GPU-Accelerated Beat Detection for Dancing Monkeys

Philip Peng
University of Pennsylvania

Yanjie Feng
University of Pennsylvania

Abstract

In music-based rhythm games, the game system needs to create patterns matching with background songs for player to play with. These patterns are often created by manually by the game developers. Such manual works have obviously limitations. As a result, Karl O'Keefe of Imperial College London created a system which can automatically employ beat detection to generate DDR-style stepfiles for arbitrary songs. However, his approach is using a brute force calculation method which is very slow to achieve the accuracy of the result.

Our Project focus on trying to figure out a way to accelerate this approach on GPU by testing different approaches and compares the results.

1. Introduction

To create a music-based rhythm games, it always requires generating some arrow patterns which can presents the rhythm of the background songs to let player to hit them. In old days, such patterns were always generated manually. Manual create those patterns is a very tedious and boring work and always leads to not very accurate results.

Later, many studies related with how to generate such pattern automatically have been done. Almost in all those methods, a real number presenting the rhythimic tmpo of a song called beats-per-minute (BPM) was mentioned as one of the most important parts to create patterns from songs.

The calculation of BPM requires large number of calculation such as sorting, sampling, checking fitness, looking up results, etc. As a result, it always takes a lot of time to calculate a suitable result. More details about how to calculate BPM will be mentioned in section 3

2. Related Work

Beating detection has been studied by many people in previous years. "Beat this" System created by Cheng et al. was based on handling certain psychoacoustic characteristics of humans to "perceive" the pulse content of a musical signal in ways similar to the human ear. Will Archer Arentz's paper "Beat Extraction from Digital Music" introduces a method taking advantage of the song's repeating sampling to generate beat patterns by analysis BPM from digital music.

3. The Dancing Monkey Framework

Our project is based on an open-source project called dancing monkey which is an implementation of "Beat Extraction from Digital Music" done by Karl O'Keefe, a student of Imperial College London. The workflow of dancing monkey's system is as follow:

1, The system takes a song and some other parameters like difficult level, output format setting, etc. as inputs.

2, The system will decoding the song files. If the file is an MP3 file, then the system will convert it to a wav file. Waveform data will be collected and stored into system for further use.

3. The system will calculate the BPM and gaps based on the waveform data.

4. Generate arrow patterns based on BPM and gaps.

In our project, we focus on trying to use GPU to accelerate step 3. The step 3 can also be divided into several small steps:

3.1, The system checks the entire song to get the peaks and troughs.

3.2, It calculates beat offsets from peaks.

3.3, It tries the first pass to find whether we can get a close BPM result in a user defined range by a brute force method step through the entire range. And check the fitness of such result.

3.4, It tries to refine the result of 3.3 by a second pass and get a final result.

Basically, 3.3 and 3.4 are just the same. The calculation and fitness checking are two big loops which we decided to use GPU techniques to replace and get more accelerated results.

4. Accelerated Experiment

In order to start our accelerate experiments, we first set up a time breakdown on the default project and try to see how long does it take for each part of the system. Details are showed in Figure 1.

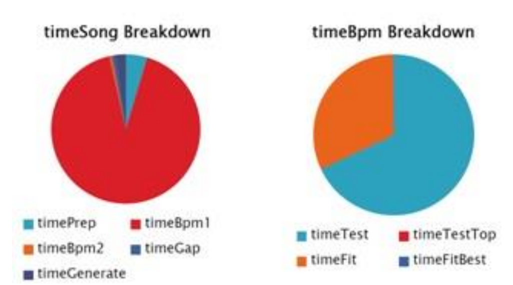


Figure 1 The time breakdown of original dancing monkey project and its BPM calculation part.

As mentioned before, our project is mainly focus on the BPM calculation part with its brute force calculation part and fitness check part. We designed several approaches to accelerate them.

4.1 Matlab Parfor

Matlab's parallel computing tools provide a "parfor loop" besides the default "for loop". It can be used similar like a "for loop" to execute a series of calculation in the loop body.

The difference between parfor and for is that the parfor will divide the calculation in its loop body to different threads on CPU. Thus, those calculations are paralleled. Such threads are called "workers" according to Matlab documents. In order to notify the CPU to run such tasks, "matlabpool" command needs to be done to let the CPU know how many workers will be assigned into the work.

However, a parfor loop doesn't accept discontinue range or a different step other than 1. So, we need to re-write the default index by creating a temporary array whose index is continued and having a step of 1. After that, we copy back the calculated results from the temporary array to default array. We made modification on both BPM calculation part and the fitness check part.

4.2 Matlab GPUarray

GPUarray is another approach we tried to increase the project's performance provided by Matlab's computing tools.

The GPUarray function can help us initialize the data directly on GPU side. And then, set a pointer to the function which we want to process the data on GPU and using Matlab's "arrayfun" with that pointer as a parameter to calculate those data in parallel. Finally, using "gather" function to get data back to CPU side.

The "arrayfun" function will decide to run on CPU or GPU depending on the data sent to it which is a similar feature like thrust.

In order to achieve this workflow, we need to modify the default 2 big loops into 2 functions. And the data used in those functions will also need to be re-construct so that they will be suitable to run on GPU.

