# INTRODUCTION TO PERFORMANCE TUNING AND OPTIMIZATION TOOLS

**Bei Wang**
beiwang@princeton.edu
Princeton University

Third Computational and Data Science School for HEP (CoDaS-HEP 2019)
July 25, 2019

# Outlines

- **Basic Concepts in Performance Tuning**
  - What is performance tuning and why it matters?
  - Performance tuning workflow
  - Typical pitfalls wrt. single node performance
  - Performance tool overview
- **Performance Tools: Demos and Hands-on**
  - How to run basic timing experiments and what they can do
  - How to use hardware counters
  - How to deal with parallelism (vectorization and threads)
- **Goals**
  - Provide basic guidance on how to understand the performance of a code using tools
  - Provide starting point for performance optimizations

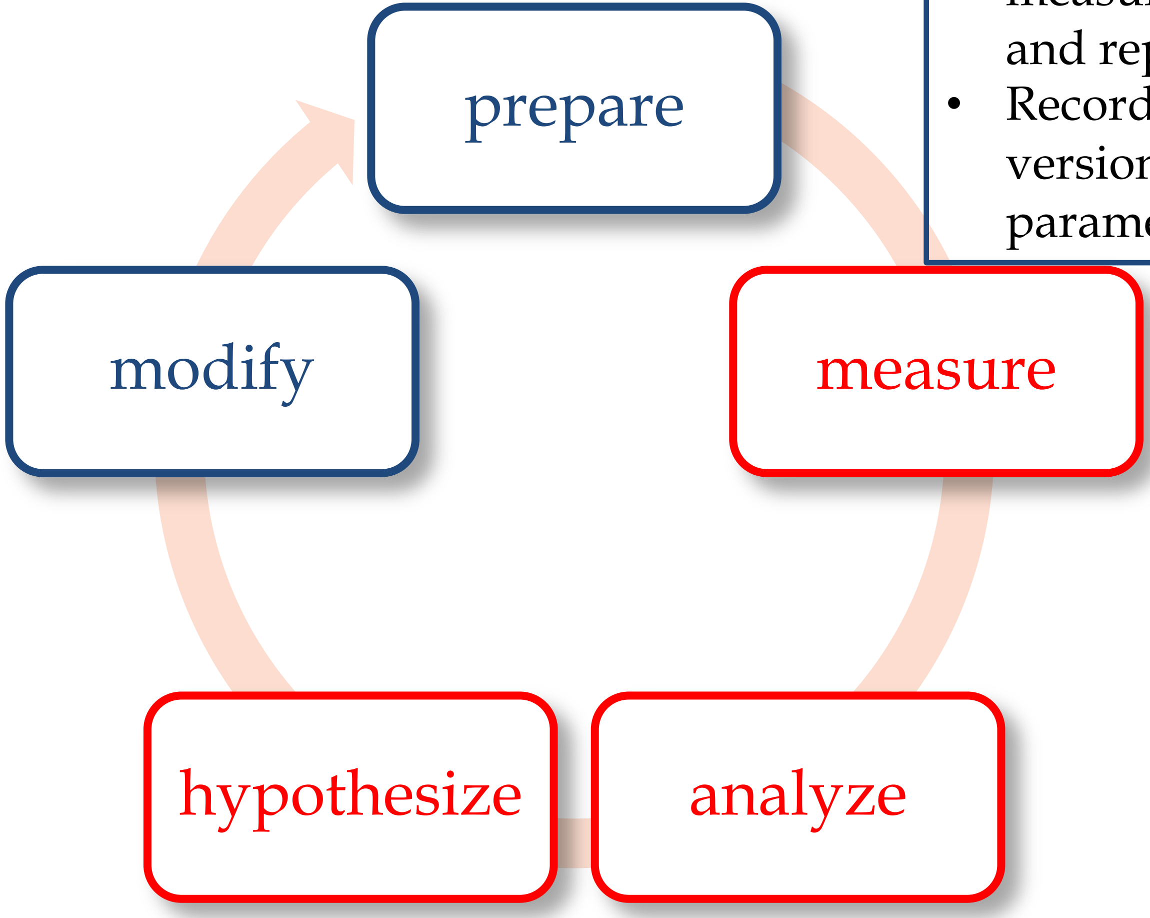
# BASIC CONCEPTS IN PERFORMANCE TUNING

# What Is and Why Performance Tuning?

- What is performance tuning?
  - The process of improving the efficiency of an application to better utilize a given hardware resource
    - Requires some understanding about the performance features of the given hardware (see CoDaS's talk "*what every physicist should know about computer architecture*" on Monday)
    - Identifying bottlenecks, determining efficiency and eliminating the bottlenecks if possible
    - Incrementally complete tuning until the performance requirements are satisfies
- Why performance matters?
  - Energy efficiency
  - Today's applications only use a fraction of the machine due to
    - Complex architectures
    - Mapping applications onto architectures is hard

# Performance Tuning Workflow

- 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 wrong result is completely irrelevant




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

# Measure

- What to measure? Choose metrics which quantify the performance of your code
  - Time, energy etc
- How to measure?
  - Linux "time" command
    - Get an idea of overall run time, but can't pin performance bottlenecks
  - Put timer (e.g., gettimeofday, MPI_Wtime, omp_get_wtime) around loops/functions
    - Works for small code base to identify hotspots, but hard to maintain and require significant priori knowledge
  - Performance tools (recommended)
    - Collect a lot data with varying granularity, cost and accuracy
    - Trace back to source code (use –g compiler flag)
    - How to collect

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

    - Sometime there is a learning curve to master the tools

# Performance Tools Overview

- **Basic OS tools**
  - Time
  - Gprof/perf
  - Valgrind/callgrind
- **Hardware counter**
  - PAPI API & tool set
- **Community open source**
  - HPCToolkit (Rice Univ.)
  - TAU (U of Oregon)
  - Open|SpeedShop (Krell)
- **Commercial products**
  - ARM MAP
  - Intel VTune Amplifier
  - Intel Advisor
  - Intel Trace Analyzer
- **Vendor supplied (free)**
  - CrayPat
  - Nvprof/pgprof

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

# Typical Pitfalls wrt. Performance: Sequential

- Where am I spending my time?
  - Find the hotspots
- Is my code computational or memory bounded?
  - Memory bounded
    - Data locality
    - TLB misses
    - L1/L2/L3 $ misses

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

  - Computational bounded
    - Fast math (see CoDaS's talk "*Floating Point Arithmetic*" on Wed)
    - Avoid type conversion

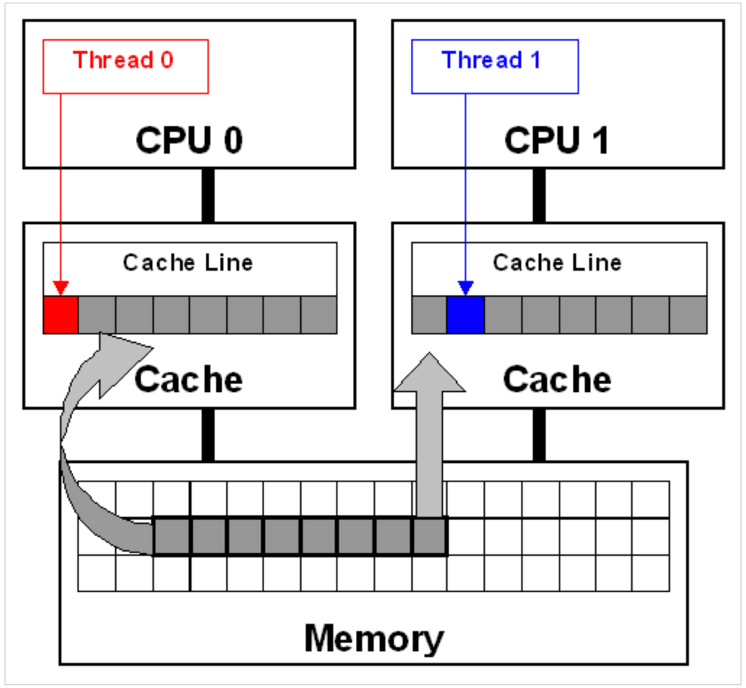
```
float x=3.14;    //bad: 3.14 is a double
float s=sin(x); //bad: sin() is a double precision function
long v=round(x); //bad: round takes and returns double
```

```
float x=3.14f;    //good: 31.4f 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
```

    - Vectorization efficiency
- Is my I/O efficient?

# Typical Pitfalls wrt. Performance: Multithreading

- Load imbalance
- False sharing
  - Occurs when threads on different processors modify variables that reside on the same cache line
  - Caused by coherent caches
  - Cache line is 64 bytes wide
- Insufficient parallelism
- Synchronization
  - Avoid synchronization with private thread storage
- Non-optimal memory placement
  - Thread affinity
  - Allocation on first touch




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

# LINUX TOOL: Perf

# PERF

- Perf is a performance analyzing tool in Linux, available in version 2.6.31
- How does it work
  - *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 list*: list available events for measurement
    - Support a list of hardware and software events
  - *perf stat*: measure total event count for a single program
    - -e: event names provided in perf list
  - *etc*
- When compiling the code, use the following flags for easier interpretation
  - -g: need debug symbols in order to annotation source
  - -fno-omit-frame-pointer: provide stack chain/backtrace

# Example: Matrix-Matrix Multiplication

Two versions of 2D matrix-matrix multiplication

```
int main(int argc, char *argv[])
{
  int matrix_size; //N*N matrix
  int max_iters=10; //number of times to call a matrix-matrix function

  //read command line input
  //set various paramaters
  if(argc<2) {
    cout<<"ERROR: expecting integer matrix size, i.e., N for NxN matrix"<<endl;
    exit(1);
  }
  else {
    matrix_size=atoi(argv[1]);
  }

  cout<<"using matrix size:"<<matrix_size<<endl;

  double **A, **B, **C; //2D arrays

  create_matrix_2D(A, B, C, matrix_size);

  init_matrix_2D(A, B, C, matrix_size);

  for (int r=0; r < max_itersl r++) {
    zero_result(C,matrix_size);
#ifdef NAIVE
    compute_naive(A,B,C,matrix_size);
#elif INTERCHANGE
    compute_interchange(A,B,C,matrix_size);
#endif
  }

  free_matrix_2D(A, B, C, matrix_size);

  return 0;
}
```

```
//NAIVE: 2D matrix-matrix multiplication
__attribute__((noinline)) void compute_naive(double **A, double **B, double **C, int matrix_size) {
  for (int i = 0 ; i < matrix_size; i++) {
    for (int j = 0;  j < matrix_size; j++) {
      for (int k = 0; k < matrix_size; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}

//INTERCHANGE: 2D matrix-matrix multiplication
__attribute__((noinline)) void compute_interchange(double **A, double **B, double **C, int matrix_size) {
  for (int i = 0 ; i < matrix_size; i++) {
    for (int k = 0; k < matrix_size; k++) {
      for (int j = 0;  j < matrix_size; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

# Set up Adroit for Hands-on

- How to log into the Adroit system
  - Login information was distributed on Monday
- Download the exercises from Github
  - *git clone https://github.com/beiwang2003/codas_perftools.git*
- Move to the codas_perftools directory
  - *cd $HOME/codas_perftools*
- Load environment module
  - *module load rh/devtoolset/7*

# Hands-on: Find Hot Spots Using 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
        │         }
        │       }
```

# Hands-on: Loop Interchange Optimization

- The *perf list* command lists all available CPU counters:

```
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]
```

- Check *man perf_event_open* to see what does each event measure

- The *perf stat* command instruments and summarizes selected CPU counters
  1. Compile the code
     - *g++ -g -fno-omit-frame-pointer -O3 -DNAIVE matmul_2D.cpp -o mm_naive.out*
  2. Run perf stat
     - *perf stat -e cpu-cycles,instructions,L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores ./mm_naive.out 500*
  3. Record the numbers for each events
  4. Compile the code
     - *g++ -g -fno-omit-frame-pointer -O3 -DINTERCHANGE matmul_2D.cpp -o mm_interchange.out*
  5. Run perf stat
     - *perf stat -e cpu-cycles,instructions,L1-dcache-loads,L1-dcache-load-misses ./mm_interchange.out 500*
  6. Compare the numbers for both cases

# Results Comparison (GCC)

NAIVE

```
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
```

INTERCHANGE

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

    2,589,454,237      cpu-cycles
    8,869,823,983      instructions              #    3.43  insn per cycle
    3,143,807,817      L1-dcache-loads
      164,669,312      L1-dcache-load-misses     #    5.24% of all L1-dcache hits

