BERKELEY LAB

BERKELEY LAB
LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF
ENERGY

Using the Roofline Model and Intel Advisor

Samuel Williams
SWWilliams@lbl.gov
Computational Research Division
Lawrence Berkeley National Lab

Tuomas Koskela
TKoskela@lbl.gov
NERSC
Lawrence Berkeley National Lab

Acknowledgements

- This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.
- This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

- Special Thanks to:
  - Zakhar Matveev, Intel Corporation
  - Roman Belenov, Intel Corporation

Introduction

Performance Models and Tools

- Identify performance bottlenecks
- Motivate software optimizations
- **Determine when we're done optimizing**
  - Assess performance relative to machine capabilities
  - Motivate need for algorithmic changes
- Predict performance on future machines / architectures
  - Sets realistic expectations on performance for future procurements
  - Used for HW/SW Co-Design to ensure future architectures are well-suited for the computational needs of today's applications.

Performance Models / Simulators

- Historically, many performance models and simulators tracked latencies to predict performance (i.e. counting cycles)

- The last two decades saw a number of latency-hiding techniques…
  - Out-of-order execution (hardware discovers parallelism to hide latency)
  - HW stream prefetching (hardware speculatively loads data)
  - Massive thread parallelism (independent threads satisfy the latency-bandwidth product)

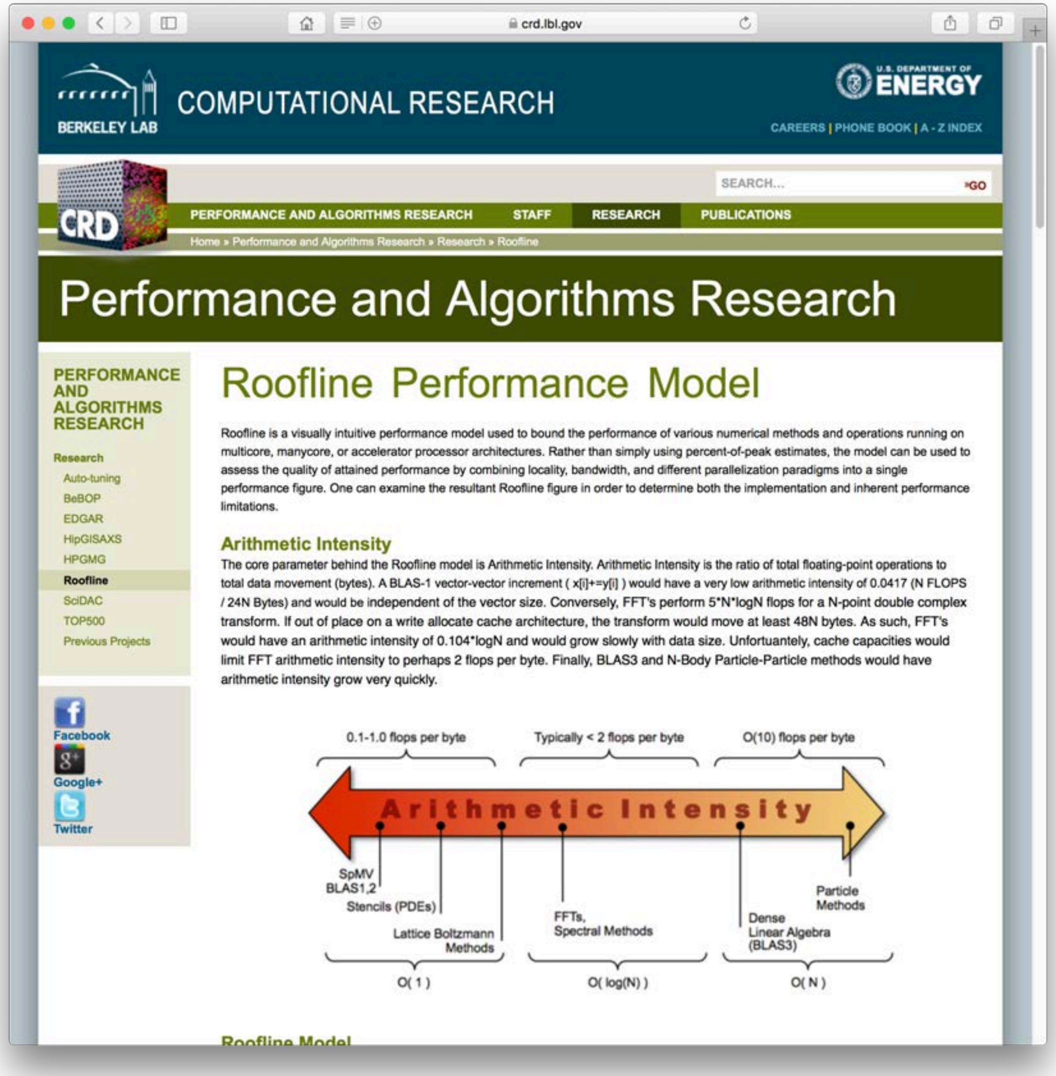
- Effectively latency hiding has resulted in a shift from a latency-limited computing regime to a **throughput-limited computing regime**

Roofline Model

- The **Roofline Model** is a throughput-oriented performance model…
  - Tracks rates not time
  - Augmented with Little's Law

    (concurrency = latency*bandwidth)
  - Independent of ISA and architecture

    (applies to CPUs, GPUs, Google TPUs[1], etc…)
- Three main components:
  - Machine Characterization (realistic performance potential of the system)
  - Monitoring (characterize application's execution)
  - Application Models (how well could my kernel perform with perfect compilers, procs, …)


crd.lbl.gov
BERKELEY LAB
COMPUTATIONAL RESEARCH
U.S. DEPARTMENT OF ENERGY
CAREERS | PHONE BOOK | A - Z INDEX
SEARCH...
»GO
CRD
PERFORMANCE AND ALGORITHMS RESEARCH
STAFF
RESEARCH
PUBLICATIONS
Home » Performance and Algorithms Research » Research » Roofline
PERFORMANCE AND ALGORITHMS RESEARCH
Research
Auto-tuning
BeBOP
EDGAR
HipGISAXS
HPGMG
Roofline
SciDAC
TOP500
Previous Projects
Facebook
Google+
Twitter

Performance and Algorithms Research

Roofline Performance Model

Roofline is a visually intuitive performance model used to bound the performance of various numerical methods and operations running on multicore, manycore, or accelerator processor architectures. Rather than simply using percent-of-peak estimates, the model can be used to assess the quality of attained performance by combining locality, bandwidth, and different parallelization paradigms into a single performance figure. One can examine the resultant Roofline figure in order to determine both the implementation and inherent performance limitations.

Arithmetic Intensity

The core parameter behind the Roofline model is Arithmetic Intensity. Arithmetic Intensity is the ratio of total floating-point operations to total data movement (bytes). A BLAS-1 vector-vector increment (x[i]+=y[i]) would have a very low arithmetic intensity of 0.0417 (N FLOPS / 24N Bytes) and would be independent of the vector size. Conversely, FFT's perform 5*N*logN flops for a N-point double complex transform. If out of place on a write allocate cache architecture, the transform would move at least 48N bytes. As such, FFT's would have an arithmetic intensity of 0.104*logN and would grow slowly with data size. Unfortuantely, cache capacities would limit FFT arithmetic intensity to perhaps 2 flops per byte. Finally, BLAS3 and N-Body Particle-Particle methods would have arithmetic intensity grow very quickly.

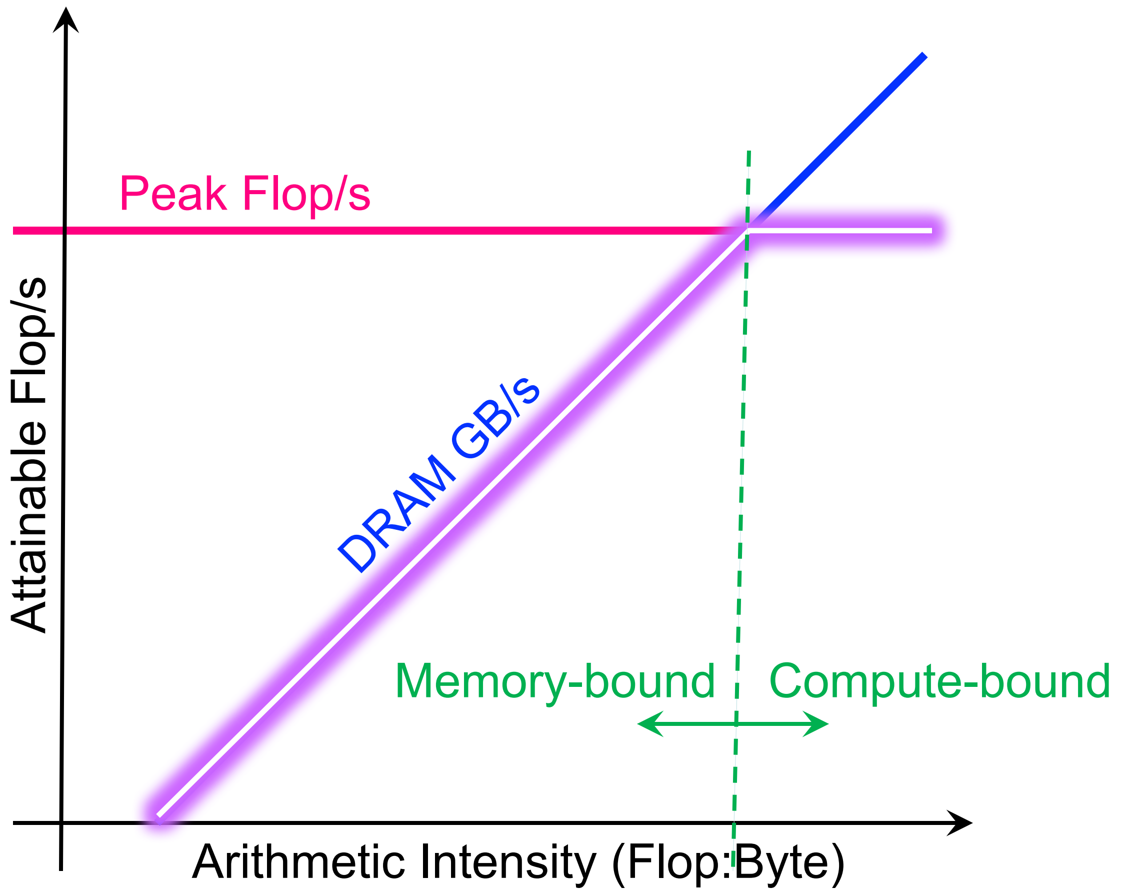
Roofline Model

https://crd.lbl.gov/departments/computer-science/PAR/research/roofline

[1] Jouppi et al, "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA, 2017.

(DRAM) Roofline

- Ideally, we could always attain peak Flop/s
- However, finite locality (reuse) limits performance.
- Plot the performance bound using Arithmetic Intensity (AI) as the x-axis…
  - **Perf Bound = min (peak Flop/s, peak GB/s * AI)**
  - AI = Flops / Bytes presented to DRAM
  - Log-log makes it easy to doodle, extrapolate performance, etc…
  - Kernels with AI less than machine balance are ultimately memory bound.



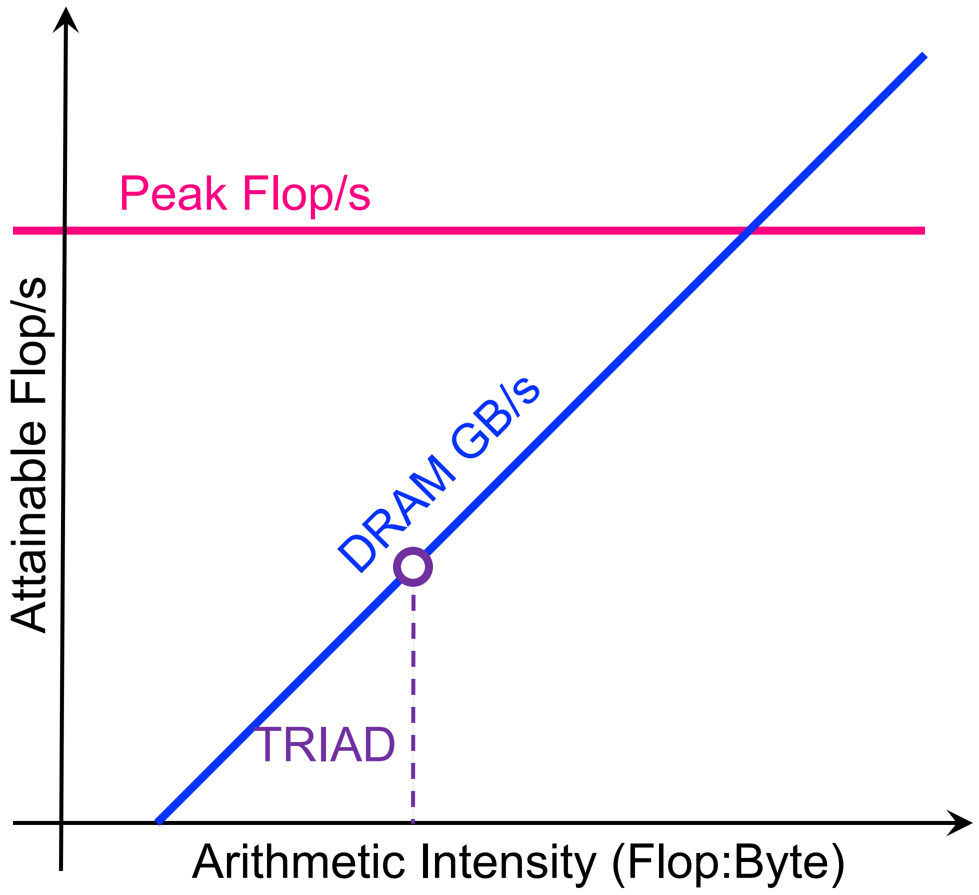
Roofline Examples

- Typical machine balance is 5-10 flops per byte…
  - 40-80 flops per double to exploit compute capability
  - Artifact of technology and money
  - **Unlikely to improve**

- Consider STREAM Triad…

```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

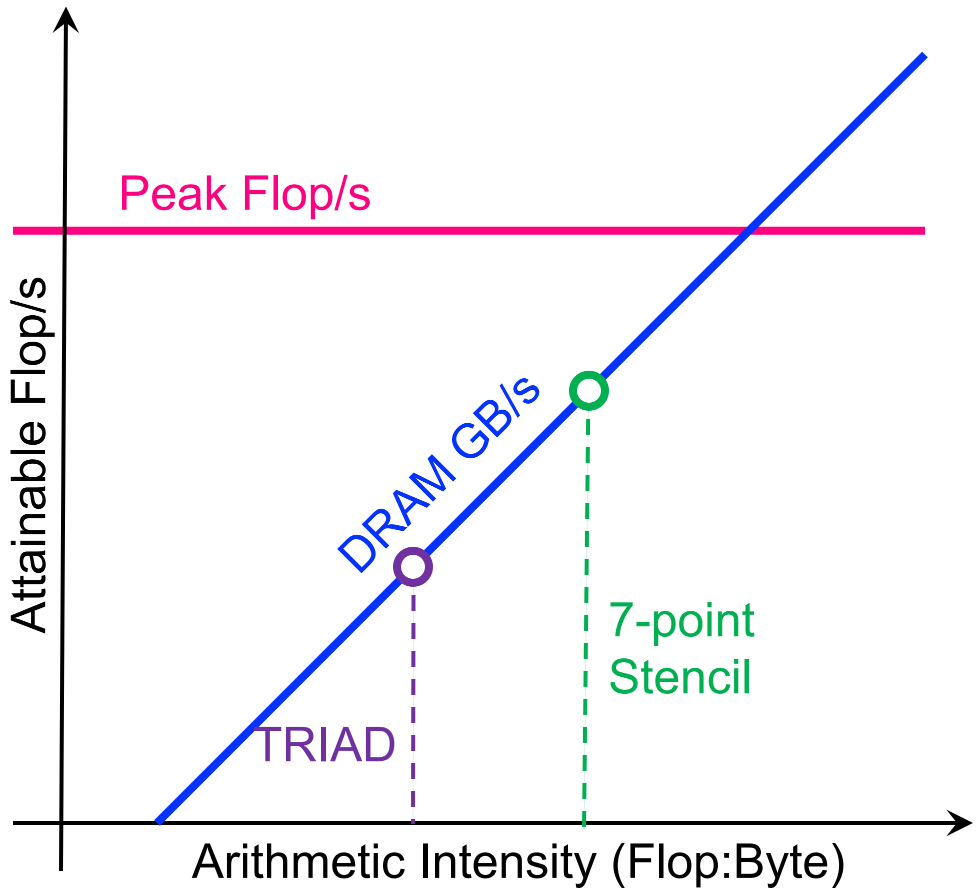
  - 2 flops per iteration
  - Transfer 24 bytes per iteration (read X[i], Y[i], write Z[i])
  - **AI = 0.166 flops per byte == Memory bound**



Roofline Examples

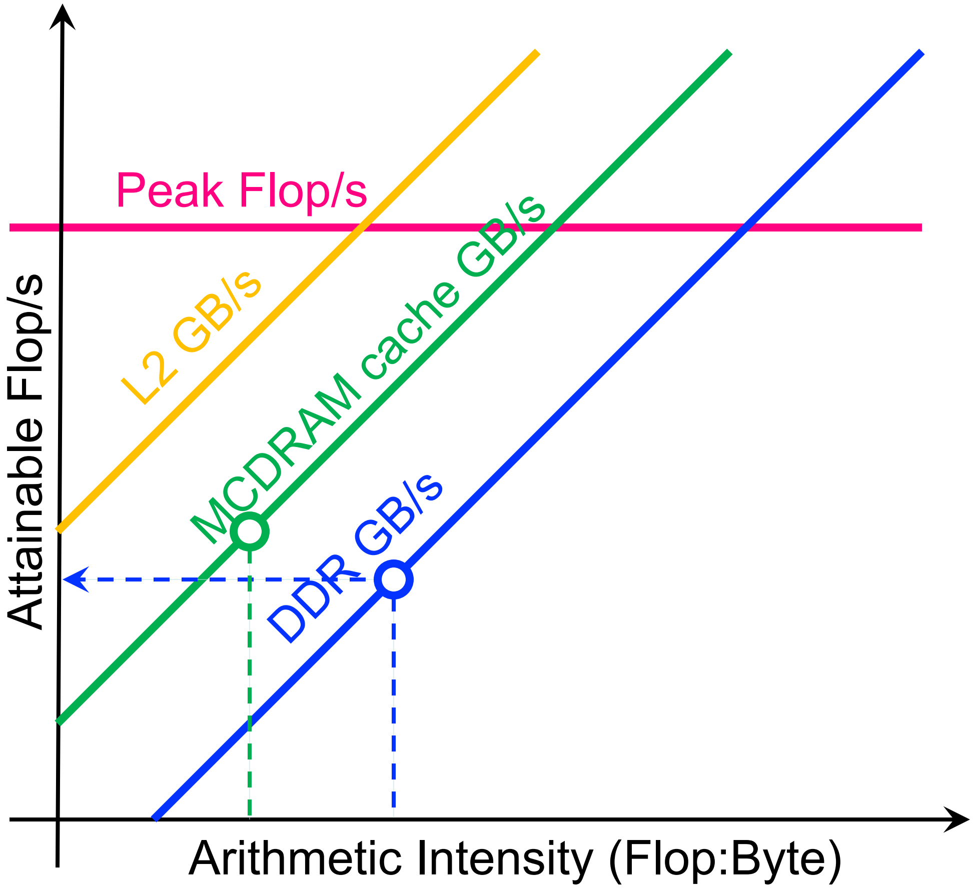
- Conversely, 7-point constant coefficient stencil…
  - 7 flops
  - 8 memory references (7 reads, 1 store) per point
  - Cache can filter all but 1 read and 1 write per point
  - **AI = 0.43 flops per byte == memory bound, but 3x the flop rate**

