CSE551 Final Project: Parallel Viterbi on a GPU

Seong Jae Lee, Miro Enev

Abstract

As a class project, we developed a distributed Viterbi path algorithm and deployed it on a NVIDIA 9800 GTX 128-core graphic processing unit (GPU). On problem sets with high dimensionality (6000 hidden states) the optimized version of our multi-core Viterbi implementation was 300 times faster than its CPU counterpart (Intel Core 2 Extreme CPU X9650 @ 3.00GHz). We conclude that the key to parallelism based performance enhancement lies in an engineering balance between software and hardware. The best results are achieved by maximally exploiting the inherent concurrency of the algorithm whilst intelligently structuring I/Os. By coalescing read accesses and caching to local memory we were able to minimize the data transfer bottlenecks which dominate the performance of our GPU's single instruction multiple data (SIMD) architecture.

Motivation

Computing power in modern applications is advanced by growing the number of parallel cores rather than manufacturing CPUs with faster clock speeds. This emphasis on parallelism is motivated by the physical limitations of transistor density on integrated circuits as well as the economics benefits of utilising mass produced micro processors. Another key factor for widespread adoption of concurrent processing is performance, as multi-core implementations of scientific algorithms often outpace their single core counterparts by several orders of magnitude [1].

Motivated by the trend towards parallelization, we harnessed new software and hardware tools to implement a multi-core version of the Viterbi path algorithm which operates over a Hidden Markov Model (HMM) instance. Given a series of observations and an HMM model (state transition and emission probability matricies) the Viterbi path finds the sequence of [hidden] system states most likely to have generated the observations. We chose the Viterbi path algorithm because its computational structure is well suited to synchronous progress and also because of its ubiquitous use in many scientific contexts including speech recognition, probabilistic inference, and genetic sequencing [2].

GPU Architecture

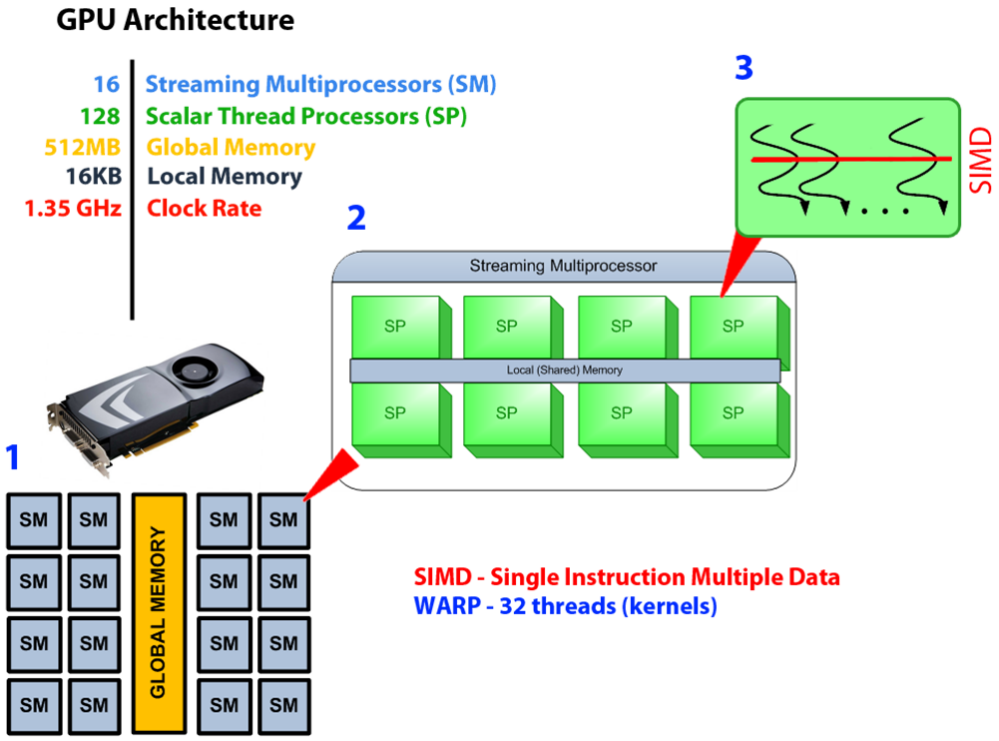
We chose to develop our algorithm on an NVIDIA GeForce 9800 GTX graphic processing unit

(GPU). We opted to use a graphics processor as a parallel computing platform rather than using a cluster of PCs because the GPU offers impressive performance at very low cost. Although the GPU has been traditionally utilized for intensive graphics applications (i.e. 3D design and gaming) its native set of operations allow for generic floating point

computations. Thus, by exploiting the flexibility of the instruction set we were able to harness the hardware to perform scientific computing rather than triangle rasterization.

