An Effective Approach for Vocal Melody Extraction from Polyphonic Music on GPU

Guangchao Yao, Yao Zheng, Limin Xiao, Li Ruan, Zhen Lin, Junjie Peng

An Effective Approach for Vocal Melody Extraction from Polyphonic Music on GPU

Guangchao Yao[1,2], Yao Zheng[1,2], Limin Xiao[1,2], Li Ruan[1,2], Zhen Lin[1,2], Junjie Peng[3]

1 State Key Laboratory of Software Development Environment,
Beijing 100191, China
2 School of Computer Science and Engineering, Beihang University,
Beijing 100191, China
3 School of Computer Engineering and Science, Shanghai University,
Shanghai 200444,China
ruanli@buaa.edu.cn

Abstract. Melody extraction from polyphonic music is a valuable but difficult problem in music information retrieval. The extraction incurs a large computational cost that limits its application. Growing processing cores and increased bandwidth have made GPU an ideal candidate for the development of fine-grained parallel algorithms. In this paper, we present a parallel approach for salience-based melody extraction from polyphonic music using CUDA. For 21 seconds of polyphonic clip, the extraction time is cut from 3 seconds to 33 milliseconds using NVIDIA GeForce GTX 480 which is up to 100 times faster. The increased performance allows the melody extraction to be carried out for real-time applications. Furthermore, the evaluation of the extraction on huge datasets is also possible. We give insight into how such significant speed gains are made and encourage the development and adoption of GPU in music information retrieval field.

1 Introduction

In the field of music information retrieval (MIR), the extraction of acoustic features is always the first step. After extraction, the features are then utilized by further applications. Melody, the fundamental frequency (F0) contour of the polyphonic music's lead vocal [1], has been a remarkable feature owing to its numerous applications in the past few years. Having melodies, we can use them in many ways: the most attractive one may arise from the field of query by humming [2], where the melody fragment of humming as a query will be fuzzily searched in the feature database. Apart from that, they are also frequently used in singing voice separation, singer identification and extraction of musical structure, etc.

Unfortunately, the promising applications cannot hide the difficulty of the melody extraction from polyphonic music. The complexity of the task is twofold — firstly, due to the superposition of all instruments which play simultaneously, the accuracy of the extraction is still hovering at a relatively low level. This can be seen from the results of the melody extraction of Music Information Retrieval Evaluation eXchange (MIREX) [3] in recent years. Secondly, most methods submitted to MIREX are very time-consuming [1]. The inherent high computational complexity of audio signal processing is the chief culprits. At the same time, for better evaluation of the algorithm, the corpora is becoming larger and larger, which may contain as many as thousands (or even millions, in the case of some commercial databases) of audio files. All these factors hinder the development and application of the melody. The progressing rate of the extraction could be greatly improved if the execution time was reduced sharply enough to allow a quicker evaluation and tuning of algorithm parameters.

The solution to the above problem relies on the improvement of the computational capability. Some researchers [4] use FPGA to accelerate the extraction of the acoustic features. However, this method features a long developing cycle and is very difficult. Clusters are another common way to solve this problem. Marsyas is used to implement efficient distributed audio analysis algorithms [5]. But the construction of a cluster is expensive and inefficient. Besides the distributed parallel method, there are some on-chip parallel ones, i.e. multi-core and many-core methods. Owing to much more cores on many-core architecture than that on multi-core one, the former can provide a tremendous amount of computational capability than the latter. A typical many-core is GPU. The latest NVIDIA card, e.g. GTX 690 has up to 3072 cores. The availability of programming models such as CUDA has also made GPU a strong candidate for performing many computation intensive tasks.

The adoption of GPU has permeated almost all areas which require significant computational resources. Many applications ranging from general signal processing or physics simulation to computational finance or computational biology can be accelerated by GPU. Battenberg [6] uses the multi-core and many-core to accelerate the non-negative matrix factorization for audio source separation, and the result reveals that the many-core architecture has more advantages than the multi-core's. Specific to the extraction of acoustic feature field, Schmädecke [7] accelerates six different exemplary features, which are among the time domain, frequency domain, and on the autocorrelated signal. However, all these features are the most basic ones of music, not including the melody feature. Although some features can be used to compute the melody, they can only work for pure voice instead of polyphonic music.

