CUDA Accelerated Audio Digital Signal Processing for Real-Time Algorithms

Nicholas Jillings¹ and Yonghao Wang²

¹ Birmingham City University, Birmingham, B4 7XG, United Kingdom
Nicholas.Jillings@mail.bcu.ac.uk

² DMT Lab, Birmingham City University, Birmingham, B4 7XG, United Kingdom
Yonghao.Wang@mail.bcu.ac.uk

ABSTRACT

This paper investigates the use of idle graphics processors to accelerate audio DSP for real-time algorithms. Several common algorithms have been identified for acceleration and were executed in multiple thread and block configurations to ascertain the desired configuration for the different algorithms. The GPU and CPU performing on the same data sizes and algorithm are compared against each other. From these results the paper discusses the importance of optimising the code for GPU operation including the allocating shared resources, optimising memory transfers and forced serialisation of feedback loops. It also introduces a new method for audio processing using GPU’s as the default processor instead of an accelerator.

1. INTRODUCTION

Audio DSP plays a major role within the music industry with songs, films and other audible media being created and processed entirely within the digital realm. Processing is traditionally performed, in a personal computer, through the central processing unit (CPU) which is programmed to apply the audio algorithm to an incoming data stream. Personal computers also house advanced graphics processing units (GPU) which create the 2D and 3D images on computer monitors. These chips have been growing in power and have reached a point where they can be programmed for alternative, general purpose computing [1]. The project focuses on real-time, time-domain algorithms. Frequency-domain algorithms are ignored due to the variations in performance of FFT libraries which can bias the results from the actual processing being measured. Real-Time ensures a stream based system instead of other audio processes such as data compression. The overall aim is to evaluate the application of GPU processing in real time digital audio signal processing. To reach this aim the following objectives should be met.
2. REVIEW OF EXISTING KNOWLEDGE

Computer processors have been categorised into 4 groups based on their processing style by Michael Flynn and is commonly referred to as Flynn’s Taxonomy. The groups are Single Instruction Single Data, Single Instruction Multiple Data, Multiple Instruction Single Data and Multiple Instruction Multiple Data. These are further known in the SISD, SIMD, MISD and MIMD form and will be referred in this form in this publication.

A modern PC CPU is based on a multi-core package which makes the CPU an MIMD processor and highly flexible for multi-tasking. However [2] indicates CPUs have been developing towards SOC (system-on-chip) packages with many cores, graphics processing, memory interface and high speed PCI-E busses all included in the CPU package. This aims to drive up power efficiency of the entire system and provide increased performance without any radical changes to the physical processing cores.

GPU chips are in everyday computers and [3] states these are personal high performance compute devices. The GPU, when used for general purpose, operates similarly to an SIMD but is closer to a Single Instruction Multiple Thread because of the flexibility in processing. The GPU today can perform all of the four states of Flynn’s Taxonomy.

In SISD mode it is one thread operating on one piece of data, whilst hugely inefficient and wasteful it is technically possible to perform this mode. SIMD requires all threads perform the same instruction synchronously but on different data, this publication focusses on the application of this mode. MISD requires threads to perform different instruction on the same data. On smaller chips where there are less threads this may be possible but because thousands of threads could be launched this is also potentially wasteful. MIMD is definitely possible through careful management of threads into groups performing individual instructions. Because of the SIMD-like architecture programming involves separating an algorithm into multiple identical operations which can be run independently of each other and therefore is best suited to algorithms with a high degree of repetition. The power of the GPU comes from its ability to execute many threads simultaneously [1]. CUDA is nVidia’s brand for their programming API and defines the chip capabilities and specifications based on the version. CUDA chip and CUDA API versions are different and are generally interchangeable with each other as new drivers are developed for older chips. The CUDA chips are backwards compatible meaning the modern Kepler (CUDA 3.0 and 3.5) chips can execute a code compiled for a CUDA 1.0 chip.

GP_gpu operations are becoming commonplace and there are papers indicating hundreds to thousand times performance speed-ups over the CPU. [4] and [5] try to show the flaws in either processing or presenting the results. Firstly the GPU and CPU are very different architectures and [4] highlights the differences in cache, core structures and memory. CPU’s have large numbers of cache, such as the Intel Core i7-960 in [4] giving 32KB L1, 256KB L2 per core and 8MB L3 for the whole processor. These cache levels allow large amounts of data to be stored. [5] also shows ways that performance can be masked and this paper attempts to avoid these pitfalls to provide as fair an overview as possible.

