Tools for Intel Xeon Phi: VTune & Advisor
Dr. Fabio Baruffa - fabio.baruffa@lrz.de
LRZ, 27.6.- 29.6.2016
Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation

Current: Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation

Now: Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture
More Architecture Parallelism

- Distributed Computing
- Multithreading Parallelism
- Vectorization
Hardware Details of Parallelism

- **Data Parallelism**: Single Instruction Multiple Data (SIMD)
- **Task/Threads Parallelism**: Multiple Tasks/Threads across multiple cores

![Diagram of Processor with Shared Memory and OpenMP Threads]

- **Processor**
  - **Shared Memory**
  - **OpenMP Threads**
  - **core**
    - logical proc
    - vector unit
  - **core**
    - logical proc
    - vector unit
  - **core**
    - logical proc
    - vector unit
  - **core**
    - logical proc
    - vector unit
Optimization process

- **Scalar optimization**: data conversion, precision consistency, …
- **Vectorization**: enable SIMD, vector dependency, …
- **Multi-threading**: enable OpenMP, scheduling, …
- **Memory access**: data layout, streaming access, …
- **Communication**: enable MPI, offloading, …
Optimization process

- **Scalar optimization**: data conversion, precision consistency, …
- **Vectorization**: enable SIMD, vector dependency, …
- **Multi-threading**: enable OpenMP, scheduling, …
- **Memory access**: data layout, streaming access, …
- **Communication**: enable MPI, offloading, …

Q: How do we check if what I have implemented is correct?
Optimization process

- **Scalar optimization**: data conversion, precision consistency, …
- **Vectorization**: enable SIMD, vector dependency, …
- **Multi-threading**: enable OpenMP, scheduling, …
- **Memory access**: data layout, streaming access, …
- **Communication**: enable MPI, offloading, …

Q: How do we check if what I have implemented is correct?

Intel® VTune Amplifier XE  
Intel® Advisor
What is Intel® VTune™ Amplifier XE

- Powerful tool for analysing the node-level performance
  - Multiple programming languages (C/C++, Fortran, .NET, Java, Assembly)
  - Support for all latest Intel® processors (incl. Intel® MIC/Haswell microarchitectures)
- Performance analysis at different levels
  - High-level (code analysis, parallelisation efficiency), no special rights needed
  - Low-level (inspection of all architectural components), module driver is required
  - Processor-specific analysis (e.g., utilisation of vector units on Intel® MIC)
- Minimal execution time overhead
  - No recompilation or special linking needed
  - H/W counter sampling and multiplexing → all interesting events gathered once
- Multiplatform (Windows/Linux, 32/64-bit) + complete command-line interface
  - Can produce very large traces (~400MB per min. of exec. time)
Hot-Spot Guided Optimization

**Typical Workflow**

VTune

- find top hotspots

Optimize

- eliminate issues, reduce hotspot time

Compiler

- identify issues in optimization report
Using Vtune Aplifier XE

1. Compile code with \texttt{–g –O2} or \texttt{–g –O3}
2. Set environment variables or use a wrapper script
3. Tweak code input for a short representative run

- Advanced hotspot analysis is a good start point
- Focusing on optimization report for detected hotspots
  \texttt{-qopt-report=5}
Performance Profiling Terminology

- **Elapsed Time**
  The total time your target application run. Wall clock time at the end of the application – Wall clock time at start of the application.

- **CPU Time**
  The amount of time a thread spends executing on a logical processor. For multiple threads, the CPU time of the threads is summed.

- **Wait Time**
  The amount of time that a given thread waited for some event to occur, such as: synchronization waits and I/O waits.
First view

Spending Time?

- Focus tuning on functions taking time
- See call stacks
- See time on source

Wasting Time?

- See cache misses on your source
- See functions sorted by # of cache misses

Waiting Too Long?

