Double-precision General Matrix Multiply (DGEMM)

Parallel Computation (CSE 260), Assignment 1

Andrew Conegliano (A53053325) Matthias Springer (A99500782)
GID G-338-665

January 22, 2014

0.1 Assumptions

The following assumptions apply within this work.

- All matrices are square matrices.
- All matrices consist of double-precision floating point values (64-bit doubles).

0.2 Notation

In this work, we use the following notation and variable names.

- n is the size of one dimension of the matrices involved. I.e., every matrix has n^2 values.
- When referring to matrices, A and B denote the source matrices and C denotes the target matrix, i.e. $AB = C$.
- b_i, b_j and b_k denote the block size for each *dimension* i, j, and k, respectively[1].

1 Runtime Environment

We optimized our DGEMM implementation for a specific runtime environment. All benchmarks and performance results are based on the following hardware and software.

1.1 Hardware

- Intel Xeon E5354 @ 2.33GHz (*Clovertown* processor)
  - 2 *Woodcrest* Core2 dies
  - 2 sockets per chip
  - Supports SSE, SSE2, SSSE3
- Memory hierarchy
  - 32 KB Level 1 cache

[1] We tried multiple levels of blocking and it is evident from the context which level of blocking we are refering to.

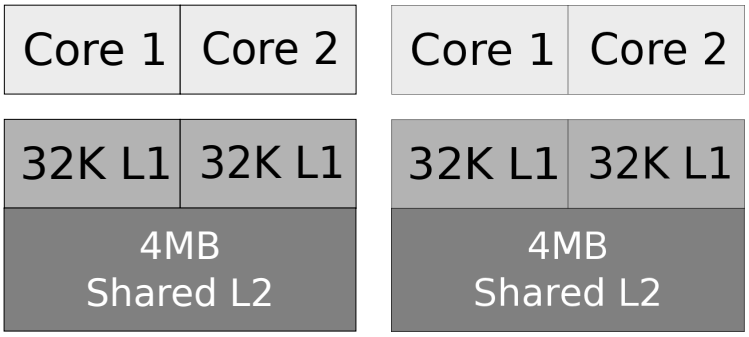

Figure 1: Architecture of the CPU

```
for i = 0 .. n - 1
     for j = 0 .. n - 1
          cij = C[i*n + j]

          for k = 0 .. n - 1
               cij += A[i*n+k] * B[k*n+j]

          C[i*n+j] = cij
```

Figure 2: Naive matrix multiplication algorithm.

- 4 MB shared Level 2 cache
- 64 byte cache line
- 16 × 128 bit SSE3 registers

1.2 Software

- CentOS 6.3 Linux (Kernel `2.6.32-358.18.1.el6.x86_64`)
- Built with GCC 4.7.3
- Compiler flags: `-foptimize-register-move -O3 -msse -funroll-loops -msse2 -msse3 -m3dnow -mfpmath=sse -mfpmath=sse -malign-double`

2 Basic Algorithm

The naive implementation of the matrix multiplication algorithm consists of three nested `for` loops, where every loop runs from 1 to n. In the innermost loop, a single addition and multiplication is done. Therefore, the runtime complexity of this algorithm is $\mathcal{O}(n^3)$ floating-point operations. We did not use an algorithm with a lower runtime complexity[2].

Figure 2 shows the basic matrix multiplication algorithm. We will reference this algorithm throughout the Section 4 and explain why our optimizations can increase the performance of the basic algorithm.

[2]Strassen's algorithm has a runtime complexity of $\mathcal{O}(n^{2.8})$.

```
for (int i = 0; i < BLOCK_SIZE; i++)
     for (int j = 0; j < BLOCK_SIZE; j++)
          for (int k = 0; k < BLOCK_SIZE; k++)
               C[i][j]+=A[i][k]+B[k][j];
```

3 Optimizations

In this section, we will describe and evaluation optimizations of our DGEMM algorithm.

