Introduction to Performance Optimization and Tuning Tools

Steve Lantz, Cornell University

CoDaS-HEP Summer School, July 19, 2023

Cornell University
Center for Advanced Computing

with thanks to Bei Wang, NVIDIA

Goals

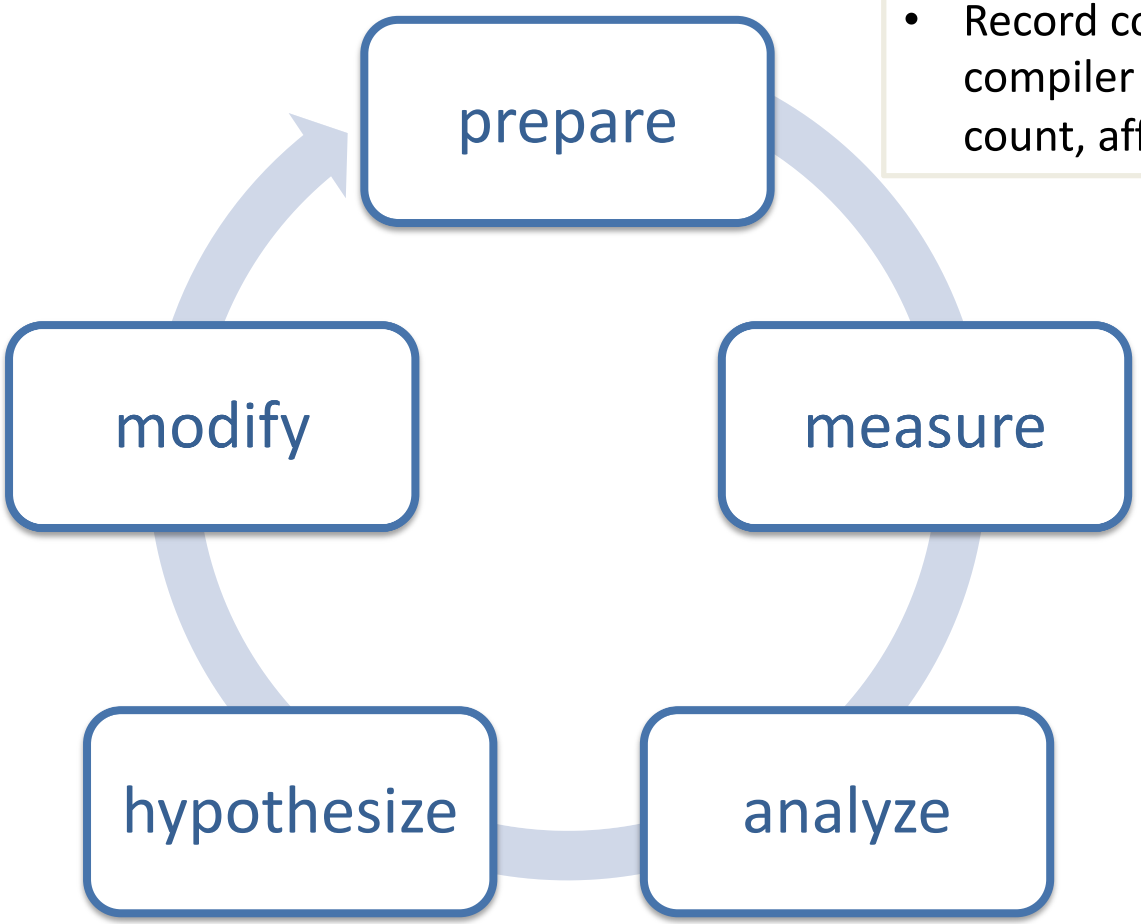
- Give an overview of what is meant by performance optimization and tuning
- Provide basic guidance on how to understand the performance of a code using tools
- Provide a starting point for performance optimizations

Performance Tuning: What Is It? Why Do It?

- What is performance tuning?
  - The process of improving the efficiency of an application to make better use of a given hardware resource
  - A cycle of identifying bottlenecks, eliminating these where possible, and rechecking efficiency – usually continued until performance objectives are satisfied
  - Writing code informed by one's understanding of the performance features of the given hardware (see previous presentations on "What Every Computational Physicist Should Know About Computer Architecture" and "Vector Parallelism on Multi-Core Processors")
- Why does performance matter?
  - Energy efficiency is becoming increasingly important
  - Today's applications only use a fraction of the machine
  - Due to complex architectures, mapping applications onto architectures is hard

The Performance Tuning Cycle

- Change only **one thing at a time**
- Consider the ease (difficulty) of implementation
- Keep **track** of all **changes**
- Apply regression test to **ensure correctness** after each change
- Remember: fast computing of a wrong result is completely irrelevant



- Choose a workload which is measurable, representative, static, reproducible, and quantifiable
- Record code generation, compiler version, compiler flags, input parameters, core count, affinity, etc.

What Do I Measure?

- Choose metrics which quantify the performance of your code
  - **Time** spent at different levels: whole program, functions, lines of code
  - **Hardware counters** can help you figure out the reasons for slow spots
- What are some easy ways to make time measurements?
  - Wrap your executable command in the Linux "time" command
    - Get an idea of overall run time: `time ./my_exe` (or `/bin/time ./my_exe`)
    - No way to zero in on performance bottlenecks
  - Insert calls to timers around critical loops/functions
    - `gettimeofday()`, `MPI_Wtime()`, `omp_get_wtime()`
    - Available in common libraries (system, MPI, OpenMP respectively)
    - Good for checking known hotspots in a small code base
    - Hard to maintain, require significant a priori knowledge of the code

Advantages of Performance Tools

- Performance tools (recommended)
  - Collect a lot data with varying granularity, cost and accuracy
  - Connect back to the source code (use -g compiler flag)
  - Analyze/visualize collected data using the tool
  - The learning curve is steep, but you can climb it gradually
- Tools generally work in one of two ways

Sampling
- Records system state at periodic intervals
- Useful to get an overview
- Low and uniform overhead
- Ex. Profiling

Instrumentation
- Records all events
- Provide detailed per event information
- High overhead for request events
- Ex. Tracing

Performance Tools Overview

- Basic OS tools
  - /bin/time
  - perf, gprof, igprof (from HEP)
  - valgrind, callgrind
- Hardware counters
  - PAPI API & tool set
- Community open source
  - HPCToolkit (Rice Univ.)
  - TAU (Univ. of Oregon)
  - Open|SpeedShop (Krell)
- Commercial products
  - Linaro Forge (DDT, MAP)
- Vendor supplied (free)
  - Intel Advisor, Intel VTune
  - Intel Trace Analyzer and Collector (MPI)
  - AMD μProf
  - CrayPat
  - NVIDIA Nsight Compute (CUDA)
  - NVIDIA pgprof (OpenACC)
  - AMD Omniprof (ROC)

No tool can do everything. Choose the right tool for the right task.

