これからの並列計算のためのGPGPU連載講座(VI)
様々なGPGPUプログラミング環境

大 島 聡 史
東京大学情報基盤センター

1　はじめに

本連載ではGPGPUプログラミング環境として主にCUDAを取り扱ってきた。CUDAは確かに現在主流のGPGPUプログラミング環境であり、今後も主要なGPGPUプログラミング環境として広く利用されることが期待される。一方でCUDA以外にもGPGPUプログラミング環境に関する研究や開発は行われている。そこで連載第六回(最終回)の今回は、CUDA以外のGPGPUプログラミング環境について解説する。

2　OpenCL

OpenCLはGPGPUを含む様々な並列計算のためのフレームワークであり、Apple社によって提案された。現在は標準化団体Khronos Groupの作業部会OpenCL Working Groupによって策定されている[1]。CUDAはGPUベンダーであるNVIDIA社自身のGPUでのみ利用可能である[*1]のに対して、OpenCLはNVIDIA社とAMD社のGPUをはじめ、マルチコアCPUやDSP(Digital Signal Processor)など、様々な並列処理ハードウェアをターゲットとしている。特にNVIDIA社と並ぶ大手GPUベンダーであるAMD社は、自社GPU向けのプログラミング環境としてOpenCLに力を入れている。

本節ではOpenCLのプラットフォームモデルや言語仕様について確認した上で、単純なソースコードを例示してOpenCLプログラムの解説を行う。なお、この記事は本連載の過去の記事を読んでいる、すなわちCUDAについてある程度の知識を持っている読者を対象とするため、CUDAとOpenCLの比較にやや重点を置いた解説を行う。

2.1　OpenCL のプラットフォームモデルとメモリモデル

OpenCLの対象とするプラットフォームモデルを図1に示す。OpenCLは階層的な並列性を持つハードウェアからなるプラットフォームモデルを有している。すなわち、Host(ホスト、汎用CPUとメインメモリを搭載した計算機)には(複数の)Compute Device(計算デバイス、具体的にはGPUカードなど)を搭載することが可能であり、各Compute Deviceは(複数の)Compute Unit(計算ユニット)によって構成されている。さらに、各Compute Unitは(複数の)Processing Element(計算エレメント)によって構成されている。

OpenCLのメモリモデルについては図2のような構成となっている。OpenCLではCUDAにおけるMPとBlockおよびSP(CUDA Core)とThreadのように、Processing Elementに対応するWork-Item(ワークアイテム)およびCompute Unitに対応するWorkgroup(ワークグループ)という割り当てがなされている。そのうえで、OpenCLのメモリは複数種類のメモリから構成され

[*1] 仕様が公開されているため、別の環境への移植もおそらく不可能ではない

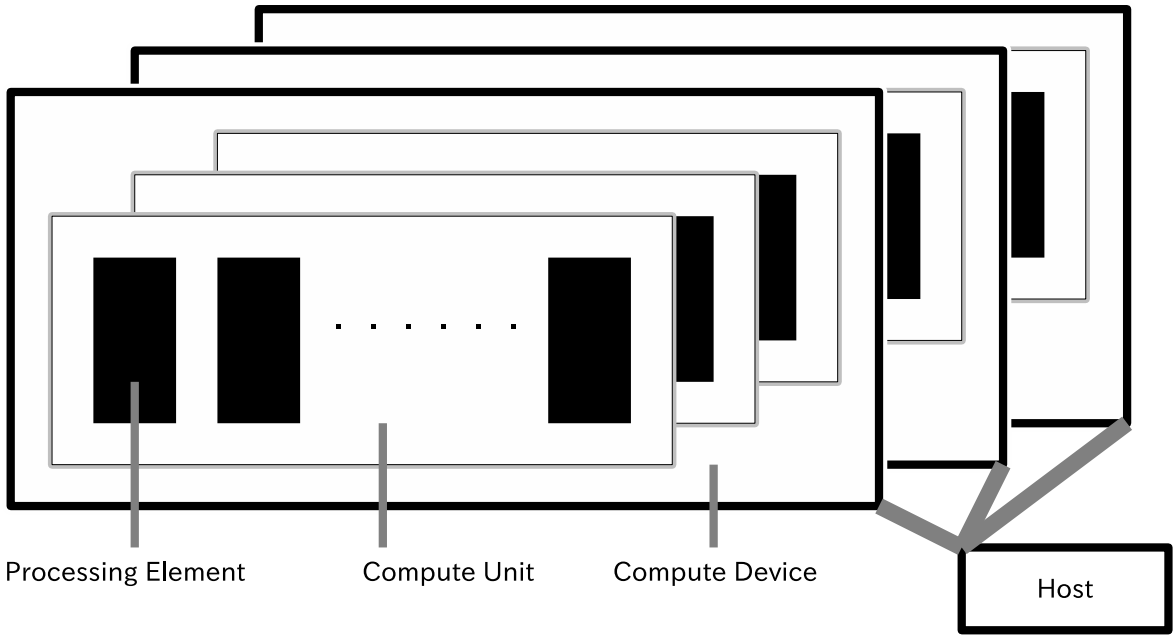

図 1　OpenCL のプラットフォームモデル

ており、Work-Item単位のPrivate Memory、Workgroup単位のLocal Memory、Compute Device単位のGlobal/Constant Memory、そしてHostのHost Memoryがある。