CUDA chips are multi-core systems broken up into cores called stream multiprocessors (often shortened to SM or SMP). The SM is broken up into the CUDA cores [6] and the number of cores inside each SM depends on the generation. The CUDA cores contain integer and single precision floating point units and is where the threads execute. The SM also holds special units to calculate addresses and also a special function unit (SFU) to computer sine, cosine, reciprocal and square roots [6]. Double precision is also in a separate unit as CUDA 1.0 did not have this unit and therefore does not support double precision.

A CUDA application is executed by arranging threads into grid of blocks. The blocks is a 3D matrix of threads and these are placed into the grid. All blocks in the grid must have the same number and order of threads. On
execution a block of threads is given to the SM which executes the threads in groups of 32, called a warp. The blocks can be assigned randomly to the SM and the warp executed is also unknown, therefore it is not possible to predict which threads will execute. This method of grids and blocks allows for automatic scaling of the application if launched on a different GPU with a different number of SMs.

Audio applications utilizing the GPU have shown significant speed ups over CPU versions. A Fast Fourier Transform was shown to be four times faster than using a high-end quad core CPU [7]. [8] used a GPU to perform additive synthesis of one million sinusoids. The CPU peaked at 439 sinusoids generated in real-time whilst the GPU peaked at 1395 sinusoids in real-time. [9] simulated wave field synthesis using an nVidia GTX 285 GPU against an Intel core i7-920 CPU. The GPU achieved speed-ups of three and a half times for frame sizes of 8192 samples, smaller frame sizes under 512 samples favoured the CPU. These are all systems which use the GPU for acceleration for items outside the scope of this project as they involve synthesis or frequency domain processing. All however show the potential performance in a CUDA processor.

[10] processed audio using the shader languages. The paper lists the process names but does not give the algorithms or methods making it difficult to interpret the results. It does highlight problems using early programmable shaders such as converting audio into vertex information to perform calculations adding extra headroom. The GPU’s of the time only supported up to 16-bit depth processing because they were still focused on graphics where higher precision was not needed [11].

Most resources specify the largest bottlenecks of GPU processing are memory latency and memory bandwidth [12]. Memory latency is the on-chip delay of fetching data from the global device memory. [8] reported it taking between 400 and 600 clock cycles. Memory transfer bandwidth is the data transfer rate from the host RAM to the GPU Global memory and [12] states it can cause a program with streaming audio to operate at 1/10th original speed.

There are several memory stores on the GPU called Local, Shared, Global, Texture and Constant [13]. The shared memory is the only store exclusively located on chip. It is a fixed amount per SM and varies from 16KB in CUDA 1.0 up to 48KB in CUDA 3.0/.5. Memory in this store is accessible by all threads of the same block. Even though an SM may execute multiple blocks, thread 1 in Block A cannot read data loaded into Block B.

The next thread-level store is the Local store and is only accessible by the individual thread. Data values created by the thread are stored here, however the local store is partially on-chip and if this fills up, it overflows into the slower global memory.

Global memory is the main GPU Heap, similar to RAM for the CPU, and is accessible to all threads of all blocks and the CPU. It is persistent between executions unlike shared and local which are cleared each time a new kernel is launched. This memory is accessed asynchronously and is moved over in blocks of 128-bit, 64-bit or 32-bit as explained in [14]. Storage in the global heap is aligned to 256-byte intervals, therefore arrays not integer multiples of 256-bytes will waste space.

In the global memory are two special areas called constant and texture. Constant is a small 64KB memory space designed for multiple reads from different threads. It can only be written to by the CPU. Texture is similar but is designed for multi-dimensional array and vertex information and is also CPU written and GPU read only. Both are accessible by all GPU threads.

When writing or reading data on the GPU, the data is transferred using the PCI-E link. This link is a high bandwidth link and at the current top specification of PCI-E version3 at 16x lanes achieves a bandwidth of just under 16GB/s per direction.

From [15] and [16] the following audio processing algorithms were extracted to test their performance on a GPU compared to a CPU:

- Gain
- IIR 1st and 2nd order filters on an Allpass filter design
- FIR filters using the Window Design method
- Biquad Filter (Direct Transform II)
- Amplitude and Ring Modulation of a Sine wave
- Dynamic Range compression

These algorithms require different algorithm solutions to be applied to truly test the GPU when applying audio processing.

3. METHODOLOGY

The algorithms were converted into a minimum of two execution patterns: CPU Only and GPU Only. Some algorithms, such as IIR and Biquad, also had a mixture
of CPU and GPU operations to see if coprocessor implementations would be beneficial, using each processor for its strengths (parallel on GPU and serial on CPU).

