Intel® Advisor Vectorization Optimization and Thread Prototyping

Software Must Vectorize & Thread or Performance Dies

True today – More true tomorrow – Difference can be substantial!

- Vectorization - Have you:
  - Recompiled for AVX2 or AVX512 with little gain
  - Wondered where to vectorize?
  - Recoded intrinsics for new arch.?
  - Struggled with compiler reports?
- Threading - Have you:
  - Threaded an app, but seen little benefit?
  - Hit a "scalability barrier"?
  - Delayed release due to sync. errors?

"**Intel® Advisor's** Vectorization Advisor fills a gap in code performance analysis. It can guide the informed user to better exploit the vector capabilities of modern processors and coprocessors."

Dr. Luigi Iapichino
Scientific Computing Expert
Leibniz Supercomputing Centre

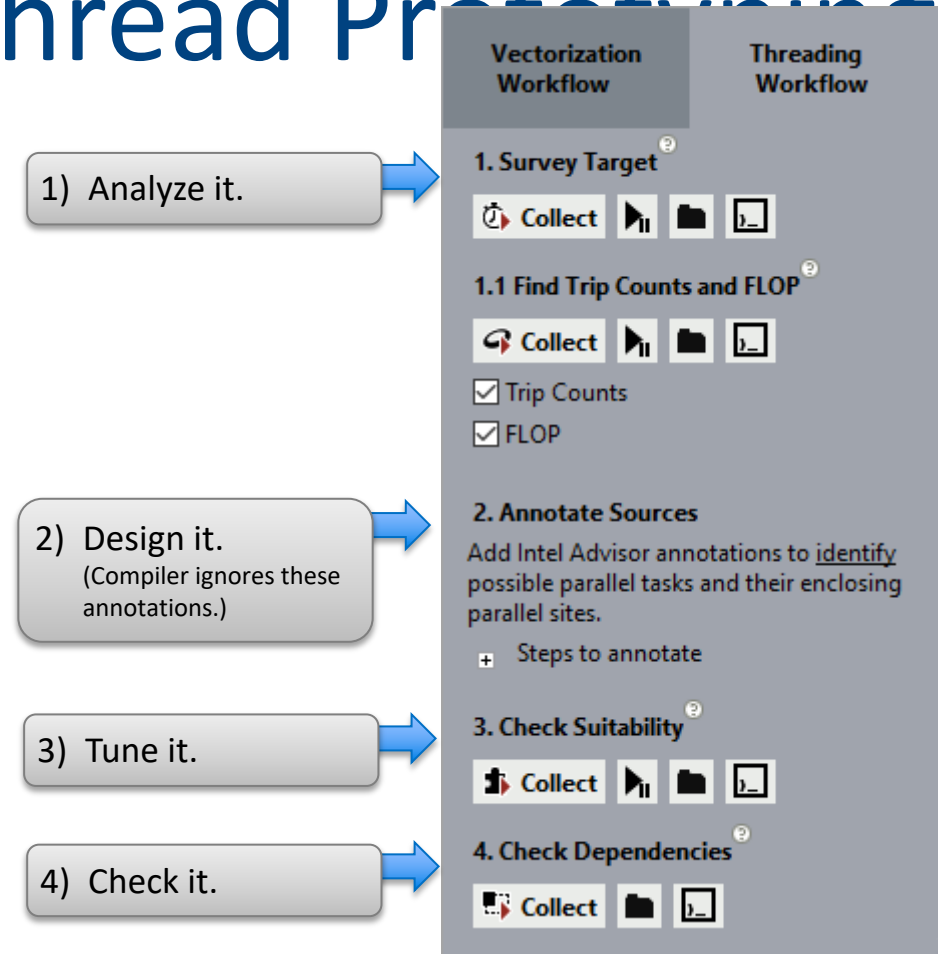
"**Intel® Advisor** has allowed us to quickly prototype ideas for parallelism, saving developer time and effort"

Simon Hammond
Senior Technical Staff
Sandia National Laboratories

Intel Advisor - Thread Prototyping

Design Parallelism

- No disruption to regular development
- All test cases continue to work
- Tune and debug the design before you implement it



Intel Advisor Annotation Concepts

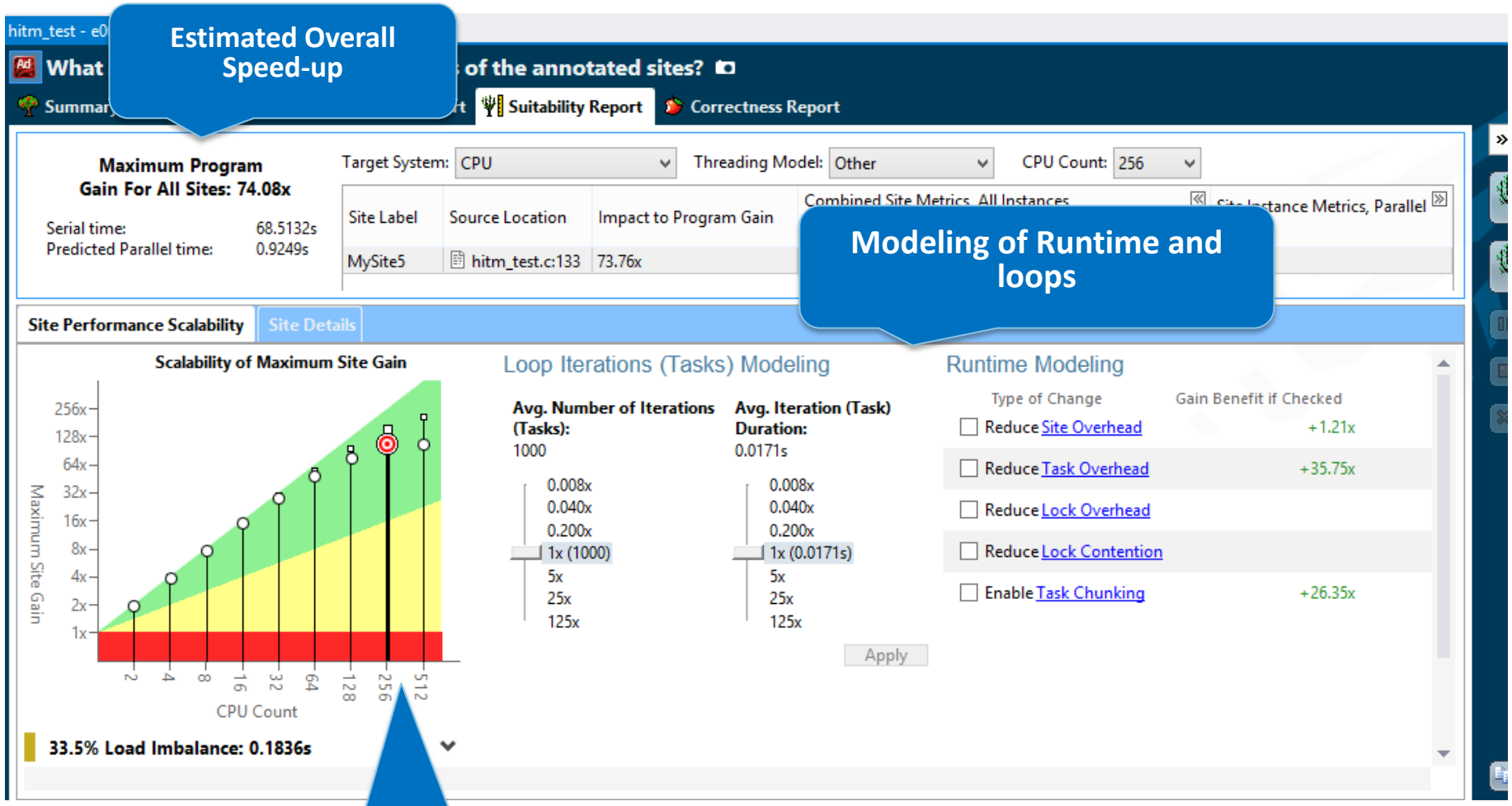
- Advisor uses 3 primary concepts to create a model
  - **SITE**
    - A region of code in your application you want to transform into parallel code
  - **TASK**
    - The region of code in a SITE you want to execute in parallel with the rest of the code in the SITE
  - **LOCK**
    - Mark regions of code in a TASK which must be serialized

NOTE

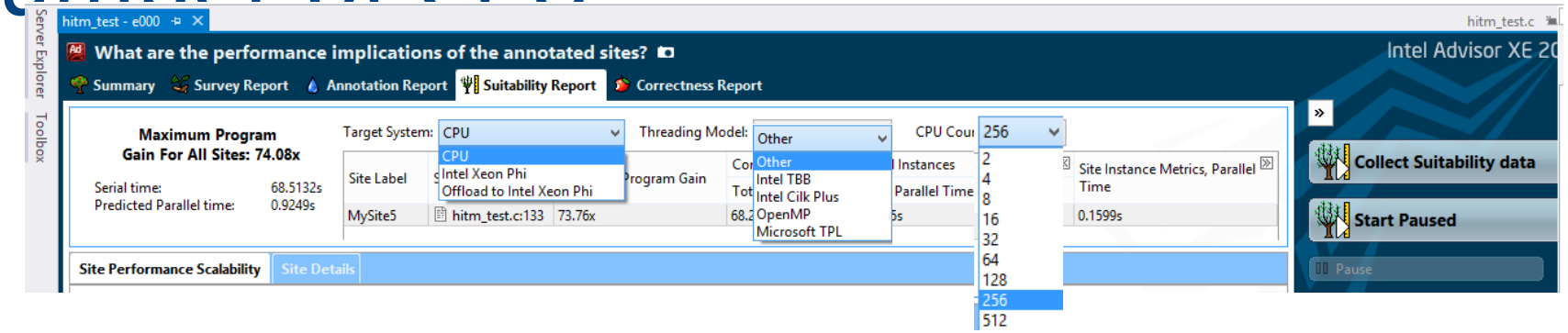
- All of these regions may be nested
- You may create more than one SITE
- Just macros, so work with any C/C++ compiler

```
#include "advisor-annotate.h"
...
void solve() {
int * queens = new int[size];
//array representing queens placed on a chess board...

ANNOTATE_SITE_BEGIN(solve);

for(int i=0; i<size; i++)
{ // try all positions in first row

ANNOTATE_ITERATION_TASK(setQueen);
setQueen(queens, 0, i);
}

ANNOTATE_SITE_END();
...
}
```

Suitability report



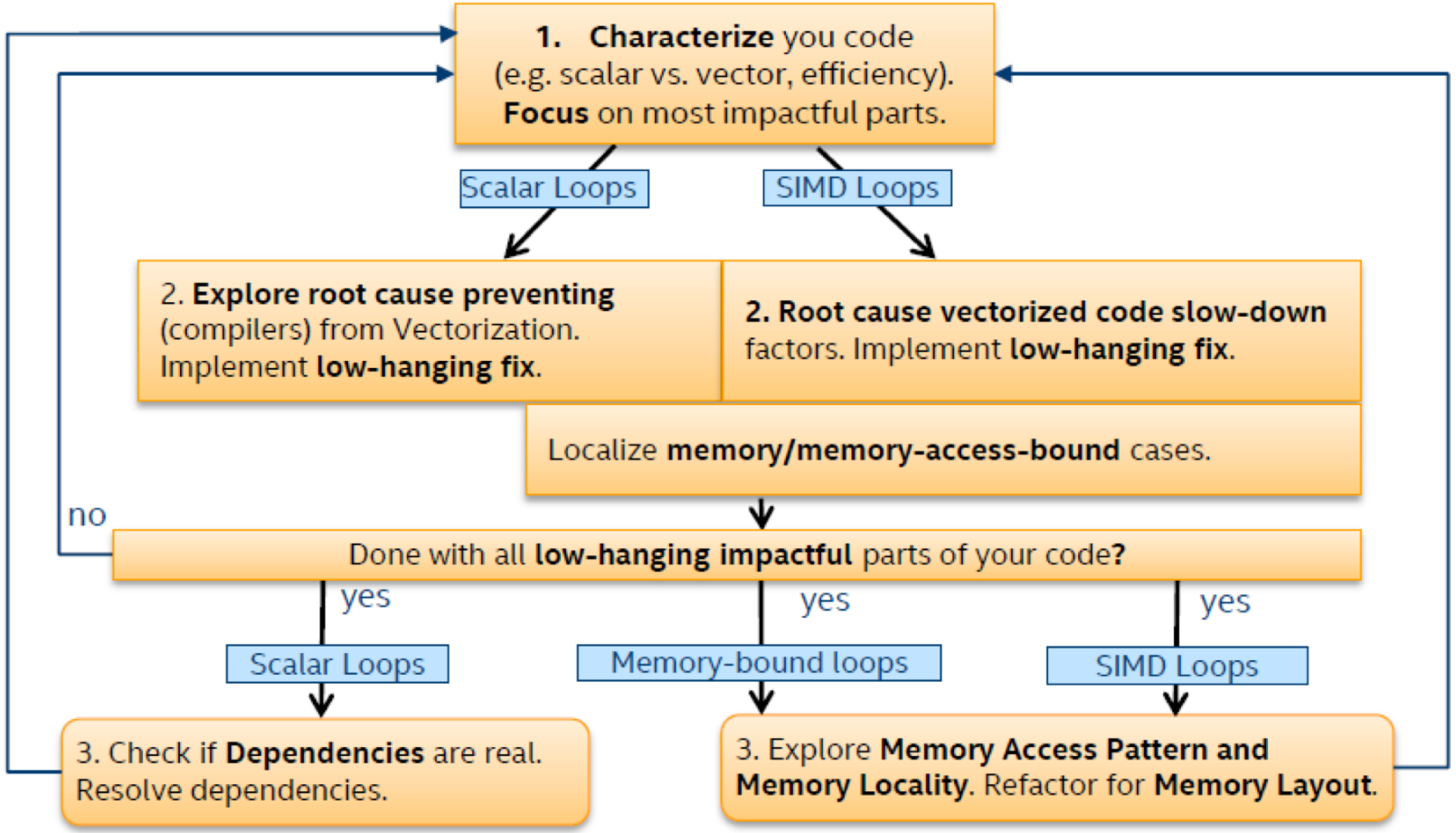
Adjustable: Target architecture, threading models and number of CPU



- To set up data collection determine the target architecture, threading model and number of CPU's
- Collect the Scalability data, and determine how it differs between the architectures and threading models.

