MEASURING THE PERFORMANCE OF REALTIME DSP USING PURE DATA AND GPU

André Jucovsky Bianchi[1], *Marcelo Queiroz*[1]
Computer Science Department, University of São Paulo, Brazil
{ajb,mqz}@ime.usp.br

ABSTRACT

In order to achieve greater amounts of computation while lowering the cost of artistic and scientific projects that rely on realtime digital signal processing techniques, it is interesting to study the performance of commodity parallel processing GPU cards coupled with commonly used software for realtime DSP. In this article, we describe the measurement of data roundtrip time using the Pure Data environment to outsource computation to GPU cards. We analyze memory transfer times to/from GPU and compare a pure FFT roundtrip with a full Phase Vocoder analysis/synthesis roundtrip for several different DSP block sizes. With this, we can establish the maximum DSP block sizes for which each task is feasible in realtime by using different GPU card models.

1. INTRODUCTION

The highly parallel nature of many Digital Signal Processing (DSP) techniques makes the use of commodity hardware for parallel processing specially useful for realtime scenarios of artistic performances, small technical applications and prototyping. To make better use of parallel processing devices it is interesting to study if different combinations of hardware and software can meet specified criteria.

Widely used by digital artists, Pure Data[2] (Pd) is a realtime DSP software licensed under free software terms, which can easily be extended and combined with control hardware through wired or wireless interfaces. Also, Pd is able to handle audio and video signals making it possible to build and control arbitrary DSP algorithms.

To enhance Pd with parallel processing capabilities, one of the lowest cost solutions nowadays is to attach to it a Graphics Processing Unit (GPU) card, to which Pd will then be able to transfer data back and forth and request (parallel) computation to be performed over it. If all this can be done in a time period of less than one DSP cycle (the period for one block of samples that is act upon by the DSP software at a time), then it may in fact be worth it to combine Pd and GPU for realtime DSP performances. This work focuses on performance measurements of common parallel tasks, such as memory transfer and kernel execution times, and uses Pd extensible design to implement interaction with the GPU using C and CUDA C code compiled as shared libraries.

The use of GPU for realtime audio signal processing has been addressed in recent work [4, 2, 3, 5, 6]. For instance, by measuring the performance of the GPU against that of a commodity CPU, Tsingos et. al. showed that for several applications it is possible to achieve dramatic speedups (factors of 5 to 100) [6]. A similar approach to ours was carried by Savioja et. al., who analyzed the GPU performance on additive synthesis, FFT (and convolution in frequency domain) and convolution in time domain [5]. Our work differs from these in two ways. First, we make use of Pure Data as the software environment for interaction with the GPU API, thus providing a look into the use of parallelism on a widely adopted realtime computer music platform. Additionally, we provide a fine-grained measurement of memory transfer times, so we can compare these with actual processing time.

Our computation outsourcing model assumes that the GPU will be used by Pd in a synchronous manner. At every DSP cycle, Pd will contact the GPU, transfer a portion of memory to it, call kernel functions over that portion of data, wait for these kernel calls to end, and then transfer data back to the host's main memory. Our aim is to perform the following measurements, in order to establish the feasibility of using a GPU-aided environment with Pd on realtime performances:

- **Memory transfer time.** Since a GPU only processes data that reside on its own memory, memory transfer can represent a bottleneck for parallel applications that use GPU: generally, the fewer the amount of data transfers the better.
- **Kernel execution time.** This is the total time used by all instructions performed on the GPU, *after* memory is transferred from the host and *before* memory is transferred back to it.
- **Full roundtrip time.** This is the total time taken to transfer data from host to device, operate on that data, and then transfer it back to the host. This is the single most important value to compare with the DSP cycle period, in order to establish the feasibility of using the GPU in realtime environments.

On the following section, we describe the setup used for performing time measurements for these tasks.

[1] This work has been supported by the funding agencies CAPES and FAPESP (grant 2008/08632-8).

[2] http://www.puredata.info/

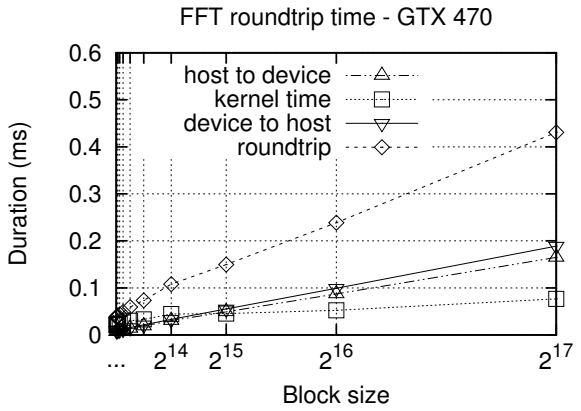

Figure 1. Memory transfer and kernel times for FFT on Geforce GTX 470.

