High-Performance Computing
– Performance Engineering Case Study –

Christian Plessl & Michael Lass

High-Performance IT Systems Group
Paderborn University, Germany

version 1.1.0 2018-02-02

Outline

- Vectorization
- Roofline model
- Case study: Performance optimization for n-body solver

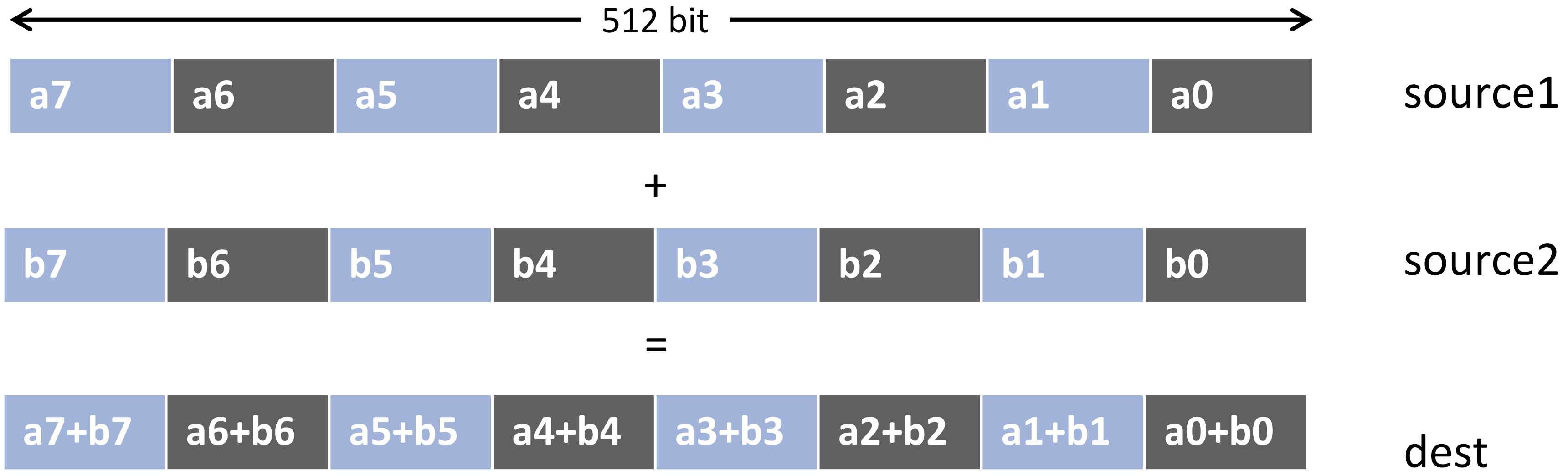
Vectorization

Vectorization

- Recent CPUs architectures have increasingly powerful SIMD, e.g. Intel:
  - width of SIMD registers
    - SSE (128bit): 2 double precision (DP) or 4 single precision (SP)
    - AVX (256bit): 4 DP / 8 SP
    - AVX512 (512bit): 8 DP / 16 SP
  - SIMD operations have become more versatile and efficient (see examples on next slides)
- Applications that do not use vectorization can only exploit a small fraction of the peak performance of modern CPUs
- We use the terms vector and SIMD instructions as synonyms; there are differences but they do not matter for this discussion

Examples for SIMD instructions

vaddpd dest, source1, source2

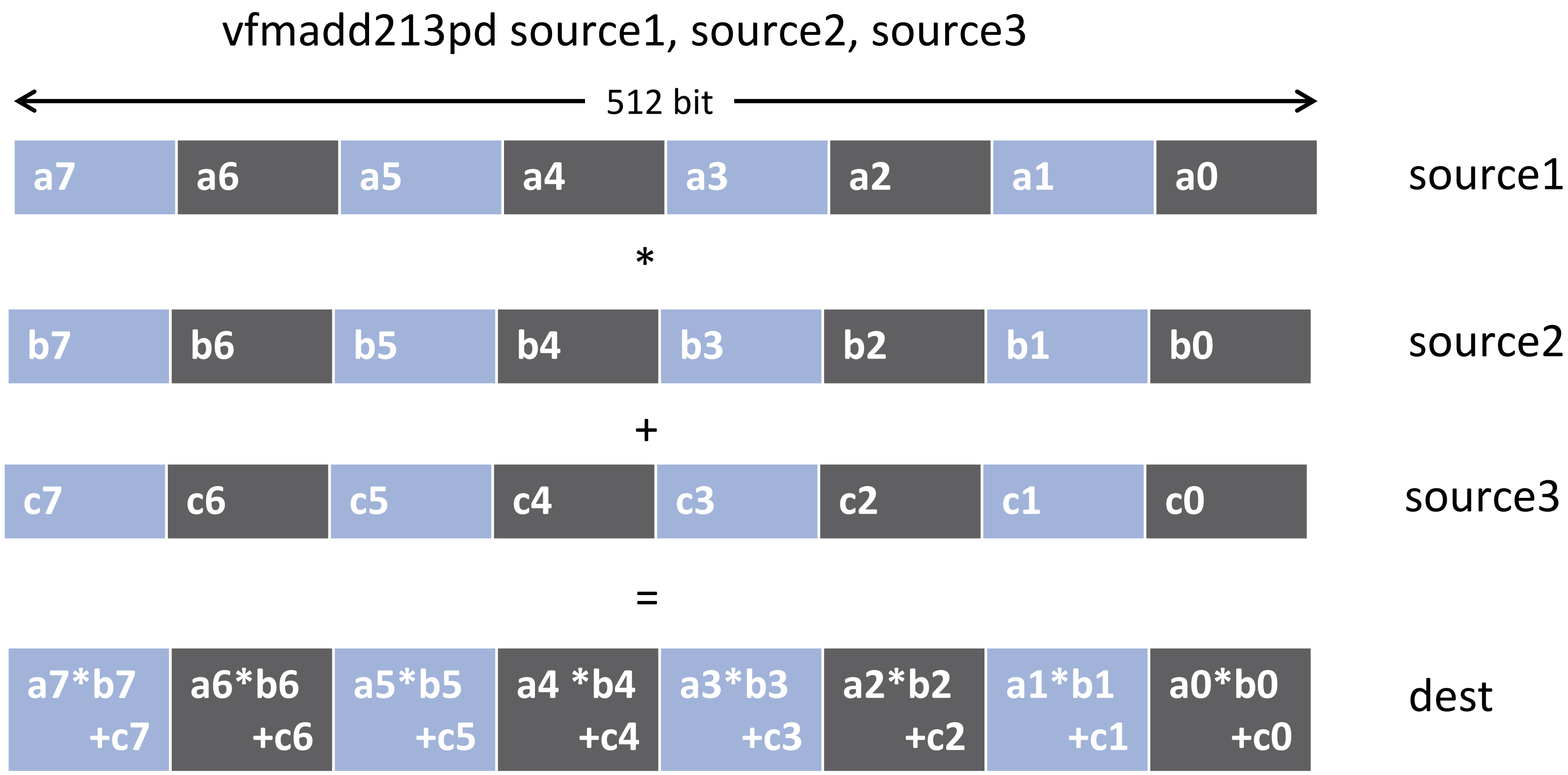


512 bit

a7	a6	a5	a4	a3	a2	a1	a0	source1
+								
b7	b6	b5	b4	b3	b2	b1	b0	source2
=								
a7+b7	a6+b6	a5+b5	a4+b4	a3+b3	a2+b2	a1+b1	a0+b0	dest

vector addition

Examples for SIMD instructions (2)



vfmadd213pd source1, source2, source3

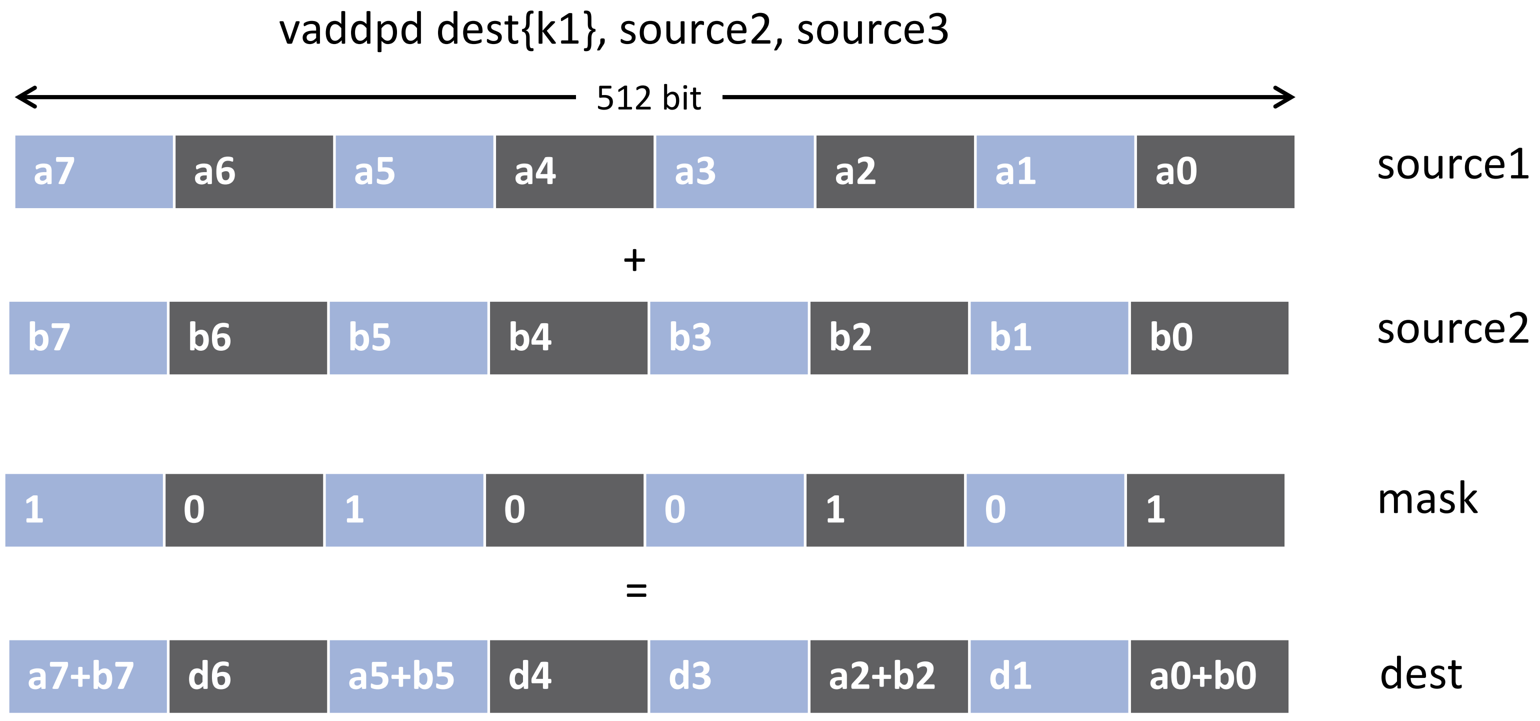
512 bit

a7	a6	a5	a4	a3	a2	a1	a0	source1
*								
b7	b6	b5	b4	b3	b2	b1	b0	source2
+								
c7	c6	c5	c4	c3	c2	c1	c0	source3
=								
a7*b7 +c7	a6*b6 +c6	a5*b5 +c5	a4 *b4 +c4	a3*b3 +c3	a2*b2 +c2	a1*b1 +c1	a0*b0 +c0	dest

fused multiply-add (FMA)

Combines two floating-point operations into a single instruction (multiplication and addition).
This instruction achieves the peak floating-point performance

Examples for SIMD instructions (3)



vaddpd dest{k1}, source2, source3

512 bit

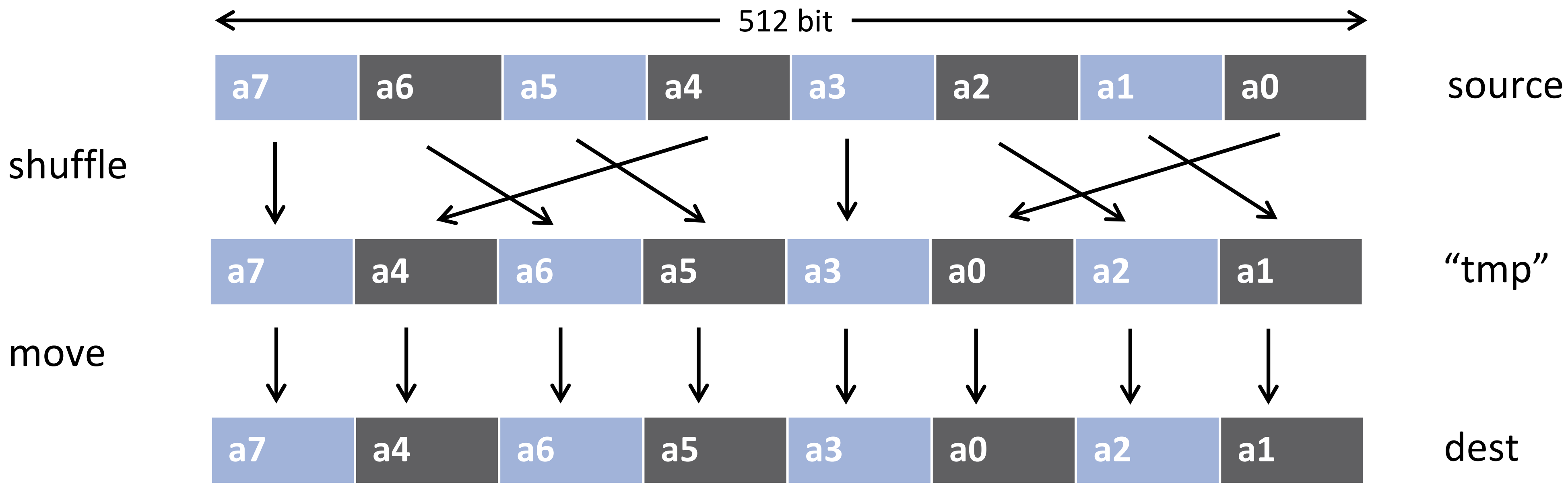
a7	a6	a5	a4	a3	a2	a1	a0	source1
+								
b7	b6	b5	b4	b3	b2	b1	b0	source2
1	0	1	0	0	1	0	1	mask
=								
a7+b7	d6	a5+b5	d4	d3	a2+b2	d1	a0+b0	dest

vector addition with masks

Can selectively write results to vector depending on mask-bit. Essential for efficient vectorization of codes with conditional execution.

Examples for SIMD instructions (4)

vpermilpd dest{k1}, source, pattern

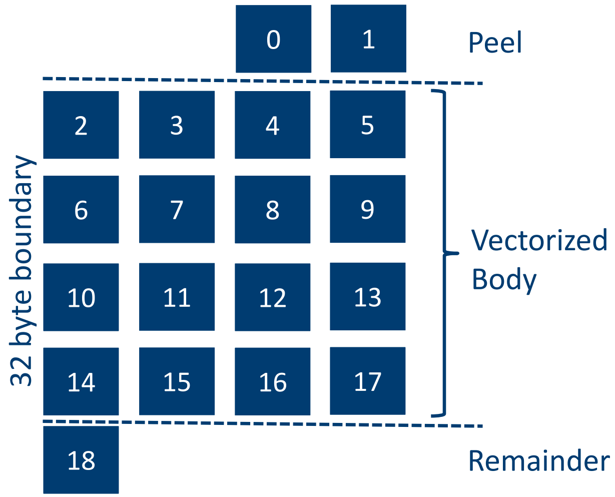


vector permutation

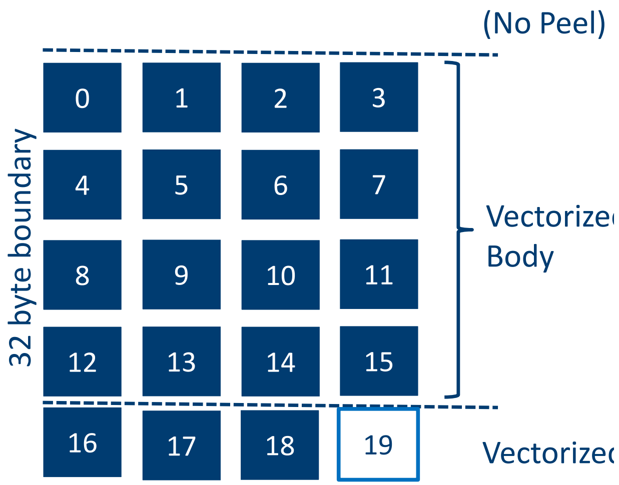
Loop Vectorization

- Vectorization is typically applied to loops
  - a couple of independent loop iterations are executed concurrently using SIMD instructions
- Challenges
  - number of loop iterations may be not evenly divisible by vector length
  - data to be processed (typ. arrays) may not be properly aligned for efficient transfer to SIMD registers
- Anatomy of a vectorized loop
  - peel (optional): used for unaligned iterations of loop, uses scalar instructions or slower SIMD instructions
  - body: fully vectorized, uses complete vector length
  - remainder (optional): process remaining iterations

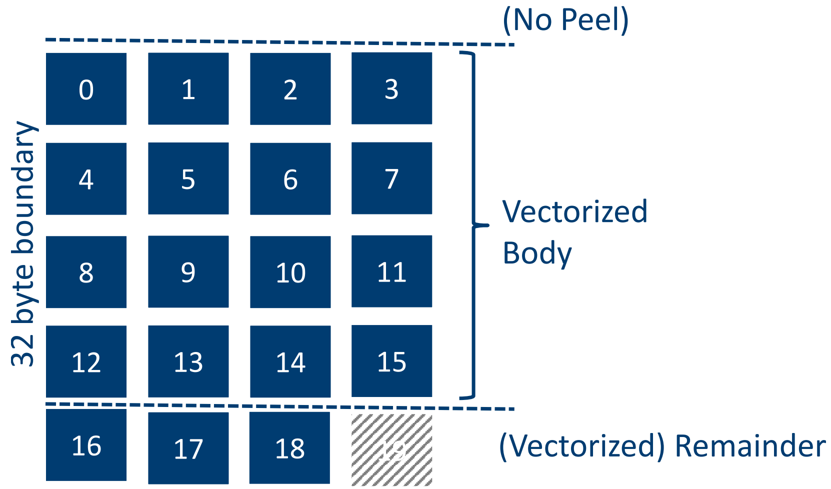
Loop Vectorization (2)



vectorized loop with peel and remainder (unaligned)



vectorized loop with remainder (aligned), remainder executed with scalar instructions



vectorized loop with remainder (aligned), remainder executed with SIMD instructions and masks

```
// xAVX
// 256 bits wide regs
// holds 4 x 64bit vals

void Func(double *pA)
{
  for (int i=0; i<19;i++)
          pA[i] = …;
}
```

How to Generate Vectorized Code

- 1. Use optimized libraries that have vectorization built-in
  - examples: Intel Math Kernel Library (MKL), AMD Core Math Library (ACML), IBM Engineering and Scientific Subroutine Library (ESSL), etc.

- 2. Auto-vectorizing compiler
  - recent compilers support automatic vectorization of code
  - compilers must make conservative assumptions to avoid breaking code
  - rewrite performance-critical code sections to make them vectorizable or to convince the compiler that vectorization is safe

- 3. Code annotations to mark sections that are safe for vectorization
  - compiler-specific directives #pragmas (e.g. ICC has #pragma ivdep)
  - OpenMP offers a portable directive #pragma omp simd

- 4. Intrinsics or assembly code

Auto Vectorization

- Most recent compilers (Intel ICC, GCC, LLVM) support automatic vectorization of regular loops
- Intel ICC
  - enable vectorization with option -vec (automatically enabled with -O3)
  - -vec-report and -opt-report generate detailed diagnostics explaining what code parts have been successfully vectorized and which parts could not be vectorized and what prevented the compiler from vectorization
  - -xHost instructs compiler to use instructions for local CPU architecture

- GCC
  - -O -ftree-vectorize enables autovectorization (automatically enabled with -O3)
  - -fopt-info-vec, -fopt-info-vec-missed generates diagnostics about successful and failed vectorization of loops
  - -march=native instructs compiler to use instructions for local CPU architecture

Obstacles to Vectorization: Non-Contiguous Memory Access

```
for (i=0; i<=MAX; i++) {
  c[i]=a[i]+b[i];
}
```

code the is ideally suited for vectorization

loading data efficiently into SIMD registers requires that data is stored contiguously in main memory

```
// arrays accessed with stride 2
for (int i=0; i<SIZE; i+=2)
b[i] += a[i] * x[i];

// inner loop accesses a with stride SIZE
for (int j=0; j<SIZE; j++) {
  for (int i=0; i<SIZE; i++) {
    b[i] += a[i][j] * x[j];
  }
}

// indirect addressing of x using index array
for (int i=0; i<SIZE; i+=2) {
  b[i] += a[i] * x[index[i]];
}
```

typical problems preventing vectorization due to non-contiguous memory access

Obstacles to Vectorization: Data Dependencies

```
A[0]=0;
for (j=1; j<MAX; j++) {
  A[j]=A[j-1]+1;
}

// equivalent to
 A[1]=A[0]+1;
 A[2]=A[1]+1;
 A[3]=A[2]+1;
 A[4]=A[3]+1;
```

Read-after-write (flow) dependency

Cannot execute several iterations concurrently, because values A[j-1] required in iteration j is known only after iteration j-1 has finished

```
for (j=1; j<MAX; j++) {
  A[j-1]=A[j]+1;
}

// equivalent to
 A[0]=A[1]+1;
 A[1]=A[2]+1;
 A[2]=A[3]+1;
 A[3]=A[4]+1;
```

Write-after-read dependency

Not safe for general parallelization but safe for vectorization! We know that no iteration with a higher value of j can be executed before iteration with lower value of j

```
for (j=1; j<MAX; j++) {
  A[j-1]=A[j]+1;
  B[j]=A[j]*2;
}

// equivalent to
 A[0]=A[1]+1;
 A[1]=A[2]+1;
 A[2]=A[3]+1;
 A[3]=A[4]+1;
```

Write-after-read dependency

Not safe for vectorization because some A[j] may be overwritten by the first SIMD instruction before they are read by the second SIMD instruction

Obstacles to Vectorization: Pointer Aliasing

```
void vadd(float *a, float *b, float *c, int n)
{
  for(int i=0; i<n; i++) {
    c[i] = a[i] + b[i];
}
```

vector add: with potential aliasing, i.e. arrays a, b, and c might partially overlap

provide meta information to compiler to enable vectorization

- a, b and c are non-overlapping → use C99 restrict keyword
- a, b, n are read-only → use const modifier

```
void vadd(const restrict float *a,
  const restrict float *b,
  const restrict float *c,
  const int n)
{
  for(int i=0; i<n; i++) {
    c[i] = a[i] + b[i];
}
```

Where Autovectorizers Tend to Fail

- Most frequent reason: data dependencies

- Further common reasons
  - potential pointer/array aliasing
  - unsuitable alignment
  - function calls in loop block
  - loop not countable (loop bound is not a runtime constant)
  - mixed data types
  - non-unit stride between elements
  - loop body too complex (register pressure)
  - profitability models deems vectorization as inefficient

- Many additional, but less likely reasons

OpenMP SIMD Constructs

The OpenMP simd Construct

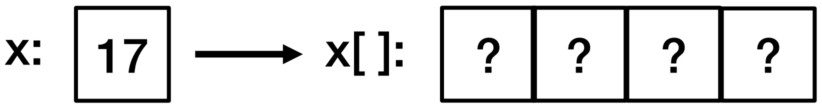
- Conceptually, vectorization is conceptually similar to loop parallelization with OpenMP work sharing or reductions
- OpenMP 4.0 introduced the simd directive
  - explicit vectorization of a loop test
  - cuts loop into chunks that fit a SIMD vector register
  - by default no parallelization, but multi-threaded worksharing variant (simd for”) is available too

```
#pragma omp simd [clauses]
for-loop
```

- Clauses are similar to for-directive

The simd Data Sharing Clauses

- `private (var-list)`
  - expand scalar variables to uninitialized vectors



x: 17 → x[ ]: ? ? ? ?

