

# 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



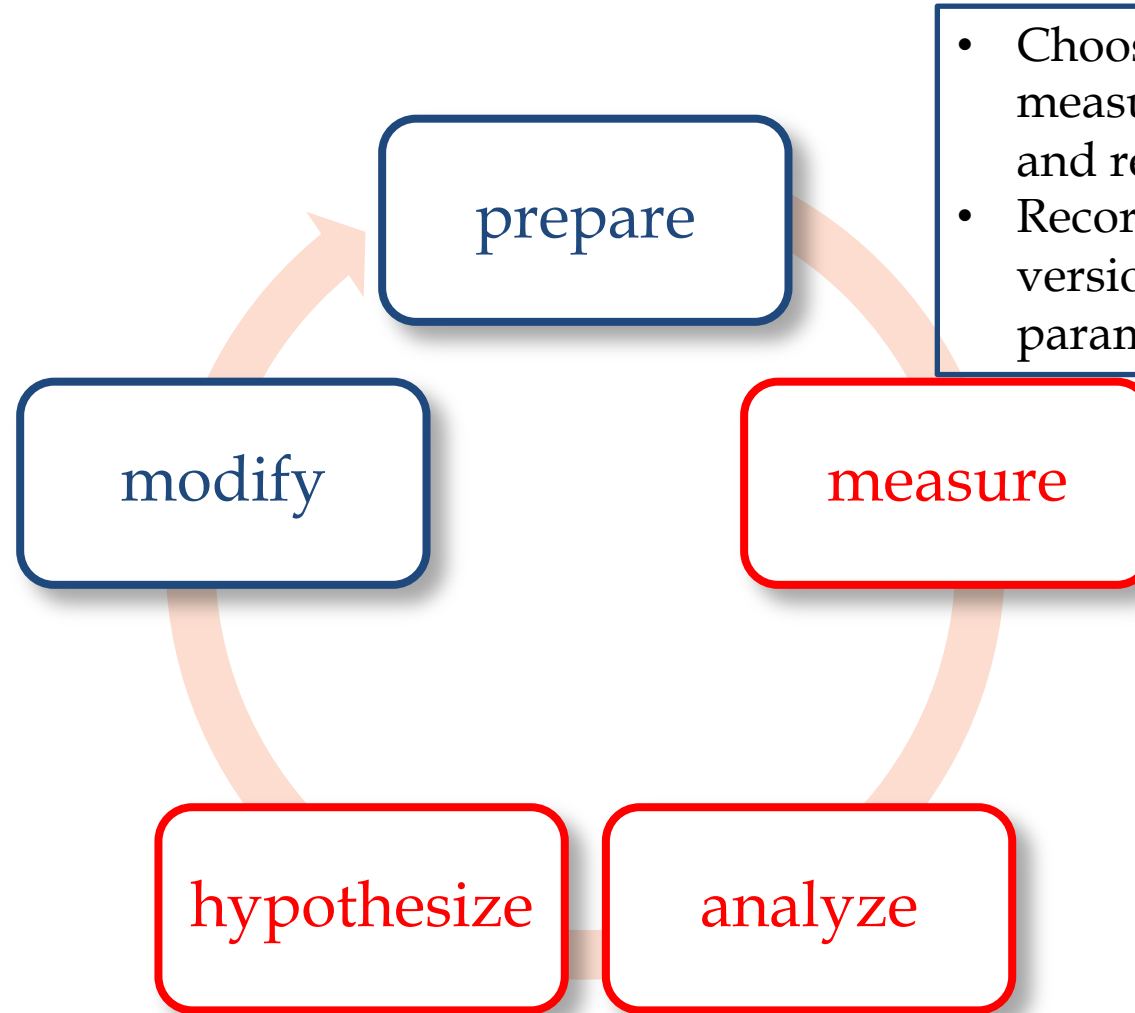
# **BASIC CONCEPTS IN PERFORMANCE TUNING**

# 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 satisfied
- 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

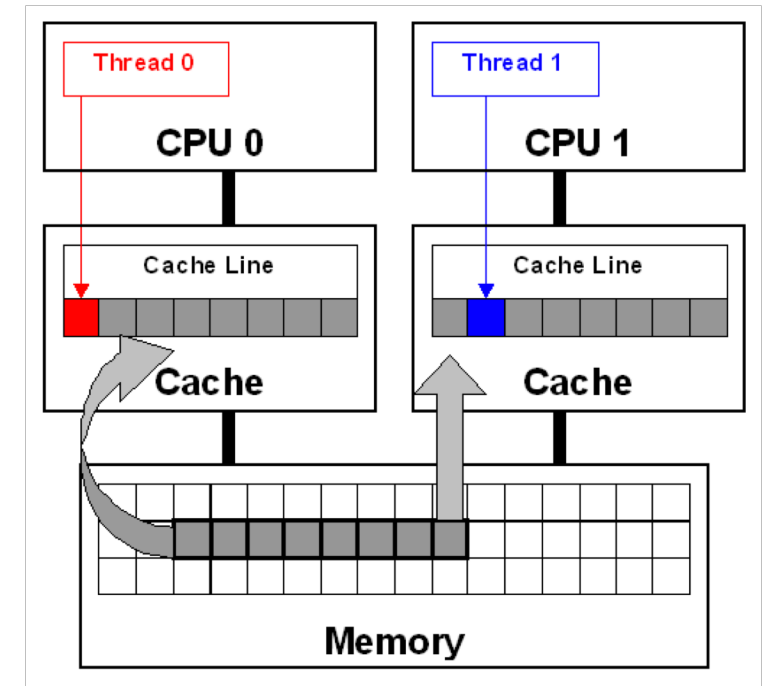
```
float x=3.14; //bad: 3.14 is a double
float s=sin(x); //bad: sin() is a double
precision function
long v=round(x); //bad: round takes
and returns double
```

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

- 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 parameters
    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_iters; 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/beiwan2003/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 -DNNAIVE 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]     | [.] 0x000000000000c2698   |
| 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        |

```

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
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
mov (%r12),%rdx
55.56 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
4014e8 <init_matrix_2D(double**, double**, 48
↑ 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 lower for interchange, reflecting its shorter elapsed time
- The number of instructions are half in interchange
- Interchange has substantial fewer L1 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**

- 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*

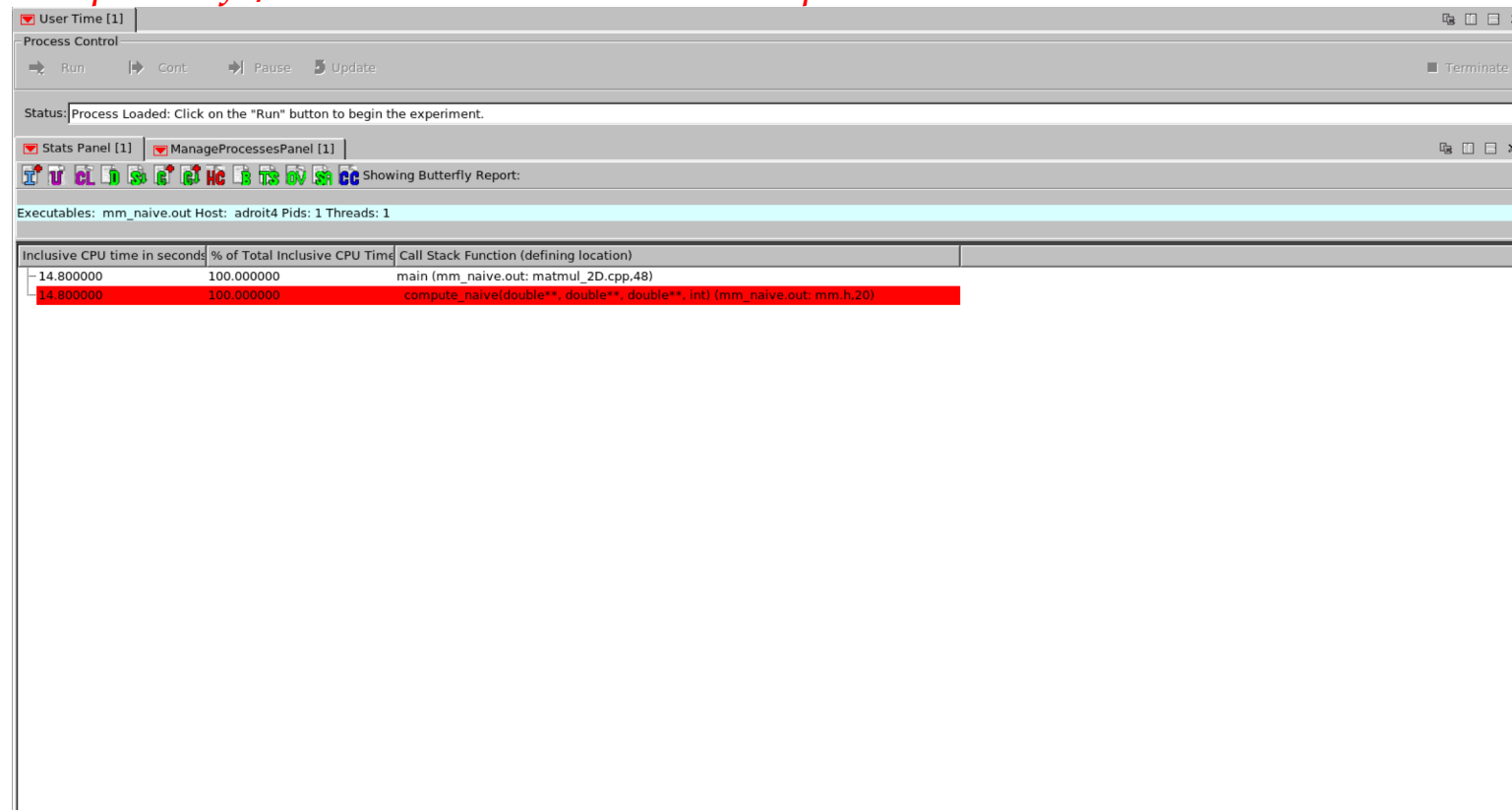
The screenshot shows the openss GUI with the following components:

- Process Control:** Includes buttons for Run, Cont, Pause, Update, and Terminate. The status bar indicates "Process Loaded: Click on the 'Run' button to begin the experiment."
- Stats Panel [1]:** Displays "Exclusive CPU time" with a value of 6.030000 and a range of >> 8.950000.
- Source Panel [1]:** Shows the source code for `/home/beiwan/codas_perftools/mm.h`. The code includes headers for `<iostream>`, `<stdlib.h>`, `<omp.h>`, and `<sys/time.h>`. It defines two functions: `compute_naive` and `compute_interchange`, both performing 2D matrix-matrix multiplication. The `compute_naive` function is highlighted in yellow, and the `compute_interchange` function is highlighted in red.

