OpenCLと共通コード

（1）基礎編

Typo訂正版: 4 Dec 2010

Hideo Matsufuru (KEK)

3 Dec 2010 共通コードミーティング@筑波大計算科学研究センター

目的と目次

- OpenCLはこれからのマルチコアプログラミングの主流（かも）
  - GPU (NVIDIA, AMD/ATI)
  - Cell B.E.
  - 組み込みプロセッサ
- 共通コードとして、OpenCLに対応するために必要な準備を考える

目次

- 基礎編（今回）
  - OpenCLとは
- 実践編
  - 高速化の実際
  - Wilson Solver によるケーススタディ
- 応用編
  - 共通コードにどう組み込むか

参考資料

[1] 土山了士、他、株式会社フィックスターズ「OpenCL入門」（インプレスジャパン, 2010）

[2] 池田成樹「OpenCL並列プログラミング」（カットシステム, 2010）

- Khronos group: http://www.khronos.org/#tab-opencl
- NVIDIA: http://developer.nvidia.com/object/opencl.html

OpenCLとは

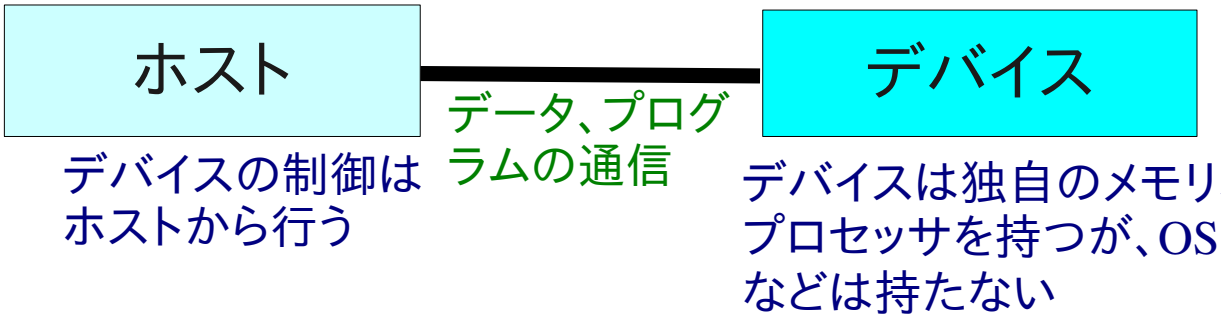
- ヘテロジニアスな並列計算機環境に適した並列プログラミングのためのフレームワーク
- ハードウェアモデル：ホストとデバイス
  - ホスト(CPU)で走るプログラムから、一部の処理をデバイス（アクセラレータ）に任せる
  - スレッド並列アーキテクチャ (SIMT: single instruction, multiple thread)
- Khronos Groupによる標準化
  - 参加企業: AMD, Apple, IBM, Intel, NVIDIA, etc
- 「フレームワーク」⇒ 実装は各プラットフォームによる
  - NVIDIA: CUDA環境の一部として提供、Tesla, GeForceを利用可
  - AMD ATI:
  - IBM: Cell B.E.を利用可
  - FOXC: Fixstars社によるコンパイラ
  - Apple: MacOS X は標準で提供

OpenCLとは

- 仕様
  - ランタイムAPI（ホストからデバイスを制御）
  - OpenCL C言語（デバイス用コードを記述）
- プログラミングモデル
  - データ並列
  - タスク並列
- いろいろな環境に対応
  - 覚える文法が一つですむ
  - とりあえず走るものは、一般的に書ける（かも）
- 高性能化も可能
  - それぞれのハードウェア特性を把握する必要
- ハードウェアを細かく制御できる文法
  - アクセラレータのメモリ領域の設定、メモリへのデータ転送など
  - 抽象化は？

ホストとデバイス

ハードウェアモデル



NVIDIA Tesla (GPGPU) の場合

ソフトウェアモデル

ハードウェアモデル

デバイスのメモリ構造

- グローバルメモリ: すべてのワークアイテムから読み書き可
  - ホストから読み書き可
- コンスタントメモリ: すべてのワークアイテムから読み込み可
  - データ書き込みはホスト側から
  - NVIDIA GPU のコンスタントメモリ
- ローカルメモリ: ワークグループ内のワークアイテム間で共有
  - スクラッチパッドメモリ: キャッシュより小さく、ソフトウェアで制御
  - NVIDIA GPU の共有メモリ
  - Cell B.E. のローカルストア
- プライベートメモリ: ワークアイテム専用（レジストリを想定）

プログラミングモデル

ホスト用コードとデバイス用コード（カーネル）から構成

- スレッド並列 (SIMT: single instruction multiple thread)
- ホストプログラム
  - プログラムの実行を制御
  - デバイスをコントロール：OpenCLランタイムAPI
- カーネル
  - 実行時にロード、コンパイル (オンラインコンパイル)
    - 先にコンパイルしておくことも可能: FOXCなど　（オフラインコンパイル）
  - OpenCL C言語で記述
  - データ並列/タスク並列の、それぞれの処理の単位(thread)を記述
  - 並列に実行

