Improving performance of the N-Body problem

Efim Sergeev

Senior Software Engineer at Singularis
Lab LLC
Contents

• Theory
• Naive version
• Memory layout optimization
• Cache Blocking Techniques
• Data Alignment to Assist Vectorization
Given $N$ bodies with initial position $\mathbf{x}_i$ and velocity $\mathbf{v}_i$, the force vector $\mathbf{f}_{ij}$ on body $i$ caused by its gravitational attraction to body $i$ is given by following:

$$
\mathbf{f}_{ij} = G \frac{m_i m_j}{||\mathbf{r}_{ij}||^2} \cdot \frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||}
$$

- where $m_i$ and $m_j$ are masses of bodies;
- $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$ is the vector from body $i$ to $j$;
- $G$ is the gravitational constant.
The total force $F_i$ on body $i$, due to its interactions with the other $N-1$ bodies, is obtained by summing all interactions:

$$F_i = \sum_{1 \leq j \leq N \atop j \neq i} f_{ij} = Gm_i \cdot \sum_{1 \leq i \leq N \atop j \neq i} \frac{m_i r_{ij}}{\|r_{ij}\|^3}$$

- **Advantages**: robust, accurate, completely general.
- **Disadvantage**: computational cost per body is $O(N)$, need $O(N^2)$ operations to compute forces on all bodies.
- **However, direct summation is a good fit with**:
  1. Individual time steps;
  2. Specialized hardware;
typedef struct { float x, y, z, vx, vy, vz; } Body;

void bodyForce(Body *p, float dt, int n) {
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < n; i++) {
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

        for (int j = 0; j < n; j++) {
            float dx = p[j].x - p[i].x;
            float dy = p[j].y - p[i].y;
            float dz = p[j].z - p[i].z;
            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
        }

        p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;
    }
}
Naive version compile & run

- Allocate resources
  - `#salloc -N 1 --partition=mic`
  - `#squeue`

- Connect to node
  - `#ssh %nodename%`
  - `#cd ./Workshop2016/src/mini-nbody`

- Compile
  - `#./build-nbody-naive.sh`

- Run
  - `# ./run-nbody-naive.sh`

<table>
<thead>
<tr>
<th>Number of bodies</th>
<th>Interaction per second $10^9$</th>
<th>naive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4069</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>16384</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>32768</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>65536</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>131072</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>262144</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>524288</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1048576</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
• 60(61) cores
• 4 HW threads/core
• 8 memory controllers 16 Channel GDDR5
• High-speed bi-directional ring interconnect Fully Coherent L2 Cache
Memory Hierarchy for Intel® Xeon Phi™ Core Architecture

- 4 Threads per Core
- 64-bit
- 512-wide
- VPU: Integer, SP, DP, 3-operand, 16-instruction
- 32KB per core
- 512 per core
- Fully Coherent
Memory Hierarchy for Intel® Xeon Phi™

Memory hierarchy

- CPU core and Vector registers
  - 32x64 bytes, 1 cycle
  - 512 avx register, 1 cycle

- L1 Data Cache
  - 32 KB, 3 cycles

- L1 Instruction Cache
  - 32 KB, 3 cycles

- L2 Cache
  - 512 KB, 11 cycles

- Interconnected

- Main Memory
  - 8-16GB, 100+ cycles
Memory Layout Transformations: Array of Structures (AoS)

Array of structures:

```c
struct {
    float x, y, z, vx, vy, vz;
} Body;
```

Array of structures in RAM looks something like following:

```
x0  y0  z0  vx0  vy0  vz0  x1  y1  z1  vx1
```

Disadvantages:

- Use memory gather-scatter operations in vectorized code
- Performing SIMD operations on the original AoS format can require more calculations
- Difficult optimize by compiler
Memory Layout Transformations: Structures of Arrays (SoA)

Structure of array:

```c
struct {
} BodySystem;
```

Array of structures in RAM looks something like following:

```
x0  x1  x2  x3  x3  ...
y0  y1  y2  y3  y4  ...
z0  z1  z2  z3  z4  ...
```

Advantages:

- Keep memory accesses contiguous when vectorization is performed
- SoA arrangement allows more efficient use of the parallelism of the SIMD technologies because the data is ready for computation in a more optimal vertical manner
- More efficient use of caches and bandwidth
Memory Layout Transformations: Loading in AVX(registers)

Memory: array of structure

<table>
<thead>
<tr>
<th>x0</th>
<th>y0</th>
<th>z0</th>
<th>vx0</th>
<th>vy0</th>
<th>vz0</th>
<th>x1</th>
<th>y1</th>
<th>z1</th>
<th>vx1</th>
</tr>
</thead>
</table>