The programs were written using Visual Studio Ultimate 2012, nVidia Nsight and CUDA runtime 5.5 (up to date at time of starting). Once the programs were designed, an initial test was performed to check the algorithms performed as expected and were stable before the main testing began. Analysis and interpretation of the results against performance form the majority of the report process. Each test was performed five times to reduce any errors generated either by the timing mechanism, system interrupts or other interferences.

The test environment was as follows:

- Operate on a generated waveform stored on system RAM
- Waveform stored in 32-bit single precision floating points (as pointed out in [5])
- Total number of 409,600 samples at 48kHz sample rate
- Frame sizes from 32 samples (32 threads in 1 block) up to 4096 samples (512 threads in 8 blocks).
- Process 1 channel to 128 channels (in eight channel increments)
- Timings performed by a custom class based on the Windows Performance Counter [14], timing entire execution (as pointed out in [5])

The test environment made the following assumptions:

1. The GPU was not being used for any intensive operations but was being used to actively generate the monitor output
2. The audio is offline but processing similar to a plugin where data must be transferred to the processor and returned within the time it would take to capture the next frame.
3. The timings is only once the execution has been handed to the algorithm and does not include any ‘one-time’ event such as coefficient calculations or memory allocations

4. The CPU is only executing in an SISD fashion with no SSE/SIMD functions

The test platform was an Alienware 17 2013 laptop running Windows 8.1. The CPU was an Intel i7-4700MQ at 2.4GHz coupled with 16GB DDR3L 1333 RAM and the GPU was an nVidia GTX780M 4096MB.

4. RESULTS

4.1. Expected Results

Past research had shown problems with CUDA systems and memory operations, therefore items which have higher memory access or low operations per sample are expected to underperform. Also pieces where the GPU is forced to perform serial algorithms would perform poorly, however this may improve when high channel counts are used since parallelisation will return, albeit inefficiently.

Small window algorithms are expected to perform better on the CPU due to the larger number of memory transfers across the PCIe lanes, which are designed for width than speed to provide throughput.

4.2. Presentation of Results

The results are shown first in the general average with a quick explanation on why the general outcome was as such. Then a brief over-view of each algorithm is shown with some highlight of their relative performance.

Generally the GPU provided ample improvement over the CPU when multiple channels are involved and the frame size above 512 samples is used. The higher both of these variables the higher the relative performance of the GPU. The coprocessor variations, where the GPU and CPU both perform a part of the algorithm, were underwhelming in their performance and in most situations were slower than either the GPU or CPU processing independently.

Algorithms which required more operations per sample performed better on the GPU. All implementations achieved real-time except some FIR filter executions on the CPU.
Figure 1 Comparison of Kernels running one channel above 1024 Frame Size

Figure 2 Comparison of Kernels running 128 channels

Figure 3 Comparison of Kernels running at Frame size of 512, all channel configurations

Figure 4 Comparison of Kernels running at Frame size of 4096, all channel configurations
Figures 1 and 2 show the performance difference when using a frame size greater than 1024 samples but between 1 and 128 channels. As can be seen when just one channel is used most of the algorithms underperform to the CPU with the CPU performing between 3x (the Amplitude and Ring Modulation) and 20x (Gain, IIR First order and FIR Delay with Allpass interpolation) faster. The FIR filter, not shown, was the only item to outperform the CPU by 5x and was not shown for clarity of the graphics. However when 128 channels is loaded the majority outperform the CPU with only the very light Gain, FIR Delay, feedback based IIR 1st order and FIR Delay with Allpass interpolation not outperforming, however not by very much. The ring and amplitude modulation were again strong performers achieving approximately 4x speed-up. The surprising performer was the Biquad achieving 20-21x speed improvement and was highly unexpected and possibly erroneous. Again the FIR filter was not shown as our implementation recorded an improvement of over 90x.

Figures 3 and 4 show the performance of all channel configurations at 512 frame size and 4096 frame size respectively. This shows the frame size has less of an impact than channel size for performance as most algorithms are spread over a very wide area. This could also be due to the large variations in block arrangement which can alter drastically the efficiency of the GPU. However it is very clear that the lighter algorithms (gain, FIR integer delay and FIR linear delay) benefit from the higher frame sizes.

4.2.1. Memory Bus

The GPU in the test rig is connected using a PCI-E 3.0 and 16 lanes directly to the CPU. A ‘dry-run’ test was performed which timed the transfer of the entire buffer at the frame size to and from the GPU. For one channel of 499,600 samples approximately 1.6MB of data is transferred but when 128 channels are used over 200MB has to be transferred. For the small frame and channel sizes there is a smaller amount of data (32 frame size of 1 channel only moves 128 bytes) and requires 12800 transfers. The largest size of 4096 samples (16KB) requires 100 transfers.

