INTEL® ADVISOR

DEMO

Dr. Fabio Baruffa

Sr. HPC Apps. Engineer, Intel IAGS

INTEL ADVISOR AND ROOFLINE

"Automatic" Vectorization Often Not Enough

A good compiler can still benefit greatly from vectorization optimization

Compiler will not always vectorize

- Check for Loop Carried Dependencies using Intel® Advisor
- All clear? Force vectorization.
  C++ use: pragma simd, Fortran use: SIMD directive

Not all vectorization is efficient vectorization

- Stride of 1 is more cache efficient than stride of 2 and greater. Analyze with Intel® Advisor.
- Consider data layout changes
  Intel® SIMD Data Layout Templates can help

Benchmarks on prior slides did not all "auto vectorize." Compiler directives were used to force vectorization and get more performance.

Arrays of structures are great for intuitively organizing data, but are much less efficient than structures of arrays. Use the Intel® SIMD Data Layout Templates (Intel® SDLT) to map data into a more efficient layout for vectorization.

Optimization Notice

Get Breakthrough Vectorization Performance

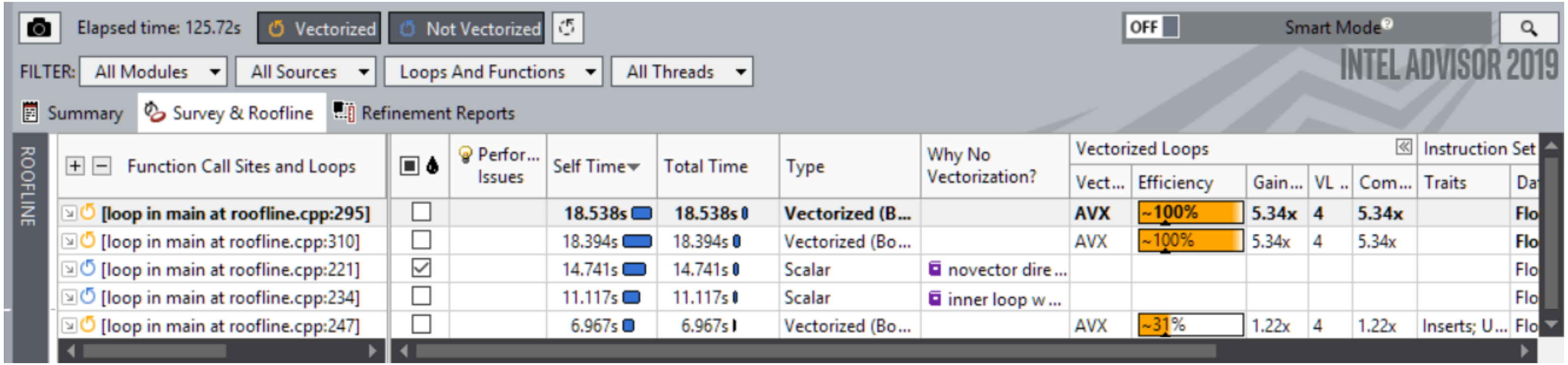
Intel® Advisor—Vectorization Advisor

Faster Vectorization Optimization

- Vectorize where it will pay off most
- Quickly ID what is blocking vectorization
- Tips for effective vectorization
- Safely force compiler vectorization
- Optimize memory stride

Data & Guidance You Need

- Compiler diagnostics + Performance Data + SIMD efficiency
- Detect problems & recommend fixes
- Loop-Carried Dependency Analysis
- Memory Access Patterns Analysis



Function Call Sites and Loops		Perfor... Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops					Instruction Set	
							Vect...	Efficiency	Gain...	VL ..	Com...	Traits	Da
[loop in main at roofline.cpp:295]	☐		**18.538s**	**18.538s**	**Vectorized (B...**		**AVX**	**~100%**	**5.34x**	**4**	**5.34x**		**Flo**
[loop in main at roofline.cpp:310]	☐		18.394s	18.394s	Vectorized (Bo...		AVX	~100%	5.34x	4	5.34x		Flo
[loop in main at roofline.cpp:221]	☑		14.741s	14.741s	Scalar	novector dire...							Flo
[loop in main at roofline.cpp:234]	☐		11.117s	11.117s	Scalar	inner loop w...							Flo
[loop in main at roofline.cpp:247]	☐		6.967s	6.967s	Vectorized (Bo...		AVX	~31%	1.22x	4	1.22x	Inserts; U...	Flo

Optimize for Intel® AVX-512 with or without access to AVX-512 hardware

http://intel.ly/advisor-xe

ROOFLINE MODEL

Cache-Aware Roofline

Next Steps

If Under the Vector Add Peak

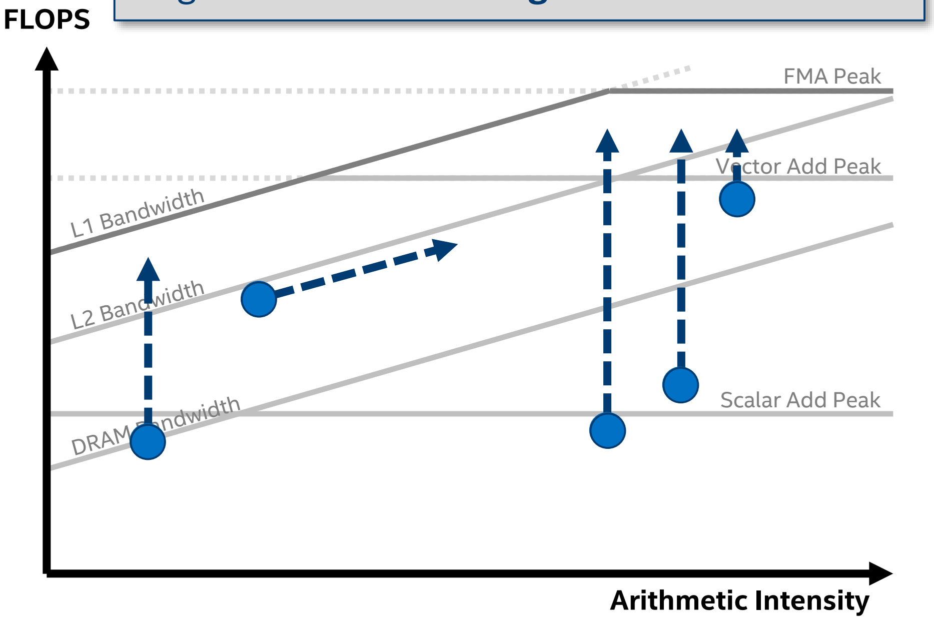
Check “Traits” in the Survey to see if FMAs are used. If not, try altering your code or compiler flags to **induce FMA usage.**

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 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 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.

