Accelerating Glassy Dynamics using GPUs

Peter Colberg

University of Toronto

July 14th, 2010

Objective: Long-time stable, optimally accelerated MD

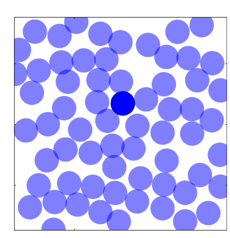
Slow dynamics of a super-cooled liquid close to the glass transition

- soft-sphere particles interact via short-ranged C^2 potential (LJ)
- propagated by a symplectic integrator (velocity-Verlet)

Parallel implementation for NVIDIA G80/G200 GPUs with CUDA

- challenge: speedup of GPU over CPU of order 100
- floating-point precision is crucial
- challenge: absent or mediocre ($1/8$ DP/SP) native double precision
- correlation functions for 10^5 or more particles
- challenge: low memory bandwidth between GPU and host system

Concurrency: Parallelization of the MD step



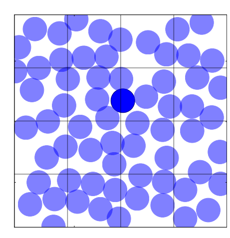
velocity-Verlet trivial to parallelize for the GPU

- n-th thread $\mapsto$ n-th particle
- coalesced (linearly ordered) memory access

forces parallelized by domain decomposition

- bin particles into variable-length cells on GPU
  - order particles into cells with radix sort
  - efficient use of memory due to consecutive cells
- construct Verlet neighbour lists on GPU
  - scan own and 26 (or 8) neighbour cells
  - store in fixed-size neighbour lists (uncoalesced)
- "skin" to delay updates to every 10-100 steps
  - reduce particle displacements on GPU

Concurrency: Parallelization of the MD step



velocity-Verlet trivial to parallelize for the GPU

- n-th thread $\mapsto$ n-th particle
- coalesced (linearly ordered) memory access

forces parallelized by domain decomposition

- bin particles into variable-length cells on GPU
  - order particles into cells with radix sort
  - efficient use of memory due to consecutive cells
- construct Verlet neighbour lists on GPU
  - scan own and 26 (or 8) neighbour cells
  - store in fixed-size neighbour lists (uncoalesced)
- "skin" to delay updates to every 10-100 steps
  - reduce particle displacements on GPU

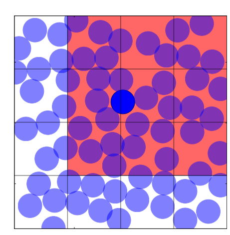
Concurrency: Parallelization of the MD step

velocity-Verlet trivial to parallelize for the GPU

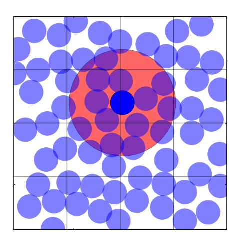
- n-th thread $\mapsto$ n-th particle
- coalesced (linearly ordered) memory access

forces parallelized by domain decomposition

- bin particles into variable-length cells on GPU
  - order particles into cells with radix sort
  - efficient use of memory due to consecutive cells
- construct Verlet neighbour lists on GPU
  - scan own and 26 (or 8) neighbour cells
  - store in fixed-size neighbour lists (uncoalesced)
- "skin" to delay updates to every 10-100 steps
  - reduce particle displacements on GPU



Concurrency: Parallelization of the MD step



velocity-Verlet trivial to parallelize for the GPU

- n-th thread $\mapsto$ n-th particle
- coalesced (linearly ordered) memory access

forces parallelized by domain decomposition

- bin particles into variable-length cells on GPU
  - order particles into cells with radix sort
  - efficient use of memory due to consecutive cells
- construct Verlet neighbour lists on GPU
  - scan own and 26 (or 8) neighbour cells
  - store in fixed-size neighbour lists (uncoalesced)
- "skin" to delay updates to every 10-100 steps
  - reduce particle displacements on GPU

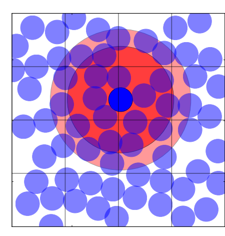
Concurrency: Parallelization of the MD step

velocity-Verlet trivial to parallelize for the GPU

- n-th thread $\mapsto$ n-th particle
- coalesced (linearly ordered) memory access

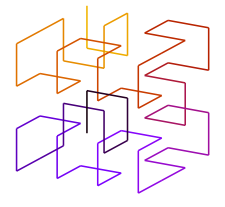
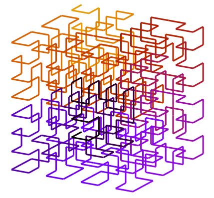
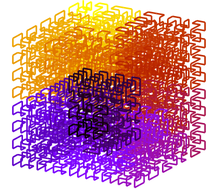
forces parallelized by domain decomposition

