め、符号を負にします。

【方法 2】配列 X 内で、例えば「-10.0」より小さい値が存在しないのであれば、(4) で配列 X の全要素に「10.0」を加えて全ての値をゼロ以上にし、(5) で cublasSasum を使用して絶対値の合計 sumY を求め、最後に「sumX = sumY-(10.0* 要素数)」とします。最大（最小）値の場合は、(6) で「10.0」を引きます。

【方法 3】この方法は、合計を求める場合のみに適用可能です。(7) のように配列 Y に「1.0」を入れ、(8) で配列 X と Y の内積を求めると sumX となります。内積の計算は、(9) のように、CUBLAS ルーチン cublasSdot（単精度）を使用します。

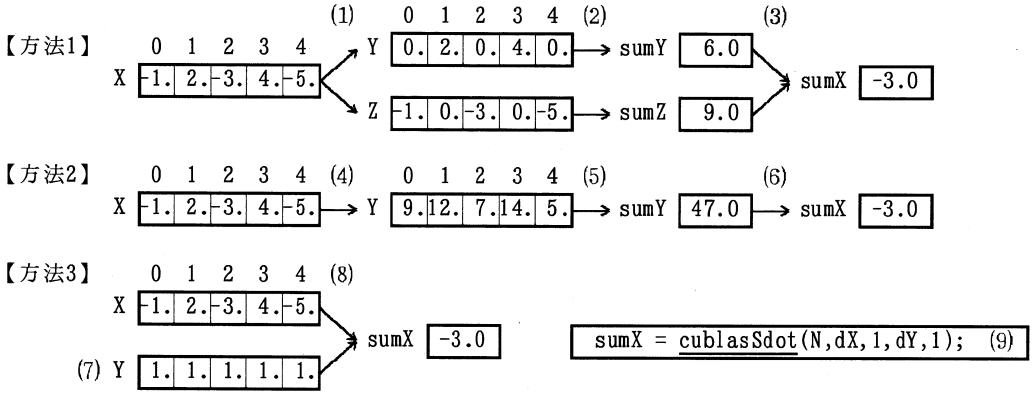


図 7-1-6

行列乗算

次に、BLAS の行列乗算ルーチン SGEMM（単精度）の CUBLAS 版ルーチン cublasSgemm の使い方を説明します。まず各行列と配列の形状を図で示します。図 7-1-8 (1) は行列乗算 $C = AB$ の行列 A，B，C（全角文字）を示します。また図 7-1-8 (2) はデバイス側の配列 dA，dB，dC、図 7-1-8 (3) はホスト側の配列 A，B，C（半角文字）を示します。通常、例えば A，dA，A の大きさは同じですが、一般的な使い方を説明するため、それぞれ異なる大きさだとします。

図 7-1-8 (3) (2) の矢印は、配列内の要素がメモリに並ぶ順番を示します。例えば配列 A と dA は、行列 A の転置になっていることに注意して下さい。また、例えば図 7-1-8 (3) の LDA (Leading Dimension of A) は、配列 A の 2 次元目（横方向）の大きさを示します。

以下で図 7-1-7 を説明します。

- ①でホスト側の配列 A，B，C を確保し、②と⑥でデバイス側の配列 dA，dB，dC を確保します。
- ③と④で、ホスト側とデバイス側の各配列の、2 次元目の大きさを設定します (図 7-1-8 (2) (3) 参照)。
- ⑤で、配列 A と B に値を設定します。
- 図 7-1-8 (3) の中央の図に示すように、ホスト側の配列 A からデバイス側の配列 dA にコピーしたい 8 個の要素は、メモリ上で不連続です。連続な場合は通常の cudaMalloc でコピーできますが、このように不連続な場合、⑦の CUBLAS ルーチン cublasSetMatrix でコピーすることができます (図 7-1-8 (2) (3) の⑦参照)。このとき、③，④で設定した配列 A と dA の 2 次元目の大きさ (LDA，LDdA) を、⑦の引数に指定します。
- 同様に⑧で、図 7-1-8 (2) (3) の⑧に示すように、ホスト側の配列 B をデバイス側の配列 dB にコピーします。
- ⑩の CUBLAS ルーチン cublasSgemm は $C = \beta C + \alpha AB$ を計算します。今回は $C = AB = 0 \cdot C + 1 \cdot AB$ を計算するので、⑨で $\alpha = 1.0$，$\beta = 0.0$ を設定し、α と β を⑩の引数に指定します。⑩の他の引数については、前述のマニュアルを参照して下さい。計算の結果、配列 dA と dB の行列積が配列 dC に入ります。
- ⑪で、図 7-1-8 (2) (3) の⑪に示すように、デバイス側の配列 dC をホスト側の配列 C にコピーします。

```
#include "cublas.h"
#define M (2)
#define N (3)
#define K (4)
int main(void){
  float A[6][4],B[5][6],C[5][4];                                  ①
  float *dA,*dB,*dC;                                              ②
  int LDA  = 4;int LDB  = 6:int LDC  = 4;                         ③
  int LDdA = 3;int LDdB = 5;int LDdC = 3;                         ④
  配列AとBに値を設定する。                                        ⑤
  cublasInit();
  cublasAlloc(5*3,sizeof(float),(void**)&dA);                     ↑
  cublasAlloc(4*5,sizeof(float),(void**)&dB);                     ⑥
  cublasAlloc(4*3,sizeof(float),(void**)&dC);                     ↓
  cublasSetMatrix(M,K,sizeof(float),A,LDA,dA,LDdA);               ⑦
  cublasSetMatrix(K,N,sizeof(float),B,LDB,dB,LDdB);               ⑧
  float alpha = 1.0f;float beta = 0.0f;                           ⑨
  cublasSgemm('N','N',M,N,K,alpha,dA,LDdA,dB,LDdB,beta,dC,LDdC);  ⑩
  cublasGetMatrix(M,N,sizeof(float),dC,LDdC,C,LDC);               ⑪
  cublasFree(dA);
  cublasFree(dB);
  cublasFree(dC);
  cublasShutdown();
    :
```

図 7-1-7

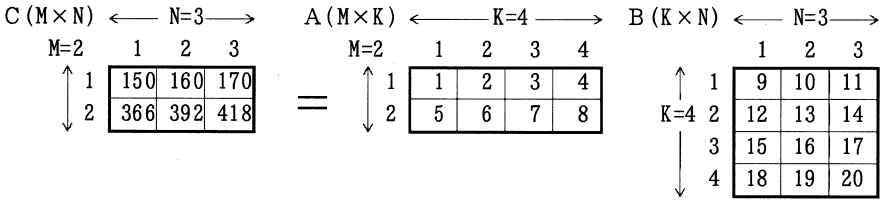


図 7-1-8 (1)　行列

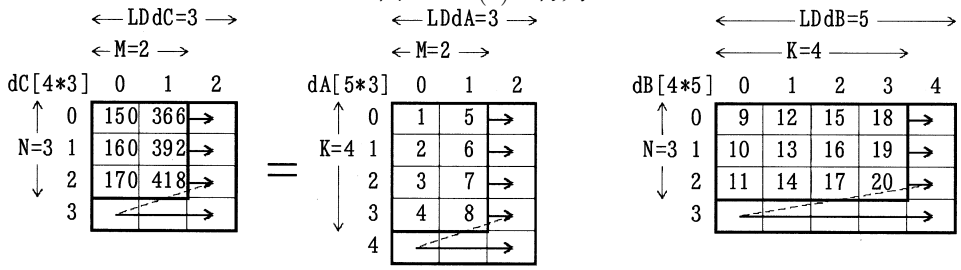


図 7-1-8 (2)　デバイス側の配列

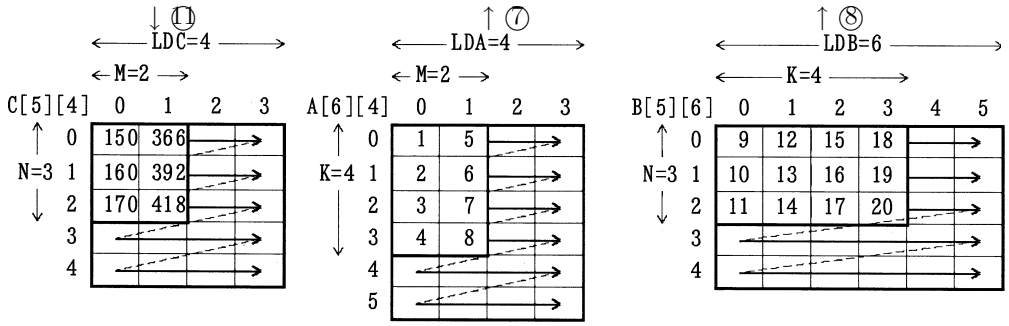


図 7-1-8 (3)　ホスト側の配列

7-2 CUFFT

CUFFT は、GPU 上で稼働する FFT（高速フーリエ変換）の数値計算ライブラリーで、1，2，3 次元の「複素数〓複素数」と「実数〓複素数」の変換を行うことができます。図 7-2-1 で、2 次元（単精度）の「複素数〓複素数」の変換を行う方法を説明します。CUFFT の詳細は、「CUDA CUFFT Library」（付録参照）を参照して下さい。

- CUFFT のルーチン (図 7-2-1 の「cufftxxxx」) を使用する場合、①を指定します。
- ②で、変換するデータの 1 次元目の要素数を N1、2 次元目の要素数を N2 とします。
- ⑤で、図 7-2-3 に示すように、ホスト側の配列 X と Z を宣言します。X[N1][N2] のように、1 次元目の要素数 N1 が左側の添字になることに注意して下さい。配列のデータ型は、CUFFT で提供されている cufftComplex 型（単精度複素数）を使用します。倍精度複素数の場合は cufftDoubleComplex 型を使用します。詳細は前述のマニュアルを参照して下さい。
- 変換前のデータの実数部を⑨で、虚数部を⑩で、配列 X に設定します。
- ⑧，⑪で、図 7-2-3 に示すように、デバイス側の配列 dX，dY，dZ を確保します。
- ⑫で、ホスト側の配列 X をデバイス側の配列 dX にコピーします。
- 下記を引数に指定して⑬を実行すると、これらの値が保管され、その保管先を示す値が、1 つ目の引数 plan (⑥で宣言し名前は任意) に戻ります。この変数をハンドルと呼びます。⑮，⑰，⑳の下線部でこのハンドルを使用します。
  - 2 つ目の引数：入力データの 1 次元目の要素数 N1 を指定します。
  - 3 つ目の引数：入力データの 2 次元目の要数数 N2 を指定します。
  - 4 つ目の引数：変換の種類を指定します。単精度の「複素数〓複素数」の変換の場合は、CUFFT_C2C となります。
- ⑭，⑯，⑱，㉑の説明は後述します。
- ⑮で、配列 dX のデータが、(CUFFT_FORWARD の指定により) 順変換され、配列 dY に入ります。配列 dX と dY は同じ配列でも構いません。
- ⑰で、配列 dY のデータが、(CUFFT_INVERSE の指定により) 逆変換され、配列 dZ に入ります。配列 dY と dZ は同じ配列でも構いません。なお、逆変換では規格化を行いません。従って配列 dZ の各要素を N1*N2 で割ると、変換前の配列 dX と同じ値になります。
- ⑲で、変換後の配列 dZ をホスト側の配列 Z にコピーします。
- ⑳で、ハンドル plan を無効にし、関連する GPU 資源を解放します。
- ⑬，⑮，⑰，⑳の戻り値を、⑦で宣言した変数 status（名前は任意）に入れ、⑭，⑯，⑱，㉑を実行すると、③（関数名は任意）が呼ばれ、戻り値をチェックし、エラーがある場合は④でルーチン名と下記の値を表示します。なお、CUFFT のルーチンによっては、下記の意味と若干異なる場合がありますので、下記のカッコ内と前述のマニュアルの各ルーチンの「Return Values」の説明を対応させて下さい。
  エラーチェックルーチンの指定は任意ですが、必ず指定するようにして下さい。
  - 1 (CUFFT_INVALID_PLAN): 無効なプランハンドル (本例では⑥の plan) が CUFFT に渡されました。
  - 2 (CUFFT_ALLOC_FAILED): CUFFT が、GPU メモリのアロケートに失敗しました。
  - 3 (CUFFT_INVALID_TYPE): ユーザーが、サポートされていないデータ型を要求しました。
  - 4 (CUFFT_INVALID_VALUE): ユーザーが、誤ったメモリポインターを要求しました。
  - 5 (CUFFT_INTERNAL_ERROR): 全ての内部ドライバーエラーに対して使用されます。
  - 6 (CUFFT_EXEC_FAILED): CUFFT が、GPU 上で FFT の実行に失敗しました。
  - 7 (CUFFT_SETUP_FAILED): CUFFT ライブラリーが、初期化に失敗しました。
  - 8 (CUFFT_INVALID_SIZE): ユーザーが、サポートされていない FFT の大きさを指定しました。
- コンパイル／リンク時に、図 7-2-2 の下線部を指定します。

```
#include <cufft.h>                                          ①
#define N1 (4)                                             ②
#define N2 (8)                                             ②
void CUFFT_ERROR_CHECK(char *msg,                          ③
                       cufftResult status){
  if (status != CUFFT_SUCCESS){
    printf("CUFFT error in %s.                             ④
            Error Code = %d¥n",msg,status);                ④
    exit(-1);
  }
}
main(){
  cufftComplex X[N1][N2],Z[N1][N2];                        ⑤
  cufftHandle plan;                                        ⑥
  cufftResult status;                                      ⑦
  cufftComplex *dX,*dY,*dZ;                                ⑧
  for(int i1=0;i1<N1;i1++){
    for(int i2=0;i2<N2;i2++){
      X[i1][i2].x = ～; (実数部)                           ⑨
      X[i1][i2].y = ～; (虚数部)                           ⑩
    }
  }
```

図 7-2-1

```
size_t size = N1*N2*sizeof(cufftComplex);
cudaMalloc((void**)&dX,size);
cudaMalloc((void**)&dY,size);                              ⑪
cudaMalloc((void**)&dZ,size);                              ⑫
cudaMemcpy(dX,X,size,cudaMemcpyHostToDevice);
status =                                                   ⑬
   cufftPlan2d(&plan,N1,N2,CUFFT_C2C);                     ⑬
CUFFT_ERROR_CHECK("cufftPlan2d",status);                   ⑭
status =                                                   ⑮
   cufftExecC2C(plan,dX,dY,CUFFT_FORWARD);                 ⑮
CUFFT_ERROR_CHECK("cufftExecC2C1",status);                 ⑯
status =                                                   ⑰
   cufftExecC2C(plan,dY,dZ,CUFFT_INVERSE);                 ⑰
CUFFT_ERROR_CHECK("cufftExecC2C2",status);                 ⑱
cudaMemcpy(Z,dZ,size,cudaMemcpyDeviceToHost);
status = cufftDestroy(plan);                               ⑲⑳
CUFFT_ERROR_CHECK("cufftDestroy",status);                  ㉑
  :
```

```
nvcc -lcufft (最適化ｵﾌﾟｼｮﾝ) test.cu
```

図 7-2-2

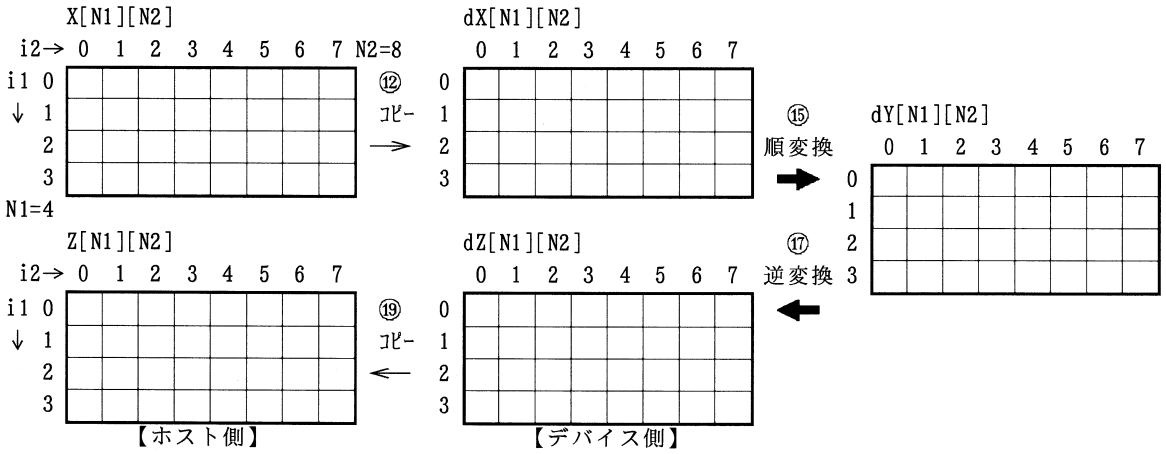


図 7-2-3

7-3　CUDPP

CUDPP (CUDA Data Parallel Primitives Library) は、CUDA SDK (1-4 節参照) に含まれている、GPU 上で稼働する数値計算ライブラリーで、合計、最大（小）値、ソート、乱数、疎行列の行列ベクトル積などのルーチンが提供されています。

CUDPP の Web サイト

CUDPP の Web サイトは `http://code.google.com/p/cudpp/` です。「CUDPP Documentation」をクリックすると、「CUDPP Documentation」の画面 (以下画面 1) が表示されます。

(1) 画面 1 内の「CUDPP Presentation」をクリックすると、CUDPP の概要を説明した資料が表示されます。
(2) 画面 1 の上の「Related Pages」をクリックし、「A Simple CUDPP Example」をクリックすると、CUDPP の簡単なプログラム例と解説が表示されます。
(3) 画面 1 の上の「Modules」をクリックし、「CUDPP Public Interface」をクリックして現れた画面の途中にある「Function Documentation」に、CUDPP の各ルーチン (`cudppScan` など) の文法の説明が表示されます。
(4) 画面 1 の上の「Files」をクリックし、「`cudpp.h`」をクリックして現れた画面の途中にある「Enumeration Type Documentation」に、CUDPP で指定する各パラメータ (`CUDPP_ADD` など) の説明が表示されます。なお、この中の一番下の「CUDPP Algorithm」内に下記が表示されており、合計、最大値などを求める専用のルーチンは、現時点 (2010 年 12 月現在) ではサポートされていないようです。
`CUDPP_REDUCE` parallel reduction (NOTE: currently unimplemented).

