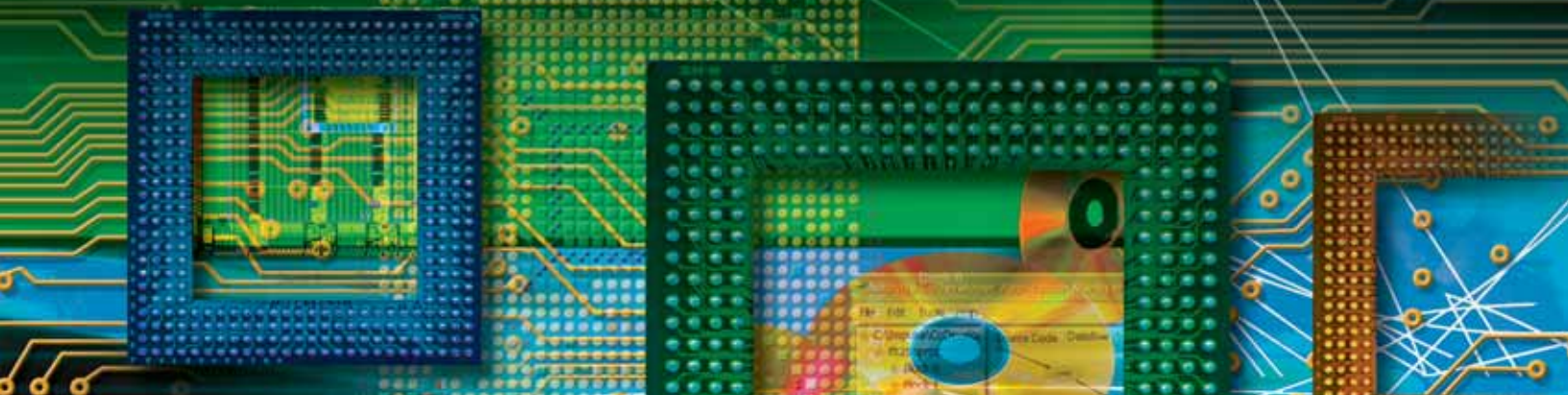


# Evaluating Hardware Acceleration Strategies Using C-to-Hardware Tools

Software-based methods enable iterative design and optimization for performance-critical applications.



by David Pellerin  
CTO

Impulse Accelerated Technologies, Inc.  
[david.pellerin@impulsec.com](mailto:david.pellerin@impulsec.com)

Scott Thibault, Ph.D.  
President

Green Mountain Computing Systems, Inc.  
[thibault@gmvhdl.com](mailto:thibault@gmvhdl.com)

Language-based ESL tools for FPGAs have proven themselves viable alternatives to traditional hardware design methods, bringing FPGAs within the reach of software application developers. By using software-to-hardware compilation, software developers now have greater access to FPGAs as computational resources.

What has often been overlooked by traditional hardware designers, however, is the increased potential for design exploration, iterative performance optimization, and higher performance when ESL tools are used in combination with traditional FPGA design methods.

In this article, we'll describe the role of C-to-hardware ESL tools for iterative design exploration and interactive optimization. We'll present techniques for evaluating alternative implementations of C-language accel-

erators in Xilinx® FPGAs and explore the relative performance of fixed- and floating-point FPGA algorithms.

## Crossing the Abstraction Gap

Before ESL tools for FPGAs existed, it was necessary to describe all aspects of an FPGA-based application using relatively low-level methods such as VHDL, Verilog, or even schematics. These design methods are still adequate, and in many cases preferable, for traditional FPGA-based hardware applications. However, when using traditional hardware design methods for creating complex control or computationally intense applications, a significant gap in abstraction (Figure 1) can exist between the original software algorithm and its corresponding synthesizable hardware implementation, as expressed in VHDL or Verilog. Crossing this gap may require days or weeks of tedious design conversion, making iterative design methods difficult or impossible to manage.

Tools providing C compilation and optimization for FPGAs can help software and hardware developers cross this gap by providing behavioral algorithm-level methods of design. Even for the most experienced FPGA designers, the potential exists for design improvements using such tools.

Although it may seem counterintuitive to an experienced hardware engineer, using higher level tools can actually result in higher performance applications because of the dramatically increased potential for design experimentation and rapid prototyping.

## Iterative Optimization Is the Key

To understand how higher level tools can actually result in higher performance applications, let's review the role of software compilers for more traditional non-FPGA processors.

Modern software compilers (for C, C++, Java, and other languages) perform much more than simple language-to-instruction conversions. Modern processors, computing platforms, and operating systems have a diverse set of architectural characteristics, but today's compilers are built in such a way that you can (to a great extent) ignore these many architectural features. Advanced optimizing compilers take advantage of low-level processor and platform features, resulting in faster, more efficient applications.

Nonetheless, for the highest possible performance, you must still make programming decisions based on a general understanding of the target. Will threading an application help or hurt performance?