- See locks by wait time
- Red/Green for CPU utilization during wait

Windows & Linux
Low overhead
No special recompiles
Typical workflow (Desktop – Creating a new project)
Typical workflow (Desktop – Creating a new project)

![Create a Project dialog box with project name 'MyProject' and location '/home/hpc/pr28fa/di29daf/courses/vtune']
Typical workflow (Desktop – Creating a new project)

Local/Remote/Coprocessor

Command-line parameters

Environment variables

Or attach to a running process
Typical workflow (Desktop – Selecting analysis type)

High-level analysis

Low-level analysis using H/W counters

Intel® Xeon Phi™

Command line invocation

27-29 June 2016

Intel MIC Programming Workshop, LRZ 2016
Typical workflow (HPC – Running from the CLI)

1. Obtain access to a compute node (job queue)
2. Invoke VTune’s command line interface
   - [environment] amplxe-cl –collect <analysis type> -- <appname> [arguments]
   - OMP_NUM_THREADS=32 amplxe-cl –collect hotspots -- ./mandelbrot 100000 0 0 1
   - amplxe-cl –help collect → Lists available collection types
3. Open analysis results from the GUI on the login node or locally
   - Select the ‘.amplxe’ file from the results directory
Performance bottlenecks are highlighted in red.

**Wall clock time**

**Cumulative CPU time**

**Potential gain estimate**

**Elapsed Time:** 3.838s

- **Total Thread Count:** 32
- **Overhead Time:** 10.936s
- **Spin Time:** 11.642s

A significant portion of CPU time is spent in synchronization or threading overhead. Consider increasing task granularity or the lock implementation (for example, by backing off then descheduling), or adjusting the synchronization.

- **CPU Time:** 81.060s
- **Paused Time:** 0s

**OpenMP Analysis**

- **Collection Time:** 3.838s
  - **Serial Time (outside any parallel region):** 0.175s (4.6%)
  - **Parallel Region Time:** 3.663s (95.4%)
    - **Estimated Ideal Time:** 1.828s (47.6%)
    - **Potential Gain:** 1.835s (47.8%)

The time wasted on load imbalance or parallel work arrangement is significant and negatively impacts the application performance. Focus on tuning OpenMP regions with the highest metric values. Make sure the workload of the regions is enough and the loop schedule...
Elapsed Time: 6 seconds
CPU Time: T1(4s) + T2(3s) + T3(3s) = 10 seconds
Wait Time: T1(2s) + T2(2s) + T3(2s) = 6 seconds
Major views – Performance overview

**CPU Usage Histogram**
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage...

**Overall CPU usage**

**OpenMP Region CPU Usage Histogram**
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously in an OpenMP region. Spin and Overhead time adds to the Idle CPU usage value. OpenMP regions in the drop-down list are sorted by Potential Gain (Elapsed Time) so it’s recommended to start exploration from the top.

**Per OpenMP region**

**Use the sliders to adjust the regions**
Major views – Thread behaviour

- Threads spinning
- Useful work
- Threads sleeping
- Concurrency
- Synchronization objects
Memory bandwidth analysis

Elapsed Time: 1.500s
CPU Time: 21.209s
Paused Time: 0s

Average Bandwidth

<table>
<thead>
<tr>
<th>Package</th>
<th>Bandwidth, GB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>package_0</td>
<td>29.726</td>
</tr>
<tr>
<td>package_1</td>
<td>0.111</td>
</tr>
</tbody>
</table>

OpenMP Region Duration Histogram

This histogram shows the total number of region instances in your application executed with a specific duration. High numbers can indicate a bottleneck. Explore the data provided in the Bottom-up, Top-down Tree, and Timeline panes to identify code regions with high memory bandwidth usage.

