FPGA-based hardware acceleration of C/C++ based applications - Part 3


Editor's Note: There are a lot of folks who are interested in accelerating their algorithms/programs written in C or C++. Many of these guys and gals are aware that FPGA-based accelerators are available, but they don't know how you actually make these little rascals perform their magic.

In order to address this, I contacted a number of the main players in this arena and asked each of them if they would be interested in penning an article that explained the process of:

1. Writing a new application (or modifying a legacy application) in C or C++ in a form suitable for acceleration.
2. Partitioning the application such that some portions will be compiled for use on the general-purpose processor and other portions will be implemented in the FPGA.
3. Actually getting those portions of the application that are to be accelerated into the FPGA by one means or another.
4. Interfacing the main application running on the general-purpose processor with those portions running on the FPGA.
5. Analyzing/profiling (debugging?) the new version of the program, part of which is running on the FPGA.

Some of the companies I contacted declined because they were too busy (which is a good place for them to be), but three stepped up to the plate:
- Part 1: DRC Computer Corporation
- Part 2: SRC Computers
- Part 3: Impulse Accelerated Technologies (this article)

Field Programmable Gate Arrays (FPGAs) are increasingly being used as platforms for embedded and high-performance computing. FPGAs can be used to deploy complete, single-chip accelerated applications, or can be used as coprocessors in larger, multiple-CPU server applications, in areas as diverse as Image Processing, Bioinformatics, and Financial Computing.

In this article, we'll show how emerging tools for software-to-hardware compilation are speeding the development of high-performance, FPGA-accelerated applications. We'll describe some of the many ways in which software-to-hardware tools can be deployed, and present two examples of performance-critical algorithms that have been implemented in FPGAs using these new tools.

The FPGA as a coprocessor
FPGAs are best known as devices for hardware integration. Hardware designers have for many years used FPGAs for logic applications including state machines, memory controllers, "glue" logic and bus interfaces. More recently, however, embedded and high performance system designers have begun using FPGAs as actual computing elements. This has been made possible in part because of increased device densities, but also by advances in FPGA tool flows.

As dedicated coprocessors, FPGAs have significant performance advantages over traditional processors due to their massively parallel architectures. Hardware-level parallelism allows FPGA-based applications to operate at 100× or more the performance of an equivalent application running on an embedded processor, and 10× or more the performance of a higher-end workstation processor.

When measured as a function of computational power efficiency, the advantages of an FPGA-based computing strategy become even more apparent. Calculated as a function of millions of operations (MOPs) per watt, FPGAs have demonstrated greater than 1,000× power/performance advantages over today's most powerful processors. And that advantage – the processing efficiency gap - continues to grow. For this reason, FPGA accelerators are now being deployed for a wide variety of power-hungry computing applications.

The adoption of FPGAs for high-performance computing applications has been slowed, however, by a historic lack of FPGA software-to-hardware compilers. Embedded systems programmers, financial algorithm developers, domain scientists and other application programmers have been reluctant to use FPGAs due to a lack of familiar programming tools.

Recently, however, a new generation of software-to-hardware tools has emerged. These tools greatly simplify the task of moving software algorithms into FPGA hardware, putting these devices within the reach of software developers. Automated and semi-automated compiler and optimization tools now make it possible for software developers to quickly prototype, optimize and implement hardware accelerators using traditional C programming techniques. One of the leading tools in this area is Impulse C, from Impulse Accelerated Technologies. The Impulse C tools include a C-to-hardware compiler as well as a set of C-compatible API functions that can be used by software programmers to create hardware-accelerated applications. This article describes how Impulse C can be used for a wide variety of such applications.

**Considering FPGAs for application acceleration**

FPGAs have come a long way since their inception, as illustrated in Fig 1. From their humble beginnings as containers for glue and control logic, FPGAs have evolved into highly capable software coprocessors, and as platforms for complete, single-chip embedded systems.
1. FPGA devices have evolved to become highly capable computing platforms.

