Implementation of NAMD molecular dynamics non-bonded force-field on the Cell Broadband Engine processor

Guochun Shi
Volodymyr Kindratenko

National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
Presentation outline

• Introduction
  • Cell Broadband Engine
  • NAno Molecule Dynamics Simulation (NAMD)

• Implementation
  • Task library and task dispatch system
  • SIMD and code optimizations for SPU

• Performance
  • Static analysis
  • Performance on hardware

• Conclusions
Cell Broadband Engine

- One Power Processor Element (PPE) and eight Synergistic Processing Elements (SPE), each SPE has 256 KB local storage
- 3.2 GHz processor
- 25.6 GB/s processor-to-memory bandwidth
- > 200 GB/s EIB sustained aggregate bandwidth
- Theoretical peak performance of 204.8 GFLOPS (SP) and 14.63 GFLOPS (DP)
NAMD

- Compute forces and update positions repeatedly
- The simulation space is divided into rectangular regions called patches
  - Patch dimensions > cutoff radius for non-bonded interaction
- Each patch only needs to be checked against nearby patches
  - Self-compute and pair-compute
NAMD kernel

NAMD SPEC 2006 CPU benchmark kernel

1: for each atom \( i \) in patch \( p_k \)
2:     for each atom \( j \) in patch \( p_l \)
3:         if atoms \( i \) and \( j \) are bonded, compute bonded forces
4:             otherwise, if atoms \( i \) and \( j \) are within the cutoff distance, add atom \( j \) to the \( i \)'s atom pair list
5:         end
6:     for each atom \( k \) in the \( i \)'s atom pair list
7:         compute non-bonded forces (L-J potential and PME direct sum, both via lookup tables)
8:     end
9: end

We implemented a simplified version of the kernel that excludes pairlists and bonded forces

1: for each atom \( i \) in patch \( p_k \)
2:     for each atom \( j \) in patch \( p_l \)
3:         if atoms \( i \) and \( j \) are within the cutoff distance
4:             compute non-bonded forces (L-J potential and PME direct sum, both via lookup tables)
5:         end
6: end
### Implementation: task library and dispatch system

**Compute task struct**

```c
typedef struct task_s {
    int cmd; // operand
    int size; // the size of task structure
} task_t;
```

```c
typedef struct compute_task_s {
    // function to find idle SPE
    task_t common;
    <user_type1> <user_var_name1>
    <user_type2> <user_var_name2>
    ...  
} compute_task_t;
```

### API for PPE and SPE

```c
int ppu_task_init(int argc, char **argv, spe_program_handle_t);
// initialization

int ppu_task_run(volatile task_t * task);
// start a task in all SPEs

int ppu_task_spu_run(volatile task_t * task, int spe);
// start a task in one SPE

int ppu_task_spu_wait(void);
// wait for any SPE to finish, blocking call

void ppu_task_spu_waitall(void);
// wait for all SPEs to finish, blocking all
```

```c
int spu_task_init(unsigned long long);

int spu_task_register(dotask_t, int);
// register a task

int spu_task_run(void);
// start the infinite loop, wait for tasks
```

### Task dispatch system

1. **start**
2. **find idle SPE**
   - no: **ppu_task_spe_wait()**
   - yes: **find a dependency-free patch pair**
      - no: start again
      - yes: **remove patch pair from the pool**
         - yes: **run pair-compute task on the idle SPE**
         - no: wait until all SPU tasks exit
3. **pool of patch pairs to be processed**
4. **pool of idle SPEs**
5. **wait until all SPU tasks exit**
6. **stop**
Implementation: SPU

- SIMD: each component is kept in a separate vector
- Data movement dominates the time
- Buffer size carefully chosen to fit into the local store

**Local store usage**

<table>
<thead>
<tr>
<th></th>
<th>Code (KB)</th>
<th>L-J table (KB)</th>
<th>Table_four (KB)</th>
<th>Atom_buffer (KB)</th>
<th>Force buffer (KB)</th>
<th>Stack others</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>25</td>
<td>55</td>
<td>45</td>
<td>30</td>
<td>18</td>
<td>83</td>
</tr>
<tr>
<td>DP</td>
<td>25</td>
<td>55</td>
<td>91</td>
<td>48</td>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>
Implementation: optimizations

- Different vectorization schemes are applied in order to get best performance
  - Self-compute: do redundant computations and fill zeros to unused slots
  - Pair-compute: save enough pairs of atoms, then do calculations
Performance: static analysis

- Distance computation code takes most of the time
- Data manipulation time is significant
Performance

NAMD kernel performance on different architectures

- 13.4x speedup for SP and 11.6x speedup for DP compared to a 3.0 GHz Intel Xeon processor
- SP performance is < 2x better than DP

Scaling and speedup of the force-field kernel as compared to a 3.0 GHz Intel Xeon processor
Conclusions

• Linear speedup when using multiple synergistic processing units

• Performance of the double-precision floating-point kernel differs by less than a factor of two from the performance of the single-precision floating-point kernel
  • Even though the peak performance of the Cell’s single-precision floating point SIMD engine is 14 times the peak performance of the double-precision floating-point SIMD engine

• The biggest challenge in using the Cell/B.E. processor in scientific computing applications, such as NAMD, is the software development complexity due to the underlying hardware architecture
Acknowledgment

• NSF grant SCI 05-25308
• IBM Linux Technology Center
• Dr. James Phillips from the Theoretical and Computational Biophysics Group
• Access to Cell blades was provided by
  • Prof. Paul Woodward from the University of Minnesota
  • Prof. Dave Bader from the Georgia Institute of Technology
• Trish Barker, Jeremy Enos, and Dr. John Larson from NCSA