Parallel Programming Final Report
Parallel Music Generator

CHANG, CHIA-CHI 509506014
Department of Electrical and Control Engineering
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
maxchang.ee09@nycu.edu.tw

CHANG, CHIH-TING 509557023
Department of Computer Science
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
ginachang.cs09@nycu.edu.tw

1 ABSTRACT

Music plays an extremely important and indispensable role in life, games, art performance, etc. If there is no music, many creations and life will no longer be interesting.

Therefore, in recent years, people have successively proposed different music generation methods, include using AI to predict and automatically generate random melody music. One of the methods is to use Markov chains AI model to randomly select and generate the next musical melody through probability distribution.

We use the Markov chain AI model combined with parallelization related technology to randomly generate music, and then use WebSocket to transmit the obtained music melody to the client and play the music.

This report will show how we used parallelization methods: (1) CUDA(2) Python multiprocessing to generate music, and compared with sequence execution, about how the speed up, parallelization processing time analysis, and other experiment results.

2 INTRODUCTION

Traditionally, music was treated as an analogue signal and was generated manually. It is a composition through the elements of melody, harmony, rhythm, and timbre [10]. By the growing of digital market such as Youtube, video games, advertisement, music generation has became more and more popular. People aim to increase effectiveness of music and earn more profit from it [6].

In 2009, Microsoft developed Songsmith, a software that generates musical accompaniment to match a singer's voice. In 2011, SuperWillow, a music generation system, is developed and is capable of producing two-instrument music pieces [11]. All these examples illustrate capability of computer to create melodious music. Through years of study and research, there are several methods in the field of music generation and Markov Model is one of the most.

Markov chain is a very powerful model for time series problems. There are a lot of applications of Markov chain gaining success like natural language process (NLP), reinforcement learning...etc. Same as natural languages, we may think music as a sequence of musical notes and so utilize Markov model to generate music.

3 PROPOSED SOLUTION

In the part of generating music, we choose to use the Markov chain model to generate the probability distribution matrix dynamically. Because there are many related libraries for Python music processing , so we decide to use Python to convert the midi file, and use the converted music information to training Markov model. Markov chain model need a large number of matrix operations. Matrix-Matrix is particularly suitable for computing with GPU, because GPU has a large number of simple and repetitive computing units, which can maximize computing efficiency through parallelism[2]. So, we use CUDA to calculate the probability distribution of the matrix, and the corresponding probability distribution generates different melody.

In order to generate music in parallel, we decide to use Python multiprocessing to implement. Python can not only use multiprocessing tool to parallel generate music by using melody files that generated from markov model, but also can set to be a server, it can help our communication by using WebSocket between the server and the client. WebSocket support user can both send and receive messages in real-time, and allow for a higher amount of efficiency compared to REST because they do not require the HTTP request/response overhead for each message sent and received[4]. We use React framework to implement client, and use tone, a Web Audio framework for creating interactive music in the browser, so that client can play each note with its tone and tone synthesizer.

4 EXPERIMENTAL METHODOLOGY

We first collected thousands of game music midi files. Game music is well known for its multiple melodic lines, repeatedly structure and limited polyphony. For example, only three notes can be played simultaneously on the Nintendo Entertainment System [3]. Those characteristics make it a good start point for parallel programming. Notation , chords, pitch and duration information is extracted from midi files and converted to text files by using Python and music21 library. The output text files contain around 10 million notes and will be used for constructing Markov matrices.

To generate more harmonic and sophisticated music, total six Markov matrices are trained for both major and minor mood: (1) high melodic line, (2) low melodic line and (3) chord. For melodic line matrices, notes are presented as (tone, duration) pair, where tone is music pitch ranging from 0 (C2) to 71 (B6), 72 is used for rest. The duration present how long a note is played and maps to an array containing 15 different rhythms, such as sixteenth notes, quarter notes, and whole notes. For chord matrices, chord is represented in base 12 and each chord is composed of three different tones. Duration information is not used because chords are usually indicators of mood in music.

