Sourcery VSIPL++ for NVIDIA CUDA GPUs

Don McCoy, Brooks Moses, Stefan Seefeld, Mike LeBlanc, Jules Bergmann
CodeSourcery, Inc.
{don, brooks, stefan, mike, jules }@codesourcery.com

Introduction

Sourcery VSIPL++ for CUDA implements the open-standard VSIPL++ signal- and image-processing API on NVIDIA CUDA-compatible GPU processors. This presentation will describe the challenges in writing programs that target GPUs and some of the advantages of the VSIPL++ open-standard API for this purpose, the programming model used in supporting GPUs within Sourcery VSIPL++, and the performance results obtained with this implementation on the Scalable SAR synthetic benchmark application [5].

In particular, Sourcery VSIPL++ provides a way of writing applications at a high level that can be ported easily to new and diverse hardware, with relatively few source-code changes to take advantage of the particular specialties of that hardware. In the present high-performance embedded computing world, the expected lifetime of software is much longer than that of hardware. This presentation will demonstrate that Sourcery VSIPL++ for CUDA can be used to achieve significant performance improvements on realistic applications, and that code based on it can be readily ported to future generations of hardware.

CodeSourcery has released a preview version of the CUDA support in Sourcery VSIPL++ version 2.1. Work on this is ongoing; additional support will be included in version 2.2 and will be described in our HPEC 2009 presentation.

The Sourcery VSIPL++ Library

Sourcery VSIPL++ [1] is a C++ library for developing high-performance signal and image-processing programs in a portable manner using the VSIPL++ standard API [2]. It provides a standard interface for multi-dimensional array data, as well as common operations (FFTs, convolutions, linear-algebraic operations and solvers, elementwise operations, and so forth) on that array data.

In this interface, the logical representation of the data is separated from the physical representation in computer memory. To change an array from a row-major storage to a column-major storage, or even to distribute it across the memory of multiple processors operating in parallel, a developer need only change the portions of the code in which the arrays are declared; the remainder of the program remains unchanged.

Similarly, the syntax of common operations on this data is standardized independently of the implementation. This again separates the developer's code from the details of the implementation. The choice of hardware target, data layout, compilation options, and back-end library availability determines whether a given FFT call (for example) uses a back-end from NVIDIA's CUDA libraries, Intel's Integrated Performance Primitives (IPP), Mercury's Scientific Algorithm Library (SAL), the FFTW library, or CodeSourcery's Cell Math Library (CML) for the Cell/B.E. processor. Much of this dispatch logic occurs at compile-time, and thus incurs a minimal run-time cost.

This separation between the library syntax and implementation allows for a range of optimizations within the library. For example, in some cases, Sourcery VSIPL++ can recognize a sequence of an FFT, a vector elementwise multiplication, and an inverse FFT as a fast-convolution operation, and dispatch this to a combined function that executes faster than the components would individually. Similarly, sequences of elementwise operations can be fused into a single loop over the array elements, rather than individual loops for each operation.

Thus, applications based on Sourcery VSIPL++ can be readily ported to new architectures, and can take advantage of performance optimizations for those architectures with a minimum of reprogramming [5].

GPUs for High-Performance Computing

A recent trend in high-performance computing is the use of graphical co-processors (GPUs) for numerical computations. NVIDIA and AMD offer software development kits for performing numerical computations using off-the-shelf graphics cards, as well as coprocessor cards specifically intended for such use [3, 4].

GPUs provide a very attractive option for numerical computations. On appropriate types of computations, they can provide very high computational speed, with rates on the order of a teraflop (10^{12} floating-point operations per second) for single-precision calculations [3, 4]. In addition, GPUs typically have a very high-bandwidth connection to large amounts of coprocessor-local memory (for example, 4 GB of GDDR3 RAM with a connection speed of around 100GB/s, on a NVIDIA Tesla C1060 card [3]), which can be attractive for certain classes of bandwidth-limited applications.

However, obtaining good performance from GPUs is challenging. The programming model is very different from CPUs; a GPU consists of dozens or hundreds of compute units, each of which is hardware-multithreaded to execute dozens of copies of the same small set of instructions on elements of array data. As a result, full utilization of the GPU requires dividing the algorithm up into thousands of identical threads.

