Optimizations on Double-precision General Matrix Multiplication (DGEMM)

A brief report for CSE 260 / Fall 2017 - Assignment #1

Dewal Gupta
A09938362
degupta@ucsd.edu

Mithun Chakaravarrti Dharmaraj
A53235626
mdharmar@ucsd.edu

MOTIVATION

Today, processors are much more faster than memory. As we know, “A system can only perform as fast as the slowest link”. By making use of DRAMs for running programs we probably might not exploit the entire power our processors can deliver. To optimize the problem of matrix multiplication, it’s essential to understand the memory hierarchy of the system we’re working on. The system we started with is a compute node on Bang cluster which has two quad-core Intel Xeon E5345 ("Clovertown") 2.33GHz CPUs, and 16GB of RAM.

Since we know that memory operations such as read & write are slow, we ultimately want to reduce it for every computation we make. The task at hand simply came down to increasing the number of floating point operations performed per load/store. This is technically called Computational Intensity (Q).

LEVEL 2 BLOCKED MATRIX MULTIPLY (for utilizing L2 cache)

The idea behind blocking is that we split the bigger matrix into smaller blocks, small enough to get the entire block into fast memory and then compute the multiplied matrix for smaller blocks while piecing them together to get the entire solution. What we avoid here is the multiple loads and stores to slow memory, by blocking them into faster memory.

We have been provided with a naive matrix multiplication code which implements blocking followed by matrix multiplication. The block size is such that all three matrices A, B, C fit into the L1 cache. Extending to this idea, we can perform a second level of blocking, such that we can make use of the L2 cache as well to reduce the loads/stores with slower memory even further.

lscpu on the bang compute node gave the following results:

```
...
L1d cache:             32K
L1i cache:             32K
L2 cache:              4096K
...
```

Data cache for L1 is 32KB. Our matrices are double matrices and for matrices A, B, C to fit into L1 cache with with block size N then,

```
3 * N * N * sizeof(double) <= 32000
```

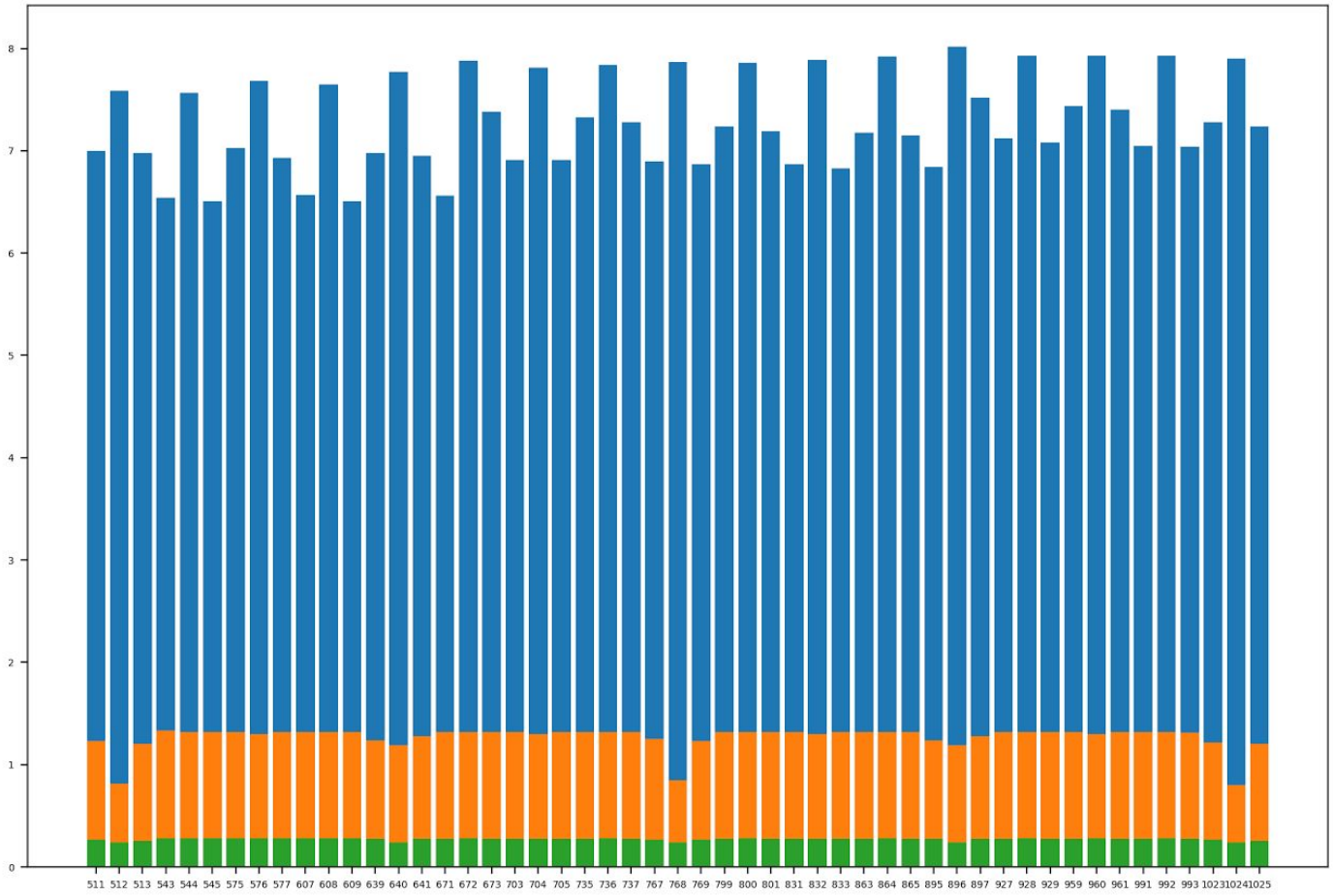
Solving this we get N <= 36. For blocking to be effective, we must choose a block size of 36 to make use of the L1 cache. Building on top of this for making use of L2 cache, a second level blocking needs to be done. Cache size of L2 is 4096KB. Then,

```
3 * N * N * sizeof(double) <= 4096000
```