What Can I Learn From Performance Tools?

- Where am I spending my time?
  - Find the hotspots
- Is my code memory bound or compute bound?
  - Memory bound code has lots of events like these (tracked by hardware counters):
    - L1/L2/L3 cache misses
    - TLB misses
  - Compute bound code has lots of events like these:
    - Pipeline stalls not due to memory events
    - Type conversions
    - Time spent in unvectorized loops
- Is my I/O inefficient?

Typical Performance Pitfalls on a Single Node

- Scattered memory accesses that constantly bring in new cache lines
  - Storing data as an array of structs (AoS) instead of a struct of arrays (SoA)
  - Looping through arrays with a large stride

More cache lines ⇒ data must be fetched from more distant caches, or from RAM

	Registers	**L1**	**L2**	**LLC**	**DRAM**
Speed (cycle)	1	~4	~10	~30	~200
Size	< KB	~32KB	~256KB	~35MB	10-100GB

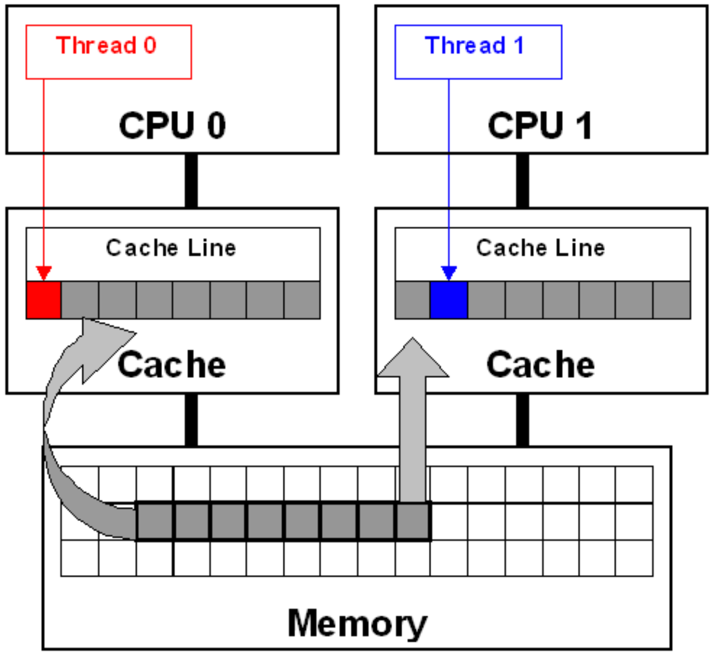
- Mismatched types in assignments

```
float x=3.14; //bad: 3.14 is a double
float s=sin(x); //bad: sin() is a double precision function
long v=round(x); //bad: round() takes and returns double
```

```
float x=3.14f; //good: 3.14f is a float
float s=sinf(x); //good: sin() is a single precision function
long v=lroundf(x); //good: lroundf() takes float and returns long
```

Typical Performance Pitfalls: Multithreading

- Load imbalance
- False sharing: when CPUs alter different variables in the same cache line ↓
  - Data aren't really shared, but caches must stay coherent
  - Data always travel together in "cache lines" of 64 bytes
- Insufficient parallelism
- Synchronization
  - Use private thread storage to avoid synchronization
- Non-optimal memory placement
  - Memory is actually allocated on first touch
  - Thread that touches first has fastest access



https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

Linux Tool: *perf*

- Perf is a performance analyzing tool in Linux
  - *perf record*: measure and save sampling data for a single program
    - *-g*: enable call-graph (callers/callee information)
  - *perf report*: analyze the file generated by perf record, can be flat profile or graph
    - *-g*: enable call-graph (callers/callee information)
  - *perf stat*: measure total event count for a single program
    - *-e event-name-1,event-name-2*: choose from event names provided by *perf list*
  - *perf list*: list available hardware and software events for measurement
- When compiling the code, use the following flags for easier interpretation
  - *-g*: generate debug symbols needed to annotate source
  - *-fno-omit-frame-pointer*: provide stack chain/backtrace

https://perf.wiki.kernel.org/index.php/Tutorial
https://www.brendangregg.com/perf.html

Example: Finding Hot Spots with *perf*

- Compile the code: *g++ -g -fno-omit-frame-pointer -O3 -DNAIVE matmul_2D.cpp -o mm_naive.out*
- Collect profiling data: *perf record -g ./mm_naive.out 500*
- Open the result: *perf report -g*

```
Samples: 7K of event 'cycles:uppp', Event count (approx.): 5629336320
  Children      Self  Command       Shared Object      Symbol
+   99.95%     0.00%  mm_naive.out  libc-2.17.so       [.] __libc_start_main
+   99.95%     0.00%  mm_naive.out  mm_naive.out       [.] main
-   99.69%    99.69%  mm_naive.out  mm_naive.out       [.] compute_naive
     __libc_start_main
     main
     compute_naive
     0.09%     0.09%  mm_naive.out  mm_naive.out       [.] init_matrix_2D
     0.06%     0.06%  mm_naive.out  libc-2.17.so       [.] __random
     0.06%     0.06%  mm_naive.out  libc-2.17.so       [.] __memset_sse2
     0.03%     0.03%  mm_naive.out  [unknown]          [.] 0xffffffff8196c4e7
     0.03%     0.00%  mm_naive.out  [unknown]          [.] 0000000000000000
     0.02%     0.02%  mm_naive.out  libc-2.17.so       [.] __random_r
     0.01%     0.01%  mm_naive.out  mm_naive.out       [.] rand@plt
     0.01%     0.01%  mm_naive.out  ld-2.17.so         [.] do_lookup_x
     0.01%     0.01%  mm_naive.out  libc-2.17.so       [.] _int_malloc
     0.01%     0.01%  mm_naive.out  libc-2.17.so       [.] intel_check_word
     0.00%     0.00%  mm_naive.out  ld-2.17.so         [.] check_match.9523
     0.00%     0.00%  mm_naive.out  [unknown]          [.] 0x00000000000c2698
     0.00%     0.00%  mm_naive.out  ld-2.17.so         [.] _dl_sysdep_start
     0.00%     0.00%  mm_naive.out  ld-2.17.so         [.] dl_main
     0.00%     0.00%  mm_naive.out  ld-2.17.so         [.] _dl_load_cache_lookup
     0.00%     0.00%  mm_naive.out  ld-2.17.so         [.] _etext
     0.00%     0.00%  mm_naive.out  ld-2.17.so         [.] _dl_map_object
     0.00%     0.00%  mm_naive.out  ld-2.17.so         [.] __libc_memalign@plt
     0.00%     0.00%  mm_naive.out  ld-2.17.so         [.] _dl_start_user
```

