Intel MIC Programming Workshop:
Vectorisation & Basic Performance Overview
Dr. Momme Allalen (LRZ)

June, 26-28, 2017 @ LRZ
Agenda

- Basic Vectorisation & SIMD Instructions
- Vector loops - how to write loops in vector format
- Intel and GNU compiler vectorisation flags
- Hands-on (Lab1)
- Intel Tool VTune Amplifier and Adviser
- Hands-on (Lab2)
- Performance overview on the Intel Xeon Phi
## 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 Manufactures

<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-v7-A - Cortex-R52</td>
<td>16<em>8b/8</em>16b/4<em>32b/2</em>64b SP</td>
</tr>
<tr>
<td></td>
<td>ARMv8-R</td>
<td>8<em>16bit/4</em>32bit/2*64bit FP</td>
</tr>
</tbody>
</table>

June, 26-28, 2017

Intel MIC Programming Workshop @ LRZ

allalen@lrz.de
Vectorisation / SIMD instruction sets

**MMX**
- 64 bit
- 128 bit

**SSE**
- 256 bit

**AVX**
- 512 bit

**MIC**
- 8 x DP
- 16 x SP

- 1 x DP
- 2 x SP
- 2 x DP
- 4 x SP
- 4 x DP
- 8 x SP
Vectorisation / SIMD instruction sets

On **KNL** each core has 2 Vector Processing Unit (VPU)

- **SP** => $512$ bit registers / $32$ bits x $2$ VPUs = $32$
- **DP** => $512$ bit registers / $64$ bits x $2$ VPUs = $16$

June, 26-28, 2017

Intel MIC Programming Workshop @ LRZ

allalen@lrz.de
Vectorisation / SIMD instruction sets

Scalar Instructions

Vector Instructions

Vector Length

= 9

= 4

= 6

= 1

= 3

= 1

= 4

= 6

= 6

= 1

= 9

= 2

= 4

= 3

= 1

= 4

= 0

= 6

= 6

= 6

= 1

Intel MIC Programming Workshop @ LRZ

June, 26-28, 2017

allalen@lrz.de
SIMD instruction sets

Scalar Loop

\[
\text{for (i = 0; i < n; i++)} \\
\]

SIMD Loop

\[
\text{for (i = 0; i < n; i+=16)} \\
A[i:(i+16)] = A[i:(i+16)] + B[i:(i+16)];
\]

Each SIMD add-operation acts on 16 numbers at time

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ

allalen@lrz.de
KNL and Vector Instruction Sets

- Intel Advanced Vector Extensions 512 (AVX-512)
- 512-bit FP/Integer Vectors
- Gather/Scatter
- Binary compatibility with Xeon
- Supported by non Intel compilers: GCC
Vector Instruction Sets modules on KNL

**AVX-512F**
Fundamentals for basic instructions such as: +, -, *, FMA, … and extension of most AVX2 instructions to 512 vector registers.

**AVX-512CD**
Conflict Detection: is set of instructions useful for (application: binning), e.g: is good for Monte Carlo calculations

**AVX-512ER**
Exponential Reciprocal calculations, functions like: exp, rcp, and rsqrt in SP and DP.

**AVX-512PF**
Prefetch instruction for gather and scatter operation.

Intel MIC Programming Workshop @ LRZ
June, 26-28, 2017
allalen@lrz.de
## 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>Only IMCI sets</td>
<td>Supports SSE/2/3/4, AVX, and AVX2 instruction sets but only on 1 of the 2 vector-units</td>
</tr>
<tr>
<td>In-order core</td>
<td>Other features: Out-of-order core, 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>

The Improvement on the KNL Hardware is not good for non optimised code
Difference between In-order and Out-of-order execution

In-order execution:

• Statically scheduled: executes instructions in sequential order.
• It will not execute next instruction until current instruction is completed.
• Have slower execution speed.

Out-of-order execution:

• Dynamically scheduled execution.
• It will execute next instruction without waiting for the previous instructions to finish unless they depend on the result.
• Faster execution speed.
Vectorisation: Approaches

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

- **Guided vectorisation** → using compiler hint and Tuning with directives (pragmas).

- **Explicit or Low level vectorisation** → C/C++ vector classes, Intrinsics /Assembly, “full control over instructions”, limited portability
Vectorisation: Approaches

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

• Guided vectorisation → using compiler hint and Tuning with directives (pragmas).

