Xcell SOFTWARE journal

SOFTWARE SOLUTIONS FOR
A PROGRAMMABLE WORLD

ISSUE 2
FOURTH QUARTER 2015

Use C/C++ to Offload Image Processing to Programmable Logic

Leverage SDSoC to Accelerate AES Encryption on Zynq

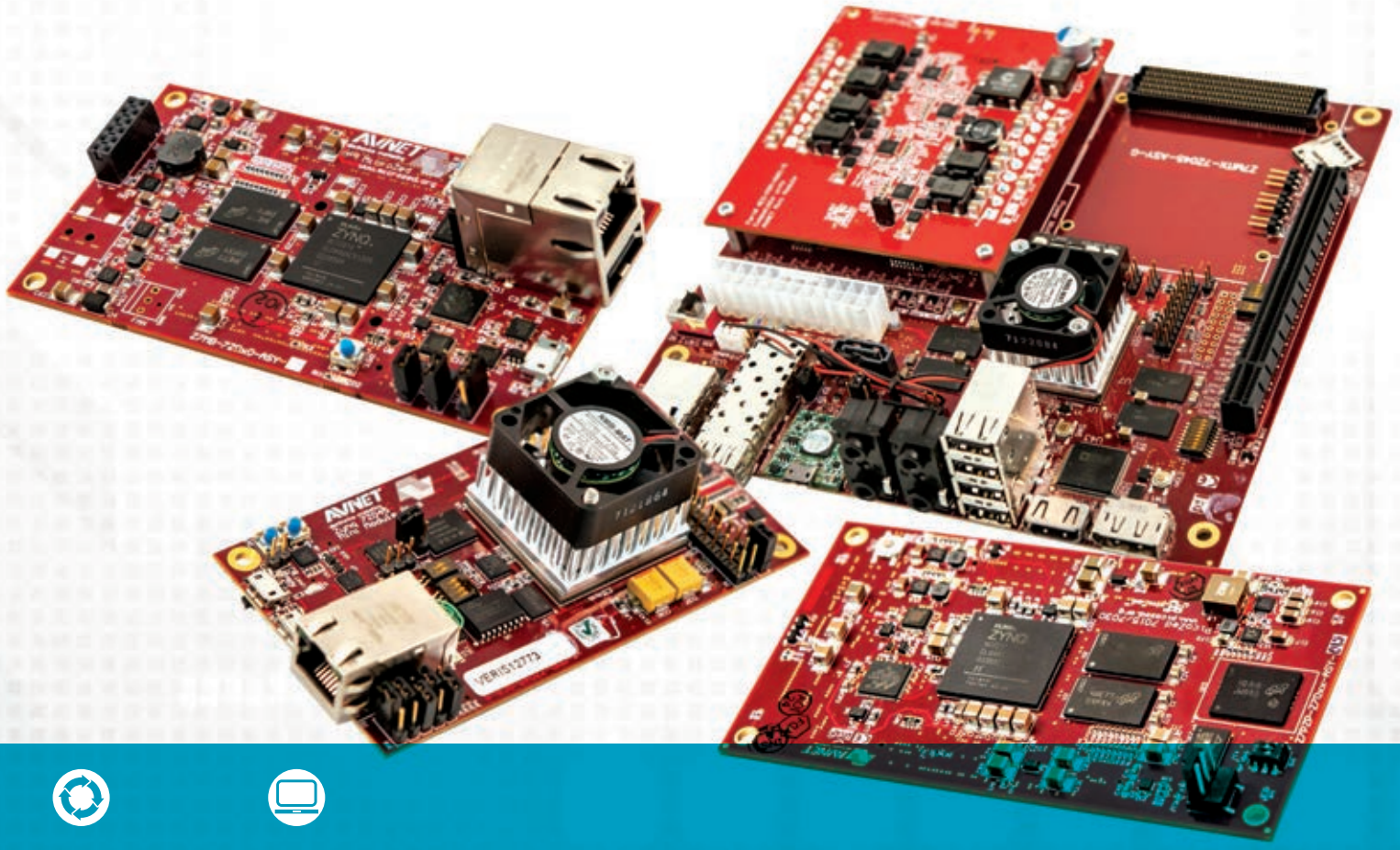
SDNet Helps NTT Hatch Lagopus FPGA Reprogrammable Platform

NI, Vivado Tools Take Machine Vision from Concept to Deployment

PLDA's QuickPlay High-level Workflow Builds Efficient FPGA Apps

XILINX
ALL PROGRAMMABLE™

www.xilinx.com/xcell



Lifecycle

Technology

Design it or Buy it?

Shorten your development cycle with Avnet's SoC Modules

Quick time-to-market demands are forcing you to rethink how you design, build and deploy your products. Sometimes it's faster, less costly and lower risk to incorporate an off-the-shelf solution instead of designing from the beginning. Avnet's system-on module and motherboard solutions for the Xilinx Zynq®-7000 All Programmable SoC can reduce development times by more than four months, allowing you to focus your efforts on adding differentiating features and unique capabilities.

Find out which Zynq SOM is right for you http://zedboard.org/content/design-it-or-buy-it

DESIGNED BY AVNET

MMP

picoZed™

microZed™ FLEXIBILITY REDEFINED MICROZED.ORG

facebook.com/avnet twitter.com/avnet youtube.com/avnet

Accelerating Your Success™

Xcell SOFTWARE Journal

PUBLISHER Mike Santarini
mike.santarini@xilinx.com
1-408-626-5981

EDITOR Diana Scheben

ART DIRECTOR Scott Blair

DESIGN/PRODUCTION Teie, Gelwicks & Assoc.
1-408-842-2627

ADVERTISING SALES Judy Gelwicks
1-408-842-2627
xcelladsales@aol.com

INTERNATIONAL Melissa Zhang,
Asia Pacific
melissa.zhang@xilinx.com

Christelle Moraga,
Europe/Middle East/Africa
christelle.moraga@xilinx.com

Tomoko Suto,
Japan
tomoko@xilinx.com

REPRINT ORDERS 1-408-842-2627

EDITORIAL ADVISERS Tomas Evensen
Lawrence Getman
Mark Jensen

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124-3400
Phone: 408-559-7778
FAX: 408-879-4780
www.xilinx.com/xcell/

Letter from the Publisher

Xciting Times Ahead with the Zynq MPSoC

For those of you doing embedded software development, it's imperative to know what system hardware resources are available to you to create optimized embedded systems. For those of you specializing in developing application software, knowing the nitty gritty details about the system resources isn't so important, but knowing you have silicon that can give you options for improving code performance is certainly a plus.

To this end, Xilinx® last quarter achieved a significant silicon milestone. In late September, Xilinx announced it had shipped to customers the first samples of its Zynq® UltraScale+™ MPSoC (see video), the follow-up to its award-winning Zynq-7000 All Programmable SoC. Whereas the Zynq SoC featured an ARM® dual Cortex™ A-9 processing system connected to programmable logic and on-board peripheral controllers on a single system-on-chip, the Zynq UltraScale+ MPSoC ups the processing power available on an SoC to a total of seven processors (64-bit, quad-core ARM Cortex-A53 and dual-core ARM Cortex-R5 real-time processors, and an ARM Mali™-400 MP GPU), an H.265/264 video codec, an advanced dynamic power management unit to optimize system power efficiency, a configuration security unit, DDR4/LPDDR4 memory interface support, and loads of on-chip programmable logic.

Since the release of the Zynq SoC in 2011, the innovations that Xilinx customers have been able to develop with the device have been truly remarkable. Without delving too deeply into the silicon details, what makes the Zynq SoC device unique are the more than 3,000 connections between the processor and the on-chip programmable logic. Those connections enable the processor and functions implemented in FPGA logic to communicate far faster than would be achievable with any two-chip or even system-in-package configuration. Customers thus have been able to create systems that simply weren't possible before. And since the Zynq SoC's launch, we have seen Zynq SoC-based innovations in just about every market Xilinx serves, from wireless communications to aerospace and defense.

Many of those innovations were created by FPGA engineering teams using the Xilinx Vivado® Design Suite of hardware design tools. Earlier this year, Xilinx took a bold leap forward by introducing the C, C++ and OpenCL™-based SDx™ development environments: SDSoC™ for Zynq SoC design, SDAccel™ for FPGA-accelerated processing and SDNet™ for software-defined networking system development. While relatively new, the SDSoC development environment is already opening up new possibilities to new users—embedded software developers—as well as traditional FPGA experts. Further opportunities for innovation arise from the ability to create a system-level representation of a system in C or C++ and then use the SDSoC environment to identify slower-running code segments and offload them to the FPGA logic for acceleration.

Now, with the combination of the SDSoC environment and the silicon foundation of the Zynq UltraScale+ MPSoC, I'm betting that we will see even more truly remarkable system innovations created by an expanding number of Xilinx users.

In this second issue of *Xcell Software Journal*, you will read how the SDSoC and SDNet environments are enabling new levels of innovation. I hope you enjoy reading the articles and are inspired to begin using the new development environments—and, of course, sharing your experiences with your peers by contributing technical articles to *Xcell Software Journal*.

— Mike Santarini
Publisher

CONTENTS

FOURTH QUARTER
2015
ISSUE 2

16

VIEWPOINT

Letter from the Publisher
Xciting Times Ahead with the Zynq MPSoC...3

COVER STORY

Use C/C++ to Offload Image Processing to Programmable Logic

6

XCELLENCE WITH SDSOC FOR EMBEDDED DEVELOPMENT

Accelerate AES Encryption with SDSoC…16

XCELLENCE WITH SDNET FOR SDN DEVELOPMENT

Innovating a Reprogrammable Network with SDNet…24

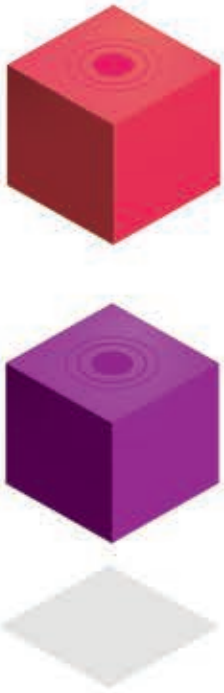
XCELLENT ALLIANCE FEATURES

Delivering FPGA Vision to the Masses…30

A Novel Approach to Software-Defined FPGA Computing…38

XTRA READING

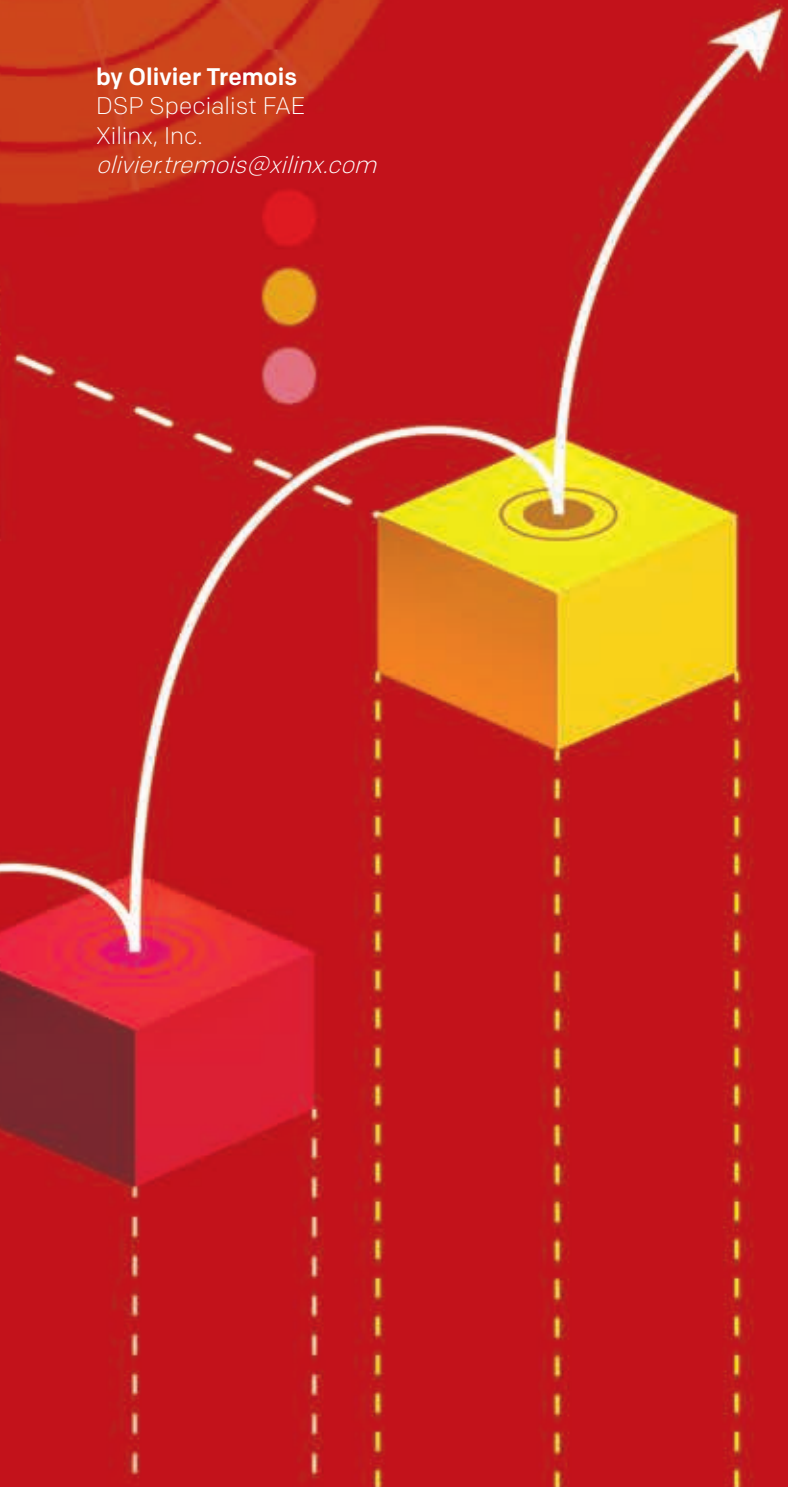
IDE Updates and Extra Resources for Developers . . . 46



Use C/C++ to Offload Image Processing to Programmable Logic

SDSoC lets programmers build complete hardware-software systems without sacrificing performance.

by Olivier Tremois
DSP Specialist FAE
Xilinx, Inc.
olivier.tremois@xilinx.com



The "standard" image processing systems found today in medical, industrial and a growing number of other applications are becoming ever more advanced. In many cases, the imaging processing complexity has already exceeded the processing capabilities of PCs with GPU acceleration. Even as design teams raise their standards for image processing quality and add product features, they must meet customer demand for more-compact, mobile, battery-powered end products.

Many existing platforms are struggling to meet such complex requirements. Luckily, design teams can leverage Xilinx® Zynq®-7000 All Programmable SoCs and the new Xilinx SDSoC™ development environment to create compact, low-power, feature-rich products with advanced imaging systems using C/C++. Let's examine how to do this by using the SDSoC environment to accelerate an image pipeline processing system. I completed this project in less than a week and was able to accelerate the system example by orders of magnitude.

Our batch image processing system will read images stored in an SD card and process them using different parameters for the noise level and for the shape used as a structuring element.

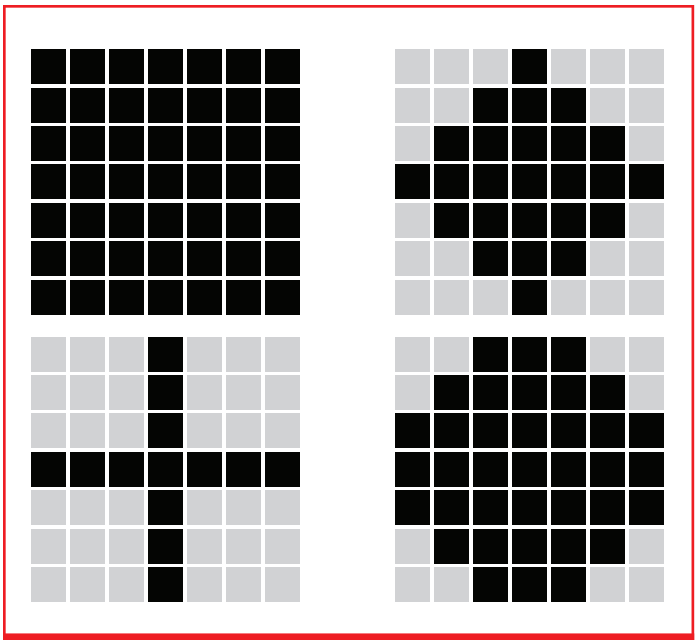


Figure 1 — Structuring elements within a 7 x 7 bounding box

BATCH IMAGE PROCESSING

Our example system acquires images using a specific camera and then processes the images in batch mode. The image size can be up to 3,000 x 2,000 pixels (6 megapixels). Although the processed image is not live video, the intent is to send the images through the image pipeline as quickly as possible. The pipeline here is pretty simple: transform an RGB image into grayscale; add salt-and-pepper noise; and filter the noisy image with three filters (dilate, median and erode).

Dilation, median and erosion filters belong to the family of rank filters, which are primarily but not exclusively applied to remove impulse noise for image enhancement. These are nonlinear filters that involve absolutely no arithmetic operations and restrict their functions to data sorting and picking. Although the algorithms are not highly complex, they consume considerable processing time when they are applied to large images because of the sequential nature of the processor, which will process 1 pixel in a given time period.

A rank filter is a nonlinear filter that computes the output image pixel by pixel. It does so by taking the neighboring pixels of the input image pixel within a specified shape called a structuring element, sorting them and picking the one that is at the pth rank. The erosion filter selects the minimum value (p = 1). Dilation selects the maximum value (p = N, where N is the number of pixels of the structuring element). Median filtering selects the median value (p = [N/2]). Classically, the structuring element is a square, a diamond or a cross (Figure 1).

Our batch image processing system will read images stored in an SD card and process them using different parameters for the noise level and for the shape used as a structuring element. The dual ARM® Cortex™-A9 cores of a Zynq-7000 SoC running at 667 MHz will perform the computations.

