# これからの並列計算のためのGPGPU連載講座(II)
# GPGPUプログラミング環境CUDA 入門編

大島聡史
東京大学情報基盤センター

## 1　本編の構成

連載第二回である今回は、GPGPUプログラミング環境CUDA(CUDA Unified Device Architecture)について紹介する。

CUDAはNVIDIA社製GPU向けのプログラミング環境であり[*1]、C/C++言語を元に独自の拡張を行った専用の言語および対応するコンパイラ(nvcc)と実行時ランタイムライブラリ、そしていくつかの数値計算ライブラリから構成されている。CUDAは現在NVIDIA社製GPUに対して最も低いレイヤーでアクセスすることが可能なプログラミング環境であり、適切に利用することでGPUの持つ性能を引き出すことができる。逆に使い方を誤ると低い性能しか得ることができない。

今回述べる内容は以下の通りである：

- CUDAの導入方法
- CUDAプログラミング入門編1
- CUDAのアーキテクチャ概要
- CUDAプログラミング入門編2

CUDAはNVIDIA社製GPU向けのプログラミング環境ではあるものの、一般的なPC環境—Windows・Linux・MaxOSX—さえあればCUDAに対応したGPUが搭載されていなくてもプログラムを作成し動作確認をすることが可能である。是非とも実際にプログラムを作成して動かしてみて欲しい。(もちろん、実機でないと評価できないことや実機でないと生じないトラブルなどがあるため、なるべく実機で試して欲しい。)

CUDAプログラミングの助けになる資料やサイトについては、NVIDIA社のwebサイトCUDA Zone(http://www.nvidia.com/object/cuda_home_new.html)に多くの資料が公開されているうえに、「“CUDA” “プログラミング”」などのキーワードでweb検索すれば日本語で多数の資料を入手可能である。また、後述するCUDA Toolkitをインストールすればいくつかのドキュメントが利用可能になる。特にプログラミングガイドは、チュートリアルをはじめとしてアーキテクチャの解説や主なAPIとその使い方が掲載されており重宝する。(日本語翻訳版も公開されているものの、英語版と比べて公開が遅いことが多いため、最新情報を参照したい場合には注意が必要である。) プログラミングガイド以外に有用な資料やwebサイトとしては以下のものが挙げられる：

NVIDIA Forums (NVIDIA**フォーラム**)　NVIDIA社の公開しているフォーラム。現状では日本語フォーラムは初歩的なプログラミングの質問が多く、英語フォーラムはより深い話題が多

[*1] 名前の示すとおり、プログラミング環境というよりはアーキテクチャそのものであるとも言えるが、ここではこだわらないことにする。

くポストされている傾向がある。(http://forums.nvidia.com/, http://forum.nvidia.co.jp/)

**はじめてのCUDAプログラミング**　日本語で書かれたCUDAプログラミングの解説本。書籍の形式をとっているためCUDAの更新に追従できないという弱点はあるものの、日本語で丁寧に解説された本であり、タイトル通りCUDAをはじめて使う人でもそうでない人でも一読の価値がある。(http://www.kohgakusha.co.jp/books/detail/978-4-7775-1477-9)

NVIDIA CUDA Information Site　フィックスターズ社の有志が運営するwikiサイト。(http://gpu.fixstars.com/)

今回の記事においては特に断りがない限りCUDAに対応したNVIDIA社製GPUのことを単にGPUと呼ぶことにする。ただし、GPUそのものにもいくつかのバージョンが存在している。本記事の一部の記述は最新バージョン(Compute Capability 1.3)のGPUにのみ該当する場合があることを断っておく。また、今回述べる内容はCUDA Toolkit version 2.3を対象としている。CUDAは比較的頻繁にバージョンアップが行われており、バージョンアップのたびにAPIの変更などが起きるため、注意していただきたい。

なお、本連載中で扱うプログラム類は筆者のwebサイト(http://www.cspp.cc.u-tokyo.ac.jp/ohshima/) にて公開予定である。

## 2　CUDAの導入方法

まずはCUDAの導入方法を簡単に説明する。CUDAを利用するには、

1. CUDA対応ドライバ(CUDA対応GPUを利用する場合のみ)
2. CUDA Toolkit
3. CUDA SDK

が必要である。いずれもNVIDIA社のCUDAダウンロードサイト (http://developer.nvidia.com/object/cuda_download.html) にて無償で公開されている。画像処理ライブラリやプロファイラ等も同様に入手することができる。現在、Windows向け、Linux向け、MaxOSX向けのツールが公開されており、具体的なダウンロード・インストール手順はOSごとに異なっているものの、いずれも実行環境に対応した実行形式ファイルやスクリプトを実行させるのみの容易な手順で導入することができる。なお、実際に開発を行うためにはそれぞれ開発環境(VisualStudioやgccなど)が必要となる。さらに、LinuxにCUDAドライバを導入するためにはカーネルのソースコードなども必要となる。各ディストリビューションのパッケージシステムなどを用いることで導入可能なものばかりなので、必要に応じて各自導入していただきたい。

ドライバや開発環境のインストールが完了した後は、サンプルプログラムを実行してみると良い。サンプルプログラムはSDKに付属しており、たとえばLinuxであればデフォルトで$HOME/NVIDIA_GPU_Computing_SDK/C/bin/linux/release (ただし$HOME/NVIDIA_GPU_Computing_SDK/Cにてmakeコマンドを実行する必要がある) 以下に配置されるはずである。サンプルプログラムはソースコードが公開されており、解説ドキュメントが付属しているものもあるため、大いに参考になるだろう。なおサンプルの中には、CUDAに対応した実機のGPUが搭載されていないと実行できないもしくは実行できても著しく性能が

