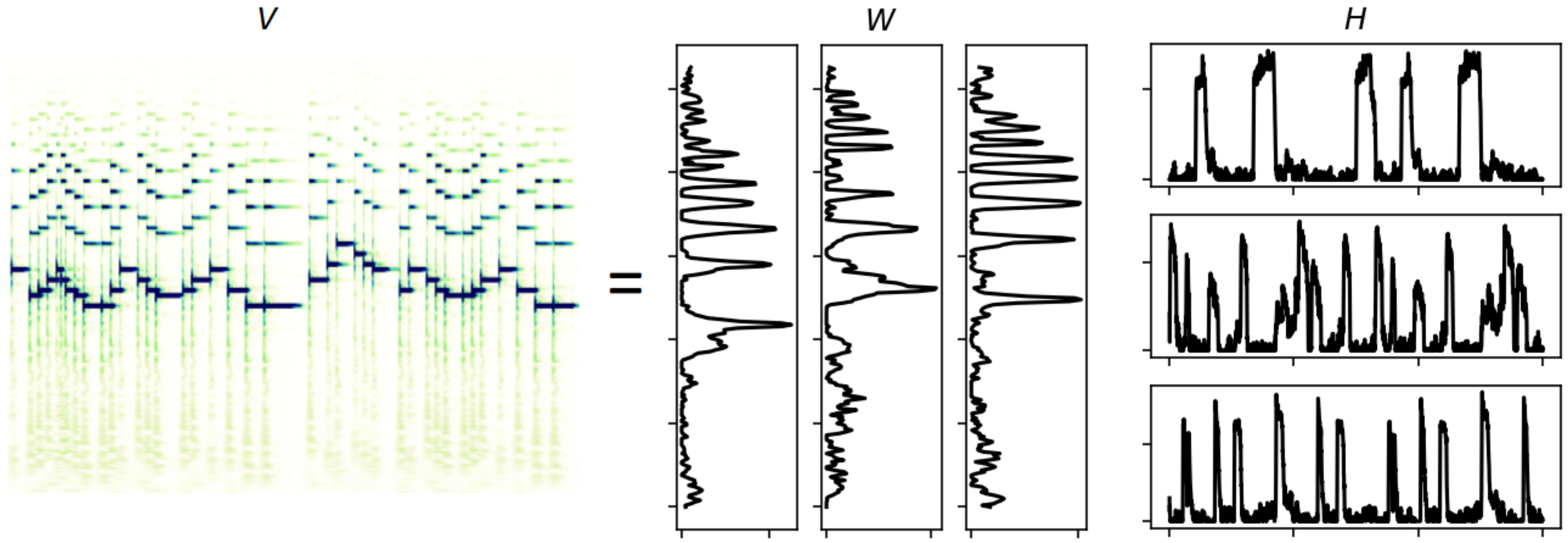
Nonnegative Matrix Factorisation for Audio Applications

