Introduction to Performance Optimization and Tuning Tools

Steve Lantz, Cornell University

CoDaS-HEP Summer School, July 24, 2024

Cornell University
Center for Advanced Computing

with thanks to Bei Wang, NVIDIA

Goals

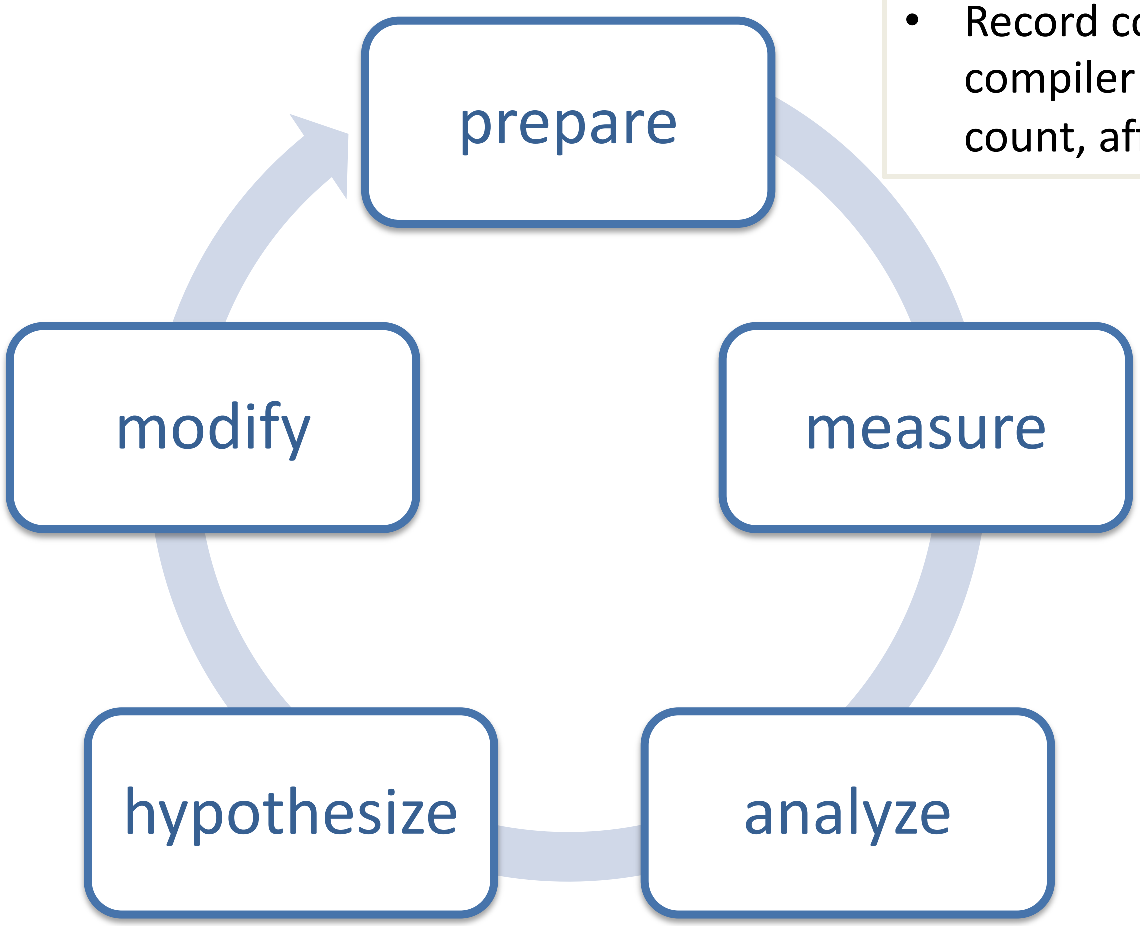
- Give an overview of what is meant by performance optimization and tuning
- Provide basic guidance on how to understand the performance of a code using tools
- Provide a starting point for performance optimizations

Performance Optimization: What Is It? Why Do It?

- What is performance optimization?
  - **Improving the efficiency** of an application to make better use of a given hardware resource
  - **Identifying bottlenecks** and eliminating them where possible, then rechecking metrics until performance objectives are satisfied
  - **Modifying code** guided by one’s understanding of the performance features of the given hardware (see prior presentations on “What Every Computational Physicist Should Know About Computer Architecture” and “Vector Parallelism on Multi-Core Processors”)
- Why does performance optimization 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**: Git is your friend!
- 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 of relevant 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

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?

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.

