KNL Optimization

Dr. Fabio Baruffa
fabio.baruffa@lrz.de
Porting and optimization methods on KNL

Three steps work…almost!

Compile and run for KNL

Optimize performance without coding
(1 week, 2X performance speed-up)

Optimization methods
- Adjust the number of processes and threads
- Use 16GB MCDRAM
- Using -xAVX512 for vectorization

Advanced optimize performance
(1-3 months, more than 3X performance speed-up)

Optimization methods
- Loop optimization, merging, nesting,…
- Parallel model optimization, such as MPI, OpenMP, hybrid
- Memory access optimization, cache tiling,…
Thank you for your attention!
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
  - **spacial 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 (when it is 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. 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>
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.
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;
};
```
Nbody example code

GSimulation.cpp:

```cpp
...
for (i = 0; i < n; i++)    // update acceleration
    for (j = 0; j < n; j++)
        real_type distance, dx, dy, dz;
        real_type distanceSqr = 0.0;
        real_type distanceInv = 0.0;

        dx = particles[j].pos[0] - particles[i].pos[0];    //1flop
        dy = particles[j].pos[1] - particles[i].pos[1];    //1flop

        distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;    //6flops
        distanceInv = 1.0 / sqrt(distanceSqr);    //1div+1sqrt

        particles[i].acc[0] += dx * G * particles[j].mass * distanceInv * distanceInv *  //6flops
distanceInv;
        particles[i].acc[1] += ...    //6flops
        particles[i].acc[2] += ...    //6flops

...    // update position and velocity
...
```
Live-session

- Go to the folder `code/nbody/base`
- Load the appropriate compiler module
- Run `make` from that directory on the login node
- Run the code with `make run`
- Play changing the number of particles
- How does the performance change?
Nbody example code

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>10.099</td>
<td>0.57452</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>956.84</td>
<td>10.098</td>
<td>0.57458</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>1036.6</td>
<td>10.097</td>
<td>0.57461</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1644.9</td>
<td>10.097</td>
<td>0.57462</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>1565.5</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1793.4</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>1848.4</td>
<td>10.097</td>
<td>0.57463</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>2304.1</td>
<td>10.097</td>
<td>0.5746</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>3098.4</td>
<td>10.097</td>
<td>0.57463</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>3324.4</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
</tbody>
</table>

# Number Threads : 1
# Total Time (s)  : 100.97
# Average Perfomance : 0.57463 +- 1.1331e-05
Nbody example code

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>10.099</td>
<td>0.57452</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>956.84</td>
<td>10.098</td>
<td>0.57458</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>1036.6</td>
<td>10.097</td>
<td>0.57461</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1644.9</td>
<td>10.097</td>
<td>0.57462</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>1565.5</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1793.4</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>1848.4</td>
<td>10.097</td>
<td>0.57463</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>2304.1</td>
<td>10.097</td>
<td>0.5746</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>3098.4</td>
<td>10.097</td>
<td>0.57463</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>3324.4</td>
<td>10.097</td>
<td>0.57464</td>
</tr>
</tbody>
</table>

# Number Threads       : 1
# Total Time (s)       : 100.97
# Average Performance : 0.57463 +- 1.1331e-05
**KNL Basic Architecture**

- **Intel® Xeon Phi Processor** 7210: 64 cores at **1.3 GHz** in a single socket
- Theoretical $P_{\text{max}} = (1.3 \times 64 \text{ cores} \times 32 \text{ DP Flops/cycles})$ GF/s = **2662 GF/s**
- **32 = 2-VPU AVX512 → 16 DP per unit x 2 FMA**
- **NB:** no SIMD code gives **83 GF/s**

![Diagram of KNL Basic Architecture]

- **L1d cache:** 32KB
- **L1i cache:** 32KB
- **L2 cache:** 1024KB
- **MCDRAM:** 16GB

- **Hyperthreading**

**Memory performance**
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: 161)
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,130 - 194"`
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,130 - 194"

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,130 - 194"

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.
Live-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,130 - 194" (maybe filter more)

• Change compiler flag to: -xMIC-AVX512

• Work on precision consistency
## Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
</tbody>
</table>
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
## Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
</tbody>
</table>
LOOP BEGIN at GSimulation.cpp(137,7)
  remark #25085: Preprocess Loopnests: Moving Out Load and Store
  [ GSimulation.cpp(150,4) ]
  remark #25085: Preprocess Loopnests: Moving Out Load and Store
  [ GSimulation.cpp(151,4) ]
  remark #25085: Preprocess Loopnests: Moving Out Load and Store
  [ GSimulation.cpp(152,4) ]
  remark #15415: vectorization support: non-unit strided load was generated for
  the variable <this->particles->pos[j][0]>, stride is 10
  [ GSimulation.cpp(143,9) ]
  remark #15415: vectorization support: non-unit strided load was generated for
  the variable <this->particles->pos[j][1]>, stride is 10
  [ GSimulation.cpp(144,9) ]
  ...
  remark #15305: vectorization support: vector length 16
  remark #15309: vectorization support: normalized vectorization overhead 0.491
  remark #15300: LOOP WAS VECTORIZED
  remark #15452: unmasked strided loads: 6
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 115
  remark #15477: vector cost: 14.500
  remark #15478: estimated potential speedup: 7.250
  remark #15488: --- end vector cost summary ---
LOOP END
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];

for (j = 0; j < n; j++)
{
    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 += ...
    az_i += ...
}

... // update position and velocity
A loop that has been automatically vectorized contains loads from memory locations which are **not contiguous** in memory → **non-unit stride load**
The compiler has issued a hardware gather/scatter instructions.

