# Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops [*]

J. Hammes, A.P.W. Böhm, C. Ross, M. Chawathe, B. Draper, R. Rinker, W. Najjar
Computer Science Department
Colorado State University
Ft. Collins, CO, U.S.A.

**Abstract** ***This paper describes a system for compiling codes written in a conventional high-level language to reconfigurable FPGA-based hardware. SA-C is a single-assignment language, and its compiler performs a variety of optimizations, some conventional and some specialized, before generating dataflow graphs. The dataflow graphs are then compiled to VHDL. Three novel compiler optimizations are described here, all based on loops with windowing behavior.*** **Loop fusion** ***with windows requires a unique approach to producer-consumer fusion.*** **Temporal Common Subexpression Elimination** ***identifies expressions that recompute values that were computed in previous iterations, and replaces them with registers.*** **Window narrowing** ***reduces window sizes after other optimizations have been applied. Finally, the performance effects of these optimizations on a four-loop sequence are shown.***

## 1 Introduction

Perhaps the biggest obstacle to the widespread use of reconfigurable computing systems lies in the difficulty of programming them. FPGAs are typically programmed in hardware description languages such as VHDL [12]. These languages require great attention to detail, including issues such as timing and low level synchronization. The Cameron Project [7, 10] has created a high-level algorithmic language, named SA-C [6], for writing image processing applications and compiling parts of them to FPGAs. The SA-C compiler now provides one-step compilation from SA-C code to ready-to-run host executable and FPGA configurations. The compiler uses dataflow graphs internally to perform a variety of optimizations, some conventional and some unusual.

Loop fusion, temporal common subexpression elimination, and window narrowing are three SA-C compiler optimizations that interact closely and provide important performance gains. The rest of this paper begins with a brief overview of SA-C. The three optimizations are then described, using some simple examples. Finally, a four-loop sequence of image processing dilation loops is used to show the performance effects of the optimizations.

## 2 The SA-C Language

The design goals of the SA-C language are

- high-level, algorithmic language
- single-assignment, for better compiler analysis and translation to DFGs
- no pointers or side effects, for better compiler analysis
- emphasis on loops and arrays

[*]This work is supported by DARPA under US Air Force Research Laboratory contract F33615-98-C-1319.

- high-level operators for IP applications
- operator syntax and precedences as in C
- variable bit-width data types
- user control of optimizations

Data types in SA-C include signed and unsigned integers and fixed point numbers, with user-specified bit widths. Since SA-C is a single-assignment language, each variable's declaration occurs together with the expression giving it its value. (This approach prevents semantically unpleasant situations such as declaring a variable in an outer code block and assigning to it in only one part of a conditional.) SA-C has multidimensional rectangular arrays whose extents are determined dynamically or statically. The type declaration **int14 M[:,6]** is a declaration of a matrix **M** of 14-bit signed integers. The left dimension is determined dynamically; the right dimension is specified by the user.

The most important aspect of SA-C is its treatment of **for** loops. A loop in SA-C returns one or more values (i.e., a loop is an expression), and has three parts: one or more generators, a loop body and one or more return values. The generators interact closely with arrays, providing a way of expressing array accesses that is concise for the user and easy for the compiler to analyze. Most interesting is the **window** generator, which extracts sub-arrays from a source array. Here is a median filter written in SA-C:

```
uint8 R[:,:] =
 for window W[3,3] in A {
  uint8 med = array_median (W);
} return (array (med));
```

The **for** loop is driven by the extraction of 3x3 sub-arrays from array A. All possible 3x3 arrays are taken, one per loop iteration. The loop body takes the median of the sub-array, using a built-in SA-C operator. The loop returns an array of the median values, whose shape is derived from the shape of A and the loop's generator. In this example, if array A had a shape of 100x200, the result array R would have a shape of 98x198. SA-C's generators can take windows, slices and scalar elements from source arrays, making it frequently unnecessary for source code to do any explicit array indexing whatsoever.

