Karen Rivera
Nick Fabrizio

# CS-575 – Term Project

## Non-Negative Matrix Factorization

Non-negative matrix factorization (NNMF) is a method used to factor, or split, a large data set into component parts containing only positive values. With the initial data set represented in rows and columns as a matrix, two matrices of smaller size can be estimated as factors of the initial matrix. The cross product of the component, factored matrices approximate the values in the initial matrix. The component matrices provide insight into the initial data set by deriving weights from the initial matrix.

As an example, take a matrix that contains 100 days of hourly data for chemical readings of a lake in which 3 factories dump their waste. The original matrix does not give a picture of which factory might be dumping which chemical, but with NNMF, two new matrices that map those chemicals to the factories can be generated. One matrix would show which chemicals are most prominent on which days and the other would show which factory dumps their waste on those corresponding days. The non-negative part would be equivalent to nothing ever happening to clean this lake (a negative reading of the chemicals). If the factories are responsible for little overlap in the chemicals they are dumping, then this NNMF model will eventually converge.

Some of the more common applications include text mining, recommendation systems and image analysis. Really any data set that includes many features that are ambiguous and can be teased apart to yield more meaningful data sets [1].

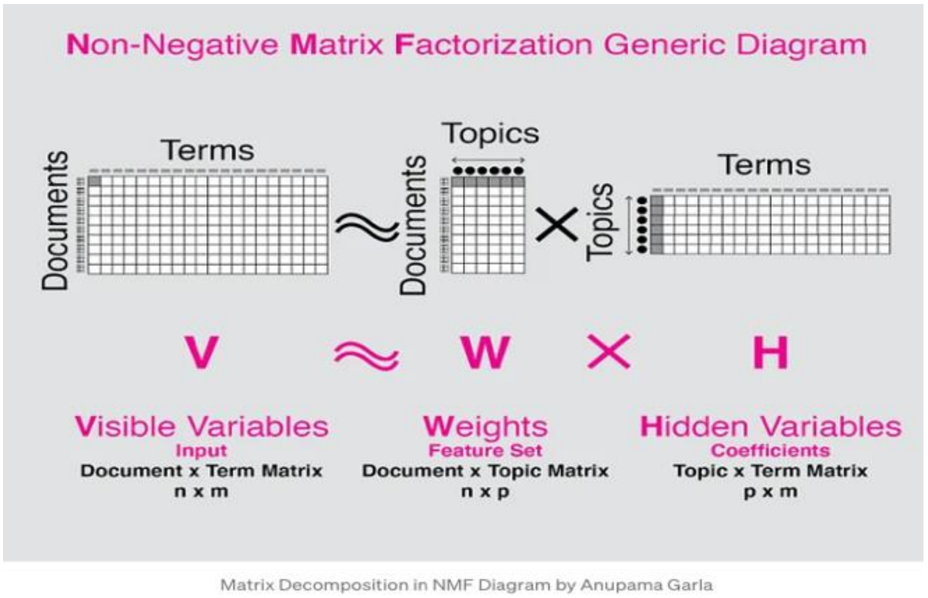


*Diagram 1. Non-Negative matrix factorization diagram and equation*

## Algorithm Description

*Non-negative Matrix Factorization*

The code analyzed for this project calculates the NNMF of a user-provided data set for a user-defined number of classes and is solved iteratively using a variation of the Itakura-Saito Divergence (ISD) criterion. The general equation that this algorithm attempts to solve is Equation 1, where V is the input matrix, W and H are the output component matrices and E is independent and identically distributed Gamma noise in the measured signal [2].

$$V \approx (WH).E$$

*Equation 1. NNMF with Multiplicative Gamma Noise*

When used with NNMF, ISD can be used to determine the quality of the factorization [3]. Rather than using π to calculate the ISD as in the standard ISD equation, this code inserts epsilon, which represents the Gamma noise, into the ISD equation for simplification. The equation used to calculate ISD in this program is shown in Equation 2 below where $P(\omega)$ is the input data matrix, $(\omega)$ is the intermediate, or estimated, data matrix and $\varepsilon$ is the Gamma noise.

$$\sum_{i=0}^{n}\left[\frac{P(\omega)+\varepsilon}{\overline{P}(\omega)+\varepsilon} - \log\left(\frac{P(\omega)+\varepsilon}{\overline{P}(\omega)+\varepsilon}\right)\right] - 1$$

*Equation 2. Modified Itakura-Saito Divergence*

This NNMF program accepts three arguments: the number of classes to use for the factorization, the number of iterations to perform and the path to the file containing the initial data matrix. The program begins by initializing the W and H output matrices with random numbers in the desired output shape. The desired output shape is the number of rows by number of input classes for the W output matrix, and the number of input classes by the number of columns for the H output matrix. The program also initializes an $\hat{X}$ matrix for holding the intermediate cross product of W and H each iteration. Each element in the $\hat{X}$ matrix is initialized to zero. During each iteration, the cross product of W and H is calculated and stored in the $\hat{X}$ matrix, and Equation 2 is used with the inputs of the original input data matrix, $\hat{X}$ and $\varepsilon$ to calculate the ISD for that iteration. If the ISD of the current iteration is either less than five percent greater than that of the previous iteration or if the difference between the current and previous ISD is less than the convergence threshold, the program stops and saves the W and H matrices to their respective files. If neither of these cases holds true, the program adjusts the W matrix using Equation 3 followed by updating $\hat{X}$ with the cross product of the updated W matrix the original H matrix. In Equation 3, $P(\omega)$ is the input data matrix, $(\omega)$ is the intermediate, or estimated, data matrix and $\varepsilon$ is the Gamma noise.

$$W\cdot\left(\frac{\left(P(\omega)+\varepsilon\right)\cdot H/\left(\overline{P}(\omega)+\varepsilon\right)^{2}}{H/\left(\overline{P}(\omega)+\varepsilon\right)}\right)^{\frac{1}{2}}$$

*Equation 3. W Matrix Update*

The program then updates the H matrix using Equation 4 followed by updating the values in $\hat{X}$ with the cross product of the updated W matrix the updated H matrix. In Equation 4, $P(\omega)$ is the input data matrix, $(\omega)$ is the intermediate, or estimated, data matrix and $\varepsilon$ is the Gamma noise. At this point, the current iteration is complete, and the next iteration begins.

$$H \cdot \left( \frac{\left(P(\omega) + \varepsilon\right) \cdot W / \left(\overline{P}(\omega) + \varepsilon\right)^2}{W / \left(\overline{P}(\omega) + \varepsilon\right)} \right)^{\frac{1}{2}}$$

*Equation 4. H Matrix Update*

*Data set creation*

Beyond the NNMF algorithm listed above, data sets used must have the potential of converging given a known number of features, like the factory dumping example in the introduction. To generate these data sets, random numbers were calculated within a range for each dimension. The ranges were intentionally overlapped so the convergence would not happen right away. Once the three data sets, which included small, medium, and large sizes, were generated, the experiments were able to be run using the sequential, optimized, vectorized and parallelized versions of the code with the confidence that the output was consistent. The small data set has 10,000 rows with three variants in the first column (factories) and three features (chemicals). The medium data set has 10,000 rows with three variants in the first column and six features. The large data set has 10,000 rows with three variants in the first column and 10 features. The sizes for the data sets were intentionally kept relatively small because the execution times for data sets with larger dimensions were measured in hours.

*Matrix Comparison*

In addition to working with consistent data sets and a constant seed for random generation during the NNMF process, a matrix comparison method was added to the end of the process. This method compares the original matrix with the output W and H matrices from the NNMF process and validates that the product is within a given threshold for each data set size. Since this was only used for validation of the changes to the original algorithm, the validation time is not included in the overall timing of the NNMF routine. For the small data set, a threshold of one was sufficient to verify the results, for the medium data set, the threshold was set to five, and for the large data set, the threshold was set to 15.