- `lastprivate(var-list)`

- `reduction(op : var-list)`
  - expand scalar variable to vector
  - compute reduction of elements of vector by applying operation op

```
double sum_all (double *a, double *b, int n)
{
  double tmp, sum;
  sum = 0.0;
  #pragma omp simd \
     private(tmp) reduction(+:sum)
  for (int i = 0; i<n; i++) {
    tmp = a[i] + b[i];
    sum += tmp;
  }
  return sum;
}
```

example: perform addition reduction on arrays a and b

Further Clause for simd Construct

- `safelen (length)`
  - maximum number of loop iterations that can be processed concurrently without breaking a dependence
- `simdlen (length)`
  - preferred number of loop iterations to be executed concurrently
  - must be less or equal to safelen with present
- `linear (list[:linear-step])`
  - the value of a variable is a linear function of the iteration number, i.e.
    $x_i = x_0 + i$ * linear-step
- `aligned (list[:alignment])`
  - the memory location of each element in the list is aligned to the number of bytes specified in the optional alignment argument
  - if alignment argument is missing, default alignment is assumed

Further Clause for simd Construct (2)

- `colapse (n)`
  - perform loop fusion, i.e. specifies how many loops are associated with the construct
  - if more then one loop is associated with simd clause, all associated loops are collapsed into one larger iteration space that is then executed with SIMD instructions

```
void work( float*b, int n, int m ){
  int i;
  #pragma omp simd safelen(16)
  for (i = m; i<n; i++) {
    b[i] = b[i-m] - 1.0f;
  }
}
```

Loop simd Construct

- Combine work sharing with vectorization
  - distribute iterations of loop across the threads in a team
  - execute each loop chunk with SIMD instructions
- `#pragma omp for simd [clauses]`
  `for-loops`

```
float sprod(float *a, float *b, int n)
{
  float sum = 0.0;
  #pragma omp for simd reduction(+:sum)
  for (int i = 0; i<n; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

example: scalar (dot) product

Function Vectorization

- Only simple functions are automatically vectorized
  - hence, function calls in OpenMP SIMD loops frequently prevent vectorization
- OpenMP allows to declare functions that can be safely called from a SIMD-parallel loop
- `#pragma omp declare simd [clauses] function-definition-or-declaration`
- Additional clause `uniform` to specify function arguments that are constant for all SIMD lanes

```
#pragma omp declare simd
float min(float a, float b) {
  return a<b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
  return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
  for (i=0; i<N; i++) {
    d[i] =min(distsq(a[i], b[i]), c[i]);
  }
}
```

Summary Vectorization

- Exploiting vectorization is key for performance on modern CPUs
- The cheapest way to vectorization is using numerical libraries and autovectorization
- OpenMP simd constructs provide a portable way to add explicit vectorization
- Vectorization is a deep topic, we have only scratched the surface here

Roofline Model

Roofline Model

- Application performance on CPUs is limited by
  - memory bandwidth, or (memory bound)
  - arithmetic performance (compute bound)
- Roofline model
  - analytical model of fundamental performance limits
  - describes an upper bound on the achievable performance based on the operational intensity (also arithmetic intensity), which is the number of operations performed per data read from DRAM (FLOPS/byte)
  - Wiliams, Waterman and Patterson: *"Roofline: An Insightful Visual Performance Model for Multicore Architectures",* Communications of the ACM, 4(52) 2009, http://dx.doi.org/10.1145/1498765.1498785
- Basic idea

$$GFLOP\ /\ \mathrm{s} = \min\begin{cases} Peak\ Computational\ Performance \\ Memory\ Bandwidth\ *\ Operational\ Intensity \end{cases}$$

Roofline Model (2)

$$GFLOP\,/\,s = \min\begin{cases} Peak\ Computational\ Performance \\ Memory\ Bandwidth\ *\ Operational\ Intensity \end{cases}$$

Computational Peak performance

Peak FLOP = 2 x 2.6 x 6 x 16 =

499 single precision FLOP/s

- 2 sockets
- 2.6 GHz
- 6 core
- 16 single precision operation per SIMD instruction (8 MUL + 8 ADD with AVX)

Memory bandwidth

Peak Mem BW = 2 x 1.6 x 8 x 4 = 102.4 GB/s

- 2 sockets
- 1.6 GHz memory frequency
- 8 bytes per channel
- 4 memory channels per CPU

2-socket Oculus Node

Roofline Model (2)

Intel Sandy Bridge Microarchitecture:

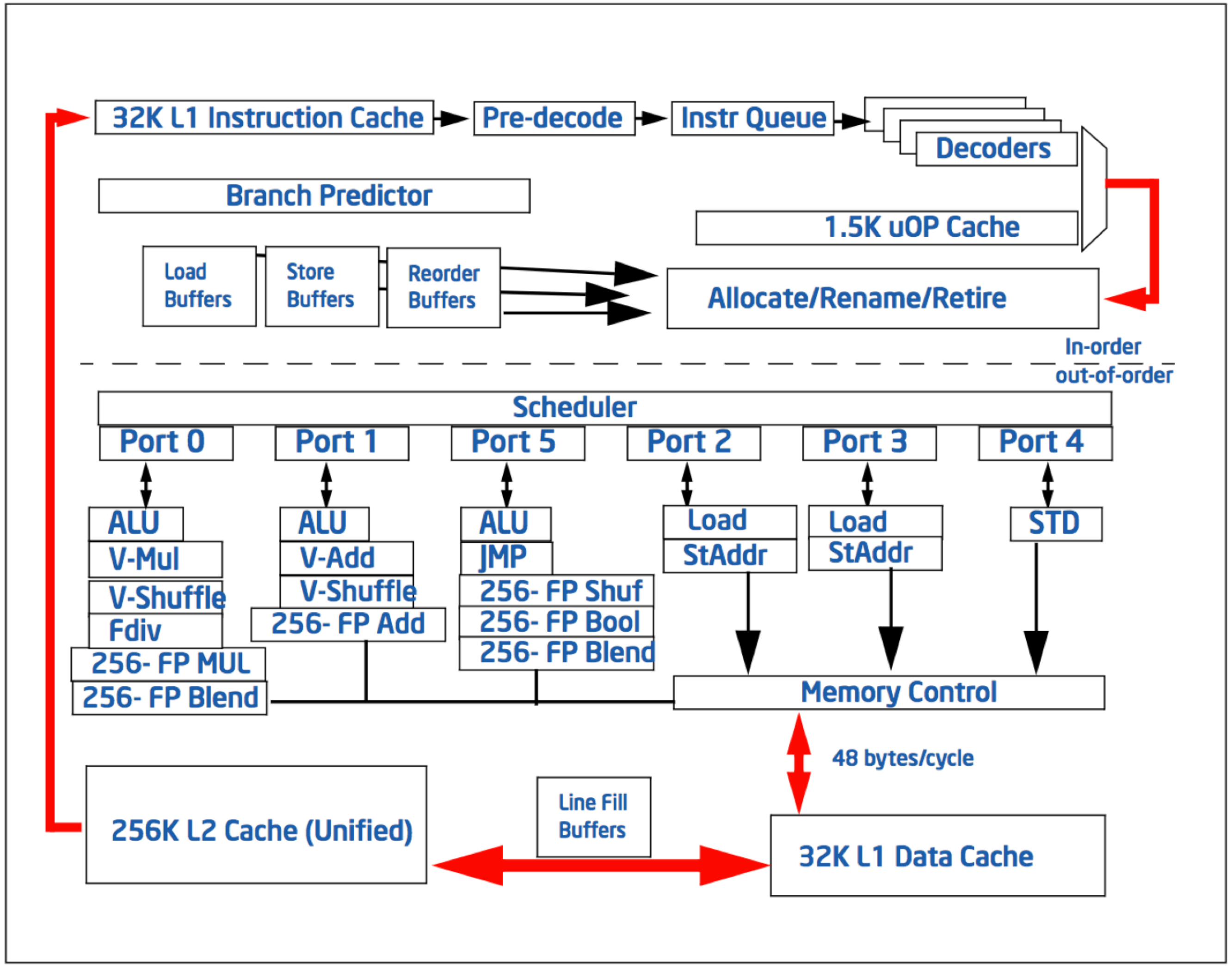
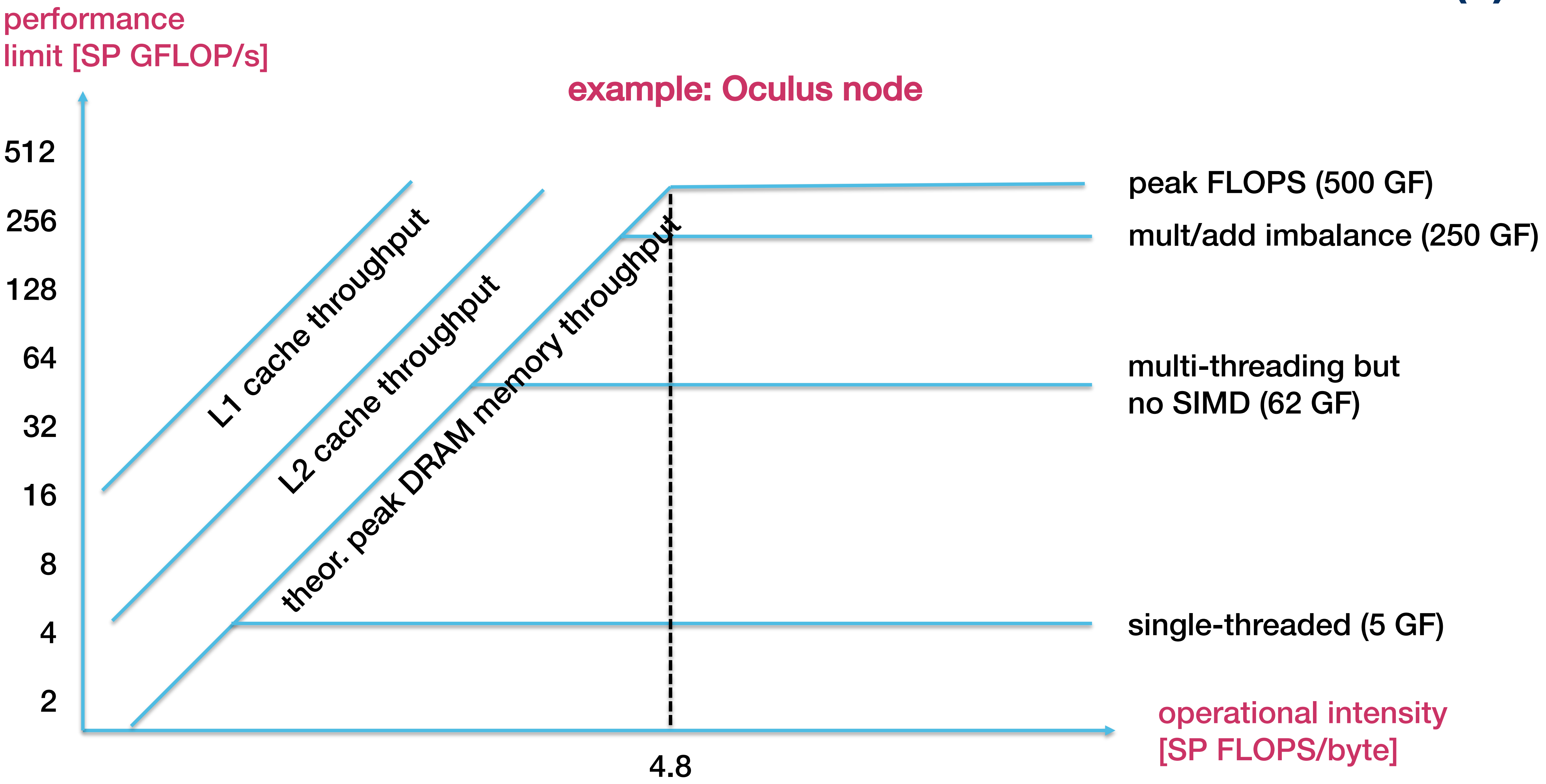
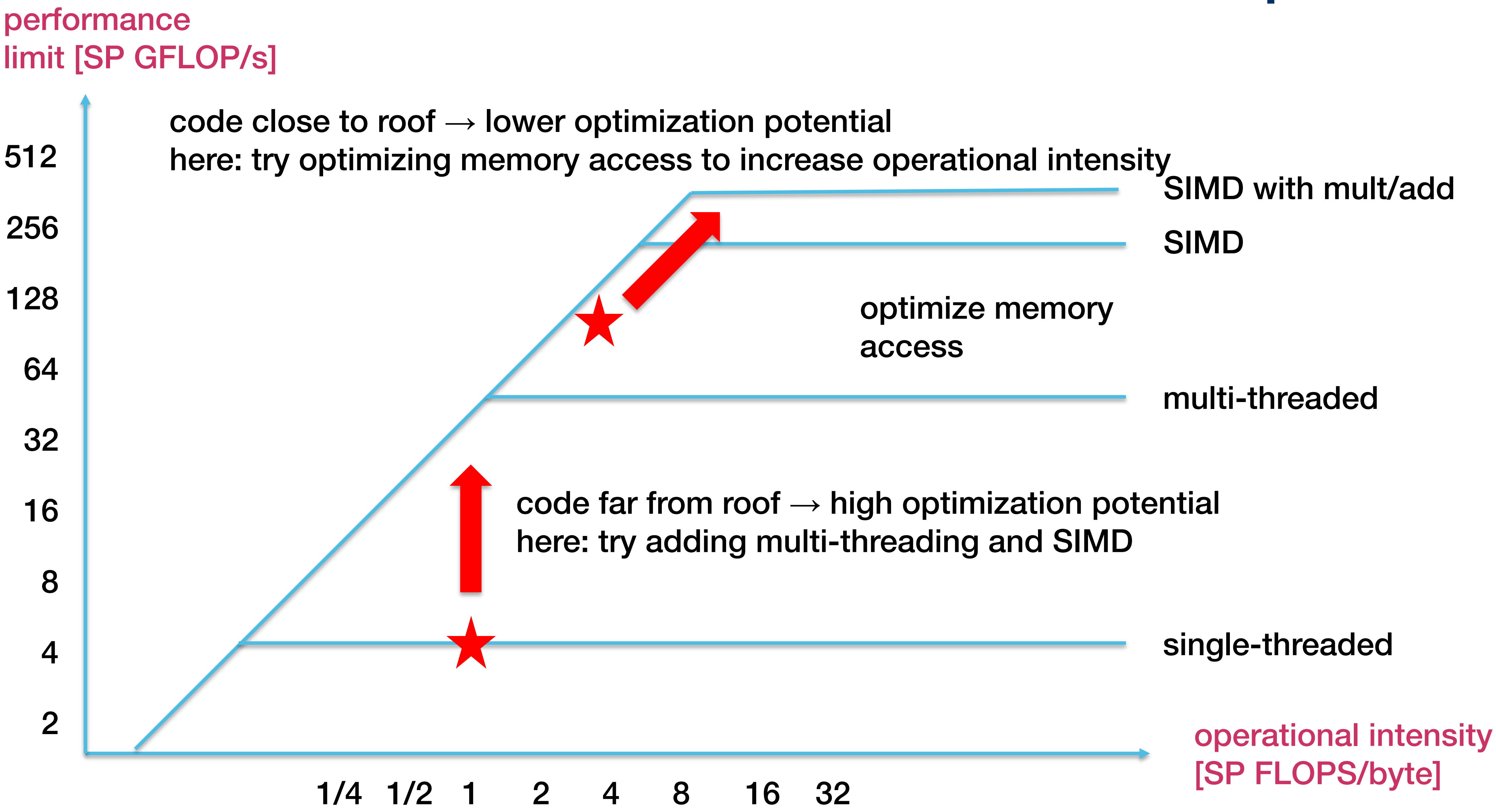


Figure 2-5. Intel Microarchitecture Code Name Sandy Bridge Pipeline Functionality

Roofline Model (3)



How to Use Roofline Model for Optimization?



code close to roof → lower optimization potential
here: try optimizing memory access to increase operational intensity

optimize memory access

code far from roof → high optimization potential
here: try adding multi-threading and SIMD

Example 1: STREAM Triad

- Current HPC machines require about 5-10 FLOPS / byte to reach peak arithmetic capabilities
  - hence, we need about 40-80 operations per double-precision value read from DRAM to reach peak compute capability
  - very hard to achieve for real codes
  - the compute to memory balance is unlikely to improve in future (technological and economical tradeoffs in computer system design)

```
#pragma omp parallel for
for(int i=0; i<N, i++) {
  z[i] = x[i] + alpha*y[i];
}
```

- Example: STREAM Triad
  - part of STREAM memory system benchmark
  - 2 FLOPs per iteration
  - transfers 24 bytes per iteration (read x[i], read y[i], write z[i]
  - OI = 2 / 24 = 0.083 FLOP/byte → clearly memory bound

Example 2: 5-point Stencil

- 5-point constant coefficient stencil
  - applies same operation to all points in 2D array
  - 5 FLOP per evaluation
  - 6 memory transfers (5 read, 1 write) per point
- Naïve implementation
  - do not consider any caching, read all data from DRAM
  - AI = 5 / (6 * 8) = 0.104 FLOP/byte → memory bound
- Consider Caching
  - if the cache can store two rows of the array all data access except for 1 read and 1 write per point can be
  - AI = 5 / (2 * 8) = 0.3125 FLOP/byte → still memory bound

```
#pragma omp parallel for

for (int t=0; t<tmax; t++) {

  for(int y=1; y<ydim+1; y++) {
    for(int x=1; x<xdim+1; x++) {

    int idx = x + y*ystride;
    new[idx] = - 3.0*old[idx]
               + old[idx-1]
               + old[idx+1]
               + old[idx-ystride]
               + old[idx+ystride];
    }
  }

}
```

Operational Intensities

- Determining operational intensities for non-trivial applications is difficult
  - performance analysis tools can determine OI from profiling/performance counter data, e.g. Intel vTune Amplifier

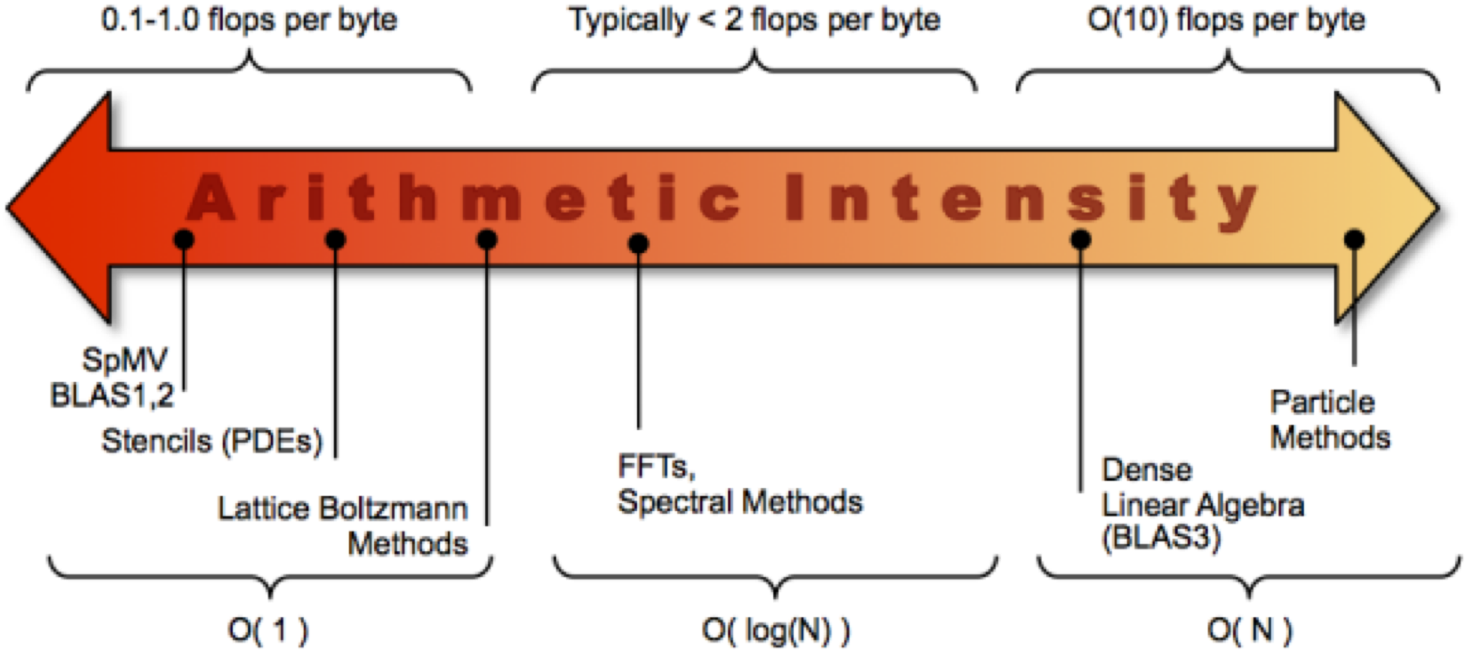


illustration: http://crd.lbl.gov/departments/computer-science/PAR/research/roofline

Case study:
Performance optimization of n-body solver

Performance Optimization: n-body

- Optimize the n-body application used earlier in this course

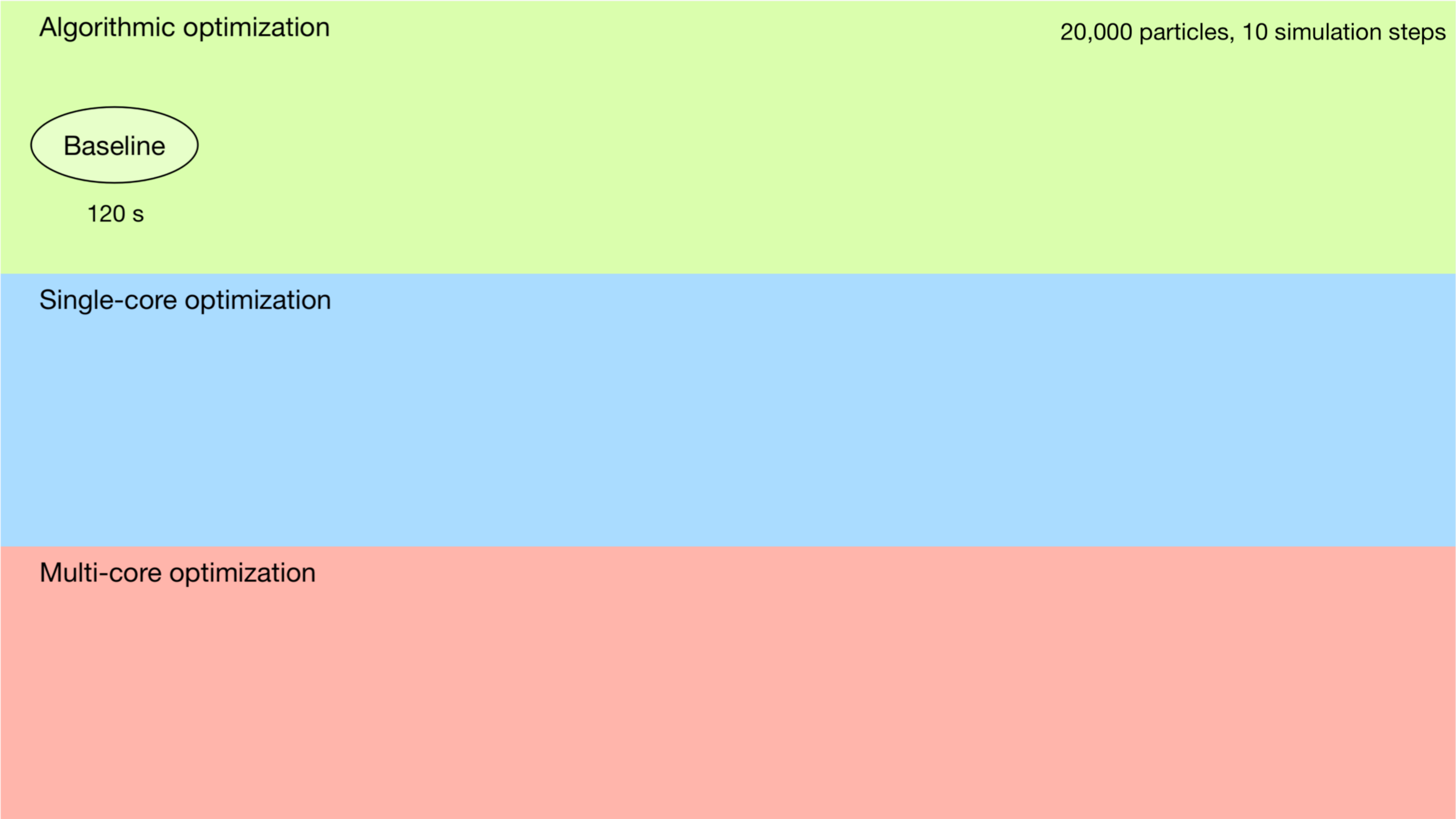
- Main application loop:

```
for (step = 1; step <= n_steps; step++) {
    t = step*delta_t;
    for (part = 0; part < n; part++)
      Compute_force(part, forces, curr, n);
    for (part = 0; part < n; part++)
      Update_part(part, forces, curr, n, delta_t);
}
```

- Compute_force(…):

```
   for (k = 0; k < n; k++) {
      if (k != part) {
         f_part_k[X] = curr[part].s[X] - curr[k].s[X];
         f_part_k[Y] = curr[part].s[Y] - curr[k].s[Y];
         len = sqrt(f_part_k[X]*f_part_k[X] + f_part_k[Y]*f_part_k[Y]);
         len_3 = len*len*len;
         mg = -G*curr[part].m*curr[k].m;
         fact = mg/len_3;
         f_part_k[X] *= fact;
         f_part_k[Y] *= fact;
         forces[part][X] += f_part_k[X];
         forces[part][Y] += f_part_k[Y];
      }
   }
```

Performance Optimization: n-body



Algorithmic optimization

20,000 particles, 10 simulation steps

Baseline

120 s

Single-core optimization

Multi-core optimization

Performance Optimization: n-body

- Intel Advisor: Tool for prototyping and optimization of
  - Vectorization
  - Memory access patterns
  - Multi threading

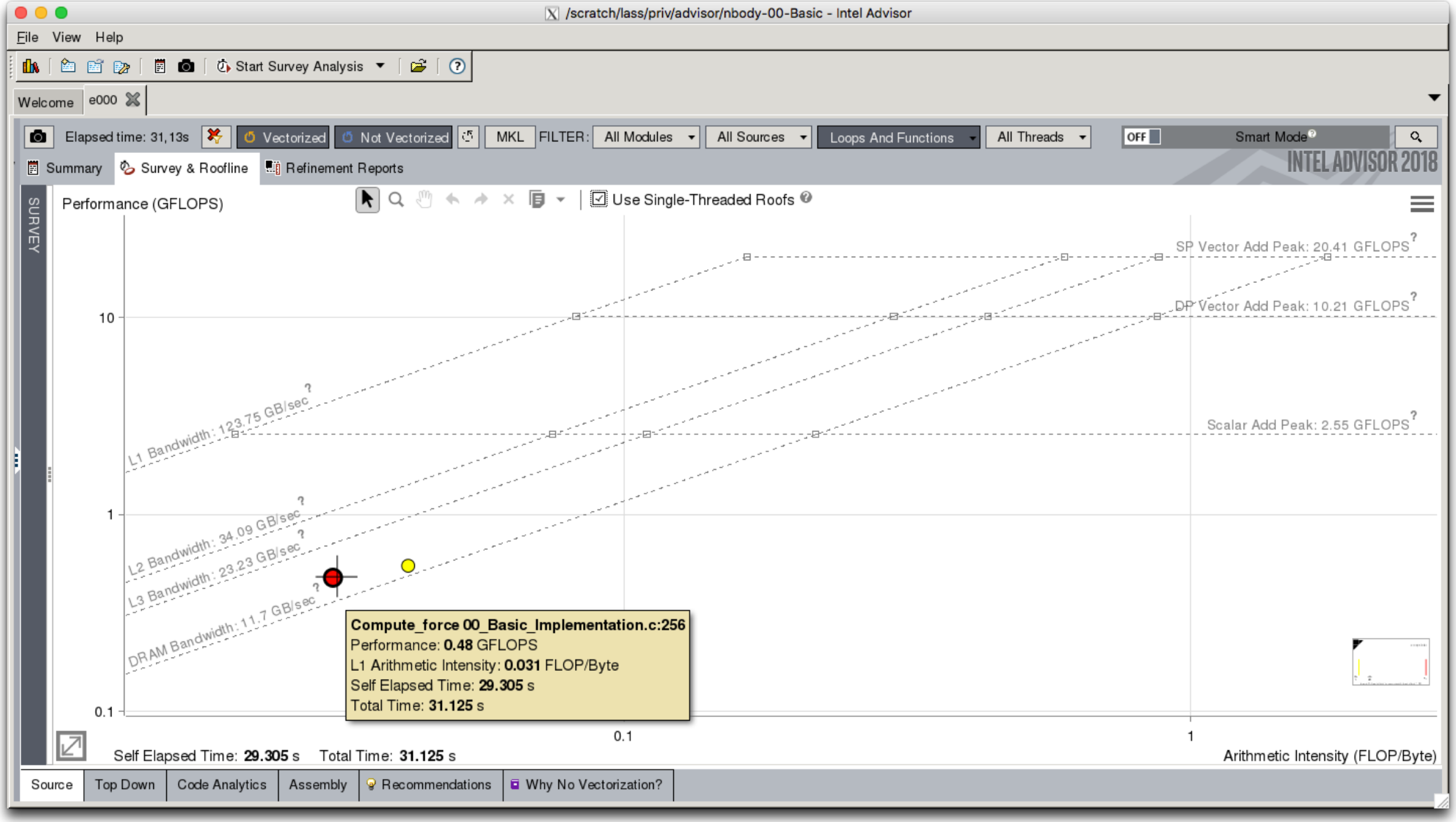
- Since 2017, Advisor plots a roofline model

- Starting Advisor on our systems:

```
module load ps_xe_2018
advixe-gui
```

Demo: Intel Advisor & n-body code

Performance Optimization: n-body



Compute_force 00_Basic_Implementation.c:256
Performance: **0.48** GFLOPS
L1 Arithmetic Intensity: **0.031** FLOP/Byte
Self Elapsed Time: **29.305** s
Total Time: **31.125** s

Self Elapsed Time: **29.305** s   Total Time: **31.125** s

Baseline: non-optimized code

Performance Optimization: n-body

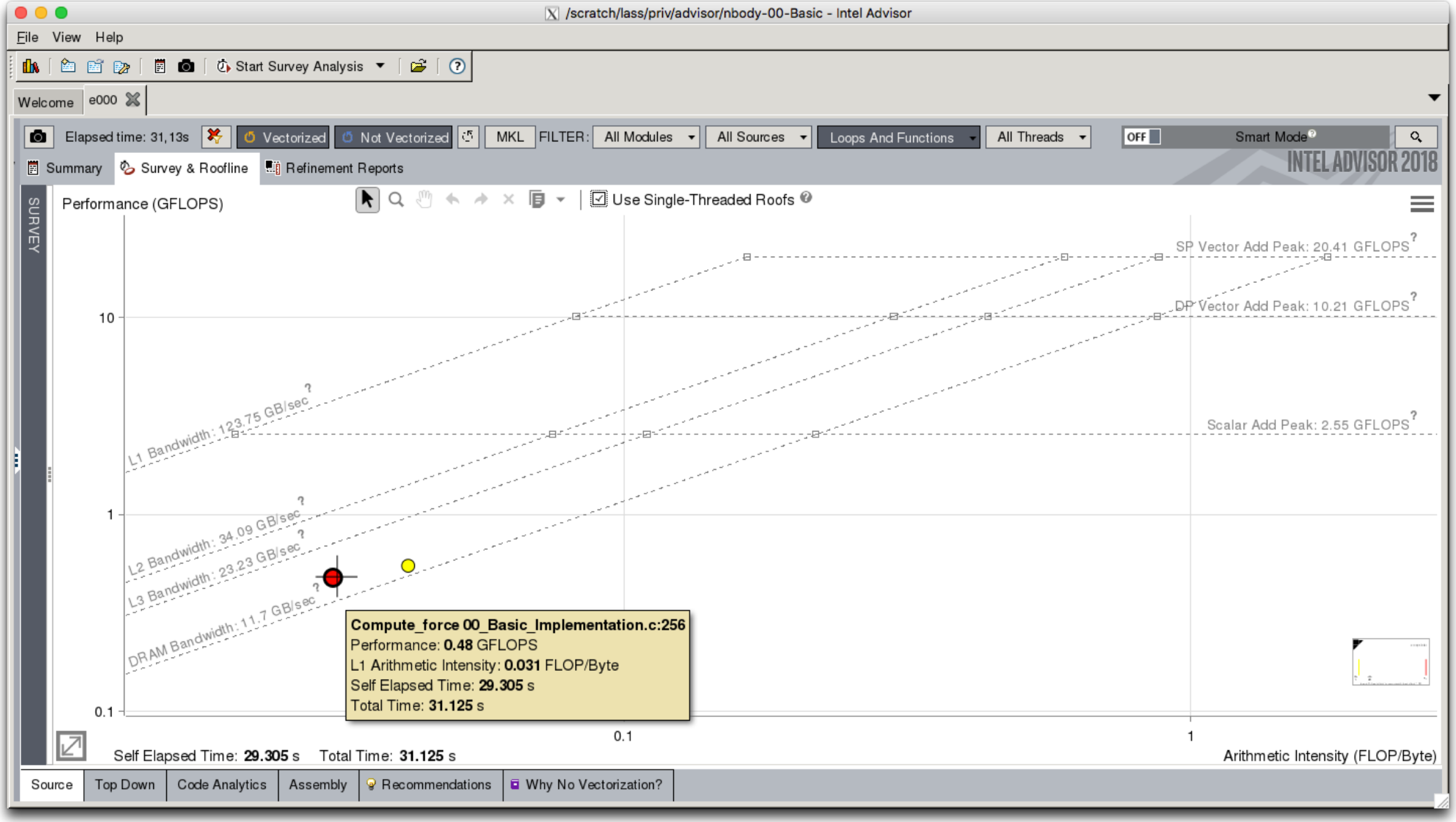
```
for (k = 0; k < n; k++) {
   if (k != part) {
      f_part_k[X] = curr[part].s[X] - curr[k].s[X];
      f_part_k[Y] = curr[part].s[Y] - curr[k].s[Y];
      len = sqrt(f_part_k[X]*f_part_k[X] + f_part_k[Y]*f_part_k[Y]);
      len_3 = len*len*len;
      mg = -G*curr[part].m*curr[k].m;
      fact = mg/len_3;
      f_part_k[X] *= fact;
      f_part_k[Y] *= fact;
      forces[part][X] += f_part_k[X];
      forces[part][Y] += f_part_k[Y];
   }
}
```

Compute forces only once for each pair of particles

```
for (k = part+1; k < n; k++) {
   f_part_k[X] = curr[part].s[X] - curr[k].s[X];
   f_part_k[Y] = curr[part].s[Y] - curr[k].s[Y];
   len = sqrt(f_part_k[X]*f_part_k[X] + f_part_k[Y]*f_part_k[Y]);
   len_3 = len*len*len;
   mg = -G*curr[part].m*curr[k].m;
   fact = mg/len_3;
   f_part_k[X] *= fact;
   f_part_k[Y] *= fact;
   forces[part][X] += f_part_k[X];
   forces[part][Y] += f_part_k[Y];
   forces[k][X] -= f_part_k[X];
   forces[k][Y] -= f_part_k[Y];
}
```

In which direction do we move in the roofline model?

Performance Optimization: n-body

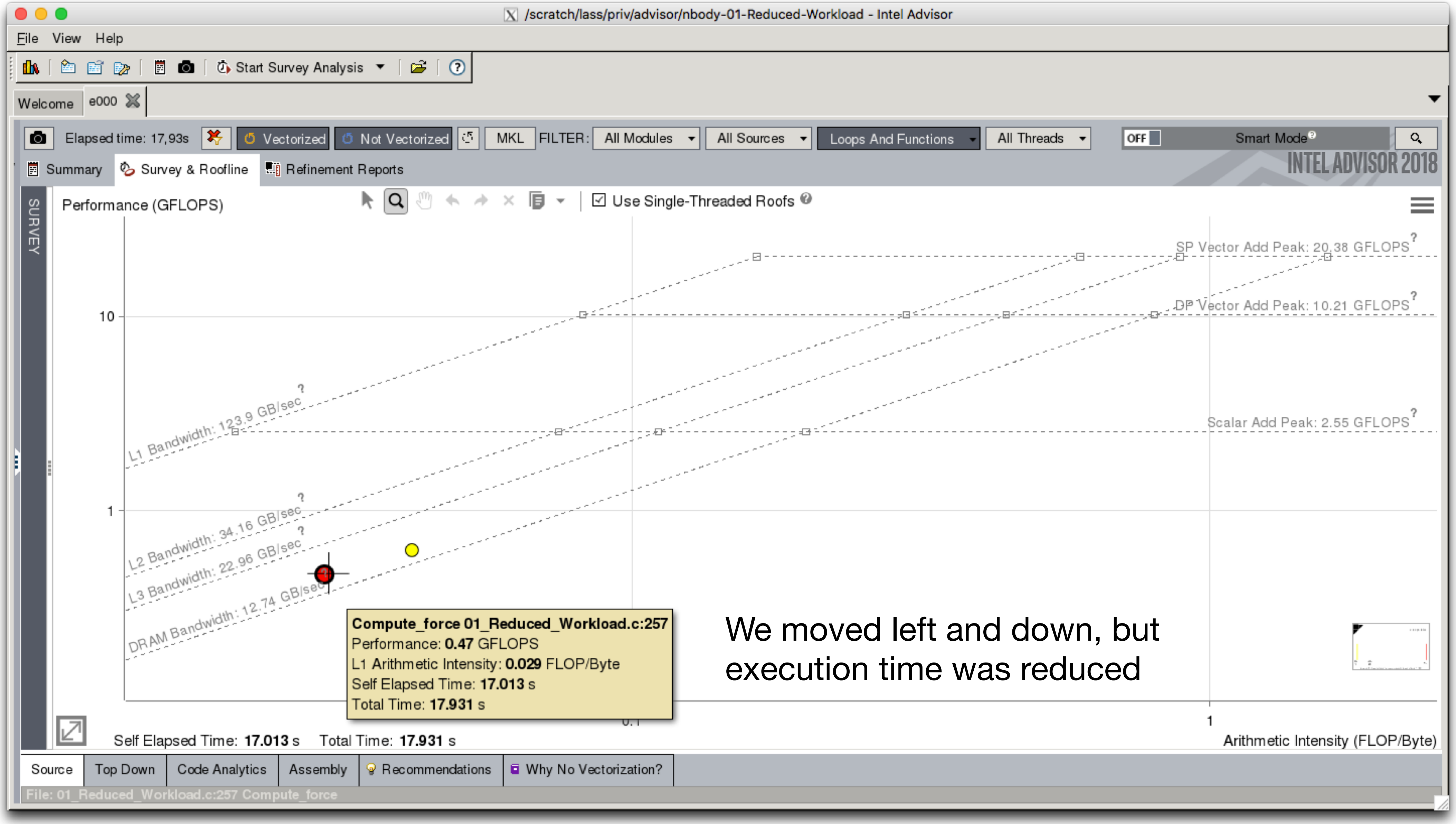


Compute_force 00_Basic_Implementation.c:256
Performance: **0.48** GFLOPS
L1 Arithmetic Intensity: **0.031** FLOP/Byte
Self Elapsed Time: **29.305** s
Total Time: **31.125** s

Self Elapsed Time: **29.305** s   Total Time: **31.125** s

Baseline: non-optimized code

Performance Optimization: n-body



/scratch/lass/priv/advisor/nbody-01-Reduced-Workload - Intel Advisor
File
View
Help
Start Survey Analysis
Welcome
e000
Vectorized
Not Vectorized
MKL
FILTER:
All Modules
All Sources
Loops And Functions
All Threads
OFF
Smart Mode
INTEL ADVISOR 2018
Summary
Survey & Roofline
Refinement Reports
SURVEY
Performance (GFLOPS)
☑ Use Single-Threaded Roofs
SP Vector Add Peak: 20.38 GFLOPS
DP Vector Add Peak: 10.21 GFLOPS
Scalar Add Peak: 2.55 GFLOPS
L1 Bandwidth: 123.9 GB/sec
L2 Bandwidth: 34.16 GB/sec
L3 Bandwidth: 22.96 GB/sec
DRAM Bandwidth: 12.74 GB/sec
10
1
0.1
1
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?

Elapsed time: 17,93s

Compute_force 01_Reduced_Workload.c:257
Performance: **0.47** GFLOPS
L1 Arithmetic Intensity: **0.029** FLOP/Byte
Self Elapsed Time: **17.013** s
Total Time: **17.931** s

Self Elapsed Time: **17.013** s Total Time: **17.931** s

File: 01_Reduced_Workload.c:257 Compute_force

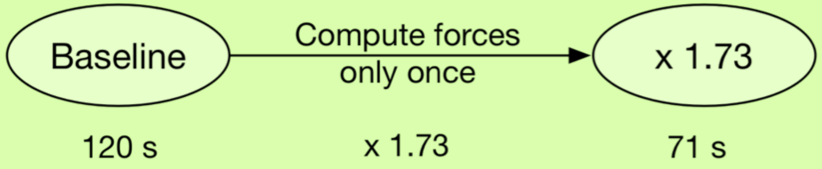
We moved left and down, but execution time was reduced

Algorithmicly optimized code

Performance Optimization: n-body

Algorithmic optimization

20,000 particles, 10 simulation steps



Baseline

Compute forces only once

x 1.73

120 s

x 1.73

71 s

Single-core optimization

Multi-core optimization

Performance Optimization: n-body

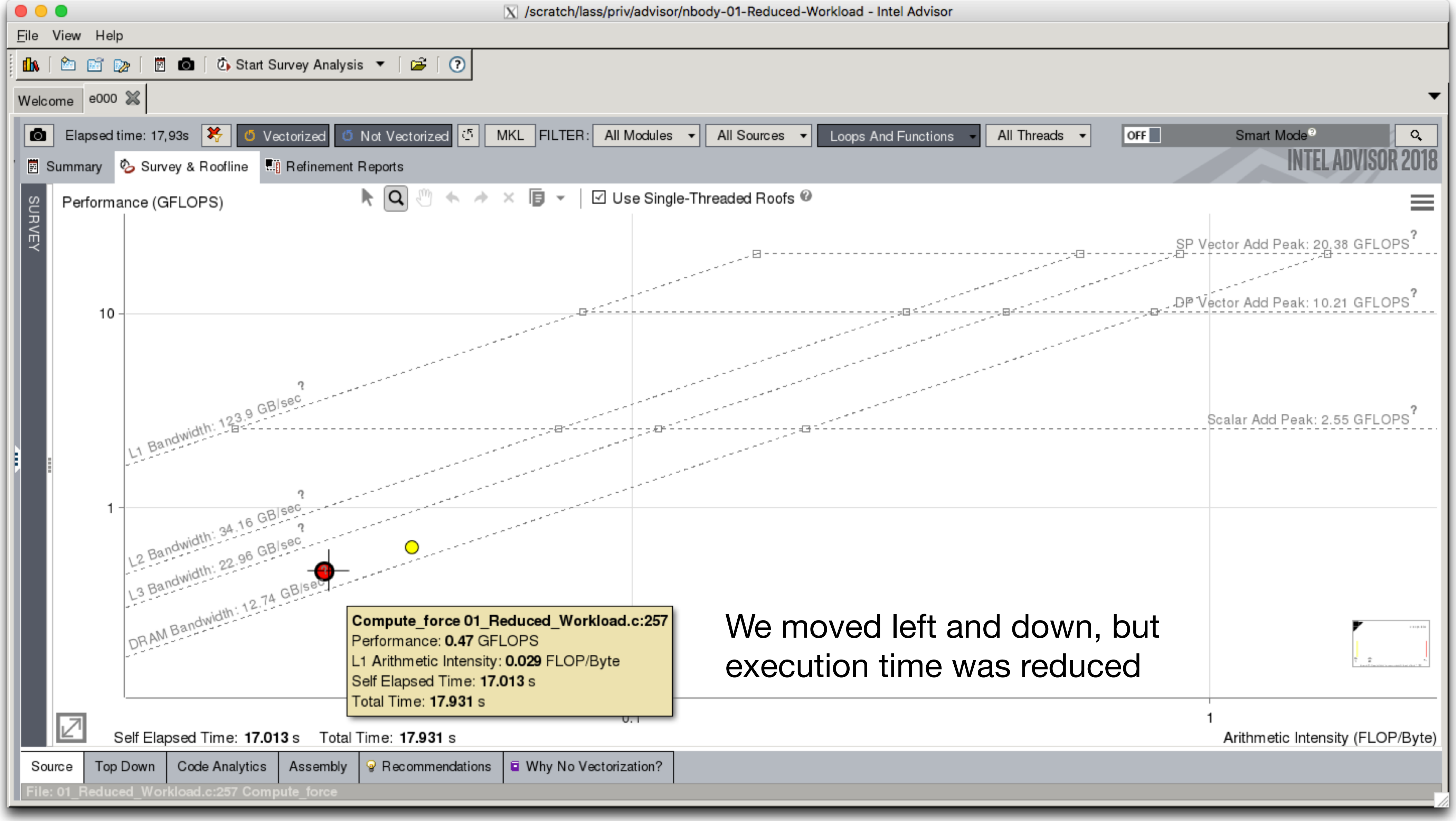
- **Enable compiler optimizations and auto vectorization:**

```
icc -O3 -xHOST -qopt-report -qopt-report-phase=vec -o nbody nbody.c
```

- **Vectorization report (excerpt):**

```
Begin optimization report for: Compute_force(int, vect_t *, struct particle_s *, int)

   Report from: Vector optimizations [vec]

