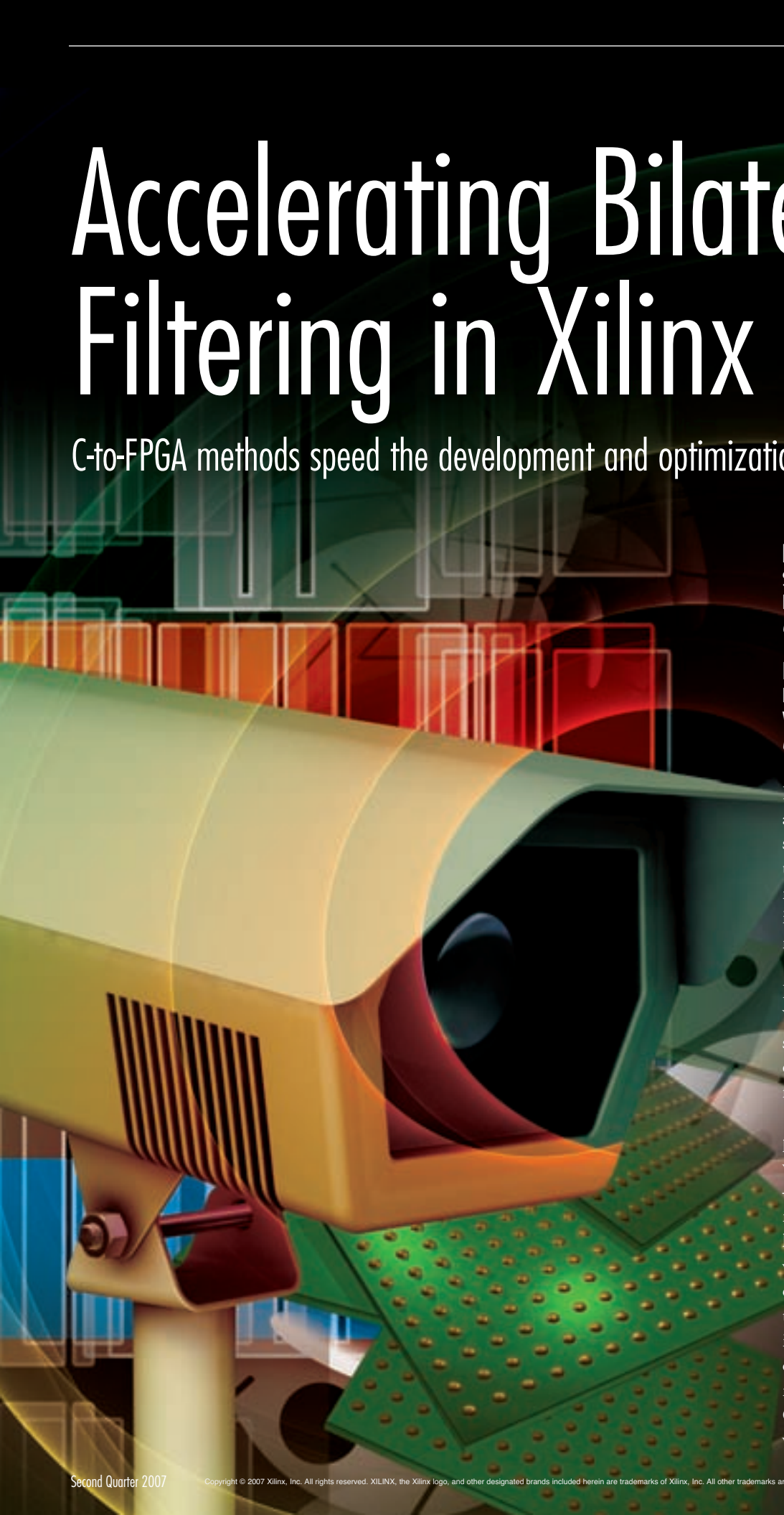
Accelerating Bilateral Filtering in Xilinx FPGAs

C-to-FPGA methods speed the development and optimization of complex filters.



by Ed Trexel
Senior Applications Engineer
Impulse Accelerated Technologies, Inc.
edward.trexel@impulsec.com

Dov Stamler
Hardware Engineer
Vigilant Technology Ltd.
dovs@vigilanttechnology.com

Bilateral filtering is used in many video applications to smooth images while preserving edge detail. One such application uses a bilateral filter to reduce random noise during video pre-processing and to improve video compression efficiency in a real-time video environment.

Vigilant Technology provides intelligent IP surveillance and security solutions. Vigilant currently supports tens of thousands of cameras in airports, government offices, financial institutions, correctional facilities, casinos, and town centers.

In this article, we'll describe how one portion of an advanced closed-circuit television (CCTV) system – the bilateral filter – has progressed from an initial software model to being implemented in actual hardware using C-to-FPGA tools. Starting with a verified C-language model, we developed a hardware implementation through the use of C-to-hardware tools and iterative design methods. To allow for quick prototyping and algorithm experimentation, we used common and familiar C-language tools for debugging and software/hardware design analysis.

