ACCELERATING QUERY-BY-HUMMING ON GPU

Pascal Ferraro
LaBRI - U. Bordeaux 1, France
PIMS/CNRS - U. Calgary, Canada
ferraro@cpsc.ucalgary.ca

Pierre Hanna
LaBRI - U. Bordeaux 1,
France
pierre.hanna@labri.fr

Laurent Imbert
Lirmm - CNRS, France
PIMS/CNRS - U. Calgary, Canada
laurent.imbert@lirmm.fr

Thomas Izard
Lirmm - U. Montpellier 2,
France
thomas.izard@lirmm.fr

ABSTRACT

Searching for similarities in large musical databases has become a common procedure. Local alignment methods, based on dynamic programming, explore all the possible matchings between two musical pieces; and as a result return the optimal local alignment. Unfortunately these very powerful methods have a very high computational cost. The exponential growth of musical databases makes exact alignment algorithm unrealistic for searching similarities. Alternatives have been proposed in bioinformatics either by using heuristics or by developing faster implementation of exact algorithm. The main motivation of this work is to exploit the huge computational power of commonly available graphic cards to develop high performance solutions for Query-by-Humming applications. In this paper, we present a fast implementation of a local alignment method, which allows to retrieve a hummed query in a database of MIDI files, with good accuracy, in a time up to 160 times faster than other comparable systems.

1. INTRODUCTION

One of the main goal of music retrieval systems is to find musical pieces in large databases given a description or an example. These systems compute a numeric score on how well a query matches each piece of the database and rank the music pieces according to this score. Computing such a degree of resemblance between two pieces of music is a difficult problem. Three families of methodologies have been proposed [1]. Approaches based on index terms generally consider N-grams techniques [2,3], which count the number of common distinct terms between the query and a potential answer. Geometric algorithms [4–6] consider geometric representations of music and compute distances between objects. Techniques based on string matching [7] are generally more accurate as they can take into account errors in the query or in the pieces of music of the database. This property is of major importance in the context of music retrieval systems since audio analysis always induces approximations. Moreover, some music retrieval application require specific robustness. Query by humming (QbH), a music retrieval system where the input query is a user-hummed melody, is a very good example. Since the sung query can be transposed, played faster or slower, without degrading the melody, retrieval systems have to be both transposition and tempo invariant. Edit distance algorithms, mainly developed in the context of DNA sequence recognition, have been adapted in the context of music similarity [8]. These algorithms, based on the dynamic programming principle, are generalisations of a local sequence alignment method proposed by Smith and Waterman [9] in the early 80's. Applications relying on local alignment are numerous and include cover detection [10], melody retrieval [8], Query-by-Humming [11], Query-by-Tapping [12], structural analysis, comparison of chord progressions [13], etc. Local alignment approaches usually provide very accurate results as shown at the recent editions of the Music Information Retrieval Evaluation eXchange (MIREX) [14].

Alignment algorithms are powerful and optimal: they always find the best alignment. However, they are also very time consuming. This drawback considerably limits their use for musical applications. For biological applications, heuristics such as BLAST and FASTA can be used to speed-up local sequence alignment while allowing for multiple regions of local similarity. These heuristics are valuable, but they may fail to report hits or may report false positives. In order to get more accurate results faster implementations of exact alignment algorithms are therefore of primary importance.

Graphics Processing Units (GPUs) have recently received lots of attention thanks to their extensive computing resources. Not only are the latest generations of GPUs very powerful graphic engines, they can also be used for General Purpose computation (GPGPU) [15]. With the recent evolutions of GPUs' architecture into a unified, highly

parallel programmable processor, and the development of programming tools and high-level programming languages such as NVIDIA's CUDA, GPUs have become a very attractive, low-cost alternative to the traditional microprocessors for computationally demanding applications that can be expressed as data-parallel computations, *i.e.* the same program is executed on many data elements in parallel. This type of parallelism is well suited to the problem of QbH on very large scale music databases, although it also brings new challenges regarding memory operations and computational resource allocations. In this paper, we present an implementation of a variant of Smith-Waterman based on local transpositions which illustrates the advantages of recent graphic cards as computation platforms.

