PRACE PATC Course: Vectorisation & Basic Performance Overview

Ostrava, 7-8.2.2017
Agenda

- Basic Vectorisation & SIMD Instructions
- IMCI Vector Extension
- Intel compiler flags
- Hands-on
- Intel Tool VTune Amplifier and Adviser
- Performance overview on the Intel Xeon Phi
Why should we care about Vectors?

- Actually no new tricks available
  - The performance must mainly come from: Nodes, Sockets, Cores and Vectors
- If we require performance we need to know whether the compiler does the right thing for us
- Keep everyone busy at all times, maximise hardware utilisation
- Overlap communication and computation, and workload distribution must be intelligent
# Evolution of Intel Vector Instruction Sets

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Year &amp; Processor</th>
<th>SIMD Width</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>1997 Pentium</td>
<td>64-bit</td>
<td>8/16/32-bit Int.</td>
</tr>
<tr>
<td>SSE</td>
<td>1999 Pentium III</td>
<td>128-bit</td>
<td>32-bit SP FP</td>
</tr>
<tr>
<td>SSE2</td>
<td>2001 Pentium 4</td>
<td>128-bit</td>
<td>8-64-bit Int., SP&amp;DP FP</td>
</tr>
<tr>
<td>SSE3-SSE4.2</td>
<td>2004-2009</td>
<td>128-bit</td>
<td>Additional instructions</td>
</tr>
<tr>
<td>AVX</td>
<td>2011 Sandy-Bridge</td>
<td>256-bit</td>
<td>SP &amp; DP FP</td>
</tr>
<tr>
<td>AVX2</td>
<td>2013 Haswell</td>
<td>256-bit</td>
<td>Int. &amp; additional instruct</td>
</tr>
<tr>
<td>IMCI</td>
<td>2012 KNC</td>
<td>512-bit</td>
<td>32/64-bit Int., SP&amp;DP FP</td>
</tr>
<tr>
<td>AVX-512</td>
<td>2016 KNL</td>
<td>512-bit</td>
<td>32/64-bit Int. SP&amp;DP FP</td>
</tr>
</tbody>
</table>

### Other Floating-Point Vector

<table>
<thead>
<tr>
<th>Manufactures</th>
<th>Instruction Set</th>
<th>Register Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>VMX</td>
<td>4 way SP</td>
</tr>
<tr>
<td></td>
<td>SPU</td>
<td>2 way DP</td>
</tr>
<tr>
<td></td>
<td>Double FPU</td>
<td>2 way DP</td>
</tr>
<tr>
<td></td>
<td>Power8 has 64 VSR each</td>
<td>2 way DP (64bit) or 4 SP(32)</td>
</tr>
<tr>
<td>Motorola</td>
<td>AltiVec</td>
<td>4 way SP</td>
</tr>
<tr>
<td>AMD</td>
<td>3DNow</td>
<td>2 way SP</td>
</tr>
<tr>
<td></td>
<td>3DNow Professional</td>
<td>4 way SP</td>
</tr>
<tr>
<td></td>
<td>AMD64</td>
<td>2 way DP</td>
</tr>
<tr>
<td>ARM 64bit</td>
<td>NEON - V7A Cortex-A</td>
<td>8 SP (16bit)</td>
</tr>
<tr>
<td></td>
<td>V6 ARM 11</td>
<td>Only 32bit (8/16) SP (8bit)</td>
</tr>
</tbody>
</table>

7-8 February 2017
### Vectorisation / SIMD instruction sets

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Data Parallel (DP)</th>
<th>Scalar Parallel (SP)</th>
<th>Bit Width</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MMX</strong></td>
<td>1 x DP</td>
<td>2 x SP</td>
<td>64 bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>128 bit</td>
</tr>
<tr>
<td><strong>SSE</strong></td>
<td>2 x DP</td>
<td>4 x SP</td>
<td>256 bit</td>
</tr>
<tr>
<td><strong>AVX</strong></td>
<td>4 x DP</td>
<td>8 x SP</td>
<td>512 bit</td>
</tr>
<tr>
<td><strong>MIC</strong></td>
<td>8 x DP</td>
<td>16 x SP</td>
<td>512 bit</td>
</tr>
</tbody>
</table>
**SIMD Fused Multiply Add (FMA)**

Instruction set to perform fused multiply-add operations.

<table>
<thead>
<tr>
<th>source 1</th>
<th>source 2</th>
<th>source 3</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>a7</td>
<td>b7</td>
<td>c7</td>
<td>a7*b7 +c7</td>
</tr>
<tr>
<td>a6</td>
<td>b6</td>
<td>c6</td>
<td>a6*b6 +c6</td>
</tr>
<tr>
<td>a5</td>
<td>b5</td>
<td>c5</td>
<td>a5*b5 +c5</td>
</tr>
<tr>
<td>a4</td>
<td>b4</td>
<td>c4</td>
<td>a4*b4 +c4</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
<td>a3*b3 +c3</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
<td>a2*b2 +c2</td>
</tr>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>a1*b1 +c1</td>
</tr>
<tr>
<td>a0</td>
<td>b0</td>
<td>c0</td>
<td>a0*b0 +c0</td>
</tr>
</tbody>
</table>

512 bit
Vectorisation: Approaches

- **Auto vectorisation** → only for loops can be auto-vectorised, you don’t need to do anything!!

