Akash Palrecha, 2017B4PS0559P

Instructor: Dr. Rajesh Kumar

MATH F266

8/07/2020

Title: Analyzing the trade-off in speed and accuracy when performing PCA in FP16 vs. FP32 precision.

ACKNOWLEDGEMENTS

This project would not have been possible without the help and guidance of many people. First and foremost, I want to thank my instructor, Dr. Rajesh Kumar, for guiding me and letting me pursue a topic of my own choice, which is also something that I am very deeply interested in. These times have been difficult, and I am very grateful to my parents and my brother for helping and making arrangements for me to productively work towards, among a lot many other things, this project over the last few months. I also want to especially thank Rachel Thomas (co-founder, Fast.ai) for putting together such a great and rigorous computational linear algebra course for free on the internet, which gave me the confidence to go forward with this project. Lastly, I would like to mention Danish Mohammed and Harsh Shah for their assistance and feedback on the project.

ABSTRACT

Matrix computations are at the heart of the AI revolution today. In industry, PCA is a standard pre-processing step for data matrices before applying advanced ML algorithms. This project proposes the use of a much faster, but less accurate FP16 precision format to carry out such computations in fault-tolerant applications where a 2-5x speedup is potentially more valuable than a small loss in accuracy.

INTRODUCTION

According to a study by OpenAI, the amount of compute needed for AI research doubles every 3.5 months since 2012. In industry, companies use terabytes of data every day to train ML models. The data pipeline for such models uses PCA as a primary step to reduce the dimensions of the data in matrix form before feeding it to the ML models. Data with millions or even billions of rows and thousands of columns is frequent in industry ML applications, and it is very typical to use PCA to remove redundant features/columns. Server costs per hour of usage (AWS, GCP, Azure) are at an all-time high, and it is crucial that we reduce the number of hours used for heavy compute tasks. (server costs just for this project are close to INR 7,000)

The advantage of doing computations in FP16 is that it offers us anywhere between 2-20x speedups in specific settings with a minimal loss in accuracy. NVIDIA originally popularized the idea in their paper titled “Mixed Precision Training” from ICLR 2018. The primary goal of FP16 training was to speed up massive deep learning models. Moreover, in their paper, the authors showed that models trained with mixed precision we as accurate as FP32/64 models for all real-world uses while taking up half the memory and a third of the time for computation. This was a huge leap forward as it allowed AI researchers to experiment with even bigger models in less than half the time than it used to take previously. Even though FP16 precision cannot represent

numbers bigger than 65536 (2^16), the core insight was that for most AI/ML use cases, the numbers involved are generally in the (-5, 5) range. Also, what matters more is the direction of the vectors as opposed to their absolute element values.