```
7 Sample Matrix-Matrix multiplication code for the CoDaS-HEP Summer School 2019
8 (This sample code is based on the CoDaS-HEP Summer School 2018 example by Ian A. Cosden.
9 Please see https://github.com/cosden/CoDaS-HEP-Perf-Tuning for details.)
10 /*****
11
12 #include <iostream>
13 #include <stdlib.h>
14 #include <omp.h>
15 #include <sys/time.h>
16 using namespace std;
17
18 //NAIVE: 2D matrix-matrix multiplication
19 __attribute__((noinline)) void compute_naive(double **A, double **B, double **C, int matrix_size) {
20     for (int i = 0; i < matrix_size; i++) {
21         for (int j = 0; j < matrix_size; j++) {
22             for (int k = 0; k < matrix_size; k++) {
23                 C[i][j] += A[i][k] * B[k][j];
24             }
25         }
26     }
27 }
28
29 //INTERCHANGE: 2D matrix-matrix multiplication
30 __attribute__((noinline)) void compute_interchange(double **A, double **B, double **C, int matrix_size) {
31     for (int i = 0; i < matrix_size; i++) {
32         for (int k = 0; k < matrix_size; k++) {
33             for (int j = 0; j < matrix_size; j++) {
34                 C[i][j] += A[i][k] * B[k][j];
35             }
36         }
37     }
38 }
39
40 //TRIANGULAR: only compute the result for the upper triangular
41 __attribute__((noinline)) void compute_triangular(double **A, double **B, double **C, int matrix_size) {
42     #pragma omp parallel for
```

# 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 "/code>./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<br>CPU time<br>in<br>seconds. | % of CPU<br>Time | papi_tot_cyc | papi_tot_ins | papi_l1_dcm | Comp.<br>Intensity | papi_tot_cyc% | Function (defining location)   |
|---|------------------|--------------|--------------|-------------|--------------------|---------------|--|
| 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

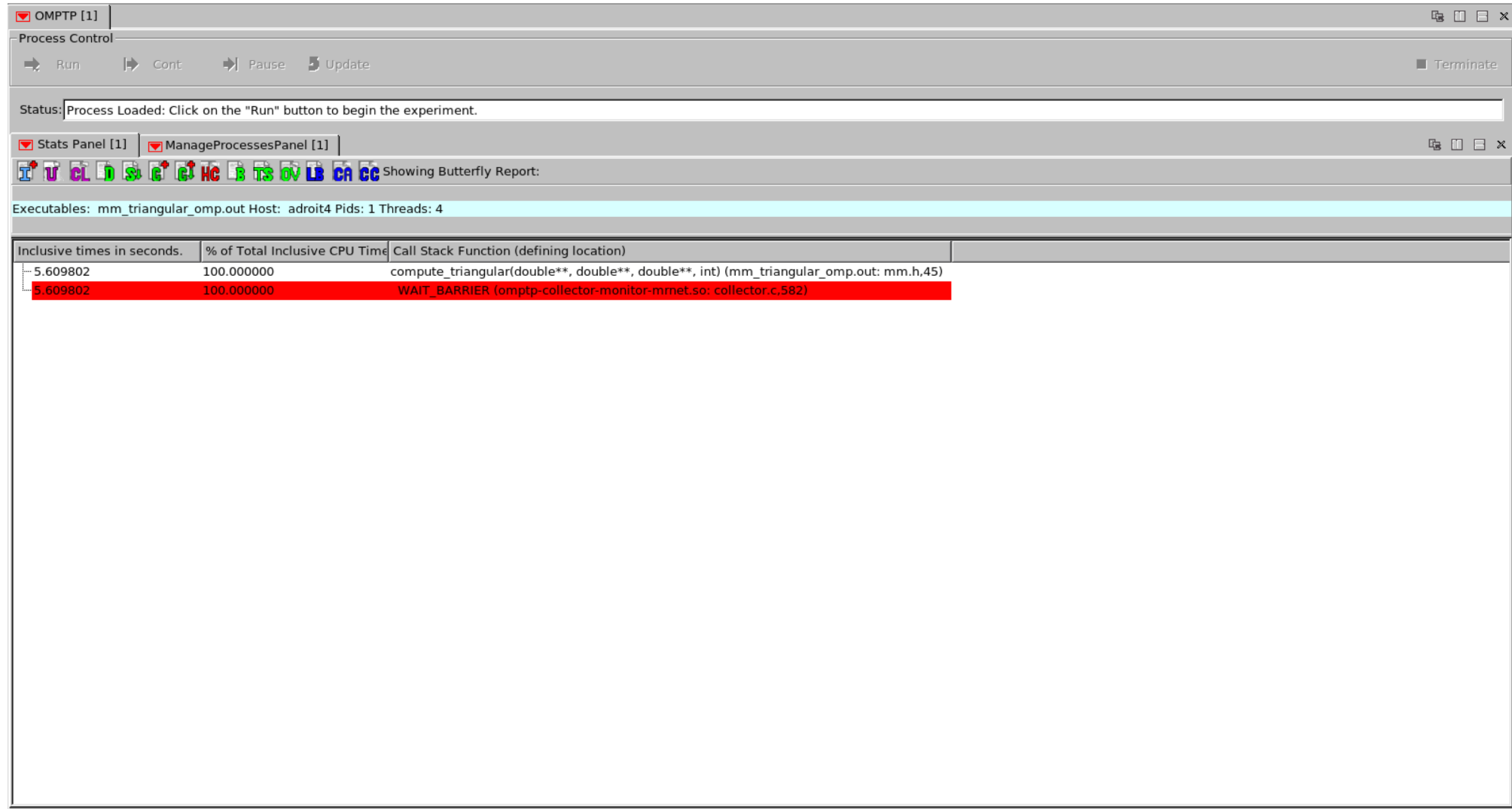
- For parallel execution, is there any load imbalance issue? How do you find the potential cause?
- OMPT API: record task time, idleness, barrier, wait barrier per OpenMP parallel region
- Let's look at the matrix-matrix example, but now we only compute the result for the upper triangular

