Vectorization Lab
Parallel Computing at TACC: Ranger to Stampede Transition

Aaron Birkland
Consultant
Cornell Center for Advanced Computing

December 11, 2012

1 Simple Vectorization

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

1.1 Setup

To begin, we will unpack the lab materials and compile the example program. Please run this lab on Lonestar, and not Ranger.

1. Make sure you are using the bash shell. Do

   ```
   echo $SHELL
   ```

   you should see `/bin/bash`. If not, run `bash` to start the bash shell.

2. Unpack the lab materials into your home directory if you haven't done so already.

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

3. Compile `simple` using the Intel compiler. We will start with an optimization level of 2, which should enable vectorization

   ```
   $ icc simple.c -O2 -o simple
   ```

4. Use the *time* utility to run the program with no arguments. Normally, it would be proper to submit the executable to be run on a compute node via the batch system. For logistic purposes, and because the runtimes are so fast, we will be running the quick examples on the login node directly.

```
$ time ./simple
  1000100.000003 .. 2047102400.004084

  real    0m0.256s
  user    0m0.256s
  sys     0m0.000s
```

   This shows that it took a quarter second to execute.

If you've reached this point, then the lab is set up correctly, and everything is working enough to continue.

1.2 Intel Compiler

We will now explore the effects of vectorization using the Intel compiler

- 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 -O2 -o simple
simple.c(19): (col. 2) remark: LOOP WAS VECTORIZED.
simple.c(26): (col. 3) remark: loop was not vectorized: not inner loop.
simple.c(25): (col. 5) remark: PERMUTED LOOP WAS VECTORIZED.
```

  This shows that *two* loops were vectorized: The initial value loading loop, and our computation loop. However, the line numbers and comments look strange. Why does it say the inner part of our loop (line 26) was not vectorized because of "not inner loop", while our outer loop (line 25) *was* vectorized? Why does it refer to our outer loop as a PERMUTED LOOP? Compilers are free to reverse the order of loops for the sake of efficiency if it is safe to do so. Do you think this was the case here?

- 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 -O2 -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.

- Run the non-vectorized program and compare execution time to our original compiled with `-O2`

```
$ time simple_no_vec
1000099.999977 .. 2047102400.017378

real    0m1.391s
user    0m1.360s
sys     0m0.004s

$ time simple
1000100.000003 .. 2047102400.004084

 real    0m0.264s
 user    0m0.256s
 sys     0m0.000s
```

  Wow, quite a difference! How much of a speedup did you observe? How does it compare to your expectations?

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.

1.3 GCC compiler

Different compilers can also vectorize code. Here, we try to compile the same code using the GCC.

- Compile the `simple` program with `-O2` and run

```
$ gcc -O2 simple.c -o simple_gcc
$ time ./simple_gcc
1000099.999977 .. 2047102400.017378

real    0m1.460s
user    0m1.392s
sys     0m0.020s
```

  We see that this is similar to the Intel non-vectorized case. In fact, GCC does not vectorize by default. Special flags are needed to enable vectorization

- Use the `-ftree-vectorize` and -ftree-vectorizer-verbose flags to enable GCC to vectorize and report

```
$ gcc -O2 -ftree-vectorize -ftree-vectorizer-verbose=3 simple.c -o simple_gcc_vec

simple.c:19: note: not vectorized: unsupported use in stmt.
simple.c:26: note: accesses have the same alignment.
simple.c:26: note: accesses have the same alignment.
simple.c:26: note: LOOP VECTORIZED.
simple.c:26: note: vectorized 1 loops in function.


$ time ./simple_gcc_vec
1000099.999977 .. 2047102400.017378

real    0m0.578s
user    0m0.528s
sys     0m0.008s
```

As we can see, the vectorized version is much better. It's not quite as good as the Intel version, however. Can you tell any differences by comparing the vectorization reports?

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.

- Compile the `dependency` program

  ```
  $ icc -O2 -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 -O2 -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 two `#pragma` directives. Look at the source and find them.
- Compile the `dependency_pragma` code:

  ```
  $ icc -O2 -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    0m1.428s
   user    0m1.144s
   sys     0m0.036s

   $ time dependency_pragma
   Given value of 0
   Sum is: 724215229516.70

   real    0m0.746s
   user    0m0.664s
   sys     0m0.012s
  ```

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

- The `dependency` and `dependency_pragma` programs can accept an integer command line argument. This is used to supply the value of the `k` array offset in our main loop. Start out by providing the value −1.

```
$ time dependency -1
Given value of -1
Sum is: 723293729503.86

real    0m6.008s
user    0m5.020s
sys     0m0.240s

$ time dependency_pragma -1
Given value of -1
Sum is: 723293158864.15

real    0m3.159s
user    0m2.684s
sys     0m0.204s
```

Notice that the `Sum` results are quite different! Our program gave a significantly different result with vectorization enabled. This is because a value of `-1` causes our loop to exhibit 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. Do any other values produce differing results?

3 Evaluating Assembly and Profiling

In this section, we will briefly look at evaluating vector assembly instructions produced by the compiler, as well as characterize the performance differences between applications with non-unit stride.

3.1 Profiling

TACC provides an excellent tool called PerfExpert for profiling applications. It can give an easy to use picture of how effectively an application is accessing data. We won't go into it in much detail, but will use it to quickly glance at stride-1 vs stride-N access.