In Section 2 we present the general concepts of sequence alignment and a variant based on local transpositions well suited to musical applications. Our parallel implementation on GPU is detailed in Section 3. Several tests and comparisons are presented in Section 4.

2. ALIGNING TWO MUSIC

In this section, we briefly present the QbH system experimented. Following Mongeau and Sankoff [8], any monophonic piece can be represented by a sequence of notes, each given as a pair *(pitch, length)*. Several alphabets and sets of numbers have been proposed to represent pitches and durations [3]. In the following, we are using the interval relative representation, *i.e.* the number of semitones between two successive notes reduced modulo 12. In the context of QbH applications, this representation presents the huge advantage to be transposition invariant.

2.1 General Sequence Alignment

Sequence alignment algorithms are widely used to compare strings. They evaluate the similarity between two strings t and q given on an alphabet A, and of respective sizes $|t|$ and $|q|$. Formally an alignment between t and q is a string z on the alphabet of pairs of letters, more precisely on $(A \cup \{\epsilon\}) \times (A \cup \{\epsilon\})$, whose projection on the first component is t and the projection on the second component is q. The letter ϵ does not belong to the alphabet A. It is often substituted by the symbol "-" and is called a *gap*. An aligned pair of z of type (a, b) with $a, b \in A$ denotes the *substitution* of the letter a by the letter b. A pair of type $(a, \text{-})$ denotes a *deletion* of the letter a. Finally, an aligned pair of type $(\text{-}, b)$ denotes the *insertion* of the letter b. A score $\sigma(t_i, q_j)$ is assigned to each pair (t_i, q_j) of the alignment. The score S of an alignment is then defined as the sum of the costs of its aligned pairs. Computational approaches to sequence alignment generally fall into two categories: *global alignments* and *local alignments*. Calculating a global alignment is a form of global optimization that forces the alignment to span the entire length of all query sequences. By contrast, local alignments identify regions of similarity within long sequences that are often widely divergent overall. In Query-by-Humming applications, since the query is generally much shorter than the reference, one favours local alignment methods.

Both alignment techniques are based on dynamic programming [9, 16, 17]. Given two strings t and q, alignment algorithms compute a $(|t| + 1) \times (|q| + 1)$ matrix T such that:

$$T[i, j] = S(t[0 \dots i], q[0 \dots j]),$$

where $S(t[0 \dots i], q[0 \dots j])$ is the optimal score between the subsequences of t and q ending respectively in position $0 \leq i \leq |t|$ and $0 \leq j \leq |q|$. Dynamic programming algorithms can compute the optimal alignment (either global or local) and the corresponding score in time $\mathcal{O}(|t| \times |q|)$ and memory $\mathcal{O}(\min\{|t|, |q|\})$ (see [9] for details).

2.2 Local Transposition

Queries produced by human beings can, not only be totally transposed, but can also be composed of several parts that are independently transposed. For example, if the original musical piece is composed of different harmonic voices, the user may sing different successive parts with different keys. In the same way, pieces of popular music are sometimes composed of different choruses sung based on different tonic. A sung query may imitate these characteristics. Moreover, errors in singing or humming may occur, especially for users that are not trained to perfectly control their voice like professional singers. From a musical point of view, sudden tonal changes are disturbing. However, if these changes last during a long period, they may not disturb listeners. Figure 1 shows an example of query having two local transpositions.

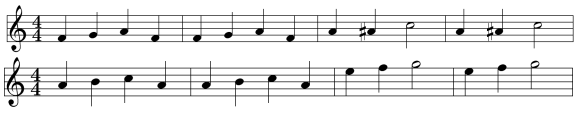

Figure 1. Example of a monophonic query not transposed (top) and a monophonic query with two local transpositions (bottom).

