Portability Initiatives for Scientific Computing and Simulation: Molecular Dynamics as a Case Study

Arnold Tharrington
Ada Sedova

Oak Ridge National Laboratory
National Center for Computational Sciences
Scientific Computing Group
Introduction

• The goal of the talk is to provide beneficial portability insights. These insights are derived from our effort to develop a performance portable molecular dynamics (MD) library.

• Targeted platforms:
  – CUDA HPC architectures
    • ORNL’s Summit - IBM POWER9 and NVIDIA Volta GPUs
  – Later will target Intel Xeon Phi HPC architectures
    • Argonne’s Aurora - 3rd Generation Intel Xeon Phi
  – What about future HPC architectures?
What is a Performance Portable Algorithm?

• **Achieves “acceptable” performance** across a variety of HPC architectures

• **Requires “minor” modifications** when porting to novel HPC architectures

• **Design characteristics**
  – A consistent, unified front-end interface for the computational scientist to use
  – Machine-level specifics and optimizations are contained in back-end to facilitate retargeting new architectures
The Molecular Dynamics Algorithm:

Given a system on $N$ atoms with positions $r_1, r_2, \ldots, r_N$ and an interacting potential $V$.

$$ \mathbf{R}(t) = \{ \mathbf{r}_1(t), \mathbf{r}_2(t), \ldots, \mathbf{r}_N(t) \} $$

$$ \mathbf{F}_i(t) = -\frac{\partial}{\partial \mathbf{r}_i} V(\mathbf{R}(t)) $$

Algorithmic Steps

- **Given an starting configuration $\mathbf{R}(t)$ at time $t$**
  $$ \mathbf{R}(t) = \{ \mathbf{r}_1(t), \mathbf{r}_2(t), \ldots, \mathbf{r}_N(t) \} $$

- **Calculate the forces at time $t$**
  $$ \mathbf{F}_i(t) = -\frac{\partial}{\partial \mathbf{r}_i} V(\mathbf{R}(t)) $$

- **Integrate to solve for $\mathbf{R}(t+\Delta t)$**
  $$ \mathbf{r}_N(t + \Delta t) \approx 2\mathbf{r}_N(t) - \mathbf{r}_N(t - \Delta t) + \frac{\Delta t^2}{m_N} \mathbf{F}_N(t) $$

- **Calculate the relevant properties at time $t+\Delta t$**
  - $t+\Delta t \rightarrow t$, $\mathbf{R}(t+\Delta t) \rightarrow \mathbf{R}(t)$, etc.
  - **Repeat**

This step is $\sim 80\%$ of the total computational time
Molecular Dynamics: Non-bonded-Forces Calculation is the Bottleneck

• Long-range electrostatic interactions forces
  – Limited by scaling of the FFTW in the PME calculation
  – Solution: Implement a performance portable long-range electrostatic solver library (Multilevel Summation Method)

• Short-range 2 body (pair-wise) forces
  – Computational complexity is $O(N)$ where $N$ is the number of atoms
  – Solution: Implement a performance portable short range force solver for the Lennard-Jones particle interactions
Preliminaries - What can I realistically do with my given resources?

• One has **performance profiled** the application with near as **close to production status**.
  – Discover computational bottlenecks
  – Discover the dominant data structures within these computational bottlenecks
  – Is the application even suitable for acceleration?
Nonexisting Code

• We choose to write our libraries in C and C++
  – Provides the best chance of compiling and running on many operating systems using various compilers
  – Used a portable subset of C/C++
    • Used C/C++ subset that works well with compiler directive-based accelerator kernels
    • Avoided exotic and experimental features of C/C++
  – Allows for a long-term conversation with the computer science community
    • Profilers, debuggers, and other development tools
Library API?

