Non-Negative Matrix Factorization with Multiplicative Updates on GPU

E4750_2021Fall_SSAA_sls2305_ala2197.report
Alex Angus (ala2197), Skyler Szot (sls2305)
Columbia University

Abstract

Non-Negative matrix factorization (NMF) is an unsupervised machine learning technique often used in recommendation systems. A typical input is a large matrix of users and items, where rows represent user ratings for each item. NMF attempts to factor this large input matrix into two smaller matrices which can be used to recommend items to a user. In this project we investigate two multiplicative update rules for solving NMF as an optimization problem: Euclidean Distance and Kullback-Leibler Divergence [1]. We first implement a serial Python approach of both algorithms in Python, and verify that they monotonically converge to a local minima. Then we implement a parallel approach in pyCUDA, and achieve identical results with a significant speed reduction for large inputs. Ultimately, we achieve a 2x and 6x speedup of parallel Euclidean Distance and KL Divergence NMF, respectively, over the optimized Scikit-Learn serial implementations of NMF on an NVIDIA Tesla T4 GPU.

Key Words*: Non-Negative Matrix Factorization (NMF), Parallel Computing, GPU, Machine Learning, CUDA*

0. Key Links

1. **NMF Project GitHub Repository**
2. **NMF Project Slide Deck**
3. **Algorithms for NMF (Lee and Seung)**

1. Overview

1.1 Problem in a Nutshell

The goal of NMF is to factor a non-negative matrix $\boldsymbol{V}$ into two factor matrices $\boldsymbol{W}$ and $\boldsymbol{H}$. Here $\boldsymbol{V}$ is interpreted as a collection of column vectors where all elements are positive or zero. The matrix product of the factor matrices ultimately approximates the original matrix as follows:

$$V \approx WH$$

Each column in $\boldsymbol{V}$ is approximated by a linear combination of the columns of $\boldsymbol{W}$ weighted by the components of $\boldsymbol{H}$ (see Figure 1). Therefore, $\boldsymbol{W}$ can be regarded as containing a basis optimized for the linear approximation of $\boldsymbol{V}$. Since relatively few basis vectors are used to represent many data vectors, good approximation can only be achieved if the basis vectors represent a latent structure of the original data.

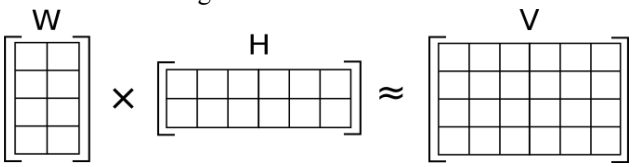


Figure 1. The matrix product of the factor matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ approximate the original non-negative matrix $\boldsymbol{V}$.

To find an approximate factorization $\boldsymbol{V} \approx \boldsymbol{WH}$, we first define cost functions to quantify the quality of the approximation [1]. Such a cost function can be constructed using some measure of distance between two non-negative matrices. We consider the Euclidean distance [2]

$$||A - B||^2 = \sum_{ij} (A_{ij} - B_{ij})^2$$

and the Kullback-Leibler Divergence (KL Divergence)

$$D(A||B) = \sum_{ij} \left(A_{ij} \log \frac{A_{ij}}{B_{ij}} - A_{ij} + B_{ij} \right)$$

to quantify the factorization and formulate it as an optimization problem [1]. The following are formal expressions of the NMF optimization problems for Euclidean Distance and KL Divergence, respectively [1].

Problem 1 *Minimize* $||\boldsymbol{V} - \boldsymbol{WH}||^2$ *with respect to* $\boldsymbol{W}$ *and* $\boldsymbol{H}$*, subject to the constraints* $\boldsymbol{W}, \boldsymbol{H} \geq 0$.

Problem 2 *Minimize* $D(\boldsymbol{V} \,||\, \boldsymbol{WH})$ *with respect to* $\boldsymbol{W}$ *and* $\boldsymbol{H}$*, subject to the constraints* $\boldsymbol{W}, \boldsymbol{H} \geq 0$.

While $||\boldsymbol{V} - \boldsymbol{WH}||^2$ and $D(\boldsymbol{V} \,||\, \boldsymbol{WH})$ are convex in only $\boldsymbol{W}$ or only $\boldsymbol{H}$, they are not convex in both $\boldsymbol{W}$ and $\boldsymbol{H}$ together. It is therefore impossible to find a global minimum analytically, and it is unlikely that numerical methods will find a global minimum, but there are many techniques that can be leveraged to find local minima. Gradient descent may come to mind as the simplest approach, but convergence can be very slow and sensitive to step size. This is inconvenient for large matrix sizes and may make it difficult to find an optimal solution.

To solve the optimization problems efficiently, we consider the multiplicative update rules presented in [1]. We replicate the multiplicative update rules here.

Theorem 1 *The Euclidean distance* $||\boldsymbol{V} - \boldsymbol{WH}||^2$ *is nonincreasing under the update rules*

$$H_{a\mu} \leftarrow H_{a\mu} \frac{(W^T V)_{a\mu}}{(W^T W H)_{a\mu}} \qquad W_{ia} \leftarrow W_{ia} \frac{(V H^T)_{ia}}{(W H H^T)_{ia}}$$

The Euclidean distance is invariant under these updates if and only if $\boldsymbol{W}$ *and* $\boldsymbol{H}$ *are at a stationary point of the distance.*

Theorem 2 *The divergence* $D(\boldsymbol{V} \,||\, \boldsymbol{WH})$ *is nonincreasing under the update rules*

$$H_{a\mu} \leftarrow H_{a\mu} \frac{\sum_i W_{ia} V_{i\mu}/(WH)_{i\mu}}{\sum_k W_{ka}} \qquad W_{ia} \leftarrow W_{ia} \frac{\sum_\mu H_{a\mu} V_{i\mu}/(WH)_{i\mu}}{\sum_\nu H_{a\nu}}$$

The divergence is invariant under these updates if and only if $\boldsymbol{W}$ *and* $\boldsymbol{H}$ *are at a stationary point of the divergence.*

Details on the proof of convergence are outlined in [1]. In brief, the multiplicative update rules emerge when the step size parameter in gradient descent is chosen such that the additive update rules become multiplicative update rules.

