DISSERTATION

# HARDWARE COMPILATION OF STREAMS AND PROCESSES

Submitted by

Monica Chawathe

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2006

COLORADO STATE UNIVERSITY

January 23, 2006

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY MONICA CHAWATHE ENTITLED HARDWARE COMPILATION OF STREAMS AND PROCESSES BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Adviser

Department Head

ABSTRACT OF DISSERTATION

HARDWARE COMPILATION OF STREAMS AND PROCESSES

A field programmable gate array (FPGA) is a reconfigurable hardware device on which highly parallel algorithms can be executed efficiently. Currently, the task of programming an FPGA is difficult. It involves understanding the hardware characteristics (including timing specifications) as well as understanding the software API issues. The Cameron project has developed a high-level language, called SA-C, for writing image processing application for FPGAs.

The goal of this dissertation is to expand the SA-C language capabilities and generalize the target hardware model of the SA-C compiler and make it more efficient. This dissertation investigates issues involved in mapping problem-size independent, space efficient circuits onto the target hardware model. It also expands the SA-C language to introduce non-strict data structures (streams) and concurrent processes. Introduction of concurrent processes not only allows mapping of time-efficient circuits, but also improves the expressibility of the language. This dissertation compares the space versus time efficiency issues involved when parallelizing algorithms. It identifies the conditions when certain parallelizing optimizations (like loop fusion) provide more benefit over concurrent processes.

Monica Chawathe
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Spring 2006

# ACKNOWLEDGMENTS

First, I would like to thank my adviser, Dr. A.P.W.Böhm. He not only got me interested in compiler research but also helped me develop my research skills. He has provided invaluable input, and has guided me through the ups and downs of this dissertation. I would also like to thank the rest of my committee, Dr. Ross McConnell, Dr. Sanjay Rajopadhye and Dr. Michael Kirby, for their support and guidance throughout. Special thanks to Dr. Bruce Draper as well as Dr. Phil Roxby for all their valuable input on the directions of my research.

Next, I would like to thank the whole Cameron project group for making the work environment so enjoyable and fun. To Jeff Hammes, who got me interested in compiler work. To Charles Ross, for making the compiler target model so clean and efficient, and for being there to help me debug the convoluted optimization errors in the compiler. Thanks Vidya, for writing concurrent process applications that tested the language as well as the compiler.

Special thanks to the support staff at CSU. Carol Calliham, Sharon Van Gorder and Susan Short have made it easier to deal with the CSU bureaucracy while Elaine Regelson has helped me deal with the ups and downs of a job search.

I would also like to thank my friends for all the moral support they provided. To Sangita and Nischal, for getting me motivated when I was down and celebrating with me when I was excited about a breakthrough. To Pierre and Meg, for allowing me to spend holidays with you when I decided to stay back and not visit my family. To my tennis team, for helping me take my mind off work.

Also, I would like to thank my parents for making me the person I am. Thanks Aayee and Baba, for instilling the importance of hard work and good work ethic. Thanks, for always motivating me and believing in me. I would also like to thank my brother, Yatin, for all the support and encouragement.

Finally, I would like to thank my husband, Alexandre, for being so supportive and for always being there for me.

# DEDICATION

To my loving husband, Alexandre.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

# Introduction

General purpose computing systems provide acceptable performance for a wide variety of applications. If an application requires better performance, one can pursue a hardware approach by designing an application specific integrated circuit (ASIC). The performance gain in the hardware approach is obtained by directly executing the computation as a parallel circuit, there by avoiding the overhead of instruction traffic and interpretation in the von Neumann model. Reconfigurable computing systems (RCS) explore a hybrid of the hardware and software solutions. In RCS, the underlying hardware is usually a field programmable gate array (FPGA). FPGAs are flexible (or reconfigurable) and can be reprogrammed repeatedly with different configurations at runtime. In RCS, they can be coupled with a conventional CPU. This host processor, under software control, performs the reconfigurations of the FPGA. It can also perform some irregular computations that are difficult to map onto the FPGA.

However, the task of programming these reconfigurable computing systems can be difficult. It involves identifying parts of the program that can be run on an FPGA and the ones that need to be executed on the conventional CPU. The task of generating configuration codes that execute the identified tasks on the FPGA requires the programmer to understand hardware characteristics including timing specifications. The programmer also needs to understand and handle interfacing the FPGA based system with the host system. This requires the programmer to deal with API software issues. Most programmers either have the software/API knowledge or can program the hardware well (but rarely have knowledge of both aspects).

The Cameron project [7] has designed a high-level, single assignment programming language called SA-C for writing image processing applications for reconfigurable systems. The goal of the SA-C language is to hide the hardware design details and the interfacing issues from the programmer. The SA-C language is an extended subset of C. It has variable bit-precision data-types and data structures like rectangular $n$-D arrays. Variables in SA-C are strict (completely defined before their use), monolithic (defined as a whole in one statement) and single assignment (can be assigned a value only once). In SA-C, the declaration of a variable and assignment of the value occurs in the same statement. These restrictions simplify the compiler analysis and exploitation of parallelism. The language supports *for* and *while* loops and treats them like expressions; each loop produces results that must be assigned to some variable.

The SA-C compiler takes a SA-C program as input, performs optimizations (general purpose as well as architecture-specific) [9] and then identifies the innermost loops as the tasks that can be mapped on an FPGA. These are not necessarily the innermost loops in the original program, because the original loops are modified by optimizations like loop-unrolling and loop fusion. The compiler generates VHDL code for the identified tasks along with C host code and interface code. Currently, the SA-C compiler targets the Annapolis MicroSystems StarFire [3] and WildStar [4] boards.

The mapping of a single loop on the FPGA restricts the applications that a programmer can effectively run on the hardware. Consider a size-$N$ FFT application. It has $log(N)$ stages; each stage has $N/2$ butterflies. (Each butterfly has one complex number multiplication, addition and subtraction). To be able to run the FFT algorithm on an FPGA, the SA-C compiler either needs to map a nested loop structure on the hardware, or it must completely unroll the inner $N/2$ butterflies and have them execute in parallel. The unrolling optimization makes the SA-C loop corresponding to the $log(N)$ stages be the inner loop for the compiler. This allows it to be mapped on the FPGA using the single-loop model. The problem with this approach is that as $N$ grows larger, the parallel butterfly circuit gets larger and may no longer fit on the FPGA.

Also, the strictness of data structures causes an increase in latency for the applications. Consider a size-$N$ pipelined sort algorithm with $log(N)$ merge stages. In the presence of strict data structures, the second stage of the network cannot start until the first stage has completely produced all the intermediate results, even though the second stage only needs the first and the third data element to start its computation. Similarly, the third stage of the network needs the first and the fifth element for its first computation. The strictness of data structures simply delays the start of execution of the next stage and thereby precludes parallel execution.

The goal of this dissertation is to increase the expressibility of SA-C, generalize the target hardware model and make it more efficient and portable. This dissertation looks at mapping multiple loops (including nested loops) onto the hardware. It identifies and implements the changes and extensions needed to the abstract hardware architecture model to be able to map multiple loops simultaneously. It also looks at expanding the SA-C language. It adds non-strict data structures like streams and allows concurrent processes in the new SA-C* language. Streams in SA-C* are simply first-in first-out (FIFO) queues (with their individual elements being strict data structures). Non-strict data structures give rise to dependent processes that run in parallel; for example, the producer and consumer of a stream can run concurrently. The dissertation explores language changes needed to express streams and processes. It also explores issues involved in executing these dependent processes in parallel on the host, in data flow simulation model, and in abstract hardware model simulation. This abstract hardware model closely mimics the FPGA hardware behavior.

The main contributions of this dissertation answered the following questions: How can the SA-C* compiler map problem-size independent, space efficient circuits onto an FPGA? How can the SA-C* compiler generate time efficient circuits on an FPGA? How can the expressibility of the SA-C* language be improved? Do we need I and M-structures in SA-C*? What are trade-offs between space efficient and time efficiency in SA-C* codes?

The organization of this dissertation is as follows: Chapter 2 provides some background. Chapter 3 covers the mapping of problem-size independent circuits. It covers restrictions in the old, abstract hardware model, followed by the expansions and modifications necessary to allow mapping of nested and aggregate loops. Chapter 4 covers the SA-C* language expansion to include non-strict data structures (streams) and concurrent processes. Chapter 7 provides an overview of the questions answered as well as the possibilities of future work.

# Chapter 2

# Background

## 2.1 Field Programmable Gate Arrays

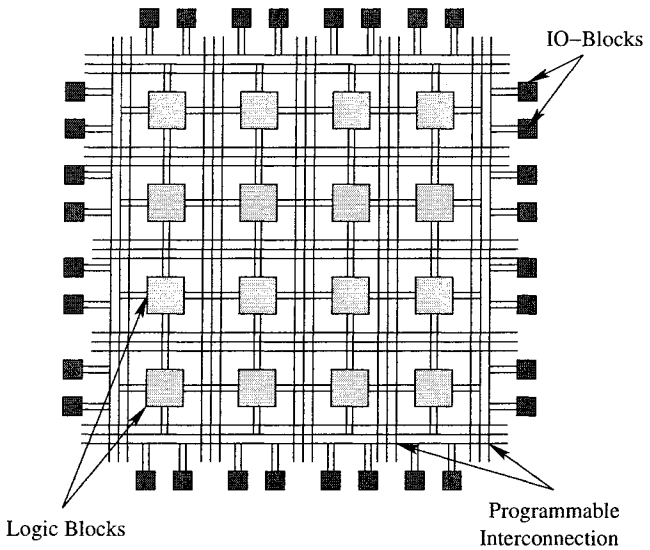



Figure 2.1: FPGA block diagram

A field-programmable gate array (FPGA) [13] [39] is a reconfigurable hardware device that acts as a computation intensive coprocessor and can be reconfigured repeatedly by a conventional CPU host processor to perform a variety of applications. It consists of an array of programmable logic blocks. A typical logic block consists of 4-input lookup tables and flip-flops. The single output of the lookup tables may or may not be registered using the flip-flops. The logic blocks are surrounded by programmable I/O blocks and are connected via programmable interconnects as shown in Figure 2.1.

FPGAs are generally slower than application-specific integrated circuits (ASICs) and draw more power. The main advantage of an FPGA over an ASIC is its ability to be re-programmed. Hence, an FPGA can be fabricated independent of the target application. This increases the time to market as mapping an application involves reprogramming the FPGA (and does not involve fabrication). Programming an FPGA involves bit-level specification of the logic blocks and interconnections between them. The identification of which logic blocks to use and how to interconnect them is fairly complex and sophisticated. Hence, the hardware designer generally provides a design for the problem using a hardware design language like VHDL or Verilog, or by providing a schematic design. Commercial software is then used to synthesize and place and route the hardware design on a specific FPGA device.

Over time FPGAs have become more complex and more difficult to program. The Xilinx [47] Virtex family of FPGAs provides on-chip block RAM capability and has dedicated multiplier circuits distributed across the FPGA chip. The Xilinx PRO family additionally provides power PCs on the chip.

## 2.2 Cameron Project

Since reconfigurable hardware is currently programmed using a hardware description language like VHDL, it is difficult to program it for large applications. Hence, there is a need to shift the paradigm for programming adaptive and reconfigurable systems from a hardware centered approach to a more software centered one. The objective of the Cameron project [7] [20] [34] was to develop a high-level, single assignment, C-like programming language for writing image processing applications for reconfigurable hardware. The first language developed by the Cameron project was called SA-C. Data-structures in SA-C were strict. This dissertation extends the SA-C language to include non-strict data-structures and concurrent processes. This new language is called SA-C*.

### 2.2.1 Old SA-C Language

The SA-C language [17] supports eight basic data-types for scalars: boolean, bits, signed integers, unsigned integers, signed fixed point numbers, unsigned fixed point numbers, floats and doubles. The current hardware run-time system does not support floating point arithmetic, thus code involving floats and doubles is always executed on the host. Since the hardware can support different bit sizes, the language allows specification of the size and/or the precision of integers and fixed point numbers. For integers, the size specifies the total number of bits to be used on the hardware. For fixed point numbers, the first size specifies the total number of bits to be used and the second size specifies the number of bits corresponding to the fraction part. (uint6, ufix16.4 are examples of SA-C data-types.)

The SA-C language also provides multi-dimensional rectangular arrays of scalar types. The images used in image processing applications can easily be expressed as rectangular arrays. The language places emphasis on access mechanisms over arrays like element generators, slice generators and window generators.

In SA-C, there are three types of statements: print, assert and assignment. All other constructs are expressions and yield values. Print and assert statements are provided for debugging purposes while the assignment statements correspond to the main logic of the program. The assignments are strict (completely defining the left-hand side variable before its use), monolithic (variables must be defined as a whole in one statement) and single assignment (a variable can be defined only once). The right-hand side of the assignment corresponds to an expression that performs the computation of the value.

Expressions can be arithmetic and logical operations over scalar types. The operators are similar to C arithmetic and logical operators. The language also supports various array operators like array_min, array_max, array_sum, array_and, array_or, etc. These operations are performed with or without an accompanying mask array.

The language supports conditional expressions ('?' ':' operator, if-else, switch-case), while-loops and for-loops. Since the old SA-C hardware model only supports mapping

of a single loop where the number of loop-iterations is known before the execution of the loop (and hence resultant array size is known), while-loops are not mapped onto the hardware. A for-loop is divided into three parts: generator(s), a loop body and return expression(s). The generators provide access mechanisms over arrays (window generator, element generator, slice generator). One can also use compound operators over the generators, such as dot products and cross products. The loop body is made up of arithmetic operators, array operators, etc. It may also contain induction variables, updated every loop iteration. This does not conflict with the single-assignment principle, because the loop body for each iteration is single-assignment. The induction variable can update its current iteration value using the previous iteration values. Thus the variable has a single value per iteration. The new value for an iteration can be assigned by using the keyword *next* instead of its type declaration of the induction variable. The return values of the loop can be the final value of a nextified variable, or loop reductions, or structure building operators. The loop reductions can produce either scalar values (*sum, min, max, and, or*) or fixed-sized arrays (*vals_at_first_min, vals_at_last_max*). They operate over a sequence of data being produced by the loop (one component per iteration). The structure building operators like *vector, matrix, array* build a one, two or $n$-dimensional arrays respectively by grouping values created at each iteration.

### 2.2.2 Old SA-C Compiler

Figure 2.2 shows the overall compilation process of the old SA-C compiler and the various components involved. The SA-C compiler takes a SA-C program as its input. It transforms the SA-C program into an intermediate form called a data dependence and control flow graph (DDCF) [18]. A DDCF is an acyclic, hierarchical data flow graph and has a direct correspondence with SA-C language constructs. Several optimizations [8] [9] [11] can be performed on the DDCF graph; some are general optimizations (like common subexpression elimination, constant-fold, dead-code elimination, operator strength reduction) while others are specific for improving the performance on reconfigurable hardware (like loop unrolling, temporal CSE [19], lookup tables). Each optimization

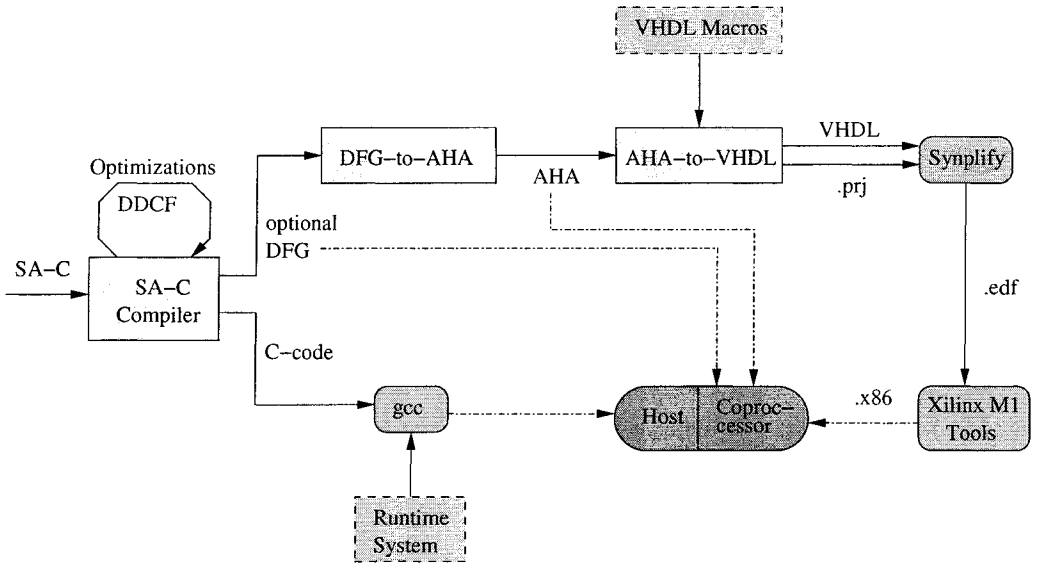



Figure 2.2: Cameron project overview

transforms the DDCF graph into a new and equivalent DDCF graph. The DDCF graph can then be transformed into plain C-code that is run on a host processor. This can be used by the programmer to verify the correctness of their SA-C program logic.

The compiler can optionally extract parts of the SA-C program that can be run on the coprocessor (a simulator or the reconfigurable hardware). In the old, restrictive, SA-C model, only the innermost for-loops can be extracted for execution on the hardware. They need not be the innermost loop from the original SA-C program but they are innermost after all the optimizations are run. The compiler then transforms these parts into flat, non-hierarchical, data flow graphs (DFGs) [21]. The rest of the DDCF is transformed into host C-code. The functionality of the DFGs can be tested using a DFG simulator as the coprocessor. DFGs show the data dependences among operations and hence show which operations can execute concurrently.

