Hands-on "Vectorization"

Execute the following steps before start the hands-on:

```
ssh -X phi04.ncc.unesp.br
cd ~/
git clone https://github.com/intel-unesp-mcp/workshop-HPC-ML.git
source /opt/intel/parallel_studio_xe_2017.4.056/psxevars.sh intel64
```

1. Identifing the computational resources of KNL:

Execute the comand **lscpu**

what are the amount of processors / cores?

How much memory is available at each cache level?

Execute the comand **numactl -H**

How many nodes are available?

2. Executing multiplication transposition[1] on KNL using different memory systems:

Execute the following commands:

```
cd ~/workshop-HPC-ML/knl/3
make clean
make runme-CPU
```

Execute the matrix transposition using mcdram

```
time numactl -m 4,5,6,7 ./runme-CPU 15000 100
```

Execute the matrix transposition using dram

```
time numactl -m 0,1,2,3 ./runme-CPU 15000 100
```

For this matrix transposition which memory system has better performance?

1https://colfaxresearch.com/multithreaded-transposition-of-square-matrices-with-common-code-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors/

3. Vectorization Report

Execute the following commands:

cd ~/workshop-HPC-ML/

icc vect.c -o vectAVX512 -O3 -qopt-report5

Open the vectorization report vect.optrpt and search for loop on main function. This loop was automaticaly vectorized, but the loop on hist function was not, due to data dependencies. The indirection in the index of variable samples inside function hist inhibited vectorization.

Execute the following command to open vectorization report

nano vect.optrpt

The new vector instruction set AVX-512, available on the new Xeon Phi KNL, provides support for indirection called Confliction Detection. Now perform the same compilation but using -xhost which sets up the compiler to use the highest vector instruction set available, in this case AVX-512:

icc vect.c -o vectAVX512 -O3 -qopt-report5 -xhost

nano vect.optrpt

4. **Vectorization Report**

Execute the following commands:

ssh -X phi02.ncc.unesp.br

cd ~/

source /opt/intel/parallel_studio_xe_2017.1.043/psxevars.sh intel64

git clone https://github.com/intel-unesp-mcp/Hands-on-Workshop.git

1) Compile the example with vec-report6 and o3

```
icc func.c -c -vec-report6 -O3 -g
icc VectorizationHandson.c func.o -o VectorizationHandson -vec-report6
-O3 -g
```

2) Open the vectorization report (VectorizationHandson.optrpt)

Note that the loop on function main was automatically vectorized;

remark #15300: LOOP WAS VECTORIZED

3) Open the vectorization report of func (func.optrpt) and note that the loops on function add_floats and on function quad were not automatically vectorized;

remark #15344: loop was not vectorized: vector dependence prevents vectorization

4) Create New Project on Intel Advisor to evaluate the application VectorizationHandson (figures 1 to 4)

   Execute Intel Advisor on terminal: advixe-gui

   create new Advisor Project using the following parameters:

   - name: Vectest
   - application: ~/handson/vectorization/VectorizationHandson
   - Source Folder: ~/handson/vectorization/

