HMM Inference with CUDA

Marc Haubenstock & Christian Brändle

June 30, 2016

Introduction - stochastic process

A *Hidden Markov Model* describe a **two-state stochastic process**[1]. There is a stochastic process that is stationary (which means its probabilistic features don't change over time) and a state space that is finite.

[1] Gernot A.Fink. *Markov Models for Pattern Recognition*. Springer London: Springer, 2014.

Introduction - finite state automaton

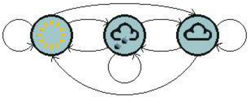	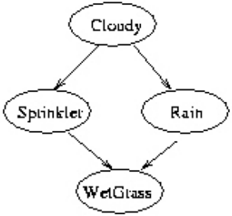
weather HMM[2]	grass HMM[3]

Table: Two finite state automatons that describe the state space of two different HMMs. Nodes correspond to states and edges to transition probabilites betwenn states that are bigger than 0.

[2] Ramesh Sridharan. *HMMs and the forward-backward algorithm.*

[3] Kevin Murphy. *A Brief Intro to Graphical Models and Bayesian Networks.*

Introduction - Markov property

The important thing is that **the behaviour of the process given at time t only depends on the immediate predecessor state**[4]. So the *Markov property* states:

$$P(S_t|S_1, S_2, \dots S_{t-1}) = P(S_t|S_{t-1}) \tag{1}$$

[4] Gernot A.Fink. *Markov Models for Pattern Recognition*. Springer London: Springer, 2014.

Introduction - graph model

Visually this can be shown with a graph model of a HMM as a sequence of hidden states X_i and the corresponding obervations Y_i - see figure 1.

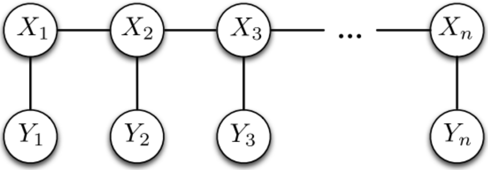

Figure:[5]

[5] Nikolas Doerfler. "Hidden Markov Models". In: ().

Introduction - output probalitities

Furthermore, the corresponding **probability distribution only depend on the current state**[6]. This is called the *output independence assumption*.

$$P(O_t|O_1, O_2, \dots O_{t-1}) = P(O_t|S_{t-1}) \tag{2}$$

The model itself is hidden because we only can observe the outputs generated, namely the *observation sequence* $O_1, O_2, \dots, O_T$.

[6] Gernot A.Fink. *Markov Models for Pattern Recognition*. Springer London: Springer, 2014.

Introduction - output probability cont.

Bayes rule

$$P(\lambda_j|O)P(O) = P(O|\lambda_j)P(\lambda_j) \tag{3}$$

what we want - Total output probability

$$P(O|\lambda) \tag{4}$$

what we get - output probability[7]

$$P(O, s|\lambda) = P(O|s, \lambda)P(s|\lambda) = \prod_{t=1}^{T} a_{s_{t-1},s_t} b_{s_t}(O_t) \tag{5}$$

with brute force - $O(TN^T)$

[7] Gernot A.Fink. *Markov Models for Pattern Recognition*. Springer London: Springer, 2014.

Introduction - symbols

$S = s_1 s_2 \cdots s_N$	a set of N states.
$V = v_1 v_2 \cdots v_{\lvert V \rvert}$	a set of distinct observations symbols. $\lvert V \rvert$ 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. $a_{ij} = P(q_{t+1} = s_j \mid q_t = s_i)$
$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1\lvert V \rvert} \\ b_{21} & b_{22} & \cdots & b_{2\lvert V \rvert} \\ \vdots & & & \\ b_{N1} & b_{N2} & \cdots & b_{N\lvert V \rvert} \end{bmatrix}$	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: $\pi_i = P(q_1 = S_i)$ [8]

[8] Chuan Liu. "A Cuda Implementation of Hidden Markov Models". In: ().

Algorithms - Computing Likelihood

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

Figure: Pesudo-Code of Forward Algorithm - determine likelihood of $P(O|\lambda)$ [4]

[9]

[9] Chuan Liu. "A Cuda Implementation of Hidden Markov Models". In: ().

Algorithms - Decoding

VITERBI(O)

initialize all cells of α to 0
$\alpha(o_1, s) \leftarrow 1$
for $t = o_2$ to o_T
    do for $i = 1$ to N
        do for $j = 1$ to N
            do $p \leftarrow a_{ji} \cdot b_{it}$
                if $\alpha(t-1, j) \cdot p > \alpha(t, i)$
                    then $\alpha(t, i) \leftarrow \alpha(t-1, j) \cdot p$
                        $backpointers(t, i) \leftarrow j$
$states = \text{BACKTRACE}(backpointers)$
return $states$

Figure: Pesudo-Code of Viterbi Algorithm - find best hidden state sequence [4]

10

[10] Chuan Liu. "A Cuda Implementation of Hidden Markov Models". In: ().