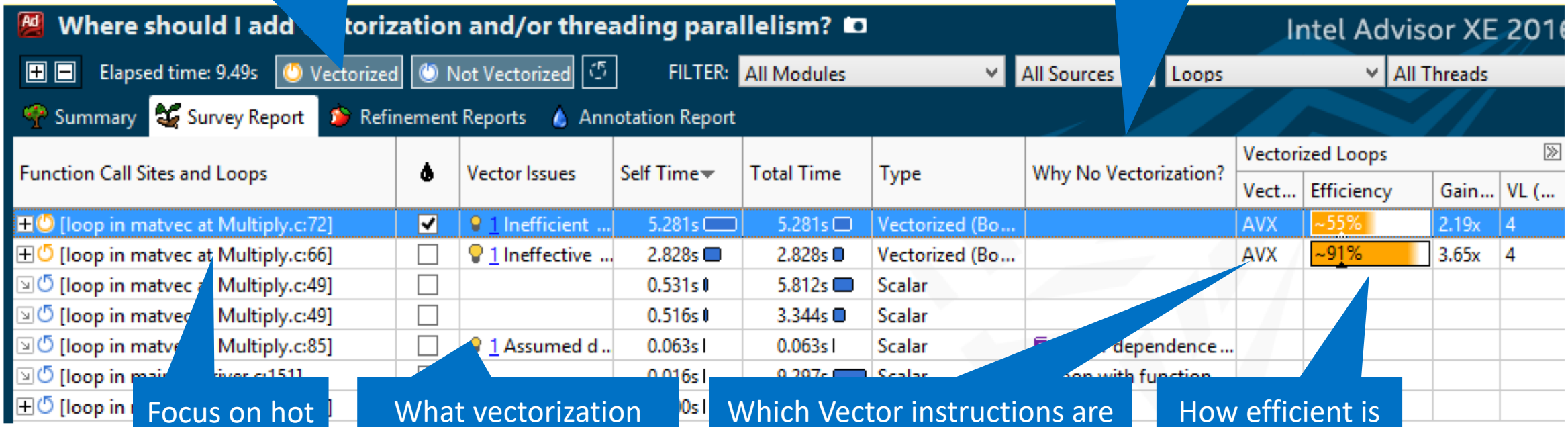
Intel® Advisor - Vectorization

Recommended methodology



The Right Data At Your Fingertips

Get all the data you need for high impact vectorization

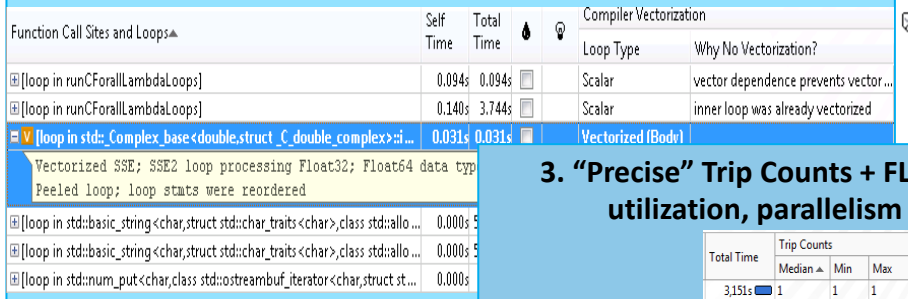


Get Fast Code Fast!

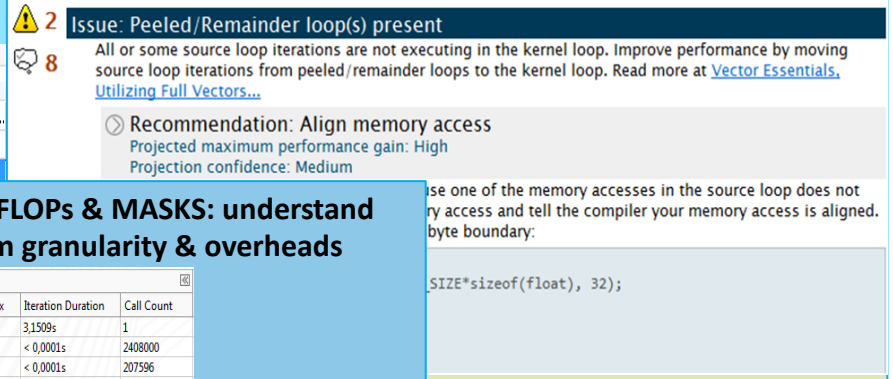
5 Steps to Efficient Vectorization - Vector Advisor

(part of Intel® Advisor, Parallel Studio, Cluster Studio **2016**)

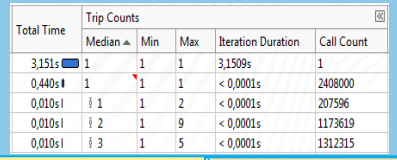
1. Compiler diagnostics + Performance Data + SIMD efficiency information



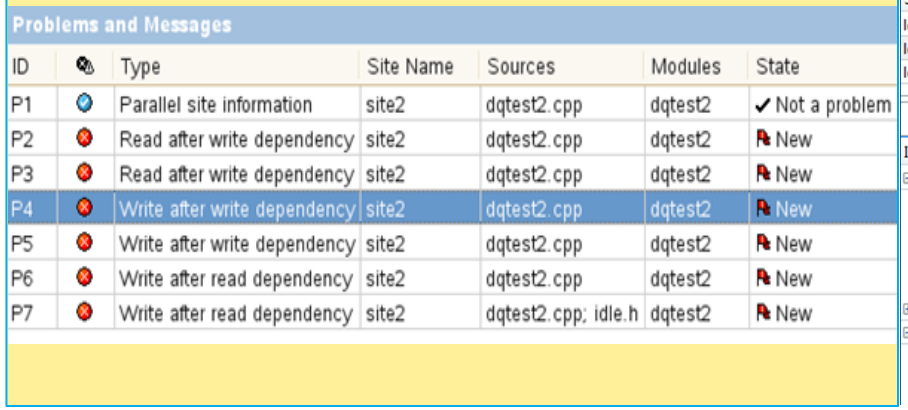
2. Guidance: detect problem and recommend how to fix it



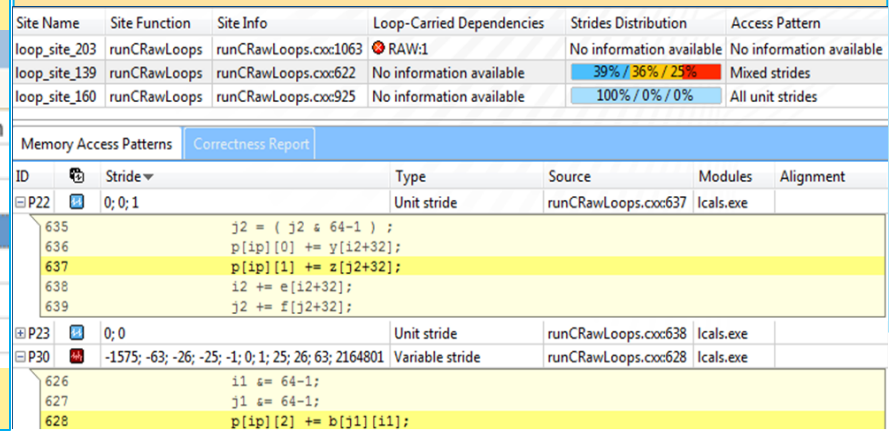
3. "Precise" Trip Counts + FLOPs & MASKS: understand utilization, parallelism granularity & overheads



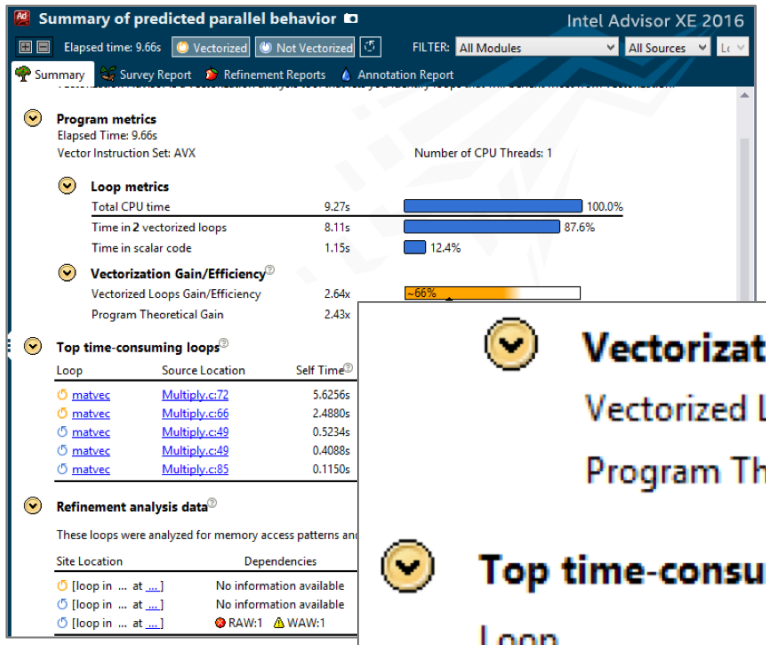
4. Loop-Carried Dependency Analysis

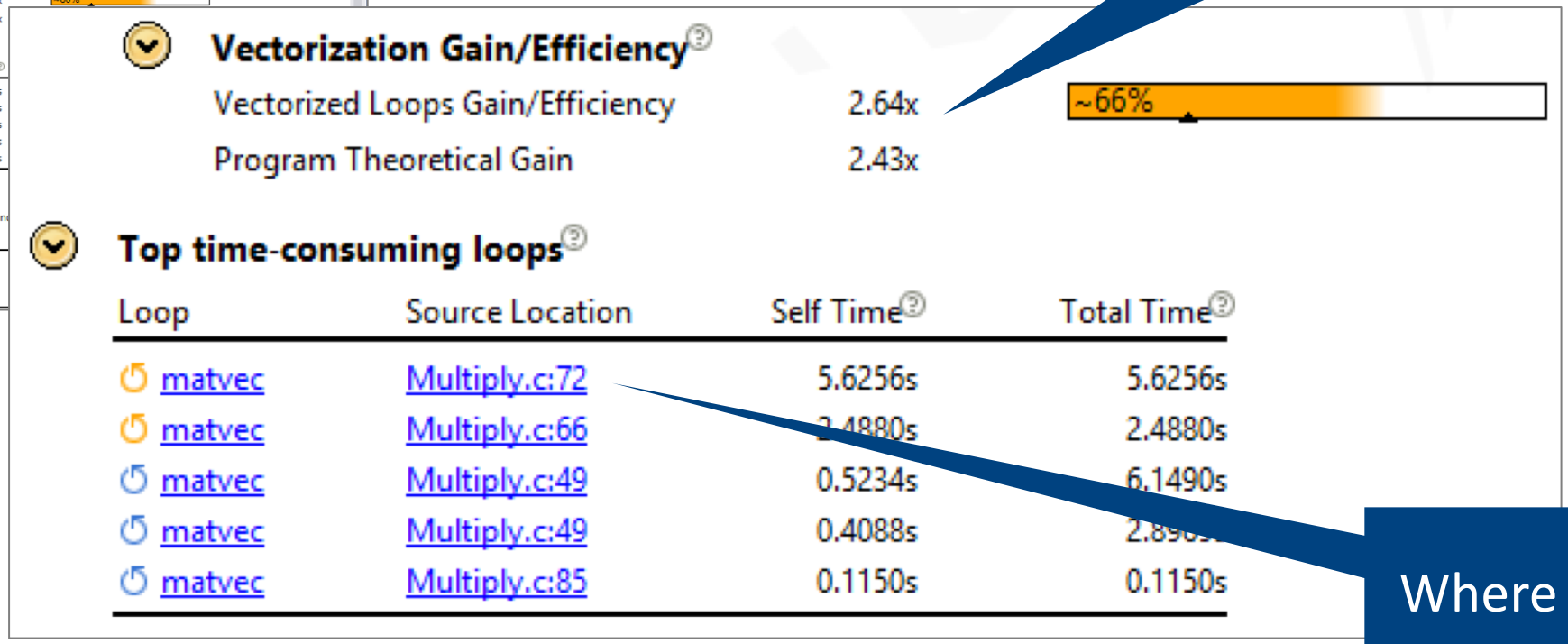


5. Memory Access Patterns Analysis



Summary View: Plan Your Next Steps



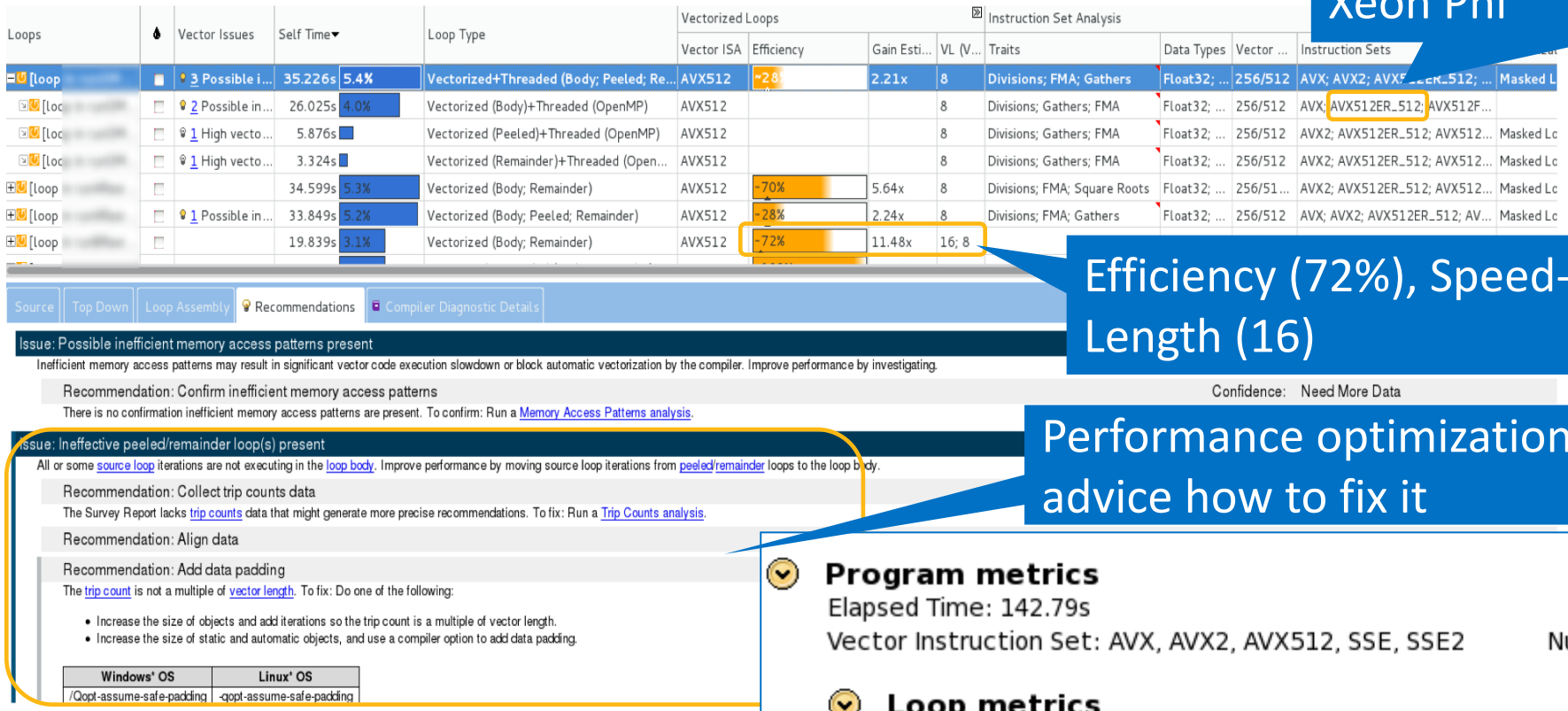


What can I expect to gain?

Where do I start?