Melody extraction from polyphonic music, as mentioned above, remains a challenging and unsolved task, the overall accuracy is around 70 % which is lower compared with melody extraction from MIDI files. Since classic features cannot estimate the melody accurately, more complicated approaches are proposed. There are some designs based on the source/filter model [8], which is sufficiently flexible to capture the variability of the singing voice and the accompaniment in terms of pitch range and timbre. Some use the salience-based method [9] to extract the melody. As for accuracy, the salience-based approaches give a better performance. But no matter which method is adopted, significant computational resources are necessarily needed. These arise from the nature of audio signal processing because the audio signal is usually partitioned into a large number of frames and the computation cost at every frame is high.

In this paper, we will use GPU to present a salience-based melody extraction approach to demonstrate the dramatic speedup achieved by GPU and to encourage MIR researchers to develop and reuse high performance parallel implementations of important MIR procedures. Actually, the system won't be built from scratch. For example, the extraction methods using Expectation Maximization (EM) can benefit from Kumar's work [10]. The methods using Support Vector Machine (SVM) will find it useful of Catanzaro's work [11].

In section 2, the salience-based melody extraction method will be described in more details. The GPU's hardware implementation, thread hierarchy and its different kinds of memories will be stated in section 3. Section 4 will introduce our melody extraction approach on CUDA. Section 5 presents the experimentation results on different GPUs and different aspects of melody extraction. Finally we propose concluding remarks and future work to be pursued in Section 6.

2 Salience-Based Melody Extraction

The salience-based design has a common structure: at first, get the spectral representation of the signal. The most popular technique is the short time Fourier transform (STFT). Secondly, use the spectral representation to compute the F0 candidates. There exist many different strategies to compute the candidates, [12] uses the harmonic summation of the spectral peaks with assigned weights, while [13] lets the possible F0 to compete for harmonics based on expectation-maximization (EM) model. At last, the melody is chosen from the candidate F0 using different methods.

The procedure to compute the candidates presents significant diversities, and our approach is based on the generation of melody contours [9]: firstly, the pitch salience at every frame will be computed by utilizing the spectral peaks, which is called multi-pitch extraction. Secondly, a set of pitch contours are created using the salient candidate pitches. Then the corresponding contour characteristics will be defined, which can be used to discriminate whether the contour belongs to the melody at the melody identification stage. At the last stage – post-processing stage, vocal melody is chosen out of all contours in a three-step singing voice detection stage with the help of contour characteristics. Fig. 1 outlines these three stages through a block diagram. More details can be found in our previous work [9].

For better comparison with the parallel melody extraction and identifying any potential performance bottlenecks, it's necessary to perform an intensive analysis of the serial approach. We will use a polyphonic music clip with duration 21 s to verify the necessity of acceleration. Table 1 shows the execution time of the main parts of the algorithm using Intel(R) Core(TM) i5 CPU 750. From the table, it can be easily seen that the extraction of melody from polyphonic music is so time-consuming that it can hardly be used in real-time applications because the extraction will be finished in seconds. For another, the parts occupying the majority of execution time can be easily found—sinusoid extraction and pitch salience on which most attention will be paid when achieving the parallel approach. The slowness of the sinusoid extraction arises from the relative high time complexity. By contrast, the calculation of pitch salience runs slowly because of its intensive floating point operation.

