STREAMING SPECTRAL PROCESSING WITH CONSUMER-LEVEL GRAPHICS PROCESSING UNITS

Victor Lazzarini, John ffitch, Joe Timoney

Depts. of Music and Comp. Sci.,
National University of Ireland
Maynooth, Ireland
vlazzarini@nuim.ie,
jpff@codemist.co.uk,
jtimoney@cs.nuim.ie

Russell Bradford

Dept. of Comp. Sci.,
University of Bath
England
rjb@cs.bath.ac.uk

ABSTRACT

This paper describes the implementation of a streaming spectral processing system for realtime audio in a consumer-level on-board GPU (Graphics Processing Unit) attached to an off-the-shelf laptop computer. It explores the implementation of four processes: standard phase vocoder analysis and synthesis, additive synthesis and the sliding phase vocoder. These were developed under the CUDA development environment as plugins for the Csound 6 audio programming language. Following a detailed exposition of the GPU code, results of performance tests are discussed for each algorithm. They demonstrate that such a system is capable of realtime audio, even under the restrictions imposed by a limited GPU capability.

1. INTRODUCTION

Graphics Processing Units (GPUs) have been used for audio processing in a variety of environments. Typically, they have been employed as co-processors in systems where their massively parallel architectures can be harnessed for signal processing programs[1]. There is now a significant number of reports in the literature demonstrating their use in the implementation of various algorithms, such as Ray Tracing [2], Wave-based Modelling [3], SMS[4], Finite Difference Physical Models[5], to cite but a few.

In this paper we investigate the use of off-the-shelf consumer-level GPUs for the implementation of frequency-domain audio processing. In such a scenario, we do not have a separate dedicated co-processor, but rely solely on the on-board GPU that is also driving the video graphics subsystem. Our goal was to study and implement efficient algorithms that could overcome the limitations of the given hardware and possibly deliver realtime performance. In particular, we are interested in developing applications that can be employed by users without the need for specialised hardware setups. Finally, we also envisage that such implementations can, in a second stage, be applied to dedicated co-processor systems in high-performance computing applications.

The processes implemented in this paper involve separate Phase Vocoder (PV) analysis and synthesis, Additive synthesis from PV data, and a Sliding PV (SPV) algorithm-based frequency domain effect[6].

1.1. Environment and toolset

The chosen environment involved a NVIDIA GT650M GPU, with 1024MB VRAM (see Table 1), running on OSX10.9. The chosen parallel development toolset was CUDA 5.5[7], running in conjuction with the LLVM/Clang C/C++ compiler, with Csound 6.02 as the host for the processing plugins. The choice of environment was dictated by two concerns: a good match for the target hardware, which CUDA is, and on the hosting side, a well-developed environment for testing of audio programs, which is provided by Csound[8] version 6[9].

Table 1: *Some specifications for the target GPU*

cores	384
clock speed	900 MHz
VRAM	1024MB
compute capability	3.0
max threads/block	1024
bandwidth	80 GB/s
multiprocessors	2
cores per multiprocessor	192

2. PROGRAMMING MODEL

The programming model supported by CUDA abstracts the GPU processors into a hierarchy of threads, blocks, and grids. At the lowest level, we have separate threads that can be grouped into blocks. A grid is a collection of blocks.

Each thread in a block can be given a one, two or three dimensional index, to facilitate computation across vectors, matrices or volumes. All threads in a block live on the same multiprocessor, execute in parallel and can share fast memory. Blocks can be scheduled in parallel in separate multiprocessors. There is an upper limit in the number of threads in a block, which depends on the compute capability of the hardware used, and in the case of the GT650M is 1024, as shown in Table 1.

Each thread has a local memory space, and can access shared memory within its block. All threads have also access to global device memory. The fast shared memory is very limited in size,

but has a higher bandwidth and lower latency than global memory. Any memory transfers between host (CPU) and device (GPU) are costly and should be minimised.

Thread execution is grouped in *warps*, which contains 32 threads. For full efficiency, it is advised that all threads in a warp have a single execution path. Divergence via conditional branches will force each branch to be executed serially until these converge back into the same path. For this reason, it is important to minimise divergent conditionals in the GPU code.

The code executed by a thread is provided in a unit called a *kernel*. For all practical purposes, this is a C/C++ function (defined by the CUDA attribute `__global__`) that is designed to run in multiple copies, concurrently. The CUDA programming extensions provide a simple syntax to launch a grid of threads based on a given kernel, with a certain number of blocks and threads per block.

CUDA programming assumes a heterogenous programming model, where a host is responsible for allocating and managing device memory, including data transfers, as well as scheduling the parallel execution on the device. Under this model, serial sections of code, running on the host, are interspersed with parallel ones running in the GPU. This is illustrated by Figure 1.