LOOP BEGIN at nbody.c(257,4)
  remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at nbody.c(257,4)
<Remainder loop for vectorization>
LOOP END
```

Performance Optimization: n-body



Elapsed time: 17,93s

☑ Use Single-Threaded Roofs

Compute_force 01_Reduced_Workload.c:257
Performance: **0.47** GFLOPS
L1 Arithmetic Intensity: **0.029** FLOP/Byte
Self Elapsed Time: **17.013** s
Total Time: **17.931** s

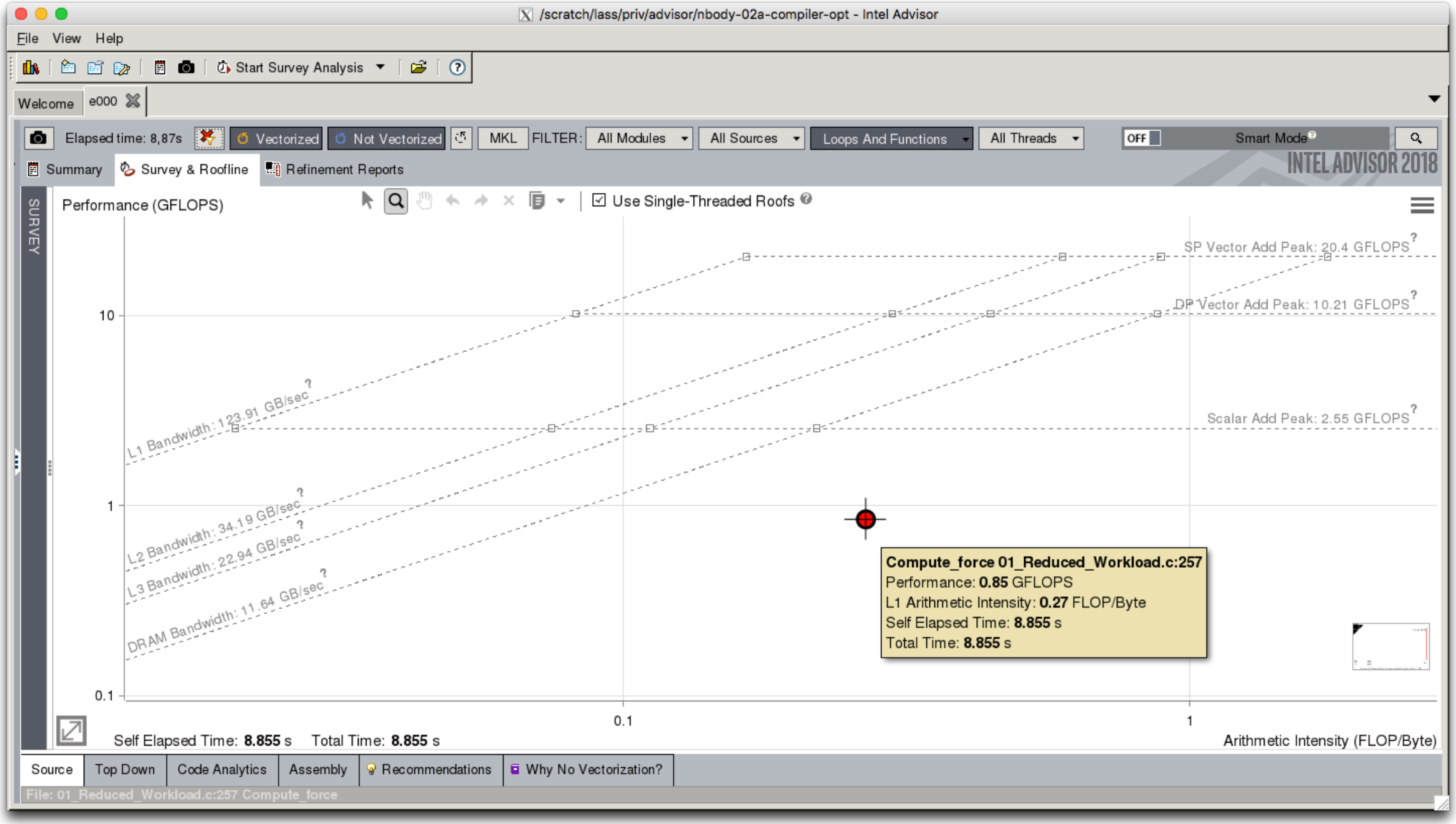
Self Elapsed Time: **17.013** s Total Time: **17.931** s

File: 01_Reduced_Workload.c:257 Compute_force

We moved left and down, but execution time was reduced

Algorithmicly optimized code

Performance Optimization: n-body



/scratch/lass/priv/advisor/nbody-02a-compiler-opt - Intel Advisor
File View Help
Start Survey Analysis
Welcome
e000
Vectorized
Not Vectorized
MKL
All Modules
All Sources
Loops And Functions
All Threads
OFF
Smart Mode
INTEL ADVISOR 2018
Summary
Survey & Roofline
Refinement Reports
SURVEY
Performance (GFLOPS)
☑ Use Single-Threaded Roofs
SP Vector Add Peak: 20.4 GFLOPS
DP Vector Add Peak: 10.21 GFLOPS
Scalar Add Peak: 2.55 GFLOPS
L1 Bandwidth: 123.91 GB/sec
L2 Bandwidth: 34.19 GB/sec
L3 Bandwidth: 22.94 GB/sec
DRAM Bandwidth: 11.64 GB/sec
10
1
0.1
0.1
1
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?

Elapsed time: 8,87s

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

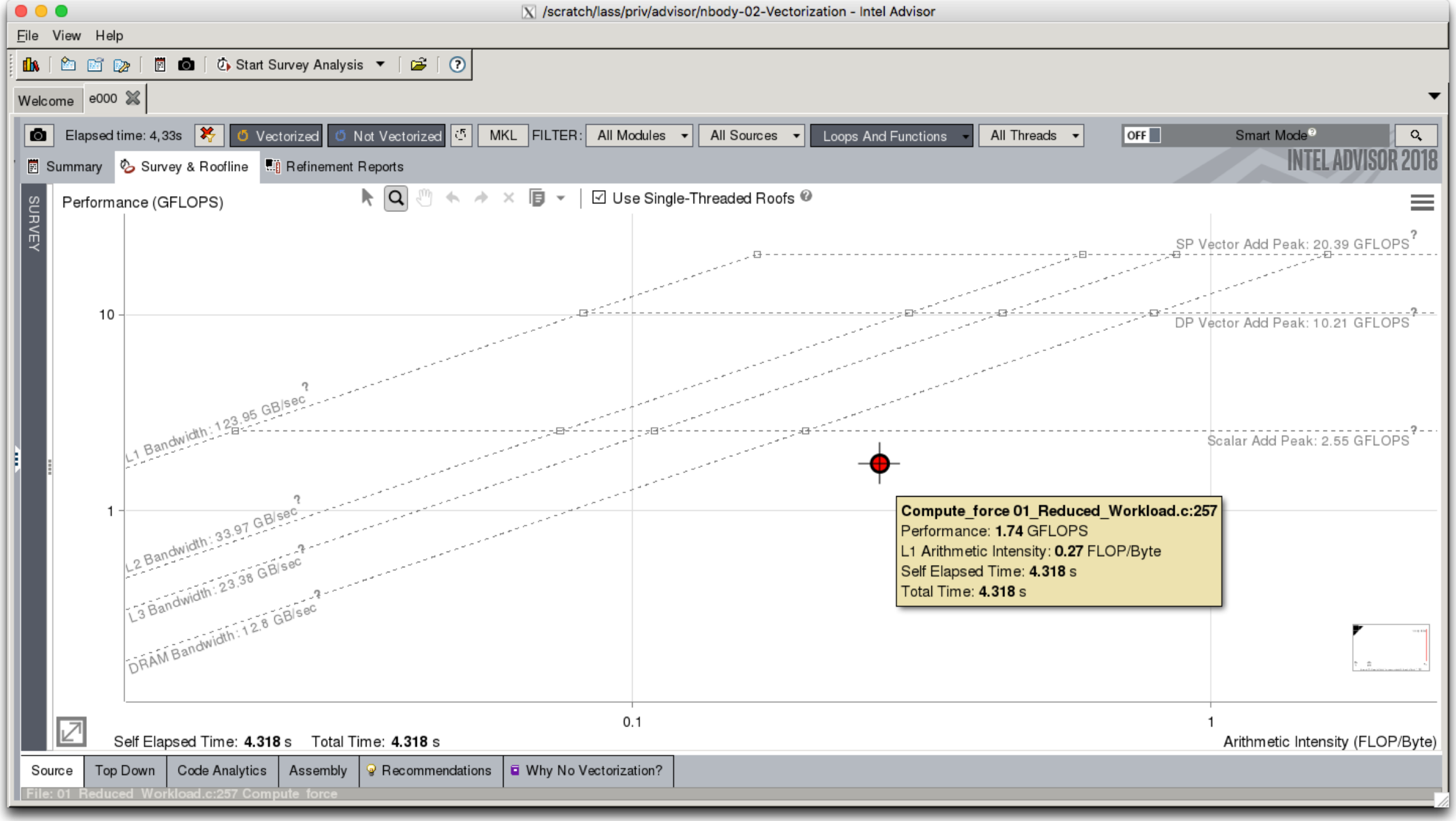
Compute_force 01_Reduced_Workload.c:257
Performance: **0.85** GFLOPS
L1 Arithmetic Intensity: **0.27** FLOP/Byte
Self Elapsed Time: **8.855** s
Total Time: **8.855** s

Self Elapsed Time: **8.855** s Total Time: **8.855** s

File: 01_Reduced_Workload.c:257 Compute_force

Compiler optimizations only

Performance Optimization: n-body

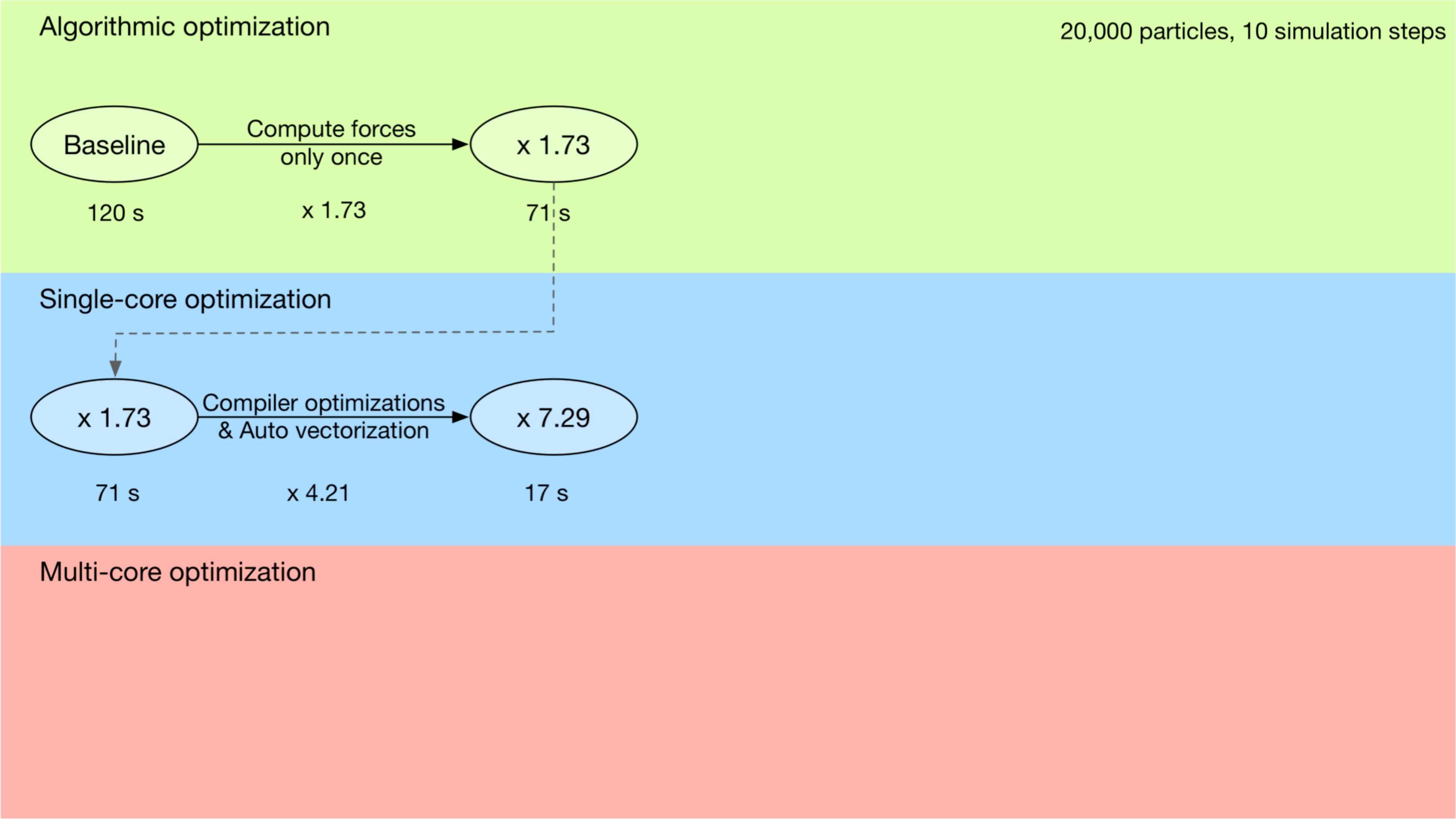

/scratch/lass/priv/advisor/nbody-02-Vectorization - Intel Advisor
File View Help
Start Survey Analysis
Welcome
e000
Elapsed time: 4,33s
Vectorized
Not Vectorized
MKL
FILTER: All Modules
All Sources
Loops And Functions
All Threads
OFF
Smart Mode
INTEL ADVISOR 2018
Summary
Survey & Roofline
Refinement Reports
SURVEY
Performance (GFLOPS)
☑ Use Single-Threaded Roofs
SP Vector Add Peak: 20.39 GFLOPS
DP Vector Add Peak: 10.21 GFLOPS
Scalar Add Peak: 2.55 GFLOPS
L1 Bandwidth: 123.95 GB/sec
L2 Bandwidth: 33.97 GB/sec
L3 Bandwidth: 23.38 GB/sec
DRAM Bandwidth: 12.8 GB/sec
10
1
0.1
1
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
File: 01_Reduced_Workload.c:257 Compute_force

Compute_force 01_Reduced_Workload.c:257
Performance: **1.74** GFLOPS
L1 Arithmetic Intensity: **0.27** FLOP/Byte
Self Elapsed Time: **4.318** s
Total Time: **4.318** s

Self Elapsed Time: **4.318** s Total Time: **4.318** s

Compiler optimizations and auto vectorization

Performance Optimization: n-body



Algorithmic optimization

20,000 particles, 10 simulation steps

Baseline → Compute forces only once → x 1.73

120 s

x 1.73

71 s

Single-core optimization

x 1.73 → Compiler optimizations & Auto vectorization → x 7.29

71 s

x 4.21

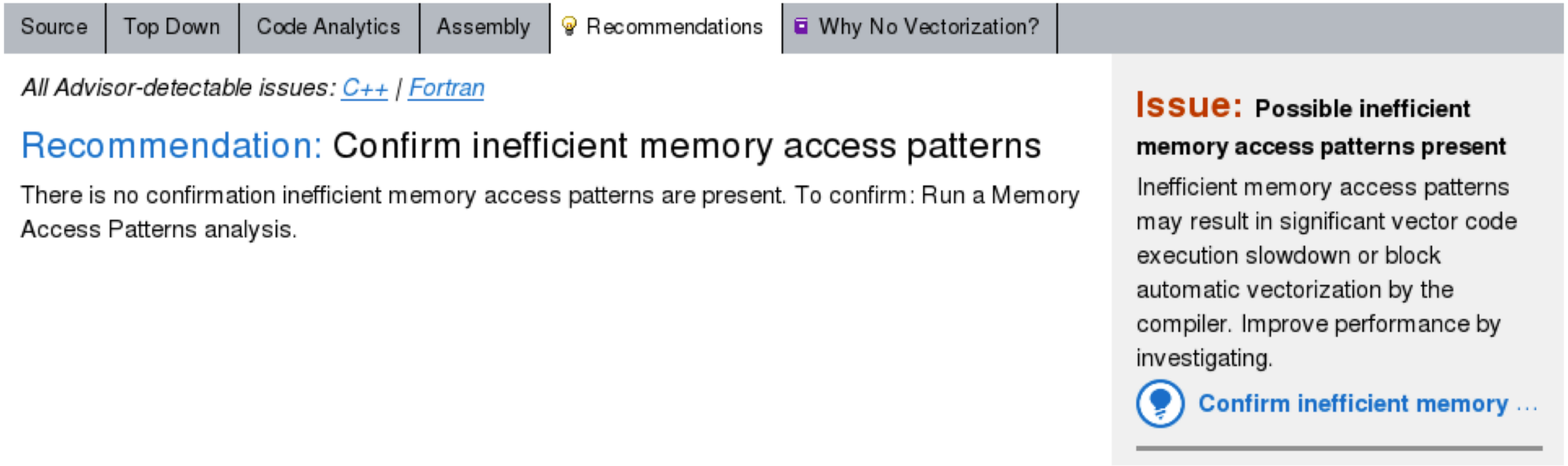
17 s

Multi-core optimization

Demo: Intel Advisor Recommendations

Performance Optimization: n-body

- Advisor gives hints on how to improve performance:



All Advisor-detectable issues: *C++* | *Fortran*

Recommendation: Confirm inefficient memory access patterns

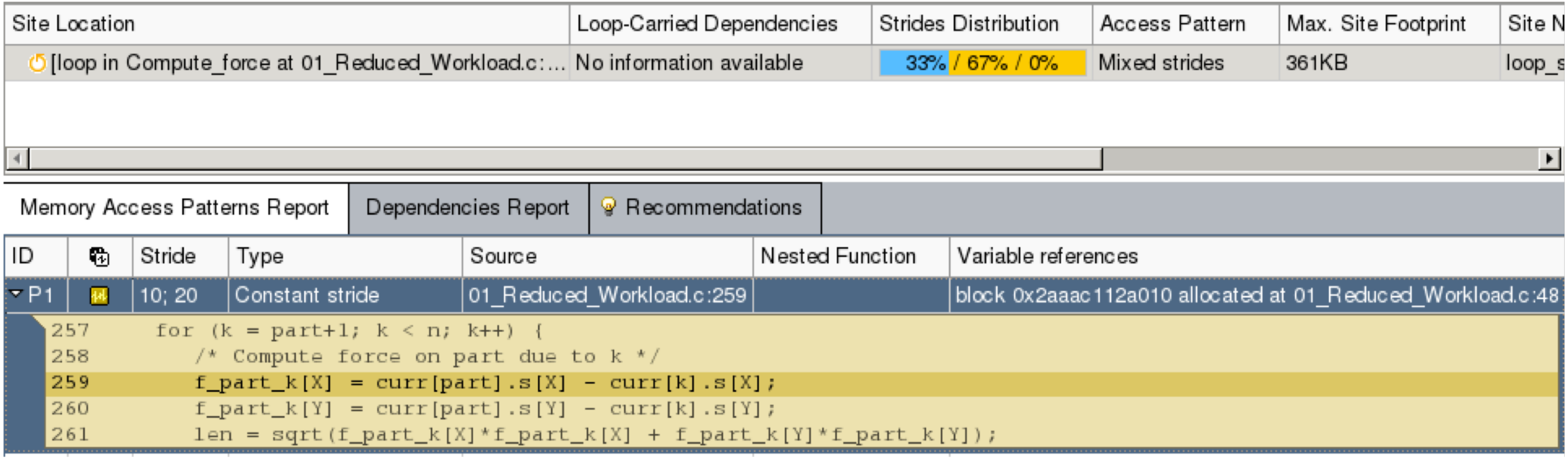
There is no confirmation inefficient memory access patterns are present. To confirm: Run a Memory Access Patterns analysis.

Issue: **Possible inefficient memory access patterns present**

Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.

Confirm inefficient memory ...

Performance Optimization: n-body



Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site N
[loop in Compute_force at 01_Reduced_Workload.c:...	No information available	33% / 67% / 0%	Mixed strides	361KB	loop_s

ID		Stride	Type	Source	Nested Function	Variable references
P1		10; 20	Constant stride	01_Reduced_Workload.c:259		block 0x2aaac112a010 allocated at 01_Reduced_Workload.c:48

```
257     for (k = part+1; k < n; k++) {
258        /* Compute force on part due to k */
259        f_part_k[X] = curr[part].s[X] - curr[k].s[X];
260        f_part_k[Y] = curr[part].s[Y] - curr[k].s[Y];
261        len = sqrt(f_part_k[X]*f_part_k[X] + f_part_k[Y]*f_part_k[Y]);
```

Performance Optimization: n-body

- Unit strides: Elements in memory accessed in consecutive order
- Constant strides: Fixed distance between accessed elements
- Irregular strides: Random access

- Why can non-unit strides be a problem?
  - Not all data transferred from memory is used
  - Elements need to be gathered into vectors
  - Memory prefetching may not provide required data

```
#define SIZE 1000
int i, j;
double *a = malloc(SIZE * SIZE * sizeof(double))
for (j=0; j < SIZE; j++) {
        for (i=0; i < SIZE; i++) {
                do_something(a[i*SIZE + j]);
        }
}
```

Performance Optimization: n-body

- In our application we use an array of structs

```
typedef double vect_t[DIM];
struct particle_s {
   double m;  /* Mass      */
   vect_t s;    /* Position */
   vect_t v;    /* Velocity */
};
[...]
curr = malloc(n*sizeof(struct particle_s));
```

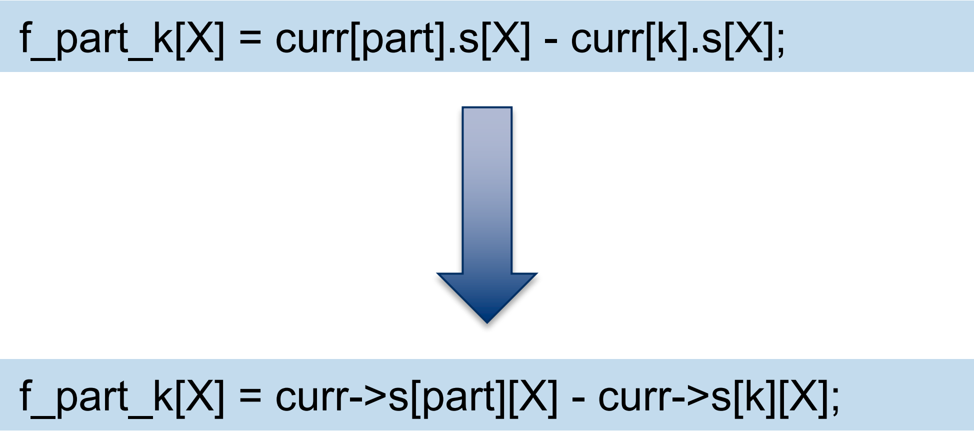
- Distance between X coordinates of two consecutive particles in memory: 40 bytes

- Solution: Use struct of arrays instead

```
typedef double vect_t[DIM];
struct particles {
   double *m;  /* Masses     */
   vect_t *s;    /* Positions  */
   vect_t *v;    /* Velocities */
};
[...]
curr = malloc(sizeof(struct particles));
curr->m = malloc(n * sizeof(double));
curr->s  = malloc(n * sizeof(vect_t));
curr->v  = malloc(n * sizeof(vect_t));
```

Performance Optimization: n-body

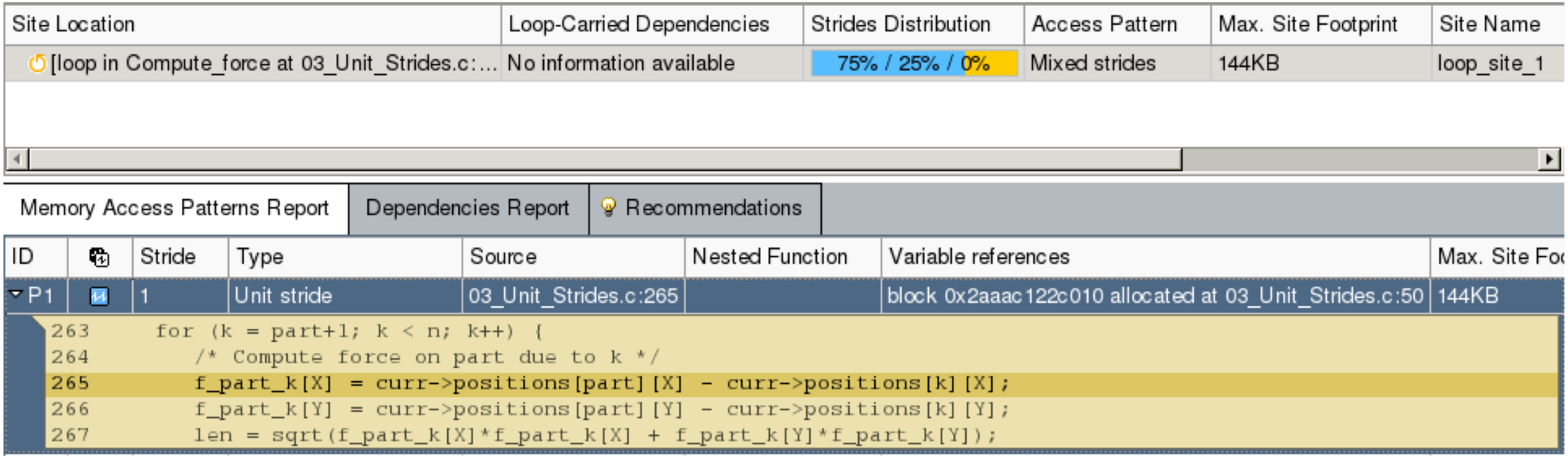
- Changing underlying data structure requires to change each access



```
f_part_k[X] = curr[part].s[X] - curr[k].s[X];
```

```
f_part_k[X] = curr->s[part][X] - curr->s[k][X];
```

Performance Optimization: n-body

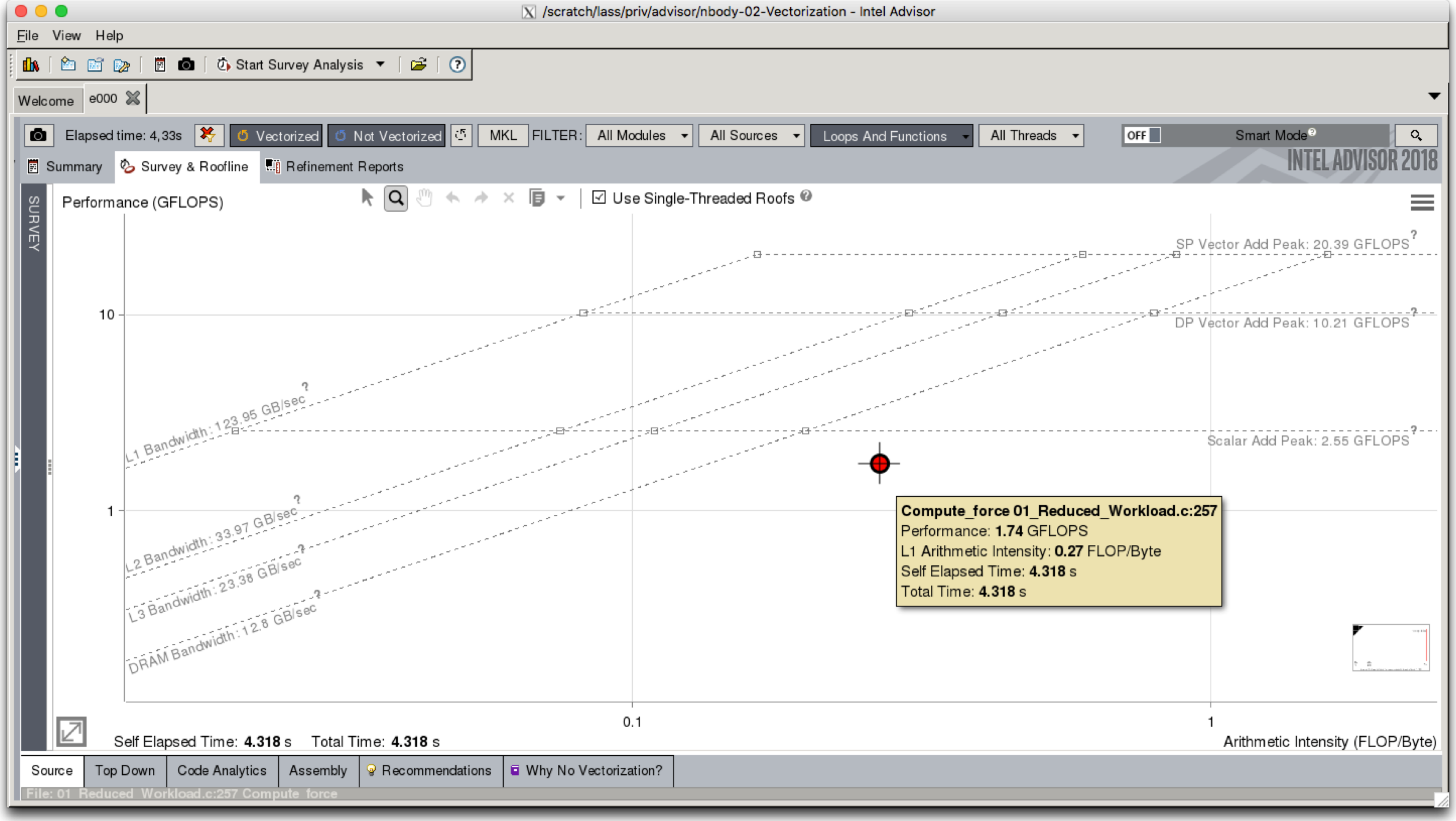


Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Max. Site Footprint	Site Name
[loop in Compute_force at 03_Unit_Strides.c:…	No information available	75% / 25% / 0%	Mixed strides	144KB	loop_site_1

ID		Stride	Type	Source	Nested Function	Variable references	Max. Site Foo
P1		1	Unit stride	03_Unit_Strides.c:265		block 0x2aaac122c010 allocated at 03_Unit_Strides.c:50	144KB