このようにOpenCLのモデルを見てみると、CUDAと似ていることに気がつくだろう。プラットフォームモデルをCUDAと見比べると、Compute Unit と MP、Processing Element とSPが同様の位置づけであることがわかる。またメモリモデルについても、Private Memory とRegister、Local Memory と Shared Memory、Global/Constant Memory と Global Memory そしてHost Memory と Host Memoryが同様の位置づけであることがわかる。そのため、既にCUDAの知識を得ているユーザにとってはOpenCLのモデルを理解することはそれほど難しくないだろう。

このようにOpenCLとCUDAとを比べてみると非常に似ており、OpenCLもGPUに特化したモデル(フレームワーク)であるように見えるかもしれない。OpenCLではこのモデルで—ただし構成上の一部のメモリ容量が0であることもありえるが— 様々な並列計算ハードウェアに対応できるとしている。また、必要条件を引き下げたOpenCL Embedded Profileも策定されており、こちらは組み込み機器における活用が期待されている。

2.2　OpenCL のプログラミングモデルと言語仕様

つづいて、OpenCLのプログラミングモデルと言語仕様について解説する。

OpenCLは、「通常のC/C++で記述されたOpenCL API関数を含むホストコード」から「OpenCL専用の記述(指示子や組み込み関数)を用いて記述された計算カーネル関数」を呼び出すというスタイルをとっている。計算カーネル関数呼び出し時に並列度の指定を行うことで、対象計算カーネルが複数の計算デバイス上の複数のCompute UnitおよびProcessing Element上で実行される。ホストと計算デバイス間のデータ転送については、別途専用の関数(API)を用いて記

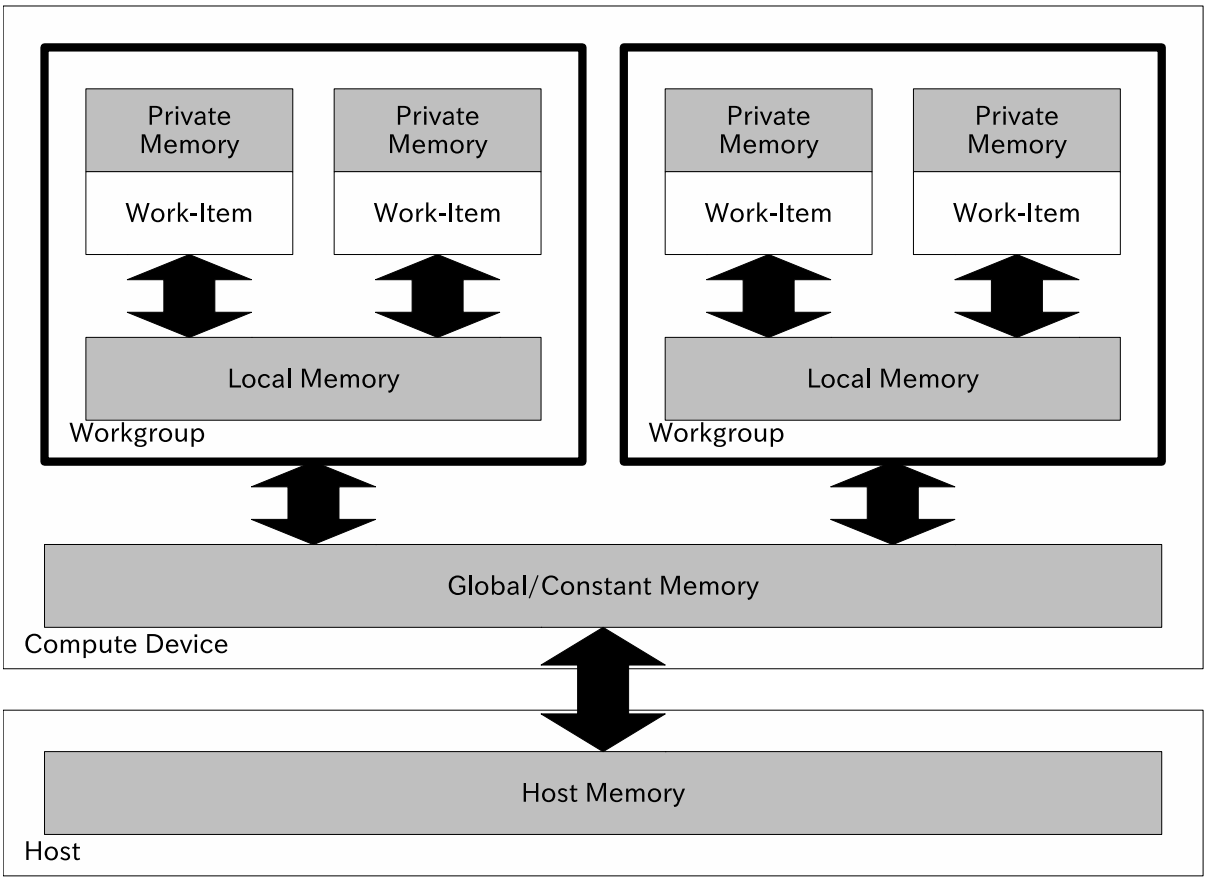

図 2　OpenCL のメモリモデル

述する必要がある。

OpenCLのプログラム記述はC/C++言語を拡張した記法で行う。C/C++言語と異なる点は、メモリや関数に専用の指示子をつける点である。例えば、GPU(計算デバイス)に実行させる計算カーネルは、*__kernel*という指示子を付加した関数という形で記述する。また、Global Memoryに配置する変数については、*__global*という指示子を付加する。これらの記述例は次節のソースコード例を参照していただきたい。

