PHY905, section 4
Homework Assignment 3

Håkon V. Treider
MSU id: treiderh

February 10, 2016

Introduction

The tests have been run on Mac OS X (10.11.3) with a 2.5 GHz Intel Core i7.

In this assignment I am going to test different approaches to the matrix transpose algorithm. First out is a basic code written without *any* optimisations in mind. Then I move on to make 16-by-16 blocks being transposed, one-by-one in "blocked" loops. After that I will run the supplied code "cotranspose.c" from the course website. Lastly, I will also test the built-in transpose function of my linear algebra library, Armadillo.

Objectives

1. Observe the impact of a lack of spatial locality
2. Experience using blocking of loops to improve locality
3. Explore the use of the cache oblivious approach to formulate algorithms

Results

The results can be seen in table 1 and visualized in a standard and loglog plot in figure 1.

It is a clear cut victory for the cache oblivious transpose code, especially for large matrices. However, the much simpler "blocked code" runs surprisingly well and follows closely behind. For the smallest matrices, the differences are minuscule and differences might stem from inaccurate time measurement as well as actual performance.

Table 1:

Size	Basic	Model	Blocked	Cache Oblivious	Armadillo
20	6.8e-07	2.8e-07	7.3e-07	1.4e-07	4.4e-07
100	9.7e-06	7.0e-06	9.3e-06	0.000008	4.9e-06
500	0.00037	0.00017	0.00037	0.00042	0.00024
800	0.0016	0.00045	0.00081	0.00069	0.00093
1000	0.0023	0.00070	0.0013	0.0012	0.0017
1500	0.018	0.0016	0.0046	0.0042	0.018
1750	0.024	0.0021	0.0060	0.0050	0.024
2000	0.034	0.0028	0.0069	0.0057	0.037
3000	0.089	0.0063	0.034	0.017	0.087
4000	0.16	0.011	0.061	0.047	0.17

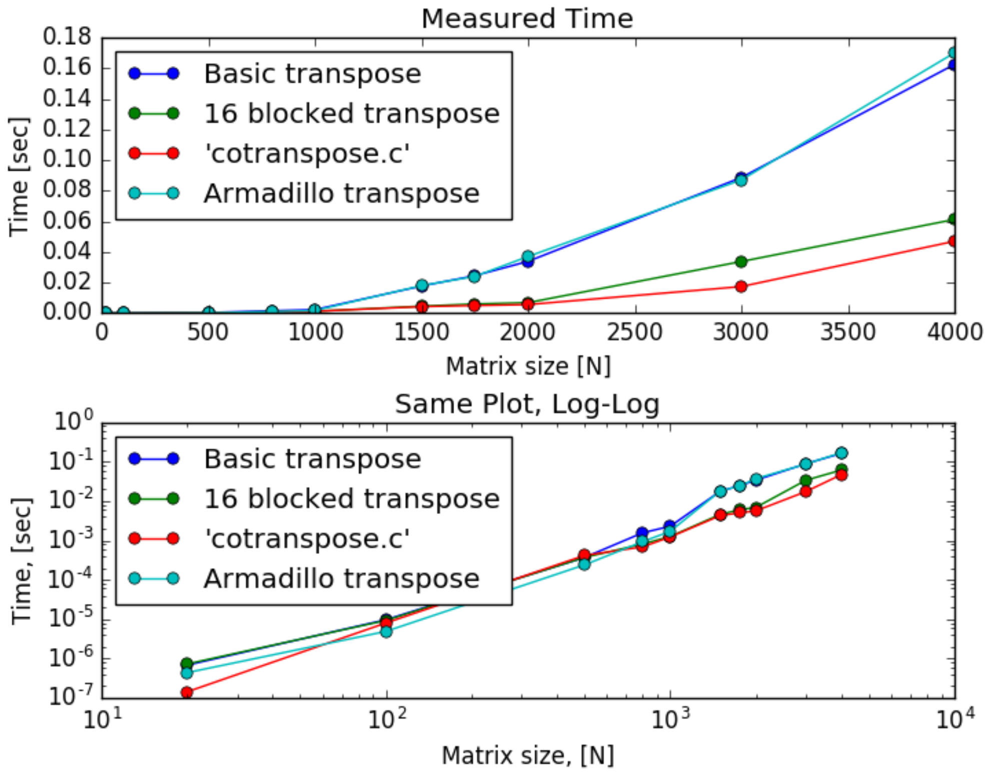

Figure 1: Differences in performance for matrix transpose algorithms

Performance expectation

From the results of Stream (from assignment 2), we could see that we got approximately a memory bandwidth of 23 GB/s under COPY operations. Let's now use this value to try to predict the performance of the basic transpose algorithm.

$$r_{\text{COPY}} \approx 23 \times 10^9 \text{ bytes/s} \tag{1}$$

$$1 \text{ transpose requires} \approx 16 \text{ bytes read/written} \tag{2}$$

$$t \approx \frac{n^2 \times 16 \text{ bytes}}{r_{\text{COPY}}} = \underline{n^2 \times 6.96 \times 10^{-10} \text{ sec}} \tag{3}$$

In figure 2 we can see that the model is good for matrix sizes up to roughly $n =$ 800-1000. After that, it predicts way to quick of a program. If we do an exponential regression, we get a better fit for the exponent of n

$$t_{v2} \approx \underline{n^{2.325} \times 6.96 \times 10^{-10} \text{ sec}}. \tag{4}$$

Using this, we get the "modified" model that can be seen in the same figure.