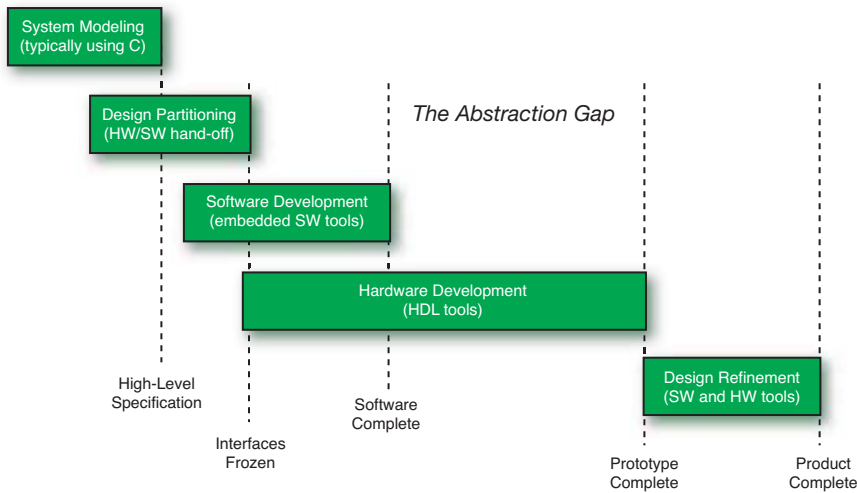


Figure 1 – Crossing the abstraction gap: in system-level design, hardware development can be the bottleneck when creating a working prototype.

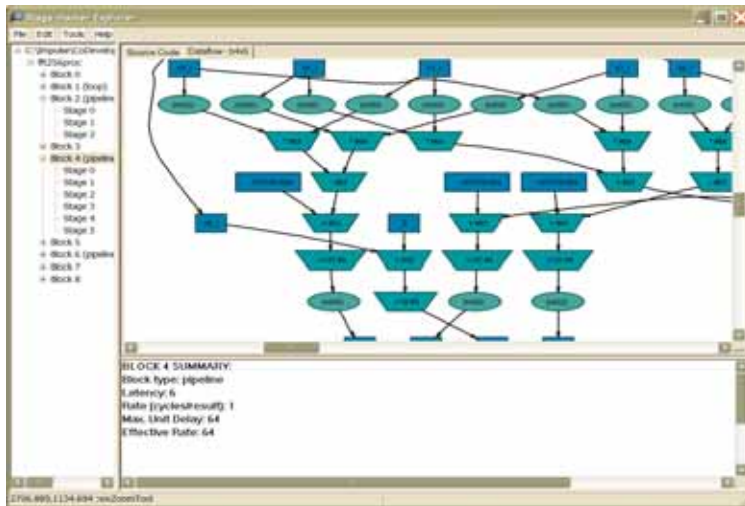


Figure 2 – A dataflow graph allows C programmers to analyze the generated hardware and perform explorative optimizations to balance trade-offs between size and speed. Illustrated in this graph is the final stage of a six-stage pipelined loop. This graph also helps C programmers understand how sequential C statements are parallelized and optimized.

Should various types of data be stored on disk, maintained in heap memory, or accessed from a local array? Is there a library function available to perform a certain I/O task, or should you write a custom driver? These questions, and others like them, are a standard and expected part of the application development process.

For embedded, real-time, and DSP application developers there are even more decisions to be made – and more dramatic performance penalties for ignoring the realities of the target hardware. For these platforms, the ability to quickly experiment with new algorithmic approaches is an

important enabler. It is the primary reason that C programming has become an important part of every embedded and DSP programmer's knowledge base. Although most embedded and DSP programmers understand that higher performance is theoretically possible using assembly language, few programmers wish to use such a low level of abstraction when designing complex applications.

What does all of this mean for FPGA programmers? For software engineers considering FPGAs, it means that some understanding of the features and constraints of FPGA platforms is critical to

success; it is not yet possible to simply ignore the nature of the target, nor is it practical to consider legacy application porting to be a simple recompilation of existing source code. For hardware engineers, it means that software-to-hardware tools should be viewed as complements to – not replacements for – traditional hardware development methods. For both software and hardware designers, the use of higher level tools presents more opportunities for increasing performance through experimentation and fast prototyping.

Practically speaking, the initial results of software-to-hardware compilation from C language descriptions are not likely to equal the performance of hand-coded VHDL, but the turnaround time to get those first results working may be an order of magnitude better. Performance improvements can then occur iteratively, through an analysis of how the application is being compiled to the hardware and through the experimentation that C-language programming allows.

Graphical tools like those shown in Figure 2 can help provide initial estimates of algorithm performance such as loop latencies and pipeline throughput. Using such tools, you can interactively change optimization options or iteratively modify and recompile C code to obtain higher performance. Such design iterations may take only a matter of minutes when using C, whereas the same iterations may require hours or even days when using VHDL or Verilog.