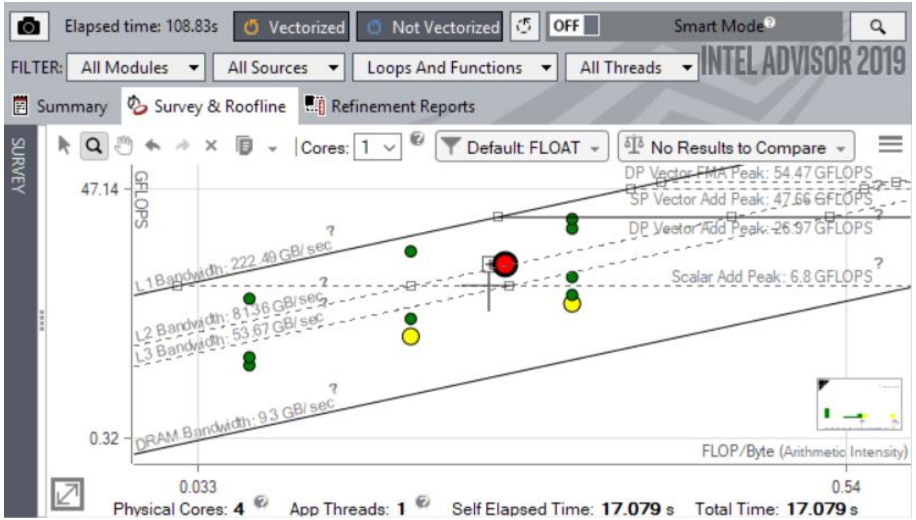
Optimization Notice

Find Effective Optimization Strategies

Intel® Advisor—Cache-aware Roofline Analysis

Roofline Performance Insights

- Highlights poor performing loops
- Shows performance ‘headroom’ for each loop
  - Which can be improved
  - Which are worth improving
- Shows likely causes of bottlenecks
- Suggests next optimization steps



Elapsed time: 108.83s

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

Cores: 1 · Default: FLOAT · No Results to Compare

Physical Cores: 4 · App Threads: 1 · Self Elapsed Time: 17.079 s · Total Time: 17.079 s

> *“I am enthusiastic about the new "integrated roofline" in Intel® Advisor. It is now possible to proceed with a step-by-step approach with the difficult question of memory transfers optimization & vectorization which is of major importance.”*

Nicolas Alferez, Software Architect
Onera – The French Aerospace Lab

Optimization Notice

ADVISOR DEMO

Validating Vectorization Success I: Compiler report

- -qopt-report[=n]: tells the compiler to generate an optimization report
  - n: (Optional) Indicates the level of detail in the report. You can specify values 0 through 5. If you specify zero, no report is generated. For levels n=1 through n=5, each level includes all the information of the previous level, as well as potentially some additional information. Level 5 produces the greatest level of detail. If you do not specify n, the default is level 2, which produces a medium level of detail.
- -qopt-report-phase[=list]: specifies one or more optimizer phases for which optimization reports are generated.
  - loop: the phase for loop nest optimization
  - vec: the phase for vectorization
  - par: the phase for auto-parallelization
  - all: all optimizer phases
- -qopt-report-filter=string: specified the indicated parts of your application, and generate optimization reports for those parts of your application.

Validating Vectorization Success II

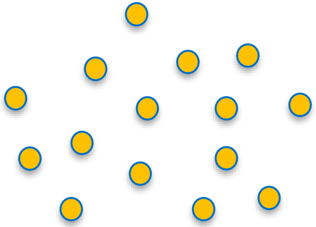
- -S: assembler code inspection
  - Most reliable way and gives all details of course
  - Check for scalar/packed or (E)VEX encoded instructions:
    Assembler listing contains source line numbers for easier navigation
  - Compiling with -qopt-report-embed (Linux*, macOS*) helps interpret assembly code
- Performance validation
  - Compile and benchmark with -no-vec –qno-openmp-simd or on a loop by loop basis via #pragma novector or !DIR$ NOVECTOR
  - Compile and benchmark with selected SIMD feature
  - Compare runtime differences

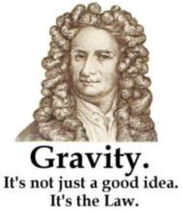
Demo: Nbody gravity simulation

Let's consider a distribution of point masses located at `r_1,…,r_n` and have masses `m_1,…,m_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;
};
```





$$\vec{F}_{ij} = \frac{G m_i m_j}{\left|\vec{r}_j - \vec{r}_i\right|^3} \left(\vec{r}_j - \vec{r}_i\right)$$

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = m\frac{d^2\vec{x}}{dt^2}$$

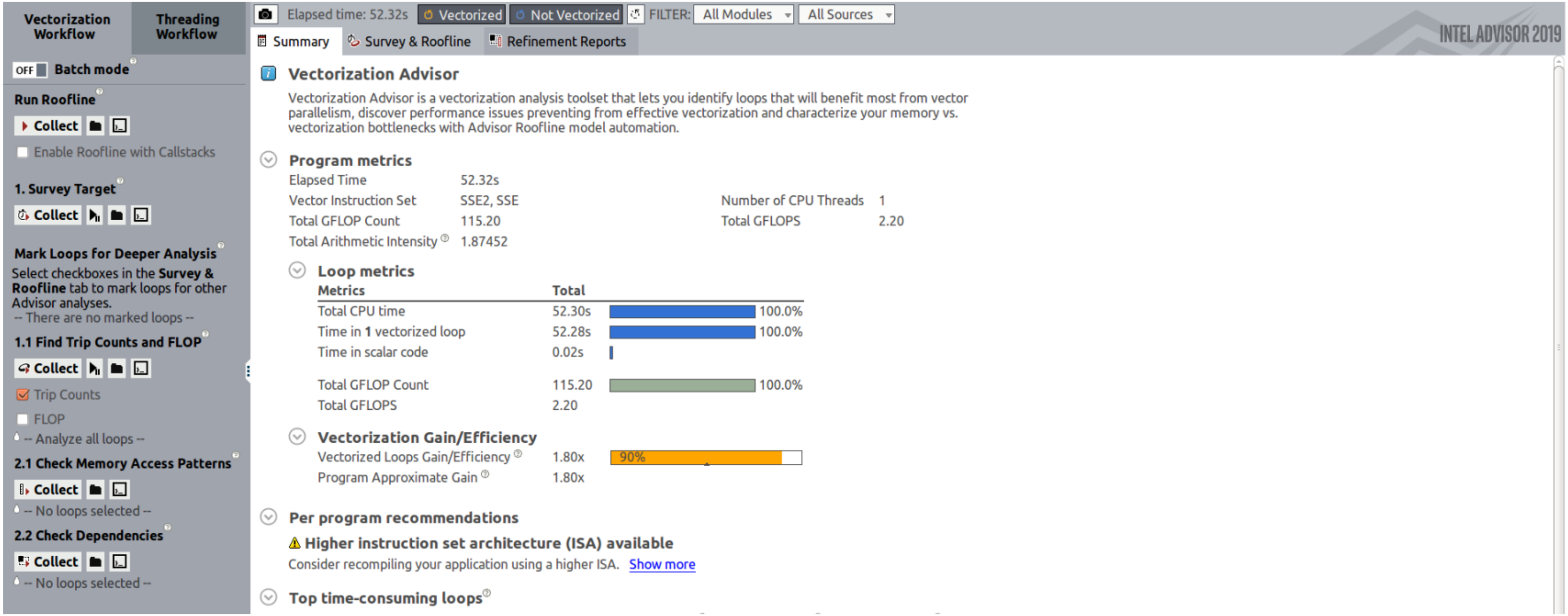
Demo: Nbody kernel implementation

