Application Experiences with Heterogeneous Computing Architecture : "SIMDization"

Hongsuk Yi

hsyi@kisti.re.kr

KISTI Supercomputing Center (Korea)

What do we do?

- Parallel Programming Models in the HC Jungle
  - Heterogeneous computing Hardware and Software
    - HW: Xeon Phi Coprocessors, AMD GPU, NVIDIA GPUs
    - SW: OpenCL, CUDA, Offload, OpenHMPP, MPI, OpenMP, Hybrid
- Main interests: Scientific Libaraies
  - GSL-CL provides high level C interfaces for GNU Scientific Library routines on CPU and GPUs using OpenCL.
    - GSL-CL (http://sourceforge.net/projects/gsl-cl/develop)
  - Development of Parallel application algorithms for nano-materials
    - KMC (KISTI Monte Carlo)
    - KMD (KISTI Molecular Dynamics)

Scaling for Multicore and Manycore

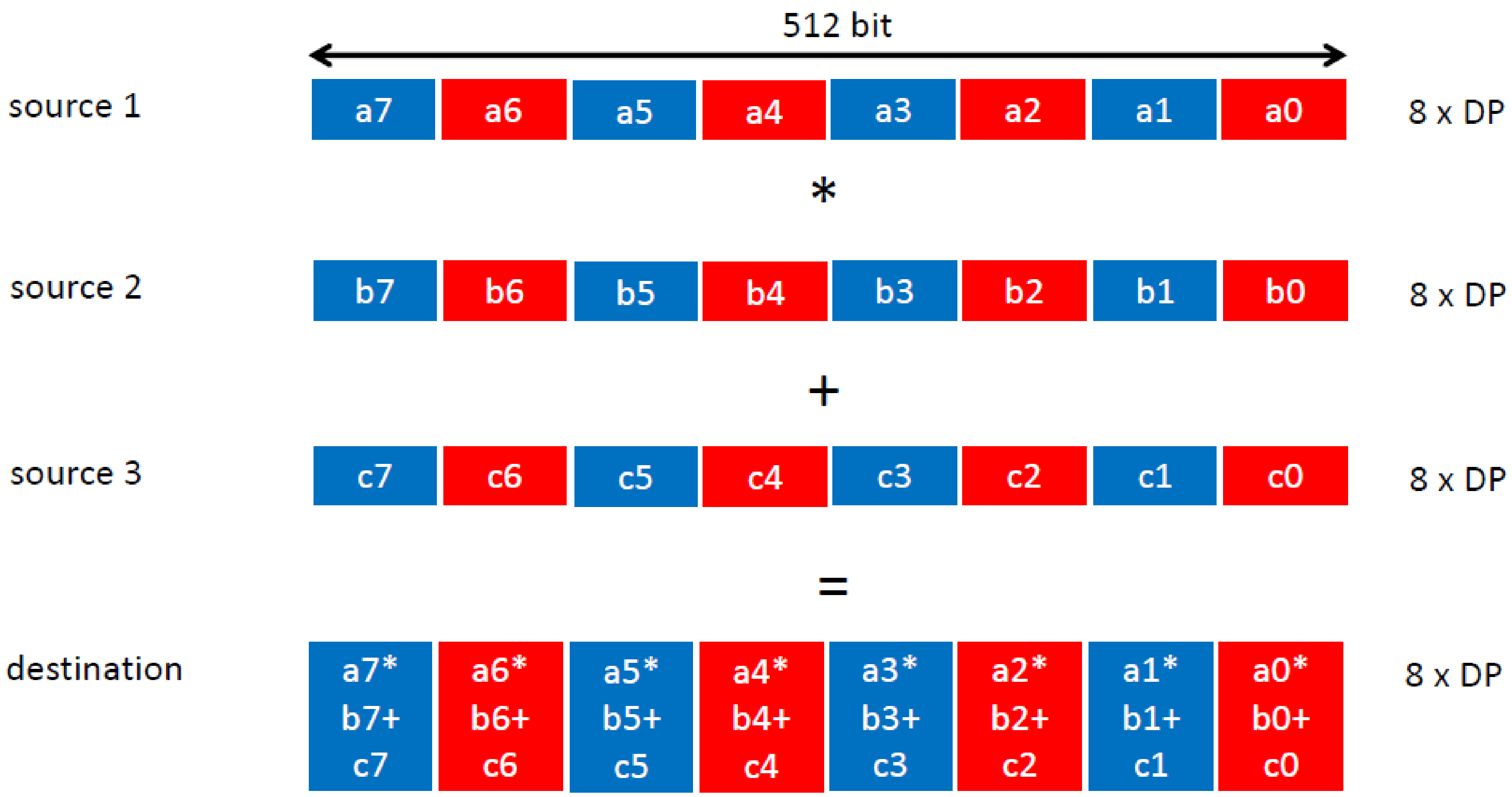
- Increasing Parallelism
  - Core scaling (cores)
  - Thread scaling/core
  - Data level parallelism (SIMD) scaling
    - SIMDization (Single Instruction Multiple Data) or Vectorization
  - **Intel AVX**
    - 256-bit wide register(ymm), 4 doubles/ instruction
    - Shipped with Sandy bridge
  - **Intel Phi**
    - 512-bit wide register(), 8 doubles/ instruction for DP