The two pieces in Figure 1 sound very similar, although the two resulting sequences are very different. This problem has been addressed in [18] by defining a local transposition algorithm. It requires to compute multiple score matrices simultaneously, one for each possible transposition value. The time complexity is $\mathcal{O}(\Delta \times |q| \times |t|)$, where Δ is the number of local transposition allowed during the comparison (for practical applications, Δ is set up to 12). Our experiments, presented in section 4, show that the local transposition algorithm provides a much better result.

2.3 Pitch/Duration Scoring Scheme

The quality of an alignment-based algorithm heavily depends on the scoring function. Results may differ significantly whether one uses a basic scoring scheme or a more sophisticated scoring function [7]. For our experiments, we use the scoring schemes introduced in [8] and [7], where the score between two notes depends on the pitch, the duration and the consonance of both notes. For example, the fifth (7 semitones) and the third major or minor (3 or

4 semitones) are the most consonant intervals in Western music [19]. The score function between two notes is then defined as a linear combination of a function σ_p on pitches (its values are coded into a matrix) and a function σ_d on durations as:

$$\sigma(a,b) = \alpha \cdot \sigma_p(a,b) + \beta \cdot \sigma_d(a,b).$$

The cost associated to a gap only depends on the note duration. Finally a penalty (a negative score) is also applied to each local transposition.

3. PARALLEL IMPLEMENTATION

3.1 GPU Architecture

Both AMD and NVIDIA build architectures with unified, massively parallel programmable units, which allow programmers to target that programmable unit directly instead of dividing work across multiple hardware units. More precisely, a GPU contains many streaming multiprocessors (MPs) each containing several elements including several cores, also called streaming processors (SPs), and various types of on-chip shared memories and registers. The MPs also share some constant memory areas with very fast access and a global uncached large memory with relatively low throughput and long latency. For example, the NVIDIA GeForce 9800 GX2 used for our experiments (see Section 4) is a dual GPU engine with 256 cores (128 per GPU) running at 1.5 GHz. These cores are regrouped into 2×16 MPs which share a global memory of 1GB with a 512-bit interface width providing a throughput of 128 GB/sec (64 GB/sec per GPU).

The MPs creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. To manage hundred of threads running several different programs, the multiprocessor employs an architecture called SIMT (single-instruction multiple-thread), which resembles SIMD (single-instruction multiple-data) vector organizations, *i.e.*, single instruction controls multiple processing elements. Unlike SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads.

3.2 GPU Computing with CUDA

Our implementation uses CUDA, a general purpose parallel computing framework developed and distributed by NVIDIA for use with their recent GPUs [1] . CUDA can be seen as an extension of C that allows developers to define C functions, called *kernels* to be executed N times in parallel by N different CUDA *threads*. CUDA threads may access data from multiple memory spaces during their execution. CUDA's programming model assumes that the CUDA threads execute on a separate *device*, whereas the rest of the program runs on a CPU. In other words, the GPU operates as a coprocessor to the *host* running the C program. Both the host and device maintain their own memory areas, allowing for concurrent programming between the CPU and the GPU(s). CUDA kernels must be compiled into binary code using `nvcc`, a C compiler for CUDA. Note that `nvcc` supports C++ programming for host functions but kernels must be written in C, possibly with templates. `nvcc` also supports device emulation.

[1] CUDA was introduced in November 2006 along with the G80 series. CUDA can be downloaded for free from `http://www.nvidia.com/object/cuda_home.html`. A list of CUDA-enabled product is available at `http://www.nvidia.com/object/cuda_learn_products.html`.