Algorithms - Learning

FORWAD-BACKWARD-EXPECTATION(O)

initialize all cells of $\alpha, \beta, \gamma, \xi$ to 0
$likelihood \leftarrow$ FORWARD(O)
$\beta(o_T, e) \leftarrow 1$
for $t = o_T$ to o_1
  do for $i = 1$ to N
    do $\gamma(t,i) \leftarrow \gamma(t,i) + (\alpha\big((t,i) \cdot \beta(t,i) / \, likelihood\big)$
    $\xi(i) \leftarrow \xi(i) + (\alpha\big((t,i) \cdot \beta(t,i) / \, likelihood\big)$
    for $j = 1$ to N
      do $p \leftarrow \alpha_{ji} \cdot b_{it}$
        $\beta(t-1, i-1) \leftarrow \beta(t-1, i-1) + \beta(t,i) \cdot p$
        $\xi(j,i) \leftarrow \xi(j,i) + \big(\alpha(t-1,j)\beta(t,i) \cdot p / \, likelihood\big)$

Figure: Pesudo-Code of Backward Forward Algorithm - learn the HMM model λ [4]

[11]

[11] Chuan Liu. “A Cuda Implementation of Hidden Markov Models”. In: ().

Algorithms

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

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

Figure: Estimation of the A and B Matricies [4]

12

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 [3].

[12] Chuan Liu. "A Cuda Implementation of Hidden Markov Models". In: ().

Implementation

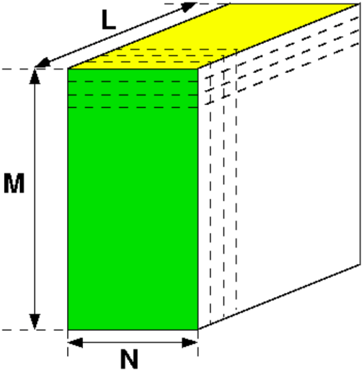

Figure: Graphical representation of the data structure presented in[13]

[13] Chuan Liu. "A Cuda Implementation of Hidden Markov Models". In: ().

Implementation cont.

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

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.

Implementation cont.

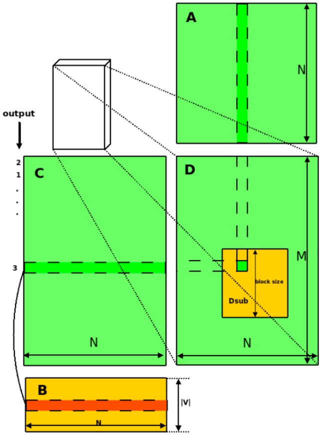

Figure: The computation of a slice[14]

[14] Chuan Liu. "A Cuda Implementation of Hidden Markov Models". In: ().

Implementation cont.

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.

Forward

```
__global__ initTrellis<<<M,N>>>(..){
  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[idx_b_i_idxOs];
  dev_3D_Trellis[idx_alpha_0i] = alpha_0_i;
}
```

Forward

```
// Computes the B matrix in term D = B .* C x A
ComputeBDevice << < M, N >> >
(M, V, T, N,
dev_O_obsSequence_2D, dev_B_obsEmissionProbs_2D, i, dev_B);

// All dimensions are multipe of Warp Sizes
// Compute W =  B .* C
pointwiseMatrixMul << <M, N >> >(dev_W, dev_B,
&dev_3D_Trellis[(i - 1) * M * N]);

// Compute D = W x A
cublasMultiplyDouble(M, N_noOfStates, N, dev_W,
dev_A_stateTransProbs_2D, &dev_3D_Trellis[i * M * N]);
```

Forward

```
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_likelihoods_1D[i] = host_A_odata[0] +
    host_A_odata[1];

}
```

Forward

```
// FROM: NVIDIA Optimizing Parallel Reduction in CUDA
__global__ void reduc1(double* g_idata, double* g_odata){
  extern __shared__ double sdata[];
  int tid = threadIdx.x;
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();
  for (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];
```

Viterbi - viterbi1D central algorithm

```
double p = a_ji * b_it;
double alpha_tm1j =
dev_Alpha_trelis_2D[idx_alpha_tm1j];
double alpha_ti =
dev_Alpha_trelis_2D[idx_alpha_ti];
double partPathProb_tm1j = alpha_tm1j * p;
if (partPathProb_tm1j >alpha_ti)
{
  dev_Alpha_trelis_2D[idx_alpha_ti] =
  partPathProb_tm1j;
  // backpointers(t,i) = j
  dev_Gamma_trellis_backtrace_2D[idx_alpha_ti] =
  idx_j;
}
```

Viterbi - kernel 1D - slicing

```
int idx_m = blockIdx.x * blockDim.x + threadIdx.x;
// check if idx_m is out of range
if (idx_m > M_noOfObsSequences)
  return;

// slice matrices according to idx_m
double *dev_Alpha_trelis_TN_2D = ...;
int *dev_Gamma_trellis_backtrace_TN_2D = ...
int* dev_O_obsSequence_1D = ...
int* dev_likeliestStateIndexSequence_1D = ...
```