Press “A” →

```
init_matrix_2D  /home/beiwang/codas_perftools/mm_naive.out
Percent│     __attribute__((noinline)) void init_matrix_2D(double **A, double **B, double **C, int matrix_size){
       │     #pragma omp parallel for
       │       for (int i=0; i<matrix_size; i++) {
       │       test   %ecx,%ecx
       │     ↓ jle    401558 <init_matrix_2D(double**, double**, b8
       │     __attribute__((noinline)) void init_matrix_2D(double **A, double **B, double **C, int matrix_size){
       │       push   %rbp
       │       lea    -0x1(%rcx),%eax
       │       mov    %rsp,%rbp
       │       push   %r15
       │       push   %r14
       │       mov    %rsi,%r14
       │       push   %r13
       │       lea    0x8(%rdi,%rax,8),%rsi
       │       push   %r12
       │       push   %rbx
       │       lea    0x8(,%rax,8),%r13
       │       mov    %rdi,%r12
       │       mov    %rdx,%r15
       │       sub    $0x18,%rsp
       │       mov    %rsi,-0x38(%rbp)
       │       nop
       │40:    xor    %ebx,%ebx
       │       nop
       │         for (int j = 0 ; j < matrix_size; j++) {
       │           A[i][j]=((double) rand() / (RAND_MAX));
       │48: → callq  rand@plt
       │       pxor   %xmm0,%xmm0
 55.56 │       mov    (%r12),%rdx
       │       cvtsi2sd %eax,%xmm0
       │       divsd  0x5e7(%rip),%xmm0        # 401ae8 <__dso_handle+0x60>
 22.22 │       movsd  %xmm0,(%rdx,%rbx,1)
       │           B[i][j]=((double) rand() / (RAND_MAX));
       │     → callq  rand@plt
       │       pxor   %xmm0,%xmm0
       │       mov    (%r14),%rdx
       │       cvtsi2sd %eax,%xmm0
       │           C[i][j]=0.0;
       │       mov    (%r15),%rax
       │           B[i][j]=((double) rand() / (RAND_MAX));
       │       divsd  0x5c7(%rip),%xmm0        # 401ae8 <__dso_handle+0x60>
 11.11 │       movsd  %xmm0,(%rdx,%rbx,1)
       │           C[i][j]=0.0;
 11.11 │       movq   $0x0,(%rax,%rbx,1)
       │       add    $0x8,%rbx
       │         for (int j = 0 ; j < matrix_size; j++) {
       │       cmp    %rbx,%r13
       │     ↑ jne    4014e8 <init_matrix_2D(double**, double**, 48
       │       add    $0x8,%r12
       │       add    $0x8,%r14
       │       add    $0x8,%r15
       │       for (int i=0; i<matrix_size; i++) {
       │       cmp    -0x38(%rbp),%r12
       │     ↑ jne    4014e0 <init_matrix_2D(double**, double**, 40
       │         }
       │       }
```

Example: Counting Cache Misses with *perf stat*

```
List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  LLC-load-misses                                    [Hardware cache event]
  LLC-loads                                          [Hardware cache event]
  LLC-store-misses                                   [Hardware cache event]
  LLC-stores                                         [Hardware cache event]
  branch-load-misses                                 [Hardware cache event]
  branch-loads                                       [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  dTLB-loads                                         [Hardware cache event]
  dTLB-store-misses                                  [Hardware cache event]
  dTLB-stores                                        [Hardware cache event]
  iTLB-load-misses                                   [Hardware cache event]
  iTLB-loads                                         [Hardware cache event]
  node-load-misses                                   [Hardware cache event]
  node-loads                                         [Hardware cache event]
  node-store-misses                                  [Hardware cache event]
  node-stores                                        [Hardware cache event]
```

- The *perf list* command lists all available CPU counters
  - Check *man perf_event_open* to see what each event measures
- The *perf stat* command instruments and summarizes selected CPU counters

  perf stat -e cpu-cycles,instructions,L1-dcache-loads,L1-dcache-load-misses ./mm_naive.out 500

```
Performance counter stats for './mm_naive.out 500':

     5,564,503,540      cpu-cycles
    10,063,662,841      instructions              #    1.81  insn per cycle
     3,767,490,743      L1-dcache-loads
     1,475,374,174      L1-dcache-load-misses     #   39.16% of all L1-dcache hits

       1.691104619 seconds time elapsed
```

  - Make changes, see if L1 load misses improve, e.g.

