www.locuz.com

Optimize Vectorization using Intel Advisor

Presented By: Mandeep Kumar

Converge to the Cloud

Scenario

- This recipe focuses on a step-by-step approach to vectorize a real-time 3D cardiac electrophysiology simulation application on an Intel® Xeon® platform using Intel® Advisor.
- This recipe describes how to use the Intel Advisor to analyze the application performance for vectorization.
- Use Intel Advisor recommendations to make changes to the source code iteratively and improve the application performance by 2.6x compared to the baseline result.

Ingredients

This section lists the hardware and software used to produce the specific result shown in this recipe:

- **Performance analysis tools:** Intel® Advisor 2020
- **Application:** Cardiac_demo sample application, available from GitHub at https://github.com/CardiacDemo/Cardiac_demo.git
- **Compiler:** Intel® C++ Compiler 2020
- **Other tools:** Intel® MPI Library 2019
- **Operating system:** Linux
- **CPU:** Intel(R) Xeon(R) Platinum 8260L

Prerequisites: Set Up Environment

1. Set environment variables for Intel C++ Compiler, Intel MPI Library, and Intel Advisor:

```
$ source <compiler-install-dir>/linux/bin/compilervars.sh intel64
$ source <mpi-install-dir>/intel64/bin/mpivars.sh
$ source <advisor-install-dir>/advixe-vars.sh
```

2. Verify that you set up the tools correctly:

```
$ mpiicc -v
$ mpiexec -V
$ advixe-cl -version
```

   If all is set up correctly, you should see a version of each tool.

Build Application

1. Clone the application GitHub repository to your local system:

```
git clone https://github.com/CardiacDemo/Cardiac_demo.git
```

2. In the root level of the sample package, create a build directory and change to that directory:

```
mkdir build
cd build
```

3. Build the application using the following command:

```
mpiicpc ../heart_demo.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -g -o heart_demo -O3
-xCORE-AVX2 -std=c++11 -qopenmp -parallel-source-info=2
```

You should see the heart_demo executable in the current directory.

If you want to run the demo:

```
export OMP_NUM_THREADS=1
mpirun -n 48 ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 100 -i
```

