CS 596 Weekly Homework Assignment 2

Christopher Ostrouchov

1 Introduction

Homework assignment 2 required that we implement matrix-matrix multiplication and triangular solve with rectangular matrix.

$$C = C + A * B$$

$$AX = B \text{where A is upper or lower triangular}$$

The predicted number for each of these operations can be easily calculated. The matrix matrix multiplication requires N^3 add and N^3 multiply instructions totaling $2N^3$. For the triangular solve with rectangular matrix the expected number of opperations is $(MN - 1)(N - 1)$.

Simple naive implementations of each of these are very short codes not worth giving snippets of in this document that can be seen in the source. 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× 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.

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

I used the well known BLAS (Basic Linear Algebra Subprograms) and LAPACK routines readily available for Linux along with a c interface for function calls. These

libraries were used in my testing to verify the correctness of each implementation. When the command **make test** is run (test takes around 20 minutes) BLAS and LAPACK is used to verify correctness at each iteration 1 - 1000 for the matrix-matrix multiplication and triangular solve. 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.

Compared with our previous asignment that was simply a serial implementation we used OpenMP an open source method for parallelizing code on shared memory architectures. Refering to the code written in source we can see that some interesting modifications we needed. Some parallel pramgas include single, critical, and for. Each of these puts restrictions on the order in which the threads execute along with the timing.

PAPI was used for accurate monitoring of the performance and processor events taking place during each of the algorithms. You will notice that there is alot of noise in the data ... sadly I need to look into this further. I started to use the papi low lever interface (which added some complexity to the code) and caused some large fluctuations in my timing. This enabled me to collect information on the real time, processor time, floating point instructions, and mega flops per second and Cache misses 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: dgemm.txt and dtrsm.txt. An excerpt of the data is shown in Table 2 & 3.

2 Results/Discussion

Table 2: Results from matrix matrix multiplication for arrays of square dimension and vector of dimension 652-654. tid indicates the statistics from each individual thread

tid	size	real time	float ops	d1 cache miss	MFLOPS
...	...	...	...	...	...
3	652	302791.000000	954627942	89131698	3152.761945
0	652	302791.000000	955223830	89141649	3154.729929
2	652	302822.000000	953370610	89121840	3148.287146
1	652	302821.000000	955056100	89132632	3153.863504
0	653	333491.000000	1071933385	89917817	3214.279801
1	653	333529.000000	1066700115	89335996	3198.222988
3	653	333529.000000	1068309705	89370086	3203.048925
2	653	333529.000000	1062293914	89363949	3185.012140
2	654	354352.000000	1155078417	89609704	3259.692106
1	654	354376.000000	1167281609	90169114	3293.907062
3	654	354352.000000	1165835456	89630316	3290.049036
...	...	...	...	...	...

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

Table 3: Results from triangular matrix rectangular solve for arrays of square dimension 895 - 898. Again the tid indicates the statistics from each individual thread

tid	size	real time	float ops	d1 cache miss	MFLOPS
...	...	...	...	...	...
2	895	356154.000000	1132730255	92534900	3180.450746
3	895	356154.000000	1128759334	92171644	3169.301297
0	895	356154.000000	1136631391	92604425	3191.404255
1	896	340644.000000	1056206634	102934834	3100.617166
2	896	340634.000000	1052316826	102926410	3089.288873
0	896	340634.000000	1059565554	102938183	3110.568980
3	896	340634.000000	1054211205	102941986	3094.850206
0	897	368485.000000	1131947417	99172737	3071.895510
2	897	368523.000000	1121103031	98756692	3042.152134
1	897	368564.000000	1130838987	98777099	3068.229634
3	897	368521.000000	1121524823	98794189	3043.313198
3	898	433638.000000	1409693002	99679507	3250.852098
...	...	...	...	...	...

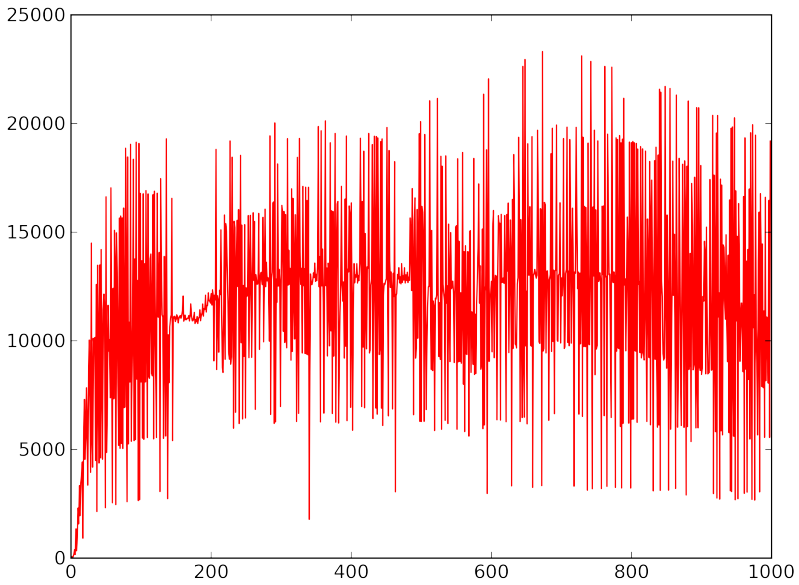

Figure 1: Plot of matrix matrix multiplication openmp algorithm run on problem sizes 1 - 1000

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 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

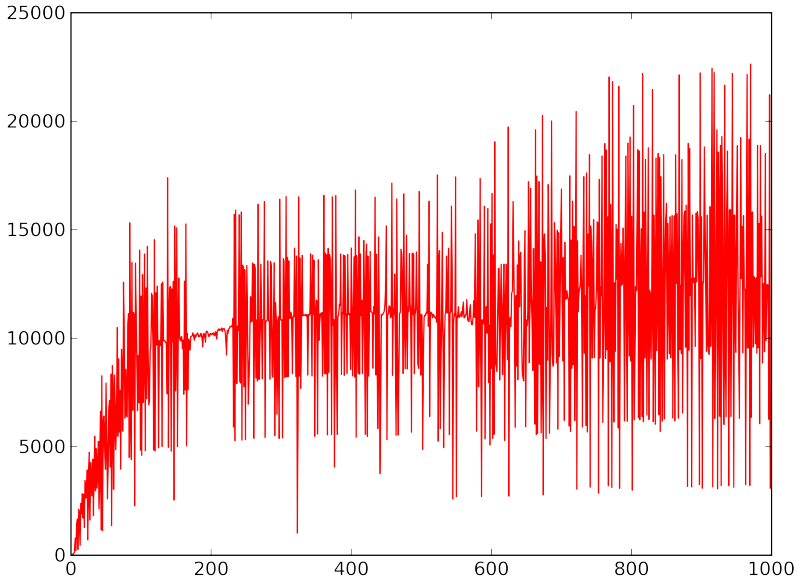

Figure 2: Plot of triangular solve openmp algorithm run on problem sizes 1 - 1000

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 and 2. 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). This memory latency addressed how it takes time for the chache memory to fill with the data it must opperate on. If the tests were run to larger sizes we would see a size in which the problem size exceeds the size of L3 cache.

The serail algorithms of each algorithm were not included as graphs in the report however was easily tested by setting the enviroment variable of the number of threads to 1. It was seen that the threads executed approximately 4 times faster. Since my plots are very noisy it was hard to see detailed differences. This noise is due to my unfamiliarity with using the PAPI library. Each iteration was tested for correctness with its equivalent BLAS routine. Compared with the serial implementation it took much larger problem sized to achieve maximum throughput.