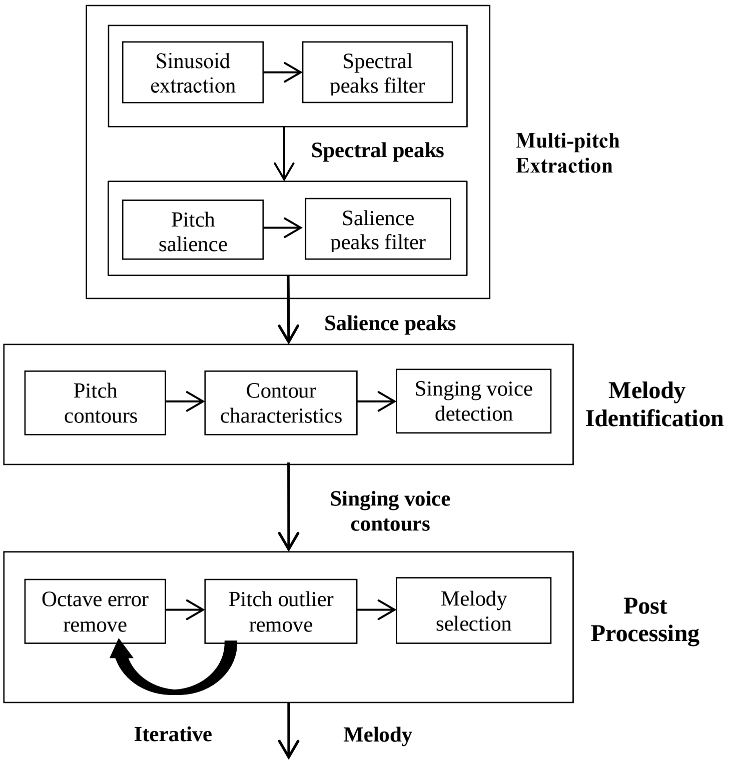

Fig. 1. Block diagram of our approach, it includes stages: Multi-pitch Extraction, Melody Identification and Post Processing. The output of first stage is salience peaks which are used to construct the contours in second stage. At the last stage, the melody is generated.

Table 1. Execution time of every part (ms).

	Sinusoid extraction	Spectral peaks filter	Pitch salience	Salience peaks filter	Melody identification	Post processing	Total time
Running time	307.8	9.7	2611.8	4.1	0.2	7.7	2941.9
Proportion	10.5%	0.3%	88.8%	0.1%	0	0.3%	1.0

3 GPU Programming Model

For higher throughput and better organization of so many cores, NVIDIA has done a lot of innovations on its GPU. The first to be mentioned will be its hierarchy programming model, which provides more choice at choosing different parallel granularities. Additionally, the diverse device memories are also very important in GPU programming. Each has its own special property and if used properly, they will bring an obvious promotion to the performance.

3.1 Programming Model

The NIVIDA GPU hardware is implemented as a set of streaming multiprocessors (SM), which manage a set of scalar processor cores (SP). The SPs at the same SM are equal peers and share some important memories. CUDA has a special programming model catering the hardware—Grid-Block-Thread model, i.e. the simple instruction multiple thread (SIMT) model. Users must group threads into blocks and construct a grid of some number of thread blocks. A block will be executed on a single SM. If all the threads in one block finish their job, a new block (if any) will be activated on the idle SM. One or more thread blocks to be processed by an SM are partitioned into warps with a size of normally 32 threads. These warps are then scheduled by the SM schedule unit for its execution. All the threads activated at a SM can be larger than the actual SPs for hiding the memory latency. Furthermore, threads in the same block can be synchronized for correctness.

3.2 Memory Hierarchy

In Table 2, the frequently-used memories and their properties are listed. Texture memory and local memory are rarely used under the circumstances of general purpose computation, so they are not included in the table. Actually, as long as we make good use of the memories listed in the table, the best optimization effect can be achieved.

Registers scattered on the SPs have the fastest memory access speed. But they can only be used by a single thread and cannot be used in inter-thread communication. For communication between threads, shared memory residing in the SM that is shared by the whole threads in one block can be used. Owing to its on-chip property, it also has a fast access speed. This makes it a key point for acceleration. It can be regarded as the cache, small but fast.

Besides the above, others are the off-chip device memories which often have a relative slow access speed. The global memory owns the biggest capacity which usually has several GBs, and its bandwidth is also higher than the bandwidth of CPU's memory. From compute capability 2.0 and above, the on-chip cache makes it more efficient. Furthermore, it's shared by all the blocks. Constant memory, as its name suggests, is used to save the constant variables. As it has on-chip cache, its access speed is very fast. From [14], the constant cache often has higher bandwidth than shared memory in spite of the off-chip property. This conclusion inspires us to use constant memory to save read-only variables.

Table 2. Properties of the frequently used memories.

Name	Action Scope	Speed	Cached
Register	Thread	Fastest	N
Shared Memory	Block	Fast	N
Global Memory	Grid	Slow	Compute Capability 2.0 and above
Constant Memory	Grid	Slow	Y

After the introduction of GPU, conclusion can be drawn that GPU is very suitable for problems which have a two-level data parallelism—one level for different blocks, and another level for different threads. If a problem doesn't show a two-level parallelism, it can also be applied on GPU as long as it has at least one-level parallelism but maybe need more tricky works. For example, the classic matrix multiplication problem must be fragmented to small square matrices to create the two-level parallelism factitiously. For another, merging two sorted arrays [15] must use more difficult techniques to transplant the problem into GPU's platform. Fortunately, in the area of audio signal processing, the problems often show a great parallelism. For an independent music, it's usually processed by frame. The inter-frame is one level parallelism at which the weak scaling [16] can be applied, and if the problem can also be processed parallel inner-frame, this is the other level parallelism at which the strong scaling will be applicable. If a frame cannot be processed parallel, we can still get one level parallelism from different music. In a nutshell, MIR is a perfect field that GPU can show its extraordinary computing capability.