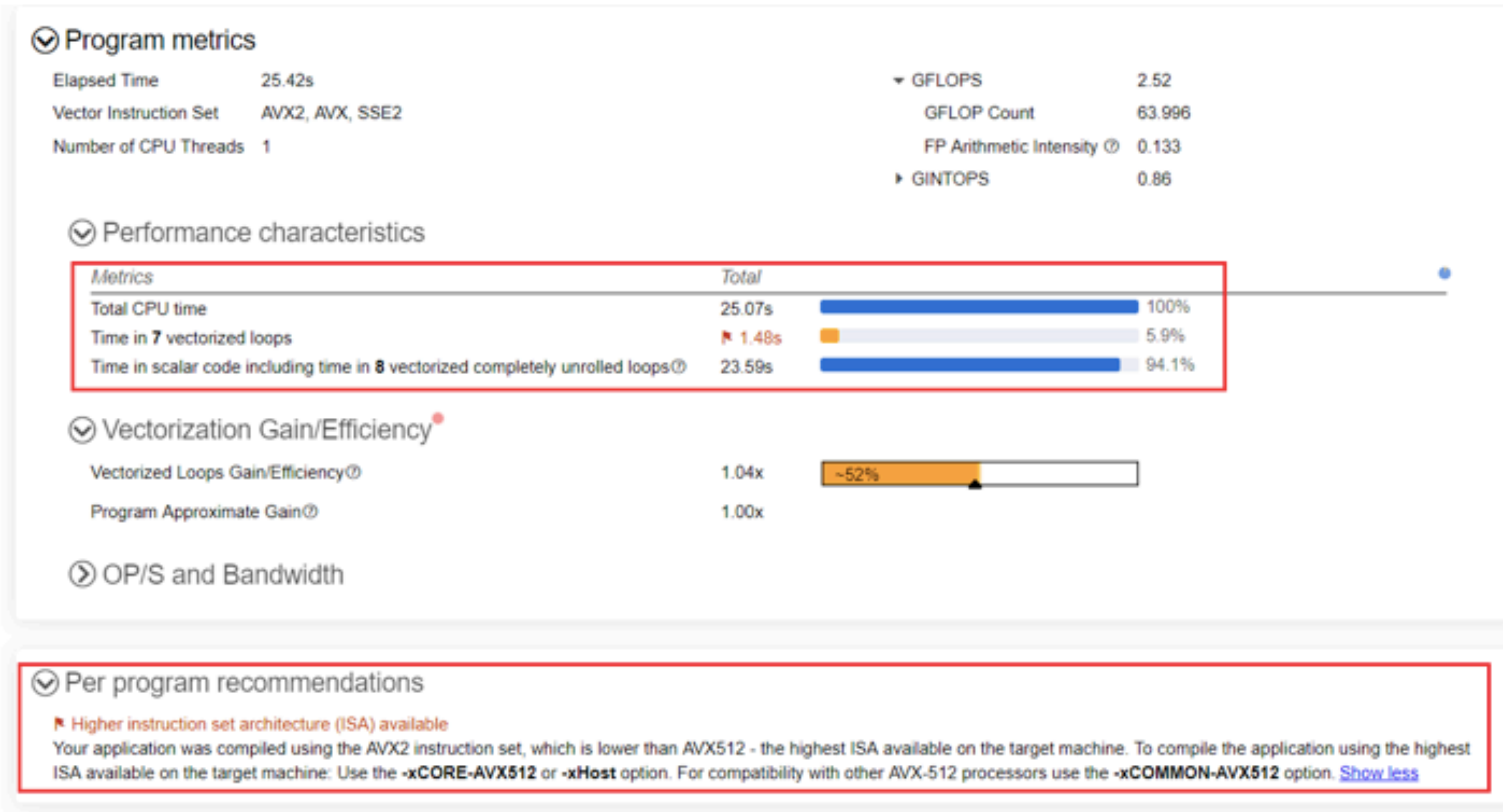
Establish a Baseline

1. Run the Survey, Trip Counts and FLOP analyses on the built heart_demo application and view the results in GUI.

```
mpiicpc ../heart_demo.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -g -o heart_demo -O3 -xCORE-AVX2
-std=c++11 -qopenmp -parallel-source-info=2

export OMP_NUM_THREADS=1

advisor --collect=survey --project-dir=./advi -- mpirun -np 48 ./heart_demo -m ../mesh_mid -s ../
setup_mid.txt -t 100 -i

advisor --collect=tripcounts --project-dir=./advi --flop --stacks --enable-data-transfer-analysis --
mpirun -np 48 ./heart_demo -m ../mesh_mid -s ../setup_mid.txt -t 100 -i

advisor --collect=map --project-dir=./advi --select=has-issue -- mpirun -np 48 ./heart_demo -m ../
mesh_mid -s ../setup_mid.txt -t 100 -i
```

Establish a Baseline

2. Go the **Summary** report and review the metrics:



Establish a Baseline

- Only 6% of the total time is spent in vectorized loops, and the remaining 94% of total time is spent in scalar code.
- The **Per Program Recommendation** suggests using Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instruction set architecture (ISA), which is the highest ISA available on the target machine.
- The heart_demo application runs and completes in **22.7** seconds.

NOTE: Time reported in the Summary may include the overhead added by Intel Advisor analyses.

Use the Highest Instruction Set Architecture Available

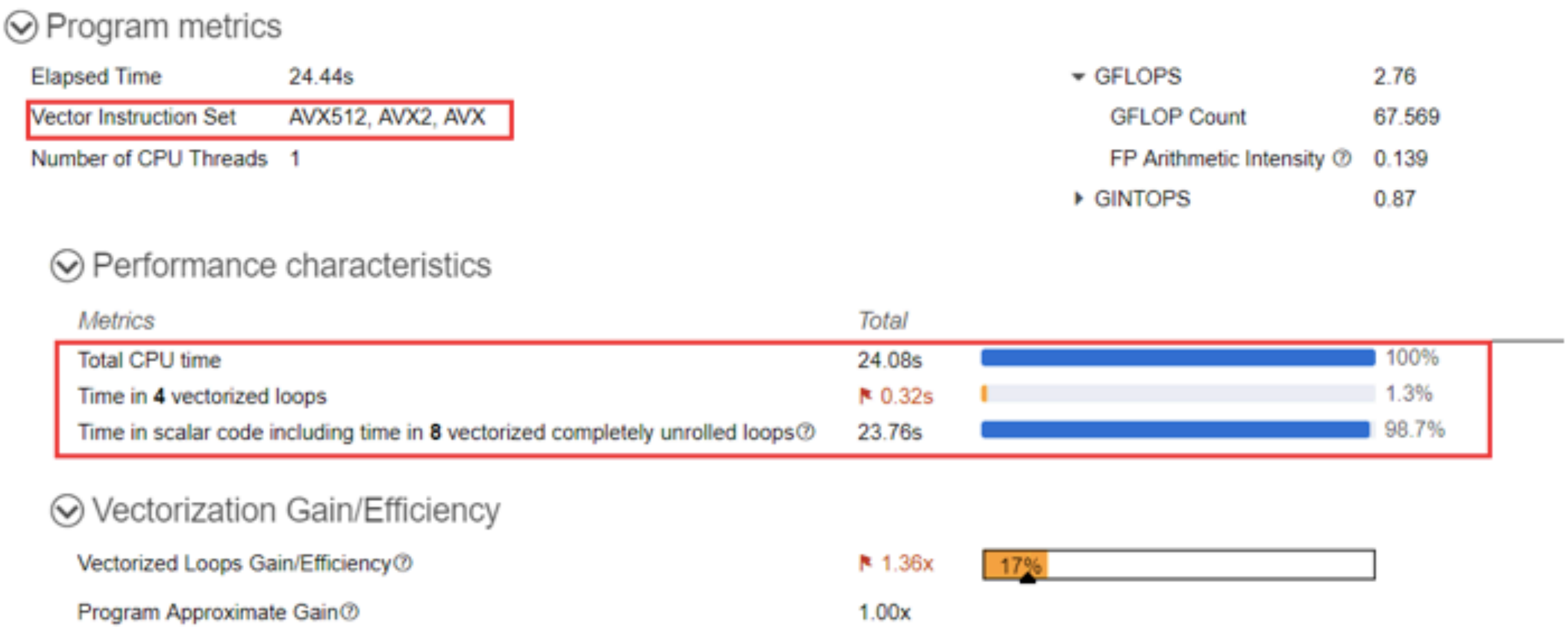
As the Intel Advisor recommends, use a higher ISA available of the machine to improve the overall application performance.