```
263     for (k = part+1; k < n; k++) {
264        /* Compute force on part due to k */
265        f_part_k[X] = curr->positions[part][X] - curr->positions[k][X];
266        f_part_k[Y] = curr->positions[part][Y] - curr->positions[k][Y];
267        len = sqrt(f_part_k[X]*f_part_k[X] + f_part_k[Y]*f_part_k[Y]);
```

Performance Optimization: n-body



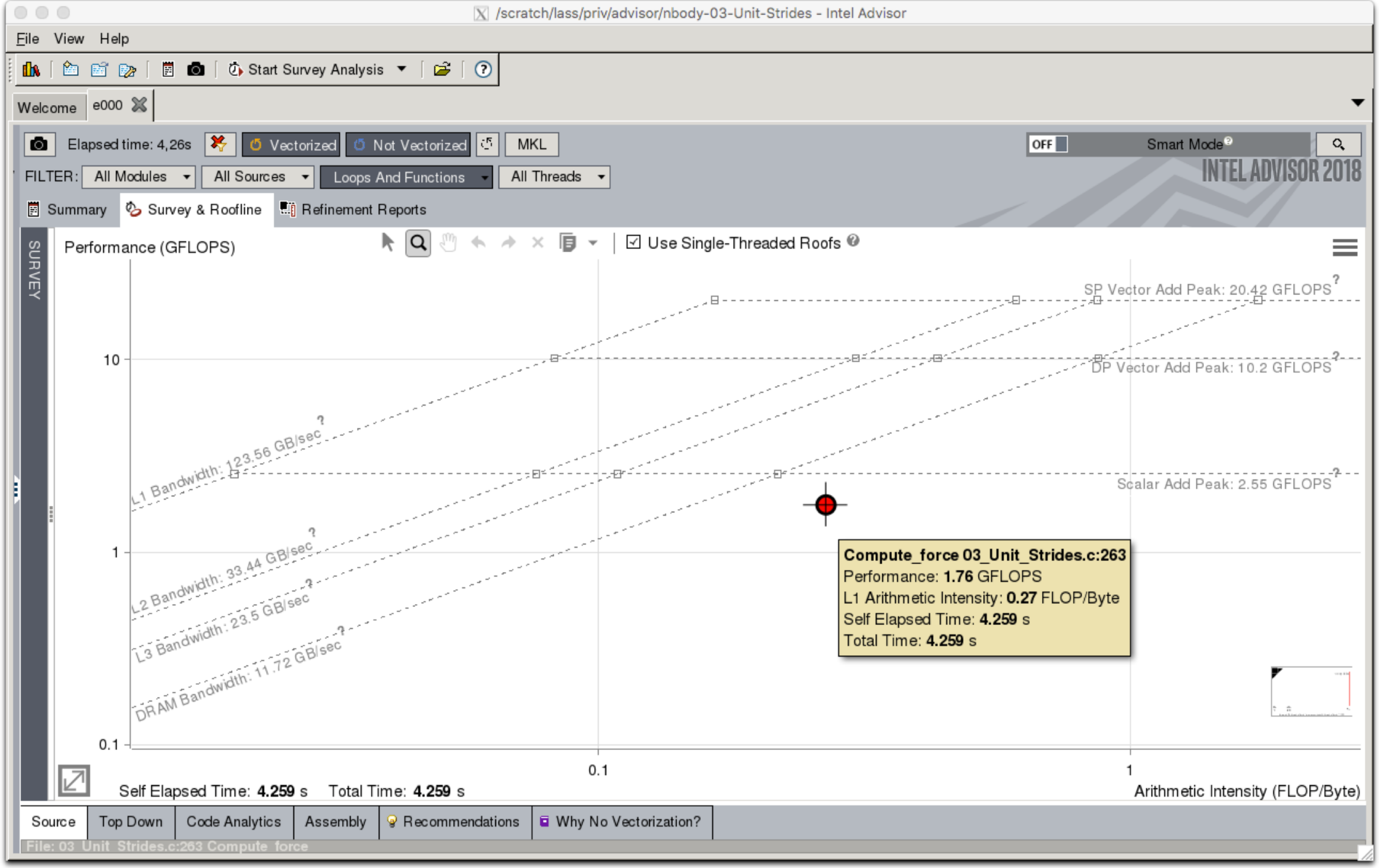
Compute_force 01_Reduced_Workload.c:257
Performance: **1.74** GFLOPS
L1 Arithmetic Intensity: **0.27** FLOP/Byte
Self Elapsed Time: **4.318** s
Total Time: **4.318** s

Self Elapsed Time: **4.318** s Total Time: **4.318** s

File: 01_Reduced_Workload.c:257 Compute_force

Compiler optimizations and auto vectorization

Performance Optimization: n-body



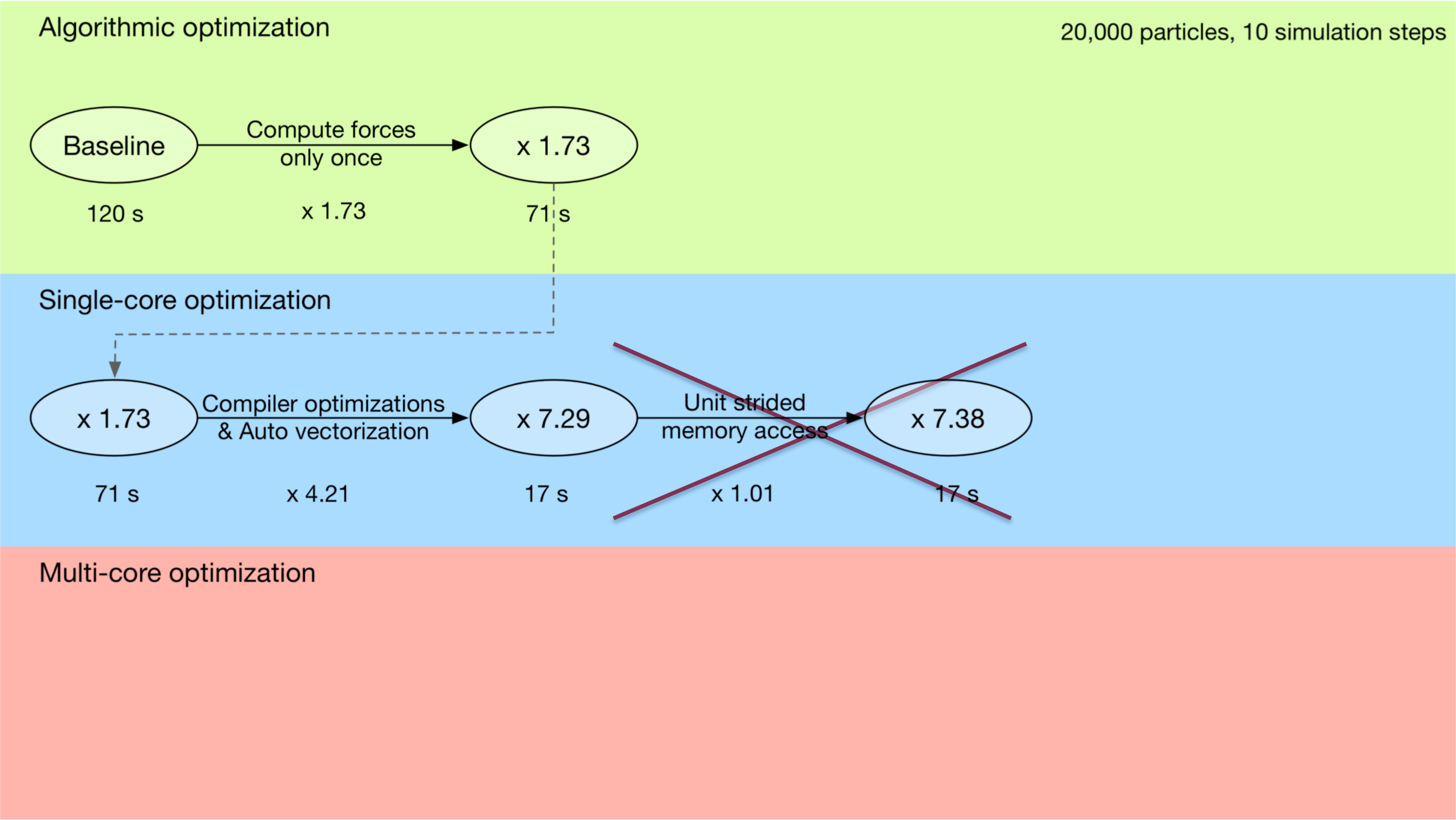
/scratch/lass/priv/advisor/nbody-03-Unit-Strides - Intel Advisor
File View Help
Start Survey Analysis
Welcome
e000
Elapsed time: 4,26s
Vectorized
Not Vectorized
MKL
OFF
Smart Mode
INTEL ADVISOR 2018
FILTER: All Modules
All Sources
Loops And Functions
All Threads
Summary
Survey & Roofline
Refinement Reports
SURVEY
Performance (GFLOPS)
☑ Use Single-Threaded Roofs
SP Vector Add Peak: 20.42 GFLOPS
DP Vector Add Peak: 10.2 GFLOPS
Scalar Add Peak: 2.55 GFLOPS
L1 Bandwidth: 123.56 GB/sec
L2 Bandwidth: 33.44 GB/sec
L3 Bandwidth: 23.5 GB/sec
DRAM Bandwidth: 11.72 GB/sec
10
1
0.1
0.1
1
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
File: 03_Unit_Strides.c:263 Compute_force

Compute_force 03_Unit_Strides.c:263
Performance: **1.76** GFLOPS
L1 Arithmetic Intensity: **0.27** FLOP/Byte
Self Elapsed Time: **4.259** s
Total Time: **4.259** s

Self Elapsed Time: **4.259** s Total Time: **4.259** s

Optimized memory access patterns

Performance Optimization: n-body



Performance Optimization: n-body

- Enable multi-threading using OpenMP

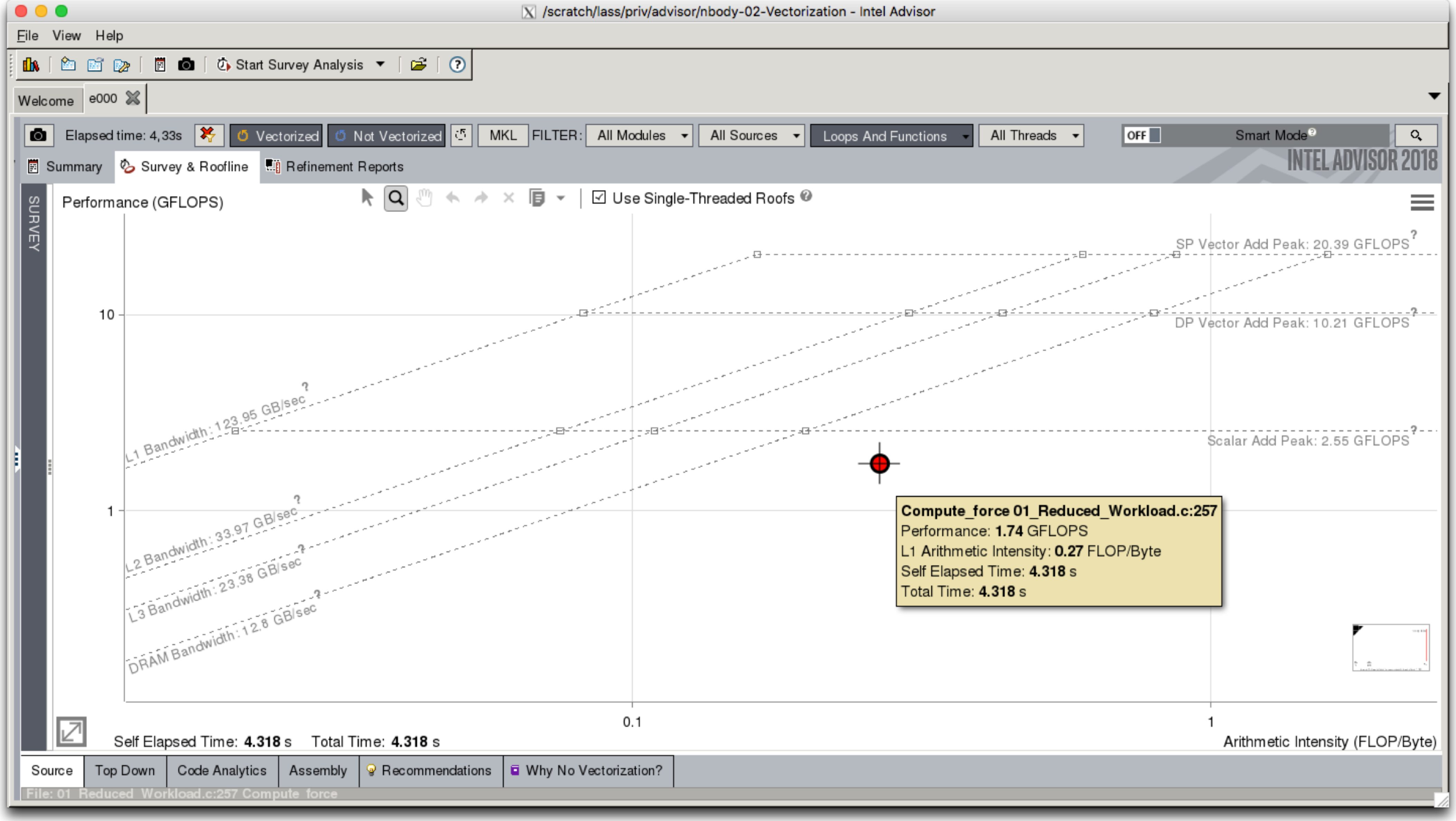
```
for (step = 1; step <= n_steps; step++) {
    t = step*delta_t;
    memset(forces, 0, n*sizeof(vect_t));
    for (part = 0; part < n-1; part++)
      Compute_force(part, forces, curr, n);
    for (part = 0; part < n; part++)
      Update_part(part, forces, curr, n, delta_t);
  }
```

```
for (k = part+1; k < n; k++) {
    [...]
    forces[part][X] += f_part_k[X];
    forces[part][Y] += f_part_k[Y];
    forces[k][X] -= f_part_k[X];
    forces[k][Y] -= f_part_k[Y];
  }
```

```
# pragma omp parallel num_threads(thread_count) default(none) \
shared(curr, forces, thread_count, delta_t, n, n_steps, output_freq) \
private(step, part, t)
  for (step = 1; step <= n_steps; step++) {
    t = step*delta_t;
#   pragma omp single
    memset(forces, 0, n*sizeof(vect_t));
#   pragma omp for
    for (part = 0; part < n-1; part++)
      Compute_force(part, forces, curr, n);
#   pragma omp for
    for (part = 0; part < n; part++)
      Update_part(part, forces, curr, n, delta_t);
  }
```

```
 for (k = part+1; k < n; k++) {
    [...]
#   pragma omp atomic
    forces[part][X] += f_part_k[X];
#   pragma omp atomic
    forces[part][Y] += f_part_k[Y];
#   pragma omp atomic
    forces[k][X] -= f_part_k[X];
#   pragma omp atomic
    forces[k][Y] -= f_part_k[Y];
  }
```

Performance Optimization: n-body



/scratch/lass/priv/advisor/nbody-02-Vectorization - Intel Advisor

Elapsed time: 4,33s

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

☑ Use Single-Threaded Roofs

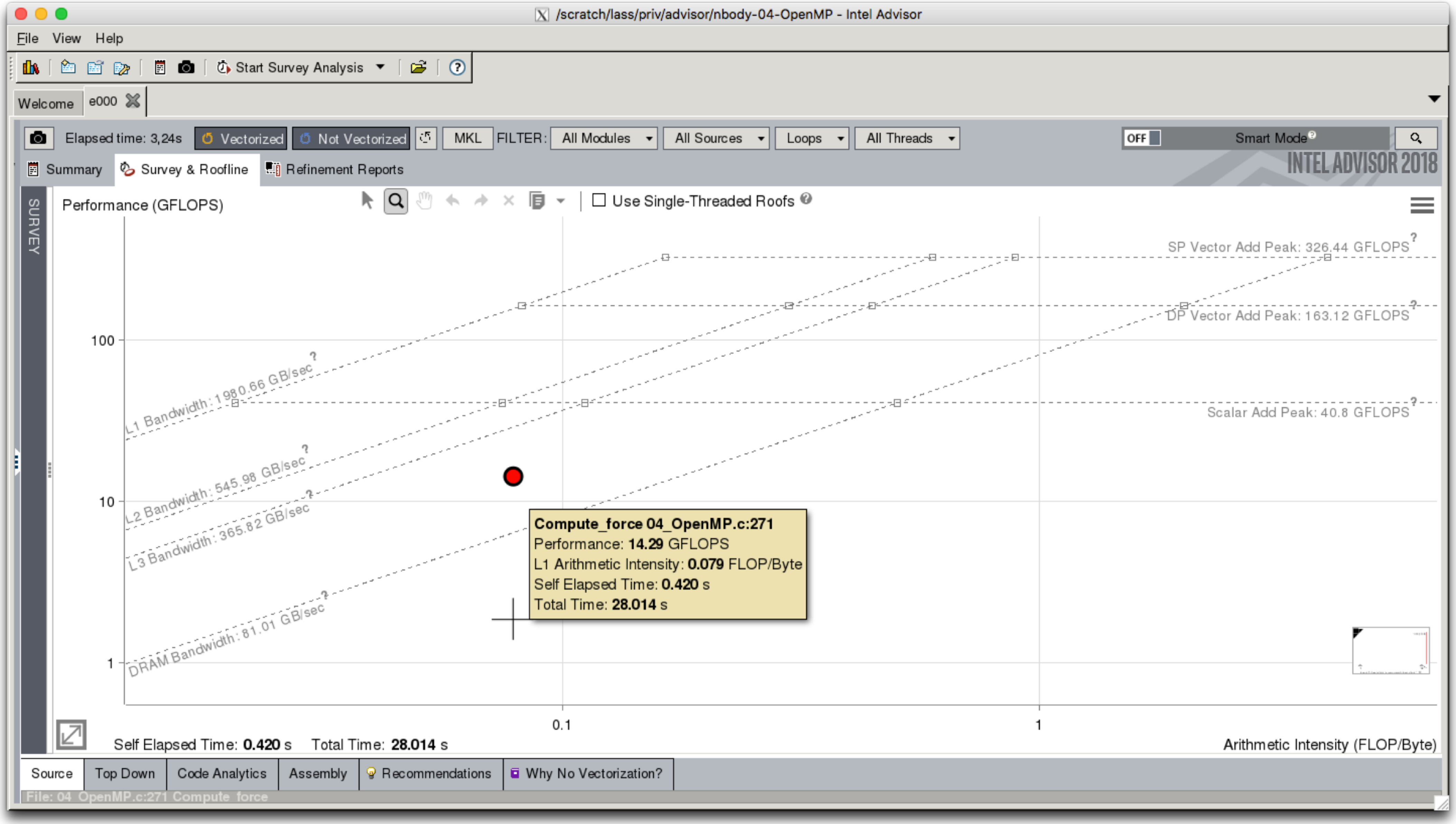
Compute_force 01_Reduced_Workload.c:257
Performance: **1.74** GFLOPS
L1 Arithmetic Intensity: **0.27** FLOP/Byte
Self Elapsed Time: **4.318** s
Total Time: **4.318** s

Self Elapsed Time: **4.318** s Total Time: **4.318** s

File: 01_Reduced_Workload.c:257 Compute_force

Compiler optimizations and auto vectorization

Performance Optimization: n-body



/scratch/lass/priv/advisor/nbody-04-OpenMP - Intel Advisor

Elapsed time: 3,24s

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

OFF Smart Mode

Compute_force 04_OpenMP.c:271
Performance: **14.29** GFLOPS
L1 Arithmetic Intensity: **0.079** FLOP/Byte
Self Elapsed Time: **0.420** s
Total Time: **28.014** s

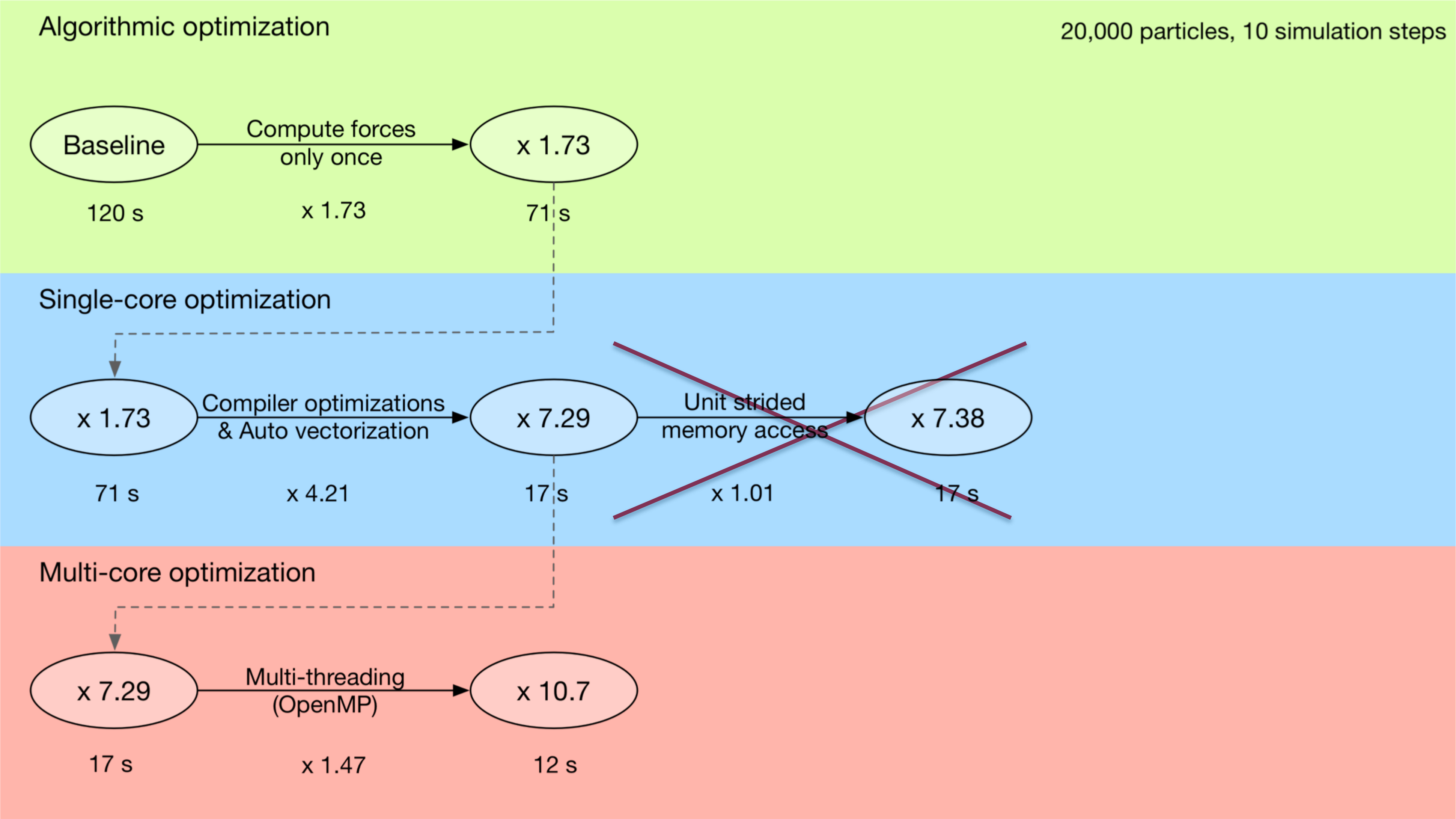
Self Elapsed Time: **0.420** s Total Time: **28.014** s

File: 04_OpenMP.c:271 Compute_force

File
View
Help
Start Survey Analysis
Welcome
e000
Vectorized
Not Vectorized
MKL
INTEL ADVISOR 2018
Summary
Survey & Roofline
Refinement Reports
SURVEY
Performance (GFLOPS)
☐ Use Single-Threaded Roofs
SP Vector Add Peak: 326.44 GFLOPS
DP Vector Add Peak: 163.12 GFLOPS
Scalar Add Peak: 40.8 GFLOPS
L1 Bandwidth: 1980.66 GB/sec
L2 Bandwidth: 545.98 GB/sec
L3 Bandwidth: 365.82 GB/sec
DRAM Bandwidth: 81.01 GB/sec
100
10
1
0.1
1
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?

Parallelized with OpenMP

Performance Optimization: n-body



Algorithmic optimization

20,000 particles, 10 simulation steps

Baseline → Compute forces only once → x 1.73

120 s | x 1.73 | 71 s

Single-core optimization

x 1.73 → Compiler optimizations & Auto vectorization → x 7.29 ~~→ Unit strided memory access → x 7.38~~

71 s | x 4.21 | 17 s | ~~x 1.01~~ | ~~17 s~~

Multi-core optimization

x 7.29 → Multi-threading (OpenMP) → x 10.7

17 s | x 1.47 | 12 s

Performance Optimization: n-body

- Intel VTune: Profiling tool with lots of different analysis types
  - Basic / Advanced Hotspot Analysis
  - Concurrency / Locks & Waits
  - General Exploration: Efficient use of microarchitecture
  - Memory bandwidth
  - IO access
  - ...

- Our parallelization did not work out, so do a *Concurrency* analysis

- Starting VTune on our systems:

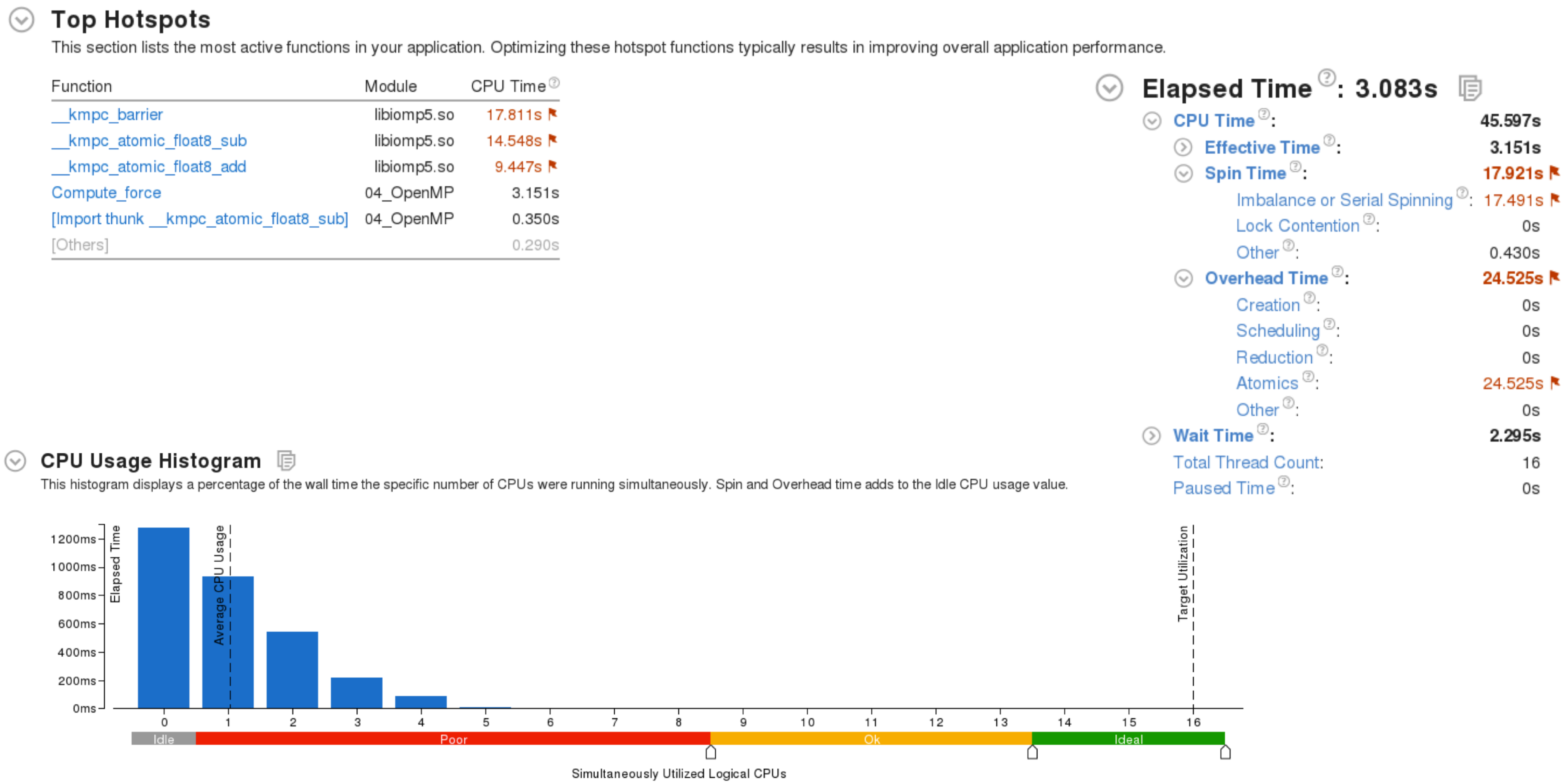
```
module load ps_xe_2018
amplxe-gui
```

Requesting a node with VTune support:

```
ccsalloc --res=rset=1:vtune=1:place=scatter:excl
```

Demo: Intel VTune Concurrency Analysis

Performance Optimization: n-body



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
__kmpc_barrier	libiomp5.so	17.811s
__kmpc_atomic_float8_sub	libiomp5.so	14.548s
__kmpc_atomic_float8_add	libiomp5.so	9.447s
Compute_force	04_OpenMP	3.151s
[Import thunk __kmpc_atomic_float8_sub]	04_OpenMP	0.350s
[Others]		0.290s

Elapsed Time: 3.083s

CPU Time:	45.597s
Effective Time:	3.151s
Spin Time:	17.921s
Imbalance or Serial Spinning:	17.491s
Lock Contention:	0s
Other:	0.430s
Overhead Time:	24.525s
Creation:	0s
Scheduling:	0s
Reduction:	0s
Atomics:	24.525s
Other:	0s
Wait Time:	2.295s
Total Thread Count:	16
Paused Time:	0s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

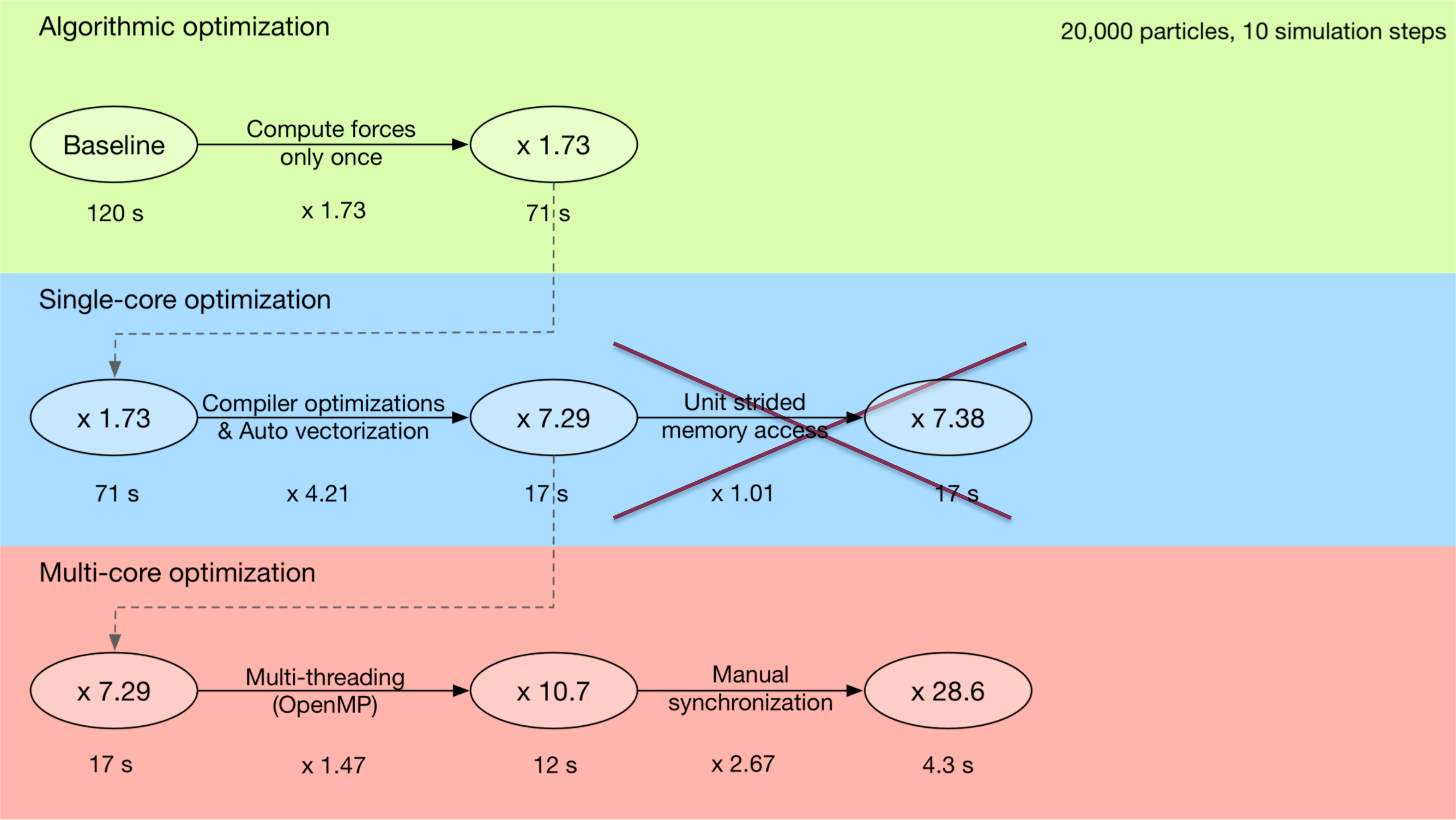
OpenMP: Basic implementation

Performance Optimization: n-body

- Atomic operations are costly
- Solution:
  - Let each thread store its own (partial) forces
  - Aggregate results after all threads have finished

```
#  pragma omp parallel num_threads(thread_count) default(none) \
shared(curr,forces,thread_count,delta_t,n,n_steps, \
        output_freq,loc_forces) \
    private(step, part, t)
  {
    int my_rank = omp_get_thread_num();
    int thread;
    for (step = 1; step <= n_steps; step++) {
      t = step*delta_t;
#     pragma omp for
      for (part = 0; part < thread_count*n; part++)
        loc_forces[part][X] = loc_forces[part][Y] = 0.0;
#     pragma omp for
      for (part = 0; part < n-1; part++)
        Compute_force(part, loc_forces + my_rank*n, curr, n);
#      pragma omp for
       for (part = 0; part < n; part++) {
        forces[part][X] = forces[part][Y] = 0.0;
        for (thread = 0; thread < thread_count; thread++) {
          forces[part][X] += loc_forces[thread*n + part][X];
          forces[part][Y] += loc_forces[thread*n + part][Y];
        }
      }
#     pragma omp for
      for (part = 0; part < n; part++)
        Update_part(part, forces, curr, n, delta_t);
    }
  }
```

Performance Optimization: n-body



Algorithmic optimization

20,000 particles, 10 simulation steps

Baseline → Compute forces only once → x 1.73

120 s — x 1.73 — 71 s

Single-core optimization

x 1.73 → Compiler optimizations & Auto vectorization → x 7.29 → ~~Unit strided memory access~~ → ~~x 7.38~~

71 s — x 4.21 — 17 s — ~~x 1.01~~ — ~~17 s~~

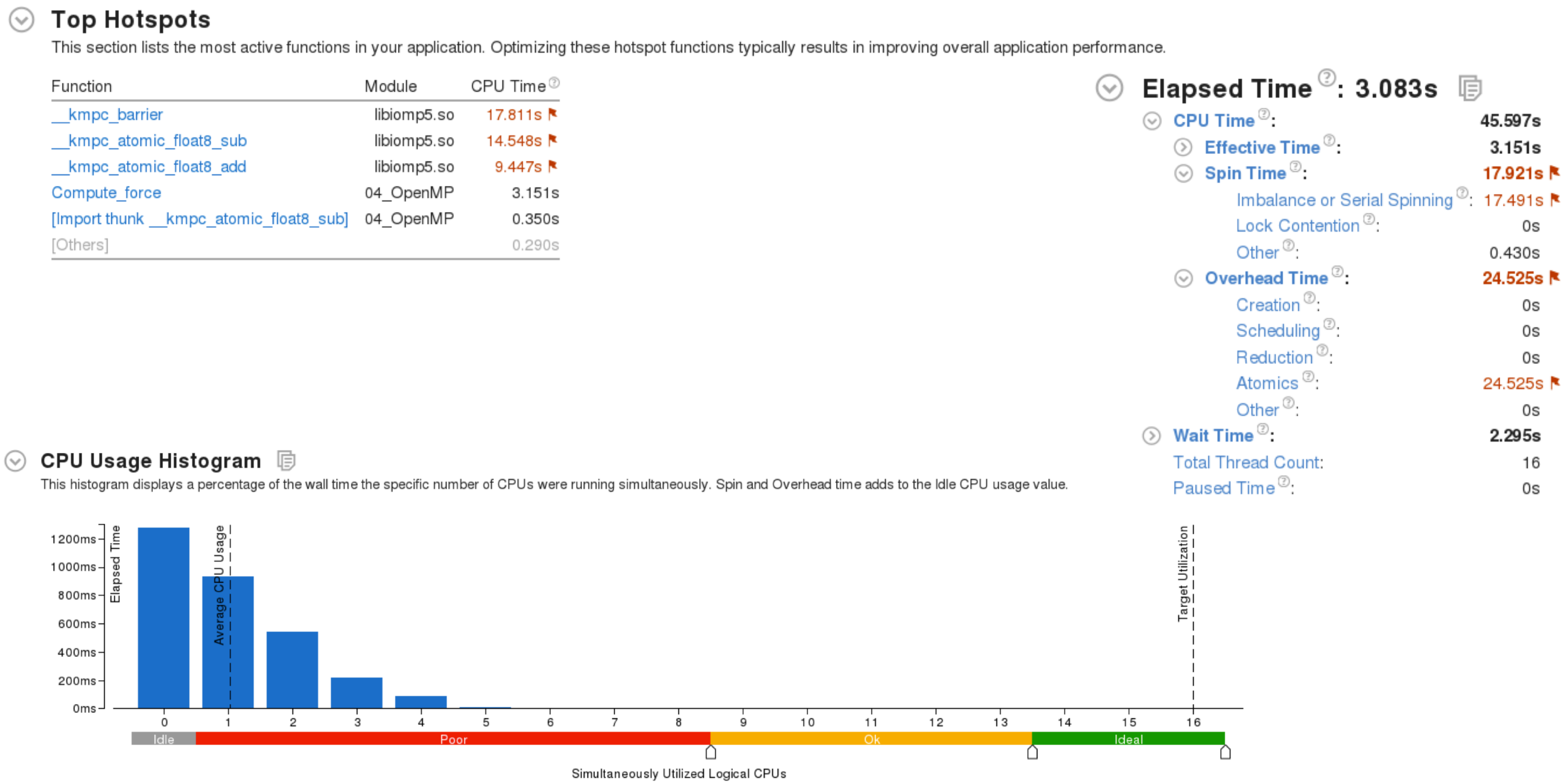
Multi-core optimization

x 7.29 → Multi-threading (OpenMP) → x 10.7 → Manual synchronization → x 28.6

17 s — x 1.47 — 12 s — x 2.67 — 4.3 s

Demo: Intel VTune Concurrency Analysis

Performance Optimization: n-body



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
__kmpc_barrier	libiomp5.so	17.811s ⚑
__kmpc_atomic_float8_sub	libiomp5.so	14.548s ⚑
__kmpc_atomic_float8_add	libiomp5.so	9.447s ⚑
Compute_force	04_OpenMP	3.151s
[Import thunk __kmpc_atomic_float8_sub]	04_OpenMP	0.350s
[Others]		0.290s

Elapsed Time: 3.083s

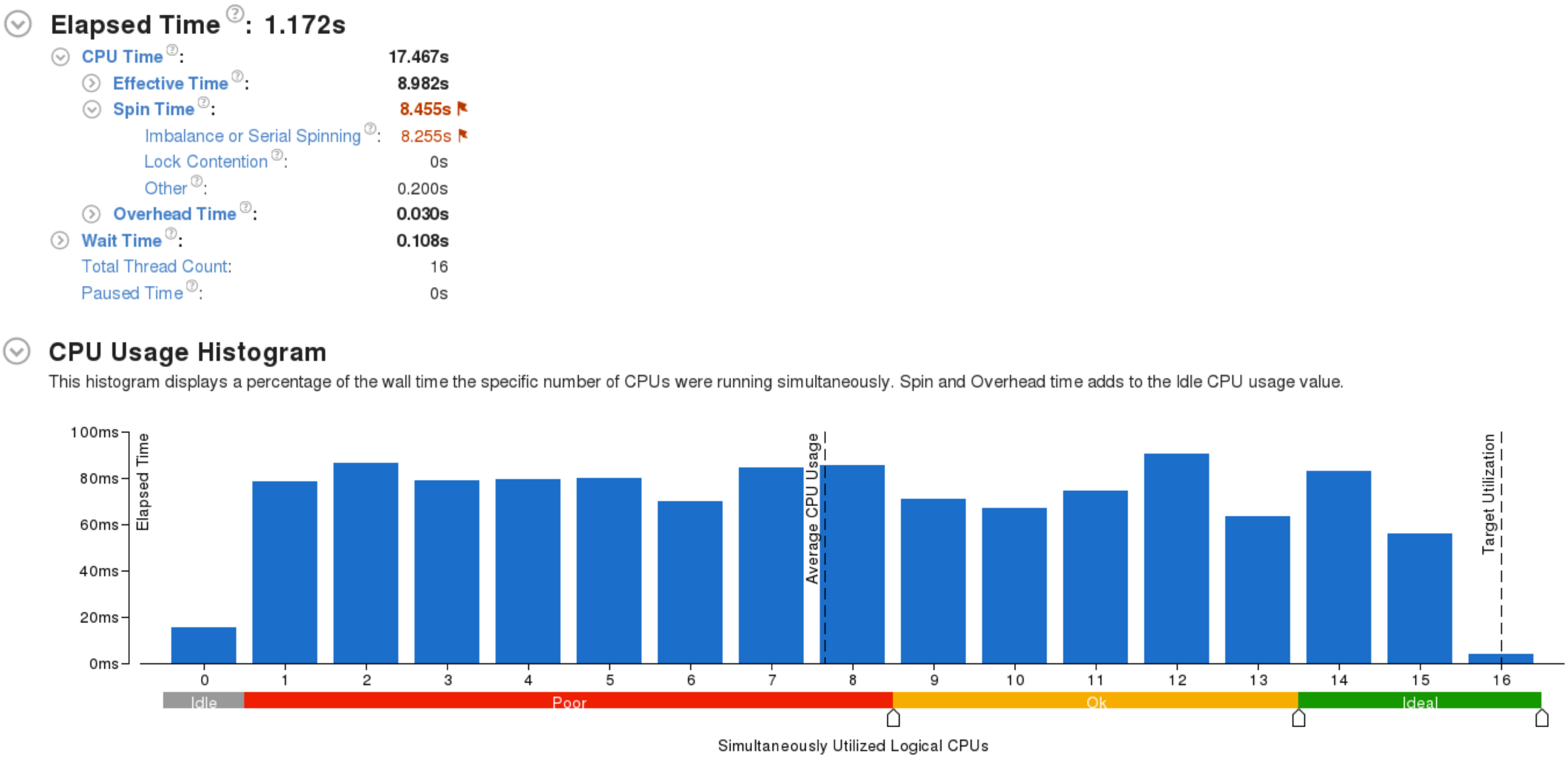
- CPU Time: 45.597s
  - Effective Time: 3.151s
  - Spin Time: 17.921s ⚑
    - Imbalance or Serial Spinning: 17.491s ⚑
    - Lock Contention: 0s
    - Other: 0.430s
  - Overhead Time: 24.525s ⚑
    - Creation: 0s
    - Scheduling: 0s
    - Reduction: 0s
    - Atomics: 24.525s ⚑
    - Other: 0s
- Wait Time: 2.295s
- Total Thread Count: 16
- Paused Time: 0s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

OpenMP: Basic implementation

Performance Optimization: n-body



Elapsed Time: 1.172s

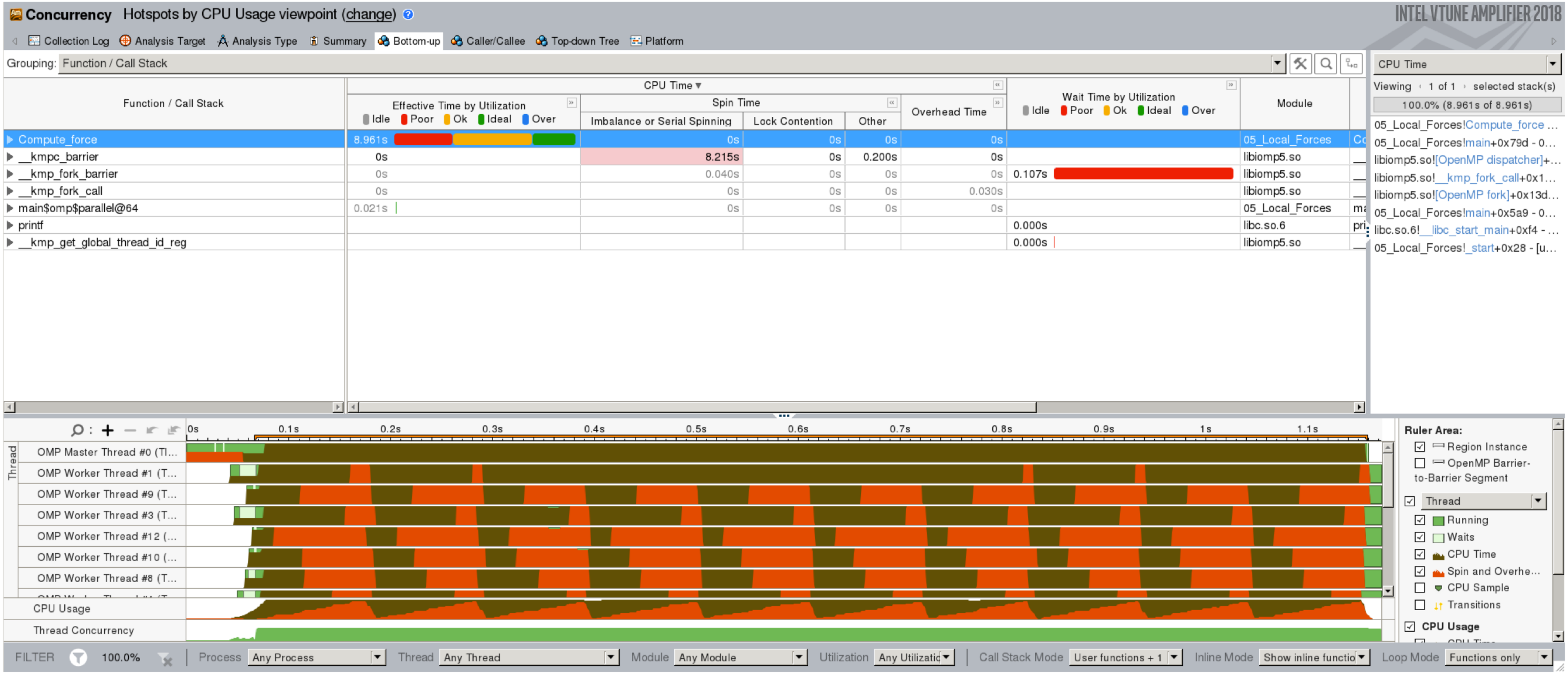
CPU Time:	17.467s
Effective Time:	8.982s
Spin Time:	8.455s ⚑
Imbalance or Serial Spinning:	8.255s ⚑
Lock Contention:	0s
Other:	0.200s
Overhead Time:	0.030s
Wait Time:	0.108s
Total Thread Count:	16
Paused Time:	0s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

OpenMP: Explicit synchronization

Performance Optimization: n-body



Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	CPU Time: Spin Time: Imbalance or Serial Spinning	CPU Time: Spin Time: Lock Contention	CPU Time: Spin Time: Other	CPU Time: Overhead Time	Wait Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Module	
Compute_force	8.961s	0s	0s	0s	0s		05_Local_Forces	Co
__kmpc_barrier	0s	8.215s	0s	0.200s	0s		libiomp5.so	__
__kmp_fork_barrier	0s	0.040s	0s	0s	0s	0.107s	libiomp5.so	__
__kmp_fork_call	0s	0s	0s	0s	0.030s		libiomp5.so	__
main$omp$parallel@64	0.021s	0s	0s	0s	0s		05_Local_Forces	ma
printf						0.000s	libc.so.6	pri
__kmp_get_global_thread_id_reg						0.000s	libiomp5.so	__

CPU Time

Viewing 1 of 1 selected stack(s)

100.0% (8.961s of 8.961s)

05_Local_Forces!Compute_force ...
05_Local_Forces!main+0x79d - 0...
libiomp5.so![OpenMP dispatcher]+...
libiomp5.so!__kmp_fork_call+0x1...
libiomp5.so![OpenMP fork]+0x13d...
05_Local_Forces!main+0x5a9 - 0...
libc.so.6!__libc_start_main+0xf4 - ...
05_Local_Forces!_start+0x28 - [u...

0s
0.1s
0.2s
0.3s
0.4s
0.5s
0.6s
0.7s
0.8s
0.9s
1s
1.1s
Thread
OMP Master Thread #0 (TI...
OMP Worker Thread #1 (T...
OMP Worker Thread #9 (T...
OMP Worker Thread #3 (T...
OMP Worker Thread #12 (...
OMP Worker Thread #10 (...
OMP Worker Thread #8 (T...
CPU Usage
Thread Concurrency

Ruler Area:

☑ Region Instance
☐ OpenMP Barrier-to-Barrier Segment

☑ Thread
☑ Running
☑ Waits
☑ CPU Time
☑ Spin and Overhe...
☐ CPU Sample
☐ Transitions

☑ CPU Usage

FILTER 100.0% | Process: Any Process | Thread: Any Thread | Module: Any Module | Utilization: Any Utilizatio | Call Stack Mode: User functions + 1 | Inline Mode: Show inline functio | Loop Mode: Functions only

OpenMP: Explicit synchronization

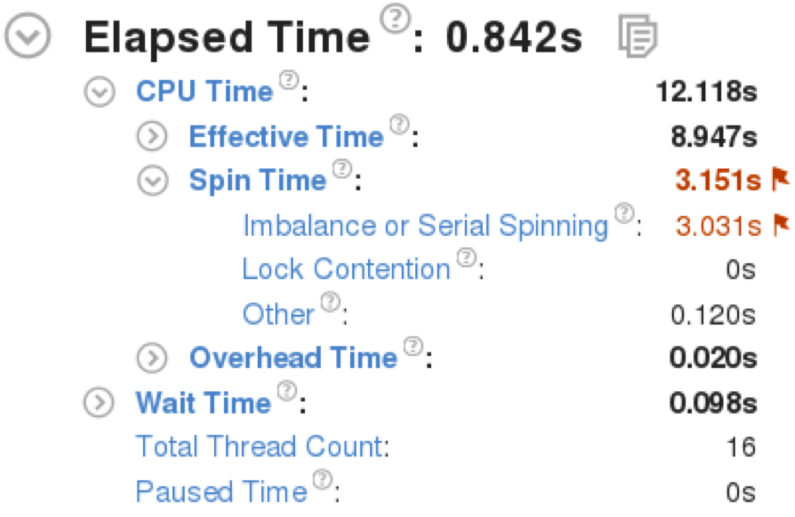
Performance Optimization: n-body

- Work imbalance between the threads:
  - First particle: all forces have to be calculated
  - Last particle: no forces have to be calculated
  - First thread has the most work, last thread the least

- Solution: Cyclic assignments from particles to threads

