GETTING STARTED WITH PERFORMANCE OPTIMIZATION AND TUNING

Bei Wang, Ph.D.
HPC Software Engineer
Research Computing, Princeton University

Princeton Research Computing Bootcamp
Oct 31, 2018

About Me

- Bei Wang, Ph.D., **HPC Software Engineer** at Research Computing
- 7 years at Princeton University Working on Research and Development in Parallel Computing Applications (plasma physics and fluid dynamics domains)
- Co-PI of **Intel Parallel Computing Center (IPCC)** at Princeton
- Office: 334 Lewis Library (TW), 111 Peyton Hall (MTF)
- Email: beiwang@princeton.edu

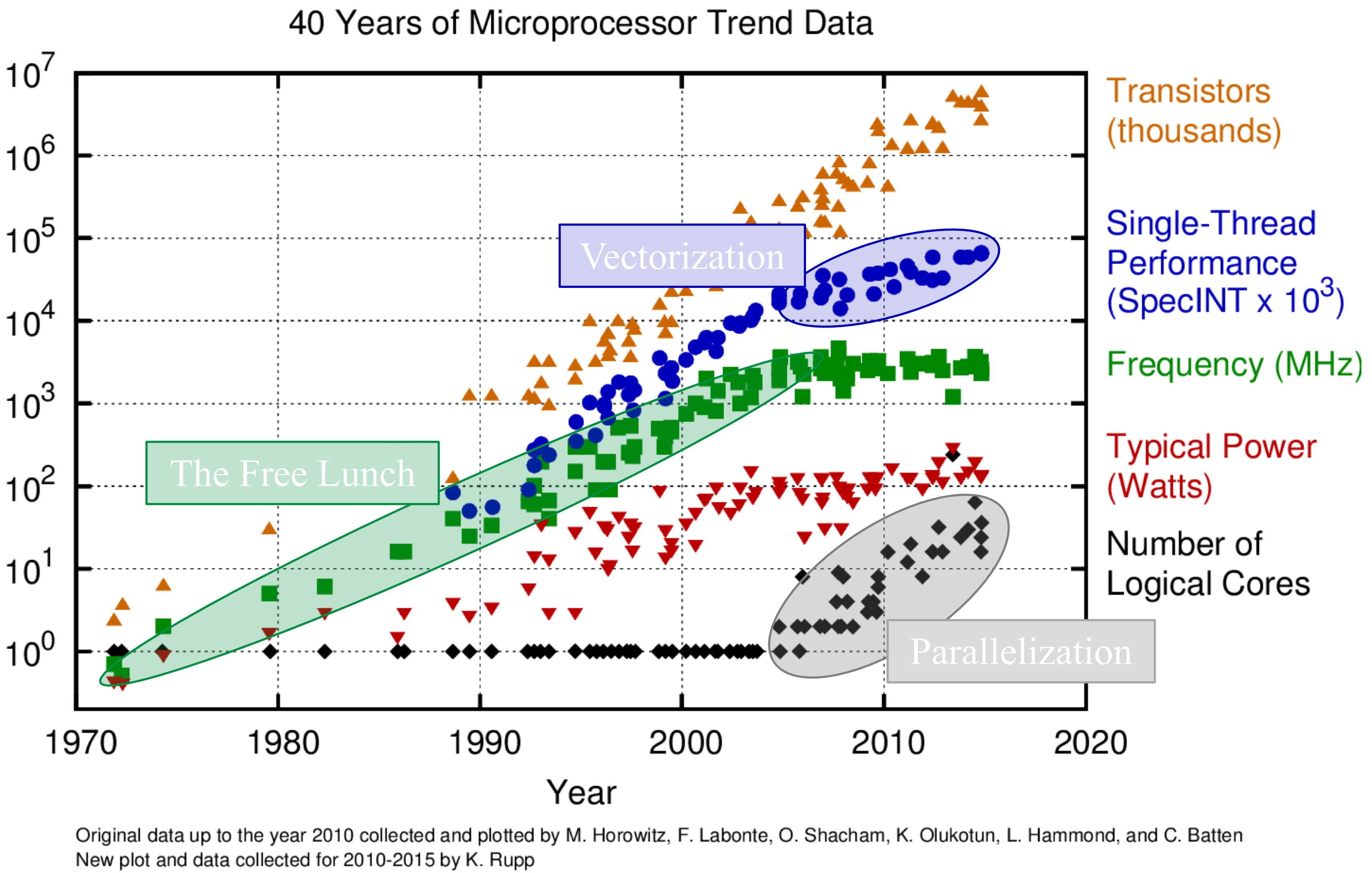
Outline

- An introduction to the **idea of performance analysis**
  - Methodology
  - Workflow
  - Measurement tools
  - Hands-on
- Focused primarily on the **HPC recourses at Princeton**
  - Hardware: Intel CPU
  - Tools: Intel performance tuning tools
  - Scientific application codes written with C/C++ and Fortran languages
  - **Most principles apply universally**

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**
  - Identifying bottlenecks, determining efficiency and **eliminating the bottlenecks** if possible
  - **Incrementally** complete tuning until the performance requirements are satisfies

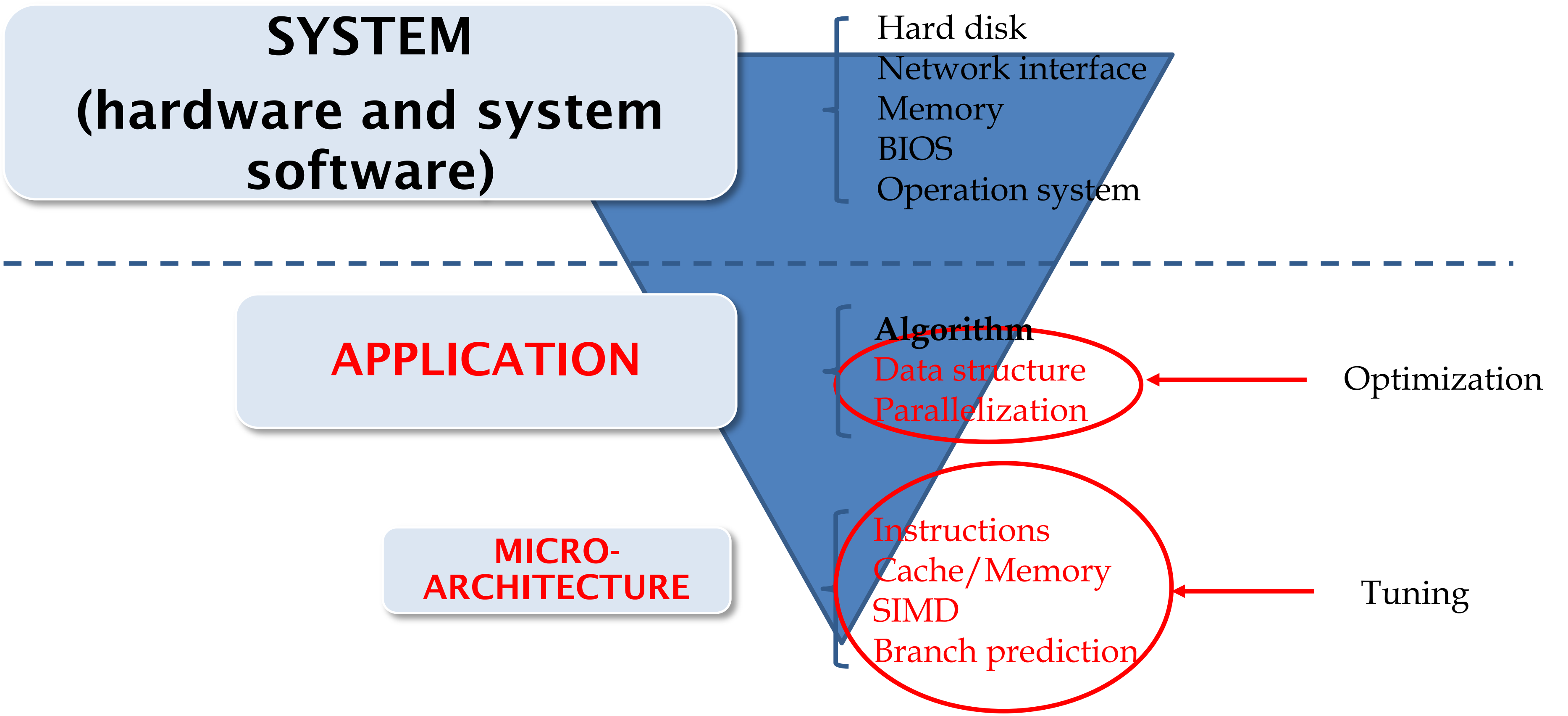
“The Free Lunch is Over”



https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Performance Analysis Methodology:

A top-down approach



"Optimizing HPC Applications with Intel Cluster Tools", Book, Alexander Supalov, Andrey Semin, Michael Klemm, Chris Dahnken, 2014

Performance Analysis Workflow:

Prepare

- **Create a benchmark**
  - Choose a workload which is **measurable, representative, static** and **reproducible**
  - Choose a performance metric which is **quantifiable**
- **Document**
  - Code generation: compiler version, flags etc

  E.g.: in Makefile

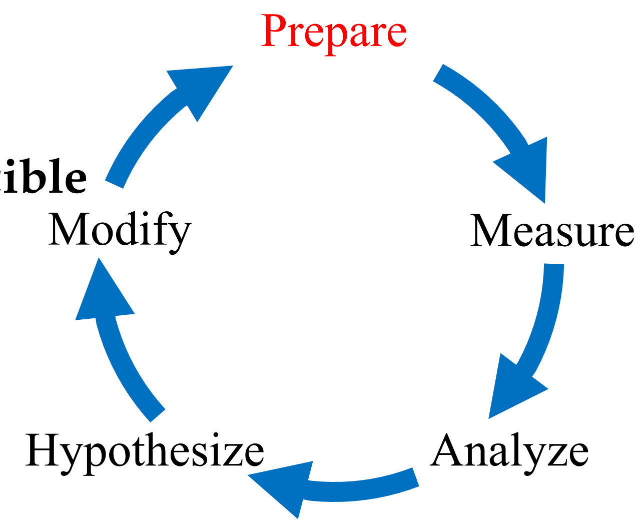
```
GIT_VERSION:=$(shell sh -c './GIT-VERSION-GEN')
COMPILER_VERSION:="$(CC)-$(shell $(CC) --version | head -n1 | cut -d' ' -f4)"
BUILD_HOST=$(shell sh -c './BUILD-HOST-GEN')

# obviously, we can't pass all options to the compiler as a compiler flag.
#'Important' flags, like optimizations, math behavior twiddles, and arch flags should go here
BUILD_FLAGS:=$(CFLAGS) $(COPTFLAGS)

BUILD_FLAGS_STR:=$(shell sh -c "printf %q \"$(BUILD_FLAGS)\"")

CINFOFLAGS=-DGIT_VERSION=\"$(GIT_VERSION)\" -DCOMPILER_VERSION=\"$(COMPILER_VERSION)\" -DBUILD_HOST=\"$(BUILD_HOST)\" -DBUILD_FLAGS=\"\
$(BUILD_FLAGS_STR)\"
```

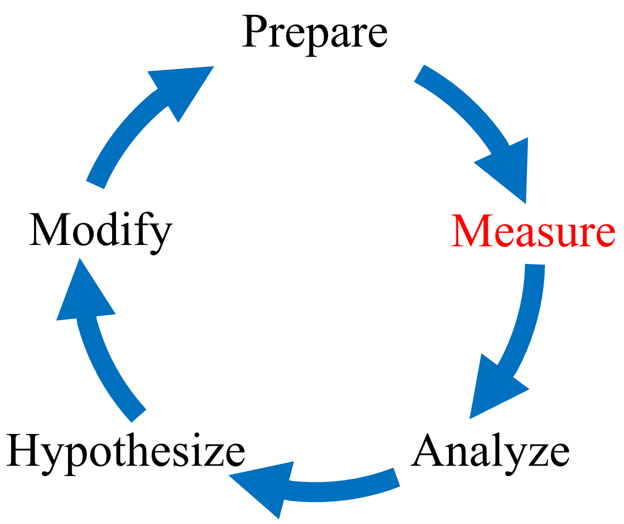
  - Basic variants: thread count, affinity, working set size



Performance Analysis Workflow:

Measure

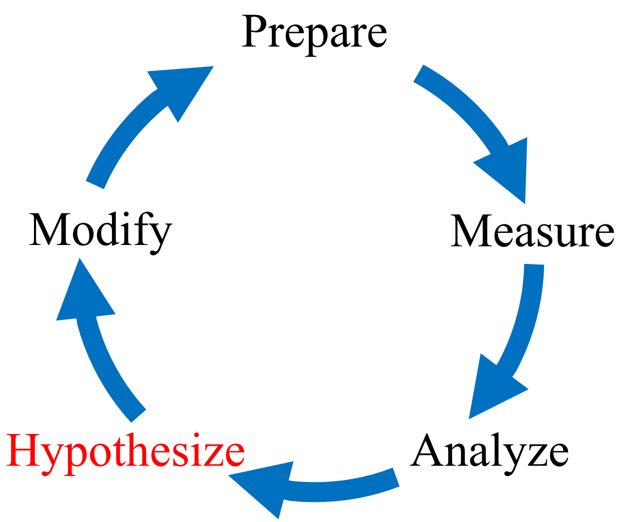
- **Time program run time**
  - linux commands: time, prof stat
  - Get an idea of overall run time
- **Put timer around loops/functions**
  - gettimeofday, MPI_Wtime, omp_get_wtime
  - Works for small code base to identify hotspots
- **Use profilers** – recommended
  - What to collect?
    - Timing, hardware counter, trip counts, call stack etc
  - How to collect?
    - Sampling-based
      - Records system state at periodic intervals
      - Not intrusive-low overhead
    - Instrumenting-based
      - Add instructions in the source code to collect detailed information for interested events
      - Intrusive-high overhead for frequent events
    - Tracing-based
      - Records all operations
      - Intrusive-high overhead



Performance Analysis Workflow:

Hypothesize

- Why is my code slow?
  - CPU bound
  - Memory bound
  - I/O bound
  - Network bound
  - Unbalanced Workload (Parallel)

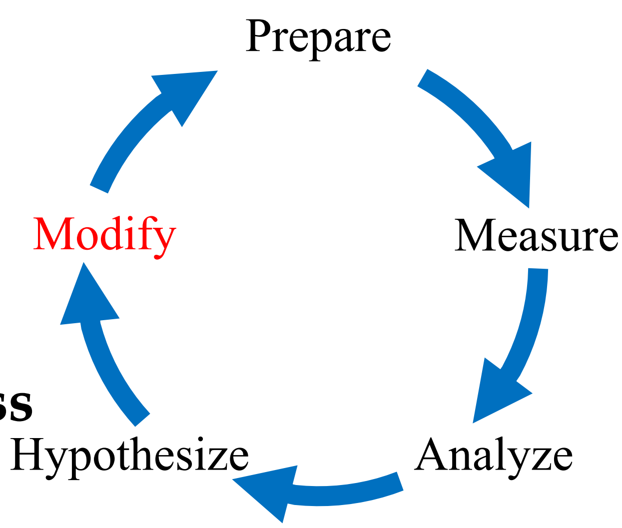


- What is the best I can expect?
  - CPU
  - Memory/Cache
  - I/O
  - Network
  - Parallel Scaling

Performance Analysis Workflow:

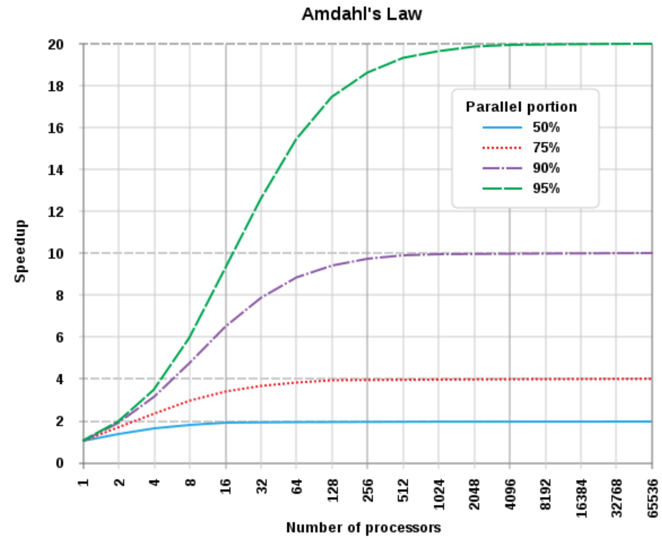
Modify

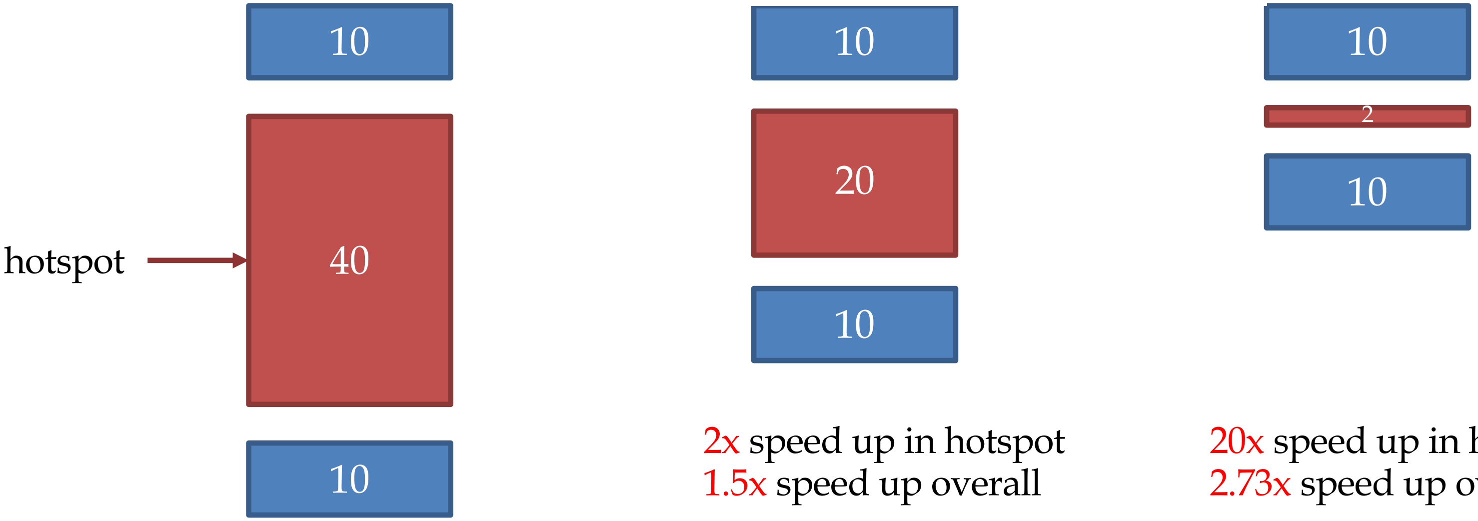
- Change only **one thing at a time**
- Consider the ease (difficulty) of implementation
- Keep **track** of all **changes**
- Apply regression test to **ensure correctness** after each change



Words to Remember:
Fast computing of wrong result is completely irrelevant!

When to Stop? Amdahl's law



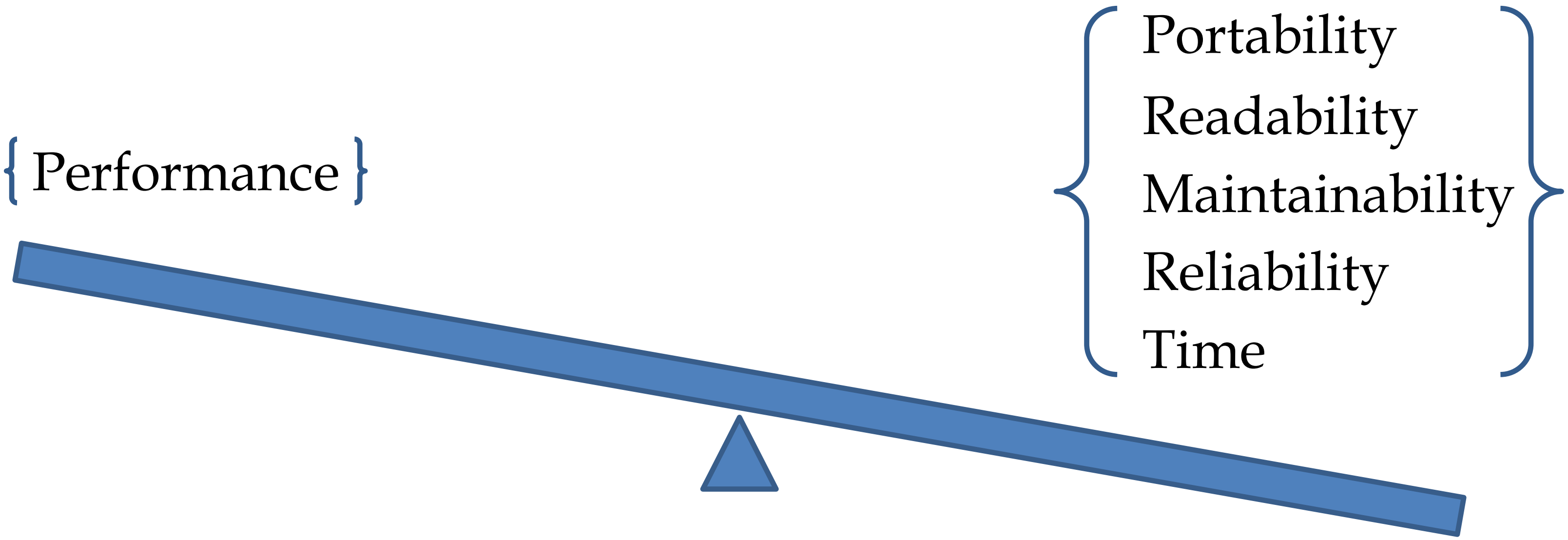


2x speed up in hotspot
1.5x speed up overall

20x speed up in hotspot
2.73x speed up overall

https://en.wikipedia.org/wiki/Amdahl%27s_law

Performance Tuning Tradeoff



Using Profilers

- Include debug symbols in executable
  - ***-g***
  - Ability to trace performance back to source code
- Use release-build optimization flags
  - E.g., ***-O3***, *-xhost* (Intel), *-ipo* (Intel)
  - Don't waste time optimizing code the compiler can do automatically!
- Keep useful debugging information
  - E.g., *-debug inline-debug-info* (Intel) or *-debug full* (Intel)
  - Sometimes the compiler will optimize out useful regions
- Include required profiling flags during compiling
  - E.g., *-pg* (Gprof)
  - Needed by instrumentation-based profiling

Popular Tools

- Many free and commercial products, for example
  - Linux **Perf**: Sampling profiler with support of hardware events on several architectures
  - Linux **Oprofile**: Sampling profiler for Linux that counts cache misses, stalls, memory fetches, etc
  - GNU **Gprof**: Several tools with combined sampling and call-graph profiling
  - Valgrind (**Callgrind**): System for debugging and profiling; supports tools to either detect memory management and threading bugs, or profile performance
  - ARM **MAP**: Performance profiler. Shows I/O, communication, floating point operation usage and memory access costs
  - Intel **VTune** Amplifier XE: Tool for serial and threaded performance analysis. Hotspot, call tree and threading analysis works on both Intel and AMD x86 processors.
  - Intel **Advisor**: Tool for vectorization
  - …

https://en.wikipedia.org/wiki/List_of_performance_analysis_tools

Tools at Princeton

- Research Computing at Princeton supports a number of licensed performance tuning tools
  - **Profiling**
    - ARM MAP
    - Intel VTune
  - **Tracing**
    - Intel Trace Analyzer and Collector
  - **Vectorization**
    - Intel Advisor
  - **Debugging**
    - ARM DDT
    - Intel Inspector

Intel Tools at Princeton

- **Application Performance Snapshot**
  - High level tool for an overview of performance

  mpirun -n <N> aps ${EXE} ${ARGS}
- **Intel VTune Amplifier**
  - Node level counter based performance profiler

  mpirun –n 1 amplxe-cl –c ${COLL} -finalization-mode=deferred -- ${EXE} ${ARGS}: -n <N-1> ${EXE} ${ARGS}
  - Recommended Collections:

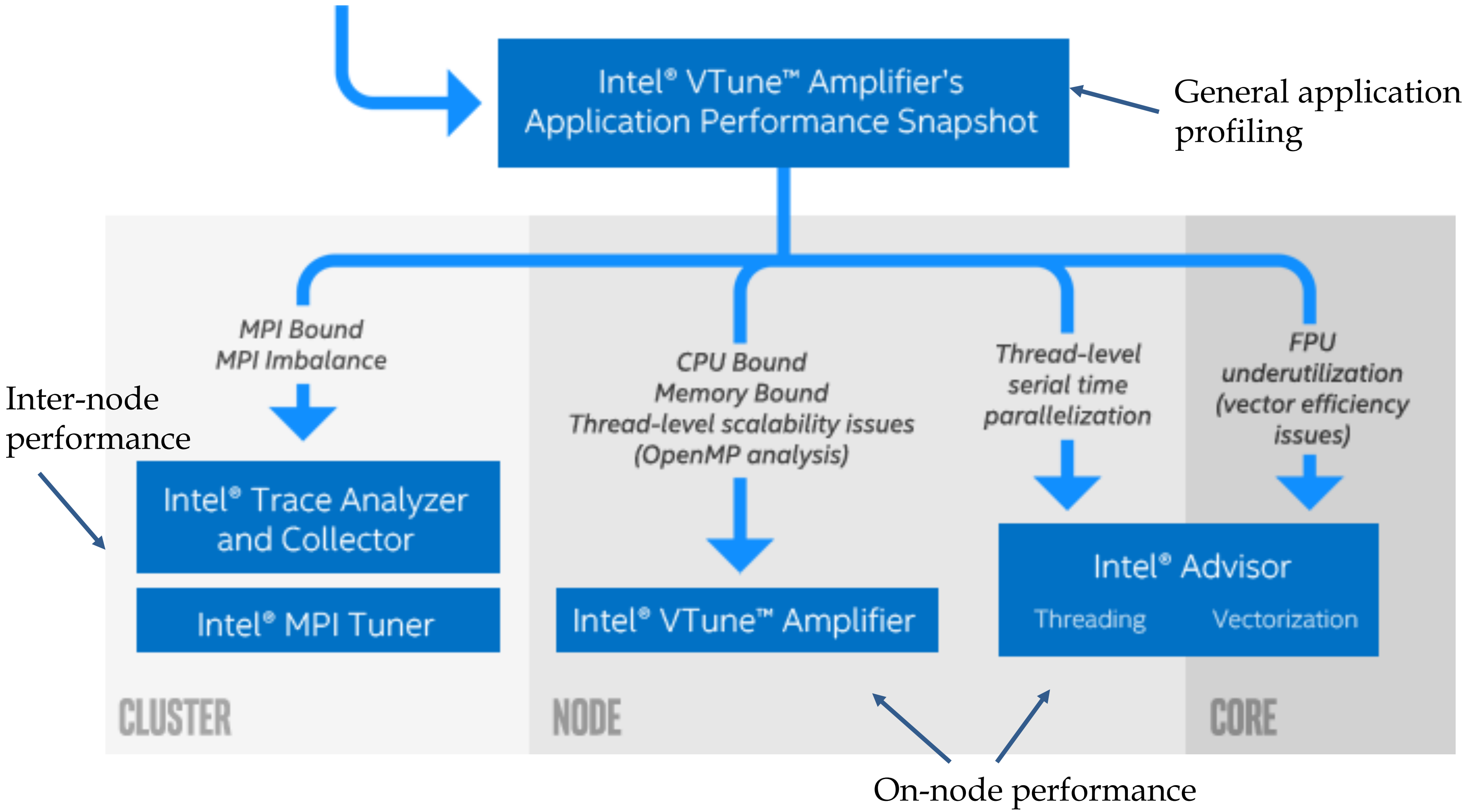
  advanced-hotspots, general-exploration, memory-access, hpc-performance
  - Finalize on headnode

  amplxe-cl -finalize -r ${RESULT_DIR} -search-dir ${PATH_TO_OBJS_AND_EXE} -source-search-dir ${PATH_TO_SOURCE}
- **Intel Advisor**
  - Node level vectorization and threading information, and roofline

  mpirun -n 1 advixe-cl -c survey -project-dir -- ${EXE} ${ARGS}: -n <N-1> ${EXE} ${ARGS}

  mpirun -n 1 advixe-cl -c tripcounts -flop -project-dir -- ${EXE} ${ARGS}: -n <N-1> ${EXE} ${ARGS}
- **Intel Trace Analyzer and Collector**
  - At scale MPI performance analyzer

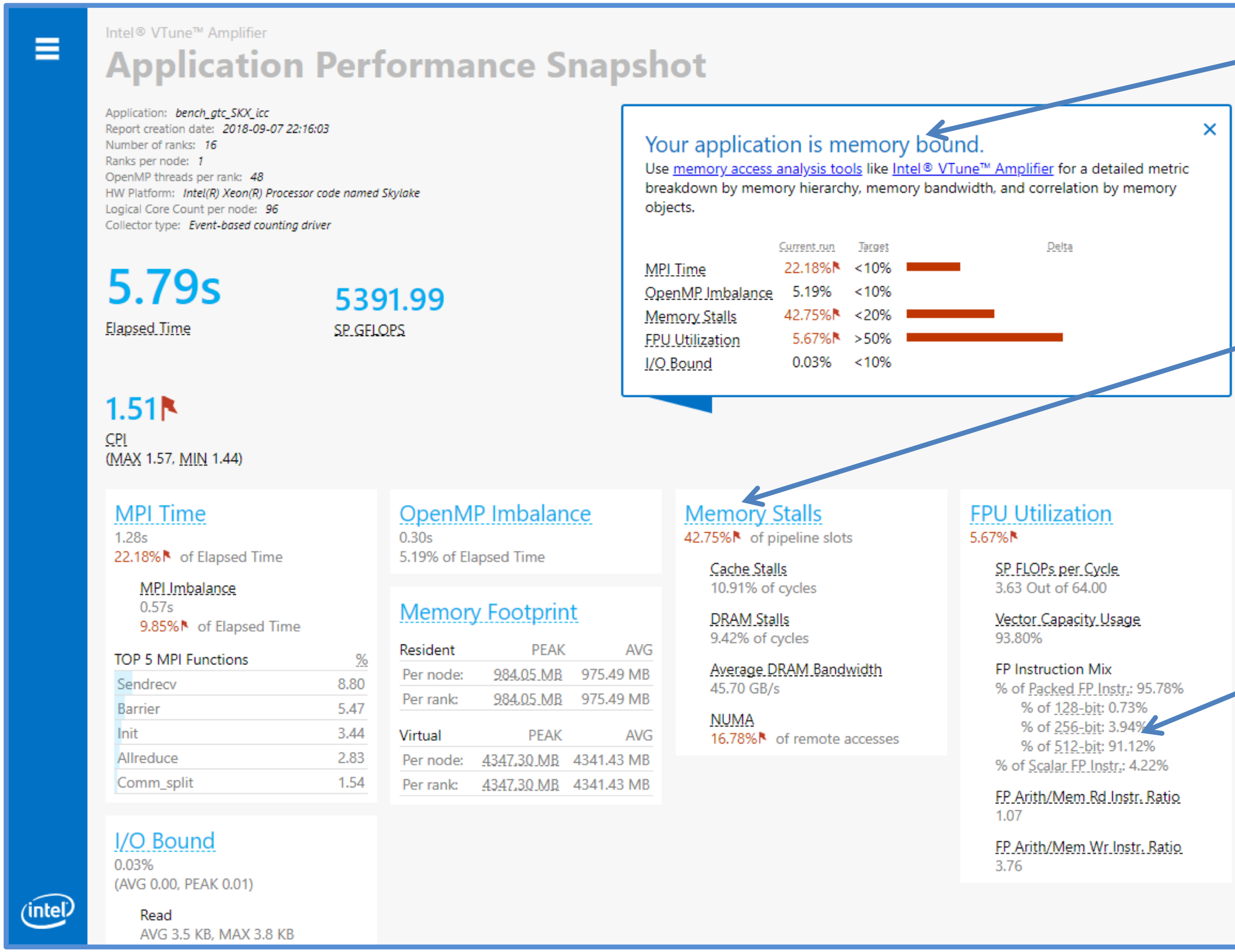
Workflow of Tool Selection



Application Performance Snapshot (APS)

- A **quick view** into a shared memory or MPI application's use of available hardware (CPU, FPU and Memory)
- Identify basic performance optimization opportunities and the **next step** for analysis
- Extremely **easy** to use
- Results shown as HTML format
- **Scales** to large jobs
- Multiple methods to obtain:
  - **Free** download from APS website: https://software.intel.com/sites/products/snapshots/application-snapshot/
  - Part of Intel Parallel Studio XE or Intel VTune Amplifier

Typical APS Report (HTML Based)