合計

配列の各要素の合計を、CUDPP の専用でないルーチン `cudppScan` を使用して求める方法を、図 7-3-1 で説明します。なお、図 7-3-1 の⑧〜⑫の詳細は上記の (4) を、⑭，⑯，⑱の詳細は上記の (3) を参照して下さい。

- CUDPP のルーチン (図 7-3-1 の「`cudppxxxx`」) を使用する場合、①を指定します。
- ⑤で、図 7-3-2 に示すように、ホスト側の配列 `X[5]` に適当な値を設定します。
- ⑥で、デバイス側の配列 `dX[5]`，`dY[5]` を確保します。
- ⑦で、図 7-3-2 に示すように、ホスト側の配列 `X` をデバイス側の配列 `dX` にコピーします。
- ⑧で、構造体 `config`（名前は任意）を宣言し、各メンバーに⑨〜⑫を設定します。
- ⑨で、使用するアルゴリズムはスキャンであることを指定します（スキャンの動作は後述します)。
- ⑩で、アルゴリズム内で行う演算が加算であることを指定します。
- ⑪で、データ型が単精度実数であることを指定します。
- ⑫で、図 7-3-2 の順序（後述します）でスキャンが行われることを指定します。
- 下記の引数を指定して⑭を実行すると、これらの指定が保管され、その保管先を示す値が、1 つ目の引数 `scanplan` (⑬で宣言、名前は任意) に戻ります。この変数をハンドルと呼びます。⑯，⑱の下線部でこのハンドルを指定します。
  - 2 つ目の引数：⑧で宣言した構造体 `config` を指定します。
  - 3 つ目の引数：処理できる最大の要素数を指定します。
  - 4 つ目の引数：入力データの行数 (本例では 1 次元なので 1) を指定します。
  - 5 つ目の引数：この引数は、上記 Web サイトの (3) では「入力データの行のピッチを要素数で指定する」となっていますが、意味がよく分かりませんでした。上記 Web サイトの (2) のプログラム例に「`0`」が指定されているので、ここでは「`0`」としました。
- ⑮，⑰，⑲の説明は後述します。
- ⑯を実行すると、配列 `dX` 内の `N` 個 (⑯の 4 つ目の引数で指定) の要素が、⑨〜⑫の設定に従って処理され、結果が配列 `dY` に入ります。本例では以下の (0)〜(4) の順にに計算が行われ、結果は図 7-3-2 のようになります。最終的に、`dY[4]` には `dX[0]`〜`dX[4]` の合計が入ります。

(0) `dY[0] = dX[0]`
(1) `dY[1] = dX[1] + dY[0]`　　(3) `dY[3] = dX[3] + dY[2]`
(2) `dX[2] = dX[2] + dY[1]`　　(4) `dY[4] = dX[4] + dY[3]`

- 計算が終了したら、⑱で、ハンドル scanplan を無効にし、関連する GPU 資源を解放します。
- ⑳で、合計の入った dY[4] をホスト側の変数 sumX にコピーし、㉑で表示します。
- ⑭, ⑯, ⑱の戻り値を、④で宣言した変数 status（名前は任意）に入れ、⑮, ⑰, ⑲を実行すると、②（関数名は任意）が呼ばれ、戻り値をチェックし、エラーがある場合は、③でエラーが発生したルーチン名と下記の値を表示します。エラーチェックルーチンの指定は任意ですが、必ず指定するようにして下さい。
  - 1: 指定されたハンドル (本例では⑬の scanplan) が無効です。
  - 2: 指定された構成 (本例では⑨〜⑫) が間違っています（例えば無効な組み合わせなど)。
  - 3: 不明または追跡不能なエラーです。
- CUDPP を使用する場合は、CUDA SDK (1-4 節参照) を導入して下さい。コンパイル／リンク時に、図 7-3-3 のようにリンクします (導入した CUDA SDK のディレクトリーが $HOME にある場合)。

```
#include "cudpp/cudpp.h"                                    ①
#define N (5)
void CUDPP_ERROR_CHECK(char *msg,                           ②
                       CUDPPResult status){
  if (status != CUDPP_SUCCESS){
    printf("CUDPP error in %s.                              ③
            Error Code = %d¥n",msg,status);                 ③
    exit(-1);
  }
}
int main(void){
  float X[N],Y[N];
  float *dX,*dY;
  float sumX;
  CUDPPResult status;                                       ④
  配列Xに値を設定する。                                     ⑤
  size_t size = N*sizeof(float);
  cudaMalloc((void**)&dX,size);                             ⑥
  cudaMalloc((void**)&dY,size);                             ⑥
  cudaMemcpy(dX,X,size,cudaMemcpyHostToDevice);
                                                            ⑦
```

```
CUDPPConfiguration config;                                  ⑧
config.algorithm = CUDPP_SCAN;                              ⑨
config.op = CUDPP_ADD;                                      ⑩
config.datatype = CUDPP_FLOAT;                              ⑪
config.options = CUDPP_OPTION_FORWARD                       ⑫
               | CUDPP_OPTION_INCLUSIVE;                    ⑫
CUDPPHandle scanplan = 0;                                   ⑬
status = cudppPlan(&scanplan,config,                        ⑭
                   N,1,0);                                  ⑭
CUDPP_ERROR_CHECK("cudppPlan",status);                      ⑮
status = cudppScan(scanplan,dY,dX,N);                       ⑯
CUDPP_ERROR_CHECK("cudppScan",status);                      ⑰
status = cudppDestroyPlan(scanplan);                        ⑱
CUDPP_ERROR_CHECK("cudppDestroyPlan",                       ⑲
                  status);                                  ⑲
cudaMemcpy(&sumX,&dY[N-1],sizeof(float),                    ⑳
           cudaMemcpyDeviceToHost);                         ⑳
printf("%f¥n",sumX);                                        ㉑
  :
```

図 7-3-1

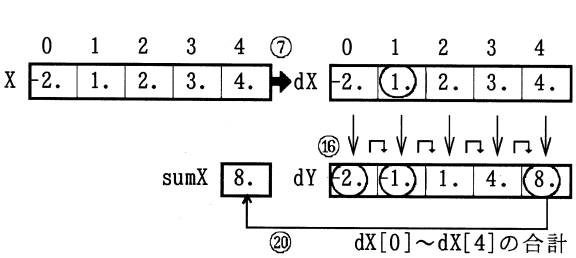


図 7-3-2

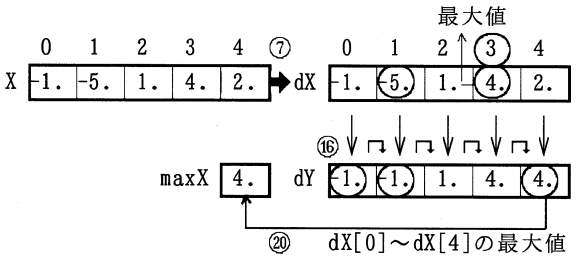


図 7-3-4

```
$ nvcc (最適化ｵﾌﾟｼｮﾝ) test.cu -lcudpp_x86_64
                -L$HOME/NVIDIA_GPU_Computing_SDK/C/common/lib/linux
                -I$HOME/NVIDIA_GPU_Computing_SDK/C/common/inc ↵
sumX = 8.000000
```

図 7-3-3

最大値

配列 X の各要素の最大値を求める場合は、図 7-3-1 の⑩を「config.op = CUDPP_MAX;」に変更します (最小値は CUDPP_MIN です)。図 7-3-4 の例では、最大値の dX[3] = 4.0 が、最終的に dY[4] に入ります。なお、最大値の入っている位置 (本例では dX[3] の「3」) も求めたい場合は、別途、配列 dX または dY の中を調べる必要があります。

7-4　その他のライブラリー

その他、GPU 上で稼働する数値計算ライブラリーを紹介します。他にも Web や文献を調べれば見つかると思います。

- 以下のライブラリーが、CUDA 3.2 から導入されました。使用方法はマニュアル（付録参照）を参照して下さい。
  CURAND: 擬似乱数生成用のライブラリー
  CUSPARSE: 疎行列計算用のライブラリー
- CULA tools (http://www.culatools.com/)
  LAPACK の GPU 版で、有償版と無償版があります。
  http://www.nvidia.co.jp/object/io_1250737175975.html に、日本語の簡単な紹介があります。
- MAGMA (Matrix Algebra for GPU and Multicore Architectures) (http://icl.cs.utk.edu/magma/)
  テネシー大学で開発した LAPACK の GPU 版です。
  http://www.nvidia.co.jp/object/io_1257669168818.html に、日本語の簡単な紹介があります。
- Thrust (http://code.google.com/p/thrust/)
  C++ の標準テンプレートライブラリー (STL) によく似た、高水準インターフェースを提供します。
  http://gpu.fixstars.com/index.php/ の「CUDA プログラミング TIPS」の「Thrust を使う」に、簡単な使い方の説明があります。
- Mersenne Twister (http://mwww.math.sci.hiroshima-u.ac.jp/~m-mat/MT/mt.html)
  (http://mwww.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index-jp.html)
  疑似乱数生成アルゴリズムのプログラムです。
- NVIDIA のサイト
  http://www.nvidia.com/object/tesla_software_jp.html（英語）の「Sample Codes and Libraries」および http://www.nvidia.co.jp/object/tesla_software_jp.html（日本語）の「サンプルコードおよびライブラリ」に、各種リンクがあります。
- Prometech MCL (http://www.gdep.jp/product/view/21)
  共役勾配法のライブラリーです。
- NTP (http://cvlab.jp/ で「NTP」をクリックして下さい。)
  行列計算 C++ ライブラリーです。
- MATLAB
  http://www.mathworks.co.jp/company/pressroom/articles/article52011.html
  http://www.mathworks.co.jp/products/new_products/latest_features.html
  http://www.nvidia.co.jp/object/IO（オー）_44225.html
  http://developer.nvidia.com/object/matlab_cuda.html
  http://www.nvidia.co.jp/object/matlab_cuda_jp.html
- Mathematica (http://wolfram.com/news/GPU.html)

第8章　プログラム例

本章では、CUDA 化したサンプルプログラムを紹介します。

8-1　多体問題

本節では、粒子の多体問題を簡単化したプログラムの CUDA 化について説明します。

粒子に働く力

図 8-1-1 (1) に示すように、1 次元の計算領域内に存在する粒子①，Ⓙの間に、同じ大きさで反対方向の引力が働き、粒子①が粒子Ⓙから受ける力は、「$f_{ij} = X_j - X_i$」(X_i，X_j は粒子①，Ⓙの座標) だとします ($f_{ij} = -f_{ji}$ となります)。計算領域内に粒子が⓪〜③の 4 個ある場合、粒子⓪に働く力の合力 F_0 は、図 8-1-1 (2) のようになります。

各粒子に働く力をまとめると図 8-1-1 (3) のようになります。例えば f_{00} はゼロなので、計算を行う必要はないですが、以下のプログラムでは、説明を簡単にするため計算を行っています。また、例えば f_{01} と f_{10} の値は同じ（符号が反対）なので一度計算するだけでよいですが、以下のプログラムでは、説明を簡単にするため二度計算しています。

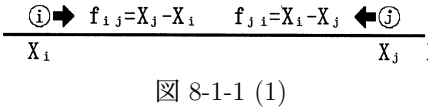


図 8-1-1 (1)

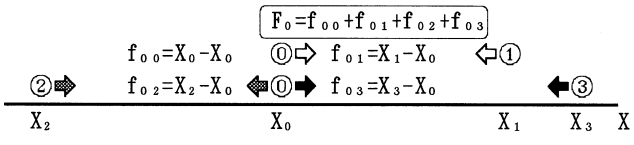


図 8-1-1 (2)

```
粒子⓪に働く力：F0 = f00 + f01 + f02 + f03 = (x0-x0) + (x1-x0) + (x2-x0) + (x3-x0)
粒子①に働く力：F1 = f10 + f11 + f12 + f13 = (x0-x1) + (x1-x1) + (x2-x1) + (x3-x1)
粒子②に働く力：F2 = f20 + f21 + f22 + f23 = (x0-x2) + (x1-x2) + (x2-x2) + (x3-x2)
粒子③に働く力：F3 = f30 + f31 + f32 + f33 = (x0-x3) + (x1-x3) + (x2-x3) + (x3-x3)
```

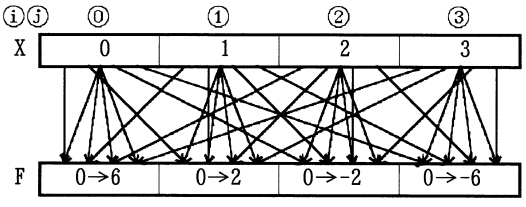
図 8-1-1 (3)

元のプログラム

粒子数が N = 4 の場合の、元のプログラムと動作を図 8-1-2 (1) (2) に示します。X[i] は粒子①の X 座標、F[i] は粒子①に働く力の合力の配列で、(1) で初期値を設定します。(3) は粒子①の力の計算を行うループ、(4) は相手の粒子Ⓙのループです。(3)，(4) で図 8-1-1 (3) の計算を行います。

```
#define N (4)
int main(void){
  int i,j;
  float X[N],F[N];
  for(i=0;i<N;i++){                          (1)
    X[i] = (float)i;
    F[i] = 0.0f;
  }
  for(i=0;i<N;i++){                          (3)
    for(j=0;j<N;j++){                        (4)
      F[i] = F[i] + (X[j]-X[i]);             (2)
    }
  }
  :
```

図 8-1-2 (1)



=(0-0) =(0-1) =(0-2) =(0-3)
+(1-0) +(1-1) +(1-2) +(1-3)
+(2-0) +(2-1) +(2-2) +(2-3)
+(3-0) +(3-1) +(3-2) +(3-3)

図 8-1-2 (2)

CUDA 化したプログラム (1)

図 8-1-2 (1) の (2) の部分を CUDA 化したプログラムと動作を、図 8-1-3 (1) (2) に示します。説明を簡単にするため、ブロック数は 2 個、ブロック当たりのスレッド数も 2 個とし、要素数 (4) がブロックあたりのスレッド数 (2) で割り切れない場合の処理は省略します (処理を行った例を 3-6 節の「■　割り切れない場合の処理」に示します)。図 8-1-3 (1) の配列 dX，dF は、図 8-1-2 (1) の配列 X，F に対応します。

(5) で、各スレッドは、自分が担当する粒子番号①を求め、(6) のループで、粒子①に働く、粒子⓪〜③からの力の合力を求めます。(6) のループ反復が N (= 4) 回なので、1 スレッドあたり、グローバルメモリ上の配列 dX と dF に対し、(8)，(9)，(10) のロードと (7) のストアを、それぞれ N (= 4) 回行ない、ロード／ストアの時間がかかります。

以下では、配列 dX と dF のロードとストアを減少させる方法を、2 段階に分けて説明します。

```
#define N (4)
__global__ void kernel(float *dX,float *dF){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  for(int j=0;j<N;j++){                                  (5)
    dF[i] = dF[i] + (dX[j]-dX[i]);                       (6)
  }   (7)       (8)        (9)    (10)
}
    :
  kernel<<<2,2>>>(dX,dF);
    :
```

図 8-1-3 (1)

