A low memory footprint OpenCL simulation of short-range particle interactions

Raymond Namyst

With Samuel Pitoiset, Inria and Emmanuel Cieren, Laurent Colombet, Laurent Soulard, CEA/DAM/DPTA

STORM
INRIA Group
Context

Short range N-body simulation

- Joint work with CEA/DAM/DPTA
- N-body problem
  - Short-range interactions
    - Forces are neglected if
      \[ \text{dist}(p_1, p_2) > r_{\text{cut}} \]
- Goals
  - Develop Exascale-ready version of the Stamp MD application
    - Multiple potentials: EAM, MEAM
    - Coulombic interactions: Ewald summation, Fast Multipoles Method
  - Support of multiple accelerators
  - Simulate hundreds of billions of atoms
    - Verify simulation results with real (nano-scale) experiments

Temperature map and melting zone (empty pores)  
Graphite/diamond transition (carbon inclusion)
Atomic simulation of materials

Interactive OpenGL + OpenCL app
Evolution of Parallel Hardware

Exascale Parallel Machines

- SuperComputers expected to reach Exascale \((10^{18} \text{ flop/s})\) by 2020

- From the programmer point of view, the biggest change will come from node architecture
  - High number of cores
  - Powerful SIMD units
  - Hybrid systems

- Extreme parallelism
  - Total system concurrency is estimated to reach \(O(10^9)\)
  - Embarrassingly parallel hardware
    - Do we really want to assign different tasks to each individual computing unit?

- Memory per single computing unit is decreasing
Context

- Joint work with CEA/DAM/DPTA

- New application (ExaStamp) is developed in C/C++
  - Based on SOTL library that exploits multiple accelerators on a single node
    - OpenCL 1.2
    - Focus on low memory footprint implementation
OpenCL programming model

Khronos group, 2008

• OpenCL is supported by majors processors/accelerators vendors
  – NVIDIA, AMD, Intel, IBM

• Initially influenced by GPU execution models
  – Later implemented on different architectures, including CPUs

• OpenCL merits
  – Forces programmers to expose massive parallelism
  – Reflects constraints of manycore architectures
    • No global synchronization
    • Need to feed large vectorization units/numerous HW threads
OpenCL programming model

- The same kernel is executed by many *work items*
  - Domain dimensions
    - 2D
    - #work items along each dim
      - 24 x 24
    - Group size along each dim
      - 8 x 8

Kernel execution

- `get_global_id(0) = 6`
- `get_global_id(1) = 10`
OpenCL programming model

Example

• “ScalVec” kernel
  – Vector “vec” lies in accelerator’s global memory
  – We use one work item per vector element

```c
__kernel void ScalVec(__global float *vec, float k) {
    int index = get_global_id(0);

    vec[index] *= k;
}
```
NVIDIA GPU Execution Model

- Streaming processor
  - Interleaved execution of sequential hardware threads
  - Context switch is free
    - Avoid stalling on memory load
- Streaming multiprocessor
  - Hosts **groups** of HW threads
    - Local memory sharing
    - Synchronization
- Global memory is shared by all streaming multiprocessors
NVIDIA GPU Execution Model

• Several OpenCL workgroups can reside on the same streaming multiprocessor
  – Limited by hardware resources
    • Registers
    • Max HW threads per SP
    • Local Memory

• Shared local memory
  – Much faster than global memory
  – Only a few kBytes!
NVIDIA GPU Execution Model

- Threads are implicitly grouped in warps
  - 32 threads
  - All threads of the same warp execute the same instruction at the same logical cycle
    • No divergence!

- The hardware tries to coalesce memory accesses inside half-warps
  - Aligned and contiguous (wrt half-warps) memory accesses must be preferred
Xeon Phi execution model

Implicit Vectorization

• The OpenCL runtime system spawns 240 OS threads
  - OS threads pinned on each core

• OpenCL workgroups are dispatched among threads
  - Each workgroup is executed sequentially by one thread
    • At least 240 workgroups are needed to feed all cores

• Kernels are implicitly vectorized along dimension 0
  - Work items are grouped to form get_local_size(0)/16 vectors

```c
__Kernel void foo(...)
    For (int i = 0; i < get_local_size(2); i++)
        For (int j = 0; j < get_local_size(1); j++)
            For (int k = 0; k < get_local_size(0); k += VECTOR_SIZE)
                Vectorized_Kernel_Body;
```
Xeon Phi execution model

Code divergence within workgroups

• Conditional code is not harmful when all work items (within a WG) are guaranteed to execute the same branch
  - If(get_local_id(1) == y) foo();

