CS 596 Weekly Homework Assignment 1

Christopher Ostrouchov

1 Introduction

Homework assignment 1 required that we implement the 2-norm, matrix-vector multiplication, and matrix-matrix multiplication.

$$\|x\|_2 = \sqrt{x^T x}$$

$$y = y + Ax$$

$$C = C + A * B$$

The predicted number for each of these operations can be easily calculated. The 2-norm operation requires N add, N multiply, and 1 $\sqrt{}$ instructions totaling $2N + 1$. The matrix vector multiplication requires N^2 add and N^2 multiply instructions totaling $2N^2$. The matrix matrix multiplication requires N^3 add and N^3 multiply instructions totaling $2N^3$.

Simple naive implementations of each of these are very short codes not worth giving snippets of in this document. However, these short implementations still give us interesting insight on the inner workings of the cpu, levels of cache, and RAM (hopefully the implementation never has to read from disk). All the test were preformed on an Intel i5 3570K. Important specs regarding this cpu are detailed in Table 1. Important to note is the cache in which the $4\times$ indicates that each cache contains one L1 and L2 data cache. The L3 cache is shared between all processors. Since our code is serial we will only be able to use 1/4 of the available L1 and L2 cache.

I used the well known BLAS (Basic Linear Algebra Subprograms) readily available for Linux along with a c interface for function calls. This library was used in my testing to verify the correctness of each implementation. When the command **make test** is run (test takes around 20 minutes) BLAS is used to verify each iteration 1 - 10000 for the norm, matrix-vector, and matrix-matrix multiplication. If **any** of these iteration does not meet the tolerance of machine epsilon 1×10^{-16} the test fails (mine always passes). Now that I have verified that I am obtaining the correct solution, it is interesting to look at the performance of my (simple) algorithms.

PAPI was used for accurate monitoring of the performance and processor events taking place during each of the algorithms. To simplify usage and verify correctness of my statistics I have used the high level API interface. This enable me to collect information on the real time, processor time, floating point instructions, and mega flops per second in a given time period using the hardware counters. After each iteration the PAPI statistic for each run are printed to the directory **data/** in the appropriate file: norm.txt, matvec.txt, and matmat.txt. An excerpt of the data is shown in Table 3 & 2.

Table 1: Relevant processor specifications for the Intel i5 3570k processor

Processor	i5 3570k
Clock Rate	3.4 GHz
Cores	4
Hyper Threading	no
L1 Data Cache	4 x 32 KB
L1 Latency	4 cycles
L2 Data Cache	4 x 256 KB
L2 Latency	11 cycles
L3 Data Cache	6 MB shared
L3 Latency	28 cycles
Flops / Cycle	8 double precision

2 Results/Discussion

Table 2: Results from matrix vector multiplication for arrays of square dimension and vector of dimension 991-999

Size	Real Time	Processor Time	FLIPS	MFLOPS
...	...	...	...	...
991	0.000797	0.000796	2022287	2540.561523
992	0.000792	0.000790	2016549	2552.593750
993	0.000795	0.000794	2049554	2581.302246
994	0.000798	0.000797	2084023	2614.834473
995	0.000799	0.000798	2057330	2578.107666
996	0.000807	0.000805	2027499	2518.632324
997	0.000803	0.000803	2066315	2573.244141
998	0.000804	0.000802	2039811	2543.405273
999	0.000807	0.000807	2073924	2569.918213
...	...	...	...	...

Each of the algorithms implemented require a predetermined number of floating point instructions to solve. Referring to Tables 2 let us look at the multiplication of a $999 by 999$ matrix times a $999 by 1$ vector. As stated in the introduction this would require $2N^2 flips = 2*999^2 = 1996002$. It is amazing how close the table follows this trend! There will always be deviations from the theoretical flips / problem because of the other programs running on the computer during the timing (thus perfect agreement is difficult) . However, perfect agreement was seen for very small problem sizes than ran quickly. Check the **data/** directory for detailed statistics other than Table 2 and 3.

