CS5220 - Final Project Report

Image Recovery using ALS Matrix Completion

Daniel Cardenas (dc862)

1 Introduction

In this project, I implement an algorithm for unknown or noisy recovering pixel values in images using an ALS matrix completion algorithm. I then attempt to parallelize the algorithm, and analyze how well it scales on a Google Cloud VM.

2 Algorithm

In this section, I'll describe the algorithm I used in short before moving on too my performance analysis.

2.1 Problem Statement

Suppose A is an $n_1 \times n_2$ image, which only has some pixels which we trust or know to be good. For convenience, let Ω be the subset of pixels in an image that we know to be "good" and have values for.

We'll split A into three different matrices, representing the red, green, and blue channels of the image. For each of each channels, we'd like to recover the pixels whose values aren't known to us.

Let A_c be a channel matrix. One way to recover the unknown values in A_c is to generate a rank-k approximation, such that $A_c \approx WZ^T$, where W has dimensions $n_1 \times k$ and Z has dimensions $n_2 \times k$. We'll say that W and Z are good approximations if they are solutions to the following optimization problem:

$$\min_{W,Z} \|A_c - WZ^T\|_{F,\Omega}^2 + \beta^2\|W\|_F^2 + \beta^2\|Z\|_F^2 \tag{1}$$

where for any matrix M $\|M\|_{F,\Omega}$ is simply the Frobenius norm of M calculated using only the values listed in Ω. [Has+15]

2.2 Alternating Least Squares

While (1) usually doesn't have easy to calculate solutions, we can iteratively compute W and Z by taking turns solving these easier optimization problems:

$$\min_{Z} \|A_c - WZ^T\|_{F,\Omega}^2 + \beta^2\|Z\|_F^2 \tag{2}$$

$$\min_W \|A_c - WZ^T\|_{F,\Omega}^2 + \beta^2 \|W\|_F^2 \tag{3}$$

Fortunately, (2) and (3) reduce to linear least squares corresponding to each row and column of A_c. Therefore, at the high level the algorithm is relatively simple:

1. Initialize W and Z
2. Loop over the following for a fixed number of iterations
   (a) Update Z to be the solution of (2)
   (b) Update W to be the solution of (3)

From my results running locally, it turns out that ~ 30 iterations is enough for convergence.

3 Optimizations and Performance

In this section, I'll talk about some of the optimizations I attempted to improve performance.

3.1 LAPACK and BLAS

As much as possible, I relied on LAPACK and BLAS routines for generating W and Z, especially for solving the linear least squares problems for the rows and columns of A_c. As the Google Cloud VM I ran the application on had Intel Cores, I also replaced the reference implementations of LAPACK and BLAS with the Intel Math Kernel Library, which appeared to result in a significant speedup during development.

Despite this speedup, the bottleneck of the algorithm still seems to be the linear least squares problems. I believe that I could significantly reduce the number of LLS problems by batching them together, but at the time of this report I had not been able to complete this optimization. So despite my consistent usage of LAPACK/BLAS routines, there are still some algorithm specific optimizations in this area that can be done.