The DFGs are converted into Abstract Hardware Architecture (AHA) graphs. These graphs contain timing and synchronization information (unlike DFGs that are data-driven). Also, the granularity of the nodes in AHA graphs is finer than in DFGs. The functionality of AHA graphs can be tested by using the AHA simulator as the coprocessor. AHA graphs can be translated into VHDL code for the reconfigurable computing

systems. (The old SA-C compiler targets the Annapolis MicroSystems StarFire[3] and WildStar[4] FPGA boards). The AHA to VHDL translator also links in static VHDL library routines during the code generation phase. This code is then synthesized into *.edf* files using Synplify [45] and then place-and-routed to obtain RCS configuration codes (*.x86* files) using Xilinx Foundation Tools. These configuration codes are run on the FPGA coprocessor.

## 2.3 Streams and Processes

### 2.3.1 Stream Processing Systems

A Stream Processing System (SPS) [43] is comprised of a collection of *modules* (processes) that compute in parallel and communicate data via *channels*. The communication between modules occurs over channels as an infinite sequence of data called *streams*. There are three types of modules.

1. *Sources*: These modules pass data into the system.
2. *Filters*: These modules perform atomic computation over their input data and produce output. They are also called *agents*.
3. *Sinks*: These modules pass the data from the system to the outside world.

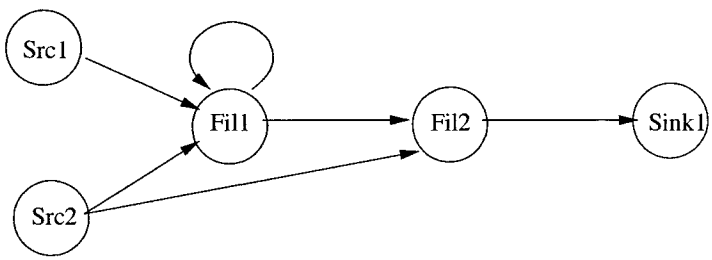



Figure 2.3: Typical stream processing system

We can classify stream processing systems by three main characteristics (of the filters and the channels).

1. *Synchronous vs. asynchronous* filters: Synchronous filters perform computation in a synchronized manner (e.g. using a global clock) but in an asynchronous system there is no synchronization with the other modules (they execute independently).

2. *Deterministic vs. non-deterministic* filters: Deterministic filters produce the same output for a given input sequence for every run while a non-deterministic filter may not always produce the same output for a given input sequence for each run. For example, a merge based on presence/absence of data on a given channel will give rise to a non-deterministic process.

3. *Uni-directional vs. bi-directional* channels: Unidirectional channels have one producer filter and one consumer filter per channel. Bidirectional channels connect two processes with the communication occurring in (possibly) both directions.

### 2.3.2 Kahn Process Model

In the Kahn model [28] [29], parallel computation is organized as autonomous computing stations connected to each other by communication lines called channels. Kahn processes are functions from input streams to output streams. They are computing units that map an input history to an output history and can run concurrently. In the Kahn model, channels (or streams) are represented as FIFOs with unbounded capacity. They carry information in one direction only (from the producer process to the consumer process). Also, writes by a process are non-blocking (that is they succeed immediately) but reads are blocking. This means that a read on an empty channel results in the process waiting for data to arrive on that channel. This model disallows checking for the presence or availability of data on an input line and hence disallows non-determinism in process networks.

The Kahn process model can be executed using a spectrum of scheduling strategies. The two extremes of the spectrum are: *pure coroutine* (where a single process is active at any time) and *fully parallel* (where all processes can run when they are not waiting for input).

1. *Coroutine*: In this mode, the scheduler marks processes as activated or deactivated to determine which process to execute. Only the active process can execute. Initially, the activation of a process is completely demand driven. The scheduler selects a process producing the final output for execution. If a scheduled process attempts a read on an empty channel, the channel is marked as *hungry* and the process attempting the read is deactivated. This is followed by activation of the producer process for that channel. Similarly, a write by a producer process on a *hungry* channel involves deactivation of the producer process and reactivation of the consumer process for that channel. Thus, the scheduler is demand driven initially but reactivation of processes becomes data-driven.

2. *Parallel Model*: In this model, both the consumer and producer process of a channel can be active concurrently. A write on a *hungry* channel results in the consumer of the channel being reactivated but does not always result in deactivation of the producer process. If the scheduler has additional resources available for concurrent execution of both the consumer and the producer process, it will not deactivate the producer process; the producer process can then continue execution in anticipation of further demands for its output irrespective of whether this extra output will be used or not. Thus in this model, some computations can be executed concurrently with the one drawback of performing some unnecessary, extra computation.

## 2.4 Communicating Sequential Processes and Occam

In Tony Hoare's communicating sequential processes (CSP) model [24], parallel processes communicate with each other using instantaneous message passing over non-buffered channels. The communication is synchronized, that is, if two processes communicate and only one of the communicating process is ready, it will block until the other process is ready for communication. The CSP model allows non-determinism using guarded communication; a guarded input command (in a consumer process) is selected for execution only if the producer for that channel is ready for communication.

Occam [25] is a parallel computing language based on the CSP programming model. It allows reads and writes over channels using the '?' and '!' operators respectively. Occam also has the *seq* and *par* constructs that allow the user to specify statements that need to be executed sequentially or in parallel. The alternation construct *alt* allows the programmer to specify guarded constructs. Each alternative can be a combination of a boolean condition and a channel read, followed by a statement block. Only a true condition and a read on a non-empty channel results in the statement block being executed.

## 2.5 Synchronous DataFlow

In a data flow model, programs are represented as directed graphs with nodes representing concurrent operations and edges representing communication. Each node has a set of associated firing rules that determine the state in which the node can execute. Generally, the firing rules are based on the presence/absence of data on the edges and the amount of data present. Synchronous DataFlow [30] is a special case of data flow in which the number of data samples produced and consumed on each edge is fixed and specified a priori. Hence the firing schedule of the nodes can be computed statically, and deadlock and boundedness properties are decidable.

The data flow graph model in SA-C differs from SDF. The amount of data samples consumed and produced by a SA-C DFG cannot be specified a priori. This is because the rates vary depending on the state of the SA-C DFG node; the rate of consumption may be different for a DFG node in an initial state while the rate of production may be different in the final state. The firing schedule of the SA-C DFG cannot be computed statically because of these variable rates.

## 2.6 I and M-Structures in pH

The pH (parallel Haskell) language is a successor to the Id dataflow language [5], using the notation and type systems of Haskell [23] by having eager/data-driven semantics.

pH can be viewed as a language with three layers: functional language core, I-structures (incremental) and M-structures (mutable). The functional core guarantees deterministic behavior and has no notion of state. pH is a non-strict functional language, that is, functions can execute and return values before arguments are available (unless restricted by data dependencies). This feature adds more expressiveness to the language.

The second layer of pH adds I-structure capability to the language. I-structures provide the capability of declaring an empty data structure in one place and assigning values to individual elements in another. At the declaration statement (or initialization), the I-structure is allocated and the components are left empty. Later, a computation can assign values to these empty components. Any computation that reads an empty component blocks until another computation fills a value in that location. I-structures are deterministic because of their "write-once" semantics.

The final layer of pH adds M-structures to the language. Unlike I-structures, they allow rewriting. The difference between assignment statements in imperative languages and M-structures is implicit synchronization. Consider

```
h!j := f(h !& j)
```

$h!\&j$ denotes and *extract* ("wait until full, read, leave empty") operation while $h!j$ is a *put* ("write and make full") operation. Note that functional core and I-structures imply sequencing of operations by the data dependencies and creation of deterministic processes. This is lost in M-structures and non-determinism is introduced.

Sequencing operations may be necessary to control non-determinism. Sequencing of operations on M-structures can be done by only using the *extract* and *put* operations (and avoiding simple reads). *pH* also allows explicit sequencing of instructions by using barriers. In pH, barriers are represented using ">>>". The statements before a barrier are executed completely before any statement after the barrier begins execution. Barriers are needed in algorithms like 2D FFT where the row-wise FFT needs to be completed before the column-wise FFT can start.

# Chapter 3

# Generalized Hardware Model for Loops

## 3.1 Old, Restrictive SA-C Loop Model

The old SA-C compiler put restrictions on which SA-C code sections could be mapped onto hardware (via data flow graphs (DFGs) and abstract hardware architecture (AHA) graphs). The restrictions were put in place because of the underlying restrictions in the old hardware model. These restrictions can be categorized as follows:

1. *Innermost loops*: The old hardware model only allowed flat, non-nested graphs. Hence, only a single, innermost loop from the SA-C program could be translated into an individual AHA graph for hardware mapping. Note that this is the innermost loop after the compiler optimizations but need not be the innermost loop in the original SA-C code.

2. *Known loop iterations*: The old hardware model also needed a loop driver that knew the number of times the loop body executed at the start of loop execution. Hence, the number of iterations of the loop mapped on the hardware needed to be a compile-time or run-time constant. This restriction implied that only for-loops could be translated into AHA graphs.

3. *Known output data size*: The old hardware model did not allow on-the-fly memory-mallocs. All the mallocs needed to be performed by the host prior to the start of

the loop. This allows scalar-reductions (like *sum, min, max*), compile-time fixed size array reductions (like *vals-at-min/max*) and *array/tile* reductions for for-loops to be translated into AHA sub-graphs.

These restrictions prevented while-loops and nested loops from being translated into AHA graphs. While-loops were simply run on the host processor while the innermost loops of the nested loops got translated into AHA graphs with their outer-loops being executed on the host processor.

### 3.1.1 Old SA-C to DFG Translation

The SA-C compiler identifies the code section in a SA-C program that satisfies the hardware model restrictions. This identified section is the core that is mapped onto the coprocessor (simulator or hardware) while the rest of the SA-C code is executed onto the host. The coprocessor core can be translated into DFGs (to be executed on the DFG simulator/coprocessor), AHA graphs (for the AHA simulator/coprocessor) or VHDL (for the FPGA coprocessor). Figure 3.1 shows the code partitioning.

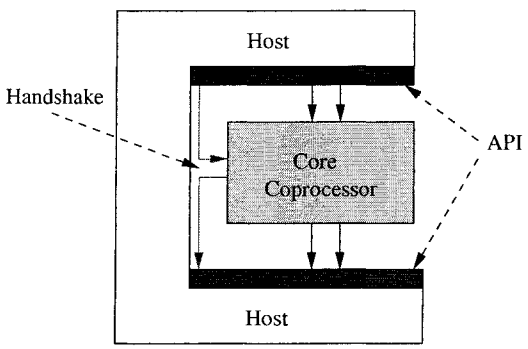



Figure 3.1: SA-C code partitioning

The SA-C compiler also adds interface code at the communication boundaries between the host and the coprocessor. This interface API is the same irrespective of whether the target coprocessor is a simulator or the FPGA hardware. This keeps host code generation independent of the target coprocessor. The host code performs the tasks of reading input data (from the command prompt, an ASCII file or a binary file),

performing irregular computations, handling interface API and displaying or saving the final results. The interface code performs the following tasks:

1. *Malloc:* Perform mallocs (on the coprocessor) for inputs and outputs of the core.
2. *Transfer run-time information*: Transfer run-time constants like address locations and dope vector information (calculated during malloc stage) to the coprocessor.
3. *Transfer inputs*: Transfer the input data to the coprocessor (in the location calculated during the malloc stage).
4. *Go signal*: Wake-up the coprocessor.
5. *Done signal*: Wait for an interrupt from the coprocessor.
6. *Transfer outputs*: Transfer the output results back from the coprocessor (from the location calculated during malloc stage).

#### 3.1.1.1 DFG Simulation

The functionality of the translated DFG graph can be tested using the DFG simulator. The DFG nodes correspond to computational units in the simulator. The edges connecting the ports of the nodes provide communication among them. Data communicating across edges is stored in unbounded FIFO queues. When a consumer requests data from an edge, it is dequeued from the FIFO queue. There are some exceptions though. Constants on input ports are sticky (always present). Also, run-time constants (data generated by the input nodes of the DFG) are sticky.

The execution of DFG nodes follow some simple firing rules. Most of the DFG nodes fire when data is available on all of its input ports. For every firing of a DFG node, non-sticky input data is dequeued from its input ports, a computation is performed and the results are outputted (queued) on its output ports. Some DFG nodes have different firing rules for the first (initial) execution and the last (final) execution. For example, Figure 3.2 shows the multiple firing rules for an induction variable; the ports for consumption and production are different during the initial, iterative and final firing.

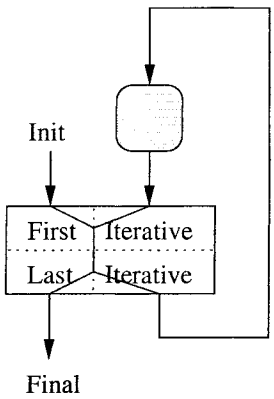



Figure 3.2: DFG firing rules for an induction variable

The simulator follows the firing rules for each DFG node. It starts the execution after receiving the wake-up signal from the host. The first nodes fired are the input nodes of the DFG. They produce the sticky, run-time constants. This is followed by execution of the rest of the DFG (based on individual node firing rules). On completion of the DFG execution, the simulator sends an interrupt (done signal) back to the host.

#### 3.1.1.2 DFG Example

Consider a SA-C program where a window of four elements slides across a 1-D image, computes the sum of the window elements and stores the resultant values in an array. Figure 3.3 shows the SA-C program for this algorithm.

```
uint32[:] main(uint32 inImg[:]) {                    // line 1
  uint32 outImg[:] =                                 // line 2
    for window W[4] in inImg {                       // line 3
      uint32 sumW = for w in W return (sum(w));      // line 4
    } return (array(sumW));                          // line 5
} return (outImg);                                   // line 6
```

Figure 3.3: A simple, for-loop SA-C program

The innermost loop (on line 4) operates on an array of compile time known size (4 elements). Hence the number of iterations for the loop is also known at compile time. The compiler fully unrolls [16] this loop to allow the sum computation over the current

window to occur in parallel. This makes the sliding-window generator loop (on line 3) become the innermost loop and it gets translated into a data flow graph shown in Figure 3.4.

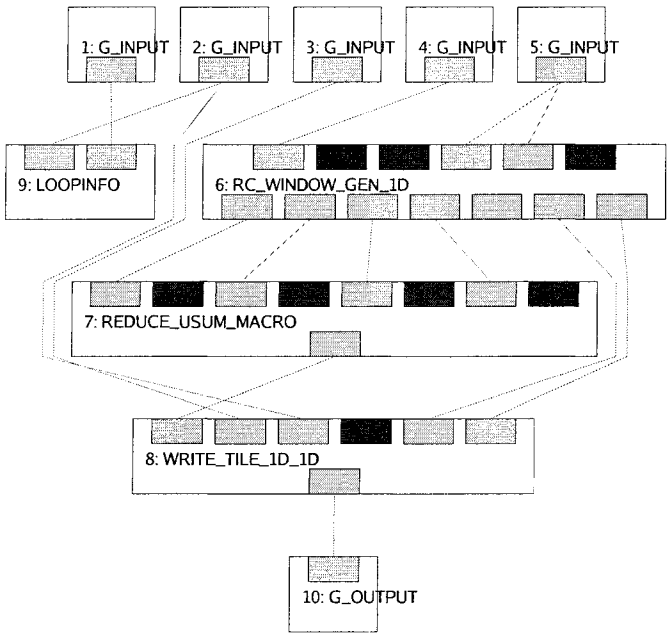



Figure 3.4: Resulting DFG for for-loop in SA-C

The simulator can test the functionality of the generated graph (shown in Figure 3.4). On receiving a wake-up signal from the host, the simulator first fires the input nodes and produces the run-time constants transferred by the host interface code as their outputs. Now, node 6 (WINDOW_GEN) has its window-size, step-size, location and dope vector information. Also, node 8 (WRITE_TILE) has the tile size information along with the address and dope vector information for the output array. Each firing of the window generator produces one valid window of four elements along with the *end_of_sequence* signal as false. Once all valid iterations are completed, the window generator produces the *end_of_sequence* signal as true (without any data on the window element edges). When a valid window is produced, node 7 (SUM_MACRO) has all its inputs and can fire. It produces a single sum result. This result is grabbed by the write-tile node and written to memory. When the write-tile receives a true *end_of_sequence* token, it

produces an output done signal. This is received by the output node and sent back to the host as an interrupt. The host interface code then simply reads the write-tile results back.

### 3.1.2 Old DFG to AHA Translation

The first version of the SA-C compiler directly translated data flow graphs to VHDL code [10]. The target model required that the loop body be purely combinational. This made handling of general array-references (inside the loop body) as well as pipelining more difficult. Also, direct translation of complex DFG nodes (like window-generator) to VHDL prevented some lower level optimizations to occur. For example, the same VHDL code was generated for both a window generator and an element generator. A new stage called Abstract Hardware Architecture [37] was added to the second version of the SA-C compiler to overcome these limitations.

The translation of DFGs to AHA graphs involves conversion of some complex DFG nodes into simpler AHA subgraphs. The AHA graph nodes differ from the DFG nodes; they are smaller, less complex and have the concept of clock and synchronization. Consider the translation of a DFG window generator node into an AHA sub-graph. In an AHA sub-graph, it is split into a token-generator (loop driver), a counter (calculating a sequence of word addresses of the input image), a read-word node, a fifo-unpack node (which splits each word into a sequence of individual pixels), a shift-register node (which groups the serial pixels to produce the sliding window) and an unpack node (which splits the packed, sliding window pixels into individual ones).

The translator can optimize the resultant AHA sub-graph in certain cases. For example, in case the pixel width and word-width are equal, the fifo-unpack node is not needed. Similarly, an element-generator does not require the shift-register and the unpack node that are needed by a window generator. During the translation from DFG to AHA graphs, the SA-C compiler first produces the most generalized AHA graph. It then performs an optimizations phase and removes the nodes that are not needed, there by making the graph smaller and more efficient.

#### 3.1.2.1 Synchronization in AHA Graphs

AHA graph nodes can be categorized into three types based on their behavior and need for synchronization.

1. *Purely combinational*: These AHA nodes do not require a clock or any synchronization. They have no internal state. e.g. adder, shifter, pack, unpack.