Copyright © 2007 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

Iterative hardware-level testing was an important part of this conversion. We accomplished this testing with an FPGA-embedded software test bench. For this purpose, the embedded processor features of the Xilinx® Virtex™-4 FX device were employed for direct, hardware-in-the-loop embedded testing of the image filtering algorithm.

Bilateral Filtering

The bilateral filter is representative of many other critical image algorithms. It works like this: for each pixel, a new value is calculated using those pixels immediately around the pixel of interest, using the equation:

$$\text{New pixel} = \frac{\sum_n f_n \times coeff_n}{\sum_n coeff_n}$$

where each coefficient is calculated by:

New coefficient

$coeff_n = ORIGCOEFF \times KERNEL$

The pixel being recalculated and those around it form a 3 x 3 matrix referred to as the "input vector." The input vector is presented to the filter with all of the pixels in parallel forming a 9-byte-wide data path. From each input vector, the filter calculates the new value for the center pixel, referred to as the "output vector," which is 1 byte wide. Calculation of the output pixel is defined by a proprietary kernel algorithm developed at Vigilant Technology, and is the heart of the bilateral filter.

Figure 1 summarizes the input and output vectors.

Starting from a C-Language Model

Vigilant Technology wrote the original implementation of the bilateral noise filter in C, using floating-point math. As a first-pass optimization, we converted all mathematical operations to unsigned fixed point. We accomplished this conversion in part by using C-language macros included with the Impulse C tools.

As a second level of optimization, we reorganized the C code to minimize the number of loops and simplify data handling, using buffered streams for all filter I/O. We defined this I/O using streaming-related interface functions provided within Impulse C.

In certain cases, we replaced array references in the C code with individual variables in order to avoid data-timing dependencies and produce parallel calculations. C-code dependencies included summations that took a linear approach of summing as coefficients were calculated, as well as a multi-tiered approach. During this manual optimization, we used macros to keep the C code relatively clean and portable.

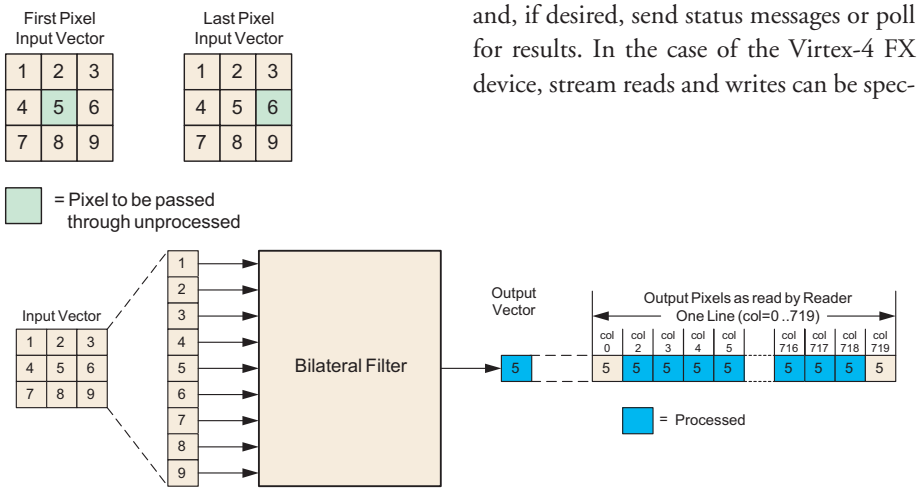


Figure 1 – Bilateral filter processing flow

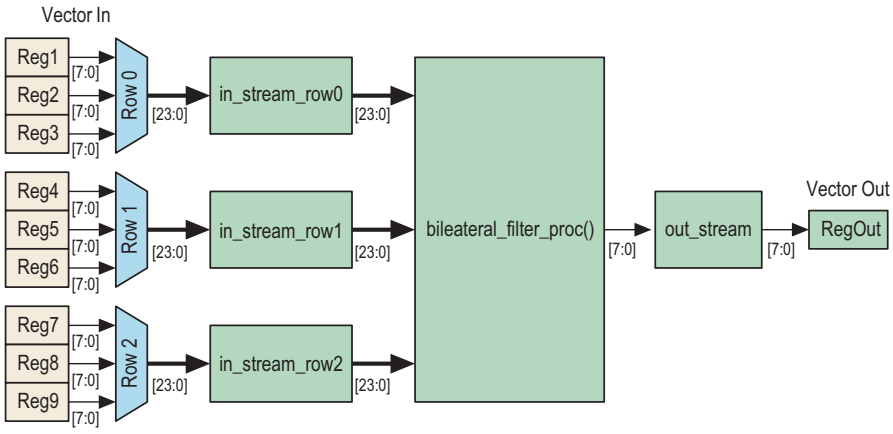


Figure 2 – Block diagram showing streams and processes

As the code was optimized and modified for hardware generation, we used a standard C debugger (part of the Borland C++ Builder tools) to step through both the original and modified C code, allowing for validation of the math and the overall functionality of the application. Throughout development, we performed real-world validation using actual video clips representing various CCTV scenarios.