SOFTWARE IMPLEMENTATION

As a starting point, we write the complete application in C++ so that we can estimate the performance of the computations on the Cortex-A9. The application contains a number of functions to read and write BMP images on the SD card, compute luminance, add noise and perform the

	#Size	#Shape	SW Latency
Test 1	1,920 x 1,080	49 (square)	29 s
Test 2	1,920 x 1,080	25 (diamond)	8.5 s
Test 3	3,000 x 2,000	25 (diamond)	8.5 s

Figure 2 — Runtime for Zynq processing system only

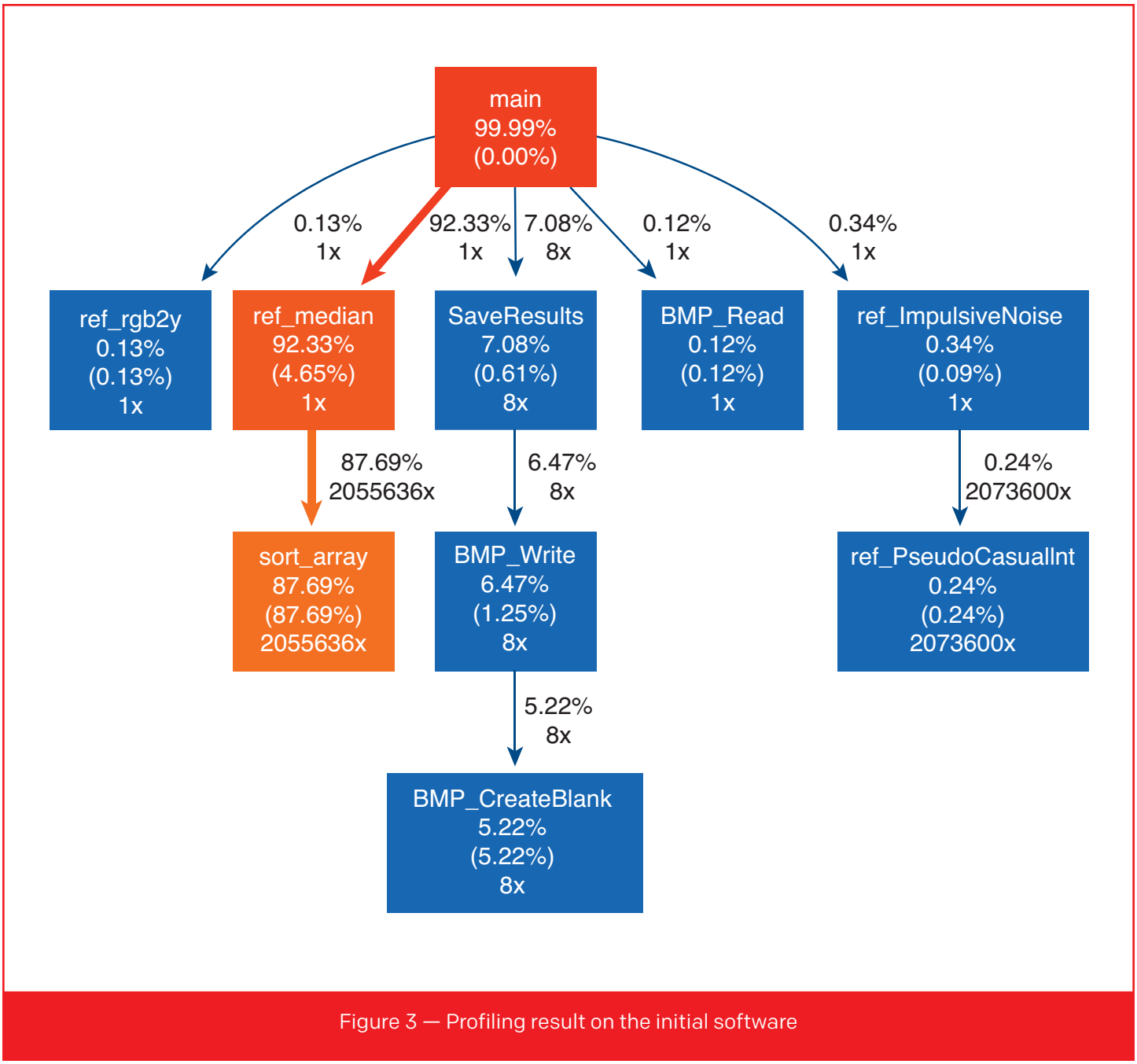


Figure 3 — Profiling result on the initial software

various filter functions. Working within the SDSoC development environment's SDDebug configuration will enable rapid implementation on the Xilinx ZC702 evaluation platform under the Linux operating system.

To generate a truly operational executable file, we select option -O3 to turn on all compiler optimization. The shape of the structuring element is a parameter of the application such that we can apply any kind of structuring element that fits within a 7 x 7-pixel bounding box. The parameters that have an impact on the pipeline latency (Figure 2) are the size of the image (#Size) and the number of active pixels in the structuring element (#Shape). Minimizing those latencies will improve system performance. FPGAs perform incredibly well on signal processing algorithms involving numerous additions and multiplications. Our system example will show that programmable logic is good not only at brute-force computations, but also at more standard data processing.

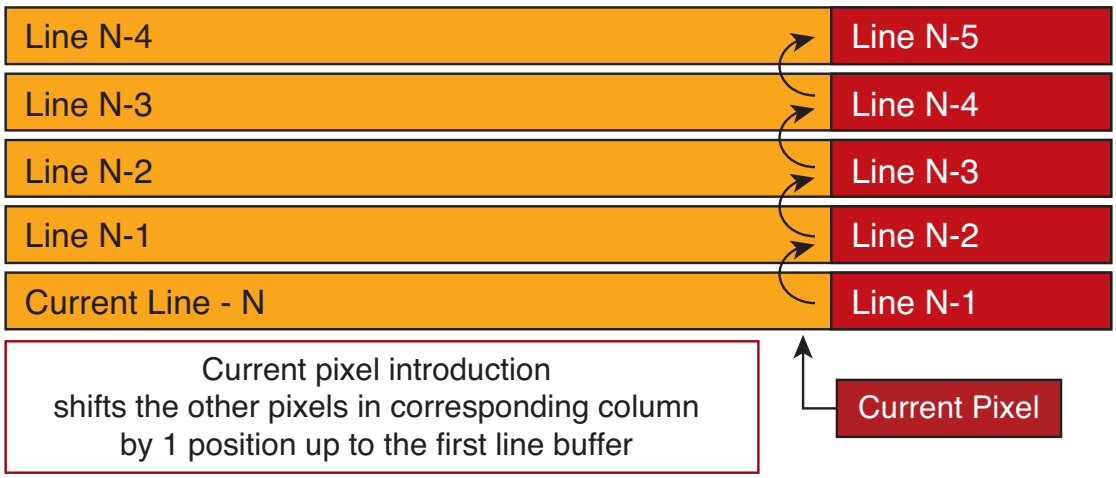


Figure 4 — Data motion in line buffers when receiving a new pixel

A basic profiling (Figure 3) shows that computing luminance from RGB values (0.13 percent) and adding noise to the pixel (0.34 percent) run pretty fast in software. The main contributor to total time is the median filter (92.33 percent). Other functions contributing to total time are file reads and saves.

MOVING A FUNCTION TO HARDWARE

The first goal for this acceleration is to be able to process one new sample every clock cycle. Some code rewriting and a rethinking of the interface can yield greater acceleration. Even if the clock rate of the on-chip programmable logic (PL) is much lower than that of the processing system (PS), being able to process one input pixel per clock should provide great acceleration.

The median filter is the only function that will shift to the hardware. Although the SDSoC environment makes it easy to shift functions into the PL with a simple right click in the environment's Project Explorer, it will not add any directive (except at the interface) or change a single line of code for performance purposes. Those modifications are the embedded programmer's responsibility, which explains why the initial acceleration generally will not be that dramatic.

The function specified above contains two nested loops to go through the entire image. It also contains subloops that go through the structuring element and sort all the elements. In this example, we use a standard bubble sort algorithm. Other reduced-complexity algorithms exist for microprocessor implementation, but the regularity of this one is more adapted to hardware implementation:

```
for ( i=0; i<HeightOfImage; i++)
for ( j=0; j<WidthOfImage; j++)
  {
  Some Code
  for ( s=0; s<NumberOfStages; s++)
  for ( k=0; k<HeightOfStructElem; k++)
  for ( l=0; l<WidthOfStructElem; l++)
  {
  Swap pixels if not correctly ordered
  }
  }
```

Because we want to be able to process 1 pixel of the output image per clock cycle, we must add a directive to start a pixel vector sort with every clock tick. We pipeline the second loop that goes over the columns of the image with an initiation interval (II) of 1. (The II is the number of clock cycles required before a new iteration of the loop can launch.) Using this single directive, the SDSoC environment will automatically unroll the remaining inner loops,

The first goal for this acceleration is to be able to process one new sample every clock cycle. Some code rewriting and a rethinking of the interface can yield greater acceleration.

allowing the hardware to process all the iterations in parallel.

Image processing algorithms implemented in single-core processors are fairly easy to code because various processor features allow smooth data movement between external memory and the processor itself. Memory caches L1 and L2 will temporarily store data that may be reused later, improving data access latency.

Such a mechanism does not exist by default in FPGAs. Although this prevents us from using the same C/C++ source code to create a hardware accelerator, it is a chance for us to design a memory cache that will have the right performance and size for our application. This is a good example of an instance in which we have to change the C/C++ source code not to keep the same functionality, but to improve the performance to a level that suits our requirements. Xilinx's Vivado® High-Level Synthesis (HLS), which is the SDSoC engine that generates register transfer level (RTL) IP from C/C++ code, will generate a hardware architecture that is adapted to our code, taking into account our directives. That's why line buffers and analysis windows are not automatically generated when analyzing the image processing code; Vivado HLS adheres to the code as written, which prevents the tool from hiding optimizations that could be done without the developer's consent.

Designers who are familiar with image processing in hardware know all about line buffers and analysis windows. In order to avoid multiple reads of the same pixel from the external memory, pixels are temporarily stored in internal memory (block RAM) and then overwritten when they are no longer useful for the remaining execution. The block RAMs have two ports that can be used as memory reads, memory writes or both. When the accelerator accepts the pixel corresponding to line L and column C, all the pixels corresponding to column C and line (L-1 … L-6) are read from the line buffer and rewritten to another location, as Figure 4 illustrates. In order to

```
pix_t line_buffer[KMED][MAX_WIDTH];
#pragma HLS ARRAY_PARTITION variable=line_buffer complete
dim=1

pix_t window[KMED*KMED];
#pragma HLS ARRAY_PARTITION variable=window complete dim=0
```

Figure 5 — Array declaration for the line buffer, analysis window and structuring element

```
// Line Buffer fill
if(col < width)
BufferFill:for(int ii = 0; ii < KMED-1; ii++)
        pixel[ii]=line_buffer[ii][col]=line_buffer[ii+1][col];

//There is an offset to accommodate the active pixel region
if((col < width) && (row < height))
{
        pix = in_pix[index_in++];
        pixel[KMED-1] = line_buffer[KMED-1][col] = pix;
}
```

Figure 6 — Data motion between the line buffers

achieve this 1-pixel/clock objective, all data movement must be performed with a throughput of one clock cycle.

Moreover, all the pixels belonging to the pixel neighborhood and the structuring element must likewise be accessed in one clock cycle. That is why we also define an analysis window that contains the specific pixels in question (and which varies from pixel to pixel). In the SDSoC environment and VHLS, the code is not timed in any way; the tool will parallelize anything that can be parallelized with respect to the resource used and our directives. In our sample batch image processing system code, we add the line buffer and analysis window to the code just by declaring two arrays with the right

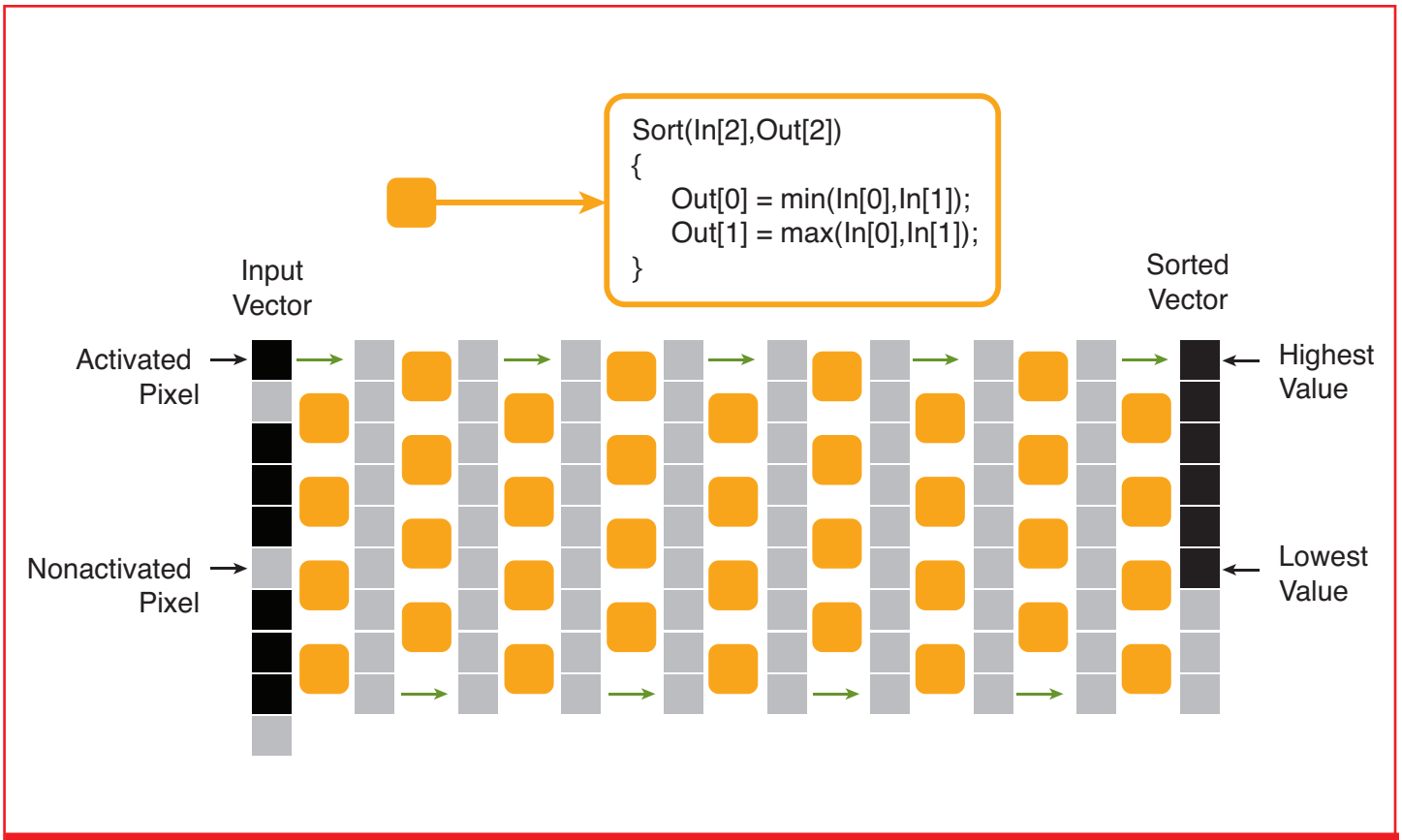


Figure 7 — Sorting network for a 10-element vector

partitioning directives (Figure 5). We then describe data motion as read/write accesses to those arrays (Figure 6).

Because it relies on data access in an array, the pixel-value sorting procedure can be complex to implement in the hardware architecture. The C code for software implementation takes the vector of the pixel that has been validated through the structuring element and sorts it using a standard bubble sort. More-efficient algorithms exist but provide a significant benefit only on larger vectors. The complexity of this algorithm is proportional to the square of the number of pixels of the structuring element—up to $(7 \times 7)^2$ for our sample design.

In hardware, the architecture must be dimensioned for the worst case. If we want to achieve our 1-pixel/clock goal, we need to implement a very regular structure. To this aim, we specify that the input vector will always have the maximum size (7 x 7) and that all nonvalidated pixels will have the value 0 so that they will be at the bottom of the sorted vector. We also dimension the number of stages for the worst case, even if the number of stages could be lower for a structuring element with fewer active pixels. Parallelization of the different stages can occur only if the same vector is not reused at each stage. The result is an array into which the initial vector enters at column index 0 and exits at column index 7 x 7 = 49 (Figures 7 and 8).

SDSOC SYSTEM COMPILER

SDSoC is not a simple full-system compiler. It performs an extensive code analysis in order to decide what kind of data mover would best suit the functions that are required to be in hardware, and to which port to connect the data mover. For each parameter of the function, we must determine whether it is best to use an ARM® AMBA® AXI4-Lite, AXI4-Full memory-mapped or AXI4-Stream data mover.

SDSoC is not a simple full-system compiler. It performs an extensive code analysis in order to decide what kind of data mover would best suit the functions that are required to be in hardware, and to which port to connect the data mover.

```
void sorting_network(pix_t window[KMED*KMED],mask_t shape[KMED*KMED],
                kmed2_t NShape,kmed2_t CompNShape,
                pix_t *pixmin,pix_t *pixmed,pix_t *pixmax)
{
static const int N = KMED*KMED;
pix_t tmin,tmax,t0,t1;

static pix_t z[N][N+1]; // Array that contains the sorting network
#pragma HLS ARRAY_PARTITION variable=z complete dim=0

unsigned int i, k, stage;

// Initialization of the first row of the network
// pixels that do not belong to the mask are set to 0
L1:for (i=0; i<N; i++)
        if(shape[i])    z[i][0] = window[i];
        else    z[i][0] = 0;

//sorting_network: This description is correct for KMED odd
L2:for (stage = 1; stage <= N; stage++)
{
        k = (stage&1)^1; // stage odd -> k=0     stage even -> k=1
        L3:for (i = k; i<N-1; i=i+2)
        {
                t0  = z[(i)][stage-1];
                t1  = z[(i+1)][stage-1];
                tmin = MIN(t0,t1);
                tmax = MAX(t0,t1);
                z[(i  )][stage] = tmin;
                z[(i+1)][stage] = tmax;
        }
        // Copy the value that has not been sorted to the next stage
        if(k==0) z[N-1][stage] = z[N-1][stage-1];
        else z[0][stage] = z[0][stage-1];
}
*pixmin = z[CompNShape][N];
*pixmed = z[CompNShape+NShape/2][N];
*pixmax = z[N-1][N];
return;
}
```

Figure 8 — Sorting network described in C

```
#pragma SDS data sys_port(in_pix:AFI, out_pix_min:AFI, out_pix_med:AFI, out_pix_max:AFI, median_mask:ACP)
#pragma SDS data data_mover(in_pix:AXIDMA_SIMPLE, out_pix_min:AXIDMA_SIMPLE, out_pix_med:AXIDMA_SIMPLE, out_pix_max:AXIDMA_SIMPLE)
#pragma SDS data mem_attribute(in_pix:CACHEABLE|PHYSICAL_CONTIGUOUS)
#pragma SDS data mem_attribute(out_pix_min:CACHEABLE|PHYSICAL_CONTIGUOUS)
#pragma SDS data mem_attribute(out_pix_med:CACHEABLE|PHYSICAL_CONTIGUOUS)
#pragma SDS data mem_attribute(out_pix_max:CACHEABLE|PHYSICAL_CONTIGUOUS)
void hw_median(pix_t in_pix[MAX_HEIGHT*MAX_WIDTH],
       pix_t out_pix_min[MAX_HEIGHT*MAX_WIDTH],
       pix_t out_pix_med[MAX_HEIGHT*MAX_WIDTH],
       pix_t out_pix_max[MAX_HEIGHT*MAX_WIDTH],
       pix_t median_mask[KMED][KMED],
       short height, short width);
```

Figure 9 — Directives in the SDSoC environment to override default behavior

We also have to determine which connector to use: the AXI4 High Performance (HP), General Purpose (GP) or Accelerator Coherency Port (ACP), or even ports from other accelerators, either built from within the SDSoC environment or contained in the board support package (BSP).

The SDSoC environment will then create a design, adding all necessary IP to make a fully functional system—a direct memory access (DMA) for AXI4 Stream data movers, for example—and will modify the C source code (instead of the initial C++ code) in order to call the hardware. In our case, the interface is pretty simple: Two input arrays and three output arrays will be accessed through AXI4-Stream and DMAs, and a few scalars will be set through AXI4-Lite. We don't have to think about setting the DMAs or look at the address at which the scalar registers are accessible; the SDSoC environment manages everything automatically, under the hood.

Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	34833713710
HW accelerated (Estimated cycles)	6936128677
Estimated speedup	5,02

Details

Performance estimates for 'hw_median' function

SW-only (Measured cycles)	27937991386
HW accelerated (Estimated cycles)	40406353
Estimated speedup	691,43

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	0	220	0
BRAM	12	140	8,57
LUT	23105	53200	43,43
FF	11192	106400	10,52

Figure 10 — Performance estimates obtained during the SDEstimate phase

When I built the sample system, I first verified that the source code was Vivado HLS compliant and then added the VHLS directives. Using specific SDSoC directives, I specified that the data would be stored contiguously in the physical space (with memory allocated using the function sds_alloc) and that I wanted a DMA to access it (Figure 9).

I then switched the build configuration to SDEstimate in order to have a first rough estimate of the acceleration that was achievable (Figure 10). I did not have to wait a long time for this step, because at this point no hardware had been built.

The SDSoC environment computes the speedup estimate from the processor runtime (computed using the hardware-adapted code, which is slower than the original, processor-adapted code, and with compiler optimization set to –O0) and the number of clock cycles (computed by VHLS as the latency of the hardware accelerator). This latency is the maximum latency of the hardware accelerator, so this estimation should be taken for what it is—a rough

At this stage, we are able to have an exact value of the acceleration obtained using the hardware accelerators, taking into account all the data transfers to and from DDR.

