Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Survival of the Synthesis - GPU Accelerating Evolutionary Sound Matching

Harri Renney* | Benedict Gaster | Thomas J. Mitchell

Department of Computer Science and Creative Technologies, University of the West of England, Bristol, UK

Correspondence
Harri Renney, UWE Bristol - Frenchay Campus, Coldharbour Ln, Bristol BS16 1QY. Email: harri.renney@uwe.ac.uk

Abstract

Manually configuring synthesiser parameters to reproduce a particular sound is a complex and challenging task. Researchers have previously used different optimisation algorithms, including evolutionary algorithms to find optimal sound matching solutions. However, a major drawback to these algorithms is that they typically require large amounts of computational resources, making them slow to execute. This paper proposes an optimised design for matching sounds generated by frequency modulation (FM) audio synthesis using the graphics processing unit (GPU). A benchmarking suite is presented for profiling the performance of three implementations: serial CPU, data-parallel CPU, and data-parallel GPU. Results have been collected and discussed from a high-end NVIDIA desktop and a mid-range AMD Laptop. Using the default configuration for simple FM, the GPU accelerated design had a speedup of 128X over the naive serial implementation and 8.88X over the parallel CPU version on a desktop with an Intel i7 9800X CPU and NVIDIA RTX GeForce 2080Ti GPU. Furthermore, the relative speedup over the naive serial implementation continues to increase beyond simple FM to more advanced structures. Further observations include comparisons between integrated and discrete GPUs, toggling optimisations, and scaling evolutionary strategy population size.

KEYWORDS:
GPU, Evolutionary Computing, Synthesis, Benchmark, Parallel

1 | INTRODUCTION

Modern technology has had a profound effect on the structure, form and performance of music. Powerful and inexpensive general-purpose systems have made musical apparatus universally available to amateur and professional composers alike. The audio synthesiser is a core component in the development of music, enabling composers to recreate acoustic instrument sounds or explore entirely new sounds electronically. Numerous synthesis techniques have been discovered and enabled the creation of a considerable range of timbres. Synthesisers expose controllable parameters, which shape the sound character of the particular synthesis architecture. Consequently, there is often a complex mapping between the control space of synthesis parameters and the timbre space of the sound's perceived character. Therefore, effective control and navigation of a synthesiser's sound space requires expert knowledge of the underlying technique, often drawing from deep theoretical and/or experiential knowledge. Steps towards achieving an automated process for mapping sound qualities to sound synthesis parameters could make synthesisers a more transparent compositional tool. Achieving this will require techniques that can efficiently search the synthesis parameter space to identify parameter configurations to match specific timbral characteristics.

Abbreviations: ES, evolutionary strategy; GA, genetic algorithm; GPGPU, general-purpose GPU; FM, Frequency Modulation;

In the field of Evolutionary Computation, there are powerful optimisation techniques designed to search complex design spaces and find optimal solutions. Evolutionary algorithms have previously been applied to synthesis parameter matching with promising results[1]. However, a major disadvantage of optimisation algorithms, including evolutionary ones, is that they typically require substantial computation time to converge on optimal solutions. Processing units with data-parallel architectures open up opportunities for increasing computational throughput for appropriate cases of the algorithm. The Graphics Processing Unit (GPU) has matured from its origins in the graphics domain into a general-purpose hardware accelerator known as a general-purpose GPU (GPGPU)[2]. GPGPU computing aims to exploit the full potential of massively parallel processing architectures for general processes[3] and evolutionary sound matching are a suitable class of problems for fine-grained parallel processing, with the potential to map efficiently to the GPU environment. Therefore, using the GPU may provide an opportunity to improve the performance of synthesis parameter matching.

This work investigates the potential performance benefits of GPU processors for synthesiser parameter matching using evolutionary computing. A design is proposed that uses the evolution strategy[4] for searching the parameter space of an FM synthesiser. The performance between CPU and GPU implementations is evaluated using a comprehensive benchmarking suite. The application of evolutionary computation for FM synthesis parameter matching is a well-explored area and has proven to be an effective and accurate technique for both simple and more advanced FM synthesis structures[5] [6] [7]. However, a major disadvantage is that they require considerable computational time, and researchers have highlighted a desire for ways to optimise this approach[8]. Therefore, this work does not advance the state-of-the-art in evolutionary computation for audio synthesis matching, rather, it proposes algorithms that accelerate existing methods by providing a data-parallel GPU design that overcomes challenges faced by the distinct architecture. The contributions of this work can be summarised as:

- A design for a GPU accelerated sound matching framework.
- The implementation of the design in the OpenCL GPGPU environment.
- A benchmarking suite for collecting performance profiles across hardware systems for serial CPU, data-parallel CPU and data-parallel GPU implementations.
- Benchmark results from different systems demonstrate the performance acceleration of the proposed design.

2 | PREVIOUS WORK

There has been extensive prior research surrounding the application of Evolutionary Computation for matching sounds using audio synthesisers. This is typically realised by optimising the parameters of a particular synthesis type using Genetic Algorithms (GA)[9] or Evolution Strategies (ES)[10]. FM has become a common application domain for sound matching, and the GA is by far the most prevalent approach adopted in the literature[9] [11] [12]. For example, in[13], a GA is used for matching sounds using FM synthesis, which uses an advanced timbral extraction technique for assessing the fitness of potential solutions. The authors demonstrate how the advanced technique increases the efficiency of the evolutionary sound matching application.

More recently, Smith[14] demonstrated the accurate matches for an electronic keyboard and piano synthesiser using a modified GA; this approach was considered promising for music producers. However, Smith stressed that the time to find solutions was an essential component when evaluating the practicality of the method. Therefore, optimising the speed of the matching process would be a beneficial next step in future work. Yee-King et al. made a comparative study between several automatic synthesiser parameter matching algorithms[15], particular focus given to neural networks and genetic algorithms. Neural networks proved to find accurate solutions in "near real-time". However, this approach first required a training phase that took a day to complete. In contrast, the genetic algorithm required no pre-training but required 2 hours to find a solution. The advantage of the genetic algorithm is that no training is required when adapting the algorithm for a new synthesiser. However, it still requires significant computation and would therefore benefit from optimisation.

The concept of using the graphics hardware for general-purpose processing was introduced in the 1990s with works like Lengyel et al., who used graphics devices for robot motion planning[16]. GPUs have advanced significantly and are now accessible for general-purpose programming outside of graphics. In 2006, NVIDIA[17] introduced the Tesla GPU, introducing the beginning of a new unified architecture. Contemporaneously, AMD made similar developments with the TeraScale GPU. NVIDIA and AMD have continued to develop comprehensive GPGPU support in all subsequent GPU architectures[18]. GPGPU has been successfully applied for processing a range of numerical and scientific computation techniques like molecular dynamics[19] and audio synthesis[20]. A well-known and successful example of this is the folding@home project[21], which reported a speedup of 20 - 30X when accelerated on the GPU. These examples motivate the exploration and establishment of GPU designs for appropriate methods and applications. Recently, Turian et al.[22] have leveraged the GPU to generate a billion sample dataset for synthesisers, including FM synth timbre and subtractive synth pitch. With the use of the GPU, this dataset is 100x larger than any other recorded datasets in the literature.

The success and nonlinear mapping of FM synthesis has generated interest in designing parameter and sound matching methods. Consequently, this paper's proposed application is designed for targeting FM synthesis in particular. The results in Section 6 starts with benchmarking the GPU accelerated design for the simple FM synthesis algorithm originally proposed by Chowning[23]. This limits the complexity of the synthesis stage and allows the benchmarking results to also emphasise the performance of the audio analysis and evolutionary computing stages. After a complete discussion of the simple FM synthesis results, two more advanced synthesisers are presented, and the evaluation of the GPU accelerated performance beyond simple FM synthesis is explored to determine the scalability of the design's capability to support other FM structures. The design proposed in this work is limited to analysing static sounds and does not support dynamic sounds that change with time. Although, steps towards supporting dynamic sounds by analysing audio in blocks is partially supported at the moment and discussed in the results.

There are other well-studied methods for synthesiser sound matching, such as the hill-climber algorithm. Although contextually applicable, the hill-climber algorithm has been considered insufficient for successful exploitation of FM synthesis parameter space by Horner in[24], whilst the evolutionary algorithm proved superior. Neural networks are another powerful method with superior real-time performance when matching sounds. However, they require extensive pre-training periods that can make them unable to handle dynamic problems. The appeal of the evolutionary approach is that it requires no pre-training and can be easily applied to match sounds using different synthesis techniques. However, the trade-off is that evolutionary algorithms typically take considerable time to then find the solutions. Improving the performance will enable larger population sizes to be used in practical time scales, improving the accuracy and the usability of the approach. The population size is a parameter of the evolutionary algorithm, enabling greater exploration of the search space at the cost of further computation. Therefore, optimising the scaling of the population will improve the evolutionary algorithm's exploration of the problem space more effectively[25]. A natural consequence of increasing the population size is that the computation increases. If the performance can be improved, the range of applications that synthesiser parameter matching can be used for will increase. Although many of the papers referenced here use the genetic algorithm, we adopt a different, albeit nearly identical, evolutionary algorithm called the evolution strategy. Both algorithms model the process of evolution to some degree, but we have chosen to work within the framework of an evolution strategy as its canonical form represents real numbers directly as floating-point numbers and so lends itself neatly to the FM matching domain. However, with the appropriate adjustment of terminology, this work equally applies to genetic algorithms. The literature on evolutionary algorithms for sound matching is well explored and has shown to be an effective method over the alternatives. Using ideally large populations in evolutionary algorithms requires significant computation and researchers suggest that finding ways to optimise this in the context of sound matching will be a beneficial advancement.