1.2 Prior Work

Non-negative matrix factorization is a popular unsupervised learning algorithm often used for recommendation systems. The use of NMF is well documented in prior works [3, 4, 5]. In terms of NMF implementations, there are many examples of existing code for both serial and parallel versions of the algorithms on the web [6]. We also find a CUDA implementation of the divergence cost function on GitHub, although this implementation has features outside the scope of our project [7]. Additionally, [8] investigates the performance of NMF multiplicative updates on various devices (Intel Xeon E5-2687W (10 physical cores), Nvidia GeForce GTX Titan X, Nvidia Tesla K80) and they find a significant speedup on GPU as compared to CPU. Also worth noting is the CPU optimized Scikit-Learn library for NMF [9] which we use to compare against our own implementations.

2. Description of Methods

The remainder of this section outlines our serial (CPU) implementation of the multiplicative updates for both the Euclidean Distance and KL Divergence cost functions. Then we discuss in detail the ***pyCUDA*** software design for multiplicative updates of the two cost functions on GPU.

2.1. Goal and Objectives

Our first goal is to implement the NMF algorithm in Python for both cost functions to show that the losses monotonically converge to local minima. To accomplish this we use a data set of 8447 *New York Times* articles [10] where each row of the data set corresponds to a single article and each column indicates the occurrence of an individual word. One element of the dataset matrix is the occurrence count of a single word in a given article. The vocabulary size (number of unique words) is 3012, meaning that our input $\boldsymbol{X}$ is a matrix of shape (8447, 3012). We approximate the input by factoring $\boldsymbol{X}$ into $\boldsymbol{W}$ and $\boldsymbol{H}$ using the multiplicative update rules with a rank dimension of $\boldsymbol{k} = \boldsymbol{25}$. Here, the rank dimension $\boldsymbol{k}$ is the intermediate dimension of the matrix multiplication between $\boldsymbol{W}$ and $\boldsymbol{H}$ with shapes (8447, $\boldsymbol{k}$) and ($\boldsymbol{k}$, 3012), respectively. We iterate for 100 multiplicative updates, after which the resultant columns of $\boldsymbol{W}$ correspond to 25 categories. Each element of $\boldsymbol{W}$ represents the relevance of a word to a category of article. We show the cost function as a function of iteration in addition to the resulting NMF generated categories.

Our second goal is to implement the same algorithms in parallel (GPU) with pyCUDA and to compare the results to the serial implementation and other parallel implementations [8]. Both the serial and the parallel implementations are initialized with the same random matrices for $\boldsymbol{W}$ and $\boldsymbol{H}$, and we verify that both implementations, after the same number of iterations, converge to the same local minimum. After demonstrating that the solutions are identical, we compare the runtimes of the serial and parallel implementations at various input sizes.

2.2. Euclidean Distance CPU Implementation

A visual representation of the multiplicative updates for the Euclidean Distance cost function is given in Figure 2, where matrix multiplication is performed within the box outlined in grey (a). In Figure 2, element-wise operations are performed between boxes (shown in Matlab syntax **A.*B** and **A./B**). The $\boldsymbol{H}$ update (b) is performed first, and the result is used in the $\boldsymbol{W}$ update (c). In Figure 3 we present a snippet of our Python code that implements the Euclidean Distance multiplicative updates illustrated in Figure 2.

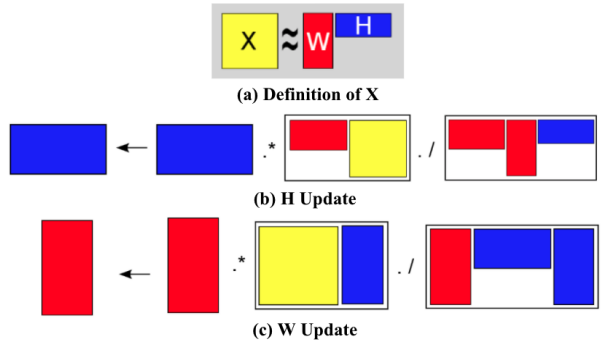


Figure 2. Visual representation of Euclidean Distance multiplicative updates for W and H.

```
if loss == 'euclidean':

    Wt = W.T # W transpose
    H = H * Wt.dot(X) / (Wt.dot(W).dot(H) + eps) # update H

    Ht = H.T # H transpose
    W = W * X.dot(Ht) / (W.dot(H).dot(Ht) + eps) # update W
```

Figure 3. Python code snippet for performing Euclidean Distance multiplicative updates of ***W*** and ***H***.

The full Python code is located in the GitHub repository in the file **nmf.py.** Note that in Python syntax **A.dot(B)** is matrix multiplication, whereas **A*B** is element-wise multiplication, and that we add a small value **eps** (on the order of 10^{-16}) for numerical stability.

Each time these updates are calculated, the values of ***W*** and ***H*** are guaranteed to better approximate the input ***X*** because of their monotonically decreasing nature. We can also see how these updates are well suited for a parallel implementation in CUDA. The updates consist of simple parallelizable operations like transpose, multiplication, etc. Figure 4 shows the Euclidean Distance loss as a function of iteration.

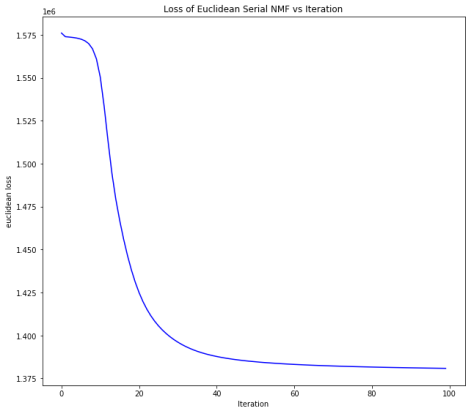


Figure 4. Euclidean Distance Loss as a function of iteration number. Note that the loss monotonically converges within 100 iterations.

2.3 Euclidean Distance GPU Implementation

For our parallel implementation of NMF, we wrote individual CUDA kernels for each elementary matrix operation. These kernels include matrix transpose, element-wise multiplication, matrix multiplication, element-wise division, and more. The CUDA kernel code is located in GitHub in the **kernels** folder.

The first step in the CUDA implementation is to transfer the input matrix ***X*** from the CPU to the GPU. We then allocate memory for the results of all intermediate calculations of the multiplicative updates. For example, we allocate an empty matrix ***Wt*** to store the result of ***W*** transpose (Step 1 of the Euclidean Distance ***H*** update outlined in Figure 5).