2. *Semi-clocked*: These nodes have the concept of a clock and state but do not handle synchronization issues. e.g. counters.

3. *Fully-clocked*: These nodes have the concept of clock, state as well as synchronization. e.g. token generator, read-word, shift-register.

Synchronization/handshaking occurs only among fully-clocked AHA nodes. Each fully-clocked node is split into two halves: the consumer half and the producer half. Each half fires independently, there by decreasing the length of the dependency chain. The combinational and semi-clocked nodes fire as a whole; they do not have the concept of halves. Thus, synchronization is necessary only among dependent producer and consumer halves of different fully-clocked nodes.

To handle synchronization at the graph level, each AHA graph is split into sections. Each section is a connected component made up of three parts: producer halves of some AHA nodes, some combinational and/or semi-clocked nodes and the consumer halves of AHA nodes (consuming the results produced). Each section can fire/execute when the producer halves can produce data and the consumer halves have space to consume and store the results. Thus the handshaking signals needed for execution are the consumer and producer half *can_half_fire* signals and the *can_section_fire* signals (obtained by anding the corresponding *can_half_fire* signals). This handshaking adds some overhead in order to satisfy the firing rules.

The AHA graph also needs an explicit *end_of_sequence* signal. This signal is produced by the loop driver (token-generator) and is passed through the first port of every fully-

clocked node. This allows the clocked nodes to reset themselves once they have completed their execution (that is, once they receive a true *end_of_sequence* signal).

The AHA simulator can test the functionality of the generated AHA graph before mapping it onto hardware. The host interface code for the AHA simulator has the same API as the DFG simulator. The compiler simply links in a different library for the AHA simulation and API. Also, the compile-time and run-time constants are considered to be sticky in the AHA model (just like in the DFG model). The AHA simulator also handles the individual *can_half_fire* signals for each AHA node and the synchronization of AHA sections with their respective *can_section_fire* signals.

#### 3.1.2.2 AHA Example

Figure 3.5 shows the AHA graph (after optimizations) for the example SA-C code shown in Figure 3.3. The DFG window-generator is translated into an AHA subgraph of node 7 (TOK_GEN), node 8 (COUNTER), node 9 (READ_WORD), node 10 (SHIFT_REGISTER) and node 11 (UNPACK). Note that the fifo-unpack (pixel serializer) AHA node is removed by optimizations because the input image is 32-bit wide (equal to memory data bus width).

For synchronization purposes, the AHA graph is split into sections. For example, node 5 (INPUT) and consumer half of node 7 (TOK_GEN) form one section that starts the loop while producer half of node 7 (TOK-GEN), node 8 (COUNTER) and consumer half of node 9 (READ_WORD) form another section which calculates the addresses for image pixels.

The *end_of_sequence* signal is explicitly produced by node 7 (TOK_GEN) and passes through the first port of every fully-clocked node before reaching node 6 (OUTPUT). In the DFG graph, the output node receives exactly one done signal (always true) when the collector (like write-tile) has completed execution. In the old AHA graph format, the output node receives a series of *end_of_sequence* values (a set of false values followed by a single true value) that have been produced by the token generator and passed through. Hence in AHA simulation, the simulator needs to wake up the host processor only after

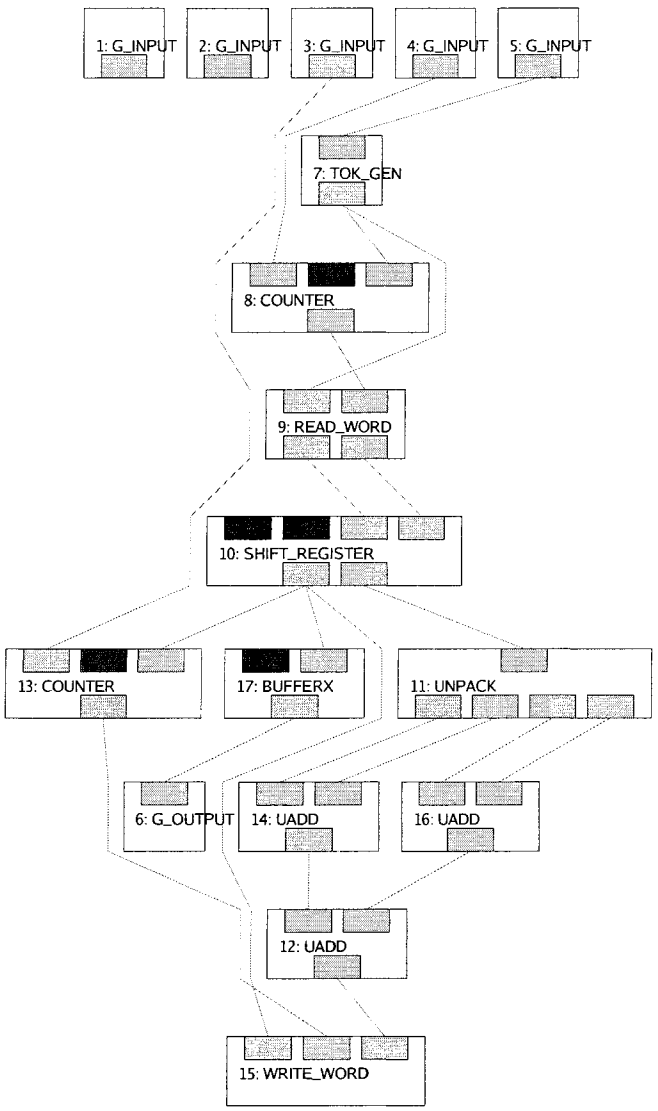



Figure 3.5: Resulting AHA graph for SA-C for-loop

receiving the final, true value on its output node; all false values received are simply ignored by the output node.

## 3.2 New, Generalized SA-C* Model

Mapping of nested loops and while loops onto hardware requires changes to the target hardware model and to the SA-C* language and the compiler. The new SA-C* language allows the programmer to select the exact loop nests that need to be mapped onto

hardware. It allows the programmer to allocate loop nests onto FPGA hardware and arrays in specific memories (or block RAMs). Selecting the individual memories (and block RAMs) in which to store the loop inputs and outputs allows the user to decrease the contention issues. The new DFG model allows nested graph formats, there by simplifying nested loop transformations. The new abstract hardware model has been simplified to make graph analysis easier.

### 3.2.1 Code and Data Allocation in SA-C*

In the restrictive model, it was easy for the compiler to identify innermost loops for mapping onto hardware. The problem with allowing nested loops onto hardware is that the compiler does not know what level of nesting needs to be mapped onto the hardware. Hence the language needs to be modified to allow the user to specify the exact loop that needs mapping. This is done using a *hardware* construct in the SA-C* language. Figure 3.6 shows a for-loop with the hardware construct in the new SA-C* language.

```
uint32[:] main(uint32 inImg[:], uint32 incr) {
  hardware() {
    uint32 outImg[:] =
      for ele in inImg {
      } return (array(ele+incr));
  };
} return (outImg);
```

Figure 3.6: For-loop with hardware construct in SA-C*

The hardware construct, by default, maps the identified loop on chip 1 of the AlphaData [2] board. The programmer can override the default board-chip combination by specifying them in the hardware construct as *hardware(board: board_identifier, chip: chip_number)*. The *board_identifier* has the format *name_number*. The *name* corresponds to the name of the board (like alphadata, wildstar, starfire) while the *number* is used to distinguish among multiple boards of the same name. Thus if the programmer had two AlphaData boards, they would be identified as *alphadata_0* and *alphadata_1*. Some

boards (like WildStar) have more than one chip on them. The programmer can select among the multiple chips by specifying the integer identifying the chip. For example, the second chip on a WildStar board is selected by setting the chip number as 2.

The SA-C* compiler by default uses memory 1 to store loop inputs and memory 2 to store loop outputs. In case of multiple inputs or outputs, the programmer may want to malloc them onto different memories (if available) or block-rams. This can decrease the contention and hence improve execution speed. The location override can be done using the *input* and *output* user pragmas on the loop. For example, the location override for the for-loop in Figure 3.6 is shown in Figure 3.7.

```
uint32 outImg[:] =
  // PRAGMA(input(inImg memory 2 -), output(bram))
  for ele in inImg {
  } return (array(ele+incr));
```

Figure 3.7: Memory location override in SA-C*

### 3.2.2 SA-C* DFG Model

The new DFG model for SA-C* is similar to the previous model in that it is made up of computational units (DFG nodes) that are connected via edges. The difference is that the new DFGs can be nested (that is, the DFG nodes can be compound). New firing rules are established to handle these compound nodes. Data needs to be transfered into the compound node and results need to be transfered out of the compound node. The constants (both compile-time and run-time) are still sticky in the new format while the other edges in the graph are represented by unbounded FIFO queues. The new DFG simulator implements the extra firing rules needed to handle the transfer of data across the boundaries of a compound node.

### 3.2.3 SA-C* Abstract Hardware Model

The SA-C* compilation process can be partitioned into two parts: compilation of SA-C* programs to the abstract hardware model (via DFGs) and translating the abstract

hardware graphs into VHDL code for mapping on FPGAs. The design and creation of the compiler has been a team effort. This dissertation focuses on the translation up to the abstract hardware model while Charles Ross' PhD dissertation work will focus on the mapping to VHDL [40].

The abstract hardware model needs extensions and changes to allow mapping of while loops and nested loops. The new hardware model, called the aggregate, hierarchical, abstract hardware architecture (AHAHA), was designed by Charles Ross following the requirement guidelines provided by me. The changes to the new model can be summarized as follows: the new AHAHA model allows the number of iterations of a loop to be data-dependent (thus allowing translation of while loops). To simplify translation of nested loops, the run-time constants in the AHAHA model are no longer sticky. Details of the new AHAHA model and the AHAHA nodes can be found in Appendix H.

Translating nested loops into AHAHA graphs also requires changes in the array interface handler of the SA-C* compiler. Mapping a single loop on hardware required interfacing only between the host processor and the coprocessor. The mallocing of space for input and output arrays as well as the transfer of loop IO was handled by the host processor. Mapping nested loops requires parts of the interfacing to be handled on the coprocessor itself. The host processor is still responsible for all the space mallocs; no run-time mallocs by the coprocessor are allowed. The generated AHAHA graphs are responsible for the serialization of the access among the producer and consumer loops. The SA-C* compiler achieves this serialization by providing some synchronization between the producer and consumer loops; the synchronization prevents the start of the consumption before the producer loop has completed execution (and generated the whole array). Section 3.3.3 explains, in detail, the interface handler in AHAHA graphs.

### 3.2.4 SA-C* Translation to New AHAHA

Consider a SA-C* for-loop program that adds a run-time constant to every element of a 1-D array (as shown in Figure 3.6). The SA-C* compiler first partitions the SA-C* code into parts that are executed on the host processor and the core that is mapped onto the

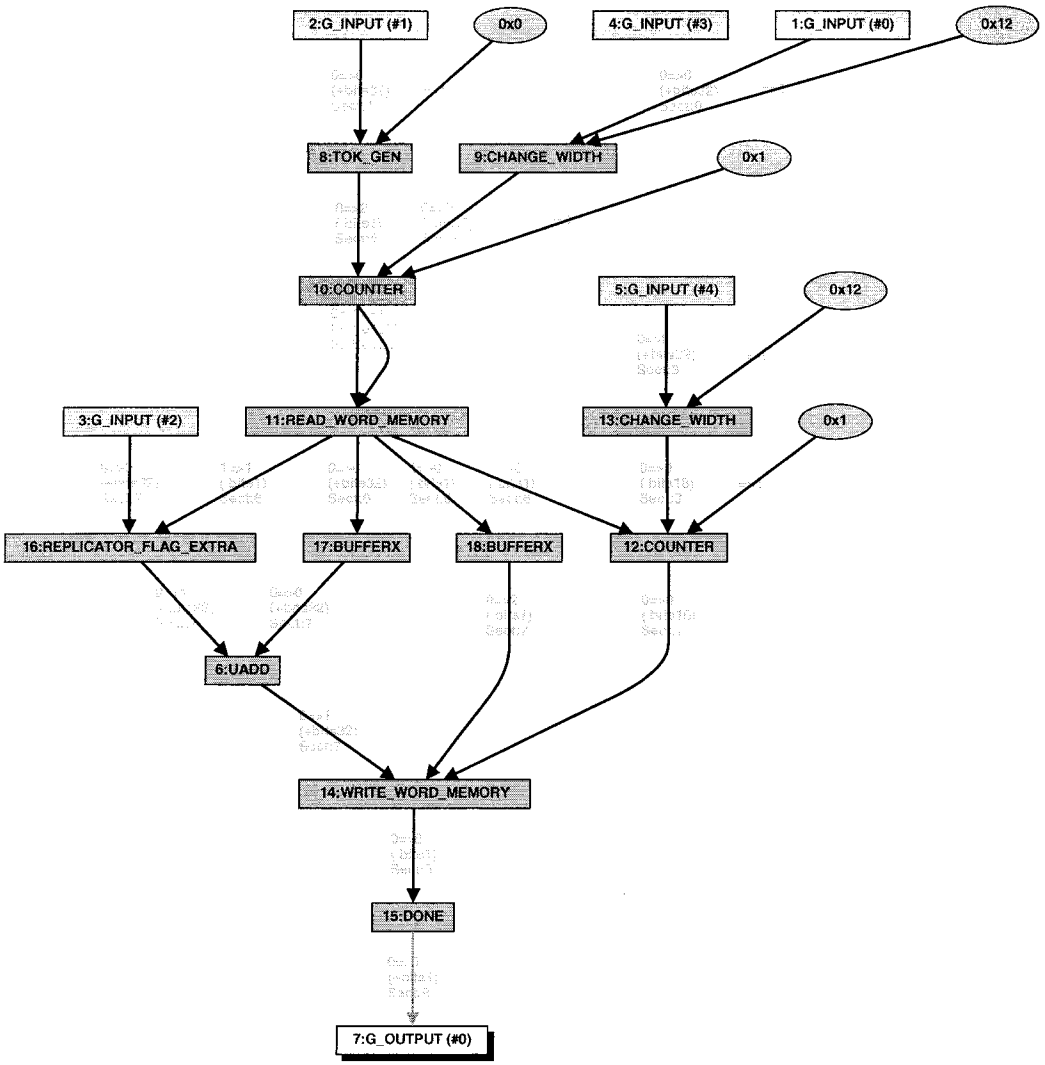



Figure 3.8: AHAHA graph for SA-C* for-loop program

coprocessor (simulator or hardware). This is followed by generation of API calls that handle the interface between the host and the coprocessor. The core is then converted to a DFG (just like in the old system). This is followed by conversion of the DFG to a generalized AHAHA graph and optimizations to simplify the final graph.

The mapping from SA-C* to DFG does not change from the old system though there are subtle differences in the translation of DFG to the new AHAHA model. The restrictions added as well as changes made to the new AHAHA model are taken into consideration in the new translator. Figure 3.8 shows the AHAHA graph for the example

SA-C* code shown in Figure 3.6. In the new model, the run-time start-address constants used by counters are converted into 18-bit (address bus width) values. Also, the counters become fully-clocked nodes. Hence the *end_of_sequence* signal produced by the token-generator passes through them (instead of bypassing them). The increment value (output of node 3) used inside the loop-body is replicated the number of times the loop-body is executed (that is, the number of times the read-word produces an output value). Also, the *end_of_sequence* signal does not bypass the write-word node (to handle write-latencies greater than one). The *end_of_sequence* signal then passes through a *done* AHAHA node; this produces exactly one true *end_of_sequence* signal.

## 3.3 New SA-C* Transformations

Generalizing the target hardware model expands the types of SA-C* codes that can be efficiently transformed into AHAHA graphs. The new model allows the number of iterations for the loop to be data dependent. This implies that while-loops can be translated into AHAHA graphs. Multiple clusters for AHAHA nodes allows conditional execution of AHAHA subgraphs; this allows translation of SA-C* code with side-effects (like array access with out-of-bound test) to AHAHA graphs. Nested DFG graph format, removal of sticky inputs at the AHAHA graph level and partial interface handling in AHAHA graphs allows translation of SA-C* nested loops into AHAHA graphs.

### 3.3.1 While Loops

The new AHAHA model allows the number of iterations for the loop to be data dependent. This implies that while-loops can be translated into AHAHA graphs. The only restriction is that the results must have known-sizes (and hence host-calculable dope vectors). Hence scalar reductions (like *sum, min, max, and, or*) and fixed-size array-reductions (like *vals_at_first_min, vals_at_last_max*) for while-loops can be translated. The structure building reductions will produce a variable sized array for a while-loop and hence are not mapped to AHAHA graphs. Variable sized arrays can be efficiently mapped onto AHAHA graphs using SA-C* streams (as explained in Section 4.4)

The body and reduction operators for a while-loop are translated to an AHAHA graph, just like the body and reductions of a for-loop. The main difference is the translation of the loop-driver. The token-generator acts as a driver for the for-loop, while the loop-test acts as the driver for the while-loop. Care must be taken when handling this loop-test. Consider a while loop that has $n$ iterations. The loop-body has $n$ valid iterations while the loop-test has $n+1$ valid iterations. Hence care must be taken when handling replication of data for the loop-test of a while-loop versus the loop-body. This is done by using a replicator for the loop-body (just like in a for-loop loop-body) but using a circulate node for replicating data for the loop-test of a while-loop.