AVX/SIMD registers

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y0</td>
<td>y1</td>
<td>y2</td>
<td>y3</td>
<td>y4</td>
</tr>
<tr>
<td>z0</td>
<td>z1</td>
<td>z2</td>
<td>z3</td>
<td>z4</td>
</tr>
</tbody>
</table>

Memory: structure of array

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y0</td>
<td>y1</td>
<td>y2</td>
<td>y3</td>
<td>y4</td>
</tr>
<tr>
<td>z0</td>
<td>z1</td>
<td>z2</td>
<td>z3</td>
<td>z4</td>
</tr>
</tbody>
</table>

AVX/SIMD registers

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y0</td>
<td>y1</td>
<td>y2</td>
<td>y3</td>
<td>y4</td>
</tr>
<tr>
<td>z0</td>
<td>z1</td>
<td>z2</td>
<td>z3</td>
<td>z4</td>
</tr>
</tbody>
</table>

void bodyForce(BodySystem p, float dt, int n) {
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < n; i++) {
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

        for (int j = 0; j < n; j++) {
            float dy = p.y[j] - p.y[i];
            float dz = p.z[j] - p.z[i];
            float dx = p.x[j] - p.x[i];
            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
        }

        p.vx[i] += dt*Fx; p.vy[i] += dt*Fy; p.vz[i] += dt*Fz;
    }
}
SoA version compile & run

• Compile
  – ../../../build-nbody-soa.sh

• Run
  – ../../../run-nbody-soa.sh
SoA optimization results

Interaction per second

10^9

Number of bodies

0 5 10 15 20 25 30 35 40

1024 2048 4069 8192 16384 32768 65536 131072 262144 524288 1048576

naive SoA
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        float dy = p.y[j] - p.y[i];
        float dz = p.z[j] - p.z[i];
        float dx = p.x[j] - p.x[i];
    }
}

Cache reloaded for each i body after every tile iteration by j

Cache L2 512KB

- p.x[0:300] → p.x[300:600] → p.x[600:900]
- p.z[0:300] → p.z[300:600] → p.z[600:900]
Cache Blocking Techniques: Fit bodies in the cache

- L2 cache size 512 KB
- Each body requires in memory 24 bytes
  - 6 fields by 4 byte, total 24 bytes
- How many bodies can be placed in L2 cache?
  - 512000 / 24 = 21300 bodies
Cache Blocking Techniques: Fit bodies in the cache

Original source:

```c
for (body1 = 0; body1 < NBODIES; body1++) {
    for (body2 = 0; body2 < NBODIES; body2++) {
        OUT[body1] += compute(body1, body2);
    }
}
```

Modified Source (with 1-D blocking):

```c
for (body2 = 0; body2 < NBODIES; body2 += BLOCK) {
    for (body1 = 0; body1 < NBODIES; body1++) {
        for (body22 = 0; body22 < BLOCK; body22++) {
            OUT[body1] += compute(body1, body2 + body22);
        }
    }
}
```
Cache Blocking Techniques: Result code

```c
void bodyForce(BodySystem p, float dt, int n, int tileSize) {
    for (int tile = 0; tile < n; tile += tileSize) {
        int to = tile + tileSize;
        if (to > n) to = n;

        #pragma omp parallel for schedule(dynamic)
        for (int i = 0; i < n; i++) {
            float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

            for (int j = tile; j < to; j++) {
                float dy = p.y[j] - p.y[i];
                float dz = p.z[j] - p.z[i];
                float dx = p.x[j] - p.x[i];
                float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
                float invDist = 1.0f / sqrtf(distSqr);
                float invDist3 = invDist * invDist * invDist;

                Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
            }
            p.vx[i] += dt*Fx; p.vy[i] += dt*Fy; p.vz[i] += dt*Fz;
        }
    }
}
```

Outer cycle by blocks
Cache optimization version compile & run

• Compile
  – `./build-nbody-block.sh`

• Run
  – `./run-nbody-block.sh`
Cache optimization results

Interaction per second: $10^9$

Number of bodies

- naive
- SoA
- Cache
Reason for Vectorization Fails & How to Succeed

• **Alignment:**
  • Align your arrays/data structures

• Most frequent reason is **dependence:**
  • Minimize dependencies among iteration

• **Non-unit stride between elements:**
  • Possible to change algorithm to allow linear/consecutive access?
Memory alignment

• All memory-based instructions in the coprocessor should be aligned to avoid instruction faults
  – Force the compiler to create data objects with starting addresses that are modulo 64 bytes.