Markov matrices are constructed by counting the number of transition states from one note to another. Due to limited storage space, the matrices are two dimension which means current note is only related to previous note. The rows are indexed by previous note and the columns are indexed by current note. We store these

matrices in row-major order, figure 1 illustrates transitions counting in Markov matrix during training phase.

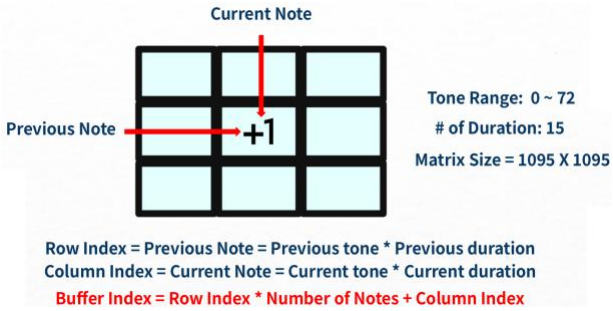

Figure 1. Markov matrix representation during training phase

During training phase, notes in text files are loaded into memory and stored in a buffer as a sequence input first. Then we allocate host and device memory for transferring buffer and Markov matrices data. To better understand the performance difference between pageable and pinned-memory allocation, we programmed and tested time required for both scenarios. Hidden states counting are done by GPU in parallel. Each thread handles portion of data in buffer and calculates transition states. To avoid possible data race when threads try to access elements in Markov matrix, atomicAdd() operation is applied to update counters on corresponding cells. After completion of Markov matrices training on GPU, data are copied back to host and saved as text files in row-major for music generation, which is done by using CPU.

In music generation step, we generated 10 melodic lines in parallel. Python multi-processing module is chosen because music generation is a CPU-bound process and GIL makes it's not suitable to use multi-threading. Markov matrices are converted to probability distribution for each row and stored as .npy format for faster loading. Algorithm for generating is almost the same as training but using current note to predicting next note, as shown in figure 2. The rows are indexed by current note and we randomly picked next note based on weighted probability. Each process will get a copy of Markov matrix needed and then generate some number of measures of music.

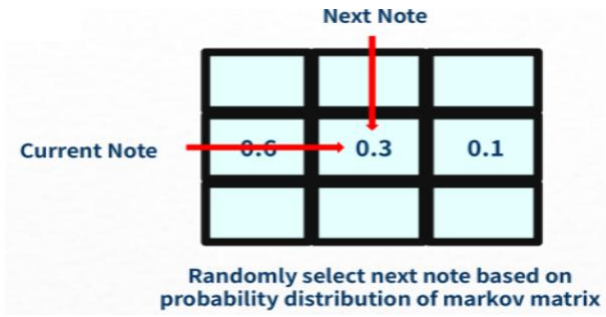

Figure 2. How notes are picked during generation phase

On the client side, we use ten tone combination music ensemble synthesizer, including piano, violin, xylo, trumpet, tuba, electronic synthesis. Each tone or chord will randomly get one of ten ensemble sounds to play, this can make the tone sound more diverse. We have designed two tunes corresponding to the mood, the one is major tune that corresponds to the happy mood, and the other is minor tune that corresponds to the sad mood, each mood will be displayed on the UI with a corresponding icon. First, users can choose major or minor music melody according to their current mood. Then, this request will send to server side, when server received message, it will use the method described earlier section to generate music. And last, server side would send the generation music back to client side, when client side received music, it will automatically play the music.

As stated, there are two portions of proposed solution could be parallelized: (1) Markov matrices training and (2) music generation. For Markov matrices training, some portion of the procedure are inherently sequential. In order to get a better understating of how parallel impact running time of training, we isolate these portions, like file parsing, and time spent will not be taken into account. Besides, we'd like to know runtime of each portion in parallel implementation, so not only time of GPU computation but data movement between GPU-CPU is also collected to help identifying which portions might be bottle-neck of our algorithm. To be more clear, we list what tests will be done as below:

1. Time comparison for sequential and CUDA training, for CUDA we test pinned and unpinned memory allocation.

2. Following 1., for each computation model we test time of (1) host memory allocation, (2) device memory allocation, (3) copy data from host to device, (4) markov training, (5) copy data from device to host and (6) free memory.