Consider a SA-C* program that finds the sum of first $n$ integers ($\sum_{i=0}^{n-1} i$). Figure 3.9(a) shows a SA-C* for-loop program for finding the sum, while Figure 3.9(b) shows a SA-C* while-loop program for the same. The AHAHA graph for the SA-C* for-loop code is shown in Figure 3.10(a) while the AHAHA graph for the SA-C* while-loop code is shown in Figure 3.10(b).

```
uint8 main(uint8 n) {
  uint8 end = n-1;
  hardware() {
    uint8 outSum =
      for uint8 i in [0~end] {

      } return (sum(i));
  };
} return (outSum);
```

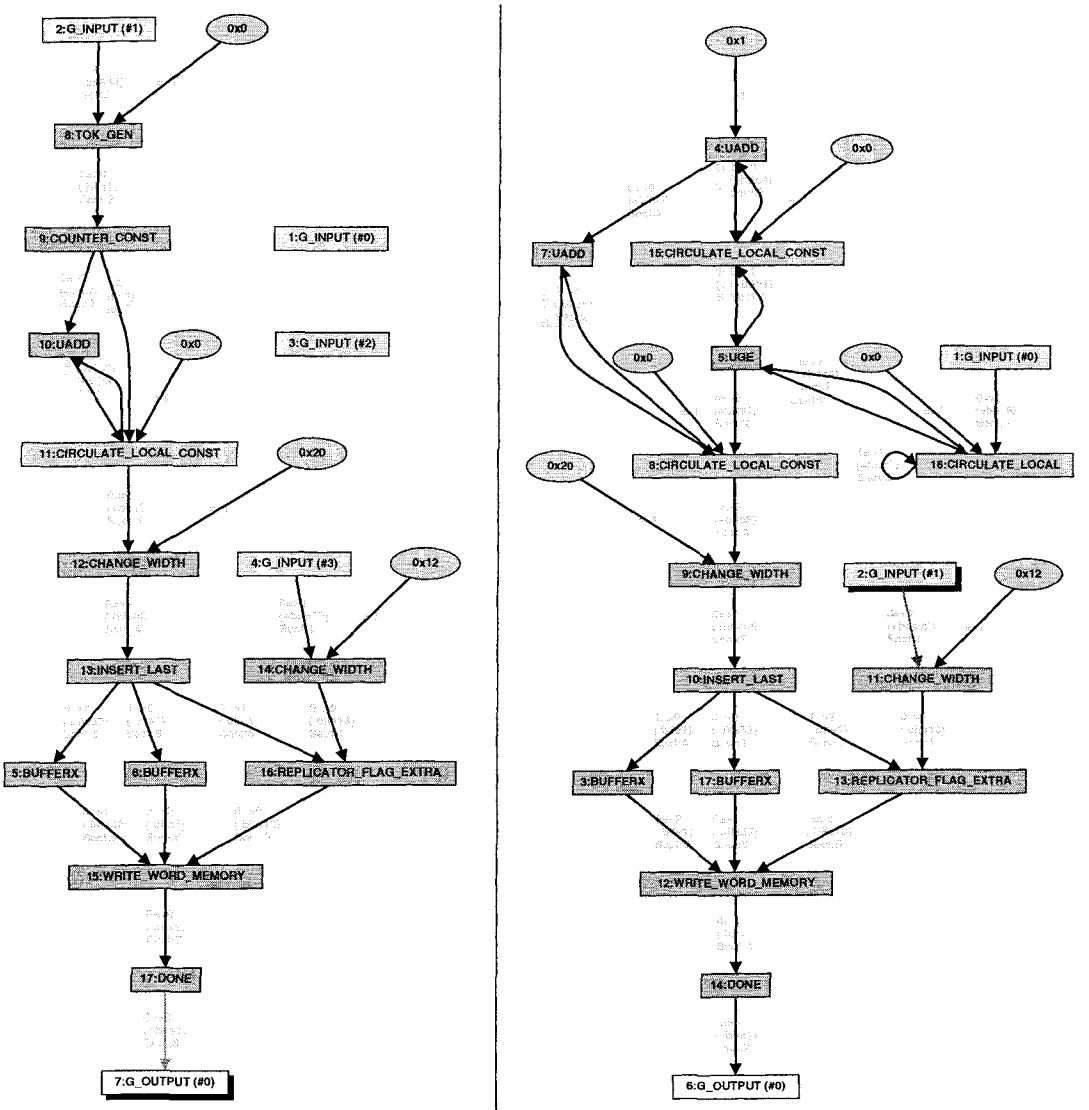
(a) For-loop

```
uint8 main(uint8 n) {
  uint8 i = 0;
  hardware() {
    uint8 outSum =
    while (i < n) {
      next i = i+1;
    } return (sum(i));
  };
} return (outSum);
```

(b) While-loop

Figure 3.9: SA-C* programs to find sum of integers

The AHAHA sub-graph for sum-reduction is the same for both the for-loop as well as the while-loop. It is made up of a summation part and a part to store the result. The summation is accomplished by a CIRCULATE node and a UADD node. This is followed by the sub-graph to write the result into memory. The difference in the AHAHA graphs lies in the driver for the loops. The scalar-generator of the for-loop is converted into a




(a) For loop

(b) While loop

Figure 3.10: Resulting AHAHA graphs for sum of integers

AHAHA sub-graph of node 8 (TOK_GEN that receives the number of iterations) and node 9 (COUNTER). The driver for the while-loop consists of a AHAHA subgraph that computes the induction variable $i$ (node 4 (UADD) and node 15(CIRCULATE)) and the loop-test comparator (node 5(UGE)). The comparison value (input $n$) is used by the loop-test multiple times and hence is replicated using node 16 (CIRCULATE).

### 3.3.2 Nested Loops

The generic graph format for creation, storing and loading of graphs in the SA-C* compiler allows nested graphs but the old DFG model did not allow such nested graphs; only flat DFGs were allowed. Allowing nested DFGs requires a small change to the target DFG model. The change requires new firing rules to handle the transfer of inputs and outputs across the boundaries of nested, compound nodes.

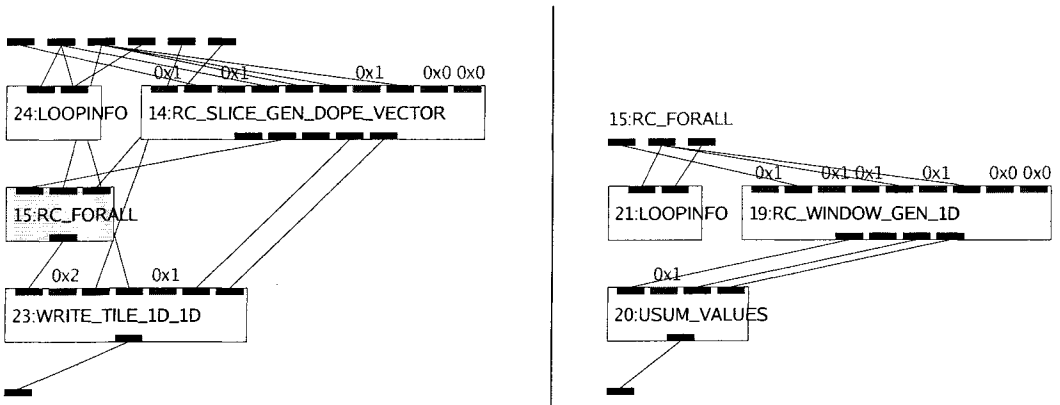
The new AHAHA model does not allow nested, compound nodes. This is because sections across nested boundaries are difficult to handle. The conversion of the nested DFG into the new AHAHA model is done in three steps: convert of each DFG node into AHAHA subgraph, flatten/remove the nested structure and optimize and make the AHAHA graph more efficient.

Consider the SA-C* program shown in Figure 3.11. This program calculates the sum of the elements of each row of a 2D array. The outer loop performs the slicing of the 2D array into rows, while the *array_sum* on line 5 performs the summing of elements per row. Figure 3.12(a) shows the outer loop with node 15 corresponding to the loop performing the *array_sum* operation. The details of the *array_sum* loop are shown in Figure 3.12(b).

```
uint32[:] main(uint32 inImg[:,:]) {              // line 1
  hardware() {                                   // line 2
    uint32 outImg[:] =                           // line 3
      for row(~,:) in inImg {                    // line 4
        uint32 sumRow = array_sum(row);          // line 5
      } return (array(sumRow));                  // line 6
  };                                             // line 7
} return (outImg);                               // line 8
```

Figure 3.11: A nested loop program in SA-C*

Node 14 (SLICE_GEN_DOPE_VECTOR) produces a sequence of start addresses (one per row). The inner for-loop (node 15) uses these sets of start address and the dope vector for each row. For each row information received, node 19 (WINDOW_GEN) produces




(a) Outer loop

(b) Inner loop

Figure 3.12: Resulting nested DFG in SA-C*

a sequence of row pixels that are consumed by node 20 (USUM_VALUES) to keep an accumulated sum result. On receiving a done signal for the current row, node 20 outputs the result (sum of the current row). This result is received by the output node of the inner loop (node 15). On arrival of this result, the inner loop (node 15) identifies the completion of one execution of itself. It transfers the result to its output port and resets itself into an initial state and gets ready for consumption of the next row. The resetting involves removal of the sticky-input tokens so that they can be replaced by the new row information.

Figure 3.13 shows the AHAHA graph for this example code. The slice generator of the DFG graph is mapped into node 8 (TOK_GEN) and node 9 (COUNTER to calculate the start address for each row) while the write-tile node is mapped into node 16 (COUNTER to calculate write address), node 18 (WRITE_WORD to write the sum result for each row) and node 19 (DONE). Since the inner for-loop is actually part of the loop body of the outer loop, the data (from input nodes) that is used by this inner loop needs to be replicated. Node 20 (REPLICATOR) repeats the number of pixels per row. Then the inner loop is converted to an AHAHA subgraph. The window generator is translated into node 10 (TOK_GEN), node 12 (COUNTER) and node 13 (READ_WORD) while the sum-value is mapped into node 14 (UADD) and node 15 (CIRCULATE). Since the

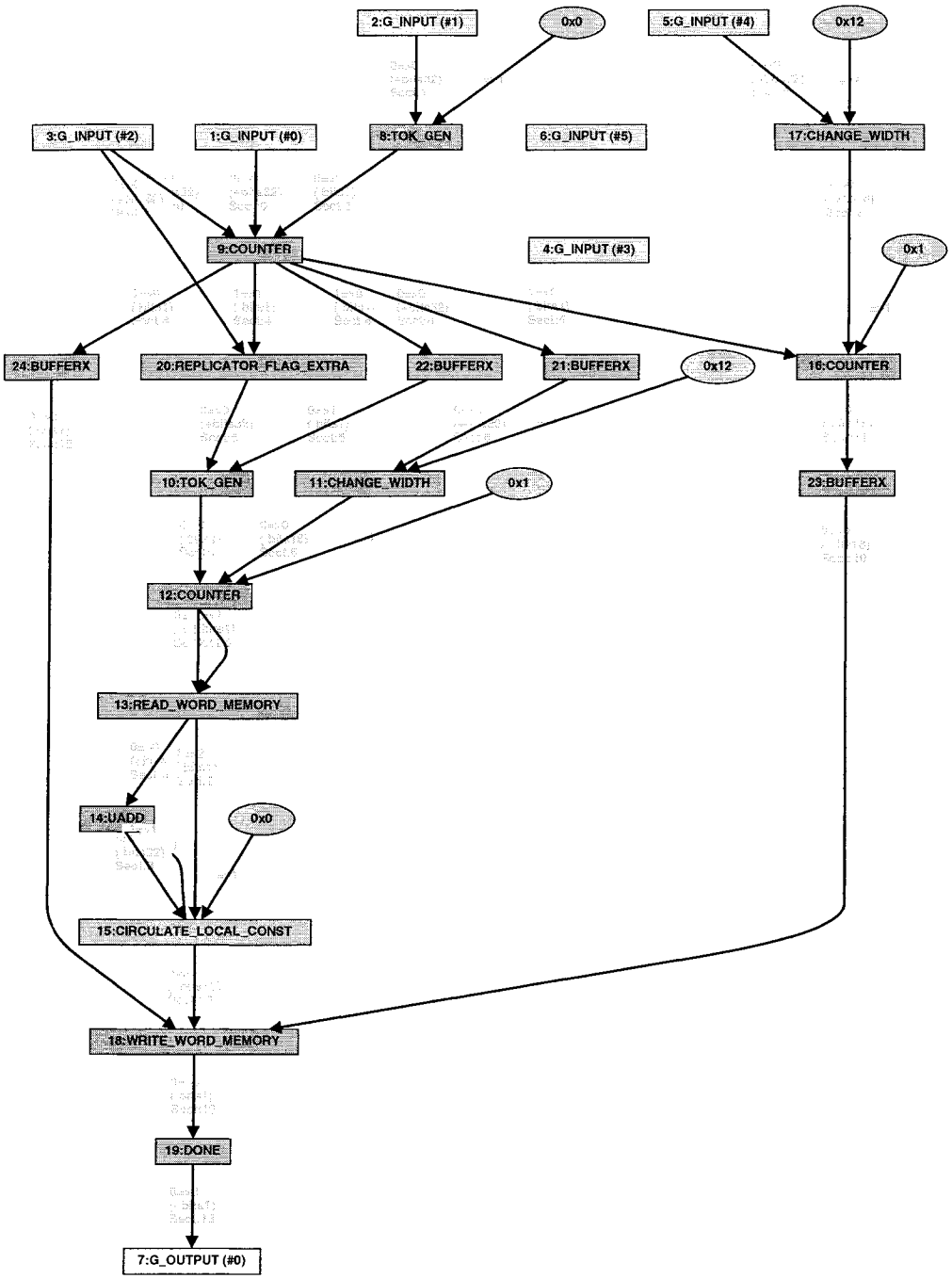



Figure 3.13: Resulting AHAHA for nested loop in SA-C*

new AHAHA graph model does not have sticky input tokens, the second phase of the conversion can simply remove the inner-loop wrapper. This removal does not affect the flow of data. Note that if a value from the host input is being used by loop body of

the inner-loop, it will get replicated twice (once by the outer loop and once by the inner loop).

### 3.3.3 Array Interfacing in SA-C*

The SA-C* programmer can specify which memory (or block RAM) to use for malloc-ing inputs and outputs of the core. The space allocation is handled by the host interface code and not by the coprocessor. The host interface code performs the malloc operations for the inputs and outputs of the core and simply transfers the run-time constants corresponding to the location information to the coprocessor.

Care must be taken when interfacing intermediate arrays (arrays not transfered to/from host). The space for intermediate arrays is allocated once by the host processor but the serialization of the access to the allocated space is handled by the coprocessor. The serialization among the producer and consumer loop is handled using a *hold* node (as shown in Figure 3.14). This node provides the location information to the consumer loop only after the producer loop has completed generating the array. If this intermediate array is created repeatedly (nested inside a loop), the producer loop must re-utilize the allocated space only after the consumer loop has completed execution. This is achieved using the *array_info_select* node (as shown in Figure 3.14).

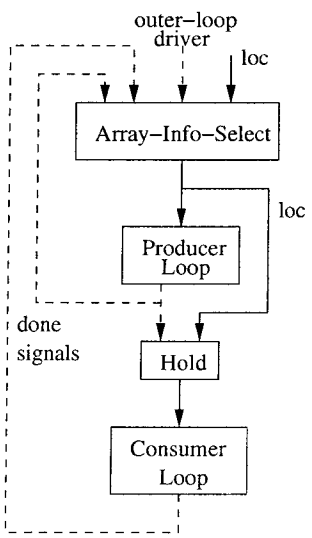



Figure 3.14: Intermediate array interface

The interface for the array induction variables differs slightly from the intermediate array interface. This is because there are multiple types of consumers of an array induction variable and only one type of consumer for intermediate arrays. The consumer loops of an intermediate array always follow the producer loop but there are three types of consumers for an induction variable (as shown in Figure 3.15): ones consuming the array generated in the previous iteration, ones consuming the array produced in the current iteration and the ones consuming the final value of the induction variable. The first two types of consumers are nested inside the loop driving the induction while the third type lies outside this driver loop. Note that the consumers of the previous iteration array never consume the final value of the induction variable while the consumers of the current iteration array never consume the initial value of the induction variable. Also, the consumers of the final value of the induction variable consume exactly one array (and not arrays generated every iteration).

```
A[:] = ....
finalA = for ..... {
  // Consumers of previous value of A
  .....
  next A = producer loop
  // Consumers of next value of A
  .....
} return (final(A));
// Consumers of final A
.....
```

Figure 3.15: A simple, for-loop SA-C program

To handle the different types of consumers of an induction variable, the AHAHA graph must provide three sets of addresses to the three different types of consumers (as shown in Figure 3.16). The *array_info_select* node generates the location of the final value of the induction variable exactly once while the other two locations are generated once for every iteration of the outer loop. Also, note that the producer loop and the consumers of the current iteration value are handled similar to the intermediate array interface using

a *hold* node. The consumers of the previous iteration value initially receive the location of the initial array from the *array_info_select* node. This is followed by location used by the producer loop in the previous iteration.

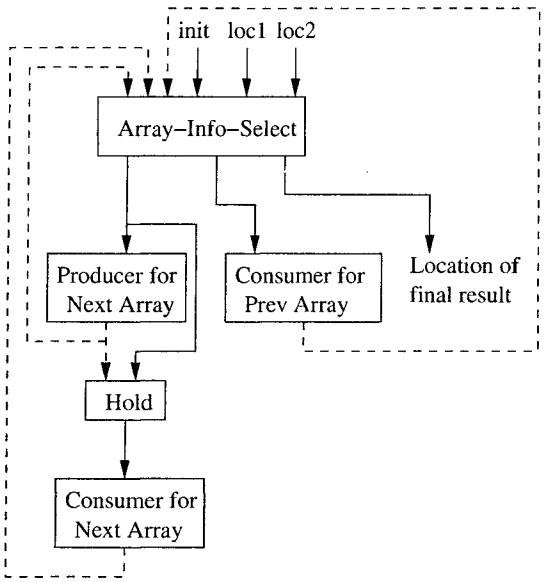



Figure 3.16: Array induction variable interface

The performance of the outer, driving loop can be improved by providing two locations for the different types of iterative consumers to toggle between, thus allowing parallel execution. The host processor actually provides three locations for an array induction variable: a read-only location containing the initial value of the induction variable, and two other locations (to improve performance by toggling their use). During each iteration of the outer loop, one of the toggle locations is used by the loops consuming the previous iteration value of the induction array while the second toggle location is used by the producer and consumer loops of the next value of the induction array.

