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

（このページは空白です。）

● 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がありましたらご容赦下さい。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、GPUメーカーのNVIDIA(ｴﾇﾋﾞﾃﾞｨｱ)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(ｸｰﾀﾞ)(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のアーキテクチャー

GPUのアーキテクチャーは、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なく、「Compute Capability」という用語を使用し、例えば「～の機能は 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CUDA3.2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)。

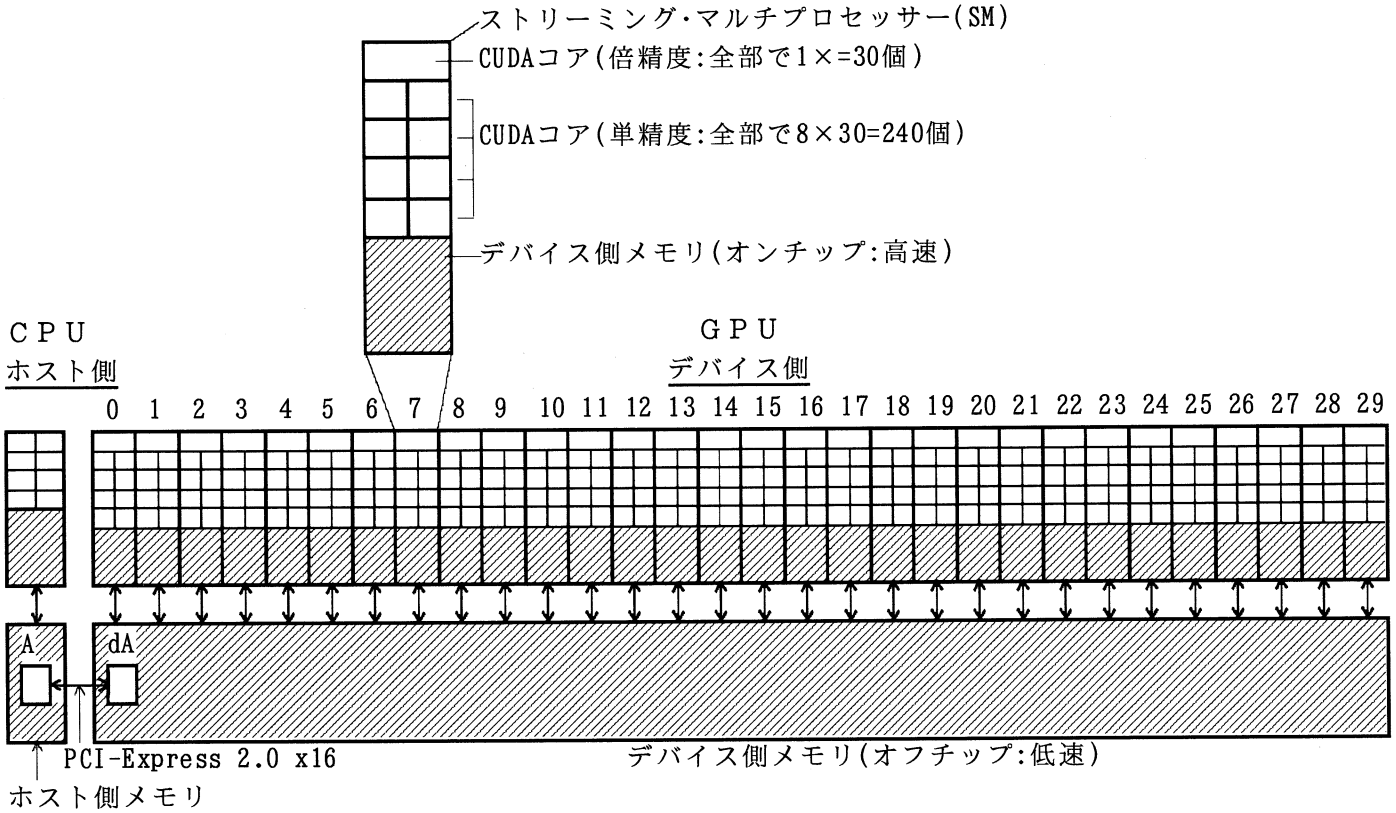


図1-2-2

■ 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ー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は元々画像処理用に使用されており、倍精度は不要でしたが、倍精度が要求されるHPC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 ReferenceManual」(付録参照)を参照して下さい。

(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 簡単な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)

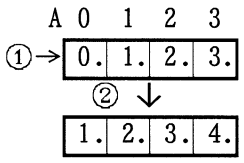


図1-3-1(2)

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で②と同じ計算を行い、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⑽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),⑽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⑾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、⑾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コアで並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)。

●⑿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)。⑿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。

⑿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⑿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のマニュアル「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「ｽﾚｯﾄﾞｲﾝﾃﾞｯｸｽ.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(0,1,2,3)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);                           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)

```
#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
  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
                        ｶｰﾈﾙ関数を実行する。  (13)
  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)

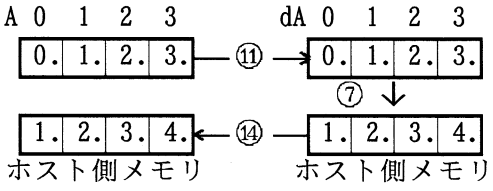


図1-3-2(2)

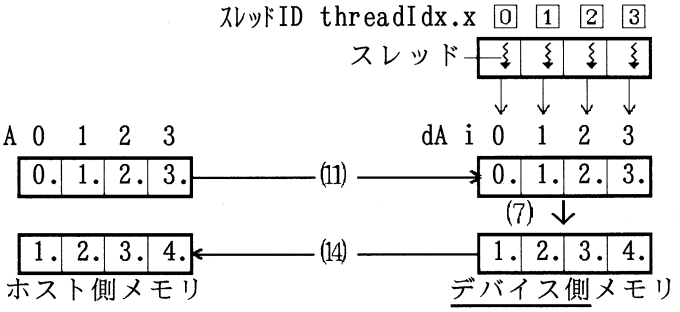


図1-3-3(2)

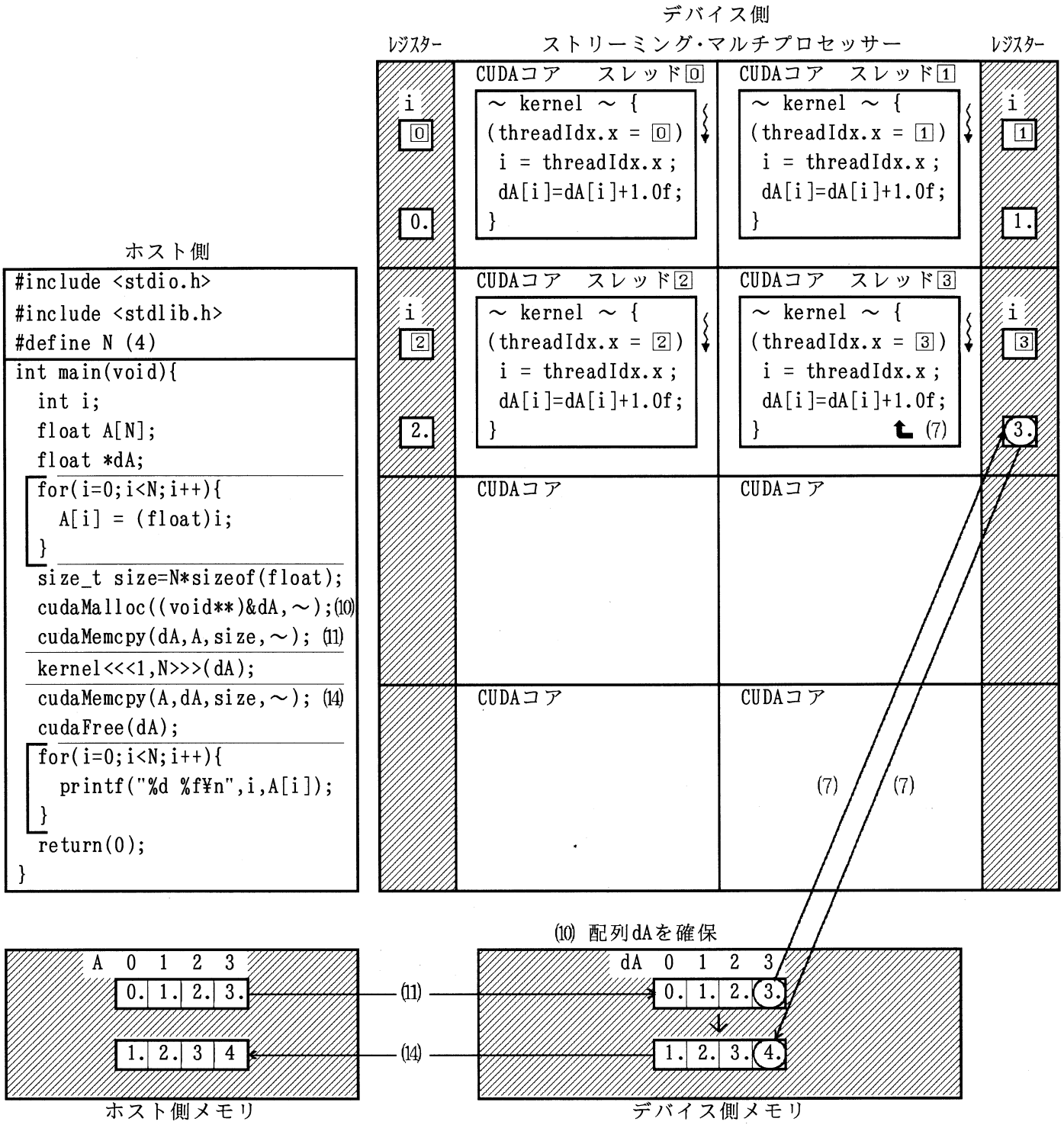

デバイス側
レジスター
ストリーミング・マルチプロセッサー
レジスター
CUDAコア　スレッド0
CUDAコア　スレッド1
CUDAコア　スレッド2
CUDAコア　スレッド3
～ kernel ～ {
(threadIdx.x = 0)
(threadIdx.x = 1)
(threadIdx.x = 2)
(threadIdx.x = 3)
i = threadIdx.x ;
dA[i]=dA[i]+1.0f;
}
(7)
CUDAコア
i
0.
1.
2.
3.
ホスト側
#include <stdio.h>
#include <stdlib.h>
#define N (4)
int main(void){
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
0. 1. 2. 3.
1. 2. 3 4
(11)
(14)
dA 0 1 2 3
0. 1. 2. 3.
1. 2. 3. 4.
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並列やスレッド並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で作成されたロードモジュルが入っている、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ドライバーAPIのバージョン)
  CUDA Runtime Version:                          3.20          (CUDAランタイムAPIのバージョン)
  CUDA Capability Major/Minor version number:    1.3           (Compute Capability 1.3)
  Total amount of global memory:                 4294770688 bytes (グローバルメモリの大きさは4Gバイト)
  Multiprocessors x Cores/MP = Cores:            30 (MP) x 8 (Cores/MP) = 240 (Cores)
                                    (ストリーミングマルチプロセッサーを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バイト)
  Total amount of shared memory per block:       16384 bytes (ブロック(注)あたりのシェアードメモリは16Kバイト)
  Total number of registers available per block: 16384        (ブロック(注)あたり利用可能な
                                                               レジスターの個数は16K個)
  Warp size:                                     32           (ワープあたりのスレッド数は32個)
  Maximum number of threads per block:           512          (ブロックあたり最大512スレッド)
  Maximum sizes of each dimension of a block:    512 x 512 x 64    (ブロック内のスレッドの最大の個数)
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1 (グリッド内のブ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       (データのコピ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       (ホストページロックメモリマッピングがサポートされている)
  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。

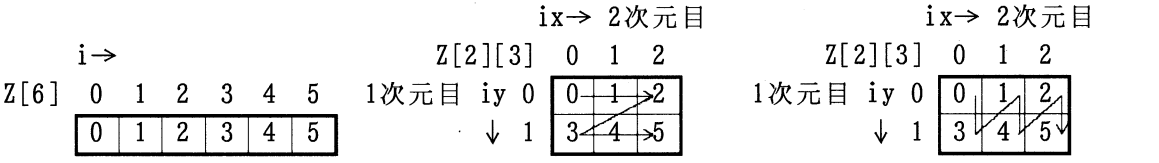
■ 2次元配列

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)。

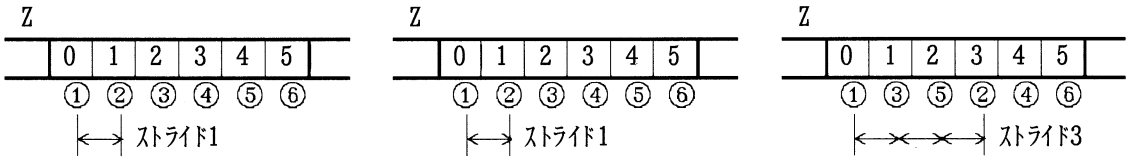
- 本書では図1-5-2に示すように、配列Z[2][3]の左側の添字を1次元目、右側の添字を2次元目と呼びます。
- 図1-5-1(2)のAに示すように、本書内の図は、2次元配列の1次元目を縦方向、2次元目を横方向とします。
- C言語の場合、2次元配列は、図1-5-1(2)のBに示す順番にメモリ上に並びます。つまり図1-5-1(2)のAの矢印の順番にメモリ上に並びます。なお、Fortranの場合はC言語と並ぶ順番が逆になりますので注意して下さい。図1-5-1(2)のCのループでは、Bの①～⑥の順にストライドが1で要素をアクセスします。
- 図1-5-1(2)とループの順番を逆にした図1-5-1(3)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。

A:本書内の図



B:メモリ上の配置



C:ループ例

```
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。

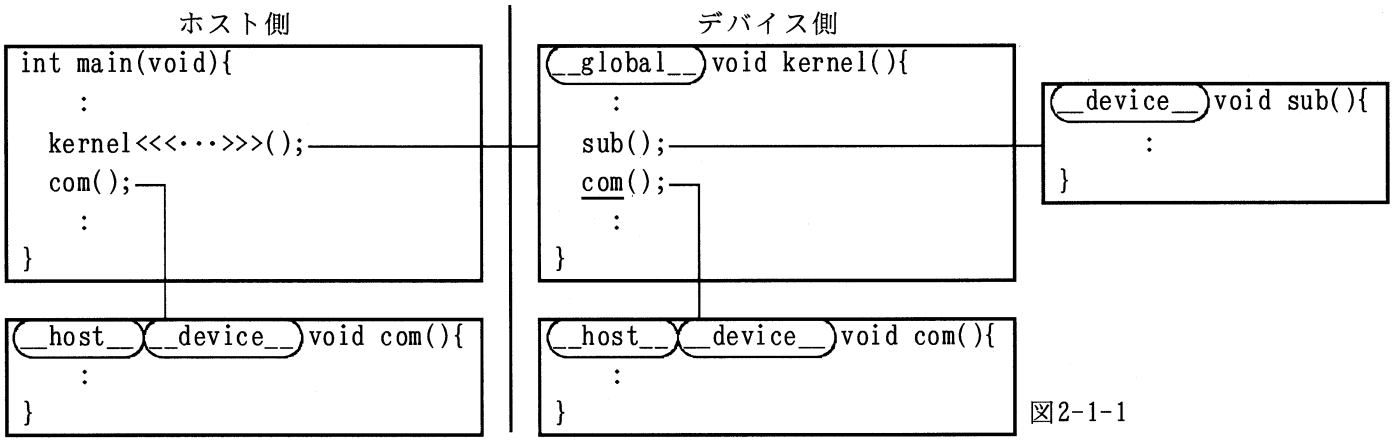


図2-1-1

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変数は、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();
    :
```

図2-1-2

実行結果

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

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

```
int main(void){
  func(1);
    :
```

図2-1-3

実行結果

```
1
2
3
4
5
```

(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);
    :
```

図2-1-4

実行結果

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

(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、1CPU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CPU0で、プロセス**1**はCPU1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**(スレッドと配列の関係)：スレッド**0**と**1**が、それぞれ64要素ずつ担当します。
- **B**(スレッドとCPUの関係)　：(例えば)スレッド**0**はCPU0で、スレッド**1**はCPU1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

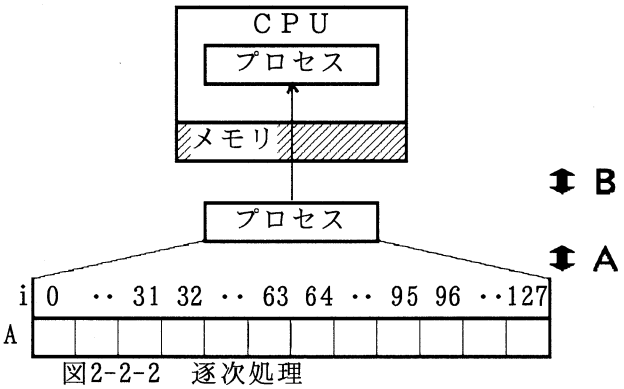


図2-2-2　逐次処理

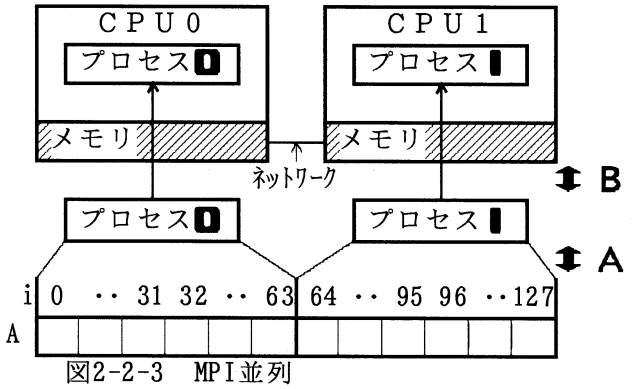


図2-2-3　MPI並列

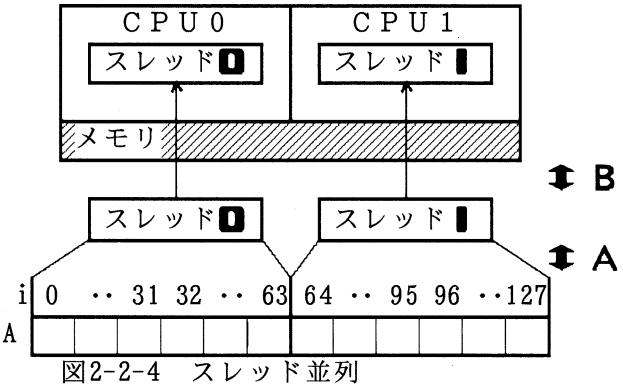


図2-2-4　スレッド並列

■ 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ID0～31)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ID31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)。

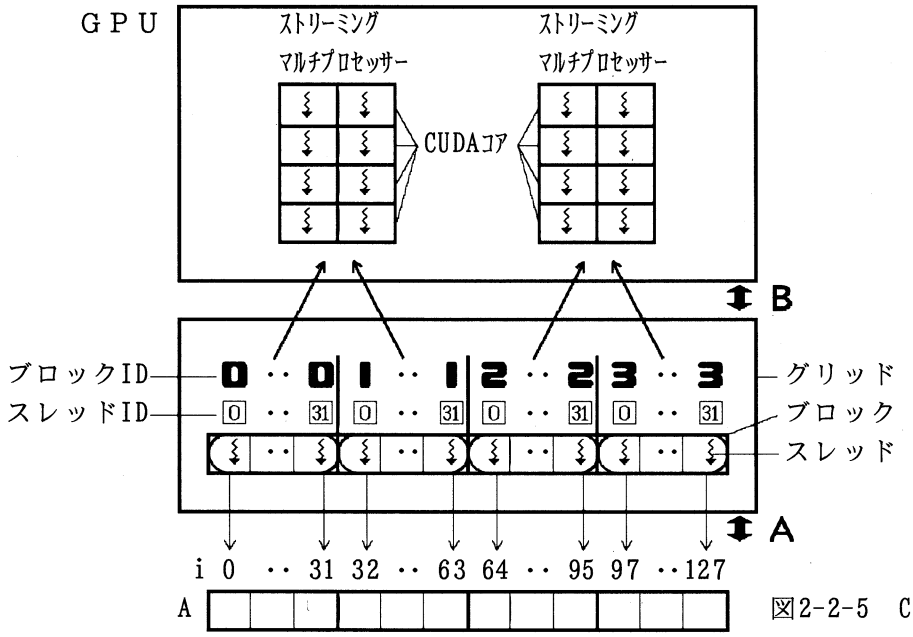


図2-2-5　CUDA並列

■ 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ID0, 1)含まれ、各ブロックにスレッドが4個(スレッドID0,1,2,3)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」((ｸﾞﾘｯﾄﾞ内の)ﾌﾞﾛｯｸの数)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」((ﾌﾞﾛｯｸ内の)ｽﾚｯﾄﾞの数)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、本例では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。

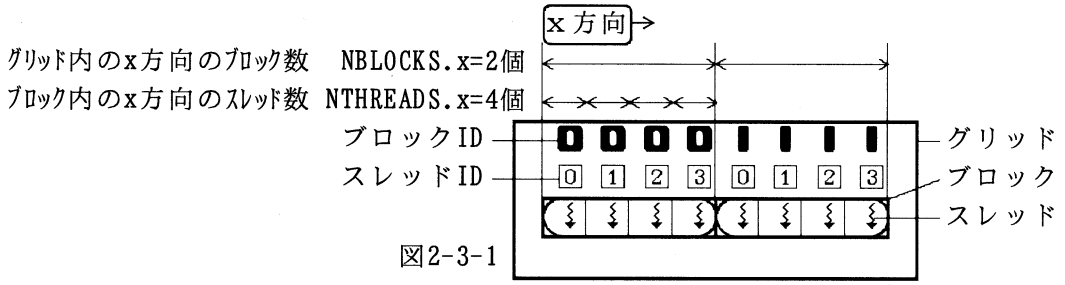


図2-3-1

(1)
```
dim3 NBLOCKS,NTHREADS; (名前は任意)     ①
NBLOCKS.x = 2;  ｸﾞﾘｯﾄﾞ内のx方向のﾌﾞﾛｯｸ数 ②
NBLOCKS.y = 1;  ｸﾞﾘｯﾄﾞ内のy方向のﾌﾞﾛｯｸ数 ③
NBLOCKS.z = 1;  ｸﾞﾘｯﾄﾞ内のz方向のﾌﾞﾛｯｸ数 ④
NTHREADS.x = 4; ﾌﾞﾛｯｸ内のx方向のｽﾚｯﾄﾞ数 ⑤
NTHREADS.y = 1; ﾌﾞﾛｯｸ内のy方向のｽﾚｯﾄﾞ数 ⑥
NTHREADS.z = 1; ﾌﾞﾛｯｸ内のz方向のｽﾚｯﾄﾞ数 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方向のスレッド数=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)。

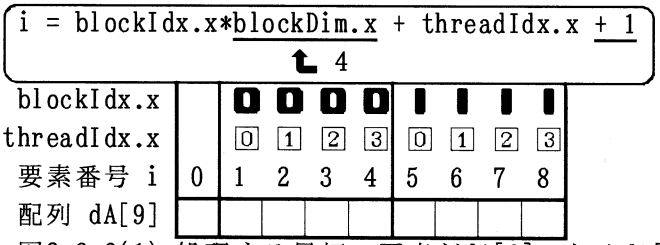


図2-3-3(1) 処理する最初の要素がdA[0]でなくdA[1]

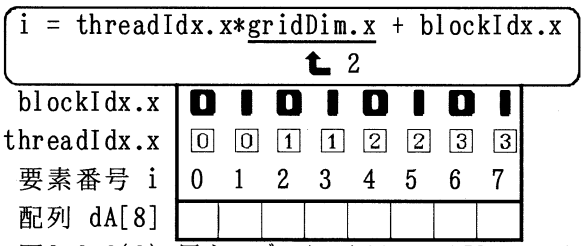


図2-3-3(3) 異なるブロックの同じスレッドIDのスレッドが連続した要素を処理

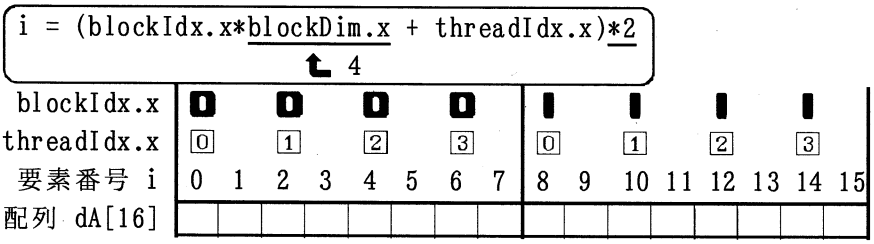


図2-3-3(2) 1要素飛びに処理

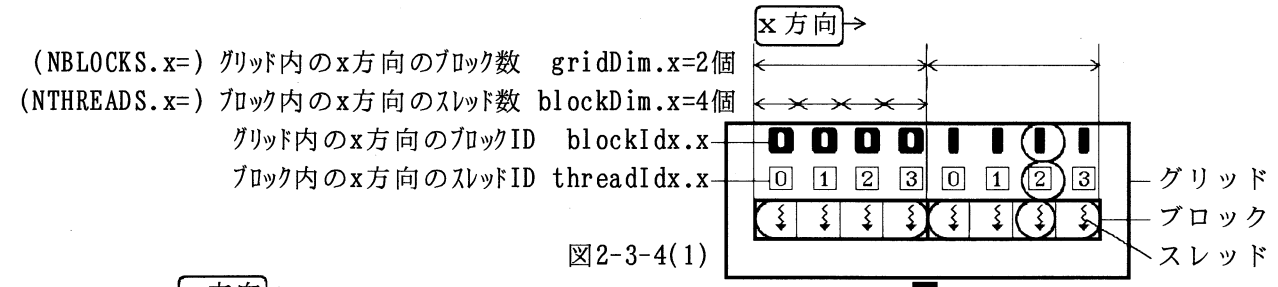


図2-3-4(1)

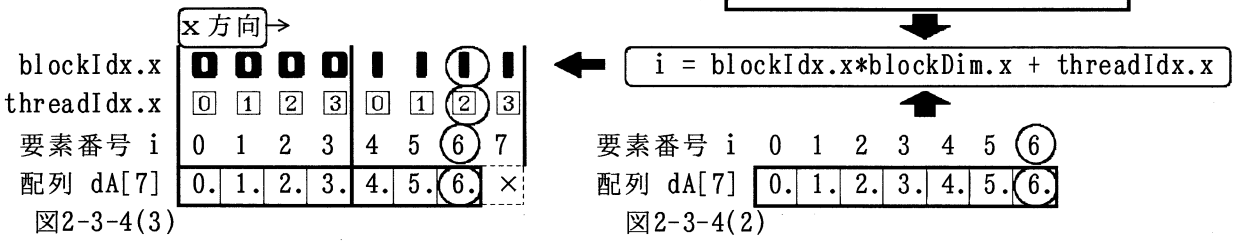


図2-3-4(3)　　図2-3-4(2)

```
__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0,1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 =   0   *   4   +   0   ; (= 0)
      i =   0   *   4   +   1   ; (= 1)
      i =   0   *   4   +   2   ; (= 2)
      i =   0   *   4   +   3   ; (= 3)                                    ⑨
      i =   1   *   4   +   0   ; (= 4)
      i =   1   *   4   +   1   ; (= 5)
      i =   1   *   4   +   2   ; (= 6)
      i =   1   *   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
ｸﾞﾘｯﾄﾞ内のx方向のﾌﾞﾛｯｸ数=2	NBLOCKS.x ①	gridDim.x ⑥
ｸﾞﾘｯﾄﾞ内のy方向のﾌﾞﾛｯｸ数=3	NBLOCKS.y ②	gridDim.y ⑦
ﾌﾞﾛｯｸ内のx方向のｽﾚｯﾄﾞ数=4	NTHREADS.x ③	blockDim.x ⑧
ﾌﾞﾛｯｸ内のy方向のｽﾚｯﾄﾞ数=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(1)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。

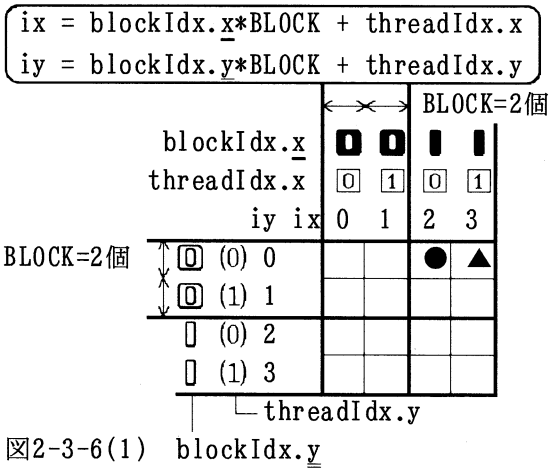


図2-3-6(1)

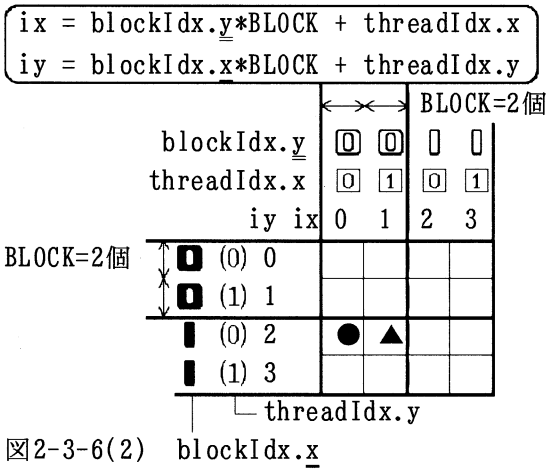


図2-3-6(2)

●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。

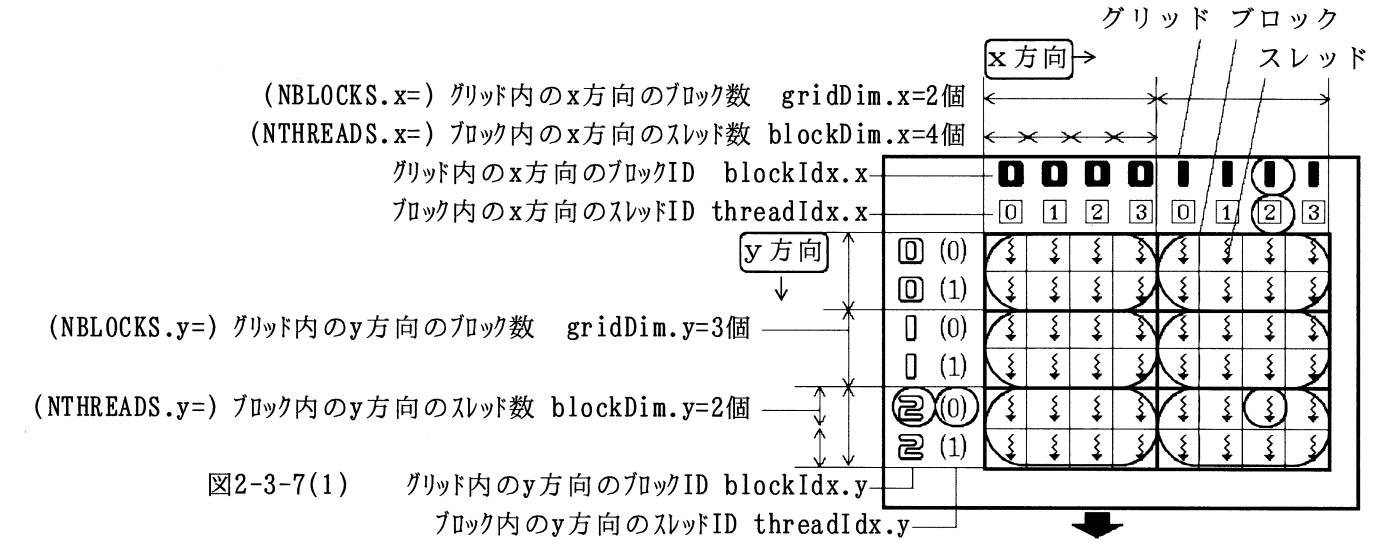


図2-3-7(1)

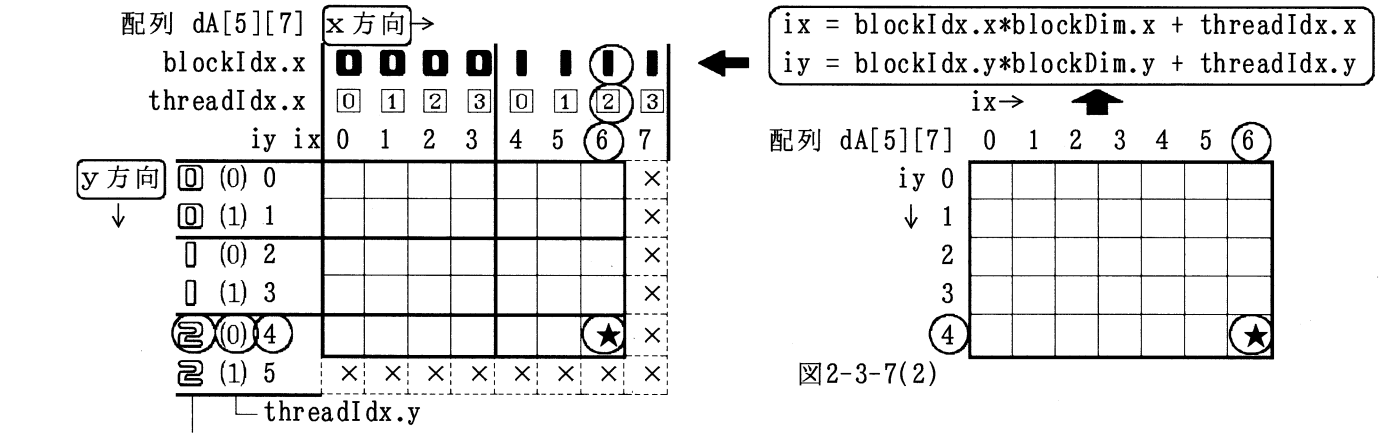


図2-3-7(2)

図2-3-7(3)

```
__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, [] ,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方向が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方向のｽﾚｯﾄﾞ数,y方向のｽﾚｯﾄﾞ数,z方向のｽﾚｯﾄﾞ数) = (8,8,8) などが、1ブロックの最大のスレッド数です。
- 本書では、2次元の場合、ブロック(**0**,**0**)、スレッド(0,(1))のように、(x,y)の順に表します。

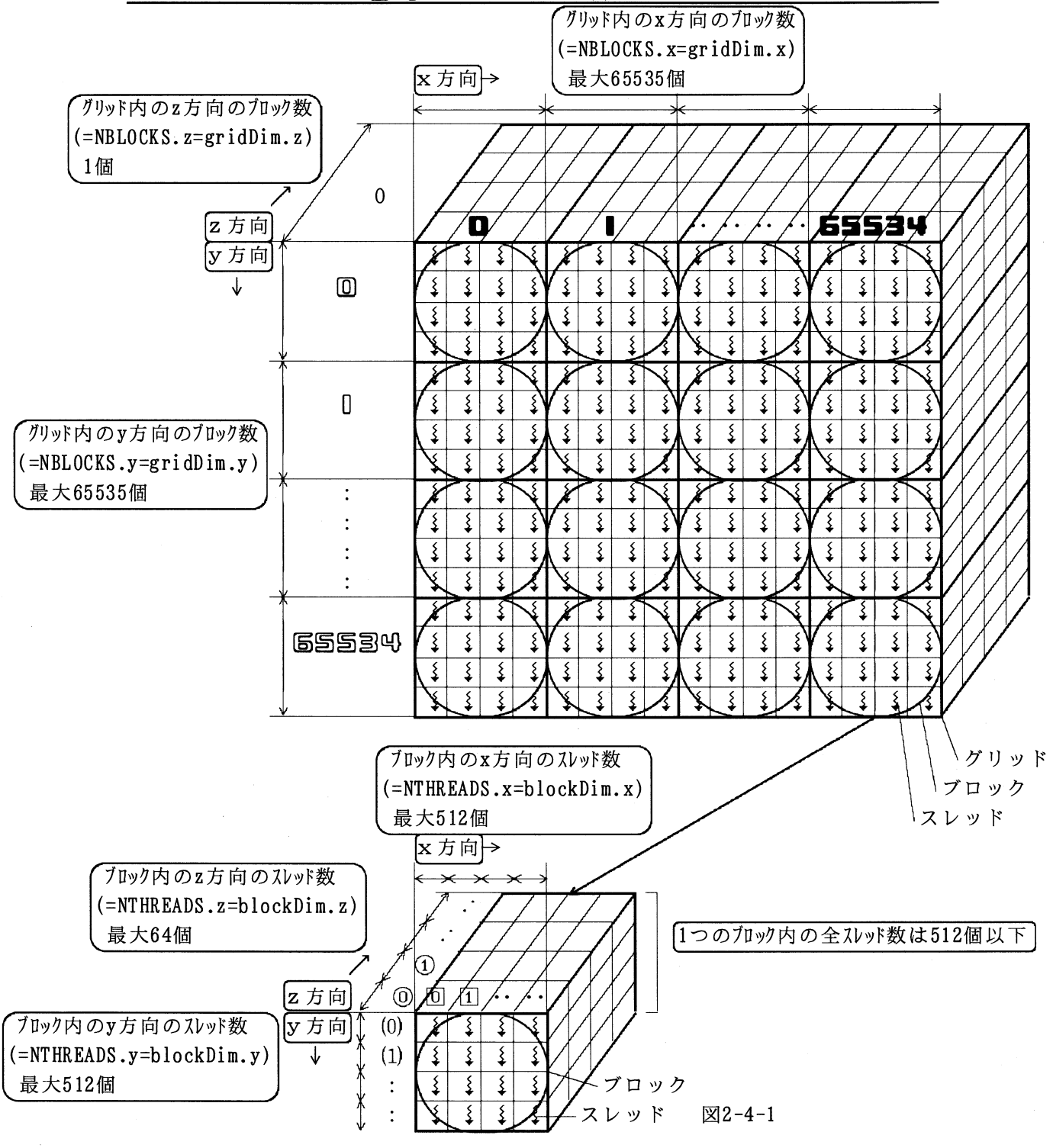


図2-4-1

■ 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、矢印に示す順(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)

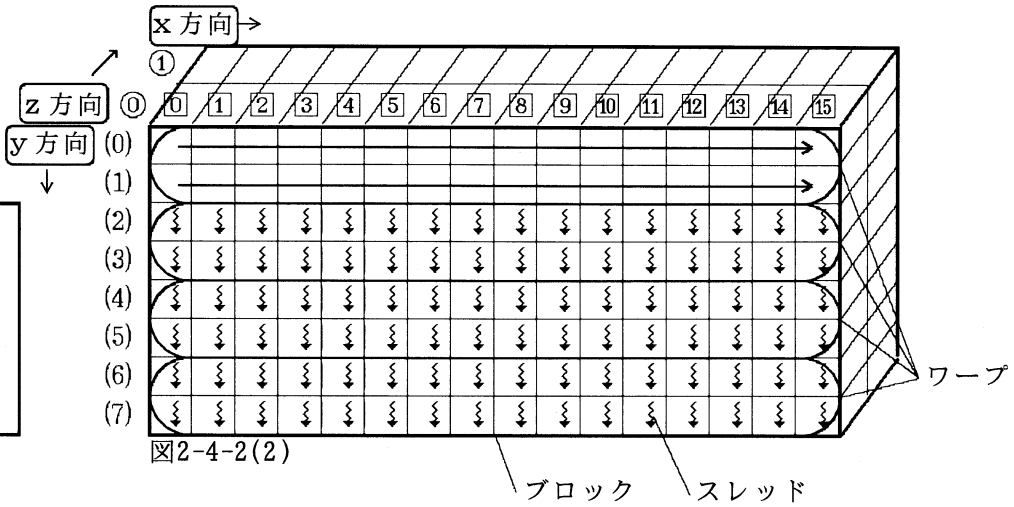


図2-4-2(2)

■ 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12×32=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6×64=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には、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)と(🟧,▯)を左のSMに、ブロック(▮,0)と(▮,▯)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(🟧,2),(▮,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(例えばメモリからのロード/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。

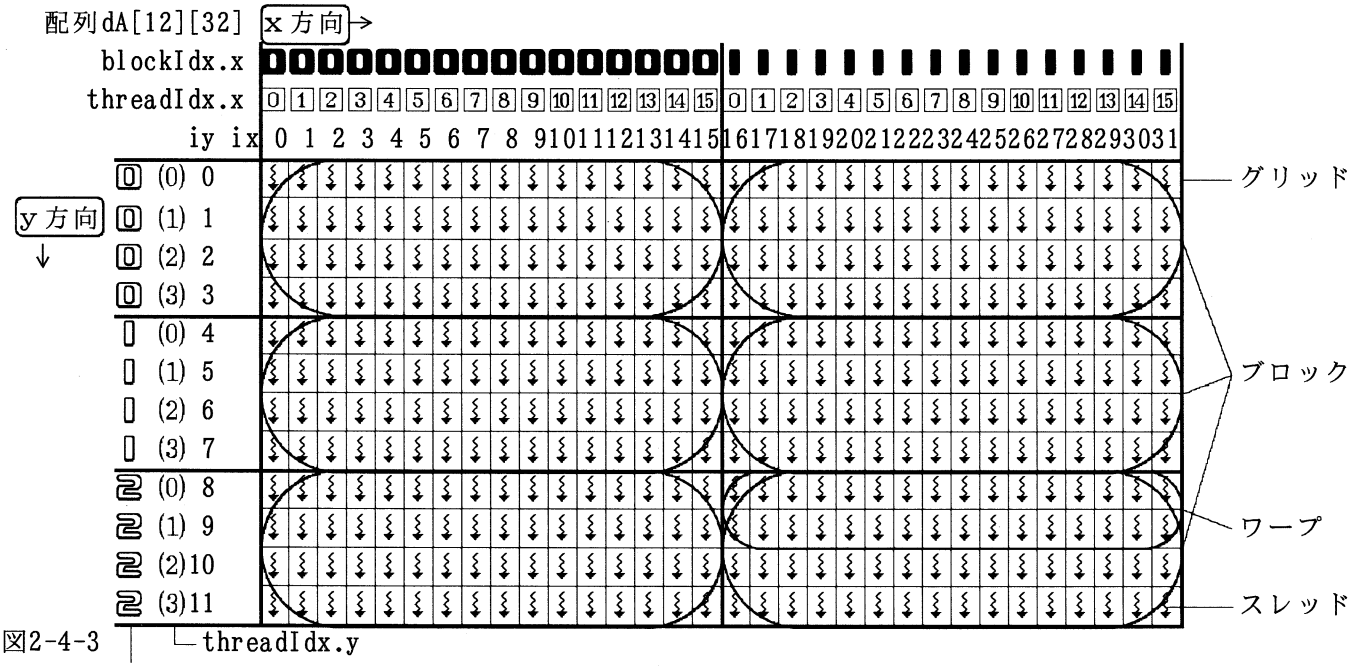


図2-4-3

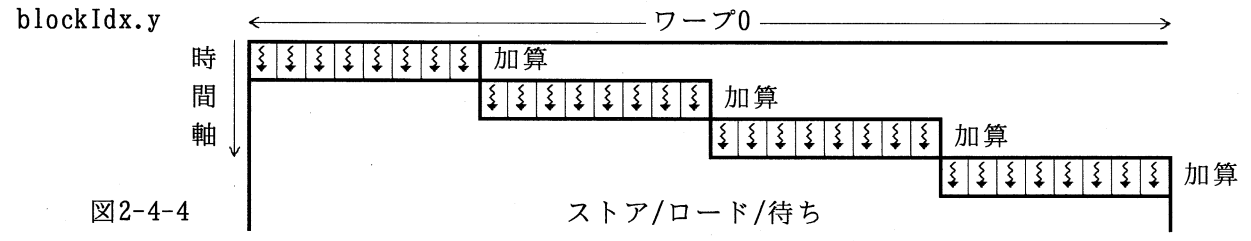


図2-4-4

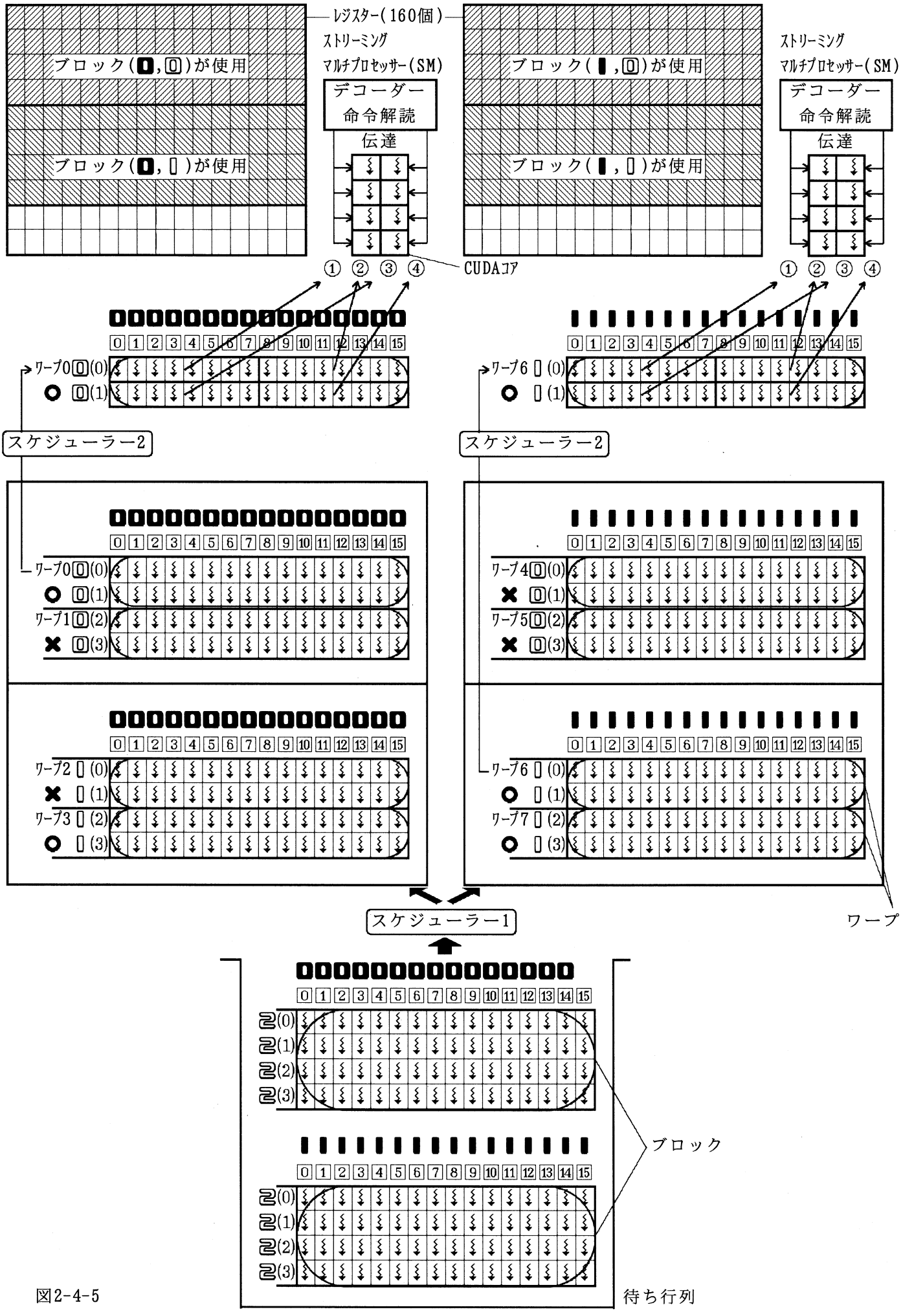

ブロック(0,0)が使用
ブロック(0,1)が使用
レジスター(160個)
ストリーミング
マルチプロセッサー(SM)
デコーダー
命令解読
伝達
ブロック(1,0)が使用
ブロック(1,1)が使用
ストリーミング
マルチプロセッサー(SM)
デコーダー
命令解読
伝達
① ② ③ ④
CUDAコア
① ② ③ ④
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
ワープ0 0(0)
0(1)
ワープ6 1(0)
1(1)
スケジューラー2
スケジューラー2
ワープ0 0(0)
0(1)
ワープ1 0(2)
0(3)
ワープ4 0(0)
0(1)
ワープ5 0(2)
0(3)
ワープ2 1(0)
1(1)
ワープ3 1(2)
1(3)
ワープ6 1(0)
1(1)
ワープ7 1(2)
1(3)
スケジューラー1
ワープ
2(0)
2(1)
2(2)
2(3)
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)。

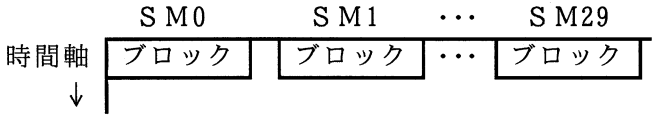


図2-5-1(1)

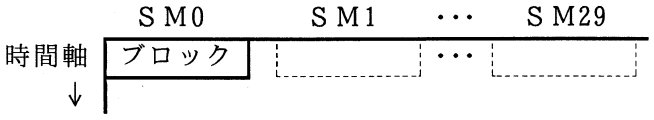


図2-5-1(2)

■ 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。

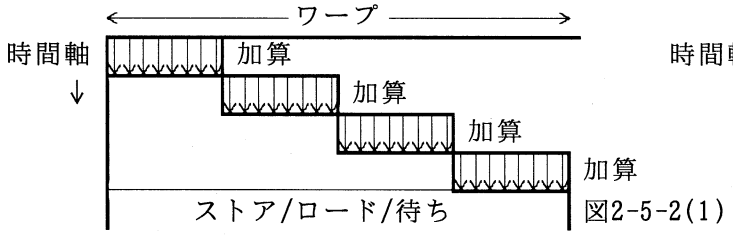


図2-5-2(1)

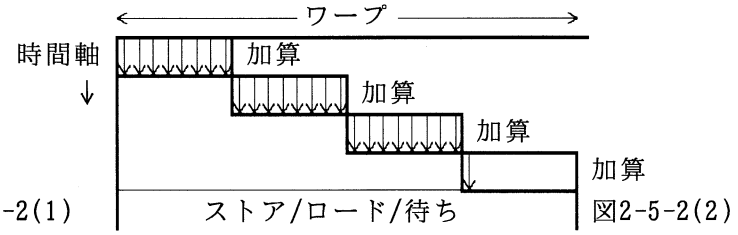


図2-5-2(2)

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　ｽﾚｯﾄﾞ数=32の場合のﾌﾞﾛｯｸ数(✖は誤り)

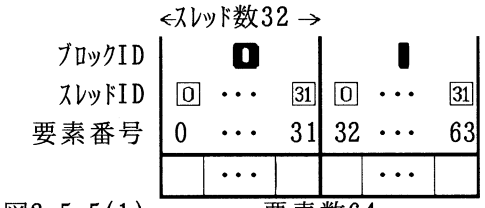


図2-5-5(1)　←――― 要素数64 ―――→

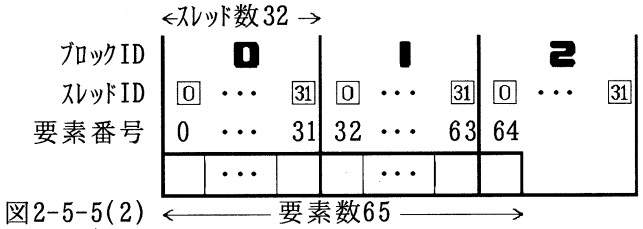


図2-5-5(2)　←――― 要素数65 ―――→