      0.787522929 seconds time elapsed
```

- The number of CPU cycles is much lowers for interchange, reflecting its shorter elapsed time
- The number of instructions are half in interchange
- Interchange has substantial fewer LL1 load misses, which indicates better data locality

Follow up exercise: change matrix dimension to 1000x1000. This will trigger more LLC and TLB misses.

# OPEN|SpeedShop

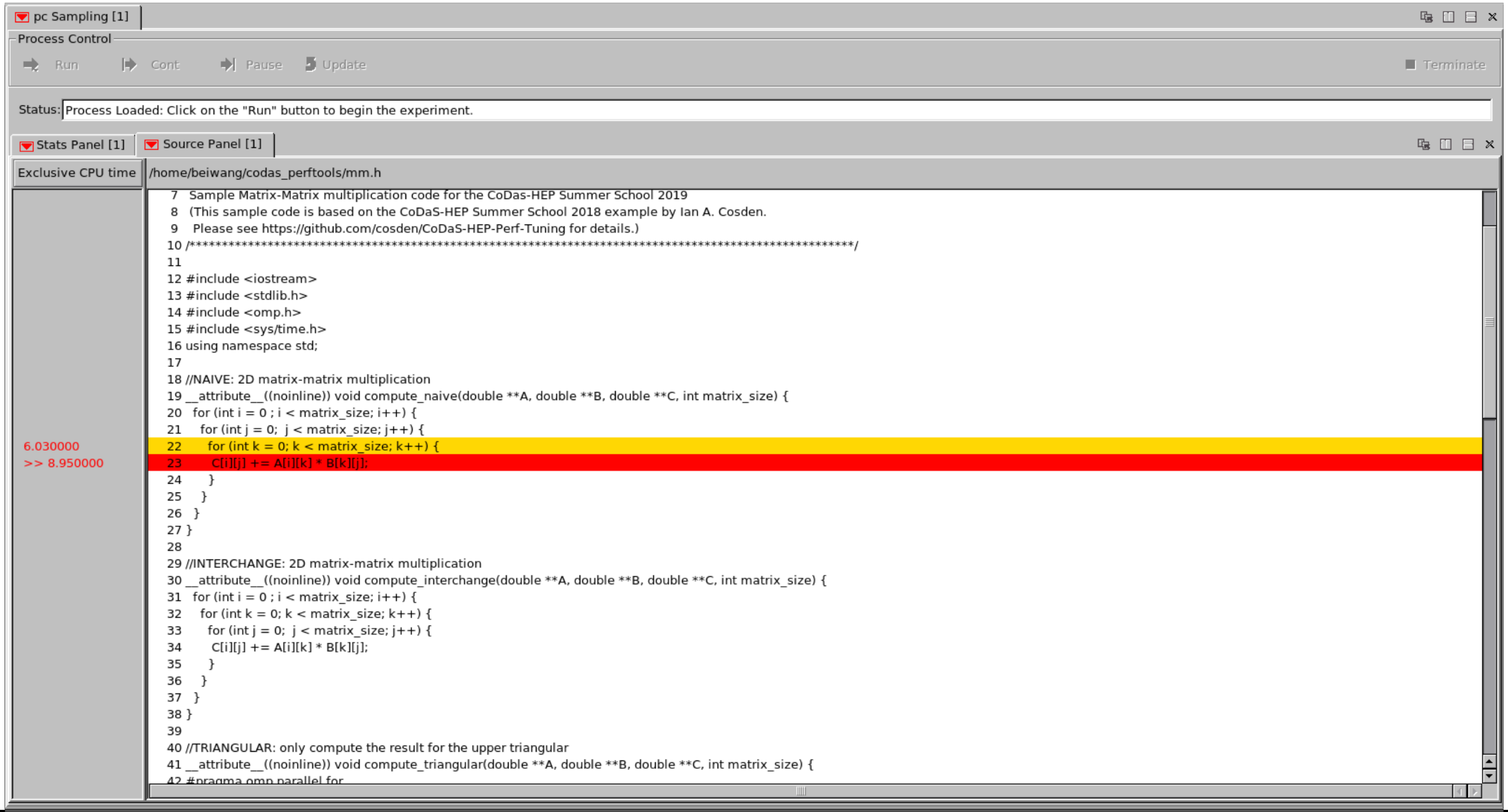
# OpenSpeedShop (O|SS)

openspeedshop.org

- Open source multi-platform performance tool
  - Available on Intel, AMD, ARM, Power PC, Power 8, GPU based systems
  - Built on top of a list of community tools, e.g., Dyninst and MRNet from UW, libmonitor from Rice, and PAPI from UTK
- O|SS gathers
  - High level summary: *cbtfsummary “normal app run script”*
  - Program counter sampling: *osspcsamp “…”*
  - Call path analysis: *ossusertime “…”*
  - Hardware performance counters: *osshwcsamp “…”*
  - OpenMP profiling and analysis: *ossomptp “…”*
  - MPI profiling and tracing: *ossmpi[p][t] “…”*
  - I/O profiling and tracing: *ossio[p][t] “…”*
  - Memory analysis: *ossmem “…”*
  - Nvidia CUDA tracing and analysis
- O|SS displays with
  - GUI: openss –f ./*.openss
  - CLI: openss –cli –f ./*.openss

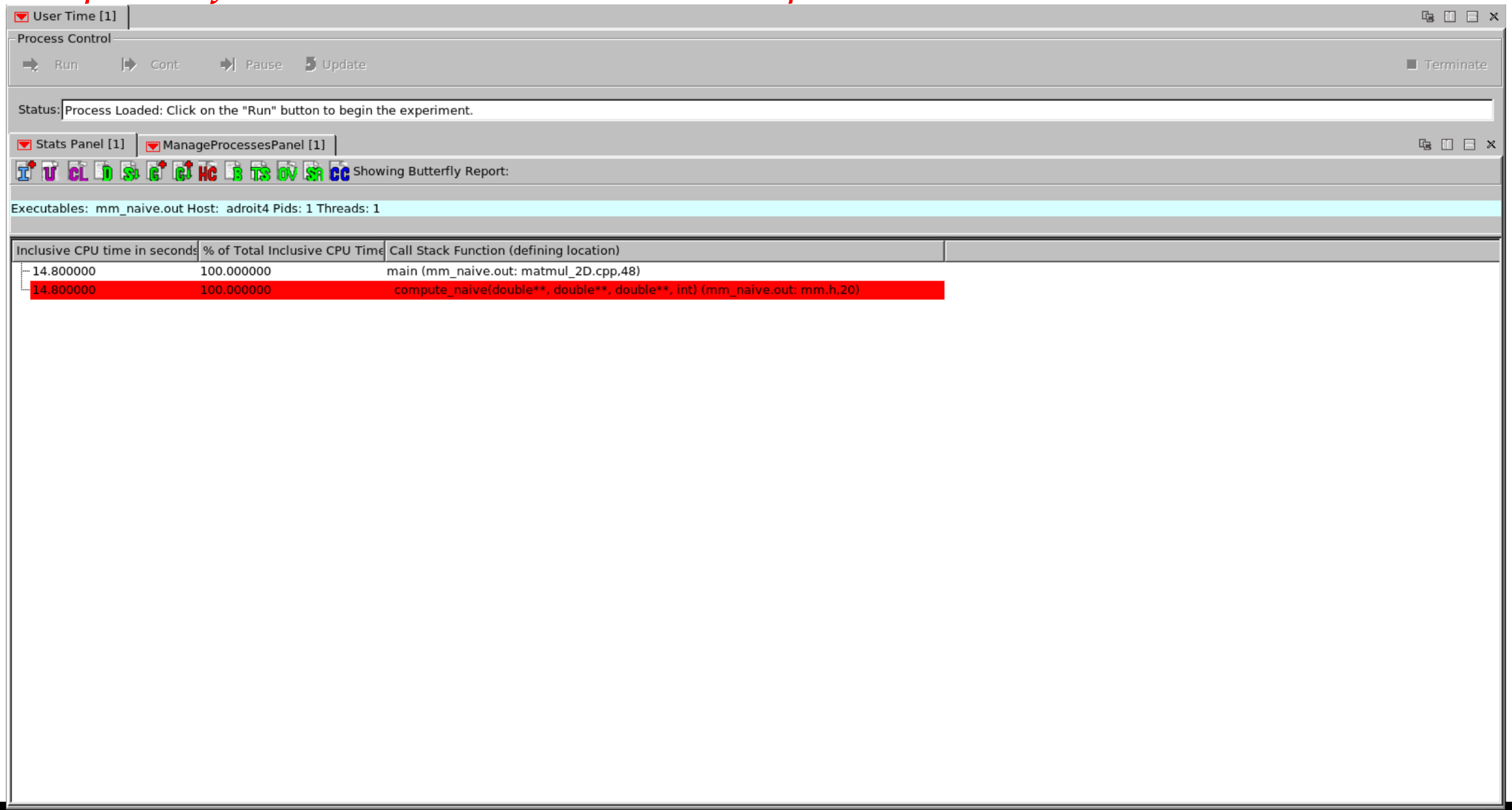
# Osspcsamp: Flat Profile Overview

- Start with flat profile overview
- Flat profile: time spent per functions or per statements
- Collect profiling data: *osspcsamp "./mm_naive.out 1000"* (this will generate a *.openss file)
- Open the result in GUI: *openss -f ./mm_naive.out-pcsamp-0.openss*



# Ossusertime: Call Graph Analysis

- Flat profile does not help you:
  - Distinguish routines called from multiple callers
  - Understand the call invocation history
- Stack traces: caller/callee relationships, inclusive/exclusive time
- Collect profiling data: *ossusertime "./mm_naive.out 1000"* (this will generate a *.openss file)
- Open the result in GUI: *openss -f ./mm_naive.out-usertime-0.openss*



# Osshwcsamp: Hardware Performance Counters

- Timing information shows where you spend your time. BUT, it doesn't show you why
- Hardware performance counters: PAPI events (use papi_avail to check available events)
- Collect profiling data: *osshwcsamp "./mm_naive.out 1000" PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_L1_DCM* (up to 6 events, this will generate a *.openss file)
- Open the result in CLI: *openss –cli –f ./mm_naive.out-hwcsamp-0.openss*
- View the result with: openss>>*expview*

```
[beiwang@adroit4 codas_perftools]$ openss -cli "mm_naive.out-hwcsamp-1.openss"
openss>>[openss]: The restored experiment identifier is:  -x 1
openss>>expview

Exclusive    % of CPU  papi_tot_cyc  papi_tot_ins  papi_l1_dcm     Comp.  papi_tot_cyc%  Function (defining location)
 CPU time        Time                                         Intensity
       in
 seconds.
14.480000   99.724518  47291712380  80007419256  13917395159  1.691785      99.804480  compute_naive(double**, double**, double**, int) (mm_naive.out: mm.h,20)
 0.010000    0.068871     23214861     59870626       165350  2.578978       0.048993  __random (libc-2.17.so)
 0.010000    0.068871      9831635     19114751        68522  1.944209       0.020749  __random_r (libc-2.17.so)
 0.010000    0.068871     32721784     53848956      9349092  1.645661       0.069056  __memset_sse2 (libc-2.17.so)
 0.010000    0.068871     26877580     44800904      7709482  1.666850       0.056722  __brk (libc-2.17.so)
14.520000  100.000000  47384358240  80185054493  13934687605  1.692226     100.000000  Report Summary
openss>>
```

# Ossomptp: OpenMP Parallel Region

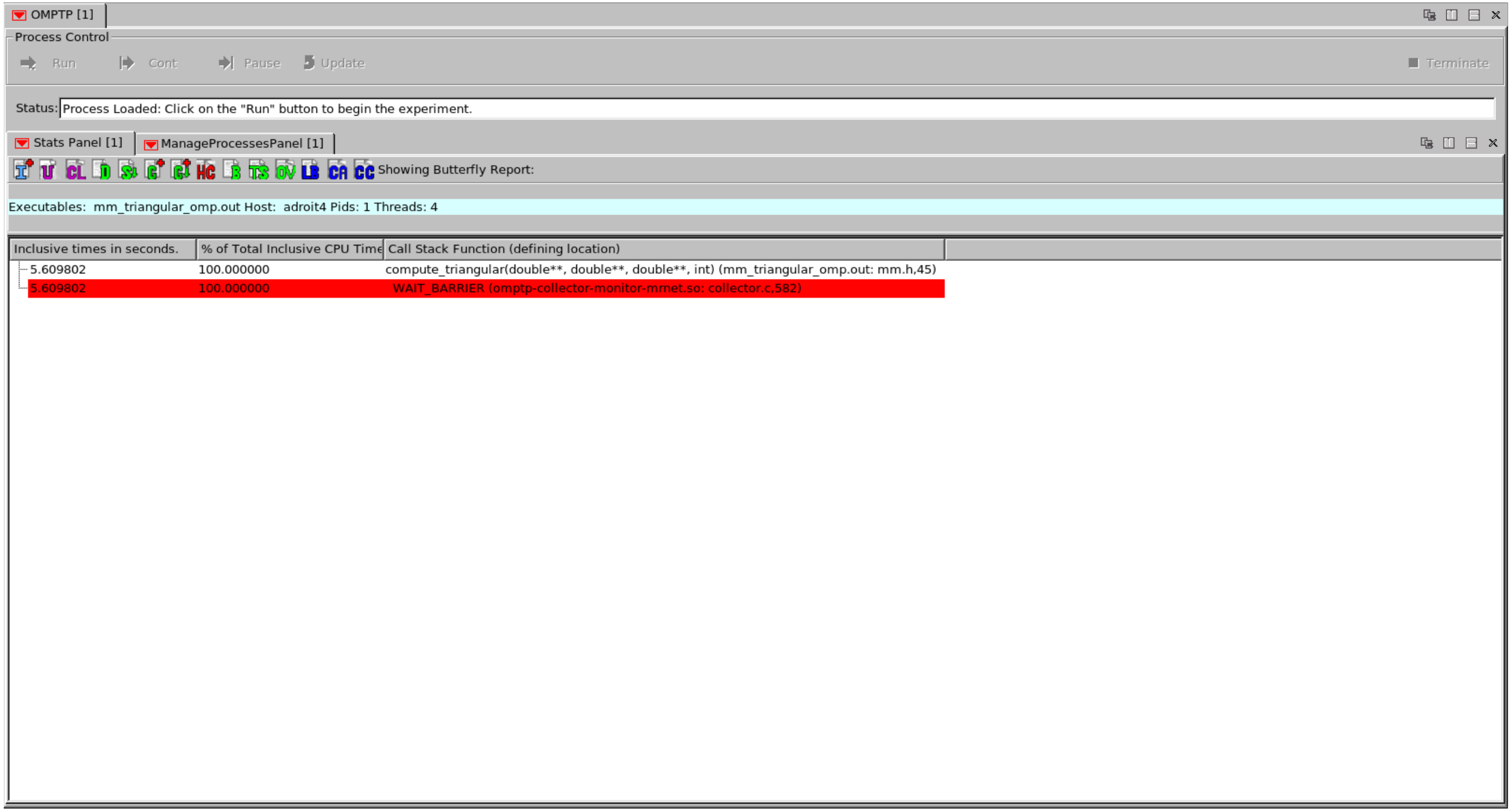
- For parallel execution, is there any load imbalance issue? How do you find the potential cause?
- OMPT API: record task time, idleness, barrier, wait barrier per OpenMP parallel region
- Let's look at the matrix-matrix example, but now we only compute the result for the upper triangular