*Python solution*

During the investigative process, data localization and blocking of the multiple loops were discovered to have very little impact on the overall time spent in the NNMF calculation. Much of the NNMF algorithm consists of multiple nested loops. A python solution was generated that uses NumPy for the dot multiplications and Frobenius norm to determine the residual. This solution removes the multiple loops, at least in the highest level of computation and was created simply for a comparison to a solution that utilizes pre-optimized solutions with NumPy. Timings during the iterations were added to be a consistent measure against the C implementation and similar thresholds were used to verify the end results. This algorithm, while producing different ISD values during the process, always met the same thresholds with a similar matrix validation method performed at the end.

Using a combination of the above algorithms, tests were run on 6 different programs:

- nmf: This is the sequential version, instrumented with floating point operation calculations and timing.
- nmfOpt: Copied from sequential with the same instrumentation and timings, this version adds blocking and initial optimization efforts.
- nmfPar: Also copied from the sequential version, this version adds OpenMP pragmas around the multiple for loops to process the data on multiple threads
- nmfVec: Copied from the sequential version, this version utilizes OpenMP vectorization and reduction for the i loops. This is an approach not yet utilized in previous programming assignments and came as a suggestion from the Intel Advisor prompts.
- nmfOptPar: Starting from nmfOpt, the same type of OpenMP pragmas were added to determine if a combined approach yields better results
- pythonNMF: Python version of NMF that eliminates the multiple for loops but still contains instrumentation for floating point operation calculations and timing. ISD calculations were also added at the same intervals to better mimic the output of the other implementations.

The 5 C programs all produce the same WxH matrices for the 3 different data sizes. This was verified by utilizing the ISD start and end values and a manual compare of the output matrices. The python version produced slightly different results for WxH but maintained the same threshold as the C implementations.

## Description of Optimization Approach

For the nmfOpt algorithm, each of the row-wise iteration loops were blocked utilizing the same block size to take advantage of data localization. Other means of optimization were assessed such as combining loops, inverting loops and reducing iterations; however, none of these optimization methods were effective or feasible due to the nature of each of the loops.

For the nmfPar algorithm, OpenMP (OMP) was integrated into the original nmf algorithm to parallelize each of the loops. In all cases, the outer loops were legally parallelizable, and in all but one case, the outer loops were parallelized. The one case in which the outer loop was not parallelized was that in which the outer loop iterates over the number of classes. Since the number of iterations in this loop will always be smaller than the number of rows and columns, the decision was made to move parallelization to the first nested loop instead. Each OMP pragma used was a parallel for pragma declaring one or more variables private. The loop initializing the ISD in the algorithm also required the use of the reduction clause since it reduces the values from each iteration.

For the nmfVec algorithm, OMP was again integrated into the sequential nmf algorithm using both the OMP parallel for and simd directives with the collapse and reduction clauses, where appropriate. These pragmas were used in all the same places that the parallel for pragmas were used in nmfPar. The reduction clause was used for the ISD initialization loop, and the collapse clause was used wherever there was a simple nested loop. The simd directive vectorizes the loop, and the collapse method combines nested loops into single loops [4].

The Python NumPy package, which is used to compute the dot product of matrices, implements highly optimized C code [5]. Depending on the dimensions of W and H, it will call an underlying function from its linear algebra library to perform the multiplication. Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) are two of these underlying libraries. Options for optimization include loop unrolling, cache-aware algorithms and vectorized instructions.

## **Experimental Setup**

### *Describe the machine*

The machines these experiments were run on are dover and montpelier. Dover and montpelier each have 12 cores per socket, an L1d cache size of 48K, an L1i cache size of 32K, an L2 cache size of 1280K and an L3 cache size of 25600K.

### *List experiments planned*

Prior to running the experiments, the results of each optimization approach were validated against the results produced by the sequential version of the algorithm. Validation was done at the time each optimized algorithm was run during the experiments. This was accomplished by integrating a validation script into the output of the algorithm. All programs tested were run seven times each, and the execution times were recorded. The executions were repeated seven times for each variant to account for normal fluctuations in execution times and provide a means for removing outliers from the result set. Each experiment was executed with varying numbers of threads from one to eight.

A constant seed for the random number generator was used to keep results of the three data set convergences consistent. Initial and final ISD values are compared with each output to ensure parallelization attempts don't change the output.

### *Sequential baseline of NMF (nmf)*

The initial code, instrumented with floating point operation counters and a final GFLOP calculation will be executed seven times to get a baseline average for each of the three data sets. The slowest and fastest times are dropped, and an average found. In addition to the average baseline time, Intel Advisor will be executed using medium accuracy to determine the roofline metrics. Initial and final ISD values will be recorded to ensure consistency between the optimization results.

### *Optimized NMF (nmfOpt)*

Block sizes ranging from 10 to 10,000, which is the maximum number of rows in the data sets, will be used to determine if blocking the i loop, rows in the data set, improves localization. Again, each test will be run seven times for each of the three data set sizes, dropping the lowest and highest values to get an average that can be compared against the sequential execution time. This optimization method will only be run using a single thread as parallelization has not yet been implemented. Intel Advisor will once again be run against this version of the code and GFLOPS, ISD starting and ending values will be compared.

### *Parallelized NMF (nmfPar)*

Each data set will be run for one to eight threads. The average time spent will once again be gathered for each thread execution, and speedup, efficiency and average execution times will be plotted against the sequential time. Intel Advisor for medium accuracy will be generated for the roofline metrics and GFLOPS, ISD starting and ending values will be compared.

*Parallelized and Optimized NMF (nmfOptPar)*
The gain achieved from blocking was minimal, but as the problem size increased, a more substantial gain from the original optimization was achieved. This process combines the gain from the blocking in nmfOpt with the gain from the parallelization approach (nmfPar) to get an overall effect. Experiments planned will be like nmfPar with 1-8 threads for the best common block size of 250. Intel Advisor roofline metrics will also be gathered for this approach.

*Vectorized NMF (nmfVec)*
Each data set will be run for one to eight threads. Average time spent will be calculated as it was with nmfPar. All GFLOPS, ISD and Intel Advisor tests will be gathered and compared per usual.

*Python implementation (pythonNMF.py)*
Only the sequential version of this code will be executed for comparison purposes to the best optimization results. Seven sequential executions will be performed to get the average of the middle five execution times. No Intel Advisor calculation will be done as this is a python solution, but GFLOPS will be calculated for comparison.

## Experimental Results

*Time comparison and ISD verification*
Small arguments: ./nmf<Opt/Par/OptPar> -N 3 -I 10000 -T 1 -B 50./test_data/small_data.dat
Medium arguments: ./nmf<Opt/Par/OptPar> -N 6 -I 10000 -T 5 -B 500 ./test_data/med_data.dat
Large arguments: ./nmf <Opt/Par/OptPar> -N 10 -I 10000 -T 15 -B 250 ./test_data/large_data.dat

| Algorithm | Problem Size | Avg Time (sec) | GFLOPS | ISD Start | ISD End |
|---|---|---|---|---|---|
| nmf | Small | 0.53962 | 2.424389 | 186297.79663714 | Converged |
| nmf | Medium | 9.21554 | 3.181786 | 1805581.09459682 | 0.16445100 |
| nmf | Large | 63.9705 | 2.543934 | 6267531.91859795 | 1.56581928 |
| nmfOpt | Small | 0.538444 | 2.535283 | 186297.79663714 | Converged |
| nmfOpt | Medium | 9.245854 | 3.200688 | 1805581.09459682 | 0.16445100 |
| nmfOpt` | Large | 62.91346 | 2.727453 | 6267531.91859795 | 1.56581928 |
| nmfPar | Small | 0.46821 | 2.893881 | 186297.79663714 | Converged |
| nmfPar | Medium | 8.45276 | 2.469560 | 1805581.09459682 | 0.16445100 |
| nmfPar | Large | 62.7672 | 2.953854 | 6267531.91859795 | 1.56581928 |
| nmfOptPar | Small | 0.467674 | 2.889739 | 186297.79663714 | Converged |
| nmfOptPar | Medium | 8.314348 | 3.580422 | 1805581.09459682 | 0.16445100 |
| nmfOptPar | Large | 58.05102 | 2.951481 | 6267531.91859795 | 1.56581928 |
| nmfVec | Small | 0.49958 | 2.519029 | 186297.79663714 | Converged |
| nmfVec | Medium | 8.29798 | 3.455687 | 1805581.09459682 | 0.16445100 |
| nmfVec | Large | 58.7616 | 2.925134 | 6267531.91859795 | 1.56581928 |

*Table 1. NMF algorithm variations average time, GFLOPS and ISD data*

Table 1 shows that all ISD start and end values are equivalent, proving that the implemented algorithms are legal. For a single thread, there are no drastic differences among the average times as seen in Figure 1.

