Bayesian Network Inference with Cuda

GPU Architecture

Marc Haubenstock (1525175)

June 29, 2016

1 Hidden Markov Model(HMM)

1.1 Overview

$S = s_1 s_2 \cdots s_N$	a set of N states.
$V = v_1 v_2 \cdots v_{\|V\|}$	a set of distinct observations symbols. $\|V\|$ denotes the number of distinct observations.
$O = o_1 o_2 \cdots o_T$	a sequence of T observations; each drawn from the observation symbol set V.
$Q = q_1 q_2 \cdots q_T$	a sequence of state; q_t denotes the state at time t.
$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & & & \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix}$	a transition probability matrix A; each a_{ij} representing the probability of moving from state s_i to state s_j, i.e. $a_{ij} = P(q_{t+1} = s_j \mid q_t = s_i)$
$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1\|V\|} \\ b_{21} & b_{22} & \cdots & b_{2\|V\|} \\ \vdots & & & \\ b_{N1} & b_{N2} & \cdots & b_{N\|V\|} \end{bmatrix}$	an emission probability matrix B, each b_{ij} representing the probability of the observation v_j being generated from a state s_i, i.e. $b_{ij} = P(o_t = v_j \mid q_t = s_i)$
$\pi = [\pi_1, \pi_2, \cdots, \pi_N]$	an initial state distribution: $\pi_i = P(q_1 = S_i)$

Figure 1: The notation of a HMM [3]

1.2 Forward

```
FORWARD(O)
1  initialize all cells of α to 0
2  α(o_1, s) ← 1
3  for t = o_2 to o_T
4        do for i = 1 to N
5              do for j = 1 to N
6                    do p ← a_ji · b_it
7                       α(t, i) ← α(t, i) + α(t − 1, j) · p
8  likelihood ← α(o_T, e)
9  return likelihood
```

Figure 2: Pesudo-Code for the Forward Algorithm [3]

The algorithm produces a data structure called a trellis, denoted by α of size $N \times T$ for a sequence of T observations of size N. The trellis can be viewed as a matrix where each cell $\alpha(t, i)$ holds the probability of being in state i while having seen the observations until t. The complexity of the algorithm is due to the for loops. For M observation sequences the complexity of the algorithm is $\mathcal{O}(MN^2T)$. Due to the dependency on a previous value as shown line 7, the outer most for loop can not be parallised. Instead, we can exploit the fact that all our M observation sequences are independent, this can be computed in parallel, reduced the complexity to $\mathcal{O}(cMT)$, where c is the execution time of the parallel code [3].

1.3 Viterbi

```
VITERBI(O)
 1  initialize all cells of α to 0
 2  α(o_1, s) ← 1
 3  for t = o_2 to o_T
 4        do for i = 1 to N
 5              do for j = 1 to N
 6                    do p ← a_ji · b_it
 7                       if α(t − 1, j) · p > α(t, i)
 8                          then α(t, i) ← α(t − 1, j) · p
 9                               backpointers(t, i) ← j
10  states = BACKTRACE(backpointers)
11  return states
```

Figure 3: Pesudo-Code for the Viterbi Algorithm [3]

1.4 Baum-Welch

```
FORWAD-BACKWARD-EXPECTATION(O)
 1  initialize all cells of α, β, γ, ξ to 0
 2  likelihood ← FORWARD(O)
 3  β(o_T, e) ← 1
 4  for t = o_T to o_1
 5        do for i = 1 to N
 6              do γ(t,i) ← γ(t,i) + (α((t,i) · β(t,i)/ likelihood)
 7                 ξ(i) ← ξ(i) + (α((t,i) · β(t,i)/ likelihood)
 8                 for j = 1 to N
 9                       do p ← α_ji · b_it
10                          β(t−1, i−1) ← β(t−1, i−1) + β(t,i) · p
11                          ξ(j,i) ← ξ(j,i) + (α(t−1, j)β(t,i) · p/ likelihood)
12
```

Figure 4: Pesudo-Code for the Backward Forward Algorithm [3]

The general structure of the Baum-Welch algorithm is similar to its forward counterpart. Most notably instead of iterating from the start of the observation sequence, the algorithm starts at the last observation, moving backwards through the trellis. The parallisation can be achieved in a similar manner to the forward algorithm, but due to the change in indexing for the trellis α in line 11 this can not be repesented as matrix multiplcation [3].