- **Guided vectorisation** → using compiler hint and pragmas.

- **Low level vectorisation** → C/C++ vector classes, Intrinsics / Assembly, “full control”
Automatic Vectorisation of Loops

When does the compiler try to vectorise?

- For C/C++ and Fortran, the compiler look for vectorisation opportunities and detect whether loop can be vectorised.
- Enabled using `–vec` compiler flag (or whenever you compile at default optimisation `–O2` or higher levels) and no source code changes.

(Other Intel vec-flags: HSW: `-xCORE-AVX2`, SKX: `-xCORE-AVX512` and KNL: `-xMIC-AVX512`)

(GNU: enabled with `-ftree-vectorize` or `–msse/-msse2` and by default at `–O3` and `–ffast-math`)

- To disable all the autovectorisation use: `-no-vec`
- Sometimes it doesn’t work perfectly and the compiler may need your assistance.
Automatic Vectorisation of Loops

- How do I know whether a loop was vectorised or not?
  - use the vector report flags: `-qopt-report=5 -qopt-report-phase=loop,vec`
    (GNU: `-ftree-vectorizer-verbose=2`)

~$ more autovec.optrpt

... LOOP BEGIN at autovec.cc (14,)
Remark #15300: LOOP WAS VECTORIZED [autovec.cc(14,3)]
LOOP END
...

- The vectorisation should improve loop performance in general
- Use Intel option `-guide-vec` to get tips on improvement
Example Automatic Vectorisation

double a[width], b[width];
for (int i = 0; i < width; i++)
    a[i] += b[i];

This loop will be automatically vectorised😊
Loops can be vectorised

- Straight line code, because SIMD instructions perform the same operation on data elements.
- Single entry and single exit.
- No function calls, only intrinsic math functions such as sin(), log(), exp(), etc., are allowed.

Loops that are not vectorisable:
- Loops with irregular memory access patterns.
- Calculation with vector dependencies.
- Anything that can not be vectorised or is very difficult to vectorise.

Example of a Loop that is not Vectorisable

```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;
    }
}
```

- `icc -c -O2 -qopt-report=5 two_exits.cpp`

  `two_exits.cpp(4) (col. 9): remark: loop was not vectorized: nonstandard loop is not a vectorization candidate.`
Example of Loops that is not Vectorisable

Existence of vector dependence

\[
\text{for (j=n; j<SIZE; j++) }
\text{ a[j] = a[j] + c * a[j-n];}
\]

Arrays accessed with stride 2

\[
\text{for (i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];}
\]

Inner loop accesses a with stride SIZE

\[
\text{for (int j=n; j<SIZE; j++) }
\text{ for (int i=0; i<SIZE; i++)}
\text{ b[i] += a[i][j] * x[j];}
\]

Indirect addressing of x using index array

\[
\text{for (i=0; i<SIZE; i+=2) b[i] += a[i] * x[index[i]];}
\]

- It may be possible to overcome these using switches, pragmas, source code changes

Useful tutorial: Using Auto Vectorisation: https://software.intel.com/en-us/compiler_15.0_vec.c
Data dependencies

**Read after Write**

```c
a[0]=0;
for (j=1; j<SIZE; j++)
    a[j]=a[j-1] + 1;
// this is equivalent to
```

**Write after Read**

```c
a[0]=0;
for (j=1; j<SIZE; j++)
    a[j-1]=a[j] + 1;
// this is equivalent to
```
SIMD instruction sets

Scalar Loop

```
for (l = 0; l < n; l++)
```

SIMD Loop

```
for (i = 0; i < n; i+=16)
    A[i:(i+16)] = A[i:(i+16)] + B[i:(i+16)];
```

Each SIMD add-operation acts on 16 numbers at time
Intel specific switches may generate vector extensions