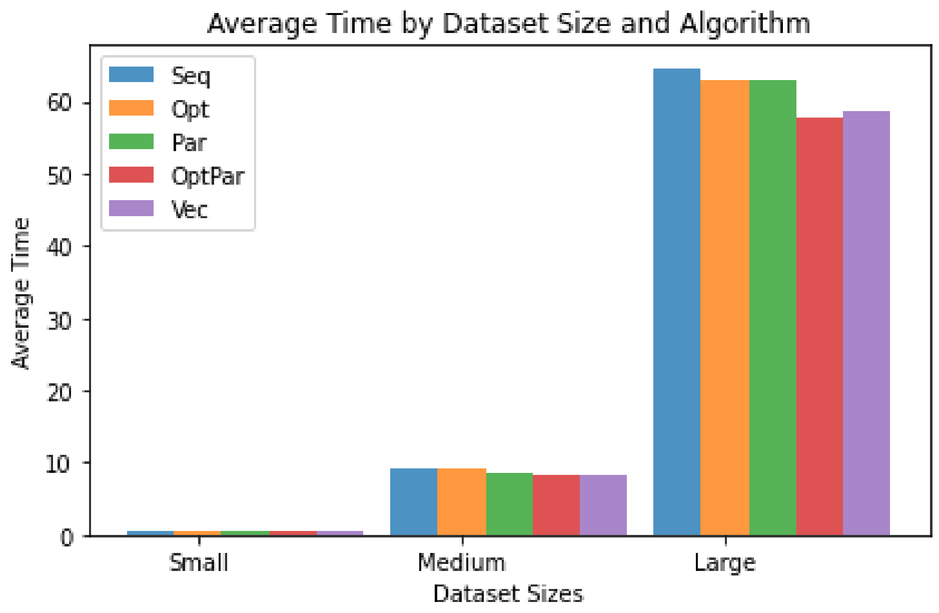
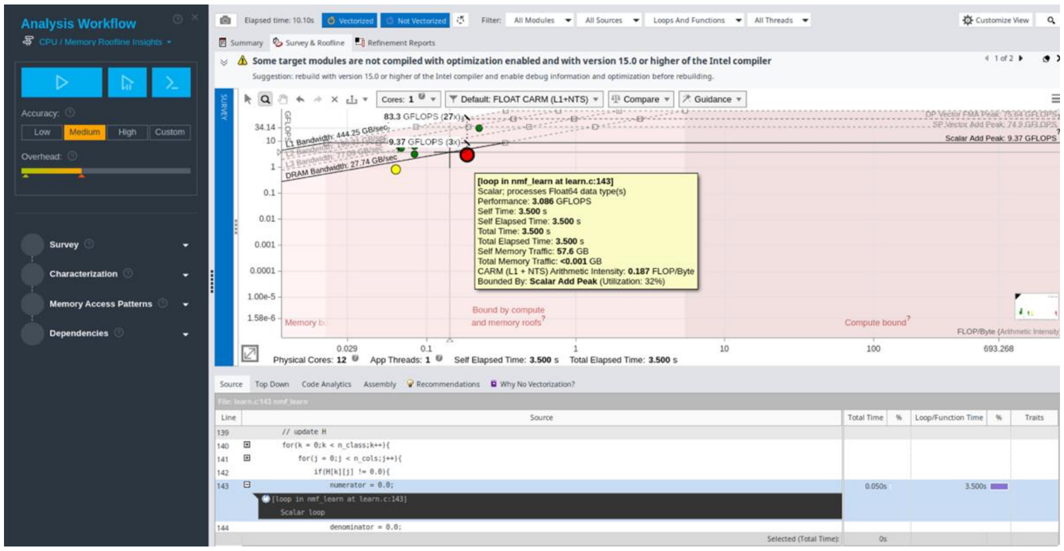


*Figure 1 Average Sequential Time for each Algorithm and Dataset Size*

## *Intel Advisor results*

Intel Advisor was run with medium accuracy for all 3 datasets but choose to focus just on the medium dataset for the results displayed here. There is so much data on each screen that it is difficult to compare results beyond a single dataset size.



Elapsed time: 10.10s

Some target modules are not compiled with optimization enabled and with version 15.0 or higher of the Intel compiler

Suggestion: rebuild with version 15.0 or higher of the Intel compiler and enable debug information and optimization before rebuilding.

Cores: 1 | Default: FLOAT CARM (L1+NTS)

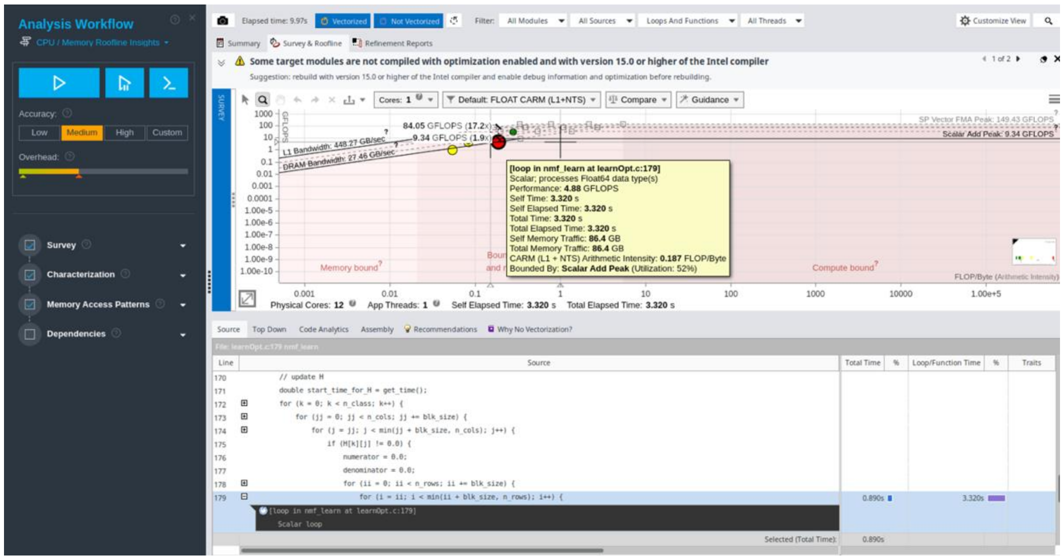
```
[loop in nmf_learn at learn.c:143]
Scalar; processes Float64 data type(s)
Performance: 3.086 GFLOPS
Self Time: 3.500 s
Self Elapsed Time: 3.500 s
Total Time: 3.500 s
Total Elapsed Time: 3.500 s
Self Memory Traffic: 57.6 GB
Total Memory Traffic: <0.001 GB
CARM (L1 + NTS) Arithmetic Intensity: 0.187 FLOP/Byte
Bounded By: Scalar Add Peak (Utilization: 32%)
```

Physical Cores: 12   App Threads: 1   Self Elapsed Time: 3.500 s   Total Elapsed Time: 3.500 s

File: learn.c:143 nmf_learn

| Line | Source | Total Time | % | Loop/Function Time | % | Traits |
|---|---|---|---|---|---|---|
| 139 | // update H | | | | | |
| 140 | for(k = 0;k < n_class;k++){ | | | | | |
| 141 | for(j = 0;j < n_cols;j++){ | | | | | |
| 142 | if(H[k][j] != 0.0){ | | | | | |
| 143 | numerator = 0.0; [loop in nmf_learn at learn.c:143] Scalar loop | 0.050s | | 3.500s | | |
| 144 | denominator = 0.0; | | | | | |
| | Selected (Total Time): | 0s | | | | |

*Figure 2. Intel Advisor Sequential Medium dataset*



Elapsed time: 9.97s

**Some target modules are not compiled with optimization enabled and with version 15.0 or higher of the Intel compiler**

Suggestion: rebuild with version 15.0 or higher of the Intel compiler and enable debug information and optimization before rebuilding.

