Operating Systems & Networks, 2008
0440949 Andreas van Cranenburgh
Wed Sep 17 16:30:57 CEST 2008

Abstract

The goal of this experiment was to measure the relative merits of optimizing matrix multiplication. Though matrix multiplication generally uses $O(n^3)$ operations (not considering the low-level complexity of scalar multiplication), this lower bound hides constant factors which can vary according to the order in which the elements of the matrices are multiplied, as well as parellelisation and vectorisation. Given a few optimized versions this paper will measure their speed and demonstrate the extent of this variation.

1 Benchmarking matrix multiplication

1.1 Measurement tool

The tools for this experiment consist of a function which initializes a structure with the current time and cpu usage (in milliseconds). After initializing some computations can be done. Immediately afterwards a second function can report how much wall clock time, user cpu time and system cpu time it has taken for the preceding computations to finish.

This is similar to the unix command "`time`," except that this tool can be used to measure a specific function instead of the execution of a whole program.

The resolution is 1 `ms` for cpu time, because the clock resolution was 1000 Hz on the computer which performed the benchmarks (`CONFIG_HZ=1000` kernel option). The wall clock time is reported in micro seconds. In estimating the overhead of the measurement functions it was discovered that this overhead is well below this resolution of 1 `ms`. This was established by calling the measurement function 100000 times and dividing the total elapsed time by this number to obtain the time per iteration. This demonstrated that the overhead amounts to `7.959e-07` for the wall clock time, and even less for cpu time. This is well beyond the desired precision of three decimals.

The smallest measureable time difference was `9.536e-07` for the wall clock time; although higher values also occurred the aforemented value seems to be the rock bottom. This result was consistently obtained in 2 iterations. This demonstrates that wall clock time has a resolution of a few microseconds (which explains why `gettimeofday` returns the result in microsecond precision).

For the cpu time `1.000e-3` was the minimal time difference, though with a variable number of iterations (ranging from 100 to 1000). This time difference conveniently matches the 1000 Hz kernel ticks.

1.2 Results

Tests ran on a 64 bit Pentium Dual at 2Ghz, with 1024 KB cache and 1024 MB main memory, the operating system was Linux 2.6. This computer was not doing anything besides running these tests.

In the following tables the left half shows wall clock time per floating point instruction, the right half is user cpu time per floating point instruction (system cpu time has been ignored, no other tasks were running). The

number of floating point instructions has been fixed at $mnp + m(n-1)p$ (number of multiplications plus additions, where m, n and p are the sizes of the matrices). The multiplications where repeated until at least 1 second of cpu time was used, and this factor was also divided for.

Results when compiling without optimizations:

```
wall: min   mean      max       sdev    cpu: min    mean      max       sdev
9.009e-09 9.114e-09 9.278e-09 1.847e-47 9.015e-09 9.098e-09 9.177e-09 0.000e+00
9.918e-09 1.863e-08 5.457e-08 8.211e-48 9.911e-09 1.862e-08 5.455e-08 1.460e-47
5.089e-09 5.284e-09 6.146e-09 5.702e-48 5.081e-09 5.284e-09 6.177e-09 1.425e-48
5.278e-09 5.690e-09 7.350e-09 1.847e-47 5.266e-09 5.680e-09 7.247e-09 9.123e-49
```

With optimizations (`gcc -O3`):

```
wall: min   mean      max       sdev    cpu: min    mean      max       sdev
1.757e-09 1.937e-09 2.215e-09 3.207e-50 1.758e-09 1.929e-09 2.165e-09 4.312e-49
2.359e-09 8.623e-09 2.849e-08 9.123e-49 2.356e-09 8.612e-09 2.848e-08 1.460e-47
1.279e-09 1.608e-09 2.419e-09 1.425e-50 1.280e-09 1.598e-09 2.418e-09 0.000e+00
1.376e-09 1.533e-09 2.072e-09 2.281e-49 1.374e-09 1.524e-09 1.997e-09 8.909e-50
```

These results are for `MatMul_1, MatMul_3, MatMul_19` and `MatMul_21` on a line by line basis. Each function has been measured with sizes ranging from 20 to 960 elements. In order to see which optimizations results in the overall best performance the results from these different sizes have been averaged together.

The first two algorithms are straightforward matrix multiplications, the only difference being that `MatMul_1` has `n1` in its outer loop and `n3` in the inner loop, where `MatMul_3` has `n2` and `n1`, respectively.

`MatMul_19` uses a divide and conquer approach, dividing the matrices in sub-blocks for which the results can fit in cache, in this case a sub-block size of 40.

Finally there's `MatMul_21`, which (aditionally) uses copy optimization. This means that the second matrix is transposed, such that its rows can be accessed just as efficiently as the columns of the first matrix. This version also does loop unrolling, just as `MatMul_19` (both unroll to 8 instructions).

To see the effect of matrix size the tests were also ran on matrices from 320 to 960 elements. The results are as follows:

```
wall: min   mean      max       sdev    cpu: min    mean      max       sdev
9.445e-10 1.003e-09 1.027e-09 1.069e-50 9.429e-10 1.001e-09 1.026e-09 4.276e-50
5.063e-09 9.844e-09 1.430e-08 4.276e-48 5.062e-09 9.840e-09 1.430e-08 1.711e-49
7.399e-10 9.166e-10 1.208e-09 1.069e-50 7.402e-10 9.168e-10 1.208e-09 1.069e-50
7.010e-10 7.295e-10 7.836e-10 0.000e+00 7.008e-10 7.281e-10 7.819e-10 2.405e-50
```

1.3 Discussion

It is clear that letting the compiler optimize the code makes a big difference. But in this case still further speed gains can be obtained by applying algorithm specific optimizations.

The difference in performance between `MatMul_1` and `MatMul_3` probably arises because the latter does not make optimal use of the fact that the matrices are presented in a row-major order representation. This negatively affects the spatial locality of reference in cache.

The most dramatic performance difference is between the first two and the second two algorithms. The divide and conquer approach apparantly really pays off in terms of cache performance.

The performance of `MatMul_19` seems to win by a small margin when small and large matrices are used. The extra work that `MatMul_21` has to do did not result in overall better minimal times, though it did have a smaller mean (this result was repeatable). When only large matrices are multiplied `MatMul_21` does beat the other algorithms, despite the extra work of transposing the second matrix.

As a final note it can be suggested that some additional speed can be gained by rewriting all for loops to do inequality checking instead of smaller-than checking, since inequality checking can be done in only one cpu instruction. This change lead to a very slight performance gain, so it is not part of the results (see `MatMul_22` and `verify()` which demonstrates the correctness of the modifications). Perhaps rewriting for loops into do while loops without index counters (using pointer inequality checking instead) could improve the speed even more.