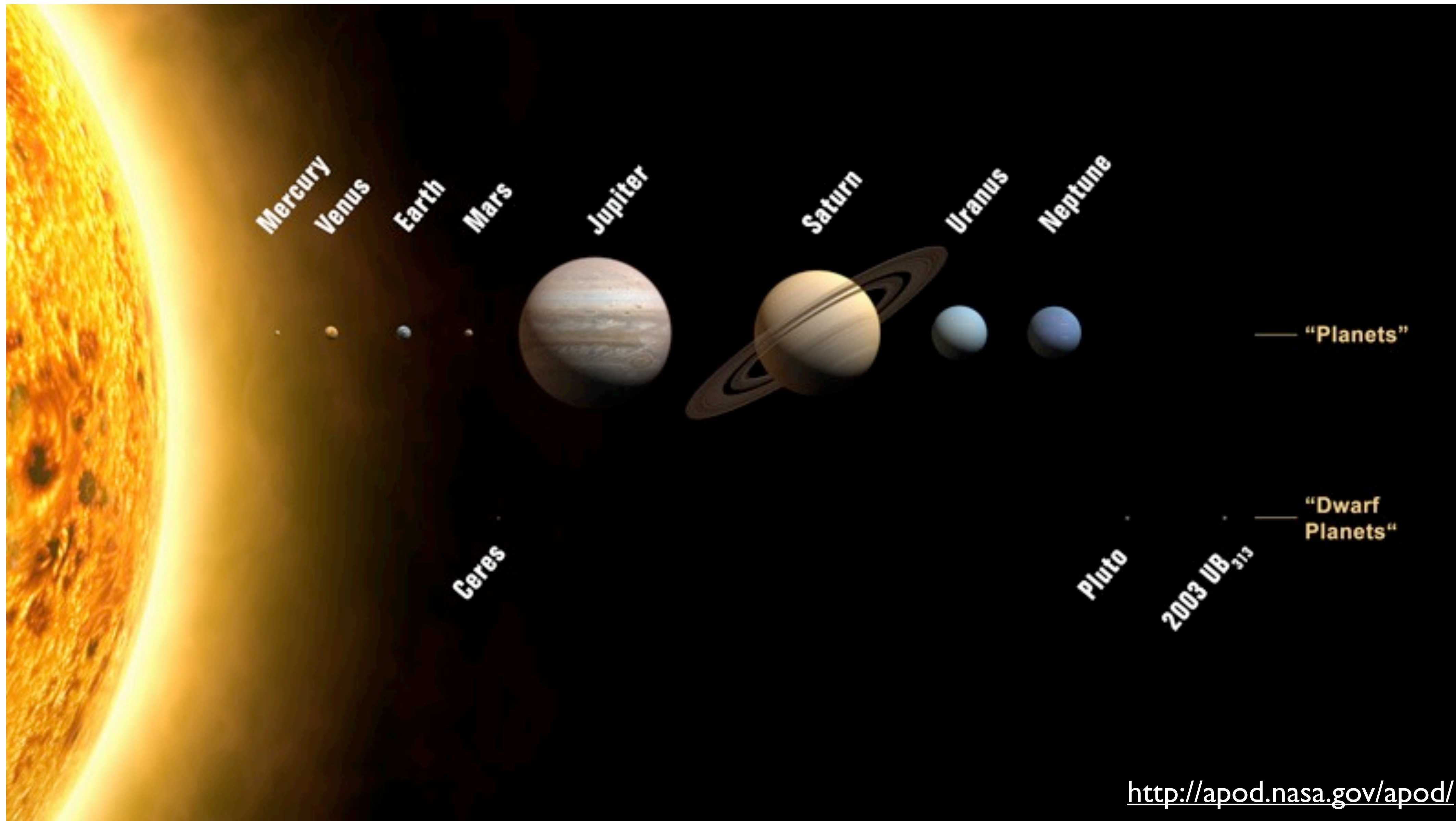
GPU, Multi/Many Core Computing I: Introduction to HPC

Exercise 1: *n*-body problem



- predict the motions of celestial body
- deduce past motions

Formulation

- n point masses $m_1, m_2, ..., m_n$
- Positions $\mathbf{q_1}(t), \mathbf{q_2}(t), ..., \mathbf{q_n}(t)$.
- Initial positions $\mathbf{q_1}(0), \mathbf{q_2}(0), ..., \mathbf{q_n}(0)$

  and velocities $\dot{\mathbf{q_1}}(0), \dot{\mathbf{q_2}}(0), ..., \dot{\mathbf{q_n}}(0)$ are known