2.1 | FM Synthesis

One of the first commercially available and successful digital audio synthesis methods was FM synthesis[23]. Invented by John Chowning, FM synthesis was discovered and initially developed in the 1970s[26]. It is regarded as a highly efficient method for generating complex and rich audio timbres with simple graphs of interconnected sinusoidal oscillators. The original equation proposed by Chowning, with some symbols modified for this paper, is given as:

$$y(t) = A\sin(ct + I\sin(mt)) \tag{1}$$

Where four parameters are exposed: peak amplitude A, carrier frequency c (rad/s), modulation frequency m (rad/s) and modulation index I. Therefore, y generates an instantaneous FM output for a given time t. The modulation frequency m sets the frequency of the modulating oscillator, the output of which is multiplied by the modulation index I to control the intensity of the frequency modulation applied to the carrier oscillator. This is then added to the input for the carrier oscillator's frequency c. Finally, the variable A is used to control the peak amplitude output. When the modulation index $\mathrm{I} = 0$, the modulation oscillator has no effect. When $\mathrm{I} > 0$, the modulation oscillator begins to affect the carrier oscillator, and symmetrically spaced intervals of frequencies occur above and below the carrier frequency. The number of side frequencies relates to the modulation index; therefore, as I increases, energy is taken from the carrier frequency and distributed further across the sideband frequencies. Bessel functions determine the amplitudes of the carrier and side frequencies and the interested reader is referred to Chowning's original work[23] for further details . FM synthesis has a nonlinear mapping between parameters and the resulting sound[27], this means that there is a non-trivial correlation between a synthesiser's parameter space and the resulting timbre space, i.e., minor changes to the input parameters can generate vastly different changes in the resulting sound. This makes it a difficult synthesiser to control and parameter match.

Simple FM synthesis is used as a fundamental building block in more complex and musically interesting synthesisers. The technique was adopted by Yamaha in the DX and TX synthesisers that were commercially produced throughout the 1980's[28] and is still a prevalent technique used in many plugins and synthesisers that are currently available[29]. In this paper, two further synthesisers are considered for GPU optimisation that build upon Chowning's simple FM synthesis technique. The first has been established as a "real-world" by Das et al. and is used to evaluate evolutionary algorithm performance[8]. In this paper, this will be referred to as nested modulator FM synthesis[30]:

$$y(t) = Asin(ct + I_1 sin(m_1 t + I_2 sin(m_2 t))) \tag{2}$$

Where y is the output of the nested modulator FM, *t* is the input variable time, *A* is the peak amplitude, *c* is the carrier frequency, *m* is a vector of modulator frequencies and *I* is a vector of modulation indices. This equation adds a second nested modulation oscillator that modulates the original modulation oscillator of the simple FM structure seen in Equation 1. This arrangement requires a total of 6 parameters to control the output sound. The third FM synthesis method considered in the scope of this paper uses the idea of combining separate FM synthesis components in parallel. For the context of this work, the parallel FM[1] will be used:

$$y(t) = \sum_{i=1}^{3} A_i \sin(c_i t + I_i \sin(m_i t)) \tag{3}$$

Where *y* is the output of the parallel FM, *t* is the input time, *A*, *c*, *m* and *I* are vectors of the peak amplitude, carrier frequencies (rad/s), modulation frequencies (rad/s) and modulation indices for each FM structure. This equation is the summation of three separate simple FM structures, each having four parameters and therefore requires a total of 12 parameters to control the output. This paper begins by considering the GPU optimisations concerning simple FM synthesis and then scales up to assess the nested modulator and parallel FM synthesis methods.

3 | DESIGN

This section covers the abstract overview of the FM synthesis parameter matcher process. Figure 1 are used to aid the reader in understanding the entire flow of the program, considering there are many stages involved. Each of the stages are explained with sufficient detail in the following subsections.

3.1 | Evolution Strategies

Evolution strategies[4] are a class of evolutionary algorithm[31] that is used to find optimal solutions within a search space. Evolution strategies follow ideas inspired by Darwinian evolution and natural selection to create progressively fitter solutions to a problem by iteratively applying mutation and recombination operations on a set or population. Evolution strategies can be used for any optimisation problem, including sound matching. In this paper, the algorithm is used to search the parameter space of an FM synthesiser to find the parameters that closely replicate a given target sound[10].

A population of solutions comprises a set of individuals where each individual contains a complete set of synthesis parameters that can be used to generate a candidate sound or solution to the sound matching problem. For the simple FM synthesiser, this corresponds to the four parameters described in Equation 1, along with an additional set of four step-size parameters used by a self-adaptive mutation operator.

Recombination blends the genetic material (or parameters) from two or more parent individuals' to generate new offspring. Figure 2, shows how the uniform discrete recombination[31] operator works, which is one of the standard ES recombination operators used in this work. The value at each position is taken from a random parent in the current population and combined to create a new individual, used in the next generation offspring population.

Figure 3, shows how the mutation operator typically works. The random values shown in red are generated for each element using a pseudorandom number generator and Gaussian Distribution[32], scaled in proportion to the step size. These values are then added to the individual parameters to introduce novelty to the population. Gaussian distribution is used to increase the likelihood of smaller mutations occurring more frequently than larger values. This results in smaller steps in values but occasionally larger steps which can be useful to explore the search space and to escape local optima. The self-adaptive mutation operator uses the step size parameter to enable distant values (exploration) as well as precise local values (exploitation) to be generated at different stages of evolution.

3.2 | Fitness

The evolutionary strategy's fitness function (or objective function) is context dependant and defined by the optimisation problem. For synthesiser parameter matching, this function requires two stages. First, a clip of audio is generated by the chosen FM synthesiser using the parameters of

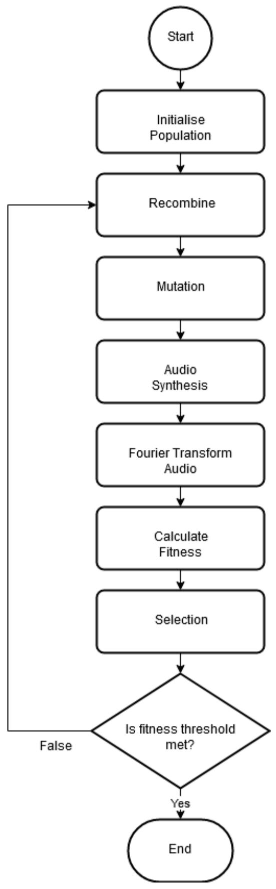

FIGURE 1 Flow diagram for the overall evolutionary parameter matching design flow.

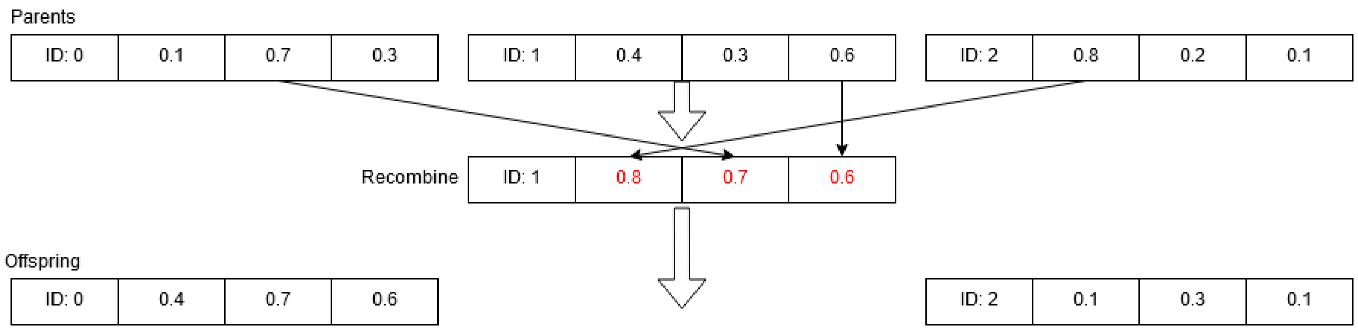

FIGURE 2 Recombine operator working on three example individuals

each population individual. Second, the generated audio is compared to the target audio, and the similarity between them is used to determine the "fitness" of each individual.

Each individual in the evolutionary strategy population has four parameters when used for simple FM synthesis parameter matching. The values of these 4 parameters vary between individuals and are mapped to the 4 parameters of the simple FM synthesis algorithm in Equation 1. A frame of audio is then generated from the FM synthesiser and stored contiguously in memory, ready for further processing to determine similarity with the target audio.

