Matrix Multiplication: Optimizations Through Cache Blocking and Vectorization

Teofilo Erin Zosa IV
Department of Computer Science and Engineering
The University of California, San Diego
La Jolla, CA 92092
PID: A53242430
tzosa@ucsd.edu

Abstract

In this assignment we were tasked with optimizing square matrix operations on a single CPU core with respect to Gigaflops per second (Gflops/s), with the goal of achieving 60% of the performance on the same operations as the ATLAS library (about 5 Gflops/s).

0.1 Terminology

In this paper:

1. A will refer to the first matrix
2. B will refer to the second matrix
3. AB will refer to the matrix that results as the product of A & B.
4. C will refer to the initially empty matrix that will contain AB at end of the matrix multiplication routine.

1 Method

1.1 Benchmarking CPU Characteristics

Performance metrics were obtained on the UCSD Bang computing cluster, whose computing units are Dell PowerEdge 1950 servers with two quad-core Intel Xeon E5345 processors. Each processor runs at a clock speed of 2.33 GHz and has two 4MB (each shared between two cores) unified L2 caches (16-way set associative). Each core has its own 32KB L1i cache (8-way set associative) and 32KB L2d cache (8-way set associative). All caches have a line size of 64 bytes.World [2017][1]

1.2 Language-level Optimizations

1.2.1 Compiler Optimizations

For program compilation, GCC 4.4.7 was used with the flags:

-O4 -ffast-math -msse3 -mfpmath=sse -ftree-vectorize -funroll-loops -funroll-all-loops

[1] Additional characteristics gleaned from 'lscpu' and 'cat /proc/cpuinfo' commands

1.2.2 Program Optimizations

The program was written in C and all functions inside 'square_DGEMM' were declared with the keywords 'static', 'inline', and 'void' to aid the compiler in optimizing for performance. The Local variables that were reused in the calculation of the Matrix AB (i.e. the element(s) of AB that were being updated) were declared with the 'register' keyword.

1.3 Cache Blocking

As performance is a function of computation, which is in turn a function of operand availability, it is important to structure computation in a way that minimizes the accesses to farther components of the memory hierarchy. In addition to keeping oft-computed values as close to the processor as possible, we can also ensure that computational units have their set of operands fit into fast memory as much as possible. This will ensure that the computational unit causes the least amount of fetches from slow memory. When performing this technique with regard to cache memory, it is known as cache blocking, breaking computation into units whose values fit into cache memory.

1.3.1 L2 Cache Blocking

Given the cache characteristics from section 1.1 and that block size b_{L2} must be $b_{L2} \leq (\frac{M_{fast}}{3})^{\frac{1}{2}}$, this implies optimal block size is $b_{L2} \leq 418$.

Experiments were conducted with values of 128, 256, 543, and 1024 to validate this hypothesis.

1.3.2 L1 Cache Blocking

Following the same reasoning as section 1.3.1, $b_{L1} \leq (\frac{M_{fast}}{3})^{\frac{1}{2}}$, implies optimal block size is $b_{L1} \leq 36$.

Experiments were conducted with values of 8, 16, 32, 64, and 128 to validate this hypothesis.

1.4 Transposing B

As C stores contiguous memory elements in row-major order, B was transposed in place in the beginning of the 'square_DGEMM' function call.

1.5 Unrolling Loops

Loop unrolling has the benefit of calculating multiple values simultaneously which make take advantage of the CPU's instruction pipeline (i.e. loads, stores, and adds can occur in parallel or while waiting for longer latency operations such as multiplies) and or spatial locality, something that matrix multiply memory accesses exhibit.

1.5.1 Iterating Over AB

The first attempt at loop unrolling occurred at during the innermost loop that updates the elements in C. This loop was unrolled to calculate up to 16^2 values for a single element in C simultaneously. This optimization attempt did not produce any significant performance benefit (presumably due to gcc's automatic vectorization).

1.5.2 Iterating Over B^T

The second attempt at loop unrolling again occurred during the innermost loop that updates the elements in C. This time, the middle loop was unrolled so that two neighboring column values of C could be calculated in parallel. This was extended to calculate four neighboring column values in parallel when possible, then fall back to two, which would deal with odd values by performing a final C element calculation serially.

[2] 16 128-bit SIMD registers were available on this CPU

1.5.3 Iterating Over A

The final attempt at loop unrolling occurred during the innermost loop that updates the elements in C. This time, the outer loop was unrolled so that eight values of C (four neighboring column values in each of two neighboring rows) could be calculated in parallel. This was not extended further due to time constraints as well as a suspicion that further increasing this value would require more operations than the available SIMD registers could accommodate.