	#Size	#Shape	Pure SW Latency	SW + HW Latency	Acceleration
Test 1	1,920 x 1,080	49 (square)	29.2 s	154.8 ms	189x
Test 2	1,920 x 1,080	25 (diamond)	8.6 s	154.7 ms	56x
Test 3	3,000 x 2,000	25 (diamond)	25 s	447 ms	56x

Figure 11 — Runtime for Zynq Processing System with accelerated function in Programmable Logic

estimate. This acceleration is almost 700x for the hardware accelerator itself. There are many file accesses that take time at the "main" level; that's why the overall acceleration is "only" 5x. In practice, we can choose the top-level function at which the global acceleration is computed so that we can obtain a more meaningful acceleration value.

The final step of the flow is to build the entire system. In this phase, all the accelerators are built and connected to the processor. The C++ source code is then modified in order to start and control these accelerators (instead of calling the original C function). At this stage, we are able to have an exact value of the acceleration obtained using the hardware accelerators, taking into account all the data transfers to and from DDR. This acceleration value also takes into account the time it takes to flush the cache, as our data is in a cacheable part of the memory.

The time taken by the hardware accelerator is proportional to the size of the image and not to the size of the structuring element. That's why the higher the number of active pixels in the structuring element, the higher the acceleration ratio will be. The latency referred to in Figure 11 is that of the full image pipeline, containing the software and hardware elements.

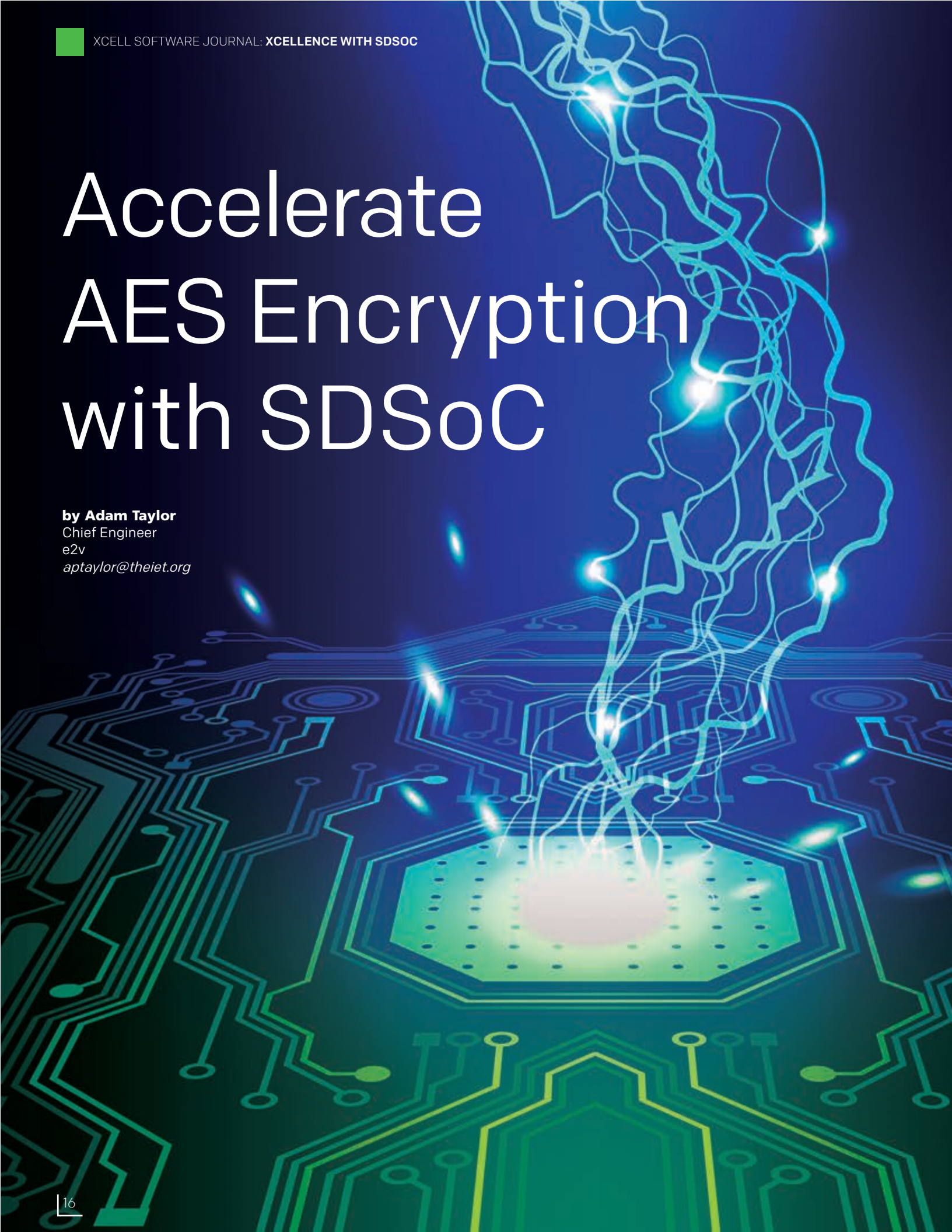
When I undertook this project, building the software application proved to be the longest phase. From there, it took less than 2 hours to modify the code so that I had fully compliant Vivado HLS code with the right directives in place to optimize the throughput. Given the size of the hardware part of this design (half the lookup table of the chip), the last stage—synthesis, place and route, bitstream, SD card—took more than 2 hours to complete.

The SDSoC environment's integrated tools for system-level profiling, automated software acceleration in programmable logic and full-system-optimizing compilation—automatically generating the right connectivity to minimize memory access bottlenecks—allowed me to go through this example project in less than a week.

That short time frame wouldn't have been possible using a standard RTL flow for the creation of the accelerator and my own programming abilities to take advantage of the different drivers to modify the C code. ■

Accelerate AES Encryption with SDSoC

by Adam Taylor
Chief Engineer
e2v
aptaylor@theiet.org



Describe the AES256 crypto algorithm in C, then speed performance in hardware.

The Advanced Encryption Standard (AES) has become an increasingly popular cryptographic specification in many applications, including those within embedded systems. Since the National Institute of Standards and Technology (NIST) selected the specification as a standard in 2002, developers of processor, microcontroller, FPGA and SoC applications have turned to AES to secure data entering, leaving and residing within their systems. The algorithm is described very efficiently at a higher abstraction level, as is used in traditional software development; but because of the operations involved, it is most efficiently implemented in an FPGA. Indeed, developers can even get some operations "for free" in the routing.

For those reasons, AES is an excellent example of how developers can benefit from the Xilinx® SDSoC™ development environment by describing the algorithm in C and then accelerating the implementation in hardware. In this article we will do just that, first gaining familiarity with the AES algorithm and then implementing AES256 (256-bit key length) on the processing system (PS) side of a Xilinx Zynq®-7000 All Programmable SoC to establish a baseline of software performance before accelerating it in the on-chip programmable logic (PL). To gain a thorough understanding of the benefits to be gained, we will perform the steps in all three operating systems the SDSoC environment supports: Linux, FreeRTOS and BareMetal.

THE ALGORITHM

AES is a symmetric block cipher that can be used with varying key lengths of 128, 192 and 256 bits. The key length determines the number of processing steps required to encrypt or decrypt data. As their name implies, block cipher algorithms work on blocks of data. The AES algorithm operates on a fixed block size of 16 bytes at a time. Thus, if we wish to encrypt fewer than 16 bytes, we must pad out the unused bytes.

Because AES is a symmetric cipher, the same actions and key are used to encrypt and decrypt information. In contrast, asymmetric algorithms such as RSA use different keys for data encryption and decryption.

Each of the four stages in the AES algorithm is applied to what is called the state. The combination of the four AES stages is called a round. The number of rounds required depends on the key length.

Quite simply, the AES state starts out as the 16 bytes we wish to encrypt. Each new step updates the state. Before processing the state, we need to format the input byte string correctly into the initial state as a 4 x 4 matrix (Figure 1).

Now that we have rearranged the initial 16 bytes into the initial state as a 4 x 4 grid, we can explore how each step manipulates its input state.

AddRoundKey: This is the only step that uses the encryption key. As we have already noted, the number of encryption algorithm rounds required depends on the key size (128, 192 or 256 bits). The encryption key must undergo key expansion to ensure that the bytes in the key are not reused during each round before use. Not surprisingly, the expanded key length is different for each key size. The expanded key size will be:

*Expanded Key Size (bytes) = 16 * (Rounds + 1)*

The operation within this step is simple. The input state bytes are exclusive-ORed with 16 bytes of the expanded key. Each round uses a different section of the expanded key; round 0 uses bytes 0 to 15, round 1 uses bytes 16 to 31 and so on. For each round, byte 1 of the state is exclusive-ORed with the least significant byte of the expanded key, byte 2 is exclusive-ORed with least significant byte + 1 and so on.

SubBytes: This step uses byte substitution to swap out state values with another value. The values within the substitution box are predefined and have been designed to have low correlation between input bits and output bits. The substitution box (S-box) is a 16 x 16 matrix. We use the

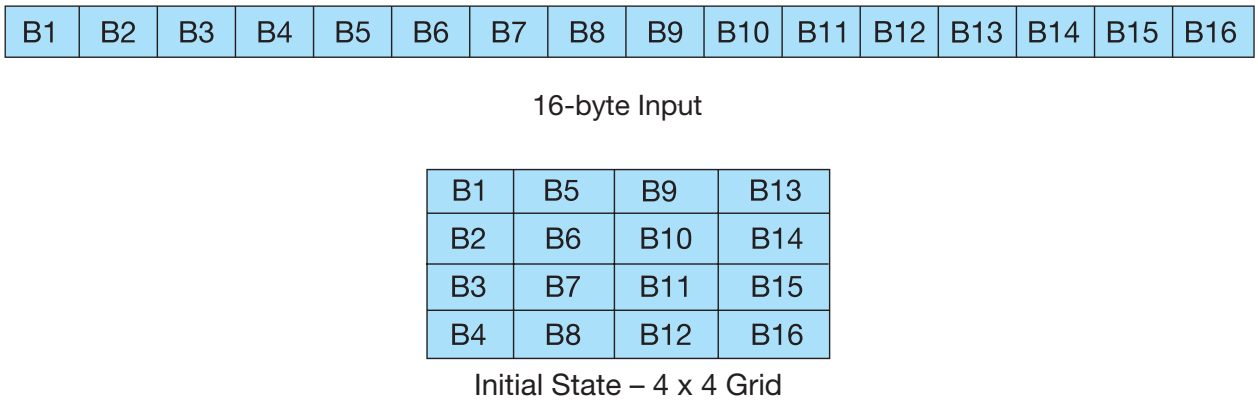


Figure 1 — Initial state translation of the 16 bytes into a 4 x 4 grid

upper and lower nibbles of the byte being substituted to index into the substitution table. For example, using the S-box encryption in Figure 2, if the first initial state byte is 0 x 69, then the substitution value 0 x F9 will replace it. The upper nibble of the state byte selects the row in the substitution box; the lower nibble selects the column. Note in Figure 2 that there are separate substitution boxes for encryption and decryption and that their content differs.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	CB	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

S-box for Encryption

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	23	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

S-box for Decryption

Figure 2 — AES S-box contents

ShiftRows: This step rearranges the input state matrix by performing a circular byte shift for each row. We rotate each row right by a different factor (Figure 3). We leave row 1 unchanged. We rotate row 2 by 1 byte, row 3 by 2 bytes and row 4 by 3 bytes. When we decrypt, we perform the same operations, but we rotate left instead of right.

MixColumns: This is the most complicated step within a round, requiring 16 multiplications and 12 exclusive-OR operations. The operations are performed column by column on the input state matrix, which is multiplied against a fixed matrix to create a new state column (Figure 4). Each entry in the column is multiplied by a row in the matrix. The results of each multiplication are XORed together to form the new state value. The first column and row to be multiplied are highlighted in Figure 4.

Here are the MixColumns equations for the first column:

*B1' = (B1 * 2) XOR (B2 * 3) XOR (B3 * 1) XOR (B4 * 1)*
*B2' = (B1 * 1) XOR (B2 * 2) XOR (B3 * 3) XOR (B4 * 1)*
*B3' = (B1 * 1) XOR (B2 * 1) XOR (B3 * 2) XOR (B4 * 3)*
*B4' = (B1 * 3) XOR (B2 * 1) XOR (B3 * 1) XOR (B4 * 2)*

This process is then repeated against the same multiplication matrix for the next column in the input state until all of the input state columns have been addressed.

Now that we understand the detailed steps needed for the AES encryption and decryption algorithms, we need to know the order in which to apply the steps in a round and whether we must apply all of

AES is described efficiently at a higher abstraction level, as in traditional software development, but is most efficiently implemented in an FPGA. Developers can even get some operations "for free" in the routing.

the steps for each round. Each AES encryption round comprises all four steps, in the following order:

1. SubBytes;
2. ShiftRows;
3. MixColumns (for rounds 1 to N – 1 only);
4. AddRoundKey (using the expanded key).

Of course, we need to be able to reverse the process and turn the unreadable cipher text back into plain text so that the encrypted information will be useful. To do so, we order the steps as follows:

1. Invert ShiftRows;
2. Invert SubBytes;
3. AddRoundKey (using the expanded key);
4. Invert MixColumns (for rounds 1 to N – 1 only).

Before executing the first round of encryption, we need to perform an initial AddRoundKey operation for both encryption and decryption.

B1	B5	B9	B13
B2	B6	B10	B14
B3	B7	B11	B15
B4	B8	B12	B16

ShiftRows Input State

B1	B5	B9	B13
B6	B10	B14	B2
B11	B15	B3	B7
B16	B4	B8	B12

Resultant Output State

Figure 3 — ShiftRows operation

B1	B5	B9	B13
B2	B6	B10	B14
B3	B7	B11	B15
B4	B8	B12	B16

Input State First Column to be Multiplied

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

Constant Multiplication Matrix

E	B	D	9
9	E	B	D
D	9	E	B
B	D	9	E

Decryption Constant Multiplication Matrix

Figure 4 — MixColumns function for encryption and decryption

Let's look at the algorithm we must use for expanding the key so that we provide sufficient key bits for the number of AddRoundKey steps to be performed (Figure 5). Key sizes of 16, 24 or 32 bytes will respectively require 44, 52, or 60 rounds for key expansion. The first bytes of the expanded key are equal to the initial key. This means that for our AES256 example, the first 32 bytes of the expanded key are the key itself. Key expansion generates the 32 additional bits for the expanded key in each iteration.

The following are the key expansion steps.

RotateWord: Similar to ShiftRows, this step reorganizes a 32-bit word such that the most significant byte becomes the least significant byte.

The first bytes of the expanded key are equal to the initial key. This means that for our AES256 example, the first 32 bytes of the expanded key are the key itself.

SubWord: This step uses the same substitution box used to make byte substitutions in the encryption.

rcon: This stage performs the exponentiation of 2 to a user-defined value. As in the MixColumns stage, rcon is performed over the Galois field (28); therefore it is common to use a precalculated lookup table for this step.

EK: This returns 4 bytes from the expanded key.

K: Like EK, this returns 4 bytes from the key.

How will we know that we have correctly implemented the encryption and key expansion algorithms? The NIST specification for AES helpfully contains a number of worked examples that we can use for checking our own implementations.

CREATING THE CODE

To ensure we can accelerate the encryption part of the AES code within the PL side of the Zynq SoC, we must develop the code from day one with this objective in mind (see the coding rules here). The first thing to consider is the architecture of the algorithm; we need to segment it properly. AES lends itself well to this approach because we can write functions for each of the stages and then call them as required. We must also write the function to be accelerated within its own file. Our software architecture will include the following.

main.c: This file contains the key expansion algorithm, the encryption key and the plain-text input, along with the call to the AES encryption function.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp

    i = 0

    while (i < Nk)
       w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
       i = i+1
    end while

    i = Nk

    while (i < Nb * (Nr+1)]
       temp = w[i-l]
       if (i mod Nk = 0)
          temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
       else if (Nk > 6 and i mod Nk = 4)
          temp = SubWord(temp)
       end if
       w[i] = w[i-Nk] xor temp
       i = i + 1
    end while
end
```

Figure 5 — Key expansion algorithm

```
void aes_enc(uint8_t state[4][4],uint8_t cipher[4][4],uint8_t ekey[240])
{

    uint8_t iteration = 0;
    //uint8_t x,y;

    uint8_t sub[4][4];
    uint8_t shift[4][4];
    uint8_t mis[4][4];
    uint8_t round[4][4];

    addroundkey(state,Ø,sub,ekey);

    loop_main : for(iteration = 1; iteration < nr; iteration++)
       {
        subbytes(sub,shift);
        shift_row_enc(shift,mix);
        mixcolumn(mix,round);
        addroundkey(round,iteration,sub,ekey);
       }
       subbytes(sub,shift);
       shift_row_enc(shift,round);
       addroundkey(round,nr,cipher,ekey);