In practice, we also encountered 2 other great challenges: first, the original for-loop highly depending on many data outside the loop which can only be treated as global variables to a modified function. However, arrayfun cannot use such global variables on GPU side; second, matlab only provides very limited data-structures on GPU side. Almost no other data-structure than GPUarray exists. But the original code requires calculating some different types of matrix. Those matrixes are hard to move to GPU.

4.3 Jacket gfor

Jacket is a matlab plug-in developed by Accelereyes. It provides lots of more GPU data structures supporting lots of more functions running on GPU. Its "gfor" loop can easily modify any default for loop to run on GPU side which they claim will significantly increase the performance of original for loop.

In order to make Jacket work on our project, we first initialize all data directly used in calculation as GPU data by using Jacket's gdouble, gones, gzeros functions.

And in order to let those data outside the "for loop" can be also used in the loop without any memory issue. We use Jacket's "local" function to provide each kernel a copy of such data so that no CPU Subscripted data error will occur.

After all those done, we cast default for loop as Jacket's gfor loop to see whether the result can be improved significantly.

5. Result

The parfor used on CPU significantly improved the performance of the original code. Details are showed in the figure 2.

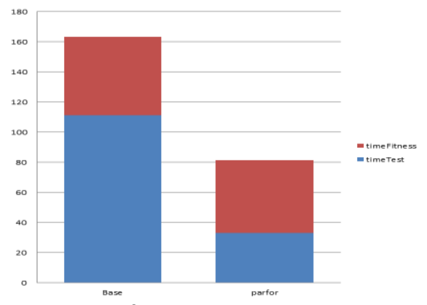


Figure 2 Performance comparisons between default and parfor version

By dividing the tasks into multiple CPU cores, the performance doubled.

After we successfully made performance improvement by using parfor function, we moved on to the GPU side. However, the GPU side's result is extremely slower than default in every single step. In order to find out why GPUarray actually made the performance decrease so much, we timing the by every single function. The result is showed in figure 3.

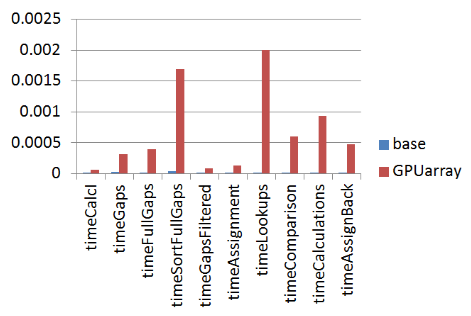


Figure 3 Performance comparisons between base and GPUarray in all basic functions.

In all those functions, GPUarray failed to win performance with default CPU code. But, all those functions need to be used a lot in original code. As a result, we failed to use GPUarray to improve the performance of the original code.

And then we move on to try Jacket. Which claims itself faster than GPUarray. However, the result we got in our project is also very slow.

The detailed timing comparisons of 2 big loops between base and Jacket are showed in Figure 4.

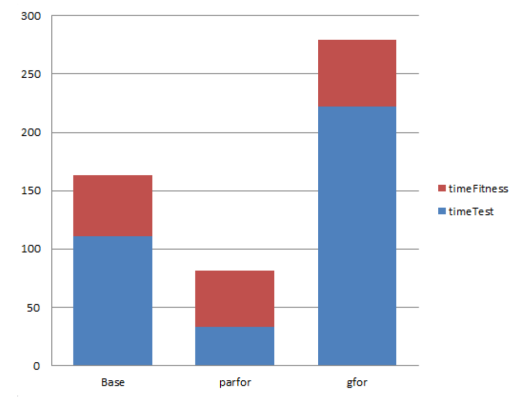


Figure 4 Performance comparisons of 2 big loops between base and Jacket in a song with about 1.6 thousand loops

6. Further Analysis

In order to find out much more details why our GPU experiments don't give us the results we want, we made some further testing on more functions and make comparisons with CPU, GPUarray, Jacket.

In all tests, the GPUarray is always the slowest except doing mod calculation. More details are showed in Figure 5.

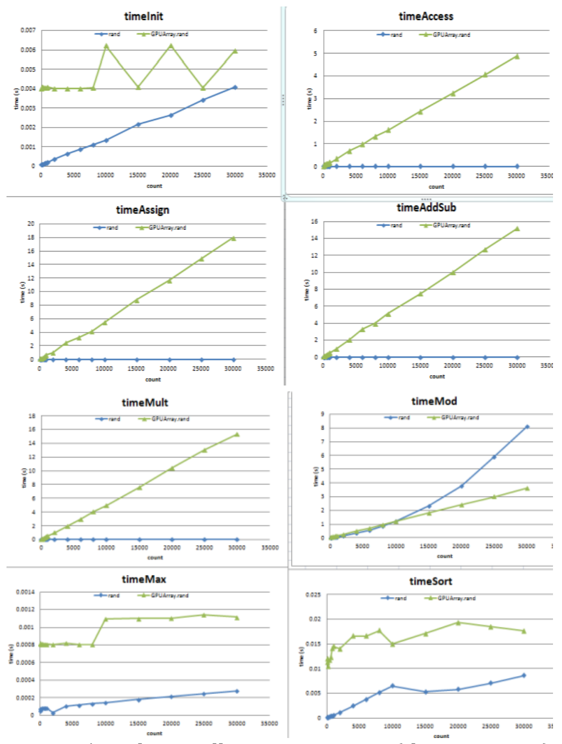


Figure 5 In almost all tests, GPUarray (the green one) takes more time than default CPU code and Jacket.

