VSIPL++ Acceleration Using Commodity Graphics Processors

Dan Campbell
Georgia Tech Research Institute - Sensors and Electromagnetics Applications Laboratory
dan.campbell@gtrigatech.edu

Introduction

Commodity Graphics Processing Units (GPUs) are application-specific processors that implement a standardized three-dimensional graphics rendering pipeline, and provide significant floating-point processing capacity at much lower cost, power consumption, and physical space compared to general purpose processors. Recent changes in GPUs have increased programmability and flexibility in portions of the rendering pipeline, allowing non-graphics applications to exploit their computational capacity. Restrictions on the programming model, lack of appropriate tools, unusual performance behavior, and other factors make exploiting GPUs a costly, difficult, and time consuming process for application developers. The Vector, Signal, and Image Processing Library (VSIPL) is an industry standard Application Programming Interface (API) for portable, high performance vector and matrix linear algebra and signal processing applications. The High Performance Embedded Computing Software Initiative (HPEC-SI) is developing parallel and object oriented extensions to VSIPL with the goal of a unifying computation and communication in a single high performance, productive, and portable API. The original VSIPL specification, advances to the VSIPL standard under the HPEC-SI program, and commodity graphics processors together constitute the basis of a system for enabling ubiquitous high-performance signal processing software development. VSIPL establishes a functional basis that spans the majority of high performance signal processing tasks, the HPEC-SI program has developed important extensions to the VSIPL standard, and commodity GPUs enable wide distribution of computing systems with high computational capacity and low cost.

Commodity Graphics Processors

Commodity graphics processing units (GPUs) are application-specific processors that implement standardized three-dimensional rendering pipelines. Recent generations of GPUs have added limited programmability to certain portions of the rendering pipeline. This programmability can be exploited to cause the GPU to perform calculations unrelated to graphics. After initial experiments demonstrated the computational potential of such an approach, GPU hardware vendors added, in subsequent generations, floating point pixel types, increased flexibility in execution models, and an array of tools that significantly expanded the number of computation problems that can be addressed by GPUs. With enough flexibility established to perform general purpose computations, GPUs have become an attractive deployment platform candidate for signal processing applications. The standard three-dimensional graphics rendering pipeline includes a stage that allows a final set of manipulations on each potential pixel (referred to as a “fragment” within OpenGL, one of the two standard graphics APIs), after all standard geometry, stenciling, lighting, and other fixed-function portions of the pipeline have been completed. The commodity GPU vendors’ primary market use is high resolution video games, with up to two million pixels per rendering frame (1600 by 1200 resolution) commonly supported. As a result, the primary application has embarrassingly parallel characteristics, and GPU vendors have been successful in obtaining increased performance by increasing the degree of parallel computation dedicated to fragment processing. This parallelization has been achieved by a variety of methods at the microarchitecture level, but is not exposed explicitly to any API. Current generation GPUs have delivered observed performance of up to 250 GFLOPS on synthetic benchmarks, in comparison to a peak theoretical computation rate for a dual core Pentium 4, 3.46GHz CPU of approximately 28 GFLOPS. In addition, GPU computational capacity is growing at a much faster pace than CPUs, so this gap is expected to continue and widen in the future. Furthermore, GPUs deliver this performance at a cost that is a fraction of the cost of an additional standalone computer, or other high performance coprocessors, such as DSPs and FPGAs.

Commodity GPU vendors hide many architectural details, and only expose programmatic access to the GPUs via two primary APIs, OpenGL and DirectX Graphics. Both of these APIs are graphics-oriented and impose many programming restrictions and hurdles specific to graphics software. They require an understanding of graphics programming to use, and force all non-graphics applications to be cast in terms of graphics operations, regardless of whether this step is required by the microarchitecture. Furthermore, there are restrictions to the execution model for fragment processing (such as looping constructs, conditional branching, and total program length), and optimization trade-offs that are often quite different from traditional processors. The restrictions, performance characteristics, and optimization modes are typically hidden, or obscurely documented. Delivering the performance capability of GPUs to deployed applications, therefore, has been difficult, expensive, and slow. Most fielded systems include large portions of hand-coded optimizations, and are developed by a small number of domain experts. This has limited the adoption of GPUs as fielded coprocessing accelerators. Several efforts have been undertaken to expand the application development infrastructure available for GPUs, mostly focusing on compilers for new languages, such as BrookGPU, Sh, and R-Stream and special-purpose functional kernels, such as GPUFFTW. An approach that has not been significantly explored for GPUs is Domain Specific Libraries (DSLs). DSLs provide an ideal connection between application domains, with relatively static functional spans, and widely varying architectures. For each new hardware platform, only the specified functions must be redeveloped and re-optimized, rather than all existing application software, or an entire compiler suite. Improved infrastructure is required in order to deliver the full capabilities of GPUs to application developers.

