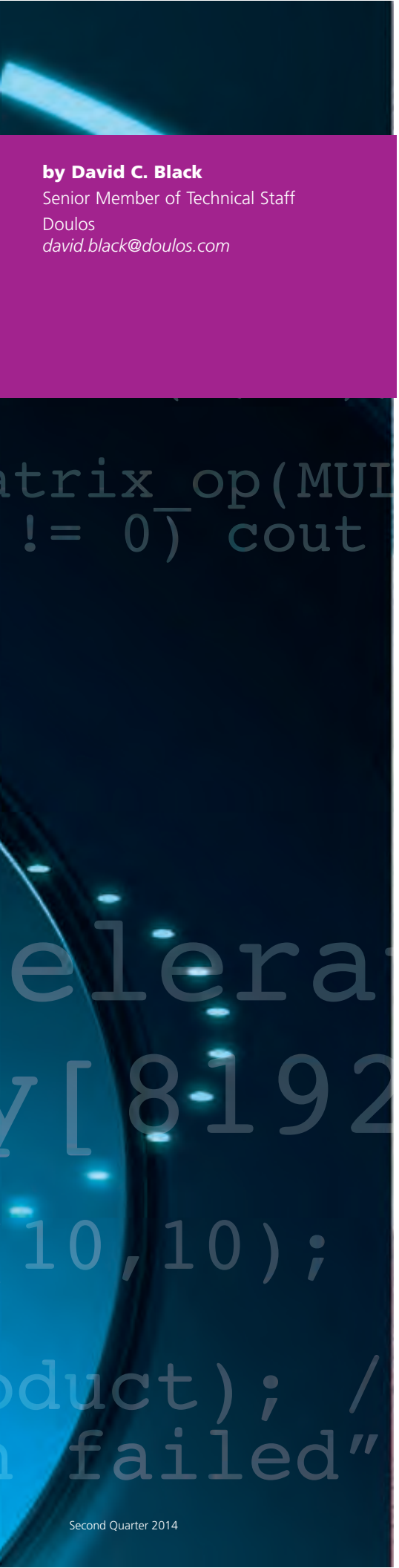
Make Slow Software Run Fast with Vivado HLS

Anyone plagued by code bottlenecks should explore the one-two punch of high-level synthesis and the Zynq SoC.

by David C. Black
Senior Member of Technical Staff
Doulos
david.black@doulos.com



Have you ever written some software that, despite your best coding efforts, didn't run as fast as desired? I have. Have you thought, "If only there were an easy way to put some of the code into multiple custom processors or custom hardware that wasn't so expensive"? After all, your application is one of many, and custom hardware takes time and money to create. Or does it?

I began rethinking this proposition recently when I heard about the Xilinx® high-level synthesis tool, Vivado® HLS. In combination with the Zynq®-7000 All Programmable SoC, which combines a dual-core ARM® Cortex™-A9 processor with an FPGA fabric, high-level synthesis opens up new possibilities in design. This class of tools creates highly tuned RTL from C, C++ or SystemC source code. Many purveyors of this technology exist, and the rate of adoption has been increasing in recent years.

So, how hard would it be to migrate some of that slow code into hardware, if indeed I could simply use Vivado HLS to do the more demanding computations? After all, I usually wrote my code in C++, and Vivado HLS used C/C++ as an input. The ARM processor cores meant I could run the bulk of my software in a conventional environment. In fact, Xilinx has even made available a software development kit (SDK) and PetaLinux for this purpose.

ARCHITECTURAL CONCERNS

As I started to think about this transformation from a software perspective, I grew concerned about the software interface. After all, HLS creates hardware dedicated to processing hardware interfaces. I needed something easy to access, like a coprocessor or hardware accelerator, to make the software go faster. Also, I didn't want to write a new compiler. To make it easy to exchange data with the rest of the software, the interface needed to look like simple memory locations where we could place the inputs and later read back the results.

Then I made a discovery. Vivado HLS supports the idea of creating an AXI slave with relatively little effort. This capability started me thinking an accelerator might not be so difficult to create after all. Thus, I found myself coding up a simple example to explore the possibilities. I was pleasantly surprised with how it turned out.

Let's take a walk through the approach I took and consider the results.

For my example, I chose to model a set of simple matrix operations such as add and multiply. I didn't want it to be constrained to a fixed size, so I would have to provide both the input arrays and their respective sizes. An ideal interface would put all the values as simple arguments to a function, such as the code in Figure 1.