3.3 CUDA implementation of QbH

The process of evaluating how well each piece of music in a database match a query (sung or hummed in the case of QbH), and rank the music pieces according to this score can be parallelized at different levels. As explained earlier, our implementation uses a variant of Smith-Waterman. Although it is possible to do so [20], our choice was not to parallelize the implementation of Smith-Waterman itself, as this approach could only provide significant improvements for extremely large sequences. In contrast, our CUDA implementation optimizes the arithmetic intensity (the ratio of arithmetic operations to memory operations) by computing in parallel all the scores of a query with every piece of music in the database. If the database contains N pieces of music, our program virtually launches N kernels executing the Smith-Waterman algorithm in parallel. The main challenges are therefore to optimize the resource allocations and the memory operations.

After the query has been converted from its original format (typically a wave audio file), we store it in a special memory area called the *texture* memory, which allows for very fast read/write operations. The texture memory is shared among all threads (Fig. 2). The database usually contain too many pieces of music to be stored in any of the cached, fast memories (texture, constant, shared memory). Therefore, all the pieces of music are stored in the global memory. On a GPU, the global memory is not cached, so it is extremely important to follow the right access pattern to get maximum memory bandwidth. Throughput of memory operations is 8 operations per clock cycle, plus 400 to 600 clock cycles of memory latency. Under some size and alignment conditions, the device is capable of reading data from global memory in a single load instruction. Moreover, the memory bandwidth can be used most efficiently when the simultaneous memory access by all the active threads can be coalesced into a single memory transaction. (For more details, see [21].)

In order to satisfy all these constraints, the pieces of music of the database are store in an array of `float2`, a CUDA structure containing two 32-bit floats, which stores the pitch and duration of each note. Although the size and alignment properties are fulfilled by this data type, storing the pieces of music sequentially, one after the other, would be very inefficient since simultaneous reading by all the threads in a single transaction would be impossible. Instead, if the database contain N pieces of music, we organize the data in memory as a one dimensional array, such that its first N entries correspond to the first note (pitch, duration) of each piece; then, the next N entries correspond

to the second note of each piece, etc.

Each thread performs its computations on its own matrix, more exactly on its $\Delta = 12$ transposition matrices. In order to minimize the amount of required memory, we only store the current row of each matrix. Moreover, to optimize memory alignment, allocation is based on the query's fixed size rather than the pieces of music's variable sizes. Finally, in order to allow simultaneous read/write operations by the active threads, the matrices are not stored at consecutive addresses but rather using the same strategy as the database. Fig. 2 describes the device architectures.

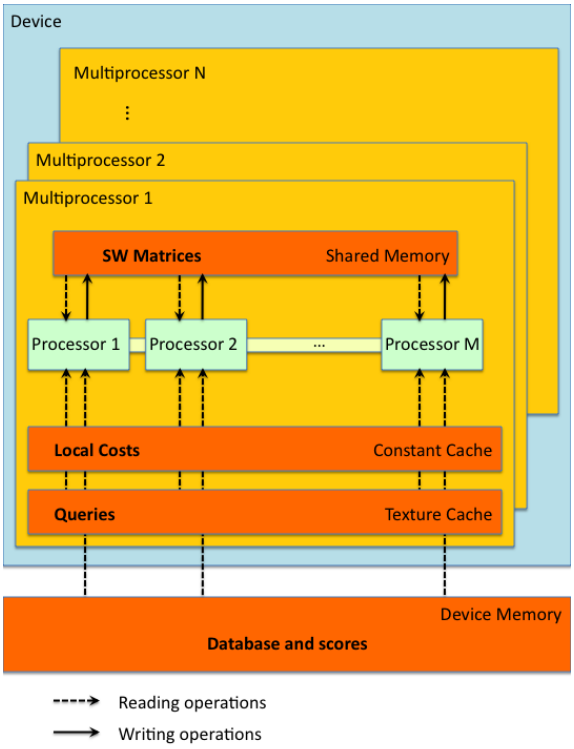

Figure 2. nVidia GPUs architecture. Each multi-processor executes both the conversion of queries from audio files to a vector of notes (stored in the texture memory) and the comparison between the query and each reference (stored in the device memory). Each processor store its intermediate Smith-Waterman matrices (only one row) in its own shared memory space. Constants costs and intermediate values are respectively stored in constant and shared memories.