Main bottleneck **identified** and next steps **suggested**

Memory Stalls measurement with breakdown by cache and DRAM

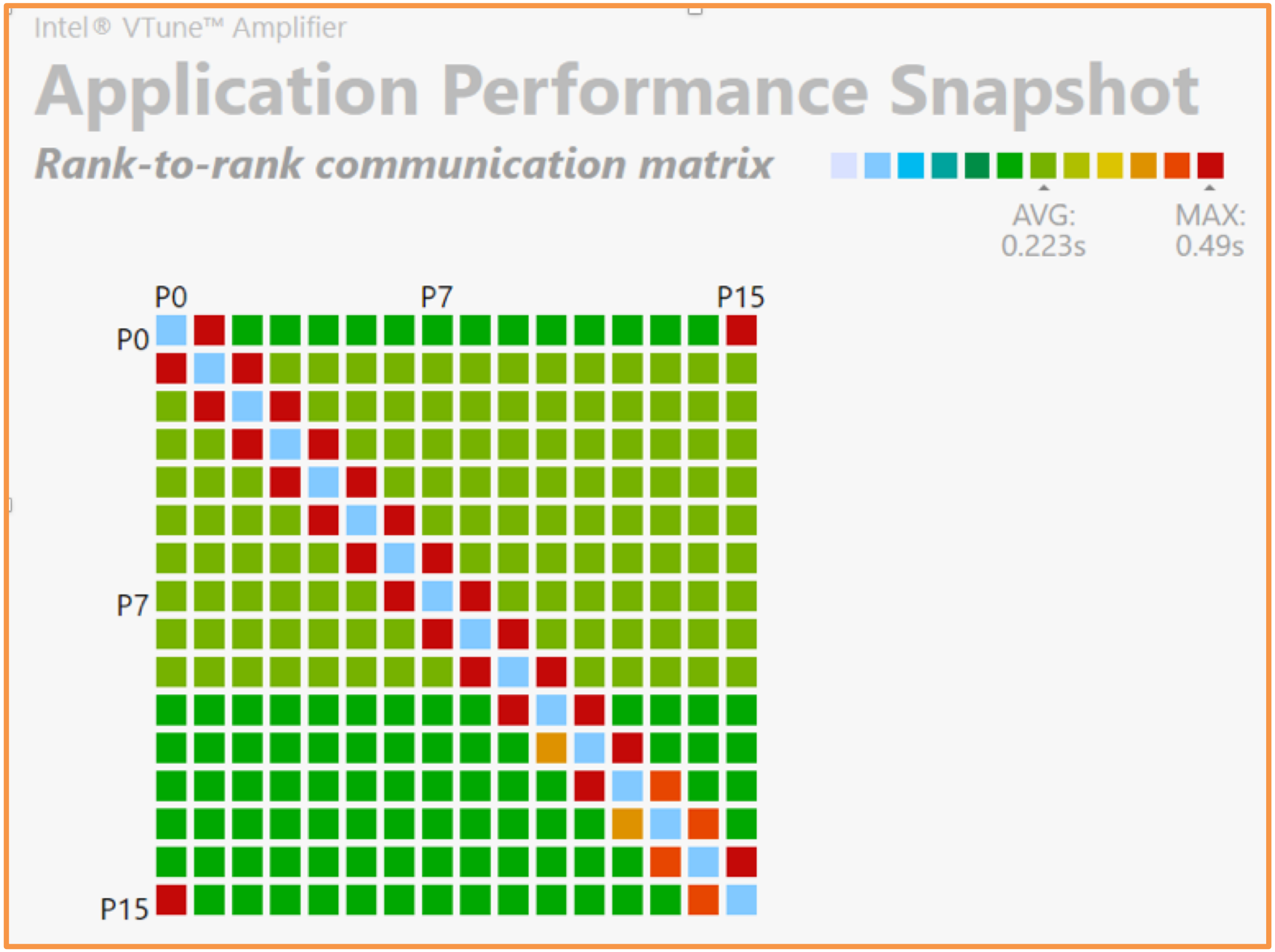
Excellent vectorization

Run the following command to collect the data and complete the analysis:

mpirun –n <N> **aps** ${EXE} ${ARGS}

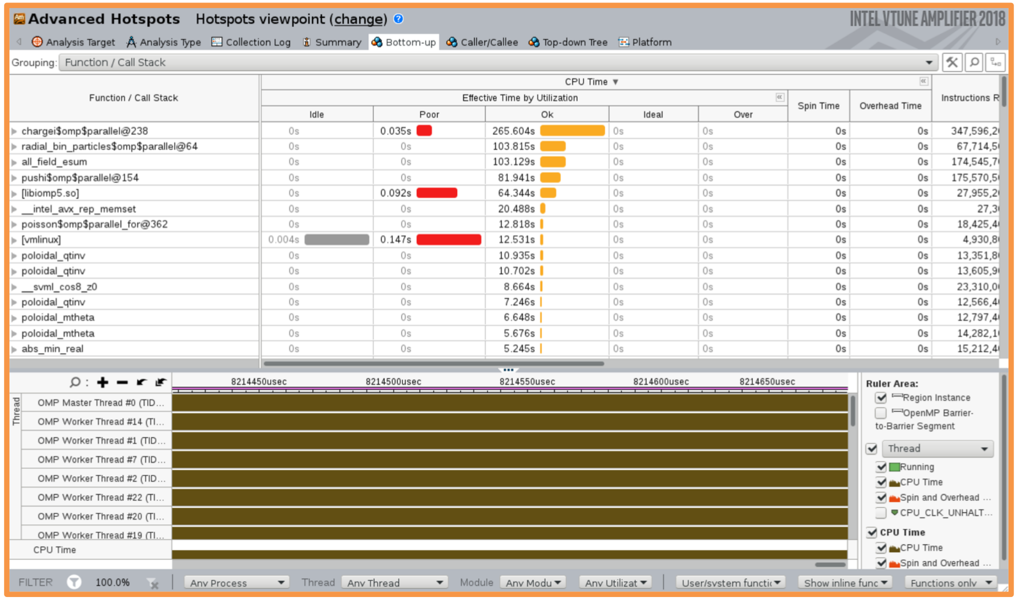
aps -report=${PATH_TO_APS_RESULT_DIR}

Rank to Rank Communication Report



Run the following command to get the report:
aps-report -g ${PATH_TO_APS_RESULT_DIR}

Intel VTune Amplifier



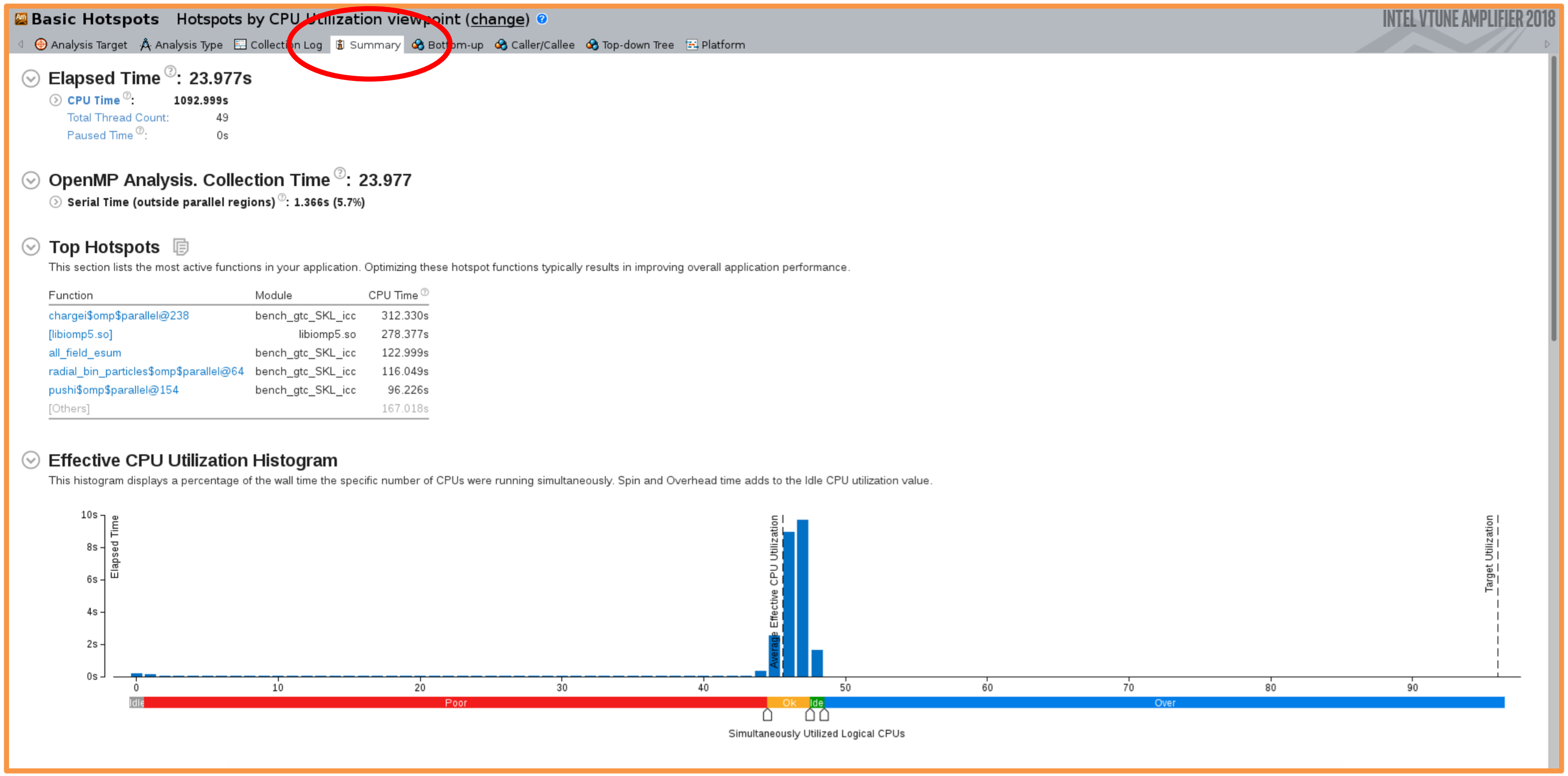
- Accurate data
  - Hotspot
  - Processor microarchitecture
  - Memory access
  - Threading
  - I/O
- Flexible
  - Linux, Windows and Mac OS analysis GUI
  - Link data to source code and assembly
  - Easy set-up, no special compiles
- **Shared memory only**
  - Serial
  - OpenMP
  - MPI on a single node

