Graphics Processing Unit Audio Signals Processing in Pure Data and PdCUDA an Implementation with the CUDA Runtime API

Charles Z. Henry
Information and Telecommunications Technology Center (ITTC), University of Kansas
2335 Irving Hill Rd
Lawrence, KS USA 66045
chenry@ittc.ku.edu

Abstract

The design of graphics processing unit (GPU) audio signals processing extensions to Pure Data (Pd) is discussed with attention to future growth in GPU computing and the complexity of programming a general solution. An implementation named PdCUDA is presented for use of GPU general programming capability for audio signals processing with Pd and the CUDA runtime application programmers interface (API).

Keywords

Pd CUDA GPU audio computing

1 Introduction

Pd users designing applications for high throughput or large numbers of channels may use general purpose computing on graphics processing unit (GPGPU) programming to offload some functions to a GPU. Indeed, there is significant data parallelism present in Pd's digital signals processing (DSP) functions that is appropriate to be directly translated to usage on GPU's. The potential exists for creating high-performance computing platforms for audio signals processing with Pd.

However, GPGPU platforms may not be appropriate or even useful for all applications. For example, digital filters may be programmed efficiently with serial programming of a recursion equation, while equivalent functions on GPU's may have no benefit. The purpose of designing a GPGPU platform for audio signals processing is not to accelerate every application possible, but to selectively apply the technology where it may have the greatest benefit. It is hoped that the development of Pd GPU extensions will spur development of useful applications that are otherwise prohibitively expensive.

An intelligent algorithm which offloads functions to additional processors of different types is highly desirable. However, the difficulty in handling the multitude of possible applications to support makes such a solution infeasible. Further problems stemming from multiprocessor concurrency are discussed in Puckett [1]. Synchronization between concurrent processes takes place in GPU programs where functions are launched asynchronously. Since asynchronous execution may be necessary to take full advantage of GPU computation, this issue is unavoidable in the context of Pd GPU extensions.

The approach taken with PdCUDA is to create a set of tools which exposes the costs of GPU programming and makes it possible for Pd programmers to write patches that are tuned for performance of the given application. At the same time, it is desired that PdCUDA manage systematic performance considerations where possible.

1.1 GPU Computing

At the time of this paper, three of the top five and 19 of the top 500 supercomputers in the world make use of GPU's [2]. The rise in widespread usage is accompanied by increases in the performance of individual devices. This brings high-performance computing out of the cold room into situations where desktop computers, laptops, or even smaller mobile devices can take benefit.

GPU's make use of highly vectorized arithmetic and logical functions organized into blocks of threads. Hardware developed for implementing pixel and vertex shaders for graphics rendering is used with a given GPGPU API to perform floating point calculations.

Only NVIDIA and AMD/ATI currently compete for the GPU market with freely available GPGPU API's. However, it is conceivable that other companies such as Intel or IBM might later develop devices with similar general computing capability. NVIDIA has developed its CUDA[1] API for the hardware it designs, while AMD/ATI has adopted the OpenCL computing standard.

1 Compute Unified Device Architecture

The choice to develop CUDA , rather than OpenCL, extensions to Pd comes down to the differences in development platform. CUDA is narrowly constructed to apply to GPGPU programming while OpenCL can be applied to run programs on other platforms as well. With added flexibility, OpenCL also becomes more complex. Secondly, hardware from NVIDIA is compatible with both CUDA and OpenCL. This presents the opportunity for each corresponding set of GPU extensions to be programmed and run on the same hardware. Therefore, CUDA presents the best choice for creating a reference design upon which OpenCL extensions may be based, with one-to-one comparisons.

The biggest concern with performance of CUDA programs is the costs of transferring data between host and device [3]. Although bandwidth between host and device is typically high, on the order of GB/s, there is always an associated latency per memory transfer on the order of microseconds. Time not spent performing computations is an overhead cost to the program execution. Where possible, data transfer is to be avoided or run concurrently with threads performing computations.

There are further issues with memory access on GPU's having to do with hierarchies of memory residing all on the same device. There are important considerations in using the right type of memory for each type of function to perform, and reducing transfer of memory between memory types. For example, constant and texture caches may be used to great effect in running large number of oscillators or table reads.

1.2 Pure Data

Pd makes use of a type of shared memory model for run-time DSP execution. Shared memory models in general apply to access control, memory allocation, and addressing in applications having multiple threads.

Specifically with Pd, memory spaces are shared among DSP routines for objects in a canvas and across non-reblocking canvas boundaries. In Pd, signals are allocated for each graphical objects' inlets and outlets. Signals are “borrowed” from one another, sharing the same memory space, according to the DSP graph sorting algorithm.

Data transfer between arrays allocated in host memory occurs only in situations where a subcanvas reblocks or resamples signals from its parent canvas. Buffers are created to store signals in the intermediate state, and the resulting arrays from resampling or reblocking are copied into newly allocated signals.

