# Designing and building applications for extreme scale systems
# CS598 SP2016
# HW5: Temporal Locality and Vectorization

**Goals**

- Use performance estimates to design a faster orthogonalization operation
- Explore vectorization of code
- Discover how well your compiler vectorizes

**Tasks**

**Part1**

In some iterative methods, the following operation is used. Given a collection of k vectors $v_i$ and a new vector w, compute

```
for i=0…k-1
    $w = w - v_i^T w\ v_i$
```

This is typically accomplished by making two routine calls for each vector $v_i$ : one to compute the dot product of w and $v_i$ and one to subtract that value times $v_i$ from vector w. The C code for this looks like (using "v" for any of the $v_i$):

```
s0 = 0;
for (i=0; i<n; i++) {
      s0 += w[i] * v[i];
}
for (i=0; i<n; i++) {
      w[i] = w[i] - s0 * v[i];
}
```

<u>Using the simple performance model (no cache effects), estimate the time for this operation as a function of compute time c, and memory access time m</u> (use the same time for either read from or write to memory). The cost of k such operations will be k times your estimate.

Because there are potentially many vectors $v_i$, one way to reduce the number of memory accesses is to exploit temporal locality for w – not just in cache but also in register. Estimate the time for the following code that applies 4 vectors v (specifically, v0, v1, v2, and v3) at a time:

```
s0 = 0, s1 = 0, s2 = 0, s3 = 0;
for (i=0; i<n; i++) {
        wi = w[i];          // A good compiler might do this for you
        s0 += wi * v0[i];
        s1 += wi * v1[i];
        s2 += wi * v2[i];
        s3 += wi * v3[i];
}
for (i=0; i<n; i++) {
        w[i] = w[i] - s0 * v0[i] - s1 * v1[i] - s2 * v2[i] - s3 * v3[i];
}
```

<u>Assume k is divisible by 4, what is your time estimate to apply all k vectors $v_i$? Finally, estimate the ratio of times for the two methods. Report that value.</u>

## Part2

In part 2, you will experiment with a code that explores the performance of different options. It also illustrates some of the things that can affect code performance, and allows you to explore your compiler's capabilities.

The tarball hw5.tgz (compressed; use tar xzf hw5.tgz to expand) contains three files: a Makefile and two source files. The two source files, morth1.c and morthr.c, contain a C program that implements the operation above. The makefile "Makefile" can be used to build the program with several compilers, but you may need to modify if for your environment. <u>Read the program and make sure you understand what it is doing.</u> <u>Then run it (using a high level of optimization) and report the results.</u> <u>Try to get a report from your compiler about how well it vectorized the code</u> (the Makefile provides flags that do that for several popular compilers).

Compare the ratios of the "sep op" column (for "separate operation") with the other 4 columns. <u>Discuss each result.</u> <u>Is it what you expected based on part 1? If not, why not?</u>

## Submit

You should submit a PDF file with 2 estimate times in Part 1; running results and report on vectorization, also the discussion.

## For more fun (but not to turn in)

1. If your compiler does *not* vectorize the routines, see if you can figure out why. Are the directives or other information that you can provide the compiler that will cause it to vectorize?
2. Every compiler is different (even versions of the same compiler). Try a different compiler and/or a different system. What changes?
3. Note that the operation that applies multiple vectors to w at once is not numerically identical (in floating-point arithmetic) to the value computed by

the original method. Compare the computations for some different choices of vectors w and $v_i$.

4. This example does 4 vectors at a time. Use your performance estimate to consider 2, 6, 8, and 16 at a time. How much of a benefit do you predict in going from 2 to 4 to 6 to 8 to 16?