It has long been recognized that many of the computing challenges in embedded and high-performance computing can be addressed using parallel processing techniques. The use of dual- or quad-core processors, multiple computer "blades", or clustered PCs has become commonplace in many different application domains. FPGAs are now being deployed alongside traditional processors in these systems, creating what might be called a hybrid multiprocessing approach to computing.

When FPGAs are added to a multiprocessing environment, opportunities exist for improving both application-level and instruction-level parallelism. Using FPGAs, it is possible to create structures that can greatly accelerate individual operations, such as a simple multiply-accumulate or a more complex sequences of integer or floating point operations, or that implement higher-level control structures such as loops. Code within the inner-most loops of an algorithm can be further accelerated through the use of instruction scheduling, instruction pipelining and other techniques. At a somewhat higher level, these parallel structures can themselves be replicated to create further degrees of parallelism, up to the limits of the target device's capacity.

The programming of software algorithms into FPGA hardware has traditionally required specific knowledge of hardware design methods, including the use of hardware description languages such as VHDL or Verilog. While these methods may be productive for hardware designers, they are typically not suitable for embedded systems programmers, domain scientists and higher level software programmers.

Fortunately, software-to-hardware tools now exist that allow software programmers to describe their algorithms using more familiar methods and standard programming languages. For example, using a C-to-FPGA compiler tool, an application and its key algorithms can be described in standard C with the addition of relatively simple library functions to specify inter-process communications. The critical algorithms can then be compiled automatically into HDL representations which are subsequently synthesized into lower level hardware targeting one or more FPGA devices. While a certain level of FPGA knowledge and in-depth hardware understanding may still be required to optimize the application for the highest possible performance, the formulation of the algorithm, the initial testing and the prototype hardware generation can now be left to a software programmer.

Using standard C for application development has many advantages, not the least of which is the opportunity to use iterative, software-oriented methods of design optimization and debugging. With the Impulse C tools, for example, both hardware and software elements of the complete application can be described, partitioned and debugged using standard C programming tools such as GCC and GDB or environments such as Microsoft Visual Studio. During this process, the application
programmer can make use of familiar C-code optimizations to increase performance without having FPGA-specific hardware knowledge. **Parallelism is the key**
The C language is not a parallel programming language, and yet FPGAs are parallel processing devices unlike any other compilation target. How is it possible that C can be used to program these devices? The answer lies in C-compatible parallel programming methods combined with compilation and optimization technologies that can automatically detect and exploit parallelism at a lower-level, for example at the level of individual subroutines and inner code loops.

In support of parallel programming for FPGA targets, Impulse C provides a standardized, multi-process programming model that supports the creation of highly parallel applications. A program written using the Impulse C APIs can be compiled and run on a variety of FPGA-based multiprocessing targets including single-FPGA platforms and platforms that combine high-end processors with FPGA accelerator modules. The streaming, shared memory and message-passing API functions provided with Impulse C allow C-language parallel processes to be described, debugged and implemented, allowing an FPGA-based application to be developed entirely in C.

Impulse C is aimed at developers of high-performance embedded systems, as well as developers of server and enterprise computing applications. The goal of Impulse C is to improve the computing performance of software applications by making it easier to harness the power of the FPGA, without the need for hardware design expertise.

**The Impulse C programming model**
A core concept behind Impulse C is a programming technique called the communicating process programming model. Impulse C provides multiple methods of communicating between parallel software and hardware processes, including stream processing. Stream processing has similarities to other parallel processing methods including single-instruction-multiple-data, or SIMD processing. But where SIMD processors use single instructions to operate on vectors, streams-oriented programming takes advantage of more flexible computing processes (in this case written in C) that operate on multiple streams of data, as shown in Fig 2.

