

## Hands-on "Vectorization"

- Vectorization Report
- Intel Advisor
  - Create Advisor Project
  - Collect Survey Data
  - Check Trip Count
  - Check Dependencies
  - Check Memory Access Patterns
  - Test xhost option

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/

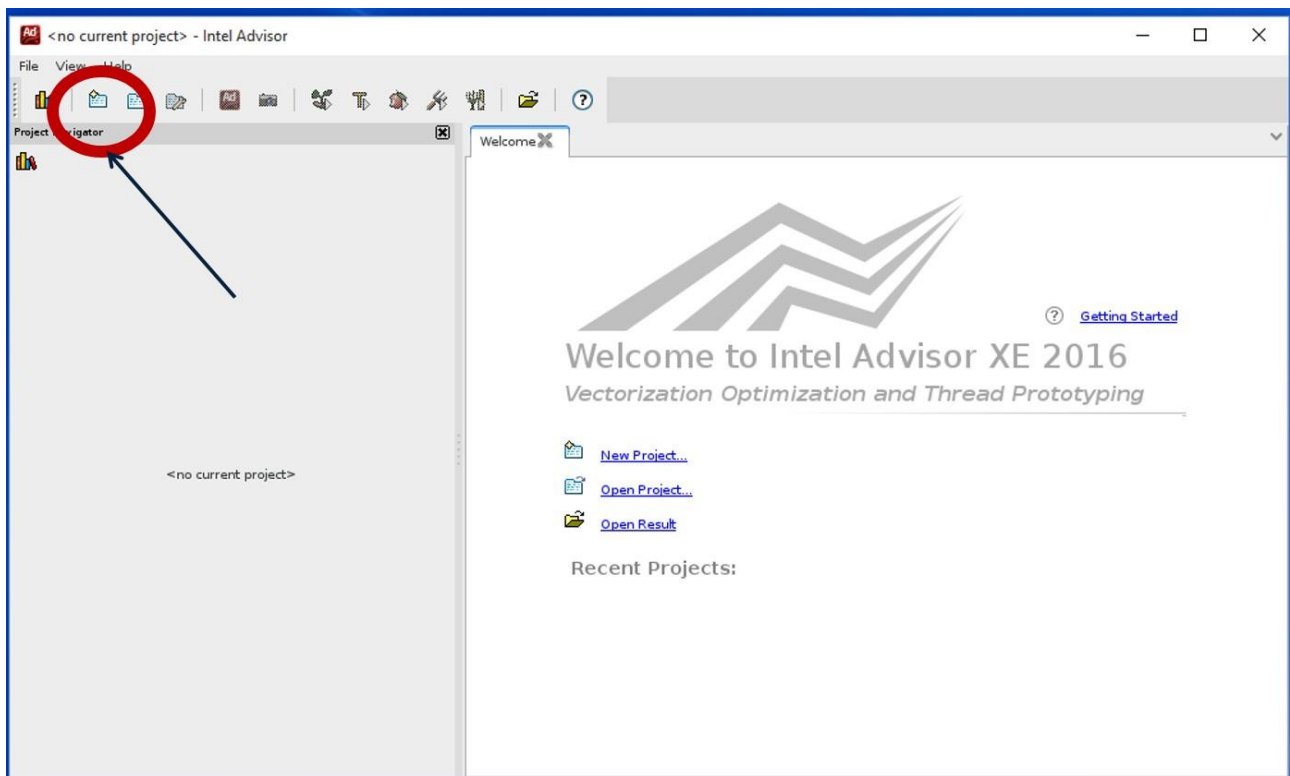


Figure 1. Main Intel Advisor Window.