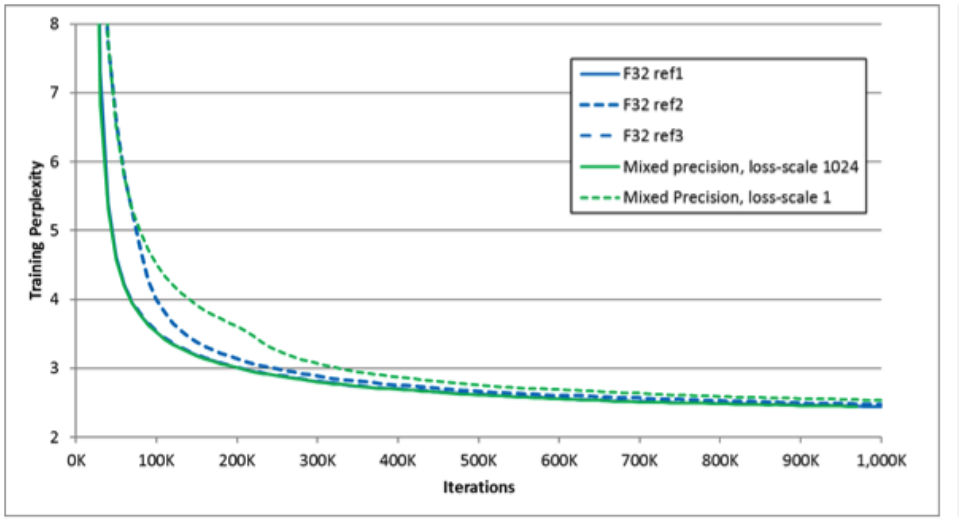

Fig. 1: Comparing deep learning models with different precision during training

Since Principal Component Analysis (PCA) involves calculating the covariance matrix, getting eigenvalues, and finally, a matrix multiplication, it only makes sense to make use of FP16 to accelerate this algorithm on GPUs (GPUs are highly optimized for matrix computations). As of now, all the off-the-shelf algorithms available in different math libraries implement PCA in FP32/64 on CPU hardware, which is very slow. Transferring the computation to GPUs allows us a 50-100x speedup even with FP32/64 precision, and then a further performance multiplier of 2-4x with FP16 precision.

METHOD

First, we will look at how PCA is implemented on the computer, followed by issues with FP16 computations and some possible workarounds.

Reducing a matrix M from N to K dimensions using PCA involves three broad steps:

1. Calculating the covariance matrix C of M
2. Getting the top K eigenvectors (sorted by eigenvalues) of C
3. Transforming M into the space defined by the above eigenvectors by a simple matrix multiplication.

Step 1: Calculating the covariance matrix C of M

1. Let M be our data matrix of size $L \times N$
2. $X = M - M^{\circ}$ (where M° is a column vector of row-wise means of M)
3. $C = \frac{(X^T \times X)}{L - 1}$ (we divide by $L - 1$ to get an unbiased estimate)

   C is our required symmetric covariance matrix of M of size $N \times N$

Step 2: Getting the top K eigenvectors (sorted by eigenvalues) of C

1. First, we obtain eigenvalues by solving the below equation to obtain λ:
   $(C - \lambda I) = 0$, where λ represents the column vector of eigenvalues.
2. Solve this equation to obtain eigenvectors associated with the top K eigenvalues by magnitude: $(C - \lambda_i)V_i = 0 \; \forall \; i \leq K$
3. Form a matrix V of size $N \times K$ where V_i is the eigenvector corresponding to the i_{th} largest eigenvalue.

Step 3: Reducing the matrix M

1. This is the most straightforward step. We simply perform: $M = M \times V$

M is now a reduced matrix of size $L \times K$ which was previously $L \times N$

Now we talk about the issues with FP16 precision. As the graph shows, the minimum distance between any two representable numbers in FP16 keeps growing very steadily and can reach 32 in the range from 2^{15} to 2^{16}. The closer we are to 0, the more accurate our calculations will be. All numbers above 65519 and below 2^{-24} are rounded off to infinity and zero, respectively, in FP16.