The SA-C* programmer can request the use of specific memories for toggle locations by using the *input* and *output* pragmas (as shown in Section 3.2.1). If the programmer requests only memory (with two locations in it) for toggling, the performance gain by parallel execution is overshadowed by memory contention among the consumers of the

previous iteration value and the consumers of current iteration value. The programmer can request two different memories or block RAMS for the toggle locations by using *memory_toggle* or *bram_toggle* in the *input* and *output* pragmas.

```
// PRAGMA(input(inImg memory_toggle 1 2 -), output(bram_toggle))
for ele in inImg
....
```

### 3.3.4 Conditional Execution

The old SA-C compiler had restrictions on which conditional codes could be mapped onto AHA graphs. Only conditional code with purely combinational logic could be mapped. Codes having memory accesses (read or write) were considered to have state information and could produce side-effect (like out-of-bounds access). Such codes were not mapped into AHA graphs in the old SA-C compiler.

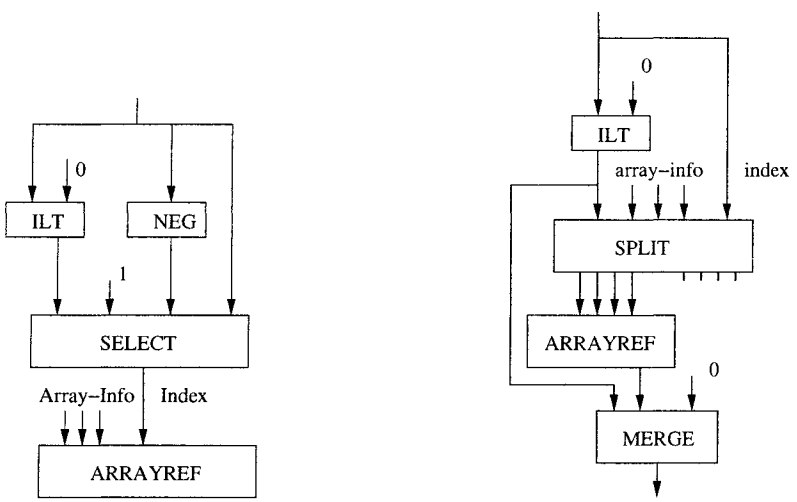
The new SA-C* compiler translates purely combinational code into AHAHA sub-graphs by executing each part of the condition in parallel and performing a selection on the results of each sub-part. Mapping of stateful code (or code that can produce side-effects) needs to be different. The AHAHA graphs needs to perform selective execution of sub-parts with side-effect instead of parallel execution.

Consider a loop body written in SA-C* shown in Figure 3.17. Line 1 calculates the absolute value of *ele*. This code is purely combinational and has no side-effects. The SA-C* compiler generates a graph that executes both parts of the conditional code in parallel along with a selector at the bottom to select the correct result based on the comparison result. Figure 3.18(a) shows the corresponding graph for SA-C* code on line 1 and 2.

```
int32 absIdx = (ele < 0)?(-ele):ele;        // line 1
int32 val1 = A[absIdx];                     // line 2
int32 val2 = (ele < 0)?0:A[ele];            // line 3
```

Figure 3.17: Conditional loop body code in SA-C*

Line 3 shows SA-C* code that performs array indexing only if the index is positive. If this code is executed in parallel, a potential side-effect (reading out of bounds) can occur. The SA-C* compiler hence generates a graph where selective execution occurs (based on the comparison result). Figure 3.18(b) shows the graph for code on line 3. The split node guides data to the array-reference (conditional sub-graph 0) only when the comparator value is false (that is, zero). Similarly, the merge node selects data from the array-reference (conditional sub-graph 0) when the comparator value is false (zero) and selects the constant zero (conditional sub-graph 1) when the comparator value is true (one).




(a) No side-effect

(b) With side-effect

Figure 3.18: Loop body graphs for conditonal code in SA-C*

## 3.4 Optimizations: Partial Unroll in Inner Dimension

Partial unrolling [9] of a loop allows multiple loop bodies to be executed in parallel. In SA-C*, the programmer can specify the amount of unrolling by using a *part_unroll* pragma. This pragma not only allows for parallel execution of loop bodies but also decreases multiple reads of the same memory locations (thus increasing data reuse).

Consider a SA-C* sliding window that overlaps in the outer dimensions. (Note that an overlapping window has a window size greater than the step size). If the loop is partially unrolled in the outer dimensions, the loop body will have data reuse among the overlapping sections of the sliding window and will read the overlapped parts only once. If there is no overlap among sliding windows, there is no reuse of data read. Since the time taken to execute outer dimensions of a SA-C* loop depends directly on the time spent in IO communication (reading of inputs and writing of outputs), the time taken to execute a loop with no data reuse will be the same (irrespective of the amount of unrolling) and will be dependent on the input and output data size.

Partial unrolling in the innermost dimension differs slightly from the outer dimensions because the host processor packs the data in the innermost dimension (to save space) before transferring it to the coprocessor. This requires the AHAHA graph to read all the words in the innermost dimension (irrespective of the window step), serialize them into pixels using a *fifo-unpack* node, identify the garbage pixels (when the innermost dimension does not end at word boundary) using a *mask* node and then pack them according to the window size and step using a *shift_register* node. The time taken for execution of the innermost dimensions of a SA-C* loop is dependent on the time spent in serialization of input pixels (and packing of output pixels). The problem with partial unrolling in the innermost dimension is that the input pixel serialization always needs to be present (to remove the garbage pixels) irrespective of the amount of unrolling. Thus partial unrolling in the innermost dimension can provide benefits only if the pixel serialization can be decreased (or completely removed).

Consider a SA-C* program (shown in Figure 3.19) that adds a scalar value to every pixel in a 1-D byte array. Since each word in memory is made up of four bytes, one would instinctively assume that performing four additions in parallel (by using partial unrolling) will result in a four fold performance gain. Figure 3.20(a) shows a partial AHAHA subgraph corresponding to the *add-scalar* SA-C* program that has been partially unrolled four times. The problem is that partial unrolling of a loop in the innermost

```
uint8[:] main(uint8 A[32], uint8 s) {
  hardware() {
    uint8 R[:] =
      // PRAGMA(part_unroll(4))
      for a in A {
      } return (array(a+s));
  };
}return (R);
```

Figure 3.19: Add-scalar SA-C* program

dimension does not remove the pixel serializer (*fifo_unpack* and *mask* nodes) and hence cannot provide advantages like partial unrolling in the outer dimensions.

The compiler can decrease the serialization of pixels if it can identify whether the pixels of the input array end on word boundaries. In such situations, the compiler can safely remove the *mask* node. In our example *add-scalar* code, the byte-array has 32 pixels (which fit perfectly in eight words). Since the image ends at a word boundary, the compiler does not generate the *mask* node during the AHAHA graph creation. Figure 3.20(b) shows the resultant AHAHA graph without the mask node. This decreases the circuit size slightly, but still does not remove the pixel serialization overhead.

Removal of the *mask* node does open up the possibility of a peephole optimization to minimize the serialization of pixels. This peephole optimization identifies the *fifo-unpack* nodes immediately followed by a *shift-register* node. The greatest common divisor (GCD) of the *depth* of the *fifo-unpack* node, the *depth* and the *step* of the *shift-register* node indicate the amount of serialization that can be decreased. In our *add-scalar* example, the GCD is four. Hence, one can decrease the serialization four-fold. This is achieved by dividing the *depth* of the *fifo-unpack* node, the *depth* and *step* of the *shift-register* node by the GCD and multiplying the serialization bit-width by the GCD (as shown in Figure 3.20(c)). Care must be taken to reorder the pixels packed in each word because $pixel_0$ is stored at the LSB in a memory word while $pixel_0$ corresponds to the MSB of the *shift-register* output. This can be handled by simply reordering the pixels using an *unpack* and a *pack* node (as shown in Figure 3.20(c)). Note that the pack and unpack

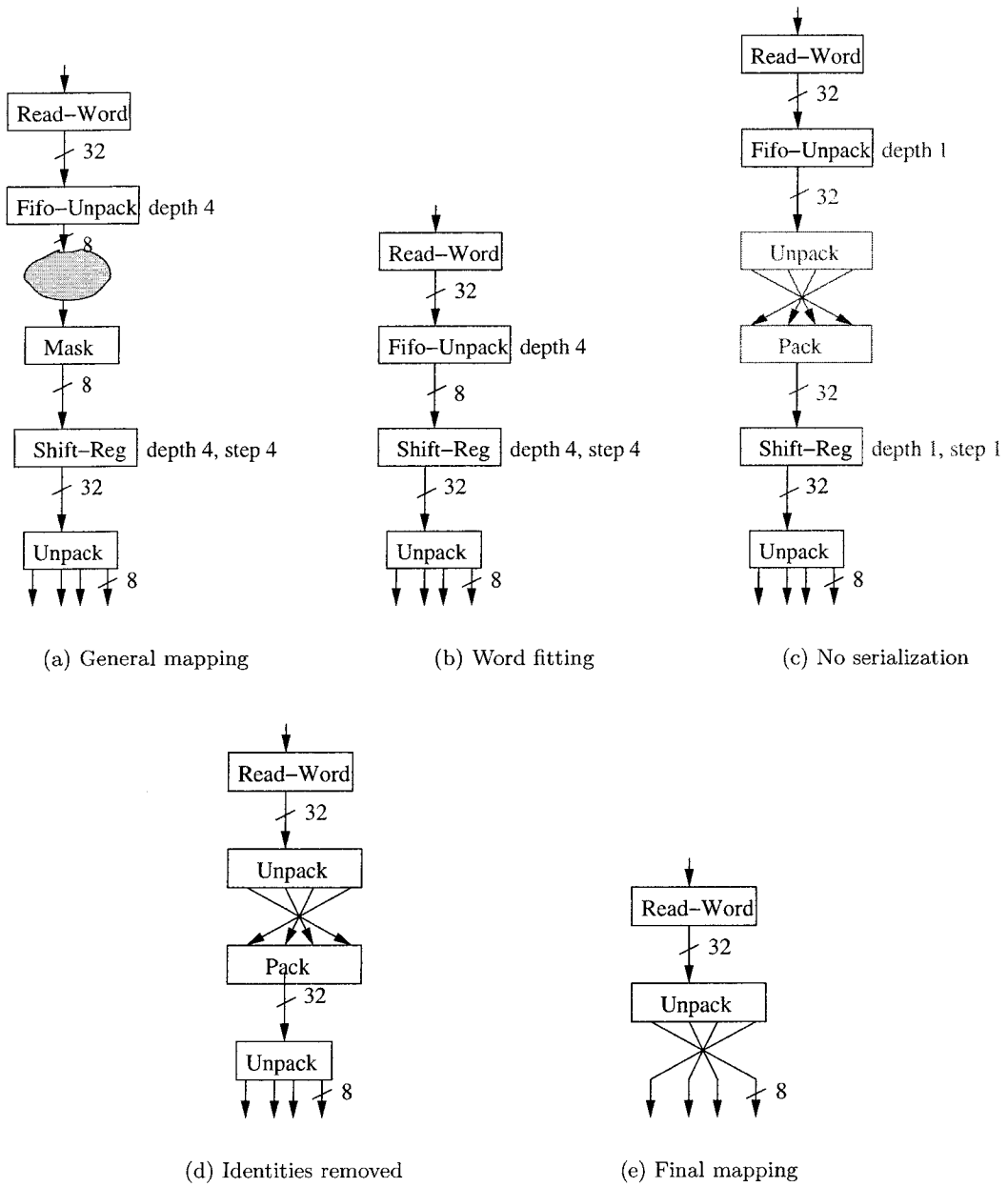



Figure 3.20: AHAHA sub-graph for partially unrolled add-scalar program

nodes add no overhead to the FPGA circuit size because they simply correspond to wire selections.

Other compiler optimizations can simplify the AHAHA graph even more. The *identities* optimizations will remove the *fifo-unpack* with depth one and the *shift-register* with

depth and step equal to one. The resulting graph is shown in Figure 3.20(d). The peep-hole optimizer will then identify the pack node (4 in, 1 out) followed by an unpack node (1 in, 4 out) and simply reconnect the edges in the AHAHA graph to produce the graph shown in Figure 3.20(e). Thus the final graph removes the serialization of pixels there by providing performance gain.

Table 3.1 shows the changes in circuit size and amount of execution cycles (at the AHAHA graph level) for different amounts of partial unrolling of the *add-scalar* SA-C* code. One can see that the number of adders in the resultant AHAHA graph grow with the amount of unrolling. This is because of replication of the loop-body. Figure 3.21 plots the amount of unroll versus the number of AHAHA simulation cycles taken to find the result. It can be noted that partial unrolling in the innermost dimension provides benefits to some extent. Partial unrolling until the window hits the first word-boundary provides the most benefit. In *add-scalar*, the pixels were 8-bits wide, that is, four pixels per word. Hence partially unrolling of two iterations as well as four iterations provided benefits. Unrolling beyond the word boundary did not provide any benefit because the bottle-neck of these programs was IO (memory reads and writes) bound. The slight increase in the total clock cycles for unrolling beyond word boundaries corresponds to the latency overhead of grouping of pixels beyond word boundaries and then serializing these larger groups into words for writes.

| Unroll | Adders | Time |
|---|---|---|
| 1 | 1 | 46 |
| 2 | 2 | 30 |
| 4 | 4 | 19 |
| 8 | 8 | 23 |
| 16 | 16 | 25 |

Table 3.1: Circuit size, time comparison for partially unrolled add-scalar

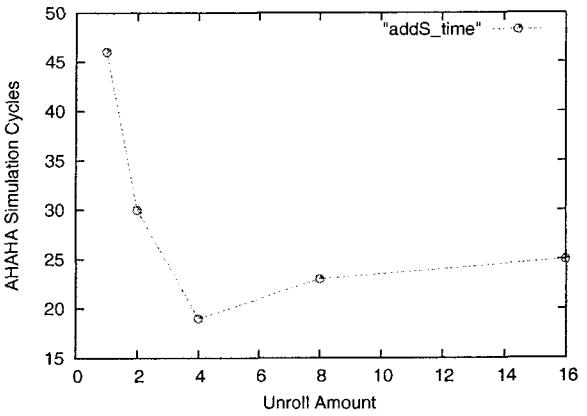



Figure 3.21: Unroll amount versus time for add-scalar in AHAHA

# Chapter 4

# Streams and Processes in SA-C*

## 4.1 Motivation

Nested loops and strict arrays in SA-C* keep the compiler analysis and hardware implementation simple, but introduce storage space overhead and latency problems. For example, in the presence of strict data structures, all input data (even large data sets) needs to be read in completely before any computation can begin, thus generating storage space overhead. Also, the next computation stage can start only after all of its input is available; it cannot start with partial availability of input data.

The SA-C* language introduces a streaming mechanism for expressing a system in which autonomous processes can coexist and interact with each other in a Kahn process network communicating via channels (streams). There are a number of potential benefits to adding the stream data type to the SA-C* language. It introduces non-strict data structures and process concurrency to the language. It also decreases the space and latency overhead in applications like Fast Fourier Transforms.

Consider a size-8 FFT computation, written in SA-C*, using the Pease Algorithm [31]. Figure 4.1(a) shows the FFT-8 computation using FFT-2 (with variable roots-of-unity) as the basic computation block. For the Pease algorithm, the reordering is a simple odd-even shuffle. In SA-C*, this computation can be written using nested loops and strict arrays. The outer loop corresponds to the three phases required and the inner loop corresponds to the reordering and the four FFT-2s needed per phase. Each FFT-2 will perform two reads of complex numbers (from locations corresponding to the

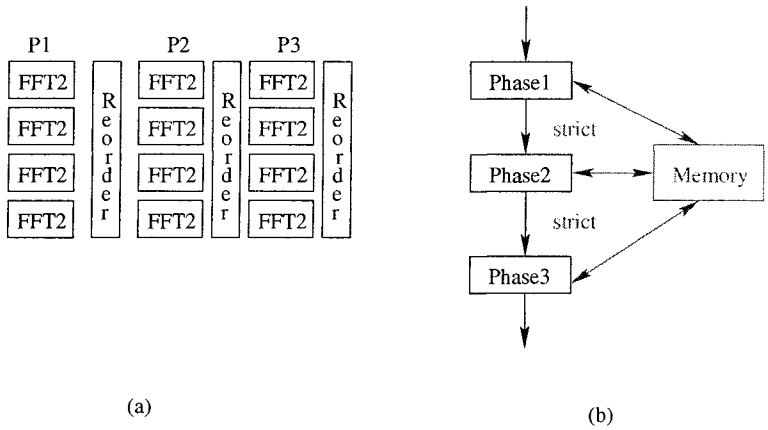



Figure 4.1: Tiling of FFT-8 using FFT-2s, strict arrays (a) tiling (b) execution model.

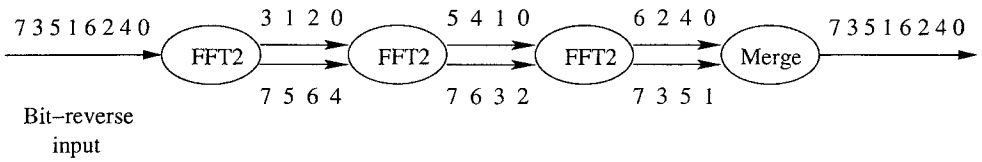



Figure 4.2: FFT-8 using three FFT-2 processes and streams

reordering), one butterfly computation and two writes of complex numbers. Strict arrays will cause the eight values computed by *phase1* (and needed by *phase2*) to be written to (and read back from) the memory (causing IO overhead). Also with strict arrays, *phase2* cannot begin until *phase1* has been completed even though partial data required to start *phase2* is available sooner. This analysis also holds between *phase2* and *phase3*. Similarly, *phase1* for the next FFT-8 cannot begin until *phase3* of the current FFT-8 is complete.