以上のように、具体的な指示子自体は異なるものの、OpenCLプログラムの動作イメージと記述方法はCUDAと非常に似通っている。一方でOpenCLの特徴としては実行時に計算カーネルを生成する(コンパイルする)ことが挙げられる。CUDAではプログラムコンパイル時に計算カーネルのコンパイルを行い、プログラム実行時にはコンパイル済みの計算カーネルをロードし実行する。これに対して、OpenCLではプログラム実行時に計算カーネルのコンパイルを行う機構が用いられる。OpenCLではこの機構により、プログラム作成(コンパイル)環境とは異なる実行環境にプログラムを持って行った場合にも実行時にコンパイラが最適なコードを生成することができるため、常に最高の性能が得られるとしている。なお、コンパイル済みの計算カーネルをロードする機構も備えられている。

2.3 OpenCL プログラムの例

本節では、実際にOpenCLプログラムソースコードおよび実行手順を見てみることにする。動作確認にはCentOS 5.5 x86_64およびCUDA 3.2RCを導入済みのTeslaC2050搭載PCを用いた。残念ながら対象ハードウェアが手元になかったため、AMD社製GPUでも同一のプログラム(ソースコード)が利用できるかの確認はできていない。

ソースコード1はOpenCLを用いた単純な配列計算プログラム(入力配列の各要素にスカラー値を掛け合わせ、出力配列として返す)におけるホストコード(CPU側で動作するプログラム)の例である。このプログラムは実際に動くプログラムであり、省略はしておらず、ある程度のエラー処理も記述してある。本プログラムの流れは以下のようになっている。

1. 16行目～24行目：GPUの初期化処理
2. 25行目～28行目：コマンドキューの作成
3. 29行目～42行目：計算カーネルのコンパイル・ロード
4. 43行目～55行目：CPU/GPUそれぞれのメモリ準備およびCPUからGPUへのデータ転送
5. 56行目～63行目：GPU演算開始指示およびGPU演算終了待ち
6. 64行目～65行目：GPUからCPUへのデータ転送
7. 66行目～68行目：計算結果の確認
8. 69行目以降：終了処理

ソースコード 1　arraytest.cpp

```
1  #include <oclUtils.h>
2
3  #define DATA_LENGTH 16
4  cl_context cxGPUContext;
5  cl_kernel kernel;
6  cl_command_queue commandQueue;
7
8  #define CHK_DO(name,o) ciErrNum=o;if(ciErrNum!=CL_SUCCESS){printf(name);printf("␣failed\n"
       );return(−1);}
9  #define CHK_ERR(name) if(ciErrNum!=CL_SUCCESS){printf(name);printf("␣failed\n");return
       (−1);}
10
11 int main(int argc, char** argv){
12   cl_platform_id cpPlatform = NULL;
13   cl_uint ciDeviceCount = 0;
14   cl_device_id *cdDevices = NULL;
15   cl_int ciErrNum = CL_SUCCESS;
16   // get platform
17   CHK_DO("oclGetPlatformID", oclGetPlatformID(&cpPlatform));
18   // get devices
19   CHK_DO("clGetDeviceIDs1", clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 0, NULL, &
         ciDeviceCount));
20   cdDevices = (cl_device_id*)malloc(ciDeviceCount*sizeof(cl_device_id));
21   CHK_DO("clGetDeviceIDs2", clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, ciDeviceCount,
         cdDevices, NULL));
22   // get context
23   cxGPUContext = clCreateContext(0, ciDeviceCount, cdDevices, NULL, NULL, &ciErrNum);
24   CHK_ERR("clCreateContext");
25   // create command−queue
26   cl_device_id device = oclGetDev(cxGPUContext, 0);
27   commandQueue = clCreateCommandQueue(cxGPUContext, device,
         CL_QUEUE_PROFILING_ENABLE, &ciErrNum);
28   CHK_ERR("clCreateCommandQueue");
```

```
29    // program setup
30    size_t program_length;
31    const char* source_path = "gpu.cl";
32    char *source = oclLoadProgSource(source_path, "", &program_length);
33    if(!source){printf("oclLoadProgSource␣failed(%s)\n", source_path);return -2000;}
34    // create the program
35    cl_program cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)&source, &
          program_length, &ciErrNum);
36    CHK_ERR("clCreateProgramWithSource");
37    free(source);
38    // build the program
39    CHK_DO("clBuildProgram", clBuildProgram(cpProgram, 0, NULL, "-cl-fast-relaxed-math", NULL,
          NULL));
40    // Create Kernel
41    kernel = clCreateKernel(cpProgram, "arraytest", &ciErrNum);
42    CHK_ERR("clCreateKernel");
43    // setup data
44    cl_mem d_A;
45    cl_mem d_R;
46    float* h_A_data = (float*)malloc(sizeof(float)*DATA_LENGTH);
47    for(int i=0; i<DATA_LENGTH; i++)h_A_data[i] = (float)(i+1);
48    float* h_R_data = (float*)malloc(sizeof(float)*DATA_LENGTH);
49    d_A = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR, sizeof(
          float)*DATA_LENGTH, h_A_data, NULL);
50    d_R = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, sizeof(float)*DATA_LENGTH,
          NULL, NULL);
51    float value = 2.0f;
52    // set args
53    clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_R);
54    clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&d_A);
55    clSetKernelArg(kernel, 2, sizeof(cl_float), (void*)&value);
56    // run kernel
57    cl_event GPUExecution;
58    size_t localWorkSize[] = {4};
59    size_t globalWorkSize[] = {DATA_LENGTH};
60    clEnqueueNDRangeKernel(commandQueue, kernel, 1, 0, globalWorkSize, localWorkSize, 0, NULL, &
          GPUExecution);
61    clFlush(commandQueue);
62    // sync
63    clFinish(commandQueue);
64    // bloking readback
65    clEnqueueReadBuffer(commandQueue, d_R, CL_TRUE, 0, sizeof(float)*DATA_LENGTH, h_R_data,
          0, NULL, NULL);
66    // check result
67    printf("before:␣"); for(int i=0; i<DATA_LENGTH; i++){printf("␣%.2f", h_A_data[i]);}printf("\n");
68    printf("after␣:␣"); for(int i=0; i<DATA_LENGTH; i++){printf("␣%.2f", h_R_data[i]);}printf("\n");
69    // release mem and event
70    clReleaseMemObject(d_A);
71    clReleaseMemObject(d_R);
72    clReleaseEvent(GPUExecution);
73    // cleanup
74    ciErrNum |= clReleaseKernel(kernel);
75    ciErrNum |= clReleaseCommandQueue(commandQueue);
76    ciErrNum |= clReleaseProgram(cpProgram);
77    ciErrNum |= clReleaseContext(cxGPUContext);
78    CHK_ERR("release");
79    free(h_A_data);
80    free(h_R_data);
81    return 0;
82  }
```