The similarity between the generated and target audio is determined by first mapping the audio signal to the frequency domain using a Fourier transform. The fast Fourier Transform (FFT)[33] is the de facto algorithm for calculating the Fourier transform of a signal efficiently. However, the FFT algorithm requires the input data to be cyclic, spanning from one end and back to the beginning. Therefore, before the FFT algorithm can process an audio signal, it must first undergo a form of pre-processing; this is known as FFT windowing. To reduce the occurrence of artefacts in

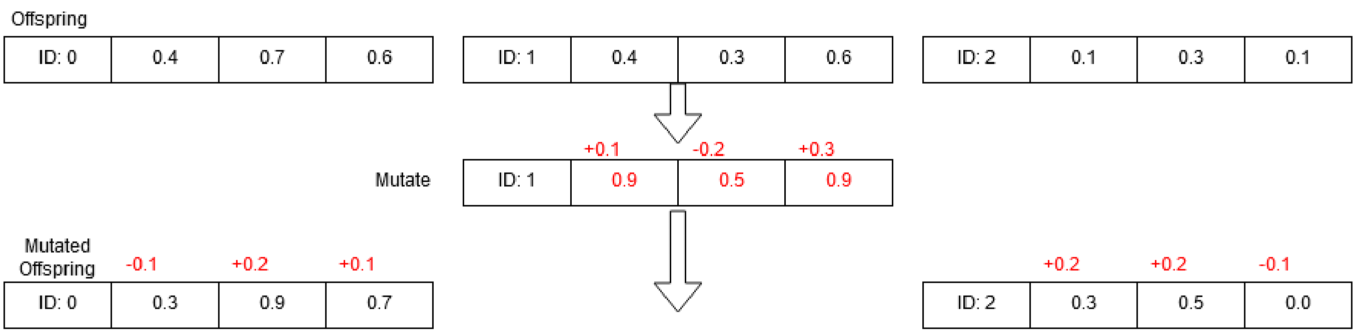

FIGURE 3 Mutation operator working on three example individuals.

the spectrum occurring as a result of frame boundary discontinuities, a windowing function is used. In this paper, a Hann window[34] is applied to each audio frame as follows:

$$\omega(n) = 0.5 \times \left(1 - cos\left(\frac{2\pi n}{N-1}\right)\right) \tag{4}$$

Where ω is the processed Hann value, n is the value of the current sample and N is the total number of samples in the window. Once the time-domain audio signal is prepared using Hann windowing, it can be processed by the FFT algorithm[35] to produce a series of complex numbers representing the same signal in the frequency domain. FFT is a highly optimised and supported algorithm that is available in software libraries such as FFTW[33], which handles the intricacies of the implementation. With each respective individual's audio signal mapped to the frequency domain, the similarity of the signal to the target audio can be calculated. A primary method for comparing the error/difference between two frequency spectra can be achieved using relative spectral error. Studies performed by Beauchamp et al.[36] have shown that the relative spectral error delivers the best correspondence to average discrimination data extracted from human listeners when compared with alternative spectral error metrics. The equation for relative spectral error is defined as:

$$rse = \sqrt{\frac{\sum\limits_{b=0}^{N_{bin}} (T_b - S_b)}{\sum_{b=0}^{N_{bin}} T_b^2}} \tag{5}$$

Where rse is the calculated relative spectral error, T and S are vectors of the target and synthesised audio frequency spectra respectively, and N_{bin} is the number of frequency bins that control the resolution of the analysed spectra. The relative spectral error (RSE) between two audio signal frequency domains T and S can be calculated. The number of frequency bins N_{bin} is the resolution of the frequencies represented in the audio frequency spectrum, this must be the same for both T and S. Using the RSE, the fitness for each individual as a candidate for matching with the target signal can be determined. As the RSE is the error between two frequency spectra, the fitness is the inverse of the RSE. Therefore, an RSE $= 0.0$ is a perfect match, whilst increasing rse identifies differences between the signals. Evolution strategies typically involve a stopping criteria, such as when a sufficiently "fit" individual is found, the evolutionary strategy iterations stop and the individual used as the solution. In the context of this work, the stopping criteria is a set number of iterations/generations to avoid an undefined number of iterations and so the computation time can be fairly measured.

The selection stage involves taking a set number of individuals from the offspring and using them in the next generation's parent population. The approach used in this paper is to sort the individuals by fitness and select the top individuals for the next parent generation. A parallel merge sort is used and has been shown to map efficiently into the GPU architecture[37].

4 | GPGPU DESIGN

The GPGPU literature contains a range of different approaches and associated terminology. In this work, the OpenCL standard language will be used[38 39 40]. The GPU architecture is comprised of streaming multiprocessors that load and execute program instructions in a parallel Single-Instruction Multiple-Data (SIMD) format. The control flow of GPU programs comprises of workitems; these are streams of execution running on

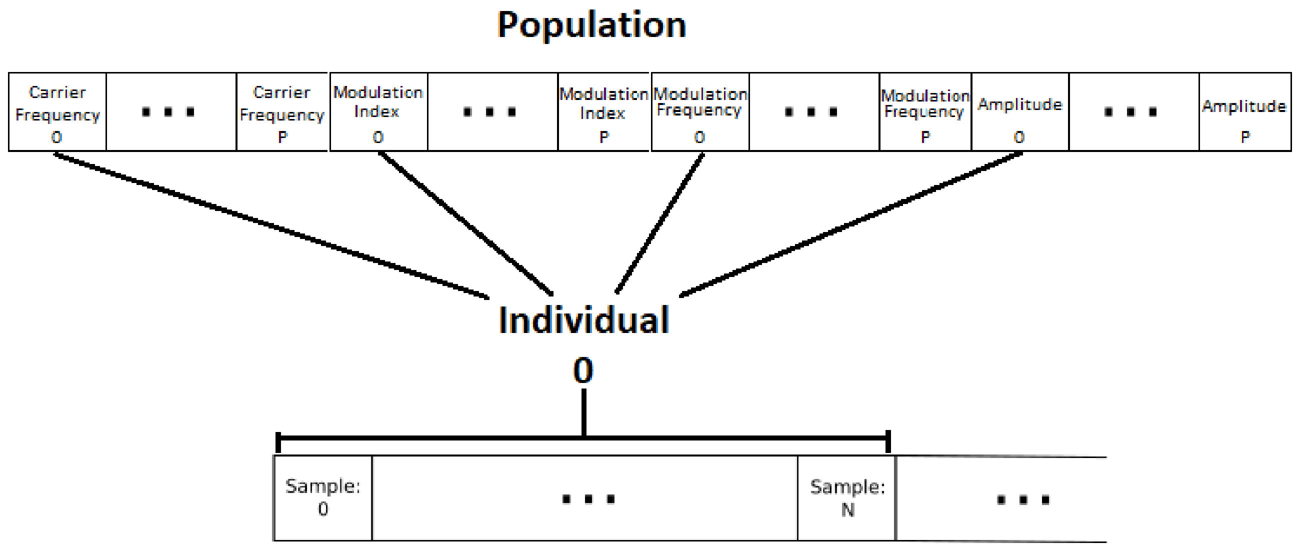

FIGURE 4 The data structure format for the GPU design.

the multiprocessors. The multiprocessors require access to regions of memory arranged in a memory hierarchy similar to a CPU, where memory types vary in size and access speed[41]. The largest region accessible by all workitems is known as global memory. It is typically the slowest region of memory to access. Local memory is a faster region only accessible in a small group of workitems, this group is called a workgroup. Finally, private memory is the smallest but fastest memory, accessible only by the workitem itself. In this section, the techniques and design choices relevant to the GPU implementation are considered.

4.1 | Data Structure

All GPU processed data starts with the ES population. Figure 4, provides a visual aid to help describe the data structure that is used in this GPU design. The first buffer indicated by "Population" contains all the individuals' parameters. Although naturally it seems coherent to place each individual's parameters contiguously in memory, this structure is sub-optimal within the GPU. Therefore, when designing data structures for the GPU, it is advantageous to store all of the first parameters contiguously, then the second parameters, and so forth. This design is commonly referred to as a structure of arrays instead of an array of structures. Various research demonstrates that this is the most efficient GPU data layout for comprehensive processing[42][43]. The parameters of an individual are fetched by indexing each array of parameters using the individual's ID. During the synthesis stage, each individual has an audio block of a set length of samples generated from the parameters. Each audio block is held contiguously in memory, against the generated audio of the next individual's parameters. With this format, the size of the buffers are: $\text{population} = P$, $\text{audioSamplesBlocks} = P * N$ where P is the population size and N is the audio block length.

The population buffer containing each individual's values is stored in one large GPU buffer as the GPU is optimised for fewer, larger memory allocations with offsets to access them than multiple smaller ones. Furthermore, the population buffer is twice the size of the population in order to store a copy of the sorted population when evaluating population fitness. Using a rotation index, an offset to the beginning of the sorted population can be used. This greatly improves performance and memory usage as it avoid copying memory between three separate population buffers.