```cpp
struct Particle {
  public:
    ...  
    real_type pos[3];
    real_type vel[3];
    real_type acc[3];
    real_type mass;
};

struct ParticleSoA {
  public:
    ...  
    real_type *pos_x,*pos_y,*pos_z;
    real_type *vel_x,*vel_y,*vel_z;
    real_type *acc_x,*acc_y,*acc_z
    real_type *mass;
};
```
SoA: unit stride access

- The **Particle** structure has **strided** access: the distance between 2 consecutive position for different particles is **10** elements.

```cpp
struct Particle
{
    public:
        ...
        real_type pos[3];
        real_type vel[3];
        real_type acc[3];
        real_type mass;
};
Particle *particles;
```

- The **ParticleSoA** structure has **unit-stride** access: the distance between 2 consecutive position for different particles is **1** element.

```cpp
struct ParticleSoA
{
    public:
        ...
        real_type *pos_x,*pos_y,*pos_z;
        real_type *vel_x,*vel_y,*vel_z;
        real_type *acc_x,*acc_y,*acc_z
        real_type *mass;
};
ParticleSoA *particles;
```

```cpp
void GSimulation :: start()
{
    //allocate particles
    particles = new Particle[get_npart()];
    init_pos();
    init_vel();
    init_acc();
    init_mass();
    ...
}
```