4 Parallel Implementation

From Table 1, the parts which most need acceleration can be easily seen. Fortunately, these parts are also the easiest parts to be parallelized. At the first stage—Multi-pitch Extraction, the main job is to transform the music data and to calculate the pitch salience. In sinusoid extraction part, CUFFT [17] is a good choice to achieve the FFT on CUDA, which is an efficient official library. In pitch salience computation part, the key operation is to calculate the salience of different frequencies at every frame. Because the salience computation of a specific frequency has no contact with the others, the computation reveals a perfectly strong scaling which can be mapped to different threads in a one-to-one mapping. Besides these two time-consuming parts, there will be two filter parts. They occupy only a small proportion of the runtime, so they can be put on CPU. But considering their stay between the sinusoid extraction and the pitch salience computation, parallelizing them on GPU will reduce the communication time between CPU and GPU. Hence we will parallelize all parts of the first stage on GPU.

In Melody Identification stage, the main job is to create a series of pitch contours and to calculate the contour characteristics. These operations have a strong data dependency. What's more, this stage occupies only a very little proportion of the whole runtime as illustrated in Table 1. So this stage will be finished on CPU.

In the last post processing stage, the operation deals with the octave error and the pitch outlier using "melody pitch mean". The calculation of melody pitch mean is finished with the help of smoothing filter which needs a lot of computation. This filter has less data dependency and it can be parallelized on GPU even the proportion of this stage is also small. Different blocks will smooth different positions, and the threads in the same block will calculate the value of the same position using reduction.

The complete heterogeneous algorithm is shown below (Note that the first stage is included for completeness). In the following section, we will focus on specific parts of the algorithm as some stages execute on CPU.

```
Algorithm    Hybrid Melody Extraction from Polyphonic Music
Input: Polyphonic Music pm
Output: Melody m
Stage I: Pre-Processing    CPU ::
    CPU :: read the polyphonic music to CPU memory
    CPU → GPU :: transfer the music to GPU memory
Stage II: Multi-pitch Extraction    GPU ::
    GPU :: rearrange the music data
    for every frame f parallel do
        GPU :: sinusoid extraction
        GPU :: spectral peaks filter
        GPU :: pitch salience computation
        GPU :: salience peaks filter
    endfor
Stage III: Melody identification    CPU ::
    GPU → CPU :: read salience peaks to CPU memory
    CPU :: pitch contour
    for every contour c do
        CPU :: contour characteristics
        CPU :: singing voice detection
    endfor
Stage IV: Post Processing mainly on CPU ::
    for i=1:3 do
        CPU → GPU → CPU :: melody pitch mean
        CPU :: remove octave error
        CPU :: remove pitch outlier
    endfor
    CPU: return the melody
```

4.1 Sinusoid Extraction

Before transforming, the music data should be transferred from host memory space to device memory space, and then rearranged to fit for FFT and consequent operations, as depicted in Fig. 2. The reason for rearrangement is to make the data suitable for CUFFT functions. The rearrangement is executed after the transfer because less data will be transferred and the rearrangement can be implemented in parallel.

Afterwards, the rearranged frames will be multiplied by Hann window. Then cufftPlan1d and cufftExecR2C are used to transform the frames. The results of FFT will be complex. The real part and the imaginary part of complex are used to get the module of the transform result. At the same time, as transform is symmetric, only half of the result at one frame will be useful.

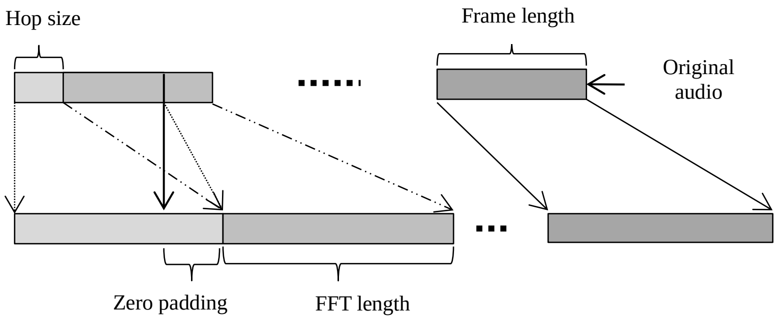

Fig. 2. Rearrangement of the original audio (For music with 44.1 kHz sample rate, frame length is 2048, hop size is 441, FFT length is 8192, zero padding is 6144)

In this part, many operations just transform the frame data from one form to another, so the pattern concurs with each other—transform the data one by one in every frame. Therefore, we adopt the same strategy to parallelize them: all the threads in one block will deal with an individual frame and different blocks will process different frames. This is a good example of strong scaling and weak scaling.