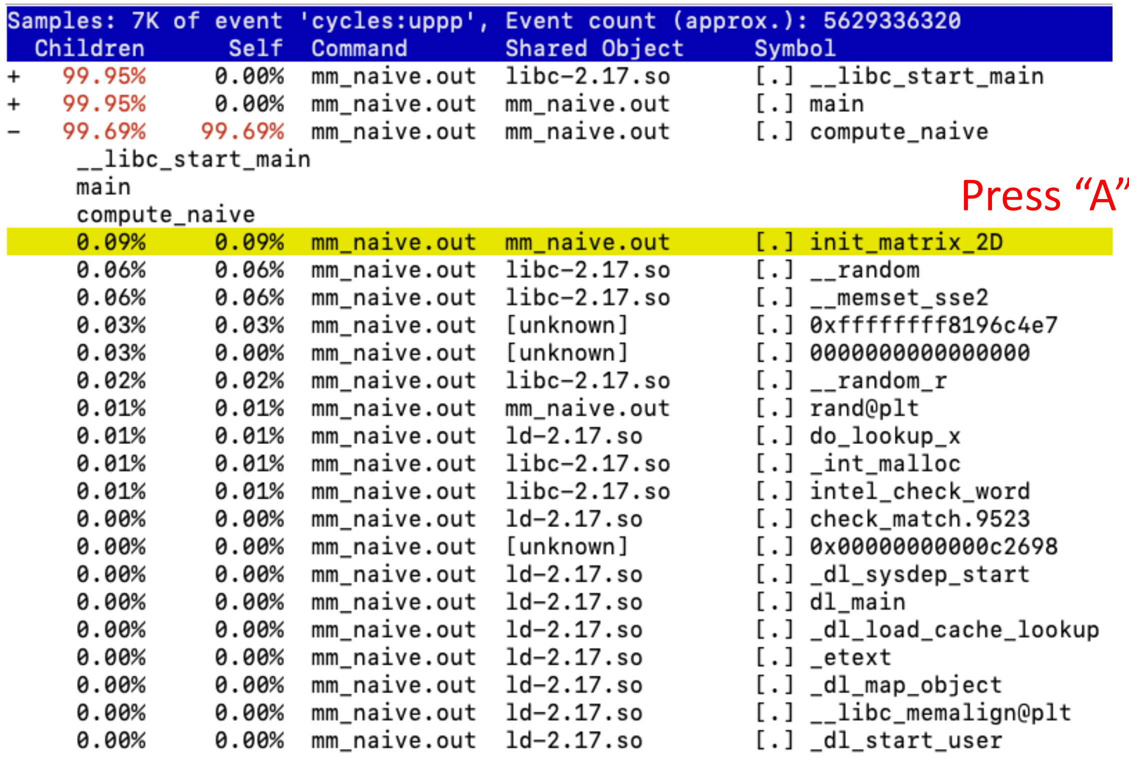
Linux Tool: *perf*

- 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

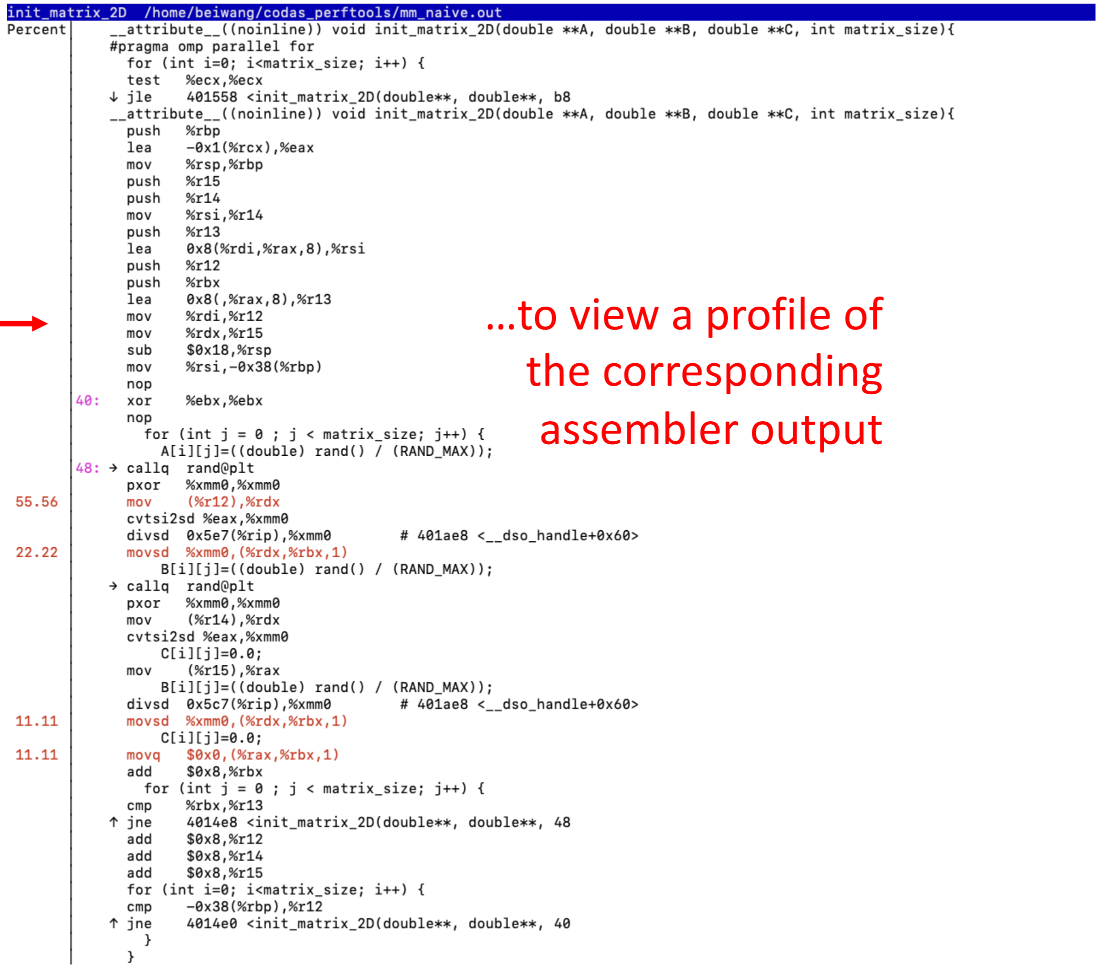
Example: Finding Hotspots with *perf*

- 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"…

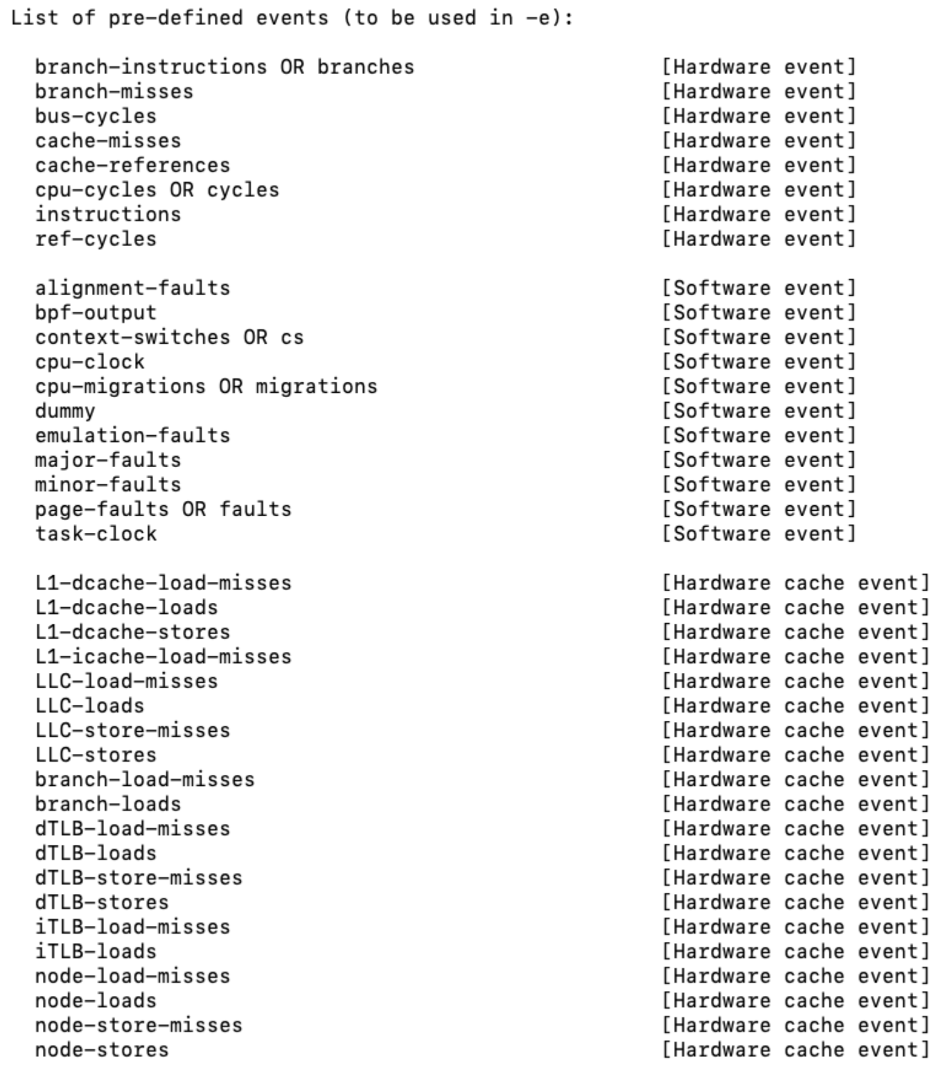


```
init_matrix_2D  /home/beiwang/codas_perftools/mm_naive.out
Percent│      __attribute__((noinline)) void init_matrix_2D(double **A, double **B, double **C, int matrix_size){
       │      #pragma omp parallel for
       │        for (int i=0; i<matrix_size; i++) {
       │        test   %ecx,%ecx
       │      ↓ 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
       │          }
       │        }
```