H Update

1. Wt = W.T
2. WtX = Wt * X
3. WtW = Wt * W
4. WtWH = WtW * H
5. WtWH + WtWH + eps
6. H .* WtX
7. H ./ WtWH

W Update

1. Ht = H.T
2. XHt = X * Ht
3. WH = W * H
4. WHHt = WH * Ht
5. WHHt = WHHt + eps
6. W = W .* XHt
7. W = W ./ WHHt

Figure 5. Euclidean Distance Multiplicative Update Steps

Between each step of the update equations, we perform a **cuda.Context.synchronize()** to ensure that all threads involved in the operation have completed before moving to the next step. A snippet of the pyCUDA code is located in Appendix 8.1.1. The full code and kernel functions can be found in GitHub in **nmf.py** and in **kernels**. Because all memory needed to perform the updates is allocated before the operations are performed, there is no need for multiple transfers from host to device during the iterative process. Host to device transfers are also not needed between iterations because ***W*** and ***H*** are updated independently on the GPU. Although we are making 14 individual kernel calls, there is very little speed compromise because all operations occur on the GPU.

2.4 KL Divergence CPU Implementation

Similar to Euclidean Distance, we provide a visual representation of the multiplicative updates for the Kullback-Leibler divergence cost function in Figure 6. In Figure 7 we present a snippet of our Python code that implements the KL Divergence multiplicative updates illustrated in Figure 6.

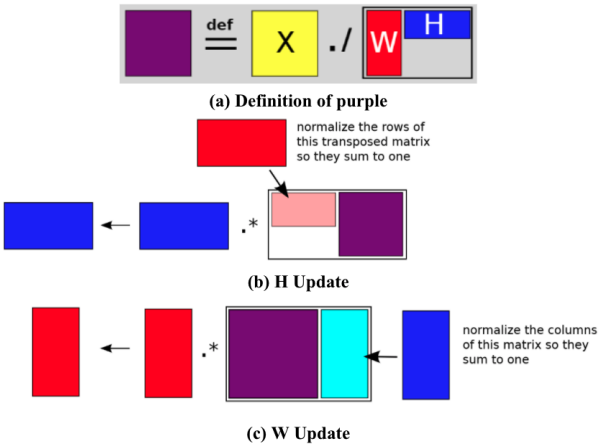


Figure 6. Visual representation of KL Divergence multiplicative updates for ***W*** and ***H***.

```
elif loss == 'divergence':

    P = X / (W.dot(H)+eps) #intermediate step (purple matrix in notes)
    Wt = W.T
    Wt = Wt / Wt.sum(axis=1).reshape(-1, 1) #normalize rows
    H = H * (Wt.dot(P))  #update H
    P = X / (W.dot(H)+eps)
    Ht = H.T
    Ht = Ht / Ht.sum(axis=0).reshape(1, -1) #normalize columns
    W = W * (P.dot(Ht))  #update W
```

Figure 7. Python code snippet for performing KL Divergence multiplicative updates of ***W*** and ***H***.

The Python implementation for KL Divergence multiplicative updates is slightly more complicated than the Euclidean Distance implementation, but again it is composed of elementary operations that can be performed in parallel with CUDA. A notable difference in the KL Divergence implementation is the sum operations. These sum operations are required to normalize the rows and columns of the matrices, and they introduce computational complexity into the KL Divergence implementations. Again, the full Python code is located in the GitHub repository in the file **nmf.py**. Similar to the Euclidean Distance implementation, we observe the KL Divergence loss to monotonically decrease, as is illustrated in Figure 8.

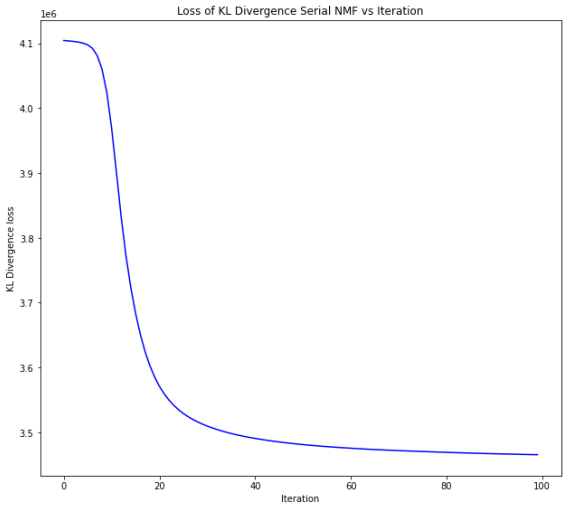


Figure 8. KL Divergence Loss as a function of iteration number. Note that the loss monotonically converges within 100 iterations.

2.5 KL Divergence GPU Implementation

In the pyCUDA implementation of KL Divergence NMF we reused many kernels from the parallel Euclidean implementation including matrix transpose, element-wise multiplication, matrix multiplication, element-wise division, and addition. We also needed to implement a method for row-wise and column-wise normalization. We achieved this by the use of a kernel for summing every row, and then a second kernel to divide every element by the sum of its row. We implemented two similar kernels for the row-wise normalization. Again, a pyCUDA code snippet can be found in the Appendix 8.1.2, and the full code and kernel functions are in GitHub in **nmf.py** and **kernels**. The order of steps performed is enumerated in Figure 9.

H Update

1. WH = W * H
2. WH = WH + err
3. P = X ./ WH
4. Wt = W.T
5. Wt_sum = sum of rows of Wt
6. Wt = Wt ./ Wt_sum
7. WtP = Wt * P
8. H = H .* WtP

W Update

1. WH = W * H
2. P = X ./ WH
3. Ht = H.T
4. Ht_sum = sum of cols of Ht
5. Ht = Ht ./ Ht_sum
6. PHt = P * Ht
7. W = W .* PHt

Figure 9. KL Divergence Multiplicative Update Steps

2.5 NYT Test Dataset

To assess the performance of our various NMF implementations, we apply them to a dataset composed of 8447 articles from the *New York Times* [10]. The dataset contains counts of 3012 unique words after a preprocessing step that groups similar words. Figure 10 illustrates the dissection of an article into relevant words by article topic [11].

