Nonnegative Matrix Factorisation

Dan Jacobellis

Tyler Masthay

CSE 392 - Parallel Algorithms

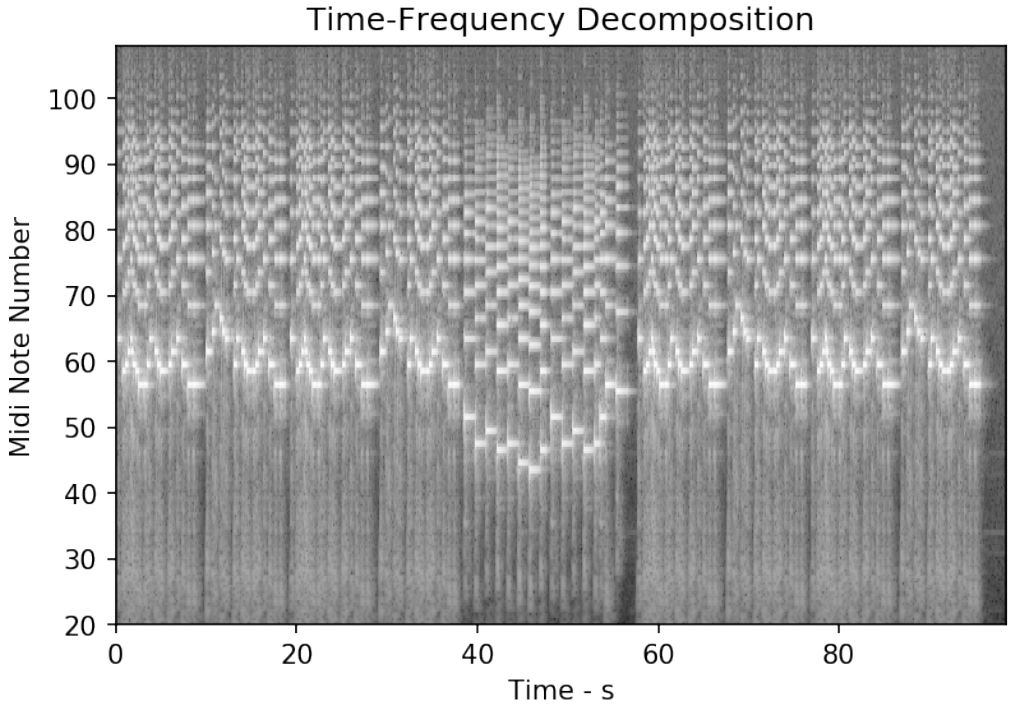
Project Proposal

Introduction

Suppose we have a matrix $\mathbf{V}$ containing nonnegative data; for example, the magnitude image of a time-frequency decomosition of an audio recording.

