Parallelization of a Denoising Algorithm for Tonal Bioacoustic Signals using OpenACC Directives

Jorge Castro*, Esteban Meneses*†
*Advanced Computing Laboratory, Costa Rica National High Technology Center
{jcastro, emeneses}@cenat.ac.cr
†School of Computing, Costa Rica Institute of Technology

***Abstract*—Automatic segmentation and classification methods for bioacoustic signals enable real-time monitoring, population estimation, as well as other important tasks for the conservation, management, and study of wildlife. These methods normally require a filter or a denoising strategy to enhance relevant information in the input signal and avoid false positive detections. This denoising stage is usually the performance bottleneck of such methods. In this paper, we parallelize a denoising algorithm for tonal bioacoustic signals using mainly OpenACC directives. The implemented program was executed in both multicore and GPU architectures. The proposed parallelized algorithm achieves a higher speedup on GPU than CPU, leading to a 10.67 speedup compared to the original sequential algorithm in C++.**

***Index Terms*—OpenACC, Graphic Processing Unit (GPU), parallel computing, bioacoustics, denoising.**

I. Introduction

Noise cancellation algorithms are important in many digital signal processing applications, like object segmentation and tracking, speech recognition, acoustic detection of animal sounds, among many others. In the field of bioacoustics, it is usually important to remove noise from field recordings to increase the accuracy of automatic detectors and classifiers, and also extract meaningful features from the signal [1]–[3]. Frequently, denoising algorithms require lots of computation time, making them impractical for real-time applications or huge data sets. A method developed to automatically count manatees, using their vocalizations, spends most of its execution time denoising the signal [4]. This denoising algorithm is parallelized in this work to efficiently process field recordings of manatee vocalizations. Also, since the main assumption of the method is that the signal is periodic or tonal it could be suitable for other species that produce tonal vocalizations (e.g. birds).

Graphic Processing Units (GPUs) have been successfully used to accelerate complex denoising algorithms in audio and images [5]–[7]. However, GPU perfomance depends on effectively using the massive number of cores on a GPU and minimizing memory transfers between CPU and GPU. Therefore, algorithms must be highly parallelizable and computationally intensive. Some low-level programming languages like CUDA and OpenCL enable to fully take advantage of GPU performance, but require high training and development efforts. On the other hand, OpenACC offers a balance between productivity and performance. It is based on high-level compiler directives or pragmas (like OpenMP) that, in some cases, can achieve similar performance to that obtained with CUDA or OpenCL [8]–[10]. Also, the OpenACC standard ensures portability and interoperability with other parallel programing standards (CUDA, OpenMP, MPI). Therefore, we decided to use OpenACC to parallelize the aforementioned denoising algorithm.

Here are the highlights of this paper:

- Section V explains how we incrementally parallelized the denoising algorithm by identifying execution bottlenecks and tackling them iteratively, using mainly OpenACC directives.
- An experimental evaluation of the accelerated algorithm, both in terms of performance and scalability, is detailed in section VI.
- A brief discussion about technical issues of the proposed parallelization and limitations of OpenACC is presented in section VII.

II. OpenACC directives

The OpenACC standard is designed for heterogeneous computing, where there is a host device and an accelerator device with separated memories. Usually, most parts of the code run on the host device (normally the CPU) and the most computationally intensive parts run on the accelerator device (normally the GPU). This is because the host is usually optimized to execute sequential operations while the accelerator is optimized to execute parallel operations with high throughput.

OpenACC is a directive-based programming model that supports incremental parallelism like OpenMP. However, OpenMP directives are prescriptive while OpenACC directives are descriptive. That means that a program parallelized with OpenMP will always be executed the same way independently of the hardware it runs on. Conversely, the compilation of a program parallelized with OpenACC depends on the target architecture. Therefore, OpenACC relies heavily on the compiler's capacity to efficiently map the exposed parallelism to the architecture used.

OpenACC defines three levels of parallelism: gang, worker, and vector. Each gang has at least one worker and the size of each worker is equal to the vector length. Thus, gang parallelism is the coarsest grained parallelism, where each gang has its own cache memory and works independently of other gangs. Worker parallelism is an intermediate level

where all workers share the same cache memory and perform vectorized operations. Finally, the vector level is the finest grained parallelism, where one single instruction operates on multiple data.

The OpenACC directives used in this paper can be classified into parallelization directives, data directives and advanced directives. Parallelization directives allow programmers to express parallelism in functions and loops. We used the following parallelization directives:

- `routine`: This directive is added to the definition of each function called in an OpenACC region. The clauses `gang`, `worker`, or `vector` may also be added to specify the level of parallelism used within the function. Also, the clause `seq` indicates that the function should run sequentially.
- `parallel loop`: This directive is placed just before a loop to inform the compiler that it is safe to parallelize it and also allow the compiler to choose how to schedule each iteration of the loop on the accelerator. It also enables the programmer to specify the level and type of parallelism used within the loop. A nested loop inside of an already parallelized outer loop can be also parallelized using the `loop` directive alone.

