Accelerating SAR Processing on COTS FPGA Hardware Using C-to-Gates Design Tools

Raymond R. Hoare II
Concurrent EDA, LLC
Pittsburgh, PA
rayhoare@concurrenteda.com

Denis Smetana
Curtiss-Wright
Ashburn, VA
dsmetana@curtisswright.com

***Abstract*—Next generation radar systems require that massive computations be performed in real time and have size, weight and power restrictions. Increasingly, cost is also playing a major role. This paper introduces two major innovations that help meet these SWaP-C requirements. The first innovation is the COTS OpenVPX board from Curtiss-Wright called CHAMP-FX4 that contains three large Virtex 7 FPGAs with a total of 10,800 DSP Slices and 4,410 on-chip Block RAMs. The second innovation is Concurrent EDA's FPGA design automation tools that analyze and synthesize C/C++ into high-performance FPGA firmware. This paper describes how SAR processing was accelerated by 39 to 46 times faster than a single core of an Intel Core i7 CPU running at 3.6GHz and consumed less than half of one FPGA, 1/6th of the available FPGA area on Curtiss-Wright's CHAMP-FX4 hardware. Total design-time was 6 weeks.**

Keywords—FPGA; COTS; SAR; Synthesis; Design Automation

I. Introduction

Radar is obviously critical to national defense and is constantly improving to take advantage of faster sensors, faster processing capabilities and new algorithms. SWaP requirements are becoming more stringent as unmanned aircraft shrink in size and expand in capabilities. SWaP-C requirements add cost as a major factor in specifying and building new systems. If development and deployment costs can be reduced, more systems can be upgraded and newer algorithms can be field tested.

One way to be more cost efficient is to utilize COTS hardware. With advances in Moore's Law, numerous chips on custom circuit boards all fit within a single chip. Field Programmable Gate Arrays have evolved from simple glue logic to massive compute and communication engines. With open-standard backplanes and FPGA-based COTS boards, computationally intensive radar systems can be created.

Design cost is also a major factor in new radar systems. Unfortunately, FPGA design cost is much higher than software design cost and requires six to eighteen months of development time after the software algorithms have been frozen. One solution is to not use FPGAs but to look to multi-core CPUs and massively threaded GPUs. While this may speed development and reduce cost, it typically comes at the expense of size, weight and power.

The ideal solution to the FPGA design-cost problem is to dramatically improve FPGA design automation tools so that FPGA designs can be completed in weeks or even days. There are now multiple tools on the market that seeks to make this a reality. Unfortunately, there are no radar benchmarks that describe the performance and productivity of these tools for modern FPGAs [3][4].

This paper describes how the Lincoln Labs / DARPA HPCS SAR benchmark was converted into an FPGA design within 6 weeks. This study was motivated by the Curtiss-Wright CHAMP-FX4 FPGA board that contains 3 large Virtex-7 FPGAs that are directly connected to high-speed sensors through a mezzanine board and to other boards through numerous multi-gigabit links. Concurrent EDA's tools were used to convert the Image Formation kernel and the Detection kernel into high-performance FPGA designs that are 39 to 46 times faster than a 3.6 GHz Core i7.

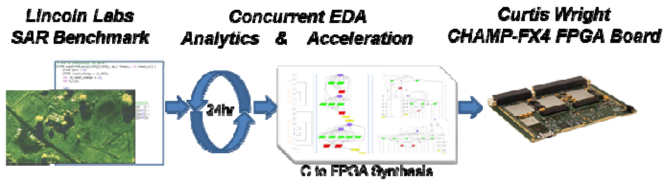


Fig. 1 Acceleration of SAR Benchmark using FPGA Design Automation Tools and COTS Hardware

II. The SAR Benchmark

The HPEC Challenge Benchmark Suite was created to provide a real-world benchmark to measure advances in processing technologies [1]. This suite includes multiple compute kernels and a Synthetic SAR benchmark. The SAR benchmark includes four kernels, Kernel 1 – Image Formation, Kernel 2 – Image Storage, Kernel 3 – Image Retrieval, and Kernel 4 – Detection. Fig. 2 is from the HPEC 2005 presentation of this Suite and shows just the computational portions of the SAR benchmark. There are multiple input sets that were synthetically generated to test the scalability of the processing being examined. As shown, the raw SAR images are transformed into SAR images by performing pulse compression, polar interpolation, FFT, and IFFT computations.

