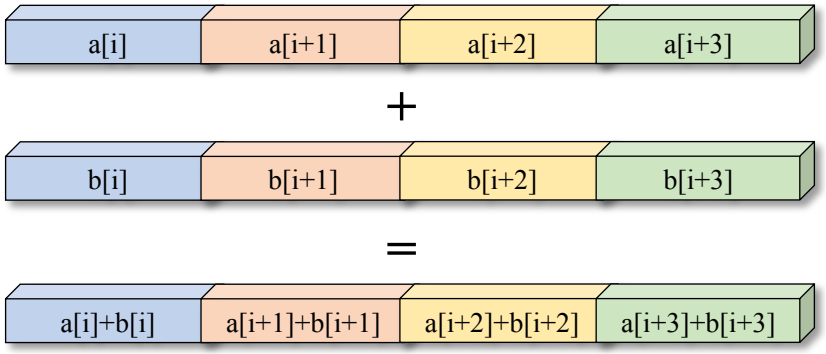
Introduction to Performance Tuning & Optimization Tools



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 26, 2018

Outline

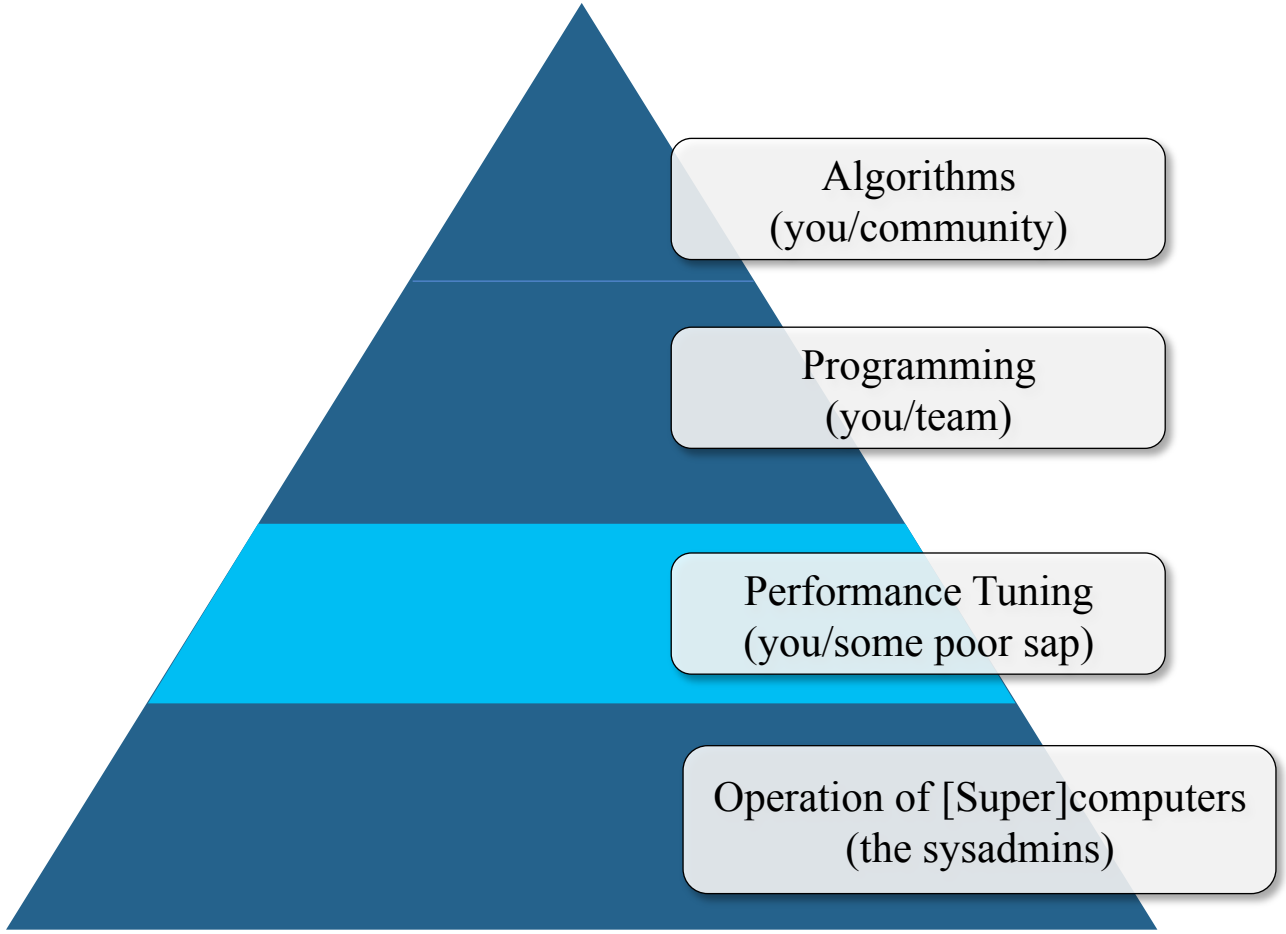
- What is Performance Tuning?
- Performance Tuning Workflow
- Software Tools
  - Hotspots: Intel VTune
  - OpenMP Scaling: Intel VTune
  - Vectorization: Intel Advisor
- Hands-on VTune Exercise
- Hardware Counters: VTune

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?



Applicable to all domains

Where does Tuning fit?

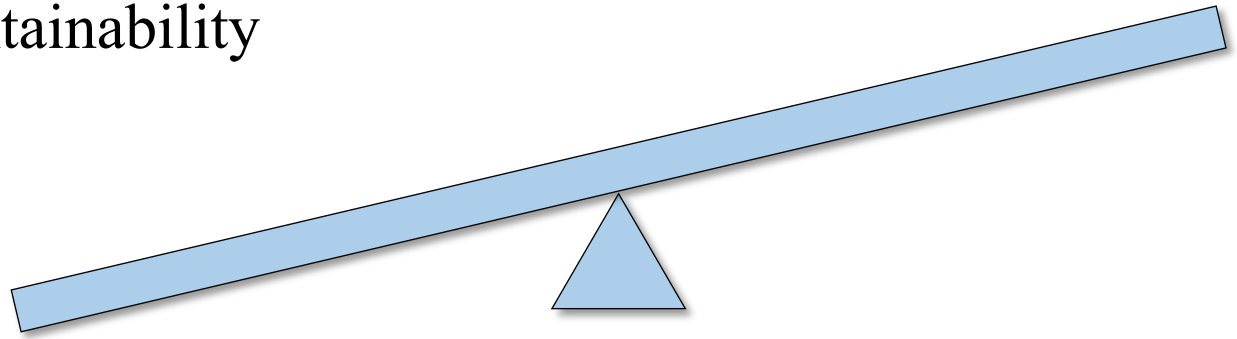
- Performance should not be ignored during development
  - Algorithm selection
    - Vectorization and Parallelization
  - Data structures
  - Hardware awareness
- *Tuning* typically happens near the end
  - What if you invest time tuning code you never use?

- Principles and techniques applicable at any stage
- We'll assume we are tuning an underperforming correct code

Tuning Tradeoff

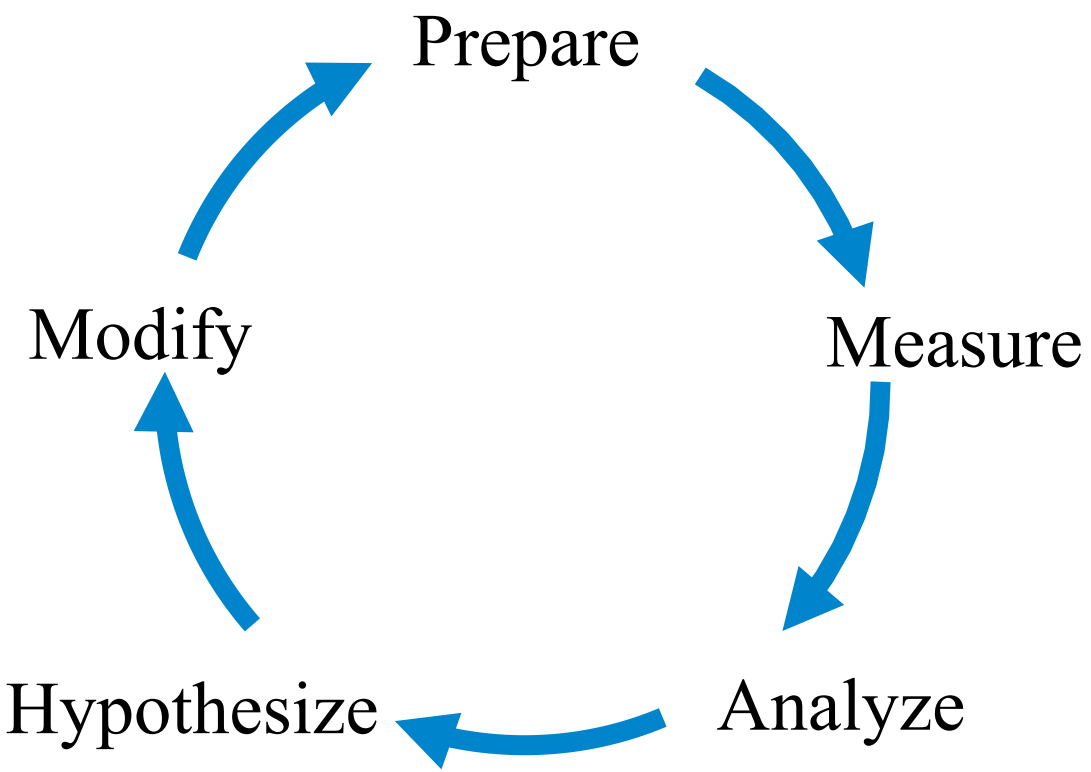
Readability
Simplicity
Maintainability
Time

Performance



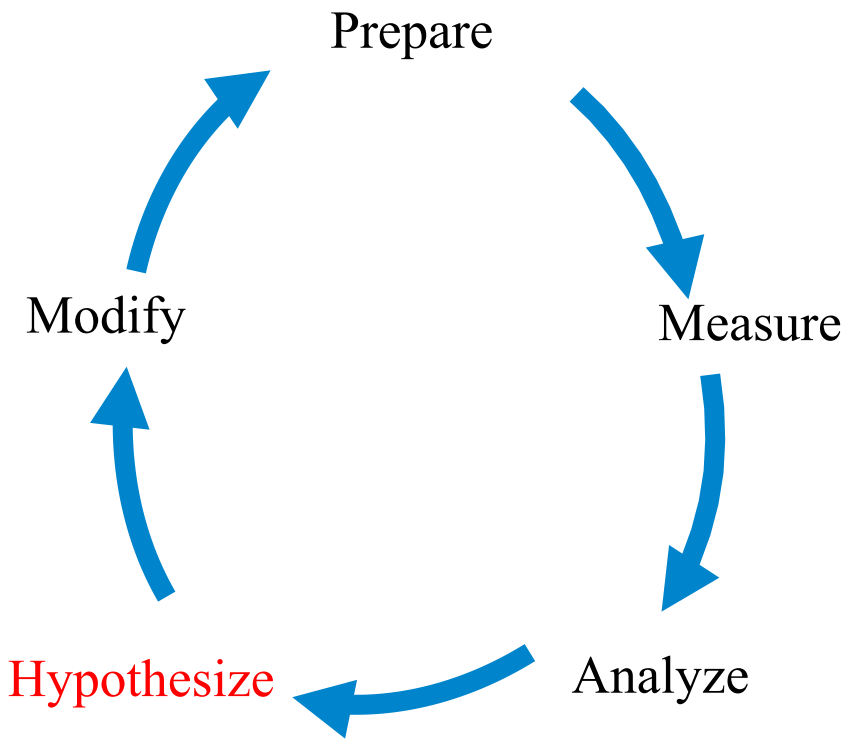
- Set goals - *Faster*
  - It could always be faster.
  - How much faster? Be specific.
  - Set an upper limit on your time investment