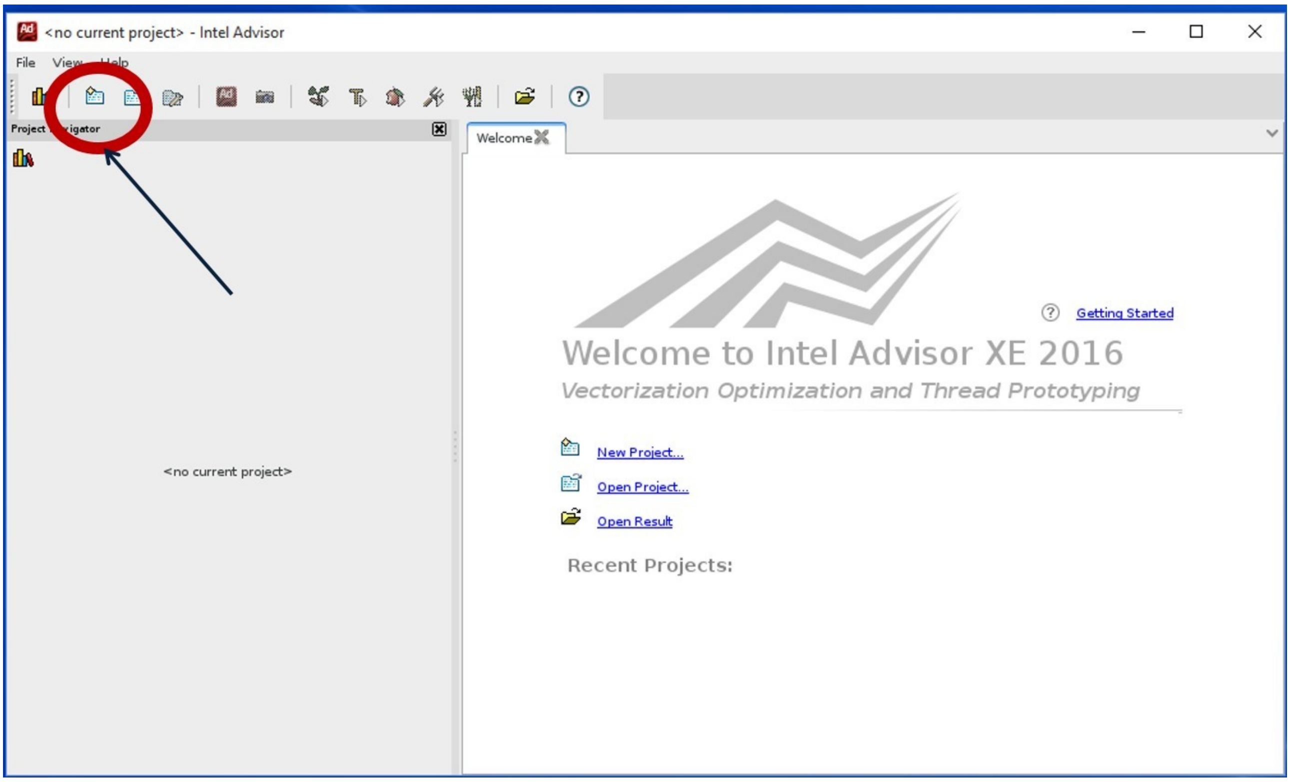

Figure 1. Main Intel Advisor Window.