The data directives detail the data movement between the host device and the accelerator device. There are structured and unstructured data regions in OpenACC. The structured data regions begin and end in the same scope and are defined using the `data` directive. On the other hand, the unstructured data regions can begin and end in different scopes and are delimited using the `enter data` and `exit data` directives. Additionally, there are data clauses to specify how and when data should be allocated or moved between the host and the accelerator. We used the following data clauses:

- `create`: Allocates memory in the accelerator for the listed variables and releases it at the end of the data region.
- `copyin`: Copies the listed variables from the host to the accelerator at the beginning of the data region. Then, it releases the memory at the end of the data region.
- `copyout`: Allocates memory in the accelerator for the listed variables and copies them back to the host at the end of the data region.
- `present`: Indicates the compiler that the listed variables are already present in the accelerator.

Also, we used the `update self` data directive to update the values of a list of variables in the host memory with their values in the device memory. The `update device` was also used to perform the inverse operation.

The advanced directives allow to overlap data movement and computation operations and also coordinate the work of multiple GPUs. We used the following advanced directives:

- `async`: This directive is added to the `parallel` and `update` directives to indicate that once the target operation is sent to the accelerator, the host can continue executing the next instructions instead of waiting for the accelerator operation to end. Also, several asynchronous queues can be used, such that all operations placed in the same queue are executedd in order, while the operations placed in different queues are executed in any order. Thus, data movement and computation operations placed in different queues can be executed simultaneously.
- `wait`: Indicates the host to wait for the past asynchronous operations.

III. Experimental setup

The experimental data used in this research consists of a 9 minute recording of manatee vocalizations taken from the dataset used in the original paper [4]. This recording was sampled at 96 kHz. For the scalability experiment we split this recording into 1, 3, 5, and 7 minute recordings.

The architecture used to run the experiments and profile the program was an Intel Xeon quadcore CPU E3-1225 v5 @ 3.30GHz equipped with a Tesla k40c GPU, which is part of Kabré Supercomputing System at CeNAT. For the final speedup results we run each version of the denoising algorithm ten times. The PGI utility `pgcudainit` was executed before running any speedup experiments, in order to remove the GPU initialization time at each run.

The PGI compiler pgc++ 18.1 was used to build the code. We used the flags `-fast`, `-ta=tesla`, `-ta=multicore` to enable vectorization, compile to NVIDIA Tesla architecture and compile to Intel Xeon architecture, respectively.

IV. Denoising algorithm

The denoising algorithm parallelized in this work is based on a wavelet thresholding using the autocorrelation function (ACF) and a dynamic clustering algorithm. The essential idea behind this algorithm is that tonal vocalizations present a slow decaying ACF, unlike most noise. So, this property is exploited in the wavelet domain where Gaussian noise can also be removed more easily and a dynamic clustering method allows the algorithm to adapt to variations in the periodicity of vocalizations and ambient noise. Fig. 1 shows the block diagram of the algorithm, which we divide in four stages: A, B, C, and D.

In stage A the noisy recording is read and processed by a high-pass filter to eliminate noise outside the frequency range of the vocalizations. In stage B the undecimated discrete wavelet transform (UDWT) is calculated over 3 ms windows, then the ACF is applied to the wavelet coefficients of each decomposition level and its rms value is calculated over a predetermined lag interval. All the wavelet coefficients are stored in a matrix $W_{J\times N}$, where J is the number of decomposition levels plus one and N is the size in number of samples of the input signal. Also, all the rms values are stored in a matrix $M_{J\times I}$, where I is the number of windows. In stage C a moving average, a dynamic clustering, and a vocalization size restriction are applied to each row of matrix $M_{J\times I}$ in order to select the threshold to apply to the wavelet coefficients in stage D. The moving average helps attenuate short-noise sounds. Then, the clustering method, based on