```
GSimulation.cpp:
...
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];
        dy = particles[j].pos[1] - particles[i].pos[1];
        dz = particles[j].pos[2] - particles[i].pos[2];

        distSqr = dx*dx + dy*dy + dz*dz + softeningSquared;
        distInv = 1.0 / sqrt(distanceSqr);
        particles[i].acc[0] += dx * G * particles[j].mass * distInv * distInv * distInv;
        particles[i].acc[1] += …
        particles[i].acc[2] += …
    }
}
...
```

Demo – nbody-sim/ver0

- Go to the folder nbody-sim/ver0
- Type *make* to compile code.
- Type *make survey* to run the Survey Analysis of Advisor.
- Once you have setup the VNC connection (see previous instructions), open the Advisor results via the GUI, typing *make open-gui* .
- For the Roofline Analysis, run *make roofline* .

Demo – nbody-sim/ver0: Advisor Summary



Select checkboxes in the **Survey & Roofline** tab to mark loops for other Advisor analyses.

-- There are no marked loops --

Elapsed time: 52.32s

Vectorization Advisor

Vectorization Advisor is a vectorization analysis toolset that lets you identify loops that will benefit most from vector parallelism, discover performance issues preventing from effective vectorization and characterize your memory vs. vectorization bottlenecks with Advisor Roofline model automation.

Program metrics

Elapsed Time	52.32s		
Vector Instruction Set	SSE2, SSE	Number of CPU Threads	1
Total GFLOP Count	115.20	Total GFLOPS	2.20
Total Arithmetic Intensity	1.87452		

Loop metrics

Metrics	Total	
Total CPU time	52.30s	100.0%
Time in **1** vectorized loop	52.28s	100.0%
Time in scalar code	0.02s	
Total GFLOP Count	115.20	100.0%
Total GFLOPS	2.20	

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	1.80x	90%
Program Approximate Gain	1.80x	

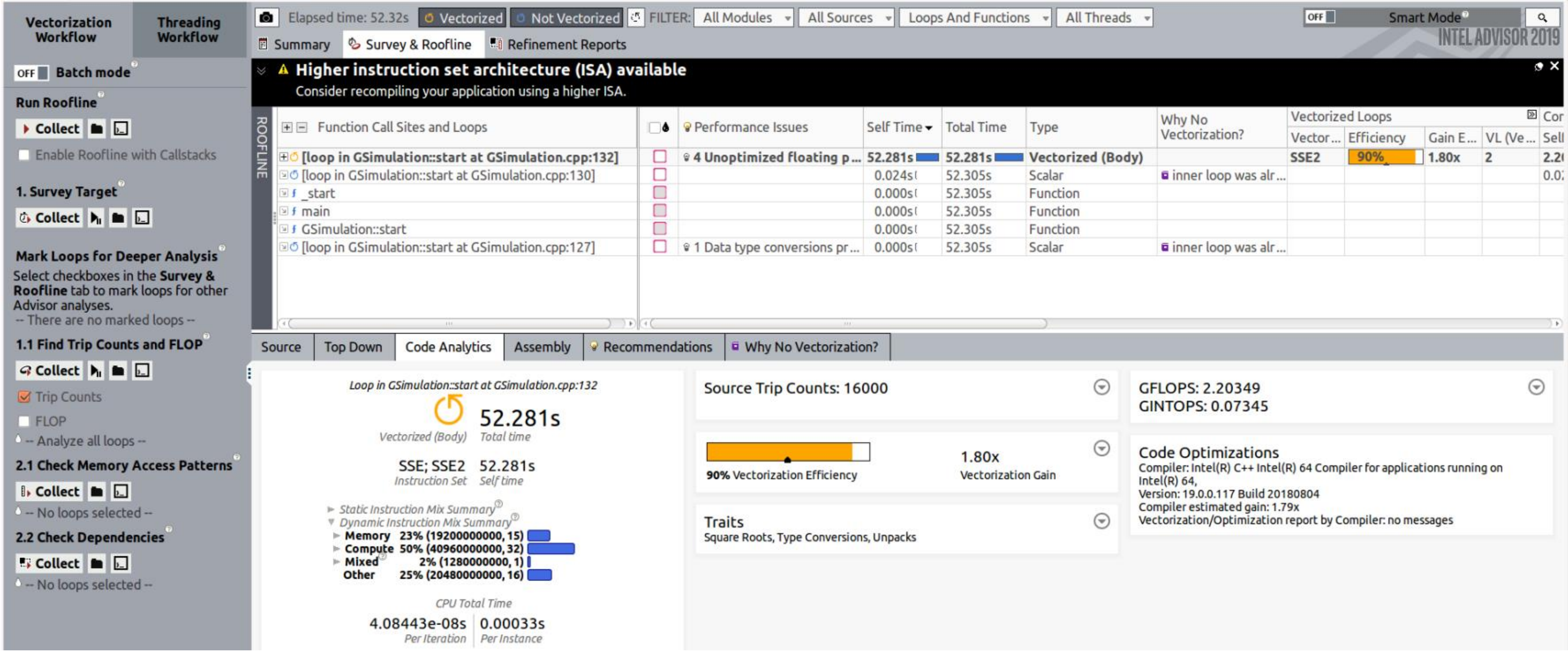
Per program recommendations

⚠ **Higher instruction set architecture (ISA) available**

Consider recompiling your application using a higher ISA. Show more

Top time-consuming loops

Demo – nbody-sim/ver0: Code Analytics



Select checkboxes in the **Survey & Roofline** tab to mark loops for other Advisor analyses.
-- There are no marked loops --

☐ Enable Roofline with Callstacks

☑ Trip Counts

☐ FLOP

-- Analyze all loops --

-- No loops selected --

-- No loops selected --

Higher instruction set architecture (ISA) available
Consider recompiling your application using a higher ISA.

Function Call Sites and Loops		Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				Cor
							Vector...	Efficiency	Gain E...	VL (Ve...	Sell
[loop in GSimulation::start at GSimulation.cpp:132]	☐	**4 Unoptimized floating p...**	**52.281s**	**52.281s**	**Vectorized (Body)**		**SSE2**	**90%**	**1.80x**	**2**	**2.2**