SA-C **for** loops may have "nextified" variables, a mechanism borrowed from other loop-oriented functional languages to allow loop-carried dependencies to be expressed [11, 9]. In the absence of nextified variables, a SA-C loop is fully parallel, and this loop-level parallelism can be exploited by the compiler in various ways. The presence of a nextified variable imposes an execution order on the loop. In SA-C this order is a row-major traversal of each array accessed by the loop's generators.

A SA-C program compiles to a host machine executable that has calls to a reconfigurable coprocessor board. The system can also compile the entire program to a host executable for efficient program debugging. When compiling calls to reconfigurable hardware, it transforms bottom-level loops into dataflow graphs (DFGs) [8], suitable for mapping onto FPGAs. The host code includes interface code that automatically downloads FPGA configurations and source data, and uploads the results for further computation on the host.

The compiler performs full loop unrolling wherever it can. (A user can override this with a pragma.) In the example above, the array_median operator is an implicit loop, which becomes explicit in the compiler's internal representation. Since the array_median is being applied to an array of known size, the compiler will fully unroll this loop and replace it with a block of code that selects the median of nine values.

The SA-C compiler attempts to translate every bottom-level loop (i.e., a loop that contains no loop) to a dataflow graph (DFG), a low-level, non-hierarchical and asynchronous program representation that will be mapped for execution on reconfigurable hardware. DFGs can be viewed as abstract hardware circuit diagrams without timing considerations taken into account. Nodes are operators and edges are data paths. The dataflow graphs are designed to allow token driven simulation, used by the compiler writer and applications programmer for validation and debugging.

There are four general classes of node types in a DFG:

- arithmetic
- low level control (e.g. selective merge)
- data extraction and routing nodes that reflect the generators that drive a loop
- data collection nodes that accumulate a loop's return values

In the present system, not all loops can be translated to DFGs. The most important limitation is the requirement that the sizes of a loop's window generators be statically known.

## 3 Optimizations and pragmas

The SA-C compiler does a variety of optimizations, some traditional and some specifically designed to suit the language and its reconfigurable hardware targets. The compiler converts the entire SA-C program to an internal dataflow form called "Data Dependence and Control Flow" (DDCF) graphs [5], on which it performs all optimizations [4]. The traditional optimizations include Common Subexpression Elimination, Constant Folding, Invariant Code Motion, and Dead Code Elimination. The compiler also does specialized variants of Loop Stripmining, Array Value Propagation, Loop Unrolling, Function Inlining, Lookup Tables, Loop Body Pipelining, and Array Blocking, along with loop and array Size Propagation Analysis. Along with these, three important optimizations are now described in more detail: Loop Fusion, Temporal Common Subexpression Elimination and Window Narrowing.

### 3.1 Loop fusion

The performance of many systems today, both conventional and specialized, is often limited by the time required to move data to the processing units. The fusion of producer-consumer loops is often helpful, since it reduces data traffic and may eliminate intermediate data structures. In simple cases, where arrays are processed element-by-element, fusion is straightforward. However, the windowing behavior of many IP operators presents a challenge. Consider the following loop pair:

```
uint8 R0[:,:] =
  for window W[2,2] in Image
      return (array (f (W)));
uint8 R1[:,:] =
  for window W[2,2] in R0
      return (array (g (W)));
```

If the Image array has extents $d_0$x$d_1$, then R0 will have extents $(d_0-1)$x$(d_1-1)$ (determined by the number of 2x2 windows that can be referenced in Image.) Similarly, the extents of R1 will be $(d_0-2)$x$(d_1-2)$. This means that these loops do not have the same number of iterations.

Nevertheless, it is possible to fuse such a loop pair by examining their data dependencies. One element of R1 depends on a 2x2 sub-array of R0, and the four values in that sub-array together depend on a 3x3 sub-array of Image. Figure 1 shows, in one dimension, the back-propagating dependencies among the three arrays. Thus it is possible to replace the loop pair with one new loop that uses a 3x3 window and has a loop body that computes one element of R1 from nine elements of Image. This fusion can be expressed directly in SA-C:

```
uint8 R1[:,:] =
  for window W[3,3] in Image {
    uint8 v00 = f (W[0:1,0:1]);
    uint8 v01 = f (W[0:1,1:2]);
    uint8 v10 = f (W[1:2,0:1]);
    uint8 v11 = f (W[1:2,1:2]);
    uint8 V[2,2] =
        {{v00,v01},{v10,v11}};
    } return (array (g (V)));
```