ホストコード

- ホスト上で動作
  - デバイスの制御にOpenCLランタイムAPIを使用（関数呼び出し）
- 手順
  - (1) デバイスを使う準備
    - プラットフォーム、デバイスの特定、コンテキスト、コマンドキューの作成
  - (2) プログラムの準備
    - ソースコードを読み込んでコンパイル、カーネル関数を指定
  - (3) メモリ領域の設定
    - デバイス上のメモリオブジェクトを設定
  - (4) デバイスの使用
    - メモリ転送、カーネルの実行
  - (5) オブジェクトの解放
- Include file:

```
#include <CL/cl.h>
```

MacOS では場所が違うので注意!

ホストコード

(1) デバイスを使う準備

緑の関数がOpenCL API

```
// get information of device platform
cl_platform_id    platform_id;
cl_device_id      device_id;
cl_uint ret, num_platforms, num_devices;

ret=clGetPlatformIDs(1, &platform_id, &num_platforms);
printf(" number of platforms: %d\n", num_platforms);

ret=clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT,
                1, &device_id, &num_devices);
printf(" number of devices: %d\n", num_devices);

// create OpenCL context
cl_context context;
context=clCreateContext(NULL, 1, &device_id,
                                 NULL, NULL, &ret);

// create command queue
cl_command_queue command_queue;
command_queue=clCreateCommandQueue(context, device_id,
                                           0, &ret);
```

cl.h で定義されるタイプ

プラットフォームを特定
（ハンドルを取得）

デバイスを特定
（デバイスハンドルを取得）

コンテキスト（実行環境）
を作成

コマンドキューを作成
（デバイス上のタスク実行
はこのキューに投入）

ホストコード

(2) プログラムの準備

```
FILE *fp;
char *source_str;
size_t source_size;
char filename[] = "./mult_float.cl";

fp = fopen(filename, "r");
source_str=(char*)malloc(MAX_SOURCE_SIZE);
source_size=fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

// create kernel program from source file
cl_program program;
program=clCreateProgramWithSource(context, 1,
          (const char**)&source_str,
          (const size_t *)&source_size, &ret);

// build kernel program
ret=clBuildProgram(program, 1, &device_id,
                                NULL, NULL, NULL);

// create OpenCL kernel
cl_kernel clmult;
clmult=clCreateKernel(program, "mult_all", &ret);
```

ソースコードを読み込む
（コンパイル済みのオブジェクトを読み込むこともできる。ここでは実行時にコンパイル：「オンライン・コンパイル」）

ソースコードをプログラムに指定

プログラムをコンパイル

コンパイルしたプログラムから、関数 “mult_all”をカーネルに指定

ホストコード

(3) メモリ領域の設定

```
// create memory object on device
 cl_mem  Vmobj = NULL;
 cl_mem  Wmobj = NULL;
 cl_mem  Umobj = NULL;

 Vmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
            Nvst*sizeof(float), NULL, &ret);

 Wmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
            Nvst*sizeof(float), NULL, &ret);

 Umobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
            Ndf*Nst*4*sizeof(float), NULL, &ret);
```

デバイス上のメモリ領域を設定

ホストコード

(4) デバイスの使用

```
// write data on device memory buffer
ret=clEnqueueWriteBuffer(command_queue, Wmobj, CL_TRUE,
            0, Nvst*sizeof(float), wf, 0, NULL, NULL);
ret=clEnqueueWriteBuffer(command_queue, Umobj, CL_TRUE,
            0, Ndf*Nst*4*sizeof(float), uf, 0, NULL, NULL);

// set arguments of kernel program
ret=clSetKernelArg(clmult, 0, sizeof(cl_mem), (void *)&Vmobj);
ret=clSetKernelArg(clmult, 1, sizeof(cl_mem), (void *)&Umobj);
ret=clSetKernelArg(clmult, 2, sizeof(cl_mem), (void *)&Wmobj);
ret=clSetKernelArg(clmult, 3, sizeof(float), (void *)&CKs2);
```

デバイスメモリへのデータ転送（コマンドキューにタスクとして投入）

カーネル引数の設定（1つづつ行う必要有）

```
// run kernel code on device
size_t  global_item_size = Nst;
size_t  local_item_size = 1;
ret=clEnqueueNDRangeKernel(command_queue, clmult, 1, NULL,
          &global_item_size, &local_item_size, 0, NULL, NULL);
```

この部分は次ページで

カーネルの実行

```
// read data from device memory buffer
ret=clEnqueueReadBuffer(command_queue, Vmobj, CL_TRUE, 0,
             Nvst*sizeof(float), vf, 0, NULL, NULL);
```

デバイスメモリからのデータ転送

ホストコード

(4) デバイスの使用: カーネルの実行

```
// run kernel code on device
size_t  global_item_size = Nst;
size_t  local_item_size = 1;
ret=clEnqueueNDRangeKernel(command_queue, clmult, 1, NULL,
          &global_item_size, &local_item_size, 0, NULL, NULL);
```