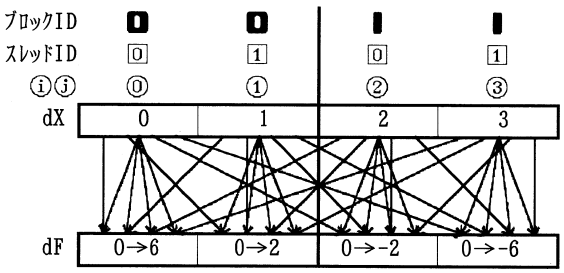


図 8-1-3 (2)

CUDA 化したプログラム (2)

図 8-1-3 (1) の (7)，(8)，(10) の配列は、添字が i なので、(6) のループ反復 (添字 j) とは関係ありません。従ってこれらの配列を、図 8-1-4 (1) の (11)，(12)，(14) のようにループの外に出し、変数 xi，fi（レジスター上に確保）に置き換えることができます。この場合の動作を図 8-1-4 (2) に示します。

(11)，(12)，(14) のロード／ストアは、それぞれ 1 回のみなので時間はかかりません。(13) のループ反復で、変数 xi，fi はレジスター上に存在するのでロード／ストアの時間がかからず、下線部に示す、グローバルメモリからの dX[j] のロードのみ時間がかかります。(13) のループ反復が N 回なので、1 スレッドあたり、dX[j] のロードを N (= 4) 回行ない、図 8-1-3 (1) (2) よりもロード／ストアの回数が減少して速度が向上します。このロードを図 8-1-4 (2) の太い矢印で示します。

このように、ループ反復とは関係のない配列をループの外に出す最適化は、通常のプログラムではコンパイラが行います。しかし CUDA では、アセンブラリスト (2-7 節参照) で確認したところ、この最適化を行っていませんでした (6-6 節参照)。

```
#define N (4)
__global__ void kernel(float *dX,float *dF){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float xi = dX[i];                                      (11)
  float fi = dF[i];                                      (12)
  for(int j=0;j<N;j++){                                  (13)
    fi = fi + (dX[j]-xi);
  }
  dF[i] = fi;                                            (14)
}
    :
  kernel<<<2,2>>>(dX,dF);
    :
```

図 8-1-4 (1)

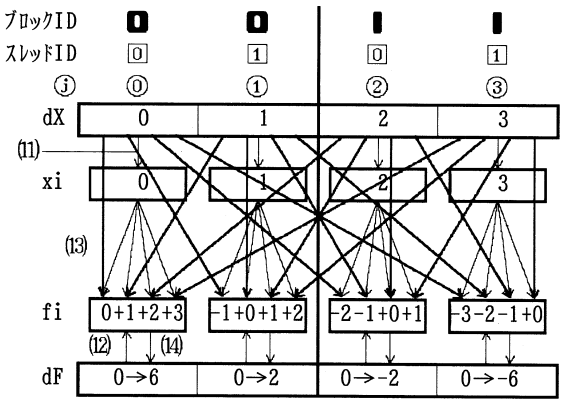


図 8-1-4 (2)

CUDA 化したプログラム (3)

図 8-1-4 (1) の (13) の下線部に示す、dX[j] のロード回数を、シェアードメモリを使用して減少させるプログラムを図 8-1-5 に示し、動作を図 8-1-6 (1) (2) に示します。

- (15) で、シェアードメモリ上に配列 dS[2] を確保します (配列 dS はブロックごとに確保され、ブロック内のスレッド数が 2 なので、配列の大きさは 2 となります)。なお、本例では要素数が N = 4 と少ないので、配列 dX[4] と同じ大きさの dS[4] を確保することもできますが、ここでは N が大きくて同じ大きさを確保できない場合を想定します。
- (18) のループはブロック数 (= 2) 回反復します。1 反復目 (j = 0) の動作を図 8-1-6 (1) に、2 反復目 (j = 2) の動作を図 8-1-6 (2) に示します。
- ループが 1 反復目のとき、(19) で、各ブロックのスレッド⓪は dX[0] を dS[0] に、スレッド①は dX[1] を dS[1] に、それぞれロードします。
- 全スレッドが (19) のロードを終了するのを保証するため、(20) で同期を取ります (詳細は 4-1 節参照)。
- (21) で、配列 dS を使用して計算を行います ((21) の jj は配列 dS の要素番号です)。
- あるスレッドが (21) を処理している間に、同じブロック内の他のスレッドがループの 2 反復目の (19) を実行して、配列 dS の値が変わってしまうのを防ぐため、(22) で同期を取ります (同期の詳細は 4-1 節参照)。
- ループが 2 反復目のとき、(19) で、各ブロックのスレッド⓪は dX[2] を dS[0] に、スレッド①は dX[3] を dS[1] に、それぞれロードします。以後の処理はループの 1 反復目と同様です。

(18) のループ反復ごとに、(19) で各スレッドは、グローバルメモリ上の dX[i] に対してロードを 1 回行います。(18) のループ反復が 2 回なので、1 スレッドあたり、(19) のロードを合計 2 回行ないます。このため、図 8-1-4 (1) (2) よりもロード回数が減少して速度が向上します。このロードを図 8-1-6 (1) (2) の太い矢印で示します。

```
#define N (4)
__global__ void kernel(float *dX,float *dF){
  __shared__ float dS[2];                          (15)
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float xi = dX[i];                                (16)
  float fi = dF[i];                                (17)
  for(int j=0;j<N;j+=blockDim.x){                  (18)
                                  ↰ 2
    dS[threadIdx.x] = dX[j+threadIdx.x];           (19)
    __syncthreads();                               (20)
    for(int jj=0;jj<blockDim.x;jj++){
      fi = fi + (dS[jj]-xi); ↰ 2                   (21)
    }
    __syncthreads();                               (22)
  }
  dF[i] = fi;                                      (23)
}
```

図 8-1-5

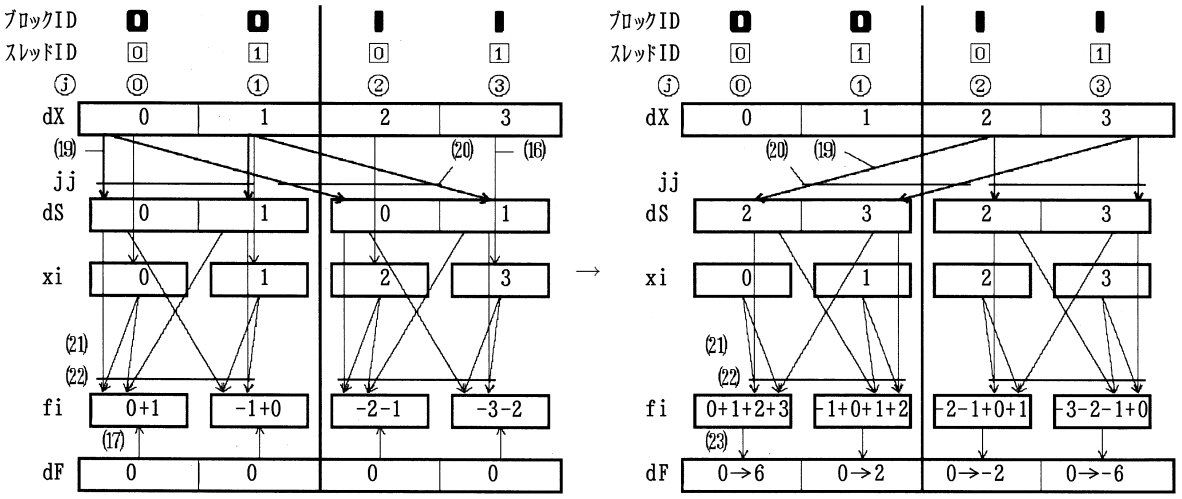


図 8-1-6 (1)　j = 0 のとき

図 8-1-6 (2)　j = 2 のとき

8-2　差分法

1 次元の場合

図 8-2-1 (1) は 1 次元の差分法を簡単化したプログラムです。②のタイムステップ・ループが反復するごとに、③で関数 func を呼び出します。関数 func では、①で配列 A を使用して配列 Anew を計算します (図 8-2-1 (2) 参照)。「境」(A[0] と A[10]) は固定境界を表し、計算は行いません。④で配列 A と Anew のポインターを入れ換え、②から再び同じ計算を行います。なお、ループ反復が終了した⑤では、配列 A が最終結果となります。

これを CUDA 化したプログラムを図 8-2-2 (1) に示します。⑨でブロック数を 3、ブロック内のスレッド数 (= BLOCK) を 4 とし、カーネル関数を実行します。配列 dA[0] は固定境界なので計算は行いませんが、ブロック 0 のスレッド[0]が、dA[0] でなく dA[1] を担当すると、コアレスアクセスの効率が悪くなるため (3-2 節参照)、ブロック 0 のスレッド[0]は dA[0] を担当します。⑦の if 文は、配列 dA の（計算すべき）要素を担当していないスレッド (例えばブロック 0 のスレッド[0]) が、⑧の計算を行わないようにするために指定します。なお、⑩の同期は念のために指定しています。

```
#define N (11)
void func(float *A,float *Anew){
  for(int i=1;i<N-1;i++){
    Anew[i] = (A[i-1]+A[i]+A[i+1])/3.0f;     ①
  }
}
int main(void){
  float *A,*Anew,*temp;
  size_t size = N*sizeof(float);
  A    = (float*)malloc(size);
  Anew = (float*)malloc(size);
  配列AとAnewに境界値と初期値を設定します。
  for(int istep=0;istep<10;istep++){          ②
    func(A,Anew);                             ③
    temp = Anew;
    Anew = A;                                 ④
       A = temp;
  }
  最終結果は配列Anewでなく配列Aに入ります。  ⑤
   :
```

図 8-2-1 (1)

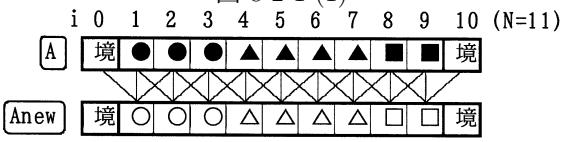


図 8-2-1 (2)

```
#define N (11)
#define BLOCK (4)
__global__ void kernel(float *dA,float *dAnew){
  int i = blockIdx.x*BLOCK + threadIdx.x;    ⑥
  if(1<=i && i<N-1){                         ⑦
    dAnew[i] = (dA[i-1]+dA[i]+dA[i+1])/3.0f;⑧
  }
}
int main(void){
  float A[N],Anew[N];
  float *dA,*dAnew,*temp;
  size_t size = N*sizeof(float);
  配列AとAnewに境界値と初期値を設定します。
  cudaMalloc((void**)&dA,size);
  cudaMalloc((void**)&dAnew,size);
  cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);
  cudaMemcpy(dAnew,Anew,
             size,cudaMemcpyHostToDevice);
  for(int istep=0;istep<10;istep++){
    kernel<<<3,BLOCK>>>(dA,dAnew);           ⑨
    cudaThreadSynchronize();                 ⑩
     temp = dAnew;
    dAnew = dA;
       dA = temp;
  }
  最終結果は配列dAnewでなく配列dAに入ります。
  cudaMemcpy(A,dA,size,cudaMemcpyDeviceToHost);
    :
```

図 8-2-2 (1)

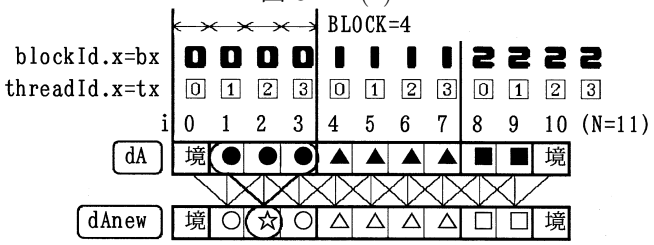


図 8-2-2 (2)

シェアードメモリの利用 (1 次元の場合)

図 8-2-2 (1) の⑧で、例えばブロック 0 のスレッド2は、グローバルメモリ上の配列 dA を 3 回ロードします (図 8-2-2 (2) の〓と㉘参照)。シェアードメモリを利用して、⑧のロードの回数を減らすプログラムを図 8-2-3 (1) に、データの動きを図 8-2-3 (2) に示します。

- ⑪でブロック数を 3、ブロック内のスレッド数 (= BLOCK) を 4 でカーネル関数を実行します。
- ①でシェアードメモリ上に配列 sA[6] を確保します。例えばブロック 1 内のスレッド0〜3は、dA[3]〜dA[8] の 6 個の要素をロードするため、配列 sA の大きさは 6 となります。
- ②と⑨の if 文は、配列 dA の（計算すべき）要素を担当していないスレッドが、計算を行わないようにするために指定します。
- ③で変数名 threadIdx.x を簡単化します。④の変数 sx は配列 sA の添字です (図 8-2-3 (2) 参照)。
- ⑤で各スレッドは、配列 dA の、自分が担当する要素を配列 sA にロードします。
- ⑥で、各ブロックの左端のスレッド (スレッド0、または i = 1 を担当するスレッド) は、配列 dA の、自分が担当する要素より 1 つ左の要素を配列 sA にロードします。同様に⑦で、各ブロックの右端のスレッド (スレッド3、または i = 9 を担当するスレッド) は、配列 dA の、自分が担当する要素より 1 つ右の要素を、配列 sA にロードします。
- ⑧で、ブロック内の全スレッドが配列 sA にロードするまで同期を取ります。なお、⑧は、計算を行わないスレッド (例えばブロック 0 のスレッド0) も実行する必要があります (4-1 節参照)。
- ⑩で、シェアードメモリ上の配列 sA を使用して計算を行います。シェアードメモリを用いることによって、グローバルメモリ上の配列 dA からのロード回数が、通常のスレッドでは⑤の 1 回に、各ブロックの左端（または右端）のスレッドでは、⑤と⑥ (または⑦) の 2 回に減少しました。

```
#define N (11)
#define BLOCK (4)
__global__ void kernel(float *dA,float *dAnew){
  __shared__ float sA[BLOCK+2];                    ①
  int sx,tx;
  int i = blockIdx.x*BLOCK + threadIdx.x;
  if(1<=i && i<N-1){                               ②
    tx = threadIdx.x;                              ③
    sx = tx+1;                                     ④
    sA[sx] = dA[i];                                ⑤
    if (tx==0 || i==1)                             ⑥
      sA[sx-1] = dA[i-1];                          ⑥
    if (tx==BLOCK-1 || i==N-2)                     ⑦
      sA[sx+1] = dA[i+1];                          ⑦
  }
  __syncthreads();                                 ⑧
  if(1<=i && i<N-1){                               ⑨
    dAnew[i] = (sA[sx-1]+sA[sx]+sA[sx+1])/3.0f;
  }                                                ⑩
}
int main(void){
    :
  kernel<<<3,BLOCK>>>(dA,dAnew);                   ⑪
    :
}
```

図 8-2-3 (1)

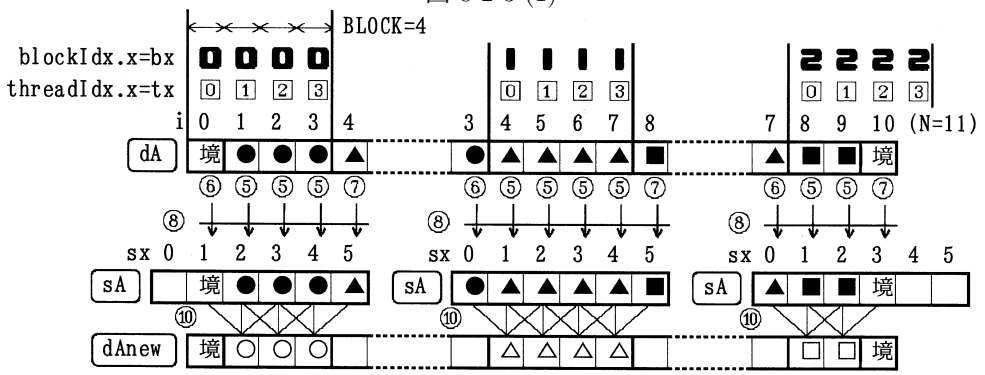


図 8-2-3 (2)

シェアードメモリの利用 (2 次元の場合)

図 8-2-5 (1) に示す 2 次元配列 dA[8][11] の場合について、図 8-2-4 のプログラムで説明します。図 8-2-4 は図 8-2-3 (1) (1 次元) を 2 次元にしただけで、ロジックはほとんど同じなので、説明は簡単に行ないます。なお、図 8-2-3 (1) と図 8-2-4 の①～⑪は対応しています。