We test speed up of music generation through multi-processing. Time needed for loading Markov matrices into memory will not be counted to isolate portion that could not be done parallel. To compare efficiency of parallel computing, we generate 10 melodic lines in both sequential and parallel and measure total time spent for each model. The 10 melodic lines are [0, 0, 1, 1, 2, 2, 0, 0, 1, 2] where 0 means chord line, 1 means low melodic line and 2 means high melodic line.

We used two machines for our project:

1. Workstation provided by this course for Markov matrices training

2. HP laptop based on Ubuntu 20.04 with Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz for music generation

5 EXPERIMENTAL RESULTS

Markov training. In figure 3, total run-time of training phase, including both memory related operation and transitions counting, is illustrated. Sequential model performs much better for all test sizes in our experiment. There might be a lot of possible factors that lead to this result and we'll discuss later.

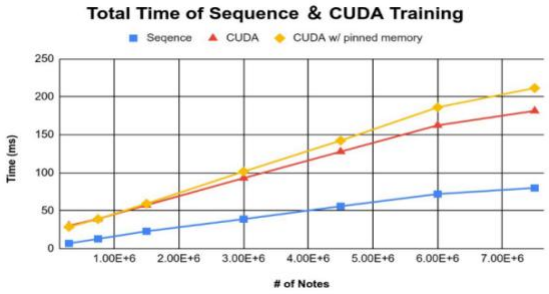

Figure 3. Total running time of Markov matrices training for each computation model

Our training algorithm is O(N) and thread number is fixed at 1024 between runs. As we expected, run-time for both sequential and parallel increase linearly as problem size becomes larger. But there is no beneficial from parallel computing as in figure 4.

As we know, game musics are usually in minor mood and the ratio of number of notes between minor and major mood is around 6 : 4 in our case. For smaller problem size, data balancing is available to simply ensure same amount of notes for both minor and major moods. But when problem size goes up, it is limited by major mood. In order to not affect hidden states of Markov matrices, we decided not to re-sample and use the ratio through all tests. This might cause run-time is dominated by training minor matrices.

The other problem is that we are not able to predict what comes next, as music is a sequence of transition states. And therefore whether the coming data is a note or chord is decided online. This incurs a potential issue of divergence because the if-else condition is probably highly unbalanced. In addition, even though music structure is complicated, the combination of possible transitions is limited which results in very sparse matrices. Most of time a row of data picked will not be used for next step and causing low locality. Both divergence and locality issues further harmed efficiency of parallel computing.

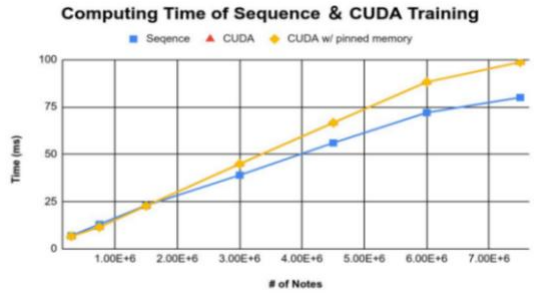

Figure 4. Training time for each computation model

Memory related issues. Memory related operations between GPU and CPU also contribute a large portion of execution time. As in figure 5. time of data transferring for pinned memory is lower comparing to unpinned memory. We look forward that the better performance of copy data will reduce time required for GPU training. But it brings overhead issues for memory allocation and free. Beyond our expectation, memory allocation and free time for pinned memory is much higher than the others. As problem size increases, the gap is widened and advantage of using pinned memory is diminished. The costs of hardware overhead make the pipeline of using pinned memory suffer from setup time and result in a slower implementation than using unpinned memory. This is the reason why execution time of pinned memory in figure 3 is always the highest.

