HPC Code Optimization Workshop

Dr. Fabio Baruffa
fabio.baruffa@lrz.de

Dr. Luigi Iapichino
luigi.iapichino@lrz.de
Outline

• Part 1:
  › Introduction and Motivation
  › Modern Computer Architecture
  › Cache and Memory System
  › Roofline model

• Part 2:
  › Optimization Process
  › Nbody Example
  › Enable vectorization: SIMD

• Part 3:
  › Data layout
  › Data alignment
  › Enable OpenMP

• Part 4:
  › Profiling tools
  › Intel® Advisor XE

• Concluding remarks:
  › Intel PCC @ LRZ
  › Intel® MIC Architecture
Optimization process
Optimization process: Few basic guidelines

- Selection of the best algorithm for the problem

- Use efficient library (why should we reinvent the wheel?)

- Optimal data layout
  - temporal locality: a resource referred at one point in time will be likely reused in the future
  - spatial locality: if a location is referred at a one point in time, its likely that a nearby location will be reused

- Use of compiler optimization flags
Performance analysis

• A real life application has several functions, routines, dependencies,…

• Code optimization and parallelization (shared/distributed memory) is a hard task. The crucial points are:
  • define a good metric for comparisons (timing, flops, memory references,…)
  • define a good representative data setup (not too long, not too short,…)
  • find bottlenecks and critical parts (profiling tools, gprof, Papi, VTune, …)

• Some practical suggestions:
  • there is no general rule
  • use always “realistic” test case to profile the application
  • use always different data sizes for your problem
  • pay attention to input/output
  • use different architectures (whenever possible)
## Common Intel® compiler flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>No optimization. Use in the early stage of dev. and debugging.</td>
</tr>
<tr>
<td>-O1</td>
<td>Optimize for size. Small objects size.</td>
</tr>
<tr>
<td>-O2</td>
<td>Maximize for speed. Includes vectorization.</td>
</tr>
<tr>
<td>-O3</td>
<td>Loop optimization, scalar replacements, efficient cache reuse.</td>
</tr>
<tr>
<td></td>
<td>Aggressive floating point optimization.</td>
</tr>
<tr>
<td>-g</td>
<td>Create symbols for debugging.</td>
</tr>
<tr>
<td>-ipo</td>
<td>Multi-file inter-procedural analysis</td>
</tr>
<tr>
<td>-qopt-report-phase: name1, ...</td>
<td>All (all phases of the optimization), loop, vec (explicit for vectorization), openmp, ipo, offload,...</td>
</tr>
<tr>
<td>-qopenmp</td>
<td>OpenMP 4.0 support</td>
</tr>
</tbody>
</table>

[https://www.nersc.gov/assets/IntelCompilerNERSC.201505013.pdf](https://www.nersc.gov/assets/IntelCompilerNERSC.201505013.pdf)
Code optimization process

- **Scalar optimization**: compiler flags, data casting, precision consistency.
- **Vectorization**: prepare the code for SIMD, avoid vector dependencies.
- **Memory access**: improve data layout, cache access.
- **Multi-threading**: enable OpenMP, manage scheduling and pinning.
- **Communication**: enable MPI, offloading computation.

https://software.intel.com/en-us/articles/what-is-code-modernization; colfaxresearch.com
Code optimization process

- **Scalar optimization**: compiler flags, data casting, precision consistency.
- **Vectorization**: prepare the code for SIMD, avoid vector dependencies.
- **Memory access**: improve data layout, cache access.
- **Multi-threading**: enable OpenMP, manage scheduling and pinning.
- **Communication**: enable MPI, offloading computation.

https://software.intel.com/en-us/articles/what-is-code-modernization; colfaxresearch.com
Nbody example
Let's consider a distribution of point masses located at points $r_1...r_n$ and have masses $m_1,...m_n$.

We want to calculate the position of the particles after a certain time using the Newton law of gravity:

$$\vec{F}_{ij} = \frac{G m_i m_j}{|\vec{r}_j - \vec{r}_i|^3} (\vec{r}_j - \vec{r}_i)$$

$$\vec{F} = m \ddot{\vec{x}} = m \frac{d \vec{v}}{dt} = m \frac{d^2 \vec{x}}{dt^2}$$

Particle.hpp:

```cpp
struct Particle {
  public:
    Particle() { init(); }
    void init() {
      pos[0] = 0.; pos[1] = 0.; pos[2] = 0.;
      vel[0] = 0.; vel[1] = 0.; vel[2] = 0.;
      acc[0] = 0.; acc[1] = 0.; acc[2] = 0.;
      mass = 0.;
    }
    real_type pos[3];
    real_type vel[3];
    real_type acc[3];
    real_type mass;
};
```
null
Hands-on session

