To SIMD or not to SIMD, or how to SIMD?

Jun Makino
RIKEN Advanced Institute for Computational Science
Exascale Computing Project
Co-Design Team
Overview

- Problems with “wide SIMD” execution units of modern microprocessors
- Particle-Particle interaction
- SIMD in the outermost loop.
- Performance
- Summary
Problems with “wide SIMD” execution units

- What is a (wide) SIMD unit in modern microprocessors?
- Why is it problematic?
What is a (wide) SIMD unit in modern microprocessors?

Examples:

- 512-bit SIMD on KNC
- 256-bit SIMD (AVX2?) on Haswell
- 256-bit HPC-ACE2 on Fujitsu SPARC64 XIfx

How it works:

- Basic idea is quite simple: each “word” of data registers contain four (256-bit) or eight (512-bit) DP floating-point numbers, or twice of that of SP numbers.
- multiple FPUs operate in parallel on multiple DP or SP words in a single register → high peak FP performance.

Sounds simple? Well...
Why is it problematic?

Simply because a naive implementation of a wide SIMD architecture and instruction set would be almost impossible to use.

From the point of view of a hardware designer, a simple (and thus natural) SIMD architecture would be able to do only

- “aligned” memory access
- element-wise SIMD add/sub/mult

and nothing else.
What “simple” SIMD units cannot do

- unaligned memory access $a[i] = b[i] + b[i+1]$;
- stride memory access $a[i] = b[i*3]$;
- indirect memory access $a[i] = b[index[i]]$;
- permutation within registers
- horizontal addition (necessary for summation $\text{sum} += a[i]$);
- conditional execution (store)

All of them could be done with reasonable efficiency on vector machines of 80-90s.
Comparison of vector arch and SIMD arch

<table>
<thead>
<tr>
<th></th>
<th>Vector</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligned access</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Conditional</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Unaligned access</td>
<td>OK</td>
<td>can do</td>
</tr>
<tr>
<td>Stride access</td>
<td>OK</td>
<td>slow</td>
</tr>
<tr>
<td>Indirect access</td>
<td>OK</td>
<td>very slow</td>
</tr>
<tr>
<td>Permutation</td>
<td>no need</td>
<td>can do</td>
</tr>
<tr>
<td>Horizontal</td>
<td>easy</td>
<td>hard</td>
</tr>
</tbody>
</table>

Most of operations introduced as “Lessons learned from Cray-1” are either slow, very slow, or difficult to implement.
By the way, why are they slow and/or difficult?

Why are things that used to be easy on vector machines so hard on modern microprocessors?

- **Short answer:** Difference in the memory architecture

- **A bit longer answer:**
  - Main memory units of vector machines had multi-bank structure (collection of many slow memory chips). There was no cache memory.
  - Stride/indirect access logic can be implemented with small additional hardware cost.
  - Modern microprocessors access memory through hierarchy of cache memory, with line size of 16-64 bytes.
  - Stride/indirect access means most of data in one cache line is discarded.
So?

- Something is deeeeply wrong with the modern use of SIMD units.
- Unfortunately, some people have to live with them.
- In the following, I give some example solutions, for particle-based simulations.
Interaction evaluation with modern SIMD units

- Traditional approach and its limitation
- Proposed approach
- Performance example
Traditional approach

Traditional way to use SIMD units for interaction calculation (Phantom GRAPE)

- Start from a double-loop structure (inner $i$ and outer $j$ loop)
- $i$ loop for particles which receive force, $j$ loop for particles which exert force
- unroll $i$ loop to apply SIMD instructions
- also unroll $j$ loop to achieve best performance if necessary
Limitations

- Unrolling i loop needs efficient register broadcast
- Unrolling j loop needs (fairly) efficient horizontal addition
- It is practically impossible to make use of Newton’s third law
- What is the best approach depends very strongly on the details of specific architectures and available instructions. No general solution available.
Proposed approach

- Apply the SIMD operation at one level higher
  - Multiple interaction lists for treecode (multiwalk algorithm)
  - Multiple cell-cell interactions for short-range interactions

- Advantages
  - Application of SIMD operation to innermost loop (multiwalk or cell-cell level) becomes trivial. Most compilers can do reasonable work.
  - Perfect use of Newton’s third law.

- Potential disadvantages
  - Data rearrangement overhead
  - Loop size imbalance
  - Increased L1 access
Performance example

For illustrative purpose only...

- Mimicking cell-cell interactions. Each cell contains 20 particles
- Original data structure: AoS (not strictly...). Each cell has its array of particles
- Calculate gravitational interaction
- Repeat calculation of 64 cell-cell interactions (25600 interactions) 10,000 times. 256M interactions.

Measurement done on g8host00 (Suzukake-dai. Xeon E5-2650V2 2.6GHz), gcc 4.4.7
Compiler flag for vectorization: -ftree-vectorize -O3 -mavx -ffast-math -fassociative-math -ftree-vectorizer-verbose=2
Result

Timing done just with “time” command....

<table>
<thead>
<tr>
<th>Code</th>
<th>execution time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full vectorized</td>
<td>0.83</td>
</tr>
<tr>
<td>Not vectorized</td>
<td>3.12</td>
</tr>
<tr>
<td>No data rearrangement</td>
<td>2.36</td>
</tr>
</tbody>
</table>

- Factor 2.8 speedup over Non-SIMD code is not bad.
- However, actual speed is about 0.31G interactions/sec. Low-accuracy Phantom GRAPE (Tanikawa et al. 2013) can do 2G interactions/sec. on a 3.4GHz Core i7 with AVX.
Why a factor of six difference?