```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  int ijk = i + j*jStride + k*kStride;
  new[ijk] = -6.0*old[ijk        ]
                + old[ijk-1      ]
                + old[ijk+1      ]
                + old[ijk-jStride]
                + old[ijk+jStride]
                + old[ijk-kStride]
                + old[ijk+kStride];
}}}
```



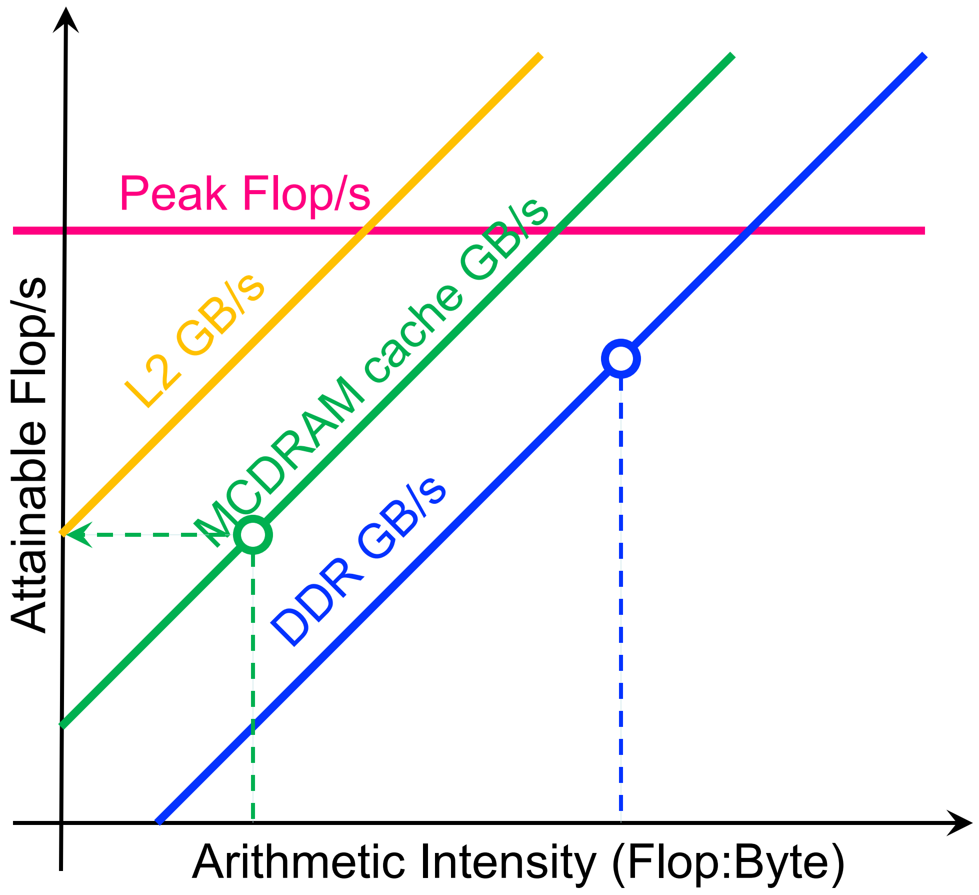
Hierarchical Roofline

- Real processors have multiple levels of memory
  - Registers
  - L1, L2, L3 cache
  - MCDRAM/HBM (KNL/GPU device memory)
  - DDR (main memory)
  - NVRAM (non-volatile memory)
- We may measure a bandwidth and define an AI for each level
  - A given application / kernel / loop nest will thus have multiple AI's
  - A kernel could be DDR-limited…



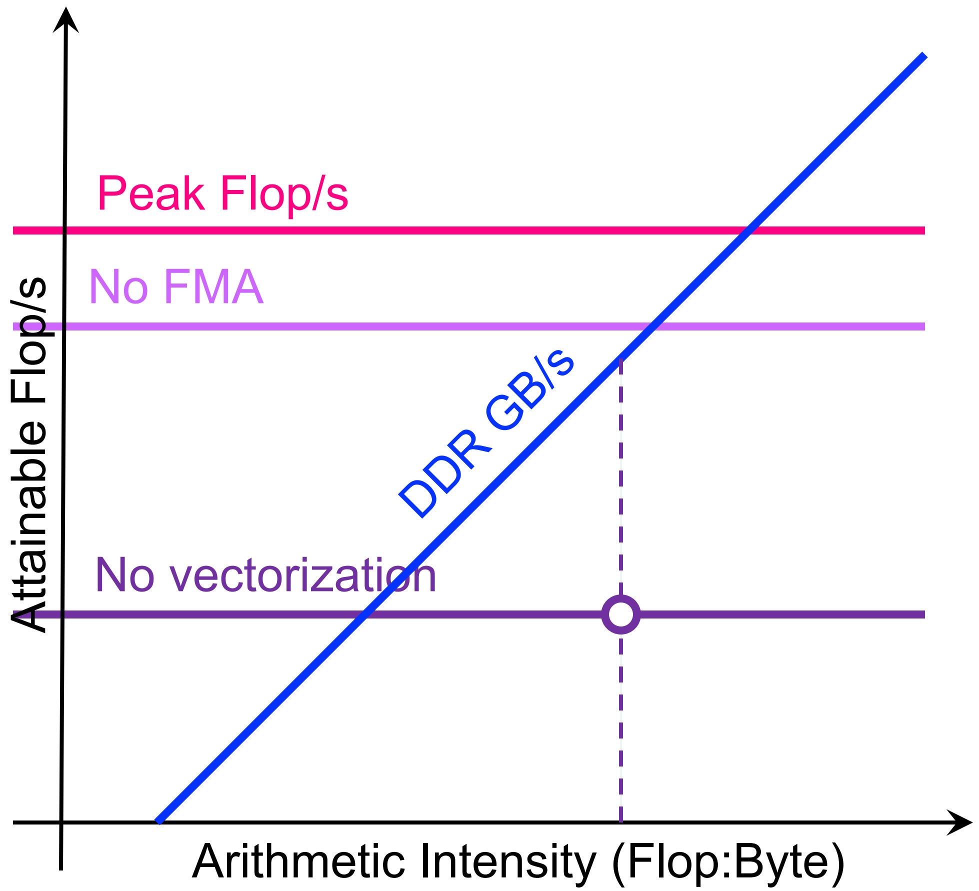
Hierarchical Roofline

- Real processors have multiple levels of memory
  - Registers
  - L1, L2, L3 cache
  - MCDRAM/HBM (KNL/GPU device memory)
  - DDR (main memory)
  - NVRAM (non-volatile memory)
- We may measure a bandwidth and define an AI for each level
  - A given application / kernel / loop nest will thus have multiple AI's
  - A kernel could be DDR-limited…
  - **or MCDRAM-limited depending on relative bandwidths and AI's**



Data, Instruction, Thread-Level Parallelism…

- We have assumed one can attain peak flops with high locality.
- In reality, this is premised on sufficient…
  - Use special instructions (e.g. fused multiply-add)
  - Vectorization (16 flops per instruction)
  - unrolling, out-of-order execution (hide FPU latency)
  - OpenMP across multiple cores
- Without these, …
  - Peak performance is not attainable
  - Some kernels can transition from memory-bound to compute-bound
  - n.b. in reality, DRAM bandwidth is often tied to DLP and TLP (single core can't saturate BW w/scalar code)



Roofline using ERT, VTune, and SDE

Basic Roofline Modeling

Machine Characterization

Potential of my target system

- How does my system respond to a lack of FMA, DLP, ILP, TLP?
- How does my system respond to reduced AI (i.e. memory/cache bandwidth)?
- How does my system respond to NUMA, strided, or random memory access patterns?
- …

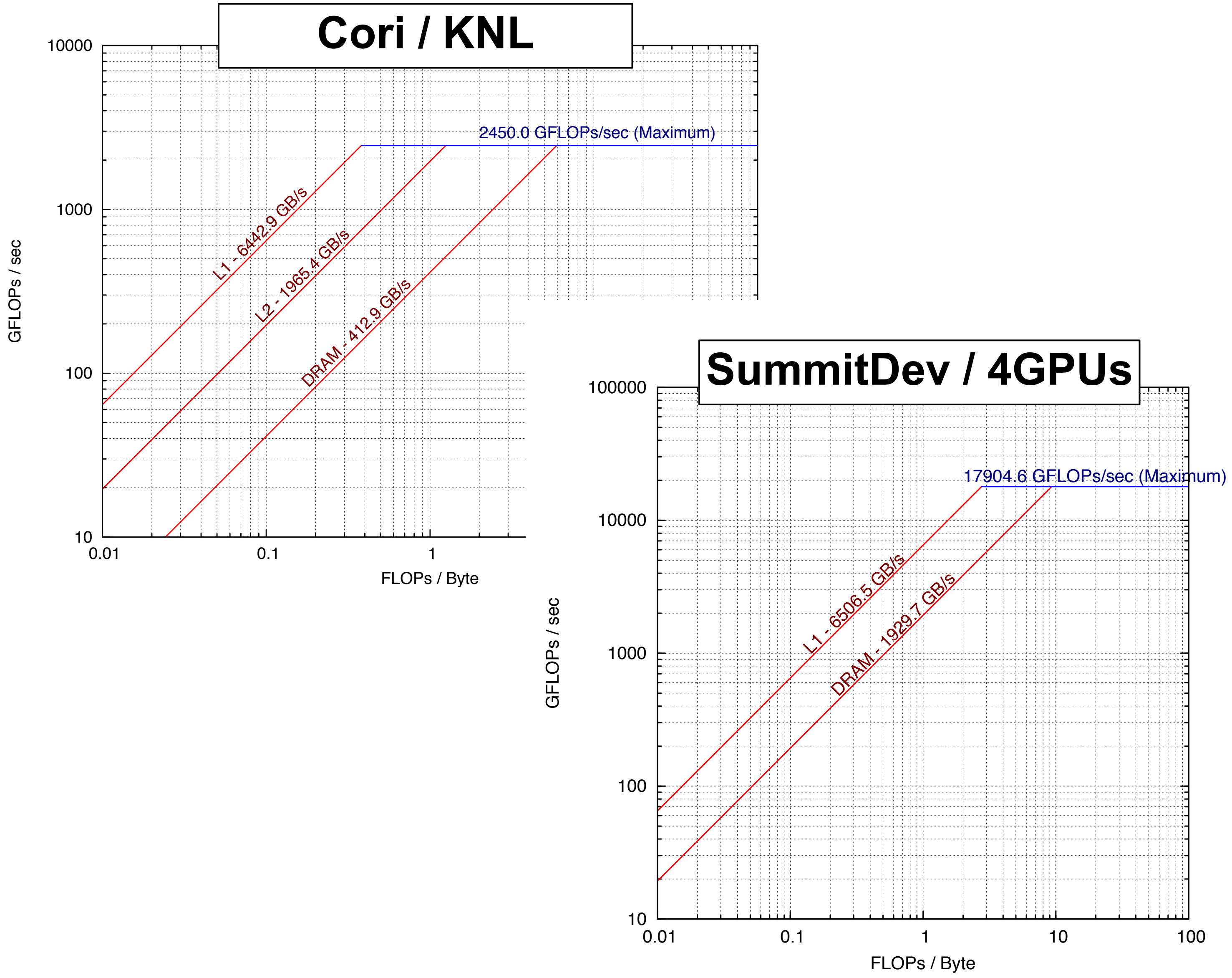
Application Instrumentation

Properties of my app's execution

- What is my app's real AI?
- How does AI vary with memory level ?
- How well does my app vectorize?
- Does my app use FMA?
- ...

How Fast is My Target System?

- Challenges:
  - Too many systems; new ones each year
  - Voluminous documentation on each
  - Real performance often less than **“Marketing Numbers”**
  - **Compilers can “give up” on big loops**
- Empirical Roofline Toolkit (ERT)
  - Characterize CPU/GPU systems
  - Peak Flop rates
  - Bandwidths for each level of memory
  - **MPI+OpenMP/CUDA == multiple GPUs**
- https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

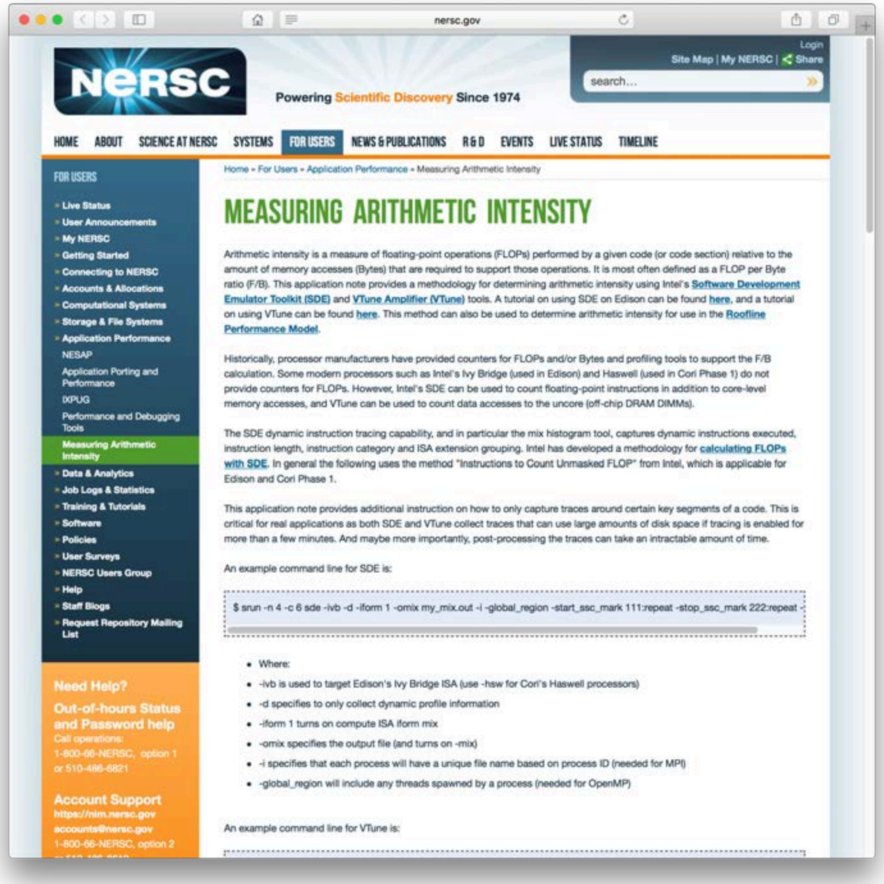


Application Instrumentation Can Be Hard…

- Flop counters **can be broken/missing in production HW** (Haswell)
- Counting Loads and Stores is a poor proxy for data movement as they **don't capture reuse**
- Counting L1 misses is a poor proxy for data movement as they **don't account for HW prefetching**.
- DRAM counters are accurate, but **are privileged and thus nominally inaccessible in user mode**
- OS/kernel changes must be approved by vendor (e.g. Cray) and the center (e.g. NERSC)

Application Instrumentation

- NERSC/CRD (==NESAP/SUPER) collaboration…
  - Characterize applications running on NERSC production systems
  - Use **Intel SDE** (binary instrumentation) to create software Flop counters (could use Byfl as well)
  - Use **Intel VTune** performance tool (NERSC/Cray approved) to access uncore counters
  - **Produced accurate measurement of Flop's and DRAM data movement on HSW and KNL**



Home » For Users » Application Performance » Measuring Arithmetic Intensity

MEASURING ARITHMETIC INTENSITY

Arithmetic intensity is a measure of floating-point operations (FLOPs) performed by a given code (or code section) relative to the amount of memory accesses (Bytes) that are required to support those operations. It is most often defined as a FLOP per Byte ratio (F/B). This application note provides a methodology for determining arithmetic intensity using Intel's Software Development Emulator Toolkit (SDE) and VTune Amplifier (VTune) tools. A tutorial on using SDE on Edison can be found here, and a tutorial on using VTune can be found here. This method can also be used to determine arithmetic intensity for use in the Roofline Performance Model.

Historically, processor manufacturers have provided counters for FLOPs and/or Bytes and profiling tools to support the F/B calculation. Some modern processors such as Intel's Ivy Bridge (used in Edison) and Haswell (used in Cori Phase 1) do not provide counters for FLOPs. However, Intel's SDE can be used to count floating-point instructions in addition to core-level memory accesses, and VTune can be used to count data accesses to the uncore (off-chip DRAM DIMMs).

The SDE dynamic instruction tracing capability, and in particular the mix histogram tool, captures dynamic instructions executed, instruction length, instruction category and ISA extension grouping. Intel has developed a methodology for calculating FLOPs with SDE. In general the following uses the method "Instructions to Count Unmasked FLOP" from Intel, which is applicable for Edison and Cori Phase 1.

This application note provides additional instruction on how to only capture traces around certain key segments of a code. This is critical for real applications as both SDE and VTune collect traces that can use large amounts of disk space if tracing is enabled for more than a few minutes. And maybe more importantly, post-processing the traces can take an intractable amount of time.

An example command line for SDE is:

```
$ srun -n 4 -c 6 sde -ivb -d -iform 1 -omix my_mix.out -i -global_region -start_ssc_mark 111:repeat -stop_ssc_mark 222:repeat -
```

- Where:
- -ivb is used to target Edison's Ivy Bridge ISA (use -hsw for Cori's Haswell processors)
- -d specifies to only collect dynamic profile information
- -iform 1 turns on compute ISA iform mix
- -omix specifies the output file (and turns on -mix)
- -i specifies that each process will have a unique file name based on process ID (needed for MPI)
- -global_region will include any threads spawned by a process (needed for OpenMP)

An example command line for VTune is:

http://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity/

NERSC is LBL's production computing division
CRD is LBL's Computational Research Division
NESAP is NERSC's KNL application readiness project
LBL is part of SUPER (DOE SciDAC3 Computer Science Institute)

Use by NESAP

- NESAP is the NERSC KNL application readiness project.
- NESAP used Roofline to drive optimization and analysis on KNL
  - Bound performance expectations (ERT)
  - Quantify DDR and MCDRAM data movement
  - Compare KNL data movement to Haswell (sea of private/coherent L2's vs. unified L3)
  - Understand importance of vectorization

- Doerfer et al., **"Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor",** *Intel Xeon Phi User Group Workshop (IXPUG),* June 2016.
- Barnes et al. **"Evaluating and Optimizing the NERSC Workload on Knights Landing",** *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS),* November 2016.

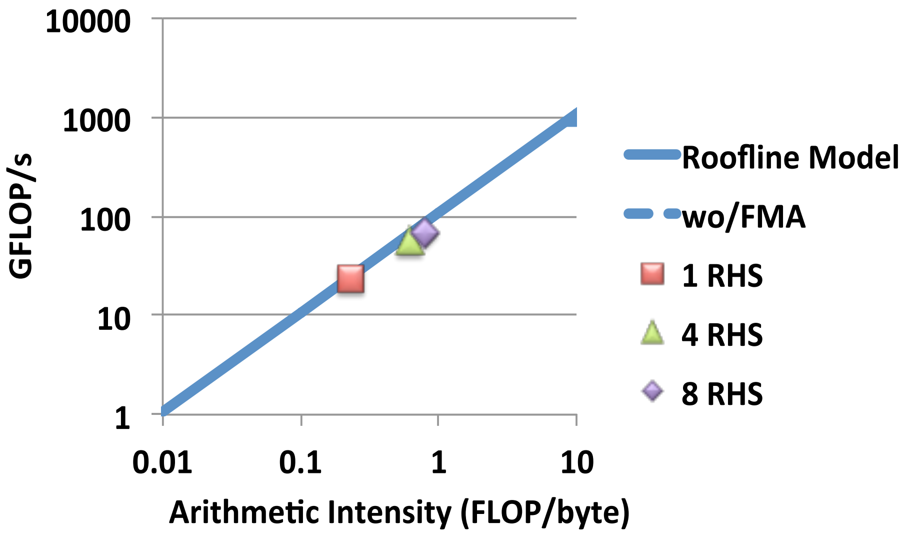
Roofline for NESAP Codes

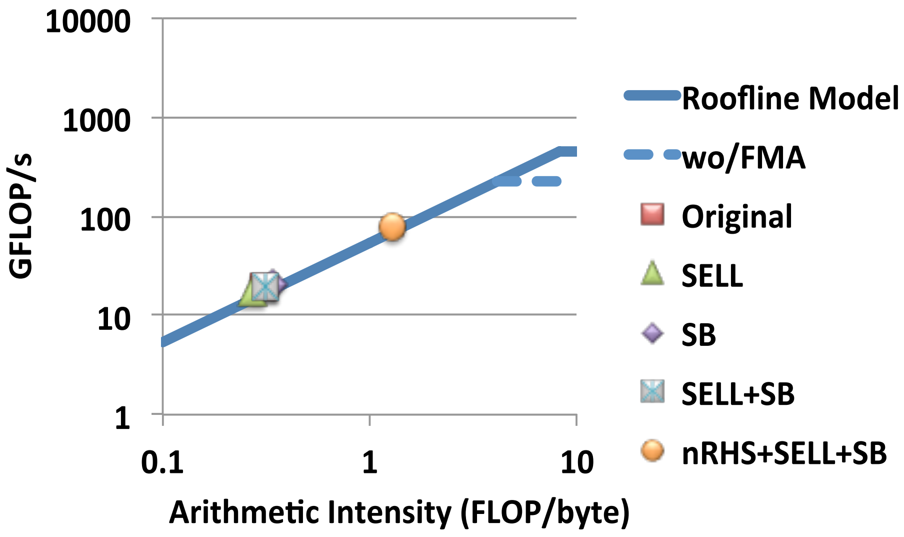
MFDn

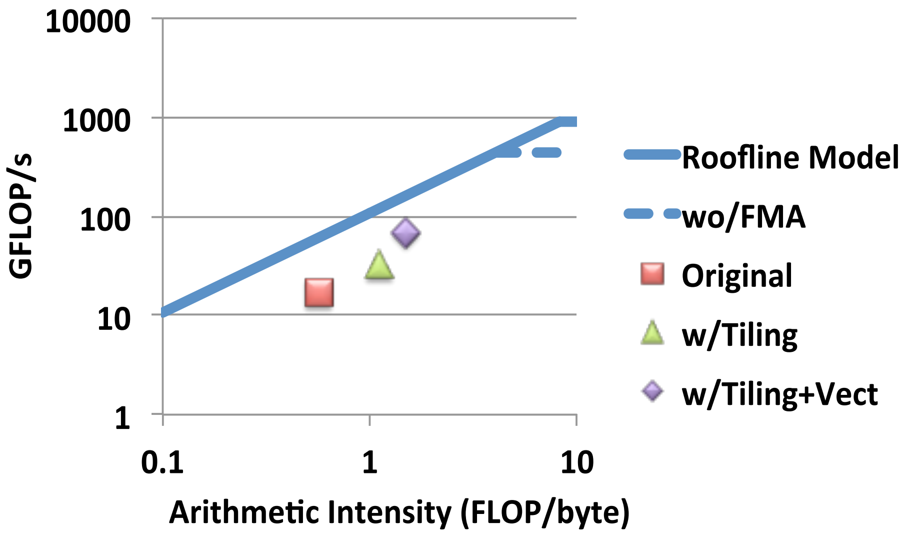
EMGeo

PICSAR

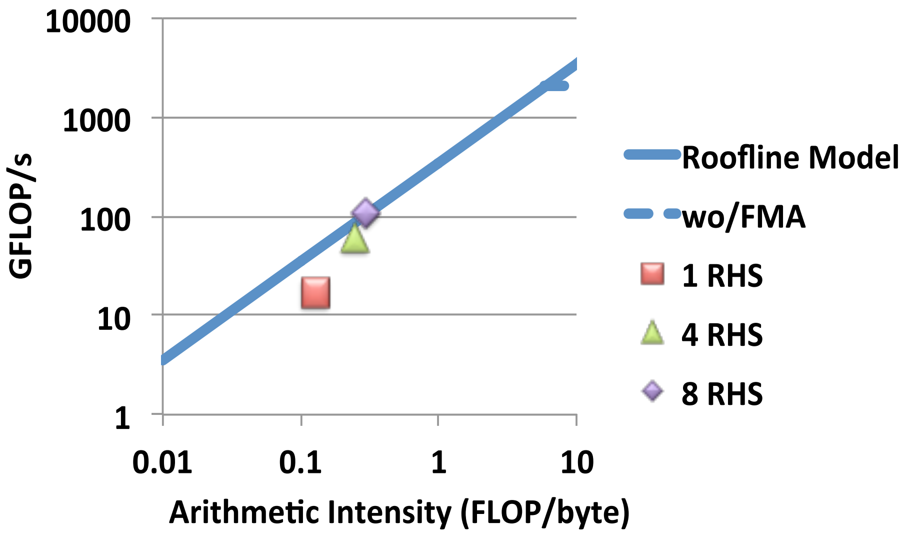
2P HSW

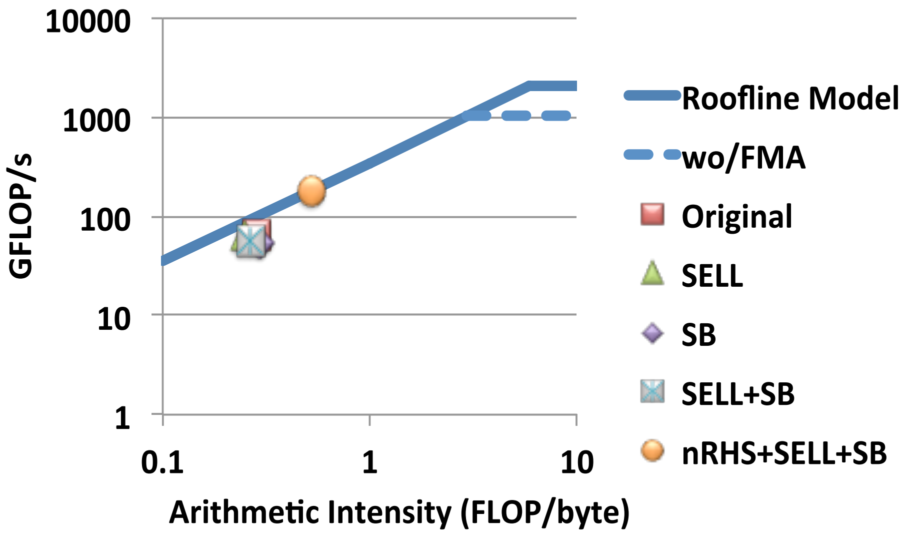


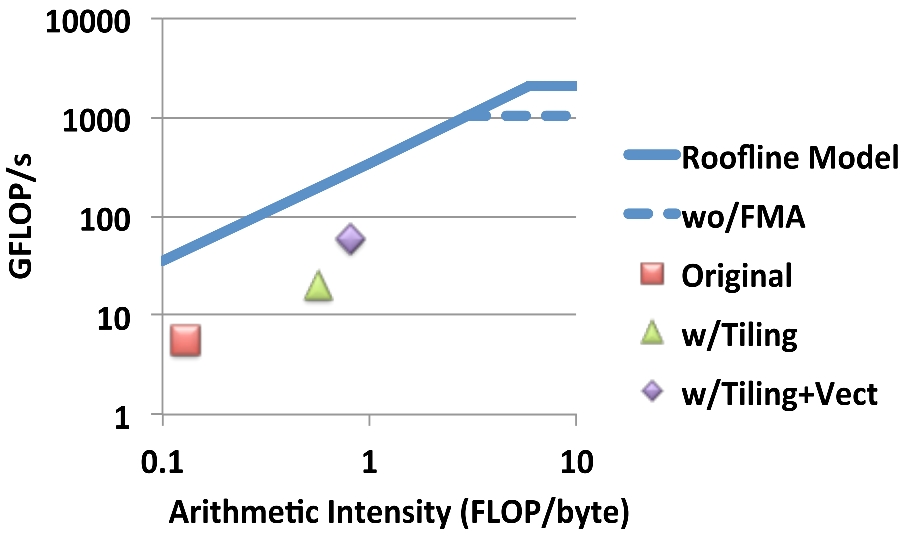




KNL

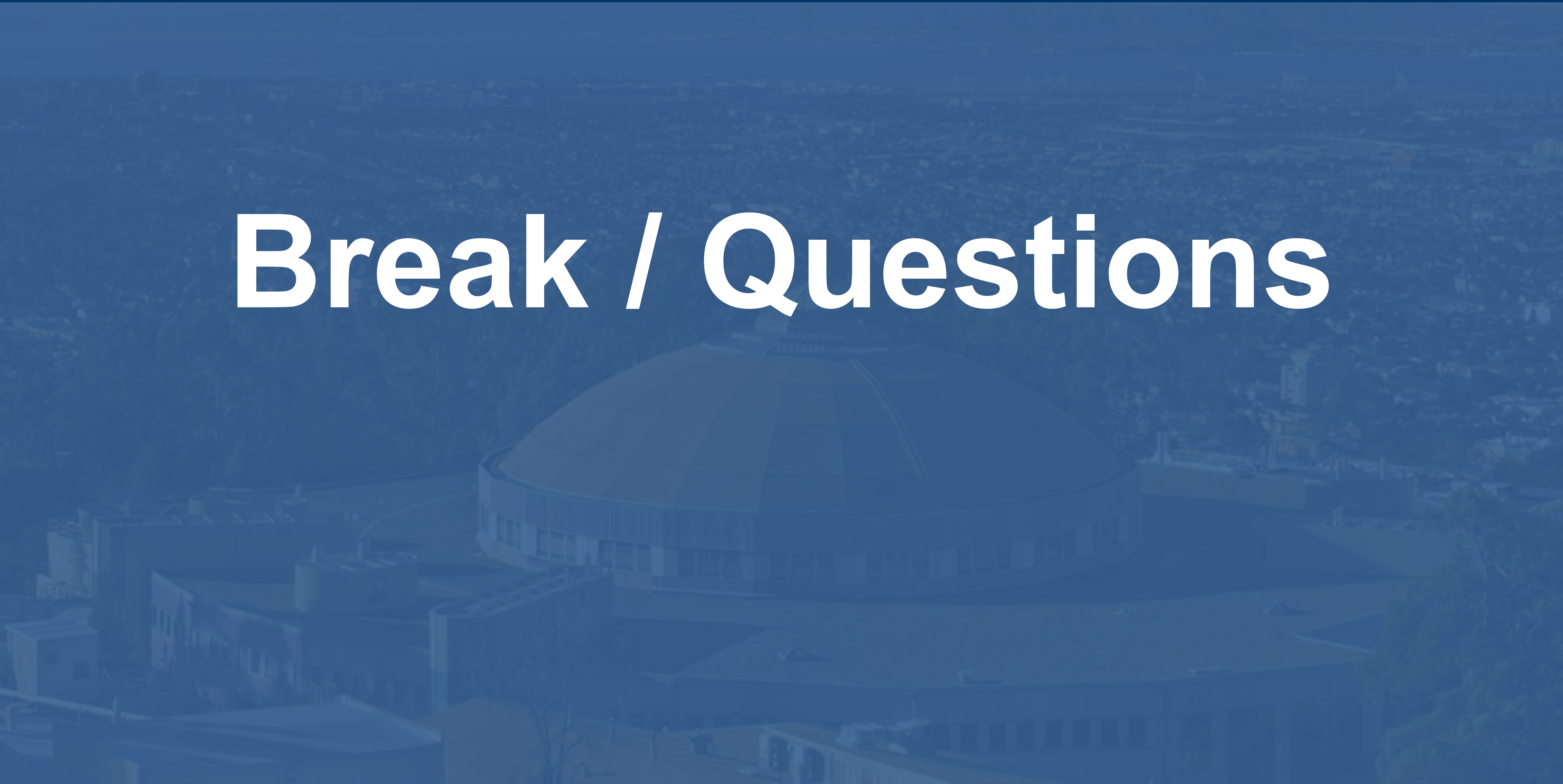






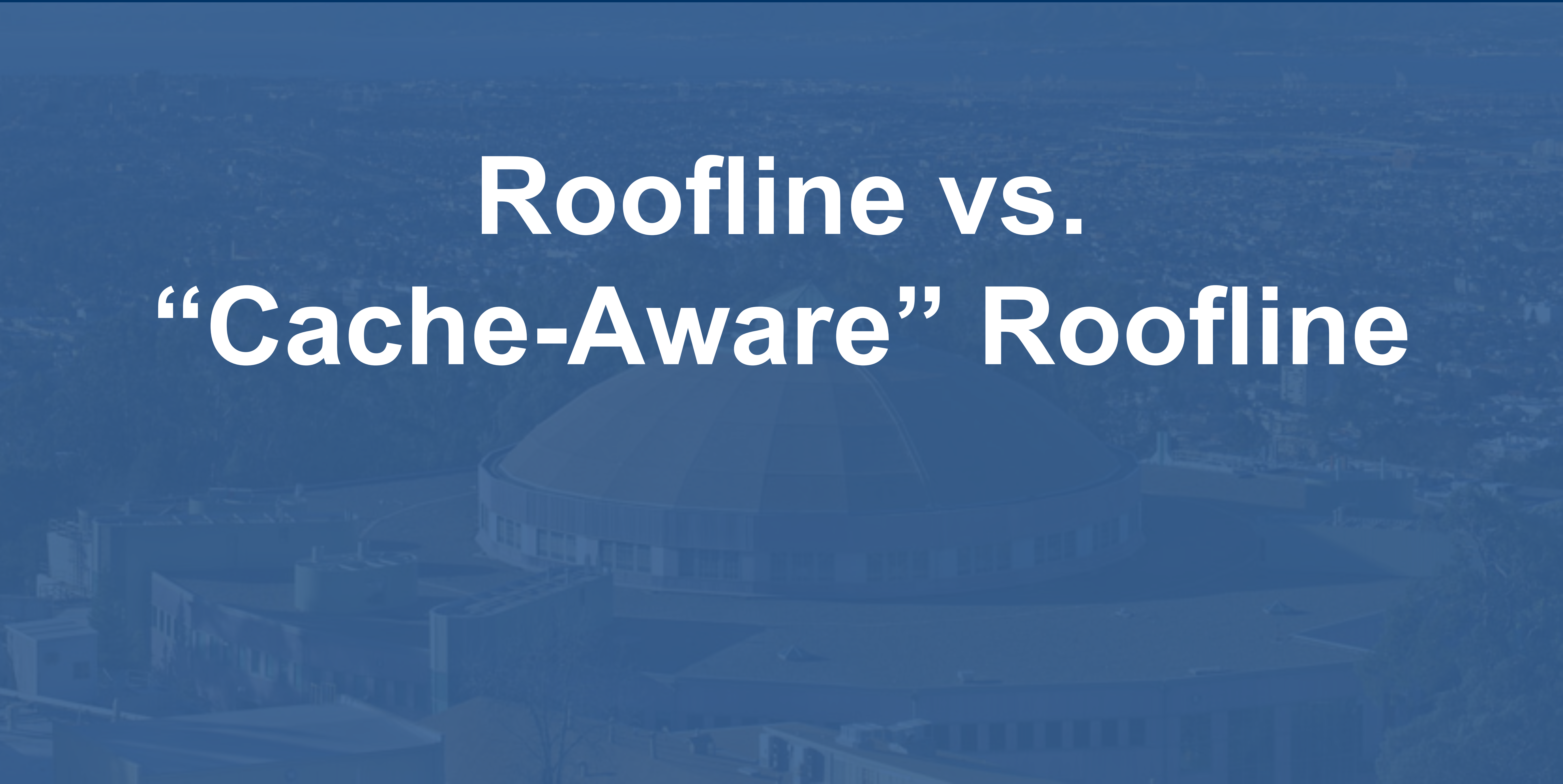
Need a integrated solution…

- Having to compose VTune, SDE, and graphing tools worked correctly and benefitted NESAP, but …
- …placed a very high burden on users…
  - **forced to learn/run multiple tools**
  - **forced to instrument each routine in their application**
  - **forced to manually parse/compose/graph the output**
- …still lacked integration with compiler/debugger/disassembly
- CRD/NERSC wanted a more integrated solution…



Break / Questions

Roofline vs. “Cache-Aware” Roofline



There are two Major Roofline Formulations:

- Original / DRAM / Hierarchical Roofline…
  - **Williams, et al, “Roofline: An Insightful Visual Performance Model for Multicore Architectures”, CACM, 2009**
  - Defines multiple bandwidth ceilings and multiple AI’s per kernel
  - Performance bound is the minimum of the intercepts and flops
- “Cache-Aware” Roofline
  - **Ilic et al, "Cache-aware Roofline model: Upgrading the loft", IEEE Computer Architecture Letters, 2014**
  - Defines multiple bandwidth ceilings, but uses a single AI (flop:L1 bytes)
  - As one looses cache locality (capacity, conflict, …) performance falls from one BW ceiling to a lower one at constant AI
- Why Does this matter?
  - Some tools use the original Roofline, some use cache-aware **== Users need to understand the differences**
  - **Intel Advisor uses the Cache-Aware Roofline Model** (alpha/experimental DRAM Roofline being evaluated)
  - CRD/NERSC prefer the hierarchical Roofline as it provides greater insights into the behavior of the memory hierarchy

Roofline

- Captures cache effects
- AI is Flop:Bytes after being ***filtered by lower cache levels***
- Multiple Arithmetic Intensities (one per level of memory)
- AI ***dependent*** on problem size (capacity misses reduce AI)
- Memory/Cache/Locality effects are ***directly observed***
- Requires ***performance counters*** to measure AI

"Cache-Aware" Roofline

- Captures cache effects
- AI is Flop:Bytes ***as presented to the L1 cache***
- Single Arithmetic Intensity
- AI ***independent*** of problem size
- Memory/Cache/Locality effects are ***indirectly observed***
- Requires static analysis or ***binary instrumentation*** to measure AI

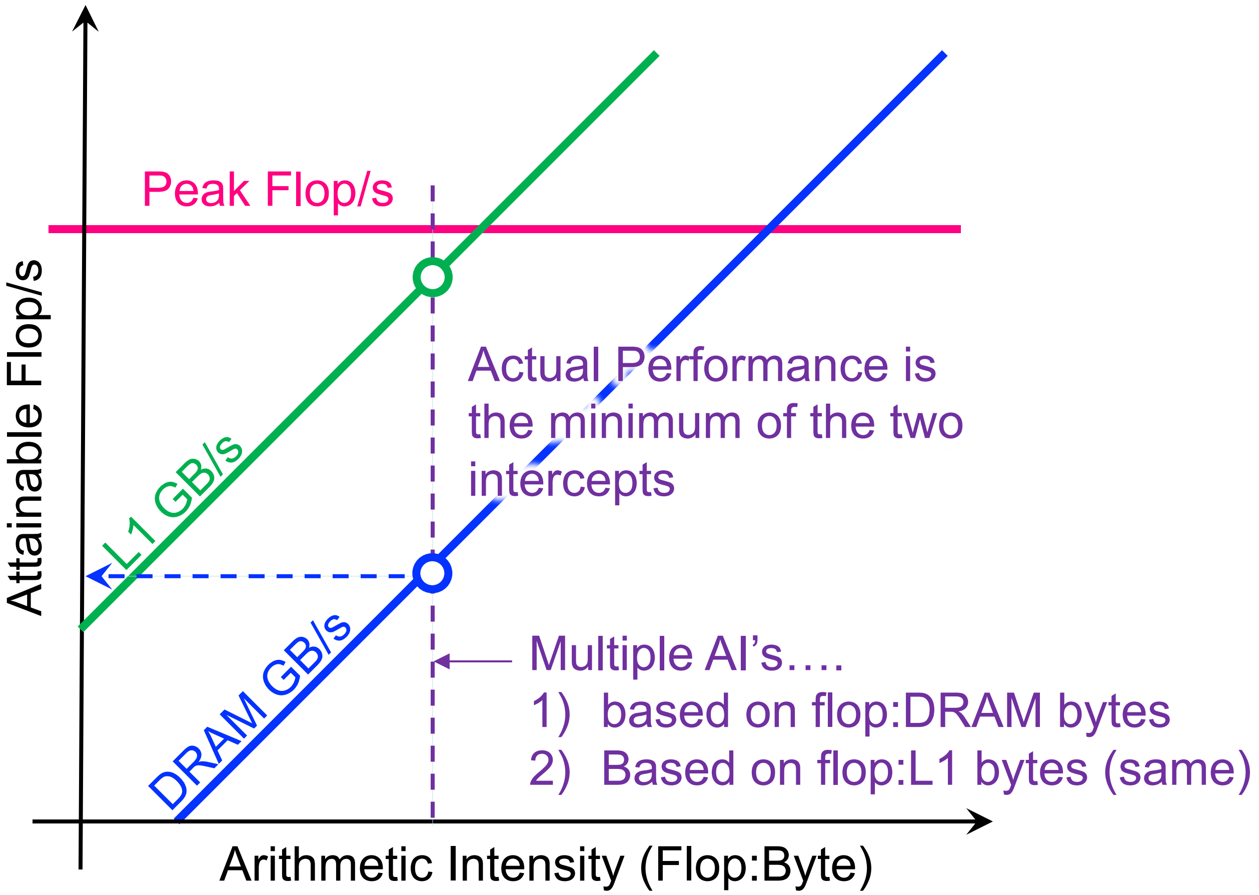
Example: STREAM

- L1 AI…
  - 2 flops
  - 2 x 8B load (old)
  - 1 x 8B store (new)
  - = 0.08 flops per byte
- No cache reuse…
  - Iteration i doesn’t touch any data associated with iteration i+delta for any delta.
- … leads to a DRAM AI equal to the L1 AI