Solving this we get N <= 413. For the time being, we are rounding block size to be 400 for the second level of caching.

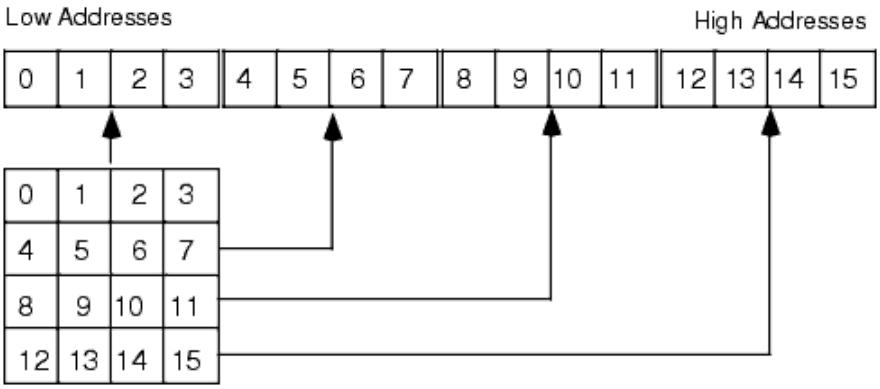
```
BLOCK_1_SIZE // blocking size for L1 cache
BLOCK_2_SIZE // blocking size for L2 cache

matrix_multiply(lda, A, B, C):
for i=0, matrix_size, BLOCK_2_SIZE:
    for j=0, matrix_size, BLOCK_2_SIZE:
        load each block of C[i,j] into L2 cache
        for k=0, matrix_size, BLOCK_2_SIZE:
            load blocks A[i,k] and B[k,j] total lda^3 times each into
L2 cache
            do_second_block()

Let C be M x N, Let A be M * K and B be K * N
do_second_block(M, N, K, A, B, C):
for i=0,M, BLOCK_1_SIZE:
    for j=0,N, BLOCK_1_SIZE:
        load each block of C[i,j] into L1 cache
        for k=0,K, BLOCK_1_SIZE:
            load blocks A[i,k] and B[k,j] total M*N*K times each into
L1 cache
            C[i,j] += A[i,k] * B[k,j]
            BLOCK_1_SIZE x BLOCK_1_SIZE matrix multiply
            Write each block C[i,j] once
```

This gives a performance of up-to **1.26 GFlop/s** as opposed to the naive matrix multiplication which has a geometric mean of 0.27 GFlop/s. But the BLAS benchmark has a performance of 7.27 GFlop/s. (Also uncommenting OPTIMIZATION on the make file gave a nominal boost). The plot is as follows:



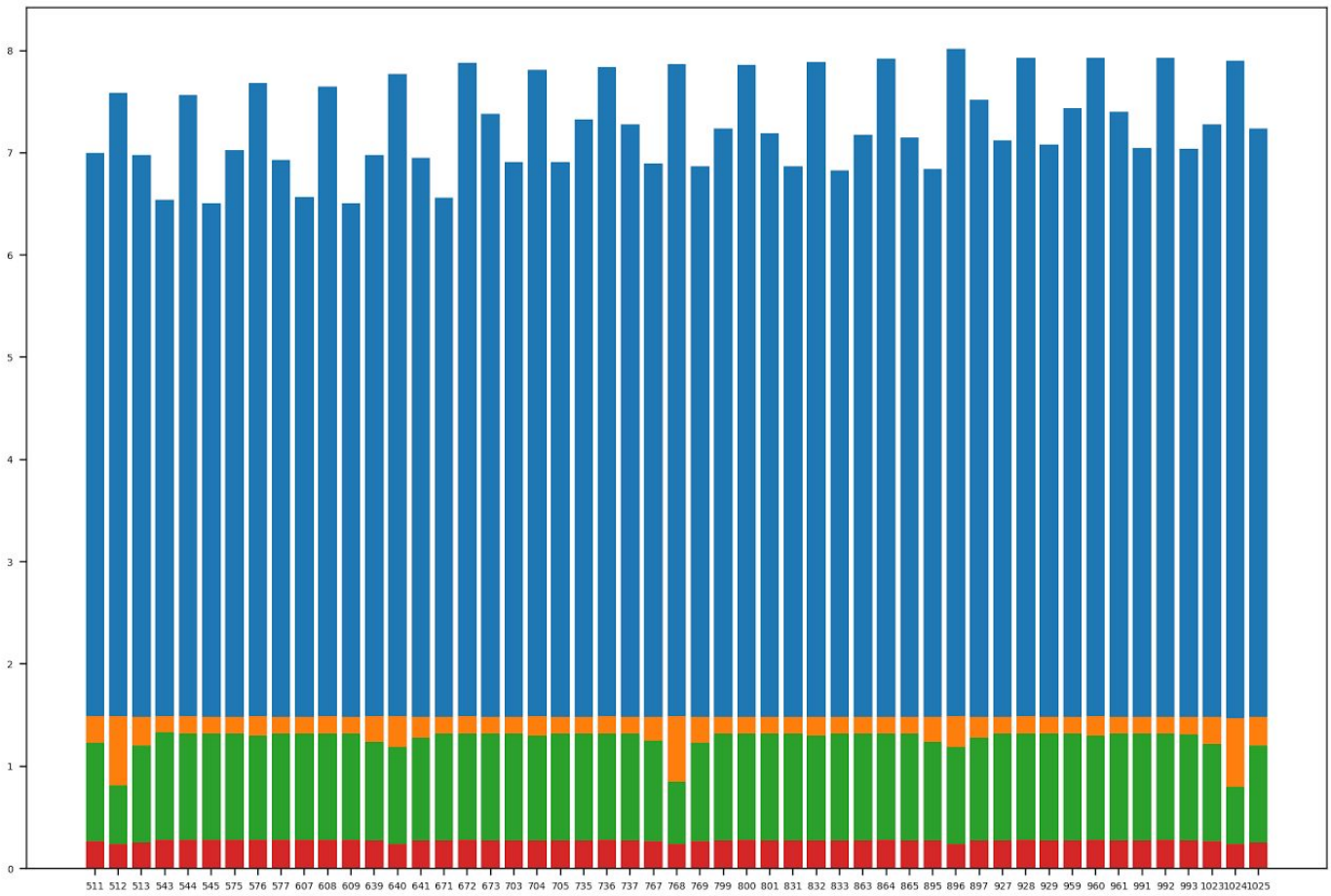
TRANSPOSING MATRIX B

We know that C uses row major ordering. But the current level two blocking multiplies each row of A in the A Block (which is fine) with each column of B in the B Block (which is not). Intuitively elements of block B are being accessed in a column major order.



```
do_transpose(A, row, column, T):
    for i=0,row,1:
        for j=0,column,1:
            T[j*row+i] = A[i*column+j]
```

This causes unwanted cache misses when the array rows are much more than what the cache line could hold and this happens very often too in larger matrices. To take advantage of the cache locality (row major order in C), we transpose the B matrix and we multiply each row of A with each row of B to get C. As expected transposing matrix gave a boost of approximately 0.25 GFlop/s. Geometric mean comes out to **1.48 GFlop/s**.



