

Assessment of the suitability of GPUs for Audio Feature Extractions

Case study: Multiple FIR filtering.

Sofia Dimoudi
Oxford e-Research Centre
University of Oxford
7 Keble Road
Oxford OX1 3QG

August 31, 2016

1 Introduction

This document reports on the experience of using Graphics Processing Units (GPUs) to accelerate the pitch-chroma audio feature extraction process, after the paper [1], in the context of the FAST project,¹ an interdisciplinary research project between Queen Mary University of London, University of Nottingham and University of Oxford, that interconnects multiple aspects of technology in music. The Oxford e-Research Centre (OeRC) is very active in GPU computing for science applications, and is part of the NVIDIA Centre of Excellence at Oxford University.

The main objectives of this work are to assess the suitability of GPU devices for this type of audio processing in a real-time setting, to transfer knowledge on methods from radio astronomy, which is ongoing work currently at the OeRC, to audio processing, and to provide a software and guidance that will serve as a starting point for any possible future work.

The work is based on an existing implementation for the VAMP audio processing plugin system [2], included in the TIPIC (Timbre-Invariant Pitch Chroma) plugin [3]. Part of the TIPIC algorithms, an FIR version of a filterbank is used as a test case and implemented in basic form in NVIDIA CUDA [4] and OpenMP. These stand alone implementations, including the basic serial implementation adapted from TIPIC, are then compared on functional parameters that affect the computational speed. An FIR filterbank on GPUs has been previously implemented for radio astronomy at the OeRC [5], and ideas derived from that work as well as advice from the authors have been used in this project.

The remainder of the document is organised as follows: Section 2 gives a brief introduction to GPU computing; section 3 gives an overview of the application and describes the software; results are presented in section 4; finally in section 5 I draw my conclusions from this experience, and suggest ideas and directions for future work.

¹This work is supported by: Fusing Semantic and Audio Technologies for Intelligent Music Production and Consumption funded by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant number EP/L019981/1, a collaboration between Queen Mary University of London, University of Nottingham and University of Oxford.

2 GPU computing with CUDA

GPUs are massively parallel processors with very high memory bandwidth, containing hundreds to thousands of floating point computing cores for data-parallel instruction execution. NVIDIA GPUs with CUDA [4] are so far the most widely used platform that is proven to produce highly accelerated applications in many scientific and engineering subjects, including signal processing and astronomy.

The GPU consists of groups of processing cores (multiprocessors) with shared and L1 cache, and register file. It has an L2 cache which is shared across all multiprocessors, and a high bandwidth on-board DRAM. The CUDA hardware model employs the SIMT architecture (Single-Instruction Multiple-Thread) in which a single instruction is executed by multiple threads on different data. Threads are organised in groups controlled by a scheduler and threads within a group execute simultaneously.

CUDA programs execute functions called kernels that issue instructions to many parallel threads. The programming model organises threads in thread blocks, and a grid of thread blocks. Threads and blocks have their ID within the thread block and grid respectively. A thread within a block has its own registers and private local memory. Threads within a block have common access to a shared memory per block. Multiple grids or kernels have access to a global memory space that resides in the on-board DRAM. The model is illustrated in figure 1. More details on the CUDA programming and hardware model can be found in the CUDA programming guide [6].

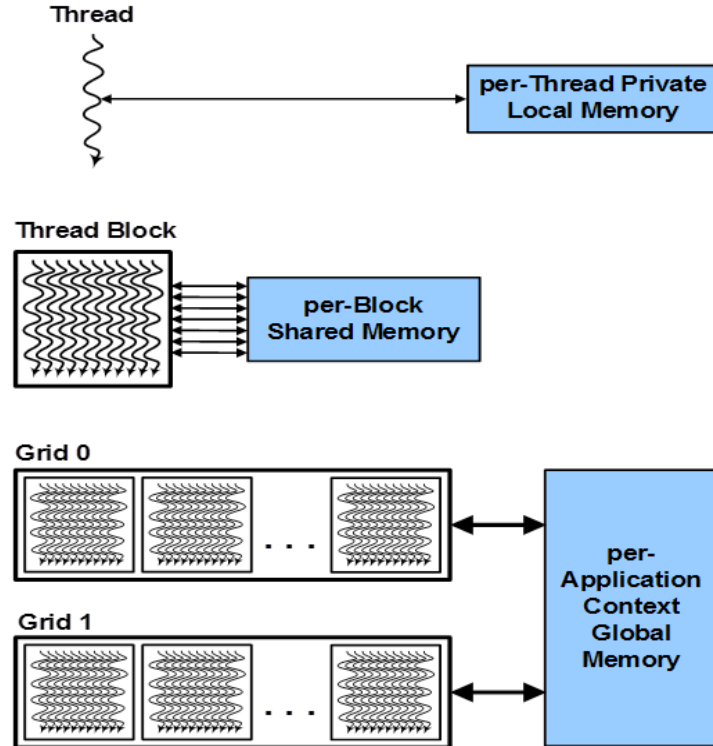


Figure 1: CUDA programming model (Image credit: NVIDIA)

Multiple CUDA kernels can be executed concurrently with CUDA streams. Within a stream, instructions are executed in order, while different streams can be executed out

of order and asynchronously. Streams are used to run small kernels, and kernels and memory transfers concurrently on the same device.