```
# pragma omp for schedule(static,1)
for (part = 0; part < n-1; part++)
    Compute_force(part, loc_forces + my_rank*n, curr, n);
```

Performance Optimization: n-body

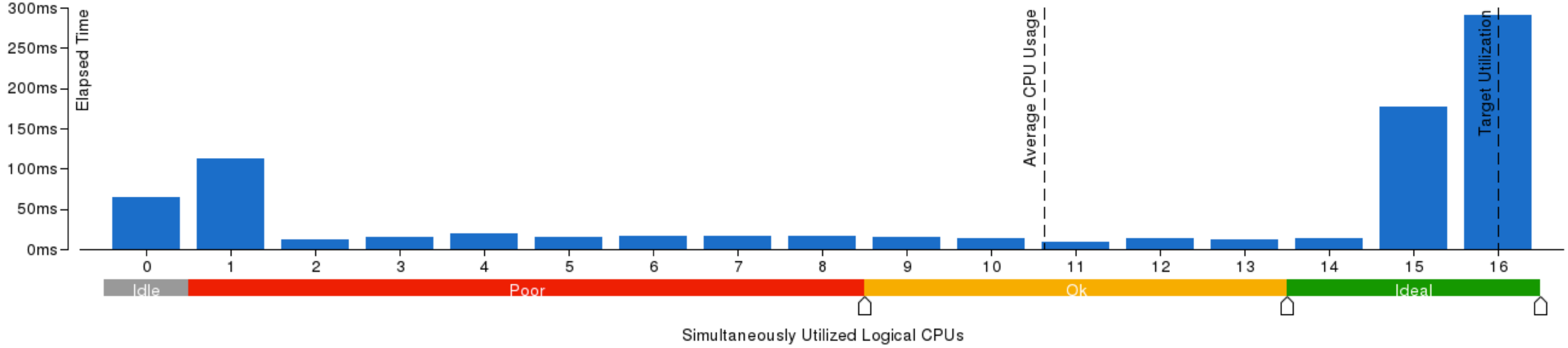


Elapsed Time: 0.842s

CPU Time:	12.118s
Effective Time:	8.947s
Spin Time:	3.151s ⚑
Imbalance or Serial Spinning:	3.031s ⚑
Lock Contention:	0s
Other:	0.120s
Overhead Time:	0.020s
Wait Time:	0.098s
Total Thread Count:	16
Paused Time:	0s

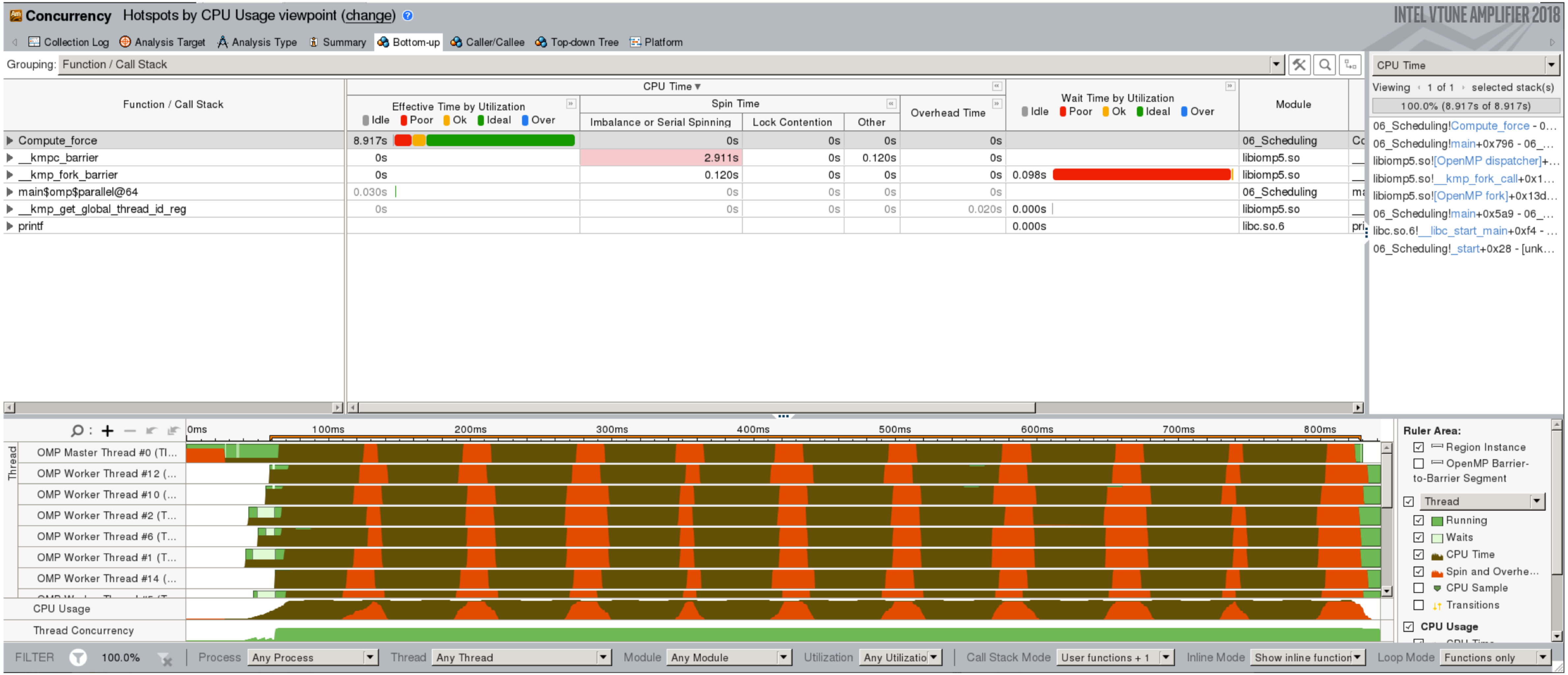
CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



OpenMP: Load balancing

Performance Optimization: n-body



Concurrency Hotspots by CPU Usage viewpoint (change)

INTEL VTUNE AMPLIFIER 2018

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

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▾					Wait Time by Utilization Idle Poor Ok Ideal Over	Module	
	Effective Time by Utilization Idle Poor Ok Ideal Over	Spin Time			Overhead Time			
		Imbalance or Serial Spinning	Lock Contention	Other				
▶ Compute_force	8.917s	0s	0s	0s	0s		06_Scheduling	Cc
▶ __kmpc_barrier	0s	2.911s	0s	0.120s	0s		libiomp5.so	__
▶ __kmp_fork_barrier	0s	0.120s	0s	0s	0s	0.098s	libiomp5.so	__
▶ main$omp$parallel@64	0.030s	0s	0s	0s	0s		06_Scheduling	ma
▶ __kmp_get_global_thread_id_reg	0s	0s	0s	0s	0.020s	0.000s	libiomp5.so	__
▶ printf						0.000s	libc.so.6	pri

CPU Time

Viewing 1 of 1 selected stack(s)

100.0% (8.917s of 8.917s)

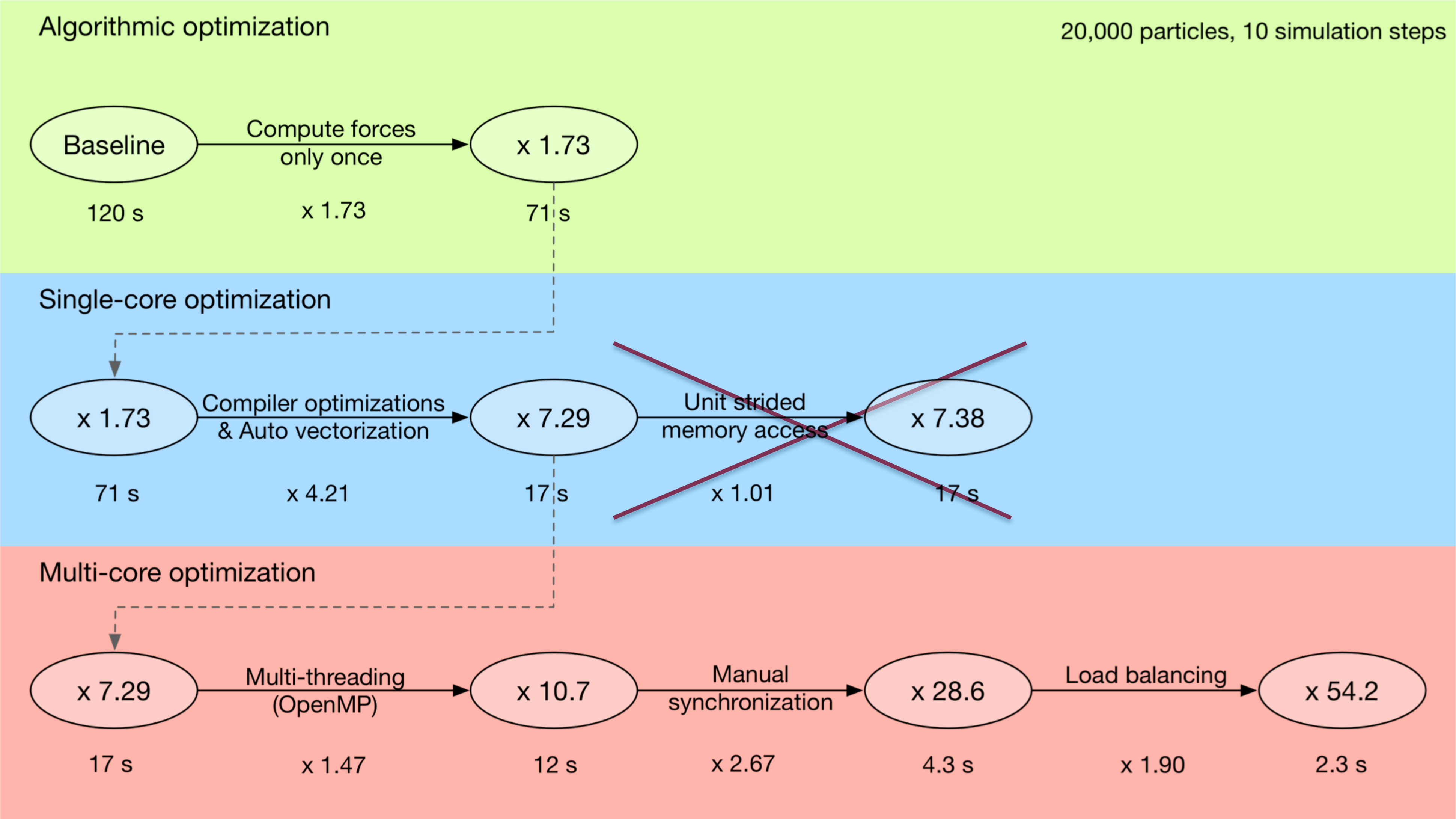
06_Scheduling!Compute_force - 0…
06_Scheduling!main+0x796 - 06_…
libiomp5.so![OpenMP dispatcher]+…
libiomp5.so!__kmp_fork_call+0x1…
libiomp5.so![OpenMP fork]+0x13d…
06_Scheduling!main+0x5a9 - 06_…
libc.so.6!__libc_start_main+0xf4 - …
06_Scheduling!_start+0x28 - [unk…

Ruler Area:
☑ Region Instance
☐ OpenMP Barrier-to-Barrier Segment
☑ Thread
☑ Running
☑ Waits
☑ CPU Time
☑ Spin and Overhe…
☐ CPU Sample
☐ Transitions
☑ CPU Usage

FILTER 100.0% | Process: Any Process | Thread: Any Thread | Module: Any Module | Utilization: Any Utilizatio | Call Stack Mode: User functions + 1 | Inline Mode: Show inline function | Loop Mode: Functions only

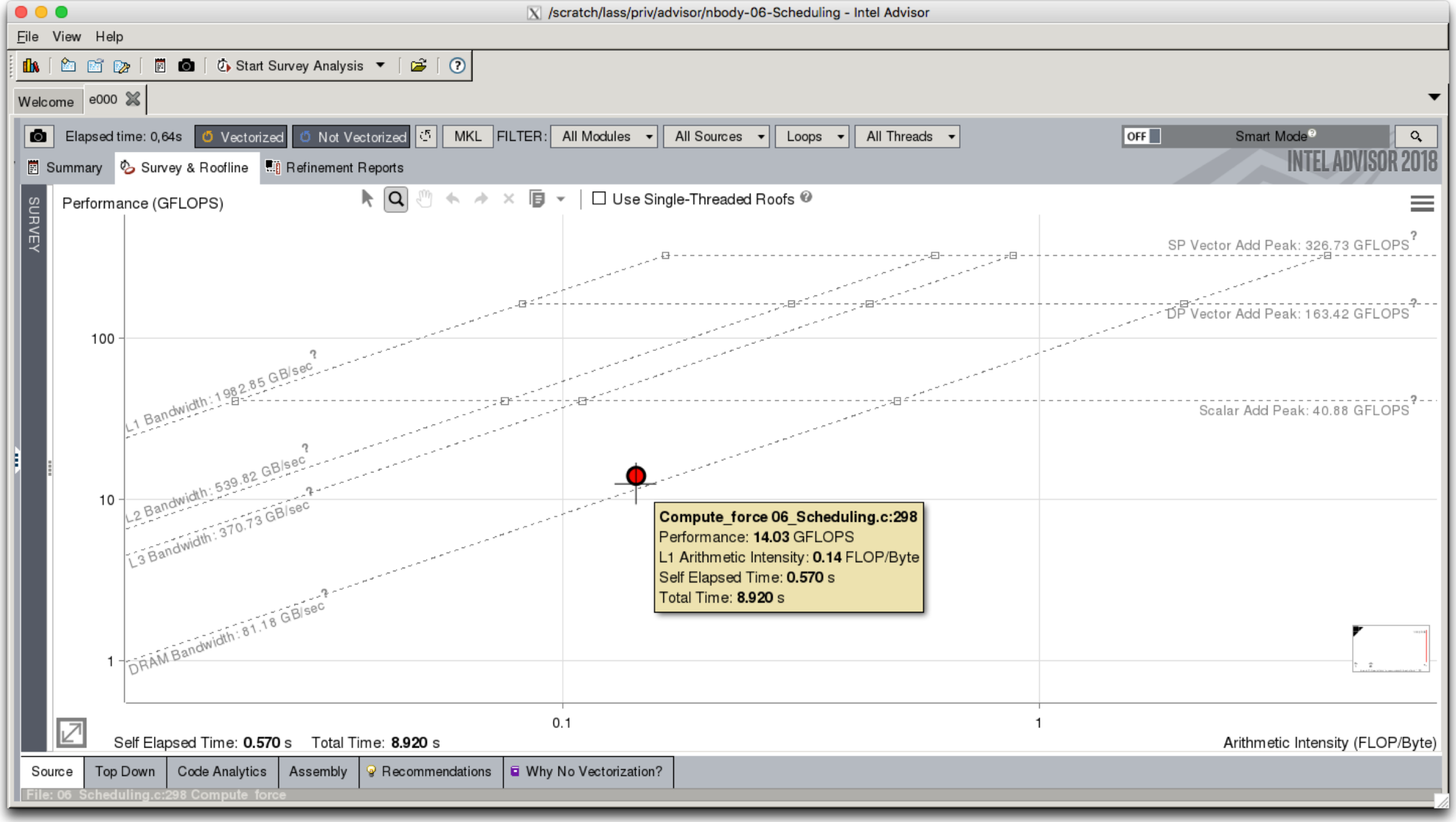
OpenMP: Load balancing

Performance Optimization: n-body



Demo: Back to Intel Advisor

Performance Optimization: n-body



/scratch/lass/priv/advisor/nbody-06-Scheduling - Intel Advisor
File View Help
Start Survey Analysis
Welcome
e000
Elapsed time: 0,64s
Vectorized
Not Vectorized
MKL
FILTER:
All Modules
All Sources
Loops
All Threads
OFF
Smart Mode
INTEL ADVISOR 2018
Summary
Survey & Roofline
Refinement Reports
SURVEY
Performance (GFLOPS)
Use Single-Threaded Roofs
SP Vector Add Peak: 326.73 GFLOPS
DP Vector Add Peak: 163.42 GFLOPS
Scalar Add Peak: 40.88 GFLOPS
L1 Bandwidth: 1982.85 GB/sec
L2 Bandwidth: 539.82 GB/sec
L3 Bandwidth: 370.73 GB/sec
DRAM Bandwidth: 81.18 GB/sec
100
10
1
0.1
1
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
File: 06_Scheduling.c:298 Compute_force

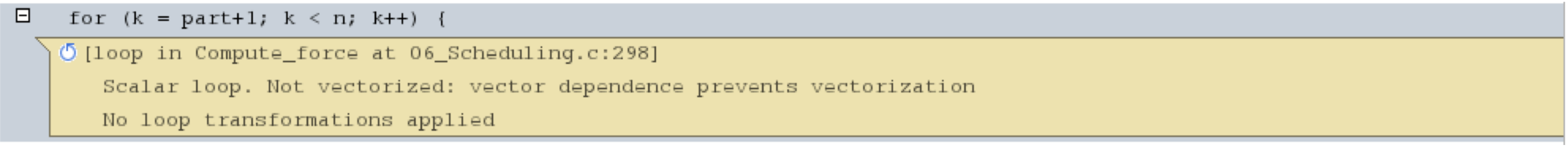
Compute_force 06_Scheduling.c:298
Performance: **14.03** GFLOPS
L1 Arithmetic Intensity: **0.14** FLOP/Byte
Self Elapsed Time: **0.570** s
Total Time: **8.920** s

Self Elapsed Time: **0.570** s Total Time: **8.920** s

OpenMP: Load balanced version

Performance Optimization: n-body

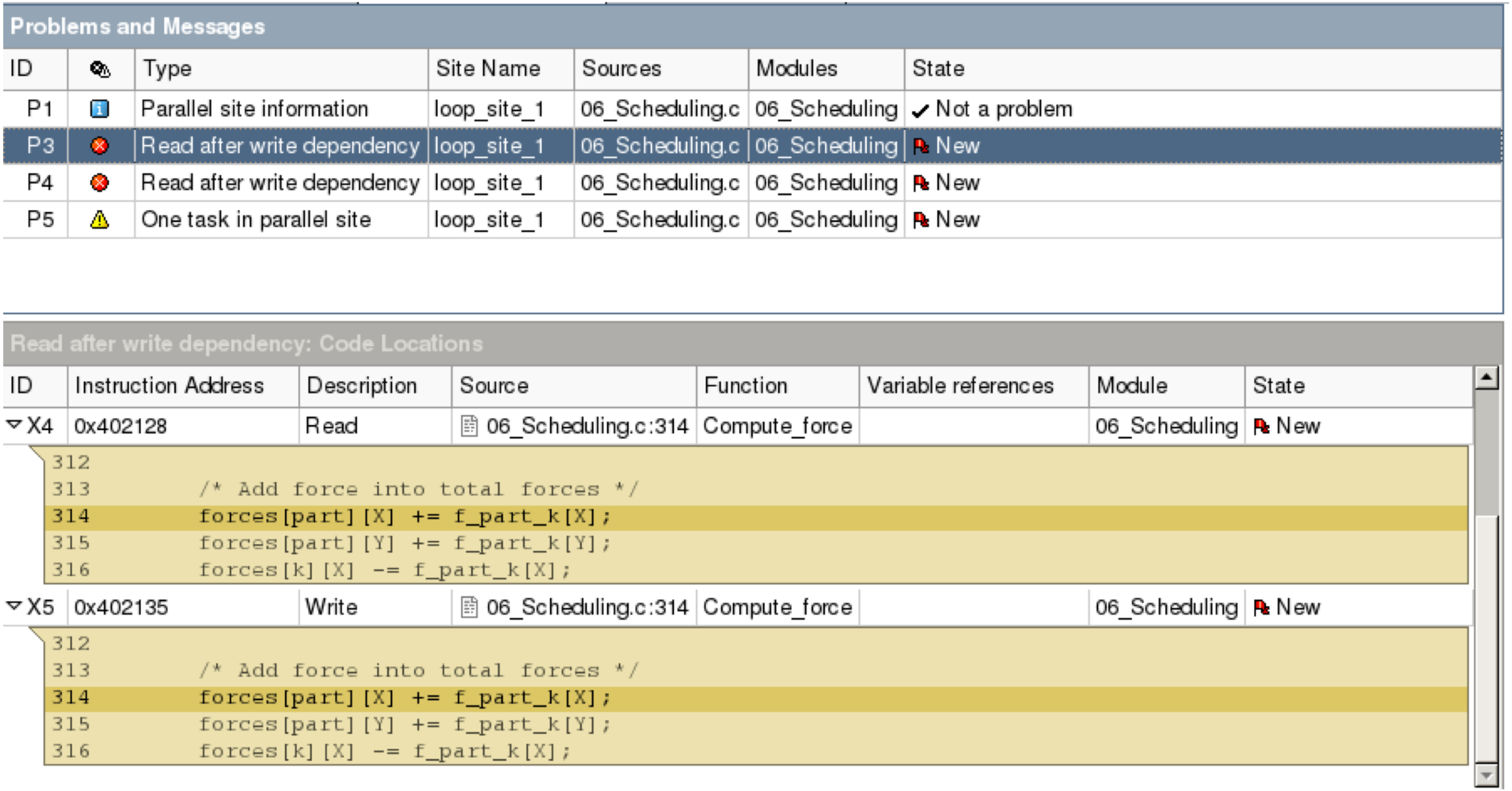
- Somewhere on our way we lost vectorization



```
for (k = part+1; k < n; k++) {
  [loop in Compute_force at 06_Scheduling.c:298]
     Scalar loop. Not vectorized: vector dependence prevents vectorization
     No loop transformations applied
```

- Advisor can do a memory dependency analysis

Performance Optimization: n-body



Problems and Messages

ID		Type	Site Name	Sources	Modules	State
P1	ℹ	Parallel site information	loop_site_1	06_Scheduling.c	06_Scheduling	✓ Not a problem
P3	⊗	Read after write dependency	loop_site_1	06_Scheduling.c	06_Scheduling	⚑ New
P4	⊗	Read after write dependency	loop_site_1	06_Scheduling.c	06_Scheduling	⚑ New
P5	⚠	One task in parallel site	loop_site_1	06_Scheduling.c	06_Scheduling	⚑ New

Read after write dependency: Code Locations

ID	Instruction Address	Description	Source	Function	Variable references	Module	State
▽ X4	0x402128	Read	06_Scheduling.c:314	Compute_force		06_Scheduling	⚑ New

```
312
313         /* Add force into total forces */
314         forces[part][X] += f_part_k[X];
315         forces[part][Y] += f_part_k[Y];
316         forces[k][X] -= f_part_k[X];
```

▽ X5	0x402135	Write	06_Scheduling.c:314	Compute_force		06_Scheduling	⚑ New

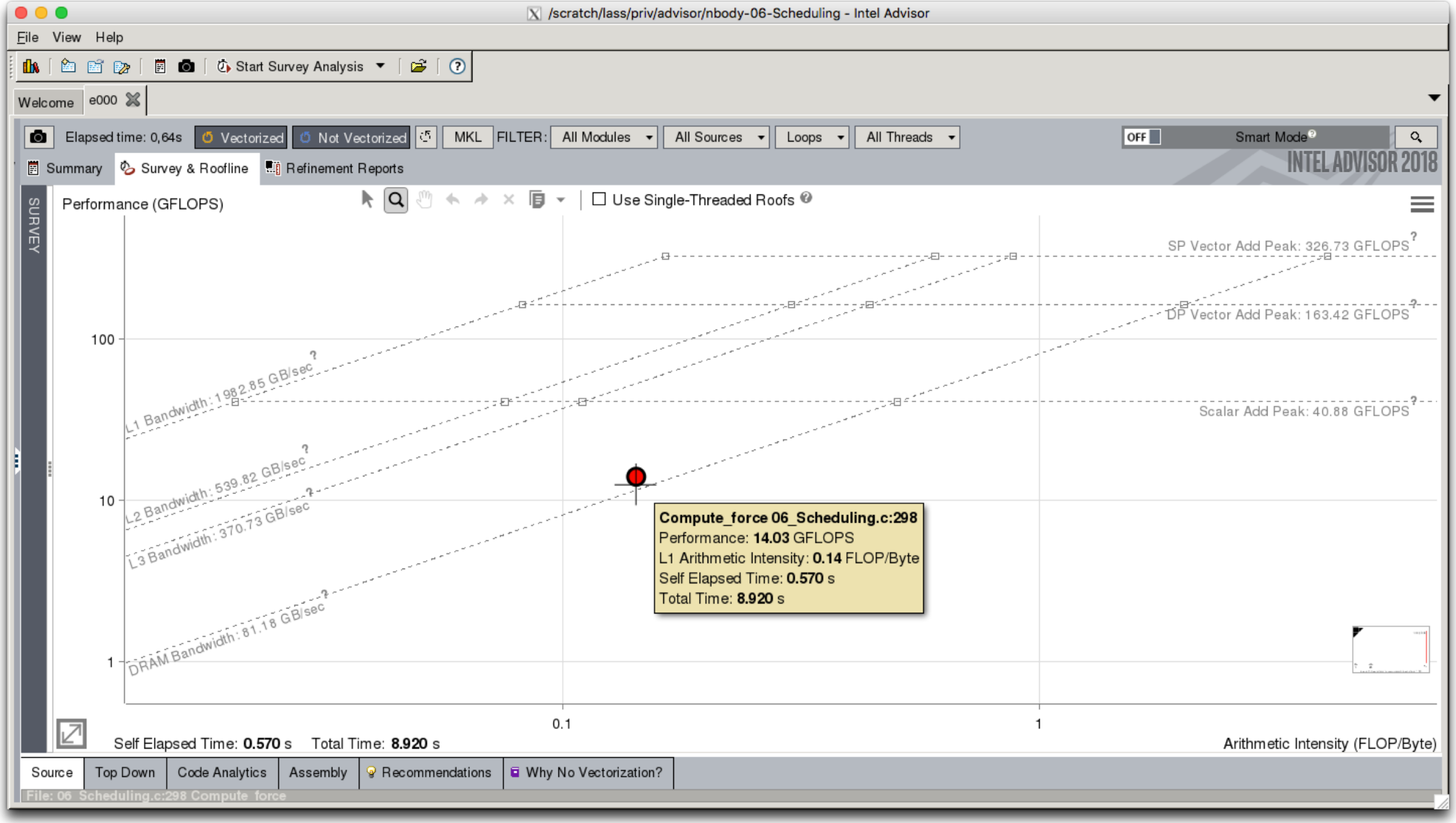
```
312
313         /* Add force into total forces */
314         forces[part][X] += f_part_k[X];
315         forces[part][Y] += f_part_k[Y];
316         forces[k][X] -= f_part_k[X];
```

Performance Optimization: n-body

- Solution: #pragma omp simd reduction(…)

```
   double forces_accu_X = 0.0;
   double forces_accu_Y = 0.0;

#pragma omp simd reduction(+:forces_accu_X,forces_accu_Y)
   for (k = part+1; k < n; k++) {

      /* Compute force between part and k */
      [...]