OpenMP Region: main$omp$parallel:16@unknown:248:255

Mem. B/W requirements per socket
Memory bandwidth analysis

Total B/W

Read B/W

Write B/W

GB/s
• Build process on the login node
• Use of the VTune GUI from the login node
  − Forward X server + data compression
    − $> \text{ssh } -\text{XC } XXXXX@supermic.smuc
    − Copy results to $\text{TMPDIR}$ (fast local filesystem) before processing
• Test runs + profiling on the compute nodes
  − Use of the VTune command-line interface
  − Obtain a compute node either interactively (in this hands-on session) or submit your job to the queue (best practice)
• In every case, remember to load the appropriate module first
  − $> \text{module load amplifier_xe/2016}$
VTune Lab 2/6

- Login to the training server
- Login to the SuperMIC login node
  - $> ssh -XC XXXXX@supermic.smuc.lrz.de
- Load the necessary modules
  - $> module load amplifier_xe/2016
- Copy the exercises to your home directory
  - $> cp –r /lrz/sys/courses/MIC_Workshop/ ~
- Build the projects and move to the first project (mandelbrot)
  - $> cd VTUNE/
  - $> make ← Always run “make” from the top-level source directory
  - $> cd mandelbrot/
Obtain a compute node interactively and load the necessary modules
- $> llsubmit /lrz/sys/courses/MIC_Workshpt/tools/job.ll
- $> llq ← Check which node is assigned to you
- $> ssh i01r13xxx-ib
- $> module load amplifier_xe

Run the first exercise
- $> cd vtune/mandelbrot
- $> OMP_NUM_THREADS=32 KMP_AFFINITY=scatter,granularity=fine ./mandelbrot 50000 0 0 1

Run it again from within VTune using a basic hotspots analysis
- $> OMP_NUM_THREADS=32 KMP_AFFINITY=scatter,granularity=fine amplxe-cl --collect hotspots -- ./mandelbrot 100000 0 0 1
- Go back to the login node
- Copy the analysis output to the fast local $TMPDIR$
  - $>\; \text{mkdir} \; -p \; \text{$TMPDIR$/vtune/mandelbrot}$
  - $>\; \text{cp} \; -r \; \text{r000hs/}$\; \text{$TMPDIR$/vtune/mandelbrot}$
- Run the VTune GUI on the results
  - $>\; \text{amplxe-gui} \; \text{$TMPDIR$/vtune/mandelbrot/r000hs/r000hs.amplxe}$ &
- Which function or loop is the hotspot?
- How good is the parallelisation efficiency?
• Go to your assigned compute node again and reload the VTune module
• Run a “locks-and-waits” analysis to get more information about parallelisation
  – $> \text{OMP\_NUM\_THREADS}=32 \text{ KMP\_AFFINITY=scatter amplxe-cl --collect locksandwaits -- ./mandelbrot 100000 0 0 1}$
• Go back to the login node again
• Copy the results to your subdirectory inside $\text{TMPDIR}$ and open it with the GUI
  – $> \text{amplxe-gui $\text{TMPDIR}/vtune/mandelbrot/r000lw/r000lw.amplxe}$ &
• What is the key performance problem?
  – Severe load imbalance due to different workload per Mandelbrot line
● Change the OpenMP load balancing scheme to “dynamic”
  – mandelbrot.c:83
  – #pragma omp for schedule(dynamic)
● Rebuild the code, rerun the VTune analysis and open the new result file
● Is the imbalance issue fixed?
  – Yes, but the critical section for drawing the lines still limits parallelism
Setting up Advisor

- $> \text{module load advisor}\_xe/2017
- Compile your code normally: -xAVX would be beneficial
- Start the GUI and create the project (see VTune): advixe-gui
- Run CLI: advixe-cl –c survey ./myapp.x
• **Compilation** – Compile the application with appropriate flags
• **Initial Survey** – General overview analysis for finding hot-spots
• **Trip Count** – Additional informations on loop/function call count
• **Investigate Loops** – View analysis details from the Advisor
• **Deeper Analysis** – Gather any additional information requested
• **Implement Optimization** – One optimization per iteration
Example code: ver1