Performance Tuning Workflow



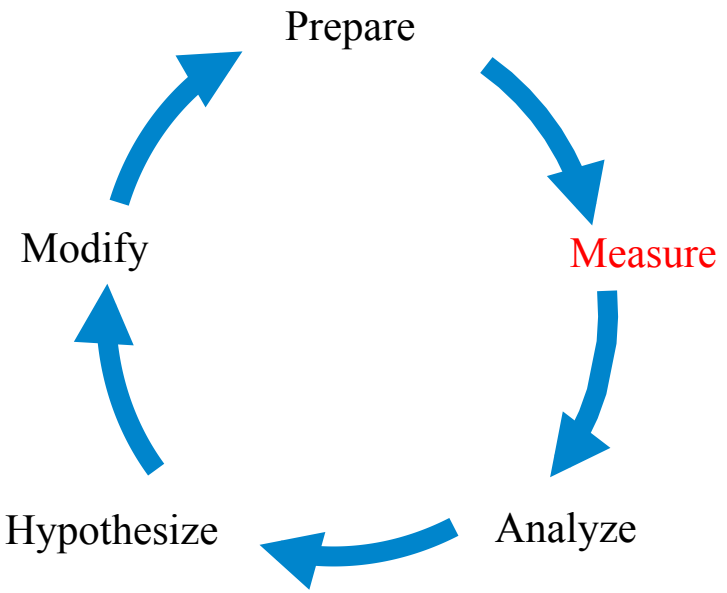
Hypothesis

- 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



Measure

- 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

```
$ time ./app-ICC 65536 > output.log

real    0m47.560s
user    0m47.441s
sys     0m0.000s
```

Un-tuned

```
$ time ./app-ICC 65536 > output.log

real    0m0.643s
user    0m19.810s
sys     0m0.044s
```

Tuned

- CPU usage ("top")
  - N*100% cpu = N threads running full-tilt
  - 100% ≠ 100% peak theoretical performance
  - Useful for threading efficiency only

```
top - 08:32:06 up 8 days, 19:47,  7 users,  load average: 0.72, 0.16, 0.09
Tasks: 771 total,   2 running, 769 sleeping,   0 stopped,   0 zombie
%Cpu(s): 42.8 us,  0.1 sy,  0.0 ni, 57.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 26372923+total, 25595838+free,  2061328 used,  5709524 buff/cache
KiB Swap:  8191996 total,  8191996 free,        0 used. 26107923+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3863 icosden   20   0  857096   5788   1620 R  1201  0.0   1:04.35 app-ICC
```

Performance Tools

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

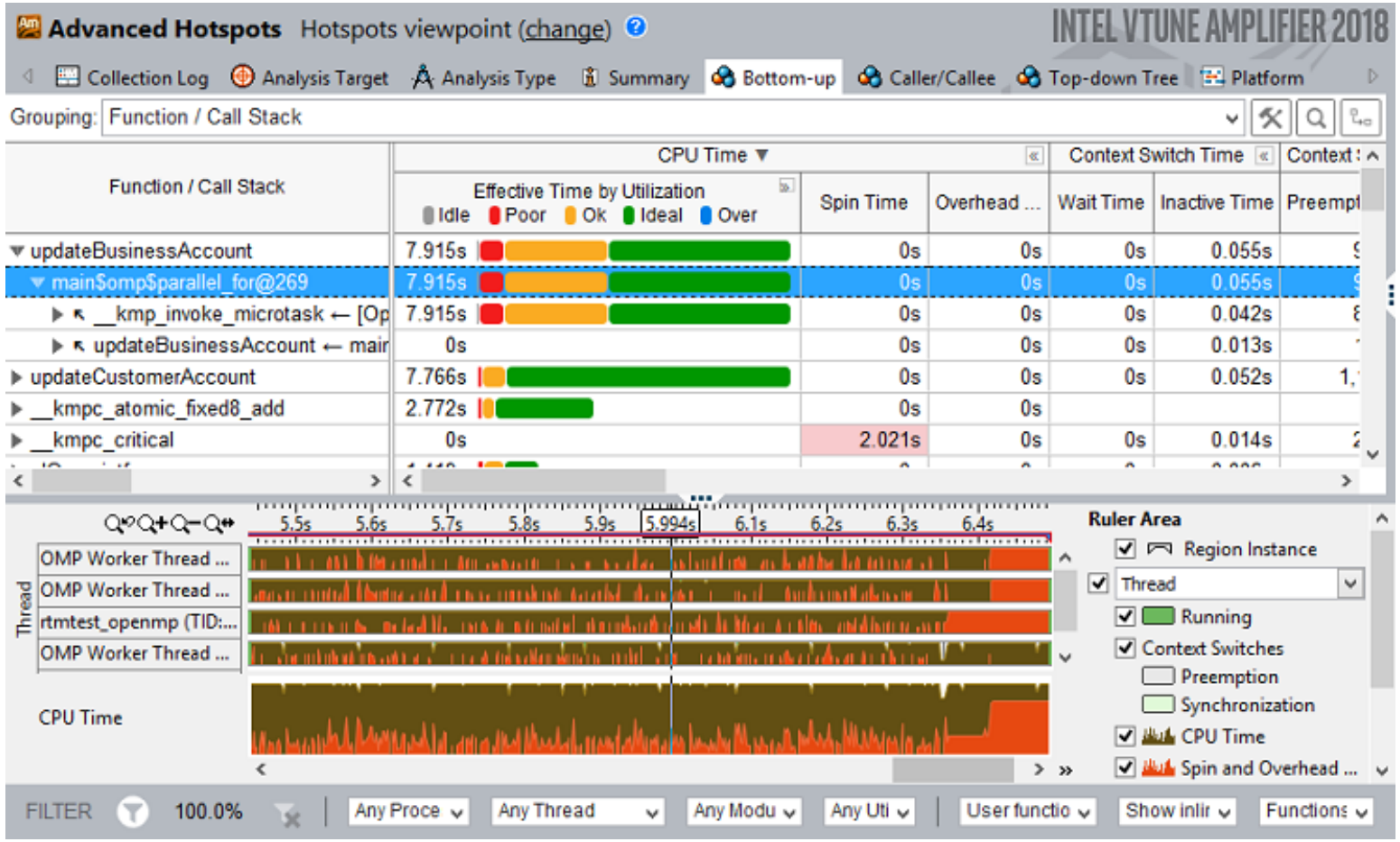
- Intel Parallel Studio
  - VTune Amplifier XE
  - Advisor

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

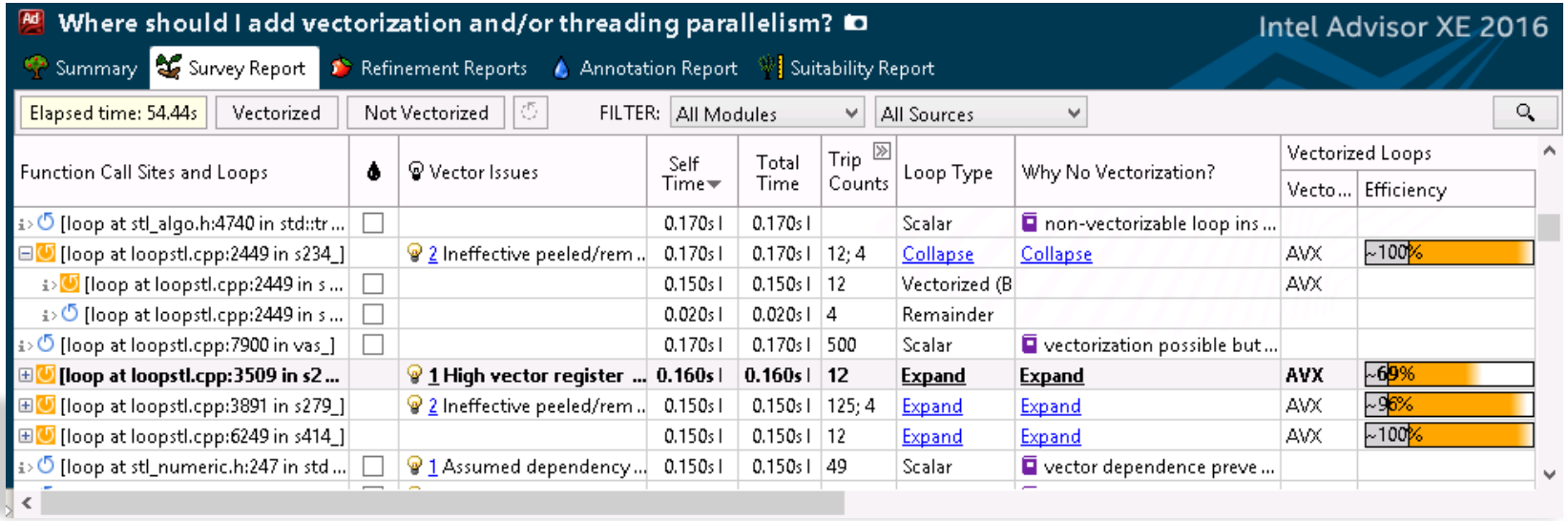
Intel VTune

- Intel VTune Amplifer XE
  - Commercial Profiler
  - Extraordinarily powerful
    (and complicated)
  - Nice GUI

- Source code profiling
- Shared memory only
  - Serial
  - OpenMP
  - MPI on single node



Intel Advisor



- Intel Advisor
  - Vectorization (and threading) advisor
- Identifies loops to target for vectorization
  - Provides efficiency statistics and tips for improvement
- Roofline Analysis