低くなるものや、GUIが利用できないと実行できないものもあるので注意していただきたい。

次章以降ではCUDAを用いたプログラムの記述方法や実行結果についていくつか紹介していくが、実行環境としてはスパコン利用者にも馴染みが深いと思われるLinuxを用いることにする。基本的に、実行環境(OS)によって大きく異なる点は無いが、異なる環境で実行する場合には適宜読み替えていただきたい。

メーカー製PCに対するGPUドライバの導入について

一部メーカー製のPCや多くのノートPCでは、NVIDIA社の提供する最新のGPUドライバを導入できない(そのため、最新版のCUDAを利用できない、もしくはCUDAそのものを利用できない) ことがあるので注意が必要である。これはメーカーが独自にカスタマイズしたドライバを必要とすることがあるためであり、メーカーによってはサポートwebサイトにてドライバ更新用のプログラムなどを公開していることもあるので、チェックしてみると良いだろう。

また、メーカーのサポートが外れるなどのリスクはあるものの、一部の機種については何らかの手段でドライバを更新できる可能性がある。例えば、PCの機種名とCUDA、もしくはPCの機種名とビデオドライバなどのキーワードを用いてweb検索を行うと、ドライバを更新する方法が公開されていることがある。参考にすると良いだろう。(実際に導入する場合には自己責任となるので注意されたい。)

## 3 CUDAプログラミング入門編1

続いて、CUDAを用いた初歩的なプログラムの例を示し、CUDAプログラムの基本的な動作の流れおよびプログラム作成方法について解説する。CUDAのアーキテクチャ等は次章で解説することにして、まずはどのようなプログラムを書くとどのように動くのか、雰囲気をつかんでいただきたい。

なお、CUDAには簡潔な記述でGPUを利用することができるRuntimeAPIと、より低レイヤーでGPUを細かく制御可能なDriverAPIがある。今回はプログラムの作成が容易なRuntimeAPIを対象として解説を行う。

### 3.1 CUDA プログラムの例と実行の流れ

まずは、図1および図2を見ていただきたい。図1は、C言語で記述した単純な配列加算プログラムのソースコードおよび同じ内容をCUDAを用いて記述したソースコードである。また図2にはソースコードをコンパイル・リンクし実行する手順および実行結果を示している。なお、CUDAプログラムの拡張子にはcuを用いることになっている。

図1からはCUDAプログラムは確かにCPUプログラムと比べて様々な記述が必要ではあるものの、特別に難解で手間のかかるプログラムではないことがおわかりいただけるだろうか。(ちなみに余談ではあるが、CUDAが登場する以前のプログラマブルシェーダを用いたプログラムでは本プログラムの3倍程度の記述と画像処理プログラミングに関する知識が要求されていた。そのうえ、直感的な記述でもなかった。) CUDAプログラムには__global__という関数の接頭辞や<<<と>>>を用いた記述、cudaで始まる関数群のように、既存のCPU向けのC言語プログ

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 256

int main(int argc, char** argv){
  int i;
  printf("CPU:¥n");
  srand(0);

  float *h_InA, *h_InB;
  h_InA = (float*)malloc(sizeof(float)*SIZE);
  h_InB = (float*)malloc(sizeof(float)*SIZE);
  for(i=0; i<SIZE; i++)    h_InA[i] = (float)(rand()%10)/10.0f;
  for(i=0; i<SIZE; i++)    h_InB[i] = (float)(rand()%10)/10.0f;
  printf("InA: "); for(i=0; i<SIZE; i++)printf(" %.2f", h_InA[i]); printf("¥n");
  printf("InB: "); for(i=0; i<SIZE; i++)printf(" %.2f", h_InB[i]); printf("¥n");

  float *h_Out;
  h_Out = (float*)malloc(sizeof(float)*SIZE);

  for(i=0; i<SIZE; i++)
    h_Out[i] = h_InA[i] + h_InB[i];

  printf("Out: "); for(i=0; i<SIZE; i++)printf(" %.2f", h_Out[i]); printf("¥n");

  free(h_InA);  free(h_InB);  free(h_Out);
  return 0;
}
```

```c
#include <stdlib.h>
#include <stdio.h>

#define SIZE 256

__global__ void arrayadd(float *fOut, float *flnA, float *flnB){
  int id = threadIdx.x + blockIdx.x*blockDim.x;
  fOut[id] = flnA[id] + flnB[id];
}

int main(int argc, char** argv){
  int i;
  printf("GPU:¥n");
  srand(0);

  cudaSetDevice(0);

  float *h_InA, *h_InB, *h_Out;
  h_InA = (float*)malloc(sizeof(float)*SIZE);
  h_InB = (float*)malloc(sizeof(float)*SIZE);
  h_Out = (float*)malloc(sizeof(float)*SIZE);
  for(i=0; i<SIZE; i++)    h_InA[i] = (float)(rand()%10)/10.0f;
  for(i=0; i<SIZE; i++)    h_InB[i] = (float)(rand()%10)/10.0f;
  printf("InA: "); for(i=0; i<SIZE; i++)printf(" %.2f", h_InA[i]); printf("¥n");
  printf("InB: "); for(i=0; i<SIZE; i++)printf(" %.2f", h_InB[i]); printf("¥n");

