

VSIPL++ Acceleration Using Commodity Graphics Processors

Dan Campbell

Sensors and Electromagnetics Applications Laboratory, Georgia Tech Research Institute, Smyrna, GA
dan.campbell@gtri.gatech.edu

Abstract

The High Performance Embedded Computing Software Initiative (HPEC-SI) is developing a unified software framework for computation and communication for high performance signal processing tasks on parallel computers. The goal of the program is to address the high cost of software in Department of Defense (DoD) systems by improving the portability and productivity of signal processing application development, while simultaneously improving performance compared to current practices. The Vector, Signal, and Image Processing Library (VSIPL) is a portable Application Programming Interface (API) that is widely used for embedded DoD signal processing systems. One portion of the HPEC-SI effort includes the development of C++ extensions for the existing VSIPL standard, called VSIPL++. 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 embedded systems that VSIPL and VSIPL++ are commonly used on share several important characteristics with GPUs, making VSIPL++ well suited to abstract and exploit GPUs. This paper describes GPU-VSIPL++, an implementation of portions of the VSIPL++ standard that exploits a GPU to accelerate computation beyond what is possible on a development workstation.

1. Background

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.^[1] 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 compute engines with high computational capacity and low cost.

1.1. VSIPL

VSIPL is an industry standard API for portable vector and matrix linear algebra and signal processing that was formed under initial direction from the Defense Advanced Research Projects Agency (DARPA), but grew to include significant contributions from a wide variety of academic, commercial, and government groups. Since publication of the VSIPL specification, VSIPL has enjoyed tremendous success, and wide adoption among new development efforts in the signal-processing domain. VSIPL has had significant impact on signal processing application portability by providing a C interface to the operations most often accelerated through proprietary libraries on embedded platforms. This has allowed high performance portability between a wide variety of platforms with minimal rewrite. VSIPL has improved productivity by raising the abstraction level of application software from a need to hand optimize performance critical portions to an ability to leave software written in a language that more closely resembles the mathematics expressed by the algorithm.

VSIPL is an API for the C programming language that also embodies an implied computational and execution model for embedded signal processors. Since it was designed for embedded signal processors, VSIPL includes the functional space typically targeted by such

processors, as well as explicit controls for managing a coprocessor-based accelerator. The VSIPPL functional space includes support for zero-dimensional (scalar) to three-dimensional (tensor) data support, floating point, integer, fixed point precisions of varying bit-depths, complex and real data types, basic linear algebra operations, signal processing filters, linear system solvers, bookkeeping, and statistical support. Additionally, VSIPPL includes data abstraction and ownership mechanisms. These allow control of data movement and synchronization between main physical memory spaces, and expression of mathematical constructs, such as vectors and matrices, abstractly, and dynamically in terms of underlying contiguous memory structures.

1.2. HPEC-SI

The HPEC-SI program addresses the high cost of DoD software development by creating a unified computation and communication API and software development framework for high performance signal processing applications. The focus of the program is to develop an API that will reduce the number of lines of code needed to write applications by two-thirds, reduce the number of lines of code required to port an application from one platform to another or another configuration of the same platform by two-thirds, and to improve the performance of applications by half. The HPEC-SI program is leveraging the success of other highly successful and widely adopted software standards. The program is developing APIs that maintains as much as possible of the functionality and general methodology of existing standards, which mitigates the cost of adoption for developers, and the risk that the API developed by the program can be implemented efficiently.

The HPEC-SI Program is comprised of three working groups, each performing a distinct task in the process of moving software approaches from the advanced research communities into deployed systems. The Applied Research Working Group (ARWG) analyzes a portion of the HPEC-SI Program goals, considers existing research and solutions for the goal, and designs a general approach for a standardizable solution to accomplish the goal. The Development Working Group (DWG) implements the solutions formed by the ARWG. This includes converting a broadly described software approach into a specific, detailed, and complete specification, creating a prototype reference implementation of the specification, developing small example applications, and developing whatever testing and compliance suites are necessary. The DWG re-implements large scale, deployed defense applications using the defined standards to show that the APIs and frameworks developed meet the HPEC-SI program goals for improved performance, portability, and development productivity. The HPEC-SI program has

been executed in phases, during which each working group focuses on a specific aspect of the overall program goals, and releases its results to the next working group or appropriate standards bodies on completion.