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
from process_audio import *
from visualizations import *
V,t,midi_nn = prep_for_nmf('tetrisA_mono.wav');
_ = plt.figure(dpi = 150, figsize=(6,4));
time_freq_image(V);
```



The problem of nonnegative matrix factorisation (NMF) amounts to factorising $\mathbf{V}$ into two factors $\mathbf{W}$ and $\mathbf{H}$ which are also nonnegative. That is,

$$\mathbf{V} \approx \hat{\mathbf{V}} = \mathbf{WH}$$

This technique can be used to learn recurring patterns in the data matrix. In this case, $\mathbf{W}$ represents a learned dictionary and the $\mathbf{H}$ represent represents the decomposition. When applied to the time-frequency decomposition of a music recording, $\mathbf{W}$ contain the learned spectral envelopes of each instrument in the recording and $\mathbf{H}$ contains a transcription of the music.

Many variations of NMF algorithms are well-established, and several libraries are available, such as `decomposition.nmf()` in scikit-learn. For the class project, We propose implementing parallel versions of these algorithms:

- To learn about NMF algorithms, which are currently an open field of research
- To gain experience and intuition for different parallel programming models by implimenting NMF using shared memory, message passing, and GPU programming
- To learn about implementation of iterative algorithms with high data parallelism
- To improve upon the performance and functionality of existing NMF libraries

Algorithmic Approach

The most widely used algorithms for NMF employ a multiplicative weight update method based on the pioneering work of Lee and Sung [1]. The algorithm consists of the following steps:

- Initialize $\mathbf{W}$ and $\mathbf{H}$ with non-negative values
- Iteratively update $\mathbf{W}$ and $\mathbf{H}$ using the following rules: (n is the iteration)

$$\mathbf{H}^{n+1}_{[i,j]} \leftarrow \frac{\left((\mathbf{W}^n)^\top \mathbf{V}\right)_{[i,j]}}{\left((\mathbf{W}^n)^\top \mathbf{W}^n \mathbf{H}^n\right)_{[i,j]}}$$

$$\mathbf{W}^{n+1}_{[i,j]} \leftarrow \frac{\left(\mathbf{V}(\mathbf{H}^{n+1})^\top\right)_{[i,j]}}{\left(\mathbf{W}^n \mathbf{H}^{n+1}(\mathbf{H}^{n+1})^\top\right)_{[i,j]}}$$

As a result, the sequential complexity is $O(n^2 m)$, where n will be fixed at 264 and m will scale with the length of the recording.

Resources Needed

We do not have an precise estimates of the resources needed except that we plan to perform dense linear algebra with CUDA on matricies with (10^6) elements.

Deliverables

We plan to test and report the performance on recordings ranging in length from 90 seconds to 10 minutes. We will compare the perfomance primarily with the scikit-learn library described in the final section. We will test the peformance on spectrograms of both monophonic and polyphonic recordings, which will affect the sparsity of the resulting matricies.

Work Description

The tasks we have identified are:

- Implement the sequential algorithm as described by Lee and Sung [1].
- Set up a boilerplate CUDA environment and code for performing linear algebra and make sure we both know how to compile with simple linear alebra programs.
- Benchmark the sequential algorithm on different problem sizes and inputs and compare to the scikit-learn performance.
- At this point, we will use what we have learned to make a plan to transition to a parallel implementation.

Initial Performance Benchmarks

A test of the scikit-learn `decomposition.nmf()` implementation using the default parameters on a 90 second, single instrument audio recording provides a starting benchmarking for the algorithm. There are a few performance characterstics to note:

- On a workstation with twelve processors, one processor is fully utilized. The remaining processors are utilized at approximately 25%.
- The wall-clock run-time on the 90 second recording is approximately 2.5 hours.
- Each iteration takes about 7.0 seconds
- The number of iterations required scales rapidly as the converge tolerance is lowered.

In [3]:

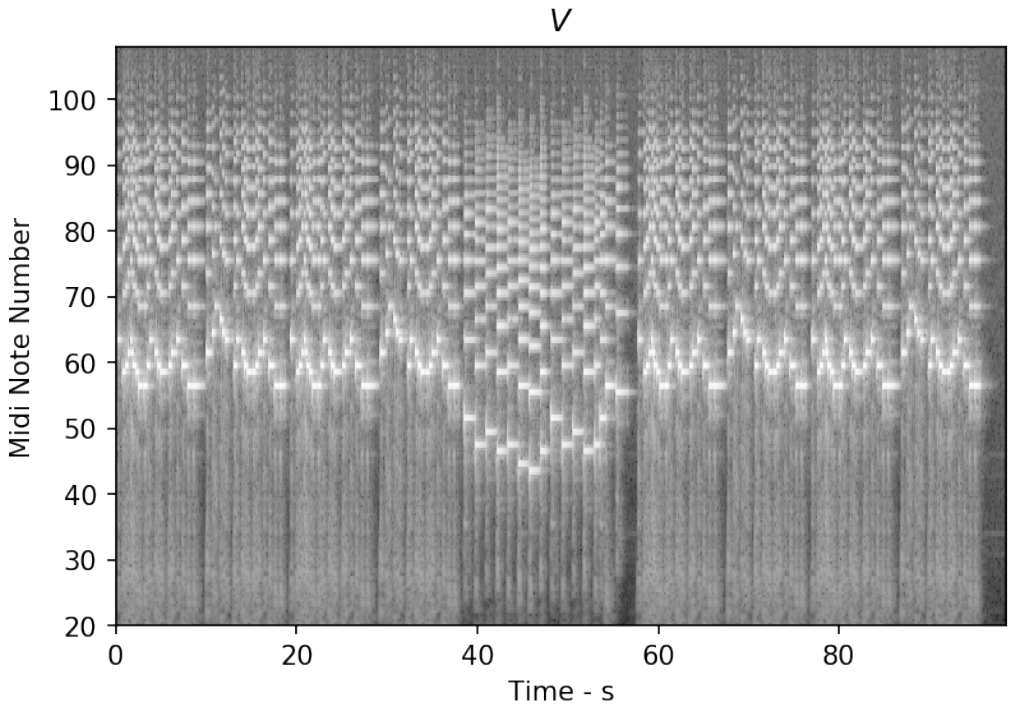
```
import sklearn.decomposition
import time
model = sklearn.decomposition.NMF(n_components=264, max_iter=20, tol = 1e-4)
t1 = time.time()
W = model.fit_transform(V)
```