- **Spatial Decomposition**
  - Electrostatic Solver
    - Input atom charges and positions
    - Returns forces
  - Nonbonded interactions
    - Much more difficult
      - Atom positions, interaction parameters, excluded interactions, modified interactions, etc.
    - Trying to replicate ease of use like FFTW library
Separate the Interface and the Implementation

Direct Algorithmic Step Data Structures

- \( Q \) – simple array of charges on CPU
- \( R \) – simple array of atomic positions on CPU

Data structures on CPU

Direct Kernel

- Accelerator Device Wrapper
- Accelerator Device
  - \( R \) – Positions on device
  - \( Q \) – Charges on device

The Accelerator Device Wrapper is actually an API between the CPU and Accelerator Device

The Accelerator Device is C++ Class allocated on the device which performs the computation on HPC compute device

HPC Compute Device (Summit GPUs)
Accelerator Device

- Accelerator device abstracts the programming model
  - Hand written CUDA kernels
  - In the future, KOKKOS, RAJA
  - 10 years from now, who knows. I just have to satisfy the accelerator device API

- Do’s for CUDA kernels
  - Simple 1D contiguous arrays whose shapes and sizes are mostly known at compile time – elemental data types
  - Classes and structures that contain simple elemental data types
  - Aligned and coalesced accesses
  - Minimize the data transfers between CPU and GPU. It is oppressive!

- Don’ts for CUDA kernels
  - Data structures that involve pointer chasing (e.g. linked lists)
Summary

• Profile application a close to production status as possible
• Separation of algorithm interface from its implementation
  – Program to an interface, not an implementation
• If coding goals and resources permit, try using an accelerated library or OpenACC
  – Other alternatives are KOKKOS, RAJA, etc.
• Avoid exotic data types
  – Simple plain old data
Additional References


Additional Slides
Existing Code

• Try an accelerated CUDA library
  – One may need to restructure data to a form amenable to the library

• Try a directive based approach like OpenACC
  – One may need to restructure data to a form amenable to the GPU execution model

• Be forewarned!
  – Data restructuring may then become the new bottleneck
  – Strongly advise significant code refactorization with respect to its data structures
    • Generally results in better CPU performance too
  – Be aware of data reuse on the GPU to minimize CPU↔GPU transfer costs
Overview of GPU Programming Model

The threads within a thread block are grouped in execution units of 32 threads called a warp – single instruction multiple data (SIMD).

Warp of 32 threads

Warp of 32 threads

0 1 ...

N-1

float y = input_array[threadId];
float area = y*y;
output_array[threadId] = area;

CUDA kernel
The Molecular Dynamics Algorithm:

Given a system on $N$ atoms with positions $r_1, r_2, \ldots, r_N$ and an interacting potential $V$.

Integrate Newton’s law (by finite difference methods)

\[
\begin{align*}
    m_1 \frac{d^2}{dt^2} \vec{r}_1 &= \vec{F}_1 \\
    m_2 \frac{d^2}{dt^2} \vec{r}_2 &= \vec{F}_2 \\
    \vdots & \quad \vdots \\
    m_N \frac{d^2}{dt^2} \vec{r}_N &= \vec{F}_N
\end{align*}
\]

Equations of motion

\[
\begin{align*}
    \vec{r}_1(t + \Delta t) &\approx 2\vec{r}_1(t) - \vec{r}_1(t - \Delta t) + \frac{\Delta t^2}{m_1} \vec{F}_1(t) \\
    \vec{r}_2(t + \Delta t) &\approx 2\vec{r}_2(t) - \vec{r}_2(t - \Delta t) + \frac{\Delta t^2}{m_2} \vec{F}_2(t) \\
    \vdots & \quad \vdots \\
    \vec{r}_N(t + \Delta t) &\approx 2\vec{r}_N(t) - \vec{r}_N(t - \Delta t) + \frac{\Delta t^2}{m_N} \vec{F}_N(t)
\end{align*}
\]
GPU Programming Data Structures/Algorithm Major Requirements

• Minimize the data transfers between CPU and GPU. It is oppressive!
• Memory Accesses Need To Be Aligned!
• Memory Accesses Need To Be Contiguous!
What about Compiling?

• How do we plan to ensure that one can compile a reasonably performant code across various HPC architectures?
Separate the Interface and the Implementation

Direct Algorithmic Step Data Structures

- $Q$ – simple array of charges on CPU
- $R$ – simple array of atomic positions on CPU

Data structures on CPU

Direct Kernel

Accelerator Device Wrapper

Accelerator Device

$KLP$ – Kernel Launch Parameters

The Accelerator Device Wrapper is actually an API between the CPU and Accelerator Device

The Accelerator Device contains a class, $KLP$, that controls the various parameters that control the kernel's performance.

HPC Compute Device (Summit GPUs)