1. Add the -xCORE-AVX512 -qopt-zmm-usage=high option to the build command to use Intel AVX-512 and rebuild the heart_demo:

```
mpiicpc ../heart_demo.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -g -o heart_demo -O3 -xCORE-AVX512 -qopt-zmm-usage=high -std=c++11 -qopenmp -parallel-source-info=2
```

Re-run the Survey

2. Re-run the Survey, Trip Counts and FLOP analyses and go to **Summary** to view the result.



After optimization, the heart_demo application runs and completes in **21.2** seconds.

Survey & Roofline

With the Intel AVX-512 instructions, the elapsed time for the auto-vectorized loops is reduced compared to the baseline. For instance, in the **Survey & Roofline** tab, loop at heart_demo.cpp:278 now runs faster and takes 0.260 seconds compared to 1.310 seconds in the baseline.

Loop self time with the default ISA:

Summary | Survey & Roofline | Refinement Reports

Higher instruction set architecture (ISA) available

Consider recompiling your application using a higher ISA.

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Vectorized Loc	
		Total Time	Self Time		Vector ISA	Eff
[loop in Task::update_coupling_v2ompparallel@278 at	2 Ineffective peel...		1.310s	Vectorized Versions	AVX2	
[loop in Task::update_coupling_v2ompparallel@278	1 Possible ineffici...		0.430s	Vectorized (Body)	AVX2	
[loop in Task::update_coupling_v2ompparallel@278	1 Possible ineffici...		0.390s	Vectorized (Body)	AVX2	
[loop in Task::update_coupling_v2ompparallel@278	1 Possible ineffici...		0.370s	Vectorized (Body)	AVX2	
[loop in Task::update_coupling_v2ompparallel@278			0.060s	Remainder		
[loop in Task::update_coupling_v2ompparallel@278			0.040s	Remainder		
[loop in Task::update_coupling_v2ompparallel@278			0.020s	Remainder		

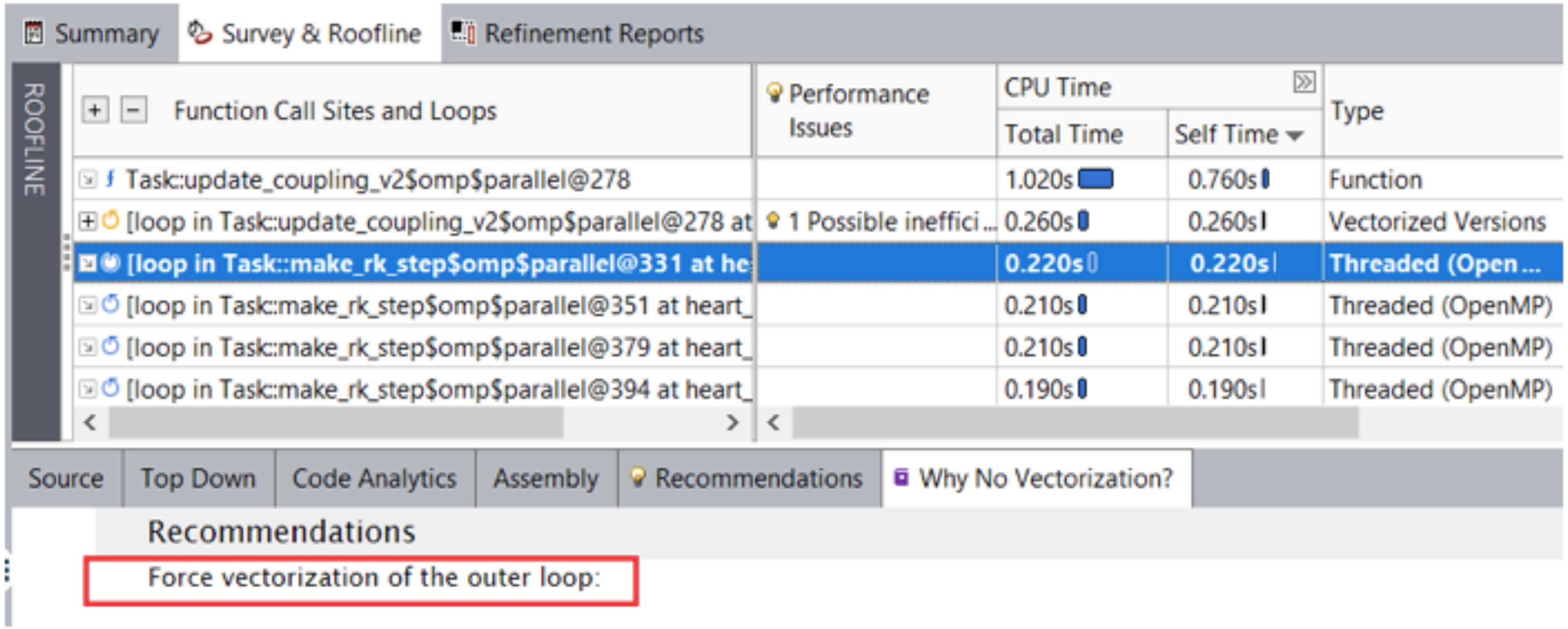
Loop self-time with Intel AVX-512:

Summary | Survey & Roofline | Refinement Reports

Function Call Sites and Loops	Performance Issues	CPU Time		Type	Vectorized
		Total Time	Self Time		Vector ISA
[loop in Task::update_coupling_v2ompparallel@278 at	1 Possible ineffici...		0.260s	Vectorized Versions	AVX512
[loop in Task::update_coupling_v2ompparallel@278	1 Possible ineffici...		0.170s	Vectorized (Body)	AVX512
[loop in Task::update_coupling_v2ompparallel@278	1 Possible ineffici...		0.060s	Vectorized (Body)	AVX512
[loop in Task::update_coupling_v2ompparallel@278	1 Possible ineffici...		0.030s	Vectorized (Body)	AVX512

Recommendation

To see Intel Advisor's recommendation on how to vectorize not-vectorized loops, navigate to the **Why No Vectorization** tab under **Survey & Roofline**. For the scalar loop at heart_demo.cpp:332, Intel Advisor recommends forcing vectorization of the outer loop.



Vectorize Outer Loop

As Intel Advisor recommends, force vectorization of the outer loop at line numbers 332, 352, 366, and 380 from heart_demo.cpp.

1. Before forcing vectorization of the outer loop, make sure that the loop is safe to vectorize by running the Dependencies analysis:
   a. Mark up loops at line numbers 332, 352, 366, and 380 from heart_demo.cpp.
   b. Run a Dependencies for the selected loops.
   c. Review the result in the following report tabs:
      - In the **Refinement Reports**, Intel Advisor finds no dependencies in the outer loop so it is safe to vectorize.
      - In the **Survey & Roofline** report tab, the Trip Count column shows that the number of trip counts for the outer loops at line numbers 332, 352, 366, and 380 from heart_demo.cpp is 588. Inner loops are completely unrolled and have the trip counts of 8 only.

Vectorize Outer Loop

2. Since the number of trip counts for outer loops is high and Intel Advisor found no dependencies in the outer loop, you can safely add **`#pragma omp parallel for simd`** clause to the outer loop for vectorization as shown below:

```
#pragma omp parallel for simd
for (int i=0; i<N; i++) {
    for (int j=0; j<DynamicalSystem::SYS_SIZE; j++)
        cnodes[i].cell.Y[j] = cnodes[i].state[j] + cnodes[i].rk4[0][j]/2.0;
    cnodes[i].cell.compute(time,cnodes[i].rk4[1]);
    for (int j=0; j<DynamicalSystem::SYS_SIZE; j++)
        cnodes[i].rk4[1][j] *= dt;
}
```

3. Rebuild the application, re-run the Survey and Trip Counts and FLOP analyses, and see the result in the **Survey & Roofline** report.

Vectorize Outer Loop

As you can see, execution time improves after vectorizing the outer loop. For example, the loop at heart_demo.cpp:332 now runs faster and takes 3.58 seconds compared to 4.2 seconds before the optimization.

Loop execution time before outer loop vectorization:

Summary | Survey & Roofline | Refinement Reports

Function Call Sites and Loops	CPU Time		Type	Vectorized Loops	
	Total Time	Self Time		Vector...	Efficiency
__libm_log_l9	0.630s	0.630s	Function		
[loop in Task:update_coupling_v2ompparallel@278 at heart_demo.cpp:282]	0.260s	0.260s	Vectorized Versions	AVX512	17%
[loop in Task:make_rk_stepompparallel@331 at heart_demo.cpp:332]	4.200s	0.220s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@351 at heart_demo.cpp:352]	4.540s	0.210s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@379 at heart_demo.cpp:380]	4.570s	0.210s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@394 at heart_demo.cpp:395]	0.190s	0.190s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@365 at heart_demo.cpp:366]	4.390s	0.190s	Threaded (OpenMP)		
[loop in Task:update_coupling_remotesompparallel@306 at heart_demo.cpp:307]	0.120s	0.100s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@306 at heart_demo.cpp:307]	0.090s	0.090s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@306 at heart_demo.cpp:307]	0.110s	0.080s	Threaded (OpenMP)		

Loop execution time after outer loop vectorization:

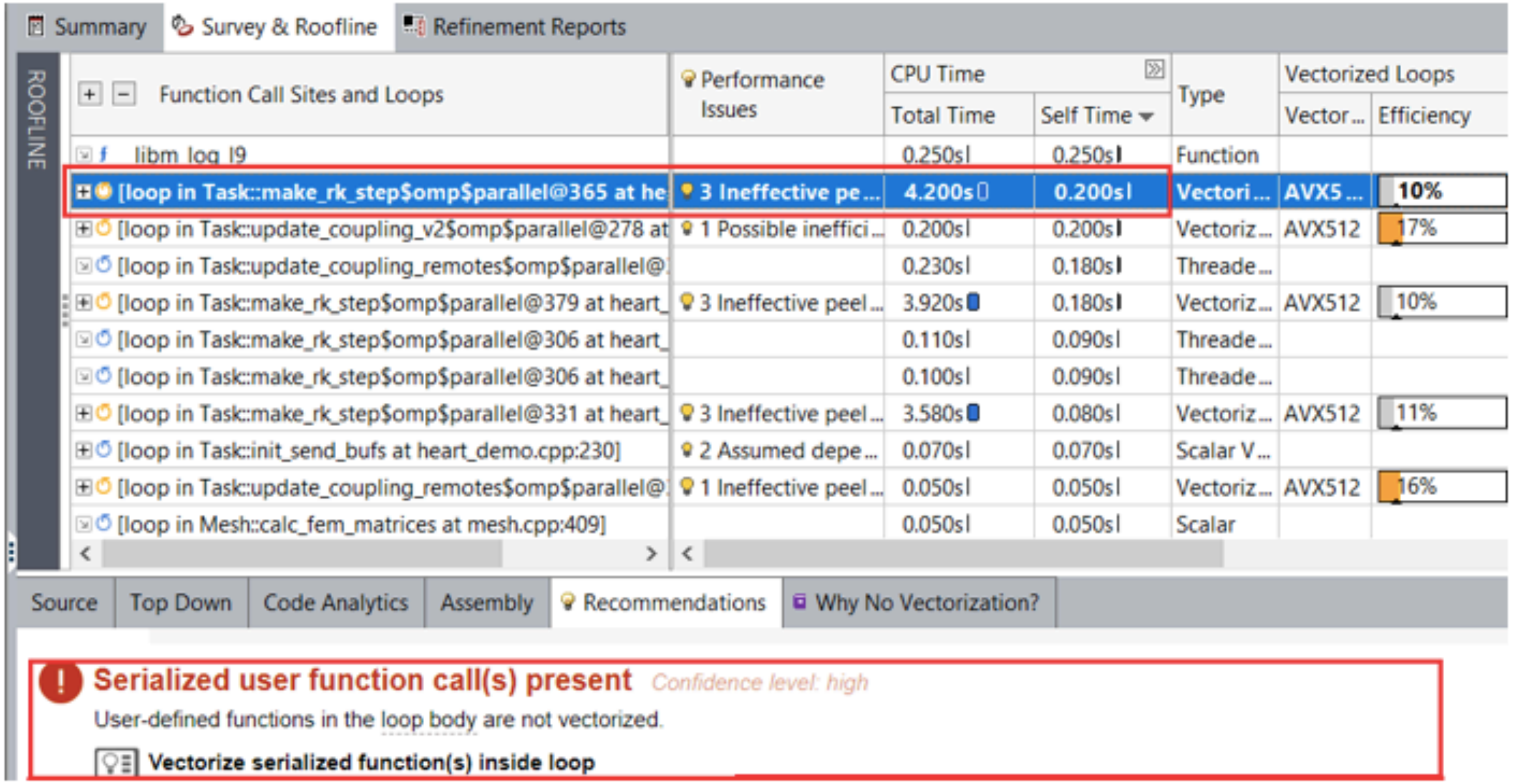
Summary | Survey & Roofline | Refinement Reports

Function Call Sites and Loops	CPU Time		Type	Vectorized Loops	
	Total Time	Self Time		Vector...	Efficiency
__libm_log_l9	0.250s	0.250s	Function		
[loop in Task:make_rk_stepompparallel@365 at heart_demo.cpp:366]	4.200s	0.200s	Vectorized+Thread...	AVX512	10%
[loop in Task:update_coupling_v2ompparallel@278 at heart_demo.cpp:282]	0.200s	0.200s	Vectorized Versions	AVX512	17%
[loop in Task:update_coupling_remotesompparallel@306 at heart_demo.cpp:307]	0.230s	0.180s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@379 at heart_demo.cpp:380]	3.920s	0.180s	Vectorized+Thread...	AVX512	10%
[loop in Task:make_rk_stepompparallel@306 at heart_demo.cpp:307]	0.110s	0.090s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@306 at heart_demo.cpp:307]	0.100s	0.090s	Threaded (OpenMP)		
[loop in Task:make_rk_stepompparallel@331 at heart_demo.cpp:332]	3.580s	0.080s	Vectorized+Thread...	AVX512	11%
[loop in Task:init_send_bufs at heart_demo.cpp:230]	0.070s	0.070s	Scalar Versions		
[loop in Task:update_coupling_remotesompparallel@306 at heart_demo.cpp:310]	0.050s	0.050s	Vectorized (Remai...	AVX512	16%
[loop in Mesh::calc_fem_matrices at mesh.cpp:409]	0.050s	0.050s	Scalar		
[loop in Task:make_rk_stepompparallel@351 at heart_demo.cpp:352]	3.370s	0.040s	Vectorized+Thread...	AVX512	10%

After optimization, the heart_demo application runs and completes in **18.5** seconds.