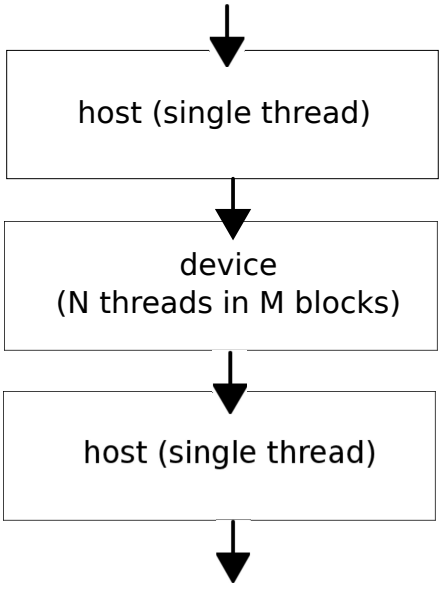

Figure 1: *Heterogenous programming model*

3. STREAMING SPECTRAL PROCESSING

Spectral processing is said to be *streaming* when time domain data is being windowed and transformed in a continuous fashion from an input signal (such as a realtime stream from an analogue-to-digital converter), producing a frequency domain signal of ordered frames at a given rate [10]. This is opposed to the case where all the input data is available for processing at once, and is more restrictive in terms of designing parallel implementations.

Typically, windows will be placed at a constant hopsize, and new output data is produced at a decimated rate, but there are also algorithms for sample-by-sample output, such as the sliding DFT[11], which is used in one of the cases studied. Thus, in the most common cases, we would only need to process spectral data at a reduced rate. This allows us to design a program that will use the GPU as a co-processor to compute the spectral data. The granularity of such process is then set to hopsize samples. This is the basic layout of the code discussed in the following sections.

3.1. Integration with Csound

The code discussed in this paper is hosted in Csound 6 as plugin opcodes (unit generators). Processing is done in vectors of *ksmps* samples, which can be set to any value above 1, with the upper value determined by the analysis hopsize in the case of the standard PV algorithms (no such limit applies to SPV). The PV analysis, synthesis and additive synthesis opcodes work with frequency domain signals (defined by the `fsig` Csound type), as well as the usual time-domain audio (and control signals). Thus, GPU processing is invoked every hopsize samples, in the case of these algorithms. The SPV implementation works solely with audio signals (as it packages analysis, transformation and synthesis in one single unit generator). It operates in fixed-size batches of 512 samples, which provide the best compromise in terms of performance and latency.

3.2. Phase Vocoder Analysis

The steps involved in PV Analysis are detailed in Figure 2 [12]. At the interval of hopsize samples, we window and rotate an input frame of time-domain data and apply a DFT to it. To obtain the PV data in a flexible amplitude + frequency (Hz) format, we then apply a conversion operation that takes the data from rectangular to polar representations and calculates the one-frame phase difference at each bin, then converts it from radians per hopsize samples to cycles per second.

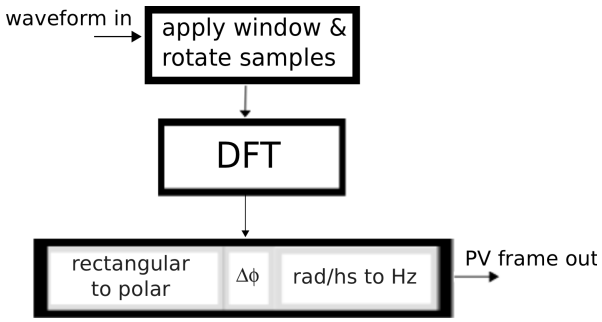

Figure 2: *PV analysis*

These three operations are good candidates for GPU co-processing, as they can be parallelised. The window and rotation operations affect each sample separately, so they can be implemented in very simple kernels.

```
__global__ void
rotatewin(float* out, float* in, float *win,
       int N, int offset){
  int k = threadIdx.x +
          blockIdx.x*blockDim.x;
  out[(k+offset)%N] = win[k]*in[k];
}
```

In the code above `k` is the thread index, which is used to access a given sample in the input and output frames, `N` is the DFT size and `offset` is the rotation offset, that depends on the current frame index and the hopsize. The kernel is made transparent to deployment on any number of blocks, so that decision of how many threads per block can be made separately (or even dynamically).