Viterbi - kernel 1D - tracking

```
// device memory initialization
TrellisInitialization2D(...);

// creating indices for viterbi1D
for (int idx_t = 1; idx_t < T; idx_t++)
  for (int idx_i = 0; idx_i < N; idx_i++)
    for (int idx_j = 0; idx_j < N; idx_j++)
    {
      viterbi1D(...);
    }
  // rescaling of Trellis'
  TrellisScaling2D(dev_Alpha_trelis_TN_2D, ...);
```

Viterbi - kernel 1D - backtrack

```
// extract most likely path of states
// that generates observation
double partPathProb_optT = 0;
unsigned int idx_j = 0;
for (int idx_i = 0; idx_i < N; idx_i++)
  if (idx_i == 0 ||
    dev_Alpha_trelis_TN_2D[(T - 1)*
    N + idx_i] > partPathProb_optT)
  {
    partPathProb_optT =
      dev_Alpha_trelis_TN_2D[(T - 1)*
      N + idx_i];
    idx_j = idx_i;
  }
```

Viterbi - kernel 1D - backtrack II

```
dev_likeliestStates_1D[T - 1] = idx_j;
for (int idx_i = 1; idx_i < T; idx_i++)
{
  dev_likeliestStates_1D[T - 1 - idx_i] =
    dev_Gamma_trellis_backtrace_TN_2D[(T - idx_i)*
    N_noOfStates +
    dev_likeliestStates_1D[T - idx_i]];
}
```

Baum-Welch

```
for (int t = T_noOfObservations - 1; t >= 0; t--) {
 UpdateGammaGPU << <M, N >> >(..);
 UpdateEpsilonReductionErrorGPU <<<M, N >>>(..);
 if (t > 0){
  // Computes the B matrix in term D = B .* C x A
  // All dimensions are multipe of Warp Sizes
  // Compute W =  B .* C i.e. beta(t,i) * b_it
  ..
  for (int j = 0; j < N_noOfStates; j++){
   UpdateEpsilonGPU <<<M,N>>>
   (dev_epsilon_3D,dev_beta_3D,
    dev_3D_Trellis_Alpha,t,dev_likelihood,j,dev_D);
  } // end of for j
 } // end of if
} // end of for t
```

Baum-Welch

```
for (int t = T_noOfObservations - 1; t >= 0; t--) {
UpdateGammaGPU << <M, N >> >(..);
UpdateEpsilonReductionErrorGPU <<<M, N >>>(..);
 if (t > 0){
  // Computes the B matrix in term D = B .* C x A
  // All dimensions are multipe of Warp Sizes
  // Compute W =  B .* C i.e. beta(t,i) * b_it
  ..
  for (int j = 0; j < N_noOfStates; j++){
   UpdateEpsilonGPU <<<M,N>>>
   (dev_epsilon_3D, dev_beta_3D,
   dev_3D_Trellis_Alpha,t,dev_likelihood,j,dev_D);
  } // end of for j
 } // end of if
} // end of for t
```

Baum-Welch

```
ColumReduction_Height <<<1, N >>>
(epsilon_reduction_grid_error, M);
ColumReduction_Height <<<N,N>>>(dev_epsilon_3D, M);

ColumReduction_Height <<< N,V>>>
(dev_gamma_3D, M_noOfObsSequences);
ColumReductionGamma_Depth <<<1, N >>>
(dev_gamma_3D, 0, V,M, gamma_reduction_grid);

EstimateA <<<N, N>>>(..);

EstimateB <<<V, N>>>(..);
```

Evaluation

First we produce the transition and emission matrices A and B. Once the matrices have been generated we use the sequence generator of Liu to produce observation sequences of length $L = 16$. Both of these stages are performed on the CPU with regular C++ code.

Evaluation

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

Where *startBenchmark* and *stopBenchmark* are wrapper functions for *cudaEventRecord*.

Evaluation

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: Test Environment

Evaluation

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

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

Evaluation

Data size (N $\times$ M)	CPU 1D	GPU 1D
64 $\times$ 64	208.53	902.71

Figure: Comparing the run time of the Viterbi algorithm in ms

Data size (N $\times$ M)	Liu CPU	Liu GPU	GPU+CPU(Avg. of 1k)
64 $\times$ 64	2138.4	35.9	115.43
128 $\times$ 128	7891.1.9	142.6	297.26
192 $\times$ 192	57681.3	339.1	722.38
256 $\times$ 256	136694.2	903.3	-
320 $\times$ 320	267038.8	1328.6	-
448 $\times$ 448	461297.6	2479.6	-
512 $\times$ 512	1094036.4	6054.8	-

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

Conclusion

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* [4] 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 reductions 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.