Dan Jacobellis, Tyler Masthay



Nonnegative Matrix Factorisation

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

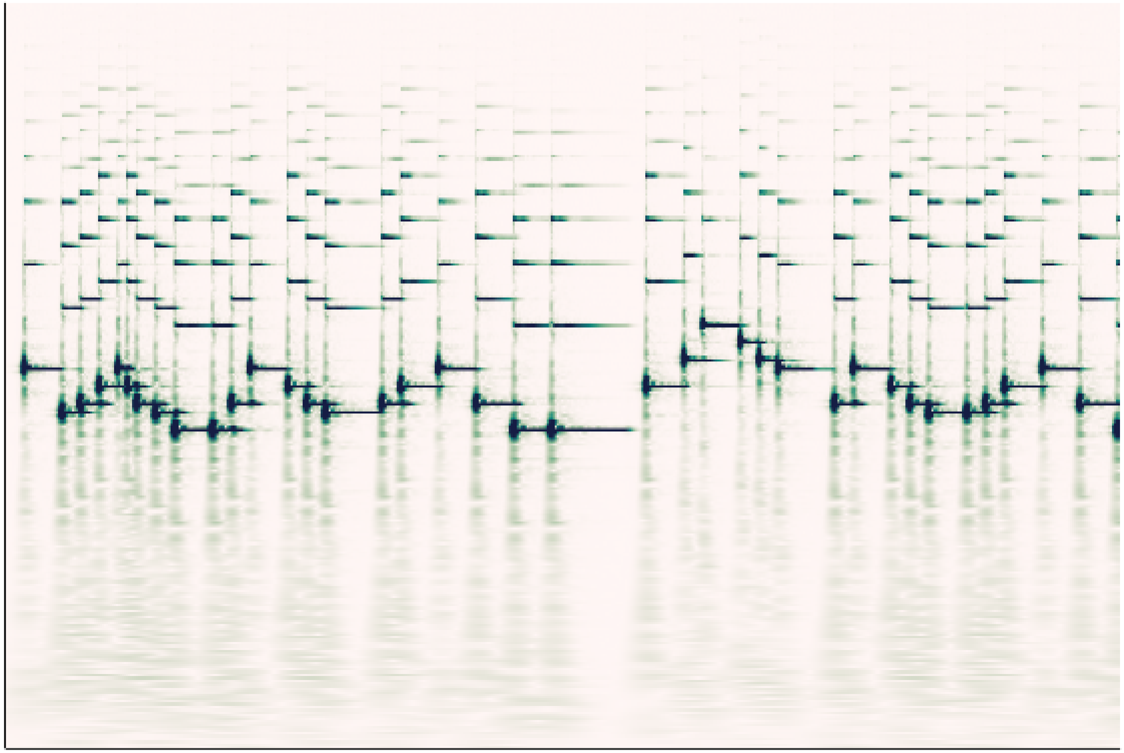
$$\mathbf{W} \in \mathbb{R}^{\mathbf{m} \times \mathbf{k}} \quad \mathbf{W} \geq \mathbf{0}$$

$$\mathbf{H} \in \mathbb{R}^{\mathbf{k} \times \mathbf{n}} \quad \mathbf{H} \geq \mathbf{0}$$

- Unsupervised learning (think SVD)
- Number of components k is a parameter
- $\mathbf{V}$ is $m \times n$
- Setting $k < m$ is a form of lossy compression