A Rich Set of Predefined Analysis Types

- **Basic analysis:**
  - hotspots: what functions use most time?
  - concurrency: identify potential parallelization opportunities/issues
- **Advanced analysis**
  - advanced-hotspots: extend the hotspots with call stacks, statistical call counts, CPI metric etc
  - general-exploration: hardware-level performance data
  - hpc-performance: overview of CPU, memory and FPU utilization
  - memory-access: identify memory-related issues
  - …

Hotspots (Summary View)

Use **hotspots analysis** to find where your program is spending the most time, ensuring your optimizations have a bigger impact.



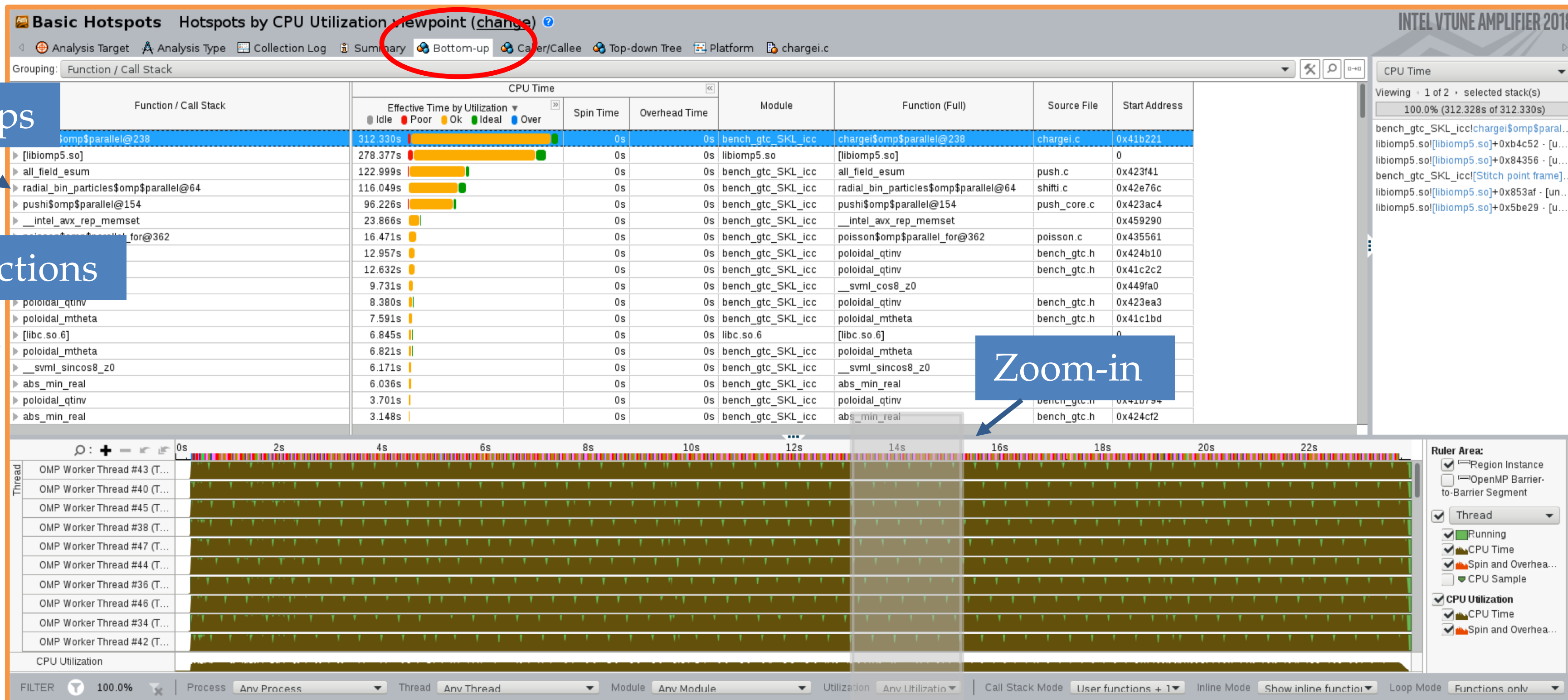
Run the following command to collect the data (remotely) and complete the analysis (locally):

amplxe-cl –collect hotspots –knob analyze-openmp=true -finalization-mode=deferred -- $<EXE> $<ARGS>

amplxe-cl –finalize –r $<RESULT_DIR> –search-dir $<OBJS_DIR> -source-search-dir $<SOURCE_DIR>

Hotspots (Bottom-up View)

Use **bottom-up view** to identify the most time-consuming functions and analyze their call flow from a function to its parent functions