2. A streams-oriented programming model employs multiple processes to increase parallelism.

In Impulse C, an input stream can be thought of as a buffered communications channel that carries packets of data from one process to another. Multiple processes described in this way operate in parallel (for example as a pipelined system of filters in a DSP application) and may themselves contain lower-level parallelism in the form of pipelined loops or other parallel computations. In a typical Impulse C process, there is an infinite loop (a C-language "do" loop) that iteratively processes packets of data as they stream in from other processes. Other methods of process-to-process communication are also provided in Impulse C, including shared memories and message passing.
signal interfaces.

The multi-process programming model of Impulse C has been explicitly designed for applications requiring a high degree of parallelism. Taking full advantage of FPGA-based platforms does require some understanding of parallel programming, and an awareness of how data dependencies and pipelining will impact overall system throughput, but these skills are well within the abilities of experienced software developers.

**Multiple uses for Impulse C**

FPGA-based platforms exist in a wide variety of forms, and users of C-to-hardware tools have a wide range of application requirements. While it is difficult to characterize a typical user of C-to-FPGA tools, there are three common modes in which these tools are used today. These three modes are module generation, embedded acceleration, and host acceleration. These three modes of use are illustrated in Fig 3.

3. C-to-FPGA tools can be used in many ways, including module generation, embedded acceleration, and host processor acceleration.

**C-to-FPGA module generation - a fast path to hardware**

The simplest use for C-to-FPGA tools is for hardware module generation. In this mode of use, one or more C-language subroutines are compiled directly into VHDL or Verilog code that can then be combined with other hardware modules. These other hardware modules may have been hand-written using HDL or obtained from third parties. Fig 4 illustrates this common usage of C-to-FPGA tools.
4. Using a C-to-hardware compiler as a module generator.

Module generation assumes that the user of the C-to-FPGA compiler is an experienced FPGA hardware developer who will combine the automatically generated hardware and its generated I/O ports with other hardware IP modules. This combining of IP might be done at the level of HDL, or via lower-level FPGA netlists during the place-and-route process. One example of this mode of use is illustrated in Fig 5, in which a DSP filter developed using C-to-hardware compilation is combined with other filters and I/O components developed using more traditional methods.

5. Combining C-language and other modules to create a complete, multi-process hardware application.

Because module generation assumes a certain level of hardware design expertise, users of C-to-hardware tools in this category are primarily interested in the productivity gains made possible by
the tools. While these users may have the skills to hand-code and hand-optimize all parts of the system using HDL, they also recognize that using faster, more iterative design methods and C-to-hardware module generation for selected components may result in a more efficient, better tested solution at the system level.

**Accelerating embedded processors**

In many, perhaps most, electronic product design teams there are both hardware and software designers working to develop a complete, integrated product. Software developers work with familiar C-language tools that might include source-level debuggers and cross-compilers targeting embedded microprocessors and DSPs. On the hardware side of the project, developers who have deep knowledge of FPGA and/or ASIC design flows work to create low-level descriptions of low-level control logic and I/O interfaces as well as designing more complex hardware algorithms.

With the advent of FPGA-embedded processors such as the Xilinx MicroBlaze and PowerPC processors, embedded design teams are now finding that a common design environment for both hardware and software is highly desirable. FPGA-specific development tools such as Xilinx Platform Studio can help to manage most aspects of peripheral selection and system bring-up for FPGA-based embedded applications. Board support packages compatible with Platform Studio can also simplify this process, resulting in a working hardware/software prototype, using provided hardware peripheral cores, in a matter of minutes. But the creation of custom hardware acceleration cores – a specialized DSP filter, for example – requires either hand-coding in HDL, or the use of C-to-FPGA tools.

For development and prototyping, widely available FPGA development boards can be used to create complete systems-on-an-FPGA. These single-chip systems might include one or more embedded soft processors, processor peripherals and associated C-language hardware accelerators. In this mode of use, the Impulse C compiler can be thought of as a peripheral generator that accepts C-language inputs and automatically creates bus-connected accelerator cores for one or more MicroBlaze soft processors. Development boards such as the Xilinx ML410 can be used in a similar manner to generate hardware-accelerated applications using on-chip PowerPC processors.