GPUの初期化処理については、PlatformIDの取得、Device数の取得およびDeviceIDの取得、およびContextの作成を行っている。もちろん、ソースコード2のように数行のコードを追加すれば搭載されている全GPUの情報を確認して選択するなどの処理も可能である。やや手順が煩雑ではあるが特別難しい処理ではなく、基本的に一度書いたものを使い回せる記述である。

終了処理についても、特筆すべき点はない。

ソースコード 2　arraytest.cpp(デバイスの確認)

```
1   // get context と create command−queue の間に挿入する
2   size_t nDeviceBytes;
3   CHK_DO("clGetContextInfo", clGetContextInfo(cxGPUContext, CL_CONTEXT_DEVICES, 0,
        NULL, &nDeviceBytes));
4   ciDeviceCount = (cl_uint)nDeviceBytes/sizeof(cl_device_id);
5   if(ciDeviceCount == 0){printf("no␣devices␣(return␣code␣%i)\n", ciErrNum);return −1;}
6   // check all devices (get device and print the device name)
7   for(unsigned int i = 0; i < ciDeviceCount; ++i){
8     // 変数device にデバイスの情報が得られるため、これを用いて使用する GPU を決めれば良い
9     cl_device_id device = oclGetDev(cxGPUContext, i);
10    printf("Device␣%d:␣", i);
11    oclPrintDevName(LOGBOTH, device); // GPU 名を表示する例
12    printf("\n");
13  }
```

メモリ確保やデータ転送については、clSetKernelArgによる引数設定などにやや手間はかかるが、適切なパラメタとともにAPIを呼び出すのみで良い。GPU演算開始指示および終了待ちについても同様にAPIを用いる。このプログラムではデータ転送に同期転送(ブロッキング転送)を使用しているが、若干の記述変更で非同期転送も可能であることを付け加えておく。

計算デバイスに処理を行わせる上で注意が必要なのがclEnqueueNDRangeKernelに渡す第5引数および第6引数(localWorkSizeとglobalWorkSize)、計算カーネルの並列度である。CUDAでは計算カーネルの実行時にThreadとBlockの数を指定すると、総数でThread数×Block数の並列度で計算が行われた。一方OpenCLでは総数とWork-Itemの数、すなわちCUDAにおけるThread数×Block数とThread数を指定する。CUDAプログラムを単純に模倣しようとすると勘違いすることもある(筆者は勘違いをしてしばらく悩んだ) ので注意が必要である。なお各WorkSizeに指定可能な値にはいくつか制限があるが、詳細についてはリファレンスを参照していただきたい。

以上の処理については、本連載の第2回で解説したCUDAプログラム(CUDA C)と比べると複雑で手間がかかる記述であり、感覚的にはCUDA CよりもむしろCUDA Driver APIに近い。しかしいずれも処理の流れ自体はCUDAに類似しており、CUDAプログラミングの知識や経験とOpenCLのAPI資料さえあれば理解・記述することができるだろう。

一方でOpenCLとCUDAの目立った違いとしては、コマンドキューの作成や計算カーネルのコンパイルが挙げられる。

コマンドキューはホストから計算デバイスに命令を送るための仕組みである。その名の通りキューであり、各計算デバイスは割り当てられたキューに入れられた処理を次々とこなしていくことになる。今回は単純なサンプルプログラムであるため、計算カーネルの実行と計算結果のリードバックを行っているのみである。

ちなみにこの連載中では紹介しなかったが、CUDAにもコマンドキューに近いものとしてStream(ストリーム)がある。StreamはGPUによる計算とCPU-GPU間のデータ転送をオーバーラップする際に必要なため、多くのアプリケーションにおいて利用する意義がある。Streamについてはプログラミングガイドやサンプルを参照していただきたい。

カーネルのコンパイルに関する記述は、これまでのCUDA Cと類似したプログラム記述と比べて手間のかかる処理であると言わざるを得ない。手順としては以下の通りである。