3 The Test Case

One of the most computationally demanding tasks of the pitch-chroma feature extraction process is the application of multiple IIR (Infinite Impulse Response) filters on the input signal. An IIR filter is represented by the difference equation (1)

$$y[n] = - \sum_{k=1}^N \alpha_k y[n-k] + \sum_{k=0}^M \beta_k x[n-k], \quad (1)$$

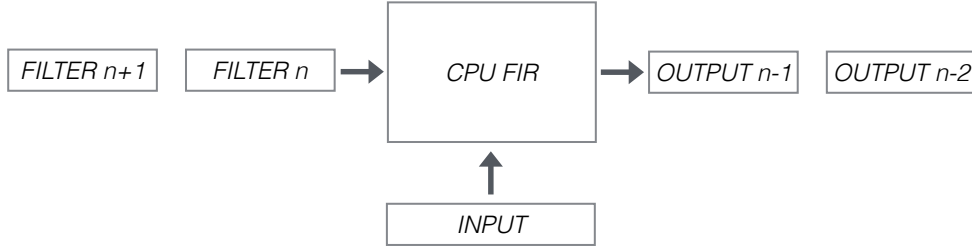
where $x[n]$ is the discrete input signal, $y[n]$ is the output, and a_k, b_k are the feedback and feed forward filter coefficients respectively. If we set $N = 0$, then the filter becomes an FIR (Finite Impulse Response) filter. The recursive nature of the IIR filter makes its computation in parallel a non-trivial task, particularly for the SIMT execution model of the GPU architecture, and also less likely to provide significant speedups. Since the IIR filter is realised with the addition of a non-recursive feed forward term and a recursive feedback term, we chose to demonstrate the use of GPUs on the FIR term alone. This choice allows us to provide a general example of GPU computation for a typical building block of many audio processing applications.

3.1 The example Code

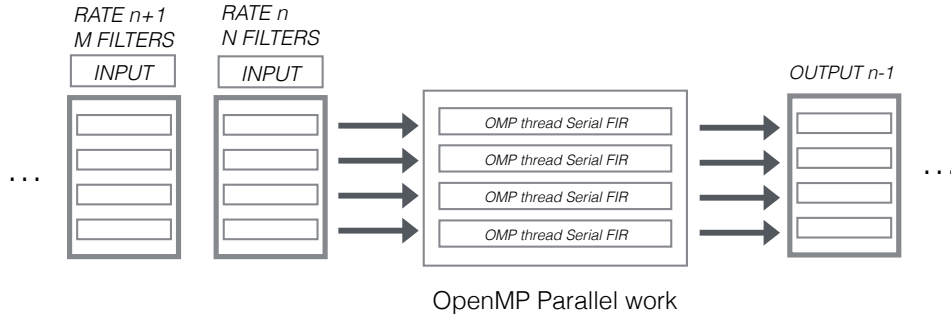
The test code is written in C and CUDA, but the CUDA functions can be called from any C++ code by including one header file. The main program is using standard C but is compiled with C++ to ensure compatibility with the C++ compiler.

The program tests a serial CPU FIR function, adapted from the Taptic plugin, which is also used with OpenMP, and a GPU function that executes groups of multiple filters in parallel and concurrently on the device. Figure 2 shows a schematic representation of the way a set of multiple filters with a set of input sampling rates are executed with each of the serial, OpenMP, and GPU implementations.

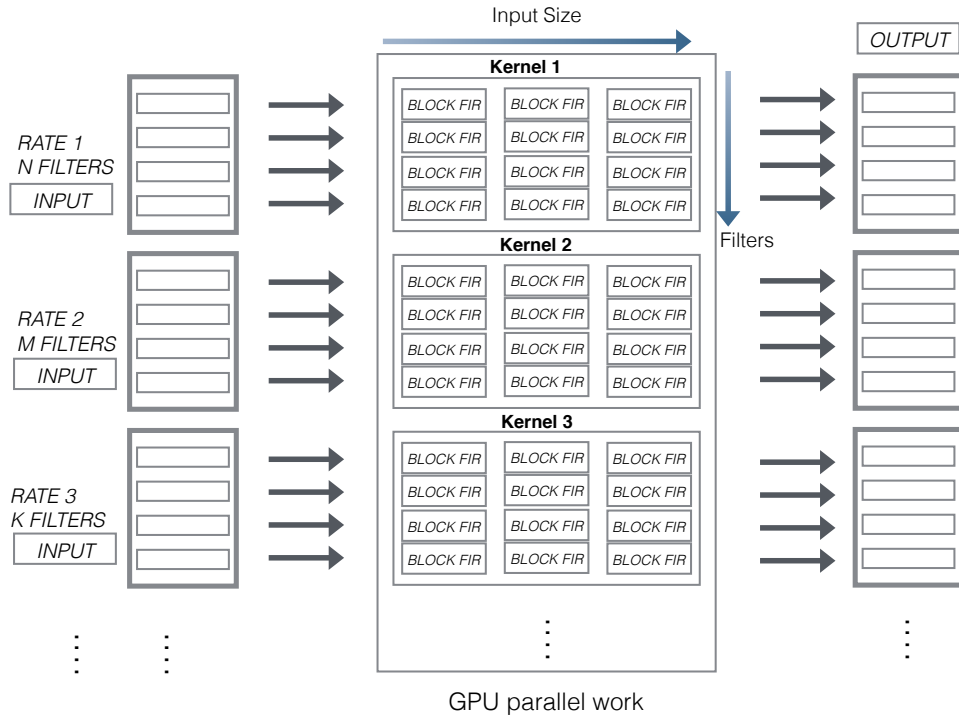
In figure 2c, kernels (1,2,3 ...) are executed concurrently using CUDA streams, allowing GPU threads to be utilised at the same time across the whole range of filters with different grid resolutions. Data are transferred to and from the GPU through the PCIe bus. This is an overhead to the execution time, however, the user can employ additional CUDA streams to overlap computations and data transfers, and effectively hide the memory transfers. The code is not currently configured to overlap successive transfers and computations, because such a configuration may depend on application requirements for the time domain processing, and thus not be representative, while it would add additional complexity to the code and make it less generic. Asynchronous data transfers are used in the current CUDA streams configuration, for the transfer and computation of multiple sampling rates of a single input block. This requires that the data are allocated to pinned memory on the host. Because our application is using relatively small blocks, this is not expected to cause a problem.