2 Results

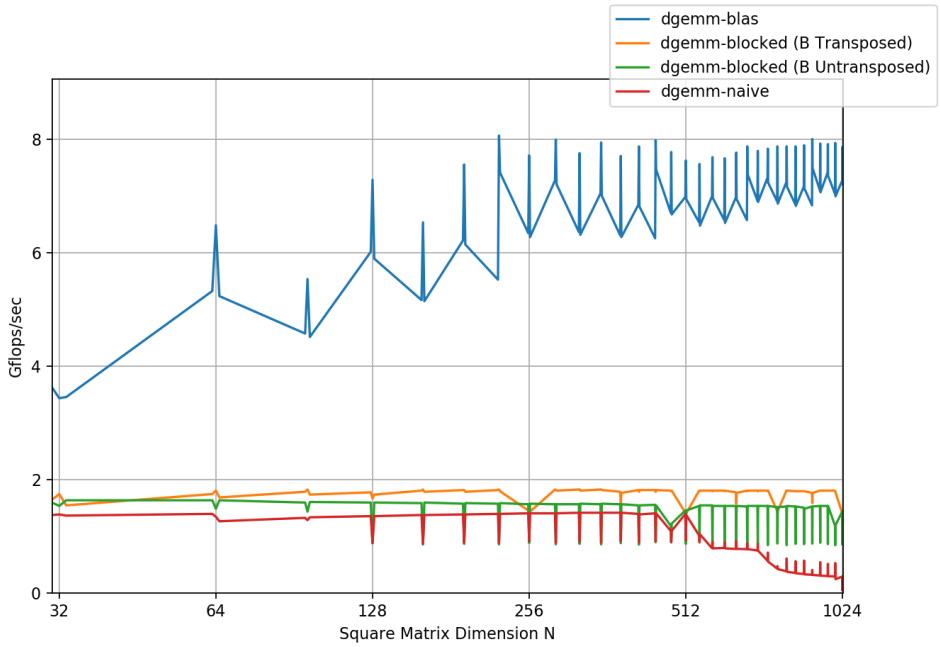

Figure 1: Performance as a Function of Transposing B

From figure 1, we can see that transposing B did have benefits with respect to total Gflop/s as well as consistent performance across matrix sizes.

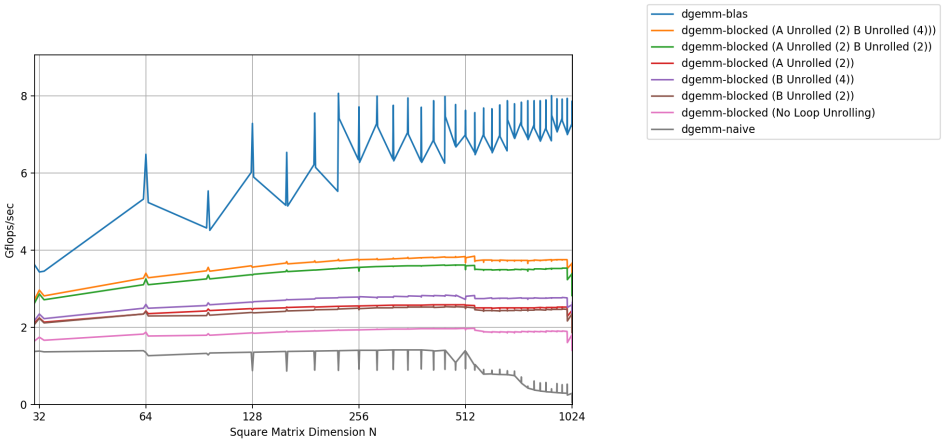

Figure 2: Performance as a Function of Unrolling Loops

As we can see from figure 2, unrolling A and B produced significant performance benefits. Unrolling either A's or B's loop alone by two resulted in equivalent performance gains. Unrolling both A and B loops by two resulted in a more sizeable performance increase than unrolling B's loop alone by four.