- 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

Examples

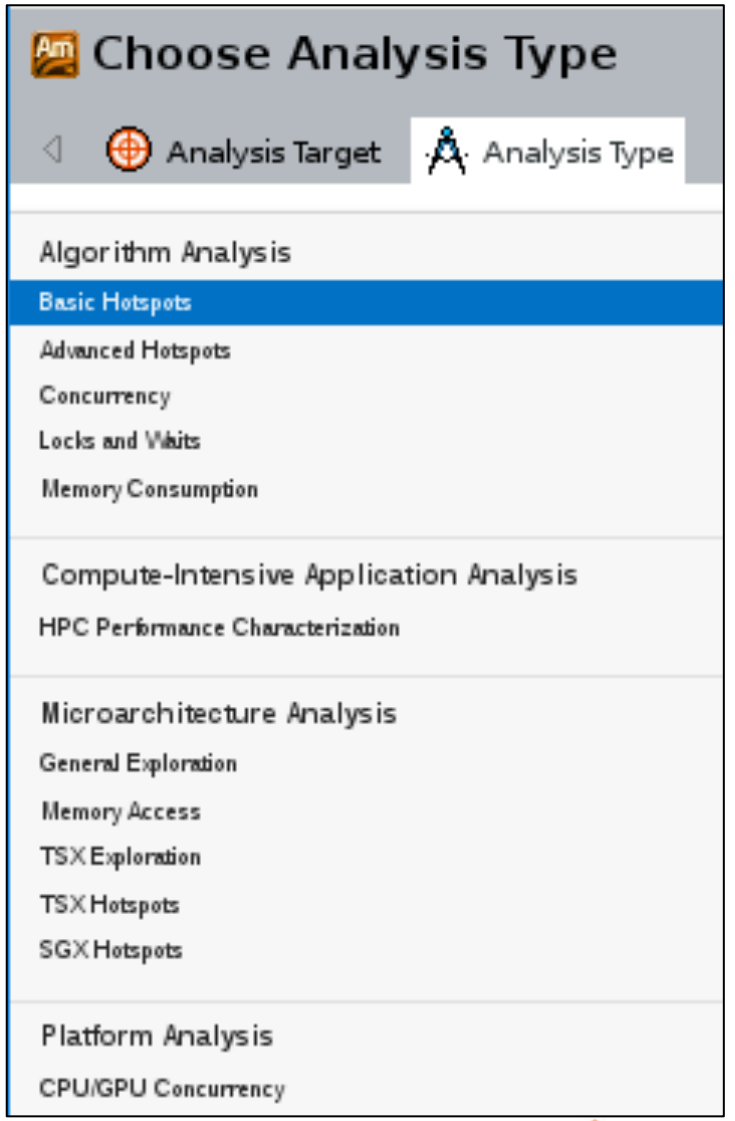
A handful of real(ish) codes to demonstrate profiling:

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

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

/home/icosden/intel/amplxe/projects/simple_md - Intel VTune Amplifier

Welcome | r004hs

Basic Hotspots Hotspots by CPU Utilization viewpoint (change)

INTEL VTUNE AMPLIFIER 2018

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Elapsed Time: 12.026s

- CPU Time: 11.850s
- Total Thread Count: 1
- Paused Time: 0s

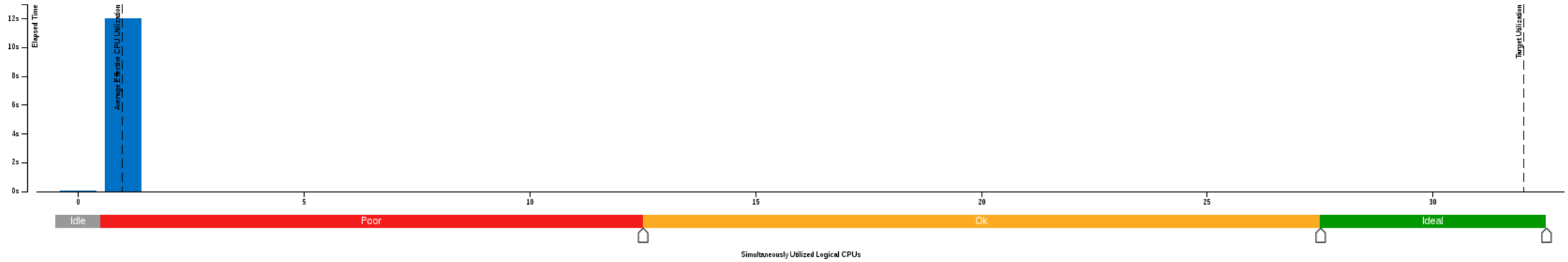
Top Hotspots

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

Function	Module	CPU Time
Atom::calc_force	md.out	6.749s
__libm_round_y8	md.out	2.231s
std::vector<double, std::allocator<double>>::operator[]	md.out	1.209s
__powr8i4	md.out	0.870s
std::pow	md.out	0.460s
[Others]		0.331s

Effective CPU Utilization Histogram

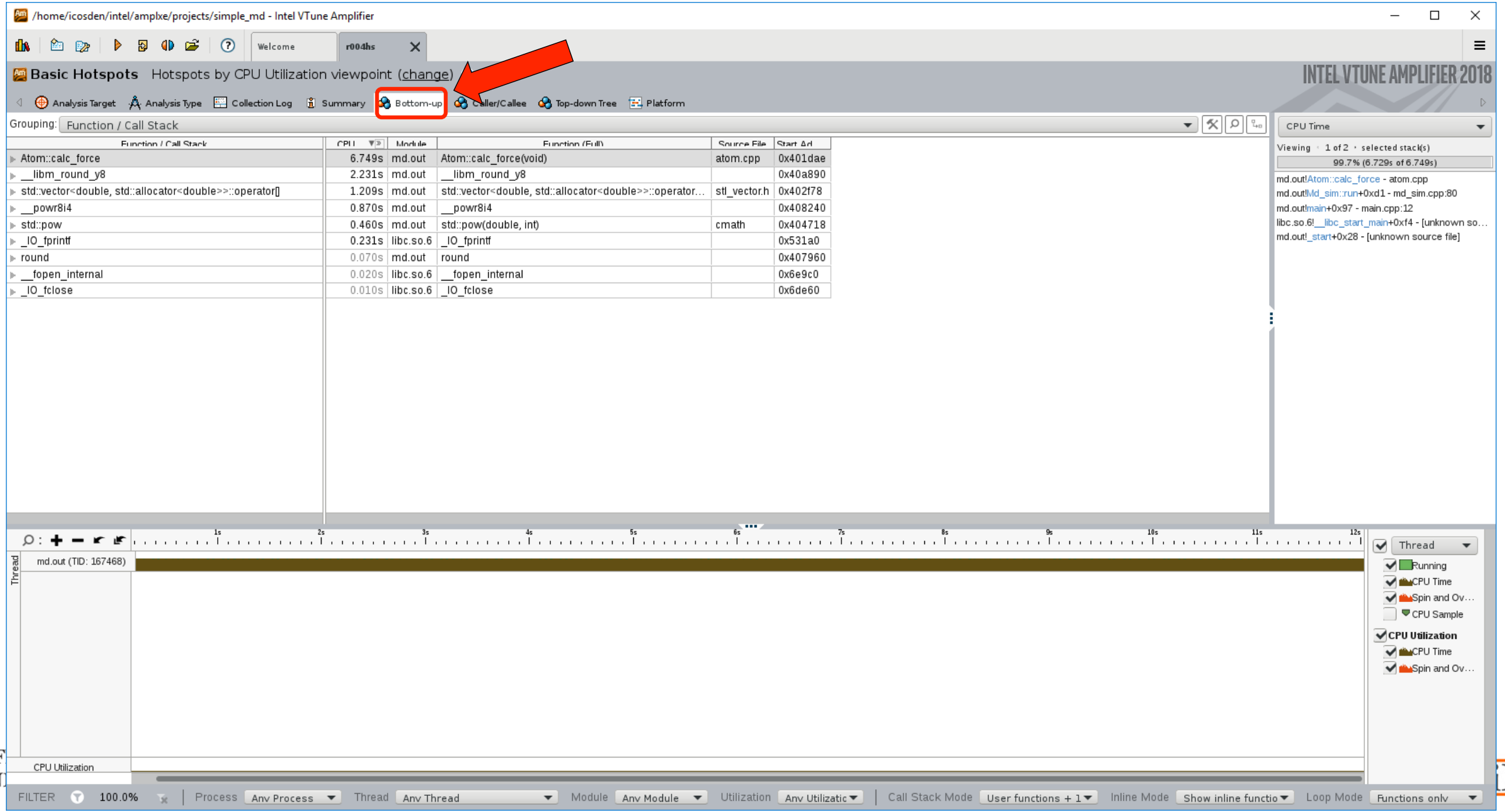
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