In the context of the GPU, memory coalescing is achieved when threads on the same streaming multiprocessor access memory simultaneously. To achieve this, consecutive threads should be programmed to access consecutive memory addresses. The GPGPU programming model exposed by OpenCL involves reconsidering the way loops iterate when processing data. The application proposed in this paper aims to optimize memory accesses where possible. This is typically achieved by using the workitem index *idxWorkitem* and loop index *i* in such a way as to access adjacent memory at once between workitems, instead of accessing memory in completely different locations in memory. An example of correctly programming the GPU to achieve memory coalescing can be found in the following FFT Hann windowing kernel:

```
void applyWindowPopulation(float* audioDate, uint32_t audioLength)
{
    int idxWorkitem = get_global_id();
    float mu = ( FFT_ONE_OVER_SIZE - 1) * 2.0f * M_PI;

    int stride_factor = audioLength;
    for(int i = 0; i < POPULATION_COUNT; i++)
    {
        int coalesced_index = stride_factor * i + global_index;
        float fftWindowSample = 1.0 - cos(index  * mu);
        audio_waves[coalesced_index] = fft_window_sample * audio_waves[coalesced_index];
    }
}
```

Here, Equation 4 is implemented on the GPU with an optimized memory access pattern. The index is multiplied with a 'stride' factor equal to the audio wave size. This forces the indices of each workitem to access memory adjacent to one another when processing the windowing. All the indices in the respective workitems then stride to the next audio wave to process when *i* increments. This means that each workitem can share the memory fetched by its neighbouring workitems without requesting a memory fetch somewhere else in memory itself. A more comprehensive description of memory coalescing on GPUs with visual aids can be found in[44].

4.2 | Processing Format

The population size determines the number of workitems the GPU allocates for processing in parallel. These workitems are dispatched to execute each individual's recombination, mutation, audio synthesis, fitness, and selection. Each individual is entirely independent from all other individuals and therefore can execute efficiently without stalling to synchronize data. The FFT processing is handled separately using the clFFT library[45], a well-established and optimized open-source implementation of FFT on the GPU.

During the audio synthesis stage, the chosen FM synthesis arrangement will typically make calls to the trigonometric functions like sin(). These are computationally expensive compared to other basic math operations. Optimized implementations might use Taylor series expansion or the CORDIC algorithm[46]. Instead of calculating these values, a lookup table of previously calculated sine values can be used[47]. As covered in Section 3.2, each individual is tasked with generating an audio block of samples using simple FM synthesis. A pre-calculated wavetable of values approximating sin is uploaded to the GPU and indexed instead of using the computationally expensive sin operator. A finite number of values can be stored in the lookup table, resulting in quantization. Therefore, sufficient resolution must be used in the lookup table. The wavetable optimized synthesis stage (without interpolation for simplicity) is demonstrated in the GPU kernel code snippet below:

```
int idxWorkitem = get_global_id();

float cur_sample = 0.0;
float pos_1 = 0.0;
float pos_2 = 0.0;

const float wavetableIncrementOne = (WAVETABLE_SIZE / 44100.0) * modulationFrequency;
for(int i = 0; i < WAVE_FORM_SIZE; i++)
{
    cur_sample = wavetable[pos_1] * modulationIndex + carrierFrequency;
    out_audio_waves[idxWorkitem * WAVE_FORM_SIZE + i] = wavetable[pos_2] * amplitude;

    pos_1 += wavetableIncrementOne;
    pos_2 += (WAVETABLE_SIZE / 44100.0) * cur_sample;

    if (pos_1 >= WAVETABLE_SIZE)
        pos_1 -= WAVETABLE_SIZE;
    if (pos_1 < 0.0)
        pos_1 += WAVETABLE_SIZE;
    if (pos_2 >= WAVETABLE_SIZE)
        pos_2 -= WAVETABLE_SIZE;
    if (pos_2 < 0.0)
        pos_2 += WAVETABLE_SIZE;
}
```

Here, the section of the code that generates the simple FM synthesis waveform is shown. This implements the equivalent of Equation 1 on the GPU using a wavetable optimisation. In this code, *m* is contained in variable "modulationFrequency", *I* in "modulationIndex", *c* in "carrierFrequency" and *A* in "amplitude". Instead of making calls to $\sin(\dots)$, the wavetable is accessed using pos_1 and pos_2 that are incremented in such a way as to access approximate pre-computed values of $\sin()$. Variable *idxWorkitem* multiplied with WAVE_FORM_SIZE is used to stride through the memory written to in a GPU optimized way.

Batching processes to the GPU is a fundamental technique involved in GPU programming. It is often used to avoid consuming GPU memory resources if an individual task is too big. This can quickly become the case here, as the population size is a controllable parameter, which can exceed GPU memory limits. The amount of GPU memory resources depends on the system's hardware. Taking the systems in Table 1 as examples, the GeForce 2080 has 8GB of memory whilst the Radeon 530 has only 2GB. The intel UHD GPU shares its 8GB of RAM with the CPU. In order to avoid misusing the GPU memory, the GPU design proposed breaks up data into manageable, equally sized blocks, loads them onto the GPU and processes them one at a time. The pseudocode for the batching is given below:

```
void parameterMatchAudio(float* aTargetAudio, uint32_t aTargetAudioLength)
{
    blockSize = objective.audioLength;
    blocks = aTargetAudioLength / blockSize;

    for (uint32_t i = 0; i < blocks; i++)
    {
        setTargetAudio(&aTargetAudio[i*blockSize], blockSize);
        initPopulationCL();
        executeAllGenerations();
    }
}
```

An additional advantage to using the batching technique is that the system is easily extended in the future to involve analysing dynamically changing sounds. This means if the characteristics of a sound changes over time, analysing each block can identify the parameters necessary to match the changing sound in each block. If this advancement was explored, there would be a set of parameters for each audio block analysed.

5 | BENCHMARKING

The benchmarking suite involves the autonomous execution and performance profiling of a collection of planned tests. Configurations of parameters are exhausted and the results collected and written to files for analysis. The benchmarking suite is deployed on different devices to collect profiles on a range of hardware systems. The GPU implementation will synchronize between each OpenCL call to confirm the action is finished to accurately measure the execution time. The benchmarking suite is open-source and publically available online [a]. The implementations here use OpenCL v1.2 in order to support the widest range of hardware possible. Three different implementations were developed from previously defined designs.

CPU Serial - In order to establish the minimal baseline performance, a serial single-core implementation targeting the CPU. Modern CPUs include multiple cores and vector processors which are not used here. The purpose is to demonstrate the impact of restricting a program to a purely serial format.

CPU OpenCL - A parallel implementation targeting the CPU using OpenCL. Following the OpenCL programming model utilizes all parallel processing components of the target hardware. In the case of the CPU, multiple cores and vector processors are utilized. This implementation will be compared to the serial CPU version, but more importantly, the massively parallel GPU version.

GPU OpenCL - Implementation using OpenCL to target parallel processing on the GPU.

[a] https://github.com/Harri-Renney/Survival_of_the_Synthesis-GPU_Accelerated_Frequency_Modulation_Parameter_Matcher

Specification	Mid-range Laptop	High-end NVIDIA GeForce
CPU	Intel Core i7-8550U	Intel Core it-9800X (16 SM)
Integrated GPU	Intel UHD Graphics 620	None
Discrete GPU	AMD Radeon 530	GeForce RTX 2080 Ti (32 SM)
CPU RAM	8GB	32GB

TABLE 1 Hardware specification used for benchmarking. Streaming Multiprocessor (SM)

Parameter	Value	Notation
Number Generations	1000	G
Number Parameters	4	D
Parent Population Size	1024	P
Offspring Population Size	7168	O
Target Audio Length	2048	T
Audio Block Size	2048	N
GPU Workgroup Size	32	W

TABLE 2 Default benchmarking parameters

5.1 | Hardware Systems

The benchmarking suite was run on a collection of different systems containing GPU devices from different hardware vendors, including integrated and discrete GPUs. GPUs are often classed as being one of two types, discrete or integrated GPUs. Discrete GPUs are separate from the CPU and connected across system buses known as PCI. However, communicating over the PCI buses involves data transfer overhead. In contrast, integrated GPUs are tightly coupled to the CPU, even sharing memory spaces. This means that the latency overhead is avoided and communication between CPU and GPU is faster than the discrete GPU[48]. However, integrated GPUs have constraints imposed by limited physical space and therefore typically have fewer and slower multistreaming processors. The hardware systems considered in this paper are detailed in Table 1. In addition, results have been collected for another system titled "High-End NVIDIA Titan". This system's results have not been included in the paper but are similar to the "High-End NVIDIA Geforce" system. The results for this system can be found in the full database of results found at the link provided in Section 6.

5.2 | Benchmarking Targets

The benchmarking suite is designed to measure the overall execution for default parameters and used to measure execution time when scaling controllable parameters. The controllable parameters affect the performance and accuracy of the algorithm and therefore play a crucial role. Unless stated, the default parameters will have the values shown in Table 2. The default parameters have been chosen as they provide solutions with acceptable fitness and give the benchmarking sufficient processing to profile and consider. Instead of using a fitness threshold as the stopping criteria, the benchmarking suite will execute a fixed number of generations. This ensures parity between results by guaranteeing that all implementations processed an equal amount of computation, mitigating any stochastic variations. The focus of this benchmarking suite is to compare the performance differences between implementations for the basic ES and FM synthesiser sound matching application. It is not to evaluate the accuracy of the solutions generated by the ES, the literature covered in Section 2 has already established ES as an effective method for accurately finding solutions. However, the reader can be reassured that the default parameters' fitness converge to an error of 0.0 and, therefore, an exact match is found using the default parameters. The benchmarking suite covers performance profiling the following aspects listed:

Implementation	Total Execution Time (s)
CPU Serial	409.98
CPU OpenCL	28.33
GPU OpenCL	3.19

TABLE 3 Total execution time of the application for the CPU and GPU implementations.
Parameters = Default **System** = High-end NVIDIA GeForce

Overall Execution - The overall execution time for the program to complete is the primary concern. The overall execution time includes all stages in Section 3, including GPU specific initialization of: The starting population, wavetable (Section 4.2), random number lookup table.