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimize for current architecture</td>
<td>-xHOST</td>
</tr>
<tr>
<td>Generate SSE v1 code</td>
<td>-xSSE1</td>
</tr>
<tr>
<td>Generate SSE v2 code</td>
<td>-xSSE2</td>
</tr>
<tr>
<td>Generate SSE v3 code (may also emit SSE v1 and SSE v2)</td>
<td>-xSSE3</td>
</tr>
<tr>
<td>Generate SSSE v3 code for Atom based processors</td>
<td>-xSSE_ATOM</td>
</tr>
<tr>
<td>Generate SSSE v3 code (may also emit SSE v1, v2 and SSE v3)</td>
<td>-xSSSE3</td>
</tr>
<tr>
<td>Generate SSE4.1 code (may also emit (S)SSE v1, v2, and v3 code)</td>
<td>-xSSE4.1</td>
</tr>
<tr>
<td>Generate SSE4.2 code (may also emit (S)SSE v1, v2, v3 and v4 code)</td>
<td>-xSSE4.2</td>
</tr>
<tr>
<td>Generate AVX code</td>
<td>-xAVX</td>
</tr>
<tr>
<td>Generate AVX2 code</td>
<td>-xAVX2</td>
</tr>
<tr>
<td>Generate Intel CPUs includes AVX-512 processors code</td>
<td>-xCORE-AVX512</td>
</tr>
<tr>
<td>Generate KNL code (and successors)</td>
<td>-xMIC-AVX512</td>
</tr>
<tr>
<td>Generate AVX-512 code for newer processors</td>
<td>-axCOMMON-AVX512</td>
</tr>
</tbody>
</table>

7-8 February 2017
Vectorisation procedure on Xeon Phi (Intel compiler)

Vectorisation: Most important to get performance on Xeon Phi

- The vectoriser for Xeon Phi works just like for the host
  - Enabled by default at optimisation level `-O2` and above
  - Data alignment should be **64 bytes** instead of 16
  - More loops can be vectorised, because of masked vector instructions, gather/scatter and fused multiply-add (FMA)
  - Try to avoid 64 bit integers (except as addresses)

- Identify a vectorised loops by:
  - Vectorisation and optimisation reports (recommended)
    - `-qopt-report=5` `-qopt-report-phase=loop,vec`
  - Unmasked vector instructions
  - Math library calls to `libsvml`
By default, both host and target compilations may generate messages for the same loop, e.g.

```
host:~/> icc -qopt-report=2 –qopt-report-phase=vec test_vec.c
```
```
test_vec.c(10): (col. 1) remark: LOOP WAS VECTORIZED.
test_vec.c(10): (col. 1) remark: *MIC* LOOP WAS VECTORIZED.
```

To get a vector report for the offload target compilation, but not for the host compilation:

```
host:~/> icc –qopt-report=2 –qopt-report-phase=vec –qoffload option,mic,compiler,”-qopt-report=2” test_vec.c
```
```
test_vec.c(10): (col. 1) remark: *MIC* LOOP WAS VECTORIZED.
test_vec.c(20): (col. 1) remark: *MIC* loop was not vectorized: existence of vector dependence.
test_vec.c(20): (col. 1) remark: *MIC* PARTIAL LOOP WAS VECTORIZED.
```
Vector Intrinsics

- If compiler vectorisation fails…

```c
#include <immintrin.h>

void vecmul(float *a, float *b, float *c, int n) {
    int i;
    __m512 va;
    __m512 vb;
    __m512 vc;
    for (i = 0; i < n; i += 16,
         a += 16, b += 16, c += 16) {
        _mm_prefetch((const char*) (a + 16), _MM_HINT_T0);
        va = _mm512_load_ps(a);
        vb = _mm512_extload_ps(b, _MM_UPCONV_PS_NONE,
                               _MM_BROADCAST32_NONE,
                               _MM_HINT_NONE);
        vc = _mm512_mul_ps(va, vb);
        _mm512_store_ps(c, vc);
    }
}
```
The arrays float A[n] and float B[n] are aligned on 16-bit SSE2 and 64 bit IMCI boundary, where n is a multiple of 4 on SSE and 16 for IMCI.

The vector processing unit on MIC implements a different instruction set with more than 200 new instructions compared to those implemented on the standard Xeon.
What you need to know about SIMD?

SIMD pragma used to guide the compiler to vectorise more loops.