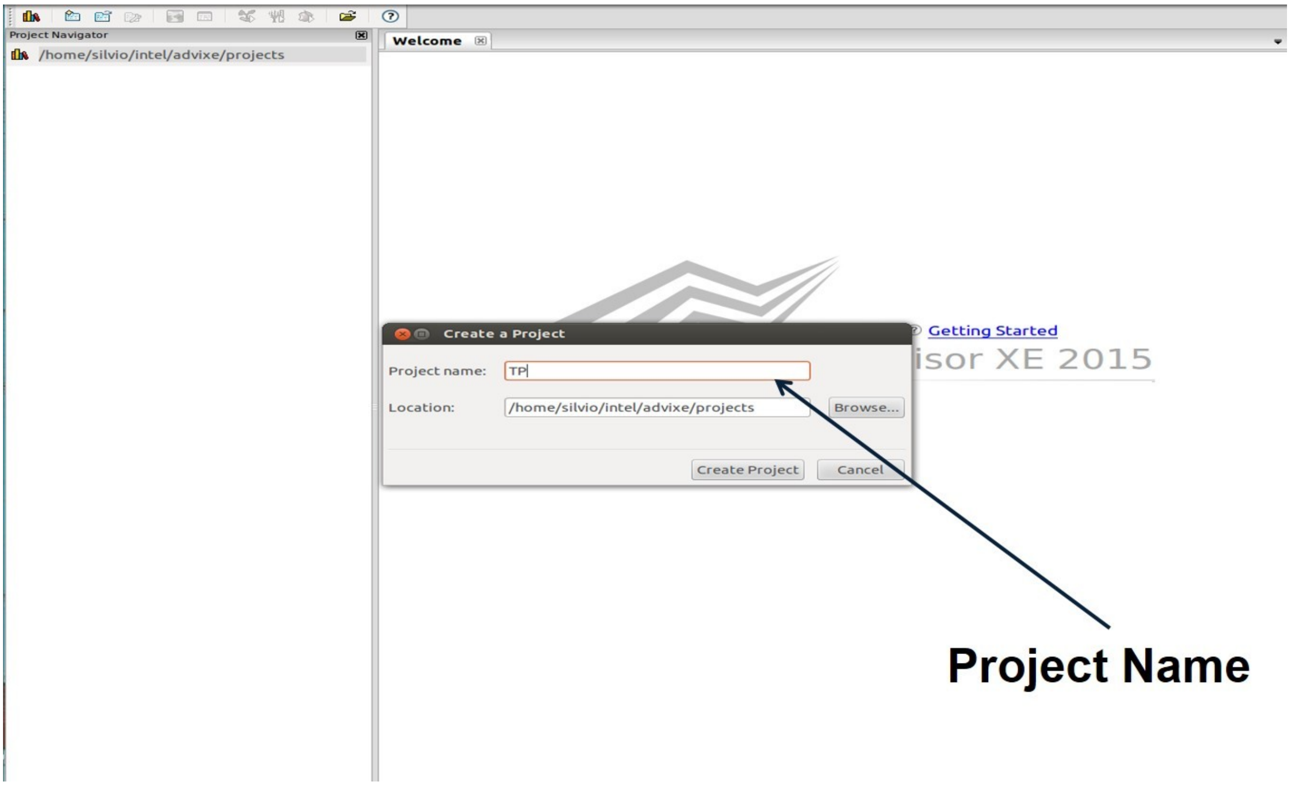

Figure 2. Creating new Project

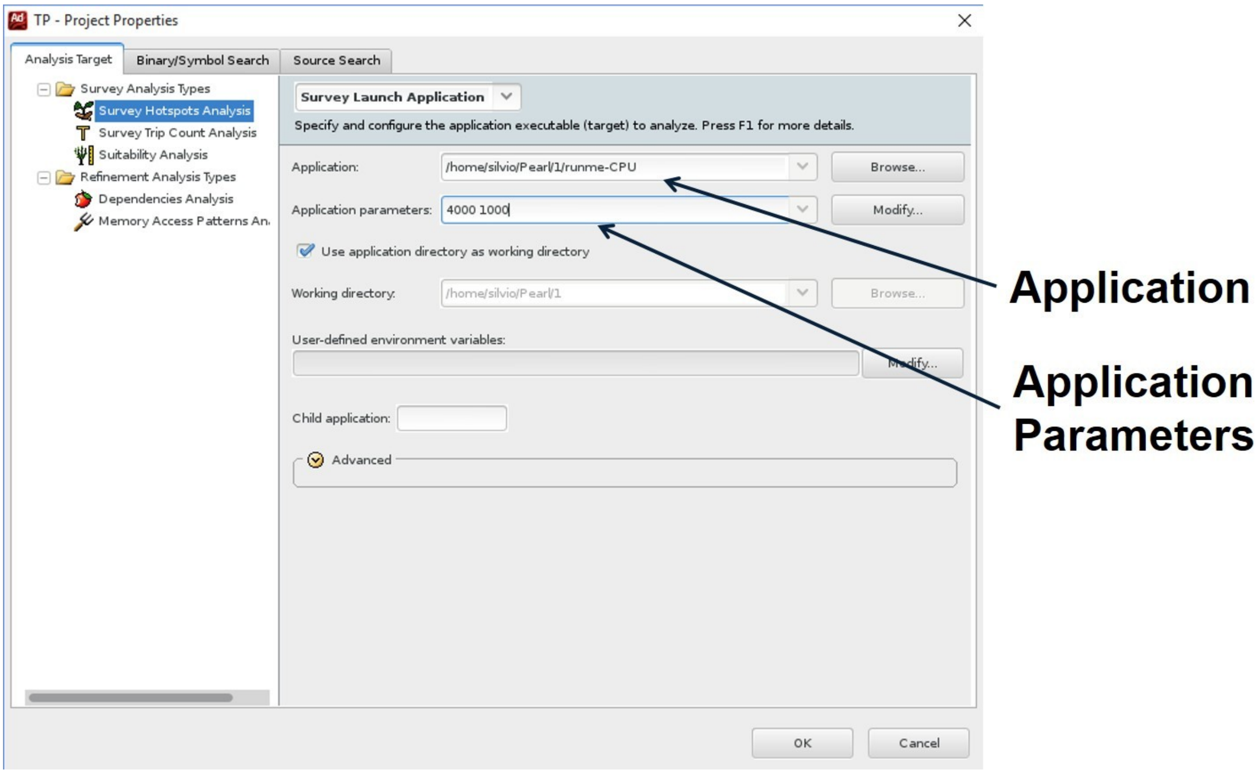