A graphical representation of the GPU structure is provided in [Figure 1]. The graphics card is composed of a set of 16 *streaming multiprocessors (SM)*, each of which is a block made up of a set of 8 computational units known as *scalar thread processors (SP)*. In our case each SM (i.e. 8 SPs) was able to simultaneously execute a *warp* of 32 threads. A key property of the architecture is that at every GPU clock cycle the threads in a warp have to [all] execute the same instruction (with potentially differing arguments). Whenever this *single instruction multiple data* (SIMD) constraint is broken concurrency is diminished (ex: threads that diverge on alternate code paths force their warp-neighbors to become inactive while waiting for convergence).

The SMs all have access to a large block of global memory (512MB). Within each SM there is also 16KB of local memory which is shared by the SPs to enable efficient communication between threads in a warp.



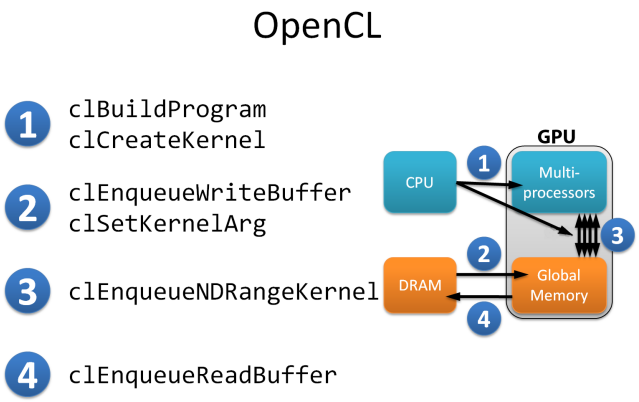
[Figure 1]. The GPU architecture is broken down into three levels of detail. At the highest level is a set of 16 streaming multiprocessors (SM) with shared access to 512MB of global memory. Each SM is composed of 8 scalar thread processors (SP) which are the core computational units. The SPs run synchronized warps of 32 threads which follow the single instruction multiple data (SIMD) paradigm. For additional hardware details refer to the Appendix.

OpenCL

The software side of our implementation was developed with OpenCL. We chose to use OpenCL because this open source framework is quickly becoming the industry standard for parallel programming on modern hardware. At its heart, OpenCL extends the C++ standard

programming language with high level APIs to distribute computations to a wide range of compliant multi-core devices (GPUs). Within the OpenCL model the CPU is used as a controller to feed inputs and trigger execution of code on the GPU.

The OpenCL workflow which we used is very standard and is depicted in [Figure 2]. The CPU begins by compiling the kernel code (synchronous portion of a program) and loading it onto the GPU; next kernel arguments and relevant data are transfered to GPU memory (from DRAM); the kernel threads are then launched on the GPU and synchronous processing is initiated; lastly, when the computations are completed, the results are pushed back into main memory.



[Figure 2]. Essential aspects of the OpenCL workflow.

Viterbi Algorithm

The Viterbi path algorithm is a dynamic programming method that finds the sequence of system states most likely to have emitted a given sequence of observations. The Viterbi algorithm was developed in the 1960's to aid in the decoding of error-correcting codes used to overcome the signal loss issues due to noisy digital communication links (i.e. cellular communications)[1]. Since its inception it has been widely adopted in many commercial and scientific contexts and its uses are continually growing.