```
#pragma omp parallel for
for(i=0;i<N;i++){
  Z[i] = X[i] + alpha*Y[i];
}
```

Example: STREAM

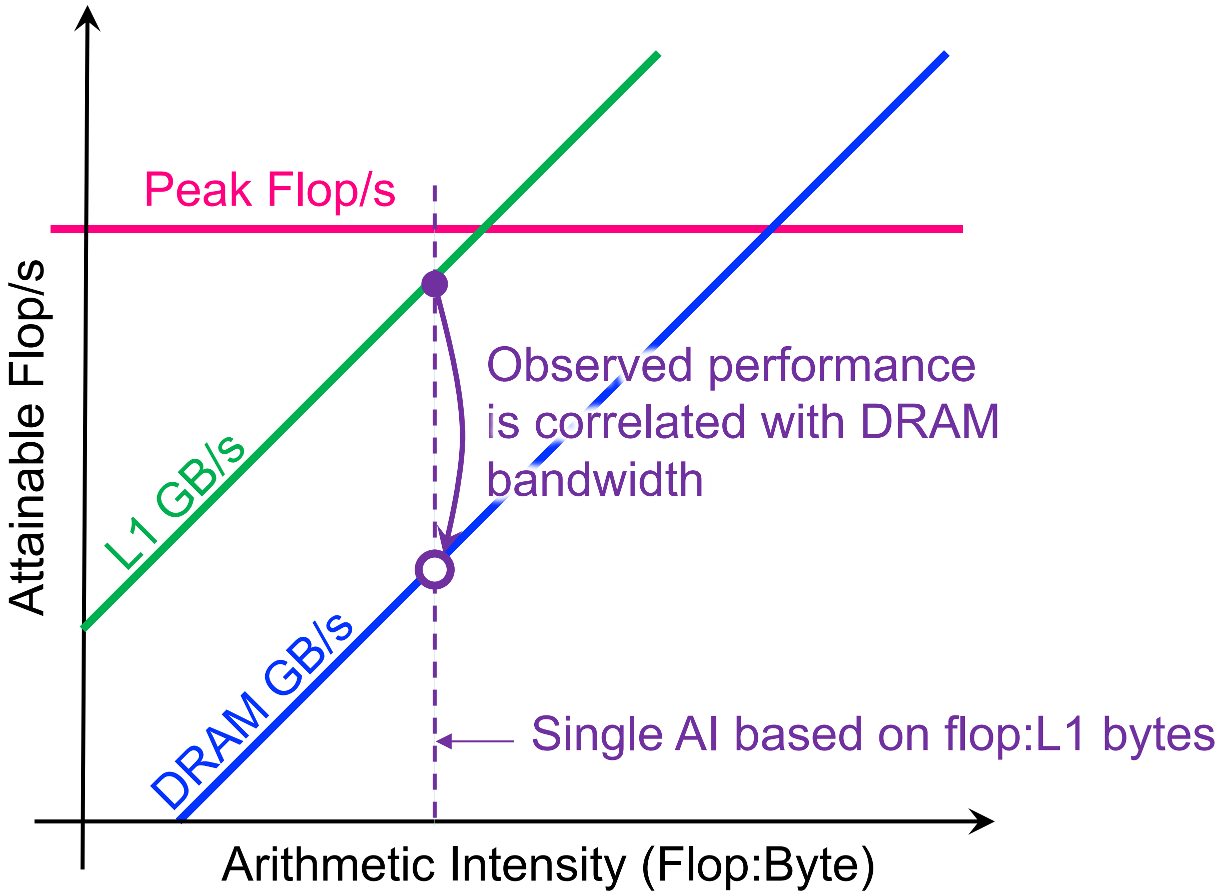
Roofline



Actual Performance is the minimum of the two intercepts

Multiple AI's….
1) based on flop:DRAM bytes
2) Based on flop:L1 bytes (same)

"Cache-Aware" Roofline



Observed performance is correlated with DRAM bandwidth

Single AI based on flop:L1 bytes

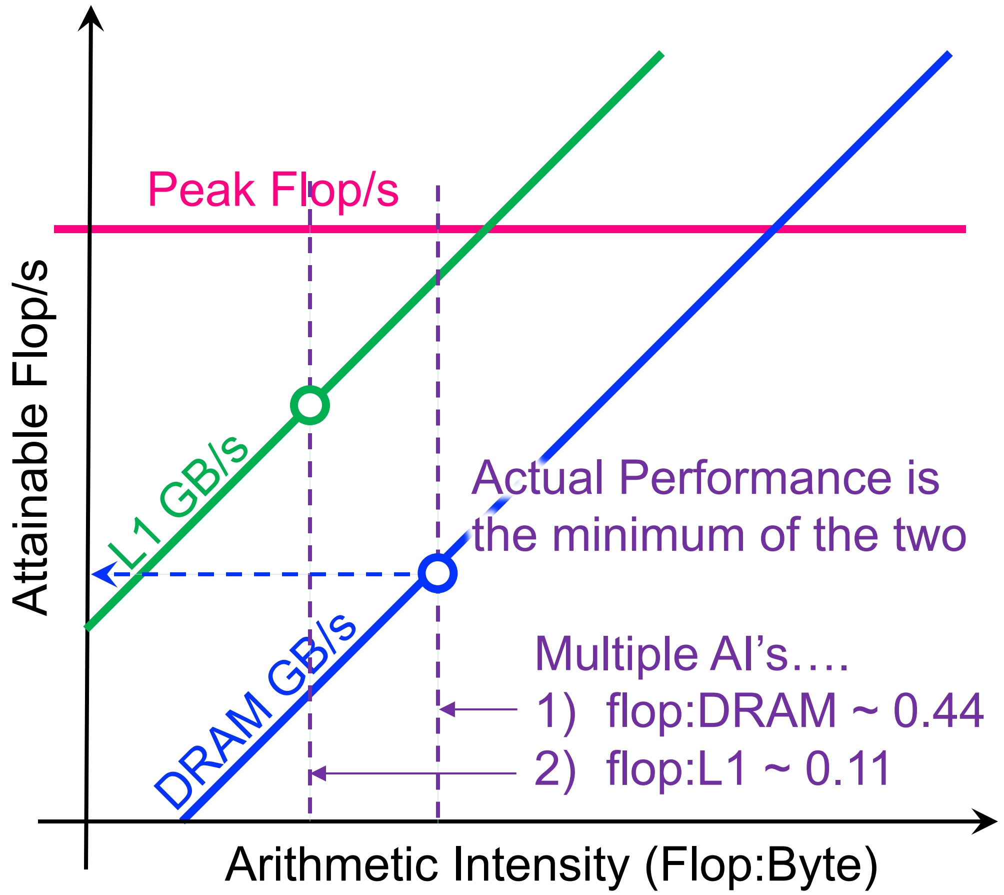
Example: 7-point Stencil (Small Problem)

- L1 AI…
  - 7 flops
  - 7 x 8B load (old)
  - 1 x 8B store (new)
  - = 0.11 flops per byte
  - some compilers may do register shuffles to reduce the number of loads.
- Moderate cache reuse…
  - old[ijk] is reused on subsequent iterations of i,j,k
  - old[ijk-1] is reused on subsequent iterations of i.
  - old[ijk-jStride] is reused on subsequent iterations of j.
  - old[ijk-kStride] is reused on subsequent iterations of k.
- … leads to DRAM AI larger than the L1 AI

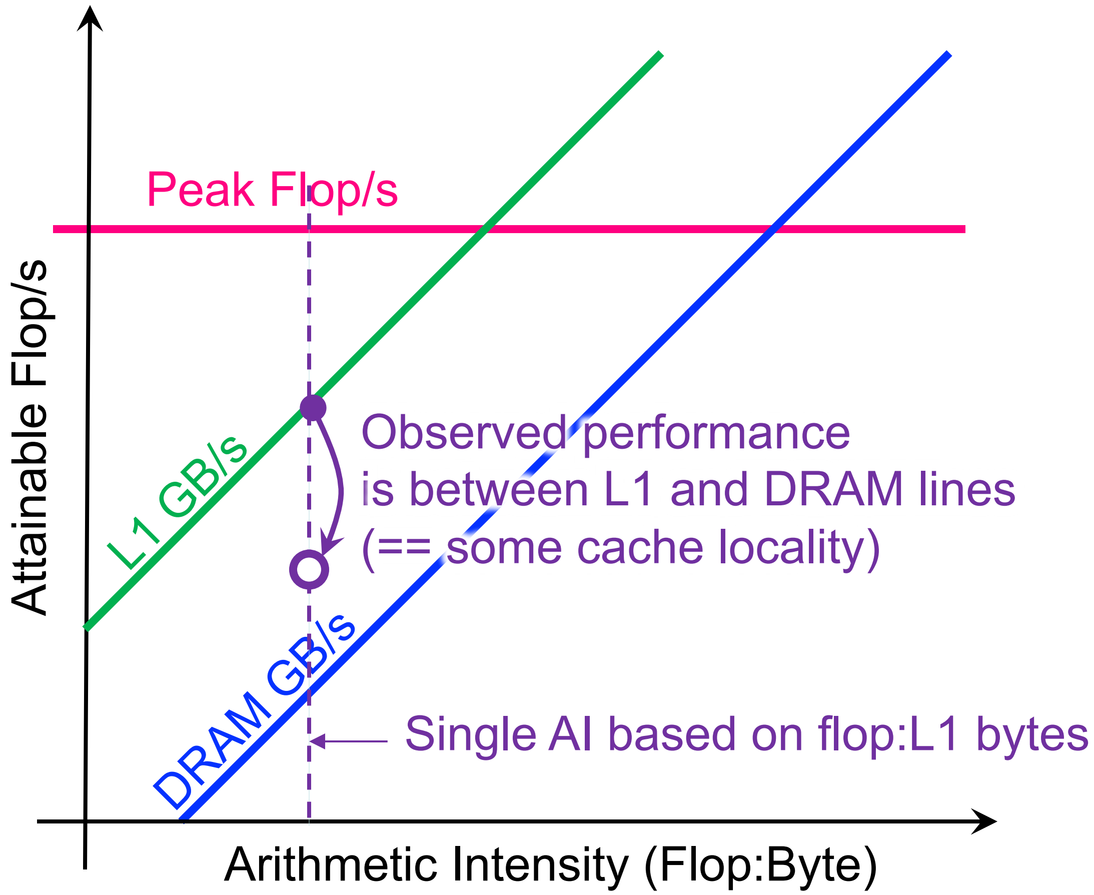
```
#pragma omp parallel for
for(k=1;k<dim+1;k++){
for(j=1;j<dim+1;j++){
for(i=1;i<dim+1;i++){
  int ijk = i + j*jStride + k*kStride;
  new[ijk] = -6.0*old[ijk        ]
                  + old[ijk-1      ]
                  + old[ijk+1      ]
                  + old[ijk-jStride]
                  + old[ijk+jStride]
                  + old[ijk-kStride]
                  + old[ijk+kStride];
}}}
```

Example: 7-point Stencil (Small Problem)

Roofline



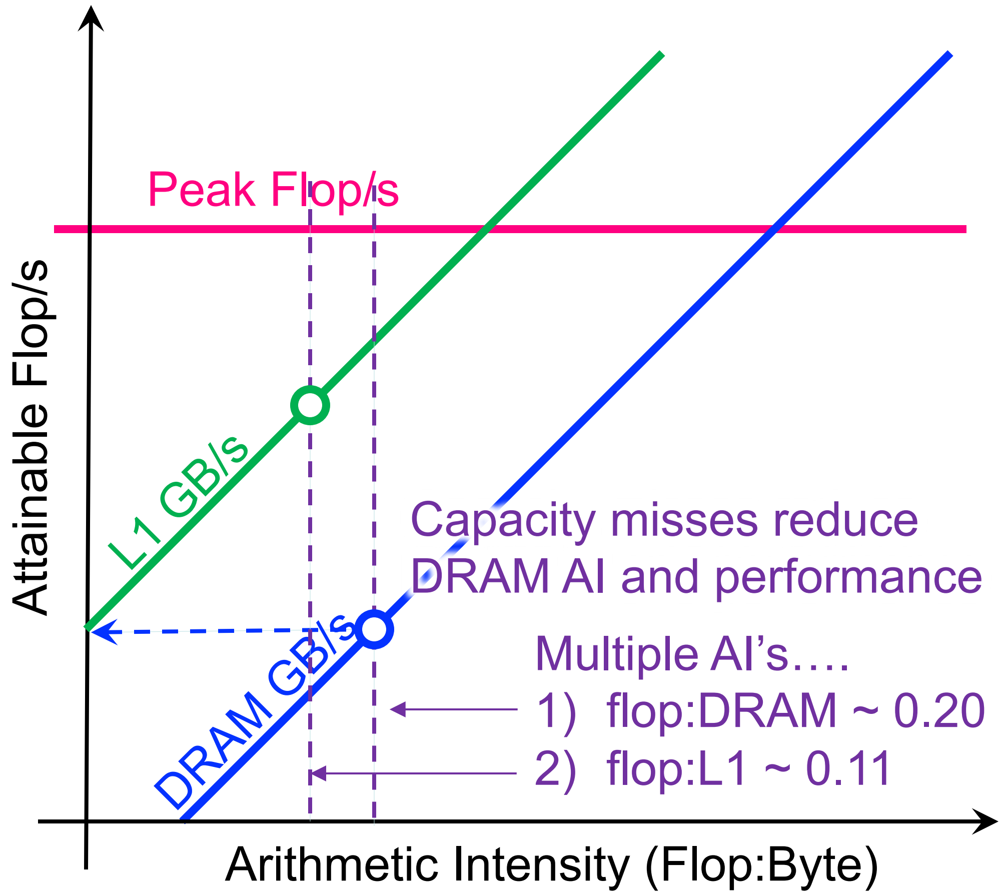
"Cache-Aware" Roofline



Peak Flop/s
Attainable Flop/s
L1 GB/s
DRAM GB/s
Observed performance is between L1 and DRAM lines (== some cache locality)
Single AI based on flop:L1 bytes
Arithmetic Intensity (Flop:Byte)

Example: 7-point Stencil (Large Problem)

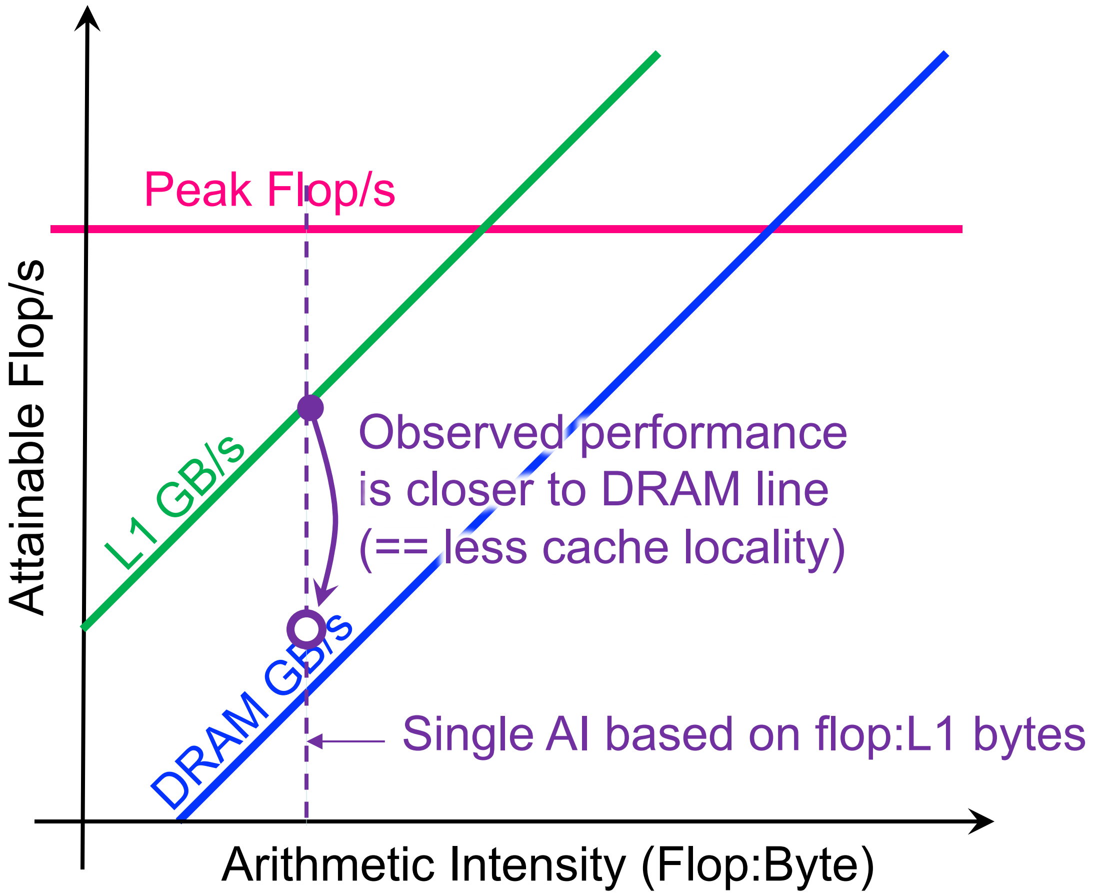
Roofline



Capacity misses reduce DRAM AI and performance

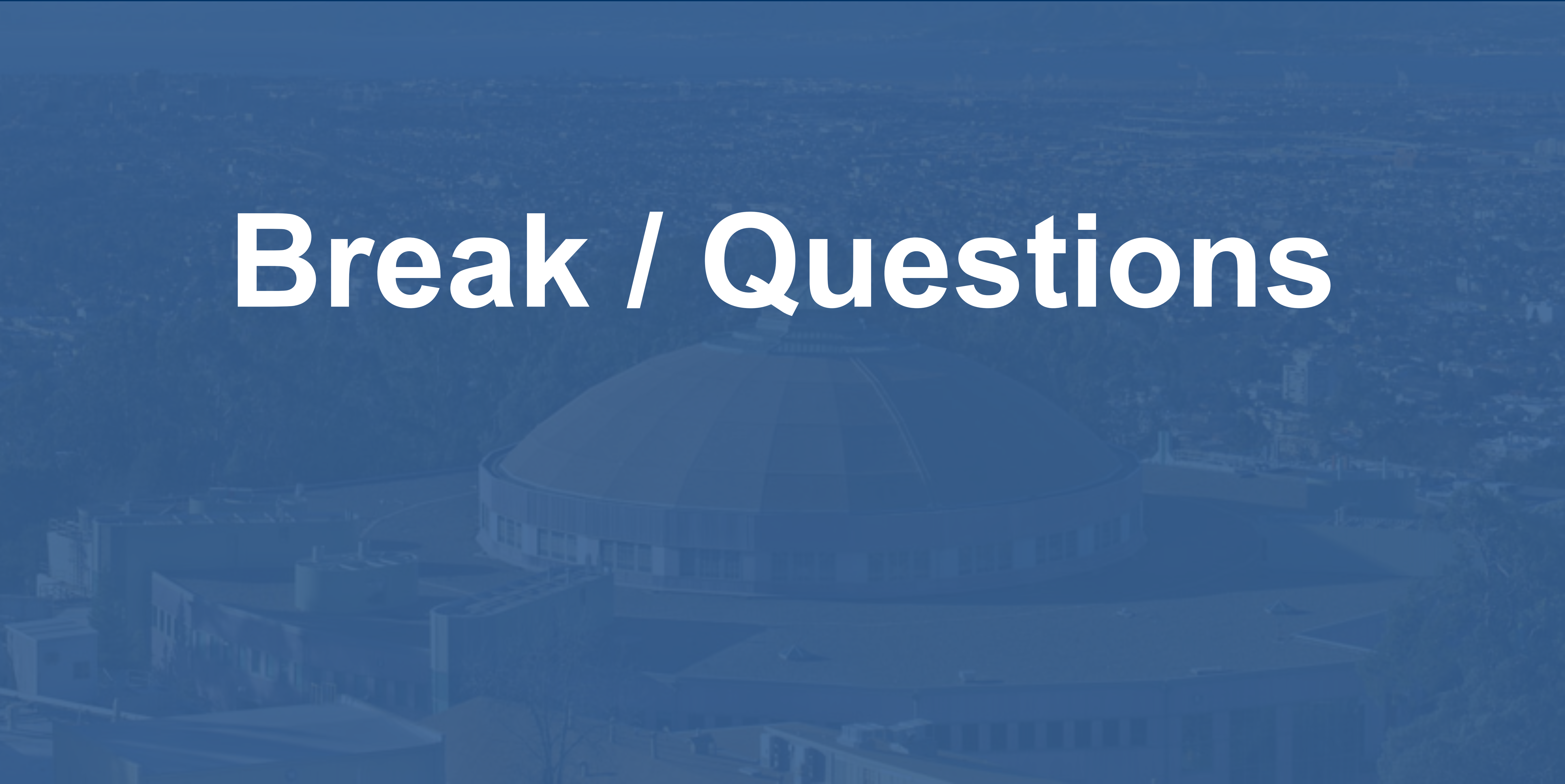
Multiple AI's….
1) flop:DRAM ~ 0.20
2) flop:L1 ~ 0.11

"Cache-Aware" Roofline



Observed performance is closer to DRAM line (== less cache locality)

Single AI based on flop:L1 bytes



Break / Questions

Intel Advisor:

Introduction and General Usage

DRAM Roofline and OS/X Advisor GUI: These are preview features that may or may not be included in mainline product releases. They may not be stable as they are prototypes incorporating very new functionality. Intel provides preview features in order to collect user feedback and plans further development and productization steps based on the feedback.

Intel Advisor

- Integrated Performance Analysis Tool
  - Performance information including timings, flops, and trip counts
  - Vectorization Tips
  - Memory footprint analysis
  - **Uses the Cache-Aware Roofline Model**
  - All connected back to source code

- CRD/NERSC began a collaboration with Intel
  - Ensure Advisor runs on Cori in user-mode
  - Push for **Hierarchical Roofline**
  - Make it functional/scalable to many MPI processes across multiple nodes
  - Validate results on NESAP, SciDAC, and ECP codes

NESAP is NERSC's KNL application readiness project
SciDAC is the DOE Office of Science's Scientific Discovery thru Advanced Computing program
ECP is the DOE's Exascale Computing Project

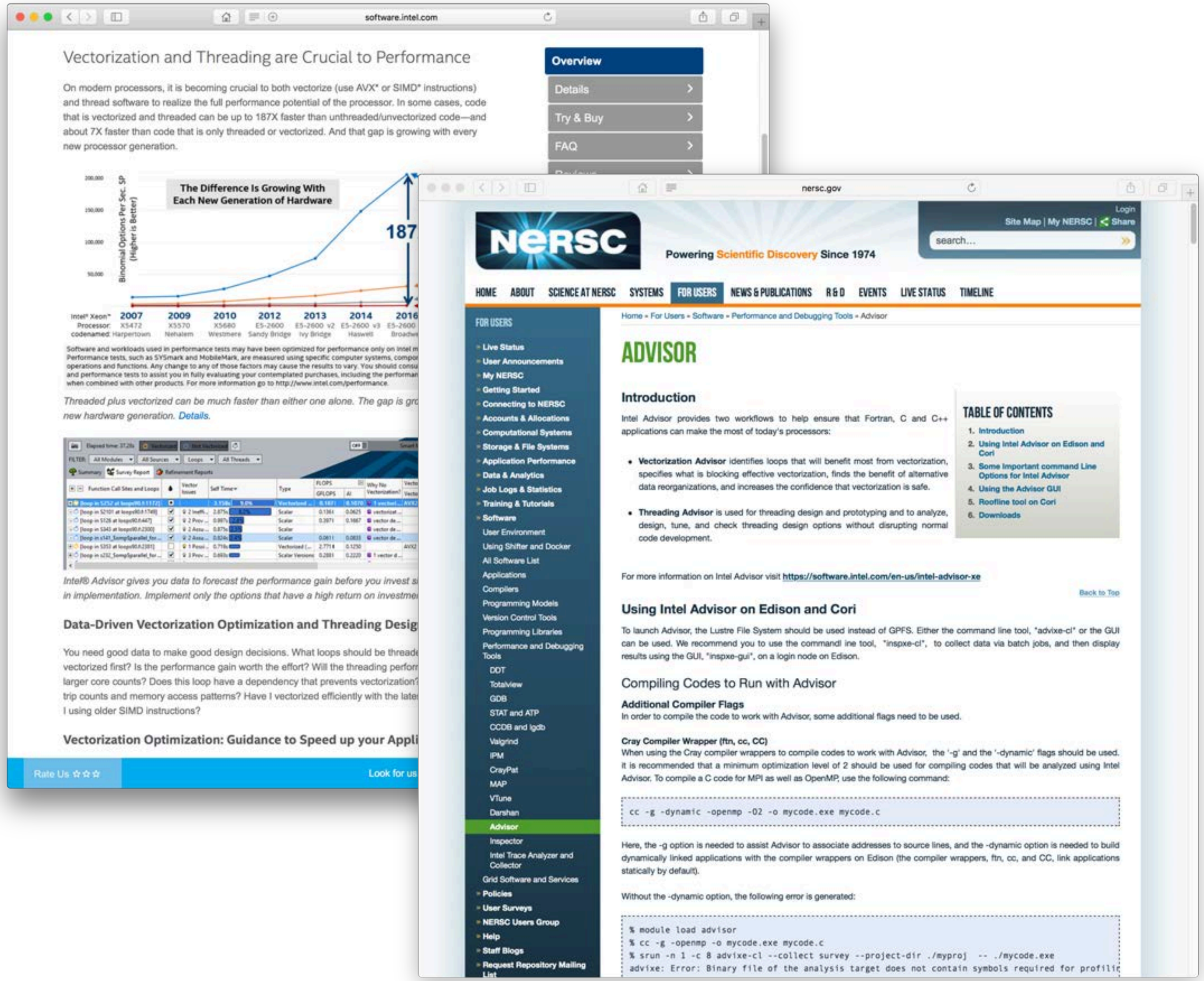
Intel Advisor (Useful Links)

Background

- https://software.intel.com/en-us/intel-advisor-xe
- https://software.intel.com/en-us/articles/getting-started-with-intel-advisor-roofline-feature
- https://www.youtube.com/watch?v=h2QEM1HpFgg

Running Advisor on NERSC Systems

- http://www.nersc.gov/users/software/performance-and-debugging-tools/advisor/



Vectorization and Threading are Crucial to Performance

On modern processors, it is becoming crucial to both vectorize (use AVX* or SIMD* instructions) and thread software to realize the full performance potential of the processor. In some cases, code that is vectorized and threaded can be up to 187X faster than unthreaded/unvectorized code—and about 7X faster than code that is only threaded or vectorized. And that gap is growing with every new processor generation.

Software and workloads used in performance tests may have been optimized for performance only on Intel m Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, compo operations and functions. Any change to any of those factors may cause the results to vary. You should consu and performance tests to assist you in fully evaluating your contemplated purchases, including the performan when combined with other products. For more information go to http://www.intel.com/performance.

Threaded plus vectorized can be much faster than either one alone. The gap is gro new hardware generation. Details.

Intel® Advisor gives you data to forecast the performance gain before you invest s in implementation. Implement only the options that have a high return on investme

Data-Driven Vectorization Optimization and Threading Desig

You need good data to make good design decisions. What loops should be threade vectorized first? Is the performance gain worth the effort? Will the threading perfor larger core counts? Does this loop have a dependency that prevents vectorization? trip counts and memory access patterns? Have I vectorized efficiently with the late I using older SIMD instructions?

Vectorization Optimization: Guidance to Speed up your Appli

Home » For Users » Software » Performance and Debugging Tools » Advisor

ADVISOR

Introduction

Intel Advisor provides two workflows to help ensure that Fortran, C and C++ applications can make the most of today's processors:

- **Vectorization Advisor** identifies loops that will benefit most from vectorization, specifies what is blocking effective vectorization, finds the benefit of alternative data reorganizations, and increases the confidence that vectorization is safe.
- **Threading Advisor** is used for threading design and prototyping and to analyze, design, tune, and check threading design options without disrupting normal code development.

TABLE OF CONTENTS

For more information on Intel Advisor visit https://software.intel.com/en-us/intel-advisor-xe

Using Intel Advisor on Edison and Cori

To launch Advisor, the Lustre File System should be used instead of GPFS. Either the command line tool, "advixe-cl" or the GUI can be used. We recommend you to use the commandl ine tool, "inspxe-cl", to collect data via batch jobs, and then display results using the GUI, "inspxe-gui", on a login node on Edison.

Compiling Codes to Run with Advisor

Additional Compiler Flags
In order to compile the code to work with Advisor, some additional flags need to be used.

Cray Compiler Wrapper (ftn, cc, CC)
When using the Cray compiler wrappers to compile codes to work with Advisor, the '-g' and the '-dynamic' flags should be used. It is recommended that a minimum optimization level of 2 should be used for compiling codes that will be analyzed using Intel Advisor. To compile a C code for MPI as well as OpenMP, use the following command:

```
cc -g -dynamic -openmp -O2 -o mycode.exe mycode.c
```

Here, the -g option is needed to assist Advisor to associate addresses to source lines, and the -dynamic option is needed to build dynamically linked applications with the compiler wrappers on Edison (the compiler wrappers, ftn, cc, and CC, link applications statically by default).

Without the -dynamic option, the following error is generated:

```
% module load advisor
% cc -g -openmp -o mycode.exe mycode.c
% srun -n 1 -c 8 advixe-cl --collect survey --project-dir ./myproj  -- ./mycode.exe
advixe: Error: Binary file of the analysis target does not contain symbols required for profilir
```

Using Intel Advisor at NERSC

- Compile…

```
use ‘-g’ when compiling
```

- Submit Job…

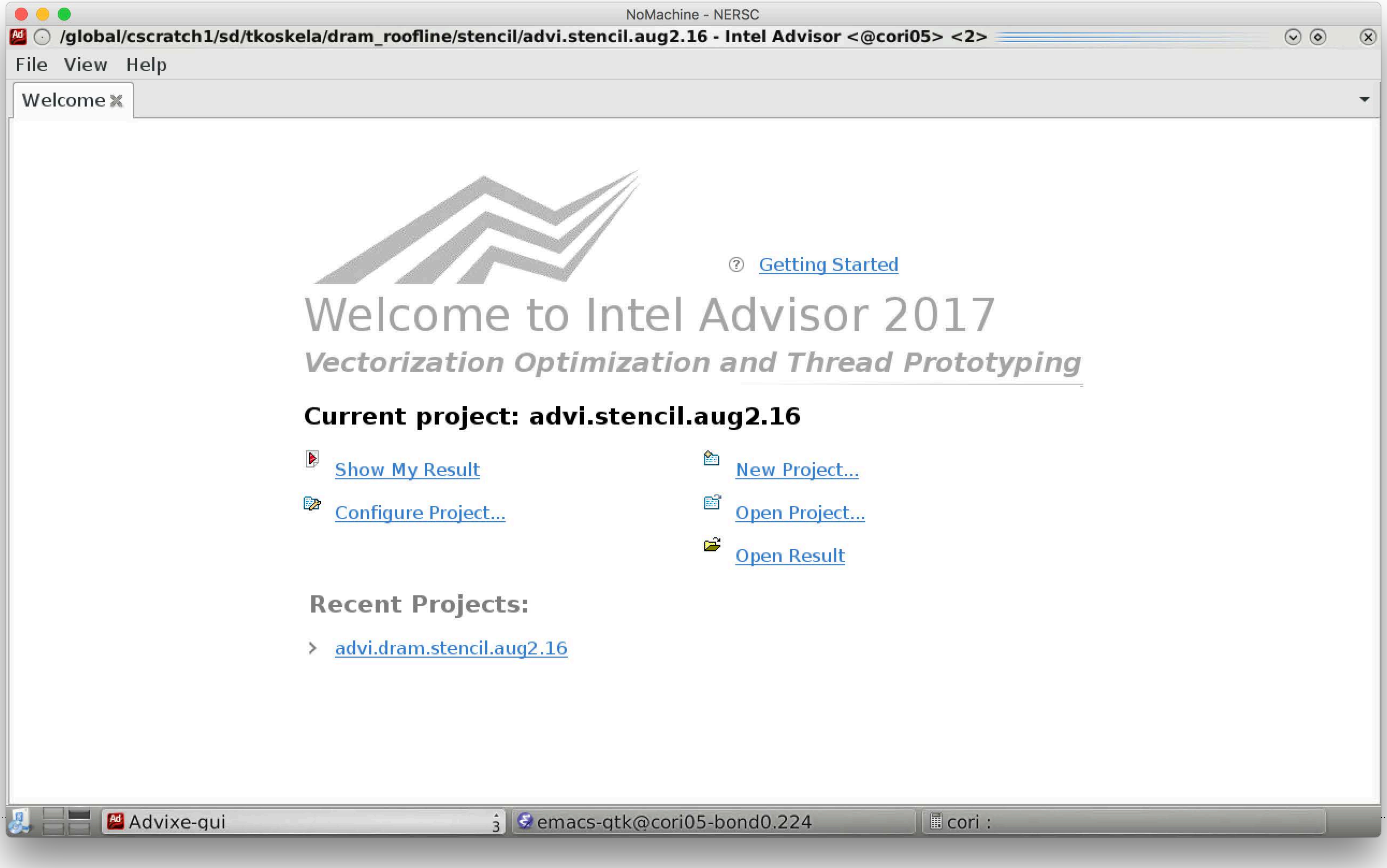
```
% salloc –perf=vtune            <<< interactive sessions; --perf only needed for DRAM Roofline
        -or-
#SBATCH –perf=vtune              <<< batch submissions; --perf only needed for DRAM Roofline
```

Benchmark…