(a) Serial execution on a single CPU.



(b) Parallel execution on a multiple CPU cores with OpenMP.



(c) Parallel execution on a GPU.

Figure 2: Execution configuration for the single CPU, OpenMP, and GPU implementations.

Each block on each CUDA kernel applies a single filter to all or part of the signal block. Shared memory is used to store the coefficients and the signal, and each thread produces one output element by iterating serially over the pre-loaded filter coefficients. The memory reads and writes from the global device memory are done in a data-parallel fashion, with coalesced memory transfers. This allows for efficient use of the device bandwidth, while exploiting memory reuse from the shared on-chip cache for fast computations. One limitation of this scheme is that the size of the thread blocks must be larger than the size of the filter. The filter size is best to be pre-defined to allow compiler optimisations such as loop unrolling.

The OpenMP implementation is a simple case of a parallel for directive on each set of equally sampled filters. This is not optimal, but it allows us to see how the parallelism is exposed on a multi-core CPU for an equally sized problem. The user can experiment in expanding the OpenMP code for optimal performance for multiple sampling rates.

The CPU codes in the example are compiled with the Intel C++ compiler for an Intel processor, and the flags `-m64 -O3 -xhost` were used to provide automatic optimisation for loops and the CPU architecture instruction sets.

4 Performance Measurements

4.1 Hardware and Software Setup

For the CPU tests, we used an Intel Xeon E5-2650 with 20 cores in total, hosted on a server of the Advanced Research Computing (ARC) facility of the Oxford University. The CPU code was compiled with the Intel C++ compiler, using OpenMP and architecture specific optimisation. These optimisations are appropriate for the unrolling of compute intensive loops, such as in our case. The

The GPU used for the tests is an NVIDIA GeForce GTX Titan X card, which is of Maxwell generation [7] with 3072 computing cores and maximum memory bandwidth 336 GB/s, and the code was compiled with CUDA 7.5.

All tests were done with 32-bit floating point precision, which is preferable for good performance on GPUs.

The GPU, serial CPU, and OpenMP codes were tested against varying input block size, filter size, number of filters, and number of sampling rates. The original application currently uses 3 different sampling rates, to which the input signal is downsampled prior to the filterbank operation. These decimate the signal by a factor of 2, 10 and 50. The filters contain 11 coefficients each. The actual total number of filters is 88, but we are experimenting with sizes that are multiples of 20. A range of input block sizes up to 32768 points was considered reasonable for audio block durations of up to the order of a ~ 100 ms with a maximum sampling rate of 192 KHz. The filter size was varied up to 50 times the initial filter length of 11. The filter length multipliers include the equivalent to the decimation factors used in downsampling, but also extra numbers to cover the intermediate space between them so as to describe the response of performance to size variations. The number of filters was varied up to 240 filters, over double that of the original application, so as to allow us to assume a case of more than one input channel in parallel. Finally, a range of 10 different input sizes were tried, each representing a down sampled signal to a particular sampling rate. The decimation factors again included the ones from the original application, and intermediate numbers were added. The test parameters are summarised in table 1.

Table 1: Test parameters.

	TIPIC parameters	Test range
Input size	user defined	1024 - 32768
Filter length	11	11, 22, 55, 110, 220, 330, 440, 550
# of filters	88	20 - 240
# of sampling rates	3	1 - 10
sampling rate decimation	2, 10, 50	2, 10, 50, 4, 16, 32, 8, 24, 36, 42

Due to time restrictions, it hasn't been possible to measure PCIe data transfer overhead systematically, but individual trials have shown timings of up to $\sim 50\%$ of the total execution time, when large numbers of filters and longer input sizes were used. The user should consider making these measurements and using CUDA streams to overlap data transfers with computations, if the overhead exceeds their application's requirements. One can also estimate the possible overhead theoretically, by dividing the number of bytes transferred by the PCIe theoretical transfer rate. An example of the theoretical transfer duration for increasing numbers of filters is listed in table 2.