VSIPL++ is a Domain Specific Library that is ideally suited to exploit the capabilities of GPUs. It is designed for signal processing, image processing, and linear algebra tasks that typically have similar levels of inherent parallelism to graphics rendering. VSIPL++ includes explicit mechanisms for managing separate memory spaces, and presents a data and storage abstraction that maps well to both its target application space and the available data storage mechanisms on GPUs. The elements of VSIPL++ that are specific to the C++ expansion and distinct from the original VSIPL are also particularly well suited to GPUs. GPUs impose relatively high per-loop, and per-data-access latency costs compared to general purpose processors. VSIPL++ includes provisions for expression-level specialization and loop fusion, which significantly mitigate these costs.

Implementation & Methodology

The implied execution and control model associated with VSIPL and VSIPL++ corresponds well with the control mechanisms that area available to applications for managing GPUs. This allows a relatively simple & straightforward implementation of the data management and simpler math functions, leaving avenues for special-purpose optimization and customization of time-critical and complex portions of the API. GPU-VSIPL++ is implemented in a layered approach, using a modified version of the reference implementation of VSIPL++, created by CodeSourcery, LLC, and a GPU-based implementation of portions of VSIPL (GPU-VSIPL). GPU-VSIPL blocks are implemented as OpenGL textures. `vsip_admit` and `vsip_release` create barrier OpenGL calls that move data to and from texture memory as needed. Blocks containing complex data types are implemented by creating two textures of the appropriate size, rather than one texture of twice the size, corresponding to the `VSIP_SPLIT` data storage rather than `VSIP_INTERLEAVED`, but is not relevant to the application programmer, except for optimizing `VSIP_ADMIT` and `VSIP_RELEASE` calls.

Simple math operations among compatible views are implemented by causing an OpenGL rendering operation that renders a rectangle of the same size as the texture holding the block data referenced by the output view. The input views are made available to the fragment processor via the texture containing the block associated with the view. Any loop-invariant variables are set via Cg runtime calls to set `uniform` Cg types. Data reads from input textures are implemented as texture sampling operations, and data are output by setting the values of the components of the output texture, corresponding to color in a graphics context. The VSIPL operation is implemented as a Cg fragment program.

Due to the restrictions placed on read and write patterns, many VSIPL operations can not be implemented as single render operations with simple Cg programs. For example, any function that performs a reduction (*e.g.* vsip_vsumval_f) must create a temporary texture and perform multiple decimating render passes until only the desired number of elements remain. Several variations of this approach are taken throughout the implementation. Some operations are more efficiently implemented as a Cg program that is constructed on the fly. For example, the vsip_firflt_f suite of functions create a Cg program at the time the vsip_fir_f object is created, based on the filter length and kernel coefficients.

Benchmarks

Several simple GPU-VSIPL++ functions were benchmarked and compared to CPU-only execution of the reference VSIPL++ implementation, using the TASP-VSIPL reference implementation as a backend For small vector sizes, the per-operation cost dominates, and the CPU-only VSIPL++ outperforms the GPU-VSIPL++. However, for larger vector sizes, GPU-VSIPL++ allows significant speedups over the CPU-only implementation. Asymptotic speedups of 15-20x were delivered for simple vector operations, and as high as 79x for FIR operations on larger vectors. Figure 1 shows the performance of a GPU-VSIPL++ FIR filter, normalized against the CPU only implementation.

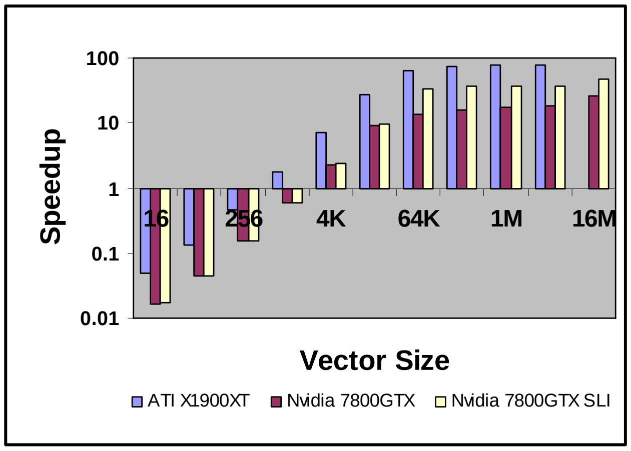

Figure 1: Normalized FIR performance