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];
    }
}
```

Function uses too many unknown pointers.
Intel-Specific Vectorisation Pragmas

- **#pragma ivdep**: Instructs the compiler to ignore assumed vector dependencies.
- **#pragma loop_count**: Specifies the iterations for the for loop.
- **#pragma novector**: Specifies that the loop should never be vectorised.
- **#pragma omp simd**: Transforms the loop into a loop that will be executed concurrently using Single Instruction Multiple Data (SIMD) instructions. (up to OpenMP 4.0)
## pragma vector

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>always</td>
<td>instructs the compiler to override any efficiency heuristic during the decision to vectorise or not, and vectorise non-unit strides or very unaligned memory accesses; controls the vectorisation of the subsequent loop in the program; optionally takes the keyword assert.</td>
</tr>
<tr>
<td>aligned</td>
<td>instructs the compiler to use aligned data movement instructions for all array references when vectorising.</td>
</tr>
<tr>
<td>unaligned</td>
<td>instructs the compiler to use unaligned data movement instructions for all array references when vectorizing.</td>
</tr>
<tr>
<td>nontemporal</td>
<td>directs the compiler to use non-temporal (that is, streaming) stores on systems based on all supported architectures, unless otherwise specified; optionally takes a comma separated list of variables.</td>
</tr>
<tr>
<td></td>
<td>On systems based on Intel® MIC Architecture, directs the compiler to generate clevict (cache-line-evict) instructions after the stores based on the non-temporal pragma when the compiler knows that the store addresses are aligned; optionally takes a comma separated list of variables.</td>
</tr>
<tr>
<td>temporal</td>
<td>directs the compiler to use temporal (that is, non-streaming) stores on systems based on all supported architectures, unless otherwise specified.</td>
</tr>
<tr>
<td>vecremainder</td>
<td>instructs the compiler to vectorise the remainder loop when the original loop is vectorised.</td>
</tr>
<tr>
<td>novecremainder</td>
<td>instructs the compiler not to vectorise the remainder loop when the original loop is vectorised.</td>
</tr>
</tbody>
</table>
#pragma offload target(mic) in(a,b:length(n*n)) inout(c:length(n*n)) {
#pragma omp parallel for
for( i = 0; i < n; i++ ) {
    for( k = 0; k < n; k++ ) {
        #pragma vector aligned
        #pragma ivdep
        for( j = 0; j < n; j++ ) {
            //c[i][j] = c[i][j] + a[i][k]*b[k][j];
            c[i*n+j] = c[i*n+j] + a[i*n+k]*b[k*n+j];
        }
    }
}}
Thread Affinity

- Pinning threads is important!
  ~$ export KMP_AFFINITY="granularity=thread,x"
  \(x=\text{compact, scatter, balanced}\)
  “See Intel compiler documentation for more information”.

~$export KMP_AFFINITY=granularity=thread,compact. ~$export KMP_AFFINITY=granularity=thread,scatter.
Tips for Writing Vectorisable Code

• Avoid dependencies between loop interactions
• Avoid read after write dependencies
• Write straight line code (avoid branches such as switch, goto or return statements, etc)
• Use efficient memory accesses by aligning your data to
  • 16-Byte alignment for SSE2
  • 32-Byte alignment for AVX
  • 64-Byte alignment for Xeon Phi
# Vectorisation on KNL vs KNC

<table>
<thead>
<tr>
<th>Knights Corner</th>
<th>Knights Landing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 VPU: Supports 512 bit vectors</td>
<td>2 VPUs</td>
</tr>
<tr>
<td>16x32-bit floats/ integers</td>
<td></td>
</tr>
<tr>
<td>8 x 64-bit doubles</td>
<td></td>
</tr>
<tr>
<td>32 addressable registers</td>
<td>Full support for packed 64-bit integer arithmetic</td>
</tr>
<tr>
<td>Supports masked operations</td>
<td>Support unaligned loads &amp; stores</td>
</tr>
<tr>
<td>Supports SSE/2/3/4, AVX, and AVX2 instruction sets</td>
<td>Supports SSE/2/3/4, AVX, and AVX2 instruction sets but only on 1 of</td>
</tr>
<tr>
<td></td>
<td>the 2 vector-units</td>
</tr>
<tr>
<td>Other features:</td>
<td>Other features:</td>
</tr>
<tr>
<td></td>
<td>Improved Gather/Scatter</td>
</tr>
<tr>
<td></td>
<td>Hardware FP Divide</td>
</tr>
<tr>
<td></td>
<td>Hardware FP Inverse square root</td>
</tr>
<tr>
<td></td>
<td>....</td>
</tr>
</tbody>
</table>
Lab 1: Vectorisation 1: nbody problem
Intel VTune Amplifier
Intel Adviser
What is Intel VTune Amplifier XE?

Where is my application:

- Spending Time? functions taking time ..etc
- Wasting Time? find cache misses and other inefficiencies.
- Waiting Too Long? see locks and cpu utilisation during waiting …
What is Intel VTune Amplifier XE?

- Is a performance profiling tool for serial, OpenMP, MPI and hybrid applications
- Helps users to collect timing performance information
- Intel VTune capable to check the threading performance, load balancing, bandwidth, I/O, overhead and much more
- Analysis is simple using a GUI to visualise results of timeline on your source code...
- Capable to look at memory access on KNL: DDR4 and MCDRAM (Flat or Cache)
- Useful for controlling memory allocation using libmemkind library (hpw_malloc)
Usage with command line `amplxe-cl`

- To print all the options type: `amplxe-cl -help`

