Performance analysis tools: Intel® VTune™ Amplifier and Advisor

Dr. Luigi Iapichino
luigi.iapichino@lrz.de
Which tool do I use in my project? A roadmap to optimisation

After having considered the MPI layer, now let’s look at the node-level performance.

Three main features of HPC application for two tools:

- Threading parallelism;
- Vectorisation;
- Memory access.

Figure: courtesy Intel
Profiling with Intel® VTune™ Amplifier

- Powerful tool for analysing the node-level performance
  - Multiple programming languages (C/C++, Fortran, .NET, Java, Assembly, Python)
  - Support for all latest Intel® processors (incl. Intel® Xeon Phi and Skylake micro-architectures)
- 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)
- 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 optimisation

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
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
Threads behaviour: locks and waits

- Threads are spinning!
- Threads sleeping
- Useful work
- Concurrency
- Synchronization

Intel VTune Amplifier
Source code view (by clicking on a function)

### Source Code Snippet

```c
#define CHECKNAN
if (IsNaN(gl)) write(3, "glf=NaN!");

!-- Conservative variables

u(xrh) = rh*glf

u(xpg) = pg - vum
u(xoe) = e_con
u(xze) = xze - b_con
u(xze) = glf*pg*rh** (gamma-1_rk)  
u(gp) = -u

!-- Fluxes

w = w + v_con(3)

f(xrh) = v_con(3)* xrh*glf
f(xv) = w* v_con-b_con(3)*b_con-e_con(3)*e_con(3)

f(kpg) = wz* glf*pg*rh** (gamma-1_rk)  
f(kze) = glf*pg*rh** (gamma-1_rk)  
```

---

Source lines
Beyond code modernisation: a very rich tool

**Additional Capabilities**

- **Single Thread**
  Optimize single-threaded performance.

- **Multithreaded**
  Effectively use all available cores.

- **System**
  See a system-level view of application performance.

- **Media & OpenCL™ Applications**
  Deliver high-performance image and video processing pipelines.

- **HPC & Cloud**
  Access specialized, in-depth analyses for HPC and cloud computing.

- **Memory & Storage Management**
  Diagnose memory, storage, and data plane bottlenecks.

- **Analyze & Filter Data**
  Mine data for answers.

- **Environment**
  Fits your environment and workflow.
Workflow directions

Find your analysis direction

**Hotspots**
Want to find out where your app spends time and optimize your algorithms?

- Hotspots
- Memory Consumption

**Microarchitecture**
Want to see how efficiently your code is using the underlying hardware?

- Microarchitecture Exploration
- Memory Access

**Parallelism**
Want to assess the compute efficiency of your multi-threaded app?

- Threading
- HPC Performance Characterization
Types of analysis in VTune

Vtune is a very rich tool, offering a wide range of options. Most relevant analyses for HPC users:

- Hotspots (early overview)
- HPC performance
- Memory access
- Microarchitecture exploration (how is the application behaving in the core pipeline?)
- Storage device / Disk IO

...
**HPC Performance analysis**

**Effective CPU Utilization**: 8.0%

- Average Effective CPU Utilization: 20.465 out of 256
- Serial Time (outside parallel regions): 9.433s (62.9%)
- Top Serial Hotspots (outside parallel regions)

  This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Wait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Serial CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>__intel_mic_avx512f_memset</td>
<td>echo-openmp</td>
<td>0.830s</td>
</tr>
<tr>
<td>[Loop at line 2034 in evolve_bcy]</td>
<td>echo-openmp</td>
<td>0.510s</td>
</tr>
<tr>
<td>[Loop at line 1861 in evolve_bcy]</td>
<td>echo-openmp</td>
<td>0.500s</td>
</tr>
<tr>
<td>[Loop at line 1997 in evolve_bcy]</td>
<td>echo-openmp</td>
<td>0.470s</td>
</tr>
<tr>
<td>[Loop at line 1898 in evolve_bcy]</td>
<td>echo-openmp</td>
<td>0.440s</td>
</tr>
<tr>
<td>[Others]</td>
<td></td>
<td>6.150s</td>
</tr>
</tbody>
</table>

*N/A is applied to non-summarizable metrics.

**Parallel Region Time**: 5.567s (37.1%)

**Back-End Bound**: 28.2% of Pipeline Slots
- L2 Hit Bound: 0.8% of Clockticks
- L2 Miss Bound: 5.9% of Clockticks
- MCDRAM Bandwidth Bound: 0.0%
- DRAM Bandwidth Bound: 0.0%
- Bandwidth Utilization Histogram

**SIMD Instructions per Cycle**: 0.060
- Instruction Mix:
  - % of Packed SIMD Instr.: 34.0%
  - % of Scalar SIMD Instr.: 66.0%
- Top Loops/Functions with FPU Usage by CPU Time

**Main HPC diagnostics:**
- **CPU Utilisation**
  - Serial fraction
  - OpenMP performance
  - CPU usage hystogram

- **Memory Access**
  - Stalls by memory hierarchy
  - Bandwidth utilisation

- **Vectorization**
  - FPU utilisation
Focus on OpenMP

Hotspots; Bottom-up; Grouping → OpenMP region / Function / Call Stack

Useful view in the source of inefficiencies of OpenMP regions.
Memory access analysis

Where are my data, and how do they move?

Explore cache misses.