- Motion of $\mathbf{q}_j$ described with a 2nd-order ODE:

$$m_j \ddot{\mathbf{q_j}} = G \sum_{k \neq j} \frac{m_j m_k (\mathbf{q}_k - \mathbf{q}_j)}{\|\mathbf{q}_k - \mathbf{q}_j\|^3}.$$

Discretization

- $6n$ degrees of freedom:

$$\mathbf{x}_j^i := \mathbf{q}_j(i\ \delta t)$$
$$\mathbf{v}_j^i := \dot{\mathbf{q}}_j(i\ \delta t)$$

- Right-hand side is evaluated as:

$$\mathbf{a}_j^i := \ddot{\mathbf{q}}_\mathbf{j}(i\ \delta t)$$
$$= G \sum_{k \neq j} \frac{m_k(\mathbf{x}_k^i - \mathbf{x}_j^i)}{\|\mathbf{x}_k^i - \mathbf{x}_j^i\|^3}.$$

- Numerical integration: Velocity-Verlet:

$$\mathbf{x}_j^{i+1} := \mathbf{x}_j^i + \delta t\ \mathbf{v}_j^i + \frac{1}{2}\delta t^2 \mathbf{a}_j^i$$
$$\mathbf{v}_j^{i+1} := \mathbf{v}_j^i + \frac{\delta t}{2}(\mathbf{a}_j^{i+1} + \mathbf{a}_j^i)$$

Performance analysis

- Example Kernel: $\mathrm{K}(\mathbf{x}, \alpha) \rightarrow x_i := \alpha\ x_i$
  - **x** is vector of n reals, alpha is a given scalar
- Quantities that determine performance:
  - the total amount (M) of transfers (read and writes in number of real components)

    => here: M = 2*n

Performance analysis

- Example Kernel: $\mathrm{K}(\mathbf{x}, \alpha) \to x_i := \alpha\ x_i$
  - **x** is vector of n reals, alpha is a given scalar
- Quantities that determine performance:
  - the total amount of transfers M = 2*n
  - total number (C) of floating point operations

    => here: C = n

Performance analysis

- Example Kernel: $\mathrm{K}(\mathbf{x}, \alpha) \rightarrow x_i := \alpha\ x_i$
  - **x** is vector of n reals, alpha is a given scalar
- Quantities that determine performance:
  - the total amount of transfers M = 2*n
  - total number of FLOP C = n
  - the number (R) of floating point operations per transferred real component
    => here: R = 0.5

Kernels

$$\mathbf{x}_j^{i+1} := \mathbf{x}_j^i + \delta t\ \mathbf{v}_j^i + \frac{1}{2}\delta t^2 \mathbf{a}_j^i,$$

$$\mathbf{v}_j^{i+1} := \mathbf{v}_j^i + \frac{\delta t}{2}\left(\mathbf{a}_j^{i+1} + \mathbf{a}_j^i\right).$$

HOW?

$$\mathrm{K1}(\mathbf{y}, \mathbf{z}, \alpha) \rightarrow y_i := y_i + \alpha\ z_i$$

$$\mathrm{K2}(\beta) \rightarrow v_i := v_i + \beta\ a_i(\mathbf{x})$$

K1

Given a scalar and two vectors of 3*n reals, update **y**

=> 3*n independent operations

K2

Given positions **x**. Update velocity **v** and acceleration **a**.

=> n independent operations

Note: First time step needs special treatment...

Nvidia CUDA/OpenCL

- Profiling tools in the CUDA toolkit
- Lot of code examples in the SDK
- OpenCL 1.1 specifications

