Introduction to Performance Tuning & Optimization Tools

Ian A. Cosden, Ph.D.
Manager, HPC Software Engineering and Performance Tuning
Research Computing, OIT, Princeton University

icosden@princeton.edu

CoDaS-HEP Summer School
July 13, 2017
Outline

• What is Performance Tuning?
• Performance Tuning Workflow
• Software Tools
  • Hotspots: Intel VTune
  • OpenMP Scaling: Intel VTune
  • Vectorization: Intel Advisor
• Hands-on VTune Exercise
• Hardware Counters: VTune
What is Performance?

• “Performance is the degree to which a computing system meets expectations of the persons involved in it.” Doherty (1970)
  • Is it doing what I want?

• “Performance...is the effectiveness with which the resources of the host computer system are utilized toward meeting the objectives of the software system.” Graham (1973)
  • Is it utilizing the resources well?
What is Tuning?

- Algorithms (you/community)
- Programming (you/team)
- Performance Tuning (you/some poor sap)
- Operation of Supercomputers (the sysadmins)

Applicable to all domains
Where does Tuning fit?

- Performance should not be ignored during development
  - Algorithm selection
    - Vectorization and Parallelization
  - Data structures
  - Hardware awareness
- Tuning typically happens near the end
  - What if you invest time tuning code you never use?
- Principles and techniques applicable at any stage
- We’ll assume we are tuning an underperforming correct code
Tuning Tradeoff

- Readability
- Simplicity
- Maintainability
- Time

- Performance

Set goals - *Faster*
  - It could always be faster.
  - How much faster? Be specific.
  - Set an upper limit on your time investment
Performance Tuning Workflow

1. Prepare
2. Measure
3. Analyze
4. Hypothesize
5. Modify

The workflow is a circular process where you prepare, measure, analyze, hypothesize, and modify in a continuous cycle.
Hypothesis

• Why is my code slow?
  • CPU bound
  • Memory bound
  • I/O bound
  • Network bound
  • Unbalanced Workload (Parallel)

• What is the best I can expect?
  • CPU
  • Memory/Cache
  • I/O
  • Network
  • Parallel Scaling
• Need some kind of monitor

• Direct Measurement a.k.a. Instrumentation
  • Event is what we want to measure
  • Accurate
  • High overhead for frequent events
  • Ex. Tracing

• Indirect Measurement a.k.a. Sampling
  • Record system state at regular interval
  • Typically low overhead
  • Not every event recorded
  • Ex. Profiling

• Measurement can influence result
Measuring

- Measuring the performance of your code
  - You can’t fix what you can’t see
  - Find the “hotspots”
    - How much time is spent in each function
    - Not always where you think it is
    - Identify regions to optimize/parallelize
  - Hardware Performance
    - Vectorization, cache misses, branch misprediction, etc.

- Do it yourself
  - Put time calls around loops/functions
  - Only works if:
    - Done carefully during development
    - On a small code base

- Use a tool
Easy Measures - Linux

- **Total Runtime (“time”)**
  - Likely not enough
  - Critically important number

- **CPU usage (“top”)**
  - \(N \times 100\%\) cpu = \(N\) threads running full-tilt
  - \(100\% \neq 100\%\) peak theoretical performance
  - Useful for threading efficiency only

- **Slurm feedback (“seff”)**
  - Essentially same as top for a cluster
  - Only useful for threading/MPI efficiency and I/O
Performance Tools

• Many free and commercial products
• Site specific – but most places have the major products

• Intel Parallel Studio
  • VTune Amplifier XE
  • Advisor

• Many others: Allinea Forge (MAP), Tau, Intel Trace Analyzer and Collector, HPCToolkit, gprof, perf, gperftools, …
Intel VTune

- Intel VTune Amplifier XE
  - Commercial Profiler
  - Extraordinarily powerful (and complicated)
  - Nice GUI
- Source code profiling
- Shared memory only
  - Serial
  - OpenMP
  - MPI on single node
• Intel Advisor
  • Vectorization (and threading) advisor
• Identifies loops to target for vectorization
  • Provides efficiency statistics and tips for improvement
• Roofline Analysis (new)

• In many ways it’s a simple profiler + GUI vec-report
Using Profilers

• Strength of all profiling tools revolves around ability to trace performance back to source code
  • Need to include debug symbols in executable
    • -g flag

• Use release-build optimization flags
  • Ex. -O3, -xhost (Intel)
  • Don’t waste time optimizing code the compiler can do automatically!