To test detection, the Template Insertion code inserts rotated letters into the SAR image. This insertion is not part of the timed benchmark as these letters represent targets that would be hidden within the input image. Kernels 2 and 3 are Image Storage and Image Retrieval and have been removed as they do not perform any computation.

Kernel 4 performs Detection by searching the entire image for the known set of targets. In this case, the targets are rotated letters. The benchmark was created to be representative of SAR processing and to be easily verified. Verification is performed by detecting the location and orientation of each letter in the image.

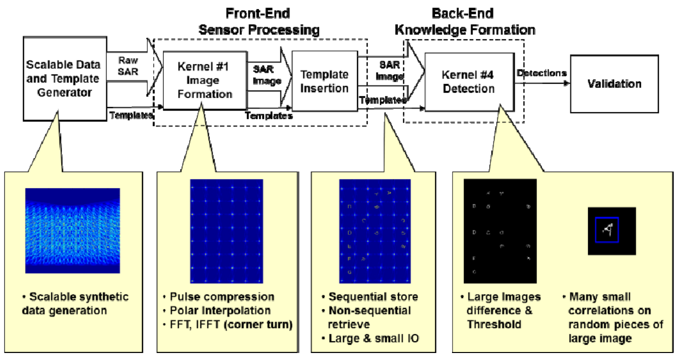

Front-End
Sensor Processing
Back-End
Knowledge Formation
Scalable Data and Template Generator
Raw SAR
Templates
Kernel #1 Image Formation
SAR Image
Templates
Template Insertion
SAR Image
Templates
Kernel #4 Detection
Detections
Validation
• Scalable synthetic data generation
• Pulse compression
• Polar Interpolation
• FFT, IFFT (corner turn)
• Sequential store
• Non-sequential retrieve
• Large & small IO
• Large Images difference & Threshold
• Many small correlations on random pieces of large image

Fig. 2 Stages of the Lincoln Labs/ DARPA HPCS SAR Benchmark [1]

III. Advances in COTS FPGA Hardware

The motivation for this work is Curtiss-Wright's 4th generation 6U FPGA card, the CHAMP-FX4 [2]. The CHAMP-FX4 is the flagship 6U product in Curtiss-Wright Defense Solutions' family of user-programmable Xilinx® Virtex®-7 FPGA-based computing products, designed to meet the needs of challenging embedded high-performance digital signal and image processing applications. The CHAMP-FX4 combines the dense processing resources of three large Xilinx Virtex-7 FPGAs with over 12 GB of memory resources, all on a rugged 6U OpenVPX™ (VITA 65) form factor module.

The three user-programmable Virtex-7 FPGAs can be either 585T or 690T FPGAs. The 690T contains more compute capabilities and is the focus on this paper. Each of the three 690T FPGAs has nearly 700,000 Logic Cells that can be used for traditional logic and moderately complex integer operations. To enable high-performance signal processing, there are also ASIC cells within the FPGA called DSP Slices that can be configured to implement various two and three input arithmetic and logic instructions. Even single and double precision floating point operations can be performed by using multiple DSP Slices together. Each of the three user-programmable FPGAs contains 3,600 DSP Slices.

Table 1 shows the total compute capability of each CHAMP-FX4 board. Like a multi-core processor, each of the DSP Slices can execute in parallel on different pieces of data. Unlike a multi-core processor or GPU, the computation and inter-DSP communication can have hard-wired data paths that interconnect the internal Block RAMs and DSP Slices together to form a massive assembly line of computations.

If the data size exceeds that of on-chip Block RAMs, then high-speed DRAM and SRAM can be used. DRAM has a higher density that performs well for streaming data on and off chip. Each CHAMP-FX4 board can have up to 12GB on on-board DDR3 DRAM. For data that requires random-access patterns, QDRII+ SRAM can be used. Unlike DRAM, the latency to any location in the SRAM is the only a few clock cycles. For this reason, processor caches use SRAM. Each CHAMP-FX4 board contains up to 108 MB of QDRII+ SRAM.

In addition to memory and processing capacity, the CHAMP-FX4 card has vast amounts of backplane I/O bandwidth directly connected to the FPGAs, easily exceeding 100 GB/s.