- 図 8-2-4 の⑪で、ブロック数を x，y 方向に 3，3、ブロック内のスレッド数を x，y 方向に BLOCKx (= 4)，BLOCKy (= 3) としてカーネル関数を実行します。
- ①でシェアードメモリ sA[5][6] を確保します。
- ②と⑨の if 文は、配列 dA の（計算すべき）要素を担当していないスレッドが、計算を行わないようにするために指定します。
- ④の変数 sx，sy は配列 sA の添字です (図 8-2-5 (2) 参照)。
- ⑤で各スレッドは、配列 dA の、自分が担当する要素を配列 sA にロードします。なお本例では、⑬に示すように、配列 dA は実際には 1 次元配列 dA[11*8] ですが、⑫のマクロを使用して、dA[IND(iy,ix)] のように 2 次元配列風に表します (3-3 節参照)。
- ⑥，⑦，[6]，[7] では、各ブロックの左端、右端、上端、下端のスレッドは、配列 dA の、自分が担当する要素より 1 つ左、1 つ右、1 つ上、1 つ下の要素を、配列 sA にロードします。ブロック (1,1) の各スレッドのロードの様子を図 8-2-5 (1) (2) に示します。
- ⑧で、ブロック内の全スレッドが配列 sA にロードするまで同期を取ります。なお、⑧は、計算を行わないスレッドも実行する必要があります (4-1 節参照)。
- ⑩で、シェアードメモリ上の配列 sA を使用して計算を行います。例えば図 8-2-5 (2) の◇内の 5 つの要素を使用して、図 8-2-5 (3) の㊛を計算します。

```
#define BLOCKx (4)
#define BLOCKy (3)
#define NX (11)
#define NY (8)
#define IND(iy,ix) ((iy)*NX+(ix))                    ⑫
__global__ void kernel(float *dA,float *dAnew){
  __shared__ float sA[BLOCKy+2][BLOCKx+2];           ①
  int tx,ty,sx,sy;
  int ix = blockIdx.x*BLOCKx + threadIdx.x;
  int iy = blockIdx.y*BLOCKy + threadIdx.y;
  if(1<=ix && ix<NX-1 && 1<=iy && iy<NY-1){          ②
    tx = threadIdx.x; ty = threadIdx.y;              ③
    sx = tx+1; sy = ty+1;                            ④
    sA[sy][sx] = dA[IND(iy,ix)];                     ⑤
    if(tx==0 || ix==1)                               ⑥
      sA[sy][sx-1] = dA[IND(iy,ix-1)];               ⑥
    if(tx==BLOCKx-1 || ix==NX-2)                     ⑦
      sA[sy][sx+1] = dA[IND(iy,ix+1)];               ⑦
    if(ty==0 || iy==1)                               [6]
      sA[sy-1][sx] = dA[IND(iy-1,ix)];               [6]
    if(ty==BLOCKy-1 || iy==NY-2)                     [7]
      sA[sy+1][sx] = dA[IND(iy+1,ix)];               [7]
  }
  __syncthreads();                                   ⑧
  if(1<=ix && ix<NX-1 && 1<=iy && iy<NY-1){          ⑨
    dAnew[IND(iy,ix)] = (sA[sy-1][sx]
        + sA[sy][sx-1] + sA[sy  ][sx]
        + sA[sy][sx+1] + sA[sy+1][sx])/5.0f;         ⑩
  }
}
int main(void){
    :
  float *dA,*dAnew,*temp;                            ⑬
  size_t size = NX*NY*sizeof(float);                 ⑬
  cudaMalloc((void**)&dA,size);                      ⑬
  cudaMalloc((void**)&dAnew,size);                   ⑬
    :
  kernel<<<dim3(3,3),dim3(BLOCKx,BLOCKy)>>>
                                  (dA,dAnew);        ⑪
    :
}
```

図 8-2-4

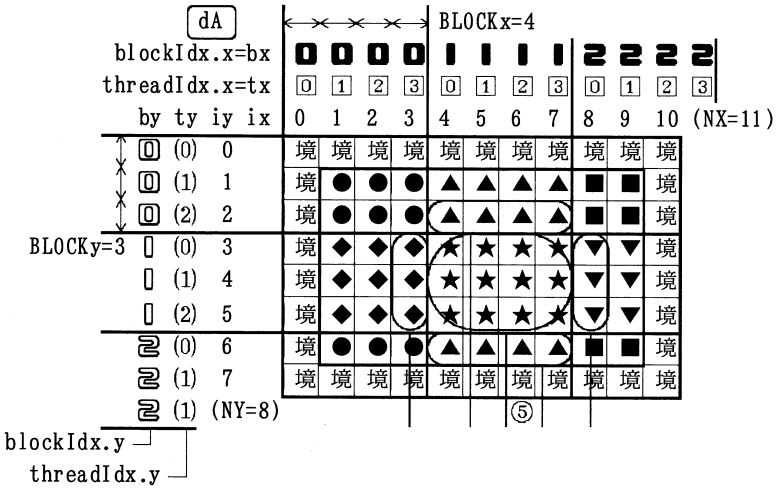


図 8-2-5 (1)

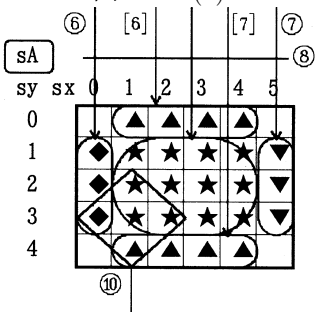


図 8-2-5 (2)

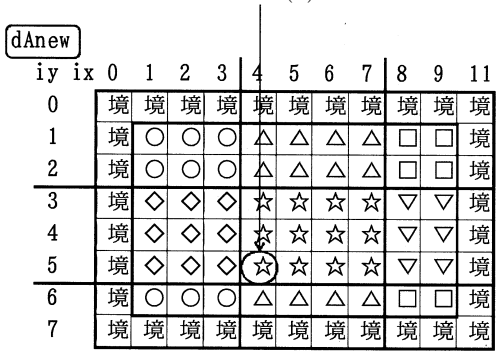


図 8-2-5 (3)

8-3　縮約演算

CUDA における縮約演算

例えば A[0]〜A[9] の合計を求めて変数 sum に代入する計算のように、配列の複数の要素 (A[0]〜A[9]) から、1 つの結果 (sum) を求める演算を、本書では縮約演算 (reduction operation) と呼びます。合計、内積、最大／最小などが代表的な縮約演算です。

MPI や OpenMP では、縮約演算を行うループの並列化は容易ですが、CUDA の場合、容易ではありません。最も手っとり早いのは、縮約演算を行う CUDA の数値計算ライブラリーの使用ですが、現在のところ、下記のように一長一短があります。

- CUBLAS (7-1 節参照) では、絶対値の合計、絶対値の最大／最小を求めるルーチンが提供されています。「絶対値の」合計や最大／最小なので、正負の値が混在している場合、そのままでは使用できません。
- CUDPP (7-3 節参照) では、合計、最大／最小を求めるルーチンが提供されています。ただし専用のルーチンではなく、合計、最大／最小以外の値（途中の値）も計算するので、合計、最大／最小だけを求めるよりも恐らく計算時間がかかると思われます。ただし、将来、合計、最大／最小だけを求めるルーチンも提供されるようです。

上記とは別に、1-4 節で説明した CUDA SDK 内のディレクトリー reduction に、合計を行う計算を CUDA 化したサンプルプログラムが入っています。その中の関数 reduce0〜reduce6 がカーネル関数の部分です。図 8-3-1 に示すように、関数 reduce0 が最も低速で、reduce6 が最も高速です。

またディレクトリー reduction/doc に、各関数の説明資料が入っています。サンプルプログラムの関数名と説明資料内の番号が、図 8-3-1 のようにずれているので注意して下さい。

関数 reduce0〜2 の問題点と解決方法は、他のプログラムを CUDA 化をする場合の参考になるので、本節では図のみ（プログラムなし）で説明します。また関数 reduce3 はプログラム例を説明します。関数 reduce4〜6 は reduce3 の改良版ですが、プログラムが複雑になるので説明は省略します。

関数	説明資料	プログラムの特徴
reduce0	Reduction #1	ワープ･ダイバージェントの問題があります。
reduce1	Reduction #2	reduce0の改良版で、バンクコンフリクトの問題があります。
reduce2	Reduction #3	reduce1の改良版で、無駄なスレッドの問題があります。
reduce3	Reduction #4	reduce2の改良版です。
reduce4	Reduction #5	reduce3に対し、ループの一部をアンローリングします。
reduce5	Reduction #6	reduce3に対し、ループ全体をアンローリングします。
reduce6	Reduction #7	reduce5のロジックの一部を改良しています。

図 8-3-1

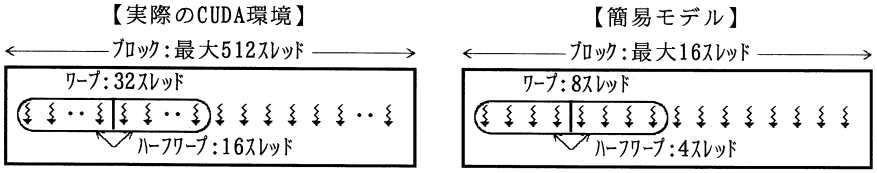
なお、付録の参考文献 [1]、および下記資料に、上記のサンプルプログラムの解説があります。

http://gpu-computing.gsic.titech.ac.jp/Japanese/Lecture/index.html

の、「第 6 回 GPU コンピューティング (CUDA) 講習会」の「CUDA プログラムの最適化」

関数 reduce0 (説明資料の Reduction #1)

実際の CUDA 環境で説明すると、紙面の制限のため図で表すことが難しいので、本節では以後、下記の簡易モデルで説明します。



関数 reduce0 の動作を図 8-3-2 (1) で説明します。説明を簡単にするため、加算する配列 A の要素数は 16 個、値はすべて 1 だとします。要素数が多い場合については、後述するプログラム例で説明します。

- 図 8-3-2 (1) の ↓ で、合計を求めるホスト側の配列 A を、デバイス側の配列 dA にコピーします。
- 要素数が 16 個、ブロックあたりの最大スレッド数が（簡易モデルでは）16 個なので、ブロック数を 1 個 (ブロック ID は 0)、ブロック内のスレッド数を 16 個 (スレッド ID は0〜15) として、カーネル関数を実行します。
- ⓪で、各スレッドは、グローバルメモリ上の配列 dA の自分が担当する要素を、シェアードメモリ上の配列 dS[16] にロードします。
- ①で、スレッド0, 2, ..., 14は、自分が担当する要素と 1 つ右隣の要素を加算します。①を以後ステップ①と呼びます。
- ステップ②で、スレッド0, 4, 8, 12は、自分が担当する要素と 2 つ右隣の要素を加算します。
- ステップ③で、スレッド0, 8は、自分が担当する要素と 4 つ右隣の要素を加算します。
- ステップ④で、スレッド0は、自分が担当する要素と 8 つ右隣の要素を加算します。
- ステップ④が終了すると、配列 dS の左端に、ブロック内の配列 dA の小計が求まります。ブロックが複数ある場合は、各ブロックの小計をさらに合計します（後述するプログラム例で説明します）。

関数 reduce0 の問題点

本節の簡易モデルでは、1 ワープは 8 スレッドなので、図 8-3-2 (1) には、〓に示すようにスレッド ID 0〜7と8〜15の 2 つのワープが含まれています。①〜④の各ステップは、図 8-3-3 に示すように、if 文による分岐になっています。各ステップで、加算を行うスレッドと行わないスレッドを、図 8-3-2 (2) の○と×で表します。ワープ内に○と×のスレッドが混在している場合、ワープ・ダイバージェント (6-5 節参照) が発生し、全スレッドが加算を行います（ただし×のスレッドは実際には加算しません)。また全スレッドが×なら加算を行いません。

その結果、下記の例では、図 8-3-2 (2) の太線の長方形で加算を行い、点線の長方形では加算を行いません。ワープ 1 のステップ④以外は全て加算を行なっており、計算時間がかかってしまいます。ワープ・ダイバージェントが多発する原因は、○のスレッドが、ブロック内で連続しておらず、とびとびになっているためです。

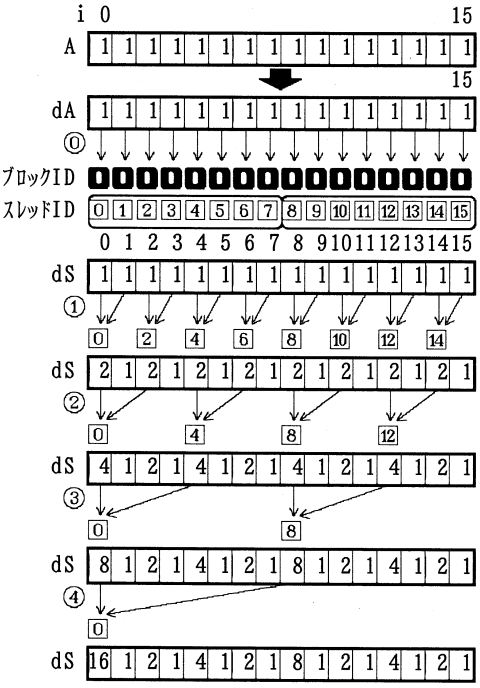


図 8-3-2 (1)

```
   :
if(ステップⓝを担当するスレッド){
   加算を行う。
}
   :
```

図 8-3-3

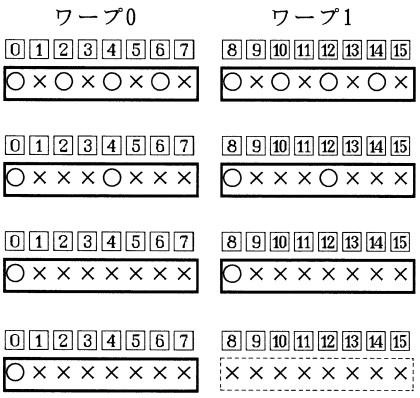


図 8-3-2 (2)

関数 reduce1 (説明資料の Reduction #2)

関数 reduce0 のワープ・ダイバージェントの問題を解決した関数 reduce1 の動作を、図 8-3-4 (1) に示します。図 8-3-2 (1) では、例えばステップ①の加算を担当するスレッド ID は、0, 2, 4, 6, ... と不連続でした。

一方図 8-3-4 (1) では、0, 1, 2, 3... と連続になります。その結果、図 8-3-4 (2) に示すように、加算を行う太線の長方形の数は、図 8-3-2 (2) よりも減少します。

関数 reduce1 の問題点

実際の環境では、3-6 節で説明したように、シェアードメモリは 16 個のバンクに分かれています。16 という値は、ハーフワープ内のスレッドの個数と同じです。本節では、ハーフワープが 4 スレッドの簡易モデルで説明しているので、シェアードメモリも 4 個のバンクに分かれているとします。図 8-3-4 (3) に、シェアードメモリ上の配列 dS[16] を示します。図中の例えば [0] は、dS[0] を表します。

以下の説明では、ブロック 0 のワープ 0 の最初のハーフワープ (図 8-3-4 (2) の〓で囲んだスレッド) の動作に着目します。図 8-3-4 (1) のステップ①の加算で、ハーフワープ内の各スレッドは、まず左側の要素 dS[0], dS[2], dS[4], dS[6] をロードし、次に右側の要素 dS[1], dS[3], dS[5], dS[7] をロードし、加算します。このうち左側の要素 (図 8-3-4 (1) の〓の要素) のロードでは、図 8-3-4 (3) の一番上の図内の ↑ に示すように、バンク 0 と 2 にアクセスが集中するため、2 ウェイ・バンクコンフリクトが発生し、速度が低下します (右側の要素のロードも同様に、2 ウェイ・バンクコンフリクトが発生します)。

同様に、ステップ②の左側の要素のロードでは、図 8-3-4 (3) の上から 2 つ目の図の ↑ に示すように、4 ウェイ・バンクコンフリクトが発生し、速度が低下します。

バンクコンフリクトが発生する原因は、ハーフワープ内の各スレッドが同時にロードする、左側または右側の要素が、配列 dS 上で連続しておらず、(ストライドが偶数で) とびとびになっているためです。

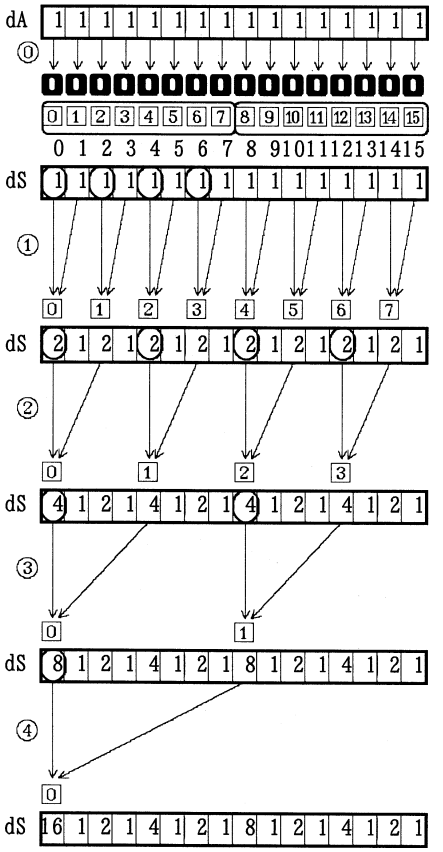


図 8-3-4 (1)

