Lab Instructions

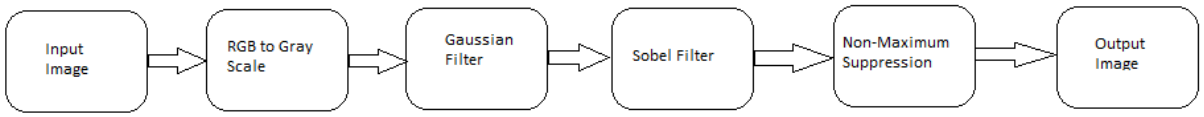
Pre-requisites:

1. Intel® Parallel Studio Professional or Ultimate Edition Installed on Linux machines (Provides Intel® C++ Compiler, Intel® Vtune Amplifier, Intel® Advisor which we will use in this lab).
2. Install OpenCV latest version:
    a. Download the source from github (https://github.com/opencv/opencv) using git clone command.
    b. Build OpenCV libraries using instructions documented at http://docs.opencv.org/trunk/d7/d9f/tutorial_linux_install.html.
3. Make sure that you have a copy of the source code for your lab which includes the lab documentation.

Introduction:

This lab will help you understand how to use Intel® Vtune Amplifier and Intel® Advisor to look for tuning opportunities and tune the code by enabling threading (using OpenMP or Intel® Threading Building Blocks [Intel® TBB]) and enabling vectorization (using OpenMP 4.0 SIMD constructs).

The source code which we are using here for the lab does the following:



Initial Steps:

Make sure the environment is ready to use Intel Software Development Tools:

a. For Intel® C++ Compiler and libraries shipped with it:

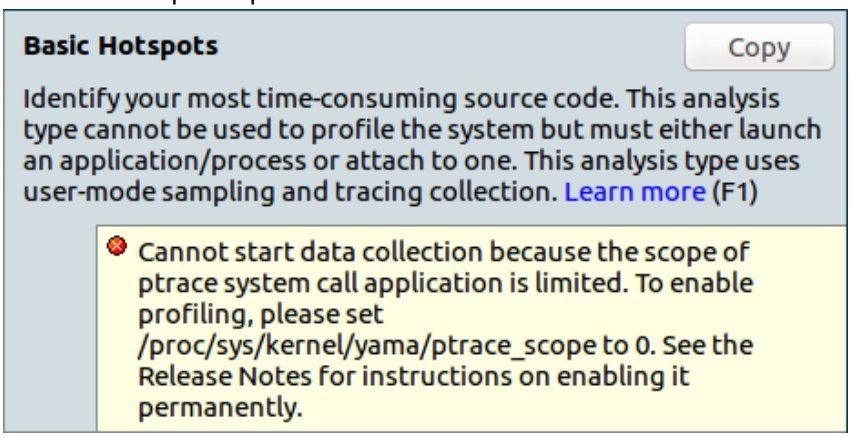
```
mpanoop84@mpanoop84: ~
mpanoop84@mpanoop84:~$ icpc
No command 'icpc' found, did you mean:
 Command 'ixpc' from package 'libixp-dev' (universe)
 Command 'icp' from package 'renameutils' (universe)
 Command 'ifpc' from package 'fp-utils-3.0.0' (universe)
icpc: command not found
mpanoop84@mpanoop84:~$
```

$ source /opt/intel/compilers_and_libraries_2017.xx.xxx/linux/bin/compilervars.sh intel64

b. For Intel® Vtune Amplifier:

$ source /opt/intel/vtune_amplifier_2017_for_systems.x.x.xxxxxx/amplxe-vars.sh intel64

i. For Basic Hotspot Report:

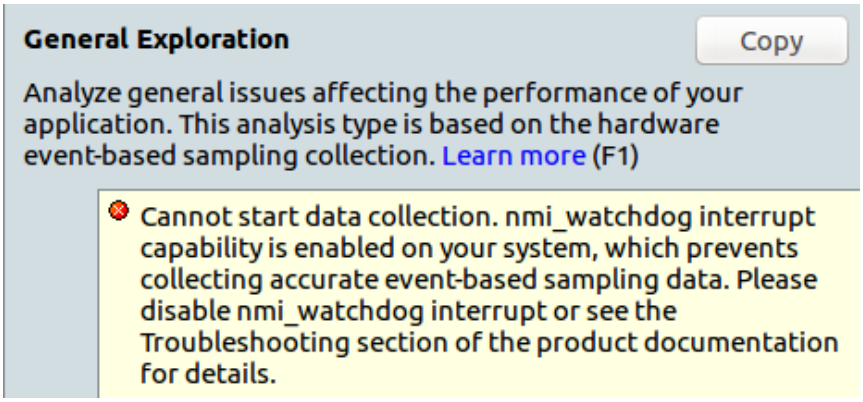


$ echo 0 > /proc/sys/kernel/yama/ptrace_scope
The above command should be executed in root privilege or sudo su mode.

Why set ptrace_scope to 0?

Yama is a security module in Linux and makes sure that Linux system is not compromised. Previously, as long as one process on linux is compromised, the user could get handle over all processes running to the machine. To avoid this potential security issue, ptrace_scope is introduced. For Vtune, we set this to 0 which means we can attach any application process which runs with the same uid as the Vtune process. So essentially we are overriding the security setting of linux for that session. More information available at https://www.kernel.org/doc/Documentation/security/Yama.txt

ii. For General Exploration Report:



$ echo 0 > /proc/sys/kernel/nmi_watchdog
The above command should be executed in root privilege or sudo su mode.

Why disable NMI Watchdog Interrupt?

NMI Watchdog interrupt is generated by the Application Programmable Interrupt Controller (APIC) on the CPU at regular interval (based on the clocking on APIC timer) to check if the CPU is hung or crashed. The NMI watchdog uses the resources that is reserved for Performance Monitoring and clobbers these resources periodically. The clobbering of the resource causes "occasional" glitches aka bad profiling runs. In order to prevent the clobbering of performance monitoring resources, the NMI watchdog timer needs to be disabled. This also leads to reproducible profiling runs.

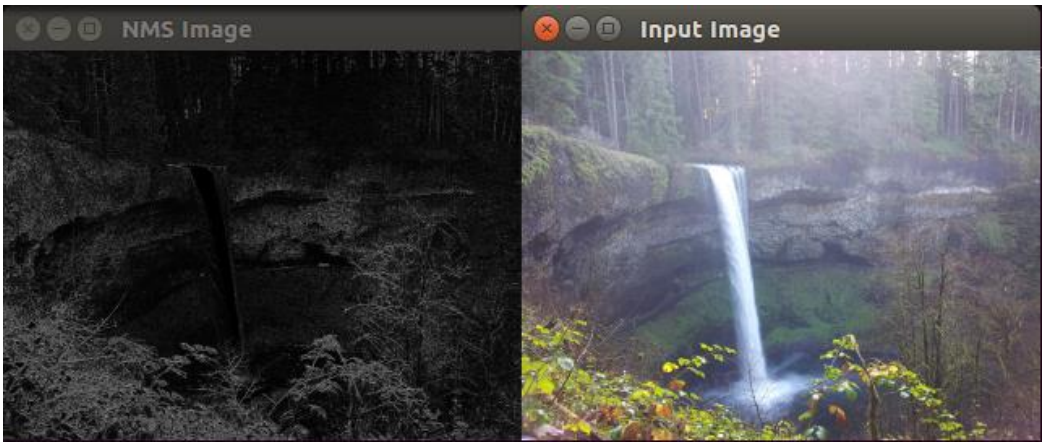
c. For Intel® Advisor:
   $ source /opt/intel/advisor_2018.x.x.xxxxxx/advixe-vars.sh intel64

Vanilla Implementation:

Go to Lab2/initial_step folder which has the sample code (sample.cc) and the Makefile to build this program.

Type in
$ make

The above command will build the program and produce the executable by name sample.
Type in
$ make run
To run the executable and it will automatically grab the input image (silverfalls.bmp) from res folder under Lab2 folder. The displays the processed image in a window along with the input image for comparison.



The build also produces an optimization report. Optimization reports generated by Intel Compiler provides us insights into the type of optimizations which the compiler does on the individual loops in the code. More information on Optimization report is available at https://software.intel.com/en-us/articles/getting-the-most-out-of-your-intel-compiler-with-the-new-optimization-reports. The

Makefile has –qopt-report5 –qopt-report-phase=vec compiler options which generates vectorization report with level 5 verbosity. By default the optimization report is dumped in a file which is <filename>.optrpt. In our case, it is sample.optrpt.

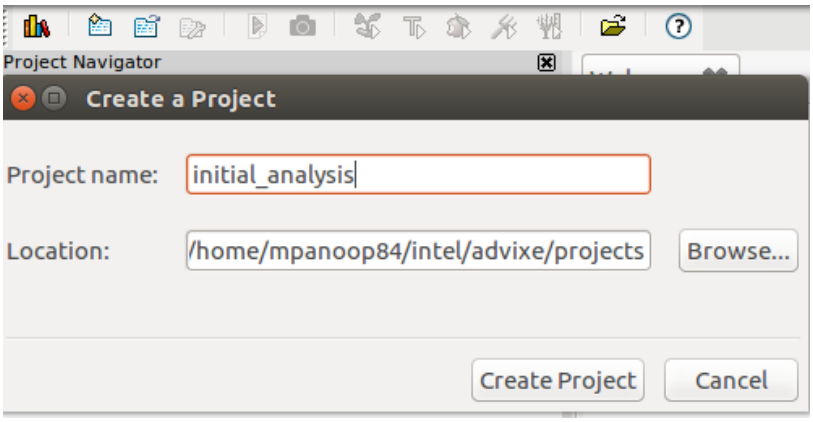
Open sample.optrpt and check the content using an editor. By default none of the loops are auto-vectorized as shown in the optimization report. But there are too many loops. Which one should I look at first? This is where the analysis tools help us give insight into the runtime performance information. We should first understand if the sample code is actually memory bound or compute bound. Intel® Advisor provides us with Roofline analysis which helps understand this in a pictorial manner.

Intel® Advisor Analysis:

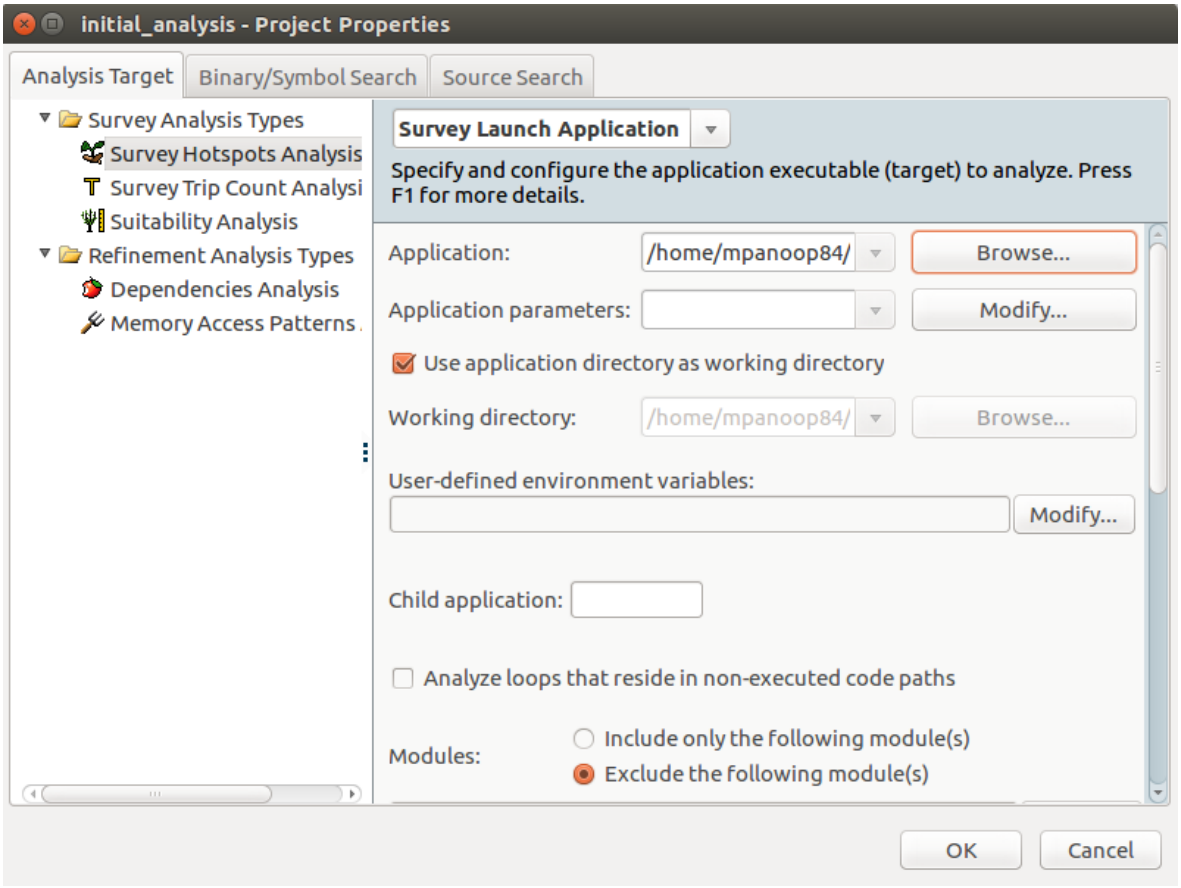
Intel® Advisor helps us understand if the sample program is memory bound or compute bound. An application is memory bound when the instructions are ready for execution but is waiting for the data from the memory. An application is compute bound when the instructions executing is not utilizing the full potential of the silicon power (be it in terms of vector compute units or the number of cores on the machine). Depending upon where the application lands up in, the kind of tuning the program demands is different. In order to determine this, first let's create an Advisor project. You can open the Advisor GUI by typing the following command on the terminal window:

$ advis-gui

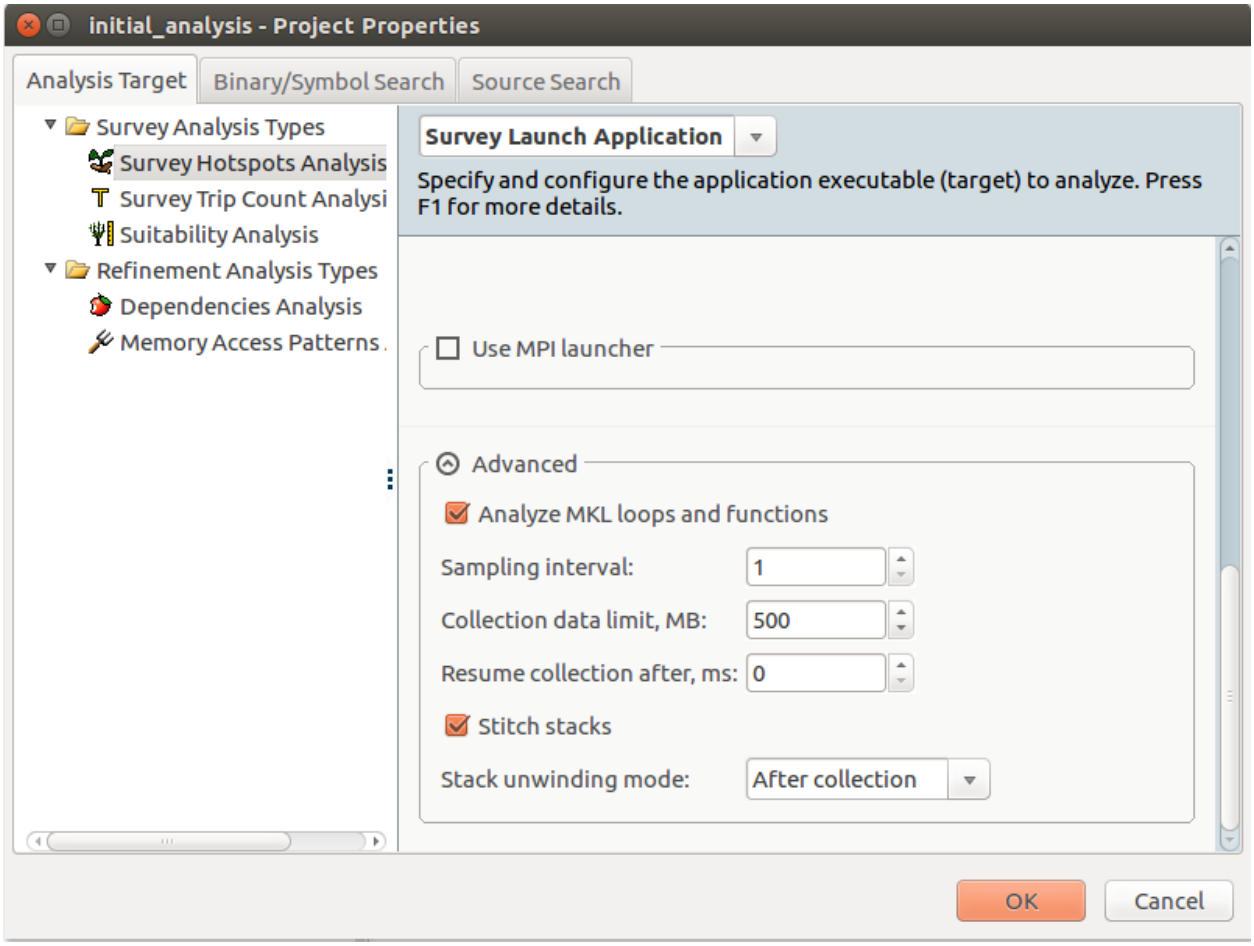
Click on the icon with Yellow + sign to create a new project. Specify the project name as shown below and click on "Create Project".



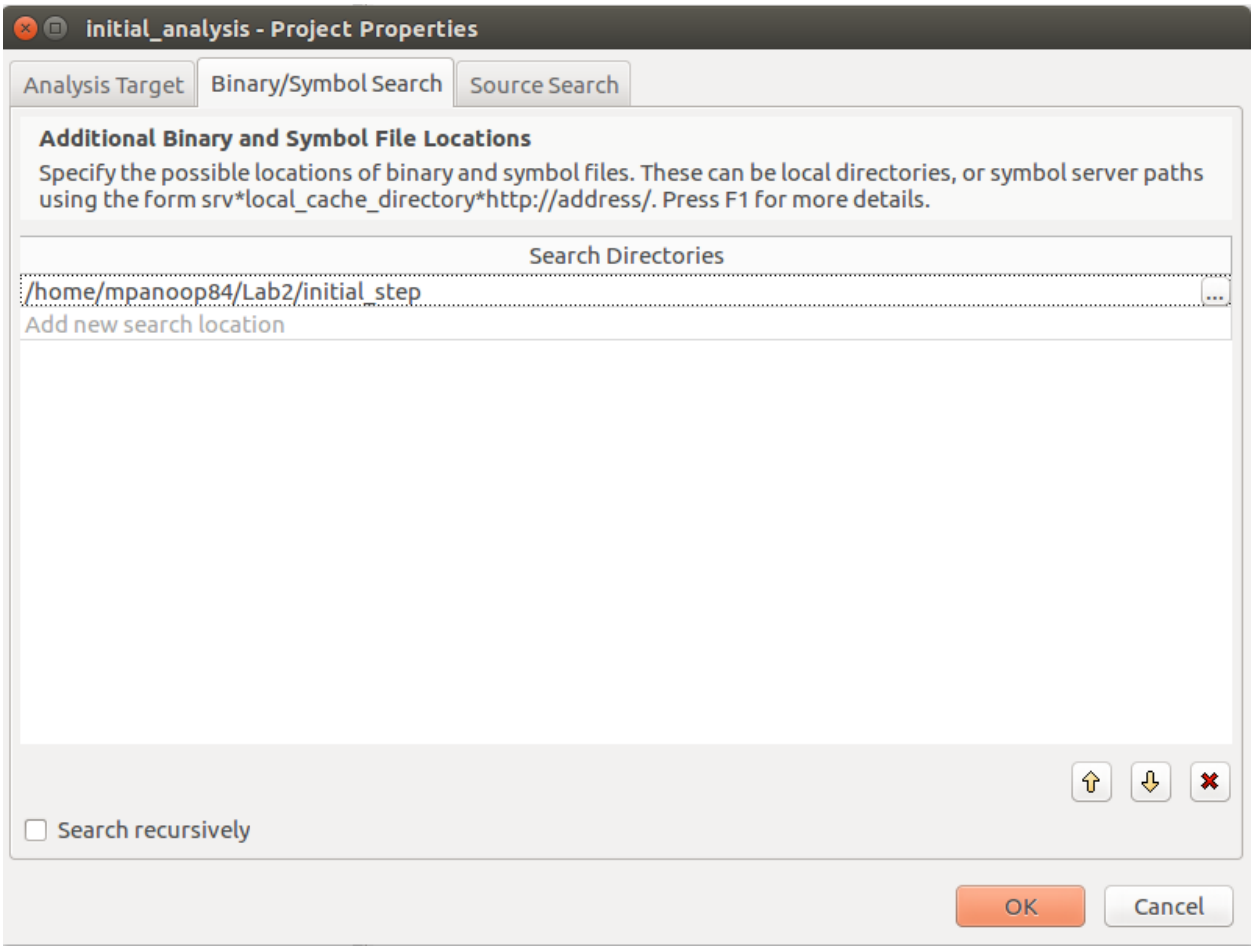
Now it will pop the below screen which asks for the application which needs to be surveyed. Select the application using the "Browse" button:



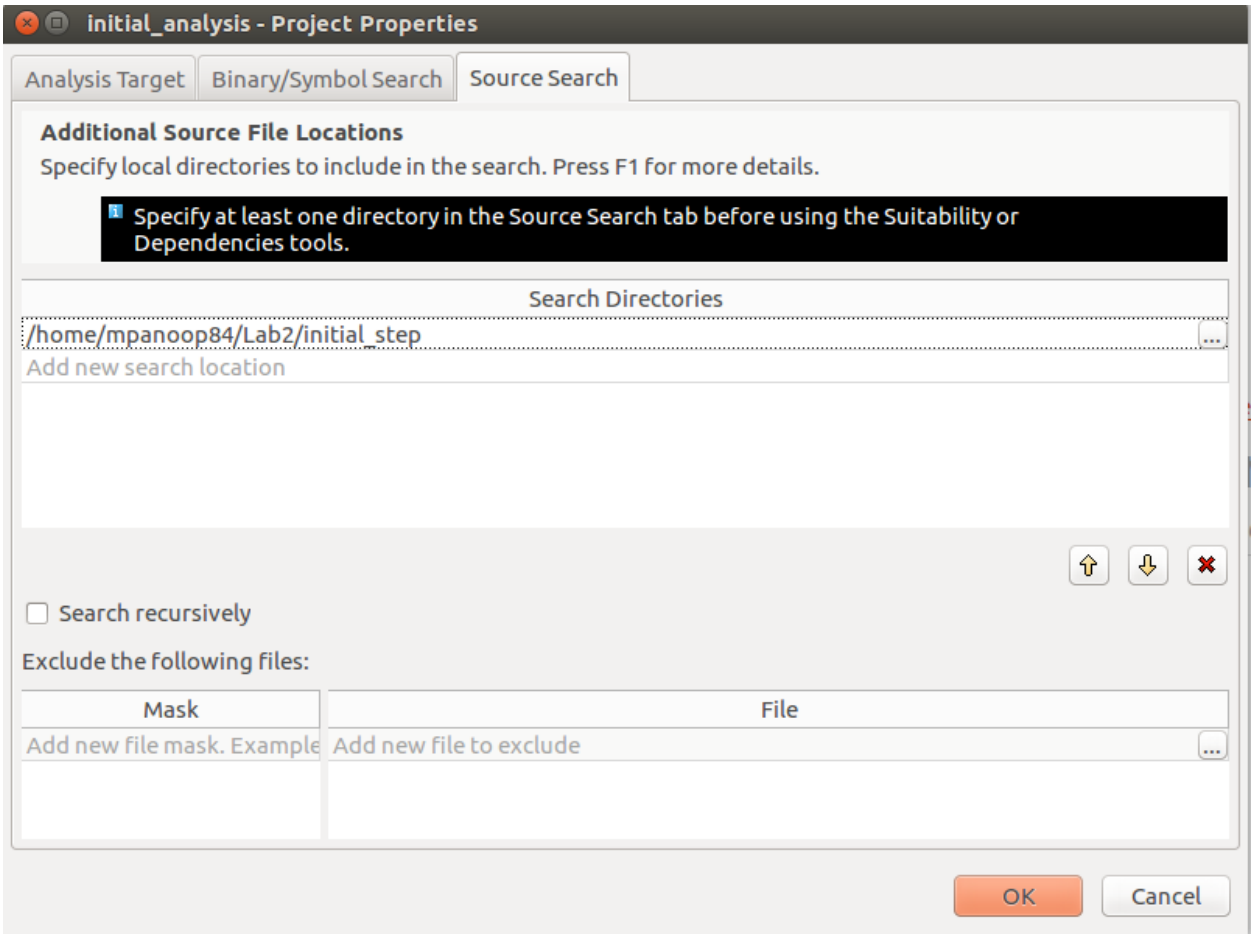
After selecting the application, scroll below to select the sampling interval. By default the sampling interval is 10 milliseconds. Since our application doesn't run that significant amount of time, it makes sense to reduce the sampling interval to 1 millisecond to get more samples. After reducing the sampling interval to 1ms as shown below, click on "OK".