```
% module load advisor
% export ADVIXE_EXPERIMENTAL=roofline_ex        <<< only needed for DRAM Roofline
% srun [args] advixe-cl -collect survey -no-stack-stitching -project-dir $DIR -- ./a.out [args]
% srun [args] advixe-cl -collect tripcounts -flops-and-masks -callstack-flops -project-dir $DIR -- ./a.out [args]
```

- Use Advisor GUI…

```
% module load advisor
% export ADVIXE_EXPERIMENTAL=roofline_ex        <<< only needed for DRAM Roofline
% advixe-gui $DIR
```



NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

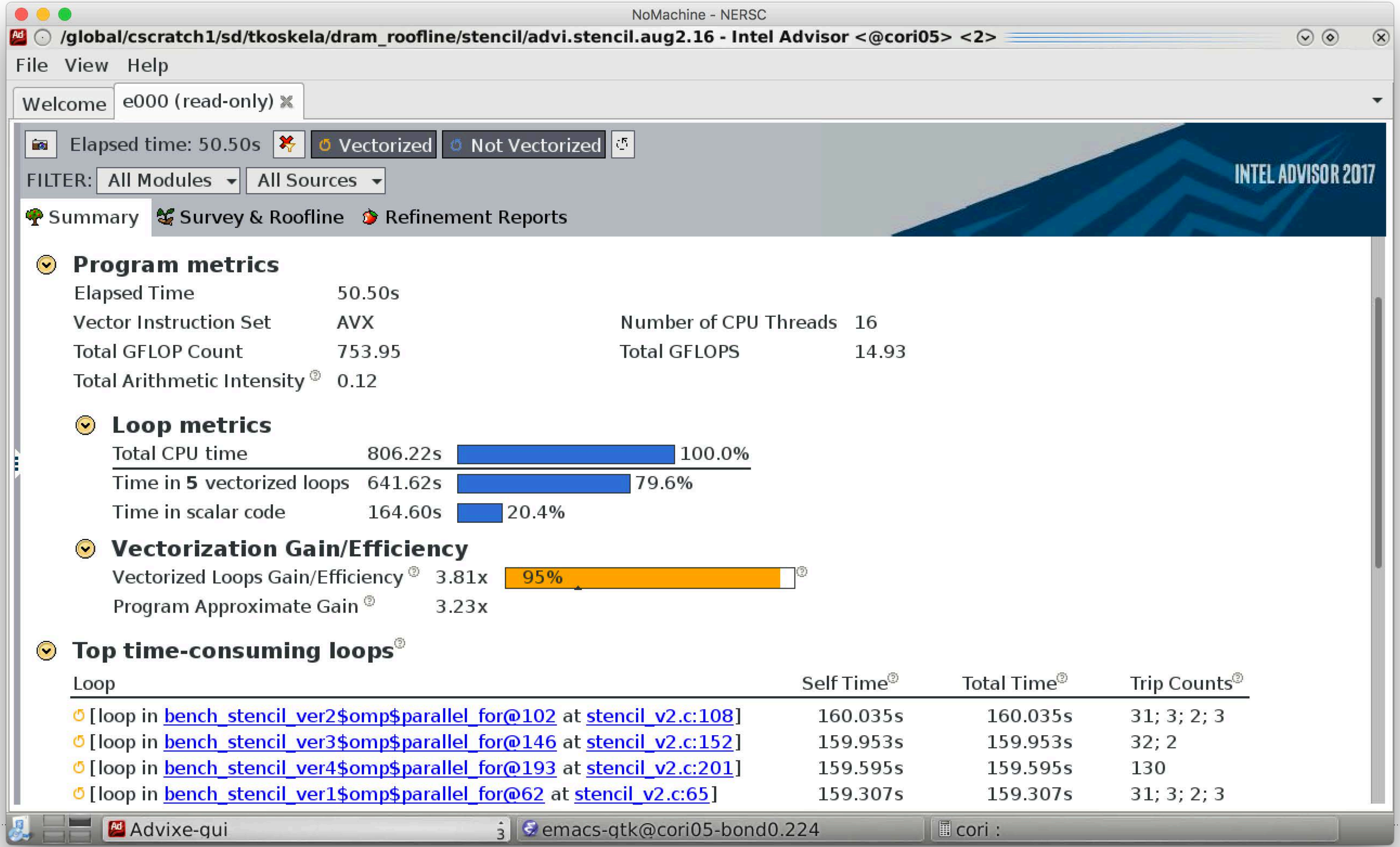
Welcome to Intel Advisor 2017

Vectorization Optimization and Thread Prototyping

Current project: advi.stencil.aug2.16

Recent Projects:

advi.dram.stencil.aug2.16



Elapsed time: 50.50s

FILTER: All Modules | All Sources

Program metrics

Elapsed Time	50.50s		
Vector Instruction Set	AVX	Number of CPU Threads	16
Total GFLOP Count	753.95	Total GFLOPS	14.93
Total Arithmetic Intensity	0.12		

Loop metrics

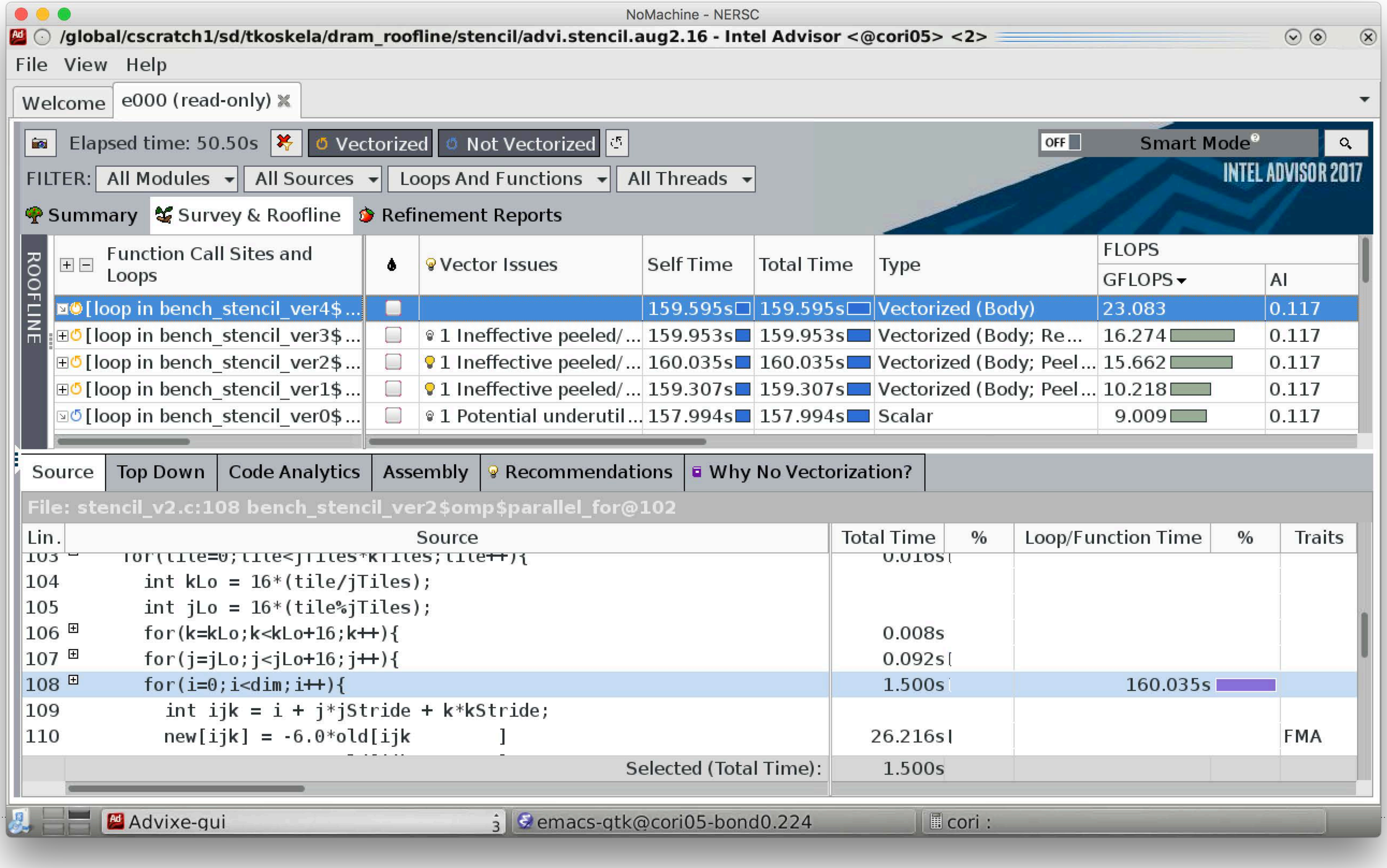
Total CPU time	806.22s	100.0%
Time in **5** vectorized loops	641.62s	79.6%
Time in scalar code	164.60s	20.4%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	3.81x	95%
Program Approximate Gain	3.23x	

Top time-consuming loops

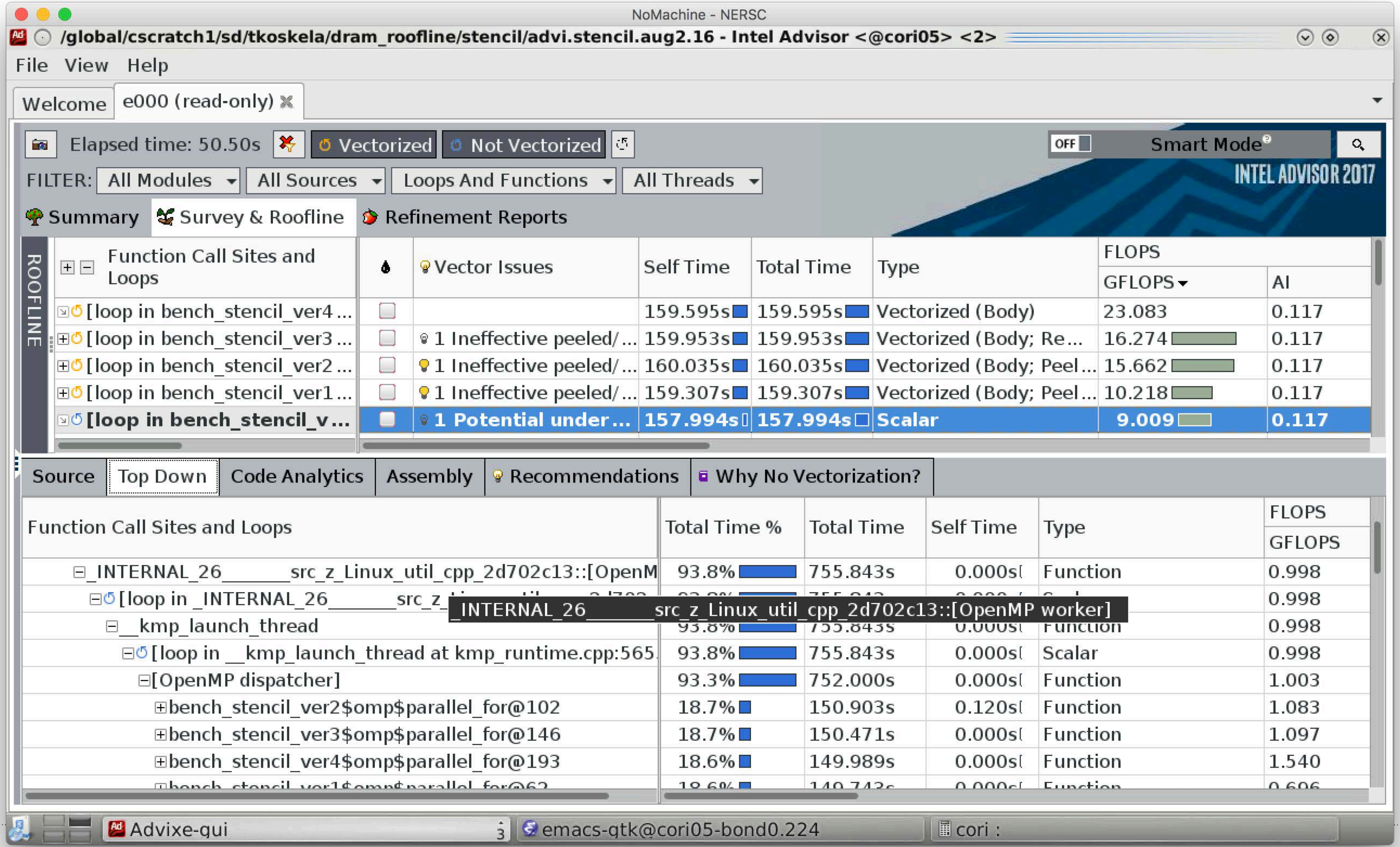
Loop	Self Time	Total Time	Trip Counts
[loop in bench_stencil_ver2$omp$parallel_for@102 at stencil_v2.c:108]	160.035s	160.035s	31; 3; 2; 3
[loop in bench_stencil_ver3$omp$parallel_for@146 at stencil_v2.c:152]	159.953s	159.953s	32; 2
[loop in bench_stencil_ver4$omp$parallel_for@193 at stencil_v2.c:201]	159.595s	159.595s	130
[loop in bench_stencil_ver1$omp$parallel_for@62 at stencil_v2.c:65]	159.307s	159.307s	31; 3; 2; 3



Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	FLOPS	
						GFLOPS	AI
[loop in bench_stencil_ver4$...	□		159.595s	159.595s	Vectorized (Body)	23.083	0.117
[loop in bench_stencil_ver3$...	□	1 Ineffective peeled/...	159.953s	159.953s	Vectorized (Body; Re...	16.274	0.117
[loop in bench_stencil_ver2$...	□	1 Ineffective peeled/...	160.035s	160.035s	Vectorized (Body; Peel...	15.662	0.117
[loop in bench_stencil_ver1$...	□	1 Ineffective peeled/...	159.307s	159.307s	Vectorized (Body; Peel...	10.218	0.117
[loop in bench_stencil_ver0$...	□	1 Potential underutil...	157.994s	157.994s	Scalar	9.009	0.117

File: stencil_v2.c:108 bench_stencil_ver2$omp$parallel_for@102

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
103	for(tile=0;tile<jTiles*kTiles;tile++){	0.016s				
104	int kLo = 16*(tile/jTiles);					
105	int jLo = 16*(tile%jTiles);					
106	for(k=kLo;k<kLo+16;k++){	0.008s				
107	for(j=jLo;j<jLo+16;j++){	0.092s				
108	for(i=0;i<dim;i++){	1.500s		160.035s		
109	int ijk = i + j*jStride + k*kStride;					
110	new[ijk] = -6.0*old[ijk]	26.216s				FMA
	Selected (Total Time):	1.500s				

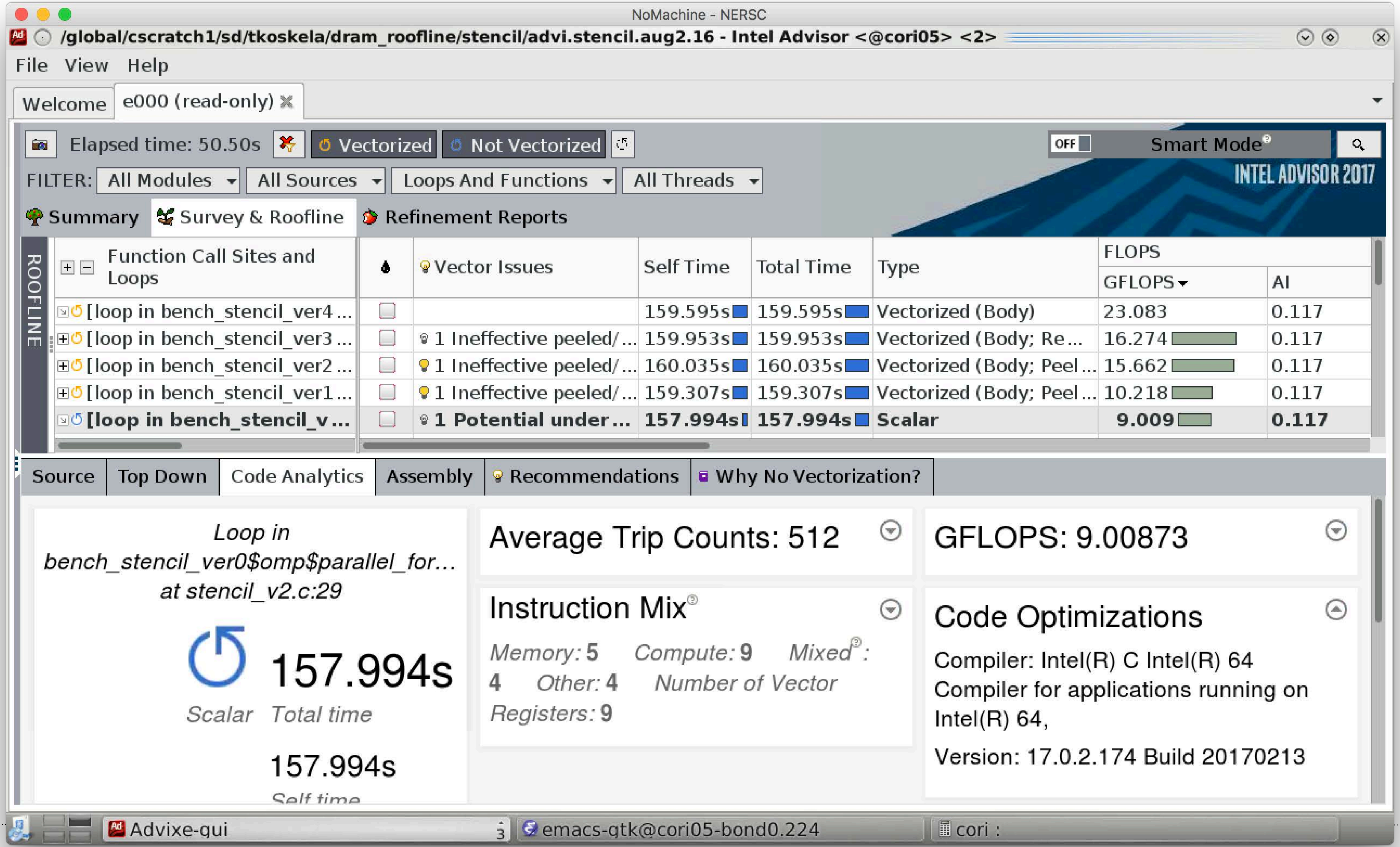


FILTER: All Modules | All Sources | Loops And Functions | All Threads

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	FLOPS	
						GFLOPS	AI
[loop in bench_stencil_ver4 ...	□		159.595s	159.595s	Vectorized (Body)	23.083	0.117
[loop in bench_stencil_ver3 ...	□	1 Ineffective peeled/...	159.953s	159.953s	Vectorized (Body; Re...	16.274	0.117
[loop in bench_stencil_ver2 ...	□	1 Ineffective peeled/...	160.035s	160.035s	Vectorized (Body; Peel...	15.662	0.117
[loop in bench_stencil_ver1 ...	□	1 Ineffective peeled/...	159.307s	159.307s	Vectorized (Body; Peel...	10.218	0.117
[loop in bench_stencil_v...	□	**1 Potential under...**	**157.994s**	**157.994s**	**Scalar**	**9.009**	**0.117**

Function Call Sites and Loops	Total Time %	Total Time	Self Time	Type	FLOPS
					GFLOPS
_INTERNAL_26______src_z_Linux_util_cpp_2d702c13::[OpenM	93.8%	755.843s	0.000s	Function	0.998
[loop in _INTERNAL_26______src_z_	93.8%	755.843s	0.000s	Scalar	0.998
__kmp_launch_thread	93.8%	755.843s	0.000s	Function	0.998
[loop in __kmp_launch_thread at kmp_runtime.cpp:565	93.8%	755.843s	0.000s	Scalar	0.998
[OpenMP dispatcher]	93.3%	752.000s	0.000s	Function	1.003
bench_stencil_ver2$omp$parallel_for@102	18.7%	150.903s	0.120s	Function	1.083
bench_stencil_ver3$omp$parallel_for@146	18.7%	150.471s	0.000s	Function	1.097
bench_stencil_ver4$omp$parallel_for@193	18.6%	149.989s	0.000s	Function	1.540
bench_stencil_ver1$omp$parallel_for@62	18.6%	149.743s	0.000s	Function	0.696

_INTERNAL_26______src_z_Linux_util_cpp_2d702c13::[OpenMP worker]



Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	FLOPS	
						GFLOPS	AI
[loop in bench_stencil_ver4 ...	□		159.595s	159.595s	Vectorized (Body)	23.083	0.117
[loop in bench_stencil_ver3 ...	□	1 Ineffective peeled/ ...	159.953s	159.953s	Vectorized (Body; Re ...	16.274	0.117
[loop in bench_stencil_ver2 ...	□	1 Ineffective peeled/ ...	160.035s	160.035s	Vectorized (Body; Peel ...	15.662	0.117
[loop in bench_stencil_ver1 ...	□	1 Ineffective peeled/ ...	159.307s	159.307s	Vectorized (Body; Peel ...	10.218	0.117
[loop in bench_stencil_v ...	□	**1 Potential under ...**	**157.994s**	**157.994s**	**Scalar**	**9.009**	**0.117**

Loop in bench_stencil_ver0$omp$parallel_for... at stencil_v2.c:29

157.994s

Scalar *Total time*

157.994s

Self time

Average Trip Counts: 512

GFLOPS: 9.00873

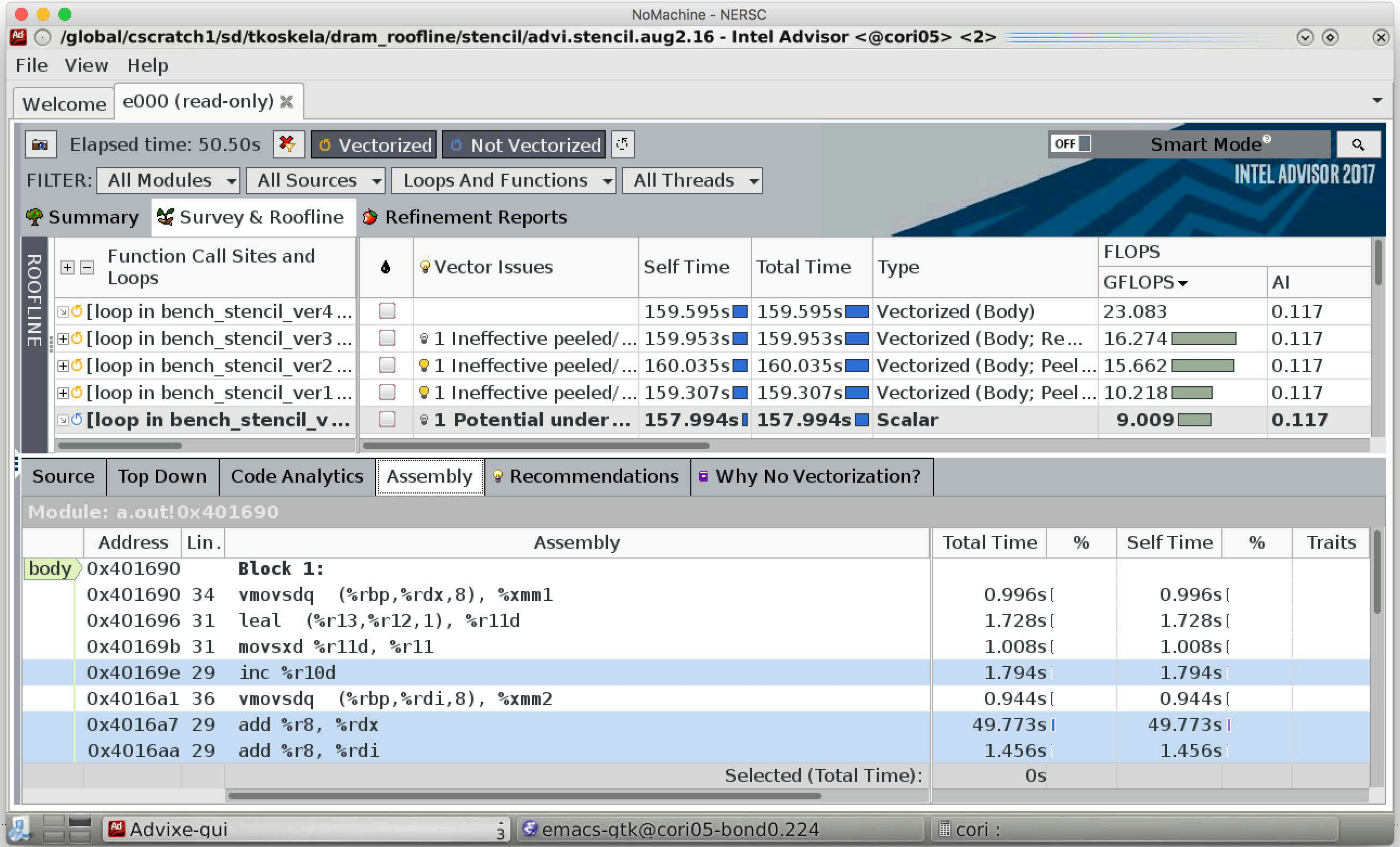
Instruction Mix

Memory: **5** *Compute:* **9** *Mixed:* **4** *Other:* **4** *Number of Vector Registers:* **9**

Code Optimizations

Compiler: Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64,

Version: 17.0.2.174 Build 20170213



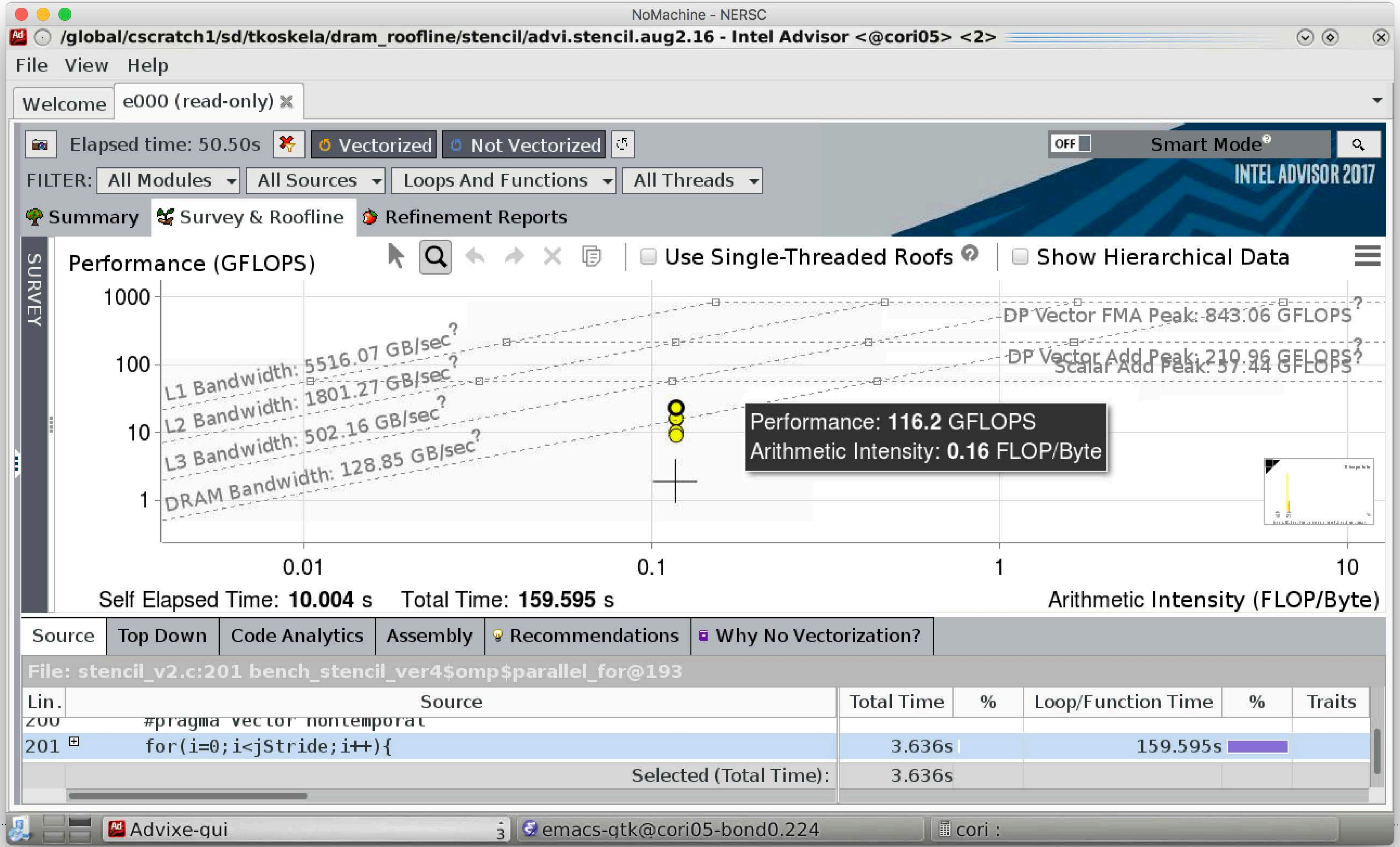
Elapsed time: 50.50s

FILTER: All Modules | All Sources | Loops And Functions | All Threads

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	FLOPS	
						GFLOPS	AI
[loop in bench_stencil_ver4 ...	□		159.595s	159.595s	Vectorized (Body)	23.083	0.117
[loop in bench_stencil_ver3 ...	□	1 Ineffective peeled/...	159.953s	159.953s	Vectorized (Body; Re...	16.274	0.117
[loop in bench_stencil_ver2 ...	□	1 Ineffective peeled/...	160.035s	160.035s	Vectorized (Body; Peel...	15.662	0.117
[loop in bench_stencil_ver1 ...	□	1 Ineffective peeled/...	159.307s	159.307s	Vectorized (Body; Peel...	10.218	0.117
[loop in bench_stencil_v...	□	**1 Potential under...**	**157.994s**	**157.994s**	**Scalar**	**9.009**	**0.117**

Module: a.out!0x401690

	Address	Lin.	Assembly	Total Time	%	Self Time	%	Traits
body	0x401690		**Block 1:**					
	0x401690	34	`vmovsdq (%rbp,%rdx,8), %xmm1`	0.996s		0.996s		
	0x401696	31	`leal (%r13,%r12,1), %r11d`	1.728s		1.728s		
	0x40169b	31	`movsxd %r11d, %r11`	1.008s		1.008s		
	0x40169e	29	`inc %r10d`	1.794s		1.794s		
	0x4016a1	36	`vmovsdq (%rbp,%rdi,8), %xmm2`	0.944s		0.944s		
	0x4016a7	29	`add %r8, %rdx`	49.773s		49.773s		
	0x4016aa	29	`add %r8, %rdi`	1.456s		1.456s		
			Selected (Total Time):	0s				



/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

Elapsed time: 50.50s

FILTER: All Modules · All Sources · Loops And Functions · All Threads

□ Use Single-Threaded Roofs

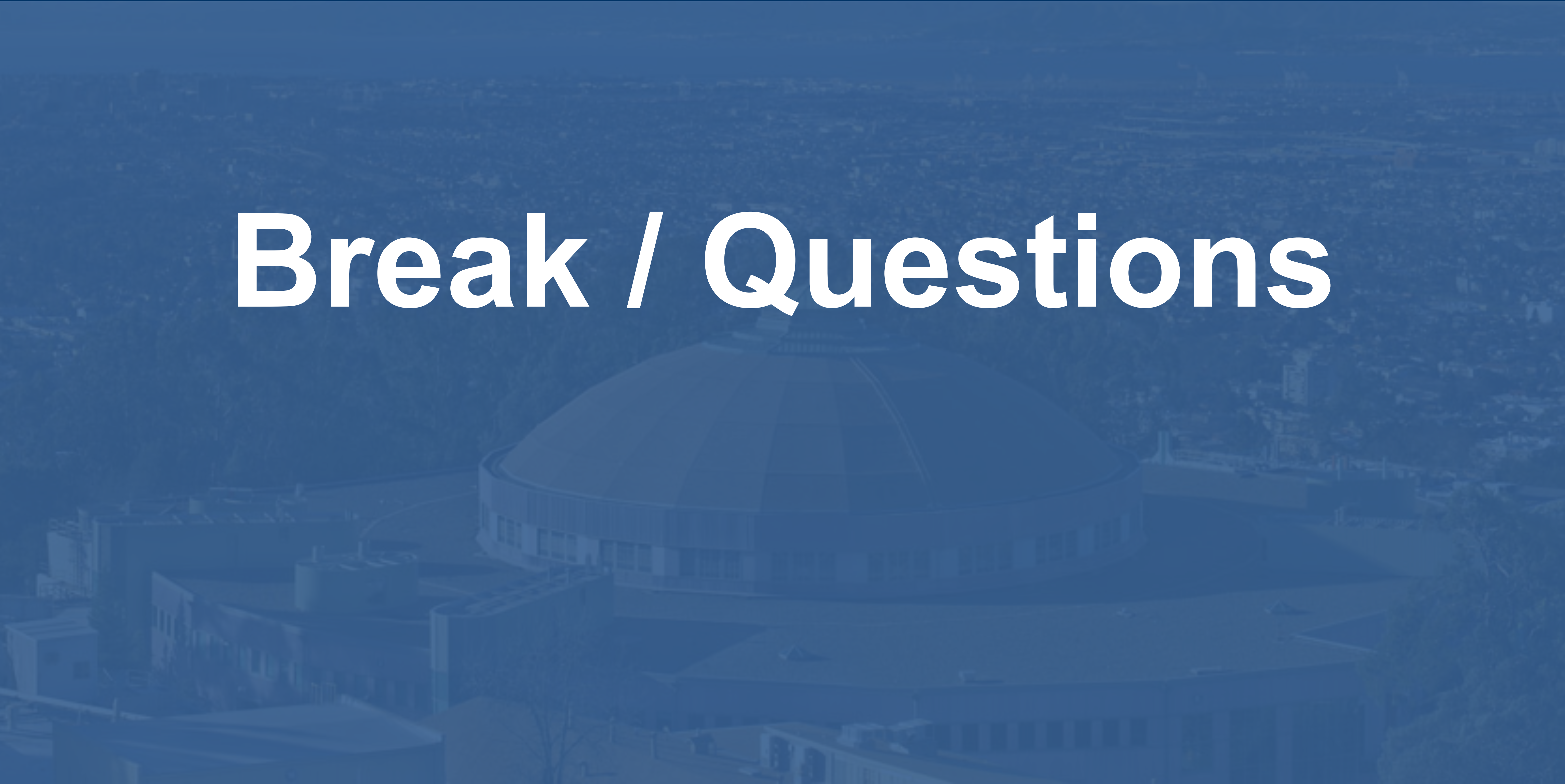
□ Show Hierarchical Data

Performance: **116.2** GFLOPS
Arithmetic Intensity: **0.16** FLOP/Byte

Self Elapsed Time: **10.004** s Total Time: **159.595** s

File: stencil_v2.c:201 bench_stencil_ver4$omp$parallel_for@193

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
200	#pragma vector nontemporal					
201	for(i=0;i<jStride;i++){	3.636s		159.595s		
	Selected (Total Time):	3.636s				



Break / Questions

Intel Advisor:

Stencil Roofline Demo*

DRAM Roofline and OS/X Advisor GUI: These are preview features that may or may not be included in mainline product releases. They may not be stable as they are prototypes incorporating very new functionality. Intel provides preview features in order to collect user feedback and plans further development and productization steps based on the feedback.

7-point, Constant-Coefficient Stencil

- Apply to a 512^3 domain on a single NUMA node (single HSW socket)
- Create 5 code variants to highlight effects (as seen in advisor)

ver0. Baseline: thread over outer loop (k), but prevent vectorization

