

# 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](mailto:beiwang@princeton.edu)

# Outline

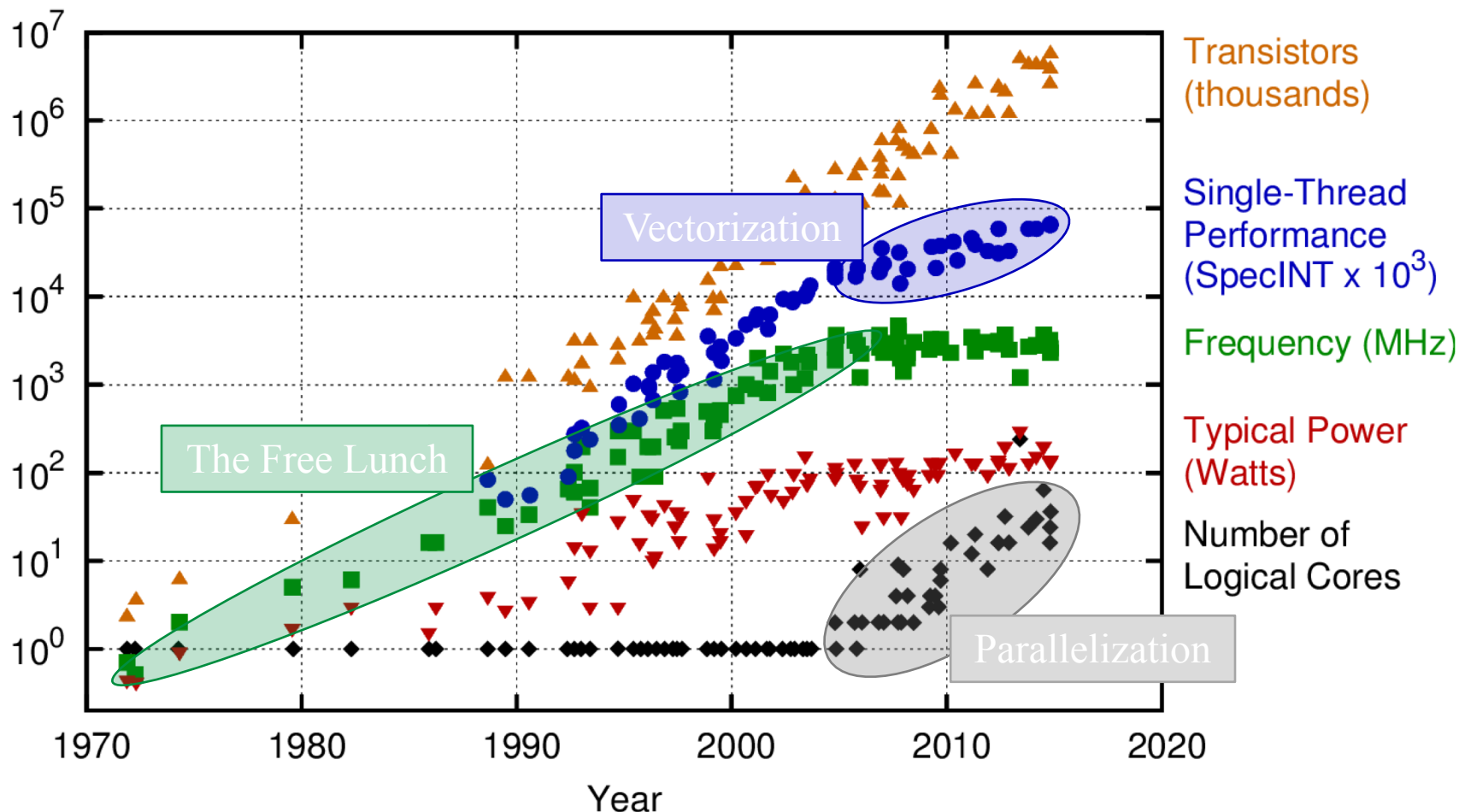
- An introduction to the **idea of performance analysis**
  - Methodology
  - Workflow
  - Measurement tools
  - Hands-on
- Focused primarily on the **HPC recourses at Princeton**
  - Hardware: Intel CPU
  - Tools: Intel performance tuning tools
  - Scientific application codes written with C/C++ and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

# “The Free Lunch is Over”

40 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

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

# Performance Analysis Methodology:

## A top-down approach

**SYSTEM**  
(hardware and system software)

Hard disk  
Network interface  
Memory  
BIOS  
Operation system

**APPLICATION**

Algorithm  
Data structure  
Parallelization

← Optimization

**MICRO-ARCHITECTURE**

Instructions  
Cache/Memory  
SIMD  
Branch prediction

← Tuning

*"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

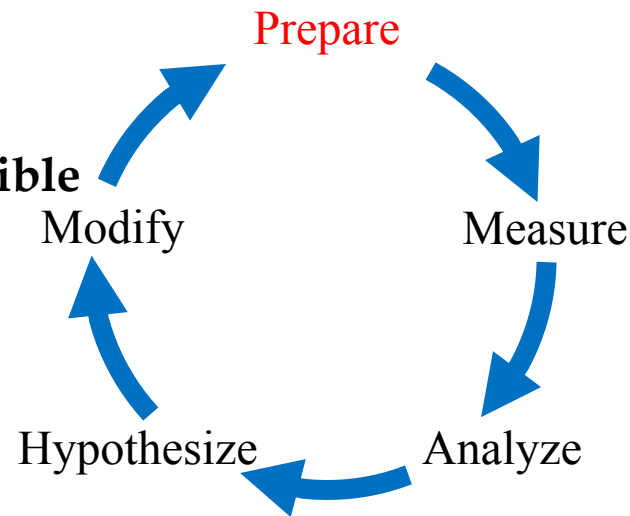
```
GIT_VERSION:=$(shell sh -c './GIT-VERSION-GEN')
COMPILER_VERSION:="$(CC)-$(shell $(CC) --version | head -n1 | cut -d' ' -f4)"
BUILD_HOST=$(shell sh -c './BUILD-HOST-GEN')

# 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)"
```