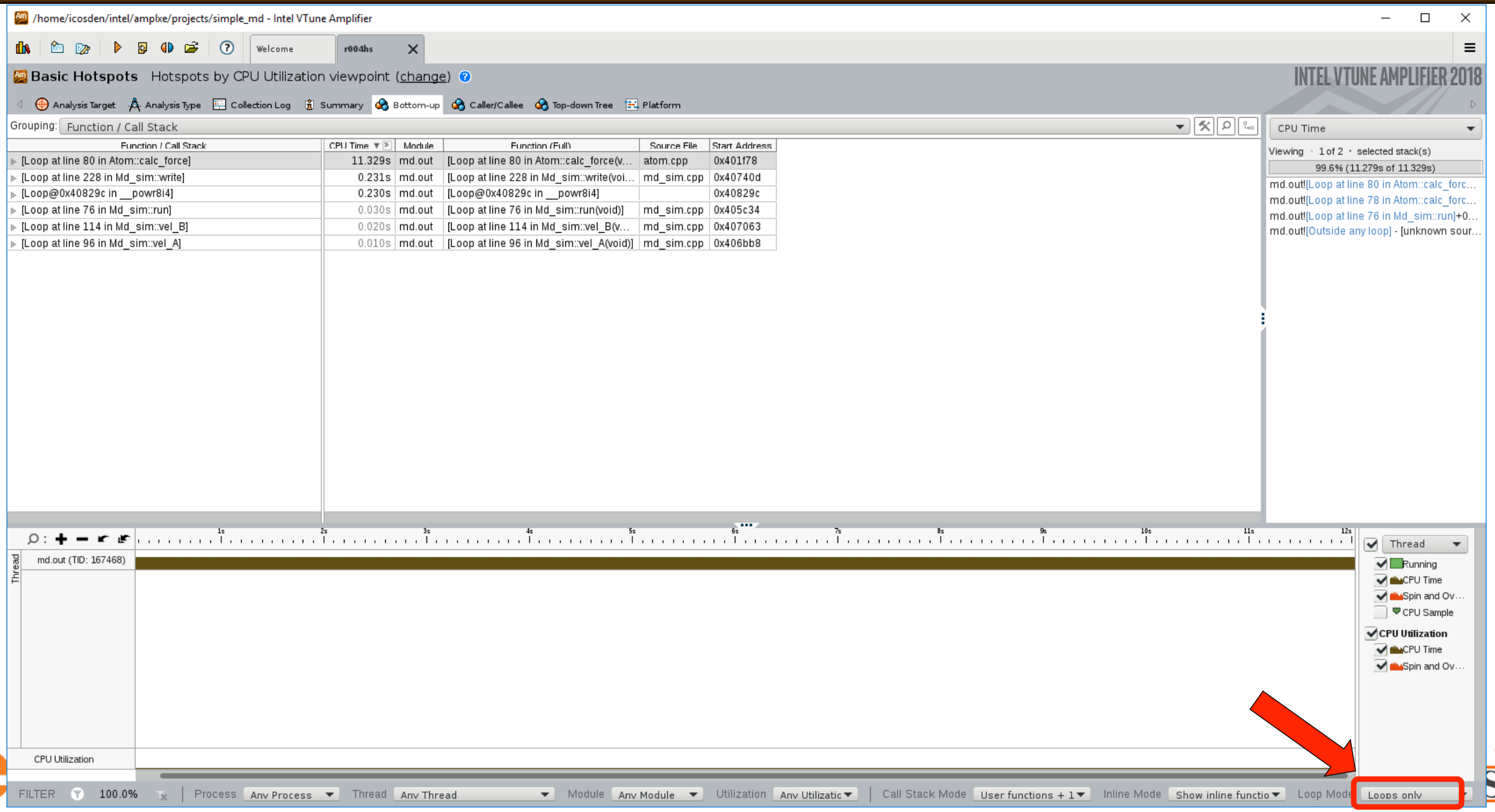
Collection and Platform Info

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

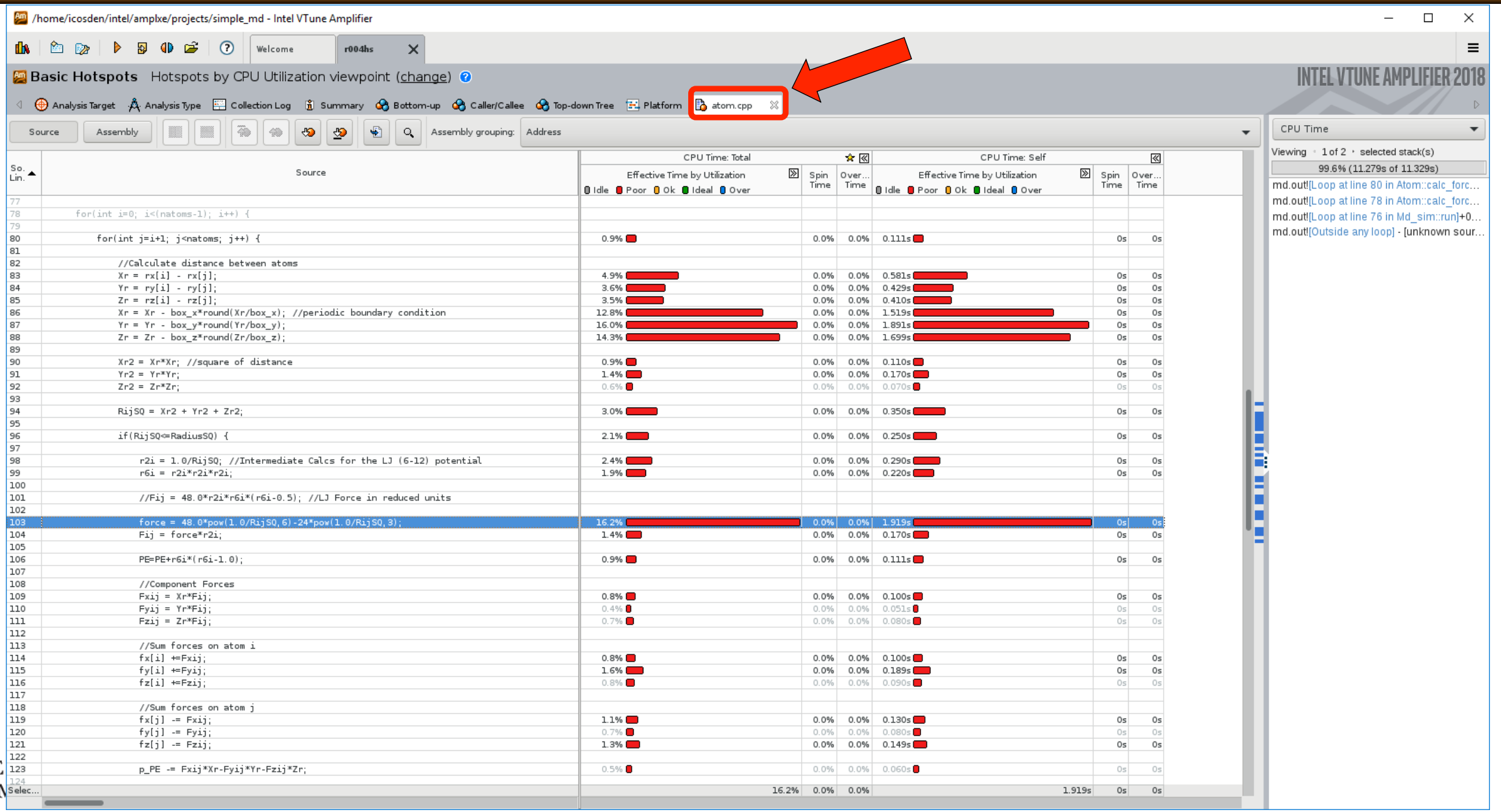
Bottom-up Tab (Functions)



Bottom-up (Loops)

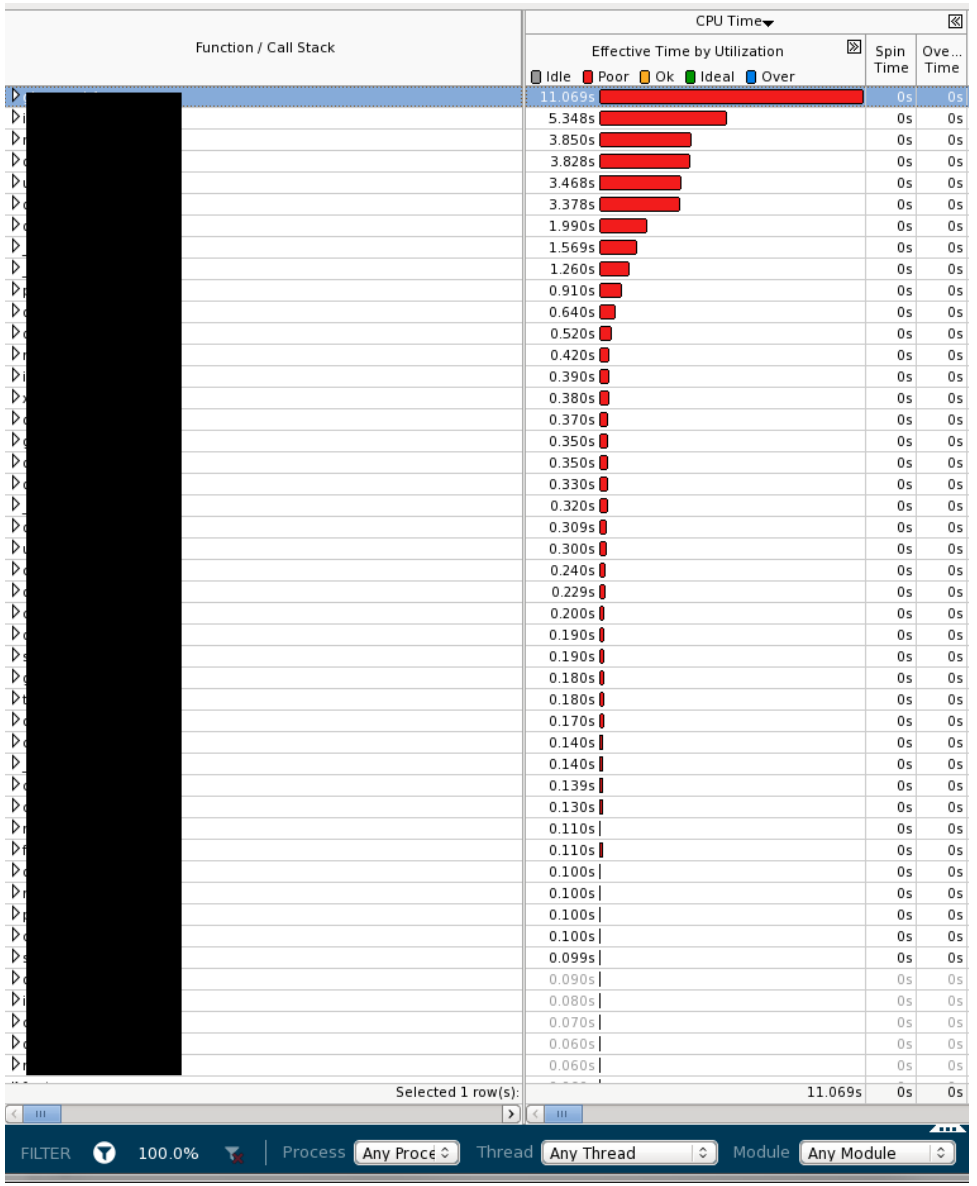


Source Code Hotspots (line-by-line)