*Look into the code!*
Example code: ver1

```c
const double t0 = omp_get_wtime();
#pragma omp parallel for collapse(3) schedule(guided)
  for (int i = 0; i < m; i++)
    for (int j = 0; j < m; j++)
      for (int k = 0; k < m; k++)
        { 
          pot.CalculateGravitationalPotential( i, j, k, computedPot[i*m*m+j*m+k]);
        }
  const double t1 = omp_get_wtime();

void potential :: CalculateGravitationalPotential(int Xi, int Xj, int Xk, real_type &pot)
{
  pot = 0.0;
  int n;
  for (n=0; n < _nMasses; n++)
    { 
      const real_type dx = particles[n].position[0] - (real_type) Xi;  //1flop
      const real_type dy = particles[n].position[1] - (real_type) Xj;  //1flop
      const real_type dz = particles[n].position[2] - (real_type) Xk;  //1flop
      const real_type dist = dx*dx+dy*dy+dz*dz;  //5flops
      pot -= G * particles[n].mass / sqrt(dist);  //2flops+di
    }
  ```
Performance

Xeon Ivy-Bridge: 17.4 GFlops
Xeon Phi KNC: 19.5 GFlops
Xeon Ivy-Bridge: 17.4 GFlops
Xeon Phi KNC: 19.5 GFlops

Are these numbers what we expect?
Let's compare to the theoretical performance!

\[ P = \#\text{cores} \times (\text{FP instructions per cycle}) \times (\text{FP operations per instruction}) \times (\text{clock frequency}) = c \times \text{FP} \times S \times f \]

Xeon Ivy-Bridge: \(16 \times 2(1\text{mul}+1\text{add}) \times 8\text{sp}(4\text{dp}) \times 2.6\text{GHz} = 665(332) \text{ GFlops}\)

Xeon Phi KNC: \(60 \times 32(16) \text{ Flops/cycle} \times 1.05\text{GHz} = 2022(1011) \text{ GFlops}\)
Let’s compare to the theoretical performance!

P = #cores \times (\text{FP instructions per cycle}) \times (\text{FP operations per instruction}) \times (\text{clock frequency}) = c \times FP \times S \times f

Xeon Ivy-Bridge: 16 \times 2(1\text{mul} + 1\text{add}) \times 8\text{sp}(4\text{dp}) \times 2.6\text{GHz} = 665(332) \text{GFlops}

Xeon Phi KNC: 60 \times 32(16) \text{Flops/cycle} \times 1.05\text{GHz} = 2022(1011) \text{GFlops}
1. **Measure baseline release build performance:** define a metric which makes sense for the code

2. **Determine hotspots using Intel VTune Amplifier:** most time-consuming functions in the application

3. **Determine loop candidates using compiler report:**
   -qopt-report=5 –qopt-report-phase=loop,vec

4. **Get advice using Intel Advisor:**
   use the vectorization analysis capability of the tool

5. **Implement vectorization recommendations**

6. **Repeat**

more info: https://software.intel.com/en-us/articles/vectorization-toolkit
Six-Step Vectorization Methodology

1. **Measure baseline release build performance:**
   define a metric which makes sense for the code

2. **Determine hotspots using Intel VTune Amplifier:**
   most time-consuming functions in the application

3. **Determine loop candidates using compiler report:**
   -qopt-report=5 –qopt-report-phase=loop,vec

4. **Get advice using Intel Advisor:**
   use the vectorization analysis capability of the tool

5. **Implement vectorization recommendations**

6. **Repeat**

more info: https://software.intel.com/en-us/articles/vectorization-toolkit
potential.optrpt

... Begin optimization report for: potential::
  CalculateGravitationalPotential(potential *, int, int, int, real_type &)
  Report from: Vector optimizations [vec]
  LOOP BEGIN at potential.cpp(25,3)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization
  remark #15346: vector dependence: assumed ANTI dependence between this line 27 and pot line 32
  remark #15346: vector dependence: assumed FLOW dependence between pot line 32 and this line 27
  LOOP END