• The compiler is able to make optimizations when the data access (including base-pointer plus index) is known to be aligned by 64 bytes
  – Compiler cannot know nor assume data is aligned inside loops without some help from the user
Memory alignment

L2 cache
composed of 64 byte cache lines

64 byte cache lines

L2 CACHE

RAM read/write alignment

0 bytes 64 bytes 128 bytes
Memory alignment

Array in memory not alignment to 64 bytes

L2 CACHE

DRAM

Array in memory loading to cache in 3 operations
Memory alignment

Array in memory not alignment to 64 bytes

L2 CACHE

Array in L2 CACHE loading to AVX register in 3 operations

every second load will be across a cache-line split

64 byte cache lines

Array in memory not alignment to 64 bytes

L2 CACHE

AVX

511  382  255  127  0

x7  x6  x5  x4  x3  x2  x1  x0

Array in L2 CACHE loading to AVX register in 3 operations
Memory alignment

Array in memory alignment to 64 bytes

L2 CACHE

DRAM

Array in memory loading to cache in 2 operations

64 byte cache lines

0 bytes

64 bytes

128 bytes
Memory alignment

Array in memory alignment to 64 bytes

L2 CACHE

64 byte cache lines

Array in L2 CACHE loading to AVX register in 2 operations
**SIMD datatypes**

- **Intel® AVX 512:**
  - Vector: 512 bit
  - Data types: 32 and 64 bit integers, 32 and 64 bit floats
  - VL: 8, 16

<table>
<thead>
<tr>
<th></th>
<th>511</th>
<th>382</th>
<th>255</th>
<th>127</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x7</td>
<td>x6</td>
<td>x5</td>
<td>x4</td>
<td>x3</td>
</tr>
<tr>
<td></td>
<td>y7</td>
<td>y6</td>
<td>y5</td>
<td>y4</td>
<td>y3</td>
</tr>
<tr>
<td></td>
<td>x7opy7</td>
<td>x6opy6</td>
<td>x5opy5</td>
<td>x4opy4</td>
<td>x3opy3</td>
</tr>
<tr>
<td></td>
<td>x2opy2</td>
<td>X1opy1</td>
<td>x0opy0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Values for VL:
- 511: 0
- 382: 1
- 255: 2
- 127: 3
- 0: 4
SIMD vectorization

- Usage of AVX vector operations for data processing can be achieved by:
  - Manual
  - Automatically by compiler
    - Minimize dependencies among iteration

```c
for (int i = 0; i < n; i++)
  float dy = p.y[j] - p.y[i];
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Difference between IVDEP & SIMD pragmas 

```c
void foo(int *a, int k, int c, int m) {
    #pragma ivdep
    for (int i = 0; i < m; i++)
        a[i] = a[i + k] * c;
}
```

```c
#pragma simd
```

• Aggressive version of IVDEP: Ignore **all** dependencies inside a loop
• It’s an imperative that forces the compiler try everything to vectorize
• Efficiency heuristic is ignored
• It can **vectorize** code legally in some cases that wouldn’t be possible:
SIMD vectorization: #pragma simd

Example: without #pragma simd

```c
void add_floats(float *a, float *b, float *c, float *d, float *e, int n) {
    int i; for (i=0; i<n; i++) {
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
    }
}
```

icc example1.c -nologo -Qvec-report2 example1.c
~\example1.c(3): (col. 3) remark: loop was not vectorized: existence of vector dependence.

Example: with #pragma simd

```c
void add_floats(float *a, float *b, float *c, float *d, float *e, int n) {
    int i;
    #pragma simd
    for (i=0; i<n; i++) {
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
    }
}
```

icc example1.c -nologo -Qvec-report2 example1.c
~\example1.c(4): (col. 3) remark: SIMD LOOP WAS VECTORIZED.
void bodyForce(BodySystem p, float dt, int n, int tileSize) {
    for (int tile = 0; tile < n; tile += tileSize) {
        int to = tile + tileSize;
        if (to > n) to = n;

        #pragma omp parallel for schedule(dynamic)
        for (int i = 0; i < n; i++) {
            float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;
            #pragma vector aligned
            #pragma simd
            for (int j = tile; j < to; j++) {
                float dy = p.y[j] - p.y[i];
                float dz = p.z[j] - p.z[i];
                float dx = p.x[j] - p.x[i];
                float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
                float invDist = 1.0f / sqrtf(distSqr);
                float invDist3 = invDist * invDist * invDist;

                Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
            }
            p.vx[i] += dt*Fx; p.vy[i] += dt*Fy; p.vz[i] += dt*Fz;
        }
    }
}
Memory alignment and SIMD vectorization: compile & run

• Compile
  – `./build-nbody-align.sh`

• Run
  – `./run-nbody-align.sh`
Memory alignment and SIMD vectorization: performance results

Interaction per second
10^9

Number of bodies

1024 2048 4096 8192 16384 32768 65536 131072 262144 524288 1048576

naive SoA Cache Aligned
SINGULARIS LAB

software development, computer vision and robotics