Slide 1

Administrivia

- Reminder: Homework 2 due Monday.

Slide 2

GPGPU

- Recall from overview/introduction: SIMD (Single Instruction, Multiple Data) model popular early on, fell of favor, and now making a comeback as "GPGPU" (General-Purpose computing on Graphics Processing Units).
- Typically SIMD a good fit for GPU hardware, *but* worth noting GPU may have its own memory, not shared with "host" CPU, which makes programming more complicated and has implications for performance.

Slide 3

OpenCL

- Early work on shared-memory and message-passing programming resulted in many competing programming environments. But eventually, OpenMP and MPI emerged as standards.
- Similarly, initially many different programming environments for GPGPU. Will OpenCL emerge as a standard? Currently unclear; maybe!
- For all of these, idea was to come up with a single standard, then allow many implementations. For MPI, standard defines concepts and library. For OpenMP, standard defines concepts, library, and compiler directives. For OpenCL, standard defines . . . more.
- First release 2008; evolving fairly rapidly. Meant to address not just GPGPU but more-general problem of "heterogeneous computing" (computing using mix of computational resources).

Slide 4

What's an OpenCL Program Like?

- First thing to note: OpenCL programs typically involve some computation on a "host" computer (main processor(s)) and some computation on the "compute device" (often the GPU — but doesn't have to be! everything designed to allow it to work with many kinds of compute devices).
- Source code for host computer in C/C++, with calls to OpenCL functions, etc. (OpenCL also defines some "opaque" data types, e.g.)
- Source code for compute device written in dialect of C. Compiled at runtime for whatever device is actually used. (Note that "device" here means something not exactly like what it has come to mean in popular usage!)

Slide 5

OpenCL — Compiling and Executing Programs

- C programs using OpenCL compiled more or less like other C programs, into object code that's linked to produce an executable — *for the host computer*.

  "More or less"? Some implementations just use a regular C compiler, possibly with flags telling it where to find library files. Others include a compiler, but one that calls a regular C compiler to do a lot of its work.

- What happens to the source code for the compute device? Often (usually? normally?) it's compiled at runtime by OpenCL library functions on the host. (And yes, *they* have to compile a variant of C.)

Slide 6

A First Example

- Worth noting that you can't really write a "hello world" program, since compute device doesn't necessarily have access to standard output!
- As a first example: Sample program `semi-hello.c` that just does a lot of setup, calls some inquiry functions, prints results (on host).
- (Review next slide, then compile and execute.)

Slide 7

OpenCL and Department Machines

- All classroom/lab machines (but not servers) OpenCL-capable, and we've installed appropriate support. We also installed something that lets programs use the CPU(s) as a "compute device".
- On machines with AMD graphics cards (all but 270A, 270L), support is part of proprietary driver. Compile with regular compiler and a few extra flags (see `Makefile.amd`). Caveats:

  OpenCL programs run remotely can't access GPU.

  Apparently a bug somewhere; programs work on the Xena machines but not on other AMD-card machines.
- On machines with NVIDIA cards (270A, 270L), support part of company's own toolkit for GPGPU, called CUDA. Compile with CUDA compiler (see `Makefile.cuda`).

Slide 8

Working With "Compute Devices"

- Goal in designing standard was to come up with something that would work on many platforms, with many kinds of compute devices, and provide access to details for programmers who care about them.
- Result is that what "host" programs need to do to use a compute device is . . . Complicated. Lengthy. Tedious.

  Appendix has self-contained code. Good for examples but tedious for program development (a lot of copy-and-paste). I instead wrote some library code and use that.

Slide 9

Working With "Compute Devices" — Concepts

- Computation on an OpenCL compute device is done using a command queue and "kernels".
- In terms of the SIMD model, a kernel defines a single instruction stream, which will be executed effectively-in-parallel on multiple data items.
- Kernels typically compiled at runtime from source code. Source code can be big text string in host program (as in examples in appendix) or read in from a file (as in my sample programs).

Slide 10

Working With "Compute Devices", Continued

- "Command queue" for device holds commands for it to execute — data transfers between host and device memories, kernel executions, etc.
- "Index space" defines a range on which to execute a kernel.
- "Work item" is one execution of a kernel.
- "Work groups" are used to group work items: A compute device can only operate on some finite number of data items truly in parallel; to operate on a larger index space, must break it up into "work groups", execute them one at a time. (But effect is the same as if they all executed in parallel.)