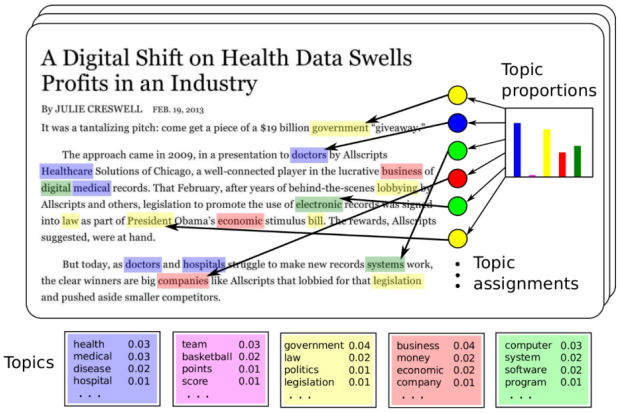


Figure 10. The NYT dataset contains unique word counts by article. We apply NMF to this dataset to "learn" the matrix $\boldsymbol{W}$, which represents the relevancy of words to unique article topics.

In the body of Figure 10 we see highlighted keywords that are enumerated in each article. The word count by article data (stored in GitHub as **data/nyt_data.txt**) is constructed into the matrix $\boldsymbol{X}$ of shape (8447, 3012) with the function **get_matrices()** in **src/data_utils.py**. We specify the rank dimension $\boldsymbol{k = 25}$ such that $\boldsymbol{W}$ and $\boldsymbol{H}$ have shapes (8447, 25) and (25, 3012), respectively. In **data/nyt_data.txt**, words are represented as integers from 1 to 3012. The mapping from word index to word is stored in the file **data/nyt_vocab.dat**. After NMF is performed on the data matrix $\boldsymbol{X}$ to produce $\boldsymbol{W}$ and $\boldsymbol{H}$, we use the function **get_topics()** in **src/data_utils.py** to remap the integers to words and to interpret $\boldsymbol{W}$ as a set of 25 learned topics with words ranked by relevance to the topic. We see these learned topics in the bottom portion of Figure 10 indicated by "Topics".

3. Results

3.1 NYT Test Dataset Results

As expected, the parallel implementations of NMF are significantly more time efficient than the serial implementations. On the NYT test set, Python Euclidean Distance executes in 45.12 s, Scikit-Learn Euclidean Distance executes in 8.283 s, and CUDA Euclidean Distance executes in 1.065 s for 100 iterations averaged over 10 trials. Similarly, Python KL Divergence executes in 128.4 s, Scikit-Learn KL Divergence executes in 66.80 s, and CUDA KL Divergence executes in 1.510 s for 100 iterations averaged over 10 trials. These results are visualized in Figure 11.

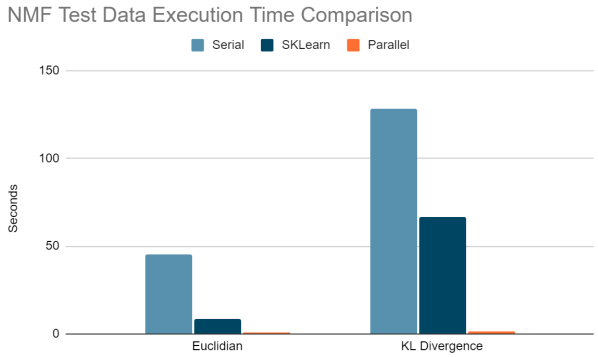


Figure 11. NMF execution time by implementation method for 100 iterations. The parallel NMF implementations for both Euclidean Distance and KL Divergence execute on the NYT test set in significantly less time than both serial implementations. Parallel implementations run on 1 NVIDIA Tesla T4 GPU.

As indicated in Figure 12, we observe the Euclidean Distance CUDA implementation to outperform the serial Python execution time by a factor of 44.5, while the serial Scikit-Learn only offers a factor of 5.6 times improvement over the Python version. The KL Divergence CUDA implementation sees an even greater relative performance, executing 86 times faster than the serial Python KL Divergence implementation, while the Scikit-Learn KL Divergence implementation executes only 1.9 times faster.

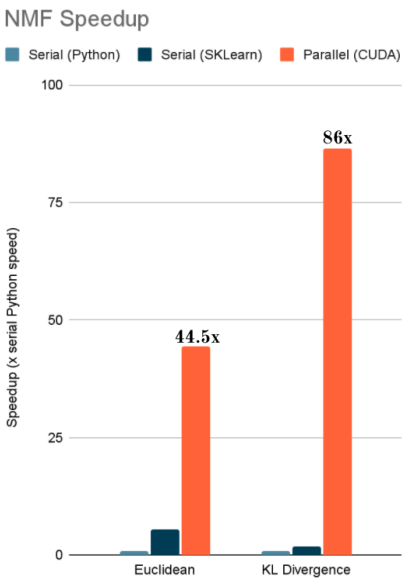


Figure 12. Speedup of NMF implementation relative to serial Python NMF. The parallel implementations of NMF execute significantly faster than the serial implementations on 1 x NVIDIA Tesla T4.

To accurately compare the performance of our serial and parallel implementations, we ensure that the $\boldsymbol{W}$ and $\boldsymbol{H}$ matrices are initialized identically in both cases by specifying the random seed in **get_matrices()**. The resulting "learned" matrix $\boldsymbol{W}$ should then be identical after 100 iterations of NMF for both the serial and the parallel results. We confirm that the results are identical for both by visualizing the output topics generated by the function **get_topics()**. Samples of these outputs are located in Appendix 8.2 and in the demonstration notebook.

3.2 Results on Various Input Sizes

The NYT test dataset results show that the parallel implementations of NMF significantly outperform the corresponding serial implementations for the input matrix size (8447, 3012) with $\boldsymbol{k}$ = 25. But, how do the results compare when we have various sized input matrices $\boldsymbol{X}$ and rank dimensions $\boldsymbol{k}$? To exhaustively evaluate the execution times of our NMF implementations, we perform a grid execution of different input parameters. See the file **execution_times.csv** which enumerates all parameter combinations. We vary input matrix size, rank dimension, and loss type for a total experiment number of 64 combinations of input parameters for each NMF implementation (serial, Scikit-Learn, parallel). These explorations are performed in the Jupyter Notebook **Measurement.ipynb**.