Four 2x2 sub-arrays are extracted from the 3x3 window, each with a call to f. These four values are combined into a 2x2 array V that represents one of the windows that the lower loop had extracted from R0. Function g is called, yielding one element of R1. Note that packing the four values into array V does not degrade

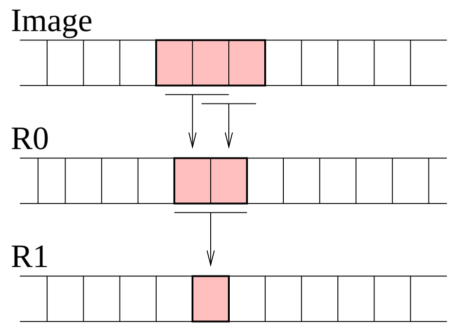

**Figure 1. Dependencies, in one dimension, of windowed loop pair.**

performance because subsequent compiler optimizations remove the array and apply the four scalar values directly to the body of function g.

Fortunately, the user can rely on the SA-C compiler to the transformation shown above. This approach has been generalized for windows of arbitrary sizes and strides. Furthermore, the resulting loop is a candidate for further fusion if its result is used by another loop. Unfortunately, this kind of fusion can create large loop bodies: The SA-C compiler aggressively inlines functions since there is no function-calling mechanism within the FPGA, and in the above example, the loop body includes four pieces of FPGA logic, each of which is a complete implementation of the function body of f. These space problems are tackled by Temporal CSE, described next.

### 3.2 Temporal Common Subexpression Elimination

Common Subexpression Elimination (CSE) is an old and well known compiler optimization that eliminates redundancies by looking for identical subexpressions that compute the same value [1]. The redundancies are removed by keeping just one of the subexpressions and using its result for all the computations that need it. This could be called "spatial CSE" since it looks for common subexpressions within a block of code.

The SA-C compiler performs conventional CSE, but it also performs *Temporal CSE*, looking for values computed in one loop iteration that were already computed in previous loop iterations. In such cases, the redundant computation can be eliminated by holding such values in registers so that they are available later and need not be recomputed.

Here is a simple example containing a temporal common subexpression:

```
for window W[3,2] in A {
  uint8 s0 = array_sum (W[:,0]);
  uint8 s1 = array_sum (W[:,1]);
  } return (array (s0+s1));
```

Here the user has performed a separate sum of each of the two columns of the window, then added the two. Notice that after the first iteration of the loop, the window slides to the right one step, and the column sum s1 in the first iteration will be the same as the column sum s0 in the next iteration. By saving s1 in a register, the compiler can eliminate one of the two column sums, nearly halving the space required for the loop body. This is the essence of Temporal CSE performed by the SA-C compiler.

To find Temporal common subexpressions, the compiler first looks at the sub-arrays that are being referenced from the window. In the above example it finds two: W[:,0] and W[:,1]. It then searches for pairs of sub-arrays that have the same shape and are horizontal shifts of each other. For each such pair, it searches the graphs beneath them for isomorphic subgraphs. Figure 2 shows a general example, assuming that the compiler has already determined that one ARR_SLICE node is a left-shift of the other. The shaded areas show the temporal common subexpressions. Note that the "external value" (external to the loop) allows the ADD nodes to be included, since it is loop invariant. In contrast, a value that is computed from inside the loop cannot be shared between the subexpressions in this way, because its value can change from iteration to iteration, and would force the ADD nodes to be excluded from the shaded areas. When the