…to view a profile of the corresponding assembler output

Cornell University
Center for Advanced Computing

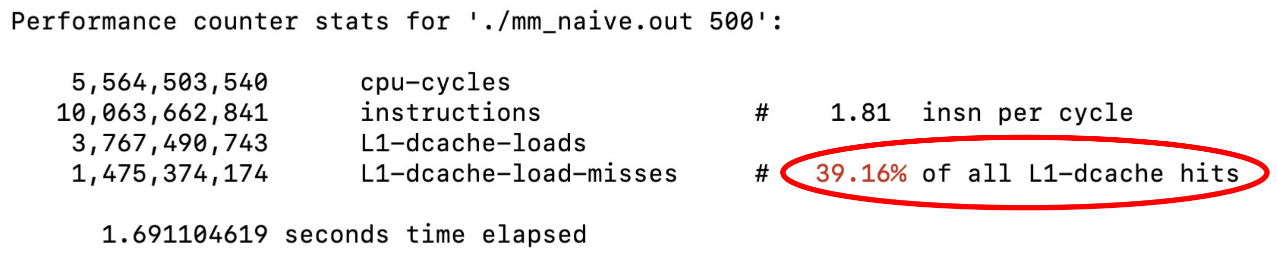
Example: Counting Cache Misses with *perf stat*



```
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.

Cornell University
Center for Advanced Computing

How Do I Fix It? Typical Pitfalls on a Single Core

- 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
```

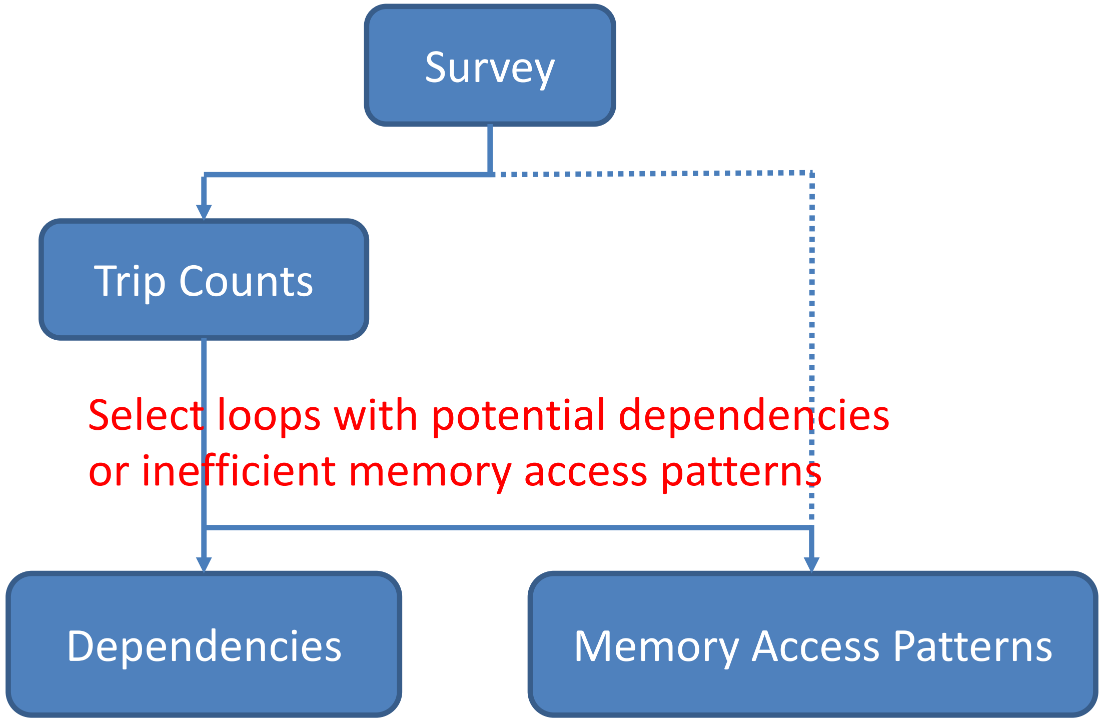
Intel Advisor

Two very useful analyses in Intel Advisor will be highlighted:

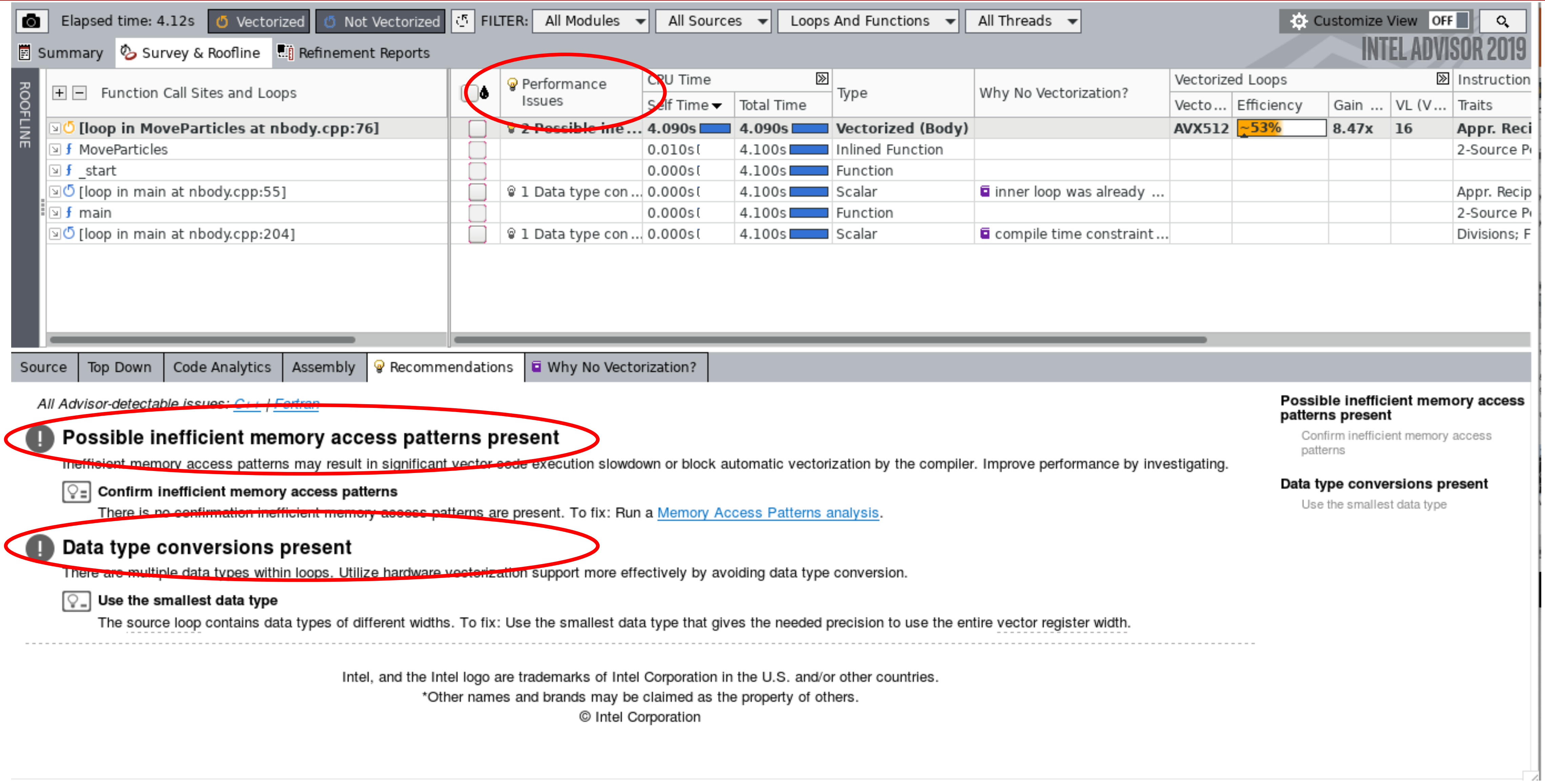
- Vectorization advisor
  - Identify the hotspots where your efforts pay off the most
  - Provide call graph information
  - Check memory access pattern, dependencies, more
  - Provide vectorization information from vectorization report
  - *Identify the performance and vectorization issues*
- Roofline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

Workflow of Vectorization Advisor

- **Survey**: profile where the code spends most of its time; then, combine it with vectorization reports from loops to provide suggestions for improvement
- **Trip Counts**: count all operations to generate a roofline plot
- **Memory Access Patterns** (MAP): see how you access the data
- **Dependencies**: determine if it is safe to force vectorization


Survey
Trip Counts
Select loops with potential dependencies or inefficient memory access patterns
Dependencies
Memory Access Patterns

Advisor Advises You About Performance Issues



Function Call Sites and Loops		Performance Issues	CPU Time		Type	Why No Vectorization?	Vectorized Loops				Instruction
			Self Time	Total Time			Vecto...	Efficiency	Gain ...	VL (V...	Traits
[loop in MoveParticles at nbody.cpp:76]		**2 Possible ine...**	**4.090s**	**4.090s**	**Vectorized (Body)**		**AVX512**	**~53%**	**8.47x**	**16**	**Appr. Reci**
MoveParticles			0.010s	4.100s	Inlined Function						2-Source P
_start			0.000s	4.100s	Function						
[loop in main at nbody.cpp:55]		1 Data type con...	0.000s	4.100s	Scalar	inner loop was already ...					Appr. Recip
main			0.000s	4.100s	Function						2-Source P
[loop in main at nbody.cpp:204]		1 Data type con...	0.000s	4.100s	Scalar	compile time constraint...					Divisions; F

All Advisor-detectable issues: C++ | Fortran

Possible inefficient memory access patterns present

Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.

Confirm inefficient memory access patterns

There is no confirmation inefficient memory access patterns are present. To fix: Run a Memory Access Patterns analysis.

Data type conversions present

There are multiple data types within loops. Utilize hardware vectorization support more effectively by avoiding data type conversion.

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.