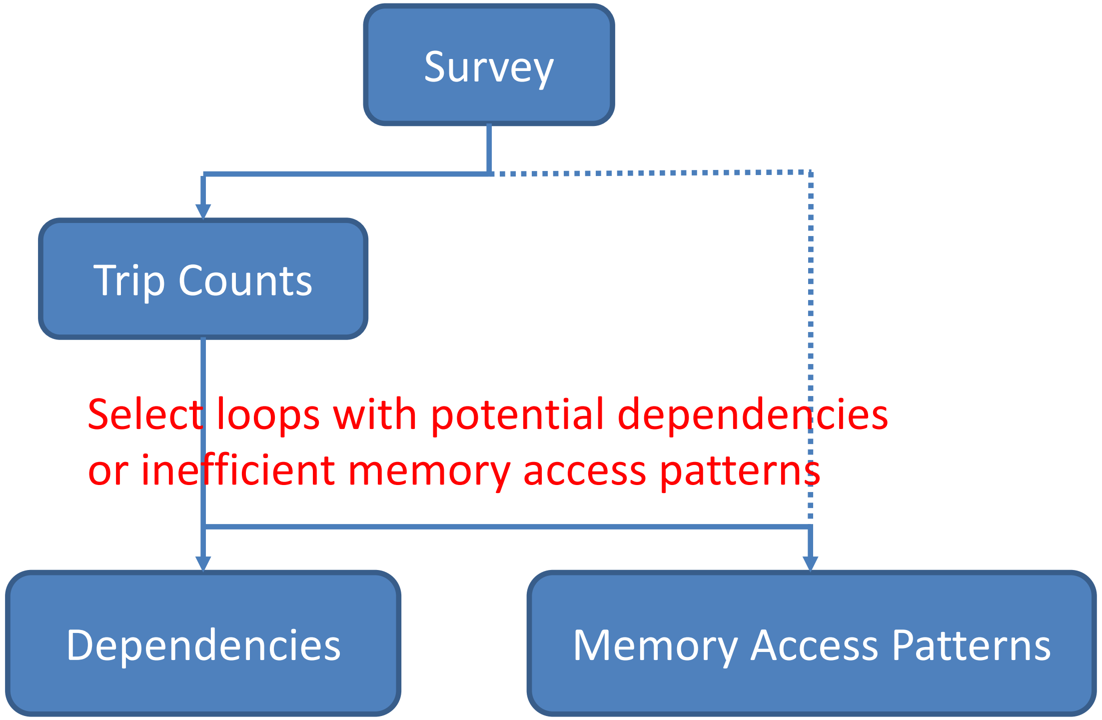
Intel Advisor

Two very useful analyses in Intel Advisor will be highlighted:

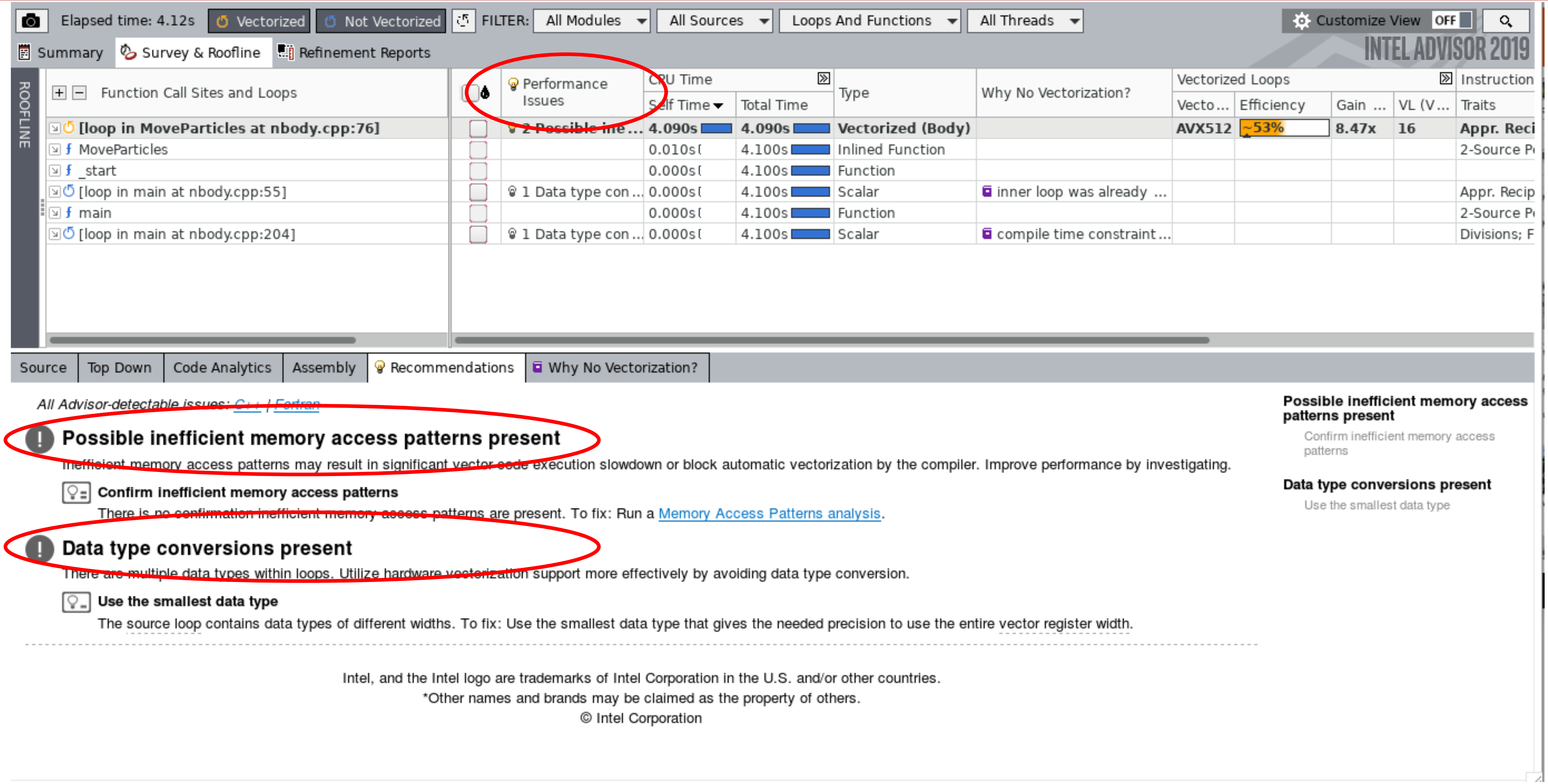
- Vectorization advisor
  - Provide vectorization information from vectorization report
  - Identify the hotspots where your efforts pay off the most
  - Provide call graph information
  - Identify the performance and vectorization issues
  - Check memory access pattern, dependencies, more
- Roofline
  - How much performance is being left on the table
  - Where are the bottlenecks
  - Which can be improved
  - Which are worth improving