The Viterbi algorithm operates over a hidden Markov model (HMM) instance and therefore follows the state machine assumption as well as the Markov chain property (i.e. the present separates the past from the future). The HMM acts as a statistical definition of a dynamical system and can be mathematically defined using the following ingredients:

> (1) ***state transition matrix*** - captures the likelihood of state transitions between the N possible internal/hidden states; an element (i, j) of this matrix represents the probability of transitioning from state i (at time t) to state j (at time t+1).
> (2) ***emission matrix*** - an element (i,j) of this matrix represents the probability with which state i can produce the emission (externally visible observation) j.
> (3) ***initial state distribution*** - the distribution of the system (likelihood of any of the N possible states) before any observations were seen (t = 0)

Given an HMM and a sequence of observations the Viterbi path algorithm finds the optimal sequence of hidden states using a forward and a backwards pass.

Viterbi Forward Pass

Given the initial probability distribution over the N hidden states at the initial time (t=0) the forward pass calculates the likelihood over all the possible states as the system evolves through time (t=0->T). This forward process is based on the following computations which are described below and visualized in [Figure 3]:

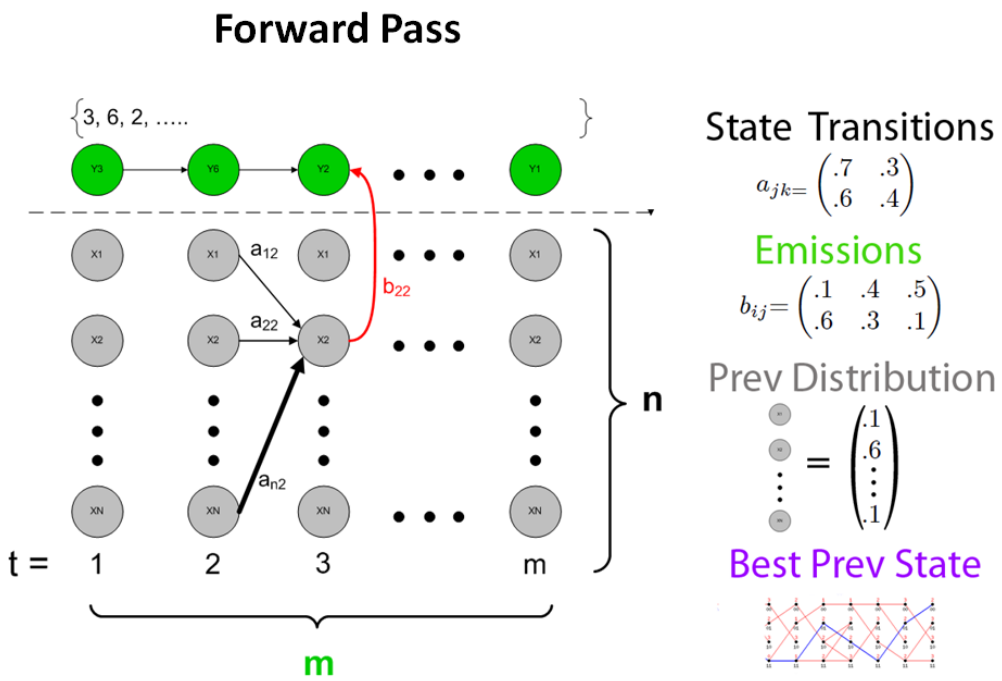
(1) **Find the most likely predecessor**: for each target state I (at time t) find the predecessor state J (among all possible predecessors) such that the following quantity is maximized

opt_J = argmax(J) log { [prob. system is in state J at time t-1] +
[prob. of transitioning from J to I] }

(2) ***Update probability of I***: Once the optimal J has been found using step (1), I's probability can be calculated as

prob I = log { [prob of opt_J at time t-1] +
[transition prob. from opt_J to I] +
[emission prob. state I gave rise to the obs at time t] }

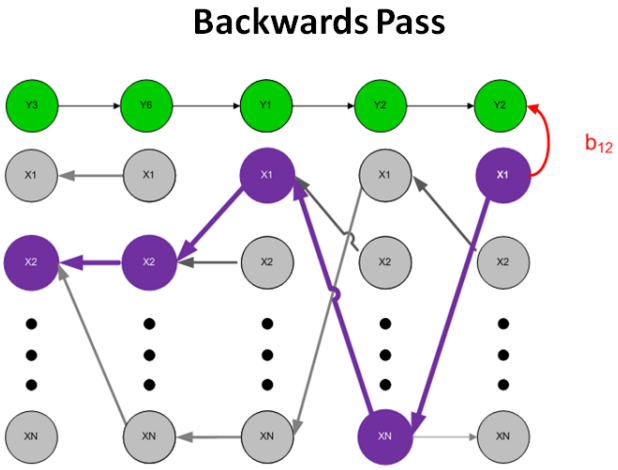
(3) ***Bookkeeping***: Save a backwards pointer to the most likely predecessor opt_J for each target state I