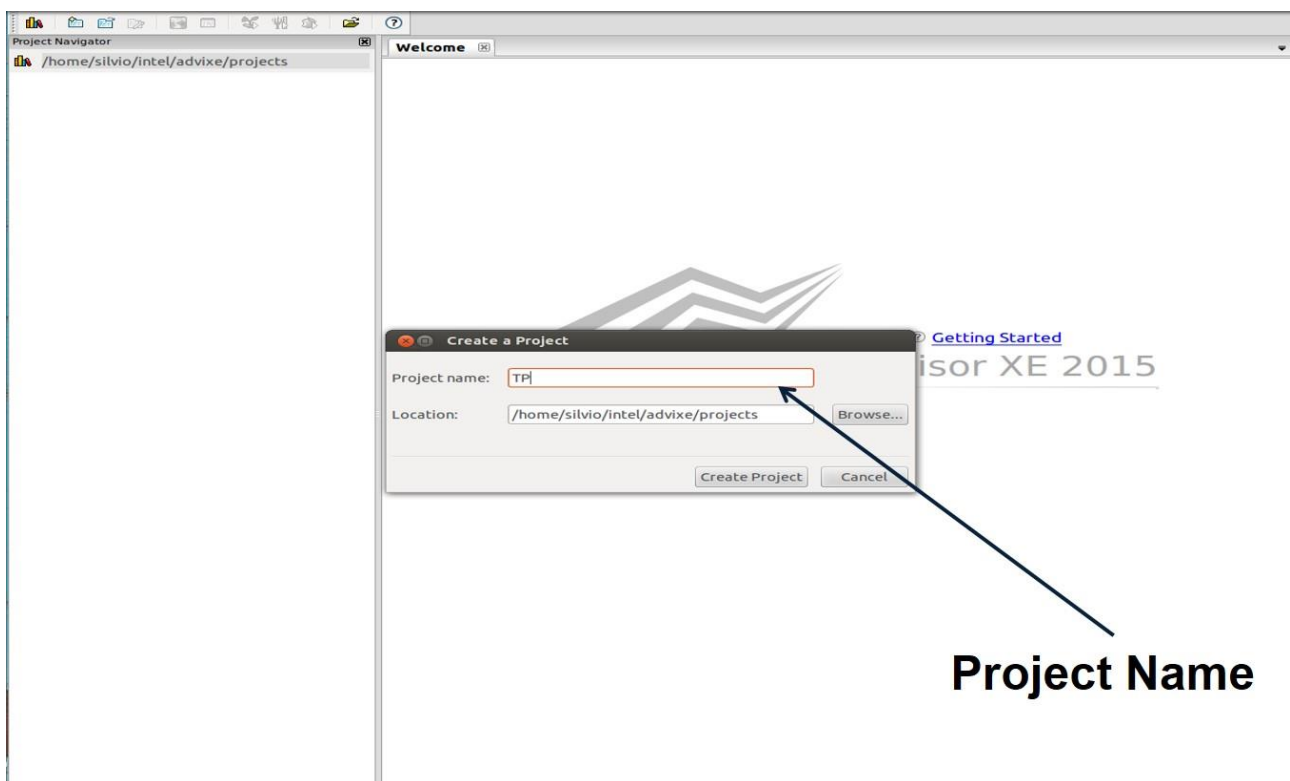


Figure 2. Creating new Project

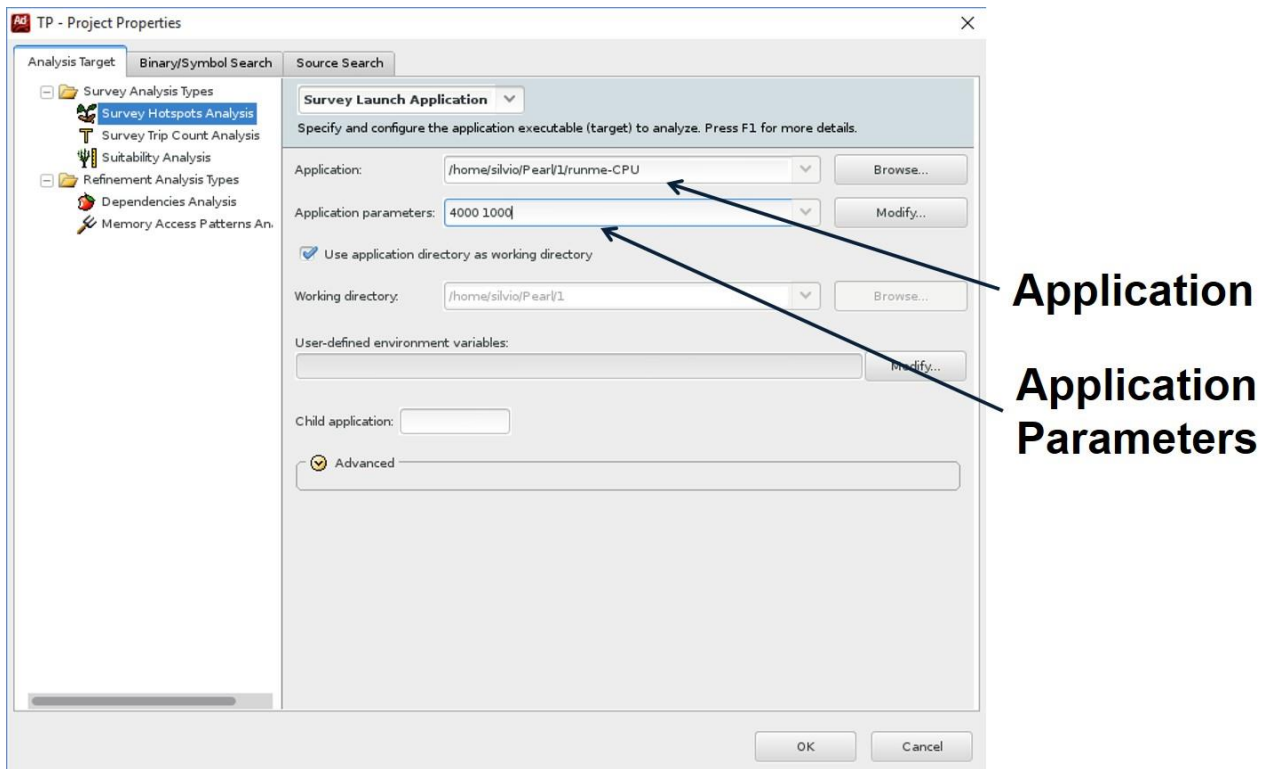