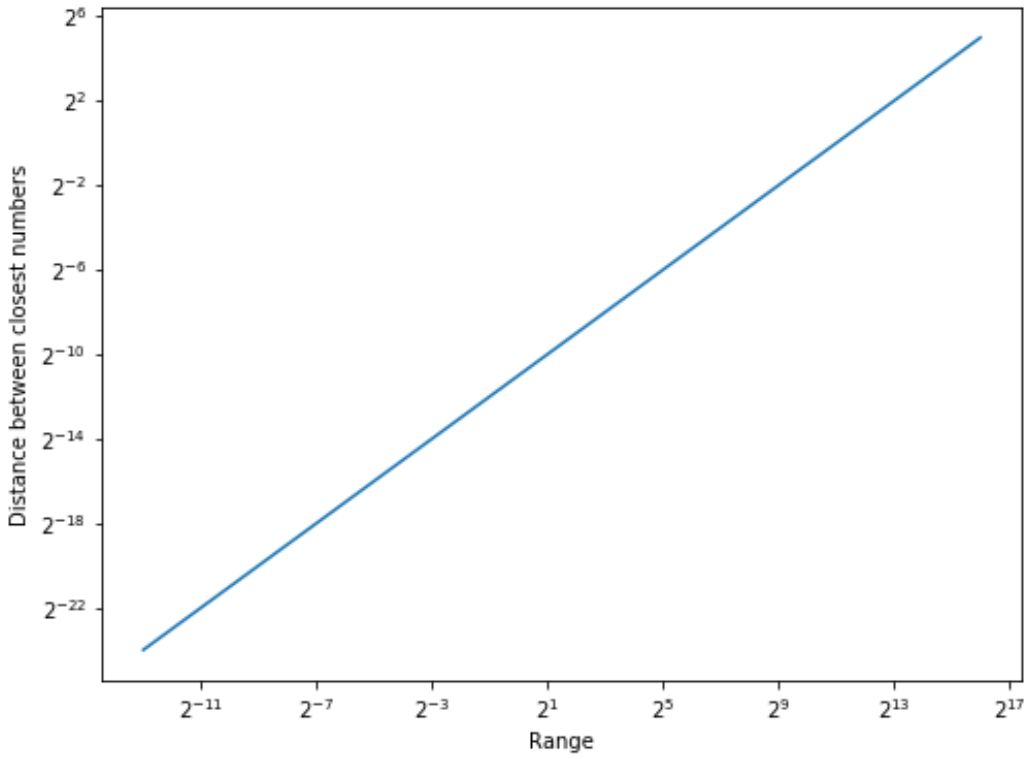

Fig. 2: Numerical range vs. precision for FP16 numbers

To work around these problems, we scale data with a high variance to the 0-1 range before performing computations. If needed, we also shift the numbers by a few bits by multiplying them with appropriate powers of 2 to keep them from rounding to zero.

EXPERIMENTS

We use the Python programming language as the primary driver for all our experiments. For numerical computation, we use the PyTorch package in python for direct support of GPU optimized matrix computations and mixed precision. Both the FP32 and FP16 versions of PCA for GPUs were not available directly and have been written from scratch using PyTorch's GPU and FP16 support. Extensive experiments were conducted with over 1600 permutations of different gaussian data distributions, each repeated over 50 times to get accurate readings. For each experiment, relative and absolute errors were recorded along with the time of execution in seconds. The errors were recorded by treating FP32 computations as fully accurate. The total time needed for one run of the whole pipeline was about 8-9 hours. For the hardware, we used an EC2 instance on Amazon Web Services with 64 GB RAM, 8 intel i7 Skylake CPUs, and an NVIDIA V100 16 GB Server GPU optimized for FP16 computations. This setup costs close to $3/hour. The data used for the experiments was generated from all possible combinations of the following parameters for a gaussian distribution with a mean of 0:

Number of rows: 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000, 1000000, 2000000, 5000000, 10000000

Number of columns: 10, 50, 100, 200, 400

Variance: 1, 4, 16, 64, 128, 512, 2048, 8192, 32768, 65519

Variants: scaled, non-scaled data

Precision: FP16, FP32

Repeats: 50 per combination

Total experiments: 16 x 5 x 10 x 2 x 2 x 50 = 80,000

The specific variance values were selected to cover particular ranges of FP16 precision. The numbers of rows were selected to accurately capture the computations at as many orders of magnitude as possible within the memory constraints of the available hardware. The same argument goes for the number of columns.