[Figure 3] Viterbi forward pass. The left panel depicts the sequence of observations and the unrolled HMM beneath. Columns of nodes are grouped into time instances and time unfolds from left to right. The observations {y's} are depicted as the sequence of green nodes, and the hidden system states {x1:xN} are shown below them in gray. The arrows are only shown for one time instance to represent how transitions and emission probabilities are calculated (the emphasized black arrow represents the optimal predecessor; the red arrow represents the emission probability). The right panel depicts all the necessary data structures.

Viterbi Backwards Pass

The output of the forward pass is a distribution of over the hidden states of the system at the final time as well as a history of backwards pointers (trellis) to the best predecessor of each state (for all time steps). The backwards pass starts from the final state with the highest probability mass and traverses the trellis backwards while accumulating a sequence of the optimal predecessors [Figure 4].



[Figure 4] Viterbi backwards pass, in this example the optimal sequence of states was {x2, x2, x1, xN, x1} for the observation sequence {y3, y6, y1, y2, y2}.

Parallelization Scheme

From a concurrency standpoint, only the forward pass of the Viterbi algorithm is of interest since the backwards pass is a serial sweep through a list of pointers. From an analytical standpoint we can define the bounds on the complexity as follows: when the total number of hidden states is *n* and the total number of observations is *m*, the forward algorithm takes $O(n^2m)$, while the backward propagation takes only $O(nm)$ computations.

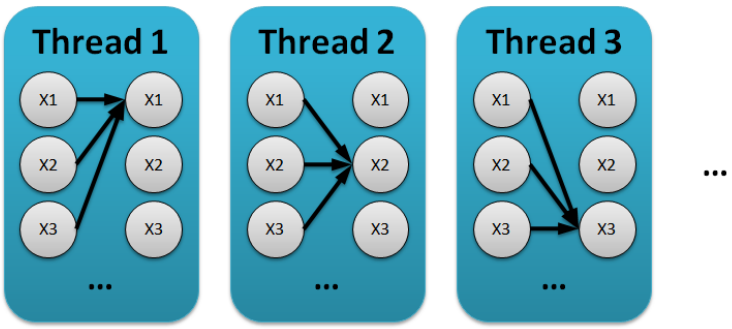
Initially we wrote the CPU serial/classic version of the algorithm [see Algorithm 1 snippet] and validated it against several online datasets [6]. Next we developed a GPU version [Algorithm 2] wherein we distributed the state computations for each time slice amongst GPU threads by assigning the i-th thread to compute the probability of state i (at time t) [Figure 5].

```
for (curr = 0; curr < nStates; curr++) {
   for (prev = 0; prev < nStates; prev++) {
      prob = prev_dist[prev] + transition[prev*nStates + curr];
      ...
```

[Algorithm 1] The original version of the algorithm

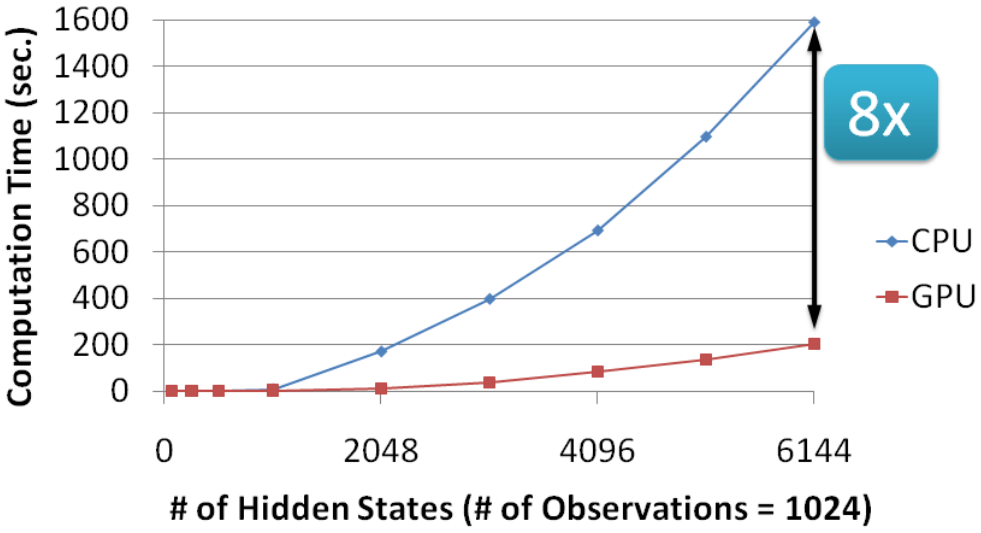
```
index = globalId * localSize + localId;
for (curr = index; curr < nStates; curr += localSize*globalSize) {
   for (prev = 0; prev < nStates; prev++) {
      prob = prev_dist[prev] + transition[prev*nStates + curr];
      ...
```