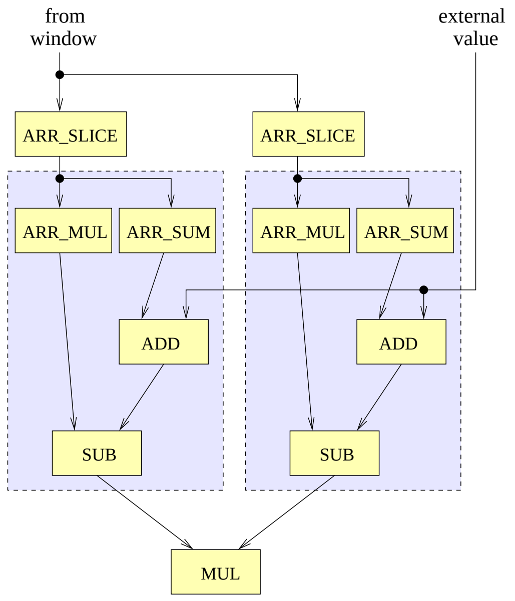

**Figure 2. Example of temporal common subexpressions in DDCF graph.**

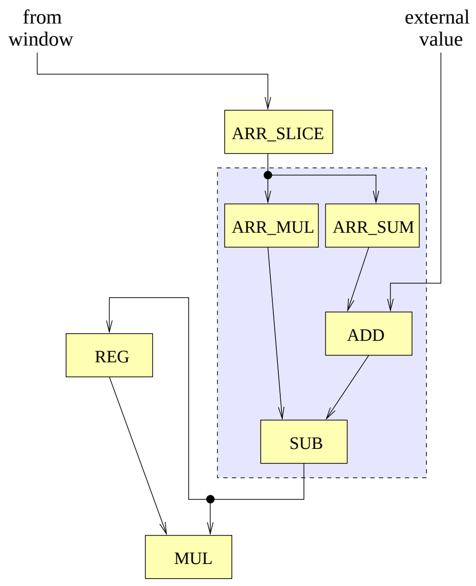

**Figure 3. Example after removal of redundant subexpression.**

common subexpression is replaced by a register, the graph of figure 3 is produced.

When Temporal CSE is performed, it is necessary to initialize the registers at the start of each horizontal traversal of the source array. This is accomplished by putting a pad on the left of the array, thereby creating extra loop iterations; the leading edge of the window begins computing just within the valid part of the array and the first iterations shift values into the registers. As will be shown shortly, this array pad is almost always eliminated by Window Narrowing.

Since extra iterations are created, the result array is larger than it should be, and it has junk values on its left corresponding to the dummy iterations that "primed" the registers at the start of each horizontal traversal. The SA-C compiler automatically deals with this by producing dataflow graphs containing "valid" signals that are fed to the result-collecting node(s) at the bottom of the loop. These signals are generated such that they tell the collectors which values to ignore.

Temporal CSE significantly reduces the space problem that can occur when windowed loops are fused. In the earlier example, where four function bodies of f were created by fusion, two of the four are eliminated by Temporal CSE.

### 3.3 Window narrowing

A useful phenomenon often occurs with Temporal CSE: one or more columns in the left part of the window are unreferenced, making it possible to eliminate those columns. Narrowing the window lessens the FPGA space required to store the window's values. Even better, narrowing the window requires a corresponding narrowing of the source array, by removing columns from the left; otherwise extra loop iterations would occur. It is serendipitous that in nearly all cases arising from loop fusion and Temporal CSE, the required narrowing of the source array exactly cancels the array pad that was introduced by Temporal CSE. In other words, an actual array pad is almost never needed!

Figure 4 shows the dataflow graph that results if TCSE and Window Narrowing are ap-

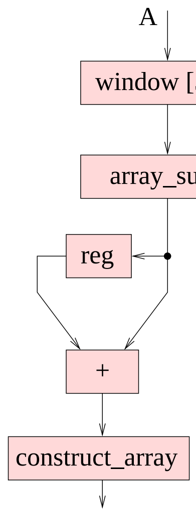

**Figure 4. Dataflow graph of loop after TCSE and Window Narrowing.**

plied to the column-sum example of section 3.2. The register is produced by the TCSE step, which in turn requires a width-one array pad on the source array and a single dummy iteration at the start of each horizontal sweep of the window. Window narrowing then removes one column of the window, as well as canceling the array pad.

### 3.4 Pipelining

After multiple applications of fusion and unrolling, the resulting loop often has a long critical path, resulting in a low clock frequency. Adding stages of pipeline registers can break up the critical path and thereby boost the frequency. The SA-C compiler uses propagation delay estimates, empirically gathered for each of the DFG node types, to determine the proper placement of pipeline registers. The user specifies to the compiler the number of register stages to place.

