Extending VForce to Include Support for NVIDIA GPUs using CUDA

Dennis Cuccaro, Nicholas Moore, Miriam Leeser
cuccaro.d@neu.edu, {nmoore, mel}@coe.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 adds support for special purpose processors (SPPs) to VSIPL++ [1], a C++ extension of the Vector, Signal, and Image Processing Library. VSIPL++ defines an object oriented API that provides a collection of commonly used signal processing algorithms and strives to enable performance, portability, and productivity.

Current implementations of VSIPL++ mainly provide support for software-only applications and do not take advantage of the growing number of SPPs available, including FPGAs and GPUs. For certain classes of applications these types of SPPs can provide significant performance improvements over general purpose CPUs.

Previous work has described in detail how VForce makes SPP implementations of signal processing algorithms seamlessly available to the VSIPL++ programmer in a portable way, as well as two applications used to characterize VForce [2, 3]. In this talk we will first summarize the VForce Framework, and then focus on our new work that extends VForce to support GPUs by making use of NVIDIA's CUDA environment. Results will be presented in the final abstract.

The VForce Framework

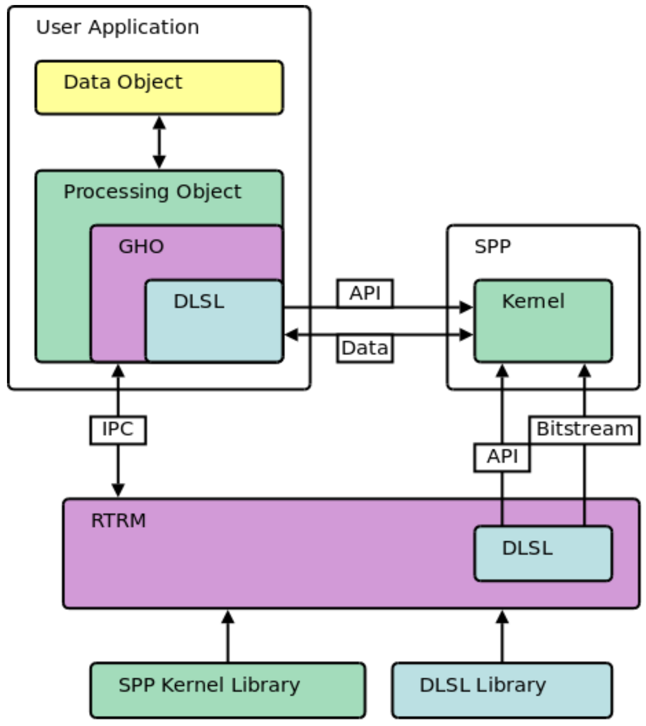

Figure 1: Conceptual Diagram of VForce

As shown in Figure 1, the VForce framework consists of several components that act in concert to provide all of the necessary hardware abstraction. In VSIPL++, the programmer interacts with processing objects that realize algorithms in software. VForce extends this model by allowing the creation of processing objects that interact with our Generic Hardware Object (GHO), which represents an abstract SPP. The GHO provides the processing object with a set of common SPP control methods, including methods for data transfer and kernel execution control, allowing the processing object to control hardware through a portable interface.

The current implementation of VForce does not bind to a specific piece of hardware and hardware API until runtime – no SPP specific code is compiled into the user application. At runtime, when the processing object requests that the GHO load an algorithm, the GHO contacts the Run Time Resource Manager (RTRM) through interprocess communication (IPC) to ask for an SPP that can execute the requested algorithm. The RTRM is a system service, running separately from the user application, that manages and allocates the SPP resources available on a given machine. Once the RTRM receives a request for hardware to execute a given algorithm, it searches through its SPP kernel library for a device that has a kernel matching the request. If a match is found, the RTRM will program the device, if applicable, and send a message over IPC to the requesting program indicating which piece of hardware and which Dynamically Linked Shared Library (DLSL) should be used to control the hardware.

VForce DLSLs all have a common set of functions, implementing a second interface that is lower-level than that provided by the GHO to the user application. A DLSL corresponding to a particular SPP uses the SPP-specific API necessary to implement the needed functionality. Once the GHO loads the correct DLSL, it uses this interface to control the designated SPP.

The net effect of these components is to provide portability and runtime adaptability. The user application and processing objects are written and compiled without knowledge of the target hardware and can be compiled and run anywhere VSIPL++ can be – including machines without SPPs or an RTRM – as VForce provides a mechanism for transparently transitioning to a software failsafe. The RTRM provides a hardware abstraction layer and the DLSLs link the abstract SPP interface to a specific SPP API.

In addition to portability, VForce emphasizes performance, and although VForce uses a run time manager, the RTRM is only involved during SPP request and relinquish. Once the GHO in the user application has loaded the correct DLSL, both control and data are direct between the user application and the SPP. Throughout the framework efforts have been made to minimize the overhead.