}
```

Figure 6 — The function to be accelerated

aes_enc.c: This file performs the encryption. We will code each of the stages as its own function so that it can be called as required for the AES round. To ensure the design is common to those implemented on processors, we use lookup tables for the mixed step's multiplications.

aes_enc.h: This file houses the definition of the aes_function and the parameters used to determine the size (e.g., mk, nb and nr).

sbox.h: This includes the substitution box used for the substitute bytes, the lookup table for the rcon function that performs key expansion, and the multiplication lookup tables for the MixColumns multiplications.

Within this structure, we can select the AES encryption function (Figure 6) as the one we wish to accelerate simply by right clicking on the function and selecting Toggle HW/SW.

To ensure that we are able to determine the baseline performance and the savings we get from the accelerating the function, we must be able to time the execution of the function. To do this, we'll use sds_clock_counter in sds_lib.h.

After I had written the source code (available on the github), I recorded a time of 36,662 processor cycles when executing the AES algorithm in software running on a single ARM® Cortex™-A9 processor core in the Zynq SoC.

OPTIMIZATION FOR ACCELERATION

Accelerating the AES algorithm is slightly more complicated than the matrix multiplication algorithm we examined in the previous issue. This is because the main loop of the AES algorithm consists of interdependent stages.

I accelerated the AES algorithm by examining the loops to see where I could unroll them, optimizing the memory bandwidth, selecting the correct frequency for the data motion clock frequency and selecting the correct frequency for the hardware functions.

The main loop of the AES encryption function comprises the functions that perform each AES step. Each function in the AES algorithm must be completed and the result computed before the next function can run. This interdependency requires us to focus our efforts

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
aes_enc_0	state	state_PORTA	IN	16*1		S_AXI_ACP:AXIDMA_SIMPLE
	cipher	cipher_PORTA	OUT	16*1		S_AXI_ACP:AXIDMA_SIMPLE
	ekey	ekey_PORTA	IN	240*1		S_AXI_ACP:AXIDMA_SIMPLE

Figure 7 — The data motion network between the PS and PL

on the AES steps created as separate functions. There is plenty of potential for optimization within these steps.

We can pipeline the AddRoundKey, SubBytes and MixColumns steps for increased performance. Within these functions, we execute the HLS Pipeline command by putting pragmas within the first loop. We should unroll the inner loop. Several of these functions read from lookup tables normally built from block RAM. We need to increase the memory bandwidth, so for this example I have specified the pragma parameter "complete," which implements the memory contents as discrete registers as opposed to BRAM.

The ability to transfer the data between the PS and the PL on the Zynq SoC is also of key importance in boosting performance. My first step was to set the data motion clock network at its highest possible clock frequency: 200 MHz. My second approach was to ensure that direct memory access was used for data transfer between the PS and PL. To do this, I had to rewrite the interface slightly and use the sds_alloc function to ensure that the data was contiguous in memory, as DMA transfer requires (Figure 7).

My third and final optimization step was to set the hardware function's clock rate at the highest frequency supported for this application: 166.67 MHz.

RESULTS ON THE SUPPORTED OPERATING SYSTEMS

When I finally put these all together and built the example, the PL-accelerated AES code ran on Linux in 16,544 processor clock cycles, or 45 percent (16,544 / 36,662) of the cycles needed when running the AES code in software alone. That's a massive 55 percent reduction in execution time for a fairly complex and interdependent algorithm.

Of course, we can select the BareMetal or FreeRTOS operating system within the SDSoC environment as well. Creating BareMetal and FreeRTOS projects and reusing the code allows a comparison of performance among the three supported operating systems. For a given project, the OS selection will depend on the mission requirements, performance budgets and response times.

Figure 8 reveals the three operating systems' performance in the Zynq SoC's PS and PL (Figure 8).

It is not surprising that FreeRTOS and BareMetal provide similar reductions, as both are much simpler implementations than the full Linux OS.

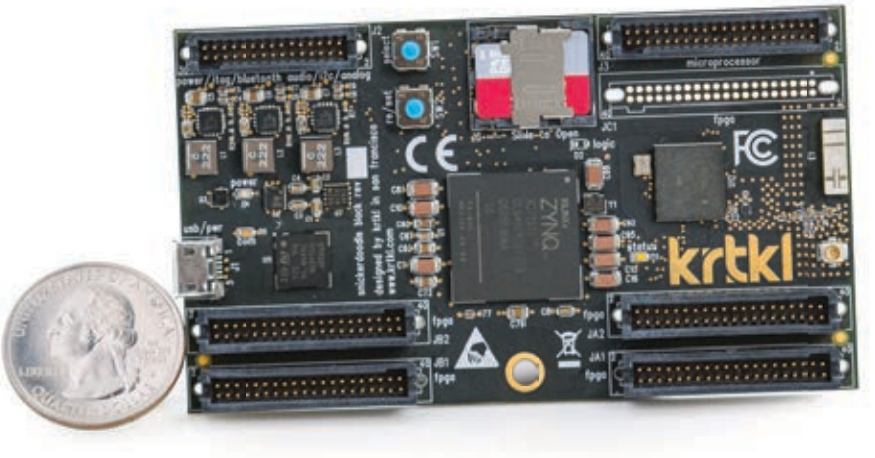
As our results show, using the SDSoC development environment to accelerate AES encryption provides a real performance improvement and is easy to achieve—without in-depth FPGA design experience. ■

Operating System	PS Only	PS with PL Acceleration	Reduction
BareMetal	28574	7102	75%
FreeRTOS	28368	7104	75%
Linux	36662	16544	54.8%

Figure 8 — OS performance in the Zynq PS and PL. FreeRTOS and BareMetal provide similar reductions.

snickerdoodle

create something different



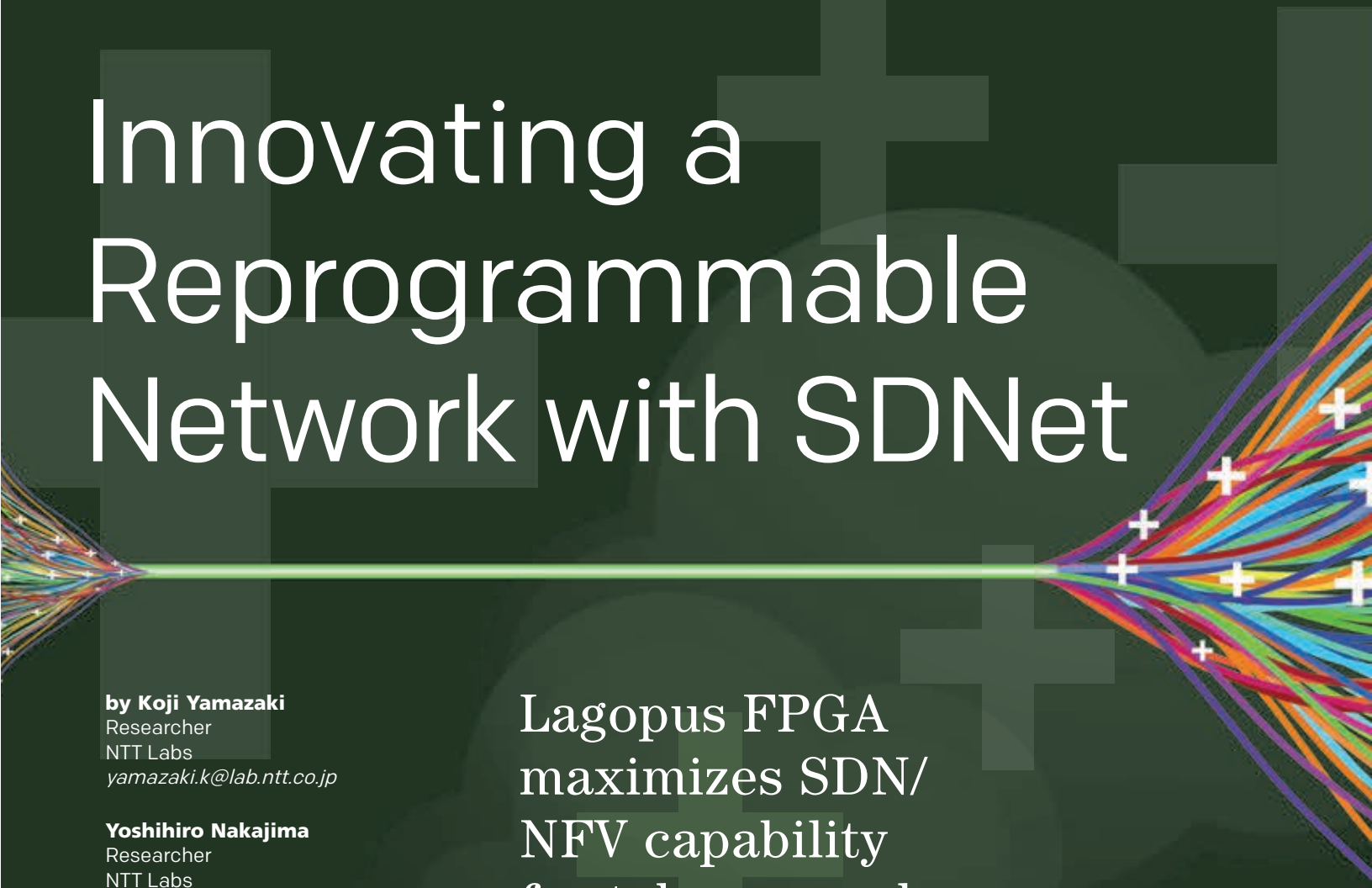
Zynq®. Wi-Fi®. Bluetooth®. $55.
Any questions?

order yours today at

snickerdoodle.io

krtkl®

Innovating a Reprogrammable Network with SDNet



Lagopus FPGA maximizes SDN/NFV capability for telecom and the cloud.

by Koji Yamazaki
Researcher
NTT Labs
yamazaki.k@lab.ntt.co.jp

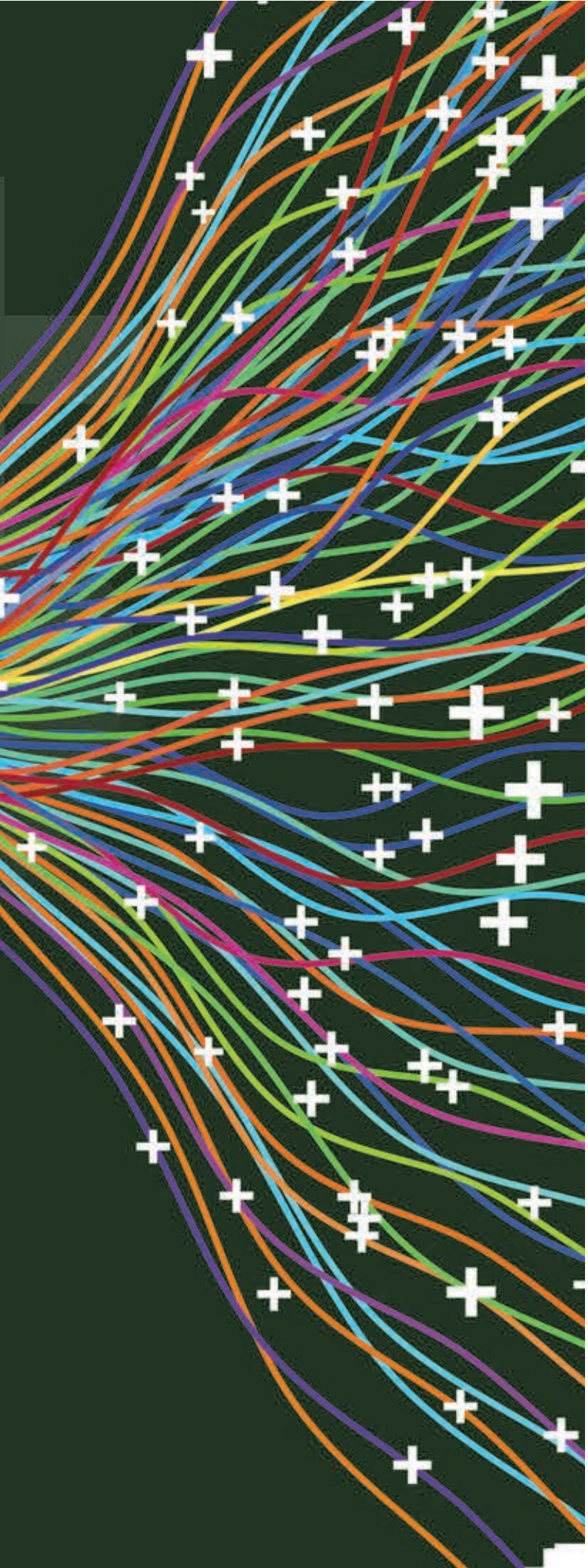
Yoshihiro Nakajima
Researcher
NTT Labs

Takahiro Hatano
Researcher
NTT Labs

Hirokazu Takahashi
Researcher
NTT Labs

Akihiko Miyazaki
Researcher
NTT Labs

Katsuhiro Shimano
Researcher
NTT Labs



Nippon Telegraph and Telephone Corp (NTT) is the holding company for a global telecommunications group that formulates management strategies and promotes research and development. We are researchers in NTT's R&D division and are leading two innovative projects for software-defined networking (SDN) and network function virtualization (NFV). For one project, we have developed a high-performance software SDN/OpenFlow switch called Lagopus [1], which we believe to be the best OpenFlow 1.3-compliant switch to have been released to date as open-source software. For our second project, we have developed a software-packet-processing-aware, 40-Gbit/second (Gbps) FPGA network interface card (NIC) called Lagopus FPGA.

Our early adoption of the Xilinx® SDNet™ Software Defined Specification Environment for Networking was key to our ability to develop these technologies. Here's how we used SDNet to meet our goals for the projects.

LAGOPUS FPGA FOR SDN/NFV EXCELLENCE

Cloud service providers and network service operators are turning to SDN as a key enabling technology for automated provisioning systems. NFV has a critical role in letting telecom operators reduce capex and opex by changing network systems from proprietary-hardware-based equipment to commodity-hardware-based systems that leverage PC servers, merchant silicon-based switches and software appliances. Many cloud service providers and telecom operators will deploy SDN and NFV for their next-generation commercial networks.

NTT Group is a leader in SDN and NFV in both the commercial-services and research spheres. NTT has

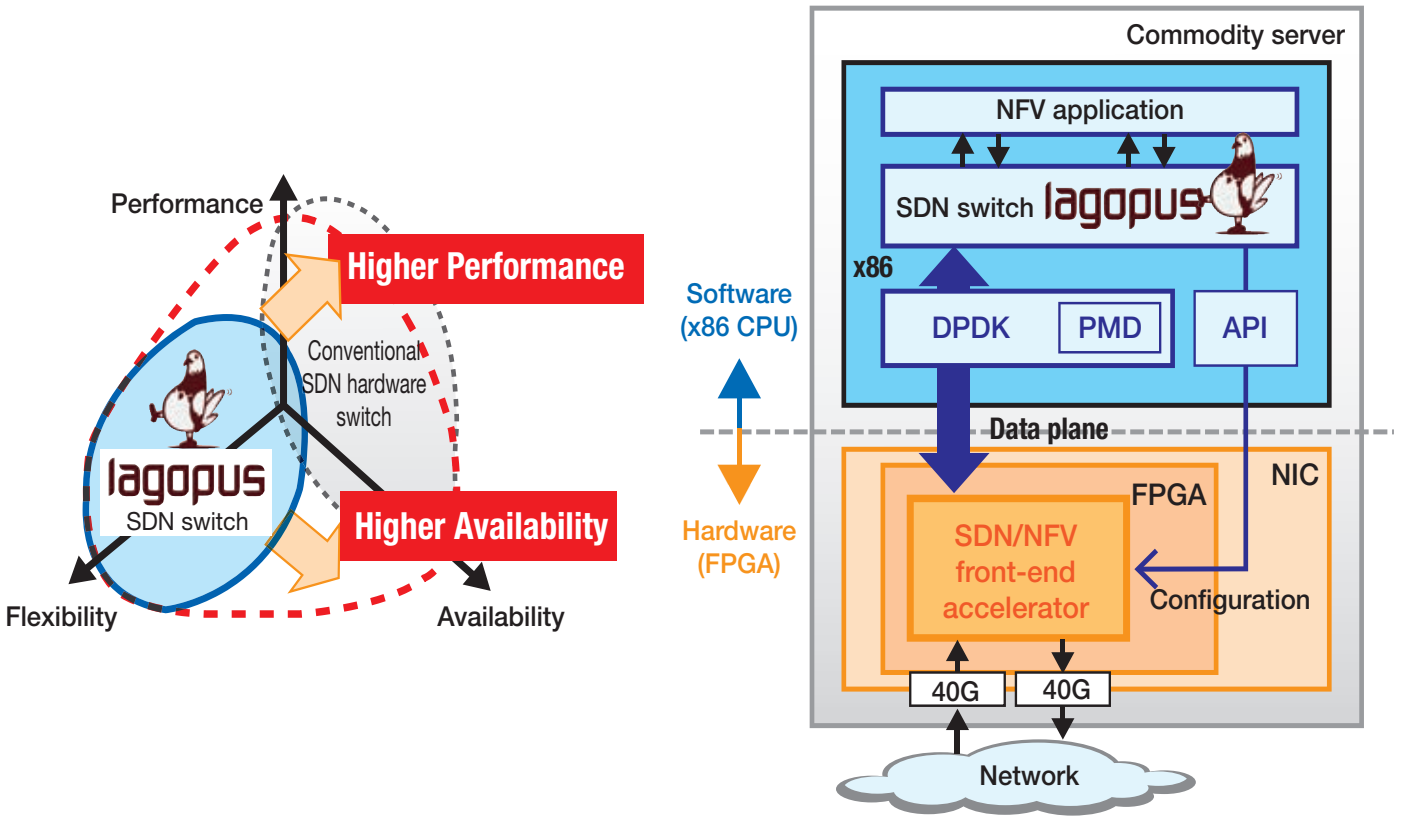


Figure 1 — Concept and architecture of Lagopus FPGA

launched an advanced SDN/NFV-related reearch endeavor called the O3 Project with funding from Japan's Ministry of Internal Affairs and Communications. Lagopus is major deliverable of the O3 Project to achieve high-performance software-packet processing and flexible flow control using the Open Networking Foundation's OpenFlow 1.3 protocol with commodity Intel x86 servers and a commodity NIC. The key benefits with Lagopus are high-performance software-packet processing at more than 10 Gbps on commodity servers; elastic network flow control for up to 1 million flow entries; and a scalable flow dispatcher for NFV applications such as virtual Provider Edge (vPE), virtual Customer Premises Equipment (vCPE) and virtual Evolved Packet Core (vEPC) frameworks.

The Lagopus FPGA project aims to explore 40/100-Gbps-capable, high-performance packet processing with flexible partitioning between software and hardware-accelerated functions on an FPGA running on commodity servers. Figure 1 shows the concept and architecture of Lagopus FPGA. The flexible architecture increases the Lagopus switch's 10-Gbps line rate as a pure software implementation to a 40-Gbps line rate via FPGA acceleration. This performance improvement comes at a cost of less than 10 percent of the x86 CPU's power dissipation. The architecture also greatly enhances our network troubleshooting ability, which is essential in a truly virtualized network.

Currently, we are co-designing an advanced software-programmable data plane for Lagopus and original hardware intellectual property (IP) for network carriers using a leading-edge FPGA and design tools, in expectation not only of gaining higher system performance, but also of reducing power and cost. In collaboration with a Xilinx team, we have successfully integrated Lagopus and our IP within 80-Gbps NIC demo boards based on Xilinx Virtex®-7 All Programmable FPGAs. We demonstrated Lagopus FPGA for the first time in February at NTT R&D Forum 2015 (Tokyo). We also presented our achievements [2] in August at Hot Chips 27 (Cupertino Calif.).

We leveraged the SDNet development environment to create the Lagopus FPGA system. The novel,

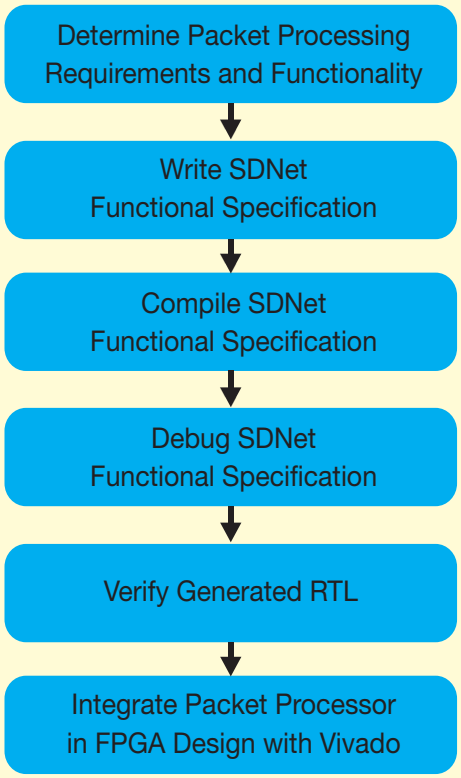
SDNet broadens Lagopus FPGA's potential utilization: The flexible, software-defined hardware design technology enables agile deployment of differentiated network services.

dynamically reprogrammable data plane packet-processing tool chain let us accelerate Lagopus and NFV applications by offloading high-intensity data plane operations such as packet classification, editing, search, load balancing and statics metering—all realized over various multigigabit Ethernet line rates (10/40/100 GbE)—to the FPGA NIC without compromising performance. We believe this is the best solution for our project to enforce the classification IP, a key component for SDN/NFV technology. The environment's quick, reconfigurable packet pipeline capability lets us quickly and easily update protocols and features for networking.

The SDNet environment broadens Lagopus FPGA's potential utilization by covering a broad range of use cases in both cloud computing data centers and wide area networks. For NTT, the flexible, software-defined hardware design technology enables agile deployment of differentiated network services.

DESIGN BASICS WITH THE SDNET ENVIRONMENT

With competition on the rise in the emerging market for SDN/NFV technology, one design challenge for the Lagopus FPGA project was to work within a tight development window in order to achieve timely deployment and promotion. We started designing the Lagopus FPGA system in October 2014 and completed our first integration just three months later, in January 2015.