```
H = model.components_
t2 = time.time()
print(t2-t1)
```

```
13.537438154220581
```

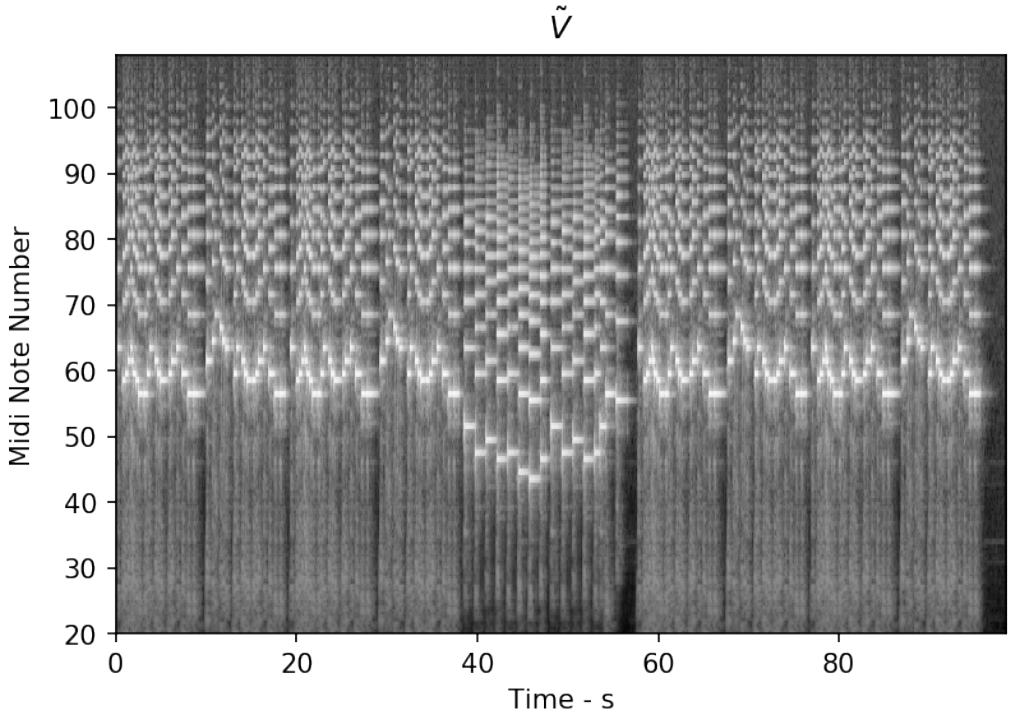
In [4]:

```
_ = plt.figure(dpi = 150, figsize=(6,4));
time_freq_image(V,'$V$');
```



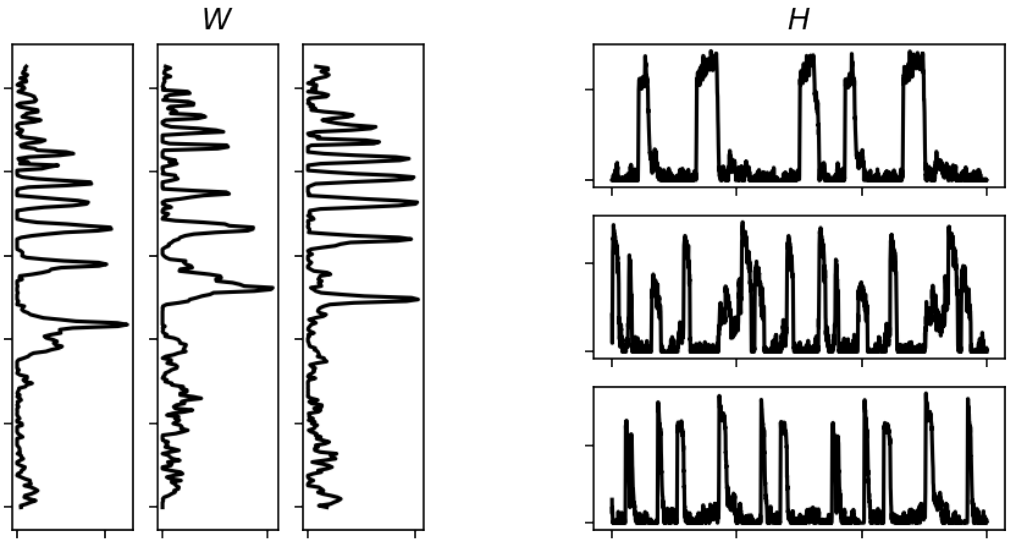
In [5]:

```
_ = plt.figure(dpi = 150, figsize=(6,4));
time_freq_image(np.matmul(W,H),'$\~{V}$');
```



In [11]:

```
_ = plt.figure(dpi = 150, figsize=(7,3.5));
transcription(W,H,num_comp=3,trunc=3000)
```



Existing Parallel Implementations

Most existsing implementations are targeted towards bioinformatics applications. As such, the algorithms are designed primarily to scale with problem size.

However, for application to audio source separation and music transcription, the problem size is usually fixed at some moderate size and the primary concern is speed. As such, we expect that the existing implementations are not ideal for our application.

References

[1] Lee, D.D., Seung, H.S., 2001. Algorithms for Non-negative Matrix Factorization, in: Advances in Neural Information Processing Systems 13. MIT Press, pp. 556–562.

[2] S. Makino, Ed., Audio source separation. New York, NY: Springer Berlin Heidelberg, 2018.

[3] E. Vincent, T. Virtanen, and S. Gannot, Audio source separation and speech enhancement. 2018.