2. METHODS

In order to measure kernel and memory transfer time using Pd and GPU, we set up a Pd external that communicates with the GPU and keeps track of elapsed time between operations. The external behaves as a normal Pd object that receives input signals and produces output signals, but actual DSP is delegated to the GPU.

Our test environment is an `Intel(R) Core(TM) i7 CPU 920 @2.67GHz` with 8 cores and 6 GB RAM, running `Ubuntu GNU/Linux 11.04` with linux kernel version `2.6.28-13-generic`, and equipped with two models of NVIDIA GPU cards: Geforce GTX 275 (240 cores, 896 MB RAM, 127.0 GB/s memory bandwidth) and Geforce GTX 470 (448 cores, 1280 RAM, 133.9 GB/s).

For implementing the Pd external we used standard C and CUDA C[3]. As a basic DSP algorithm implementation, we make use of CUFFT[4], NVIDIA's implementation of the FFT algorithm which is compatible with the widely used FFTW collection of C routines[5].

2.1. Implementations

To be able to evaluate the performance of our setup for realtime DSP, we started with an implementation of a pure FFT external that transfers data into the GPU, runs the FFT over it and then transfers data back to the host, thus providing the host with a frequency domain description of the signal block. The results for different block sizes on the GTX 470 can be seen in Figure 1, and will be discussed in the next section.

In order to estimate how much more computation we may export to the GPU while preserving realtime operation, we implemented a full Phase Vocoder [1] analysis and synthesis engine. This also allows for a comparison of the time spent by the GPU in regular user code versus device-specific professionally-engineered library code.

[3]http://developer.nvidia.com/cuda-toolkit
[4]http://developer.nvidia.com/cufft
[5]http://www.fftw.org/

An implementation of the Phase Vocoder for the GPU can use parallelism in two ways. First, it can estimate the instantaneous amplitude and frequency for each oscillator by making use of the parallel FFT as we just saw. After that, as the result for each synthesized output sample does not depend on the calculation of other sample values, the PV can perform one additive synthesis for each output sample of a DSP block in parallel. Thus, an implementation of the Phase Vocoder on the GPU uses the same amount of data transfer between host and device as the pure FFT algorithm, but comprises more kernel calls and more computation inside each kernel. The results for memory transfer and kernel time of our Phase Vocoder tests for different block sizes can be seen in Figure2, and will also be discussed in the next section.

2.2. Tests

We ran the FFT and PV algorithms for a period equal to 100 DSP blocks, for block sizes of 2^i with $6 \leq i \leq 17$, and then calculated the mean time taken for data transfer (back and forth) and full PV analysis and synthesis.

The maximum block size considered of $2^{17} = 131072$ samples corresponds to a period of about 3 seconds of audio. The execution time of the full Phase Vocoder for block sizes of more than 2^{17} samples largely exceeds the corresponding DSP period, so this block size seems enough to provide upper bounds for feasibility of computation as a function of block sizes.

As observed by Savioja et al. [5] the additive synthesis snippet is computationally intensive and very sensitive regarding the method used to obtain each sample value of the sinusoidal oscillators. We have compared the performance of 5 different implementations: (1) 4-point cubic interpolation, (2) linear interpolation, (3) table lookup with truncated index, (4) GPU's trigonometric primitives (specifically, we used the `sinf()` function of CUDA API, which computes a double precision floating point number), and (5) GPU's built-in texture lookup functions. The results for each implementation can be seen in Figure 2 and will be discussed in the next section.

3. RESULTS

To illustrate the results obtained, we present graphs for the tests made with batch processing, i.e. letting Pd run as fast as it can without waiting for the full periods of DSP cycle to end to produce new samples. Wav files were used as input and output for convenience and the FFT and PV algorithms were carried out for different block sizes as specified in the last section. Test results were generated for both models of GPU, but as memory transfer times and FFT kernel execution time varied as expected between models (due to the differences of transfer speed and number of cores), we show these results just for the faster model.

Figure 1 presents the result of memory transfer times and FFT kernel execution times for different block sizes on the GTX 470. Figure 2 shows kernel times for the full

PV analysis and synthesis as a function of block size for both cards.

By comparing Figures 1 and 2, we can see that there is a noticeable difference, of some orders of magnitude, between the time it takes to run the pure FFT and the full PV. Comparing the time taken by the two different algorithms on the same model of card, we see that the FFT itself takes time comparable to the memory transfers time, of about tenths of milliseconds, while the full PV implementation will take many seconds for larger block sizes. This means that hundreds of pure FFT executions could occur in a DSP cycle while only few full PV analysis and synthesis can actually be performed on the same amount of time.