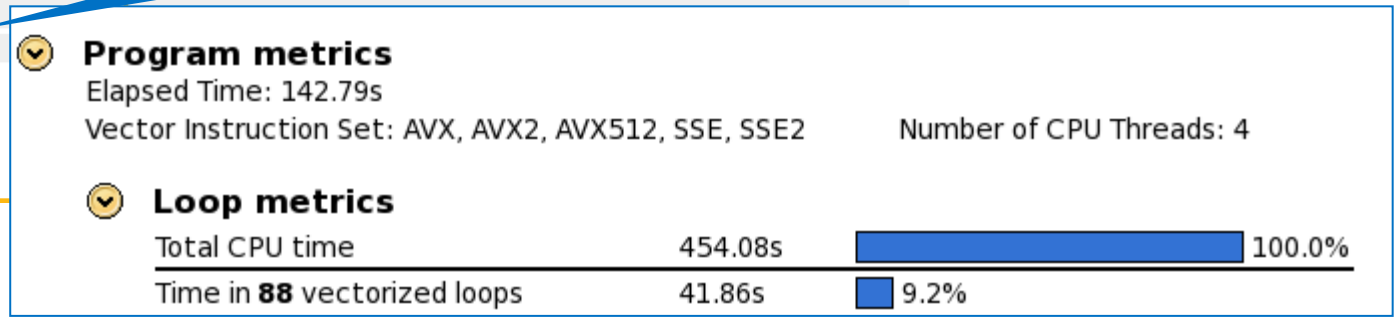
Vectorization Advisor runs on and optimizes for Intel® Xeon Phi™ architecture

AVX-512 ERI – specific to Intel® Xeon Phi



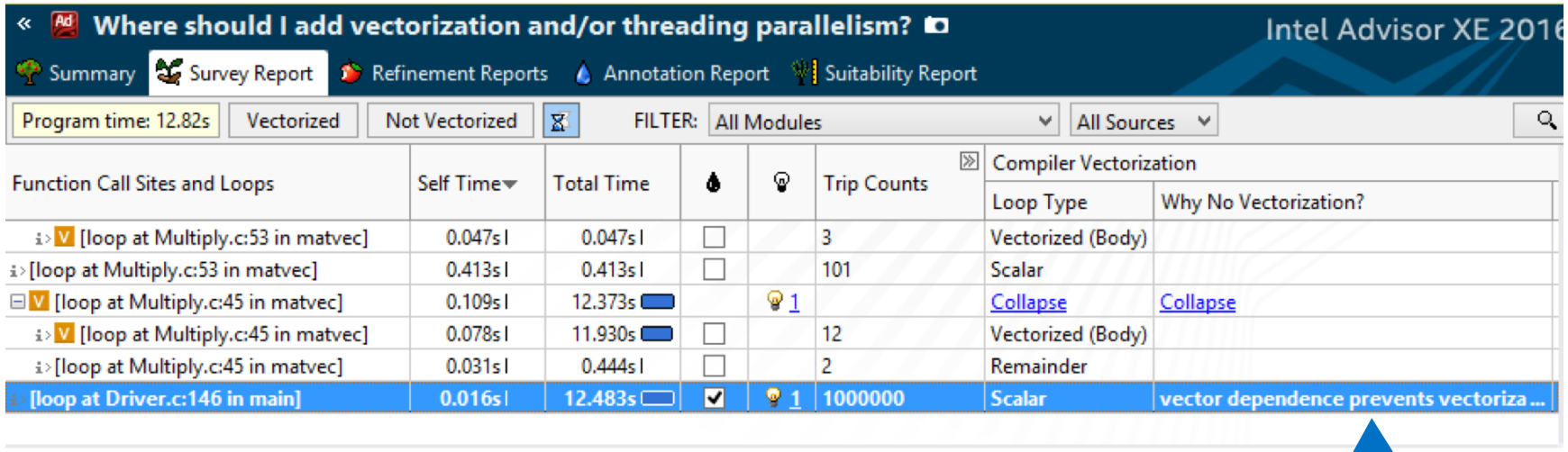
Efficiency (72%), Speed-up (11.5x), Vector Length (16)

Performance optimization problem and advice how to fix it



Check if It Is Safe to Vectorize

Loop-Carried Dependencies Analysis Verifies Correctness



2.1 Check Dependencies

Identify and explore loop-carried dependencies for marked loops. Fix the reported problems.

Collect

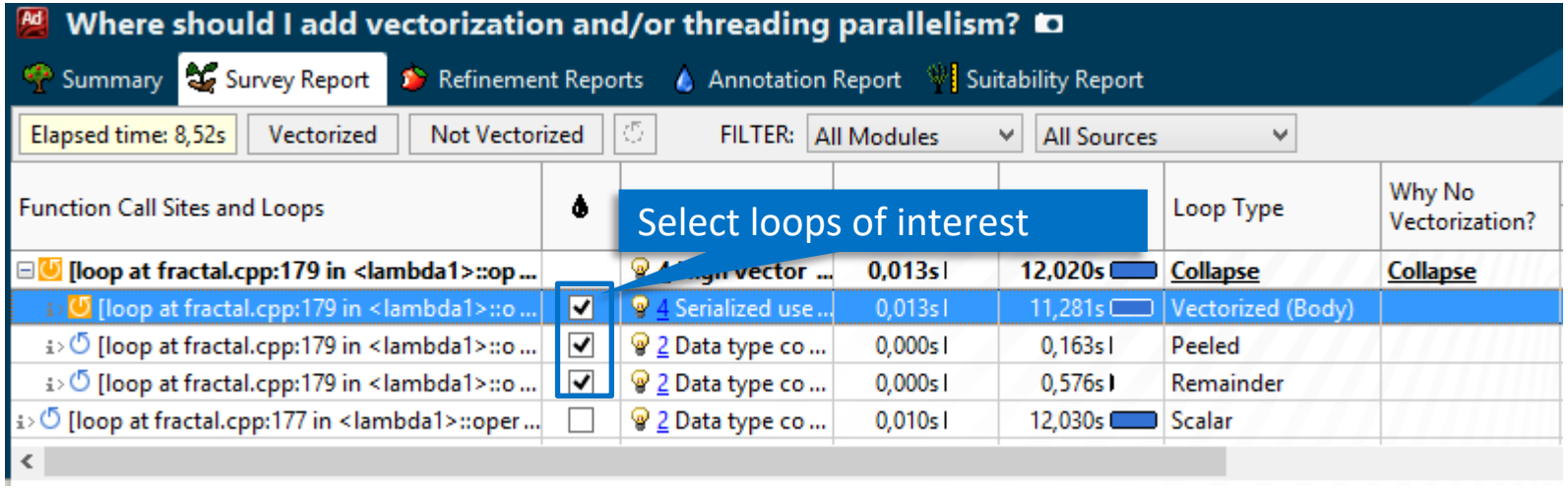
Command Line

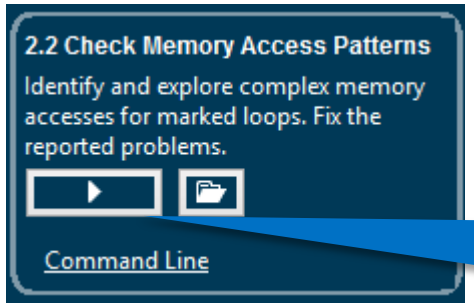
Select loop for Dependency Analysis and press play!

Vector Dependence prevents Vectorization!

Improve Vectorization

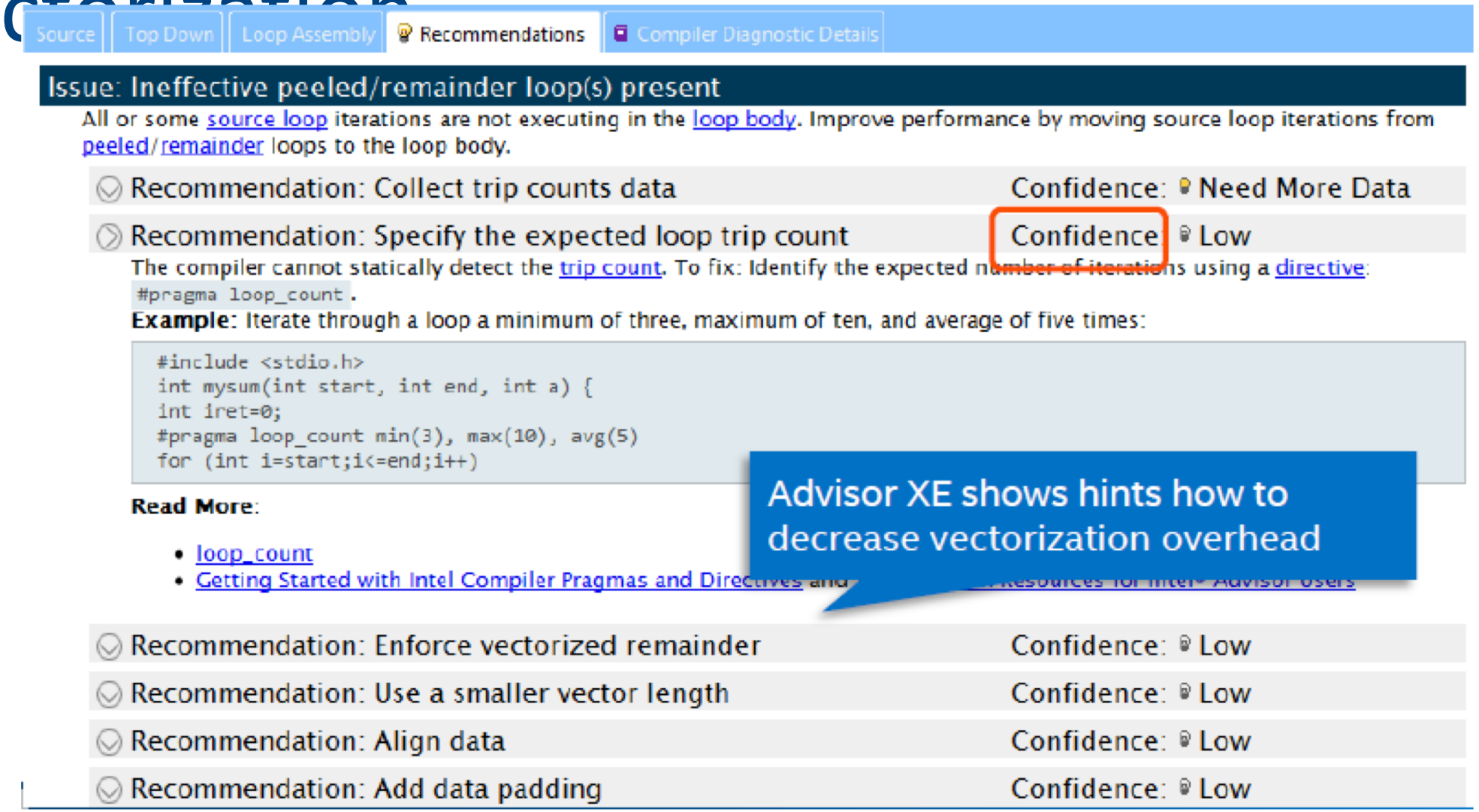
Memory Access Pattern Analysis





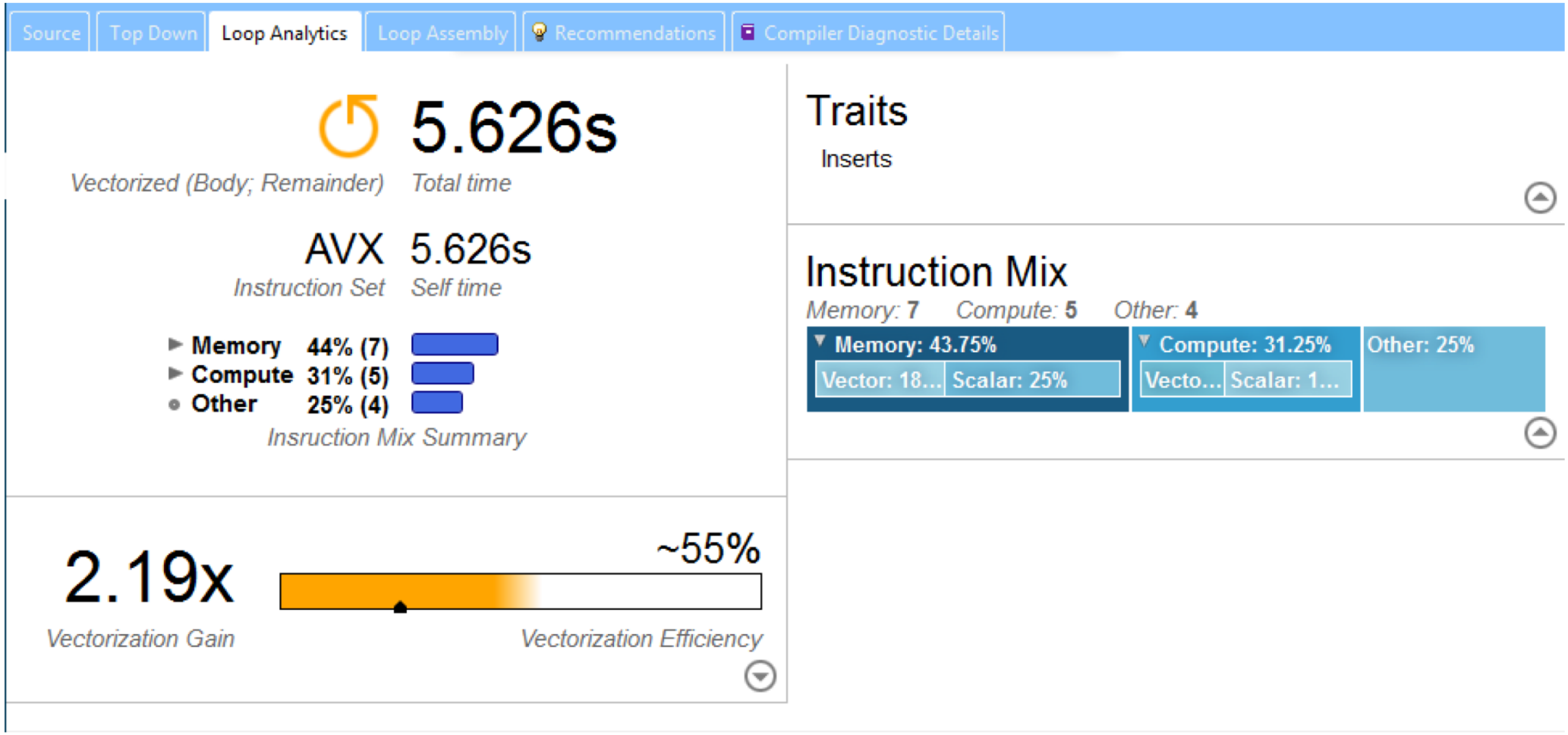
Run Memory Access Patterns analysis, just to check how memory is used in the loop and the called function