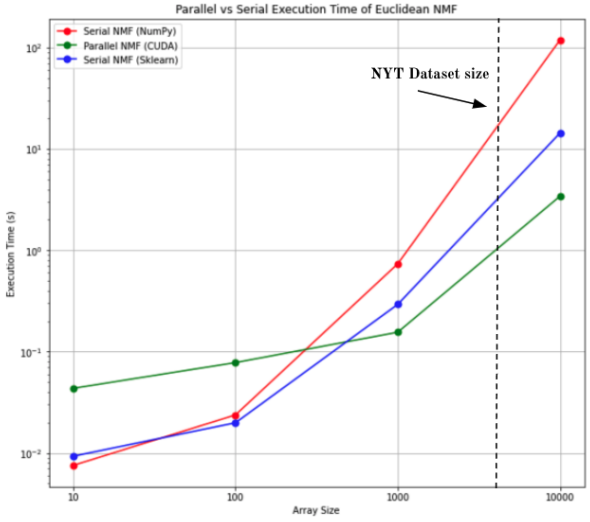


Figure 13. Execution time (including memory transfer) vs Input array size of Euclidean Distance NMF for serial and parallel implementations. 100 iterations are performed for each implementation and input array size. K = 25.

Here we select plots from two subsets of the total experiment space. In Figure 13 and 14 we measure execution time (including memory transfer steps) as a function of input matrix size when $\boldsymbol{k}$ = 25 for Euclidean Distance and KL Divergence implementations, respectively. We choose these subsets such that we can see the relationships between the execution times on the NYT test dataset ($\boldsymbol{k}$ = 25) and the arbitrarily sized input matrices.

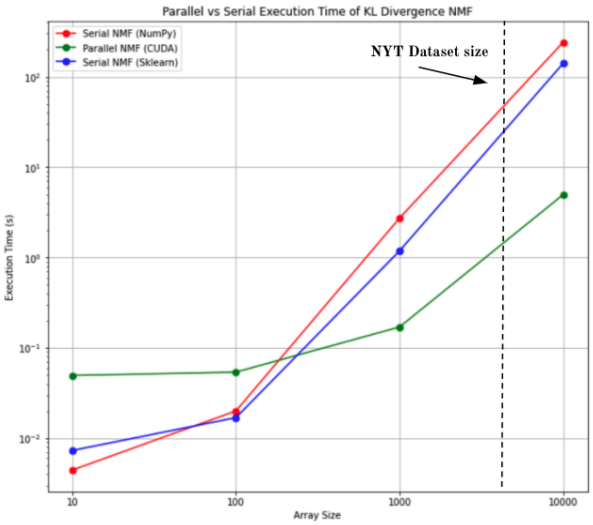


Figure 14. Execution time (including memory transfer) vs Input array size of KL Divergence NMF for serial and parallel implementations. 100 iterations are performed for each implementation and input array size. K = 25.

In Figures 13 and 14, note the vertical dotted lines around Array Size = 5000. These vertical lines indicate the size of the NYT test dataset input matrix size. We expect the execution time curves in Figures 13 and 14 to intersect the dotted lines at approximately the same times as indicated in Figure 11, and this is indeed what we observe. Also in Figures 13 and 14, we see that the serial execution times (blue, red) are shorter than the parallel (green) execution times for input matrix sizes less than approximately 300 x 300 elements. For larger input matrices, the speedup of the parallel implementations overcomes the memory transfer overhead necessary for the GPU implementations. We observe speedups on the order of 10^2 relative to the serial implementation execution times for large input matrix sizes (10^8 elements). It is likely that we would see even larger speedups for input matrices larger than 10^8 elements, but matrices larger than 10^8 elements exhaust the host memory of our current setup (4 x Intel Haswell CPU with 15 GB memory (total)), and thus additional engineering is required to facilitate larger input matrix experiments. The full series of measurements is recorded in **execution_times.csv** and corresponding plots are located in Appendix 8.3 as well as in the **figures** directory. Additionally, we directly compare corresponding

implementation execution speeds (Euclidean Distance vs. KL Divergence) over various input sizes and consistently find that Euclidean Distance NMF executes in less time than KL Divergence NMF for larger input sizes (> 10^6 input elements). Figures supporting these results are located in Appendix 8.3.5.

4. Demonstration

We provide a Jupyter Notebook **Demonstration.ipynb** to demonstrate the functionality and organization of our code. The notebook guides the user through this project in a self-explanatory way and presents elementary results. We recommend looking through it to get a quick, comprehensive overview of our work.

5. Conclusion and Further Work

Our results indicate that parallelization of NMF using CUDA can significantly reduce execution time and convergence speed when the number of input matrix elements is large. Additionally, our parallel pyCUDA implementation of both Euclidean Distance and KL Divergence NMF outperform the CPU optimized Scikit-Learn implementation of NMF in terms of execution time for large input matrices. These results are consistent with [8] in which NMF implementations on CUDA capable devices (Nvidia GeForce GTX TITAN X and Nvidia Tesla K80) significantly outperform corresponding CPU (Intel Xeon E5−2687W v3 @ 3.10GHz) implementations. The test dataset sizes of [8] are similar to ours, with input matrices of sizes 4096 x 165 and 3584 x 2414 (ours is 8447 x 3012).

To extend this project and create an even more robust set of results, future work may include measurement of NMF implementation performances on a significantly larger dataset. While the NYT test dataset is not trivially small (10^7 input matrix elements), practical large scale applications of NMF may necessitate input matrices that are many orders of magnitude larger [12]. This poses practical engineering challenges as host and device memory easily become saturated with such large amounts of data.

7. References

[1] D. D. Lee and H. S. Seung, “Algorithms for Non-negative Matrix Factorization,” Advances in Neural Information Processing Systems, no. 13, 2000.

[2] “Euclidean distance,” Wikipedia, 19-Dec-2021. [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_distance#:~:text=In%20mathematics%2C%20the%20Euclidean%20distance,being%20called%20the%20Pythagorean%20distance. [Accessed: 19-Dec-2021].

[3] D. D. Lee and H. S. Seung, “Learning the parts of objects by non-negative matrix factorization,” Nature, vol. 401, no. 6755, pp. 788–791, 1999.

[4] Y.-X. Wang and Y.-J. Zhang, “Nonnegative Matrix Factorization: A Comprehensive Review,” IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, vol. 25, no. 6, pp. 1336–1353, Jun. 2013.

[5] X. Lin and P. C. Boutros, “Optimization and expansion of non-negative matrix factorization,” BMC Bioinformatics, vol. 21, no. 1, 2020.