```
//TRIANGULAR: only compute the result for the upper triangular
__attribute__((noinline)) void compute_triangular(double **A, double **B, double **C, int matrix_size) {
#pragma omp parallel for
  for (int i = 0 ; i < matrix_size; i++) {
    for (int j = 0;  j < matrix_size-i; j++) {
      for (int k = 0; k < matrix_size; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

- Compile the code: *g++ -g -O3 –fopenmp –DTRIANGULAR matmul_2D.cpp -o mm_triangular_omp.out* (export OMP_NUM_THREADS=4)
- Collect profiling data: *ossomptp "./mm_triangular_omp.out 1000"* (this will generate a *.openss file)

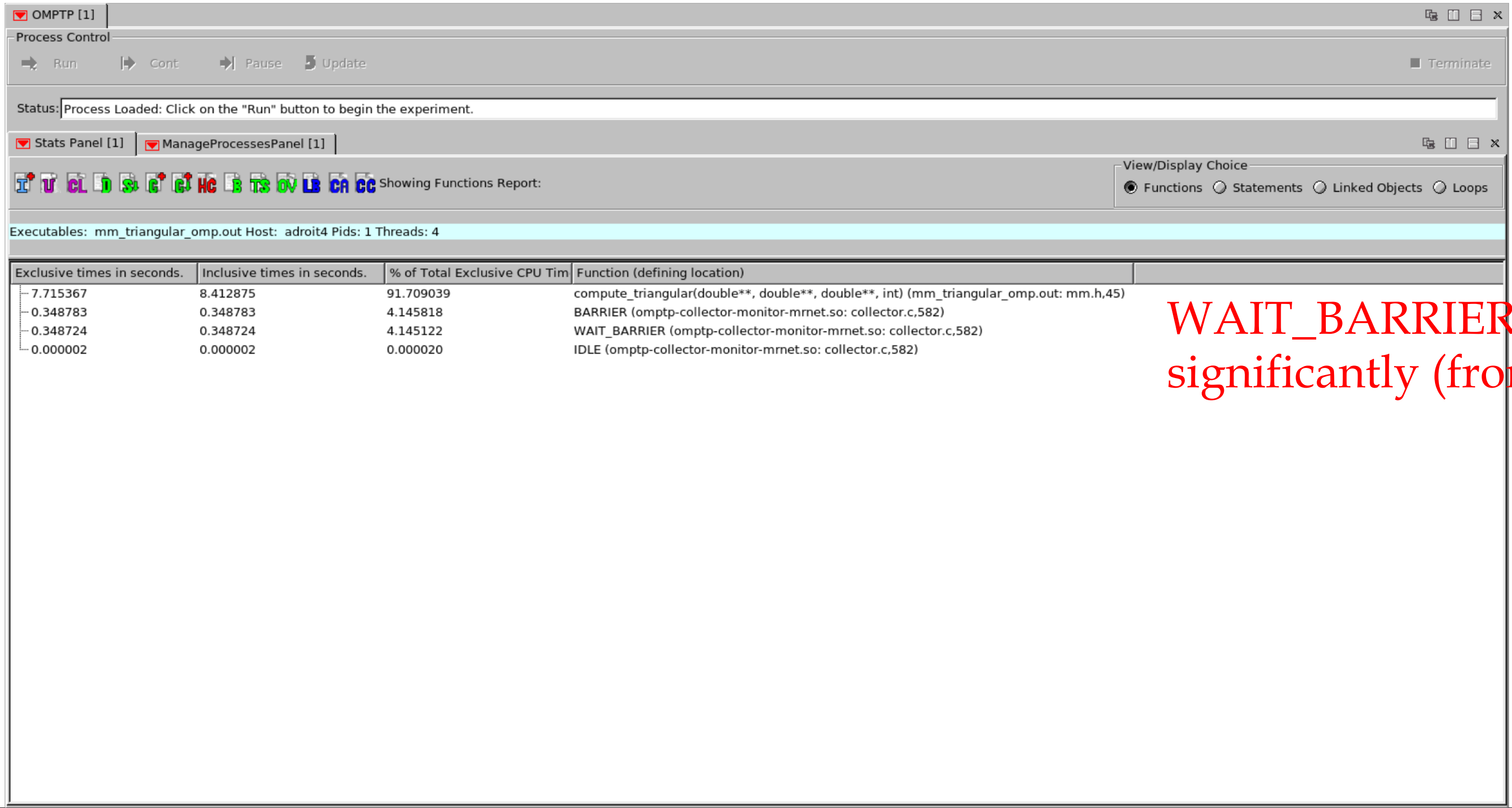
# Ossomptp: OpenMP Parallel Region

- Open the result in GUI: *openss –f ./mm_triangular_omp.out-omptp-0.openss*



# Using OMP Clause "schedule(dynamic)"

```
//TRIANGULAR: only compute the result for the upper triangular
__attribute__((noinline)) void compute_triangular(double **A, double **B, double **C, int matrix_size) {
#pragma omp parallel for schedule(dynamic)
  for (int i = 0 ; i < matrix_size; i++) {
    for (int j = 0;  j < matrix_size-i; j++) {
      for (int k = 0; k < matrix_size; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
#endif
```




| Exclusive times in seconds. | Inclusive times in seconds. | % of Total Exclusive CPU Tim | Function (defining location) |
|---|---|---|---|
| 7.715367 | 8.412875 | 91.709039 | compute_triangular(double**, double**, double**, int) (mm_triangular_omp.out: mm.h,45) |
| 0.348783 | 0.348783 | 4.145818 | BARRIER (omptp-collector-monitor-mrnet.so: collector.c,582) |
| 0.348724 | 0.348724 | 4.145122 | WAIT_BARRIER (omptp-collector-monitor-mrnet.so: collector.c,582) |
| 0.000002 | 0.000002 | 0.000020 | IDLE (omptp-collector-monitor-mrnet.so: collector.c,582) |

WAIT_BARRIER time has reduced significantly (from 5.6s to 0.35s)

# Another Important Focus: Efficient Vectorization

- The CoDaS's talk "*Vector Parallelism for Kalman-Filter-Based Particle Tracking on Multi- and Many-Core Processors*" has covered many important aspects of vectorization
- This lecture will mainly focus on how to examine vectorization efficient using tools, e.g., Intel Advisor
- Analysis tools:
  - Compiler vectorization report
    - GCC: -fopt-info-vec
    - Intel: -qopt-report=5
  - Look at assembly code
  - Measure performance with PAPI counters, e.g., PAPI_DP_OPS, PAPI_VEC_DP etc
  - Intel Advisor

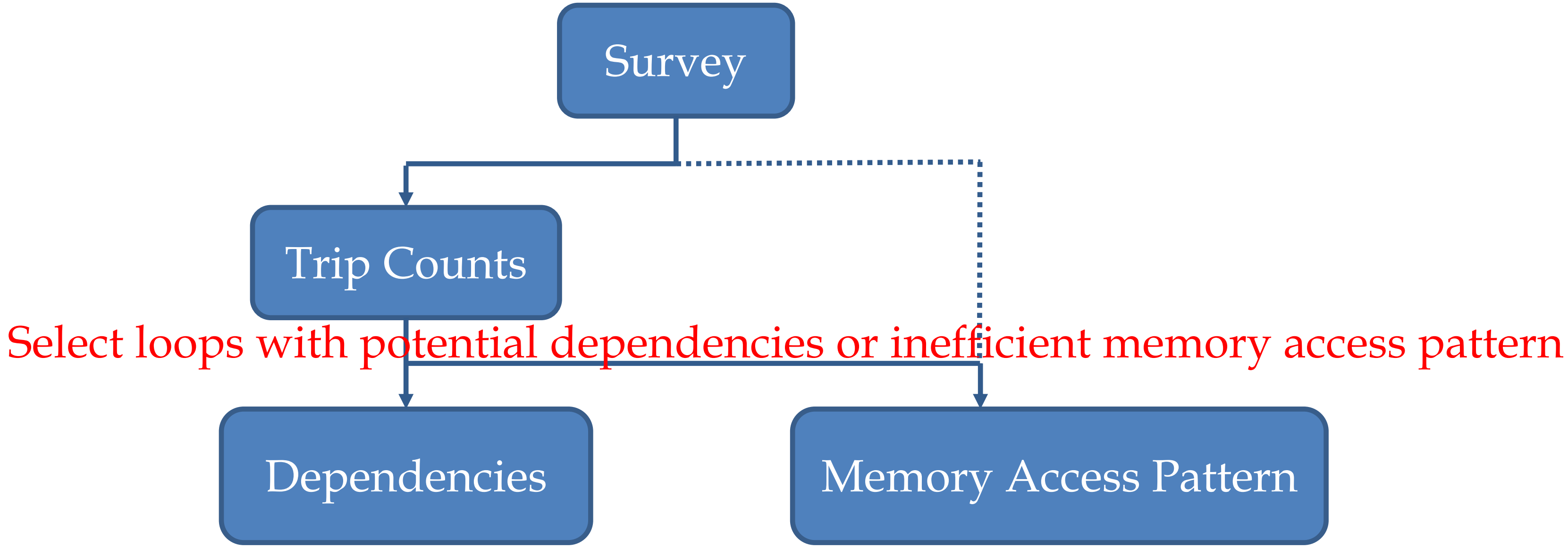
# Intel Advisor

# Vectorization Advisor & Roofline

- 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
  - Check dependencies
  - More …
- Roofline
  - How much performance is being left on the table
  - Where are the bottlenecks
  - Which can be improved
  - Which are worth improving

