AUD-SWIPE-P: A PARALLELIZATION OF THE AUD-SWIPE PITCH ESTIMATION ALGORITHM USING MULTIPLE PROCESSES AND THREADS

Saúl Calderón and Gabriel Alvarado
Research Center for Information and Communication Technology
University of Costa Rica
San Jose, Costa Rica
email: saul.calderonramirez@ucr.ac.cr, a60258@ecci.ucr.ac.cr

Arturo Camacho
School of Computer Science and Informatics
University of Costa Rica
San Jose, Costa Rica
email: arturo.camacho@ecci.ucr.ac.cr

ABSTRACT
In this work we implement a parallel version of the pitch estimation algorithm Aud-SWIPE. The new implementation, Aud-SWIPE-P produces a remarkable acceleration, making it usable in applications where it was not efficient before because of its high computational cost. We parallelized the algorithm using multiple threads and processes to speed up different steps of the algorithm, and take advantage of different architectures. The goal for Aud-SWIPE-P’s performance was to achieve an execution time below the signal duration, opening the possibility to create a real-time software using Aud-SWIPE as pitch estimation algorithm. The implementation was evaluated running the algorithm in different real-world scenarios. The acceleration achieved was around 4.4, compared with the sequential version of the algorithm. In sound files with a sampling frequency of 10 kHz or less, the real-time execution goal was fully achieved.

KEY WORDS
Pitch detection, multiple threads, multiple processes, MPI-2, OpenMP, performance.

1 Introduction
Given a complex tone, its pitch is defined as the frequency of the pure tone that best matches its perceptual height. Pitch estimation has multiple applications such as language learning assistance [1, 2], emotion recognition [3, 4], music transcription [5, 6, 7], among others. Several algorithms have attempted to solve the pitch estimation problem, being Aud-SWIPE [8] among the most successful. This algorithm has proved to be an effective pitch estimator with a relatively low miss rate, compared with other algorithms. However, Aud-SWIPE’s computational cost is rather high, possibly limiting its usage.

The current work aims to improve Aud-SWIPE’s execution time. Multiple-core and multiple-processor architectures bring opportunities to improve Aud-SWIPE’s performance using either a multiple threading or a multiple processes approach. We used both approaches, initially evaluating the algorithm’s bottlenecks and lastly, identifying in which steps of the algorithm we should use a multiple process or a multiple thread approach. To create multiple processes we used the MPI-2 standard (MPICH2 implementation) and to breed multiple threads, OpenMP’s API [9]. The MPI-2 standard allows users to create processes dynamically and it has proven to produce good performance and stability in multiple processor systems. OpenMP simplifies multiple threads implementation, allowing users to create threads writing simple pre-processor directives (useful in data level parallelism).

The major difference between using the multiple process and multiple threads approach consists in the memory management. Breeding two or more processes implicates creating a new memory address space, making a message passing protocol to share information between them necessary. For multiple threads, the memory space is shared, hence variables can be accessed by different threads. Using a multiple process technique is going to enable the application to use multiple processors that do not share memory such as cluster systems.

Frequently, cluster systems also have a multiple core architecture in every processor, hence multiple threads running the application will also benefit its performance. Multiple core architectures are standard in today’s desktop computer systems, and also are becoming more popular in mobile systems, meaning that these architectures might also have a performance boost running the application. Implementing a multiprocessor approach needs to take into account memory management issues that will affect the end-performance of the application, such as the amount of data that is passed through the processes and number of times that is done through the code. Also, creating a process has a performance consideration to take into account, given the creation of a new copy of the program’s address space. These aspects of the multiprocessor approach need to be considered as it brings an additional overhead to the application’s performance, and might wipe out any performance improvement achieved by the multiple processes running in the application.

DOI: 10.2316/P.2013.795-013
2 SWIPE and Aud-SWIPE

In order to define the steps of the algorithm to be parallelized using multiple processes and threads, in what follows, we detail the basic workflow of Aud-SWIPE.

2.1 SWIPE

The Sawtooth Waveform Inspired Pitch Estimator (SWIPE) was developed by Camacho and Harris [10]. Its main idea was taken from previous pitch detectors. It consists in measuring the average distance between valleys and peaks on the spectrum (APVD), at harmonics of the pitch. This is achieved through the use of kernels like the one shown in Figure 1.