Similarly, the conversion to PV parameters is eminently parallel, dealing with each bin separately. Since 0Hz does not need to be processed, we offset the thread index to start from bin 1. Computation is done in double precision, but PV data is stored as single-precision values following the convention for fsigs in Csound.

```
__device__ double modTwoPi(double x)
{
  x = fmod(x,TWOPI);
  return x <= -PI ? x + TWOPI :
    (x > PI ? x - TWOPI : x);
}

__global__ void
topvs(float* frame, double* oldph,
     double scal, double fac){
  int k = threadIdx.x +
       blockIdx.x*blockDim.x + 1;
  int i = k << 1;
  float re = frame[i], im = frame[i+1];
  float mag = sqrtf(re*re + im*im);
  double phi = atan2(im,re);
  double delta = phi - oldph[k-1];
  oldph[k-1] = phi;
  frame[i] =  mag;
  frame[i+1] = (float)
          ((modTwoPi(delta) + k*scal)*fac);
}
```

Finally, the DFT is implemented using the CUDAFFT library `cufftExecR2C()` function, which is optimised to run on the GPU hardware. Processing from real to complex data is in place, in the format expected by the windowing and rotation, and PV conversion kernels. This simplifies the memory handling, as there is only the need to copy the input data to the GPU and copy the PV output back to the host once. This is the minimum necessary data transfer to and from the device. The location of the frame, waveform, and phase history data is not defined in the kernel code, but in the current implementation, global device memory is used. Shared memory cannot be used for two reasons: in the case of waveform and frame data, it would require access to shared memory pointers by the host which is not available; and in the case of phase history, it would break the unit generator reentrancy condition.

In summary, we have the following sequential steps performed at each hopsize interval:

1. A frame of waveform samples is copied to the device
2. A kernel of N threads running `rotatewin()` is launched
3. DFT is performed with `cufftExecR2C()`
4. A kernel of N/2-1 threads running `topvs()` is launched.
5. A frame of amp + frequency data is copied from the device

Around this GPU-specific code, the host takes care of collecting the input samples into the waveform frames that will be sent to the device, as well as making the output frequency-domain signal available to the rest of Csound.

3.3. Phase Vocoder Synthesis

PV synthesis basically re-trace the steps of analysis in reverse (Figure 3). As before, we have three highly parallel steps. The conversion from PVS parameters into rectangular spectral data is provided by the following kernel:

```
__global__ void
frompvs(float* inframe, double* lastph,
    double scal, double fac) {
  int k = threadIdx.x +
    blockIdx.x*blockDim.x + 1;
  int i = k << 1;
  float mag = inframe[i];
  double delta = (inframe[i+1]
                       - k*scal)*fac;
  double phi = fmod(lastph[k-1]
                      + delta, TWOPI);
  lastph[k-1] = phi;
  inframe[i] =  (float) (mag*cos(phi));
  inframe[i+1] = (float) (mag*sin(phi));
}
```

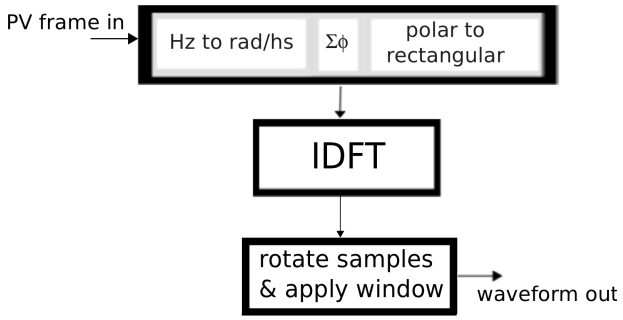

Figure 3: *PV synthesis*

Rotation and windowing is, as in the analysis case, very straightforward:

```
__global__ void
winrotate(float* out, float* in, float *win,
            int blocks, int N, int offset){
  int k = threadIdx.x +
       blockIdx.x*blockDim.x;
  out[k] = win[k]*in[(k+offset)%N];
}
```

The steps involved in PV synthesis are:

1. A frame of PV data is copied to the device
2. A kernel of N/2-1 threads running `frompvs()` is launched.
3. Inverse DFT is performed with `cufftExecC2R()`
4. A kernel of N threads running `winrotate()` is launched
5. A frame of waveform samples is copied from the device

3.4. Additive Synthesis

At face value, additive synthesis appears to be a very suitable technique for GPU implementation, given the fact that it is based on generating independently-computed sinusoidal streams and mixing them together. However, in practice there are some issues that need to be solved, namely

- a suitable sinusoidal oscillator design
- memory use/access

For full-spectrum synthesis using all the analysis data, we use two steps, both involving independent computations, which can be parallelised: an oscillator bank, and the parameter update.

In designing the oscillator that will be used we have to consider in particular the fact that conditionals are very costly in GPU code (as discussed above). A standard table lookup oscillator with floating-point indexing is not very efficient because of the conditional checks and moduli operations for index bounds. An alternative is to use integer indexing and with fast wrap-around using bitmasks. In addition, we have observed that table memory access has an inherent cost (even if the table is loaded to shared memory, which has the fastest access), and direct use of trigonometric functions is about 23% faster.