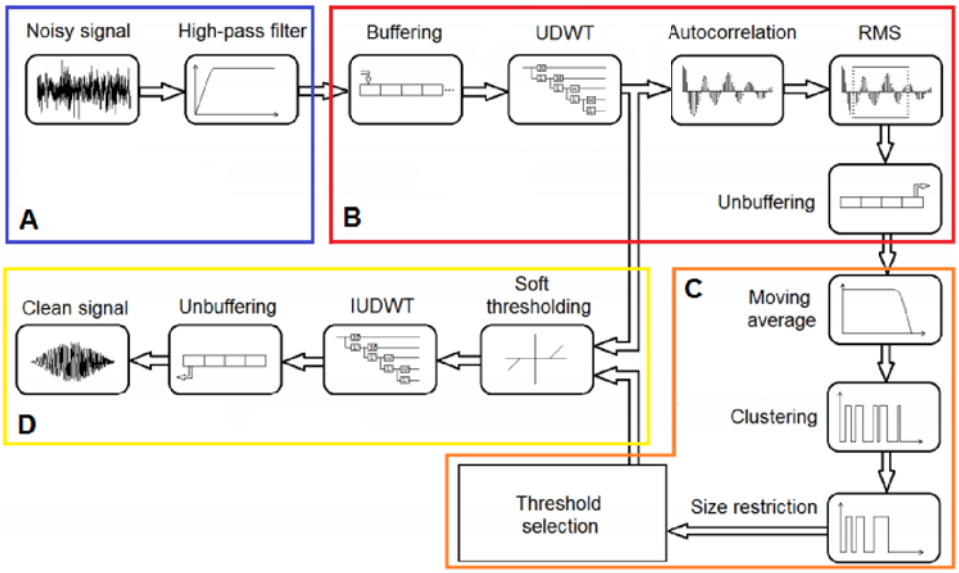

Fig. 1. Block diagram of the denoising algorithm.

a k-means algorithm evaluated over multiple window sizes, separates each element in one of two possible classes using the maximum class centroids separation as criteria. The classes, represented by zero and one, indicate the absence or possible presence of a vocalization, respectively. Later, short sequences of zeros are switched to ones, while sequences of ones longer than the maximum or shorter than the minimum length of a vocalization are switched to zero. In stage D, the resulting matrix $H_{J\times I}$ is used to select the threshold to apply to the wavelet coefficients using the soft thresholding rule and the inverse undecimated discrete wavelet transform (IUDWT) is calculated to obtain the denoised signal. The threshold applied to the wavelet coefficients $w_i(j,k)$ at each window i and level j is calculated as:

$$\eta_{i,j} = \begin{cases} \max_k\{w_i(j,k)\} & \text{if } H(j,i) = 0 \\ \sigma(2\log(N_w))^{1/2} & \text{if } H(j,i) = 1, \end{cases} \quad (1)$$

where k is the shift parameter, N_w is the size of the window in number of samples, $\sigma(2\log(N_w))^{1/2}$ is the universal threshold [11], and σ is the noise level, which is estimated as the median absolute deviation (MAD) of the high frequency coefficients of the first decomposition level. The soft thresholding rule sets to zero all coefficients with absolute value less or equal than the threshold and shrinks to zero the other coefficients by a value equal than the threshold. Finally, the IUDWT is applied to get the clean signal.The original paper features a more detailed explanation of the denoising method. We use the same parameters values of the original denoising algorithm, designed for tonal manatee vocalizations [4].

V. Parallelization of the Denoising Algorithm

The original denoising algorithm is implemented in Matlab and spends about 761 seconds processing the nine minute recording of manatee vocalizations, sampled at 96 kHz. This execution time is longer than the input recording, which makes the algorithm impractical for processing huge datasets. So, the first step to accelerate the program is to implement it in C++. Once this was done, we obtained an initial 11.36 speedup. Then, we profiled the program to understand which stages were more time-consuming. As shown in Table I, stages B and D were the most time consuming, amounting to 93.55% of the total execution time. Therefore, we first focused on parallelizing these stages. Following Amdahl's law, the theoretical maximum speedup S_{max} that can be achieved if we parallelize stages B and D is:

TABLE I
Profiling results of the sequential denoising algorithm in C++ using a 9 minute recording of manatee vocalizations.

Stage	Duration (s)	Contribution
A	0.90	1.33%
B	32.67	48.62%
C	3.44	5.12%
D	23.46	44.93%
Total	67.19	100%

$$S_{max} = \frac{1}{(1-p)} = \frac{1}{(1-0.9355)} = 15.50. \quad (2)$$

A. Parallel Loops and Data Movement

Algorithm 1 shows the pseudocode of stage B. The input for this stage is the high-pass filtered recording $\hat{x}$ and the outputs are the wavelet coefficients of all windows $W_{J\times N}$ and the rms-values matrix $M_{J\times I}$.