Figure 3. Configuring Project.

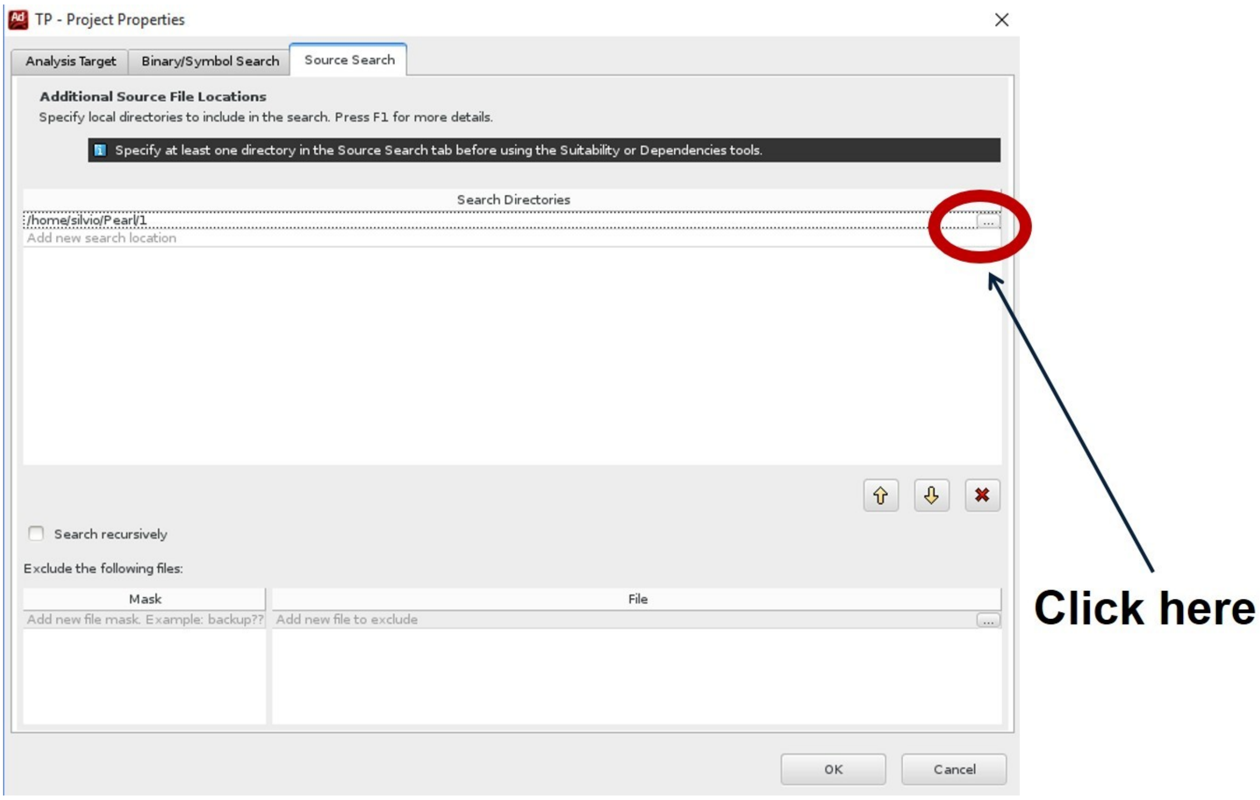

Figure 4. Setup search directory.