![Image showing data transfer rate over the PCI-E link at various frame sizes and channel counts](image)

Figure 5 Data Transfer rate of audio samples over the PCI-E link at various frame sizes and channel counts

Figure 5 shows that the link performs poorly if less than 256KB (65,536 samples) per transfer is used with a steep drop-off, however the link also only just exceeds 5GB/s which indicates there is a large portion of time being lost to prepare the move.

4.2.2. Amplitude and Ring Modulation

\[ y[n] = (1 + am[n]) \cdot x[n] \] (1)

\[ y[n] = am[n] \cdot x[n] \] (2)

Equations (1) and (2) show amplitude and ring modulation respectively. Both of these take an incoming data stream \(x\) and multiplies it with a modulation stream \(m\), generally a sine wave. This provides a useful time to use the GPU’s special SFU to calculate the sine wave, however it is only accessible when using the CUDA intrinsic (`_sine()` instead of `sine()`) and also carries some error.

As following the general result both implementations of each only achieve the highest results of 3x-4x performance when there are multiple channels and a large frame size. Both underperform massively when only one channel of data is used, however their multi-channel performance with sub 512 frame size comparable to the CPU. When using the SFU to calculate the sine both showed next to no significant speed improvement to justify the slightly decreased accuracy. However the fast sine did use less memory and overall less operations per sample.

4.2.3. Biquad Filter

This filter comprises of an IIR and FIR section in a Direct Form I layout using the 4 delay registers as in
equation (3). The implementations were all CPU and GPU and then a coprocessor variation where the FIR part of equation (4) was processed on the GPU and the results transferred to the CPU to perform the serial IIR part of equation (5).

\[
y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_0y[n-1] - a_1y[n-2] \tag{3}
\]

\[
y_{\text{FIR}}[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] \tag{4}
\]

\[
y_{\text{IR}}[n] = y_{\text{FIR}}[n] - a_0y[n-1] - a_1y[n-2] \tag{5}
\]

The first surprising result was the irregularity of the processing time when the CPU only processor was used. This irregularity occurred all 5 times the process was executed. When under 40 channels of audio was used, at all frame sizes, the system performed fairly consistently at 0.0062ms per sample. However this increased up to 0.138ms per sample at 128 channels, consistently across the various frame sizes. Because of this the GPU became the dominant processor. Both the IIR and FIR sections were low-weight for storage and operations. The FIR was ‘heavier’ because of the extra sample and coefficient to process.

4.2.4. Dynamic Range Compression

This publication focuses on a simple feed-forward model proposed in [12] using a full-wave rectification with an exponential moving average to smooth the signal before entering a gain computer. Because of this three separate kernels could be used for each of the three sections and therefore provided many levels of calculation. Because of its complexity and various requirements this struggled to outperform the CPU but did match it for the majority of states above 32 channels. Surprisingly it performed best in the 48-96 channels under 2048 samples. This may be because of the heavy Exponential Moving Average filter which required 26 registers and 58 operations per sample. Also the system moves from highly parallel full wave rectification to the forced serial moving average and back to the parallel gain computer. In total 104 operations per sample are performed.

4.2.5. FIR Delay with an integer delay

This is a very simple filter as it uses a single delay with a single coefficient. It is often referred to as a comb filter as the spectral output resembles that of a comb with evenly spaced notches and peaks.

\[
y[n] = x[n] + gx[n-M] \tag{6}
\]

Because of its simple nature it is mostly about memory access as threads need to access multiple incoming values and then return the computed value. The GPUs poor global memory performance shows as the GPU never outperforms the CPU with a maximum relative speed of 84.78%. The algorithm was implemented using a coalesced memory access and the previous sample taken from the stored shared memory, however the first thread of the block must access a non-coalesced or shared stored value.