  ```
  amplxe-cl <-action> [-action-option] [-global-option]
  [[- -] target [target options]]
  ```

**action**: collect, collect-with, report….

**[-action-option]**: modify behaviour specific to the action

**[-global-option]**: modify behaviour in the same manner for all actions, e.g: -q, -quiet to suppress non essential messages

**[- -]target**: the target application to analyse

**target options**: application options
Using a submission script: amplxe-cl

- Write a submission script based on your resource manager, load all the needed modules and set the environment variables, and launch your program with:
  - `amplxe-cl -collect memory-access -knob analyze-mem-objects=true -no-summary -app-working-dir . - ./exec`

- or to collect only the hotspots on the given target, use:
  - `amplxe-cl -collect hotspots - - mpiexec -n 8 ./exec other-options`

- To generate the hotspots report for the result directory r00hs
  - `amplxe-cl -report hotspots -r r00hs`
Usage of VTune

- compile your code with “-g” for source code and add “-lmemkind” for memory analysis
- compile, e.g:
  - mpiicc -g -O3 -xHOST -qopenmp -lmemkind source.c
  - ifort -g -O3 -xHOST -qopenmp -lmemkind source.f90
- Execute: first load the VTune module and run
  - GUI: amplxe-gui
  - or with a command line: amplxe-cl
- Analyse the results with VTune Amplifier
  - GUI: amplxe-gui
  - or command line: amplxe-cl

7-8 February 2017
Intel MIC Programming Workshop @ Ostrava
Usage: VTune (amplxe-gui r001hs/r001hs.amplxe)

- Elapsed Time
- Top Hotspots
- CPU Usage
# VTune Collections

<table>
<thead>
<tr>
<th>Collections</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotspots</td>
<td>identify the most time consuming sections on the code</td>
</tr>
<tr>
<td>advanced-hotspots</td>
<td>Adds CPI, higher frequency low overhead sampling</td>
</tr>
<tr>
<td>disk-io</td>
<td>Disk IO preview, not working in Stamped (requires root access)</td>
</tr>
<tr>
<td>concurrency</td>
<td>CPU utilisation, threading synchronisation overhead</td>
</tr>
<tr>
<td>memory-access</td>
<td>Memory access details and memory bandwidth utilisation (useful for MCDRAM on KNL )</td>
</tr>
<tr>
<td>hpc-performance</td>
<td>Performance characterisation, including floating point unit and memory bandwidth utilisation</td>
</tr>
</tbody>
</table>
## Useful options

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-data-limit</td>
<td>Override default maximum data collection size</td>
</tr>
<tr>
<td>-no-summary</td>
<td>Do not produce text summary</td>
</tr>
<tr>
<td>-no-auto-finalize</td>
<td>Do not finalise data analysis after collection</td>
</tr>
<tr>
<td>-finalize</td>
<td>Carry out data analysis after collection</td>
</tr>
<tr>
<td>-start-paused</td>
<td>Start application without profiling</td>
</tr>
<tr>
<td>-resume-after=X</td>
<td>Resume profiling after X seconds</td>
</tr>
<tr>
<td>-duration=Y</td>
<td>Profiling only for Y seconds</td>
</tr>
<tr>
<td>analyse-openmp=true</td>
<td>Determine inefficiencies in OpenMP regions</td>
</tr>
<tr>
<td>analyze-memory-objects=true</td>
<td>Determine arrays using most memory bandwidth (highest L2 miss rates)</td>
</tr>
</tbody>
</table>
Intel Adviser
About the Adviser and Capabilities

• Adviser is a vectorisation optimisation and shared memory threading assistance tool for C, C++ and Fortran code
• Adviser supports both serial, threaded, and MPI applications
• Current version is 2017.1.0

Vectorisation:
• Evaluate the efficiency of vectorised code and key SIMD bottlenecks
• Check for loop-carried dependencies dynamically
• Identify memory versus compute balance and provide register utilisation

Threading Advisor:
• find where to add parallelism and identify where the code spends its time.
• Predict the performance you might achieve with the proposed code parallel regions
• Predict the data sharing problems that occur in the proposed parallel code regions
How to use the Adviser

- Load the adviser module and set the environment variables
  - module load adviser
- Compile with (-g, -O2, -vec, -simd, -qopenmp, -qopt-report=5,...etc.)
  - ifort -g -xHOST -O2 -qopt-report=5 source.f90
- Collect the information data
  - advixe-cl -c survey - ./exec
- Analyse the data with
  - advixe-gui
### Summary of predicted parallel behavior

**Vectorization Advisor**
Vectorization Advisor is a vectorization analysis tool that lets you identify loops that will benefit most from vectorization.

#### Program metrics
Elapsed Time: 0.54s
Vector Instruction Set: AVX

<table>
<thead>
<tr>
<th>Metric Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total CPU time</td>
<td>4.73s</td>
</tr>
<tr>
<td>Time in 2 vectorized loops</td>
<td>4.12s</td>
</tr>
<tr>
<td>Time in scalar code</td>
<td>0.61s</td>
</tr>
<tr>
<td>Number of CPU Threads</td>
<td>10</td>
</tr>
</tbody>
</table>

#### Vectorization Gain/Efficiency

- **Vectorized Loops Gain/Efficiency**: 5.78x, -72%
- **Program Theoretical Gain**: 5.16x

#### Top time-consuming loops

<table>
<thead>
<tr>
<th>Loop</th>
<th>Source Location</th>
<th>Self Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute</td>
<td>4nbody.c.74</td>
<td>4.0681s</td>
<td>4.0681s</td>
</tr>
<tr>
<td>compute</td>
<td>4nbody.c.93</td>
<td>0.0480s</td>
<td>0.0480s</td>
</tr>
<tr>
<td>main</td>
<td>4nbody.c.235</td>
<td>0s</td>
<td>4.5920s</td>
</tr>
<tr>
<td><a href="z/Linux_util.c.769">OpenMP worker</a></td>
<td>0s</td>
<td>4.2300s</td>
<td></td>
</tr>
<tr>
<td>compute</td>
<td>4nbody.c.67</td>
<td>0s</td>
<td>4.0681s</td>
</tr>
</tbody>
</table>

#### Collection details

- **Frequency**: 2.60 GHz
- **Logical CPU Count**: 16
- **Operating System**: Linux
- **Computer Name**: login12
for (j = 0; j < SIZE; j++)
{
    if (i + j || i > j)
    {
        float distance3;
        float distanceSqr = 0.0f, distanceInv = 0.0f;
        distance0[] = x_objects[j1] - x_objects[j];
        distance1[] = y_objects[j1] - y_objects[j];
        distance2[] = z_objects[j1] - z_objects[j];

        distanceSqr = sqrt(distance0[] * distance0[] + distance1[] * distance1[] + distance2[] * distance2[]);  
        distanceInv[] = 1.0f / sqrt(distanceSqr);
        ax_objects[] = distance0[] * mass_objects[] * distanceInv[] * distanceInv[] * distanceInv[];
        ay_objects[] = distance1[] * mass_objects[] * distanceInv[] * distanceInv[] * distanceInv[];
        az_objects[] = distance2[] * mass_objects[] * distanceInv[] * distanceInv[] * distanceInv[];
    }
}

#pragma omp for reduction(+?:mean)
for (i = 0; i < SIZE; i++)// update position
{
    vx_objects[] = ax_objects[] * timestep * damping;
    vy_objects[] = ay_objects[] * timestep * damping;
    vz_objects[] = az_objects[] * timestep * damping;
    x_objects[] = vx_objects[];
    y_objects[] = vy_objects[];
    z_objects[] = vz_objects[];
}
## Useful options

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>survey</td>
<td>Helps you detect and select the best places to add parallelism in your code</td>
</tr>
<tr>
<td>Trip Counts</td>
<td>Helps you to collect loop interaction statistics</td>
</tr>
<tr>
<td>Suitability</td>
<td>Helps you predict the likely performance impact of adding parallelism to the selected places</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Helps you predict and eliminate data sharing problems before you add parallelism. 50-500 times slower</td>
</tr>
<tr>
<td>Memory Access Patterns (MAP)</td>
<td>Helps you to collect data on memory access strides 3-20 times slower</td>
</tr>
</tbody>
</table>
Intel Advisor annotations

- **Step 1**: In the Intel Advisor GUI: build applications and create new project.
- **Step 2**: Display the adviser XE workflow and run the Survey Tool (to discover parallel opportunities).
- **Step 3**: Display sources in the survey source window and find where to add Intel Advisor parallel site task annotations.