To build and apply SWIPE’s kernel to the signal spectrum, the following main aspects were taken into account:

Using the square root of the spectrum: SWIPE’s kernel is applied to the square root amplitude of the spectrum. Other algorithms use the logarithm or the square of the amplitude spectrum, producing the negative effects described in [10].

Using a variable window size: SWIPE uses different window sizes for different candidates to produce a better fit between the spectral lobes and the kernel. This is achieved through the use of Hann windows of size \( 8/f \), where \( f \) is the frequency of the pitch candidate.

Weighting the amplitude of the harmonics: Giving the same weight to every harmonic may cause that subharmonics of the pitch get the same or a better score than the true pitch. To solve this problem, SWIPE applies a weight \( 1/k^{1/2} \) to the \( k \)-th harmonic.

Blurring of the harmonics: Observing the spectrum only at harmonic locations of the pitch candidate may lead to considerable chance of error in the pitch detection of inharmonic signals. To alleviate this, SWIPE uses a cosine as a smooth weighting function.

Number of harmonics: Pitch detectors may use a fixed number of harmonics or as many as possible (limited by the Nyquist frequency). The creators of SWIPE found that using harmonics up to 3.3 kHz for speech and 5 kHz was enough to achieve the maximum effectiveness possible.

Warping the frequency: The human ear, more specifically the cochlea (the biological system that perceives sound), has a nearly logarithmic distribution of frequency sensitivity, meaning that it is more sensible to changes in low frequencies than in high frequencies. SWIPE takes this into account to warp the frequency scale, taking more samples in low frequency region than in the high frequency region. There are several scales that aim to model the frequency response of the human ear. For SWIPE, it was found that the ERB scale produced the best results among them [10, 11].

2.2 Aud-SWIPE

Aud-SWIPE is a pitch estimation algorithm based on SWIPE. Aud-SWIPE adds a pre-processing stage to improve SWIPE’s response to sounds that have an absence of the fundamental frequency or some low order harmonics. Signals such as telephone speech have this particular problem of losing harmonics because of the band limit imposed in the telephone system. The solution to this problem is to recover the missing or weak harmonics of the signal. Aud-SWIPE solves this problem by applying half-wave rectification to the output of a gamma-tone filterbank. This new stage added by Aud-SWIPE consists of the following steps:

Flattening the spectral envelope: The main goal of this step is to amplify the strength of high frequency harmonics, causing all harmonics of the spectrum have approximately the same amplitude. For this purpose Aud-SWIPE uses a filter inspired in the human outer-middle ear.

Isolation of sets of consecutive harmonics: This step applies a gamma-tone filter bank to the flattened spectrum, in order to separate the signal’s harmonics into several channels. This filter bank is based on the cochlea’s frequency response. The collection of the resulting channels is stored in a matrix \( X \). Each row of this matrix corresponds to the response of a different segment of the cochlea.

Channels alignment: Each channel has a different time response. Since high frequency channels have a faster response than low frequency channels, a channel alignment is needed.

Harmonics generation using half-wave rectification: In this step, half-wave rectification is applied to generate new harmonics.
Spectrum calculation: A spectrum is calculated for every pitch candidate $f$ at every channel $c$ using a Hann window of size $8/f$. However, using a different window size for every pitch candidate is computationally expensive. A trade-off is to compute the spectrum using only power-of-two window sizes, and share spectra for all candidates within half an octave of the candidate for which the number of samples in $8/f$ seconds is a power of two. Finally, to compute the score of every candidate, the scores produced by using the two closest power-of-two window sizes are linearly combined to produce the final score of the candidate.

Harmonics filtering and weighting: At every channel, harmonics beyond the characteristic frequency of the channel $f_c$ and low-order harmonics might be excessively strengthened after the half-wave rectification. Hence, those harmonics need to be removed or attenuated. This is achieved by using a slide-shaped filter, with maximum at $f_c$ and minimum at zero.

ESRAS (Enhanced Square-Root Amplitude Spectrum) calculation: The enhanced spectrum with the generated harmonics is created by taking the square root of its amplitude at every channel.

Application of SWIPE’s kernel to ESRAS: Finally the ESRAS is compared with the kernels built by SWIPE, to produce a score for every pitch candidate. Figure 2 shows the workflow of Aud-SWIPE’s implementation.