Vectorize Outer Loop

The efficiency of the vectorized outer loops is still low. Select the vectorized outer loop in the **Survey & Roofline** to see the recommendations for it. For example, the loop at heart_demo.cpp:366 has the Serialized user function call(s) present performance issue, and Intel Advisor suggests vectorizing serialized functions inside loop to fix it. Other vectorized outer loops at heart_demo.cpp:332, 352, and 380 have the same performance issue.



Use SIMD Function

Intel Advisor detects a serialized user function calls in the loop at heard_demo:366 and recommends to vectorize it.

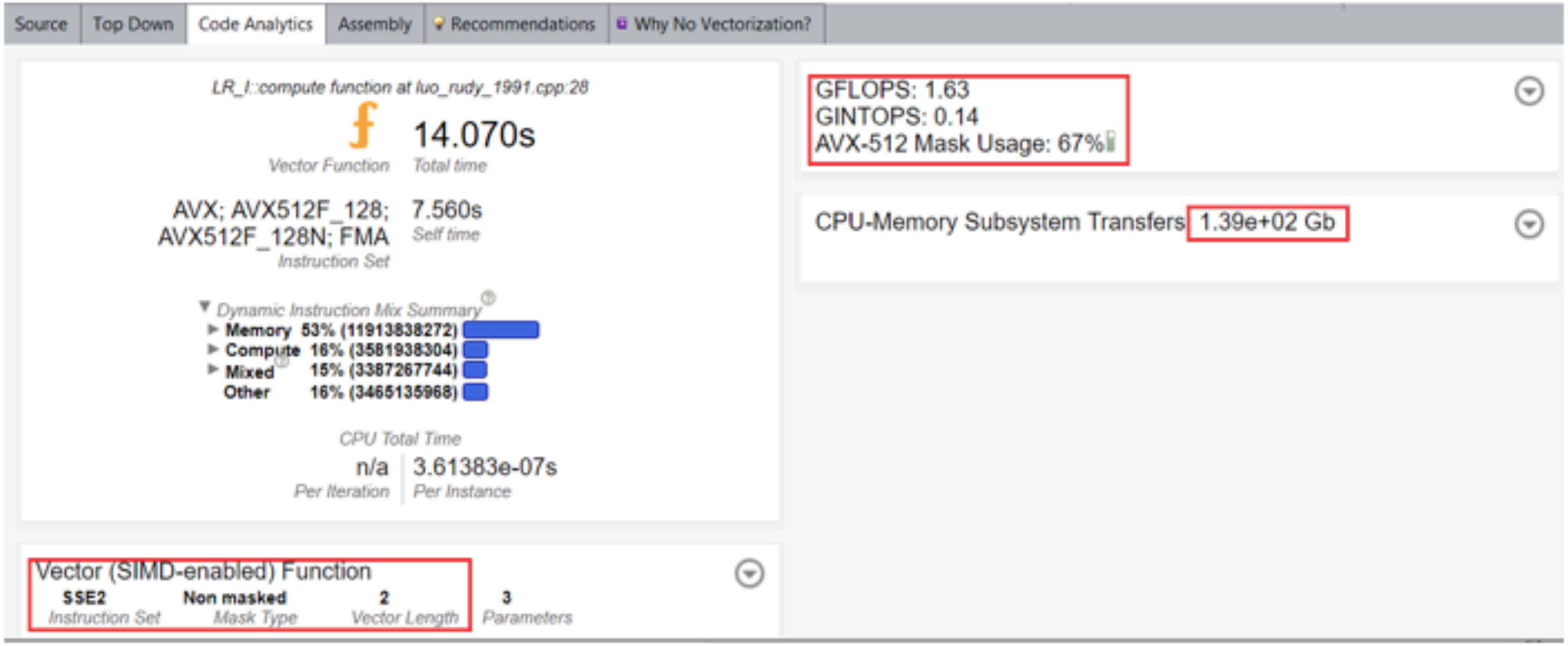
1. Add #pragma omp declare simd to the compute function in luo_rudy_1991.hpp:

```
#pragma omp declare simd
void compute (double time, double *out);
```

The DECLARE SIMD construct enables creating SIMD versions of a specified subroutine or function. These versions can be used to process multiple arguments from a single invocation in a SIMD loop.

Use SIMD Function

2. Rebuild the application, re-run the Survey and Trip Counts and FLOP analyses, and see the result in the **Survey & Roofline** report.



After optimization, the heart_demo application runs and completes in **16.4** seconds.

Optimize SIMD Usage

After vectorizing a function call, you can see in the Code Analytics that the vector length is 2, and Intel® Streaming SIMD Extensions 2 (Intel® SSE2) instruction set was used instead of vector length 8 and instruction set Intel AVX-512.