```
//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*



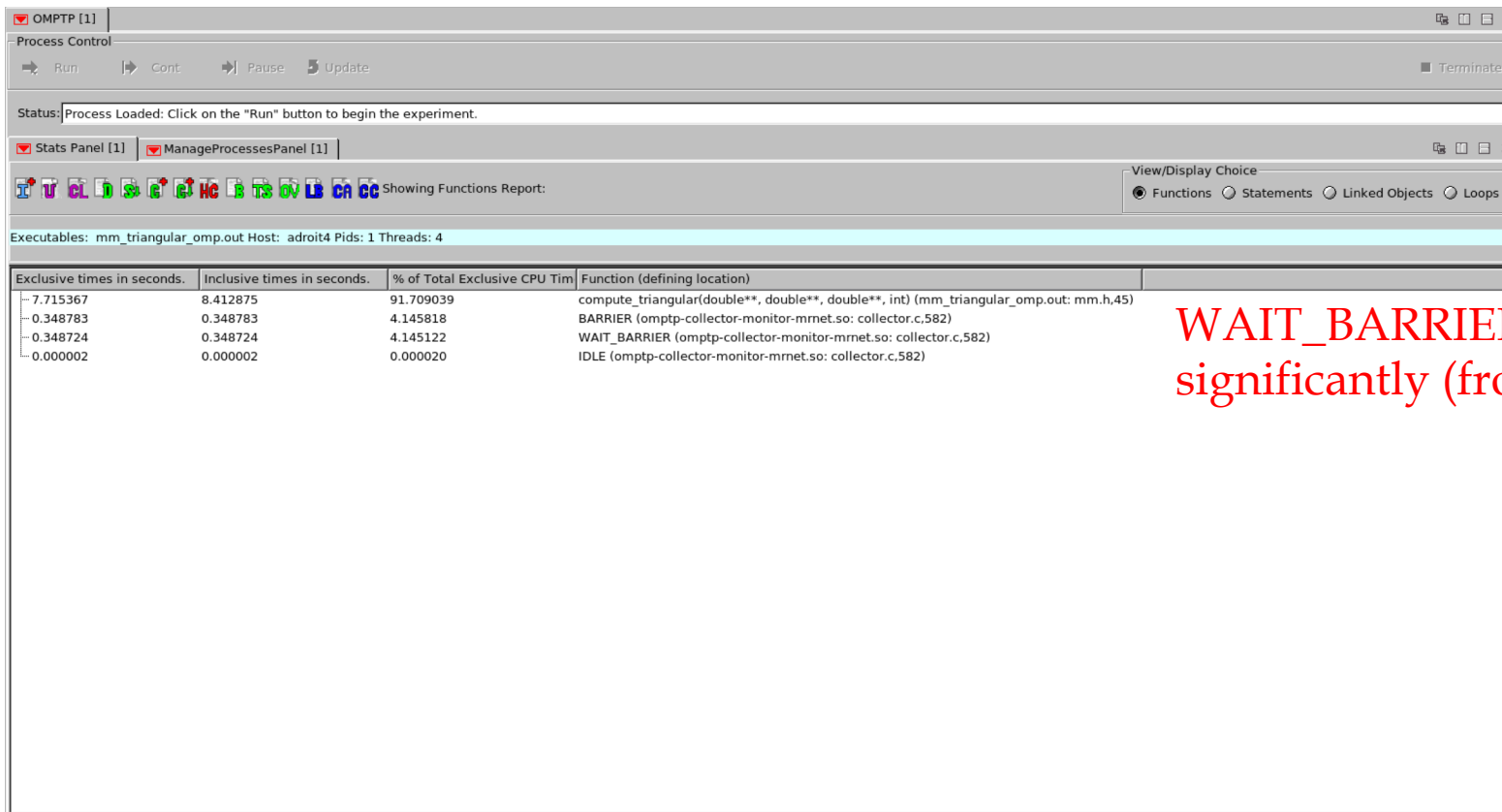
The screenshot displays the OMPTP (OpenMP Parallel Region) GUI. The window is titled "OMPTP [1]" and contains several panels:

- Process Control:** Includes buttons for "Run", "Cont", "Pause", "Update", and "Terminate". The status bar indicates "Process Loaded: Click on the 'Run' button to begin the experiment."
- Stats Panel [1]:** Shows a "ManageProcessesPanel [1]" and a "Showing Butterfly Report:" section.
- Executables:** Displays "mm\_triangular\_omp.out" on host "adroit4" with "Pids: 1" and "Threads: 4".
- Table:** A table with three columns: "Inclusive times in seconds.", "% of Total Inclusive CPU Time", and "Call Stack Function (defining location)".

| Inclusive times in seconds. | % of Total Inclusive CPU Time | Call Stack Function (defining location)  |
|-----------------------------|-------------------------------|--|
| 5.609802                    | 100.000000                    | compute_triangular(double**, double**, double**, int) (mm_triangular_omp.out: mm.h,45) |
| 5.609802                    | 100.000000                    | WAIT_BARRIER (omptp-collector-monitor-mrnet.so: collector.c,582)                       |

# 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
```



The screenshot shows the OMPTP [1] interface. The top section includes a 'Process Control' bar with buttons for Run, Cont., Pause, and Update, and a 'Terminate' button. Below this is a 'Status' bar indicating 'Process Loaded: Click on the "Run" button to begin the experiment.' The main area contains a 'Stats Panel [1]' and a 'ManageProcessesPanel [1]'. A 'View/Display Choice' dropdown is set to 'Functions'. The 'Executables' section shows 'mm\_triangular\_omp.out Host: adroit4 Pids: 1 Threads: 4'. The 'Showing Functions Report' table displays the following data:

| 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

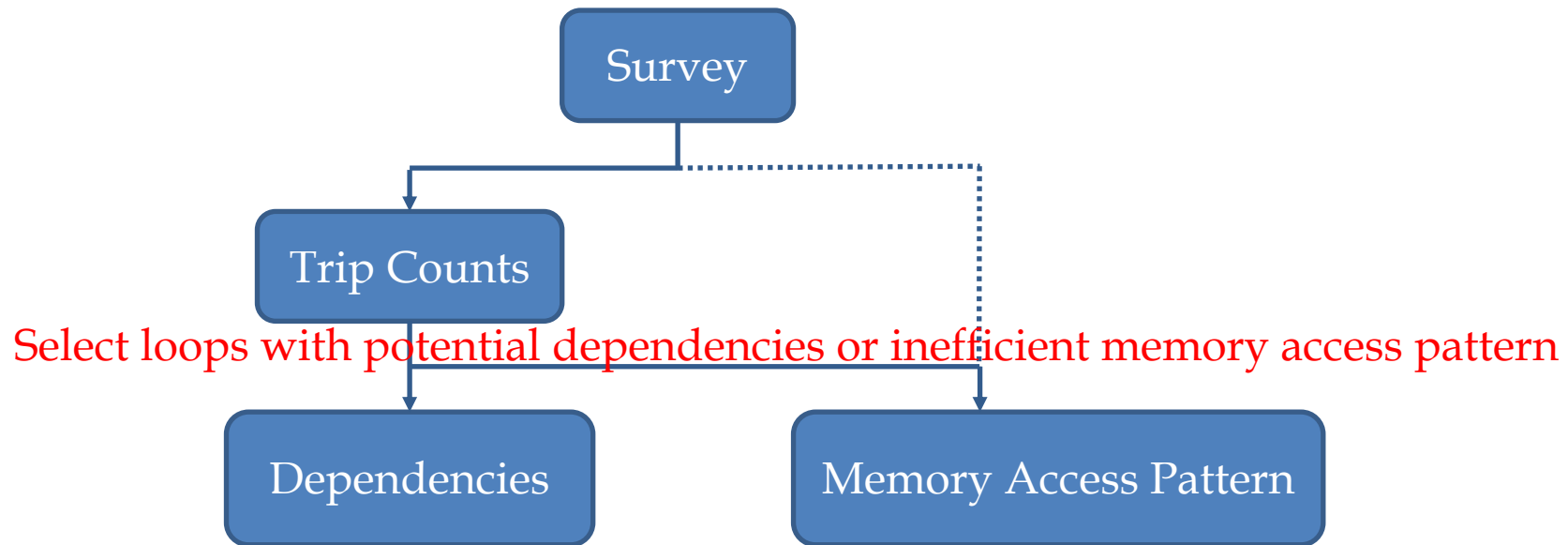


# Vectorization Advisor & Roofline

- 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 GUI: *advixe-gui mm-advisor*

The screenshot displays the Intel Advisor 2019 interface. At the top, it shows 'Elapsed time: 9.29s' and status indicators for 'Vectorized' and 'Not Vectorized'. The 'FILTER' is set to 'All Modules' and 'All Sources'. The 'Survey Source' is 'mm.h'. The 'Refinement Reports' tab is active.

The main panel shows the source code for 'compute\_interchange' in 'mm.h:33'. The code is a nested loop structure. The 'Total Time' for the selected code is 7.890s. The 'Loop/Function Time' for the innermost loop is 9.190s. The 'Traits' column indicates 'FMA' for the innermost loop.

The 'Call Stack' on the right shows the following entries:

- compute\_interchange - mm.h:33
- compute\_interchange - mm.h:32
- compute\_interchange - mm.h:31
- compute\_interchange - mm.h:30
- main - matmul\_2D.cpp:72
- main - matmul\_2D.cpp:47
- \_libc\_start\_main
- \_start

The bottom panel shows the assembly code for the module 'mm\_interchange\_icpc.out!0x401250'. The assembly code is shown in a table with columns for Address, Line, Assembly, Total Time, %, Self Time, %, and Traits. The assembly code includes instructions like 'xor %ebp, %ebp', 'mov \$0x1, %r13d', 'xor %ebx, %ebx', 'test %esi, %esi', 'jz 0x4012e2 <Block 8>', 'Block 5: 10000000', 'movq (%rdx,%r8,8), %r15', 'movq (%rdi,%r8,8), %r14', 'movq (%r10,%r11,8), %r13', 'Block 6: 5000000000', 'vmovsdq (%r14,%r11,8), %xmm1', and 'inc %rbp'. The 'Total Time' for the selected assembly code is 0.480s.

# 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*

Elapsed time: 0.84s Vectorized Not Vectorized FILTER: All Modules All Sources

Summary Survey & Roofline Refinement Reports

INTEL ADVISOR 2019

| Site Location                              | Loop-Carried Dependencies | Strides Distribution     | Access Pattern           | Footprint Estimate               |                               |
|--|---------------------------|--------------------------|--------------------------|----------------------------------|-------------------------------|
|  |                           |                          |                          | Max. Per-Instruction Addr. Range | First Instance Site Footprint |
| ▼ [loop in compute_interchange at mm.h:..] | ✔ No Dependencies Found   | No Information Available | No Information Available | No Information Available         | No Information Available      |

