

A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing

Stephen M. Scalera
Sanders, A Lockheed Martin Company
PTP2-B007
65 River Road
Hudson, NH 03051
stephen.m.scalera@lmco.com
Phone: (603) 885-0679 FAX: (603) 885-7623

Dr. John J. Murray & Steve Lease
Signal Systems Corporation
294 Tolstoy Lane
PO Box 787
Severna Park, MD 21146-0787
ssc@aol.com
Phone: (410) 431-7148 FAX: (410) 431-8884

Abstract

Dynamic reconfiguration of field programmable gate arrays (FPGAs) has recently emerged as the next step in reconfigurable computing. Sanders, A Lockheed Martin Company, is developing the enabling technology to exploit dynamic reconfiguration. Sanders, working with Signal Systems Corporation, identified key elements to the successful utilization of context switching. Two applications, time-domain beamforming and optical flow, are described in an attempt to reveal some of the inherent computational enhancements afforded by context switching. Conclusions are drawn as a result of this work and future work is described.

1 Introduction

Previous research has examined performance enhancement available from run-time reconfiguration. Current reconfiguration time, milliseconds, although acceptable for some applications, such as the Sanders *SPEAKeasy* reconfigurable "softradio", is unacceptable for many real-time systems. Partial reconfiguration reduces reconfiguration time additionally. Complete FPGA reconfiguration at a rate that far exceeds the necessary persistence of a hardware function is believed to be tomorrow's reconfigurable computing model. Dynamic FPGA reconfiguration, which permits data sharing between configurations, is referred to as *context switching reconfigurable computing* (CSRC).

A context switching FPGA is currently being developed at Sanders. Several computational models are suggested to exploit the features of a CSRC FPGA, however, it is believed that the true potential of CSRC requires a shift in approaches to algorithm implementation. The capabilities of the CSRC architecture have the potential to support improved implementations of signal processing algorithms over current generation mathematical approaches. The emphasis of this paper is to reveal the results of a mathematical benefit analysis of context switching on a CSRC FPGA. In this study, candidate algorithms are assessed to reveal potential benefits afforded by a CSRC implementation. Two of these candidate algorithms, time domain beamforming and optical flow, are analyzed in detail to size the applications to CSRC components and to evaluate the benefits of CSRC technology over current reconfigurable computing (FPGA) devices. Conclusions are advanced in section 4.

2 Time Domain Beamforming

At the core of sonar and radar signal processing is the notion of beamforming. Incoming waves (acoustic or electromagnetic) are received by an array of sensors and the sensed signals are digitized and processed to

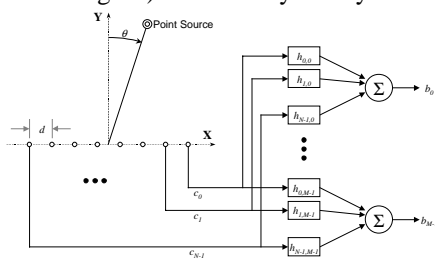


Figure 2.1: Time Domain Beamforming

As shown in the diagram, d is the distance between sensors, θ is the angle of arrival of the incoming wave, and c is the speed of propagation of the wave. This establishes a fundamental functional relationship between the wave's angle of arrival and the time of arrival of the wave at each channel sensor. Time domain beamforming passes the channel signals through delay filters, which approximate time delays, to temporally align them so that a

determine the direction of the source of the incoming waves. Figure 2.1 shows a line array of uniformly spaced sensors in the presence of a point source. The point source is taken to be distant enough so that circular effects are negligible. Thus, when the wavefront arrives at the line array it is planar, and the angle of arrival θ , is measured with respect to the normal of the line array. Under these conditions, the signals on each channel of the line array are related to one another by simple time delays: $c_i(t) = c_0(t - \tau_i)$, where $c(t)$ is the continuous time signal sensed on channel i , and $\tau_i = i \frac{d}{c}$.