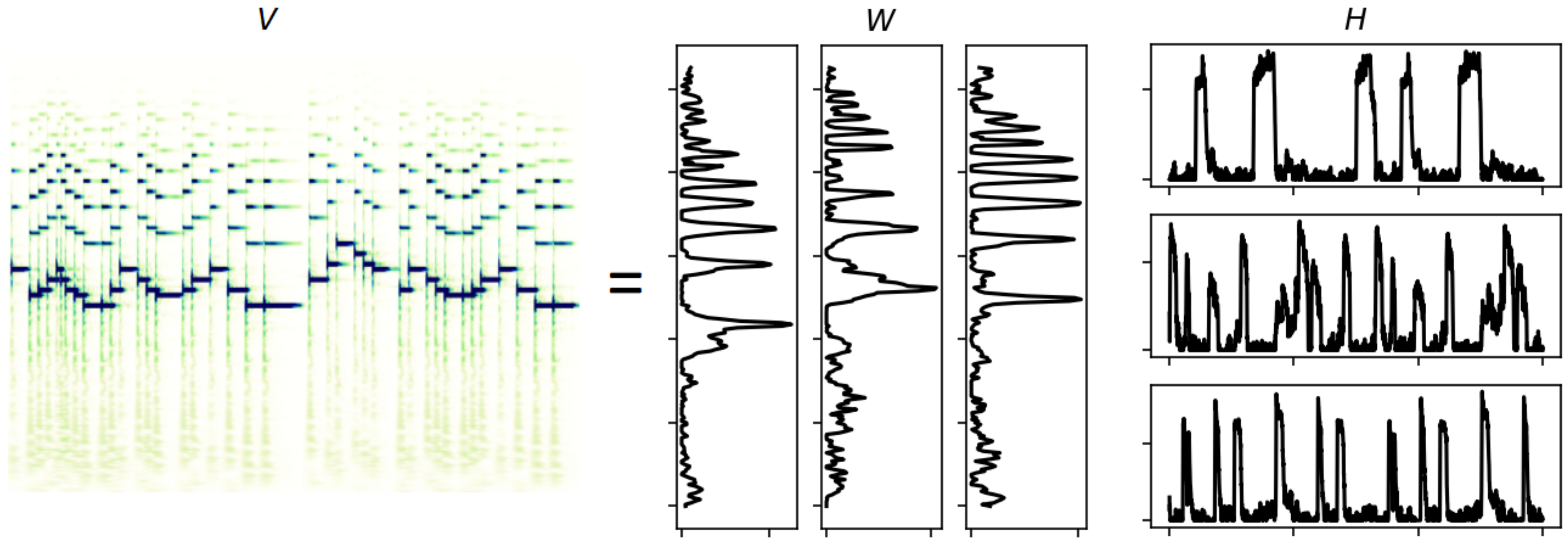
Audio Example

$$\mathbf{V} = |\mathbf{CQT}(\mathbf{x})| =$$



Time-frequency representation of 'Korobeiniki' played on piano. The frequency is on a logarithmic scale.

Audio Example



Algorithm: Multiplicative Update

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

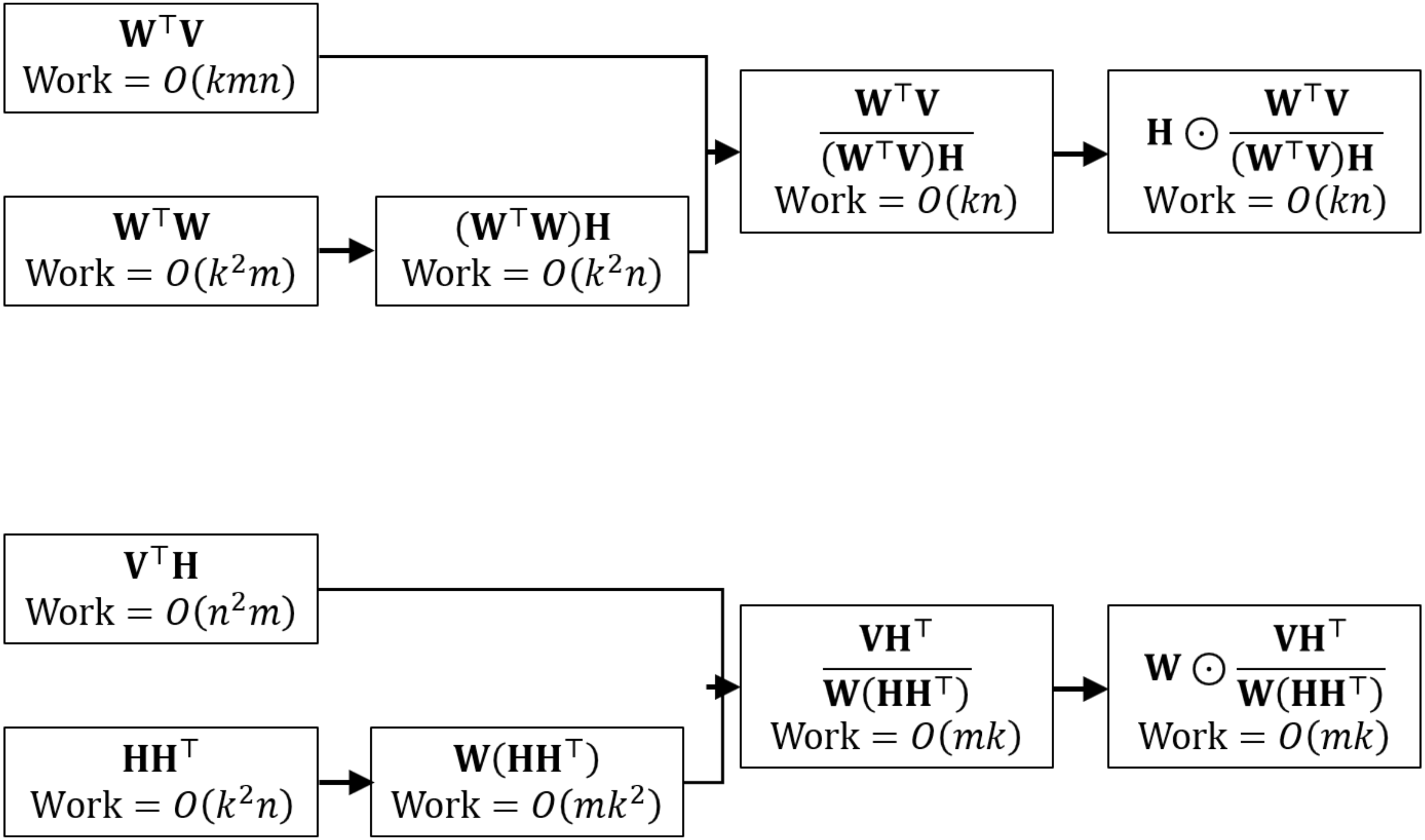
$$\mathbf{H}^{n+1}_{[i,j]} \leftarrow \mathbf{H}^{n}_{[i,j]} \odot \frac{\left((\mathbf{W}^n)^\top \mathbf{V}\right)_{[i,j]}}{\left((\mathbf{W}^n)^\top \mathbf{W}^n \mathbf{H}^n\right)_{[i,j]}}$$