### Analyzing Algorithms

To illustrate how design iteration and C-language programming can help when prototyping algorithms, consider a DSP function such as a fast Fourier transform (FFT) or an image-processing function such as a filter that must accept sample data on its inputs and generate the resulting filtered values on its outputs. By using C-to-hardware tools, we can easily try a variety of different implementation strategies, including the use of different pipeline depths, data widths, clock rates, and numeric formats, to find a combination of size (measured as FPGA slice count) and speed (measured both as clock speed and data throughput) that meets the specific requirements of a larger hardware/software system.

To demonstrate these ideas, we conducted a series of tests with a 32-bit implementation of the FFT algorithm. The FFT we chose includes a 32-bit FIFO input, a 32-bit FIFO output, and two clocks, allowing the FFT to be clocked at a different rate than the embedded processor with which it communicates. The algorithm itself is described using relatively straightforward, hardware-independent C code. This implementation of the FFT uses a main loop that performs the required butterfly operations on the incoming data to generate the resulting filtered output.

Because this FFT algorithm is implemented as a single inner loop representing a radix-4 butterfly, we can use the automatic pipelining capabilities of the Impulse C compiler to try a variety of pipeline strategies. Pipelining can introduce a high degree of parallelism in the generated logic, allowing us to achieve higher throughput at the expense of additional hardware. This pipeline itself may be implemented in a variety of ways, depending on how fast we wish to clock the FFT.

Another aspect to this algorithm (and others like it) is that it can be implemented using either fixed- or floating-point math. For a high-end processor such as an Intel Pentium or AMD Opteron, the choice of floating point is obvious. But for an FPGA, floating point may or may not make sense given the lack of dedicated floating-point units and the corresponding expense (in terms of FPGA slice count) of generating additional hardware.

Fortunately, the Impulse C tools allow the use of either fixed- or floating-point operations, and Xilinx tools are capable of instantiating floating-point FPGA cores from the generated logic. This makes performance comparisons relatively easy. In addition, the Impulse tools allow pipelining for loops to be selectively enabled, and pipeline size and depth to be indirectly controlled through a delay constraint.

Table 1 shows the results of two different optimization and pipelining strategies for the 32-bit FFT, for both the fixed- and floating-point versions. We generated results using the Impulse C version 2.10 tools in combination with Xilinx ISE™

FFT Data Type	Pipelining Enabled?	Slices Used	FFs Used	LUTs Used	Clock Frequency	Total FFT Cycles
Fixed Point	No	3,207	2,118	5,954	80 MHz	1,536
Fixed Point	Yes	2,810	2,347	5,091	81 MHz	261
Floating Point	No	10,917	7,298	20,153	88 MHz	10,496
Floating Point	Yes	10,866	7,610	19,855	74 MHz	269

Table 1 – 32-bit FFT optimization results for fixed- and floating-point versions, with and without pipelining enabled

Stage Delay Constraint	Slices Used	FFs Used	LUTs Used	Clock Frequency	Pipeline Stages
300	1,360	1,331	2,186	60 MHz	5
200	1,377	1,602	2,209	85 MHz	7
150	1,579	2,049	2,246	99 MHz	9
100	1,795	2,470	2,392	118 MHz	11
75	2,305	3,334	2,469	139 MHz	15

Table 2 – 16-bit image filter optimization results for various pipelining and stage delay combinations

software version 8.1, and targeting a Xilinx Virtex™-4 LX-25 device. Many more choices are actually possible during iterative optimization, depending on the goals of the algorithm and the clocking and size constraints of the overall system. In this case, there is a clear benefit from enabling pipelining in the generated logic. There is also, as expected, a significant area penalty for making use of floating-point operations and a significant difference in performance when pipelining is enabled.

Table 2 shows a similar test performed using a 5x5 kernel image filter. This filter, which has been described using two parallel C-language processes, demonstrates how a variety of different pipeline optimization strategies (again specified using a delay constraint provided by Impulse C) can quickly evaluate size and speed trade-offs for a complex algorithm. For all cases shown, the image filter data rate (the rate at which pixels are processed) is exactly one half the clock rate.

For software applications targeting FPGAs, the ability to exploit parallelism (through instruction scheduling, pipelining, unrolling, and other automated or manual techniques) is critical to achieving

quantifiable improvements over traditional processors.

But parallelism is not the whole story; data movement can actually become a more significant bottleneck when using FPGAs. For this reason, you must balance the acceleration of critical computations and inner-code loops against the expense of moving data between hardware and software.

Fortunately, modern tools for FPGA compilation include various types of analysis features that can help you more clearly understand and respond to these issues. In addition, the ability to rapidly prototype alternative algorithms – perhaps using very different approaches to data management such as data streaming, message passing, or shared memory – can help you more quickly converge on a practical implementation.

## Conclusion

With tools providing C-to-hardware compilation and optimization for Xilinx FPGAs, software and hardware developers can cross the abstraction gap and create faster, more efficient prototypes and end products. To discover what C-to-hardware technologies can do for you, visit [www.xilinx.com/esl](http://www.xilinx.com/esl) or [www.impulsec.com/xilinx](http://www.impulsec.com/xilinx). 