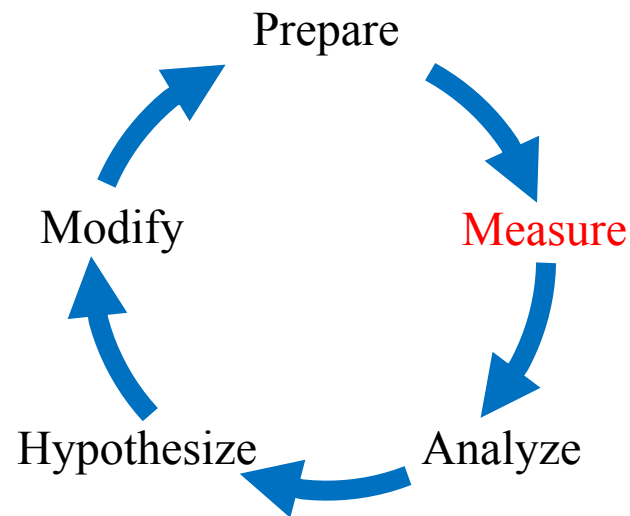
- Basic variants: thread count, affinity, working set size



# 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

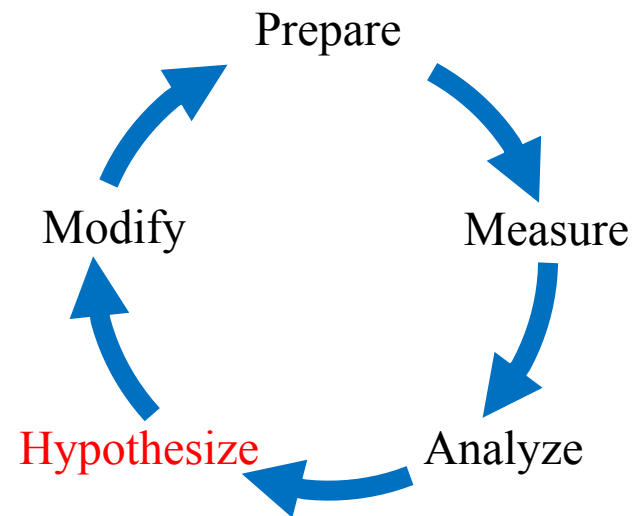




# Performance Analysis Workflow:

## Hypothesize

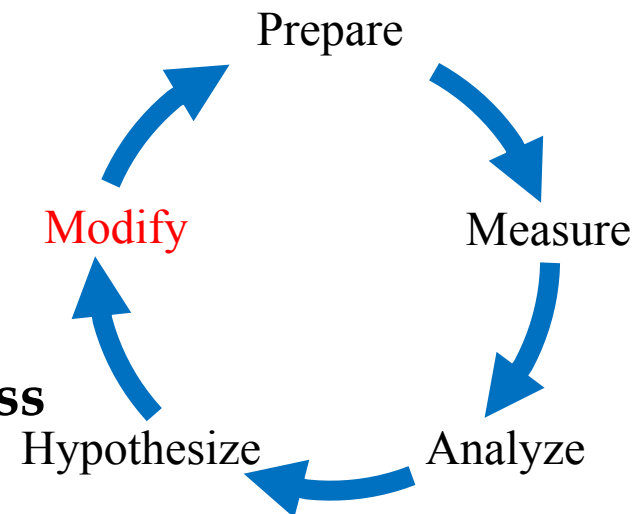
- Why is my code slow?
  - CPU bound
  - Memory bound
  - I/O bound
  - Network bound
  - Unbalanced Workload (Parallel)
- What is the best I can expect?
  - CPU
  - Memory/Cache
  - I/O
  - Network
  - Parallel Scaling



# Performance Analysis Workflow:

## Modify

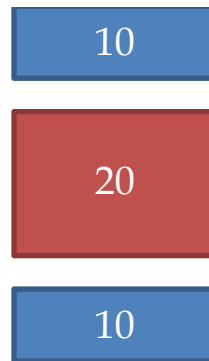
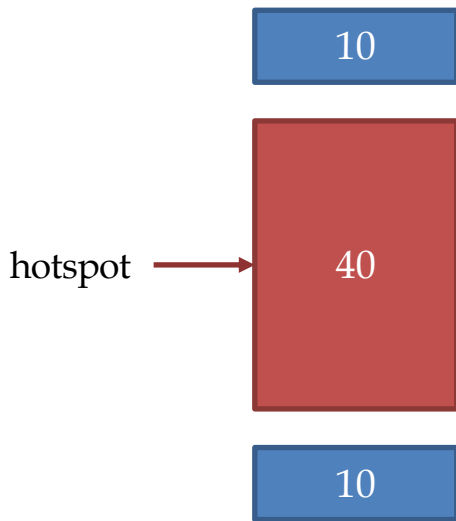
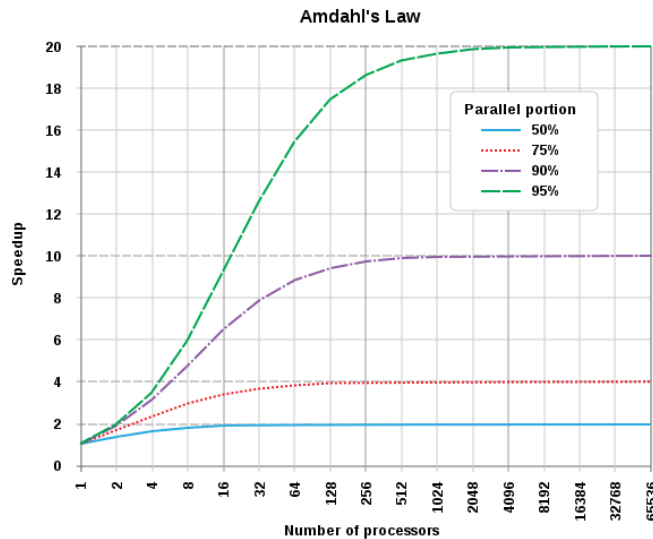
- Change only **one thing at a time**
- Consider the ease (difficulty) of implementation
- Keep **track** of all **changes**
- Apply regression test to **ensure correctness** after each change



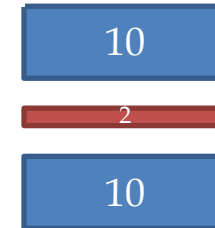
Words to Remember:

Fast computing of wrong result is completely irrelevant!

