ECE 574 – Cluster Computing
Lecture 19

Vince Weaver

http://web.eece.maine.edu/~vweaver
vincent.weaver@maine.edu

4 April 2019
Announcements

• HW#8 was due
  as always many of the problems were C related, this time pointers and casting

• HW#9 will be assigned
OpenCL

• CUDA is only for NVIDIA GPUs

• What if you have Intel or AMD (ATI) chip? Or ARM MALI? or Raspberry Pi Vcore IV?

• OpenCL is sort of like CUDA, but cross-platform

• Not only for GPUs, but can target regular CPU, DSP, FPGAs, etc

• Vendor provides a driver
• Khronos (the OpenGL + Vulkan people?) also run OpenCL

• Windows, OSX, Linux
OpenCL History

- Started by Apple
- Donated to Khronos
- Apple has abandoned it
OpenCL program Flow

- Allocate host buffer
- Get platform/device
- Set up platform
- Choose device
- Create context
- Create command queue
- Create memory buffer on device
- Copy buffer to device
- Create a program kernel
• Build kernel
• Set arguments
• Execute
• Read back results
• clean up and wait to finish
• Release
Platforms

- Query number of platforms
- clGetPlatformIDs
- Then malloc space, and use same function to get info
- Can iterate and get NAME, VENDOR, VERSION
Devices

• Now when got platform, similarly clGetDeviceIDs

• Why multiple? Intel or AMD CPU might have both CPU and GPU
Memory Hierarchy

- global – shared by all, but high latency
- constant – read only by all but cpu, smaller, a bit faster
- local – shared by a group of cores on device
- register – per element
Language

- Based on C
- Pointers annotated with memory level
- Some things not allowed: recursion, function pointers
- Regular data types, some others like vectors
- With OpenCL 2.x more similar to C++
- Plan is to merge it with Vulkan
Iterations in the kernel

- A lot like CUDA, where split into 1D, 2D, or 3D grid.
- `get_global_id()`;
- `get_local_id()`;
- `get_num_groups()`;
- `get_group_size()`;
- `get_group_id()`
Calling the kernel

clEnqueueNDRangeKernel(command_Queue, kernel, work_dim, global_work_offset, global_work_size, local_work_size, event_wait_list, event)
Contexts
Command Queue

- FIFO or out of order (always issued in order)
OpenCL ICD (installable client driver)
Memory Config

- Memory flags. r/w, ro, wo
- Whether host pointer or not
- Using host pointers might not be fast if
- depending on arch
- sub-buffer?
Copying to/from Device

- copying mem
- host to device
- device to device
Creating Kernels

- The driver builds them
- From source, you pass in as a string
- Can get binary-only kernels (why?)
  - Proprietary?
  - also, not have to build each time
- `clCreateProgramWithSource()`
- `clCreateProgramWithBinary()`
- Can get error log of build when something went wrong
SPIR – standard portable Intermediate Representation
Setting kernel arguments

- must do this before running
Executing Kernel
Synchronization

• when needed?
• single device, out of order queue
• multiple devices?
• coarse grained
  ◦ clFlush/clFinish
• fine grained
  ◦ event based
• memory fences?
• CL event, for communicating
OpenCL C Programming

- Own built-in data types: basic app vector app_vector char cl_char charn cl_charn etc
  why? portable. sadly sizes not same on windows/linux
- n element 2, 3, 4, 8, 16 sizes
- “half” type for 16-bit fp
- address space qualifiers
  - __global
  - __local
  - __constant
- `__private`
OpenCL – compiling

gcc -I include -L /lib -lOpenCL saxpyc -o saxxpy
void saxpy ( int n, float a, float *x, float *y, float *out ) {
    for (i = 0; i < n; i++) {
        out[i] = a * x[i] + y[i];
    }
}

#include <CL/cl.h>

const char * saxpy_kernel =
"__kernel

void saxpy_kernel(float a,
"__global float *A,
"__global float *B,
"__global float *out) {
"int index=get_global_id(0);
"}\
"

#define N 1024

int main(int argc, char **argv) {
    int i;
    float alpha=5.0;
    float *A=(float *)malloc(sizeof(float)*N);
    float *B=(float *)malloc(sizeof(float)*N);
    float *C=(float *)malloc(sizeof(float)*N);

    for(i=0;i<N;i++) { A[i]=i; B[i]=10*i; C[i]=0;}

    // get platform
    cl_platform_id *platforms=NULL;
    cl_uint num_platforms;
    cl_int clStatus=clGetPlatformIDs(0,NU,&num_platforms);
    platforms=(cl_platform_id *)malloc(sizeof(cl_platform_id)*num_platforms);
    clStatus=clGetPlatformIDs(num_platforms, platforms,NULL);

    cl_device_id *device_list=NULL;
    cl_uint num_devices;
    clStatus = clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,0,NULL,&num_devices);
    device_list=(cl_device_id *)malloc(sizeof(cl_device_ids)*num_devices);
clStatus = clGetDeviceIDs(plaforms[0], CL_DEVICE_TYPE_GPU, num_devices, device_list, NULL);

// create context
cl_context context;
context = clCreateContext(NULL, num_devices, device_list, NULL, NULL, &clStatus);

// Create command queue
cl_command_queue command_queue = clCreateCommandQueue(context, device_list, 0, &clStatus);

// Create memory buffer on device
cl_mem A_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY, VECTOR_SIZE * sizeof(float), NULL, &clStatus);
cl_mem B_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY, VECTOR_SIZE * sizeof(float), NULL, &clStatus);
cl_mem C_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY, VECTOR_SIZE * sizeof(float), NULL, &clStatus);

// Copy buffer a and b to device
clStatus = clEnqueueWriteBuffer(command_queue, A_clmem, CL_TRUE, 0, VECTOR_SIZE * sizeof(float), A, 0, NULL, NULL);
clStatus = clEnqueueWriteBuffer(command_queue, B_clmem, CL_TRUE, 0, VECTOR_SIZE * sizeof(float), B, 0, NULL, NULL);

// Create a program
cl_program program = clCreateProgramWithSource(context, 1, (const char **)&saxpy_kernel, NULL, &clStatus);

// Build program
clStatus = clBuildProgram(program, 1, device_list, NULL, NULL, NULL);

// Create OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "saxmpu_kernel", &clStatus);

// Set Arguments
clStatus = clSetKernelArg(kernel, 0, sizeof(float), coid *)&alpha);
clStatus = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&A_clmem);
clStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&B_clmem);
CclStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&B_clmem);

// execute kernel

size_t global_size = VECTOR_SIZE;
size_t local_size = 64;
clStatus = clEnqueueNDRangeKernel(command_Queue, ekernel, 1, NULL, &global_size, &local_size, 0, NULL, NULL);

// Read out results
clStatus = clEnqueueReadBuffer(command_queue, C_clemem, CL_TRUE, 0, VECTOR_SIZE * sizeof(float), C, 0, NULL, NULL);

// cleanp
clStatus = clFlush(command_queue);
clStatus = clFinish(command_queue);

// display results
for (i = 0; i < VECTOR_SIZE; i++)
    printf("%d %d %d alhpha" A[i], B[i], Cpi[]);

// release
clstatus = ClReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseMemoObject(A_clmem);
clstatus=clReleaseCommandQueue(command_Queue);
clStatus=clReleaseContet(context);
free(A); BCplatformsdevice_list