4.2.6. FIR Delay with fractional delays

This is similar to the integer delays but uses a fractional delay line requiring interpolation. This delay increases the computation of the otherwise simple filter. From [12] a linear, allpass and spline interpolation algorithm was tested. The test did not account for any audible quality.

\[
y[n] = x[n-(M+1)] \cdot \text{frac} + x[n-M] \cdot (1-\text{frac}) \tag{7}
\]

Linear interpolation in equation (7) is the simplest form requiring only 2 multiplications and two additions. The GPU compiler combined these into two fused multiply addition (FMA) operations. Despite this optimisation the GPU never outperformed except for the high 128 channel / 4096 frame size (but only at 1.1x). It did however match the CPU’s performance time for the majority of tests except for sub 40 channels or sub 1024 frame sizes.

\[
y[n] = x[n-(M+1)] \cdot \text{frac} + x[n-M] \cdot (1-\text{frac}) \tag{8}
\]

Allpass interpolation, equation (8), is similar to the linear interpolation but includes a feedback element to add a polynomial quality. This, however, forces the GPU into a serialised mode and therefore increased the time on the GPU although not massively indicating that this is still a memory-locked algorithm. Spline interpolation is highly complex requiring many mathematical calculations to be performed. To perform
the equation a total of 433 operations must be performed (154 additions, 79 multiplications, 178 fused multiply additions and 22 special function units). It is unknown how many functions are redundant as many additions will be used for the address mapping and the fused multiply additions for the powers. This did outperform the CPU following the same general result curve, however it definitely was not spectacular achieving at best a 1.7x speed-up improvement.

4.2.7. Gain

Gain is one of the simplest DSP effects as it is a simple straightforward multiplication of the incoming signal with a value. Because of the low arithmetic content the CPU achieved incredible speeds of around 0.8 nanoseconds per sample. The GPU’s best speed was two times slower than the CPU and this was mostly due to the CPU having a spike of 27.3% in processing time up to 1.1 nanosecond at 128 channels.

4.2.8. IIR Filters

These filters have a feedback element requiring them to be processed in a serial fashion. Because of this restriction extra setups were created where one part was performed on the GPU and the other on the CPU. The IIR filters are based on the allpass design as described in [15].

\[ y_{allpass}[n] = c x[n] + x[n-1] - cy[n-1] \] (9)

\[ y[n] = 0.5(x[n] \pm y_{allpass}[n]) \] (10)

Equation (9) shows the allpass stage and equation (10) the combining with the original signal to create the filter. Adding the filter creates a lowpass whilst subtracting creates a highpass first order filter. Three coprocessor modes were tested: GPU operating equation (9) but not the feedback element which is processed on the CPU as is equation (10); CPU performing all of equation (9) and the GPU all of equation (10) and finally GPU operating equation (9) but not the feedback element which is processed on the CPU and equation (10) processed on the GPU.

These three tests should provide ample configurations for immediate parallelisation of respectable sections. A similar breakdown is performed for the second order. For the 1st order filter, independent processing on the CPU consistently outperformed the GPU however the GPU did follow the trend of better performance at high channel counts. The first coprocessor implementation is mostly a memory operation similar to the FIR Delay filter and had similar results albeit lower as there is more overhead with extra CPU processing. The second implementation requires more memory operations as it is combining two independent streams, however it does not provide an improvement. The final implementation may have been theoretically using each processor for their correct uses, however this is actually the worst performing of the three, most likely due to the extra memory transfer involved.

The 2nd order filter however did provide some improvements in the independent GPU configuration. At high channel counts the processor achieved 1.8x of speed-up over the CPU for channel counts above 64 channels and between 512 and ~3000 samples. However all coprocessor implementations still performed poorly at 0.8x, 0.85x and 0.78x of the CPU speed. These were closer than the 1st order filter and indicates that perhaps higher order IIR filters may perform well.

4.2.9. FIR Filter

FIR filters are heavy arithmetic functions requiring many multiplications and additions to perform. Due to their performance two methods of implementation have come around. The first is to apply through a convolution filter, which is the traditional method, whilst the second converts the signal and filter to the frequency domain, apply the filter and convert back to the time domain. This provides an identical signal and can be performed quicker. This implementation focuses on the slow convolution method as in equation (11).

\[ y[n] = \sum_{n=0}^{N} h[n] x[N-n] \] (11)

To adequately test the system the FIR coefficient count would equal the thread size, therefore a 512 thread, 1 block frame size would create a 512 FIR filter size. The filters were designed using the frequency-sampling method which meant the desired frequency response was created in the frequency domain, converted to the time domain and windowed using a Hann window.

The CPU, shown in Figure 6, managed to process the signals in a fairly linear fashion with the step up in frame size matching the time increase. The CPU does fail to meet real-time in certain points of the test. For instance, in the above figure, the CPU at 160 samples achieves a time of 0.187\(\mu\)s per sample time however when applied across 128 channels the time to process one sample of every channel grows to 23.94\(\mu\)s, slightly above the 20.83\(\mu\)s maximum for a real-time system at 48 kHz.
For the GPU processing, two approaches were taken. The first approach has each calculate the entire filter response for one output sample, this is the most similar to the original FIR equation (1). The second rotated this and has each thread hold one FIR coefficient and perform that multiplication on each incoming sample before an addition to collect all the values from the threads to calculate the output sample. This is similar to how the slow convolution method is performed if writing it out.