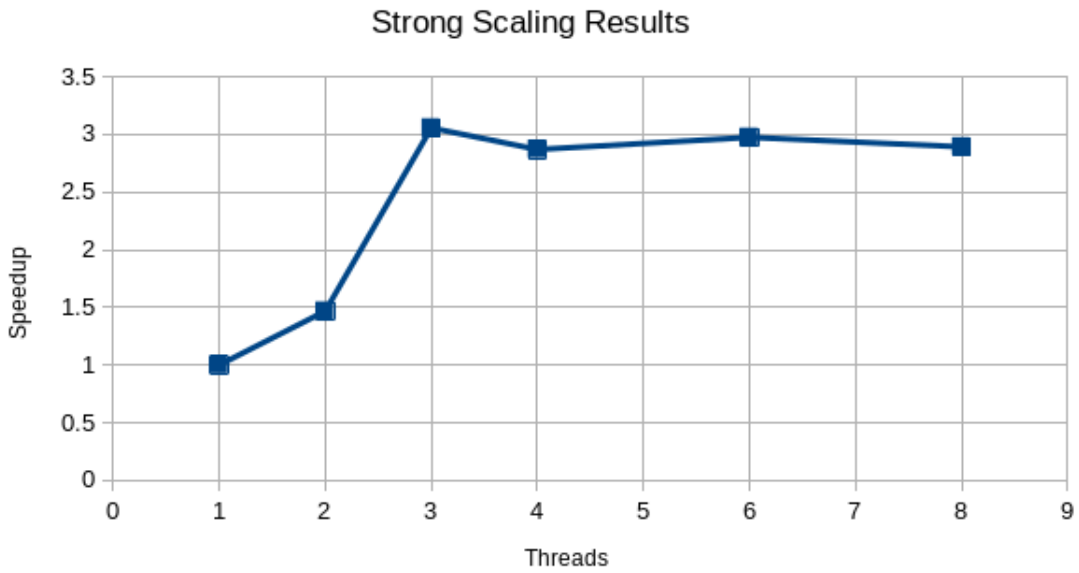
3.2 Parallelization with OpenMP

As mentioned in the previous section, I actually split the original image into three different matrices and run the ALS algorithm on each matrix separately. So an obvious optimization that I tried right away was to run the algorithm in parallel for each matrix. This simple optimization was actually the most effective optimization I implemented (after using a more efficient LAPACK/BLAS implementation), which I'll talk more about in the scaling studies.

Another optimization I tried that seemed to not be as effective was solving the LLS problems within each iteration of the ALS algorithm in parallel. To talk about why, I'll turn to the scaling studies I ran.

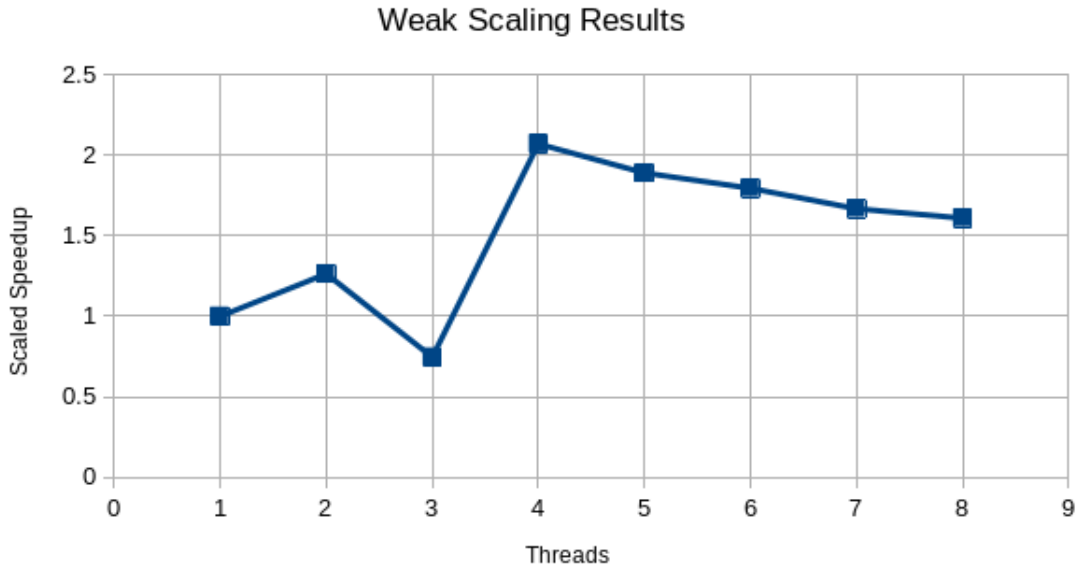
Strong Scaling To analyze how much our optimizations leads to a speedup, I ran strong scaling experiments on the c2-standard-8 instance in Google Cloud, which has 8 cores to work with. For the strong scaling experiment, I used an 892×698 image ($n_1 = 892$ and $n_2 =$

698) and used the parameters $k = 223$, $\beta = 10$, and fixed the number of iterations to 30.



As demonstrated in the graph above, we're able to achieve a maximum of a 3x speedup right at 3 threads, with performance plateauing immediately after. I believe this directly reflects that the biggest gains in parallelization comes from running the ALS algorithm on the three channel matrices simultaneously. I believe that to improve the strong scaling results past 3 threads, I'd need to focus more on restructuring how I pass the LLS problems to the LAPACK solver. It's possible that the LAPACK routines are only using a single thread, so I'd need to analyze and investigate this portion further.

After running the strong scaling experiments, I also ran weak scaling experiments using the same cloud instance, image, and parameters as before. To measure the scaled speedup, we also continuously increase k with the number of threads.



Similarly to the strong scaling experiments, we see a significant speedup after three threads

- although the experiment using exactly three threads seemed to greatly underperform. Unfortunately, my current implementation of the algorithm seems to have poor weak scaling, as the scaled speedup seems to reach only a maximum of 2x at 4 threads and only decreases after that. Again, this seems to demonstrate that the loop of the ALS algorithm does not currently parallelize well.

Overall, while I'm happy with the results of the algorithm on small images it's clear there's more work to be done before it should be used on a large system with a large number of images that need to be processed.

References

[Has+15] Trevor Hastie et al. "Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares". In: *J. Mach. Learn. Res.* 16.1 (Jan. 2015), pp. 3367–3402. ISSN: 1532-4435.