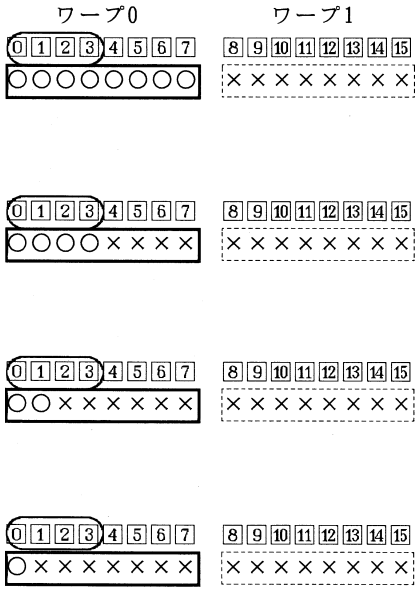


図 8-3-4 (2)

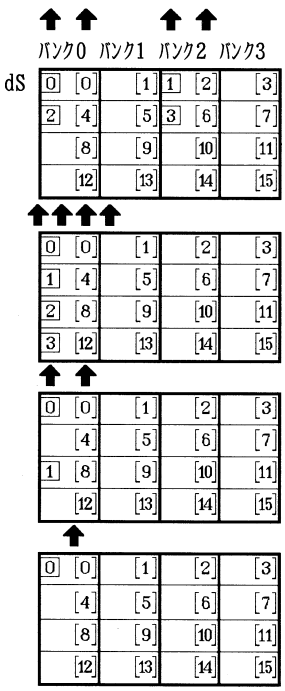


図 8-3-4 (3)

関数 reduce2 (説明資料の Reduction #3)

関数 reduce1 のバンクコンフリクトの問題を解決した関数 reduce2 の動作を、図 8-3-5 (1) に示します。

まず、ワープ・ダイバージェントですが、各ステップで加算を行うスレッド ID は reduce1 のときと同じなので、図 8-3-5 (2) も図 8-3-4 (2) と同じになり、問題はありません。

次にバンクコンフリクトですが、ブロック 0 のワープ 0 の最初のハーフワープ (図 8-3-5 (2) の〓で囲んだスレッド) 内の各スレッドが加算する左側の要素 (図 8-3-5 (1) の〓の要素) のロードでは、図 8-3-5 (3) の図内の ↑ に示すように、どのステップでもバンクコンフリクトは発生しません。これは、図 8-3-5 (1) で、各ステップの計算結果が常に左詰めに入るため、ハーフワープ内の各スレッドが同時にロードする、左側または右側の要素が、配列 dS 上で連続するからです。

関数 reduce2 の問題点

関数 reduce2 では、図 8-3-5 (1) から分かるように、ブロック内の0〜15のスレッドのうち、後半の8〜15のスレッドは、⓪のロードを行うだけで、一度も加算を行っていないため、せっかく割り当てられた CUDA コアが無駄になっています。そこで、8〜15のスレッドも、加算を 1 回行うように改良したのが、次に説明する関数 reduce3 です。

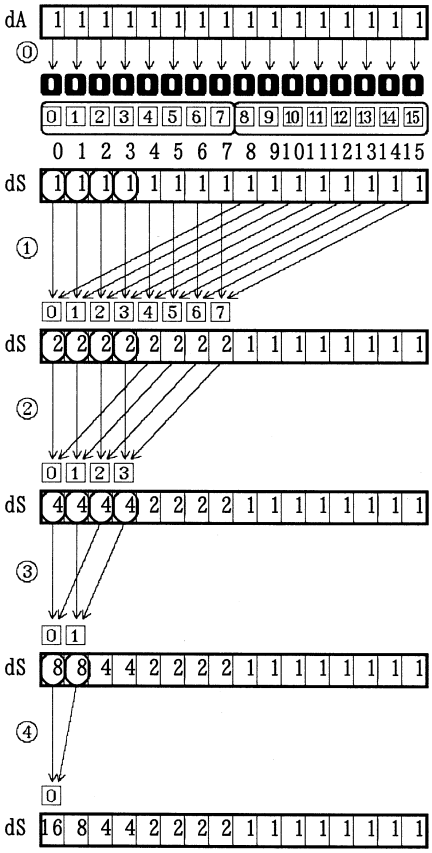


図 8-3-5 (1)

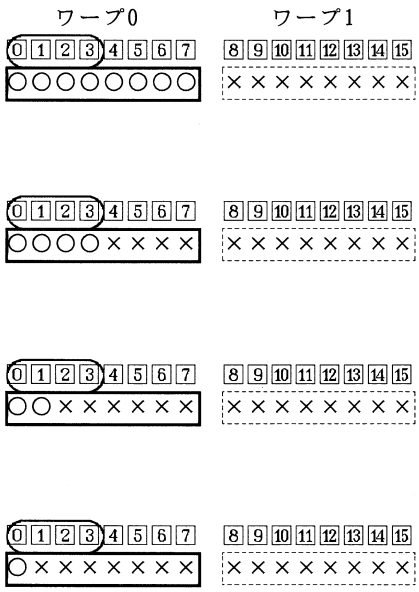


図 8-3-5 (2)

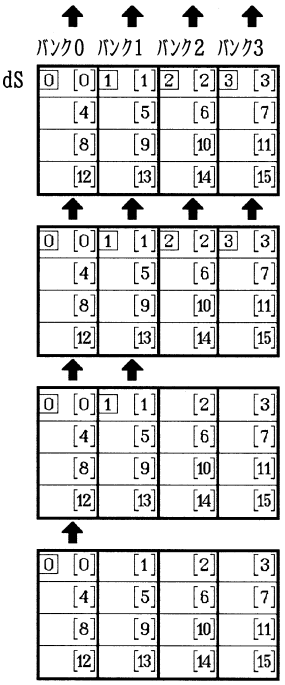


図 8-3-5 (3)

関数 reduce3 (説明資料の Reduction #4)
関数 reduce2 との比較

関数 reduce3 の動作を図 8-3-6 で説明します。図 8-3-6 は、図 8-3-7 のプログラム例の説明にも使うため、合計を求める配列 A が大きく (A[39]) なっています。プログラムの説明に入る前に、図 8-3-6 の左上のステップ⓪〜④を、図 8-3-5 (1) と比較します。

図 8-3-5 (1) では、ステップ⓪で、各スレッドは配列 dA の自分の担当する要素をロードするだけでした。図 8-3-6 では、ステップ⓪で、各スレッドは 2 つの要素をロードして加算を行います。従って、後半の8〜15のスレッドもステップ⓪で加算を 1 回行います。

ステップ①〜④の動作は図 8-3-5 (1) と全く同じなので、説明は省略します。

図 8-3-5 (1) と図 8-3-6 を比較すると（正確な比較ではありませんが)、図 8-3-5 (1) では、①〜④の 4 ステップで加算を 15 回 (1 ステップで平均 3.75 回) 行なっているのに対し、図 8-3-6 では、⓪〜④の 5 ステップで加算を 31 回 (1 ステップで平均 6.2 回) 行っており、CUDA コアの稼働率が高いため、効率が上がります。

プログラムの説明（概要）

関数 reduce3 のプログラム例を図 8-3-7 に示します。このプログラムは、CUDA SDK の reduce のサンプルプログラムを参考にしましたが、一部簡単化しており、また変数名が異なる部分もあります。まず、プログラムの全体の流れを図 8-3-6 で説明します。

- 配列 A[N] (N = 39) を加算するとします。また、ブロックあたりのスレッド数を 16 とします。
- ホスト側のプログラムで、配列 A をデバイス側の配列 dA にコピーします。
- 本例では、カーネル関数 reduce3 を 2 回実行します。まず 1 回目の実行を、ブロック数を 2 個 (ブロック ID = 0, 1) で行います。1 回目の動作を図 8-3-6 の上半分に示します。
- 図 8-3-6（上半分）の⓪で、ブロック 0 と 1 の各スレッドは、（本例では）以下のように動作します。これは、加算する要素が (2 個でなく) 1 個または 0 個のスレッドが、2 個の要素を加算するのを防ぐのが目的です。
  - ブロック 0 の全スレッドは、配列 dA の 2 個の要素をロード／加算し、シェアードメモリ上の配列 dS に代入します。
  - ブロック 1 のスレッド0〜6は、配列 dA の 1 個の要素をロードし、シェアードメモリ上の配列 dS に代入します。
  - ブロック 1 のスレッド7〜15は、配列 dS のゼロクリアのみを行います。
- 各スレッドはステップ①〜④で加算を行います。その結果、ブロック 0 ではブロック内の小計「32」が dS[0] に入り、ブロック 1 ではブロック内の小計「7」が dS[0] に入ります。
- ブロック 0, 1 のスレッド0は、⑤で、自分のブロックの小計 dS[0] を、グローバルメモリ上の配列 dB[0], dB[1] にストアします。
- これで関数 reduce3 の 1 回目の実行が終了し、いったんホスト側のプログラムに戻ります。
- ホスト側のプログラムでは、配列 dA と dB のポインターを入れ替え、配列 dB を dA に、配列 dA を dB にします。
- カーネル関数 reduce3 の 2 回目の実行を、ブロック数を 1 個 (ブロック ID = 0) で行います。2 回目の動作を図 8-3-6 の下半分に示します。
- 図 8-3-6（下半分）の⓪で、ブロック 0 の各スレッドは、以下の処理を行います。
  - ブロック 0 のスレッド0, 1は、配列 dA の 1 個の要素をロードし、シェアードメモリ上の配列 dS に代入します。
  - ブロック 0 のスレッド2〜15は、配列 dS のゼロクリアのみを行います。
- ブロック 0 の各スレッドは、ステップ①〜④で加算を行い、配列 A の総合計「39」が dS[0] に入ります。
- ブロック 0 のスレッド0は、⑤で、配列 A の総合計が入った dS[0] を、グローバルメモリ上の配列 dB[0] にストアします。
- これで関数 reduce3 の 2 回目（最後）の実行が終了し、ホスト側のプログラムに戻ります。
- ホスト側のプログラムでは、⑥で、配列 A の総合計が入った dB[0] を、ホスト側の変数 sum にコピーします。

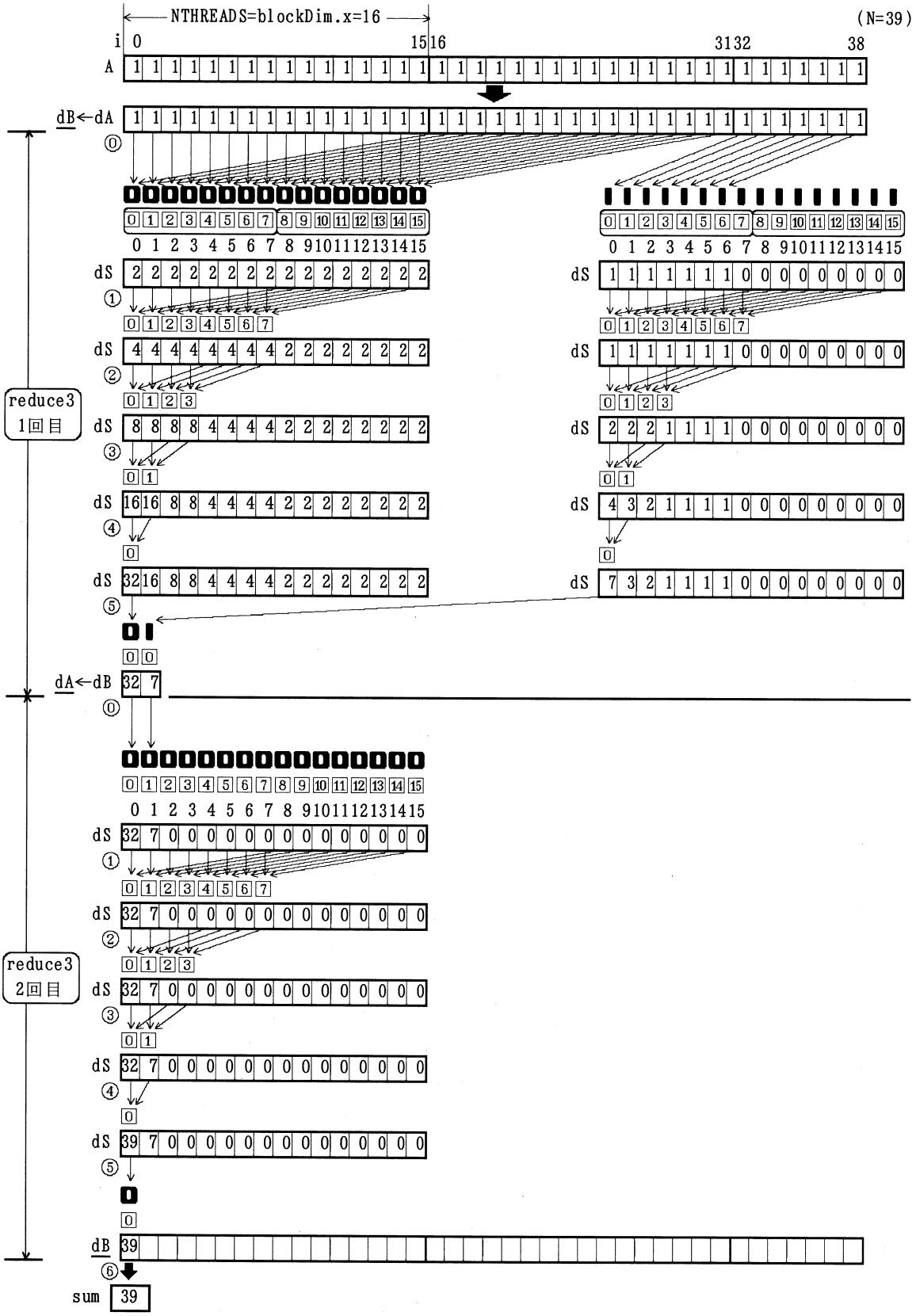

NTHREADS=blockDim.x=16
(N=39)
i
A
dB←dA
dS
reduce3
1回目
dA←dB
reduce3
2回目
dB
sum
39

図 8-3-6

プログラムの説明（詳細）

図 8-3-6 の動作を行う、図 8-3-7 のプログラムを説明します。

- [1] で、配列 A の、合計を求める要素数を N (= 39) とします。
- [2] で、ブロックあたりのスレッド数 NTHREADS を設定します。図 8-3-6 の簡易モデルに合わせて 16 に設定していますが、実際の環境では、2 のベキ乗で、32 の倍数で 512 までの値 (例えば 256 や 512) に設定して下さい。
- ホスト側プログラムの [15] で、念のため、要素数がゼロ以下のときはエラーにしています。
- [16] で、配列 A に値を設定します。
- [17] で、デバイス側の配列 dA を確保し、配列 A を dA にコピーします。
- [18] の変数 n には、[22] でコールするカーネル関数 reduce3 が合計する要素数を設定します。本例では、図 8-3-6 に示すように、カーネル関数 reduce3 の 1 回目のコールでは [18] で n = 39 (= N) 個を、2 回目のコールでは [25] で n = 2 (= NBLOCKS) 個を設定します。
- [19] の変数 NBLOCKS には、[22] でコールするカーネル関数 reduce3 のブロック数を指定します。1 スレッドあたり 2 個の要素を (ステップ⓪で) 合計するので、1 ブロックあたり 16 (NTHREADS) × 2 = 32 個までの要素を合計することができます。したがって、合計する要素数 n が例えば 1〜32 の範囲ならブロック数 NBLOCKS は 1 個、33〜64 の範囲ならブロック数 NBLOCKS は 2 個となります (本例では BLOCKS は 2 個です)。これを [19] で計算します。
- [20] で、要素数がブロック数 (本例では 2 個) である配列 dB を確保します。これは、図 8-3-6 の上半分の最後の⑤で使用します。
- [21] の無限ループが反復するごとに、[22] で [3] のカーネル関数 reduce3 をコールします。本例では、図 8-3-6 に示すように、カーネル関数 reduce3 は 2 回コールされます。まずカーネル関数 reduce3 の 1 回目の動作を説明します。
- [4] で、シェアードメモリ上に、大きさ NTHREADS の配列 dS を確保します。
- [5] で、各スレッドが図 8-3-6 の⓪で加算する 2 つの要素のうち、左側の要素の添字 i を設定します。例えばブロック 0 のスレッド0ではi = 0，ブロック 1 のスレッド0ではi = 32 となります。
- 図 8-3-6（上半分）の⓪で、各スレッドは（基本的に）2 つの要素をロードし、加算し、配列 dS にストアします。これを [6]〜[8] で行います。[6] で左側の要素をロードし、[7] で右側の要素を加算し、[8] で配列 dS にストアします。このとき、ブロック 1 のスレッド7〜15が左側の要素をロードしないように、[6] の if 文が指定されています。また、ブロック 1 の全スレッドが右側の要素をロードしないように、[7] の if 文が指定されています。
- [9] で、同一ブロック内の全スレッドが配列 dS に値をストアするまで、同期を取ります。
- [10] の「s>>=1」は、「変数 s (本例では s は 2 のベキ乗) の値を 1 ビット右にシフトして s に代入する」という意味です。例えば 2 進数の 00001000 (10 進数の 8) を 1 ビット右にシフトすると 00000100 (10 進数の 4) になります。従って「s>>=1」は「s の値を 1/2 にする」という意味になります。本例では、[10] で s は 8 (= 16/2) → 4 → 2 → 1 と変化します。
- [10] のループ反復で s が 8，4，2，1 のとき、[11]，[12] で図 8-3-6（上半分）のステップ①，②，③，④の加算を行います。s が 8，4，2，1 のとき、[11] の if 文が真になって加算を行うスレッド ID を以下の○に示し、[12] で行う計算を以下の右式に示します。変数 s は、加算する 2 つの要素間の距離（単位は要素数）を表します。