  float *d_InA, *d_InB, d_Out;
  cudaMalloc((void**)&d_InA, sizeof(float)*SIZE);
  cudaMalloc((void**)&d_InB, sizeof(float)*SIZE);
  cudaMalloc((void**)&d_Out, sizeof(float)*SIZE);
  cudaMemcpy(d_InA, h_InA, sizeof(float)*SIZE, cudaMemcpyHostToDevice);
  cudaMemcpy(d_InB, h_InB, sizeof(float)*SIZE, cudaMemcpyHostToDevice);

  arrayadd<<< 16, 16 >>>(d_Out, d_InA, d_InB);

  cudaMemcpy(h_Out, d_Out, sizeof(float)*SIZE, cudaMemcpyDeviceToHost);
  printf("Out: "); for(i=0; i<SIZE; i++)printf(" %.2f", h_Out[i]); printf("¥n");

  free(h_InA);  free(h_InB);  free(h_Out);
  cudaFree(d_InA); cudaFree(d_InB); cudaFree(d_Out);
  return 0;
}
```

GPU上で実行される「カーネル関数」

GPU上のメモリ確保および
CPUからGPUへのデータ転送

カーネル関数の呼び出し

GPUからCPUへのデータ転送

図 1　C 言語 (上) と CUDA(下) の配列加算プログラム例

C 言語でのコンパイルと実行

```
> ls
cpu.c
> gcc -O3 cpu.c
> ls
a.out cpu.c
> ./a.out
CPU:
InA:  0.30 0.60 0.70 0.50 0.30 0.50 0.60 0.20 0.90 0.10 0.20 0.70 (以下省略)
InB:  0.00 0.60 0.40 0.60 0.20 0.50 0.80 0.60 0.20 0.80 0.40 0.70 (以下省略)
Out:  0.30 1.20 1.10 1.10 0.50 1.00 1.40 0.80 1.10 0.90 0.60 1.40 (以下省略)
>
```

CUDA でのコンパイルと実行