The idea of this algorithm is to (randomly) initialise the A and B matrices, perform the forward, then the backward pass on them. Finally with the computed values we can re-estimate the matrices and repeat the process.

Sadly numerous errors in the pseudo code made the implementation more difficult than presented. Specifically in line 9, the probability p is produced from the transition matrix A not the trellis, as is evident from the inconsistent indexing at α. Moreover, at line 10. the β values is not dependent on $i-1$ but rather i [2].

$$\hat{a}_{ij} = \frac{\xi(i,j)}{\xi(i)}$$
$$\hat{b}_{jt} = \frac{\gamma(j,t)}{\xi(j)}$$

Figure 5: Estimation of the A and B Matricies [3]

Similarly the estimated values of the emission matrix B is not dependent on $\epsilon(j)$, but

should be on γ, as it is defined in literature [2].

2 Implementation

2.1 Overview

As seen in the previous section it is not possible to fully parallize the individual algorithms due to their dependence on a value of a previous time t. However, hmms are usually computed with many observation sequences $\mathbf{M}$. Each algorithm acts on one such sequence. The main idea presented in [3] exploits this by parallizing over $\mathbf{M}$ instead of T.

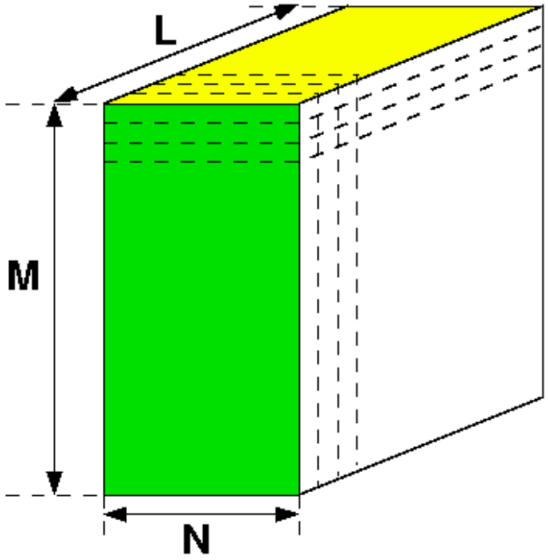

Figure 6: Graphical representation of the data structure presented in *A Cuda Implementation of Hidden Markov Models*[3]

Thus, the for-loop of the T domain remains in the code while the CUDA kernel computes all M slices at time t in parallel. Graphically this can be seen as moving through the 3D data structure either front-to-back, in the forward case, or back-to-front in the backward case. The current element is being computed by applying the following formulae.

$$D_{ij} = B_{O_i} . * C_i \times A_j$$

Where B_{O_i} is the row of the emission matrix at observation O_i, C_i is the previous slice and A_j is the j$^{\text{th}}$ column of the transition. matrix A.

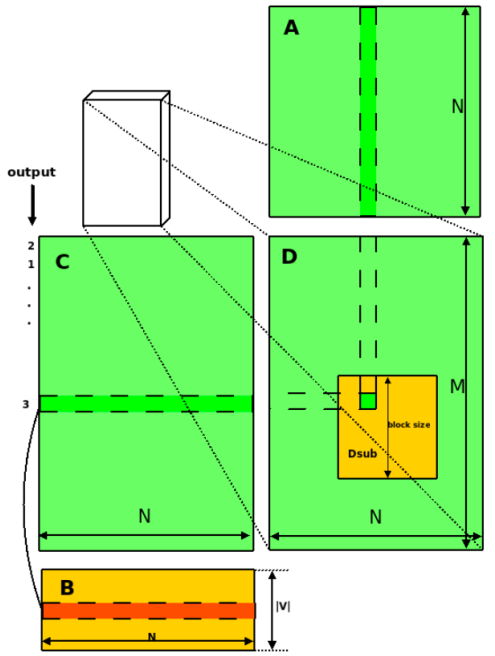

Figure 7: The computation of a slice[3]

Finally a stipulation of the outlined method is that:

- The number of states and sequences must be a multiple of block size 16
- The number of output sequences must be of the same length.