Program Stage Execution - In order to highlight the computational implications at each stage of the processing, the stages in Section 3 are independently timed. This is useful to expose potential bottlenecks and the most intensive stages.

Audio Analysis Block Size - The size of each block of audio analysed at a time by the application. The block size controls the amount of data transfers between CPU and GPU. Therefore, the impact of scaling the audio block size will be evaluated in these results.

Population scaling - It has been shown that increasing the population size in evolutionary algorithms can produce fitter solutions in a more complicated problem space. Measuring how the performance and accuracy are affected when changing population size is an important aspect.

optimisations On/Off - In order to demonstrate any performance benefits from using the GPU optimisation in Section 4, implementations employing the optimisations will be profiled against implementations that do not use them.

Discrete vs Integrated - The benchmarks are executed on an integrated and discrete GPU in the same system. This maintains a common environment excluding the target GPU, provides insight into the performance of integrated and discrete GPUs for fair comparison.

6 | RESULTS

In this section, the results collected across the benchmarking suite are presented, along with a discussion of the results. This paper considers the salient results collected for the two systems outlined in Table 1. The full collection of results including another High-End NIVIDA Titan setup have been made available online [b]. Particular focus on the results are given to the High-End NVIDIA GeForce desktop that better reflects the expected audience of synthesis parameter matching. The laptop is used to highlight integrated vs discrete GPUs and the difference between a high-end desktop and mid-range laptop. Readers can use the open-source benchmarking suite provided in Section 5 to collect results for additional systems.

6.1 | CPU vs GPU

Comparing the execution speed between the CPU and GPU designs is of primary interest. The results shown in Table 3 include the three implementation's total time with the default parameters for the High-End NVIDIA system. The CPU serial version takes considerably longer at 409.98s, whilst the OpenCL CPU and GPU are 28.33s and 3.19s. The CPU has parallel vector processors available and it can be seen that a clear improvement of 14X speedup is achieved when using them by the CPU OpenCL version. GPUs advance this data-parallel processing further, achieving a speedup of 128X over the serial CPU version. This demonstrates that not only are ES and FM synthesis suitable for data-parallel processing, but that they are also suitable in combination using the design proposed in this paper. Comparing the CPU OpenCL version to the GPU OpenCL version, it can be seen that the GPU has a speedup of 8.88X over the CPU. This is expected as the GPU architecture is designed to maximize the data-parallel processing whilst the CPU is not.

6.2 | Implementation Stages Compared

The execution time for each stage in the application on the High-end NVIDIA system has been recorded and displayed in Figure 5. The execution time is marked in ms on a logarithmic scale using the default parameters. When comparing each implementation, the stages follow a similar pattern of the CPU serial taking the most time, followed by CPU OpenCL and finally GPU OpenCL. There is a particularly high improvement on the GPU for the "synthesis", "Window" and "FFT" stages. All these stages are included in the design and highlight the importance of this novel approach

[b]https://muses-dmi.github.io/benchmarking/benchmarking_database_survival_of_the_synthesis

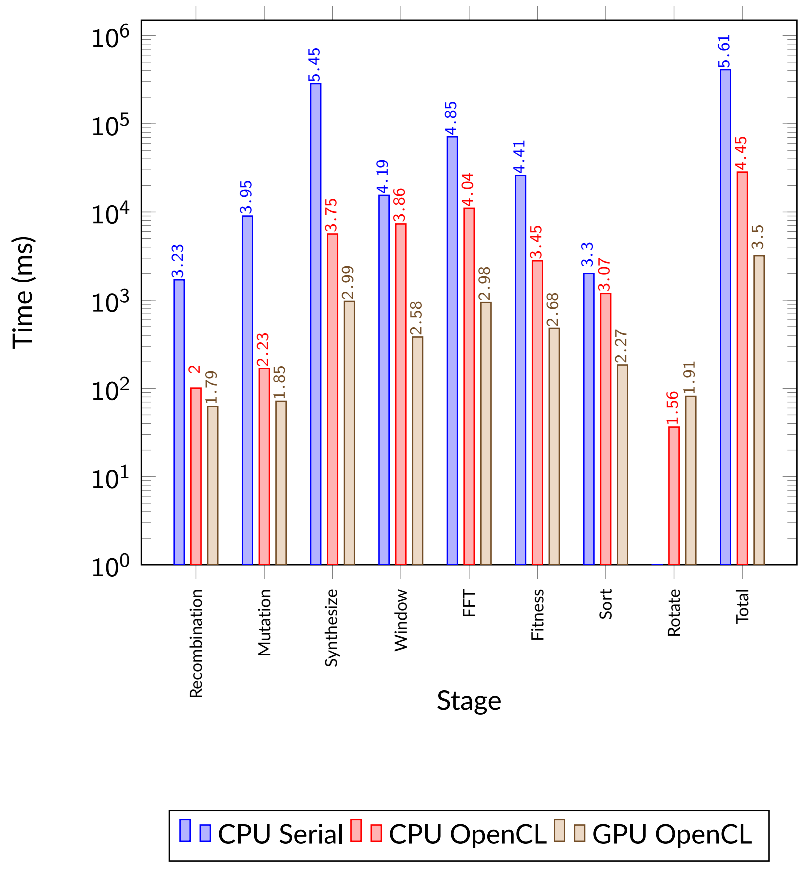

FIGURE 5 Execution time of each stage across the CPU and GPU implementations. **Parameters** = Default **System** = High-end NVIDIA GeForce

considering the domain-specific stages and the evolutionary algorithm. The data-parallel version improves over the CPU versions in all stages, except for the "Rotate" stage, where the CPU serial had a comparatively negligible 0.3544ms. This is a negligible amount as it only involves updating a variable in CPU memory. In contrast to the OpenCL versions that require an OpenCL function with added overhead. For example, the GPU version requires updating the rotation index in GPU memory, this involves a data transfer over the PCI interface to update the variable in GPU memory. This is a stage the CPU will naturally surpass the OpenCL versions. However, it is a small difference to the vast improvements observed in the rest of the stages.

6.3 | Hardware Systems Compared

The High-End NVIDIA desktop is expected to execute faster for all population sizes as it utilizes the more powerful NVIDIA GeForce RTX 2080 Ti, whilst the mid-range Laptop contains a less capable AMD Radeon 530. It can be seen in Figure 6 that the NVIDIA 2080 GPU shows little increase in computation time when the population is scaled up to 32768. Whilst the AMD 530 GPU has a roughly directly proportional increase in execution time with population size, the NVIDIA 2080 architecture contains more multistreaming-processors and therefore can process more individuals in parallel. This demonstrates, as is the case with most processes in computing, that the speed and throughput of processing units is still an important factor for data-parallel processors.

6.4 | Kernel Execution Time Ratio

Figure 7 shows the relative execution times for every stage of the GPU OpenCL implementation running with default parameters. This highlights where the majority of the processing takes place. The stages processing the ES population: "Recombine", "Mutate" and "Sort" together only account for less than 13% of the time. These stages execute quickly as they only process the population $(P + O) * D$, which is $(1024 + 7168) * 4 = 32,768$

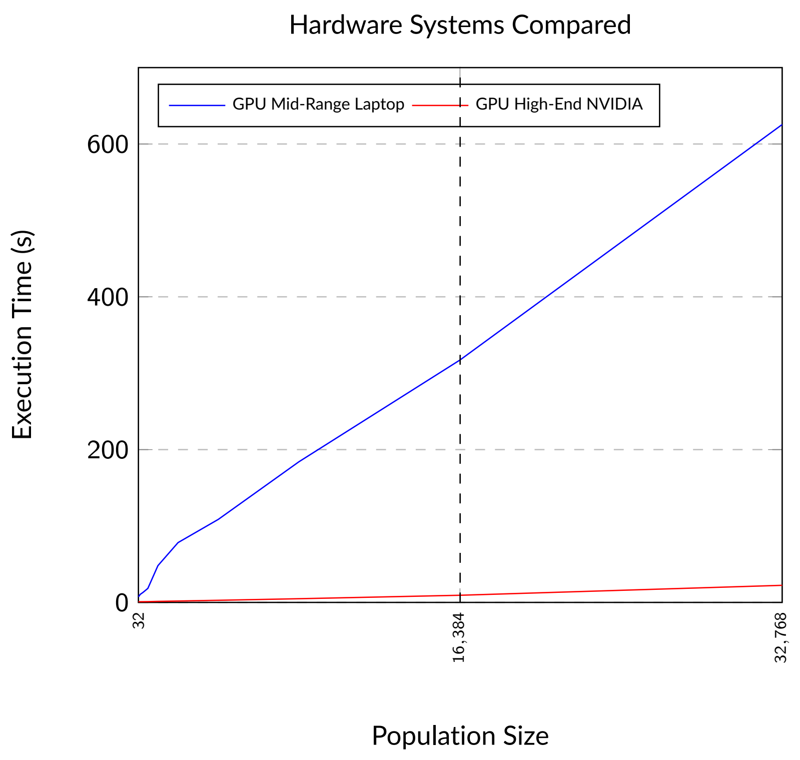

FIGURE 6 Execution time when scaling the population size for the High-End and Mid-range systems. **Parameters**: G = 1000, P+O = Scaled **System** = High-end NVIDIA GeForce & Mid-range Laptop

floating point values for the default parameters. The stages processing audio for the population: "Synthesis", "Window", "CLFFT" and "Fitness" take considerably longer accumulating 87.43% of the time. The audio stages require processing $(P+O)*N$ which is $(1024+7168)*2048 = 16,777,216$ floating point values. The majority of the execution time is taken up by the synthesis and CLFFT stages, taking around 30% of the time each. This highlights the importance of the design this paper describes as it not only efficiently processes the evolutionary algorithm stages, but incorporates the domain-specific processes that take up the majority of the execution time.