The original Aud-SWIPE’s code was written in Matlab, but was ported to C in order to use multiple processes and threads in an open software environment. Both, the MATLAB and C version of the algorithm receive as parameters the pitch candidates range $f_{\text{min}}$–$f_{\text{max}}$, the hop size $dt$, and the minimum score (threshold) the best candidate must reach in order to be recognized as a valid pitch. Otherwise, the signal is declared as having no pitch.

3 Aud-SWIPE execution-time analysis and bottleneck identification

Aud-SWIPE’s performance is influenced by two key variables: pitch candidates range and signal length (sound length and sampling frequency). The first one is important since as the frequency band is increased, there are more pitch candidates, and also, more window sizes need to be used. The second variable, the signal length, has an obvious impact on Aud-SWIPE’s performance, as any computer algorithm has an execution time that depends on the input size.

To enable the identification of bottlenecks, a stressing test bed was designed in order to make time differences between steps visible. It consisted on running Aud-SWIPE’s original C implementation through 15 sound files with individual violin notes, with a time length between 5 s and 6 s. Aud-SWIPE’s pitch candidates are between 196 Hz and 3520 Hz, with $dt = 1$ ms, (violin’s frequency band) and 44 100 samples per second. The processor used for this particular test was the CORE i7 system (see Table 2). Several clocks were embedded in the code, measuring the execution time of the most important steps of the algorithm, as described in Figure 2. Table 1 shows the obtained results. The right-most column shows the proportion of time taken to execute the corresponding step. The results were checked against the real total execution time for every file, and they were consistent with the sum of the execution time in all steps. Steps with an asterisk in Table 1, were executed for every window size. The execution times correspond to the accumulated time in the step, and not the time for every window size.

Table 1 shows that Aud-SWIPE’s main bottleneck is located in the pitch strength (scores) calculation. This is not a surprise since in this step the kernels are built and are compared to the ESRAS. The second most important bottleneck is the half wave rectification. The reason why it takes so much time is not the rectification itself but upsampling the signal before the rectification and downsampling it after. This up and downsampling is necessary to avoid aliasing. The spectrum and ESRAS calculations together are the third most important bottleneck. In these steps, Fourier transforms using different window sizes are computed and summed to obtain the ESRAS. Calculating the Fourier transform has a significant computational cost, which is heightened with the number of times that it is needed, one time for every window size and another one for every channel that is also divided in multiple windows. The fourth step of the algorithm taking the highest amount of time is the isolation in sets of consecutive harmonics, a reasonable fact since the application of the gammatone filterbank is done several times.

4 Parallel Aud-SWIPE

Figure 2 shows the sequential workflow. Every step (block in the diagram) takes as input the output from the previous step. This means that, at a first glance, a multiple instruction (MI) solution cannot be implemented, hence, a single instruction with multiple data (SIMD) approach is more feasible. Also the algorithm creates a lot of data during its execution, for different pitch candidates, window sizes, filter banks, etc. Those operations will be benefited with a data level parallelization. An advantage of using a SIMD approach is that synchronization problems usually are not complex.

To choose in which steps multiple processes and threads will work, we contemplated important characteristics of both parallelization techniques mentioned at the Introduction (Section 1). Hence, the number of processes created must be the minimum necessary, should execute the maximum number of steps possible and communicate with each other with the smallest possible messages. Multiple processes using the MPI-2 API can be created dynam-
Weighting of harmonics for each window size (raised cosine weighting)

Fast fourier transform for each window size (Spectrum calculation)

Isolation of sets of consecutive harmonics into channels (gamma-tone filterbank)

Flattening of spectral envelope (out-mid ear filtering)

Computation of loudness for each window size (square-root-amplitude summation)

Refinement of scores for each window size (linear combinations)

Selection of best candidate (find maximum score)

Pitch

Figure 2. Aud-SWIPE’s workflow.

4.1 Using OpenMP to implement multiple threads in Aud-SWIPE

As observed in Section 3, there are four important bottlenecks in the code: pitch strength calculation, half-wave rectification, spectrum and ESRAS calculation, and isolation in sets of consecutive harmonics (harmonics into channels or the gammatone filtering). Those steps take a considerable amount of time since they are executed several times. This characteristic allows an ease threading parallelization using OpenMP. The parallelization of each step of the algorithm is detailed as follows.

