CS267 HW 1

Optimizing Matrix Multiplication

Members & contribution

- **YuXuan Liu**: I wrote the initial AVX implementation, tried changing the row-column ordering, implemented padding, tried masked instructions, tried loop unrolling, tried compiler optimizer flags, simple tweaking of the block size.
- **Jingran Zhou**: I implemented three levels of blocking (i.e., L1, L2, and register), extended the AVX implementation to utilize multiple registers to achieve register-level blocking, selected the most optimal block sizes for each level, implemented memory alignment, and experimented with various optimization pragmas and discovered the most effective one (i.e., loop unrolling), tried both row-major and column-major and decided to stick with the latter.

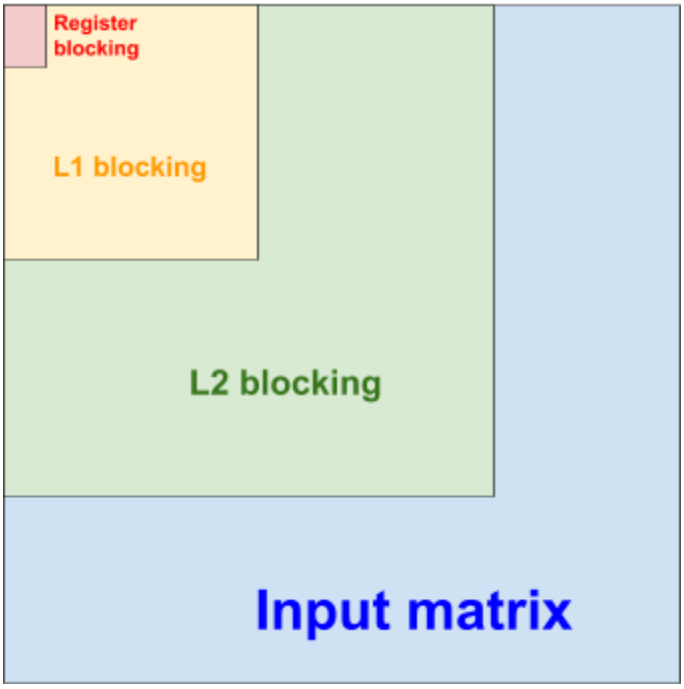
Optimization & analysis

AVX intrinsics

We used `_mm256_loadu_pd()`, `_mm256_set1_pd()`, `_mm256_fmadd_pd()`, and `_mm256_store_pd()` to batch four multiplications in one instruction. Using AVX intrinsics in the naive implementation increased the peak percentage from 4.2% to 9.6%. The performance improved because these SIMD instructions could batch operations on four doubles into a single instruction. However, we were not able to utilize AVX intrinsics' full potential until we implemented register-level blocking. The synergy between AVX intrinsics and register-level blocking released more significant performance gains.

Three-level blocking

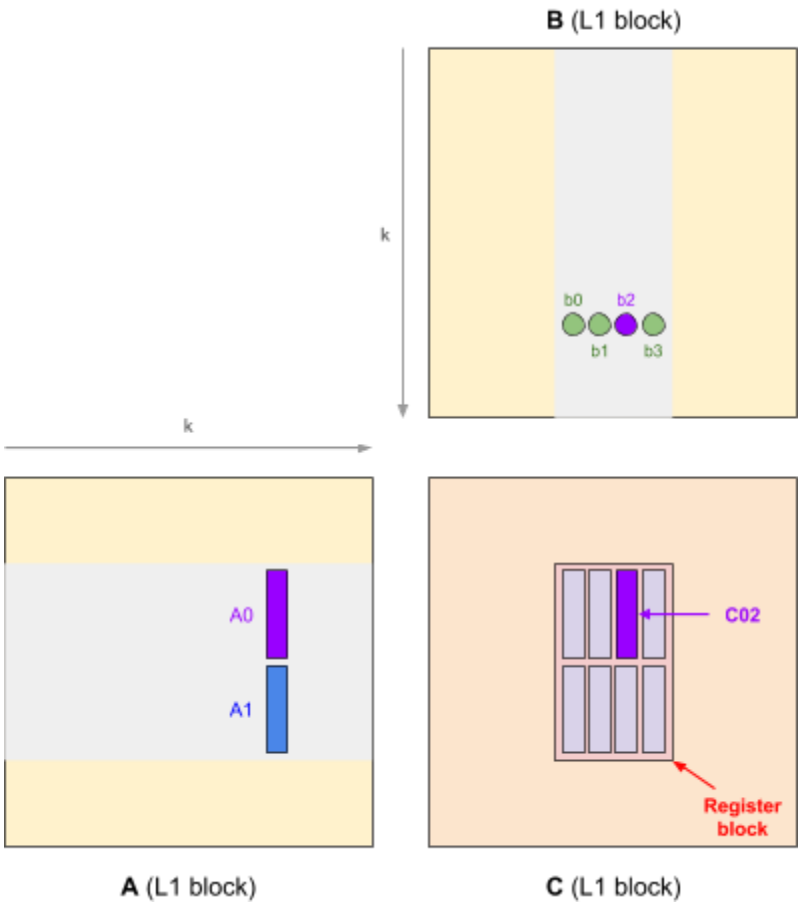
Blocking is one of the most important optimization techniques that we used. For matrix multiplication, blocking enables us to reuse nearby data multiple times, yielding better locality while reducing cache misses. The skeleton code provided already had one single level of blocking. We first tried extending it to two levels, and the improved performance eventually led us to implement three levels of blocking, as shown in the figure below.