The conditions for handling signal allocation, sharing, and data transfer is determined by a collection of variables, which for the purposes of this article will be called the DSP state. A dspcontext structure is created for each canvas when the DSP graph is being built. It is important to differentiate that canvas/subcanvas relations with block~ and switch~ objects make up the user interface to control DSP state while the dspcontext directs the behavior of signals' memory addressing during the DSP graph generation.

The DSP state is composed of the dspcontext data structure elements dc_toplevel, dc_reblock, and sc_switched. There are 6 possible states, since a toplevel dspcontext cannot also be reblocked. The code which parses the logic on DSP state is contained in d_ugen.c, most notably the routines ugen_done_graph and ugen_doit.

2 Design of GPU Extensions

2.1 Design Goals

Produce clean easily maintained code. A forward look at the practical aspects of GPU computing is appropriate. An implementation must track changes in both the Pure Data and GPU computing API's. With a growing list of architectures and devices to support, the code itself must be laid out in the most logical and consistent manner for developers. A stable implementation is one that abstracts the GPU computing functions so that externals developers will not have to re-write their code with respect to changes to the GPU computing API or the Pd API.

Provide an unambiguous user interface. There are clear pitfalls to users for a mixture of host and GPU computing externals, namely performance issues involving non-uniform memory access. While a given implementation expects to handle much of the memory considerations “behind the scenes,” users will need to create efficient patches by knowing at which stages their programs incur latency or other computing costs.

Make improvements to the Pd DSP scheduler specific to the GPU computing platform in use, for example, differentiating between GPU's operating in pinned host memory found on many low-power laptops and those with external memory found in most desktop graphics cards.

Make GPU routines profile-able. Measurement of the device performance is important for specifying an appropriate platform for a given application. This is also a benefit to users when designing an algorithm by comparing actual throughput against theoretical and practical performance maximums.

Make DSP run-time performance efficient. GPU computing functions can be written to be optimized for the types of data used by Pd, such as specific functions for operating on arrays with lengths in powers of two. The implementation should be written to prefer specialization over generalization and present developers with sets of functions commonly used in programming Pd externals.

2.2 Usage Cases

Some considerations in designing GPU extensions for Pd become evident by examining usage cases. Presented are usage cases for single GPU functions scaling to serial graph portions and to multiple data-independent branches.

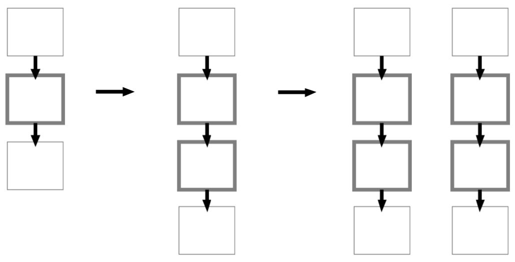

Fig. 1. Usage cases increasing in complexity. Dark rectangles represent GPU based perform functions.

In the simplest case, the GPU perform routine must transfer data into the GPU memory space, perform an operation, and transfer data back. The overhead incurred in performing multiple functions is two data transfer operations per function. Pd externals can be written to require no modifications to Pd itself as long as the perform routine expects an argument with signals allocated in the host memory space.

Supposing there were multiple GPU perform functions to be performed serially, it is possible to maintain common memory allocation, consistent with the way that Pd handles signals. Then, the overhead with performing the DSP routines in series is 2 data transfer operations, independent of the number of operations performed in series.

With multiple data-independent branches, the ordering of the DSP chain becomes important. It is possible to overlap data transfer and computation which hides the additional latency posed by multiple branches. The data transfer routines need to be added to the DSP chain in a breadth-first fashion and run asynchronously. Next, the GPU routines would be added to the DSP chain depth-first per branch,in the order of the branches' dependence on data transfer. Additional synchronization between data transfer and computation may be needed to reduce the cost of data transfer. Great care has to be taken with optimizing concurrency because such synchronizations pose a risk to real-time behavior.

3 Implementation

The description of PdCUDA focuses on details relevant to create a basic platform for externals development. Altogether, there are 84 different DSP perform routines to write in order to duplicate the functionality of Pd Vanilla. Many of these are further dependent on other Pd objects such as arrays, which are not implemented in PdCUDA at this time.

The signal data structure contains a pointer to array s_vec and its length s_n. This same information is what is necessary to allocate arrays in GPU memory and perform operations on those arrays. Signals are used interchangeably regardless of which memory space they use.

Most importantly, PdCUDA does not modify or duplicate the dspchain structure. CUDA specific perform routines are scheduled in a way that preserves the same data dependency relations present in Pd.

3.1 Extending the DSP State

Additional information is needed to determine signal handling when multiple memory spaces are involved. Data transfer between the host and device occurs whenever a dspcontext and its parent dspcontext have different values of dc_hascuda. New signals are allocated in GPU memory when dc_hascuda is set, and where a non-reblocked dspcontext and its parent dspcontext have the same value of dc_hascuda, signals are allowed to be borrowed from the parent dspcontext.