By default, the compiler uses the SSE instruction set. For better SIMD performance, use vecabi=cmdtarget compiler option or specify processor clause in vector function declaration.

To improve SIMD performance with the compiler option:

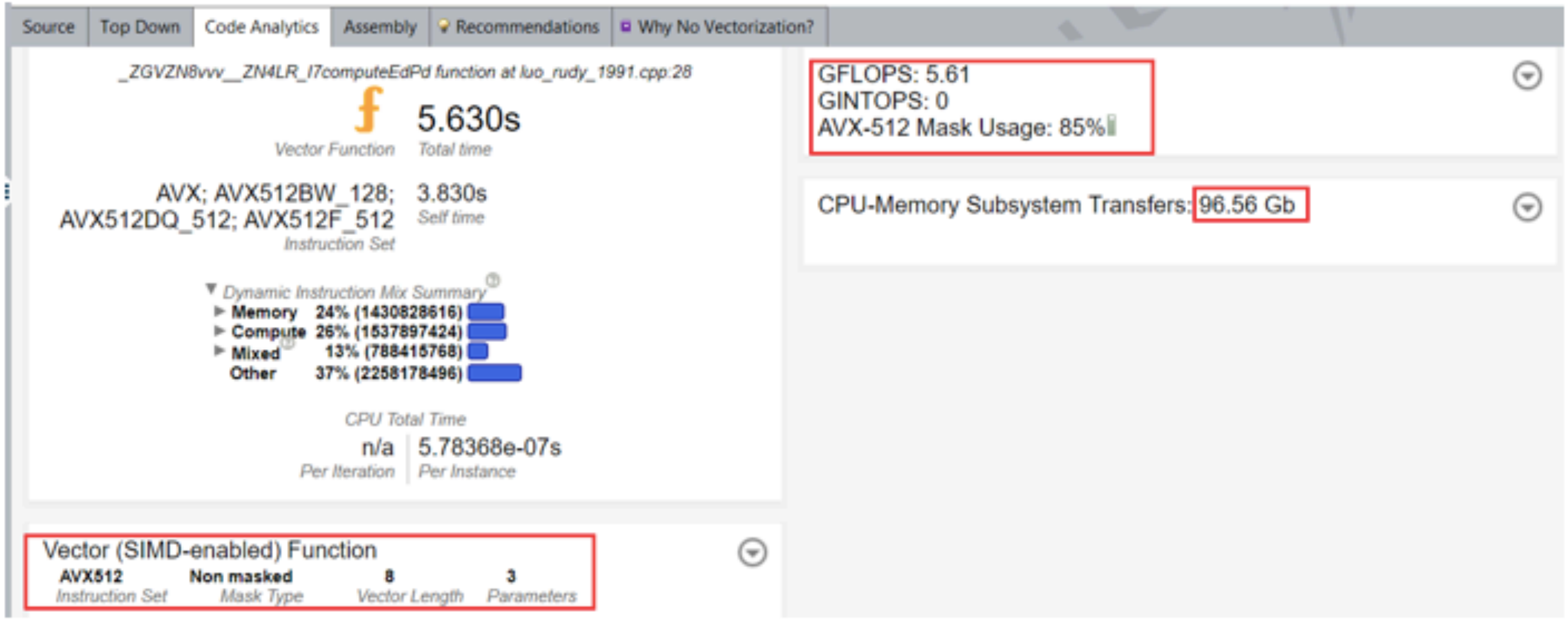
1. Add the -vecabi=cmdtarget option to the build command to generate Intel AVX-512 variant for the vectorized function and rebuild the heart_demo:

```
mpiicpc ../heart_demo.cpp ../luo_rudy_1991.cpp ../rcm.cpp ../mesh.cpp -g -o heart_demo -O3 -xCORE-AVX512 -qopt-zmm-usage=high -vecabi=cmdtarget -std=c++11 -qopenmp -parallel-source-info=2
```