The first level of blocking is already provided in the raw code. We call it "*L2 blocking*" because we use this level of blocking to "warm-up" the L2 cache. Since the L2 cache is relatively large, the block size at this level, `BLOCK_SIZE_L2`, is the largest. If a block is loaded into the L2 cache, we can retrieve it more efficiently (as opposed to loading it from the main memory). Once we have a block, we enter the next level of blocking via a function call.

The second level is *L1 blocking*, which is almost identical to the previous L1 blocking except for a smaller block size since the L1 cache is smaller than L2. We will discuss the specific choice of `BLOCK_SIZE_L2` and `BLOCK_SIZE_L1` in a while. L1 blocks are the basis for our final level of blocking.

The third level, and arguably the most crucial level, is *register blocking*, in which we use AVX intrinsics. The process of register blocking is illustrated below. We define each register block as an 8-by-4 matrix of doubles, or equivalently, a 2-by-4 matrix of vectors. We chose this dimension based on many factors, including the fact that `_m256d` are four-double vectors, the limitation that Haswell has only 24 registers, and `BLOCK_SIZE_L2` and `BLOCK_SIZE_L1`, which we shall discuss below in detail. To compute one register block in `C`, we iterate over A and B in the corresponding rows and columns, accumulating their product into `C`. The key point is: *In each iteration (e.g., `k = 7`), we explicitly compute all eight vectors in the register block at once, without using any loop*. For example, the figure below depicts an iteration. Note that we can update $C_{02} = C_{02} + A_0 \circ [b_2\ b_2\ b_2\ b_2]$. Similarly, we update all eight vectors in the register block of `C` simultaneously. Then, we move `k` forward into the next iteration, where we repeat this calculation.



In a way, we "unrolled" the loop for iterating over `C`, eliminating the concomitant overhead. Moreover, with the power of AVX intrinsics, we are able to process multiple values at the same time and perform the calculation efficiently using FMA. Thanks to three levels of blocking coupled with AVX intrinsics, we had a considerable performance gain of ~10%.

Padding

If we want to utilize blocking as described above, padding is necessary. This is because we cannot simply assume that the input matrix can always be perfectly partitioned into evenly sized blocks. For example, we cannot perform blocking directly on a 61-by-61 matrix using 8-by-8 blocks, since `61 % 8 == 5`. The remaining 5 rows/columns do not fill the block size of 8 thus have to be dealt with in the next level.

It is clear that the input matrices have to be padded to a multiple of a divisor, `D` (i.e., they should be of dimensions $cD \times cD$ for some constant c). Here, the crux of the problem is to choose an appropriate `D`. Notice whenever we have modulo rows/columns that cannot fill the current block, we deal with it in the next level using smaller blocks. Recall the smallest and final level of blocking is register blocks, which are 8-by-4 sub-matrices. Therefore, we set `D` to 8, so that any misfitting rows/columns can be dealt with register blocks.

Padding requires memory allocation and copying, which is a price to pay. Fortunately, the benefits of vectorization outweigh this initial cost, offering another boost of ~8% in our performance.

Memory alignment

When allocating memory for padding, we encountered **a strange bug**. Initially, we tried the most common `malloc()` function, which resulted in a computation error on Cori, but not on a local laptop with *the same compiler configuration and programming environment*. Moreover, on Cori, the computation succeeds for *some* input but not all. We posted on Piazza and went to an office hour, but still could not locate the bug. Later, with some additional trial-and-error, we discovered that the function `calloc()` works on both Cori and our local laptops. However, it is noticeably slower than `malloc()`, which we discovered is because `calloc()` initializes the allocated memory with 0, while `malloc()` does not have this initialization. This is where we finally identified the bug: We did not explicitly initialize the padded memory with 0; On macOS, with `malloc()`, the system automatically performs this initialization; By contrast, Linux does not perform such initialization.