Finally, VForce is easily extensible, encourages code and kernel reuse, and helps to separate areas of expertise. SPP acceleration can be added to a new platform by providing a DLSL (generally only one is needed per platform) and the SPP kernels for algorithms to be accelerated. Processing objects and kernels can be used in multiple applications. The SPP kernels can be implemented by a domain expert or through the use of high-level tools, while application developers do not have to be as knowledgeable about low-level machine specific details.

VForce Performance Characteristics

Previous work includes two applications, an FFT and a beamforming application, built for and run on a Cray XD1 and a Mercury 6U VME system. The FFT application demonstrated portability as the user application code was identical. Overhead testing on the Cray XD1 revealed that data copying was a source of overhead when VForce was used to run the FFT on the XD1's FPGAs. The copying arose from a need to move data from opaque VSIPL++ views into data buffers that matched the requirements needed for DMA data transfer use. However, testing also revealed that there was negligible overhead compared to a pure VSIPL++ version of the FFT program when the VForce program relied on the software failsafe FFT. This was the case despite the extra VForce layers, including IPC with the manager, implying that the overhead of the framework was minimal when data copying is not necessary. This, combined with an identified but yet to be implemented mechanism for avoiding the data copy when using hardware, promises that VForce should be able to add minimal overhead when executing in software or hardware.

The beamforming application changed between the Cray and Mercury platforms to take advantage of features added to VForce after the Mercury version was completed. The main new feature, concurrent execution and data transfers, allowed the Cray XD1 beamformer to overlap software execution on the CPU with multiple FPGA operations: kernel execution, data transfer to the FPGA, and data transfer from the FPGA. For many platforms, utilizing the high level of concurrency enabled by VForce is necessary to obtain the largest speedups offered by many new parallel architectures.

Adding GPU Support to VForce

Our previous work had focused on platforms where the SPP was an FPGA. In order to demonstrate the generality of the approach taken by VForce for supporting SPPs, we have recently implemented support for GPUs to VForce using NVIDIA's CUDA environment. CUDA (Compute Unified Device Architecture) is a development environment that allows developers to write programs that can execute on a NVIDIA GPU in C with a few extensions. CUDA includes development tools and provides mechanisms for taking advantage of the large amount of parallelism available on recent GPUs.

Before an implementation was attempted, two models for algorithm execution on GPUs were considered. The first option was to treat GPUs similarly to the way VForce interacts with FPGAs. A single executable binary, a `cubin` file in the case of the CUDA tools, would be launched and interacted with based on only the specified GHO functions, forcing kernel developers to exactly match the single execution pattern implemented in the DLSL. While this method is straightforward and would result in relatively simple DLSL, it does not take best advantage of the characteristics of CUDA.

The second option was for the DLSL to provide a mechanism to load pre-compiled units of more general code written for the CUDA API. This option allows the individual library elements to execute an arbitrary amount of code that could take advantage of CUDA features like streams, launching multiple kernels, and other CUDA libraries like CUFFT and CUBLAS. This also allows library element designers more flexibility to implement algorithms by minimizing the constraints VForce places on GPU kernel designs – a general goal of the VForce project.

While both mechanisms can be useful in different scenarios, the second option was chosen for implementation due to its greater flexibility, and an FFT library element was created that uses the CUFFT library. The same VSIPL++ FFT test program used on the Cray XD1 compiles and runs unmodified on the target NVIDIA Tesla GPU hardware [4].

The first implementation of the CUDA DLSL does not provide a mechanism for the FFT library element to maintain state, requiring the creation and destruction of all of the CUFFT library elements with each use of the FFT, preventing the application from amortizing FFT setup overhead over multiple runs and resulting in a constant time of about 0.79 seconds per FFT iteration, which is slow. A second version of the CUDA DLSL is currently being debugged and will allow the library elements much more flexibility, including the ability to maintain the CUFFT data structures over multiple FFT iterations. Results from the second implementation will be presented at the workshop.

Conclusions & Future Work

We have demonstrated that VForce supports portability across several platforms and SPP types while introducing a very small amount of overhead. In the future we will work on improving the performance of the NVIDIA GPU support as well as improving the performance and flexibility of VForce. In addition, we plan to extend VForce to support other platforms, including other GPUs and the STI Cell processor, as well as to develop more demonstration applications.

References

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

[2] N. Moore, A. Conti, M. Leeser and L. Smith King, *"VForce: An Extensible Framework for Reconfigurable Supercomputing,"* **IEEE Computer**, pp. 39-49. March 2007.

[3] N. Moore, A. Conti M. Leeser and L. Smith King, *"Writing Portable Applications that Dynamically Bind at Run Time to Reconfigurable Hardware,"* **IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM),** pp. 229-238. April 2007.

[4] http://www.nvidia.com/object/tesla_computing_solutions.html