ECE 574 – Cluster Computing
Lecture 20

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

4 April 2023

Announcements

- Don't forget HW#8

HW#8 Notes

- Remember that CUDA is a little like MPI, in that the GPU is a separate machine without a shared memory space
- You have to make sure you are passing by reference, you can't pass a CPU pointer as an argument and expect it to work
- It is hard to debug. If getting weird results, try backing things out step at a time until it does what you expect and then adding things back on

Non-CUDA Acceleration Libraries

OpenACC

- Sort of like OpenMP but can offload to GPU as well as CPUs
- Cray, CAPS, Nvidia and PGI
- Designed for use in heterogeneous CPU/GPU systems
- Like OpenMP, annotate existing code

OpenACC – Using it

- Need a compiler that supports it
- GCC only got support for OpenACC 2.5 in version 9.1
- If you want to run on gpu you need nvc (NOTE: not the same as nvcc) which is nvidia's version of the PGI compiler
- Note, you don't need to allocate memory on device and copy back/forth, it does it for you
- include openacc.h
- Pragmas, like with OpenMP

  - to define/copy data: `#pragma acc data`
  - to tell the compiler to parallelize a region. It might be conservative, so you might have to give it extra info to get better performance `#pragma acc kernels`
  - to parallelize a loop (note, you need to make sure it is safe to do this): `#pragma acc parallel loop`
- Various runtime functions as well, e.g. `acc_get_num_devices()`
- Compile code with `-fopenacc`
- It's hard to tell even when code compiles/runs if it's actually being accelerated

Other Low-Level Accelerator Libraries

- For graphics, OpenGL and DirectX/3D too abstract, not match all hardware
- Issues like efficient use of DMA, command buffers, etc.
- Try to get CPU and GPU working better together
- Defunct OpenGL-style Graphics Libraries:
  - Glide (3dfx)
  - Mantle (AMD)
- Other low-level GPU libraries: GNM (playstation 4), NVN (Nvidia/Switch)

Apple Metal

- Metal – from Apple, their replacement for OpenCL. C++ like, sort of a mix of OpenCL and OpenGL

Others

- WebGPU – GL/GPGPU Javascript (currently under development)
- WebCL – OpenCL Javascript bindings
- OpenVG – 2d vector graphics accel
- Lots more on wikipedia (?)

Vulkan

- More modern OpenGL
- Supposedly OpenCL merging into Vulcan?
- based on AMD Mantle
- Is a bit beyond this class

OpenCL – Open Computing Language

- The main competitor to CUDA?
- 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, 2008
- Donated to Khronos
- Apple has abandoned it
- AMD chose it instead of Metal
- OpenCL 1.0 (2009)
- OpenCL 1.1 (2010)
- OpenCL 1.2 (2011)
- OpenCL 2.0 (2013)
    - Shared virtual memory

- OpenCL 2.1 (2015)
  - Can use C++ in kernels
- OpenCL 2.2 (2017)
  - Support for SPIR-V intermediate language
- OpenCL 3.0 (2020)
  - OpenCL 1.2 is baseline
  - All 2.x and 3.x features optional?
  - Changed up the C++ and code generation, based on LLVM
- Grumblings of somehow merging functionality with Vulkan?

Installing OpenCL (Linux)

- You install opencl
- You also need to install an ICD (installable client driver) for the device you want to run on
- You can have multiple ICDs installed
- NVIDIA is actually easiest, especially if you already have CUDA going
- AMD as of 2022 the open-source drivers don't support OpenCL

  You can install OpenCL from the proprietary drivers but that might not work well

- Intel GPU has project could Beignet
- There are also CPU/software, emulated, and other ICDs

OpenCL program Flow

Similar to CUDA but *much* more verbose

- 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

Getting things Going

- Much more of a pain than CUDA, lots of manual and boilerplate code
- I'll provide it for you

First – Platforms

```
cl_int clGetPlatformIDs(cl_uint num_entries,
                 cl_platform_id *platforms,
                 cl_uint *num_platforms);
```

- Query number of platforms
- You can call with num_entries 0, platforms NULL to get number of platforms
- Then `malloc()` space to get all the info
- You can also hard-code a number to read, but that's not as flexible

Iterating platform info

```
for(i=0;i<num_platforms;i++) {
     err   = clGetPlatformInfo(platform[i], CL_PLATFORM_NAME,
                 sizeof(platform_name[i]), platform_name[i],
                 &returned_size);
     if (err != CL_SUCCESS) {
                 printf("Error: Failed to get platform info! %s\n",
                        cl_getErrorString(err));
     return EXIT_FAILURE;
}
```

- Can iterate and get NAME, VENDOR, VERSION
- Need to allocate space for strings

Error printing aside

- OpenCL doesn't have equivalent of `strerror()`
- You just get a number on error
- You can implement your own (I provide one)

Initializing Devices

```
cl_int clGetDeviceIDs(
    cl_platform_id platform,
    cl_device_type device_type,
    cl_uint num_entries,
    cl_device_id* devices,
    cl_uint* num_devices);
```

- Now when you have the platform, you can get the devices for that platform
- Why multiple? Can you have multiple GPUs on same platform?
  Can you have a CPU that also has integrated GPU?
- Device type: CL_DEVICE_TYPE_ALL,

CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_CPU, etc

Iterating Devices

```
cl_int clGetDeviceInfo(
    cl_device_info param_name,
    size_t param_value_size,
    void *param_value,
    size_t *param_value_size_ret)
```

- You can also iterate devices to get info too

Initializing the Context

```
cl_context clCreateContext(const cl_context_properties *properties,
                   cl_uint num_devices,
                   const cl_device_id *devices,
                   void ( CL_CALLBACK *pfn_notify) const char *errinfo,
                   const void *private_info, size_t cb,
                   void *user_data,
                   cl_int *errcode_ret)
```

- A context manages the host/device interaction
- We need one for each OpenCL kernel we call
- Callback function can be used to return errors from the kernel, can set to 0/NULL if don't care

Creating the Command Queue

```
cl_command_queue clCreateCommandQueueWithProperties(
        cl_context context,
        cl_device_id device,
        const cl_queue_properties *properties,
        cl_int *errcode_ret);
```

- Creates command queue

Note on Kernel

- 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

Loading Kernel – From Source

- Just-in-time compilation
- How can you do that? Just include the kernel as plain text and it gets compiled right when you run the program
- Upside: your executable can be moved to other machines with different backends and it will just work
- Downsides: needs to compile the code every time you run it

Loading Kernel – Binary

- Can get binary-only kernels (why?)
  - Proprietary?
  - also, not have to build each time
- `clCreateProgramWithBinary()`

Including the Kernel

- Just have it in a string in your file
- Have it on disk but do some `#include` magic
- Have it in a file on disk and load it into a string
- Intermediate representation?

Notes on kernel (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

Example

```
const char *saxpy_kernel = "\n"
        "__kernel\n"
        "void saxpy(\n"
        "    const unsigned int n,\n"
        "    const float a,\n"
        "    __global float *x,\n"
        "    __global float *y) {\n"
        "\n"
        "    int i = get_global_id(0);\n"
        "\n"
        "    if (i < n) {\n"
        "       y[i] = a * x[i] + y[i];\n"
        "    }\n"
        "}\n"
        "\n";
```

Loading the kernel from source code

```
cl_program clCreateProgramWithSource(cl_context context,
                                     cl_uint count,
                                     const char **strings,
                                     const size_t *lengths,
                                     cl_int *errcode_ret)
```

Building the Kernel

```
cl_int clBuildProgram(cl_program program,
                      cl_uint num_devices,
                      const cl_device_id *device_list,
                      const char *options,
                      void (CL_CALLBACK *pfn_notify)
                           ( cl_program program, void *user_data),
                      void *user_data)
```

- Essentially just launch a compiler on the kernel source code
- Can get build info (the build log)
- Can pass command line arguments
- Can release kernel when done (TODO)

Create the Kernel

```
cl_kernel clCreateKernel ( cl_program  program,
    const char *kernel_name,
    cl_int *errcode_ret)
```

- Note the function name is the same as specified in kernel

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

Allocating Memory

```
cl_mem clCreateBuffer ( cl_context context,
   cl_mem_flags flags,
   size_t size,
   void *host_ptr,
   cl_int *errcode_ret)
```

- Parameters like `CL_MEM_READ_WRITE`, `CL_MEM_READ_ONLY`, etc.

Copying Memory Host to Device

```
cl_int clEnqueueWriteBuffer(cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_write,
                            size_t offset,
                            size_t size,
                            const void *ptr,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)
/* Example */
err   = clEnqueueWriteBuffer(commands, dev_x, CL_TRUE, 0,
                        sizeof(float) * N, x, 0, NULL, NULL);
```

-
- OpenCL 2.0 allows sharing virtual address space so you might not have to copy?

Setting up arguments

```
cl_int clSetKernelArg(
    cl_kernel kernel,
    cl_uint arg_index,
    size_t arg_size,
    const void* arg_value);

err |= clSetKernelArg(kernel_saxpy,
    0, sizeof(unsigned int), &N);
```

- Set arguments to pass to kernel

Getting size of workgroup kernel

```
cl_int clGetKernelWorkGroupInfo(cl_kernel kernel,
                                cl_device_id device,
                                cl_kernel_work_group_info param_name,
                                size_t param_value_size,
                                void *param_value,
                                size_t *param_value_size_ret)
```

- Determine how wide we can be, sort of like the max thread count in CUDA
- Can set up three-dimensional thread type things like CUDA but easier not to if we fit

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()

Launching the kernel

```
cl_int clEnqueueNDRangeKernel (
   cl_command_queue command_queue,
   cl_kernel kernel,
   cl_uint work_dim,
   const size_t *global_work_offset,
   const size_t *global_work_size,
   const size_t *local_work_size,
   cl_uint num_events_in_wait_list,
   const cl_event *event_wait_list,
   cl_event *event)
```

- Launch the kernel

Command Queue

- FIFO or out of order (always issued in order)

Querying 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

Freeing stuff at end

- Good idea

OpenCL – compiling

```
gcc -I include -L /lib -lOpenCL saxpyc -o saxxpy
```

Demo, sample code

- Try out `clinfo` program
- Run saxpy with 0, 1, and 2 devices
- Note slowdown as it JITs

SPIR – standard portable Intermediate Representation