# When to Stop? Amdahl's law



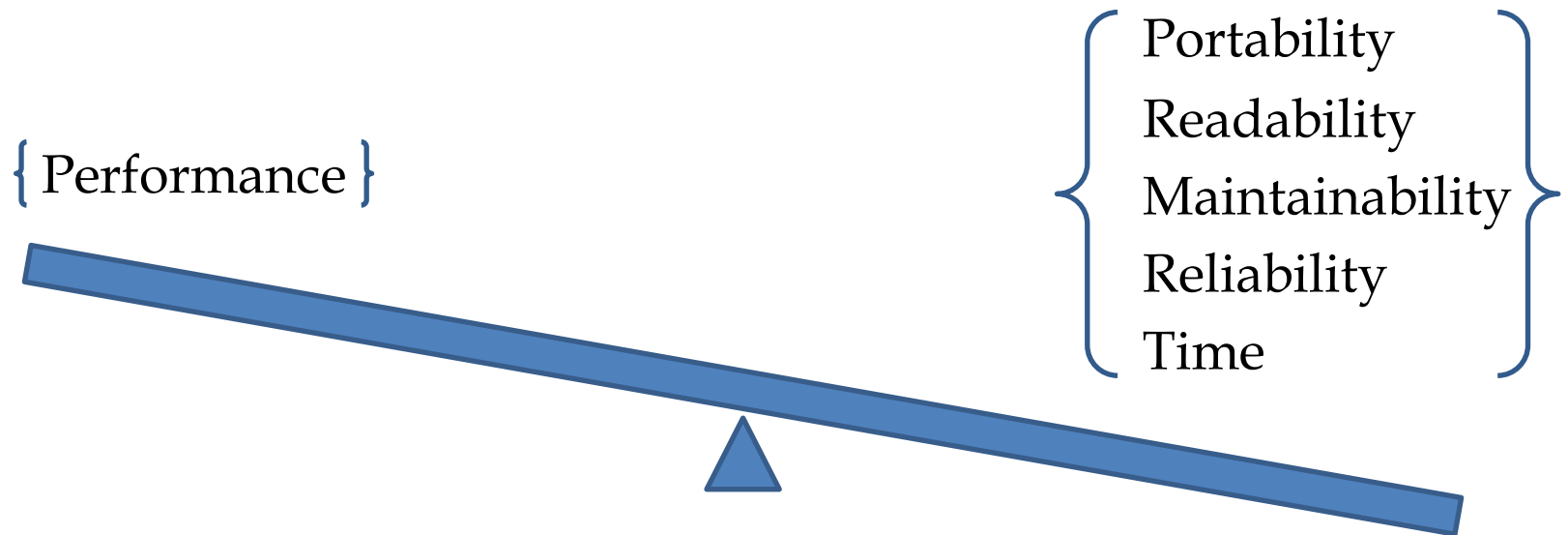
2x speed up in hotspot  
1.5x speed up overall