To simplify and streamline the development flow for embedded processor applications Xilinx provides a set of tools and libraries collectively known as the Embedded Development Kit, or EDK. *Fig 6* illustrates the design flow using Impulse C in combination with the Platform Studio tools provided by Xilinx as part of the EDK tools.

![Diagram](image_url)

6. *Using C-to-FPGA tools in combination with Xilinx Platform Studio to create a software/hardware*...
When using this design flow to create embedded application accelerators, the Impulse C compiler serves as a peripheral generator, using platform-specific knowledge and automatically creating all necessary bus interface wrappers associated with the generated hardware, as illustrated in Fig 7. Using such a design flow, it is practical for a software engineer - one who has experience with embedded systems but not FPGA design - to specify, generate and bring-up a complete, hardware-accelerated application, with no need to write HDL for any part of the system.

7. Accelerated MicroBlaze embedded application developed using C-to-hardware tools.

Applications for this type of hybrid software/hardware embedded application include DSP, video processing, robotics, radar processing, secure communications, and many others. Accelerating server applications
A third mode of use, and one that is getting increased attention in recent years, is the use of C-to-FPGA tools to create processor accelerators for high performance server applications, as shown in Fig 8. In this mode of use, the C-to-FPGA compiler flow is similar to the flow described for embedded processor acceleration, but rather than generating a tightly coupled embedded processor peripheral core, the tool generates an accelerator that will be connected - possibly at run-time - to a host processor via a standard processor bus.
8. A hybrid CPU+FPGA accelerated server application.

Fig 9 illustrates the design flow using C-to-FPGA tools to create a hot processor accelerator. In this usage, note that there is assumed to be a software-side API library that supports communication across the system bus, which may be PCI, PCI Express, HyperTransport, Cardbus, FSB or some other standard interface.

9. Using C-to-FPGA tools in combination with host-side development tools to create a hybrid CPU+FPGA accelerated server application.

From a design tool perspective, this mode of use is very similar to that of embedded acceleration, with one important difference: while embedded acceleration examples normally involve a static, always present processor accelerator, host acceleration applications may involve the dynamic loading and unloading of accelerator modules as needed for specific applications being executed on the host.

Applications that are excellent candidates for FPGA acceleration include financial analytics, seismic imaging, bio-informatics, plasma modeling and other scientific computing applications.

Iterative optimization is critical to success
While it is tempting to think that C-to-hardware compilation can be a trivial, push-button process, the reality is that FPGAs are very different types of devices than traditional processors. Compiler tools have improved dramatically in recent years, but it remains important that users of C-to-FPGA tools gain enough knowledge about the underlying device architecture to make good C coding decisions. C-to-FPGA compiler vendors provide many examples and guidelines to help with these
design decisions.

An important technique when using these tools is design iteration. In many of the applications being developed with Impulse C, for example, automatic pipelining and parallelizing of inner code loops plays a key role. Because the compiler does not know everything about your design goals, it is important to use the various reports generated during the entire design flow, including compiler pipeline and instruction scheduling information, synthesis resource estimates, and the timing reports generated during FPGA place-and-route, to determine whether there are improvements that can be made at the level of the C source code. By observing published C coding guidelines and working with the interactive aspects of the tools (as shown in Fig 10) it is possible to explore dozens or even hundreds of different loop optimizations in a matter of minutes.

10. Interactive pipeline optimization tools can help to increase overall system performance and efficiency.

In fact, this ability to experiment, and to iterate on an algorithm at a high level, is what makes C-to-hardware so compelling. While it may be true that an experienced HDL programmer can, given unlimited time, produce results superior to those achieved by today’s C-to-FPGA compilers, it is also true that most projects have a finite development window. C-to-FPGA tools such as Impulse C provide a significant productivity boost, and provide new, more efficient ways of creating FPGA-based algorithms.

