Parallel-Generation for Video Game Music

A parallel implementation of music generation using Markov chains and neural networks for the 15-418 Final Project (Fall 2018)

Annie Xu and Michael You

You can click the "Checkpoint" link above or go to: https://mikinty.github.io/Parallel-Video-Game-Music/checkpoint

15-418 Final Project Checkpoint

Updated Schedule

Dates	Goal	Status	Responsible Party	Description
Nov 4	Lit Review + Model	Done	Both	Learn about how MIDI is generated, and the various modifications to Markov Models and similar algorithms. Other research can include looking into video game music theory to properly determine the structure and rules we wish to follow. We want to decide on our model, determine why it works well, and what algorithm we will use to generate our music.
Nov 8	Formatting and	Done	Both	Decide on the proper data structures to use, format input MIDI files into

	Structure			usable data with C# and Python libraries, resolve storage issues, and decide on structure of client and server for real-time music generation.
Nov 16	Basic Parallel Alg.	Done	Annie	Implement a CUDA program that generates matrices for our program, based on the formatted and edited files from preprocessing in the previous step.
Nov 18	Music Generation Basics	Done	Both	Implement very basic sequential algorithm to generate playable MIDI files from Markov Chains
Nov 19	Checkpoint	Done	Both	Summarize progress so far on project
Nov 23	Music Generation	Ready to Start	Annie	Implement parallelization in music generation algorithm to generate playable MIDI files.
Nov 26	Server-Client Interface	Planned Out	Michael	Implement the proper way to network between client (which plays the MIDI file) and server (which provides the generated MIDI file)
Nov 30	Optimize Matrix Generation	Ready to Start	Annie	Debug and optimize the matrix generation, to speedup training time. Play with atomic vs. locks vs. other methods
Dec 3	Analyze Matrix Generation	Planned Out	Michael	Continue optimization of matrix generation, and plot and analyze resulting graphs
Dec 7	Optimize Music Generation	Planned Out	Michael	Optimize the music generation algorithm. Explore the balance between parallelizing melodic lines and harmonious output.
Dec 10	Analyze Music Generation	Planned Out	Annie	Continue optimization of music generation, and plot and analyze resulting graphs
Dec 14	Wrap-up	Not Ready	Both	Complete final report

Dec 15	Presentation	Not Ready	Both	Practice presentation

Summary of Work Completed

A lot of progress has been made in planning and structuring the final algorithms we wish to implement. After the week of literature review, we discovered new challenges with MIDI file formatting and the data structures we wanted to use. We spent an extra few days organizing and implementing algorithms to reformat the input data into usable and conveniently readable information for CUDA. We designed a straightforward file format to store all the note transition information needed as input to the matrix generation algorithm, and used C# and Python libraries such as `drywetmidi` and `music21` to process the input MIDI files into this format. After solving this challenge, we spent more time planning and discussing implementation plans for the remaining parts of the project, and developed more detailed predicted challenges and solutions for those parts.

With a more developed sense of the algorithms, it was much easier to implement both the sequential and parallel versions of the matrix generation algorithm. The algorithm reads input files in sequence, storing the necessary information in an array structure. We then use CUDA to count the number of note transitions in each section of the array, and atomic add operations to increment the output matrices with these counts. We also finalized the matrix dimensions and number of total output matrices and their format. However, there are still a few bugs and failing test cases with the parallel matrix generation algorithm, for which we have solutions and need to implement. For music generation, we successfully implemented a sequential music generation algorithm based on the trained matrices. Next steps can proceed in 3 directions:

1. Optimizing and analyzing the matrix generation
2. Parallelizing the music generation
3. Implementing the client interface

Summary of Challenges Faced and Changes Made

After a thorough literature review and algorithm plan, we ran into some challenges when starting our implementation of the matrix generation. The most difficult and unexpected challenge was the re-formatting of input MIDI files into a usable text file. MIDI files contain less detailed information than sheet music - they can be missing key signatures and tempo markings, making it impossible to tell how long each note is. However, we need this information in order to correctly count the note transitions in the music. Therefore, we had to search for algorithms that would analyze MIDI files to guess the correct tempo markings and key signatures. Thankfully, we were able to find many useful libraries, but they were written in C# and Python. In order to pre-process our input MIDI files into something that CUDA would be able to work with, we had to develop pre-processing programs in C# and Python which transformed input MIDI files to text files following a particular structure (which we designed). These text files were much simpler and presented the useful information in a straightforward way, allowing for easier data manipulation in C++ and CUDA. While this issue set us back by half a week, we believe the more convenient input file format to the matrix generation algorithm will allow for cleaner code, easier optimization and smoother analysis later on.

Another setback was determining the proper size of the probability matrices. Larger matrices would provide more detailed probabilities, and possible generate better sounding music, but be very hard to work with and take up a lot of storage. Since we are casing on both note length and note tone, of which there are already about 32 common lengths and 12 note tones per octave, this increases the matrix size very quickly. Ultimately, we decided on a 2-layer matrix (i.e. we case on the last 2 notes to determine the next note) to balance between musical sound and storage space. To store such matrices for later use, we decided on the most straightforward storage method - text files, since they were easy to read and also were going to be small enough to reasonably reuse in another run of the program.