20x speed up in hotspot  
2.73x speed up overall

[https://en.wikipedia.org/wiki/Amdahl%27s\\_law](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](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}
```

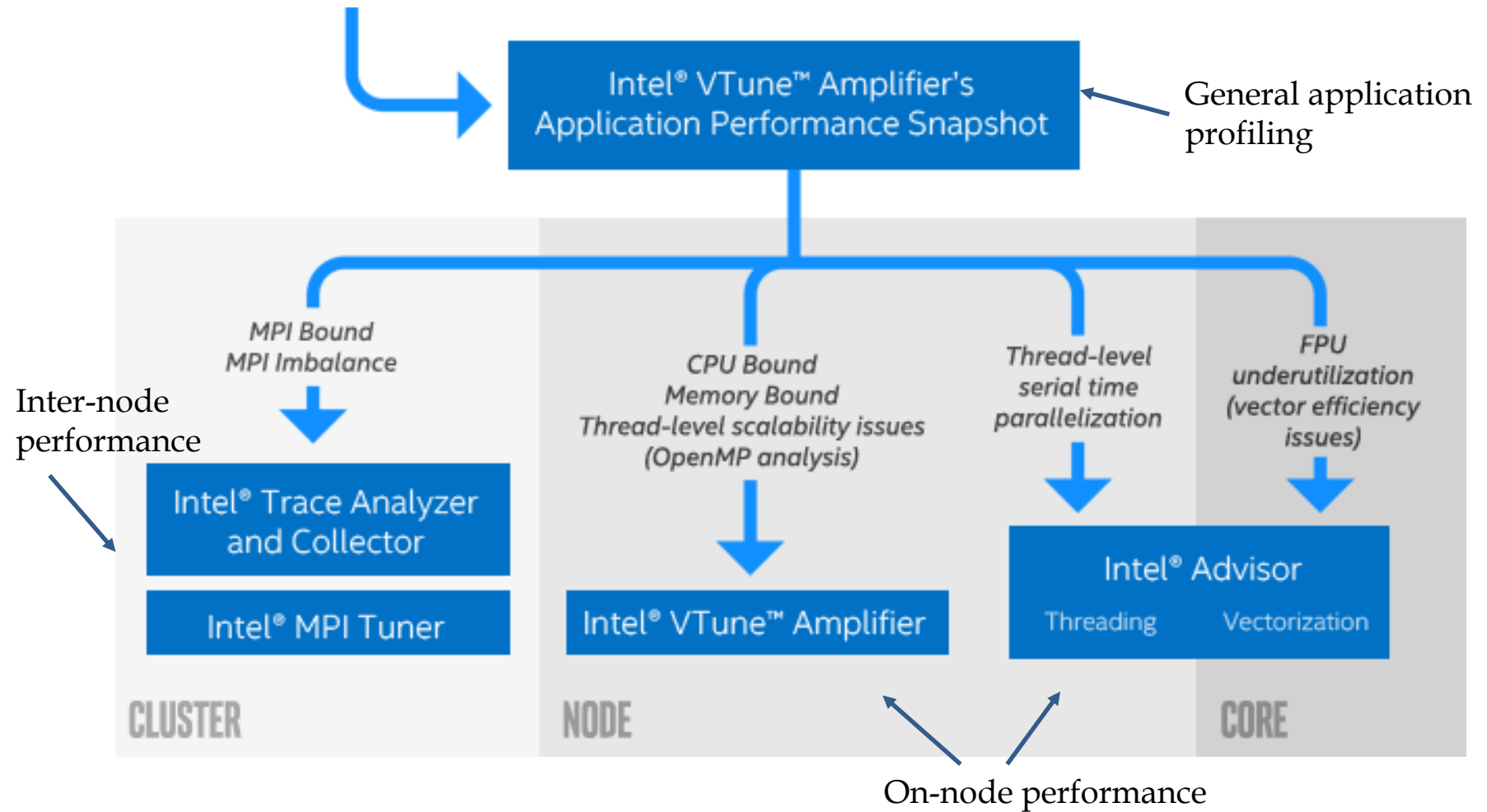
```
mpirun -n 1 advixe-cl -c tripcounts -flop -project-dir -- ${EXE} ${ARGS}: -n <N-1> ${EXE}  
${ARGS}
```

- **Intel Trace Analyzer and Collector**