4.2 Spectral Peaks filter

The spectral peaks of transform encompass the possible pitches and are extracted for further dispose. The key of the filter is getting all the peaks first. For traditional serial algorithm, it's a very easy goal to achieve as it just traverses the array forward and returns the found peaks. When transplanting to GPU platform, the complexity appears. Although different frames can be processed in parallel, the parallelism in a frame cannot be utilized well. The difficulty is not how to find the peaks but how to save them. If merely counting the numbers, the different threads in a block can count the peaks on shared memory and count them all using efficient reduction method. But if we need to save the peaks at the same time, the story will be different. There is a possibility that some threads in a warp find the peaks simultaneously and the conflict happens when saving them. The problem here is that counting variable will be visited simultaneously by different threads in a block. One solution relies on the use of atomic operation which serializes the visit of memory. Unfortunately, this solution has a severe drawback—reducing the access efficiency, so does the speed.

In order to balance the efficiency and the correctness, different strategies are adopted to specific length of frames. For the original transform result, the frame length is half of the FFT length plus one, i.e. 4097. The order of magnitude of the peaks will stabilize at 10^2 levels. For the sake of high efficiency, the "space for time" strategy is used. More specifically, the equal sized array with the original frame is allocated for holding the peaks, as depicted in Fig. 3. The found peaks will be put in position which has the same index with the peaks in the original frame. In this way, the threads can save the peaks without the access conflict. The reason we waste the space to save the peaks is that the peaks will be further disposed, and none of all will be saved.

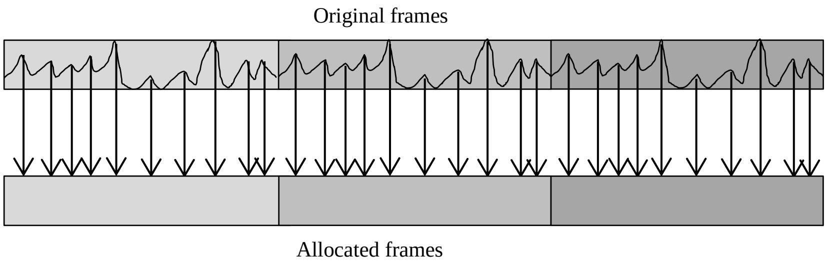

Fig. 3. Space for time strategy for finding the peaks (All elements in allocated frames are zero except the peaks)

The consequent work is to find the max peak which is used to carry out the filter. Finding the max value of an array is a classic problem on CUDA which can be finished using the CUBLAS [18] library or achieved by ourselves using reduction. Once getting the max peak, we can perform the filter of peaks. The problems here are the same as finding peaks, namely how to save the remaining peaks. But unlike the previous finding peaks, the order of magnitude of the peaks has reduced to 10^1 levels from experiment, so this means the remaining peaks in the array is rather sparse (length of thousands of array has dozens of peaks). From the above result, the probability of conflict when the threads in a warp save the peaks simultaneously will be low. In this case, we can use atomic operation to finish the filter.

Although the atomic operation assures the correctness of counting, the result of atomic operation cannot be used directly as it may be added by another thread before saving the peaks. The solution is using the return value of the atomic function of CUDA. Because it returns the old value and incrementing the old value will exactly be the right position for the remaining peak. In addition, owing to the fast access speed and shared by the whole threads in a block, the shared memory is a good place to put the counting variable. The part of salience peaks filter is the same as the spectral peaks filter, so it will not be described in detail.

4.3 Pitch Salience

After filtering the spectral peaks, they will be used to calculate the pitch salience. The calculation reveals a perfect parallelization as the calculation of each bin has no data dependency with other bins' calculation. So it can be easily transplanted to GPU platform, the strategy is different blocks process different frames, and threads in a block dispose a frame's calculation of 480 different pitch saliences. On one hand, due to the frequent access of spectral peaks and their small size, putting them in shared memory is a good idea to accelerate the access speed. On the other hand, because the calculation of pitch salience has so many floating point operations, the requirement for registers are huge and this will limit the threads number activated at the same time. Our solution is to use the constant memory to hold the constant variables needed in calculation. This is a good approach to optimize. The other used optimizations comprise the extraction of common sub-expressions and loop unrolling, etc. Although no advanced techniques are adopted, we can still achieve a high speed-up from the following evaluation.

