VFORCE: VSIPL++ for Reconfigurable Computing Environments

Albert Conti, Nicholas Moore, Miriam Leeser
{aconti, nmoore, mel}@ece.neu.edu
Dept. of Electrical and Computer Engineering
Northeastern University, Boston, MA

Laurie Smith King
lking@holycross.edu
Dept. of Computer Science and Mathematics
College of the Holy Cross, Worcester, MA

VSIPL++ (the Vector/Signal/Image Processing Library) is a collection of object-oriented interfaces to commonly used signal processing algorithms, such as the Fast Fourier Transform and FIR filters [1]. A reference implementation for VSIPL++ is available. Vendors can provide their own implementations that run efficiently on their platforms while still maintaining portability. Up until now, this approach has been limited to software implementations. However, many signal processing algorithms run much faster when implemented in reconfigurable hardware or on other heterogeneous processing elements.

The goal of this project is to make hardware implementations for signal processing algorithms seamlessly available to the VSIPL++ programmer. In VSIPL++, the programmer interacts with processing objects that realize algorithms in software. We extend this model by allowing a processing object to refer to a specialized hardware implementation. We interpose a hardware object that abstracts details of programming the actual hardware and transferring data. No detailed knowledge of the hardware is required of the programmer so development time decreases while code portability increases. Our approach works with many types of heterogeneous processing. In this abstract we focus on the FPGA platforms that we are targeting.

Project Design Goals

We are designing a framework that extends VSIPL++ to add support for special purpose hardware implementations. Our framework supports the following design goals:

- Existing VSIPL++ programs require only a small amount of modification to use hardware acceleration.
- Adding support for new hardware platforms is straightforward via a modular backend.
- Support for concurrency allows hardware and software functions to run in parallel.
- Hardware specific errors are masked; the VSIPL++ programmer does not need to write hardware specific exception handling code.
- Platforms with multiple processors and multiple FPGAs are supported.
- The framework is flexible enough to adapt to new developments in the VSIPL++ specification, such as support for parallel programming.

Processing Framework

Our framework is shown in Figure 1. In VSIPL++, the programmer interacts with processing objects that realize algorithms in software. In our framework, the programmer still works with processing objects, but in addition to running in software, the computation can also be run on a FPGA. This is facilitated by the introduction of hardware objects that encapsulate the hardware specific information and manage communication and control of the FPGAs. Every FPGA board has its own hardware class that is derived from the same virtual base class. The base class defines a common interface for all hardware objects. A particular FPGA board's hardware class contains all of the vendor and model specific information needed to communicate with its corresponding FPGA, including the necessary APIs and bitstream locations and characteristics. The FPGA box shown at the upper right in Figure 1 represents the reconfigurable hardware itself.

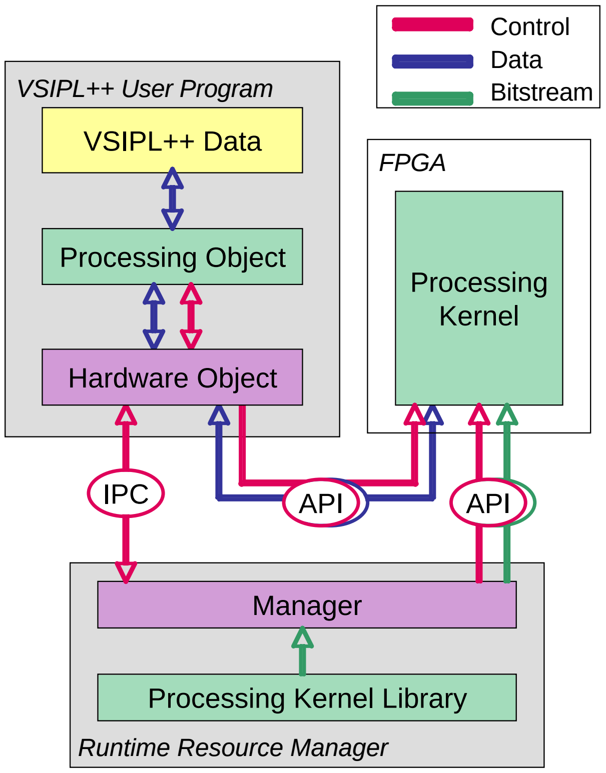

Figure 1: Processing Object Framework

Distributed Processing: Our framework incorporates a Run Time Resource Manager (RTRM) shown at the bottom of Figure 1. The RTRM exists as a separate program, either run on a separate CPU or as a distinct process in a multitasking environment. The RTRM maintains the pool of available reconfigurable hardware in a shared environment and distributes them to VSIPL++ applications upon request. The manager controls access to shared hardware and brokers the assignment of tasks to particular FPGA hardware. To minimize overhead, the manager is involved only during the request of processing resources and during initialization of the reconfigurable hardware. The RTRM is not involved during computation or data transfer. If no reconfigurable hardware is available that can perform the requested algorithm, the RTRM will respond appropriately so that the processing object can transparently default to performing the computation in software.