[loop in GSimulation::start at GSimulation.cpp:130]	☐		0.024s	52.305s	Scalar	inner loop was alr...					0.0
_start	☐		0.000s	52.305s	Function						
main	☐		0.000s	52.305s	Function						
GSimulation::start	☐		0.000s	52.305s	Function						
[loop in GSimulation::start at GSimulation.cpp:127]	☐	1 Data type conversions pr...	0.000s	52.305s	Scalar	inner loop was alr...					

Loop in GSimulation::start at GSimulation.cpp:132

52.281s *Total time*
Vectorized (Body)

SSE; SSE2 *Instruction Set*
52.281s *Self time*

Static Instruction Mix Summary
Dynamic Instruction Mix Summary

Memory	**23% (19200000000, 15)**
Compute	**50% (40960000000, 32)**
Mixed	**2% (1280000000, 1)**
Other	**25% (20480000000, 16)**

CPU Total Time
4.08443e-08s *Per Iteration*
0.00033s *Per Instance*

Source Trip Counts: 16000

90% Vectorization Efficiency

1.80x Vectorization Gain

Traits
Square Roots, Type Conversions, Unpacks

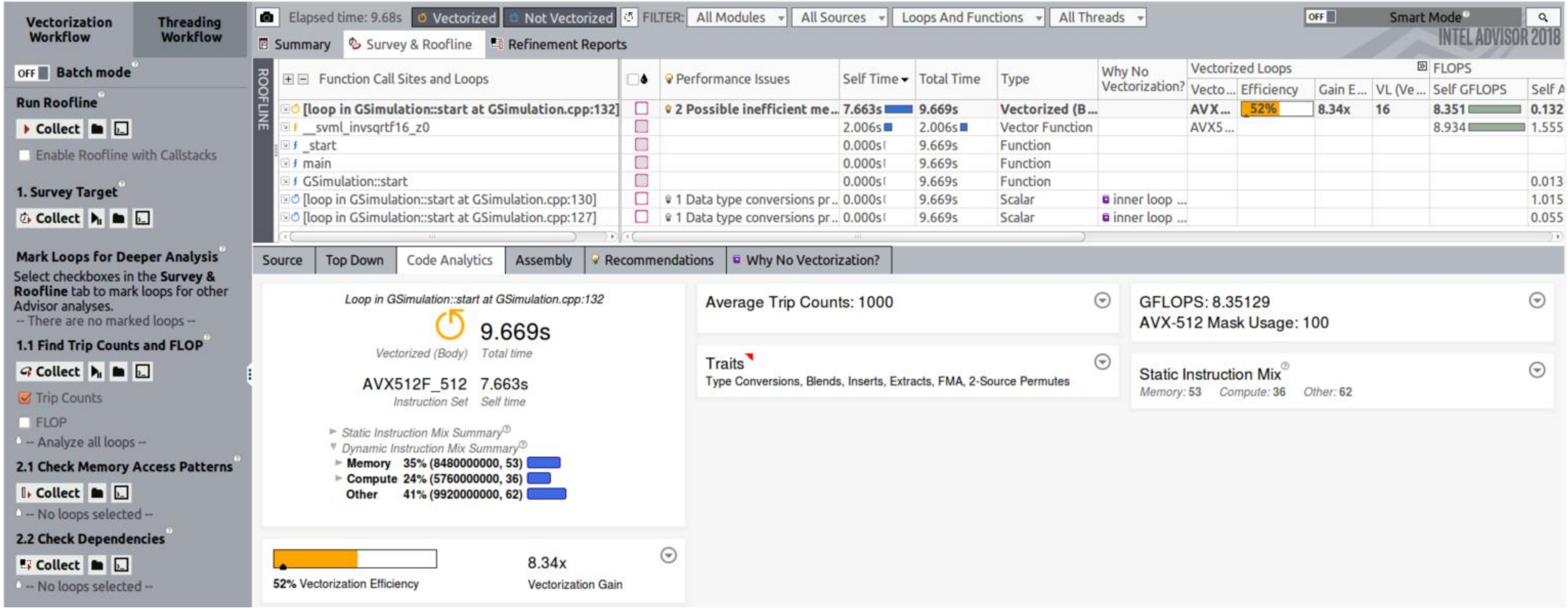
GFLOPS: 2.20349
GINTOPS: 0.07345

Code Optimizations
Compiler: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64,
Version: 19.0.0.117 Build 20180804
Compiler estimated gain: 1.79x
Vectorization/Optimization report by Compiler: no messages

Demo – nbody-sim/ver-avx512

- Go to the folder nbody-sim/ver-avx512
- Type *make* to compile code.
- Type *make survey* to run the Survey Analysis of Advisor.
- Once you have setup the VNC connection (see previous instructions), open the Advisor results via the GUI, typing *make open-gui* .
- For the Roofline Analysis, run *make roofline* .

Demo – nbody-sim/ver-avx512: Code Analytics



Select checkboxes in the **Survey & Roofline** tab to mark loops for other Advisor analyses.

-- There are no marked loops --

Function Call Sites and Loops	☐	Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				FLOPS	
							Vecto...	Efficiency	Gain E...	VL (Ve...	Self GFLOPS	Self A
[loop in GSimulation::start at GSimulation.cpp:132]	☐	**2 Possible inefficient me...**	**7.663s**	**9.669s**	**Vectorized (B...**		**AVX...**	**52%**	**8.34x**	**16**	**8.351**	**0.132**
__svml_invsqrtf16_z0	☐		2.006s	2.006s	Vector Function		AVX5...				8.934	1.555
_start	☐		0.000s	9.669s	Function							
main	☐		0.000s	9.669s	Function							
GSimulation::start	☐		0.000s	9.669s	Function							0.013
[loop in GSimulation::start at GSimulation.cpp:130]	☐	1 Data type conversions pr...	0.000s	9.669s	Scalar	inner loop ...						1.015
[loop in GSimulation::start at GSimulation.cpp:127]	☐	1 Data type conversions pr...	0.000s	9.669s	Scalar	inner loop ...						0.055

Loop in GSimulation::start at GSimulation.cpp:132

9.669s *Total time*

Vectorized (Body)

AVX512F_512 *Instruction Set*

7.663s *Self time*

Static Instruction Mix Summary

