Jack Harrison

Prof. Shuangshuang Jin

CPSC 4780

2 December 2022

Final Project Report

Introduction

For my final project, I chose to investigate speeding up performance of a Hidden Markov Model. A Hidden Markov Model in a two-dimensional domain space uses a nested for loop to update beliefs about each item in the domain space, one iteration per item in the space. Thus, it is a clear candidate for potential improvement by parallel processing. I found that in fact GPU processing did provide a substantial speedup: on a domain space of 128 x 128, my GPU implementation executes 100 iterations in approx. 2.2 seconds, whereas the CPU takes 15.9 seconds. This is a speedup of approx. 7.2. I completed this project alone; I did not have a partner,

Background

In Fall 2021, I took CPSC 4420, Artificial Intelligence, with Dr. Abolfazl Razi. Our last project assignment for that class was to implement a Hidden Markov Model. In particular, for that project my partner (a student named Ross Brown) and I implemented the algorithm functions for a Hidden Markov Model application written by Pei Xu (peix@g.clemson.edu) and Ioannis Karamouzas (ioannis@g.clemson.edu). (Note: My use of a partner in the 4420 project (but not in this one) and credit to the original app authors are both stated in comments in hmm.py.)

In the 4420 project, my partner and I implemented the timeUpdate() and observe() functions for both the exact-inference and particle-filtering implementation (4 functions total).

In thinking about good candidates for GPU speedup for the CPSC 4780 Final Project, it occurred to me that the exact-inference implementation uses a two-dimensional nested for loop, with one iteration for each item in the domain space of beliefs about the agent. Such an implementation seemed a clear candidate. (Note: The particle-filtering implementation does not span the entire domain space, and is not a particularly good candidate for GPU speedup, so I did not change it for this project.)

Implementation

In the assignment zipped directory, I have included two subdirectories: orig_implementation and new_implementation. orig_implementation contains the original implementation from my CPSC 4420 project, with the following tweaks to hmm_app.py:

1) I increased the domain space from 20 x 30 to 128 x 128 to better test scalability on the GPU.
2) I reduced the time between iterations from 100 ms to 1 ms via the first argument to self.canvas.after() on lines 158 and 160 to focus on computation. (The application is governed by the graphics library TKinter, which does not allow 0 ms between canvas flushes, i.e. iterations.)
3) I added timer functionality. This proved a bit tricky, as I had to ensure that the timer only began once the user presses “Run,” and only counted iterations when the agent is moving, i.e. the HMM is running.

The differences between orig_implementation and new_implementation are all in hmm.py. I used the Python library PyCUDA (with Prof. Jin’s permission) to call CUDA kernels from the Python implementations of observe() and timeUpdate(). My CUDA code is contained in two places: kernels.cu and the string beginning on line 94 in hmm.py. I originally wrote the CUDA code in kernels.cu for ease of development as well as modularity, and then copied the code to hmm.py for low-overhead input to the PyCUDA SourceModule() function.

The implementation is fairly straightforward. In both timeUpdate() and observe(), I converted the Python procedures inside the nested for loops, as well as the functions they call, to CUDA C++ in corresponding kernels. I call the kernels through PyCUDA in the respective functions. As the reader will notice in my implementation, PyCUDA handles much of the overhead of memory allocation, memory transfer, and synchronization.

General Notes

- I had to use atomicAdd() in the timeUpdate() function, as each each belief in the domain space must update other beliefs. I was concerned that this would significantly harm the speedup, but it appears that the speedup is still quite large. I could have reversed the transition function, and then called it for each of a belief's surrounding beliefs, to avoid the need for atomicAdd, but I did not for two reasons:
    - This would only work for this particular transition function, and would make the application unable to handle other possible transition functions of higher complexity.
    - The substantial increase in computation required to call all the surrounding beliefs in a thread meant that the performance gain was ambiguous.
    - This would decrease the degree to which the use of GPU is "apples-to-apples" with CPU.
- This implementation required me to learn about handling 2D array transfer in CUDA, and in particular about pitched memory.
- While the domain size I used is a power of 2 (128 x 128), the implementation is robust to other sizes.
- PyCUDA made CUDA development massively easier, as it handled much of the overhead.
- The application has overhead to computation in the following areas: graphics, the aforementioned timer between graphics flushes, and the application functionality.

Results and Analysis

On a domain space of 128 x 128, I found the following execution times for 100 iterations:

- CPU: 15.8774 seconds
- GPU: 2.2031 seconds

This is a speedup of 7.21, which is quite significant. It is certainly true that some of this speedup comes from just the use of C instead of Python, as the former is more low-level and thus faster, However, I suspect much of the speedup does in fact come from use of parallelism in FLOPs, given the nature of the domain space. It is worth noting that we see this drastic speedup despite the use of atomicAdd() in the timeUpdate() kernel.

References Used in Implementation

Of course, the primary reference source I used was the aforementioned starter code from my previous CPSC 4420 project. This is contained in orig_implementation (with three changes in hmm_app.py, noted earlier).

Additionally, I consulted the NVIDIA CUDA docs, the PyCUDA docs, and StackOverflow for various implementation questions. These sources mostly provided me general design understanding rather than particular code, though I did copy one useful function, normal_pdf from this post, as noted in the source code.