Vectorization Lab
High Performance Computing on Stampede

Aaron Birkland
Cornell Center for Advanced Computing

Jan 14, 2014

1 Simple Vectorization

This exercise serves as an introduction to using a vectorizing compiler. We will work with code containing a tight loop that should be easily vectorizable. Our goal is to try out various compiler options and compare vectorized with non-vectorized code on Stampede.

1. Unpack the lab materials into your home directory, and change into the `vector` directory.

   ```
   $ cd
   $ tar xvf ~tg459572/LABS/vector.tar
   $ cd vector
   ```

2. We noted that the Intel compiler starts applying vectorization with `-O2`. Let's see if we can view a vectorization report to see what it did.

   ```
   $ icc simple.c -vec-report=2 -O3 -o simple
   simple.c(19): (col. 2) remark: LOOP WAS VECTORIZED.
   simple.c(26): (col. 3) remark: LOOP WAS VECTORIZED.
   simple.c(25): (col. 5) remark: loop was not vectorized: not inner loop.
   ```

   This shows that *two* loops were vectorized: The initial value loading loop, and our computation loop.

3. Now that the compiler has told us that it vectorized our loops, let's verify this by compiling with vectorization disabled.

   ```
   $ icc simple.c -no-vec -vec-report=2 -O3 -o simple_no_vec
   ```

Notice that all the vectorization reports disappeared, even though we specified reporting as a compile option. When vectorization is disabled, the reports disappear.

4. As mentioned in the talk, the Intel compiler will use SSE (128-bit) instructions by default. Compile the code with vectorization enabled, but add the argument `-xAVX` to the compilation flags to use 256-bit AVX. Name your executable `simple_avx`.

5. Now compile vectorized and non-vectorized variants of the code to run natively on the MIC coprocessor. Use the compile flag `-mmic` to compile for the MIC architecture.

```
  $ icc simple.c -mmic -O3 -o simple.mic
  $ icc simple.c -no-vec -mmic -O3 -o simple_no_vec.mic
```

6. The `simple.sh` batch file will record the execution time each of our vectorized and non-vectorized applications. Take a look at the batch script, then run it and examine the output.

```
  $ sbatch simple.sh
  $ cat slurm-951653.out
simple_no_vec: 0.67
simple 0.37
simple_avx 0.25
simple_no_vec.mic 13.22
simple.mic 2.78
```

7. Lastly, the intel compiler flag `-xhost` can be used to automatically detect all the advanced features of the hardware (like AVX). The downside is that the resulting binaries may only be run on machines with an architecture similar to Stampede (i.e. with 256-bit AVX instructions). Try compiling with `-xhost` and see if the runtime is similar to the `-axAVX` example from before.

As we have seen, vectorization on the Intel compiler can be simple and straightforward. Correlating vectorization reports with the source code can be a little bit tricky, especially if the compiler implements optimizations such as loop reordering. However, as long as we have some sense of what the compiler ought to be doing, this can usually be figured out with a little effort.

2 Assisted Vectorization

This lab involves code that contains a data dependency. We will use this to further explore vectorization reports, then use directives to override the compiler's default behaviour.

2.1 Advanced Vector Reports

We will use vector reports to examine problems the compiler is having when trying to vectorize code.

- First, load the latest intel compiler

  ```
  $ module load intel/14.0.1.106
  ```

- Compile the `dependency` program

  ```
  $ icc -xhost -O3 -vec-report=2 dependency.c -o dependency
  ```

  In the report, you will see that the compiler has vectorized some loops, but not others. Pay particular attention to the line regarding data dependency:

  ```
  dependency.c(33): (col. 2) remark: loop was not vectorized: existence of
  vector dependence.
  ```

- Try to compile with different vectorization report options. The Intel compiler expects values ranging from 0 to 5. They do not necessarily progress in order of detail. Try each level and note the differences. Is any report level particularly enlightening?

  For example, trying option 4 might look like:

  ```
  $ icc -xhost -O3 -vec-report=4 dependency.c -o dependency
  dependency.c(33): (col. 2) remark: loop was not vectorized: not inner loop.
  dependency.c(33): (col. 2) remark: loop was not vectorized: existence of
  vector dependence.
  dependency.c(47): (col. 6) remark: loop was not vectorized: not inner loop.
  dependency.c(48): (col. 4) remark: loop skipped: multiversioned.
  ```

The vector report listed several ANTI and FLOW dependencies around line 33. In the code, this is the line where `compute()` is called. Can you guess why the compiler chose line 33? Also, why did the compiler find multiple kinds of dependencies?

2.2 Compiler directives

We will now use compiler directives to force the compiler to assume there is no data dependency in our loop.

- `dependency_pragma.c` is identical to our original dependency code, except for the addition of `#pragma` directives. Look at the source and find any directives.

- Compile the `dependency_pragma` code:

  ```
  $ icc -xhost -O3 -vec-report=3 dependency_pragma.c -o dependency_pragma
  ```

  Compare the vectorization reports of `dependency` vs `dependency_pragma`. Do you notice where `loop was not vectorized` has been replaced by `LOOP WAS VECTORIZED`?

- Run `dependency` and `dependency_pragma` to see if the vector hints increased performance:

  ```
  $ time dependency
   Given value of 0
   Sum is: 724215229516.70

   real    0m.161s
   user    0m.161s
   sys     0m0.001s


   $ time dependency_pragma
   Given value of 0
   Sum is: 421476947100.00

   real    0m0.067s
   user    0m0.067s
   sys     0m0.001s
  ```

  We more than doubled our performance by allowing our inner loop to be vectorized!

  Notice that the `Sum` results are quite different! Our program gave a significantly different result with vectorization enabled. This is because our loop to exhibits a "read-after-write" or "flow" dependency. In `dependency_pragma`, the compiler was told to ignore the possibility of dependencies, so it produces an incorrect result. This is why correcting the compiler by way of compiler directives can be tricky - you need to be absolutely sure that vectorization will not produce a mathematically incorrect results if you tell it to ignore dependencies.