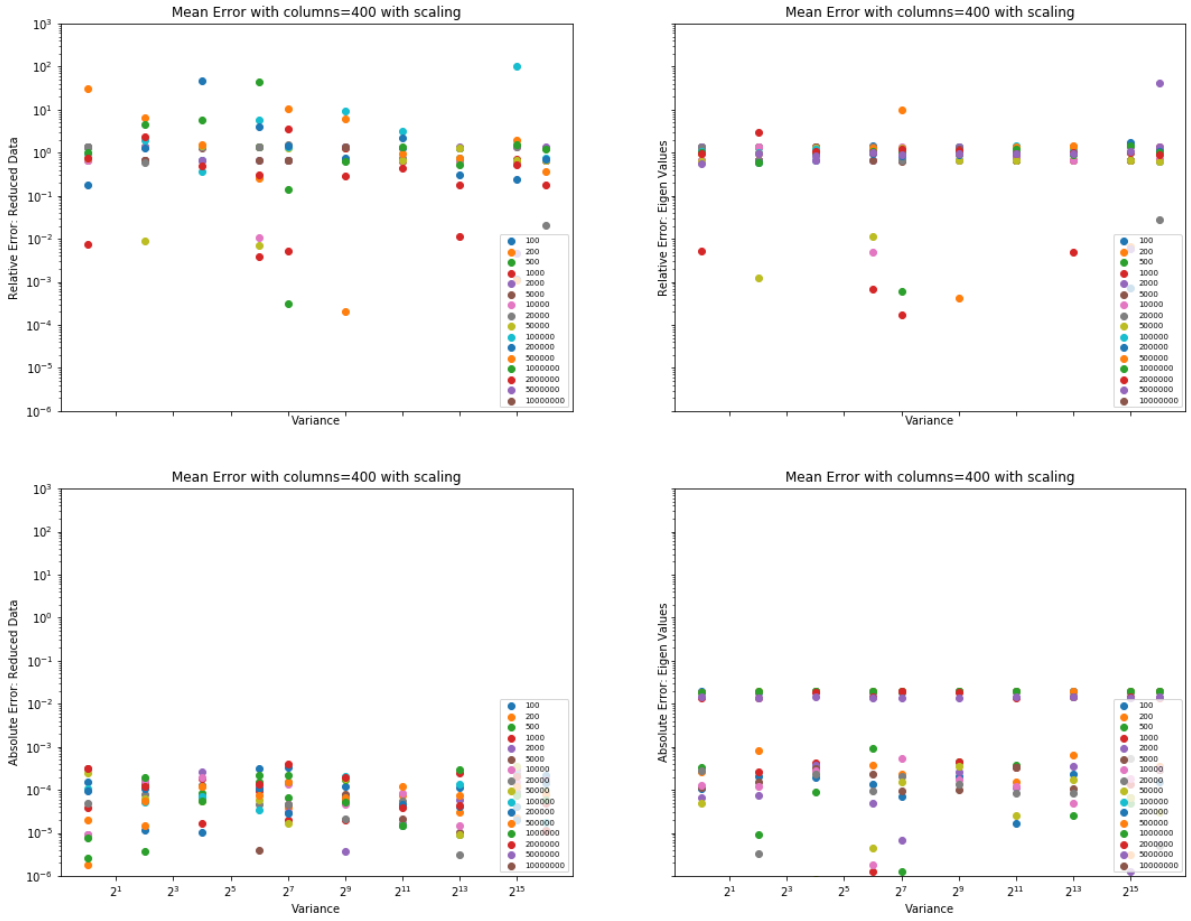

Fig. 3: Mean errors for different data distributions by variance and number of rows. The data was scaled before performing PCA.

As the graphs above show for scaled data, a majority of the relative errors lie within the 1~1.4% range, with some outliers. The absolute errors, again, are pretty low and mostly lie below ~0.0001. Thus, both the eigenvalues and the reduced data are accurately constructed with FP16 precision when the data is scaled to the [0,1] range.