6.5 | Integrated vs Discrete

Figure 8 shows the results comparing the integrated Intel and AMD discrete GPU in the mid-range laptop. It can be seen that initially, when the population size is small, the integrated GPU performs better than the discrete GPU. This is expected as using the PCI-e system bus to communicate between the discrete GPU and the CPU takes considerably longer than sharing a memory space. The communication overhead surpasses any benefits of using the discrete GPU for smaller population sizes, whilst the integrated GPU avoids the communication overhead using unified memory. However, at around population size 2048, the performance of the discrete GPU exceeds the integrated GPU. The more powerful discrete GPU scales more efficiently for larger population sizes and the benefit exceeds the communication overhead involved.

6.6 | Optimized vs non-optimized

The results when comparing the use of the FM synthesis lookup table optimisation described in Section 4.2 are compared against the use of the original trigonometric functions in Figure 9. This optimisation has a huge effect on the performance of the synthesis stage, for simple FM synthesis, this involves replacing two calls to the sin() function per sample with direct accesses into a wavetable. Back in Figure 7 it can be seen that the synthesis stage was one of the most consuming stages, therefore, this optimisation offers a 4X speedup in the synthesis stage for the default parameters. This significant improvement results in an overall 2X speedup for an optimisation affecting just the synthesis stage. This demonstrates the potential improvements context specific optimisations can make for intensive processing involved in the fitness stage.

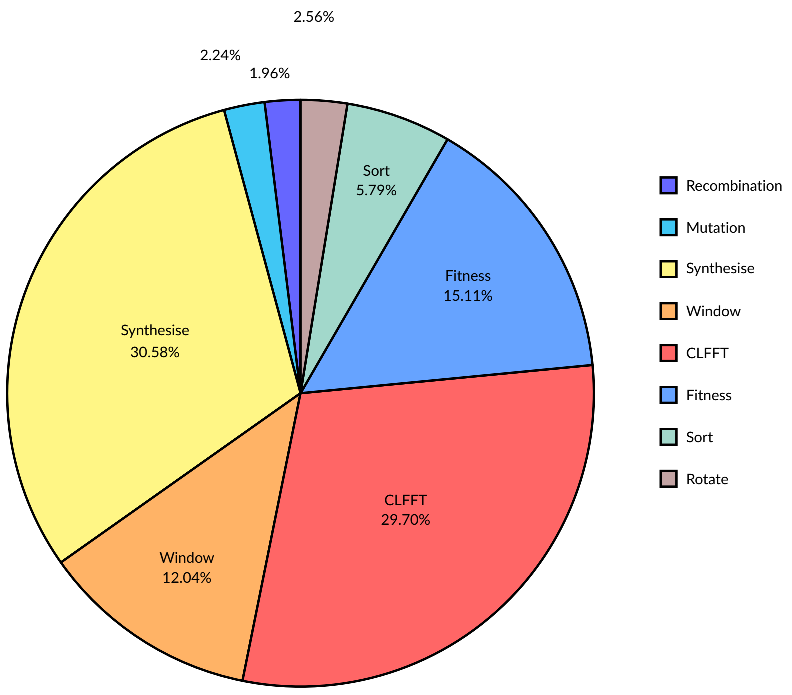

FIGURE 7 Ratio of execution time across stages in the GPU OpenCL implementation.
Parameters = Default **System** = High-end NVIDIA GeForce

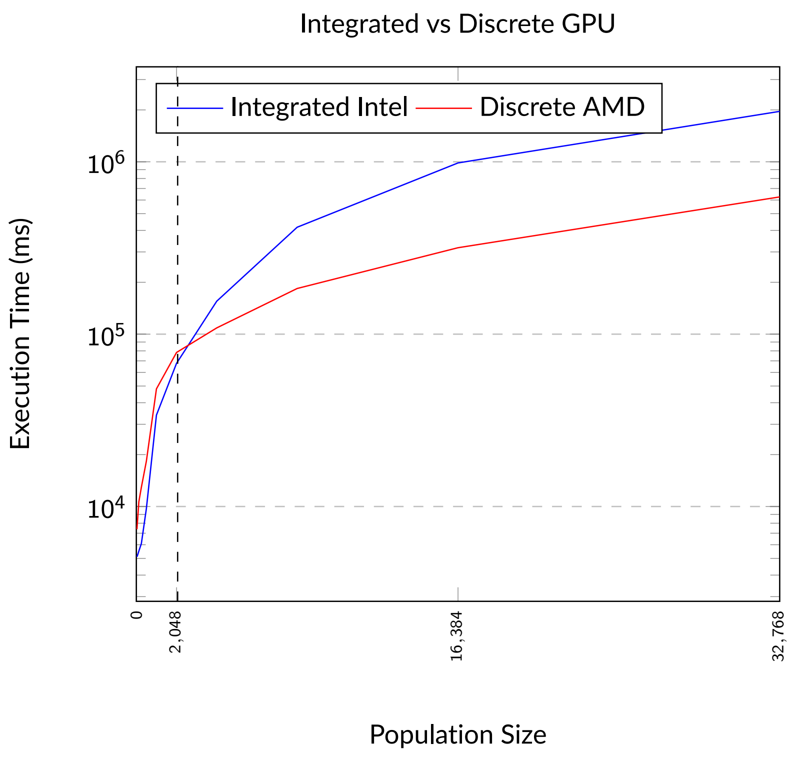

FIGURE 8 Execution time when scaling the population size for the GPU OpenCL implementation targeting the integrated and discrete GPU.
Parameters: Default, G = 1000, P+O = Scaled **System** = Mid-range Laptop

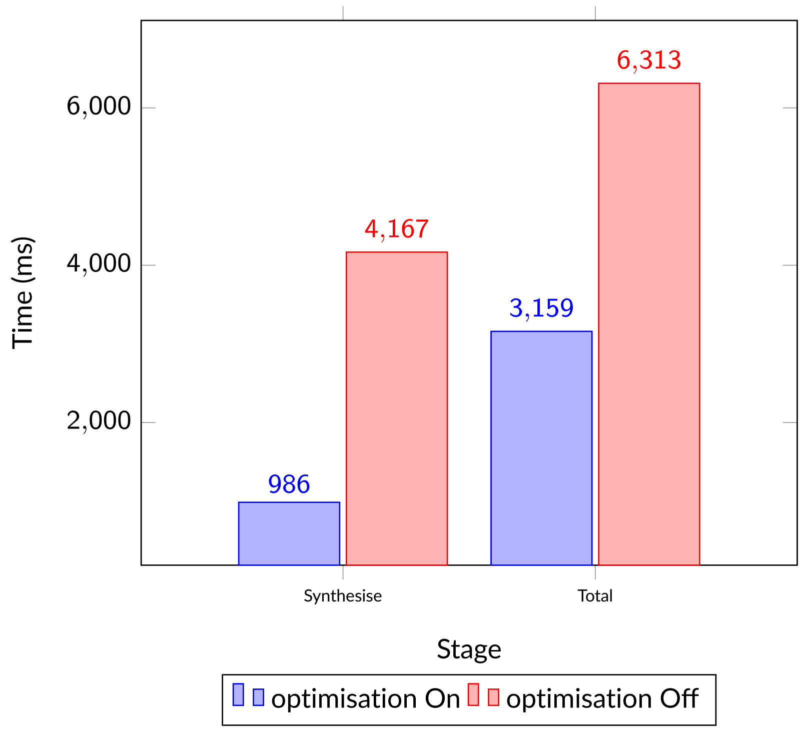

FIGURE 9 The total and synthesis stage execution time when the FM wavetable lookup optimisation is on and off. **Parameters** = Default **System** = High-end NVIDIA GeForce

6.7 | Population Scaling

Figure 10, provides a comparison between the implementations on the High-End NVIDIA system when scaling the population size. The execution time is plotted on a logarithmic scale in seconds. The CPU Serial version initially starts at a higher execution time at 1.87s compared to the GPU OpenCL version taking 0.89s. It is clear that as the population size scales, the execution time for the CPU versions is significantly higher. The GPU version's execution time increases gradually as its ability to process far more individuals in parallel is realised. At population size 16,384, the GPU version runs at approximately 10s, while the CPU OpenCL version records 55s, roughly 5X slower and the CPU Serial at 870s, 87X slower. The GPU continues to show it provides a speedup even at higher population sizes, beyond the 8192 size used in the default configuration. The GPU performance achieved brings the application considerably closer to a practical, real-time tool. Complex synthesisers that require considerably large population sizes to find accurate solutions will benefit further by using GPUs over CPUs.

