OpenCL and lattice common code

(1) Foundation

English version: 24 Mar 2011

Hideo Matsufuru (KEK)

Based on the slide at 3 Dec 2010 Lattice QCD common-code meeting

Goals and contents

- OpenCL is (maybe) standard of multi-core programming
  - GPU (NVIDIA, AMD/ATI)
  - Cell B.E.
  - Integrated processor
- Preparation to OpenCL is needed for common-code project

Contents

- Foundation (this time)
  - Introduction to OpenCL
- Performance Tuning (next)
  - Practical performance tuning
  - Case study with Wilson Solver
- Applications
  - How to incorporate into common code system ?

References

[1] R.Tsuchiyama et al., Fixstrars Ltd., "Introduction to OpenCL" (Japanese), Impress Japan, 2010.
[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] S.Ikeda, "OpenCL parallel programming", Cut system, 2010.
[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

What is OpenCL ?

- Parallel computing framework suitable to **heterogeneous** parallel computers
- Hardware model: host and devices
  - From program running on host, part of operation is charged to devices (accelerators)
  - Thread parallel architecture (SIMT: single instruction, multiple thread)
- Standard drawn up by Khronos Group
  - Many companies: AMD, Apple, IBM, Intel, NVIDIA, etc
- ''Framework'' ⇒ implementation depends on platform
  - NVIDIA: provided as a part of CUDA: Tesla, GeForce
  - AMD ATI:
  - IBM: Cell B.E.
  - FOXC: Compiler developed by Fixstars Ltd.
  - Apple: MacOS X provides as default

What is OpenCL ?

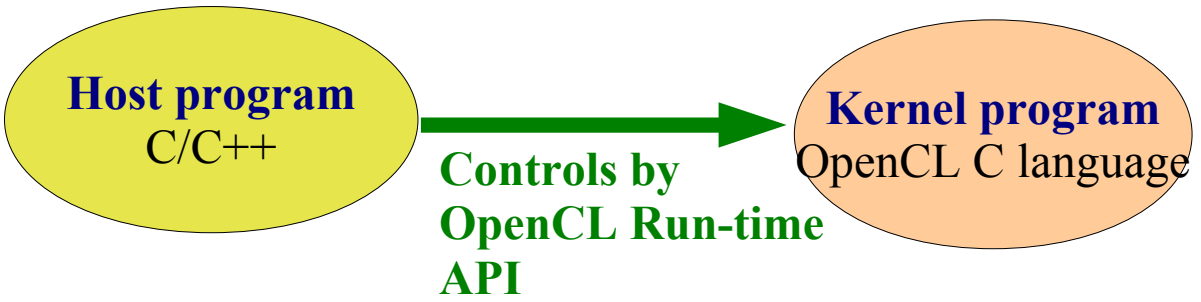
- Specification
  - Run-time API (controls device from host)
  - OpenCL C Language (describes device code)
- Programming model
  - Data parallel
  - Task parallel
- Corresponds various environment
  - Needs to learn just one grammar
  - Can write generic code (hopefully)
- Performance tuning is possible
  - Needs to understand individual hardware features
- Can control hardware in detail
  - Setting accelerator's memory domains, transfer of data to memory, etc.

Host and device

Hardware model

Example:
NVIDIA Tesla (GPGPU)

Software model



Hardware model

Device memory structure

- Global memory: all work items can read/write
  - Host can also read/write
- Constant memory: all work items can read
  - Only host can write
  - Ex: NVIDIA GPU's constant memory
- Local memory: shared by work items in a work group
  - Scratch pad memory: smaller than cache, controlled by software
  - Ex: NVIDIA GPU's shared memory
  - Ex: Cell B.E.'s local store
- Private memory: dedicated to work item (~registry)

Programming model

Composed of host code and device code (kernel)

- Thread parallel (SIMT: single instruction multiple thread)
- Host program
  - Controls execution of program
  - Controls device: OpenCL run-time API
- kernel
  - Loaded and compiled at execution (online compile)
    - Possible to compile in progress: such as FOXC (offline compile)
  - Described by OpenCL C language
  - Describes each thread (unit of operations) in data/task parallel computation
  - Executed in parallel