```cpp
void GSimulation :: start()
{
    //allocate particles
    particles = new ParticleSoA[get_npart()];
    particles->pos_x = new real_type[get_npart()];
    particles->pos_y = new real_type[get_npart()];
    particles->pos_z = new real_type[get_npart()];
    particles->vel_x = new real_type[get_npart()];
    particles->vel_y = new real_type[get_npart()];
    ...
```
Live-session

- Go to the folder `code/nbody/ver3`
- 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`
- Does the compiler automatically vectorize the inner loop? Why?
Data dependencies

LOOP BEGIN at GSimulation.cpp(143,20)
  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at GSimulation.cpp(146,4)
  remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at GSimulation.cpp(149,6)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed ANTI dependence between this->particles->pos_x[j] (155:3) and this->particles->acc_z[i] (164:3)
  remark #15346: vector dependence: assumed FLOW dependence between this->particles->acc_z[i] (164:3) and this->particles->pos_x[j] (155:3)
  LOOP END

LOOP END

LOOP BEGIN at GSimulation.cpp(171,4)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed FLOW dependence between this line 173 and this line 185
  remark #15346: vector dependence: assumed ANTI dependence between this line 185 and this line 173
  LOOP END

LOOP END
Data dependencies

Vectorization changes the order of the operation inside a loop, since each SIMD instruction operates on several data at once. Vectorization is only possible if this does not change the results.

**ANTI** dependence: write-after-read (WAR). Statement $i$ precedes $j$, and $i$ uses a value that $j$ computes: $2 \rightarrow 3$

**FLOW** (true) dependence: read-after-write (RAW). Statement $i$ precedes $j$, and $i$ uses a value that $j$ computes: $1 \rightarrow 2$, $2 \rightarrow 4$

1: $x = 1$;
2: $y = x + 2$;
3: $x = z - w$;
... 4: $x = y / z$;

for $i=0; i<N-1; i++$
    $a[i] = a[i+i] + b[i]$;

for $i=0; i<N; i++$
    $a[i] = a[i-i] + b[i]$;
Vectorization: How much we can gain!

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.

<table>
<thead>
<tr>
<th>SIMD</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>[a[0], a[1], a[2], a[3]] 4 floats</td>
</tr>
<tr>
<td>SSE2</td>
<td>[a[0], a[1]] 2 doubles</td>
</tr>
<tr>
<td>AVX</td>
<td>[a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7]] 8 floats</td>
</tr>
</tbody>
</table>
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.
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><code>always</code></td>
<td>Force vectorization even when it might be not efficient.</td>
</tr>
<tr>
<td></td>
<td><code>[un]aligned</code></td>
<td>Use `[un]aligned data movement instructions for all array vector references.</td>
</tr>
<tr>
<td></td>
<td><code>[non]temporal(var1[,...])</code></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><code>[no]vecreminder</code></td>
<td>Do (not) vectorize the remainder loop when the main loop is vectorized.</td>
</tr>
<tr>
<td></td>
<td><code>[no]mask_readwrite</code></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><code>vectorlength(n1[,...])</code></td>
<td>Assume safe vectorization for the given vector length values or data type.</td>
</tr>
<tr>
<td></td>
<td><code>vectorlengthfor(dtype)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>private(var1[,...])</code></td>
<td>Which variables are private to each iteration; <code>firstprivate</code>, initial value is broadcasted to all private instances; <code>lastprivate</code>, last value is copied out from the last instance.</td>
</tr>
<tr>
<td></td>
<td><code>firstprivate(var1[,...])</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>lastprivate(var1[,...])</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>linear(var1:step1[,...])</code></td>
<td>Letting know the compiler that <code>var1</code> is incremented by <code>step1</code> on every iteration of the original loop.</td>
</tr>
<tr>
<td></td>
<td><code>reduction(op:var1[,...])</code></td>
<td>Which variables are reduction variables with a given operator.</td>
</tr>
<tr>
<td></td>
<td><code>[no]assert</code></td>
<td>Warning or error when vectorization fails.</td>
</tr>
<tr>
<td></td>
<td><code>[no]vecreminder</code></td>
<td>Do (not) vectorize the remainder loop when the main loop is vectorized.</td>
</tr>
</tbody>
</table>
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 \texttt{pragma simd} you inform the compiler to not do these tests and to vectorize.

\texttt{simd-example.c: icc -qopt-report=2 -c simd-example.cpp}

\begin{verbatim}
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];
    }
}
\end{verbatim}
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];
    }
}
```

 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.

```
simd-example.c: icc -qopt-report=2 -c simd-example.cpp
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];
    }
}
```

**Too many unknown pointers**

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.

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.
Live-session

- Go to the folder `code/nbody/ver3`
- Run `make clean` to remove old files
- Try to use `#pragma omp simd`
Live-session: solution

- Solution in the folder code/nbody/ver3

GSimulation.cpp:

```cpp
...  // update acceleration

#pragma omp simd
for (j = 0; j < n; j++)
{
    real_type distance, dx, dy, dz;
    real_type distanceSqr = 0.0f;
    real_type distanceInv = 0.0f;
    ...

    particles->acc_x[i] += dx * G * particles->mass[j] * distanceInv * distanceInv * distanceInv;  //6flops
    particles->acc_y[i] += ...  //6flops
    particles->acc_z[i] += ...  //6flops
}

...  // update position and velocity
```
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>1.9722e+06</td>
<td>0.37769</td>
<td>15.354</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>2.4651e+06</td>
<td>0.37759</td>
<td>15.358</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>2.2521e+07</td>
<td>0.37747</td>
<td>15.363</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1.2662e+08</td>
<td>0.37706</td>
<td>15.38</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>4.5866e+08</td>
<td>0.3772</td>
<td>15.374</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1.2487e+09</td>
<td>0.37717</td>
<td>15.375</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>2.2531e+09</td>
<td>0.37714</td>
<td>15.376</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>3.5807e+09</td>
<td>0.37733</td>
<td>15.368</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>5.2184e+09</td>
<td>0.37712</td>
<td>15.377</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>7.3094e+09</td>
<td>0.37712</td>
<td>15.377</td>
</tr>
</tbody>
</table>