Dynamic Instruction Mix Summary

Memory	**35% (8480000000, 53)**
Compute	**24% (5760000000, 36)**
Other	**41% (9920000000, 62)**

52% Vectorization Efficiency

8.34x Vectorization Gain

Average Trip Counts: 1000

Traits

Type Conversions, Blends, Inserts, Extracts, FMA, 2-Source Permutes

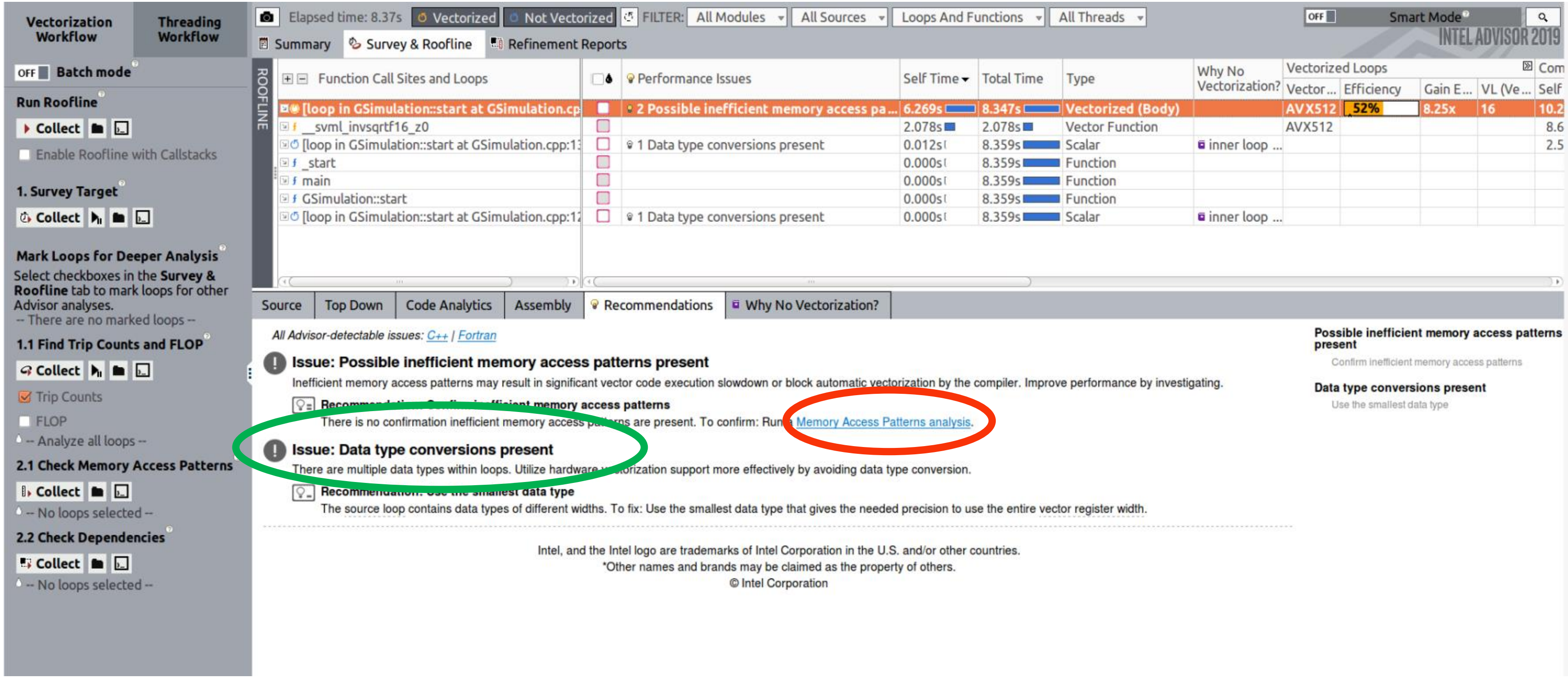
GFLOPS: 8.35129

AVX-512 Mask Usage: 100

Static Instruction Mix

Memory: 53 *Compute:* 36 *Other:* 62

Demo – nbody-sim/ver-avx512: Recommendations



Function Call Sites and Loops		Performance Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops				Com
							Vector ...	Efficiency	Gain E...	VL (Ve...	Self
[loop in GSimulation::start at GSimulation.cp	☐	2 Possible inefficient memory access pa...	6.269s	8.347s	Vectorized (Body)		AVX512	52%	8.25x	16	10.2
__svml_invsqrtf16_z0	☐		2.078s	2.078s	Vector Function		AVX512				8.6
[loop in GSimulation::start at GSimulation.cpp:13	☐	1 Data type conversions present	0.012s	8.359s	Scalar	inner loop ...					2.5
_start	☐		0.000s	8.359s	Function						
main	☐		0.000s	8.359s	Function						
GSimulation::start	☐		0.000s	8.359s	Function						
[loop in GSimulation::start at GSimulation.cpp:12	☐	1 Data type conversions present	0.000s	8.359s	Scalar	inner loop ...					

All Advisor-detectable issues: C++ | Fortran

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.

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: Data type conversions present

There are multiple data types within loops. Utilize hardware vectorization support more effectively by avoiding data type conversion.

Recommendation: Use the smallest data type

The source loop contains data types of different widths. To fix: Use the smallest data type that gives the needed precision to use the entire vector register width.

Possible inefficient memory access patterns present

Confirm inefficient memory access patterns

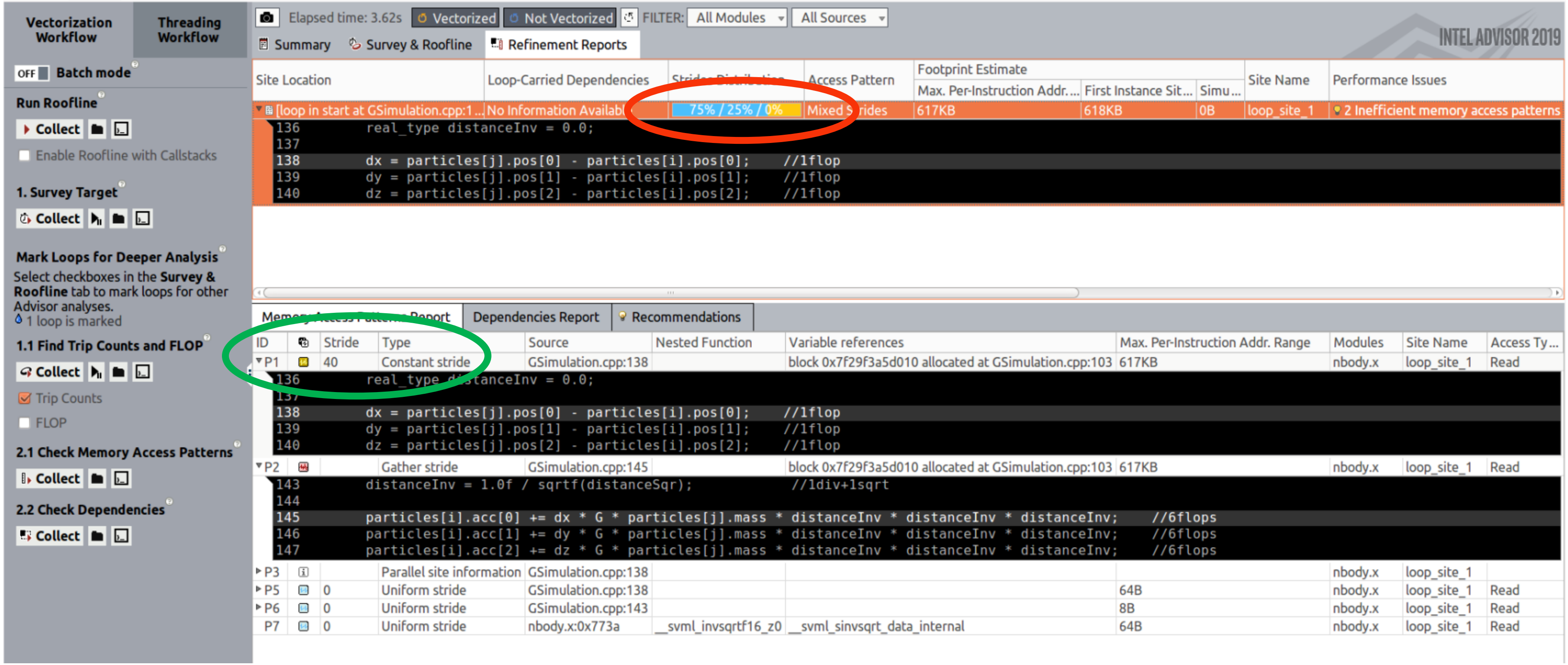
Data type conversions present

Use the smallest data type

Demo – nbody-sim/ver-avx512

- Go to the folder nbody-sim /ver-avx512
- Type *make clean-results* to delete the previous data.
- Generate a new Survey Analysis of Advisor and Roofline:
    - *make roofline*
- To run the MAP Analysis: *make map*

Demo – nbody-sim/ver-avx512: Map Analysis



Select checkboxes in the **Survey & Roofline** tab to mark loops for other Advisor analyses.

1 loop is marked

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Footprint Estimate: Max. Per-Instruction Addr...	Footprint Estimate: First Instance Sit...	Footprint Estimate: Simu...	Site Name	Performance Issues
[loop in start at GSimulation.cpp:1...	No Information Availabl	75% / 25% / 0%	Mixed Strides	617KB	618KB	0B	loop_site_1	2 Inefficient memory access patterns