- Go to the folder code/nbody/base
- Load the appropriate compiler module
- Run `make` from that directory on the login node
- Reserve a compute node: `llsubmit job.ll`
- Run `llq` and login via `ssh` to the compute node
- Run the code with `make run`
- Play changing the number of particles
- How does the performance change?
Run the default test case on CPU:
```bash
./nbody.x
```

-----------------------------
Initialize Gravity Simulation
```
nPart = 2000; nSteps = 500; dt = 0.1
```
-----------------------------

<table>
<thead>
<tr>
<th>s</th>
<th>dt</th>
<th>kenergy</th>
<th>time (s)</th>
<th>GFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>5</td>
<td>701.22</td>
<td>2.5345</td>
<td>2.2881</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>956.84</td>
<td>2.5453</td>
<td>2.2783</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>1036.6</td>
<td>2.5447</td>
<td>2.2789</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1644.8</td>
<td>2.545</td>
<td>2.2786</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>1565.2</td>
<td>2.5446</td>
<td>2.2789</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1793.8</td>
<td>2.5452</td>
<td>2.2784</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>1849.6</td>
<td>2.5449</td>
<td>2.2787</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>2315.9</td>
<td>2.545</td>
<td>2.2786</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>3017.5</td>
<td>2.545</td>
<td>2.2786</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>3292.1</td>
<td>2.5259</td>
<td>2.2958</td>
</tr>
</tbody>
</table>

# Number Threads      : 1
# Total Time (s)      : 25.42
# Average Performance : 2.2808 +- 0.0056869
-----------------------------
Run the default test case on CPU:
./nbody.x
===============================================
Initialize Gravity Simulation
nPart = 2000; nSteps = 500; dt = 0.1

<table>
<thead>
<tr>
<th>s</th>
<th>dt</th>
<th>kenergy</th>
<th>time (s)</th>
<th>GFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>5</td>
<td>701.22</td>
<td>2.5345</td>
<td>2.2881</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>956.84</td>
<td>2.5453</td>
<td>2.2783</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>1036.6</td>
<td>2.5447</td>
<td>2.2789</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1644.8</td>
<td>2.545</td>
<td>2.2786</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>1565.2</td>
<td>2.5446</td>
<td>2.2789</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1793.8</td>
<td>2.5452</td>
<td>2.2784</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>1849.6</td>
<td>2.5449</td>
<td>2.2787</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>2315.9</td>
<td>2.545</td>
<td>2.2786</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>3017.5</td>
<td>2.545</td>
<td>2.2786</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>3292.1</td>
<td>2.5259</td>
<td>2.2958</td>
</tr>
</tbody>
</table>

# Number Threads   : 1
# Total Time (s)   : 25.42
# Average Perfomance: 2.2808 +- 0.0056869
===============================================

Where and how do we start optimizing the code?
Scalar and general optimization

• The code of part of it can be compiled with more aggressive optimization (-O3) [loop fusion, unroll-and-jam,…]

• Processor specific optimization: -xSSE4.2, -xAVX (E3 and e5 family ), -xCORE-AVX2 (v3), -xCORE-AVX512 (Skylake), -xMIC-AVX512 (KNL), -mmic (KNC)

• Floating point semantics: -fp-model=precise, fast=1,2, …

• Precision of constant and variables: consistent use of single and double precision
Scalar and general optimization

- The code of part of it can be compiled with more aggressive optimization (-O3) [loop fusion, unroll-and-jam,…]

- Processor specific optimization: -xSSE4.2, -xAVX (E3 and e5 family ), -xCORE-AVX2 (v3), -xCORE-AVX512 (Skylake), -xMIC-AVX512 (KNL), -mmic (KNC)

- Floating point semantics: -fp-model=precise, fast=1,2, …

- Precision of constant and variables: consistent use of single and double precision

<table>
<thead>
<tr>
<th>Type</th>
<th>Decimal Point</th>
<th>Exponent</th>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>no</td>
<td>no</td>
<td>none</td>
<td>0,1,300</td>
</tr>
<tr>
<td>long</td>
<td>no</td>
<td>no</td>
<td>1 or L</td>
<td>0L,1L,10000000000000L</td>
</tr>
<tr>
<td>double</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
<td>0.0,1.0,1.0e100</td>
</tr>
<tr>
<td>float</td>
<td>yes</td>
<td>yes</td>
<td>f or F</td>
<td>0.0F,1.0F,1.0e10F</td>
</tr>
<tr>
<td>long double</td>
<td>yes</td>
<td>yes</td>
<td>1 or L</td>
<td>0.0L,1.0L,1.0e1000L</td>
</tr>
</tbody>
</table>

Table 4.4: Conventions for defining literal constants in C and C++.
Scalar and general optimization

- The code of part of it can be compiled with more aggressive optimization (-O3) [loop fusion, unroll-and-jam, ...]