coherent summation of the delayed channels amplifies signals associated with waves arriving from a particular angle. A coherent summation of N delayed channels that amplifies the signals from the point source in our

example of Figure 2.1 is:
$$\sum_{i=0}^{N-1} c_i(t - \tau_{N-i-1})$$

By changing θ , and consequently the delays τ we can change the direction in which the above summation looks. Moreover, as illustrated in Figure 2.1, the channel data can be processed in parallel to produce M beams, each with a different look angle. In particular, the j^{th} beam which has look angle θ_j is the coherent summation:

$$b_j(t) = \sum_{i=0}^{N-1} c_i(t - \tau_{N-i-1,j})$$
, where the time delays are given by: $\tau_{i,j} = i \frac{d}{c} \sin \theta_j$. In Figure 2.1, the time delays are approximated by realizable filters h_{ij} , each approximating a delay of $\tau_{N-i-1,j}$. A digital time domain beamforming algorithm using finite impulse response (FIR) filters $h_{ij}(n)$ is expressed as:
$$b_j(n) = \sum_{i=0}^{N-1} \sum_{l=0}^{L-1} h_{i,j}(l) c_i(n-l)$$
, where n is the discrete time sampling index, T denotes the sampling period, and L is the length of the FIR filters, which must be large enough to account for the longest time delay needed: $L > \max_{i,j} \left(\frac{\tau_{i,j}}{T} \right)$

In practice, the time delay FIR filters will only have K non-zero coefficients which interpolate the appropriate data samples to approximate delays which are not integer multiples of the sampling period. Thus, the evaluation of each FIR filter requires only K multiplies (rather than L). Additionally, the beamforming FIR filters also exhibit channel symmetry: $h_{i,j}(n) = h_{N-i-1,j}(L-n-1)$ which can be exploited to halve the number of multiplies

as long as the data from both channels i and $N-I-1$ are available:
$$b_j(n) = \sum_{i=0}^{\frac{N-1}{2}} \sum_{l=0}^{\frac{L-1}{2}} h_{i,j}(l) (c_i(n-l) + c_{N-i-1}(n+L-1-l))$$
, for even N . Finally, we note that the above symmetry is usually maintained even when the beamforming FIR filter coefficients are shaded with, for example, Taylor windows.

The time domain beamforming algorithm is thus naturally parallel in that the computations for each beam may be done simultaneously. It can be further segmented by evaluating the beamforming equation over subsets of channel indices to generate partial beams, which are then summed to complete the beams. The partial beam

for channel i and beam j is:
$$p_{i,j}(n) = \sum_{l=0}^{L-1} h_{i,j}(l) c_i(n-l)$$
. Subbeams are formed when partial beams are summed

over a subset I of the available channels:
$$s_{I,j}(n) = \sum_{i \in I} p_{i,j}(n)$$
. The fact that the beamforming expression lends itself naturally to parallel implementations allows for a simple mapping onto the CSRC hardware which exploits symmetry and identically zero FIR filter coefficients for computational advantage.

Each CSRC FPGA stores N_f channel time histories of length L in configurable logic blocks (LCs) which are configured as RAM accessible from all context layers. Thus, F CSRC FPGAs can store the time histories for $N=N_f F$ channels. Based on the number of remaining LCs, each of the C context layers can process M_c subbeams. Thus, a single CSRC FPGA can compute $M=M_c C$ subbeams associated with its N_f channel inputs. Figure 2.2 illustrates this decomposition for a single CSRC FPGA, where the input channel vector $C_{I(i_f)}(n)$ is the set of N_f channel histories $c(n)$ for which $i \in I(i_f)$. The FPGA index i_f thus determines which group of channel histories is presented to each of the CSRC FPGAs, $i_f=0,1,\dots,F-1$. In order to exploit the channel symmetry discussed earlier for computational advantage, the number of channel histories stored on each CSRC FPGA, N_f , must be even. Furthermore, channels i and $N-I-1$ must both be contained in one of the groups $I(i_f)$.

