Advanced Multicore Programming

Dr Andrew Moss

[1]BTH Karlskrona

September 9, 2011

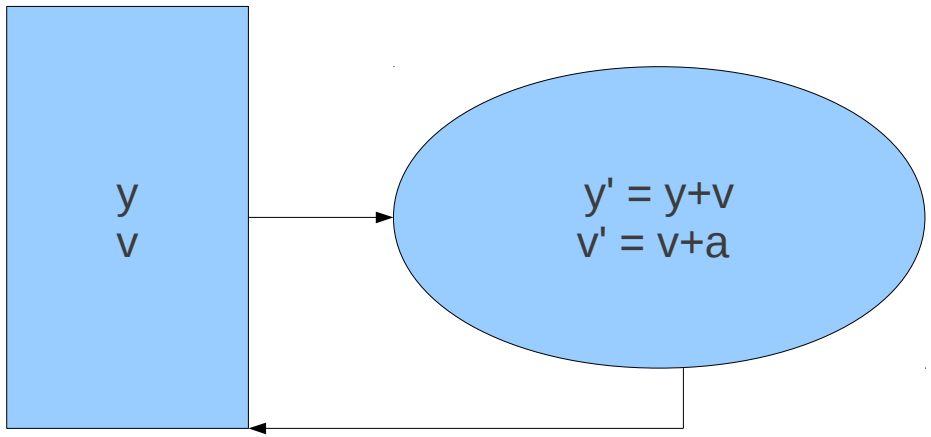
Kernels

A kernel is the "core", or "center" of a problem.

- A numeric kernel is the equation we get after removing control flow.
- If we have a problem where the core evalutes the same maths on the same I/O
  - Definitely fits well into a SIMD model.

Example (bouncing)

- updateState kernel simulates a particle defined by (y, v)



Simulations

Most interesting simulations are continous systems.

- Described by differential equations.
- Running the system = extracting particular states
  - Integration.

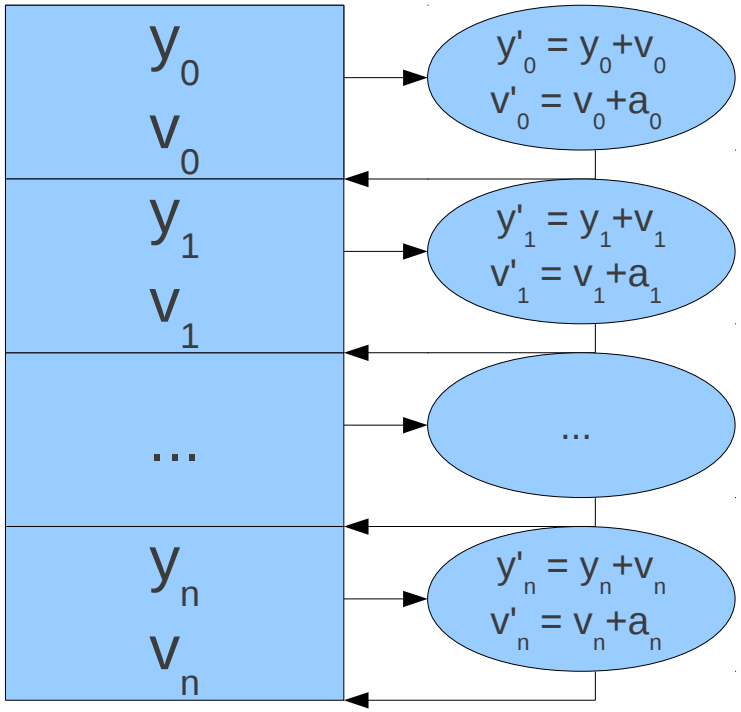
Simplest form of integration on a computer

- Euler's Method (good wikipedia page)
- In simple terms treat the differential as a first-order approx.
- Add it.
- Repeat.
- Which is what the first slide says (removed the intermediate steps)

Kernels II

Bouncing problem is a "hello world" for SIMD

- Every particle has independent motion
- Not one (y, v), but a large set of $(y_0, v_0), \ldots, (y_n, v_n)$
- Each kernel reads in a particular (y_i, v_i)
  - Computes the next step (y_i', v_i')
  - Outputs that part of the state
- Inputs / Outputs are disjoint for every kernel instance (perfect parallelism)
- There is something about this picture that differs from the code (avoids a caching problem)



Meshes

Running the same code in each thread

- Hopefully we are processing different data
- Each thread needs to know which one it is...

The bouncing example uses a 1D set of particles

- Thread IDs are one-dimensional
- Hardware will typically support 2D, and 3D IDs
- For higher dimensional problems it is just a striding function

One piece of code + dense (contiguous range) of parameters = mesh

Debugging Techniques

You will need two types of debugging

- Things that go wrong on the host (quite standard)
- Things that go wrong inside a kernel (need to be more inventive)

Host debugging

- Standard debugger on UNIX systems is `gdb`
- Ensure you have debugging enabled in the compiler (`-g` switch)
- Load your binary into gdb: `gdb progname`
- New command line interface inside the debugger
- Some basic commands
  - `run` will start execution
  - `bt` will show you a stack-trace
  - `quit` takes you back to the shell

More debugging

If you are used to watches / breakpoints inside an IDE debugger.

- As long as you have symbols in the binary you can do this in gdb
  - More info in `gdb`
- Or, if that is too much hassle, use `printf` for debugging
  - Too much text output is less of a problem in UNIX
  - Use `grep` or other filters to find what you are looking for.

On the card

- No debugging facilities (breakpoints, stepping etc).
- Need to be quite old fashioned at diagnosing problems
- Read the code for the Buffer/BufferCL classes
- Insert Card to Host transfers to pull back debugging
  - Remove them again for performance measurement
- Use a region of the array for extra debugging info
  - Write into it from the kernel.

Resources

Language Reference

- `http://www.khronos.org/files/opencl-quick-reference-card.pdf`

Runtime Reference

- `http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/`

Issues to remember

OpenCL has been designed to be familiar to a C programmer

- But it is not C . . .
- Subtle differences are the ones that will catch you out
- Place to check `http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf`

Arithmetic may be different to your expectations

- Rounding mode
- Overflow
- . . .

Remember that casting is doing some implicit arithmetic for you. . .

Exercises

Use separate source / produce separate binaries for each

- I want to see that you know how to build code on top of the framework
- I want to see that you know how to extend the Makefile

These are some simple starting points. Looking for simple exercises that build on the demo code — suggestions welcome.

- Sticking to simple 2D position / velocity
  - Regular sinusoidal motion (ie waves rippling across mesh)
  - Generate a color buffer as well
- Change state to position / velocity in 3D
  - Switch mesh to GL_POINTS and draw particle positions.
  - Bounce from all six walls.
  - Particle fountain (no collisions).
    - Simple version with "one shot" random spread.
    - Continous fountain - recycle "dead" particles.
  - Six sets of particles (each a different colour), different gravity direction.

Exercises II

Inelastic collisions

- Six sets of particles, each set with a different gravity vector.
- All particles should collide with all walls, losing some momentum each time.
- When particles come to rest reinitialise them (fountain).

Particle to particle collisions

- Check for collisions between all pairs of particles
- Will need to treat each particle as a volume. . .
- Don't worry about efficiency too much
  - Small sets (hundreds) of particles
  - More about exploring the programming model than optimising the simulation