The complete, iterative C-to-hardware compilation process can be summarized as follows:
1. Describe the complete application (software and hardware) in C language and use a standard C debugger such as Microsoft Visual Studio to verify the algorithm and create a baseline for testing. During this process, legacy C code may be employed either for direct comparison purposes (for example as part of a C code validation test bench) or as the basis for hardware-targeted algorithms.

2. Partition the C application, assigning critical subroutines to specific Impulse C hardware processes. Standard C profiling tools may be used during this phase to help with partitioning decisions. Impulse C API functions are introduced at this point to describe process-to-process communication.

3. Use automated and interactive optimization tools to analyze and improve the performance of the hardware-accelerated functions. During this phase, use your standard C compiler and debugger of choice to verify that correct algorithm behavior is maintained, and to debug multi-process communications.

4. Use the Impulse C compiler to create synthesizable hardware representing the hardware-accelerated processes, in the form of automatically generated HDL files (either VHDL or Verilog). If desired, make use of HDL simulation tools to verify bit-accurate and cycle-accurate behavior and compare to software-only baseline results.

5. Use the Impulse C interactive optimizer to view and iteratively experiment with different C-level optimization strategies. If necessary, return to the original C code to introduce fixed-width data values, modify the C code statements or adjust optimization settings via C code pragmas. If the application includes pipelined loops, use the pipeline graph to explore different pipelining strategies for higher throughput.

6. Export the generated HDL files to the FPGA optimization and mapping tools. Use synthesis reports to determine clock speeds and resources. If needed, use this information to again make iterative adjustments in the C code or to refine the optimization settings, for example to improve pipeline throughput rates and increase clock speeds, or to balance logic resource requirements.

7. If there is an embedded processor involved (for example being used as an embedded test generator), export the software side of the application to the FPGA platform tools, such as Xilinx Platform Studio, for cross-compilation onto the embedded processor.

8. Download the resulting FPGA bitmap and any software-side executable code to the target FPGA device or board-level platform.

9. Run the application on the target FPGA.

**FPGAs for random number generation**

Random number generation is a critical part of many high-performance computing applications, including statistical modeling and certain types of financial analysis algorithms. Researchers using random number generators require good quality, evenly distributed numbers, and may require that random numbers be generated at a very high rate. For developers of hardware-accelerated simulation algorithms, such as in the domain of financial computing, arbitrage, and weather modeling, a fast and efficient hardware random number generate is a critical component of the system.

The Mersenne Twister is a pseudorandom number generator described in 1998 by Makoto
Matsumoto and Takuji Nishimura of Keio University in Yokohama, Japan. The algorithm is on a matrix linear recurrence over a finite binary field, and provides fast generation of very high quality pseudorandom numbers, with a very large period.

At Pico Computing in Seattle, a Mersenne Twister random number generator has been implemented using a Xilinx Virtex-5 LX50 device. This random number generator, described using Impulse C, uses ten parallel C-language processes that act as independent Mersenne Twister generators. The C code representing one of the ten parallel processes is shown in Fig 11 (Mersenne Twister random number generator (inner code loop) described using Impulse C).

In the source code shown, the `co_stream` data types and related `co_stream_open`, `co_stream_read`, `co_stream_write`, and `co_stream_close` functions are provided as part of the Impulse C library, and are used to represent streaming data. These ten streaming processes feed a collector process, also written in C, that manages the movement of the random numbers to a host computer, as shown in Fig 12.