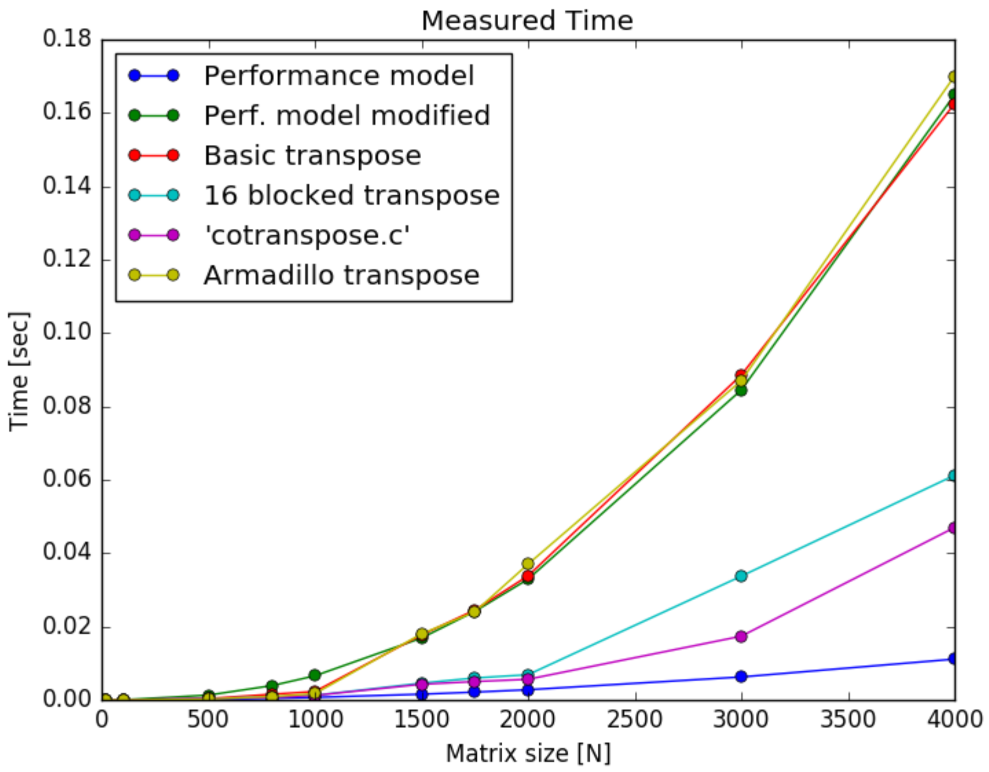

Figure 2: Model performance included in the plot of time measurment of matrix transpose algorithms

Code

```
#include <iostream>
#include <armadillo>
#include <time.h>
#include <algorithm> # min

using namespace std;
using namespace arma;

int main(){
    /* This code compares different approaches to transposing a matrix.
     * Uncomment ONE method at the time */

    // Matrix sizes, n x n and max number of tests
    vec nvec = {20,   100,  500,  800, 1000, 1500, 1750, 2000, 3000, 4000};
    vec test = {5000,2000, 1000,  500, 300,  200,  100,  75,   50,   25};

    clock_t start, finish;

    int i,j,k,l,n,T;
    double comp_time = 1e100;
    double new_time;
    vec sum_time = zeros(nvec.n_elem);

    for (l=0; l<nvec.n_elem; l++){
        n = nvec(l);
        T = test(l);
        mat X(n,n,fill::randu); // Will generate matrix with random elements
        mat Y(n,n);             // Will hold the transpose

        for (k=0; k<T; k++){    // Iterations for each n
            start = clock();
            // --- Start of methods ---
            ...
            // --- End of methods ---
            finish   = clock();
            new_time = (finish-start)/(double) CLOCKS_PER_SEC;
            if (new_time < comp_time){
                comp_time = new_time;
            }
            if (k > 0){ // Skip "cold start"
                sum_time(l) += new_time;
            }
        }
        cout << "n: " << n << ", tests : " << T + Y(n-1,n-1)*0.0 << endl;
        cout << "t min: " << comp_time << ", t avg: " << sum_time(l)/(k-1) << endl;
        cout << "------------" << endl;
        comp_time = 1e100; // Prepare for next iteration
    }
    return 0;
}
```

...and here are the methods:

```
// Standard transpose
for (i=0; i<n; i++){
    for (j=0; j<n; j++){
        Y(j,i) = X(i,j);
    }
}

// Block transpose
int i1,j1;
int bsize = 16;
for (i=0; i<n; i+=bsize){
    for (j=0; j<n; j+=bsize){
        for (i1=i; i1<min(i+bsize-1,n); i1++){
            for (j1=j; j1<min(j+bsize-1,n); j1++){
                Y(i1,j1) = X(j1,i1);
            }
        }
    }
}

//Inbuilt transpose function (armadillo)
Y = X.t();
```