6.8 | Audio Analysis Block Size Scaling

Figure 11 shows the total time of the GPU OpenCL application when scaling audio block size on the NVIDIA 2080 discrete GPU. The audio blocks size determines the number of audio samples processed and analysed on the GPU at a time. Decreasing the audio block size splits the target audio into further separate blocks for analysis. There are two reasons a smaller block size might be considered. First, the GPU memory would not be able to accommodate the larger blocks of audio. Second, this approach is advantageous if the audio being analysed is dynamic. However, decreasing the audio block size increases the number of dispatches to the GPU, which increases the communication overhead over the PCI bus. The overhead adds up considerably, resulting in severely reduced performance when the audio block size is below 128. Beyond block size of 128, the performance reaches a significantly improved state. Continuing to increase the audio block size to the target audio size has a less significant performance growth. These results show that although a smaller block size can be used, a size below 128 begins to have a significant impact on performance. Therefore, dynamic audio samples that involve frequent changes in timbre will be challenging to process with the current design and will need to be improved to support it.

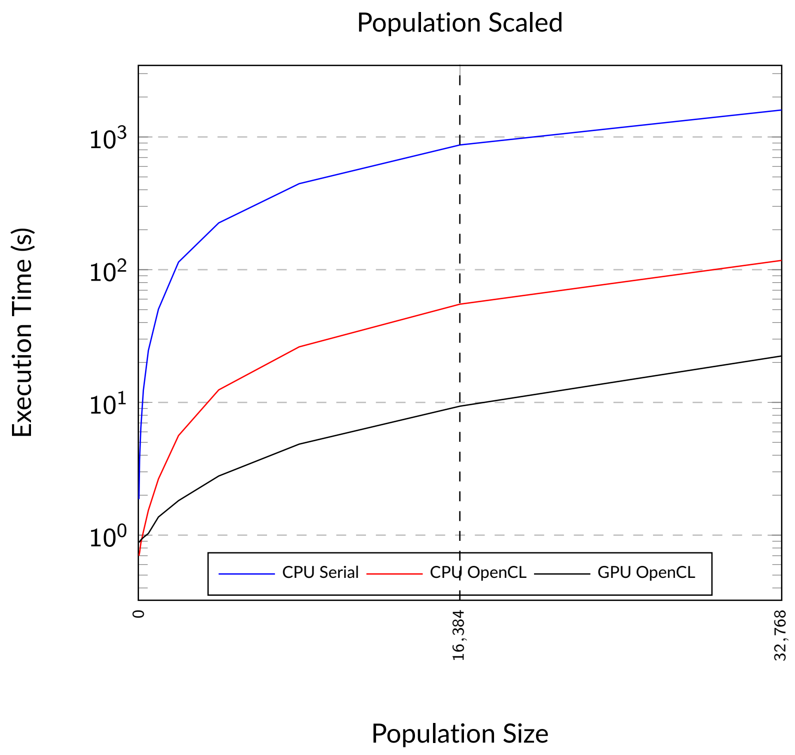

FIGURE 10 Execution time when scaling the population size for all implementations.
Parameters: G = 1000, P+O = Scaled **System** = High-end NVIDIA GeForce

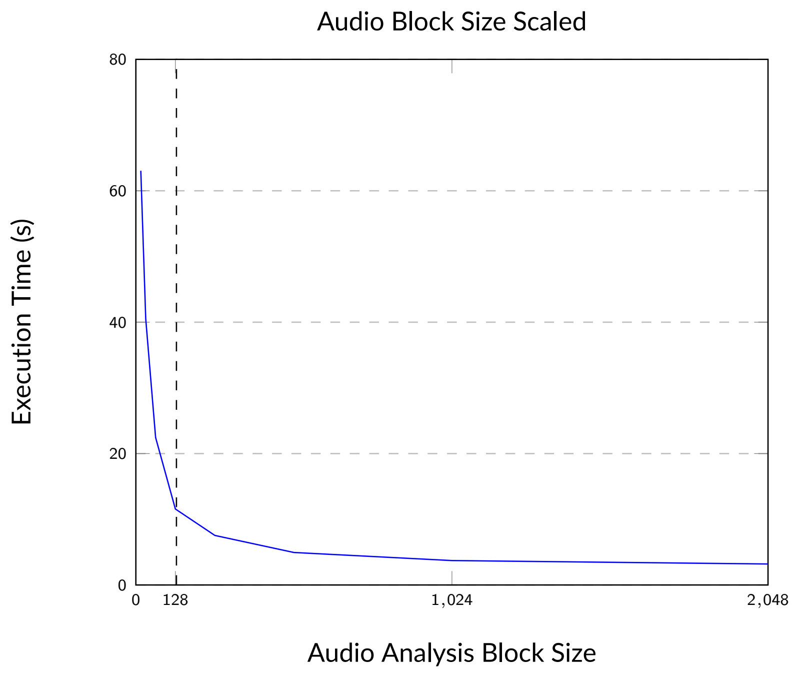

FIGURE 11 Execution time when scaling audio block size for the GPU OpenCL implementation.
Parameters: Default, N = Scaled **System** = High-end NVIDIA GeForce

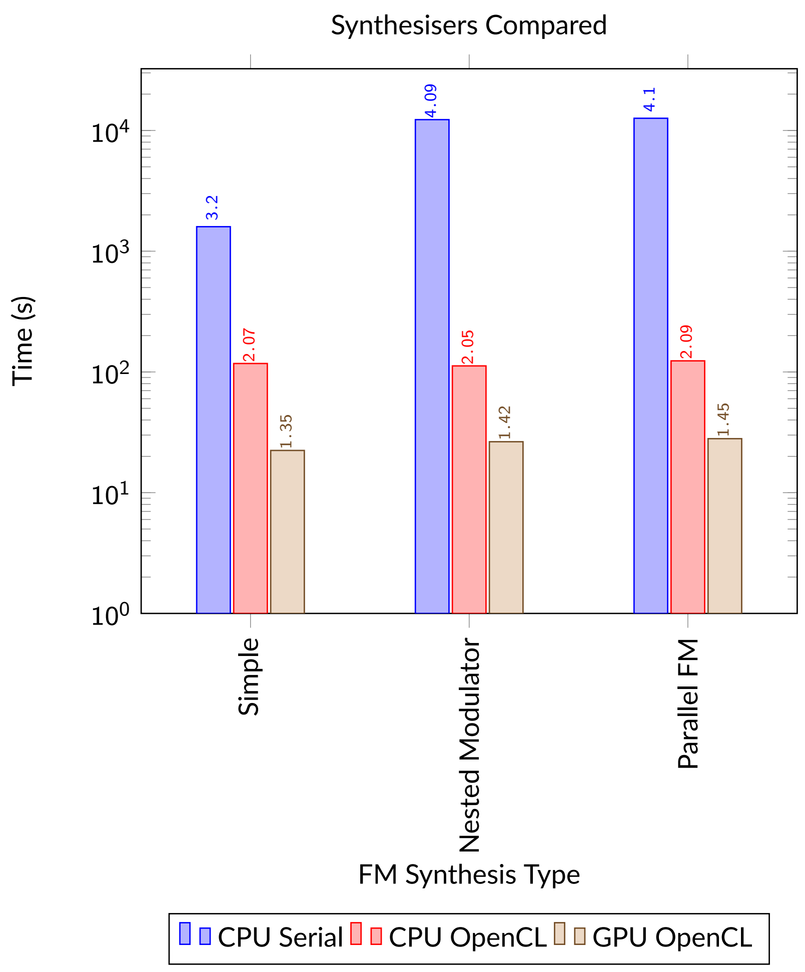

FIGURE 12 Execution time of three synthesis types for the CPU and GPU implementations.
Parameters: Default, P+O=32768 **System** = High-end NVIDIA GeForce

6.9 | Advanced FM Synthesisers

To demonstrate the scalability of the GPU optimised framework, two further FM synthesisers have been used in place of the simple FM. The first is nested modulator FM and the second is parallel FM synthesis. Both of these multi-operator FM synthesisers have been described in Section 2.1. Figure 12 presents the results of the simple, nested modulator and parallel FM synthesisers on all three implementations. Again, a large population size of 32768 has been used and the time in seconds is again plotted on a logarithmic scale. It can be seen for the CPU Serial implementation, both the nested modulator and parallel synthesisers take considerably longer with an additional 10699s and 11024s respectively. By contrast, the GPU OpenCL implementation only requires an additional $\approx$ 4s for the nested modulator and $\approx$ 6s for parallel FM. The results suggest that the GPU accelerated framework for handling the evolutionary computation for parameter matching supports more advanced forms of FM synthesis. Although this cannot be extrapolated to all possible arrangements of FM synthesis, for these two examples, it continues to improve performance over a naive serial implementation.