Possible inefficient memory access patterns present

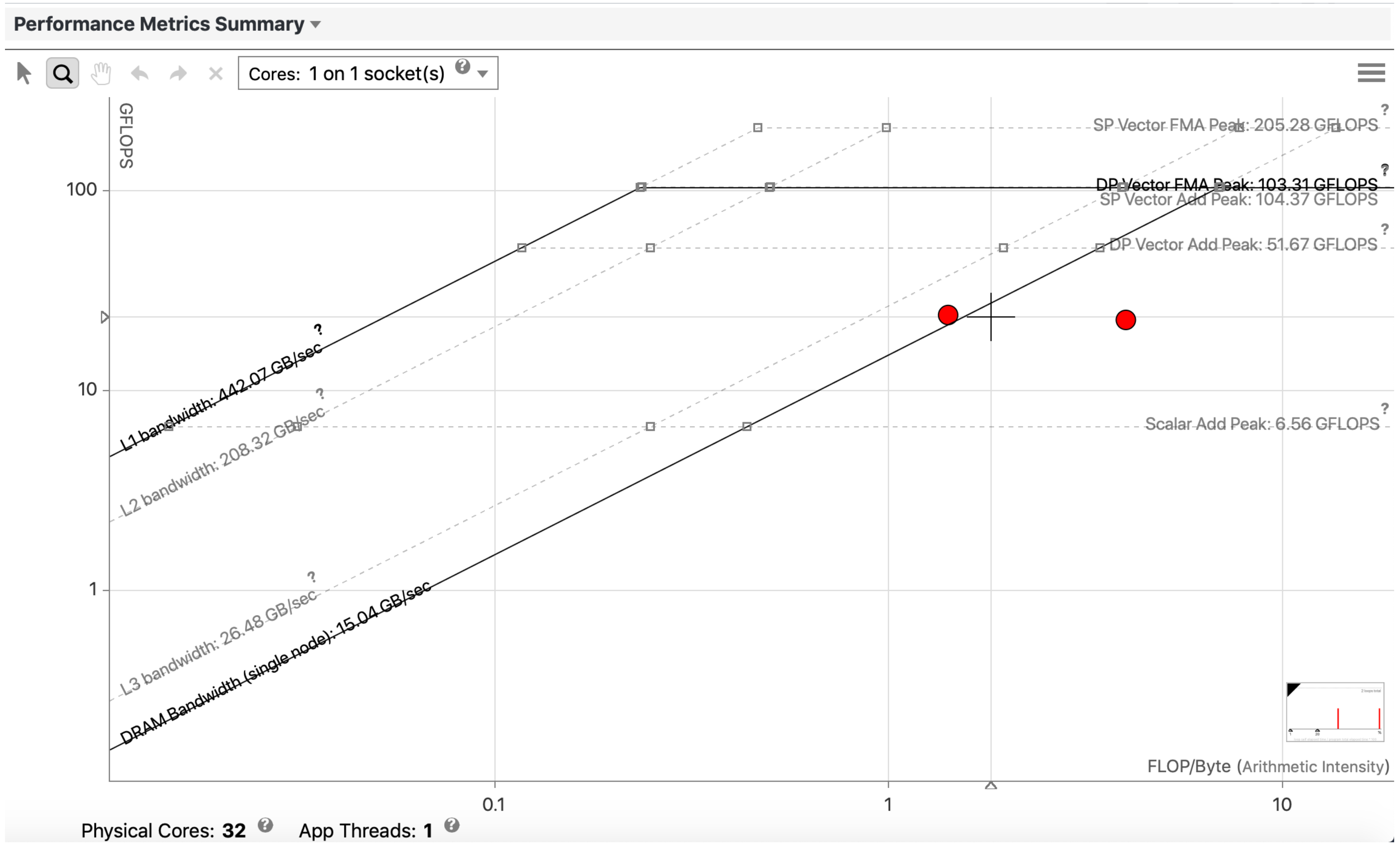
Confirm inefficient memory access patterns

Data type conversions present

Use the smallest data type

Intel, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
© Intel Corporation

Roofline Analysis: What Is It?



GFLOP/s:
Billions of floating-point operations per second

FLOP/byte: Arithmetic intensity

Cornell University
Center for Advanced Computing

What Does Roofline Analysis Tell You?

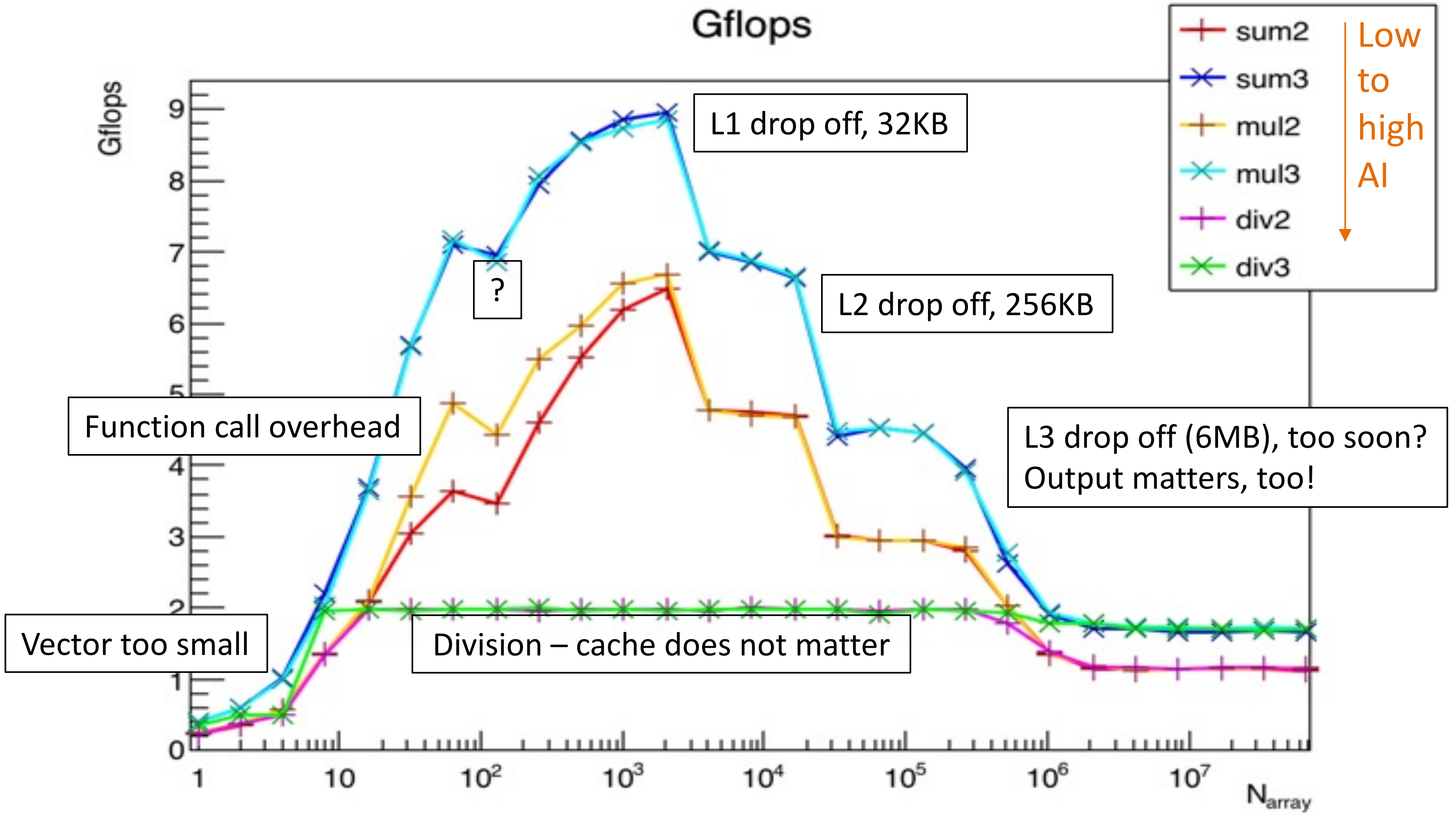
- The “roofline” is a performance ceiling related to *hardware characteristics*
  - It tells you the highest flop/s rate possible on a specific CPU for a given piece of code
- The *arithmetic intensity* or AI (flop/byte) of a code is a *software characteristic*
  - It tells you what part of the roofline(s) your code lies under
  - Code with low AI is limited by its need to load/store data from/to memory (sloped roofs)
  - Otherwise, it is limited by the maximum flop/s rate of the CPU (flat roofs)
  - Thus, roofline analysis tells you whether your code is *memory bound* or *compute bound*
- Investigate functions that fall way below the roofline for their AI value
  - Try to modify such functions so they approach the highest feasible roof
  - It can be shown that the AI needed to reach *theoretical peak* flop/s (the highest flat roof) implies that 50% of operands are vector constants: they never leave registers!