## 4 System Description

The SA-C compiler can be used in three modes:

1. Compile the entire program to a host executable. This is useful in the early stages of code development, for quick compiles.

2. Compile the program to dataflow graphs, called by a host executable. The system has a token-driven dataflow simulator that allows validation of the dataflow graphs produced by the compiler. There is an optional “view” mode that displays the DFGs and allows the user to single-step the execution and watch the values flowing through the graph.

3. Compile the program to FPGA codes, called by a host executable. The Cameron compiler produces VHDL code [3] that is compiled and place-and-routed by commercial software tools.

The SA-C run-time system has I/O formats that are compatible with standard image processing formats PGM and PPM. This means that after compilation to host and FPGA codes, the program is ready to run immediately on standard image files.

The current test platform in Cameron is the Starfire Board, produced by Annapolis Microsystems [2], which has a single XCV1000-BG560-4 Virtex FPGA made by Xilinx [13]. The board contains six local memories of one megabyte each. Each of the memories is addressable in 32 bit words; all six can be used simultaneously if needed. The Starfire board is capable of operating at frequencies from 25 MHz to 180 MHz. In communicates over the PCI bus with the host computer at 66 MHz.

## 5 A practical example

The Cameron Project has created a complete compilation system that compiles, with one user command, a high-level SA-C program to FPGA configurations along with a host code executable that takes care of configuring the FPGA, downloading source data, executing the FPGA loop, and uploading the result data. This makes it easy for a user to do quick experiments, obtaining executable programs in a matter of minutes or hours. The experiments described in this section were compiled in less than an hour, with nearly all the time taken by the commercial place-and-route tool.

The effects of the SA-C compiler's optimizations are now demonstrated using a well known example from the image processing world. It is common, during feature extraction, to perform a sequence of "dilate" operations, followed by a sequence of "erode" operations. There are a number of approaches to dilation and erosion; here the dilate operation will consist of using a 3x3 window to traverse the image, yielding a SA-C loop as follows:

```
uint8[:,:] dilate (uint8 A[:,:]) {
 uint8 R[:,:] =
  for window W[3,3] in A {
   uint8 m0 = array_max (W[:,0]);
   uint8 m1 = array_max (W[:,1]);
   uint8 m2 = array_max (W[:,2]);
   uint8 V[3] = {m0,m1,m2};
   uint8 mx = array_max (V);
   uint8 val =
     array_min ((uint9)(mx)+1, 255);
   } return (array (val));
 } return (R);
```

The dilate loop is typically applied multiple times. In this example, there will be four of the above loops, connected in a producer-consumer fashion.

Before fusion, the dilation loop can have Temporal CSE and Window Narrowing applied to the window columns, producing a loop with a 3x1 window and two registers. Fusion then takes place, and after each fusion Temporal CSE and Window Narrowing are again performed. The result, after the four loops are fused, is a single loop with a 9x1 window generator. The loop body contains 32 registers, 32 3-input MAX nodes, 16 2-input MIN nodes and 16 increment nodes.

If Temporal CSE and Window Narrowing had not been performed, the four loops after fusion would have a 9x9 window and a loop body with 200 3-input MAX nodes, 84 2-input MIN nodes, 84 increment nodes and no registers. Clearly these optimizations are creating a significant saving of FPGA logic space, as well as reducing data movement and intermediate arrays.

| | single loop, four times | fused loop |
|---|---|---|
| time | 28.0 msec | 16.3 msec |
| freq | 28.7 MHz | 25.0 MHz |
| space | 4% | 11% |

**Table 1. Performance comparison of unfused and fused dilate loops.**

The dilate loops, unfused and fused, have been run on a Xilinx Virtex FPGA. The fused loops were also pipelined by the SA-C compiler, to maintain a reasonable clock frequency. Table 1 shows the execution times. The tests were run on a 420x305-pixel 8-bit grayscale image.