```
#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。

	ﾌﾞﾛｯｸ数	ﾌﾞﾛｯｸ内のｽﾚｯﾄﾞ数
(1)	15	32
(2)	30	16
(3)	20	24

	ﾌﾞﾛｯｸ数	ﾌﾞﾛｯｸ内のｽﾚｯﾄﾞ数
(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。

配列が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、処理する要素数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。

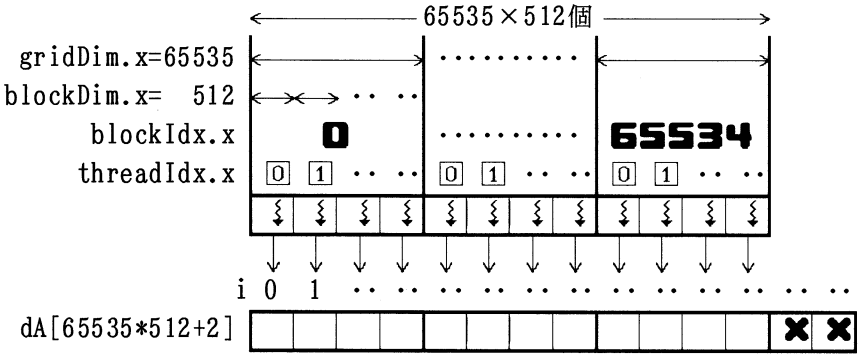


図2-6-1

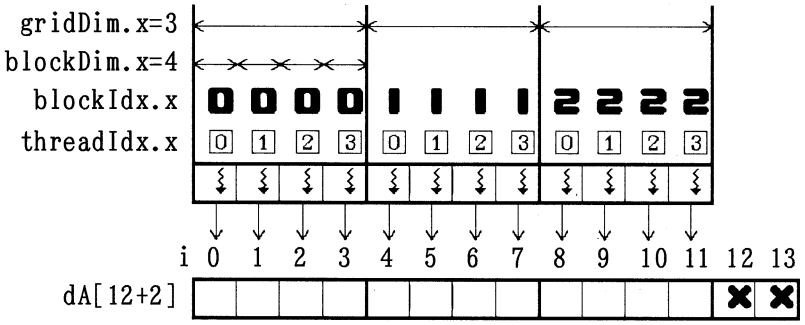


図2-6-2

【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。

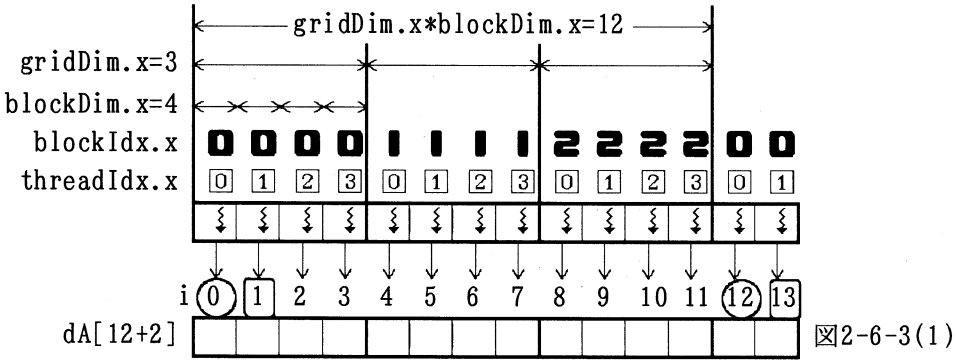


図2-6-3(1)

```
#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],[0])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。

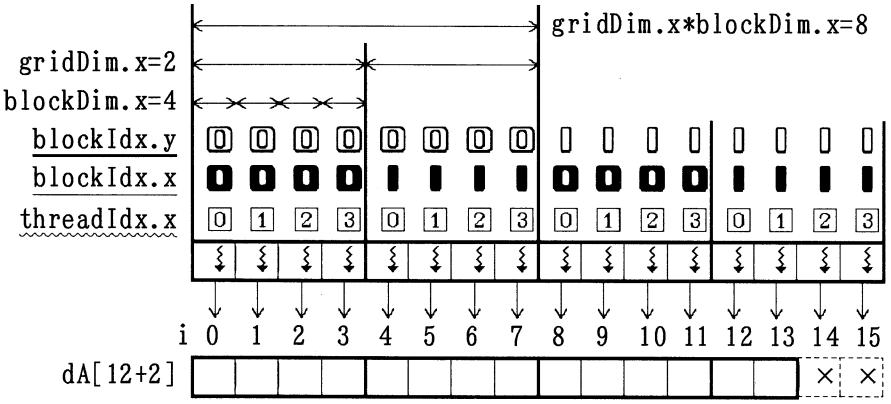


図2-6-4(1)

```
#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。

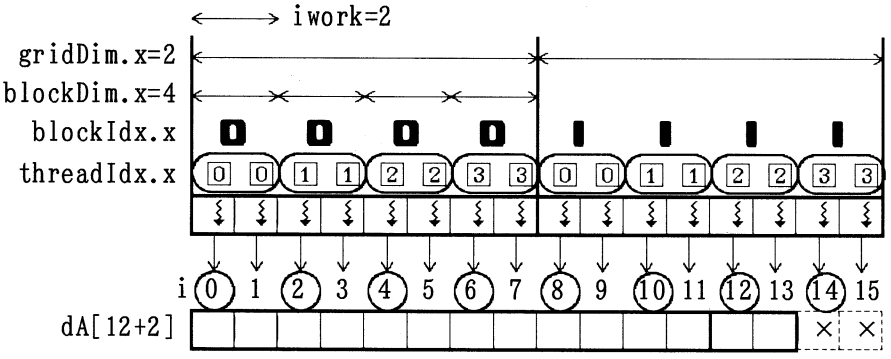


図2-6-5(1)

```
#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)

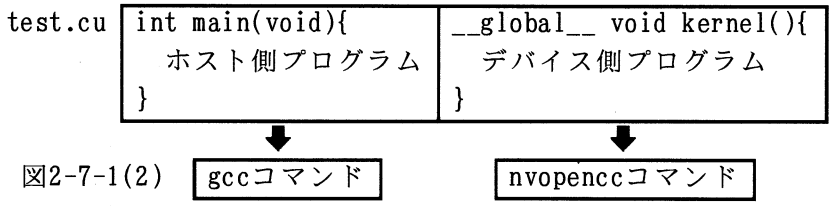


図2-7-1(2)

■ 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の本格的活用法」（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(大文字のｵｰとｾﾛ)」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、通常は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(大文字のｵｰとｾﾛ)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
//  -O0 (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を指定しないと誤動作します(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は、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(詳細は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

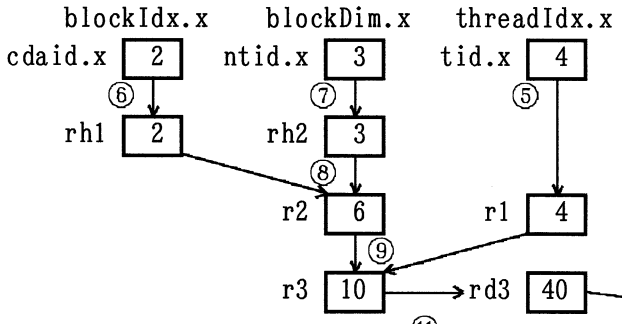
図2-7-9(3)の(6)は、アセンブラコード内で使用するレジスターを示します。u(u16など)は符号なし整数、fは実数、s(本例では未使用)は符号付き整数です。また16,32,64はビット数です。例えば(7)は、32ビット(4バイト)の実数(つまり単精度実数)のレジスターf1～f5を示します(ただし、アセンブラ命令内で全部使用されるとは限りません)。

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ﾊﾞｲﾄ目)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。




```
1 __global__ void kernel(float *dA){
2   int i = blockIdx.x*blockDim.x + threadIdx.x; [1]
      10↵      ↰2      ↰3       ↳4
3   dA[i+1] = dA[2] + 3.4f;                      [2]
4 }  11↵       ↰1.0
```

図2-7-9(1) kernel.cu

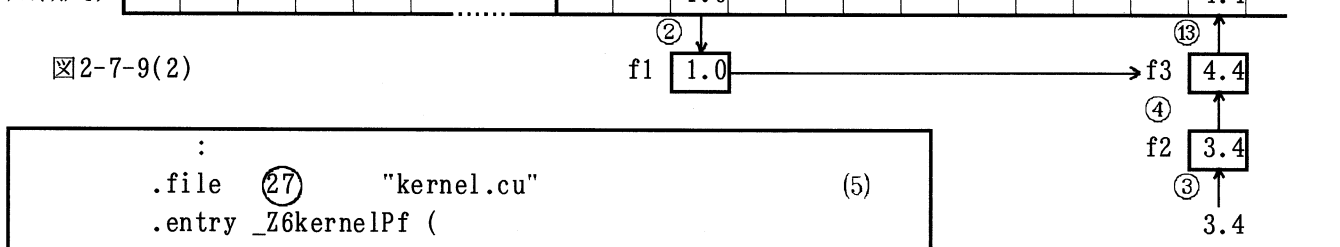


図2-7-9(2)


```
        :
        .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
```

図2-7-9(3)

【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
```

【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=10×4)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。効率のよい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)から構成されています。

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。

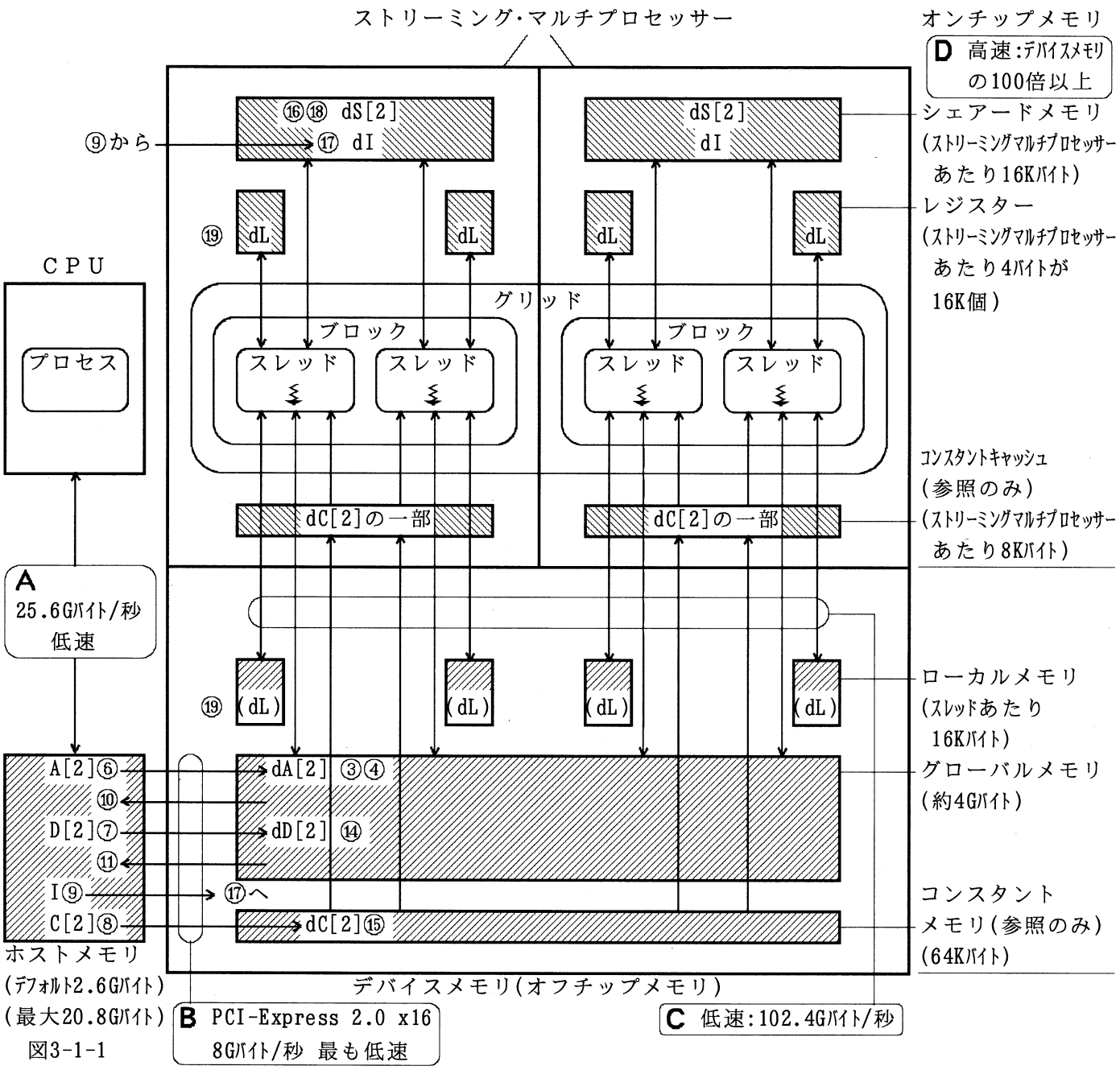


図3-1-1

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4G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

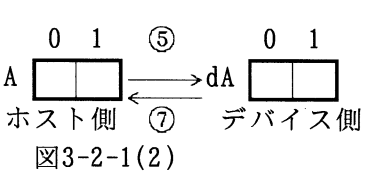


図3-2-1(2)

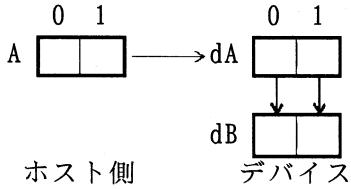


図3-2-1(3)

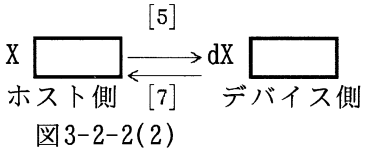


図3-2-2(2)

```
float *A;                                  ⑨
A = (float*)malloc(size);                  ⑨
```

```
cudaMalloc((void**)&dA,2*sizeof(float));                ⑩
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。

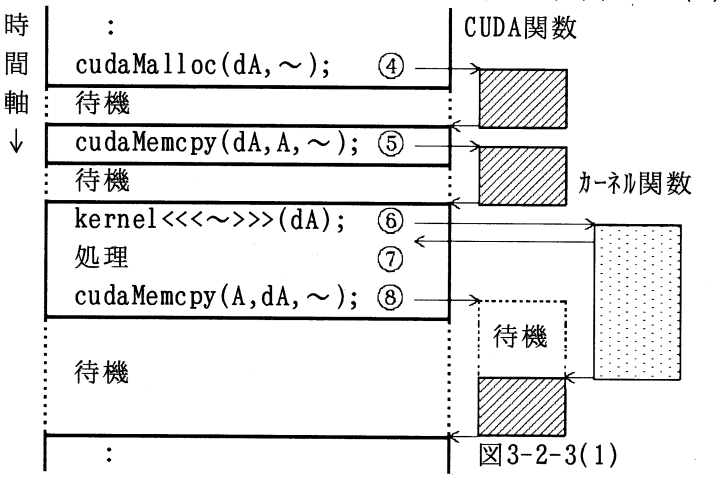


図3-2-3(1)

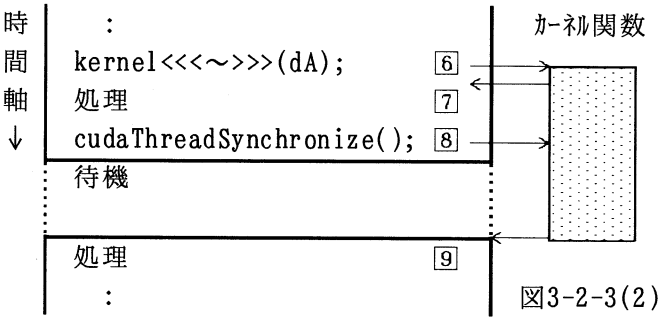


図3-2-3(2)

```
    :
for(itime=0;itime<100;itime++){
  kernel<<<～>>>(dA);
}
    :
```
図3-2-4(1)

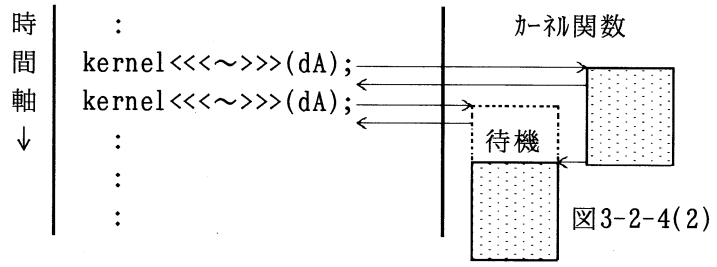


図3-2-4(2)

```
    :
for(itime=0;itime<100;itime++){
  kernel<<<～>>>(dA);
  cudaThreadSynchronize();
}
    :
```
図3-2-5(1)

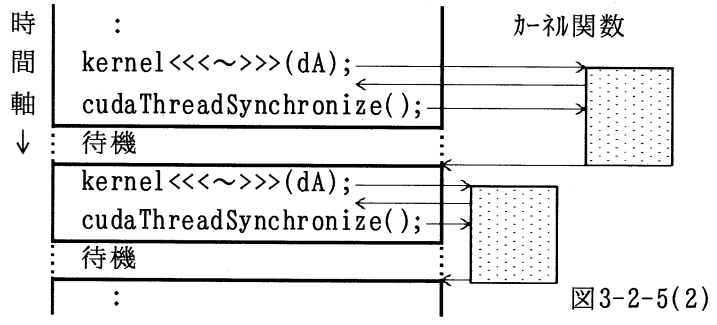


図3-2-5(2)

■ 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ﾊﾞｲﾄ,32ﾊﾞｲﾄ,64ﾊﾞｲﾄ,･･･)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。

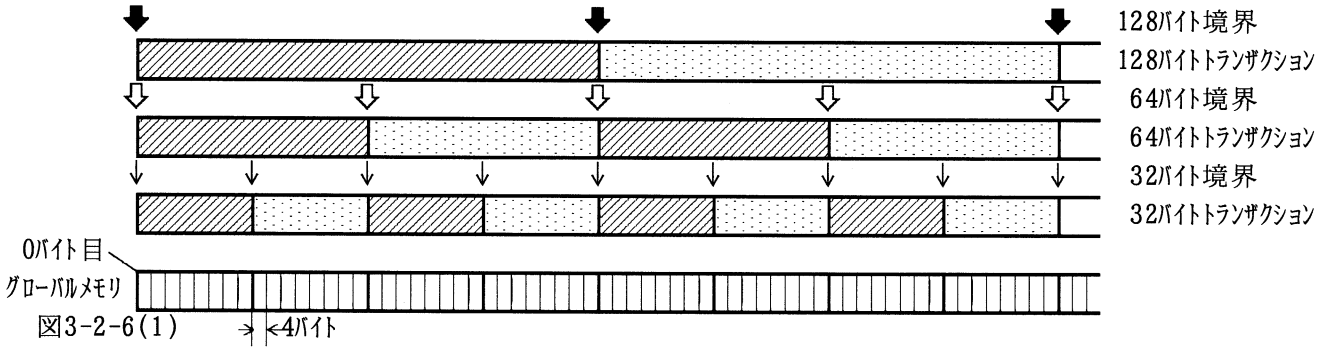


図3-2-6(1)

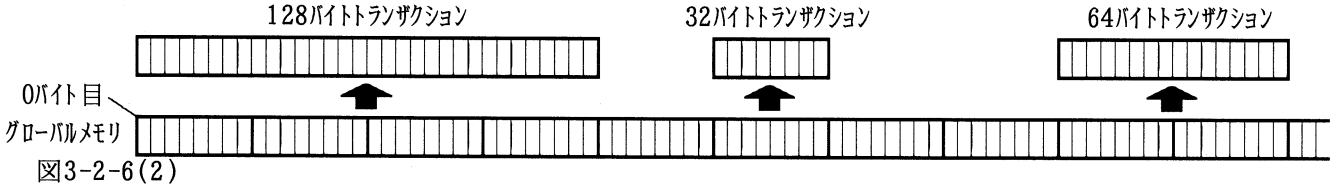


図3-2-6(2)

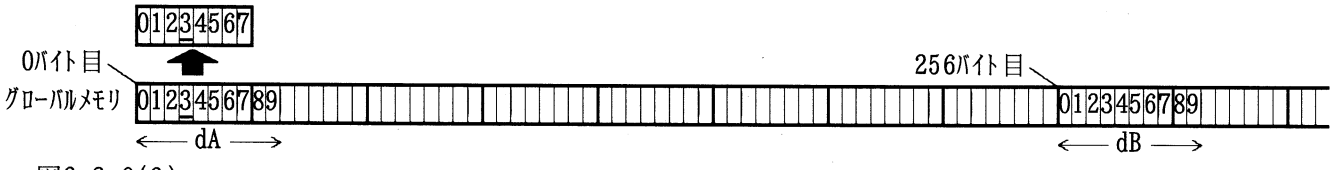


図3-2-6(3)

```
__global__ void kernel(float *dA){
  dA[3] = dA[3] + 1.0f;          ①
}
```
図3-2-7

```
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
    :
```

■ 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。

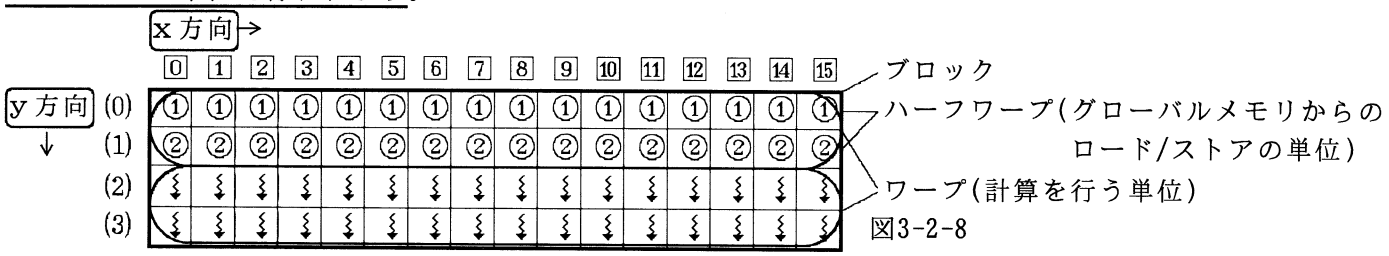


図3-2-8

図3-2-9(1)のプログラムでは、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、自分の担当する要素を、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
}
```
図3-2-9(1)

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

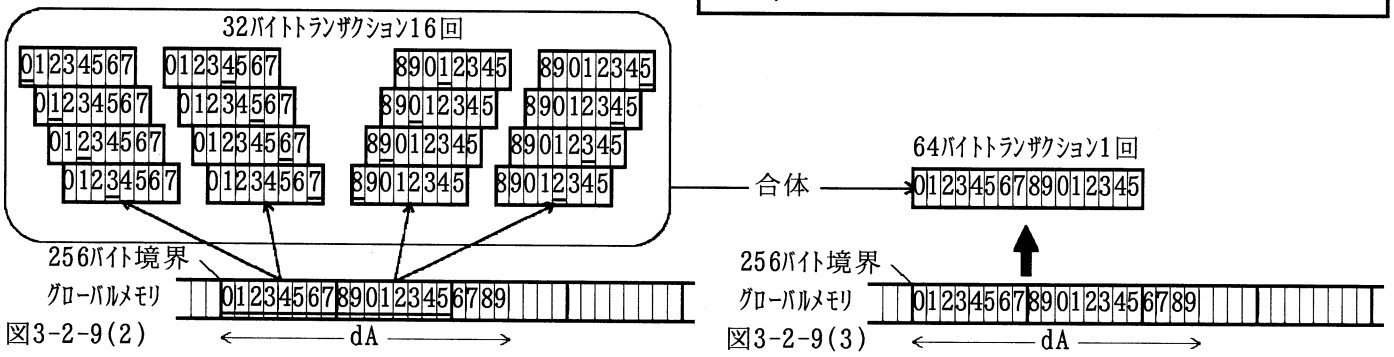


図3-2-9(2)　図3-2-9(3)

■ 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。

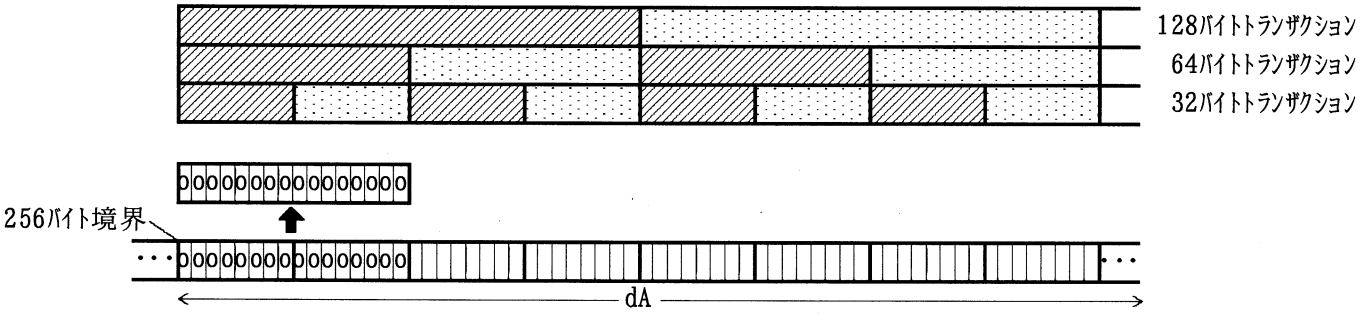
stride=2,3,4,16,32の場合、前述のプロファイラーを使用して調べたところ、ロード/ストアのトランザクションは、図3-2-11の(2)～(6)のようになりました。図から分かるように、ストライドが長くなると、トランザクションの量や回数が多くなり、コアレスアクセスの効率が悪くなります((6)は全くコアレスアクセスされていません)。可能であれば、なるべくストライドが短くなるようにして下さい。

```
__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
}
```

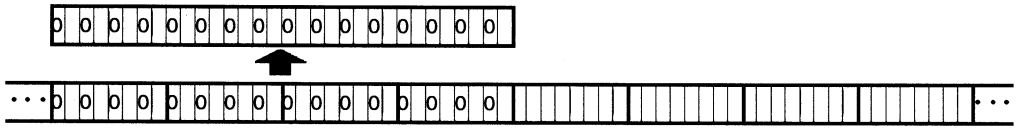
```
int main(void){
  float *dA;
    :
  kernel<<<1,16>>>(dA);
    :
```

図3-2-10

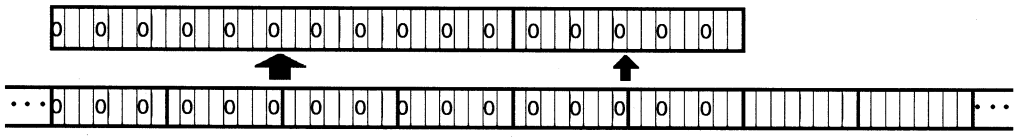
(1) stride=1の場合：64バイトトランザクション1回 ○



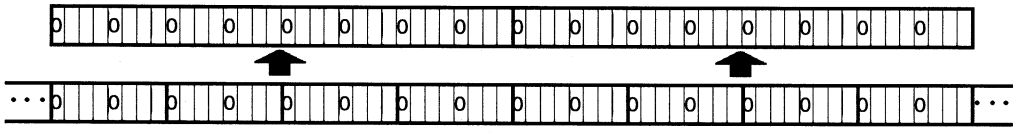
(2) stride=2の場合：128バイトトランザクション1回 ×



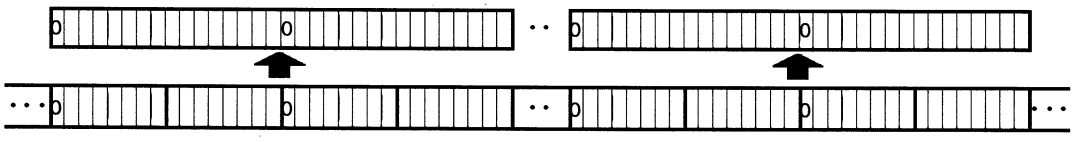
(3) stride=3の場合：128バイトトランザクション1回+64バイトトランザクション1回 ×



(4) stride=4の場合：128バイトトランザクション2回 ×



(5) stride=16の場合：128バイトトランザクション8回 ×



(6) stride=32の場合：32バイトトランザクション16回 ×

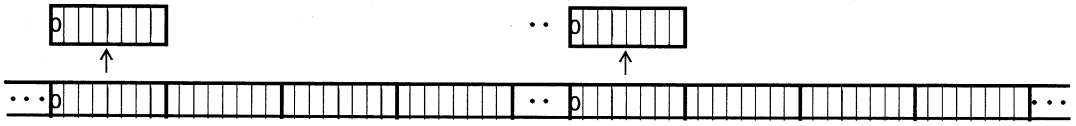


図3-2-11

■ 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

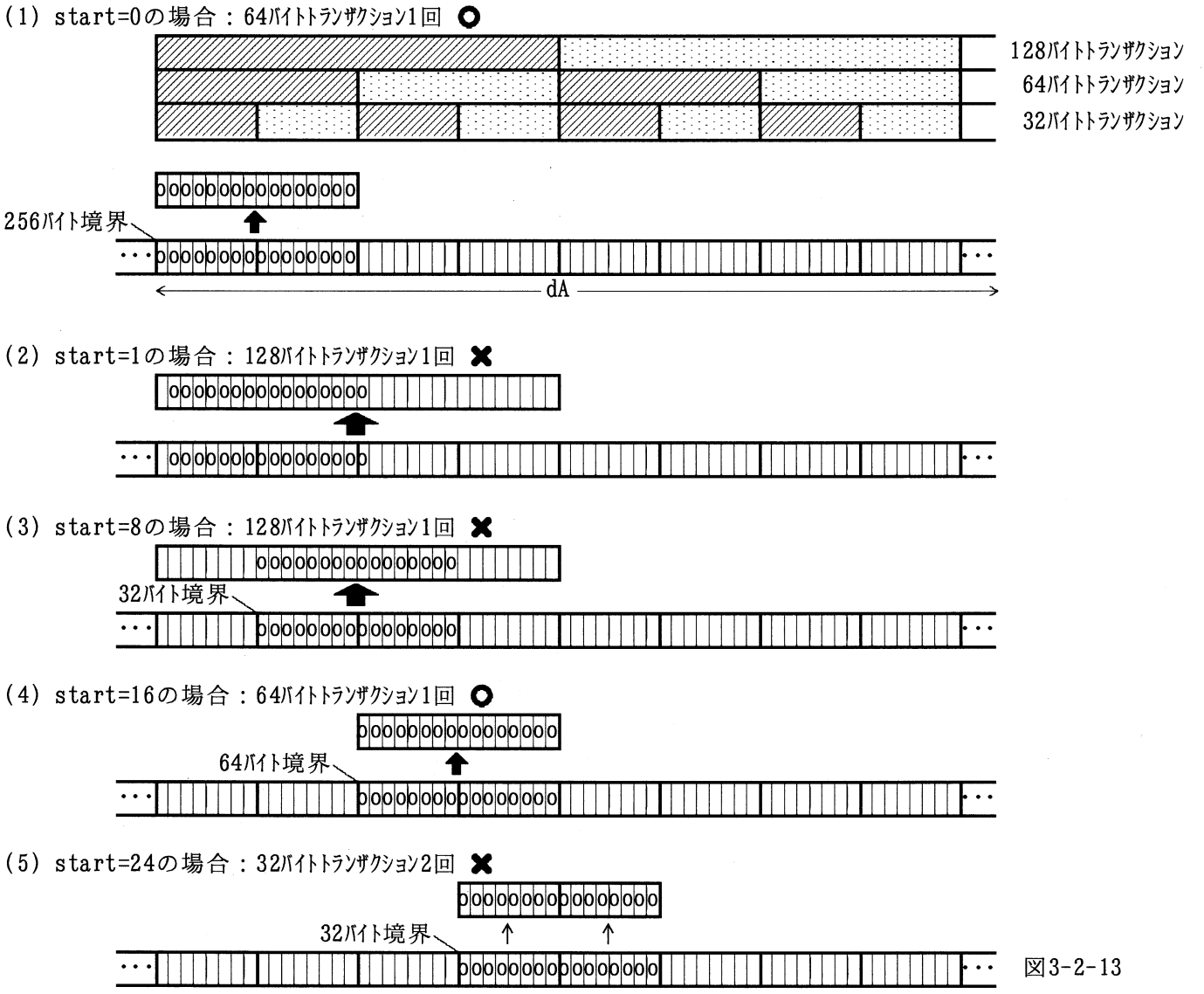


図3-2-13

■ 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ﾊﾞｲﾄ)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{                                   ⑦
  float a;
  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ﾊﾞｲﾄのﾒﾝﾊﾞｰが2個なので
  cudaMemcpy(dAB,AB,16*8,～HostToDevice)         ⑪
  kernel<<<1,16>>>(dAB);
    :
```

図3-2-14(2)

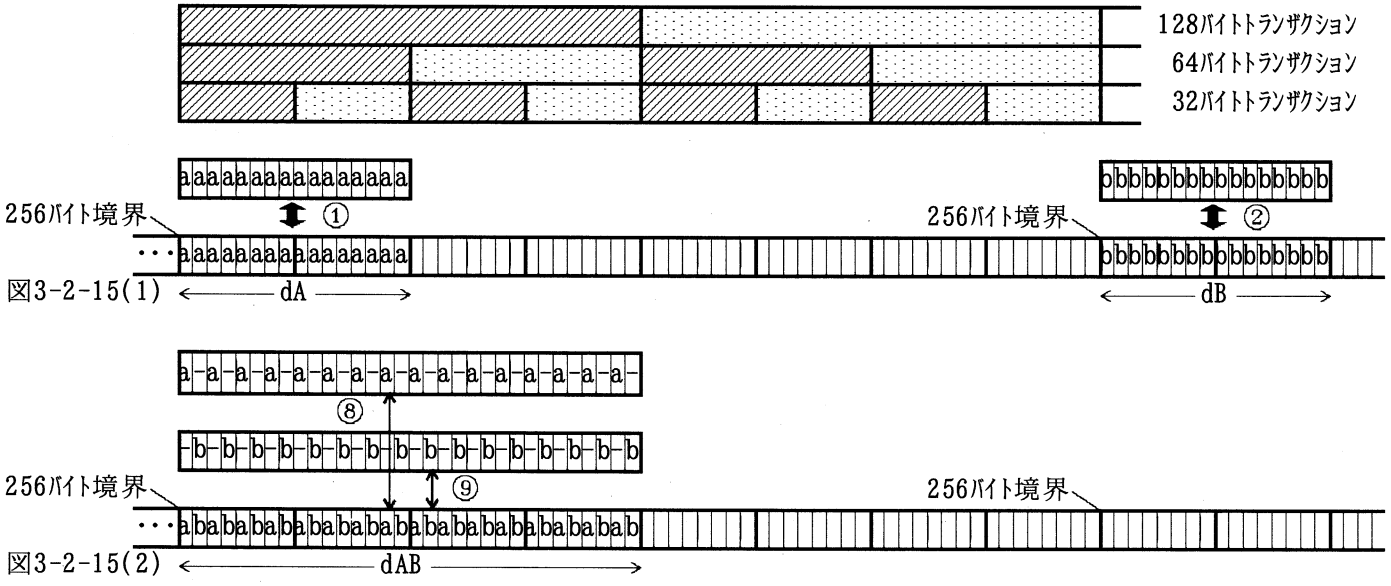


図3-2-15(1)

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

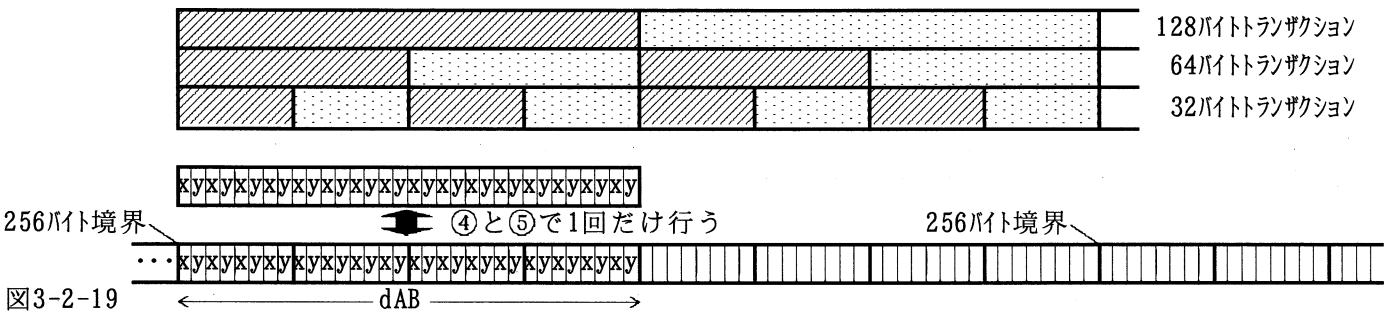


図3-2-19

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(iy,ix)]を、2次元配列dA[iy][ix]で表すことにします。

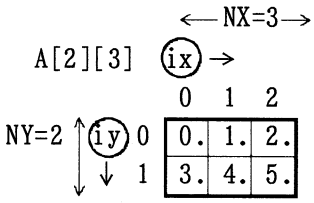


図3-3-1(1)

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)

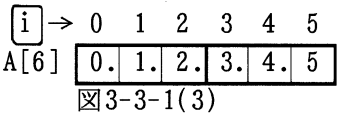


図3-3-1(3)

