Advanced High Performance Computing

Lecture Notes

jerakrs

February 2018

Abstract

This document is the lecture note of *Advanced High Performance Computing*. The course code is *COMS30006* and the unit director is *McIntosh Smith*.

1 BlueCrystal Phase 4

Phase 4 (bc4login.acrc.bris.ac.uk) is primarily intended for large parallel jobs and for work requiring the Nvidia P100 GPUs. The home page can get more infomations about BlueCrystal Phase 4 and the BlueCrystal Phase 4 User Documentation is the document of BlueCrystal Phase 4.

2 Lattice Boltzmann

Lattice Boltzmann methods (LBM) (or thermal Lattice Boltzmann methods (TLBM)) is a class of computational fluid dynamics (CFD) methods for fluid simulation. The first assigment is using MPI to improve the Lattice Boltzmann algorithm.

2.1 Data Structure

The 'speeds' in each cell are numbered as follows:

```
6  2  5
 \ | /
3–0–1
 / | \
7  4  8
```

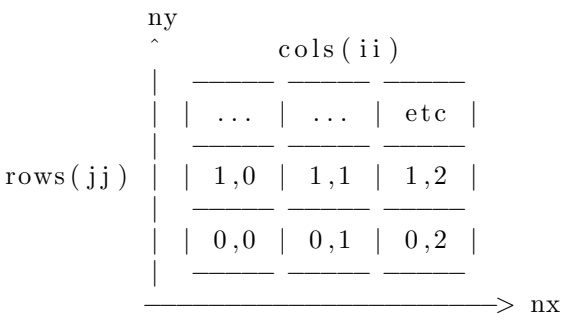
A 2D grid:

```
          cols
       ___ ___ ___
      | D | E | F |
rows  ___ ___ ___
      | A | B | C |
      ___ ___ ___
```

'unwrapped' in row major order to give a 1D array:

```
 ___ ___ ___ ___ ___ ___
| A | B | C | D | E | F |
 ___ ___ ___ ___ ___ ___
```

Grid indicies are:

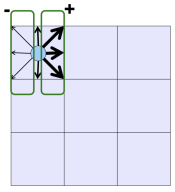


2.2 Basic Code Structure

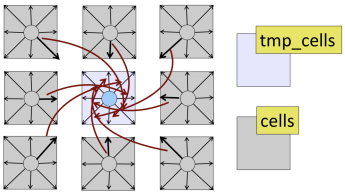
- initialise()
- for each timestep:
  - accelerate flow()
  - propagate()
  - rebound()
  - collision()
  - av_velocity()
- write_values()
- finalise()

Initialise function will load params, allocate memory, load obstacles and initialise fluid particle densities.

Accelerate flow will modify one row of cells, add to the right pointing speeds and subtract from the left pointing speeds.



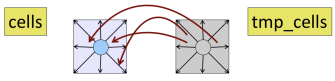
(a) Accelerate Flow



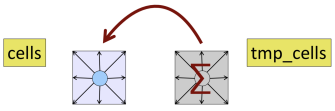
(b) Propagation

Propagation, after a time interval, each particle will move to the neighboring node in its direction.

Rebound, for obstacle cells, it will mirror the speeds in each cell.



(a) Rebound



(b) Collision

Collision, for non-obstacle cells, it will perform 'relaxation' arithmetic in each cell.

3 MPI-3

3.1 Domain Decomposition

Decompose by columns, rows, tiles.

3.2 Communication Pattern

MPI_Send and **MPI_Recv** is asynchronous when there is a system buffer, but blocking if copying large messages into a buffer since its own overheads.

MPI_Isend and **MPI_Irecv** are non-blocking API, but the program needs that the data is exchange completely be- fore the next round. Hence, the **MPI_Wait** is used to wait for full data exchange.

MPI_Sendrecv will send and receive a message.

3.3 Remote Memory Access

Allows one process to access the memory associated with another.

```
int MPI_Win_create(void      *base_addr /* in */,
                   MPI_Aint  size       /* in */,
                   int       disp_unit  /* in */,
                   MPI_Info  info       /* in */,
                   MPI_Comm  comm       /* in */,
                   MPI_Win   *win       /* out */);
```

4 OpenCL

OpenCL is low-level, fast and close to the metal (OpenCL Hand Book). Microprocessor trends, individual processors have many (possibly heterogeneous) cores. The course example of opencl Advanced hpc example.

The BIG idea behind OpenCL is replacing loops with functions (kernels) executing at each point in a problem domain.

