CS267 - Homework 1 - Single-core matrix multiplication

Ankur Mahesh - Praj Mohan - Quentin Nicolas

February 2023

We attempted the following optimization techniques: Loop reordering, using a SIMD micro-kernel, adding an extra level of blocking and repacking the input matrices. The last thing we did was to tune the block sizes. Overall, the SIMD microkernel accounted for most of the speedup we obtained. Here is a graph of the best performance we could reach, compared to BLAS and to the naive algorithm:

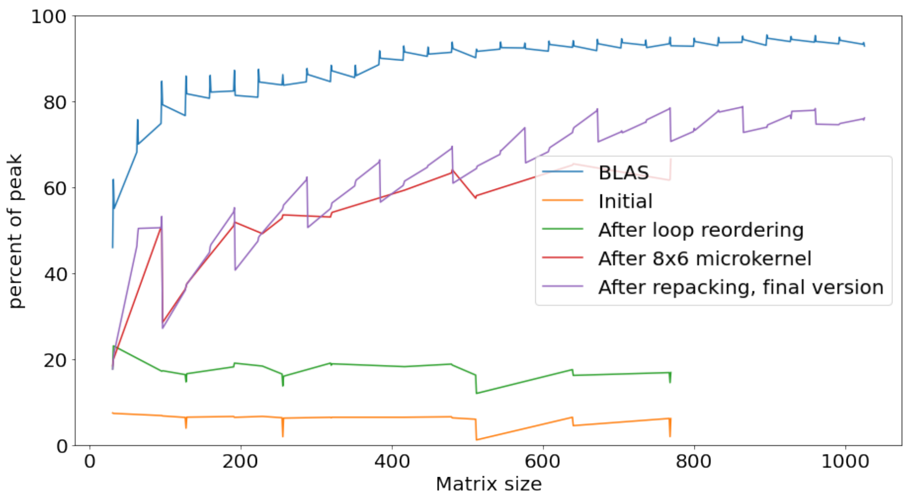

Figure 1: Performances (in % peak) as a function of matrix size for the naive implementation, our implementation and BLAS.

Optimization 1: Loop reordering

Loop reordering at the highest level (when looping over blocks) offered little improvement. At the lowest level, we expect the best possible order to be J - K - I (outer loop over j, then over k, then over i), where indices correspond to the following convention:

$$C_{ij} = C_{ij} + A_{ik}B_{kj} \tag{1}$$

Indeed, looping first over j means we access elements of B and C in a contiguous manner (as they are stored in column-major order). Then, looping over k means we also access columns of A contiguously, and means that the innermost loop (over i) will access different elements of C at each iteration, allowing some instruction-level parallelism.

We indeed observed that this setup performed best, and obtained significant speedup: the naive blocked implementation had a mean performance of 6% of peak on the small subset of matrices, while the JKI setup yielded 17%. Looping over k, then j, then i, gave a mean performance of 16% of peak.

Optimization 2: SIMD Microkernel

Each x86_64 CPU has 16 256-bit registers (that can each hold four doubles). Our goal, in this step, is to write a set of vectorized instructions (using AVX intrinsics) to perform the multiplication of small matrix blocks, using registers the most efficiently. The best implementation we found is as such:

- We operate on blocks of C of size 8×6.
- Our innermost loop is actually performed over k.
- The instruction set is as such:
  1. Load the 8×6 block of C in 12 registers, using `_mm256_loadu_pd` (for non-aligned data) or `_mm256_load_pd` (for 64-bit aligned data, when the matrices have been repacked first). Call this block $C_{1..8,1..6}$ Then, for each k (iterate over the larger block width):
  2. Load $A_{1..8,k}$ in two registers.
  3. Load $B_{k,1}$ and broadcast to one register (using `_mm256_set1_pd`)
  4. Use two FMA instructions to perform $C_{1..4,1} = C_{1..4,1} + A_{1..4,k} \times B_{k,1}$ and $C_{5..8,1} = C_{5..8,1} + A_{5..8,k} \times B_{k,1}$
  5. Repeat the above two operations with $B_{k,2}, B_{k,3}, ..., B_{k,6}$ (with all operations written explicitely, no looping over j)
  6. Store the registers holding $C_{1..8,1..6}$ back into memory (`_mm256_store_pd` or `_mm256_storeu_pd`).

This way, we use at most 15 vectors at a time and they can all fit in registers. We had initially tried a different approach where we would multiply 4×4 blocks of A and B (rather than a $8 \times K$ block of A and a $K \times 6$ block of B), but this proved less efficient. Indeed, the initial method was writing back and forth elements of C between memory and registers, whereas the present one loads them to registers, performs all operations on them, and writes them back only once.

Using a microkernel requires padding for matrices whose size is not a multiple of 24 (least common multiple of 6 and 8). Our first implementation simply copied A, B and C from the start into matrices of size $\lceil lda/24 \rceil \times 24$ (with zero padding at the edges). We obtain very significant speedup, with a mean performance of around 50% of peak on the smaller subset of matrices - with 60-65% of peak for the bigger matrix sizes, where the $\mathcal{O}(n^2)$ cost of copying becomes negligible.