- Clock speed: 3.4GHz vs 2.6GHz
- Use of Newton’s 3rd law: 7 more operations/interaction
- Compiler used full-accuracy (SP) square root and division...

These combined gives difference of factor 3 or around.
How the innermost loop looks like

```c
for(is=0;is<NCELL;is++){
    REAL dx, dy, dz;
    REAL r2inv, r3inv, mir3inv, mjr3inv;
    dx=xi[i][0][is]-xj[j][0][is];
    dy=xi[i][1][is]-xj[j][1][is];
    dz=xi[i][2][is]-xj[j][2][is];
    r2inv = 1.0f/(dx*dx+dy*dy+dz*dz);
    r3inv = r2inv*sqrtf(r2inv);
    mir3inv= r3inv*mi[i][is];
    mjr3inv= r3inv*mj[j][is];
    ai[i][0][is] -= dx*mir3inv;
    ai[i][1][is] -= dy*mir3inv;
    ai[i][2][is] -= dz*mir3inv;
    aj[j][0][is] += dx*mjr3inv;
    aj[j][1][is] += dy*mjr3inv;
    aj[j][2][is] += dz*mjr3inv;
}
```
Some details...

With gcc 4.8.2,

\[ r_{inv} = \frac{1.0}{\sqrt{dx^2+dy^2+dz^2}}; \]
\[ r_{3inv} = r_{inv} \times r_{inv} \times r_{inv}; \]

is MUCH FASTER then

\[ r_{2inv} = \frac{1.0}{dx^2+dy^2+dz^2}; \]
\[ r_{3inv} = r_{2inv} \times \sqrt{r_{2inv}}; \]

but not with gcc 4.4.7... With 4.8.2, the above code resulted asm code which uses vrcpps and vrsqrtps (why??) (Thanks to KN)
.L16:

```
vmovaps (%rsi,%rax), %xmm2
movq -229432(%rbp), %r10
vmovaps (%r12,%rax), %xmm1
vsubps 0(%r13,%rax), %xmm2, %xmm2
vmovaps (%r11,%rax), %xmm0
vsubps (%r15,%rax), %xmm1, %xmm1
vmovaps (%rbx,%rax), %xmm7
vsubps (%r10,%rax), %xmm0, %xmm0
movq -229496(%rbp), %r10
vmulps %xmm2, %xmm2, %xmm3
vmulps %xmm1, %xmm1, %xmm4
vaddps %xmm3, %xmm4, %xmm3
vmulps %xmm0, %xmm0, %xmm4
vaddps %xmm4, %xmm3, %xmm3
vdivps %xmm3, %xmm5, %xmm3
vsqrtps %xmm3, %xmm4
```
vmulps  %xmm4, %xmm3, %xmm3
vmulps  (%r10,%rax), %xmm3, %xmm4
vmulps  (%r14,%rax), %xmm3, %xmm3
vmulps  %xmm2, %xmm4, %xmm6
vmulps  %xmm2, %xmm3, %xmm2
vsubps  %xmm6, %xmm7, %xmm6
vmovaps  %xmm6, (%rbx,%rax)
vmulps  %xmm1, %xmm4, %xmm6
vmulps  %xmm0, %xmm4, %xmm4
vmovaps  (%rcx,%rax), %xmm7
vmulps  %xmm1, %xmm3, %xmm1
vsubps  %xmm6, %xmm7, %xmm6
vmulps  %xmm0, %xmm3, %xmm0
vmovaps  %xmm6, (%rcx,%rax)
vmovaps  (%rdx,%rax), %xmm6
vsubps  %xmm4, %xmm6, %xmm4
vmovaps  %xmm4, (%rdx,%rax)
vaddps  (%r9,%rax), %xmm2, %xmm2
vmovaps  %xmm2, (%r9,%rax)
vaddps (%r8,%rax), %xmm1, %xmm1
vmovaps %xmm1, (%r8,%rax)
vaddps (%rdi,%rax), %xmm0, %xmm0
vmovaps %xmm0, (%rdi,%rax)
addq $16, %rax
cmpq $256, %rax
jne .L16
Operation counts

<table>
<thead>
<tr>
<th>Operation</th>
<th>counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP sub</td>
<td>6</td>
</tr>
<tr>
<td>SP add</td>
<td>5</td>
</tr>
<tr>
<td>SP mul</td>
<td>12</td>
</tr>
<tr>
<td>SP div</td>
<td>1</td>
</tr>
<tr>
<td>SP sqrt</td>
<td>1</td>
</tr>
</tbody>
</table>

- Very reasonable asm code
- The biggest loss of performance from the use sqrt+division (can be avoided with gcc 4.8).
- 14 memory loads and 6 memory stores. Can be reduced to 7 loads and 3 stores...
Summary

- SIMD units on modern microprocessors are very hard to use.
- One possible way to make an efficient use of them is to rearrange data and loop structure so that only simple (and thus efficient) instructions appear in the innermost loop.
- For particle interaction calculation, this can be achieved by applying SIMD on multiwalk (or multi cell-cell) level.
- Achieved performance with gcc automatic vectorization is acceptable, but to achieve really good performance we need a way to let compiler generate low-precision VRSQRTPS...