# Number Threads : 1
# Total Time (s)  : 3.7734
# Average Perfomance : 15.374 +- 0.0051804
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>1.9722e+06</td>
<td>0.37769</td>
<td>15.354</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>2.4651e+06</td>
<td>0.37759</td>
<td>15.358</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>2.2521e+07</td>
<td>0.37747</td>
<td>15.363</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1.2662e+08</td>
<td>0.37706</td>
<td>15.38</td>
</tr>
<tr>
<td>250</td>
<td>25</td>
<td>4.5866e+08</td>
<td>0.3772</td>
<td>15.374</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1.2487e+09</td>
<td>0.37717</td>
<td>15.375</td>
</tr>
<tr>
<td>350</td>
<td>35</td>
<td>2.2531e+09</td>
<td>0.37714</td>
<td>15.376</td>
</tr>
<tr>
<td>400</td>
<td>40</td>
<td>3.5807e+09</td>
<td>0.37733</td>
<td>15.368</td>
</tr>
<tr>
<td>450</td>
<td>45</td>
<td>5.2184e+09</td>
<td>0.37712</td>
<td>15.377</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>7.3094e+09</td>
<td>0.37712</td>
<td>15.377</td>
</tr>
</tbody>
</table>

# Number Threads : 1
# Total Time (s)  : 3.7734
# Average Performance : 15.374 +/- 0.0051804
## Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
</tbody>
</table>
Live-session: solution

- Solution in the folder `code/nbody/ver4`

**GSimulation.cpp:**

```cpp
...  
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++)
    {
        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;    //6flops
        ay_i += ...    //6flops
        az_i += ...    //6flops
    }

...  // update position and velocity
```
### Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
</tbody>
</table>
## Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
</tbody>
</table>


**Unaligned access**

LOOP BEGIN at GSimulation.cpp(159,2)

  <**Peeled loop for vectorization**>

LOOP END

LOOP BEGIN at GSimulation-nodep.cpp(159,2)

  ...

remark #15389: vectorization support: reference **this->particles->pos_x[j]** has **unaligned access**

[Gsimul...(168,6)]

  ...

remark #15381: **vectorization support: unaligned access used inside loop body**
remark #15305: vectorization support: vector length 16
remark #15309: vectorization support: normalized vectorization overhead 1.026
remark #15300: **PEEL LOOP WAS VECTORIZED**
remark #15442: entire loop may be executed in remainder
remark #15454: masked aligned unit stride loads: 1
remark #15456: masked unaligned unit stride loads: 5
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 189
remark #15477: vector loop cost: 29.370
remark #15478: estimated potential speedup: 5.160
remark #15488: --- end vector loop cost summary ---

LOOP END

LOOP BEGIN at GSimulation.cpp(159,2)

  <**Reminder loop for vectorization**>

LOOP END
Data alignment

The compiler cannot know if your data is aligned to a multiple of the vector register width. This could effect the performance.

A pointer \( p \) is aligned to a memory location on a \( n \)-byte boundary if:
\[
((\text{size}_t)p\%n==0)
\]

For AVX, alignment to 32byte boundaries (4 DP words) allows a single reference to a cache line for moving 4 DP words into the registers.

---

**Single Cache access for 4 DP words**

- **cache line 0** → Load 4DP words
- **cache line 1** → Load 4DP words
- **cache line 2** → Load 4DP words

**Across Cache line access for 4 DP words**

- **cache line 0** → 2 Loads 4DP words
- **cache line 1** → 2 Loads 4DP words
- **cache line 2** → 2 Loads 4DP words

---

**32byte Aligned**

**Non-Aligned**
Data alignment

On the Stack: for declared variables the Intel® C/C++ compiler aligned the data naturally:

```c
float f; //4-byte aligned  double d; //8-byte aligned
```

For array data an attribute is necessary:

```c
float array[N] __attribute__((aligned(32))); //32-byte aligned
```

On the Heap: the array can be allocated/deallocate with special functions:

```c
#include <malloc.h>
...
float *array = (float*) __mm_malloc(N*sizeof(float), 32);
...
__mm_free(array);
```
Data alignment

**SSE**: works better with *16 bytes* alignment

**Why?**: the *XMM* registers are 16 bytes (i.e. 128 bits)

**Penalties**:
- *Unaligned* access vs aligned access (but still in the same cache line) 40% worse.
- *Unaligned* access vs aligned access (but split over *cache* line) 500% worse.