The DSP state now must include dc_hascuda and whether a dspcontext and its parent dspcontext have the same value of dc_hascuda. The total number of states becomes 20. This maintains compatibility with Pd, with the original 6 possible values of DSP state being a subset of the new DSP state values.

3.2 Separation Between Memory Spaces

The symbol "cuda_dsp" is introduced in order to keep CUDA based routines separate from their host counterparts. When canvas_dodsp runs for a given cucanvas, it finds all instances of objects from its gl_list with the symbol "cuda_dsp" and adds their ugens to the dspcontext.

An existing class may then be extended to work with PdCUDA by adding an additional method, instead of adding an entirely new class for the same purpose. In this scheme, there is no risk of mixture between memory spaces.

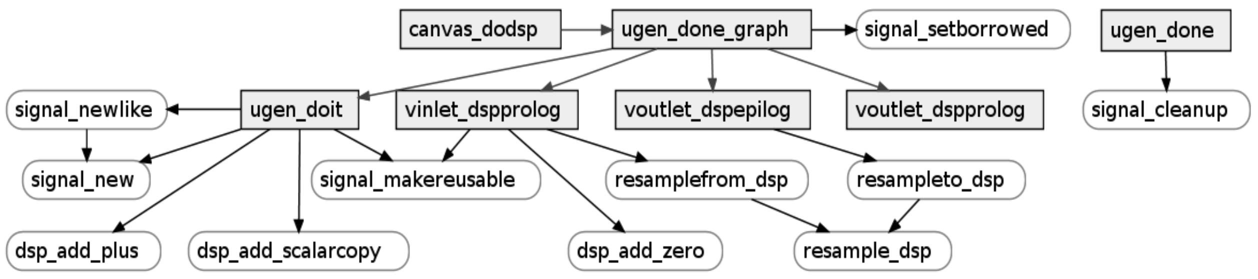

Fig. 2 Relevant portions of the Pd DSP sorting routines. Functions in dark rectangles are to be modified to add logic for handling conditions dependent on DSP state. Functions in light ovals are specialized. It is required to write a new instance of each specialized function for GPU computing with distinct names.

3.3 User Interface

User control over the application of CUDA routines needs to be handled at the canvas level. This works within expected user interfaces presented by Pd and provides the user capability to control the coarseness of organizing CUDA based patches. The symbol “cucanvas” is introduced for creating root level canvases. The canvas (glist) data structure is extended by adding a gl_hascuda element. The user interface also adds a subcanvas creation symbol “cu”.

It is possible to create subcanvases of any type within canvases of another type. This is important for creating abstractions that are handled in the same way that Pd abstractions are commonly used.

3.4 Modifications to Pd Code

The data structures glist and dspcontext are added to store the additional hascuda variable required for the extension to DSP state. Next, the portions of code dealing with DSP graph sorting and memory allocation are affected in two different ways. Note that vinlet and voutlet data structures each contain a pointer to the canvas in which they reside. As long as the functions in the canvas_dodsp call graph have access to the canvas gl_hascuda or the dspcontext dc_hascuda variables (and their parents), they can be directed to handle signals correctly for the GPU.

There are also many functions that are do not have access to any canvas, dspcontext, vinlet, or voutlet. These functions are specialized, in that they perform operations on signals, specifically allocated in host memory. These functions are “cloned” to create CUDA versions of the same function. Fig 2 above shows the relevant portions of the canvas_dodsp call graph. In total, 7 functions must be modified and 11 functions must be cloned in order to create the basic system for handling signals correctly in host and GPU memory.

Then, the first version of PdCUDA to be programmed includes files replacing g_canvas.c, d_ugen.c, d_resample.c, and d_io.c. Also included are files which separate out the code added and the code to be compiled with the CUDA compiler.

3.5 The PdCUDA API

The goal is to allow Pd externals developers to create CUDA based perform routines using only C code, not requiring advanced understanding of CUDA or the GPU hardware in use. The challenges discussed in this article require a unified approach to achieve any kind of meaningful gains in performance. Functions in the PdCUDA API will be added as the project gets closer to duplicating much of the functionality of Pd Vanilla.

4 Conclusion

Late in this process, I realized that the choice of modifying the canvas class could be replaced by other means of controlling the DSP state. It should be possible to add a cuda~ object modeled after block~/switch~ which is capable of handling the extended DSP state.

PdCUDA is under development. Milestones in the project will be added to the project's sourceforge page as work progresses.

Refrences

[1] M. Puckette: “Multiprocessing for Pd,” Pure Data Convention, Sao Paolo, Brazil, 2009.

[2] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon “Highlights - June 2011,” www.top500.org, June 2011.

[3] “CUDA C Best Practices Guide Version 3.2,” August 2010.