Get specific advice for Improving Vectorization

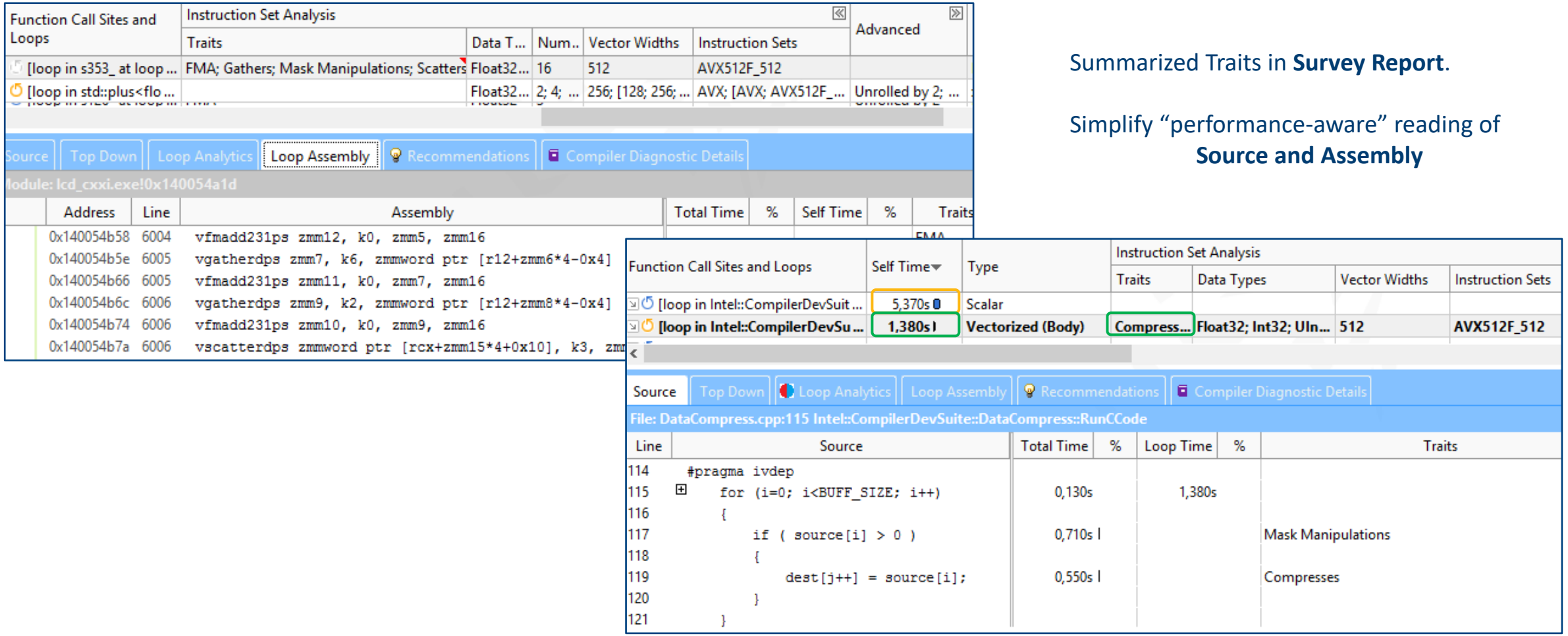


Loop Analytics

Get detailed information about your loops



Highlight “impactful” AVX-512 instructions.
Survey Static Analysis - AVX-512 ***“Traits”***



Summarized Traits in **Survey Report**.

Simplify “performance-aware” reading of **Source and Assembly**

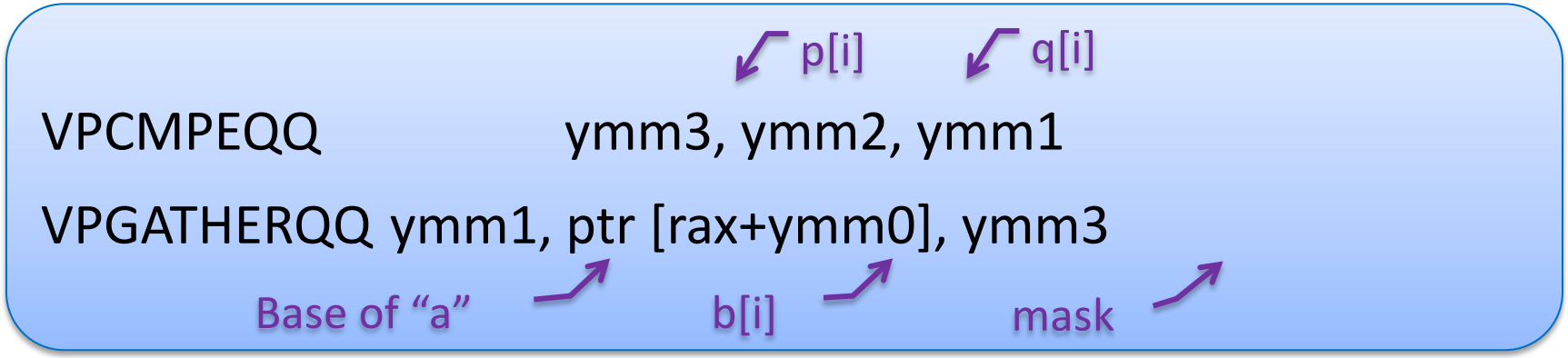
Gather/Scatter analysis is very important for AVX-512

- AVX512 Gather/Scatter in wider use than on previous instruction sets
- Many more applications can now be vectorized
- Gives good average performance but far from optimal
- Much greater need for Gather/Scatter profiling
- With Intel® Advisor you get both dynamic and static gather/scatter information

Gather Instructions

- c[i] = a[b[i]] //indirect reference

if (p[i] == q[i]) c[i] = a[b[i]] // also masked

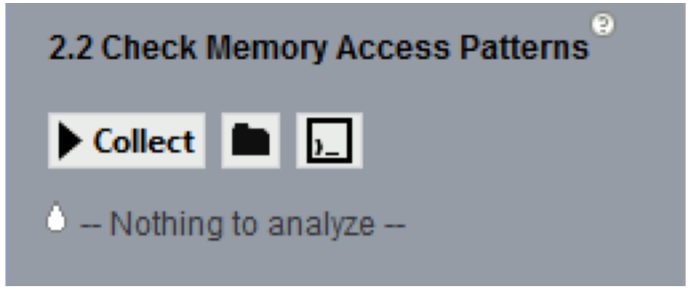


Fundamental building block for sparse or indirect memory accesses, easing vectorization

Irregular access patterns decreases performance!

Gather profiling

- Run Memory Access Pattern Analysis (MAP)



0%:percentage of memory instructions with unit stride or stride 0 accesses

Unit stride (stride 1) = Instruction accesses memory that consistently changes by one element from iteration to iteration

Uniform stride (stride 0) = Instruction accesses the same memory from iteration to iteration

50%: percentage of memory instructions with fixed or constant non-unit stride accesses

Constant stride (stride N) = Instruction accesses memory that consistently changes by N elements from iteration to iteration
Example: for the double floating point type, stride 4 means the memory address accessed by this instruction increased by 32 bytes, (4*sizeof(double)) with each iteration

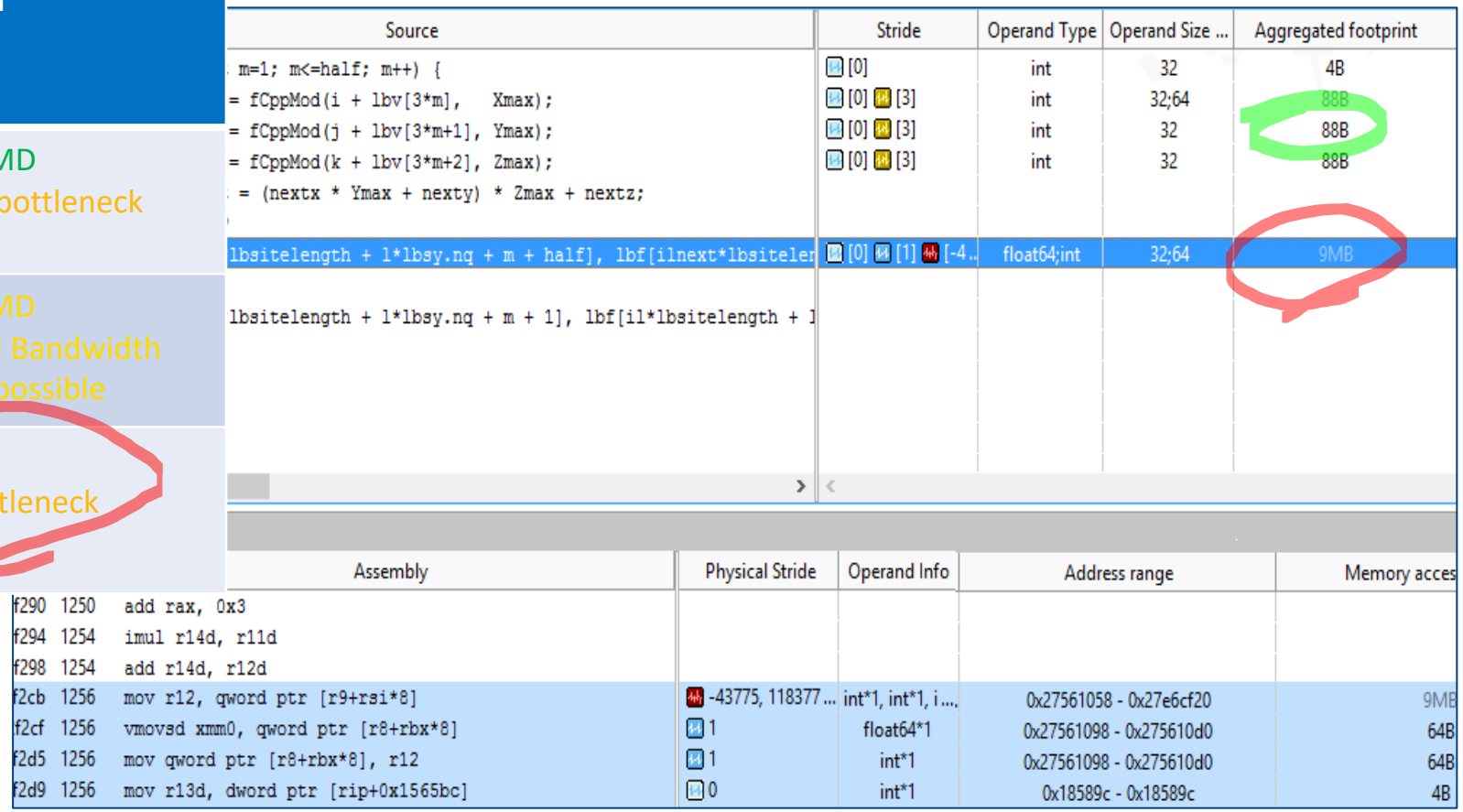
50%: percentage of memory instructions with irregular (variable or random) stride accesses

Irregular stride = Instruction accesses memory addresses that change by an unpredictable number of elements from iteration to iteration
Typically observed for indirect indexed array accesses, for example, a[index[i]]

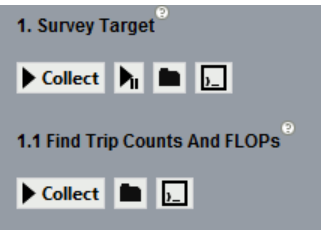
- gather (irregular) accesses, detected for v(p)gather* instructions on AVX2 Instruction Set Architecture

Am I bound by VPU/CPU or by Memory?:
Advisor Memory Access Pattern and Footprint

Footprint	Small enough	Big enough
Access Pattern		
Unit Stride	Effective SIMD No Latency and BW bottlenecks	Effective SIMD Bandwidth bottleneck
Const stride	Medium SIMD Latency bottleneck possible	Medium SIMD Latency and Bandwidth bottleneck possible
Irregular Access, Gather/Scatter	Bad SIMD Latency bottleneck possible	Bad SIMD Latency bottleneck



AVX-512 FLOPS and Mask profiler



FLOPs, Masks, Trip Counts

Median	GFLOPs/s	Arithmetic Intensity	Mask Utiliz...	GBytes/s	GFLOP
19	2,456	0.125		19.6498	3.94488
4; 3	2,351	0.125	63,29%	18.8111	0.36693
19	2,136	0.0795455		26.8513	2.50206
19	1,910	0.0681818		28.011	1.07231
3	1,774	0.0833333		21.2898	0.11287
4	1,192	0.0666667		17.8726	0.1505
19	0,911	0.0681818		13.3635	0.0285

Low mask population -> low performance (in spite of "high SIMD efficiency")

Function Call Sites and Loops	Vector Issues	Self Time	Type	FLOPS			Vect... ps			Instruction Set Analysis	
				GFLOPS	AI	Mask Utilizatio	...ector ISA	Efficiency	Gain Esti...	VL	Traits
[loop in s2711 at loops90.f:16 ...		0,010s	Vectorized (Remainder)	0,800	0,1000		AVX512	~56%	8.98x	16	FMA
[loop in s252 at loops90.f:1172]	2 Ineffective peeled/r...	0,171s	Vectorized Versions	1,684	0,0968	100%	AVX512	~41%	13.05x	16;	Blends; Divisions; Extracts;
[loop in s116 at loops90.f:257]	1 Ineffective peeled/r...	0,100s	Vectorized (Body; Remainder)	4,951	0,0833	88%	AVX512	~79%	12.68x	16	Unpacks
[loop in s174 at loops90.f:765]	1 Ineffective peeled/r...	0,080s	Vectorized (Body; Remainder)	1,251	0,0833	78%	AVX512	~41%	13.05x	16;	Unpacks
[loop in s173 at loops90.f:7 ...	1 Ineffective peeled ...	0,090s	Vectorized (Body; Remainder)	1,111	0,0833	78%	AVX512	~41%	13.05x	16;	Unpacks
[loop in s152 at loops90.f:624]	1 Ineffective peeled/r...	0,010s	Vectorized (Body; Remainder)	10,001	0,0833	89%	AVX512	~37%	11.70x	16;	FMA
[loop in s121 at loops90.f:324]	1 Ineffective peeled/r...	0,020s	Vectorized (Body; Remainder)	4,950	0,0833	88%	AVX512	~36%	11.47x	16;	Unpacks
[loop in s151s at loops90.f:609]	1 Ineffective peeled/r...	0,020s	Vectorized (Body; Remainder)	4,950	0,0833	88%	AVX512	~36%	11.47x	16;	Unpacks
[loop in s131 at loops90.f:521]		0,020s	Vectorized (Body)	4,808	0,0833	100%	AVX512	~36%	11.47x	32	
[loop in s119 at loops90.f:302]	1 Ineffective peeled/r...	0,040s	Vectorized (Body; Remainder)	2,450	0,0833	88%	AVX512	~35%	11.25x	16;	Unpacks
[loop				1,649	0,0833	89%	AVX512	~35%	11.25x	16;	Unpacks
[loop				1,400	0,0833	88%	AVX512	~35%	11.25x	16;	Unpacks
[loop				0,797	0,0833		AVX512	~22%	3.55x	16	FMA
[loop				0,840	0,0833	78%	AVX512	~19%	2.97x	16	2-Source Permutes; Gather

Efficiency, FLOPS and Arithmetic Intensity correlation (more memory bound -> lower SIMD performance)

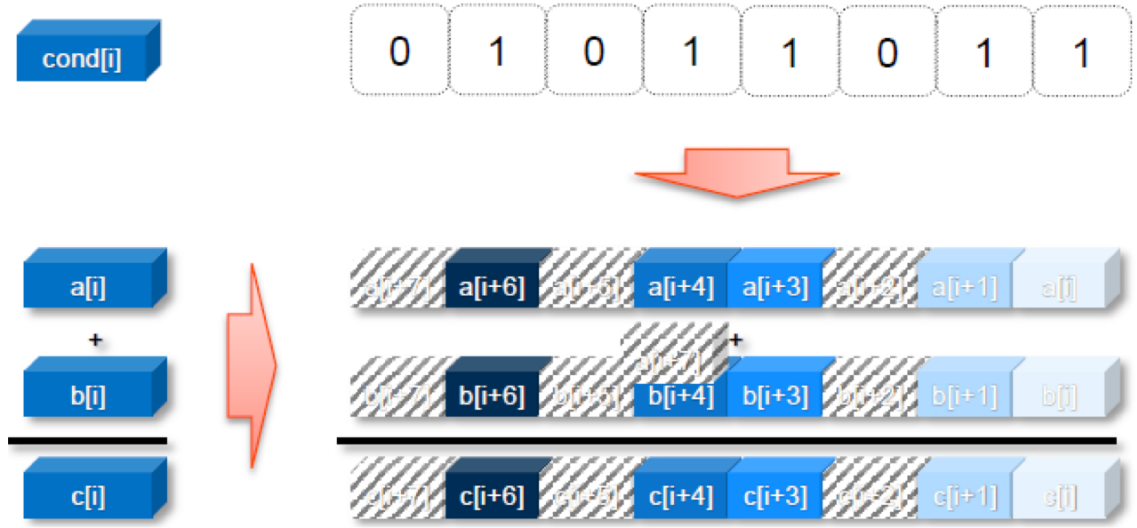
Why is Mask Utilization important?