Table 2: Theoretical data transfer duration based on a PCIe 3.0 x16 maximum bandwidth of 16 GB/s, for an input size 32768.

Number of filters	transfer duration (ms)
20	0.15
60	0.46
120	0.92
180	1.37
240	1.83

4.2 Results and Comparisons

Figures 3a and 3b show the timing and speedup between CPU, GPU and OpenMP execution for a varying input block size, up to 32768 points. In this test, the input size across all filters was kept constant, which represents a constant sampling rate for all filters. This was done in order to assess the timing response to block size in an absolute manner. The filter size was kept constant to 11 points. Three different numbers of filters were used, to reflect the different capacities of a single sampling rate.

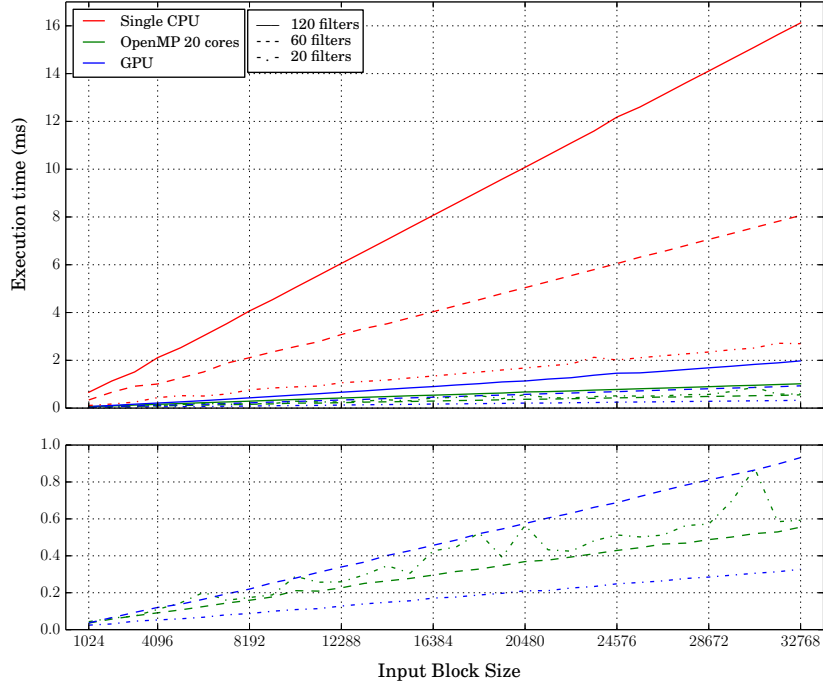
The execution time in all cases appears to grow linearly with size, except for the case with the smallest number of filters using OpenMP, which shows a variability, but this can be attributed to the effect of load balancing for a small amount of work (20 filters, 1 filter per OpenMP thread). The OpenMP code performs better than the GPU in all but this case. This is better shown in figure 3b where the speedup of each parallel code can be seen against the single CPU execution. In this plot we also observe that for a higher number of filters the GPU speedup appears to drop slightly with input size, while the opposite happens with OpenMP. This is indicative of the PCIe device to host transfer

overhead, which grows with the amount of data produced in the output. However, the parallel execution is below 2 ms, so this delay may not be significant.

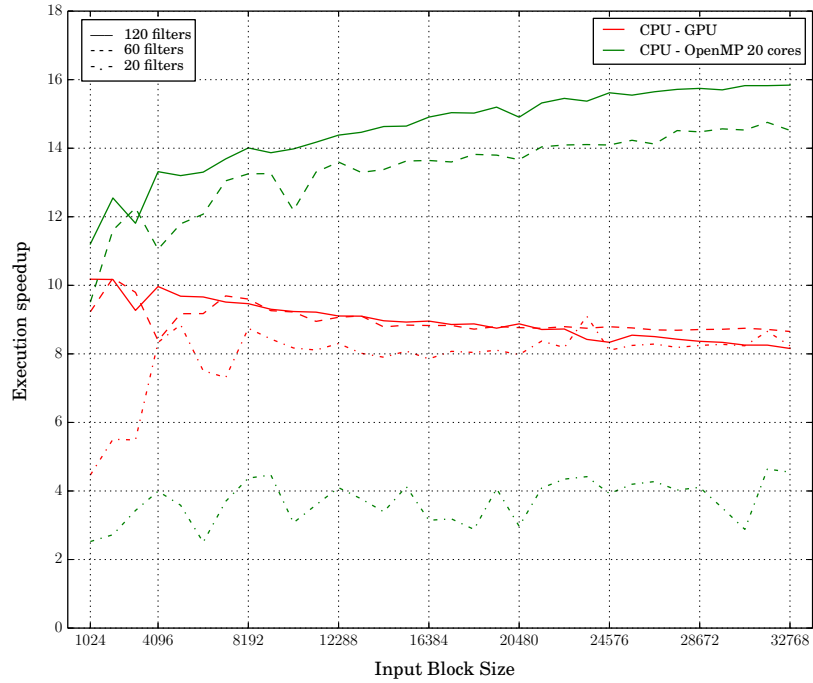
Figures 4a and 4b show the timing and speedup for varying filter size. In this case the input size was kept constant, as well as the sampling, and the response to filter size for 120 filters was examined. The GPU appears to do better in all but the smallest filter size. This indicates the limitation of CPU parallelism, as longer loops are processed. For the GPU, the computing power is exposed with increasing compute load and constant amount of output data transfers, which shows that data transfers is a limiting factor. Another factor on the GPU may be the size of the parallel thread blocks, that depends on the filter size. Tuning for this purpose has not been possible due to the limitation of the dependance on filter size in the current configuration, but it is possible that we can obtain performance gains simply by tuning the CUDA thread blocks. The GPU execution is also more stable across filter sizes, and grows linearly.