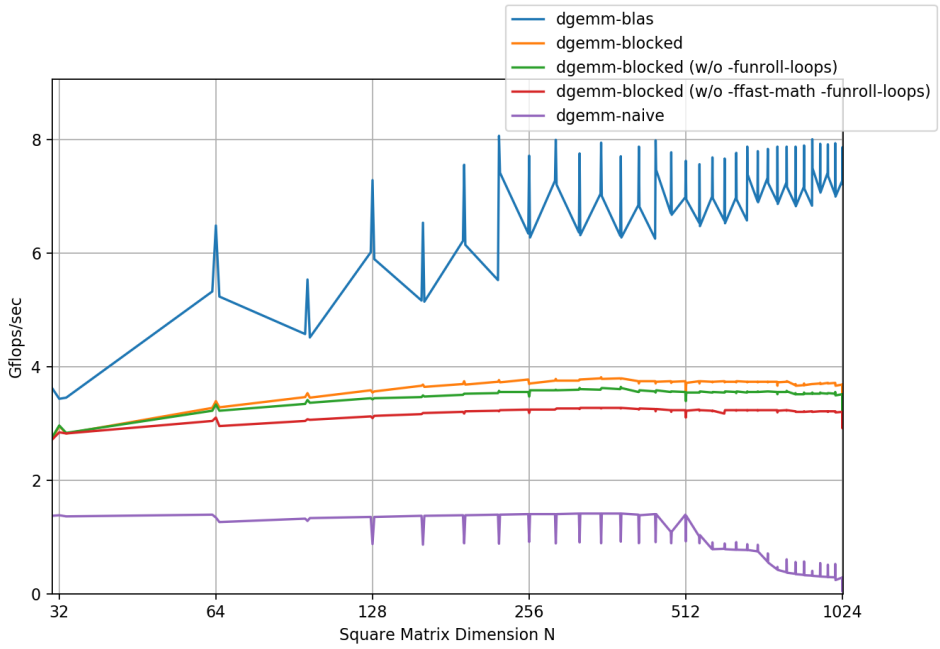

Figure 3: Performance as a Function of GCC Flags

From figure 3 we see that certain compiler flags, notably '-ffast-math' and '-funroll-loops', did in fact impart noticeable performance on their respective compiled programs. Programs compiled without either of these flags achieved about 3.0 Gflops/s. Those compiled with '-ffast-math' achieved around 3.5 Gflops/s, and those compiled with both '-ffast-math' and '-funroll-loops' achieved about 3.7 Gflops/s.

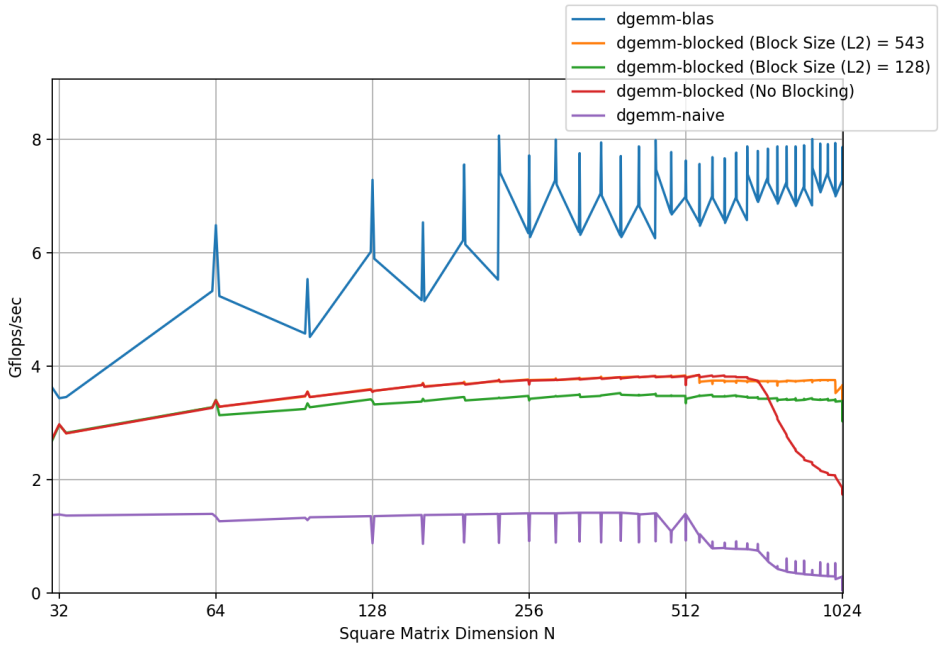

Figure 4: Performance as a Function of b_{L2} ($b_{L1} = 543$)

In figure 4 we find that, although the theoretically optimal value should have been $b_{L2} \leq 418$, in practice, this was not the case, presumably due to compiler optimizations. Instead, $b_{L2} = 543$ produced the best results (this value was chosen for the final value of b_{L2} due to this fact) .[3]. Of interest, we also see that performance for the non-blocking version of the program degraded at about the same dimension of N as the naive matrix multiplication program.