27-29 June 2016 Intel MIC Programming Workshop, LRZ 2016
ANTI and FLOW dependence

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

**FLOW (True) dependence**: read-after-write (RAW)
Statement i precedes j, and i uses a value that j computes
row: 1->2 and 2->4

```
1: x = 1;
2: y = x + 2;
3: x = z - w;
... 
4: x = y / z;
```
Example code: ver2

Look into the code!
Performance Comparison

ver1 Xeon: 17.4 GFlops  5.2% of the peak
ver1 KNC: 19.5 GFlops  1.9% of the peak

ver2 Xeon: 35.5 GFlops  11% of the peak  speedup: 2x
ver2 KNC: 60.8 GFlops  6.1% of the peak  speedup: 3.2x
Example code: ver2

Look into Advisor!
Example code: ver2

```c
void potential :: CalculateGravitationalPotential(int Xi, int Xj, int Xk, real_type &pot)
{
    int n;
    real_type local_p = 0.0;
    pot = 0.0;
    #pragma simd reduction(-: local_p),
    for (n=0; n < _nMasses; n++)
    {
        const real_type dx = particles[n].position[0] - (real_type) Xi;   //1flop
        const real_type dy = particles[n].position[1] - (real_type) Xj;   //1flop
        const real_type dz = particles[n].position[2] - (real_type) Xk;   //1flop
        const real_type dist = dx*dx+dy*dy+dz*dz;                        //5flops

        local_p -= G * particles[n].mass / sqrt(dist);                  //2flops+div+sqrt = 4flops
    }
    pot = local_p;
}
```
Example code: ver2 vectorization report

potential.optrpt

... LOOP BEGIN at potential.cpp(26,3)

remark #15328: vectorization support: gather was emulated for the variable this: strided by 4 [ potential.cpp(28,26) ]
remark #15328: vectorization support: gather was emulated for the variable this: strided by 4 [ potential.cpp(29,26) ]
remark #15328: vectorization support: gather was emulated for the variable this: strided by 4 [ potential.cpp(30,26) ]
remark #15328: vectorization support: gather was emulated for the variable this: strided by 4 [ potential.cpp(33,20) ]
remark #15305: vectorization support: vector length 4
remark #15309: vectorization support: normalized vectorization overhead 0.527
remark #15301: SIMD LOOP WAS VECTORIZED
remark #15460: masked strided loads: 4
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 83
remark #15477: vector loop cost: 23.250
remark #15478: estimated potential speedup: 3.540
remark #15488: --- end vector loop cost summary ---
LOOP END
AoS into SoA

Array of Structures (sub-optimal)

Structure of Arrays (optimal)

http://colfaxresearch.com/
Example code: ver3

Look into the code!
Example code: ver3

```c
potential :: potential(const int nMasses)
{
    _nMasses = nMasses;
    particles = new Particle;

    particles->pos_x = new real_type[_nMasses];
    particles->pos_y = new real_type[_nMasses];
    particles->pos_z = new real_type[_nMasses];
    particles->mass = new real_type[_nMasses];

    void potential :: CalculateGravitationalPotential(int Xi, int Xj, int Xk, real_type G)
    {
        int n;
        real_type local_p = 0.0;
        pot = 0.0;
        #pragma simd reduction(-: local_p)
        for (n=0; n < _nMasses; n++)
        {
            const real_type dx = particles->pos_x[n] - (real_type) Xi;       //1flop
            const real_type dy = particles->pos_y[n] - (real_type) Xj;       //1flop
            const real_type dz = particles->pos_z[n] - (real_type) Xk;       //1flop
            const real_type dist = dx*dx+dy*dy+dz*dz;                        //5flops

            pot -= G * particles->mass[n] / sqrt(dist);                     //2flops+div+sqrt = 4flops
        }
    }

    struct Particle
    {
        public:
            Particle() { init(); }
            void init()
            {
                pos_x = NULL;
                pos_y = NULL;
                pos_z = NULL;
                mass = NULL;
            }
            real_type *pos_x;
            real_type *pos_y;
            real_type *pos_z;
            real_type *mass;
    }
}
```