```
class:: Tuple[out] {
  struct{
    DST_MAC  : 48,
    SRC_MAC  : 48,
    VLAN_ID  : 16,
    ETH_TYPE  : 16
  }
}L2KEY;
```

```
classVLAN :: Section[2] {
  ...
  method update ={
    L2KEY.VLAN_ID        =vlan,
  ...
  }
  method increment_offset = size[];
}
```

```
$ ppp -pxFile class_sdnet.px -bus lbus -dw 256 optimizedRTL
Target optimizedRTL completed.
```

Figure 2 — Design flow of the SDNet environment

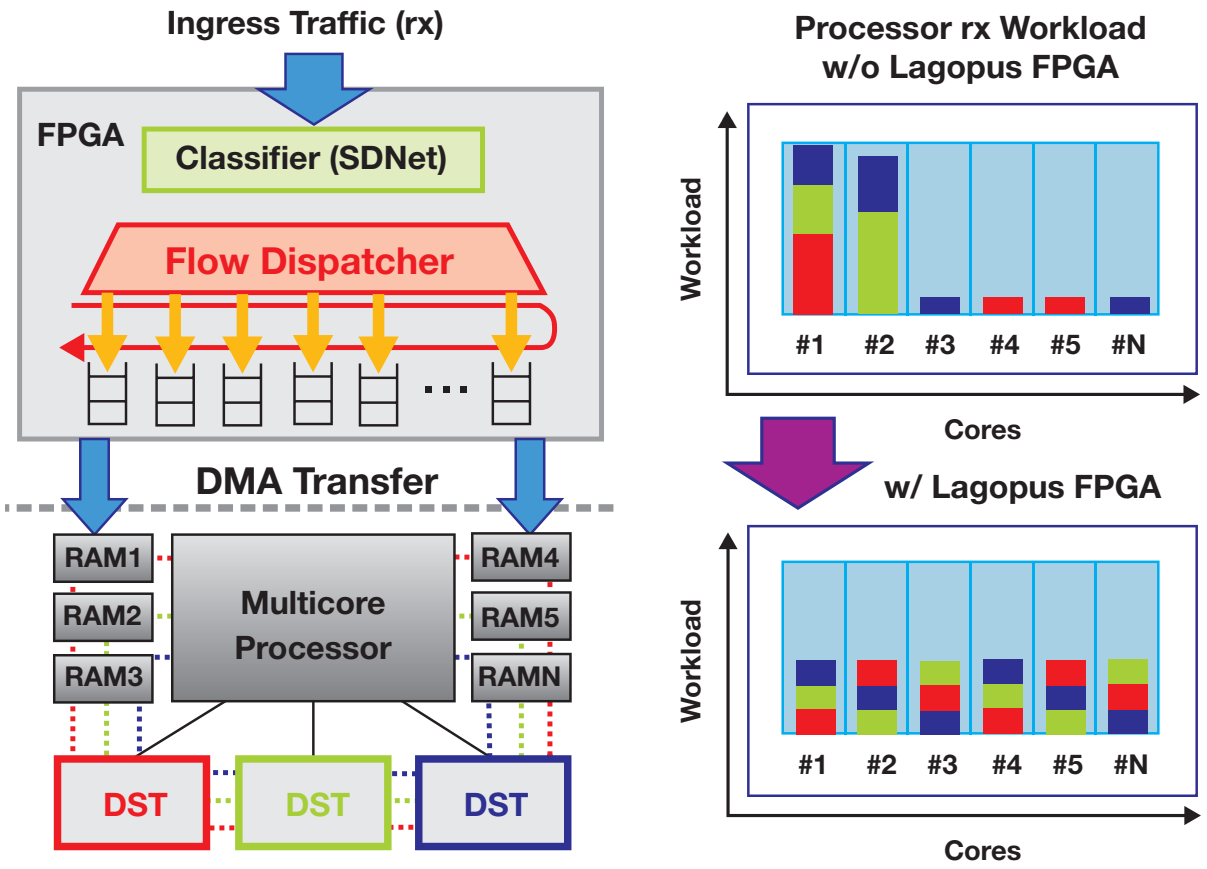


Figure 3 — FPGA flow classification and dispatch

That was quite an accomplishment, given the complexity of the system design. Figure 1 shows the top-level architecture of the Lagopus FPGA system, which comprises four technical software layers, including a soft FPGA IP bundle: (1) NFV applications; (2) the Lagopus software switch; (3) a hardware abstraction layer, such as an application programming interface (API) and Intel's Data Plane Development Kit (DPDK), a set of libraries and drivers for x86 fast packet processing; and (4) the FPGA NIC IP core suite. The multiple technical layers can make it difficult to trace the source of issues such as dropped packets and performance degradation, hampering the ability to debug and immediately isolate faults; indeed, this is a key challenge for all SDN/NFV architectures. To overcome these difficulties, we leveraged the SDNet environment and Xilinx's Vivado® Design Suite.

We started the design of Lagopus FPGA by determining our requirements for packet-processing functionality and mapping out a development flow. Figure 2 shows a general description of the development flow and an example code snippet of the SDNet specification. We decided to create a perfect-match filter that uses key information from a virtual LAN. With this, we can accelerate Lagopus' software data plane on the x86 by offloading hardware classification to the FPGA NIC. We can configure the filter entries via a DPDK flow director API by injecting flow entries with the OpenFlow protocol between Lagopus and the SDN controller.

To implement this strategy, we created a corresponding SDNet functional description as shown in the Figure 2 code snippet. We then fed the code into the SDNet compiler, specifying options such as bus type, bus width and generated RTL type. The compilation completed within a few seconds. The actual code size of the SDNet functional description was about 250 lines of code. In contrast, the RTL equivalent comprised several tens of thousands of code lines. Considering that we were working under an intense schedule, we very much appreciated the simplicity of the SDNet specification. It would have been impossible to design

We achieved 40-Gbps wire-speed software packet processing with Lagopus FPGA, at a cost of less than 10 percent of x86 power dissipation.

and verify such a complicated module in RTL from scratch given our development time constraints.

For the next step, we integrated the generated RTL with other peripheral IP on the Vivado Design Suite by employing a Tool Command Language (Tcl) shell. Figure 3 shows the integrated SDNet classifier and our customized flow dispatcher, which we targeted to program a Xilinx Virtex-7 XC7VX690T FPGA.

Since the classified packet flow (targeting 32 receive [rx] DMA queues) can be dispatched efficiently with Lagopus' software data plane on an x86 multicore CPU, the integrated FPGA design enables the system not only to reduce the CPU cycles of the OpenFlow worker threads of Lagopus, but also to balance the workload on each core (Figure 3). As a result, we achieved higher-performance, 40-Gbps wire-speed software packet processing with Lagopus FPGA, at a cost of less than 10 percent of the x86 CPU's power dissipation, as Figure 4 shows.

The SDNet environment and the Vivado Design Suite facilitated our project launch, letting us maximize the feature set, optimize the performance and lower the power consumption of the Lagopus FPGA system. NTT R&D's leadership in SDN/NFV and our use of Xilinx's SDNet development environment will enable us to bring revolutionary changes to the telecom and cloud infrastructure. Toward that end, we continue to refine our design technique by leveraging a softly defined, reprogrammable SDNet load module. Dynamic and rapid modification of the SDNet specification, including the API, will provide further benefit for us when we define future platforms. ■

REFERENCES

[1] *http://lagopus.github.io/*

[2] K. Yamazaki, Y. Nakajima, T. Hatano and A. Miyazaki, "Lagopus FPGA: A reprogrammable data plane for high-performance software SDN switches." Presented at Hot Chips 27, August 2015.

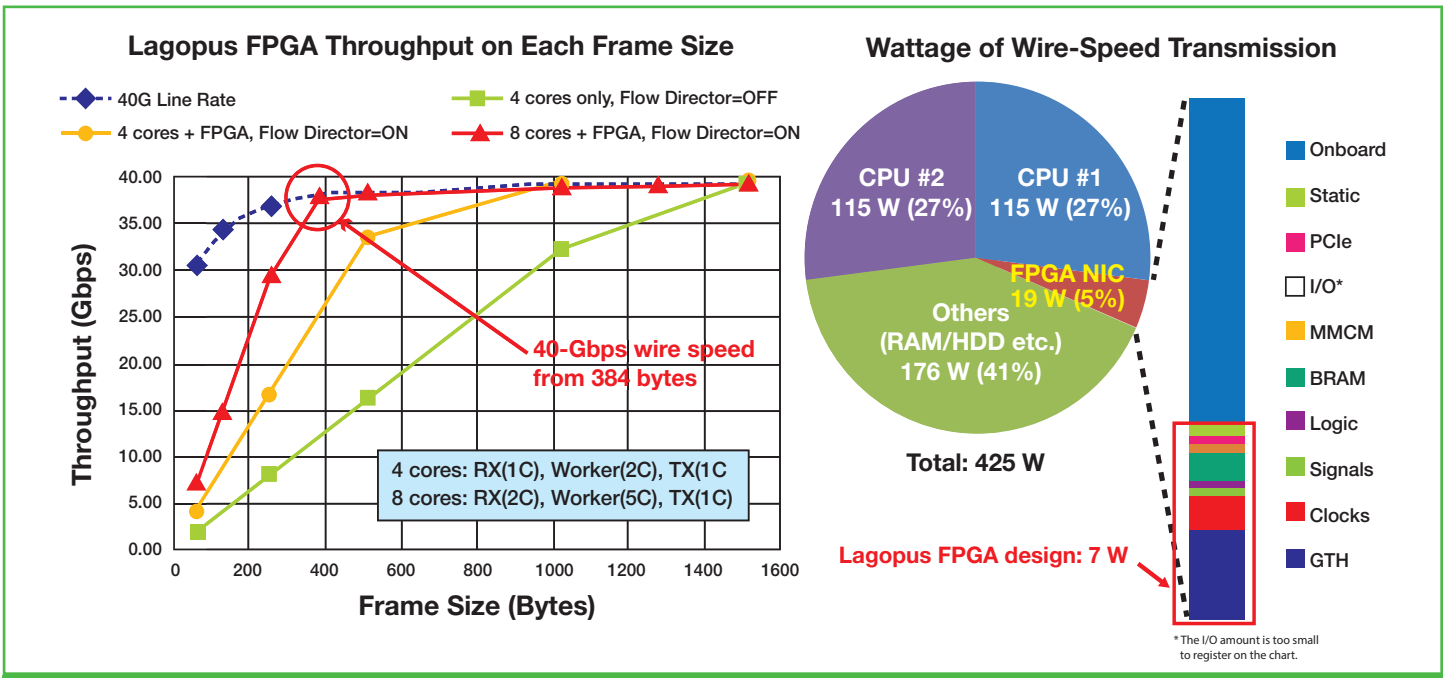


Figure 4 — Performance vs. power dissipation

Delivering FPGA Vision to the Masses

NI's Vision Development Module and Vision Assistant take machine vision from idea to prototype to application deployment.

by Kiran Nagaraj
Senior Software Engineer
National Instruments Corp.
kiran.nagaraj@ni.com

Christophe Caltagirone
Senior Software Engineer
National Instruments Corp.
christophe.caltagirone@ni.com

Dinesh Nair
Chief Architect
National Instruments Corp.
dinesh.nair@ni.com

Vision systems are going mainstream. The cost/benefit analysis and possible application of technology are now at a point where engineers are designing vision into everything from autonomous vehicles to consumer-electronics quality inspection systems. This mass adoption is driving vision out of the lab, into embedded systems and onto the factory floor. The deployed systems often require advanced synchronization with I/O, many widely distributed cameras or vision in the control loop. As processes and applications become more complex, vision systems are requiring faster and more advanced processing as well as tighter timing and synchronization.

To meet those requirements, vision system designers are increasingly relying on heterogeneous processing platforms comprising a combination of real-time processors and FPGA, GPU or DSP processing elements that can handle specialized tasks, I/O requirements and processing performance needs. Smart cameras, frame grabbers and vision systems are all leveraging heterogeneous architectures to meet application requirements.

The parallel processing capability of FPGAs, such as those in the Xilinx® All Programmable FPGA lineup, is a natural fit for implementing many image processing algorithms. FPGAs can be used for performing both data-intensive processing and high-speed sensor measurements. The devices also have incredibly low latency, which is critical for vision applications because latency accounts for the time that elapses until a decision is made based on the image data. FPGAs can help avoid jitter and thus serve as highly deterministic processing units.

Building a heterogeneous system that includes an FPGA, however, introduces serious programming challenges for system designers. As time-to-market pressures mount, vision system designers need the ability to prototype a solution with complex features quickly. Programming on heterogeneous

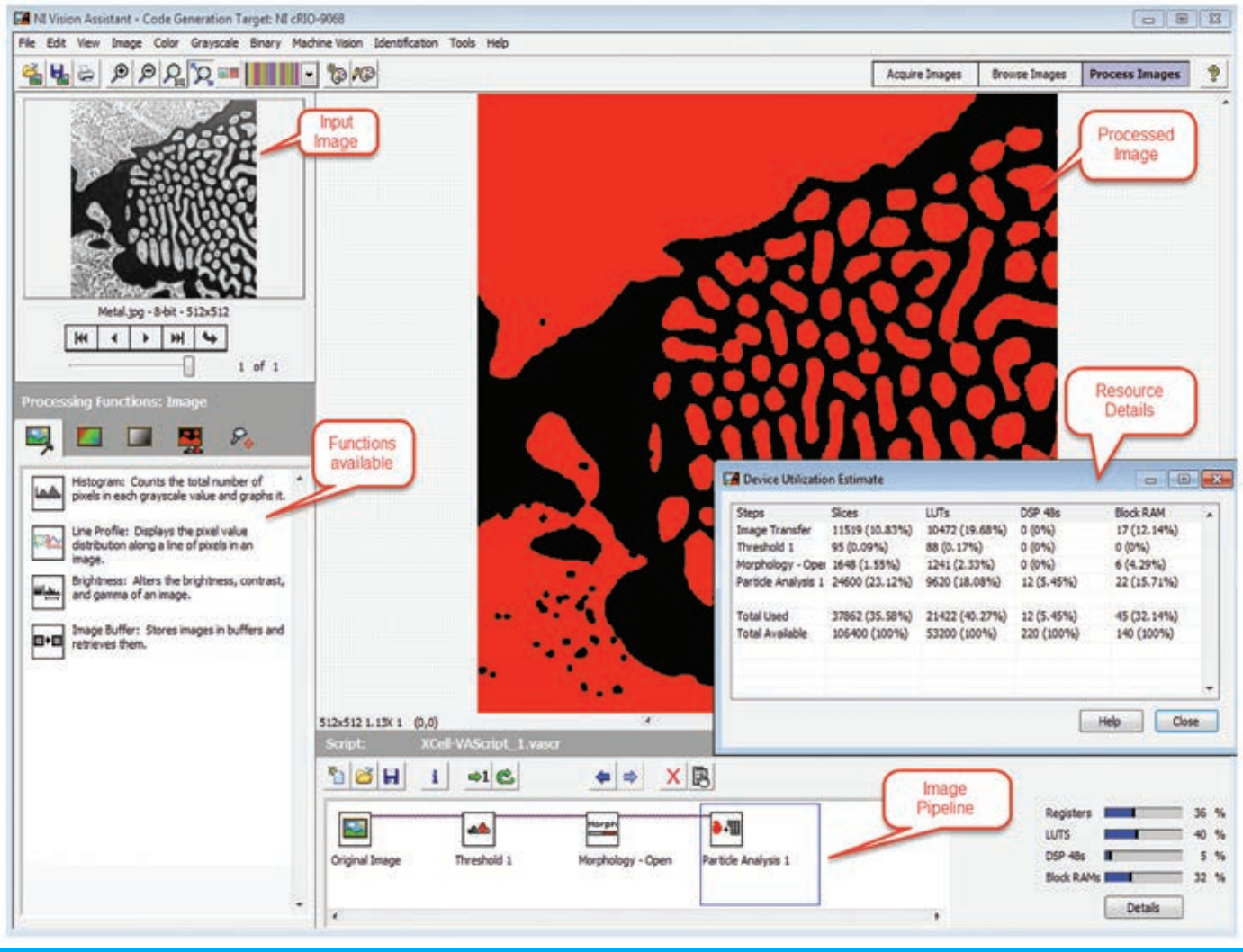


Figure 1 — Vision Assistant with device utilization estimates

systems requires a tool that can help the domain expert design intellectual property (IP) functions on multiple platforms and test the vision algorithm before compiling and running the algorithm on the target hardware. The tool should allow easy access to throughput and resource usage information throughout the prototyping process.

NI refers to this as algorithm engineering: the process by which you, the domain expert, can focus on solving the problem at hand without being preoccupied with the underlying hardware technology. NI's Vision Development Module (VDM) with Vision Assistant arms you with that capability.

VDM with Vision Assistant helps in fast prototyping and code generation, FPGA resources estimation, automatic code parallelization, and synchronization of parallel streams (for such tasks as latency balancing). VDM includes more than 50 FPGA image processing functions as well as functions to transfer images efficiently between the processor and the FPGA. You can use Vision Assistant within VDM to rapidly prototype and develop FPGA vision applications.

CONFIGURATION-BASED PROTOTYPING

Vision Assistant is a configuration-based prototyping tool that empowers you to iterate on image processing algorithms and see how changes in parameters affect the image. With Vision Assistant, you can visualize the output (processed image) after every vision block in an image pipeline (Figure 1). You can use the tool to test different algorithms and parameters on different sets of images without having to compile

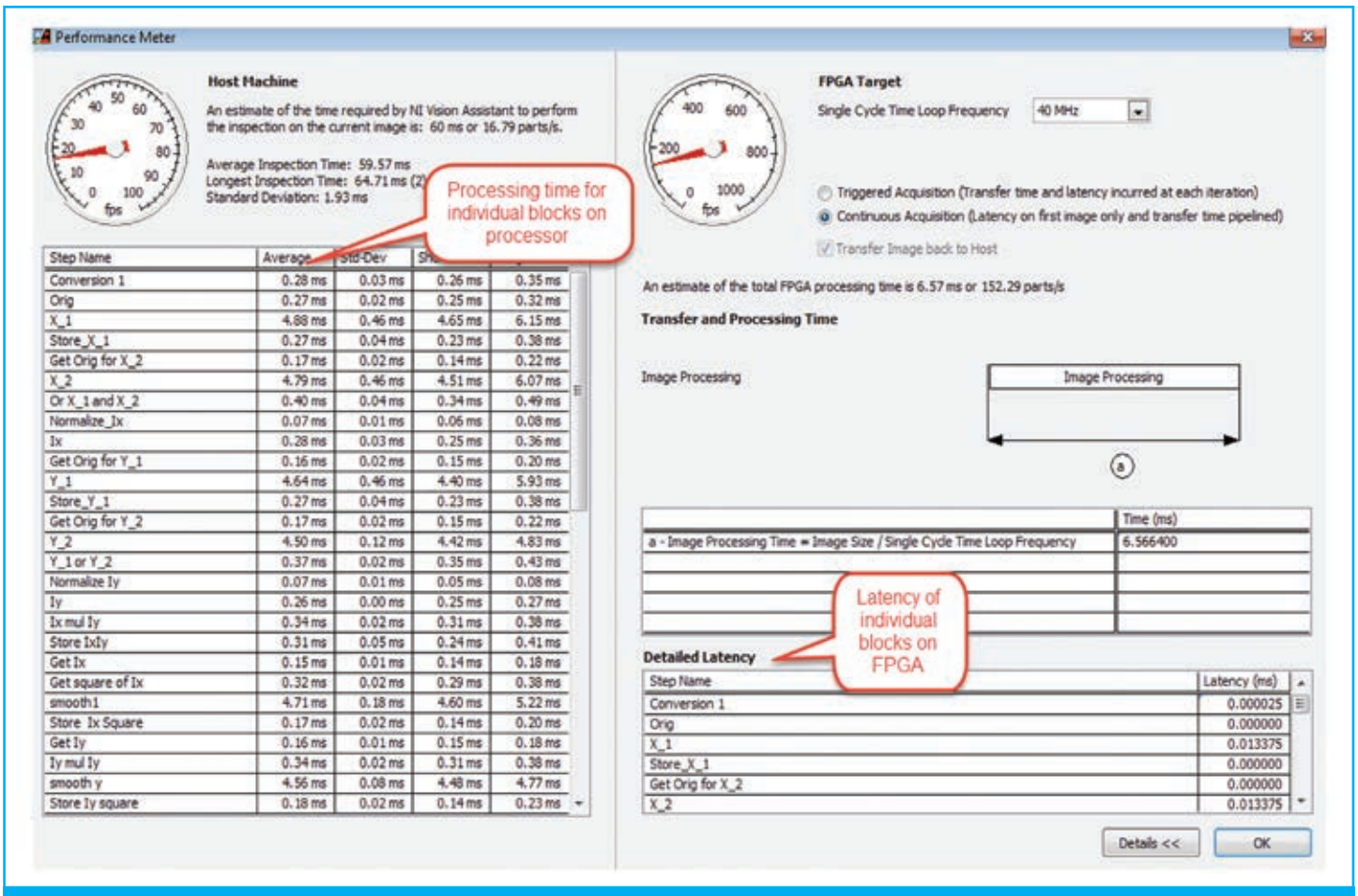
You can use Vision Assistant to test the results of algorithms in the prototyping environment and the deployed code to ensure that your implementation yields the same results.

your IP, thereby greatly reducing the time required to design your vision algorithm.

NI has customized the tool to handle FPGA programmers' requirements. The key concerns when building any algorithm on an FPGA are resource consumption on the FPGA fabric, the latency of the pipeline and the maximum frequency the algorithm can achieve on a specific fabric. Vision Assistant helps by providing an estimate of the resources consumed for each block in the image pipeline. You can use the tool to test the results of algorithms in the prototyping environment and the deployed code to ensure that your implementation yields the same results.

One consideration is which kernel size to use for an image filtering operation. The choice of kernel size affects resource usage and latency in the pipeline, with a larger kernel usually requiring more resources than a smaller one.

To select the most appropriate kernel size for your application, you can use Vision Assistant to experiment until you achieve the best performance in terms of minimal resource consumption and maximum performance. A real-time estimate of



Step Name	Average	Std-Dev		
Conversion 1	0.28 ms	0.03 ms	0.26 ms	0.35 ms
Orig	0.27 ms	0.02 ms	0.25 ms	0.32 ms
X_1	4.88 ms	0.46 ms	4.65 ms	6.15 ms
Store_X_1	0.27 ms	0.04 ms	0.23 ms	0.38 ms
Get Orig for X_2	0.17 ms	0.02 ms	0.14 ms	0.22 ms
X_2	4.79 ms	0.46 ms	4.51 ms	6.07 ms
Or X_1 and X_2	0.40 ms	0.04 ms	0.34 ms	0.49 ms
Normalize_Ix	0.07 ms	0.01 ms	0.06 ms	0.08 ms
Ix	0.28 ms	0.03 ms	0.25 ms	0.36 ms
Get Orig for Y_1	0.16 ms	0.02 ms	0.15 ms	0.20 ms
Y_1	4.64 ms	0.46 ms	4.40 ms	5.93 ms
Store_Y_1	0.27 ms	0.04 ms	0.23 ms	0.38 ms
Get Orig for Y_2	0.17 ms	0.02 ms	0.15 ms	0.22 ms
Y_2	4.50 ms	0.12 ms	4.42 ms	4.83 ms
Y_1 or Y_2	0.37 ms	0.02 ms	0.35 ms	0.43 ms
Normalize Iy	0.07 ms	0.01 ms	0.05 ms	0.08 ms
Iy	0.26 ms	0.00 ms	0.25 ms	0.27 ms
Ix mul Iy	0.34 ms	0.02 ms	0.31 ms	0.38 ms
Store IxIy	0.31 ms	0.05 ms	0.24 ms	0.41 ms
Get Ix	0.15 ms	0.01 ms	0.14 ms	0.18 ms
Get square of Ix	0.32 ms	0.02 ms	0.29 ms	0.38 ms
smooth1	4.71 ms	0.18 ms	4.60 ms	5.22 ms
Store Ix Square	0.17 ms	0.02 ms	0.14 ms	0.20 ms
Get Iy	0.16 ms	0.01 ms	0.15 ms	0.18 ms
Iy mul Iy	0.34 ms	0.02 ms	0.31 ms	0.38 ms
smooth y	4.56 ms	0.08 ms	4.48 ms	4.77 ms
Store Iy square	0.18 ms	0.02 ms	0.14 ms	0.23 ms

Step Name	Latency (ms)
Conversion 1	0.000025
Orig	0.000000
X_1	0.013375
Store_X_1	0.000000
Get Orig for X_2	0.000000
X_2	0.013375

Figure 2 — Performance meter utility

The vision FPGA IP of the Vision Development Module lets developers use massively parallel processing and the Vivado High-Level Synthesis tool to achieve fully pipelined, low-latency, architecture-optimized vision functions on the FPGA.