[Algorithm 2] Distributed version of the algorithm



[Figure 5] The i-th thread is responsible for updating the probability of state i (at time t).

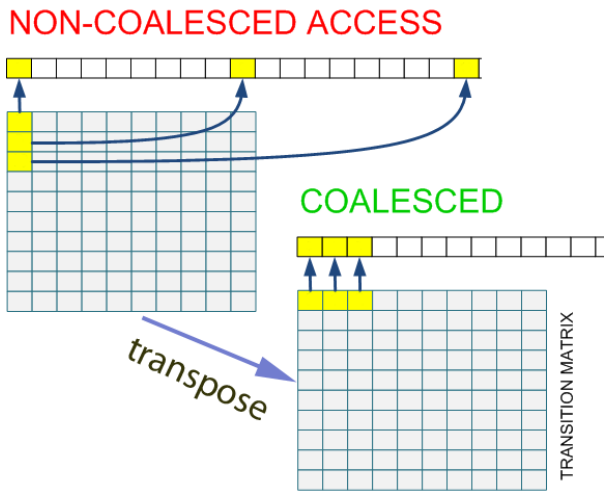
The distributed version led to significant improvements (8x) over the single core computation. This can be seen in [Figure 6] below. In our evaluation we fixed the length of the observation sequence and only manipulated the number of hidden states; this design choice was motivated by the fact that the sequence length influences the complexity in a linear fashion whereas the number of hidden states n produces a quadratic term (recall that the forward algorithm is $O(n^2m)$ -- see Appendix 2).



[Figure 6] CPU vs GPU (i.e. [Alg 1] vs [Alg 2])

Optimizations

As we looked for further methods of optimization we quickly realized through code profiling that the main bottleneck of our distributed Viterbi algorithm (version 2) was the memory access in the transition matrix. When we though about this further we were reminded that our transition 'matrix' was just a long array, and that the threads in a warp were attempting to synchronously access memory locations that were spread across global memory blocks even though the data items were adjacent in the matrix representation [Figure 7].



[Figure 7] Transposing the transition matrix to coalesce memory accesses.

To alleviate this problem we created the next version [Algorithm 3], in which we transposed the transition matrix (leading to coalesced reads). The results of this optimization led to a 7x improvement over the previous GPU version [Algorithm 2]. We attribute the speedup to the high latency of global memory reads (about 400~600 cycles) which is a key performance consideration. Because one large transfer from global memory is much faster than many small ones, the coalesced reads led to significant improvements.