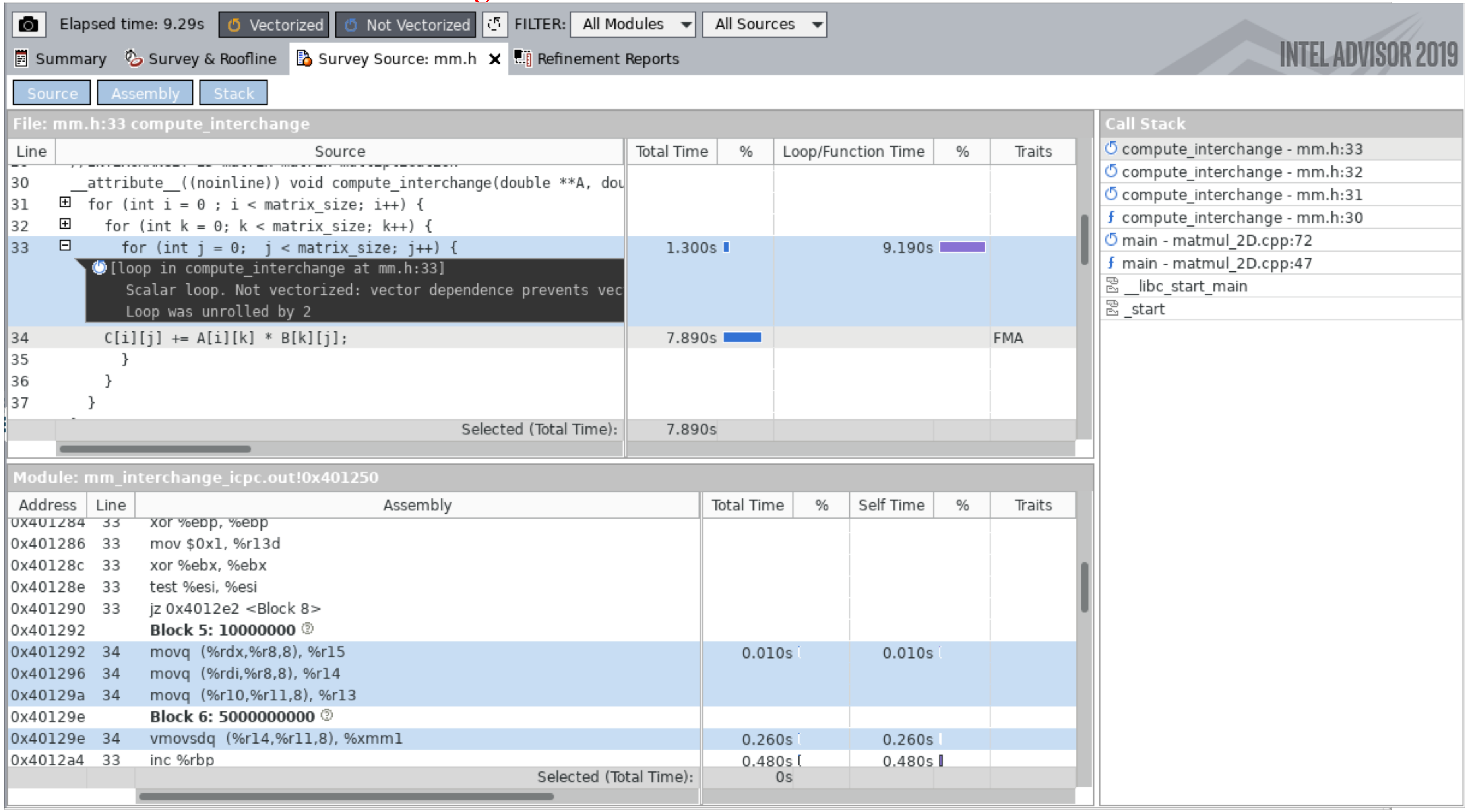
# Workflow of Vectorization Advisor

- **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



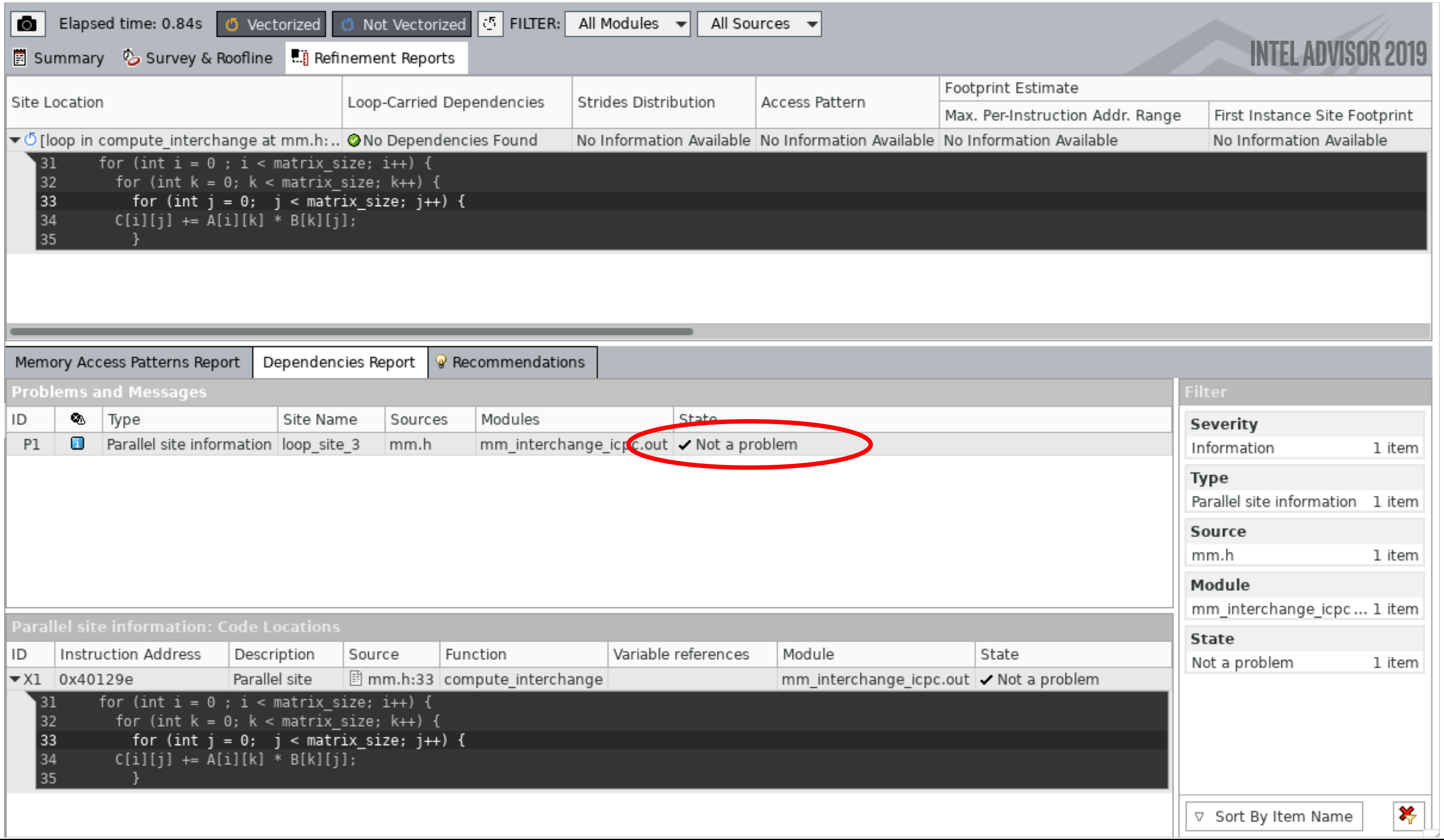
# Survey Analysis

- Compile the code: *icpc -g -O3 –xhost -DINTERCHANGE matmul_2D.cpp -o mm_interchange_icpc.out*
- Collect the survey data: *advixe-cl -c survey -project-dir mm-advisor -- ./mm_interchange_icpc.out 1000*
- Open the result in GUIL: *advixe-gui mm-advisor*



# Dependency Analysis

- Check dependency: *advixe-cl -c dependencies -mark-up-list=3 -project-dir ./mm-advisor -- ./mm_interchange_icpc.out 1000*
- Open the result in GUI: *advixe-gui mm-advisor*