- bin particles into variable-length cells on GPU
  - order particles into cells with radix sort
  - efficient use of memory due to consecutive cells
- construct Verlet neighbour lists on GPU
  - scan own and 26 (or 8) neighbour cells
  - store in fixed-size neighbour lists (uncoalesced)
- "skin" to delay updates to every 10-100 steps
  - reduce particle displacements on GPU



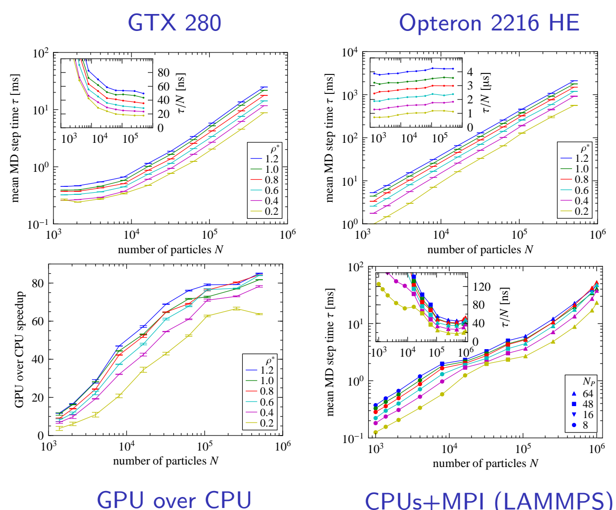
Texture locality: Hilbert's space-filling curve

- coordinates of interacting particles read in random order
  - fetch coordinates via read-only texture cache
  - small texture cache size necessitates memory locality

- periodically reorder particles in memory
  - Hilbert space-filling curve maps 3D $\mapsto$ 1D
  - recursively generate Hilbert curve on GPU using 8 vertex rules
  - generate permutation using radix sort
  - reorder particles using texture reads and coalesced writes

Performance: 80-fold speedup of GPU over CPU



Double-single floating-point precision

NVE ensemble simulation runs over 10^7 steps

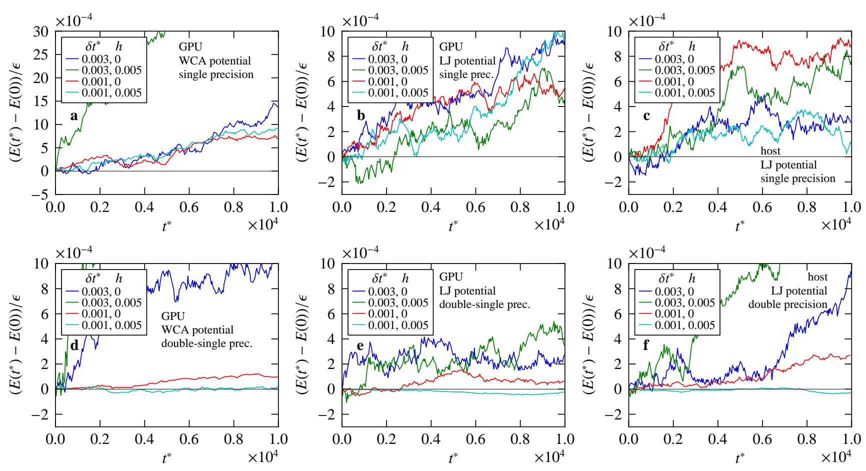
- large center of mass velocity drift with repulsive LJ potential
- smaller time-step does not improve energy conservation
- C^2 smoothing function does not decrease energy drift

Cause: accumulation of rounding errors

Solution: implement double-single using native single precision

- DSFUN90 package provides double-single Fortran routines
- porting to CUDA straightforward
- only needed in force summing and Verlet integrator
- execution times (GTX 280) increase merely by 20%

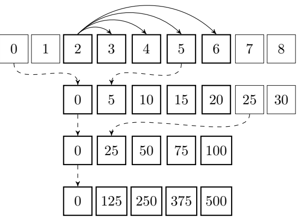
Precision: Energy conservation



Evaluation of time-correlation functions

Observe slowing dynamics over many decades in time

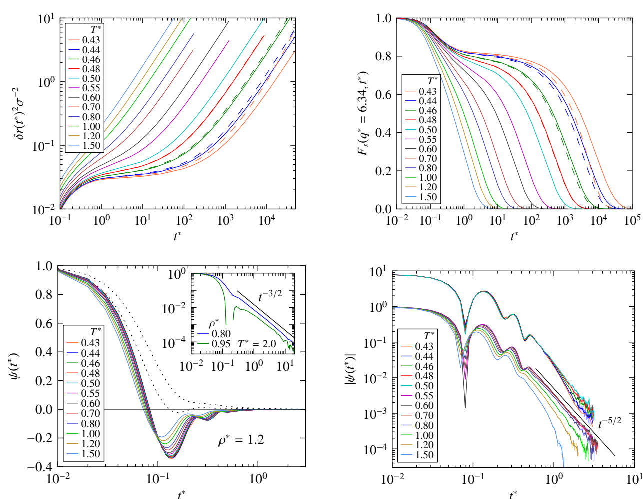
- evaluate on logarithmic time-grid using blocking scheme
- time interval Δt increases with block level
- blocks are correlated within a block level