Slide 11

Working With "Compute Devices", Continued

- Memory accessible to kernels comes in different varieties:
  - Global memory — accessible to all work items, allocated and written/read by host.
  - Constant memory — similar to global memory, but read-only to work items.
  - Local memory — accessible to all work items, can be allocated by host or (statically!) in kernel.
  - Private memory — accessible to a single work item. Also static allocation only.
- All distinct from host memory.

Slide 12

OpenCL — Vector Addition Example

- Example: Toy program to add two vectors, producing a third vector, with compute device doing actual addition. So . . .
- Sequential code for this problem is simple — a `for` loop over all elements in the vectors' index range.
- Multithreaded version in OpenMP almost trivial. Message-passing version in MPI more work but not too bad. GPGPU version in OpenCL?

Slide 13

OpenCL — Vector Addition Example, Host Program

- Create the three vectors (as arrays).
- Fill in input values and copy to compute device's memory.
- Tell device to do addition.
- Copy results back from device memory.
- Check/print results.

Slide 14

OpenCL — Vector Addition Example, Kernel

- Should be done once for each element of the vectors' index range (so, these are the "work items").
- Computation is basically the body of the sequential-code loop.

  (This may be too "fine-grained" to perform well, but for a first example keep it simple?)

Slide 15

OpenCL — Program Start-Up

- MPI programs are supposed to start with call to `MPI_Init`, to set up the MPI environment. OpenCL programs also start with some setup, but *much* more involved.
- First step is to find suitable device. Devices are typically grouped by "platforms" (e.g., CPU, NVIDIA GPU). OpenCL functions let you find available platforms and then within each one look for devices of whatever type you're interested in.

  Result is a "device ID", to be used to access device.

  (My library function `getDevice()`.)

Slide 16

OpenCL — Program Start-Up, Continued

- Once you have a device ID, can start setting things up to use it (my library function `initialize()`):
- Create a "context" — information about device, associated memory, etc.
- Create a command queue for the device.
- Create a "program object" — basically a dynamically-generated library of kernels, built by compiling at runtime from source. Source can be defined as long string in program or read from file.

Slide 17

OpenCL — Data Transfer

- A complication in OpenCL programming: Must transfer data from host to compute device, and vice versa. Details of doing this are, well, detailed.
- First step is to create “buffers”.
- To actually do the copying, put on device’s command queue command to write to or read from a buffer, specifying destination or source location in host.

Slide 18

OpenCL — Defining Kernels

- Source for kernel(s) compiled at runtime, as discussed.
- Written in dialect of C.
- Can specify different kinds of parameters. Some library functions available.

Slide 19

OpenCL — Executing Kernels

- To execute a kernel — well, again, it's messy!
- First step is to build the kernel from the program object. (Done in my `initialize()`.)
- Next step is to set up arguments.
- And then to execute kernel, put command on command queue to do so, specifying kernel, index range, and size of "workgroup". (Inquiry function lets you find out maximum for that.)
- Finally, wait for command to finish.

Slide 20

OpenCL — Tidying Up

- MPI programs supposed to end with call to `MPI_Finalize`. Analogous cleanup should happen at end of OpenCL program.

  (My library function `finalize()` does some of this.)

Slide 21

OpenCL — Review/Recap

- OpenCL was intended to be very portable but also not to hide too much from the programmer.
- As a result, programmers must deal with a lot of low-level details.
- Good news is that a lot of those details are the same from program to program. So I wrote some library functions (`utility.h` under "Sample programs"). Okay for you to use them if you promise to read and (try to) understand!

Slide 22

Numerical Integration in OpenCL

- What does our familiar example look like in OpenCL?
- A first thing to note is that OpenCL provides only a single-precision floating-point type(!).
- It doesn't seem quite right, then, to compare results with code that computes using double precision. So I first wrote a version of the sequential code that uses `float` rather than `double`, and .... And that took me down a different rabbit hole (more next time).

Slide 23

Numerical Integration in OpenCL, Continued

- Basic strategy for parallelizing — split iterations of main processing loop among UEs and the combine results — same. (UEs here are work items.)
- *Could* make each loop iteration a work item (as in vector addition example), but might not work out too well — adding each tiny increment to a larger result seems like it would be a bottleneck. So adopt same strategy as for MPI and Java and have each work item compute several iterations.
- And then how to combine . . .

Slide 24

Numerical Integration in OpenCL, Continued

- Unlike OpenMP and MPI, OpenCL doesn't have anything built in to help with reduction. We can write our own (as we did in Java), but . . .
- Synchronizing among work items can be difficult: "Barrier" synchronization is available within each work group, but there's no way to apply it across work groups(!).
- So our strategy . . . Next time!

Slide 25

Minute Essay

- Questions?