A system consisting of F CSRC FPGAs, each receiving N_f channels, can thus process $N=N_f F$ channels into $F M_c C$ subbeams, which are then summed to produce $M_c C$ beams. If a greater number of beams is required, this architecture can simply be replicated (any number of times), with each such system capable of producing $M_c C$ independent beams. Operationally, each CSRC FPGA is presented with the current sample for each of its input channels simultaneously. These inputs are pushed into the time history buffers (tap delay lines) in the first context layer, which also computes the first set of subbeams. The FPGA then sequences through the

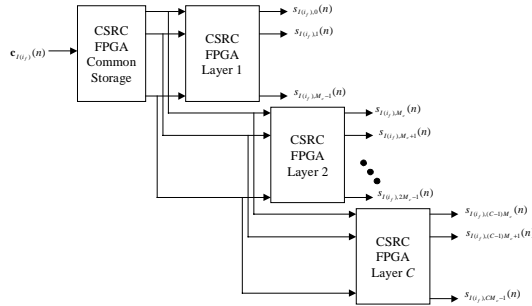


Figure 2.2: Subbeam decomposition onto CSRC FPGA

remaining context layers to produce the rest of the subbeams. Each context layer sends its output to the same set of I/O pins, so that the subbeams are multiplexed on these lines.

A slight variation to the architecture described above is to push new samples into the channel history buffers in all context layers (not just the first context layer). Each beam would thus be computed every C samples, (but the samples are now arriving C times faster). The advantage of this architecture is that the control lines which specify the active context layer, and hence the output beams, can be commanded in any order. This provides a

Number of Channels	2	2
Number of Subbeams per Context layer	3	2
Length of Channel Histories	256	256
Length of Beamforming FIRs	4	4
Channel and FIR Coefficient Data Width (bits)	8	8
Subbeam Data Width (bits)	16	16
CLBs per RAM bit	0.0625	0.0625
CLBs per Channel Data Width Multiply	40	40
CLBs per One Bit Addition	1	1
CLBs allocated for RAM	256	256
CLBs allocated for Multiplies	480	320
CLBs allocated for Additions	288	192
Total CLBs Required	1024	768

Table 2.1: CSRC Context Layer Sizing For Beamforming

hardware device that fits nicely into typical sonar and radar sensing operations which command a search pattern until a target is detected, and then track the target by selecting beams immediately around it.

Table 2.1 performs sizing calculations for a single context layer of a CSRC FPGA for the beamforming application. Using two input channels and two subbeams per context layer keeps the implementation below 80 percent utilization since each context layer has 1024 logic cells (LCs) in the current Sanders design. The calculations in the table include taking advantage of the beamformer

channel symmetry and the savings that accrue from configuring multiplies where one operand is known a priori and is hard wired in the context layer.

3 Optical Flow

The goal of the plume detection DARPA challenge problem is to segment a plume from the background in gray scale video frames. The proposed approach is to assume that the plume is the only moving object in the frames. Using an optical flow algorithm to estimate velocities within the frame, the plume can be segmented by thresholding the estimated velocities. The plume detection problem seeks architectures which can process 256 by 256 images with 12-bit pixels at a frame rate of approximately 1 kHz.

3.1 Optical Flow Algorithm Description

Using the assumption of constant illumination, the optical flow algorithm considers any changes in pixel intensity in time to be caused by object motion. The intensity change due to object motion can be expressed as: $I_t = -u(x, y, t)I_x - v(x, y, t)I_y$, [3-1] where I is the image intensity, u and v are the x and y velocities of objects within the image, $I_t = \partial I(x, y, t)/\partial t$, $I_x = \partial I(x, y, t)/\partial x$, and $I_y = \partial I(x, y, t)/\partial y$. Equation 3-1 does not account for second order effects at moving object boundaries. Employing estimates \hat{u} and \hat{v} for the unknown object velocities, a measure of the accuracy of the estimates is $e(x, y, t) = I_t + \hat{u}(x, y, t-1)I_x + \hat{v}(x, y, t-1)I_y$, where e is the residual of the estimated change in intensity with respect to time. If the image is divided into subimages, the total error can be expressed as $E = \sum_{\text{subimage}} e(x, y, t)$.