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
  - Again, 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 the AI value
- 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

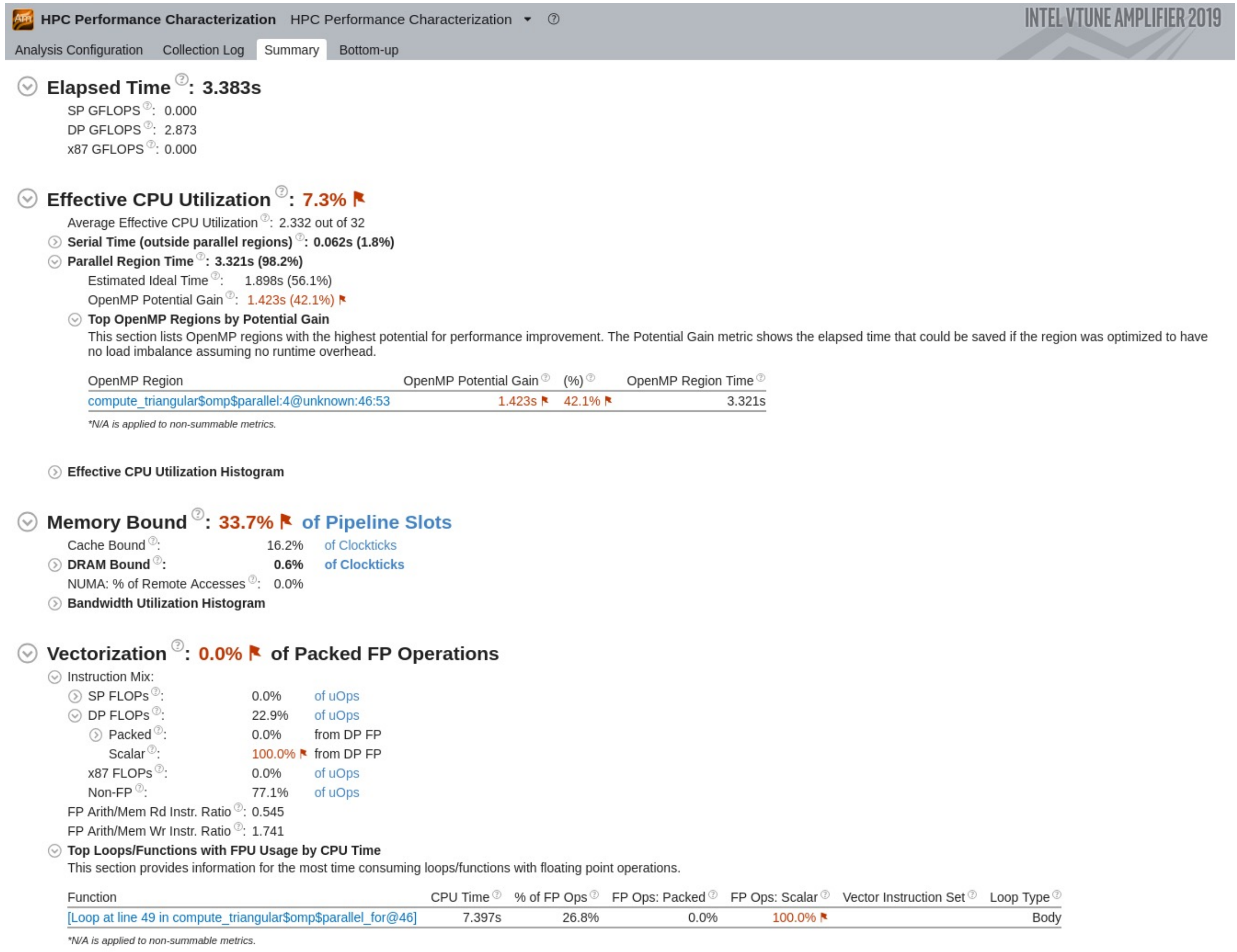
Effect of AI and Caches on GFLOP/s

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



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


HPC Performance Characterization HPC Performance Characterization
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration Collection Log Summary Bottom-up

Elapsed Time: 3.383s
SP GFLOPS: 0.000
DP GFLOPS: 2.873
x87 GFLOPS: 0.000

Effective CPU Utilization: 7.3%
Average Effective CPU Utilization: 2.332 out of 32
Serial Time (outside parallel regions): 0.062s (1.8%)
Parallel Region Time: 3.321s (98.2%)
Estimated Ideal Time: 1.898s (56.1%)
OpenMP Potential Gain: 1.423s (42.1%)
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_triangular$omp$parallel:4@unknown:46:53	1.423s	42.1%	3.321s

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

Effective CPU Utilization Histogram

Memory Bound: 33.7% of Pipeline Slots
Cache Bound: 16.2% of Clockticks
DRAM Bound: 0.6% of Clockticks
NUMA: % of Remote Accesses: 0.0%
Bandwidth Utilization Histogram

Vectorization: 0.0% of Packed FP Operations
Instruction Mix:
SP FLOPs: 0.0% of uOps
DP FLOPs: 22.9% of uOps
Packed: 0.0% from DP FP
Scalar: 100.0% from DP FP
x87 FLOPs: 0.0% of uOps
Non-FP: 77.1% of uOps
FP Arith/Mem Rd Instr. Ratio: 0.545
FP Arith/Mem Wr Instr. Ratio: 1.741
Top Loops/Functions with FPU Usage by CPU Time
This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time	% of FP Ops	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set	Loop Type
[Loop at line 49 in compute_triangular$omp$parallel_for@46]	7.397s	26.8%	0.0%	100.0%		Body