5) Start Survey Target Report (figure 5)

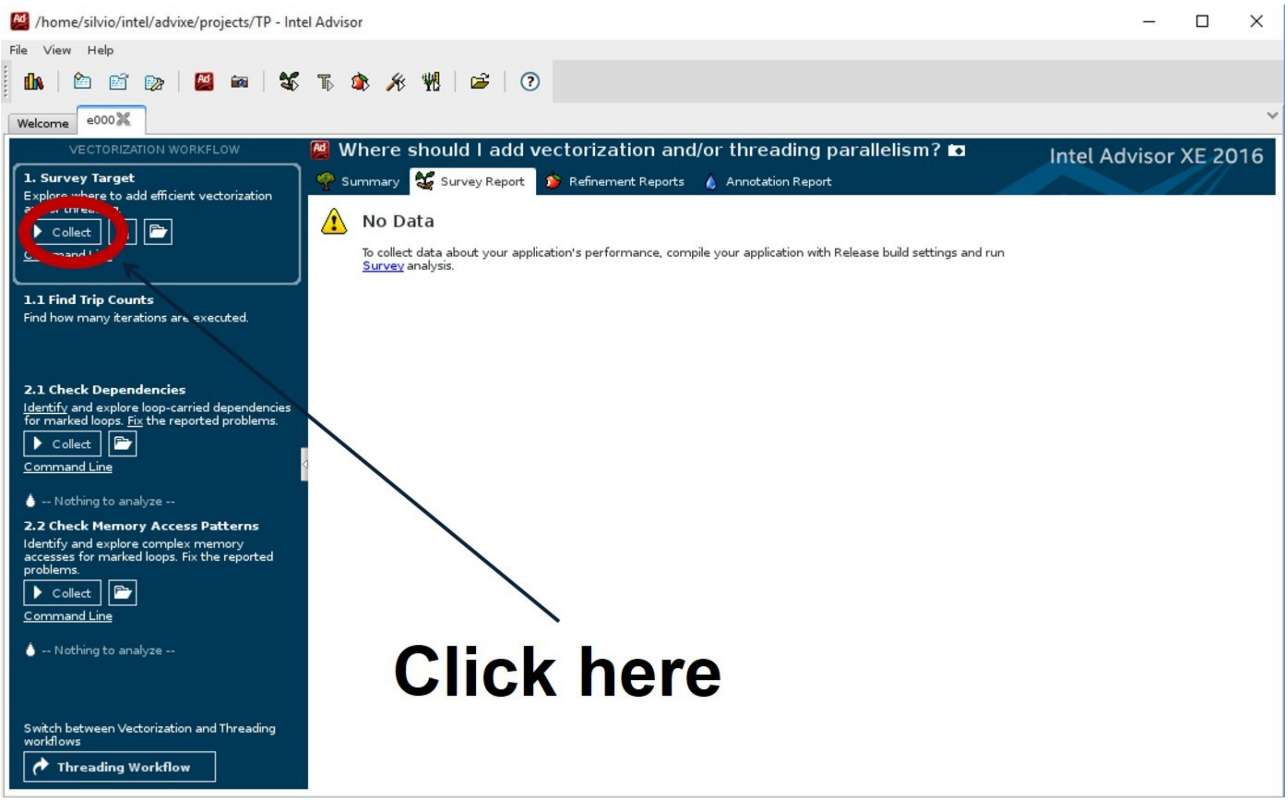

Figure 5. Survey Target Report.

6) Execute Trip Count Analysis (figure 6):

How many times the loop on function quad2 was executed?

Trip Counts				
Median	Min	Max	Call Count	Iteration Duration
5	5	5	1	
3	3	3	1	
1024	1024	1024	1	
64	64	64	1024	
1024	1024	1024	1	0.0009s
1024	1024	1024	1024	< 0.0001s
512	512	512	1048576	< 0.0001s

Figure 6. Trip Count Results.