In the presence of streams and concurrent processes, one can have three FFT-2 processes (one for each phase) communicating with streams (as shown in Figure 4.2). This is an adaptation of the Pease FFT algorithm [31]. Each FFT-2 block reads a pair of values from one of its input streams, performs the butterfly computation and splits the results on its output streams. The first four values are read from the top stream while the next four are read from the bottom stream. Thus each FFT-2 process can start its computation when the previous FFT-2 has written three values (two on its top stream and one on its bottom stream). Thus there is a latency of three results per phase.

## 4.2 Streams and Processes in SA-C* Language

Streams and processes introduce high level, process concurrency in SA-C*. This decreases the algorithm latency. Also, an infinite set of image frames from a camera cannot be represented in SA-C* using strict data structures. One can view this input as a 3D array but it would require us to establish the number of frames in advance. Also the amount of memory required to store this 3D array would be very large (and dependent on the number of frames). Streams allow loading only a small section of the image sequence at a time, thus decreasing the storage space overhead for large problems.

They also introduce the capability of expressing interactive systems. The only cyclic dependency allowed in SA-C* with strict data structures is the loop-carried dependency of induction variables. This makes designing and coding of interactive systems, using only strict data-structures, difficult. Consider a system of processes that interact (potentially) forever; process P1 produces output that triggers process P2, and process P2 produces output that triggers process P1. In the presence of streams, this simply involves declaring the two concurrent processes and making the cyclic interconnections among its ports.

Streams also introduce the capability of conditional activation of processes. Consider a system where process P1 receives a stream of fingerprints and performs an initial scan to determine their relevance (say acceptability of the fingerprint). Only the relevant fingerprints are transmitted to a process P2 that performs a more detailed scan (say perform a match to a database). Hence process P2 is conditionally activated only for acceptable fingerprints.

### 4.2.1 Streams in SA-C*

A SA-C* stream has rank-one, and carries the history of its components in time (behaving like a FIFO queue). It follows the Kahn model semantics. Hence checking of presence/absence of data on a SA-C* stream is not allowed. The difference between a SA-C* stream and a Kahn model stream is that in SA-C* a stream is represented as a fixed-size FIFO buffer (versus an unlimited size FIFO buffer in the Kahn model).

The components of a stream can be of a single, strict SA-C* data type (like uint8, int32[:,:], fix16.5[2,3]). Each stream has a single producer process and a single consumer process. (Note that multiple consumers of a stream in SA-C* can be handled easily by creating multiple copies of the given stream). In SA-C*, streams are denoted using the keyword *stream*. Example declarations are shown below.

```
stream uint8 s            // stream of uint8s
stream int32 s[8]         // stream of size-8 1D array of int32s
stream fix25.11 s[:,:]    // stream of 2D arrays of fix25.11s
```

The termination of a stream is indicated by the presence of a special *end of stream (EOS)* token at the end of the components of the stream. A consumer process can check whether the stream has already terminated or not (that is, is the next component available on the stream an *EOS* or not) by simply type-casting the stream variable into a boolean type.

The components of a streams can be accessed by the consumer process using functions *peek* and *get*. *peek(S)* and *get(S)* return the value of the next component on the stream *S*; *get(S)* dequeues that component from the stream while *peek* leaves the stream unmodified. Both *peek* and *get* operations performed on an already terminated stream result in run-time errors. If no value is present on a non-terminated input stream, the *get/peek* calls block until a component arrives on the stream. Also, testing for the availability of a component on a stream is not allowed in SA-C*. For example, a SA-C* process cannot wait for data to arrive on one of its two input streams. This decision excludes non-determinism and time-dependent behavior.

A producer process can append components to a stream using the keyword *put*. *put(<expr>, S)* sends the result of the evaluation of the expression *expr* on stream *S*. A stream is closed (or terminated) when its producer process completes its execution. In SA-C*, the programmer can also explicitly close a stream before completion of the process by using the keyword *close*. Executing the *put* operation on an already terminated stream results in a run-time error.

In SA-C* (prior to introducing streams), there were three type of statements: print, assert and assignment. Streams introduce *get, peek, put* and *close* statements. In SA-C* (without streams), loops and conditionals were considered to be expressions that returned values and only occurred on the right-hand side of an assignment. In the presence of streams, loops (for and while) and conditionals (if-then-else and switch-case) can perform a *put* operation and need not return a value. Hence in certain situations, they can be considered as statements (with no return values).

### 4.2.2 SA-C* Processes

A SA-C* process is a computational unit that consumes (operates on) one or more input streams producing one or more output streams. The number and types of streams in a SA-C* process is fixed (similar to the number of arguments and return values in a SA-C function). A process declaration can also be parameterized in SA-C* (using the keyword *param*) to represent a family of processes. A process declaration has the following template and is similar to the Kahn process template [29].

```
process <name> (in <stream-type-var-list>, out <stream-type-var-list>,
                /* Optional parameter list */
                param <type-var-list>) {
    <process-body>
};
```

### 4.2.3 SA-C* Process Networks

A SA-C* process network is made up of a number of processes that need to run in parallel. It is defined by instantiating SA-C* processes and making interconnections among them. Also, it is a compile-time fixed network and cannot be reconfigured at run-time. The process network declaration follows the following template:

```
process network main(in  <stream-list> /*optional*/,
                     out <stream-list>) {
    <optional-stream-list-declaration for intermediate streams>
```

```
    <optional-variable-list-declaration>
    <process-instantiation-statements>
};
```

The process network is made up of the instantiated SA-C* processes as well as the input and output processes that interact with the environment (e.g. camera, file-system). The SA-C* process network can have zero or more input streams but must have one or more output streams. The input processes are the ones that trigger the execution of the process network by placing components on the input buffers. When the network has no input stream, it is considered to be self-starting, that is, it does not need a trigger from the environment/outside world. In such networks, one (or more) of the instantiated process(es) must trigger the execution of the network by producing some components of its streams before going into a steady-state of computation.

### 4.2.4 Buffer Sizes for SA-C* Streams

In SA-C*, a stream is considered to be a FIFO queue. An unbounded queue size requires run-time mallocs on the hardware. Since all mallocs are done by the host processor in the SA-C* hardware model, a SA-C* stream must have a compile-time known buffer size. The SA-C* process model allows circularity as well as conditional *gets* and *puts* on streams. This makes the rate of production and consumption of streams, and hence their maximal size, impossible for the compiler to predict. Hence, the SA-C* programmer must have the capability to define the buffer sizes required to avoid deadlocks. The default buffer size in the SA-C* model is sixteen. The programmer can override this default buffer size for each stream.

Care must be taken when deciding the buffer sizes because fixed sized buffers can potentially introduce deadlocks. Also, since each stream has a producer and a consumer process, there are two locations where the buffer size can be specified. If the buffer size is specified at both locations, the compiler will perform a sanity check to confirm that the specified sizes are equal. Also, if the size is not specified at either location, then the default size of sixteen will be assumed.

### 4.2.5 SA-C* Stream Example

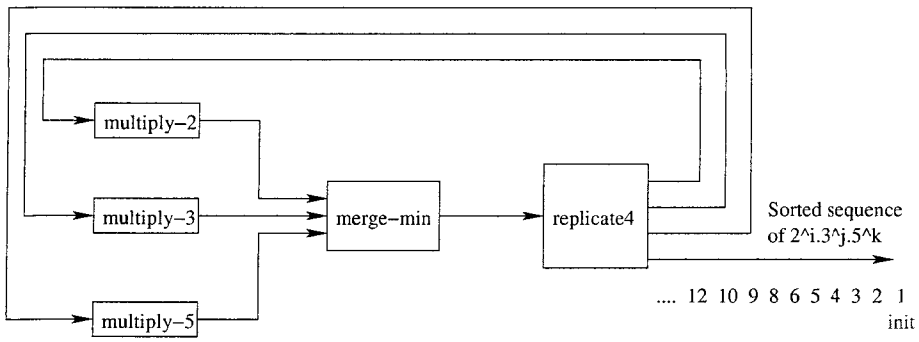



Figure 4.3: Algorithm to calculate powers of 2, 3 and 5 and their multiplications

Consider an algorithm to calculate a sorted sequence of values $2^i.3^j.5^k$. Figure 4.3 shows a computation network to evaluate the required sequence. It requires three multiplication processes, a merge process and a replicator process (to handle single-consumer per stream) model. The SA-C* multiplication process can be written as a parameterized process that multiplies by a factor $n$ (as shown below).

```
process multiply (in stream uint8 inS, out stream uint8 outS,
                  param uint8 n) {
    while (inS) {    // Repeat an action on a non-terminated stream
        put (get(inS) * n, outS);
    };
    close (outS);        // optional statement
};
```

This parameterized *multiply* process can be used in the $2^i.3^j.5^k$ algorithm. The SA-C* process network for evaluating the sorted sequence can be written as follows:

```
process network main (out stream uint32 powers)
{
   stream uint32 multIn0; stream uint32 multIn1;
   stream uint32 multIn2; stream uint32 multOut0;
```

```
    stream uint32 multOut1; stream uint32 multOut2; stream uint32 minS;
    instantiate multiply (multIn0, multOut0, 2);
    instantiate multiply (multIn1, multOut1, 3);
    instantiate multiply (multIn2, multOut2, 5);
    instantiate merge_min (multOut0, multOut1, multOut2, minS);
    instantiate replicate4 (minS, multIn0, multIn1, multIn2, powers);
};
```

Since the $2^i.3^j.5^k$ process network does not receive an input stream from the environment, the *merge_min* process triggers the system by *putting* a 1 on *minS* stream before entering into steady-state of merging the minimum values. The SA-C* code for all the processes in this algorithm can be found in Appendix A.

## 4.3 Stream and Process Mapping in SA-C* Compiler

### 4.3.1 Dependencies in Stream Operations

In the absence of non-strict data structures in SA-C*, the order of evaluation of variables can be determined by simple data dependence analysis. Since SA-C* is a single assignment language, there is only one definition of a strict variable applicable for every use; hence only true dependencies exists for strict data structures. Once streams are introduced in SA-C*, the assumption about strictness as well as the property of monolithic evaluation do not hold.

The order of evaluation of stream operations depends on their textual order in the SA-C* program. In order to keep the internal data-dependence model unchanged, the SA-C* compiler introduces explicit data dependences (in the form of boolean, trigger flags) among the operations on each stream. Stream operations like *get* and *put* modify the stream on which they operate and hence they can be considered in the same way as writes to a memory location. Operations like *peek* and the test to check whether the stream is closed or not, do not modify the stream and can be considered like reads to a memory location. Hence the compiler can identify the true dependences, anti-

dependences, input dependences and the output dependences [1] [33] on each stream variable and introduce explicit data dependencies (in the form of boolean, trigger flags) among these operations.

For example, consider the following SA-C* code that finds the difference between a pair of components arriving on an input stream (assuming there are even number of components).

```
while (inS) {
  put( get(inS) - get(inS), outS );
};
```

The compiler serializes the stream operations by using their textual ordering and by introducing dependencies among them. It converts the above SA-C* code into the DDCF graph shown in Figure 4.4.

### 4.3.2 Implementation of Streams and Processes

The SA-C* process network is made up of input processes (obtaining data from the world), intermediate processes (performing some computation on the data streams) and output processes (providing results to the world). The intermediate processes can be executed on the host processor, the simulator coprocessor, or the hardware coprocessor but the input and output processes are always executed on the host processor. The input and output processes currently communicate with the world using the file-system. Each input and output process is associated with its file; an input process reads the input stream from its file while the output process stores its result stream in a file.

The host implementation of processes requires mapping of the concurrent modules and streams to C-code. The concurrent processes are handled using POSIX threads with each process mapped to a thread. The stream communication is implemented using fixed-size buffers that are shared by the consumer and producer process threads; buffer access is handled using mutex locks. The mutex locks allow exactly one process to have access to a buffer at a given time.

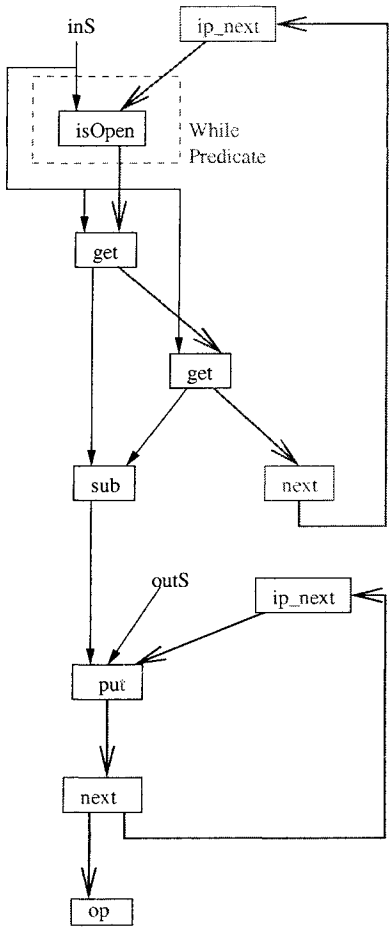



Figure 4.4: Dependences in stream operations.

Since the target hardware model already follows a parallel execution model, mapping of processes does not involve any changes to the target hardware model. Translating the stream communication to the hardware model does require changes and extensions to the model. Each SA-C* stream (mapped on the coprocessor) has a centralized handling unit that manages the FIFO buffering of the stream components. Hence, each stream operation needs to communicate with its handling unit. This is achieved by associating a unique *id* with each stream. The stream handler receives this *id* along with the buffer size and information about the data-type being buffered. The *id* is also provided to all the nodes operating on the stream. These operator nodes can then send their requests to the associated stream handler. (Note that no arbitration is needed among the multiple

stream operator nodes; this is because the handler receives exactly one request at a time because of the serialization among the operator nodes).

Changes are also required in the host-coprocessor interface code to handle the stream communication between the process(es) mapped on the host and the process(es) mapped on the coprocessor. A separate POSIX thread is created (per coprocessor) to handle all the stream communications between the host and that coprocessor. This interface thread handles the interrupts from the coprocessor, the transfers of input streams (when data is available) to the coprocessor and the transfers of output streams from the coprocessor (when an interrupt is received).

#### 4.3.2.1 Stream Interface with Coprocessor

The interface process needs to handle the stream communications with the coprocessor as well as handle the interrupts from the coprocessor. These tasks are handled by the interface process in the order shown below.

1. Load coprocessor configuration and reset the coprocessor.
2. Wait until some data (components of input streams) available for transfer to the coprocessor. Transfer the available stream components.
3. Start the coprocessor.
4. Wait for interrupt from the coprocessor.
5. Transfer results from the output streams of the coprocessor if space available on the host processor buffers. If no space is available on the host processor for any transfer, sleep until a consumer process of these streams creates space and wakes up the coprocessor thread (producer thread for that stream). Once space is available and transfer is complete, continue.
6. Test and check whether all the processes mapped on the coprocessor are done. If done, goto step 9.

7. If the coprocessor is not yet done, transfer components of input streams (if data available on the host processor for transfer and space available on the coprocessor to receive the transfer).

8. Go back to step 4.

9. Read all of the components of output streams back from the coprocessor.

10. Stop the coprocessor and unload its configuration.

#### 4.3.2.2 Scalar Streams at AHAHA Level

A scalar stream follows the basic principle of having a central handler with stream operations communicating with it. Each scalar stream handler receives the stream *id* along with the buffer *depth* and *bit-width* for the scalar components. The handler then creates a buffering mechanism that can hold *depth* number of elements, each of *bid-width* size. There are three types of scalar stream handlers that a coprocessor needs: an input stream, a local stream and an output stream handler. The input stream provides communication from the host to the coprocessor while the output stream provides communication from the coprocessor to the host. Both the producer and consumer process for a local stream are mapped on the coprocessor.

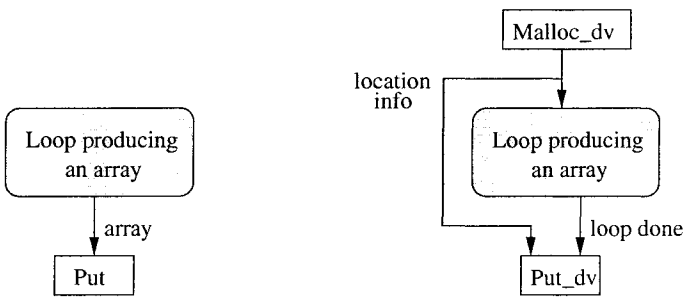
#### 4.3.2.3 Array Streams at AHAHA Level

Mapping of an array stream in AHAHA is different from that of scalar streams. The hardware model does not allow run-time mallocs. Hence, the hardware model cannot allocate a new location in memory (or BRAM) at run-time for every array component being generated or used on the hardware. Instead, the hardware model manages a pool of already malloced locations and cycles through (reuses) these locations multiple times. To be able to malloc a pool of locations in advance, the SA-C* compiler puts restrictions on the types of array streams that can be mapped to the coprocessor. The compiler only allows mapping of stream of arrays where the size of the components is known at compile-time; streams of variable-sized arrays are not mapped to the hardware model.

The host processor calculates the amount of location space required to hold an array of compile-time known size. Since the buffer depth is also known at compile time, the host processor mallocs enough space to hold *depth* number of arrays before starting any processes mapped on the coprocessor.

The coprocessor uses this pool of pre-allocated locations when generating arrays that are the components of an array stream. The malloced locations can be in one of the following three states: available (to store a new array), being used (during creation of an array) or waiting for consumption (of the generated array). The array stream handler on the coprocessor manages the different states that a malloced location can be in.

The array stream handler manages the pool of descriptors (dope-vectors) for the pre-allocated locations by using two internal buffers (instead of one in case of scalar streams); the first buffer *locs_available* keeps track of locations available for storing new arrays while the second buffer *locs_to_consume* keeps track of locations that already contain an array that can be consumed. Initially, all the allocated locations are available for storing new arrays. A loop, producing the array that needs to be appended to the stream, requests a location from the *locs_available* buffer (of corresponding array stream handler). The loop uses this allocated location to store the new array being created. Once the array has been created, the location of the array is appended to the *locs_to_consume* buffer. Figure 4.5 shows the conversion of the *put* operation over an array stream.