The subimage error, E , can then be used to update the velocity estimates using the relations $\hat{u}(x, y, t) = \hat{u}(x, y, t-1) - bEI_x$ and $\hat{v}(x, y, t) = \hat{v}(x, y, t-1) - bEI_y$, where b is a learning parameter. The choice of b and subimage size affect the performance and stability of the optical flow algorithm.

3.2 FPGA Architecture

The ability of the CSRC FPGAs to retain memory during context switching makes them well suited for computing the partial derivatives of the optical flow algorithm. By loading data from successive frames, a single FPGA can compute I_t , I_x , and I_y using one layer for each partial derivative. The partial derivative data can then be passed to a second FPGA to perform the error calculation and velocity estimate updates. Figure 3.1 shows a block diagram of a processing block containing two CSRC FPGAs with a shared 128 Kbyte memory. Both FPGAs have access to the RAM and the two FPGAs have data and control connections between themselves. The 64-bit data connections allow four words to be processed in parallel. A mechanism for arbitration of the RAM bus between the FPGAs and external access would be required. Use of dual-port RAM could simplify the arbitration and improve performance.

The sizing of the plume detection problem discussed in later sections indicates that eight of the processing blocks shown in Figure

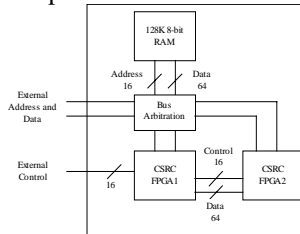


Figure 3.1. CSRC Plume Detection Processing Block

3.1 would be necessary for 256 by 256 pixel images at a frame rate of 1 kHz. Figure 3.2 shows a CSRC architecture block diagram containing eight processing blocks, each with external data and control connections. No communication between the processing blocks would be necessary. The following sections discuss the mapping of the plume detection problem to the architecture shown in Figure 3.2 and the corresponding resource allocation.

3.3 Algorithm Mapping

The plume detection problem can be implemented on the CSRC architecture shown in Figure 3.2 by having the processing blocks operating in parallel on different subimages. Dividing the 256 by 256 image into sixteen 64 by 64 pixel subimages allows each processing block to serially operate upon two subimages. The subimage data is loaded into the processing blocks' RAM from the external interface. The external interface steps through the processing blocks reading the updated velocity estimates and writing the next video frame for the appropriate subimage. Given proper RAM bus arbitration, the subimages can be double buffered to allow the next frame to be loaded while the FPGAs operate on previous frames. A similar double buffering approach could be used for the output velocity estimates.

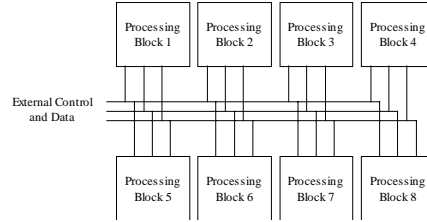


Figure 3.2. CSRC Plume Detection

Table 3.1 describes the mapping of the plume detection problem for a single subimage to the CSRC processing block shown in Figure 3.1. Nine processing steps are listed in Table 3.1 in the order in which they

Step	FPGA1		FPGA2	
	Layer	Operation	Layer	Operation
1	1	Read partial subimage (16x16) $I_{(x,y,t)}$ and previous frame partial subimage $I_{(x,y,t-1)}$ from RAM		
2	2	Compute I_x and output I_x to FPGA2 (4 pixels at a time)	1	Compute E , I_x and keep a running sum of E
3	3	Compute I_y and output I_y to FPGA2 (4 pixels at a time)	1	Compute E , I_y and keep a running sum of E
4	4	Compute I_x and output I_x to FPGA2 (4 pixels at a time)	2	Add I_x to E and compute bE product
5	Repeat steps 1 through 4 to complete subimage			
6	1	Read partial subimage (16x16) $I_{(x,y,t)}$ from RAM		
7	2	Compute I_x and output I_x to FPGA2 (4 pixels at a time)	3	Update E
8	3	Compute I_y and output I_y to FPGA2 (4 pixels at a time)	3	Update E
9	Repeat steps 6 through 8 to complete subimage			