Table I Compute Capability of the Rugged CHAMP-FX4 OpenVPX Card

	Virtex 7 X690T	CHAMP-FX4 Total
Logic Cells	693,120	2,079,360
DSP Slices	3,600	10,800
Block RAMs/FIFOs	1,470	4,410
DDR3 SDRAM	2 GB x 2 banks	12 GB (6 banks total)
QDRII+ SRAM	18 MB x 2 banks	108 MB (6 banks total)

IV. FPGA Design Automation

FPGAs follows Moore's Law and typically have twice the compute capability as the prior generation. Budgets, on the other hand, don't double but tend to shrink. Additionally, time-to-market pressures are reducing design-time. How can engineering teams perform twice the work is less time?

One method to accelerating design times is to break designs down into components and reuse the components that don't change. This does, however, assume that only a portion of the design is changing and that there is a well-documented, well tested repository of components available. This method is worthwhile and can be productive but it also increases the work for each new component.

Third-party components called Intellectual Property (IP) Cores are another way to reduce design complexity. Xilinx has a large library of IP Cores that can simplify the design of communications and low-level computations. Concurrent EDA and other vendors also offer IP Cores that can be utilized within a design to perform standard computational tasks such as FFTs, matrix operations, filtering, compression and various other common functions. Even if 25% of a design is comprised of reusable components or IP Cores, then engineering teams are still faced with 1.5 times the design work when chips capacity doubles.

Without large design teams, automation tools are required for high density FPGA designs. VHDL and Verilog are hardware description languages that are typically used for FPGA design. These languages require that the designer specify the exact operations that are to be performed within each clock tick. For high-performance designs, the engineer must manually determine the amount of logic that can be performed within a fixed number of nanoseconds. If one portion of the design has too much logic, then a slower system clock must be used and the entire design performance is reduced. It is therefore critical to have highly trained engineers who are experts at VHDL and Verilog. Highly trained experts at VHDL or Verilog are expensive and rare and even they are not able to improve their productivity with VHDL/Verilog by 1.5 times every 18 months. New languages and tools are thus required.

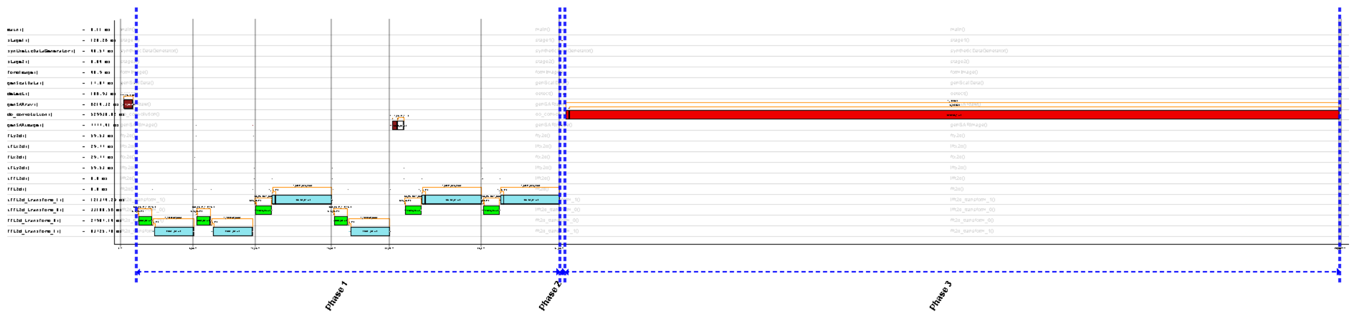


Fig. 3 Concurrent Analytics Execution Graph of the SAR Benchmark.

High-level synthesis tools enable designs to convert higher-level languages such as C into FPGA gates. Granted, C is not typically considered a high-level language to a software engineer, but when compared to VHDL and Verilog it fits the definition nicely. Xilinx has recently released a tool called HLS that enables well-structured C code to be converted into FPGA gates by inserting pragma statements into the code to drive the synthesis tools [3]. Altera has an OpenCL high-level synthesis tool but it requires that the algorithms be rewritten in highly parallel OpenCL [4] and like a GPU, it makes heavy use of external memory. Both tools, however, have their own underlying structure that is not human readable or testable in the same way as VHDL and Verilog designs.

Concurrent EDA has two tools that are used to convert algorithms into high-performance FPGA designs. The first tool, Concurrent Analytics, is used at the start of a design to: (1) determine where an algorithm spends most of its time, (2) estimate the amount of FPGA area require for each part of the algorithm and (3) identify areas of the algorithm that need to be optimized, such as system calls.