- At scale MPI performance analyzer



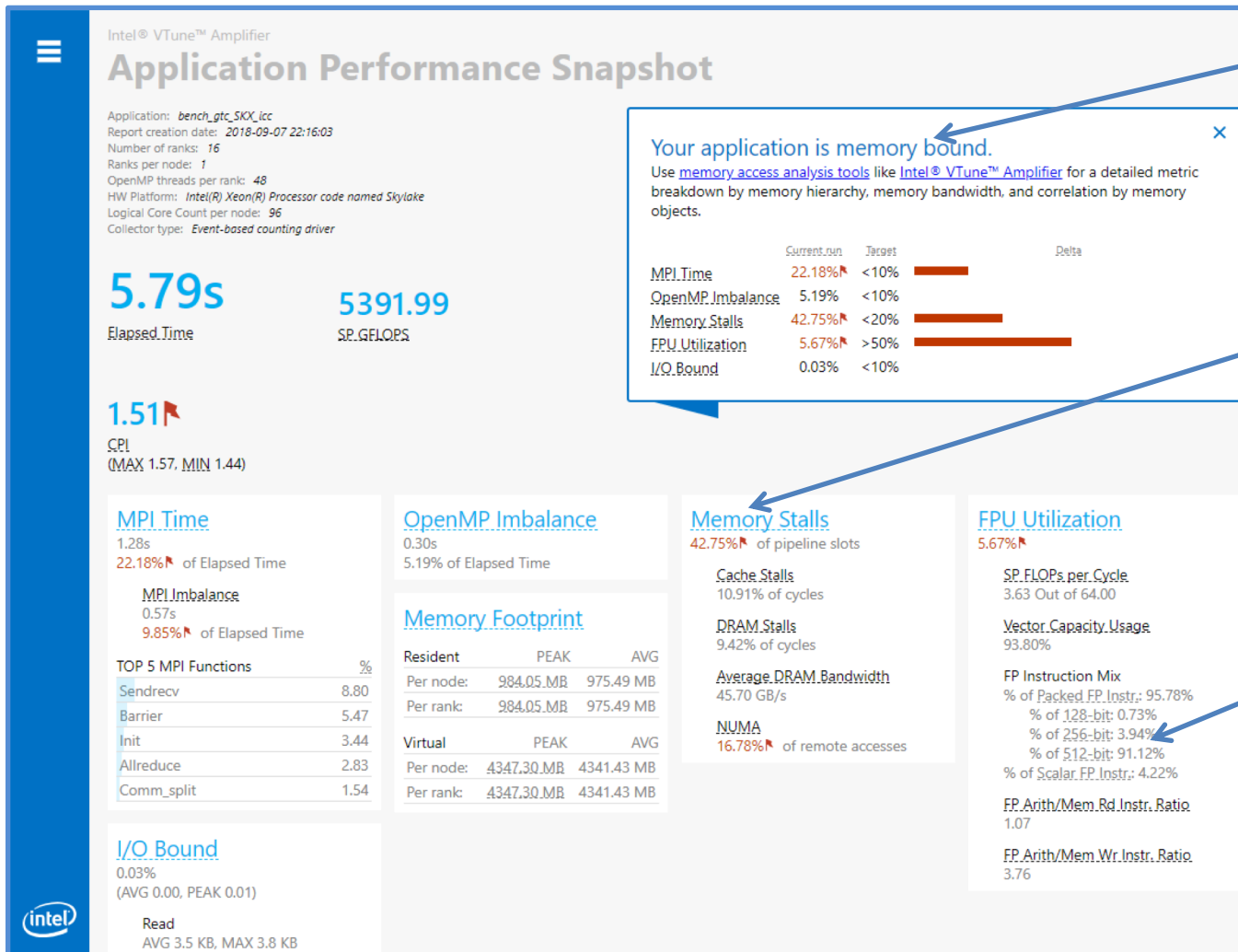
# Workflow of Tool Selection



# 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)



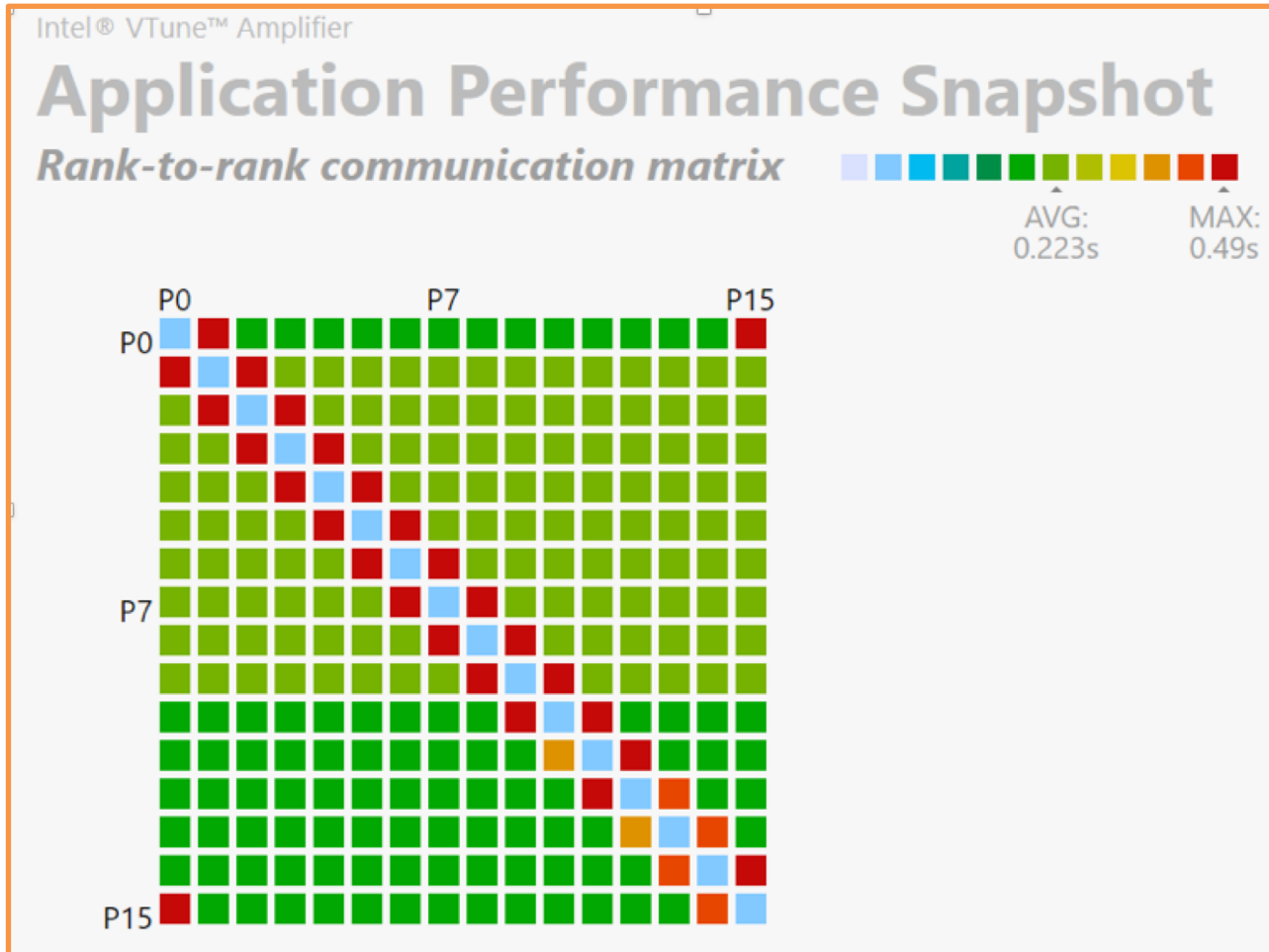
Main bottleneck identified and next steps suggested

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

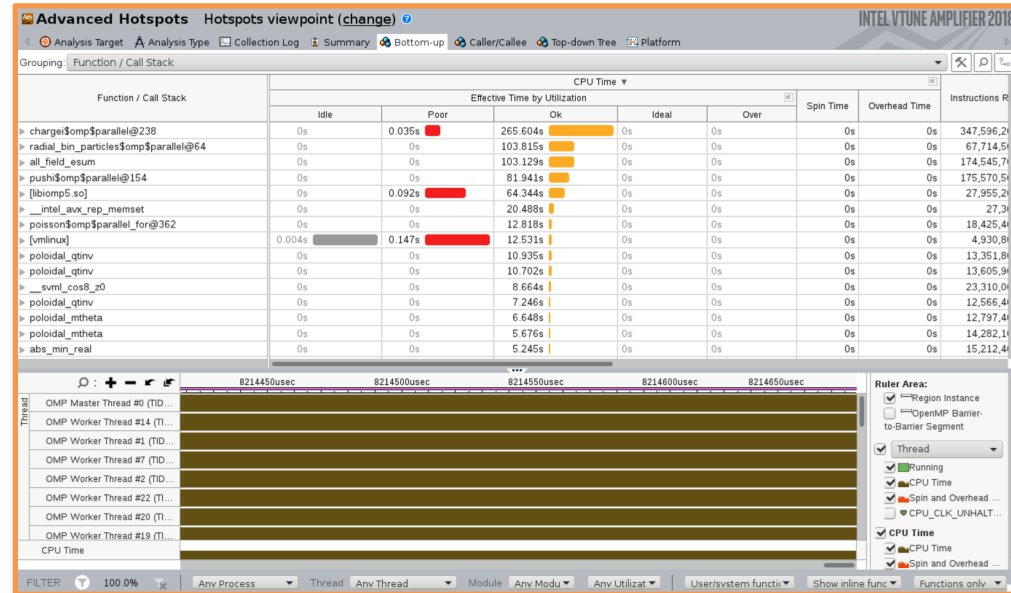


Run the following command to get the report:

```
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

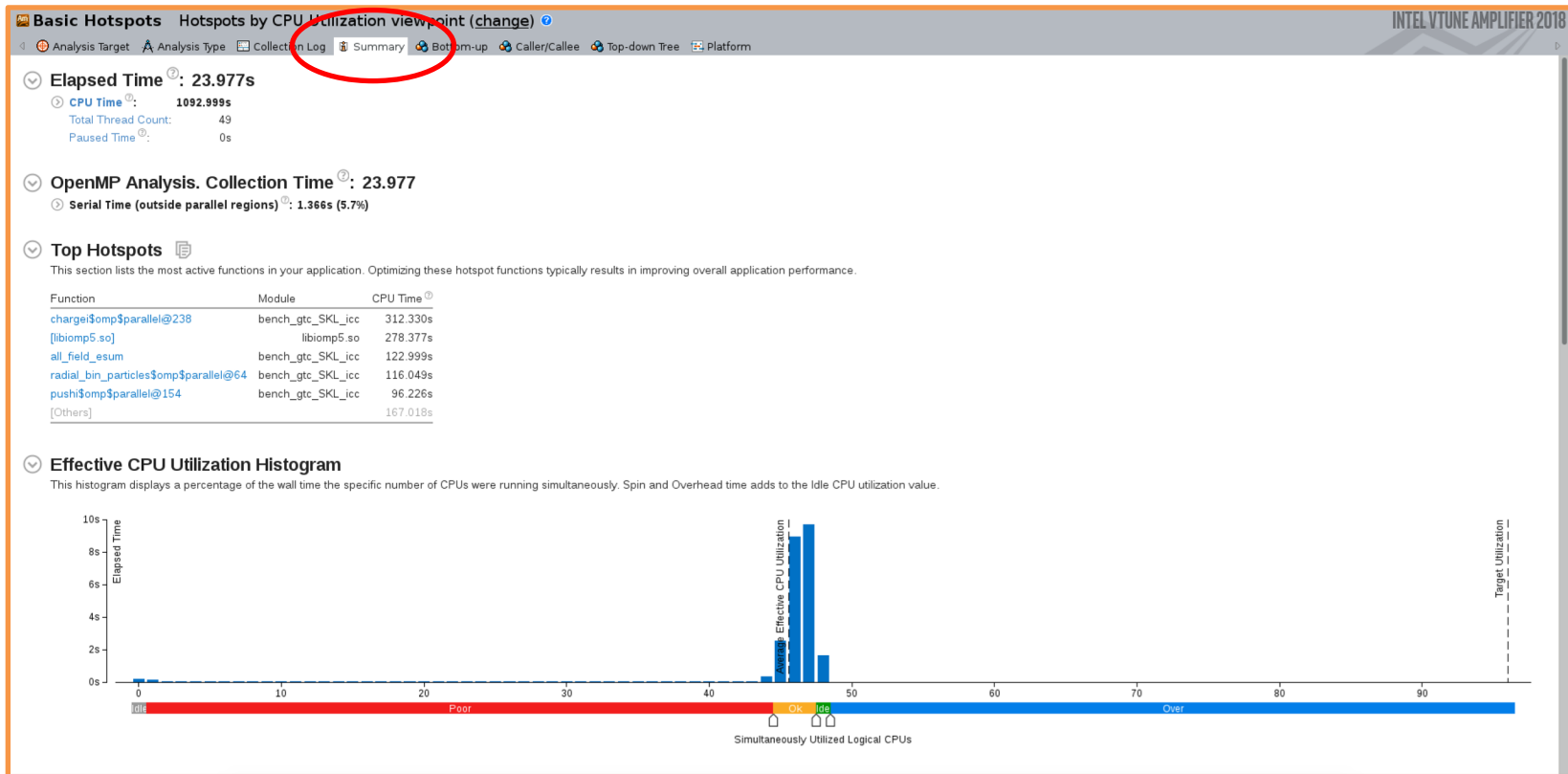


# A Rich Set of Predefined Analysis Types

- **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)

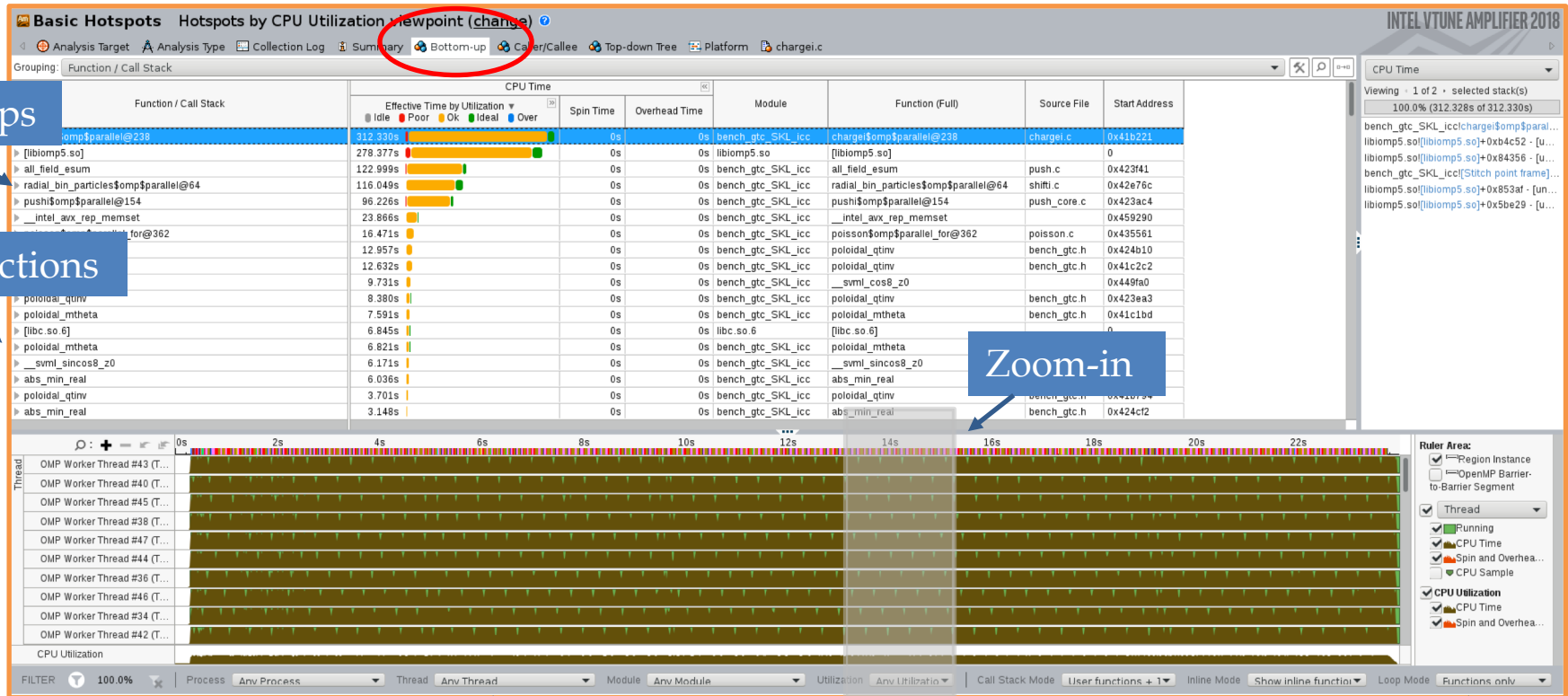
Use **hotspots analysis** to find where your program is spending the most time, ensuring your optimizations have a bigger impact.



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

Basic Hotspots Hotspots by CPU Utilization viewpoint (change)

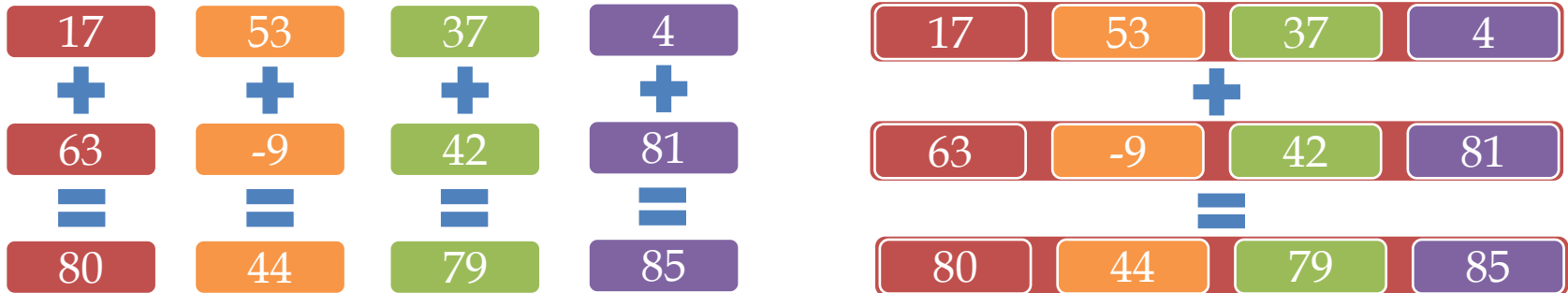
Analysis Target Analysis Type Collection Log Summary Bottom-up Caller/Callee Top-down Tree Platform charge.c