5 Evaluation

Various experiments are presented to clarify the effectiveness of GPU in our approach. At first, the overall performance of the system and the parallelized parts on each hardware platform are presented. After this, the performance along with the music length is evaluated. Furthermore, the efficiency of our peaks filter is demonstrated. At last, the influence to the ultimate accuracy is tested.

We have implemented our algorithm on two different commodity GPUs, whose hardware specifications are listed in Table 3. The CPU version runs on an Intel(R) Core(TM) i5 CPU 750 platform.

The corpus used to evaluate the efficiency of the system is from MIR-1k which is a publicly available dataset proposed in [19]. It has 1000 song clips with a duration ranging from 3 to 12 seconds, and the total length is up to 133 minutes.

Table 3. GPU Specifications.

Properties	**GTX 285**	**GTX 480**
Number of cores	240	480
SMs	30	15
Cores per SM	8	32
Global Memory(GB)	1	1.5
Memory Bandwidth (GB/s)	159	177
Shared Memory (KB)	16	48
Register (KB)	16	32
GPU core clock rate (MHz)	648	700
Compute Capacity	1.3	2.0

5.1 Overall Performance

The same polyphonic music clip as in performance analysis of serial approach is used to measure the overall performance of the system and the parallelized parts. The runtime of the parallelized parts and the whole system on different platforms is demonstrated in Fig. 4.

From Fig. 4, we can see all accelerated parts achieve a positive speed up, especially two most time-consuming parts. In the pitch salience computation part, the runtime is reduced from seconds to lower than 10ms. This is a huge acceleration. The high acceleration can attribute to the perfect parallelism and the use of shared memory. Inspired by these two parts, the whole system also gets a considerable acceleration—for a length of 21 s audio clip, the extraction time is reduced to less than 40 ms on GTX 480. The speedup is nearly to 100 times. This implementation is sufficient for real-time applications, such as query by humming. What's more, our system can be applied to massive datasets which are often used to verify the effectiveness of algorithms. The approach can reduce the time to tune the variables or to develop new methods dramatically. So researchers can pay more attention to the algorithm itself rather than the performance.

Except the two parts mentioned above, other parts get a relative small acceleration. Even though they occupy a small proportion of execution time, it's meaningful to find out a reasonable explanation. In post processing stage, a 2-times acceleration is achieved approximately on GTX 285 platform. The final cause is shown in two aspects: at first, the proportion which can be accelerated is only 70 %. This means the speed-up upper bound is only 3.3 times according to the Amdahl's law. Secondly, the algorithm complexity of smoother filter is only *O(n)*, so the ratio of operations to elements transferred is *O(1)*. Performance benefits can only be more readily achieved when this ratio is higher. Furthermore, although a high parallelization exists in the smooth filter, the time is wasted on the space allocation on GPU and transfer between the host and the device. So we can reach such a conclusion that a high proportion of parallelizable part and computation complexity are the prerequisites for high speed-up. The reason pitch salience computation can also achieve a very high speed-up even its computation complexity is *O(n)* is that it has no transfer between the host and the device. In addition, the previous hidden coefficient is large. The peaks filter part will be better explained in detail later.

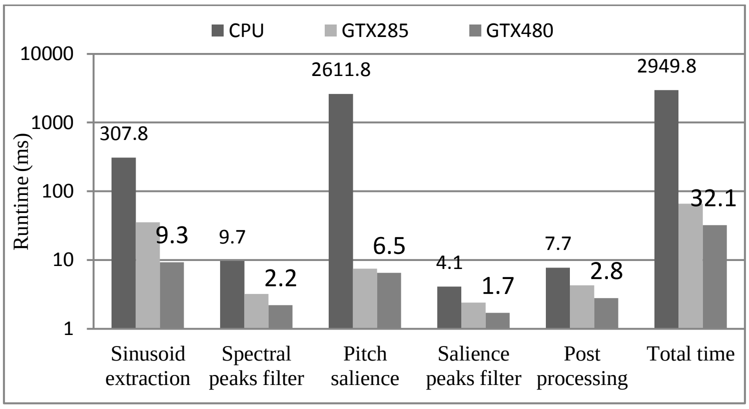

Fig. 4. Performance comparison of different parts and the whole system (Time for CPU and GTX 480 is marked for clarity)

5.2 Influence of music length

As the melody extraction is processed mostly frame by frame, the extraction time will increase gradually with the growing of audio length, as illustrated in Fig. 5. Although the entire trend of the line growth is incremental, there still exist some points which descend with the growing of audio length. This is because of the influence of specific audios. That means the extraction procedure has different levels in difficulty. It is obvious that polyphonic music with lower energy of background will be much easier to extract the melody than the music with a strong energy of background intuitively as the former has less candidate pitches, and this will reduce the computation time of pitch salience.