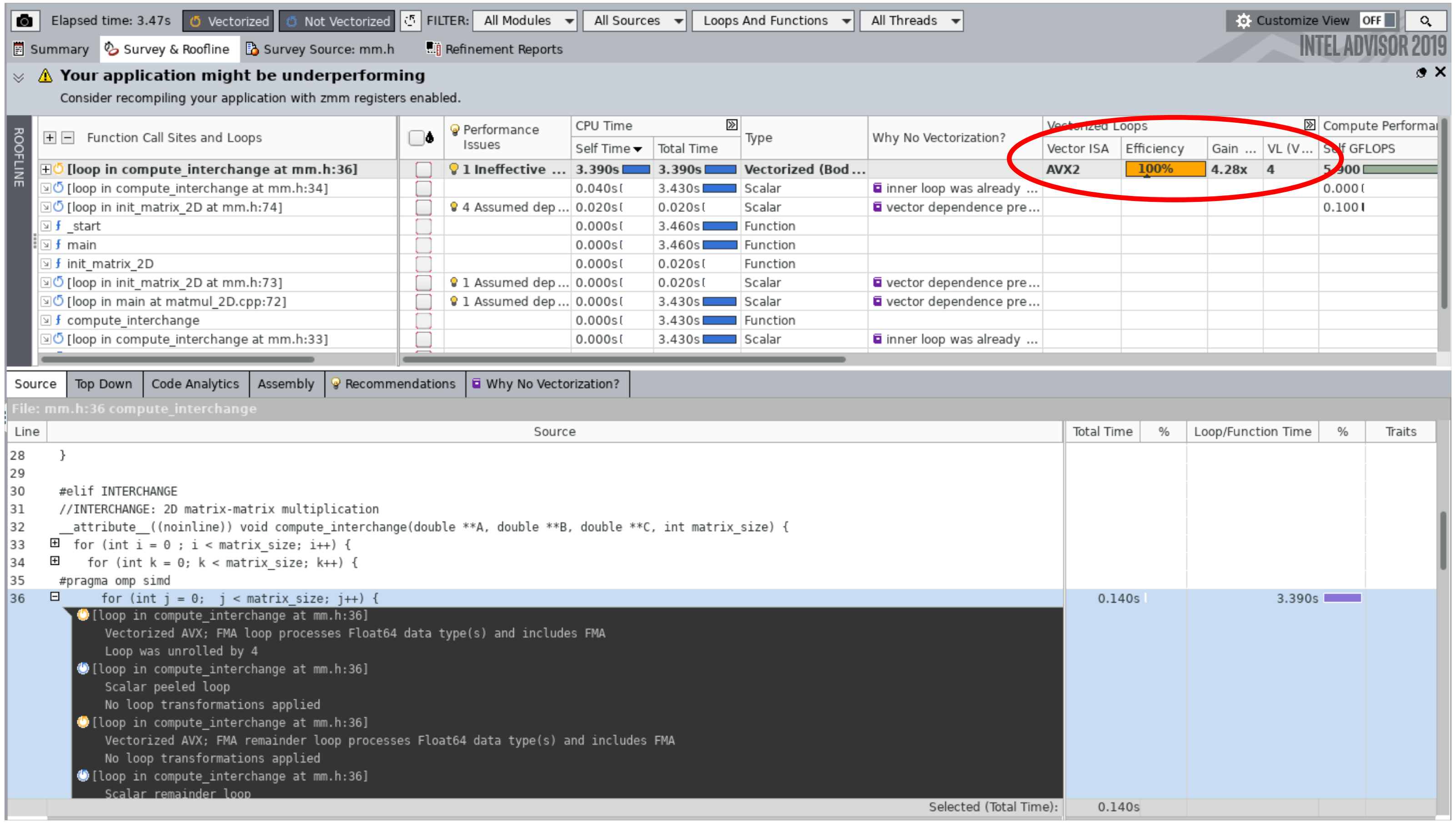
# Resolve Point Aliasing

- We can help the compiler to resolve the dependency complaining caused by point aliasing by:
  - "restrict" keyword and -restrict -std=c90 compiler flag
  - #pragma (GCC) ivdep
  - #pragma omp simd
- We choose OpenMP simd pragma here

```
//INTERCHANGE: 2D matrix-matrix multiplication
__attribute__((noinline)) void compute_interchange(double **A, double **B, double **C, int matrix_size) {
  for (int i = 0 ; i < matrix_size; i++) {
    for (int k = 0; k < matrix_size; k++) {
#pragma omp simd
      for (int j = 0;  j < matrix_size; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

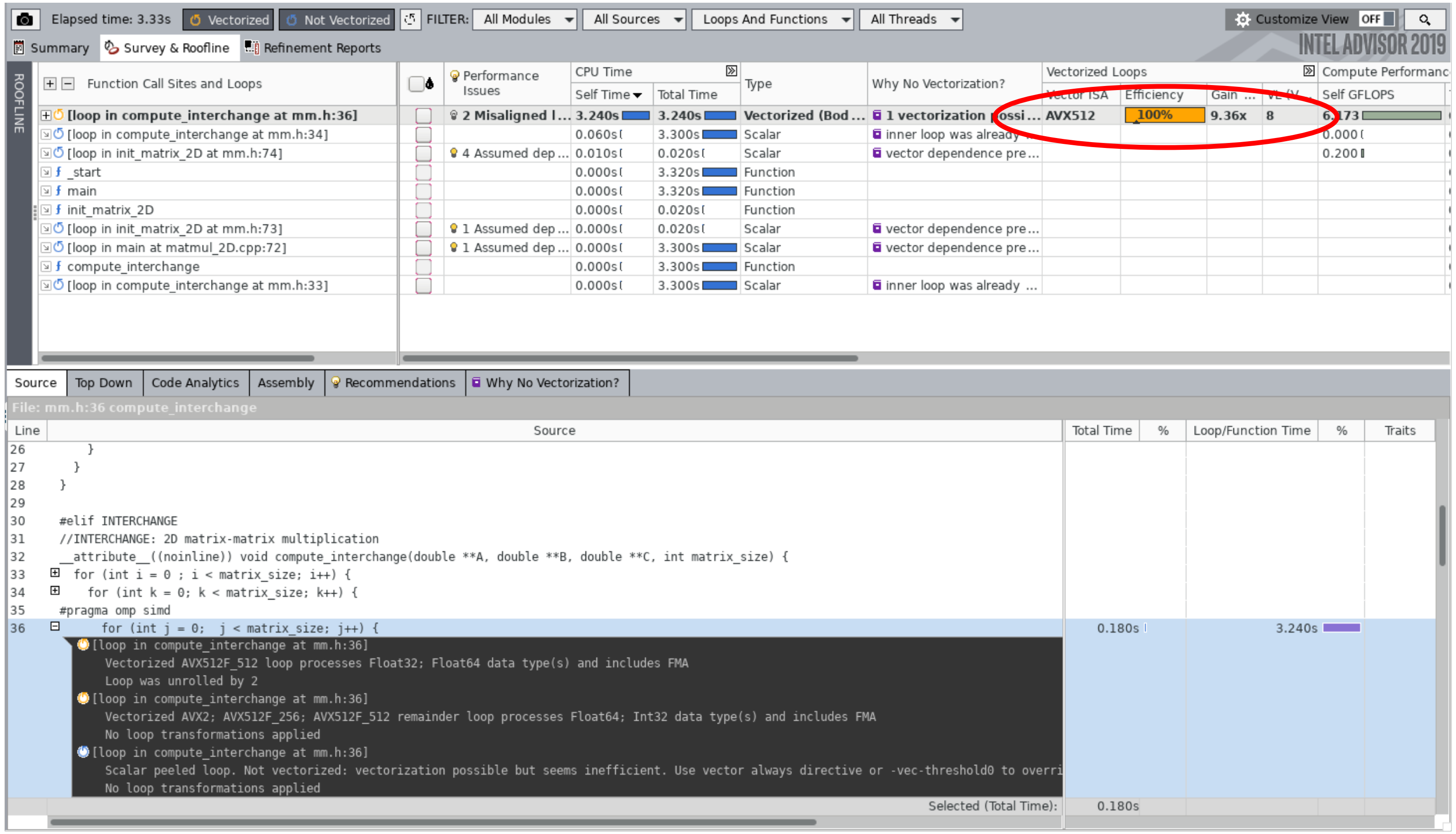
# Re-run Survey Analysis

- Compile the code: *icpc -g -O3 –xhost –qopenmp-simd -DINTERCHANGE matmul_2D.cpp -o mm_interchange_icpc.out*
- Collect the survey data: *advixe-cl -c survey -project-dir mm-advisor -- ./mm_interchange_icpc.out 1000*
- Open the result in GUI: *advixe-gui mm-advisor*



# Using 512-bit ZMM register

- Compile the code: *icpc -g -O3 -xhost -qopenmp-simd -qopt-zmm-usage=high -DINTERCHANGE matmul_2D.cpp -o mm_interchange_icpc.out*
- Collect the survey data: *advixe-cl -c survey -project-dir mm-advisor -- ./mm_interchange_icpc.out 1000*
- Open the result in GUI: *advixe-gui mm-advisor*



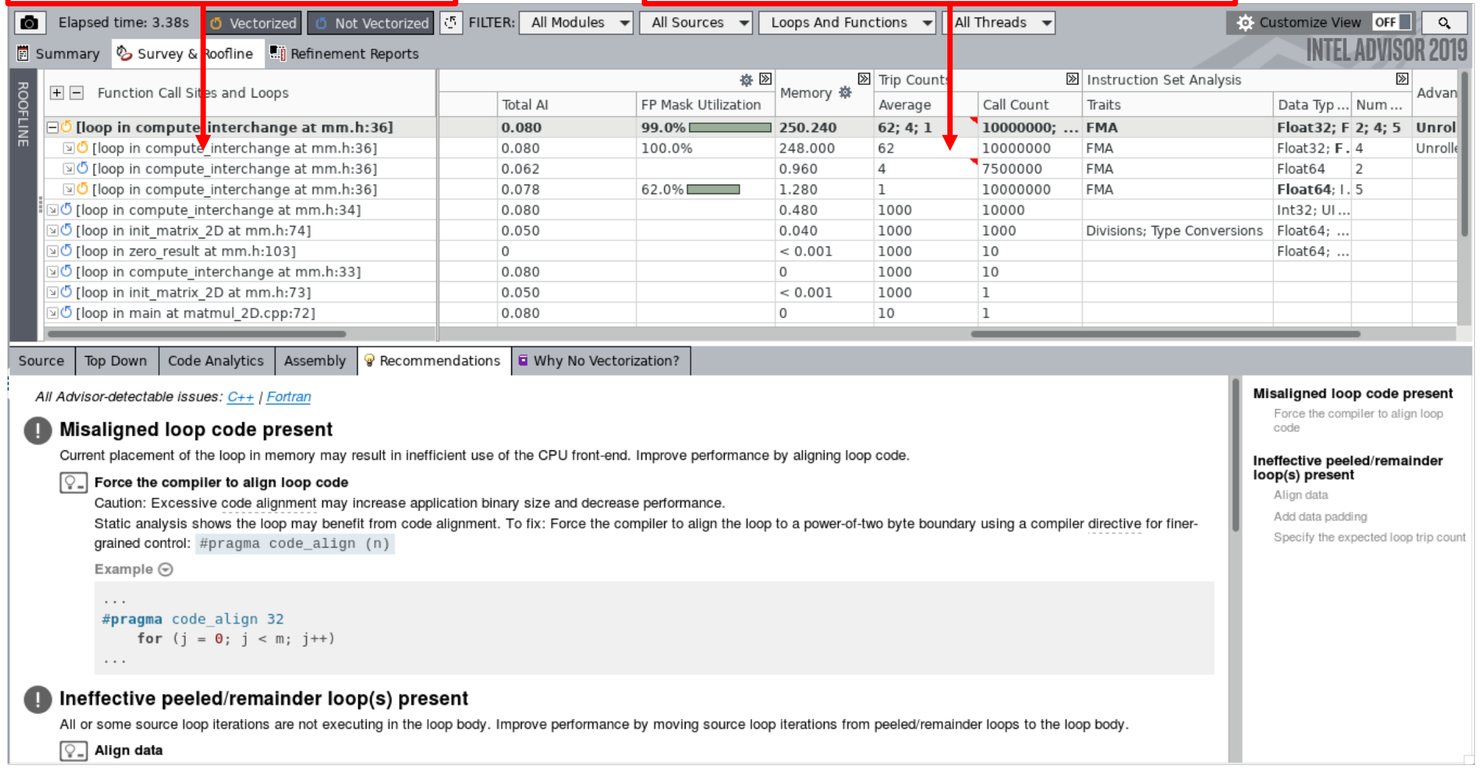
# Trip Counts Analysis

- Collect the trip counts data
  - *advixe-cl -c tripcounts -project-dir mm-advisor -- ./mm_interchange_icpc.out 1000*
  - Note: we need to first carry out "survey" analysis and use the same project directory for "tripcounts"

- **Trip Counts** analysis shows you loop trip counts and call counts
  - The best vectorization requires the scalar trip count to be divisible by the vector length, or you get remainder loops
  - Call counts amplify the importance of tuning a given loop

Loops with peels and remainders can be expanded

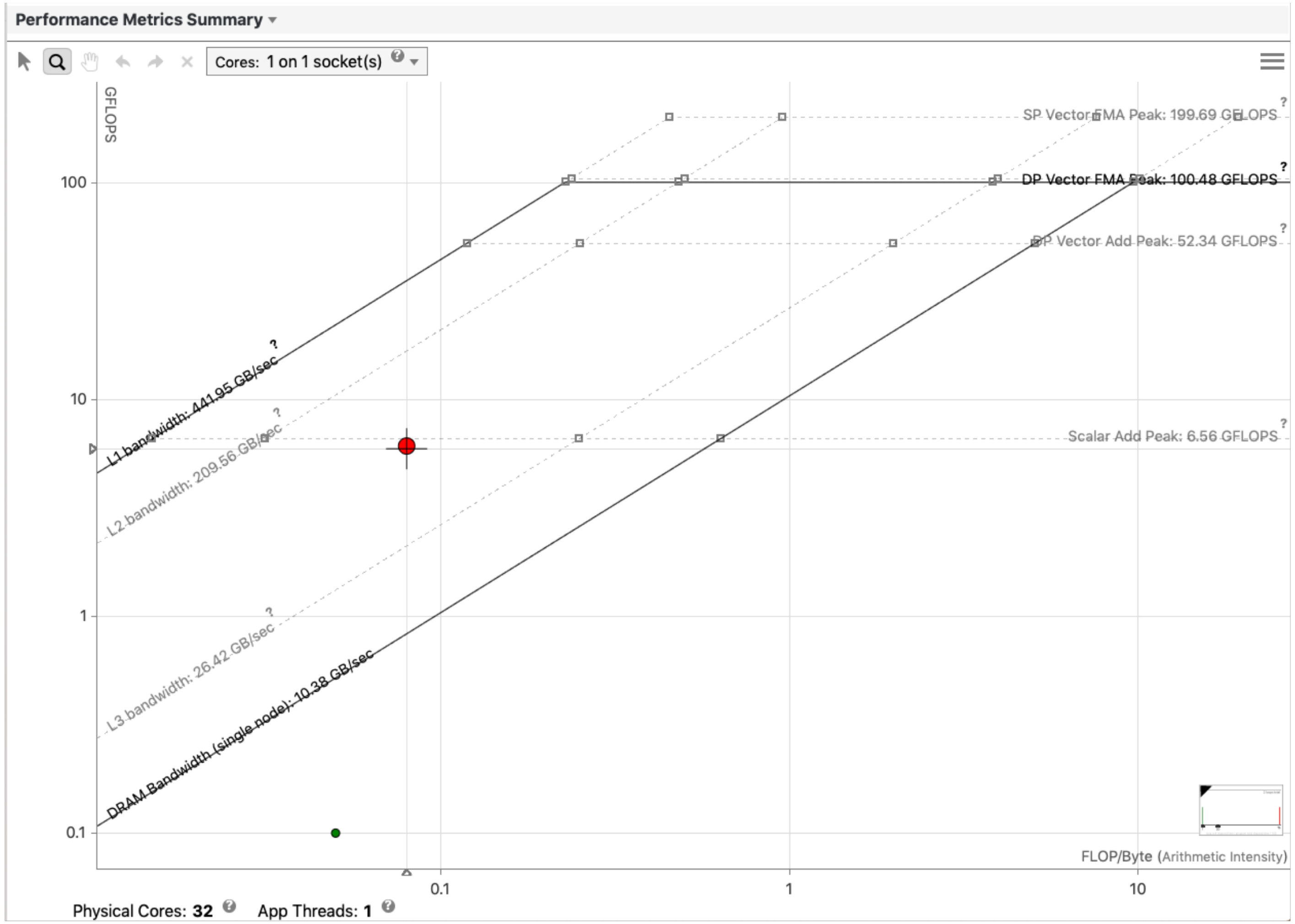
Show number of trip counts for body, peeled and remainders



# Roofline Chart

- Trip counts analysis also collects FLOPS (FLoating-point Operations Per Seconds)
- Collecting FLOPS allows the plotting of a Roofline chart

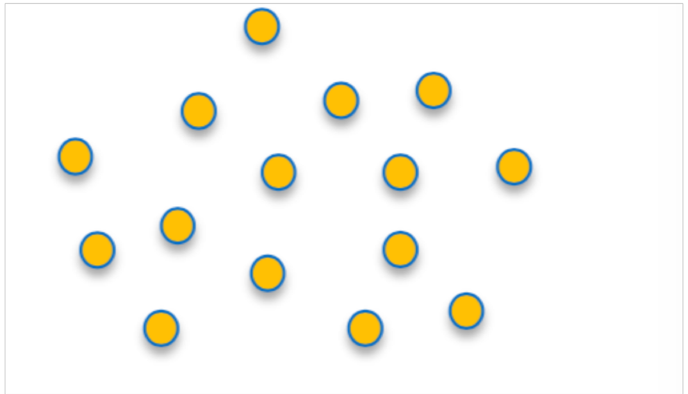
- A visual representation of application performance in relation to hardware limitations, including memory bandwidth and computational peaks
- The horizontal axis is Arithmetic Intensity, a measurement of FLOPs per byte accessed. The vertical axis is performance.
- Provide performance insights
  - Highlights poor performing loops
  - Shows performance "headroom" for each loop
    - Which can be improved
    - Which are worth improving
  - Shows likely causes of bottlenecks
  - Suggest next optimization steps



# N-Body Problem