12. Ten random number generation processes are used to feed a collector process.

To allow for fast prototyping and performance analysis, the Pico Computing E-16 card shown in Fig 13 was used to implement and test the random number generator application. The E-16 card is a very small form-factor card (smaller, in fact, than a credit card) that can be clustered to create a low-power, yet high-performance computing platform. For this test, a single E-16 was plugged into the CardExpress slot of a standard Windows-based notebook computer. Two tests were run to evaluate the performance of the generated hardware. In the first test, the performance of the FPGA-based accelerated random number generator was compared to the performance of an equivalent C-language algorithm running on the PC as a software application. This test showed that the random number generator performance, while impressive in terms of numbers generated per estimated watt of power, was limited by the throughput limitations of the single-lane PCIe interface being used to transfer data from the FPGA to the host PC.
As second test was then run with the data transfer to the PC disabled, better emulating the environment in which the random number generator would likely be used (feeding random numbers to other hardware processing elements). In this test the results were quite dramatic, with random numbers being generated at a rate substantially greater than that of the host processor, but at approximately 1/18 the clock rate and with orders of magnitude lower power consumption.

After the initial testing had been performed, a series of iterative optimizations were performed, including an analysis of the generated pipeline structures, which led to some re-coding of the C-language statements to achieve a faster clock rate and more efficient hardware. The final performance numbers are shown in Fig 14.

<table>
<thead>
<tr>
<th>Processor/FPGA</th>
<th>Clock Rate</th>
<th>Numbers per second</th>
<th>FPGA slices (Virtex-5 LX50)</th>
<th>Acceleration (vs. 2 GHz PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium desktop PC</td>
<td>2 GHz</td>
<td>81 M</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>FPGA, 1 process with host I/O</td>
<td>111 MHz</td>
<td>37 M</td>
<td>1622 (5%)</td>
<td>.46X</td>
</tr>
<tr>
<td>overhead</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPGA, 20 processes with host I/O</td>
<td>111 MHz</td>
<td>40 M</td>
<td>17,426 (60%)</td>
<td>.49X</td>
</tr>
<tr>
<td>overhead</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPGA, 20 processes without host</td>
<td>111 MHz</td>
<td>740 M</td>
<td>17,426 (60%)</td>
<td>9.1X</td>
</tr>
</tbody>
</table>

14. Acceleration results for random number generation, using Pico E-14 card, with and without I/O overhead.
This particular example demonstrates that I/O throughput will have a significant impact on the performance gains that can be achieved with FPGA acceleration. In this case, the FPGA card being used for testing limits the data throughput to around 160 MB per second, which equates to a maximum of 40 million 32-bit values. By disabling the transfer of results data to the host and allowing the random number generator to run at-speed, we can see that the combination of multiple parallel processes (in this case 20) and automatic pipelining of the random number generator processes results in an impressive 9× performance increase relative to a desktop PC.

**Accelerating bioinformatics searches**

Biomedical computing is another domain in which FPGA acceleration can be applied. Whether for the purpose of new drug discovery and therapies, or for gaining a fundamental understanding of the human genome, computationally intensive search and analysis algorithms play critical roles.

In the domain of bioinformatics, molecular biologists employ a computing technique known as Multiple Sequence Alignment (MSA) to compare and identify specific regions within protein families. The computing of MSAs requires significant computing resources. Computing MSAs on traditional desktop processors using progressive alignment methods may require many hours of processing for just a few hundred sequences.

The most common method of accelerating these comparisons is to use large-scale computing clusters. These clusters, however, are not efficient in terms of power usage, and do not scale linearly in performance as the size of the input data grows. The rapid growth of biological sequence databases, coupled with the need for faster analysis of genetic data, has caused biologists to seek faster ways to compute MSAs. FPGA-based acceleration is one way to obtain the required speedups.

At Nanyang Technological University in Singapore, researchers Yan Lin Aung, Douglas Leslie Maskell, and Timothy Francis Oliver used FPGA-based platforms and the Impulse C compiler to achieve dramatic speedups of MSA calculations, using less power than would be required using clustered CPUs.

When calculating MSAs on the FPGA, a method of generating local alignment scores was developed that accepts the following inputs:

- Protein sequences for optimal local alignments
- A pre-defined substitution matrix
- A Gap Opening Penalty Value
- A Gap Extension Penalty Value

With these inputs, optimal local alignment scores could be generated for the input protein sequences being analyzed.