Source	Assembly
228 const real q0 = par	0x41b482 246 test %r8d, %r8d
229 const real q1 = par	0x41b485 246 jle 0x41b4b5 <Block 6>
230 const real q2 = par	0x41b487 246 Block 4:
231 const real a = par	0x41b487 238 mov %r13d, %eax
232 const real ainvs = 1.0	0x41b48a 238 imul %r14d, %eax
233	0x41b48e 238 add %r13d, %eax
234 #if FINE_TIMER	0x41b491 246 movsxd %r13d, %rdx
235 double start_t = timer	0x41b494 246 cmp \$0xc, %eax
236 #endif	0x41b497 246 jle 0x41ce2d <Block 129>
237	0x41b49d 246 Block 5:
238 #pragma omp parallel	0x41b49d 247 movsxd %r14d, %rax
239 {	0x41b4a0 247 xor %esi, %esi
240 const int tid	0x41b4a2 247 imul %rdx, %rax
241 const int nthreads	0x41b4a6 247 add %rax, %rdx
	0x41b4a9 247 shl \$0x3, %rdx
	0x41b4ad 247 mov %rbx, %rdi
	0x41b4b0 247 callq 0x459290 <_intel_avx_rep_memset>
	0x41b4b5 247 Block 6:
	0x41b4b5 484 movl 0x628(%rsp), %eax
	0x41b4bc 251 mov %r13d, %ebx
	0x41b4bf 484 movsxd %eax, %rax
	0x41b4c2 251 add %r15d, %r15d
	0x41b4c5 251 imul %eax, %ebx
	0x41b4c8 251 movq (%rsp), %rdx
	0x41b4cc 251 imul %ebx, %r15d
	0x41b4d0 251 movq (%rdx), %rcx
	0x41b4d3 253 leal (%r13,%r13,1), %edx
	0x41b4d8 484 movq %rax, 0xec8(%rsp)
	0x41b4e0 253 imul %edx, %eax
	0x41b4e3 251 movsxd %r15d, %r15
	0x41b4e6 499 movsxd %ebx, %rbx
	0x41b4e9 499 movq %rbx, 0xeb0(%rsp)
	0x41b4f1 251 leaq (%rcx,%r15,8), %rdi
	0x41b4f5 251 movq %rdi, 0x1f08(%rsp)
	0x41b4fd 253 test %eax, %eax
	0x41b4ff 253 jle 0x41b52f <Block 9>
	0x41b501 253 Block 7:
	0x41b501 253 imul %r14d, %edx
	0x41b505 253 leal (%rdx,%r13,2), %eax
	0x41b509 253 cmp \$0xc, %eax
	0x41b50c 253 jle 0x41cdd0 <Block 121>
	0x41b512 253 Block 8:
	0x41b512 254 xor %esi, %esi
	0x41b514 254 leal (%r14,%r14,1), %eax
	0x41b518 254 imul %r13d, %eax
	0x41b51c 254 leal (%rax,%r13,2), %edx
	0x41b520 254 movsxd %edx, %rdx
	0x41b523 254 shl \$0x3, %rdx
	0x41b527 254 vzeroupper
	0x41b52a 254 callq 0x459290 <_intel_avx_rep_memset>
	0x41b52f 254 Block 9:
	0x41b52f 260 test %ebx, %ebx
	0x41b531 260 jle 0x41b6ad <Block 24>
	0x41b537 260 Block 10:
	Highlighte... 1.3%

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)

Single Precision (16)



Double Precision (8)



*"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

The screenshot displays the Intel Advisor 2019 interface. The left sidebar shows the 'Vectorization Workflow' with options for 'Run Roofline', '1. Survey Target', 'Mark Loops for Deeper Analysis', '1.1 Find Trip Counts and FLOP', '2.1 Check Memory Access Patterns', and '2.2 Check Dependencies'. The main panel shows the 'Vectorization Advisor' results for a program with an elapsed time of 81.27s. Key metrics include: Total CPU time (81.06s), Time in 13 vectorized loops (79.62s, 98.2%), Time in scalar code (1.44s), Total GFLOP Count (402.55), and Total GFLOPS (4.95). A 'Loop metrics' table is shown below, and a 'Top time-consuming loops' table lists the most significant loops.

Loop	Self Time	Total Time	Trip Counts
[loop in mm1 at mm.b:27]	47.680s	47.680s	31; 1; 1
[loop in mm2 at mm.b:39]	16.200s	16.200s	124; 1
[loop in mm3 at mm.b:51]	14.620s	14.620s	124; 1
[loop in mm1 at mm.b:26]	0.650s	48.330s	1000
[loop in MKL_BLAS at ?]	0.350s	0.350s	64

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

# Survey

Are loops vectorized?      What impedes performance      How much time is spending?      Are you using the latest instruction set?      Vectorization efficiency

Function Call Sites and Loops	Self Time	Total Time	Type	Why No Vectoriza...	Vectorized Loops
[loop in mm1 at mm.h:27]	48.030s	48.030s	Vectorized (Bo...		AVX512 29%
[loop in mm2 at mm.h:39]	16.140s	16.140s	Vectorized (Body; ...		AVX512 100%
[loop in mm2 at mm.h:39]	15.620s	15.620s	Vectorized (Body)		AVX512
[loop in mm2 at mm.h:39]	0.520s	0.520s	Vectorized (Peeled)		AVX512
[loop in main at mm.h:51]	14.810s	14.810s	Vectorized (Body; ...		AVX512 100%
[loop in mm1 at mm.h:26]	0.640s	48.670s	Scalar	inner l...	