1. ヘッダーファイル(今回のサンプルでは用いていない)やソースファイルをoclLoadProgSource関数でセット
2. clCreateProgramWithSource関数でセットしたソースを読み込む
3. clBuildProgram関数でコンパイルおよびリンク
4. clCreateKernel関数でカーネル関数を設定

今回のように単一ソースコードからなる単純な計算カーネルを利用する場合はともかく、計算カーネルの規模が大きくなった場合にはこれらコンパイルに関する一連の処理も伸びてしまい、さらなる手間となることが懸念される。

つづいて、計算カーネルをソースコード3に示す。CUDAと同様に、この計算カーネル(関数)が計算デバイス上の計算エレメントによって一斉に実行されることになる。さらに、具体的な記述方法こそ異なるものの、各Work-Itemごとに一意に得られるIDを用いてそれぞれ異なるデータを処理するという基本的な考えもCUDAと同様である。このプログラムは配列の 1 要素にスカラー値を掛け合わせているだけであるが、デバイスレベルで一意なIDを得ることができる関数get_global_idを用いることで各Work-Itemがそれぞれ1つずつの配列要素を計算し、全体として配列に対する計算を達成している。

ソースコード 3　gpu.cl

```
1 #define DATA_LENGTH 16
2 __kernel void arraytest(__global float *R, __global float *A, float value){
3   int i;
4   i = get_global_id(0);
5   R[i] = A[i] * value;
6 }
```

計算カーネルに見られる指示子の違いについては、関数や変数に対する指示子がCUDAと若干異なるのはともかく、関数の引数に対しても指示子が必要な点はCUDAとの明確な違いである。CUDAは引数に指示子が無いため記述がより簡潔である反面、メモリの扱いがコンパイラ任せの部分があり、特に初期のCUDAコンパイラでは変数が実際にどのメモリに配置されるかわからない旨の警告がしばしば見受けられた。(そして、実行時エラーに見舞われることが多々あった。) 一方でOpenCLは変数のメモリ配置をより細かく指定できることになるため、プログラマが意図したとおりに記述・最適化が行いやすいケースもあるだろう。

最後に、実際にOpenCLプログラムをコンパイル・実行する際の手順について確認しておく。CUDAプログラムをコンパイルする際には専用のコンパイラnvccを利用した。一方でOpenCLについては専用のコンパイラが存在せず、図3に示すようにgcc(g++)等のコンパイラを用いてライブラリ指定を行いつつ通常のコンパイルを行えばよい。もちろん、実行時に計算カーネルをコンパイルする都合上、実行時に適切な位置に計算カーネルのソースコードが配置されている必要がある。また、計算カーネルに文法エラーなどがあった場合にはプログラム中のカーネルコンパイルでエラーすることになる。

以上、駆け足ではあるがOpenCLの概要と使い方を説明した。OpenCLもCUDA同様に、正しく動くプログラムを作成するだけならそう難しくはないが、CUDA(CUDA C)と比べると記述量が多く手間がかかってしまう。CUDA Driver APIに対するCUDA C、もしくはOpenGLにおけるGLUTのようなプログラム記述を容易にするライブラリ(ラッパー)が普及すれば、OpenCLを利用するプログラマの負担も減るだろう。また、正しく動くだけならともかく、高速なプロ

```
$ ls
arraytest.cpp gpu.cl
$ g++ -O3 -m64 -o arraytest arraytest.cpp -lOpenCL \
-I/path_to_cudasdk_3.1/OpenCL/common/inc -I/path_to_cudasdk_3.1/shared/inc \
-L/path_to_cudasdk_3.1/OpenCL/common/lib -L/path_to_cudasdk_3.1/shared/lib \
-loclUtil_x86_64 -lshrutil_x86_64
$ ls
arraytest arraytest.cpp gpu.cl
$ ./arraytest
before:  1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00（以下省略）
after :  2.00 4.00 6.00 8.00 10.00 12.00 14.00 16.00 18.00 20.00（以下省略）
```

図 3　OpenCL プログラムのコンパイルと実行の例

グラムを作成するには知識と手間が必要である。さらに、同じプログラムが複数のハードウェアで実行できるとはいえ、同じプログラムが複数のハードウェアで高速に動作するとは限らず、ハードウェアに合わせた最適化プログラミングが必要になることに留意せねばならない。

3　その他のGPGPUプログラミング環境

3.1　既存の並列化プログラミング環境を利用した GPGPU プログラミング環境

GPGPUが登場するよりも前から、様々な並列処理プログラムが様々な手段で記述されてきた。たとえばC/C++言語を用いたプログラミングにおいては、スレッド並列化にpthreadやOpenMPが、またプロセス並列化にMPIが長い間利用されてきている。GPGPUプログラミングも並列化プログラミングの一種ととらえられることから、既存の並列化プログラミング環境を用いてGPGPUプログラミングが行えれば、アプリケーションプログラマの手間や負担が軽減されることが期待できる。

そこで筆者らは、OpenMP指示子を用いて記述されたプログラムからCUDAプログラムを作成するプログラム変換ツールOMPCUDA[3]の作成を行っている。OMPCUDAは、OpenMP指示子の挿入された対象プログラムからparallel for指示子(forループの並列化を意味する指示子)で指定されたforループを抽出し、GPU上で実行されるカーネルに変換したうえで、カーネルが正しくGPU上で実行できるようにCPU-GPU間のデータ転送等の処理を挿入する、という処理を行う。OpenMPにおいてはループの各処理を各スレッドが並列に実行するが、OMPCUDAにおいてはループの各処理をGPU上のSPが並列に実行することになる(図4)。