```c
#include <advisor-annotate.h>

... 
ANNOTATE_SITE_BEGIN();
... 
ANNOTATE_ITERACTION_TASK(task1);
... 
ANNOTATE_SITE_END();

ANNOTATE_DISABLE_COLLECTION_POP; 
... 
ANNOTATE_DISABLE_COLLECTION_PUSH;
```

```c
use advisor_annotate
...
CALL ANNOTATE_SITE_BEGIN()
...
CALL ANNOTATE_ITERACTION_TASK("task1")
...
CALL ANNOTATE_SITE_END()

CALL ANNOTATE_DISABLE_COLLECTION_POP()
...
CALL ANNOTATE_DISABLE_COLLECTION_PUSH()
```
Muti-run analysis with Adviser

- **Survey**
  - `advixe-cl -c survey - -search-dir src:./ - - ./exe`
- **Trip Counts**
  - `advixe-cl -c tripcounts - -search-dir src:./ - - ./exe`
- **Suitability (with site annotations in source code)**
  - `icc -g -xHOST -O2 -qopt-report=5 source.c $ADVISOR_INC $ADVISOR_LIB`
  - `advixe-cl -c suitability —search-dir src:=./ - - ./exec`
- **Dependencies**
  - `advixe-cl -c dependencies -track-stack-variables - -search-dir src:./ - - ./exe`
- **Memory Access Patterns (MAP)**
  - `advixe-cl -c map -record-stack-frame -record-mem-allocations - -search-dir src:./ - - ./exe`
Lab 2: Vectorisation 2: nbody problem
Vectorisation Procedure

• Quantify performance and baseline measurement
• Define a standard metric for all future improvements
• What system components are stressed during runtime (CPU, memory, disks, network)?
• Find the hotspots using VTune Amplifier
• Identify the loop candidate for adding parallelism using the compiler report flags
• Get advices using Intel Advisor
• Add parallelism in the recommended regions
• Check the results and repeat the previous steps
Step 1: check the not vectorised loops

- get the vectorisation report
  - `icc -g -O2 -qopt-report=5 -qopt-report-phase=loop,vec -parallel -mmic -qopenmp nobody.c -o exec.mic`

vi nbody.optrpt

LOOP BEGIN at nbody.c(66,2) inlined into nbody.c(129,3)
remark #15542: loop was not vectorized; inner loop was already vectorized
remark #25018: Total number of lines prefetched=6
remark #25019: Number of spatial prefetches=6, dist=8
remark #25021: Number of initial-value prefetches=3
remark #25139: Using second-level distance 2 for prefetching spatial memory
reference [ nbody.c(87,5) ]
remark #25139: Using second-level distance 2 for prefetching spatial memory
reference [ nbody.c(86,5) ]
remark #25139: Using second-level distance 2 for prefetching spatial memory
reference [ nbody.c(85,5) ]
remark #25015: Estimate of max trip count of loop=10000
Step 1: check the not vectorised loops

- Change in the source code
  - Vectorise the loop with SIMD pragma