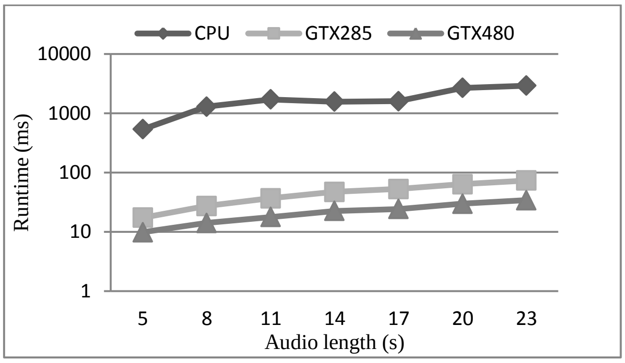

Fig. 5. Extraction time along with the growing of audio length on different platforms

5.3 Efficiency of parallelized finding peaks

From Fig. 4, we can also see that the two peak filter parts have a relative low speed-up. This phenomenon arises from the massive space allocation on device memory under the situation of no transfer between the device and the host, and the low algorithm complexity. For a problem with a low computation complexity, the execution time will be dominated by the space allocation on GPU and transfer between the host and the device. Under such a circumstance, reusing GPU device memory will be important and it's also possible to reuse previous allocated space to put the found peaks. For example, the space for rearrangement of original audio is large enough to hold the peaks. In this way, the time of release and allocation space can be reduced once.

The peaks filter can be divided into more fine-grained steps: firstly, find all the peaks, and then the max peak. Thirdly, execute the first level peaks filter. At last, do the second level peaks filter. If just ignoring the space allocation to compare the computation time, we can still see the positive effect of our parallelization, as illustrated in Table 4.

Table 4. Execution Time of Each Step at the Spectral Peaks Filter (ms).

Hardware	Find peaks	Max peak	First filter	Second filter
CPU	8.54	0.29	0.38	0.44
GTX 285	1.26	0.32	0.29	0.07
GTX 480	0.24	0.38	0.11	0.03

Owing to the "space for time" strategy, the operation of finding peaks achieves an obvious speed-up. The classic problem of finding max doesn't demonstrate a speed-up, and the too few data are to blame. The filter using atomic operation shows a small speed-up. It is understandable that few data and access conflict both exist in the extraction. The second filter works better than the first one as the possibility of conflict is smaller than that in the first filter. From Table 4, we can also see that the allocation and release time occupy a big proportion if adding up all the time and comparing it with Fig. 4.

5.4 Influence to the accuracy

The operation of double-precision floating point on GPU is time-consuming, so the single-precision floating point is adopted in our system. Although the precision is reduced, the accuracy doesn't drop. The overall accuracy on our parallel system is 73.7 %, the same as the serial approach on dataset MIR-1k. But the extraction time is reduced from nearly one hour to less than one minute. This promotes the efficiency of development tremendously. So we can spend more time on the improvement of the accuracy of the algorithm itself, and do not need to concern about the performance.

6 Conclusions and Future Work

Melody extraction from polyphonic music is a valuable problem because the melody can be used in many valuable applications. However, the relative long extraction time and the low accuracy limit its extensions. The extraction can be accelerated by GPU due to its high computation capability. In this paper a fast extraction approach based on GPU is presented. The results show that GPUs are well suited for audio signal processing problems as its characteristic is that the frame is processed one by one. Our parallel implementation reduces computation times by nearly two orders of magnitude on GTX 480. The acceleration is so tremendous that our system can be applied to some real-time applications, such as query by humming. Moreover, another benefit from acceleration is that it can greatly reduce the development time, so we can take more time to improve the accuracy of melody extraction.

In the future, we will implement a real query by humming application using our parallel extraction approach. At the same time, the tune of parameters and new methods about melody extraction will be verified on GPU platform deeply.