4.1.1 Cochlea’s response calculation parallelization

The cochlea’s response matrix \( X \), as shown in Figure 2, has one channel per row, representing the cochlea’s time response of a given frequency band or channel. This process takes a considerable amount of computer resources, as seen in Table 1. As seen in Figure 2, building the cochlea’s response matrix requires applying \( N \) gammatone filters (approximately \( N = 32 \), with one channel per filter). After it, the half wave rectification is applied on every channel. These steps would execute faster if, ideally, a different thread \( T_i \) (\( i = 1, 2, ..., N \)) computes the output of its corresponding gammatone filterbank \( i \), and also computes the upsampling, the half wave rectification and the downsampling of its output.

4.1.2 Fourier Transform of matrix \( X \)’s channels

For every window size, the matrix with the cochlea’s response in all channels is converted to the frequency domain, using the Fourier transform, producing matrix \( \hat{X} \). After this, we take its square root and sum it over channels to obtain the ESRAS, which is an approximation to the specific loudness of the signal. It is stored in a matrix \( L \). Finally the algorithm applies SWIPE’s kernels to obtain the score for every pitch candidate (matrix \( S \)). This is done for \( M \) windows (\( M \) depends of the signal length and its sampling frequency). To make every thread independent from each other, each thread \( T_j \) (\( j = 1, 2, ..., M \)) process, ideally, a time window \( j \), calculating the spectrum on every channel for the same window \( j \). This allows the thread \( T_j \) to build the ESRAS, without waiting for the other threads to finish.

4.1.3 Score matrix calculation parallelization

The step corresponding to the pitch strength calculation (scores) is the most expensive step of the algorithm. This step takes the ESRAS or specific loudness matrix \( L \) as input (with one row per window), and applies SWIPE’s kernel in each channel, one window at a time, to obtain the score of the pitch candidate. For each window size, there is a set of corresponding pitch candidates that build the score
matrix $S$. The execution time is proportional to the number of pitch candidates $K$. To parallelize the execution for this step, ideally, each thread $T_k'' (j = 1, 2, ..., K)$ takes the pitch candidate $k$, and calculates its score using SWIPE’s kernel.

### 4.1.4 Using MPI-2 to implement multiple processes in Aud-SWIPE

As seen in previous sections, the steps that produce the largest bottlenecks in Aud-SWIPE have been parallelized already. However, to make use of multiprocessor platforms without shared memory, it is necessary to create multiple processes. This increases the concurrency level of the algorithm as well. As mentioned at the beginning of section 4, it is convenient to create the processes only once and make them cover as much code as possible. As seen in the algorithm description in Figure 2, Aud-SWIPE computes the spectrum for $U$ window sizes, to finally compute the ES-RAS in all channels. Hence, ideally, each window size $u$ should be computed by processes $P_n'' (n = 1, 2, ..., NP)$, hence in this case $NP = U$.

Then, the distribution of tasks for each process $P_n''$ is implemented as follows:

1. The master process $P_1''$ opens the sound file, and computes matrix $X$ (isolation of sets of consecutive harmonics into different channels) and half-wave rectifies the outputs. Upsampling and downsampling by a factor of two is performed before and after rectification, respectively, to reduce aliasing. After that, the master process creates $NP - 1$ new processes. As mentioned at the beginning of Section 4, the number of processes $NP$ vary depending on the desired parallelization degree (PD) and the number of window sizes. The parallelization degree ranges from 0 to 1, where 0 means $NP = 1$ and 1 means that $NP = U$.

2. The master process sends matrix $X$, containing one channel per row, to the $NP - 1$ slave processes.

3. Window sizes are distributed through the $NP$ processes using a round-robin algorithm, ensuring that the workload is distributed evenly, in case that $NP < U$.

4. Each slave $P_n'' (n = 2, ..., NP)$ sends back the scores calculated for its range of pitch candidates to the master, which builds the final score matrix $S$ and determines the winner candidate at every instant of time, returning them in a vector $p$.

Figure 3 shows the parallelization scheme implemented in Aud-SWIPE-P.