To check the response against varying number of filters (figures 5a and 5b) we used a small input length of 4096 points, again at a constant rate. Based on the findings of the previous test, it makes sense to see the behaviour for different filter sizes as well, and the code was run for three different filter sizes. The GPU is always faster for less than 40 filters, and for longer filters it is faster and increases at a very small rate. As seen in the previous tests, increasing filter numbers affect the transfer overhead. An increase in input size will simply apply a linear factor to the execution of both CPU and GPU.

For the number of sampling rates it was not possible to do exhaustive tests, but a fixed set of 10 input size dividers was used, based on the ones currently used in the Tpic plugin. The number of filters per rate were distributed equally. To represent downsampling, every group of filters that belongs to a specific rate, works on an input signal whose size is divided with a predetermined number. This means that compared to the previous cases where the size was fixed across filters, this execution generally contains less data, but with different sizes. Figures 6a and 6b show that, while the OpenMP implementation has a drop in performance as the number of rates increases, the GPU fares better with more input rates. This is a consequence of the way parallelism is implemented for OpenMP, which only acts in parallel across each group of filters, and executes the groups one after the other. As this is a basic OpenMP implementation, this result is not conclusive, and could improve if the threads are distributed across the whole range of filters, as indicated both by the previous results, and from the timings on the first four rates on figure 6a. For the GPU this result demonstrates how we can keep all the work executing in parallel irrespective of the grid sizes by using asynchronous execution for smaller loads, as is the case for each of multiple rates.

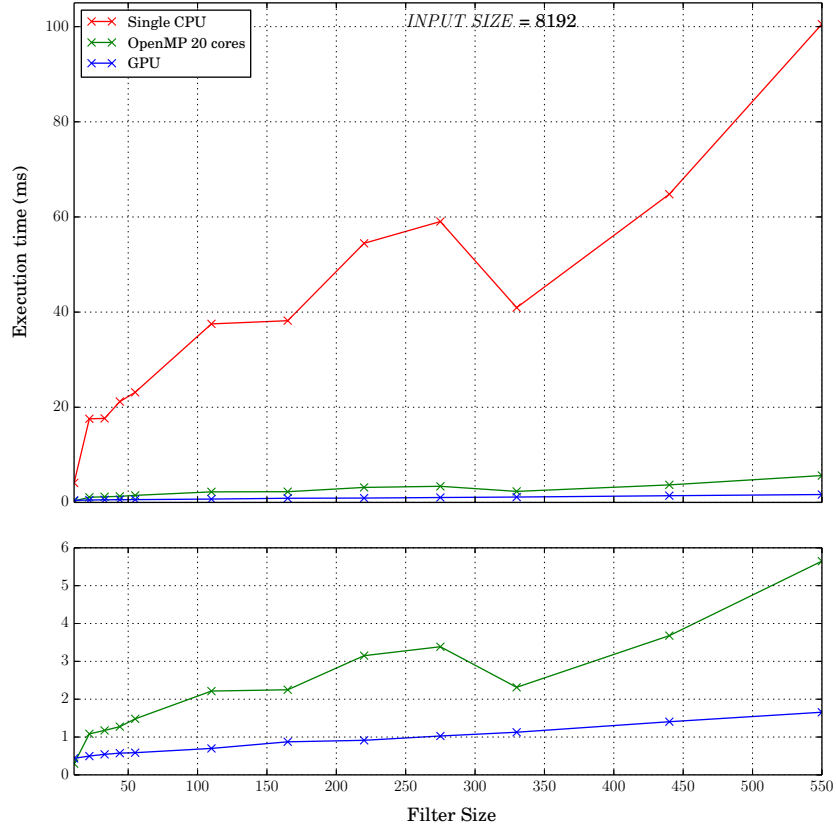


(a) Execution times of multiple filters against input block size for 3 different numbers of filters. The bottom plot ‘zooms’ into the OpenMP and GPU timings.