Finally, we had to discuss many implementation choices when writing the music generation algorithm. Originally, we believed it would be straightforward to assign every part in the music to a different thread, and allow threads to compute their assigned parts with their assigned matrices. We noted the complications with needing a conductor to determine tempo, chord interpretation, key signature, and other factors, but did not deeply ponder these issues. When faced with actually implementing the algorithm, we needed to determine how to set these global settings and how store the generated music as proper MIDI messages. Ultimately, we decided to allow user input for these global settings, as this allows for more personalized music creation, and the settings are static. We still need to discuss more on how to properly format the generated music in CUDA and translate between CUDA data structures and MIDI messages, as there are many trade-offs. Therefore, parallelization of this algorithm will take a little more time.

Updated Goals and Deliverables

Overall, we now have a much clearer picture of the real-time video game music generator that is our ultimate goal. Our matrix training algorithm will still train multiple probability matrices in parallel over a large collection of files in the video game music genre, with significant speedup over sequential algorithms. In addition, we have finalized a total of 20 such matrices, split by theme (of which there are 5) and part (4-part harmony). As stated above, the training is based on a large collected of input files, which have been preprocessing by other algorithms from their original MIDI format. The training is based heavily in matrix sum operations and in particular atomic increments into cells, where different threads may attempt to increment the same cell. It must also calculate the sum along rows of the matrix. Therefore, we still expect similar speedup goals as in general matrix parallel sum algorithms. In order to get a clear picture of this speedup, we will not include the pre-processing times when comparing to general matrix sum algorithms (although we will look at it in to see what dominates the computation time).

For the second component, in musical synthesis, we still plan to see significant speedup in the generation of multiple melodic lines. In the best case, this speedup would be linear in the number of melodic lines, although the many dependencies noted above may result in this being unlikely. We have fully planned out the algorithm for such generation, and have a sequential version. However, we still need to discuss how to properly parallelize the algorithm. We are still on track for this goal.

As for the focus on video game music, it is unlikely that we will change the algorithm to fit the genre. The music theory on the genre is deep and confusing, and it would take more time to understand than we have. In addition, we do not believe this would help the parallelization in the algorithm, which is the main idea.

The stretch goal on wide accessibility has become almost a necessity of our algorithm. In order to produce sound from the generated MIDI messages, we require a client, e.g. our own laptops, to play the sound, in addition to the server running CUDA that will do the actual music processing and generation. Therefore, we need to set up a server that can serve clients, most likely through a web browser. Since we are implementing a client interface, we have decided to give the client the ability to customize more features of the music they want generated, such as:

- Category (Basic, Happy, Journey, Melancholy, Tension)
- Orchestration
- Tempo
- Length

Another stretch goal would be to demonstrate the feasibility of using parallel computation for video game music in industry. Namely, generating video game music in real time along with gameplay. However, with all the setbacks we have already faced and the complicated nature of music theory and MIDI file formats, we don’t believe there will be enough time to continue this analysis. Since the stretch goal of the client has become somewhat of a requirement to produce music, we will be focused on that goal instead, and set this one aside. Consideration of how we would extend our project to be used in a live gameplay context will be discussed in the final writeup.

List of Updated Goals

1. A parallel matrix generation algorithm, which takes in input files of our newly designed format and produces 20 probability matrices that will be used in music generation.
2. A significant speedup of the matrix generation algorithm, rivaling that of the general parallel matrix sum algorithms
3. A parallel music generation algorithm, which uses the matrices generated to produce music in real time
4. Significant speedup of the music generation algorithm over its sequential counterpart
5. A basic Server-Client interface which allows the client to pick global settings and play the music generated by the Server
6. In-depth analysis of the speedup of the above algorithms, both with and without the pre-processing time
7. Working Demo that can play music on presentation day!

Updated Demo Plan

The demo will be the same as originally stated. There will be speedup graphs that showcase the increased performance in utilizing GPUs over using sequential code. In addition, we will have live music generation during the demo that showcases our fast, parallel algorithm still produces music that retains harmony. The combination of melodious music and speedup graphs shows our understanding of the balance between dependencies in music and our parallel algorithm. In addition, we will allow people to try out our client interface that can request new music to be generated from the server. The various settings that can be tuned on the client interface will demonstrate the versatility of our algorithm, and the real-time demo will show the practicality and speed of our algorithm.

Remaining Challenges

Since the sequential algorithms have been implemented, much of the remaining work lies in the optimization and parallelization of working algorithms. After running into the implementation challenges from formatting in the first week, we planned out all of the algorithms for matrices and music generation in detail. Therefore, it should just be a matter of coding and debugging to get working parallel algorithms. In addition, we already have many ideas to optimize our algorithm - for example, for matrix generation, we realized there is synchronization when threads attempt to increment the same matrix cells. This seems like a `lock` vs. `atomic_add` type of problem, for which we can test different solutions to resolve. For music generation, we note that threads don’t necessarily need to synchronize with each other, as long as the overall server can match up times - this can lead to many ways to reduce synchronization costs between threads and the main program. Since we have many ideas on optimization, we believe the speedup goals for our algorithm are reachable with just more thought and time.

The main concern is the server-client interface. While we can store MIDI messages and other information in data structures on CUDA or in our C++ programs, we have never tried to send this information out through a web connection to another device. We are also not sure how to get our laptops to receive real-time MIDI messages and translate them to music. We imagine we would only need a simple message transfer system, but we have to worry about considerations such as how we might want to reduce buffering, latency between packets, packet length, packet design, communication protocol, ... the list of factors to design and test out is plentiful and will present a challenge to us when we try to implement our server-client interface.

However, we have thought of some possible architectures to use for the interface, for example using a receiving buffer on the client end for the incoming server packets. To reduce latency, we were going to use some UI tricks and buffering to make sure that the client always has data to read from. As for playing the music, we have looked into Tone.js, which looks like a very robust, easy to use and thus promising library to use on the client side for playing music.