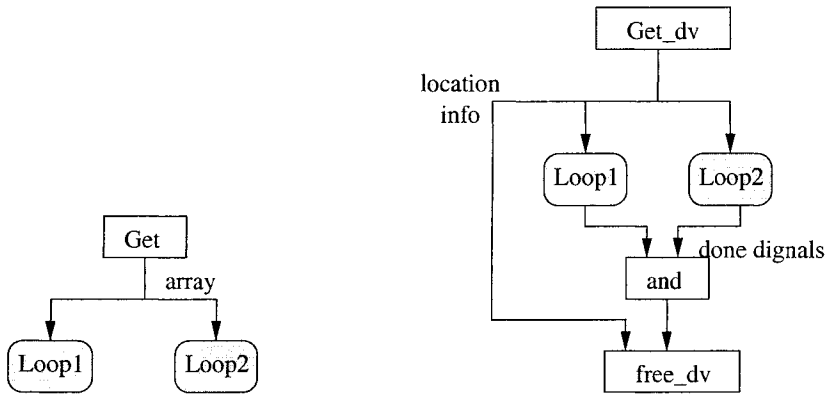

(a) DDCF (b) AHAHA

Figure 4.5: Put operation on array streams in SA-C*

The *malloc_dv* operation requests the next available location from the *locs_available* buffer of the stream handler. The *put_dv* operation appends this location into the *locs_to_consume* buffer only after the producer loop has completed the production of the array. The *malloc_dv* operations need to be serialized to prevent too many malloc requests arriving (in case of multiple executions of loop bodies). This will prevent deadlocks from occurring in case of too many malloc requests. Since the *malloc_dv* and *put_dv* operations communicate with different buffers of the stream handler, the operations can be serialized independent of each other. The introduction of a separate serialization flag for the mallocs can improve the performance of the translated graph.

A get operation on a stream of arrays involves obtaining the location from the *locs_to_consume* buffer. Once the array (whose location was dequeued from the buffer) has been completely consumed, its location can be recycled for next use. Figure 4.6 shows the conversion of the *get* operation over a stream of arrays. The *get_dv* node obtains the location information of the array to be consumed while the *free_dv* node recycles the location for its next use. An array can have multiple consumers in the SA-C* model. Hence the *done* signal from the multiple consumers must be grouped together before being sent to the *free_dv* node. This is not required for the put operation because each array has exactly one producer. The *free_dv* operations do not need any serialization as they are executed exactly when a consumed array needs to be freed; also the order of freeing does not matter as this does not cause a possibility of deadlock.

In the presence of peek operations, the consumers of an array component can be the targets of a peek operation and the targets of the next get operation that is executed on the stream. This implicit aliasing ,along with conditionals and loops, make it difficult (if not impossible) to identify all the consumers of an array component at compile time. Hence, the compiler cannot identify the location for the *free_dv* operation on an array component (especially one obtained by a peek operation). Hence, the compiler does not allow peek operations on a stream of arrays to be mapped on to the hardware model; they are executed on the host processor.




(a) DDCF (b) AHAHA

Figure 4.6: Get operation on array streams in SA-C*

The streams with both the producer and consumer processes mapped on the coprocessor have all their operations and the handler mapped completely on the coprocessor. Care must be taken when handling the input and output streams of the sub-network mapped on the coprocessor. The stream handlers for the IO streams for the coprocessor-mapped sub-network are always mapped on the coprocessor. The host processor communicates with these stream handlers to handle transfers of the IO streams. The transfer of input streams requires the host to identify how many arrays it can transfer; it is the minimum of number of arrays created on the host and how much space is available on the stream handler. The host processor then transfers the arrays one by one in the available locations obtained from the stream handler. The transfer of output streams requires the host to identify the minimum of the amount of buffer space available on the host and the number of arrays already generated on the coprocessor. It then transfers one array at a time; once an array is transferred, the host explicitly requests the stream handler on the coprocessor to recycle the array location (that is, the host explicitly requests a *free_dv*).

## 4.4 Variable Sized Arrays in SA-C* using Streams

Structure building operators for a while-loop produce variable sized (data dependent size) arrays. Since run-time mallocs are not allowed on the coprocessor, such while-loops are executed completely on the host processor. Also, some structure building operators for SA-C* loops allow new sub-structures to be appended (or ignored) based on a mask. Such conditional appending of sub-structure is not mapped on the coprocessor; it is completely executed on the host processor. This prevents codes like run-length encoding from being written in SA-C* loop model.

Streams allow variable sized data-structures and conditional appending to be effectively mapped onto the coprocessor. This is because only a fixed number of results are stored on the coprocessor at a given time, thus eliminating the need for run-time mallocs on the coprocessor. Appendix B shows the SA-C* stream code for run-length encoding.

# Chapter 5

# Algorithms

The algorithms presented in this chapter test the expressibility of the new SA-C* language as well as analyze the benefits (and drawbacks) of mapping codes on FPGAs. Since the mapping of AHAHA graphs on hardware is not yet completed (and outside the scope of this dissertation), the analysis of programs is done at the AHAHA simulator level.

## 5.1 Complexity Measures

The time complexity of SA-C* code is measured in terms of the total number of AHAHA simulation clock cycles required for execution of the program. This measurement differs slightly from the number of clock cycles needed on the hardware. This is because the AHAHA simulation does not take into consideration the time taken to transfer data to and from the host. Also, the simulation of AHAHA nodes does not always mimic the behavior found on the hardware. For example, the simulation does not handle memory contention overhead. Also, the start-up latencies of some nodes in simulation differs from that on the hardware by a cycle or two.

At a higher level, the time complexity of an algorithm can be measured as the number of times each loop body (or process) is executed and the time between each execution. This measure is especially useful in concurrent process as well as pipelined parallelism analysis, as it relates to how often each process operates on new data, thus giving the computation rate.

The critical path of an AHAHA graph is measured in the number of instructions/ operations executed sequentially. The operations can be classified as follows:

1. *NO_OPs*: Some AHAHA nodes are mapped onto hardware as simple shuffling of wires. Hence they provide no overhead in terms of space or time. For example, change-width, pack, unpack, left shift, right shift.

2. *Mathematical Operators*: These AHAHA nodes perform basic arithmetic or logical operations or value comparisons. For example, add, subtract, multiply, negate, and, or, not, less-than, greater-than, equal, not equal.

3. *Control*: The nodes correspond to the control structure of for loops, while loops or conditionals being mapped onto hardware. For example, token generator, counter, address calculator, mask, shift register, fifo pack, fifo unpack, hold.

4. *IO*: These nodes are associated with communication. For example, read word, write word (both to memory as well as block rams), stream get, stream put.

5. *Delay*: These nodes are used to delay the transfer of data across edges in the graph. For example, buffer, pipeline register.

The space complexity of a SA-C* program can be measured in two forms.

1. *Circuit Size*: This is measured in terms of number of instructions at the AHAHA graph level. This is done by simply counting the number of AHAHA nodes in the graph. The nodes are classified in order to understand the graph size better. This is because some nodes (*NO_OPs*) take no space on the hardware while operators like multiply take a lot of circuit space.

2. *Storage*: This measures the amount of space used to store data (input, output as well as intermediate results). This is done by counting the number of buffers used in stream communication and the amount of space needed to store data in memory and block RAMs.

## 5.2 Tridiagonal Solver

A tridiagonal system of linear equations has the form $A.x = b$ where $A$ is a tridiagonal matrix, $x$ and $b$ are vectors (as shown in Figure 5.1). This system can be solved iteratively using the Jacobi method. This algorithm demonstrates the nested loops capability of the SA-C* compiler.

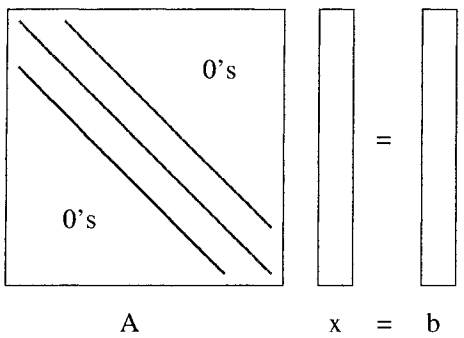



Figure 5.1: Tridiagonal system of linear equations

The Jacobi method [6, 12] examines each of the $n$ equations in the linear system of equations $A.x = b$ independently. Solving equation $i$ for the value $x_i$, while assuming the other entries of $x$ remain fixed, gives us the Jacobi method (as shown in Equation (5.1)). This method is easy to understand and implement but is slow to converge (in comparison to Gauss Seidel).

$$\sum_{j=0}^{n-1} a_{ij} xj = b_i$$

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)}}{a_{ii}} \tag{5.1}$$

In matrix terms, the Jacobi method can be expressed as follows using the $D$, $L$ and $U$ parts (the diagonal, strictly upper triangular and strictly lower triangular respectively).

$$(L + D + U).x = b \tag{5.2}$$

$$D.x^{(k)} = b - (L + U).x^{(k-1)} \tag{5.3}$$

$$x^{(k)} = D^{-1}.b - D^{-1}.(L + U).x^{(k-1)} \tag{5.4}$$

Let the matrix parts be represented as vectors *Dv, Lv, Uv* where *Dv* contains the diagonal elements, *Lv* contains a zero followed by the lower diagonal elements and *Uv* contains the upper diagonal elements followed by zero. The Jacobi solver then reduces to Equation (5.5).

$$x_i^{(k)} = \frac{b_i}{Dv_i} - \left(\frac{Lv_i}{Dv_i} * x_{i-1}^{(k-1)} + \frac{Uv_i}{Dv_i} * x_{i+1}^{(k-1)}\right) \tag{5.5}$$

Each update of $x_i$ (along with the comparison with the previous iteration value) needs to read three elements from array $x$ ($x_{i-1}$, $x_i$ and $x_{i+1}$). A sliding window access over the array avoids multiple reads of the same array element of $x$. When using a sliding window implementation, care must be take to handle boundary cases (where only partial windows are needed). This can be achieved by creating a perimeter (with zeros) around the $x$. Let *Dib, DiL* and *DiU* represent the vectors containing the values $b_i/Dv_i$, $Lv_i/Dv_i$, $Uv_i/Dv_i$. Figure 5.2 shows the pseudo-code for the sliding-window implementation using vectors *Dib, DiL* and *DiU*. Actual SA-C* code can be found in Appendix C.

```
while (largeDeltaExists)
    xperim = {0, x0, ..., xn-1, 0}
    for i in 0 .. n − 1 dot window xwin[3] in xperim
        x[i] = (Dib[i] − (DiL[i] ∗ xwin[0] + DiU[i] ∗ xwin[2]))
        if (abs(x[i] − xwin[1]) > diff_threshold)
            Set largeDeltaExists
```

Figure 5.2: Iterative tridiagonal solver - Algorithm

Since the innermost loop is not unrolled, the AHAHA graph generated is the same irrespective of the size $n$ of the array $x$. Table 5.1 shows the statistics of the different types of AHAHA nodes in the resultant graph for the tridiagonal solver in SA-C*. The two multiplies correspond to the code calculating the new $x_i$. The six add/subtract operators are as follows: one add and one subtract to calculate new $x_i$, one subtract to find the difference from the previous value of $x_i$, one add to keep track of the iteration

count of the solver, one subtract to calculate the index to read $x$ from when calculating *xperim* and one add to calculate the address (start address + index offset) to read from when calculating *xperim*.

| Operator Type | Node Count |
|---|---|
| NoOps | 35 |
| Add/Subtract/Negate | 6 |
| Multipliers | 2 |
| Comparators | 13 |
| Bit-logic ops | 18 |
| Selectors | 6 |
| Loop and data control | 23 |
| IO | 13 |
| Registers | 27 |

Table 5.1: Node classification for tridiagonal solver in AHAHA

Since the total number of iterations the solver needs to complete the computation is data dependent, the total simulation time will vary for different sets of arrays (of the same size) but the time taken for each iteration of the solver will be the same for equal sized arrays. Hence, time analysis is done on the time taken to complete one iteration of the solver for different sized arrays. Figure 5.3 shows the plot of the time taken for each iteration of the solver versus the different sizes of the array $x$. Since each iteration of the solver corresponds to two sequential loops (one calculating the array *xperim* and one calculating the new array $x$), each taking $n + constant$ cycles, each iteration of the solver needs $2n + constant$ cycles to complete. This is verified by the slope of two on the timing plot in Figure 5.3.

Let us now consider SA-C* code where the loop calculating the new values for array $x$ is partially unrolled. Let $x$ be an array of size 8 and let us consider the cases with no unrolling, unrolling by two and four iterations and fully-unrolling the loop. Table 5.2 shows how the number of operators increases with increase in unrolling. One can see that the number of add/subtract nodes and multiply nodes increase by a factor of three and two respectively for each extra unroll done. This directly corresponds to the two multiples, one add and two subtracts needed to compute each new $x_i$. As explained in

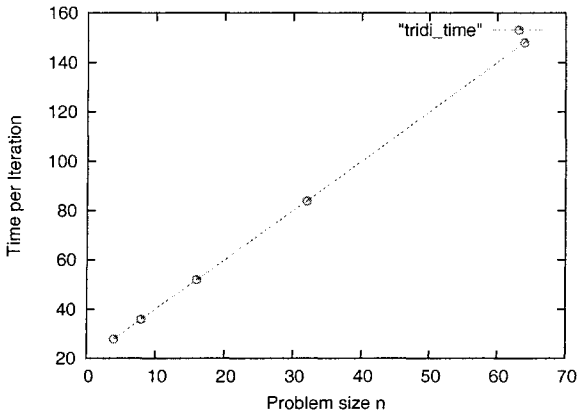



Figure 5.3: Array size versus time for tridiagonal solver in AHAHA

Section 3.4, the partial unrolling in the innermost dimension gives us a time benefit only when the amount of unrolling is less than the amount of data packed per word (for each loop input and loop output). When the unrolling is greater than the amount of packing, the execution time is memory IO bound. Since the data-type used in calculations is *fix16.8*, only two data-items can be packed per memory word. Hence only partial unrolling (by a factor of two) improves the time taken to find the tridiagonal solution. This is also visible from the readings in Table 5.2. The average time taken per tridiagonal solver can also be improved by running multiple solvers in parallel on different FPGA boards.

| Unroll | Add/Subtract | Multipliers | Others | Time |
|---|---|---|---|---|
| 1 | 6 | 2 | 114 | 393 |
| 2 | 9 | 4 | 124 | 335 |
| 4 | 15 | 8 | 152 | 352 |
| 8 | 27 | 16 | 288 | 468 |

Table 5.2: Circuit size, time comparison for partially unrolled tridiagonal solver

## 5.3 K-Means Clustering

The K-Means algorithm demonstrates nested loops and aggregates of loops (with different bounds) in the SA-C* compiler. This algorithm [27] clusters (or partitions) $N$ data

points into $K$ disjoint subsets such that each point belongs to the cluster with the nearest centroid. The centroid is defined as the average of all the points in the cluster. The distance to the centroid can be calculated as the Manhattan's distance, the sum-of-squares criterion or weighted distance. Equation 5.6 shows the identification of the closest cluster for the $n$-th data point (using weighted distance) with vector $v_n$ representing the $n$-th data point, vector $w$ representing the weights, vector $\mu_j$ representing the $j$-th centroid and $\bigotimes$ returning the index $j$ corresponding to the minimum, calculated dot product.

$$\bigotimes_{j=0}^{K-1}(v_n - \mu_j).w \tag{5.6}$$

K-Means clustering is a simple, iterative, re-estimation procedure because the clusters depend on the centroids and the centroids depend on the clusters. The algorithm is as follows:

1. Choose the total number of clusters and assign each point randomly to a cluster.
2. Compute the centroid for each cluster: the mean of all the points in that cluster.
3. For each point, assign it to the the cluster with the nearest centroid.
4. Repeat steps 2 and 3 until the algorithm converges.

This K-Means algorithm does not always yield the same result for the same input data set; it depends on the initial assignment of points to the clusters. The main advantages of this algorithm are its simplicity and speed, which allows it to run efficiently on large datasets. It minimizes intra-cluster variance, but does not ensure that the solution given is a global minimum. Depending upon the initial value selections the algorithm may converge to a local minimum versus a global minimum.

In the SA-C* implementation, the data point is selected to be a triple of the pixel value and location in the array ($< val, i, j >$). The weight vector is selected as $< 4, 1, 1 >$ to give the pixel value higher weight over the location. This weight vector minimizes rectangular partitioning of the image. The pseudo-code for this implementation is shown

in Figure 5.4. Actual SA-C* code can be found in Appendix D. (Note that the recalculation of the centroids is done using two loops: one recalculating the sums and the other recalculating the means).

```
for all points p in Image
   Initialize cluster[p] as unknown
Select k equi-spaced points on the diagonal as initial centroids
while (numChanges > threshold)
   numChanges = 0
   for all points p
      Calculate weighted distance from each cluster's centroid
      Select the cluster k' with minimum distance
      if k' != cluster[p]
        increment numChanges
        cluster[p] = k'
   for all clusters
      update/recalculate cluster centroid
```

Figure 5.4: K-Means pseudo-code

The size of the AHAHA graph generated for the SA-C* K-Means code is independent of the image size and cluster size because the inner loops are not unrolled. Table 5.3 shows the node statistics for the SA-C* K-Means algorithm. The first set of the readings (column 2) correspond to the SA-C* code shown in Appendix D. This algorithm can be made more efficient by simply merging the inner loop calculating the sums and the inner loop calculating the means. This decreases the amount of circuit space used, memory used and total time taken by an order of cluster size. The second set of readings (column 3) in Figure 5.3 show the AHAHA node statistics for the more efficient SA-C* code with merged inner loops.