- Processor specific optimization: -xSSE4.2, -xAVX (E3 and e5 family ), -xCORE-AVX2 (v3), -xCORE-AVX512 (Skylake), -xMIC-AVX512 (KNL), -mmic (KNC)

- Floating point semantics: -fp-model=precise, fast=1,2, ...

- Precision of constant and variables: consistent use of single and double precision

- Precision of functions: in MKL (scalar arithmetics) there is single and double precision version of the math functions

- Strength reduction: replacing expensive operations with one less expensive (see GSimulation.cpp line: 163)
Optimization report

- `-qopt-report[=N]`: default level is 2
- `-qopt-report-phase=<vec,loop,openmp,...>`: default is all
- `-qopt-report-file=stdout | stderr | filename`
- `-qopt-report-filter="GSimulation.cpp,113-233"`
Optimization report

- `-qopt-report[=N]`: default level is 2
- `-qopt-report-phase=<vec,loop,openmp,...>`: default is all
- `-qopt-report-file=stdout | stderr | filename`
- `-qopt-report-filter="GSimulation.cpp,113-233"

Let’s see the report in action!
Optimization report

- `-qopt-report[=N]`: default level is 2
- `-qopt-report-phase=<vec,loop,openmp,...>`: default is all
- `-qopt-report-file=stdout | stderr | filename`
- `-qopt-report-filter="GSimulation.cpp,113-233"

Let’s see the report in action!

- `-qopt-report-phase=vec -qopt-report=5`
- **Level 1**: Reports when vectorization has occurred.
- **Level 2**: Adds diagnostics why vectorization did not occur.
- **Level 3**: Adds vectorization loop summary diagnostics.
- **Level 4**: Adds additional available vectorization support information.
- **Level 5**: Adds detailed data dependency information diagnostics.
Hands-on session

- Go to the folder `code/nbody/base`
- Load the appropriate compiler module
- Run `make clean` to remove the old files
- Change the `Makefile` adding the compiler flags to generate the report: `-qopt-report=5`
- Reduce the amount of output: `-qopt-report-filter="GSimulation.cpp,113-233"` (maybe filter more)
- Change compiler flag to: `O3 -xAVX`
- Work on precision consistency
FP conversions

LOOP BEGIN at GSimulation.cpp(150,7)
remark #25444: Loopnest Interchanged: (1 2) --> (2 1)
remark #15541: loop was not vectorized: inner loop was already vectorized
[ GSimulation.cpp(150,7) ]

LOOP BEGIN at GSimulation.cpp(148,5)

remark #15417: vectorization support: number of FP up converts: single precision to double precision 1  [ GSimulation.cpp(163,4) ]
remark #15418: vectorization support: number of FP down converts: double precision to single precision 1  [ GSimulation.cpp(163,4) ]
remark #15417: vectorization support: number of FP up converts: single precision to double precision 6

remark #15452: unmasked strided loads: 6
remark #15453: unmasked strided stores: 3
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 150
remark #15477: vector loop cost: 44.120
remark #15478: estimated potential speedup: 3.28
remark #15487: type converts: 20
remark #15488: --- end vector loop cost summary ---
LOOP END
LOOP END
**Final results of the Nbody example**

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
<th>% of the Peak E5-2650 v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>2.28 GFs</td>
<td>4.8 % (0.6 %)</td>
</tr>
<tr>
<td>ver1</td>
<td>-O3 -xAVX / scalar optimization / 1 thread</td>
<td>6.07 GFs</td>
<td>14.4 % (1.8 %)</td>
</tr>
</tbody>
</table>
Today's CPUs have different levels of parallelism (see previous slides).

Vectorization is the process of converting a scalar algorithm to one which works on multiple elements in one step.

**SIMD** instructions operate on multiple data elements (128-bits registers). Intel® during the years has increased the number and the size of that registers.