Cores: 1 | Default: FLOAT CARM (L1+NTS)

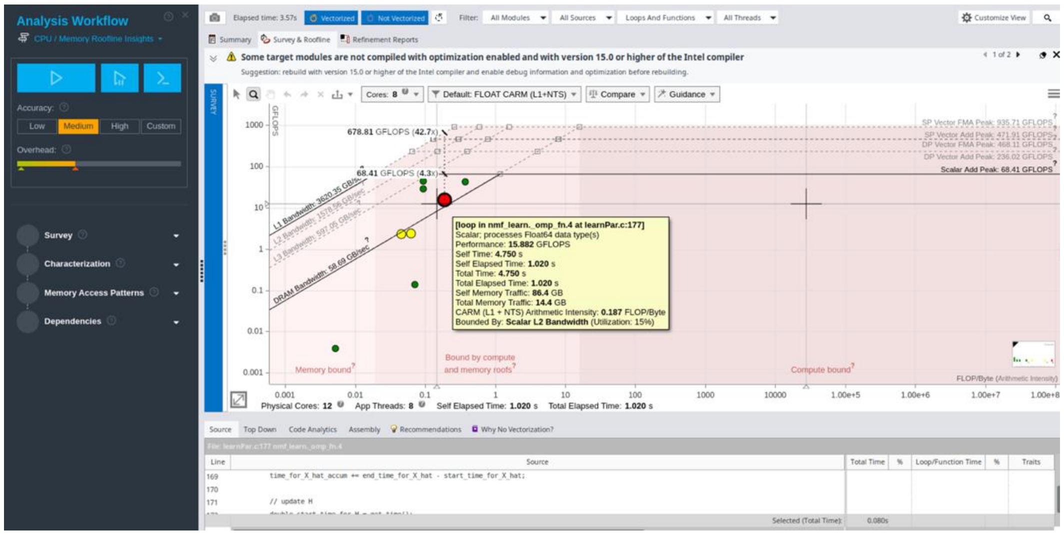
**[loop in nmf_learn at learnOpt.c:179]**
Scalar; processes Float64 data type(s)
Performance: **4.88** GFLOPS
Self Time: **3.320** s
Self Elapsed Time: **3.320** s
Total Time: **3.320** s
Total Elapsed Time: **3.320** s
Self Memory Traffic: **86.4** GB
Total Memory Traffic: **86.4** GB
CARM (L1 + NTS) Arithmetic Intensity: **0.187** FLOP/Byte
Bounded By: **Scalar Add Peak** (Utilization: 52%)

Physical Cores: **12** App Threads: **1** Self Elapsed Time: **3.320** s Total Elapsed Time: **3.320** s

File: learnOpt.c:179 nmf_learn

| Line | Source | Total Time | % | Loop/Function Time | % | Traits |
|---|---|---|---|---|---|---|
| 170 | `// update H` | | | | | |
| 171 | `double start_time_for_H = get_time();` | | | | | |
| 172 | `for (k = 0; k < n_class; k++) {` | | | | | |
| 173 | `for (jj = 0; jj < n_cols; jj += blk_size) {` | | | | | |
| 174 | `for (j = jj; j < min(jj + blk_size, n_cols); j++) {` | | | | | |
| 175 | `if (H[k][j] != 0.0) {` | | | | | |
| 176 | `numerator = 0.0;` | | | | | |
| 177 | `denominator = 0.0;` | | | | | |
| 178 | `for (ii = 0; ii < n_rows; ii += blk_size) {` | | | | | |
| 179 | `for (i = ii; i < min(ii + blk_size, n_rows); i++) {` | 0.890s | | 3.320s | | |
| | `[loop in nmf_learn at learnOpt.c:179]` `Scalar loop` | | | | | |
| | Selected (Total Time): | 0.890s | | | | |

*Figure 3. Intel Advisor Optimized Medium dataset*



Elapsed time: 3.57s

**Some target modules are not compiled with optimization enabled and with version 15.0 or higher of the Intel compiler**

Suggestion: rebuild with version 15.0 or higher of the Intel compiler and enable debug information and optimization before rebuilding.

Cores: 8 | Default: FLOAT CARM (L1+NTS)

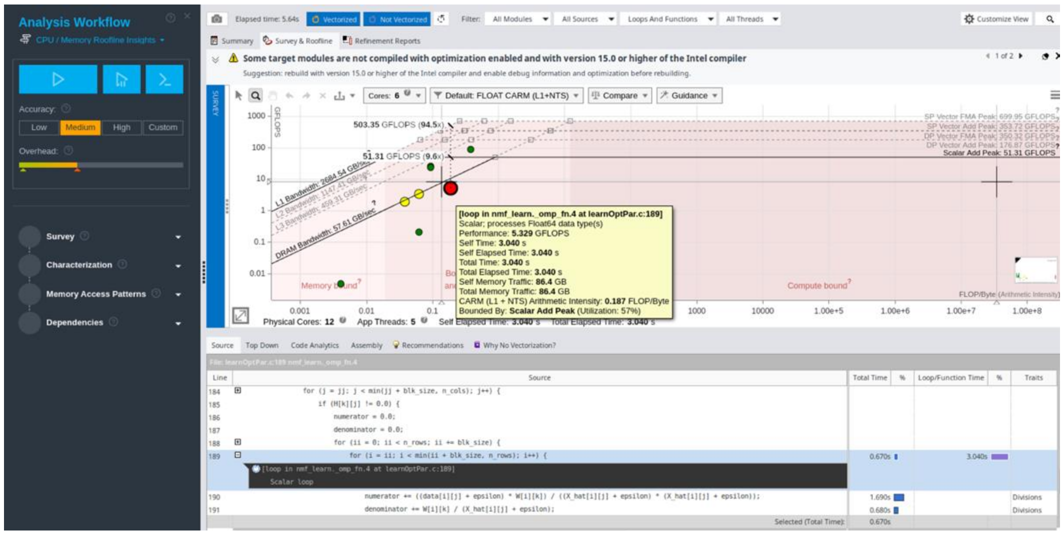
**[loop in nmf_learn._omp_fn.4 at learnPar.c:177]**
Scalar; processes Float64 data type(s)
Performance: **15.882** GFLOPS
Self Time: **4.750** s
Self Elapsed Time: **1.020** s
Total Time: **4.750** s
Total Elapsed Time: **1.020** s
Self Memory Traffic: **86.4** GB
Total Memory Traffic: **14.4** GB
CARM (L1 + NTS) Arithmetic Intensity: **0.187** FLOP/Byte
Bounded By: **Scalar L2 Bandwidth** (Utilization: 15%)

Physical Cores: **12** App Threads: **8** Self Elapsed Time: **1.020** s Total Elapsed Time: **1.020** s

File: learnPar.c:177 nmf_learn._omp_fn.4

| Line | Source | Total Time | % | Loop/Function Time | % | Traits |
|---|---|---|---|---|---|---|
| 169 | `time_for_X_hat_accum += end_time_for_X_hat - start_time_for_X_hat;` | | | | | |
| 170 | | | | | | |
| 171 | `// update H` | | | | | |
| | Selected (Total Time): | 0.080s | | | | |

*Figure 4. Intel Advisor Parallelized Medium dataset (run with 8 threads – best for medium ds)*



Elapsed time: 5.64s

**Some target modules are not compiled with optimization enabled and with version 15.0 or higher of the Intel compiler**

Suggestion: rebuild with version 15.0 or higher of the Intel compiler and enable debug information and optimization before rebuilding.