Workflow of Vectorization Advisor

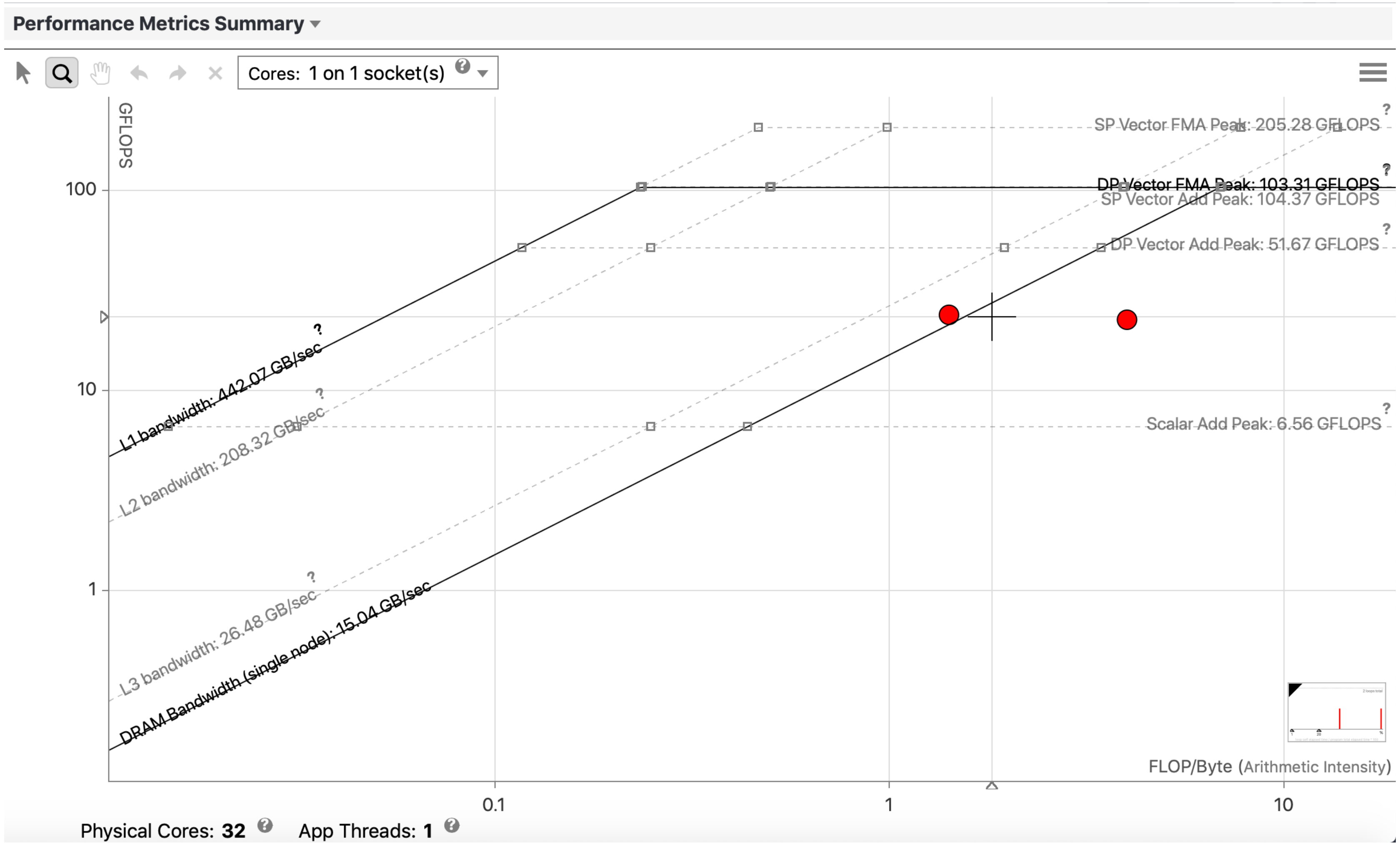
- **Survey**: find the vectorization information for loops and provide suggestions for improvement
- **Trip Counts**: generate a Roofline Chart
- **Memory Access Patterns** (MAP): see how you access the data
- **Dependencies**: determine if it is safe to force vectorization



Advisor Advises You About Performance Issues



Roofline Analysis: What Is It?