Before parallelizing stage B, we transferred the necessary data from the CPU to the GPU. First, we used the `copyin` directive to only move vector $\hat{x}$ to the GPU, since it is not necessary to copy it back to the CPU. Then, we used

Algorithm 1 Stage B

Input: $\hat{x}$
Output: $W_{J\times N}$, $M_{J\times I}$
1: **for** $i = 0 : I$ **do** ▷ Window loop
2: $\hat{x}_i = \hat{x}[i * N_w : i * N_w + N_w]$
3: $w_i = \text{UDWT}(\hat{x}_i)$
4: $W[:, i * N_w : i * N_w + N_w] = w_i$
5: **for** $j = 0 : J$ **do** ▷ level loop
6: temp_vec = ACF($w_i[j, :]$)
7: $M[j, i]$ = rms_val(temp_vec)
8: **end for**
9: **end for**

the unstructured data directive `enter data create` to allocate memory in the GPU for matrix $W_{J\times N}$ and keep it for stage D. Finally, we used the `copyout` directive to allocate memory for matrix $M_{J\times I}$ and copy it back to the CPU at the end of stage B.

The initial parallelization of stage B was pretty straightforward, since each window and decomposition level is processed independently. First, we parallelized the outer for loop using a `parallel loop` directive. Then, we vectorized functions on lines 2, 3, and 4. Afterwards, we parallelized the inner for loop using a `loop` directive. Finally, we vectorized functions in lines 6 and 7.

Algorithm 2 shows the pseudocode of stage D. The inputs for this stage are the wavelet coefficients of all windows $W_{J\times N}$ and matrix $H_{J\times I}$. The output of this stage is the denoised signal y.

Algorithm 2 Stage D

Input: $W_{J\times N}$, $H_{J\times I}$
Output: y
1: **for** $i = 0 : I$ **do** ▷ Window loop
2: $w_i = W[:, i * N_w : i * N_w + N_w]$
3: $u = \text{MAD}(w_i[0, :])$
4: **for** $j = 0 : J$ **do** ▷ level loop
5: temp_vec = $w_i[j, :]$
6: **if** $H[j, i] == 0$ **then**
7: temp_vec = temp_vec $* 0$
8: **else**
9: temp_vec = soft_thresh(temp_vec,u)
10: **end if**
11: $w_i[j, :]$ = temp_vec
12: **end for**
13: $y[i * N_w : i * N_w + N_w]$ = IUDWT(w_i)
14: **end for**

To avoid unnecessary data transfers to the GPU we used the `present` directive to indicate that matrix $W_{J\times N}$ is already present in the GPU. Then, we used the `copyin` directive to move matrix $H_{J\times I}$ to the GPU. Finally, we used the `copyout` directive to allocate memory for the output vector y in the GPU and copy it back to the CPU at the end of stage D. All the data movement directives were placed just before the outer loop.

TABLE II
SPEEDUP OBTAINED AFTER PARALLELIZING THE LOOPS AND SPECIFYING THE DATA MOVEMENT OF STAGES B AND D.

Program	Duration (s)	Speedup
C++ Sequential	67.19	1
Multicore	23.39	2.87
GPU	65.20	1.03

Since each iteration of the window and level loop are also independent in this stage, we parallelized it following a similar scheme to the one used on stage B. First, we parallelized the window and level loop using a `parallel loop` and a `loop` directive, respectively. Then, we vectorized all the functions and data copies, corresponding to lines 2, 3, 5, 7, 9, 11, and 13.

After adding all the aforementioned directives to the code, we recompiled the program for both multicore and GPU architectures and measured the execution time again, as Table II shows. Contrary to what we expected, the speedup obtained with the GPU was negligible. However, the speedup obtained with the CPU was almost 3. Since we were processing a long audio recording, our first hypothesis was that the data movement was the bottleneck of the program. Therefore, we proceeded to move the data asynchronously between the CPU and GPU.

B. Asynchronous Data Movement

To move the data asynchronously between the CPU and GPU we first divided the outer loop of stages B and D into blocks of windows. Thus, we added an additional loop before the window loop that iterates through blocks of 128 windows. This parameter was empirically adjusted.

In stage B, we changed the `copyin` directive used to move $\hat{x}$ to the GPU to both a `create` and `update` directives. The `create` directive was placed just before the block loop to allocate memory in the GPU for $\hat{x}$. The `update` directive was placed just before the window loop to iteratively move each block of $\hat{x}$ to the GPU. Then, we added the `async` clause to the `update` and `parallel loop` directives in order to overlap data movement and computation time. We used 4 asynchronous queues since it produced the best results. Finally, we added a `wait` directive at the end of the block loop.

In stage D, we eliminated the `copyout` directive used to move the output vector y from the GPU to the CPU at the end of this stage. Instead, we used the `create` and `update` directives. The `create` directive was placed just before the block loop, to allocate memory in the GPU for vector y. The `update` directive was placed at the end of the block loop to iteratively move vector y from the GPU to the CPU. Later, we added the `async` clause to the `parallel loop` and `update` directives in order to overlap data movement and computation time. We also used 4 asynchronous queues since

TABLE III
SPEEDUP OBTAINED AFTER ASYNCHRONOUSLY MOVING DATA IN STAGES B AND D.