ステップ	s	threadIdx.x 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	[12]の式　　s ↓
①	8	○○○○○○○○×××××××××	dS[threadIDx.x] = dS[threadIdx.x] + dS[threadIdx.x+8]
②	4	○○○○×××××××××××××	dS[threadIDx.x] = dS[threadIdx.x] + dS[threadIdx.x+4]
③	2	○○×××××××××××××××	dS[threadIDx.x] = dS[threadIdx.x] + dS[threadIdx.x+2]
④	1	○××××××××××××××××	dS[threadIDx.x] = dS[threadIdx.x] + dS[threadIdx.x+1]

- [10] の各ループ反復の最後に [13] で、同一ブロック内の全スレッドが [12] の加算を終了するまで同期を取ります ([11] の if 文が偽のスレッドは、実際には加算を行いません)。
- [10] のループ反復が終了すると、図 8-3-6 の上半分に示すように、各ブロックの dS[0] に、そのブロックでの小計が入ります。[14] で、ブロック 0，1 のスレッド0は、図 8-3-6（上半分）の⑤に示すように、小計を [20] で確保した配列 dB[0]，dB[1] にストアし、ホスト側のプログラムに戻ります。

- [23] の同期は念のために指定しています。
- [24] の時点で、変数 NBLOCKS には、[19] で設定した「2」が設定されています。これは、1 回目の反復でのブロックの数、すなわち図 8-3-6（上半分）の⑤で設定した、配列 dS 内の小計の数を表します。この値が「1」より大きい場合は、まだ総合計が求まっていないので、[21] の無限ループから抜けず、[25] に進みます。
- カーネル関数 reduce3 の 2 回目の実行では、配列 dB 内の NTHREADS 個 (本例では 2 個) の小計を合計します。まず [25] で、合計する要素数 n を NBLOCKS (本例では 2) 個とし、[26] ([19] と同じ) で、2 回目の実行でのブロック数 NBLOCKS を決定します。合計する要素数 n が例えば 1〜32 の範囲ならブロック数 NBLOCKS は 1 個、33〜64 の範囲ならブロック数 NBLOCKS は 2 個となります (本例では NBLOCKS は 1 個です)。
- [27] で配列 dA と dB のポインターを入れ替えて、配列 dB を dA に、配列 dA を dB にします (図 8-3-6 参照)。
- [21] の無限ループが 2 反復目になり、[22] で [3] のカーネル関数 reduce3 を再び実行します。
- カーネル関数 reduce3 の 2 回目の動作は 1 回目と同じなので、説明は省略します (図 8-3-6 の下半分参照)。
- [14] で、ブロック 0 のスレッド⓪は、図 8-3-6（下半分）の⑤に示すように、配列 A の総合計の 39 を配列 dB[0] にストアし、ホスト側のプログラムに戻ります。
- [24] の時点で、変数 NBLOCKS には、[26] で設定した「1」が設定されています。これは、2 回目の反復でのブロックの数、すなわち図 8-3-6（下半分）の⑤で設定した、配列 dS 内の小計の数を表します。この値が「1」のとき、総合計が求まったので、加算を終了し、[21] の無限ループから抜けます。
- [28] で、図 8-3-6 の⑥に示すように、配列 A の総合計 dB[0] をホスト側の変数 sum にコピーします。
- なお、計算終了後に再びこの計算を行う場合、[17] と [20] で確保した配列 dA, dB が、[27] のポインターの入れ替えによって逆になっている場合がありますので注意して下さい。

```
#define N (39)                          [1]
#define NTHREADS (16)                   [2]   実際は256や512などを設定します。

__global__                                                          [3]
  void reduce3(float *dA,float *dB,int n){                          [3]
  __shared__ float dS[NTHREADS];                                    [4]
  int i = blockIdx.x*(blockDim.x*2)                                 [5]
        + threadIdx.x;     ← NTHREADSと同じ                         [5]
  float temp;
  if(i<n){                                                          [6]
    temp = dA[i];
  }else{
    temp = 0.0f;
  }
  if(i+blockDim.x<n){                                               [7]
    temp = temp + dA[i+blockDim.x];
  }
  dS[threadIdx.x] = temp;                                           [8]
  __syncthreads();                                                  [9]
  for(int s=blockDim.x/2;s>0;s>>=1){                                [10]
                  ← NTHREADSと同じ
    if(threadIdx.x<s){                                              [11]
      dS[threadIdx.x] = dS[threadIdx.x]                             [12]
                      + dS[threadIdx.x+s];                          [12]
    }
    __syncthreads();                                                [13]
  }
  if(threadIdx.x==0) dB[blockIdx.x] = dS[0];                        [14]
}
```

```
int main(void){
  int n,NBLOCKS;
  float A[N],*dA,*dB,*dTEMP,sum;
  if(N<=0){                                                         [15]
    printf("ERROR¥n");
    return(-1);
  }
  配列Aに値を設定します。                                            [16]
  size_t size = N*sizeof(float);                                    [17]
  cudaMalloc((void**)&dA,size);
  cudaMemcpy(dA,A,size,
             cudaMemcpyHostToDevice);
  n = N;                                                            [18]
  NBLOCKS = (n+(NTHREADS*2)-1)/(NTHREADS*2);                        [19]
  size = NBLOCKS*sizeof(float);                                     [20]
  cudaMalloc((void**)&dB,size);                                     [20]
  while(1){                                                         [21]
    reduce3<<<NBLOCKS,NTHREADS>>>(dA,dB,n);                         [22]
    cudaThreadSynchronize();                                        [23]
    if(NBLOCKS==1) break;                                           [24]
    n = NBLOCKS;                                                    [25]
    NBLOCKS
      = (n+(NTHREADS*2)-1)/(NTHREADS*2);                            [26]
    dTEMP = dA;                                                     [27]
      dA = dB;
      dB = dTEMP;
  }
  cudaMemcpy(&sum,dB,sizeof(float),                                 [28]
             cudaMemcpyDeviceToHost);                               [28]
  printf("sum = %f¥n",sum);
    :
```

図 8-3-7

8-4　行列乗算

行列乗算 $C = AB$ は、CUBLAS (7-1 節参照) で提供されているので、実用的には CUBLAS のルーチンを使用するのが簡単です。本節では、シェアードメモリを使用するプログラムのサンプルとして、参考までに、行列乗算を CUDA 化するプログラムを説明します。

元の行列乗算のプログラムを図 8-4-1 に示します。行列 A，B，C は 4×4 の正方行列とします。

シェアードメモリを使用しないプログラム

図 8-4-1 を、まず、シェアードメモリを使用せずに CUDA 化したプログラムを図 8-4-2 に示します。

- [7] と図 8-4-3 に示すように、ブロック数を 2×2、ブロック内のスレッド数を 2×2 とします。
  なお、図中では、blockIdx.x を bx、blockIdx.y を by、threadIdx.x を tx、threadIdx.y を ty と略記します。
- [1] で、ブロック内の各辺のスレッド数の 2 を、定数 BLOCK で表します。なお、行列の行数 (= 列数) の N (= 4) は BLOCK (= 2) で割り切れるとし、割り切れない場合の処理は省略します。
- [2]，[3] で、各スレッドが担当する dC[i][j] の i と j の値を決定します (図 8-4-4 参照)。
- [4] で変数 sum をゼロクリアします。
- [5] で各スレッドは、行列 dA と dB の自分が担当する要素の乗算を行います。
- [6] で各スレッドは、計算結果の sum を、自分が担当する dC[i][j] に代入します。
- 例えば (bx, by) = (1, 0) のブロックの、(tx, ty) = ([0], (1)) のスレッドは、図 8-4-4 の配列 dA の〓と配列 dB の〓の内積を計算し、結果が配列 dC の○に入ります。

[5] で、1 スレッドあたり、グローバルメモリ上の配列 dA と dB の各要素のロードを、それぞれ N (= 4) 回行うので時間がかかります。このロードの回数を、次ページで説明するように、シェアードメモリ (3-6 節参照) を使用して減らします。

```
#define N (4)
    :
float A[N][N],B[N][N],C[N][N];
float sum;
  :
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    sum = 0.0f;
    for(k=0;k<N;k++){
      sum = sum + A[i][k]*B[k][j];
    }
    C[i][j] = sum;
  }
}
  :
```

図 8-4-1

```
#define N (4)
#define BLOCK (2)                              [1]
  __device__ float dA[N][N],dB[N][N],dC[N][N];
__global__ void kernel(){
  int i = blockIdx.y*BLOCK + threadIdx.y;      [2]
  int j = blockIdx.x*BLOCK + threadIdx.x;      [3]
  float sum = 0.0f;                            [4]
  for(int k=0;k<N;k++){
    sum = sum + dA[i][k]*dB[k][j];             [5]
  }
  dC[i][j] = sum;                              [6]
}
int main(void){
    :
  kernel<<<dim3(2,2),dim3(2,2)>>>();           [7]
    :
```

図 8-4-2

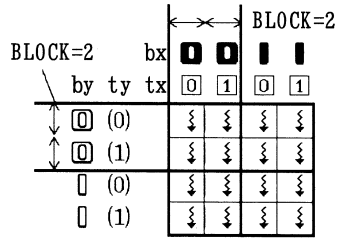


図 8-4-3

図 8-4-4

シェアードメモリを使用するプログラム（式の展開）

図 8-4-2 に対してシェアードメモリを使用する方法を説明します。図 8-4-5 (1) の行列 dA と dB 内の、太線で囲んだ 2 × 2 の小行列〓を 1 つの要素と見なすと、dA と dB の乗算結果は図 8-4-5 (2) のように 2 × 2 の行列になります。各要素内の「〓＋〓」の計算を行うと、図 8-4-5 (3) の行列 dC となります。

図 8-4-5 (3) の配列 dC 内の各要素を、ブロック数 2 × 2、ブロック内のスレッド数 2 × 2 の各スレッドが担当します。例えば (1,0) のブロック内の 2 × 2 個のスレッドは、図 8-4-5 (2) と図 8-4-5 (3) の〓内を担当します。この部分を抽出すると図 8-4-6 になり、(1,0) のブロック内の 2 × 2 個のスレッドは、この計算を行います。

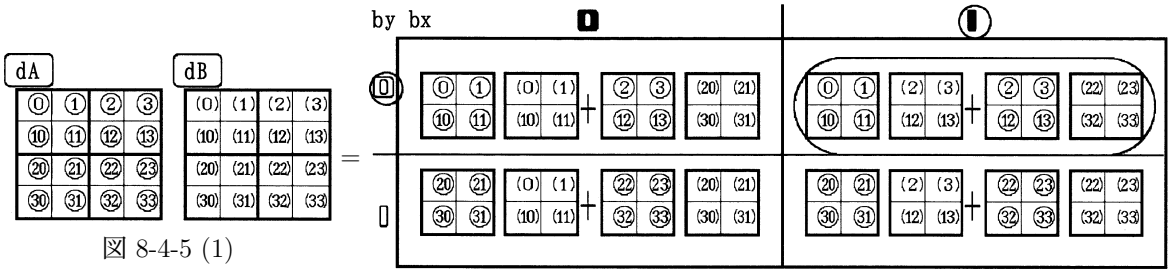


図 8-4-5 (1)

図 8-4-5 (2)

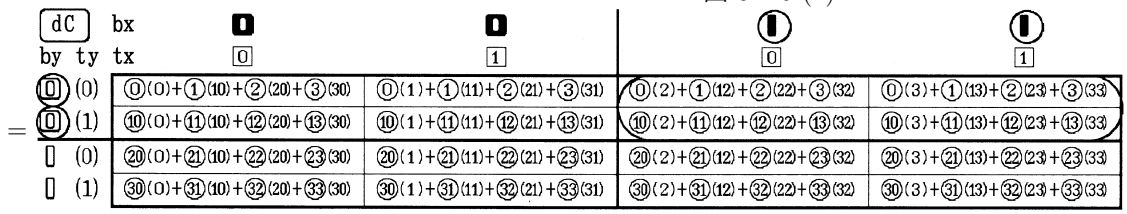


図 8-4-5 (3)

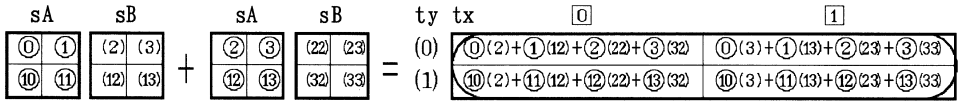


図 8-4-6　(1,0) のブロック内の 2 × 2 個のスレッドが行う計算

シェアードメモリを使用するプログラム

以後、図 8-4-6 の計算を行う、ブロック (1,0) 内の 2 × 2 個のスレッドの動作を中心に説明します。図 8-4-2 に対してシェアードメモリを使用したプログラムを図 8-4-7 に示します。プログラムを実行したときの、ブロック (1,0) 内の 2 × 2 個のスレッドの動作を図 8-4-8 (1) (2) に示します。

- 図 8-4-7 の [0] で、シェアードメモリ上に配列 sA，sB を確保します。sA，sB は、図 8-4-6 の〓の計算 (2 回) で使用します。なお、配列 sA，sB の大きさを例えば 16 × 16 にしても、バンクコンフリクトは発生しません。理由は 3-6 節の「■　行列乗算での使用例」を参照して下さい。
- [1] で各スレッドは変数 sum をゼロクリアします。変数 sum はスレッドごとに別のレジスターに置かれます。
- [2] のループは、（本例では）k = 0,2 と 2 回反復します。k = 0 のとき、図 8-4-6 の左側の〓を図 8-4-8 (1) のように計算し、k = 2 のとき、図 8-4-6 の右側の〓を図 8-4-8 (2) のように計算します。
- まず [2] で k = 0 となり、[3] と [4] で、例えばブロック (1,0) 内の 2 × 2 個の各スレッドは、図 8-4-8 (1) の [3] の○を配列 sA に、[4] の○を配列 sB に、1 スレッドが 1 要素ずつロードします。
- 同一ブロック内の全スレッドが [3]，[4] のロードを完了する前に、あるスレッドが [6] の計算に入るのを防ぐため、[5] で同期を取ります。
- [6] で各スレッドは、図 8-4-6 の左側の〓のうち、自分が担当する部分の行列計算を行い、結果を変数 sum に加算します。その結果、各スレッドの変数 sum の値は、図 8-4-8 (1) のようになります。
- 同一ブロック内のあるスレッドが [6] の計算を完了しないうちに、[6] の計算を終了した他のスレッドが [2] の 2 回目の反復の [3]，[4] で、配列 sA，sB に次の値を代入してしまうのを防ぐため、[7] で同期を取ります。
- [2] で k = 2 となり、[3] と [4] で、例えばブロック (1,0) 内の 2 × 2 個の各スレッドは、図 8-4-8 (2) の [3] の○を配列 sA に、[4] の○を配列 sB に、1 スレッドが 1 要素ずつロードします。[5]，[7] は k = 0 のときと同じです。

- [6] で各スレッドは、図 8-4-6 の右側の〓〓のうち、自分が担当する部分の行列計算を行い、結果を変数 sum に加算します。その結果、各スレッドの変数 sum の値は、図 8-4-8 (2) のようになります。
- 最後に [8] で、例えばブロック (1,0) 内の 2 × 2 個の各スレッドは、図 8-4-8 (2) の [8] の〓を配列 dC に、1 スレッドが 1 要素ずつストアします。
- 1 スレッドあたり、グローバルメモリからのロードが図 8-4-7 の [3], [4] の 2 × 2 回、ストアが [8] の 1 回で、残りは [6] で行う高速なシェアードメモリ sA, sB のロード／ストアのみなので、図 8-4-2 よりも速度が向上します。

```
#define N (4)
#define BLOCK (2)
  __device__ float dA[N][N],dB[N][N],dC[N][N];
__global__ void kernel(){
  __shared__ float sA[BLOCK][BLOCK],        [0]
                   sB[BLOCK][BLOCK];        [0]
  int i = blockIdx.y*BLOCK + threadIdx.y;
  int j = blockIdx.x*BLOCK + threadIdx.x;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  float sum = 0.0f;                         [1]
  for(int k=0;k<N;k+=BLOCK){                [2]
    sA[ty][tx] = dA[i][k+tx];               [3]
    sB[ty][tx] = dB[k+ty][j];               [4]
    __syncthreads();                        [5]
    for(int kk=0;kk<BLOCK;kk++){
      sum = sum + sA[ty][kk]*sB[kk][tx];    [6]
    }
    __syncthreads();                        [7]
  }
  dC[i][j] = sum;                           [8]
}
int main(void){
    :
  kernel<<<dim3(2,2),dim3(2,2)>>>();
    :
```