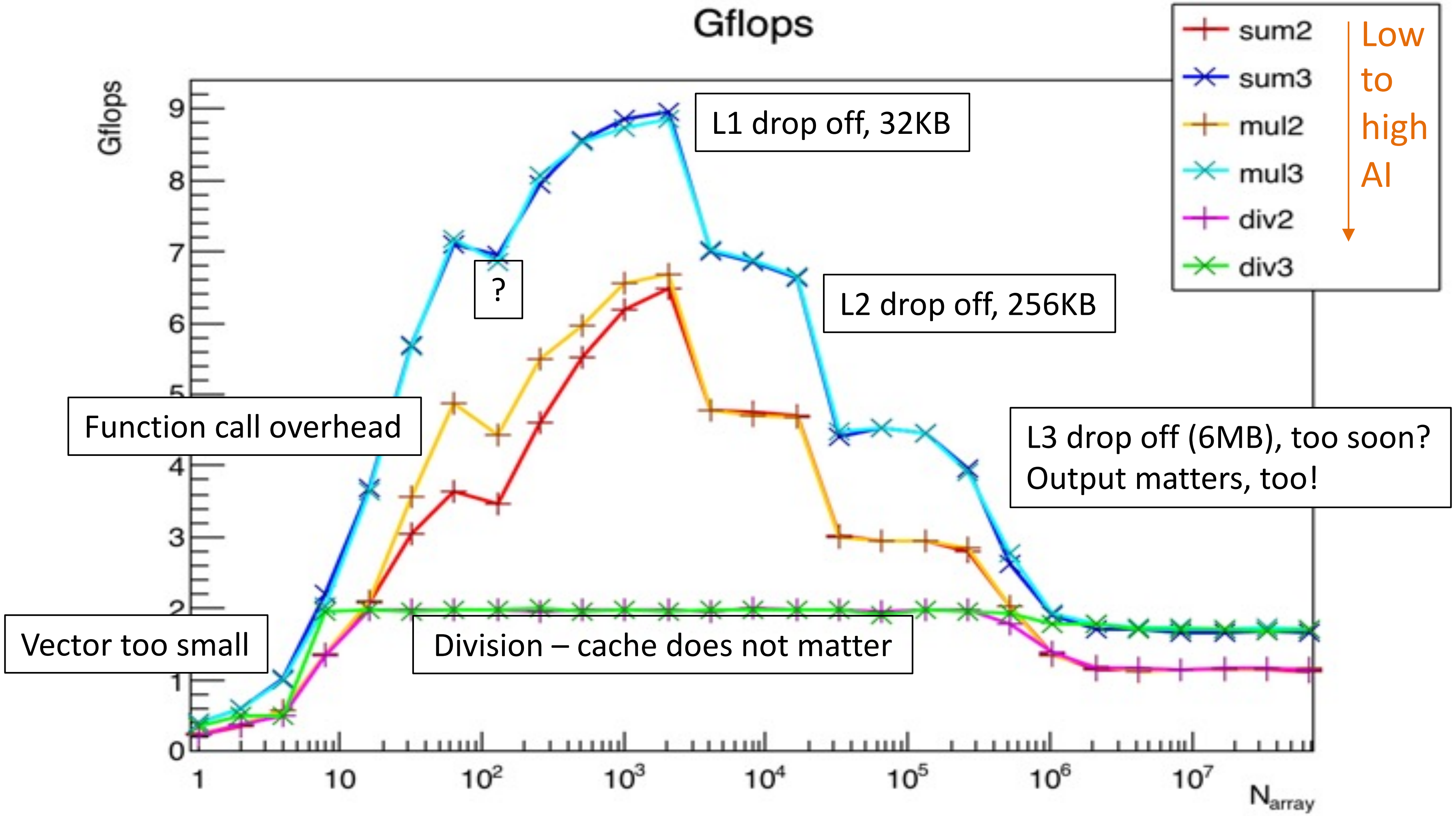


Towards Peak Flop/s: Arithmetic Intensity

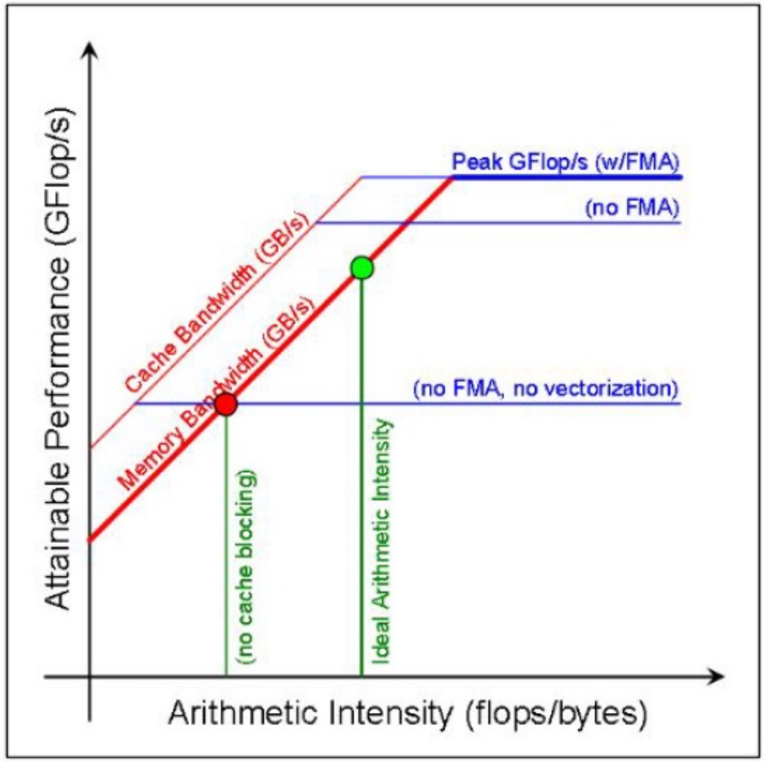
- Arithmetic intensity or AI is the number of flops executed by a code divided by the bytes of memory that are required to perform the computations
  - AI is an intrinsic property of the code
- Even a simple stride-1 loop may not get the peak flop/s rate, if its AI is low
  - VPU becomes stalled waiting for loads and stores to complete
  - Delays become longer as the memory request goes further out in the hierarchy from L1 to L2 (to L3?) to RAM
  - Even if the right vectors are in L1 cache, there is limited bandwidth from L1 to registers!
- If the goal is to maximize flop/s, you'll want to try to improve AI
- Also want threads to work on independent, cache-size chunks of data
  - Watch out for false sharing, where 2 threads fight needlessly over a cache line

Effect of AI and Caches on GFLOP/s

Data taken on a laptop (2.6 GHz, vector width 8):

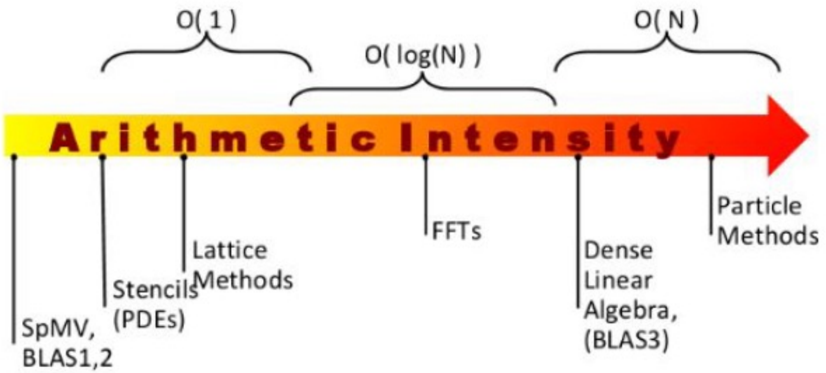


Roofline Analysis Explained



$$Attainable\ FLOPs/\sec = \min\begin{cases} Peak\ FLOPs/\sec, \\ Peak\ Memory\ Bandwidth \times Arithmetic\ Intensity \end{cases}$$