Filter by process, thread & other controls

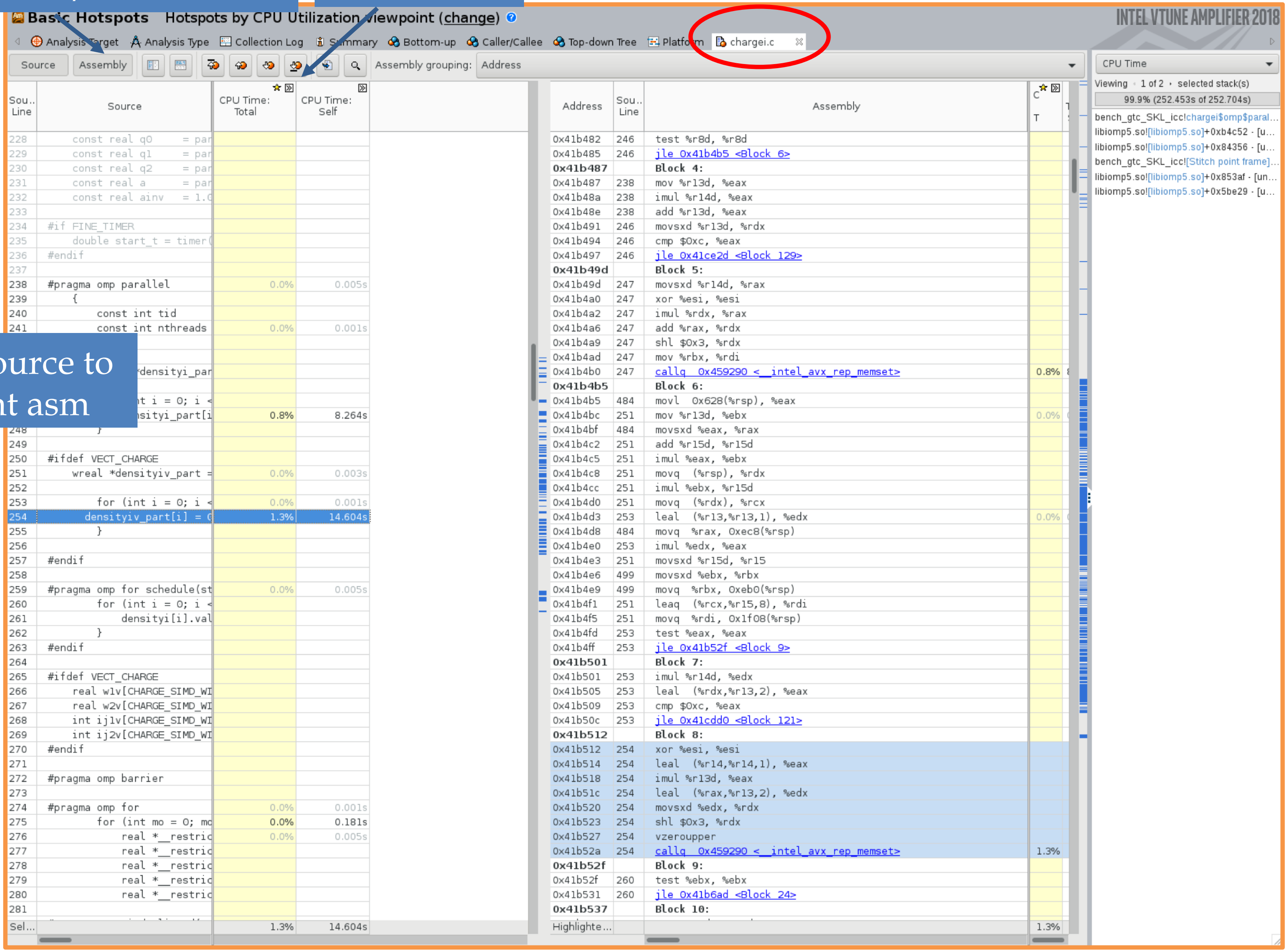
Filter by function or loop

Double Click Function to See Source Line

View source/asm or both

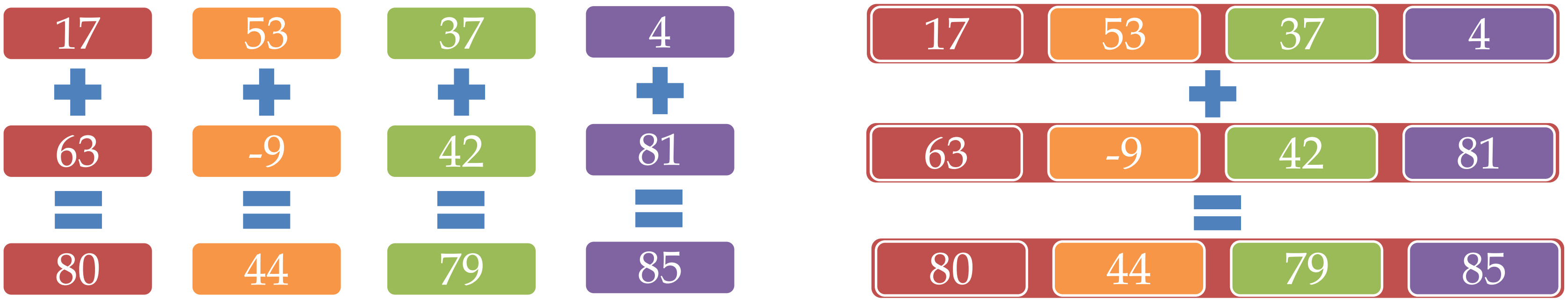
CPU time

Select source to highlight asm

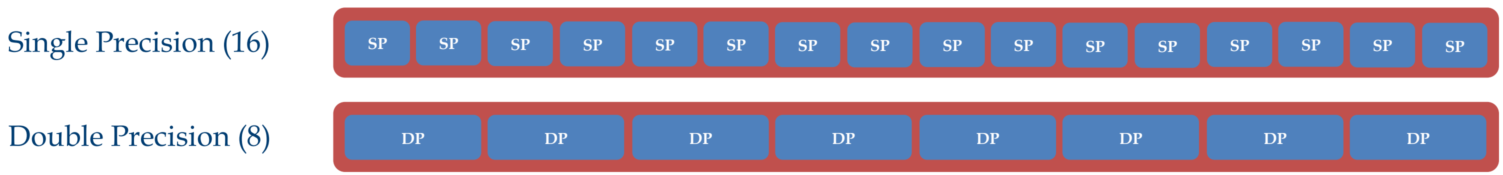


Vectorization 101

Modern computers have vector registers and SIMD (Single Instruction Multiple Data) instructions. This allows one CPU to do multiple calculations at once.



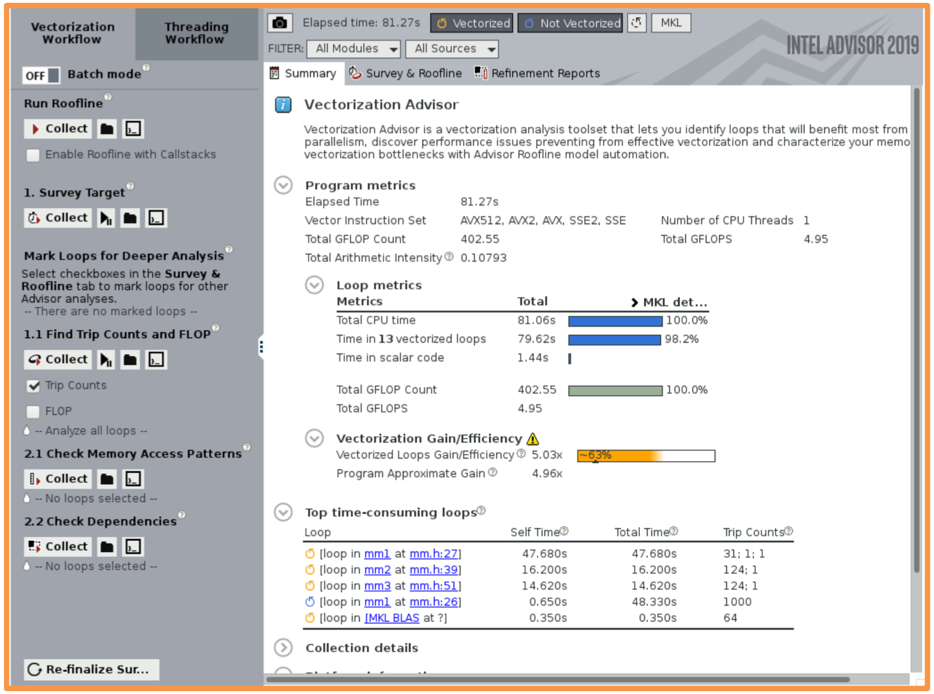
The size of the vector register varies by the architecture. Skylake Server architecture (at Tigercpu of Princeton) has a vector length of 512 bits (8 doubles or 16 floats)