Both of these performed ample improvements in the majority of situations, even in a single channel as long as the frame size was larger than 512 samples. The first version attained a speed, on average, between 20x and 40x, peaking at 60x. The second version was significantly slower but still faster than the CPU.

The first implementation exclusively operated using fused multiply additions indicating why there was such a significant speed-up. The second implementation could not use this and had to perform individual additions and multiplications. However the cause for the performance drop is most likely due to the 2048 operations performed on SFU, potentially causing a bottleneck as threads stall waiting for spare SFU’s to become available. These are linked to the addition tree which uses modulo to calculate if a thread should take part in that round of addition or not.

5. DISCUSSION

The results show the GPU can provide some impressive performance compared to the CPU, however it is definitely not a ground-breaking level of performance. From the results there are four critical areas which determine the performance of the GPU for real-time audio:

- Global Memory Access Patterns
- Arithmetic Interpretation by the compiler
- Block and thread arrangement
- Shared Resource management

As already discussed accessing the global memory can be the largest bottleneck impacting the performance of the processor. The GPU masks the problem through two important functions: coalescing the access and stalling the warp. Coalescing the memory access simply means if a warp all access a continuous block of memory (divisible at 128, 64 and 32 bytes) the GPU will move the entire block to reduce the number of calls to the global memory. On CUDA 2.0 chips and higher this is improved further through the use of a L1 cache which has lines of 128-bytes. This means that repeated access to the same memory bank can be sped up through the automatic use of the L1 cache.

The next function is the stalling of the warp. Whilst this sounds extreme it happens all the time when a warp cannot process the next line due to either memory access or resources to be freed. When this happens the warp scheduler picks the next ready warp and executes it, thereby processing continues whilst waiting for global memory. This is most effective with a high number of warps and explains why the larger channel/frame size implementations had impressive speeds as memory access could be masked effectively.

As [1] states the more processing that is performed on the GPU shared or on-chip registers the better as these are designed for high throughput data transfers. The shared memory block does open the risk for bank conflicts where multiple threads try to access the same shared memory address.

The largest bottleneck for memory is the transfer between the CPU and GPU of data and in the small frame sizes is the cause of the majority of processing time. One channel of 32 frame size of the gain kernel takes approximately 600ms to compute the 409,600 samples. The total transfer of data takes 499ms, 83.16% of the total compute time.

The next problem is how the compiler might interpret code, especially perturbing to floating point operations. One common problem is shown in equation (12) below.

\[ a = a + (b \times c) \quad (12) \]
This is because the compiler has two methods of interpreting the results, as investigated by [18]. It can perform a multiplication of $b$ and $c$ and then adding the result to $a$ before assigning a memory operation. However this requires two operations and the danger is that there are two rounding stages, after the multiplication and then after the addition which will introduce error. The solution is to use MAD (Multiply Addition) operator or FMA (Fused Multiply Addition) on the GPU. This achieves greater accuracy than performing it serially and can increase performance as the operation counts reduce. The 1st implementation of the FIR filter shows the ideal implementation of the problem where the system is a continual repetition of multiplication and addition to calculate the next sample. The final consideration, and easiest to solve, when programming is for the block and thread sizes. As stated earlier, threads are grouped in blocks which are arranged in a grid. The blocks are sent to the stream multiprocessors, multiple blocks can coexist on a stream multiprocessor so long as there are enough local resources (registers, shared memory and maximum thread count). The previous two items, memory and operation efficiency, can be directly affected by the block and thread layout. As shown by all of the GPU results the small frame sizes are inefficient and generally never beat the CPU. This is because the small frame sizes are performed by having small thread counts. Frame size is the multiple of the block and thread count, for example a frame size of 128 can be one block of 128 threads or four blocks of thirty two threads. In either situation having multiple warps per SM (one warp is 32 threads) enables the GPU to mask certain actions, including the memory access.

As discussed earlier, when a warp stalls due to waiting for a resource or data the SM can switch to the next ready warp. If insufficient warps are loaded such that all are stalled the SM will wait until one is available. The ideal situation is that there are enough warps such that, during execution, all the SMs are fully utilized by warps.