Fig. 3 shows the Execution Graph that Concurrent Analytics generated after analyzing the SAR Benchmark. Fig. 5 shows the same graph zoomed in on the Image Formation Kernel and Fig. 7 is zoomed in on the Target Detection Kernel. The Execution Graph lists all of the software functions on the left and shows the execution of each function and loop from the start of the program on the far left to the end of the program on the far right. The vertical lines were added after execution and are used to measure particular sections, or phases of execution.

Fig. 3 shows that Phase 1 (Kernel 1 Image Formation) and Phase 3 (Kernel 4 Detection) consume the vast majority of the execution time. In this "relative view" of the code, each of the rectangles represent a small portion of the code with the width being the relative time consumed by each section. By moving the phase boundaries around, each phase will contain different loops and software functions. This enables the design to be partitioned into multiple FPGA blocks that can be executed in parallel as meta-pipeline stages. Note that we excluded a function in Phase 1 because this is part of the data generation portion of the benchmark that is replaced by physical sensors in a real system. Additionally, it can be seen that Phase 2 consumes a very small amount of time.

Concurrent EDA's second tool is Concurrent Acceleration. This tool converts C code into high-performance FPGA firmware. At this time of this article, over 1 million lines of production VHDL have been generated using Concurrent Acceleration. This conversion is performed at the software function-level so that the resulting FPGA firmware can be called from software and integrated into a combined hardware/software system, or used in a hardware-only system.

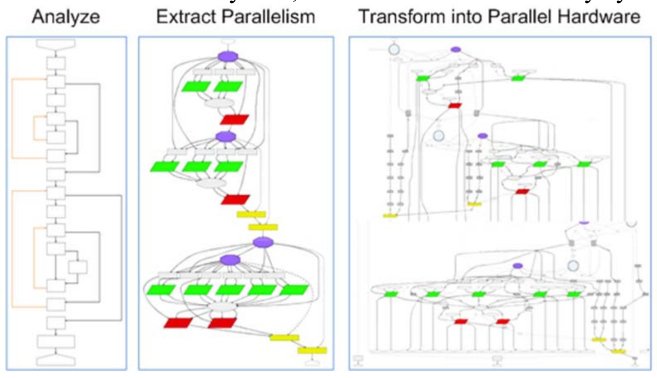


Fig. 4 Concurrent EDA's Acceleration C-to-Gates Synthesis Tool

As seen in Fig. 4, the software function is analyzed and loops are identified. These loops are illustrated as orange back arcs in the diagram and in this example, the function contains two doubly-nested loops. Each of these loops is then analyzed and loop-level parallelism is extracted. The second graph in this figure is a hardware-level graph that separates loop control (purple-loop start and yellow-loop end), memory reads (green), compute (clear) and memory writes (red). With loop-level parallelism, each block on this diagram can be executed in parallel. Data flow control is used to ensure sequential consistency between the loops. The last step is to pipeline the compute blocks and the loop. The result is a highly pipelined, highly parallel FPGA implementation of the algorithm that the software user specified. The output is human readable VHDL that is traceable back to the graphs and to the source code.

As part of Concurrent Acceleration, test scripts are generated that reads data from data files that were generated by the software algorithm. Detailed simulations are performed using standard FPGA simulation tools, such as ModelSim and Aldec. The output of the circuit is also written to files and can be compared against the data that the software algorithm produces. The outputs are bit-for-bit accurate with software and can be compared using the standard diff command. Code coverage and other verification tools can be run on the generated VHDL just as if it was hand written. This enables DO-254 verification.

V. SAR Benchmark Execution and Objectives

The objective of this case study was to demonstrate that FPGAs can be rapidly designed using Concurrent EDA's design tools and that these designs can significantly outperform high-end processors. The entire process took 6 weeks and resulted in two FPGA designs that were taken through place-and-route and were fully simulated to verify correctness. Integration on the CHAMP-FX4 was not undertaken at the time of this writing but external memory interfaces were generated to match the width of the QDR memories and DMA routines were included in the coding and timing of the benchmarks.