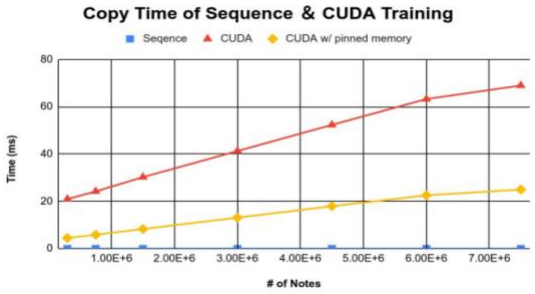

Figure 5. Data copy time for each computation model

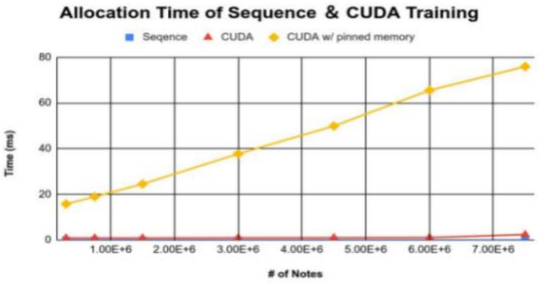

Figure 6. Memory allocation time for each computation model

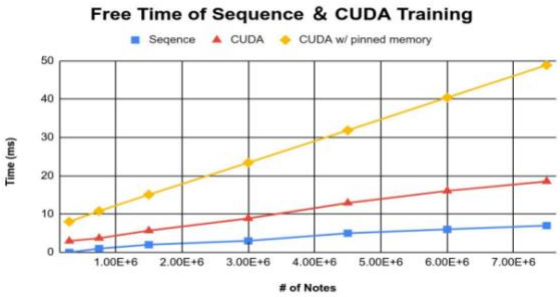

Figure 7. Memory free time for each computation model

To get visualization of time ratio for each part in training pipeline, we draw a pie chart as figure 8. It's surprisingly that data transfer of unpinned memory model takes 44% of total time. For pinned memory model, time of memory allocation and free is even higher than computation time. All of these illustrate that the problem size might not be big enough to take advantage from parallel computing and ignore hardware overheads. Execution time of each portion are collected in Table 2. in last page.

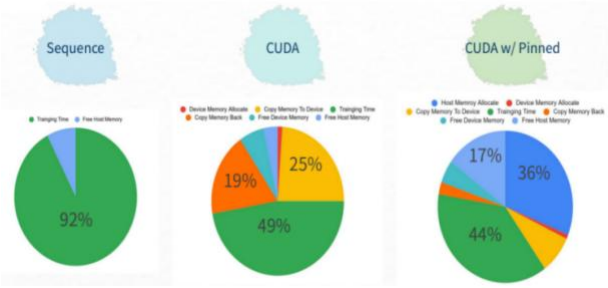

Figure 8. Time ratio distribution for each computation model

As in figure 9,initially when problem size is smaller than 5000, execution time for multi-processing is higher because of overhead of creating new process. As we increase number of beats generated to 200,000, the speedup is at it's pick around 2.5X. After that, the speedup decreased gradually and finally fixed around 2.2X. For multiprocessing, data in each process is not shareable and so when each process completed it's generation, music beats have to be passed between processes. Communication costs cause high overhead especially when problem size is large. Besides, music generation is also suffered from low locality and sparse matrices. Besides, if number of processors is less than 10, which means some of the processors will be assigned more than one task and execution time will be dominated by those processors. All above are reasons why it's impossible to get theoretical speed up and caps around 2X 2.5X.

An interesting thing was found when we tested our multi-processing implementation. If we don't add a random seed generator in function which computes next notes, the music created by different processes which use the same Markov matrix will be identical . It's because in UNIX like system, the parent process uses os.fork() to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic[1]. We illustrate this phenomena in table 1 by generating notes 1 million times and take last notes of 10 melodic lines. when random seed generator is not used for multiprocessing, the notes and duration computed are equal for the same kind of melodic line as shown in last column of table 1.