```
136        real_type distanceInv = 0.0;
137
138        dx = particles[j].pos[0] - particles[i].pos[0];    //1flop
139        dy = particles[j].pos[1] - particles[i].pos[1];    //1flop
140        dz = particles[j].pos[2] - particles[i].pos[2];    //1flop
```

ID		Stride	Type	Source	Nested Function	Variable references	Max. Per-Instruction Addr. Range	Modules	Site Name	Access Ty...
P1		40	Constant stride	GSimulation.cpp:138		block 0x7f29f3a5d010 allocated at GSimulation.cpp:103	617KB	nbody.x	loop_site_1	Read

```
136        real_type distanceInv = 0.0;
137
138        dx = particles[j].pos[0] - particles[i].pos[0];    //1flop
139        dy = particles[j].pos[1] - particles[i].pos[1];    //1flop
140        dz = particles[j].pos[2] - particles[i].pos[2];    //1flop
```

ID		Stride	Type	Source	Nested Function	Variable references	Max. Per-Instruction Addr. Range	Modules	Site Name	Access Ty...
P2			Gather stride	GSimulation.cpp:145		block 0x7f29f3a5d010 allocated at GSimulation.cpp:103	617KB	nbody.x	loop_site_1	Read

```
143        distanceInv = 1.0f / sqrtf(distanceSqr);            //1div+1sqrt
144
145        particles[i].acc[0] += dx * G * particles[j].mass * distanceInv * distanceInv * distanceInv;    //6flops
146        particles[i].acc[1] += dy * G * particles[j].mass * distanceInv * distanceInv * distanceInv;    //6flops
147        particles[i].acc[2] += dz * G * particles[j].mass * distanceInv * distanceInv * distanceInv;    //6flops
```

ID		Stride	Type	Source	Nested Function	Variable references	Max. Per-Instruction Addr. Range	Modules	Site Name	Access Ty...
P3			Parallel site information	GSimulation.cpp:138				nbody.x	loop_site_1	
P5		0	Uniform stride	GSimulation.cpp:138			64B	nbody.x	loop_site_1	Read
P6		0	Uniform stride	GSimulation.cpp:143			8B	nbody.x	loop_site_1	Read
P7		0	Uniform stride	nbody.x:0x773a	__svml_invsqrtf16_z0	__svml_sinvsqrt_data_internal	64B	nbody.x	loop_site_1	Read

Optimization Notice

Memory access pattern

How should I access data ?

- Unit stride access are faster

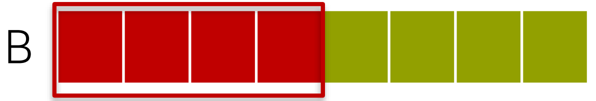
```
for (i=0; i<N; i++)
     A[i] = B[i]*d
```

- Constant stride are more complex

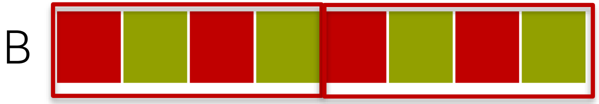
```
for (i=0; i<N; i+=2)
     A[i] = B[i]*d
```

- Non predictable access are usually bad

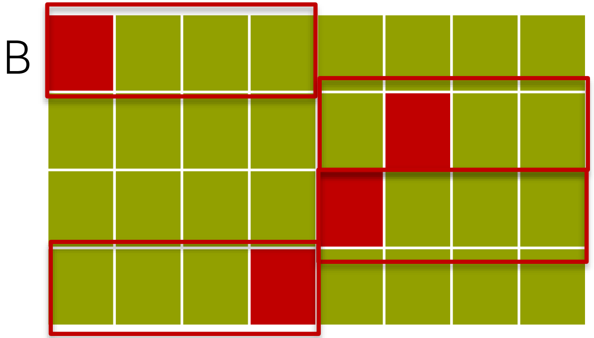
```
for (i=0; i<N; i++)
     A[i] = B[C[i]]*d
```



For B, 1 cache line load computes 4 DP



For B, 2 cache line loads compute 4 DP with reconstructions

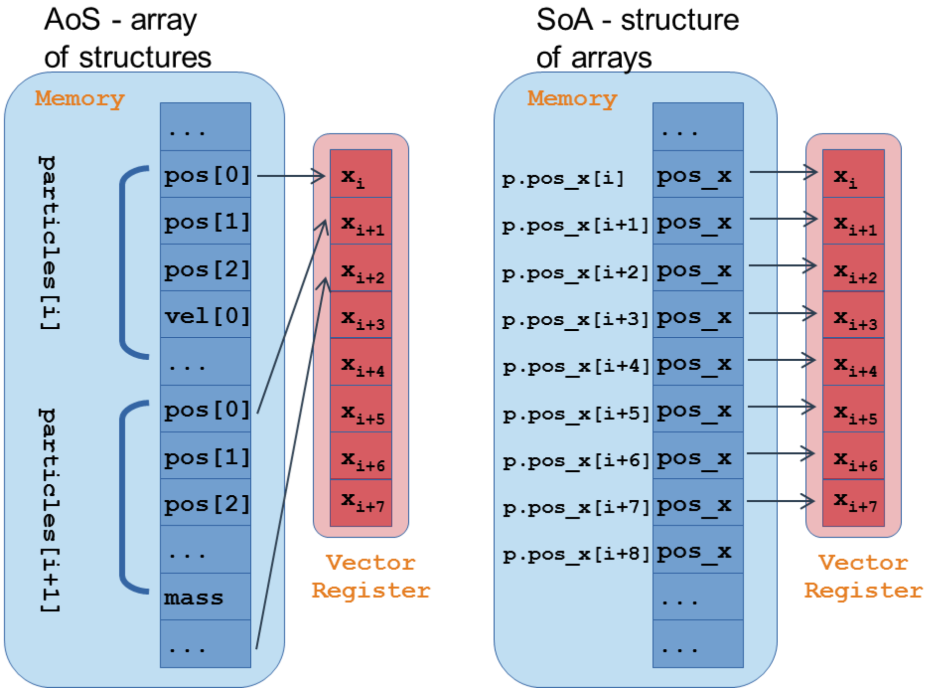


For B, 4 cache line loads compute 4 DP with reconstructions, prefetching might not work

Non-unit stride load: AoS vs SoA

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

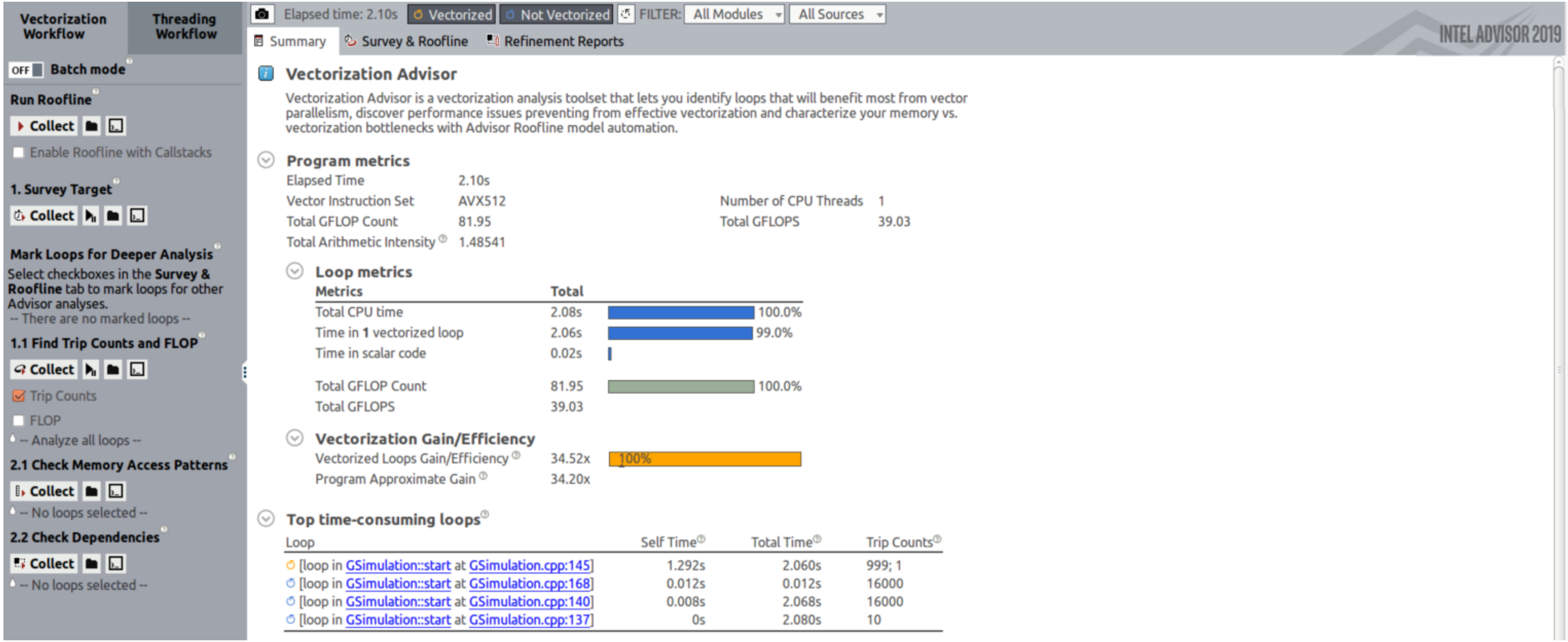
```
struct Particle
{
  public:
    ...
    real_type pos[3];
    real_type vel[3];
    real_type acc[3];
    real_type mass;
};

