PHY905, section 4
Homework Assignment 5

Håkon V. Treider
MSU id: treiderh

March 28, 2016

Introduction

The tests have been run on Mac OS X (10.11.4) with a 2.5 GHz Intel Core i7.

In this homework assignment, I am going to look at the performance gained by vectorization of code, in particular for-loops. The operation I will use is the orthogonalization of a vector w with respect to a set of k vectors $v_0, v_1, ..., v_{k-1}$.

I will also inspect how well my compiler, Clang, recognizes and vectorizes automatically.

Performance model

As instructed in the assignment text, I will consider cache to be "infinitely" fast and only consider read/writes from memory as well as floating point operations. In the simplest approach ($f^1(c, r)$), we repeat the process for each vector v_i. In the more advanced approach ($f^2(c, r)$), we vectorize "by hand" by computing all k (i.e. 4 `floats` or `ints` at the time) vectors in one loop, maximizing temporal locality for w_i (thus helping the compiler). Counting carefully, we get the following results:

$$f^1(c, r) = k(5r + 4c)n \tag{1}$$

$$f^2(c, r) = \frac{k}{4}(11r + 10c)n \tag{2}$$

Using $k = 4$, $c = 0.2 \cdot 10^{-9}$ and $r(= w) = 0.5 \cdot 10^{-9}$ we have the ratio of the two methods:

$$r = \frac{f^1}{f^2} = \frac{4(5r + 4c)}{11r + 10c} \approx \underline{1.76} \tag{3}$$

Test results

I downloaded the given program and found the following compiler flag gives great insight into what the compiler manages (and don't manages) to vectorize on it's own:
`-Rpass-analysis=loop-vectorize`
The Clang compiler on newer Macs have the vectorization on by default, but this can easily be switched off by compiling with the extra argument:
`-fno-vectorize`
If you combine these two, it will simply state that no loop where vectorized, even though it recognizes all of them as potential candidates. Running the code, as is, with different level of optimization and flags, I get the results listed in table 1.

Let's compare the computation cost of "separate operations" versus the rest for test run (2). Comparing with the routine for applying the four vectors simultaneously we get a ratio: $r = 4.07/1.48 \approx 2.75$. This is an impressive speedup, and basically confirms that our 4 floats are vectorized and computed at the same time. We do of course not get a speedup of 4X, since we then have to apply these back into `w[i]` with a loop over n.

Comparing with the results of using the same *routine*, just in a separate file gives $r = 4.07/2.74 \approx 1.49$, which is more in line with my expectation (model). Here I suspect no vectorization, but rather compiler optimization is the main difference between the two. It seems the compiler can't decide if the data sent out isn't "tampered" with, so it can't risk vectorizing it in order to not give potentially wrong results (say there were a dependance on previously computed values).

The next ratio out, which by all means is the same as the previous, the "w+1" tests to see if there is a small/big penalty in speed with data that is not "naturally aligned". As answered by professor William Gropp on the course web page: *It used to be that (almost) all architectures had a big penalty for this. But since programmers rarely paid attention to this, some chip designers devoted more transistors and power to eliminating that penalty.* This is also what we see in the computation time, no penalty.

The restrict attribute gives no additional speedup, as my compiler doesn't need it to exploit it. *A restricted pointer references data that is not referenced through any other pointer.* However, larger (or other systems) like Blue Waters apparently can have huge speedups by setting this attribute, as seen in the lecture 13.

Some last thoughts about one of the other test run, namely (1). The only way our compiler is able to vectorize the "poorly" written loop, i.e. the "separate operations" in the same file, is to allow *aggressive optimization*. This is not guaranteed to give correct results, but does so in this case where the dependency

Table 1: Computation cost per array element computed [10^{-9} sec]

Test run	Comp. flags	n	sep op	in file	sep file	w+1	restrict
(1)	-Ofast (aggressive)	10000	1.20	1.46	2.79	2.81	2.68
(2)	-O3	10000	4.07	1.48	2.74	2.71	2.73
(3)	-O3 -fno-vectorize	10000	5.45	2.26	2.87	2.84	2.70
(4)	-O0	10000	20.0	7.61	9.86	9.64	9.67

is simple. What's even more interesting, is that this is even slightly faster...! Not much, but on average 20%. In all but the "no compiler optimization"-level cases the results from the routines in "separate file" is the same. This tells us something important about what the compiler is able to do with respect to both vectorization and optimization techniques!