```
> ls
gpu.cu
> nvcc -O3 gpu.cu -I${HOME}/NVIDIA_GPU_Computing_SDK/C/common/inc
> ls
a.out gpu.cu
> ./a.out
GPU:
InA:  0.30 0.60 0.70 0.50 0.30 0.50 0.60 0.20 0.90 0.10 0.20 0.70 (以下省略)
InB:  0.00 0.60 0.40 0.60 0.20 0.50 0.80 0.60 0.20 0.80 0.40 0.70 (以下省略)
Out:  0.30 1.20 1.10 1.10 0.50 1.00 1.40 0.80 1.10 0.90 0.60 1.40 (以下省略)
>
```

図 2　配列加算プログラムの実行手順および実行結果

ラムにはない特徴的な記述が含まれていることがわかる。また図2からは、CUDAプログラムもCPUプログラムと同様にソースコードを専用のコンパイラ(nvcc)でコンパイル・リンクすることで実行ファイルが生成されること、生成されたファイルを実行すればGPU上でプログラムが動作すること(a.out以外に特殊なファイルを生成したり必要としたりはしないこと)がわかる。

CUDAプログラムにおいては、図1中に記したように、プログラムに記述された処理の全てがGPU上で実行されるわけではない。むしろ明示的に指示された部分以外はCPU上で実行される。この「明示的な指示」を行うのに用いるのが__global__などの「関数に対する指示子」と、<<<と>>>を用いた「GPU呼び出し記述」である。CUDAコンパイラnvccはこれらの指示子・呼び出し記述を目印としてソースコードを解析・分離し、CPUによって実行される部分はCPU向け、GPUによって実行される部分はGPU向けのコンパイルを行い、それぞれを結合した最終的な実行可能プログラムを出力する。

「関数に対する指示子」と記したが、CUDAにおいてGPUに実行させる処理の単位は関数である。これを「GPUカーネル」や「GPUカーネル関数」、もしくは単に「カーネル」などと呼ぶ。計算量が大きいため計算時間が長くなおかつ並列度が高い部分を関数として抽出し、抽出した関数をGPUに実行させることでプログラム全体の実行時間を削減する、というのがCUDAの基本的な戦略となる。(これはCUDAのみならず、ClearSpeed Advanced AcceleratorやCell BEにおけるSPEなど、「アクセラレータ」と呼ばれるハードウェアを用いる場合の基本的な戦略であると言える。)

CUDAにおける「関数に対する指示子」には以下の3つがある。複数の指示子を指定するこ

とも可能である。使い分ける上で特に難しい点はなく、間違えて記述した場合にはnvccに明確に指摘されるだろう。

__global__　CPUに呼び出されてGPU上で実行される関数
__device__　GPUに呼び出されてGPU上で実行される関数
__host__　CPUに呼び出されてCPU上で実行される関数(global/device関数と同名の関数をCPU上でも実行したい場合に使用する)

また、CPUとGPUは独立したメモリを持つ。CUDAにおいてはCPUからGPU上のメモリに対する読み書きには制限があり、GPUからCPU上のメモリを操作することはできない。CPUからGPUに対するメモリの読み書き、すなわちCPU-GPU間のデータ転送については、カーネル関数の前後でAPI（cudaMemcpyなどのAPI関数）を用いて行う必要がある。これは、MPIのようにデータの送受信両方を明示的に記述するプログラミングとも、OpenMPのようにデータの送受信が不要なものとも異なっている。他の環境で並列化プログラミングを行ってきた経験がある場合には特に注意されたい。

次にcudaから始まるAPI関数について簡単に説明する。今回利用しているAPIは以下の通りである。

cudaSetDevice　使用するGPUのIDを指定する関数
cudaMalloc　GPU上のメモリを確保する関数(以下の関数では本関数によって得られたポインタを用いる)
cudaMemcpy　CPU-GPU間のデータ転送を行う関数(CPUからGPUとGPUからCPUの両方で同じ関数を使用し転送方向は第4引数で制御する)
cudaFree　確保したGPU上のメモリを解放する関数

これらの関数以外にも、CUDAにより様々な関数が提供されている。詳細についてはサンプルプログラムやプログラミングガイドを参照していただきたい。

nvccのコンパイルオプションやその他の機能について

本節では特に断りなくnvccをgccと同様のオプションを用いて実行した。しかし、nvccはgccと同じオプションに対応しているわけではない。例えばgccを使用する際に良く用いられる-Wallオプション(全ての警告を表示するオプション)には対応していない。
nvccには様々なオプションと機能がある。たとえば-cubinや-ptxといったオプションを利用すると、GPUによって実行される部分のみをコンパイルしバイナリ形式や中間表現PTX形式として得ることができる。(これらの機能はRuntimeAPIを利用する場合には使わないため、今回は解説しない。)
頻繁に利用するオプションとしては、対象とするGPUの世代を指定する-archオプションがある。CUDAは世代(Compute Capability)によって対応する機能などに違いがあり、新しいGPU向けの機能を使用するプログラムを古い世代のGPUで実行することができない。各GPUにどのarchオプションを指定すれば良いかについては、プログラミングガイドを参照していただきたい。
nvccが対応しているオプションの一覧は-hオプションによって得ることができる。興味があれば確認してみて欲しい。

## 3.2 CUDA における並列処理

前節ではCPUからGPUを操作する方法やCPU-GPU間のデータ転送について確認した。つづいて本節ではGPUカーネルに注目してみることにする。

前節で用いた図1の主要な計算部分を比較すると、CPUプログラムである前者にはループ構造が含まれているのに対して、GPUプログラムである後者にはループ構造が含まれていないにも関わらず配列の各要素に対する計算が行えていた。CUDAプログラムにおいては、GPU上の多数のプロセッサそれぞれにおいて同一のGPUカーネルが動作する。実行される各インスタンスは個別のIDを持つため、IDを元に計算対象のデータを特定し変更することができる(図3)。動作のイメージとしては、pthreadのようにスレッド毎に個別のプログラムを実行する並列化よりは、rankを用いて処理を分けるMPIにやや近いと言えるだろう。