```
31 for (int i = 0 ; i < matrix_size; i++) {
32   for (int k = 0; k < matrix_size; k++) {
33     for (int j = 0; j < matrix_size; j++) {
34       C[i][j] += A[i][k] * B[k][j];
35     }
36   }
37 }
```

Memory Access Patterns Report Dependencies Report Recommendations

Problems and Messages

| ID | Type                      | Site Name   | Sources | Modules                 | State           |
|----|---------------------------|-------------|---------|-------------------------|-----------------|
| P1 | Parallel site information | loop_site_3 | mm.h    | mm_interchange_icpc.out | ✔ Not a problem |

Filter

Severity  
Information 1 item

Type  
Parallel site information 1 item

Source  
mm.h 1 item

Module  
mm\_interchange\_icpc... 1 item

State  
Not a problem 1 item

Parallel site information: Code Locations

| ID   | Instruction Address | Description   | Source  | Function            | Variable references | Module                  | State           |
|------|---------------------|---------------|---------|---------------------|---------------------|-------------------------|-----------------|
| ▼ X1 | 0x40129e            | Parallel site | mm.h:33 | compute_interchange |                     | mm_interchange_icpc.out | ✔ Not a problem |

```
31 for (int i = 0 ; i < matrix_size; i++) {
32   for (int k = 0; k < matrix_size; k++) {
33     for (int j = 0; j < matrix_size; j++) {
34       C[i][j] += A[i][k] * B[k][j];
35     }
36   }
37 }
```

Sort By Item Name

# 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`

Elapsed time: 3.47s | Vectorized | Not Vectorized | FILTER: All Modules | All Sources | Loops And Functions | All Threads | Customize View OFF

Summary | Survey & Roofline | Survey Source: mm.h | Refinement Reports

⚠ Your application might be underperforming  
Consider recompiling your application with zmm registers enabled.

| Function Call Sites and Loops            | Performance Issues | CPU Time  | Type       | Why No Vectorization? | Vectorized Loops | Compute Performance |
|--|--------------------|-----------|------------|-----------------------|------------------|---------------------|
|  |                    | Self Time | Total Time |                       | Vector ISA       | Self GFLOPS         |
| [loop in compute_interchange at mm.h:36] | 1 Ineffective ...  | 3.390s    | 3.390s     | Vectorized (Bod ...   | AVX2             | 5.900               |
| [loop in compute_interchange at mm.h:34] |                    | 0.040s    | 3.430s     | Scalar                |                  | 0.000               |
| [loop in init_matrix_2D at mm.h:74]      | 4 Assumed dep ...  | 0.020s    | 0.020s     | Scalar                |                  | 0.100               |
| f_start                                  |                    | 0.000s    | 3.460s     | Function              |                  |                     |
| f_main                                   |                    | 0.000s    | 3.460s     | Function              |                  |                     |
| f_init_matrix_2D                         |                    | 0.000s    | 0.020s     | Function              |                  |                     |
| [loop in init_matrix_2D at mm.h:73]      | 1 Assumed dep ...  | 0.000s    | 0.020s     | Scalar                |                  |                     |
| [loop in main at matmul_2D.cpp:72]       | 1 Assumed dep ...  | 0.000s    | 3.430s     | Scalar                |                  |                     |
| f_compute_interchange                    |                    | 0.000s    | 3.430s     | Function              |                  |                     |
| [loop in compute_interchange at mm.h:33] |                    | 0.000s    | 3.430s     | Scalar                |                  |                     |

Source | Top Down | Code Analytics | Assembly | Recommendations | Why No Vectorization?

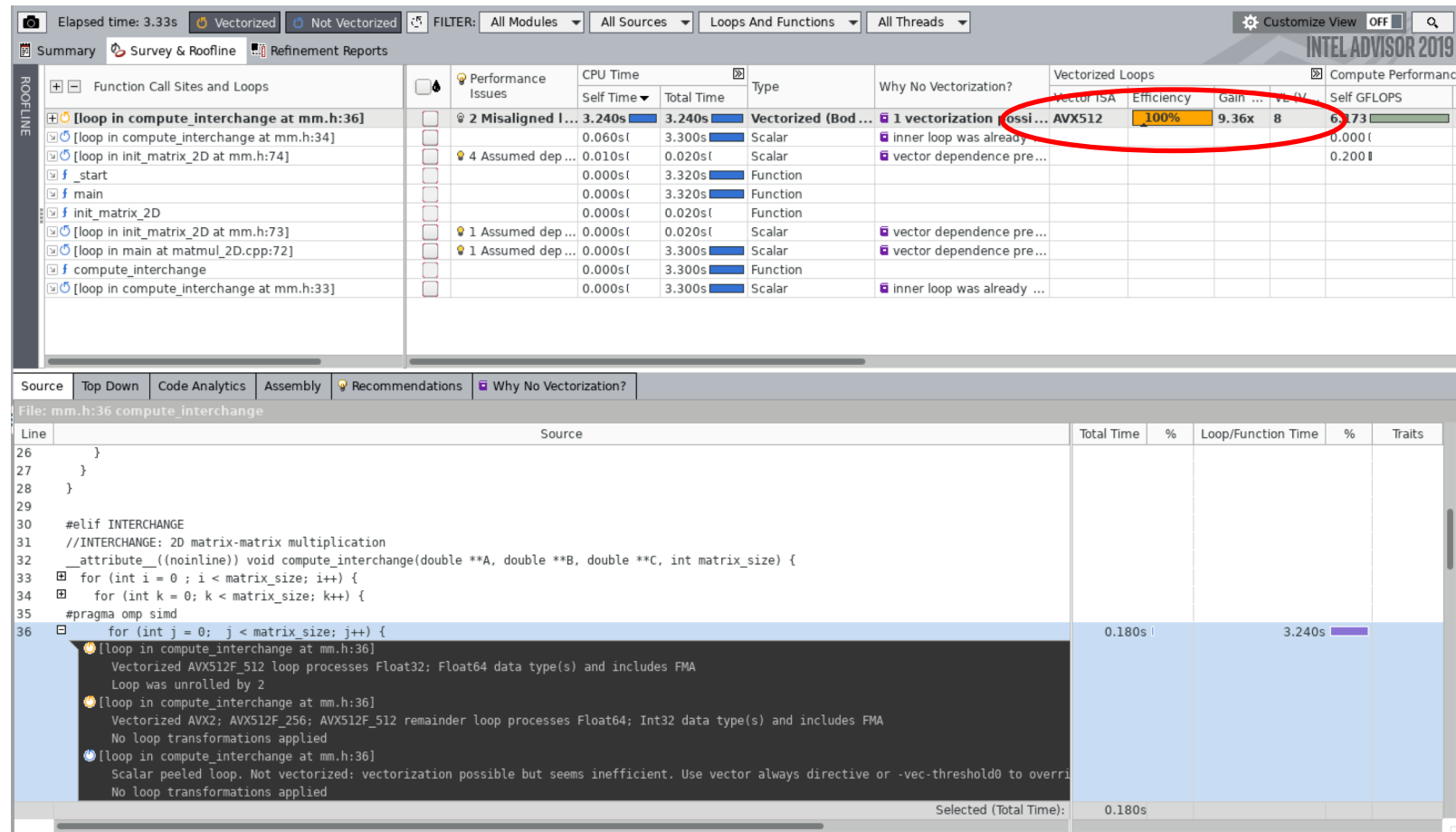
File: mm.h:36 compute\_interchange

| Line | Source  | Total Time | % | Loop/Function Time | % | Traits |
|------|---|------------|---|--------------------|---|--------|
| 28   | }   |            |   |                    |   |        |
| 29   |   |            |   |                    |   |        |
| 30   | #elif INTERCHANGE   |            |   |                    |   |        |
| 31   | //INTERCHANGE: 2D matrix-matrix multiplication  |            |   |                    |   |        |
| 32   | __attribute__((noinline)) void compute_interchange(double **A, double **B, double **C, int matrix_size) { |            |   |                    |   |        |
| 33   | for (int i = 0; i < matrix_size; i++) {   |            |   |                    |   |        |
| 34   | for (int k = 0; k < matrix_size; k++) {   |            |   |                    |   |        |
| 35   | #pragma omp simd  |            |   |                    |   |        |
| 36   | for (int j = 0; j < matrix_size; j++) {   | 0.140s     |   | 3.390s             |   |        |
|      | [loop in compute_interchange at mm.h:36]  |            |   |                    |   |        |
|      | Vectorized AVX; FMA loop processes Float64 data type(s) and includes FMA                                  |            |   |                    |   |        |