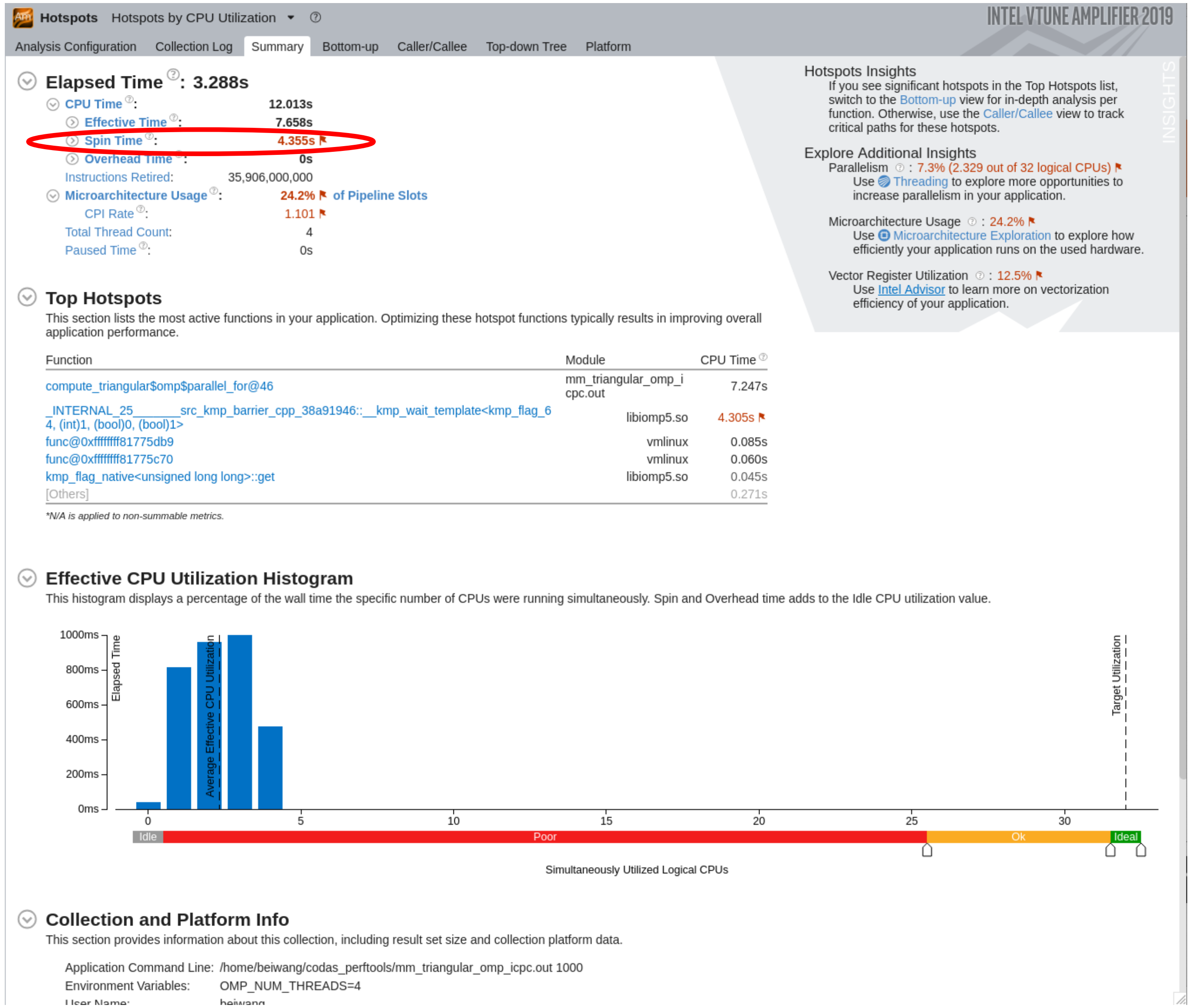
*N/A is applied to non-summable metrics.

Cornell University
Center for Advanced Computing

VTune is a (Very) Full System Profiler

- Memory Access Efficiency
  - Stalls by memory hierarchy; bandwidth utilization
  - Tip: Use Memory Access analysis
- Vectorization: FPU Utilization
  - FLOPS estimates from sampling
  - Tip: Use Intel Advisor for precise metrics and vectorization optimization
- Threading: CPU Utilization
  - Serial vs. parallel time; top OpenMP regions by potential gain
  - Tip: Use hotspot OpenMP region analysis for more detail
- MPI Imbalance Metric
  - Metric for performance of rank on critical path
  - Computational bottlenecks and outlier rank behavior

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

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_triangular$omp$parallel_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

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

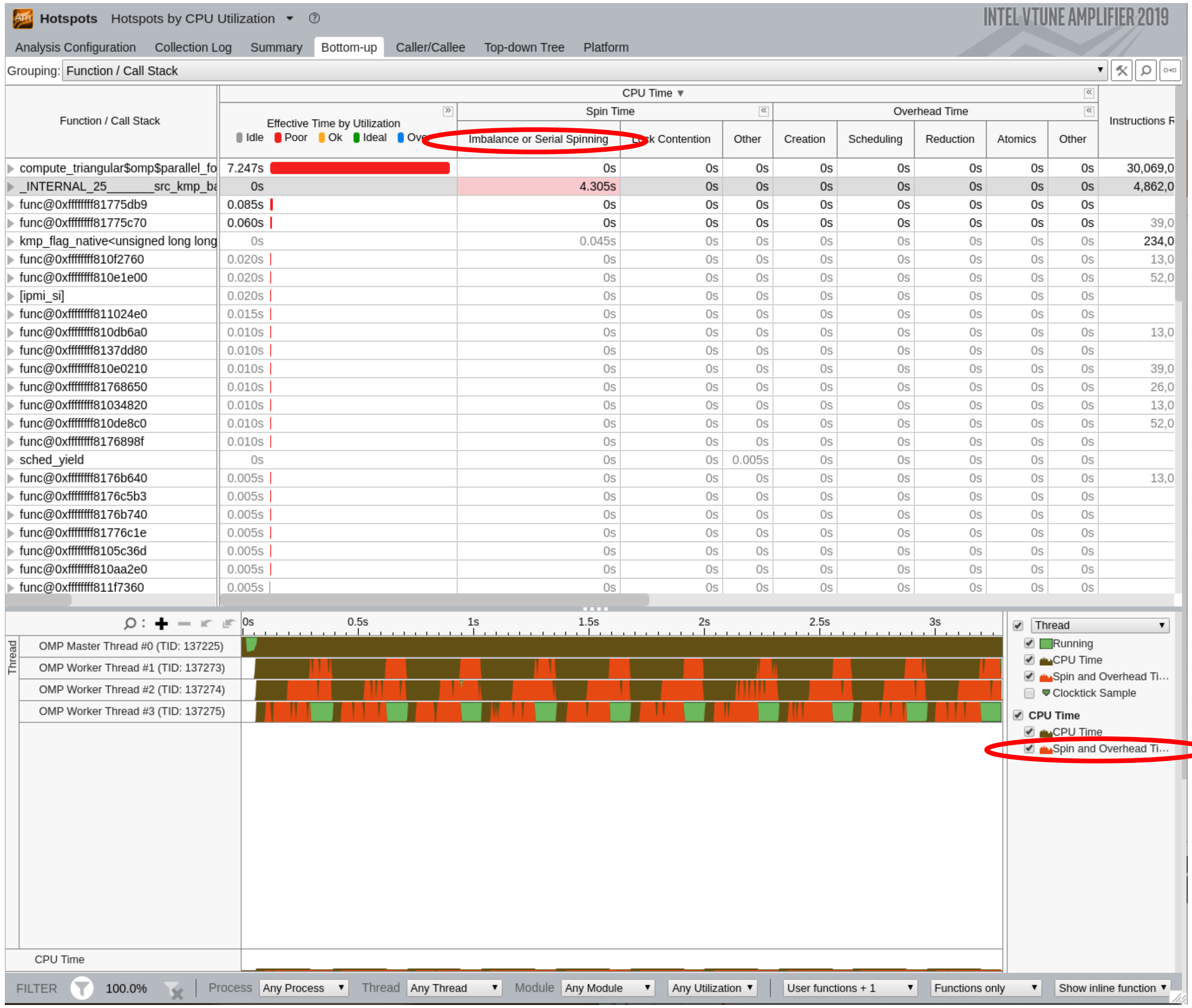
Parallelism : 7.3% (2.329 out of 32 logical CPUs)
Use Threading to explore more opportunities to increase parallelism in your application.