Program	Duration (s)	Speedup
C++ Sequential	67.19	1
Multicore	23.39	2.87
GPU	37.32	1.80

TABLE IV
COMPARISON OF RUNTIME DISTRIBUTION AFTER MOVING DATA ASYNCHRONOUSLY IN STAGES B AND D.

Stage	Multicore	GPU
A	3.81%	2.40%
B	40.98%	6.43%
C	16.61%	9.67%
D	38.60%	81.50%

it produced the best results. Finally, we added a `wait` directive immediately after the block loop.

Table III shows the speedup obtained once we applied all the changes. Although the runtime has been reduced almost to half, the GPU version of the program is still slower than the CPU version and it is very far from the upper bound according to Amdhal's law. Thus, we profiled both program versions to investigate the difference in execution time at each stage. Table IV shows that there are significant differences in the execution time distribution of both programs. It also shows that the main bottleneck of the GPU program is in stage D. Therefore, in the next section we focus on determining what was causing this bottleneck in the GPU version and how to eliminate it.

C. Optimization of Median Computation

After inspecting the code of stage D, we realized that the performance bottleneck was caused by the computation of the universal threshold to be applied to the wavelet coefficients. The universal threshold is computed at line 3 of Algorithm 2 through the MAD function. This function first calculates the median of the input vector. Then, it calculates the median of the absolute difference between each element of the input vector and its median. Thus, the MAD function involves the computation of the median twice. The median was originally calculated by first sorting the array and then taking the $(N_w/2)$-th element, where N_w is the size of the array in this case. The insertion sort algorithm used to arrange the vector, runs sequentially and takes on average ${N_w}^2$ operations. Hence, the bottleneck in stage D is due to the high time complexity of the insertion sort algorithm and the poor sequential performance of the GPU.

To eliminate the performance bottleneck we substituted the insertion sort algorithm with the quick select algorithm [12]. This algorithm does not sort the vector, but finds the k-th smallest element in a more efficient way. It takes on average N_w operations to find the median.

The performance achieved using the selection algorithm for both CPU and GPU is shown in Table V. This time, the GPU version of the parallelized program obtained a higher speedup than the multicore version. Table VI shows

TABLE V
SPEEDUP OBTAINED AFTER SUBSTITUTING THE INSERTION SORT ALGORITHM WITH THE QUICK SELECT ALGORITHM.

Program	Duration (s)	Speedup
C++ Sequential	67.19	1
Multicore	17.01	3.95
GPU	8.55	7.86

TABLE VI
COMPARISON OF RUNTIME DISTRIBUTION AFTER SUBSTITUTING THE INSERTION SORT ALGORITHM WITH THE QUICK SELECT ALGORITHM.

Stage	Multicore	GPU
A	5.25%	10.41%
B	56.38%	28.06%
C	22.81%	42.40%
D	15.56%	19.12%

how the distribution of execution time changed for the GPU and multicore versions. Both versions presented a significant reduction of the runtime of stage D, but specially the GPU version. Also, due to the high reduction of the runtime of stages B and D, now stage C is the slowest one in the GPU version. Thus, in order to further improve the speedup of the algorithm, we focus on accelerating stage C in the next section.

D. Dynamic Clustering Parallelization

The slowest method in section C is the dynamic clustering algorithm. We focused on parallelizing the most computationally demanding part of this method, which is based on two nested loops. The outer loop iterates over a vector of different window sizes. The inner loop iterates over the windows consecutively extracted from the whole signal. Then, the k-means algorithm is computed for each window, using $k = 2$ and a maximum of 100 iterations. The centroids of the two classes are stored for each window. Finally, after all windows of a specific size are processed, the obtained centroids are interpolated for each signal sample and stored.

The clustering algorithm requires lots of dynamic memory allocation at each iteration of the outer and inner loop. Otherwise, it would require big and complex data structures allocated before the loops. Since the dynamic memory size is very limited in the GPU and the data movement between CPU and GPU is critical, we considered this algorithm unsuitable to be parallelized in the GPU. Instead, we used OpenMP to parallelize it. Thus, we just placed an `omp parallel` directive in the outer loop to distribute the work among the CPU threads.

The speedup obtained after parallelizing the clustering algorithm using OpenMP is shown in Table VII. Both the CPU and GPU versions of the program obtained a similar improvement. Table VIII shows the final distribution of the runtime in both versions. The portion of time spent at stage C is now less than the time spent at stages B or D for both CPU and GPU versions.

TABLE VII
SPEEDUP OBTAINED AFTER PARALLELIZING THE DYNAMIC CLUSTERING ALGORITHM WITH OPENMP.