The SAR Benchmark was widely used for the DARPA HPCS program. George Washington University has a nicely packaged version available in C and this code was used as the starting point [5]. This code was also used as the reference implementation for performance comparison. The benchmark has three image sizes that can be used 382x266, 762x512 and 1144x756. The largest image size, 1144x756 was selected and used. The Makefile has fftw turned off, SSCA_SCALE=3 (the largest image), and SSCA_DOUBLE turned off (single precision floating point is used). Both single and double precision floating point operations are supported by Concurrent EDA's tools and future work will include regenerating the FPGA cores with double precision turned on. To enable function and variable names to be included in the binary, we turned on Dwarf2 information by adding the flags "-g3 -gdwarf-2" to the gcc compiler parameters. The Dwarf2 flag only adds information to the execution binary and does not alter the machine code that is executed. As such, it does not impact execution performance. The default parameter file param_file.txt was utilized.

Concurrent Analytics was used to perform detailed timing analysis for the Execution Graphs but only the clock_gettime() function was used to measure software execution performance. This ensures that Concurrent Analytics did not impact software performance as it wasn't used during those execution runs.

VI. FPGA-Acceleration of SAR Image Formation

Kernel 1 in the SAR Benchmark is Image Formation. Fig. 5 shows Phase I of the Execution Graph that was generated by Concurrent Analytics on the SAR Benchmark. This phase consumed 32.5% of the total execution time and shows the functions that were executed in this phase. Note that the vast majority of the time is spend in computing the 2D iFFT and FFT. The Execution Graph is a "relative view" mode and function calls that do not consume an appreciable portion of the execution time are shown as dots and are hard to see in this figure. All of the functions below genSARimage() were executed but many of them are wrapper functions that call the ifft2d_transform_0/1() and fft2d_transform_0/1() functions. It can clearly be seen that these ifft2d/fft2d functions dominate the execution time of Kernel 1. To be precise, 99.6% of Kernel 1 time is spent in these functions. As such, these functions were implemented in FPGA logic and the calling functions were not. In a production system, both would be required.

A number of optimizations were performed on the ifft2d/fft2d functions to speed-up its FPGA execution. The calculation of the sine and cosine tables were moved out of the function and hard-coded. Floating point math (single and double precision) is fully supported in FPGAs by using library components that utilize multiple DSP Slices. This works quite well and Concurrent EDA's tools automatically map these operations to highly optimized floating point libraries.

Floating point operations require multiple clock cycles for FPGAs (and for CPUs) and when they are used to accumulate sums it becomes a performance bottleneck. For a pipelined FPGA design this is a major hurdle as it is analogous to a slow spot in an assembly line. This is not a problem for non-accumulators as multiple stages can be used to perform the computation without any penalty. In the iFFT/FFT functions, there are two floating point accumulators.

In contrast, large fixed point accumulators perform well and only require a single clock cycle. To determine the best fixed point format, a detailed analysis of the floating point exponents was performed using Concurrent EDA's Floating Point Analysis libraries. In the iFFT/FFT functions, only the two floating point accumulators were converted to fixed-point and float-to-fixed and fixed-to-float conversion routines were used to integrate with the rest of the floating point computations. These modifications were all performed in

