Optimizing matrix multiplication

Amitabha Banerjee
abanerjee@ucdavis.edu

Present compilers are incapable of fully harnessing the processor architecture complexity. There is a wide gap between the available and achieved performance of software. Thereby, the need for performance tuning. Performance tuning of the simple matrix multiplication has indeed been a very tough and challenging project. In this work, we discuss some of the optimization techniques, which gave us substantial improvements.

Test Setup:
The machine used for the test had a Pentium IV processor with a 16KB L1 cache and 512 KB L2 cache. We started from the short sweet matrix multiply code, which we call basic matrix multiply. We used the gcc version 3.2.1 compiler.

```
Basic_matrix_multiple (A,B,C,m)
  for i= 1 to m
    for j= 1 to m
      for k= 1 to m
        C(i,j) = C(i,j)+ A(i,k)*B(k,j)
```

The optimization techniques were applied in the following steps:

1) **L1 cache blocking optimizations**: Here the idea is to partition the big matrices into uniform blocks. Matrix multiplication is carried out block by block. Details of the algorithm are in [1]. Choosing the optimal block size is very important as well. The idea here is to fit the two blocks, which are being multiplied into the L1 cache so that the self-interference [2] misses are minimized. The optimal block size can be calculated as:
   $2 * (blockSize)^2 * wordSize = L1\ cache\ size.$
   In our case, the *wordSize* is 8 bytes because the matrix is of double data type. Substituting the values, we calculate the *blockSize* to be 32. To verify, we plot the performance of blocked matrix multiply on a 512 * 512 matrix while varying the block sizes from 16 to 64 in Figure 1. Note that we choose only multiples of 2 here, the reason being that the L1 cache has a line size of 4 words, and therefore non-multiples of 2, make the block matrix size non-multiples of 4, which tends to be inefficient. As expected we observe that the performance peaks at block size of 32.

2) **Compiler flag optimizations:** gcc provides various compiling options to improve performance. A detailed list of all the compiler options can be found using the *man* command.
   In our case we found the 3 compiler options: -O4 –funroll-loops –ffast-math particularly useful in speeding up the performance. The –ffast-math options is not recommended always, because it makes optimizations to the IEEE floating point

standard which may yield incorrect results in some cases. However in the case of matrix multiply it worked pretty well.

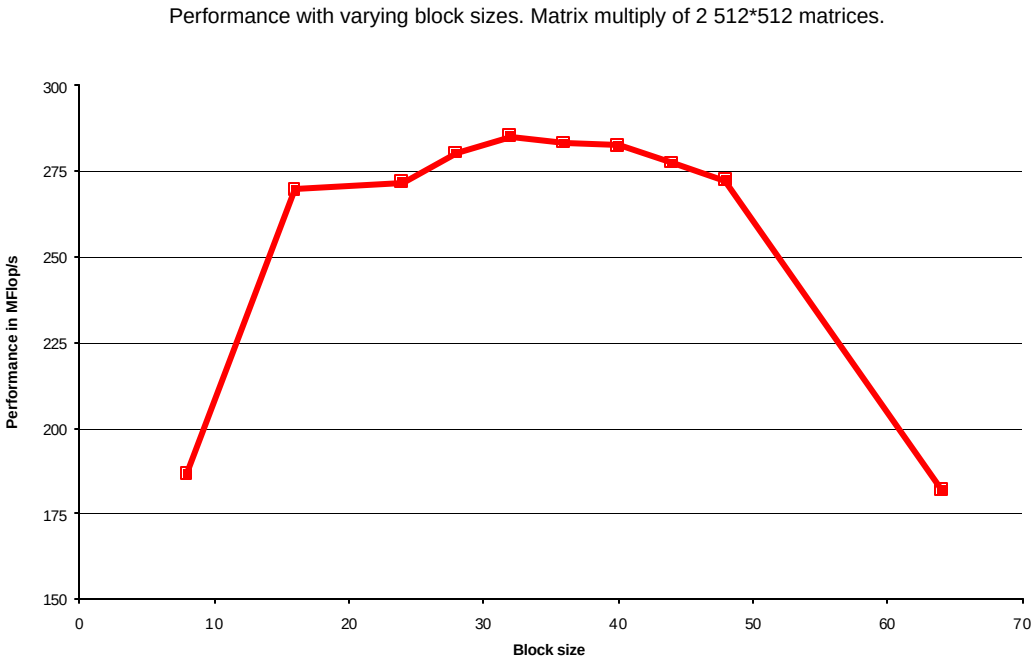


Figure 1: Performance with varying the block factor.