4. TESTS AND RESULTS

4.1 Benchmark

Our experiments are based on the query data corpus proposed for the QbH tasks at the MIREX 2007 and 2008 and three different noise databases. Roger Jang's corpus is composed of 2797 queries, along with 48 ground-truth MIDI files [2], with the particularity that all queries start at the beginning of the references. The first database (called DB1) consists of the 48 ground-truth MIDIs and a subset of 2000 MIDI noise files from the Essen Collection [3]. The whole Essen Collection, made of 5982 files together with the 48 ground-truth MIDIs files, is called DB2. Finally, since the ground-truth MIDIs are rather short while Essen collection mainly consists of long data files, we also consider a third database, called DB3, proposed during the MIREX 2005, which is a subset of the RISM A/II (International inventory of musical sources) collection, composed of 17433 short excerpt of real world compositions.

We have tested our implementation on three different platforms. Their characteristics are given in Table 1. For each algorithm we have measured the time on both the CPU alone and the CPU together with the GPU used as a parallel coprocessor.

	OS	CPU	GPU
W1	NVidia Tesla Workstation running Linux, CUDA v2.1	3GHz Intel Core 2 Duo	NVidia GeForce 9800 GX2, 512 MB memory
W2	Mac Pro running Mac OS X 10.5, CUDA v2.2	Two 2.8GHz Intel Xeon Quad Core	NVidia GeForce 8800 GT, 512 MB Memory
L1	MacBook Pro running Mac OS X 10.5, CUDA v2.2	2.53 GHz Intel Core 2 Duo	NVidia GeForce 9400 M, shared memory with CPU

Table 1. Characteristics of our three platforms

Regarding the algorithms, we have implemented the original Smith-Waterman algorithm (SW) and our extension based on local transposition alignment (LT). Since the queries are known to be at the beginning of the references, we have implemented variants of the above algorithms that only compare the query with the beginning of each MIDI files (in Table 2 we only report timings for size of the query plus 10 notes). These variants are respectively called SW10 and LT10.

We evaluate the quality of our music retrieval system using two measures. The Mean Reciprocal Rank (MRR), *i.e.* the average of the reciprocal ranks of the first correct answer, computed for a sample of N queries as

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{r_i},$$

where r_i is the rank of the first correct answer for the ith query. And the top-X ratio which reports the proportion of queries for which $r_i \leq X$.

4.2 Smith-Waterman vs Local Transposition Alignment

We first evaluate the quality, given in terms of MRR and top-5 ratio, of SW and LT on the three databases. For

[2] http://www.cs.nthu.edu.tw/~jang

[3] http://www.esac-data.org/

DB1, SW reaches a MRR of 0.274 and a top-5 ratio of 30.3%, while LT reaches a MRR of 0.684 and a top-5 ratio of 75.4%. The MRR increases when the size of the query is taken into account during the comparison: SW10 reaches a MRR of 0.295 and a top-5 ratio of 32.3%, while LT10 reaches a MRR of 0.732 and a top-5 ratio of 79.4%. We observe the same behaviour for DB2 and DB3. Fig. 3 shows that the MRRs obtained for databases of different sizes remains roughly the same. The local transposition alignment method provides better results than Smith-Waterman.

For a suggestive comparison, the method submitted by Wu and Li, which performed best at MIREX 2008 reaches a MRR of 0.9 on a well choosen subset of 2000 MIDI files from the Essen collection. However, since we could not find the original database used for the MIREX 2008 competition, our DB1 consists of 2000 randomly chosen MIDI files from the Essen collection. It therefore contains several copies of the same noise files, which automatically impacts the quality of our results.