**Rule of thumb**: Try to align to the SIMD register size
- *MMX*: 8 Bytes; *SSE2*: 16 Bytes; *AVX*: 32 Bytes; *AVX512/MIC*: 64 Bytes.

Also try to align blocks of data to *cacheline* size – i.e. 64 Bytes.
The compiler can generate a **Peel** and **Reminder** loop in case where:
- The loop trip count is known only during runtime
- The alignment is not known during compilation

Then the compiler generates a check in code at the beginning of the loop to verify its assumptions. This could cause inefficiency, since every time enters the loops, it does these checks.

```plaintext
for(j = 0; j < N; j++) array[j] = ...
```

<table>
<thead>
<tr>
<th>Peel</th>
<th>vector iteration</th>
<th>vector iteration</th>
<th>vector iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masked</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector iteration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual Padding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cache line boundary</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Reminder</th>
<th>Scalar iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Padding</td>
<td></td>
</tr>
</tbody>
</table>

---

**Vectorization**
Live-session

- Go to the folder code/nbody/ver4
- Run make clean to remove old files
- Try to use the compiler report
- Replace the new/delete statements with the memory alignment functions
- Does the compiler report say what you expect?
- Solution: replace Gsimulation.cpp with Gsimulation-align.cpp
## Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
<tr>
<td>ver4a</td>
<td>Aligned 64 Bytes / 1 thread</td>
<td>24.61 GFs</td>
</tr>
</tbody>
</table>
Final remarks on vectorization

- Not always the compiler does what we want, we need to give suggestions: `__assume_aligned(...)`. Without this step, the compiler will not detect the optimal alignment for accesses using such arrays. Alignment is generally unknown at compile time.

- We have changed *not too much* 😊 in the code.

- We can give hints if the compiler does not vectorize as we expect. (#pragma vector, #pragma omp simd)

- Very good speedup: ~10.2x
Enabling vectorization

Auto Vectorization

Guided Vectorization

Low-level Vectorization

Compiler options

Report hints
Adding #pragmas
Change few lines of code

Vector intrinsics
ASM code

Easy to use

Programmer control

62 Vectorization
Intel® intrinsic instructions

Intrinsics are like library functions, but directly understood by the compiler. They are almost translated into assembly code and are hardware specific.

Intel Cilk Plus includes extensions to C and C++ that allows for parallel operations on arrays. The intent is to allow users to express high-level vector parallel array operations. This helps the compiler to effectively vectorize the code. Array notation can be used for both static and dynamic arrays.

It is supported in C/C++ Intel compiler and GCC 4.9.

The vectorization become explicit: \texttt{array-expression[lower-bound : length : stride]}

```c
int main(int argc, char **argv)
{
    cost int array_size = 10;
    int a[array_size];
    int b[array_size];

    // Initialize array using for loop
    for (int i = 0; i < array_size; i++)
    {
        a[i] = 5;
    }

    // Initialize the array using Array Notation. Since the array is
    // statically allocated, we can use default values for the start index (0)
    // and number of elements (all of them).
    b[:] = 5;
}
```

https://www.cilkplus.org/
Past, present and future of Intel® SIMD

Current Intel® Xeon® Processors

1997 Multimedia Extensions (MMX)

1999 Streaming SIMD Extensions (SSE)

2008 Advanced Vector Instructions (AVX)

2013 AVX2

AVX-512

2015

64-bit SIMD

128-bit SIMD

512-bit SIMD

Past, present and future of Intel® SIMD

Future Intel® Co/Processors (including Knights Landing)

Future Intel® Xeon® Processors

Intel® Many Core Instructions (IMCI)

Current Intel® Xeon® Phi Coprocessors ( Knights Corner)
Multi-threading: OpenMP introduction

