Computer Physics Communications 195 (2015) 95–101

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Optimizing legacy molecular dynamics software with directive-based offload

W. Michael Brown [a,*], Jan-Michael Y. Carrillo [b], Nitin Gavhane [c], Foram M. Thakkar [c], Steven J. Plimpton [d]

[a] Intel Corporation, Portland, Oregon, USA

[b] Center for Nanophase Materials Sciences and Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA

[c] Computational Centre of Expertise, Shell India Markets Private Limited, Bangalore, India

[d] Multiscale Science, Sandia National Laboratories, Albuquerque, NM, USA

ARTICLE INFO

Article history:
Received 10 November 2014
Received in revised form
3 April 2015
Accepted 5 May 2015
Available online 14 May 2015

Keywords:
Molecular dynamics
Xeon Phi
GPU
Coprocessor
Accelerator
Many-core

ABSTRACT

Directive-based programming models are one solution for exploiting many-core coprocessors to increase simulation rates in molecular dynamics. They offer the potential to reduce code complexity with offload models that can selectively target computations to run on the CPU, the coprocessor, or both. In this paper, we describe modifications to the LAMMPS molecular dynamics code to enable concurrent calculations on a CPU and coprocessor. We demonstrate that standard molecular dynamics algorithms can run efficiently on both the CPU and an x86-based coprocessor using the same subroutines. As a consequence, we demonstrate that code optimizations for the coprocessor also result in speedups on the CPU; in extreme cases up to 4.7X. We provide results for LAMMPS benchmarks and for production molecular dynamics simulations using the Stampede hybrid supercomputer with both Intel® Xeon Phi™ coprocessors and NVIDIA GPUs. The optimizations presented have increased simulation rates by over 2X for organic molecules and over 7X for liquid crystals on Stampede. The optimizations are available as part of the "Intel package" supplied with LAMMPS.

1. Introduction

Issues with power consumption and heat dissipation have led to a trend towards many-core processors as an approach to increase parallelism with electrical power efficiency. Although bootable many-core processors are expected to be available in the future, current high performance computers exploit many-core processors with a hybrid configuration using nodes containing traditional CPUs along with graphics processing units (GPUs) or Intel® coprocessors. In these designs, source code modifications are necessary in order to efficiently use the system.

In our previous work, we have focused on the design of efficient algorithms to use GPU accelerators for large-scale molecular dynamics (MD) simulations [1–4]. For this work, a separate library was designed for the LAMMPS molecular dynamics software [5] with MD algorithms modified to run efficiently on GPUs. This library could be compiled using either CUDA or OpenCL. Although this approach has allowed for GPU-acceleration in production simulations, the use of a separate programming language and different algorithms on the CPU and GPU introduces additional code complexity and requires optimization of separate code paths depending on the target. For example, in MD, redundant calculation is typically used to avoid memory conflicts for force updates on the GPU where greater than 10,000 threads can be in flight simultaneously. In the case of 3-body potentials, this requires up to 3 times as many force calculations compared to the CPU algorithm, new neighbor list routines, and doubling of the volume of the border regions between neighboring MPI tasks in the spatial decomposition [4].

The Intel® Xeon Phi ™ coprocessor is an x86-based many-core processor that also connects to the host through the PCI express bus. Because the coprocessor runs a full-service Linux operating system, there are several options for using the coprocessor in HPC systems. These include "native" mode, where code is run solely on the coprocessors without involving the host processor, "symmetric" mode, where MPI tasks run on both the CPUs and the coprocessor, and "offload" mode, where the host offloads some of

* Corresponding author.
E-mail addresses: michael.w.brown@intel.com (W. Michael Brown), carrillojy@ornl.gov (J.-M.Y. Carrillo), Nitin.Gavhane@shell.com (N. Gavhane), Foram.Thakkar@shell.com (F.M. Thakkar), sjplimp@sandia.gov (S.J. Plimpton).

http://dx.doi.org/10.1016/j.cpc.2015.05.004
0010-4655/

the work to be performed on the coprocessor. The best choice will depend upon a number of factors, but for legacy HPC software, "offload" provides some advantages in that optimizations can be focused on select compute-intensive routines without consideration or alteration of the distributed memory parallelization. Therefore, we chose this approach to evaluate modification to the LAMMPS MD software to allow for utilization of Intel® Xeon Phi ™ coprocessors. Because the architecture is x86-based, we also examine how the modifications influence performance on more traditional Intel® Xeon® CPUs.

2. Methods

2.1. LAMMPS

In this work, we are considering enhancements to the LAMMPS molecular dynamics package [5]. LAMMPS is parallelized via MPI, using spatial-decomposition techniques that partition the simulation domain into smaller subdomains, one per processor. In this approach, each MPI task owns a set of *local* atoms that are within the subdomain, but also stores data for *ghost* atoms that are owned by another MPI task and within the cutoff distance at the subdomain borders. For these atoms at the borders, MPI communication can occur at every timestep because their energy is affected by atoms owned by multiple MPI tasks.