7) Check dependency of inner loop on function add_floats and on function quad (figure 7)

- Mark Loop for deeper analysis;
- Click on “check dependency”;

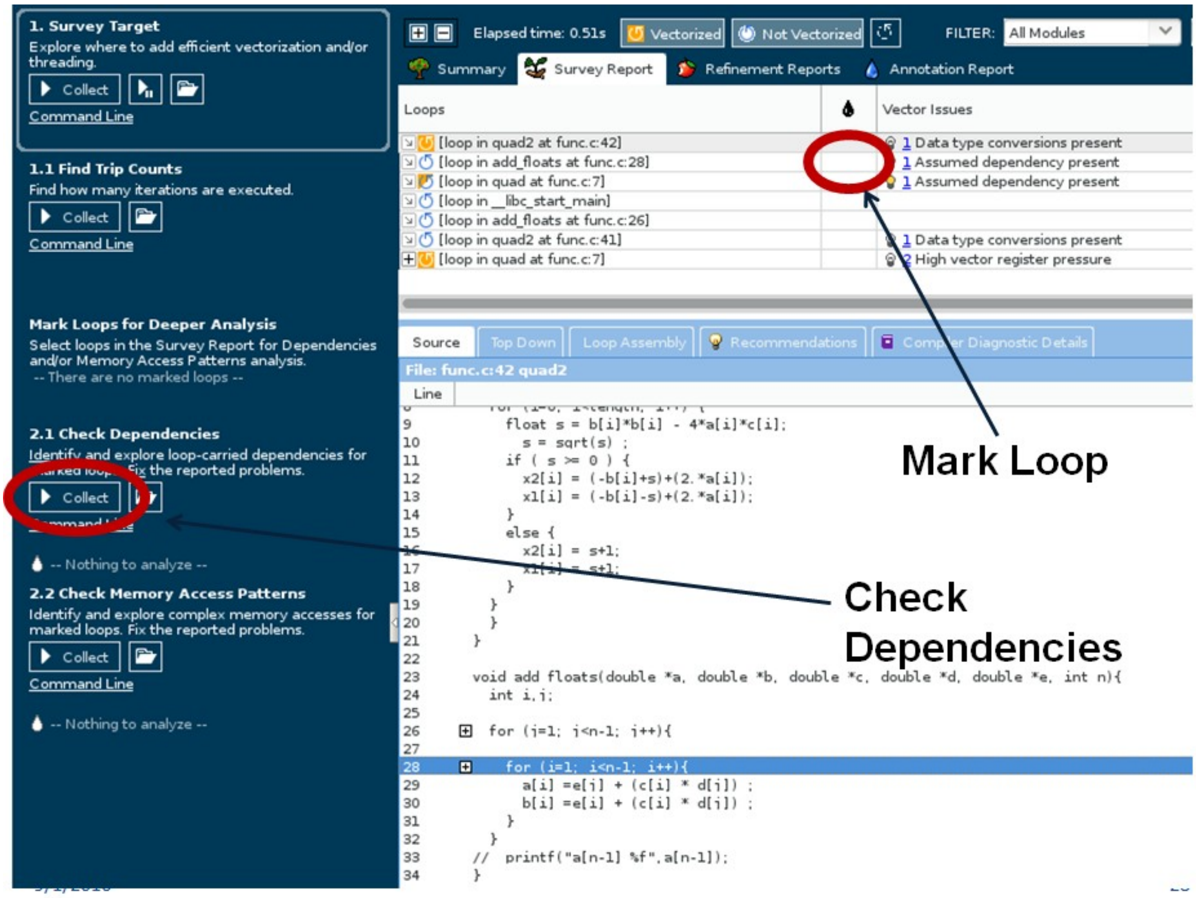

Figure 7. check dependency.

Note that no dependencies were found, so It is safe to vectorize the loop;

8) Include “#pragma ivdep directive” in top of inner loop “for (i=0; i<n; i++)” on function add_floats

```
#pragma ivdep
for (i=0; i<n; i++){
```

9) Include keyword **restrict** on all arguments that receives pointers on function quad (func.c):

```
void quad(int length, double * restrict a, double * restrict b, double *
restrict c, double * restrict  x1, double * restrict  x2)
```

10) Recompile the example and run survey target again

```
icc func.c -g -c -vec-report6 -O3 -restrict
icc VectorizationHandson.c -g func.o -o VectorizationHandson -vec-report6
-O3
```

Note that this loop was vectorized.