OMPCUDAの実装は、既存のOpenMP処理系であるOMNI OpenMP compiler(以下OMNI)[2]を拡張して行っている。OMNIの構造およびOMPCUDAとの関係を図5に示す。OMNIが複数の言語に対するフロントエンドや中間表現を操作するためのツールキットを備えているため、OMPCUDAは主に中間表現レベルでの実装で機能を達成している。現在のところOMPCUDAは入力されたソースコードをシンプルにCUDA化しており、GPU上で高い性能が得られるような書き換えは行えていない。またベースとなっているOMPCUDAも最近のOpenMPやC/C++およびFortranの仕様に対応していない。そのためOMPCUDAは実用的なアプリケーションにおいて利用するには力不足であり、今後さらなる実装が必要である。

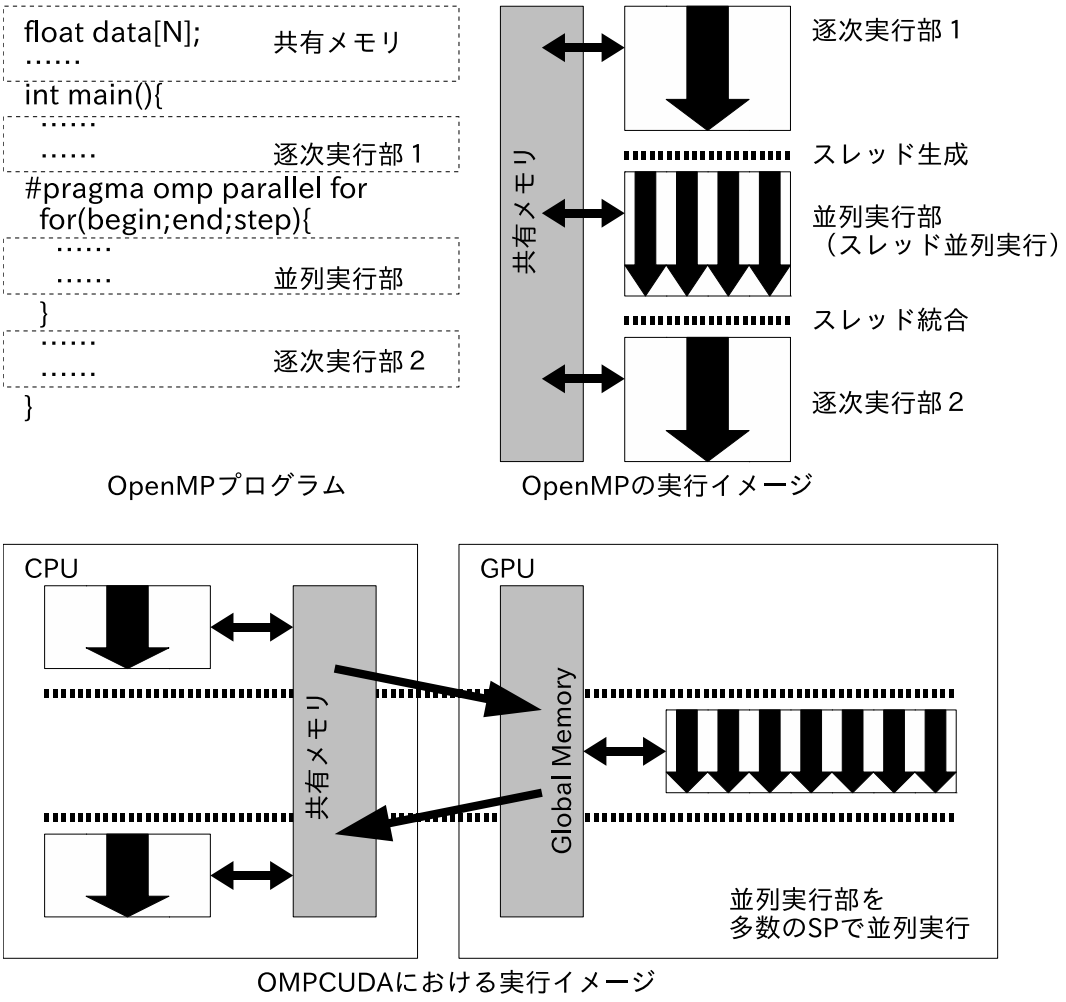

図 4　OpenMP プログラムの実行イメージと OMPCUDA の関係

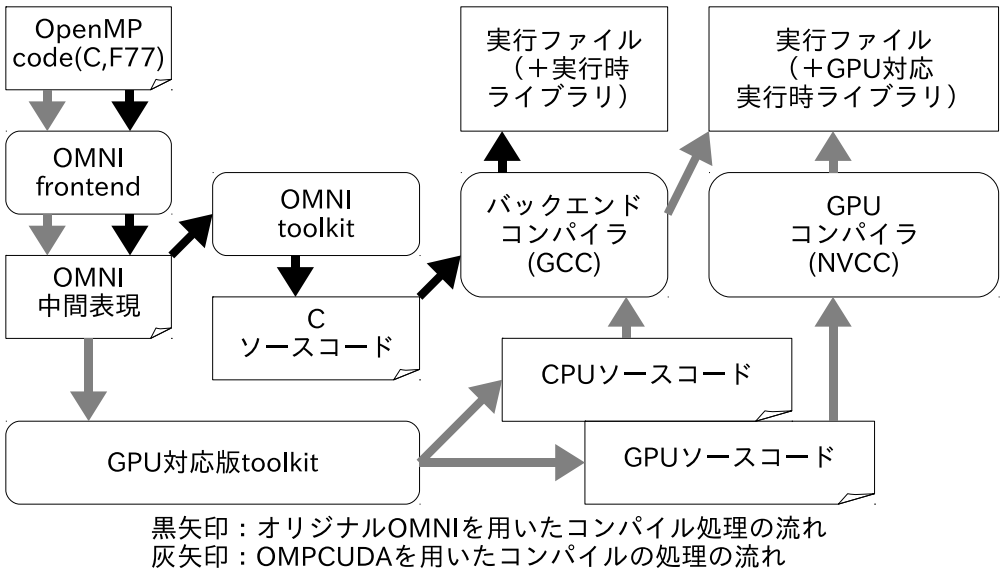

図 5　OMNI と OMPCUDA の関係

OMPCUDA以外にも、例えばLeeら[5]が同様にOpenMPを用いてCUDAを利用するソフトウェアの開発を行っている。またOpenMPではないが、指示文を用いたGPGPUプログラミング環境としてはPGI社のアクセラレータコンパイラ[4]や、CAPS社のHMPP[6]などがある。

3.2 様々なプログラミング環境から GPGPU を利用するためのソフトウェア

CUDAとOpenCLはC/C++言語を拡張した言語によるプログラミング環境を提供している。C/C++言語は確かに利用者が多く普及している開発環境であるが、HPCの分野ではC/C++以外にFortranも多く利用されており、また他の分野ではJava, Perl, Python, Ruby, .NETなど様々なプログラミング環境(言語)が利用されている。そのため、これらの様々な使い慣れたプログラミング環境からCUDAやOpenCLを使いたいという要求も多く、実際に様々なプログラミング環境からGPUを利用するためのソフトウェアが開発されている。本節ではこうしたいわゆる「言語バインディング」の例をいくつか紹介する。

PyCUDA[7], PyOpenCL[8]

PyCUDAおよびPyOpenCLはPythonからCUDAやOpenCLを利用するためのバインディングである。いずれもPythonプログラム中に文字列としてCUDAやOpenCLのプログラムを記述しておき、提供されているクラスを介してCPU-GPU間のデータ転送やGPUの制御を行う。そのため、Pythonの言語仕様で記述することができるのはホスト側で動作するプログラムのみとなる。ベースとなるライブラリはC++で記述されている。

Ruby-OpenCL[9]

Ruby-OpenCLもPyOpenCLと同様に、Rubyプログラム中に文字列としてOpenCL計算カーネルを記述しておき、ラッパー関数を用いてGPUにアクセスするタイプの言語バインディングである。上記のPythonバインディングと同様に、Rubyの言語仕様で記述することができるのはホスト側で動作するプログラムのみとなる。