2.2 Forward

```
initTrellis<<<M,N>>>(..);

for(int t = 0; t < T; t++){
...

 // copute B_o
computeBRow<<<M,N>>>
(M_noOfObsSequences, V_noOfObsSymbols, T_noOfObservations,
N_noOfStates, dev_O_obsSequence_2D, dev_B_obsEmissionProbs_2D, i, dev_B);

cudaDeviceSynchronize();

 // W = B_o .* C_i
pointwiseMul<<<M,N>>>
(dev_W, dev_B, &dev_3D_Trellis[(i - 1) * M_noOfObsSequences * N_noOfStates]);
```

```
cudaDeviceSynchronize();

// wrapper function for cublas multiply; D = W x A
cublasDeviceMultiply(M_noOfObsSequences, N_noOfStates, N_noOfStates, dev_W,
dev_A_stateTransProbs_2D, &dev_3D_Trellis[i * M_noOfObsSequences * N_noOfStates]);


cudaDeviceSynchronize();

...


}

double *last_slice
= &dev_3D_Trellis[(T_noOfObservations - 1) * M_noOfObsSequences * N_noOfStates];

// compute likelihood
for (int i = 0; i < M_noOfObsSequences; i++)
{

int smBytes = 64 * sizeof(double);
int grid = N_noOfStates / 64;
reduce_1 <<< grid, 64, smBytes >>>(&last_slice[i*N_noOfStates], dev_A_odata);

memcpyVector(host_A_odata, dev_A_odata, N_noOfStates, cudaMemcpyDeviceToHost);

host_likelihoods_1D[i] = host_A_odata[0] + host_A_odata[1];

}
```

The implementation follows the pseudo-code outlined by Liang. First we compute the slices of our 3D data structure by looping over T. Instead of only computing a single value as outlined by the eqution, we compute all values for a slice at once.Once we computed all values, we compute the likelihood for each observation sequence. No pseudo-code was given, however, by looking at literature we know we have to compute this as a summation of all terms for a single observation sequence [2]. We implementing this summation as a reduction in N. The reduction uses the simple interleaved model and is outlines as the first method in *Optimizing Parallel Reduction in CUDA* [1].

2.3 Viterbi

2.4 Baum-Welch

```
initMatrix<<<N_noOfStates,V_noOfObsSymbols>>>(dev_B_prime_3D,M_noOfObsSequences);
initMatrix<<<N_noOfStates,N_noOfStates>>>(dev_A_prime_3D, M_noOfObsSequences);

// overwrite the "true" transfer and emission matricies with the random ones
copyMatrix(dev_A_prime_3D, host_A_stateTransProbs_2D,
 N_noOfStates, N_noOfStates, M_noOfObsSequences);
copyMatrix(dev_B_prime_3D, host_B_obsEmissionProbs_2D,
N_noOfStates, V_noOfObsSymbols, M_noOfObsSequences);

initBeta<<<M_noOfObsSequences, N_noOfStates>>>(dev_beta_3D, V_noOfObsSymbols);

ForwardAlgorithmSet(host_Pi_startProbs_1D, host_A_stateTransProbs_2D,
host_B_obsEmissionProbs_2D,host_O_obsSequences_2D, N_noOfStates,
V_noOfObsSymbols, T_noOfObservations, M_noOfObsSequences,host_likelihoods_1D...);
```