The first primary task of the HPEC-SI Development Working group was to establish C++ extensions of the VSIPPL standard, known as VSIPPL++. The functional space established by the VSIPPL standard provides a clear basis for the computation goals of the HPEC-SI program, but it has shortcomings that make it insufficient to meet the HPEC-SI program goals. Design goals for this process included the inclusion of operator overloading, simplification of the VSIPPL API, and maintaining a forward path for expression level optimizations and seamless parallelization of VSIPPL++ applications. The first version of the VSIPPL++ specification was completed by the HPEC-SI program, and released to the VSIPPL Forum in November 2003. An updated revision was approved by the VSIPPL Forum in August 2004.^[2]

Because VSIPPL is specified in C, which does not support templates and overloading, the API requires separate function signatures for each of the accepted data types for each of the specified functions. This has resulted in a specification with many hundreds of function names and data types where as few as one tenth as many would otherwise suffice. Reducing these groups of functions to one, intuitively named function significantly simplifies the software development process. Also, operator overloading allows additional intuitive code to be written. For example, each of the function signatures in Table 1 expresses the same basic vector operation.

Table 1. VSIPPL element-wise addition functions
A = B + C

<code>vsip_vadd_f</code>	(A, B, C)
<code>vsip_vadd_i</code>	(A, B, C)
<code>vsip_cvadd_f</code>	(A, B, C)
<code>vsip_rcvadd_f</code>	(A, B, C)
<code>vsip_madd_f</code>	(A, B, C)
<code>vsip_madd_i</code>	(A, B, C)
<code>vsip_rcmadd_f</code>	(A, B, C)
<code>Vsip_cmadd_f</code>	(A, B, C)

Note: `_f` represents one of at least four different specific floating point precision specifiers, and `_i` represents one of approximately 40 specific integer precision specifiers

This adds complexity to the API, complexity to applications developed using the API, and limits performance optimizations available at the expression

level. The inclusion of precision-templated object types for data elements such as scalars, vectors, matrices, and tensors further reduces complexity and improves code readability and maintainability.

The performance goal of VSIPL was to deliver the portability and productivity gains without significantly diminishing performance compared to proprietary interfaces. While this goal was met, it is not sufficient to meet the HPEC-SI program's performance goals. Existing products, such as the Portable

Expression Template Engine (PETE) has demonstrated dramatic performance improvements in compound vector operations over standard C++ operator overloading. By making use of template engines and operator overloading, it is possible to evaluate compound vector operations at an expression level at compile time, outperforming C API based approaches that handle each atomic vector operation individually. As an example, consider the vector operation:

$$A = B * C + D$$

Using a C-based atomic vector operation API (e.g., VSIPL, Core-Lite profile), this would be expressed in a manner resembling:

```
vsip_vmul_f (B, C, temp);  
vsip_vadd_f (temp, D, A);
```

Since the interface to the math library functions is at the operation level, rather than the expression level, opportunities for optimization are lost. Expression templating allows the above expression to be fused into one operation per vector element on architectures supporting a fused multiply-add operation, allows improved cache coherency by reducing the operation to a single pass through the vector, eliminates one loop-pass (which is helpful in some architectures with high per-loop costs) and eliminates the need for temporary storage. While this is a simple example, the same types of optimizations are available to many expressions.

1.3. Graphics Processing Units (GPUs)

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 may be exploited to cause the GPU to perform calculations unrelated to graphics.^[3,4] After initial experiments demonstrated the computational potential of such an approach, GPU hardware vendors added, in subsequent hardware generations, floating point pixel types, increased flexibility in the fragment processor execution models, and an array of tools that significantly expanded the

number of computation problems that may be addressed by GPUs.

GPUs provide a significant amount of computational capacity, and with enough flexibility established to perform general-purpose computations, have become an attractive deployment platform candidate for signal processing applications. The standard 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 application-level programming interfaces. 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, 3.46 GHz 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.^[5,6] Both of these APIs are graphics-oriented and impose many programming restrictions and hurdles specific to graphics software. They require a fair level of graphics understanding to use, and force all non-graphics applications to be cast in terms of graphics operations, regardless of whether this step is actually 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 poorly documented. Therefore, delivering the performance capability of GPUs to deployed applications has been difficult, expensive, and slow. A small number of domain experts have developed most fielded systems, which include large portions of hand-coded optimizations. 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 GPUFFT^[7-10]. An approach that has not been significantly exploited 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 reoptimized, rather than 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 DSL 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 mapping 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 the above costs.