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

$\odot$ and division are element-wise.

[2] 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.

Algorithm: Multiplicative Update



Parallelization on GPU : Motivation

- Matricies remain stationary in memory
- Matrix multiplies have high computational intensity compared to memory
- Single precision is sufficient
- Would like to use consumer hardware

Name	Clock(MHz)	GFLOPS(FP32)
Adreno 616	750	384
Adreno 630	710	727
Adreno 640	585	899

GPU Implementation

- Used Julia bindings to CUDA (CURAND, CUBLAS)
- Compiler allows easily mapping high level syntax to GPU
- Compiler finds best way to send data back and forth between device and host

```julia
a = CuArray([1., 2., 3.])
function apply(op, a)
   i = threadIdx().x
   a[i] = op(a[i])
return
end
@cuda threads=length(a) apply(x->x^2, a)


julia> a
3-element CuArray{Float32,1}:
1.0
4.0
9.0
```

```
@device_code_ptx @cuda apply(x->x^2, a)
apply(.param .b8 a[16])
{
        ld.param.u64    %rd1, [a+8];
        mov.u32         %r1, %tid.x;

        // index calculation
        mul.wide.u32    %rd2, %r1, 4;
        add.s64         %rd3, %rd1, %rd2;
        cvta.to.global.u64      %rd4, %rd3;

        ld.global.f32   %f1, [%rd4];
        mul.f32         %f2, %f1, %f1;
        st.global.f32   [%rd4], %f2;

        ret;
}
```