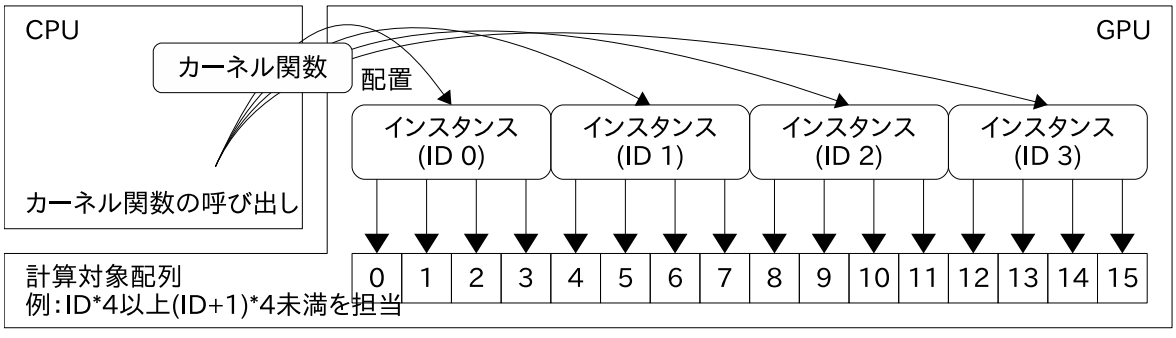


図 3　CUDA における ID を用いた並列処理

CUDAでは各インスタンスにthreadIdとBlockIdという二階層のIDが割り当てられる。また、IDの範囲、すなわちCPU向けの並列化プログラミングで言うところのスレッド数やプロセス数については、GPUカーネルを呼び出す際に指定する。図1における<<<と>>>の間に記述

されている値がこの指定に該当する。図1の例では、16*16=256のインスタンスが生成され、GPU上のプロセッサに割り当てられて実行されることになる。そのため、長さ256の配列に対する計算が明示的なループ記述無しに実行されているわけである。

CUDAにおけるインスタンスの割り当ては、既存のCPU向け並列化プログラミングと比べるとやや複雑である。これについてはGPUのハードウェアアーキテクチャについて知らなくては理解が困難であるため、次章で説明する。

## 4　CUDAのアーキテクチャ

本章では、CUDAのハードウェアモデル、実行モデル、メモリモデルについて解説する。前章で紹介したCUDAプログラムの基本的な構造やCUDAプログラミングの手順と照らし合わせて理解して欲しい。

### 4.1　物理的なハードウェア構成

はじめに、CUDAが対象としているNVIDIA社製GPUの中でも2010年2月現在の最新アーキテクチャであるGT200アーキテクチャ(以下GT200)におけるハードウェア構成を示す。

図4はGPUにおける演算器とメモリの構成である。基本的な構成としては、ScalarProcessor(SP)と呼ばれるシンプルなプロセッサが8つ集まってMultiProcessor(MP)を構成しており、MPがGPUのグレードによって複数個搭載されている。さらに、MP毎に独立したメモリ(同一MP内のSPでのみ共有されるメモリ)とMP間で共有されるメモリが搭載されている。各メモリの特徴や使い方については4.3節で紹介する。このように、演算器とメモリが階層性を持っているのがGPUの特徴である。(ちなみに、NVIDIA社製の旧世代GPUやAMD社製GPUも演算器とメモリに階層性を持つアーキテクチャを採用している。)

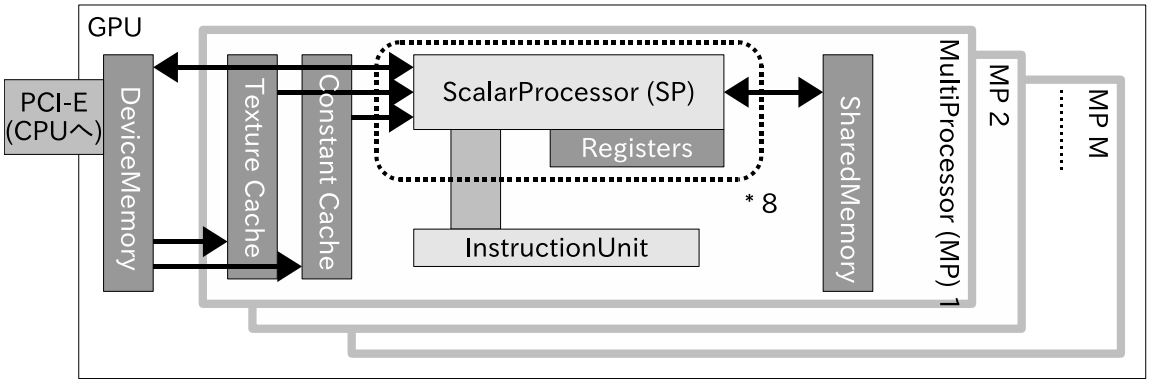


図 4　NVIDIA 社製 GPU における演算器とメモリの構成

GPUには1GPUあたり最大30のMPが搭載されているため、SP数は最大で240に達する。(TeslaS1070は1筐体に4GPUが搭載されているハードウェアであり、1GPUあたりのSP数はやはり240である。) さらに後述するが、GPUはCPUと異なり物理コア数よりも高い並列度での並列処理に適した設計となっており、1GPUあたりでは1000を超える並列性にまで性能がスケールする。ただし、SPはマルチコアCPUにおけるCPUコアのように独立して演算を行えるわけではない。既存のCPUと比較するのであれば、SPはSIMDコアであり、CPUコアに対応す

るのはMPということになる。コア数が30あるCPUはもちろん並列度が高いCPUであると言えるものの、単純にコア数が240あるCPUと考えるのは現実にそぐわないため注意が必要である。

さて、SPはSIMDコアであると述べたが、実際に同一MP上のSPは同時に異なる演算を行うことができない。複数のSPがそれぞれ異なるデータに対して同じ演算を行うデータ並列処理がCUDAにおける基本的な並列処理である。また、SPはインオーダーであり分岐予測器も持たないシンプルな計算コアである。そのため、SPは同一クロックのCPUコアと比べると演算性能が低く、GPUは多数のSPを活用可能な並列度の高い問題でなくては高い性能を得ることができない。

一方でMP単位での並列処理を考えた場合は、異なるMP間で同時に異なる演算を行うことができるため、データ並列処理にこだわる必要はない。ただし、MP単位での並列処理では同期をとれる範囲に気をつけなくてはならない。同一MP内のSP同士では容易に同期をとることができる(カーネル関数内で__syncthreads関数を呼び出すだけで同期をとることができる)一方で、異なるMP間のSP同士で同期をとるためには一度GPUカーネルを終了してCPUに制御を戻さなくてはならない。GPU上の共有メモリに対して排他的な演算を行うことができるatomic関数を用いて同期を実現することも不可能ではないが、性能が低下する可能性が大きい。

## 4.2 物理的な構成と実行モデル

GPUに搭載されている演算器はSPとMPの階層性を持つことを述べた。SPは8個ごとにMPを構成しており、MPがGPUのグレードによって複数個搭載されているのがGT200の物理的な構成である。CPUがCPUコア数以上のインスタンス（スレッドやプロセス）を割り当てられると時分割実行を行うのと同様に、GPUもSP数やMP数以上のインスタンスを割り当てられると時分割実行を行うことになる。CPUがCPUコア数よりも多くのインスタンスを実行しようとした場合、一般的にはCPUコア数以下のインスタンス数の場合と比べて性能が低下する。一方でGPUは、コンテキストスイッチのコストが非常に低く、GPUにハードウェア実装されているスケジューラはメモリを読み書きする際にSPが待つ必要がある場合に積極的にコンテキストスイッチを行う。そのため、物理的なSP数やMP数よりも多くのインスタンスを生成した方がメモリアクセスのレイテンシが隠蔽されて高い性能を得ることができる。

CUDAにおいてはSP単位のジョブをスレッド(Thread)、MP単位のジョブをブロック(Block)もしくはスレッドブロック(Thread Block)、さらにカーネル実行単位をグリッド(Grid)と呼ぶ(図5)。図中ではいずれも一次元で表現しているが、実際にはブロックは二次元、スレッドは三次元の空間を割り当てることができる。ただし、同一のグリッド内においてブロックごとにスレッド数を変更することはできない。同一ブロック内のスレッドは全て同じMPに割り当てられ、また各ブロックはコンテキストスイッチが行われても異なるMPへと割り当てられることはない。どのブロックがどのMPに割り当てられるかをプログラマが制御することはできない。一般的にスレッド数は128程度以上割り当てると良いとされている。ただし、各スレッドがどれだけの資源(レジスタや共有メモリ)を使うかなどにより最適な値は異なるため、最大の性能を得るためにはアプリケーション毎に数を調整して性能を測定してみる必要がある。