2. Implementation and Methodology

The implied execution and control model associated with VSIPL and VSIPL++ corresponds well with the control mechanisms that are available to applications for managing GPUs. This allows a relatively simple and 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. In order to expand the potential user-base, GPU-VSIPL++ is implemented in a layered approach. The reference implementation of VSIPL++, created by CodeSourcery, LLC under support from the HPEC-SI program, is built as a wrapper that must be linked against an existing (C-based) VSIPL implementation.^[11] GPU-VSIPL++ consists of a GPU-based implementation of portions of VSIPL (GPU-VSIPL), various additional GPU-related functions, a version of the VSIPL++ reference implementation that has been modified to make use of the additional functions provided by GPU-VSIPL, and to use the “lazy operator”

technique to reduce loop counts and temporary storage use for selected functions. The GPU-VSIPL portion is implemented using OpenGL and the Cg language as the interface to the GPUs.^[12]

Blocks are the fundamental data storage mechanism in VSIPL and VSIPL++. In GPU-VSIPL, blocks are implemented as OpenGL textures using the `texel` element type that each represent four unclamped floating-point numbers. Textures are created during any `vsip_blockcreate_p` or `vsip_viewcreate_p` function call. The size of the texture is selected to the smallest power of two square that can contain the requested number of elements. This results in block size to texture-sized breakpoints that are powers of four, starting with four. Recent GPUs support non-power of two sizes, as well as non-square textures, but there are performance penalties in some cases for using such textures, so they are avoided where possible. `VSIP_ADMIT` and `VSIP_RELEASE` create barrier OpenGL calls that move data to and from texture memory as appropriate. Blocks containing complex data types are implemented by creating two textures of the appropriate size, rather than one texture of twice the size. This corresponds 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.

Vectors, matrices, and tensors in VSIPL and VSIPL++ are known as views, which are defined in terms of blocks. Views are entirely bookkeeping abstractions that do not directly contain data. They are defined in terms of a block, an offset, a per-dimension stride, and a per-dimension length. GPU-VSIPL implements views as simple bookkeeping abstract data types, but also strives to maintain unit stride, for reasons that will be discussed in the following paragraphs.

Current GPUs restrict write operations to one, predetermined, output buffer location per fragment. In addition, index arithmetic within fragment programs is very costly, due to the lack of integer support in fragment processors, and the need to convert between one-dimensional block coordinates and two-dimensional texture coordinates. This implies that for optimal performance, operation outputs must write to every element within a specific range of a block (and thus the output view is of unit stride), that all input blocks must be held within textures of the same size, and that the output and input indices match for every data element in the operation (thus requiring the input blocks to be of unit stride as well). Workarounds for layouts that do not meet these restrictions are implemented for some functions, but in most cases, the implementation falls back to a CPU-based solution for these layouts.

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. The Cg program for the function `vsip_vsma_f` is shown in listing 1.

The first call to each VSIPL function causes a runtime compilation of the applicable Cg program, resulting in a startup delay. The rendering operation causes the Cg program to be executed for each index in the output texture range.

Due to the restrictions placed on read and write patterns, many VSIPL operations cannot 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 remains. 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 creates a Cg program at the time the `vsip_fir_f` object is created, based on the filter length and kernel coefficients.

Listing 1. Cg Program for `vsip_vsma_f`

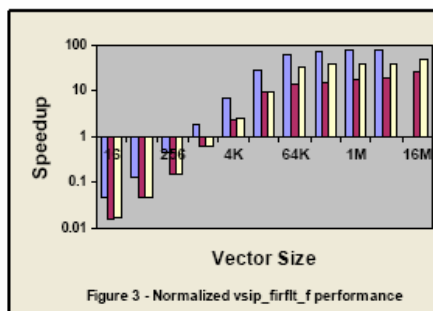
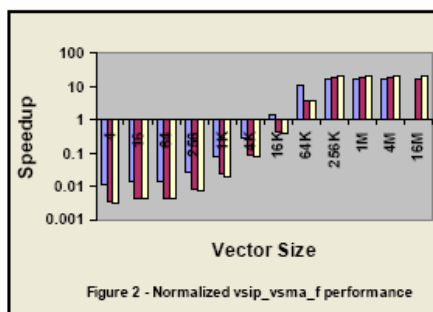
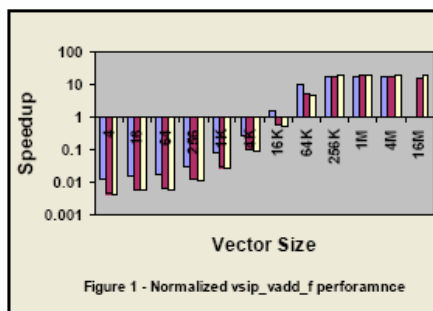
```
void main(float2 tc0 : TEXCOORD0,
         out float4 col : COLOR,
         uniform float4 beta,
         uniform samplerRECT decal0,
         uniform samplerRECT decal1)
{
    col = texRECT (decal0, tc0) * beta +
          texRECT(decal1, tc0);
}
```