**Design method**

As described earlier, there are many possible ways to use Impulse C, depending on the type of software/hardware platform is being targeted. In this example, the algorithm developers made the decision to begin initially with the implementation of a single MSA distance matrix calculation kernel, and to use an embedded processor within the FPGA that would serve as an in-system test data generator. By using an embedded Xilinx MicroBlaze processor and a Xilinx ML403 development board, the researchers were able to perform rapid prototyping and generate actual performance measurements, without the need to manually create hardware description language (HDL) code for any part of the system.
After compilation from C-language to HDL, the Xilinx FPGA mapping, placement and routing tools were then invoked to convert the generated HDL into FPGA bit streams. The bit stream was then loaded into the FPGA before computation was started. On the software side of the application, a C code test application was cross-compiled onto the embedded MicroBlaze processor, and loaded into the FPGA along with the hardware bitmap.

**Increasing performance through iterative optimization**

For initial testing, the first stage of the MSA application was implemented together with the embedded MicroBlaze processor serving as an embedded test bench and communicating with the MSA hardware via a Fast Simplex Link (FSL) bus interface. Both the MicroBlaze processor and the MSA application were clocked at 100 MHz on the Virtex-4 FX12 device. Using this configuration, sequences with lengths of around 500 proteins were compared, and the relative performance of the MicroBlaze processor and the FPGA were compared. The initial speedups showed that the generated, non-optimized hardware would run at a rate 77 times faster than the MicroBlaze. Although this was a significant speed increase over the embedded MicroBlaze process, the performance of the first prototype hardware did not exceed that of a desktop processor, in this case a 2 Ghz Pentium processor system being used for baseline testing. Nonetheless the results were promising, particularly in light of the small amount of FPGA resources required for the generated hardware, and the correspondingly low power consumption of the FPGA implementation.

After these initial results, a series of iterative optimizations were performed. First, Impulse C compiler pragmas were used to pipeline and optimize the inner code loops of the algorithm. This optimization, which required changes to only a few lines of C source code, resulted in more than doubling the speed of the algorithm. Other optimizations related to C coding styles, in particular the combining of otherwise redundant loops, resulted in additional gains. Through a process of optimization and re-verification that required a few days of work, the performance of the single-process algorithm was increased dramatically, resulting in an implementation that easily exceeded the performance of the desktop processor, while using a smaller amount of FPGA resources.

Most importantly, throughout these optimizations the C source code was maintained and validated in a hardware-independent manner; the C code does not make specific reference to clocks and resets, or make reference to cycle-by-cycle behaviors. This example therefore represents an untimed, behavioral method of programming for FPGAs. In fact, because Impulse C is fully compatible with standard C development tools and compilers, the same source code can be compiled and debugged, along with additional C code representing a software test bench, using standard C tools.

After a single MSA distance matrix acceleration process had been developed and tested using Impulse C, it was a relatively simple matter to add additional processing elements (PEs) within the pipelined inner code loop. The C source code for the resulting inner code loop is shown in Fig 15 (Inner code loop of the MSA search algorithm, consisting of two pipelined processing elements (PEs)).

This allowed the Impulse C implementation of the MSA distance matrix algorithm to perform as many as eight simultaneous comparisons in a single pipelined process.

