CS 267 Homework 1 Part 2

Xingyou Song [*], Yao Yuan [†‡]

Due Feburary 9, 2018

1 Parallel Optimizations

Recall from part 1 that our implementation had the following:

1.0.1 Original Method

Our original code consists of the following functions:

- weird_transformation - This transforms the input matrix by stretching its columns by STRIDE times, while reducing its rows by STRIDE times (and padding the last row accordingly if there is uneven divisibility).
- compute - This performs AVX vectorization to update each 8x4 block of the C-matrix, given input as "smaller blocks", using the FMA intrinsic. It also performs the 4x4 and 2x2 cases as needed.
- divide_into_small_blocks - This cuts an input block into smaller blocks, with each smaller block passed into compute.
- divide_into_large_blocks - This cuts the entire matrix inputs into blocks to be passed into divide_into_large_blocks.
- vectorized_FMA - This takes in 8x4 matrix from C and performs the correct updates. The 2x2 and 4x4 cases are also written to handle edge cases.

[*]xsong@berkeley.edu - Xingyou wrote the report, presented techniques and strategies for optimization, and coded some parts

[†]yao_yuan@berkeley.edu - Yao coded a significant part of the code in C and tested using various compiler settings

[‡]Our Third Partner, Zhiwei Yao, dropped the course.

Algorithm 1 divide_into_large_blocks(lda, A,B,C)

$A_{weird} \leftarrow weird\_transformation(A)$
$LARGE\_M = 128$
$LARGE\_N = 256$
$LARGE\_K = 512$
for $i = 0, i < lda, i+ = LARGE\_M$ **do**
  for $j = 0, j < lda, j+ = LARGE\_N$ **do**
    for $k = 0, k < lda, k+ = LARGE\_K$ **do**
      divide_into_small_blocks(block(i,j,k))
    end for
  end for
end for

Algorithm 2 divide_into_small_blocks(M, N, K)

$A_{weird} \leftarrow weird\_transformation(A)$
$SMALL\_M = 8$
$SMALL\_N = 8$
$SMALL\_K = 512$
for $k = 0, k < K, k+ = SMALL\_K$ **do**
  for $j = 0, j < N, j+ = SMALL\_N$ **do**
    for $i = 0, i < M, i+ = SMALL\_M$ **do**
      compute(small_block(i, j, k))
    end for
  end for
end for

Algorithm 3 compute(M, N, K)

Vectorized_two_by_two (M, N, K) (handles corners)
Vectorized_four_by_four (M, N, K) (handles edges)
Vectorized_FMA(8×4 M, N, K) (handles main inner rectangle)

1.0.2 OpenMP Method

Now consider our code, after adding OpenMP, which was written in the do_block_large function. We divided each i-blocking for each thread, as shown below:

Algorithm 4 divide_into_small_blocks(M, N, K)

$A_{weird} \leftarrow weird\_transformation(A)$
$LARGE\_M = 16$
$LARGE\_N = 16$
$LARGE\_K = 1024$
PARALLEL NUM_THREADS(min(32, $lda/LARGE\_M$))
PRAGMA OMP FOR
for $i = 0, i < M, i+ = LARGE\_M$ **do**
  for $j = 0, j < N, j+ = LARGE\_N$ **do**
    for $k = 0, k < K, k+ = LARGE\_K$ **do**
      compute(small_block(i, j, k))
    end for
  end for
end for

Furthermore, we also parallelized the weird_transformation function, for faster memory accesses.

1.1 Parallel Method

We moved the pragma omp parallel in between for loops of code, attempting to find the best emplacement. Note that in the code, we set the parameters manually: For small block values:

- $SMALL\_M = 8$
- $SMALL\_N = 8$
- $SMALL\_K = 512$

and for large block values:

- $LARGE\_M = 16$
- $LARGE\_N = 16$
- $LARGE\_K = 1024$

Note that this generates very long blocks for performance. This was made long so that each thread does not have colliding calculations with any other block. Furthermore, the longer the block, the more consecutive the data is in memory for cache optimization. The theoretical reasoning for why inserting this into the beginning of do_large_block seemed to improve the most:

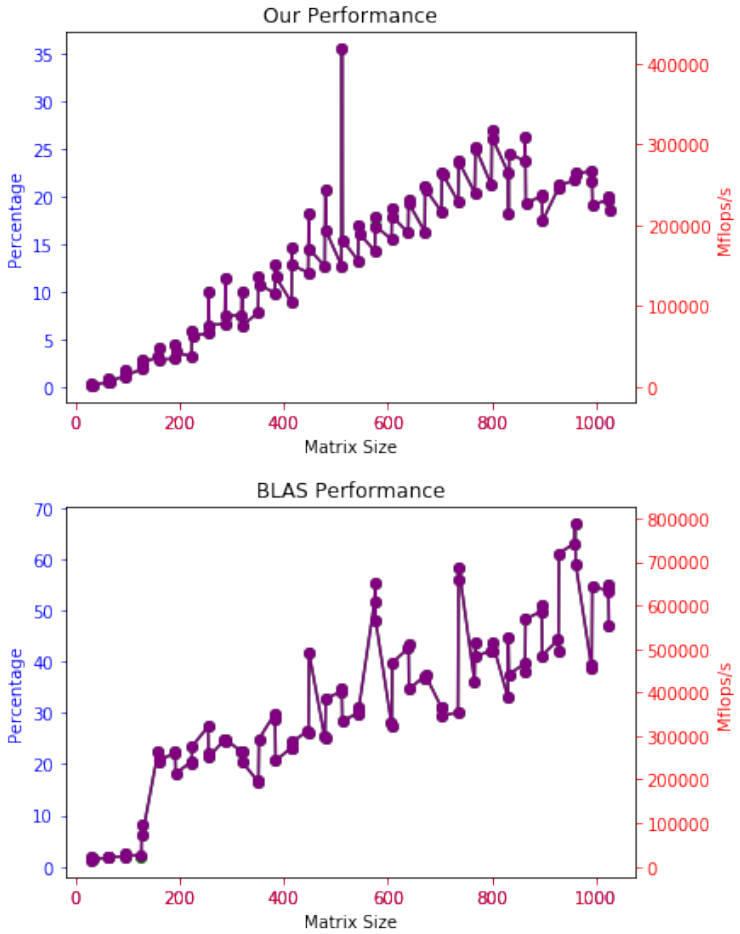
- We want to insert the pragma omp parallel for, somewhere deep within the hierarchy, to prevent any shared memory overheads.
- However, we also want to put the pragma somewhere up the depth, to prevent any thread spawning/barrier methods from generating high overhead.

Our parameters for block sizes also changed, in order to optimize this configuration, found empirically. BLAS average was 31%, and our blocked average was 14 %. Our worst performance is found for smaller matrices, but asymptotically stable performance for larger matrices.

1.2 Performance

1.2.1 Cori

Figure 1: Performance on Cori, BLAS and Blocked.

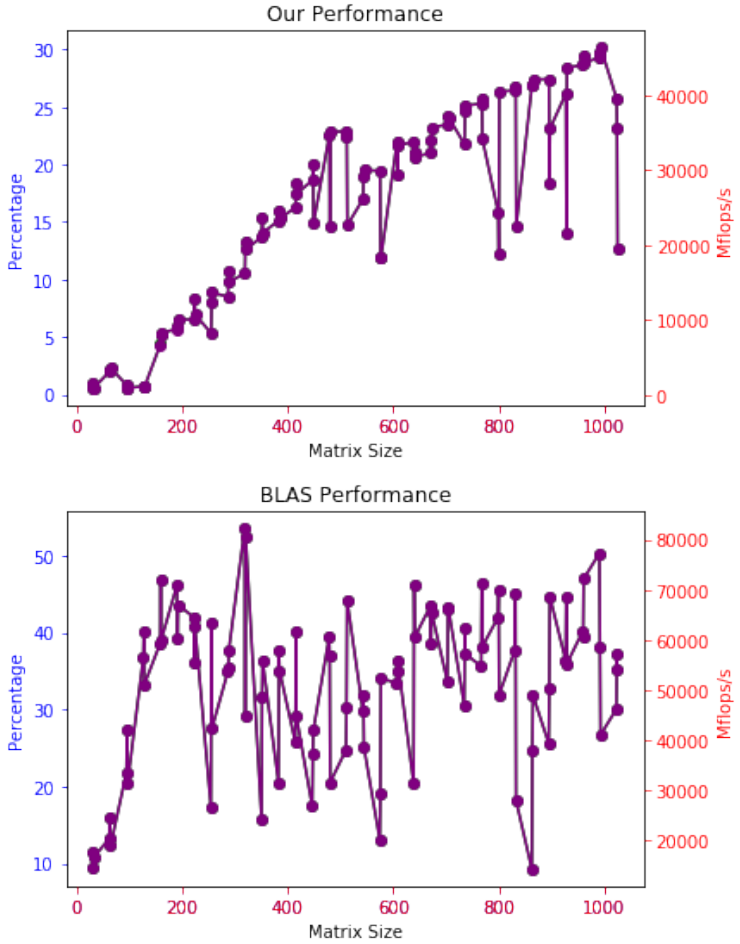


1.3 Performance On Local Computer

We performed the same code on a personal laptop, Lenovo Y510p with Intel i7 4700MQ 2.4 GHz, which allows AVX/AVX2 instructions. It also had the same cache specifications as Cori. The CPU allows for maximum frequency 3.4 GHz, but this was disabled. Thus this implies that the theoretical optimum is 4 cores * 2.4 GHz * 8 vector width * 2 flops for FMA = 153.6 GF/s, similar to the benchmark on Cori as well (for 4 cores). BLAS performed at 33 %, while our blocked performed at 16 %. This had the same behavior (i.e. outperforming Cori relatively) as part 1, possibly due to slight optimizations in Y510p's architecture.

Note that however, the performance was much more unstable on the Y510p than on Cori, for both BLAS and blocked. Perhaps this is due to the Y510p computer's operating system not fully committing the hardware for the simulation, due to other background processes running as well.

Figure 2: Performance on Lenovo Y510p, BLAS and Blocked.



2 Other Methods Considered

Our other methods primarily consisted of testing where to insert the pragma omp line. As mentioned in the above method, we found that having our original parameters caused too much overhead and shared cache problems, no matter where we inserted the pragma. Furthermore, we inserted time stamps to check in our code, the primary causes of slow-down. Ultimately we found that the weird_transformation was 20% of the run-time for serial, while computation was 80 %. These figures were the same after parallelization.