• Explicit or Low level vectorisation → C/C++ vector classes, Intrinsics /Assembly, “full control over instructions”, limited portability
Automatic Vectorisation of Loops

When does the compiler try to vectorise?

- For C/C++ and Fortran, the compiler looks for vectorisation opportunities and detects whether a 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
Optimisation report phases

- The compiler reports optimisations consists of 9 Phases:
  - **LOOP**: Loop Nest Optimisations
  - **PAR**: Auto-Parallelisation
  - **VEC**: Vectorisation
  - **OPENMP**: OpenMP
  - **OFFKOAD**: Offload
  - **IPO**: Interprocedural Optimisations
  - **PGO**: Profile Guided Optimisation
  - **CG**: Code Generation Optimisation
  - **TCOLLECT**: Trace Analyser Collection

- Compiler Option for multiple phase reporting:
  - `-qopt-report-phase=VEC,OPENMP,IPO,LOOP`

- Default is “ALL” phases
Optimisation report levels

• The compiler’s optimisation report have 5 verbosity levels
• Specifying report verbosity level:

Compiler Option: \texttt{-qopt-report=N}

Example, VEC Phase levels:
  Level1: reports when vectorisation has occurred
  Level2: adds diagnostics why vectorisation did not occur
  Level3: adds vectorisation loop summary diagnostics
  Level4: adds additional available vectorisation support information
  Level5: adds detailed data dependency information diagnostics
double a[vec_width], b[vec_width];

....

//loop
for (int i = 0; i < vec_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 us 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

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

Arrays accessed with stride 2

```
for (i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];
```

Inner loop accesses a with stride SIZE

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

Indirect addressing of x using index array

```
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

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de
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
```
GNU Support for Automatic vectorisation with AVX-512

up to GCC >= 4.9.1 supports AVX-512 instruction set

```
host:~/> g++ prog.cc -mavx512f -mavx512er -mavx512cd -mavx512pf
```

For automatic vectorisation support add: -O3

```c
//....prog.cc....../
for (int i = 0; i < n; i++)
    B[i] = A[i] + B[i];
host:~/> g++ -s prog.cc -mavx512f -O3
host:~/> cat prog.s
....
vmovapd -16432(%rbp,%rax), %zmm0
vaddpd -8240(%rbp,%rax) , %zmm0, %zmm0
vmovapd %zmm0, -8240(%rbp,%rax)
```

make sure that the vector operations are operating on the zmm0 registers
Vectorisation: Approaches

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

• Guided vectorisation → using compiler hint and Tuning with directives (pragmas).

• Explicit or Low level vectorisation → C/C++ vector classes, Intrinsics /Assembly, “full control over instructions”, limited portability
What you need to know about SIMD?

SIMD pragma used to guide the compiler to vectorise more loops

Example with: 

```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 SIMD instructions. (up to OpenMP 4.0)
<table>
<thead>
<tr>
<th><strong>Pragma</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always</code></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 <code>assert</code></td>
</tr>
<tr>
<td><code>aligned</code></td>
<td>instructs the compiler to use aligned data movement instructions for all array references when vectorising</td>
</tr>
<tr>
<td><code>unaligned</code></td>
<td>instructs the compiler to use unaligned data movement instructions for all array references when vectorizing</td>
</tr>
<tr>
<td><code>nontemporal</code></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><code>temporal</code></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><code>vecremainder</code></td>
<td>instructs the compiler to vectorise the remainder loop when the original loop is vectorised</td>
</tr>
<tr>
<td><code>novecremainder</code></td>
<td>instructs the compiler not to vectorise the remainder loop when the original loop is vectorised</td>
</tr>
</tbody>
</table>
Example for vectorisation pragmas

```c
#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];
        }
    }
}
```
Vectorisation: Approaches

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

• Guided vectorisation → using compiler hint and Tuning with directives (pragmas).

• Explicit or Low level vectorisation → C/C++ vector classes, Intrinsics /Assembly, “full control over instructions”, limited portability
double a[vec_width], b[vec_width];
...

//
__m512d a_vec = _mm512_load_pd(a);
__m512d b_vec = _mm512_load_pd(b);
a_vec = _mm512_add_pd(a_vec, b_vec);
_mm512_store_pd(a, a_vec);

This is explicitly vectorised
Vector Intrinsics

- If compiler automatic vectorisation fails...