Table 3.1. CSRC Optical Flow Algorithm Mapping

switches to context layer 3 to compute I_y and FPGA2 continues to use layer 1 to add $\hat{v} I_y$ to E . FPGA1 then switches to layer 4 to compute I_x for step 4 and FPGA2 switches to layer 2 to add I_x to E . Steps 2 through 4 are repeated for each partial subimage until the entire subimage has been processed and the E summation is complete. When the E summation is complete, FPGA layer 2 computes the product bE used for velocity estimate updates. For step 6, FPGA1 reloads context layer 1 and reloads the most recent frame of the partial subimage. FPGA1 then computes I_x and I_y in steps 7 and 8 the same as in steps 2 and 3 and FPGA2 switches to layer 3 to use I_x and I_y to update the velocity estimates which are stored in RAM. FPGA2 could use its fourth layer following step 8 for additional functions such as velocity thresholding or grayscaling. Steps 6 through 8 are repeated until the velocity estimates have been updated for the entire subimage. The FPGA and memory resource utilization for this mapping of the optical flow algorithm are discussed in the following sections.

3.4 Processing Requirements

Table 3.2 lists the number of FPGA clock cycles necessary for each step of the optical flow

Step	Cycles	Notes (all steps operate on 4 words at a time, memory accesses are assumed to take 2 cycles)
1	$C_1=298$	$(17 \times 5 + 16 \times 4) \times 2$ Most recent frame requires 17×17 to compute I_x and I_y
2	$C_2=136$	$(16 \times 4) \times 2 + 8$ Pipelined operation with 8 cycles of setup and 2 cycles per 4-word vector (2 multipliers)
3	$C_3=136$	$(16 \times 4) \times 2 + 8$ Pipelined operation with 8 cycles of setup and 2 cycles per 4-word vector (2 multipliers)
4	$C_4=68$	$(16 \times 4) \times 1 + 8$ Pipelined operation with 4 cycles of setup, 1 cycle per 4-word vector and 4 cycles for bE product
5	$C_5=10208$	$16 \times (C_1 + C_2 + C_3 + C_4)$ 64x64 subimage made up of 16 partial subimages
6	$C_6=170$	$(17 \times 5) \times 2$ Only most recent frame required
7	$C_7=272$	$(16 \times 4) \times 4 + 16$ Pipelined operation with 16 cycles of setup and 4 cycles (1 read and 1 write) per 4-word vector
8	$C_8=272$	$(16 \times 4) \times 4 + 16$ Pipelined operation with 16 cycles of setup and 4 cycles (1 read and 1 write) per 4-word vector
9	$C_9=11424$	$16 \times (C_6 + C_7 + C_8)$ 64x64 subimage made up of 16 partial subimages
Image total	346112	$16 \times (C_5 + C_9)$ 256x256 image made up of 16 subimages

Table 3.2. CSRC Optical Flow Processing Requirements

algorithm. Each 64-bit RAM access is assumed to take two clock cycles. The number of clock cycles required for the pipeline operation consists of two factors: (1) A one time cost for each algorithm step which accounts for the entire length of the pipeline (2) The cost for the slowest part of the pipeline for each 4-word operation. For step 1, FPGA1 reads the most recent subimage frame and the previous frame into memory. Taking into account the 2-cycle, 64-bit reads and the extra data required for the I_x and I_y calculations, step 1 requires 298 cycles for each subimage. In step 2 the slowest part of the pipeline is the calculation of $\hat{u} I_x$. The FPGA only has enough resources for two multipliers, therefore the 4-word multiplication must be done in two steps of two multiplications. With a total pipeline length of 8 cycles and a per vector cost of 2 cycles, step 2 requires 136 cycles for each partial subimage. Functionally, step 3 is identical to step 2. Step 4 does not require multipliers and therefore is assumed to consume 1 cycle per vector with an overhead of 4 cycles and an extra 4 cycles for computing the bE product. The cycle cost of step 5 is the partial subimage cost of steps 1 through 4 times the 16 partial subimages which compose a subimage. Step 6 is similar to step 1 except that the previous frame is not necessary since I_x does not need to be computed. The memory accesses for reading and writing the velocity estimates are the slowest parts of the pipeline for steps 7 and 8. With two memory accesses at 2 cycles per access the per vector cost is 4 cycles. Step 8 computes the cost of steps 6 through 8 for an entire subimage. The total image cost is then computed based upon the subimage costs for steps 5 and 9. Given an 80 MHz FPGA and a 1000 Hz frame rate, 80000 cycles are available per processing block for each frame. Eight processing blocks would provide 640000 cycles per frame which would require each processing block to operate at 54 percent of its estimated capacity. The low estimated utilization allows for errors in estimates, slower FPGA parts or added functionality such as velocity thresholding.