Optimization 3: second level of blocking - repacking

The L1 cache holds 64 kilobytes, which theoretically can theoretically hold three matrices of size 51 by 51 with double precision. Similarly, the L2 cache, with 0.5 MB, can hold three matrices of size 146 by 146. We thus add a second level of blocking, where the outermost loops break up matrices in blocks of size `L2_BLOCK_SIZE`, and the next level breaks them up in blocks of size `L1_BLOCK_SIZE`.

We also repack the matrices at the very start. We used `_mm_malloc` so that the matrices are 64-byte aligned (and we can use AVX intrinsics for aligned data). Matrices are repacked in a block-contiguous way (see figure 2), and blocks are ordered in column-major order. At the repacking stage, we also pad the matrices with zeros so that their length is a multiple of 24. This is inefficient for small matrices, but further efforts (see below) gave worse results overall.

Using these two optimizations, we got overall performances of 55-60% of peak. Significant improvement was obtained for larger matrix sizes, with performances close to 80% of peak for some sizes. More on block size tuning in the next section.

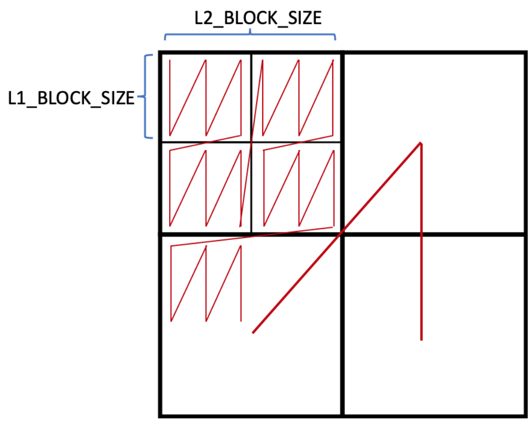

Figure 2: Our repacking layout

We also tried an approach where blocks were repacked right before being used (i.e. lower down in the pipeline, and not a the very start). When getting to the L1 block level, blocks of A, B and C was repacked so that their accesses at the microkernel level were contiguous in memory, i.e. using a different repacking layout for each (see figure below, taken from `https://www.cs.utexas.edu/users/flame/BLISRetreat2019/images/BLIS_gemm.png`). Unfortunately, we were not able to obtain better performances with this scheme, so we sticked to the simpler one.

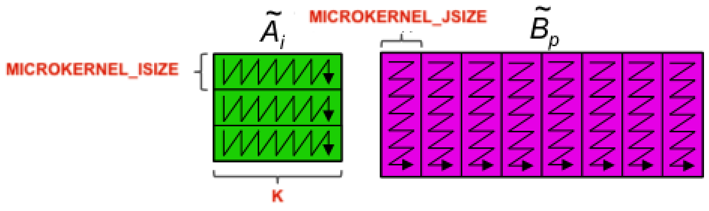

Figure 3: Attempted repacking layouts for A and B and C

Optimization 4: Tuning block sizes

The last optimization we performed was a grid search through all block size combinations, for different fixed matrix sizes. We only searched for block sizes that were a multiple of 24, so that they would be divided equally in small blocks at the microkernel level.

Figure 4 shows the performance (in % peak) for two matrix sizes (200 and 1000) as a function of block size combination. The performance seems to depend very little on block size when the L1 block size is bigger than 48 (the range of L2 block sizes tried is between 96 and 576). This is probably because our repacking method is not performed right. We selected `L1_BLOCK_SIZE` = 48 and `L2_BLOCK_SIZE` = 144, because it appeared optimal for lda= 1000 and it respected the expected bounds given by the cache sizes.

Performance dips

Some obvious performance dips are visible in Figure 1 on our best implementation. These dips all correspond to matrix sizes just above multiples of 24: for such sizes, our large padding is costly because we end up multiplying matrices of size lda+23 instead of lda. Using a more clever repacking, or even masked instructions at the microkernel level, cound help remedy these dips.

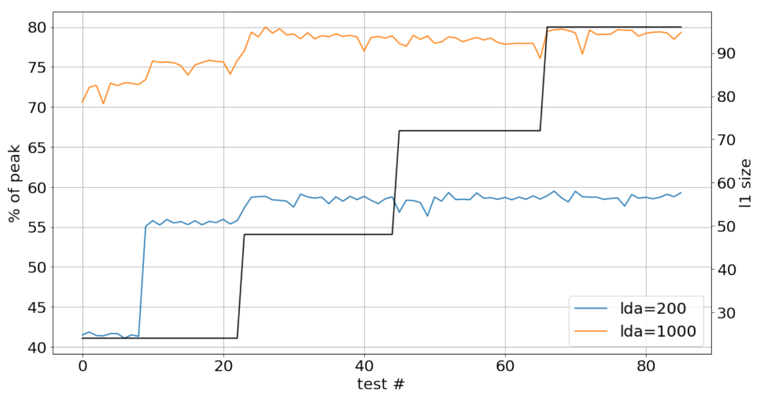

Figure 4: Performance of dgemm for lda=200 and 1000, as a function of block size combination. Combinations on the x-axis are organized as (24,96),(24,120),(24,144),...,(48,96),(48,120),...,(96, 552),(96, 576)