Two Nallatech FPGA Control Abstraction APIs: One Using A Home-Brew API and Later Using The OpenCL Interface

Craig Steffen
Innovative Systems Laboratory
NCSA/University Of Illinois Urbana/Champaign
csteffen@ncsa.uiuc.edu

Several vendors have produced compilers that translate high-level programming language code into low-level Hardware Design Languages (HDLs) (Verilog and VHDL) that are appropriate to create designs that will run in the fabric of a Field Programmable Gate Array (FPGA). These compilers enable researchers and domain scientists to translate inner loop kernel code from applications into something that will run on the FPGA without the burden of having detailed understanding of the hardware (for instance, at the look-up table level). The compilers allow the functional units of the FPGA as abstracted hardware execution units or arrays that follow special rules. As long as the rules are abstracted, the scientific programmer can couch the specialized code in the special high-level language and the compiler does the rest.

However, the outer logic of the programming harness that goes along with these compilers generally assumes the FPGA is the whole purpose of the program. The example program will be built with initialization calls for the FPGA and the API calls to initialize the FPGA, set its arguments, transfer data, and close down at the end. Somewhere in the program will be a section that says "insert user code here". Unfortunately, this model is backwards from the way that scientific collaborations construct their code. The outer layers of scientific applications are built to deal with data abstraction, calibration, file interpretation and configuration tasks. Then their inner layers will be the computational core that spends its time doing the computation that is important to the scientific work. Instead of a model of "insert your code here", which would mean inserting an enormous amount of code into the vendors example, a much more useful model would be one where the example "program" containing the FPGA is instead a library that lives behind a function call API.

Unfortunately, implementing such an API takes time, and requires the vendor to commit to delivering products with a certain function call interface. Additionally, any interface that would cover all the possible features different vendor's products and capabilities would be enormously complicated, but a single-vendor-only function would not be accepted by the rest of the field. Standards organizations traditionally step into this gap. The OpenFPGA organization[1] has an active effort to create an open FPGA software interface[2], but perhaps that effort is made too difficult by starting from too abstract a point of view. The end goal is a completely over-arching standard, but perhaps a reasonable starting point is to start with a non-overall API that just corresponds to one vendor's internal API, use that as a development base, and expand it to a generalized interface from there.

As far as what to target with a generalized interface, the fairly new OpenCL[3] specification provides an interesting answer. OpenCL is a parallel programming API consisting mostly of a parallel

1 HTTP://www.openfpga.org
2 HTTP://www.openfpga.org/pages/standards.aspx
3 HTTP://www.khronos.org/opencl/

programming language but it also contains an “OpenCL Platform Layer” and an “OpenCL Runtime” that allows the user program to configure, control, and feed data to hardware processors. The intention of OpenCL is to compile code using its parallel language and then execute that code on the hardware. So far, AMD has released hardware and a compiler stack compiles the OpenCL language both to its general-purpose CPU and its parallel GPU processors. An ideal situation would be for someone to create an OpenCL language to FPGA compiler along with an interface that corresponded to the OpenCL interface. However, until that happens, interface builders can take advantage of provisions within the OpenCL runtime environment to use and control *Native* computational kernels (that is, that are not compiled from the OpenCL language but are brought in from outside already compiled.
http://www.nallatech.com/Development-Tools/dime-c.html
This paper shows two separate APIs that the Innovative Systems Laboratory created to control FPGA processors from built by Nallatech[4], using images compiled with Nallatech's Dime-C[5] C-to-FPGA compiler and software stack. The first is an API built on top of the Dime-C/DimeTalk software stack available with the Nallatech H101 HPC FPGA accelerator. This API was created to purely mirror the functionality of the Nallatech software controls but with a purely C functional model. The second interface has been built on top of the Nallatech Abstraction Layer (NAL) available with their Front-Side-Bus Accelerator platform. Neither of these interfaces are meant to be end products in or of themselves. These are reported here to provide examples of a vendor-level API extended to the level of being user-friendly interfaces for application programming.

Proprietary “Nnalli” Interface

The NCSA/Nallatech Interface (NNALLI) API that ISL created was covered in a poster[6] at SAAHPC last year, but it is reviewed here for completeness. We defined a very simple set of interfaces that allowed the user program to load a bitstream, load scalar parameters and data to the FPGA, start the FPGA process running, check on its status and wait for it to finish, and then copy the results back out. Here is a list of functions in the Nnalli API:

```
nnalli_init_next_card()
nnalli_run()
nnalli_wait()
nnalli_write_integer_scalar_args()
nnalli_write_4byte_values()
nnalli_write_8byte_values()
nnalli_read_4byte_values()
nnalli_read_8byte_values()
```

These libraries manipulate a state machine within the library that keeps track of the state of the FPGA contains the corresponding control information. Before compiling their user code together with the nnalli library, the user must import the proper network.h file that comes from Nallatech's DimeTalk compiler software that contains the location of the bitfile and the definitions of the I/O ports in the FPGA processor image.

OpenCL-Based Accelerator Control Library

As discussed above, the OpenCL itself (which stands for “Open Computing Language) is mostly

4 HTTP://www.nallatech.com/
5 HTTP://www.nallatech.com/Development-Tools/dime-c.html
6 HTTP://saahpc.ncsa.illinois.edu/09/papers/Steffen_paper.pdf

designed around its parallel language to address both multi-core CPUs and massively parallel GPUs and similar hardware. The standard also contains functions that will control opaque (non-compilable) objects. We have created a library that sits behind the OpenCL API and configures and runs the Nallatech Front-Side Bus module[7], but the user-side application only ever sees the OpenCL function calls.

Our library controls and runs the FPGA using functions clCreateCommandQueue, clEnqueueNativeKernel, and clFlush, after making input data blocks available via memory object creation calls and clEnqueueWriteBuffer.

This library is currently in prototype stage and only implements the OpenCL functions that are specifically necessary for FPGA control, but ultimately this code will provide and entire functional OpenCL implementation that, on a machine with an FPGA accelerator of the appropriate type, will be able to accelerate algorithms without the calling application knowing what the processor type is. The goal of this process is to be able to write an application with certain key core computations performed through a set of OpenCL function calls. Then depending on what machine the code was compiled and linked on, the library that enabled the accelerator hardware available on the local machine would be linked against. So the same piece of code run on different machines with different accelerator types would compile and run the same but with different hardware accelerators, without the code being different. ISL does not currently have a machine available with the FSB module and another hardware accelerator, say, an GPU process. But the concept can be applied forward to heterogeneous computing.

Thanks to the National Science Foundation and the National Archives and Records Administration who provided the funding for the Nallatech hardware and this work via Cooperative Agreement NSF OCI 05-25308 and Cooperative Support Agreement NSF OCI 05-04064.

7 Craig Steffen, Gildas Genest, "Nallatech In-Socket FPGA Front-Side Bus Accelerator," Computing in Science and Engineering, pp. 78-83, March/April, 2010