vision IP functions, as in Figure 1, is a useful feature to have during prototyping.

Running multiple image pipelines in parallel is a common requirement. Such scenarios dictate that at the time the pipelines merge into a single pipeline, the latency of the parallel pipelines must be balanced. NI provides a synchronization buffer as part of its vision FPGA IP toolset. Vision Assistant automatically computes the latencies in the pipeline and ensures balanced latency at the time that the parallel pipelines merge to configure the synchronization buffer for you. This guarantees that the FIFOs in the synchronization buffers have sufficient depth based on the maximum latency of the pipelines.

The performance meter utility in Vision Assistant estimates the maximum time taken for processing each frame (Figure 2), letting you know the collective latency of all the blocks in the pipeline. Most of the processors in NI's hardware portfolio have a real-time operating system running on them, so using Vision Assistant makes it easy to estimate the time required to execute a vision function.

For those who are new to LabVIEW, it should be noted that Vision Assistant ensures the creation of a fully functional project, including all dependencies, such as transfer virtual instruments (VIs) and DMA FIFOs, and the image acquisition logic. A VI is similar to a function or subroutine in other programming languages. Transfer VIs are required to transfer the image data between the host/acquisition logic and the FPGA. DMA FIFOs do not involve the host processor; therefore, they are the fastest available method for transferring large amounts of data between the FPGA target and the host. The acquisition logic depends on whether the vision system is based on inline processing or coprocessing. Vision Assistant also helps you create other VIs, such as the Host VI, which runs on the processor, and the FPGA VI. You would then compile the FPGA VI using Xilinx Vivado® tools to generate a bitstream for deployment on the FPGA.

It is important to note that the system that houses the image processing pipeline can be broadly categorized as inline processing or coprocessing, depending on where the acquisition logic resides. In inline processing, the acquisition logic resides on the FPGA; the camera is configured using the acquisition logic and the image is processed on the FPGA. The results and the processed image are then sent back to the host for evaluation and further analysis. In coprocessing, the acquisition logic for the camera resides on the processor. Transferring the image from the processor to the FPGA and then sending the processed image back from the FPGA to the processor require a finite amount of time. You also can partition the processing of the image pipeline between the processor and the FPGA.

As a developer of a vision system that uses an FPGA, you need to be aware of the throughput that the FPGA can achieve. You can use throughput information and real-time resource estimation to determine how many functions (IP blocks) you can deploy to the FPGA. In a coprocessing scenario, the processor performance determines the final throughput. This is true when using the FPGA IP functions that NI ships with the Vision Development Module because those functions are fully pipelined and yield better performance than most processors.

PROTOTYPE TO DEPLOYMENT

The vision FPGA IP of the Vision Development Module lets developers use massively parallel processing and the Xilinx Vivado High-Level Synthesis (HLS) tool to achieve fully pipelined, low-latency, architecture-optimized vision IP on the FPGA. Vision FPGA IP from NI currently targets three Xilinx FPGA families—Kintex®-7, Virtex®-5 and Spartan®-6—as well as the Xilinx Zynq®-7000 All Programmable SoCs.

An image can be viewed as a two-dimensional array, and operation on an image is mostly matrix-based. FPGAs' inherent parallelism enables their high-speed performance. You can achieve the matrix operations on the image using loops; you can unroll the loops and take advantage of the parallelism feature of the FPGA to perform several tasks after unrolling. LabVIEW FPGA and the LabVIEW FPGA IP Builder are the primary tools developers use to create vision IP on FPGAs.

Vision FPGA IP functions are single-pixel processing, so they accept 1 pixel from a pixel stream and then output 1 pixel. The IP functions interact with one another using enable-based handshaking or a four-wire handshaking protocol. The primary reason for this implementation is that the complexity of the control path increases with the number of functions in the image pipeline, thus requiring a seamless handover of data between the functions. The four-wire protocol ensures lossless data transfer between vision FPGA IP functions placed in a single-cycle timed loop (SCTL); using the SCTL ensures that the modules in the loop clock at a user-specified frequency.

Figure 3 shows an FPGA VI that depicts the four-wire protocol and the synchronization buffer to merge the pixel. The four-wire protocol is designed for algorithms that run in parallel; it improves throughput by ensuring that the data is processed in a producer-consumer architecture. Further, the four-wire handshake consumes minimal resources on the FPGA. This is critical because the protocol constitutes overhead for the underlying vision functionality.

Vision FPGA IP also gives you the flexibility of adding custom code within the pipeline to provide an open environment. The custom code requires a wrapper VI that has the four-wire handshake implementation. You can then insert custom code in the image pipeline. You must ensure that the custom code is fully pipelined; otherwise it might affect the integrity of the pipeline. You can implement your custom

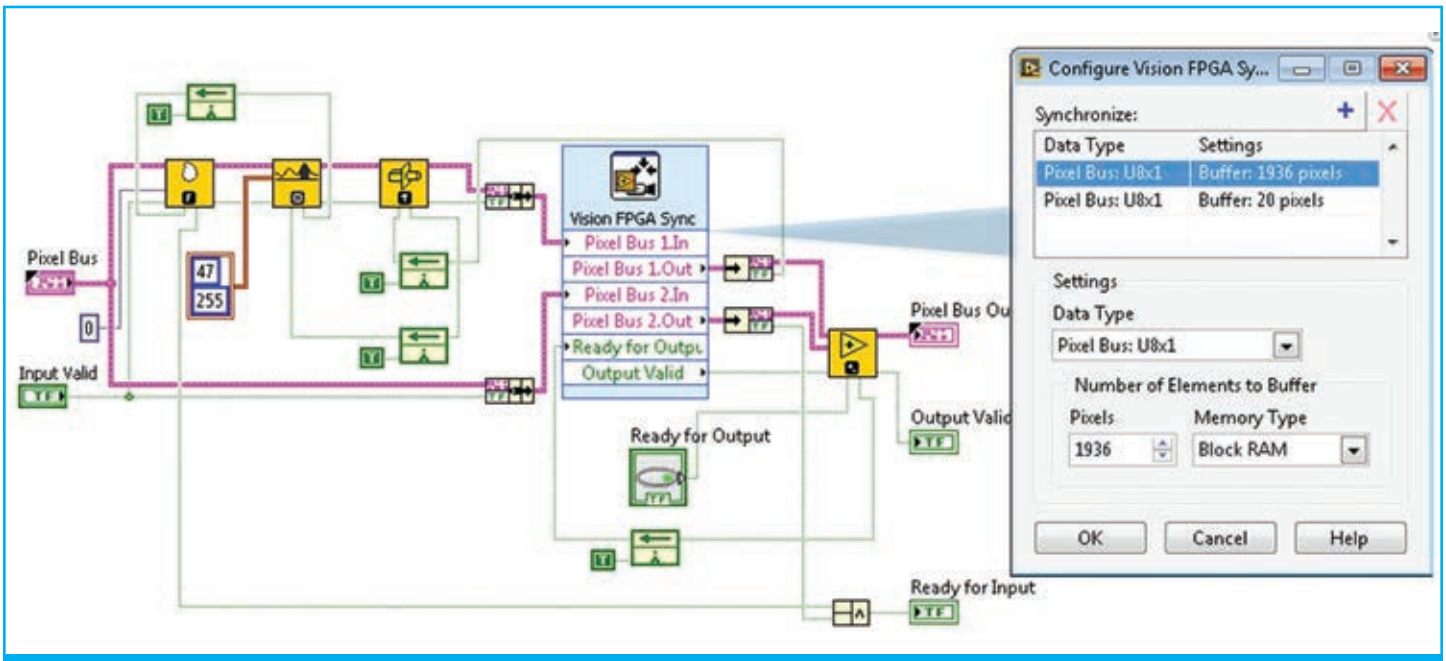


Figure 3 — Internals of the FPGA VI with a synchronization node

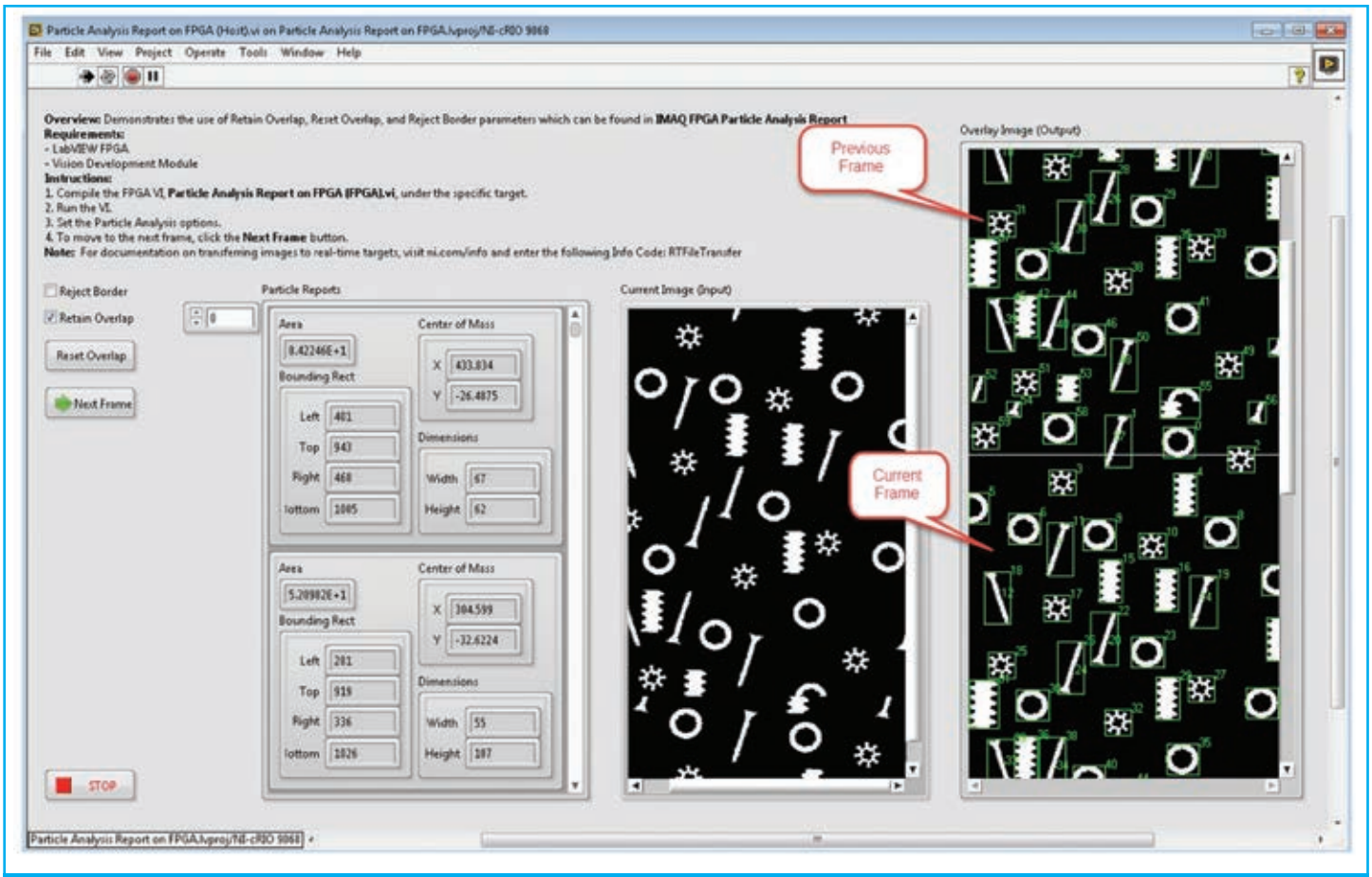


Figure 4 — Particle Analysis Report example with retain overlap enabled

code using LabVIEW, or you can use existing code in VHDL through an HDL integration node in LabVIEW FPGA.

The vision FPGA IP toolset provides preprocessing functions such as edge detection filters, convolution filters, lowpass filters, gray morphology, binary morphology and threshold. It also includes vision IP functions that perform arithmetic and logical operations, as well as functions that output results such as the centroid. Another function, the Simple Edge Tool, finds edges along a line and is useful for caliper applications. The Quantify function accepts a masked image as well as the image stream to be processed and returns a report that has information about the area, mean and standard deviation of the regions defined by the masked image. Linear Average computes the average pixel intensity (mean line profile) on all or part of the image.

The latest addition to NI's vision FPGA IP list is the Particle Analysis Report. You can perform particle analysis, or blob analysis, to detect connected regions or groupings of pixels in an image and then make selected measurements of those regions. With this information, you can detect flaws on silicon wafers, detect soldering defects on electronic boards or locate objects in motion control applications.

A unique feature of this IP is that it can detect particles when the particle information is spread across two frames. NI ships a Particle Analysis Report example with VDM; Figure 4 shows the Host VI with the image display. This capability is needed in inspection systems, where you cannot always ensure that the objects under inspection are captured in a single frame.

Nearly 70 percent of NI's vision FPGA IP functions were developed using the IP Builder, a utility in LabVIEW FPGA that allows you to code in graphical code using LabVIEW and then output RTL code using Vivado HLS. The major advantage of this approach is that users familiar with graphical coding can develop the application along with a directive file that states their frequency and latency

Vivado HLS is a good fit for vision development because it helps abstract algorithmic descriptions and data-type specifications from the generated C code of the IP Builder.

requirements. Using LabVIEW IP Builder with Vivado HLS generates the appropriate VHDL code. You can use array-based operations on images, and Vivado HLS ensures that, based on the directives set, the VI will achieve the required frequency of operation and minimum latency.

Vivado HLS is a good fit for vision development because it helps abstract algorithmic descriptions and data-type specifications (integer, fixed-point) from the generated C code of the IP Builder. It also generates the necessary simulation models for early testing of functionality. The generated architecture-aware VHDL code yields high-quality, highly repeatable results.

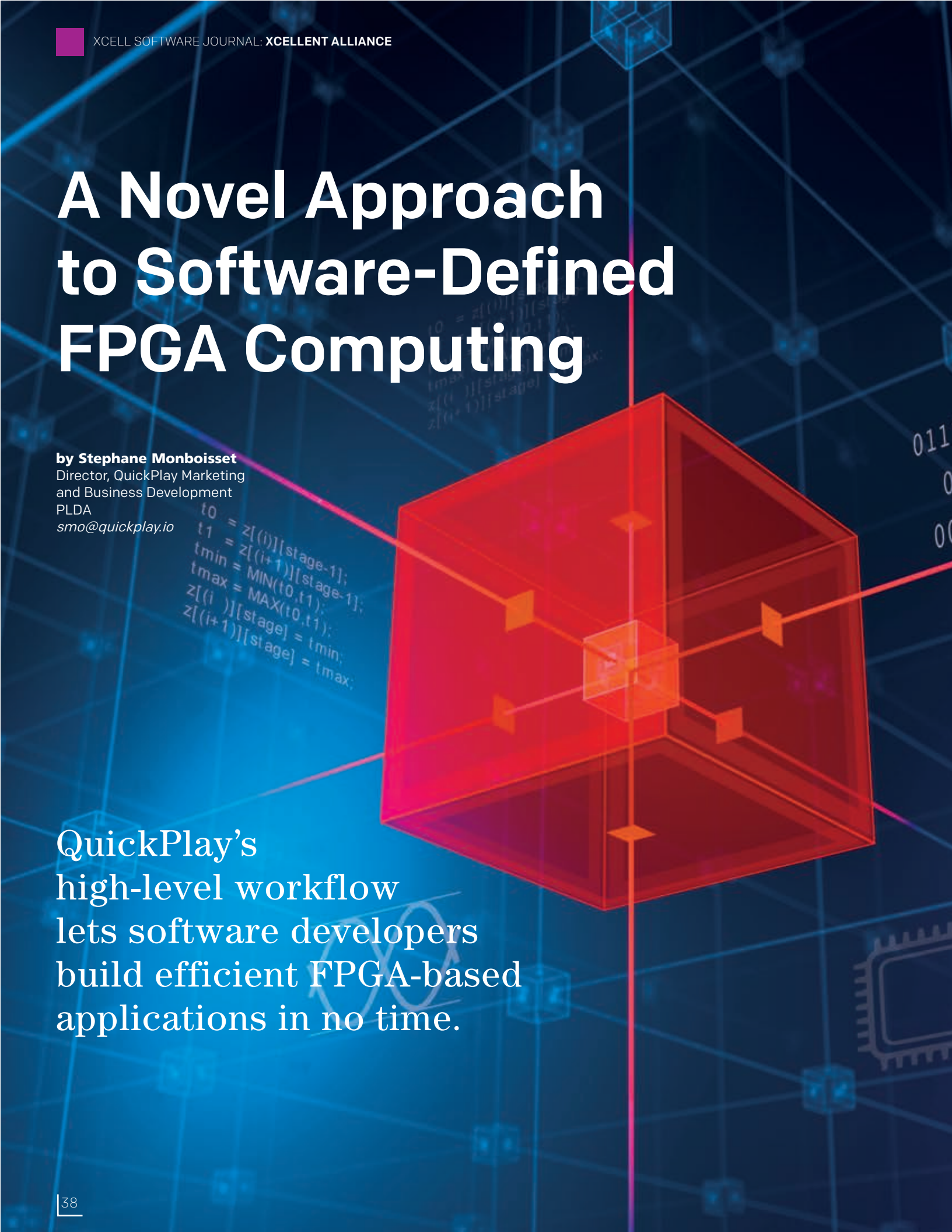
NI is committed to the concept of providing open, flexible systems with the right software tools to leverage them. Developers are designing vision systems based on heterogeneous architectures into a growing range of applications. The next frontier for the software design of these heterogeneous systems could be for the compiler or application development engine to decide intelligently where to deploy the components of an algorithm, using the capabilities and resources of the various system components (CPU, GPU and FPGA) to make that determination.

As more-advanced products and processes push the limits of what vision systems are asked to do, application developers will require an effective prototyping and algorithm development environment for vision functionality. Providing the right tools to developers and domain experts will fuel the next wave of innovation in vision system design for the masses.

If you are interested in trying out NI's vision FPGA IPs, you need to install LabVIEW FPGA and VDM. You can do so initially for a 30-day evaluation period and then extend or purchase the license at *ni.com/vision*. ■

A Novel Approach to Software-Defined FPGA Computing

by Stephane Monboisset
Director, QuickPlay Marketing and Business Development
PLDA
smo@quickplay.io



QuickPlay's high-level workflow lets software developers build efficient FPGA-based applications in no time.

With the rise of the Internet of Things and Big Data processing, the need for transferring and processing data has skyrocketed, and CPUs alone can no longer address the exponential increase. Adding more processors and more virtual machines to run a given application just doesn't cut it, as there is only so much that can be parallelized on multiple CPUs for a given application. Field-programmable gate arrays, on the other hand, have the requisite I/O bandwidth and processing power, not only from a pure processing standpoint but, equally important, from a power standpoint. For data-center equipment manufacturers, the use of FPGAs has long been an appealing prospect. Intel's recent acquisition of the second-largest FPGA vendor is further testament that a CPU-only solution no longer suffices.

The major roadblock to more-widespread FPGA adoption has been the complexity of implementing them. Until now, the only way to develop an application on an FPGA-based platform has been to deal with some of the lowest levels of hardware implementation. This has kept a large potential customer base—software developers—away from the devices and has made life increasingly complicated for traditional FPGA designers.