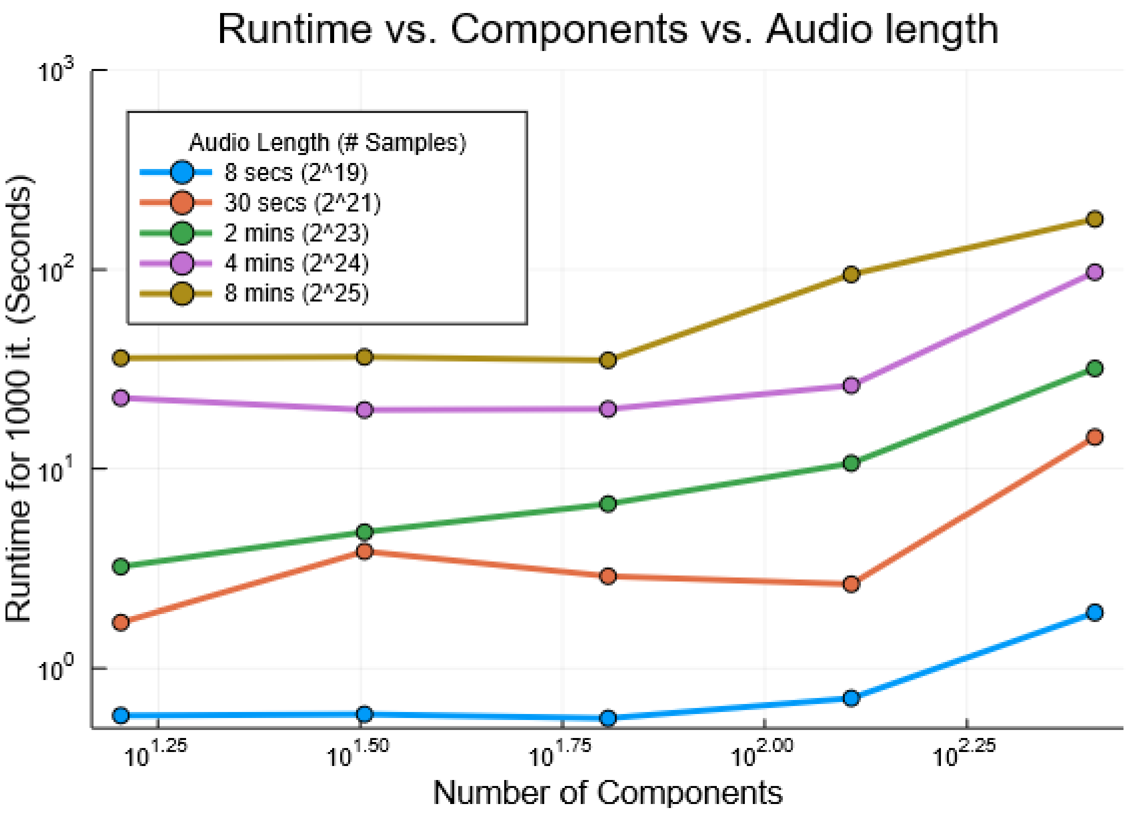
[2] https://github.com/JuliaGPU/CUDAnative.jl (https://github.com/JuliaGPU/CUDAnative.jl)

GPU Implementation

- Must be careful and explicit with types and constants
- Otherwise, compiler will think you want to move all of the data off of the GPU and back

```
API calls:   91.92% 11.2821s    40 282.05ms  26.070us 1.18052s  cuMemcpyDtoH
              2.23% 273.60ms     3 91.198ms  692ns    273.59ms  cuDevicePrimaryCtxRelease
              2.16% 264.86ms     6 44.143ms  682ns    185.30ms  cudaFree
              0.91% 111.74ms   172 649.66us  2.9950us 9.1952ms  cuModuleUnload
              0.86% 105.85ms     1 105.85ms  105.85ms 105.85ms  cuDevicePrimaryCtxRetain
              0.74% 90.441ms 1346  67.192us 3.4970us 460.51us  cuMemAlloc
              0.25% 30.511ms 3856  7.9120us 5.2800us 246.57us  cudaLaunchKernel
```

GPU Performance



- Runtime is relatively flat as the number of components increases up to a point
- We can't fit everying in GPU memory after the number of components reaches ~128
- After this point, we are forced to send data between CPU and GPU *every* iteration
- Fortunately, the behavior as a function of the audio length is well-behaved