3.5 FPGA Resource Requirements

Table 3.3 lists the number of LCs used in each layer for memory and for arithmetic functions. When a LC is used as memory, the LC is assumed to provide 16 bits of storage. Implementing an adder in the FPGA is

FPGA	Layer	Required CLBs		
		Total	Memory (16 bits per LC)	Computational (1 LC per bit add, 300 LC per 12 bit by 12 bit general coefficient multiply)
1	1	545	17×17+16×16	0
1	2	593	17×17+16×16	12×4
1	3	593	17×17+16×16	12×4
1	4	593	17×17+16×16	12×4
2	1	698	2	300×2+24×4
2	2	298	2	200+24×4
2	3	633	1	300×2+16×2
2	4	N/A		

Table 3.3. CSRC Optical Flow Algorithm FPGA Resource

required to require one LC for each bit in the adder. For example, a 12-bit adder would require 12 LCs. A general coefficient multiply, where both multiplicands are variable, consumes an estimated 300 LCs. For the first layer of FPGA1 no arithmetic functions are performed, but LCs are required to store the partial subimage frames. FPGA1, layers 2 through 4, maintains the memory loaded by layer 1 and each require 4 12-bit adders. Layer 1 of FPGA2 needs two 12-bit

multipliers for the $\hat{u} I_x$ or $\hat{v} I_y$ calculation, four 24-bit adders for the E summation and 24 bits of storage for E .

FPGA2 layer 2 has the same requirements as layer 1 without the multipliers. Without the need for the same multipliers as layer 1, layer 2 is a good place to perform the bE multiplication at the end of a subimage. The bE multiplication can be done as a constant coefficient. Given b is a 12-bit constant and E is 24 bits, the bE multiplication would consume 200 LCs. The third layer of FPGA2 requires two 12-bit multipliers and two 16-bit adders for the velocity estimate updates and 12-bits of storage for bE . The maximum number of LCs used by any FPGA layer is 698.

3.6 Memory Requirements

Table 3.4 lists the RAM requirements for the CSRC optical flow architecture. The optical flow algorithm stores the frame data, I , and the velocity estimate data, \hat{u} and \hat{v} , in the RAM. Additional memory is assumed to be used for double buffering the input frame data and the output velocity estimates. The total memory requirement for each frame is 896 KB. Each of the 8 processing blocks having 128 KB providing a total of 1 MB.

Data	Size (bytes)	
$k(x,y,t+1)$	131072	2,256,256
$k(x,y,t)$	131072	2,256,256
$k(x,y,t-1)$	131072	2,256,256
$\hat{u}(x,y,t)$	131072	2,256,256
$\hat{u}(x,y,t-1)$	131072	2,256,256
$\hat{v}(x,y,t-1)$	131072	2,256,256
$\hat{v}(x,y,t)$	131072	2,256,256
Total	917304	

Table 3.4. CSRC Optical Flow Algorithm RAM Requirements

3.7 CSRC FPGA Alternative Architectures

Alternatives to the CSRC FPGA optical flow architecture include ASIC, integer DSP and standard reconfigurable FPGA architectures. Table 3.5 lists the possible optical flow architectures and their respective advantages and disadvantages. The CSRC FPGA architecture implements the 800 Mops optical flow algorithm in real-time while maintaining flexibility for changes to the algorithm. A disadvantage of the CSRC FPGA architecture is that programming of the algorithm would be more difficult than for a general purpose processor such as an integer DSP. An ASIC implementation of the optical algorithm could provide the fastest processing