3.1 Blocking for L1 Cache

To increase locality, we implemented blocking. This restricts the computations into chunks that that fit inside the cache. In the basic algorithm, we read `A` row by row left-to-right. `B`, on the other hand, is read column by column from the top to the bottom. Every time we access a value in a matrix, we prefetch the next 7 elements in that row[3], in case we have a cache miss.

For fringe cases, i.e. the edges where it does not divide evenly into bock size, are handled by calling the naive implementation. Because the ratio of the number of fringe cases to the number of non fringe cases is so slow as matrix size grows, performance is only slightly affected for smal matrix sizes.

3.2 Blocking for L2 Cache

The size of the Level 2 cache is 4 MB. Therefore, we can store $\frac{4 \cdot 1024^2}{8}$ matrix entries in the L2 cache. Since we have three matrices `A`, `B`, and `C`, blocking for the L2 cache can increase the performance only if $n > \sqrt{\frac{4 \cdot 1024^2}{8 \cdot 3}} = 418$. In our benchmarks, we did not deal with matrices bigger than 1200 entries. Therefore, we do no blocking for the L2 cache in our final implementation. We implemented L2 blocking in an intermediate version and it lead to a performance decrease, especially for small matrix sizes.

3.3 Matrix Transposition

To take advantage of cache locality, we transpose the matrix `B` before running the actual algorithm. In the basic version of the algorithm, we access `B` column by column. For sufficiently big matrices, every subsequent access into `B` triggers a cache miss and loads a new cache line into the L1 cache. The L1 cache is 32 KB big, resulting in $\frac{32 \cdot 1024}{64 \cdot 3} = 170$ cache lines per matrix. When we start accessing the 171th row, the first row is evicted from the cache[4]. This problem is solved by L1 blocking as long as the block size is smaller or equal to 170. For bigger block sizes, transposing the matrix helps, because when reading the matrix row by row, we have a lot of cache hits after every cache miss due to prefetching.

3.4 Register Blocking and Loop Unrolling

3.5 Matrix Buffering

Matrix buffering is the process of duplicating the whole matrix or a part of the matrix in the memory. `B` is buffered entirely before the run of the actual algorithm, as part of the matrix transposition. `A` is buffered partly: when blocking for the L1 cache, we buffer `A` block by block, i.e. we only buffer the parts of `A` that are contained the current block of `A`. We tried buffering the matrix `C`, as well, but it did not result in a performance increase.

[3] A cache line contains 8 doubles.

[4] Assuming a LRU replacement strategy.