The interface to the hardware would need to have a simple way to map the function arguments to memory locations. Figure 2 shows a memory layout to support this mapping. The registers would hold information about how matrices were laid out and what the desired operations would be. The *command* register would indicate which operation to do. This would allow me to combine several simple operations into one piece of hardware. The *status* register would simply be a way to know if the operation was in progress or had finished successfully. Ideally, the device would also support an interrupt.

Going back to the hardware design, I learned that Vivado HLS allows for array arguments to specify small memories. Thus, the functionality would be described with a function such as Figure 3 shows.

Assuming the ability to synthesize the AXI slave, how would this fit with the software? My normal coding environment assumes Linux. Fortunately, Xilinx provides PetaLinux, and conveniently PetaLinux provides a mechanism known as the User I/O device. UIO allows a simple approach to mapping the new hardware into user memory space, and provides the ability to wait for an interrupt. This means you avoid the awkward time and process of writing a device driver. Figure 4 illustrates the system.

```
Matrix operand1(5,10), operand2(10,5), product(10,10);
int status;
status = matrix_op(MUL, operand1, operand2, product); // product = operand1 * operand2;
if (status != 0) cout << "ERROR: multiplication failed" << endl;
```

Figure 1 – Example call to accelerator

Addr	Register name	Dir	Bits	Contents	
0	Matrix0_ptr	RW	32	Address of matrix 0 data	
4	Matrix0_shape	RW	32	Rows matrix 0	Cols matrix 0
8	Matrix1_ptr	RW	32	Address of matrix 1 data	
12	Matrix1_shape	RW	32	Rows matrix 1	Cols matrix 1
16	Matrix2_ptr	RW	32	Address of matrix 2 data	
20	Matrix2_shape	RW	32	Rows matrix 2	Cols matrix 2
24	Matrix3_ptr	RW	32	Address of matrix 3 data	
28	Matrix3_shape	RW	32	Rows matrix 3	Cols matrix 3
32	-reserved-	-	32		
36	-reserved-	-	32		
40	Command	RW	32	0	enum
44	Status	RW	32	0	enum
8192 x 32 memory					

Figure 2 – Register summary table

```
int Accelerator(int registers[16], int memory[8192]);
```

Figure 3 – Accelerator function API

There are of course a few drawbacks to this approach. For instance, the UIO device cannot be used with DMA, so you must construct matrices in the device memory and manually copy them out when done. A custom device driver in the future could address that issue if needed.

SYNTHESIZING THE HARDWARE WITH VIVADO HLS

Back to the topic of synthesizing the AXI slave. How difficult would this be? I found the coding restrictions to be quite reasonable. Most of the C++ language could be used with the exception of the dynamic allocation of memory.

After all, hardware doesn't manufacture itself during operation. This fact also restricts the use of the Standard Template Library (STL) functions, because they make heavy use of dynamic allocation. As long as the data remains static, most features are available. At first this task appeared onerous, but I realized it wasn't a huge deal. Also, Vivado HLS allows for C++ classes, templates, functions and operator overloading. My matrix operations could easily be wrapped in a custom matrix class.

Adding the I/O to create an AXI slave was easy. Simply add some pragmas to indicate which ports participate and what protocol they would use.

Running the synthesis tool was fairly easy as long as I didn't push all the knobs.

Running the synthesis tool was also fairly easy as long as I didn't push all the knobs. Figure 5 shows the overall steps involved, which I won't describe in detail here. Vivado HLS needs a bit of direction as to the target technology and clock speed. After that the process involved keeping an eye on the reports for violations of policy, and studying the analysis report to ensure Vivado HLS had done what I expected. Tool users need to have some appreciation for the hardware aspects, but technology classes exist to cover that issue. There is also the matter of running simulations both before and after synthesis to verify the expected behavior.

The Vivado IP Integrator made connecting the AXI slave into the Zynq SoC hardware a breeze, and removed concerns that signals would be hooked up incorrectly. Xilinx even has a profile for my development system, the ZedBoard, and IP Integrator exports data for the software development kit.

UNCLOGGING THE BOTTLENECKS

I am truly pleased with the results, and hope to do more with this chip-and-tool set combination. I have not explored all the possibilities. For instance, Vivado HLS also supports an AXI master interface. AXI would allow the accelerator to copy the matrices from external memory (although security issues might exist for this case). Nevertheless, I highly recommend that anyone looking at code bottlenecks in their software should look at this tool set. Ample training classes, resources and materials exist to enable a fast ramp, including those from Doulos. See *www.doulos.com* for more information.