Cores: 6 | Default: FLOAT CARM (L1+NTS)

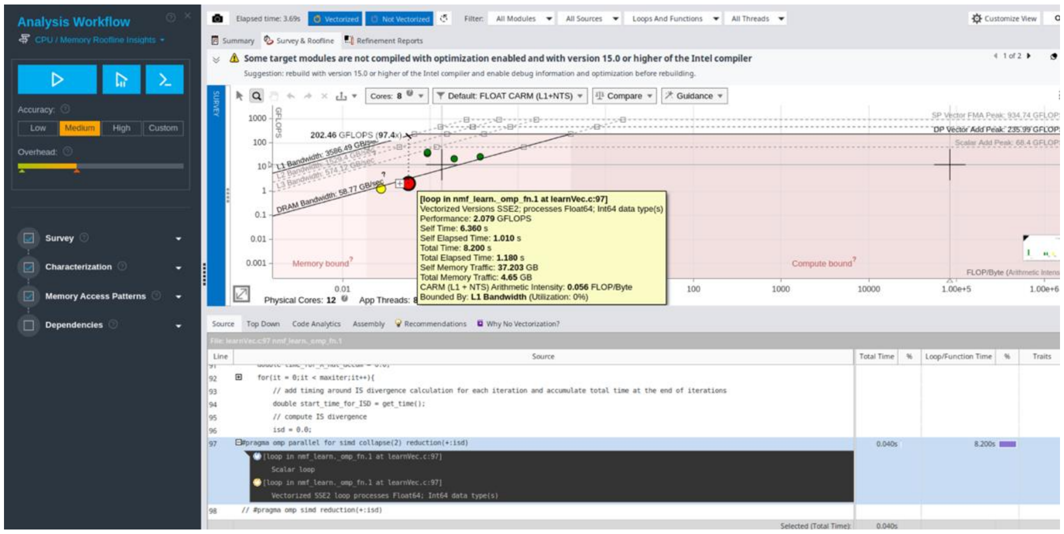
**[loop in nmf_learn._omp_fn.4 at learnOptPar.c:189]**
Scalar; processes Float64 data type(s)
Performance: **5.329** GFLOPS
Self Time: **3.040** s
Self Elapsed Time: **3.040** s
Total Time: **3.040** s
Total Elapsed Time: **3.040** s
Self Memory Traffic: **86.4** GB
Total Memory Traffic: **86.4** GB
CARM (L1 + NTS) Arithmetic Intensity: **0.187** FLOP/Byte
Bounded By: **Scalar Add Peak** (Utilization: 57%)

Physical Cores: **12** App Threads: **5** Self Elapsed Time: **3.040** s Total Elapsed Time: **3.040** s

File: learnOptPar.c:189 nmf_learn._omp_fn.4

| Line | Source | Total Time | % | Loop/Function Time | % | Traits |
|---|---|---|---|---|---|---|
| 184 | `for (j = jj; j < min(jj + blk_size, n_cols); j++) {` | | | | | |
| 185 | `if (H[k][j] != 0.0) {` | | | | | |
| 186 | `numerator = 0.0;` | | | | | |
| 187 | `denominator = 0.0;` | | | | | |
| 188 | `for (ii = 0; ii < n_rows; ii += blk_size) {` | | | | | |
| 189 | `for (i = ii; i < min(ii + blk_size, n_rows); i++) {` | 0.670s | | 3.040s | | |
| | `[loop in nmf_learn._omp_fn.4 at learnOptPar.c:189]` `Scalar loop` | | | | | |
| 190 | `numerator += ((data[i][j] + epsilon) * W[i][k]) / ((X_hat[i][j] + epsilon) * (X_hat[i][j] + epsilon));` | 1.690s | | | | Divisions |
| 191 | `denominator += W[i][k] / (X_hat[i][j] + epsilon);` | 0.680s | | | | Divisions |
| | Selected (Total Time): | 0.670s | | | | |

*Figure 5. Intel Advisor Opt/Par Medium dataset (run with 5 threads – best for medium ds)*



Some target modules are not compiled with optimization enabled and with version 15.0 or higher of the Intel compiler

Suggestion: rebuild with version 15.0 or higher of the Intel compiler and enable debug information and optimization before rebuilding.

[loop in nmf_learn._omp_fn.1 at learnVec.c:97]
Vectorized Versions SSE2; processes Float64; Int64 data type(s)
Performance: **2.079** GFLOPS
Self Time: **6.360** s
Self Elapsed Time: **1.010** s
Total Time: **8.200** s
Total Elapsed Time: **1.180** s
Self Memory Traffic: **37.203** GB
Total Memory Traffic: **4.65** GB
CARM (L1 + NTS) Arithmetic Intensity: **0.056** FLOP/Byte
Bounded By: **L1 Bandwidth** (Utilization: 0%)

```
for(it = 0;it < maxiter;it++){
    // add timing around IS divergence calculation for each iteration and accumulate total time at the end of iterations
    double start_time_for_ISD = get_time();
    // compute IS divergence
    isd = 0.0;
#pragma omp parallel for simd collapse(2) reduction(+:isd)        0.040s        8.200s
  [loop in nmf_learn._omp_fn.1 at learnVec.c:97]
    Scalar loop
  [loop in nmf_learn._omp_fn.1 at learnVec.c:97]
    Vectorized SSE2 loop processes Float64; Int64 data type(s)
// #pragma omp simd reduction(+:isd)
```

*Figure 6. Intel Advisor Vec Medium dataset (run with 8 threads – best for medium ds)*

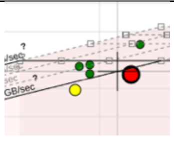
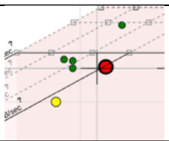
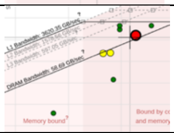
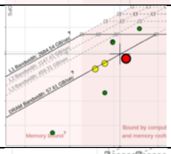
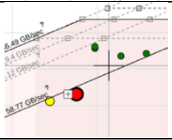
| Algorithm | Overall dot clustering | Highest GFLOPS | Code location of highest dot | Scalar Add Peak | Vector Add Peak | TMP | Bounded By |
|---|---|---|---|---|---|---|---|
| nmf |  | 34.332 | Update W, learn.c:113 | 9.37 | 75.64 | 151.49 | DP Vector Add Peak (91%) |
| nmfOpt |  | 33.616 | Update W, learnOpt.c:134 | 9.34 | 37.79 | 149.43 | DP Vector Add Peak (88%) |
| nmfPar (8 threads) |  | 45.000 | Update X_hat, learnPar.c:162 | 68.41 | 236.02 | 935.71 | Scalar Add Peak (8%) |
| nmfOptPar (5 threads) |  | 89.638 | Update W, learnOptPar.c:142 | 51.34 | 176.87 | 699.95 | DP Vector Add Peak (10%) |
| nmfVec (8 threads) |  | 41.539 | Update X_hat, learnVec.c:209 | 68.4 | 471.95 | 934.74 | Scalar Add Peak (7%) |

*Table 2. Intel Advisor Highest Achieving Loop Comparison*

Table 2 may suggest that nmfOptPar provides the best results with the highest GFLOP, but that was just the most efficient of the loops. The other, less performant loops pull that algorithm down. nmfPar and nmfVec were both Scalar Add Peak bound instead of Vector Add Peak bound. nmfPar and nmfVec also have multiple green dots clustered near the highest value shown in the table. It is difficult to compare the clustering because the scales are different between each algorithm, but it is a good visual tool to see the disbursement of the problem spots in the code.