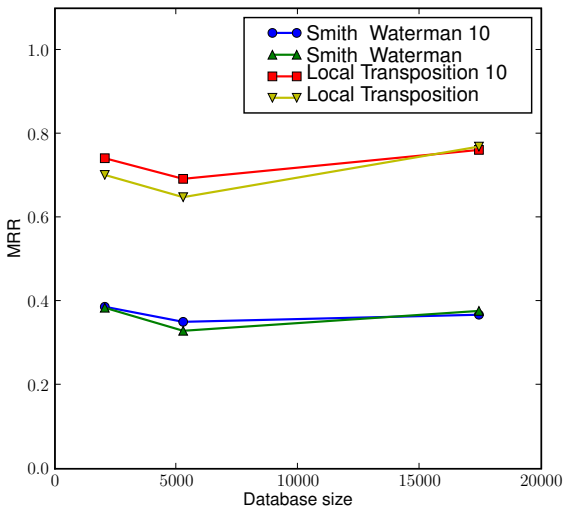

Figure 3. MRRs obtained for SW, SW10, LT, LT10 on our three databases

4.3 CPU vs GPU

Although LT provides very good results in terms of quality, it is very time consuming. On our fastest CPU (W2), a Mac Pro equipped with two 2.8 GHz Intel Xeon Quad Core processors, the analysis on DB1 takes more than 326 minutes with LT10 ($\sim$ 7 sec. per query) and more than 595 minutes with LT ($\sim$ 13 sec. per query). This computation time even reaches 1282 minutes for LT10 ($\sim$ 27.5 sec. per query) on the largest database DB3, which contains more than 17000 MIDI files.

As shown in Table 2, our CUDA implementations provide impressive speed-ups. The local transposition algorithms (LT and LT10) which gives the highest MRRs and top-X ratios, are up to 162 times faster than their CPU counterpart. This is achieved for DB3 on our Mac Pro configuration W2; the analysis of more than 17000 MIDI files using LT10 was completed in 473 seconds ($\sim$ 0.16 s per query). Note also that the local transposition algorithm is perfectly adapted to parallel implementations as it is only slightly slower than SW.

It is important to remark that our CUDA implementation leads to significant improvements even on our lightest configuration (L1), a laptop not designed to perform heavy graphic computations and which embeds a cheap, on-chip graphic card (NVidia 9400 M). The analysis of DB1 only takes 470 seconds, *i.e.* $\sim$ 0.16 s per query. During the MIREX 2008, the fastest implementation for the analysis of a database similar to DB1 was performed in 1699 seconds on a AMD Athlon XP 2600+ running at 1.9GHz. Our fastest implementation, running on W1, completed the analysis of the 2797 queries in only 301 seconds, that is almost 6 times faster.

	Config.	SW10	SW	LT10	LT
DB1	L1	7:33	7:33	7:50	40:54
	W1	4:12	5:05	5:01	20:16
	W2	6:00	6:00	6:01	14:42
DB2	L1	7:53	7:51	15:10	79:45
	W1	6:55	6:54	6:10	25:46
	W2	6:18	6:18	6:16	20:40
DB3	L1	9:20	9:19	41:31	44:48
	W1	5:15	6:05	10:09	25:27
	W2	7:25	7:27	7:53	21:15

Table 2. Timings of the different algorithms on various GPUs and databases in mm:ss

5. CONCLUSIONS

Local transposition alignment algorithms are very powerful. Using QbH as an experimental application, our variant of Smith-Waterman lead to very good results. We believe that this type of algorithm would give even better results for other music retrieval systems, such as cover detection, where the query is significantly larger and contains fewer errors than a sung or hummed query. Our implementation takes advantage of the immense computing resources offered by the most recent graphic cards. These low-cost devices regroup hundreds of cores that can operate in parallel and sufficient memory to store large musical databases. A great care must be taken when programming memory operations as a bad allocation strategy can have a significant impact on the computation time. At this time, we have not yet optimized the pre-processing phase of the system. In particular, the analysis and conversion of the queries (wave audio files) is running exclusively on the CPU and takes between 75-90% of the overall computation time. Our next task will be to implement this stage on GPU using the CUDA CUFFT library. We anticipate significant improvements in terms of speed.