Optimize SIMD Usage

2. Re-run the Survey, Trip Counts and FLOP analyses and go to **Survey & Roofline > Code Analytics** to view the result.

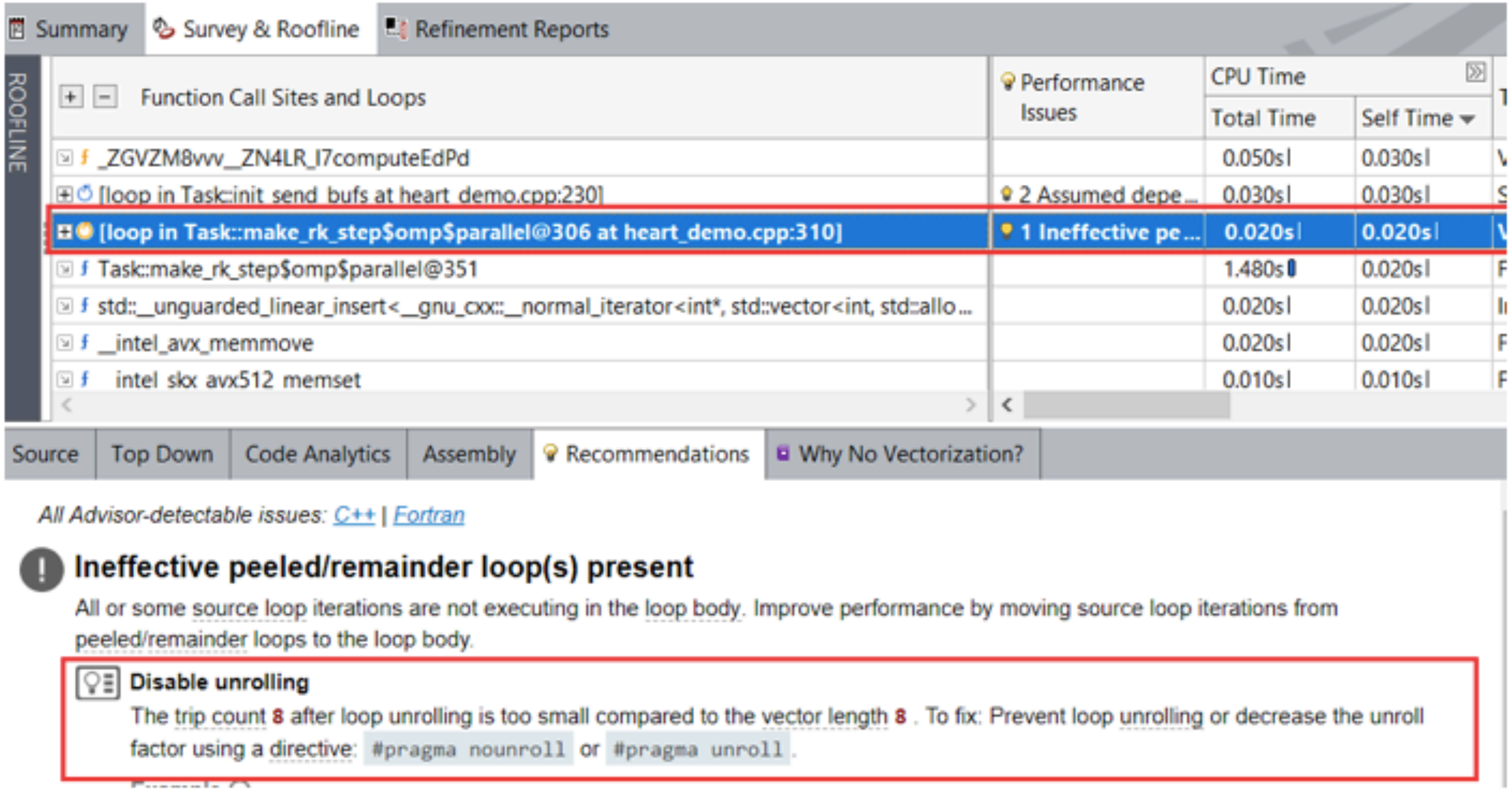
With this change, the vector length is 8 and the instruction set is Intel AVX-512.



After optimization, the heart_demo application runs and completes in **9.2** seconds.

Disable Unroll

For the vectorized inner loop at heart_demo.cpp:310, Intel Advisor also detects a vectorized remainder loop and recommends to *disable unrolling* for it.



Disable Unroll

To disable unrolling:

1. Add #pragma nounroll to the top of the inner loop at heart_demo.cpp:310, which was vectorized as a remainder loop.
   With this change, the loop is executed as a vectorized body with 100% FPU utilization compared to 36% masked utilization for the vectorization remainder.

2. Rebuild the application, rerun the Survey and Trip Counts and FLOP analyses, and see the result in the Survey & Roofline > Code Analytics report.

After optimization, the heart_demo application runs and completes in **8.9** seconds.

Performance Summary

- The optimization methods described here resulted in an overall 2.6x improvement from baseline on a single node.
- The applied optimizations can improve application performance as well:

# of Nodes / # of Ranks	Baseline Time	Optimized Time	Time Improvement
1/48	22.7 seconds	8.9 seconds	2.55x
2/96	11.5 seconds	5.15 seconds	2.23x
4/192	6.6 seconds	3.5 seconds	1.88x

Roofline Compare

- You can compare the baseline and optimized results with the Roofline Compare feature and the visualize performance improvement between loops.

- The image below compares the performance of the application before and after optimization. The Roofline chart highlights the performance difference for the loop at heart_demo.cpp:352 before and after optimizations showing the performance improvement of 80.82% with respect to FLOPS.

Key Take-Aways

To help you efficiently vectorize loops/functions of your application, Intel Advisor provides:

- Recommendations that you can follow to improve performance
- Insights into actual performance of the application against hardware-imposed performance ceilings by plotting a Roofline chart
- Deeper analyses like Dependencies and Memory Access Patterns to gain insights into data dependencies and memory accesses in a loop

By following Intel Advisor recommendations, you improved performance of the real-time 3D cardiac electrophysiology simulation heart_demo application by 2.55x compared to the baseline results.

Thanks!