3) **Software optimizations**: There is a bunch of software optimizations that we considered. Most of these are based on C performance coding guidelines, which may be found in detail in [1, 3]. Over here, we shall elaborate on ideas relevant to our context.
    a) *Software unrolling and register reuse*: We unrolled the innermost loop two times. This helped us load a common value used in an outer loop into a register. It is important to spot data, which is used recurrently and store them in registers explicitly, so that they do not have to be repeatedly fetched from the L1 or L2 caches.
    b) *Avoiding pointer arithmetic in arrays*: If *c* were an array, it is much more efficient to address an element as c[10] instead of *(c + 10). The former can be executed in one LOAD instructions, because LOAD instructions in most processors allow address computations. The latter takes two instructions.
    c) *Avoiding inequality comparisons*: The BRNZ (Branch if Nonzero) instruction is extremely efficient, so it helps make use of it as much as possible. Therefore its is helpful to convert the inequalities to equality operations. For example, a inequality based for-do loop may be replaced by an equality based do-while loop.
    d) *Interspacing add-multiply operations*: Typically add and multiply operations are performed in different processing units. Placing a bunch of multiply operations together reduces efficiency because the instructions

might be stalled because of unavailability of the multiplication unit, while the addition unit is free. In this context, it is useful to remember that multiplication operations typically have much higher latencies than arithmetic operations. Therefore it is efficient to have add and multiply operations interleaved.

4) **Copy optimization**: We observed that there were substantial dips in performance on matrices of sizes of powers of 2. e.g. On 128 * 128, 256 * 256 and 512 * 512 matrices. This can be explained as follows. In matrix multiplication a column of data multiplies a row of data. Assume that the matrix is stored in row-major format. When the matrix is of size 2^N, the addresses of the elements in a column are also separated by powers of 2. Since the L1 uses a direct mapped cache, all these elements in a column map to the same cache line. This leads to an extraordinary number of cache misses.
   The remedy is a technique known as copy optimization. Before starting matrix multiplication, the matrices are copied into memory locations, the first matrix being stored in row-major format block-wise, while the second matrix is stored in column major format block-wise. This also helps in achieving spatial locality. Copy optimization thus helps reduce the dips in performance at powers of 2, while improving performance also because of spatial locality. Because copy optimization involves the additional overhead of copying data, it is not recommended for small sized matrices for which the impact is much less but overhead is more.

5) **L2 cache blocking optimizations** : The final step optimization involves an additional level of blocking for the L2 cache. The L2 cache size is 512 KB. So the maximum matrix size that it can hold may be computed by the formula:

$$2 * (blockSize)^2 * wordSize = L2\ cache\ size$$

   which gives the optimal block size to be 181. We found the optimal achieved at the block size of 160.

Figure 2 shows the results of our optimizations. Note that the performance of basic matrix-matrix multiply degrades heavily with huge dips at around powers of 2. The factor improvements with each degree of optimization on a 479 * 479 size matrix are indicated in Table 1. We note that substantial improvements are achieved at each step. We also note that the dips in performance at powers of 2 are much less after the copy optimizations.

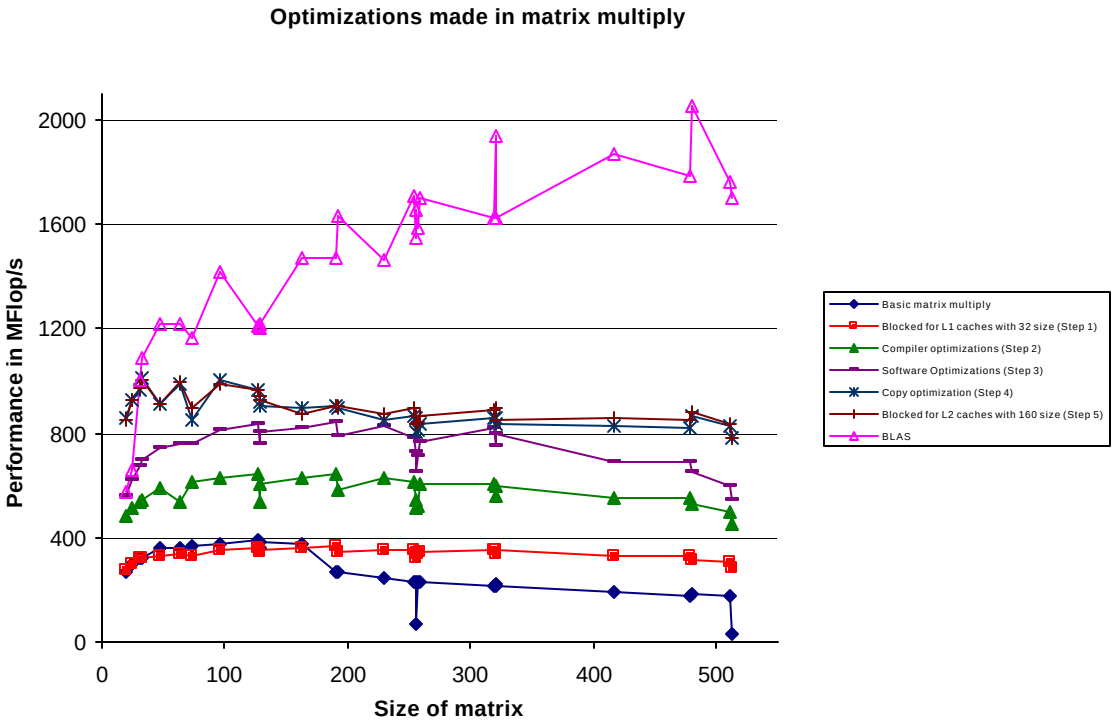


Figure 2: Performance tuning for matrix multiply.