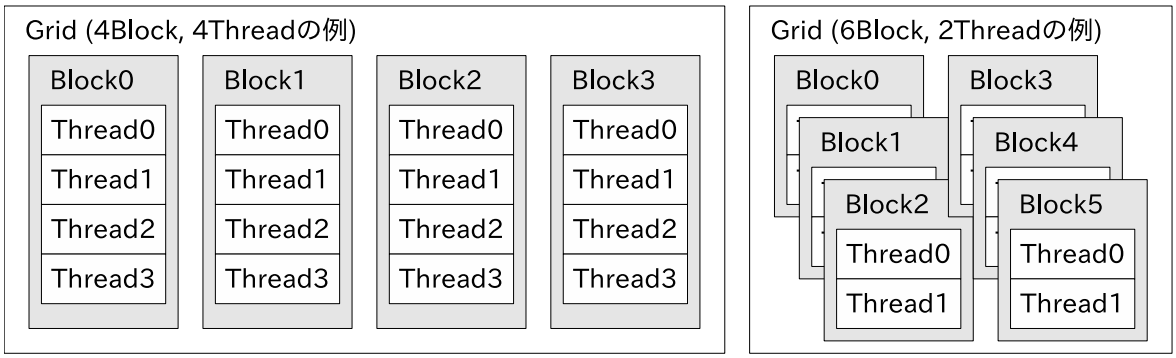


図5　SP・MP とスレッド・ブロック

## 4.3　メモリの分類と使い分け

最後に、GPU上に搭載されたメモリについて解説する。

図6にGPUにおけるメモリの分類を示す。GPUには複数種類のメモリが搭載されており、それぞれ異なる特性を備えている。以下に各メモリの概要を示す。

Registers　各MPごとに独立して搭載されているレジスタ。スレッド毎に独立したレジスタとして利用可能。GT200ではMP毎に16384本。高速で低レイテンシ。デバイス関数内で宣言された通常の変数は基本的にレジスタへ割り当てられる。CPUから直接値を読み書きすることはできない。

SharedMemory　各MPごとに独立して搭載されているメモリ。同一MP内の各スレッドからは共有メモリとして利用可能。GT200では各MP毎に16KB。高速で低レイテンシ。__shared__指示子を用いて宣言された変数がSharedMemoryとして扱われる。CPUから直接値を読み書きすることはできない。Gridをまたいでデータを保持することはできない。
メモリがバンクに分かれており、同時に複数のスレッドが同一のバンクにアクセスすると性能が劣化する（バンクコンフリクト）ため、最適化の際には注意が必要。(次回の連載にて説明する。)

GlobalMemory　GPU全体で共有されるメモリ。全ブロック・全スレッドから共有メモリとして利用可能。いわゆるビデオメモリ(VRAM)容量分利用可能だが、TextureMemoryと共用。連続アクセスに対しては高速だが高レイテンシ、ランダムアクセスに対しては低速で高レイテンシ。__device__指示子を用いて宣言された変数の他、__global__関数の引数がGlobalMemoryとして扱われる。CPUからAPIを介して値を読み書きすることができる。Gridをまたいでデータを保持することができる。

ConstantMemory　GPU全体で共有されるメモリ。GPU全体で64KB。全ブロック・全スレッドから共有メモリとして利用可能だが、GPUからは読むことしかできない。MP毎にConstantMemoryに対するキャッシュが8KB搭載されている。__constant__指示子を用いて宣言された変数がConstantMemoryとして扱われる。CPUからAPIを介して値を設定することができる。Gridをまたいでデータを保持することができる。

TextureMemory　GPU全体で共有されるメモリ。いわゆるビデオメモリ(VRAM)容量分利用可

能だが、GlobaleMemoryと共用。全ブロック・全スレッドから共有メモリとして利用可能だが、GPUからは読むことしかできない。MP毎にTextureMemoryに対するキャッシュが6KBから8KB搭載されている。専用の構造体を用いて記述する必要がある。CPUからAPIを介して値を設定することができる。Gridをまたいでデータを保持することができる。

LocalMemory　実態としてはGlobalMemoryであり、スレッド毎のレジスタ数が多すぎる場合に自動的に割り当てられる。

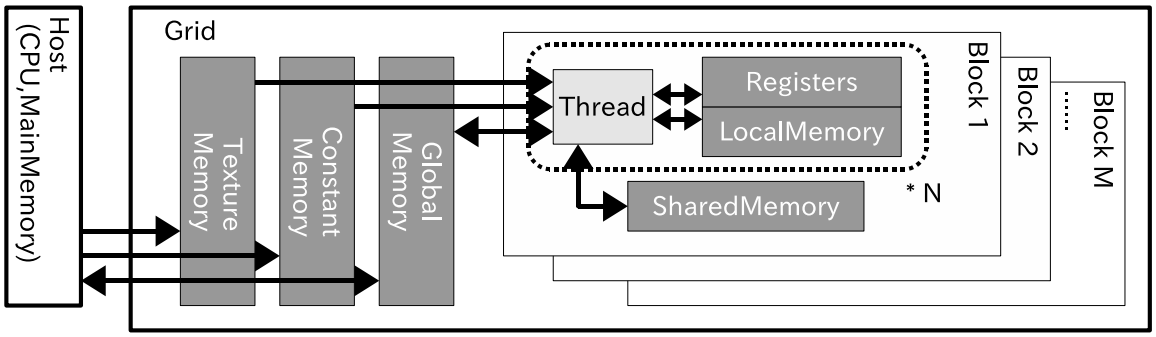


図 6　メモリモデル

各メモリを適切に利用することはGPUを用いて高い性能を得る上で非常に重要である。次章では各メモリを用いたプログラミングの簡単な例を、次回以降の連載では高性能を得るための最適化プログラミングの基本的な考え方と例、および実例を用いた最適化の例を紹介する。

## 5　CUDAプログラミング入門編2

これまでに説明してきたように、CUDAは既存のCPU(C言語プログラミング)よりも複雑な独自の実行モデルやメモリモデルを備えている。そこで本章では、特に使用する機会が多いと思われるGlobalMemoryとSharedMemoryについて、いくつか小さなテストプログラムを作成して基本的な使い方を確認することにする。なお、今回は基本的な使い方の確認を中心とし、性能に関する内容は次回の内容とする。

### 5.1　GlobalMemory を用いたプログラミング

まずはGlobalMemoryを用いたプログラミングについて見てみることにする。

3.1節で示した図1のサンプルプログラムでは、__device__の記述こそ無かったものの既にGlobalMemoryを利用していた。それは、__global__関数の引数である。4.3節で述べたように、__global__関数の引数はGlobalMemoryとして扱われる。これは配列でも配列以外の変数でも同様である。

GlobalMemoryの特徴としては、GPU上の全SPで共有するメモリであることと、グリッドをまたいで値が保持されることが挙げられる。これらの特徴を確認できるプログラムとして図7を作成した。このプログラムは、1スレッドを持つブロックを2つ作成し、1つ目のグリッド(__global__関数)では各スレッドがGlobalMemoryへの書き込みを行い、2つ目のグリッド(__global__関数)で

はもう1つのスレッド(異なるブロックのスレッド)が書き込んだ内容を読み出すというプログラムである。このプログラムからGlobalMemoryがブロック(MP [*2])を超えて共有されていることが確認できるとともに、2つのグリッド(__global__関数)の間で明示的なメモリコピーを行っていないことからGlobalMemory値がグリッドをまたいで保持されていることも確認できる。

それでは、GlobalMemoryの同一のメモリに対して複数のスレッドが同時に書き込みを行うとどうなるだろうか。これまでに並列処理プログラムを書いた経験があれば想像できるように、GPUにおいても同一のメモリに対する読み書きには気をつけなくてはならない。図8は、多数のスレッドから同一のGlobalMemory上の変数に対して加算を行うという、並列処理においてありがちな「正しく動作しないプログラム」である。本プログラムの実行結果は実行タイミングに依存し、一意に定まらない。これを「正しく動作させる」、すなわちスレッド数分だけ確実に加算させる方法としては、「atomic関数」を利用するのが容易である。図8における