[6] Ahmadvh, “Non-negative matrix factorization implemented in python/the notebook.ipynb at master · AHMADVH/non-negative-matrix-factorization---implemented-in-python,” *GitHub*. [Online]. Available: https://github.com/ahmadvh/Non-Negative-Matrix-factorization---Implemented-in-python/blob/master/The%20notebook.ipynb. [Accessed: 19-Dec-2021].

[7] Ebattenberg, “NMF-Cuda/nmf.cu at master · ebattenberg/NMF-Cuda,” GitHub. [Online]. Available: https://github.com/ebattenberg/nmf-cuda/blob/master/cuda/nmf.cu. [Accessed: 19-Dec-2021].

[8] S. Koitka and C. M. Friedrich, “nmfgpu4R: GPU-Accelerated Computation of the Non-Negative Matrix Factorization (NMF) Using CUDA Capable Hardware,” The R Journal, vol. 8, no. 2, pp. 382–392, Dec. 2016.

[9] “Sklearn.decomposition.NMF,” scikit. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html. [Accessed: 19-Dec-2021].

[10] Ckaavya, “Ckaavya/topic-modeling-for-the-new-york-times-news-dataset: An nonnegative matrix factorization approach for classifying news topics,” GitHub. [Online]. Available: https://github.com/ckaavya/Topic-Modeling-for-The-New-York-Times-News-Dataset. [Accessed: 19-Dec-2021].

[11] Paisley, J. (2021, November). Elen 4720 Machine Learning for Signals, Information, and Data Lecture 18. Lecture.

[12] Netflix, “Netflix prize data,” Kaggle, 13-Nov-2019. [Online]. Available: https://www.kaggle.com/netflix-inc/netflix-prize-data. [Accessed: 19-Dec-2021].

8. Appendices

8.1.1 Snippet of pyCUDA Euclidean Distance Updates

Allocate Memory on GPU for pyCUDA Euclidean Distance

```
if loss == 'euclidean':
    """
    Begin Euclidean Update
    """
    #define intermediate buffers on gpu for H update
    Wt_d = gpuarray.zeros(((K, N)), dtype=np.float32)
    WtX_d = gpuarray.zeros(((K, M)), dtype=np.float32)
    WtW_d = gpuarray.zeros(((K, K)), dtype=np.float32)
    WtWH_d = gpuarray.zeros(((K, M)), dtype=np.float32)

    # define itermediate buffers for W update
    Ht_d = gpuarray.zeros(((M, K)), dtype=np.float32)
    WH_d = gpuarray.zeros(((N, M)), dtype=np.float32)
    WHHt_d = gpuarray.zeros(((N, K)), dtype=np.float32)
    XHt_d = gpuarray.zeros(((N, K)), dtype=np.float32)
```

H Update pyCUDA Euclidean Distance

```
"""
Begin Euclidean Update H
"""
# Wt = W.T
# H = H * Wt.dot(X) / (Wt.dot(W).dot(H) + eps)

# Wt = W.T
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, K)
event = func_tran(W_d, Wt_d, np.int32(K), np.int32(N), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# Wt * X = WtX
block_dim, grid_dim = context.block_dims, context.grid_dims2d(K,M)
event = func_mul(Wt_d, X_d, WtX_d, np.int32(K), np.int32(N), np.int32(N), np.int32(M), np.int32(K), np.int32(M), block=block_dim, grid=grid_dim)

# Wt * W = WtW
block_dim, grid_dim = context.block_dims, context.grid_dims2d(K,K)
event = func_mul(Wt_d, W_d, WtW_d, np.int32(K), np.int32(N), np.int32(N), np.int32(K), np.int32(K), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# WtW * H = WtWH
block_dim, grid_dim = context.block_dims, context.grid_dims2d(K,M)
event = func_mul(WtW_d, H_d, WtWH_d, np.int32(K), np.int32(K), np.int32(M), np.int32(K), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# WtWH + eps
event = func_add(WtWH_d, np.float32(eps), np.int32(K), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# H .* WtX elementwise
event = func_ele_mul(H_d, WtX_d, np.int32(K), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# H / WtWH elementwise
event = func_div(H_d, WtWH_d, np.int32(K), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()
"""
End Euclidean Update H
"""
```

W Update pyCUDA Euclidean Distance

```
"""
Begin Euclidean Update W
"""
# Ht = H.T #W Transpose
event = func_tran(H_d, Ht_d, np.int32(M), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# X * Ht = XHt
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N,K)
event = func_mul(X_d, Ht_d, XHt_d, np.int32(N), np.int32(M), np.int32(M), np.int32(K), np.int32(N), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# W * H = WH
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N,M)
event = func_mul(W_d, H_d, WH_d, np.int32(N), np.int32(K), np.int32(K), np.int32(M), np.int32(N), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# WH * Ht = WHHt
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N,K)
event = func_mul(WH_d, Ht_d, WHHt_d, np.int32(N), np.int32(M), np.int32(M), np.int32(K), np.int32(N), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# WHHt + eps
event = func_add(WHHt_d, np.float32(eps), np.int32(N), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# W .* XHt elementwise
event = func_ele_mul(W_d, XHt_d, np.int32(N), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# W / WHHt elementwise
event = func_div(W_d, WHHt_d, np.int32(N), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()
"""
End Euclidean Update W
"""
```

8.1.2 Snippet of pyCUDA KL Divergence Updates

Allocate Memory on GPU for KL Divergence Updates

```
elif loss == 'divergence':
    """
    Begin Divergence Update
    """

    # define intermediate buffers for H update
    WH_d = gpuarray.zeros(((N,M)), dtype=np.float32)
    P_d = gpuarray.zeros(((N,M)), dtype=np.float32)
    Wt_d = gpuarray.zeros(((K,N)), dtype=np.float32)
    Wt_sum_d = gpuarray.zeros(((K,1)), dtype=np.float32) #sum rows
    WtP_d = gpuarray.zeros(((K,M)), dtype=np.float32)

    # define intermediate buffers on gpu for W update
    Ht_d = gpuarray.zeros(((M,K)), dtype=np.float32)
    Ht_sum_d = gpuarray.zeros(((1,K)), dtype=np.float32) #sum cols
    PHt_d = gpuarray.zeros(((N,K)), dtype=np.float32)
```