```
#pragma novector                                 // prevent simd
int ijk = i*iStride + j*jStride + k*kStride;     // variable iStride to confuse the compiler
```

ver1. Enable vectorization

```
int ijk = i + j*jStride + k*kStride;             // unit-stride inner loop
```

ver2. Eliminate capacity misses

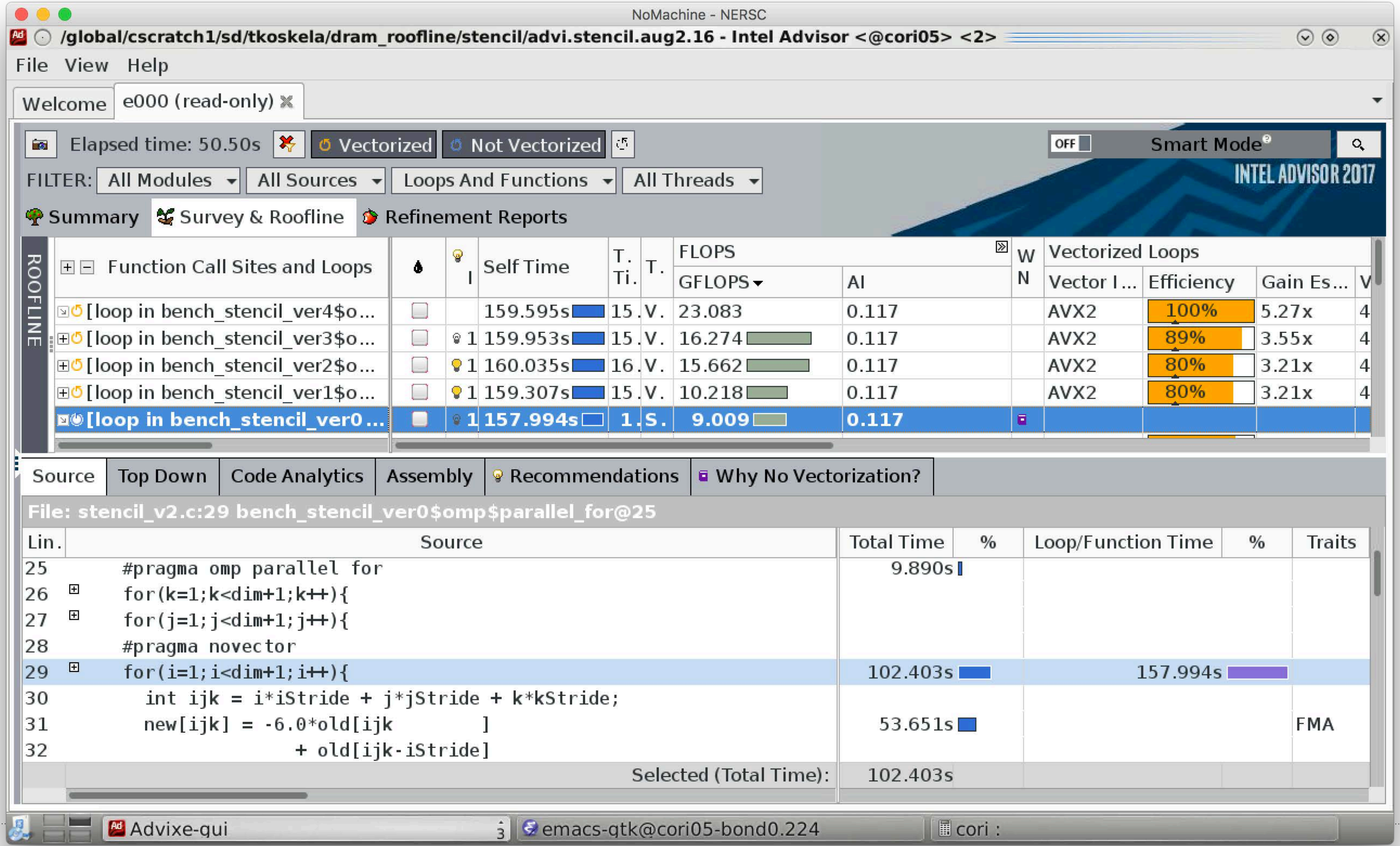
2D tiling of j-k iteration space *// working set had been O(6MB) per thread*

ver3. Improve vectorization

Provide aligned pointers and strides

ver4. Force vectorization / cache bypass

