GPU VSIPL: High-Performance VSIPL Implementation for GPUs

Andrew Kerr, Dan Campbell, Mark Richards
Georgia Institute of Technology, Georgia Tech Research Institute
{andrew.kerr, dan.campbell}@gtri.gatech.edu, mark.richards@ece.gatech.edu

Abstract

In this paper, we introduce GPU VSIPL[1]*, an implementation of the Vector Signal Image Processing Library (VSIPL) Core Lite profile developed for the graphics processing unit (GPU). This implementation achieves a speedup of one to two orders of magnitude over the reference implementation. Because VSIPL is an open standard for high-performance platform-independent applications, GPU VSIPL provides an abstraction layer that leverages the GPU with no additional development costs.*

Introduction

Commodity graphics processing units (GPUs) are highly parallel programmable microprocessors. The current high end GPUs achieve a peak performance of 500 GFLOP/s in single precision. Previous methods of performing general-purpose computation on GPUs required algorithm implementations be cast as 3D graphics operations with prohibitive limitations on programmable vertex and pixel shader length, control flow, and arthmetic capabilities. In 2006, NVIDIA released the Compute Unified Device Architecture (CUDA) [1] development platform for the GPU. CUDA specifies extensions to the C programming language for writing program "kernels" directly targeting GPUs. This enhances the viability of GPUs as a general-purpose computing solution by providing a straight-forward programming model and language exposing the GPU's many parallel data paths. However, achieving high performance requires careful consideration of the GPU architecture and rigorous optimization efforts, the results of which are not portable to other architectures. Consequently, a domain specific library implemented for the GPU is desirable for convenient application development.

The Vector Signal Image Processing Library (VSIPL) [2] is a portable API for implementing high-performance signal processing applications while retaining platform independence. VSIPL supports memory abstractions for utilizing coprocessors with disjoint memory spaces. A signal processing application may structure input data in a block, admit it once to VSIPL's memory management, perform computations on that data, and release only the block containing the final result. Intermediate results are not transferred between system and GPU memory avoiding unnecessary latencies and communications overhead. This capability distinguishes VSIPL from other signal processing libraries that permit random access to data.

GPU VSIPL is an implementation of the VSIPL Core Lite profile [3] for the GPU. This implementation achieves high performance by performing all processing on the GPU. By requiring all data blocks be admitted to VSIPL before operations may be called on them, GPU VSIPL effectively hides the disjoint memory spaces from client applications. New and existing applications written with VSIPL may leverage the performance of the GPU simply by linking with GPU VSIPL.

VSIPL Core Lite Compliance

The VSIPL Core Lite profile is a subset of the VSIPL specification and covers single-precision floating-point blocks and vector views. VSIPL Core Lite specifies a rich set of support functions for creating, modifying, and destroying blocks and managing associated buffers on the GPU and system memory. Element-wise mathematical operations defined for real- and complex-valued vectors, inner products, scalar operations on vectors, and extrema searching are implemented. FIR filtering is defined with respect to explicit optimization hints for minimizing execution time or memory utilization. An out-of-place Fast Fourier Transform using the CUDA FFT library distributed by NVIDIA is provided. Additionally, a histogram and portable random number generator defined in [2] are implemented.

The GPU VSIPL implementation is distributed as a static library with C linkage. It was implemented with NVIDIA's CUDA 1.1 programming language and C++ compiled with Visual Studio 2005. GPU VSIPL passes all compliance tests of the VSIPL Test Suite [4].

Implementation for the GPU

Kernels were implemented in the CUDA programming language as function templates with both datatype and operation as template parameters thereby avoiding redundancy and minimizing the scope of optimization efforts. Optimization techniques for the GPU focus on the following elements: parallelism, memory access optimizations, and loop and data flow optimziations. We will show how these techniques undertaken during the implementation of GPU VSIPL's FIR filtering result in high utilization and performance.

[1]This work was supported in part by DARPA and AFRL under contracts FA8750-06-1-0012 and FA8650-07-C-7724. The opinions expressed are those of the authors.

To maximize utilization of the GPU's arithmetic units, an algorithm must be partitioned into blocks and threads with many active threads and few synchronization points. Numerous concurrent threads permit active threads to perform computations while others wait for memory accesses to complete. Reduction operations like dot product and `sumval` require synchronization between blocks and must be implemented with two kernels. The first kernel applies the reduction operator to parts of the input vector by launching many blocks, while the second kernel, implemented as a single block, applies the reduction operator to the results from the first. These may be executed consecutively without incurring additional PCI-Express bus latency via CUDA *streams* in which kernels and their arguments are serialized and dispatched in a single call.