- **SSE**: 4 floats
- **SSE2**: 2 doubles
- **AVX**: 8 floats
Requirements for Auto-Vectorization

To be vectorizable, loops must meet the following criteria:

1. **Countable**: the loop trip count must be known at entry of the loop at runtime. Exit of the loop must not be data dependent.
Requirements for Auto-Vectorization

To be vectorizable, loops must meet the following criteria:

1. **Countable**: the loop trip count must be known at entry of the loop at runtime. Exit of the loop must not be data dependent.

2. **Single entry and single exit**: this is implied by countable.

```c
void no_vec(float a[], float b[], float c[]) {
    int i = 0.;
    while (i < 100) {
        a[i] = b[i] * c[i];
        // this is a data-dependent exit condition:
        if (a[i] < 0.0)
            break;
        ++i;
    }
}

remark: loop was not vectorized:
    nonstandard loop is not a vectorization candidate.
```
Requirements for Auto-Vectorization

To be vectorizable, loops must meet the following criteria:

1. **Countable**: the loop trip count must be known at entry of the loop at runtime. Exit of the loop must not be data dependent.

2. **Single entry and single exit**: this is implied by countable.

3. **Straight-line code**: the code must not branch inside the loop; do not break the SIMD operation on consecutive data.
Requirements for Auto-Vectorization

To be vectorizable, loops **must** meet the following criteria:

1. **Countable**: the loop trip count must be known at entry of the loop at runtime. Exit of the loop must not be data dependent.

2. **Single entry and single exit**: this is implied by countable.

3. **Straight-line code**: the code must not branch inside the loop; do not break the SIMD operation on consecutive data.

4. **The innermost loop of a nest**: the only exception is in the case of prior optimization, like loop unrolling, exchange,...
Requirements for Auto-Vectorization

To be vectorizable, loops must meet the following criteria:

1. **Countable**: the loop trip count must be known at entry of the loop at runtime. Exit of the loop must not be data dependent.

2. **Single entry and single exit**: this is implied by countable.

3. **Straight-line code**: the code must not branch inside the loop; do not break the SIMD operation on consecutive data.

4. **The innermost loop of a nest**: the only exception is in the case of prior optimization, like loop unrolling, exchange,…

5. **No function call**: the two major exception are for intrinsic math functions and inlined functions.
# Intel® compiler directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vector</strong></td>
<td>always</td>
<td>Force vectorization even when it might be not efficient.</td>
</tr>
<tr>
<td></td>
<td>[un]aligned</td>
<td>Use [un]aligned data movement instructions for all array vector references.</td>
</tr>
<tr>
<td></td>
<td>[non]temporal(var1[,...])</td>
<td>Do or do not generate non-temporal (streaming) stores for the given array variables. On Intel® MIC architecture, generates a cache-line-evict instruction when the store is known to be aligned.</td>
</tr>
<tr>
<td></td>
<td>[no]vecreminder</td>
<td>Do (not) vectorize the remainder loop when the mail loop is vectorized.</td>
</tr>
<tr>
<td></td>
<td>[no]mask_readwrite</td>
<td>Enables/disables memory speculation causing the generation of [non-]masked loads and stores within conditions.</td>
</tr>
<tr>
<td><strong>simd</strong></td>
<td>vectorlength(n1[,...]) vectorlengthfor(dtype)</td>
<td>Assume safe vectorization for the given vector length values or data type.</td>
</tr>
<tr>
<td></td>
<td>private(var1[,...]) firstprivate(var1[,...]) lastprivate(var1[,...])</td>
<td>Which variables are private to each iteration; firstprivate, initial value is broadcasted to all private instances; lastprivate, last value is copied out from the last instance.</td>
</tr>
<tr>
<td></td>
<td>linear(var1:step1[,...])</td>
<td>Letting know the compiler that var1 is incremented by step1 on every iteration of the original loop.</td>
</tr>
<tr>
<td></td>
<td>reduction(opr:var1[,...])</td>
<td>Which variables are reduction variables with a given operator.</td>
</tr>
<tr>
<td></td>
<td>[no]assert</td>
<td>Warning or error when vectorization fails.</td>
</tr>
<tr>
<td></td>
<td>[no]vecreminder</td>
<td>Do (not) vectorize the remainder loop when the mail loop is vectorized.</td>
</tr>
</tbody>
</table>

From presentation: M. Fernandez, Bayncore Best practices for vectorization
**Vectorization via: #pragma simd**

The compiler helps you to vectorize your code performing a series of tests to determine if the vectorization is possible and efficient. With `pragma simd` you inform the compiler to not do these tests and to vectorize.