*nmfOpt Results*

| | | Block Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Problem Size | Seq | 10 | 25 | 50 | 100 | 250 | 500 | 1000 | 2000 | 5000 | 10000 |
| small | 0.53962 | 0.5520 | 0.5425 | 0.5380 | 0.53855 | 0.5399 | 0.5474 | 0.5349 | 0.5357 | 0.5321 | 0.5412 |
| medium | 9.21554 | 9.5409 | 9.3788 | 9.3917 | 9.3637 | 9.3725 | 9.34729 | 9.4383 | 9.4586 | 9.3526 | 9.4503 |
| large | 63.9705 | 60.815 | 62.229 | 60.572 | 60.5402 | 60.517 | 60.9396 | 61.201 | 62.127 | 60.779 | 60.6673 |

*Table 3. Execution times of nmfOpt compared to nmf by block size*

Table 3 and Figures 7, 8 and 9 show the execution times of the sequential, nmf, algorithm and the nmfOpt algorithm with varying block sizes. Although the optimal block size for a machine with an L1d cache size of 48K and a data type of double should be around 6000 as shown in Equation 5, the optimal block size for nmfOpt varied based on the size of the problem. For the small data set, the optimal block size was 50, which was only slightly better than using a block size of 100. Blocking the small data set yielded a performance marginally better than the nmf algorithm. The optimal block size for the medium data set was 500, and the performance of nmfOpt at this block size had a slower execution time than the nmf algorithm. For nmfOpt with the large data set, the best performance was achieved with a block size of 250, and it outperformed the nmf algorithm by a larger margin than when using the small data set. For this reason, using even larger data sets may show greater improvements in execution times when utilizing blocking. Another reason blocking is not very beneficial for this algorithm may be due to the data being stored in a two-dimensional array. Since all the data sets used had a small number of columns and a large number of rows, either transposing the input data sets or converting them to one-dimensional arrays and using strided access to retrieve the values might yield a greater benefit with regard to data locality than the original algorithm. Overall, blocking the nmf algorithm did not provide any significant performance improvements.

$$\frac{48000\,bytes}{\frac{8000\,bytes}{double}} = 6000\,doubles \qquad \textit{Equation 5}$$

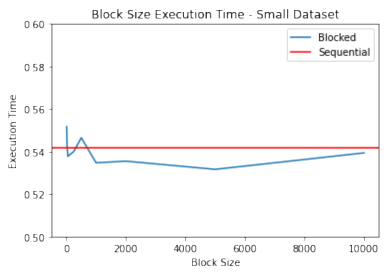


*Figure 7. Execution times for nmfOpt compared to nmf by block size for small data*

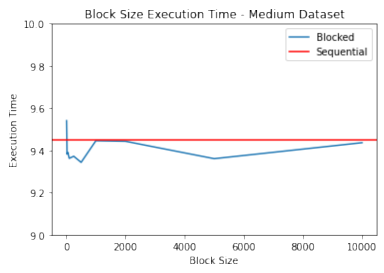


*Figure 8. Execution times for nmfOpt compared to nmf by block size for medium data*

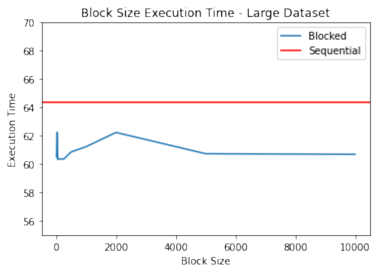


*Figure 9. Execution times for nmfOpt compared to nmf by block size for large data*

## *nmfPar Results*

| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 0.00377 | 0.472 | 1.145 | 1.145 |
| 1 | 2 | 0.00800 | 0.379 | 1.426 | 0.713 |
| 2 | 3 | 0.00905 | 0.276 | 1.958 | 0.653 |
| 3 | 4 | 0.01687 | 0.301 | 1.795 | 0.449 |
| 4 | 5 | 0.01422 | 0.279 | 1.937 | 0.387 |
| 5 | 6 | 0.00840 | 0.300 | 1.801 | 0.300 |
| 6 | 7 | 0.01446 | 0.310 | 1.743 | 0.249 |
| 7 | 8 | 0.00926 | 0.306 | 1.766 | 0.221 |

*Table 4. Execution times, speedup and efficiency for nmfPar compared to nmf with small data*

| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 0.03472 | 8.497 | 1.086 | 1.086 |
| 1 | 2 | 0.02679 | 5.678 | 1.626 | 0.813 |
| 2 | 3 | 0.05424 | 3.741 | 2.467 | 0.822 |
| 3 | 4 | 0.05578 | 3.650 | 2.529 | 0.632 |
| 4 | 5 | 0.05917 | 3.229 | 2.858 | 0.572 |
| 5 | 6 | 0.14038 | 2.813 | 3.281 | 0.547 |
| 6 | 7 | 0.05577 | 2.795 | 3.302 | 0.472 |
| 7 | 8 | 0.05937 | 2.701 | 3.417 | 0.427 |

*Table 5. Execution times, speedup and efficiency for nmfPar compared to nmf with medium data*

| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 0.03366 | 54.924 | 1.175 | 1.175 |
| 1 | 2 | 0.10714 | 33.682 | 1.917 | 0.958 |
| 2 | 3 | 0.28366 | 24.392 | 2.647 | 0.882 |
| 3 | 4 | 0.44626 | 18.682 | 3.456 | 0.864 |
| 4 | 5 | 0.09126 | 15.657 | 4.124 | 0.825 |
| 5 | 6 | 0.32024 | 15.062 | 4.286 | 0.714 |
| 6 | 7 | 0.25921 | 14.547 | 4.438 | 0.634 |
| 7 | 8 | 0.12160 | 14.588 | 4.426 | 0.553 |

*Table 6. Execution times, speedup and efficiency for nmfPar compared to nmf with large data*

Tables 4, 5 and 6 show the execution times, speedup and efficiency for the nmfPar algorithm as compared to the nmf algorithm by number of threads for each data set size, and Figures 10 and 11 show the speedup and efficiency, respectively, for nmfPar using all data set sizes. Speedup was achieved with each data set size; however, the amount of speedup varied based on size of the data set. Using the small data set, nmfPar achieved a maximum speedup of 1.95 at three threads. With the medium data set, a maximum speedup of 3.4 was achieved at eight threads, and a speedup of 4.43 was achieved at seven threads using the large data set. These results show that greater performance improvements can be achieved as the data set size increases. Based on the results displayed in Figure 11, efficiency decreases steadily as more threads are utilized with a sharp decrease beyond three threads. Combining the results from Figures 10 and 11 show that when using the small data set, there is no reason to use more than three threads. For the medium and large data sets, performance improves with more threads and so the loss in efficiency is outweighed by the gains in performance.

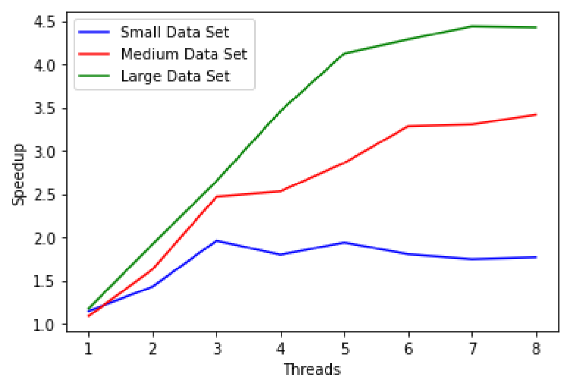


*Figure 10. Speedup for nmfPar compared to nmf with all data*

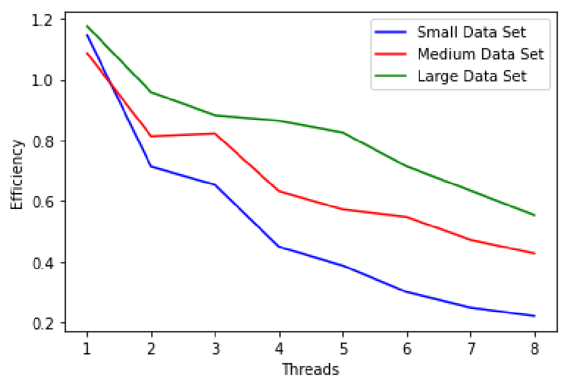
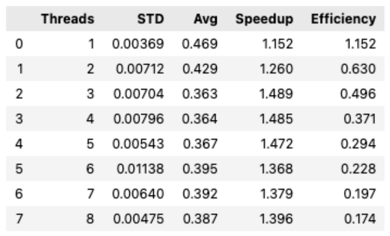