• Sometimes the compiler will optimize out useful regions
  • Recommendations:
    • -debug inline-debug-info, -debug full (Intel)

• For difficult problems use more than one profiler
A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune
VTune Example

• Code compiled with -g to enable source code profiling

• Example serial C++ code with:
  • No multithreading
  • No MPI
  • Some I/O

• Use “Basic Hotspots” analysis
  • This will be subject of hands-on exercise later
VTune – Profile Result

Elapsed Time: 11.193s
- CPU Time: 10.760s
- Total Thread Count: 1
- Paused Time: 0s

Top Hotspots
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>CPU Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic-scale-force</td>
<td>md.out</td>
<td>9.073s</td>
</tr>
<tr>
<td>__libc_start_main</td>
<td>libc.so.6</td>
<td>1.176s</td>
</tr>
<tr>
<td>stdio_open</td>
<td>md.out</td>
<td>0.271s</td>
</tr>
<tr>
<td>__libc_close</td>
<td>libc.so.6</td>
<td>0.080s</td>
</tr>
<tr>
<td>stdio_close</td>
<td>md.out</td>
<td>0.050s</td>
</tr>
<tr>
<td>[Others]</td>
<td>N/A*</td>
<td>0.100s</td>
</tr>
</tbody>
</table>

*MNA is applied to non-measurable metrics.

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 value.

Collection and Platform Info
This section provides information about this collection, including result set size and collection platform data.

Application Command Line: /home/kosden/svn/IntrfPerfTuningTrunk/example_codesimple_md_code/md.out input.txt
Collection Container: 2.10.0-922.36.2.el7.x86_64 NAME=Rpm-Target-Profile-Version-5.1.18 PRETTY_NAME=Snapshot labs:3.2 Average-IP: 40.223
Bottom-up (Loops)
Source Code Hotspots (line-by-line)
### Same Real Code – Top-down

<table>
<thead>
<tr>
<th>Function Stack</th>
<th>Effective Time by Utilization</th>
<th>Spin Time</th>
<th>Overhead Time</th>
<th>Effective Time by Utilization</th>
<th>CPU Time: Self</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td><img src="image" alt="Graph" /></td>
<td>0%</td>
<td>0%</td>
<td><img src="image" alt="Graph" /></td>
<td>0%</td>
</tr>
<tr>
<td>start</td>
<td><img src="image" alt="Graph" /></td>
<td>0%</td>
<td>0%</td>
<td><img src="image" alt="Graph" /></td>
<td>0%</td>
</tr>
<tr>
<td>_libc_start_main</td>
<td><img src="image" alt="Graph" /></td>
<td>0%</td>
<td>0%</td>
<td><img src="image" alt="Graph" /></td>
<td>0%</td>
</tr>
<tr>
<td>main</td>
<td><img src="image" alt="Graph" /></td>
<td>0%</td>
<td>0%</td>
<td><img src="image" alt="Graph" /></td>
<td>0%</td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
</tbody>
</table>
A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
   • VTune

2. OpenMP Scaling
   • VTune

3. Vectorization
   • Advisor

4. Cache performance & Hardware Metrics
   • VTune
**Strong Scaling**

- **Strong Scaling**
  - Fixed problem size
  - Measure how solution time decreases with more processors

![Diagram showing Strong Scaling]

- **1 Proc**
  - Time: $T$ sec
  - Size: $N$

- **3 Procs**
  - Time: $T/3$ sec
  - Size: $N$
Scaling and Performance

Example Strong Scaling of MPI and OpenMP code

![Graph showing speed up over serial with number of MPI processes or OpenMP threads.]
OpenMP Threading Efficiency

• Example code was seeing ~3x speed up with 16 threads
  • Why? How to fix?

• Zoom in on threading timeline in VTune:
  • Brown = good
  • Green (idle) and Orange (overhead) = bad
A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune
Example code compiled with intel compiler with flags: --no-vec and --no-simd

- Purposefully turns off auto-vectorization
- Useful with debugging or performance benchmarking only
  - Or faking poorly vectorized code

Then built normally
- --g (debugging symbols) and all release flags

Ran on Broadwell node
- AVX2 instruction set (FMA)
- Two, 256-bit vector registers
  - Single precision (32 bit) floating point numbers: 8 per register possible
Higher instruction set architecture (ISA) available

Consider recompiling your application using a higher ISA.

// Components of the gravity force on particle i
float Fx = 0, Fy = 0, Fz = 0;

// Loop over particles that exert force: vectorization expected here
#pragma vector aligned
for (int i = 0; i < nParticles; i++) {
    // Avoid singularity and interaction with self
    const float softening = 1e-20;
    // Newton's law of universal gravity
    const float dx = particle.x[i] - particle.x[i];
    const float dy = particle.y[i] - particle.y[i];
    const float dz = particle.z[i] - particle.z[i];
    const float drSquared = dx*dx + dy*dy + dz*dz + softening;
    const float drSqrt = sqrtf(drSquared);
    const float drSqrtRecip = 1.0f / drSqrt;
Vectorized - Advisor

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

- **Vectorized Loops**
  - Type: Vectorized
  - Why No Vectorization?
    - AVX: 20%
    - Not vectorized
    - Instruction Set: AVX
    - Data Type: FMA, Square Roots
  - FMA: Square Roots
    - Extracts: 16
    - nbody.cc30

- **Instruction Mix**
  - Memory: 4
  - Compute: 17
  - Other: 2
  - Number of Vector Registers: 14

**Vectorization Gain**
- 7.71x
- ~96%
Hands-on Exercise

• Goal: Identify hotspots in sample code
  • Targets for parallelization and/or optimization

