Assignment 1

Yunqi Zhang
Ivan Motyashov

1. Introduction

Our goal is to implement the single core, blocked matrix multiplication algorithm, along with a number of optimizations in an effort to achieve the best GFLOPs rate, and carry out a set of experiments to assess the performance benefits.

2. Optimizations

Starting with the naive block matrix multiplication, we've done the optimizations listed below:

A. **Preload the matrices into memory before usage:** In the process of block multiplication, a number of elements is used more than once. For example, Aik[i][k] for k = 0..BSZ is used once for every j=0..BSZ, or exactly BSZ times. To increase spatial locality, thus improving cache performance, we read the corresponding blocks Aik, Bkj, and Cij into block-sized 2D grids prior to each iteration of block multiplication.

B. **Transpose the Bkj matrix:** Before each iteration of block multiply, we transpose the matrix Bkj, allowing us to read the matrix row by row, as opposed to column by column, making use of increased spatial locality to improve cache performance.

C. **Use variables instead of expressions as array indices:** We use variables to replace the expressions as array indices in the loops in order to reduce the number of operations. This optimization buys comparatively very little performance.

D. **Unroll loops:** By partially unrolling the inner loop, we are able to improve performance by reducing the branch penalties, and possibly allowing the compiler to pre-calculate array offsets. The best increase in performance in our environment is achieved when we unwind the loop by a factor of 4, a number determined experimentally. [This introduces the limitation that b must be >= 4]

E. **Make use of registers:** We unroll some loops and use named variables to store copies of the matrices. The most used of these intermediate variables are stored in registers to significantly improve performance. In particular, we use accumulator variables, declared with the *register* keyword to store intermediate sums, as opposed to updating the Cij block in-place.

3. Experiment Environment

We did all our experiments on the Bang node with the environment showed in Table-1.

Table-1: Experiment environment

Label	Property
CPU Model	Intel ® Xeon ® CPU E5345 @ 2.33GHz
L1 Cache	32 KB
L2 Cache	8 MB (Shared by 4 cores)
Operating System	Linux version 2.6.32-220.13.1.e16.x86_64
Compiler	GCC 4.4.6 (Red Hat 4.4.6-3)

4. Data

First, we did some experiments to figure out the performance as a function of matrix size n, as shown below.

Table-2: Performance for different matrix size n

n	128	256	384	512	640	768	896	1024
Unblocked	0.7538	0.7623	0.6136	0.7682	0.7579	0.4342	0.3780	0.0360
Hand Blocked	2.4161	2.4422	2.4600	2.4567	2.4437	2.3928	2.3558	2.3675
BLAS	3.2711	2.3359	3.1578	2.1209	3.2062	2.2342	2.9335	1.5030

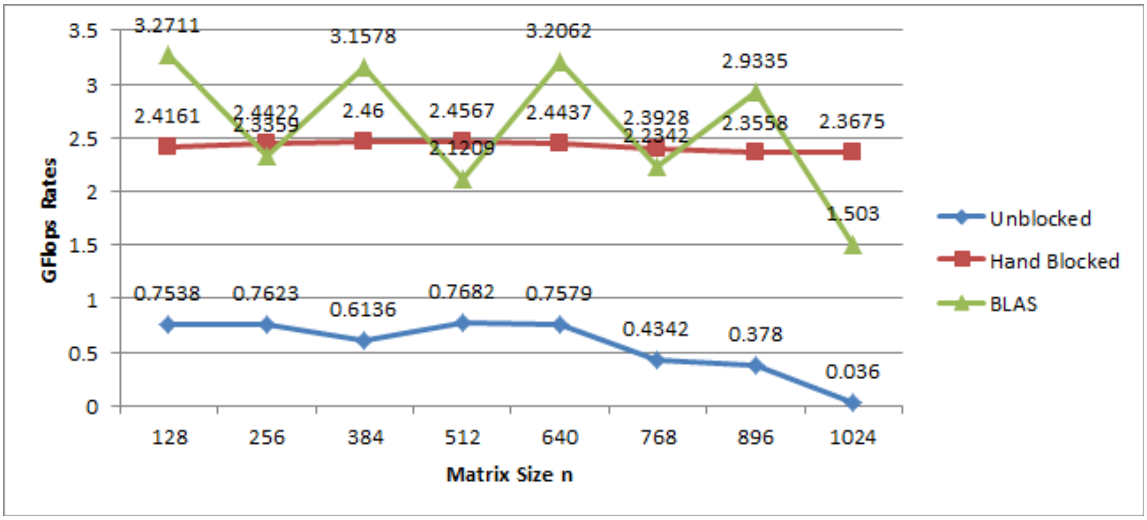

Figure-1: Performance under different matrix size n

Then we tried to measure the blocking factor on performance. We did our experiments with the matrix size n = 1024.