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

Thread Timelines Showing “Spin and Overhead”



Hotspots Hotspots by CPU Utilization INTEL VTUNE AMPLIFIER 2019

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

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	CPU Time: Spin Time: Imbalance or Serial Spinning	CPU Time: Spin Time: Lock Contention	CPU Time: Spin Time: Other	CPU Time: Overhead Time: Creation	CPU Time: Overhead Time: Scheduling	CPU Time: Overhead Time: Reduction	CPU Time: Overhead Time: Atomics	CPU Time: Overhead Time: Other	Instructions R
compute_triangular$omp$parallel_fo	7.247s	0s	0s	0s	0s	0s	0s	0s	0s	30,069,0
_INTERNAL_25________src_kmp_ba	0s	4.305s	0s	0s	0s	0s	0s	0s	0s	4,862,0
func@0xffffffff81775db9	0.085s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff81775c70	0.060s	0s	0s	0s	0s	0s	0s	0s	0s	39,0
kmp_flag_native<unsigned long long	0s	0.045s	0s	0s	0s	0s	0s	0s	0s	234,0
func@0xffffffff810f2760	0.020s	0s	0s	0s	0s	0s	0s	0s	0s	13,0
func@0xffffffff810e1e00	0.020s	0s	0s	0s	0s	0s	0s	0s	0s	52,0
[ipmi_si]	0.020s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff811024e0	0.015s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff810db6a0	0.010s	0s	0s	0s	0s	0s	0s	0s	0s	13,0
func@0xffffffff8137dd80	0.010s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff810e0210	0.010s	0s	0s	0s	0s	0s	0s	0s	0s	39,0
func@0xffffffff81768650	0.010s	0s	0s	0s	0s	0s	0s	0s	0s	26,0
func@0xffffffff81034820	0.010s	0s	0s	0s	0s	0s	0s	0s	0s	13,0
func@0xffffffff810de8c0	0.010s	0s	0s	0s	0s	0s	0s	0s	0s	52,0
func@0xffffffff8176898f	0.010s	0s	0s	0s	0s	0s	0s	0s	0s	
sched_yield	0s	0s	0s	0.005s	0s	0s	0s	0s	0s	
func@0xffffffff8176b640	0.005s	0s	0s	0s	0s	0s	0s	0s	0s	13,0
func@0xffffffff8176c5b3	0.005s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff8176b740	0.005s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff81776c1e	0.005s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff8105c36d	0.005s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff810aa2e0	0.005s	0s	0s	0s	0s	0s	0s	0s	0s	
func@0xffffffff811f7360	0.005s	0s	0s	0s	0s	0s	0s	0s	0s	

Thread
- OMP Master Thread #0 (TID: 137225)
- OMP Worker Thread #1 (TID: 137273)
- OMP Worker Thread #2 (TID: 137274)
- OMP Worker Thread #3 (TID: 137275)

CPU Time

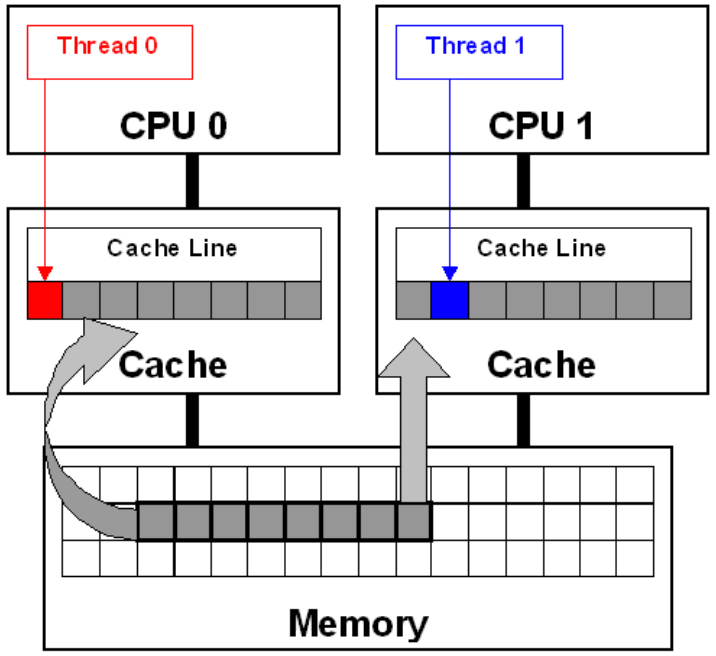
- ✓ Thread
  - ✓ Running
  - ✓ CPU Time
  - ✓ Spin and Overhead Ti...
  - Clocktick Sample
- ✓ CPU Time
  - ✓ CPU Time
  - ✓ Spin and Overhead Ti...

FILTER 100.0% | Process: Any Process | Thread: Any Thread | Module: Any Module | Any Utilization | User functions + 1 | Functions only | Show inline function

Cornell University
Center for Advanced Computing

How Do I Fix It? Typical Pitfalls in Multithreading

- Insufficient parallelism: not enough work to divide among threads
- False sharing: threads 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
- Load imbalance
  - Other threads wait idly for the overworked one
- Needless 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 (NUMA)



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