7 | CONCLUSION

This paper has presented the design for a GPU optimised algorithm for parameter matching for FM synthesis. The designs have been implemented as serial CPU, parallel CPU and parallel GPU versions in a benchmarking suite. The benchmarking suite is open-source and can be used to evaluate the performance of the implementations on different hardware systems. Benchmarking results were collected on various systems and the salient results were discussed. For the default parameter configuration on a high-end desktop, the GPU had a speedup of 128X over the serial CPU version and 8.88X over the parallel CPU version. This highlights the significant improvement potential when using parallel processing in general, but also the massively parallel architecture of the GPU in comparison to the CPU. The population size of the ES has a significant impact on the execution time of all implementations. However, the impact of the GPU version was significantly lower than the other implementations, suggesting that the GPU design proposed can be used to process larger population sizes more rapidly, increasing the usability of parameter matching as a suitable tool for music creators. The results show that the performance benefits apply to simple FM synthesis and extend to support more advanced arrangements such as nested modulator and parallel FM synthesisers. Another controllable GPU parameter is the data block size, this was scaled and was shown to harm the GPU processing time when block sizes below 128 are used. This is a weakness of the GPU design caused by the data transfer overhead between CPU and GPU. An optimised design needs to be adapted to better support this use-case.

ACKNOWLEDGMENTS

We would like to acknowledge the consistent support and insight from colleagues in the Computer Science Research Centre and the Creative Technologies Lab. This work was supported by a grant from the HSA Foundation and the UWE 50-50 PhD fund. Extended thanks to Jonathan Andrew and Professor Simon McIntosh-Smith at the University of Bristol for the guidance their research provided. Special thanks to Dr David Creasey for his discussions on FM synthesis.

References

1. Mitchell T. Automated evolutionary synthesis matching. *Soft Computing* 2012; 16(12): 2057–2070.
2. Owens JD, Luebke D, Govindaraju N, et al. A survey of general-purpose computation on graphics hardware. In: . 26. Wiley Online Library. ; 2007: 80–113.
3. Luebke D, Harris M, Govindaraju N, et al. GPGPU: general-purpose computation on graphics hardware. In: ACM. ; 2006: 208.
4. Rechenberg I. Cybernetic solution path of an experimental problem. *Royal Aircraft Establishment Library Translation 1122* 1965.
5. Mitchell TJ, Creasey DP. Evolutionary sound matching: A test methodology and comparative study. In: IEEE. ; 2007: 229–234.
6. Horner A. Nested modulator and feedback FM matching of instrument tones. *Speech and Audio Processing, IEEE Transactions on* 1998; 6: 398 - 409. doi: 10.1109/89.701371
7. Yee-King M, Roth M. A comparison of parametric optimisation techniques for musical instrument tone matching. In: Goldsmiths, University of London. ; 2011.
8. Das S, Suganthan PN. Problem definitions and evaluation criteria for CEC 2011 competition on testing evolutionary algorithms on real world optimization problems. *Jadavpur University, Nanyang Technological University, Kolkata* 2010: 341–359.
9. Horner A, Beauchamp J, Haken L. Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis. *Computer Music Journal* 1993; 17(4): 17–29.
10. Mitchell TJ. *An exploration of evolutionary computation applied to frequency modulation audio synthesis parameter optimisation*. PhD thesis. University of the West of England, 2010.
11. Lee KY, Yang FF. Optimal reactive power planning using evolutionary algorithms: a comparative study for evolutionary programming, evolutionary strategy, genetic algorithm, and linear programming. *IEEE Transactions on Power Systems* 1998; 13(1): 101-108. doi: 10.1109/59.651620
12. Macret M, Pasquier P, Smyth T. Automatic calibration of modified fm synthesis to harmonic sounds using genetic algorithms. 2012.
13. Lai Y, Jeng SK, Liu DT, Liu YC. Automated optimization of parameters for FM sound synthesis with genetic algorithms. In: Citeseer. ; 2006: 205.
14. Smith BD. Play it Again: Evolved Audio Effects and Synthesizer Programming. In: Springer. ; 2017: 275–288.
15. Yee-King MJ, Fedden L, d'Inverno M. Automatic programming of VST sound synthesizers using deep networks and other techniques. *IEEE Transactions on Emerging Topics in Computational Intelligence* 2018; 2(2): 150–159.

16. Lengyel J, Reichert M, Donald BR, Greenberg DP. Real-time robot motion planning using rasterizing computer graphics hardware. *ACM SIGGRAPH Computer Graphics* 1990; 24(4): 327–335.

17. Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro* 2008; 28(2): 39–55.

18. McClanahan C. History and evolution of gpu architecture. *A Survey Paper* 2010; 9.

19. Rapaport DC. GPU molecular dynamics: Algorithms and performance.. *arXiv: Computational Physics* 2020.

20. Renney H, Gaster BR, Mitchell T. OpenCL vs: Accelerated finite-difference digital synthesis. In: ; 2019: 1–11.

21. Desell T, Waters A, Magdon-Ismail M, et al. Accelerating the MilkyWay@Home Volunteer Computing Project with GPUs. In: Wyrzykowski R, Dongarra J, Karczewski K, Wasniewski J., eds. *Parallel Processing and Applied Mathematics*Springer Berlin Heidelberg; 2010; Berlin, Heidelberg: 276–288.

22. Turian J, Shier J, Tzanetakis G, McNally K, Henry M. One Billion Audio Sounds from GPU-enabled Modular Synthesis. *arXiv preprint arXiv:2104.12922* 2021.

23. Chowning J, Bristow D. FM theory and applications. *By Musicians for Musicians. Tokyo: Yamaha Music Foundation* 1986.

24. Horner A. A comparison of wavetable and FM parameter spaces. *Computer Music Journal* 1997; 21(4): 55–85.

25. Arabas J, Michalewicz Z, Mulawka J. GAVaPS-a genetic algorithm with varying population size. In: IEEE. ; 1994: 73–78.

26. Chowning JM. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the audio engineering society* 1973; 21(7): 526–534.

27. Roth M, Yee-King M. A comparison of parametric optimization techniques for musical instrument tone matching. In: Audio Engineering Society. ; 2011.

28. Fukuda Y. *Yamaha DX7 digital synthesizer*. Music Sales Corp . 1985.

29. Albano J. An Overview Of Logic Pro X's Powerful Synths. https://macprovideo.com/article/audio-software/an-overview-of-logic-pro-x-s-powerful-synths; 2016. Accessed: 2020-05-03.

30. Horner A. Nested modulator and feedback FM matching of instrument tones. *IEEE Transactions on Speech and Audio Processing* 1998; 6(4): 398–409.

31. Beyer HG, Schwefel HP. Evolution strategies–A comprehensive introduction. *Natural computing* 2002; 1(1): 3–52.

32. Patel JK, Read CB. *Handbook of the normal distribution*. 150. CRC Press . 1996.

33. Frigo M, Johnson SG. FFTW: An adaptive software architecture for the FFT. In: . 3. IEEE. ; 1998: 1381–1384.

34. Harris FJ. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE* 1978; 66(1): 51–83.

35. Schloss J. cooley_tukey. 2019. Accessed: 2019-10-10.

36. Beauchamp JW, Horner A. Error metrics for predicting discrimination of original and spectrally altered musical instrument sounds. *The Journal of the Acoustical Society of America* 2003; 114(4): 2325–2325.

37. Davidson A, Tarjan D, Garland M, Owens JD. *Efficient parallel merge sort for fixed and variable length keys*. IEEE . 2012.

38. Group KOW, others . The OpenCL Spec V2. 0. 2013.

39. Gaster B, Howes L, Kaeli DR, Mistry P, Schaa D. *Heterogeneous computing with openCL: revised openCL 1*. Newnes . 2012.

40. Corporation N. NVIDIA Fermi Compute Architecture Whitepaper. 2009.

41. Hestness J, Keckler SW, Wood DA. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In: IEEE. ; 2014: 150–160.

42. Mei G, Tian H. Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. *SpringerPlus* 2016; 5(1): 1–18.

43. Micikevicius P. GPU performance analysis and optimization. In: . 3. ; 2012.

44. Kim DH. Evaluation of the performance of GPU global memory coalescing. *Evaluation* 2017; 4(4).

45. Natarajan B. clfft. https://clmathlibraries.github.io/clFFT; 2013. Accessed: 2019-10-21.

46. Bertrand M. The CORDIC method for faster sin and cos calculations. *C Users Journal* 1992; 10(11).

47. Raghunath KJ, Rambaud MM. Sine/cosine lookup table. 1999. US Patent 5,937,438.

48. Renney H, Gaster BR, Mitchell TJ. There and Back Again: The Practicality of GPU Accelerated Digital Audio. 2020.

How to cite this article: Renney H,, Gaster BR,, Mitchell T. (2016), Survival of the Synthesis - GPU Accelerating Evolutionary Sound Matching, *Concurrency Computat Pract Exper*, *2021;33:N/A*.