|      | Loop was unrolled by 4  |            |   |                    |   |        |
|      | [loop in compute_interchange at mm.h:36]  |            |   |                    |   |        |
|      | Scalar peeled loop  |            |   |                    |   |        |
|      | No loop transformations applied   |            |   |                    |   |        |
|      | [loop in compute_interchange at mm.h:36]  |            |   |                    |   |        |
|      | Vectorized AVX; FMA remainder loop processes Float64 data type(s) and includes FMA                        |            |   |                    |   |        |
|      | No loop transformations applied   |            |   |                    |   |        |
|      | [loop in compute_interchange at mm.h:36]  |            |   |                    |   |        |
|      | Scalar remainder loop   |            |   |                    |   |        |

Selected (Total Time): 0.140s

# 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

The screenshot displays the Intel Advisor 2019 interface. At the top, a status bar shows 'Elapsed time: 3.38s', 'Vectorized', and 'Not Vectorized'. Below this, a table lists various loops and functions with their respective metrics. A red arrow points to the 'Trip Count' column, which shows values like '62; 4; 1' for the first loop. Another red arrow points to the 'Call Count' column, showing '10000000; ...'. The bottom section of the interface contains recommendations, including 'Misaligned loop code present' and 'Ineffective peeled/remainder loop(s) present', with detailed explanations and code examples for each.

| Function Call Sites and Loops            | Total AI | FP Mask Utilization | Memory  | Trip Count | Call Count    | Traits                      | Data Typ ... | Num ... | Advan |
|--|----------|---------------------|---------|------------|---------------|-----------------------------|--------------|---------|-------|
| [loop in compute_interchange at mm.h:36] | 0.080    | 99.0%               | 250.240 | 62; 4; 1   | 10000000; ... | FMA                         | Float32; F   | 2; 4; 5 | Unrol |
| [loop in compute_interchange at mm.h:36] | 0.080    | 100.0%              | 248.000 | 62         | 10000000      | FMA                         | Float32; F   | 4       | Unrol |
| [loop in compute_interchange at mm.h:36] | 0.062    |                     | 0.960   | 4          | 7500000       | FMA                         | Float64      | 2       |       |
| [loop in compute_interchange at mm.h:36] | 0.078    | 62.0%               | 1.280   | 1          | 10000000      | FMA                         | Float64; I   | 5       |       |
| [loop in compute_interchange at mm.h:34] | 0.080    |                     | 0.480   | 1000       | 10000         |                             | Int32; UI    |         |       |
| [loop in init_matrix_2D at mm.h:74]      | 0.050    |                     | 0.040   | 1000       | 1000          | Divisions; Type Conversions | Float64; ... |         |       |
| [loop in zero_result at mm.h:103]        | 0        |                     | < 0.001 | 1000       | 10            |                             | Float64; ... |         |       |
| [loop in compute_interchange at mm.h:33] | 0.080    |                     | 0       | 1000       | 10            |                             |              |         |       |
| [loop in init_matrix_2D at mm.h:73]      | 0.050    |                     | < 0.001 | 1000       | 1             |                             |              |         |       |
| [loop in main at matmul_2D.cpp:72]       | 0.080    |                     | 0       | 10         | 1             |                             |              |         |       |

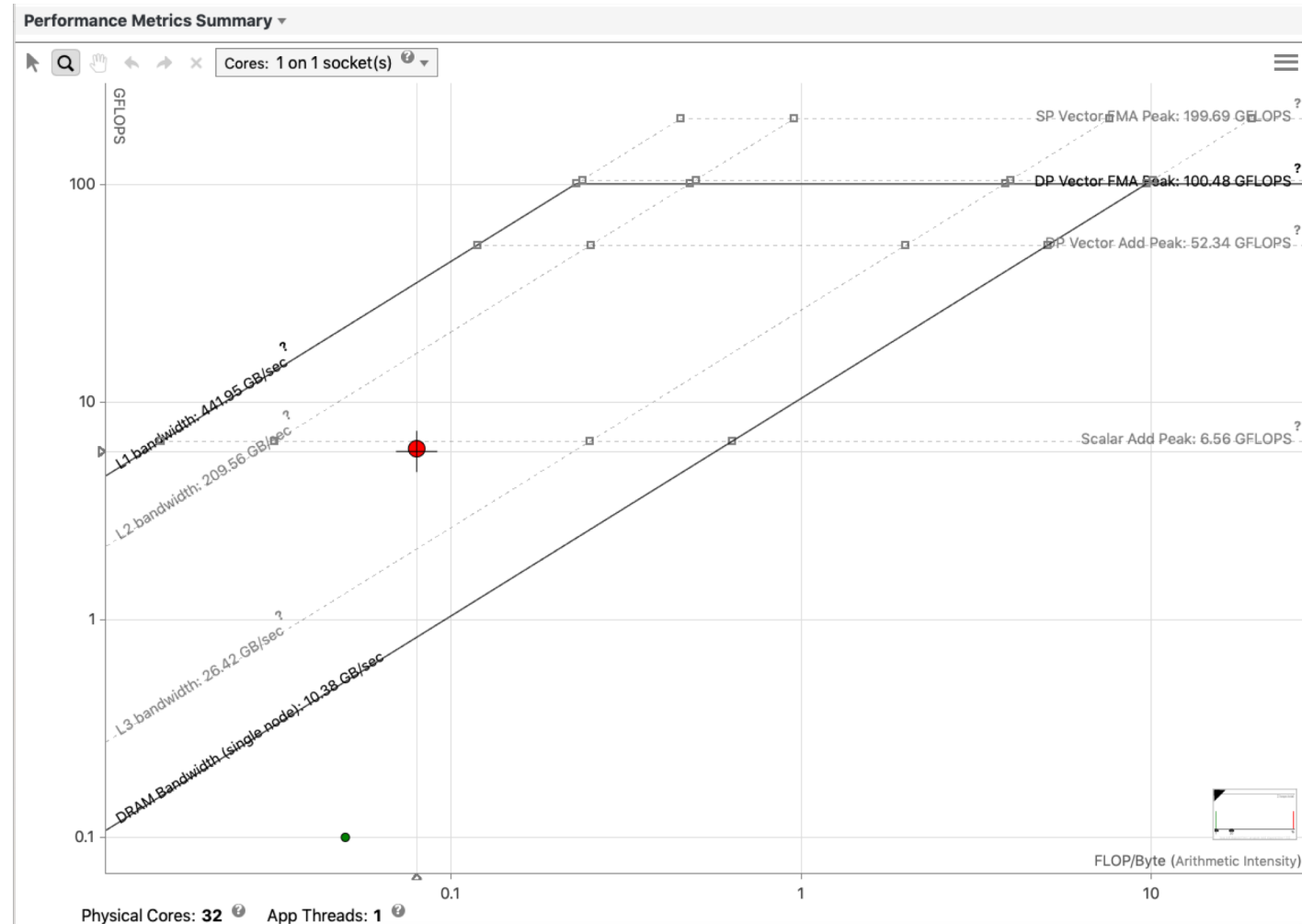
**Misaligned loop code present**  
Current placement of the loop in memory may result in inefficient use of the CPU front-end. Improve performance by aligning loop code.  
Force the compiler to align loop code  
Caution: Excessive code alignment may increase application binary size and decrease performance.  
Static analysis shows the loop may benefit from code alignment. To fix: Force the compiler to align the loop to a power-of-two byte boundary using a compiler directive for finer-grained control: `#pragma code_align (n)`  
Example  
...  
`#pragma code_align 32`  
`for (j = 0; j < m; j++)`  
...

**Ineffective peeled/remainder loop(s) present**  
All or some source loop iterations are not executing in the loop body. Improve performance by moving source loop iterations from peeled/remainder loops to the loop body.  
Align data  
Add data padding  
Specify the expected loop trip count



# Roofline Chart

- Trip counts analysis also collects **FLOPS** (FLoating-point **O**perations Per **S**econds)
- 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;

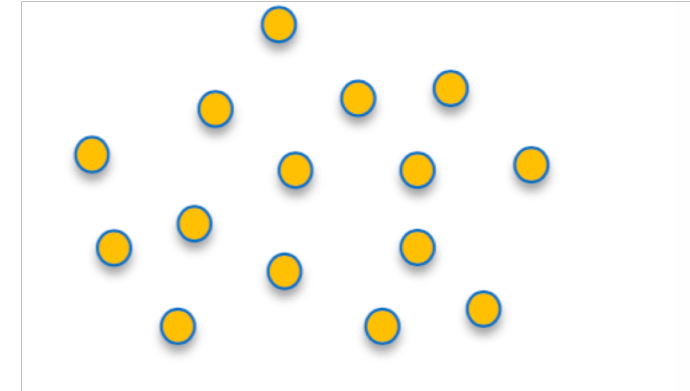
    const float xi = particle[i].x;
    const float yi = particle[i].y;
    const float zi = particle[i].z;

    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;
}
```



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

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

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?

