Administrivia

- HW2 out, due Nov. 2.
- HW3 will be out by Nov. 2.
- OpenCL test repo coming (Odyssey & your machines)
- Final Projects: start forming groups, brainstorming ideas, posting on Piazza, proposing ideas.
Final Projects

• Read over cs205.org Projects page, look at projects from previous years.

• Proposals will be due Nov. 6, we’ll provide reviews back by the 9th.
  • You can propose an idea before then, even if not completely ready.
  • Proposing early will make it easier for everyone.
Introduction to OpenCL

https://github.com/fasrc/cs205_Intro_to_Odyssey
test.py
OpenCL

- Device-independent, but GPU-like computational framework.
- Portable: can run on non-GPU hardware.
- On Mac/Linux: pip install pyopencl.
- On Windows: Google, ask on Piazza.
- Try running the test program.
Overview of an OpenCL program

Setup - find device
Allocate buffers
Transfer data CPU→GPU
Launch work on GPU
Transfer data GPU→CPU
Shutdown
CPU <> GPU transfer
Memory

• CPU allocates space in GPU global and transfers data to it.

• That transfer is (usually) slow compared to on-GPU memory bandwidth.

```python
host_x = np.random.uniform(0, 1, N).astype(np.float32)
x = cl.Buffer(context, cl.mem_flags.READ_ONLY, N * 4)
cl.enqueue_copy(queue, x, host_x)
```
GPU programs = "Kernels"

• Once data is ready on GPU, CPU launches processing via a Kernel.

• Written in something much like C. Usually not very complicated code.

kernel_src = """ ... """

program = cl.Program(context, kernel_src).build()
```c
_kernel void
add_vectors(__global float* z,
    __global const float *x,
    __global const float *y,
    const unsigned int n) {

    unsigned int thread_id = get_global_id(0);
    // Make sure we stay in-bounds
    if (thread_id < n)
        z[thread_id] = x[thread_id] + y[thread_id];
}
```

kernel_src = """
__kernel void
add_vectors(__global float* z,
            __global const float *x,
            __global const float *y,
            const unsigned int n) {

    unsigned int thread_id = get_global_id(0);
    // Make sure we stay in-bounds
    if (thread_id < n)
        z[thread_id] = x[thread_id] + y[thread_id];
}
"""
__kernel void add_vectors(__global float* z, __global const float *x, __global const float *y, const unsigned int n) {

    unsigned int thread_id = get_global_id(0);
    // Make sure we stay in-bounds
    if (thread_id < n)
        z[thread_id] = x[thread_id] + y[thread_id];
}


__kernel void
add_vectors(__global float* z,
            __global const float *x,
            __global const float *y,
            const unsigned int n) {

    unsigned int thread_id = get_global_id(0);
    // Make sure we stay in-bounds
    if (thread_id < n)
        z[thread_id] = x[thread_id] + y[thread_id];
}
CPU controls high-level schedule

- Memory transfers, kernel launches, etc.
- These are all put in a "queue".
- Operations in queue happen in order.
- There might be multiple queues on one GPU = parallel threads of GPU work.

```python
program = cl.Program(context, kernel_src).build()
program.add_vectors(queue, global_size, local_size, z, x, y, np.uint32(N))
```
kernel_src = ""
__kernel void
add_vectors(__global float* z,
   __global const float *x,
   __global const float *y,
   const unsigned int n) {

    unsigned int thread_id = get_global_id(0);
    // Make sure we stay in-bounds
    if (thread_id < n)
        z[thread_id] = x[thread_id] + y[thread_id];
}
"""
Many, many Threads

- Often, there will be one "thread" per input or output element, or pixel in an image.

- The number of threads is the "global_size" argument to kernel.

```python
host_x = np.random.uniform(0, 1, N).astype(np.float32)
global_size = host_x.shape

program.add_vectors(queue, global_size, local_size, z, x, y, np.uint32(N))
```


```
kernel_src = ""
__kernel void
add_vectors(__global float* z, 
    __global const float *x, 
    __global const float *y, 
    const unsigned int n) {

    unsigned int thread_id = get_global_id(0);
    // Make sure we stay in-bounds
    if (thread_id < n)
        z[thread_id] = x[thread_id] + y[thread_id];
}
"""
```

```
program.add_vectors(queue, global_size, 
    z, x, y, np.uint32(N))
```
__kernel void square(__global float* input, __global float* output)
{
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}
__kernel void square(__global float* input, 
    __global float* output)
{
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}

get_global_id(0) = 7
Work Groups

• Global threads divided up into "work groups."

• Size of work group controlled by `local_size` argument at kernel launch.

• Typical sizes are 64-128, multiple of 8.

• Work groups can share local memory (if requested).

• Synchronization between threads **only** at group level.
global_size = (16,)
local_size = (8,)

In Kernel:

global_id(0) = 3
local_id(0) = 3

get_global_id(0) = 3
get_local_id(0) = 3
get_group_id(0) = 0
get_group_size(0) = 8
get_num_groups(0) = 2

get_global_id(0) = 12
get_local_id(0) = 4
get_group_id(0) = 1
get_group_size(0) = 8
get_num_groups(0) = 2
Local Memory

- Local memory is a sort of non-automatic cache.
- Can be allocated per group at launch.
- Becomes part of the thread context.
- Too large, and bad things happen.
Many small contexts

good latency hiding
Few large contexts
poor latency hiding

Fetch/Decode

ALU 1
ALU 2
ALU 3
ALU 4
ALU 5
ALU 6
ALU 7
ALU 8

1 core (4 threads)
Next time

• Getting info about GPUs
• Multidimensional Kernels
• Memory reads & "coalescing".
• Synchronization.
• Using Local memory.

• Try to get OpenCL installed. Test it on your machine.
• If you have a GPU, you may not need Odyssey.
Synchronization

mem_fence(CLK_LOCAL_MEM_FENCE &| CLK_GLOBAL_MEM_FENCE)
barrier(CLK_LOCAL_MEM_FENCE &| CLK_GLOBAL_MEM_FENCE)

mem_fence - ensures all threads in a work group see consistent memory.

barrier - all threads reach the barrier, then call mem_fence.