```
#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。

図3-3-4(1) ○

```
#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(2) ✖

```
#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
   :
```

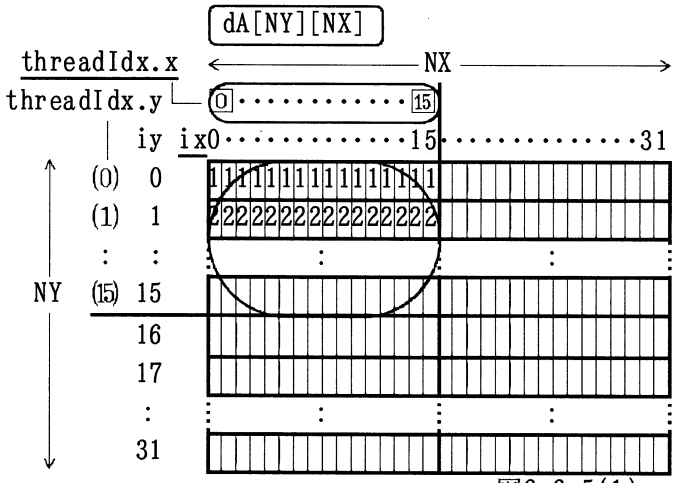


図3-3-5(1)

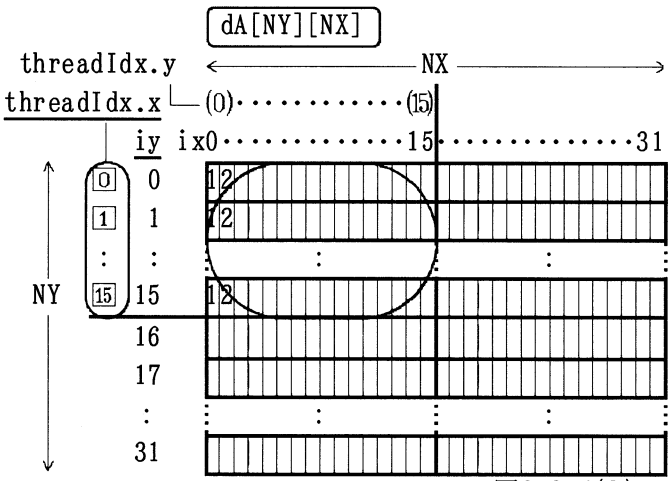


図3-3-5(2)

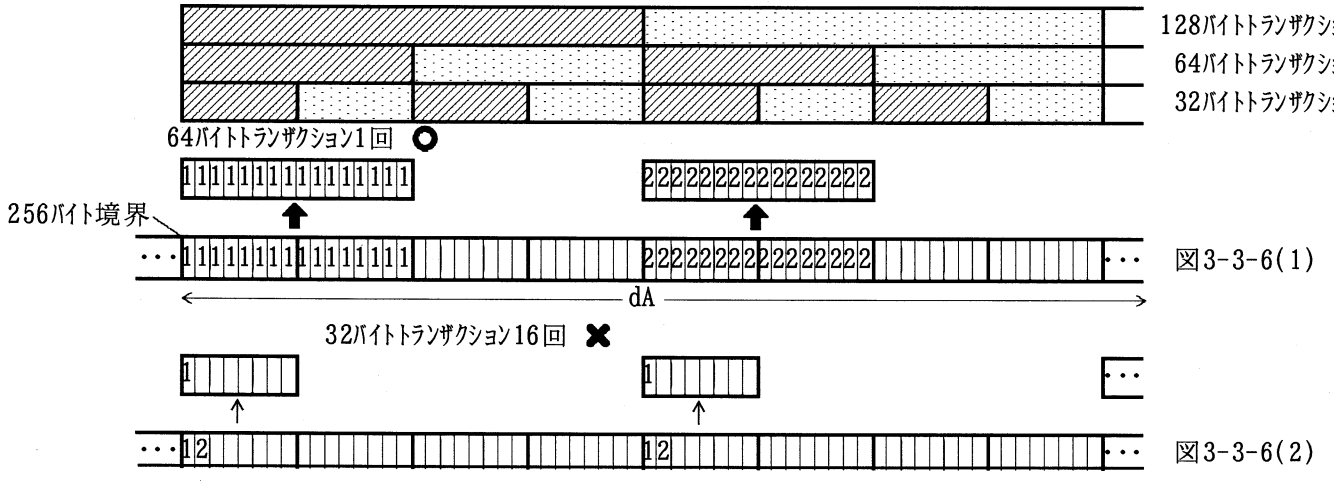


図3-3-6(1)

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)

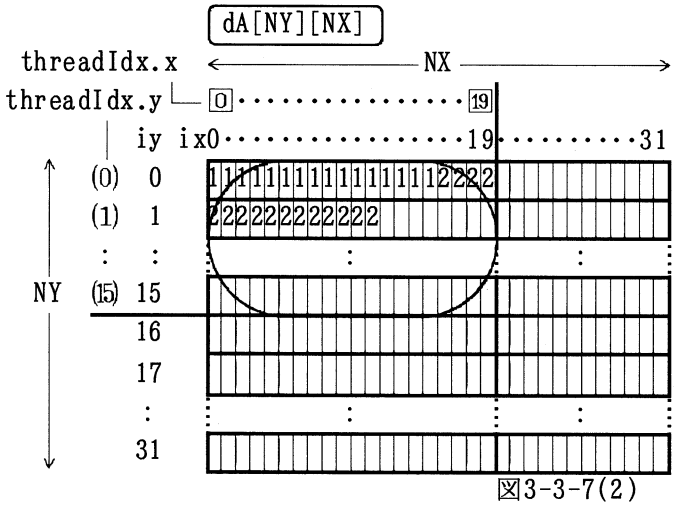


図3-3-7(2)

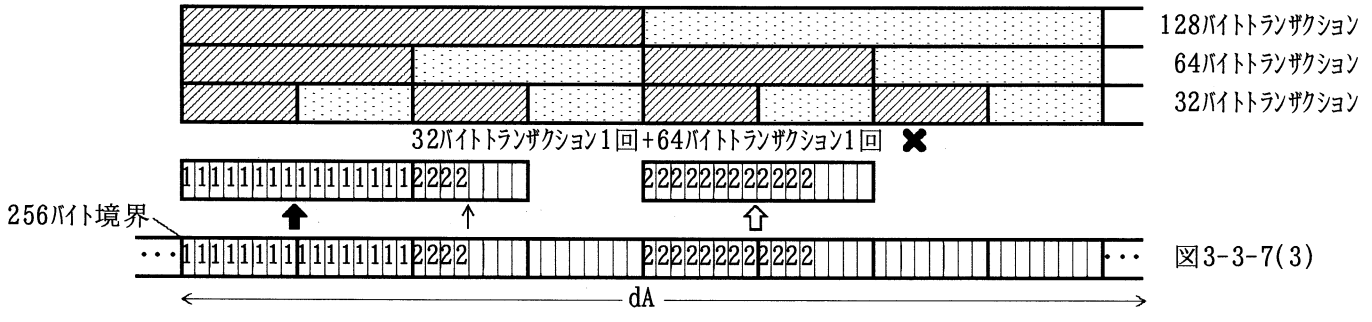


図3-3-7(3)

■ 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)

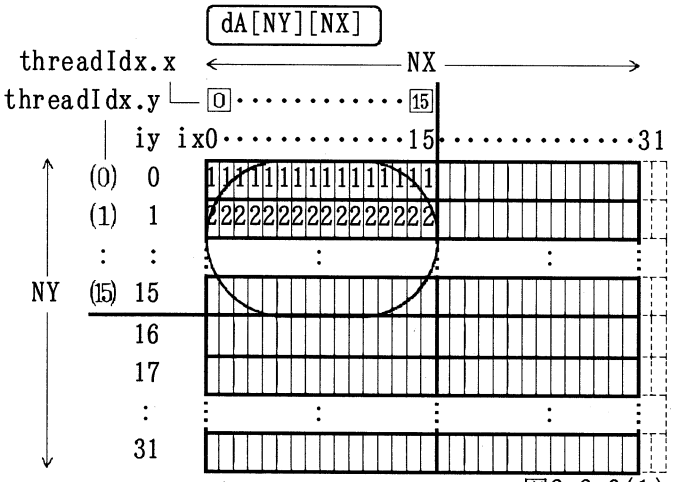


図3-3-9(1)

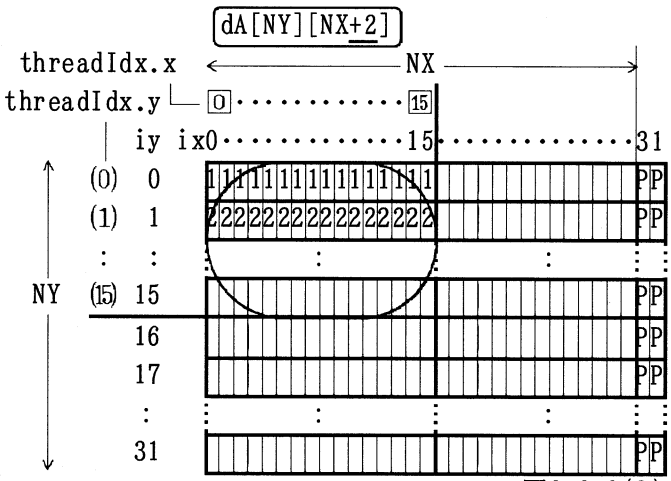


図3-3-9(2)

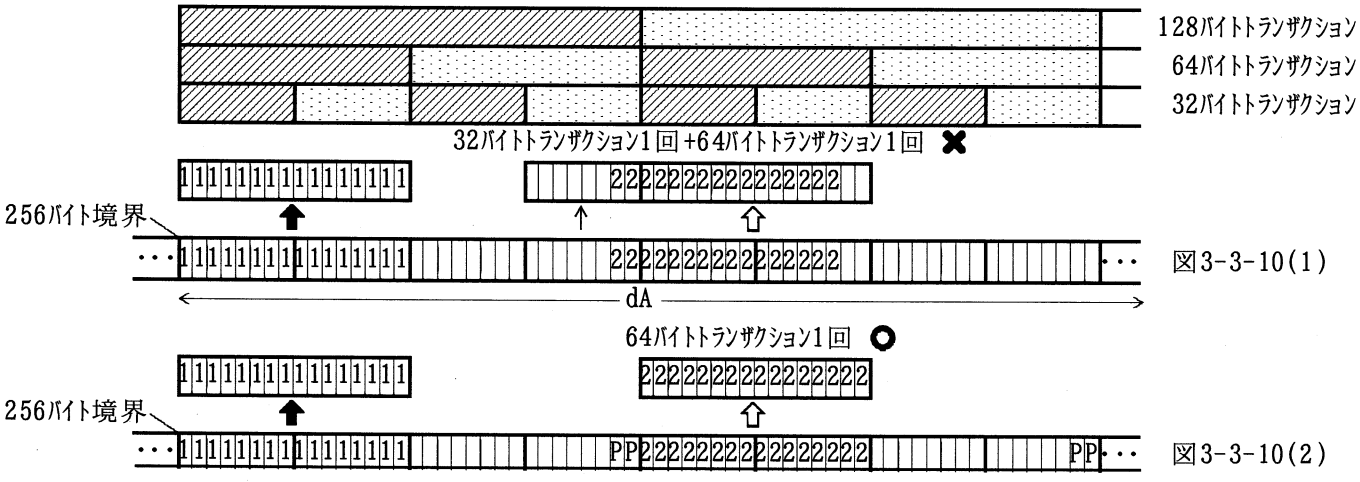


図3-3-10(1)

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。

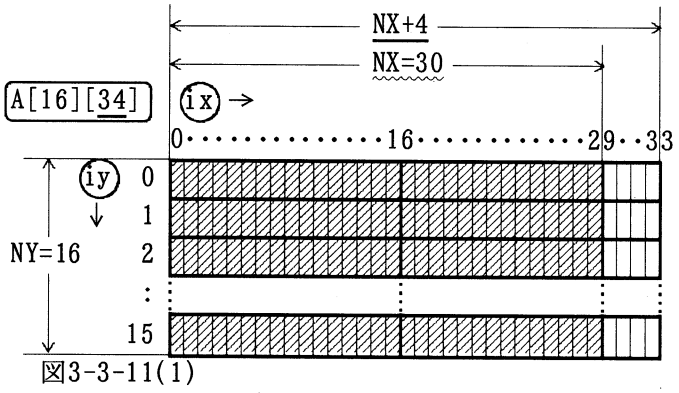


図3-3-11(1)

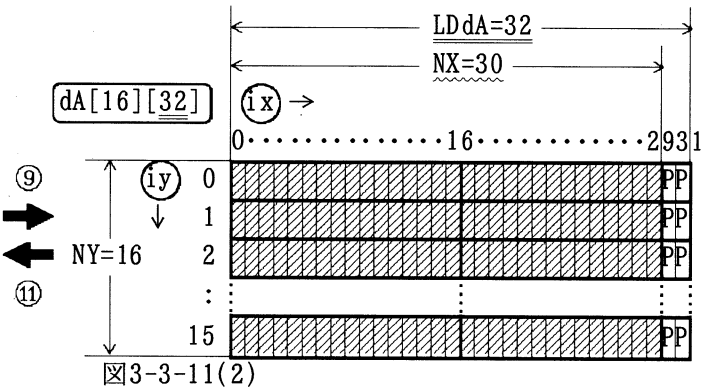


図3-3-11(2)

```
#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
  }
}
```
図3-3-12

```
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);
    :
```

■ 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、図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。

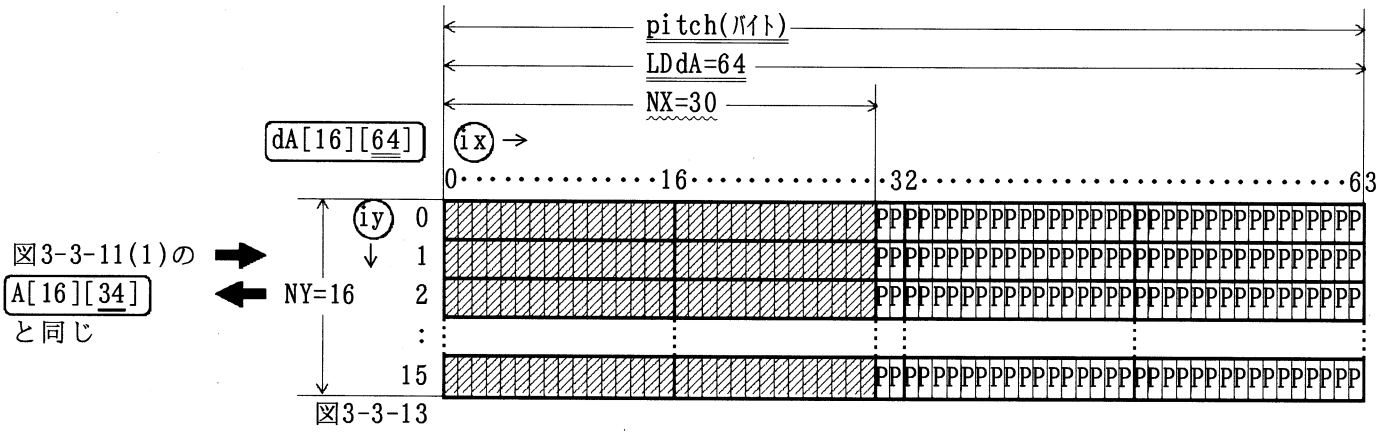


図3-3-13

```
#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;
  }
}
```
図3-3-14

```
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);
    :
```

■ 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。

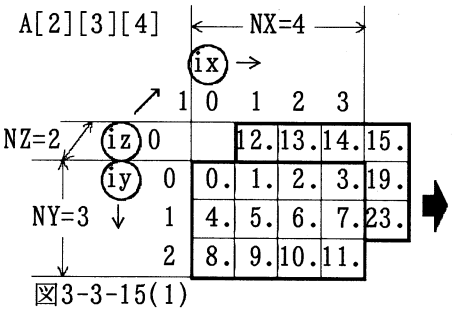


図3-3-15(1)

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

図3-3-15(2)

i →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A[24]	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.	17.	18.	19.	20.	21.	22.	23.

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。

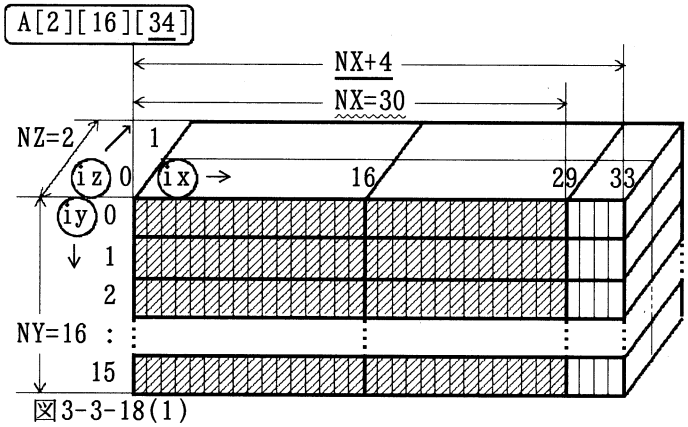


図3-3-18(1)

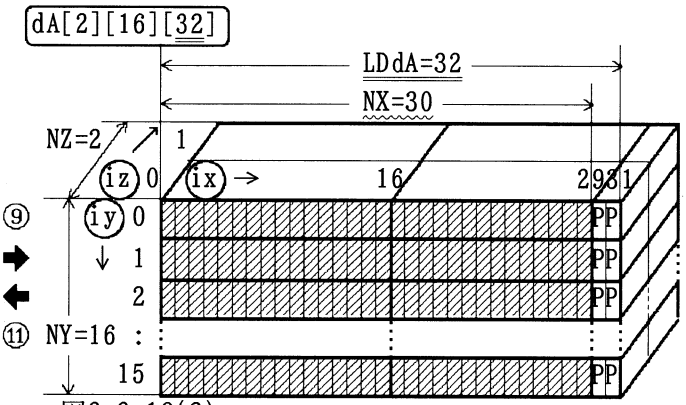


図3-3-18(2)

```
#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
  }
}
```

図3-3-19

```
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);
    :
```

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cuda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

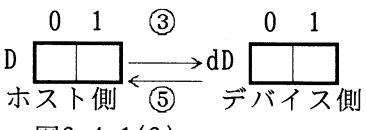


図3-4-1(2)

```
__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

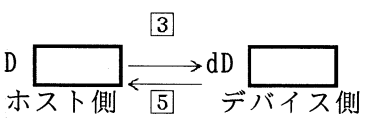


図3-4-2(2)

■ 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)

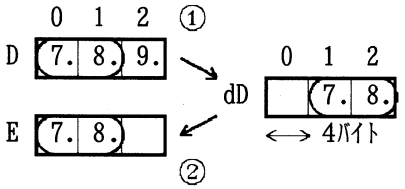


図3-4-3(2)

■ 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)

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

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

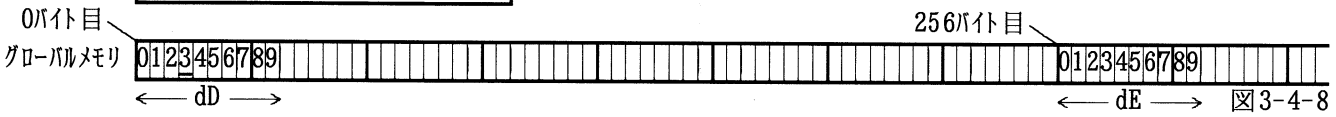


図3-4-8

●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。

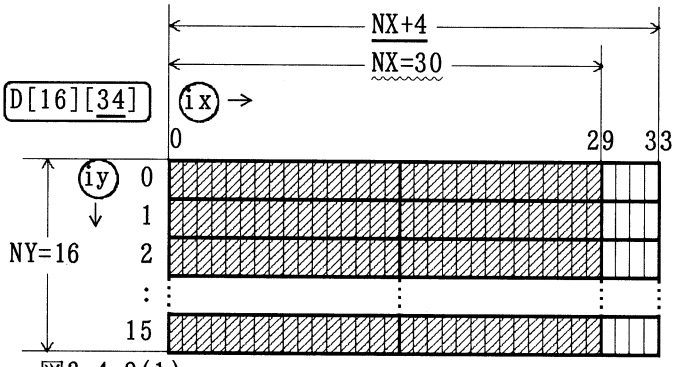


図3-4-9(1)

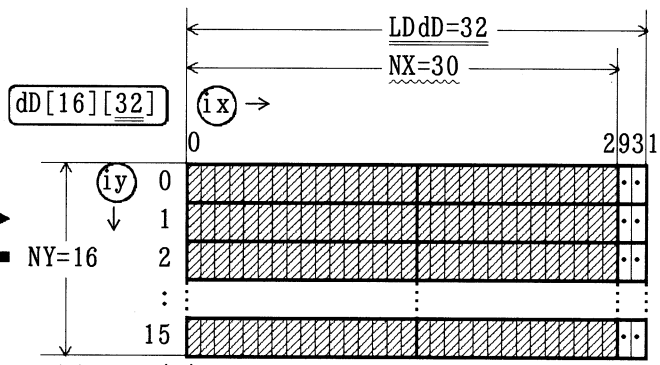


図3-4-9(2)

```
__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

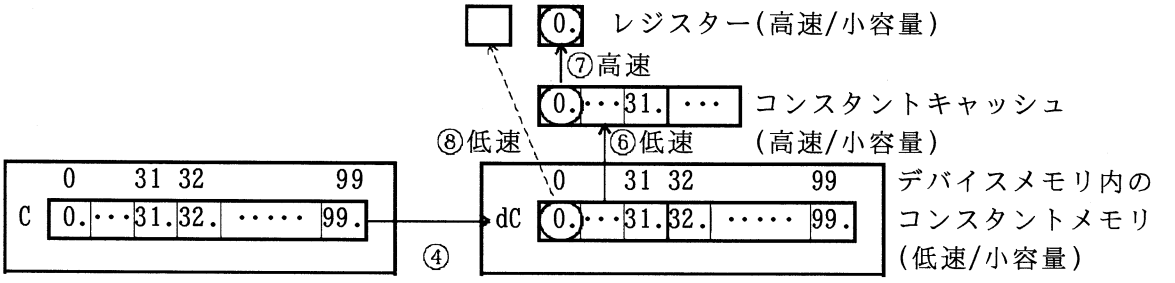


図3-5-2

■ 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) ○

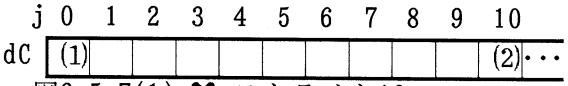


図3-5-7(1) ✖ ストライド10

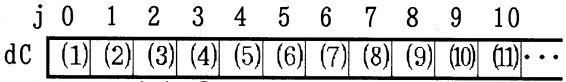


図3-5-7(2) ○ ストライド1

●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) ○

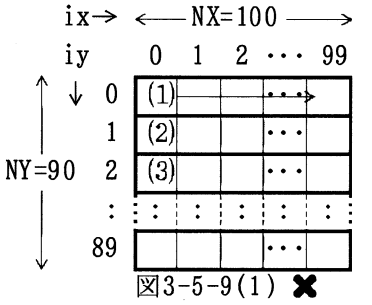


図3-5-9(1) ✖

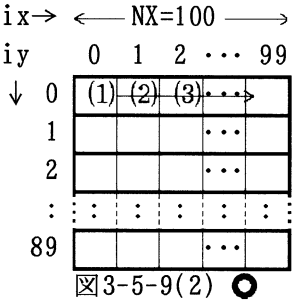


図3-5-9(2) ○

●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;
}
```

図3-5-10(1) ✖

```
#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
    :
```

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以上ではシェアード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

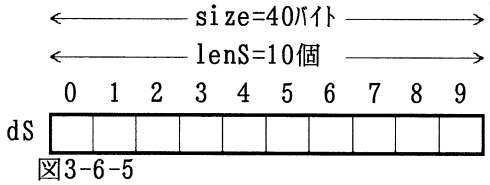


図3-6-5

```
#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。

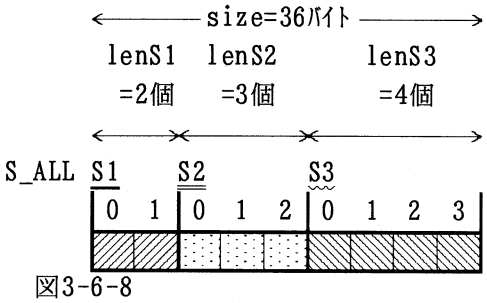


図3-6-8

```
__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,･･･)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,･･･)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。

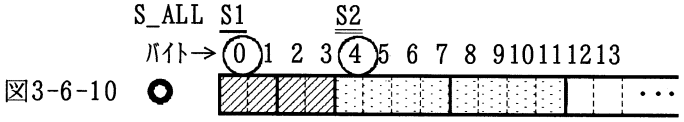


図3-6-10 ○

short S1[2]
float S2[2]

```
short *S1 = (short*)S_ALL;
float *S2 = (float*)S1[2];
```

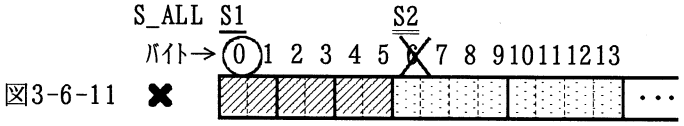


図3-6-11 ✖

short S1[3]
float S2[2]

```
short *S1 = (short*)S_ALL;
float *S2 = (float*)S1[3];
```

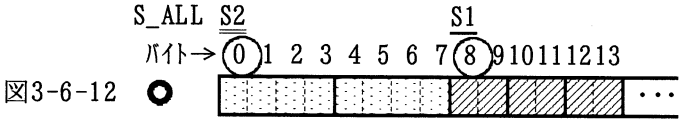


図3-6-12 ○

float S2[2]
short S1[3]

```
float *S2 = (float*)S_ALL;
short *S1 = (short*)S2[2]
```

■ 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)を図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

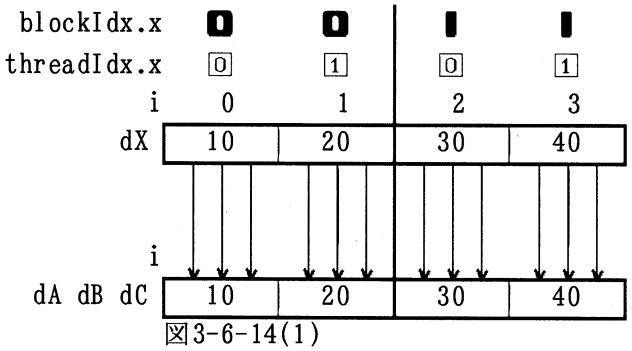


図3-6-14(1)

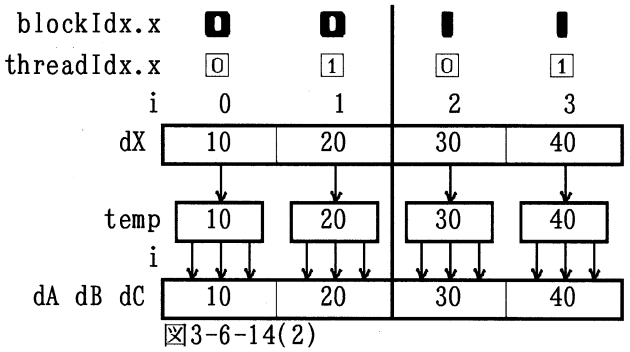


図3-6-14(2)

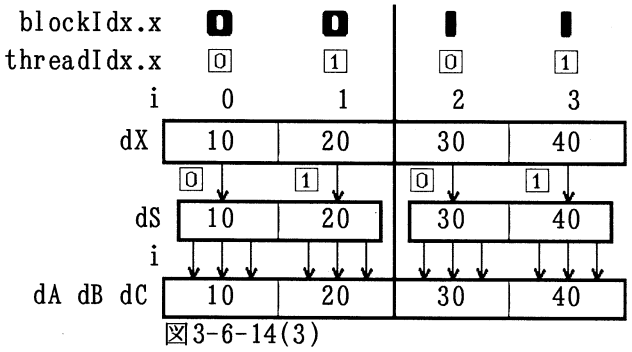


図3-6-14(3)

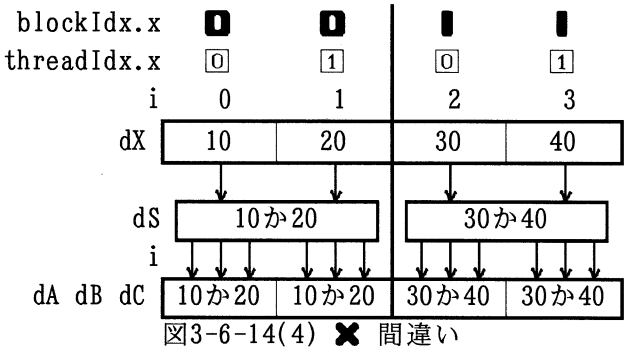


図3-6-14(4) ✖ 間違い

■ 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回行なっており、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、図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)

```
__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)

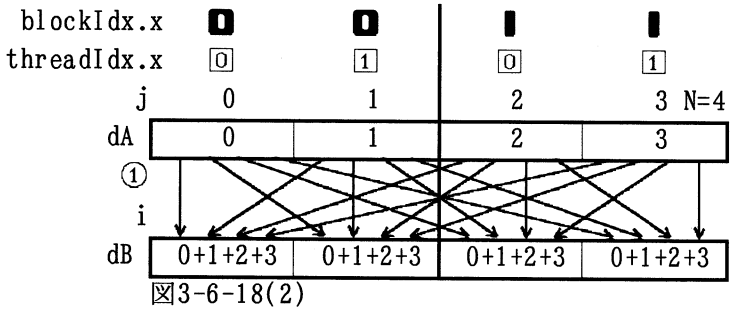


図3-6-18(2)

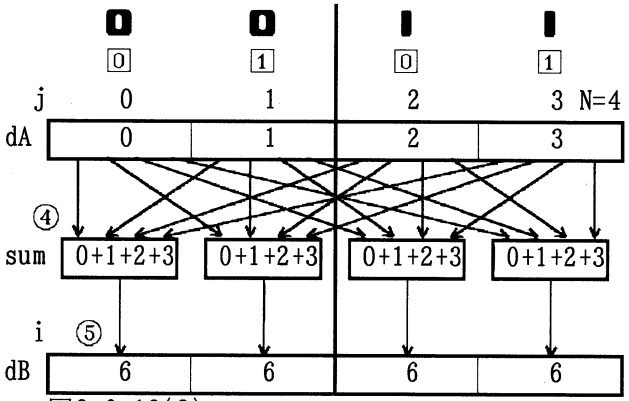


図3-6-19(2)

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)