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;
};
```



AoS - array of structures

SoA - structure of arrays

Demo – nbody-sim/ver-soa: Report



Elapsed time: 2.10s

Select checkboxes in the **Survey & Roofline** tab to mark loops for other Advisor analyses.

-- There are no marked loops --

-- Analyze all loops --

-- No loops selected --

-- No loops selected --

Vectorization Advisor

Vectorization Advisor is a vectorization analysis toolset that lets you identify loops that will benefit most from vector parallelism, discover performance issues preventing from effective vectorization and characterize your memory vs. vectorization bottlenecks with Advisor Roofline model automation.

Program metrics

Elapsed Time	2.10s		
Vector Instruction Set	AVX512	Number of CPU Threads	1
Total GFLOP Count	81.95	Total GFLOPS	39.03
Total Arithmetic Intensity	1.48541		

Loop metrics

Metrics	Total	
Total CPU time	2.08s	100.0%
Time in **1** vectorized loop	2.06s	99.0%
Time in scalar code	0.02s	
Total GFLOP Count	81.95	100.0%
Total GFLOPS	39.03	

Vectorization Gain/Efficiency

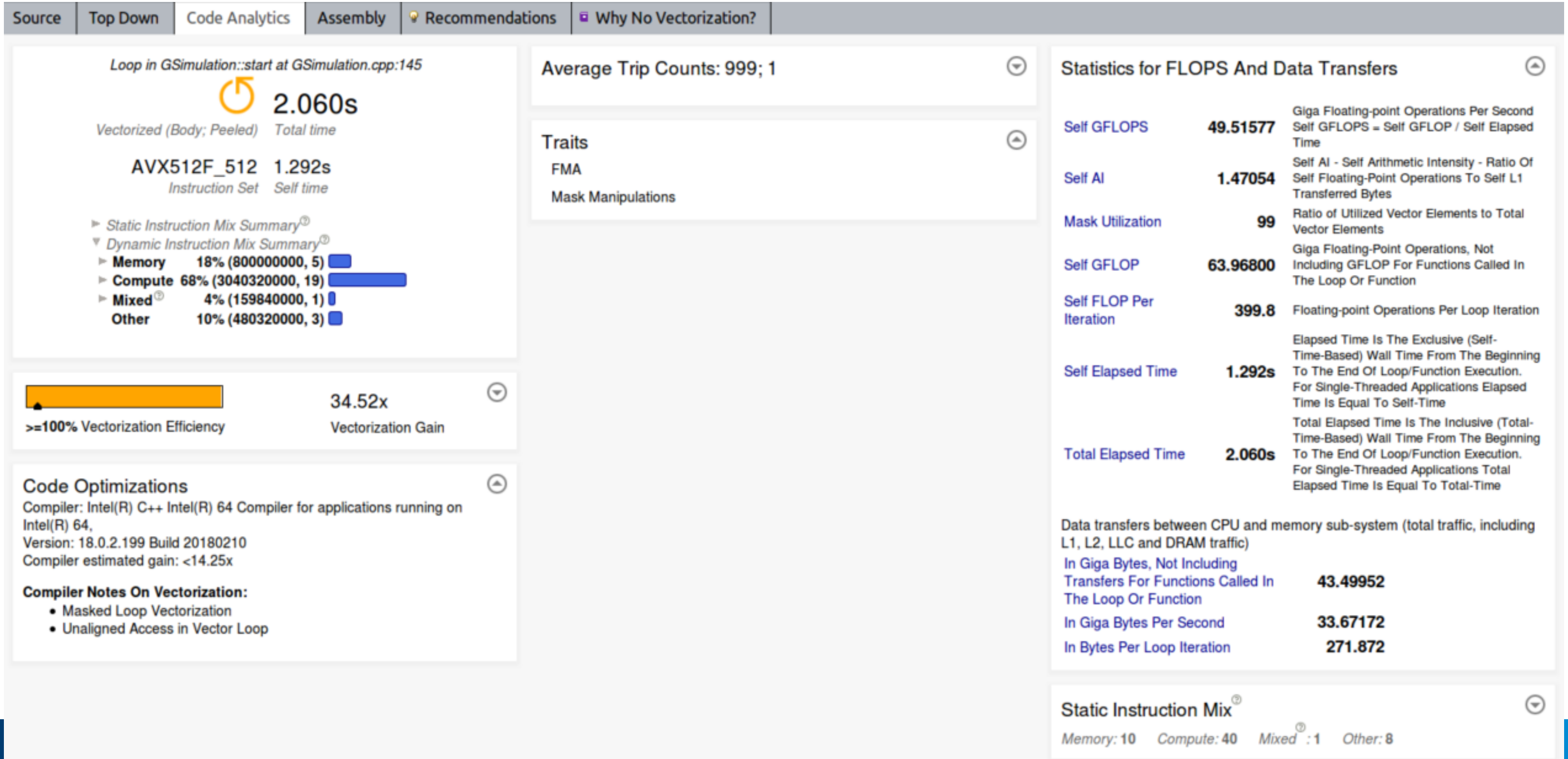
Vectorized Loops Gain/Efficiency	34.52x	100%
Program Approximate Gain	34.20x	

Top time-consuming loops

Loop	Self Time	Total Time	Trip Counts
[loop in GSimulation::start at GSimulation.cpp:145]	1.292s	2.060s	999; 1
[loop in GSimulation::start at GSimulation.cpp:168]	0.012s	0.012s	16000
[loop in GSimulation::start at GSimulation.cpp:140]	0.008s	2.068s	16000
[loop in GSimulation::start at GSimulation.cpp:137]	0s	2.080s	10

Optimization Notice

Demo – nbody-sim/ver-soa: Code Analytics



Loop in GSimulation::start at GSimulation.cpp:145

2.060s

Vectorized (Body; Peeled) *Total time*

AVX512F_512 1.292s

Instruction Set *Self time*

Static Instruction Mix Summary

Dynamic Instruction Mix Summary

Memory	18% (800000000, 5)
Compute	68% (3040320000, 19)
Mixed	4% (159840000, 1)
Other	10% (480320000, 3)

>=100% Vectorization Efficiency

34.52x

Vectorization Gain

Code Optimizations

Compiler: Intel(R) C++ Intel(R) 64 Compiler for applications running on Intel(R) 64,
Version: 18.0.2.199 Build 20180210
Compiler estimated gain: <14.25x

Compiler Notes On Vectorization:

- Masked Loop Vectorization
- Unaligned Access in Vector Loop