図 8-4-7

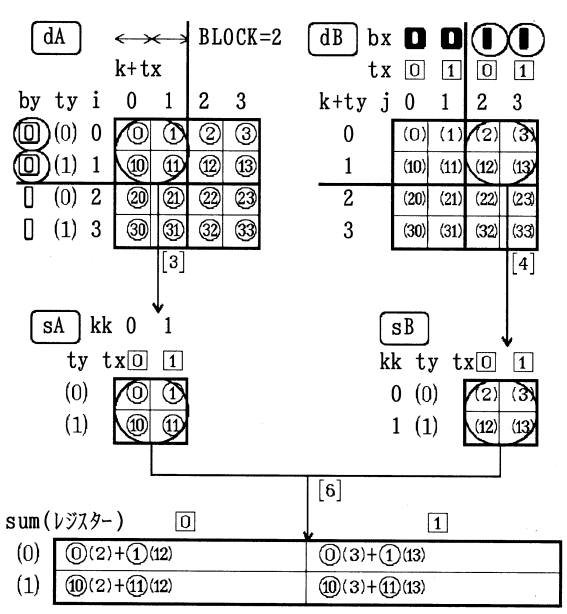


図 8-4-8 (1)　k = 0 のとき

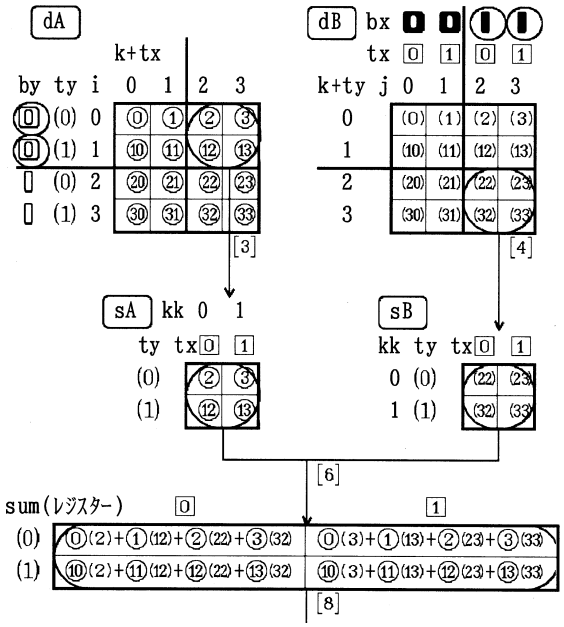


図 8-4-8 (2)　k = 2 のとき

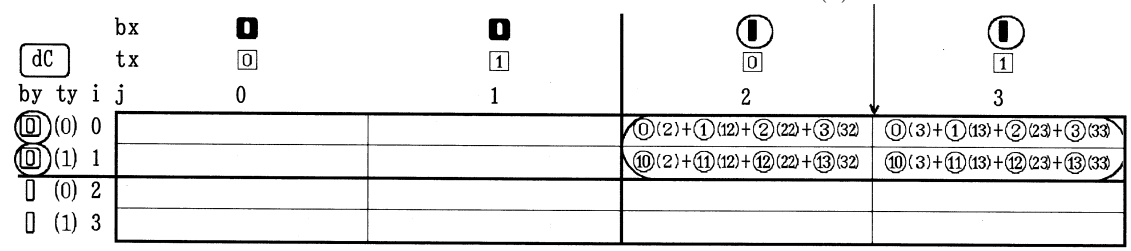


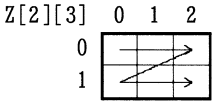
図 8-4-8 (3)

8-5　行列の転置

本節では、正方行列の各要素を転置するプログラムを CUDA 化する方法を説明します。実際のプログラムで、行列の転置を CUDA 化することはあまりないと思いますが、CUDA 化の際、コアレスアクセスやバンクコンフリクトなどを考慮する必要があり、他のプログラムを CUDA 化するときの参考になるため、取り上げました。

2 次元配列の復習

転置プログラムは、ブロック ID、スレッド ID と、2 次元配列の添字の対応付けが分かりにくいので、まず C 言語の 2 次元配列について簡単に復習します (1-5 節参照)。例えば配列 Z[2][3] は、本書では下図に示すように、配列 Z[2][3] の右側の添字を横方向、配列 Z[2][3] の左側の添字を縦方向で表します。また C 言語の場合、各要素はメモリ上で図の矢印の順に並びます (Fortran では図の縦方向に並びます)。



元のプログラム

元の転置プログラムを図 8-5-1 に示します。①で、図 8-5-2 (1) (2) に示すように、配列 A[32][32] の各要素を転置して配列 B[32][32] に代入します。

```
#define N (32)
int main(void){
  int ix,iy;
  float A[N][N],B[N][N];
    :
  for(iy=0;iy<N;iy++){
    for(ix=0;ix<N;ix++){
      B[ix][iy] = A[iy][ix];    ①
    }
  }
    :
```

図 8-5-1

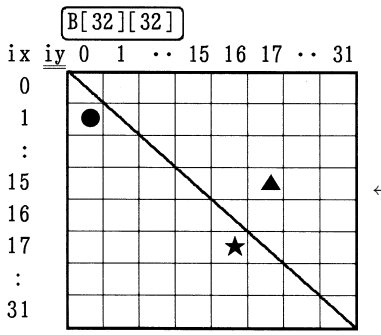


図 8-5-2 (2)

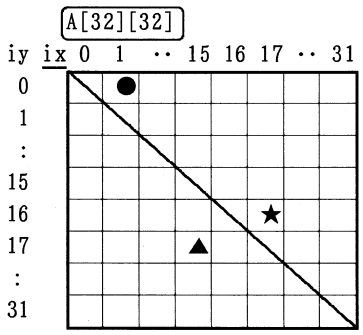


図 8-5-2 (1)

CUDA 版プログラム（チューニングなし）

CUDA 化したプログラム（チューニングなし）を図 8-5-3 に示します。なお、説明を簡単にするため、要素数がブロックあたりのスレッド数で割り切れるとし、割り切れない場合の処理は省略します。

- ③で、図 8-5-5 (1) に示す配列 dA[32][32] と、図 8-5-5 (2) に示す配列 dB[32][32] を確保します。カーネル関数内で 2 次元配列を扱うため、__device__ 修飾子を使用します。
- ⑦でブロック数を 2 × 2、ブロック内のスレッド数を 16 × 16 とし、⑧でカーネル関数を実行します。ブロック／スレッドの構成を図 8-5-4 に示します (図 8-5-4 は配列 dA，dB の図ではありません)。
- ⑥で、図 8-5-5 (1) の配列 dA の要素をロードし、転置して図 8-5-5 (2) の配列 dB にストアします。
  配列 dA[iy][ix] の右側の添字 ix が図 8-5-5 (1) の横方向になります。各ブロック／スレッドが配列 dA の要素を図 8-5-5 (1) のように担当する場合、添字 ix は、x 方向のブロック ID (0,1) と x 方向のスレッド ID (0〜15) から決まり、これを④で設定します。同様に添字 iy は、y 方向のブロック ID (0,1) と y 方向のスレッド ID ((0)〜(15)) から決まり、これを⑤で設定します。
  配列 dB[ix][iy] では、右側の添字 iy が図 8-5-5 (2) の横方向になります。
- 3-2 節で説明したように、配列のロード／ストアはハーフワープ単位で行われます。図 8-5-4 で、例えば x 方向に並んでいる太い〓内の 16 個のスレッドはハーフワープです。以後、この〓を対象ハーフワープと呼び、動作を検討します。対象ハーフワープのブロック ID は (1,0)、スレッド ID は (0, (0))〜(15, (0)) です。
  対象ハーフワープ内の各スレッドが担当する要素は、図 8-5-5 (1) では太い〓に示すようにメモリ上で連続しているので、配列 dA からのロードは高速なコアレスアクセスになりますが、図 8-5-5 (2) では〓に示すようにメモリ上で飛び飛びになっているので、配列 dB へのストアはコアレスアクセスにならず低速になります (3-3 節参照)。

```
#define N (32)
#define BLOCK (16)                                    ②
  __device__ float dA[N][N],dB[N][N];                 ③
__global__ void kernel(){
  int ix = blockIdx.x*BLOCK + threadIdx.x;            ④
  int iy = blockIdx.y*BLOCK + threadIdx.y;            ⑤
  dB[ix][iy] = dA[iy][ix];                            ⑥
}
int main(void){
    :
  dim3 NBLOCKS(2,2),NTHREADS(BLOCK,BLOCK);            ⑦
  kernel<<<NBLOCKS,NTHREADS>>>();                     ⑧
    :
```

図 8-5-3

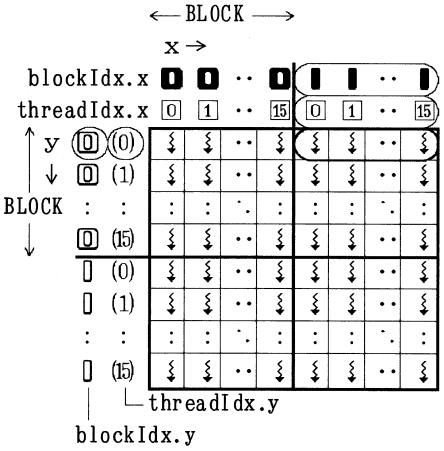


図 8-5-4

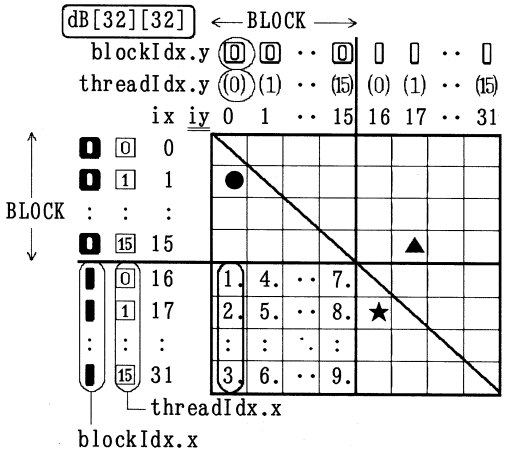


図 8-5-5 (2)

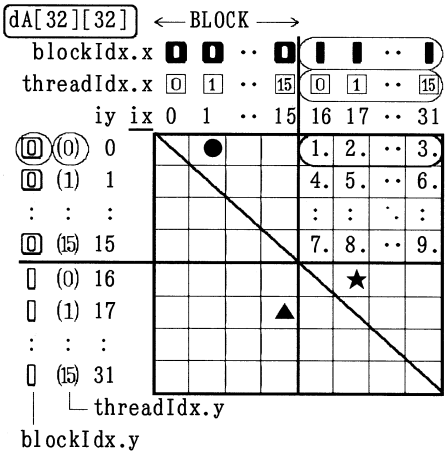


図 8-5-5 (1)

CUDA 版プログラム（チューニングあり）

配列 dB へのストアがコアレスアクセスにならない問題を、シェアードメモリを使用して解決するプログラムを図 8-5-6 に示します。

まずデータの動きを概観します。例えばブロック (1, 0) の各スレッドは、図 8-5-7 (1) に示す配列 dA の右上の部分を、図 8-5-8 (1) に示すシェアードメモリ上の配列 dS にロードします。次に、図 8-5-8 (2) (図 8-5-8 (1) と同じ) に示すシェアードメモリ上の配列 dS を、転置して図 8-5-7 (2) に示す配列 dB の左下にストアします。

前述の対象ハーフワープに着目すると、対象ハーフワープ内の各スレッドは、図 8-5-7 (1) の〓を、図 8-5-8 (1) の〓にロードします。図 8-5-7 (1) の〓内の要素はメモリ上で連続しているので、配列 dA からのロードは高速なコアレスアクセスになります。

対象ハーフワープが、図 8-5-8 (1) の〓の部分をそのまま転置して図 8-5-7 (2) にストアした場合、ストアする場所は点線の〓になります。ところが〓内の要素はメモリ上で飛び飛びになっているため、配列 dB へのストアはコアレスアクセスにならず、低速になってしまいます。

一方、対象ハーフワープが、図 8-5-8 (2) の〓の部分を転置して図 8-5-7 (2) にストアした場合、ストアする場所は〓になります。〓内の要素はメモリ上で連続しているため、配列 dB へのストアは高速なコアレスアクセスになります。従って、本プログラムではこの方法でストアを行います。

この方法を用いた場合の、各配列の横方向、縦方向のスレッド ID について説明します。図 8-5-7 (1) に示すように、対象ハーフワープのスレッド ID は (0, (0))〜(15, (0)) なので、図 8-5-8 (1) では、〓から分かるように横方向のスレッド ID が0〜15になります。同様に、図 8-5-8 (2) では、〓から分かるように縦方向のスレッド ID が0〜15になり、図 8-5-7 (2) では、〓から分かるように横方向のスレッド ID が0〜15になります。

次に、この方法を用いた場合の、配列 dB の横方向、縦方向のブロック ID について説明します。対象ハーフワープのブロック ID は (1, 0) なので、図 8-5-7 (2) のように、左下のブロックの横方向を 0 で縦方向を 1 にするか、あるいは図 8-5-9 のように、左下のブロックの横方向を 1 で縦方向を 0 にするか、2 通りの方法が考えられます。ところが、例えば図 8-5-7 (1) の左上の（要素●を含む）ブロックの場合、ブロック ID が (0, 0) なので、図 8-5-7 (2) では左上（正しい）になりますが、図 8-5-9 では右下（誤り）になってしまい、ブロックの位置がおかしくなります。従って、図 8-5-7 (2) が正しいブロック ID となります。

以下、図 8-5-6 のプログラムを説明します。

- ⑨で、配列 dS[16][17] をシェアードメモリとして宣言します (17 になっている理由は後述します)。
- ⑫で、図 8-5-7 (1) の配列 dA を、図 8-5-8 (1) の配列 dS にロードします。図 8-5-7 (1) の横方向は ix なので、⑫の配列 dA の右側の添字は ix になります。添字 ix は、図 8-5-7 (1) に示すように、x 方向のブロック ID (0, 1) と x 方向のスレッド ID (0〜15) から決まり、これを⑩で設定します。図 8-5-8 (1) の横方向は x 方向のスレッド ID (0〜15) なので、⑫の配列 dS の右側の添字は threadIdx.x になります。
- ⑯で、図 8-5-8 (2) の配列 dS を、図 8-5-7 (2) の配列 dB にストアします。図 8-5-8 (2) の横方向は y 方向のスレッド ID ((0)〜(15)) なので、⑯の配列 dS の右側の添字は threadIdx.y になります。図 8-5-7 (2) の横方向は ix なので、⑯の配列 dB の右側の添字は ix になります。添字 ix は、図 8-5-7 (2) に示すように、y 方向のブロック ID (0, 1) と x 方向のスレッド ID (0〜15) から決まり、これを⑭で設定します。
- 各スレッドは配列 dS にロードした要素 (図 8-5-8 (1) の〓) と（一般に）異なる要素 (図 8-5-8 (2) の〓) を配列 dS からストアするため、⑫でブロック内の全スレッドが配列 dS にロードしてからでないと、⑯のストアを行うことはできません。このため、⑬でブロック内の全スレッドの同期を取ります。
- 対象ハーフワープ内の各スレッドは、図 8-5-8 (2) に示すように、シェアードメモリ上の配列 dS の〓の部分を配列 dB にストアします。シェアードメモリの大きさが dS[16][16] だと、配列 dS から〓をロードする際、16 ウェイのバンクコンフリクト (3-6 節参照) が発生します。これを回避するため、⑨で dS[16][17] と宣言し、計算に使わない一列をパディングしています (図 8-5-8 (2) の着色した部分)。

```
#define N (32)
#define BLOCK (16)
  __device__ float dA[N][N],dB[N][N];
__global__ void kernel(){
  __shared__ float dS[BLOCK][BLOCK+1];        ⑨
  int ix = blockIdx.x*BLOCK + threadIdx.x;    ⑩
  int iy = blockIdx.y*BLOCK + threadIdx.y;    ⑪
  dS[threadIdx.y][threadIdx.x] = dA[iy][ix];  ⑫
  __syncthreads();                            ⑬
  ix = blockIdx.y*BLOCK + threadIdx.x;        ⑭
  iy = blockIdx.x*BLOCK + threadIdx.y;        ⑮
  dB[iy][ix] = dS[threadIdx.x][threadIdx.y];  ⑯
}
```

