PHY905, section 4
Homework Assignment 2

Håkon V. Treider
MSU id: treiderh

February 5, 2016

Introduction

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

Before turning to the standardised Stream benchmark, I will write my own code to try to estimate the memory bandwidth. For this I will use a very simple copy-routine of an n-length one dimensional array. I will time this and then count the bytes that are read and written and from this calculate the rate.

Objectives

1. Gain and understanding of sustained memory bandwidth
2. Become familiar with the effect of data size on sustained memory bandwidth
3. Become familiar with the STREAM benchmark
4. Use STREAM to predict performance for Sparse Matrix-Vector product

Results

The results can be seen in table 1 and visualized in a semilog plot in figure 1. From this we can see a peak in the rates at around $n = 10^4$. For both larger and smaller values of n, we have declining rates.

Table 1: Summary of time spent copying n array elements, with an estimate on memory bandwidth (rate)

n	Time for loop (s)	Rate (MB/s)
250	2.02E-07	19.8E03
1000	6.70E-07	23.9E03
5000	3.07E-06	26.1E03
1E4	5.77E-06	27.7E03
5E4	3.03E-05	26.3E03
1E5	6.07E-05	26.4E03
5E5	3.53E-04	22.6E03
1E6	8.15E-04	19.6E03
5E6	4.54E-03	17.6E03

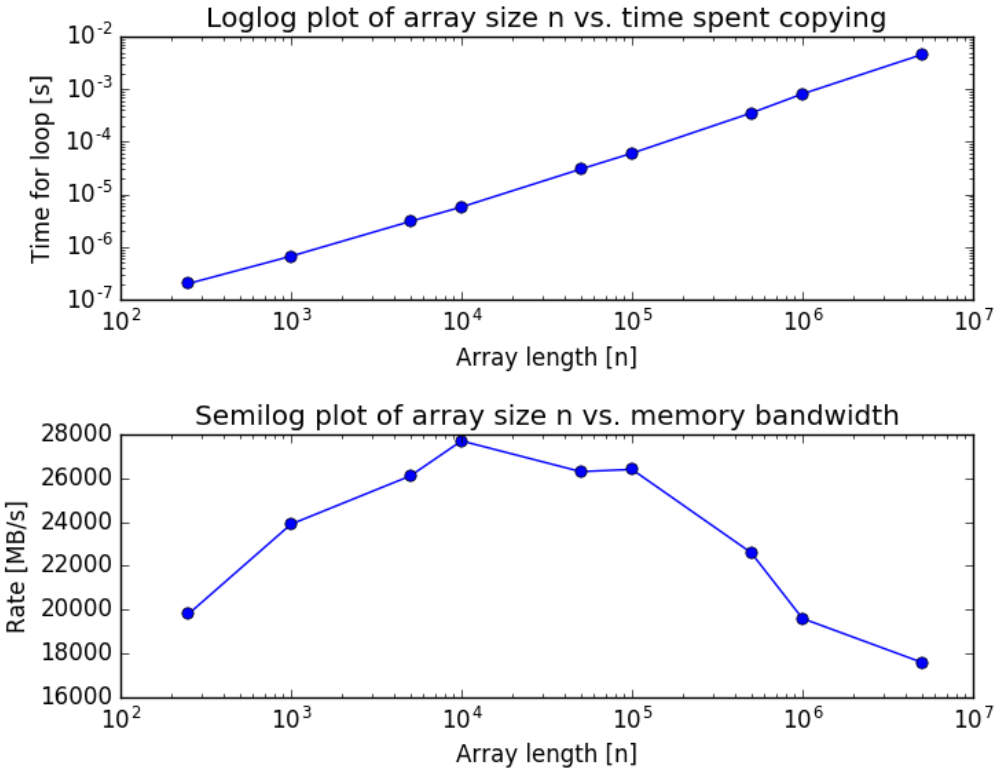

Figure 1: Loglog plot of time spent copying n array elements as a function of n and semilog plot of how the estimate on memory bandwidth (rate) scales with n

Stream Benchmark

After downloading the source code for the stream benchmark and changing the array size to 3 million elements, which is roughly four times the size of the L3 cache (which in my case is 6 MB), I got the results which can be seen in table 2. Stream uses single precision, i.e. 4 bytes per element.

Performance expectation

From the results of Stream, we can see that we get 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 sparse matrix-vector multiply operation. By the assignment text we assume that read and write require the same bandwidth. The memory operations that are run n times are negligible compared to the ones run nnz times. This is the load of a double (8 byte) and an integer (4 byte). This means that per multiply-add (MA) operation we need to load 12 bytes.

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

$$1\text{MA} \ \text{ requires} \approx 12 \ \text{ bytes} \ \text{ loaded} \tag{2}$$

$$t \approx \frac{nnz \times 12 \ \text{ bytes}}{r_{\text{COPY}}} = \underline{nnz \times 5.217 \times 10^{-10} \ \text{ sec}} \tag{3}$$

Let's now use the formula and see if we are able to guess with a reasonable precision. For $n = 1000$, we have $nnz \approx 5 \times 10^6$. The formula then gives an estimate of 0.0026 sec, while the test ran in 0.0087 sec. For $n = 3000$, we have $nnz \approx 45 \times 10^6$. The formula then gives an estimate of 0.024 sec, while the test ran in 0.095 sec. As we can see, the accuracy is within the same order of magnitude at least.

$$\text{FLOPS} \approx \frac{2}{5.217 \times 10^{-10}} \tag{4}$$

$$\text{GigaFLOPS} \approx 0.96 \tag{5}$$

I suspect I have done something wrong in the FLOPS-calculation, since the answer seems pretty low. However, what I can say is that the computation seems to be limited by the sustained memory bandwidth and not clock speed of the processor.

Conclusion

When comparing the results of Stream to my own program, the memory bandwidth rates are in good agreement. They both report rates in the 17-23 GB/s range, with some differences for the different array sizes and choice of operations. For the Stream benchmark, the absolute difference between best and worst runs are quite small (10^{-4} sec).

Table 2: Stream results for array size: 3×10^6

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	23380.2	0.002390	0.002053	0.003037
Scale:	18743.7	0.002849	0.002561	0.003625
Add:	19731.5	0.004104	0.003649	0.004778
Triad:	20247.4	0.004220	0.003556	0.006908

Code

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

double mysecond(void);

int main()
{
    double t0,t1,T,r,t_run;

    vec n_vals = {250, 1000, 5e3, 1e4, 5e4, 1e5, 5e5, 1e6, 5e6};
    vec nmbr_of_tests = {5221468, 1593036, 339095, 170965, 32543, \\
                         16196, 2328, 1212, 208};

    int i,j,k,n,tests;
    // Loop through all elements of a and copy to c
    for (j=0; j<n_vals.n_elem; j++){
        T     = 0; // Total time (aiming for 1 sec)
        n     = n_vals(j);
        tests = nmbr_of_tests(j);

        // Initialize vectors of length n:
        // a is filled with random numbers and c is zeroes
        vec c = zeros(n);
        vec a = randu(n);

        for (k=0; k<tests; k++){
            t0 = mysecond();
            for (i=0; i<n; i++){
                c(i) = a(i);
            }
            t1 = mysecond();
            T += t1-t0;
        }

        t_run = T/tests;
        r     = n*8*2 / (1e6*t_run); // MB/s
        cout << "n, tests:      " << n << ", " << tests << endl;
        cout << "Tot. t, t/run: " << T << ", " << t_run << endl;
        cout << "Rate MB/s:     " << r << endl;
        cout << " " << endl;
    }
    return 0;
}
```