OpenCL

Jingke Li

Portland State University

What Is OpenCL?

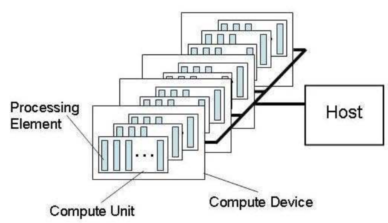
OpenCL = Open Computing Language

An open specification developed by the (open-membership) Khronos Group.

OpenCL provides a framework for writing and running parallel programs across a heterogeneous platform consisting of CPUs, GPUs, and other processors. It includes

- A C99-based language for writing *compute kernels* (functions that execute on OpenCL devices), and
- APIs that are used to manage computing tasks and data objects.

OpenCL Platform Model



(Figure credit: OpenCL Specification)

- A host connected to one or more *compute devices*.
- A compute device can be a CPU, a GPU, or some other processor.
- Computation is defined over an N-dim global domain (N=1, 2, or 3).
- All elements in the N-D domain execute in data-parallel fashion.

Work-Items and Work-Groups

- The smallest work unit is called a *work-item*, which corresponds to computation carried out at a single element of the N-D domain.
- To correspond to a typical GPU's organization, a set of work-items are grouped together to form a *work-group*.
- The size of work-groups are up to the programmer to specify. It must evenly divide the N-D domain size. (Different sizes may have different performance implications.)
- Work-items can be synchronized at work-group level, but not at the global level.

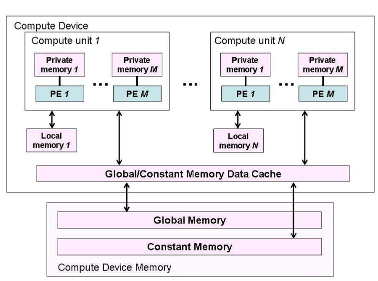
OpenCL Memory Model

Private Memory
— per work item

Local Memory
— shared within a work-group

Global Memory
— main memory for a compute device

Constant Memory
— special section of the global memory



(Figure credit: OpenCL Specification)

Memory Consistency

- OpenCL uses a relaxed consistency memory model; i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item, memory has load/store consistency.
- Local memory is consistent across work-items in a single work-group at a work-group barrier.
- Global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel.

OpenCL Programming Model

An OpenCL application consists of two programs:

- *Compute Program*
  - Contains a collection of *kernels* and other functions
  - Similar to a dynamic library
- *Host Program*
  - Handles I/O, memory management, and kernel scheduling

Even for simple appplications, both programs can be very complex when performance optimization is involved.

Typical Data Flow

- Host produces/captures data
- Host loads/builds kernels
- Host copy data to device DRAM
- Kernel loads data from DRAM into local memory
- Work-items execute, in parallel, on data in local memory
- Once work-items are done, move data back into device DRAM
- Move results back to host

OpenCL C

- Derived from ISO C99
  - no function pointers, recursion, variable-length arrays, and bit fields
- Additions to the language for parallelism
  - work-items and work-group, vector types, synchronization
- Address space qualifiers
  - `__global`, `__local`, `__constant`, and `__private`
- Built-in functions
  - for math, work-item/work-group, vector, and synchronization
- Optimized image access

Example: Array Operations

Sequential Code:

```
#define n 512
int main(int argc, char** argv)
{
  float a[n], b[n], sum[n], prod[n];
  int i;
  ...
  for (i=0; i<n; i++) {
    sum[i] = a[i] + b[i];
    prod[i] = a[i] * b[i];
  }
}
```


OpenCL Compute Program

A compute program consists of a set of kernel definitions. A kernel is a data-parallel function written from *individual element*'s view.

```
// array_ops.cl

__kernel void add(__global const float *a,
    __global const float *b, __global float *sum)
{
    int id = get_global_id(0);
    sum[id] = a[id] + b[id];
}

__kernel void mul(__global const float *a,
    __global const float *b, __global float *prod)
{
    int id = get_global_id(0);
    prod[id] = a[id] * b[id];
}
```


OpenCL Host Program

```
// test_array_ops.c

#define n 512
int main(int argc, char** argv)
{
  float a[n], b[n], sum[n], prod[n];
  int buf_size = sizeof(float) * n;

  // Run OpenCL:
  //   Set up context and command queue
  //   Allocate device memroy
  //   Load and build programs/kernels
  //   Execute kernels
  //   Cleanup
}
```

Set Up Context and Command Queue

Select a device and create a context and a command queue.

```
cl_context context =
    clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                            NULL, NULL, NULL);

cl_command_queue cmd_queue =
    clCreateCommandQueue(context, NULL, 0, NULL);
```