The most problematic issue with additive synthesis on the GPU is memory access, which can take up a significant amount of the total process time. Additive synthesis, in comparison to PV synthesis, can be relatively memory-hungry. For example, it requires a minimum of N $\times$ H floating-point numbers, where N is the number of bins and H the hopsize. For the full reconstruction of a 2048-point analysis (1024 bins), hopping by 256 samples, we have a 1MB memory requirement for single-precision samples. This memory would need to be accessed twice by the device: for writing by each oscillator, and for reading at a mixdown stage. In this particular case, memory access costs can amount to almost 70% of the total computation time. Reducing the hopsize does not mitigate the problem, because it will increase the number of times a given kernel executes. A solution is to use atomic additions, if these are available and sufficiently fast. In this case, the mix down of each sample can be at the time of the sample generation, and no further memory access is required. In the cases where atomic operations are costly, then we will need to write every partial to memory first, and, in a second sequential step, mix all of them down (in parallel). CUDA offers a very efficient atomic addition for float samples, so we can take advantage of it.

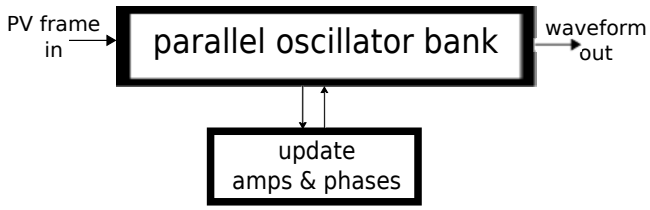

Figure 4: *Additive synthesis*

Of course, is always possible to synthesise a smaller number of bins, which would reduce both memory access and raw computation. In any case, each sample of each partial can be independently calculated (see [13]). For this we can spawn N $\times$ H kernels, each contributing a single sample to their respective partial, in effect parallelising across bins and samples. The additive synthesiser as implemented here is shown on Figure 4. The kernel used compute each sample is shown below:

```
#define MAXNDX ((MYFLT) 0x40000000)
#define PHMASK   0x3FFFFFFF

__global__ void sample(float *out,
  float *frame, float pitch,
  int64_t *ph, float *amps,
  int bins, int vsize, MYFLT sr) {
  int t = (threadIdx.x +
      blockIdx.x*blockDim.x);
  int n =  t%vsize;
  int h = t/vsize;
  int k = h<<1;
  int64_t lph;
  float a = amps[h], ascl = ((float)n)/vsize;
  MYFLT fscal = pitch*MAXNDX/sr;
  lph = (ph[h] + (int64_t)
   (n*round(frame[k+1]*fscal))) & PHMASK;
  a += ascl*(frame[k] - a);
  atomicAdd(&out[n],
           a*sinf((2*PI*lph)/FMAXLEN));
}
```

It takes in single-precision amplitudes and frequencies in a PV-format `frame`, which has been copied from the host into the device, and writes its output sine wave to `out`, For sake of efficiency, we interpolate amplitudes linearly, but frequencies only change at hopsize intervals. Output memory is accessed via an atomic addition. The layout of kernels with respect to bins and samples is shown in 5.

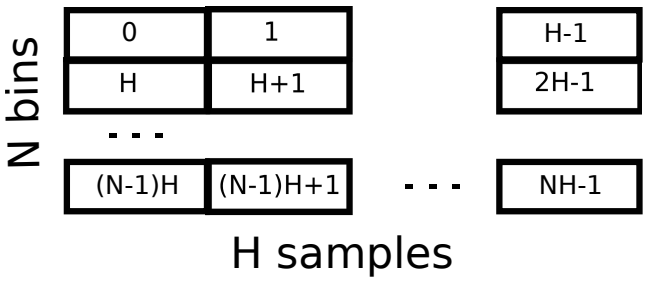

Figure 5: *Layout of kernels for additive synthesis*

The synthesis expression for each kernel is given by

$$s_{hn}(n) = \left\{a_h(t-1) + [a_h(t) - a_h(t-1)]\frac{n}{H}\right\} \times sin(\phi_h(t) + \omega_h(t)n) \quad (1)$$

where h and n are the bin and sample indexes, respectively, H is the hopsize, and t is the time in hopsize samples. The bin amplitudes are found in $a_h(t)$ and $\omega_h(t) = 2\pi\frac{f_h(t)}{f_s}$ are the bin frequencies (with f_s as the sampling rate and f_h the bin frequency in Hz).