```c
#include <immintrin.h>

void vecmul(float *a, 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_BROADCAST32NONE,
                          _MM_HINT_NONE);
    vc = _mm512_mul_ps(va, vb);
    _mm512_store_ps(c, vc);
    }
```
IMCI Instruction Set

IMCI: Initial Many-Core Instruction set

**IMCI is not SSE/or AVX**

SSE2 Intrinsics

```c
for (int i=0; i<n; i+=4)
  __m128 A_vec=_mm_load_ps(A+i);
  __m128 B_vec=_mm_load_ps(B+i);
  Aavec=_mm_add_ps(A_vec, B_vec);
  _mm_store_ps(A+i, A_vec);
}
```

IMCI Intrinsics

```c
for (int i=0; i<n; i+=16)
  __m512 A_vec=_mm512_load_ps(A+i);
  __m512 B_vec=_mm512_load_ps(B+i);
  Aavec=_mm512_add_ps(A_vec, B_vec);
  _mm512_store_ps(A+i, A_vec);
}
```

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.
Vectorisation procedure on Xeon Phi

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
Intel Optimisation flags

- **Precision**
  - -fp-model precise

- **Performance**
  - -fp-model fast=2
  - -no-prec-div
  - -no-prec-sqrt
  - -fno-alias
  - -ftz
  - -align all
  - -march=native

June 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de
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>
## Intel specific switches may generate vector extensions

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>To generate optimised code for KNL</td>
<td>-xMIC-AVX512</td>
</tr>
<tr>
<td>To generate optimised code for Xeon SKX</td>
<td>-xCORE-AVX512</td>
</tr>
<tr>
<td>Cross platform 2 versions: baseline and KNL</td>
<td>-axMIC-AVX512</td>
</tr>
<tr>
<td>Cross platform 2 versions: baseline and Xeon SKX</td>
<td>-axCORE-AVX512</td>
</tr>
<tr>
<td>3 versions: baseline, KNL and Xeon SKX</td>
<td>-axMIC-AVX512,CORE-AVX512</td>
</tr>
<tr>
<td>Generate AVX-512 code for KNL and SKX</td>
<td>-xCOMMON-AVX512</td>
</tr>
<tr>
<td>Generate code for KNC</td>
<td>-mmic</td>
</tr>
</tbody>
</table>
GNU Support for Automatic vectorisation with AVX-512

Cross Platform (KNL and SKX):
-xCOMMON-AVX512 generate:
AVX-512F and AVX-512CD

-Xeon Processors
-xCORE-AVX512 generate:
AVX-512F, AVX512CD, AVX512BW, AVX-512DQ, AVX512VL

-Xeon Phi
-xMIC-AVX512 generate:
AVX-512F, AVX-512CD, AVX-512ER, AVX512FP

June, 26-28, 2017
Intel MIC Programming Workshop @ LRZ
allalen@lrz.de
• Pinning threads is important!

```bash
~$ export KMP_AFFINITY="granularity=thread,\textit{x}\" \\
x=compact, scatter, balanced
```

“See Intel compiler documentation for more information”.

```bash
~$export KMP_AFFINITY=granularity=thread,compact. \\
~$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
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`

- module load amplifier_xe/2017 or amplifier_xe/2018
- 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
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-ojects=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
Load the adviser module and set the environment variables
- module load advisor_xe/2017 or advisor_xe/2018

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

Number of CPU Threads: 10

**Loop metrics**

Total CPU time: 4.73s

Time in 2 vectorized loops: 4.12s

Time in scalar code: 0.61s

100.0% 87.0% 13.0%

**Vectorization Gain/Efficiency**

Vectorized Loops Gain/Efficiency: 5.78x

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>[OpenMP worker]</td>
<td>z/Linux_util.c:769</td>
<td>0s</td>
<td>4.2300s</td>
</tr>
<tr>
<td>compute</td>
<td>4nbody.c:67</td>
<td>0s</td>
<td>4.0681s</td>
</tr>
</tbody>
</table>

**Collection details**

**Platform information**

Frequency: 2.60 GHz

Logical CPU Count: 16

Operating System: Linux

Computer Name: login12
Where should I add vectorization and/or threading parallelism?