**Shared memory** system: the RAM can be accessed by several different CPUs.

**OpenMP** is designed for multi-processor/core, shared memory machine in **UMA/NUMA** architecture.

- A set of compiler directives and API for multithreading applications
- An explicit (not automatic) programming model, offering the programmer full control over parallelization

[www.openmp.org](http://www.openmp.org)
Fork-join model

- OpenMP uses the fork-join model for parallel execution

- The program starts as a single process: the master thread

- During the execution the master thread creates a team of parallel threads \(\rightarrow\) FORK

- The subsequent part is executed in parallel

- When all the threads terminate the execution, they synchronize and terminate \(\rightarrow\) JOIN
Live-session: Nbody example

code/nbody/ver4/GSimulation.cpp:
# Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
<tr>
<td>ver4a</td>
<td>Aligned 64 Bytes / 1 thread</td>
<td>24.61 GFs</td>
</tr>
<tr>
<td>ver5</td>
<td>OpenMP / 128 threads</td>
<td>1414.1 GFs</td>
</tr>
</tbody>
</table>
The `-fp-model` switch lets you choose the floating point semantics at coarse granularity:

- `fast [=1]` allows value `unsafe` optimizations (default)
- `fast=2` allows additional optimizations
- `precise` value-safe optimizations only

More informations:
https://www.nccs.nasa.gov/images/FloatingPoint_consistency.pdf

We all have to think to the 3 points:

- Accuracy
- Reproducibility
- Performance
## Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
<tr>
<td>ver4a</td>
<td>Aligned 64 Bytes / 1 thread</td>
<td>24.61 GFs</td>
</tr>
<tr>
<td>ver5</td>
<td>OpenMP / 128 threads</td>
<td>1414.1 GFs</td>
</tr>
<tr>
<td>ver6</td>
<td>-fp-model fast=2 / 128 threads</td>
<td>1702.9 GFs</td>
</tr>
</tbody>
</table>
Loop tiling
KNL memory model

- On-package high-bandwidth memory (HBM) – MCDRAM
- Optimized for arithmetic performance and bandwidth (not latency)
Cache optimization

Original:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    ...=...*b[j];
```

Tiled:

```
for (ii=0; ii<m; ii+=TILE)
  for (j=0; j<n; j++)
    for (i=ii; i<ii+TILE; i++)
      ...=...*b[j];
```

```
<table>
<thead>
<tr>
<th>i=0</th>
<th>j=0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- cached, LRU eviction policy
- cache miss (read from memory, slow)
- cache hit (read from cache, fast)

**Cache size:** 4
**TILE:** 4

(must be tuned to cache size)

Cache hit rate without tiling: 0%
Cache hit rate with tiling: 50%

LOOP TILING

© Colfax International, 2013–2017
Live-session: Nbody example

code/nbody/ver6/GSimulation.cpp:
## Results of the Nbody example

<table>
<thead>
<tr>
<th>Version</th>
<th>Optimization / Comments</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>-O2 / 1 thread</td>
<td>0.57 GFs</td>
</tr>
<tr>
<td>ver1</td>
<td>-O2 -xMIC-AVX512 / scalar optimization / 1 thread</td>
<td>2.37 GFs</td>
</tr>
<tr>
<td>ver2</td>
<td>No FP convert / 1 thread</td>
<td>7.34 GFs</td>
</tr>
<tr>
<td>ver3</td>
<td>#pragma omp simd / wrong results</td>
<td>15.37 GFs</td>
</tr>
<tr>
<td>ver4</td>
<td>#pragma omp simd reduction / 1 thread</td>
<td>23.32 GFs</td>
</tr>
<tr>
<td>ver4align</td>
<td>Aligned 64 Bytes / 1 thread</td>
<td>24.61 GFs</td>
</tr>
<tr>
<td>ver5</td>
<td>OpenMP / 128 threads</td>
<td>1414.1 GFs</td>
</tr>
<tr>
<td>ver6</td>
<td>-fp-model fast=2 / 128 threads</td>
<td>1702.9 GFs</td>
</tr>
<tr>
<td>ver6tile</td>
<td>loop tiling /128 threads / 65536 particles</td>
<td>2322.2 GFs</td>
</tr>
</tbody>
</table>
Learn how we did it in IPCC