- Compile the `stride` and `stride_bad` programs

```
$ icc -O2 stride.c -o stride
$ icc -O2 stride_bad.c -o stride_bad
```

- Load the PerfExpert module

```
$ module load java papi perfexpert
```

- Submit the stride program for profiling via `perf_job.sh`.

```
$ qsub perf_job.sh
```

  Note that this script actually launches an executable called `perfexpert_run_exp`, with our executable as an argument. This is the profiler. Results will be written to the current directory.

- Wait until the results are available. These results will be written to a file named `experiment-PerfExpert.<jobID>.xml`. Run PerfExpert to display an analysis. Give it the arguments `.1` and the name of the experiment xml file. This will direct PerfExpert to generate a profiling report for all routines that consume at least 10% of the execution time.

```
$ perfexpert .1 experiment-PerfExpert.o836399.xml
```

  You should see a report like the following. Note the GFLOPS section where it reports packed (vector) and scalar computations. The performance assessment below that is expressed in terms of clock cycles per instruction. High values means that more clock cycles are being required to execute each instruction.. For L1, L2, or L3 caches, high LCPI values means that clock cycles are being wasted while waiting for data to be transferred from memory.

```
  Input file: "experiment-PerfExpert.o836399.xml"
Total running time for "experiment-PerfExpert.o836399.xml" is 1.884 sec

Loop in function main() (100% of the total runtime)
===============================================================================
ratio to total instrns        %  0.........25..........50.........75........100
   - floating point      :   21 **********
   - data accesses       :   29 **************
* GFLOPS (% max)         :   28 *************
   - packed              :   28 *************
   - scalar              :    0 *
-------------------------------------------------------------------------------
performance assessment     LCPI good......okay......fair......poor......bad....
* overall                :  1.8 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound estimates
* data accesses          :  8.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L1d hits            :  1.3 >>>>>>>>>>>>>>>>>>>>>>>>>
```

```
   - L2d hits           :  0.0 >
   - L2d misses         :  0.8 >>>>>>>>>>>>>>>>>
   - L3d misses         :  6.5 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses  :  0.1 >
   - L1i hits           :  0.1 >
   - L2i hits           :  0.0 >
   - L2i misses         :  0.0 >
* data TLB              :  0.1 >>
* instruction TLB       :  0.0 >
* branch instructions   :  0.2 >>>
   - correctly predicted :  0.1 >>>
   - mispredicted       :  0.0 >
* floating-point instr  :  0.4 >>>>>>>>
   - fast FP instr      :  0.4 >>>>>>>>
   - slow FP instr      :  0.0 >
```

This report means that our stride 1 example performs decently at about 28GFlops, but doesn't have the best data access patterns.

- Take a look at `stride_bad.c`. Notice that we change the order of array indexes in our inner loop. Previously, we were using an effecient stride 1 construct. With this change, we are using stride >> 1. We change

  ```
  a[i][j] += b[i][j];
  ```

  to

  ```
  a[j][i] += b[i][j];
  ```

- Next submit the `stride_bad` program and view its results with PerfExpert

  ```
  $ qsub perf_job_bad.sh
  ```

  Then when the results are ready

  ```
  $ perfexpert .1 experiment-PerfExpert.o836400.xml
  ```

  You'll see that the results are much worse! We went down from 28 GFlops to 3, even though all computations are still vectorized(packed). Most of the CPU time is wasted with cache misses. This shows us that poor data access patterns can completely negate any benefits of vectorization.

3.2 Evaluating Assembly

While assembly language is sometimes cryptic c and hard to understand, it can be useful to glance at the assembly instructions of a vectorized program to verify that a loop has vectorized well. In particular, it is helpful to get a quick sense of how many aligned vs unaligned instructions are being issues, or look for occurrences of scalar instructions in sections that ought to be vectorized.

3.3 A look at assembly

The gnu binutils package contains some very useful utilities for examining the assembly of binaries. In this exercise, we'll look at `objdump` to dump an executable's assembly instructions and correlate them with the source code.

- Compile the `stride` program. This performs simple addition between two dimensional arrays. We will compile with the `-g` option to include debugging symbols in the binary. This will allow us to correlate the code with the assembler instructions.

  ```
  $ icc -g -O2 stride.c -o stride
  ```

- Use `objdump` to dump the assembly. We will dump it to a file in order to look at it:

  ```
  $ objdump -S stride > stride.out
  ```

- Open the file stride.out, and look for an occurrence of `sum_elements()` in the text next to a block of assembly instructions

  ```
          sum_elements();
  4009de:       0f 28 04 c5 00 46 60    movaps 0x604600(,%rax,8),%xmm0
  4009e5:       00
  4009e6:       66 0f 58 04 c5 00 4e    addpd  0x300f4e00(,%rax,8),%xmm0
  4009ed:       0f 30
  4009ef:       0f 29 04 c5 00 46 60    movaps %xmm0,0x604600(,%rax,8)
  4009f6:       00
  4009f7:       48 83 c0 02             add    $0x2,%rax
  4009fb:       48 3d 00 e1 f5 05       cmp    $0x5f5e100,%rax
  400a01:       72 db                   jb     4009de <main+0x15e>
  ```

  This is the content of our primary loop. In terms of assembly, this looks "good". We see simple, aligned vector instructions that bear some resemblance to our task. If we instead saw `movups`, we would know that the data is unaligned. Likewise, `addsd` would imply scalar addition rather than vector addition.