Memory access times (latency) have in the past 10 years has become a problem. Table 1 details the latency (cpu cycles) associated with LOAD/STORE from L1, L2, and L3 cache. The matrices that we deal with are much larger than the 6 MB available in L3 shared cache (other programs are using the L3 cache as well). We can most likely assume that the kernel has given our process sole usage of one core. Thus the L1 and L2 cache

Table 3: Results from matrix matrix multiplication for arrays of square dimension 336 - 343

Size	Real Time	Processor Time	FLIPS	MFLOPS
...	...	...	...	...
336	0.037303	0.037260	130749045	3509.099609
337	0.036557	0.036487	117064843	3208.398682
338	0.034178	0.034139	115351612	3378.880859
339	0.037116	0.037072	118944266	3208.466309
340	0.034005	0.033966	117571565	3461.448730
341	0.038046	0.038000	120450324	3169.745361
342	0.036507	0.036465	123026693	3373.829590
343	0.038565	0.038467	123572697	3212.433838
...	...	...	...	...

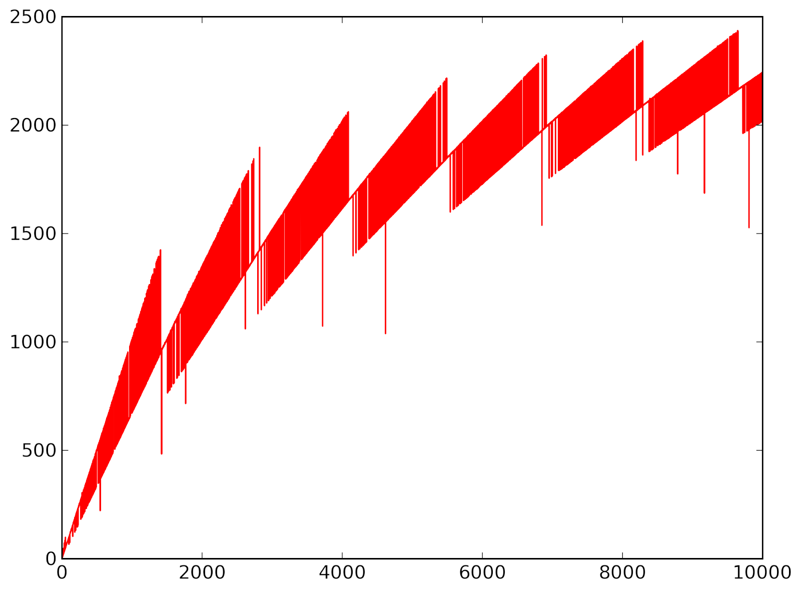

Figure 1: Plot of two norm algorithm run on problem sizes 1 - 10000

for that processor are only used by our program. Not mentioned in Table 1 is the cost of accessing memory from RAM, approximately 60 ns. In 60 ns our Intel i5 3570k processor can calculate $60 * 3.40 * 8 = 1632$ flops! (CHECK). We can only hope that we have given the processor enough work in this amount of time otherwise the cpu will be starved of work and will have to wait for the recourse to arrive (known as CPU starvation).

After a short discussion on memory latency, we can begin to understand the results seen in Figures 1, 2, and 3. For benchmarking purposes all iterations of each algorithm were written so as to ensure that the data was not in cache at the beginning of each call. Randomized locations in a huge array were chosen as inputs to the algorithm (credit to Dr. Jack Dongera for the idea).