3 elements suppressed

SIMD Utilization = 5/8 (62.5%)

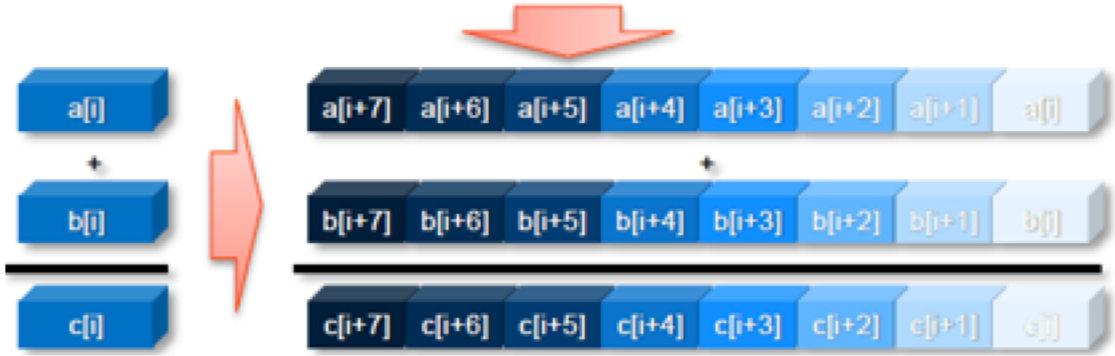
Not utilizing full vectors!!

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



Fully utilized!

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



Advisor Roofline: under the hood

Roofline application profile:

Axis Y: **FLOP/S** = **#FLOP** (mask aware) / **#Seconds**

Axis X: **AI** = **#FLOP** / **#Bytes**

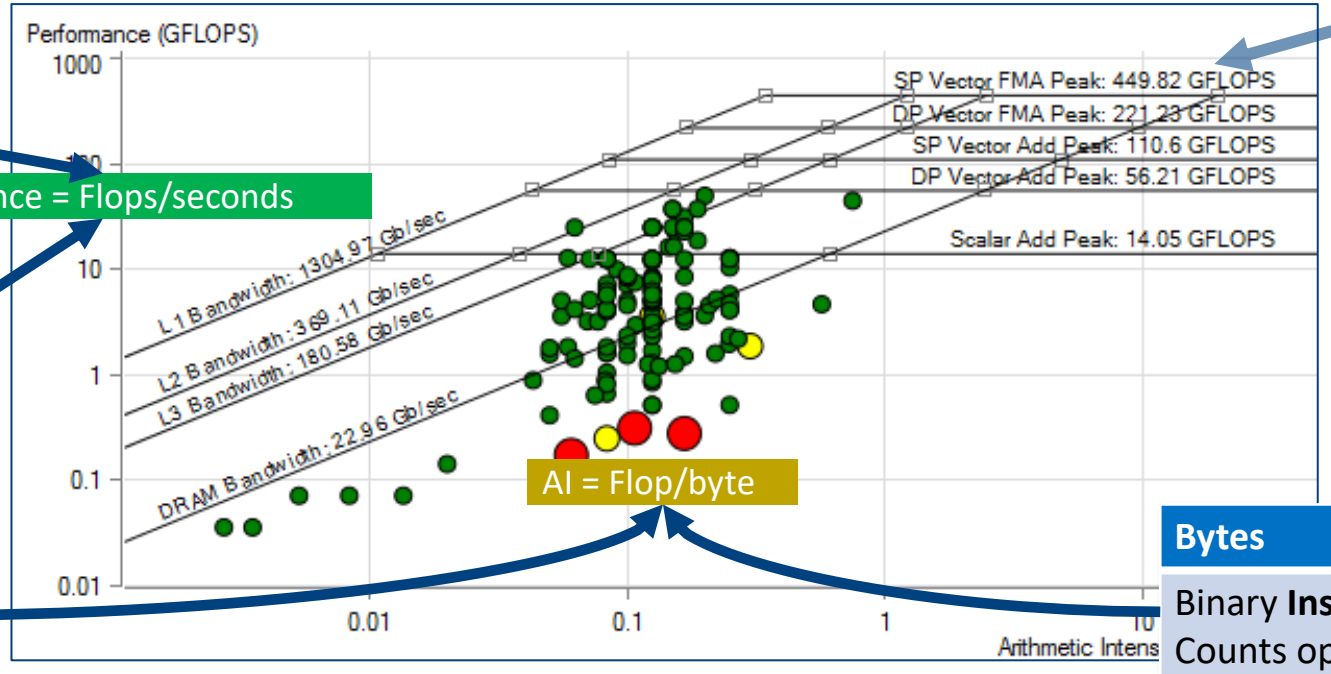
Seconds

User-mode **sampling**

Root access not needed

Roofs

Microbenchmarks
Actual peak for the current configuration



Performance = Flops/seconds

AI = Flop/byte

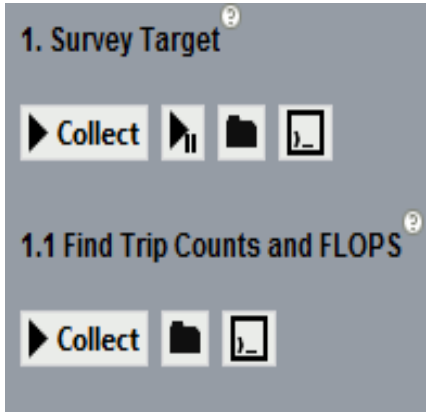
#FLOP

Binary **Instrumentation**
Does not rely on CPU counters

Bytes

Binary **Instrumentation**
Counts operands size (not cachelines)

Getting Roofline in Advisor



FLOP/S **= #FLOP/Seconds**	**Seconds**	**#FLOP Count** **- Mask Utilization** **- #Bytes**
Step 1: Survey - Non intrusive. *Representative* - Output: Seconds (+much more)	✓	
Step 2: FLOPS - Precise, instrumentation based - Physically count Num-Instructions - Output: #FLOP, #Bytes		✓

Cache-Aware Roofline

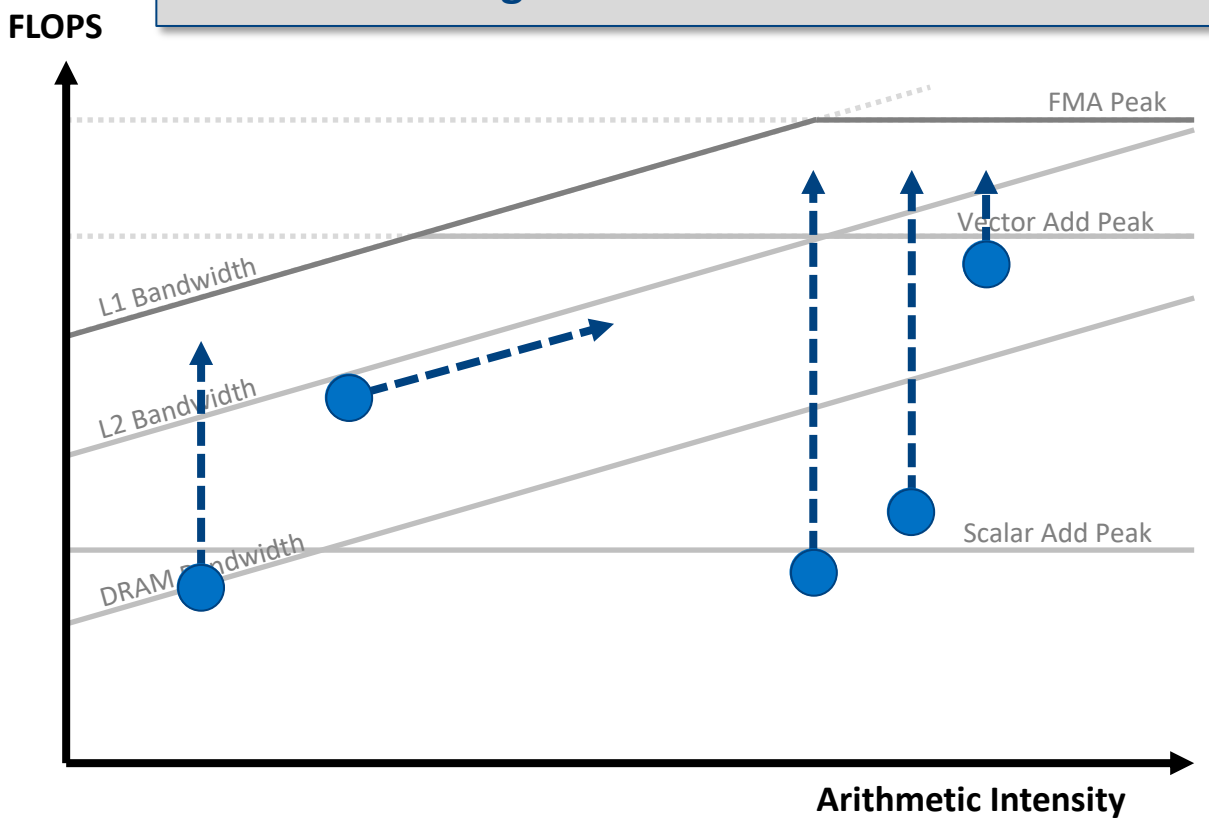
Next Steps

If under or near a memory roof…

- Try a MAP analysis. Make any appropriate **cache optimizations**.
- If cache optimization is impossible, try **reworking the algorithm to have a higher AI.**

If Under the Vector Add Peak

Check “Traits” in the Survey to see if FMAs are used. If not, try altering your code or compiler flags to **induce FMA usage.**



If just above the Scalar Add Peak

Check **vectorization efficiency** in the Survey. Follow the recommendations to improve it if it’s low.

If under the Scalar Add Peak…

Check the Survey Report to see if the loop vectorized. If not, try to **get it to vectorize** if possible. This may involve running Dependencies to see if it’s safe to force it.

Typical Vectorization Optimization Workflow

- There is no need to recompile or relink the application, but the use of -g is recommended.

1. Collect survey and tripcounts data
   - Investigate application place within roofline model
   - Determine vectorization efficiency and opportunities for improvement
2. Collect memory access pattern data
   - Determine data structure optimization needs
3. Collect dependencies
   - Differentiate between real and assumed issues blocking vectorization

Nbody demonstration

Nbody gravity simulation

https://github.com/fbaru-dev/nbody-demo (Dr. Fabio Baruffa)

- Let's consider a distribution of point masses m_1,…,m_n located at r_1,…,r_n.
- We want to calculate the position of the particles after a certain time interval using the Newton law of gravity.

```
struct Particle
{
 public:
   Particle() { init();}
   void init()
   {
     pos[0] = 0.; pos[1] = 0.; pos[2] = 0.;
     vel[0] = 0.; vel[1] = 0.; vel[2] = 0.;
     acc[0] = 0.; acc[1] = 0.; acc[2] = 0.;
     mass   = 0.;
   }
   real_type pos[3];
   real_type vel[3];
   real_type acc[3];
   real_type mass;
};
```

```
for (i = 0; i < n; i++){          // update acceleration
   for (j = 0; j < n; j++){
        real_type distance, dx, dy, dz;
        real_type distanceSqr = 0.0;
        real_type distanceInv = 0.0;

        dx = particles[j].pos[0] - particles[i].pos[0];
     …

        distanceSqr = dx*dx + dy*dy + dz*dz + softeningSquared;
     distanceInv = 1.0 / sqrt(distanceSqr);

        particles[i].acc[0] += dx * G * particles[j].mass *
                               distanceInv * distanceInv * distanceInv;
        particles[i].acc[1] += …
        particles[i].acc[2] += …
```

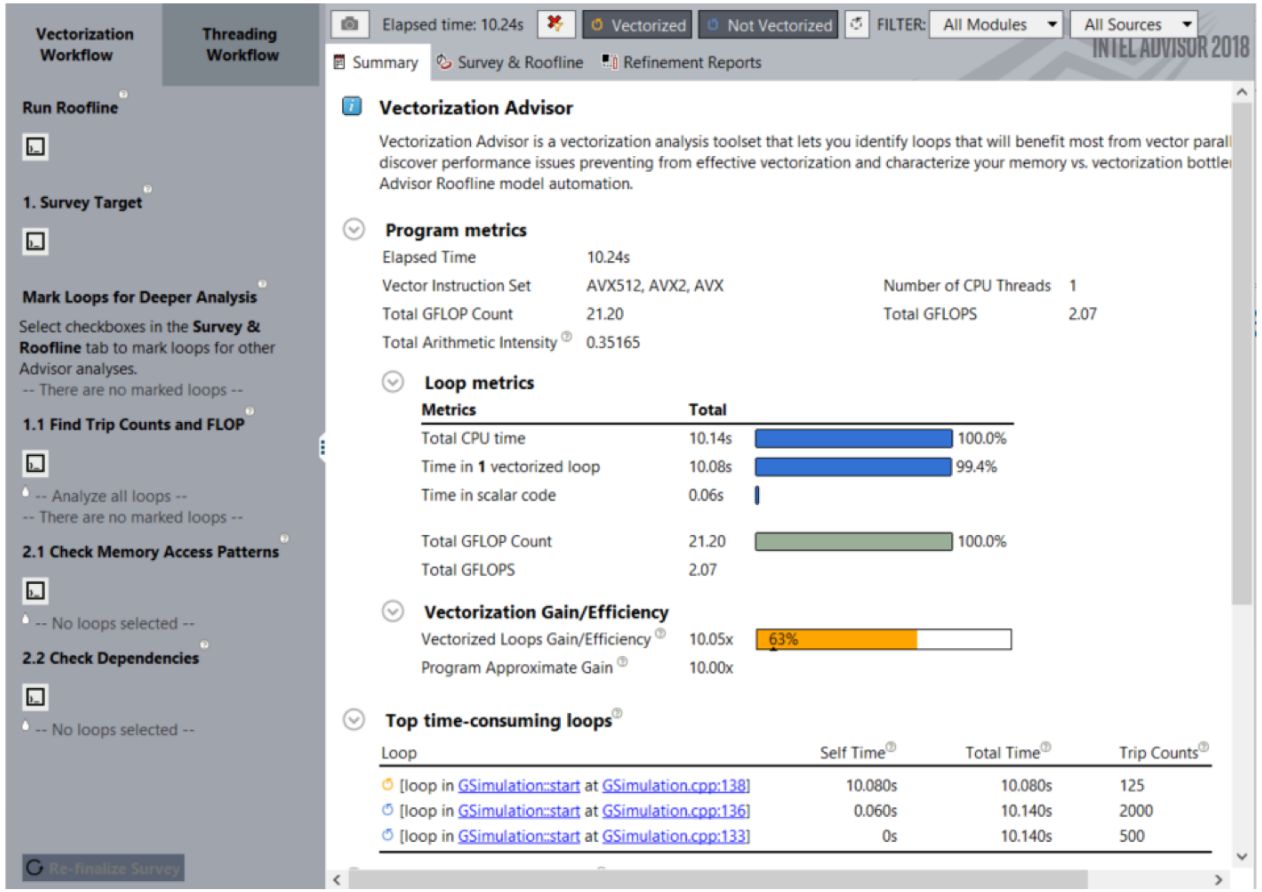
Collect Roofline Data

- Starting with version 2 of the code we collect both survey and tripcounts data:

```
mpirun -n 1 -N 1 advixe-cl --collect survey --project-dir ./adv_res --search-dir src:=./ \
                           --search-dir bin:=./ -- ./nbody.x
mpirun -n 1 -N 1 advixe-cl --collect tripcounts -flops-and-masks --project-dir ./adv_res \
                           --search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
mpirun –n 1 –N 1 advixe-cl –collect roofline (available starting 2018 U1)
```