図 8-5-6

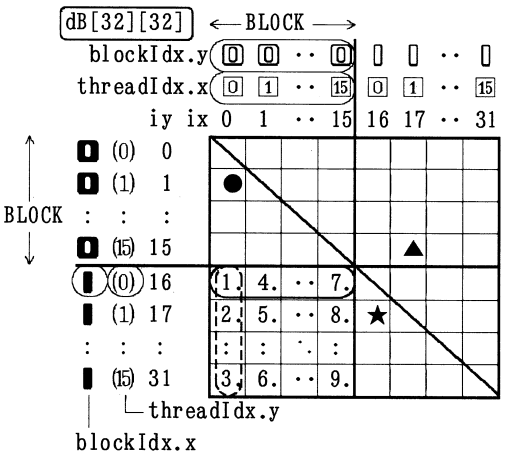


図 8-5-7 (2)

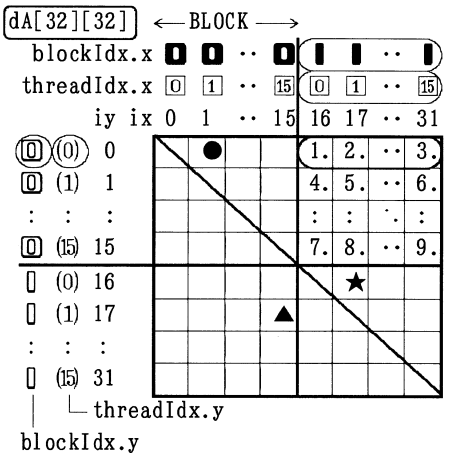


図 8-5-7 (1)

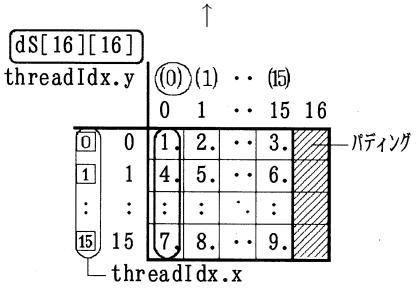


図 8-5-8 (2)

=

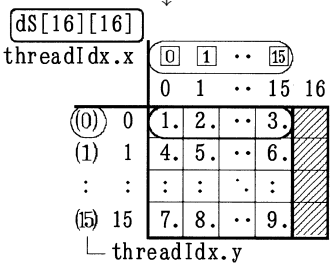


図 8-5-8 (1)

×

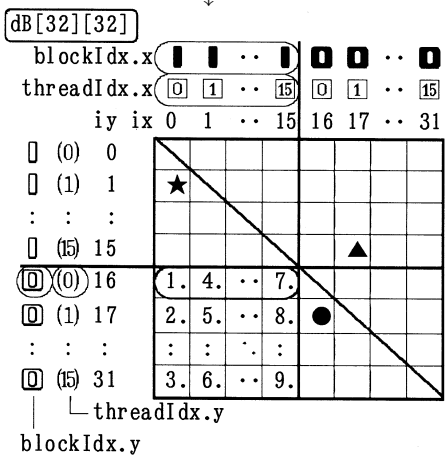


図 8-5-9　×

付 録

CUDA のマニュアル

理研 RICC の /usr/local/cuda/doc に、理研 RICC に導入されている CUDA のマニュアル（下記）が置かれています。「」内はマニュアル名、() 内は本書での関連する節です。なお、CUDA の最新版のマニュアルをダウンロードする場合は、http://developer.nvidia.com/cuda-downloads で「GET LATEST CUDA TOOLKIT PRODUCTION RELEASE」をクリックして下さい。

- CUDA_C_Programming_Guide.pdf　CUDA 全般の説明（本書全体）
  「CUDA C Programming Guide」
- CUDA_Toolkit_Reference_Manual.pdf　CUDA 関数の引数の説明（本書全体）
  「CUDA Reference Manual」
- CUDA_C_Best_Practices_Guide.pdf　CUDA のチューニングに関する説明 (本書 6-6 節)
  「CUDA C Best Practices Guide」
- nvcc.pdf　nvcc のコンパイルオプションの説明 (本書 2-7 節)
  「The CUDA Compiler Driver NVCC」
- ptx_isa_1.4.pdf　アセンブラ命令の説明 (本書 2-7 節)
  「PTX: Parallel Thread Execution ISA Version 1.4」
- ptx_isa_2.2.pdf　アセンブラ命令の説明 (本書 2-7 節)
  「PTX: Parallel Thread Execution ISA Version 2.2」
- cuda_memcheck.pdf　「cuda_memcheck」コマンドの説明 (本書 4-2 節)
  「cuda-memcheck User Manual」
- cuda_gdb.pdf　デバッガー (CUDA-GDB) の説明 (本書 4-3 節)
  「CUDA-GDB (NVIDIA CUDA Debugger)」
- Compute_Profiler.txt　プロファイラーの説明 (本書 4-5 節)
- CUBLAS_Library.pdf　BLAS の CUDA 版の説明 (本書 7-1 節)
- CUFFT_Library.pdf　FFT の CUDA 版の説明 (本書 7-2 節)
- CURAND_Library.pdf　擬似乱数生成ルーチンの CUDA 版の説明 (本書 7-4 節)
- CUSPARSE_Library.pdf　疎行列計算ルーチンの CUDA 版の説明 (本書 7-4 節)
- Fermi_Compatibility_Guide.pdf
- Fermi_Tuning_Guide.pdf
- OpenCL_Best_Practices_Guide.pdf
- OpenCL_Extensions
- OpenCL_Implementation_Notes.txt
- OpenCL_Jumpstart_Guide.pdf
- OpenCL_Programming_Guide.pdf
- OpenCL_Programming_Overview.pdf
- CUDA_Developer_Guide_for_Optimus_Platforms.pdf
- CUDA_VideoDecoder_Library.pdf
- EULA.txt

NVIDIA 社が作成した資料、Web サイト

NVIDIA 社が作成したマニュアル以外の資料、および GPU/CUDA に関する Web サイトを以下に示します。

- 「CUDA ZONE」
  http://www.nvidia.co.jp/cuda/（日本語）http://www.nvidia.com/cuda/（英語）
- 「エヌビディアジャパンチャンネル」
  http://www.youtube.com/user/NVIDIAJapan　セミナーの動画です。
- 「CUDA Programming Guide Version 1.1」（日本語訳）　CUDA の古い版 (2008 年 3 月) の日本語訳です。
  http://www.nvidia.co.jp/docs/IO（オー）/51174/NVIDIA_CUDA_Programming_guide_1.1_JPN.pdf
- 「GPU コンピューティングの初歩」
  http://http.download.nvidia.com/developer/cuda/jp/IntoGPUComputing_jp.pdf
- 「CUDA 実践エクササイズ」
  http://www.nvidia.co.jp/docs/IO/59373/Exercise_Instructions.pdf
- 「CUDA テクニカルトレーニング Vol I: CUDA プログラミング入門」
  http://www.nvidia.co.jp/docs/IO/59373/VolumeI.pdf
- 「CUDA テクニカルトレーニング Vol II: CUDA ケーススタディ」
  http://www.nvidia.co.jp/docs/IO/59373/VolumeII.pdf
- 「ホワイトペーパー NVIDIA の次世代 CUDA コンピュートアーキテクチャ：Fermi」
  http://www.nvidia.co.jp/object/fermi_architecture_jp.html の右上
- 「ソフトウェア開発ツール」
  http://www.nvidia.co.jp/object/tesla_software_jp.html
- 「GPU コンピューティングソリューションファインダー」
  http://www.nv-event.jp/solution-finder/index.php

理化学研究所情報基盤センター主催の講習会

下記は、当情報基盤センターが過去に開催した、GPU/CUDA の講習会の資料です。

- 「RICC ユーザ講習会 GPGPU の利用」(2009/11/5，2010/3/12)
  「RICC Portal」にログインし (登録ユーザーのみ)、左上の「Documents」をクリックし、「Programming Class Materials」にあります。また左上の「Example」をクリックすると、この講習会テキストに掲載されているサンプルプログラムがあります。

外部の講習会

外部で開催されている GPU/CUDA の講習会です。

- 「エヌビディア主催トレーニング」nvidia 社が開催するトレーニングコース
  http://www.nvidia.co.jp/object/cuda-dev-program-jp.html
- 「GPU コンピューティング研究会講習会」の配布資料
  http://gpu-computing.gsic.titech.ac.jp/Japanese/Lecture/index.html
- 「実践力アップ！　CUDA プログラミングセミナー」フィックスターズ
  http://www.fixstars.com/company/event/seminar.html
- 「GPGPU を用いた高効率アプリケーション開発基礎講座〜デモ付〜」日本テクノセンター
  http://www.j-techno.co.jp/test/index.cgi?mode=Sem&unit=2010100100
- 「GPU トレーニングコース」爆発研究所
  http://bakuhatsu.jp/computational/gpu-training/
- 「CUDA セミナー」テクノロジー・ジョイント
  http://www.tjc.ne.jp/ja/services/seminar
- 「AOC プランニング」
  https://sites.google.com/a/aocplan.com/web/gpgpu/business-results

- 「GPGPU 超並列プログラミングの基礎」日本アイ・ビー・エム
  http://www-06.ibm.com/jp/lsj/search/index.shtml で、コースのクイック選択を「GPGPU」として検索して下さい。
- HPC システムズ (2010 年 11 月現在開催準備中) http://www.hpc.co.jp/seminar.html
- 「研究者のための GPU コンピューティング入門半日集中セミナー」プロメテック・ソフトウェア
  http://www.prometech.co.jp/seminar/2009/07/2009-08-20.html
- 「セミナー・イベント情報」日本 GPU コンピューティング・パートナーシップ
  http://www.gdep.jp/page/index/seminar
- 「GPGPU セミナー」サードウェーブ
  http://gpgpu.dospara.co.jp/
- 「GPGPU/CUDA を用いたアプリケーション開発の基礎講座」R&D 支援センター
  http://www.rdsc.co.jp/seminar/110810.html

Web サイト

GPU/CUDA 関連の解説やリンク集などの Web サイトです。

- 東京工業大学青木教授の講義資料
  http://www.ocw.titech.ac.jp/ で「講義を探す」の欄に「GPU コンピューティング」とキーインし、「検索する」をクリックします。表示された画面で「講義ノート」をクリックします。
- 「これからの並列計算のための GPGPU 連載講座」
  http://www.cc.u-tokyo.ac.jp/publication/news/ (2010 年 1 月、3 月、5 月、7 月、9 月)
- 「GPGPU 概説」http://gpu.para.media.kyoto-u.ac.jp/lecture/H22/CSEA/11nbody.pdf
- 「GPGPU プログラミング入門」
  http://www.hpcs.is.tsukuba.ac.jp/~msato/lecture-note/prog-env2009/slide-GPGPU-prog.pdf
- 「GPGPU イントロダクション」
  http://www.opencae.jp/wiki/images/201010020(← オー)hshima_gpgpututorial.pdf
- 「CUDA GPU Computing への誘い」http://thinkit.co.jp/book/2010/07/02/1646
- 「CUDA 技術を利用した GPU コンピューティングの実際（前編）（後編）」
  http://www.kumikomi.net/archives/2008/06/12gpu1.php　http:// 左と同じ /2008/10/22gpu2.php
- 「超並列プロセサ—GeForce アーキテクチャと CUDA プログラミング」
  http://journal.mycom.co.jp/special/2008/cuda/index.html
- 「科学技術計算向け演算能力が引き上げられた GPU アーキテクチャ Fermi」
  http://journal.mycom.co.jp/articles/2009/10/07/nvidia_fermi/menu.html
- 「CUDA」http://tech.ckme.co.jp/cuda.shtml
- 「NVIDIA が次世代 GPU アーキテクチャ「Fermi」を発表」
  http://pc.watch.impress.co.jp/docs/column/kaigai/20091001_318463.html
- GPU 関連記事　http://www.4gamer.net/words/001/W00171/
- 「Wiki CUDA」http://imd.naist.jp/~fujis/cgi-bin/wiki/index.php?CUDA
- 「CUDA 入門・サンプル集」http://cudasample.net/
- 「NVIDIA CUDA Information Site」http://gpu.fixstars.com/index.php/
- 「PGI コンパイラ技術情報・TIPS」http://www.softek.co.jp/SPG/Pgi/tips.html
- 「PGI テクニカル情報・コラム」http://www.softek.co.jp/SPG/Pgi/TIPS/para_guide.html
- 「GPU で最速を極める BLOG」http://topsecret.hpc.co.jp/wiki/index.php
- 「G-DEP の高速演算記」http://www.gdep.jp/column
- 「ゼロから始める GPU コンピューティング」http://www.gdep.jp/page/view/203
- 「GPGPU 斬り捨て御免」http://hojin.dospara.co.jp/gpgpu_column/
- 「GPU コードジェネレーター」http://www.otb-japan.co.jp/gimmick/index.html
- 「CUDA 関連リンク」http://www.yasuoka.mech.keio.ac.jp/gpu/
- 「ハイゼンベルク・マシーンと gpgpu」http://www.iitaka.org/gpgpu.html

- 「CUDA, supercomputing for the Masses:」http://www.drdobbs.com/cpp/207200659
  下記は 2010 年 9 月 10 日現在の記事のタイトルです。下の方の「For More Information」に、下記の「Part 2」以降の記事があります。
  - Part 1　CUDA lets you work with familiar programming concepts while developing software that can run on a GPU
  - Part 2　A first kernel
  - Part 3　Error handling and global memory performance limitations
  - Part 4　Understanding and using shared memory (1)
  - Part 5　Understanding and using shared memory (2)
  - Part 6　Global memory and the CUDA profiler
  - Part 7　Double the fun with next-generation CUDA hardware
  - Part 8　Using libraries with CUDA
  - Part 9　Extending Hign-level Languages with CUDA
  - Part 10　CUDPP, a powerful data-parallel CUDA library
  - Part 11　Revisiting CUDA memory spaces
  - Part 12　CUDA 2.2 Changes the Data Movement Paradigm
  - Part 13　Using texture memory in CUDA
  - Part 14　Debugging CUDA and using CUDA-GDB
  - Part 15　Using Pixel Buffer Objects with CUDA and OpenGL
  - Part 16　CUDA 3.0 provices expanded capabilities
  - Part 17　CUDA 3.0 provices expanded capabilities and makes development easier
  - Part 18　Using Vertex Buffer Objects with CUDA and OpenGL
  - Part 19　Parallel Nsight Part 1: Configuring and Debugging Applications
  - Part 20　Parallel Nsight Part 2: Using the Parallel Nsight Analysis capabilities

質問を投稿できる Web サイト

下記のフォーラムで、GPU/CUDA 関連の質問を投稿することができます（ユーザー登録が必要です)。

- 「NVIDIA フォーラム」http://forum.nvidia.co.jp/
- (英語)「NVIDIA Forums」http://forums.nvidia.com/

研究会／勉強会

- GPU コンピューティング研究会　http://gpu-computing.gsic.titech.ac.jp/index-j.html
- オープン CAE 学会　並列計算分科会　http://www.opencae.jp/ の「事業のご案内」の「分科会」
- GPU コンピューティング勉強会　http://gpgpu.unitcom.co.jp/ の右の「TOPICS」
- GPGPU 勉強会　http://epa.scitec.kobe-u.ac.jp/~gpgpu/

書籍／雑誌（発売予定のものもあります）

[1]

- 「はじめての CUDA プログラミング」青木尊之、額田彰　著（工学社）
  http://www.kohgakusha.co.jp/support/cuda/index.html に正誤表があります。
- 「CUDA 高速 GPU プログラミング入門」岡田賢治　著（秀和システム）
- 「先端グラフィックス言語入門」安福健祐、伊藤拡、大熊建保　著（フォーラムエイト）
- 「GPGPU による並列処理」アスキードットテクノロジーズ (2009 年 12 月号)
- 「GPGPU プログラミング」アスキードットテクノロジーズ (2010 年 08 月号)
- 「日経ソフトウエア」(2011 年 8 月号) の記事「これから始める GPU プログラミング」
- 「プロセッサを支える技術」Hisa Ando　著（技術評論社）7 章 GPGPU と超並列処理
- 「CPU & GPU がわかる本」（工学社）
- （英語）Programming Massively Parallel Processors
- （上記の日本語訳）CUDA プログラミング実践講座―超並列プロセッサにおけるプログラミング手法
- （英語）CUDA by Example: An Introduction to General-Purpose GPU
- （上記の日本語訳）汎用 GPU プログラミング入門
- （英語）NVIDIA GPU Programming: Massively Parallel Programming with CUDA
- （英語）Multi-core Programming with Cuda and Opencl
- （英語）CUDA Application Design and Development
- 「OpenCL 入門」フィックスターズ　著（インプレスジャパン）
- 「OpenCL 並列プログラミング」池田成樹　著（カットシステム）
- 「OpenCL 入門」奥薗隆司　著（秀和システム）
- 「並行コンピューティング技法」千住治郎　訳（オライリー・ジャパン）

[2]

- 「チューニング技法入門」http://accc.riken.jp/HPC/training.html