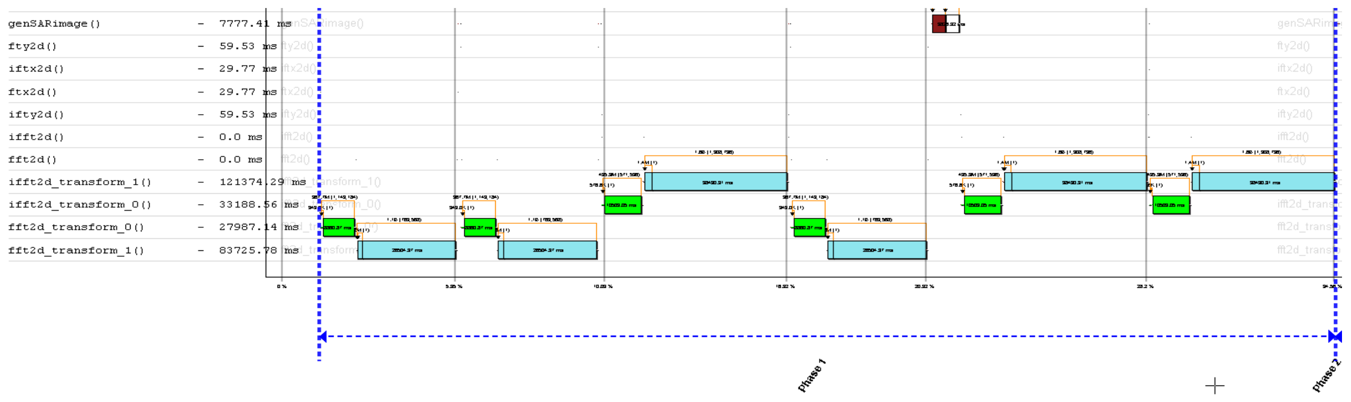


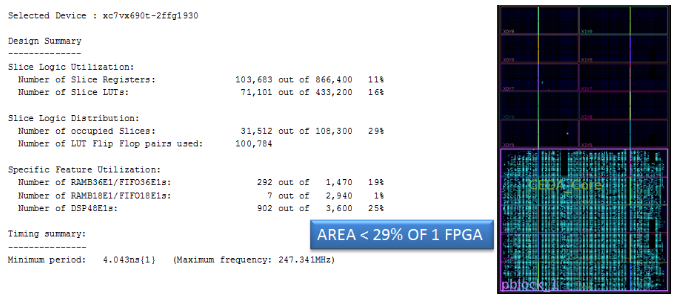
Fig. 5 Concurrent Analytics Execution Graph of the Kernel 1 Image Formation in the SAR Benchmark

software and verified against the original code. A small number of values had a 1-bit error in the least significant position of the floating point results. This optimization, however, did not impact the overall verification of the benchmark and was used to generate the FPGA design.

Concurrent Acceleration is a synthesis tool that converts C/C++ into high-performance FPGA firmware. This tool first uses gcc to convert C/C++ code into an x86 binary and then converts the x86 machine instructions into complex control and data flow graphs. These graphs represent the exact computation that is performed by a processor. Loops are turned into pipelined control logic and operations are turned into highly optimized compute pipelines. Each x86 operation is mapped directly into VHDL or mapped to a VDHL library component for more complex multi-cycle operations such as floating point. The result is a highly parallel pipelined design optimized for the particular FPGA being targeted.

To accelerate Image Formation, all four of the ifft2d/fft2d functions were merged into a single software function that is called twelve different times for each Kernel 1 execution. Two parameters are passed in to tell the function which variation should be performed. This did make the code larger but it also enabled a single hardware block to be created that implements all of the necessary ifft2d/fft2d compute functions.

Since FPGAs perform very well when pipelined, it is beneficial to have a large loop body that can be turned into a long pipeline of computations. In the original code, however, there are only a few operations and while this would result in a small FPGA core, it would limit performance. In software, we unrolled the inner loops 16 times to increase the pipeline size.



```
Selected Device : xc7vx690t-2ffg1930

Design Summary
--------------
Slice Logic Utilization:
  Number of Slice Registers:            103,683 out of 866,400   11%
  Number of Slice LUTs:                  71,101 out of 433,200   16%

Slice Logic Distribution:
  Number of occupied Slices:             31,512 out of 108,300   29%
  Number of LUT Flip Flop pairs used:   100,784

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:             292 out of   1,470   19%
  Number of RAMB18E1/FIFO18E1s:               7 out of   2,940    1%
  Number of DSP48E1s:                       902 out of   3,600   25%

Timing summary:
---------------
Minimum period:   4.043ns{1}   (Maximum frequency: 247.341MHz)
```

Fig. 6 FPGA Implementation Results for the Image Formation Kernel

As a result, the amount of memory bandwidth required drastically increased. Fortunately, Concurrent EDA's compiler recognized that these were sequential accesses and created a 1024-bit wide memory so that all 16 variables are read in a single cycle. In total, 292 out of 1,470 internal RAMs (19%) were used in parallel to implement the ifft2d/fft2d functions.

The Concurrent Acceleration compiler generated VHDL from the new design. Using data generated from the software as input, the generated VHDL was simulated and the output data was compared with the output that the software generated. The results were bit-for-bit identical. The VHDL was then synthesized and implemented into a placed-gates design for a Virtex 7 X690T FPGA. The area Design Summary and an image of the FPGA floor plan is shown in Fig. 6. Note that no more than 25% of the each of the chip resources was used. An area constraint was used to limit the design to half the chip and this helped achieve a faster fMax. The overall clock rate of this design was 247MHz.