Fig 16 shows the performance results achieved through this iterative optimization process, using a single Impulse C pipelined process as well as multiple pipelined processes.
<table>
<thead>
<tr>
<th>Processor/FPGA</th>
<th>Clock Rate</th>
<th>Execution time (seconds)</th>
<th>FPGA slices (Virtex-4 LX25)</th>
<th>Accel. (vs. Micro-Blaze)</th>
<th>Accel. (vs. 2 GHz PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MicroBlaze with on-chip block RAM</td>
<td>100 MHz</td>
<td>41.3</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Pentium desktop PC</td>
<td>2 GHz</td>
<td>.454</td>
<td></td>
<td>91X</td>
<td>1</td>
</tr>
<tr>
<td>FPGA, 1 PE, no pipeline optimizations</td>
<td>100 MHz</td>
<td>.538</td>
<td>1369 (12%)</td>
<td>77X</td>
<td>.84X</td>
</tr>
<tr>
<td>FPGA, 1 PE with pipeline optimizations</td>
<td>100 MHz</td>
<td>.209</td>
<td>1376 (13%)</td>
<td>198X</td>
<td>2.2X</td>
</tr>
<tr>
<td>FPGA, 2 pipelined PEs</td>
<td>100 MHz</td>
<td>.145</td>
<td>1509 (14%)</td>
<td>235X</td>
<td>3.1X</td>
</tr>
<tr>
<td>FPGA, 8 pipelined PEs</td>
<td>100 MHz</td>
<td>.102</td>
<td>4229 (33%)</td>
<td>405X</td>
<td>4.5X</td>
</tr>
<tr>
<td>FPGA, 2 processes, each with 2 pipelined PEs</td>
<td>100 MHz</td>
<td>.074</td>
<td>5118 (47%)</td>
<td>558X</td>
<td>6.1X</td>
</tr>
</tbody>
</table>

16. Acceleration results for MSA distance matrix algorithm using single and multiple pipelined C process.

The table in Fig 16 also shows the impact of creating multiple parallel accelerators, in this case two parallel instances of the 2-PE pipelined process. The results of this algorithm partitioning (illustrated in Fig 17) demonstrate that parallelism can take many forms, including multiple processes, multiple pipelined PEs within a single loop, and lower-level automatic parallelizing of individual C-language statements.
17. Using two pipelined processes increases the system-level performance with a near-linear increase in performance and size.

As these results demonstrate, the use of additional PEs within the automatically pipelined inner code loop substantially increased the performance of the algorithm, with a corresponding increase in hardware resource requirements. By adding a second parallel hardware process (for a total of 4 pipelined PEs), performance was increased yet again, with a nominal increase in hardware resources.

The example has shown that a heavily pipelined, parallel algorithm can be written entirely in C. Through iterative optimization techniques, the algorithm was boosted to more than 6× the performance of a desktop processor, at 1/20 of the clock rate and with correspondingly lower power consumption.

Note that adding additional accelerators (implemented using multiple parallel instances of the pipelined C-language process) and using multiple FPGAs for acceleration would result in further increases in performance. In fact, by employing multiple FPGA devices, it would be entirely practical to create an FPGA computing cluster of hundreds, or even thousands, of distinct MSA search processes. By using multiple FPGAs in a cluster, MSA searches that previously required many hours to complete on a desktop workstation could be performed in just a few minutes, at very little increase in overall power usage.

Summary
The above examples demonstrate how C-language compiler tools can help with the rapid creation of hardware accelerators for performance-critical applications. As we have seen, creating an efficient hardware-accelerated application does require a certain amount of experience with parallel programming and iterative pipeline optimization. Tools such as Impulse C, however, bring the needed skills well within the reach of software application developers.

Acknowledgements
The authors would like to thank Greg Edvenson of Pico Computing, and Yan Lin Aung, Douglas Leslie Maskell, and Timothy Francis Oliver of Nanyang Technological University in Singapore for their assistance with this article.

Authors
David Pellerin is co-founder and CTO of Impulse Accelerated Technologies, and is the author of five books on the subject of programmable logic, FPGAs and related technologies, most recently including Practical FPGA Programming in C (Prentice Hall, 2005).

Ed Trexel is a Senior Applications Engineer at Impulse. He has extensive experience with high-performance embedded applications including image processing, VOIP, and FPGA-based systems. Ed is an electrical engineering graduate of the University of Colorado.

Mei Xu is an Applications Engineer at Impulse. She has experience with FPGA-based hardware/software and embedded systems including CDMA wireless and network infrastructure. Mei holds a Masters Degree in Electrical Engineering and Computer Science from the University of California at Berkeley.