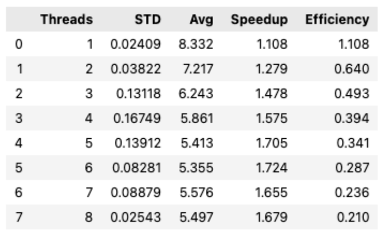
*Figure 11. Efficiency for nmfPar compared to nmf with all data*

## *nmfOptPar Results*



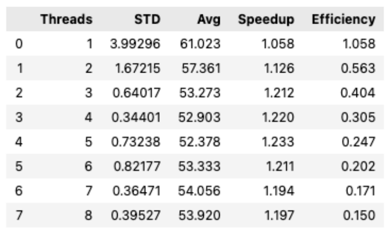
| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 0.00369 | 0.469 | 1.152 | 1.152 |
| 1 | 2 | 0.00712 | 0.429 | 1.260 | 0.630 |
| 2 | 3 | 0.00704 | 0.363 | 1.489 | 0.496 |
| 3 | 4 | 0.00796 | 0.364 | 1.485 | 0.371 |
| 4 | 5 | 0.00543 | 0.367 | 1.472 | 0.294 |
| 5 | 6 | 0.01138 | 0.395 | 1.368 | 0.228 |
| 6 | 7 | 0.00640 | 0.392 | 1.379 | 0.197 |
| 7 | 8 | 0.00475 | 0.387 | 1.396 | 0.174 |

*Table 7. Execution times, speedup and efficiency for nmfOptPar compared to nmf with small data*



| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 0.02409 | 8.332 | 1.108 | 1.108 |
| 1 | 2 | 0.03822 | 7.217 | 1.279 | 0.640 |
| 2 | 3 | 0.13118 | 6.243 | 1.478 | 0.493 |
| 3 | 4 | 0.16749 | 5.861 | 1.575 | 0.394 |
| 4 | 5 | 0.13912 | 5.413 | 1.705 | 0.341 |
| 5 | 6 | 0.08281 | 5.355 | 1.724 | 0.287 |
| 6 | 7 | 0.08879 | 5.576 | 1.655 | 0.236 |
| 7 | 8 | 0.02543 | 5.497 | 1.679 | 0.210 |

*Table 8. Execution times, speedup and efficiency for nmfOptPar compared to nmf with medium data*



| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 3.99296 | 61.023 | 1.058 | 1.058 |
| 1 | 2 | 1.67215 | 57.361 | 1.126 | 0.563 |
| 2 | 3 | 0.64017 | 53.273 | 1.212 | 0.404 |
| 3 | 4 | 0.34401 | 52.903 | 1.220 | 0.305 |
| 4 | 5 | 0.73238 | 52.378 | 1.233 | 0.247 |
| 5 | 6 | 0.82177 | 53.333 | 1.211 | 0.202 |
| 6 | 7 | 0.36471 | 54.056 | 1.194 | 0.171 |
| 7 | 8 | 0.39527 | 53.920 | 1.197 | 0.150 |

*Table 9. Execution times, speedup and efficiency for nmfOptPar compared to nmf with large data*

Tables 7, 8 and 9 show the execution times, speedup and efficiency for the nmfOptPar algorithm as compared to the nmf algorithm by number of threads for each data set size, and Figures 12 and 13 show the speedup and efficiency, respectively, for nmfOptPar using all data set sizes. For this set of experiments, the results from the nmfOpt experiment were integrated, and the optimal block sizes for each data set size was used, 50 for the small data set, 500 for the medium data set and 250 for the large data set. Similar to the nmfPar algorithm, speedup was achieved for each data set, but the speedup was much smaller for the nmfOptPar algorithm only achieving a speedup of 1.7 at six threads. These results reinforce those of the nmfOpt algorithm demonstrating that blocking is not beneficial for this algorithm and these data set sizes.

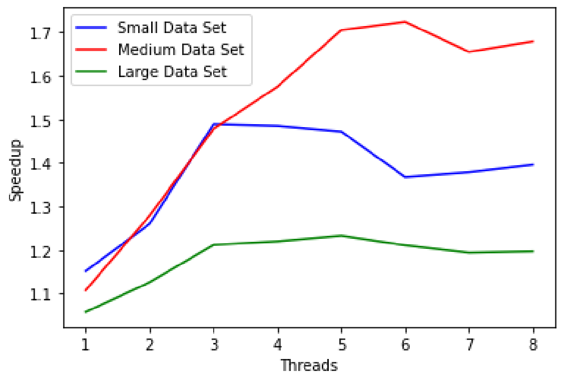


*Figure 12. Speedup for nmfOptPar compared to nmf with all data*

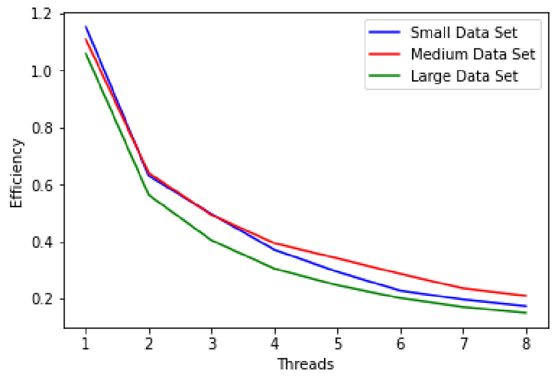


*Figure 13. Efficiency for nmfOptPar compared to nmf with all data*

## *nmfVec Results*

| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 0.00457 | 0.501 | 1.079 | 1.079 |
| 1 | 2 | 0.00197 | 0.408 | 1.324 | 0.662 |
| 2 | 3 | 0.00863 | 0.315 | 1.716 | 0.572 |
| 3 | 4 | 0.00997 | 0.302 | 1.789 | 0.447 |
| 4 | 5 | 0.00499 | 0.295 | 1.832 | 0.366 |
| 5 | 6 | 0.01690 | 0.307 | 1.760 | 0.293 |
| 6 | 7 | 0.01204 | 0.324 | 1.668 | 0.238 |
| 7 | 8 | 0.00946 | 0.319 | 1.694 | 0.212 |

*Table 10. Execution times, speedup and efficiency for nmfVec compared to nmf with small data*

| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 0.03602 | 8.326 | 1.109 | 1.109 |
| 1 | 2 | 0.04725 | 5.721 | 1.613 | 0.806 |
| 2 | 3 | 0.20256 | 3.979 | 2.320 | 0.773 |
| 3 | 4 | 0.11617 | 3.504 | 2.634 | 0.658 |
| 4 | 5 | 0.11263 | 3.372 | 2.737 | 0.547 |
| 5 | 6 | 0.05121 | 3.037 | 3.039 | 0.507 |
| 6 | 7 | 0.05089 | 2.834 | 3.257 | 0.465 |
| 7 | 8 | 0.12426 | 2.814 | 3.280 | 0.410 |

*Table 11. Execution times, speedup and efficiency for nmfVec compared to nmf with medium data*

| | Threads | STD | Avg | Speedup | Efficiency |
|---|---|---|---|---|---|
| 0 | 1 | 4.25070 | 60.512 | 1.067 | 1.067 |
| 1 | 2 | 0.42588 | 34.250 | 1.885 | 0.942 |
| 2 | 3 | 0.99105 | 24.853 | 2.598 | 0.866 |
| 3 | 4 | 0.76870 | 19.166 | 3.369 | 0.842 |
| 4 | 5 | 0.59361 | 14.937 | 4.322 | 0.864 |
| 5 | 6 | 0.40981 | 14.771 | 4.371 | 0.728 |
| 6 | 7 | 0.37407 | 14.763 | 4.373 | 0.625 |
| 7 | 8 | 0.05879 | 15.105 | 4.274 | 0.534 |

*Table 12. Execution times, speedup and efficiency for nmfVec compared to nmf with large data*

Tables 10, 11 and 12 show the execution times, speedup and efficiency for the nmfVec algorithm as compared to the nmf algorithm by number of threads for each data set size, and Figures 14 and 15 show the speedup and efficiency, respectively, for nmfVec using all data set sizes. Similar to the other parallelized algorithms, speedup was achieved with the greatest speedup when using the large data set. The nmfVec algorithm did achieve a greater speedup than either of the other two parallelized algorithms at 4.37, but this is much lower than expected when using vectorization. These results in combination with the roofline charts suggest that there is more room for improvement with vectorization of this algorithm.