```c
int i = 0;
#pragma simd reduction(-: mass_objects)
for (i = 0; i < SIZE; ++i)
{
    x_objects[i] = -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX;
    y_objects[i] = -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX;
    z_objects[i] = -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX, -1.0f + 2.0f * rand() / (float)RAND_MAX;
    vx_objects[i] = -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f;
    vy_objects[i] = -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f;
    vz_objects[i] = -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f, -1.0e-4f + 2.0f * rand() / (float)RAND_MAX * 1.0e-4f;
    ax_objects[i] = 0.0f;
    ay_objects[i] = 0.0f;
    az_objects[i] = 0.0f;
    mass_objects[i] = (float)SIZE + (float)SIZE * rand() / (float)RAND_MAX;
}
```
Step2: Check SIMD loops

- Check the vectorisation report
  - Vectorise the loop with SIMD pragma

```plaintext
remark #15525: call to function 'rand' is serialized   [ nbody.c(161,95) ]
remark #15525: call to function 'rand' is serialized   [ nbody.c(161,149) ]
remark #15525: call to function 'rand' is serialized   [ nbody.c(165,53) ]
remark #15301: SIMD LOOP WAS VECTORIZED
remark #15451: unmasked unaligned unit stride stores: 9
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 1984
remark #15477: vector loop cost: 1951.430
remark #15478: estimated potential speedup: 1.010
remark #15485: serialized function calls: 19
```

- Check the performance
- compile again and check the vectorisation report

```plaintext
LOOP BEGIN at nbody.c(66,2) inlined into nbody.c(236,7)
remark #15542: loop was not vectorized: inner loop was already vectorized
remark #25018: Total number of lines prefetched=6
remark #25019: Number of spatial prefetches=6, dist=8
remark #25021: Number of initial-value prefetches=3
remark #25139: Using second-level distance 2 for prefetching spatial memory reference   [ nbody.c(87,5) ]
remark #25139: Using second-level distance 2 for prefetching spatial memory reference   [ nbody.c(86,5) ]
remark #25139: Using second-level distance 2 for prefetching spatial memory reference   [ nbody.c(85,5) ]
remark #25015: Estimate of max trip count of loop=10000
```
Step 3: Check the Vector length used (typically 8 or 16)

- take a look into the source code and add the IMCI vector extensions on the loops

```c
#pragma omp for private(i,j)
for (i = 0; i < SIZE; i++) // update velocity
{
  #pragma ivdep
  //or #pragma vector always
  for (j = 0; j < SIZE; j++)
  {
    if (i < j || i > j)
    {
      float distance[3];
      float distanceSqr = 0.0f, distanceInv = 0.0f;
    