The matrices and β values are initialized and copied into host memory for the forward algorithm.

```
for (int t = T_noOfObservations - 1; t >= 0; t--) {

UpdateGammaGPU << <M_noOfObsSequences, N_noOfStates >> >
(dev_gamma_3D, dev_beta_3D, dev_3D_Trellis_Alpha, t, dev_likelihood, V_noOfObsSymbols);

cudaDeviceSynchronize();

UpdateEpsilonReductionErrorGPU <<<M_noOfObsSequences, N_noOfStates >>>
(epsilon_reduction_grid_error, dev_beta_3D, dev_3D_Trellis_Alpha, t, dev_likelihood);

if (t > 0){

// Computes the B matrix in term D = B .* C x A
ComputeBDevice << < M_noOfObsSequences, N_noOfStates >> >
(M_noOfObsSequences, V_noOfObsSymbols, T_noOfObservations,
N_noOfStates, dev_O_obsSequences_2D, dev_B_obsEmissionProbs_2D, t, dev_B);
cudaDeviceSynchronize();
```

```
// All dimensions are multipe of Warp Sizes
// Compute W =  B .* C i.e. beta(t,i) * b_it
pointwiseMatrixMul << <M_noOfObsSequences, N_noOfStates >> >
(dev_W, dev_B, &dev_beta_3D[t * M_noOfObsSequences * N_noOfStates]);

cudaDeviceSynchronize();

// Compute D = W x A, D = beta(t,i)*p
cublasMultiplyDouble(M_noOfObsSequences,
N_noOfStates, N_noOfStates, dev_W, dev_A_stateTransProbs_2D, dev_D);

cudaDeviceSynchronize();

updateBeta << <M_noOfObsSequences, N_noOfStates >> >
(dev_beta_3D, dev_D, t, V_noOfObsSymbols);

cudaDeviceSynchronize();


for (int j = 0; j < N_noOfStates; j++){
UpdateEpsilonGPU <<<M_noOfObsSequences, N_noOfStates >>>
(dev_epsilon_3D, dev_beta_3D,
dev_3D_Trellis_Alpha, t, dev_likelihood, j, dev_D);

cudaStatus = cudaDeviceSynchronize();
} // end of for j

} // end of if

} // end of for t
```

The values presented in the pseudo code (Figure 3) at lines 6 and 7 can be computed easily as CUDA kernels. Similarly the values of β follow a similar procedure as the α values in the forward algorithm, and can be computed accordingly. The ϵ values on the other hand depend on 3 variables i, j, m. Because of this, we opted for a simple approach and iterate over N, to compute this value, increasing the complexity to $\mathcal{O}(cMNT)$.

```
// Estimate Matricies -  // sum up all values and reductions then divide

ColumReduction_Height <<<1, N_noOfStates >>>
(epsilon_reduction_grid_error, M_noOfObsSequences);
```

$$\hat{b}_j(v_k) = \frac{\text{expected number of times in state j observing symbol } v_k}{\text{expected number of times in state j}}$$

Figure 8: Definition of b hat taken form [2]

```
ColumReduction_Height <<<N_noOfStates, N_noOfStates >>>
(dev_epsilon_3D, M_noOfObsSequences);

ColumReduction_Height <<< N_noOfStates,V_noOfObsSymbols>>>
(dev_gamma_3D, M_noOfObsSequences);
ColumReductionGamma_Depth <<<1, N_noOfStates >>>
(dev_gamma_3D, 0, V_noOfObsSymbols,M_noOfObsSequences, gamma_reduction_grid);

EstimateA <<<N_noOfStates, N_noOfStates>>>
(dev_A_prime_3D, dev_epsilon_3D, epsilon_reduction_grid,
epsilon_reduction_grid_error, 0, M_noOfObsSequences);

EstimateB <<<V_noOfObsSymbols, N_noOfStates>>>
(dev_B_prime_3D, dev_gamma_3D, gamma_reduction_grid, 0, M_noOfObsSequences);
```

Since the formula for computing the estimate $\hat{b}$ is incorrect we had to adopt our own approach.

This is implemented as the reduction concerning the gamma data structures in our code. First, we compute all values of $\gamma_t(i, j)$ by summing over M, then we compute $\gamma_t(j)$ by summing over the symbol domain V, and store this in our reduction data structure.

3 Evaluation

3.1 Data Generation & Setup

Our data generation happens in two passes. First we produce the transition and emission matrices A and B. All values in these matrices are random, but picked such that each row sums to 1. Once the matrices have been generated we use the sequence generator of Liu [2] to produce observation sequences of length $L = 16$. Both of these stages are performed on the CPU with regular C++ code.