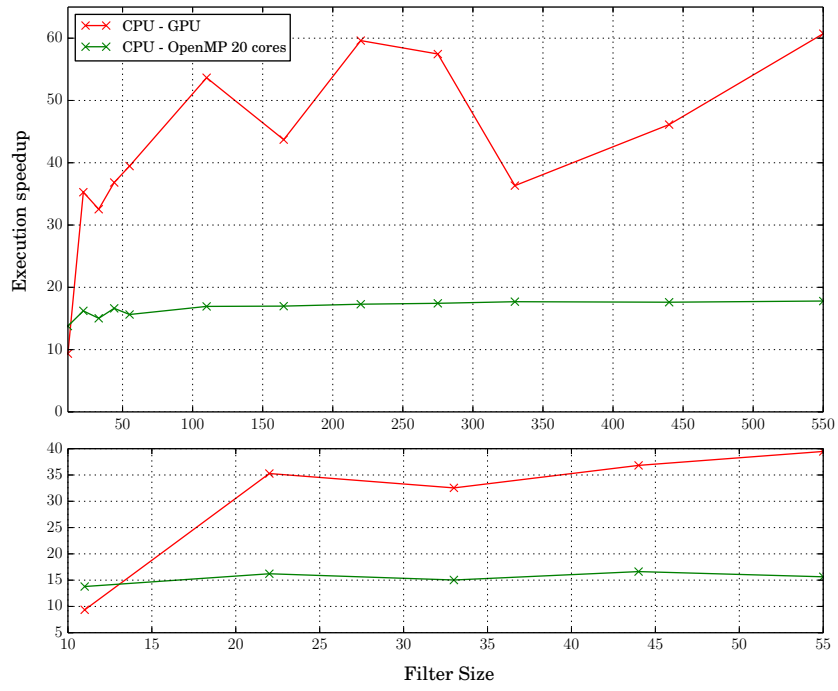


(b) Speedups between devices for the execution of multiple filters against input block size for 3 different numbers of filters.

Figure 3: Comparisons against input block size.

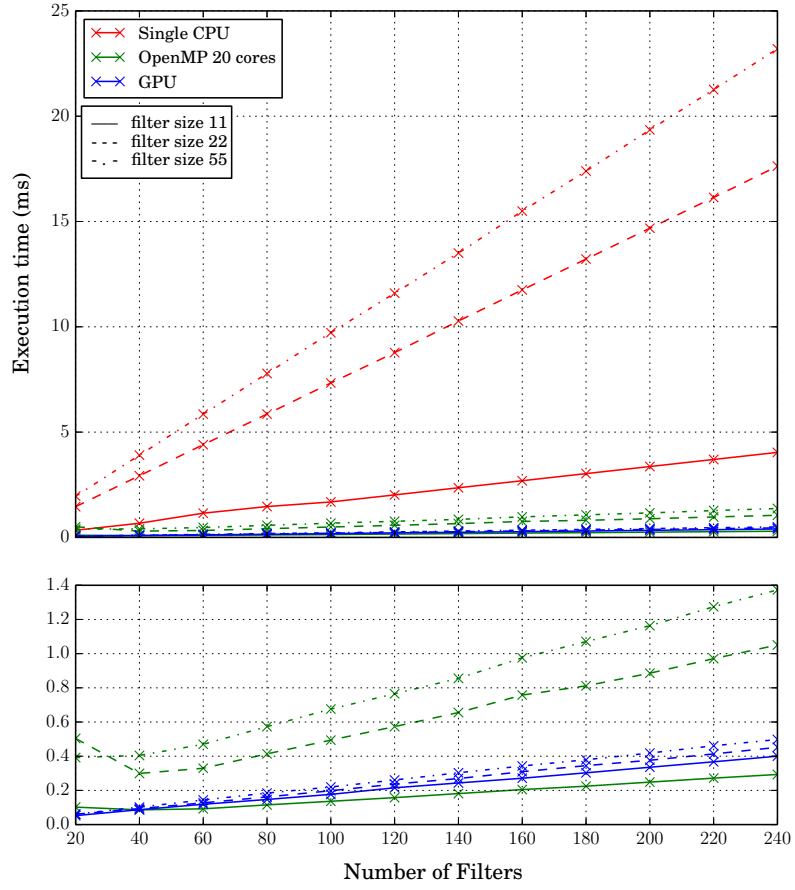


(a) Execution times of 120 filters against filter length. The bottom plot 'zooms' into the OpenMP and GPU timings.

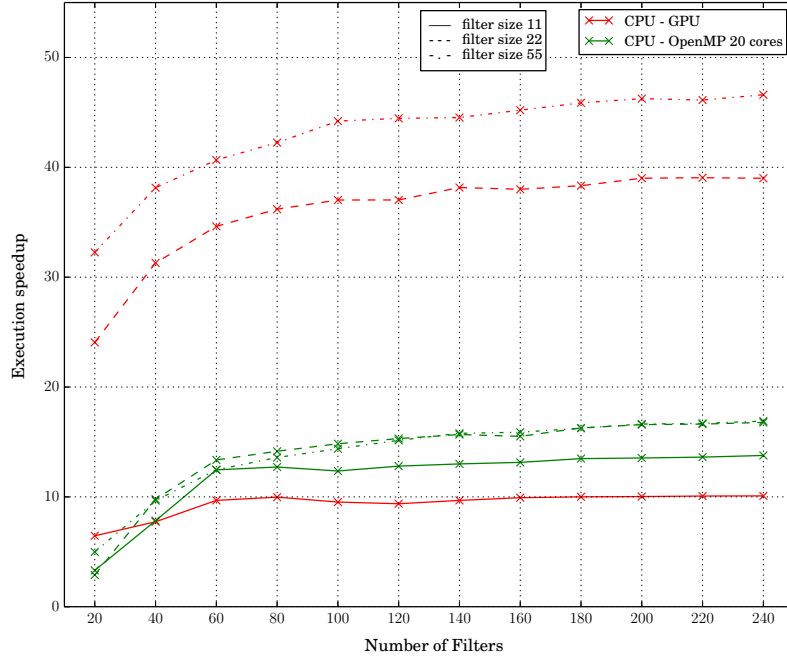


(b) Speedups between devices for the execution of 120 filters against filter length. The bottom plot stretches to filter lengths up to 55

Figure 4: Comparisons against filter length.