```
struct Particle {
  float x, y, z;
  float vx, vy, vz;
};
```

```
for (int i = 0; i < nParticles; i++) {
  // Components of the gravity force on particle i
  float Fx = 0, Fy = 0, Fz = 0;

  const float xi = particle[i].x;
  const float yi = particle[i].y;
  const float zi = particle[i].z;

  for (int j = 0; j < nParticles; j++) {
    // Newton's law of universal gravity
    const float dx = particle[j].x - xi;
    const float dy = particle[j].y - yi;
    const float dz = particle[j].z - zi;

    const float drPower32  = pow(drSquared, 3.0/2.0);

    const float drPower32Inv = 1.0f / drPower32;
    // Calculate the net force
    Fx += dx * G * drPower32Inv;
    Fy += dy * G * drPower32Inv;
    Fz += dz * G * drPower32Inv;
  }

  // Accelerate particles in response to the gravitational force
  particle[i].vx += dt*Fx;
  particle[i].vy += dt*Fy;
  particle[i].vz += dt*Fz;
}
```



$$\vec{F}_{ij} = \frac{G m_i m_j}{\left|\vec{r}_j - \vec{r}_i\right|^3}\left(\vec{r}_j - \vec{r}_i\right)$$

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = m\frac{d^2\vec{x}}{dt^2}$$

The example code assumes m=1 for all particles

# Hands-on: Explore Survey Analysis

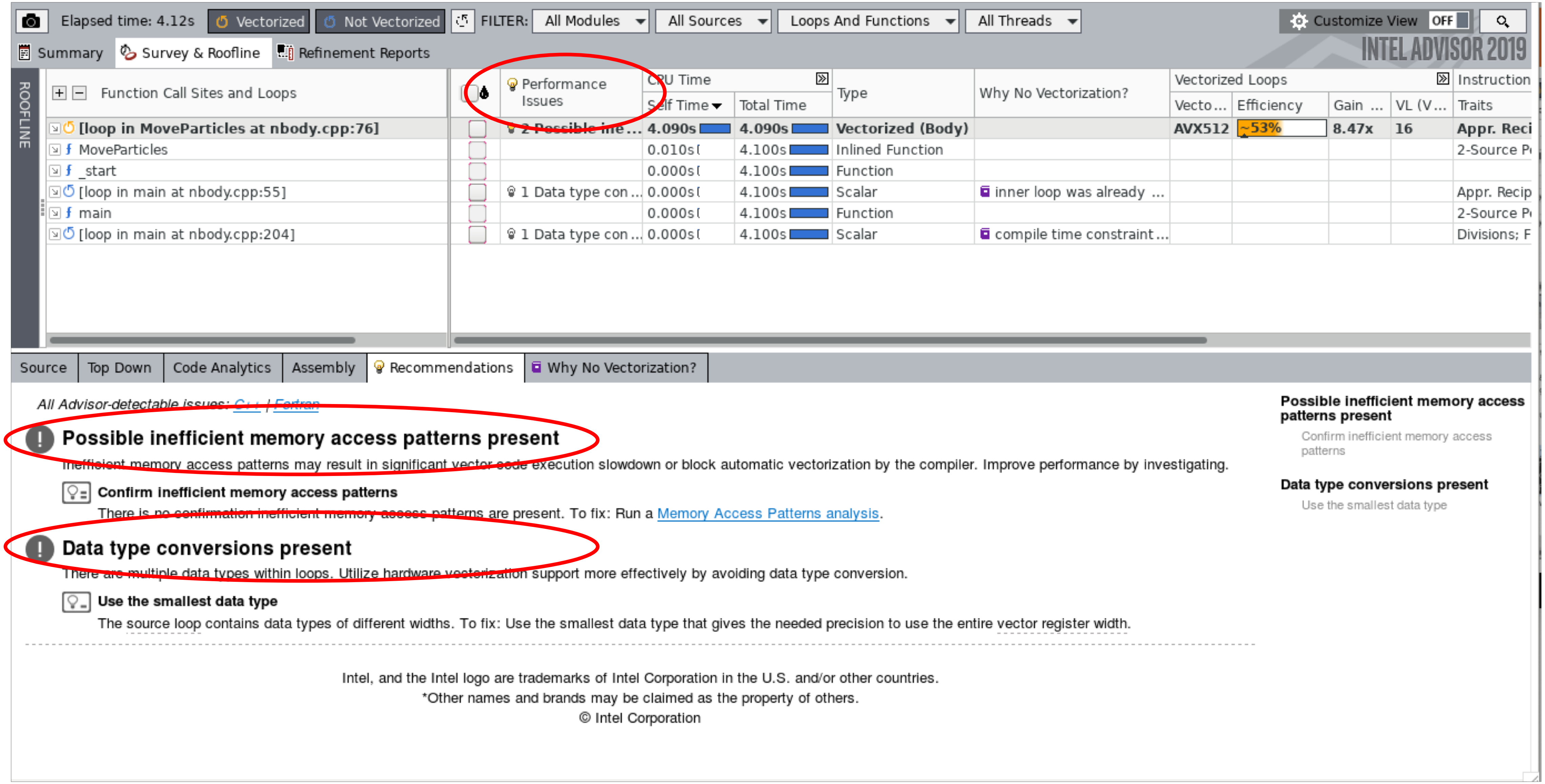
Windows 1

- Log into Adroit
  - *ssh –l <user> adroit.princeton.edu*
- Load environment modules
  - *module load intel*
- Compile the code
  - *icpc -g -O2 -xhost -qopt-zmm-usage=high -qopenmp nbody.cpp -o nbody.out*
- Run the provided script to submit a Advisor wrapped job to the scheduler
  - *./submit_to_scheduler*

Windows 2

- Log into Adroit with X11 forwarding
  - *ssh –Y -C <user>@adroit.princeton.edu*
  - Will need local xserver (XQuartz for OSX, Xming for Windows)
- Load environment modules
  - *module load intel intel-advisor*
- Open the resulting directory with Intel Advisor
  - *advixe-gui nbody-advisor*
  - Click "Show My Result"
- Explore "Survey" report

# Any Performance Issue?



# Revisit N-Body Code

```
struct Particle {
  float x, y, z;
  float vx, vy, vz;
};
for (int i = 0; i < nParticles; i++) {
  // Components of the gravity force on particle i
  float Fx = 0, Fy = 0, Fz = 0;

  const float xi = particle[i].x;
  const float yi = particle[i].y;
  const float zi = particle[i].z;

  for (int j = 0; j < nParticles; j++) {
    // Newton's law of universal gravity
    const float dx = particle[j].x - xi;
    const float dy = particle[j].y - yi;
    const float dz = particle[j].z - zi;

    const float drPower32  = pow(drSquared, 3.0/2.0);

    const float drPower32Inv = 1.0f / drPower32;
    // Calculate the net force
    Fx += dx * G * drPower32Inv;
    Fy += dy * G * drPower32Inv;
    Fz += dz * G * drPower32Inv;
  }

  // Accelerate particles in response to the gravitational force
  particle[i].vx += dt*Fx;
  particle[i].vy += dt*Fy;
  particle[i].vz += dt*Fz;
}
```