The RTRM encapsulates the platform specific information in a system, and keeps track of the resources available and their status. In addition, the RTRM may use knowledge of the environment, profiling, or any other means to make optimal decisions about hardware allocation.

Error Handling: The processing/hardware class hierarchy contains an exception handling mechanism. If there is an error when dealing with the FPGA, the processing object catches the error and will transparently default to running the given algorithm in software through the matching VSIPL++ function. The processing class will throw exceptions to the application programmer in the same situations that VSIPL++ would.

Concurrency: The VSIPL++ standard does not currently support concurrency. Our processing class contains the VSIPL++ method that blocks until the computation is complete to respect the VSIPL++ specification. In addition, three methods not present in standard VSIPL++ objects were added to achieve a level of concurrency. The first two new methods, *start* and *status*, are non-blocking and allow the software to initiate the FPGA computation and poll for completion respectively. The *finish* method blocks until it can retrieve the output of the FPGA computation. These methods permit but do not require a VSIPL++ programmer to take advantage of the performance increases that can be obtained when the CPU continues to work while an FPGA computation is running.

Current Status

Our initial work was a simplified version of the framework that did not include the resource manager. The goal was to demonstrate that the same application could easily be adapted to several different hardware platforms. This first version implements a master/slave model with a general purpose processor (GPP) acting as a master, and the FPGA hardware as the slave. The GPP provided the functionality of the VSIPL++ calling program as well as the RTRM.

Initially, a demo consisting of a 16-point FFT algorithm class was implemented using VSIPL++ software. Next, a hardware class and a 16-point FFT bitstream were developed for the Annapolis Wildcard II FPGA board [2], and the demo application was adapted to use the Wildcard II implementation.

A second hardware class was created to support a Mercury VantageRT FCN board class with a 16-point FFT bitstream. To use the Mercury VantageRT FCN board [3], the demo code only required changes to include the appropriate header and the instantiation of the Mercury VantageRT hardware object instead of the Annapolis Wildcard II hardware object. The hardware is requested explicitly by user code that passes the processing object as a constructor argument. A generalized FFT processing class was also developed. To match the use of VSIPL++ classes, the FFT processing class uses the same template parameters and includes all of the VSIPL++ FFT's methods. The hardware constructor differs in that it includes an extra argument: a handle to a specific FPGA.

Our new framework supports more general hardware models than the simplified version. Multiple GPPs and multiple FPGAs can now be supported in addition to the simpler, master/slave model. The user program no longer needs to request a specific hardware object, but requests a generic hardware implementation. Hardware classes for the Mercury MCJ6 FCN [3] and the Cray XD1 [4] are under development. For the Mercury hardware class, FPGAs are allocated by the RTRM on a first come, first served basis.

Future Work

The proposed processing framework has been shown to be effective for harnessing FPGA resources in an environment where one VSIPL++ program accesses a predetermined FPGA resource. We will continue to add more drop-in replacements for processing algorithms, as well as additional hardware classes and bitstreams. Hardware classes for the SGI RASC [5] and possibly the IBM/Sony/Toshiba Cell processor [6] are planned.

Future work also includes a more fully realized implementation of the RTRM and removing programmer involvement in hardware allocation altogether. Whereas today a few simple changes in headers and constructor calls are required to use a different hardware class, we plan to move towards using different hardware dynamically based on availability with no application level coding changes required. Requests to the RTRM could be serviced by returning a generic hardware pointer as well as a reference to a dynamically loaded library that contains the code needed to map the processing object requests onto the specific hardware API. This would achieve our goal of making the VSIPL++ code portable while providing high performance.

References

[1] http://www.hpec-si.org/

[2] http://www.annapmicro.com/wc2.html

[3] http://www.mc.com/products/boards.cfm?prodtype=boards

[4] http://www.cray.com/products/xd1/index.html

[5] http://www.sgi.com/products/rasc/

[6] J. A. Kahle, M. N. Day, et al., "Introduction to the Cell multiprocessor," IBM Journal of Research and Development, 49:4/5, 2005.