Average Trip Counts: 999; 1

Traits

FMA

Mask Manipulations

Statistics for FLOPS And Data Transfers

Self GFLOPS	**49.51577**	Giga Floating-point Operations Per Second Self GFLOPS = Self GFLOP / Self Elapsed Time
Self AI	**1.47054**	Self AI - Self Arithmetic Intensity - Ratio Of Self Floating-Point Operations To Self L1 Transferred Bytes
Mask Utilization	**99**	Ratio of Utilized Vector Elements to Total Vector Elements
Self GFLOP	**63.96800**	Giga Floating-Point Operations, Not Including GFLOP For Functions Called In The Loop Or Function
Self FLOP Per Iteration	**399.8**	Floating-point Operations Per Loop Iteration
Self Elapsed Time	**1.292s**	Elapsed Time Is The Exclusive (Self-Time-Based) Wall Time From The Beginning To The End Of Loop/Function Execution. For Single-Threaded Applications Elapsed Time Is Equal To Self-Time
Total Elapsed Time	**2.060s**	Total Elapsed Time Is The Inclusive (Total-Time-Based) Wall Time From The Beginning To The End Of Loop/Function Execution. For Single-Threaded Applications Total Elapsed Time Is Equal To Total-Time

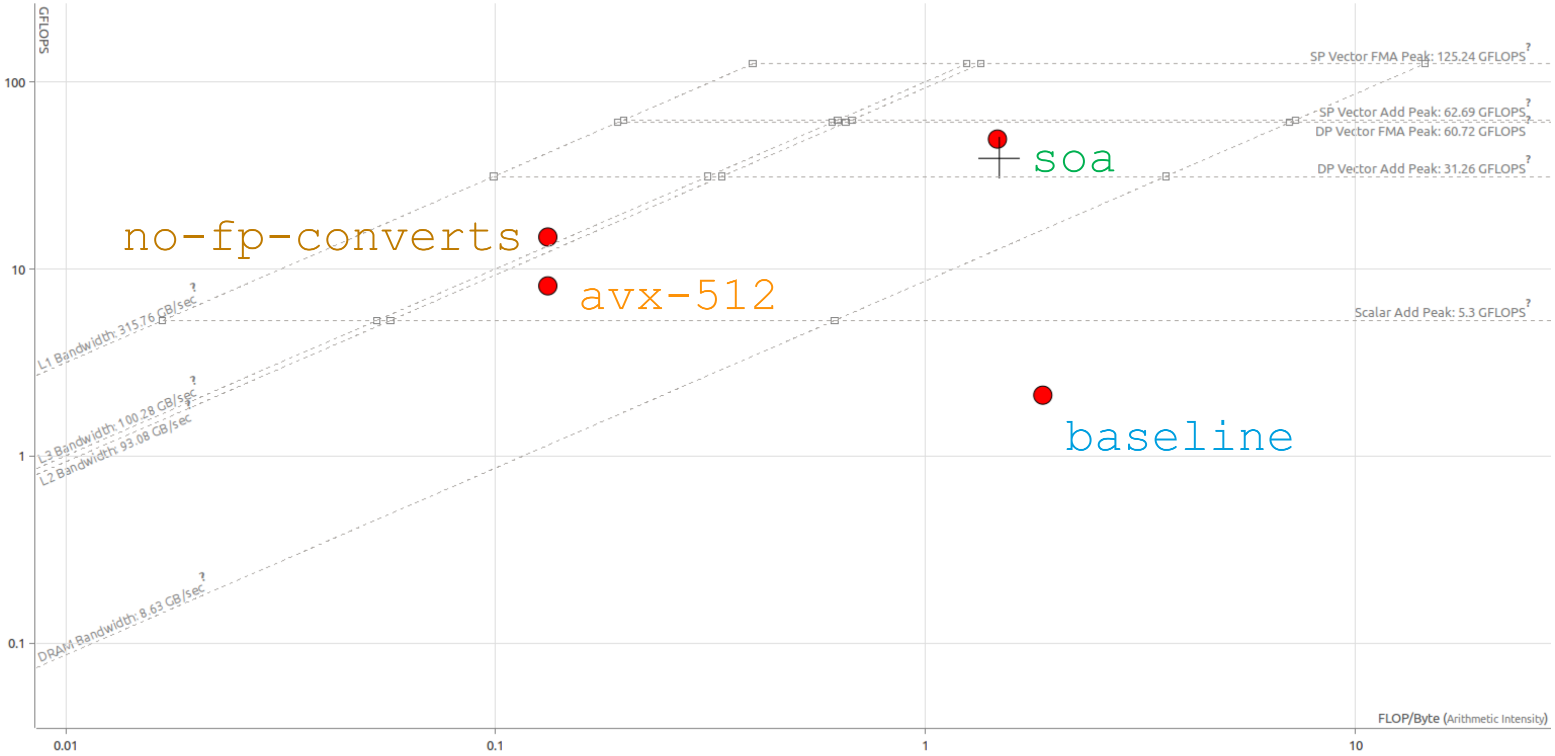
Data transfers between CPU and memory sub-system (total traffic, including L1, L2, LLC and DRAM traffic)

In Giga Bytes, Not Including Transfers For Functions Called In The Loop Or Function	**43.49952**
In Giga Bytes Per Second	**33.67172**
In Bytes Per Loop Iteration	**271.872**

Static Instruction Mix

Memory: **10** *Compute:* **40** *Mixed:* **1** *Other:* **8**

Demo – Roofline Comparison



Optimization Notice

Performance comparison

- Precision of constant and variables: consistent use of single and double precision

Optimization Options	Performance
Baseline	1.3 GFs
-O2 -xcore-avx512 –qopt-zmm-usage=high	9.0 GFs
No FP converts	21.1 GFs
Data-layout optimization (100% vec. eff.)	37.7 GFs
Memory alignment	47.7 GFs

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. The results above were obtained on Google* Cloud Platform, compute engine, n1-standard-2 (2 vCPUs, 7.5 GB memory), CPU platform Intel® Skylake, Zone us-east1-b, running Ubuntu 16.4 and using the Intel® C++ Compiler version 19.

Q&A

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Copyright © 2018, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

intel®