AMD OpenCL

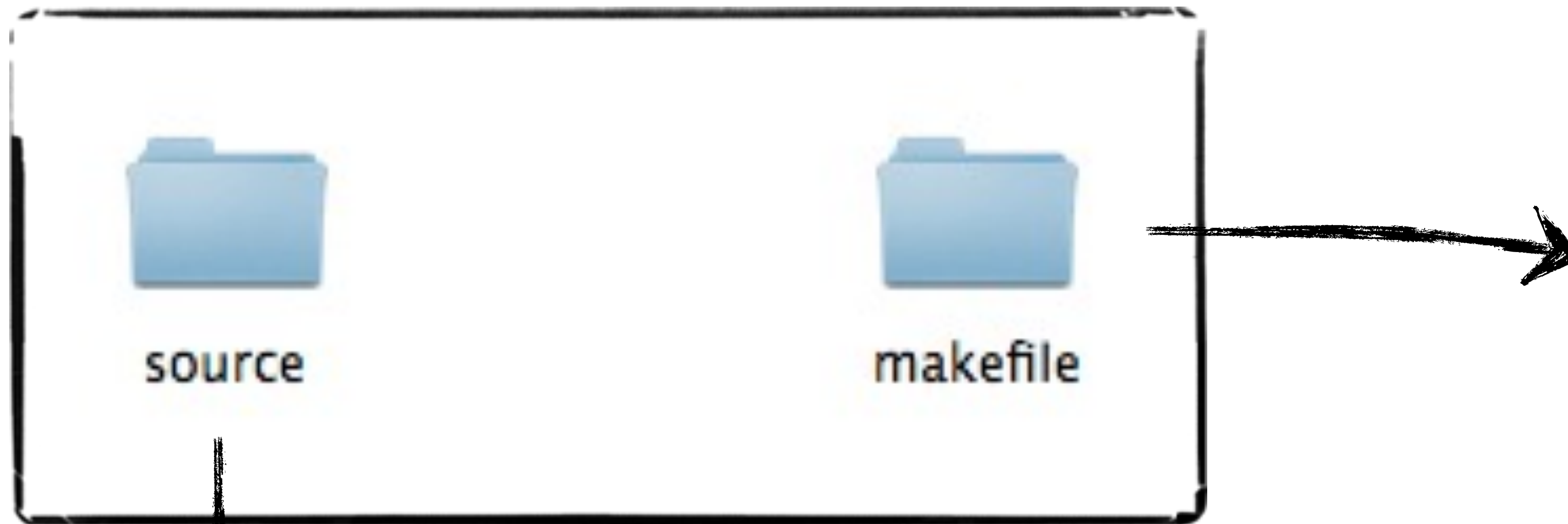
- Sample codes in the AMD APP SDK
- OpenCL 1.1 specifications

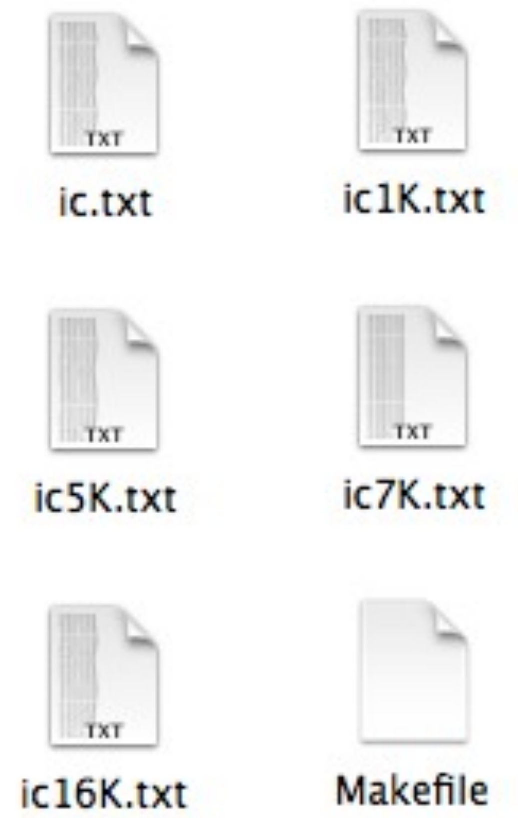
OpenCL Profiling

- Profiling on Windows:
  - AMD APP (contains different useful instruments)
  - Nvidia OpenCL Profiler
- Profiling on Linux/MacOs:
  - if Nvidia GPU: use Nvidia OpenCL profiler
  - else: not possible (or is it?)