JCuda[10]

JCudaはJavaからCUDAを利用するためのバインディングである。上記のPython/Rubyバインディングとは異なり、別途作成したGPUバイナリ(PTX)を呼び出して利用するためのラッパーと、CUDAが提供する各種ライブラリ(CUBLAS, CUFFTなど)を呼び出すためのAPIが主体となっている。

CUDA Fortran[11]

CUDA FortranはFortranからCUDAを利用するためのバインディングであり、商用コンパイラであるPGIコンパイラに含まれている。CUDA FortranはC/C++とFortranが似ている(単純に置き換えられる記述が多い)こともあり、実際のプログラム記述もほぼCUDA Cの記述をFortranの仕様で置き換えたものと言える。

以上いくつかの言語バインディングを挙げたが、基本的には計算カーネルの記述にCUDAやOpenCLを使う必要があり、「様々なプログラミング言語」のみでGPUを活用することは難

しいのが現状である。逆に言えば、GPUカーネル以外は使い慣れた様々な言語で記述できる、使い慣れた言語で記述したプログラムの中で計算量が多い特定の部分のみをGPUに任せるといった使い方に適していると言えるだろう。

3.3 ストリーミング言語

GPGPUの発展とは独立した平行処理/並列処理のパラダイムとして、ストリーミング処理およびそれを記述するストリーミング言語の研究が行われてきた。GPUによる高性能並列計算の可能性とプログラミングの難しさが知られるようになると、ストリーミング言語を用いたGPUプログラミングの研究が注目された。残念ながらCUDAが登場したことでストリーミング言語を用いたGPUプログラミングへの注目度は下がってしまったが、本節ではGPUに対応したストリーミング言語のいくつかを簡単に紹介する。

ストリーミング処理は、計算対象のデータをストリーム(入力ストリーム/出力ストリーム)、計算内容をカーネルと定義し、プロセッサがカーネルを次々にストリームへと作用させていくという概念を持っている。これに対応するストリーミング言語も、言語仕様としてストリームとカーネルを明示的に記述する言語仕様を採用している。なお、CUDAもストリーミング言語の一種と見なされることがある。

特に2005年頃にはGPUに対応した複数のストリーミング言語が登場した。当時注目されたストリーミング言語の例としては、BrookGPU[12]やRapidMind[14]などが挙げられる。

BrookGPUは米Stanford Universityで開発されたオープンソースのストリーミング言語である。図6にBrookGPUを用いた配列加算プログラムの例を示す。この例ではカーネル(kfunc)とストリーム(streamRead,streaWrite)が明示的に示されていることがよくわかる。図中のカーネルでは2つの変数を加算して別の変数への代入を行っているが、各変数は配列であり、実行時には配列の各要素に対する演算がGPU上の多数の演算器によって独立に並列処理される。

