Lab 1 - Back of the envelope calculations

Objectives

At the end of this lab you should be able to

- Carry out a 'back of the envelope' calculation to see if an application will benefit from running on an Intel® Xeon^{TM} Phi coprocessor.
- Build an application and run it natively on the coprocessor to confirm your calculations

The example application `pi.c` has four functions that you can use to se how well the Xeon Phi responds to parallelism, vectorisation and high bandwidths. Each of the functions is included or exclude from a build by the macros listed on column two.

Function	**Macro**	**Characteristic**
`CalcPi`	CALC_PI	Vectorised and parallelised
`CalcPiNoVec`	CALC_PI_NO_VEC	Not vectorised, but is parallelised
`CalcPiNoParallel`	CALC_PI_NO_PARALLEL	Vectorised but not parallelised
`CalcPiNoParallelNoVec`	CALC_PI_NO_PARALLEL_NO_VEC	Neither Vectorised nor Parallelised
`testBytes`	TEST_BYTES	High Bandwidth

Table 1: A list of functions in the code pi.

Each of the functions are included or exclude from a build by the macros listed on column two.

As you run through the lab you will be asked to 'guestimate' what the likely speed you will obtain by running the code on xeon phi.

Introduction

To make best use of an Intel® XeonTM Phi coprocessor, any application must be well parallelised and well vectorised. For good performance, your application should run over 90% parallel and be at least 90% vectorised.

Activity 1: A quick smoke test

1.1 Building and running the application on your workstation

Step 1. Build the application using the make file, and the run it from the command line to make sure it works and record the time each part of the program takes.

```
make clean

export OMP_NUM_THREADS=16

make step1

./step1.xeon
```

Function	Time Taken
`CalcPi`	
`CalcPiNoVec`	
`CalcPiNoParallel`	
`CalcPiNoParallelNoVec`	
`testBytes`	

Table 2: Results of running the code on the Xeon Workstation.

Step 2. Making a best guess at speed up

Given what you've learnt about how different kinds of code run on the Xeon Phi, make a best guess at the following Questions :

Function	Will it Run Faster on Xeon Phi (Yes or NO)?	Guesstimate much faster it will run
`CalcPi`	YES / NO	
`CalcPiNoVec`	YES / NO	
`CalcPiNoParallel`	YES / NO	
`CalcPiNoParallelNoVec`	YES / NO	
`testBytes`	YES / NO	

Table 3 : A first estimate of speedup

Step 3. Now build the application and run it on the Xeon Phi and record the results in Table 4.

```
make step3

ssh mic0 cd /CLASSFILES/lab-01

./run-mic.sh ./step3.mic

exit
```

Function	**Time Taken**	**Speedup (compared to Table 2)**
`CalcPi`		
`CalcPiNoVec`		
`CalcPiNoParallel`		
`CalcPiNoParallelNoVec`		
`testBytes`		

Table 4 : Results of running on Xeon Phi

Activity 2: Measuring Vectorisation

You can gauge the extent your application has been vectorised by running the application with and without vectorisation, and then comparing the difference in the runtime.

Note that

- This activity uses a cut-down version of the application and only includes the functions `CalcPi` and `CalcPiNoVec`.
- The number of iterations in the parallelised loops is reduced, so that you are not left waiting for a long time for the program to complete. This is controlled by the macro NUM_STEPS which is redefined in the Makefile.
- We reduce the Parallelism to 1 thread, so as to reduce any side-effect of having parallelism and vectorisation together.

Step 4: Build and run the application, recording the time the program takes to run. The makefile will add the option `-x`SSE2, to generate SSE2 instructions.

```
make step4

export OMP_NUM_THREADS=16

./step4.sse2.xeon
```

Elapsed Time ______________________________

Step 5: Now rerun the application, but set the number of threads to 1.

```
make step4

export OMP_NUM_THREADS=1

./step4.sse2.xeon
```

Elapsed Time ______________________________

Step 6: Now build the application. The makefile will add the options `-no-vec and -no-simd`, which will disable the auto-vectoriser in the compiler

```
make step6

export OMP_NUM_THREADS=1

./step6.novec.xeon
```

Elapsed Time ______________________________

Step 7: Calculate the Fraction of code that has been vectorised using the formula below:

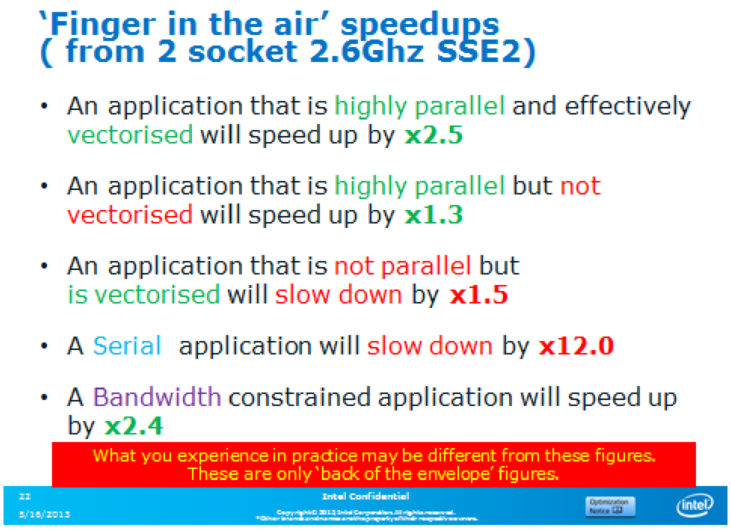
`Tv = ((T1 – T2)/Vector Length-1) * Vector Length` ______________________

`Vector Fraction = Tv/T1` ______________________

Where **T1** (from Step 6) is the time taken with one thread and vectorisation disabled, **T2** (from step 5) is the time taken with one thread and vectorisation enabled.

Vector Length is the number of floating point values that can fit in the 128 bit SSE registers - Since the program uses doubles, then the Vector Length is 2.

Step 8: Now guesstimate what the speed of application will be when running on a Xeon Phi, by applying the 'finger in the air' rules of thumb to the runtime values of ***step 4.***


‘Finger in the air’ speedups (from 2 socket 2.6Ghz SSE2)

- An application that is highly parallel and effectively vectorised will speed up by **x2.5**
- An application that is highly parallel but not vectorised will speed up by **x1.3**
- An application that is not parallel but is vectorised will slow down by **x1.5**
- A Serial application will slow down by **x12.0**
- A Bandwidth constrained application will speed up by **x2.4**

What you experience in practice may be different from these figures. These are only ‘back of the envelope’ figures.

22 Intel Confidential
5/16/2013

Calculate the expected speedup by

- Applying the first rule to the vectorised portion of the code, (which is also highly parallel).
- Applying the second rule to the non-vectorised part of the code.

Step 9: Now build the application and run it on the Xeon Phi and record the results .

```
make step9

ssh mic0 cd /CLASSFILES/lab-01

./run-mic.sh ./step9.mic

exit
```

Questions:

1. How accurate was the Guesstimate? Can you suggest any reasons for the inaccuracies?
2. Examine `pi.c`. Does the code look ‘normal’, or does it have any coprocessor-specific parts?
3. Look at the makefile. Were there any extra compiler options used to enable the application to be suitable for running on an Intel® Xeon™ Phi?

Activity 3: Measuring Concurrency

1.2 Measuring concurrency

Step 10. You can use use Amplifier XE to see how parallel your code is by running a concurrency analysis using the command line version of Intel ® VTune™ Amplifier XE.

Note we are using the application built in Step 1.

Do the following (Your results will look be similar to those shown in Figure :

```
export OMP_NUM_THREADS=16
amplxe-cl -c concurrency -- ./step1.xeon
```

Concurrency ____________________

Logical CPU Count ________________

Elapsed Time ____________________

```
Result Info
-----------
Parameter                 r000cc
------------------------  ---------------------------------------------------------------
------------
Application Command Line  ./pi
CPU Name                  Intel(R) Xeon(R) E5 processor
Computer Name             snbws3
Environment Variables
Frequency                 3100000000
Logical CPU Count         16
MPI Process Rank
Operating System          2.6.32-279.el6.x86_64 Red Hat Enterprise Linux Server release 6.
3 (Santiago)
Result Size               1371932
User Name

Summary
-------
```

Figure 2: A sample output from running a concurrency analysis

Step11 (optional). You can work out the parallel efficiency of your code 'by hand' rather than using VTune Amplifier XE. To do this re-run the application using just 1 thread and record the results.

```
export OMP_NUM_THREADS=1
./step1.xeon
```

Time Taken _____________________

Now work out the parallel and serial time, where **T1** is the time taken with one thread (from this step), **T2** is the time taken with 16 threads (from step 1):

Tp = ((T1 – T2)/Num Cores-1) * Num Cores _____________________

Parallel Fraction = (Tp/T1) _____________________

Parallel Time = Time Taken * Parallel Fraction _______________________

Serial Time = Time Taken - Parallel Time _______________________

Questions

The answer may be slightly different from what you calculated in Step 10, can you suggest any reasons why?