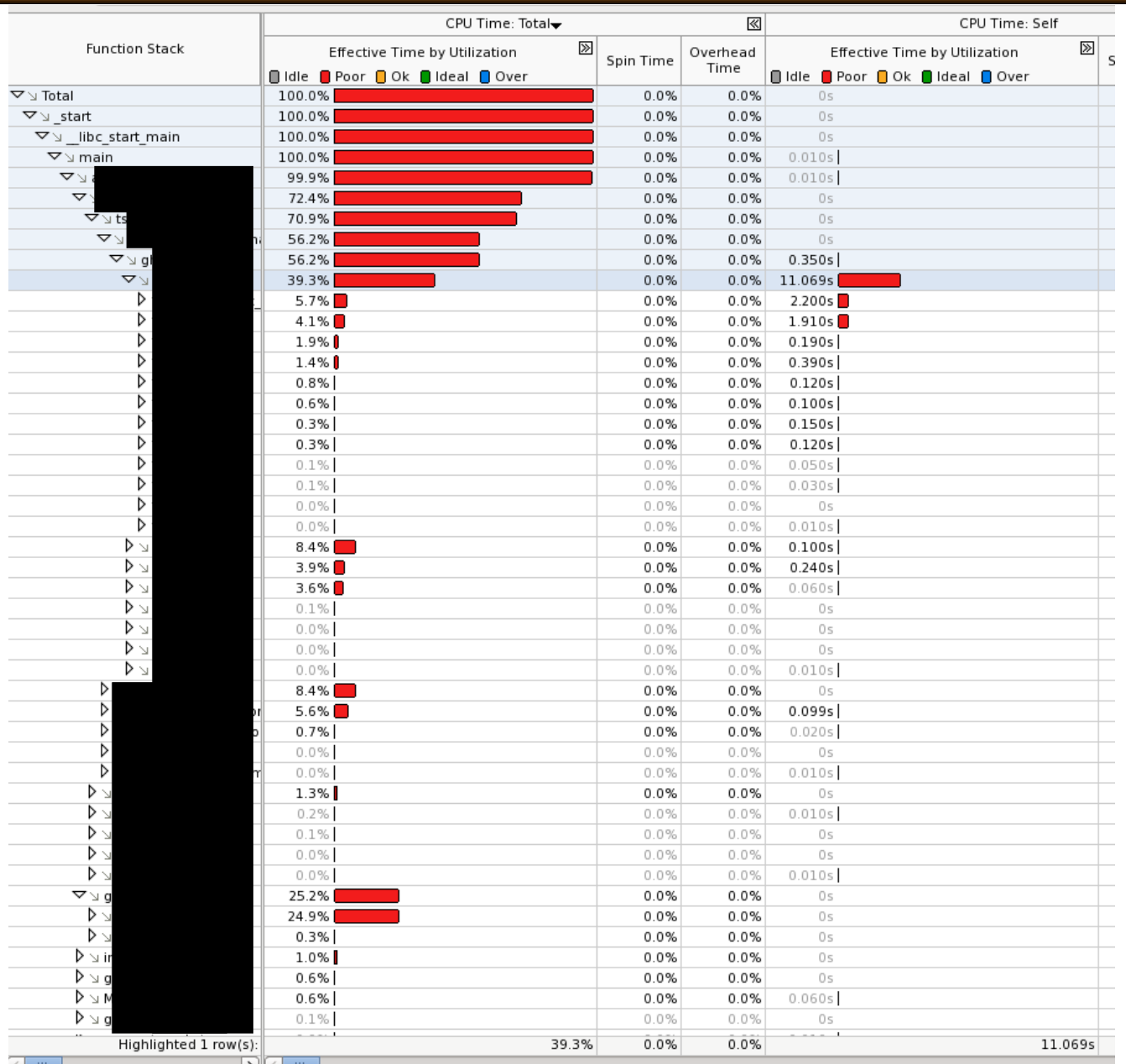


So. Lin.	Source	CPU Time: Total — Effective Time by Utilization (Idle Poor Ok Ideal Over)	CPU Time: Total — Spin Time	CPU Time: Total — Over... Time	CPU Time: Self — Effective Time by Utilization (Idle Poor Ok Ideal Over)	CPU Time: Self — Spin Time	CPU Time: Self — Over... Time
77							
78	for(int i=0; i<(natoms-1); i++) {						
79							
80	for(int j=i+1; j<natoms; j++) {	0.9%	0.0%	0.0%	0.111s	0s	0s
81							
82	//Calculate distance between atoms						
83	Xr = rx[i] - rx[j];	4.9%	0.0%	0.0%	0.581s	0s	0s
84	Yr = ry[i] - ry[j];	3.6%	0.0%	0.0%	0.429s	0s	0s
85	Zr = rz[i] - rz[j];	3.5%	0.0%	0.0%	0.410s	0s	0s
86	Xr = Xr - box_x*round(Xr/box_x); //periodic boundary condition	12.8%	0.0%	0.0%	1.519s	0s	0s
87	Yr = Yr - box_y*round(Yr/box_y);	16.0%	0.0%	0.0%	1.891s	0s	0s
88	Zr = Zr - box_z*round(Zr/box_z);	14.3%	0.0%	0.0%	1.699s	0s	0s
89							
90	Xr2 = Xr*Xr; //square of distance	0.9%	0.0%	0.0%	0.110s	0s	0s
91	Yr2 = Yr*Yr;	1.4%	0.0%	0.0%	0.170s	0s	0s
92	Zr2 = Zr*Zr;	0.6%	0.0%	0.0%	0.070s	0s	0s
93							
94	RijSQ = Xr2 + Yr2 + Zr2;	3.0%	0.0%	0.0%	0.350s	0s	0s
95							
96	if(RijSQ<=RadiusSQ) {	2.1%	0.0%	0.0%	0.250s	0s	0s
97							
98	r2i = 1.0/RijSQ; //Intermediate Calcs for the LJ (6-12) potential	2.4%	0.0%	0.0%	0.290s	0s	0s
99	r6i = r2i*r2i*r2i;	1.9%	0.0%	0.0%	0.220s	0s	0s
100							
101	//Fij = 48.0*r2i*r6i*(r6i-0.5); //LJ Force in reduced units						
102							
103	force = 48.0*pow(1.0/RijSQ,6)-24*pow(1.0/RijSQ,3);	16.2%	0.0%	0.0%	1.919s	0s	0s
104	Fij = force*r2i;	1.4%	0.0%	0.0%	0.170s	0s	0s
105							
106	PE=PE+r6i*(r6i-1.0);	0.9%	0.0%	0.0%	0.111s	0s	0s
107							
108	//Component Forces						
109	Fxij = Xr*Fij;	0.8%	0.0%	0.0%	0.100s	0s	0s
110	Fyij = Yr*Fij;	0.4%	0.0%	0.0%	0.051s	0s	0s
111	Fzij = Zr*Fij;	0.7%	0.0%	0.0%	0.080s	0s	0s
112							
113	//Sum forces on atom i						
114	fx[i] +=Fxij;	0.8%	0.0%	0.0%	0.100s	0s	0s
115	fy[i] +=Fyij;	1.6%	0.0%	0.0%	0.189s	0s	0s
116	fz[i] +=Fzij;	0.8%	0.0%	0.0%	0.090s	0s	0s
117							
118	//Sum forces on atom j						
119	fx[j] -= Fxij;	1.1%	0.0%	0.0%	0.130s	0s	0s
120	fy[j] -= Fyij;	0.7%	0.0%	0.0%	0.080s	0s	0s
121	fz[j] -= Fzij;	1.3%	0.0%	0.0%	0.149s	0s	0s
122							
123	p_PE -= Fxij*Xr-Fyij*Yr-Fzij*Zr;	0.5%	0.0%	0.0%	0.060s	0s	0s
124							
Selec...		16.2%	0.0%	0.0%	1.919s	0s	0s

Bottom-up – Real Code



Same Real Code – Top-down



Function Stack	CPU Time: Total – Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)	CPU Time: Total – Spin Time	CPU Time: Total – Overhead Time	CPU Time: Self – Effective Time by Utilization (Idle / Poor / Ok / Ideal / Over)
Total	100.0%	0.0%	0.0%	0s
_start	100.0%	0.0%	0.0%	0s
__libc_start_main	100.0%	0.0%	0.0%	0s
main	100.0%	0.0%	0.0%	0.010s
[redacted]	99.9%	0.0%	0.0%	0.010s
[redacted]	72.4%	0.0%	0.0%	0s
[redacted]	70.9%	0.0%	0.0%	0s
[redacted]	56.2%	0.0%	0.0%	0s
[redacted]	56.2%	0.0%	0.0%	0.350s
[redacted]	39.3%	0.0%	0.0%	11.069s
[redacted]	5.7%	0.0%	0.0%	2.200s
[redacted]	4.1%	0.0%	0.0%	1.910s
[redacted]	1.9%	0.0%	0.0%	0.190s
[redacted]	1.4%	0.0%	0.0%	0.390s
[redacted]	0.8%	0.0%	0.0%	0.120s
[redacted]	0.6%	0.0%	0.0%	0.100s
[redacted]	0.3%	0.0%	0.0%	0.150s
[redacted]	0.3%	0.0%	0.0%	0.120s
[redacted]	0.1%	0.0%	0.0%	0.050s
[redacted]	0.1%	0.0%	0.0%	0.030s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0.010s
[redacted]	8.4%	0.0%	0.0%	0.100s
[redacted]	3.9%	0.0%	0.0%	0.240s
[redacted]	3.6%	0.0%	0.0%	0.060s
[redacted]	0.1%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0.010s
[redacted]	8.4%	0.0%	0.0%	0s
[redacted]	5.6%	0.0%	0.0%	0.099s
[redacted]	0.7%	0.0%	0.0%	0.020s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0.010s
[redacted]	1.3%	0.0%	0.0%	0s
[redacted]	0.2%	0.0%	0.0%	0.010s
[redacted]	0.1%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0s
[redacted]	0.0%	0.0%	0.0%	0.010s
[redacted]	25.2%	0.0%	0.0%	0s
[redacted]	24.9%	0.0%	0.0%	0s
[redacted]	0.3%	0.0%	0.0%	0s
[redacted]	1.0%	0.0%	0.0%	0s
[redacted]	0.6%	0.0%	0.0%	0s
[redacted]	0.6%	0.0%	0.0%	0.060s
[redacted]	0.1%	0.0%	0.0%	0s
Highlighted 1 row(s):	39.3%	0.0%	0.0%	11.069s