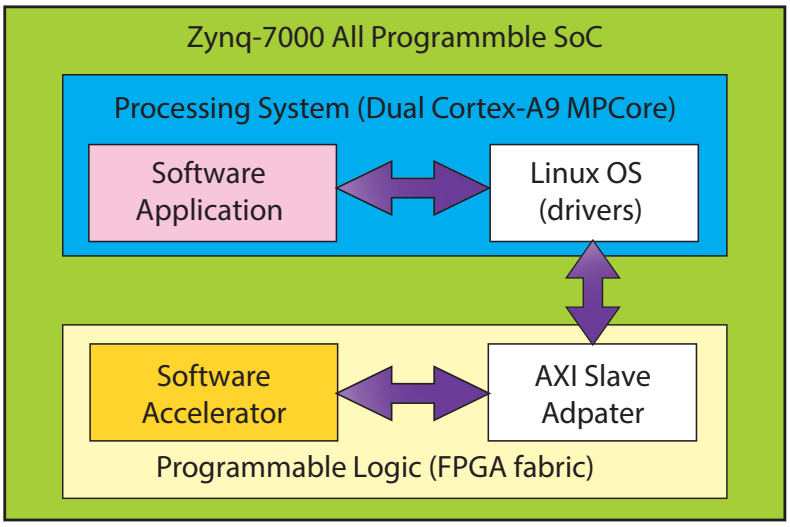


Figure 4 – System diagram

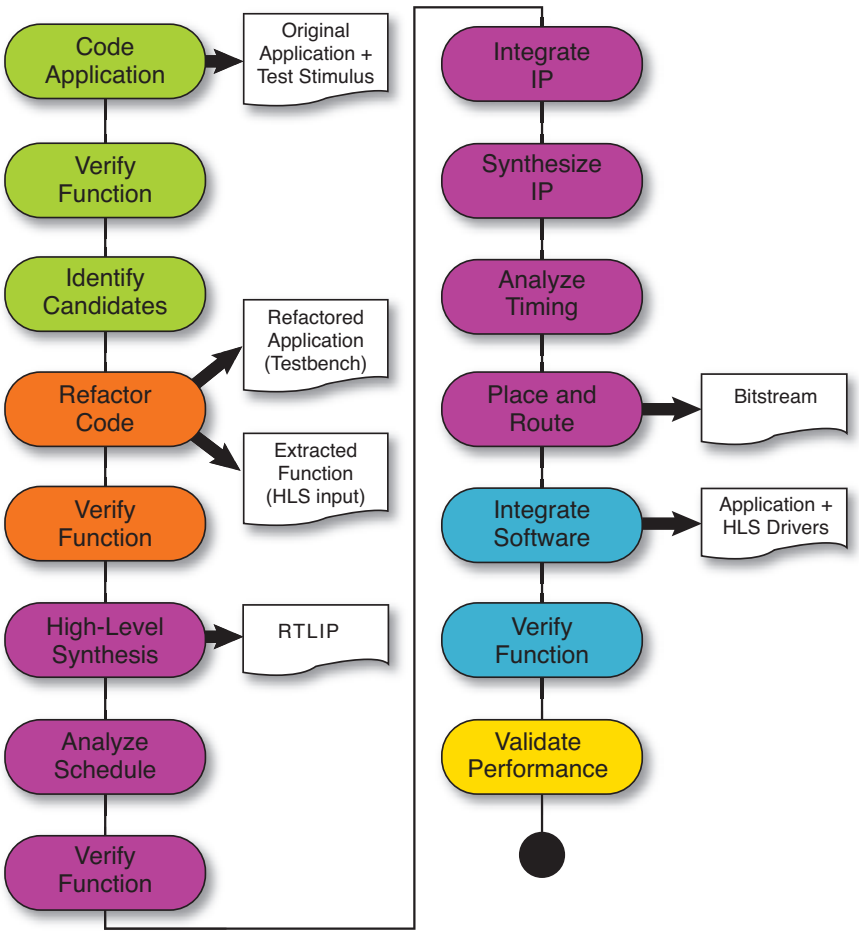


Figure 5 – Steps in design flow