What is vector

- Scalar computing
  - $c = a + b$, where a, b, c mean *scalar* floating-point variables.
  - ex. $a = 1 * e^{-2}$
- Vector computing
  - $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$, where $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ mean arrays(vector) of floating-point variable and $'+'$ means element-wise addition

- Both addition operations are executed per one CPU cycle.
- If vectors(floating point arrays) have two elements, 200 % speed is expected than scalar addition.
- Xeon Phi supports vectors with 8 elements in double precision.

SIMD Fused Multiply Add



KMD : KISTI Molecular Dynamics

- Empirical Potential MD
  - is very good algorithm
  - Newtonian EOM + Rather simple form of potentials
  - Our choice is Tersoff potential on covalent C, Si, Ge
    - J. Tersoff, PRL 56, 632 (1986), PRB 37, 6991 (1988)

Tersoff empirical potential

$$V_{ij}=f_C(r_{ij})[a_{ij}f_R(r_{ij})+b_{ij}f_A(r_{ij})]\ ,$$

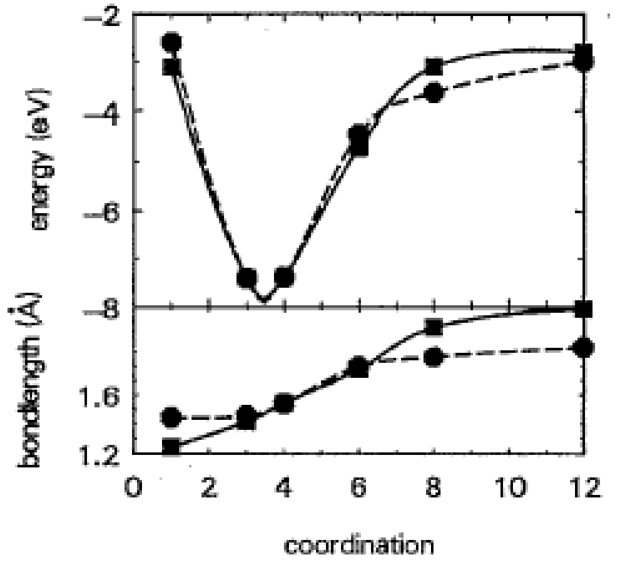
$$f_R(r)=A\exp(-\lambda_1 r)\ ,$$

$$f_A(r)=-B\exp(-\lambda_2 r)\ ,$$

$$b_{ij}=(1+\beta^n\zeta_{ij}^n)^{-1/2n}\ ,$$

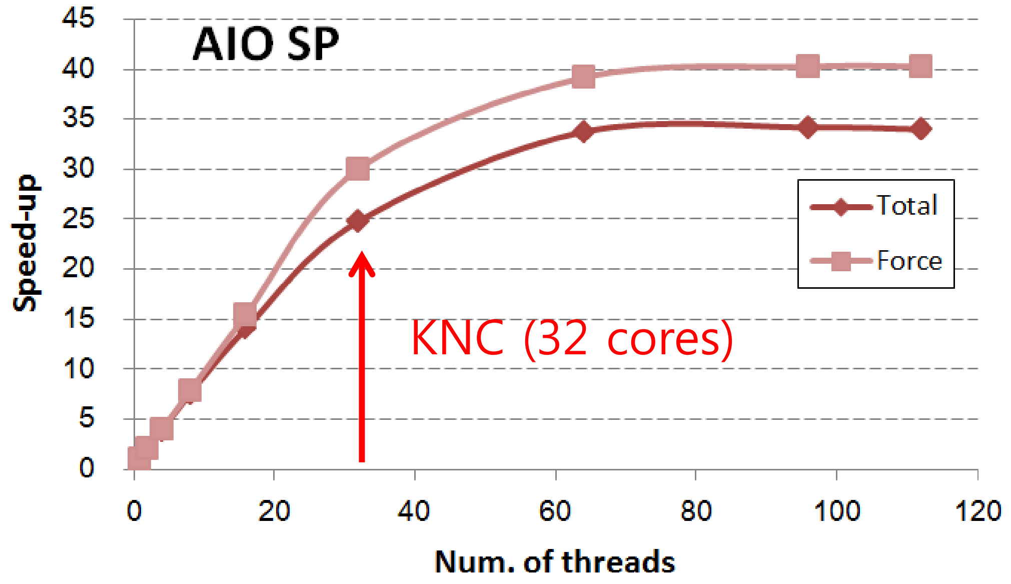
$$\zeta_{ij}=\sum_{k\,(\neq i,j)} f_C(r_{ik})g(\theta_{ijk})\exp[\lambda_3^3(r_{ij}-r_{ik})^3]\ ,$$

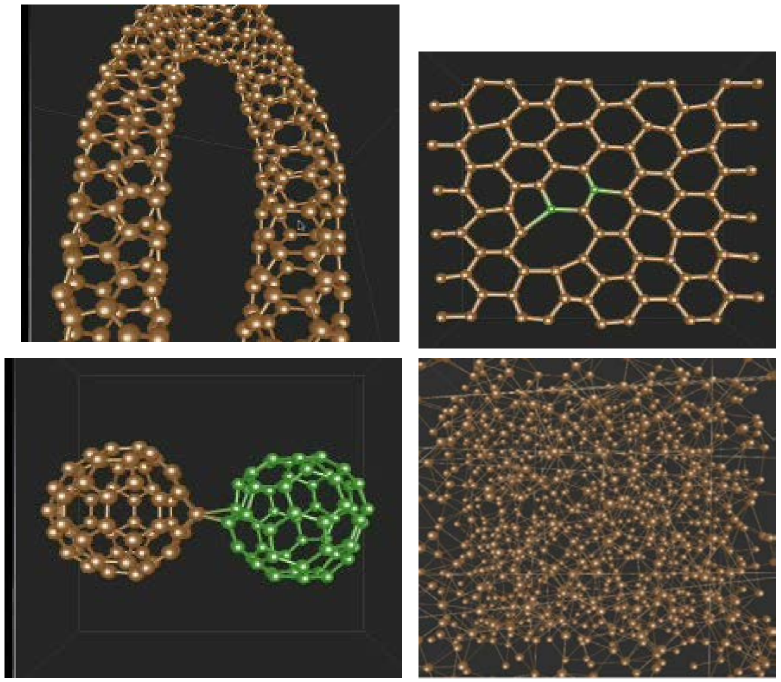
$$g(\theta)=1+c^2/d^2-c^2/[d^2+(h-\cos\theta)^2]\ ,$$



1. Cohesive energy per atom (eV), and bond length

“Native” mode is very easy





Near-theoretical linear performance scaling with the number of cores

SIMDization Strategies

- Auto Vectorization

```
Vij  = fc * (A*exp(-lambda1 * rij) - B*bij*exp(-lambda2 * rij));
```

- Intrinsic (SSE, AVX, MIC, Phi)

```
v_tmp  = _mm256_pow_pd(_mm256_mul_pd(beta, zij), n1))
```

- Class (SSE, AVX, MIC, Phi)

```
F64vec4  v_tmp = pow(F64vec4(beta) * zij,  F64vec4(n))
```