A separate second step is needed to update the synthesis parameters for each bin (amplitudes and oscillator phases). The phases $\phi_h(t)$ are updated according to:

$$\phi_h(t+1) = \phi_h(t) + \omega_h(t)H \quad (2)$$

and the amplitudes are updated directly from the input PV frames. This operation is parallel across bins:

```
__global__ void update(float *frame,
     float *amps, int64_t *ph,
     int vsize, MYFLT sr){
```

```
 int h = threadIdx.x
    + blockIdx.x*blockDim.x;
 int k = h << 1, i;
 ph[h]  = (ph[h] + (int64_t)
    (vsize*round(pitch*frame[k+1]*MAXNDX/sr)))
     & PHMASK;
 amps[h] = frame[k];
}
```

The memory transfer from device to host that follows the execution of these kernels is limited to a hopsize vector of floating-point samples.

3.5. Sliding Phase Vocoder

If the hopsize is set to its smallest value, 1, the process can be seen in another way. The advantages and drawbacks are described elsewhere[6], but the algorithm is highly parallel. This can be seen as the extreme case for phase vocoding, but it offers possibilities for use of the GPU architecture.

3.5.1. The Underlying Mathematics

The Discrete Fourier Transform (DST) is defined by the formula

$$F_t(n) = \sum_{j=0}^{N-1} f_{j+t} e^{-2\pi i j n/N} \tag{3}$$

where the PCM-coded input signal is f_t, and $F_t(n)$ are the n frequency (complex) amplitudes for time t, and N is the (assumed) cyclic period of the signal.

If we know the values $F_t(n)$ we can determine $F_{t+1}(n)$:

$$F_{t+1}(n) = \sum_{k=0}^{N-1} f_{k+t+1} e^{-2\pi i k \frac{n}{N}} \tag{4}$$

$$= \sum_{k=1}^{N} f_{k+t} e^{-2\pi i (k-1) \frac{n}{N}} \tag{5}$$

$$= \left(\sum_{k=0}^{N-1} f_{k+t} e^{-2\pi i k \frac{n}{N}} - f_t + f_{t+N} \right) e^{2\pi i \frac{n}{N}} \tag{6}$$

$$= \left(F_t(n) - f_t + f_{t+N}\right) e^{2\pi i \frac{n}{N}} \tag{7}$$

If the values of $F_t(n)$ are kept on the GPU the need for data transfer is much reduced, but we can make use of the transfer of blocks of data for f_t and f_{t+N} at the expense of some latency.

There is however an immediate problem; the window cannot be applied in the time domain. The solution in this case is to apply the window as a frequency-domain convolution. That is to say, it can be applied after the calculation of the DFT as multiplication of the spectral transform of the window. Indeed for cosine-based windows this operation is simple[14].

In the paper by Moorer ([15]) a complicated inverse formula is developed. However it requires N^2 data to be maintained and is clearly impractical, especially on a memory-limited GPU. Instead we use a direct calculation of the definition of the inverse DFT:

$$f_t = \frac{1}{N} \sum_{n=0}^{N-1} F_t(n) e^{2\pi i t n/N} \tag{8}$$

but as we only need consider one value of t for each frame this is more efficient than the formula would suggest. For a single point $t = 0$ this simplifies to

$$\frac{1}{N} \sum_{j=0}^{N-1} F_0(j) \tag{9}$$

3.5.2. Implementation

A GPU-based Csound opcode was developed from the code in [16], where we take an audio input, apply a Transformational FM process[6] and resynthesise it. In this application, the sliding PV allows the unique effect of audio-rate frequency modulation of spectral data. The initialisation function is required to organise CUDA memory for the bin data and the pre-calculate a number of constants (*e.g.* $e^{2\pi i \frac{n}{N}}$). The main processing is done in small vectors of samples and it involves the following steps

1. A vector of samples is copied to the device
2. The sliding DFT is performed (on sample-by-sample basis), in parallel across bins by N/2+1 `slide()` threads (where N is the DFT size).
3. DFT to PV conversion, followed by frequency modification, and finally, PV to DFT conversion is performed by N/2+1 `fmsyn()` threads
4. Reconstruction is performed in parallel across the time-domain samples by vectorsize `reconstruct()` kernels.
5. A vector of samples is copied from the device to the host

4. RESULTS AND DISCUSSION

In this section, we would like to demonstrate that it is possible to execute all of the code discussed in realtime, with low-latency wherever possible, which was the original requirement that we have set out to prove. In testing these conditions, we employ a soundfile as input to the process, and make the requirement for realtime that computation time is less than the duration of input data. For a low-latency condition, we need to have the ratio between computation time and input duration fairly small so that any significant jitter in the computation load is not translated as dropped samples (also known as *xruns*). Tests included running the code to the digital-to-analog converter in realtime using buffer sizes of 128 frames (3ms at $f_s = 44100$) without xruns, which can also characterise a low-latency condition. Timings were taken from the total computation time recorded by Csound, which lumps the serial and parallel code, but since the interest here is the feasibility of the system as whole, this is exactly what we want to measure. The reported times are the average of five runs, but we have observed very little deviation in the individual results. Below, we discuss the individual results for each process.

