Vectorisation Exercise

Adrian Jackson

1st July 2019

1 Introduction

In these exercise we are going to investigate the vectorisation performance on the ARCHER nodes. We will use the optimisation reports from the Intel compiler to tell us about how well the compiler is vectorising things. This is done by using the compiler flag `-qopt-report=5`. For every source file that is compiled this will produce a text file called *.optrpt (where * o. This will have vectorisation reports of every loop in the file.

To get started with the exercises copy the followinf file:

```
/home/z01/shared/VectorSoure.tar.gz
```

You should be able unpack it with the command `tar zxvf VectorSource.tar.gz`

We are going to use the Intel compiler for these exercises so make sure you have it loaded. You can check with the command `module list` which will show you the modules you currently have loaded. You should have the `PrgEnv-intel` module loaded, if you don't unload the current programming environment (`module unload PrgEnv-cray`) and load the Intel one (`module load PrgEnv-intel`).

2 Auto Vectorisation

2.1 Code restructuring

The exercise explores the automatic vectorisation the Intel compiler will perform on loops in a simple code, and the reports it gives about what it has done. There you can choose either a C or Fortran example to examine and experiment with. The C and Fortran examples are slightly different. Choose either C or Fortran and then move into the `restructure` directory (i.e. `VectorisationSource/C/restructure` or `VectorisationSource/Fortran/restructure`). For this exercise we aren't going to run the code on the compute nodes, just compile and investigate the compiler optimisation reports.

2.1.1 C

1. Go into the C directory and compiler the code by typing `make`.
2. Read the optimisation report that has been produced (`vectorisation.optrpt`). We are primarily interested in the core computational kernel, the routine `vector_add`. Has the compiler managed to vectorise the code?

3. Add the `restrict` keyword to the function arguments of type `float *` in `vector_add`, re-compile and examine the optimisation report for the file again. Has it changed?
4. Now align the array allocates and re-compile. Examine the optimisation report, is it fully vectorising now?
5. Finally, tell the compiler that the arrays have been aligned (this can be done in a number of different ways, see the lecture slides), re-compile and examine the optimisation reports.

2.1.2 Fortran

1. Once in the Fortran directory compile the code by typing `make`.
2. Read the optimisation report that has been produced (`vectorisation.optrpt`). We are primarily interested in the core computational kernel, the routine `vector_add`. Has the compiler managed to vectorise the code?
3. Align the array allocates and re-compile. Examine the optimisation report, is it vectorising now?
4. Tell the compiler that the arrays have been aligned (this can be done in a number of different ways, see the lecture slides). Re-compile and examine the optimisation reports.
5. Now try using array notation for the `vector_add` kernel. Re-compile and view the vectorisation reports? Has this changed anything?

2.2 Bad Compiler Advice

In this exercise we explore what happens if you tell the compiler a loop can be vectorised that does not follow the restrictions required for vectorisation. The code to use for this exercise is in the `badadvice` directory. For this exercise we **are** going to run the code on the system, we have provided you a batch script (runbadprog.sh for the C example and runvect.sh for the Fortran one) to do this. We have also provided a makefile to build the code.

2.2.1 C

1. Compile the code, read the optimisation reports, is the external function being vectorised?
2. Run the code by submitting a batch job (i.e. `qsub runbadprog.sh`). Look at the output that is produced.
3. Add a compiler pragma above the loop in the `simple_assign` function to force the compiler to vectorise the loop. Re-compile and check the optimisation reports to make sure the loop is now being vectorised (note, we have not fixed the alignment so we will not get perfect vectorisation in this example).
4. Re-run the code on the system and compare the output to the original code. Are there differences between the results for this run and the original run?
5. Add the `-no-vec` flag to the makefile, re-compile and re-run to make sure the difference in results is caused by the vectorisation.
6. What is the vectorisation doing in this example?

2.2.2 Fortran

1. Compile the code, read the optimisation reports, is the `vector_add_accum` subroutine being vectorised?
2. Run the code by submitting a batch job (i.e. `qsub runvect.sh`). Examine the output that is produced.
3. Now, force the compiler to vectorise the loop inside the `vector_add_accum` routine by adding this directive before the loop:

```
!dir$ simd
```

4. Re-compile, check the optimisation reports to ensure it is actually vectorising that loop, and re-run the code on the system. Compare the output you get now to the original output, is it the same?

3 Explicit vectorisation

As well as using the compiler to identify and exploit vectorisation, it is possible to manually specify the vectorisation you require for the application. This can be done using vector intrinsics, that match specific vector instructions, or by using higher level programming techniques, such as the OpenMP simd directives.

In this exercise we will focus on using the higher level programming approaches, i.e. OpenMP simd.

3.1 OpenMP simd

Revisit the first exercise you undertook, the restructure exercise, but implement the vectorisation using OpenMP simd directives rather than asserting alignment to the compiler. There is a new directory, `simd`, with the original code for you to vectorise using the OpenMP simd command.

Note, whilst OpenMP simd will force the compiler to vectorise, you still are responsible for ensuring that arrays used are aligned, otherwise you may get incorrect results or the application may crash.