KNL tools

Dr. Fabio Baruffa
fabio.baruffa@lrz.de
Profiling tools
Which tool do I use? A roadmap to optimization

We will focus on tools developed by Intel, available to users of the LRZ systems.

Again, we will skip the MPI layer.

VTune is a very rich tool, we will touch it only quickly.

We will dedicate more time (and hands-on) to Advisor.

more informations: Intel “The Parallel Universe”, 23, p.4
Profiling with Intel® VTune Amplifier XE

- Powerful tool for analyzing the node-level performance
  - Multiple programming languages (C/C++, Fortran, .NET, Java, Assembly)
  - Support for all latest Intel® processors (incl. Intel® MIC / Broadwell micro-architectures)
- Performance analysis at different levels
  - High-level (code analysis, parallelization 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

1. Compile code with `-g -O2` or `-g -O3`

2. Set the environment variables or use a wrapper script

3. Tweak code input for a short representative run

VTune
find top hotspots

Optimize
eliminate issues, reduce hotspot time

Compiler
identify issues in optimization report

Profiling tools
Performance overview

**Wall-clock time**

**Cumulative CPU time**

Performance bottlenecks are highlighted in red

Overall CPU usage
Threads behaviour

Function level profiling

Time line of the application

Profiling tools
Threads behaviour: locks and waits

Threads are spinning!

Threads sleeping

Useful work

Concurrency

Synchronization
Source code view

Source line
Closing remarks

The tool is useful and can be used to find:

- Hotspots in the code and possible bottlenecks
- Characterization of the parallelization efficiency
- Possible locks and spinning threads in the application

- More advanced profiling is provided using special kernel modules (memory bandwidth, hardware event-based sampling, ...)
- Instrumenting the code for reducing the amount of profiling part in the application
Roofline model
The *roofline model* allows to understand the performance limit of an application, based on operational intensity (algorithm specific) and on hardware specifics (memory bandwidth).

The expected performance is defined as:

$$P = \min(P_{\text{max}}, I b_s)$$

- $P_{\text{max}} =$ Applicable peak performance of a loop, assuming that data comes from L1 cache (this is not necessary $P_{\text{peak}}$

- $I =$ Computational/arithmetic intensity ("work" per Byte transferred) over the slowest data path utilized ("the bottleneck")

- $b_s =$ Applicable peak bandwidth of the slowest data path utilized
Roofline model

Peak performance of 2-socket Ivy-Bridge node

Peak: 448 GFlops/s
Roofline model

Peak performance of 2-socket Ivy-Bridge node

- Peak: 448 GFlops/s
- Stream BW: 78.5 GB/s
Roofline model

Peak performance of 2-socket Ivy-Bridge node

Peak: 448 GFlops/s
Stream BW: 78.5 GB/s
Roofline model

Peak performance of 2-socket Ivy-Bridge node

Peak: 448 GFlops/s
Stream BW: 78.5 GB/s
Core Performance: 22.4 GFlops/s

Roofline model
The core parameter behind the Roofline model is **Arithmetic Intensity**. Arithmetic Intensity is the ratio of total floating-point operations to total data movement (bytes). A **BLAS-1** vector-vector increment \( (x[i]+=y[i]) \) would have a very low arithmetic intensity of 0.0417 \( (N \text{ FLOPS} / 24N \text{ Bytes}) \) and would be independent of the vector size.

https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/
Roofline model: example daxpy.cpp

- **DAXPY**: $y[i] = a \times x[i] + y[i]$, double precision, $i=0,\ldots,N-1$

- **2 Flops** for each element of $x$ and $y$
  - well balanced: 1 multiply, 1 add
  - need to load $x[i]$ and $y[i]$ for each ‘$i$’: $2 \times 8 = 16$ bytes (a is the register)
  - need to write out $y[i]$: another 8 bytes
  - **Arithmetic intensity**: $2$ FLOPS/24 Bytes $= 1/12 \approx 0.083$
  - Speed of light performance (working from main memory)
    - on Ivy-Bridge with mem bw of 38 GB/s: $3.6$ GFlops/s
      - even the socket peak is $166.4$ GFlops/s
    - If $x$ and $y$ fit into cache, higher cache BW $\rightarrow$ higher performance
Intel® Advisor XE
Profiling with Intel® Advisor XE

• Modern HPC processors explore different level of parallelism: between the cores (multi-threading), within a core (vectorization)

• Adapting applications to take advantage of so high parallelism is defined often as code modernization

• The Intel® Advisor XE is a software tool for vectorization optimization and thread prototyping

• The tool guides the software developer to resolve issues during the vectorization process
Creating a new project via GUI

Interface similar to VTune
Setting up the application

Command-line parameters

Environment variables
Vectorization analysis workflow

Start

1. Run Survey

Edit & compile

2. Check the Trip-counts

Take Snapshot

4. Check Memory Access Patterns

Deeper-dive analysis

3. Check Dependencies

(Mark-up Loops)
5 Steps to efficient vectorization

1. Compiler diagnostics + Performance Data + SIMD efficiency information

2. Guidance: detect problem and recommend how to fix it

3. “Accurate” Trip Counts + FLOPs: understand utilization, parallelism granularity & overheads

4. Loop-Carried Dependency Analysis

5. Memory Access Patterns Analysis

Profiling tools
Profiling with Intel® Advisor XE

- How to improve performance
- ISA
- Hot-spots
- What prevents vectorization
- Report from the loop

Profiling tools
Profiling with Intel® Advisor XE

Vectorization informations

Number of vector registers

Traits

Application intensity
Profiling with Intel® Advisor XE

### Where should I add vectorization and/or threading parallelism?

<table>
<thead>
<tr>
<th>Ops</th>
<th>Vector Issues</th>
<th>Self Time</th>
<th>Total Time</th>
<th>Type</th>
<th>Why No Vectorization?</th>
<th>Instruction Set Analysis</th>
<th>Compiler Diagnostic Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start at GSimulation.cpp:139</td>
<td>1 Data type conversion</td>
<td>47.816s</td>
<td>47.816s</td>
<td>Scalar</td>
<td>vectorization problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start at GSimulation.cpp:136</td>
<td>1 Data type conversion</td>
<td>0.000s</td>
<td>0.012s</td>
<td>Scalar</td>
<td>outer loop was n...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start at GSimulation.cpp:165</td>
<td>1 Data type conversion</td>
<td>0.012s</td>
<td>0.012s</td>
<td>Scalar</td>
<td>outer loop was n...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Useful suggestion:**

Recommendations to enable vectorization

**Recommendation:** Use the smallest data type

The *source loop* contains data types of different widths. To fix: Use the smallest data type that gives the needed precision to use the entire vector register width.

**Example:** If only 16-bits are needed, using a short rather than an int can make the difference between eight-way or four-way SIMD parallelism, respectively.
Loop vectorized (ver2)

Profiling tools

Vectorization efficiency

Vector length

Loop analytics
Loop vectorized and improved efficiency

Additional informations
Memory access pattern

Stride distribution
Roofline analysis with Advisor - base

Profiling tools
Roofline analysis with Advisor - ver4

Profiling tools
Closing remarks: 6 steps vectorization methodology

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

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

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

4. Get advise using Intel® Advisor: use the vectorization analysis capability of the tool

5. Implement vectorization recommendations

more informations: https://software.intel.com/en-us/articles/vectorization-toolkit