BrookGPUはカーネルを実行するバックエンドとしてDirectX, OpenGL, CPU, CTM(Close to the Metal、AMDが当時プッシュしていた開発環境)に対応することで、単一ソースコードから様々なハードウェアに対応した。後にAMD社は自社GPU向けの開発環境としてBrookGPUを元にしたBrook+[13]を提供している。

RapidMindはUniversity of Waterlooで開発されたオープンソースのメタプログラミング言語Sh[15]を元に加RAPIDMIND社が開発したストリーミング言語である。図7にRapidMindを用いたプログラムの例を示す。RapidMindもBrookGPUと同様にカーネルとカーネルへの入出力を明示的に指定する一方で、カーネルの記述方法はよりRapidMindに特有の記述になっている。

PetaFLOPS・ExaFLOPSという高性能な計算環境の時代を迎えて、またGPGPUを含めたヘテロジニアス環境の活用が重要な時代を迎えて、プログラムをどのように記述するか、何を用いて記述するかは改めて難しい問題として認知されている。ストリーミング言語が再度注目されることになるかはわからないが、各計算をどこで行うかやデータの配置・移動をどうするかを記述するという考え方は今後の並列化プログラミング言語に引き継がれるだろう。

以上、第六回(最終回)の今回は本連載でこれまでに扱ってきたCUDAとは違う様々なプログラミング環境についての解説を行った。現在のGPGPUにおいてはCUDAが非常に大きな影響

```
// カーネル
kernel void kfunc (float x<>,
 float y<>, out float z<>) {
  z = x + y;
}
int main() {
  float a<100>;
  float b<100>;
  float c <100>;
  // 入力ストリームの指定
  streamRead(a, data1);
  streamRead(b, data2);
  // カーネルの実行
  kfunc(a, b);
  // 出力ストリームの指定
  streamWrite(c, result);
  return 0;
}
```

図 6　BrookGPU のソースコード

```
int main() {
  // カーネル
  Program kfunc=BEGIN {
    In<Value3f>x, y;
    Out<Value3f>z;
    z = x + y;
  } END;
  // 入出力ストリームの指定
  Array<1, Value3f> a(512);
  Array<1, Value3f> b(512);
  Array<1, Value3f> c(512);
  // カーネルの実行
  c = kfunc(a, b);
  return 0;
}
```

図 7　RapidMind のソースコード

力を持っているのは事実であり、今後もしばらく大きな影響力を維持することは間違いないだろう。一方で、OpenCLもNVIDIA社製のGPU以外でも、さらにはGPU以外でも利用可能であることから注目されており。さらにC/C++言語以外の様々な方法でGPGPUを利用できることは、GPGPUがさらに広く利用される上で重要である。

この連載は、年々注目と影響力の高まっているGPGPUに関する情報を本センターのユーザに対して提供することを目的として一年にわたって続けてきた。本稿を執筆している2010年11月上旬現在、既に中国のスーパーコンピューターTianhe-1A (2009年11月のTOP500でRadeonHDを用いて5位を達成した「天河一号」の更新版)が14,336個のCPUと7,168枚のTeslaM2050を駆使してLINPACKベンチマークで2PFLOPSを突破し、次回のTOP500(2010年11月版)の頂点に立つことが確実だと報じられている。また東工大のTSUBAMEも4,224枚のTeslaM2050を搭載したTSUBAME 2.0にリプレイスされて稼働を開始しており、TOP500およびGreen500の上位ランクインが期待されている。これらの結果は本記事が公開される頃には判明しているだろう。GPGPUは現在も急速に発展している技術であり、今後もさらにGPUの性能を活用するスーパーコンピューター「GPUスパコン」は増加するだろう。本センターは現時点でGPUを搭載した計算資源の提供を行っていないが、現在の動向からすれば本センターが近い将来GPUスパコンを提供することも考えられる。来るべきGPUスパコンに向けて、という意味も込めて、今後も本スパコンニュースにてGPGPUに関する情報を不定期に提供していく予定である。

参考文献

[1] OpenCL - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/

[2] M.Sato, S.Satoh, K.Kusano, and Y.Tanaka. Design of OpenMP Compiler for an SMP Cluster.

In EWOMP ’ 99, pp. 32—39, 1999.

[3] 大島聡史, 平澤将一, 本多弘樹. OMPCUDA : GPU 向けOpenMP の実装. HPCS2009 2009 年ハイパフォーマンスコンピューティングと計算科学シンポジウム, pp.131—138, 2009.

[4] PGI. PGI Accelerator Compilers, http://www.pgroup.com/resources/accel.htm

[5] Seyong Lee, Seung-Jai Min, Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp.101-110, 2009.

[6] CAPS. HMPP Workbench, http://www.caps-entreprise.com/fr/page/index.php?id=49\&p_p=36

[7] PyCUDA, http://mathema.tician.de/software/pycuda

[8] PyOpenCL, http://mathema.tician.de/software/pyopencl

[9] Ruby-OpenCL, http://ruby-opencl.rubyforge.org/

[10] jCuda, http://www.jcuda.org/

[11] CUDA Fortran, http://www.pgroup.com/resources/cudafortran.htm

[12] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. SIGGRAPH 2004, 2004.

[13] AMD. Brook+. SC07 BOF Session presentation, November 2007.

[14] Michael D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In GSPx Multicore Applications Conference, 2006.

[15] Michael McCool and Stefanus Du Toit. Metaprogramming GPUs with Sh. A K Peters Ltd, 2004.