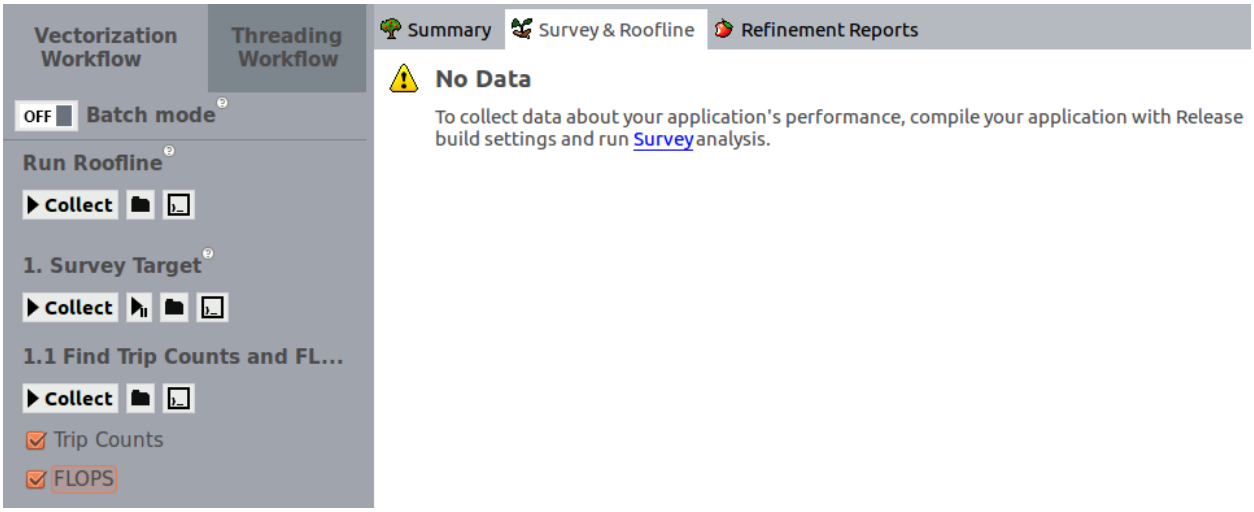
The next tab will ask for the directories which host the binary and debug information for the binaries and libraries:



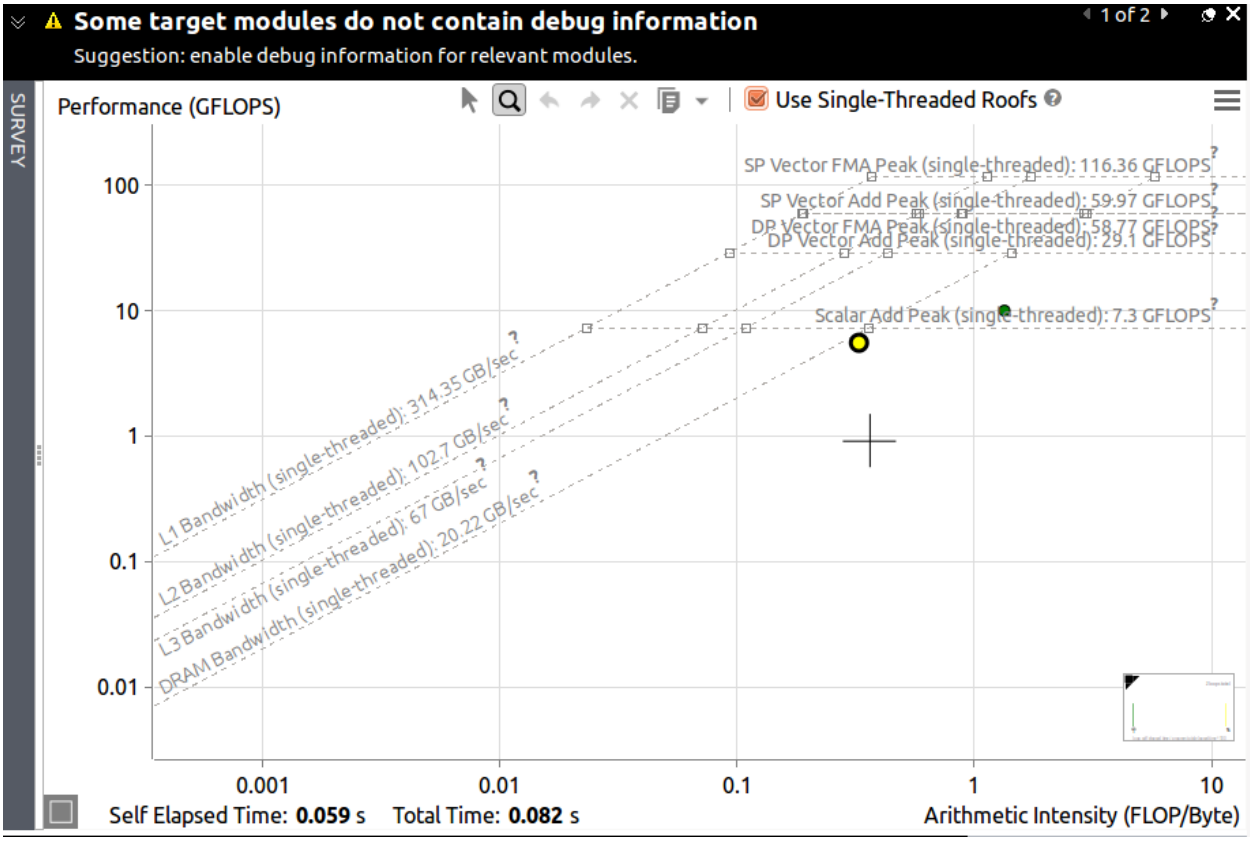
The last tab will ask for the directory which host the source code of the program. If you have your application source in a hierarchical manner across folders, you can specify the top folder in the hierarchy and click on “Search recursively” check box. Once the directories are selected, click on “OK” button.



You should see the below screen for the new project created. The left panel list all types of analysis which Advisor supports. We are currently on Vectorization workflow which will help us find opportunities for Vectorization. Select the FLOPs checkbox and click on “Collect” button right under Run Roofline. This will run the Roofline analysis and display the information on top hotspots of the program in a visual plot.



The Roofline analysis takes a few minutes to run for the following reason: It does both survey and FLOP analysis one after the other. The Visual plot looks like the one shown below with X-Axis for Arithmetic Intensity and Y-axis for Performance in GFLOPs. The kernels in the sample program will appear as colored circles in the plot. The bigger the circle, most time is spent on that kernel and is worth spending time to tune. The color of the circle determines how well the kernel utilizes the resources on the target silicon. Red color is least efficient, Yellow color suggest there are tuning opportunities and Green is relatively in good shape. The below roofline is cache aware and this information on the cache bandwidth as well as GFLOPs for SGEMM, DGEMM is got by running benchmarks when the analysis is triggered. So these number will vary from machine to machine. Since we are running a single threaded sample code as of now, we select the "Use Single-Threaded Roofs" checkbox. Ideally we want our kernels towards top right side of this plot meaning higher values for both Arithmetic intensity and GFLOPs. Arithmetic Intensity stands for how much compute is done per byte of data fetched from memory. This essentially is compute to memory access ratio. If we have memory bound kernel, the bandwidth will be the real bottleneck and this will result in the kernel being under the DRAM bandwidth capacity towards the left side of the plot. If the kernels are on the right side of the plot, then they are compute intensive. In general our final goal is push the kernels from bottom left to top right.



Hovering the mouse over the individual dots in the plot will list the kernel name and also the values for both L1 Arithmetic intensity and Performance in GFLOPs. To see the survey results, click on the "Survey" on the left side of the screen. The survey result will list the kernels in descending order of time taken by them w.r.t to overall execution time of the program. It also states if the individual kernels are vectorized

or not. If not vectorized, it mentions the reason why the vectorization failed. Intel® Advisor gets this information from Intel Compiler Optimization Report.

Initial Step:

$ make run

./sample

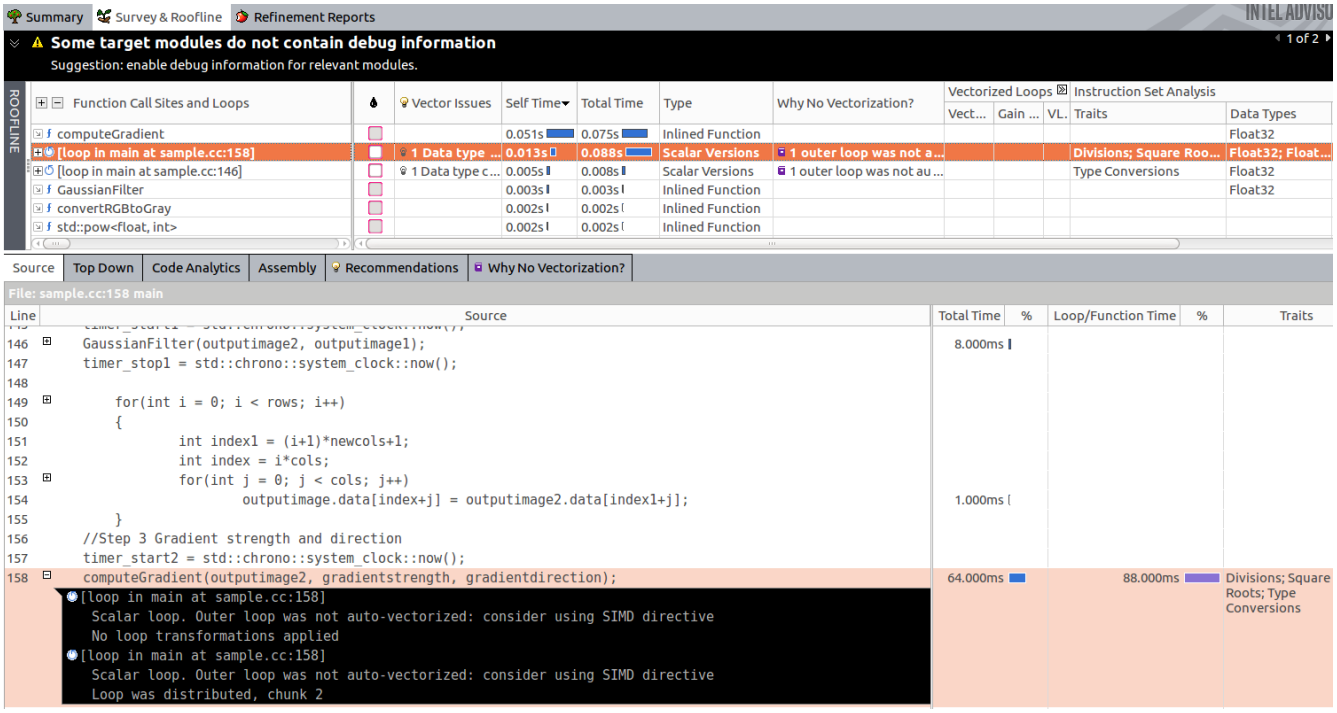
Time elapsed for 1 frame = 0.447226 seconds

Time taken by RGB to GrayScale = 0.00973719 seconds

Time taken by Gaussian Filter = 0.0315406 seconds

Time taken by ComputeGradient = 0.33964 seconds

Time taken by Non-Maximum Suppression = 0.0551606 seconds



As you can see from above Advisor Survey report, the computeGradient() called from line number 158 of sample.cc is the hottest loop in the sample program. Also this hot loop doesn't auto-vectorize using Intel Compiler. Advisor clearly states to consider the possibility of outer loop vectorization. Intel Compiler provides us with Optimization report which provides information on what optimizations are enabled by the compiler. Compiler option -qopt-report<n> is the option used to enable optimization

report with “n” being the verbosity of the optimization report (higher the value of n, more is the verbosity). Compiler also provides a means of filtering the optimization report based on different optimization phases like inlining, vectorization, openmp, profile-guided optimization etc. In this sample, we filter the optimization report for vectorization phase using -qopt-report-phase=vec. By default the optimization report is redirected to a file <sourcefilename.optrpt>. The report generated for computeGradient() is shown below:

LOOP BEGIN at sample.cc(51,2)

<Distributed chunk2>

remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at sample.cc(55,3)

remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at sample.cc(60,4)

remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

LOOP BEGIN at sample.cc(63,5)

remark #15389: vectorization support: reference gradientx[k1+1][k2+1] has unaligned access [sample.cc(65,20)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(65,44)]

remark #15389: vectorization support: reference gradienty[k1+1][k2+1] has unaligned access [sample.cc(66,20)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15381: vectorization support: unaligned access used inside loop body

remark #15335: loop was not vectorized: vectorization possible but seems inefficient. Use vector always directive or -vec-threshold0 to override

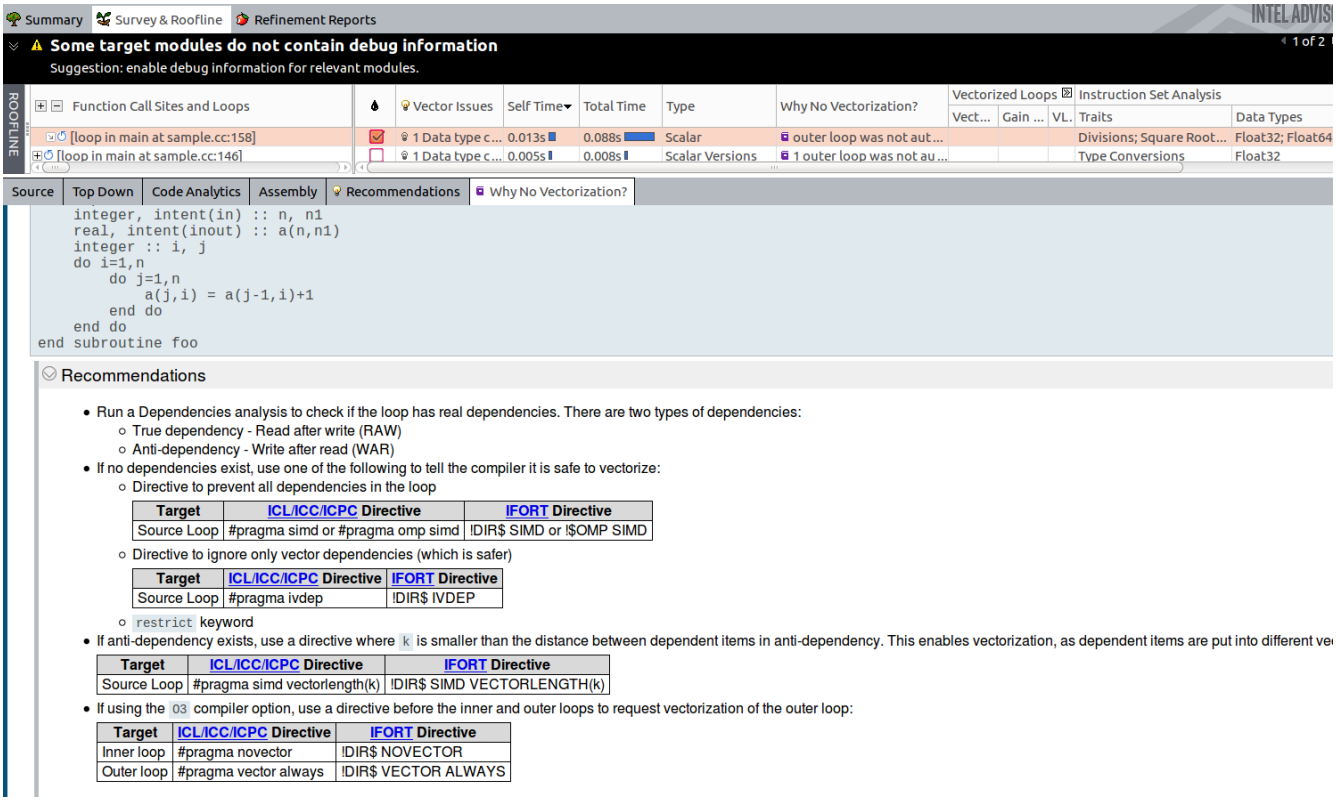
remark #15305: vectorization support: vector length 2

remark #15309: vectorization support: normalized vectorization overhead 2.273

remark #15450: unmasked unaligned unit stride loads: 4

```
      remark #15475: --- begin vector cost summary ---
      remark #15476: scalar cost: 19
      remark #15477: vector cost: 11.000
      remark #15478: estimated potential speedup: 0.620
      remark #15487: type converts: 2
      remark #15488: --- end vector cost summary ---
    LOOP END

    LOOP BEGIN at sample.cc(63,5)
    LOOP END

    LOOP BEGIN at sample.cc(63,5)
    LOOP END
  LOOP END
 LOOP END
LOOP END
```

Compiler's auto-vectorizer always tries to vectorize the inner most loop. But in this case, the trip count of the innermost loop is just 3. So the vectorization is not efficient. But we should consider the outer loop vectorization. There are 4 nested loops in this function and the second loop iterates along the one row. As a thumb rule for optimization on Intel Architecture, always target outermost loop for threading and inner loop for vectorization. Since the second loop with loop index variable "j" has more trip count (equal to number of columns in the image), this becomes a good candidate for vectorization. In order to vectorize outer loop, we can see Vector Advisor recommends:



Step1:

Please look at step1-vectorization folder for the sample program which enables vectorization. Following is the code change (marked in red font):

```
for(int i = 1; i < rows-1; i++)
{
    int index = i*cols+1;
    int gindex = (i-1)*gcols;
    #pragma omp simd private(gradient_x, gradient_y, degrees)
    for(int j = 0; j < cols-2; j++)
    {
        int index1 = index+j;
        int gindex1 = gindex+j;
        gradient_x = gradient_y = 0.0f;
```

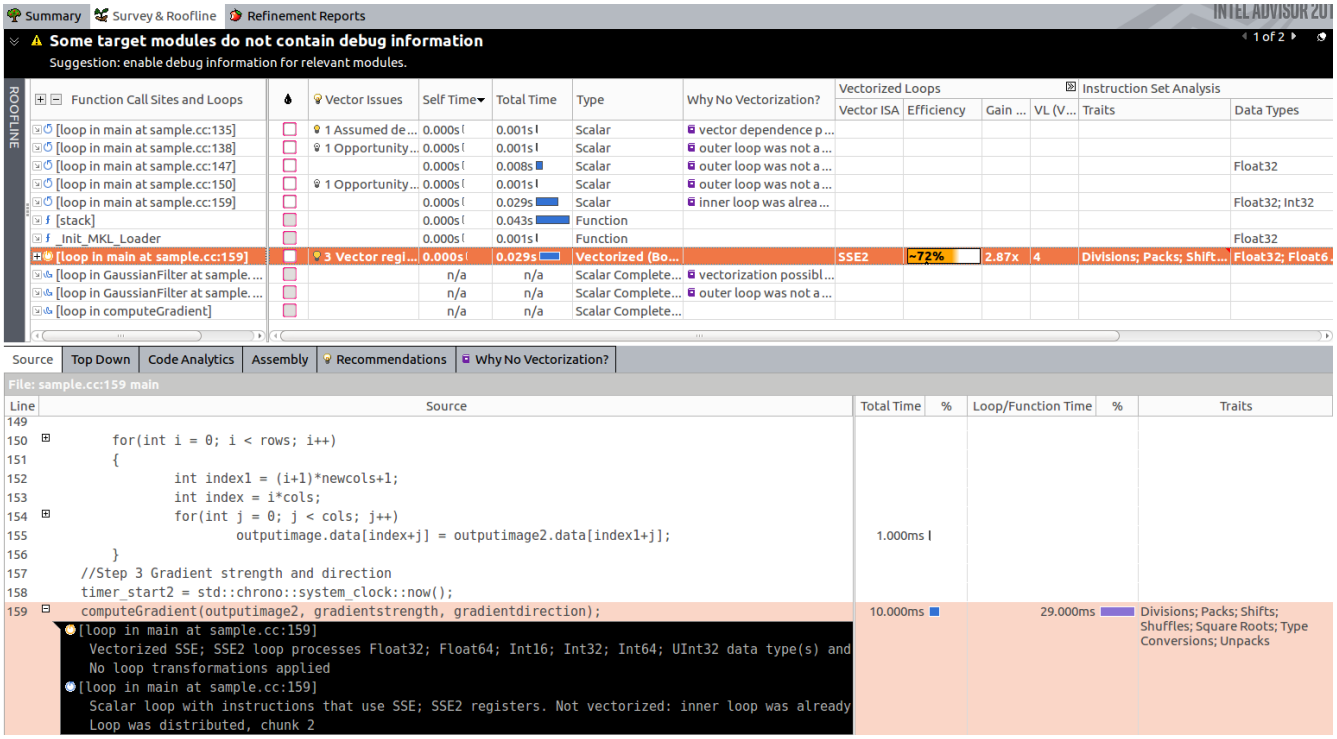
```
            for(int k1 = -1; k1 <= 1; k1++)
            {
                int index2 = index1+(k1*cols);
                for(int k2 = -1; k2 <=1; k2++)
                {
                    gradient_x += gradientx[k1+1][k2+1] * input.data[index2+k2];
                    gradient_y += gradienty[k1+1][k2+1] * input.data[index2+k2];
                }
            }
            strength.data[gindex1] = sqrt(pow(gradient_x, 2) + pow(gradient_y, 2));
            degrees = floor(((atan2(gradient_y, gradient_x) * 180 / PI)/45) + 0.5f);
            direction.data[gindex1] = (degrees == 4)?0:(degrees*45);
        }
    }
```

To explicitly specify vectorization we use OpenMP4.0 SIMD pragma. OpenMP SIMD feature can be enabled in Intel Compiler using compiler option -qopenmp-simd.

```
$ make
icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -std=c++11 -c sample.cc
icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location
icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -o sample sample.o

~/Lab2/step1-vectorization$ make run
./sample
Time elapsed for 1 frame = 0.199773 seconds
Time taken by RGB to GrayScale = 0.0052162 seconds
Time taken by Gaussian Filter = 0.0292243 seconds
Time taken by computeGradient = 0.106005 seconds
```

Time taken by Non-Maximum Suppression = 0.0506127 seconds

Running the Advisor Survey report on the new sample executable shows the following:



The report clearly shows that the loop is vectorized and it uses SSE2 instructions. Intel Compiler's default target architecture is SSE2. The traits show that there are some type conversions happening which is leading inefficiency in vectorization. To get more information to the kind of type conversions, please open the sample.optrpt file and you should see the following:

LOOP BEGIN at sample.cc(56,3)

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference strength->data[gindex1] has unaligned access [sample.cc(70,4)]

remark #15389: vectorization support: reference direction->data[gindex1] has unaligned access [sample.cc(72,4)]

remark #15381: vectorization support: unaligned access used inside loop body

remark #15305: vectorization support: vector length 4

remark #15309: vectorization support: normalized vectorization overhead 0.221

remark #15417: vectorization support: number of FP up converts: single precision to double precision 1 [/usr/local/include/c++/7.1.0/cmath(418,14)]

remark #15417: vectorization support: number of FP up converts: single precision to double precision 1 [/usr/local/include/c++/7.1.0/cmath(418,14)]

remark #15417: vectorization support: number of FP up converts: single precision to double precision 1 [sample.cc(71,14)]

remark #15418: vectorization support: number of FP down converts: double precision to single precision 1 [sample.cc(50,2)]

remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

remark #15450: unmasked unaligned unit stride loads: 18

remark #15451: unmasked unaligned unit stride stores: 2

remark #15475: --- begin vector cost summary ---

remark #15476: scalar cost: 572

remark #15477: vector cost: 198.750

remark #15478: estimated potential speedup: 2.860

remark #15482: vectorized math library calls: 2

remark #15486: divides: 2

remark #15487: type converts: 24

remark #15488: --- end vector cost summary ---

LOOP BEGIN at sample.cc(61,4)

LOOP BEGIN at sample.cc(64,5)

LOOP END

LOOP BEGIN at sample.cc(64,5)

LOOP END

LOOP BEGIN at sample.cc(64,5)

```
    LOOP END
  LOOP END
LOOP END
```

Step 2:

Navigate to step2-type-conversion folder and build the new executable using the new sample program. The changes made to the following program to reduce the FP up and down converts:

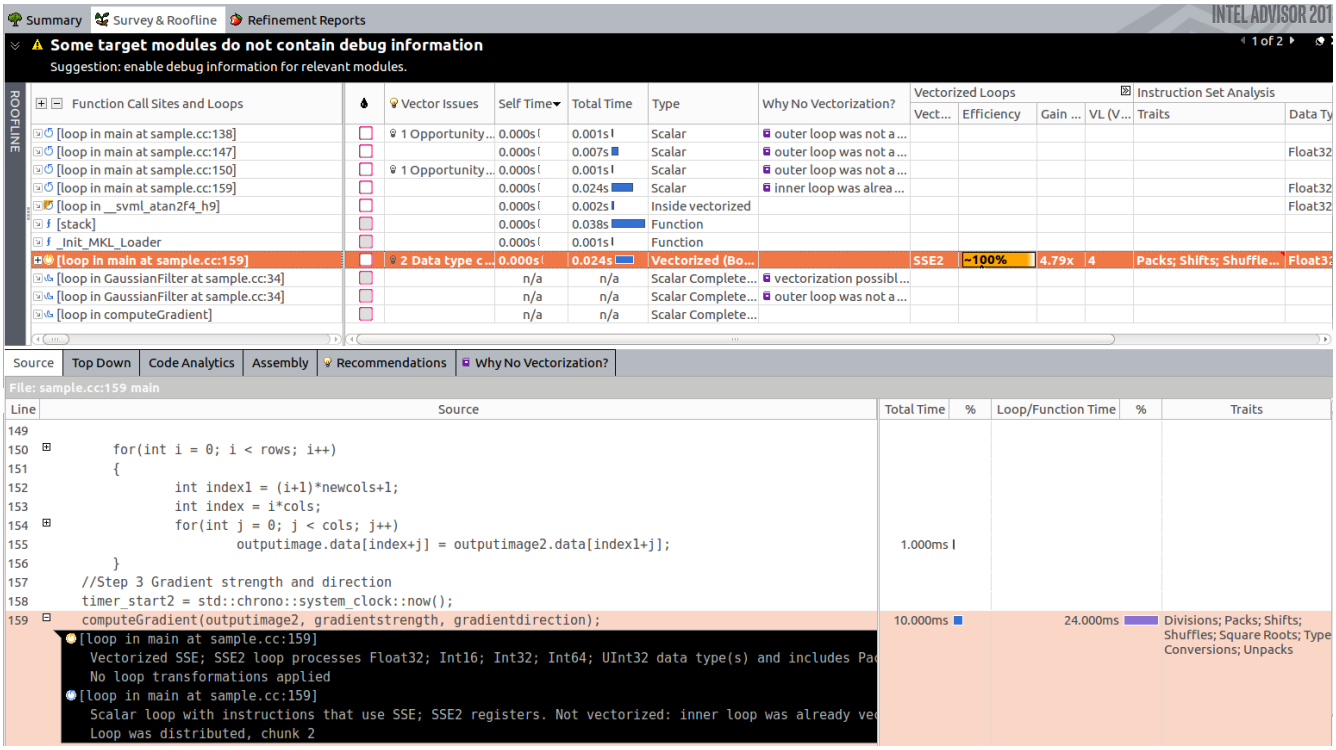
```
for(int i = 1; i < rows-1; i++)
{
    int index = i*cols+1;
    int gindex = (i-1)*gcols;
    #pragma omp simd private(gradient_x, gradient_y, degrees)
    for(int j = 0; j < cols-2; j++)
    {
        int index1 = index+j;
        int gindex1 = gindex+j;
        gradient_x = gradient_y = 0.0f;
        for(int k1 = -1; k1 <= 1; k1++)
        {
            int index2 = index1+(k1*cols);
            for(int k2 = -1; k2 <=1; k2++)
            {
                gradient_x += gradientx[k1+1][k2+1] * input.data[index2+k2];
                gradient_y += gradienty[k1+1][k2+1] * input.data[index2+k2];
            }
        }
        strength.data[gindex1] = sqrtf(powf(gradient_x, 2.f) + powf(gradient_y, 2.f));
```

```
            degrees = floorf(((atan2(gradient_y, gradient_x) * 180 / PI)/45) + 0.5f);
            direction.data[gindex1] = (degrees == 4)?0:(degrees*45);
        }
    }
```

Functions sqrt(), pow() and floor() takes floating point double precision values. That's the reason the type conversion is happening. Replacing these functions with its float equivalent gets rid of this issue as demonstrated below:

```
$ make
icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -std=c++11 -c sample.cc
icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location
icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -o sample sample.o

~/Lab2/step2-type-conversion$ make run
./sample
Time elapsed for 1 frame = 0.180935 seconds
Time taken by RGB to GrayScale = 0.00722456 seconds
Time taken by Gaussian Filter = 0.0306101 seconds
Time taken by computeGradient = 0.0813415 seconds
Time taken by Non-Maximum Suppression = 0.0493627 seconds
```

By fixing this, Advisor Survey report shows the vectorization efficiency is 100% as shown below:



The compiler optimization report (sample.optrpt) shows the following:

LOOP BEGIN at sample.cc(56,3)

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(66,44)]

remark #15389: vectorization support: reference input->data[index2+k2] has unaligned access [sample.cc(67,44)]

remark #15389: vectorization support: reference strength->data[gindex1] has unaligned access [sample.cc(70,4)]

remark #15389: vectorization support: reference direction->data[gindex1] has unaligned access [sample.cc(72,4)]

remark #15381: vectorization support: unaligned access used inside loop body

remark #15305: vectorization support: vector length 4

remark #15309: vectorization support: normalized vectorization overhead 0.373

remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

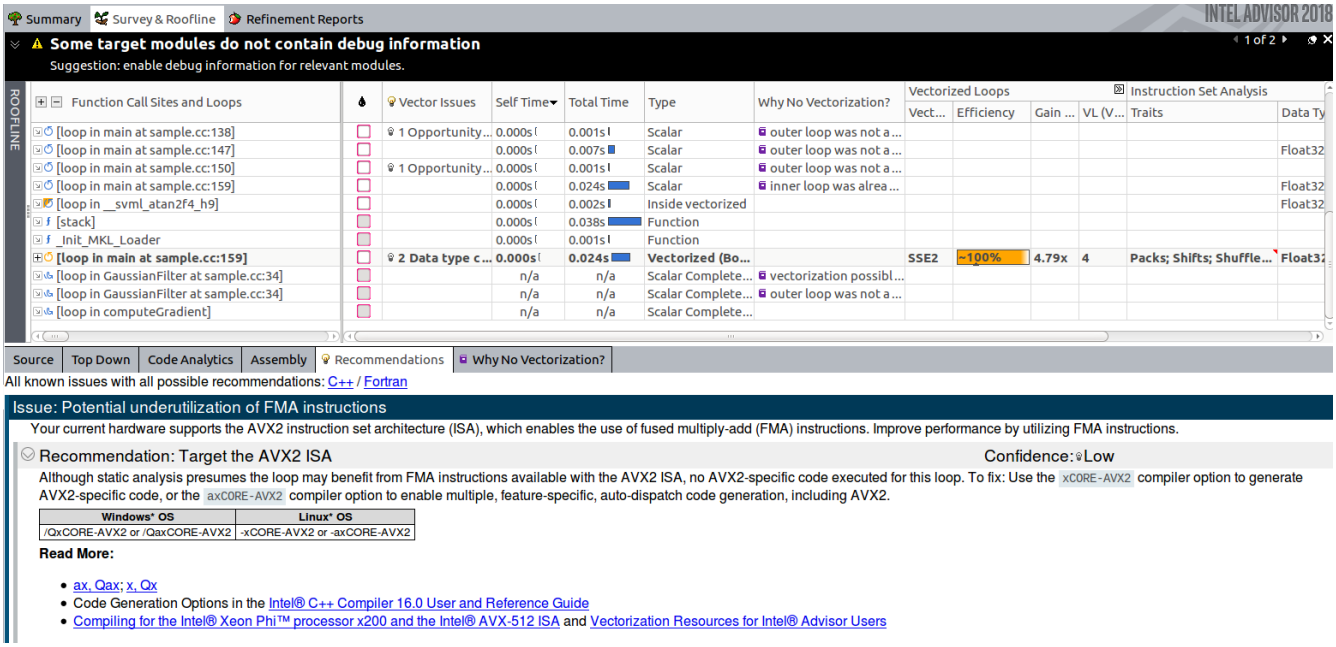
remark #15450: unmasked unaligned unit stride loads: 18

remark #15451: unmasked unaligned unit stride stores: 2

remark #15475: --- begin vector cost summary ---

```
    remark #15476: scalar cost: 556
    remark #15477: vector cost: 116.000
    remark #15478: estimated potential speedup: 4.760
    remark #15482: vectorized math library calls: 2
    remark #15486: divides: 2
    remark #15487: type converts: 20
    remark #15488: --- end vector cost summary ---

    LOOP BEGIN at sample.cc(61,4)

      LOOP BEGIN at sample.cc(64,5)
      LOOP END

      LOOP BEGIN at sample.cc(64,5)
      LOOP END

      LOOP BEGIN at sample.cc(64,5)
      LOOP END
    LOOP END
  LOOP END
```

The number of type converts is reduced from 24 to 20 and no more up and down converts of floating point values.



The recommendation tab of Advisor suggest that that target platform supports AVX architecture which support twice the vectorlength of SSE2 and also supports Fused Multiply Add (FMA) instructions. Our program will benefit from FMA instructions since it is a convolutional kernel will a weight multipler followed by a addition.

Step 3:

To enable Intel Compiler to target AVX architecture, you should use the compiler option -xCORE-AVX2. Navigate to step3-AVX folder to see the performance improvement targeting AVX architecture without changing the code.

$ make

icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -xCORE-AVX2 -std=c++11 -c sample.cc

icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location

icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -o sample sample.o

~/Lab2/step3-AVX$ make run

./sample

Time elapsed for 1 frame = 0.109059

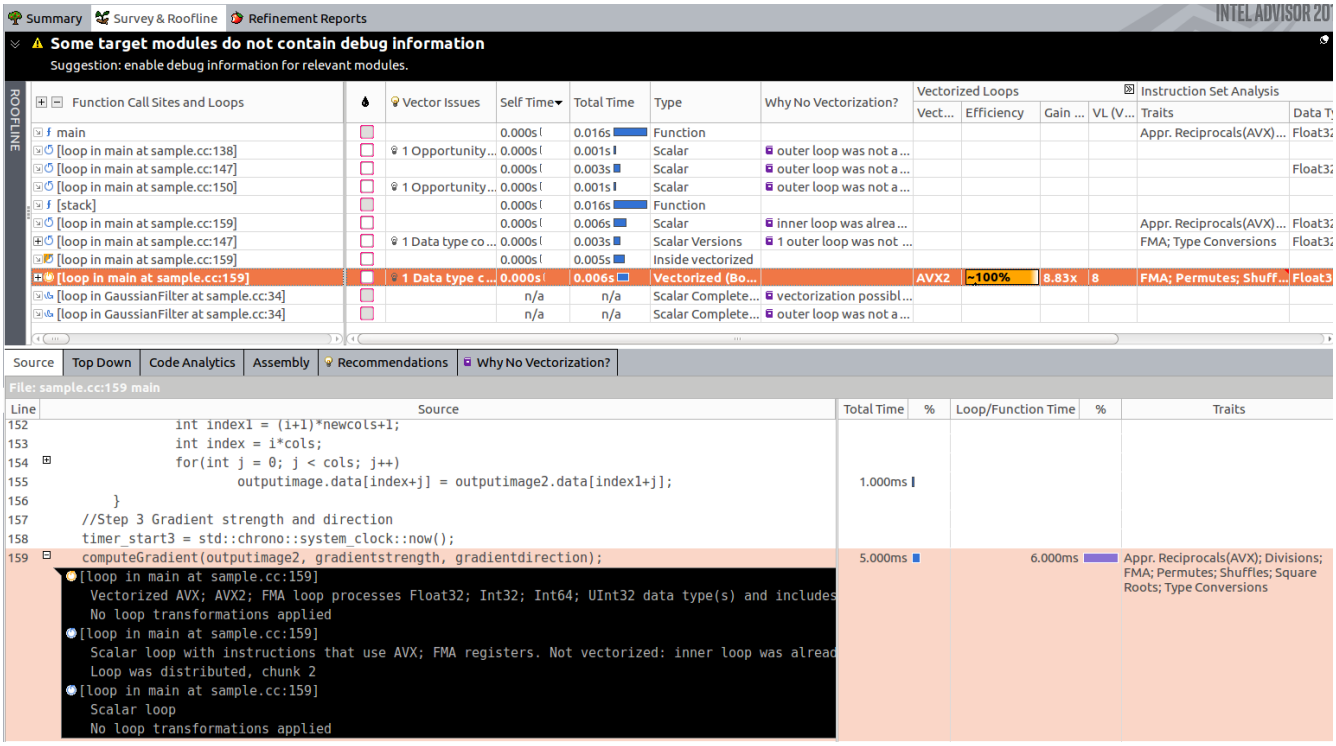
Time taken by RGB to GrayScale = 0.00573428 seconds