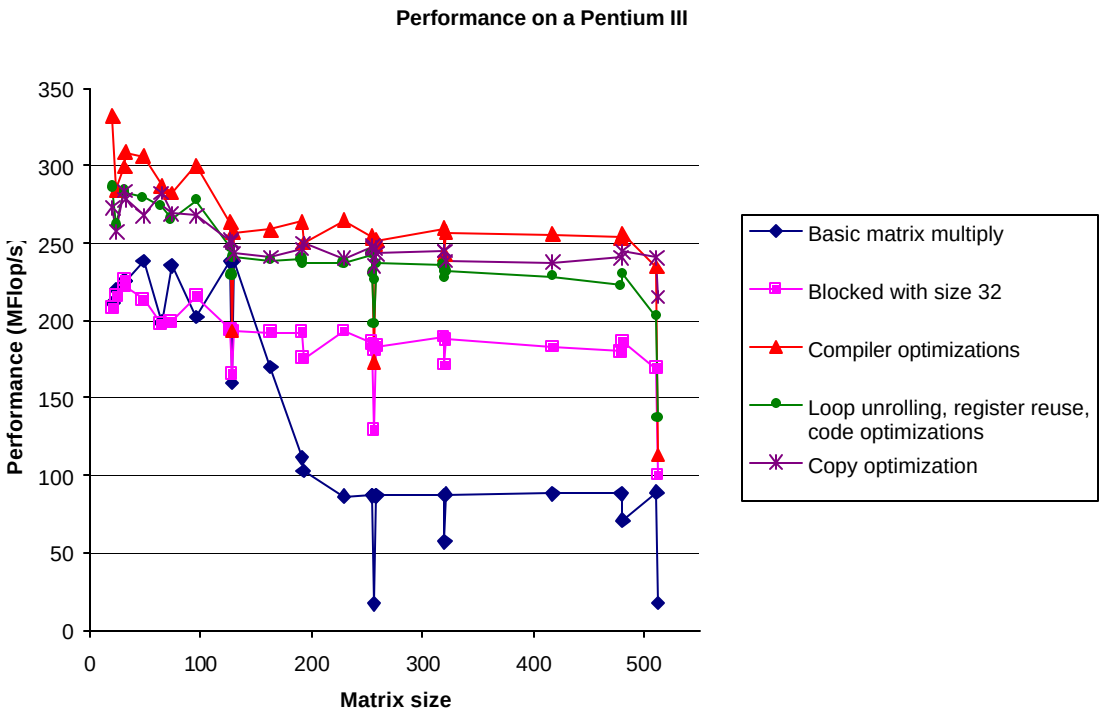
Optimization	Performance improvement
Basic matrix multiply	1.0
L1 cache blocking optimizations (1)	1.8
Compiler flag optimizations (2)	3.1
Software optimizations (3)	3.8
Copy optimization (4)	4.6
L2 cache blocking optimizations (5)	4.7
BLAS from ATLAS	9.9

Table 1: Performance improvement on a 479 * 479 matrix

After all these optimizations we compare the performance with that achieved using the BLAS library supplied by ATLAS which can be downloaded at [4]. As can be seen in Table 1, BLAS beats our best performance by greater than a factor of 2. BLAS has advantages of taking into account the complete architecture of the machine, such as the number of floating point and integer results, the number of buses available etc.

All these optimizations are tailored to our machine. We now run the optimizations on a different machine. This time we choose a Pentium III processor. The results are plotted in Figure 3. We observe that the improvements are not as much as we obtained on a Pentium IV machine. Some of our optimizations like L1 block sizes and L2 block sizes

were considered particular to the Pentium IV processor and they are not tuned to the Pentium III processor. Moreover the disparities between the achieved performances for matrix multiply and available performance for a Pentium IV is huge, and thus our optimizations kick in much better. This shows that with faster processors, the need for performance tuning is very important to harness the full capacity of the processor.



Some more optimizations, which have been documented in literature, which may lead to marginal improvements, are as follows:

1) Fringe handling: Since the matrix size is not always a multiple of the block size, some blocks are the left over fringes after all the blocks of block size have been carved out. Some optimizations recommended for handling fringes are to have fine-tuned code for 1* 2, 1 * 3, 2 * 3, 2 * 4, 3 * 4 etc. fringes, and call these when required.
2) Memory pre-fetch: A lot of cache misses are intrinsic misses [2], during which the L1, L2 caches are being loaded with data from the memory. Pre-fetch can help reduce these latencies. Usually compilers do not support these techniques and hence these operations have to be done in assembly level code, which makes it difficult to implement them.

Important references:

[1] D. Parello, O. Temam and Jean-Marie Verdun, "On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance – Matrix-Multiply revisited.", Supercomputing, 2002.

[2] Monica S. Lam, Edward E. Rothberg and Michael E. Wolf, “The Cache Performance and Optimizations of Blocked Algorithms”, ASPLOS IV, 1991.
[3] PHiPAC technical report available at:
http://www.icsi.berkeley.edu/~bilmes/phipac/tr-98-035.pdf
[4] ATLAS BLAS available at www.netlib.org/altas .