カーネルの実行
（コマンドキューに
タスクとして投入）

- ワークグループとワークアイテム
  - 一定数のワークアイテムでワークグループを構成
    - `local_item_size` で指定
    - グループ内のアイテム数は1以上、上限はプラットフォームに依存
  - ワークグループ数
    - `global_item_size` で指定
    - 上限はプラットフォームに依る
  - それぞれのインデックスは次元を持つ場合がある
  - インデックスの次元や最大値を取得する関数有

ホストコード

(5) オブジェクトの解放

```
// release memory buffer on device
ret=clReleaseMemObject(Vmobj);
ret=clReleaseMemObject(Umobj);
ret=clReleaseMemObject(Wmobj);

// release OpenCL kernel
ret=clReleaseKernel(clmult);

// release OpenCL items
ret=clFlush(command_queue);
ret=clFinish(command_queue);
ret=clReleaseProgram(program);
ret=clReleaseCommandQueue(command_queue);
ret=clReleaseContext(context);
```

デバイス上のメモリオブジェクトの解放

カーネルの解放

コマンドキュー、プログラム、コンテキストの解放
（解放の前にキューのフラッシュも）

カーネルコード

- デバイス上で動作: OpenCL C言語で記述

```
// mult_float.cl
#include "lattsize_ocl.h"

__kernel void mult(__global float *v2, __global float *u,
                   __global float *v1, float CKs){

float vt1[Nvc], vt2[Nvc];
float wt1r, wt1i, wt2r, wt2i;

int ist = get_global_id(0);

int ix = ist % Nx;
int iyzt = ist/Nx;
int nn = (ix+1) % Nx;
int iv = Nvc*ND*ist;
int in = Nvc*ND*(nn + iyzt*Nx);

for(int ic=0; ic < Ncol; ic++){
  vt1[2*ic  ] = v1[2*ic   +0 +in] - v1[2*ic+1 +3 +in];
  vt1[2*ic+1] = v1[2*ic+1 +0 +in] + v1[2*ic   +3 +in];
  vt2[2*ic  ] = v1[2*ic   +1 +in] - v1[2*ic+1 +2 +in];
  vt2[2*ic+1] = v1[2*ic+1 +1 +in] + v1[2*ic   +2 +in];
}
.....
```

__kernel はホストから呼び出せるデバイス上の関数であることを指定

__global はグローバルメモリ上のデータであることを指定;
__global
__constant
__local
__private
が指定可能
（省略すると__private）

グローバルアイテムのIDを取得する関数
（今はワークグループ内のアイテム数=1なのでローカルIDは不使用）

OpenCL C言語

- 標準のC言語(C99)に制限と拡張を加えたもの
- 制限の例　([1] p.102参照)
  - カーネル関数の引数に渡すポインタは、__global, __constant, __local 修飾されたものに限る
  - カーネル関数の引数にはポインタのポインタを渡せない
  - C99の可変長配列、フレキシブル配列は使えない
  - 可変引数マクロは使えない
  - 標準ヘッダは使えない
  - 再帰できない
  - カーネル関数の戻り値はvoidでなければならない
  - double は実装されない場合がある
- int, long などはビット幅固定 (int: 32, long: 64, etc.)
- half 型（16 bit 浮動小数点数、IEEE 754で定義）

OpenCL C言語

- アドレス空間修飾子（__は省略可）
  - __global　　グローバルメモリ
  - __constant　コンスタントメモリ
  - __local　　ローカルメモリ
  - __private　　プライベートメモリ
- 組み込み関数
  - 算術関数、幾何関数
  - ワークアイテム制御関数（ワークアイテムIDの取得など）
- ベクタデータ
  - SIMDユニットを利用するため
  - 例：float2, float4, float8
  - 要素の参照: x,y,z,t, s0, …, sf（s のあと16進数表記）などを使う

利用環境

- 以下では、NVIDIA CUDA OpenCL環境を利用
  - Ajisai server@KEKで利用可能
- CUDA によるコーディングの alternative

Portingの手順

- ホスト用コードを基に、処理をデバイスへ移してゆく
  - これまでのコード → ホスト上で動く
  - デバイス上で動作するコードを別に作成
  - ホストからデバイスを制御するためのAPI
- 現在のコードは、Wilson mult のみデバイスで実行
  - Solver 全体をデバイスでできるように
  - パフォーマンスチューニング
  - Class構造はどうする？
  - このあたりはこれからの課題

Memo

- ホスト用コードとして、割合ベタ書きなものがあると移植が容易
- 格子サイズなどのパラメター: 実行時コンパイルなら、ホストコードが include ファイルを書き出すことも可能か
- 実行時コンパイルの場合、カーネルソースコードの置き場所に気を配る必要有
- デバイス使用のためのアイテム（コンテキスト、プログラムキュー）は、一度だけ作るようにしておきたい（singleton ?）