    ....
    }
  }
}
```

- Check the performance and what do you think!!
Step 4: Check the array alignment status

Alignment status for every array used

LOOP BEGIN at nbody.c(154,7) inlined into nbody.c(214,3)

remark #15389: vectorization support: reference x_objects has unaligned access [ nbody.c(156,7) ]
remark #15389: vectorization support: reference y_objects has unaligned access [ nbody.c(157,7) ]
remark #15389: vectorization support: reference z_objects has unaligned access [ nbody.c(158,7) ]
remark #15389: vectorization support: reference vx_objects has unaligned access [ nbody.c(159,7) ]
remark #15389: vectorization support: reference vy_objects has unaligned access [ nbody.c(160,7) ]
remark #15389: vectorization support: reference vz_objects has unaligned access [ nbody.c(161,7) ]
remark #15389: vectorization support: reference ax_objects has unaligned access [ nbody.c(162,7) ]
remark #15389: vectorization support: reference ay_objects has unaligned access [ nbody.c(163,7) ]
remark #15389: vectorization support: reference az_objects has unaligned access [ nbody.c(164,7) ]
remark #15388: vectorization support: reference mass_objects has aligned access [ nbody.c(165,7) ]

LOOP BEGIN at nbody.c(73,4) inlined into nbody.c(236,7)

remark #15389: vectorization support: reference x_objects has unaligned access [ nbody.c(79,5) ]
remark #15389: vectorization support: reference y_objects has unaligned access [ nbody.c(80,5) ]
remark #15389: vectorization support: reference z_objects has unaligned access [ nbody.c(81,5) ]
remark #15388: vectorization support: reference mass_objects has aligned access [ nbody.c(85,5) ]
remark #15388: vectorization support: reference mass_objects has aligned access [ nbody.c(86,5) ]
remark #15388: vectorization support: reference mass_objects has aligned access [ nbody.c(87,5) ]
Performance overview on Xeon Phi
Performance Comparison

- Sandy-Bridge-EP: 2 sockets × 8 cores @ 2.7 GHz.
- Xeon Phi: 60 cores @ 1.0 GHz.

- **DP FLOP/s:**
  - Sandy-Bridge: 2 sockets × 8 cores × 2.7 GHz × 4 (SIMD) × 2 (ALUs) = 345.6 GFLOP/s
  - Xeon Phi: 60 cores × 1 GHz × 8 (SIMD) × 2 (FMA) = 960 GFLOP/s

Factor 2.7
Memory Bandwidth Comparison

- Sandy-Bridge:
  2 sockets $\times$ 4 memory channels $\times$ 6.4 GT/s $\times$ 2 bytes per channel = 102.4 GB/s

- Xeon Phi:
  8 memory controllers $\times$ 2 channels/controller $\times$ 6.0 GT/s $\times$ 4 bytes per channel = 384 GB/s.

- For complicated memory access patterns: memory latency / cache performance is important. Xeon Phi caches less powerful than Xeon caches (e.g. no L1 prefetcher etc.).
When is Xeon Phi expected to deliver better performance than the host:

1. **Bandwidth-bound code**: If memory access patterns are streamlined so that application is limited by memory bandwidth and not memory-latency bound.

2. **Compute-bound code**: high arithmetic intensity (# operations / memory transfer).

3. **Code should not be dominated by Host <-> MIC communication** limited by slow PCIe v2 bandwidth of 6 GB/s.
Can my code benefit from large vectors?

Is Intel Xeon Phi the Right Coprocessor for my Code?

Is my code ready for high levels of parallelism?

Am I motivated to pursue high levels of parallelism?

Can my code scale to over 100 threads?

Can my code benefit from more memory bandwidth?

Can my code benefit from large vectors?
To Achieve a Good Performance on Xeon Phi

- Data should be aligned to 64 Bytes (512 Bits) for the MIC architecture, in contrast to 32 Bytes (256 Bits) for AVX and 16 Bytes (128 Bits) for SSE.
- Vectorisation is very important for the MIC architecture due to the large SIMD width of 64 Bytes.
- Use the new instructions like `gather/scatter`, `FMA`..etc. which allow more loops to be parallelised on the MIC than on an Intel Xeon based host.
- Use pragmas like `#pragma ivdep`, `#pragma vector always`, `#pragma vector aligned`, `#pragma simd` etc. to achieve autovectorisation.
- Autovectorisation is enabled at default optimisation level `-O2`.
- Let the compiler generate vectorisation reports using the compiler option `-vecreport2` to see if loops were vectorised for MIC (Message "*MIC* Loop was vectorised" etc).
- The options `-opt-report-phase hlo` (High Level Optimiser Report) or `-opt-report-phase ipo_inl` (Inter procedural Optimiser, inline expansion of functions) may also be useful.
- Explicit vector programming is also possible via the new `SIMD` constructs from OpenMP 4.1.
Summary

- Concerning the ease of use and the programmability Intel Xeon Phi is a promising hardware architecture compared to other accelerators like GPGPUs, FPGAs or former CELL processors or ClearSpeed cards.
- Codes using MPI, OpenMP or MKL etc. can be quickly ported. Some MKL routines have been highly optimised for the MIC.
- Due to the large SIMD width of 64 Bytes vectorisation is even more important for the MIC architecture than for Intel Xeon based systems.
- It is extremely simple to get a code running on Intel Xeon Phi, but getting performance out of the chip in most cases needs manual tuning of the code due to failing auto-vectorisation.
- MIC programming enforces programmer to think about SIMD vectorisation
  → Performance on current /future Xeon based systems also much better with MIC-optimised code.
Xeon Phi References

● Books:

● Intel Xeon Phi Programming, Training material, CAPS
● Intel Training Material and Webinars
● V. Weinberg (Editor) et al., Best Practice Guide - Intel Xeon Phi, http://www.prace-project.eu/Best-Practice-Guide-Intel-Xeon-Phi-HTML and references therein
Acknowledgements

- **IT4Innovation**, Ostrava.

- Partnership for Advanced Computing in Europe (**PRACE**)

- **Intel**

- **BMBF** (Federal Ministry of Education and Research)
Thank you.