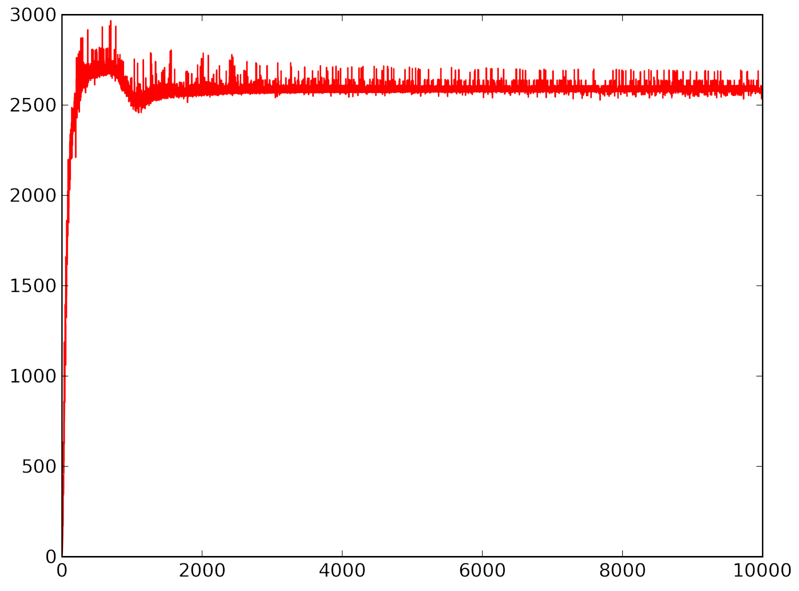

Figure 2: Plot of matrix vector multiplication algorithm run on problem sizes 1 - 10000

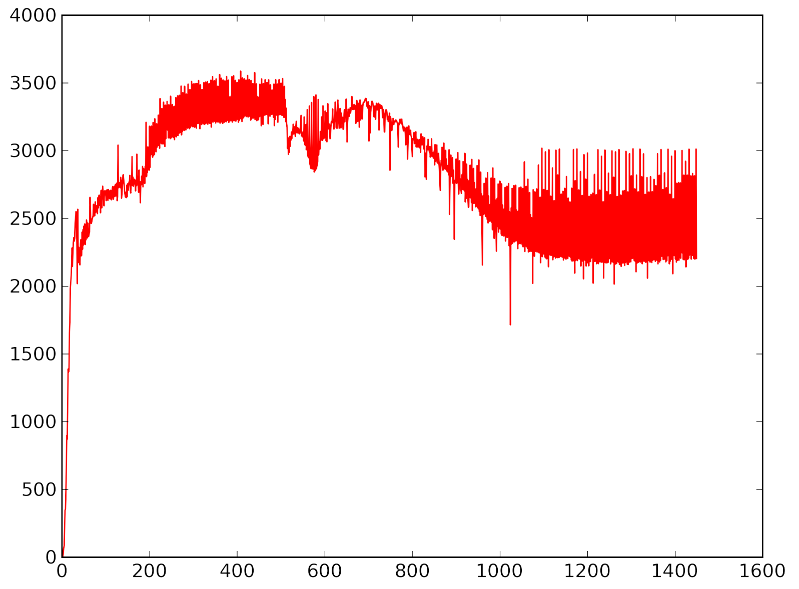

Figure 3: Plot of matrix matrix multiplication algorithm run on problem sizes 1 - 1400

The results of the 2-norm calculation of a vector can be seen in Figure 1. We can see that it takes quite awhile for the algorithm to reach its peak performance of around 2.5 GFlops. This is due to the time that it takes to fill the cache and fully use it. This phenomena is seen in the matrix vector and matrix matrix multiplication as well. One unique feature of Figure 1 not seen in the other graphs is the banded structure. I believe this feature is relation to the cost of a cache miss reading from RAM. Future work would use PAPI to diagnose if this is the issue. In Figures 2 and 3 we can see that the optimal performance also drops off after a large enough problem size from the optimal 3.4 Gflops. In Figure 2 this occurs around a problem size of 800 (correlating with the size of cache).

$$(800 doubles * 800 doubles + 800 doubles) * 8 byes = 6487200 bytes \approx 6MB$$

In Figure 3 we can see a distinct drop off around 600.

$$2 * (600 double * 600 doubles) * 8 bytes = 5760000 bytes \approx 6MB$$

These are strong indications that the performance drops off as the problem size exceeds the size of the largest cache L3.