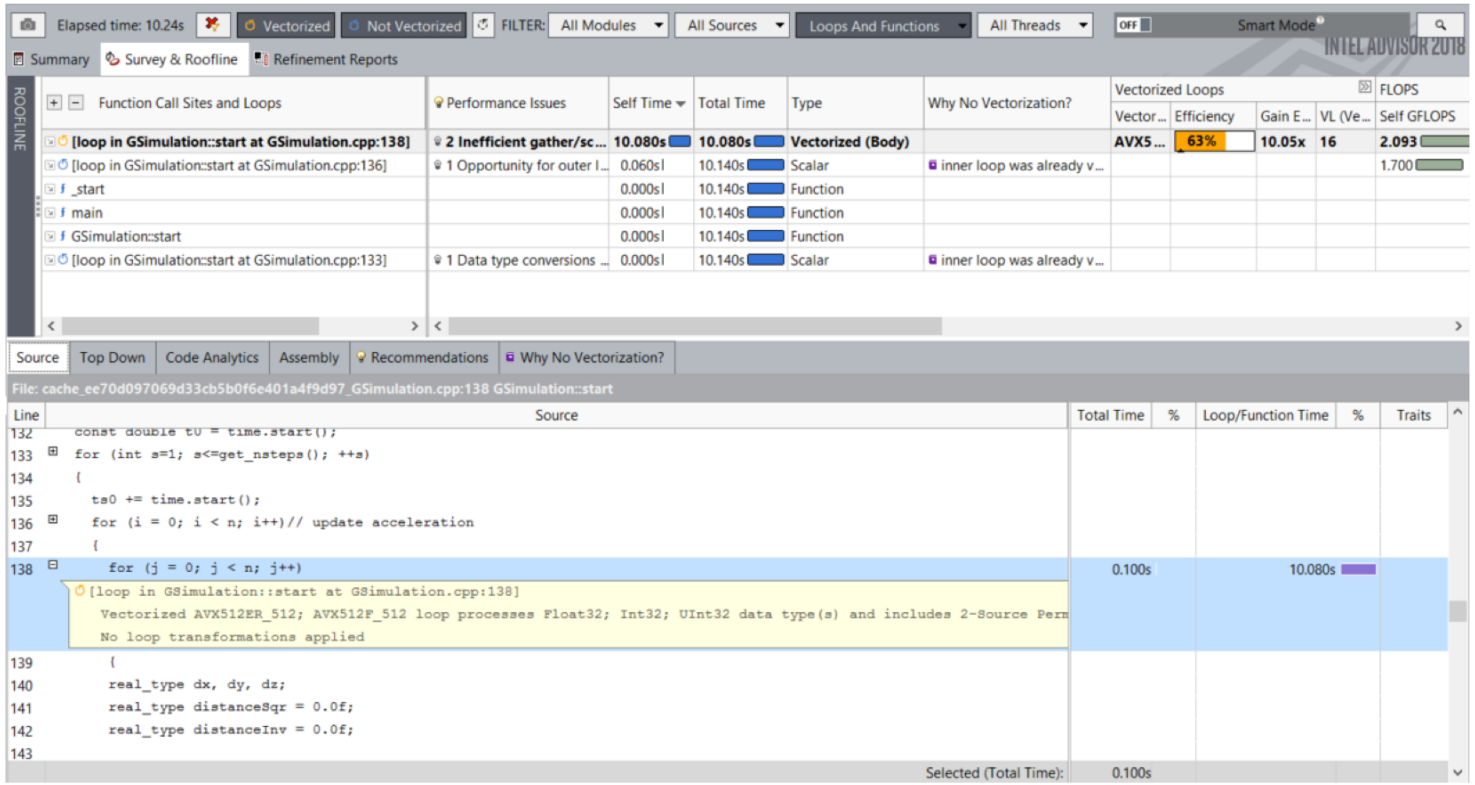
- If finalization is too slow on compute add -no-auto-finalize to collection line.

Summary Report



- GUI left panel provides access to further tests
- Summary provides overall performance characteristics
- Lists instruction set(s) used
- Top time consuming loops are listed individually
- Loops are annotated as vectorized and non-vectorized
- Vectorization efficiency is based on used ISA, in this case Intel® Advanced Vector Extensions 512 (AVX512)

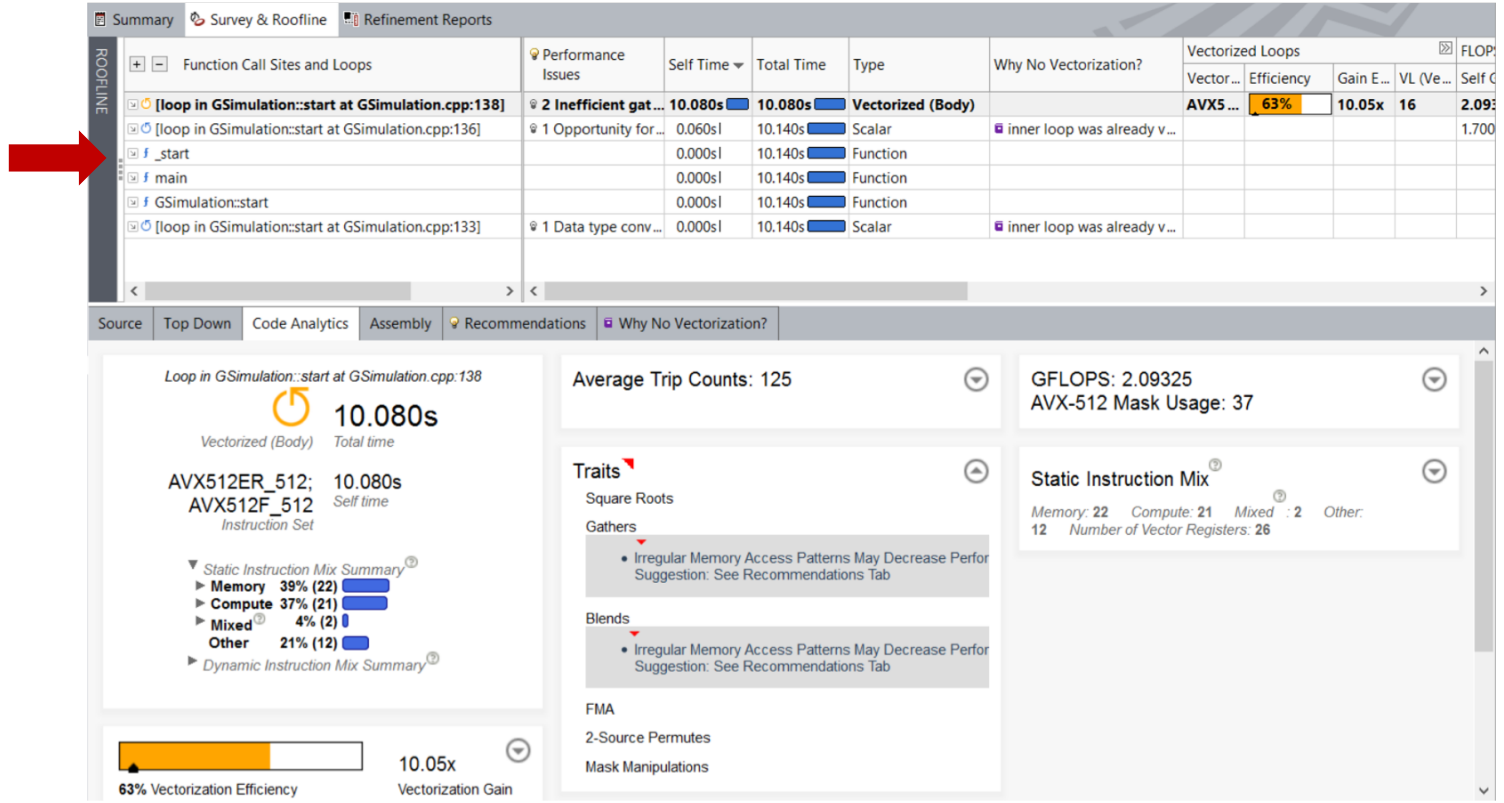
Survey Report (Source)



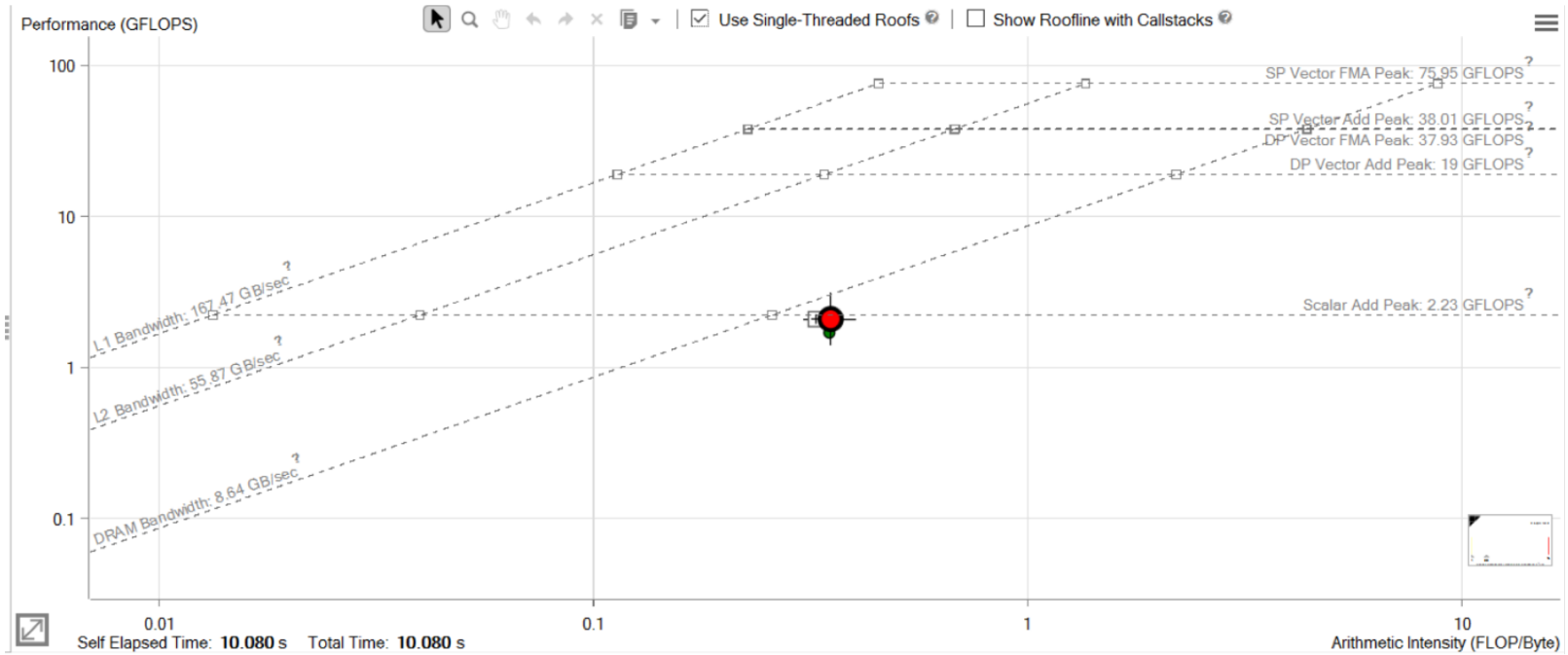
- Inline information regarding loop characteristics
- ISA used
- Types processed
- Compiler transformations applied
- Vector length used
- …

Survey Report (Code Analytics)

- Detailed loop information
  - Instruction mix
  - ISA used, including subgroups
  - Loop traits
    - FMA
    - Square root
    - Gathers / Blends point to memory issues and vector inefficiencies



CARM Analysis

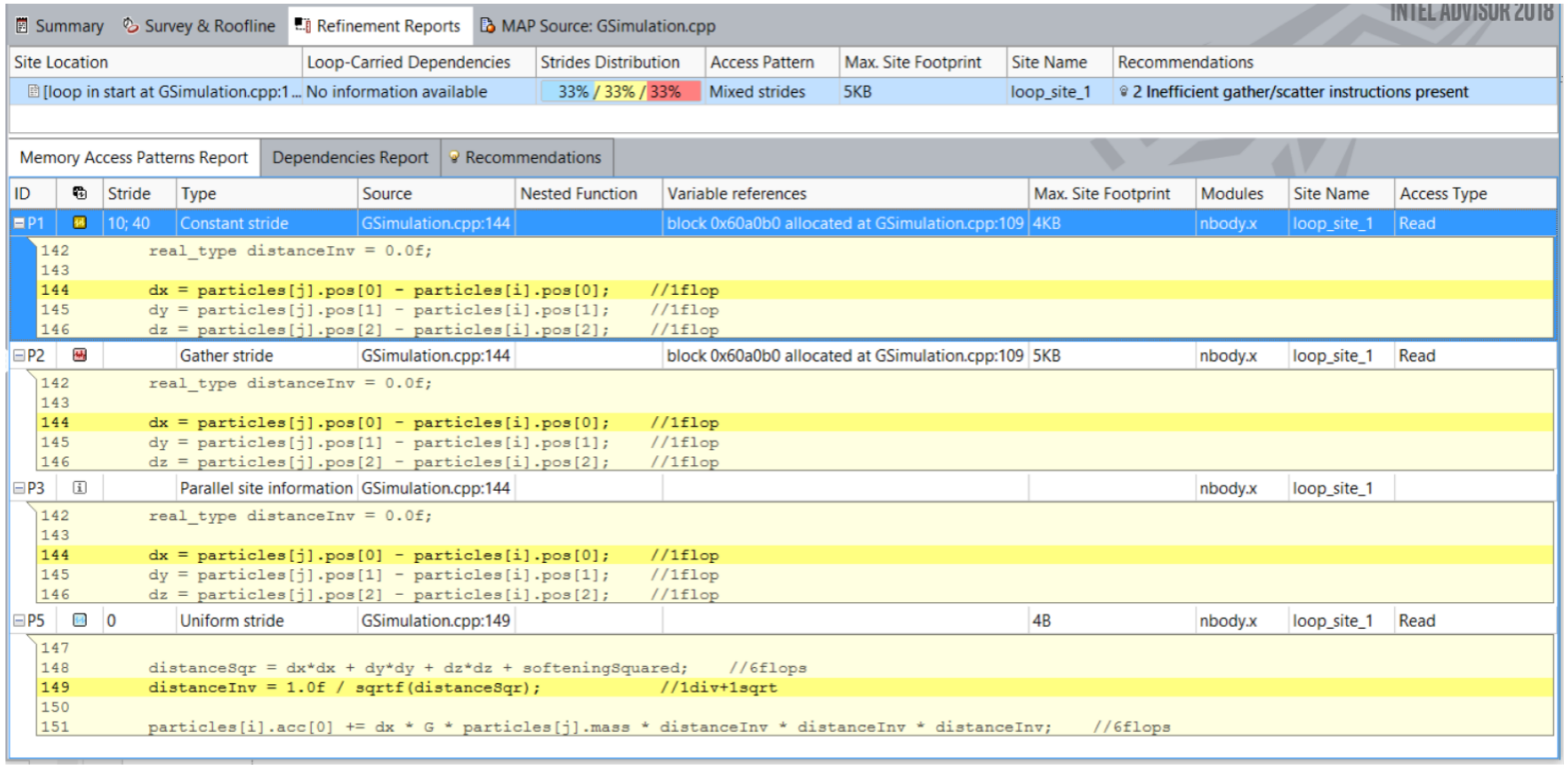


- Using single threaded roof
- Code vectorized, but performance on par with scalar add peak?
- Irregular memory access patterns force gather operations.
- Overhead of setting up vector operations reduces efficiency.