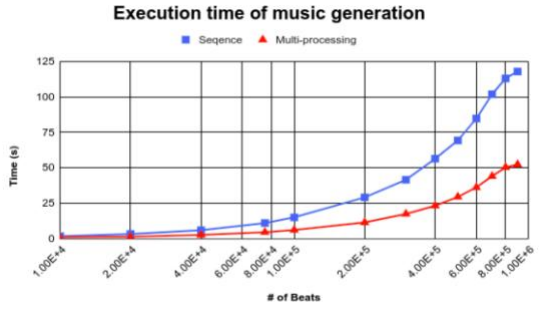

Figure 9. Execution time of music generation for sequential and parallel model

Table 1: Last notes generated for sequential and multi-processing model after million times

Sequence	Multiprocess w/ seed	Multiprocess w/o seed
[1655, 7]	[198, 4]	[176, 7]
[108, 0]	[121, 1]	[176, 7]
[42, 0]	[50, 9]	[36, 0]
[50, 1]	[48, 2]	[21, 0]
[24, 2]	[1492, 11]	[176, 7]
[72, 2]	[673, 3]	[36, 0]
[153, 8]	[25, 1]	[176, 7]
[127, 4]	[16, 1]	[21, 0]
[42, 2]	[53, 1]	[36, 0]
[5, 2]	[32, 0]	[21, 0]

6 RELATED WORK

Yi-Wen Liu (2006) proposed that music can be thought of as random processes, and music style recognition can be thought of as a system identification problem. A general framework for modeling music using Markov chains is described, and based on this framework, a two-way composer identification scheme is demonstrated [9].

Chuan Liu (2009) implement a prototype program for Hidden Markov model(HMM) training and classification on the NVIDIA CUDA platform. The evaluation shows CUDA implementation achieves performance of 23.3 GFLOP/s and has up to 800× speedup over the implementation based on single core CPU; and for Baum-Welch algorithm, the performance is 4.3 GFLOP/s and there is also 200× speedup over CPU implementation[8].

Andries Van Der Merwe and Walter Schulze (2010) used music written in a certain style as training data, parameters are calculated for Markov chains and hidden Markov models to capture the musical style of the training data as mathematical models [12].

Zaky Hassani and Aciek Wuryandari (2013) aims to generate music for composer and for others at real-time. It reads MIDI file copliant to the Standard MIDI File (SMF) and outputs MIDI file too. The melody generator itself uses Markov Chain/Model. The process is to read existing musics as samples, the program will calculate parameters of those musics and with those calculated parameters, a melody will be generated [7].

William Clinton (2019) used the principle of Markov Chains in the Python system that the music being inputted for improvisation was create in the MIDI format which could be processed by Python using the 'Mido' library. Processing music in this fashion immediately adds automation to the method as any random MIDI file can be read and analysed in the same way [5].

7 CONCLUSIONS

We create a music generator from scratch by using Markov matrices. The whole architecture consists of pre-processing, Markov matrices training, music generation and interacting with clients. We explored different parallelism techniques to optimize our algorithm, involving CUDA GPU training and Python multiprocessing module. For CUDA, execution time of each portion of training pipeline was tested to identify bottle neck of GPU parallel computing. We found that high hardware overheads exist when allocate and free memory of pinned memory model. Also, the problem size and potential

unbalanced data and if-else condition make it even harder to get benefit from GPU SIMT structure. Low locality and sparse matrices are other issues further decrease efficiency of GPU training. Time spend for Markov matrices is dominated by memory related operations. For music generation, overhead of new processes creation is significant when problem size is smaller than 5000. The speed up caps around 2X 2.5X for problem size larger than 20,000 because of high communication costs between parent and child processes and not evenly assigned tasks for each processor. In UNIX like system, child process is a fork of parent process and so random seed generator is required.