4.1. PV analysis

The following Csound 6 code was used to test the PV analysis process. It consists of a soundfile input, the GPU-run analysis (`cudanal`) and a standard CPU-based PV synthesis (`pvsynth`, also used to provide a means to check the correctness of the output).

```
/* soundfile input */
asig = diskin:a("flutec3.wav",1,0,1)
/* GPU  PV analysis */
fsig = cudanal(asig,
               ifftsize,
               ihopsize,
               ifftsize, 1)
/* PV synthesis */
asig = pvsynth(fsig)
asig = linenr(asig,0.005,0.01,0.01)
out(asig)
```

Table 2: *GPU PV analysis program times for a 60-sec run.*

(DFT size, hopsize)	time (secs)
(1024, 128)	2.95
(1024, 256)	1.68
(2048, 256)	2.20
(2048, 512)	1.28

For this algorithm, we have observed that the best performance is obtained by maximising the number of threads in a block. Thus we distribute the threads so that they fill the blocks completely, up to the limit of 1024 threads. Since the number of threads is determined by the DFT size, we will be using single blocks for transforms of less than 1024 samples and multiple blocks above this.

The times for a 60-sec run of the program with various combinations of DFT size and hopsize are shown in table 2. This indicates that the best match in terms of performance is that of a 2048 transform every 512 samples. We can assess this as a generally efficient performance, with timings more than 20× faster than the realtime limit.

4.2. PV synthesis

Similarly to above, in order to isolate the performance of the synthesis process, we employ a program that uses an analysis element running in the CPU (`pvsanal`), followed by the GPU synthesis code (`cudasynth`):

```
/* soundfile input */
asig = diskin:a("flutec3.wav",1,0,1)
/* PV analysis */
fsig = pvsanal(asig,
               ifftsize,
               ihopsize,
               ifftsize, 1)
/* GPU PV synthesis */
asig = cudasynth(fsig)
asig = linenr(asig,0.005,0.01,0.01)
   out(asig)
```

Results are shown in table 3. They also indicate a reasonable performance, confirming the best combination of parameters obtained in the analysis tests.

4.3. PV analysis & synthesis

Also interesting is the combination of GPU analysis and synthesis, and following the results above, we can predict that they will be well within realtime capabilities. This is the program used (and the results are shown on Table 4):

Table 3: *GPU PV synthesis program times for a 60-sec run.*

(DFT size, hopsize)	time (secs)
(1024, 128)	3.30
(1024, 256)	1.84
(2048, 256)	2.65
(2048, 512)	1.44

```
/* soundfile input */
asig = diskin:a("flutec3.wav",1,0,1)
/* GPU PV analysis */
fsig = cudanal(asig,
               ifftsize,
               ihopsize,
               ifftsize, 1)
/* GPU PV synthesis */
asig = cudasynth(fsig)
asig = linenr(asig,0.005,0.01,0.01)
out(asig)
```

Table 4: *GPU PV analysis & synthesis program times for a 60-sec run.*

(DFT size, hopsize)	time (secs)
(1024, 128)	4.72
(1024, 256)	2.57
(2048, 256)	3.03
(2048, 512)	1.73
(4096, 512)	1.98
(4096, 1024)	1.20
(8192, 1024)	1.64
(8192, 2048)	1.01
(16384, 2048)	1.38
(16384, 4096)	0.86

These results demonstrate that a full PV analysis/synthesis program can be run quite efficiently on the GPU. However, if we compare it to PV code run sequentially in a high-performance CPU (in this case based on a 2.8GHZ Intel I7 processor), we see that it does not compare too well (Table 5) unless the DFT size is significantly large. Even though the scope of this research is not to draw comparisons between CPU and GPU capabilities for audio processing, it is important to note these results. The major shortcomings of the GPU for the particular processes discussed here are to do with the costs involved in launching kernels in a reasonably fine grain (determined by the hopsize), and memory access. If we examine the sequential steps involving the GPU in the analysis or synthesis code, we will see that these are the most costly portions of the code (with their average computation load):

- memory transfers: 40 - 45%
- FFT: 30 - 35%
- PV parameter conversion: 15 - 20%

None of these issues are particularly avoidable as they are not represented in sections of code that are good candidates for optimisation.