• In other cases, code has to be “predicated” and both IF & ELSE parts are executed for all work items

```c
If(get_global_id(0) % n == 0) 
    res = IF_code();
Else 
    res = ELSE_code();
```

```c
gid16 = get16_global_id(0);
Mask = compare16int((gid % broadcast16(32)), 0)
Res_if = IF_code();
Res_else = ELSE_code();
Res = (res_if & mask) | (res_else & not(mask));
```
A low memory footprint Lennard-Jones kernel

\[ F_{ij} = \begin{cases} 
24 \frac{\epsilon}{r} \left( \left( \frac{\sigma}{r} \right)^6 - \left( \frac{\sigma}{r} \right)^{12} \right) & \text{if } r \leq r_c \\
0 & \text{otherwise.} 
\end{cases} \]  

(1)

where \( r \) is the distance between \( i \) and \( j \). \( \sigma \) and \( \epsilon \) are constants chosen to fit the physical properties of the simulated material. When the distance between a pair of atoms is greater than the cut-off radius \( r_c \), forces are neglected.

\[ \vec{F}_{i*} = \sum_{j \neq i} F_{ij} \hat{u}_{ij} \text{ with } \hat{u}_{ij} = \frac{\vec{i}_{ij}}{r} \] 

(2)
A low memory footprint Lennard-Jones kernel

Data structures

Coordinates

\[
\begin{align*}
&x \\
&y \\
&z
\end{align*}
\]

\[
\begin{align*}
x_0 & \quad x_1 & \quad x_2 & \quad \ldots & \quad x_{N-1} \\
y_0 & \quad y_1 & \quad y_2 & \quad \ldots & \quad y_{N-1} \\
z_0 & \quad z_1 & \quad z_2 & \quad \ldots & \quad z_{N-1}
\end{align*}
\]

N = # atoms

Structures of Arrays (SoA)

\[
\text{ROUND}(N)
\]
A low memory footprint Lennard-Jones kernel

Splitting the domain into cubic boxes

• Short-range interactions
  – Forces are neglected beyond a distance of $R_{\text{cut}}$

• Dimension of boxes
  – $R_{\text{cut}} \times R_{\text{cut}} \times R_{\text{cut}}$
  – At most, 27 boxes explored to find neighbors
A low memory footprint Lennard-Jones kernel

Data structures

Coordinates

Boxes

(For the sake of simplicity, we only depict a compact “array of atoms” in the remaining of the presentation)
A low memory footprint Lennard-Jones kernel

Sorting atoms per bounding box

• Bucket-sort
  – Two phases (ie. Kernels)
    • Prefix Sum
      How many atoms in each box?
    • Sorting the array of atoms

![Diagram showing bucket-sort and prefix sum](image-url)
A low memory footprint Lennard-Jones kernel

Parallelization strategy

• We spawn one thread per atom
  – Workgroup size = TILE

• Each workgroup “covers” a variable number of boxes
  – Depends on material density

• Work on contiguous atoms
  – Along x axis

9 atoms spread across 7 boxes
A low memory footprint Lennard-Jones kernel

Parallelization strategy

- Nine steps to compute forces with neighbors
  - Always load contiguous neighbors (x axis)

- At each step, we have to load the contents of N+2 boxes
  - May involve several steps
A low memory footprint Lennard-Jones kernel

Parallelization strategy

• Nine steps to compute forces with neighbors
  – Always load contiguous neighbors (x axis)

• At each step, we have to load the contents of N+2 boxes
  – May involve several steps
A low memory footprint Lennard-Jones kernel

Parallelization strategy

• Nine steps to compute forces with neighbors
  – Always load contiguous neighbors (x axis)

• At each step, we have to load the contents of N+2 boxes
  – May involve several steps
A low memory footprint Lennard-Jones kernel

Variants

- Differences between GPU and Xeon Phi code
  - No Tiling in local shared memory on Xeon Phi
    - No physical scratchpad memory, so shared data is allocated in global memory!
  - Less data alignment constraints on Xeon Phi
  - No need to focus on local memory bank conflicts on Xeon Phi

- Code divergence within workgroups
  - Maybe costly on GPU
  - Can prevent successful vectorization on Intel Xeon Phi
Performance Results

NVIDIA K20C

Throughput and Memory Footprint vs. Number of Atoms
Performance Results

Intel Xeon Phi 5110

- Throughput (#atoms/s)
- Mem. footprint (MB)
- #atoms

Throughput (simple) - solid blue line
Mem. footprintmémoire (simple) - dashed blue line
Throughput (double) - solid green line
Mem. footprint (double) - dashed green line
Performance Results

![Bar chart showing performance results for different kernels and configurations.](image)

- **Kernel**:
  - overhead
  - update pos
  - force
  - box sort
  - box copy
  - box scan
  - box count
  - box reset

- **Time (us) per iteration for 1M atoms**
  - M2075, simple
  - K20c, simple
  - Xeon, simple
  - Phi, simple
  - M2075, double
  - K20c, double
  - Xeon, double
  - Phi, double

The chart compares the performance of different kernels and configurations, with the Y-axis representing time in microseconds (us) and the X-axis representing different configurations and kernels.
Performance Results

Kernel
- overhead
- update pos
- force
- box sort
- box copy
- box scan
- box count
- box count
- reset