      /* Add force into total forces */
      forces_accu_X += f_part_k[X];
      forces_accu_Y += f_part_k[Y];
      forces[k][X] -= f_part_k[X];
      forces[k][Y] -= f_part_k[Y];

   }

   forces[part][X] += forces_accu_X;
   forces[part][Y] += forces_accu_Y;
```

Performance Optimization: n-body



/scratch/lass/priv/advisor/nbody-06-Scheduling - Intel Advisor
File View Help
Start Survey Analysis
Welcome e000
Vectorized
Not Vectorized
MKL
FILTER:
All Modules
All Sources
Loops
All Threads
OFF
Smart Mode
INTEL ADVISOR 2018
Summary
Survey & Roofline
Refinement Reports
SURVEY
Performance (GFLOPS)
Use Single-Threaded Roofs
SP Vector Add Peak: 326.73 GFLOPS
DP Vector Add Peak: 163.42 GFLOPS
Scalar Add Peak: 40.88 GFLOPS
L1 Bandwidth: 1982.85 GB/sec
L2 Bandwidth: 539.82 GB/sec
L3 Bandwidth: 370.73 GB/sec
DRAM Bandwidth: 81.18 GB/sec
100
10
1
0.1
1
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
File: 06_Scheduling.c:298 Compute_force

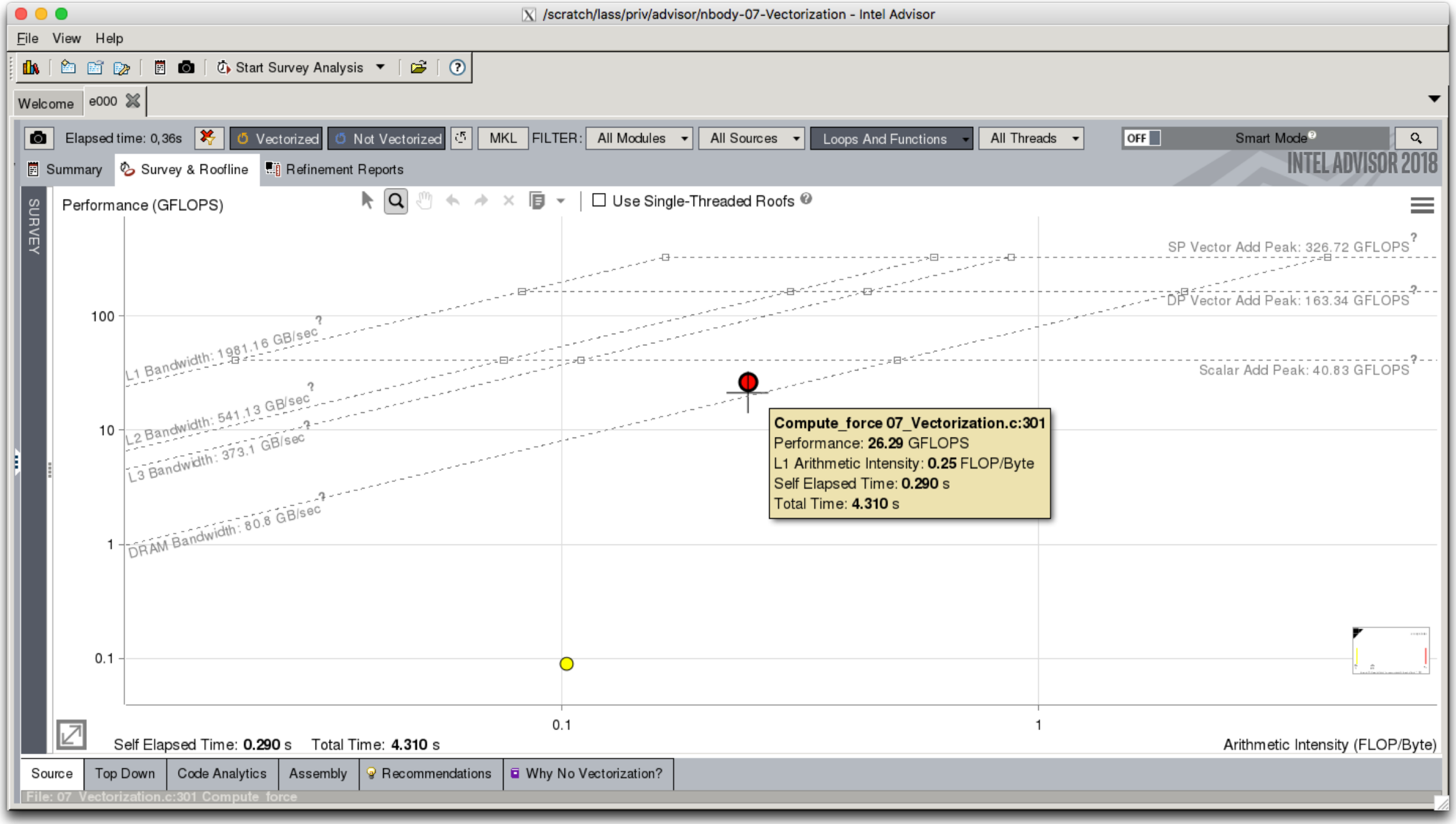
Elapsed time: 0,64s

Compute_force 06_Scheduling.c:298
Performance: **14.03** GFLOPS
L1 Arithmetic Intensity: **0.14** FLOP/Byte
Self Elapsed Time: **0.570** s
Total Time: **8.920** s

Self Elapsed Time: **0.570** s Total Time: **8.920** s

OpenMP: Load balanced version

Performance Optimization: n-body



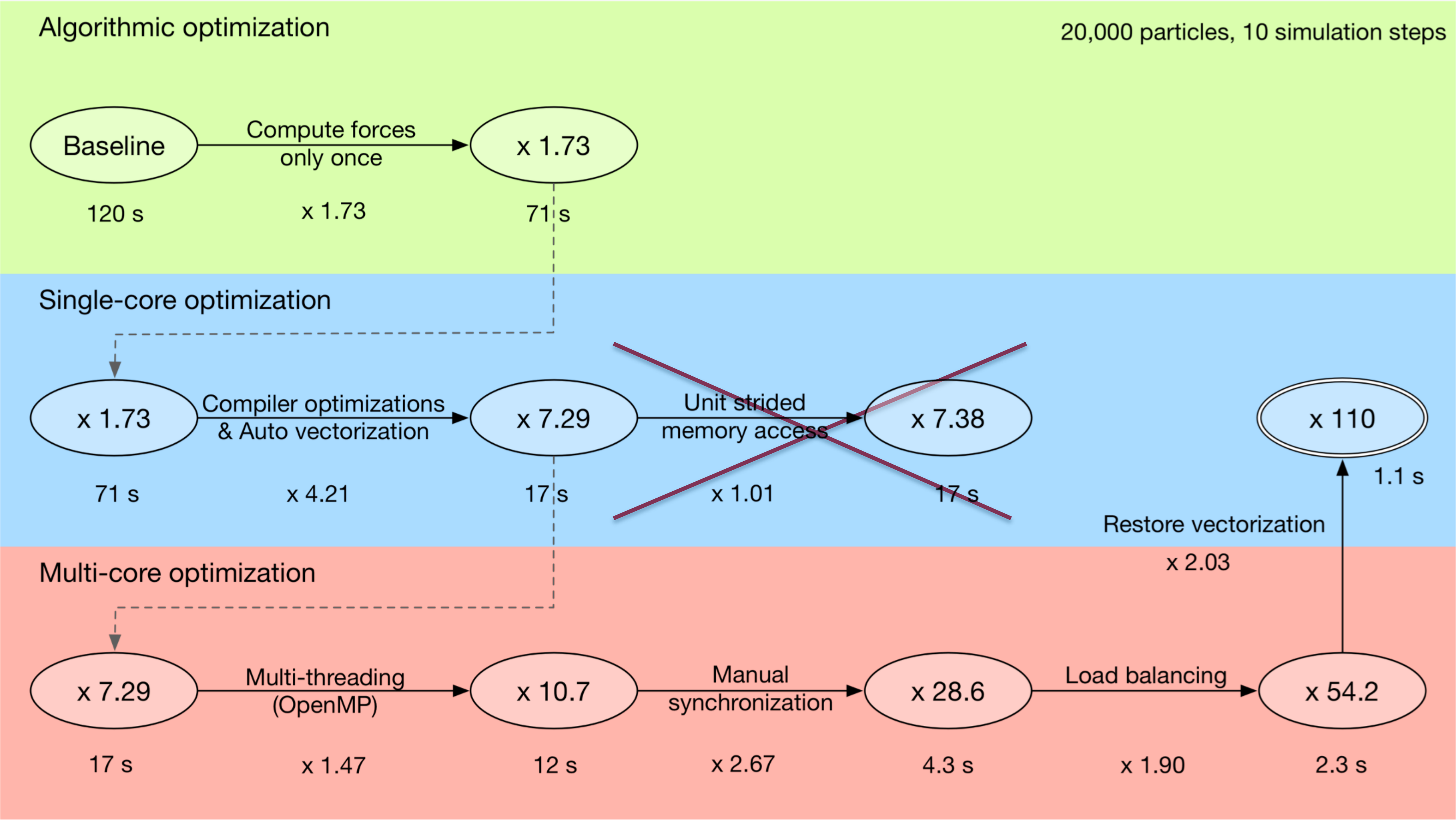
/scratch/lass/priv/advisor/nbody-07-Vectorization - Intel Advisor
File View Help
Start Survey Analysis
Welcome
e000
Elapsed time: 0,36s
Vectorized
Not Vectorized
MKL
FILTER:
All Modules
All Sources
Loops And Functions
All Threads
OFF
Smart Mode
INTEL ADVISOR 2018
Summary
Survey & Roofline
Refinement Reports
SURVEY
Performance (GFLOPS)
Use Single-Threaded Roofs
SP Vector Add Peak: 326.72 GFLOPS
DP Vector Add Peak: 163.34 GFLOPS
Scalar Add Peak: 40.83 GFLOPS
L1 Bandwidth: 1981.16 GB/sec
L2 Bandwidth: 541.13 GB/sec
L3 Bandwidth: 373.1 GB/sec
DRAM Bandwidth: 80.8 GB/sec
100
10
1
0.1
0.1
1
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
File: 07_Vectorization.c:301 Compute_force

Compute_force 07_Vectorization.c:301
Performance: **26.29** GFLOPS
L1 Arithmetic Intensity: **0.25** FLOP/Byte
Self Elapsed Time: **0.290** s
Total Time: **4.310** s

Self Elapsed Time: **0.290** s Total Time: **4.310** s

OpenMP: Load balanced and vectorized version

Performance Optimization: n-body

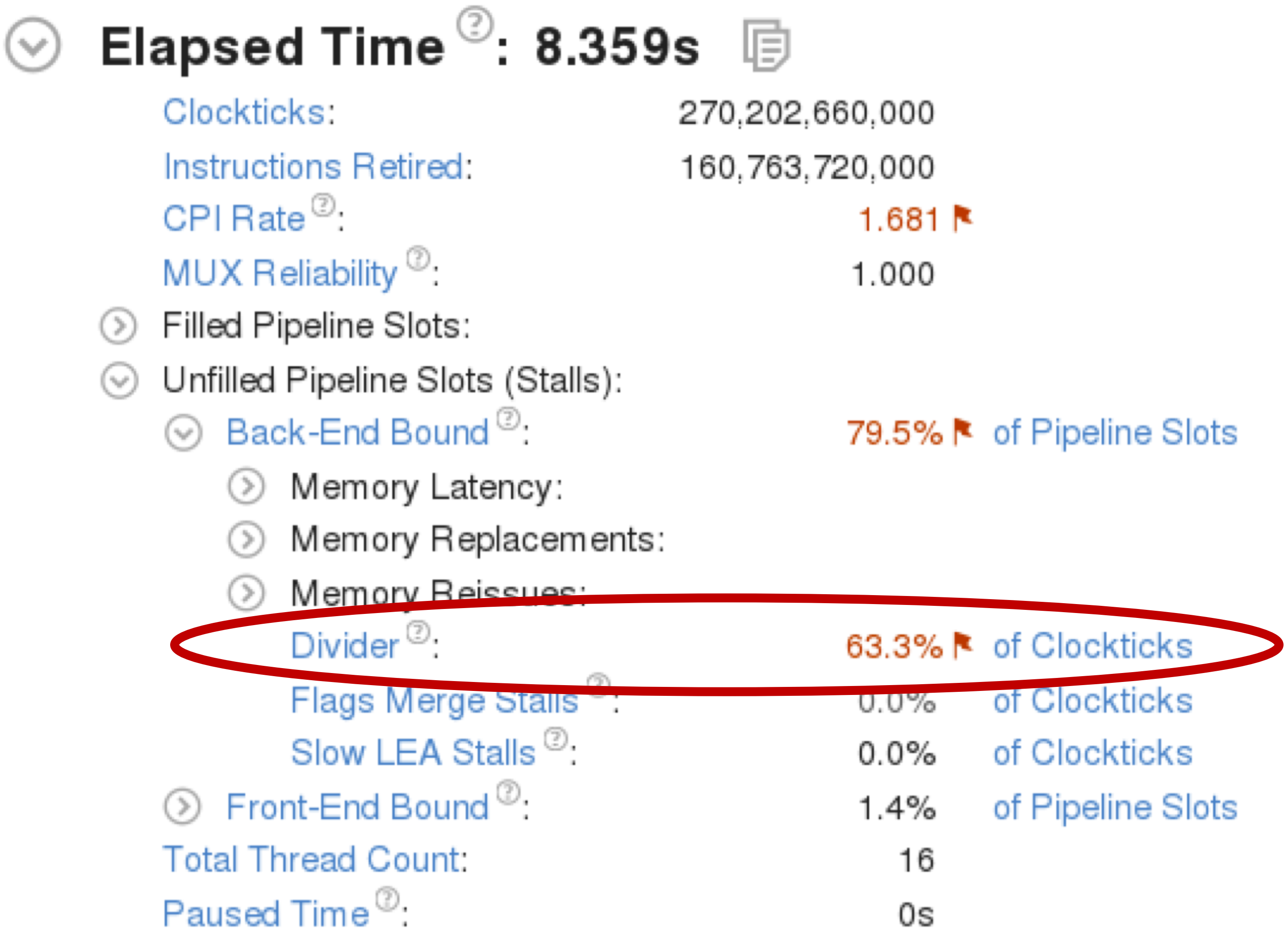


Performance Optimization: n-body

- Still, we only achieve 26 GFLOP/s where 163 GFLOP/s should be possible
- Are we memory or compute limited?
- Do a *General Exploration* in VTune

Demo: Intel VTune General Exploration

Performance Optimization: n-body



Elapsed Time: 8.359s

Clockticks:	270,202,660,000	
Instructions Retired:	160,763,720,000	
CPI Rate:	1.681 ⚑	
MUX Reliability:	1.000	
Filled Pipeline Slots:		
Unfilled Pipeline Slots (Stalls):		
Back-End Bound:	79.5% ⚑	of Pipeline Slots
Memory Latency:		
Memory Replacements:		
Memory Reissues:		
Divider:	63.3% ⚑	of Clockticks
Flags Merge Stalls:	0.0%	of Clockticks
Slow LEA Stalls:	0.0%	of Clockticks
Front-End Bound:	1.4%	of Pipeline Slots
Total Thread Count:	16	
Paused Time:	0s	

We spend > 60% of the time in our division!

Performance Optimization: n-body

- We achieved a 110x speedup by
  - Optimizing the algorithm
  - Using compiler optimizations
  - Using OpenMP for parallel execution
  - Reducing the synchronization effort
  - Balancing the load between threads
  - Making use of vectorization

- Tools can help in this process
  - We used VTune to find bottlenecks and load imbalance
  - We used Advisor to enable vectorization and optimize memory access patterns

- Optimization is an iterative process
  - You may have to revisit things you have optimized earlier

Profiling of MPI applications

Profiling of MPI applications

- Questions when optimizing an MPI application:
  - How much time is spent for communication?
  - How much data is transferred between the nodes?
  - Is the load balanced between all ranks?

- MPI implementations provide statistics and benchmarks to assist

- We focus on Intel MPI here

Profiling of MPI applications

- To collect basic statistics, export environment variable I_MPI_STATS

```
module load ps_xe_2018
export I_MPI_STATS=all

# local execution
mpirun -n 16 ./mpi_nbody_red 1024 10 0.001 10 g

# execution on OCuLUS
ccsalloc -I -c 64 impi -- ./mpi_nbody_red 524288 10 0.001 10 g
```

Profiling of MPI applications

- stats.ipm
- IPM: Integrated Performance Monitoring

```
#                         [total]      <avg>        min          max
# entries                 64           1            1            1
# wallclock               21905.1      342.267      341.467      344.917
# user                    21436.7      334.948      332.975      335.58
# system                  23.0293      0.359832     0.180638     0.881514
# mpi                     1231.47      19.2418      7.94775      33.3769
# %comm                                5.62186      2.31862      9.74161
# gflop/sec               NA           NA           NA           NA
# gbytes                  0            0            0            0
#
#
#                           [time]        [calls]      <%mpi>       <%wall>
# MPI_Sendrecv_replace      753.383       40960        61.18        3.44
# MPI_Init                  470.983       64           38.25        2.15
# MPI_Scatterv              6.15997       128          0.50         0.03
# MPI_Bcast                 0.94546       384          0.08         0.00
# MPI_Type_commit           0.000687122   128          0.00         0.00
# MPI_Finalize              0.000604153   64           0.00         0.00
# MPI_Type_free             0.000304937   128          0.00         0.00
# MPI_Type_contiguous       0.000223637   64           0.00         0.00
# MPI_Comm_size             6.91414e-05   64           0.00         0.00
# MPI_Type_vector           6.31809e-05   64           0.00         0.00
# MPI_Comm_rank             6.24657e-05   64           0.00         0.00
# MPI_Type_create_resized 5.38826e-05   64           0.00         0.00
# MPI_Type_get_extent       1.40667e-05   64           0.00         0.00
# MPI_Wtime                 1.26362e-05   128          0.00         0.00
# MPI_TOTAL                 1231.47       42368        100.00       5.62
```

Profiling of MPI applications

- stats.txt

- detailed per-rank statistics

```
~~~~ Process 0 of 64 on node node01-002 lifetime = 344917304.99

Data Transfers
Src    Dst    Amount(MB)      Transfers
-----------------------------------------
000 -->000   0.000000e+00   0
000 -->001   4.000028e+00   348
000 -->002   0.000000e+00   0
000 -->003   0.000000e+00   0
000 -->004   0.000000e+00   0
[...]
000 -->063   1.600000e+02   640

[...]

Communication Activity
Operation      Volume(MB)      Calls Min time    Avr time    Max time    Total time
-----------------------------------------
P2P
Csend          4.000028e+00   348   0.95        8.58        366.93      2985.95
CSendRecv      0.000000e+00   0     0.00        0.00        0.00        0.00
Send           0.000000e+00   0     0.00        0.00        0.00        0.00
SendRecv       1.600000e+02   640   445.13      24101.42    74103.12    15424910.31
Bsend          0.000000e+00   0     0.00        0.00        0.00        0.00
Rsend          0.000000e+00   0     0.00        0.00        0.00        0.00

[...]
```

Profiling of MPI applications

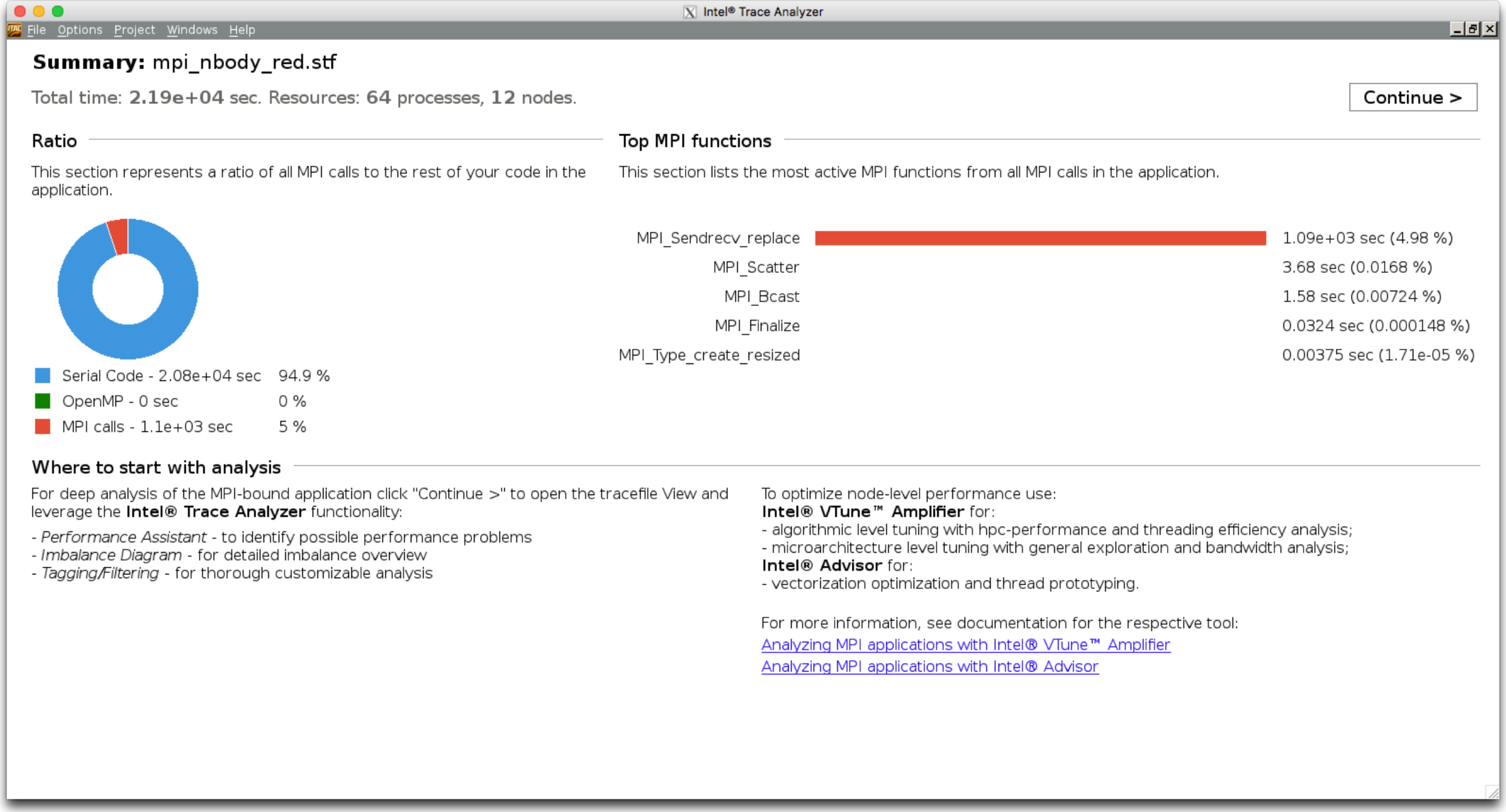
- Intel Trace Analyzer: Tool to visualize and analyze MPI traces

- Enable collection of traces by adding -trace to mpirun:

```
module load ps_xe_2018

# local execution
mpirun -trace -n 16 ./mpi_nbody_red 1024 10 0.001 10 g

# execution on OCuLUS
ccsalloc -I -c 64 impi -trace -- ./mpi_nbody_red 524288 10 0.001 10 g

# start trace analyzer
traceanalyzer mpi_nbody_red.stf
```

Demo: Intel Trace Analyzer

Profiling of MPI applications



Summary: mpi_nbody_red.stf

Total time: **2.19e+04** sec. Resources: **64** processes, **12** nodes.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦	Serial Code - 2.08e+04 sec	94.9 %
🟩	OpenMP - 0 sec	0 %
🟥	MPI calls - 1.1e+03 sec	5 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Sendrecv_replace	1.09e+03 sec (4.98 %)
MPI_Scatter	3.68 sec (0.0168 %)
MPI_Bcast	1.58 sec (0.00724 %)
MPI_Finalize	0.0324 sec (0.000148 %)
MPI_Type_create_resized	0.00375 sec (1.71e-05 %)

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

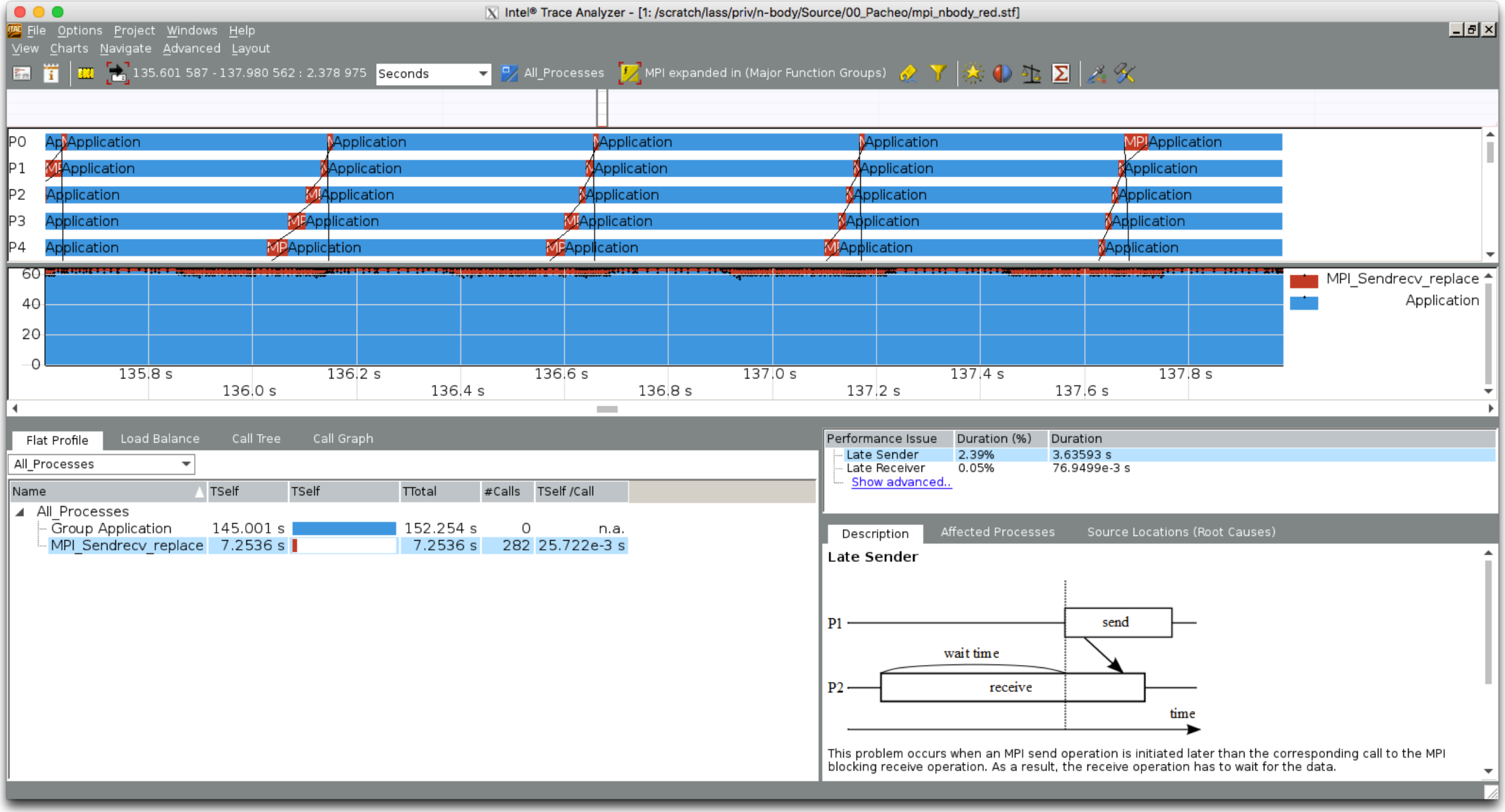
- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use:
Intel® VTune™ Amplifier for:
- algorithmic level tuning with hpc-performance and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;

Intel® Advisor for:
- vectorization optimization and thread prototyping.

For more information, see documentation for the respective tool:
Analyzing MPI applications with Intel® VTune™ Amplifier
Analyzing MPI applications with Intel® Advisor

Profiling of MPI applications



Intel® Trace Analyzer - [1: /scratch/lass/priv/n-body/Source/00_Pacheo/mpi_nbody_red.stf]

135.601 587 - 137.980 562 : 2.378 975 Seconds

All_Processes

MPI expanded in (Major Function Groups)

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
All_Processes					
Group Application	145.001 s		152.254 s	0	n.a.
MPI_Sendrecv_replace	7.2536 s		7.2536 s	282	25.722e-3 s

Performance Issue	Duration (%)	Duration
Late Sender	2.39%	3.63593 s
Late Receiver	0.05%	76.9499e-3 s
Show advanced..		

Late Sender

This problem occurs when an MPI send operation is initiated later than the corresponding call to the MPI blocking receive operation. As a result, the receive operation has to wait for the data.

Acknowledgements

- This lecture is based materials from these sources
  - Tutorial: Performance Tuning of Scientific Codes with the Roofline Model, SuperComputing 2017
  - Tutorial: Advanced OpenMP, Supercomputing 2017
  - Tutorial: Application Optimization and Vectorization, Intel 2017
  - Intel 64 and IA-32 Architectures Optimization Reference Manual
  - Intel Developer Zone documentation on vectorization

Change Log

- 1.1.0 (2018-02-02)
  - added slides about MPI profiling
- 1.0.0 (2018-01-30)
  - initial version of slides