Time taken by Gaussian Filter = 0.0191398 seconds

Time taken by computeGradient = 0.0217945 seconds

Time taken by Non-Maximum Suppression = 0.0520314 seconds

Advisor Survey report shows the following:



Division instruction is more expensive instruction but produces more accurate results. Since you don't highly accurate results in Autonomous Driving(AD)/Machine Learning(ML) space we can instead instruct the compiler instead to generate a reciprocal followed by multiply instruction which is much faster than div instruction. By default with -O2 optimization, Intel Compiler generates div instructions and from optimization report of computeGradient function, it is clear that there are 2 div instructions created. We can avoid the generation of div instruction by using compiler option -no-prec-div:

~/Lab2/step4-division$ make

icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -xCORE-AVX2 -no-prec-div -std=c++11 -c sample.cc

icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location

icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -o sample sample.o

~/Lab2/step4-division$ make run

./sample

Time elapsed for 1 frame = 0.104826 seconds

Time taken by RGB to GrayScale = 0.00512914 seconds

Time taken by Gaussian Filter = 0.0186378 seconds

Time taken by computeGradient = 0.0207084 seconds

Time taken by Non-Maximum Suppression = 0.049872 seconds

Now that we have enabled explicit vectorization, we can try building this program with GCC compiler which supports OpenMP 4.0 SIMD. To enable GCC build in step4, change CC to g++ instead of icpc in the Makefile. You should see the following when you build the program using g++:

~/Lab2/step4-division$ make

g++ sample.cc -g -O2 -ffast-math -fopenmp-simd -march=haswell -std=c++11 -ftree-vectorize -fopt-info-vec -c sample.cc

/usr/local/include/opencv2/core/mat.inl.hpp:786:22: note: basic block vectorized

sample.cc:24:19: note: basic block vectorized

sample.cc:51:19: note: basic block vectorized

sample.cc:112:21: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:786:22: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:1313:5: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:1317:16: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:1330:18: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:786:22: note: basic block vectorized

sample.cc:24:19: note: basic block vectorized

sample.cc:51:19: note: basic block vectorized

sample.cc:112:21: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:786:22: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:1313:5: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:1317:16: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:1330:18: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

/usr/local/include/opencv2/core/mat.inl.hpp:403:11: note: basic block vectorized

g++ -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -o sample sample.o

~/Lab2/step4-division$ make run

./sample

Time elapsed for 1 frame = 0.390921 seconds

Time taken by RGB to GrayScale = 0.00906728 seconds

Time taken by Gaussian Filter = 0.0662739 seconds

Time taken by computeGradient = 0.249924 seconds

Time taken by Non-Maximum Suppression = 0.0538032 seconds

With GCC compiler, the code doesn't vectorize inspite of using OpenMP SIMD explicit vectorization. This clearly explains why Intel Compiler has an edge in terms of vectorization over GCC and the corresponding performance improvement is 10x.

Step 5:

We made sure the code utilizes one processing core efficiently by vectorizing the kernel. Now we can introduce threading to make sure the kernel runs across multiple cores efficiently. This can be accomplished using OpenMP threading or TBB threading model. The OpenMP threading enabled code is under step5-OpenMP-threading folder. The code in computeGradient() to enable OpenMP threading is as follows:

```
#pragma omp parallel for private(degrees, gradient_x, gradient_y)
for(int i = 1; i < rows-1; i++)
{
    int index = i*cols+1;
    int gindex = (i-1)*gcols;
    #pragma omp simd private(gradient_x, gradient_y, degrees)
    for(int j = 0; j < cols-2; j++)
    {
        int index1 = index+j;
        int gindex1 = gindex+j;
        gradient_x = gradient_y = 0.0f;
        for(int k1 = -1; k1 <= 1; k1++)
        {
            int index2 = index1+(k1*cols);
            for(int k2 = -1; k2 <=1; k2++)
            {
                gradient_x += gradientx[k1+1][k2+1] * input.data[index2+k2];
                gradient_y += gradienty[k1+1][k2+1] * input.data[index2+k2];
            }
        }
        (*strength).data[gindex1] = sqrtf(powf(gradient_x, 2.f) + powf(gradient_y, 2.f));
        degrees = floorf(((atan2(gradient_y, gradient_x) * 180.f / PI)/45.f) + 0.5f);
        (*direction).data[gindex1] = (degrees == 4)?0:(degrees*45);
    }
```

```
}
```

~/Lab2/step5-OpenMP-threading$ make

icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -xCORE-AVX2 -no-prec-div -qopenmp -std=c++11 -c sample.cc

icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location

icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -qopenmp -o sample sample.o

~/Lab2/step5-OpenMP-threading$ make run

./sample

Time elapsed for 1 frame = 0.106369 seconds

Time taken by RGB to GrayScale = 0.00590195 seconds

Time taken by Gaussian Filter = 0.0198485 seconds

Time taken by computeGradient = 0.00999099 seconds

Time taken by Non-Maximum Suppression = 0.0601309 seconds

In order to enable TBB threading instead of OpenMP, use parallel_for template function offered by TBB instead of OpenMP pragma. You cannot invoke the computeGradient function directly using parallel_for. The challenge is that we need to provide the range of values which in our case is the 0 to number of rows in the image. The sample code which enables TBB threading is available in step5-TBB-threading folder. Below are the code changes:

```
void computeGradient(uchar *input, uchar *strength, uchar *direction, int rows, int cols, const blocked_range<int> &p)
{
    float gradientx[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    float gradienty[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};
    //int rows = input.rows;
    //int cols = input.cols;
    int gcols = cols - 2;
```

```
    float gradient_x, gradient_y;
    float degrees;
    for(int i = p.begin(); i < p.end(); i++)
    {
        int index = i*cols+1;
        int gindex = (i-1)*gcols;
        #pragma omp simd private(gradient_x, gradient_y, degrees)
        for(int j = 0; j < cols-2; j++)
        {
            int index1 = index+j;
            int gindex1 = gindex+j;
            gradient_x = gradient_y = 0.0f;
            for(int k1 = -1; k1 <= 1; k1++)
            {
                int index2 = index1+(k1*cols);
                for(int k2 = -1; k2 <=1; k2++)
                {
                    gradient_x += gradientx[k1+1][k2+1] * input[index2+k2];
                    gradient_y += gradienty[k1+1][k2+1] * input[index2+k2];
                }
            }
            strength[gindex1] = sqrtf(powf(gradient_x, 2.f) + powf(gradient_y, 2.f));
            degrees = floorf(((atan2(gradient_y, gradient_x) * 180.f / PI)/45.f) + 0.5f);
            direction[gindex1] = (degrees == 4)?0:(degrees*45);
        }
    }
    return;
}
```

and at call site:

```
timer_start3 = std::chrono::system_clock::now();
auto compGrad = std::bind(computeGradient, outputimage2.data, gradientstrength.data, gradientdirection.data, newrows, newcols, _1);
parallel_for(blocked_range<int>(1, newrows-1, 100), compGrad);
timer_stop3 = std::chrono::system_clock::now();
```

~/Lab2/step5-TBB-threading$ make

icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -xCORE-AVX2 -no-prec-div -std=c++11 -c sample.cc

icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location

icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -ltbb -o sample sample.o

~/Lab2/step5-TBB-threading$ make run

./sample

Time elapsed for 1 frame = 0.0999311 seconds

Time taken by RGB to GrayScale = 0.00541552 seconds

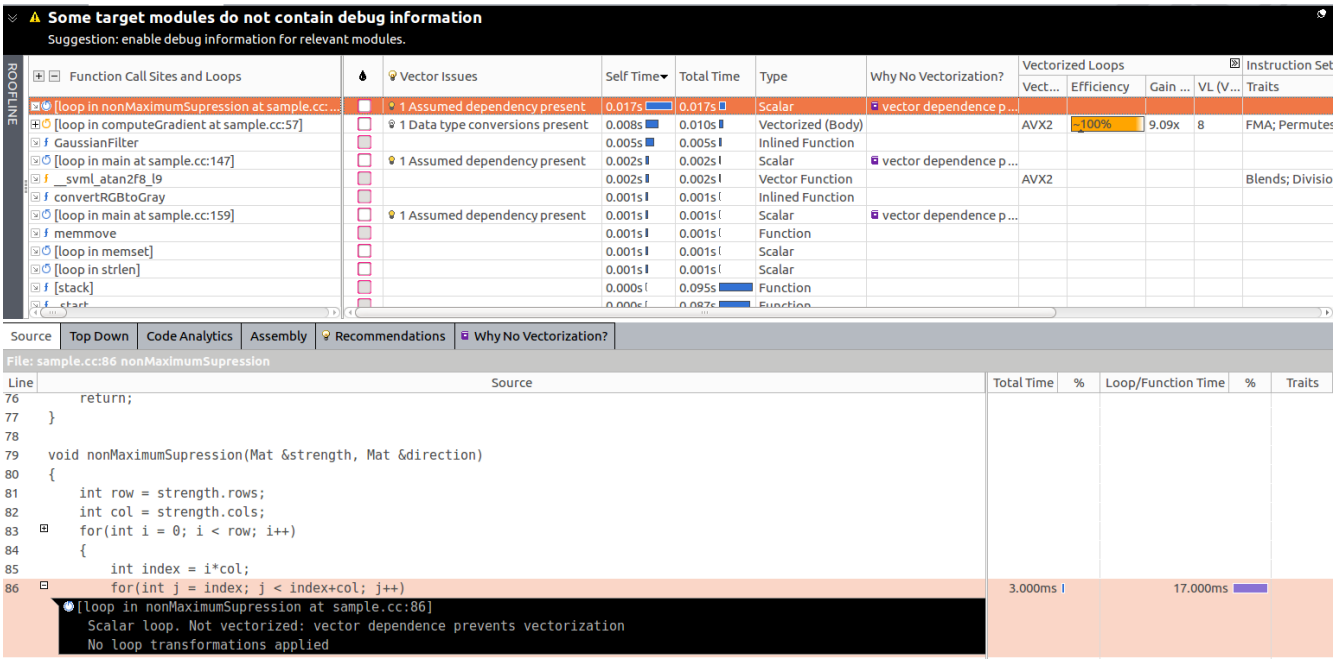
Time taken by Gaussian Filter = 0.0211469 seconds

Time taken by computeGradient = 0.00874305 seconds

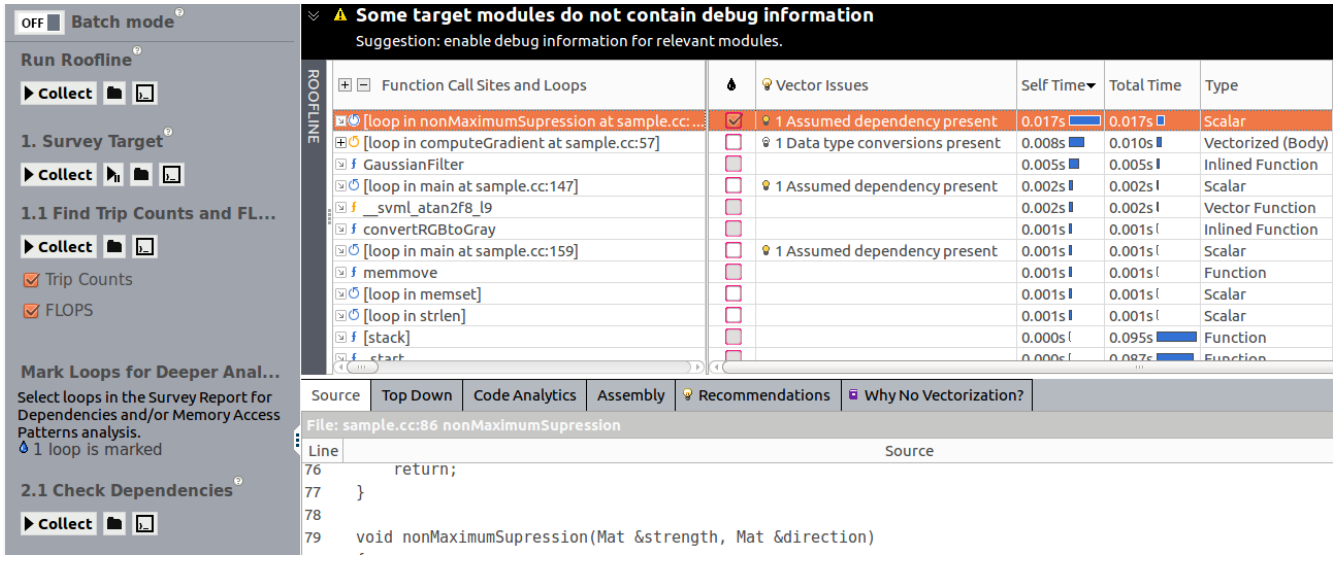
Time taken by Non-Maximum Suppression = 0.0543862 seconds

We accelerated the kernel computeGradient by 42x. Since computeGradient function is optimized, now non-Maximum suppression algorithm is the one which consumes the most time.