% of total time

M2070: simple
K20c: simple
Xeon: simple
Phi: simple
M2070: double
K20c: double
Xeon: double
Phi: double
Performance Results

Multi-accelerators configuration (K20C + Xeon)
Integration into ExaStamp

Object-oriented software architecture

Diagram:
- Node
- Domain
- Grid
- Array<Cell>
- Comm. Manager
- Integration Scheme
- MPI
- Node
- Domain - Grid
- Cell
Integration into ExaStamp

Object-oriented software architecture

- Vectorized code derived from single potential definition

```c
void operator () {
    double *ep_i,
    *fx_i, *fy_i, *fz_i,
    *rx_i, *ry_i, *rz_i;

    vector_t t0, t1, t2, t3, t4, t5;

    t0.load (rx_i);
    t1.load (ry_i);
    t2.load (rz_i);

    t3 = inv(t0*t0 + t1*t1 + t2*t2);
    t4 = t3 * _sigma2;
    t5 = t4 * t4 * t4;
    t4 = t5 * t5;

    t5 = _2epsilon * t5;
    t4 = _24epsilon * t4 * t3;

    t0 = t0 * t4;
    t1 = t1 * t4;
    t2 = t2 * t4;

    t0.store (fx_i);
    t1.store (fy_i);
    t2.store (fz_i);
    t5.store (ep_i);
}
```

\[
t_5 = 2 \varepsilon \left[ \left( \frac{\sigma}{\| \mathbf{r}_i \|} \right)^6 - \left( \frac{\sigma}{\| \mathbf{r}_i \|} \right)^6 \right]
\]

\[
t_4 = 24 \varepsilon \left[ 2 \left( \frac{\sigma}{\| \mathbf{r}_i \|} \right)^6 - \left( \frac{\sigma}{\| \mathbf{r}_i \|} \right)^6 \right] \frac{1}{\| \mathbf{r}_i \|^2}
\]

Flags used to select right intrinsics instructions (at compile time):

(a) `<no flag>`
(b) `__vectorize_sse`
(c) `__vectorize_avx`
(d) `__vectorize_mic`
Integration into ExaStamp

Preliminary performance results

Efficiency with 65 millions atoms on 512 cores (Curie @ TGCC), EAM potential

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>Efficiency</th>
<th>Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>89.5%</td>
<td>-25%</td>
</tr>
<tr>
<td>128</td>
<td>91.4%</td>
<td></td>
</tr>
</tbody>
</table>

Grain time for 1.8 millions atoms on 2 Haswell sockets (TERA 1000), EAM potential

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>Time (s)</th>
<th>Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>1.60967e-05</td>
<td>-33%</td>
</tr>
<tr>
<td>24</td>
<td>1.22596e-05</td>
<td></td>
</tr>
</tbody>
</table>
Current status

- Preliminary OpenCL implementation works quite well
  - More than 90M atoms on accelerators equipped with 5GB RAM
  - Single-precision Lennard-Jones potential
    - More than 20 millions atoms/s on modern GPUs,
      more than 12 millions atoms/s on Intel Xeon Phi
  - No GPU specific tricks
    - Texture memory
  - Low memory consumption
    - Can cope with a high number of particles as well as low density materials
  - Integrated into ExaStamp [MuCoCos 2014]

- Much potential (and ideas) for improvement
  - Several optimizations are still applicable
    - But will impact code complexity
  - OpenCL kernels slightly differ from one arch to another
  - Tuning (workgroup size, occupancy) is painful
What Programming Model for Exascale?

Software @ Exascale

- Millions of tasks/threads to occupy hundreds of thousands of cores
  - Express massive parallelism
  - Reuse and compose existing codes/algorithms

- Portability AND Efficiency over wide range of processors and accelerators
  - Ability to cope with multiple variants of code
  - Autotuning

- Ability to automatically exploit multiple heterogeneous computing units
  - Load balancing
  - Data movements, prefetching
  - Synchronizations
We need to push a new programming framework

1. Manycore oriented Programming Model

2. True cooperation between Compilation Tools and Runtime Systems
OpenCL forms an excellent basis for 1) Manycore oriented programming model

- OpenCL is a standard [Khronos 2008] which meets some of the aforementioned requirements
  - Efficiency
  - Portability

- OpenCL is a good vehicle for expressing extreme parallelism
  - Start from pure, extreme parallelism… then map over a restricted set of resources
    E.g. cores, vector units, streaming processors
A New OpenCL-based programming framework

Tight integration between compilation tools and runtime systems

• OpenCL alone is far from meeting Exascale applications needs!
  – No performance portability
    • Programs are optimized with a target architecture in mind
  – No provision for transparent multi-accelerator programming
  – No provision for adaptive parallelism

• We propose to build a Framework on the OpenCL ecosystem
  – OpenCL language extensions
    • Higher-level programming
  – OpenCL code generation tools
    • Deal with multiple variants of kernels, parameters autotuning
  – Powerful runtime system mechanisms
    • Dynamic kernel splitting, load balancing