Vforce: Aiding the Productivity and Portability in Reconfigurable Supercomputer Applications via Runtime Hardware Binding

Nicholas Moore, Miriam Leeser
{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++ for Reconfigurable Computing (Vforce) is a middleware framework that extends VSIPL++ (a C++ extension of the Vector, Signal, and Image Processing Library) to include support for special purpose processors (SPPs) [1, 2]. VSIPL++[1] provides a collection of object-oriented interfaces to commonly used signal processing algorithms with the goal of aiding performance, portability, and productivity. While multiple VSIPL++ implementations exist, they all provide software only solutions that do not take advantage of special purpose hardware available on recent reconfigurable supercomputers. For some applications, this hardware can provide significant performance improvements over general purpose CPUs.

Vforce makes 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 single Generic Hardware Object that abstracts details of programming special purpose hardware and transferring data. The same GHO is called by the user for all types of hardware targeted. The combination of a GHO with Dynamically Linked Shared Objects (DLSOs) makes Vforce able to adapt to different types of hardware at run time, making it more dynamic. In addition, we present new results using Vforce on a Cray XD1.[2]

The Vforce Framework

As shown in Figure 1, Vforce consists of several components that work together to provide hardware abstraction. With both VSIPL++ and Vforce the application programmer uses processing objects that provide a software implementation of the given algorithm. However, each Vforce processing object also provides a second implementation that allows the algorithm to be run on a SPP. The SPP implementation utilizes Vforce's GHO which provides a common API for interacting with SPPs. This API provides for configuration, control, and data transfer for SPPs. When a particular algorithm is called by the user application, the GHO requests hardware capable of executing that algorithm from the Run Time Resource Manager. The RTRM exists as a separate process from the user application and manages and allocates the SPP resources available on any given machine.

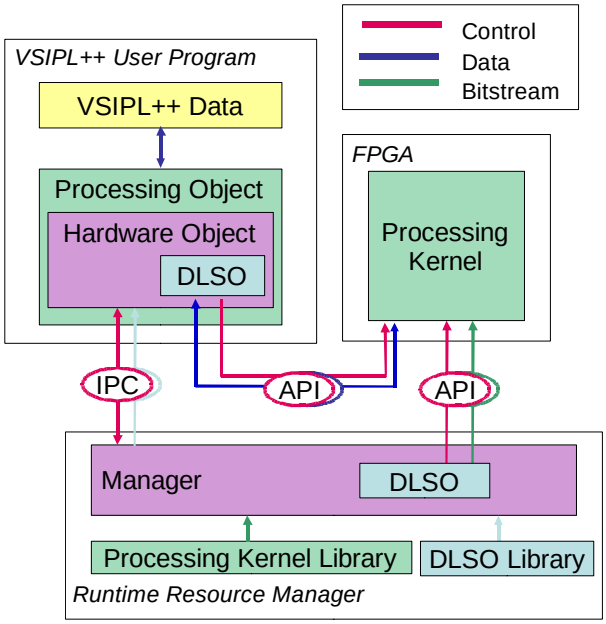

Figure 1: Processing Object Framework

Once the RTRM receives a request, it examines its library of kernels for a suitable implementation. The manager's reply contains two items: which hardware, if any, to use and which DLSO to use to control the specified hardware. The GHO will look in the DLSO for a specific set of functions that implement the API provided by the GHO. These functions call the SPP's native API completing the determination of low-level machine specific code from the abstract GHO interface. If there is an error at any part of this process, the processing object transparently defaults to the software implementation. Once the user application is done with the hardware, the GHO will return the SPP to the RTRM.

Vforce provides portability and runtime adaptability. Both the user application and processing objects can be developed without knowledge of a SPP's API, capabilities, or presence. The program is even compiled without this information, as the appropriate control code is loaded into the binary at run time, allowing Vforce applications to support heterogeneous or dynamically changing systems, as well as allowing the same application code to compile on multiple systems with minimal effort.

The framework is also extensible and enables high levels of code reuse. A DLSO only needs to be written once per SPP API, and can be shared by all applications accessing a specific type of SPP. The manager may require some machine specific behavior, but will only need to be written

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

[2] http://www.cray.com/products/xd1/

at most once per platform. In many cases, a generic manager that uses the same DLSOs as the user application can be used. Finally, to add individual algorithms, the software processing class only needs to be written once per algorithm and can be shared among platforms and applications. A hardware kernel created for the given algorithm must also be provided. These can be created by domain experts or through the use of high-level language compiler tools. The kernel can then be reused to accelerate many different programs. Vforce does not specify anything that needs to be implemented in hardware, which allows vendors and hardware designers to easily incorporate existing designs. The strength of Vforce lies in this reuse of code across platforms and SPP kernels across applications combined with freeing application programmers from dealing with low-level machine specific details.