**simd-example.c: icc -qopt-report=2 -c simd-example.cpp**

```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];
    }
}
```
Vectorization via: #pragma simd

The compiler helps you to vectorize your code performing a series of tests to determine if the vectorization is possible and efficient. With `pragma simd` you inform the compiler to not do these tests and to vectorize.

```
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];
    }
}
```

LOOP BEGIN at simd-example.c(3,2)
remark #15344: loop was not vectorized: vector dependence prevents vectorization.
First dependence is shown below. Use level 5 report for details
remark #15346: vector dependence: assumed FLOW dependence between line 4 and line 4
remark #25439: unrolled with remainder by 4
LOOP END
Vectorization via: #pragma simd

The compiler helps you to vectorize your code performing a series of tests to determine if the vectorization is possible and efficient. With `pragma simd` you inform the compiler to not do these tests and to vectorize.

```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];
    }
}
```

**simd-example.c:** icc -qopt-report=2 -c simd-example.cpp

**LOOP BEGIN at simd-example.c(3,2)**

- **remark #15344:** loop was not vectorized: vector dependence prevents vectorization.
- First dependence is shown below. Use level 5 report for details
  - **remark #15346:** vector dependence: assumed FLOW dependence between line 4 and line 4
  - **remark #25439:** unrolled with remainder by 4

**LOOP END**
Vectorization via: #pragma simd

The compiler helps you to vectorize your code performing a series of tests to determine if the vectorization is possible and efficient. With `pragma simd` you inform the compiler to not do these tests and to vectorize.

```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];
    }
}
```

The user can enforce vectorization with `pragma simd`.

What is happening with `pragma vector`? It is still under the discretion of the compiler.

Data shared: #pragma simd reduction

The **pragma simd** gives the developer the **full control** on the vectorization but...


With great power comes great responsibility!

In case of data shared which needs to be reduced:

```c
double return_sum(float *a, float *b, float *c, int n) {
    double sum=0;
    #pragma simd reduction(+:sum)
    for (int i=0; i<n; i++)
        sum += a[i] + b[i] * c[i];
    return sum;
}
```

Since the loop runs effectively in parallel by doing two (or four, or eight, etc.) operations simultaneously, the variable **sum** is updated by different iterations and then a **race condition** occurs and the results can be wrong. With **reduction** the compiler generate code to work on private copies of sum and then gather together to get the correct answer.
Hands-on session

• Go to the folder code/nbody/ver1

• The ver1 is the solution of the previous hands-on

• Run make clean to remove old files and recompile

• Change the compiler flags to generate the report:
  -qopt-report=5 -qopt-report-filter="GSimulation.cpp,130-204"
  -qopt-report-phase=vec

• Try to use in the appropriate place:
  #pragma simd
Hands-on session

- Go to the folder `code/nbody/ver1`
- The `ver1` is the solution of the previous hands-on
- Run `make clean` to remove old files and recompile
- Change the compiler flags to generate the report:
  - `qopt-report=5`  `qopt-report-filter="GSimulation.cpp,130-204"
  - `qopt-report-phase=vec`
- Try to use in the appropriate place:
  `#pragma simd`
- Solution in the folder: `code/nbody/ver2`
Hands-on: solution

- Solution in the folder **code/nbody/ver2**

**GSimulation.cpp:**

```c++
...
for (i = 0; i < n; i++)    // update acceleration
real_type ax_i = particles[i].acc[0];
real_type ay_i = particles[i].acc[1];
real_type az_i = particles[i].acc[2];
#pragma simd reduction(+:ax_i,ay_i,az_i)
for (j = 0; j < n; j++)
{
    if (j != i)
    {
        real_type distance, dx, dy, dz;
        real_type distanceSqr = 0.0f;
        real_type distanceInv = 0.0f;
        ...

        ax_i += dx * G * particles[j].mass * distanceInv * distanceInv * distanceInv;  // 6flops
        ay_i += ...  // 6flops
        az_i += ...  // 6flops
    }
}
...
// update position and velocity
```
## Final results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
<th>% of the Peak E5-2650 v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>2.28 GFs</td>
<td>4.8% (0.6 %)</td>
</tr>
<tr>
<td>ver1</td>
<td>-O3 -xAVX / scalar optimization / 1 thread</td>
<td>6.07 GFs</td>
<td>14.4% (1.8 %)</td>
</tr>
<tr>
<td>ver2</td>
<td>#pragma simd / 1 thread</td>
<td>10.7 GFs</td>
<td>25.6% (3.2 %)</td>
</tr>
</tbody>
</table>