Allocate Device Memory

```
cl_int err;

cl_mem a_buf = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              buf_size, NULL, NULL);
cl_mem b_buf = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              buf_size, NULL, NULL);
cl_mem sum_buf = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                buf_size, NULL, NULL);
cl_mem prod_buf = clCreateBuffer(context, CL_MEM_READ_WRITE,
                                 buf_size, NULL, NULL);

// Copy data from host to device
err = clEnqueueWriteBuffer(cmd_queue, a_buf, CL_TRUE, 0,
                           buf_size, (void*)a, 0, NULL, NULL);
err = clEnqueueWriteBuffer(cmd_queue, b_buf, CL_TRUE, 0,
                           buf_size, (void*)b, 0, NULL, NULL);
```


Load and Build Programs/Kernels

```
cl_kernel kernel[2];

char *program_source = load_program_source("array_ops.cl");
cl_program program =
    clCreateProgramWithSource(context, 1,
        (const char**)&program_source, NULL, &err);

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

kernel[0] = clCreateKernel(program, "add", &err);
kernel[1] = clCreateKernel(program, "mul", &err);

err = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &a_buf);
err = clSetKernelArg(kernel[0], 1, sizeof(cl_mem), &b_buf);
err = clSetKernelArg(kernel[0], 2, sizeof(cl_mem), &sum_buf);
err = clSetKernelArg(kernel[1], 0, sizeof(cl_mem), &a_buf);
err = clSetKernelArg(kernel[1], 1, sizeof(cl_mem), &b_buf);
err = clSetKernelArg(kernel[1], 2, sizeof(cl_mem), &prod_buf);
```


Execute Kernels

Enqueue kernel computation; push them to the device for execution; then read back the results.

```
size_t global_work_size = n;
err = clEnqueueNDRangeKernel(cmd_queue, kernel[0],
          1, NULL, &global_work_size, NULL, 0, NULL, NULL);
err = clEnqueueNDRangeKernel(cmd_queue, kernel[1],
          1, NULL, &global_work_size, NULL, 0, NULL, NULL);

clFinish(cmd_queue); // synchronization

err = clEnqueueReadBuffer(cmd_queue, sum_buf, CL_TRUE, 0,
                          buffer_size, sum, 0, NULL, NULL);
err = clEnqueueReadBuffer(cmd_queue, prod_buf, CL_TRUE, 0,
                          buffer_size, prod, 0, NULL, NULL);
```


Cleanup

```
clReleaseKernel(kernel[0]);
clReleaseKernel(kernel[1]);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleaseContext(context);
```


Example: Matrix Transpose

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

⇒

0,0	1,0	2,0	3,0
0,1	1,1	2,1	3,1
0,2	1,2	2,2	3,2
0,3	1,3	2,3	3,3

From memory's view:

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,3

⇓

0,0	1,0	2,0	3,0	0,1	1,1	2,1	3,1	0,2	1,2	2,2	3,2	0,3	1,3	2,3	3,3

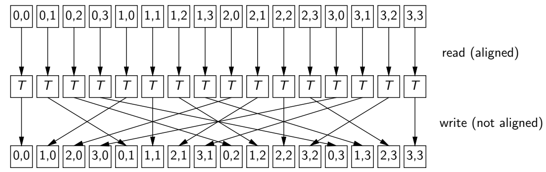
Naive Kernel

Work directly with global memory:

```
__kernel void naive_transpose(
    __global float *idata, __global float* odata, int nx, int ny)
{
    unsigned int id_x, id_y, idx_in, idx_out;

    id_x = get_global_id(0);
    id_y = get_global_id(1);
    idx_in  = id_y * nx + id_x;
    idx_out = id_x * ny + id_y;
    odata[idx_out] = idata[idx_in];
}
```


Problem with Naive Kernel

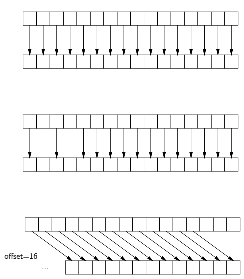
Global memory access pattern:



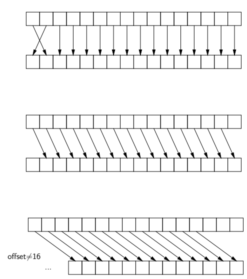
If global memory accesses are aligned, they can be coalesced (into 64-byte chuncks on current GPUs). Peformance difference: 16x.