Figure 7 shows the effect of one block, one channel performing a modified gain test without any PCIe transfers. As shown, each time the number of threads is doubled the processing time roughly halves. This shows that there are periods when the warps are stalled, most likely for memory read and writes, and if there is another warp available the SM can switch to execute. However a block is assigned to an SM and all threads in that block must execute on that SM, therefore on large cards with multiple SMs the overall usage is low. Figure 8 shows the performance gain of the same modified gain kernel when used with one or two blocks. As can be seen with all of the threads the performance increase is, on average, two times faster when using two blocks. The more blocks the better the scalability between different GPU's when deploying a product. However multiple blocks and threads all lead to increased latencies, despite the decreased processing time. Therefore the impressive returns for the GPU is when multiple channels are launched. This effectively increases the amount of data without increasing the latency. For the serial based CPU this is a problem as more data must be processed in the same space of time, but for the massively parallel GPU this helps to mask actions and provides ample warps.

**Figure 7** Total processing time, in ms, of gain on the GPU without memory transfers for various thread counts, 1 block 1 channel.

**Figure 8** Comparison of gain, no memory and one channel, at various thread counts with one thread block and two thread blocks.

Using the amplitude modulation, which requires more arithmetic operations, again with no PCIe transaction, at
1,024 frame size (4 blocks at 256 threads) on one channel the GPU reports that only 8.37% of the time there was an available warp to execute. This means that for the majority of the time the processor was waiting for a warp to become available. The highest stall reason was for dependency meaning data was not available for processing from the global memory. It also performs at a dismal 6 GFLOPS.

Low levels of available warps is not necessarily a bad thing if the warp is not stalled but processing or it is waiting for chip resources such as double precision and SFUs. This process however does not use either of these and therefore it is purely down to poor memory masking. Ideally there should be at least one available warp per scheduler per cycle meaning the next time the warp scheduler wants to execute, there is something to execute.

At 64 channels the card behaves in a saturated manner with over 53.36% of Issue Efficiency. Also it hits 5.74 warps available per cycle indicating that the card should be fairly saturated, indicated in processing at up to 65 GFLOPS. At the full 128 channels the card achieves 49.37% of Issue Efficiency at 4.09 warps available. It also achieves 74 GFLOPS of throughput. Therefore there is a limit where launching more channels may not increase the throughput of the card, however it will still, most likely, improve the performance against the CPU.

The final consideration is the use of resources and is, for some resources, very difficult to accurately manage. The obvious, user controlled resource is the shared memory and is useful across all CUDA cards. The shared memory allows inter-thread cooperation within a block and has a high bandwidth since it is located on-chip. This shared memory is a finite resource and can be a limiting factor on how many blocks are loaded onto a SM for execution, ultimately determining the number of warps available. For instance the GTX780M has a default 48KB of shared memory per SM and can load a total of 16 blocks. Therefore, to ensure that the shared memory is never a limiting factor no more than 3KB should be loaded, or 768 single precision numbers, into the cache. If 512 threads are loaded and you require a delay line, you may need to store 1,024 samples which would cause this to be a limiting factor. However at this size, other factors, including the maximum number of threads per warp, will be the limiting factor (2,048 on GTX780M or 4 blocks at 512).

For the majority of the time the shared memory is less useful than as a high speed store or if you do need interconnectivity between threads, such as in the FIR Kernel, or to provide your own cache where the threads all load the data frame into shared and continue to load it locally as it is required, useful on CUDA 1.x which does not have any caches. Therefore the cache and shared memory can perform a switch, whereby the L1 cache grows to 48KB and the shared memory shrinks to 16KB allowing for a better use of the cache and potentially increasing the performance.

5.1.1. Divergent Branches

Algorithms with a feedback element, such as the IIR filters, must operate serially as the output of the next sample requires the previous output sample. The CPU operates in this mode, therefore no real change is required. However for the GPU some threads must be disabled. Since CUDA runs an enhanced C language, traditional flow controls such as ‘switch’ and ‘if’ still work.

This project implements the feedback by only using the first thread of the channel to process the audio. This split from the multiprocessing mode is called divergence. If all the threads inside a warp are doing the same thing, even though the threads in a block were different, this would not be branch divergence. In the case of feedback systems there must be warp divergence as there can be only one thread to compute the information for the feedback loop, hence referred to earlier as forced serialization.

The only way to avoid this would be to create a kernel which would have the threads managing different channels. For instance in a 32 channel system, this part of the system could launch 32 threads, each processing a separate channel. Although branch divergence would be lost, the SM would be heavily underused (as only one warp would be initialised on one SM). This would need to be reviewed for smaller cards with less SMs as this disadvantage may reduce.