Figure 3. Configuring Project.

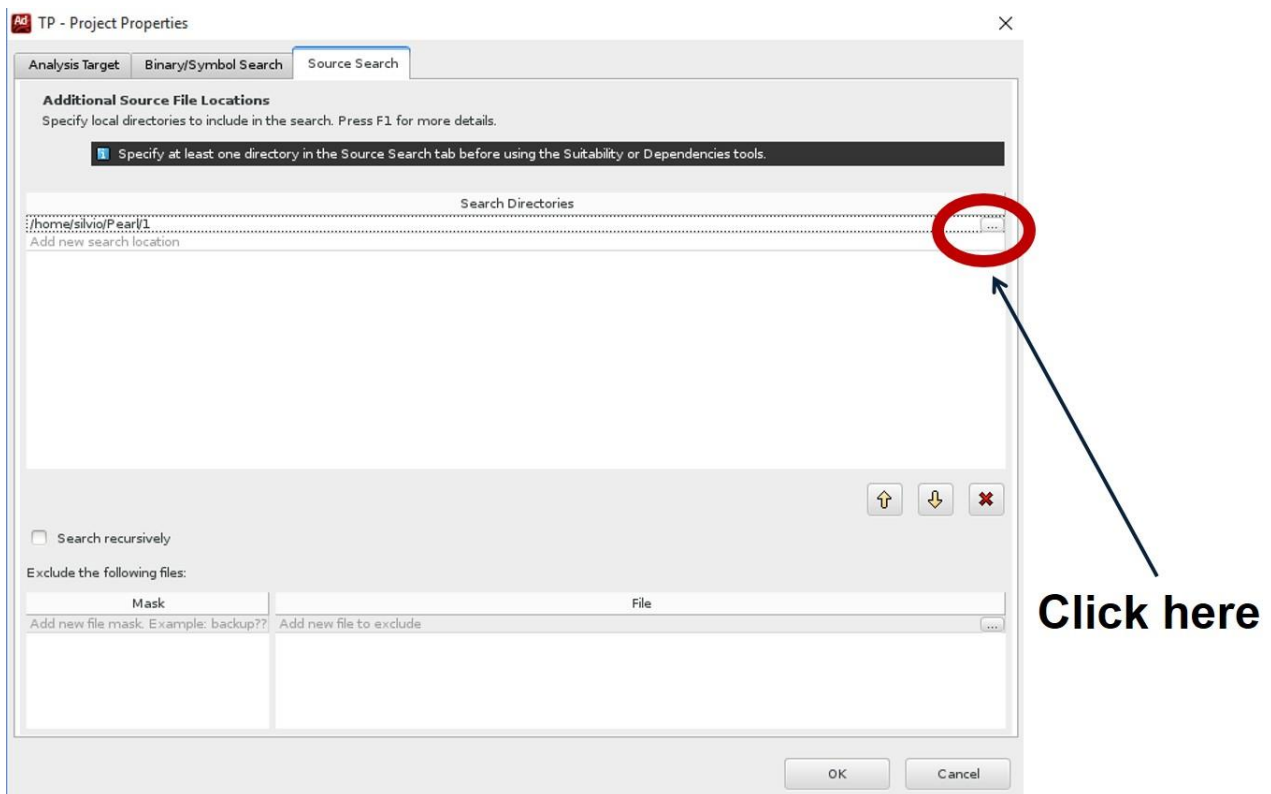


Figure 4. Setup search directory.