### 5 Tests

Important aspects to measure in the parallelized version of Aud-SWIPE (Aud-SWIPE-P) are the following:

1. Execution time of Aud-SWIPE-P with different degrees of parallelization, to identify the best degree for every case. Aud-SWIPE-P was executed in the test bed defined in Table 3. The test bed consists of recordings from different instruments, playing one note at a time. After testing Aud-SWIPE-P with every possible parallelism degree (PD), the best PD is compared against execution times from the original Aud-SWIPE implementation. These tests are executed in System A (see Table 2).

2. Execution time of Aud-SWIPE-P with PD = 0 (one process) to compare Aud-SWIPE-P performance with (using OpenMP only) and without MPI-2, and determine the benefit of using MPI-2.

3. Scalability, with a test consisting in a comparison of Aud-SWIPE-P’s execution time in three different systems: A, B, and C, as defined in Table 2. It was run on an extensive number of sound files (4323) from several instruments, with a fixed frequency range of 100-2000 Hz.

### 5.1 Aud-SWIPE-P with different Parallelism Degrees

Figure 4 shows how Aud-SWIPE-P performance scales as more processes are created, and in general, a positive trend of a performance boost is obtained with a higher number of processes working.

In order to measure if the goal of executing a sound file in less time than its length (in seconds), we define the success rate (SR) as 1, if the algorithm is executed in less time than the length of the sound file, and 0 otherwise. The data shown in Table 3 comprises sound files with a sampling frequency of 10kHz and 44.1kHz. Aud-SWIPE-P was executed with the frequency ranges shown in Table 3 and $dt = 0.001$.

Regarding the test results, Table 4 shows the execution time of the sequential version of Aud-SWIPE. The SR does not reach 100% on any instrument in the sequential version. The average SR is close the 40%, and processing all the files from four instruments did not meet the success
Figure 3. Data level parallelization scheme using MPI-2 and OpenMP.

Table 1. Execution time results of Aud-SWIPE. Steps with an asterisk are executed one time per window size.

<table>
<thead>
<tr>
<th>Step</th>
<th>Time [s]</th>
<th>Rel. exec. time [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute raised-cosine weights*</td>
<td>0.004</td>
<td>0.02</td>
</tr>
<tr>
<td>Compute window size pitch strength contribution*</td>
<td>0.021</td>
<td>0.09</td>
</tr>
<tr>
<td>Align Channels</td>
<td>0.070</td>
<td>0.29</td>
</tr>
<tr>
<td>Out-mid ear filter</td>
<td>0.351</td>
<td>1.47</td>
</tr>
<tr>
<td>Gammatone filtering</td>
<td>1.324</td>
<td>5.55</td>
</tr>
<tr>
<td>Spectrum and ESRAS calculations*</td>
<td>4.139</td>
<td>17.35</td>
</tr>
<tr>
<td>Half wave rectification</td>
<td>6.693</td>
<td>28.05</td>
</tr>
<tr>
<td>Pitch strength calculation (SWIPE kernels)*</td>
<td>11.259</td>
<td>47.19</td>
</tr>
<tr>
<td>Total</td>
<td>23.859</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 2. Systems features.

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>Number of cores</th>
<th>Cache Memory (MB)</th>
<th>CPU Frequency (MHz)</th>
<th>RAM size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Intel core i7 2600</td>
<td>8 (4 physical)</td>
<td>8</td>
<td>3400</td>
<td>8</td>
</tr>
<tr>
<td>B</td>
<td>4 × 2 × Intel Xeon 5462</td>
<td>4</td>
<td>6</td>
<td>2800</td>
<td>32</td>
</tr>
<tr>
<td>C</td>
<td>Athlon II X4 620</td>
<td>4</td>
<td>2</td>
<td>2600</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3. Test bed for Aud-SWIPE-P.