Program	Duration (s)	Speedup
C++ Sequential	67.19	1
Multicore	14.57	4.61
GPU	6.30	10.67

TABLE VIII
COMPARISON OF RUNTIME DISTRIBUTION AFTER PARALLELIZING THE DYNAMIC CLUSTERING ALGORITHM WITH OPENMP.

Stage	Multicore	GPU
A	6.08%	14.15%
B	66.23%	38.10%
C	9.57%	21.78%
D	18.12%	25.96%

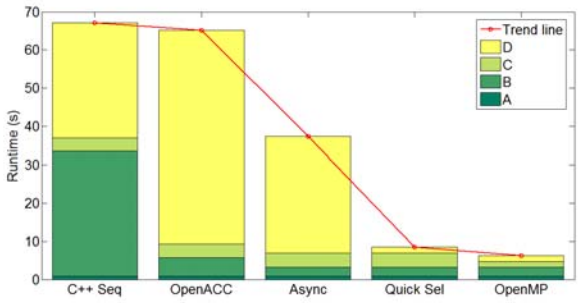

Fig. 2. Runtime decomposition of each improvement of the GPU program.

VI. EVALUATION

A summary of the speedup obtained with each improvement of the GPU program is shown in Table IX. The speedup is calculated with respect to the sequential programs in Matlab (speedup$_M$) and C++ (speedup$_C$). Also, the fully optimized multicore program is shown in the last row of this table for comparison purposes. The coefficient of variation is 0.002 or less for each version of the program. The migration of the Matlab program to C++ reduces the computation time by approximately 695 s, leading to a 11.36 speedup. It is also worth noting that the initial OpenACC version produces an insignificant improvement of 1.03 to the performance of the sequential C++ program. Conversely, the asynchronous data movement and the optimization in the computation of the median produce the most significant improvements in performance of the sequential C++ program, reducing the execution time in approximately 28 s and 29 s, respectively. The highest speedup is achieved by the fully optimized GPU program which lasts only 6.28 s and provides a 10.67 speedup over the sequential C++ version. On the other hand, a fully parallelized multicore CPU version gets only a 4.62 speedup.

The runtime decomposition of each improvement of the GPU program is shown in Fig. 2. In the initial C++ sequential version of the program, stages B and D consume 32.67 s and 30.19 s, respectively, which represent 93.55% of the total execution time. After adding the initial OpenACC parallel and data directives, the total execution time remains almost the same but there are big changes in its distribution. The portion of the total execution time spent on stage B decreases from 48.62% to 7.41%, while it increases from 44.93% to 85.65% for stage D. After moving the data asynchronously, the execution time of stage B drops from 4.83 s to 2.40 s, while it drops from 55.84 s to 30.42 s for stage D. However, stage D still consumes 81.50% of the total execution time. The optimization of the median calculation using the quick select algorithm reduces the execution time of stage D from 30.42 s to 1.64 s. Thus, stage C becomes the slowest one, consuming 3.53 s corresponding to 42.40% of the total execution time. Finally, the runtime of stage C is reduced to 1.37 s by parallelizing the dynamic clustering algorithm using OpenMP. The final distribution of execution time is already shown in Table VIII.

The runtime decomposition of the fully optimized GPU program for different audio file sizes is shown in Fig. 3. The program scales linearly when increasing the input data size. Also, the distribution of the runtime presents small changes according to the data size. The percentage of runtime consumed by stage A increases from 12.66% to 14.04% when increasing data size from 60 s to 180 s, however, it remains almost constant for larger input data sizes. On the other hand, the percentage of runtime consumed by stage B progressively decreases from 48.12% to 38.10% when increasing data size from 60 s to 540 s. Conversely, the percentage of runtime consumed by stage C progressively increases from 15.94% to 21.78% when increasing data size from 60 s to 540 s. Finally, the percentage of runtime consumed by stage D increases from 23.28% to 26.15% when increasing data size from 60 s to 180 s and it remains almost constant for larger input data sizes. Therefore, stage B is the one that best scales and stage C is the one that worst scales when increasing the input data size.

The effect of varying the input data size and block size (explained in section V-B) on the runtime of the fully optimized GPU program is shown in Fig. 4. The runtime decreases from 10.57 s to 6.29 s when increasing the block size from 32 to 128 for a recording of $N = 540$ s. After that point, the runtime remains rather constant up to 2056 block size, where it increased to 6.81 s. This behavior is similar for all data sizes tested.

TABLE IX
COMPARISON OF THE DIFFERENT VERSIONS OF THE DENOISING ALGORITHM.