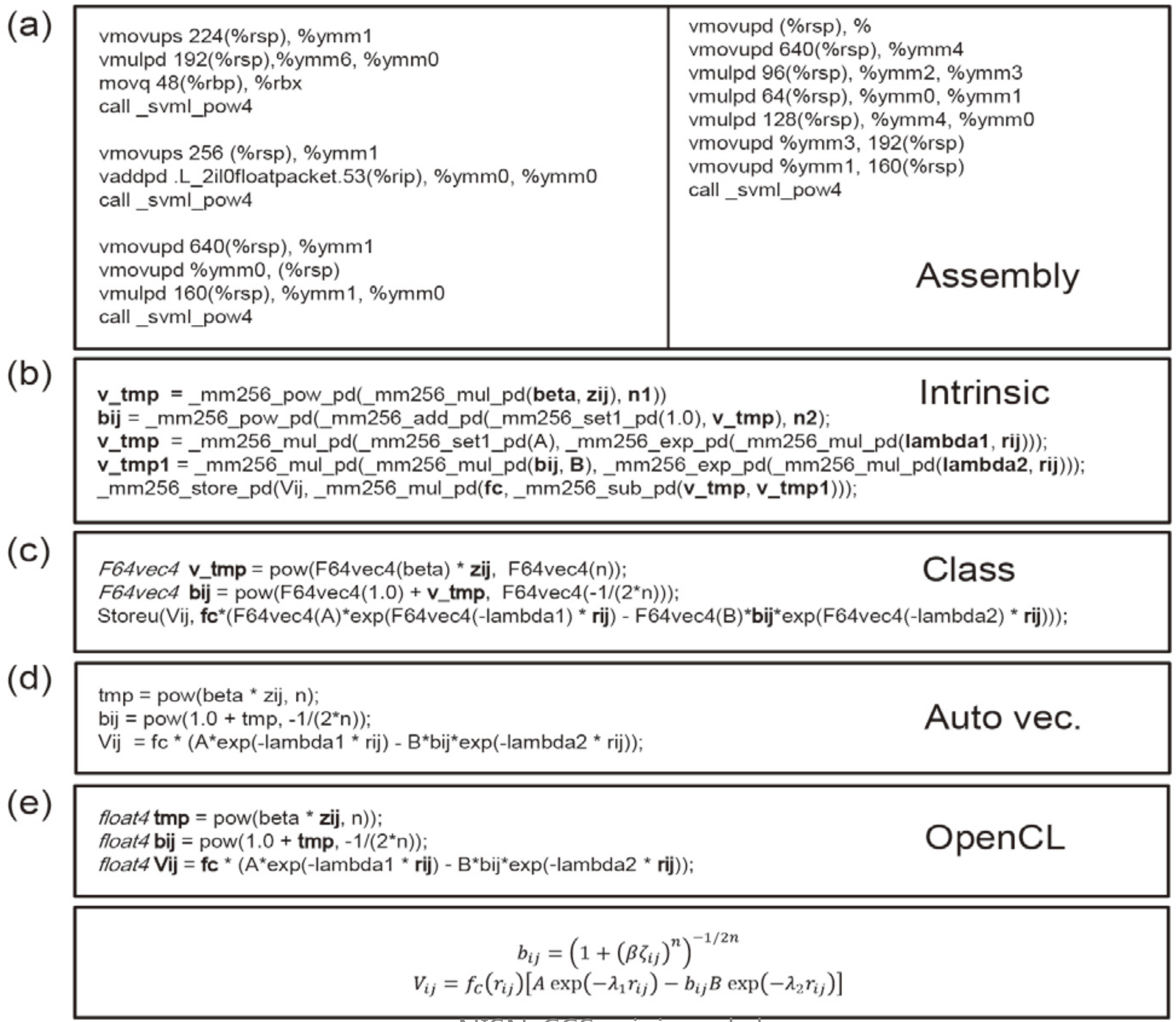
- Assembly

```
vmulpd 160(%rsp), %ymm1, %ymm0
call _svml_pow4
```

- OpenCL (AMD GPU, HD5870, HD6950)

```
float4 Bij = pow(1.0 + tmp, -1/(2*n));
```