H Update KL Divergence

```
"""
Begin Divergence Update H
"""
# P = X / (W.dot(H)+eps) #intermediate step (purple matrix in notes)
# Wt = W.T
# Wt = Wt / Wt.sum(axis=1).reshape(-1, 1) #normalize rows
# H = H * (Wt.dot(P))  #update H

# W.dot(H)
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, M)
event = func_mul(W_d, H_d, WH_d, np.int32(N), np.int32(K), np.int32(K), np.int32(M), np.int32(N), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# WH + eps
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, M)
event = func_add(WH_d, np.float32(eps), np.int32(N), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# X / WH (saved as P_d)
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, M)
event = func_divC(X_d, WH_d, P_d, np.int32(N), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# Wt = W.T
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, K)
event = func_tran(W_d, Wt_d, np.int32(K), np.int32(N), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# Wt.sum(axis=1).reshape(-1, 1) #sum rows
block_dim, grid_dim = context.block_dims, context.grid_dims2d(K, 1)
event = func_row_sum(Wt_d, Wt_sum_d, np.int32(K), np.int32(N), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# Wt = Wt / Wt_sum_d #elementwise
block_dim, grid_dim = context.block_dims, context.grid_dims2d(K, N)
event = func_row_div(Wt_d, Wt_sum_d, np.int32(K), np.int32(N), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# Wt.dot(P)
block_dim, grid_dim = context.block_dims, context.grid_dims2d(K, M)
event = func_mul(Wt_d, P_d, WtP_d, np.int32(K), np.int32(N), np.int32(N), np.int32(M), np.int32(K), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# H .* WtP elementwise
block_dim, grid_dim = context.block_dims, context.grid_dims2d(K, M)
event = func_ele_mul(H_d, WtP_d, np.int32(K), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()
"""
End Divergence Update H
"""
```

W Update KL Divergence

```
"""
Begin Divergence Update W
"""
# P = X / (W.dot(H)+eps)
# Ht = H.T
# Ht = Ht / Ht.sum(axis=0).reshape(1, -1) #normalize columns
# W = W * (P.dot(Ht))  #update W

# W.dot(H)
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, M)
event = func_mul(W_d, H_d, WH_d, np.int32(N), np.int32(K), np.int32(K), np.int32(M), np.int32(N), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# WH + eps
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, M)
event = func_add(WH_d, np.float32(eps), np.int32(N), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# X / WH (saved as P_d)
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, M)
event = func_divC(X_d, WH_d, P_d, np.int32(N), np.int32(M), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# Ht = H.T #H transpose
block_dim, grid_dim = context.block_dims, context.grid_dims2d(K, M)
event = func_tran(H_d, Ht_d, np.int32(M), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# Ht.sum(axis=0).reshape(1, -1) #sum columns
block_dim, grid_dim = context.block_dims, context.grid_dims2d(1, K)
event = func_col_sum(Ht_d, Ht_sum_d, np.int32(M), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# Ht = Ht / Ht_sum_d #elementwise
block_dim, grid_dim = context.block_dims, context.grid_dims2d(M, K)
event = func_col_div(Ht_d, Ht_sum_d, np.int32(M), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# P.dot(Ht)
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, K)
event = func_mul(P_d, Ht_d, PHt_d, np.int32(N), np.int32(M), np.int32(M), np.int32(K), np.int32(N), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()

# W .* PHt elementwise
block_dim, grid_dim = context.block_dims, context.grid_dims2d(N, K)
event = func_ele_mul(W_d, PHt_d, np.int32(N), np.int32(K), block=block_dim, grid=grid_dim)
cuda.Context.synchronize()
"""
End Divergence Update W
"""
```

8.2 Sample Output ("Learned" W)

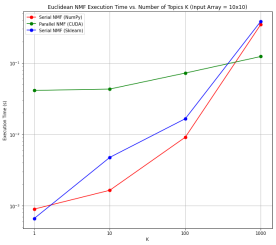
	Topic_1	Topic_2	Topic_3	Topic_4	Topic_5	Topic_6	Topic_7	Topic_8	Topic_9	Topic_10
0	0.0240 percent	0.0623 states	0.0198 campaign	0.0421 father	0.0098 mile	0.0195 official	0.0263 pay	0.0146 company	0.0239 music	0.0202 second
1	0.0197 rate	0.0546 american	0.0180 vote	0.0382 graduate	0.0078 water	0.0134 force	0.0194 member	0.0137 computer	0.0128 performance	0.0201 hit
2	0.0159 increase	0.0426 country	0.0179 political	0.0294 son	0.0075 hour	0.0132 military	0.0159 worker	0.0111 system	0.0126 play	0.0169 third
3	0.0155 rise	0.0236 world	0.0175 party	0.0293 school	0.0073 car	0.0131 government	0.0159 money	0.0097 service	0.0111 perform	0.0149 start
4	0.0133 economy	0.0150 americans	0.0168 election	0.0279 mrs	0.0068 place	0.0111 leader	0.0144 job	0.0096 technology	0.0105 song	0.0140 win
5	0.0123 report	0.0135 international	0.0164 republican	0.0261 daughter	0.0063 drive	0.0104 war	0.0136 union	0.0087 sell	0.0100 concert	0.0136 game
6	0.0113 low	0.0133 foreign	0.0163 candidate	0.0244 mother	0.0057 line	0.0083 troop	0.0119 law	0.0078 customer	0.0095 dance	0.0131 inning
7	0.0111 economic	0.0098 nation	0.0144 democratic	0.0214 student	0.0056 foot	0.0076 peace	0.0116 service	0.0077 information	0.0093 night	0.0123 pitch
8	0.0102 growth	0.0098 european	0.0135 leader	0.0201 receive	0.0055 area	0.0074 attack	0.0114 official	0.0076 offer	0.0092 program	0.0123 home
9	0.0085 decline	0.0093 trade	0.0134 voter	0.0196 marry	0.0053 road	0.0070 agreement	0.0112 bill	0.0073 product	0.0084 sound	0.0116 victory