"Expertly tune your application" Intel webinar, Carlos Rosales-Fernandez, 2018

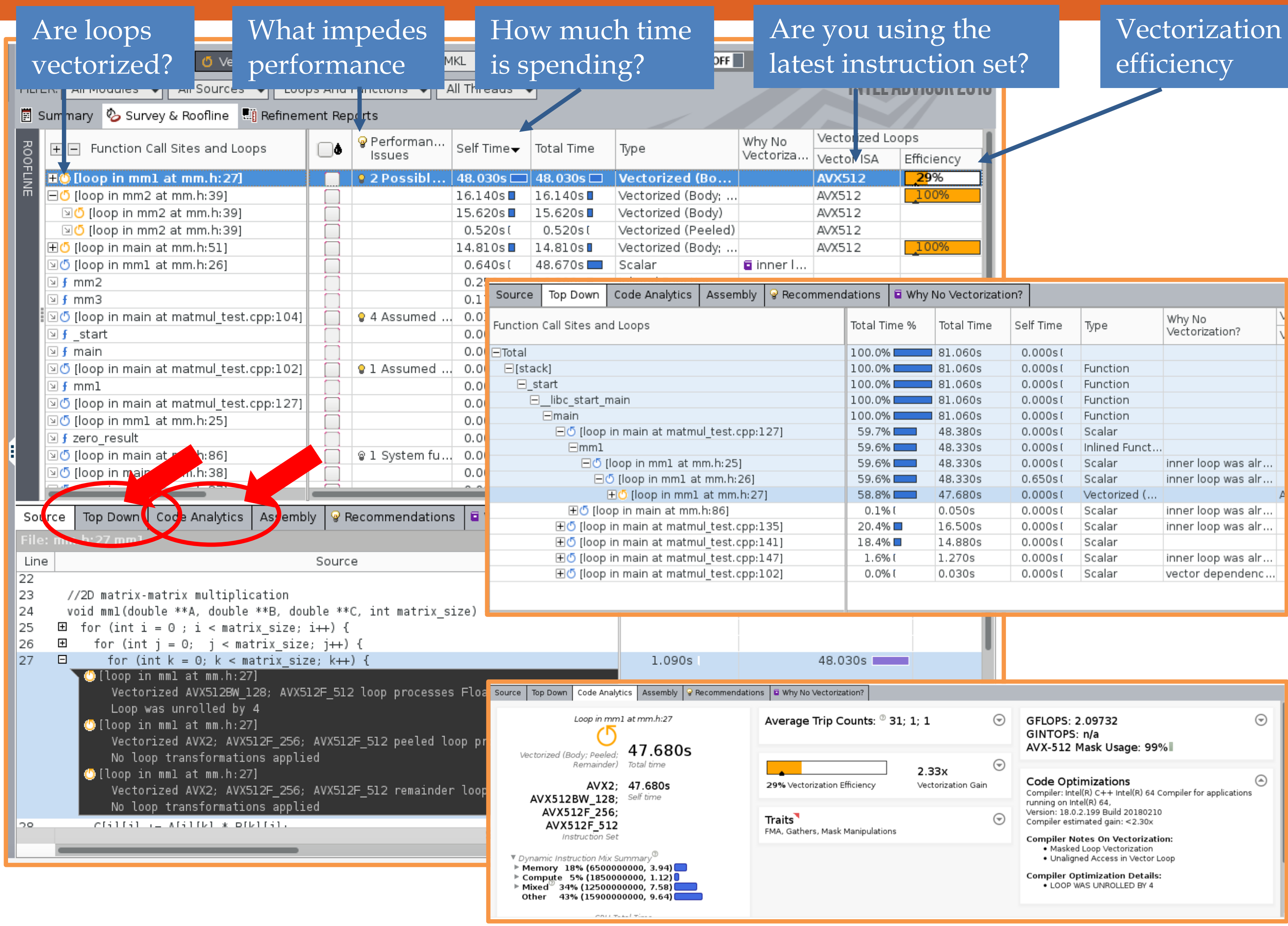
Intel Advisor

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



- Threading Advisor
  - Suitability: predict how well your proposed threading model will scale

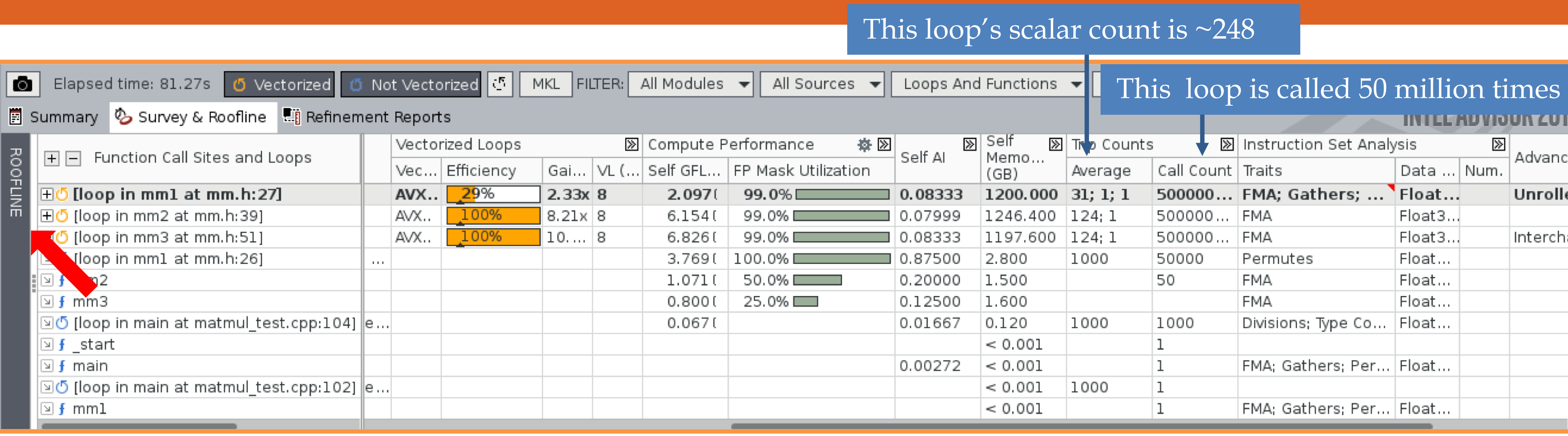
Survey

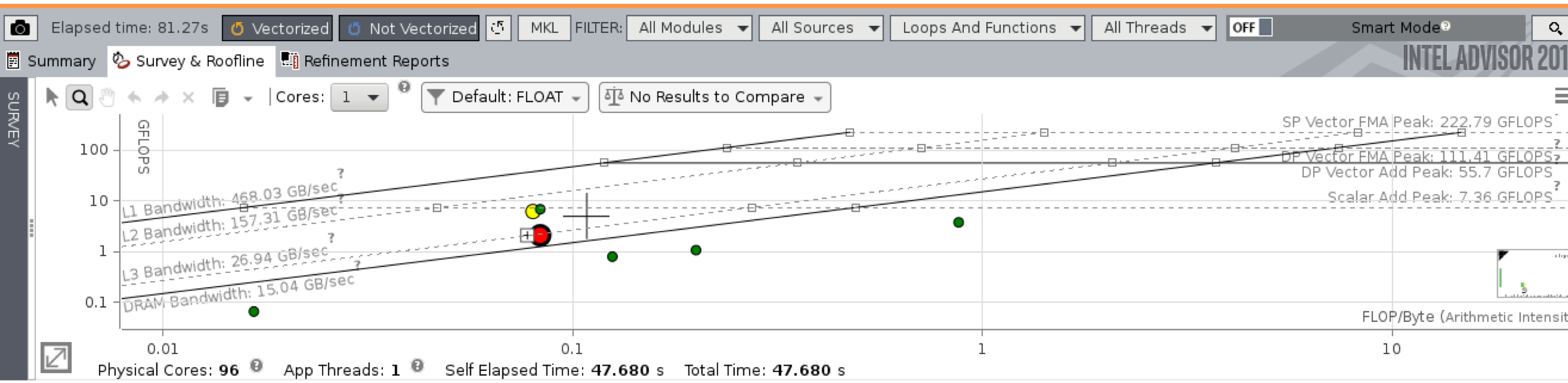


Run the following command to collect the data (remotely):

advixe-cl –c survey -project-dir $<PROJ_DIR> -no-auto-finalize -- $<EXE> $<ARGS>

Trip Counts





Run the following command to collect the data (remotely):

advixe-cl –c tripcounts -flop -project-dir $<PROJ_DIR> -no-auto-finalize -- $<EXE> $<ARGS>

Note: it is important to use the same project directory as the survey analysis

References

- **Optimizing HPC Applications with Intel® Cluster Tools,** Alexander Supalov; Andrey Semin; Michael Klemm; Christopher Dahnken, Apress, 2014
- https://software.intel.com/en-us/application-snapshot-user-guide
- **https://software.intel.com/en-us/vtune-amplifier-cookbook**
- **https://software.intel.com/en-us/advisor/documentation/view-all**

Hands-on

- Goal: Identify hotspots in sample code
  - Targets for optimization
- Test code has 4 functions: mm[1-4]
  - Each does a different version of matrix-matrix multiplication $C=A\times B$
- Each function is called 50 times
  - Where should we optimize?

Adroit Test Set Up

- Enable X11 forwarding
  - "ssh –Y -C <user>@adroit.princeton.edu
  - Will need local xserver (XQuartz for OSX, Xming for Windows)
- Clone the repo

git clone https://github.com/beiwang2003/Bootcamp2018-Perf-Tuning.git

- Follow instructions in repo Readme.md

- What functions are most/least expensive?

- ※What are the vectorization efficiency of each loop?

※ if you have extra time