3.1. Memory transfer times

Regarding memory transfer, we can observe on Figure 1 that the time it takes to transfer a certain amount of memory from host to device and back seems approximately linear in relation to block size. This seems reasonable once the bandwidth of data transfer between the host and the GPU is constant (see table on Section 2). Despite that, we should notice that it is unwise to use the memory bandwidth value to make assumptions regarding memory transfer speed, because this relation depends on the architecture implementation of the memory hierarchy.

Also, we could determine that memory transfer results are very close for the FFT and PV implementations. This also seems reasonable given that the amount of memory transferred on both implementations is the same. This indicates that the number of kernel calls or the amount of time kernels take working on data seems to have no influence on memory transfer speed.

Finally, we should notice that the difference of time taken for memory transfer on the two different GPU models is very small, and in both scenarios the back trip takes a little longer than host-to-device transfer. Again, the small difference between models is the expected behaviour because, despite being from different architectural NVIDIA GPU families, both models have high advertised memory bandwidth compared to the amount of data transferred during the tests (a maximum of about 4 MB at each DSP cycle).

3.2. Kernel times

Both sequential and parallel versions of the FFT algorithm are $O(n \log n/p)$ where p is the number of available processors. As we spawn tens of thousands of parallel threads at each DSP cycle (i.e. $n \gg p$), the number p of processors can be regarded as a constant that depends only on the GPU card the user has access to. Thus, the expected tendency is for the FFT kernel time to overcome memory transfer times as blocks get bigger in size.

As for the PV implementation, as noted in the last section, we could determine that the part that consumes the most GPU time is the oscillator sum of the PV synthesis. To be able to get a feel of how much resources

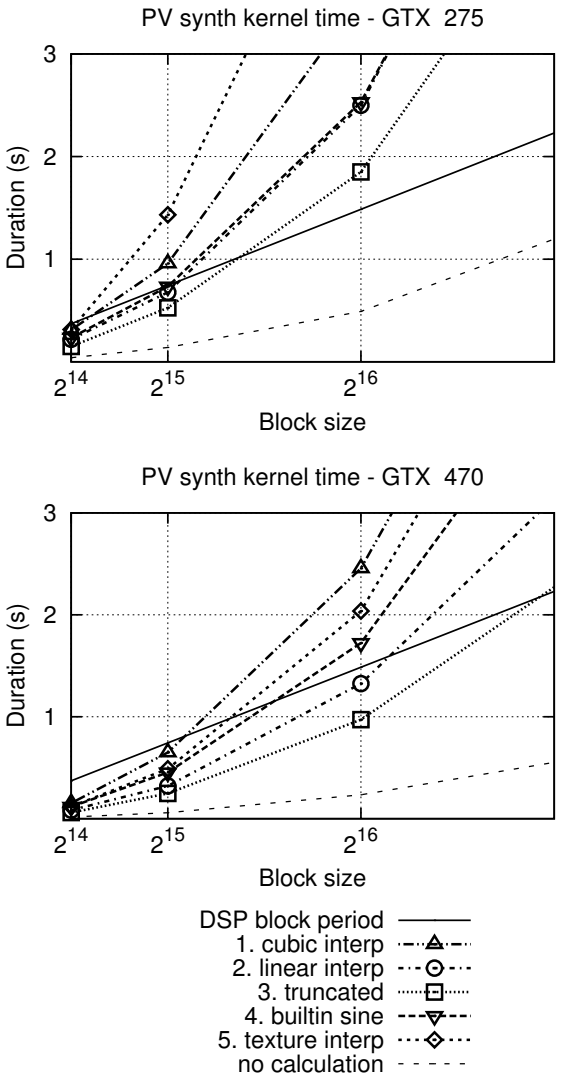

Figure 2. Kernel times for different implementations of oscillator calculation running on different card models.

user-defined computation consumes, we compared 5 different implementations of the oscillator calculation. Three of them use a 1024−point sine wave lookup table, previously calculated and copied from the host to the device, and two of them use GPU's built-in functions to aid the calculation. The five different oscillator implementations are: (1) table lookup with 4-point cubic interpolation, (2) table lookup with 2-point linear interpolation, (3) table lookup with truncated index, (4) direct use of the built-in sine wave function, and (5) table lookup with (linearly interpolated) texture fetching.