6. ACKNOWLEGMENT

This work has been partially sponsored by the French ANR SIMBALS (JC07-188930) and ANR Brasero (ANR-06-BLAN-0045) projects.

7. REFERENCES

[1] N. Orio. Music retrieval: A tutorial and review. *Foundations and Trends in Information Retrieval*, 1(1):1–90, 2006.

[2] S. Doraisamy and S. Rüger. Robust polyphonic music retrieval with N-grams. *Journal of Intelligent Information Systems*, 21(1):53–70, 2003.

[3] A. L. Uitdenbogerd. *Music Information Retrieval Technology*. PhD thesis, RMIT University, Melbourne, Victoria, Australia, July 2002.

[4] E. Ukkonen, K. Lemström, and V. Mäkinen. Geometric algorithms for transposition invariant content-based music retrieval. In *Proceedings of the 4th International Conference on Music Information Retrieval, ISMIR 2003*, pages 193–199, 2003.

[5] R. Typke, R. C. Veltkamp, and F. Wiering. Searching notated polyphonic music using transportation distances. In *Proceedings of the ACM Multimedia Conference*, pages 128–135, 2004.

[6] R. Typke and A. Walczak-Typke. A tunneling-vantage indexing method for non-metrics. In *Proceedings of the 9th International Conference on Music Information Retrieval, ISMIR 2008*, pages 351–352, 2008.

[7] P. Hanna, P. Ferraro, and M. Robine. On optimizing the editing algorithms for evaluating similarity between monophonic musical sequences. *Journal of New Music Research*, 36(4):267–279, 2007.

[8] M. Mongeau and D. Sankoff. Comparison of musical sequences. *Computers and the Humanities*, 24(3):161–175, 1990.

[9] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[10] J. Serrà, E. Gómez, P. Herrera, and X. Serra. Chroma binary similarity and local alignment applied to cover song identification. *IEEE Transactions on Audio, Speech and Language Processing*, 16:1138–1151, 2008.

[11] R. B. Dannenberg, W. P. Birmingham, B. Pardo, N. Hu, C. Meek, and G. Tzanetakis. A comparative evaluation of search techniques for query-by-humming using the MUSART testbed. *Journal of the American Society for Information Science and Technology*, 58(5):687–701, 2007.

[12] P. Hanna and M. Robine. Query by tapping system based on alignment algorithm. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, 2009. (to appear).

[13] J. P. Bello. Audio-based cover song retrieval using approximate chord sequences: Testing shifts, gaps, swaps and beats. In *Proceedings of the 8th International Conference on Music Information Retrieval, ISMIR 2007*, pages 239–244, September 2007.

[14] J. S. Downie, M. Bay, A. F. Ehmann, and M. C. Jones. Audio cover song identification: MIREX 2006-2007 results and analyses. In *Proceedings of the 9th International Conference on Music Information Retrieval, ISMIR 2008*, pages 51–56, 2008.

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[16] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[17] D. Gusfield. *Algorithms on Strings, Trees and Sequences – Computer Science and Computational Biology*. Cambridge University Press, Cambridge, 1997.

[18] J. Allali, P. Ferraro, P. Hanna, and C. Iliopoulos. Local transpositions in alignment of polyphonic musical sequences. In *String Processing and Information Retrieval Symposium, 14th International Symposium, SPIRE 2007, Proceedings*, volume 4726 of *Lecture Notes in Computer Science*, pages 26–38. Springer, October 2007.

[19] F. J. Horwood. *The Basis of Music*. Gordon V. Thompson Limited, Toronto, Canada, 1944.

[20] Y. Liu, W. Huang, J. Johnson, and S. Vaidya. GPU accelerated Smith-Waterman. In *Computational Science, ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 188–195. Springer, 2006.

[21] NVIDIA CUDA. *Programming Guide*, April 2009. Version 2.2. Available at `http://www.nvidia.com/object/cuda_home.html`.