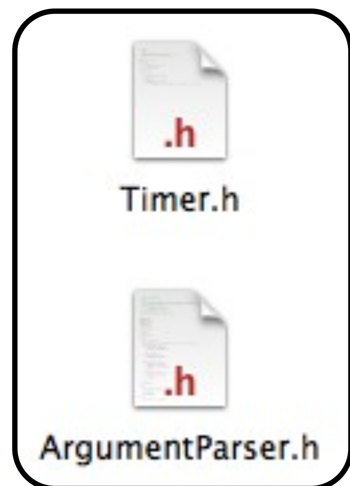
Code skeleton

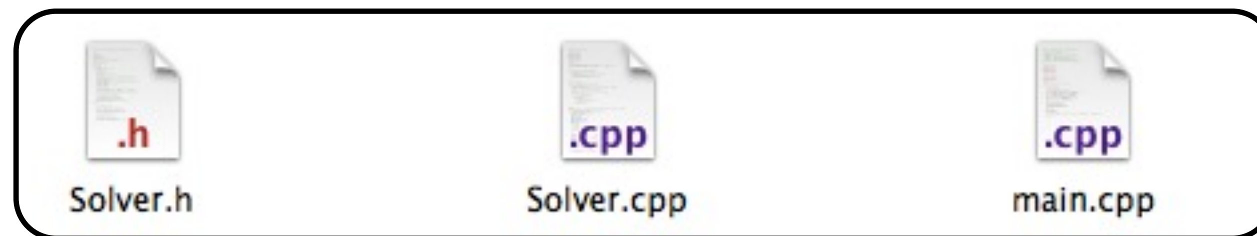
Makefile & initial conditions

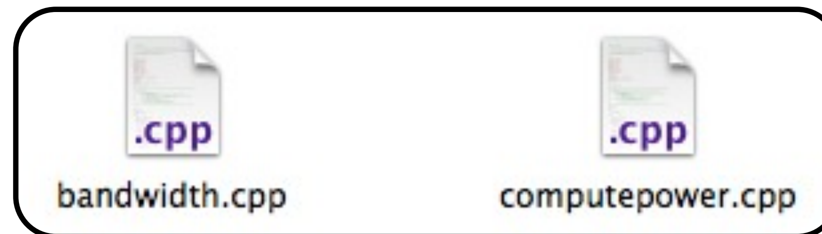




n-body solver







Helpers

Benchmarks

USAGE

- USAGE: ./gravity [OPTIONS]
- OPTIONS:
  - -file FILE : reads in FILE as initial condition
  - -tend TEND : simulation runs until t = TEND
  - -dt DT : time step DT is used
  - -silent : turn off diags and dumps (to do timings)
- DEFAULT: ./gravity -file ic.txt -tend 5 -dt 0.01

Makefile

- USAGE:
  - “make” compiles n body solver
  - “make bandwidth” compiles bandwidth.cpp (ditto with computepower)
- SETUP:
  - change flags in “USER SETUP”
- ADDING FILES:
  - change flag “OBJECTS”

Makefile setup

```
########################
# USER SETUP
########################
# use release or debug settings
config ?= release
# uncomment if you want to use CUDA or OpenCL
#USE_CL = yes
#USE_CUDA = yes
# define compiler
CC=gcc
LD=$(CC)
# NVCC only needed to compile CUDA files
#NVCC = /usr/cuda/bin/nvcc
NVCC = /usr/local/cuda/bin/nvcc
# contains cuda.h for CUDA and CL/cl.h for OpenCL (not needed for Mac)
#GPUINC = -I/usr/cuda/include
GPUINC = -I/Developer/GPU\ Computing/C/common/inc/
# contain libcudart and libcuda for CUDA and libOpenCL for OpenCL (not needed for Mac)
#GPULIB = -L/usr/cuda/lib64 -L/usr/lib64/nvidia
GPULIB = -L/usr/local/cuda/lib
########################
# define files to compile
OBJECTS = main.o Solver.o
########################
```

Override with "make config=debug"

Needed to include proper libraries etc

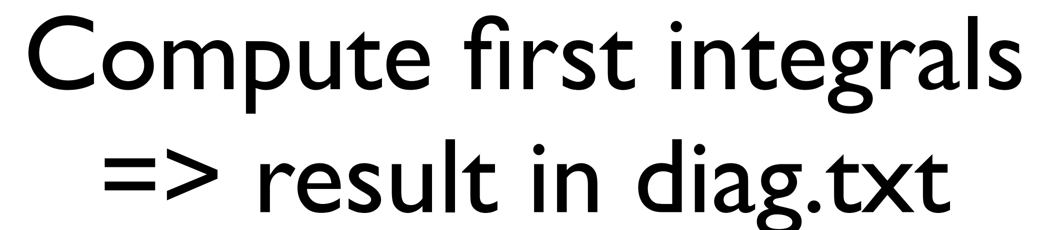
Default values in CAB H57

Default values for Mac

Add Foo.o to compile Foo.cpp (with gcc) or Foo.cu (with nvcc)