```
__assume(jstride%8 == 0);                        // stride by variable is still aligned
#pragma omp simd, vector nontemporal             // force simd; force cache bypass
```



NoMachine - NERSC

/global/cscratch1/sd/tkoskela/dram_roofline/stencil/advi.stencil.aug2.16 - Intel Advisor <@cori05> <2>

Elapsed time: 50.50s

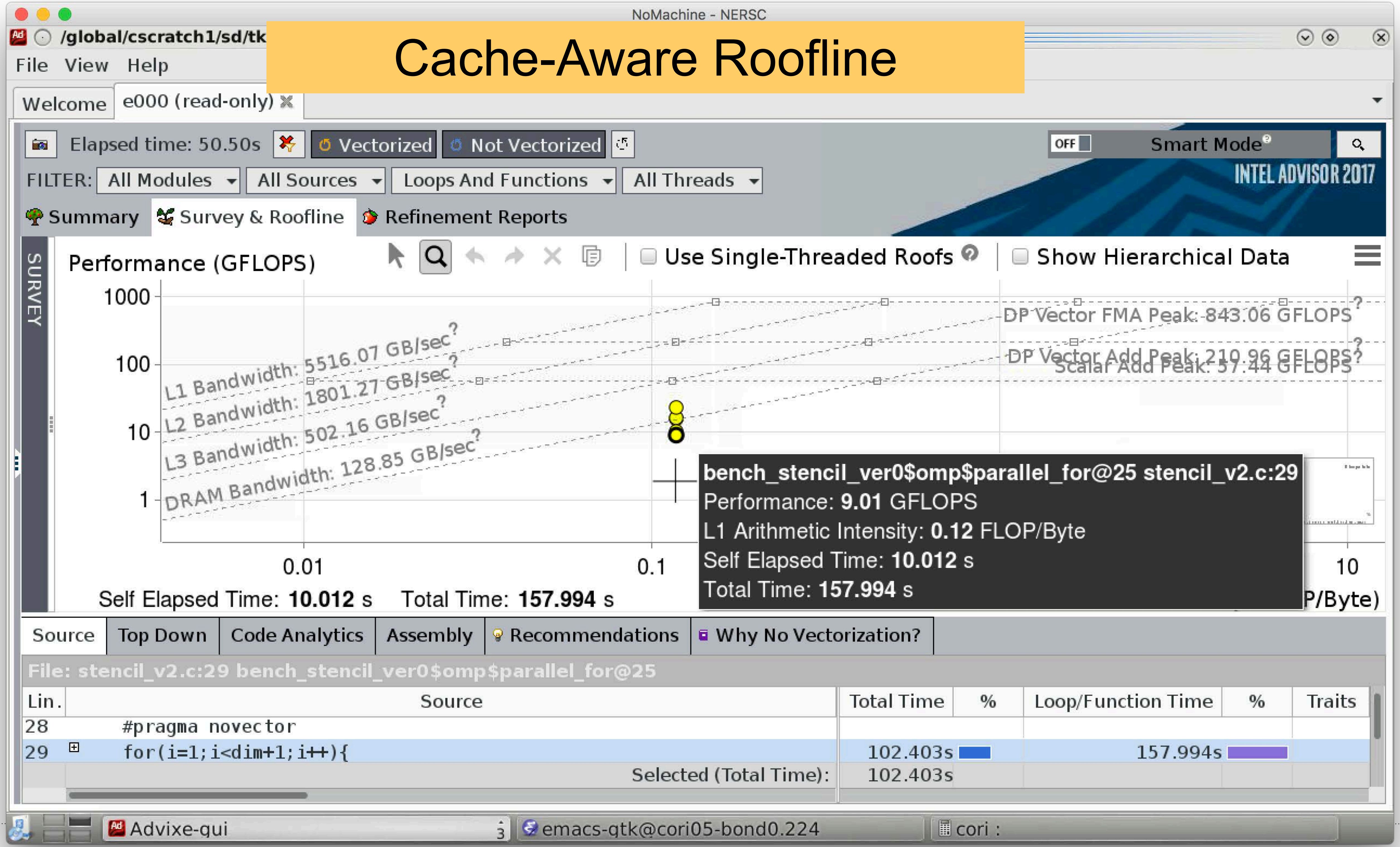
FILTER: All Modules | All Sources | Loops And Functions | All Threads

Function Call Sites and Loops		I	Self Time	T. Ti.	T.	FLOPS GFLOPS	AI	W N	Vectorized Loops Vector I...	Efficiency	Gain Es...	V
[loop in bench_stencil_ver4$o...	□		159.595s	15	V.	23.083	0.117		AVX2	100%	5.27x	4
[loop in bench_stencil_ver3$o...	□	1	159.953s	15	V.	16.274	0.117		AVX2	89%	3.55x	4
[loop in bench_stencil_ver2$o...	□	1	160.035s	16	V.	15.662	0.117		AVX2	80%	3.21x	4
[loop in bench_stencil_ver1$o...	□	1	159.307s	15	V.	10.218	0.117		AVX2	80%	3.21x	4
[loop in bench_stencil_ver0...	□	1	157.994s	1	S.	9.009	0.117	■				

File: stencil_v2.c:29 bench_stencil_ver0$omp$parallel_for@25

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
25	`#pragma omp parallel for`	9.890s				
26	`for(k=1;k<dim+1;k++){`					
27	`for(j=1;j<dim+1;j++){`					
28	`#pragma novector`					
29	`for(i=1;i<dim+1;i++){`	102.403s		157.994s		
30	`int ijk = i*iStride + j*jStride + k*kStride;`					
31	`new[ijk] = -6.0*old[ijk        ]`	53.651s				FMA
32	`+ old[ijk-iStride]`					
	Selected (Total Time):	102.403s				

Cache-Aware Roofline



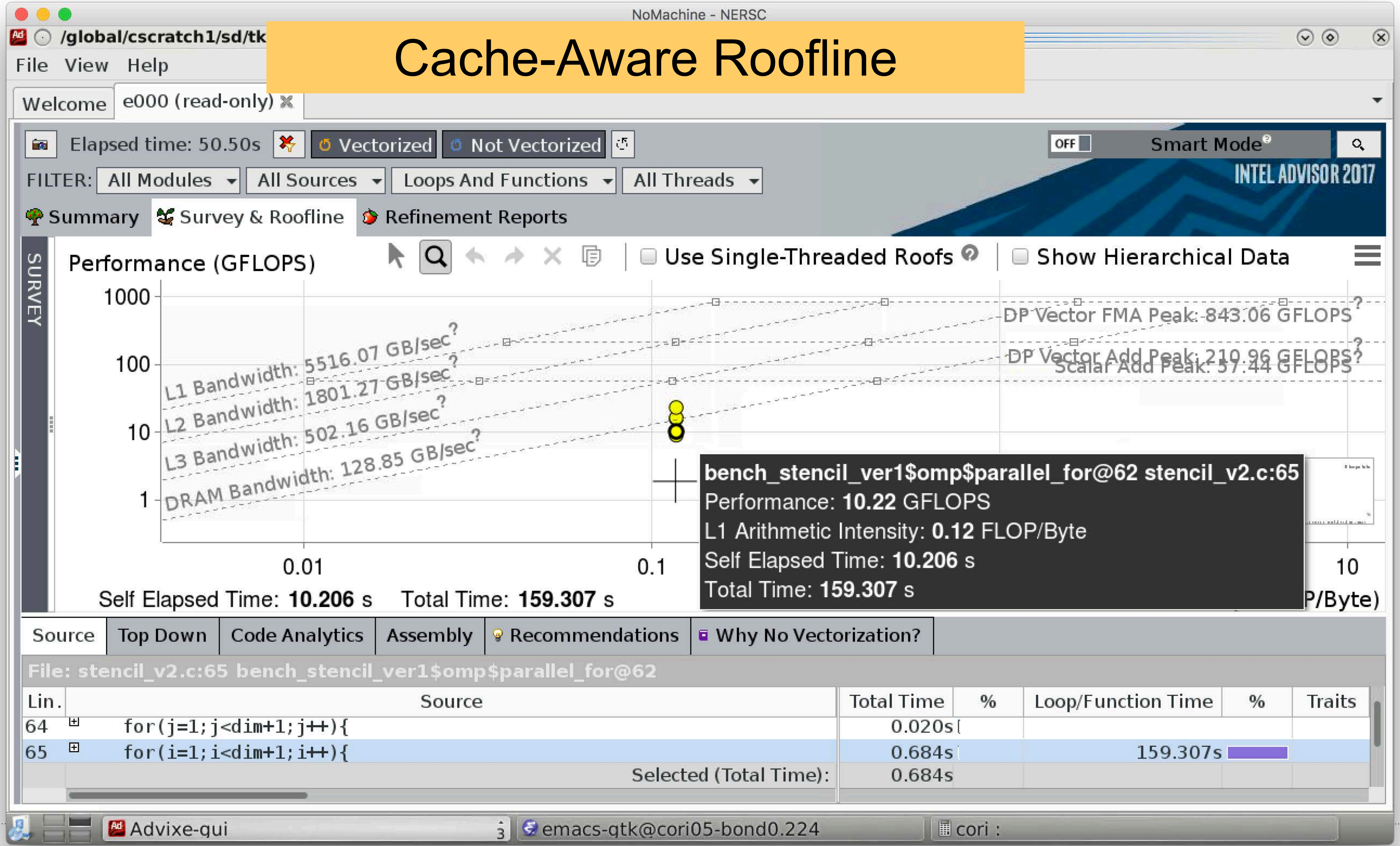
bench_stencil_ver0$omp$parallel_for@25 stencil_v2.c:29
Performance: **9.01** GFLOPS
L1 Arithmetic Intensity: **0.12** FLOP/Byte
Self Elapsed Time: **10.012** s
Total Time: **157.994** s

Self Elapsed Time: **10.012** s Total Time: **157.994** s

File: stencil_v2.c:29 bench_stencil_ver0$omp$parallel_for@25

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
28	#pragma novector					
29	for(i=1;i<dim+1;i++){	102.403s		157.994s		
	Selected (Total Time):	102.403s				

Cache-Aware Roofline

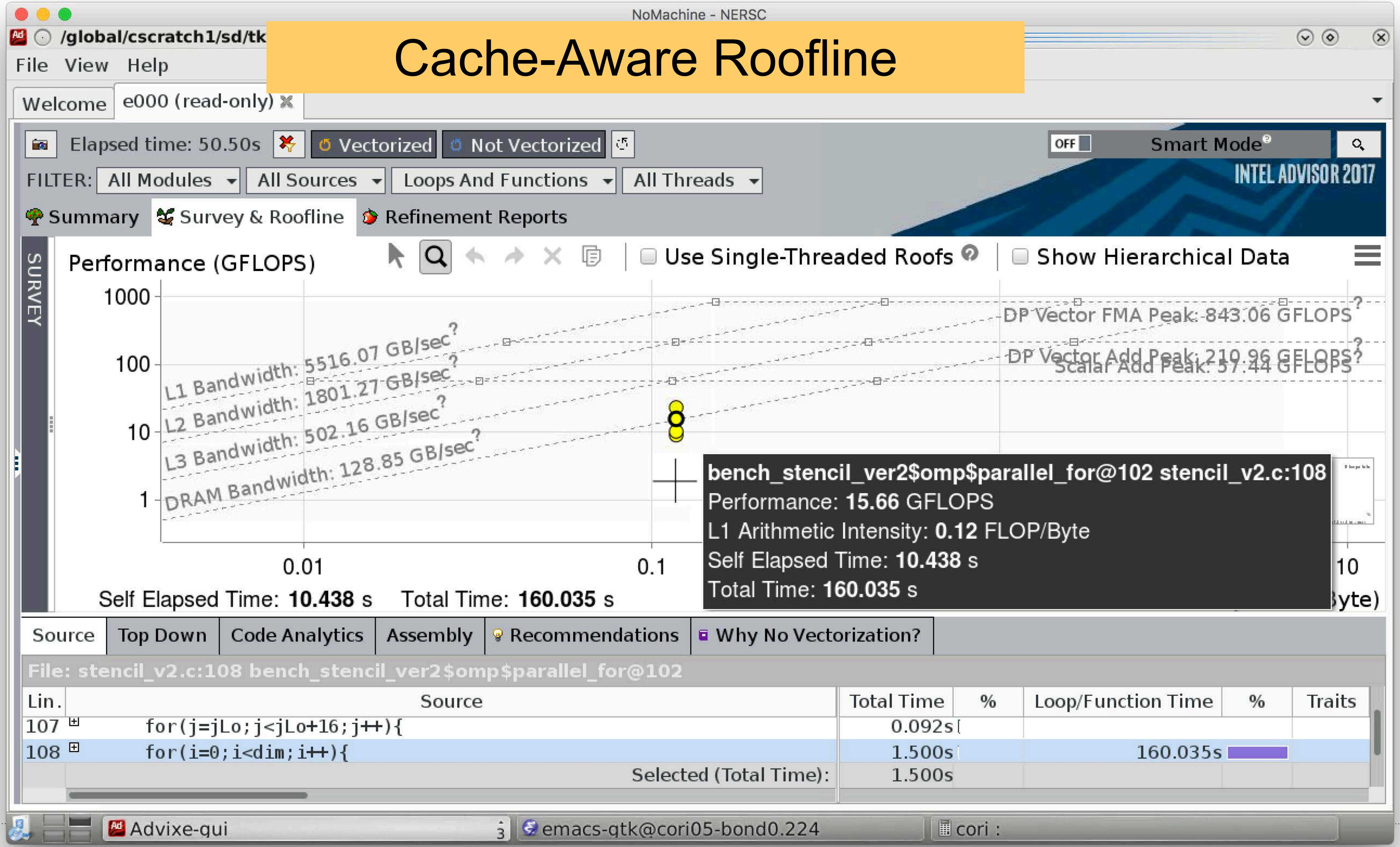


bench_stencil_ver1$omp$parallel_for@62 stencil_v2.c:65
Performance: **10.22** GFLOPS
L1 Arithmetic Intensity: **0.12** FLOP/Byte
Self Elapsed Time: **10.206** s
Total Time: **159.307** s

File: stencil_v2.c:65 bench_stencil_ver1$omp$parallel_for@62

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
64	`for(j=1;j<dim+1;j++){`	0.020s				
65	`for(i=1;i<dim+1;i++){`	0.684s		159.307s		
	Selected (Total Time):	0.684s				

Cache-Aware Roofline



bench_stencil_ver2$omp$parallel_for@102 stencil_v2.c:108

Performance: **15.66** GFLOPS

L1 Arithmetic Intensity: **0.12** FLOP/Byte

Self Elapsed Time: **10.438** s

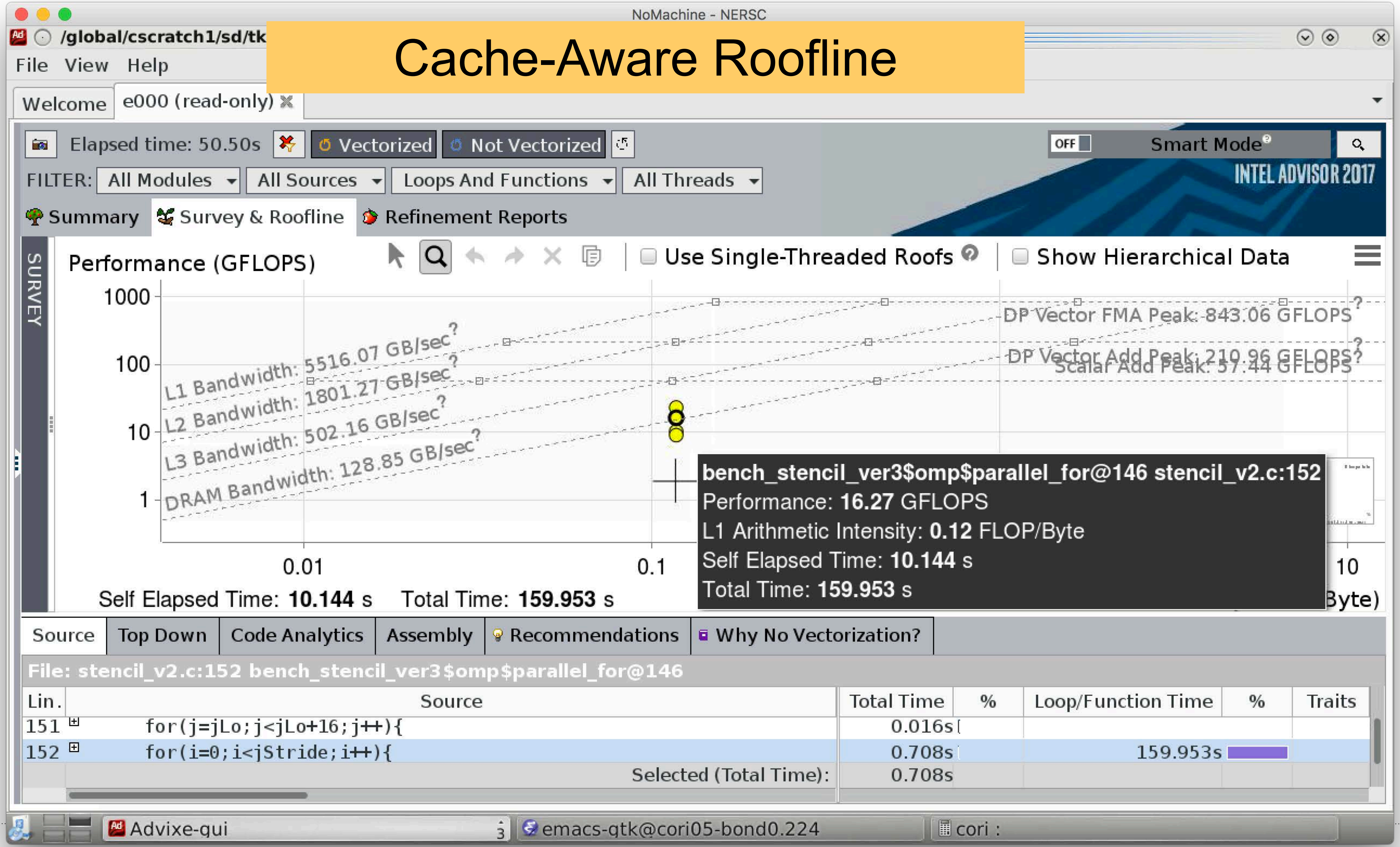
Total Time: **160.035** s

Self Elapsed Time: **10.438** s Total Time: **160.035** s

File: stencil_v2.c:108 bench_stencil_ver2$omp$parallel_for@102

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
107	`for(j=jLo;j<jLo+16;j++){`	0.092s				
108	`for(i=0;i<dim;i++){`	1.500s		160.035s		
	Selected (Total Time):	1.500s				

Cache-Aware Roofline



Elapsed time: 50.50s

FILTER: All Modules | All Sources | Loops And Functions | All Threads

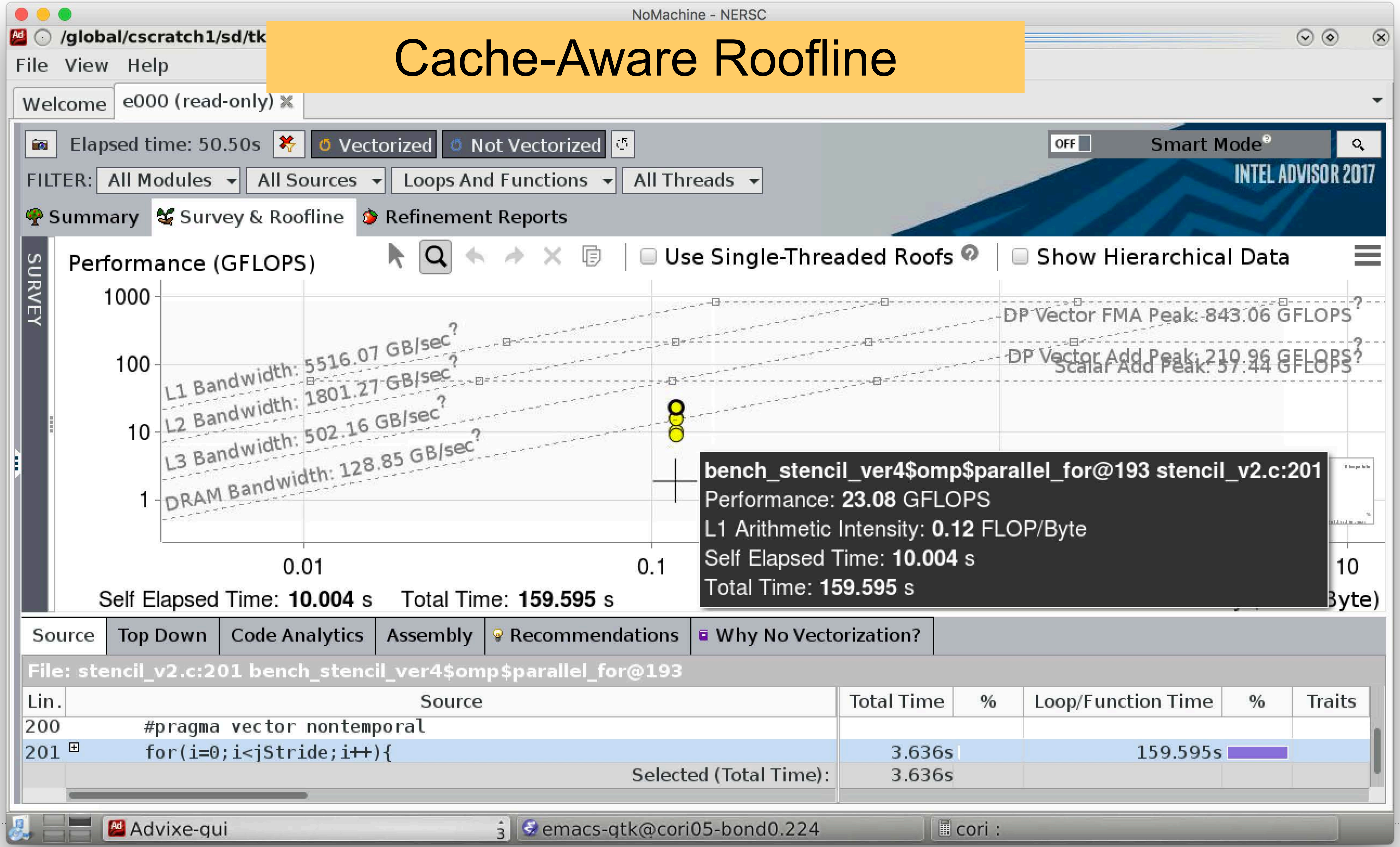
bench_stencil_ver3$omp$parallel_for@146 stencil_v2.c:152
Performance: **16.27** GFLOPS
L1 Arithmetic Intensity: **0.12** FLOP/Byte
Self Elapsed Time: **10.144** s
Total Time: **159.953** s

Self Elapsed Time: **10.144** s Total Time: **159.953** s

File: stencil_v2.c:152 bench_stencil_ver3$omp$parallel_for@146

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
151 ⊞	for(j=jLo;j<jLo+16;j++){	0.016s				
152 ⊞	for(i=0;i<jStride;i++){	0.708s		159.953s		
	Selected (Total Time):	0.708s				

Cache-Aware Roofline

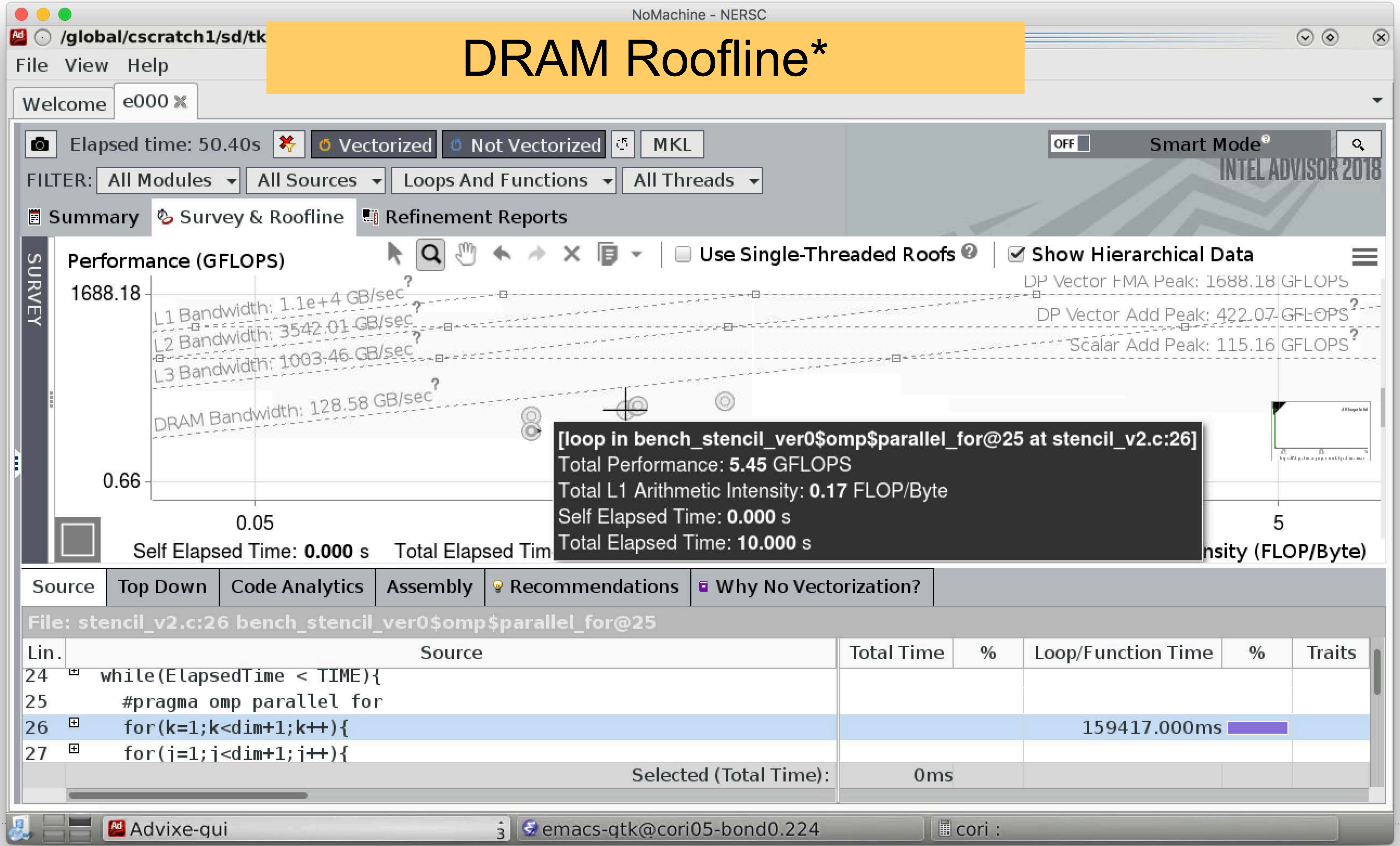


bench_stencil_ver4$omp$parallel_for@193 stencil_v2.c:201
Performance: **23.08** GFLOPS
L1 Arithmetic Intensity: **0.12** FLOP/Byte
Self Elapsed Time: **10.004** s
Total Time: **159.595** s

File: stencil_v2.c:201 bench_stencil_ver4$omp$parallel_for@193

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
200	#pragma vector nontemporal					
201	for(i=0;i<jStride;i++){	3.636s		159.595s		
	Selected (Total Time):	3.636s				

DRAM Roofline*



Elapsed time: 50.40s

FILTER: All Modules | All Sources | Loops And Functions | All Threads

☐ Use Single-Threaded Roofs

☑ Show Hierarchical Data

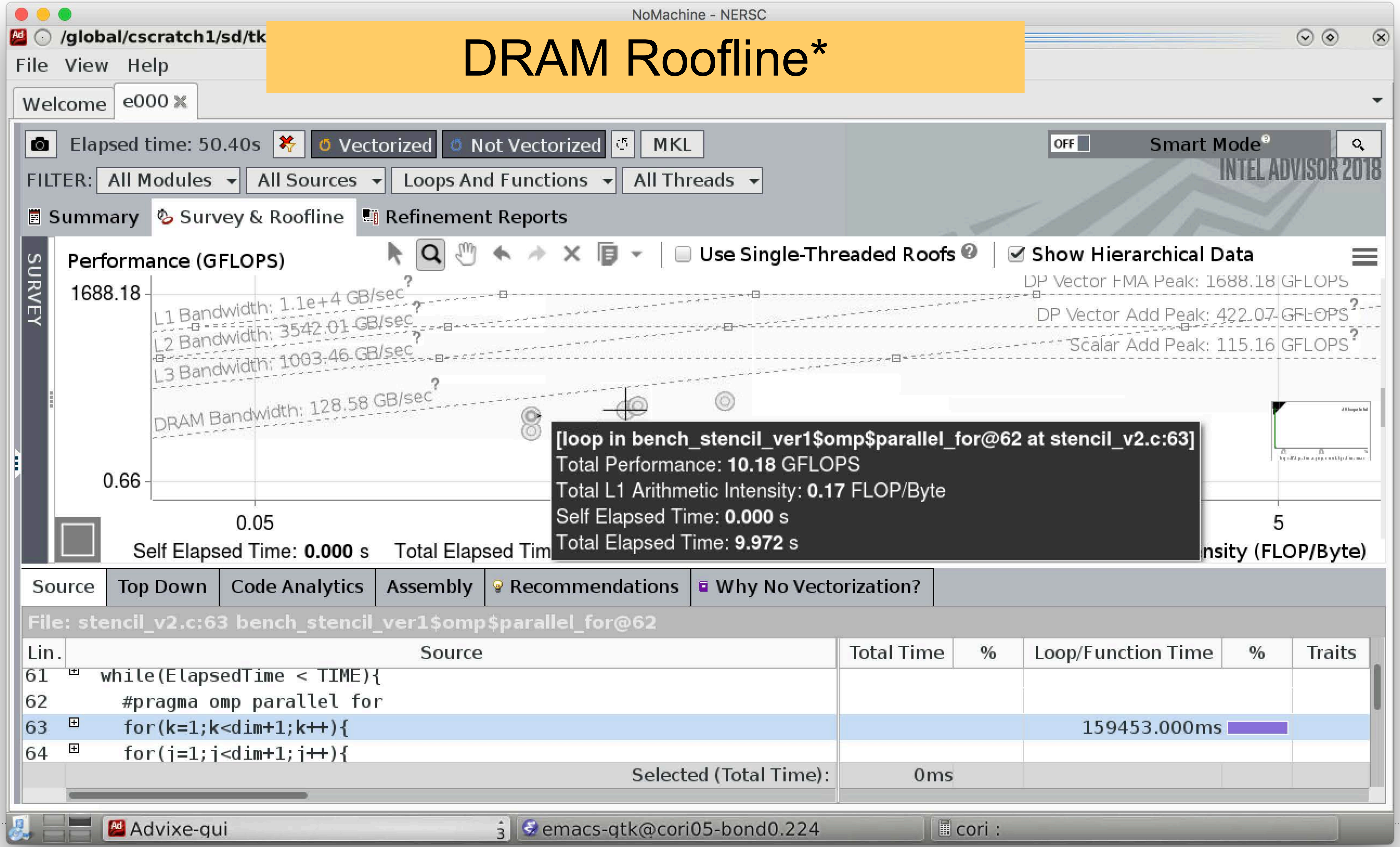
[loop in bench_stencil_ver0$omp$parallel_for@25 at stencil_v2.c:26]
Total Performance: **5.45** GFLOPS
Total L1 Arithmetic Intensity: **0.17** FLOP/Byte
Self Elapsed Time: **0.000** s
Total Elapsed Time: **10.000** s

Self Elapsed Time: **0.000** s Total Elapsed Tim

File: stencil_v2.c:26 bench_stencil_ver0$omp$parallel_for@25

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
24	`while(ElapsedTime < TIME){`					
25	`  #pragma omp parallel for`					
26	`  for(k=1;k<dim+1;k++){`			159417.000ms		
27	`  for(j=1;j<dim+1;j++){`					
	Selected (Total Time):	0ms				

DRAM Roofline*

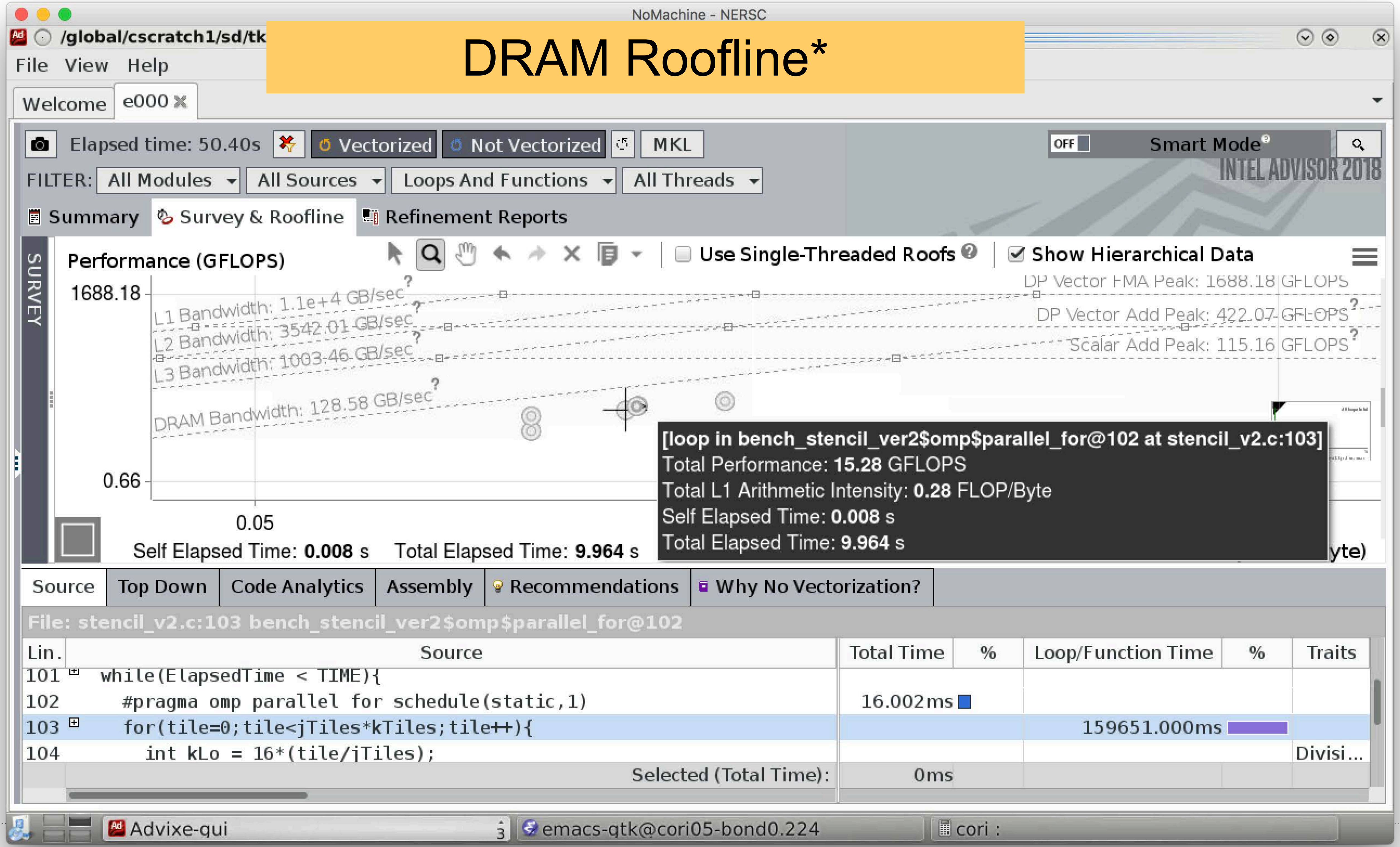


[loop in bench_stencil_ver1$omp$parallel_for@62 at stencil_v2.c:63]
Total Performance: **10.18** GFLOPS
Total L1 Arithmetic Intensity: **0.17** FLOP/Byte
Self Elapsed Time: **0.000** s
Total Elapsed Time: **9.972** s

File: stencil_v2.c:63 bench_stencil_ver1$omp$parallel_for@62

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
61	⊞ `while(ElapsedTime < TIME){`					
62	`#pragma omp parallel for`					
63	⊞ `for(k=1;k<dim+1;k++){`			159453.000ms		
64	⊞ `for(j=1;j<dim+1;j++){`					
	Selected (Total Time):	0ms				

DRAM Roofline*



Elapsed time: 50.40s

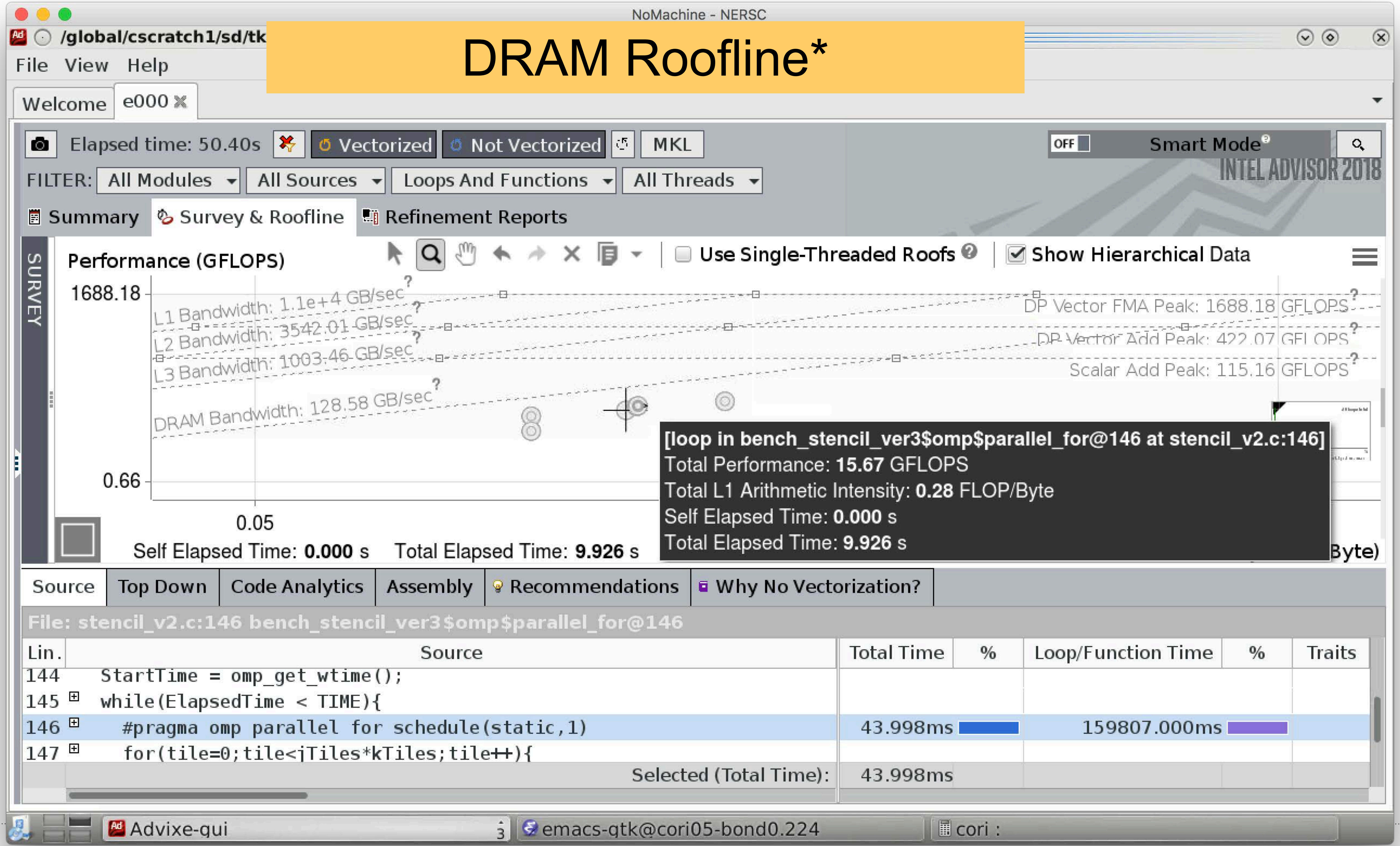
[loop in bench_stencil_ver2$omp$parallel_for@102 at stencil_v2.c:103]
Total Performance: **15.28** GFLOPS
Total L1 Arithmetic Intensity: **0.28** FLOP/Byte
Self Elapsed Time: **0.008** s
Total Elapsed Time: **9.964** s

Self Elapsed Time: **0.008** s Total Elapsed Time: **9.964** s

File: stencil_v2.c:103 bench_stencil_ver2$omp$parallel_for@102

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
101 ⊞	`while(ElapsedTime < TIME){`					
102	`  #pragma omp parallel for schedule(static,1)`	16.002ms				
103 ⊞	`  for(tile=0;tile<jTiles*kTiles;tile++){`			159651.000ms		
104	`    int kLo = 16*(tile/jTiles);`					Divisi...
	Selected (Total Time):	0ms				

DRAM Roofline*



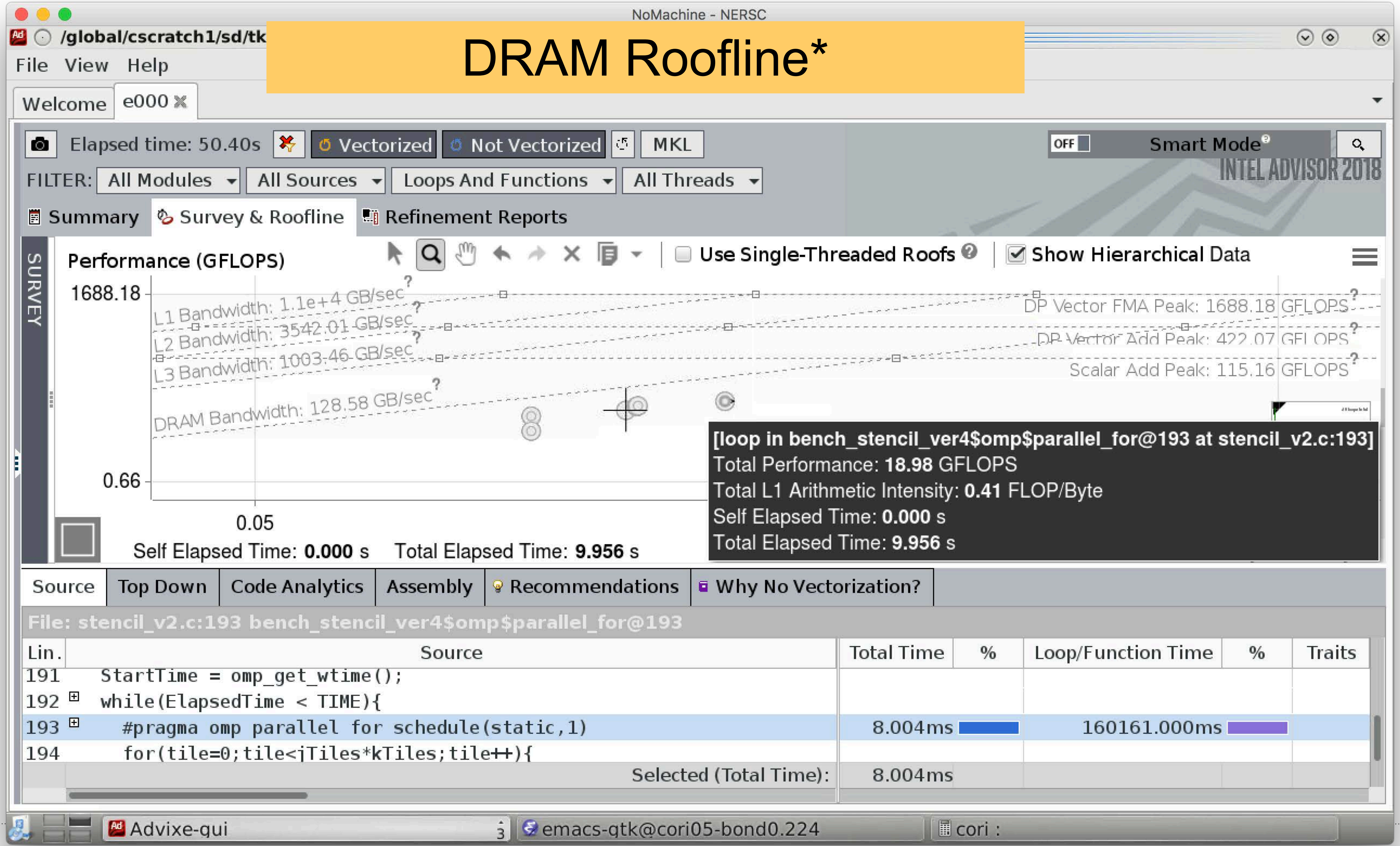
Self Elapsed Time: **0.000** s Total Elapsed Time: **9.926** s

[loop in bench_stencil_ver3$omp$parallel_for@146 at stencil_v2.c:146]
Total Performance: **15.67** GFLOPS
Total L1 Arithmetic Intensity: **0.28** FLOP/Byte