3. Benchmarks

Several simple GPU-VSIPL++ functions were benchmarked and compared to CPU-only execution of the reference VSIPL++ implementation, using the TASP-VSIPL^[13] reference implementation as a backend. The testing workstation was a desktop PC equipped with an AMD Opteron CPU with 1-megabyte cache, operating at 2400 MHz, with 1 gigabyte of system random access memory (RAM). GPU-VSIPL++ was tested with three

GPU configurations: an ATI X1900XT GPU, an nVidia 7800GTX GPU, and a dual nVidia 7800GTX GPU configured in Scalable Link Interface (SLI) mode. The VSIPL++ functions tested were the VSIPL++ operator equivalents to `vsip_vadd_f`, `vsip_vsma_f`, and `vsip_firflt_f` (kernel length of 12). The performance (in terms of operations per time) for tests is shown normalized to CPU-only performance, in Figures 1, 2, and 3 respectively.

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. The relatively poor performance of GPU-VSIPL++ reflects the high per-render-operation latency cost previously discussed. The ATI GPUs had a smaller maximum texture size, which resulted in a maximum vector length shorter (by a factor of four) than the nVidia based configurations.



■ ATI X1900XT ■ Nvidia 7800GTX □ Nvidia 7800GTX SLI

4. Summary and Future Work

VSIPL, C++ Extensions to VSIPL developed under the HPEC-SI program, and commodity graphics processors are ideally suited for use together. GPUs deliver very high computational throughput at a fraction of the cost of equivalent amounts of CPU, FPGA, or specialty DSP capacity. Despite their advantages, GPUs have had limited adoption for signal processing tasks primarily due to the lack of appropriate development infrastructure, and the difficulty of programming and optimizing software. Domain specific libraries, such as VSIPL++, provide an integration path for new technologies with less disruption and development than hand optimizations or new language development. VSIPL++ is an ideal API for exploiting the capacity of GPUs for signal processing tasks. An implementation of portions of the VSIPL++ standard has been developed that utilizes GPUs to accelerate processing, and is shown to deliver significant acceleration over CPU-only implementations of VSIPL++ for large vectors.

Future work on GPU-VSIPL++ will primarily be focused on increased coverage of the VSIPL++ specification, and increased performance. The VSIPL and VSIPL++ specifications cover a wide array of functionality, and simplify applications by creating an encapsulated memory abstraction. The memory abstraction and functional span of the VSIPL and VSIPL++ specifications require implementations to cover a number of edge conditions and special cases that are important for deployed systems, but are not central to prototype and demonstration implementations. The decision to implement GPU-VSIPL++ via GPU-VSIPL simplified the development process, but sacrificed some performance. The abstraction barriers presented by the original VSIPL specification interfere with the benefits of VSIPL++ when a VSIPL++ implementation is built around a VSIPL implementation. A direct GPU-VSIPL++ implementation that is built entirely in C++ should show improved performance over the results obtained.

References

1. Shwartz, David A., Randall R. Judd, William J. Harrod, and Dwight P. Manly, "VSIPL 1.2 API." Published electronically at <http://www.vsipl.org>.
2. "VSIPL++ Specification 1.01.," Published electronically at <http://www.hpec-si.org/spec-1.01-final.pdf>.
3. Buck, I., "GPGPU: General-purpose computation on graphics hardware—GPU computation strategies and tricks." *ACM SIGGRAPH Course Notes*, August 2004.
4. Owens, J., D. Luebke, N. Govindaraju, M. Harris J. Kruger, A. Lefohn, and T. Purcell, "A Survey on General Purpose Computation on Graphics Hardware." *Eurographics*, 2005.
5. Segal, M. and K. Akeley, "The OpenGL Graphics System: A Specification (Version 2.0—October 22, 2004)." Published electronically at <http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf>.
6. "Microsoft DirectX", <http://www.microsoft.com/windows/directx/>.
7. Buck, I., T. Foley, D. Horn, J., Sugerma, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware." *ACM Transactions on Graphics*, Proceedings of SIGGRAPH 2004.
8. McCool, M. and S. DuToit, *Metaprogramming GPUs with Sh*, AK Peters, Ltd., Wellesley, MA, 2004.
9. Mackenzie, K., D. Campbell, and P. Szilagy, "A streaming Virtual Machine for GPUs." High Performance Embedded Computing Workshop, 2005.
10. Govindaraju, N. and D. Manocha, "GPUFFT: High Performance Power-of-Two FFT Library using Graphics Processors." <http://gamma.cs.unc.edu/GPUFFT/>.
11. "VSIPL++ Reference Implementation 1.01." Published electronically at <http://www.hpec-si.org/>.
12. "Cg Language Specification." Published electronically at ftp://download.nvidia.com/developer/cg/Cg_Specification.pdf.
13. Judd, R., "TASP VSIPL Core Plus Plus." Available electronically at <http://www.vsipl.org/software/>.