Code	Arch	Duration (s)	Speedup$_M$	Speedup$_C$
Matlab	CPU	761.74 ± 1.73	1	0.09
C++ Seq	CPU	67.07 ± 0.11	11.36	1
OpenACC	GPU	65.29 ± 0.07	11.67	1.03
Async	GPU	37.24 ± 0.05	20.45	1.80
Quick Sel	GPU	8.53 ± 0.01	89.34	7.87
OpenMP	GPU	$\mathbf{6.28 \pm 0.01}$	**121.20**	**10.67**
Multicore	CPU	14.51 ± 0.02	52.49	4.62

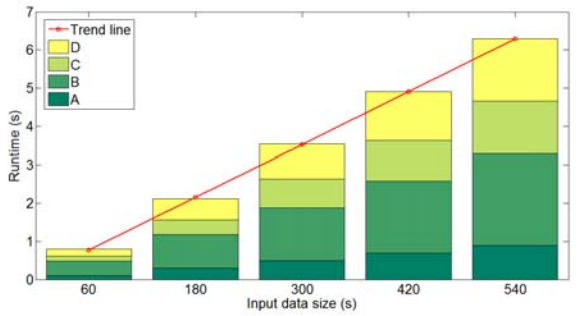

Fig. 3. Effect of input data size on the runtime decomposition of the fully optimized GPU program.

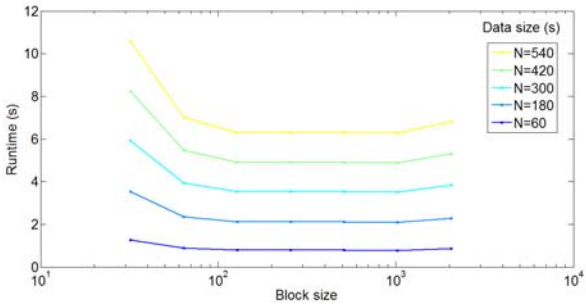

Fig. 4. Effect of varying block and input data size on the runtime of the fully optimized GPU program.

VII. Discussion

The best speedup results presented in table IX are the product of a mixed contribution of OpenACC directives, OpenMP directives and the substitution of the selection sort algorithm with the quick select algorithm. In order to better understand how OpenACC directives affect the obtained speedup, we used the sequential C++ program with the quick select algorithm, as the base version to compare with others. Thus, Table X shows the obtained speedup for each parallelized version of the modified algorithm. In this case, both the initial OpenACC version and the version with asynchronous data movement obtain a higher speedup. However, the final GPU version (OpenMP QS) obtains only a 7.34 speedup. Also, the final multicore version (Multicore QS) provides a 3.18 speedup.

The variability in performance of the parallelized algorithm was negligible for recordings of manatee vocalizations with different levels of noise. Thus, we did not include any of these results. Also, we did not experiment with different number of gangs, workers or vector size in the parallel loops, since we did not want to over-specify the parallelization parameters for the used architecture. This decision also makes the code more performance portable.

TABLE X
Impact of using OpenACC directives on the sequential optimized C++ program.

Code	Arch	Duration (s)	Speedup
C++ Seq QS	CPU	46.13 ± 0.11	1
OpenACC QS	GPU	15.32 ± 0.06	3.01
Async QS	GPU	8.53 ± 0.01	5.41
OpenMP QS	GPU	$\mathbf{6.28 \pm 0.01}$	**7.34**
Multicore QS	CPU	14.51 ± 0.02	3.18

The CUDA unified memory feature, that enables an automatic data movement between the CPU and GPU, was not used since it decreased the performance of our parallelized algorithm. However, it could be a useful tool to quickly obtain preliminary results.

Some limitations of OpenACC were found when allocating dynamic memory and using structs or complex data structures. Thus, it was not possible to allocate dynamic memory and the structs were decomposed into single variables. These limitations affect the clarity of the code and increase the development efforts. Also, as stated in section III, we had to use the PGI utility `pgcudainit` to ensure that GPU initialization time did not affect our experiments.

To use the denoising method with other types of vocalizations the high-pass filter and the number of levels of the wavelet transform have to be adjusted depending on the frequency range of the vocalizations and the sample rate of the recordings. Also, the size of the windows and length restrictions applied to the signal, depend on the duration of the target vocalizations.

VIII. Related Work

Other audio processing algorithms have been parallelized using mostly CUDA and OpenMP. For example, mono aural source separation algorithms have been parallelized using CUDA on a NVIDIA GTX 280, achieving speedups up to 86 [13]. Also, a speech separation algorithm based on recurrent neural networks has been accelerated using CUDA [14]. The non-negative matrix factorization algorithm, used for audio source separation, has also been parallelized using OpenMP and CUDA [15]. In this case, the speedup obtained with respect to the sequential C program was 4.32 using OpenMP and 18.57 using CUDA. In a similar way, the fast Fourier transform has been parallelized using OpenMP and CUDA [16]. This time, for a small number of frames OpenMP achieved a better speedup than CUDA, otherwise, CUDA performed better.