27-29 June 2016  Intel MIC Programming Workshop, LRZ 2016
Performance Comparison

ver1 Xeon: 17.4 GFlops  5.2% of the peak
ver1 KNC: 19.5 GFlops  1.9% of the peak

ver2 Xeon: 35.5 GFlops  11% of the peak  speedup: 2x
ver2 KNC: 60.8 GFlops  6.1% of the peak  speedup: 3.2x

ver3 Xeon: 35.2 GFlops  11% of the peak  speedup: 1x
ver3 KNC: 129.1 GFlops  13% of the peak  speedup: 2x
Example code: ver3

Look into Advisor!
Example code: ver3 vectorization report

potential.optrpt

... LOOP BEGIN at potential.cpp(31,3)

remark #15389: vectorization support: reference this has unaligned access [ potential.cpp(33,60) ]
remark #15389: vectorization support: reference this has unaligned access [ potential.cpp(34,60) ]
remark #15389: vectorization support: reference this has unaligned access [ potential.cpp(35,60) ]
remark #15388: vectorization support: reference this has aligned access [ potential.cpp(38,5) ]
remark #15381: vectorization support: unaligned access used inside loop body
remark #15305: vectorization support: vector length 4
remark #15309: vectorization support: normalized vectorization overhead 0.488
remark #15301: SIMD LOOP WAS VECTORIZED
remark #15442: entire loop may be executed in remainder
remark #15448: unmasked aligned unit stride loads: 1
remark #15450: unmasked unaligned unit stride loads: 3
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 88
remark #15477: vector loop cost: 21.500
remark #15478: estimated potential speedup: 3.900
remark #15488: --- end vector loop cost summary ---

LOOP END
Data Alignment

SSE 128 bit

16-byte aligned load

64-byte cache line

SSE 128 bit

16-byte unaligned load

64-byte cache line
Example code: ver4

Look into the code!
<table>
<thead>
<tr>
<th>Version</th>
<th>Processor</th>
<th>GFlops</th>
<th>% of the peak</th>
<th>Speedup</th>
<th>VL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ver1</td>
<td>Xeon</td>
<td>17.4</td>
<td>5.2%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ver1</td>
<td>KNC</td>
<td>19.5</td>
<td>1.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ver2</td>
<td>Xeon</td>
<td>35.5</td>
<td>11%</td>
<td>2x</td>
<td></td>
</tr>
<tr>
<td>ver2</td>
<td>KNC</td>
<td>60.8</td>
<td>6.1%</td>
<td>3.2x</td>
<td></td>
</tr>
<tr>
<td>ver3</td>
<td>Xeon</td>
<td>35.2</td>
<td>11%</td>
<td>1x</td>
<td></td>
</tr>
<tr>
<td>ver3</td>
<td>KNC</td>
<td>129.1</td>
<td>13%</td>
<td>2x</td>
<td></td>
</tr>
<tr>
<td>ver4</td>
<td>Xeon</td>
<td>35.2</td>
<td>11%</td>
<td>1x</td>
<td></td>
</tr>
<tr>
<td>ver4</td>
<td>KNC</td>
<td>130.7</td>
<td>13%</td>
<td>1x</td>
<td></td>
</tr>
<tr>
<td>ver4f</td>
<td>Xeon</td>
<td>69.9</td>
<td>21%</td>
<td>2x</td>
<td>8</td>
</tr>
<tr>
<td>ver4f</td>
<td>KNC</td>
<td>283.5</td>
<td>28%</td>
<td>2.1x</td>
<td>16</td>
</tr>
</tbody>
</table>
Thank you for your attention!

Contact: fabio.baruffa@lrz.de