Source	Top Down	Code Analytics	Assembly	Recommendations	Why No Vectorization?
Function Call Sites and Loops					
Total	100.0%	81.060s	0.000s		
[stack]	100.0%	81.060s	0.000s		
_start	100.0%	81.060s	0.000s		
_libc_start_main	100.0%	81.060s	0.000s		
main	100.0%	81.060s	0.000s		
[loop in main at matmul_test.cpp:127]	59.7%	48.380s	0.000s		
mm1	59.6%	48.330s	0.000s		Inlined Funct...
[loop in mm1 at mm.h:25]	59.6%	48.330s	0.000s		inner loop was alr...
[loop in mm1 at mm.h:26]	59.6%	48.330s	0.650s		inner loop was alr...
[loop in mm1 at mm.h:27]	58.8%	47.680s	0.000s		Vectorized (...
[loop in main at mm.h:86]	0.1%	0.050s	0.000s		inner loop was alr...
[loop in main at matmul_test.cpp:135]	20.4%	16.500s	0.000s		inner loop was alr...
[loop in main at matmul_test.cpp:141]	18.4%	14.880s	0.000s		Scalar
[loop in main at matmul_test.cpp:147]	1.6%	1.270s	0.000s		inner loop was alr...
[loop in main at matmul_test.cpp:102]	0.0%	0.030s	0.000s		vector dependenc...

```
23 //2D matrix-matrix multiplication
24 void mml(double **A, double **B, double **C, int matrix_size)
25 {
26     for (int i = 0; i < matrix_size; i++) {
27         for (int j = 0; j < matrix_size; j++) {
28             for (int k = 0; k < matrix_size; k++) {
```

[loop in mm1 at mm.h:27]  
Vectorized AVX512BW\_128; AVX512F\_512 loop processes Flo...  
Loop was unrolled by 4

[loop in mm1 at mm.h:27]  
Vectorized AVX2; AVX512F\_256; AVX512F\_512 peeled loop pr...  
No loop transformations applied

[loop in mm1 at mm.h:27]  
Vectorized AVX2; AVX512F\_256; AVX512F\_512 remainder loop...  
No loop transformations applied

```
        C[i][j] += A[i][k] * B[k][j];
```

Dynamic Instruction Mix Summary

- Memory 18% (6500000000, 3.94)
- Compute 5% (1850000000, 1.12)
- Mixed 34% (12500000000, 7.58)
- Other 43% (15900000000, 9.64)

Average Trip Counts: 31; 1; 1

47.680s Total time

AVX2; 47.680s Self time

AVX512BW\_128;  
AVX512F\_256;  
AVX512F\_512  
Instruction Set

2.33x Vectorization Gain

29% Vectorization Efficiency

GFLOPS: 2.09732  
GINTOPS: n/a  
AVX-512 Mask Usage: 99%

Code Optimizations  
Compiler: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64, Version: 18.0.2.199 Build 20180210  
Compiler estimated gain: <2.30x

Compiler Notes On Vectorization:  
• Masked Loop Vectorization  
• Unaligned Access in Vector Loop

Compiler Optimization Details:  
• LOOP WAS UNROLLED BY 4

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

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

# Trip Counts

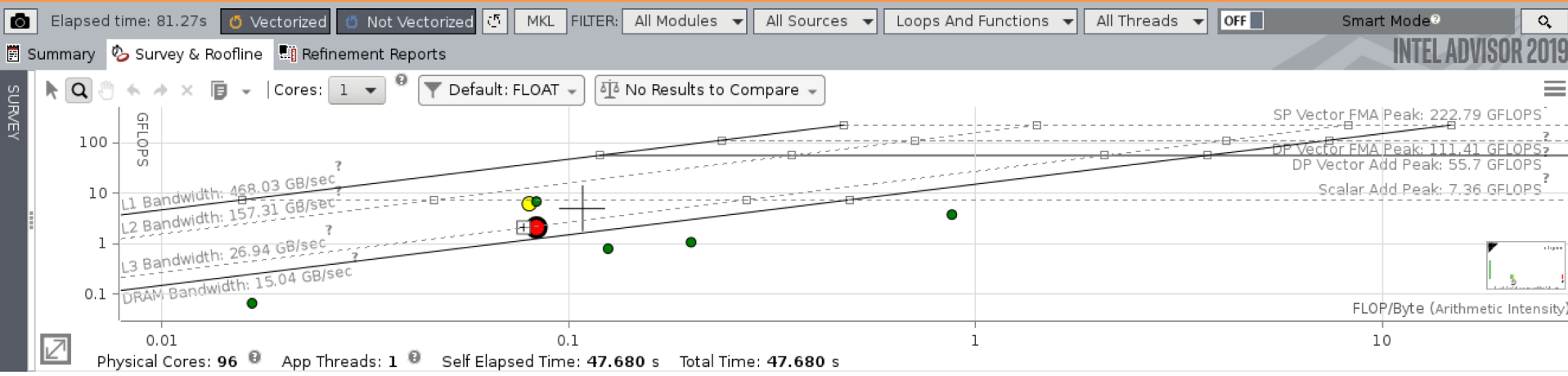
This loop's scalar count is ~248

This loop is called 50 million times

Elapsed time: 81.27s Vectorized Not Vectorized MKL FILTER: All Modules All Sources Loops And Functions

Summary Survey & Roofline Refinement Reports

Function Call Sites and Loops	Vectorized Loops	Compute Performance	Self AI	Self Memo... (GB)	Trip Counts	Instruction Set Analysis
	Vec... Efficiency Gai... VL (...)	Self GFL... FP Mask Utilization			Average Call Count	Traits Data ... Num.
[loop in mm1 at mm.h:27]	AVX.. 29% 2.33x 8	2.097( 99.0%	0.08333	1200.000	31; 1; 1 500000...	FMA; Gathers; ... Float...
[loop in mm2 at mm.h:39]	AVX.. 100% 8.21x 8	6.154( 99.0%	0.07999	1246.400	124; 1 500000...	FMA Float3...
[loop in mm3 at mm.h:51]	AVX.. 100% 10.... 8	6.826( 99.0%	0.08333	1197.600	124; 1 500000...	FMA Float3...
[loop in mm1 at mm.h:26]	...	3.769( 100.0%	0.87500	2.800	1000 50000	Permutes Float...
f mm2		1.071( 50.0%	0.20000	1.500	50	FMA Float...
f mm3		0.800( 25.0%	0.12500	1.600		FMA Float...
[loop in main at matmul_test.cpp:104]	e...	0.067(	0.01667	0.120	1000 1000	Divisions; Type Co... Float...
f _start				< 0.001	1	
f main			0.00272	< 0.001	1	FMA; Gathers; Per... Float...
[loop in main at matmul_test.cpp:102]	e...			< 0.001	1000 1	
f mm1				< 0.001	1	FMA; Gathers; Per... Float...



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