Next step is clear: perform a Memory Access Pattern analysis

Memory Access Pattern Analysis (Refinement)

```
mpirun -n 1 -N 1 advixe-cl --collect map -mark-up-list=1 --project-dir
./adv_res --search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```



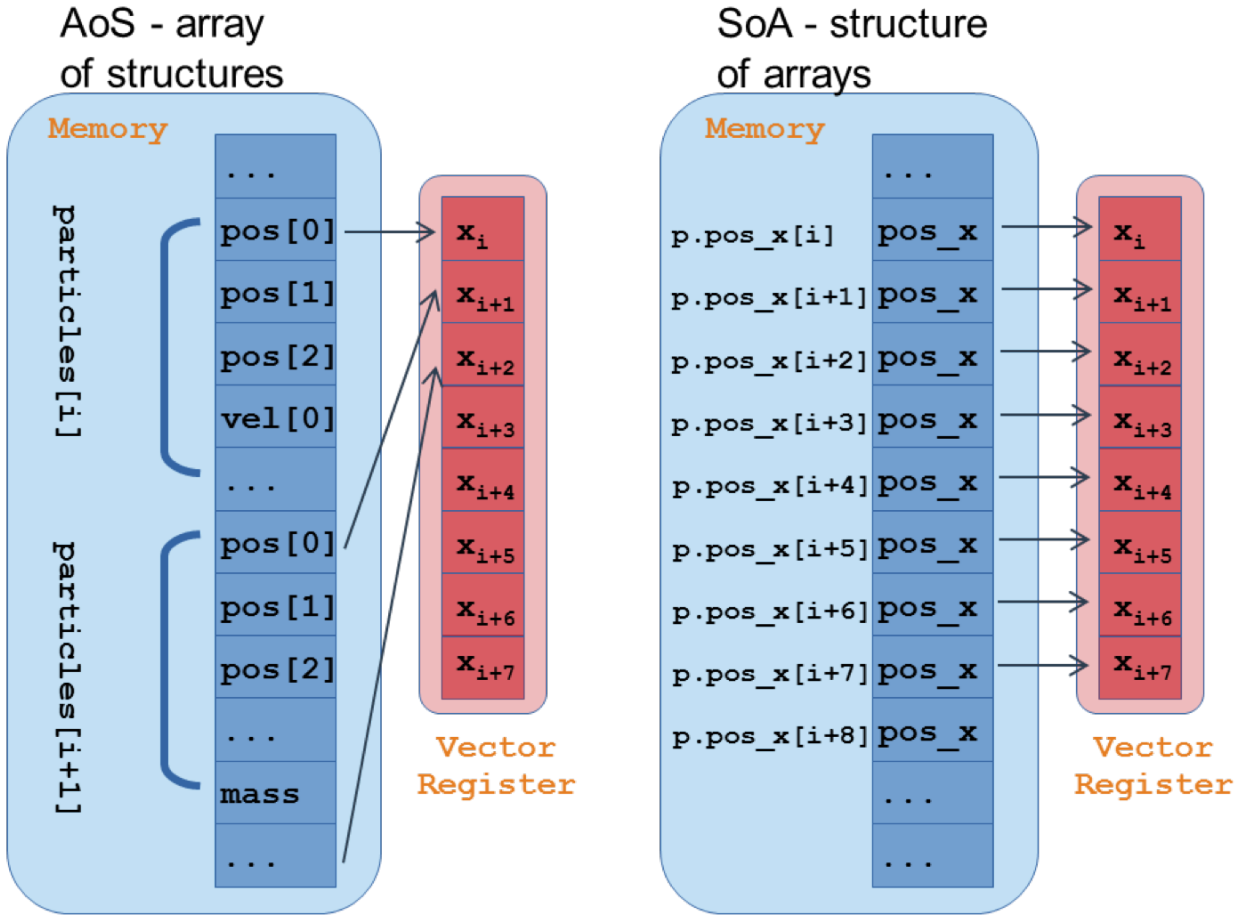
- Storage of particles is in an Array Of Structures (AOS) style
- This leads to regular, but non-unit strides in memory access
- 33% unit
- 33% uniform, non-unit
- 33% non-uniform
- Re-structuring the code into a Structure Of Arrays (SOA) may lead to unit stride access and more effective vectorization

Vectorization: gather/scatter operation

- The compiler might generate gather/scatter instructions for loops automatically vectorized where memory locations are not contiguous

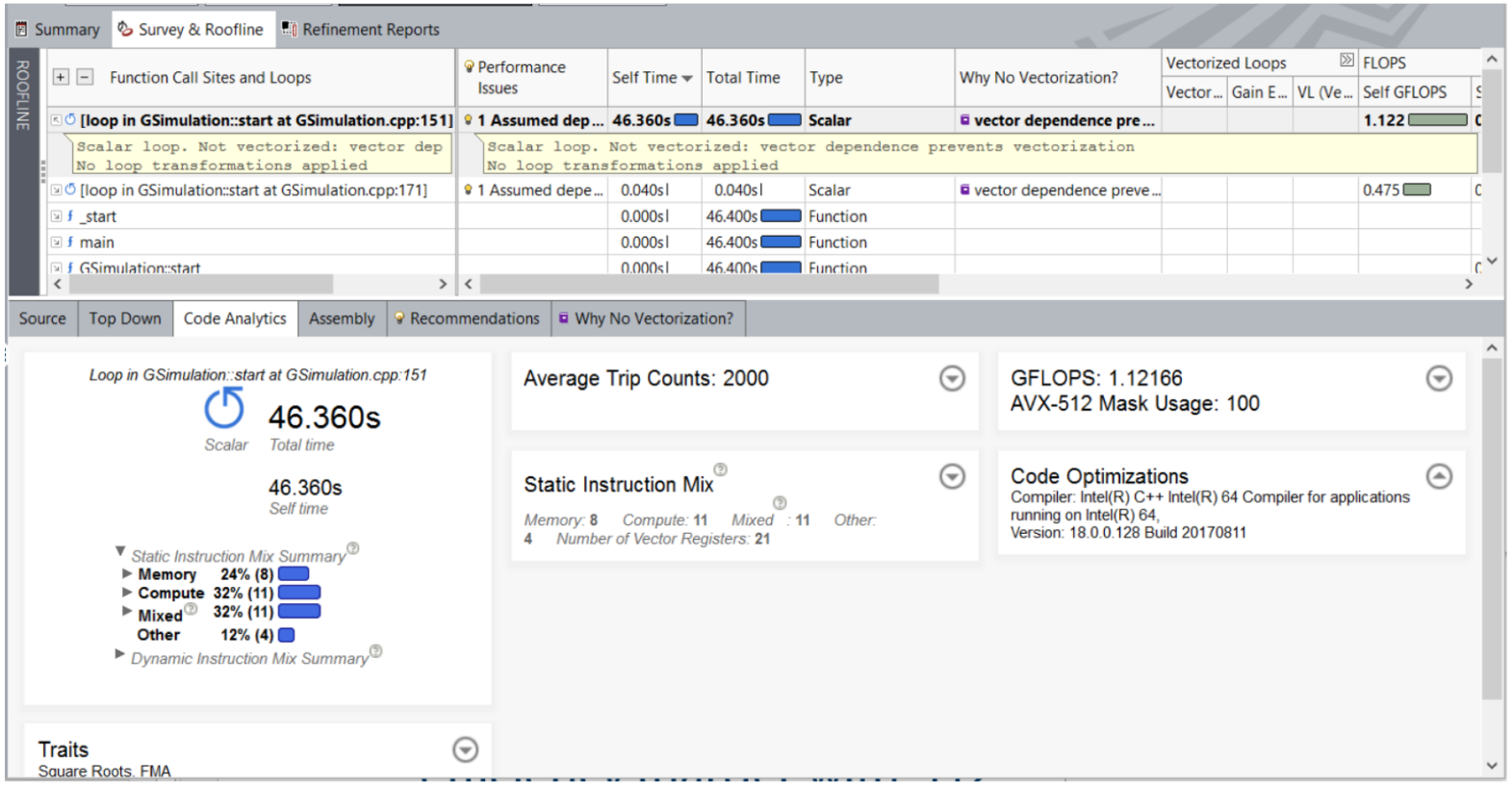
```
struct Particle
{
 public:
   ...
   real_type pos[3];
   real_type vel[3];
   real_type acc[3];
   real_type mass;
};
```

```
struct ParticleSoA
{
 public:
   ...
   real_type *pos_x,*pos_y,*pos_z;
   real_type *vel_x,*vel_y,*vel_z;
   real_type *acc_x,*acc_y;*acc_z
   real_type *mass;
};
```



Performance After Data Structure Change

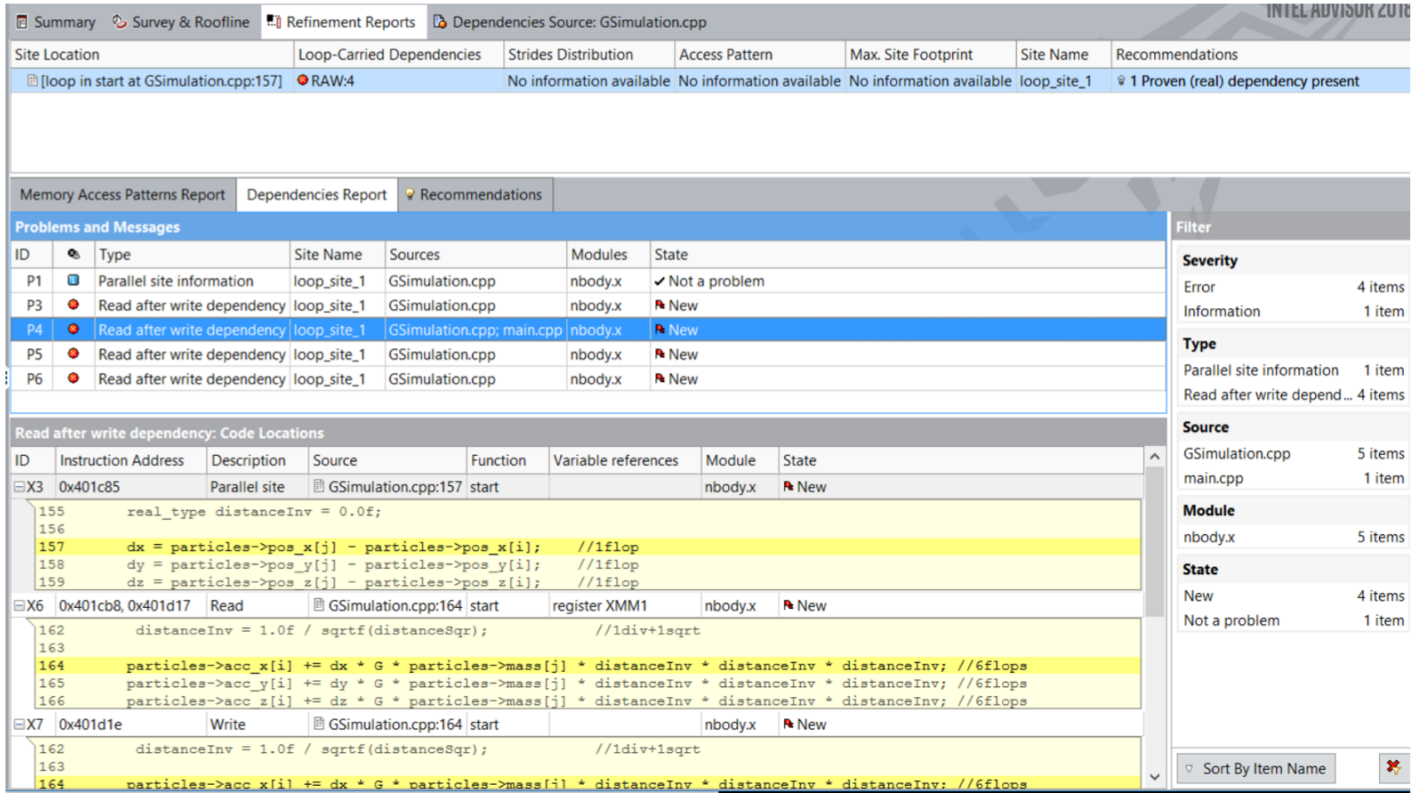
- In this new version (version 3 in github sample) we introduce the following change:
- Change particle data structures from AOS to SOA
- Note changes in report:
- Performance is lower
- Main loop is no longer vectorized
- Assumed vector dependence prevents automatic vectorization



Next step is clear: perform a Dependencies analysis

Dependencies Analysis (Refinement)

```
aprun -n 1 -N 1 advixe-cl --collect dependencies --project-dir ./adv_res \
                --search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```



- Dependencies analysis has high overhead:
- Run on reduced workload
- Advisor Findings:
- RAW dependency
- Multiple reduction-type dependencies

Recommendations

Memory Access Patterns Report | Dependencies Report | Recommendations

All Advisor-detectable issues: C++ | Fortran

Recommendation: Resolve dependency

The Dependencies analysis shows there is a real (proven) dependency in the loop. To fix: Do one of the following:

- If there is an anti-dependency, enable vectorization using the directive `#pragma omp simd safelen(length)` , where `length` is smaller than the distance between dependent iterations in anti-dependency. For example:

```
#pragma omp simd safelen(4)
for (i = 0; i < n - 4; i += 4)
{
    a[i + 4] = a[i] * c;
}
```

- If there is a reduction pattern dependency in the loop, enable vectorization using the directive `#pragma omp simd reduction(operator:list)` . For example:

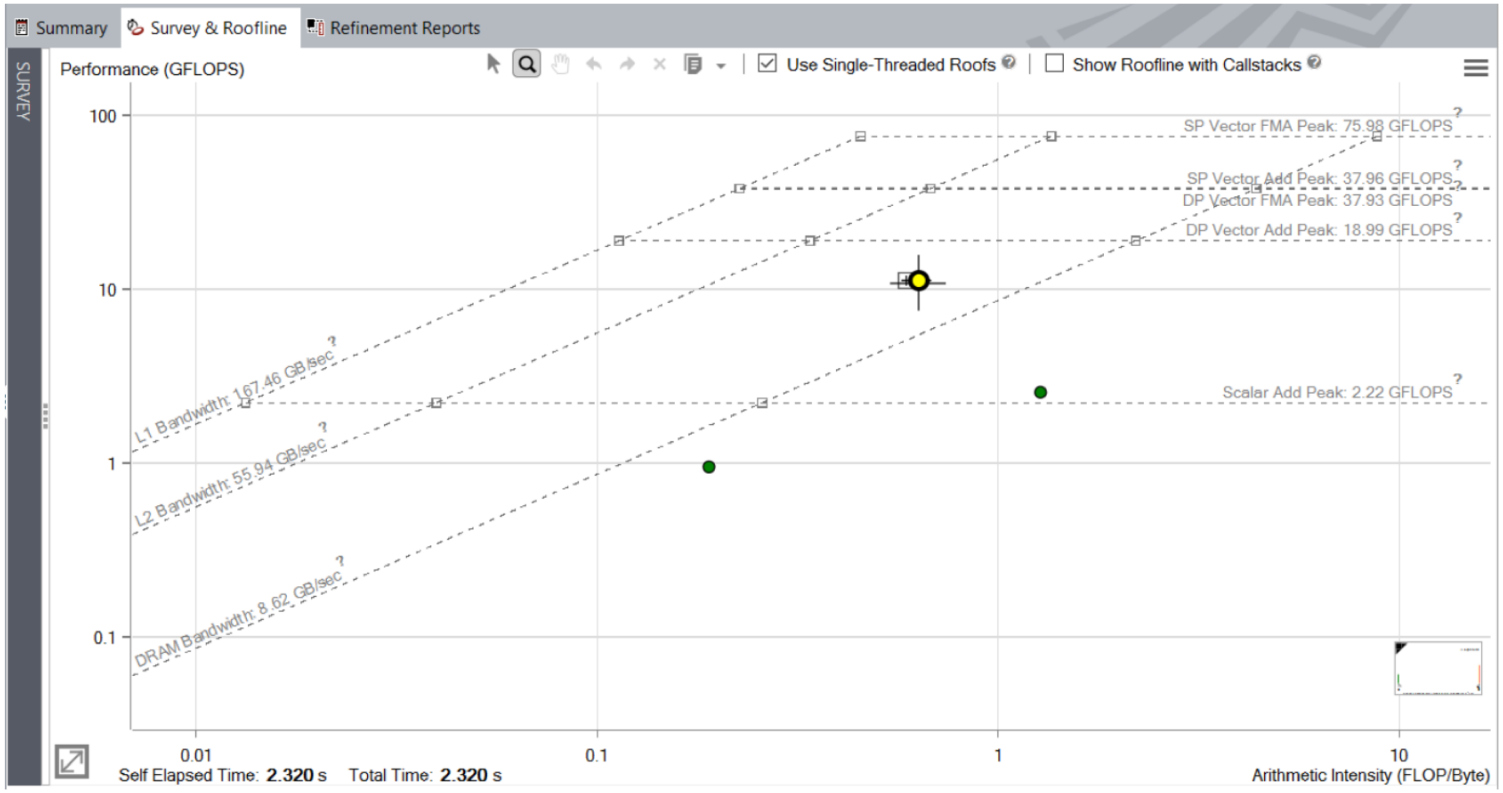
```
#pragma omp simd reduction(+:sumx)
for (k = 0;k < size2; k++)
{
    sumx += x[k]*b[k];
}
```

ISSUE: PROVEN (REAL) DEPENDENCY PRESENT

The compiler assumed there is an anti-dependency (Write after read - WAR) or true dependency (Read after write - RAW) in the loop. Improve performance by investigating the assumption and handling accordingly.

Resolve dependency

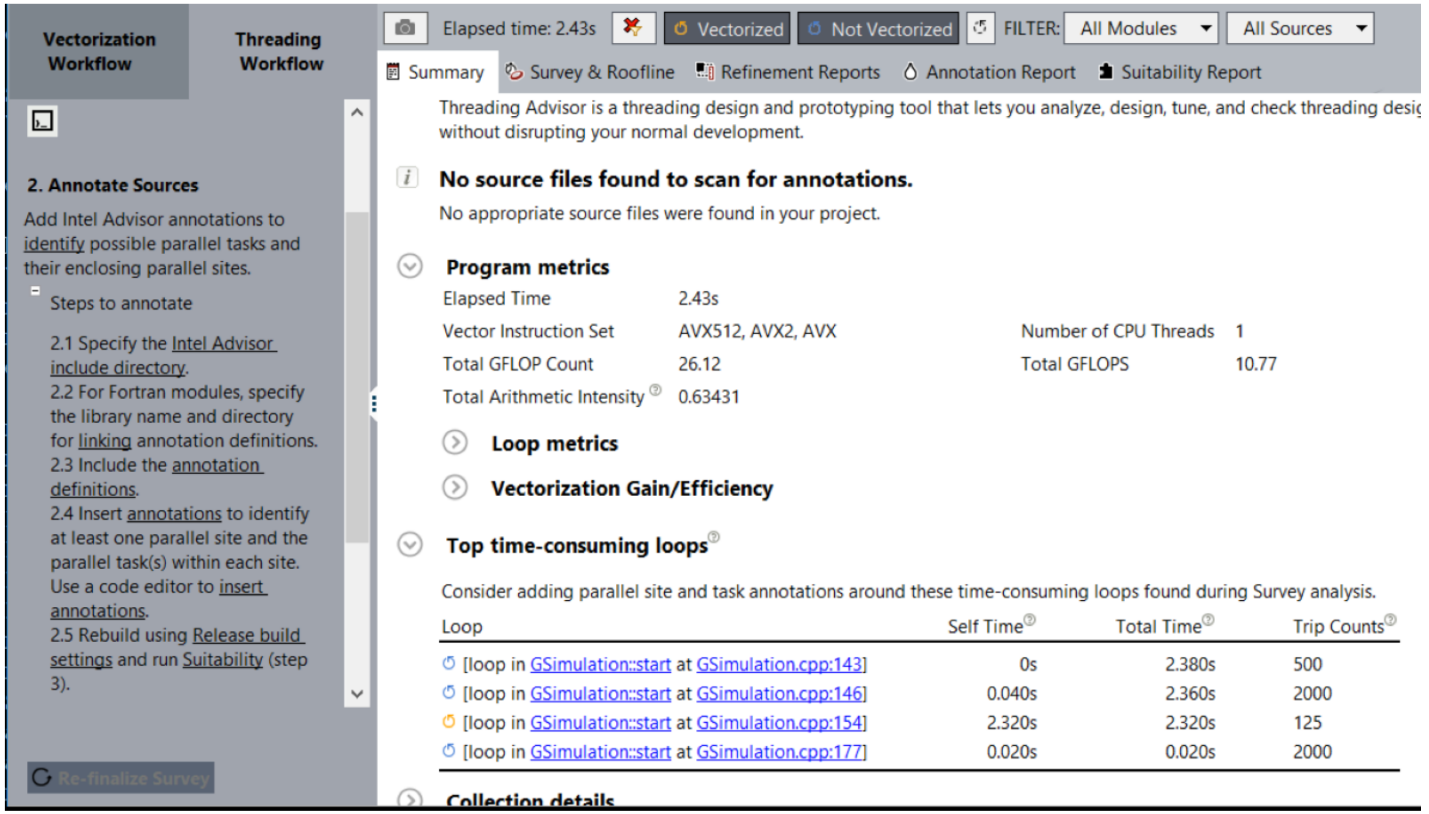
Performance After Resolved Dependencies



New memory access pattern plus vectorization produces much improved performance!

What next?

- Let’s explore threading with a suitability analysis.
- Recompile including annotation definitions
- Add headers to file
- Annotate suggested loops
- Run suitability collection



Annotating the code

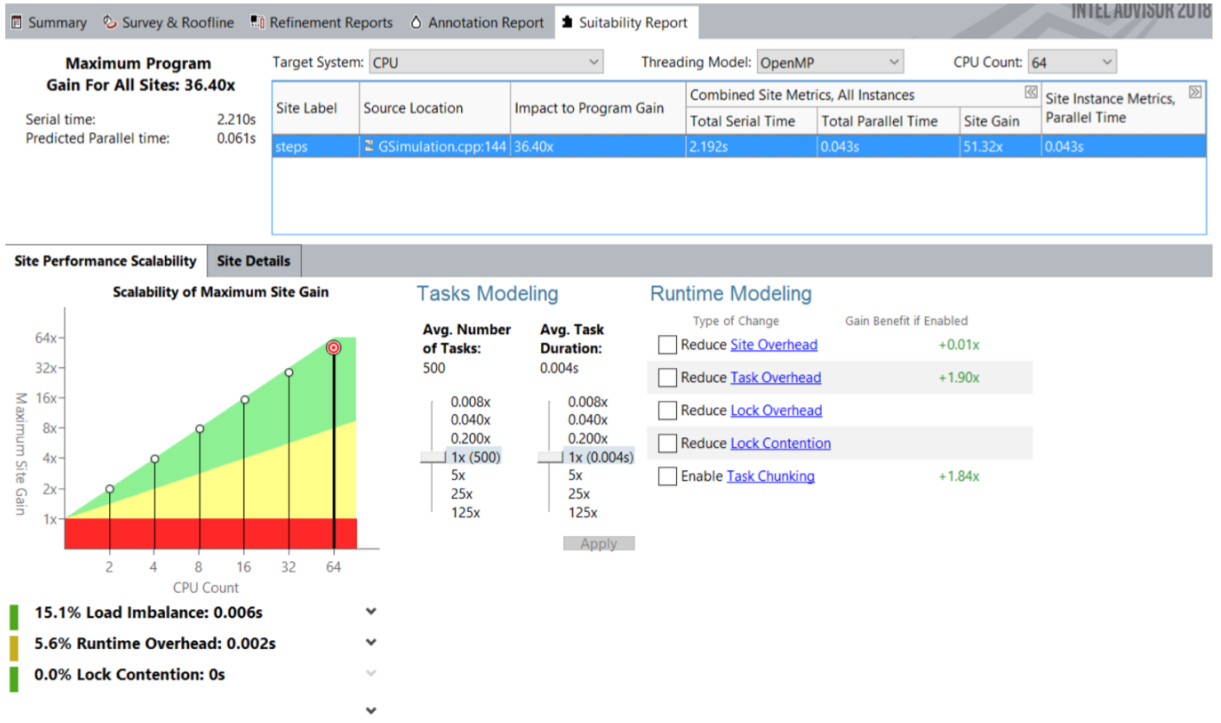
- Add annotations as shown on the left sample
- Complex sites may be analyzed in more detail using task sections if needed
  - ANNOTATE_SITE_BEGIN / ANNOTATE_SITE_END
  - ANNOTATE_TASK_BEGIN / ANNOTATE_TASK_END
- Recompile including annotation definitions:
  - `-I/opt/intel/advisor/include`
- Collect suitability data

```
#include "advisor-annotate.h"
...

ANNOTATE_SITE_BEGIN(steps)
for (int s=1; s<=get_nsteps(); ++s)
{
  ...

  ANNOTATE_TASK_BEGIN(particles)
  for (i = 0; i < n; i++)
  {
    ...

  }
  ANNOTATE_TASK_END(particles)
}
ANNOTATE_SITE_END(steps)
```

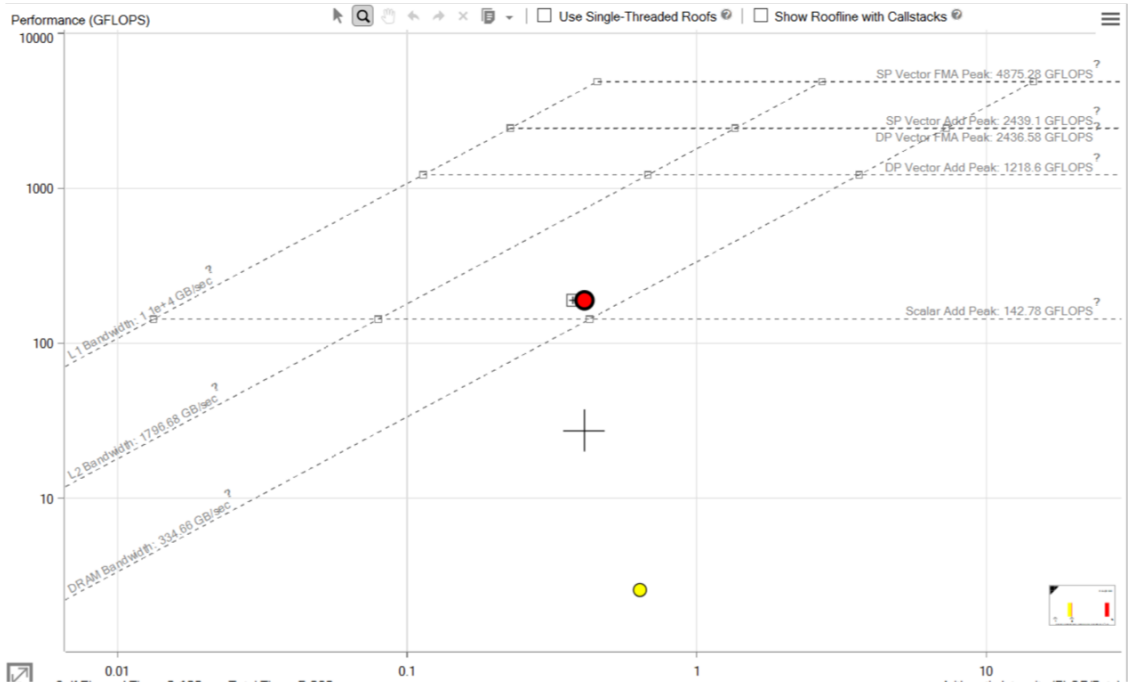
```
mpirun -n 1 -N 1 advixe-cl --collect suitability --project-dir ./adv_res \
                --search-dir src:=./ --search-dir bin:=./ -- ./nbody.x
```

Suitability report

- Good speedup expected, but far from ideal (~56% efficiency).
- Modeling shows that increasing the task length would improve efficiency.
- Next step: add omp parallel region to code and re-test



Roofline for Threaded Version



Now using regular roofline, instead of single-threaded

Still room for improvement, but at this point we need additional detail regarding shared resource utilization

```
    for (int s=1; s<=get_nsteps(); ++s)
    {
      ts0 += time.start();

    #pragma omp parallel for
      for (i = 0; i < n; i++)    // update
  acceleration
      {
        ...
        real_type ax_i = particles->acc_x[i];
        real_type ay_i = particles->acc_y[i];
        real_type az_i = particles->acc_z[i];

    #pragma omp simd reduction(+:ax_i,ay_i,az_i)
        for (j = 0; j < n; j++)
        {
          real_type dx, dy, dz;
          real_type distanceSqr = 0.0f;
          real_type distanceInv = 0.0f;

      dx = particles->pos_x[j] - particles->pos_x[i];
```