Recent methodologies for FPGA design, centered on high-level synthesis (HLS) tools and leveraging software programming languages such as OpenCL™, C and C++, have provided a sandbox for software developers to reap the benefits of FPGA-based hardware acceleration in numerous applications. But the methodologies often fall short in one essential respect: enabling software developers to define and configure, on their own, the hardware infrastructure best suited for their application. The industry has continued to pursue the holy grail of a high-level workflow for implementing applications on FPGA-based platforms that does not require specific FPGA expertise.

Over the past five years, PLDA has developed just such a workflow. Called QuickPlay, it efficiently addresses the implementation complexity challenge and enables multiple use models for FPGA development. But one of its core sources of value is the way in which it lets software developers take applications intended for CPUs and implement them, partially or fully, on FPGA hardware. QuickPlay leverages all of the FPGA resources, turning these powerful but complex devices into software-defined platforms that yield the benefits of FPGAs without the pain of hardware design.

Consider a software algorithm that can be broken down into two functions: Data is processed into one function and is then sent to another for further processing. From a software perspective, this implementation is as simple as a call to Function1() followed by a separate call to Function2(), using pointers to the location of the data to be processed.

Implementing such an algorithm on an FPGA-based hardware platform without the right hardware abstraction tool flow would require the software developer to come up with a hardware design resembling that in Figure 1 (where Kernel 1 and Kernel 2 are the respective hardware implementations of Function 1 and Function 2). The hardware design would need to include two elements: the control plane and the data plane.

The control plane is the execution engine that generates clocks and resets, manages system startup, orchestrates data plane operations, and performs all housekeeping functions. The data plane instantiates and connects the processing elements, Kernel 1 and Kernel 2, as well as the necessary I/O interfaces required to read data in and write processed data out. In our example, those interfaces are Ethernet and PCI Express (PCIe), as Figure 1 shows, though different application requirements will call for different I/O interfaces.

A software developer could easily generate Kernel 1 and Kernel 2 using an HLS tool that compiles the

QuickPlay leverages all of the FPGA resources, turning these powerful but complex devices into software-defined platforms that yield the benefits of FPGAs without the pain of hardware design.

software functions Function1() and Function2(), typically written in C or C++, into FPGA hardware descriptions in VHDL or Verilog, without requiring specific hardware expertise. Every other element in the design that is not algorithmic in nature (interfaces, control, clocks and resets), however, could not be generated with HLS tools, and hardware designers would have to design them as custom hardware description language functions or IP. The job of sourcing those elements and connecting them poses yet another challenge, as some elements may not be readily available or may have different interfaces (type and size), clocking requirements, specific startup sequences and so on.

Beyond the design work—and equally challenging—is the implementation work, which includes mapping the design onto the resources of the selected FPGA platform, generating the appropriate constraints, and confirming that those constraints are met after logic synthesis and implementation on the FPGA hardware. It can take even an experienced hardware designer weeks to achieve a working design on a new piece of FPGA hardware.

Thus, any tool that aims to enable software developers to augment their applications with custom hardware must be able to:

- create functional hardware from pure software code;
- incorporate existing hardware IP blocks if needed;
- infer and create all of the support hardware (interfaces, control, clocks, etc.);
- support the use of commercial, off-the-shelf boards and custom platforms;
- ensure that the generated hardware is correct by construction so that it requires no hardware debug; and
- support debug of functional blocks using standard software debug tools only.

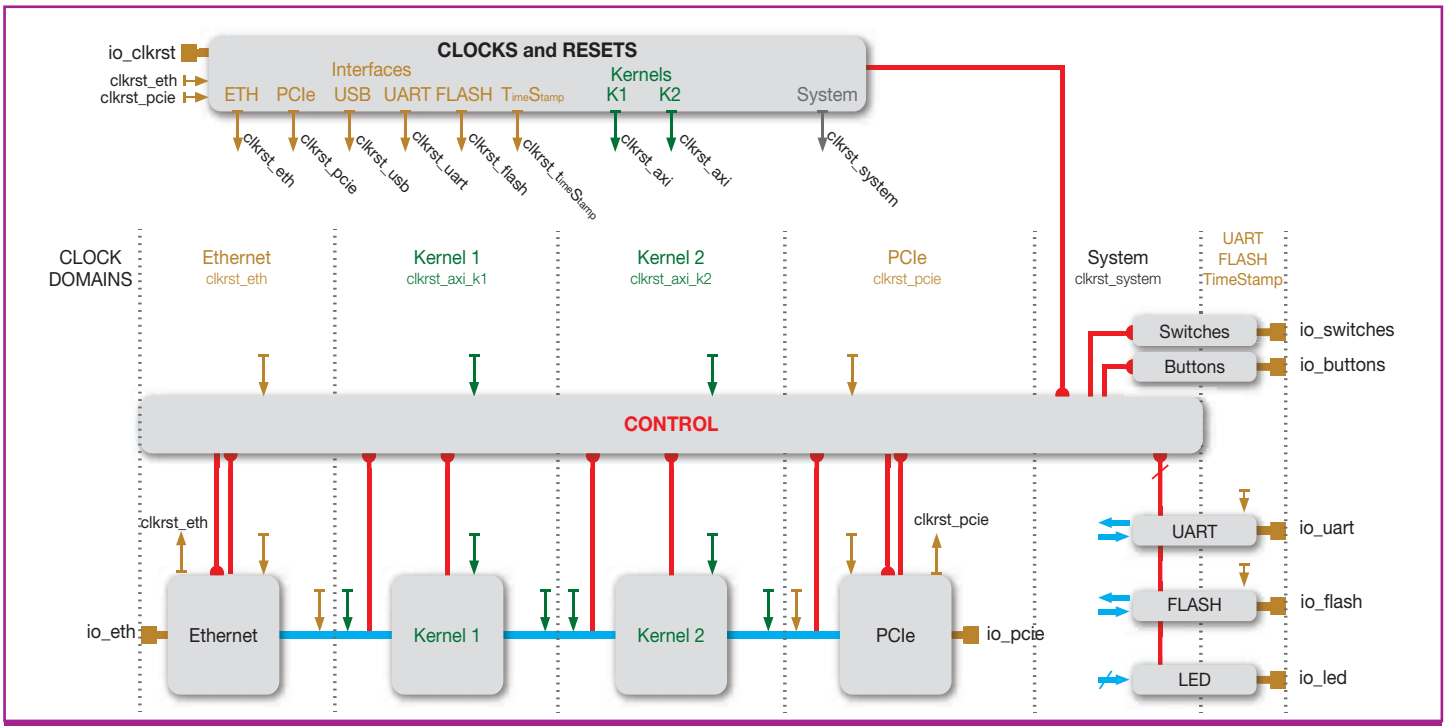


Figure 1 — A detailed hardware implementation of a two-function algorithm using traditional FPGA tools

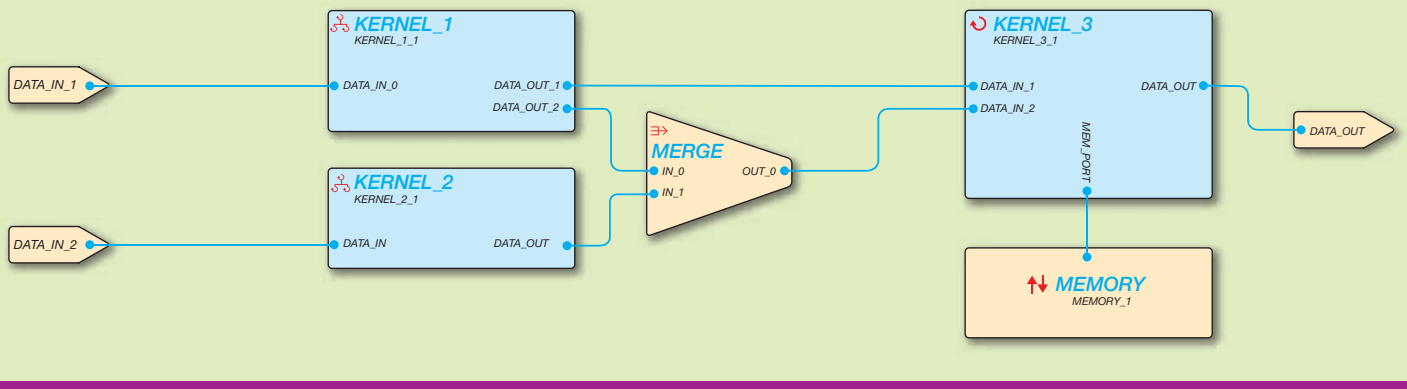


Figure 2 — A design example in QuickPlay

PLDA engineered QuickPlay from the ground up to meet all of those requirements, thereby enabling pure software developers to specify, build and integrate FPGAs into their software architectures with minimal effort.

SOFTWARE-CENTRIC METHODOLOGY

The overall process of implementing a design using QuickPlay is straightforward:

1. Develop a C/C++ functional model of the hardware engine.
2. Verify the functional model with standard C/C++ debug tools.
3. Specify the target FPGA platform and I/O interfaces (PCIe, Ethernet, DDR, QDR, etc.).
4. Compile and build the hardware engine.

The process seems simple; but if it is to work seamlessly, it is critical that the generated hardware engine be guaranteed to function identically to the original software model. In other words, the functional model must be deterministic so that, no matter how fast the hardware implementation runs, both hardware and software executions will yield the exact same results.

Unfortunately, most parallel systems suffer from nondeterministic execution. Multithreaded software execution, for example, depends on the CPU, on the OS and on nonrelated processes running on the same host. Multiple runs of the same multithreaded program can have different behaviors. Such nondeterminism in hardware would be a nightmare, as it would require debugging the hardware engine itself, at the electrical waveform level, and thus would defeat the purpose of a tool aimed at abstracting hardware to software developers.

QuickPlay uses an intuitive dataflow model that mathematically guarantees deterministic execution, regardless of the execution engine. The model consists of concurrent functions, called kernels, communicating with streaming channels. It thus correlates well with how a software developer might sketch an application on a whiteboard. To guarantee deterministic behavior, the kernels must communicate with each other in a way that prevents data hazards, such as race conditions and deadlocks. This is achieved with streaming channels that are (1) FIFO-based, (2) blocking read and blocking write, and (3) point-to-point.

Those are the characteristics of a Kahn Process Network (KPN), the computation model on which PLDA built QuickPlay. Figure 2 shows a QuickPlay design example illustrating the KPN model.

The contents of any kernel can be arbitrary C/C++ code, third-party IP or even HDL code (for the hardware designers). QuickPlay then features a straightforward design flow (Figure 3).

Let's take a closer look at each step of the QuickPlay design process.

Step 1: Pure software design. At this stage you create your FPGA design by adding and connecting processing kernels in C and by specifying the communication channels with your host software. QuickPlay's Eclipse-based integrated development environment (IDE) provides a C/C++ library with a simple API to create kernels, streams, streaming ports and memory ports, and to read and write to and from streaming ports and memory ports.

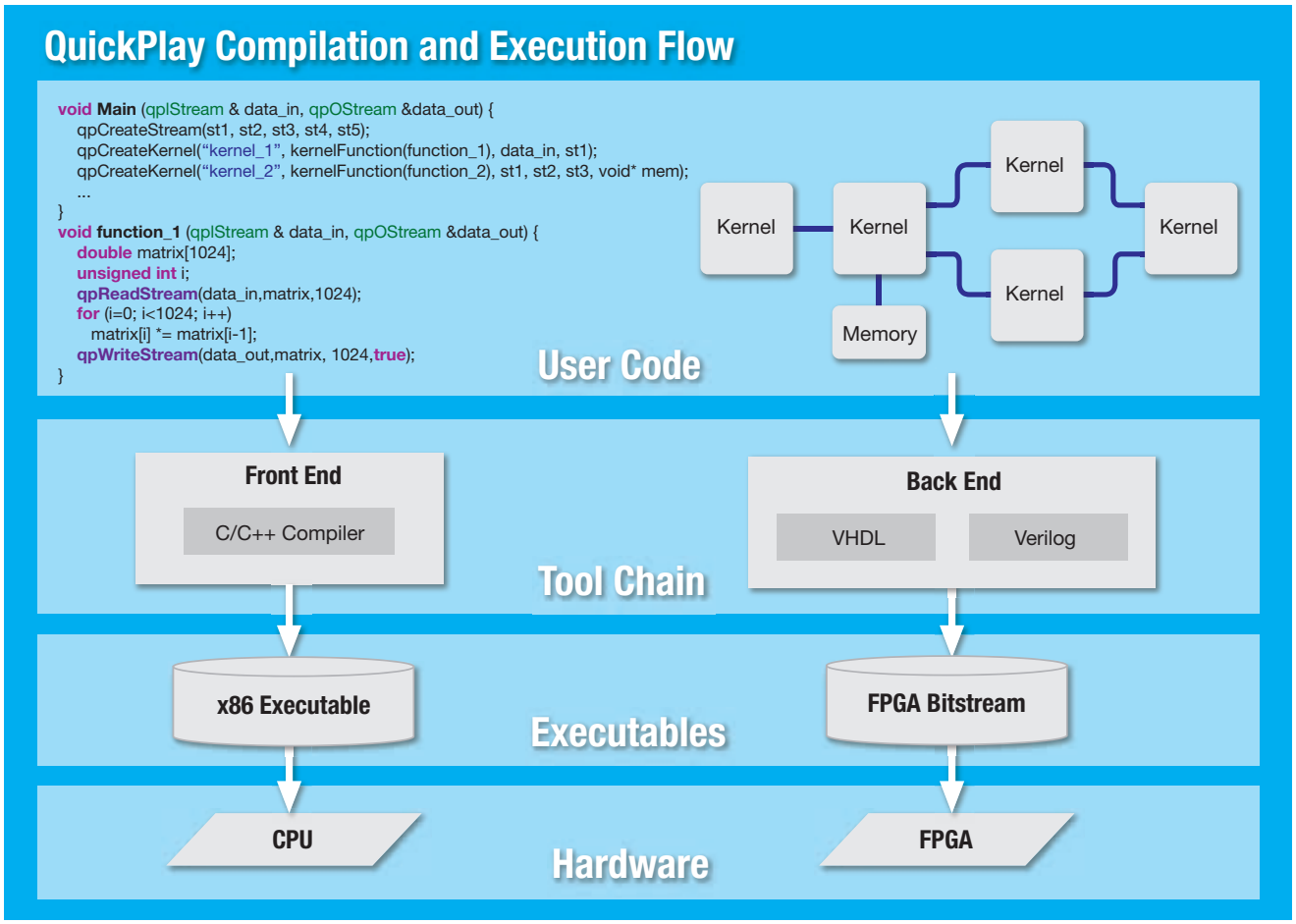


Figure 3 – QuickPlay features a straightforward design flow.

In addition, the QuickPlay IDE provides an intuitive graphical editor that allows you to drag and drop kernels and other design elements and to draw streams.

Step 2: Functional verification. In this step, the focus is on making sure that the software model written in Step 1 works correctly. You do this by compiling the software model on the desktop, executing it with a test program that sends data to the inputs, and verifying the correctness of the outputs. The software model of the FPGA design is executed in parallel, with a distinct thread for each kernel to mimic the parallelism of the actual hardware implementation.

You would then debug your software model using standard software debug techniques and tools such as breakpoints, watchpoints, step-by-step execution and printf. (You will probably want to run more tests once the implementation is in hardware; we'll deal with that shortly.) From a design flow standpoint, this is where you do all of your verification. Once you are done with this debug phase and have fixed all functional issues, you will not need any further debugging at the hardware level.

It's important to remember that the functional model involves none of the hardware infrastructure elements. In the example above, the focus is on a simple, two-function model; none of the system aspects added in Figure 1 (such as the communication components, the control plane, and clocking and resets) are in play during this modeling and verification phase.

Step 3: Hardware generation. This step generates the FPGA hardware from your software model. It involves three simple actions:

1. Using a drop-down menu in the QuickPlay GUI, select the FPGA hardware into which you want to implement your design. QuickPlay can implement designs on a growing selection of off-the-shelf boards

Once you are done with the software debug phase and have fixed all functional issues, you will not need any further debugging at the hardware level.

that feature leading-edge Xilinx® All Programmable FPGAs, PCIe 3.0, 10-Gbit Ethernet, DDR3 SDRAM, QDR2+ SRAM and more.

2. Select the physical interfaces (and therefore the protocols) to map to the design input and output ports. These are also simple menu selections. The choice will depend on the interfaces that are available on the FPGA board you have selected, such as PCIe, TCP/IP over 10-Gbit Ethernet and UDP over 10-Gbit Ethernet. Selecting the communication protocol automatically invokes not only the hardware IP block required to implement the connection, but also any software stacks layered over it, so that the complete system is created.

3. Launch the build process. This will run the HLS engine (creating hardware from C code), create the needed system hardware functions (the control plane logic in our original example) and run any other tools necessary (for example, Xilinx's Vivado® integrated design environment) to build the hardware images that the board will require. No manual intervention is required to complete this process.

Step 4: System execution. This is similar to the execution of the functional model in Step 2 (functional verification), except that now, while the host application still runs in software, the FPGA design runs on the selected FPGA board. This means that you can stream real data in and out of the FPGA board and thereby benefit from additional verification coverage of your function. Because

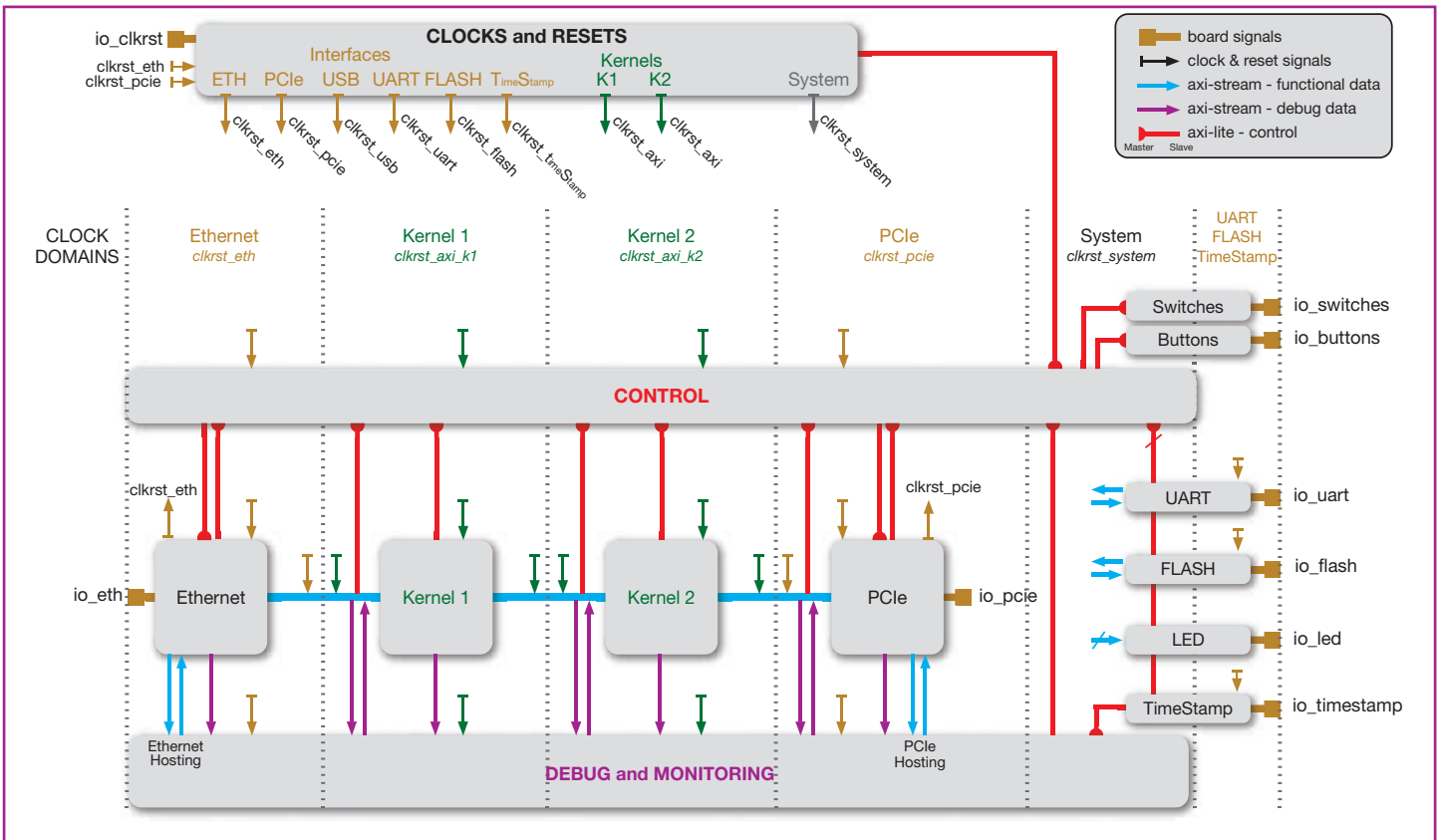


Figure 4 — The debug infrastructure is automatically created.

As a result of the abstraction that QuickPlay provides, the algorithms remain pure, focused solely on data manipulation and independent of the underlying communication details.

this will run so much faster, and because you can use live data sources, you are likely to run many more tests at this stage than you could during functional verification.