The time taken for each oscillator implementation can be seen on Figure 2. We have also plotted the DSP cycle period for each block size as well as a "control" implementation with no oscillator calculation at all, to work as a basis for comparison. Notice, though, that the control implementation includes the full PV analysis (which comprises one FFT run for the oscillators' phase estimation) and also the main synthesis loop, but with no oscil-

lator calculation (the loop just sums a small constant float number at each round).

Regarding implementations (1), (2) and (3), we can see that they behave as expected, i.e. they grow roughly proportionally, according to the number of operations involved; truncated table lookup is faster than linear interpolation, which is in turn faster than cubic interpolation. Consistently, all of them are faster on the GTX 470 when compared to the same implementation running on the GTX 275.

The built-in sine wave implementation (4) takes approximately the same time as implementation (2) on the GTX 275, but has a proportionally worse performance on the GTX 470, achieving an intermediate result between implementations (1) and (2).

As for the built-in texture fetching with linear interpolation, also known as implementation (5), its behaviour is a bit difficult to explain. On the GTX 275, it has the worse behaviour of all methods, taking 40% longer than the second most expensive, cubic interpolation. On the other hand, on the GTX 470 implementation (5) has a performance comparable to the others, being a little bit faster than cubic interpolation. We assume that this behaviour has something to do with differences in texture memory fetching implementations on different card models.

From these plots we can see that, for the GTX 275, blocks with size bigger than 2^{16} take more time to compute any of the Phase Vocoder implementations than the time available for realtime applications. A similar result was found for the GTX 470 with blocks bigger than 2^{17} samples.

We can also observe that Phase Vocoder times grow superlinearly for all implementations. Since the number of oscillators in the additive synthesis is roughly half the number of samples in a block, a quadratic computational complexity $\mathcal{O}(n^2)$ is expected. The degree of parallelism brought by the GPU is not enough to produce differences in profile for the variant oscillator lookup methods implemented, and the differences in scale are accountable by the hidden constant in the big-O notation.

4. DISCUSSION

Given the results described in the last section, we can conclude that small implementation differences can have significant results regarding kernel time consumption on the GPU. It is not clear whether the numerical quality of the built-in sine function is better than our 4-point cubic interpolation, but it is clear that conscious choices have to be made in order to use the GPU's full potential for larger block sizes.

We can also conclude that, if we restrict the roundtrip to few memory transfers in each direction, then there is no need to bother with memory transfer time as its magnitude is of the order of tenths of milliseconds while DSP block periods are of the order of several milliseconds even for the smaller block sizes considered.

By analyzing the time results of the full PV analysis and synthesis, it is possible to obtain the maximum block size for the realtime use of each oscillator implementation for each card model. The maximum number of samples achieved in each scenario is summarized below:

model \ implementation	1	2	3	4	5
GTX 275	2^{14}	2^{15}	2^{15}	2^{15}	2^{14}
GTX 470	2^{15}	2^{16}	2^{16}	2^{15}	2^{15}

4.1. Future work

Many interesting questions arose from this investigation, and there are plenty of directions to be followed. We summarize below some approaches which we shall pursue in future works:

- One of the present works of our Computer Music Research Group at IME/USP[6] deals with audio distribution over computer networks. It would be very interesting to analyze the possibility of outsourcing computation over the network to remote machines with GPU cards installed on them. Would it be worth it to use remote parallel resources for realtime scenarios given network delay?
- By using GPU's asynchronous execution possibilities, it is possible to decouple memory transfer and kernel execution from the Pd DSP cycle. We could, for example, use a producer/consumer model to feed intermediate buffers that could then be played independently from computation control. This could also give us better results for computing performance.

5. REFERENCES

[1] M. Dolson, "The phase vocoder: A tutorial," *Computer Music Journal*, vol. 10, no. 4, pp. 14–27, 1986.

[2] E. Gallo and N. Tsingos, "Efficient 3D audio processing on the GPU," in *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors*. ACM, August 2004, pp. 2004–2004.

[3] C. Henry, "GPU audio signals processing in Pure Data, and PdCUDA an implementation with the CUDA runtime API," in *Pure Data Convention*, 2011.

[4] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '03. Eurographics Association, 2003, pp. 112–119.

[5] L. Savioja, V. Välimäki, and J. O. Smith, "Audio signal processing using graphics processing units," *J. Audio Eng. Soc*, vol. 59, no. 1/2, pp. 3–19, 2011.

[6] N. Tsingos, W. Jiang, and I. Williams, "Using programmable graphics hardware for acoustics and audio rendering," *J. Audio Eng. Soc*, vol. 59, no. 9, pp. 628–646, 2011.

[6] http://compmus.ime.usp.br/