$$Arithmetic\ Intensity = \frac{Total\ FLOPs}{Total\ Bytes}$$



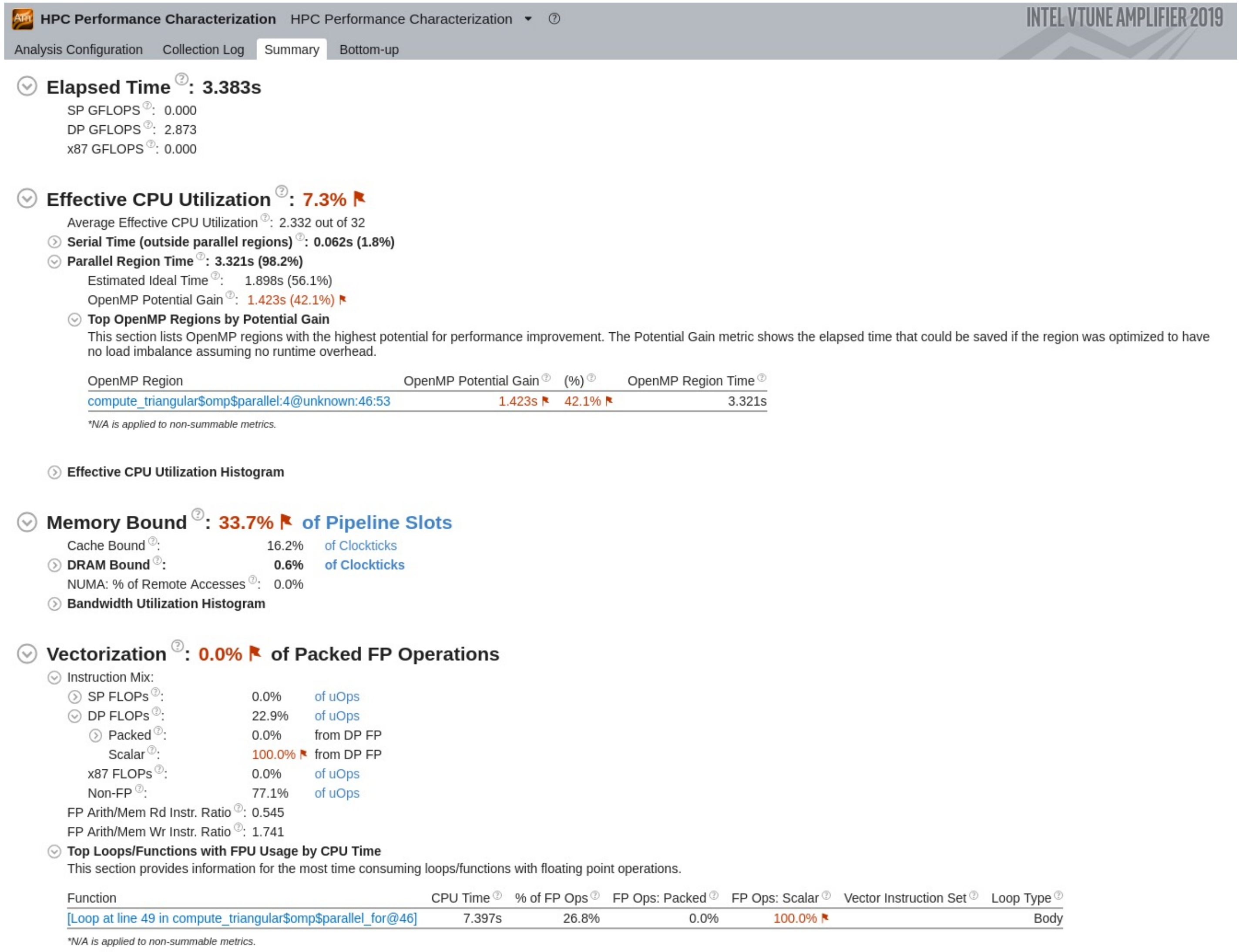
Deslippe et al., "Guiding Optimization Using the Roofline Model," tutorial presentation at IXPUG2016, Argonne, IL, Sept. 21, 2016. https://anl.app.box.com/v/IXPUG2016-presentation-29

What Does Roofline Analysis Tell You?

- Roofline analysis is a way of telling whether a piece of code is *compute bound* or *memory bound*
  - The “roofline” is a performance ceiling related to *hardware characteristics*
- The *arithmetic intensity* or AI (flop/byte) of a code tells you what part of the roof the code is under
  - AI is a *software characteristic* telling you the extent to which the code is limited by its need to load and store data from/to memory
- The roofline sets the highest flop/s rate possible for a given piece of code
  - If some of your functions fall way below that rate, you may need to investigate why
  - It’s possible to show that the AI needed for reaching *theoretical peak* flop/s (the highest flat roof) implies that 50% of operands are vector constants, i.e., they are loaded just once and never leave registers!

Intel VTune

- Covers all aspects of execution
  - Hotspots
  - Processor microarchitecture
  - Memory accesses
  - Threading
  - I/O
- Flexible
  - GUI in Linux, Windows and macOS
  - Drills down to source code, assembly
  - Easy setup, no special compiling
- Shared memory only
  - Serial or OpenMP
  - MPI, but only within a single node



Hotspots Analysis

Hotspots Hotspots by CPU Utilization ▾ ⓘ INTEL VTUNE AMPLIFIER 2019

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

Elapsed Time: 3.288s

CPU Time:	12.013s
Effective Time:	7.658s
Spin Time:	4.355s ⚑
Overhead Time:	0s
Instructions Retired:	35,906,000,000
Microarchitecture Usage:	24.2% ⚑ of Pipeline Slots
CPI Rate:	1.101 ⚑
Total Thread Count:	4
Paused Time:	0s

Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

Explore Additional Insights
Parallelism: 7.3% (2.329 out of 32 logical CPUs) ⚑
Use Threading to explore more opportunities to increase parallelism in your application.

Microarchitecture Usage: 24.2% ⚑
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

Vector Register Utilization: 12.5% ⚑
Use Intel Advisor to learn more on vectorization efficiency of your application.

INSIGHTS

Top Hotspots

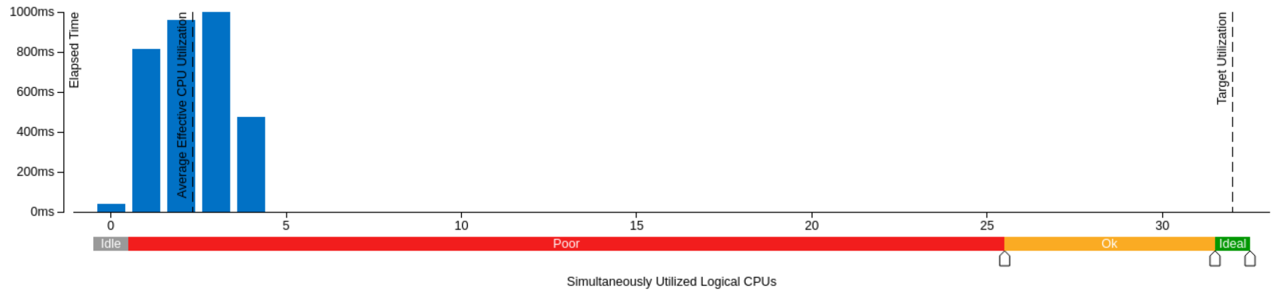
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
compute_triangularompparallel_for@46	mm_triangular_omp_icpc.out	7.247s
_INTERNAL_25_______src_kmp_barrier_cpp_38a91946::__kmp_wait_template<kmp_flag_64, (int)1, (bool)0, (bool)1>	libiomp5.so	4.305s ⚑
func@0xffffffff81775db9	vmlinux	0.085s
func@0xffffffff81775c70	vmlinux	0.060s
kmp_flag_native<unsigned long long>::get	libiomp5.so	0.045s
[Others]		0.271s

N/A is applied to non-summable metrics.

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.