Acknowledgments. This research was funded by the Hi-tech Research and Development Program of China (863 Program) under Grant No.2011AA01A205, the National Natural Science Foundation of China under Grant No.61232009, the Doctoral Fund of Ministry of Education of China under Grant No.20101102110018, Beijing Natural Science Foundation under Grant No.4122042, the fund of the State Key Laboratory of Software Development Environment under Grant No.SKLSDE-2012ZX-07 and Shanghai Science and Technology Innovation Action Plan under Grant No.11511500400.

References

1. Poliner, G. E., Ellis, D. P. W., Ehmann, F., G ́omez, E., Steich, S., and Ong, B., “Melody transcription from music audio: Approaches and Evaluation,” IEEE Trans. on Audio, Speech and Language Process., vol. 15, no. 4, pp. 1247–1256, 2007.
2. Dannenberg, R. B., Birmingham, W. P., Pardo, B., Hu, N., Meek, C., and Tzanetakis, G., “A comparative evaluation of search techniques for query-by-humming using the MUSART testbed,” J. of the American Soc. for Inform. Science and Technology, vol. 58, no. 5, pp. 687–701, Feb. 2007.
3. Downie, J. S., “The music information retrieval evaluation exchange 2005–2007: A window into music information retrieval research,” Acoustical Science and Technology, vol. 29, no. 4, pp. 247–255, 2008.
4. Schmidt, E.M., West, K., Kim, Y.E., “Efficient Acoustic Feature Extraction for Music Information Retrieval Using Programmable Gate Arrays,” Proceedings of the 2009 International Society for Music Information Retrieval Conference, Kobe, Japan: ISMIR.
5. Bray, S. and Tzanetakis, G., “Distributed audio feature extraction for music,” in Proceedings of the International Conference on Music Information Retrieval, 2005, pp. 434–437.
6. Battenberg, E. and Wessel, D., (2009) “Accelerating nonnegative matrix factorization for audio source separation on multi-core and many-core architectures,” International Society for Music Retrieval Conference, 2009.
7. Schmädecke, I., Mörschbach, J., and Blume, H., “GPU-based acoustic feature extraction for electronic media processing,” in Proc. 14th ITG Conf. Electronic Media Technology, Dortmund, Germany, 2011.
8. Ozerov, A., Philippe, P., Bimbot, F., and Gribonval, R., “Adaptation of bayesian models for single-channel source separation and its application to voice/music separation in popular songs,” IEEE Trans. on Audio, Speech, and Language Process., vol. 15, no. 5, pp. 1564–1578, Jul.2007.
9. Guangchao, Y., Yao, Z., Limin, X., Li, R. and Yongnan, L., “Efficient Vocal Melody Extraction from Polyphonic Music Signals.” Electronics and Electrical Engineering. 2013,19(6):103-108.
10. Kumar, N. S. L. P., Satoor, S., and Buck, I.. “Fast parallel expectation maximization for gaussian mixture models on gpus using cuda.” In Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications, pages 103–109, Washington, DC, USA, 2009. IEEE Computer Society.
11. Catanzaro, B., Sundaram, N., and Keutzer, K., “Fast Support Vector Machine Training and Classification on Graphics Processors.” In Proceedings of the 25th International Conference on Machine Learning, pages 104–111, 2008.
12. Klapuri, A., “Multiple fundamental frequency estimation by summing harmonic amplitudes,” inProc. 7th Int. Conf. on Music Inform. Retrieval, Victoria, Canada, Oct. 2006, pp. 216–221.
13. Goto, M., “A real-time music-scene-description system: predominant-f0 estimation for detecting melody and bass lines in real-world audio signals,” Speech Communication, vol. 43, pp. 311–329, 2004.
14. Yang, Y., Xiang, P., Mantor, M. and Zhou, H., “Fixing Performance Bugs: An Empirical Study of Open-Source GPGPU Programs,” In International Conference on Parallel Processing, 2012.
15. Green, O., McColl, R., and Bader, D.A., "GPU merge path: a GPU merging algorithm," in Proc. ICS, 2012, pp.331-340.
16. NVIDIA, CUDA C Best Practices Guide 4.1, 2012.
17. NVIDIA, CUFFT Library 4.1, 2012.
18. NVIDIA, CUDA CUBLAS Library 4.1, 2012.
19. Hsu, C. L. and Jang, J. S., “On the improvement of singing voice separation for monaural recordings using the MIR-1K dataset,” IEEE TASLP, volume 18, pp. 310-319, 2010.