**Survey Target**

1. Survey Target

- [ ] Batch mode

**Function Call Sites and Loops**

<table>
<thead>
<tr>
<th>Loop in compute at 4nbod...</th>
<th>Vectorized</th>
<th>AVX</th>
<th>5.73x</th>
<th>Extrav ...</th>
<th>Float3 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>0.048s</td>
<td>0.048s</td>
<td>-100%</td>
<td>3.92x</td>
<td>Float3 15</td>
</tr>
</tbody>
</table>

Vectorized AVX loop processes float32 data type(s) and includes Extracts, Inserts, Square Roots, Type Conversions. No loop transformations applied.

Vectorized AVX loop processes float32 data type(s)

Loop stmts were reordered.

2. Check Dependencies

- Nothing to analyze...

3. Check Memory Access Patterns

- Nothing to analyze...

---

```
for (j = 0; j < SIZE; j++)
{
    if (i + j || i > j)
    {
        float distance[3];
        float distanceSqr = 0.0f, distanceInv = 0.0f;
        distance[0] = x_objects[j] - x_objects[i];
        distance[1] = y_objects[j] - y_objects[i];
        distance[2] = z_objects[j] - z_objects[i];

        distanceSqr = sqrt(distance[0]*distance[0] + distance[1]*distance[1] + distance[2]*distance[2]);
        distanceInv = 1.0f / sqrt(distanceSqr);
        ax_objects[i] += distance[0] * G * mass_objects[i] ... distanceInv * distanceInv;
        ay_objects[i] += distance[1] * G * mass_objects[i] ... distanceInv * distanceInv;
        az_objects[i] += distance[2] * G * mass_objects[i] ... distanceInv * distanceInv;

        //pragma omp for reduction(+,mean)
        for (i = 0; i < SIZE; ++i) // update position
        {
            vx_objects[i] = ax_objects[i] * timestep * damping;
            vy_objects[i] += ay_objects[i] * timestep;
            vz_objects[i] += az_objects[i] * timestep;
        }
    }
}
```
## 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`
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
Lab 2: Vectorisation 2: nbody problem
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, -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, -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, -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

```
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

```
LOOPS 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

<table>
<thead>
<tr>
<th>Location</th>
<th>Remarks</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference x_objects has unaligned access</td>
<td>nbody.c(156,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference y_objects has unaligned access</td>
<td>nbody.c(157,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference z_objects has unaligned access</td>
<td>nbody.c(158,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference vx_objects has unaligned access</td>
<td>nbody.c(159,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference vy_objects has unaligned access</td>
<td>nbody.c(160,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference vz_objects has unaligned access</td>
<td>nbody.c(161,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference ax_objects has unaligned access</td>
<td>nbody.c(162,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference ay_objects has unaligned access</td>
<td>nbody.c(163,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference az_objects has unaligned access</td>
<td>nbody.c(164,7)</td>
</tr>
<tr>
<td>nbody.c(154,7)</td>
<td>vectorization support: reference mass_objects has aligned access</td>
<td>nbody.c(165,7)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Remarks</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbody.c(73,4)</td>
<td>vectorization support: reference x_objects has unaligned access</td>
<td>nbody.c(79,5)</td>
</tr>
<tr>
<td>nbody.c(73,4)</td>
<td>vectorization support: reference y_objects has unaligned access</td>
<td>nbody.c(80,5)</td>
</tr>
<tr>
<td>nbody.c(73,4)</td>
<td>vectorization support: reference z_objects has unaligned access</td>
<td>nbody.c(81,5)</td>
</tr>
<tr>
<td>nbody.c(73,4)</td>
<td>vectorization support: reference mass_objects has aligned access</td>
<td>nbody.c(85,5)</td>
</tr>
<tr>
<td>nbody.c(73,4)</td>
<td>vectorization support: reference mass_objects has aligned access</td>
<td>nbody.c(86,5)</td>
</tr>
<tr>
<td>nbody.c(73,4)</td>
<td>vectorization support: reference mass_objects has aligned access</td>
<td>nbody.c(87,5)</td>
</tr>
</tbody>
</table>

Introduction

Intel MIC Programming Workshop @ LRZ

allalen@lrz.de
Summary

- Concerning the **ease of use and the programmability** Intel Xeon Phi is almost compared to other accelerators like GPGPUs, Mali-GPUs, 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 the actual 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**
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.