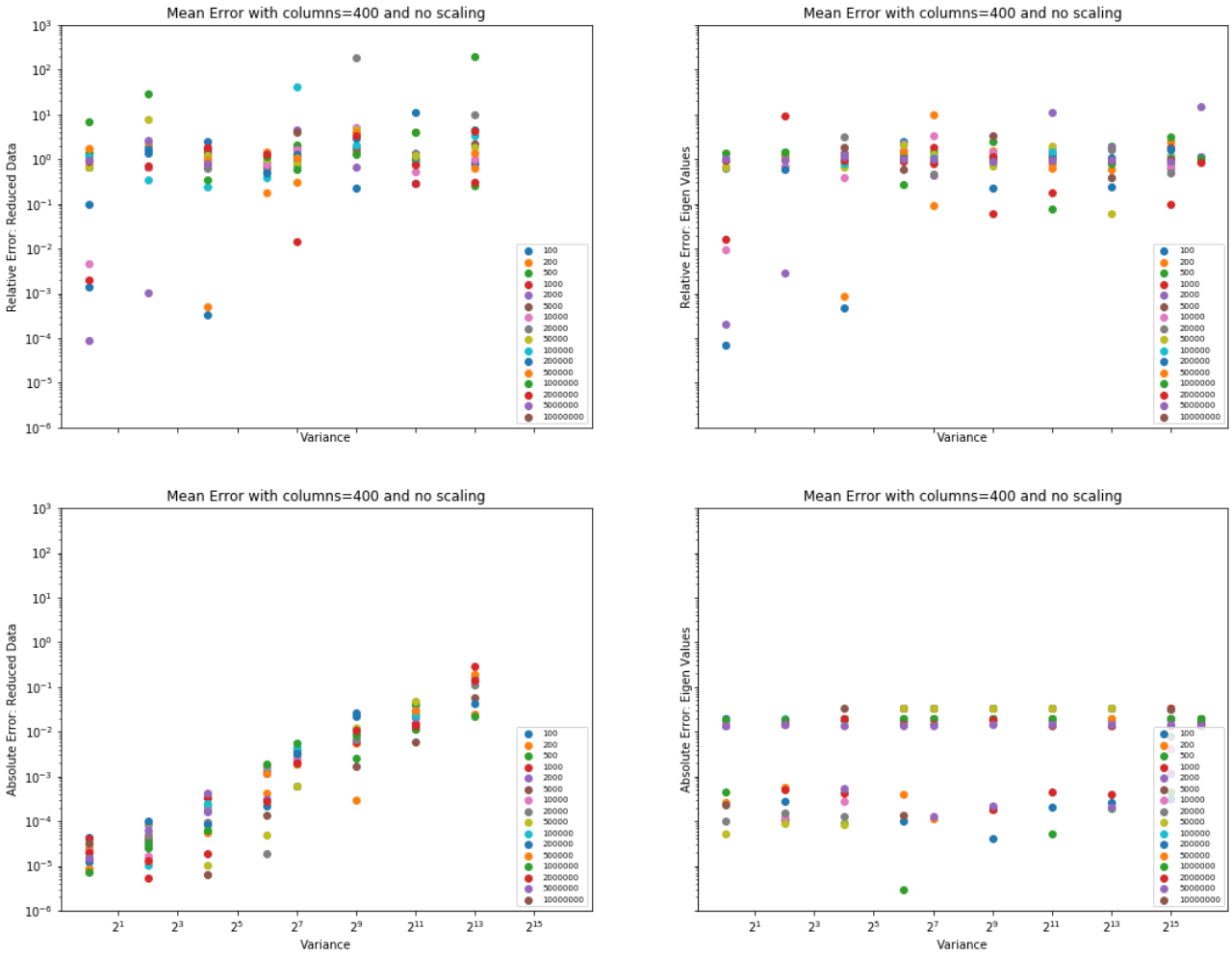

Fig. 4: Mean errors for different data distributions by variance and number of rows. The data was not scaled before performing PCA

We see a similar pattern of errors for non-scaled data as for data that is scaled. The exception is the upward slope that we observe in the absolute errors for the reduced data that keeps increasing as we move up the variance line. The reason for that is simply the fact that since the relative errors are still similar, the absolute errors will increase to account for the bigger scale of the data. We realize that the graphs above are not the best representations of the errors in the data, and hence, we provide a heatmap of errors in the next page to more visibly reflect the accuracy of the method.