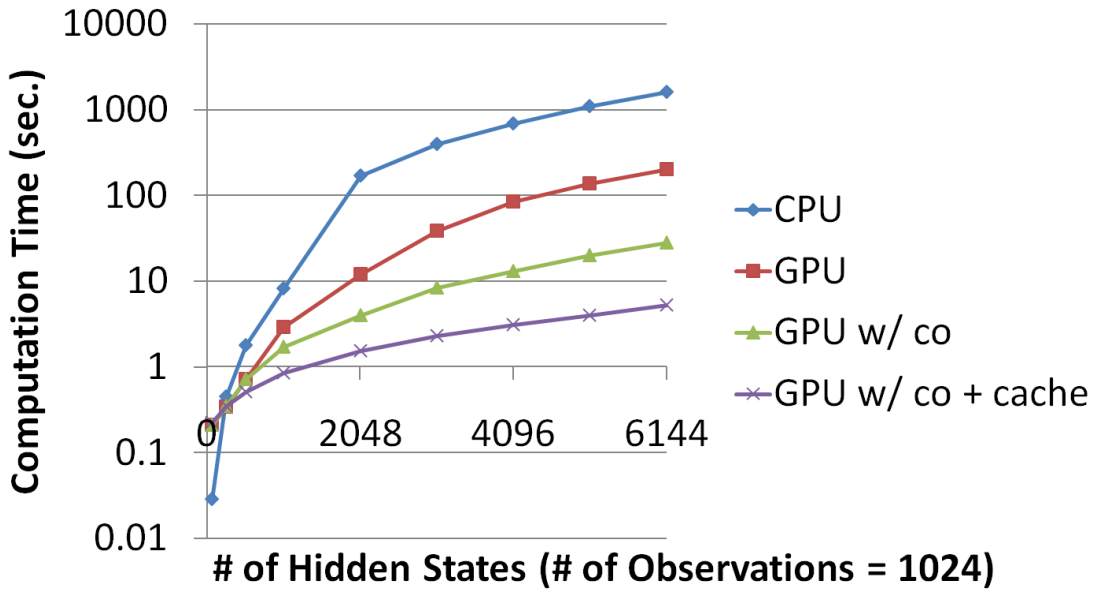
```
index = globalId * localSize + localId;
for (curr = index; curr < nStates; curr += localSize*globalSize) {
   for (prev = 0; prev < nStates; prev++) {
      prob = prev_dist[prev] + transition[prev + nStates*curr];
      ...
```

[Algorithm 3] Transition matrix is now transposed, enabling a coalesced global memory access

When compared to global memory local memory is 100 times faster. Therefore, our final optimization made use of caching in local memory to reduce global memory access latencies. In our last version we combined local caching of the probability distribution at time t-1 (needed in the first two steps of the Viterbi forward computation) with the coalesced memory access paradigm from Algorithm version 3. This led to our best performer - [Algorithm 4] which was 5.6 times faster than Algorithm 3 [see Figures 8].

```
index = globalId * localSize + localId;
for (curr = index; curr < nStates; curr += localSize*globalSize) {
   for (prev = 0; prev < nStates; prev++) {
if (prev % localSize == 0) {
      cache[localId] = prevDist[prev + localId];
      barrier(CLK_LOCAL_MEM_FENCE);
      }
      prob = cache[prev % localSize] + transition[prev + nStates*curr];
      ...
```

[Algorithm 4] Introduced a shared variable *cache*



[Figure 8] Performance results of all 4 algorithm versions side by side - log scale computational time.
For non-log scale see Appendix 3.

Conclusion

In conclusion we were able to successfully exploit the parallel structure of a the Viterbi path algorithm - a popular dynamic programming tool - and deploy it on a NVIDIA GPU. We had to learn how to extend the power of the GPU beyond graphics and harness its massively concurrent hardware while overcoming memory latencies. Through iterative improvements we were able to produce a distributed Viterbi path solver with coalesced global memory access and shared memory caching. Along the way we also gained familiarity with a flexible open source language that makes our experience portable to a large (and growing) set of OpenCL compliant multi-core hardware platforms. We feel that the parallelization project was a very valuable venture and we hope to bring our new knowledge to bear in future research endeavors.