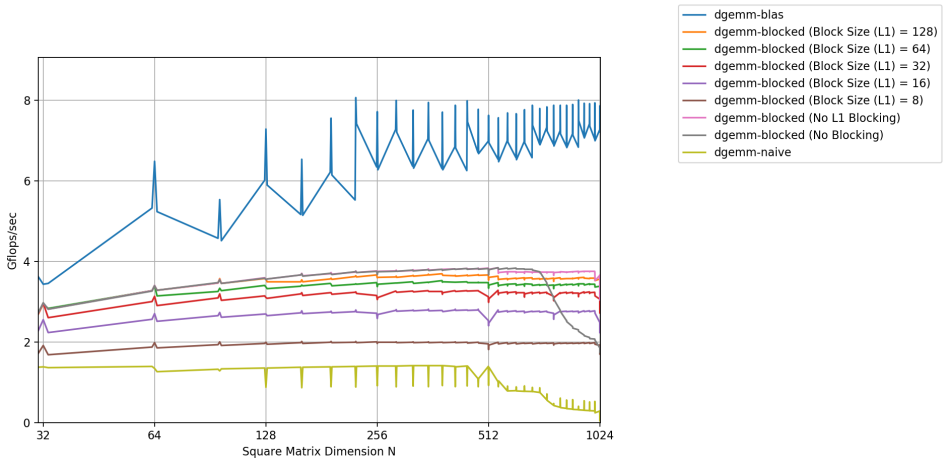

Figure 5: Performance as a Function of b_{L1} ($b_{L2} = 543$)

We find from figure 5 that, again, the optimal value for the L1 cache block size differed from the theoretically optimal block size of $b_{L1} \leq 36$. Instead, quite surprisingly, performance was positively associated with increasing values of b_{L1} until $b_{L1} = b_{L2}$, at which point increasing size did not improve performance. This makes sense as b_{L1} can never exceed b_{L2}.

[3] $b_{L2} = 256$ performed slightly worse than $b_{L2} = 543$ and $b_{L2} = 1024$ achieved the same performance as $b_{L2} = 543$

3 Analysis

3.1 Program Behavior

Algorithm 1 DGEMM Blocked

Input:

The square matrices A and B of dimensions NxN that we are multiplying together.
An empty (i.e. filled with all zeros) NxN matrix C which we will store elements of the matrix AB into.
Block sizes $block_{L2}$ and $block_{L1}$ which are used to determine in the block-stride at level 2 and level 1 of the cache blocking, respectively.

Output: AB stored in C.

Neither A nor B are edited.
procedure SQUARE_DGEMM(N, A, B, C) ▷ The product of NxN matrices A, B stored in C
    $B \leftarrow B^T$ ▷ Transpose B in place
    $blockSize \leftarrow block_{L2}$
    for $A_{row} = 0; A_{row} < N; A_{row}$+=$blockSize$ **do**
        for $B_{row} = 0; B_{row} < N; B_{row}$+=$blockSize$ **do**
            for $k = 0; k < N; k$+=$blockSize$ **do**
                `do_L1_block`
    ▷ Do $DGEMM_{blocked}$, with $blockSize = block_{L1}$ and $N = min(N, blockSize)$, and then call do_block

    $B \leftarrow B^T$ ▷ Transpose B back prior to exit
return E_n;

The program takes in matrices A, B, and C as contiguous one-dimensional arrays as well as N. B is transposed in place and the matrices are broken into outer-level blocks of size $b_{L2}xb_{L2}$, which are in turn broken into inner-level blocks of size $b_{L1}xb_{L1}$. The inner-level blocks of A and B are then iterated over to produce intermediate values for their corresponding elements in C. See figure 1 and 2 and section 1 for specific details.

3.2 Development Process

I initially began optimizing the matrix multiplication routine by implementing the transposition of B. I then began adding compiler flags until performance plateaued. I then added in a second level of cache blocking. I then made my first attempt at loop unrolling by manually unrolling the inner loop which calculated elements over C did not result in better performance, presumably also due to GCC's autovectorization.