GPU threads may issue 32-, 64-, and 128-bit reads and writes in a single instruction. If these addresses are consecutive over consecutive threads, they are coalesced into a single transaction thereby maximizing memory bandwidth. Because the GPU does not have a large cache and shared memory capacity is only 16 kB per multiprocessor, efficient memory access patterns are essential to achieving high performance. Algorithms benefit from implementing data pipelines in shared memory with accesses structured to avoid bank conflicts among threads.

The GPU is heavily pipelined and performs floating point multiply and multiply-add with a latency of four cycles. In contrast, integer multiplication requires sixteen cycles. Computing addresses in global memory for every element may require many more cycles than actually performing the desired computation, and is avoided by choosing inexpensive address calculations and structuring the algorithm such that expensive calculations are invariant throughout a thread. Moreover, loop invariants must be explicitly hoist to the preamble of loops, as the CUDA compiler does not perform this automatically. Unrolling loops explicitly and modifying loop bodies to avoid loop-carried register dependencies contribute additional performance.

Performance Results

To demonstrate the performance of GPU VSIPL, an application was written to determine average runtime of VSIPL functions. By linking the application with first the TASP_VSIPL_CORE_Plus [5] library and then the GPU VSIPL library, performance results from each were obtained. The test platform is an Intel Core2 Q6600 at 2.4 GHz running Windows XP Professional with 2 GB of system memory. The GPU is an NVIDIA GeForce 8800 GTX with 768 MB of video memory.

GPU VSIPL demonstrated speedups for element-wise functions ranging from 20x to over 350x faster than the reference implementation depending on function and number of arguments. Other functions also demonstrated high speedup. Figure 1 illustrates GPU VSIPL's time-domain FIR filtering performance as filter length is varied from 16 elements to 256; the input vector length is 2^{20} elements. GPU VSIPL's speedup remains approximately 82x for all filter lengths. Figure 2 illustrates GPU VSIPL's FFT performance for input vectors of increasing size. Average speedup was 14.5x, yet the peak speedup observed for vectors of size 2^{20} and larger was 28.5x.

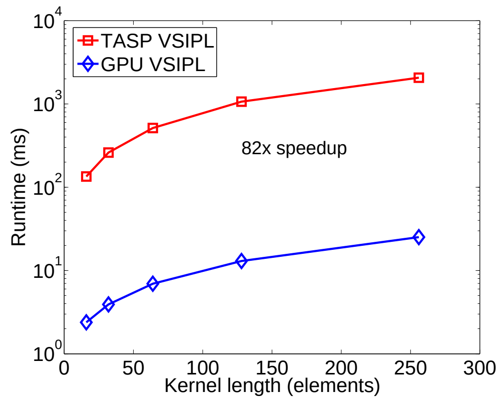

Figure 1: Time-domain FIR filtering runtime.

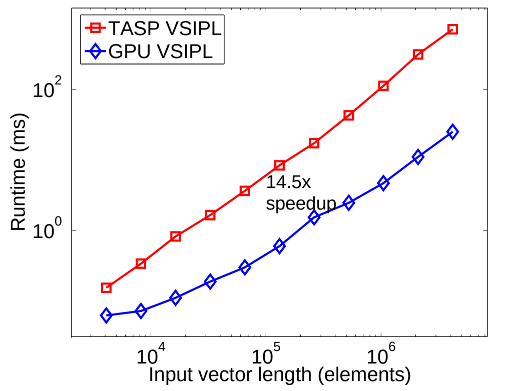

Figure 2: FFT runtime.

Conclusion

GPU VSIPL was developed to utilize the high performance available with modern GPUs and expose this performance to application developers in a platform-independent manner. The low price of GPUs and their high performance makes them a desirable architecture for high-performance computing. VSIPL provides memory and computing abstractions so that applications may perform high-level operations using an implementation optimized for a particular architecture.

References

[1] *NVIDIA CUDA Programming Guide 1.1*, `http://www.nvidia.com/object/cuda_get.html`

[2] *VSIPL 1.2 Specification*, `http://www.vsipl.org`

[3] *VSIPL Core Lite Profile Specification*, `http://www.vsipl.org/software`

[4] *VSIPL Core Lite Test Suite*, `http://www.vsipl.org/software`

[5] R. Judd, "TASP_VSIPL_CORE_PLUS," `http://www.vsipl.org/software`