```
__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++){                ↑
        sum = sum + dS[jj];                        ⑱
      }                                            ↓
    }
    __syncthreads();
  }
  if(i<N) dB[i] = sum;                             ⑲
}
```

図3-6-21(1)

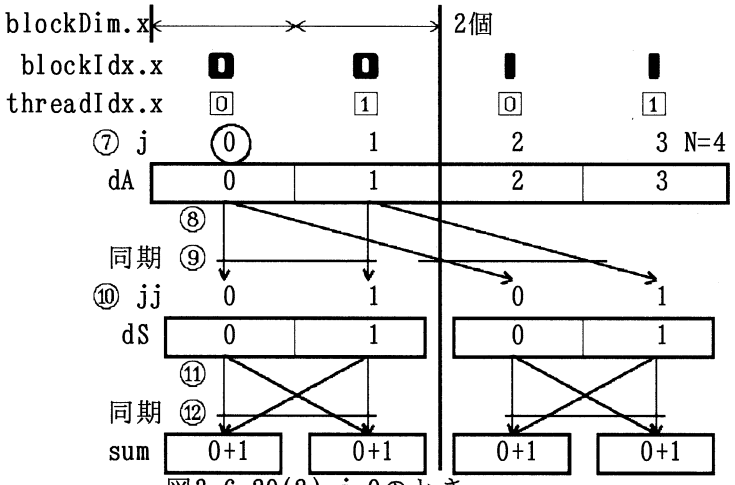


図3-6-20(2) j=0のとき

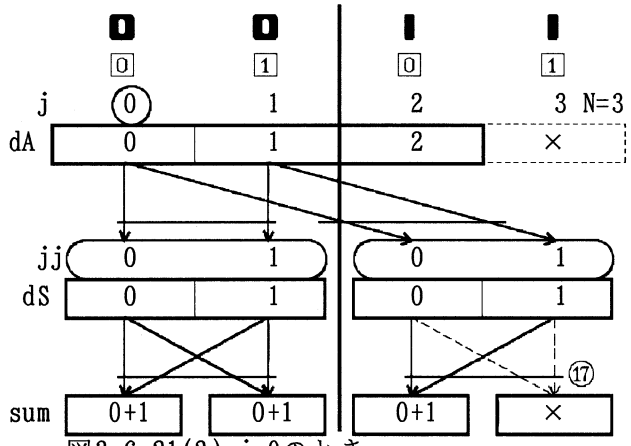


図3-6-21(2) j=0のとき

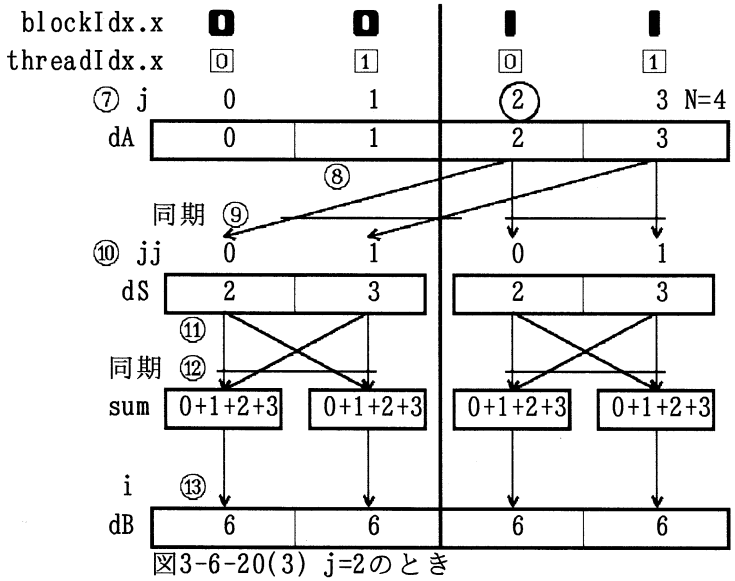


図3-6-20(3) j=2のとき

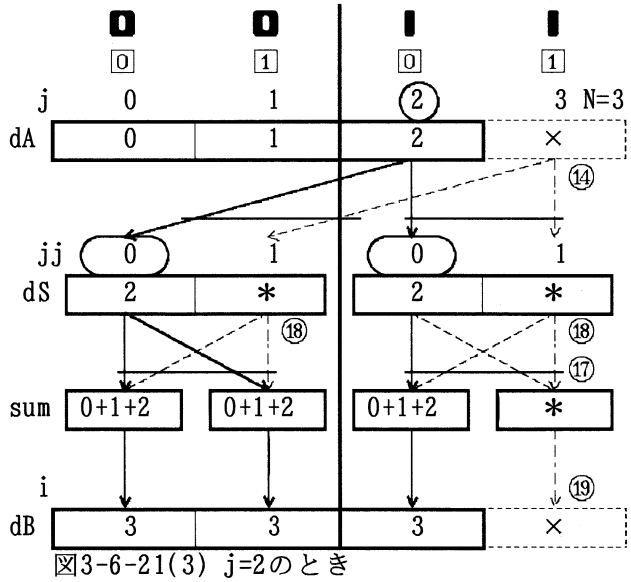


図3-6-21(3) j=2のとき

■ 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。

←→ ストライド1

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 S[0]	1 S[1]	2 S[2]	3 S[3]	4 S[4]	5 S[5]	6 S[6]	7 S[7]	8 S[8]	9 S[9]	10 S[10]	11 S[11]	12 S[12]	13 S[13]	14 S[14]	15 S[15]
S[16]	S[17]	S[18]	S[19]	S[20]	S[21]	S[22]	S[23]	S[24]	S[25]	S[26]	S[27]	S[28]	S[29]	S[30]	S[31]
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:

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。

←→×←→ ストライド2

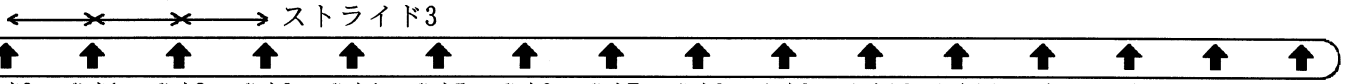
バンク0	バンク1	バンク2	バンク3	バンク4	バンク5	バンク6	バンク7	バンク8	バンク9	バンク10	バンク11	バンク12	バンク13	バンク14	バンク15
0 S[0]	S[1]	1 S[2]	S[3]	2 S[4]	S[5]	3 S[6]	S[7]	4 S[8]	S[9]	5 S[10]	S[11]	6 S[12]	S[13]	7 S[14]	S[15]
8 S[16]	S[17]	9 S[18]	S[19]	10 S[20]	S[21]	11 S[22]	S[23]	12 S[24]	S[25]	13 S[26]	S[27]	14 S[28]	S[29]	15 S[30]	S[31]

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。

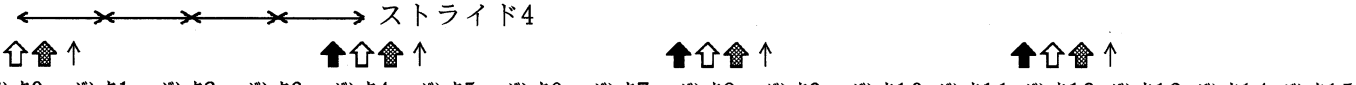
●stride=3(または奇数)：ストライドが奇数の場合、バンクコンフリクトは発生しません。



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 S[0]	S[1]	S[2]	1 S[3]	S[4]	S[5]	2 S[6]	S[7]	S[8]	3 S[9]	S[10]	S[11]	4 S[12]	S[13]	S[14]	5 S[15]
S[16]	S[17]	6 S[18]	S[19]	S[20]	7 S[21]	S[22]	S[23]	8 S[24]	S[25]	S[26]	9 S[27]	S[28]	S[29]	10 S[30]	S[31]
S[32]	11 S[33]	S[34]	S[35]	12 S[36]	S[37]	S[38]	13 S[39]	S[40]	S[41]	14 S[42]	S[43]	S[44]	15 S[45]	S[46]	S[47]

図3-6-24(1)

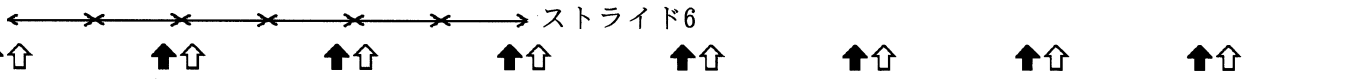
●stride=4：4ウェイ・バンクコンフリクトになります。



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 S[0]	S[1]	S[2]	S[3]	1 S[4]	S[5]	S[6]	S[7]	2 S[8]	S[9]	S[10]	S[11]	3 S[12]	S[13]	S[14]	S[15]
4 S[16]	S[17]	S[18]	S[19]	5 S[20]	S[21]	S[22]	S[23]	6 S[24]	S[25]	S[26]	S[27]	7 S[28]	S[29]	S[30]	S[31]
8 S[32]	S[33]	S[34]	S[35]	9 S[36]	S[37]	S[38]	S[39]	10 S[40]	S[41]	S[42]	S[43]	11 S[44]	S[45]	S[46]	S[47]
12 S[48]	S[49]	S[50]	S[51]	13 S[52]	S[53]	S[54]	S[55]	14 S[56]	S[57]	S[58]	S[59]	15 S[60]	S[61]	S[62]	S[63]

図3-6-24(2)

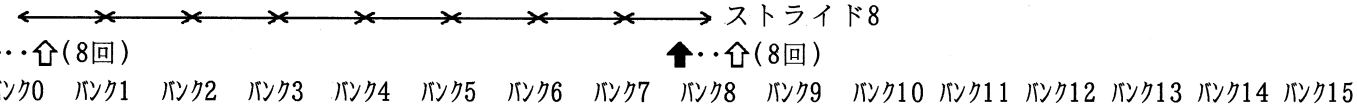
●stride=6：6ウェイでなく、2ウェイ・バンクコンフリクトになります。



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 S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	1 S[6]	S[7]	S[8]	S[9]	S[10]	S[11]	2 S[12]	S[13]	S[14]	S[15]
S[16]	S[17]	3 S[18]	S[19]	S[20]	S[21]	S[22]	S[23]	4 S[24]	S[25]	S[26]	S[27]	S[28]	S[29]	5 S[30]	S[31]
S[32]	S[33]	S[34]	S[35]	6 S[36]	S[37]	S[38]	S[39]	S[40]	S[41]	7 S[42]	S[43]	S[44]	S[45]	S[46]	S[47]
8 S[48]	S[49]	S[50]	S[51]	S[52]	S[53]	9 S[54]	S[55]	S[56]	S[57]	S[58]	S[59]	10 S[60]	S[61]	S[62]	S[63]
S[64]	S[65]	11 S[66]	S[67]	S[68]	S[69]	S[70]	S[71]	12 S[72]	S[73]	S[74]	S[75]	S[76]	S[77]	13 S[78]	S[79]
S[80]	S[81]	S[82]	S[83]	14 S[84]	S[85]	S[86]	S[87]	S[88]	S[89]	15 S[90]	S[91]	S[92]	S[93]	S[94]	S[95]

図3-6-24(3)

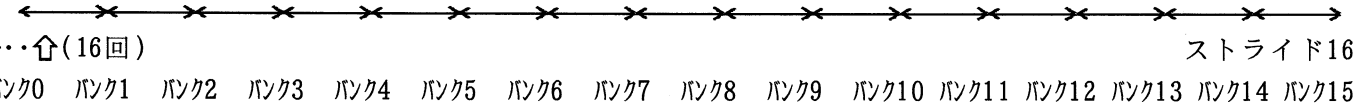
●stride=8：8ウェイ・バンクコンフリクトになります。



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 S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	1 S[8]	S[9]	S[10]	S[11]	S[12]	S[13]	S[14]	S[15]
2 S[16]	S[17]	S[18]	S[19]	S[20]	S[21]	S[22]	S[23]	3 S[24]	S[25]	S[26]	S[27]	S[28]	S[29]	S[30]	S[31]
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:
14 S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	15 S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]

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。



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 S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	S[9]	S[10]	S[11]	S[12]	S[13]	S[14]	S[15]
1 S[16]	S[17]	S[18]	S[19]	S[20]	S[21]	S[22]	S[23]	S[24]	S[25]	S[26]	S[27]	S[28]	S[29]	S[30]	S[31]
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:
15 S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]	S[ ]

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○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。

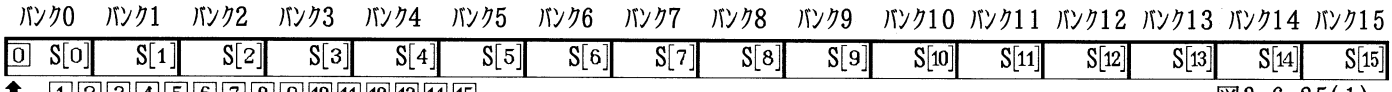


図3-6-25(1)

●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。

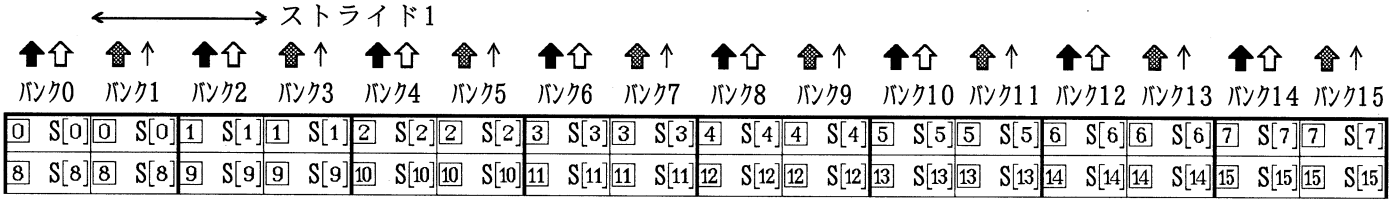


図3-6-25(2)

●stride=18：2ウェイ・バンクコンフリクトになります。

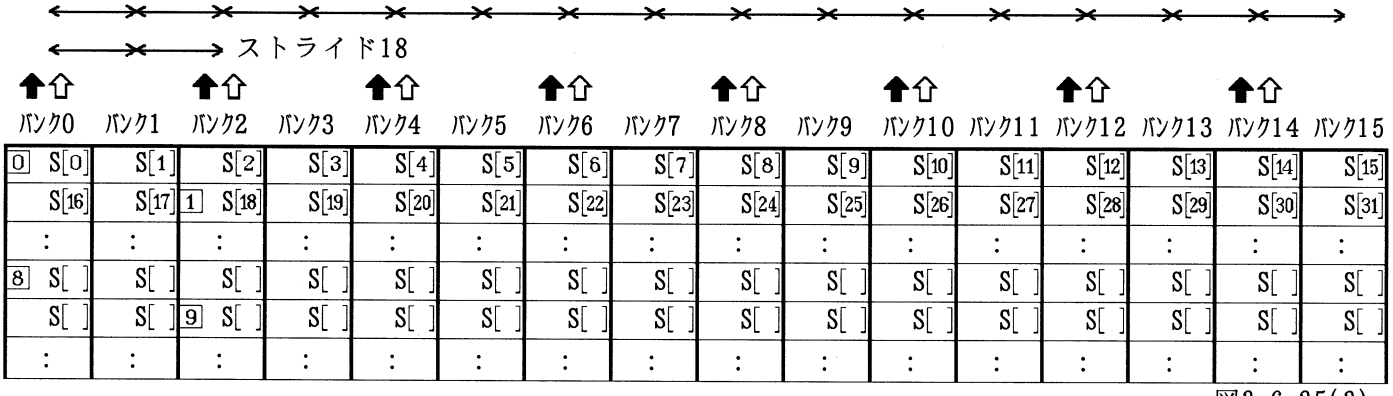


図3-6-25(3)

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。

③warp_serializeの値 ＝ (②nウェイ・バンクコンフリクト － １) × 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(図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) ○

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

図3-6-29(1) ✖

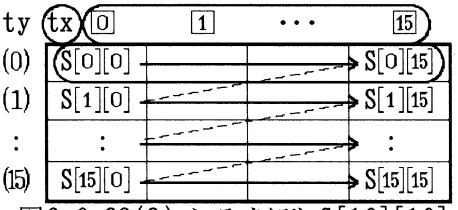


図3-6-28(2) シェアードメモリ S[16][16]

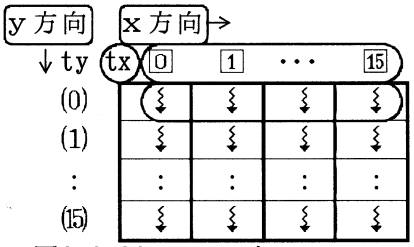


図3-6-30 スレッド

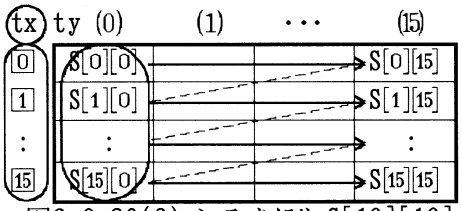


図3-6-29(2) シェアードメモリ S[16][16]

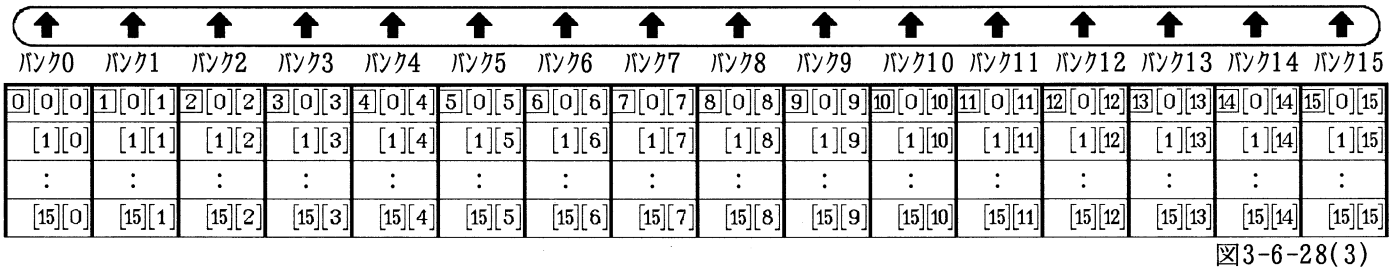


図3-6-28(3)

↑･･⇧(16回)

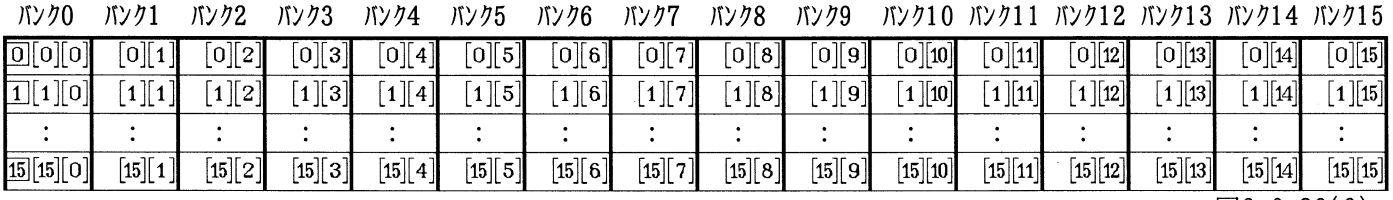


図3-6-29(3)

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)

tx \ ty	(0)	(1)	・・・	(15)	16
0	S[0][0]			S[0][15]	S[0][16]
1	S[1][0]			S[1][15]	S[1][16]
:				:	
15	S[15][0]			S[15][15]	S[15][16]

図3-6-30(2) ｼｪｱｰﾄﾞﾒﾓﾘ S[16][17]　　└――パディングした部分

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]
P[0][16]	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]	P[1][16]	2[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	[2][6]	[2][7]	[2][8]	[2][9]	[2][10]	[2][11]	[2][12]	[2][13]
[2][14]	[2][15]	P[2][16]	3[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	[3][6]	[3][7]	[3][8]	[3][9]	[3][10]	[3][11]	[3][12]
[3][13]	[3][14]	[3][15]	P[3][16]	4[4][0]	[4][1]	[4][2]	[4][3]	[4][4]	[4][5]	[4][6]	[4][7]	[4][8]	[4][9]	[4][10]	[4][11]
[4][12]	[4][13]	[4][14]	[4][15]	P[4][16]	5[5][0]	[5][1]	[5][2]	[5][3]	[5][4]	[5][5]	[5][6]	[5][7]	[5][8]	[5][9]	[5][10]
[5][11]	[5][12]	[5][13]	[5][14]	[5][15]	P[5][16]	6[6][0]	[6][1]	[6][2]	[6][3]	[6][4]	[6][5]	[6][6]	[6][7]	[6][8]	[6][9]
[6][10]	[6][11]	[6][12]	[6][13]	[6][14]	[6][15]	P[6][16]	7[7][0]	[7][1]	[7][2]	[7][3]	[7][4]	[7][5]	[7][6]	[7][7]	[7][8]
[7][9]	[7][10]	[7][11]	[7][12]	[7][13]	[7][14]	[7][15]	P[7][16]	8[8][0]	[8][1]	[8][2]	[8][3]	[8][4]	[8][5]	[8][6]	[8][7]
[8][8]	[8][9]	[8][10]	[8][11]	[8][12]	[8][13]	[8][14]	[8][15]	P[8][16]	9[9][0]	[9][1]	[9][2]	[9][3]	[9][4]	[9][5]	[9][6]
[9][7]	[9][8]	[9][9]	[9][10]	[9][11]	[9][12]	[9][13]	[9][14]	[9][15]	P[9][16]	10[10][0]	[10][1]	[10][2]	[10][3]	[10][4]	[10][5]
[10][6]	[10][7]	[10][8]	[10][9]	[10][10]	[10][11]	[10][12]	[10][13]	[10][14]	[10][15]	P[10][16]	11[11][0]	[11][1]	[11][2]	[11][3]	[11][4]
[11][5]	[11][6]	[11][7]	[11][8]	[11][9]	[11][10]	[11][11]	[11][12]	[11][13]	[11][14]	[11][15]	P[11][16]	12[12][0]	[12][1]	[12][2]	[12][3]
[12][4]	[12][5]	[12][6]	[12][7]	[12][8]	[12][9]	[12][10]	[12][11]	[12][12]	[12][13]	[12][14]	[12][15]	P[12][16]	13[13][0]	[13][1]	[13][2]
[13][3]	[13][4]	[13][5]	[13][6]	[13][7]	[13][8]	[13][9]	[13][10]	[13][11]	[13][12]	[13][13]	[13][14]	[13][15]	P[13][16]	14[14][0]	[14][1]
[14][2]	[14][3]	[14][4]	[14][5]	[14][6]	[14][7]	[14][8]	[14][9]	[14][10]	[14][11]	[14][12]	[14][13]	[14][14]	[14][15]	P[14][16]	[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][15]	P[15][16]

図3-6-30(3)

■ 行列乗算での使用例

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

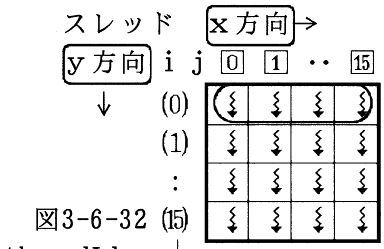


図3-6-32

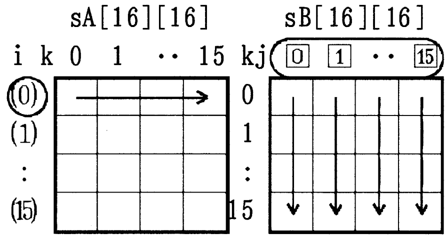


図3-6-33

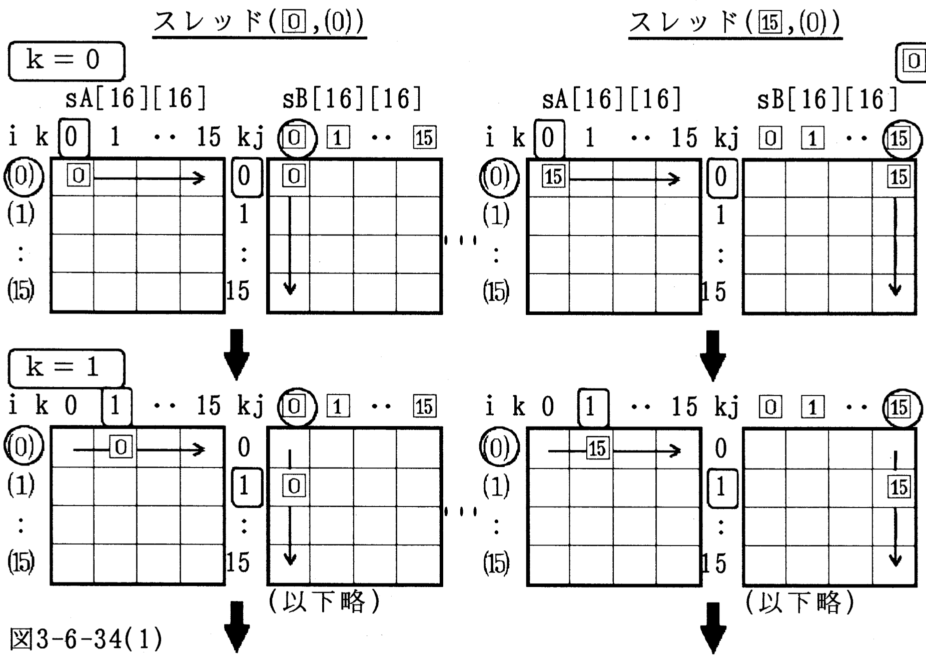


図3-6-34(1)

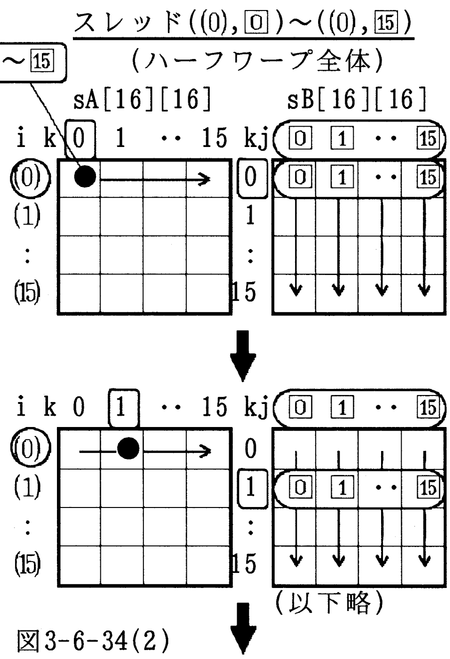


図3-6-34(2)

■ 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)

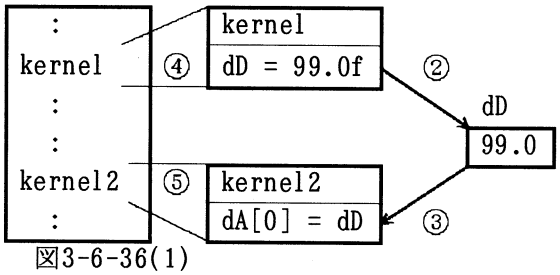


図3-6-36(1)

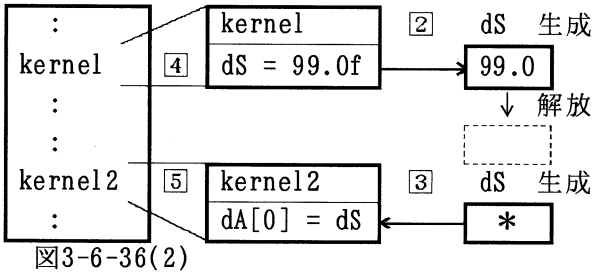


図3-6-36(2)

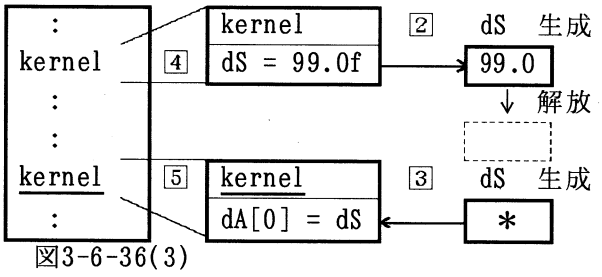


図3-6-36(3)

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。

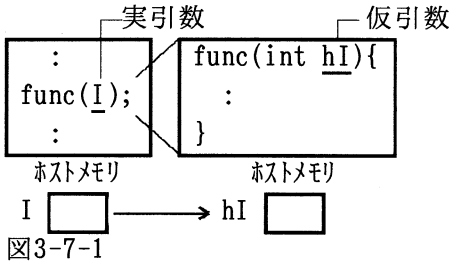


図3-7-1

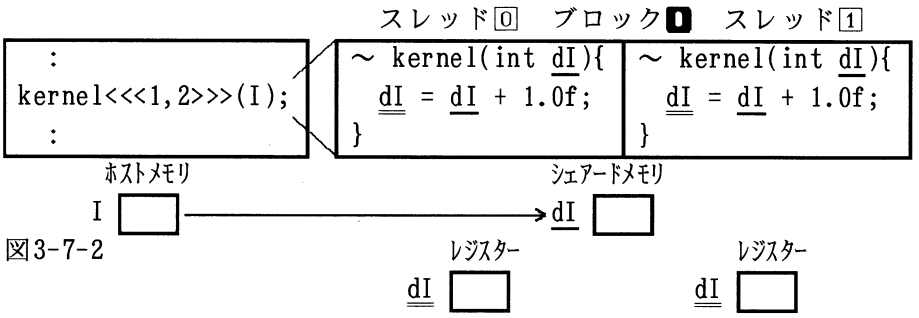


図3-7-2

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64×4=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16KB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、テストを何万回行なっても、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スレッドず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。まとめると、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 Programmning 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。

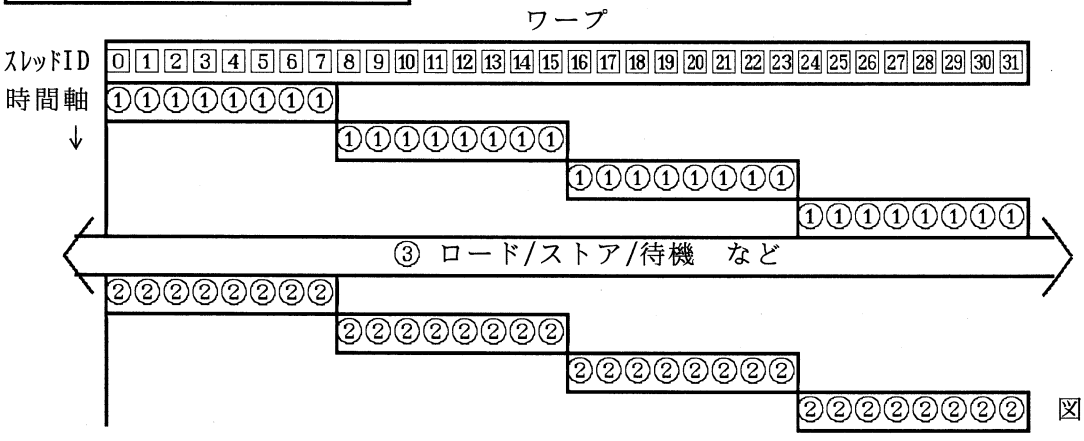
逆に、①と②の間で明示的に同期を取るべきなのに取り忘れたプログラムを、ブロックあたり32スレッド以下でテストしたため自動的に同期が取られ、計算結果が(たまたま)合っていたというケースも考えられます。この場合は、スレッド数を多くすると計算結果がおかしくなり、バグが表に現れます。従って、同期を使用するプログラムのテストは、ある程度たくさんのブロック数、スレッド数で行うようにして下さい。

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。

```
__global__ void kernel(){
    ①;                    同期
    ②;
}

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

図4-1-3(1)



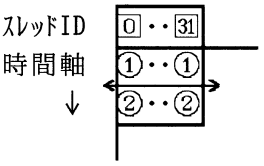


図4-1-3(3)

図4-1-3(2)

■ 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=▌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)

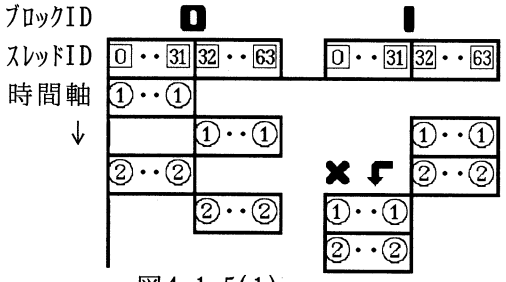


図4-1-5(1)

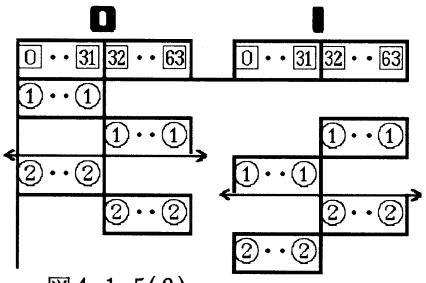


図4-1-5(2)

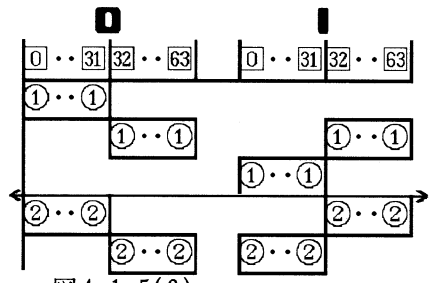


図4-1-5(3)

■ __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ID0～31)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ID0,1が含まれ、2つ目のワープにはスレッドID2,3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ID0,1の所属するワープが先に④を終了し、スレッドID2,3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ID0,1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ID2,3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

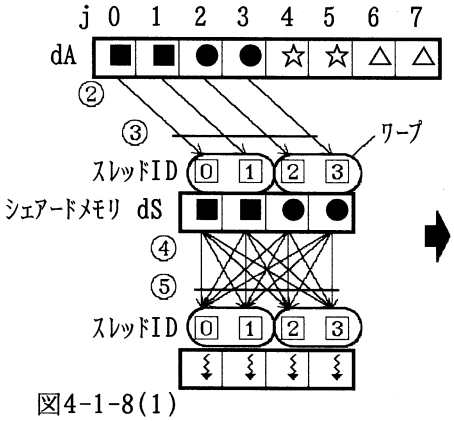


図4-1-8(1)

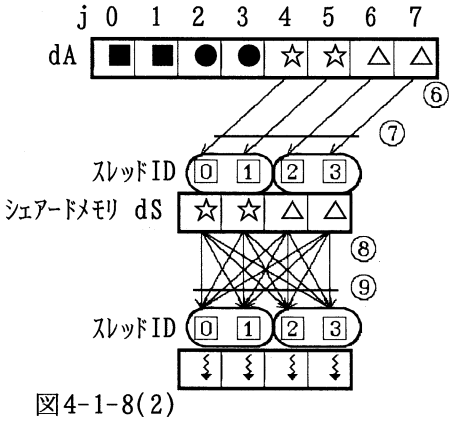


図4-1-8(2)

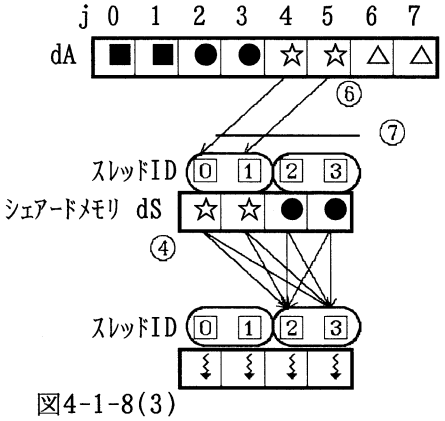


図4-1-8(3)

■ 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cudaThreadSynchronize()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)。

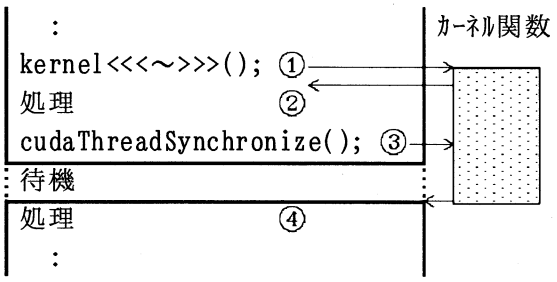


図4-1-9(1)

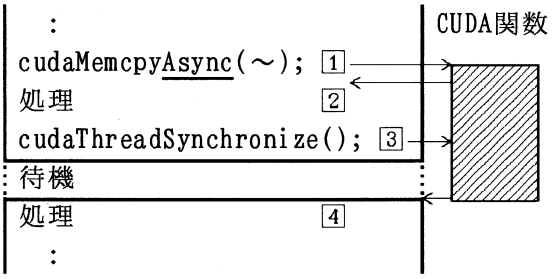


図4-1-9(2)

■ 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ID0の全スレッドが図4-1-15(1)の[1]を、ブロックID1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

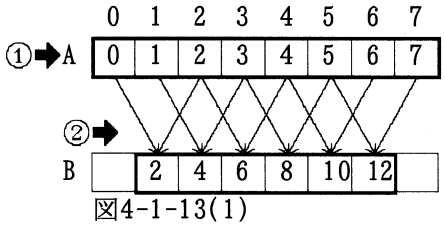


図4-1-13(1)

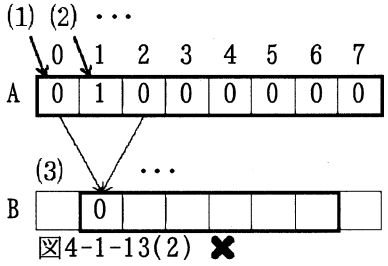


図4-1-13(2) ✖

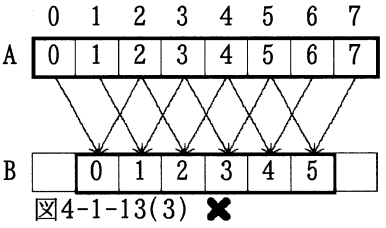


図4-1-13(3) ✖

```
__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

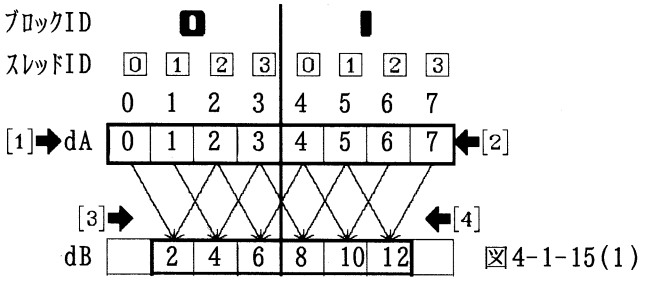


図4-1-15(1)

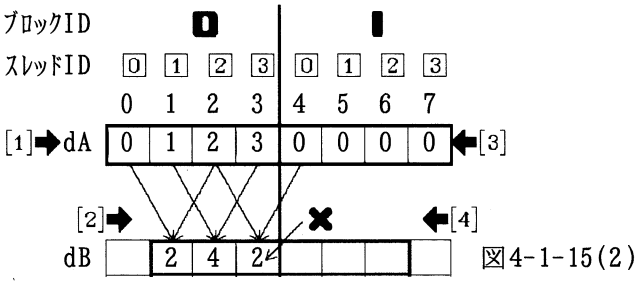


図4-1-15(2)

■ 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変数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 Capabilitiy 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()

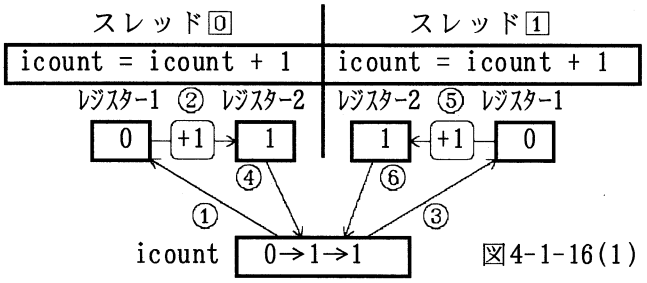


図4-1-16(1) ✖

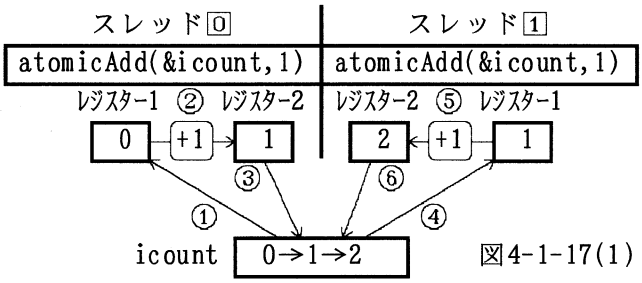


図4-1-17(1)

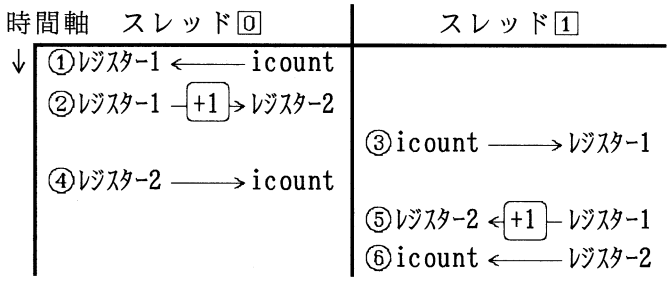


図4-1-16(2) ✖

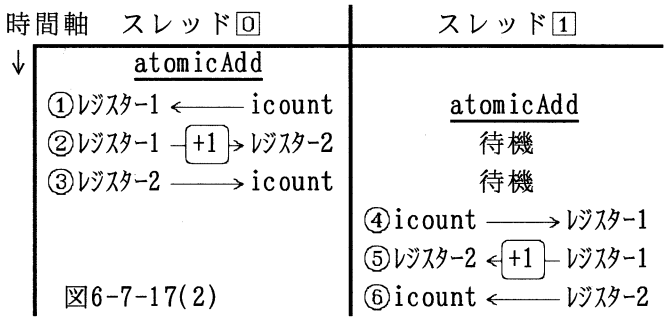


図6-7-17(2)

```
__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)
}
```

図4-1-18

```
   :
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)
}
   :
```

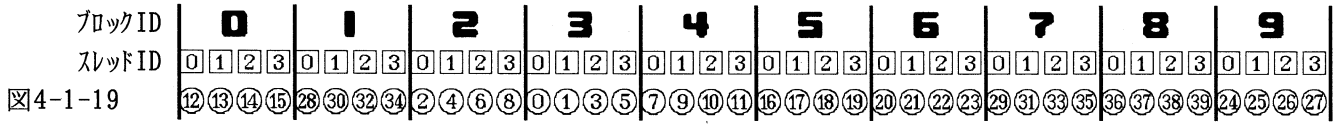


図4-1-19

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「cutilSafe Call」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_inline.h>                  ⑬
    :
  cutilSafeCall(cudaMalloc((void**)&dA,size));
    :                                      ⑭
```

図4-2-3

```
#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

```
$ 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ID30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動作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ID30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。

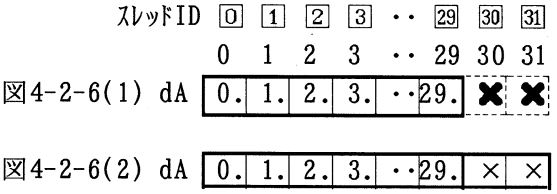
図4-2-5(1)を通常に実行した場合、発生したセグメンテーション・フォールトは検出されず、プログラムは異常終了せず、メッセージも表示されません。図4-2-2のチェックルーチンを付加しても同様です。
CUDAで提供されている「cuda-memcheck」コマンドを使うと、セグメンテーションフォールトを検出することができます(詳細はcuda-memecheckのマニュアル(付録参照)を参照して下さい)。図4-2-7の⑥の下線部を付けて実行すると(実際はシェルを使用して実行します)、⑦の下線部に示すように、カーネル関数kernel内で、読み込み時にセグメンテーションフォールトが発生したことが表示されます。
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)




```
$ 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。

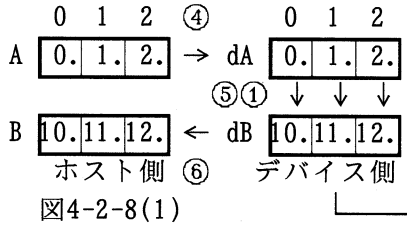


図4-2-8(1)

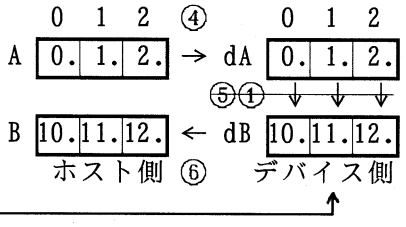


図4-2-8(2)

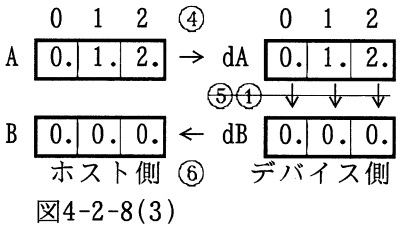


図4-2-8(3)

```
#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

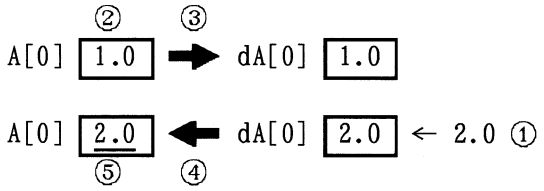


図4-2-14(1)

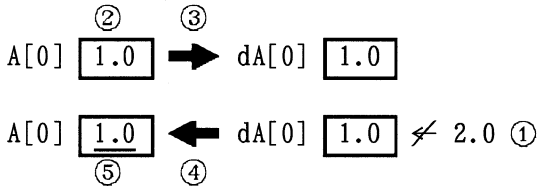


図4-2-14(2)

```
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、単精度だと結果が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(ｵｰｾﾛ)」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と図4-2-17の下線部をdoubleにし、倍精度で実行すると、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(ｵｰｾﾛ) -Xopencc -O0(ｵｰｾﾛ) -test.cu ↵          ①
(ホスト　：単精度) 3.33333325386047363281  ②
(カーネル：単精度) 3.33333349227905273438  ③
$ nvcc -O0(ｵｰｾﾛ) -Xopencc -O0(ｵｰｾﾛ) -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
```

図4-3-1(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4-3-2(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;
```

図4-3-2(2)

```
(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4-3-3(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.
```

図4-3-3(2)

```
(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.
```

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。

i = blockIdx.x(=1)*blockDim.x(=64) + 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;
```

図4-3-5(2)

```
(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))]
```

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、ohter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

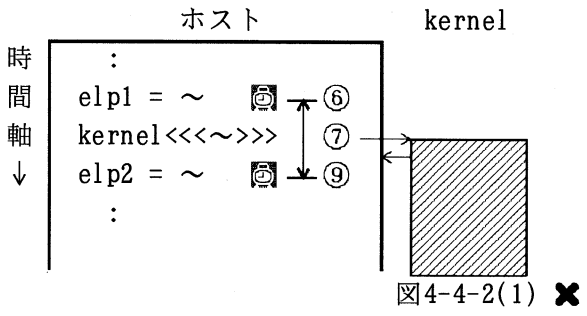


図4-4-2(1) ✖

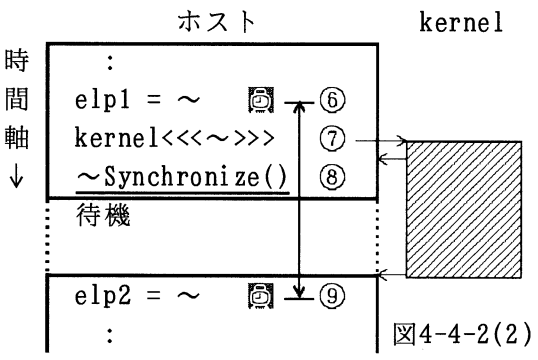


図4-4-2(2)

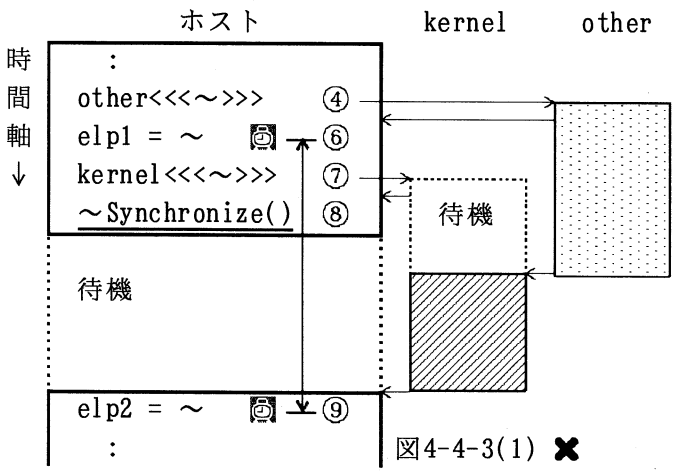


図4-4-3(1) ✖

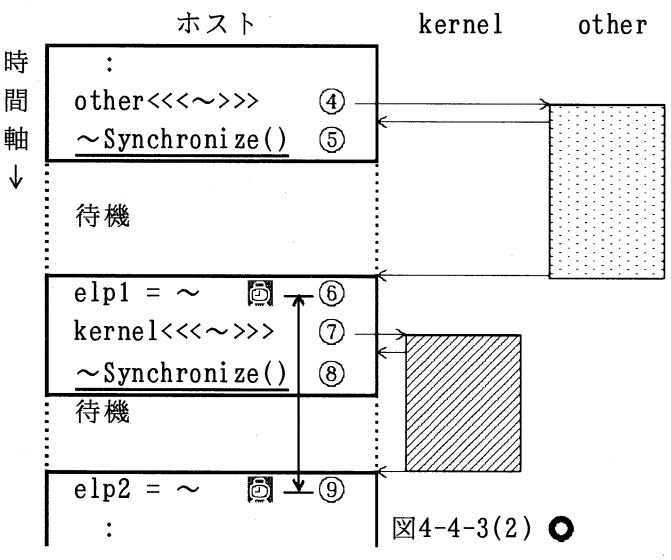


図4-4-3(2) ○

■ 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;  測定したい部分 ③
  clock_t iend = clock();    ④
  dELAPSE[i] = iend - ista;    ⑤
}
```

図4-4-4(1)

```
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]);
  }
    :
```

```
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、ワープの占有率(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、そのファイル内に、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30×2×128×4(ﾊﾞｲﾄ)＝30720(ﾊﾞｲﾄ)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●,◎,○で示します。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つのワ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。

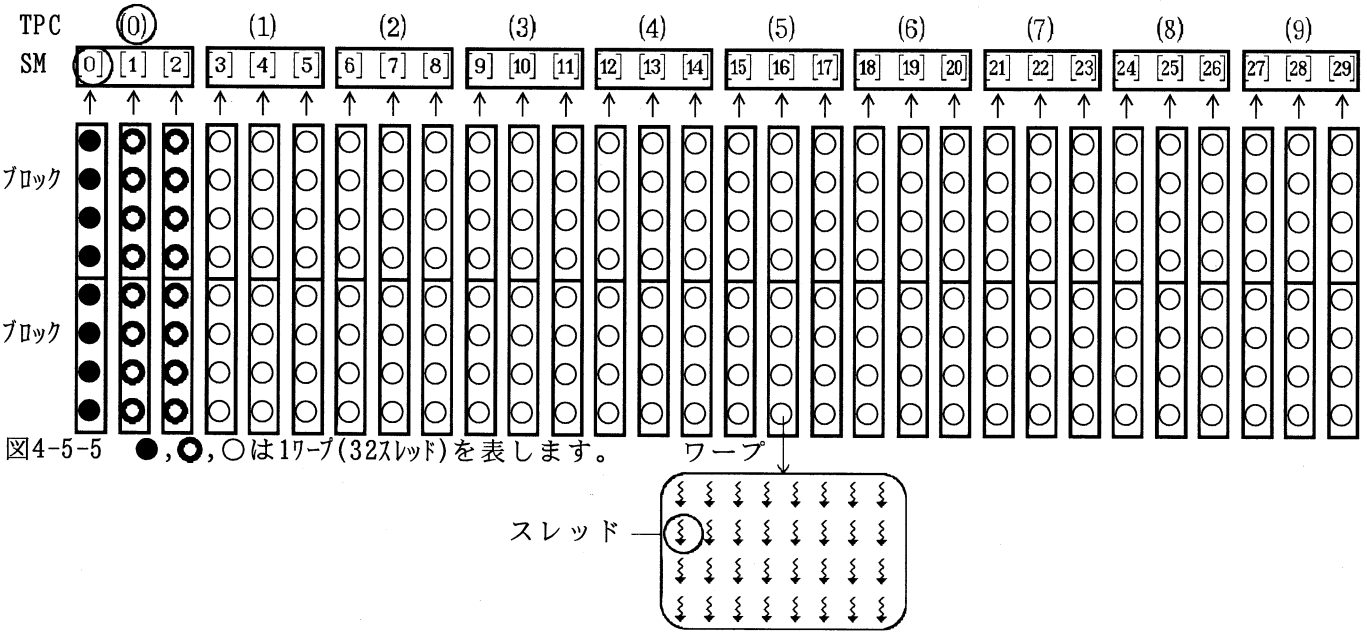


図4-5-5　●,◎,○は1ワープ(32スレッド)を表します。

●（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)