```
data[0] += 1;
```

を

```
atomicAdd(&data[0], 1);
```

に書き換えれば意図した結果(この場合では16384という数値)が得られることになる。ただし、多数のスレッドでatomic関数を使用すると性能低下を招くため多用するべきではない。これに関連するサンプルとしては、各スレッドの持つデータを集めるリダクション処理のサンプル(reduction)が参考になるだろう。

以上がGlobalMemoryの基本的な使用方法と注意点である。この他、性能について特に重要なこととしてメモリアクセスをできる限り連続アクセス(コアレスなメモリアクセス) になるようにすることなどが挙げられるが、これらについては次回解説する。

## 5.2 SharedMemory を用いたプログラミング

SharedMemoryは低レイテンシで高速な共有メモリではあるものの、局所的な共有メモリであり容量が小さいため、使用がやや難しいメモリであると言える。今回は入門編ということで、SharedMemoryを用いることでGlobalMemoryへのアクセス回数を減らして性能を向上させる、という基本的な使い方を紹介する。

テストプログラムとして、「各スレッドはブロックごとにGlobalMemoryの決められた範囲のデータを収集し、スレッドIDを掛け合わせる」という処理を考えることにする。この処理自体に意味はないが、「スレッド毎に決められた範囲のデータを収集する」「隣接スレッドは互いに近い範囲のデータを処理する」という点については既存のCPU向けプログラム(並列処理プログラムを含む)において様々な場面で用いられている処理である。(格子点データの時間発展などに応用できるはずである。)

テストプログラム(SharedMemoryを用いないカーネル関数とSharedMemoryを用いたカーネル関数) を図9に示す。fData配列に注目すると、kernel1ではGlobalMemory上にある必要なデー

[*2] ブロックと MP の割り当てを制御することはできないが、今回のようにブロック数が MP 数より少ない場合には各ブロックは別々の MP に割り当てられると考えて良い

プログラム (test1.cu)

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

__device__ float globalarray[2];

__global__ void kernel1(){
  if(blockIdx.x==0){
    globalarray[0] = 111.11f;
  }else{
    globalarray[1] = 222.22f;
  }
}
__global__ void kernel2(float *array){
  if(blockIdx.x==0){
    array[0] = globalarray[1];
  }else{
    array[1] = globalarray[0];
  }
}

int main(int argc, char** argv){
  int i;
  printf("GPU:\n");
  srand(0);

  cudaSetDevice(0);

  float h_Out[2];
  float *d_Out;
  cudaMalloc((void**)&d_Out, sizeof(float)*2);

  kernel1<<< 2, 1 >>>();
  kernel2<<< 2, 1 >>>(d_Out);

  cudaMemcpy(h_Out, d_Out, sizeof(float)*2, cudaMemcpyDeviceToHost);
  printf("Out: "); for(i=0; i<2; i++)printf(" %.2f", h_Out[i]); printf("\n");

  cudaFree(d_Out);
  return 0;
}
```