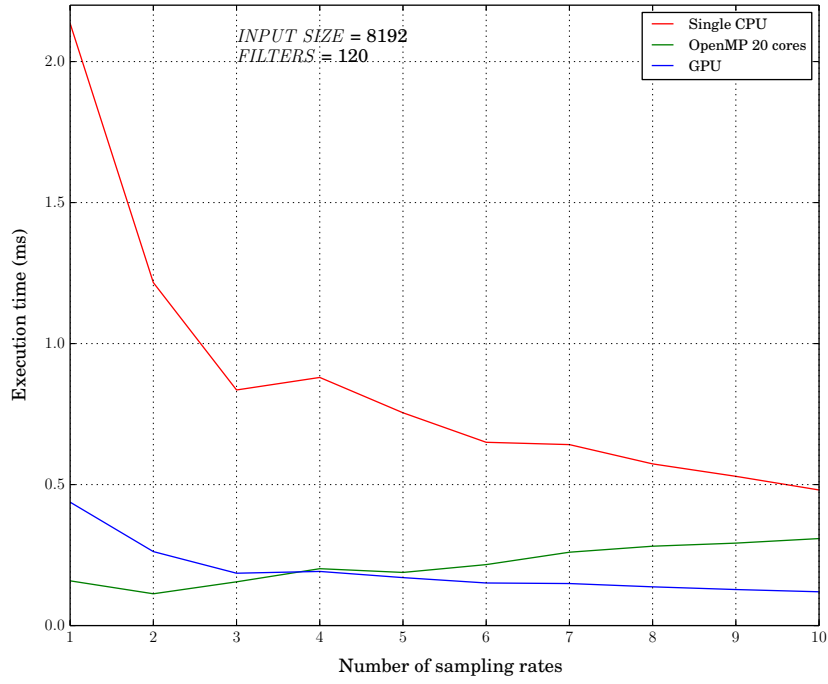


(a) Execution times of multiple filters against number of filters.

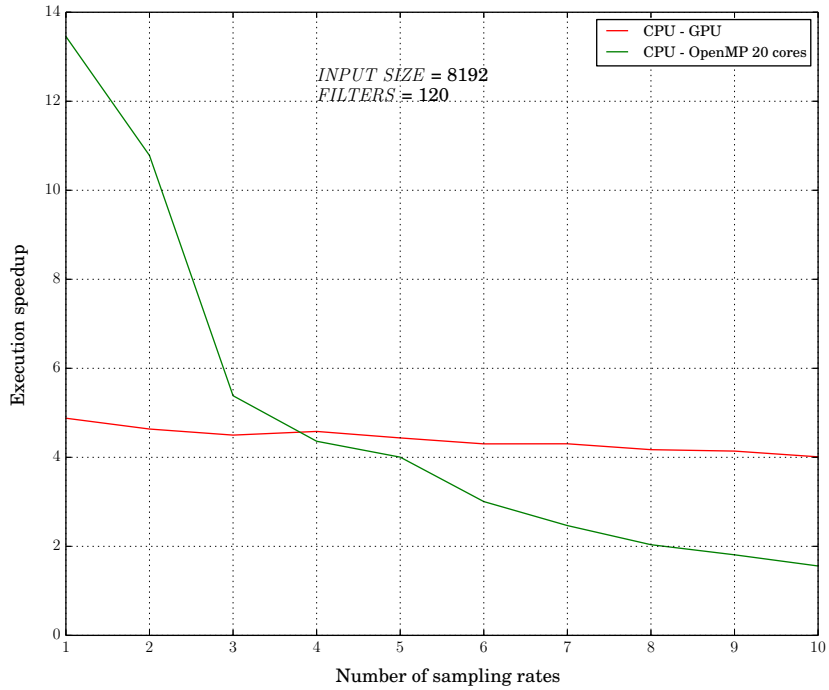


(b) Speedups between devices for the execution of multiple filters against number of filters.

Figure 5: Comparisons against number of filters.



(a) Execution times of multiple filters against number of sampling rates.



(b) Speedups between devices for the execution of multiple filters against number of sampling rates.

Figure 6: Comparisons against multiple sampling rates.

5 Conclusions and suggested further work

The work conducted represents a very basic parallel GPU implementation and is intended to give a general assessment of the parallel behaviour of the filtering application, in order to indicate application areas that would benefit from using GPUs. Both GPU and CPU OpenMP implementations have room for optimisation, but the type of optimisation will depend on the usage scenario.