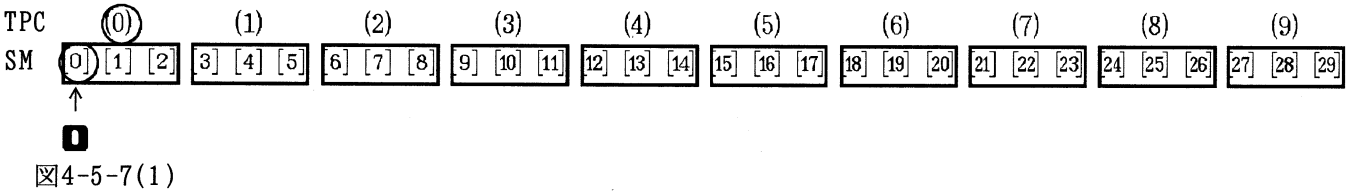


図4-5-7(1)

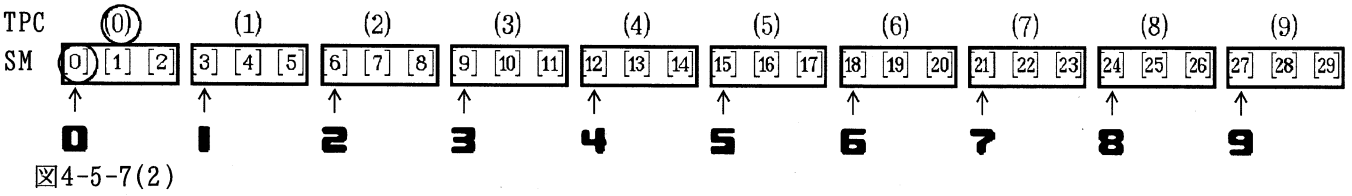


図4-5-7(2)

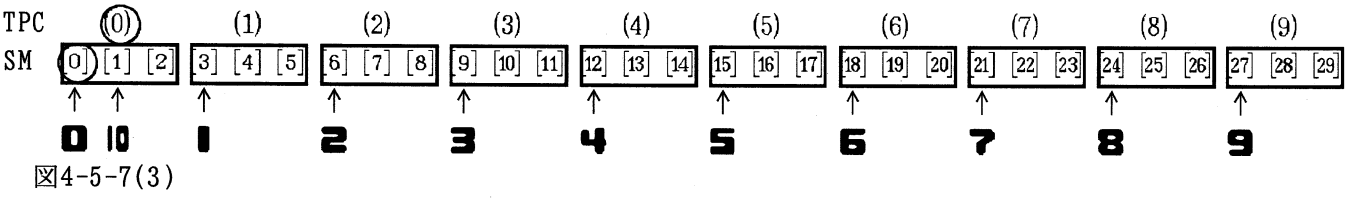


図4-5-7(3)

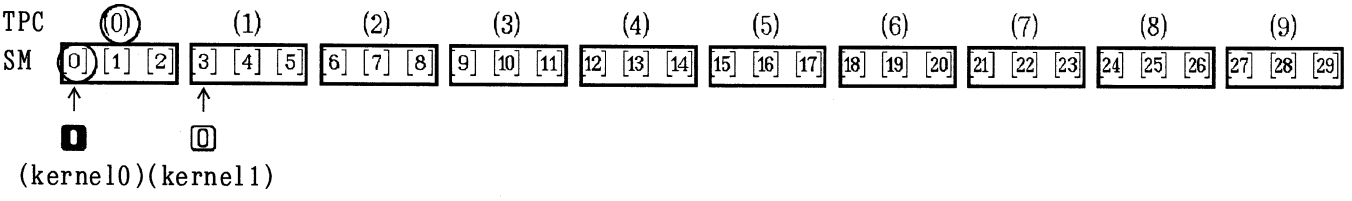


図4-5-7(4)

●「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下さい。

```
NV_Warning: Signal ﾊﾟﾗﾒｰﾀ名 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」: [ﾜｰﾌﾟの実行命令数]×[SM[0]で実行したﾜｰﾌﾟ数] 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」: [ﾜｰﾌﾟの(実際に実行した)分岐命令数(if文など)]×[SM[0]で実行したﾜｰﾌﾟ数] 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」: [ﾜｰﾌﾟのﾜｰﾌﾟ･ﾀﾞｲﾊﾞｰｼﾞｪﾝﾄ(６－５節参照)した分岐命令数]×[SM[0]で実行したﾜｰﾌﾟ数] 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 ] ～
```

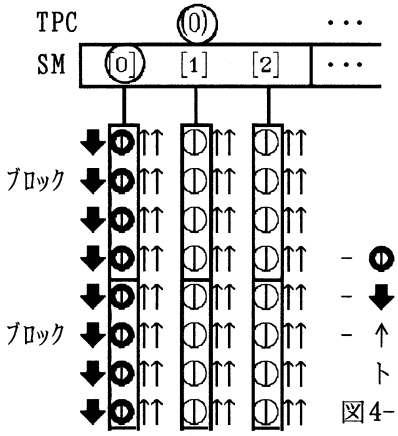
【注目点】前述の「gld_request」「gst_request」と、「gld_32b」「gld_64b」「gld_128b」「gst_32b」「gst_64b」「gst_128b」を使用して、グローバルメモリに対するロード/ストアが、最も効率よくコアレスアクセスされているか(３－２節参照)を知ることができます。これを以下で説明します。

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)[ﾜｰﾌﾟのﾛｰﾄﾞ命令要求回数] ＝ (11)[ﾊｰﾌﾜｰﾌﾟの(ﾄﾗﾝｻﾞｸｼｮﾝの)ﾛｰﾄﾞ回数] になっている。 (10)[ﾜｰﾌﾟのﾛｰﾄﾞ命令要求回数] ＜ (11)[ﾊｰﾌﾜｰﾌﾟの(ﾄﾗﾝｻﾞｸｼｮﾝの)ﾛｰﾄﾞ回数] 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。



- ❶と①はワープを、❲ と ❳はハーフワープを示します。
- ⬇は、ワープの、グローバルメモリからの1回のロード命令の要求を示します。
- ↑は、ハーフワープの、グローバルメモリからの
  トランザクションの1回のロードを示します。

図4-5-8

まず、以下の3つの値を使用して、(10)[ﾜｰﾌﾟのﾛｰﾄﾞ命令要求回数](1つの❶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ﾌﾞﾛｯｸ内のﾜｰﾌﾟ数」：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❶の数)。

(10)[ﾜｰﾌﾟのﾛｰﾄﾞ命令要求回数] (1つの❶あたりの⬇の数)
＝ [ﾜｰﾌﾟのﾛｰﾄﾞ命令要求回数]×[SM[0]内で実行したﾜｰﾌﾟ数] ÷ [SM[0]内で実行したﾜｰﾌﾟ数]
＝ 「gld_request」 ÷ (「sm_cta_launched」×「1ﾌﾞﾛｯｸ内のﾜｰﾌﾟ数」)
＝ 8回 ÷ (2個 × 4個)
＝ 1回

次に、以下の3つの値を使用して、(11)[ﾊｰﾌﾜｰﾌﾟの(ﾄﾗﾝｻﾞｸｼｮﾝの)ﾛｰﾄﾞ回数](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ﾌﾞﾛｯｸ内のﾊｰﾌﾜｰﾌﾟ数」：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)[ﾊｰﾌﾜｰﾌﾟの(ﾄﾗﾝｻﾞｸｼｮﾝの)ﾛｰﾄﾞ回数] (1つの❲ または ❳あたりの↑の数)
＝ [ﾊｰﾌﾜｰﾌﾟの(ﾄﾗﾝｻﾞｸｼｮﾝの)ﾛｰﾄﾞ回数]×[TPC(0)内で実行したﾊｰﾌﾜｰﾌﾟ数] ÷ [TPC(0)内で実行したﾊｰﾌﾜｰﾌﾟ数]
＝ 「gld_64b」 ÷ (「cta_launched」×「1ﾌﾞﾛｯｸ内のﾊｰﾌﾜｰﾌﾟ数」)
＝ 48回 ÷ (6個 × 8個)
＝ 1回

以上より、A：[ﾜｰﾌﾟのﾛｰﾄﾞ命令要求回数] ＝ [ﾊｰﾌﾜｰﾌﾟの(ﾄﾗﾝｻﾞｸｼｮﾝの)ﾛｰﾄﾞ回数] (＝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」:[ﾊｰﾌﾜｰﾌﾟ(注1)のﾛｰｶﾙﾒﾓﾘからのﾛｰﾄﾞ回数]×[TPC(0)(注2)で実行したﾊｰﾌﾜｰﾌﾟ数] が表示されます。
●「local_store」:[ﾊｰﾌﾜｰﾌﾟ(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ﾌﾟ数] 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」:[ﾜｰﾌﾟのｼｪｱｰﾄﾞﾒﾓﾘ(注1)のﾊﾞﾝｸｺﾝﾌﾘｸﾄの回数(注2)]×[SM[0]で実行したﾜｰﾌﾟ数] 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。

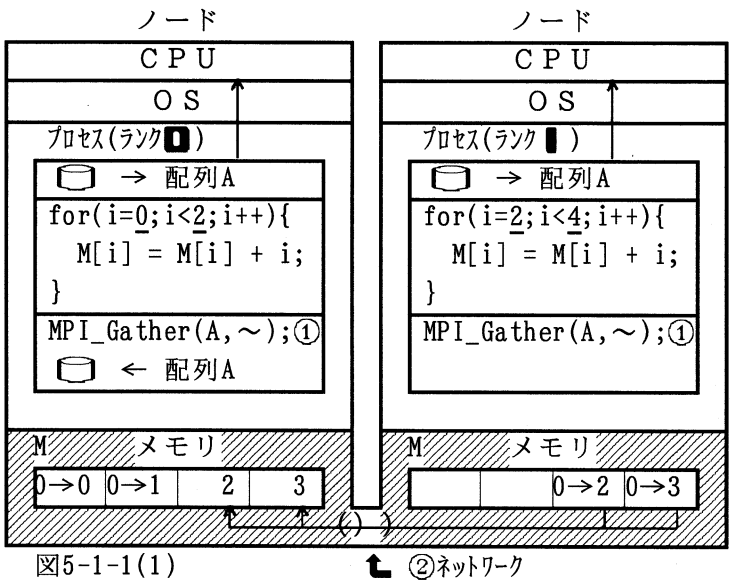


図5-1-1(1)

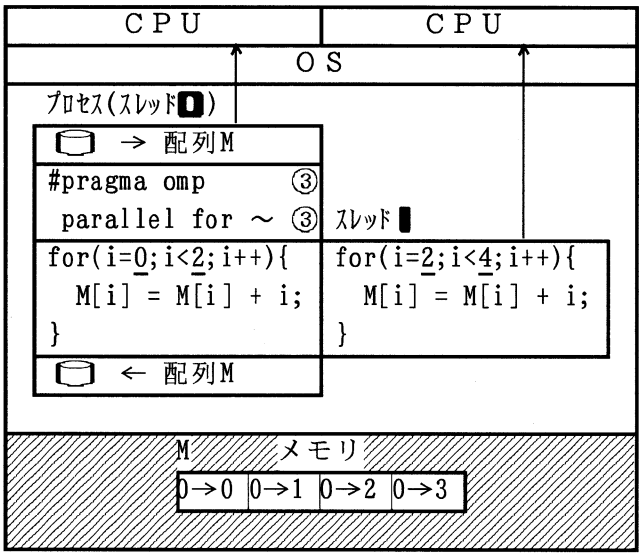


図5-1-1(2)

■ 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CPU100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、CPU2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。

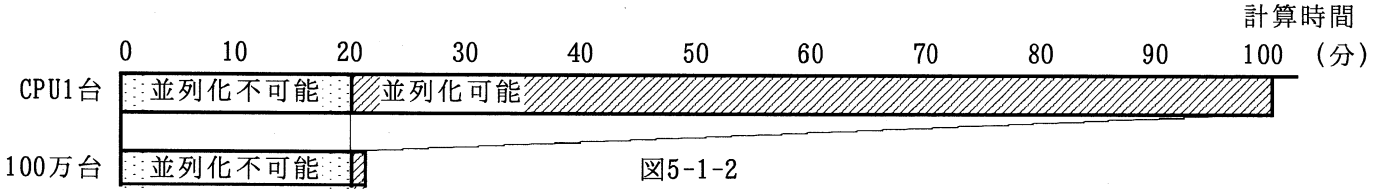


図5-1-2

■ 速度向上率を上げるためには

CPU1台で100分かかっているプログラムは、理想的には、CPU4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CPU3)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(CPU0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。

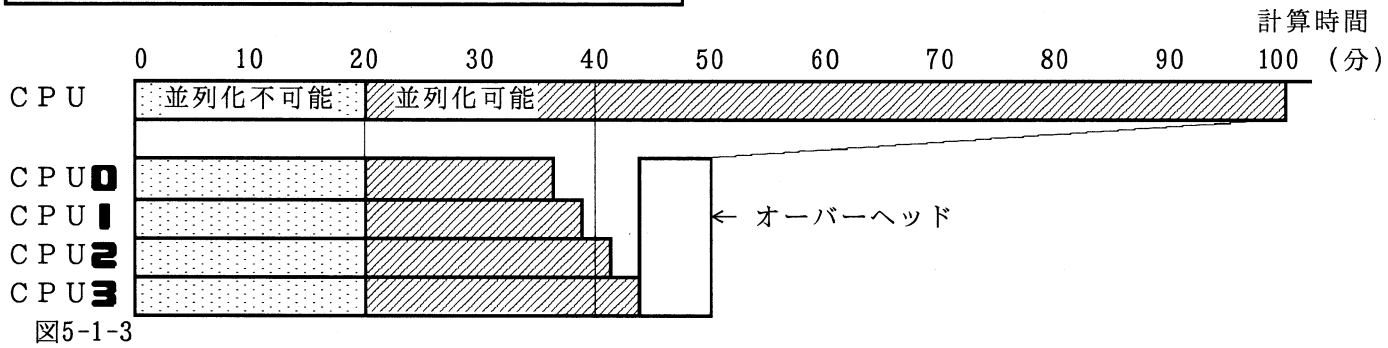


図5-1-3

■ 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。

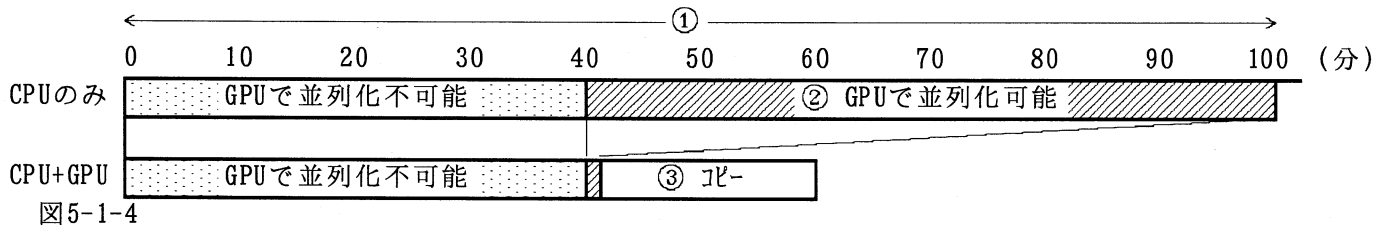


図5-1-4

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。

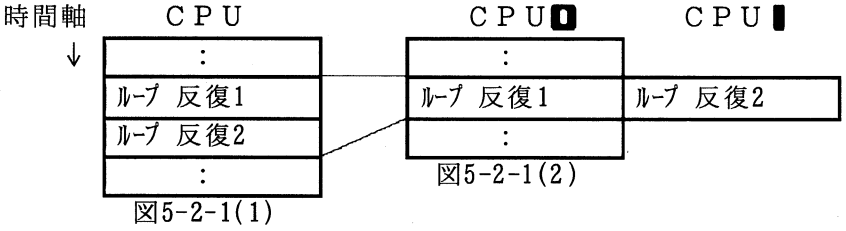


図5-2-1(1)　　図5-2-1(2)

■ 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。これを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。

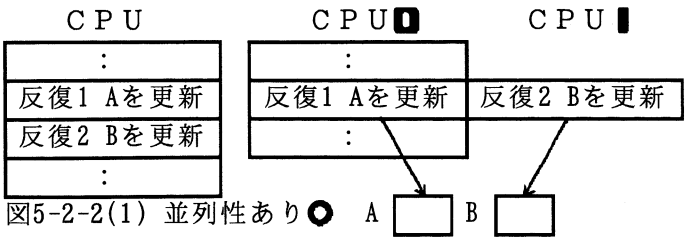


図5-2-2(1) 並列性あり○

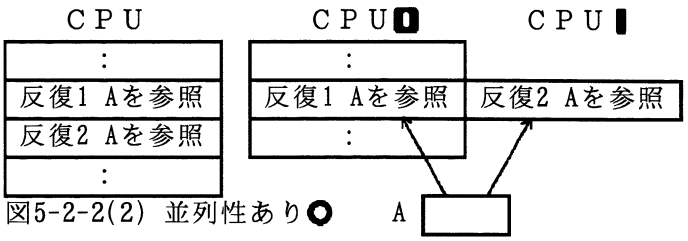


図5-2-2(2) 並列性あり○

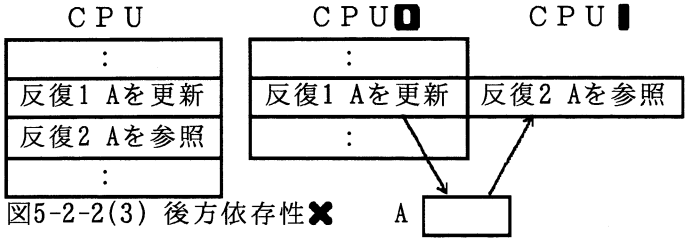


図5-2-2(3) 後方依存性✖

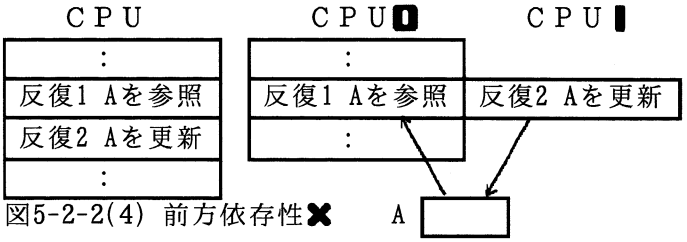


図5-2-2(4) 前方依存性✖

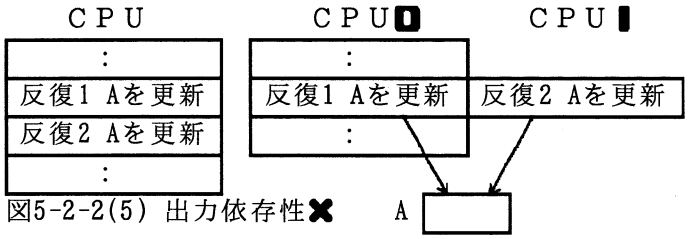


図5-2-2(5) 出力依存性✖

■ 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、実行後は「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、CPU0の処理(①)とCPU1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

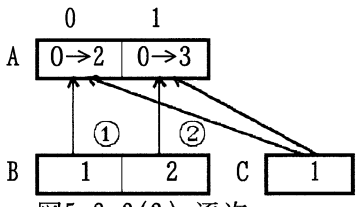


図5-2-3(3) 逐次

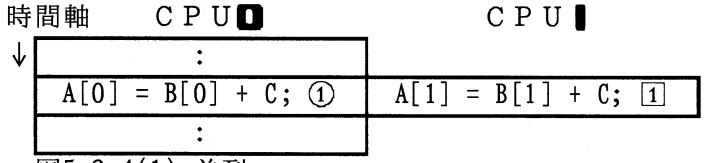


図5-2-4(1) 並列

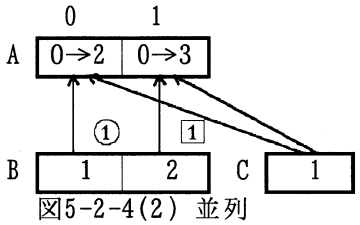


図5-2-4(2) 並列

■ 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、CPU0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、CPU1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

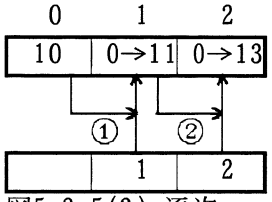


図5-2-5(3) 逐次

時間軸　　CPU❶　　CPU❙

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

図5-2-6(1) 並列(①が先に実行)

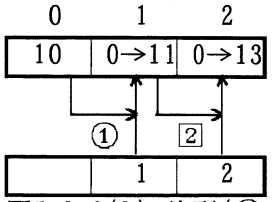


図5-2-6(2) 並列(①が先に実行)

時間軸　　CPU❶　　CPU❙

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

図5-2-7(1) 並列(①が先に実行)

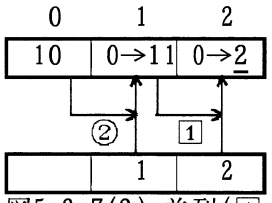


図5-2-7(2) 並列(①が先に実行)

■ 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、CPU❶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❙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

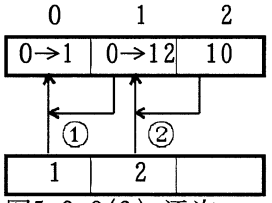


図5-2-8(3) 逐次

時間軸　　CPU❶　　CPU❙

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

図5-2-9(1) 並列(①が先に実行)

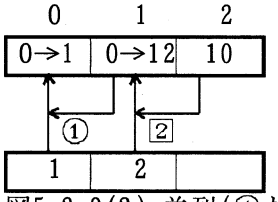


図5-2-9(2) 並列(①が先に実行)

時間軸　　CPU❶　　CPU❙

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

図5-2-10(1) 並列(①が先に実行)

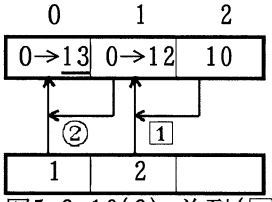


図5-2-10(2) 並列(①が先に実行)

■ 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▮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▮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

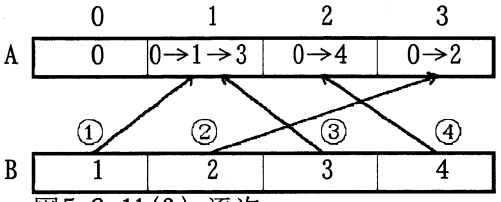


図5-2-11(3) 逐次

時間軸↓	CPU▮	CPU▮
	:	
	A[1] = B[0]; ①	A[1] = B[2]; [2]
	A[3] = B[1]; ③	A[2] = B[3]; [4]
	:	

図5-2-12(1) 並列(①が先に実行)

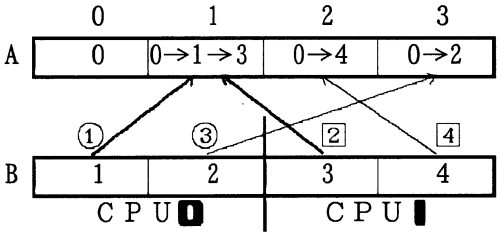


図5-2-12(2) 並列(①が先に実行)

時間軸↓	CPU▮	CPU▮
	:	
	A[1] = B[0]; ②	A[1] = B[2]; [1]
	A[3] = B[1]; ④	A[2] = B[3]; [3]
	:	

図5-2-13(1) 並列([1]が先に実行)

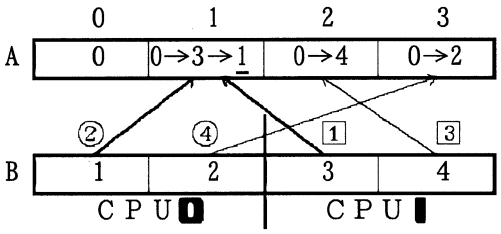


図5-2-13(2) 並列([1]が先に実行)

```
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

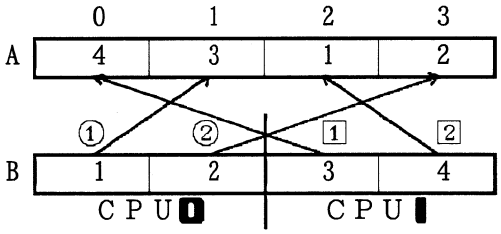


図5-2-14(3) 並列

■ 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)

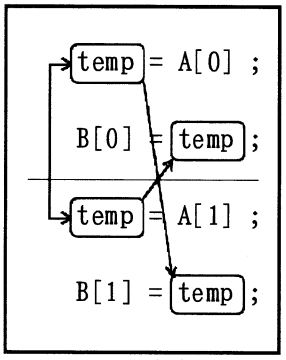


図5-2-16(2)

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

図5-2-17(1)

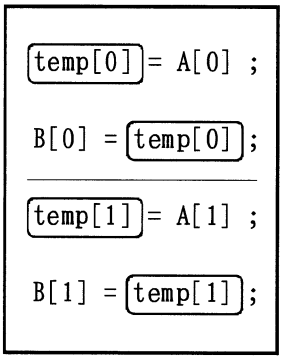


図5-2-17(2)

```
__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。

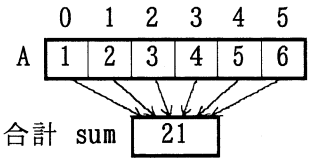


図5-2-19

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CPU❶とCPU❙が図5-2-21(2)の数字の順に動作したとします。するとCPU❶が⑦で、「1」を変数sumにストアし、その後CPU❙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、Ⓐ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Ⓑ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、CPU❶が代表してⒷの加算を行う方法や、ロック機能(あるCPUがⒷの加算を実行中に、他のCPUはⒷ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)

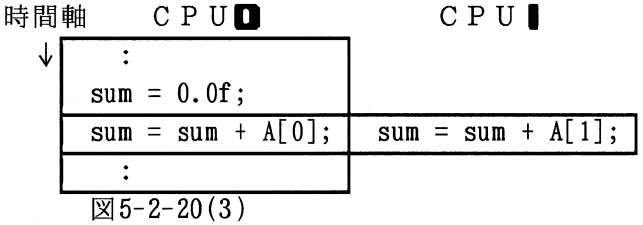


図5-2-20(3)

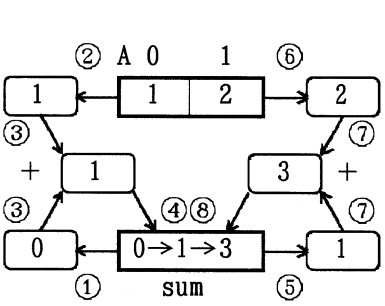


図5-2-21(1)

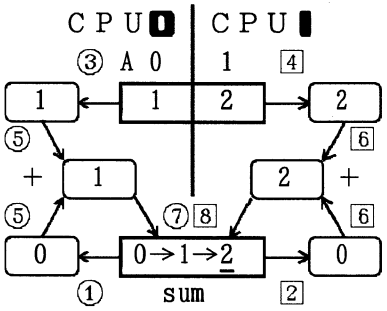


図5-2-21(2)

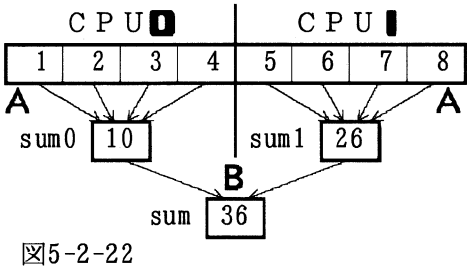


図5-2-22

■ 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)　　(2) (a + b) + c ≠ 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4CPUで、(+)の加算が「早い者順」に行われる場合は、同じ4CPU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]
- ２ＣＰＵ：(A[0] + A[1] + A[2] + A[3]) + (A[4] + A[5] + A[6] + A[7])
- ４ＣＰＵ：(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。

```
   :
for(i=0;i<100;i++){ ①
  sub1();
}
   :
```

```
void sub1(){
  計算
  sub2();
  計算
}
```

```
void sub2(){
  計算
}
```

図5-2-23(1)

```
   :
for(i=99;i>=0;i--){
  sub1();
}
   :
```

図5-2-23(2)

```
   :
for(i=0;i<3;i++){ ②
  A[i] = B[i] + C;
}
   :
```

図5-2-24(1)

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

図5-2-24(2)

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

図5-2-25(1)

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

図5-2-25(2)

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

図5-2-25(3)

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

図5-2-25(4)

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

図5-2-25(5)

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

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❙の[2]がCPU❚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。

```
C[1] = B[1];   1反復目
A[1] = C[0];
C[2] = B[2];   2反復目
A[2] = C[1];
```

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) ○

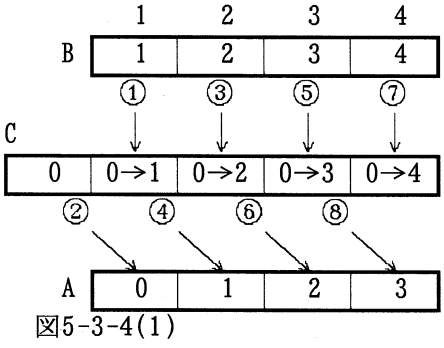


図5-3-4(1)

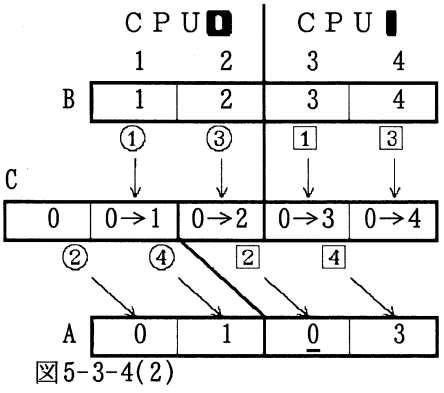


図5-3-4(2)

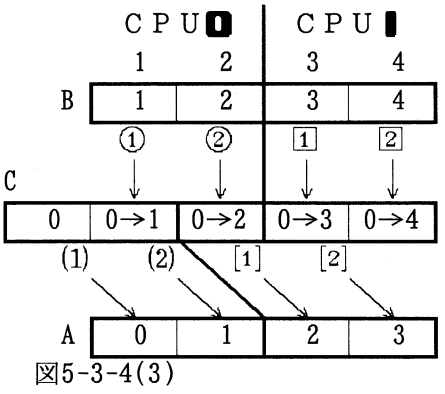


図5-3-4(3)

■ 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でループを離脱します。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++){
  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]を無視した場合、並列化による速度向上率は4CPUで2倍、8CPU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4CPUで4倍、8CPU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)

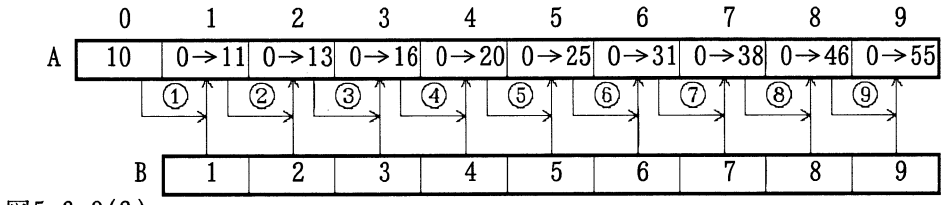


図5-3-8(2)

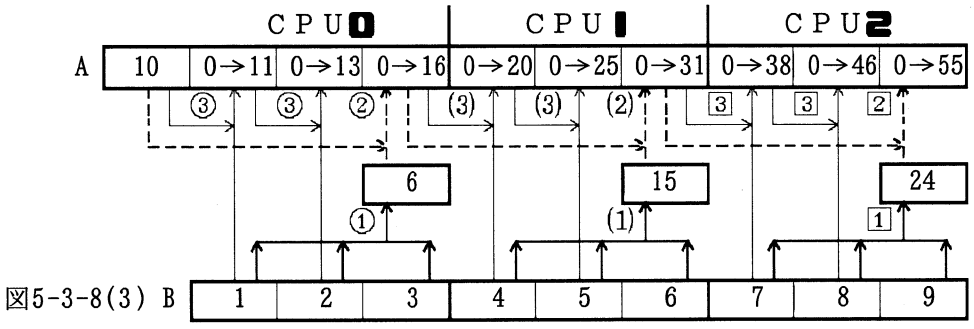


図5-3-8(3)

■ 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)

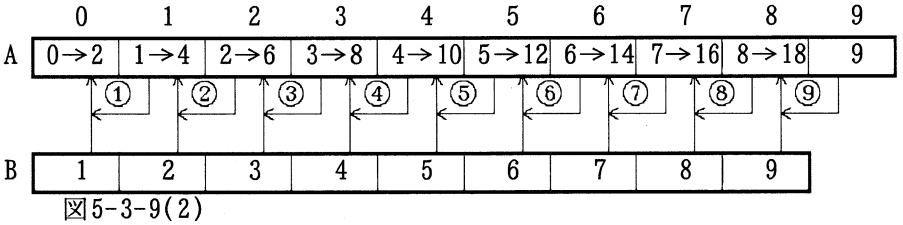


図5-3-9(2)

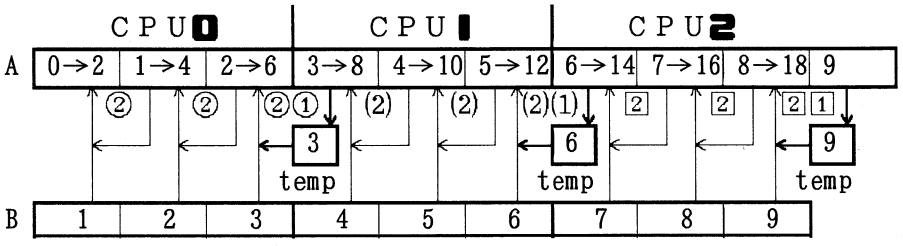


図5-3-9(3)

■ 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。CPU2台で並列に実行し、CPU■が④を計算する前にCPU▌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)

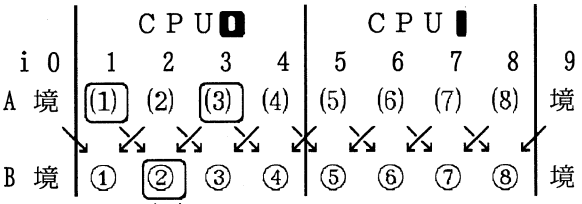


図5-3-10(3)

```
   :
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)

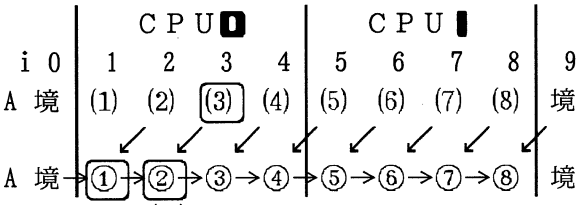


図5-3-11(3)

```
   :
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) ○

図5-3-12(2)

```
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。
```

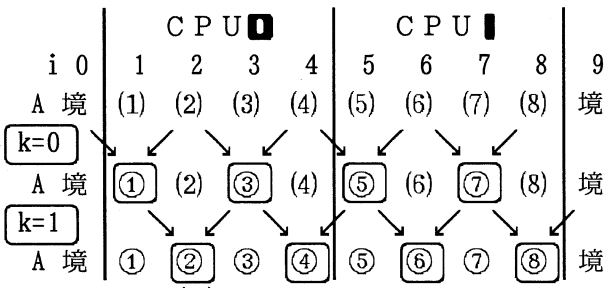


図5-3-12(3)

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2CPU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、並列化のため、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)

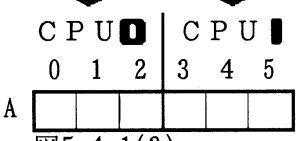


図5-4-1(3)

■ 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。

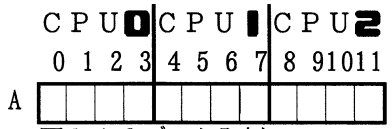


図5-4-2 ブロック分割

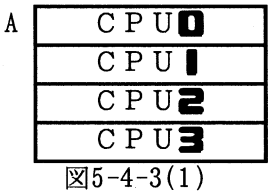


図5-4-3(1)

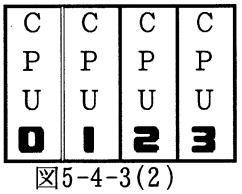


図5-4-3(2)

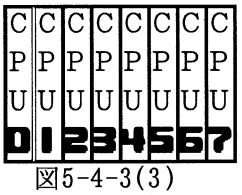


図5-4-3(3)

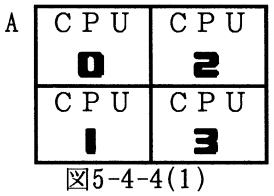


図5-4-4(1)

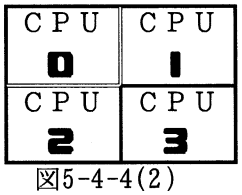


図5-4-4(2)

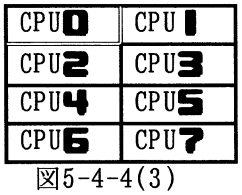


図5-4-4(3)

(2) サイクリック分割/ブロックサイクリック分割

CPU0,1,2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)の〇に示すように、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、CPU0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CPU0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。

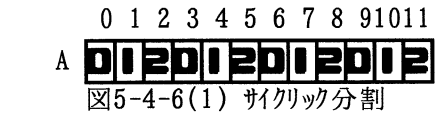


図5-4-6(1) サイクリック分割

図5-4-7(1) ブロックサイクリック分割

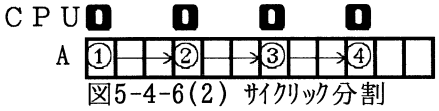


図5-4-6(2) サイクリック分割

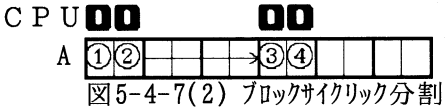


図5-4-7(2) ブロックサイクリック分割

図5-4-6(3) サイクリック分割

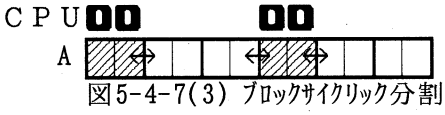


図5-4-7(3) ブロックサイクリック分割

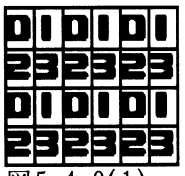


図5-4-8(1)

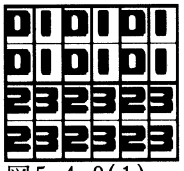


図5-4-8(1)

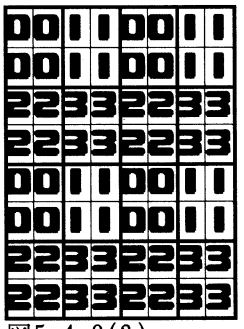


図5-4-8(3)

■ 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。

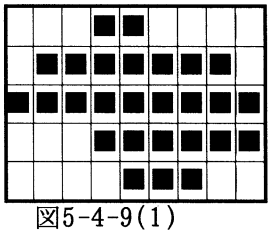


図5-4-9(1)

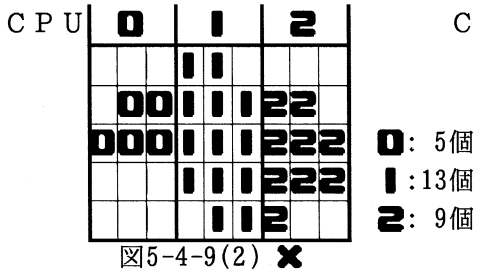


図5-4-9(2) ✖

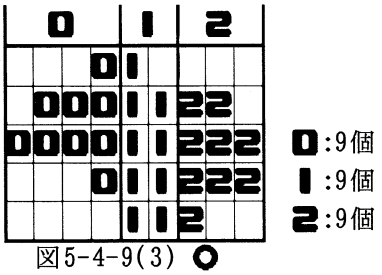


図5-4-9(3) ○

(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。

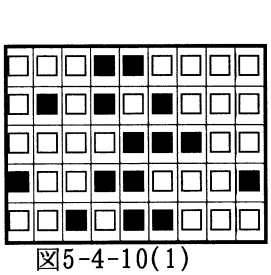


図5-4-10(1)

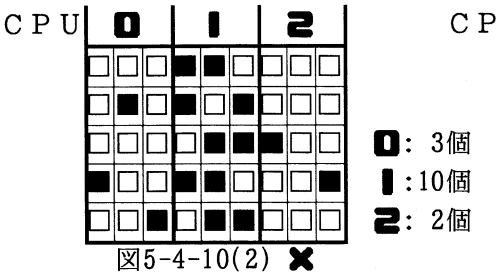


図5-4-10(2) ✖

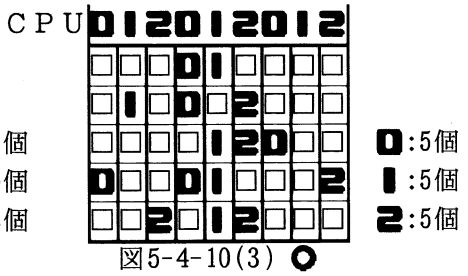


図5-4-10(3) ○

(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。

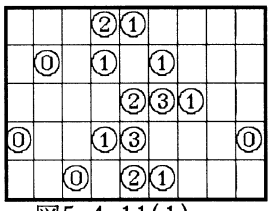


図5-4-11(1)

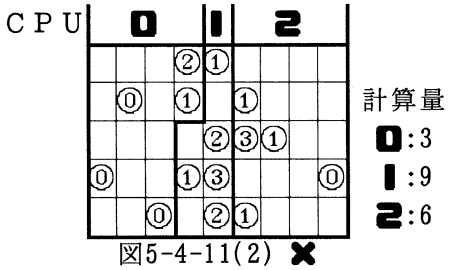


図5-4-11(2) ✖

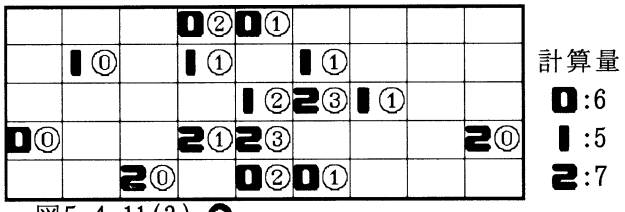


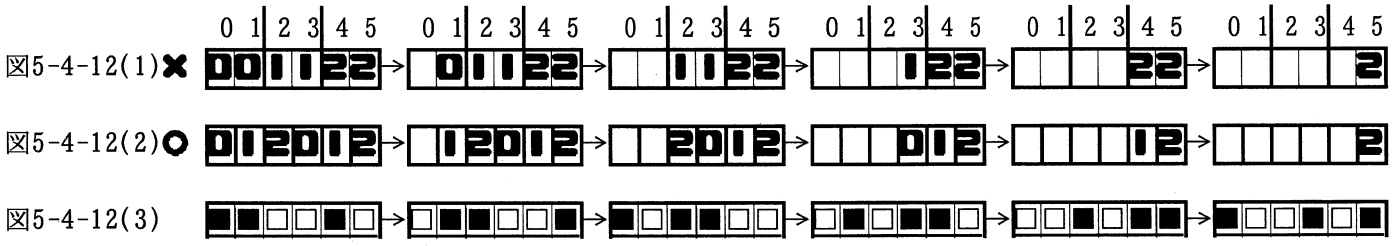
図5-4-11(3) ○

(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。



■ マスタースレーブ(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タースレーブ(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。

マスタースレーブ方式では、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、マスタースレーブ方式で、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)をマスター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、マスター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。