To be implemented

```cpp
/** Type to use for floating points. */
typedef float Real;

class Solver
{
protected:
    /** Number of particles. */
    int n_particles;

    /** Are we in silent mode? */
    bool silent;

    /** Position of particle i is x[j] with j in [3*i,3*i+2]. */
    std::vector<Real> x;
    /** Velocity of particle i is v[j] with j in [3*i,3*i+2]. */
    std::vector<Real> v;
    /** Acceleration of particle i is a[j] with j in [3*i,3*i+2]. */
    std::vector<Real> a;
    /** Mass of particle i is m[i]. */
    std::vector<Real> m;

    /** K1 kernel computing y[j] += alpha * z[j]. */
    void _K1_kernel(Real * y, const Real * z, const Real alpha);

    /** K2 kernel computing accelerations and v[j] += beta * a[j]. */
    void _K2_kernel(const Real beta);

    /** Dump current state to xyz file. */
    void _dumpToFile(size_t stepid, Real t);

    /** Get and report first integrals as diagnostics. */
    void _diag(size_t stepid);

public:

    /** Standard constructor. */
    Solver(): n_particles(0), silent(false) { }

    /** Read in particles from given file. */
    void initializeFromFile(std::string filename);

    /** Turn on silent mode. */
    void shutUp() { silent = true; }

    /** Run simulation until time_end with timestep dt. */
    void run(Real time_end, Real dt);
};
```



Hints

- From std::vector<Real> vr to Real * pr
  - Storage of vector is contiguous in memory
    => pr = &vr.front()
- Conservation of first integrals
  - will not be conserved exactly (why?)

To gather

- statistics about single time steps (time, GFLOP/s)
  - you need to implement a timer!

To report

- overall execution time
- 20th and 80th percentiles (for simulation steps):
  - time
  - GFLOP/s (for both K1 and K2)
- M, C, R
- estimated peak performance

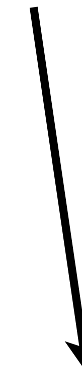
Benchmarks

- bandwidth.cpp
  - `make bandwidth`
  - `./bandwidth -SSE false -size 100 -nthreads 1 -double false`
- computepower.cpp
  - `make computepower`
  - `./computepower -SSE false -nthreads 1 -double false`

```cpp
/** Type to use for floating points. */
typedef float Real;

class Solver
{
protected:
    /** Number of particles. */
    int n_particles;

    /** Are we in silent mode? */
    bool silent;

    /** Position of particle i is x[j] with j in [3*i,3*i+2]. */
    std::vector<Real> x;
    /** Velocity of particle i is v[j] with j in [3*i,3*i+2]. */
    std::vector<Real> v;
    /** Acceleration of particle i is a[j] with j in [3*i,3*i+2]. */
    std::vector<Real> a;
    /** Mass of particle i is m[i]. */
    std::vector<Real> m;

    /** K1 kernel computing y[j] += alpha * z[j]. */
    void _K1_kernel(Real * y, const Real * z, const Real alpha);

    /** K2 kernel computing accelerations and v[j] += beta * a[j]. */
    void _K2_kernel(const Real beta);

    /** Dump current state to xyz file. */
    void _dumpToFile(size_t stepid, Real t);

    /** Get and report first integrals as diagnostics. */
    void _diag(size_t stepid);

public:

    /** Standard constructor. */
    Solver(): n_particles(0), silent(false) { }

    /** Read in particles from given file. */
    void initializeFromFile(std::string filename);

    /** Turn on silent mode. */
    void shutUp() { silent = true; }

    /** Run simulation until time_end with timestep dt. */
    void run(Real time_end, Real dt);
};
```

To be extended

- Use polymorphism
- Implement the class SolverGPU



You can use either **OpenCL** or **CUDA**

To report

- Statistics as in Question 3
- Peak GB/s and FLOP/s of GPU

Show us your results with two VMD movies!

- 1 movie for multicore CPU
- 1 movie for GPU

The current implementation does not consider symmetry of forces:

- how would you exploit this information on a GPU?
- can you estimate the improvement?

VMD

- Obtain from http://www.ks.uiuc.edu/Research/vmd/
- Load data: File/New molecule...
  - Load output.xyz from simulation
- Change visuals: Graphics/Representation...
  - “Drawing method” set to “VWD” to get spheres
- Movie: Extensions/Visualization/Movie Maker...
  - Set “Movie Settings” to “Trajectory”