Creating the special kernel would also cause a problem with memory access. The current model, used in this project, launches the full thread size. This is so all the threads can transfer the data between the shared memory and global memory in a coalesced manner, gaining performance by reducing the total memory transfers. Shared memory is available to all threads of that block so the single processing thread can access the collected information. Even if a group of threads spend their time idle the overall advantage for memory throughput from coalescing will help.

The next performance improvement, as experienced most prevalently in the IIR 2nd order on the full GPU, is a return to a parallelised nature. The GTX780M can launch 4 warps per core simultaneously (a total of 64 warps per SM) which means that per SM a total of 64 serial operations could be launched.
5.1.2. GPU Focus

The underpinning failure for the most algorithms is the memory transfer. To resolve this an alternative method of GPU acceleration is proposed, to assume the GPU is the default processor. This would mean when audio data came in it was immediately transferred to the GPU, not moved back and forth each time, thereby only requiring one set of transfers for the system.

By assuming the GPU will operate all the processing the overall execution time should drop, especially where channels use multiple algorithms. Even in situations where the GPU was not expected to perform well, such as the IIR filters, performance gains should be made once combined as a network.

Whilst the GPU in its assistant role can provide ample improvements for multichannel processing, it is not the case for a single channel. In the dominant processor role, extra CUDA runtime functions could be engaged. This includes the ability to run multiple kernels simultaneously. The CUDA runtime automatically schedules and distributes the blocks to the SMs. Another feature is the ability for CUDA kernels to launch other kernels, thereby breaking items up further, although this is only available on GPUs of CUDA 3.5 or higher (nVidia, 2012).

Bundling these functions together into an Application Programme Interface (API) would ease the creation of far more complex and creative effects than have been implemented in this project.

The final point is on expansion. Whilst this project was tested on a laptop the same principles apply to desktop GPUs. Expansion in a desktop requires installing another CUDA ready GPU. The CUDA runtime allows the specifying of individual GPUs by the application and assessing GPU capabilities such as CUDA version, clock rates, RAM size and SM count (nVidia, 2013b). In an ideal system a low power GPU could run the simple data processing and then the data could be transferred to the more powerful GPU for the more intensive applications. Another method of operation would be to divide the channels up between identical, or similar performance, GPUs. For instance in a 64 channel session GPU A could process channels 1-32 and GPU B process 33-64, both of them operating on 32 channels. This would be the most efficient method of expansion as it would again require no memory transfers between processors.

This system of GPU only audio processing may become easier to implement through the use of GPU Direct technology (nVidia, 2013d) which allows multi-GPU systems to share their global memory with each other over the PCIe link, bypassing the CPU (as long as they are on the same PCIe Bus). It also expands this to allow other PCIe devices to directly access the GPU Ram store as well. The example is for cross network GPU computing but, with a properly designed driver interface, it may be possible for future PCIe audio interfaces to directly drop frames onto the GPU.

6. CONCLUSION

On the given test platform the CUDA language provides an acceptable alternative to traditional CPU processing. Whilst it is not faster for a single channel, due to the low data rate, the GPU still maintains real-time performance. The CUDA environment does prove suitable for multichannel processing achieving immense speed improvements whilst retaining accuracy. It also allows for great flexibility and portability between different nVidia processors, a key factor in its design allowing for easy cross-device programming.

As expected items which are forced to process serially due to their inclusion of a feedback path were the worst performers, however processors with high numbers of warp executors could parallelise the multi-channel implementations and therefore can still outperform the CPU.

The largest bottleneck, indicated in previous work and confirmed in this project, was the memory transfer to and from the GPU. Although the physical transfer was saturated, the time to prepare for the transfer added large overheads, lowering the overall transfer speed. Finally careful attention must be paid to the distribution of resources and that sometimes larger thread counts can actually decrease performance. This includes careful attention to shared resources including shared memory and SFU allocations.

Using GPU’s for real-time audio processing is possible and recommended in high data pressure environments so long as these crucial points are considered.

7. FURTHER RESEARCH

There are three avenues to expand upon this project. The first is to test on other hardware, including generational differences and desktop environments. The second is to design a system where the GPU is the default processor, such that the GPU performs entire system processing, minimising the memory transfers. A final field would be to expand the scope to include other audio processing including offline processing (data processing and data compression algorithms), synthesis and externally adaptive / feature based processing techniques.
8. **ACKNOWLEDGEMENTS**

This project was supported by Yonghao Wang and Birmingham City University as an undergraduate dissertation for Sound Engineering and Production, BSc.

9. **REFERENCES**


https://developer.nvidia.com/sites/default/files/aka
mai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf
[Accessed 4 February 2014].