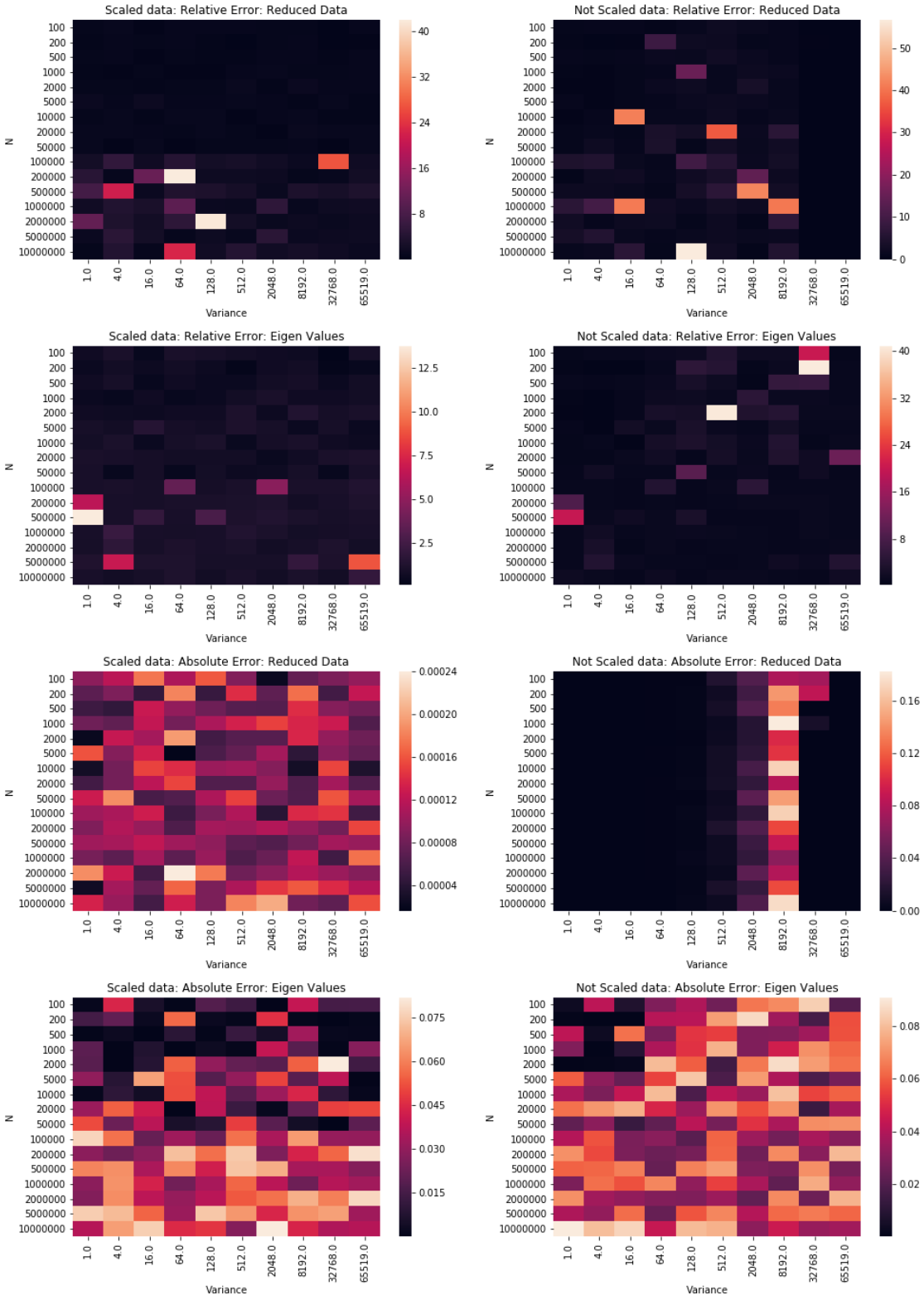

Fig. 5: Heatmap showing FP16 PCA errors for both scaled and non-scaled versions of the data.

The relative errors for both the reduced data and eigenvalues for the non-scaled variant are much higher than those of scaled data. They also seem to contain outliers more generally across the spectrum, as opposed to just at the bottom left as in the scaled data. On reason for high errors in the region with low variance and a higher number of rows is the fact that smaller numbers, when divided with N > 1000,000, tend to round down to zero in an FP16 setting.

