Slide 1

Administrivia

- Homework 2 due next Wednesday. Okay? How are you doing with it? I'm considering making a slight revision in how to measure accuracy as a function of number of UEs.
- I'm intending to record another make-up lecture to partly compensate for Monday. Earlier this week!

Slide 2

OpenCL — Review/Recap

- Explicitly defines computation in terms of some parts that execute on a "host computer" and others that execute on a "compute device" (typically a GPU but doesn't have to be).
- Intended to be very portable but also to not hide too much from the programmer.
- Result is that programmers must deal with a lot of low-level details. However, many of those details the same from program to program and can be encapsulated in a library. I wrote one for my own use; you can use it too.

Slide 3

OpenCL — Numerical Integration Review/Recap

- Look again at code.
- Questions?

Slide 4

A Few Words About Design Patterns

- Title of our book includes the word “patterns”.
- What do we mean? “Design patterns”.

Slide 5

A Few (More) Words About Design Patterns

- Idea originated with architect Christopher Alexander (first book 1977). Basic idea: Look for problems that have to be solved over and over, and try to come up with "expert" solution, then write it in a form accessible to others. Usually this means adopting "pattern format" to use for all patterns. Characteristics of a good pattern:
  - Neat balancing of competing "forces" (tradeoffs).
  - Name either tells you what it's about, or is a good addition to vocabulary.
  - "Aha!" aspect.
- First used in CS in OOD/OOP, about 1987. Really started to take off in OO community with "Gang of Four" book (Gamma, Helms, Johnson, and Vlissides; 1995). Now can find people writing patterns in many, many areas.
- Simple low-level example — iterator.

Slide 6

"A Pattern Language for Parallel Programming"?

- Goal of our book (and preceding work): Apply this idea in parallel computing.
- We started out looking for patterns representing high-level structures for parallel programs, thinking there might be a dozen of them.
- At some point we realized we also wanted to talk about how you get from the original problem to one of these structures — i.e., how do expert parallel programmers think about how to decompose a problem, etc.? and also about commonly-occurring data structures and program structures, and how to map high-level designs/structures into real programming environments.
- After much thought and discussion . . .

Slide 7

"A Pattern Language for Parallel Programming", Continued

- Eventually: Four-layer "pattern language". (Note that "pattern language" connotes common vocabulary more than grammatical structure. Not a programming language!)
- We figured it would be a starting point but might need to be revised and extended. Indeed, that's so, especially (IMO) to adapt to changing state of the world.
- Much work has been done on that, primarily by Mattson and Sanders and a group at UC Berkeley. Project seems to be somewhat stalled at this point, but maybe someday?

Slide 8

Overall Organization of Our Pattern Language

- Four "design spaces" corresponding to phases in design.
  - *Finding Concurrency* — how to decompose problems, analyze decomposition.
  - *Algorithm Structure* — high-level program structures.
  - *Supporting Structures* — program structures, data structures.
  - *Implementation Mechanisms* — generic discussion of programming environment "building blocks".
- Idea is that you start at the top, work your way down, possibly with some backtracking.

Slide 9

Finding Concurrency — Preview

- Decomposition patterns (*Task Decomposition*, *Data Decomposition*): Break problem into tasks that maybe can execute concurrently.
- Dependency analysis patterns (*Group Tasks*, *Order Tasks*, *Data Sharing*): Organize tasks into groups, analyze dependencies among them.
- *Design Evaluation*: Review what you have so far, possibly backtrack.

Slide 10

Algorithm Structure — Preview

- *Task Parallelism* — decompose problem into lots of tasks, independent or nearly so. Example: numerical integration.
- *Divide and Conquer* — decompose recursively as in divide-and-conquer algorithms. Examples: quicksort, mergesort.
- *Geometric Decomposition* — decompose based on data (by rows, by columns, etc.). Distinguised from *Task Parallism* by dependencies among points/regions. Example: Mesh-based computation.
- *Recursive Data* — rethink computation to expose unexpected concurrency. Ignore for now.
- *Pipeline* — decompose based on assembly-line analogy.
- *Event-Based Coordination* — decompose problem into entities interacting asynchronously.

Slide 11

Supporting Structures — Preview

- Program structure patterns:
  - *SPMD* (Single Program, Multiple Data) — “like an MPI program”.
  - *Loop Parallelism* — “like an OpenMP program”.
  - *Master/Worker* — like the name suggests.
  - *Fork/Join* — when none of the others fits.
- Data structure patterns:
  - *Shared Queue* — example of applying *Shared Data*.
  - *Distributed Array*.

Slide 12

Implementation Mechanisms — Preview

- Generic discussion of “building blocks” for parallel programming — analogous to assignment, if/then/else, loops in procedural programming languages. (Can think of this as “what basic questions do I ask about a new parallel programming environment?”)
- Three basic categories:
  - UE management.
  - Synchronization.
  - Communication.

Slide 13

Example Applications

- Before starting on *Finding Concurrency* patterns — two example applications to be used as running examples.
- (Next time.)