Table 5: *CPU-based PV analysis & synthesis program times for a 60-sec run.*

(DFT size, hopsize)	time (secs)
(1024, 128)	1.24
(1024, 256)	0.69
(2048, 256)	1.28
(2048, 512)	0.70
(4096, 512)	1.34
(4096, 1024)	0.74
(8192, 1024)	1.36
(8192, 2048)	0.75
(16384, 2048)	1.40
(16384, 4096)	0.77

Nevertheless, the results point to the fact that the GPU may be used as a means of freeing up some computation load from the CPU in a multicore/multiprocessor operation scenario. We also anticipate a considerable speedup on more capable hardware, where we are likely to see the GPU outperforming these CPU results.

4.4. Additive synthesis

In general, Additive synthesis does not perform as well as PV synthesis, and in the parallel case there is still a significant difference in performance between the two techniques, even if at times it is the gap is considerably less than in the serial case. The program used for tests is the following (`cudasynth` implements the additive algorithm):

```
asig = diskin:a("flutec3.wav",1,0,1)
fsig = pvsanal(asig, ifftsize, ihopsize,
                        ifftsize, 1)
asig = cudasynth(fsig,1,1,ibins)
asig = linenr(asig,0.001,0.01,0.01)
out(asig)
```

The results are shown in Table 6, with regards to bins and hopsizes (DFT sizes are shown for completion, but they do not influence computation load), with GPU and CPU times side-by-side. A highly optimised serial additive synthesis algorithm was used for this comparison, replacing the `cudasynth` opcode in the listing above.

Overall, the performance is still well within the range for low-latency realtime performance (< 12% of total input duration at the worst case), Comparatively, additive synthesis proves to be a good match for the GPU, especially in the case of full-spectrum reconstruction, and with larger hopsizes. The parallel code performs worse only in the case where the hopsize is small comparatively to the number of bins. This is mostly due to the fact that, in this case, the balance between the parallel load and the process granularity is not ideal. We can observe that this makes an important contribution to the computation cost. The granularity penalty is shown by comparing the cost of calling one grid of 65536 threads (256 bins, 256 hopsize) and two grids containing 32768 threads each (256, 128), where we observe almost 100% slowdown. This shows that GPUs are more suited to larger batches of data, which is not ideal in the streaming processing case.

Table 6: *GPU additive synthesis program times for a 60-sec run.*

(DFT size, bins, hopsize)	GPU time (secs)	CPU time (secs)
(1024, 128, 128)	4.93	3.28
(1024, 128, 256)	3.70	3.01
(1024, 256, 128)	7.20	5.77
(1024, 256, 256)	3.37	5.46
(1024, 512, 256)	4.20	10.76
(2048, 256, 512)	3.04	5.65
(2048, 512, 512)	3.94	10.55
(2048, 1024, 512)	6.87	20.89

These results are very encouraging, and follow other reports of additive synthesis, such as [13], but are not as extremely performant as one might anticipate (the best speed up is of the order of 3) . However the conditions in our case are much more restrictive than in other tests. We have implemented here a fully-flexible general-purpose application of additive synthesis, where we cannot run the processing in large batches, or apply other cost-saving measures that would maximise the GPU processing load. In particular, in order to keep latency and realtime control to a satisfactory minimum, as well as have good reconstruction quality, processing granularity is never bigger than 1/4 DFT size. We also should note that the results obtained in [17] are more in line with the ones reported in this paper.

4.5. Sliding PV

The sliding phase vocoder CUDA opcode (`cudasliding`) combines analysis, frequency scaling and resynthesis. It was tested with the Csound program

```
asig = diskin:a("flutec3.wav",1,0,1)
amod = 1
asig2 = cudasliding(asig,amod,idftsize)
asig = linenr(asig2,0.005,0.01,0.01)
out(asig)
```

and the performance compared with a similar program running solely on the host computer CPU.

Table 7: *GPU and CPU sliding PV program times for a 60-sec run.*

DFT size	GPU time (secs)	CPU time (secs)
512	33.05	68.794
1024	37.98	138.29
2048	54.99	272.33

The results are shown in Table 7. As can be seen that the times using the GPU are within real time, but considerably slower than the standard phase vocoder with GPU support (Table 4). The figures also are slower than reported by [16] on different hardware with more computing capacity. It also suggests that much more work will be needed if the Sliding Constant-Q algorithm[18] that

needs three streams of SDFT to be calculated is to be available for realtime on consumer-level GPUs.

5. CONCLUSIONS

In this paper, we set out to investigate the implementation of streaming spectral processing operations in a consumer-level GPU attached to an off-the-shelf desktop computer under a commonly-used music programming environment, Csound. The full CUDA source code for these unit generators, and a CMake build script, can be found in the Csound git repository:

https://github.com/csound/csound.git

These opcodes are fully integrated into the standard system and are included in the present release (6.03, April 2014).