実行結果

```
>nvcc -O3 -o test1 test1.cu -I/home/ohshima/NVIDIA_GPU_Computing_SDK/C/common/inc
>./test1
GPU:
Out:  222.22 111.11
>
```

図 7　GlobalMemory を用いるテストプログラム 1(GlobalMemory の基本動作)

プログラム (test2.cu)

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

__global__ void kernel1(int *data){
  data[0] += 1;
}

int main(int argc, char** argv){
  printf("GPU:\n");
  srand(0);

  cudaSetDevice(0);

  int h_Out = 0;
  int *d_Out;
  cudaMalloc((void**)&d_Out, sizeof(int));
  cudaMemcpy(d_Out, &h_Out, sizeof(int), cudaMemcpyHostToDevice);

  kernel1<<< 128, 128 >>>(d_Out); // 128*128=16384parallel

  cudaMemcpy(&h_Out, d_Out, sizeof(int), cudaMemcpyDeviceToHost);
  printf("Out: "); printf("%d\n", h_Out);

  cudaFree(d_Out);
  return 0;
}
```

実行結果

```
>nvcc -O3 -o test2 test2.cu -I/home/ohshima/NVIDIA_GPU_Computing_SDK/C/common/inc
>./test2
GPU:
Out: 2
>
```

図 8　GlobalMemory を用いるテストプログラム 2(正しく加算されないプログラム)

タを各スレッドがそれぞれ取得している。そのため、データの取得と足し合わせに合計でスレッド数＊データサイズ分のGlobalMemoryアクセスが、各スレッド単位でもデータサイズ分の逐次的なGlobalMemoryアクセスが必要である。一方でkernel2では途中の計算にSharedMemoryを用いることで、合計のGlobalMemoryアクセスはデータサイズ分のみ、各スレッド単位では1アクセスのみに削減することができている。kernel2ではSharedMemoryへのアクセスを必要とする計算が追加されているが、SharedMemoryはGlobalMemoryアクセスよりもレイテンシが小さいため追加された分の時間よりも削減される時間の方が大きく、全体として性能向上を期待することができる。

残念ながら今回のプログラムは、全体のデータ量や並列度が小さすぎる・SharedMemory上でのデータ足し合わせも並列化するべきである・GlobalMemoryを高速に利用できる問題である、といった事情があり、実際にSharedMemoryを利用するメリットが十分に得られる問題ではない。SharedMemoryを活用するイメージとして理解していただきたい。

```
// 問題を単純にするため、ブロックあたりのスレッド数は 256 とする

// SharedMemory を用いない場合
__global__ void kernel1(float *fOut, float *fData){
  int i;
  float tmp = 0.0f;
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  // 各スレッドが GlobalMemory のデータを取得し足し合わせる
  for(i=0; i<256; i++){
    tmp += fData[i];
  }
  // ID を掛け合わせて返す
  tmp *= (float)threadIdx.x;
  fOut[id] = tmp;
}

// SharedMemory を用いる場合
__global__ void kernel2(float *fOut, float *fData){
  __shared__ float sData[256];
  int i;
  float tmp = 0.0f;
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  // 各スレッドが GlobalMemory の部分データを取得し SharedMemory に格納する
  sData[threadIdx.x] = fData[threadIdx.x];
  // スレッド間の同期
  __syncthreads();
  // 特定のスレッド上で SharedMemory 上のデータを足し合わせる
  if(threadIdx.x==0){
    for(i=1; i<256; i++){
      sData[0] += sData[i];
    }
  }
  // スレッド間の同期
  __syncthreads();
  // 各スレッドが ID を掛け合わせて返す
  tmp = sData[0] * (float)threadIdx.x;
  fOut[id] = tmp;
}
```

図 9　SharedMemory を用いるテストプログラム

以上、第二回の今回はCUDA対応GPUアーキテクチャの概要と、CUDAプログラムの概要および実行方法について紹介した。次号では、GPU上に搭載された階層的な演算器とメモリをより効果的に活用するための最適化プログラミングについて紹介する予定である。
(次号に続く)