To the best of our knowledge this is the first work that uses OpenACC directives to parallelize a bioacoustic algorithm.

IX. Conclusions and Future Work

In this work we accelerated an audio denoising algorithm by using mainly OpenACC directives, but also some algorithmic optimizations and OpenMP directives. The proposed parallelized algorithm achieves a speedup of 10.67 on a Tesla k40c, when compared to the sequential algorithm in C++. Also, it achieves a speedup of 121.20 on a Tesla k40, when compared to the original algorithm in Matlab. Also, this algorithm scales linearly when the input data increases.

The speedup obtained on a Xeon quadcore CPU is lower than the achieved on a Tesla k40c. It is 4.62 compared to

the sequential algorithm in C++ and 52.49 compared to the original algorithm in Matlab.

Further optimizations could be obtained by parallelizing the moving average or the high-pass filter. However, the speedup may be insignificant due to the small portion of time spent in these methods, even for big input data size. This method can also be further parallelized using MPI to distribute the work among multiple GPUs.

ACKNOWLEDGMENTS

This research was partially supported by a machine allocation on Kabré supercomputer at the Costa Rica National High Technology Center. We thank NVIDIA Corporation for a hardware grant that provided us with a GPU to run the experiments presented on this paper.

REFERENCES

[1] A. Brown, S. Garg, and J. Montgomery, "Automatic and efficient denoising of bioacoustics recordings using mmse stsa," *IEEE Access*, vol. 6, pp. 5010–5022, 2018.

[2] Y. Ren, M. T. Johnson, and J. Tao, "Perceptually motivated wavelet packet transform for bioacoustic signal enhancement," *The Journal of the Acoustical Society of America*, vol. 124, no. 1, pp. 316–327, 2008.

[3] Z. Yan, C. Niezrecki, and D. O. Beusse, "Background noise cancellation for improved acoustic detection of manatee vocalizations," *The Journal of the Acoustical Society of America*, vol. 117, no. 6, pp. 3566–3573, 2005.

[4] J. M. Castro, M. Rivera, and A. Camacho, "Automatic manatee count using passive acoustics," in *Proceedings of Meetings on Acoustics 169ASA*, vol. 23, no. 1. ASA, 2015, p. 010001.

[5] A. Márques and A. Pardo, "Implementation of non local means filter in gpus," in *Iberoamerican Congress on Pattern Recognition*. Springer, 2013, pp. 407–414.

[6] P. Brakel, D. Stroobandt, and B. Schrauwen, "Bidirectional truncated recurrent neural networks for efficient speech denoising," in *14th Annual conference of the International Speech Communication Association*, 2013.

[7] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the gpu–past, present and future," *Medical image analysis*, vol. 17, no. 8, pp. 1073–1094, 2013.

[8] J. Hahnfeld, C. Terboven, J. Price, H. J. Pflug, and M. S. Müller, "Evaluation of asynchronous offloading capabilities of accelerator programming models for multiple devices," in *International Workshop on Accelerator Programming Using Directives*. Springer, 2017, pp. 160–182.

[9] H. Matsufuru, S. Aoki, T. Aoyama, K. Kanaya, S. Motoki, Y. Namekawa, H. Nemura, Y. Taniguchi, S. Ueda, N. Ukita *et al.*, "Opencl vs openacc: lessons from development of lattice qcd simulation code," *Procedia Computer Science*, vol. 51, pp. 1313–1322, 2015.

[10] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. A. Jarvis, "Accelerating hydrocodes with openacc, opencl and cuda," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 465–471.

[11] D. L. Donoho and J. M. Johnstone, "Ideal spatial adaptation by wavelet shrinkage," *biometrika*, vol. 81, no. 3, pp. 425–455, 1994.

[12] C. A. R. Hoare, "Algorithm 65: Find," *Commun. ACM*, vol. 4, no. 7, pp. 321–322, Jul. 1961.

[13] F. Weninger and B. Schuller, "Optimization and parallelization of monaural source separation algorithms in the openblissart toolkit," *Journal of Signal Processing Systems*, vol. 69, no. 3, pp. 267–277, 2012.

[14] F. Weninger, F. Eyben, and B. Schuller, "Single-channel speech separation with memory-enhanced recurrent neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 3709–3713.

[15] E. Battenberg, A. Freed, and D. Wessel, "Advances in the parallelization of music and audio applications." in *ICMC*, 2010.

[16] B. Medetov, A. Koishigarin, A. Yskak, K. Niazaliev, and A. Naurzbayeva, "A comparative analysis of openmp and cuda performance as exemplified by the computation of fourier transform," *Recent Contributions to Physics (Rec. Contr. Phys.)*, vol. 61, no. 2, pp. 108–114, 2018.