OFFICE of INFORMATION TECHNOLOGY

PICSciE

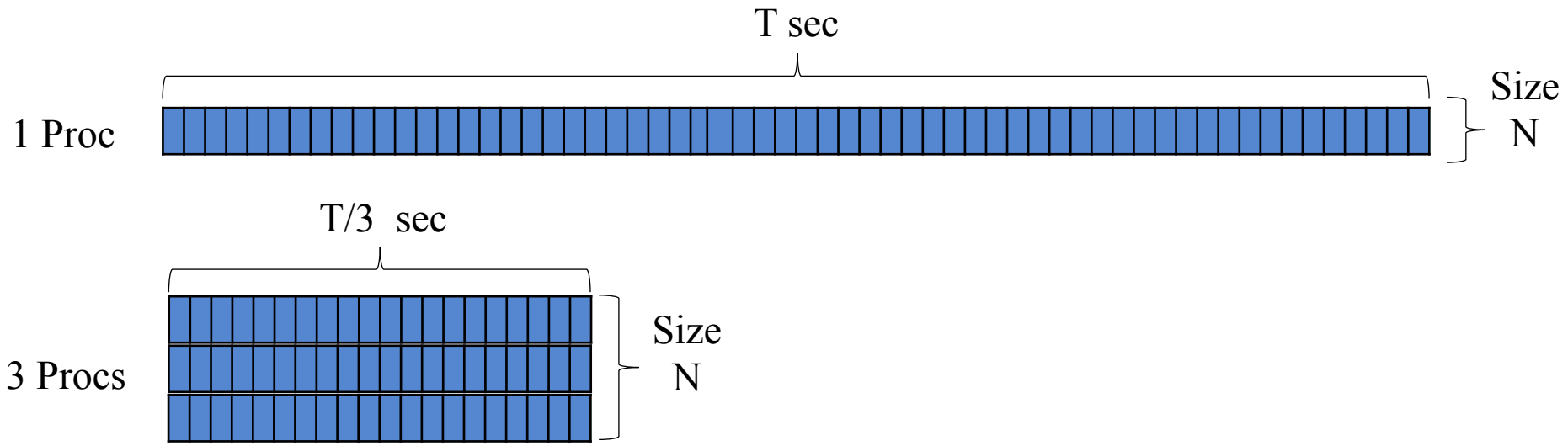
Examples

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

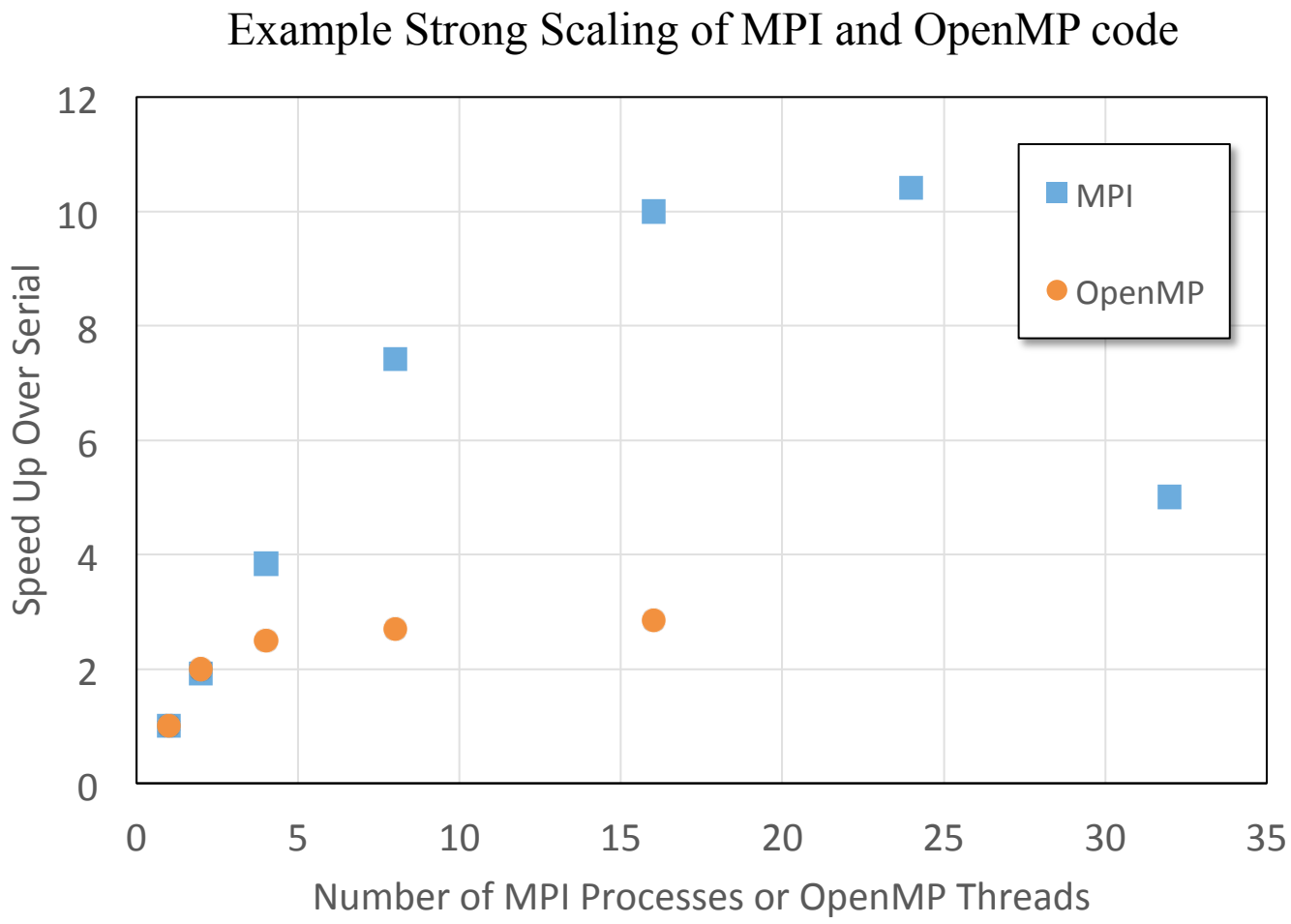
1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune

Strong Scaling

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

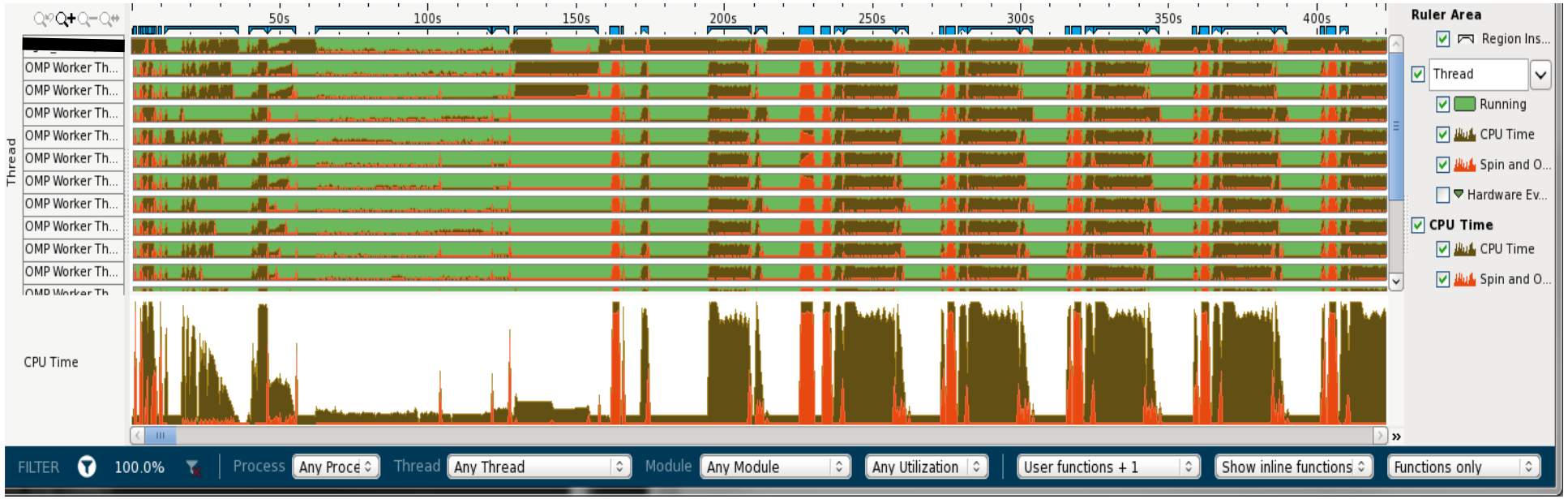


Scaling and Performance



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



Examples

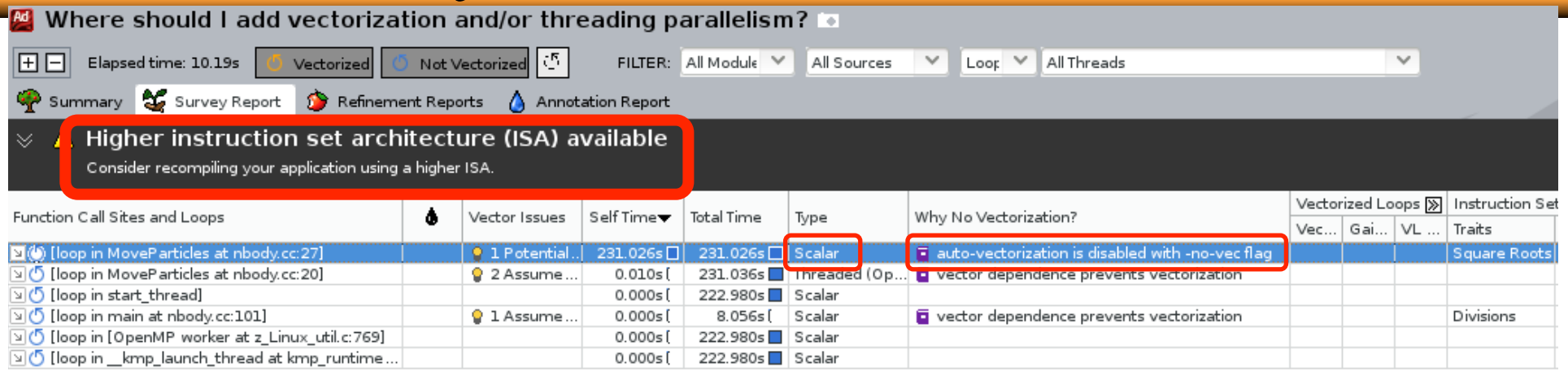
A handful of real(ish) codes to demonstrate profiling:

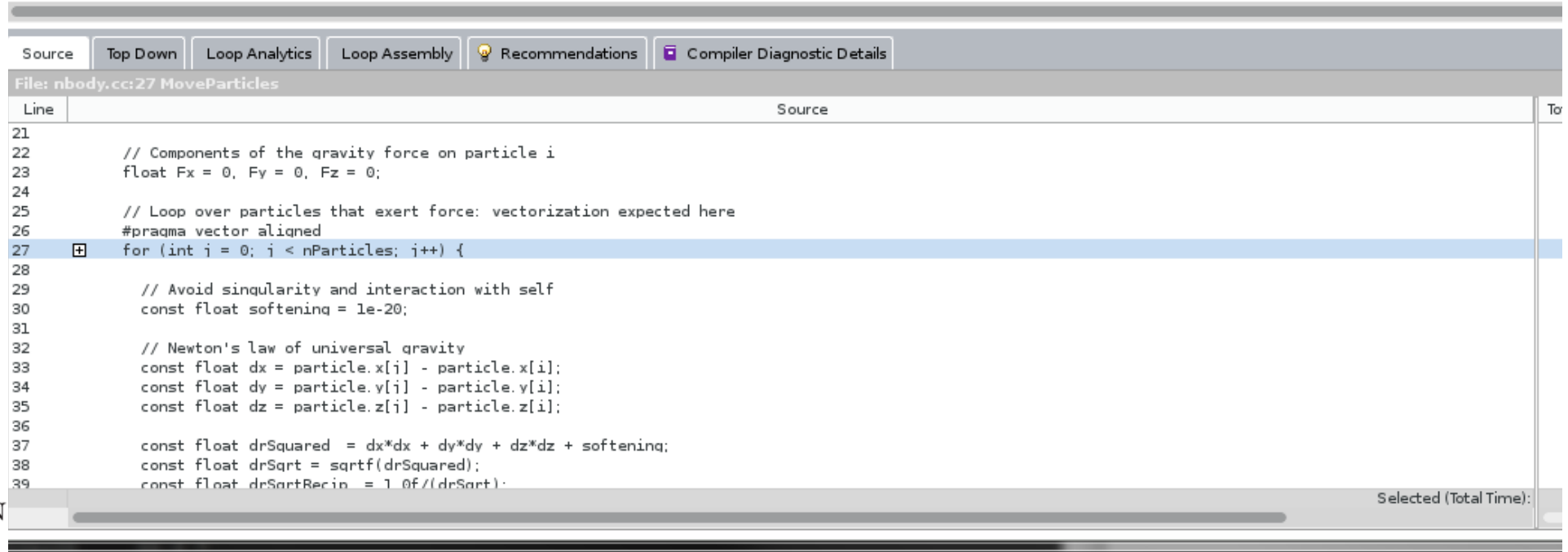
1. Hotspots
   - VTune
2. OpenMP Scaling
   - VTune
3. Vectorization
   - Advisor
4. Cache performance & Hardware Metrics
   - VTune