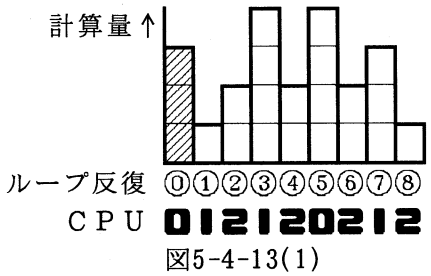


図5-4-13(1)

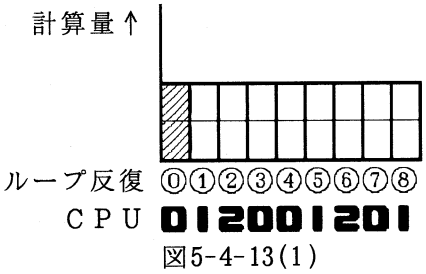


図5-4-13(1)

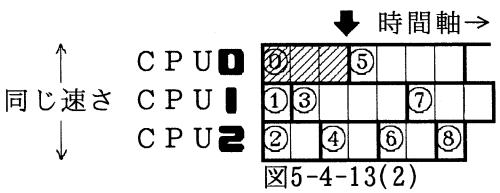


図5-4-13(2)

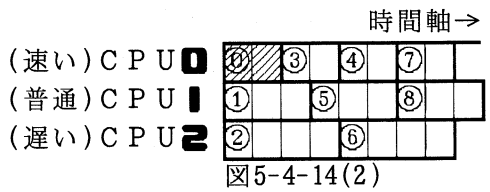


図5-4-14(2)

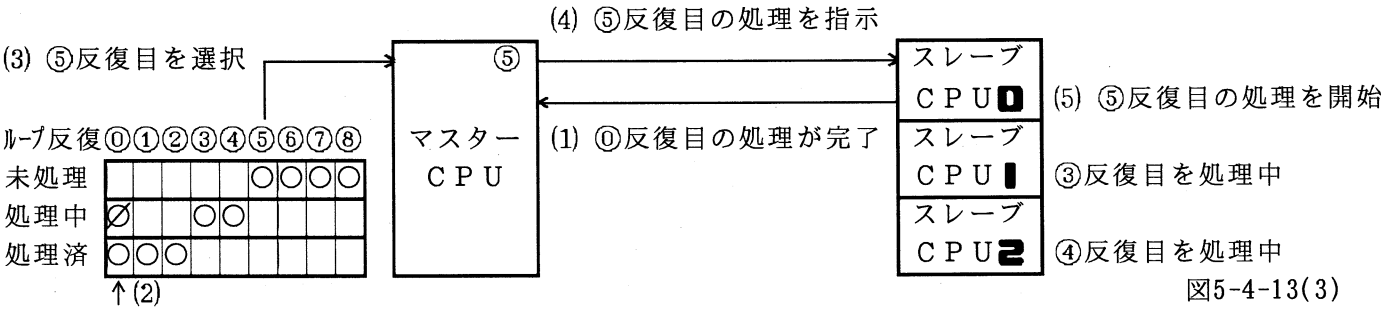


図5-4-13(3)

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2CPU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2CPU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();
      :
  }
    :
```

図5-5-1(1)

```
void func(){
    :
  for(iy=0;iy<100;iy++){ ← ②
    for(ix=0;ix<100;ix++){ ← ①
      A[iy][ix] = ～;
    }
  }
    :
```

図5-5-1(2)

```
    :
for(iy=0;iy<100;iy++){ ②
      CPU■                    CPU■
  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 ✖

```
    :
  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) ✖

```
    :
      CPU■                    CPU■
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;
    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

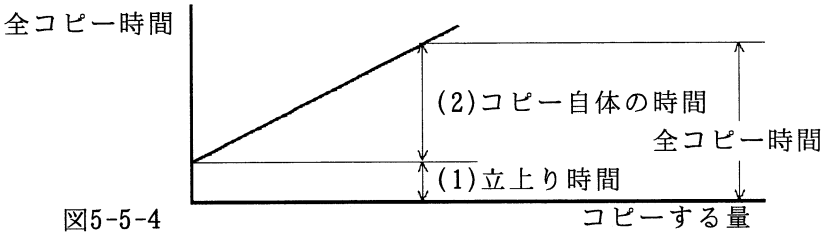


図5-5-4

(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。

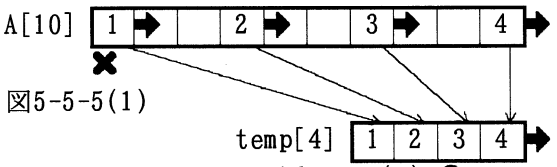


図5-5-5(1) ✖

図5-5-5(3) ○

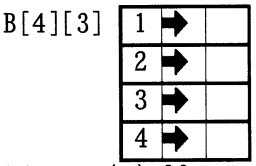


図5-5-5(2) ✖

●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。

行列のｻｲｽﾞがN×Nの場合：(ｺﾋﾟｰ量のｵｰﾀﾞｰ)/(計算量のｵｰﾀﾞｰ) ＝ N^2/N^3 ＝ $1/N$
行列のｻｲｽﾞが(2N)×(2N)の場合：(ｺﾋﾟｰ量のｵｰﾀﾞｰ)/(計算量のｵｰﾀﾞｰ) ＝ $(2N)^2/(2N)^3$ ＝ $1/(2N)$

```
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) ✖

```
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
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) ✖

```
  :
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。

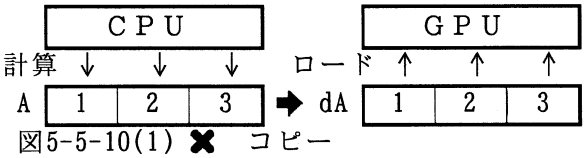


図5-5-10(1) ✖

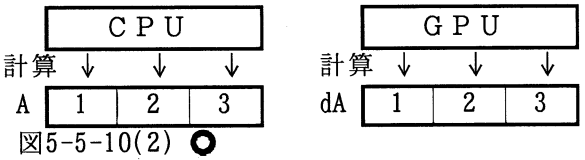


図5-5-10(2) ○

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。

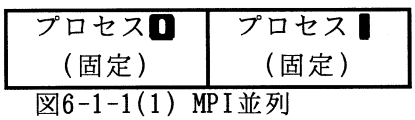


図6-1-1(1) MPI並列

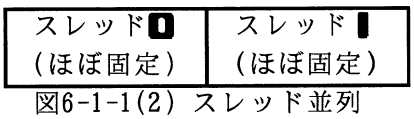


図6-1-1(2) スレッド並列

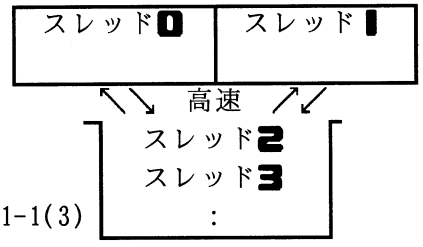


図6-1-1(3)

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)

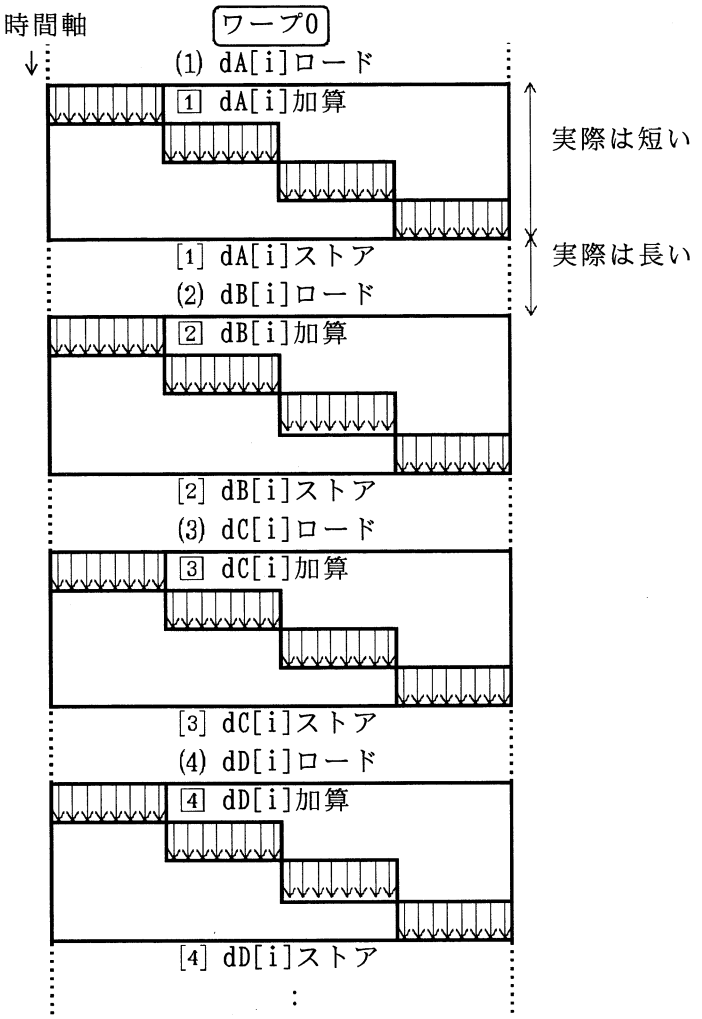


図6-1-2(2)

```
__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)

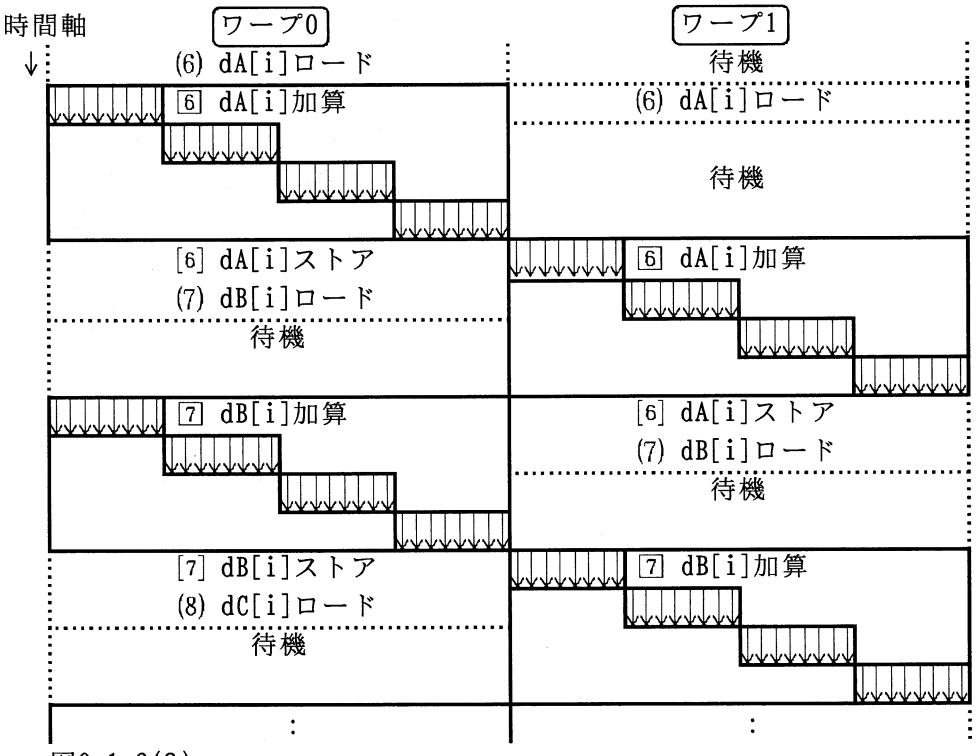


図6-1-3(2)

■ 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。

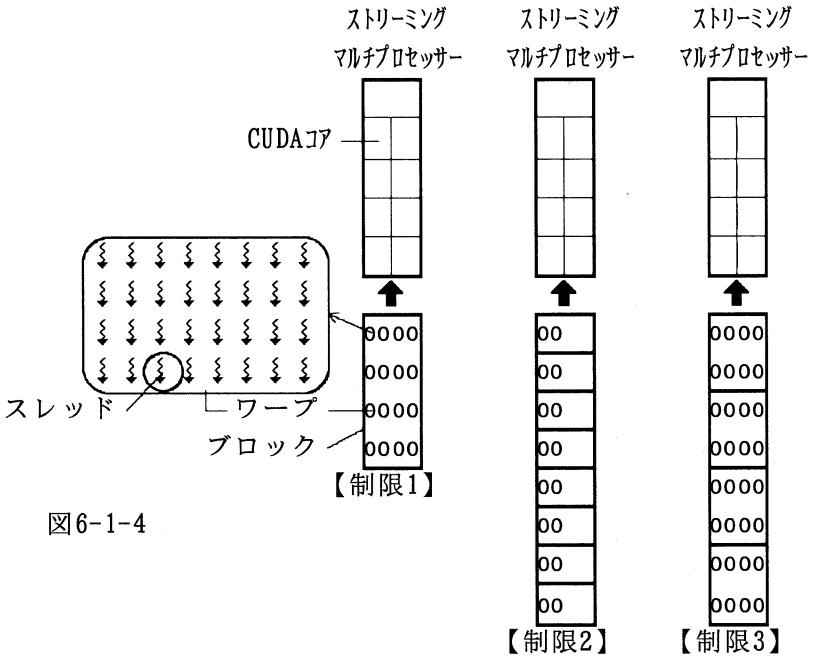


図6-1-4

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×[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。

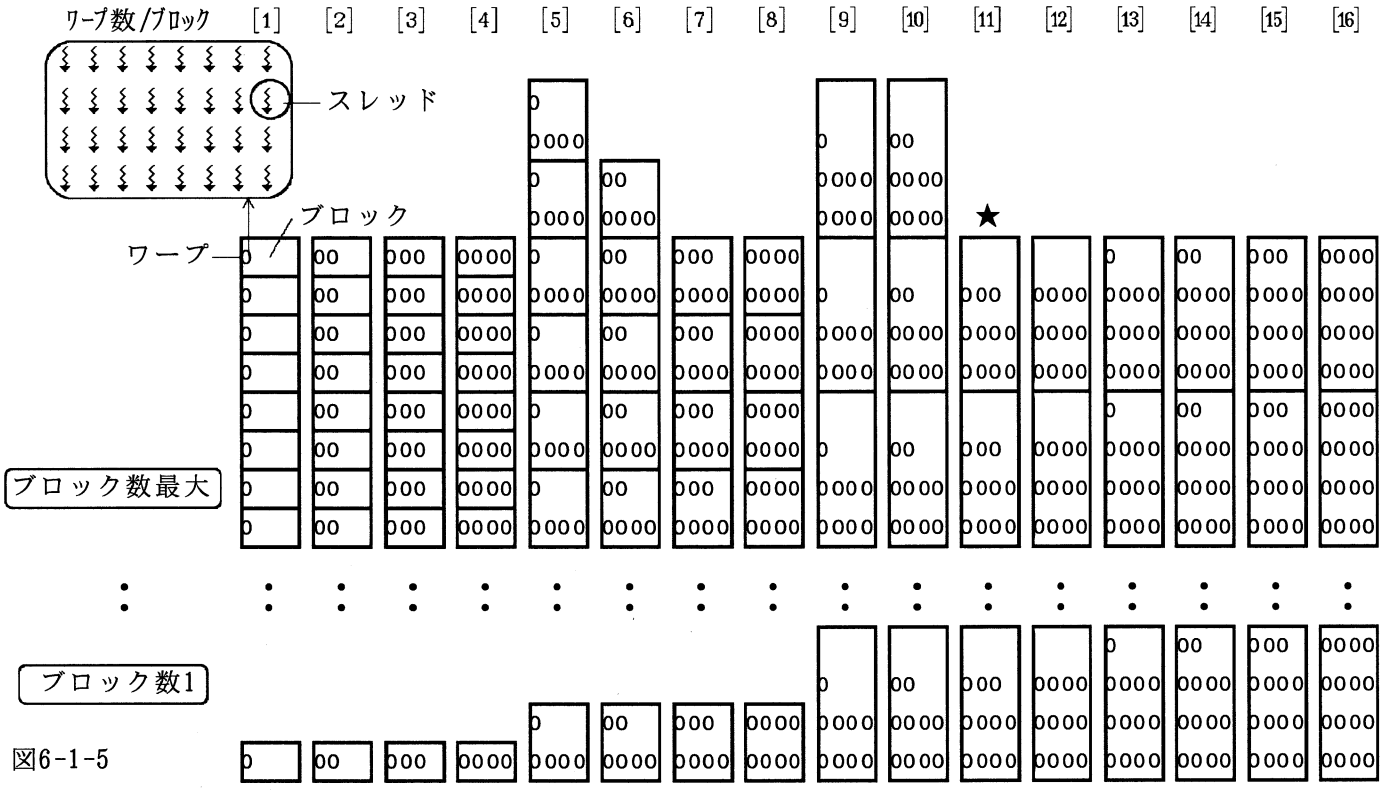


図6-1-5

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÷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÷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つのｽﾄﾘｰﾐﾝｸﾞﾏﾙﾁﾌﾟﾛｾｯｻｰあたりのﾜｰﾌﾟ数が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。

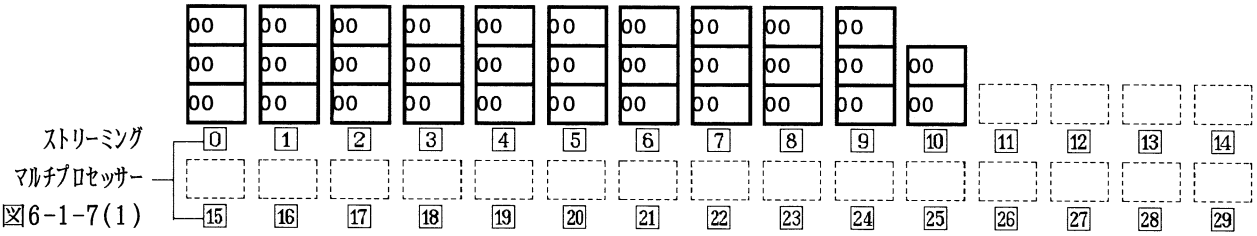


図6-1-7(1)

(2) ケース[2]で1つのｽﾄﾘｰﾐﾝｸﾞﾏﾙﾁﾌﾟﾛｾｯｻｰあたりのﾜｰﾌﾟ数が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)。

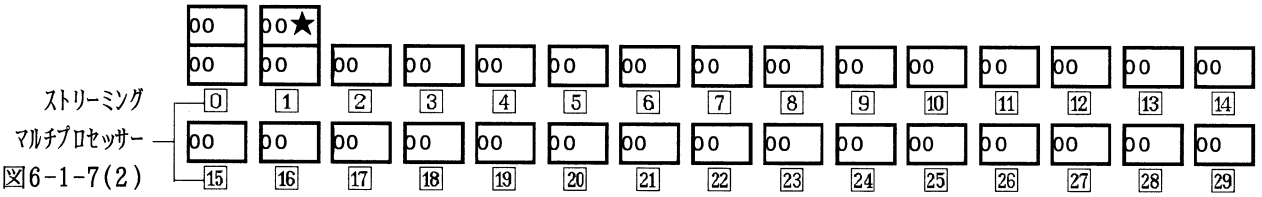


図6-1-7(2)

(3) ケース[2]で1つのｽﾄﾘｰﾐﾝｸﾞﾏﾙﾁﾌﾟﾛｾｯｻｰあたりのﾜｰﾌﾟ数が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。

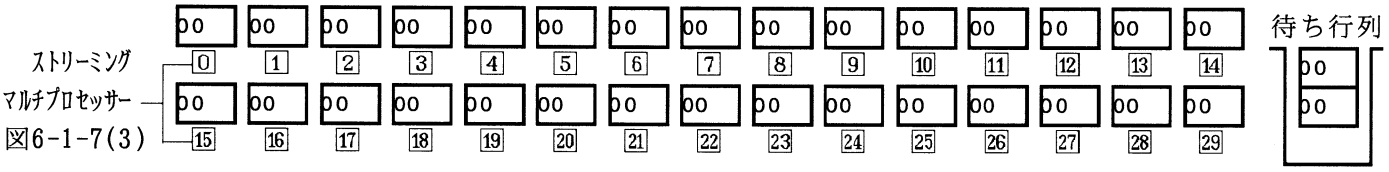


図6-1-7(3)

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÷30＝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)÷32 ＝ 407 （注）割り算の結果は切り捨てます。

[全ワープ数]÷30 ＝ 407÷30 ＝ 13.6個 （1つのｽﾄﾘｰﾐﾝｸﾞ・ﾏﾙﾁﾌﾟﾛｾｯｻｰで処理する平均ﾜｰﾌﾟ数）

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。

ﾜｰﾌﾟ数/ﾌﾞﾛｯｸ	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
ｽﾚｯﾄﾞ数/ﾌﾞﾛｯｸ	32	64	96	128	160	192	224	256	288	320	352	384	416	448	480	512
8	⑧	⑯	㉔	㉜												
ｽﾄﾘｰﾐﾝｸﾞ 7	⑦	⑭	㉑	㉘												
ﾏﾙﾁﾌﾟﾛｾｯｻｰ 6	⑥	⑫	⑱	㉔	㉚											
あたりの 5	⑤	⑩	⑮	⑳	㉕	㉚										
ﾌﾞﾛｯｸ数 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ﾊﾞｲﾄ)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(ﾊﾞｲﾄ)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÷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ﾊﾞｲﾄ)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

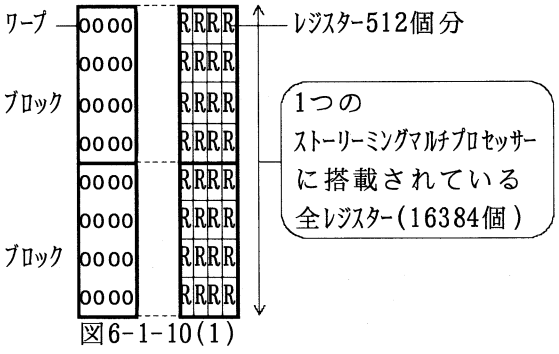


図6-1-10(1)

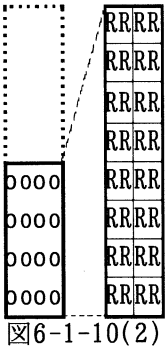


図6-1-10(2)

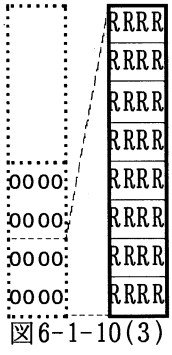


図6-1-10(3)

■ 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ﾊﾞｲﾄ))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。

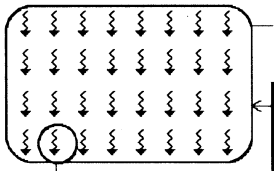
```
__global__ void kernel(float *dA){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  dA[i] = dA[i] + 1.0f;
}
```

図6-1-11 test.cu

```
int main(void){
    :
  kernel<<<100,480>>>(dA);                    ①
    :
}
```

図6-1-12

```
$ nvcc -Xptxas -v -c test.cu ↵                ②
  ～ Used 2 registers , 8+16 bytes smem       ③
```



1つのｽﾄﾘｰﾐﾝｸﾞ・ﾏﾙﾁﾌﾟﾛｾｯｻｰ(SM)上の
(5) ｽﾚｯﾄﾞ( )数:30×32=960個
(6) ﾜｰﾌﾟ(o)数：2×15=30個
(7) ﾌﾞﾛｯｸ(□)数:2個
(8) ﾜｰﾌﾟの占有率:30÷32=94%

使用する資源が少ない場合
↳ ブロック数2

図6-1-13(1)

使用する資源が
資源が多い場合
↳ ブロック数1

図6-1-13(2)

使用する資源が
非常に多い場合
↳ 実行不能

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ﾊﾞｲﾄ個以下)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(ﾊﾞｲﾄ)(16384ﾊﾞｲﾄ以下)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)(ﾊﾞｲﾄ)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。

temp ＝ {((2)＋63)／64}＊64 ← 割り算の結果は切り捨てます。
(10) ＝ {(temp*(3)＋511)／512}＊512 ← 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(ﾊﾞｲﾄ)が表示されます。本来ならば (11)＝(4) で24(ﾊﾞｲﾄ)となるはずですが、512(ﾊﾞｲﾄ)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(ﾊﾞｲﾄ)で、割り当てられるシェアードメモリの容量が512(ﾊﾞｲﾄ)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(ﾊﾞｲﾄ)のときは512(ﾊﾞｲﾄ)、513～1024(ﾊﾞｲﾄ)のときは1024(ﾊﾞｲﾄ)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 ← 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■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■に所属するとします)は、ブロック■の8個の全ワープが②に到達すれば、ブロック■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。

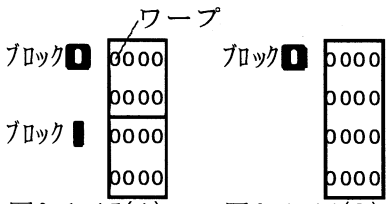


図6-1-15(1)　図6-1-15(2)

```
__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)

1.) Select Compute Capability (click):	1.3 ▼	(1)(設定) Compute Capabilityの「1.3」を選択
2.) Enter your resource usage:		使用する資源の量を設定する
Threads Per Block	480	(2)(設定) ブロックあたりのスレッド数
Registers Per Thread	2	(3)(設定) スレッドあたりのレジスター数
Shared Memory Per Block (bytes)	24	(4)(設定) ブロックあたりのシェアードメモリの容量(バイト)
3.) GPU Occupancy Data is displayed here and in the graphs:		GPU占有率のデータが以下とグラフに表示
Active Threads per Multiprocessor	960	(5)(表示) SMあたりに同時に存在できるスレッド数
Active Warps per Multiprocessor	30	(6)(表示) SMあたりに同時に存在できるワープ数
Active Thread Blocks per Multiprocessor	2	(7)(表示) SMあたりに同時に存在できるブロック数
Occupancy of each Multiprocessor	94%	(8)(表示) 占有率: SMあたりに同時に存在できるワープ数/32 (%)
:(中略)		
Allocation Per Thread Block		ブロックあたりに割り当てられる資源
Warps	15	(9)(表示) ワープ数
Registers	1024	(10)(表示) レジスター数
Shared Memory	512	(11)(表示) シェアードメモリの容量(バイト)
:(以下略)		

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。

(6)で表示された
1つのSM上に同時に存在できるワープ数

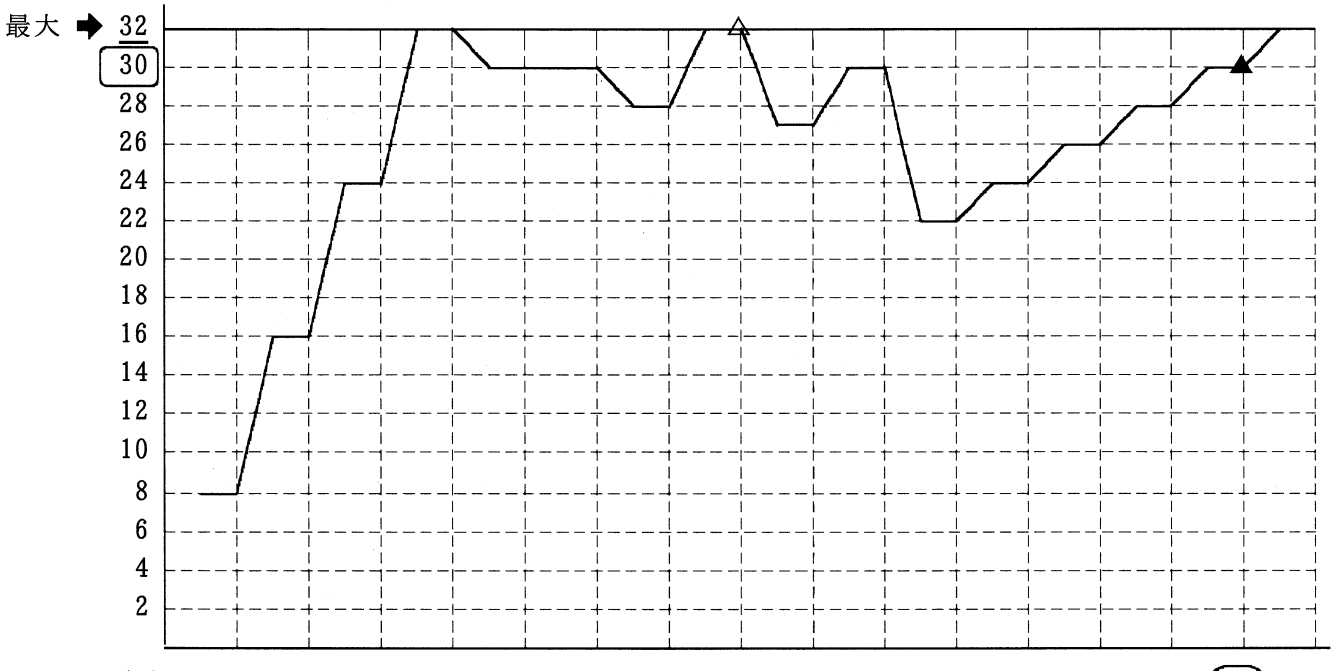


図6-1-16(1)　32　64　96　128　160　192　224　256　288　320　352　384　416　448　480　512
(2)で設定したブロックあたりのスレッド数

(6)で表示された
1つのSM上に同時に存在できるワープ数

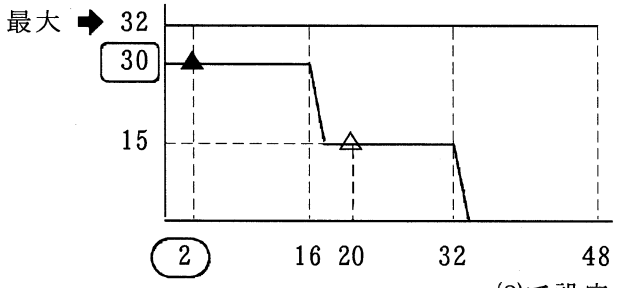


図6-1-16(2)　(3)で設定したスレッドあたりのレジスター数

(6)で表示された
1つのSM上に同時に存在できるワープ数

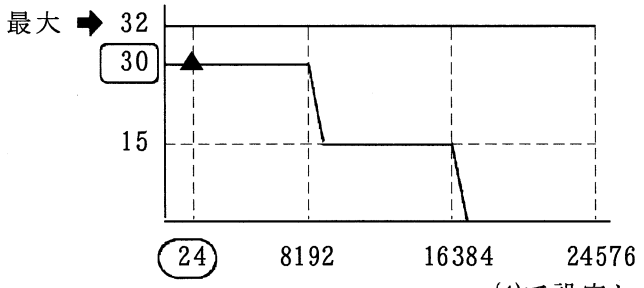


図6-1-16(3)　(4)で設定したブロックあたりのシェアードメモリの容量(バイト)

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。

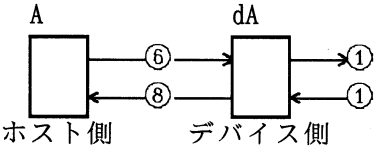


図6-2-1(1)

■ 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。

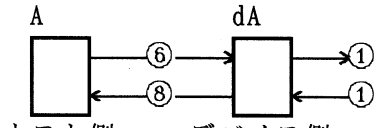


図6-2-4(1)

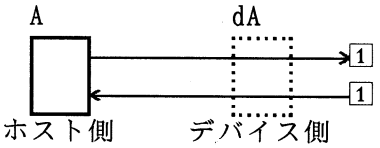


図6-2-4(2)

```
#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)

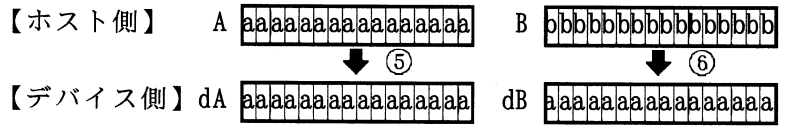


図6-2-7(1)

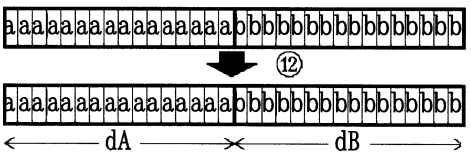


図6-2-7(2)

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。

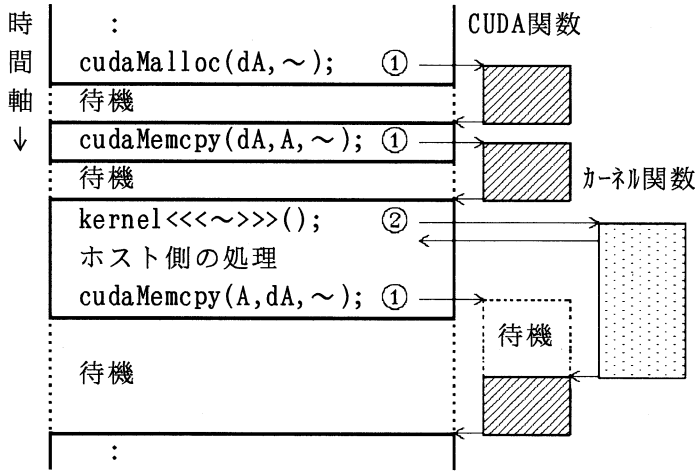


図6-3-1

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)。

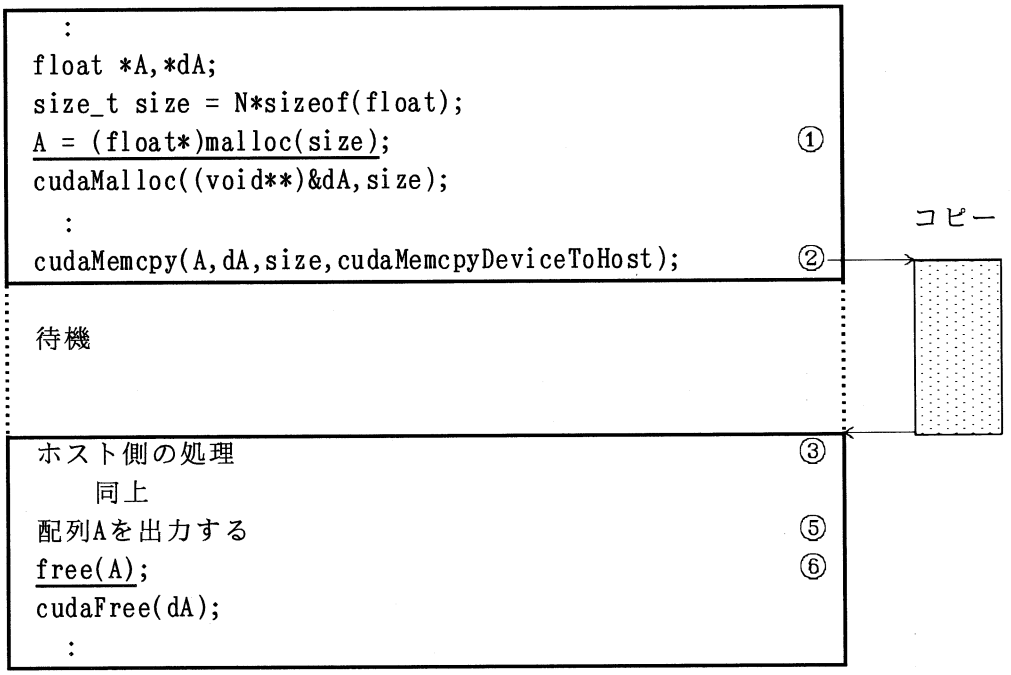


図6-3-2(1)

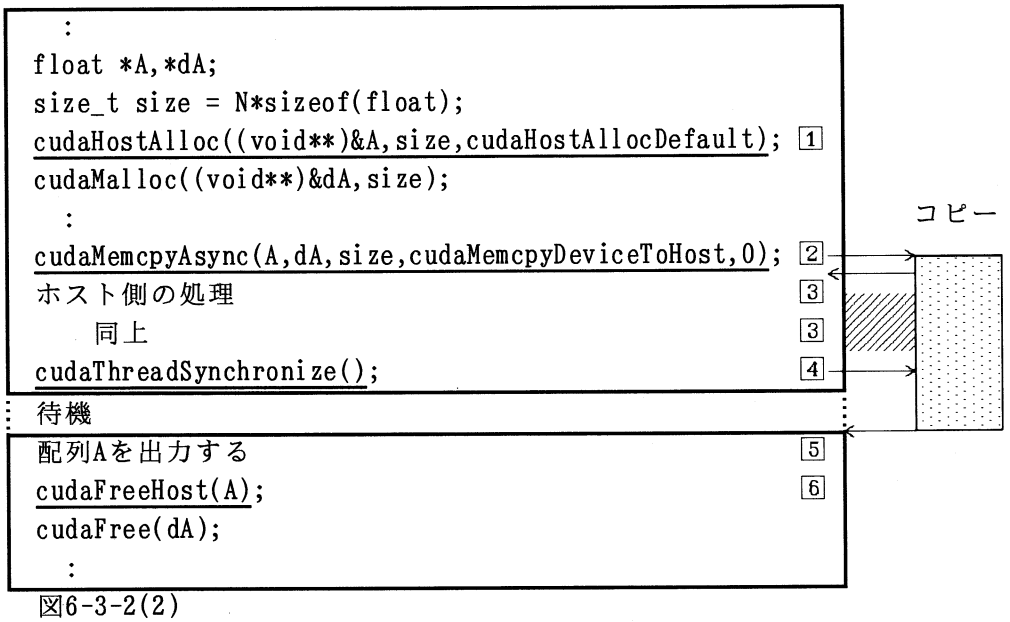


図6-3-2(2)

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ID⓪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ID⓪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ID⓪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ID⓪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)

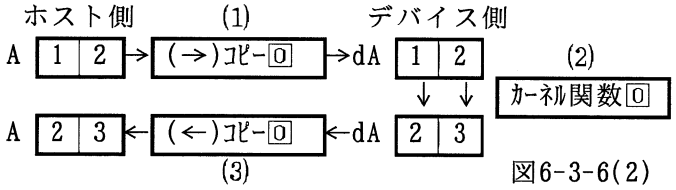


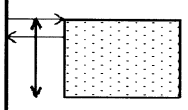
図6-3-6(2)

■ 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ID0)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。

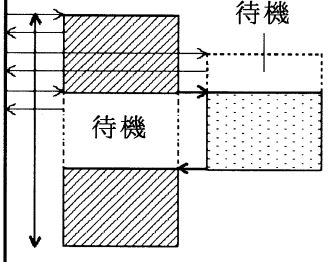
```
  :
cudaMemcpy(dA,A,～); (1)
kernel<<<～>>>(dA);  (2)
ホスト側の処理
    同上
    同上
    同上
cudaMemcpy(A,dA,～); (3)
  :
```

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。

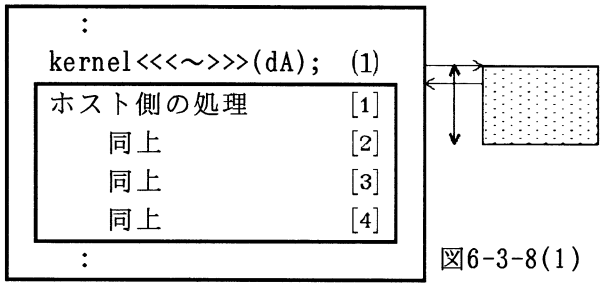


図6-3-8(1)

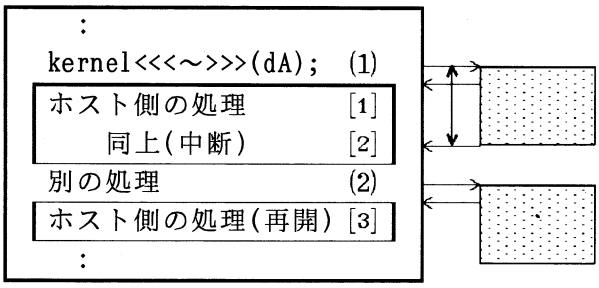


図6-3-8(2)

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)

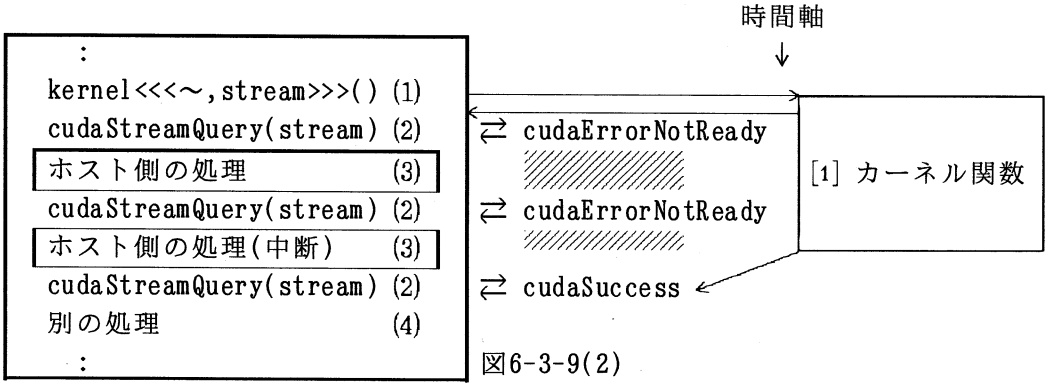


図6-3-9(2)

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ID①、図6-3-10(2)の3つの関数をストリームID②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)。

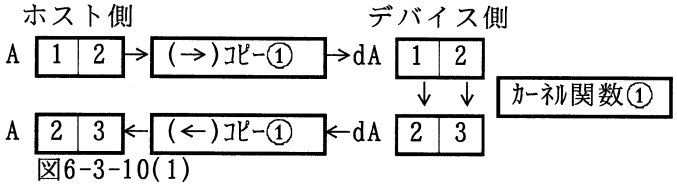


図6-3-10(1)

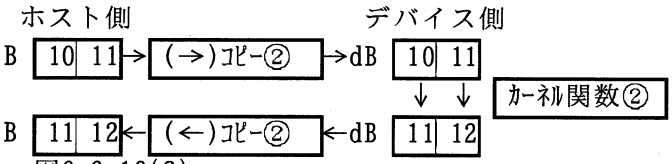


図6-3-10(2)

【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ID①、図6-3-11(2)の3つの関数をストリームID②のように、複数のストリームにすることができます。

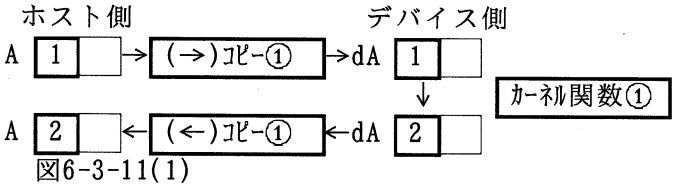


図6-3-11(1)

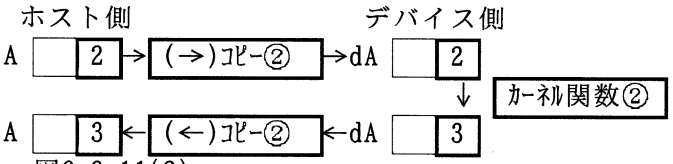


図6-3-11(2)