## 5) Start Survey Target Report (figure 5)

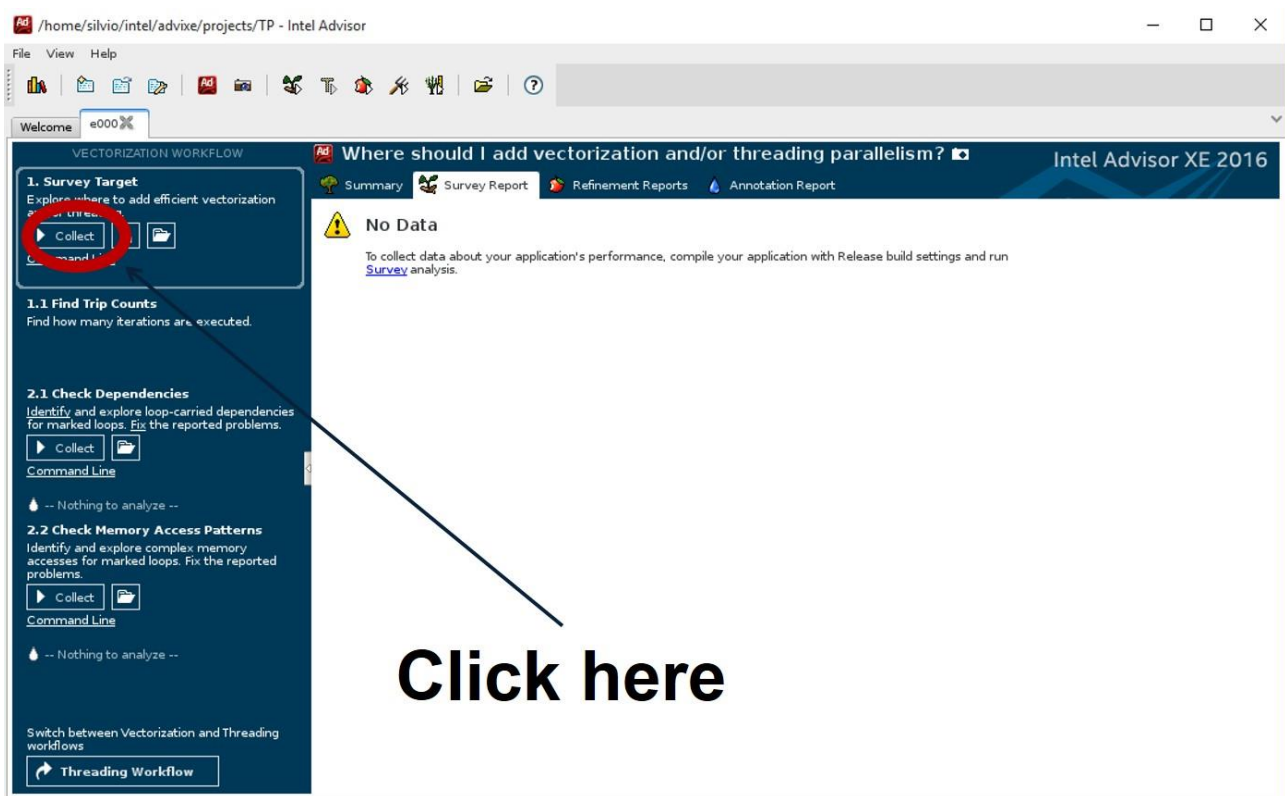


Figure 5. Survey Target Report.