On observing the graph we can see that the dips on matrices with sizes 512, 768, 1024 have been stabilized by the transpose.

REGISTER TILING WITH SSE INTRINSICS

Taking advantage of the Intel XEON processors which provide SIMD extensions as SSE intrinsics we can move a level up from L1 cache to the fastest memory - Registers. SSE intrinsics provide load, store and arithmetic

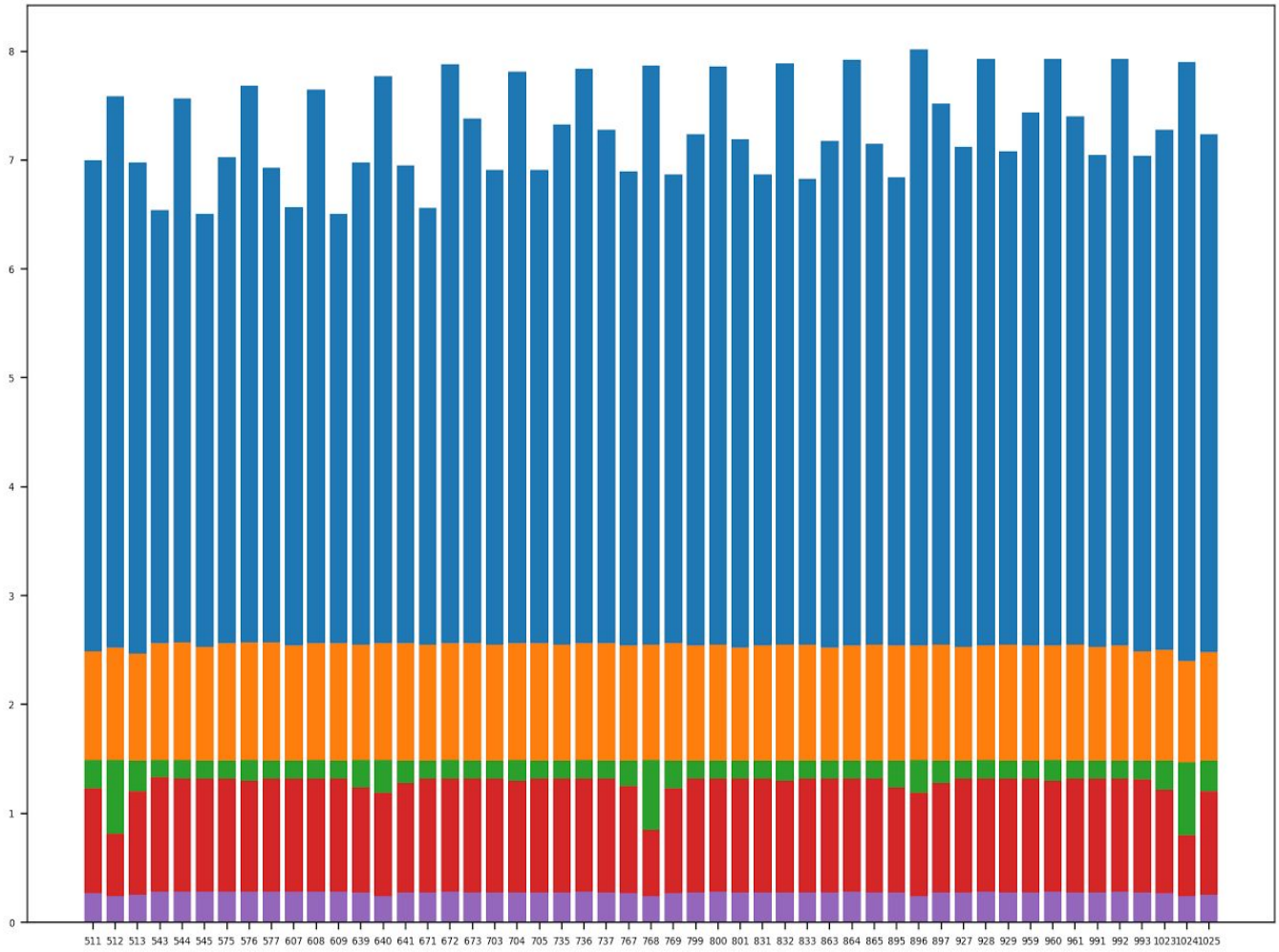
operations which are run as SIMD. This is interesting because the inner loop where the actual arithmetic calculation takes place can be unrolled - doing multiplication on pair doubles on registers.