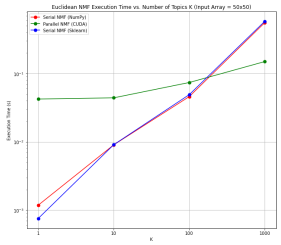
Topic_16	Topic_17	Topic_18	Topic_19	Topic_20	Topic_21	Topic_22	Topic_23	Topic_24	Topic_25
0.0129 art	0.0107 serve	0.0161 question	0.0218 film	0.0192 building	0.0402 company	0.0194 thing	0.0282 case	0.0245 play	0.0129 public
0.0097 artist	0.0097 add	0.0121 ask	0.0175 movie	0.0185 city	0.0296 executive	0.0138 tell	0.0218 court	0.0241 team	0.0093 change
0.0077 exhibition	0.0091 minute	0.0102 report	0.0168 play	0.0131 house	0.0217 president	0.0133 ask	0.0214 lawyer	0.0240 game	0.0091 issue
0.0070 painting	0.0088 pepper	0.0093 issue	0.0140 television	0.0131 build	0.0204 business	0.0118 lot	0.0172 law	0.0190 player	0.0084 power
0.0064 museum	0.0087 food	0.0078 write	0.0120 director	0.0120 area	0.0191 chief	0.0114 feel	0.0144 state	0.0164 season	0.0078 policy
0.0061 collection	0.0084 oil	0.0077 tell	0.0118 character	0.0104 home	0.0138 chairman	0.0093 really	0.0143 judge	0.0140 point	0.0069 political
0.0057 early	0.0083 cook	0.0071 interview	0.0103 production	0.0099 community	0.0112 vice	0.0090 start	0.0137 charge	0.0135 win	0.0062 problem
0.0055 form	0.0082 cup	0.0064 meeting	0.0097 star	0.0096 live	0.0104 industry	0.0086 put	0.0115 trial	0.0116 coach	0.0057 system
0.0055 design	0.0080 sauce	0.0064 add	0.0091 actor	0.0096 resident	0.0093 large	0.0083 little	0.0108 legal	0.0097 second	0.0056 believe
0.0054 view	0.0076 taste	0.0060 member	0.0085 direct	0.0092 site	0.0089 director	0.0082 keep	0.0094 file	0.0088 victory	0.0047 view

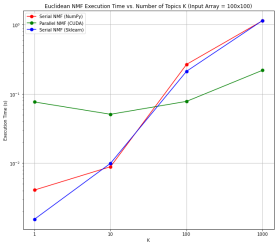
8.3 Result Figures

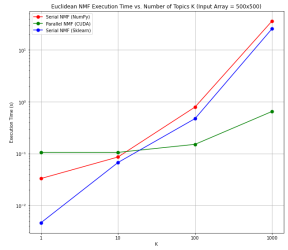
8.3.1 Euclidean Distance - Various K

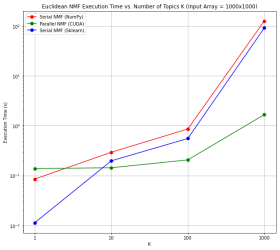
Note that plots may appear small, but they are full resolution images. Zoom in to see plot details.

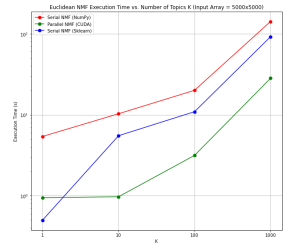




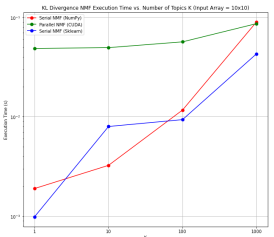


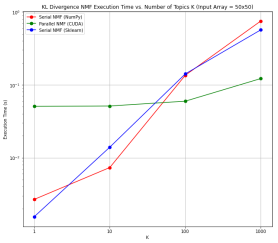


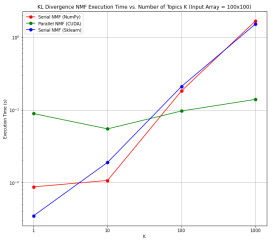


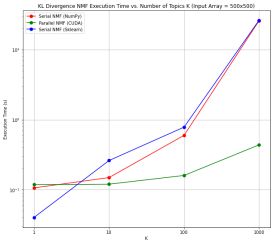


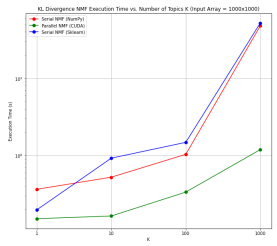
8.3.2 KL Divergence - Various K

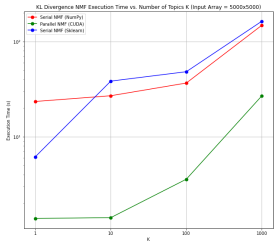




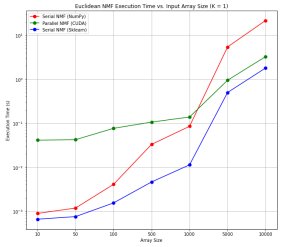


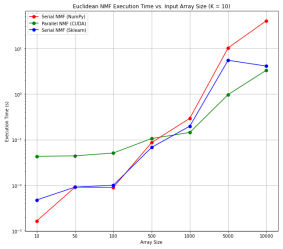


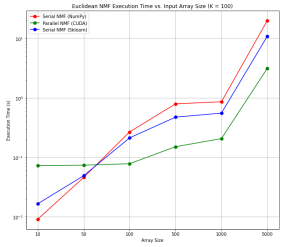


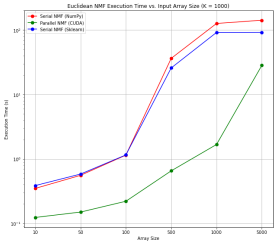


8.3.3 Euclidean Distance - Various Input Size

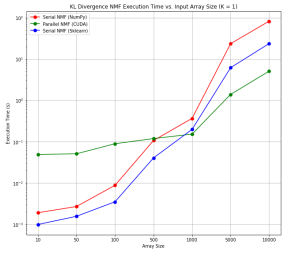


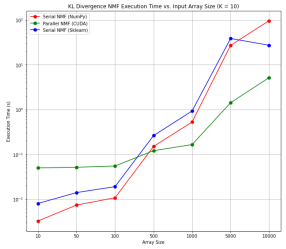


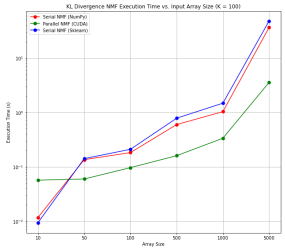


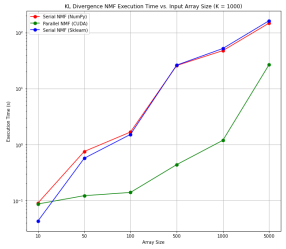


8.3.4 KL Divergence - Various Input Size









8.3.5 Euclidean Distance vs. KL Divergence Execution Speeds