## 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”;

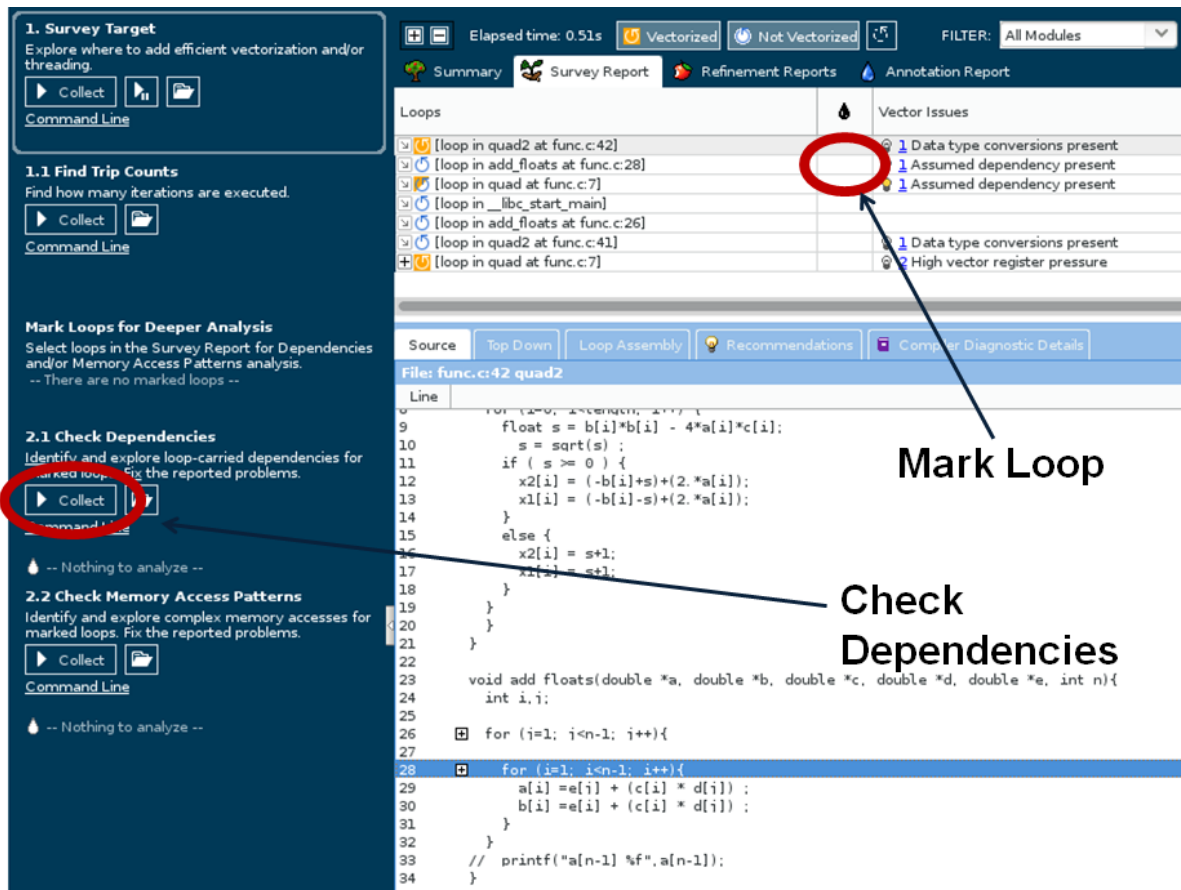


Figure 7. check dependency.

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

- Compile file stride.c

```
icc stride.c -o stride -g
```

- Run the application and get the execution time  
time ./stride
- Mark inner Loop of function main of file **stride.c** for deeper analysis “for(i=0; i<40000; i++)” line 23 ;
- Click on “check dependency”;
- Open the refinement reports (figure 8)

Note that the stride distribution is “constant stride”

**Check for loop-carried dependencies in your application**

Elapsed time: 4.04s | Vectorized | Not Vectorized | FILTER: All Module | All Source | Loops | All

Summary | Survey Report | **Refinement Reports** | Annotation Report

Site Location	Loop-Carried Dependencies	Strides Distribution	Access Pattern	Site Name
[loop in multiply0 at multiply.c:1...	✓ No dependencies found	No information available	No information available	loop_site_54
[loop in multiply0 at multiply.c:1...	No information available	50% / 50% / 0%	Mixed strides	loop_site_184

Memory Access Patterns Report | Dependencies Report | Recommendations

**Problems and Messages**

ID	Type	Site Name	Sources	Modules	State
P1	Parallel site information	loop_site_54	multiply.c	matrix.icc	✓ Not a problem

**Parallel site information: Code Locations**

ID	Instruction Address	Description	Source	Function	Variable references	Module	State
X1	0x4028be	Parallel site	multiply.c:181	multiply0		matrix.icc	✓ Not a problem

```

179     for(i=0; i<msize; i++) {
180         for(k=0; k<msize; k++) {
181             for(j=0; j<msize; j++) {
182                 c[i][j] = c[i][j] + a[i][k] * b[k][j];
183             }
184         }
185     }

```

Figure 8. Refinement Report Window.

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

Change the structure on file stride.c from AOS to SOA using the following code:

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

Change the body of inner loop on main function:

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

Change the body of second inner loop on main function:

```
for(i=0; i<40000; i++) {
    obj.x[i]= obj.y[i]+ obj.z[i] + randV;
}
```

Recompile the application and run the Check Memory Access Pattern again. Note that the stride distribution is unit stride now.

- Run the application and get the execution time again

```
time ./stride
```

Note that the performance may be improved.

### 13) Recompile application using AVX:

Recompile application using xhost

```
icc stride.c -o stride -g -O3 -xhost
```

Collect Survey Data again

Note that now the code was compiled using AVX