Being aware of data being unaligned, we use unaligned loads and stores into/from registers from/into memory. Although unaligned loads/ stores are slower, this could give a considerable boost to our computations owing to the SIMD calculations. Let's manually unroll the outer couple loops before seeing how this has given us a boost.

UNROLLING OUTER LOOPS

We can unroll i and j together to get performance boost on top of k. This step gives a performance boost because we reduce load per step. Note that we're still loading/storing unaligned. Because, we accumulate values for four different positions in matrix C we get a fairly decent boost from this performance optimization. At the end of this step, we're already using 8 registers. Bang cluster compute nodes have a maximum of 16 registers.

At the end of Register tiling and unrolling loops we are up on the computational speed to **2.54 GFlop/s**. The plot of our current implementation (orange) VS benchmarks & previous implementations is given below.



Code for unrolling i+=2, j+=2 with SSE Intrinsics for inner loop k

```
register __m128d sum = _mm_setzero_pd();
// 3 more registers
register __m128d a_1, a_2; // for A matrix
register __m128d b_1, b_2; // for B matrix
for(int k=0; k<K; k += 2) {
     a_1 = _mm_loadu_pd(&A[i*K+k]);
     b_1 = _mm_loadu_pd(&T[K*j+k]);
     if (i+1 >= M && j+1 >= N) {
          a_2 = _mm_setzero_pd();
          b_2 = _mm_setzero_pd();
     } else if (i+1 >= M) {
          b_2 = _mm_loadu_pd(&T[K*(j+1)+k]);
          a_2 = _mm_setzero_pd();
     } else if (j+1 >= N) {
          a_2 = _mm_loadu_pd(&A[(i+1)*K+k]);
          b_2 = _mm_setzero_pd();
     } else {
          a_2 = _mm_loadu_pd(&A[(i+1)*K+k]);
          b_2 = _mm_loadu_pd(&T[K*(j+1)+k]);
     }
          sum = _mm_add_pd(sum, _mm_mul_pd(a_1,b_1));
          . . . . for all 4
     }
_mm_storeu_pd(s_i1j1, sum);
```

Note that we are using unaligned load and store intrinsics as opposed to aligned, since our matrices might have non-multiple of 2 sizes as well.