Intel Advisor Example

- 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

Poorly Vectorized - Advisor

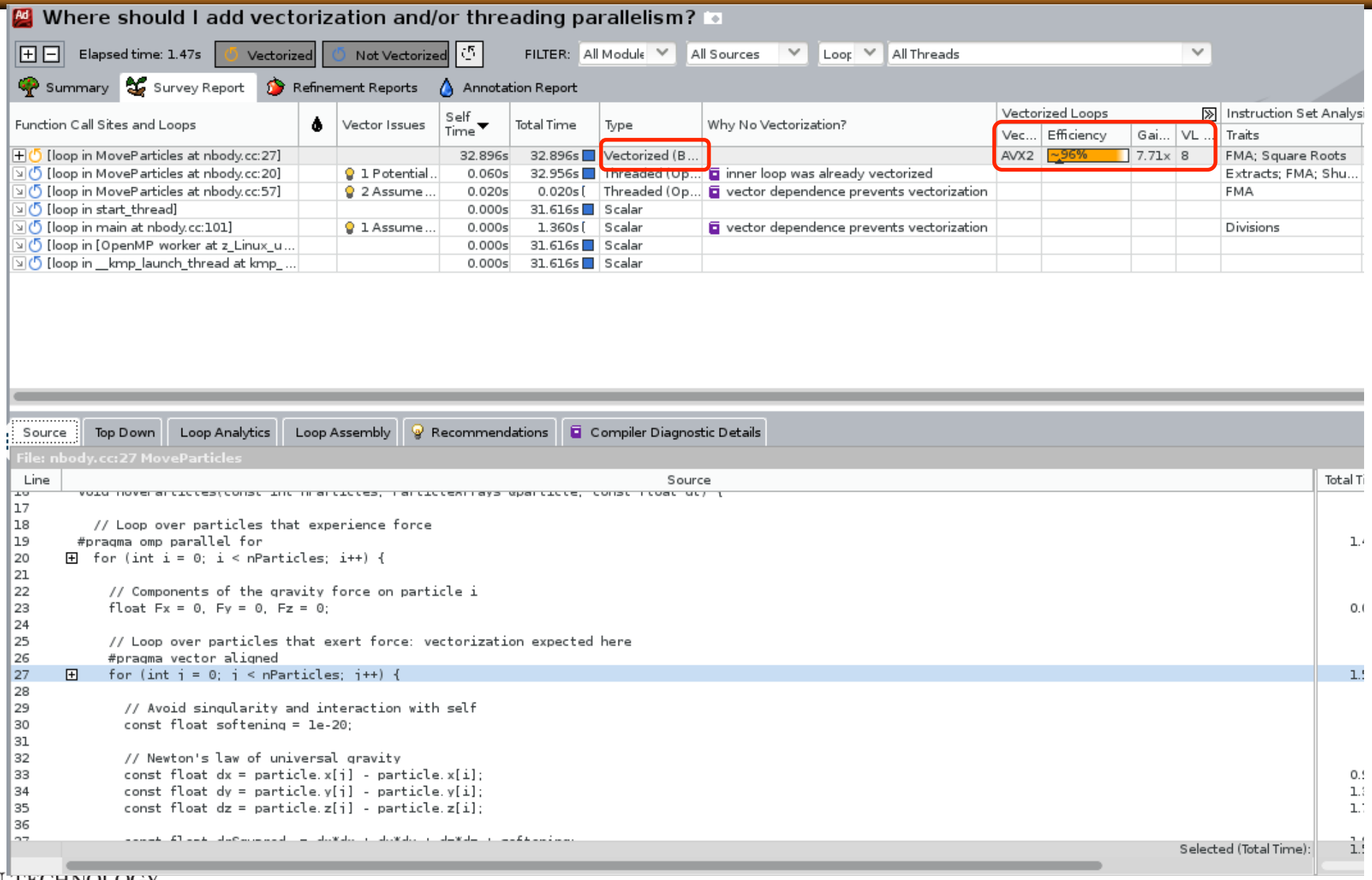




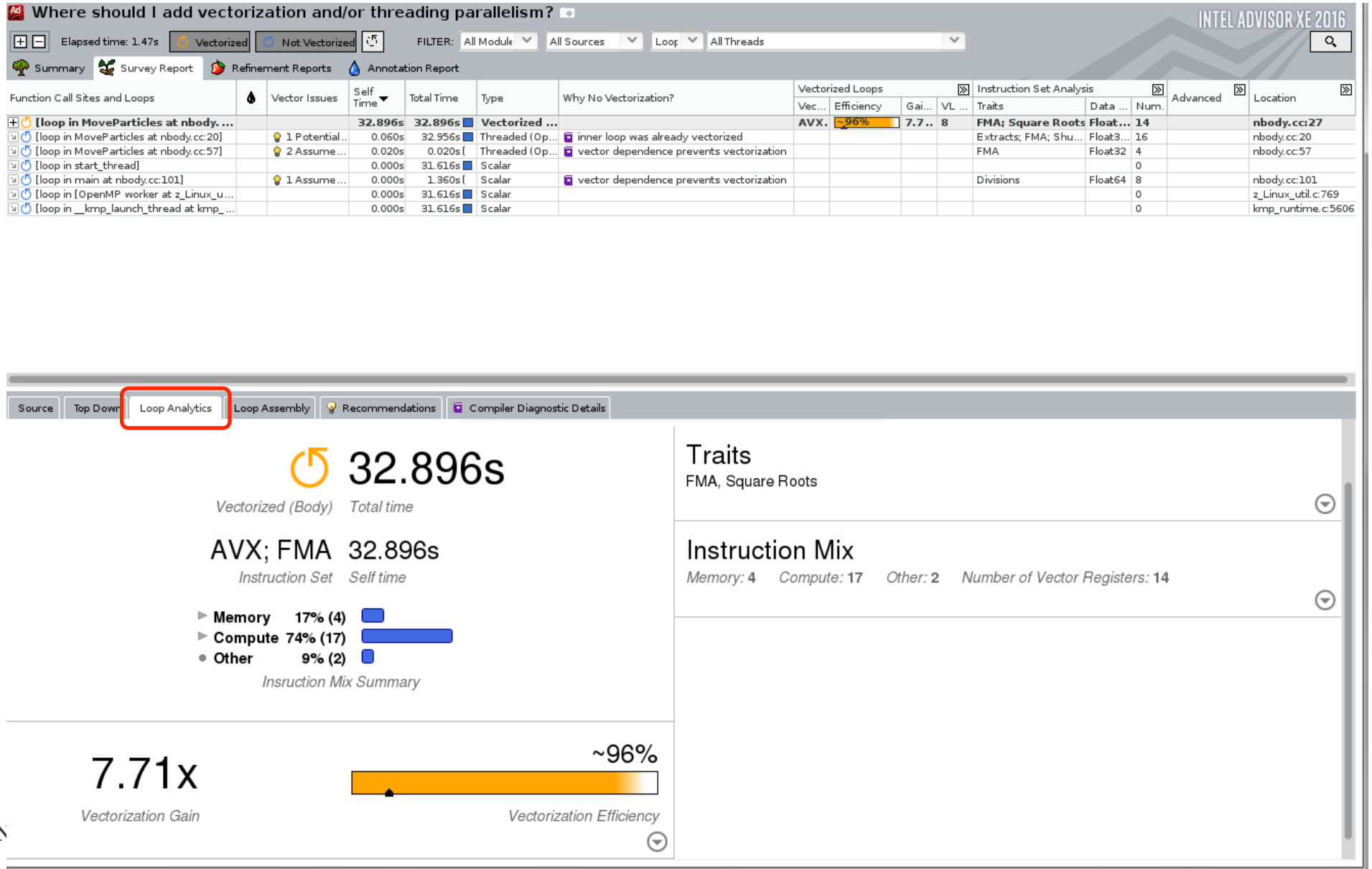
OFFICE of INFORMATION

PICSciE

Vectorized - Advisor



Vectorized - Advisor



OFFICE of INFORMATION

PICSciE

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=A×B
- 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

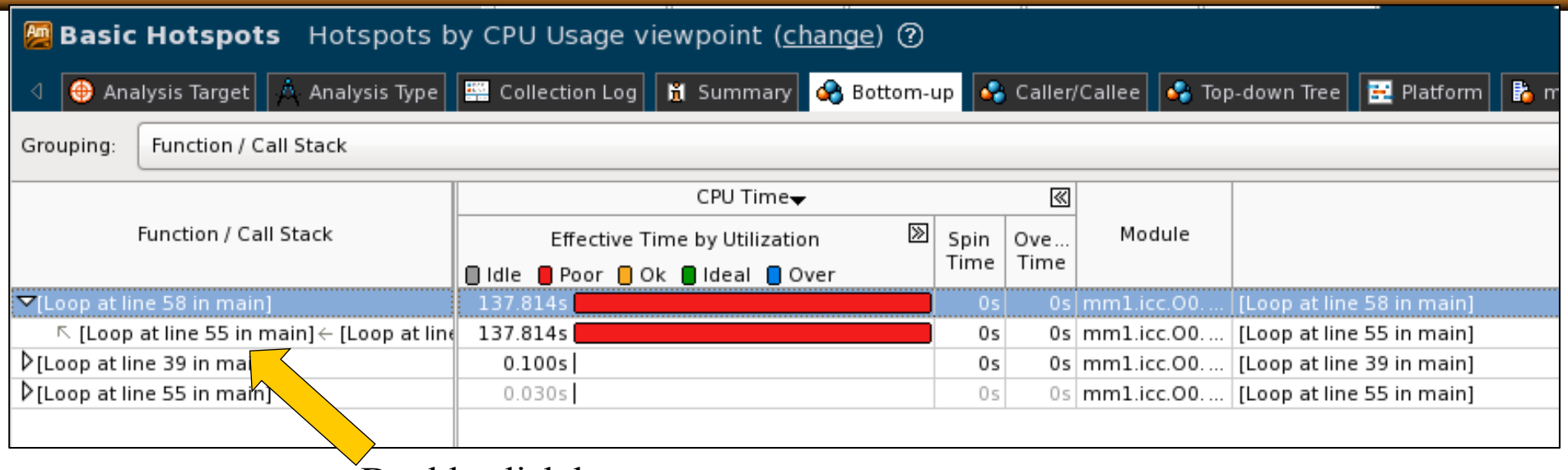
- Basic profilers “sample” the current program counter to see what instruction the cpu is currently executing.
- Then, after collecting enough samples, can give a good approximation as to what the program was doing.