Efforts have been made to minimize the performance impact of Vforce. The RTRM does not handle any data transfer, avoiding a potential bottleneck. The manager may perform initial configuration of an SPP, such as loading a bitstream onto an FPGA, but once an SPP is delivered to a user program all communication is direct between the user program and hardware. The manager keeps track of which kernels are loaded onto which SPPs and will favor pre-loaded SPPs in an attempt to minimize configuration overhead. In addition, data copying is kept to a minimum, but cannot always be eliminated due to the opaque nature of data blocks in VSIPL++.

Applications & Performance

To date, the main targets for Vforce have been the Cray XD1 and a Mercury 6U VME system. An FFT processing object has been developed that closely mimics the FFT specified by VSIPL++. The new FFT can be used as a drop in replacement for the VSIPL++ FFT while enabling SPP use. It also extends the VSIPL++ version by providing support for concurrent execution of the FFT on the SPP with other code on the CPU.

A Cray XD1 FFT bitstream was created using a Xilinx Coregen FFT kernel. The conversions between float and fixed point representation necessary to utilize the 24-bit fixed point FFT kernel and the final floating point scaling operation are handled by the Northeastern Univ. Vfloat library[3]. Additionally, the bitstream dynamically scales the fixed point representation to maximize precision when data with smaller dynamic ranges is used.

The FFT bitstream was used to study the overhead imposed by Vforce. Four scenarios were examined: 1) Using the native C API to use the FFT bitstream. 2) Using the Vforce stack to utilize the FFT bitstream. 3) Running the VSIPL++ software FFT. 4) Running the Vforce framework with no bitstream available so that the software fall-back FFT is run. For all four scenarios a wide variety of iterations and FFT size combinations were benchmarked. Comparing the first two cases looks at the overhead when dealing with hardware, while the second two cases examine the overhead of the software alone. The hardware comparison illustrated the overhead associated with copying data out of VSIPL++ views and into buffers of known organization. The impact of the data copying grows as the FFT data size grows. We are investigating ways to minimize this data copying. In the software comparison, the overhead due to Vforce compared to that using VSIPL++ is minimal. Differences in runtime of up to 3% were discovered.

A second application, developed for the Cray XD1 and Mercury 6U VME systems, is a 3D time-domain beamformer with adaptive weights. Beamforming was chosen because it provides an application where concurrent processing could be exploited. The weighted multiply accumulate step runs on the SPP while the weight update calculations are done on the CPU. The Mercury implementation, tested over a large combination of numbers of beams, sensors, and weight update periods, showed a speedup over the software only version ranging from 1.2 to over 200, with the best speedups coming in less realistic scenarios. The weight update computation completely dominated the total runtime. When considering the SPP implementation, the data transfer dominates.

The version of Vforce used for the Mercury implementation did not support non-blocking DMA transfer, limiting the amount of concurrent activity that could be supported. The newer Vforce version used for the XD1 beamformer does support non-blocking data transfers. The Cray XD1 implementation utilizes this by double buffering the input data sent to the FPGA, allowing the general purpose processor to compute weights while the FPGA DMA engine pulls new data to the FPGA at the same time as streaming back the results of the ongoing weighted multiply accumulate operation. However, the implementation does not double buffer the array parameter data, forcing the FPGA to pause for weight updates. Despite this, a larger level of concurrency is able to be achieved on the XD1 by hiding data transfers. In addition, the data transfer is relatively minor compared to the FPGA processing time, which allowed for more simultaneous processing by the FPGA and CPU.

Conclusions & Future Work

The Vforce framework supports portability across a variety of platforms and SPPs with minimal overhead. In the future, we plan to expand support to include more platforms, including GPUs, the Cell processor, and other reconfigurable supercomputers, as well as to develop more demo applications.

1. Moore, N., et al., *Vforce: An Extensible Framework for Reconfigurable Supercomputing.* Computer, 2007. **40**(3): p. 39-49.

2. Moore, N., et al., *Writing Portable Applications that Dynamically Bind at Run Time to Reconfigurable Hardware*, in *Field Programmable Custom Computing Machines*. 2007, IEEE.

[3] http://www.ece.neu.edu/groups/rcl/projects.html