Self Elapsed Time: **0.000** s
Total Elapsed Time: **9.926** s

File: stencil_v2.c:146 bench_stencil_ver3$omp$parallel_for@146

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
144	`StartTime = omp_get_wtime();`					
145	`while(ElapsedTime < TIME){`					
146	`#pragma omp parallel for schedule(static,1)`	43.998ms		159807.000ms		
147	`for(tile=0;tile<jTiles*kTiles;tile++){`					
	Selected (Total Time):	43.998ms				

DRAM Roofline*


NoMachine - NERSC
/global/cscratch1/sd/tk
File View Help
Welcome e000
Elapsed time: 50.40s
Vectorized
Not Vectorized
MKL
OFF Smart Mode
INTEL ADVISOR 2018
FILTER: All Modules All Sources Loops And Functions All Threads
Summary Survey & Roofline Refinement Reports
SURVEY
Performance (GFLOPS)
☐ Use Single-Threaded Roofs
☑ Show Hierarchical Data
1688.18
0.66
0.05
L1 Bandwidth: 1.1e+4 GB/sec
L2 Bandwidth: 3542.01 GB/sec
L3 Bandwidth: 1003.46 GB/sec
DRAM Bandwidth: 128.58 GB/sec
DP Vector FMA Peak: 1688.18 GFLOPS
DP Vector Add Peak: 422.07 GFLOPS
Scalar Add Peak: 115.16 GFLOPS
Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?
Advixe-gui
emacs-gtk@cori05-bond0.224
cori :

Self Elapsed Time: **0.000** s Total Elapsed Time: **9.956** s

[loop in bench_stencil_ver4$omp$parallel_for@193 at stencil_v2.c:193]
Total Performance: **18.98** GFLOPS
Total L1 Arithmetic Intensity: **0.41** FLOP/Byte
Self Elapsed Time: **0.000** s
Total Elapsed Time: **9.956** s

File: stencil_v2.c:193 bench_stencil_ver4$omp$parallel_for@193

Lin.	Source	Total Time	%	Loop/Function Time	%	Traits
191	`StartTime = omp_get_wtime();`					
192 ⊞	`while(ElapsedTime < TIME){`					
193 ⊞	`  #pragma omp parallel for schedule(static,1)`	8.004ms		160161.000ms		
194	`  for(tile=0;tile<jTiles*kTiles;tile++){`					
	Selected (Total Time):	8.004ms				

Wrap up / Questions

Roofline/Advisor Tutorial at SC'17

- Sunday November 12th
- 8:30am-12pm (half-day tutorial)
- multi-/manycore focus

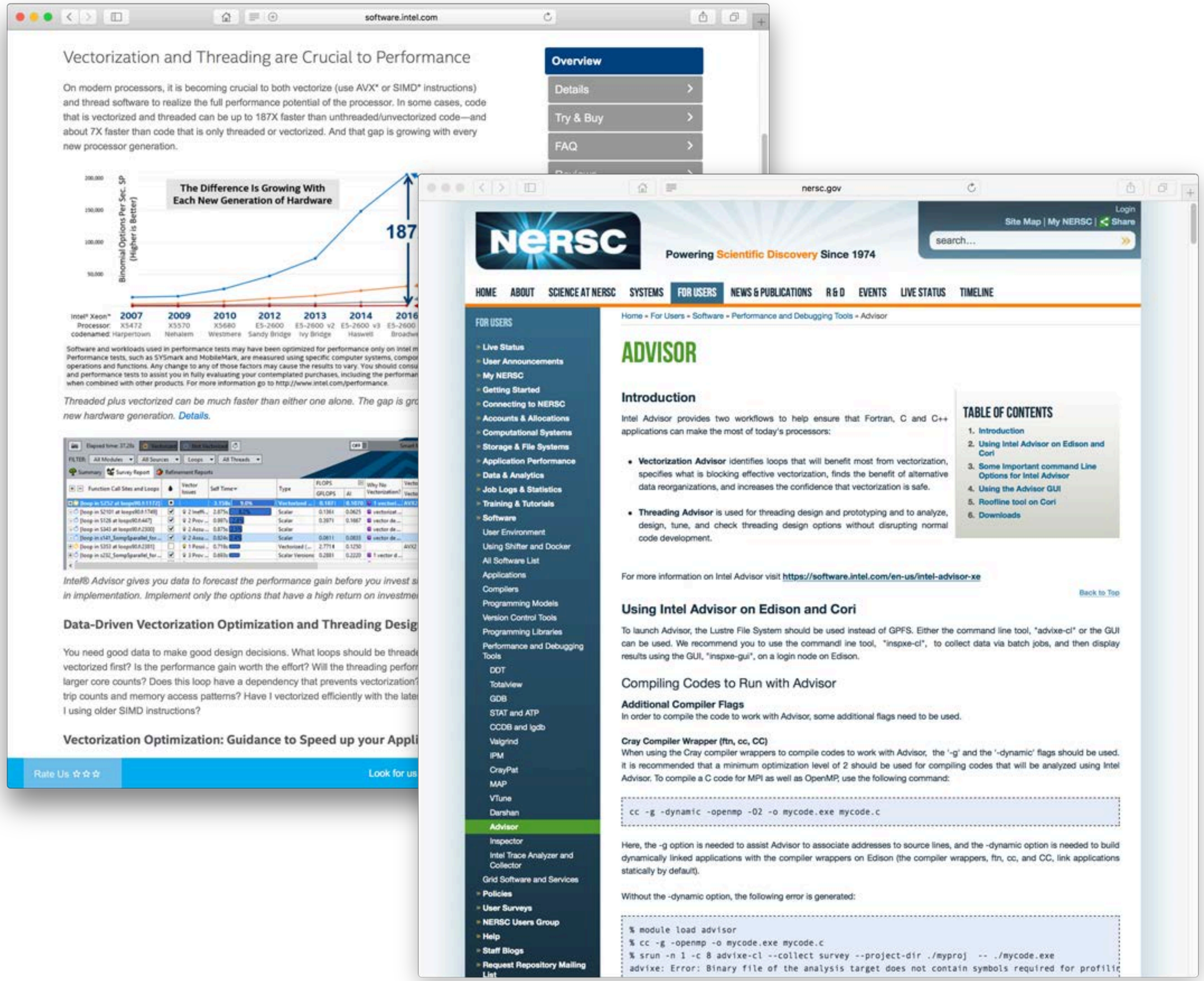
Intel Advisor (Useful Links)

Background

- https://software.intel.com/en-us/intel-advisor-xe
- https://software.intel.com/en-us/articles/getting-started-with-intel-advisor-roofline-feature
- https://www.youtube.com/watch?v=h2QEM1HpFgg

Running Advisor on NERSC Systems

- http://www.nersc.gov/users/software/performance-and-debugging-tools/advisor/



Vectorization and Threading are Crucial to Performance

On modern processors, it is becoming crucial to both vectorize (use AVX* or SIMD* instructions) and thread software to realize the full performance potential of the processor. In some cases, code that is vectorized and threaded can be up to 187X faster than unthreaded/unvectorized code—and about 7X faster than code that is only threaded or vectorized. And that gap is growing with every new processor generation.

Software and workloads used in performance tests may have been optimized for performance only on Intel m Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, compo operations and functions. Any change to any of those factors may cause the results to vary. You should consu and performance tests to assist you in fully evaluating your contemplated purchases, including the performan when combined with other products. For more information go to http://www.intel.com/performance.

Threaded plus vectorized can be much faster than either one alone. The gap is gro new hardware generation. Details.

Intel® Advisor gives you data to forecast the performance gain before you invest si in implementation. Implement only the options that have a high return on investme

Data-Driven Vectorization Optimization and Threading Desig

You need good data to make good design decisions. What loops should be threade vectorized first? Is the performance gain worth the effort? Will the threading perfor larger core counts? Does this loop have a dependency that prevents vectorization? trip counts and memory access patterns? Have I vectorized efficiently with the late I using older SIMD instructions?

Vectorization Optimization: Guidance to Speed up your Appli

Home » For Users » Software » Performance and Debugging Tools » Advisor

ADVISOR

Introduction

Intel Advisor provides two workflows to help ensure that Fortran, C and C++ applications can make the most of today's processors:

- **Vectorization Advisor** identifies loops that will benefit most from vectorization, specifies what is blocking effective vectorization, finds the benefit of alternative data reorganizations, and increases the confidence that vectorization is safe.
- **Threading Advisor** is used for threading design and prototyping and to analyze, design, tune, and check threading design options without disrupting normal code development.

TABLE OF CONTENTS

For more information on Intel Advisor visit https://software.intel.com/en-us/intel-advisor-xe

Using Intel Advisor on Edison and Cori

To launch Advisor, the Lustre File System should be used instead of GPFS. Either the command line tool, "advixe-cl" or the GUI can be used. We recommend you to use the commandl ine tool, "inspxe-cl", to collect data via batch jobs, and then display results using the GUI, "inspxe-gui", on a login node on Edison.

Compiling Codes to Run with Advisor

Additional Compiler Flags
In order to compile the code to work with Advisor, some additional flags need to be used.

Cray Compiler Wrapper (ftn, cc, CC)
When using the Cray compiler wrappers to compile codes to work with Advisor, the '-g' and the '-dynamic' flags should be used. It is recommended that a minimum optimization level of 2 should be used for compiling codes that will be analyzed using Intel Advisor. To compile a C code for MPI as well as OpenMP, use the following command:

```
cc -g -dynamic -openmp -O2 -o mycode.exe mycode.c
```

Here, the -g option is needed to assist Advisor to associate addresses to source lines, and the -dynamic option is needed to build dynamically linked applications with the compiler wrappers on Edison (the compiler wrappers, ftn, cc, and CC, link applications statically by default).

Without the -dynamic option, the following error is generated:

```
% module load advisor
% cc -g -openmp -o mycode.exe mycode.c
% srun -n 1 -c 8 advixe-cl --collect survey --project-dir ./myproj  -- ./mycode.exe
advixe: Error: Binary file of the analysis target does not contain symbols required for profilir
```

Acknowledgements

- This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.
- This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- Special Thanks to:
  - Zakhar Matveev, Intel Corporation
  - Roman Belenov, Intel Corporation