To evaluate the runtime of the CUDA kernels we used NVIDIAs *cudaEventRecord*. An example of how we implement this in code is as follows.

```
startBenchmark(..);
for (int i = 0; i < ITERATIONS; i++)
{
// run kernel
}
stopBenchmark(..);
```

Where *startBenchmark* and *stopBenchmark* are wrapper functions for *cudaEventRecord*. To evaluate the correctness of our algorithms we compared them to the output of Lui's implementation with the same data set. However, the forward algorithm was able to be executed alone in Liu's code. Therefore we wrote our own serial implementation and compared it against that. Although this is not very precise, the algorithm is simple enough to write from scratch.

From this we can assert that our forward and viterbi algorithms produce correct results. However, the Baum-Welch algorithm does not. Clearly there is an error in the algorithm, but due to the sloppy defintion of this alorithm presented in [3], it is not clear where this error is. Nevertheless, we give our results of all algorithms in the next section.

3.2 Comparison

GPU	GForce GTX 860M
CPU	intel i7-4710HQ @2.50 GHz with 8 GB of RAM
OS	Windows 10
CUDA	CUDA 7.5 CM 3.0
Host Compiler	Visual Studio 2013

Figure 9: Test Environment

Data size (N × M)	Liu CPU	Liu GPU	GPU (Average of 1000 Iterations)
64 × 64	687.7	10.2	13.6
128 × 128	5621.9	19.74	18.15
192 × 192	18990.5	40.93	35.93
256 × 256	45031.6	71.77	47.20
320 × 320	88090.8	128.44	77.09
448 × 448	152374.8	208.08	124.99
512 × 512	360899.3	410.17	-

Figure 10: Comparing the run time of the forward algorithm in ms

Data size (N × M)	CPU 1D	GPU 1D
64 × 64	208.53	902.71

Figure 11: Comparing the run time of the BW algorithm in ms

Data size (N × M)	Liu CPU	Liu GPU	GPU + CPU(Average of 1000 Iterations)
64 × 64	2138.4	35.9	115.43
128 × 128	7891.1.9	142.6	297.26
192 × 192	57681.3	339.1	722.38
256 × 256	136694.2	903.3	-
320 × 320	267038.8	1328.6	-
448 × 448	461297.6	2479.6	-
512 × 512	1094036.4	6054.8	-

Figure 12: Comparing the run time of the BW algorithm in ms

3.3 Conclusion

Examining figure 9. we can see that our algorithm outperforms Liu's code, however the paper was published in 2009. Therefore, it is safe to assume that our set up was using better hardware and compiler technologies than available to Liu. Nevertheless, we can see that the performance gain of the GPU code is very significant.

Most notably the measurement from the last row is missing. This is most probably due to a memory leak in our code as experiments with a low iteration count still complete.

The results for the Viterbi algorithm were very underwhelming. Although we do everything in CUDA kernels, the kernel still follows the serial structure of the pseudocode i.e. for loops. Nevertheless, we can see that, maybe not surprisingly, CUDA is not good for implementing serial code.

Therefore, for the Baum-Welch algorithm we incorporated CPU measurements into our results. The reason for this is that some work is done on the CPU side and it would falsify our measurements if we would omit this. For this reason our measurements are much larger than Liu's pure GPU performance. As mentioned before, our Baum-Welch implementation fails to produce correct values, however we do not believe this to be a problem of the algorithm's general structure. Therefore, the algorithms overall complexity will likely not change for a correct implementation.

As before omitted values indicate a memory leak.

In general it can be seen that by using the GPU considerable performance gains can be made when applied to Hidden Markov Models. Further improvements can be made by using Texture Memory [3] as data structures such as the transition and emission matrices are read only for the forward and viterbi algorithm. Furthermore, dynamic parallelism can be use to optimize the code further, as renditions can be spawned form inside CUDA kernels and thus don't require context switching back to the CPU. These could be topics for future projects.

References

[1] Mark Harris. Optimizing parallel reduction in cuda. 0.

[2] Daniel Jurafsky and James H Martin. Speech and language processing. 2014.

[3] Chuan Liu. A cuda implementation of hidden markov models. 2009.