We have demonstrated that each one of the processes detailed here can be executed in realtime with low latency. The standard algorithms can all generally be executed with good performance, and, among these, additive synthesis is comparatively less efficient, although the parallel version generally outperforms the serial one. With the novel SPV process, we see significant gains, with up to 5× speedup, where the improvements allow the code to be used in realtime. We have identified that the major costs are related to memory transfers from host to device and vice-versa, and device memory access. We believe that this work demonstrates that consumer-level GPU processing can be harnessed for audio applications. In particular a number of novel digital audio effects can be designed to take advantage of the GPU implementation for realtime performance.

6. REFERENCES

[1] L. Savioja, V. Valimaki, and J. O. Smith, "Audio signal processing using graphics processing units," *J. Audio Eng. Soc.*, vol. 59, no. 1, pp. 3–19, 2011.

[2] Niklas Roeber, Ulrich Kaminski, and Maic Masuch, "Ray Acoustics Using Computer Graphics Technology," in *10th Proc. Digital Audio Effects (DAFx-2007)*, Bordeaux, France, 2007.

[3] L Savioja, "Use of GPUs in room acoustic modeling and auralization," in *Proceedings of the International Symposium on Room Acoustics (ISRA 2010)*, Melbourne, Australia, 2010.

[4] P-Y Tsai, T-W Wang, and Su Alvin, "GPU-based Spectral Model Synthesis for Real-time Sound Rendering," in *13th Proc. Digital Audio Effects (DAFx-2010)*, Graz, Austria, 2010.

[5] B Hamilton and C Webb, "Room Acoustics Modelling Using GPU-Accelerated Finite Difference and Finite Volume Methods on a Face-Centered Cubic Grid," in *16th Proc. Digital Audio Effects (DAFx-2013)*, Maynooth, Ireland, 2013.

[6] Russell Bradford, Richard Dobson, and John ffitch, "The Sliding Phase Vocoder," in *Proceedings of the 2007 International Computer Music Conference*, Suvisoft Oy Ltd, Ed. August 2007, vol. II, pp. 449–452, ICMA and Re:New, ISBN 0-9713192-5-1.

[7] "CUDA C Programming Guide," `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2014.

[8] Richard J. Boulanger, Ed., *The Csound Book: Tutorials in Software Synthesis and Sound Design*, MIT Press, February 2000.

[9] J. Fitch, V. Lazzarini, S. Yi, Gogins M., and A Cabrera, "The New Developments in Csound 6," in *Proc. Intl. Computer Music Conf.*, Perth, Australia, 2013.

[10] V Lazzarini, J Timoney, and T. Lysaght, "Spectral Signal Processing in Csound 5," in *Proc. Intl. Computer Music Conf.*, New Orleans, USA, 2006.

[11] Russell Bradford, Richard Dobson, and John ffitch, "Sliding is Smoother than Jumping," in *ICMC 2005 free sound*, SuviSoft Oy Ltd, Tampere, Finland, Ed. Escola Superior de Música de Catalunya, 2005, pp. 287–290, `http://www.cs.bath.ac.uk/~jpff/PAPERS/BradfordDobsonffitch05.pdf`.

[12] V. Lazzarini, *The Audio Programming Book*, chapter The Phase Vocoder, pp. 91–122, MIT Press, Cambridge, Massachusets, 2010.

[13] L. Savioja, V. Valimaki, and J. Smith, "Real-time additive synthesis with one million sinusoids using a gpu," in *Proc. 128th AES*, London, UK, 2010.

[14] John ffitch, Richard Dobson, and Russell Bradford, "Sliding DFT for Fun and Musical Profit," in *6th International Linux Audio Conference*, Frank Barknecht and Martin Rumori, Eds., Kunsthochscule für Medien Köln, March 2008, LAC2008, pp. 118–124, Tribun EU, Gorkeho 41, Bruno 602 00, ISBN 978-80-7399-362-7.

[15] James A. Moorer, "Audio in the New Millennium," *J. Audio Eng. Soc.*, vol. 48, no. 5, pp. 490–498, May 2000.

[16] Russell Bradford, John ffitch, and Richard Dobson, "Real-time Sliding Phase Vocoder using a Commodity GPU," in *Proceedings of ICMC2011*. University of Huddersfield and ICMA, August 2011, ICMC, pp. 587–590, ISBN 978-0-9845274-0-3.

[17] A. J. Bianchi and M. Queiroz, "MEASURING THE PERFORMANCE OF REALTIME DSP USING PURE DATA AND GPU," in *Proc. Intl. Computer Music Conf.*, Ljubljana, Slovenia, 2012.

[18] Russell Bradford, Richard Dobson, and John ffitch, "Sliding with a Constant *Q*," in *Proc. of the Int. Conf. on Digital Audio Effects (DAFx-08)*, Espoo, Finland, Sep 1-4 2008, DAFx08, pp. 363–369, ISBN 978-951-22-9517-3.