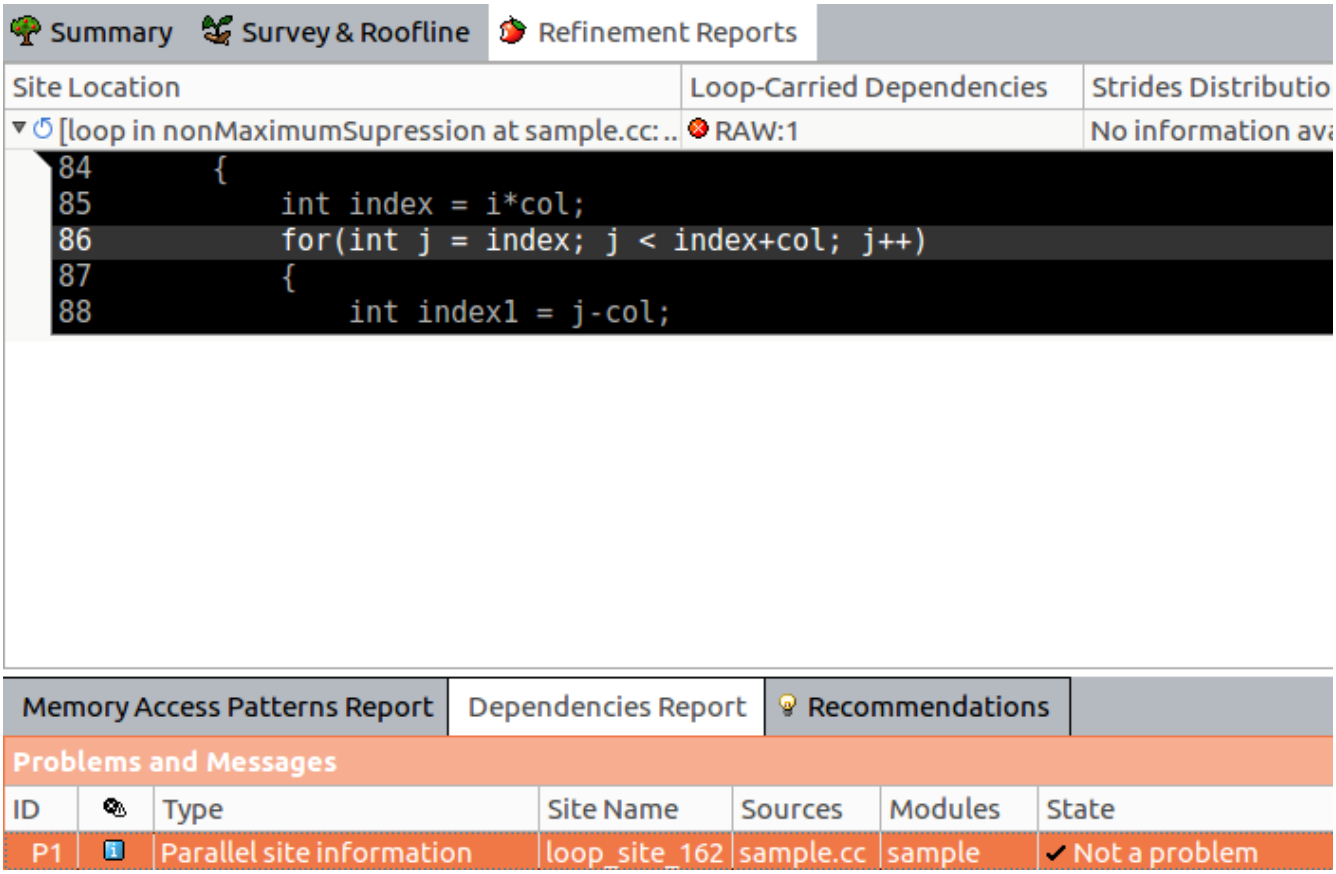
Step 6:



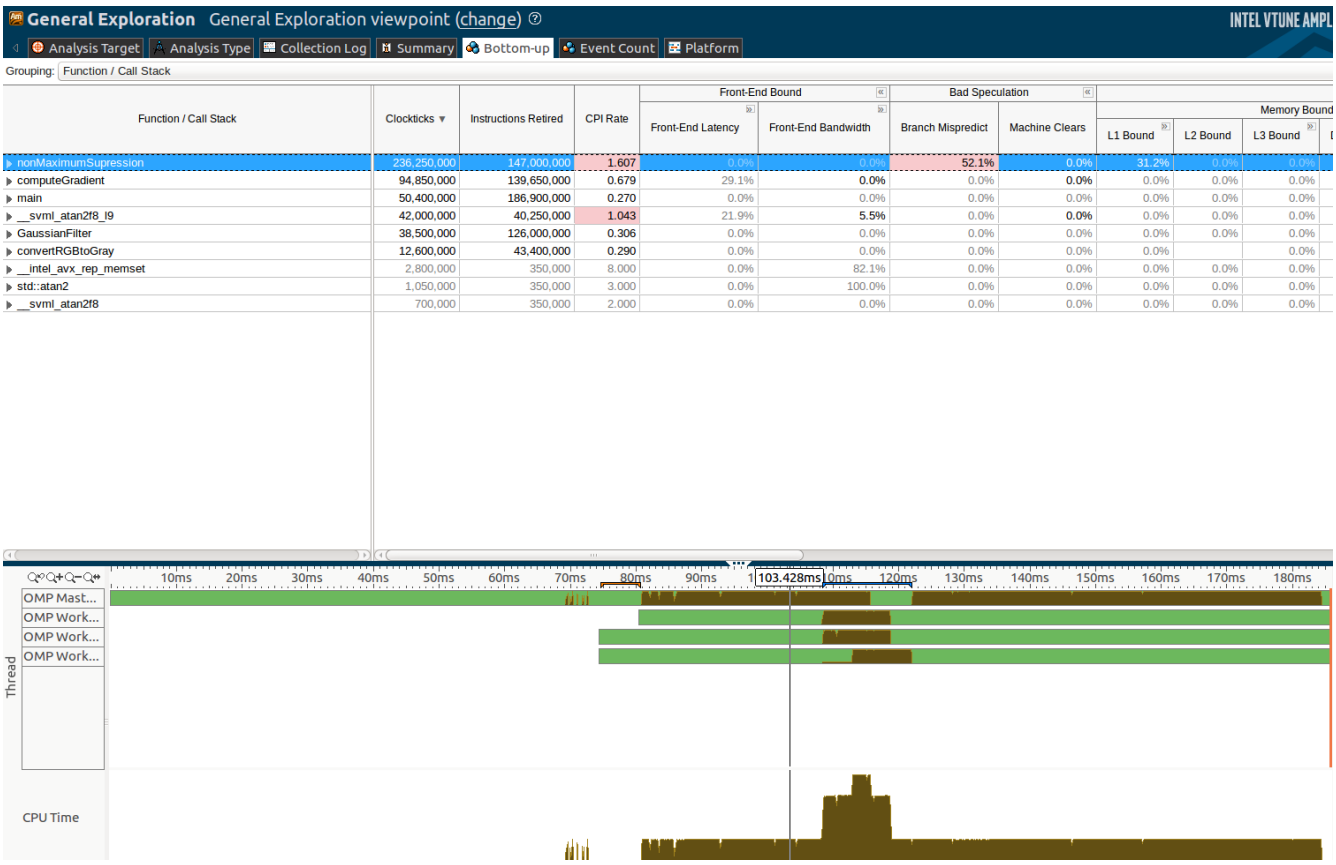
This function doesn't vectorize by default since compiler assumes vector dependence. Advisor provides a means of selecting this kernel and check if there is a real vector dependence during runtime using Check Dependency Refinement Analysis (Click on the "Collect" button next to Check Dependencies).



The analysis result states there is no real vector dependence:



Intel Compiler provides us with pragma ivdep (ignore vector dependence) which allows the developer to annotate the loops were the compiler assumes a vector dependence but there is no real vector dependence during runtime. The code which does this optimization is available in step6-NMS-Vec folder. This is the loop which loop body with lot of branches which are all equally probable as shown in Vtune General Exploration report:



To enable vectorization, we can annotate the inner loop with #pragma ivdep as shown below:

```
for(int i = 0; i < row; i++)
{
    int index = i*col;
    #pragma ivdep
    for(int j = index; j < index+col; j++)
    {
        int index1 = j-col;
        int index2 = j+col;
        switch(direction.data[j])
        {
```

```
                    case 0: if(!(strength.data[j]>strength.data[index1] &&
strength.data[j]>strength.data[index2]))
                        strength.data[j] = 0;
                        break;
                    case 45: if(!(strength.data[j]>strength.data[index1-1] &&
strength.data[j]>strength.data[index2+1]))
                        strength.data[j] = 0;
                        break;
                    case 90: if(!(strength.data[j]>strength.data[index-1] &&
strength.data[j]>strength.data[index+1]))
                        strength.data[j] = 0;
                        break;
                    case 135: if(!(strength.data[j]>strength.data[index1+1] &&
strength.data[j]>strength.data[index2-1]))
                        strength.data[j] = 0;
                        break;
                }


            }
        }
```

~/Lab2/step6-NMS-Vec$ make

icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -xCORE-AVX2 -no-prec-div -qopenmp -std=c++11 -c sample.cc

icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location

icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -qopenmp -o sample sample.o

~/Lab2/step6-NMS-Vec$ make run

./sample

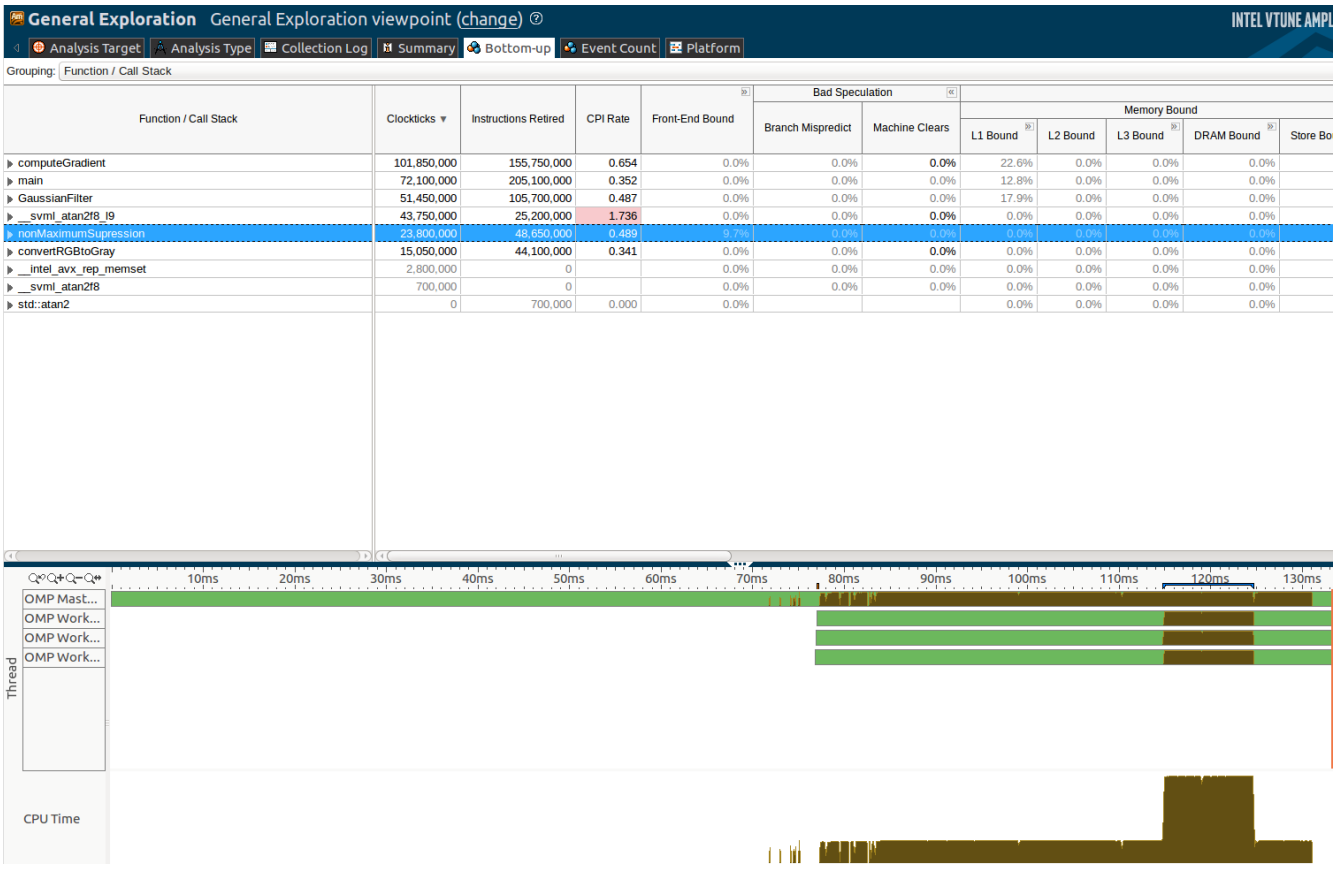
Time elapsed for 1 frame = 0.0670867 seconds