To Jacket, our tests shows it is good at handling math calculations with large numbers of data. However, in our project, a song rarely has that large number of data.

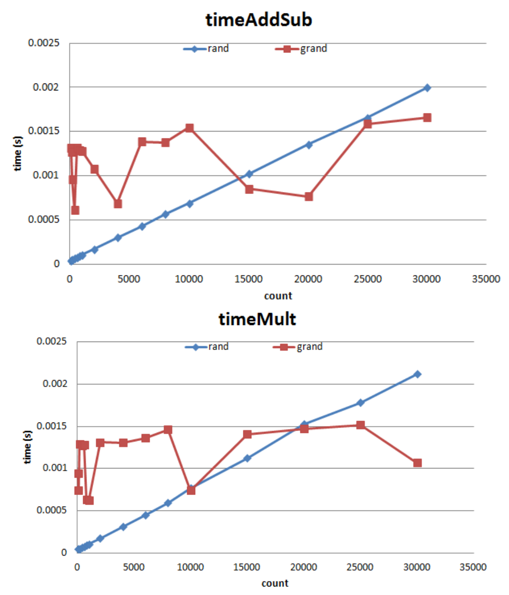
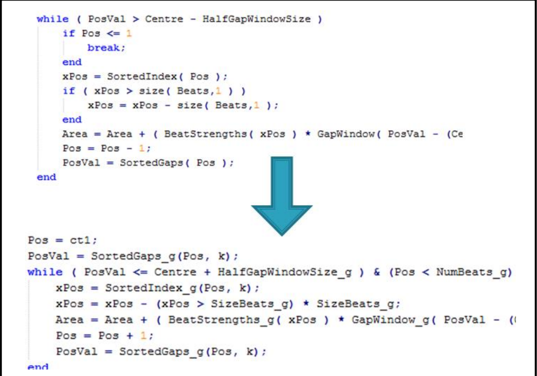


Figure 6 Math Calculation performance comparisons of Jacket and CPU. Notice that, Jacket can gain better performance only when the data set is large enough.

We also found Jacket is better in sorting, initialization data on GPU. However, Jacket is bad in data accessing. As a result, its advantage is balanced by its drawback in our project so that we didn't get a good result.

7. Conclusion

In this project, GPU acceleration proved to be infeasible due to 1) the small size of the data being operated on, and 2) the algorithm's frequent usage of data access, which outweighed any speed improvements in other areas.



```
  while ( PosVal > Centre - HalfGapWindowSize )
      if Pos <= 1
          break;
      end
      xPos = SortedIndex( Pos );
      if ( xPos > size( Beats,1 ) )
          xPos = xPos - size( Beats,1 );
      end
      Area = Area + ( BeatStrengths( xPos ) * GapWindow( PosVal - (Ce
      Pos = Pos - 1;
      PosVal = SortedGaps( Pos );
  end

Pos = ct1;
PosVal = SortedGaps_g(Pos, k);
while ( PosVal <= Centre + HalfGapWindowSize_g ) & (Pos < NumBeats_g)
    xPos = SortedIndex_g(Pos, k);
    xPos = xPos - (xPos > SizeBeats_g) * SizeBeats_g;
    Area = Area + ( BeatStrengths_g( xPos ) * GapWindow_g( PosVal - ((
    Pos = Pos + 1;
    PosVal = SortedGaps_g(Pos, k);
end
```

One option we explored later was tweaking/optimizing the original code itself. This lead to drastic speed performance unrelated to parallelization, as shown in Figure 7. It is interesting to note that upon applying gfor to the tweaked code, the final times were still slower. Any further optimization would require rewriting the program in a different language (e.g. direct C code with CUDA) to avoid the MATLAB overhead from data access.

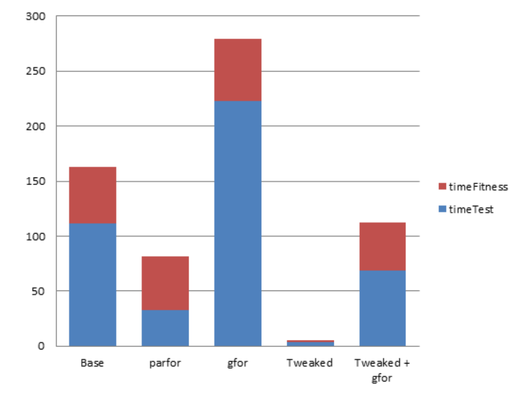


Figure 7 Final results with tweaked code

8. Bibliography

Karl O'Keeffe, "Dancing Monkeys", MEng Individual Project Report 18th June 2003

Will Archer Arentz, "BEAT EXTRACTION FROM DIGITAL MUSIC"