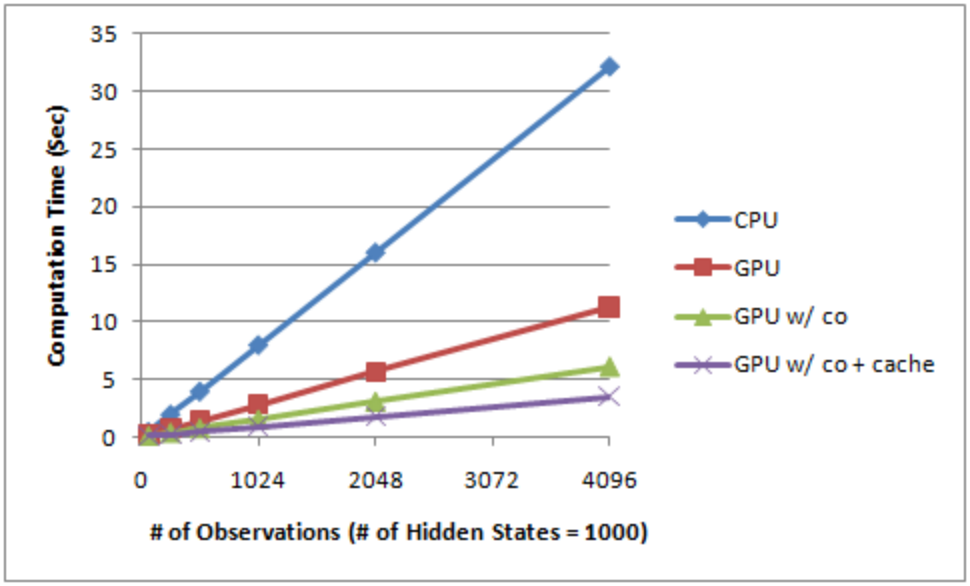
References

- Wikipedia, Viterbi Algorithm, http://en.wikipedia.org/wiki/Viterbi_algorithm
- David Luebke et al., GPU Architecture: Implications & Trends (2008), http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf
- Catanzaro et al., Fast Support Vector Machine Training and Classification on Graphics Processors, http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-11.pdf
- NVIDIA Corporation, OpenCL Optimization (2009), http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Best_Practises_For_OpenCL_Programming.pdf
- NVIDIA Corporation, GeForce 9800 GTX, http://www.nvidia.com/object/product_geforce_9800_gtx_us.html
- HMM Programming Project http://inst.eecs.berkeley.edu/~cs188/sp08/projects/hmm/project_hmm.html

Appendix 1: GPU Specifications

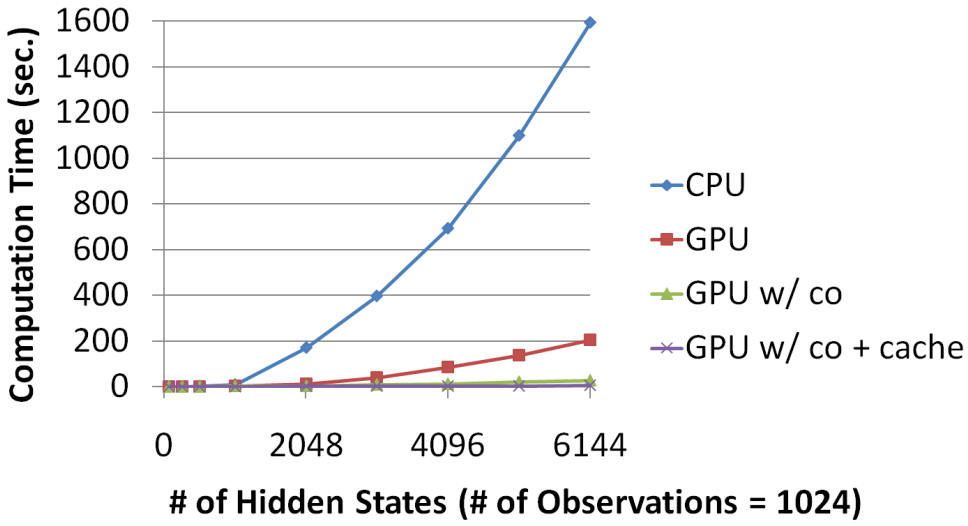
Number of Multiprocessors	16
Multiprocessor Width	8
Multiprocessor Local Store Size	16KB
Number of Stream Processors	128 (= 16 x 8)
Peak General Purpose IEEE SP	356GFlops
Clock Rate	1.35GHz
Memory Capacity	512MB
Memory Bandwidth	70.4GB/s

Appendix 2: [Linear] Effects of Observation Sequence Length



Comparison of algorithms on randomly generated HMMs with different number of observations.

Appendix 3: All algorithms side by side (performance) non log scale


Computation Time (sec.)
1600
1400
1200
1000
800
600
400
200
0
0
2048
4096
6144
# of Hidden States (# of Observations = 1024)
CPU
GPU
GPU w/ co
GPU w/ co + cache