Table-3: Performance under different block size b

b	16	32	64	128	256	512	1024
Hand Blocked	2.3675	2.8204	2.8685	2.7496	2.7971	2.5945	0.9054
BLAS	1.5030	2.7902	5.5183	6.7205	7.2443	7.6119	7.8933

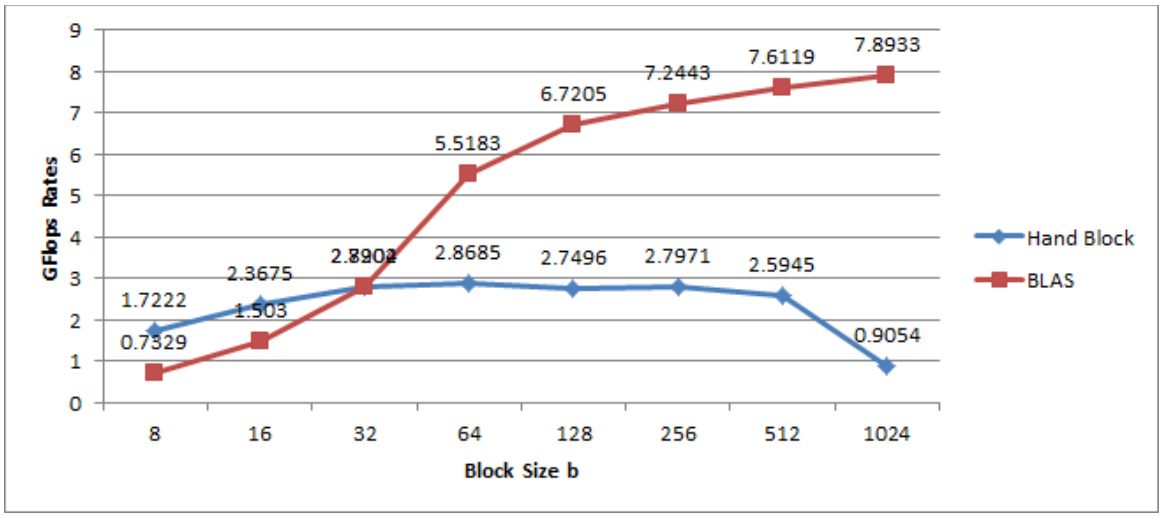

Figure-2: Performance under different block size b

5. Analysis

In theory, times for the blocked and unblocked algorithms should be comparable for smaller values of n, because, when b is close to n (the ratio b/n is close to 1), the blocked algorithm reduces to unblocked matrix multiplication (since there are fewer blocks to multiply and each multiplication of blocks uses an unblocked algorithm). Also, when n is small, the unblocked algorithm makes good use of cache (which is one of the principal factors responsible for the performance boost achieved using the blocked algorithm). The times will diverge significantly when n becomes large enough to provoke many capacity cache misses in the unblocked case.
In practice, however, we applied a number of optimizations to the block multiplication algorithm, which make our blocked multiplication noticeably faster than the provided unblocked algorithm even for small values of n.

For the unblocked algorithm, we observe an abrupt and colossal drop in performance for n=1024 (from ~0.4 to ~0.03 GFLOPs). By looking at the algorithm, we can hypothesize that the L2 cache will likely contain the entire matrix A (since each row A[i] is referenced for every j and k) and a row of C and B each. This would require a cache of size n * (n+2) * 64 bits. In particular, for n=1023 we need a cache of size 8388600 bytes ~ 7.99 megabytes. For n=1024, we need 8404992 bytes, which is slightly greater than 8MB. It seems likely, then, that the L2 cache is 8MB large.

The performance of blocked multiplication remains more or less constant as a function of block size b, except for the extremes. Performance visibly deteriorates when b = n = 1024 and when b < 32. When b = n, we are essentially performing unblocked multiplication, since we are only dealing with one block. When b is very small, it is possible that we are not making very good use of the L1 cache. Since memory access is slow, we want to load as much data into cache as possible in one operation. However, if the cache is large enough to fit more than 3 blocks (Cij, Aik, Bkj) at a time, we could be utilizing it more efficiently by increasing the block size and thus decreasing the number of member accesses. After a certain threshold (when the cache cannot hold more than 3 blocks), we expect performance to remain more or less constant. According to Table-3, this threshold is reached for n=32. Therefore, the L1 cache is big enough to fit 3 blocks, but too small to fit 6 blocks, allowing us to estimate its size as being >= 24KB and < 48KB.