```
struct ParticleArrays {
  float *x, *y, *z;
  float *vx, *vy, *vz;
};
```

```
const float drPower32  = powf(drSquared, 3.0f/2.0f);
```

# Re-run Survey Analysis

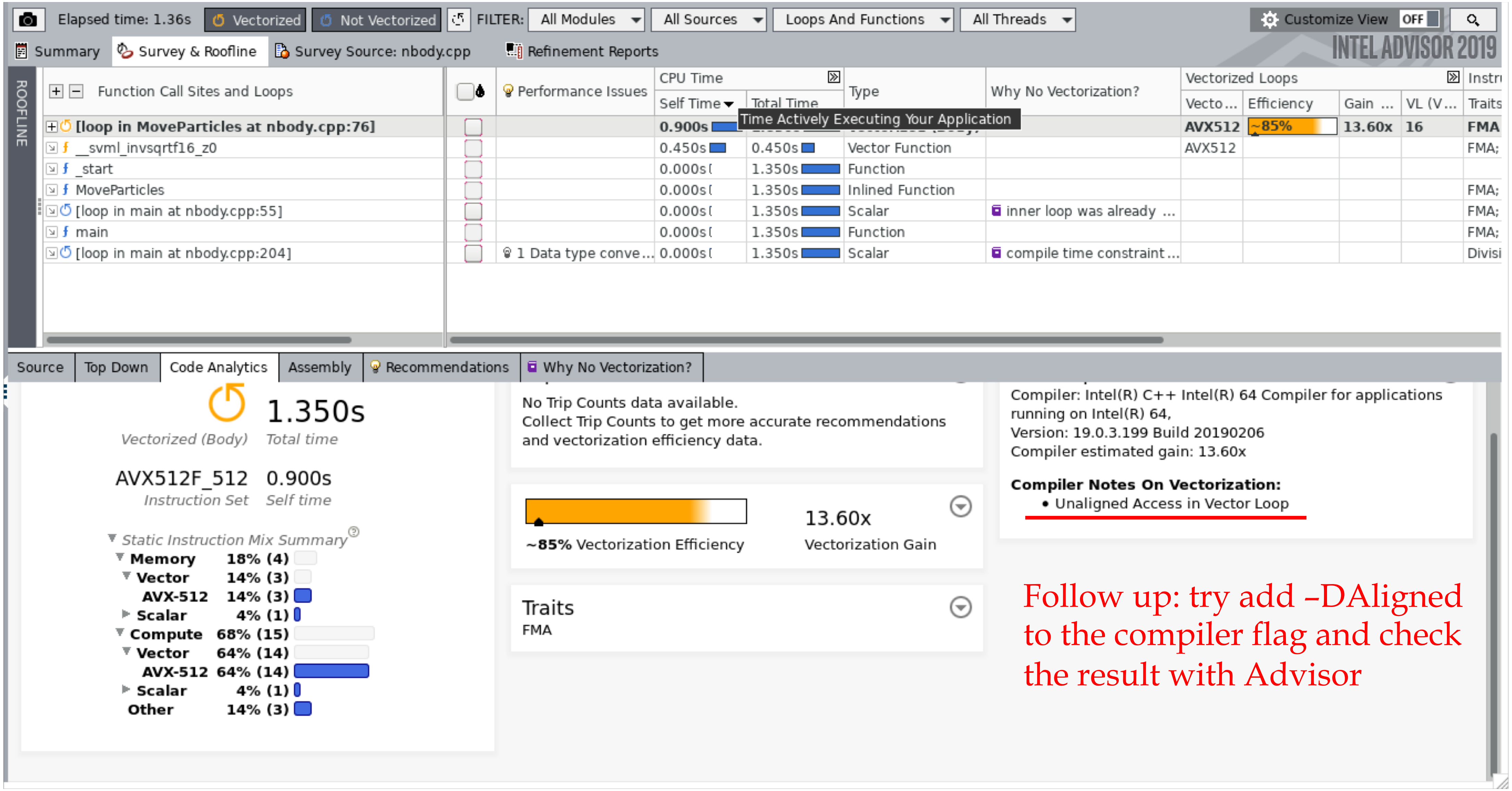
Windows 1

- Compile the code
  - *icpc -g -O2 -xhost -qopt-zmm-usage=high -qopenmp –DSoA -DNo_FP_Conv nbody.cpp -o nbody.out*
- Re-run the provided script to submit a Advisor wrapped job to the scheduler
  - *./submit_to_scheduler*

Windows 2

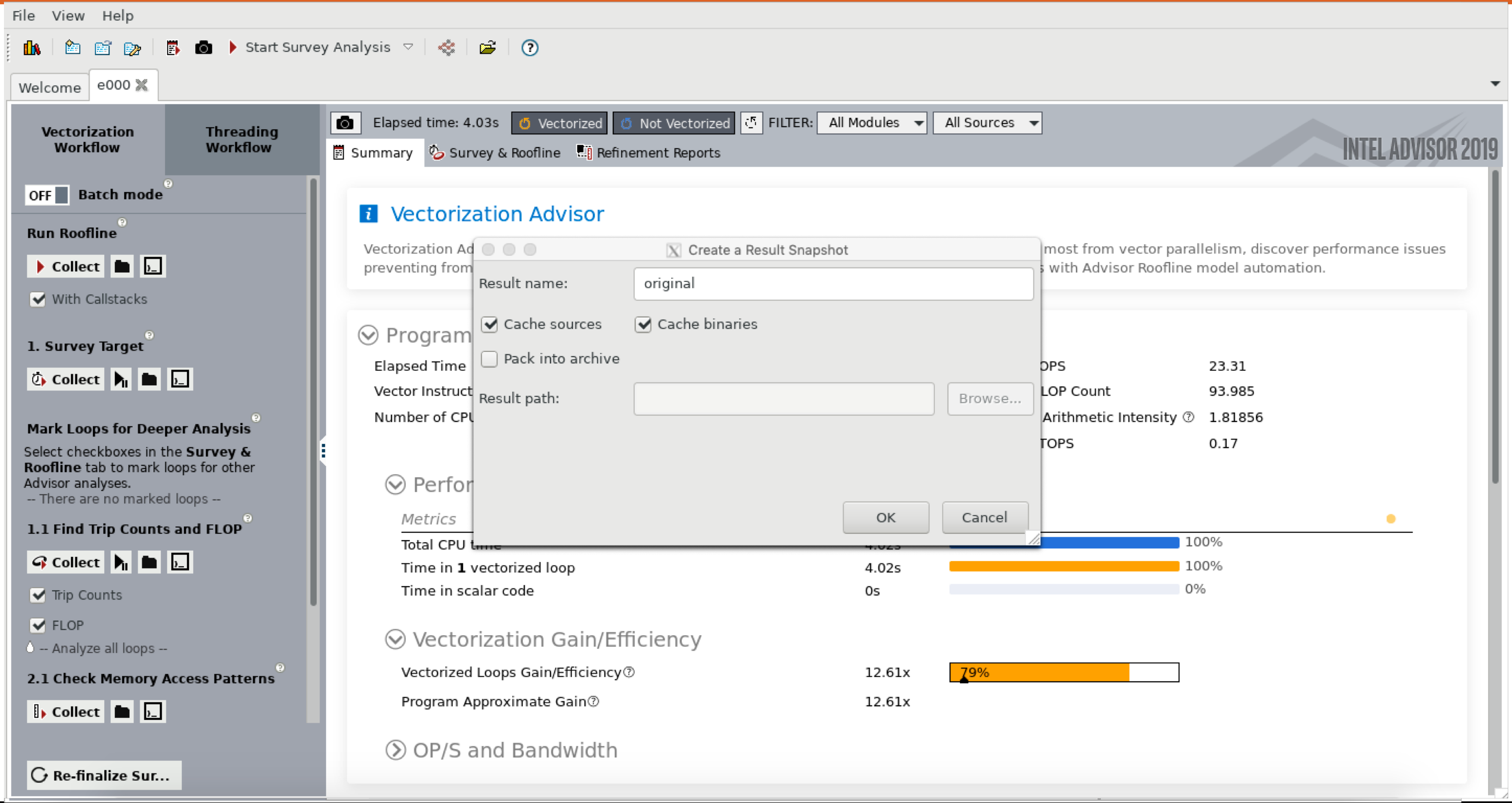
- Re-open the resulting directory with Intel Advisor
  - *advixe-gui nbody-advisor*
  - Click "Show My Result"
- Explore "Survey" report

# Any Remaining Performance Issue?



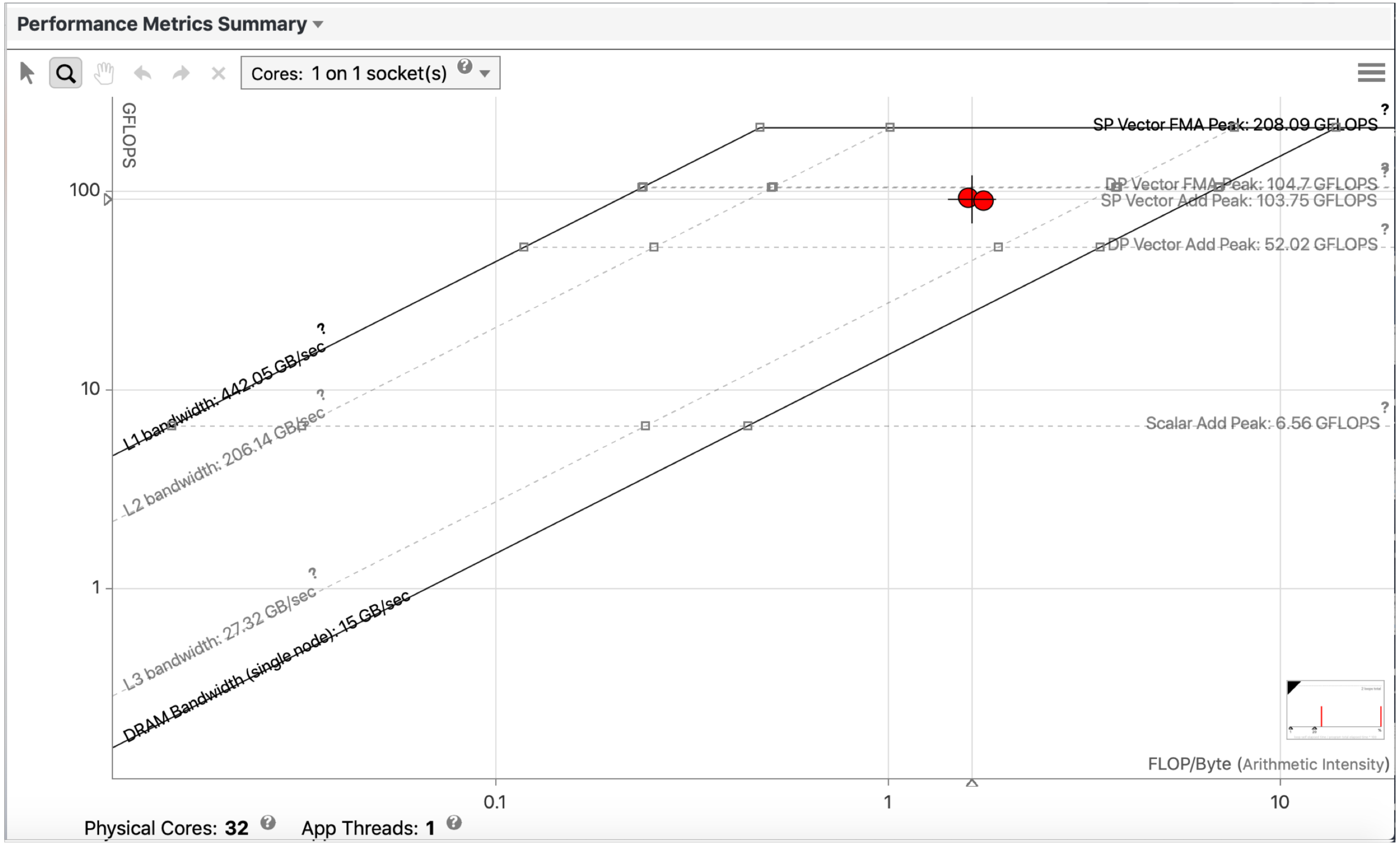

Follow up: try add –DAligned to the compiler flag and check the result with Advisor

# Create Snapshot for Comparison



# Roofline Comparison

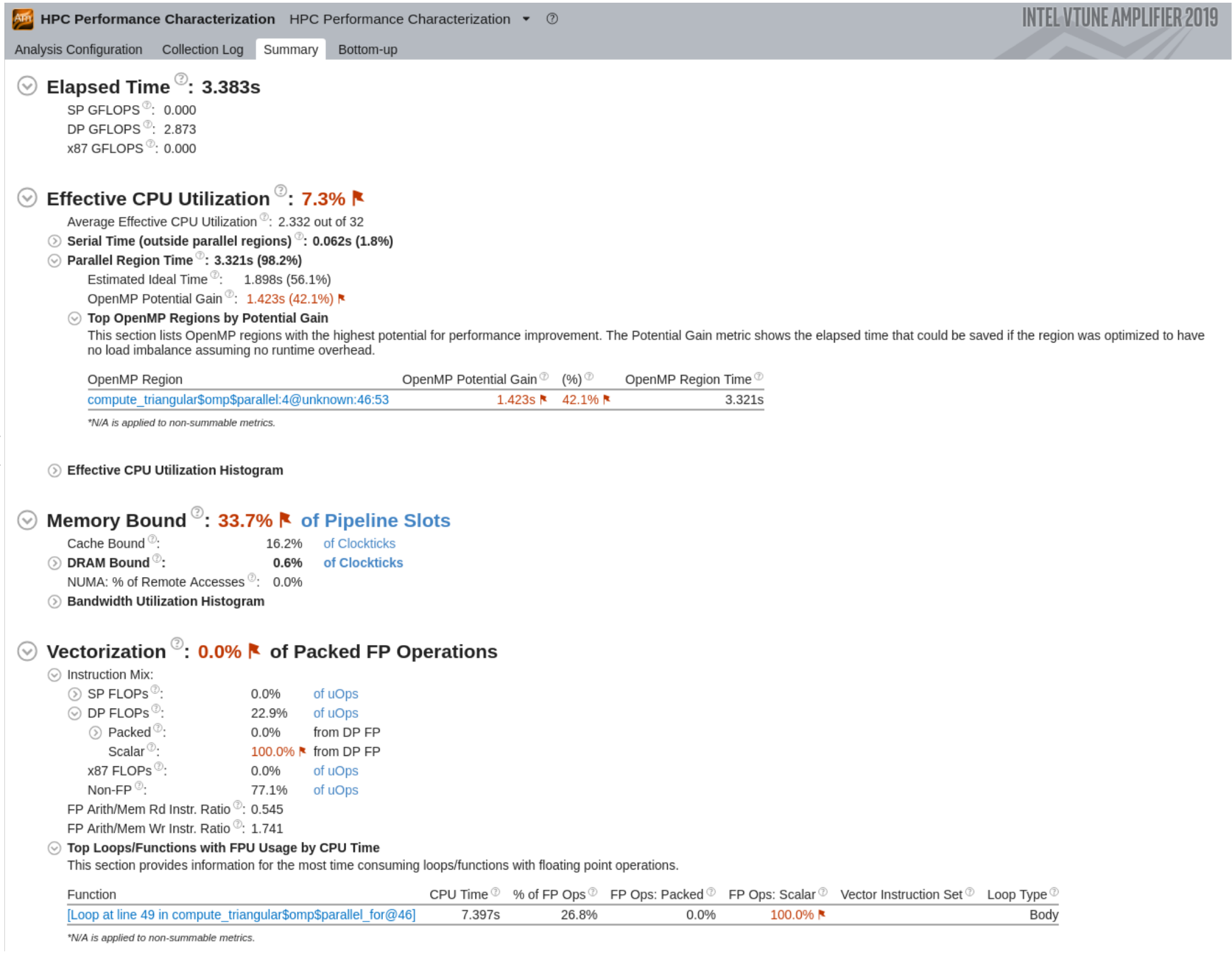
- Can you make a roofline chart for the original code and the optimized one?



# Intel VTune

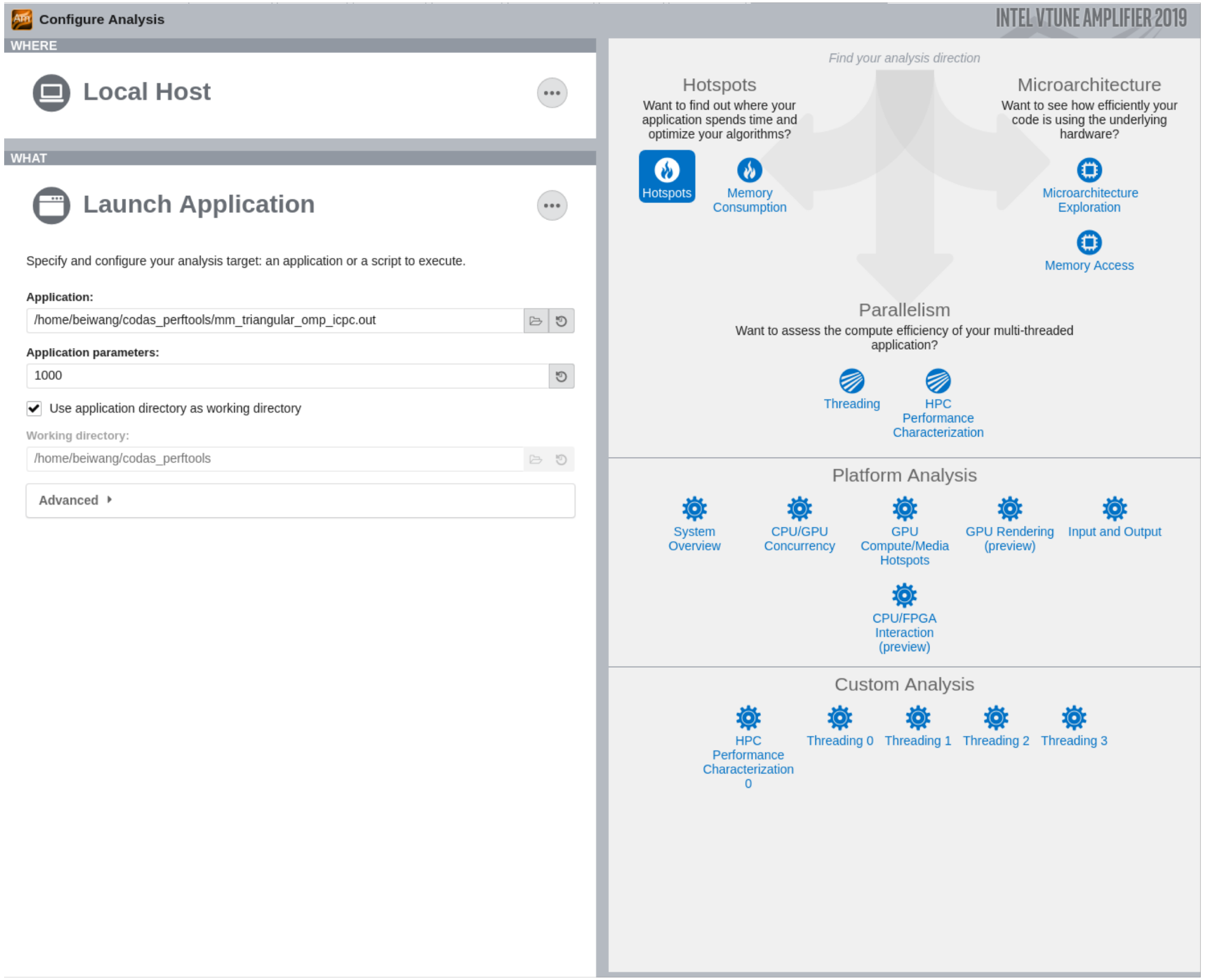
# Intel VTune Amplifier

- Accurate data
  - Hotspot
  - Processor microarchitecture
  - Memory access
  - Threading
  - I/O
- Flexible
  - Linux, Windows and Mac OS analysis GUI
  - Link data to source code and assembly
  - Easy set-up, no special compiles
- **Shared memory only**
  - Serial
  - OpenMP
  - MPI on a single node

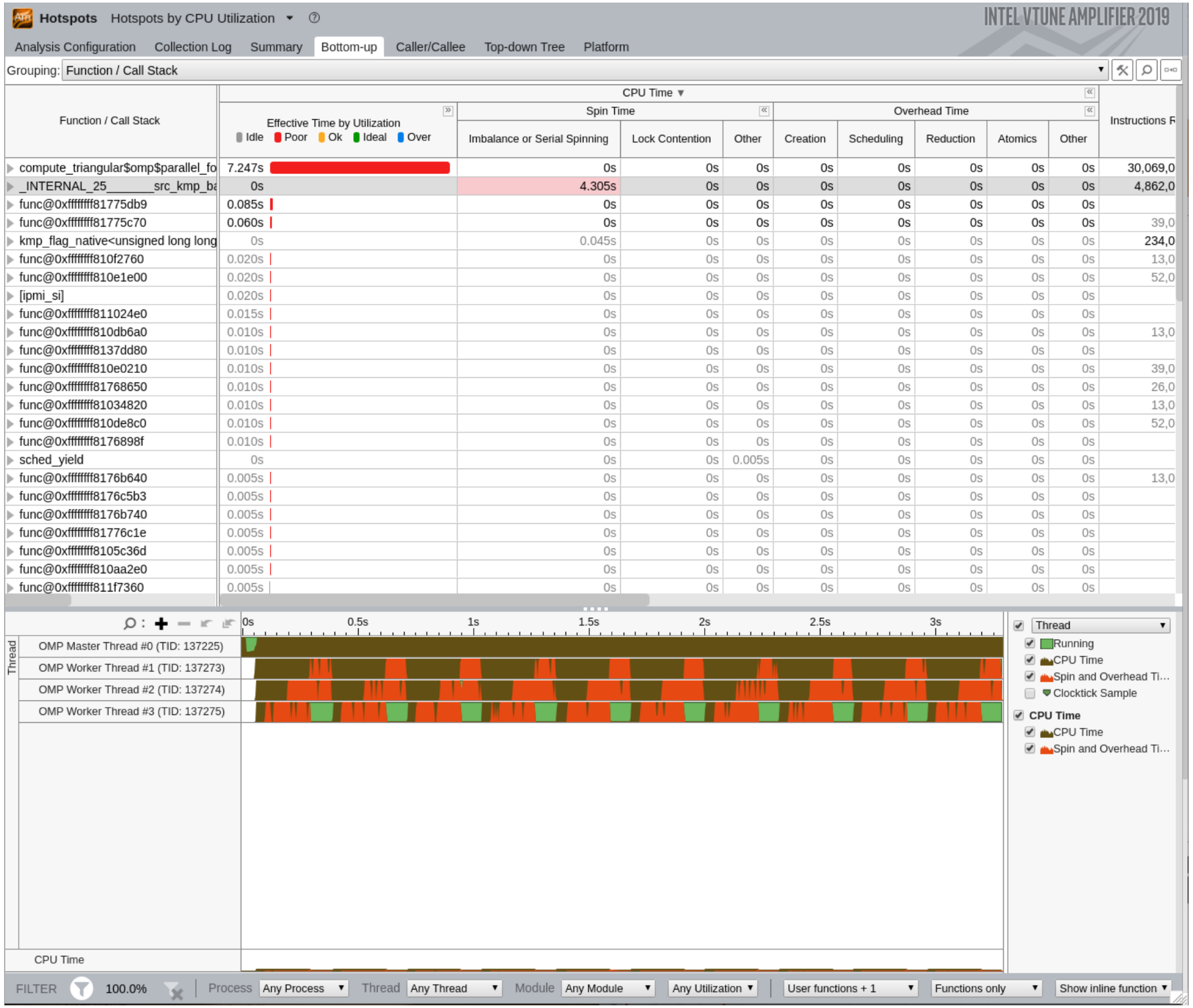


# A Rich Set of Predefined Analysis Types

- **Hotspots**: what functions use most time?
- **Microarchitecture Exploration**: hardware-level performance data
- **Memory Access**: identify memory-related issues
- **HPC Performance Characterization**: overview of CPU, memory and FPU utilization
- **Threading**: Identify potential parallelization opportunities/issues