At any rate, we had to pay the price of initialization, which caused a **performance dip**. However, we discovered a way to compensate for this cost -- memory alignment, supported by the function `_mm_malloc()`. Since we are using Intel AVX vectorization (specifically 256-bit access), it is recommended to adopt a 32-byte alignment. With aligned memory, our four-double vector operations will likely experience fewer cache misses. The performance gain is about 5%.

Block size tuning

Increasing block size: using block size 8 instead of block size 4 was able to increase performance from 9.6% -> 22%. This is due to spatial locality; by working in a block of 8, adjacent loads are faster since they will be stored into the cache on the first read.

Tuning block size involves a trade-off: If the block size is too small, we cannot fully exploit spatial locality; But if it is too large, the block might not fit into the cache. Therefore, we need multiple trial-and-errors to pick a sweet spot. We have already settled on a register block size, so there are two remaining block sizes to pick: L2 and L1. Our goal is to pick block sizes as large as possible but still fit into the cache.

We perform a rough calculation to estimate the optimal block sizes. Haswell has 64 KB of L1 cache and 256 KB of L2 cache per core, and each double takes 8 bytes. Thus, there can be 8,000 and 32,000 doubles in L1 and L2 cache, respectively. Since we are dealing with 3 matrices, each matrix can at most contain 2,666 and 10,666 doubles for L1 and L2. This translates into an upper bound for L1 and L2 block sizes of 51 and 103. To simplify block alignment, we decide to use powers of two as block sizes. Consequently, we set `BLOCK_SIZE_L1` to 32 and `BLOCK_SIZE_L2` to 64. Experiments on Cori confirm that our choice of block size indeed worked, providing a performance boost of ~5%.

Loop unrolling pragma

Beside manually unrolling the loop over `C` as illustrated above, the GCC loop unrolling pragma also turns out to be helpful. Unrolling reduces the loop overhead of incrementing, checking the loop condition, and jumping, which in turn improves performance. We first experimented with `unroll-loops` and `peel-loops`, and the

former produced a better percentage. Then, we identified several functions in our program that are suitable for loop unrolling (i.e., loops that are simple, small, and/or has a fixed size), where we added the attribute `__attribute__((optimize("unroll-loops")))`. These attempts resulted in a percentage increase of ~8%.

Loop reordering

Since the input matrix is in *column-major* format, we took extra steps to ensure that our loop iterations visit contiguous blocks as much as possible to achieve better locality. Specifically, the index of the element on the *i*-th row and *j*-th column is now `i + j * lda`. Therefore, the inner loop should iterate using `i`, and the outer loop should use `j`. In this way, we are able to visit cache lines in a contiguous manner, resulting in a better spatial locality. The performance gain coming from loop reordering is moderate, at around 3%.

Compiler optimization flags

We tried two common GCC optimizing flags: `O3` and `Ofast`. However, both failed to produce any significant changes. It is possible that the GCC optimizations were not significant enough for our simple function.

Inline functions

The functions we defined use the `inline` keyword, which might make the binary larger, but the code runs slightly faster. This is because we no longer need to push/pop parameters and addresses to/from the stack.

Masked instructions

As an alternative to deal with blocks that are not a multiple of 8, we tried to use masked loads and stores so that only a subset of the entries will be read for irregular block sizes. However, it turns out masked loads were not supported by in this assignment's environment and we ran into a compile error when trying to use it.

Performance on different machines

We ran our code on a MacBook with an Intel 2.8 GHz processor, whose `max_speed` is 44.8 according to our calculation. The benchmark showed an average percentage of the peak of 64.7281%, which is noticeably higher than the result on Cori (48% - 49%).

The MacBook on which we ran the code is using the Kaby Lake microarchitecture, a more recent successor to Haswell. We hypothesize that the enhanced performance above is due to the differences between Kaby Lake and Haswell. Unfortunately, we do not have enough time and resources to conduct a meticulously controlled experiment to validate our hypothesis. Nonetheless, the specification data of these two microarchitectures support our theory: While Haswell has a max clock speed of 4,400 MHz, Kaby Lake has a higher max speed at 4,500 MHz. In addition, Kaby Lake uses the 14 nm lithography process, an improvement compared with Haswell's 22 nm process. Therefore, it is not surprising that we obtained a better result locally.