These threads are very lightweight and limited in capability compared to typical threads on CPUs. There is no direct inter-thread communication, and output from a thread is constrained to a specific indexed location within an output array. Conditional branches within the threads are typically supported only to a very limited extent; the level of nesting

is limited, and they are inefficient unless all threads take the same branch.

In addition, GPUs share data-locality challenges with other coprocessors. Although there is a large amount of fast local memory associated with the coprocessor, the data channel between this and the system memory is much slower; on the order of 4-6 GB/s (a factor of 20 less than the GPU-local memory bandwidth) in a typical system[1]. Thus, it is important to avoid extraneous data movement between the GPU and the host system's memory.

Sourcery VSIPL++ GPU Programming Model

The Sourcery VSIPL++ GPU programming model offers a range of GPU support. Unmodified VSIPL++ code can take advantage of GPU-accelerated functions simply by recompilation with the appropriate version of the library, and additional performance improvements can be obtained with minimal source-code changes.

Existing VSIPL++ data block types correspond to data stored in the system memory, and for these the GPU is used as a computational backend to accelerate individual VSIPL++ operations. Thus, this model is applicable to unmodified user code, but each VSIPL++ operation that executes on the GPU requires the library to move the data from the system memory to the GPU prior to the operation, and move it back afterwards.

Sourcery VSIPL++ for CUDA also introduces a new GPU-aware data block type, which represents an array that the library can store either in system memory or in the GPU's memory. This allows the library to only perform data movement when it is necessary. When sequential GPU operations are performed on data stored in a GPU-aware block, the data will remain in the GPU memory between the operations, and will only be returned to system memory when the CPU requires access to it. In order to use the GPU-aware block data storage within a VSIPL++ application, the only required changes are to the declaration of the relevant array objects.

As of this writing, we have included preliminary implementations of GPU operations in Sourcery VSIPL++ that are based on the CUDA FFT and BLAS backends. We anticipate expanding these to include optimizations and significant additional functionality, particularly with regards to fused operations, and will describe the details of that in our presentation.

Results Preview

The following results will be presented at HPEC 2009.

First, we will discuss the state of GPU support in available versions of Sourcery VSIPL++. In particular, we will mention the range of functions that are accelerated using the GPU, and provide performance data indicating the speedups that can be expected by using them and the conditions in which they are useful.

Second, we will demonstrate the performance of our GPU support on an existing VSIPL++ application — the Scalable SAR synthetic benchmark algorithm used in previous Sourcery VSIPL++ presentations [5] — and describe the performance improvements obtained with varying amounts of optimization to the existing code, as well as the code modifications required to implement those optimizations.

Conclusion

The VSIPL++ programming model allows programmers to easily port VSIPL++ programs to run on GPUs supported by Sourcery VSIPL++. Many of the accommodations required to obtain good performance on GPUs can be encapsulated within the library, such as usage of parallel functions and reducing data movement between the GPU and system memory.

References

[1] CodeSourcery, Inc. Sourcery VSIPL++. [online] http://www.codesourcery.com/vsiplplusplus.

[2] CodeSourcery, Inc. VSIPL++ Parallel Specification 1.0. Georgia Tech Res. Corp. 2005 [online] http://www.hpec-si.org.

[3] NVIDIA Corporation. NVIDIA Tesla C1060 Computing Processor Specifications. 2009. [online] http://www.nvidia.com/object/product_tesla_c1060_us.html.

[4] AMD, Inc. AMD Firestream 9270. 2009. [online] http://ati.amd.com/technology/streamcomputing/product_firestream_9270.html.

[5] J. Bergmann, M. LeBlanc, D. McCoy, B. Moses and S. Seefeld. Scalable SAR with Sourcery VSIPL++ for the Cell/B.E. HPEC Workshop Proceedings. 2008. [online] http://www.ll.mit.edu/HPEC/agendas/proc08/agenda.html.

[1] As measured on our NVIDIA Tesla C1060-based development system.