Trajectories are kept in GPU memory to avoid bandwidth bottleneck

- correlation functions are averaged using reduction

Dynamic properties of the Kob-Andersen mixture



- C++ templates for code reuse and performance

```
template <
    typename Vector
  , dsfloat (*CorrelationFunction)(Vector/*r*/, Vector/*r0*/, Vector/*q*/)
  , typename T, typename U
>
__global__ void accumulate(T const* g_r, T const* g_r0, U q_vector, ...)
{
    dsfloat sum = 0;
    for (uint i = GTID; i < n; i += GTDIM) {
        sum += CorrelationFunction(g_r[i], g_r0[i], q_vector);
    }
    ...
}
// bind CUDA kernel to cuda::function<>
... = accumulate<vector<float, 3>, incoherent_scattering_function>;
... = accumulate<vector<float, 2>, incoherent_scattering_function>;
```

- 4D/3D/2D (double-single) vector algebra template functions

```
vector<dsfloat, dimension> vcm = 0;        // first moment <v>
dsfloat vsq = 0;                           // second moment <v^2>
for (unsigned int i = GTID; i < npart; i += GTDIM) {
    vector_type v;
    unsigned int tag;
    tie(v, tag) = untagged<vector_type>(g_v[i]);
    vcm += v;
    vsq += inner_prod(v, v);
}
// reduced values for this thread
s_vcm[TID] = vcm; s_vv[TID] = vv;
__syncthreads();
// compute reduced value for all threads in block
reduce<threads / 2, complex_sum_>(vcm, vv, s_vcm, s_vv);
```

- random number generators and distributions

```
typedef variant<set<rand48_rng, ...> > random_number_generator;
static __constant__ random_number_generator rng;

// fill array with normal distributed random numbers in [0.0, 1.0)
template <typename Rng>
__global__ void normal(float* v, unsigned int len, float mean, float sigma)
{
    // read random number generator state from global device memory
    typename Rng::state_type state = get<Rng>(rng)[GTID];

    for (unsigned int k = GTID; k < len; k += 2 * GTDIM) {
        tie(v[k], v[k + GTID]) = normal(get<Rng>(rng), state, mean, sigma);
    }
    // store random number generator state in global device memory
    get<Rng>(rng)[GTID] = state;
}
```

- CUDA C++ wrapper for integration into host code (g++, icc)

```
// fill array with normal distributed random numbers in [0.0, 1.0)
template <typename Rng>
void random<Rng>::normal(cuda::vector<float>& g_v, float mean, float sigma)
{
    try {
        cuda::configure(rng.dim.grid, rng.dim.block);
        get_random_kernel<Rng>().normal(g_v, g_v.size(), mean, sigma);
        cuda::thread::synchronize();
    }
    catch (cuda::error const& e) {
        LOG_ERROR("CUDA: " << e.what());
        throw exception("failed to fill vector with normal random numbers");
    }
}
```

Conclusion

- soft-sphere MD simulation optimal for CUDA acceleration
  - 80-fold speedup of GPU over CPU!
- optimized performance by use of basic parallel algorithms
  - radix sort
  - reduction
- numerical precision crucial for long-time stability
  - single precision simulations may yield incorrect results
  - double-single precision implements double with native single precision
  - acceptable performance penalty if used only for summing
- avoid GPU to host (and vice versa) memory bandwidth bottleneck
  - online evaluation of time-correlation functions

I would hereby like to thank. . .

- Professor Raymond Kapral
  - University of Toronto, Ontario, Canada

- Dr. Felix Höfling
  - Max Planck Institute for Metals Research, Stuttgart, Germany

- PD Dr. Jürgen Horbach, Dr. Matthias Sperl
  - German Aerospace Centre (DLR), Köln, Germany

- Professor Thomas Franosch
  - Universität Erlangen, Germany

- Professor Erwin Frey
  - Ludwig-Maximilians-Universität München, Germany

. . . and thank you!

Preprint: P. H. Colberg and F. Höfling, `arXiv:0912.3824`.

`http://research.colberg.org/projects/halmd`

HALMD is freely available under the GNU General Public License.

Stay tuned for *fully modular* HALMD v0.1.0!