Question 5: Matrix Multiplication

Sean Marshallsay: sm1183

The code was compiled with

```
$ clang --version
Apple LLVM version 7.0.0 (clang-700.1.76)
Target: x86_64-apple-darwin15.0.0
Thread model: posix
$ clang --std=c11 -lblas -O3  matmul.c matrix.c   -o matmul
```

and linked to `OpenBLAS 0.2.15`. The code was then run on a 2 GHz quad-core Intel Core i7-2635QM CPU, which had a theoretical peak performance of 32 GFLOPS. The computer had a 32 KiB L1 data cache, a 256 KiB L2 cache and a 6 MiB L3 cache. The number of bytes N_B in the two matrices used in the matrix multiplication is given by

$$N_B = 2 \cdot 8 \cdot N^2$$

which means that matrices for which $N < 128$ will reside entirely in the L2 cache but anything larger must make use of the L3 cache.

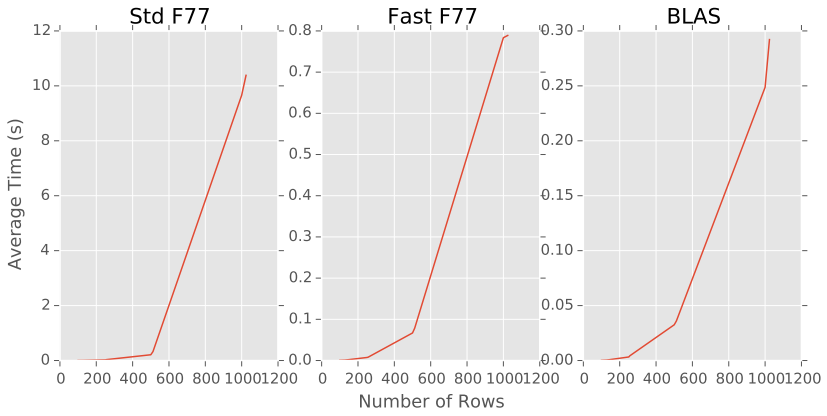

Figure 1: The timings of the three algorithms plotted on individual graphs, showing the N^3 dependence of the algorithm's time.

Each of the three algorithms was run ten times and the average time taken to perform a single multiplication (in each algorithm) was used in the resulting calculations. The raw times have been plotted in figure 1, the shape of

these graphs adds credence to the assumption that all algorithms require $2N^3$ operations where N is the number of rows in the square matrices.

The efficiency of each algorithm was calculated using

$$eff = \frac{2N^3}{t} \cdot \frac{1}{f_{peak}}$$

where t is the average time taken to perform a single matrix multiplication between two $N \times N$ matrices and f_{peak} is the theoretical peak number of floating operations per second. This is essentially the ratio of achieved FLOPS to theoretical peak FLOPS. The efficiencies are summarised in table 1 and plotted in figure 2.

Table 1: The efficiency (achieved FLOPS as a fraction of the theoretical peak FLOPS) of the three algorithms. Each algorithm was run ten times for each matrix size and an average is displayed here. N is the number of rows in the square matrix.

N	Std F77	Fast F77	BLAS
100	0.0384	0.1420	0.4370
128	0.0393	0.1321	0.3352
250	0.0425	0.1318	0.3028
256	0.0322	0.1292	0.2423
500	0.0372	0.1168	0.2402
512	0.0257	0.1068	0.2327
1000	0.0064	0.0797	0.2513
1024	0.0064	0.0850	0.2297

The two F77 algorithms show a general decrease in efficiency as N gets large but the BLAS algorithm is less certain, larger matrices would need to be tested.

The Std F77 algorithm reads one of the matrices in column-major order, since each row of the matrix is $\sim$2 KiB only 16 rows will fit in the L1 cache at once (likely fewer because part of the other matrix will also reside in the L1 cache) so the cache must be refreshed every 16 rows but only one value is read from each row. Larger amount of pointer arithmetic will also occur because the pointer is not just incremented by one each time, causing an extra performance hit. In the Fast F77 algorithm both matrices are accessed in row-major order so many more values can be read before the L1 cache needs to be refreshed. This is why the Fast F77 runs generally faster than the Std F77 algorithm.

There is a drop in efficiency between $N = 100$ and $N = 128$ in the Fast F77 and BLAS algorithms. Since the Std F77 algorithm is unaffected this drop is

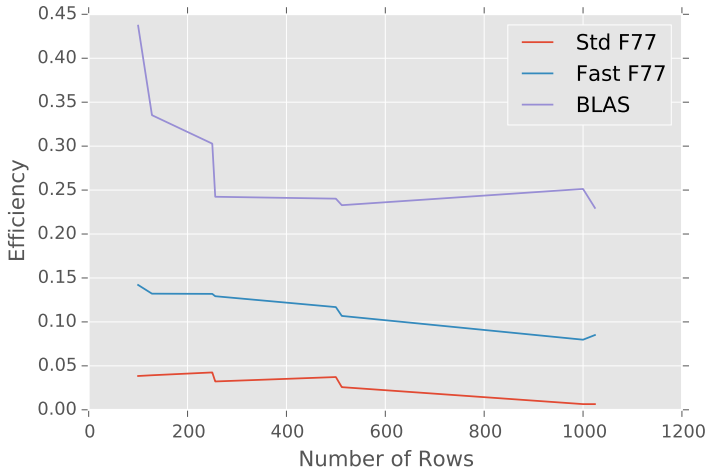

Figure 2: The timing of the three algorithms plotted together.

unlikely to be from cache-thrashing because the Std F77 should be more prone to cache-thrashing, instead it is likely that the drop is because the L3 cache must now be used which has a much read time.

The other drops in efficiency ($250 \to 256$, $500 \to 516$ and $1000 \to 1024$) are due to the associative cache. An associative cache of C lines will map line 1 and line $C+1$ to the same place in the cache. When the size of the matrices is a power of two (like the cache sizes are) it becomes more likely that one part of the two matrices which is mapped to the same line will be accessed consecutively.