Implementation



(a)

```
vmovups 224(%rsp), %ymm1
vmulpd 192(%rsp),%ymm6, %ymm0
movq 48(%rbp), %rbx
call _svml_pow4

vmovups 256 (%rsp), %ymm1
vaddpd .L_2il0floatpacket.53(%rip), %ymm0, %ymm0
call _svml_pow4

vmovupd 640(%rsp), %ymm1
vmovupd %ymm0, (%rsp)
vmulpd 160(%rsp), %ymm1, %ymm0
call _svml_pow4
```

```
vmovupd (%rsp), %
vmovupd 640(%rsp), %ymm4
vmulpd 96(%rsp), %ymm2, %ymm3
vmulpd 64(%rsp), %ymm0, %ymm1
vmulpd 128(%rsp), %ymm4, %ymm0
vmovupd %ymm3, 192(%rsp)
vmovupd %ymm1, 160(%rsp)
call _svml_pow4
```

Assembly

(b) Intrinsic

```
v_tmp  = _mm256_pow_pd(_mm256_mul_pd(beta, zij), n1))
bij = _mm256_pow_pd(_mm256_add_pd(_mm256_set1_pd(1.0), v_tmp), n2);
v_tmp  = _mm256_mul_pd(_mm256_set1_pd(A), _mm256_exp_pd(_mm256_mul_pd(lambda1, rij)));
v_tmp1 = _mm256_mul_pd(_mm256_mul_pd(bij, B), _mm256_exp_pd(_mm256_mul_pd(lambda2, rij)));
_mm256_store_pd(Vij, _mm256_mul_pd(fc, _mm256_sub_pd(v_tmp, v_tmp1)));
```

(c) Class

```
F64vec4  v_tmp = pow(F64vec4(beta) * zij,  F64vec4(n));
F64vec4  bij = pow(F64vec4(1.0) + v_tmp,  F64vec4(-1/(2*n)));
Storeu(Vij, fc*(F64vec4(A)*exp(F64vec4(-lambda1) * rij) - F64vec4(B)*bij*exp(F64vec4(-lambda2) * rij)));
```

(d) Auto vec.

```
tmp = pow(beta * zij, n);
bij = pow(1.0 + tmp, -1/(2*n));
Vij  = fc * (A*exp(-lambda1 * rij) - B*bij*exp(-lambda2 * rij));
```

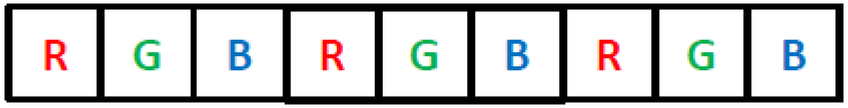
(e) OpenCL

```
float4 tmp = pow(beta * zij, n));
float4 bij = pow(1.0 + tmp, -1/(2*n));
float4 Vij = fc * (A*exp(-lambda1 * rij) - B*bij*exp(-lambda2 * rij));
```

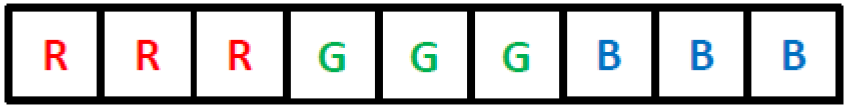
$$b_{ij} = \left(1 + (\beta\zeta_{ij})^n\right)^{-1/2n}$$
$$V_{ij} = f_C(r_{ij})\left[A\exp(-\lambda_1 r_{ij}) - b_{ij}B\exp(-\lambda_2 r_{ij})\right]$$

Data structure : AOS vs SOA

- Array of structure (AOS) for random access
  - Array of 3-dimensional position vectors
  - double position[3 * n];
  - {x1, y1, z1, x2, y2, z2, ... } which is usual coding style.



- Structure of array (SOA) for contiguous acess
  - double position_x[n], position_y[n], position_z[n];
  - {x1, x2, x3, ...}, {y1, y2, y3, ...}, {z1, z2, z3, ...}
  - SOA should be careful because it accompanies over all rewriting the code completely



Summary

- We showed simple SIMDization for the Tersoff algorithm is not efficient.
  - Modified the algorithm through profiling and data dependancy
- Full SIMDization gives over 50% performance of the theoretical peak in both AVX and Intel Xeon Phi.
  - SIMDization is easily merged into well established MPI and OpenMP parallelization.
- SIMDization will be a key factor gradually in HPC of molecular dynamics.

Contact
Hongsuk Yi, Ph.D.
KISTI Supercomputing Center
hsyi@kisti.re.kr

Thanks to

Hogyun Jeong (KISTI)
Seungmin Lee (KISTI)
Heungsik Kim (KISTI)
Byoungwon Choe (Intel)