Host code

- Uses OpenCL run-time API (functions) to control device
- Steps:
  - (1) preparation to use device(s)
    - Specify platform and device(s)
    - Generate context and command queue
  - (2) setup of program
    - Read source code and compile it, specify kernel function(s)
  - (3) setup of memory area
    - Setup memory objects on device(s)
  - (4) use of device(s)
    - Transfer data to memory, execution of kernel
  - (5) free of objects
- Include file:

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

Caution: different path on MacOS

Host code

(1) preparation to use device(s)

Green functions: 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);
```

types defined in cl.h

Specify platform
(get a handle)

Specify device(s)
（get device handle(s)

Generate context
(execution environment)

Generate command queue
（submit tasks on devices
to this queue)

Host code

(2) setup of program

```
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);
```

Read source code
(here online compile; offline
Compile is also possible)

Specify source code as
a program

Compile the program

Specify function "mult_all"
as a kernel in compiled program

Host code

(3) setup memory area

```
// 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);
```

Setup memory area on device

Host code

(4) use of device

```
// 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);

// 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);

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

This part is explained in next page

Transfer data to device memory (submitted to command Queue as a task)

Set up kernel argument (one-by-one)

Execution of kernel

Transfer data from Device memory

Host code

(4) Use of device: execution of kernel

```
// 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);
```

Execution of kernel (submitted to command Queue as a task)

- Work-group and work-item
  - Work-group is composed of some number of work-items
    - Specified by local_item_size
    - #item ≥ 1 within a work-group, max depends on platform
  - #work-group
    - Specified by global_item_size
    - Max depends on platform
  - Each index may have dimension
  - Functions to get index dim. And max value

Host code

(5) free objects

```
// 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);
```

Free memory objects on device

Free kernel

Free command queue, program, and context (after flushing)

Kernel code

- Runs on device: described by OpenCL C language

```
// 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 nn = (ix+1) % Nx;
int iv = Nvc*ND*ist;
int in = Nvc*ND*(nn + (ist/Nx)*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 specifies that this device function can be called from host code

__global specifies these data on global memory;
__global
__constant
__local
__private
are available
(default: __private)

Get ID of global item (now #item within work-group=1 is assumed; then No local ID)

OpenCL C language

- Standard C language (C99) + restriction/extension
- Example of restriction (Cf. [1] p.102 for details)
  - Only pointers with __global, __constant, or __local can be used in arguments of kernel functions
  - Pointer of pointer cannot be used as argument of kernel
  - Variable/flexible array of C99 are not available
  - Variable argument macro N/A
  - Standard header N/A
  - Recursion N/A
  - Returned type of kernel must be void
  - double may not be implemented
- int, long etc with fixed bit width (int: 32, long: 64, etc.)
- half type (16-bit floating point number, IEEE 754)

OpenCL C language

- Address space qualifier (__ can be omitted)
  - __global global memory
  - __constant constant memory
  - __local local memory
  - __private private memory
- Built-in function
  - Arithmetic functions, geometric functions
  - Work-item control functions (getting ID, etc.)
- Vector data type
  - To use SIMD unit
  - Example: float2, float4, float8
  - Element is referred with x,y,z,t, or s0, …, sf (s+hex#)

Test environment

- In the following, NVIDIA CUDA OpenCL environment is used
  - Available on ajisai servers@KEK
- Alternative to CUDA coding

Porting steps

- Based on a host code, move operations to device step-by-step
  - Code so far: runs on host
  - Write alternative device code
  - Use of OpenCL API to control device
- Present code (very preliminary): only use device for Wilson mult
  - Whole solver algorithm should be carried out on device
  - Performance tuning
  - How to construct class structure ?
  - Future subjects

Memo

- Convenient if there is a straightforward -style host code: easy to translate to device code
- Parameters (lattice size etc.): for online compile, host code can write down include file
- For online compile, kernel source code must exist in some specified path (or built in host code with string?)
- Items to use device (context, program queue) should be generated only once (singletom pattern?)