- 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

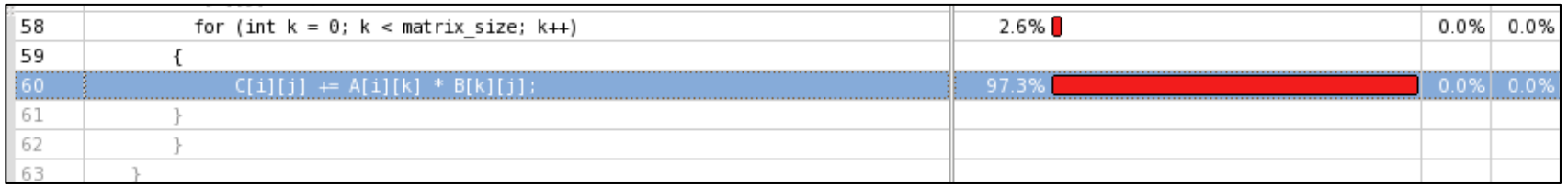
- 2D Matrix multiplication from before – mm1
  - 3 nested for loops
  - Let's assume we don't know why it is slow or even if it is slow

VTune Hotspots – Bottom-up View



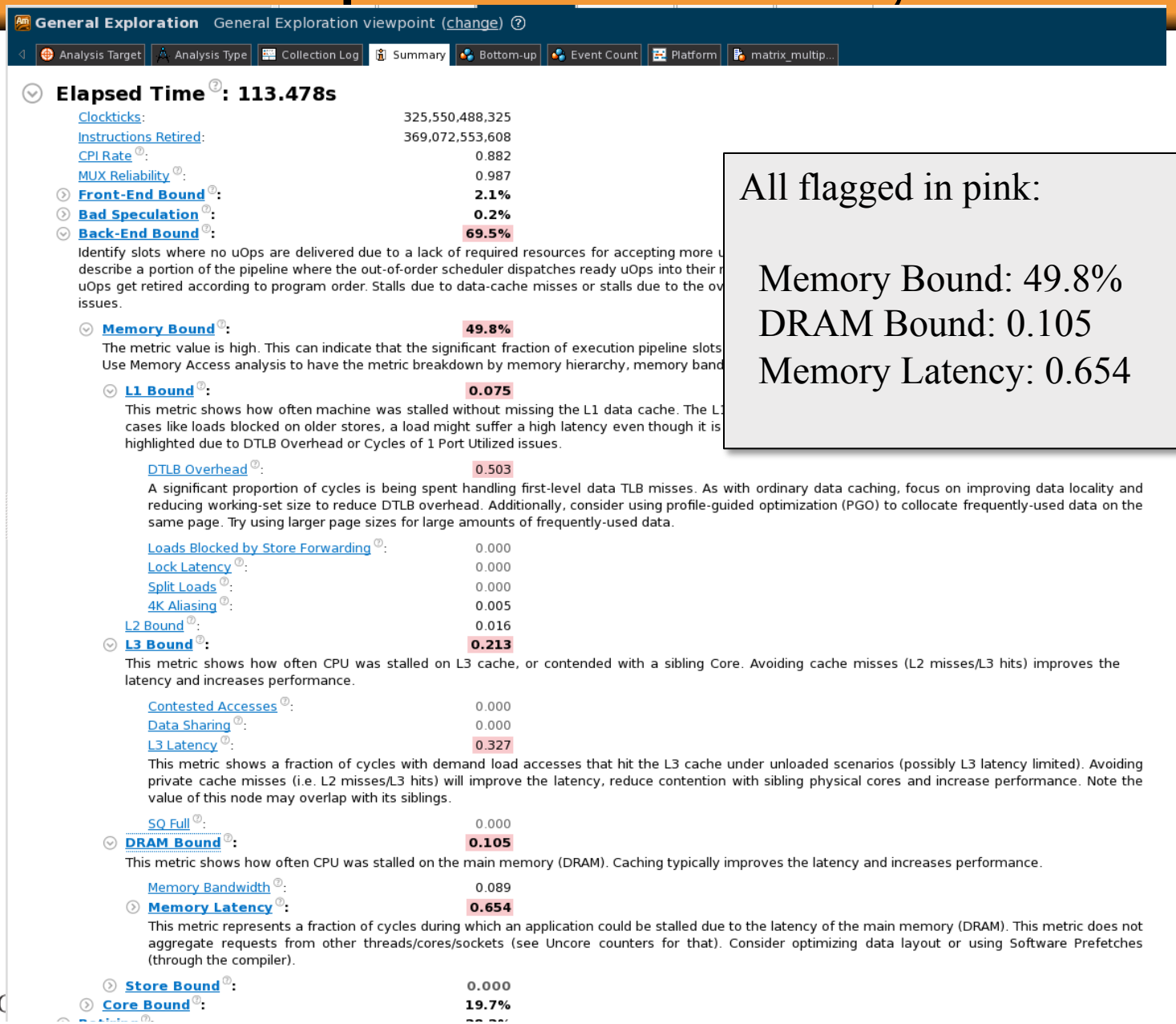
Double click here

opens source below:



Clearly there is one very expensive line inside nested for loops. *But why?*

General Exploration Analysis - VTune



All flagged in pink:

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

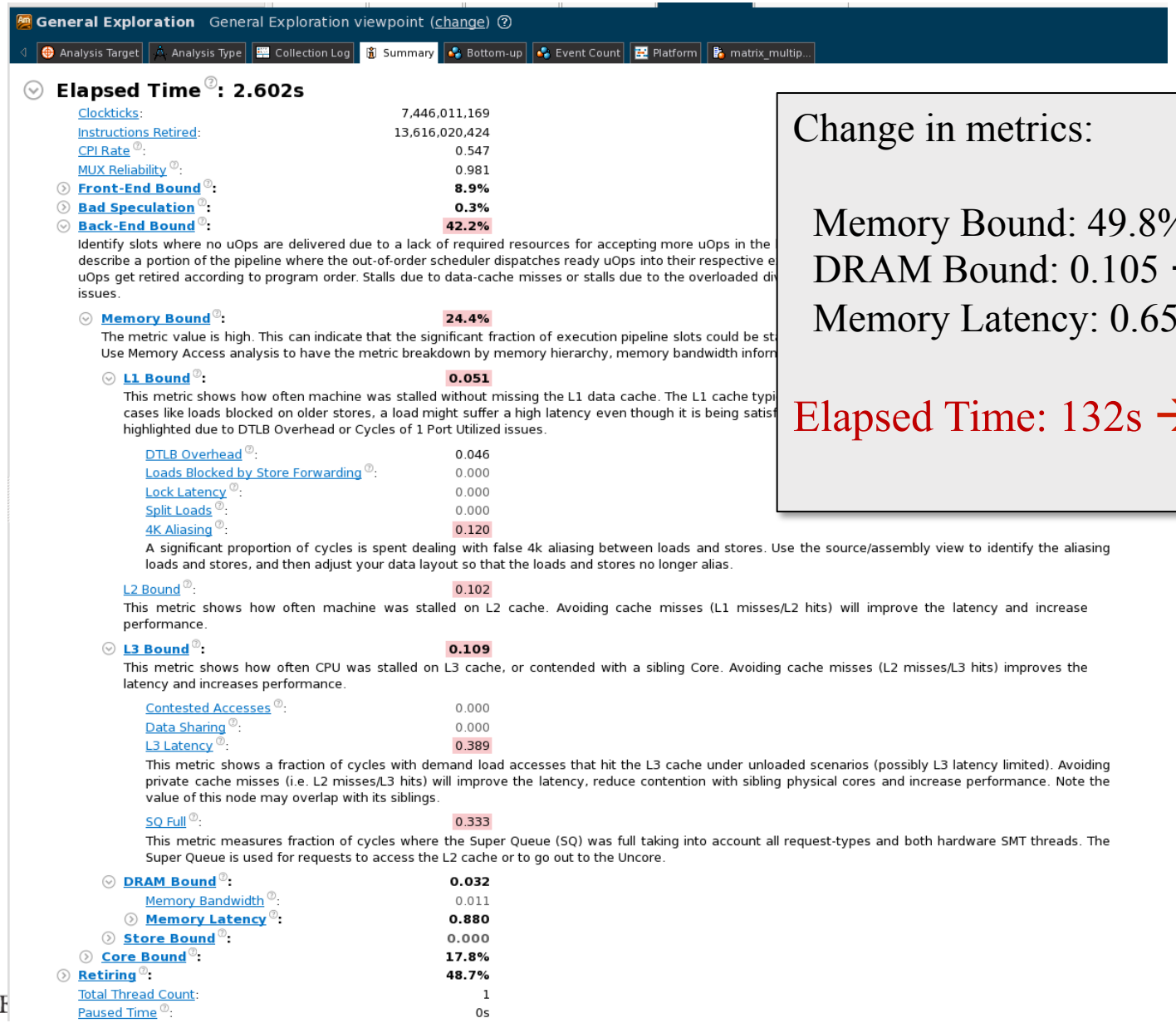
Hardware Counters

- Remember every generation of chips has different counters
- VTune developers have calculated *derived metrics* that they feel best represent real problems
  - For example the Haswell/Broadwell hardware counter: MEM_LOAD_UOPS_RETIRED.L3_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 Reodering



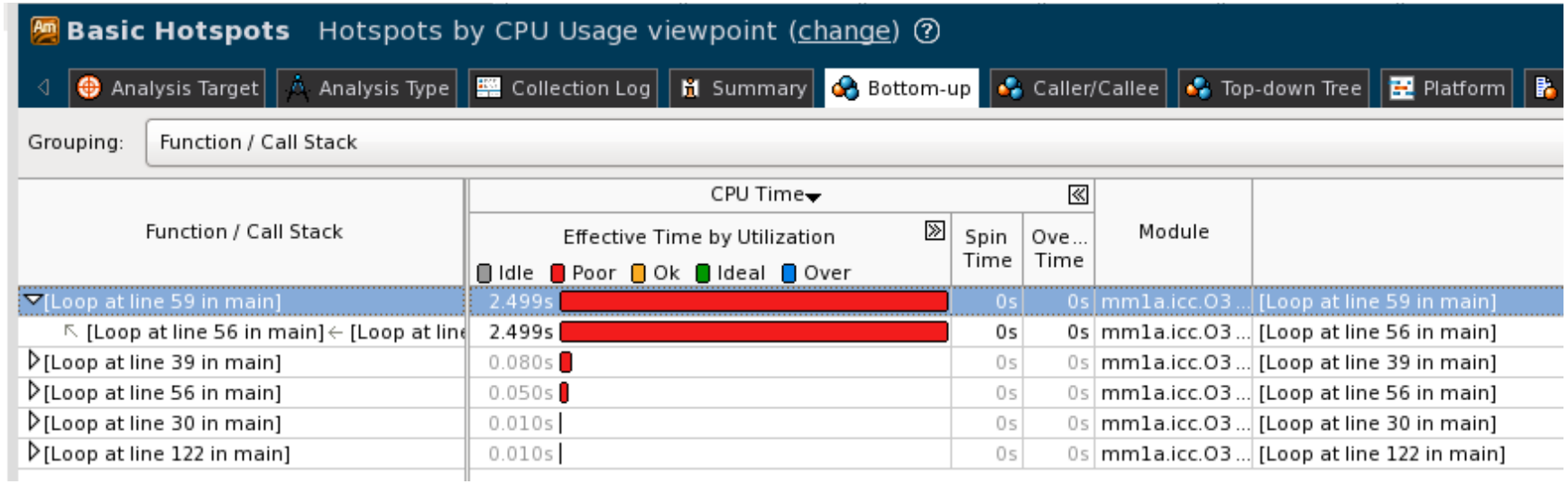
Change in metrics:

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

Elapsed Time: 132s → 2.6s

Results

- Reording 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
  - Free Intel tools for students: https://software.intel.com/en-us/qualify-for-free-software/student
- 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**