Implicit (automatic) Vectorisation

In this you will learn how to

- Enable and disable implicit (automatic) vectorisation.
- Prove that code has been vectorised by
  - Using the compiler vectorisation report
  - Using the compiler's assembler listing
- Generate vectorised code that is architecture agnostic

ACTIVITY 4-1: BUILDING THE EXAMPLE APPLICATION

In this activity you build a matrix multiplication application. At the heart of the application is a triple nested loop that does a matrix multiply using arrays of doubles (Tip: this information is important!).

Setting Up the Build Environment

1. Navigate to the folder Chapter4\4-5

2. If you are using Linux, you will need to comment -out the Windows-specific variables at the beginning of the Makefile and uncomment the Linux variables.

3. Open an Intel Parallel Studio XE command prompt.

- **Windows**

  Start | All Programs | Intel Parallel Studio 2013 | Command Prompt
  | Parallel Studio XE with Intel Compiler | Intel 64 Visual Studio

- **Linux**

To make the Parallel Studio XE tools available from a shell, source the following scripts (or add the commands to your ./bash_profile):

```
source /opt/intel/composerxe/bin/compilervars.sh  intel64
source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
source /opt/intel/inspector_xe/inspxe-vars.sh
```

Building and Running the Program

4. Build the application intel.O2.exe using the Intel compiler:

- Linux

```
make  clean
make  CFLAGS="-O2" TARGET=intel.O2
```

(Note : the CFLAGS a capital 'O' followed by a number)

- Windows

```
nmake  clean
nmake  CFLAGS="/O2" TARGET=intel.O2
```

5. Run the program `intel.O2.exe` and record the results in the table overleaf. Use the lowest time as the benchmark figure.

Note that if your CPU supports Turbo Boost Technology Mode, you may want to disable it in the BIOS. See your PC's handbook for instructions.

6. Repeat step 4, adding the CFLAG option /Qvec- (Windows) or -vec- (Linux) to disable the auto-vectorization (notice the minus sign at the end of the option).

- Linux

```
make  clean
make  CFLAGS="-O2 -vec-"  TARGET=novec
.\novec.exe
```

- Windows

```
nmake  clean
nmake  CFLAGS="/O2 /Qvec-" TARGET=novec
```

Run the new executable `novec.exe`, and record the results in the table below.

The two executables from steps 1 and 2 should run at different speeds.

TARGET	**Shortest time taken**
O2	
novec	

Calculate the speedup of the vectorised version, using the formula *Speedup = New Speed / Old Speed.*

```
Speedup =  ____________  / ____________  = ___________________
```

Is this the result you were expecting? (Hint, default vectorisation uses SSE2).

What reasons might there be for the speedup being different from what you expected?

<u>ACTIVITY 4-2: USING THE COMPILER TO PROVE THAT THE CODE HAS BEEN VECTORISED</u>

1. Investigate how vectorization differed in previous two builds by generating a vectorization report for both builds. To do this, add the option /Qvec-report2 (Linux: -vec-report2) to the CFLAGS.

e.g.

```
nmake  clean
nmake  CFLAGS="/O2 /Qvec- /Qvec-report2" TARGET=novec

nmake  clean
nmake  CFLAGS="/O2 /Qvec-report2" TARGET=intel.O2
```

Looking at `intel.O2.exe,` *how many lines where successfully vectorised?*

Make a list of the different reasons why loops were not vectorised in the table below:

Reasons why loop was not vectorised	**Location**

2. Repeat the previous step in this activity, but generate a much more verbose reports by using the option /Qvec-report6 (Linux: `-vec-report6`) in the CFLAGS.

e.g.

```
nmake  clean
nmake  CFLAGS="/O2 /Qvec-report6" TARGET=intel.O2
```

What new information is supplied in this report?

3. And yet miles more visibility …

NOTE this step uses an experimental feature of the compiler.

- Read the document `VecAnalysis Python_ Script.pdf` - which is in the ~/CLASSFILES/resources directory.
- Extract the python scripts ~/CLASSFILES/resource/vecanalysis.tgz.

```
$ tar xvzf ~/CLASSFILES/resource/vecanalysis.tgz
```

- Rebuild the application using -vec-report7.
- Following the instructions in `VecAnalysis Python_ Script.pdf` to create an annotated listing.

Which report level do you now consider to be the most useful?

4. **Examining the Disassembler**

Rebuild the application with the -S option - this will generate an assembler listing.

➢ Linux

```
make  clean
make  CFLAGS="-O2 -S" TARGET=intel.O2
```

➢ Windows

```
nmake  clean
nmake  CFLAGS="/O2 -S" TARGET=intel.O2
```

NOTE: You will get linker errors, but the assembler files will have been created.

First thing you'll notice is that the compiler generated ***an*** assembler file for each source file.

Look at the generated files. Can you see any evidence that the code has been vectorised?

(HINT: look for packed instructions in chapter4.asm/chapter4.s. Use the line number you recorded earlier to identify the correct place in the assembler file)

- ITS IMPORTANT YOU RENAME THE ASSEMBLER FILE to a **txt** extension ONCE YOU HAVE GENERATED IT.
- Once you have RENAMED the assembler file, DELETE any assembler files

➢ Linux

```
rm *.s
```

➢ Windows

```
del *.s
```

ACTIVITY 4-3: USING MORE ADVANCED VECTORISATION OPTIONS

Enhancing the Auto-Vectorization Options

1. Build and run the application using the /QxAVX option (Linux: -xAVX) and record the results below.

For example:

➢ Linux

```
make  clean
make  CFLAGS= "-O2 -xAVX "  TARGET=intel.AVX

.\intel.AVX.exe
```

➢ Windows

```
nmake  clean
nmake  CFLAGS= "/O2 /QxAVX "  TARGET=intel.AVX

intel.AVX.exe
```

Shortest Time Taken to run ______________________________

Did the code run faster? Was the increase in performance what you expected?

Building for AVX2

2. Build and run the application using a different AVX2:

- Linux

```
make  clean
make  CFLAGS= "-O2 -xCORE-AVX2 "  TARGET=intel.xAVX2

.\intel.xAVX2.exe
```

- Windows

```
nmake  clean
nmake  CFLAGS= "/O2 /QxCORE-AVX2 " TARGET=intel.xAVX2

intel.xAVX2.exe
```

What happens when you run the program? If it failed to run, can you suggest why?

Creating a Portable Application

3. Rebuild using the /Qax CORE-AVX2 (Linux: -ax CORE-AVX2) option:

- Linux

```
make  clean
make  CFLAGS= "-axCORE-AVX2 "  TARGET=intel.axAVX2

.\intel.axAVX2.exe
```

- Windows

```
nmake  clean
nmake  CFLAGS= "/QaxCORE-AVX2 " TARGET=intel.axAVX2

intel.axAVX2.exe
```

5. Run the program. The program should run fine, even if your CPU does not support AVX.

6. Spend time examining the output of the axAVX option. (use the –S option, or one of the vec-report options to get better visibility).

Can you explain the mechanism that the code uses to safely run?