REFERENCES

[1] [n.d.]. 3.9.5 Documentation. https://docs.python.org/3/.
[2] [n.d.]. *Matrix-Matrix Multiplication on the GPU with Nvidia CUDA.* https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/
[3] [n.d.]. Video game music - Wikipedia. https://en.wikipedia.org/wiki/Video_game_music.
[4] [n.d.]. *When to use a HTTP call instead of a WebSocket (or HTTP 2.0).* https://blogs.windows.com/windowsdeveloper/2016/03/14/when-to-use-a-http-call-instead-of-a-websocket-or-http-2-0/
[5] William Clinton. 2019. Music improvisation in Python using a Markov Chain Algorithm. (2019).
[6] Majid Farzaneh and Rahil Mahdian Toroghi. 2020. GGA-MG: Generative Genetic Algorithm for Music Generation. *arXiv preprint arXiv:2004.04687* (2020).
[7] Zaky Hassani and Aciek Ida Wuryandari. 2016. Music generator with Markov Chain: A case study with Beatme Touchdown. In *2016 6th International Conference on System Engineering and Technology (ICSET)*. IEEE, 179–183.
[8] Chuan Liu. 2009. cuHMM: a CUDA Implementation of Hidden Markov Model Training and Classification. (2009).
[9] Yi-Wen Liu and Eleanor Selfridge-Field. 2002. Modeling music as markov chains: Composer identification.
[10] Sanidhya Mangal, Rahul Modak, and Poorva Joshi. 2019. LSTM Based Music Generation System. *arXiv preprint arXiv:1908.01080* (2019).
[11] monku1002. 2019. Music Generation with Markov Models. Retrieved May 31, 2009, from the World Wide Web: https://musicgeneration.home.blog/2019/04/30/music-generation-with-markov-models/.
[12] Andries Van Der Merwe and Walter Schulze. 2010. Music generation with markov models. *IEEE MultiMedia* 18, 3 (2010), 78–85.

Table 2: Execution time for each portion of music training

# of Notes	7500000	6000000	4500000	3000000	1500000	750000	300000
Sequence							
Host Memroy Allocate	0	0	0	0	0	0	0
Device Memory Allocate	0	0	0	0	0	0	0
Copy Memory To Device	0	0	0	0	0	0	0
Trainging Time	**80**	**72**	**56**	**39**	**23**	**13**	**7**
Copy Memory Back	0	0	0	0	0	0	0
Free Device Memory	0	0	0	0	0	0	0
Free Host Memory	7	6	5	3	2	1	0
Totol time	**80**	**72**	**56**	**39**	**23**	**13**	**7**
Parallel not pinned							
Host Memroy Allocate	0	0	0	0	0	0	0
Device Memory Allocate	2.4	1.1	1.01	0.95	0.88	0.88	0.86
Copy Memory To Device	50.46	44.68	33.87	22.63	11.49	5.69	2.28
Trainging Time	**98.67**	**88.17**	**66.65**	**44.93**	**22.72**	**11.51**	**6.67**
Copy Memory Back	18.54	18.56	18.45	18.62	18.72	18.49	18.63
Free Device Memory	11.54	10.05	7.91	5.87	3.67	2.7	1.95
Free Host Memory	7	6	5	3	2	1	1
Totol time	**181.61**	**162.55**	**127.89**	**93**	**57.48**	**39.28**	**30.39**
Parallel Pinned							
Host Memroy Allocate	73.56	64.38	48.84	36.72	23.57	18.09	14.95
Device Memory Allocate	2.37	1.13	1.08	1.01	0.93	0.88	0.84
Copy Memory To Device	21.58	19.19	14.57	9.71	4.94	2.48	1.08
Trainging Time	**98.74**	**88.18**	**66.67**	**45.08**	**22.76**	**11.53**	**6.6**
Copy Memory Back	3.35	3.35	3.35	3.35	3.35	3.35	3.35
Free Device Memory	11.99	10	7.87	5.87	3.77	2.71	1.95
Free Host Memory	36.92	30.42	23.99	17.55	11.3	8.07	6.06
Totol time	**211.59**	**186.24**	**142.37**	**101.73**	**59.32**	**39.04**	**28.77**