To test relative performance of the FPGA design, a clock-level simulation was used for three different calls to this function. The simulation was executed and the number of clock cycles was multiplied by the clock period to get execution time. Note that the clock rate used is after place-and-route and is thus accurate. This is a deterministic method of measuring performance that correlates exactly with actual hardware using the same clock input. The original optimized but not unrolled software was used as the software performance standard. Accurate software timers were inserted into the ifft2D/fft2D code to measure the time of each call and an Intel Core i7-3820 running at 3.6GHz was used to execute the program. Software malloc()/free() times were excluded from the software execution timers as these should have been statically declared to save time.

Table II shows the performance of six different calls to the ifft2D/fft2d function with different data sizes. The end result is that each call is about 39.3 to 40 times faster in the FPGA than in software.

It should be noted that only 1 CPU core and 1 FPGA core are being used. 3 to 4 FPGA Cores could fit in each of the three FPGAs if only SAR Image Formation was being performed. Thus, even higher performance with an FPGA can be achieved by using multiple cores.

Table II Performance Results of ifft2D/fft2D in Image Formation

Data Size	1 FPGA Core @ 247 MHz	1 Core i7 CPU @ 3.6 GHz	Speedup
480 x 1072	0.14 s	5.67 s	40.0x
1072 x 480	0.06 s	2.54 s	39.3x
1072 x 1144	0.36 s	14.30 s	39.7x
756 x 1144	0.25 s	10.09 s	39.7x
1144 x 756	0.17 s	6.71 s	39.7x

VII. FPGA-Acceleration of SAR Target Detection

Kernel 4 of the SAR Benchmark is Target Detection. In this benchmark, rotated letters were added to the image after Kernel 1 was completed. As shown in Fig. 7, 100% of the time in this phase is spent in *do_convolution()* which is called 10,773 times by *detect()*. The Execution Graph shows the total number of times each loop is executed and in parenthesis, it shows the number of iterations of the loop each time it is called. The edges between each rectangle show the number of times it was traversed and the time inside the rectangle shows the time spent in each section of code.

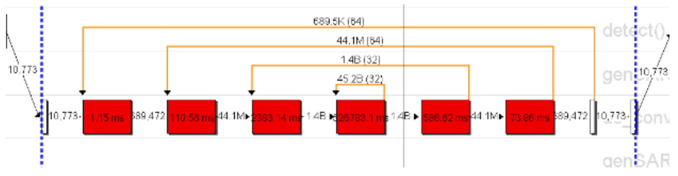


Fig. 7 Concurrent Analytics Execution Graph of Kernel 4 Target Detection

The *do_convolution()* source code is shown in Fig. 8. For each call, a subset of the SAR Image is selected and a particular 32x32 dimensioned template is used. Each of the outer loops specifies a starting location in the X and Y dimension for a 32x32 convolution of the sub image with the template. The multipliers in the inner loop are floating point and are cast to a *long* integer before it is accumulated. Since the accumulator is an integer, the accumulator can be executed in a single cycle and does not present a restriction on converting the code into a parallel pipeline.

Like Kernel 1, the loop body in Kernel 4 is small. A very small FPGA design could be created directly from this code but it would have limited parallelism. We unrolled the entire inner loop to make a larger loop body and thus created a compute pipeline that is 32 times larger than the unmodified code. This would appear to be an issue because it requires that we concurrently read 32 values from the sub image and that we also concurrently read 32 values from the template. Fortunately, we have hundreds of FPGA Block RAMs even in the smaller Virtex 7 FPGA.

```
tData do_convolution(tData *template, tData **subimage){
    tData maxval = 0;
    tData *sub = subimage[0];

    for( int convx=0; convx<SSCA_FONT_SIZE*2; convx++ ){
        for( int convy=0; convy<SSCA_FONT_SIZE*2; convy++ ){
            tData val = 0;
            for( int k=-SSCA_FONT_SIZE / 2; k< SSCA_FONT_SIZE / 2; k++ )
                for( int l=-SSCA_FONT_SIZE / 2; l<SSCA_FONT_SIZE/2; l++ )
                    val += sub[ (convx+k) * SSCA_FONT_SIZE*4 + convy+l]
                           * template[(k+SSCA_FONT_SIZE/2) * SSCA_FONT_SIZE
                           + (l+SSCA_FONT_SIZE/2)];

            if( maxval < val )
                maxval = val;
        }
    }

    return maxval;
}
```