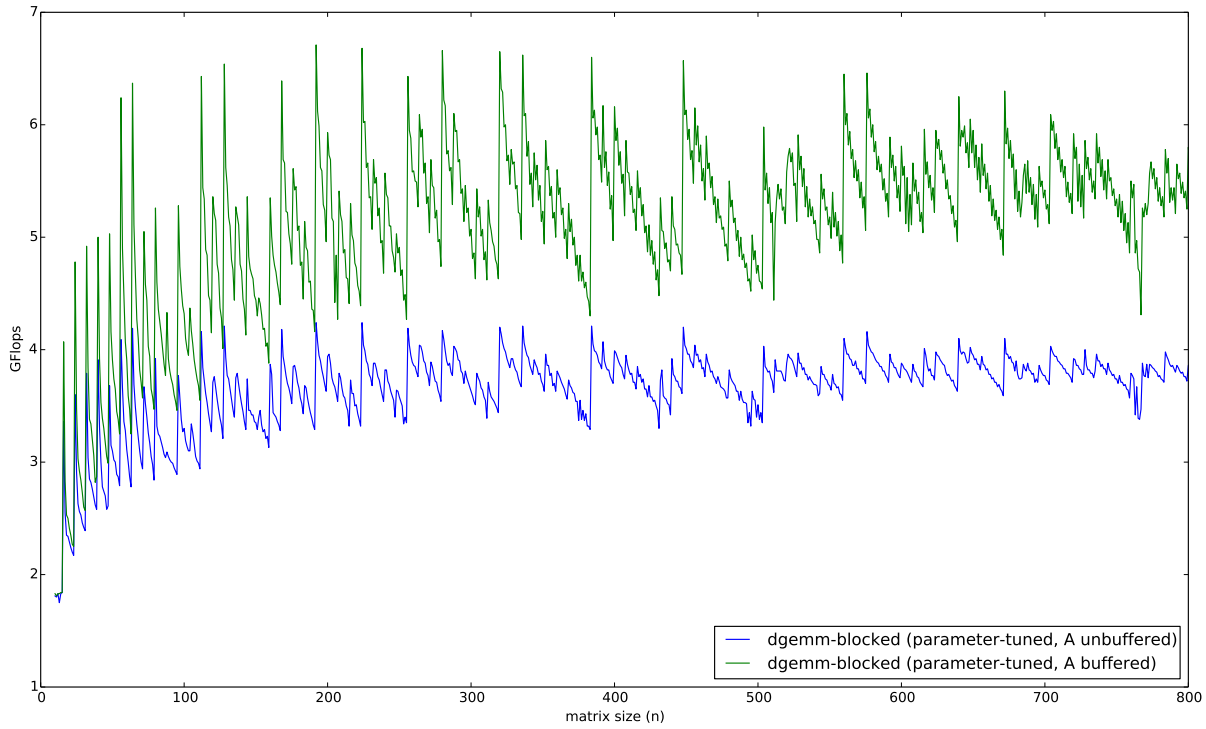

Figure 3: Performance of our parameter-tuned blocking version, with and without buffering `A`.

3.5.1 Memory Alignment

The buffers for `A` and `B` are 16-byte aligned. This is important for vectorization, because it allows for aligned loads that are considerably faster than unaligned loads. In case the matrix size n is odd, we add a 8 byte padding at the end of each row. Therefore, every row is 16-byte aligned[5].

3.5.2 TLB Cache

In general, matrix buffering is sometimes used to minimize TLB cache misses. The copied part of the matrix is often smaller and fits in less memory pages. We chose the block sizes for L1 blocking in such a way that every block of `A`, `B`, and `C` fits entirely into the L1 cache. The CPU that we optimized for has a primary (first-level) TLB cache of size 64 and a secondary (second-level) TLB cache of size 512. Since the L1 cache can hold at most $\frac{32 \cdot 1024}{64} = 512$ cache lines, the TLB cache is big enough to cover a block of all three matrices at the same time.

3.5.3 Performance

Figure 3 compares the performance of our fully-optimized implementation with and without buffering `A`. Note, that for the unbuffered version, we had to use unaligned loads for `A` in the vectorized code, because the matrix size n could be odd.

[5]We are reading both `A` and the transposed version of `B` row by row.

3.6 Streaming SIMD Extensions (SSE) and Loop Unrolling

SSE provides a set of vector registers that can contain multiple numbers at a time and support arithmetic operations on the whole vector at the same time. We tried and evaluted several implementations with a different number of vector registers and different levels of loop unrolling.

3.6.1 Unrolling `k += 2`

Our first approach was to vectorize the innermost `for` loop and unroll parts of the loop for `k += 2`.

```
for i = 0 .. b_i - 1
    for j = 0 .. b_j - 1
        cij = zero vector

        for k = 0 .. b_k - 1 step 2
            a = load(A + i*lda + k)
            b = load(B + j*lda + k)
            cij = cij + a*b

        C[i*lda + j] += cij[0] + cij[1]
```

Figure 4: Basic block function with unrolling for `k += 2`.

This leads to a significant performance increase, for the following reasons.

- SSE multiplications and additions operate on two doubles at the same time. Therefore, we can in theory process twice as many data at the same time.
- In line 13, the processor can multiply and add at the same time, because it has fused multiply-add unit.
- The loop in line 9 is executed fewer times, because we increased the step size. Therefore, we have less branches in the assembly code and less additions of the loop variable `k`.