4 Appendix

4.1 CUDA Kernels

```
// FROM: NVIDIA Optimizing Parallel Reduction in CUDA
__global__ void reduce_1(double* g_idata, double* g_odata){

extern __shared__ double sdata[];

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

sdata[tid] = g_idata[i];

__syncthreads();

for (unsigned int s = 1; s < blockDim.x; s *= 2){
if (tid % (2 * s) == 0){
sdata[tid] += sdata[tid + s];
}

__syncthreads();
}

if (tid == 0){
g_odata[blockIdx.x] = sdata[0];
}
}

__global__ void pointwiseMatrixMul(double * dev_w, double *dev_A, double* dev_B){

unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;

unsigned int idx_k = ix;

dev_w[idx_k] = dev_A[idx_k] * dev_B[idx_k];

}

__global__ void AlphaTrellisInitializationGPU(double *dev_3D_Trellis, const double *dev_

int obs_index = blockIdx.x * T_noOfObservations;
int obs_start = dev_O_obsSequences_2D[obs_index];
int idx_b_i_idxOs = threadIdx.x*V_noOfObsSymbols + obs_start;
int idx_alpha_0i = blockIdx.x * N_noOfStates + threadIdx.x;
```

```
int idx_pi_i = threadIdx.x;

double alpha_0_i
= dev_Pi_startProbs_1D[idx_pi_i] * dev_B_obsEmissionProbs_2D[idx_b_i_idxOs];
dev_3D_Trellis[idx_alpha_0i] = alpha_0_i;


}

__global__ void initMatrix(double* matrix_3D, int depth){

int idx_3D = threadIdx.x*gridDim.x*depth + blockIdx.x;

matrix_3D[idx_3D] = 1.0f / blockDim.x;

}

__global__ void initBeta(double* beta_3D, int T_noOfObservations){

// 2D index
int idx = blockIdx.x*blockDim.x + threadIdx.x;

int offset = gridDim.x*blockDim.x*(T_noOfObservations - 1);

beta_3D[offset + idx] = 1;

}

__global__ void updateBeta(double* dev_beta_3D, double* dev_D,
int t, int T_noOfObservations){

int idx_2D = blockIdx.x*blockDim.x + threadIdx.x;
int idx_t_minus_1 = (t - 1)*blockDim.x*gridDim.x + idx_2D;


dev_beta_3D[idx_t_minus_1] += dev_D[idx_2D];
}

__global__ void EstimateB(double* dev_update, double*dev_source,
 double* reduction_grid, int m, int M){

int idx_3D = m * blockDim.x + blockIdx.x*blockDim.x*M + threadIdx.x;
int idx_top = blockIdx.x*blockDim.x*M + threadIdx.x;
```

```
int reduction_idx = threadIdx.x;

double val = dev_source[idx_top];
dev_update[idx_top] = val / reduction_grid[reduction_idx];

}

__global__ void EstimateA(double* dev_update, double*dev_source,
 double* reduction_grid, double* reduction_grid_error, int m, int M){

int idx_3D = m*blockDim.x + blockIdx.x*blockDim.x*M + threadIdx.x;
int idx_top = blockIdx.x*blockDim.x*M + threadIdx.x;

int reduction_idx = m*blockDim.x + blockIdx.x;
int reduction_idx_1D = blockIdx.x;

double val = dev_source[idx_top];
dev_update[idx_top] = val / (reduction_grid_error[reduction_idx_1D]);


}

__global__ void ColumReduction_Height(double* dev_update, int M){

int start = threadIdx.x + blockIdx.x*M*blockDim.x;


for (int i = 1; i < M; i++){

int idx = start + i*blockDim.x;

dev_update[start] += dev_update[idx];

}



}

__global__ void ColumReductionGamma(double* dev_update, int m, int M){

int start = threadIdx.x + blockIdx.x*blockDim.x*M;

for (int i = 1; i < M; i++){
```

```
int idx = start + i*blockDim.x;

dev_update[start] += dev_update[idx];

}



}

__global__ void ColumReductionGamma_Depth(double* dev_update, int m,
 int V,int M, double* grid){

int start = threadIdx.x;

for (int i = 0; i < V; i++){

int idx = start + i*blockDim.x*M;

grid[threadIdx.x] += dev_update[idx];

}

}
```