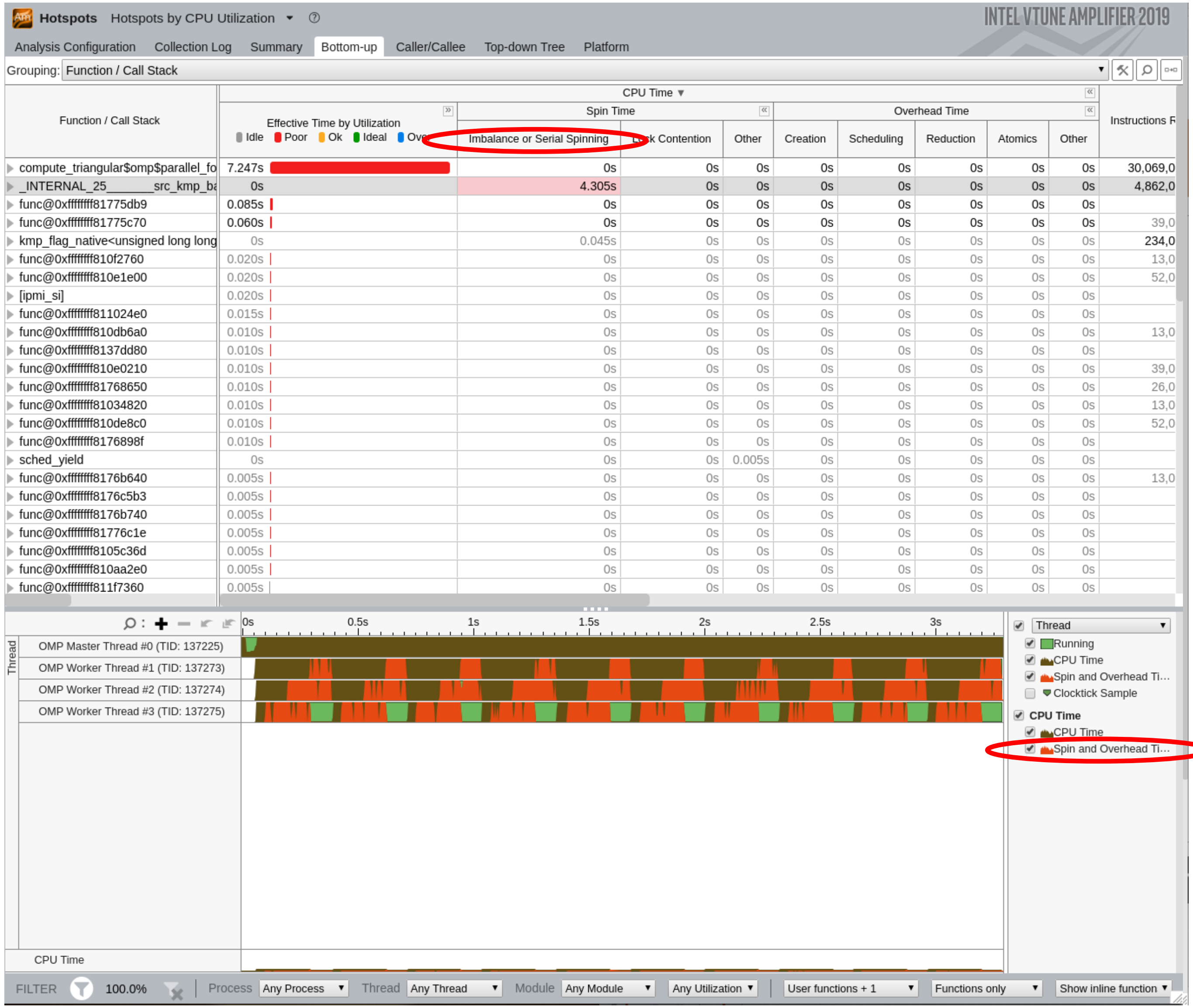


Collection and Platform Info

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

Application Command Line: /home/beiwang/codas_perftools/mm_triangular_omp_icpc.out 1000
Environment Variables: OMP_NUM_THREADS=4
User Name: beiwang

Thread Timelines Showing "Spin and Overhead"



CPU Utilization by Threads

Threading Threading Efficiency

INTEL VTUNE AMPLIFIER 2019

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

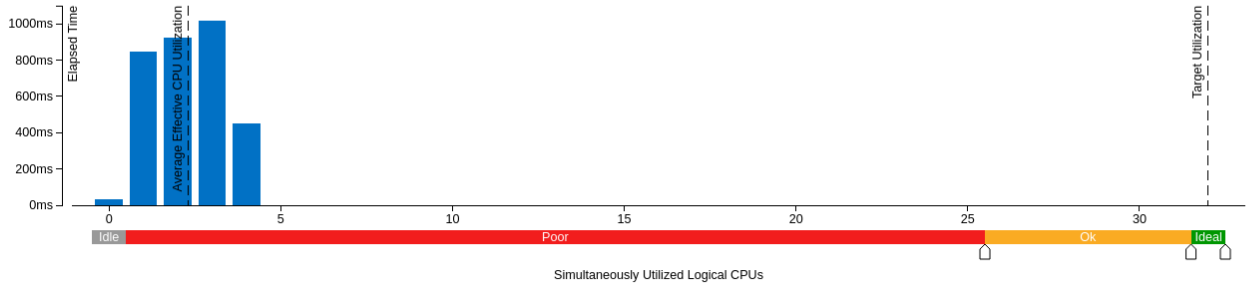
Elapsed Time: 3.270s

Paused Time: 0s

Effective CPU Utilization: 11.4% (3.640 out of 32 logical CPUs)

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.



OpenMP Analysis. Collection Time: 3.270

Serial Time (outside parallel regions): 0.072s (2.2%)

Parallel Region Time: 3.199s (97.8%)

Estimated Ideal Time: 1.854s (56.7%)

OpenMP Potential Gain: 1.345s (41.1%)

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain	(%)	OpenMP Region Time
compute_triangularompparallel:4@unknown:46:53	1.345s	41.1%	3.199s

N/A is applied to non-summable metrics.

Total Thread Count: 4

Inactive Wait Time with poor CPU Utilization: 0.915s (100.0% from Inactive Wait Time)

Inactive Sync Wait Time: 0.901s

Preemption Wait Time: 0.014s

Top functions by Inactive Wait Time with Poor CPU Utilization.

This section lists the functions sorted by the time spent waiting on synchronization or thread preemption with poor CPU Utilization.

Function	Module	Inactive Wait Time	Inactive Sync Wait Time	Inactive Sync Wait Count	Preemption Wait Time	Preemption Wait Count
pthread_cond_wait	libpthread-2.17.so	0.712s	0.712s	9	0s	0
__kmp_fork_barrier	libiomp5.so	0.187s	0.187s	2	0s	0
syscall	libc-2.17.so	0.007s	0s	0	0.007s	31
func@0xffffffff8176ec40	vmlinux	0.007s	0s	0	0.007s	1
__kmp_affinity_set_init_mask	libiomp5.so	0.001s	0.001s	2	0s	0
[Others]		0.001s	0.001s	9	0.000s	13

N/A is applied to non-summable metrics.