Autogenerating gpu-accelerated LAMMPS pair styles

Nathan Chong, Nicolas Loriant, Paul Kelly
{nyc04, nloriant, phjk}@doc.ic.ac.uk
Software Performance Optimisation Group

George Marketos, Catherine O’Sullivan
{g.marketos, cath.osullivan}@imperial.ac.uk
Geotechnics Group
Doing experiments...

Connectivity = number of contacts a particle has with other contacts

Using DEM to develop a rational basis for dam filter design
__global__ void bpp_compute_kernel(
    int nparticles,
#ifdef AOS_LAYOUT
    struct particle *particle_aos,
#else
    struct particle particle_soa,
#endif
    int *numneigh, double *force, double *torque
#ifdef NEWTON_THIRD
    , double3 *fdelta, double3 *tdeltaj
#endif ) {

    __shared__ double ftmp[NSLOT*3];
    __shared__ double ttmp[NSLOT*3];
    int jj = threadIdx.x; int idx = blockIdx.x.x;
    if (idx < nparticles && jj < numneigh[idx]) { ... }

vs. writing experiments
Implementation details

- CPU or GPU?
- Choice of decomposition
- Data-layout and orchestration
- Algorithm tradeoffs
- Effective use of resources
Hardware choices

• Intel SCCC, MIC

• AMD Fusion

• NVIDIA Fermi, Kepler, Maxwell
Opportunity for Software Optimization Group

- Targeting granular community of LAMMPS
- Make it easy to write pair styles that execute with good performance on a variety of hardware platforms
Key idea: abstraction

• Separate *specification* and implementation

• pairwise interaction (mechanical model)

• data/access pattern

• implementation details (eg, how to decompose the problem onto parallel hardware)
Related work

• Molecular Dynamics: OpenMM, NAMD

• Unstructured Grid: OP2, Liszt

• Other: SPIRAL, FFTW, ATLAS
The NxM problem

LJ  Hertz  Hooke  GB

multicore/avx implementations

LJ  Hertz  Hooke  GB

cuda gpu implementations
Abstraction

LJ  Hertz  Hooke  GB

Simple, abstract specifications

pairgen

multicore/avx implementations

CUDA GPU implementations
**Step 1. Create parameter file**

```plaintext
name: hertz
parameters:
- name: x
type: double
arity: 3
set: P
access: RO
- name: v
- name: omega
- name: radius
arity: 1
name: type
arity: 1
type: int
- name: force
access: SUM
- name: torque
access: SUM
- name: shear
set: N
access: RW
constants:
- name: dt
- name: nktv2p
- name: yeff
- name: geff
- name: betaeff
```
Step 1. Create parameter file

We invoke pairgen at the command line:

$ pairgen.py hertz.yml
Step 2. Add detail to generated code

```c
__device__ void hertz_pair_kernel(
    double xi[3], double xj[3],
    double vi[3], double vj[3],
    double omegai[3], double omegaj[3],
    double radiusi, double radiusj,
    double massi, double massj,
    int typei, int typej,
    double forcei_delta[3],
    double torquei_delta[3],
    double shear[3],
    int *touch) {
    //fill me in
}
```
Step 3. Autogenerated pairgen output (implementation)

```c
__global__ void bpp_compute_kernel(
    int nparticles,
#ifdef AOS_LAYOUT
    struct particle *particle_aos,
#else
    struct particle particle_soa,
#endif
    int *numneigh, double *force, double *torque
#ifdef NEWTON_THIRD
    , double3 *fdelta, double3 *tdeltaj
#endif ) {
    __shared__ double ftmp[NSLOT*3];
    __shared__ double ttmp[NSLOT*3];
    int jj = threadIdx.x; int idx = blockIdx.x;
    if (idx < nparticles && jj < numneigh[idx]) { ... }
```
Experimental evaluation

• Simulation data taken from raining of particles into a box under gravity
• Constant number of particles, 45.6K
• Number of neighbors increases as simulation proceeds
• Vary number of neighbors for problem size
## Implementations

<table>
<thead>
<tr>
<th>Neighbor list datastructure</th>
<th>Decomposition</th>
<th>Neighbor list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>serial</strong></td>
<td>None</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>dnl</strong></td>
<td>Dense-neighbor</td>
<td>Thread per neighbor</td>
</tr>
<tr>
<td><strong>tpp-full</strong></td>
<td>Sparse-neighbor</td>
<td>Thread per particle</td>
</tr>
<tr>
<td><strong>bpp-full</strong></td>
<td>Sparse-neighbor</td>
<td>Block per particle</td>
</tr>
<tr>
<td><strong>tpp-half</strong></td>
<td>Sparse-neighbor</td>
<td>Thread per particle</td>
</tr>
<tr>
<td><strong>bpp-half</strong></td>
<td>Sparse-neighbor</td>
<td>Block per particle</td>
</tr>
</tbody>
</table>

Using `pairgen`
Conclusions

• Make it easy to write pair styles that execute with good performance on a variety of hardware platforms

• Pairgen addresses abstraction and performance (but not yet performance portability)
Future work

• Modifying pairgen to achieve a dense neighbour implementation (*ongoing*)

• Generating implementations for different hardware

• Extending this domain-specific approach to other parts of the LAMMPS simulation loop (in particular, *fixes*).

• Dynamically switching between implementation choices as simulation characteristics change
Nathan Chong

- http://www.doc.ic.ac.uk/nyc04/
- nyc04@doc.ic.ac.uk
- Please email for information on test driving pairgen!
Experiment details
# Toolset

<table>
<thead>
<tr>
<th></th>
<th>Version</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>g++</td>
<td>4.4.3</td>
<td>-O2</td>
</tr>
<tr>
<td>nvcc</td>
<td>3.2</td>
<td>-arch sm_13</td>
</tr>
</tbody>
</table>
Hardware

• Intel P4 3.6GHz
• 2GB memory
• NVidia GeForce GTX 480 (Fermi)
• 1.5GB global memory
• 15 multiprocessors/120 cores