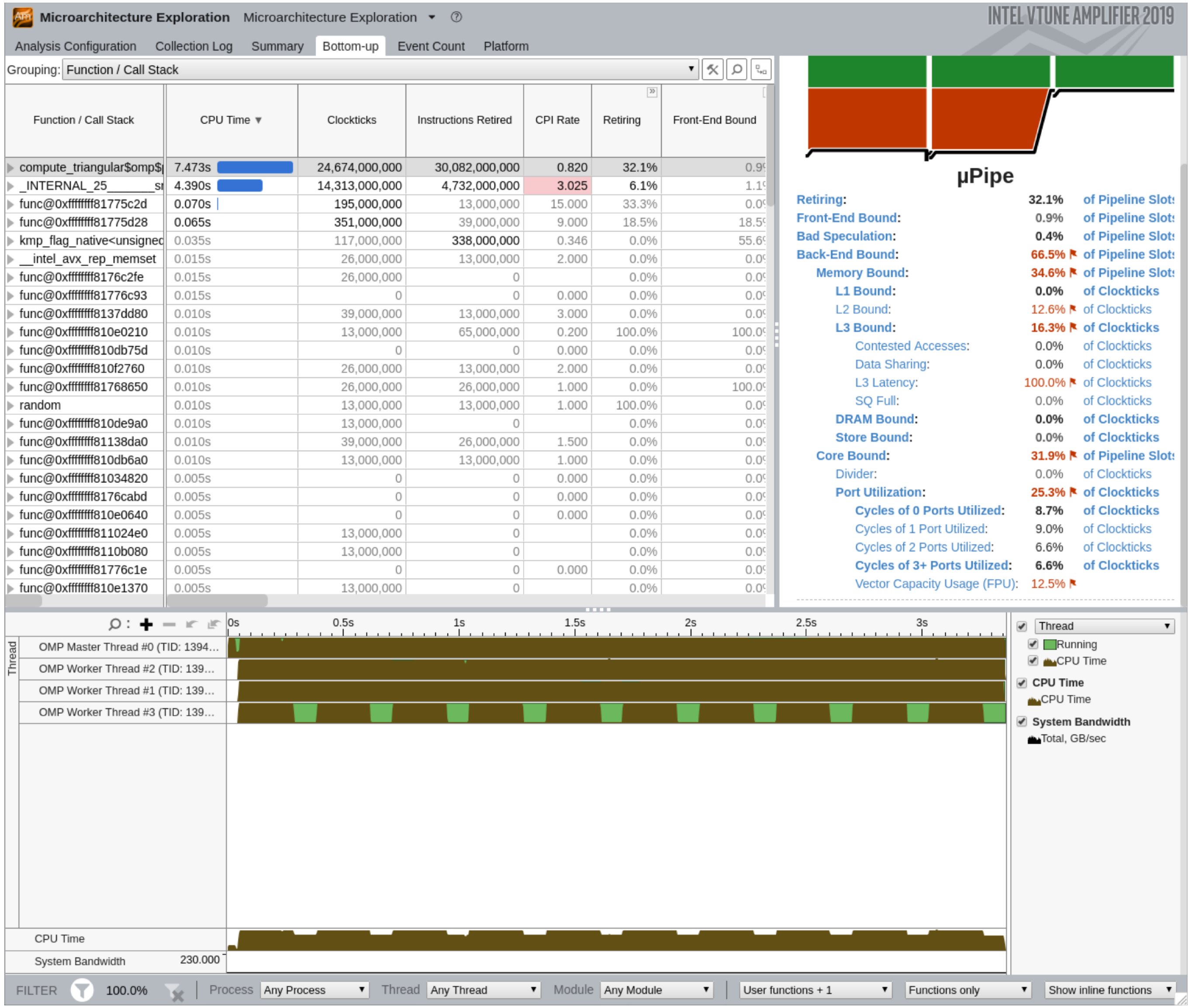


# Hotspots




| Function / Call Stack | CPU Time | | | | | | | | | Instructions R |
|---|---|---|---|---|---|---|---|---|---|---|
| | Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over) | Spin Time | | | Overhead Time | | | | | |
| | | Imbalance or Serial Spinning | Lock Contention | Other | Creation | Scheduling | Reduction | Atomics | Other | |
| 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 | |

# Microarchitecture Exploration



# Threading

**Threading** Threading Efficiency  INTEL VTUNE AMPLIFIER 2019

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

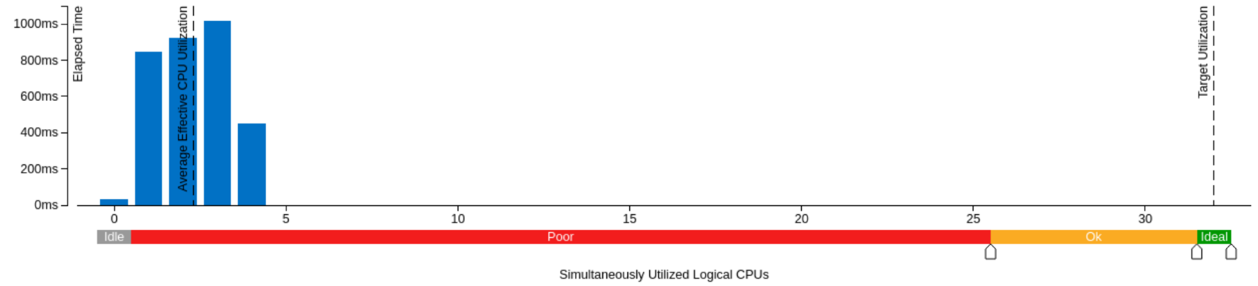
## Elapsed Time: 3.270s

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_triangular$omp$parallel: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.*

# Suggest Next Steps

- 1. L2 and L3 cache issue: try blocking technique
- 2. Thread load imbalance: try "#pragma omp parallel for schedule(dynamic)" for the outer most loop
- 3. Vectorization: try "#pragma omp simd" for the inner most loop

# References


- “Introduction to Performance Tuning & Optimization Tools”, CoDaS-HEP 2018, Ian Cosden, https://github.com/cosden/CoDaS-HEP-Perf-Tuning
- “Compiling and Tuning for Performance using Intel Advanced Vector Extensions 512”, SC18, Intel Speakership Tutorial, Carlos Rosales-Fernandez
- ”How to Analyze the Performance of Parallel Codes 101”, SC18 Tutorial, https://openspeedshop.org/2018/11/sc18-how-to-analyze-the-performance-of-parallel-codes-101/
- “Vector Parallelism on Multi-Core Processors”, CoDas-HEP 2019, Steve Lantz
- Perf: https://perf.wiki.kernel.org/index.php/Tutorial, http://www.brendangregg.com/perf.html