Elapsed time: 4.12s Vectorized Not Vectorized FILTER: All Modules All Sources Loops And Functions All Threads Customize View OFF

Summary Survey & Roofline Refinement Reports

INTEL ADVISOR 2019

| Function Call Sites and Loops           | Performance Issues  | CPU Time  |            | Type              | Why No Vectorization?      | Vectorized Loops |            |          |          | Instruction Traits |
|---|---------------------|-----------|------------|-------------------|----------------------------|------------------|------------|----------|----------|--------------------|
|   |                     | Self Time | Total Time |                   |                            | Vecto...         | Efficiency | Gain ... | VL (V... |                    |
| [loop in MoveParticles at nbody.cpp:76] | 2 Possible ineff... | 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. Reci         |
| 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       |

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

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

# 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?

Elapsed time: 1.36s Vectorized Not Vectorized FILTER: All Modules All Sources Loops And Functions All Threads Customize View OFF

Summary Survey & Roofline Survey Source: nbody.cpp Refinement Reports

INTEL ADVISOR 2019

| Function Call Sites and Loops           | Performance Issues | CPU Time  |            | Type             | Why No Vectorization?      | Vectorized Loops |            |          |          | Instr  |
|---|--------------------|-----------|------------|------------------|----------------------------|------------------|------------|----------|----------|--------|
|   |                    | Self Time | Total Time |                  |                            | Vecto...         | Efficiency | Gain ... | VL (V... |        |
| [loop in MoveParticles at nbody.cpp:76] |                    | 0.900s    | 1.350s     | Vector Function  |                            | AVX512           | ~85%       | 13.60x   | 16       | FMA    |
| __svml_invsqrtf16_z0                    |                    | 0.450s    | 0.450s     | Function         |                            | AVX512           |            |          |          | FMA;   |
| _start                                  |                    | 0.000s    | 1.350s     | Inlined Function |                            |                  |            |          |          | FMA;   |
| MoveParticles                           |                    | 0.000s    | 1.350s     | Scalar           | inner loop was already ... |                  |            |          |          | FMA;   |
| [loop in main at nbody.cpp:55]          |                    | 0.000s    | 1.350s     | Function         |                            |                  |            |          |          | FMA;   |
| main                                    |                    | 0.000s    | 1.350s     | Scalar           | compile time constraint... |                  |            |          |          | Divisi |
| [loop in main at nbody.cpp:204]         |                    | 0.000s    | 1.350s     | Scalar           |                            |                  |            |          |          |        |

Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

1.350s  
Vectorized (Body) Total time

AVX512F\_512 0.900s  
Instruction Set Self time

Static Instruction Mix Summary

- Memory 18% (4)
- Vector 14% (3)
  - AVX-512 14% (3)
- Scalar 4% (1)
- Compute 68% (15)
  - Vector 64% (14)
    - AVX-512 64% (14)
  - Scalar 4% (1)
  - Other 14% (3)

No Trip Counts data available.  
Collect Trip Counts to get more accurate recommendations and vectorization efficiency data.

~85% Vectorization Efficiency

13.60x  
Vectorization Gain

Traits  
FMA

Compiler: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64,  
Version: 19.0.3.199 Build 20190206  
Compiler estimated gain: 13.60x

**Compiler Notes On Vectorization:**

- Unaligned Access in Vector Loop

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



# Create Snapshot for Comparison

The screenshot shows the Intel Advisor 2019 interface. The main window displays the 'Vectorization Advisor' tab, which includes a 'Program' section with 'Elapsed Time' (4.03s), 'Vector Instructions' (93.985), and 'Number of CPU' (0.17). The 'Performance' section shows 'Total CPU time' (4.02s), 'Time in 1 vectorized loop' (0s), and 'Time in scalar code' (0s). The 'Vectorization Gain/Efficiency' section shows 'Vectorized Loops Gain/Efficiency' (12.61x) and 'Program Approximate Gain' (12.61x). The 'OP/S and Bandwidth' section is also visible.

A dialog box titled 'Create a Result Snapshot' is open, allowing the user to save the current results. The dialog includes the following fields and options:

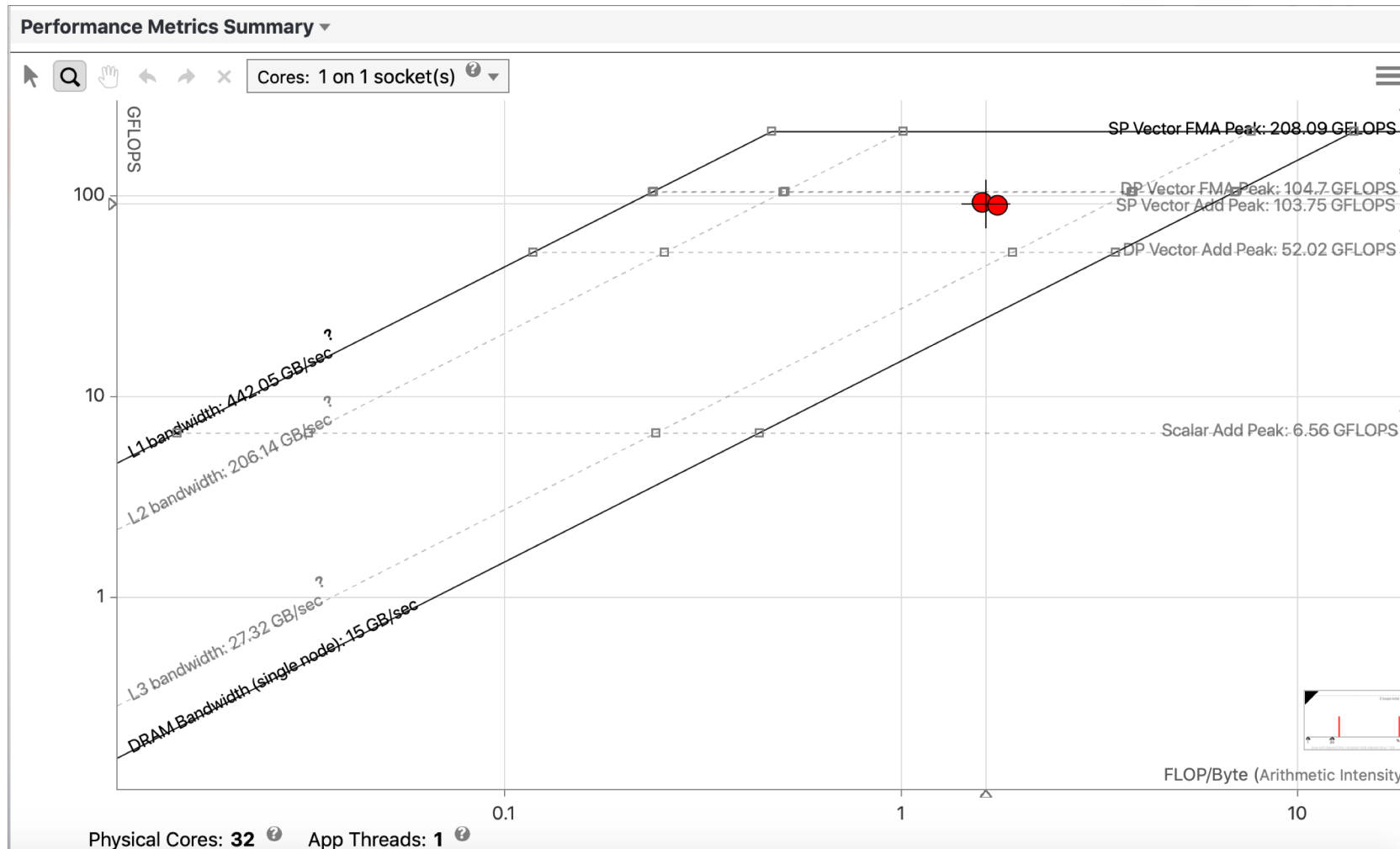
- Result name:** original
- ☒ Cache sources
- ☒ Cache binaries
- ☐ Pack into archive
- Result path:** (empty field with a 'Browse...' button)
- Buttons:** OK, Cancel

The background window also shows a sidebar with 'Vectorization Workflow' and 'Threading Workflow' tabs, and a 'Run Roofline' section with 'Collect' and 'With Callstacks' options. The '1. Survey Target' section includes 'Collect' and 'Mark Loops for Deeper Analysis' options. The '2.1 Check Memory Access Patterns' section includes 'Collect' and 'Re-finalize Sur...' options.