We tried increasing the step size and loading more values of `A` and `B` at the same time, in order to have less branches in the assembly code and to take advantage of more vector registers. In the previous code we used only 3, maybe 4[6], registers, but the CPU has 16 vector registers. Increasing the step size to 4 or 8 had, however, only a small effect on the performance. Unrolling `i` and `j` instead, yields a much greater performance increase.

3.6.2 Unrolling `j += 2, k += 2`

We could increase the performance by unrolling both `j += 2` and `k += 2` at the same time.

[6]Depending on how the compiler translates line 16.

```
for i = 0 .. b_i - 1
    for j = 0 .. b_j - 1
        cij_1 = zero vector
        cij_2 = zero vector

        for k = 0 .. b_k - 1 step 2
            a = load(A + i*lda + k)
            b_1 = load(B + j*lda + k)
            b_2 = load(B + (j+1)*lda + k)
            cij_1 += a * b_1
            cij_2 += a * b_2

        C[i*lda + j] += cij_1[0] + cij_1[1]
        C[i*lda + j + 1] = cij_2[0] + cij_2[1]
```

Figure 5: Optimized block function with unrolling for `j += 2, k += 2`.

Note, that in comparison to Figure 4, the value `a` is read only once instead of twice. That is the main reason why this code is faster. Besides, we save some branches and arithmetic operations due to the increased step size of `j`. In total, we use 5, maybe 7, vector registers. This suggests that it may be useful to do more loop unrolling. Unrolling for `j += 4, k += 2` or `j += 2, k += 4` had only little effect on the performance.

3.6.3 Unrolling `i += 2, j += 2, k += 2`

The code for unrolling all three loops adds another load of `A + (i+1)*lda + k` to the code in Figure 5. Besides, we need two more vector registers `cij_3` and `cij_4`, because we accumulate values for four different positions in `C`. This optimization incrases the performance, because `b_1` and `b_2` are loaded only once, for two different values `a_1` and `a_2`. We use at least 8 vector registers at this point.

3.6.4 Unrolling `i += 2, j += 2, k += 4`

This is what we did in the final version of our implementation. Increasing the step size of `k` does not save any loads but reduces the number of branches and uses more vector registers: 4 registers `cij_*` to accumulate the increment of c_{ij}, 8 vector registers for loading values of `A` and `B`, and probably some vector registers for storing intermediate results for `_mm_unpackhi_pd`. By unrolling the loops even further, we can not take advantage of additional vector registers, because the CPU has only 16 vector registers.

3.6.5 Performance

Figure 6 shows that, in most cases, unrolling for `i += 2, j += 2, k += 4` performs best. Therefore, we use this version in our implementation.

3.7 Parameter Tuning

Parameter tuning is the task of trying (brute-force) a set of parameters that affect the performance.

3.7.1 Motivation

As shown before, for certain matrix sizes n, the blocked algorithm performes better with certain block sizes. One obvious reason is that for a fixed matrix size n, the block size determines the size of the fringe cases. The code for fringe cases is less optimized than the code for regular cases and fringe cases might take less