Traditional loop

```
void
mul(const  int n,
    const  float *a,
    const  float *b,
           float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

OpenMP

```
void
mul(const  int n,
    const  float *a,
    const  float *b,
           float *c)
{
  int i;
#pragma omp parallel for
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Figure 3: The big idea behind OpenCL

4.1 OpenCL Memory model

- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory /Constant Memory*
  - Visible to all work-groups
- *Host memory*
  - On the CPU

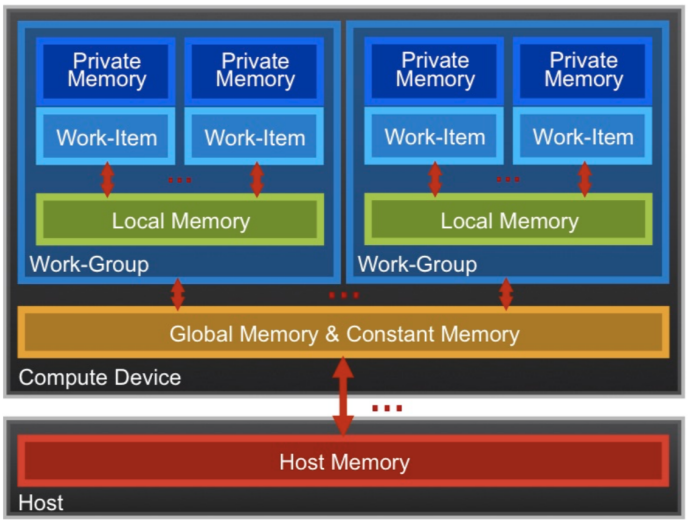


Figure 4: OpenCL Memory Model

4.2 Implementation of OpenCL

OpenCL has two parts, kernel code and host code. For the host code, there are five simple steps: Define the platform, Create and Build the program, Setup memory objects, Define the kernel and Submit commands.

Define the platform

- Grab the first available platform (e.g. *Nvidia*, *Intel*, ...) :

```
err = clGetPlatformIDs(1, &firstPlatformId,
                                  &numPlatforms);
```

- Use the first GPU device the platform provides:

```
err = clGetDeviceIDs(firstPlatformId,
            CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
```

- Create a simple context with a single device:

```
context = clCreateContext(firstPlatformId, 1,
                        &device_id, NULL, NULL, &err);
```

- Create a simple command-queue to feed our device:

```
commands = clCreateCommandQueue(context, device_id,
                                                0, &err);
```

Figure 5: Define the Platform

Create and Build the program define the source code for the kernel-program as a string literal or read it from a file.

- Build the OpenCL program object (this action is performed by the host program, the result is a device program):

```
program = clCreateProgramWithSource(context, 1
          (const char**) &KernelSource, NULL, &err);
```

- Compile the OpenCL program to create a “dynamic library”of kernels from which specific kernels can be referenced later:

```
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);
```

Figure 6: Create and Build the Program

Setup memory objects: The buffers are declared on the host as type **cl_mem**. It needs arrays in host memory hold the original host-side data.

```
cl_mem d_a = clCreateBuffer(context,
    CL_MEM_READ_ONLY,
    sizeof(float)*N, h_a, NULL);
```

(a) Create the Buffer

```
clEnqueueReadBuffer(queue, d_c, CL_TRUE,
      sizeof(float)*N, h_c,
      NULL, NULL, NULL);
```

(b) Set the Data to Buffer

Define the kernel

- Create a kernel object from the kernel function "vadd", which must be present in "program":

```
kernel = clCreateKernel(program, “vadd”, &err);
```

- Set the arguments of the kernel function "vadd" to our memory objects:

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
```

Figure 8: Define the Kernel

Enqueue commands

```
err = clEnqueueNDRangeKernel(commands, kernel, 1,
              NULL, &global, &local, 0, NULL, NULL);
```

Figure 9: Enqueue the Commands

For the kernel code:

- Function qualifiers
  - __kernel qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued
  - Kernels can call other kernel-side functions
- Address space qualifiers
  - __global, __local, __constant, __private
  - Pointer kernel arguments must be declared with an address space qualifier
- Work-item functions
  - uint get_work_dim() number of dimensions in use (1, 2, or 3)
  - size_t get_global_id(uint n) global work-item ID in dim “n”
  - size_t get_local_id(uint n) work-item ID in dim “n” inside work-group
  - size_t get_group_id(uint n) ID of work-group in dim “n”
  - size_t get_global_size(uint n) num of work-items in dim “n”
  - size_t get_local_size(uint n) num of work-items in work group in dim “n”
- Synchronization functions
  - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
  - Memory fences - provides ordering between memory operations

Figure 10: OpenCL C Language Restrictions

4.3 Optimisation of OpenCL

Branching, GPUs tend not to support speculative execution, which means that branch instructions have high latency.

Conditional execution

```
// Only evaluate expression
// if condition is met
if (a > b)
{
  acc += (a - b*c);
}
```

Selection and masking

```
// Always evaluate expression
// and mask result
temp = (a - b*c);
mask = (a > b ? 1.f : 0.f);
acc += (mask * temp);
```

Figure 11: Branching

Memory access patterns, GPU prefers to access memory continuouslyin successive work-items.

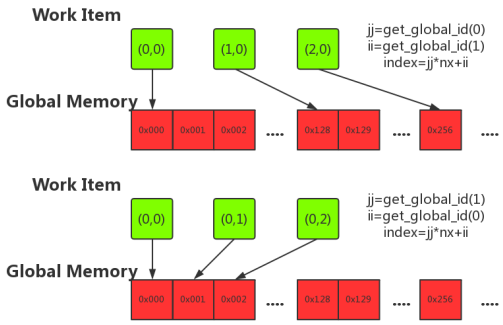


Figure 12: Memory Access Patterns