# Roofline Comparison

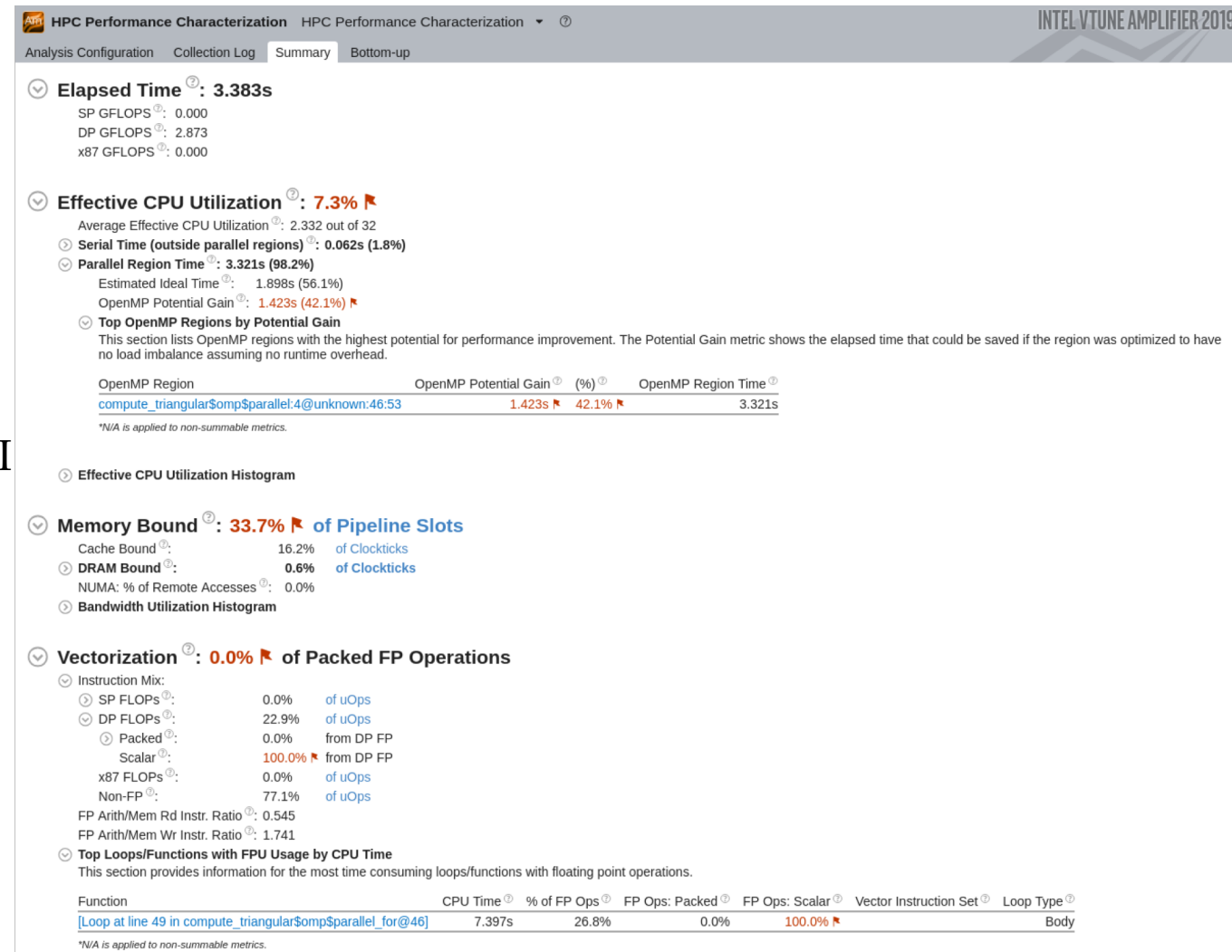
- Can you make a roofline chart for the original code and the optimized one?



# 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

The image displays the 'Configure Analysis' window of Intel VTune Amplifier 2019. The window is divided into two main sections: 'WHERE' and 'WHAT'.

**WHERE:** Shows 'Local Host' as the analysis target.

**WHAT:** Shows 'Launch Application' as the analysis type. Below this, there are fields for 'Application' (set to `/home/beiwang/codas_perftools/mm_triangular_omp_icpc.out`), 'Application parameters' (set to `1000`), and a checkbox for 'Use application directory as working directory' (checked). The 'Working directory' is set to `/home/beiwang/codas_perftools`. An 'Advanced' button is also visible.

To the right of the configuration window is a diagram titled 'Find your analysis direction'. It shows a central arrow pointing down to 'Parallelism' (Want to assess the compute efficiency of your multi-threaded application?). From this central point, arrows branch out to 'Hotspots' (Want to find out where your application spends time and optimize your algorithms?) and 'Microarchitecture' (Want to see how efficiently your code is using the underlying hardware?).

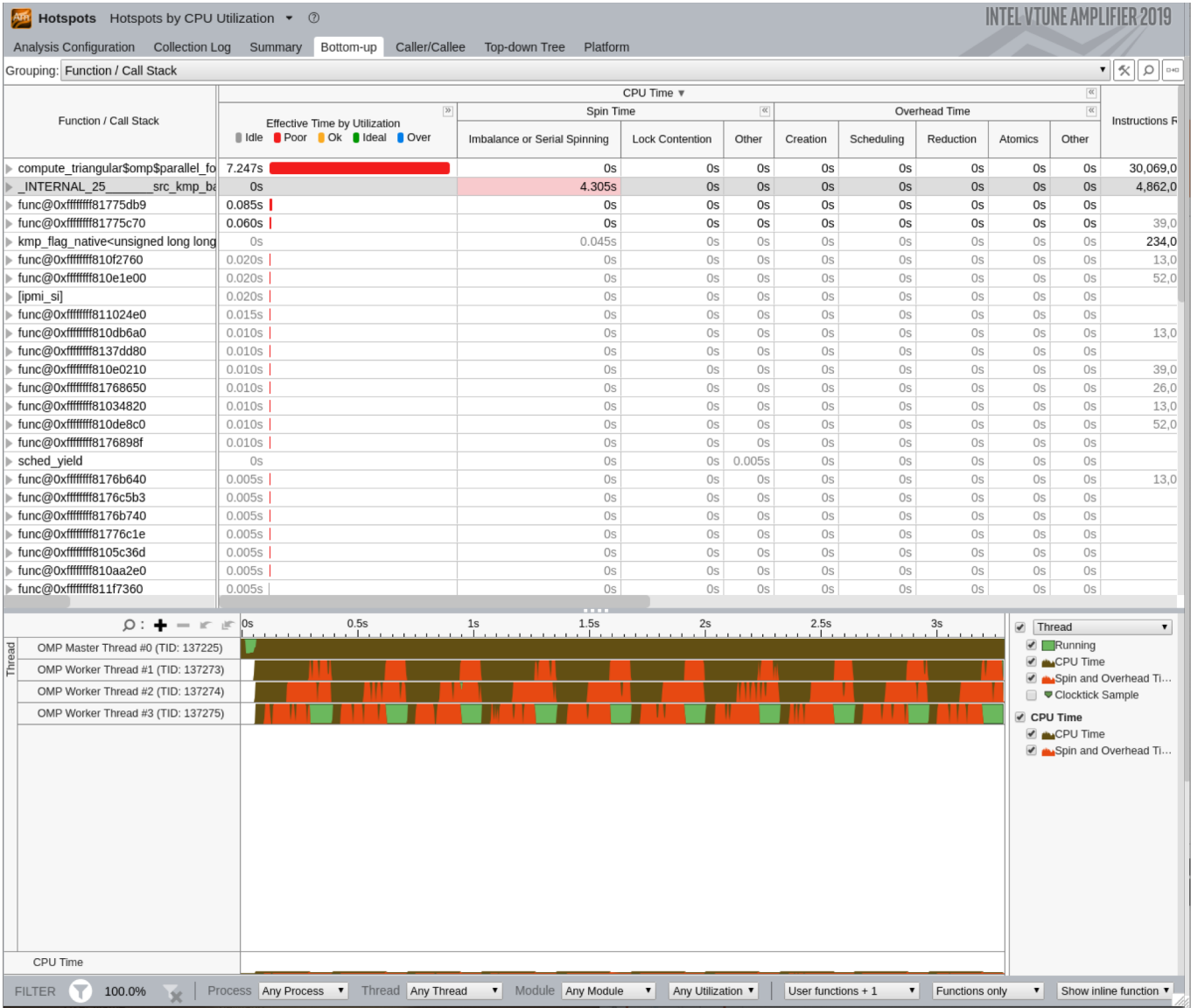
Below the diagram, there are three main categories of analysis types:

- Hotspots:** Includes 'Hotspots' and 'Memory Consumption'.
- Microarchitecture:** Includes 'Microarchitecture Exploration' and 'Memory Access'.
- Parallelism:** Includes 'Threading' and 'HPC Performance Characterization'.