• Test code has 4 functions: mm[1-4]
  • Each does a different version of matrix-matrix multiplication C=AxB

• Each function is called a different number of times
  • Approximate call frequency:
    • mm1: 10%
    • mm2: 20%
    • mm3: 30%
    • mm4: 40%
  • Where should we optimize?
Adroit Test Set Up

- Enable X11 forwarding
  - `ssh -Y -C <user>@adroit.princeton.edu`
  - Will need local xserver (XQuartz for OSX, Xming for Windows)
- Clone the repo

```
https://github.com/cosden/CoDaS-HEP-Perf-Tuning
```

- Follow instructions in repo Readme.md

- What functions are most/least expensive?
- Change threshold values to select only the fastest function
Hands-on Discussion

• During a break feel free to try
  • GUI on head node
  • Other analyses on compute node
    • advanced-hotspots
    • general-exploration
  • Your code?
Examples

A handful of real(ish) codes to demonstrate profiling:

1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune
Hardware Counters

• Basic profilers “sample” the current program counter to see what instruction the CPU is currently executing.

• Then, after collecting enough samples, can give a good approximation as to what the program was doing.

• Performance Hardware Counters
  • Special purpose registers to track hardware events

• VTune can access hardware counters
  • Other profiling tools can access too, many build on top of open-source PAPI

• Literally hundreds of possible events to measure
  • Different in name and meaning for different chips
Hardware Counter Code Example

• 2D Matrix multiplication from before – mm1
  • 3 nested for loops
  • Let’s assume we don’t know why it is slow or even if it is slow
Double click here
opens source below:

```c
for (int k = 0; k < matrix_size; k++)
{
    C[i][j] += A[i][k] * B[k][j];
}
```

Clearly there is one very expensive line inside nested for loops. *But why?*
All flagged in pink:

Memory Bound: 49.8%
DRAM Bound: 0.105
Memory Latency: 0.654
Hardware Counters

• Remember every generation of chips has different counters
• VTune developers have calculated derived metrics that they feel best represent real problems
  • For example the sandybridge/ivybridge hardware counter: MEM_LOAD_UOPS_RETIRED.LLC_MISS_PS may not truly indicate if the CPU was stalled waiting for the data.
  • Colored in pink when they represent a value that might warrant investigation.

• My experience with cache misses:
  • Memory Bound, DRAM Bound, & Memory Latency together are very good indicators that cache is not being used well.

• How to fix cache misses?
  • In this case: Reorder nested loops (mm2)
After Loop Reordering

Change in metrics:

- **Memory Bound**: 49.8% → 24.4%
- **DRAM Bound**: 0.105 → 0.032
- **Memory Latency**: 0.654 → 0.880

Elapsed Time: 132s → 2.6s

- **Elapsed Time**: 2.602s
  - Clockticks: 7,446,011,169
  - Instructions Retired: 13,616,020,424
  - CPI Rate: 0.547
  - Mutex Contention: 0.981
  - Front-End Bound: 8.9%
  - Bad Speculation: 0.3%
  - Back-End Bound: 42.2%

Identify slots where no uOps are delivered due to a lack of required resources for accepting more uOps in the pipeline. Describe a portion of the pipeline where the out-of-order scheduler dispatches ready uOps into their respective slot but uOps get retired according to program order. Stalls due to data cache misses or stalls due to the overdriven data path are highlighted.

- **Memory Bound**: 24.4%
  - The metric value is high. This can indicate that a significant fraction of execution pipeline slots could be spent on handling data cache misses or other memory latency issues.

- **L1 Bound**: 0.051
  - This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache type might be caching false 4k aliasing between loads and stores. Use the source assembly view to identify the aliasing loads and stores, and then adjust your data layout so that the loads and stores no longer alias.

- **L2 Bound**: 0.102
  - This metric shows how often machine was stalled on L2 cache. Avoiding cache misses (L1 misses/L2 hits) will improve the latency and increase performance.

- **L3 Bound**: 0.109
  - This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance.

- **DRAM Bound**: 0.032
  - Memory Bandwidth: 0.011
  - Memory Latency: 0.880
  - Store Bound: 0.000

- **Core Bound**: 17.8%
  - Retiring: 48.7%
  - Total Thread Count: 1
  - Paused Time: 0%

The change in metrics shows an improvement in most performance metrics, especially in memory-bound and DRAM-bound performance. The elapsed time has significantly reduced from 132s to 2.6s.
Results

- Reordering nested loops resulted in a better use of cache and a 50x speedup
- Clues in the VTune metrics (hardware counters)
- Note that "optimized" case still showed MANY pink highlighted metrics
  - Something will always be limiting performance
  - Always requires interpretation and consideration
Final Remarks

• Measurement is key
• Tools are helpful
• Fast computing of wrong results is completely irrelevant!
  • Have correctness test(s)
  • Test after each modification
• Don’t fall into the “tuning trap”
  • Remember what really matters: Total Runtime