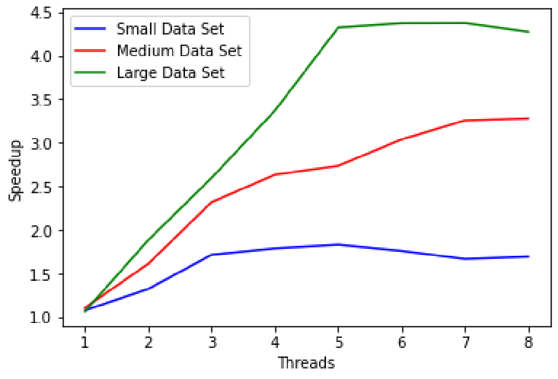


*Figure 14. Speedup for nmfVec compared to nmf with all data*

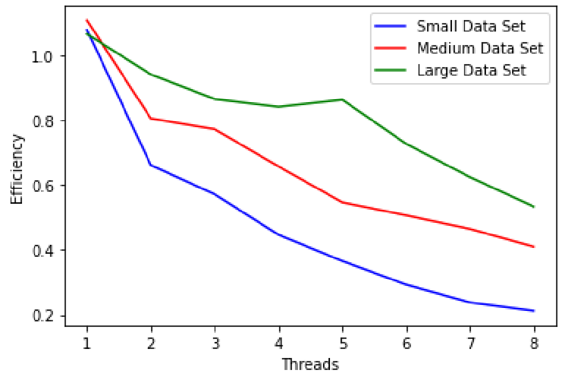


*Figure 15. Efficiency for nmfVec compared to nmf with all data*

## *pythonNMF Results*

| Problem Size | Avg Time (sec) | GFLOPs |
|---|---|---|
| Small | 0.50113 | 1.62688 |
| Medium | 3.67393 | 4.20979 |
| Large | 17.4842 | 4.86172 |

*Table 13. Execution times and GFLOPs for pythonNMF as compared to nmf*

Table 13 shows the executions times and GFLOPs for pythonNMF. Figures 16, 17 and 18 show the executions times for pythonNMF compared to each of the other algorithms. Figure 19 shows a view of execution times of pythonNMF compared to only the top two performing algorithms. Figures 16, 17 and 18 show that as the data set size increases, pythonNMF does comparatively better against the other algorithms. Although the performance of pythonNMF improves with increasing data set size, it does not outperform either nmfPar or nmfVec. This suggests that while pythonNMF utilizes optimized C code, the ability to adjust the program directly in C for this specific task leads to lower execution times. It is important to note that while nmfPar and nmfVec outperformed pythonNMF, it took much less time and knowledge to write pythonNMF than the others. This means that when the value of the ease of use of Python is weighed against the performance cost, the tradeoff of using Python might be reasonable.

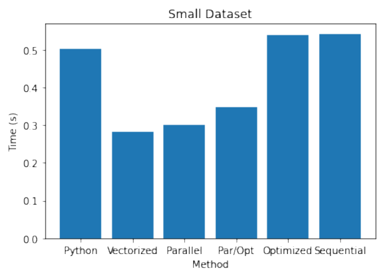


*Figure 16. Execution times for algorithm variants for small data*

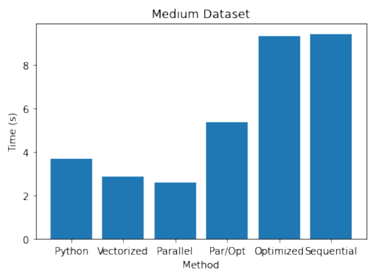


*Figure 17. Execution times for algorithm variants for medium data*

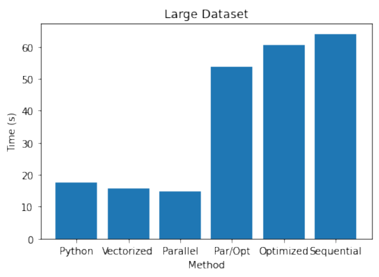


*Figure 18. Execution times for algorithm variants for large data*

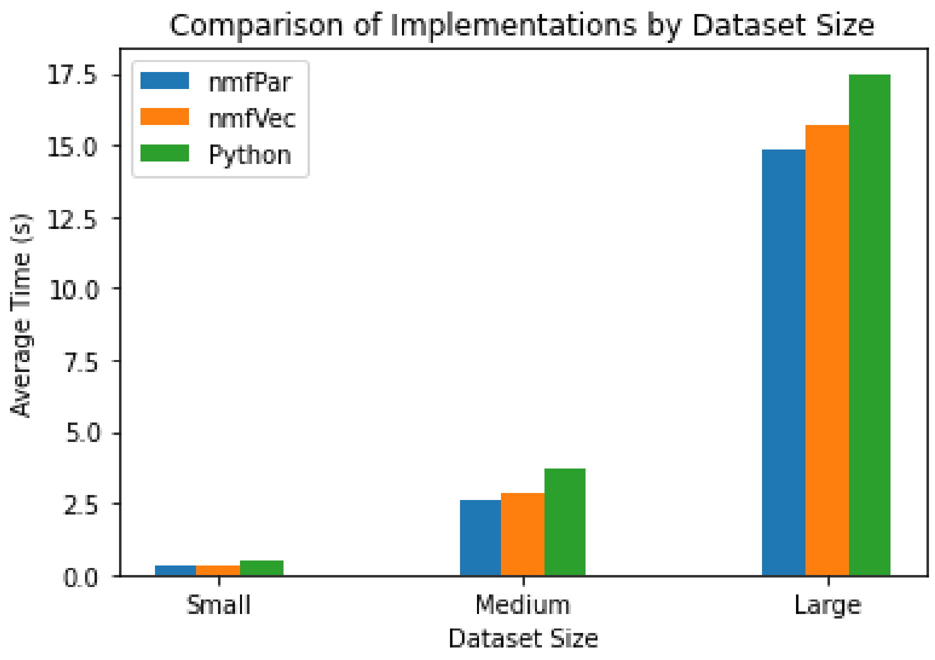


*Figure 19. Execution times for pythonNMF compared to nmfPar and nmfVec for all data set sizes*

## References

[1] Eunus, S. I. (2022). What is Non-Negative Matrix Factorization (NMF). Retrieved from https://medium.com/codex/what-is-non-negative-matrix-factorization-nmf-32663fb4d65 on March 16th, 2023.
[2] Févotte, C., Bertin, N., & Durrieu, J. L. (2009). Nonnegative Matrix Factorization with the Itakura-Saito Divergence: With Application to Music Analysis. *Neural Computation*, *21*(3), 793-830.
[3] Wikipedia. (2018). Itakura-Saito Distance. Retrieved from https://en.wikipedia.org/wiki/Itakura%E2%80%93Saito_distance on March 15th, 2023.
[4] OpenMP. (2018). OpenMP API Specification: Version 5.0 November 2018: 2.9.3 SIMD Directives. Retrieved from https://www.openmp.org/spec-html/5.0/openmpsu42.html on May 2nd, 2023.
[5] NumPy. (2022). What is NumPy? Retrieved from https://numpy.org/doc/stable/user/whatisnumpy.html on May 3rd, 2023.