LAMMPS is a general purpose MD code capable of simulating biomolecules, polymers, materials, and mesoscale systems. It is also designed in a modular fashion with the goal of allowing additional functionality to be easily added. This is achieved via a variety of different style choices that are specified by the user in an input script and control the choice of force-field, constraints, time integration options, diagnostic computations, etc. At a high level, each style is implemented in the code as a C++ virtual base class with an appropriate interface to the rest of the code. For example, the choice of pair style (e.g. lj/cut for Lennard-Jones with a cutoff) selects a pairwise interaction model that is used for force, energy, and virial calculations. Individual pair styles are child classes that inherit the base class interface. Thus, adding a new pair style to the code (e.g. lj/cut/hybrid for a Lennard-Jones potential optimized for hybrid execution, is as conceptually simple as writing a new class with the appropriate handful of required methods or functions, some of which may be inherited from the parent-class pair style (lj/cut in this case).

2.2. Offload

For this work, we have used the Intel® Language Extensions for Offload (LEO) directives to handle data allocation on the coprocessor, asynchronous data transfer and computation offload, and synchronization. Because directives are used, the code can be compiled by any C++ compiler and used on machines that do not contain coprocessors. Additionally, LEO supports an *if* clause allowing the same routine to be called with and without offload. This can be used to perform computations on the CPUs and the coprocessor simultaneously. The LEO model is advantageous in that the code inside an offloaded region supports C++ and Fortran, has no restrictions on function calls to other routines, and can use different parallel programming models such as OpenMP, POSIX Threads, or Intel® Cilk™ Plus. Example directives with LEO are given in the listing:

```
double *a, *b, *c;
int N, C, F, L;

// ... setup code ...

// Allocate N elements on coprocessor 'C' for
// host allocation 'a', 'b', and 'c'
#pragma offload_transfer target(mic:C) \
  nocopy(a,b,c:length(N) alloc_if(1) free_if(0))

// Free memory used by x on coprocessor
#pragma offload_transfer target(mic:C) \
  nocopy(a,b,c:alloc_if(0) free_if(1))

// Offload computation to coprocessor
// - only if F == 1, otherwise compute on host
// - only transfer L elements, if L == 0, nothing
// - non-blocking offload, host pointer used as ref
// - do not allocate or free any arrays
#pragma offload target(mic:C) if(F) \
  in(a,b:length(L) alloc_if(0) free_if(0)) \
  out(c:length(L) alloc_if(0) free_if(0)) \
  signal(c)
{
  for (int i = start; i < end; i++)
    c = foo(a, b);
}

// Block until asynchronous offload with ref c done
#pragma offload_wait target(mic:C) wait(c)
```

Offload is used for both neighbor list builds and calculation of short-range terms including energies, forces, torques, and virials. In order to obtain performance, memory allocation on the coprocessor is never performed within a loop, unless necessary to grow an allocation to fit needed data. Additionally, data transfer for constant quantities (atom types, charges, etc.) is only repeated on timesteps when neighbor lists are re-built, since those are the only steps when atoms migrate to new processors, changing the per-atom data structures.

In order to utilize both the CPU and coprocessor on hybrid machines, offload for a fraction of the work is supported. The fraction is supplied as an optional parameter for the simulation run. If the offload fraction is 0, the coprocessor is not used. In the default case, dynamic load balancing is performed to automatically adjust the fraction at each neighbor list build based on the computational time being monitored on both the host and coprocessor. The objective is to minimize idle time on both. Currently, we are unable to attain data transfer times using LEO and therefore they are not included in the load balance calculations. The fraction for load balancing is calculated using a weighted mean with a 0.1 weight for the current timestep and 0.9 for the previous value. The load balance fraction is calculated so as to balance the time for neighbor list and short-range calculations on the coprocessor versus neighbor list, short-range, long-range, and bond/angle calculations on the host, since they run concurrently.

We have implemented two different approaches for dividing the work between the host and the coprocessor. In the first, the local atoms are divided so that the host and coprocessor each loop over the neighbors of a distinct set of atoms. In the second, the coprocessor is still assigned a subset of the local atoms, but no ghost atoms are included in the neighbor lists used on the coprocessor. The host loops over all local atoms, but for those evaluated on the coprocessor, only ghost atoms are included in the neighbor list. This approach offers more flexibility in overlapping MPI communications with computation on the coprocessor. In both cases, neighbor list builds keep track of the minimum and maximum indices for atoms used in the lists. Because LAMMPS employs methods to sort atom data based on spatial location, this approach can be used to reduce the amount of data transfer with the coprocessor and also the range of atoms involved in force accumulation for thread-private arrays.