■ 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。

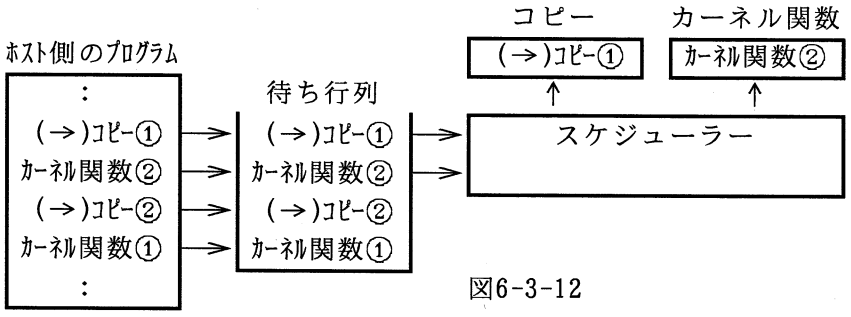
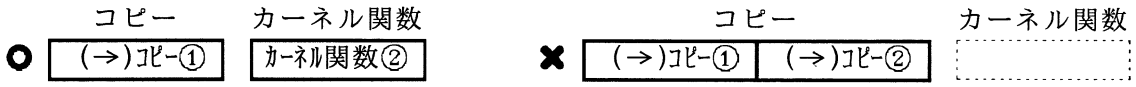


図6-3-12

■ 複数のタスクが同時に実行できる条件

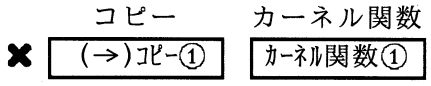
複数のタスクが同時に実行できる条件について説明します。

[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。

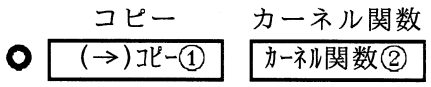


[条件2] ストリームID[0]のタスクは、他のタスクと同時に実行することはできません。

[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「(→)ｺﾋﾟｰ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。

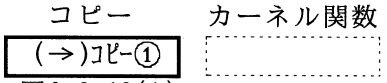


図6-3-13(1)

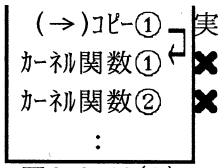


図6-3-13(2)

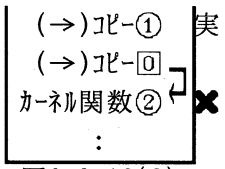


図6-3-13(3)

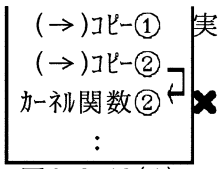


図6-3-13(4)

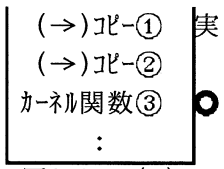


図6-3-13(5)

■ 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ID⓪のタスクは存在しないので、ストリームID⓪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「(→)ｺﾋﾟｰ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ID①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「(→)ｺﾋﾟｰ①」のみが動きます。
(4)「(→)ｺﾋﾟｰ①」の実行が終了したら、図6-3-14(3)のT2に示すように、「(→)ｺﾋﾟｰ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「(→)ｺﾋﾟｰ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ID①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「(→)ｺﾋﾟｰ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。

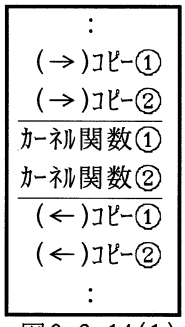


図6-3-14(1)

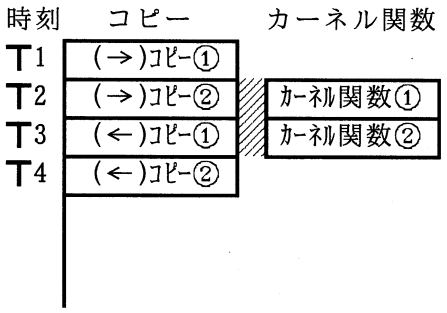


図6-3-14(2)

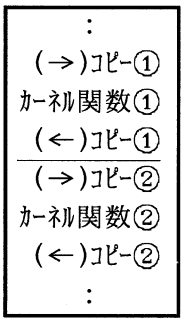


図6-3-15(1)

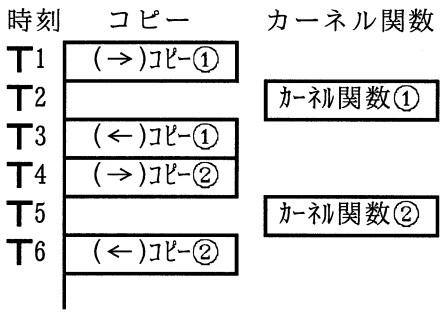


図6-3-15(2)

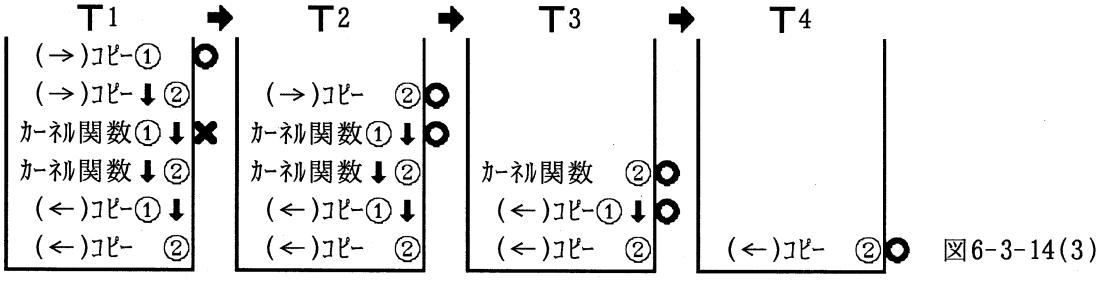


図6-3-14(3)

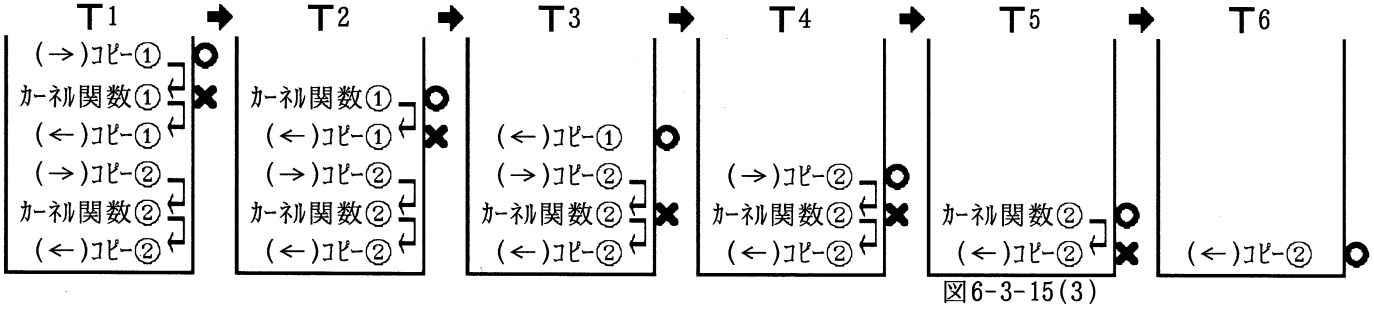


図6-3-15(3)

■ 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

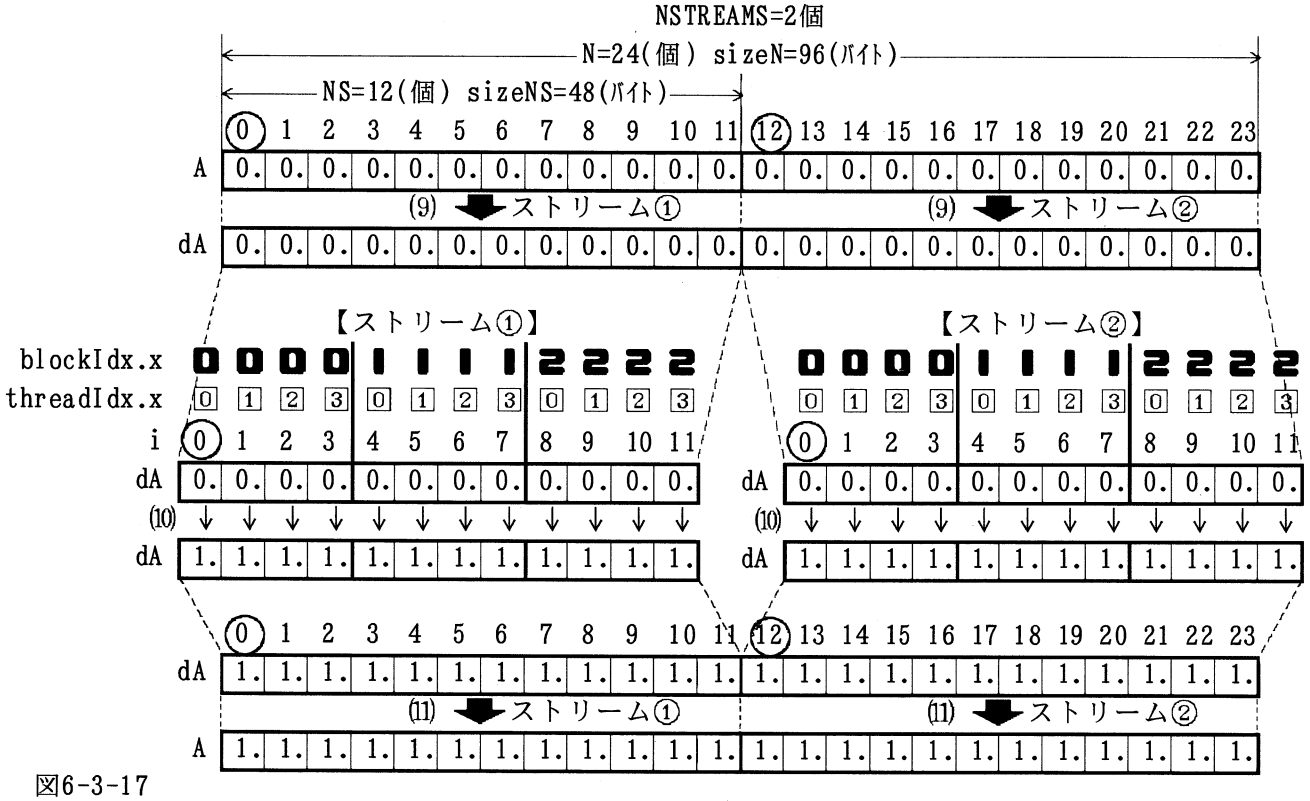


図6-3-17

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。

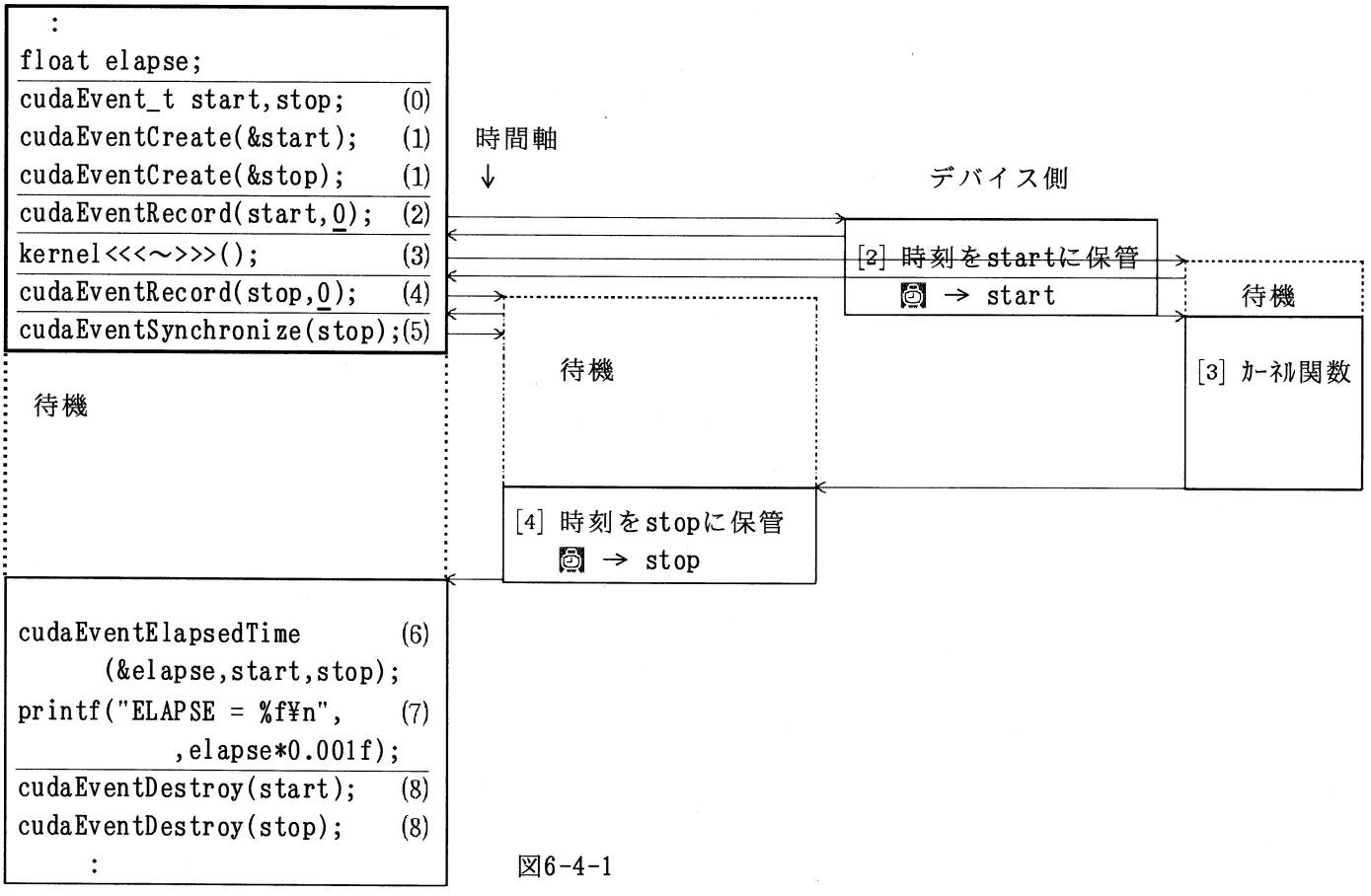


図6-4-1

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

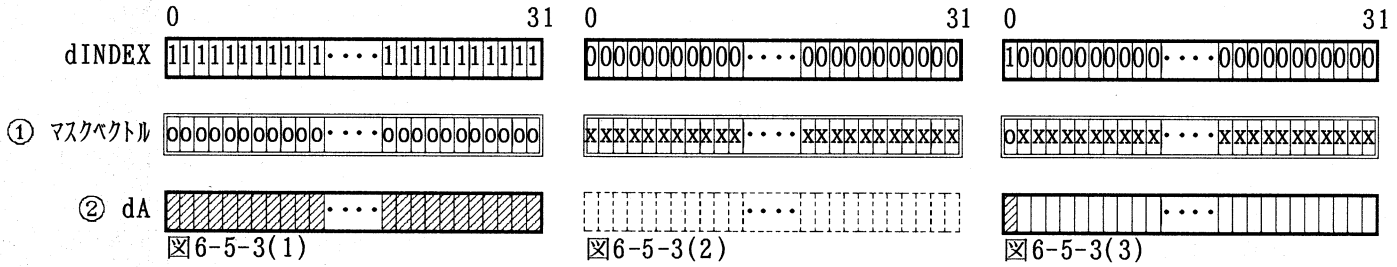


図6-5-3(1)　図6-5-3(2)　図6-5-3(3)

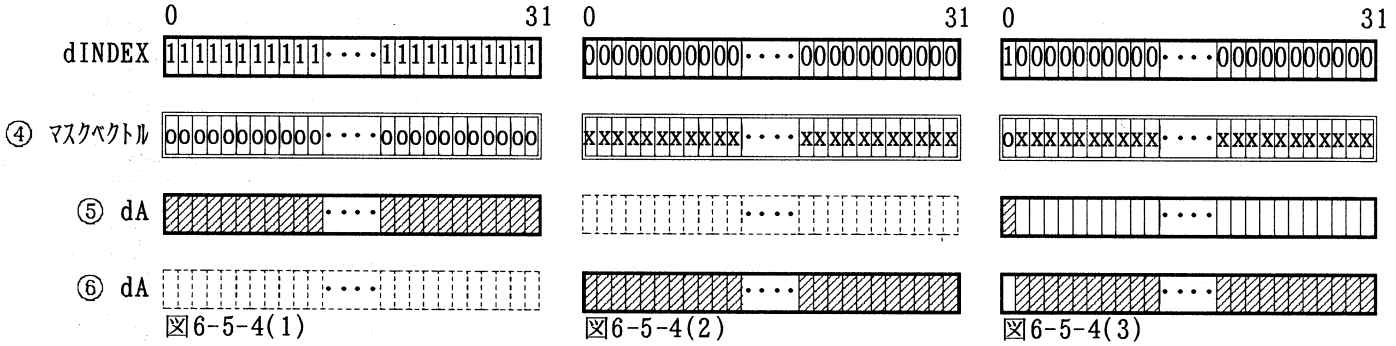


図6-5-4(1)　図6-5-4(2)　図6-5-4(3)

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

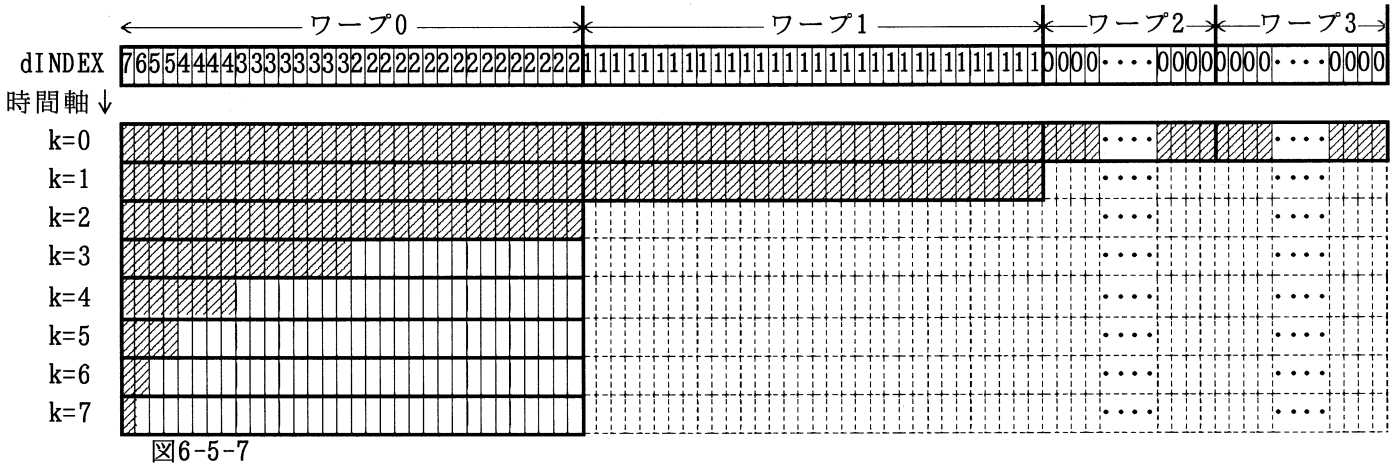


図6-5-7

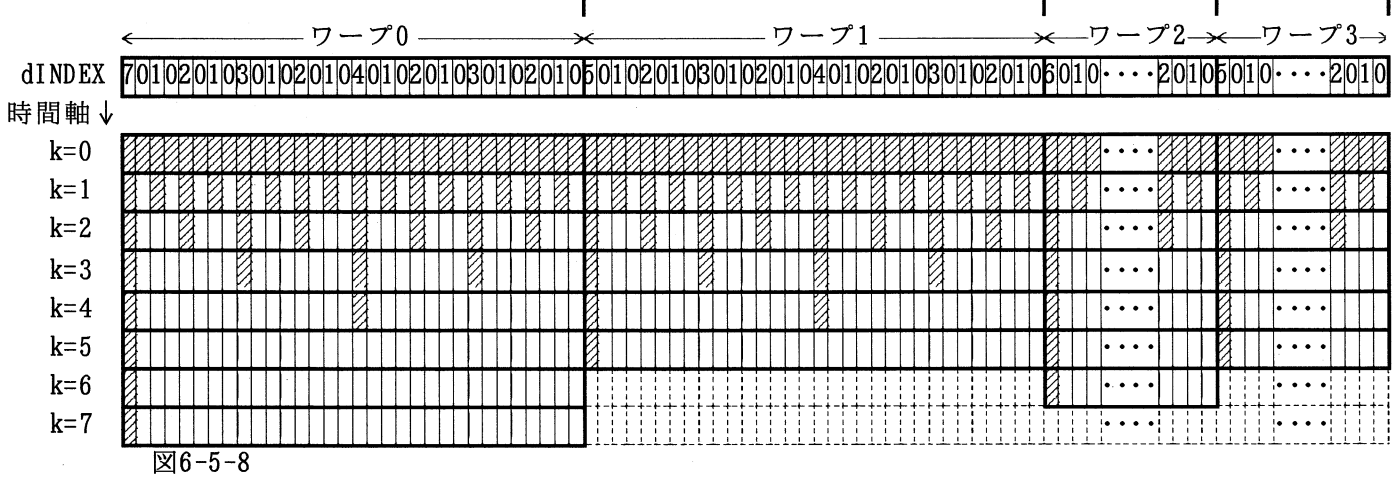


図6-5-8

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=2^3 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) ｱﾝﾛｰﾘﾝｸﾞなし

```
    :
  for(i=0;i<6;i+=2){
    A[i]   = B[i];
    A[i+1] = B[i+1];
  }
    :
```

図6-7-1(2) 2段にｱﾝﾛｰﾘﾝｸﾞ

```
    :
  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段にｱﾝﾛｰﾘﾝｸﾞ

```
    :
  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) ﾙｰﾌﾟの展開

(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。⑧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)

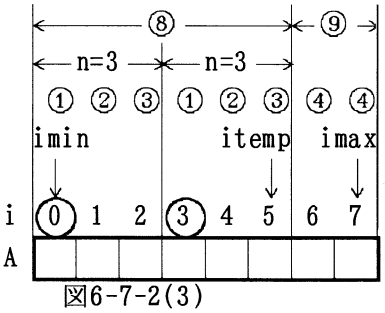


図6-7-2(3)

■ 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「-03」の場合です(２－７節参照)。下記の下線部のように「-02」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、「-02」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 -02 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。

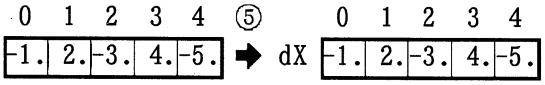


図7-1-1(2)

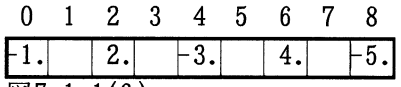


図7-1-1(3)

```
#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)

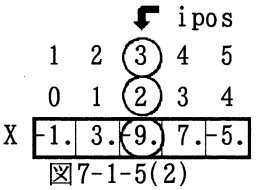


図7-1-5(2)

```
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。

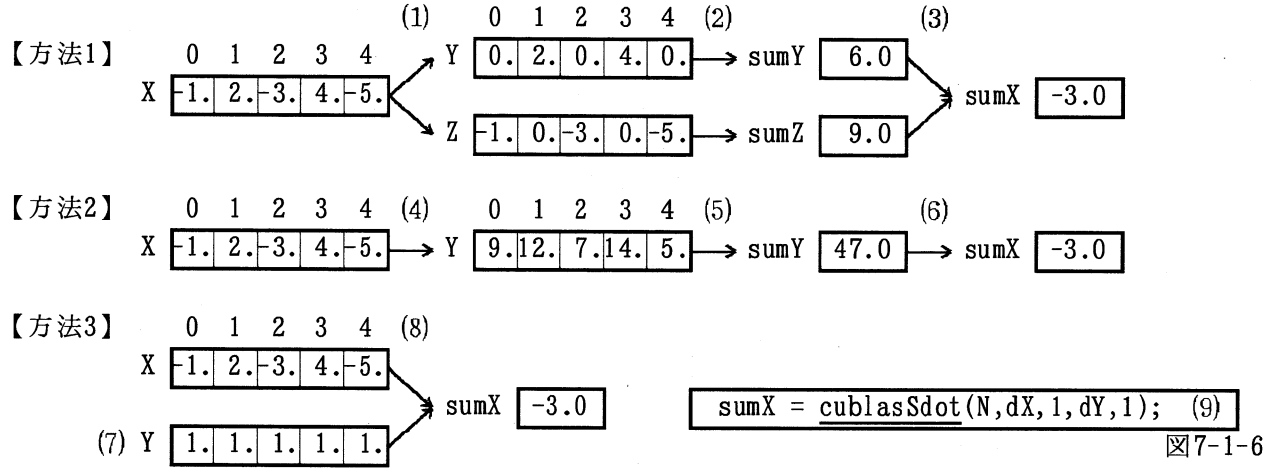


図7-1-6

■ 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。

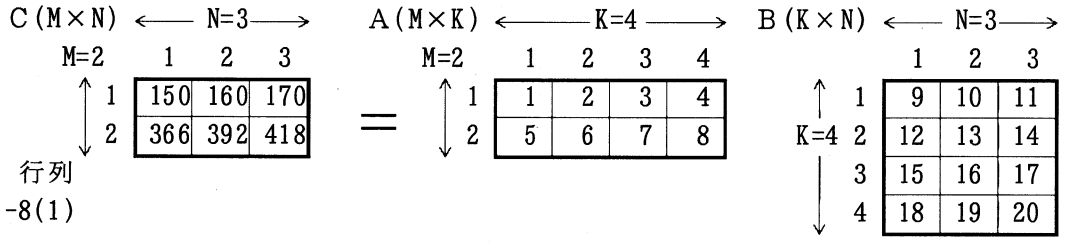
図7-1-8(3)(2)の矢印は、配列内の要素がメモリに並ぶ順番を示します。例えば配列AとdAは、行列Ａの転置になっていることに注意して下さい。また、例えば図7-1-8(3)のLDA(Leading Dimension of A)は、配列Aの2次元目(横方向)の大きさを示します。

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＝βＣ＋αＡＢ を計算します。今回は Ｃ＝ＡＢ＝0･Ｃ＋1･ＡＢを計算するので、⑨でα=1.0,β=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。

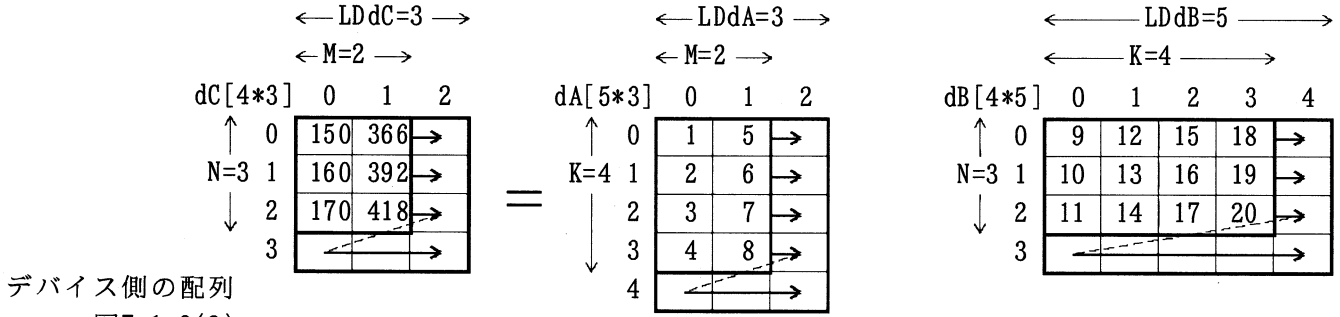
```
#include "cublas.h"
#define M (2)
#define N (3)
#define K (4)
int main(void){
  float A[6][4],B[5][6],C[5][4];                                  ①
  float *dA,*dB,*dC;                                              ②
  int LDA  = 4;int LDB  = 6:int LDC  = 4;                         ③
  int LDdA = 3;int 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);                     ⑥
  cublasAlloc(4*5,sizeof(float),(void**)&dB);                     ⑥
  cublasAlloc(4*3,sizeof(float),(void**)&dC);                     ⑥
  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



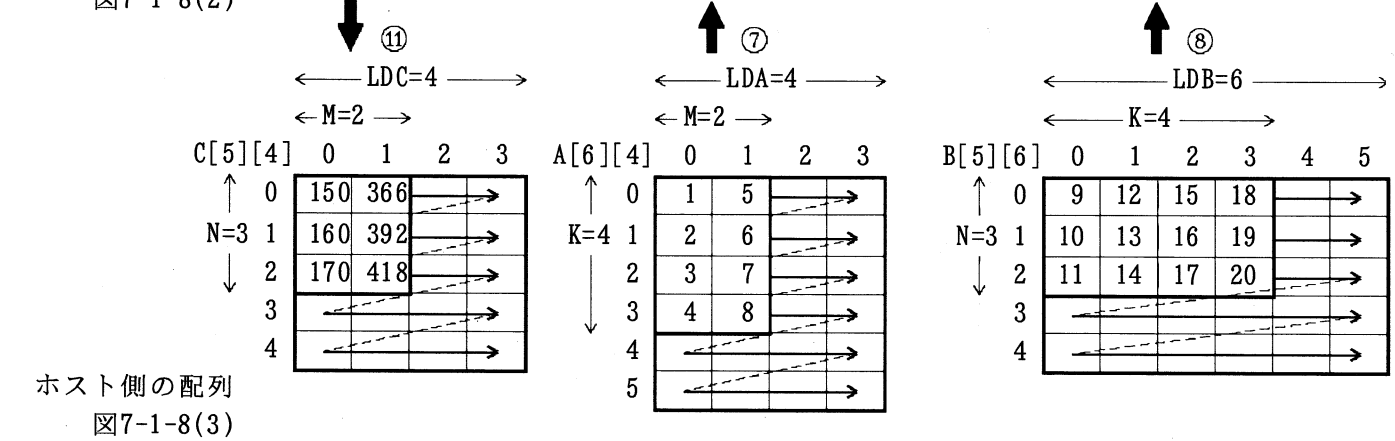
行列

図7-1-8(1)



デバイス側の配列

図7-1-8(2)



ホスト側の配列

図7-1-8(3)

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

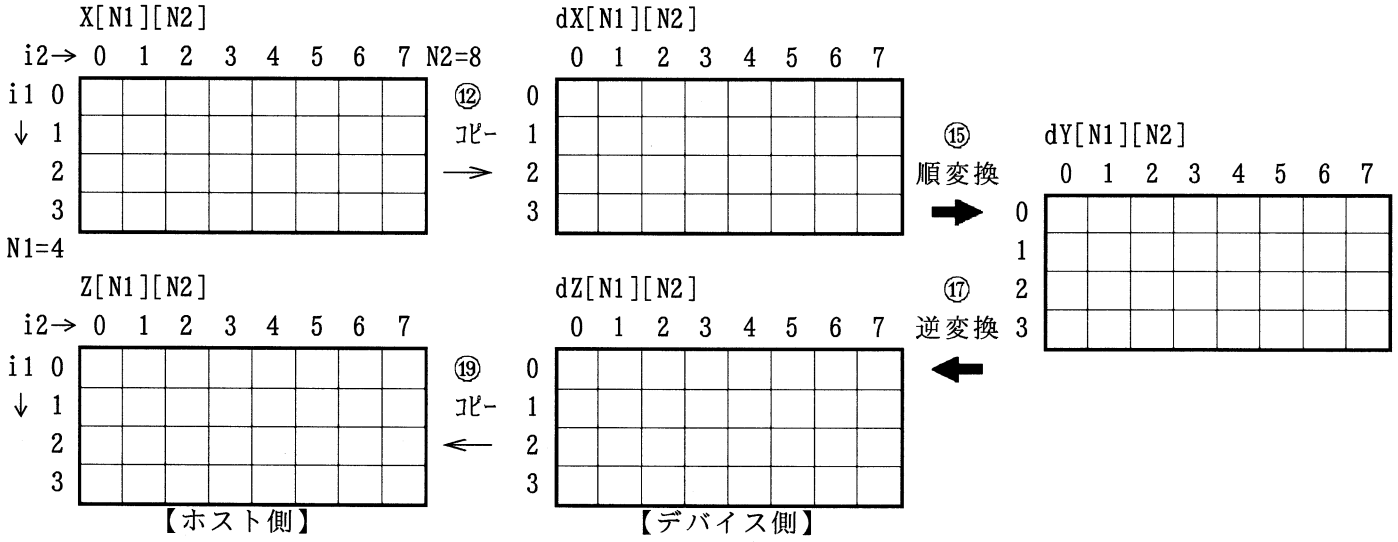


図7-2-3

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「CUDPPAlgorithm」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) dY[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
```

図7-3-1

```
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
  :
```

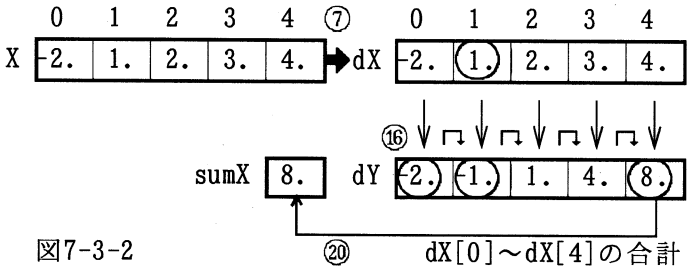


図7-3-2

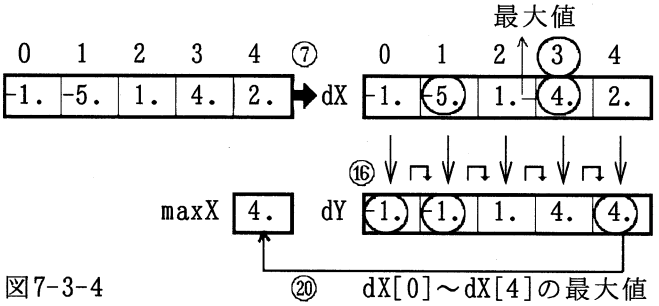


図7-3-4

```
$ 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、CUDA3.2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

●CULAtools (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「SampleCodes and Libraries」
および http://www.nvidia.co.jp/object/tesla_software_jp.html(日本語) の「ｻﾝﾌﾟﾙｺｰﾄﾞおよびﾗｲﾌﾞﾗﾘ」
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ⓘ,ⓙの間に、同じ大きさで反対方向の引力が働き、粒子ⓘが粒子ⓙから受ける力は、「$f_{ij}=X_j-X_i$」(X_i,X_jは粒子ⓘ,ⓙ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。

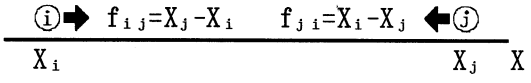


図8-1-1(1)

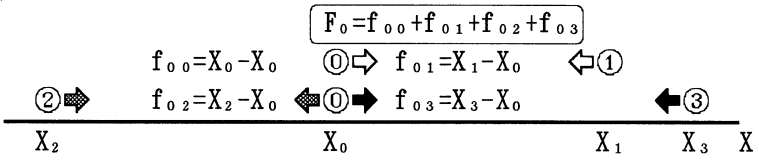


図8-1-1(2)

粒子⓪に働く力：$F_0 = f_{00} + f_{01} + f_{02} + f_{03} = (x_0-x_0) + (x_1-x_0) + (x_2-x_0) + (x_3-x_0)$
粒子①に働く力：$F_1 = f_{10} + f_{11} + f_{12} + f_{13} = (x_0-x_1) + (x_1-x_1) + (x_2-x_1) + (x_3-x_1)$
粒子②に働く力：$F_2 = f_{20} + f_{21} + f_{22} + f_{23} = (x_0-x_2) + (x_1-x_2) + (x_2-x_2) + (x_3-x_2)$
粒子③に働く力：$F_3 = f_{30} + f_{31} + f_{32} + f_{33} = (x_0-x_3) + (x_1-x_3) + (x_2-x_3) + (x_3-x_3)$

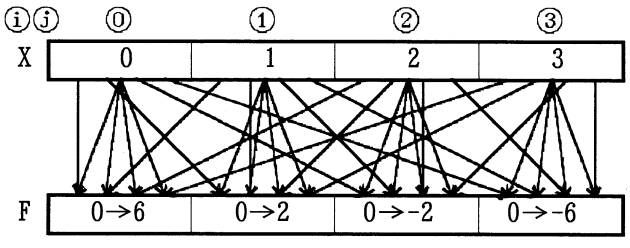
図8-1-1(3)

■ 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ⓘのX座標、F[i]は粒子ⓘ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ⓘ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++){
    X[i] = (float)i;
    F[i] = 0.0f;                          (1)
  }
  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ⓘを求め、(6)のループで、粒子ⓘ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)

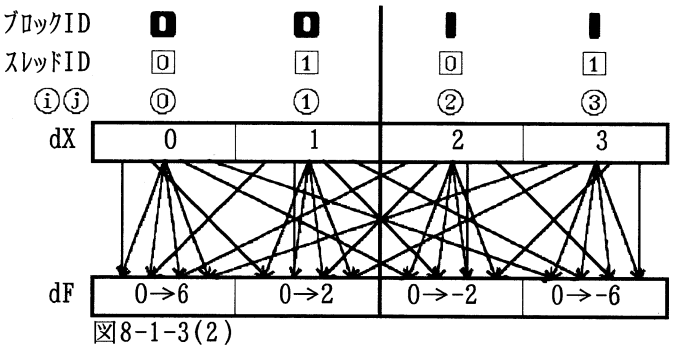


図8-1-3(2)

■ 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++){                          ↑
    fi = fi + (dX[j]-xi);                        (13)
  }                                              ↓
  dF[i] = fi;                                    (14)
}
    :
  kernel<<<2,2>>>(dX,dF);
    :
```

図8-1-4(1)

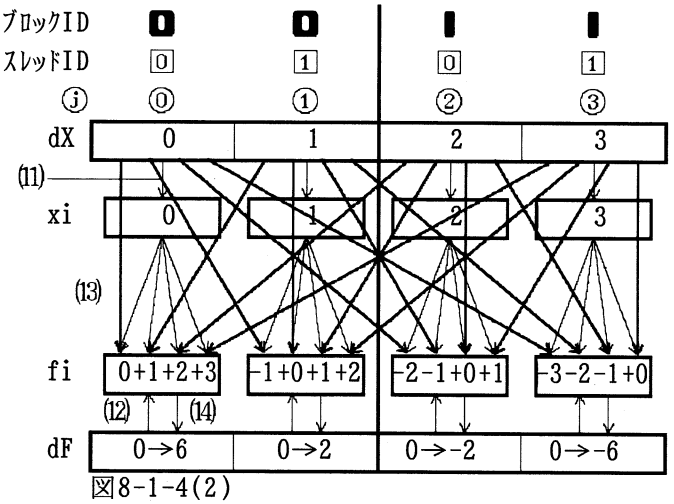


図8-1-4(2)

■ 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)
```

図8-1-5

```
  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)
}
```

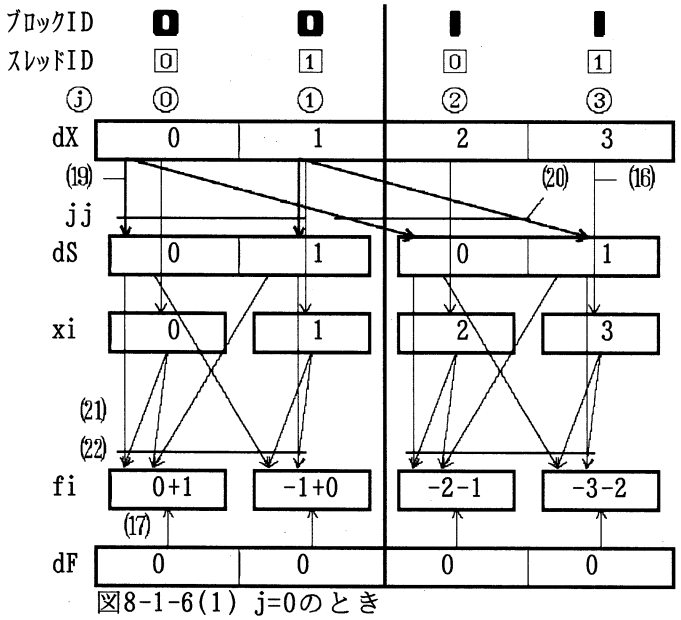


図8-1-6(1) j=0のとき

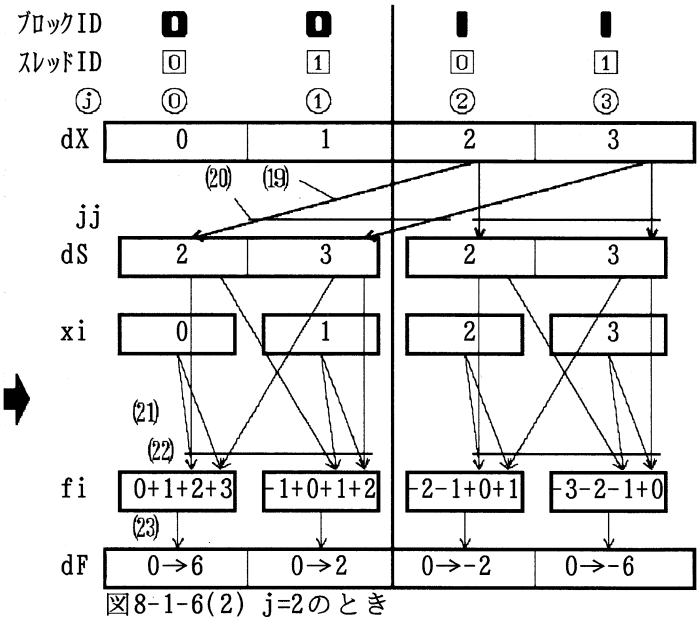


図8-1-6(2) j=2のとき

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■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■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■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)

```
#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)

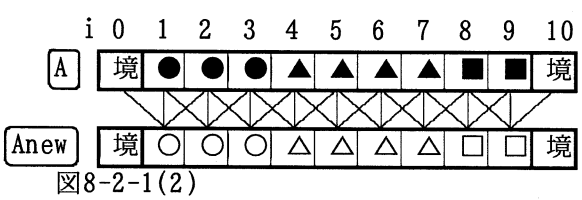


図8-2-1(2)

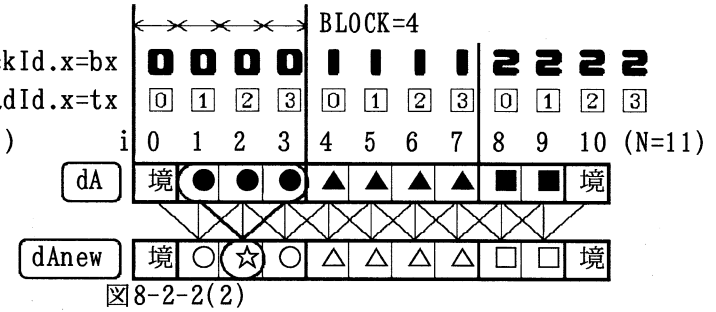


図8-2-2(2)

■ 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☆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)