3.2.1 SSE Intrinsics

Next, I attempted to vectorize code by hand through SSE intrinsics. This attempt was fraught with days of extreme pain and anguish. Aligned loads (i.e. '_mm_load_pd') were much faster (0.5 Gflops/s) than unaligned loads (i.e. '_mm_loadu_pd'), but only worked on even-dimension arrays. This is due to the fact that SIMD instructions must be 16-byte aligned. To deal with this fact, I manually tried to align the matrices through the initialization of new arrays that were declared to be 16-byte aligned and the use of 'memcpy'. This was unsuccessful for arrays that were declared to have an even number of elements and those without. This strategy only worked if odd-dimension matrices were copied row by row into a new, zero-padded even-dimension array. This resulted in a severe penalty to performance, with the program performing worse than the naive program. Only utilizing unaligned loads resulted in worse performance than a non-SIMD version. This was true even when utilizing all 16 SIMD registers to calculate a single element of C in parallel. It was at this point that I abandoned the use of SSE intrinsics.

Algorithm 2 Matrix Multiply Routine Over Blocks

Input:

1 Sub-blocks A_b and B_b of the matrices A and B, respectively of dimensions $block_{L1}xblock_{L1}$.

2 Sub-block of the matrix C corresponding A_b and B_b which has the intermediate results of previous blocks' operations.

3 $block_{L1}$ which is used to determine the end point of a block.

Output: AB stored in C.

4

5 **procedure** DO_BLOCK(N, I, J, K, A, B, C) ▷ The sub-product of NxN matrices A, B stored in C

6 $blockSize \leftarrow block_{L1}$

7 **for** $A_{row} = 0; A_{row} < blockSize - 1; A_{row}$+=2 **do**

8 **for** $B_{row} = 0; B_{row} < blockSize - 3; B_{row}$+=4 **do**

9 **for** $i = 0; i < 4; ++k$ **do**

10 $cij[i] = C[A_{row}][B_{row} + i]$

11 $c_{i+1}j[i] = C[A_{row} + 1][B_{row} + i]$

12 **for** $k = 0; k < blockSize; ++k$ **do**

13 $a \leftarrow A[A_{row}][k]$

14 **for** $i = 0; i < 4; ++k$ **do**

15 $cij[i]$+=$a * B[B_{row}][k]$

16 $a1 \leftarrow A[A_{row} + 1][k]$

17 **for** $i = 0; i < 4; ++k$ **do**

18 $c_{i+1}j[i]$+=$a1 * B[B_{row}][k]$

19 **for** $i = 0; i < 4; ++k$ **do**

20 $C[A_{row}][B_{row} + i] = cij[i]$

21 $C[A_{row} + 1][B_{row} + i] = c_{i+1}j[i]$

22 ▷ Remaining rows of A_b are handled sequentially.

23 ▷ Remaining rows of B_b are handled in pairs, and then sequentially.

3.2.2 Program Optimizations

In tandem with the attempts to hand-vectorize code, I also took advantage of C-specific keywords. I declared all functions as 'static inline void' and used the 'register' keyword when declaring the variables which would store the temporary values for the elements of C which we were iterating over.

3.2.3 Loop Unrolling

Next I realized I could unroll the loops in a different way to calculate multiple elements of C in parallel. It was then that I found that unrolling the loop over B did result in better performance. Iterating in pairs resulted in 1 Gflop increase (1.8 to 2.7). Iterating in multiples of fours resulted in a 0.1 Gflop increase (2.7 to 2.8). I then attempted to unroll the loop over A, which produced a 1 Gflop increase (2.7 to 3.7). Due to time constraints, I was not able to test how much more performance could be gained from this approach.

3.3 Supporting Data

3.3.1 Optimal L1 Cache Block Size

Two-level blocking having a negative effect on performance was a paradoxical finding and perhaps is again due to the autovectorization that GCC performed on the program. Perhaps when $b_{L1} = b_{L2}$, GCC realized the blocking loops were the same, meaning the second level of blocking was not doing any useful work. It could then remove it and instead optimize the matrix multiplication loops.

In both figure 4 and 5 we see optimal performance when we let $b_{L1} = b_{L2} = N$ until we get to $N \geq 700$. Presumably it's at this point that fetches to memory become more frequent, severely degrading performance.

4 Future Work

In the future, it would be interesting to delve deeper into how ATLAS handles its matrix multiplies and also what combination of unrolling results in the best performance.

References

CPU World. Intel xeon e5345 specifications, 2017. URL `http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5345%20-%20HH80563QJ0538M%20(BX80563E5345A%20-%20BX80563E5345P).html`.