Architecture	Advantages	Disadvantages
CSRC FPGA	Fast Flexible	Moderately difficult programming
ASIC	Very fast Power and volume efficient	Very difficult programming Expensive hardware Not flexible
Integer DSP	Very flexible Easy programming	Slow
standard reconfigurable FPGA (1 FPGA for each CSRC FPGA layer)	Fast Flexible	Moderately difficult programming Expensive hardware Volume costly
standard reconfigurable FPGA (1 FPGA for each CSRC FPGA)	Flexible	Slow Difficult implementation

Table 3.5. Optical Flow Algorithm Architecture

speed along with being power and volume efficient, however an ASIC architecture would have a large up front cost for design and would not be flexible in regards to changes in the algorithm. Using an integer DSP for the optical flow algorithm would allow for the easiest and most flexible programming, however an integer DSP would only provide about 20 Mops. The integer DSP architecture could achieve the required 800 Mops but would need 40 DSPs, which would be costly.

Two possible approaches exist for using a standard reconfigurable FPGA architecture for the optical flow algorithm. One approach is to utilize one standard FPGA for each CSRC FPGA layer. Using this approach the speed and flexibility of the CSRC FPGA architecture can be achieved, but the hardware and associated cost would be increased by a factor of over three. Alternatively, one standard FPGA can be used for each CSRC FPGA. Since reprogramming a standard FPGA takes approximately 30 ms, which corresponds to 30 frames at a 1 kHz frame rate, reprogramming could not be used in the 1-to-1 FPGA replacement architecture. Without the ability to reprogram, a different mapping of the optical flow algorithm to the FPGAs where the different functions are distributed across FPGAs would be needed. Spreading the functions across FPGAs complicates the data flow and likely creates memory access bottlenecks. Hence, the CSRC FPGA architecture provides a combination of speed and flexibility for the optical flow algorithm which cannot be matched by other architectures.

4 Conclusions

A number of signal processing algorithms have been assessed with regard to their potential to exploit the CSRC hardware architecture for computational advantage. This research indicates that the algorithms with the most potential to benefit from the CSRC architecture are those that share a significant number of memory locations between context layers. This mode of computation embodies the notion of moving the algorithm through the data. The bottleneck in an FPGA-based system is oftentimes the off-chip memory access. In other words, the most expensive part of the algorithm implementation is transferring data on and off chip. This is further burdened by the age-old pin limitation problem. The CSRC architecture allows for reduced cycles of data access by allowing the data to be loaded in *once*, process the data with a context, reconfigure (but retain the data) and continue processing with the second context, the third context, and so forth. Since multiple contexts and the concept of virtual hardware alleviate the requirement of confining the logic to a physical resource size, the data need only be transferred on chip at the start of computation and off chip at the end of computation.

Although history has shown that FPGAs are typically developed to be general in nature, this work suggests that a move towards specific algorithm development will be beneficial with the advent of context switching. Being able to develop fixed coefficient multiples, key-specific decryption algorithms, or sub-branches of a binary tree classifier algorithm are just a few of the possibilities afforded by CSRC. In essence, if a circuit can be designed that is specific, therefore smaller, typically faster, and utilizing less power, doing so makes sense given that designs can be cached and or swapped in and out in real-time.

5 Future Work

As previously mentioned, Sanders is currently developing a CSRC FPGA that is capable of single cycle reconfiguration as well as data sharing between configuration layers. Future publications will document that effort.

6 Acknowledgments

This effort was supported by DARPA/ITO under contract number F30602-96-C-0350.

7 References

- [1] A. DeHon, "Reconfigurable Architectures for General-Purpose Computing", PhD Dissertation—MIT, 1996.
- [2] J. Burns, A. Donlin, J. Hogg, S Singh, M. de Wit, "A Dynamic Reconfiguration Run-Time System", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.
- [3] S. Kelem, "Mapping a Real-Time Video Algorithm to a Context-Switched FPGA", Poster Session, *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.
- [4] S. Trimberger, "A Time-Multiplexed FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.