The time improvement of the fused loop is mostly attributable to the smaller number of reads and writes to the local board memory. In the unfused case, the loop has a 3x3 window, so each row of the image is read from local memory into the FPGA three times (ignoring edge effects), and the result array is written. Thus each source pixel accounts for about four pixel memory accesses, and the four loop executions together account for about sixteen accesses per source pixel. The fused loop, on the other hand, has a 9x1 window, so it does nine pixel reads and one pixel write per source pixel. This 10/16 ratio would lead one to expect the fused loop to take about 17.5 msec if all other factors were equal. There are, however, other factors at work here; the clock frequency is a bit lower for the fused loop, yet it shows performance gains beyond those that can be accounted for by counting memory accesses. There is still much to learn in studying the performance of the codes compiled by this system.

It is also interesting to note that the four loops, fused, took less chip space than four individual loops would have taken. This demonstrates that TCSE and Window Narrowing have been highly effective at saving FPGA space, the purpose for which these optimizations were created.

## 6 Conclusions and Future Work

The Cameron Project has created a language, called SA-C, for compilation of image processing applications that target FPGAs. Various optimizations, both conventional and novel, have been implemented in the SA-C compiler. These optimizations are focused on reducing execution time and/or reducing FPGA space.

This paper has described three window-based optimizations. Loop fusion of window-based loops involves a unique approach to fusion, a side effect of which is the re-computation of some values that were computed in previous iterations. Temporal Common Subexpression Elimination looks for such expressions and replaces them with registers that carry the previously computed values into the iterations that can use them. This serves to significantly ameliorate the code size growth that this kind of fusion can produce. Finally, Window Narrowing allows left columns of windows, when unreferenced, to be eliminated. Unused window columns often occur as an effect of applying the previous optimizations.

These optimizations have been demonstrated using a loop that implements the "dilate" image processing operator. Dilation is repeatedly applied, and a fused sequence of four dilate loops showed significant time performance gains with very reasonable FPGA space cost.

Performance evaluation of the SA-C system has just begun. The fusion example in this paper can be further enhanced by other optimizations already in the compiler, especially loop stripmining. As performance issues become clearer, the system will be given greater ability to evaluate various metrics including code space, memory use and time performance, and to evaluate the tradeoffs between conventional functional code and lookup tables.

More compiler optimizations are under way, including further manipulations of window generators. Also, stream data structures are being added to the SA-C language, which will allow multiple cooperating processes to be mapped onto FPGAs.

## References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[2] Annapolis Micro Systems, Inc., Annapolis, MD. *WILDFORCE Reference Manual*, 1997. `www.annapmicro.com`.

[3] M. Chawathe, M. Carter, C. Ross, R. Rinker, A. Patel, and W. Najjar. Dataflow graph to VHDL translation. Technical report, Colorado State University, Dept. of Computer Science, 2000.

[4] J. Hammes. *Compiling SA-C to Reconfigurable Computing Systems.* PhD thesis, Colorado State University, 2000.

[5] J. Hammes and W. Böhm. *The SA-C Compiler DDCF Graph Description*, 1999. Document available from `www.cs.colostate.edu/cameron`.

[6] J. Hammes and W. Böhm. *The SA-C Language - Version 1.0*, 1999. Document available from `www.cs.colostate.edu/cameron`.

[7] J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT'99*, Oct. 1999.

[8] J. Hammes, R. Rinker, D. McClure, W. Böhm, and W. Najjar. *The SA-C Compiler Dataflow Description*, 1999. Document available from `www.cs.colostate.edu/cameron`.

[9] J. McGraw and et al. *SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2.* Lawrence Livermore National Laboratory, memo m-146 rev. 1 edition, 1985.

[10] W. Najjar. The Cameron Project. Information about the Cameron Project, including several publications, is available at the project's web site, `www.cs.colostate.edu/cameron`.

[11] R. Nikhil. *Id Version 90.0 Reference Manual.* Computational Structures Group Memo 284-1, Massachusetts Institute of Technology, 1990.

[12] D. Perry. *VHDL.* McGraw-Hill, 1993.

[13] Xilinx, Inc. *Virtex 2.5V Field programmable Gate Arrays: Preliminary Product Description*, Oct. 1999. `www.xilinx.com`.