Fig. 8 do_convolution() Source Code in the Target Detection Kernel

The FPGA Design Summary results are shown in Fig. 9. Even though the inner loop was unrolled 32 times, the FPGA area is well under 10% of the chip area. Additionally, the clock frequency achieved is 311MHz. The place-and-route results are also shown in Fig. 9 and clearly shows room on the chip and a sparsely packed layout.

```
Selected Device : 7vx690tffg1930-2

Design Summary
--------------
Slice Logic Utilization:
  Number of Slice Registers:              45,287 out of 866,400    5%
  Number of Slice LUTs:                   31,672 out of 433,200    7%

Slice Logic Distribution:
  Number of occupied Slices:              13,308 out of 108,300   12%
  Number of LUT Flip Flop pairs used:     45,356

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:               64 out of   1,470    4%
  Number of RAMB18E1/FIFO18E1s:                7 out of   2,940    1%
  Number of DSP48E1s:                        192 out of   3,600    5%

Timing summary:
---------------
  Minimum period:   3.207ns{1}   (Maximum frequency: 311.818MHz)
```

AREA < 10% OF 1 FPGA

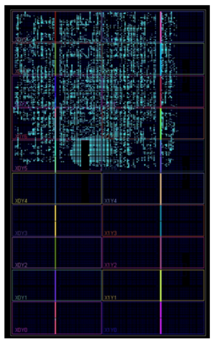


Fig. 9 FPGA Implementation Results for the Target Detection Kernel

TABLE III PERFORMANCE RESULTS OF TARGET DETECTION

	HW (s)	SW (s)	Speedup
do_convolution()	0.43	20.086	46.7 x

VIII. CONCLUSION

The objective of this paper was to demonstrate the benefit of COTS FPGA hardware and FPGA design automation tools. Specifically, we examined the hardware resources of a the three Virtex 7 FPGAs that are on the Curtiss-Write CHAMP-FX4 OpenVPX board and found they each contain 3,600 DSP Slices for a total of 10,800 DSP Slices per board. This fits most SWaP-C requirements for SAR processing and contains a massive amount of compute horsepower.

Next we examined both the productivity and the performance of FPGA designs when using Concurrent EDA's FPGA design automation tools. Concurrent Analytics was used to rapidly quantify where the SAR Benchmark spends its time during execution. Concurrent Acceleration was used to synthesize the SAR C code into VHDL and into FPGA gates.

Kernel 1, Image Formation was converted into an FPGA design using Concurrent Acceleration. It was found that the different 2D iFFT/FFT function calls consumed 99.5% Kernel 1. These functions were collapsed into a single parameterized function. A floating point accumulator was found, analyzed and replaced with a single-cycle fixed-point accumulator. To extract more parallelism, the inner loop was unrolled 16 times in software. Concurrent Acceleration was then used to convert the software function into an FPGA design. The area was less than 29% of one FPGA and achieved 247MHz. When compared to a Core i7 CPU at 3.6GHz, the FPGA version was 39 to 40 times faster.

Kernel 4, Target Detection was also converted into an FPGA design using Concurrent Acceleration. This kernel spent all of its time performing millions of 32x32 convolutions with template images. The inner loop was completely unrolled (32 times) and 64 Block RAMs were used to ensure all data values could be read in a single cycle. The resulting FPGA design consumed less than 10% of a single Virtex 7 X690T and outperformed single-threaded software running on a 3.6 GHz Core i7 by 46.7 times.

This entire case study took 6 weeks from initial software download through simulation and place-and-route and demonstrates that FPGAs can be designed quickly with Concurrent EDA's tools straight from C software.

[1] P. Luszczek , J. J. Dongarra , D. Koester , R. Rabenseifner , B. Lucas , J. Kepner , J. McCalpin , D. Bailey and D. Takahashi *Introduction to the HPC challenge benchmark suite*, 2005

[2] "The Curtisss-Wright CHAMP-FX4 6U OpenVPX Virtex-7" http://www.cwcdefense.com/products/dsp-fpga/6u-vpx-vxs/champ-fx4.html

[3] Xilinx, *Vivado High-Level Synthesis*, [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/

[4] Altera, *Achieve Power-Efficient Acceleration with OpenCL on Altera FPGAs*, [Online]. Available: http://www.altera.com/products/software/opencl/opencl-index.html

[5] GWU, *UPC Generated C Code from SSCA Benchmark* [Online]. Available: http://threads.hpcl.gwu.edu/sites/ssca3