Time taken by RGB to GrayScale = 0.00595433 seconds

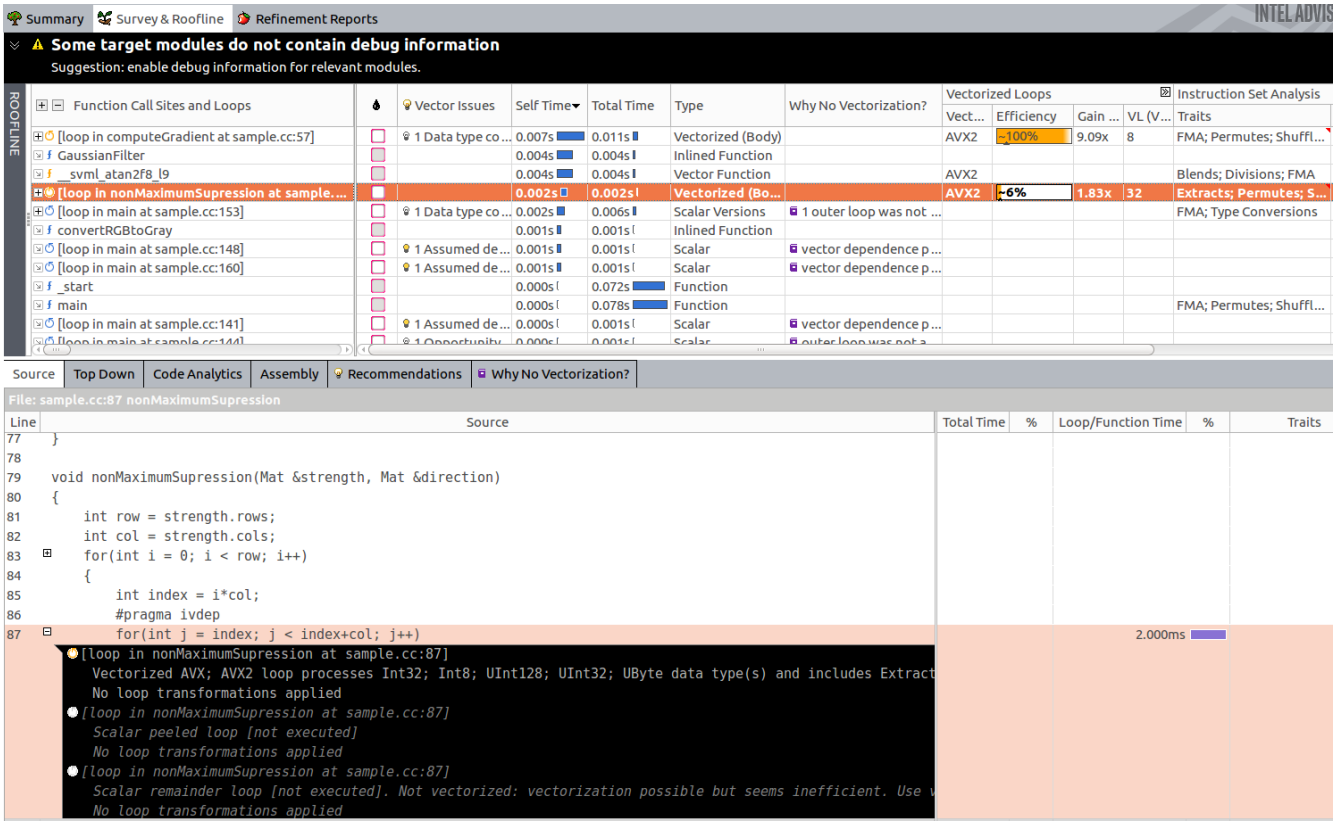
Time taken by Gaussian Filter = 0.0310803 seconds

Time taken by computeGradient = 0.00954408 seconds

Time taken by Non-Maximum Suppression = 0.00604505 seconds



As shown in the Vtune General Exploration report, there is no more Bad speculation once the vectorization is enabled because in this case. Though the vectorization efficiency is just 6% as shown by Advisor survey report, we still have better performance since the vectorlength is 32 for AVX architecture (data type we operate on in this loop is unsigned char).



Step 7:

Next step is to enable threading. For this case, we just enable threading using OpenMP but same TBB threading in previous step applies here too. The sample code which enables this optimization is available in step7-NMS-OpenMP folder. The code change is shown below:

```
#pragma omp parallel for
for(int i = 0; i < row; i++)
{
    int index = i*col;
    #pragma ivdep
    for(int j = index; j < index+col; j++)
    {
        int index1 = j-col;
        int index2 = j+col;
```

```
                switch(direction.data[j])
                {
                    case 0: if(!(strength.data[j]>strength.data[index1] &&
strength.data[j]>strength.data[index2]))
                            strength.data[j] = 0;
                            break;
                    case 45: if(!(strength.data[j]>strength.data[index1-1] &&
strength.data[j]>strength.data[index2+1]))
                            strength.data[j] = 0;
                            break;
                    case 90: if(!(strength.data[j]>strength.data[index-1] &&
strength.data[j]>strength.data[index+1]))
                            strength.data[j] = 0;
                            break;
                    case 135: if(!(strength.data[j]>strength.data[index1+1] &&
strength.data[j]>strength.data[index2-1]))
                            strength.data[j] = 0;
                            break;
                }

            }
        }
```

~/Lab2/step7-NMS-OpenMP$ make

icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -xCORE-AVX2 -no-prec-div -qopenmp -std=c++11 -c sample.cc

icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location

icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -qopenmp -o sample sample.o

~/Lab2/step7-NMS-OpenMP$ make run

./sample

Time elapsed for 1 frame = 0.066083 seconds

Time taken by RGB to GrayScale = 0.00780288 seconds

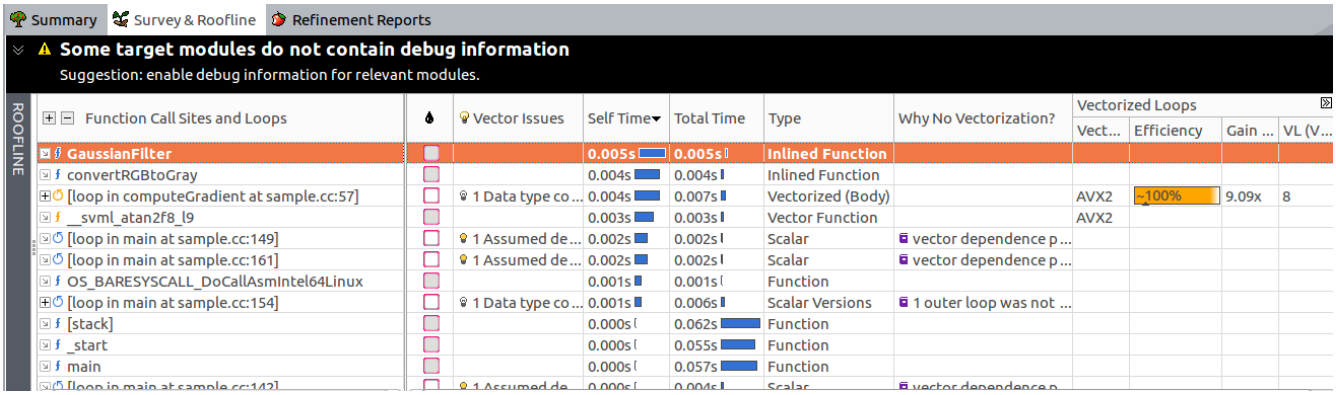
Time taken by Gaussian Filter = 0.0254764 seconds

Time taken by computeGradient = 0.00954917 seconds

Time taken by Non-Maximum Suppression = 0.00237935 seconds

Step 8:

From the Advisor survey report, it is clear that the next kernel which takes most time now is Gaussian Filter.



Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vectorized Loops: Vect...	Efficiency	Gain ...	VL (V...
GaussianFilter		0.005s	0.005s	Inlined Function					
convertRGBtoGray		0.004s	0.004s	Inlined Function					
[loop in computeGradient at sample.cc:57]	1 Data type co...	0.004s	0.007s	Vectorized (Body)		AVX2	~100%	9.09x	8
__svml_atan2f8_l9		0.003s	0.003s	Vector Function		AVX2			
[loop in main at sample.cc:149]	1 Assumed de...	0.002s	0.002s	Scalar	vector dependence p...				
[loop in main at sample.cc:161]	1 Assumed de...	0.002s	0.002s	Scalar	vector dependence p...				
OS_BARESYSCALL_DoCallAsmIntel64Linux		0.001s	0.001s	Function					
[loop in main at sample.cc:154]	1 Data type co...	0.001s	0.006s	Scalar Versions	1 outer loop was not ...				
[stack]		0.000s	0.062s	Function					
_start		0.000s	0.055s	Function					
main		0.000s	0.057s	Function					

We can OpenMP threading and vectorization in Gaussian Filter kernel as shown below:

```
#pragma omp parallel for private(value)
for(int i = 1; i < rows; i++)
{
    int index = i*cols+1;
    #pragma omp simd private(value)
    for(int j = index; j < index+cols; j++)
    {
        value = 0.0f;
```

```
            for(int k1 = -1; k1 <= 1; k1++)
            {
                int index1 = j+(k1*cols);
                for(int k2 = -1; k2 <= 1; k2++)
                    value += filter[k1+1][k2+1]*input.data[j+k2];
            }
            output.data[j] = value;
        }
    }
```

~/Lab2/step8-Gaussian-OpenMP-Vec$ make

icpc sample.cc -g -O2 -qopt-report5 -qopt-report-phase=vec -qopenmp-simd -xCORE-AVX2 -no-prec-div -qopenmp -std=c++11 -c sample.cc

icpc: remark #10397: optimization reports are generated in *.optrpt files in the output location

icpc -lopencv_core -lopencv_imgproc -lopencv_imgcodecs -lopencv_highgui -qopenmp -o sample sample.o

~/Lab2/step8-Gaussian-OpenMP-Vec$ make run

./sample

Time elapsed for 1 frame = 0.0429904 seconds

Time taken by RGB to GrayScale = 0.0184087 seconds

Time taken by Gaussian Filter = 0.00180491 seconds

Time taken by computeGradient = 0.00963621 seconds

Time taken by Non-Maximum Suppression = 0.00227702 seconds

Step 9:

Optimizing RGB to Gray scale kernel by introducing both threading and vectorization. Here we just have one loop and the pragma used to enable both threading and SIMD is shown below:

```
    #pragma omp parallel for simd
    for(int i = 0; i < rows*cols; i++)
    {
        int index = i*3;
        output.data[i] = 0.2989f*input.data[index] + 0.5870f*input.data[index+1] + 0.1140f *
input.data[index+2];
    }
```

```
~/Lab2/step9-RGBtoGray-OpenMP-Vec$ make run
./sample
```

Time elapsed for 1 frame = 0.026765 seconds

Time taken by RGB to GrayScale = 0.006335 seconds

Time taken by Gaussian Filter = 0.00180292 seconds

Time taken by computeGradient = 0.0100703 seconds

Time taken by Non-Maximum Suppression = 0.00228489 seconds

Naive Bayes Learning Algorithm