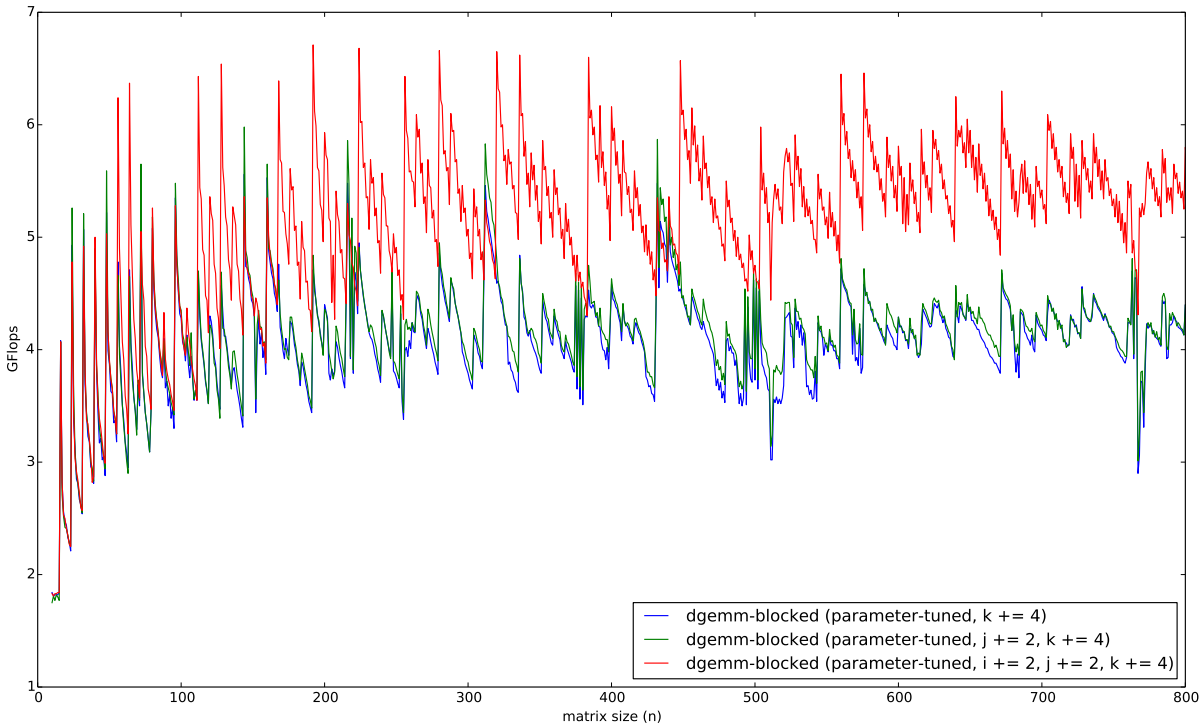

Figure 6: Performance of our parameter-tuned blocking version, with different levels of loop unrolling.

advantage of the cache. Also remember, that a cache line is 64 bytes long and the size of a fringe case might not be divisible by the cache line size, i.e. a cache line is eventually not filled entirely with data from the fringe case. Therefore, we want to keep the fringe cases as small as possible. Let f_i be the size of the fringe cases in dimension i[7].

$$f_i = n \mod b_i$$

For example, for $n = 372$, $b_i = 16$ might be a better block size than $b_i = 32$, because it results in fringe cases of size 4 instead of 20.

3.7.2 Search Space

We tried values for b_i, b_j and b_k for the L1 cache blocking. Due to certain implementation details, b_i and b_j must always be the same value. A cache line is 64 bytes (8 doubles) long. Therefore, in order to reduce the search space, it makes sense to only take a look at multiples of 8. We tried all possible combinations of $b_i = b_j, b_k \in \{8, 16, 24, 32, 40, 48, 56, 64\}$, resulting in 64 parameter combinations. We ran the program for every matrix size from $n = 10$ to $n = 1200$. For every value of n, we then selected the parameters b_i and b_k with the highest performance.

3.7.3 Implementation

We hard-coded the block size parameters for all matrix sizes from $n = 10$ to $n = 1200$. Based on the matrix size, a different version of the algorithm with the respective block size is run. For all other matrix sizes, we use $b_i = b_j = 8$, $b_k = 64$, because it performed good on average.

[7]The same argument applies for the other dimensions.