In general, the tests have shown that both CPU and GPU parallel codes provide large speedups over the serial CPU with minimal programming effort, and that the execution times delivered are at the range of a few milliseconds across the parameter space examined. The OpenMP code performs at similar timings to the GPU except the case of longer filter lengths, where the GPU appears to have a good advantage. This is an interesting finding because a longer filter length could be useful for example to improve frequency band separation at a minimal computational cost.

The CPU used is a server class processor, and it is expected that for devices of lower specifications the performance will be slower. The results show that the OpenMP speedup with 20 threads is nearly stable at around 15 times for numbers of filters over 60 and a fixed filter size. A less powerful CPU may scale this result to the number of cores, but other factors, such as clock frequency, cache and I/O specification and features could affect performance. The GPU currently suffers from PCIe memory transfers, but hiding these transfers can be easily handled with software.

There is potential for improvement in performance on GPUs if the different parts of the chroma feature extraction process are run on the GPU. This appears possible for the parts of rate conversion and short term energy calculation at least, since they also contain windowing operations. The benefit of this over a CPU would be the increased on-board memory bandwidth that the GPU offers, in particular if caches are used as in this example. For the part of the IIR recursive term, it is possible to employ techniques that have been used before in GPU computing, such as collaborative thread patterns [8] and block ordering [9] to enable parallel computation of sequential tasks, but one could also consider a mathematical representation that can derive the output of the high order difference equation from the N th previous output element instead of the previous element, such as to allow a minimal parallel operation within a thread block. An FIR approximation could also be investigated as an option to remove slowing down of the application because of the recursive part.

Although the GPU and OpenMP both seem to perform similarly and sufficiently fast, the benefits of using each architecture depend on the application scenario. If high performance CPUs or multiple processors can be used, the OpenMP implementation could prove a good choice because it does not require much additional programming effort, although the equipment cost may be higher. In the cases where processing of multiple signals is required, and the latency requirements are more strict, such as in live performances for example, the GPU may be preferable, because the computational scheme it uses is scalable, and it can easily handle more channels on a single device. If data transfers are overlapped on the GPU, based on the theoretical figures of table 2 it is likely that the gain will be higher with multiple channels per GPU, when compared to workstation CPUs. Furthermore, it can be more cost effective and energy efficient to use multiple GPUs to handle multiple signals on a single machine, thus applying an additional level of parallelism, than to do so with server class CPUs such as the one used in these tests. Embedded systems using System-on-Chip modules such as the NVIDIA Tegra and Jetson [10] can enable savings in energy with similar performance.

A potential usage of the application for both CPU and GPU, particularly for live

performances, would fit a device such as the IBM microservers [11] currently under research within the DOME project [12, 13], which also collaborates with the FAST project on computational musicology. The microservers are System on Chip designs that integrate a server motherboard on a single microchip, enabling multiple computing nodes in a small box with high computational performance and low cost and energy consumption. The current version uses PowerPC processor cores, but in the future they may include embedded GPU designs. A further investigation on the capabilities of the current microserver and the existing embedded GPU cards for the chroma feature extraction would be very useful in exploring future directions for real-time processing for the FAST project.

The final conclusion is that the application has gains from parallel computing, and, within a limited parameter space, both platforms perform similarly. Future research is encouraged with GPUs and CPUs to include more parts of the chroma extraction processing on GPUs, for applications with multiple signal inputs, as well as comparisons with lower end CPUs. In particular for live performances and very low latency, there is a good potential for useful research output in testing the DOME microserver and embedded GPU systems.

References

- [1] Meinard Mller and Sebastian Ewert. Towards timbre-invariant audio features for harmony-based music. *IEEE Transactions on Audio, Speech, and Language Processing*, 18(3):649–662, 2010.
- [2] C. Cannam. Vamp. <https://code.soundsoftware.ac.uk/projects/vamp>.
- [3] C. Cannam and S. Ewert. Timbre-Invariant Pitch Chroma. <https://code.soundsoftware.ac.uk/projects/tipic>.
- [4] NVIDIA. CUDA Zone. <https://developer.nvidia.com/cuda-zone>, .
- [5] K. Adámek, J. Novotný, and W. Armour. A polyphase filter for many-core architectures. *Astronomy and Computing*, 16:1 – 16, 2016. ISSN 2213-1337.
- [6] NVIDIA. CUDA Programming Guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide>, .
- [7] NVIDIA. NVIDIA Maxwell Architecture. <https://developer.nvidia.com/maxwell-compute-architecture>, .
- [8] Tony Scudiero. Memory Bandwidth Bootcamp: Collaborative Access Patterns . <http://on-demand-gtc.gputechconf.com/gtc-quicklink/7pklu>.
- [9] M. Giles. Oxford University Course on CUDA Programming on NVIDIA GPUs, Lecture 4. <https://people.maths.ox.ac.uk/gilesm/cuda/>.
- [10] NVIDIA. NVIDIA Embedded Computing. <https://developer.nvidia.com/embedded-computing>, .
- [11] IBM Research. IBM and ASTRON 64-bit Microserver. <http://www.research.ibm.com/labs/zurich/microserver>.
- [12] DOME project homepage. ASTRON and IBM Center for Exascale Technology. <http://www.dome-exascale.nl>.

[13] Wikipedia entry. Dome Project. https://en.wikipedia.org/wiki/DOME_project.