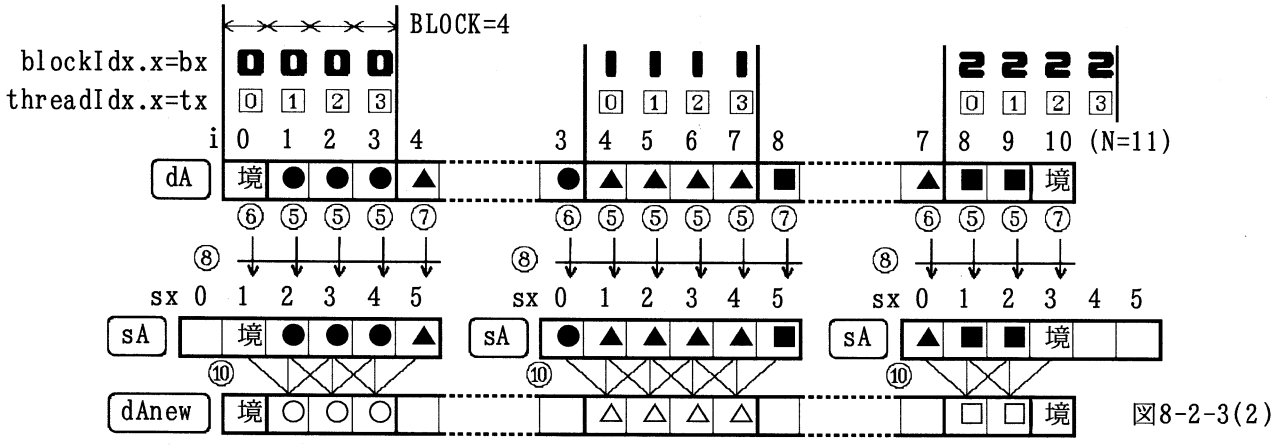


図8-2-3(2)

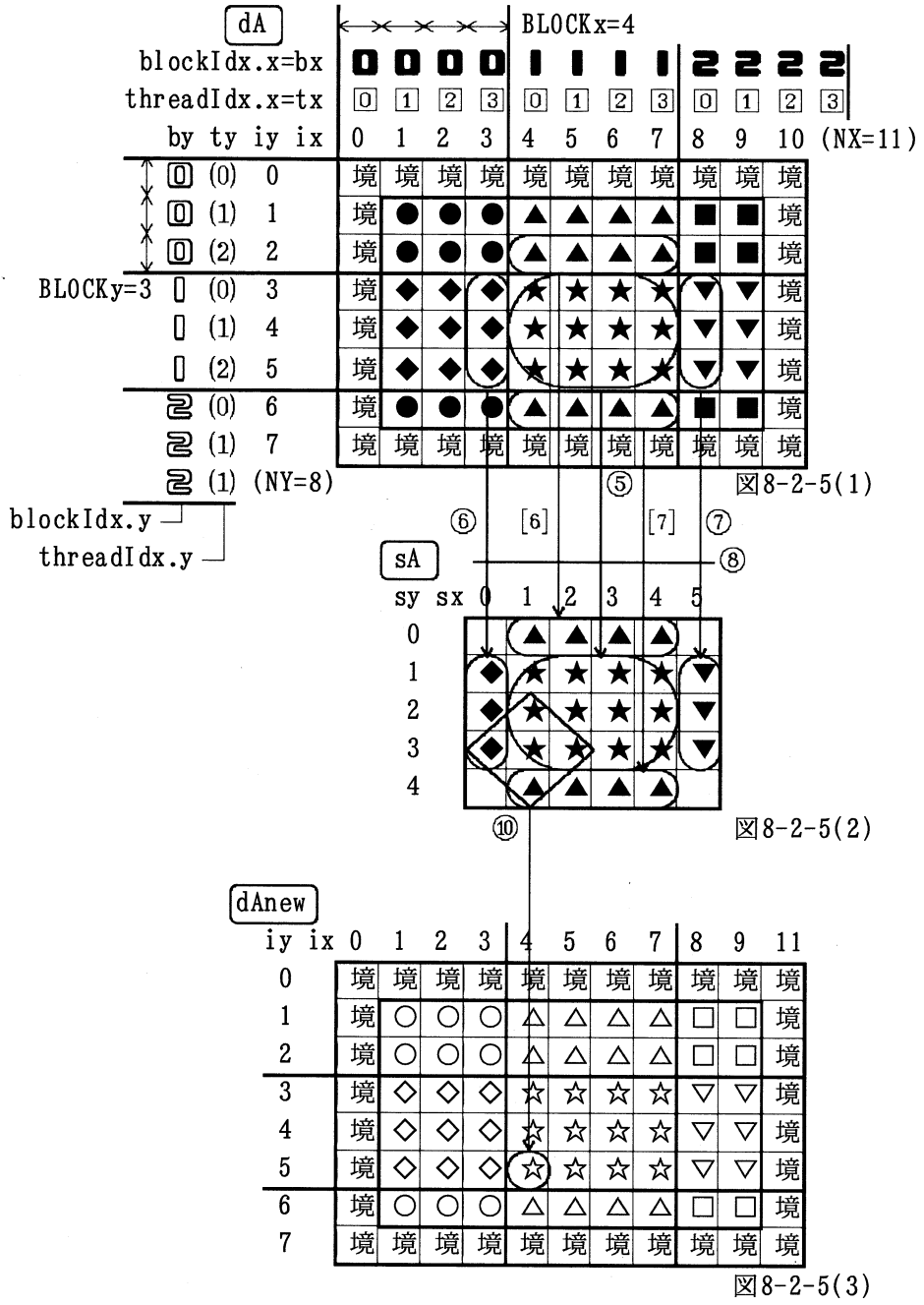
■ シェアードメモリの利用(2次元の場合)

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(▮,▯)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;
  size_t size = NX*NY*sizeof(float);                 ⑬
  cudaMalloc((void**)&dA,size);
  cudaMalloc((void**)&dAnew,size);
    :
  kernel<<<dim3(3,3),dim3(BLOCKx,BLOCKy)>>>
                                  (dA,dAnew);        ⑪
    :
}
```

図8-2-4


dA
BLOCKx=4
blockIdx.x=bx
threadIdx.x=tx
by ty iy ix
(NX=11)
BLOCKy=3
(NY=8)
blockIdx.y
threadIdx.y
境
図8-2-5(1)
sA
sy sx
図8-2-5(2)
dAnew
iy ix
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。

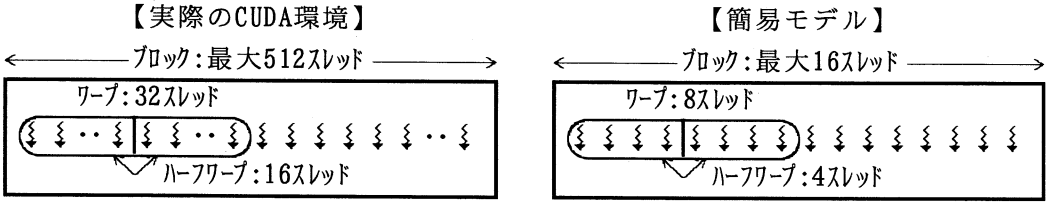
関数	説明資料	プログラムの特徴
reduce0	Reduction #1	ワープ・ダイバージェントの問題があります。
reduce1	Reduction #2	reduce0の改良版で、バンクコンフリクトの問題があります。
reduce2	Reduction #3	reduce1の改良版で、無駄なスレッドの問題があります。
reduce3	Reduction #4	reduce2の改良版です。
reduce4	Reduction #5	reduce3に対し、ループの一部をアンローリングします。
reduce5	Reduction #6	reduce3に対し、ループ全体をアンローリングします。
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ID0～7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。

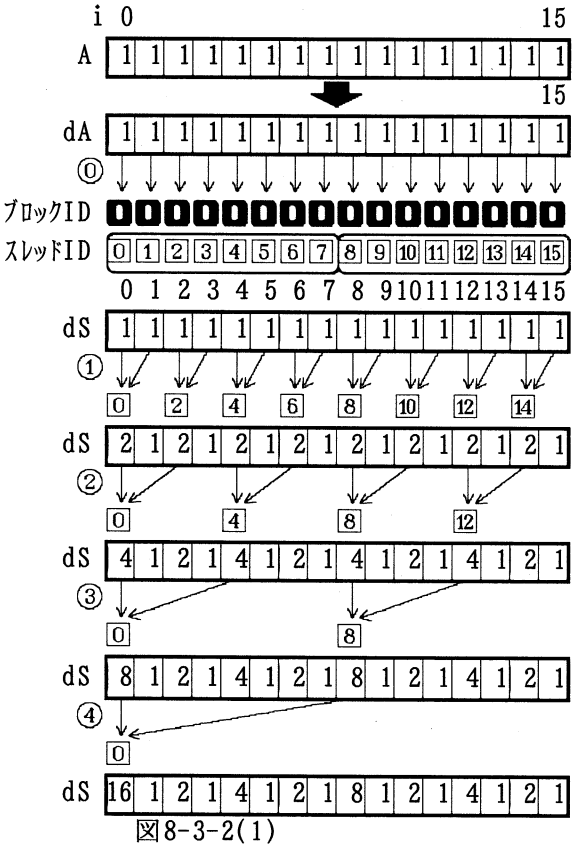


図8-3-2(1)

```
   :
if(ステップⓝを担当するスレッド){
   加算を行う。
}
   :
```
図8-3-3

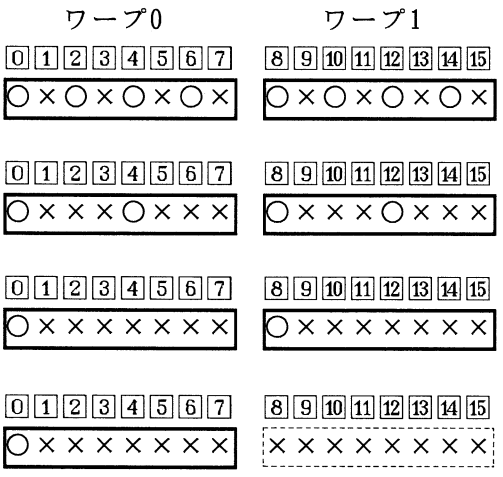


図8-3-2(2)

■ 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。

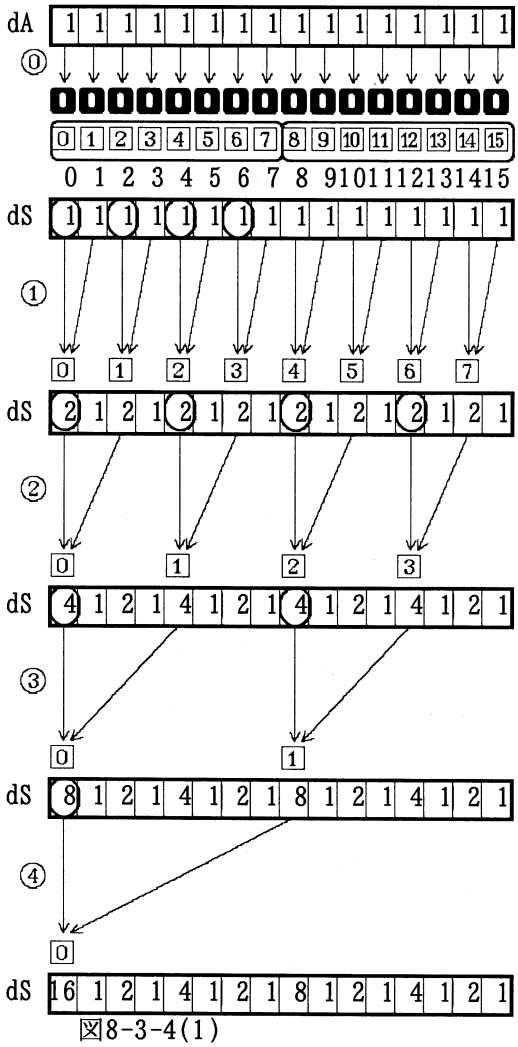


図8-3-4(1)

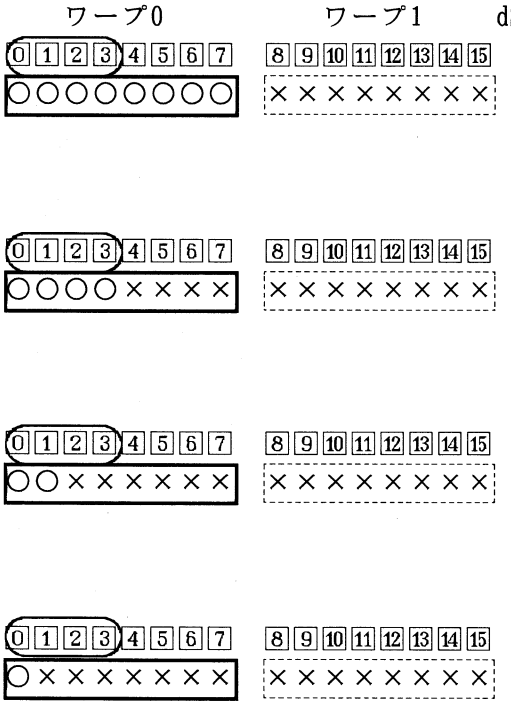


図8-3-4(2)

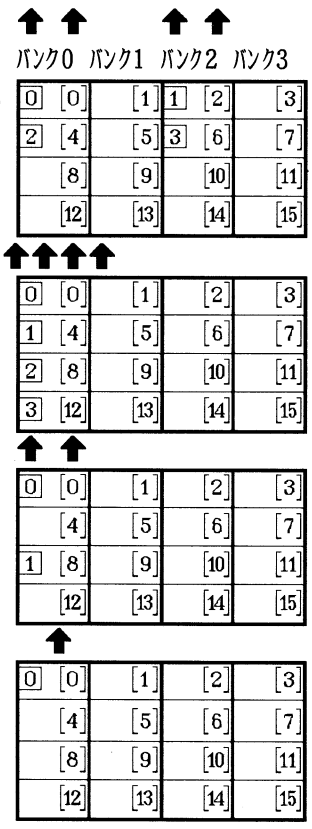


図8-3-4(3)

■ 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❶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。

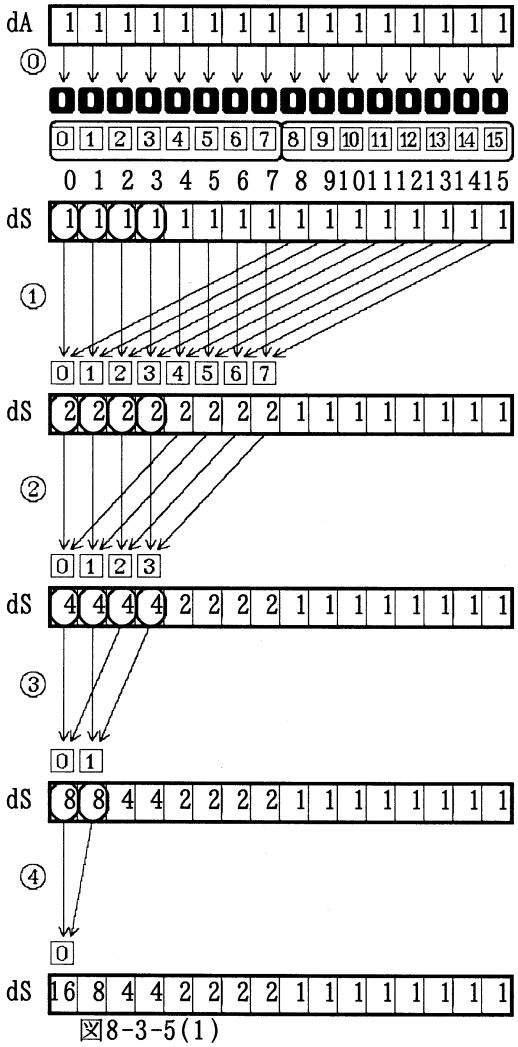


図8-3-5(1)

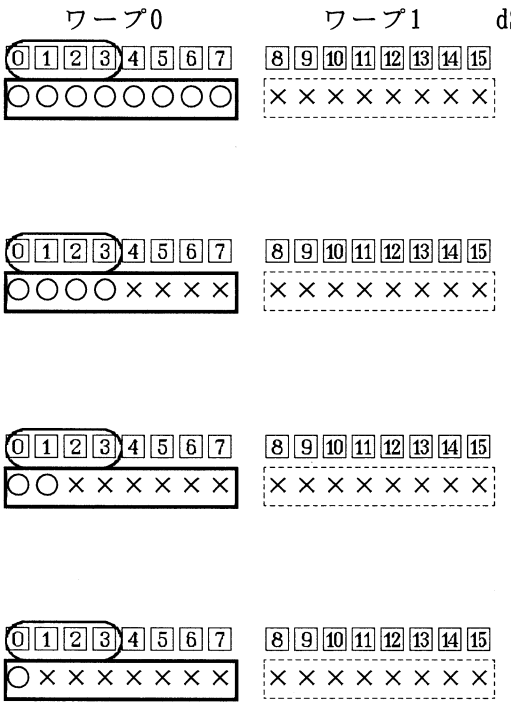


図8-3-5(2)

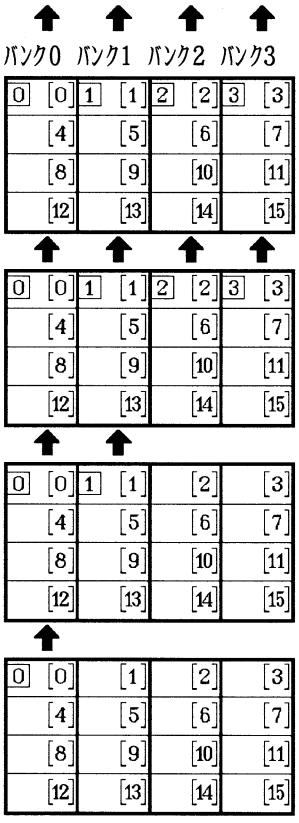


図8-3-5(3)

■ 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=■,▌)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■と▌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■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▌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▌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■ではブロック内の小計「32」がdS[0]に入り、ブロック▌ではブロック内の小計「7」がdS[0]に入ります。
●ブロック■,▌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[■],dB[▌]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=■)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■の各スレッドは、以下の処理を行います。
- ブロック■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■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■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■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[■]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。

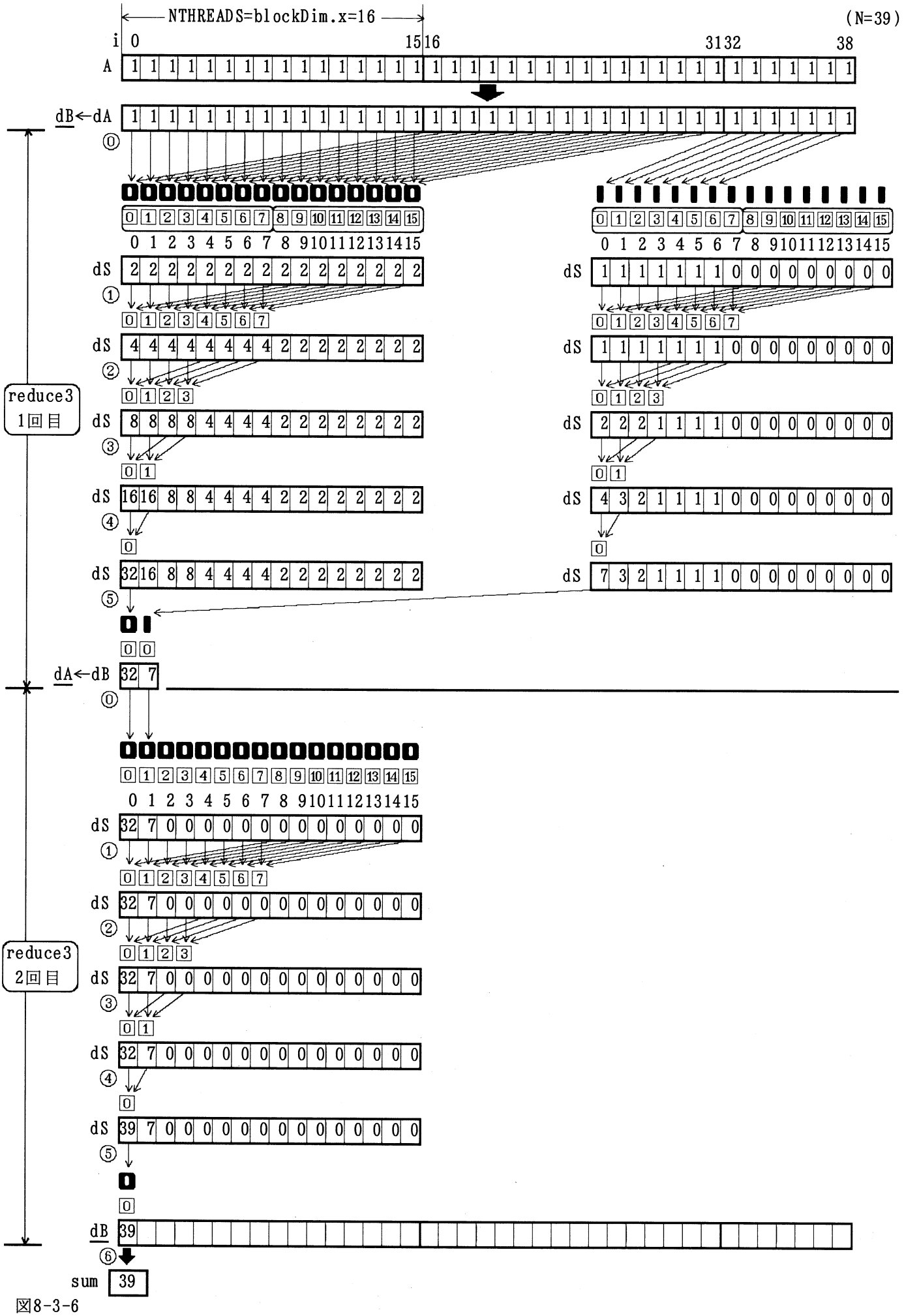

NTHREADS=blockDim.x=16
(N=39)
i
A
dB←dA
dS
reduce3
1回目
dA←dB
reduce3
2回目
dB
sum
39

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Ⓞ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Ⓞ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■のスレッド0ではi=0，ブロック▌のスレッド0ではi=32となります。
●図8-3-6(上半分)のⓄ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▌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▌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■,▌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[■],dB[▌]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■のスレッド□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[■]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]  実際は256や512など
#define NTHREADS (16)   [2]  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){                                          ↑
    temp = dA[i];                                   |
  }else{                                            [6]
    temp = 0.0f;                                    |
  }                                                 ↓
  if(i+blockDim.x<n){                               ↑
    temp = temp + dA[i+blockDim.x];                 [7]
  }                                                 ↓
  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]
}
```

図3-8-7

```
int main(void){
  int n,NBLOCKS;
  float A[N],*dA,*dB,*dTEMP,sum;
  if(N<=0){                                         ↑
    printf("ERROR¥n");                              |
    return(-1);                                     [15]
  }                                                 ↓
  配列Aに値を設定します。                           [16]
  size_t size = N*sizeof(float);                    ↑
  cudaMalloc((void**)&dA,size);                     |
  cudaMemcpy(dA,A,size,                             [17]
             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                                         [26]
      = (n+(NTHREADS*2)-1)/(NTHREADS*2);            [26]
    dTEMP = dA;                                     ↑
      dA = dB;                                      [27]
      dB = dTEMP;                                   ↓
  }
  cudaMemcpy(&sum,dB,sizeof(float),                 [28]
             cudaMemcpyDeviceToHost);               [28]
  printf("sum = %f¥n",sum);
    :
```

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)=(▮,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

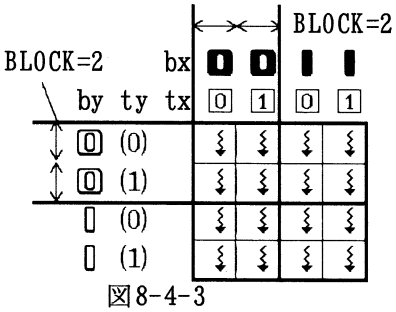


図8-4-3

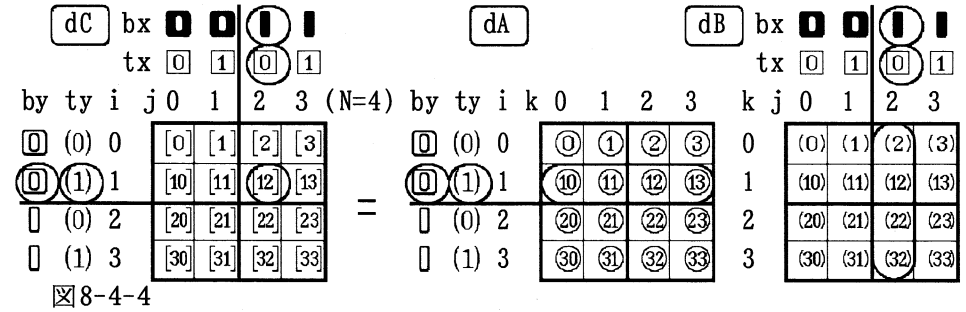


図8-4-4

■ 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(▮,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、(▮,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。

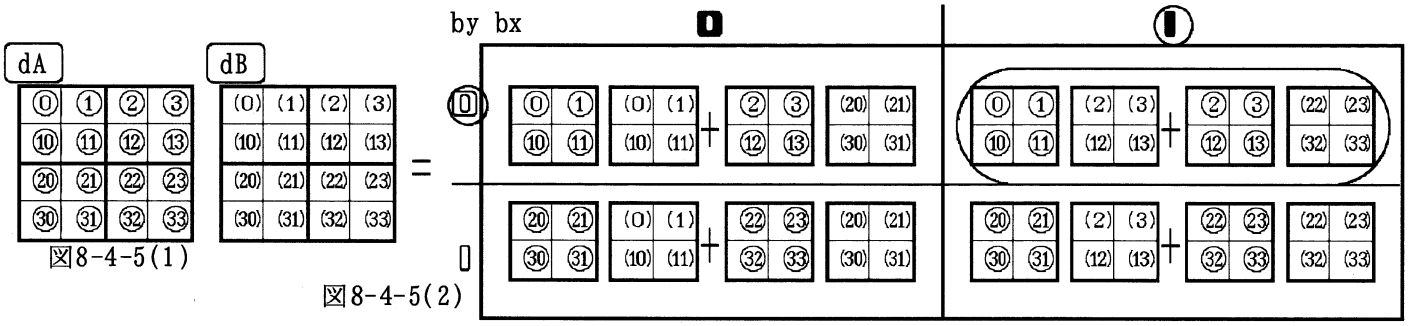


図8-4-5(1)　　図8-4-5(2)

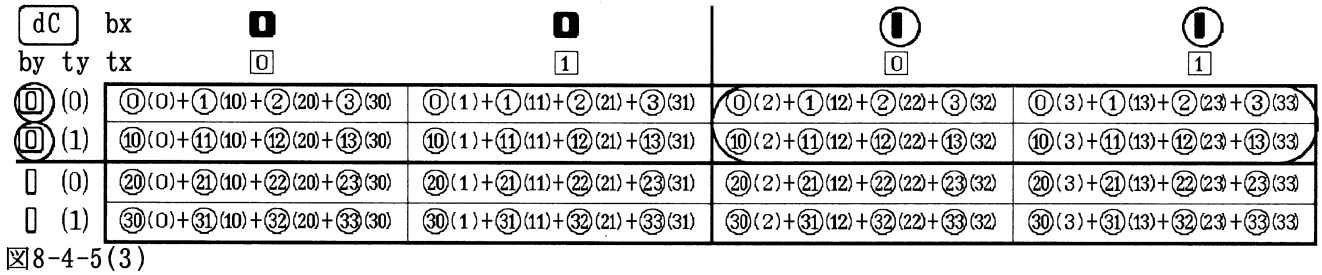


図8-4-5(3)

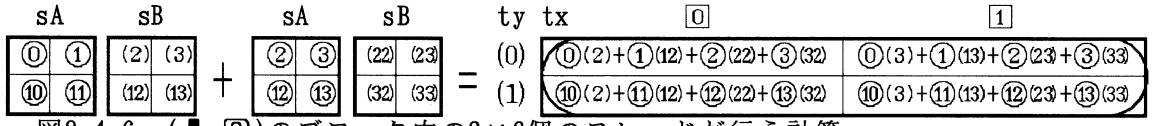


図8-4-6　(▮,0)のブロック内の2×2個のスレッドが行う計算

■ 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(▮,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(▮,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(▮,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(▮,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 田田 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(▮,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++){            [6]
      sum = sum + sA[ty][kk]*sB[kk][tx];
    }
    __syncthreads();                        [7]
  }
  dC[i][j] = sum;                           [8]
}
int main(void){
    :
  kernel<<<dim3(2,2),dim3(2,2)>>>();
    :
```

図8-4-7

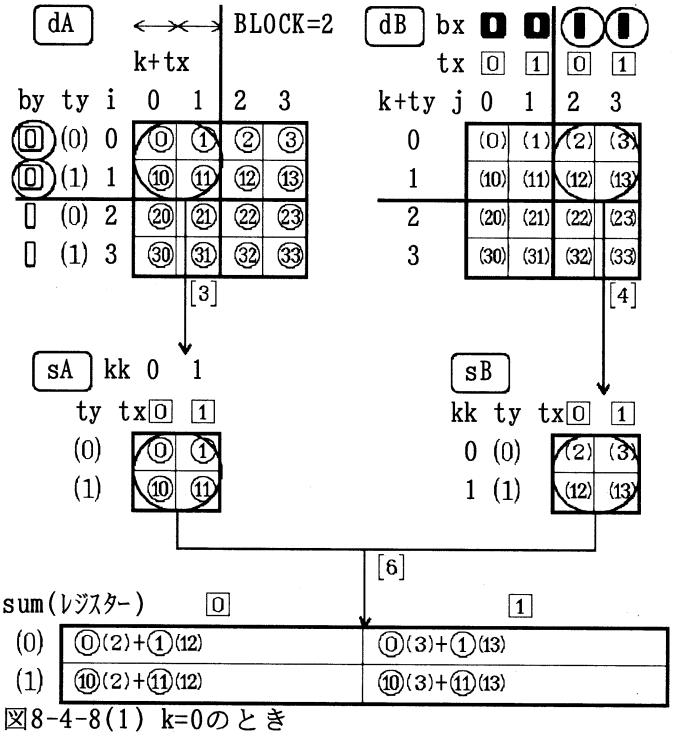


図8-4-8(1) k=0のとき

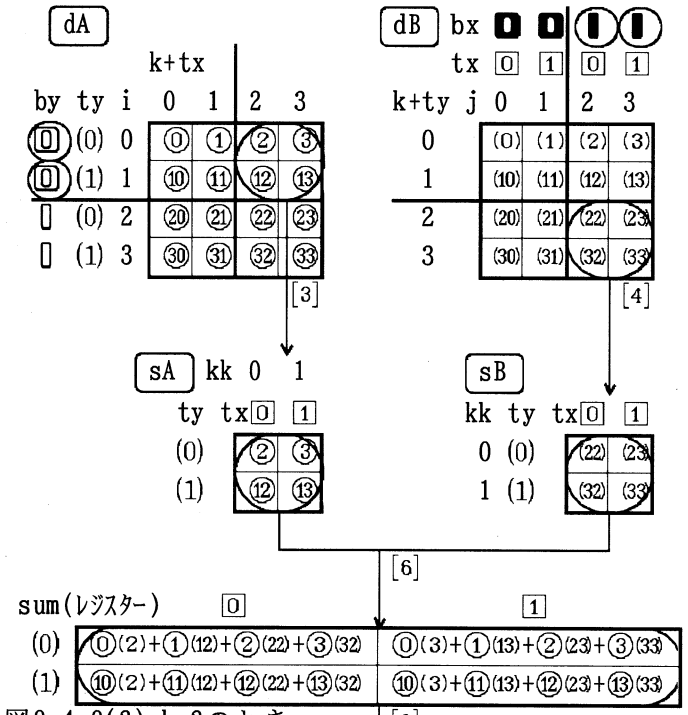


図8-4-8(2) k=2のとき

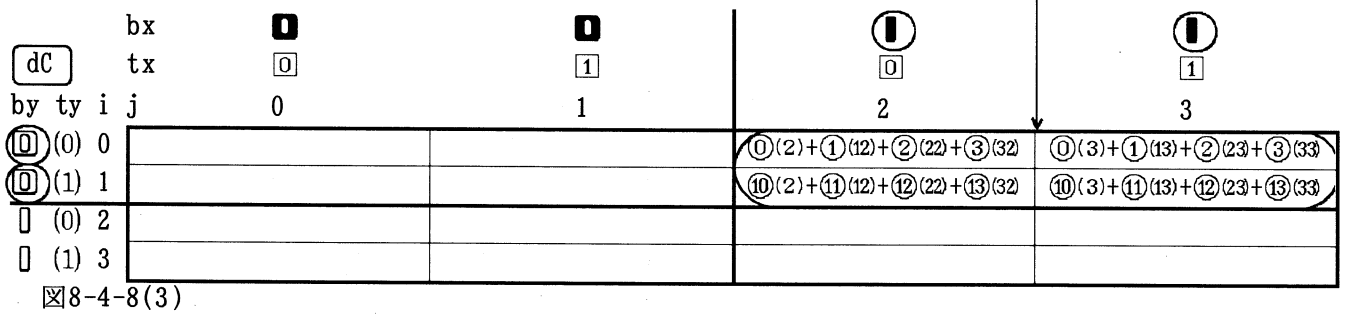


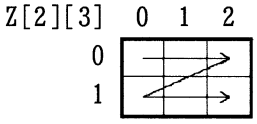
図8-4-8(3)

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

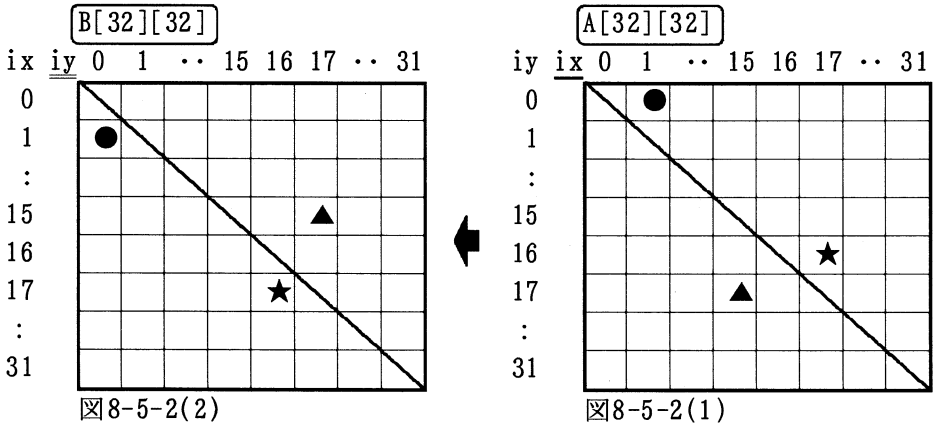


図8-5-2(2)　　図8-5-2(1)

■ 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(■,▌)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,▯)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(▌,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

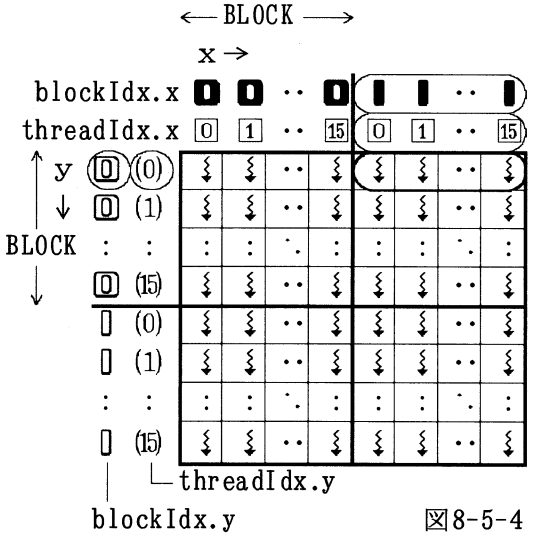


図8-5-4

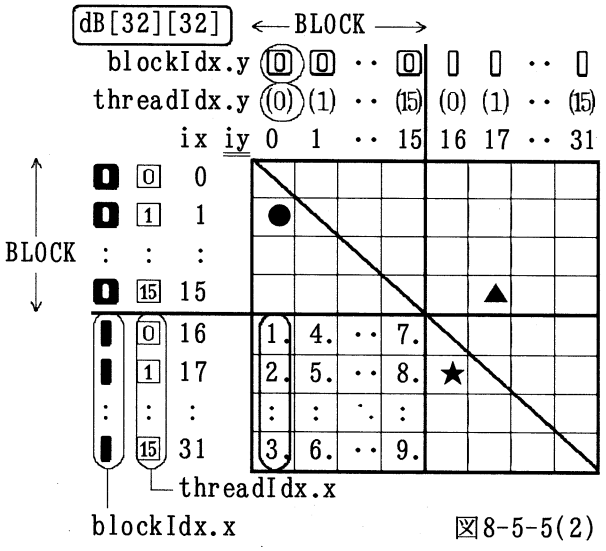


図8-5-5(2)

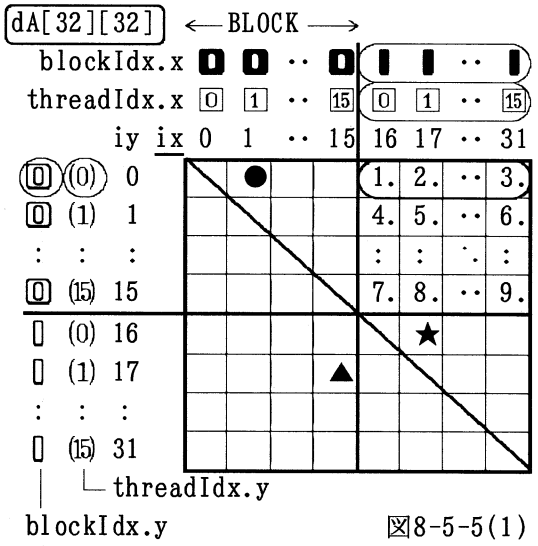


図8-5-5(1)

■ 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(▮,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(▮,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▮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▮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)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(■,▮)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,▯)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

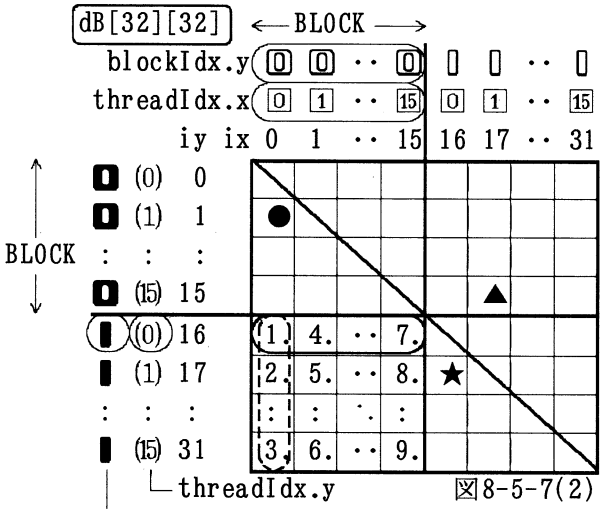


図8-5-7(2)

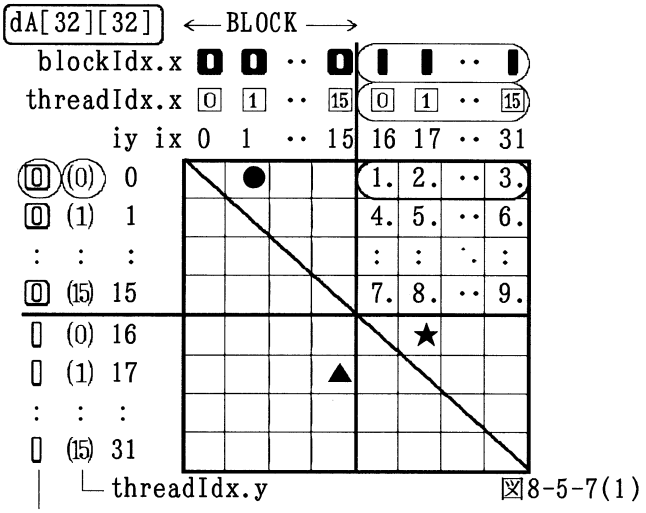


図8-5-7(1)

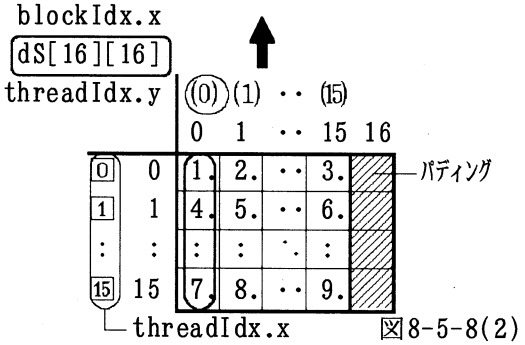


図8-5-8(2)

＝

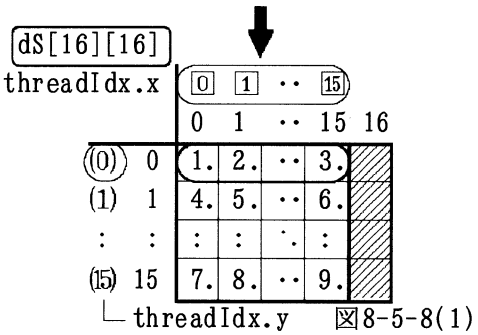


図8-5-8(1)

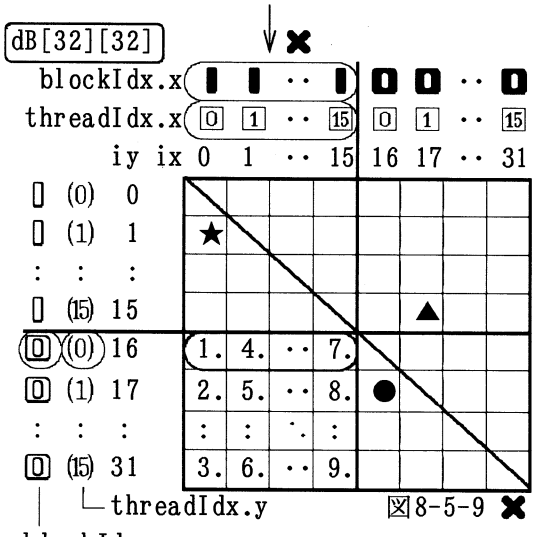


図8-5-9

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-)/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-)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
- Part10 CUDPP, a powerful data-parallel CUDA library
- Part11 Revisiting CUDA memory spaces
- Part12 CUDA 2.2 Changes the Data Movement Paradigm
- Part13 Using texture memory in CUDA
- Part14 Debugging CUDA and using CUDA-GDB
- Part15 Using Pixel Buffer Objects with CUDA and OpenGL
- Part16 CUDA 3.0 provices expanded capabilities
- Part17 CUDA 3.0 provices expanded capabilities and makes development easier
- Part18 Using Vertex Buffer Objects with CUDA and OpenGL
- Part19 Parallel Nsight Part 1: Configuring and Debugging Applications
- Part20 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