Optimise NUMA latency and scalability

Auto detect system bandwidth

Intel® Xeon Phi™ processor MCDRAM (high bandwidth memory) analysis supported.
Closing remarks

The tool is useful and can be used to detect:

- Hotspots in the code and possible bottlenecks
- Characterisation of the parallelisation efficiency
- Possible locks and spinning threads in the application

- More advanced profiling is provided using special kernel modules (memory bandwidth, hardware event-based sampling, ...)

- Advanced: instrumenting the code for reducing the amount of profiling part in the application

More on vectorisation, memory access and performance profiling: Intel® Advisor.
Intel® Advisor
Background: bring modern code to modern machines

Requirements to modern code:

- Vectorized (using AVX, AVX512)
- Threaded
- Efficient memory access

Today we go through some of the available tools for analyzing and profiling the code performance.
Profiling with Intel® Advisor

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

- Adapting applications to take advantage of so high parallelism is defined often as code modernisation

- Intel® Advisor is a software tool for vectorisation optimisation and thread prototyping

- The tool guides the software developer to resolve issues during the vectorisation process
Workflows in Advisor

Threading workflow
How to design a threading parallelization?
Interesting, but out of the scope of code optimization as discussed yesterday.

Vectorization workflow
• Survey of the target
• Trip counts and FLOPS → Roofline analysis
• Memory access pattern analysis
• Check for data dependencies
Why to optimize vectorization?

Even a good compiler can benefit greatly from vectorization optimization.

The compiler will not always auto-vectorize
➢ Check for loop-carried dependencies
➢ All clear? Force vectorization.
  C++ use: `pragma simd`, Fortran use: SIMD directive

Not all auto-vectorization is efficient
➢ Stride of 1 is more cache efficient than stride of 2 and greater. Use the memory access pattern analysis
➢ Consider data layout changes, as seen yesterday
Creating a new project via GUI

Interface similar to VTune
Four Steps to Efficient Vectorization

Intel® Advisor – Vectorization Advisor

1. Compiler diagnostics + Performance Data + Roofline Model + SIMD efficiency

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

- Issue: Peeled/Remainder loop(s) present
- All or some source loop iterations are not executing in the kernel loop. Improve performance by moving source loop iterations from peeled/remainder loops to the kernel loop. Read more at Vector Essentials Utilizing Full Vectors.

- Recommendation: Align memory access
  - Projected maximum performance gain: High
  - Projection confidence: Medium
  - The compiler created a peeled loop because one of the memory accesses in the source loop does not start at a data boundary. Align the memory access and tell the compiler your memory access is aligned.
  - Example aligning memory using a 32-byte boundary:
    ```c
    float *array; array = (float *)_mm_malloc(ARRAY_SIZE*sizeof(float), 32);
    // Somewhere else
    _assume_aligned(array, 32);
    // Use array in loop
    ```

3. Memory Access Patterns Analysis

4. Loop-Carried Dependency Analysis

<table>
<thead>
<tr>
<th>Site Name</th>
<th>Site Function</th>
<th>Site Info</th>
<th>Loop-Carried Dependencies</th>
<th>Strides Distribution</th>
<th>Access Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop_site100</td>
<td>runCranLoops</td>
<td></td>
<td>No information available</td>
<td>No information available</td>
<td>Mixed strides</td>
</tr>
<tr>
<td>loop_site100</td>
<td>runCranLoops</td>
<td></td>
<td>No information available</td>
<td>No information available</td>
<td>All unit strides</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problems and Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
</tr>
<tr>
<td>P4</td>
</tr>
<tr>
<td>P5</td>
</tr>
<tr>
<td>P6</td>
</tr>
<tr>
<td>P7</td>
</tr>
</tbody>
</table>
Profiling with Advisor: summary

Main features of the run

Overall quality of vectorisation

Hotspots for vectorisation
Profiling with Advisor: survey

- How to improve performance
- ISA
- Hot-spots
- What prevents vectorisation
- Report from the loop
Profiling with Advisor: survey – code analytics

- **Vectorisation informations**
- **Number of vector registers**
- **Traits**
- **Loop features**
Profiling with Advisor: survey - recommendations

Useful suggestion

Recommendations to enable/improve vectorisation
Example: loop vectorised but with low efficiency

[Image of a software interface showing loop analytics, vector length, and vectorization efficiency.]
Same loop, after extensive optimisation
Memory access pattern

Stride distribution

Memory footprint
Roofline model and performance

Peak performance of 2-socket Ivy-Bridge node

- How do I evaluate if a code / a loop / a function has potential for optimisation?
- What are my optimisation targets?
- When am I “satisfied” with the optimisation?

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

Intel Advisor
Roofline analysis: comparison

Physical Cores: 256  App Threads: 1  Self Elapsed Time: 2.080 s  Total Time: 2.080 s
Closing remarks: vectorisation methodology in six steps

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


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

5. Implement vectorisation recommendations

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

Intel® VTune™ Amplifier: rich tool for profiling and improving the node-level performance of HPC applications (and much more)


Intel® Advisor: focused on vectorisation and memory access. It allows to easily produce the roofline survey of an application.

Advisor website: https://software.intel.com/en-us/advisor

Acknowledgements

Thanks to Heinrich Bockhorst and Fabio Baruffa (Intel) for providing me with training material, as starting point for this tutorial.