11) Check Memory Access Pattern

- Mark inner Loop of function quad2 for deeper analysis (for(i=0; i<40000; i++) ;
- Click on “check dependency”;
- Open the refinement reports (figure 8)

Note that the stride distribution is “constant stride”

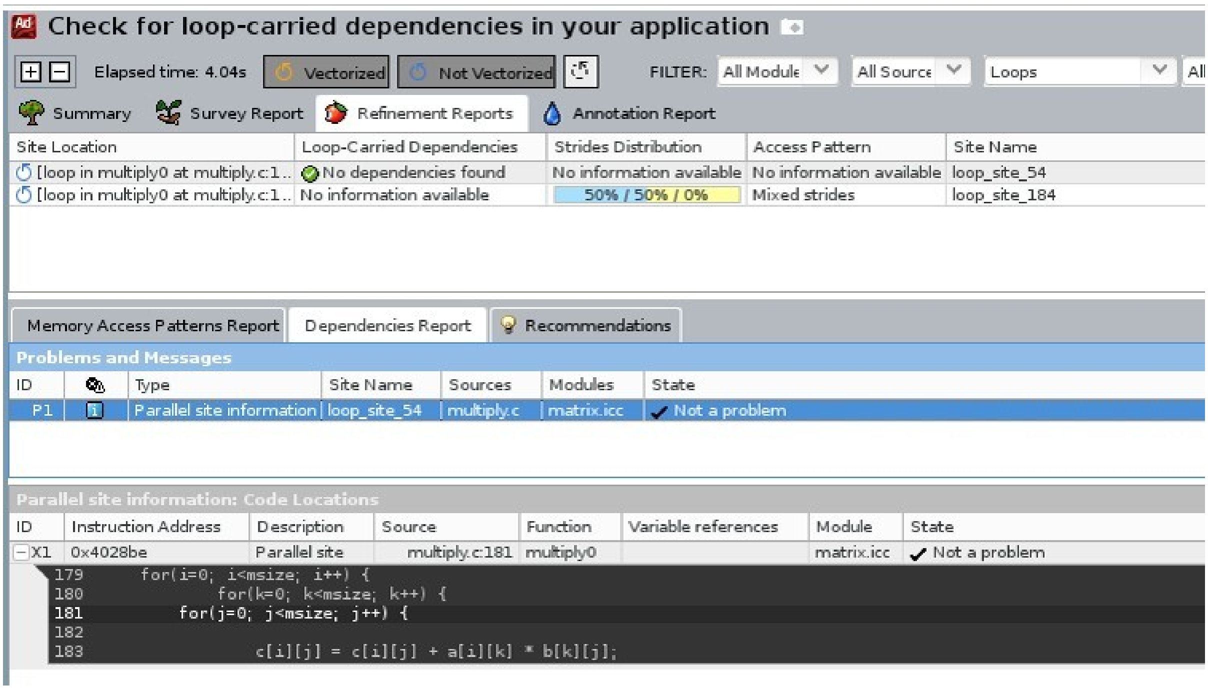

Figure 8. Refinement Report Window.

12) Redesign the structure to SOA (Structure of Arrays) format

Change the structure on file func.h from AOS to SOA

```
struct coordinate {
    float x[40000], y[40000], z[40000];
} obj;
```

Change the body of inner loop on quad2 function:

```
obj.x[i]=i + randV;
obj.y[i]=i*i+ randV;
obj.z[i]=i+i+ randV;
```

Run the Check Memory Access Pattern again. Note that the stride distribution is unit stride now.

13) Recompile application using AVX:

Recompile application using xhost

```
icc func.c -g -c -vec-report6 -O3 -restrict -xhost
icc VectorizationHandson.c -g func.o -o VectorizationHandson -vec-report6 -O3 -xhost
```

Collect Survey Data again

Note that now the code was compiled using AVX