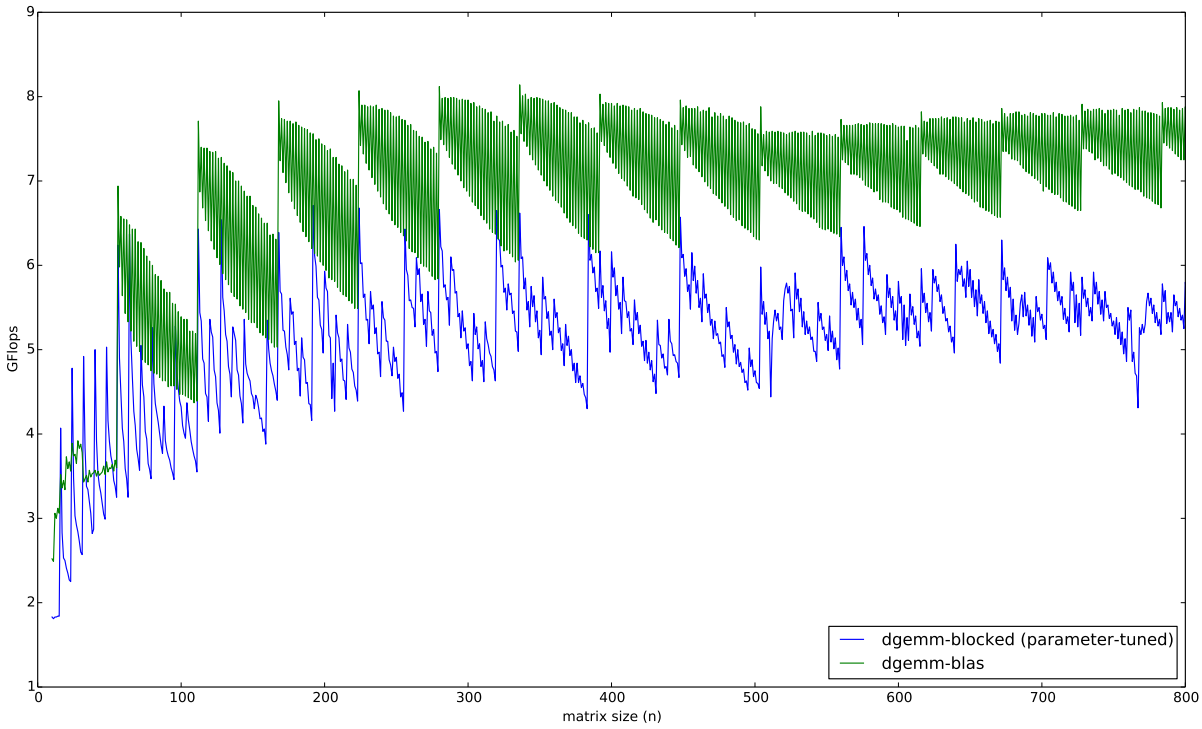

Figure 7: Performance of our parameter-tuned blocking version, in comparison to ATLAS.

3.7.4 Results and Evaluation

Figure 7 shows the performance of the parameter-tuned version of our implementation. It is interesting to see, that for some very small matrix sizes, our implementation is faster than ATLAS. The Appendix contains some more graphs without parameter tuning, but for fixed block sizes. We can observe that the performance of the algorithm reaches a peak when the matrix size is a multiple of one or two of block sizes. In that case, we have no fringe cases.

3.8 Constant Propagation and Code Duplication

During parameter tuning we collected certain values b_i, b_j, and b_k that perform well with certain matrix sizes. Instead of passing these parameters as arguments, we duplicate the source code with preprocessor macros and plug in the block size values as constants. This increases the size of the program and its compilation time, but also the performance. When dealing with known constants, the compiler can use some additional optimizations like further loop unrolling or *strength reduction*, i.e. replacing expensive operations like `i * 8` by less expensive operations like `i << 3`.

We also introduced variables for indices of `A`, `B`, and `C`, instead of recomputing the position on every array access. These index variables are updated after running a loop. This can improve the performance a little bit, because a single addition may be cheaper than some multiplications.

Appendix

Performance Tuning