Figure 2 illustrates how we implemented and tested the bilateral noise reduction filter using the Impulse C streaming interfaces and a Xilinx ML410 development board.

On the software side of the application (in this case on the PowerPC™ 405 processor used for hardware-level testing), Impulse C functions open and close data streams, read or write data on the streams, and, if desired, send status messages or poll for results. In the case of the Virtex-4 FX device, stream reads and writes can be specified as operations that take advantage of the auxiliary peripheral unit (APU) interface, which provides a high-performance software-to-hardware serial interface capable of transferring as many as 128 bits of data in a single transaction.

The embedded PowerPC allowed us to exercise the filter at-speed, using the processor as a test generator.

Generating and Testing the FPGA Hardware

When generating hardware, the Impulse C compiler extracts low-level parallelism from otherwise sequential C-language statements. Structures such as loops are evaluated at a high level and either unrolled or pipelined to increase throughput. These compiler optimizations save application developers a great deal of time.

The Impulse C compiler generates hardware in either VHDL or Verilog. This hardware is synthesized using Xilinx ISE™ software. On the processor side, the compiler generates run-time libraries ready for use on the embedded PowerPC or MicroBlaze™ processor.

Iterative, hardware-level unit testing was an important part of this project. The embedded PowerPC allowed us to exercise the filter at speed, using the processor as a test generator. Using this approach allowed us to validate the correctness of the data, as well as to directly measure the filter performance and throughput.

In fact, an entirely C-language testing method using the embedded PowerPC proved to be far more practical than HDL simulation given the much faster, hardware-speed operation of tests involving video data. Xilinx Platform Studio (XPS) tools proved invaluable during this process. XPS includes a board support package for the Xilinx ML410 board, allowing the embedded PowerPC to be set up as a test generator and communicate with the hardware filter through the Virtex-4 FX APU interface. The Impulse tools include XPS hardware and software export features that greatly simplified the creation of a custom, APU-connected peripheral from the generated bilateral filter hardware.

ISE tools provided the primary measurements of size and clock speed performance, while the ML410 board provided the final determination of whether the generated filter worked properly in hardware. HDL simulation was used at various points in the development process to validate the functionality of the generated HDL code, but more importantly to observe cycle-level operation and find opportunities for C code optimization.

HDL simulation was of limited value for heavily pipelined sections of the generated hardware, but proved quite useful when analyzing sequential blocks and stages of generated HDL code. This HDL-level analysis could be related directly back to the C source code through the use of the Impulse interactive optimizer, which provides visualizations of C-code parallelism and helps to match the generated hardware code to corresponding sections of the original C-language source.

Iterative Optimization

After we developed an initial, non-optimal hardware prototype, we began a process of iterative optimization to meet system timing and size constraints. Based on previous experience with hand-optimized HDL implementations of similar filters, available resources, and real-time video requirements, Vigilant specified a budget of 1,000 FPGA slices, along with a target video data rate of 60 frames per second.

An initial restructuring of the algorithm placed the critical loops in two parallel individual processes. This method of optimization at first appeared promising, with a relatively high clock rate of 143 MHz, but was limited by a noncontiguous pipeline. This characteristic caused the system performance to be limited by the latency of the slowest pipelined process. Even with the two processes running in parallel in a system-level pipeline, the best rate was substantially slower than the required pixel rate.

After additional analysis, we made various modifications to the C code, including the use of a single pipelined process for the main filtering loop. The result of this series of optimizations was a filter that met the slice-count budget and achieved a single-frame rate of just over 15 ms (66 frames per second) using an FPGA clock rate of just 27 MHz, which is the input clock rate of a D1 video stream received in BT.601 format. These results are summarized in Table 1.

Filter Version	Occupied Slices	Frame Rate
Parallel Multi-Process	3,592	12.11 Hz
Pipelined Multi-Process	1,570	10.10 Hz
Pipelined Single Process	930	65.31 Hz

Table 1 – Iterative C-language optimizations resulted in incremental improvements

The widely differing performance results seen in this example highlight the importance of using appropriate C programming methods for FPGAs. Although C-to-FPGA tools can dramatically reduce the amount of development time required to convert complex algorithms to hardware, these tools must be applied carefully. You may need to use iterative optimization methods to achieve results comparable to hand-crafted HDL. Fortunately, these techniques are well within the experience and skills of embedded software developers.

Conclusion

Software-to-hardware design tools can play an important role in the fast prototyping and development of complex FPGA algorithms. However, some level of hand-optimization will be required to obtain performance comparable to hand-crafted HDL. Iterative design methods combined with interactive design and optimization tools are an efficient way to manage the complexity of large embedded and high-performance computing applications.

For additional information about C-to-FPGA programming and optimization, visit *www.ImpulseC.com/xilinx*.