Step 5: System debug. Because you're running so many more tests now than you were doing during the functional verification phase, you're likely to uncover functional bugs that weren't uncovered in Step 2. So how do you debug now?

As already noted, you never have to debug at the hardware level, even if a bug is discovered after executing a function in hardware. Because QuickPlay guarantees functional equivalence between the software model and the hardware implementation, any bug in the hardware version has to exist in the software version as well. This is why you don't need to debug in hardware; you can debug exclusively in the software domain.

Once you have identified the test sequence that failed in hardware, QuickPlay can capture the sequence of events at the input of the design that generated the faulty operation and replay it back into the software environment, where you can now do your debug and identify the source of the bug using the Eclipse debugger.

This is possible because QuickPlay automatically provisions hardware with infrastructure for observing all of the critical points of the design. You can disable this infrastructure to free up valuable hardware real estate. Figure 4 shows the example system with added debug circuitry. Without QuickPlay, some sort of debug infrastructure would have to be inserted and managed by hand; with QuickPlay, this all becomes automatic and transparent to the software developer.

The overall process is to model in software, then build the system and test in hardware. If there are any bugs, import the failing test sequences back into the software environment, debug there, fix the source code and then repeat the process. This represents a dramatic productivity improvement over traditional flows.

Step 6 (optional): System optimization. Once you have completed the debug phase, you have a functional design that operates on the FPGA board correctly. You may want to make some performance optimizations, however, and this is the proper time to do that, as you already know that your system is running correctly.

The first optimization you should consider is to refine your functional model. There are probably additional concurrency opportunities available; for example, you might try decomposing or refactoring functions in a different way. At this level, optimizations can yield spectacular performance improvements. Needless to say, doing so with a VHDL or Verilog design would require significant time, whereas doing the modifications in C would be a quick and straightforward process.

Second, you may want to try a different FPGA board with a faster FPGA. Because the mapping from the functional model to the board is so easy, it's a simple matter to try a variety of boards in order to select the optimal one.

The third optimization has to do with the hardware kernels that QuickPlay creates via high-level synthesis. While the resulting hardware is guaranteed to operate correctly and efficiently, it may not operate as efficiently as hardware handcrafted by a hardware engineer. At this stage, you have several options: You can optimize your code and tune QuickPlay HLS settings to improve the generated hardware, use Vivado HLS to generate more-efficient hardware, or have a hardware designer handcraft the most critical blocks in HDL.

None of these optimization steps is mandatory, but they provide options when you need better-performing hardware and have limited hardware design resources available. A hardware engineer may be able to help with these optimizations. Once you have made any of these changes, simply repeat the build process.

A UNIVERSAL STREAMING CONDUIT

QuickPlay provides a universal streaming API that entirely abstracts away the underlying physical communication protocol. Streaming data is received via the ReadStream() function and is sent out using the

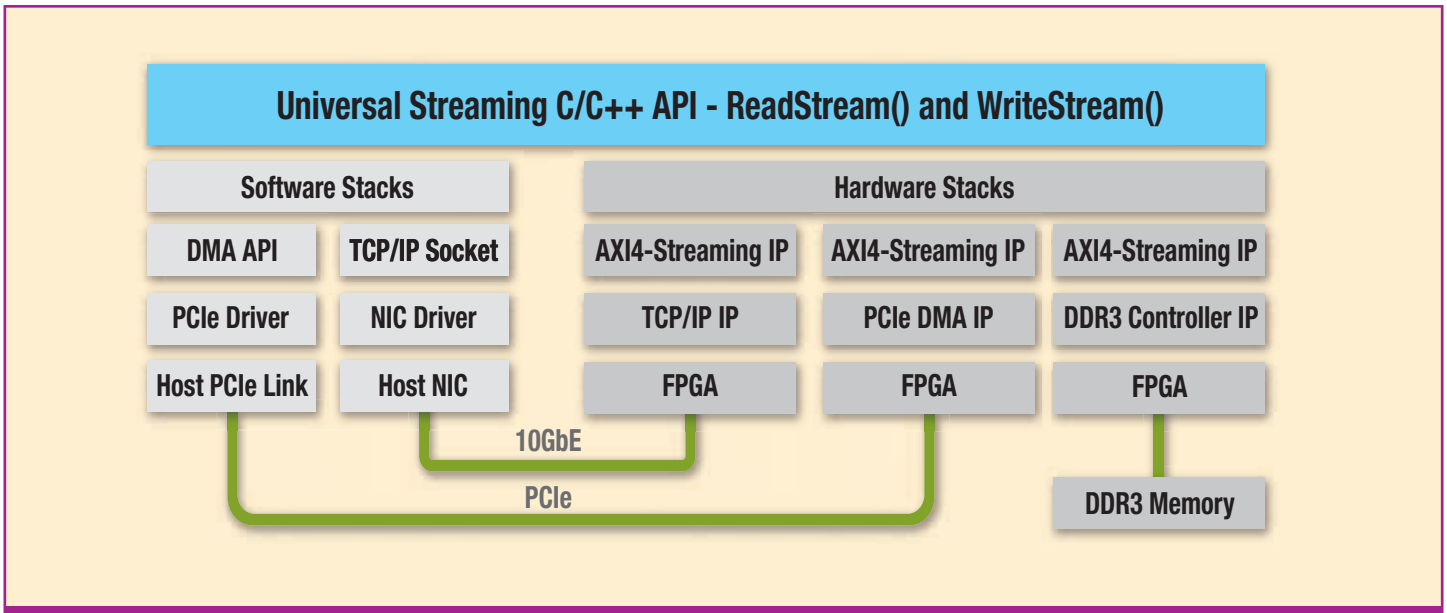


Figure 5 — Selecting the desired protocol sets up the required hardware and software stacks.

WriteStream() function. Those functions can be used to send and receive data between kernels, to embedded or board-level memory, or to an embedded or external host CPU, thus providing broad architectural flexibility with no need for the developer to comprehend or manage the underlying low-level protocols.

The selected protocol determines the hardware through which that data arrives and departs. At present, QuickPlay supports ARM® AMBA® AXI4-Stream, DDR3, PCIe (with DMA) and TCP/IP; more protocols are being added and will be added as demand dictates. Selecting the desired protocol sets up not only the hardware needed to implement the protocol, but also the software stacks required to support the higher protocol layers, as shown in Figure 5.

QuickPlay manages the exact implementation of these reads and writes (size, alignment, marshaling, etc.). The most important characteristic of the ReadStream() and WriteStream() statements is that they are blocking: When either statement is encountered, execution will not pass to the next statement until all of the expected data has been read or written. This is important for realizing the determinism of the algorithm.

The "binding" between the generic ReadStream() and WriteStream() statements and the actual underlying protocol hardware occurs at runtime via the QuickPlay Library. This not only prevents the communication details from cluttering up the software program, but also provides modularity and portability. The communication protocol can easily be changed without requiring any changes to the actual kernel code or host software. The ReadStream() and WriteStream() statements will automatically bind to whichever protocol has been selected, with no effect on program semantics.

As a result of the abstraction that QuickPlay provides, the software algorithms remain pure, focusing solely on data manipulation in a manner that's completely independent of the underlying communication details.

PRODUCTION-QUALITY OUTPUT

Depending on the HLS tool being used, results might be improved by learning coding styles that result in more efficient hardware generation, but that is optional.

While in other situations the hardware platform you use may be viewed simply as a prototyping vehicle, the systems you create using QuickPlay are production-worthy. Going from a purely software implementation to a hardware-assisted or hardware-only implementation traditionally takes months. QuickPlay reduces that time to days.

The QuickPlay methodology achieves the long-sought goal of allowing software engineers to create hardware implementations of all or portions of their application. By working in their familiar domain, software engineers can make use of custom hardware as needed, automatically generating hardware-augmented applications that operate more efficiently and can be production-ready months ahead of handcrafted designs. ■

Xtra, Xtra

Xilinx® is constantly refining its software and updating its training and resources to help software developers design innovations with the Xilinx SDx™ development environments and related FPGA and SoC hardware platforms. Here is list of additional resources and reading. Check for the newest quarterly updates in each issue.

SDSOC™ DEVELOPMENT ENVIRONMENT

The SDSoC environment provides a familiar embedded C/C++ application development experience, including an easy-to-use Eclipse IDE and a comprehensive design environment for heterogeneous Xilinx All Programmable SoC and MPSoC deployment. Complete with the industry's first C/C++ full-system optimizing compiler, SDSoC delivers system-level profiling, automated software acceleration in programmable logic, automated system connectivity generation and libraries to speed programming. It lets end-user and third-party platform developers rapidly define, integrate and verify system-level solutions and enable their end customers with a customized programming environment.

- SDSoC Backgrounder (PDF)
- SDSoC User Guide (PDF)
- SDSoC User Guide: Getting Started (PDF)
- SDSoC User Guide: Platforms and Libraries (PDF)
- SDSoC Release Notes (PDF)
- Boards, Kits and Modules
- SDSoC Video Demo
- Buy/Download

SDACCEL™ DEVELOPMENT ENVIRONMENT

The SDAccel environment for OpenCL™, C and C++ enables up to 25x better performance/watt for data center application acceleration leveraging FPGAs. A member of the SDx family, the SDAccel environment combines the industry's first architecturally optimizing compiler supporting any combination of OpenCL, C and C++ kernels, along with libraries, development boards, and the first complete CPU/GPU-like development and run-time experience for FPGAs.

- SDAccel Backgrounder
- SDAccel Development Environment: User Guide
- SDAccel Development Environment: Tutorial
- Xilinx Training: SDAccel Video Tutorials
- Boards and Kits
- SDAccel Demo

SDNET™ DEVELOPMENT ENVIRONMENT

The SDNet environment, in conjunction with Xilinx All Programmable FPGAs and SoCs, lets network engineers define line card architectures, design line cards and update them with a C-like environment. It enables the creation of "Softly" Defined Networks, a technology dislocation that goes well beyond today's Software Defined Networking (SDN) architectures.

- SDNet Backgrounder — Xilinx
- SDNet Backgrounder — The Linley Group
- SDNet Demo

SOFTWARE DEVELOPMENT KIT (SDK)

The SDK is Xilinx's development environment for creating embedded applications on any of its microprocessors for Zynq®-7000 All Programmable SoCs and the MicroBlaze™ soft processor. The SDK is the first application IDE to deliver true homogeneous- and heterogeneous-multiprocessor design and debug.

- Free SDK Evaluation and Download ■

This year's best release.

Xcell Publications

Solutions for a Progammable World

The definitive resource for software developers speeding C/C++ & OpenCL code with Xilinx SDx IDEs & devices

The Award-winning Xilinx Publication Group is rolling out a brand new trade journal specifically for the programmable FPGA software industry, focusing on users of Xilinx SDx™ development environments and high-level entry methods for programming Xilinx All Programmable devices.

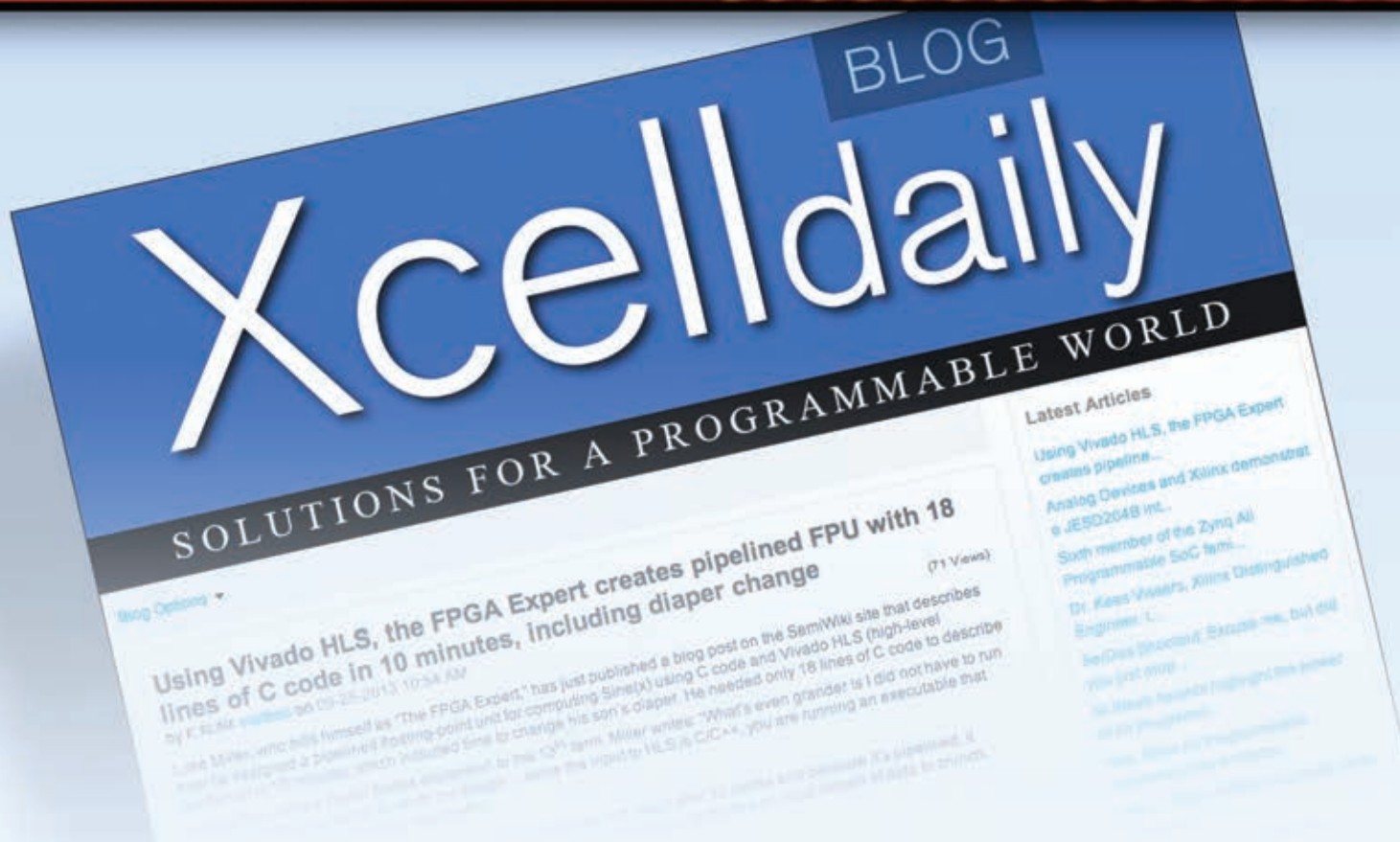
This is where you come in.

Xcell Software Journal is now accepting reservations for advertising opportunities in this new, beautifully designed and written resource. Don't miss this great opportunity to get your product or service into the minds of those who matter most. Call or write today for your free advertising packet!

For advertising inquiries (including calendar and advertising rate card), contact xcelladsales@aol.com or call: 408-842-2627.

XILINX
ALL PROGRAMMABLE™

Xcell Journal Adds New Daily Blog



Xilinx has extended the Award Winning Journal and added an exciting new *Xcell Daily Blog*. The new site provides dedicated readers with a frequent flow of content to help engineers leverage the flexibility and extensive capabilities of Xilinx products, ecosystem, and customers to create All Programmable and Smarter Systems.

Recent

- *ArrayFire uses Xilinx SDAccel for real-time video image feature detection at SC15*
- *Adam Taylor's MicroZed Chronicles, Part 108: Creating our hardware definition for SDSoC*
- *Adam Taylor starts new SDSoC design article series on Embedded.com*
- *Zynq-based Red Pitaya Open Instrumentation Platform adds simplified Visual Programming language for beginners*
- *Xilinx System Generator 2015.3: MathWorks' HDL Coder integration means you can tweak performance but keep "easy to use"*

Visit Blog: www.forums.xilinx.com/t5/Xcell-Daily/bg-p/Xcell

XILINX
ALL PROGRAMMABLE.