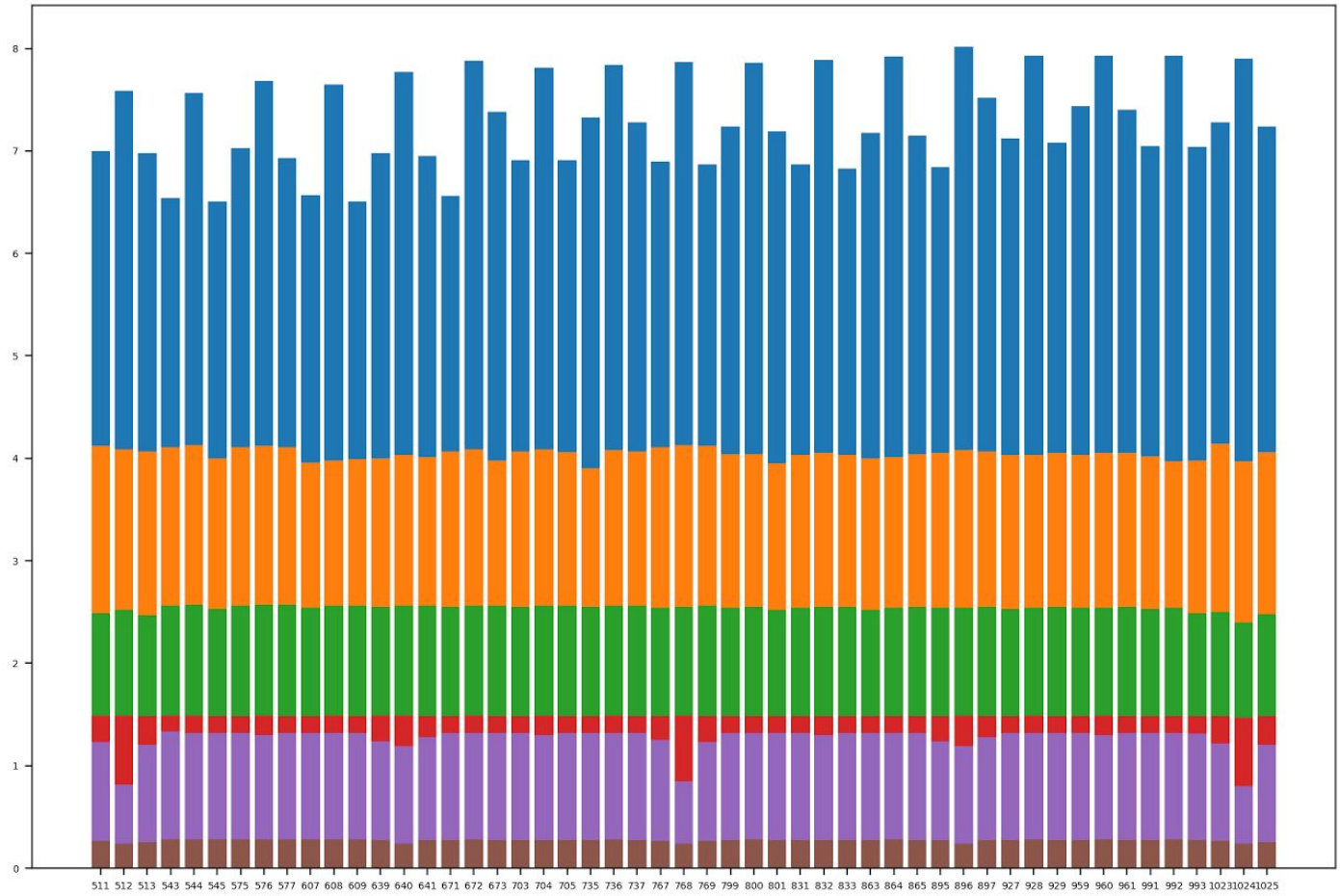
MATRIX BUFFERING AND MEMORY ALIGNMENT

In this step, we explore the possibilities of aligned loads and stores. The SSE intrinsics support pair double aligned loads and stores which are 128-bit length. The problem we have right now is that the matrices which are being multiplied might have odd number of columns which can cause the alignment to mess up when we're blocking. Note that we're using even numbered block sizes. To solve this, we should buffer that into a 0 padded matrix with even number of columns, before using aligned loads to registers. Even though, matrix buffering is a costly operation, the boost given by aligned loads were very significant.

For example, Let's say matrix A is a 4x3 matrix and B is a 3x4 matrix & A and B are to be multiplied. Matrix buffering would basically convert A to a 4x4 matrix by padding last column with zeros. Since we're transposing B, the resultant matrix is a 4x3 matrix. And it's buffered to a padded matrix 4x4 with last column padded with

zeros. Note that we're padding it with zeros, so that during aligned multiply-add this padded column won't have any effect. After memory buffering, aligned loads & stores for registers we're giving a performance of **4.04 GFlop/s.**

Following is the plot between, our current implementation (orange) VS benchmark & previous implementations.



AUTOMATIC PARAMETER TUNING

We wrote a python script that runs the current implementation for all block sizes ranging from 300 - 600 averaging GFlop/s across all matrix sizes to get the best fitting block size for the problem which could take advantage of the 64-byte cache line.

After the automated performance tuning reports, we updated our block sizes to

- **400** - blocking for L2 cache.
- **36** - blocking for L1 cache.

ENVIRONMENT

HARDWARE

Intel Xeon E5354 @ 2.33GHz (Clovertown processor)
8 cores per compute node
Supports Intel SSE3 SIMD extensions.

Memory Hierarchy
16 x 128 bit registers (SSE3)
32 KB L1 instruction cache
32 KB L1 data cache
4 MB L2 cache

SOFTWARE

OS
Linux version 2.6.32

gcc version
4.4.7

FUTURE IMPROVEMENTS

1. A closer look at the graph shows that there are significant dips at certain matrix sizes. Because of fringing effects, for certain matrix sizes a different block size is necessary for the speed up. We could hard code the default block size & run a different algorithm for specific matrix sizes.

2. Another improvement is use the AVX extensions. This could provide a significant boost on the current implementation given that AVX uses 256 bit registers, which is the double of what we're currently using. (AVX provides even more and wider registers)

3. Alternatively we could increase the step size of k to 4 rather than 2. Increasing the step size of k does not save any loads but reduces the number of branches and uses more vector registers: 4 registers to accumulate the increment of cij , 8 vector registers for loading values of A and B, and probably some vector registers for storing intermediate results. By unrolling the loops even further, we can not take advantage of additional vector registers, because the compute node has only 16 vector registers.