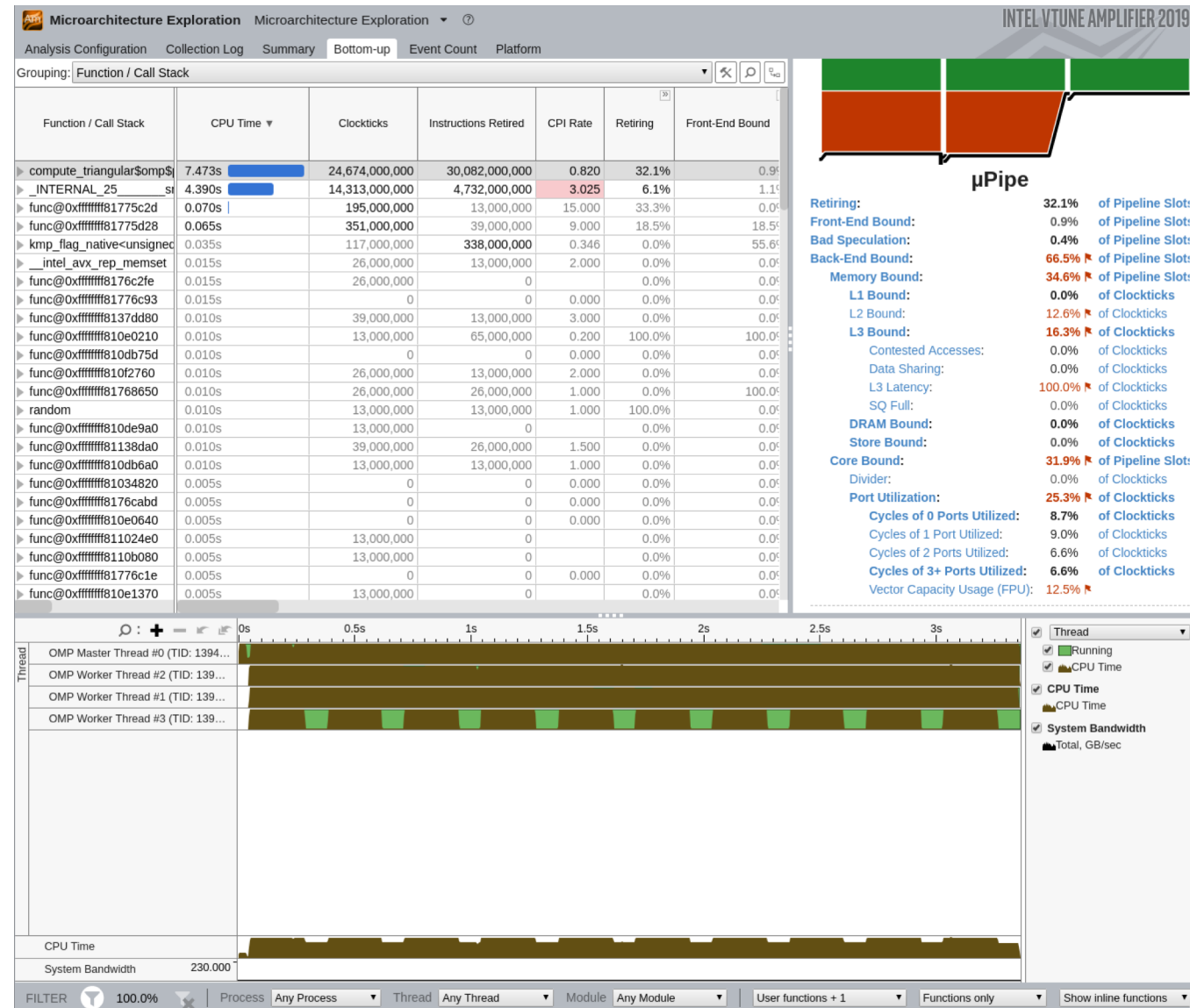
Below these, there are two more categories:

- Platform Analysis:** Includes 'System Overview', 'CPU/GPU Concurrency', 'GPU Compute/Media Hotspots', 'GPU Rendering (preview)', 'Input and Output', and 'CPU/FPGA Interaction (preview)'.
- Custom Analysis:** Includes 'HPC Performance Characterization 0', 'Threading 0', 'Threading 1', 'Threading 2', and 'Threading 3'.

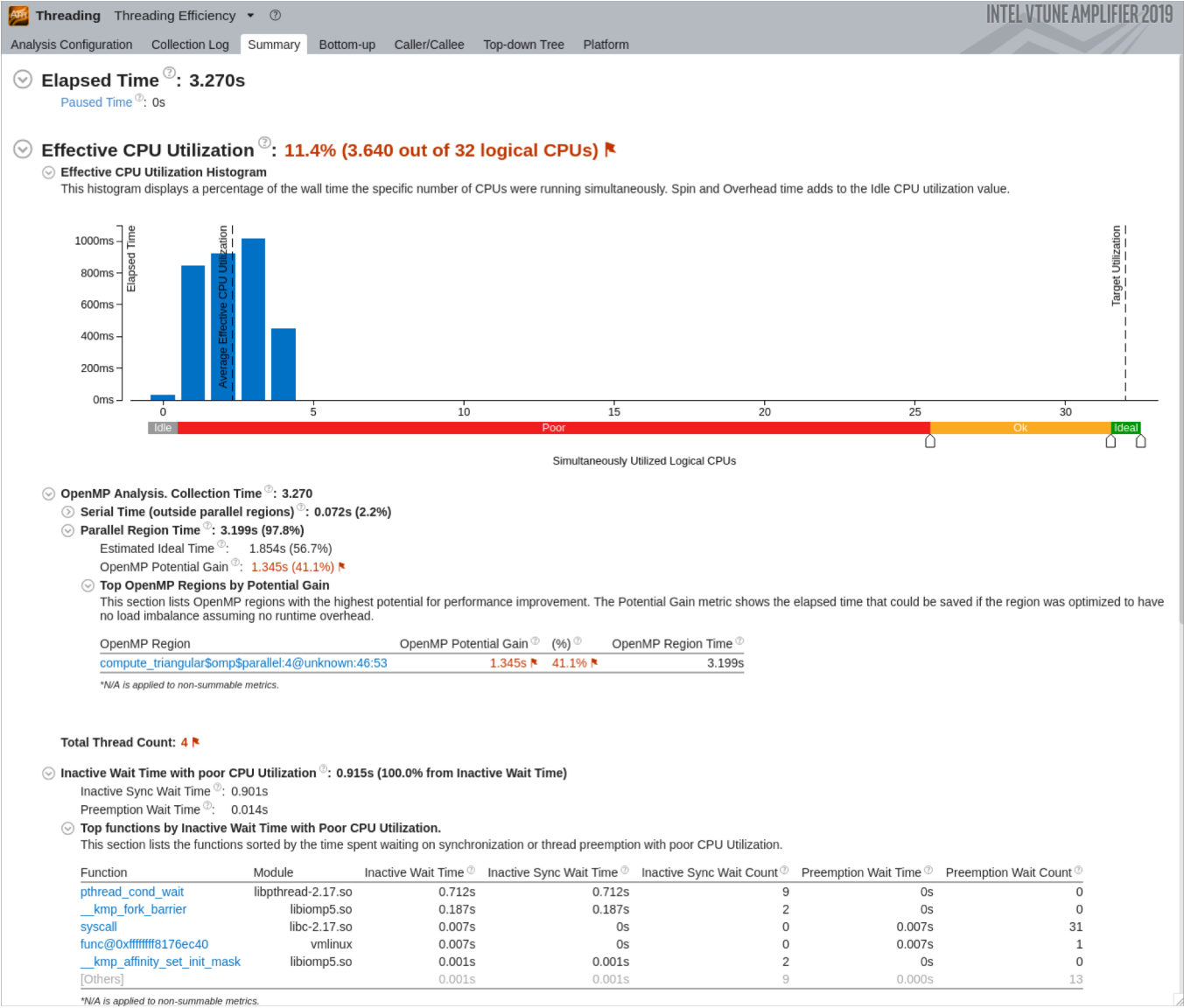
# Hotspots



# Microarchitecture Exploration



# Threading



# 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>