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Freq. range [Hz]</th>
<th>N° wind. sizes</th>
<th>Avg. SL [s]</th>
<th>Samp. freq.[kHz]</th>
<th>SL st. dev. [s]</th>
<th>N° files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trumpet</td>
<td>150–1175</td>
<td>4</td>
<td>4.29</td>
<td>10</td>
<td>0.50</td>
<td>226</td>
</tr>
<tr>
<td>Flute</td>
<td>250–2349</td>
<td>4</td>
<td>2.26</td>
<td>10</td>
<td>1.02</td>
<td>212</td>
</tr>
<tr>
<td>Oboe</td>
<td>250–1760</td>
<td>3</td>
<td>1.49</td>
<td>10</td>
<td>0.29</td>
<td>110</td>
</tr>
<tr>
<td>Tuba</td>
<td>37–350</td>
<td>4</td>
<td>1.65</td>
<td>10</td>
<td>1.06</td>
<td>148</td>
</tr>
<tr>
<td>Clarinet</td>
<td>131–1760</td>
<td>4</td>
<td>2.75</td>
<td>10</td>
<td>0.11</td>
<td>13</td>
</tr>
<tr>
<td>Bass</td>
<td>30–500</td>
<td>6</td>
<td>1.95</td>
<td>44.1</td>
<td>1.41</td>
<td>65</td>
</tr>
<tr>
<td>Piccolo</td>
<td>523–3951</td>
<td>4</td>
<td>4.53</td>
<td>44.1</td>
<td>0.04</td>
<td>11</td>
</tr>
<tr>
<td>Viola</td>
<td>131–2100</td>
<td>5</td>
<td>2.27</td>
<td>44.1</td>
<td>1.76</td>
<td>39</td>
</tr>
<tr>
<td>Violin</td>
<td>196–3520</td>
<td>5</td>
<td>6.11</td>
<td>44.1</td>
<td>0.79</td>
<td>15</td>
</tr>
</tbody>
</table>
condition of finishing the algorithm with a lower time than the signal length. The instrument with the longest average execution time is the violin (23.9 s). This is due to the large pitch range of the instrument and the longest average file length, with a high sampling frequency (44.1 kHz).

In Table 5 the best PDs were chosen for every instrument, showing a substantial performance improvement. The highest SR possible was obtained for four instruments. All the samples from the piccolo failed to be executed in less time than the sound length of the sample, and also the violin samples had a low average of success rate. However in those bad cases, the algorithm is very close to be executed in less time than the sound length (with an average time difference of 1% or 5% from the average sound length). Using the best PD, the average SR increases from 39% to 76%. In the worst cases, the high sampling frequency, the long sound length, and the big frequency band to search for pitch candidates influences more to reach an unsatisfactory result.

Regarding PD, Table 5 shows that the value that produces the best performance is around 75% and 100%.
6 Conclusion

A detailed knowledge of the algorithm to be parallelized allowed us a fast and effective bottleneck identification. The bottlenecks found in the steps that generated the cochlea’s channels matrix, the spectrum calculation for each window (applying Fourier transforms), the ESRAS calculation, and the application of SWIPE’s kernel, were the most computationally demanding parts of the algorithm. Using multiple processes and threads with a data level parallelism approach, helped us to attack the bottlenecks in different parts of the algorithm. Threads were used to parallelize the most critical steps of the algorithm, and processes were created to execute multiple steps, reaching a point where each MPI process was handling one or more window sizes, and created multiple threads to apply SWIPE’s kernel to every window of the ESRAS. The results showed an impressive performance boost of Aud-SWIPE-P compared with Aud-SWIPE. An acceleration by a factor of 4 in the best system tested (System A in Table 2), using both multiple threads and processes, is a satisfactory achievement. However, there is still room for improvement, since the success rate moved from 39% to 76%. Aud-SWIPE-P had problems to fulfill the performance goal of estimating pitch in less time than the duration of the recording. In files with more than 4 seconds of length, high sampling frequencies and an extensive pitch candidates range, this was not achieved. Then, we recommend using Aud-SWIPE-P in the smallest pitch candidates frequency band possible, and with monaural sound files with the lowest sampling frequency possible allowed by Nyquist’s law.

Another finding is that in the multi-processor platform available for testing, system C, the performance with multiple processes was actually worse in some cases, because of the message passing overhead. This means that Aud-SWIPE-P scalates better in multicore desktop systems.

Finally, the exploration of a new parallelization approach is necessary. GPUs are an option to enable more threads, since these architectures typically contain hundreds of processors, and they are intended to be used in data level parallelism problems. We expect to work with these architectures in the near future. If you want to use Aud-SWIPE-P code, please send an email, and we will send you the repository address of the project.

Acknowledgements

We thank Kyle Gorman for facilitating his C implementation of SWIPE. Some parts of his code were used in the implementation of Aud-SWIPE-P. Also we used the FIR2 function implementation made by Alexander Iakovlev.

References