We finally consider the speed gains with the new FP16 setting on GPU hardware. As per experiments, most of the gains are only consistently visible once we have more than 200 columns in our matrix and more than 100,000 rows. Theoretically, the gains should be visible in with much smaller matrices (since FP16 requires us to manipulate half the number of bits as opposed to FP32). In practical settings, though, the overheads associated with moving data from RAM to the GPU and converting it to FP16 outweigh the gains in computation speed for smaller matrices. However, once we cross a threshold, we immediately start seeing consistent gains of 2-3x in speed.

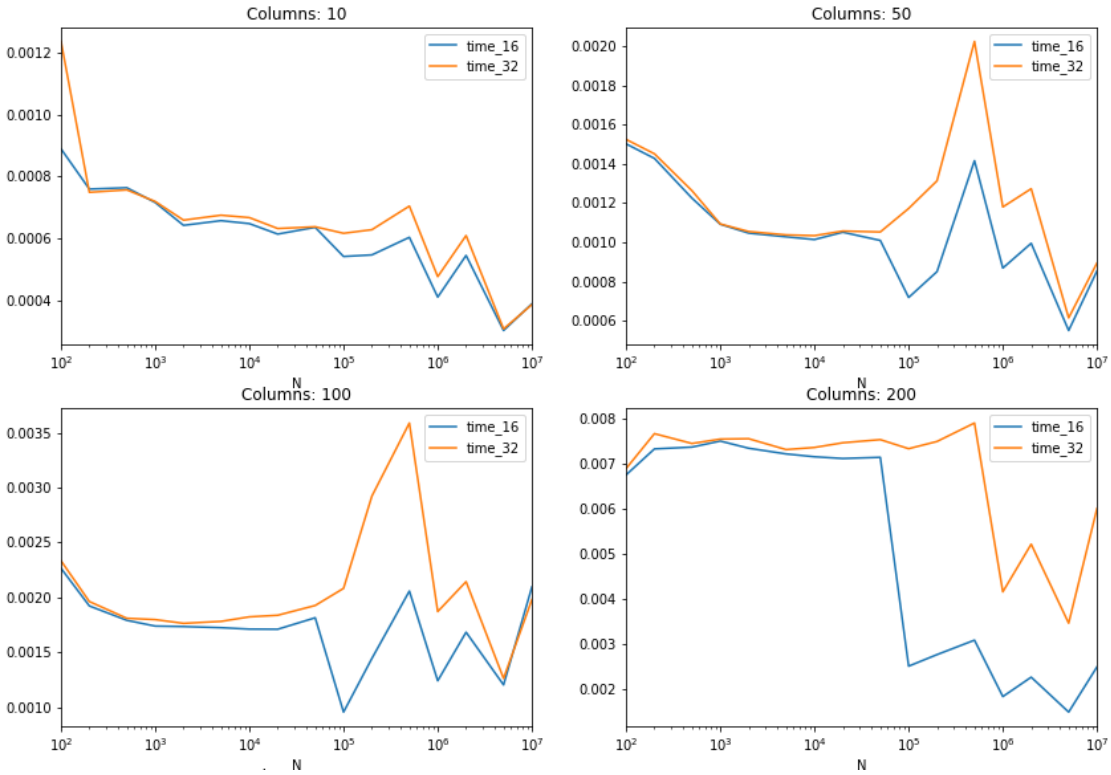

Fig. 6: Number of rows vs. Computation time for different numbers of rows

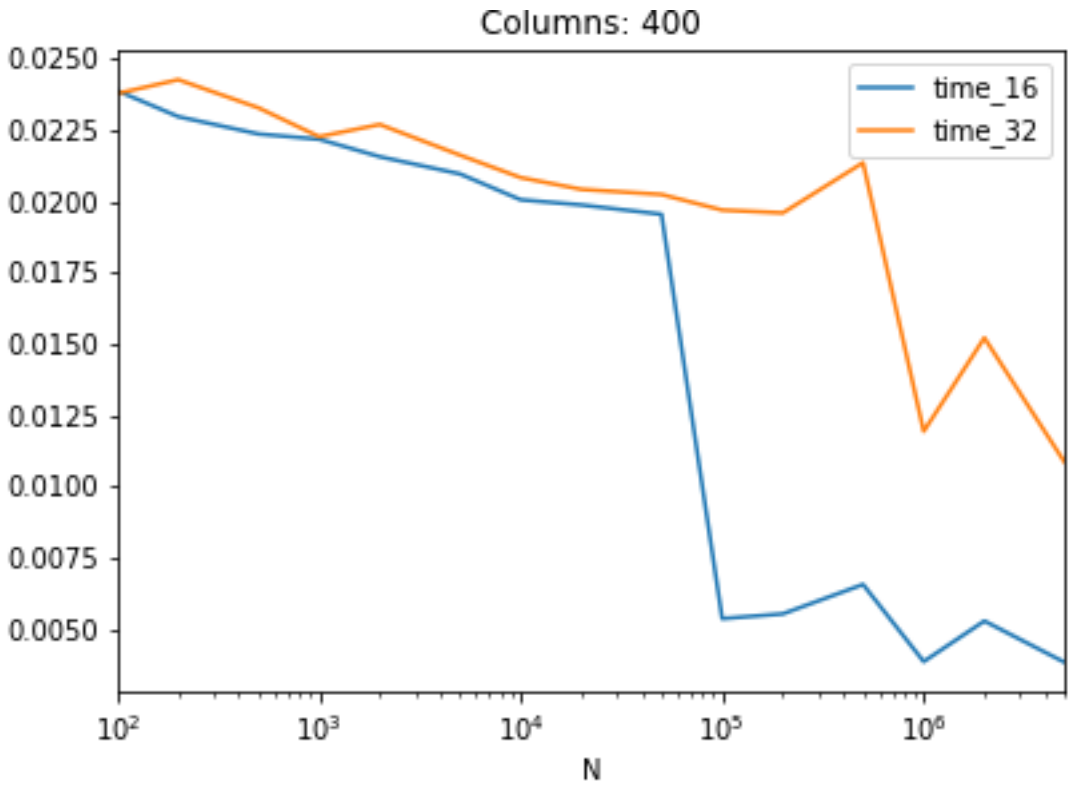

Fig. 7: Number of rows vs. Computation time for 400 rows

We notice that although there are slight gains in speed with 100 columns or less, it is only after 200 columns that we start seeing consistent and manifold gains in computation speed. What is consistent across all observations is that we need to cross 100,000 rows to be able to see any gains at all, no matter the number of columns. Another counter-intuitive effect is that the computation time is *decreasing* as the number of rows increase (and hence, the matrix size increases). This is a result of massive parallelization effects that kick in when GPUs process huge matrices. This effect only occurs on GPUs up to a certain limit, and never occurs on CPUs. Overall, we can expect a consistent minimum of 2x multiplier for performance when we have matrices with more than 200 rows and 100,000 columns.

Remarks and other observations

We note that as N grows larger ($> 10^7$), the division step in covariance calculation can cause some values to round to zero. Also, the matrix multiplications, if not done with scaled data, can cause some numbers to grow beyond 65519 and hence round to infinity. To deal with such issues, we clip all values of the covariance matrix between 0 and 65519. We can also multiply the matrices by small powers of 2 to move smaller numbers away from 0, as suggested by NVIDIA in their paper. According to our experiments, as long as the data lies in the 0-1 range, it should be possible to safely perform PCA with minimal error rates, even with 10 million rows of data.

CONCLUSIONS

As hypothesized, performing PCA in FP16 precision gives us not merely incremental, but manifold gains in computation time. It is to be noted that this method is only suitable for when we have huge matrices with more than 100,000 rows. Also, such advantages are only currently supported by some newer NVIDIA GPUs and Google TPU hardware. The errors that low precision inherently introduces are generally < 1% and can work for a lot of real-world problems that are sufficiently fault-tolerant. As FP16 numbers take up half as much memory as FP32 numbers, they can allow us to process datasets that are double the size of what was previously possible in a fraction of the time.

Bibliography

Micikevicius, Paulius et al. “Mixed Precision Training.” *ArXiv* abs/1710.03740 (2018): n. pag.