Global Memory Access Patterns

Aligned:

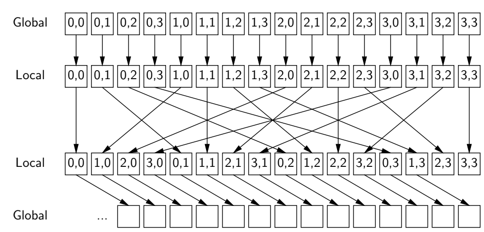


Not-aligned:



Better Kernel Using Local Memory

- Load data from global memory to local memory in aligned form.
- Perform block transpose in local memory.
- Write data back to global memory in aligned form.



Better Kernel (cont.)

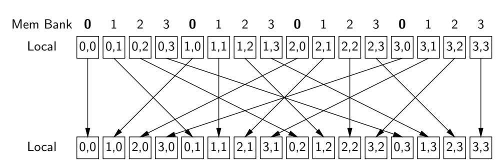
```
__kernel void better_transpose(
    __global float *idata, __global flo at* odata,
    __local float *block, int nx, int ny, blockSize)
{
    unsigned int gid_x, gid_y, g_src, g_dst, grp_x, grp_y;
    unsigned int lid_x, lid_y, l_src, l_dst;

    gid_x = get_global_id(0);
    gid_y = get_global_id(1);
    g_src = gid_y * nx + gid_x;
    lid_x = get_local_id(0);
    lid_y = get_local_id(1);
    l_src = lid_x * blockSize + lid_y;
    block[l_src] = idata[g_src];

    barrier(CLK_LOCAL_MEM_FENCE);
    gid_x = get_group_id(1) * blockSize + lid_y;
    gid_y = get_group_id(0) * blockSize + lid_x;
    g_dst = gid_y * ny + gid_x;
    l_dst = lid_y * blockSize + lid_x;
    odata[g_dst] = block[l_dst];
}
```


Still a Problem ...

Local memory access conflicts:

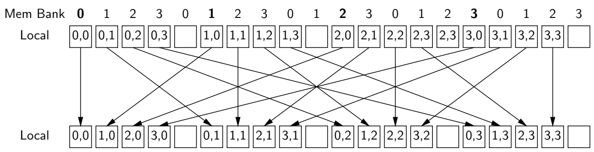
- Local memory is organized into memory banks (on current GPUs, there are typically 16 banks)
- Simultaneous multiple accesses from different threads to the same memory bank will be serialized.

(For illustration, assume 4 memory banks.)



Fixing the Problem

To create a production-quality matrix-transpose kernel, the data arrays have to be *padded*:



This will result in an even-more complex kernel code.

Matrix Transpose Kernel (from Apple Inc.)

```
#define PADDING         (32)
#define GROUP_DIMX      (32)
#define LOG_GROUP_DIMX  (5)
#define GROUP_DIMY      (2)
#define WIDTH           (256)
#define HEIGHT          (4096)

__kernel void transpose(
    __global float *output,
    __global float *input,
    __local float *tile)
{
  int block_x = get_group_id(0);
  int block_y = get_group_id(1);
  int local_x = get_local_id(0) & (GROUP_DIMX - 1);
  int local_y = get_local_id(0) >> LOG_GROUP_DIMX;

  int local_input  = mad24(local_y, GROUP_DIMX + 1, local_x);
  int local_output = mad24(local_x, GROUP_DIMX + 1, local_y);
  int in_x = mad24(block_x, GROUP_DIMX, local_x);
  int in_y = mad24(block_y, GROUP_DIMX, local_y);
  int input_index = mad24(in_y, WIDTH, in_x);

  int out_x = mad24(block_y, GROUP_DIMX, local_x);
  int out_y = mad24(block_x, GROUP_DIMX, local_y);
  int output_index = mad24(out_y, HEIGHT + PADDING, out_x);

  int global_input_stride  = WIDTH * GROUP_DIMY;
  int global_output_stride = (HEIGHT + PADDING) * GROUP_DIMY;
  int local_input_stride  = GROUP_DIMY * (GROUP_DIMX + 1);
  int local_output_stride = GROUP_DIMY;
```


Matrix Transpose Kernel (cont.)

```
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  tile[local_input] = input[input_index];       local_input += local_input_stride; input_index += global
  ... // total 16 groups of statements

  barrier(CLK_LOCAL_MEM_FENCE);

  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  output[output_index] = tile[local_output]; local_output += local_output_stride; output_index += global_
  ... // total 16 groups of statements
}
```

Total # lines in the program: 148 (many contain multiple statements!)

Performance Tuning

Overhead come from many places:

- Compiling programs
- Moving data to/from devices
- Starting kernels
- Synchronization
- Divergent execution at work-item level
- Non-coalesced global memory accesses
- Local memory access conflicts

To get the best performance, one needs to

- know the details of the target device
- refine kernels to optimize memory operation performance
- take tuning runs