Running a K-Means algorithm written in C (with image size $n = 32$ and number of clusters $k = 16$) on *davis* (a Dell Dimension 8400 with a 3.2GHz P4 and 1Gb RAM) took 0.005152sec. The optimized, SA-C* K-Means code will take an estimated 0.011291 seconds if it is run on an FPGA board at 25MHz (as shown in Table 5.3). (The frequency

estimate of 25 MHz is used because it is the minimum frequency at which the available FPGA boards run). Thus simply merging the inner-loops does not provide enough speed-up.

| Classification Type | Basic code | With merged loops |
|---|---|---|
| Operators | | |
| NoOps | 204 | 164 |
| Add/Subtract/Negate | 105 | 105 |
| Multipliers | 3 | 3 |
| Comparators | 89 | 73 |
| Bit-logic ops | 101 | 77 |
| Selectors | 89 | 81 |
| Loop and data control | 158 | 133 |
| IO | 37 | 21 |
| Registers | 244 | 218 |
| Memory | | |
| Image Array | $n^2$ | $n^2$ |
| Nextified Cluster | $3n^2$ | $3n^2$ |
| Sums | $8k$ | 0 |
| Nextified Means | $9k$ | $9k$ |
| Timing for $n = 32$, $k = 16$ | | |
| Per iteration cycles | 21735 | 21711 |
| Total cycles | 282597 | 282278 |
| Time estimate @ 25MHz | 0.011303 sec | 0.011291 sec |

Table 5.3: Classification for K-Means algorithm in AHAHA

Since the input image and cluster numbers are both one byte each, partial unrolling in the innermost dimension (up to four iterations) will provide some speedup. For horizontal unrolling to be successful, the image rows must fit perfectly at word boundaries. Unrolling in the vertical dimension does not provide any speedup because there is no overlap among the sliding windows in the outer dimension. Hence the execution time remains IO bound. The horizontal unrolling of the loop updating the cluster information is not done using the *part_unroll* pragma. This is because the compiler will replicate the complete, inner loop comparing the current data point with the cluster information. Since the loop control logic takes significant space on the hardware, the unrolling is achieved by modifying the SA-C* code by hand. In the hand unrolled version, only the

loop body doing the comparison with cluster information is replicated and not the loop control itself. Table 5.4 shows the increase in circuit size and the decrease in amount of AHAHA clock cycles taken by hand unrolling in the innermost dimension. Thus, after unrolling, the code running on the FPGA (at 25MHz) beats K-Means code running on a Dell Dimension 8400 with a 3.2GHz P4 and 1Gb RAM.

| Classification Type | No unroll | Unroll 2 | Unroll 4 |
|---|---|---|---|
| Operators | | | |
| NoOps | 164 | 172 | 176 |
| Add/Subtract/Negate | 105 | 177 | 321 |
| Multipliers | 3 | 3 | 3 |
| Comparators | 73 | 93 | 133 |
| Bit-logic ops | 60 | 60 | 60 |
| Selectors | 146 | 215 | 353 |
| Loop and data control | 133 | 135 | 136 |
| IO | 21 | 21 | 21 |
| Registers | 173 | 179 | 191 |
| Total | 879 | 1056 | 1395 |
| Timing for $n = 32$, $k = 16$ | | | |
| Per iteration cycles | 18543 | 9327 | 4717 |
| Total cycles | 241094 | 121286 | 61354 |
| Time estimate @ 25MHz | 0.009643 sec | 0.004851 sec | 0.002454 sec |

Table 5.4: Partial unroll comparison for K-Means algorithm in AHAHA

## 5.4 Fast Fourier Transforms

The Fast Fourier Transforms have been selected to demonstrate the limitations of strict data structures in SA-C* and to show the advantages of introducing streams and concurrent processes in SA-C*.

A Discrete fourier transform (DFT) is used to analyze frequencies contained in the signal sampled in time. They reveal relative strengths of any periodic components in the input signal. DFT maps a sequence of $n$ complex numbers $x_0, x_1, ...., x_{n-1}$ into $y_0, y_1, ..., y_{n-1}$ using Equation 5.7.

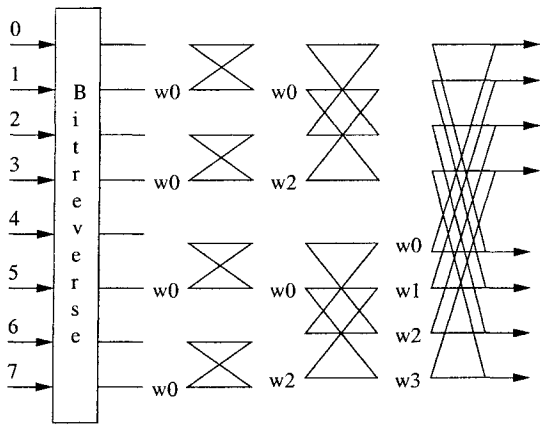
$$y_j = \sum_{k=0}^{n-1} x_k.\omega_n^{jk}, j = 0, ..., n-1 \qquad where \qquad (5.7)$$
$$\omega_n^{jk} = e^{-\frac{2i\pi}{n}jk}$$

Evaluating Equation 5.7 directly takes O($n^2$) operations. A fast fourier transform performs the same computation in O($n.logn$) operations. FFT uses a divide and conquer algorithm that partitions a size-$n$ FFT into a combination of two size-$\frac{n}{2}$ FFTs, some reordering followed by $\frac{n}{2}$ multiplications by roots-of-unity and size-2 FFTs. Figure 4.1 shows the O($n.logn$) structure for FFT algorithm.

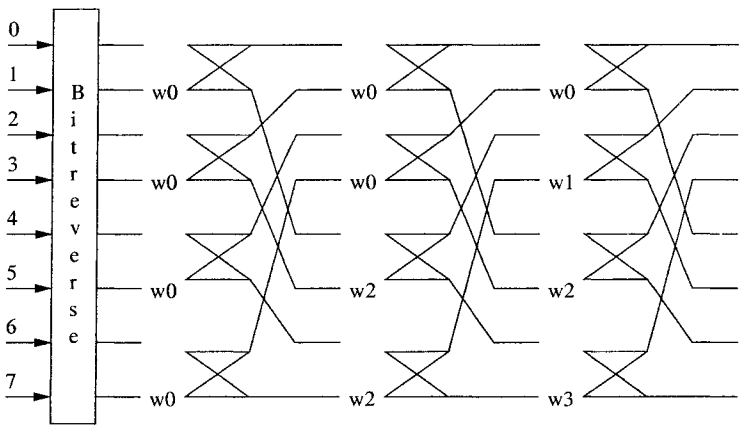
The most common O($n.logn$) FFT algorithm is the Cooley-Tukey algorithm. Figure 5.5(a) shows the computational network for a size-8 FFT using the Cooley-Tukey algorithm. This algorithm is harder to map on an FPGA because the data reordering is different after each FFT stage. Instead, it is easier to map an FFT algorithm with the same reordering after each stage. The Pease FFT algorithm [31] has the same reordering after each FFT stage ; Figure 5.5(b) shows the network for size-8 FFT using the Pease algorithm.

Figure 5.6 shows the pseudo-code for the Pease algorithm. This pseudo-code can be implemented in SA-C* using nested loops and strict arrays. The bit-reverse reordering (at the start of the FFT algorithm) is executed on the host. The host also calculates all the roots-of-unities because sine (and cosine) calculations cannot be mapped efficiently on the FPGA. The coprocessor executes the repetitive parts of the Pease algorithm; the odd-even shuffle along with the $log(n)$ stages of size-2 FFTs and multiplications with roots-of-unity are mapped on the coprocessor.

Column 2 of Table 5.5 shows the AHAHA graph statistics for this SA-C* implementation. The arithmetic operators in the AHAHA graph correspond to the complex math of the root-of-unity multiplication and the FFT-2 computational unit while the rest of the AHAHA nodes correspond to loop control and array storage. The performance of this algorithm can be improved by merging the two inner-loops (the FFT-2 computations and the reordering). This decreases the amount of loop control needed. It also doubles




(a) Cooley-Tukey




(b) Pease

Figure 5.5: Computational network for FFT-8

```
BitRevIn[:] = bitreverse(In);
res[:] = for log(n) stages
    bflyRes[:] = for window W[2] in BitRevIn step(2)
        Compute FFT-2(W[2], variable root-of-unity)
    next BitRevIn =
        Reorder bflyRes using odd-even shuffle
```

Figure 5.6: Pseudo-code for Pease FFT

the performance because the storage of *bflyRes* in memory (or BRAM) is avoided at every FFT stage. Column 3 of Table 5.5 shows the AHAHA statistics of the optimized Pease algorithm.

| Statistic | Pease Algorithm | Improved (Bfly, Reorder Merged) | Improved (FFT-4 block) |
|---|---|---|---|
| AHAHA graph | | | |
| Add/Subtract nodes | 9 | 13 | 49 |
| Multiply nodes | 4 | 4 | 16 |
| Total AHAHA nodes | 266 | 191 | 374 |
| Timing (in AHAHA cycles) for FFT-16 | | | |
| AHAHA cycles / stage | 60 | 28 | 28 |
| Total AHAHA cycles | 266 | 130 | 73 |

Table 5.5: AHAHA statistics for Pease algorithm using strict arrays (basic and optimized version)

The performance of this merged-loop algorithm can be improved by using larger sized FFT blocks as basic computational units (instead of using the smaller FFT-2 blocks). Column 4 of Table 5.5 shows the AHAHA statistics of the Pease algorithm using FFT-4 as the basic block. Since an FFT-4 is made up of four FFT-2 blocks (organized in two phases), the total number of phases decreases by half but the size of the arithmetic calculations increases four-fold. The modified loop control logic handles the larger window-size. The time required for each stage remains unchanged; this is because the amount of data read in (or written) at each stage remains the same. The total time taken for the computation decreases by half because the number of stages decreases by half.

Strict data structures slow down the multi-stage computational network; they force the next phase to start only after the previous phase has finished execution. Using concurrent processes and streams allow the next phase to start earlier (as shown in Figure 5.7 and explained in detail in Section 4.1). For a FFT-8 algorithm, Phase 2 can start as soon as Phase 1 has written 3 values (two on the top stream and one on the bottom stream); the same logic holds for Phase 3. This allows all the phases to be

active at the same time, thus providing performance gain. Also, multiple input data sets (each having 8 elements) can be passing through the process network at the same time. Actual SA-C* code for the FFT algorithm using concurrent processes can be found in Appendix E.

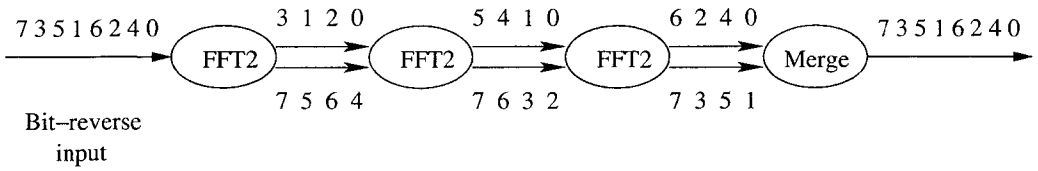



Figure 5.7: FFT-8 using three FFT-2 processes and streams

Table 5.6 compares the Pease algorithm implementations using strict and non-strict data structures for different sized FFT problems. The first implementation looked at is the optimized (merged loop) version of Pease FFT algorithm using strict arrays, nested loops and FFT-2s as the basic computation block. The second implementation is the stream version of the algorithm using concurrent FFT-2 computational units.

| | Strict Arrays | | Streams | |
|---|---|---|---|---|
| FFT Size | Cycles/Stage | Total Cycles | #Butterflies | Cycles/Data Set |
| 8 | 20 | 78 | 3 | 40 |
| 16 | 28 | 130 | 4 | 80 |
| 32 | 44 | 238 | 5 | 160 |
| 64 | 76 | 474 | 6 | 320 |
| N | N+12 | (N+12).log(N)+18 | log(N) | 5.N |

Table 5.6: AHAHA statistics for Pease algorithm - problem size versus time

The AHAHA simulation for the strict arrays algorithm takes O(N) cycles per stage and O(N.log(N)) cycles for the complete FFT. For the FFT implementation using concurrent processes, the number of concurrent FFT-2 modules required is log(N) (number of stages in the algorithm). It takes O(N) time to produce each set of FFT results in steady state, thus providing a time improvement over the implementation using strict arrays. Also, this concurrent module implementation requires O(log(N)) concurrent work with O(N) time. Hence this implementation is optimal.

## 5.5 Neural Networks

A neural network is made up of *units* and *links* [41]. A numeric weight is associated with each link. (These weight are the primary means of storage in a neural network, and learning usually takes place by updating these weights). Each unit has a set of input and output links associated with it and it performs a simple computation. The computation can be split into two components: a linear component that computes the weighted sum of the input values and a non-linear component (called the *activation* function) that produces the results for each output link. (Step functions, sign functions, sigmoid functions are examples of activation functions).

A neural network can be categorized based on the connections among the units. In a *feed-forward* neural network, the links are unidirectional and there are no cycles, that is, the network is a directed, acyclic graph (DAG). In a *layered* feed-forward network, each unit is linked only to units in the next layer; there are no links to the previous layer, to the same layer or links that skip layers. Units in a layered neural network can be categorized as follows: *input*, *output* and *hidden* units. The input units are triggered by the environment and produce the input values. The output units produce the final output results while the hidden units are the internal units and have no direct connection to the outside world.

A neural network can also be trained to learn a function, given enough training samples. Initially, the network links start with random weights. The network weights are then updated to try to make it consistent with the training samples. This is achieved by making small adjustments to the weights in order to minimize the difference between the observed and expected values. The updating process is generally composed of multiple *epochs*; each epoch involves updating all the weights for all the training samples.

The most popular method of updating weight in a multilayer network is called *back-propagation*. The network computes an output vector for a given input vector using these weights. If this computed output vector matches the expected result, nothing is done. If there is an error (a difference between the observed and expected output), the

weights are adjusted to reduce this error. The blame for the error is divided among the contributing weights. Consider a two-layer network. The weight of the link connecting the hidden unit $j$ to output unit $i$ is updated using Equation 5.8 while the weight of the link between the input unit $k$ and the hidden unit $j$ is updated using Equation 5.9. (g' is the derivative of the activation function and $\alpha$ is the learning rate. Care must be taken when selecting $\alpha$ to avoid overshooting).

$$W_{j,i} = W_{j,i} + \alpha * a_j * \Delta_i \qquad where \qquad (5.8)$$

$$\Delta_i = Err_i * g'(in_i) \qquad and$$

$$Err_i = Observed_i - Expected_i$$

$$W_{k,j} = W_{k,j} + \alpha * I_k * \Delta_j \qquad where \qquad (5.9)$$

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i$$

Figure 5.8 shows the pseudo-code for a two-layered, feed-forward neural network (only calculation, non-learning) using a sigmoid as the activation function. This can be implemented in SA-C* using nested loops and strict arrays. The sigmoid function requires exponentiation and division that are difficult to map on the FPGA. For efficient mapping on the FPGA, an approximation of the sigmoid function is used in the SA-C* code. The function is divided into regions with a straight-line representing each region. The SA-C* sigmoid function identifies the straight-line corresponding to the input and then calculates the output using the slope and offset of the identified line. The circuit size of the SA-C* nested loop code is independent of the number of input, output and hidden layers but the amount of storage required for the inputs, outputs and intermediate results depends on all three factors.

The SA-C* compiler does not allow mapping of loops with 1-D row tiling reduction (that produce 2-D arrays) on the FPGA. This prevents mapping of neural networks with multiple output units on the FPGA; only neural networks with a single output unit can be mapped on the coprocessor. This restriction can be easily overcome by creating a streams of 1-D output results.

```
for each input data set
   hres[:] = for each hidden unit
      Perform weighted sum(input)
      Call sigmoid(weighted sum)
   ores[:] = for each output unit
      Perform weighted sum(hres)
      Call sigmoid(weighted sum)
```

Figure 5.8: Neural networks algorithm

Appendix F shows a SA-C* neural network calculating a two-input, one-output XOR function using a two-layered, feed-forward neural network. The number of output units is one (equal to the number of outputs) and the number of hidden units is selected to be five. It uses an approximated sigmoid as the activation function. The loop performing the hidden layer computation takes 22 cycles per input data set while the output layer computation takes 7 cycles per output result. The version written using strict data data structures (for the sequence of input data, output of the hidden layer and final results) requires 25 cycles per input data set in steady state, while the version written using streams requires 22 cycles per input data set in steady state. The stream version allows concurrent execution; the hidden layer can begin computation on the next input data set while the output layer is working on the previous data set. Hence the bottle neck for the stream version is the compute-intensive hidden layer (22 cycles). One would expect the version using strict data structures to take $29(= 22 + 7)$ cycles because strict data structures prevent concurrent execution. Instead, it takes slightly fewer cycles because the strictness comes into play once the hidden layer attempts to write the results of hidden layer units (for the next input data set). This allows the computation of the first hidden layer unit to complete in parallel with the output layer calculation. The writing of this result (and hence the start of computation of the second unit) is delayed until after the output layer has completed its execution and released access to the strict, intermediate result array.

```
Init hidden and output layer weights randomly
for multiple epochs
   for each training set
      hres[:] = forward pass hidden layer(input, hwts)
      ores[:] = forward pass output layer(hres, owts)
      Calculate error as ores - expectedRes
      Update hwts and owts using error
```

Figure 5.9: Learning neural networks algorithm

Figure 5.9 shows the pseudo-code for a two-layered, feed-forward learning neural network. Implementing this in SA-C* requires induction variables (weights) to be updated inside a two-deep nested loop. Since induction variables can be updated in SA-C* only at one level nesting, the learning network cannot be implemented using nested loops and strict data structures. The two-level deep updates can be avoided by using concurrent processes and streams. Figure 5.10 shows the concurrent process model for a learning neural network while Appendix G shows the actual SA-C* code.

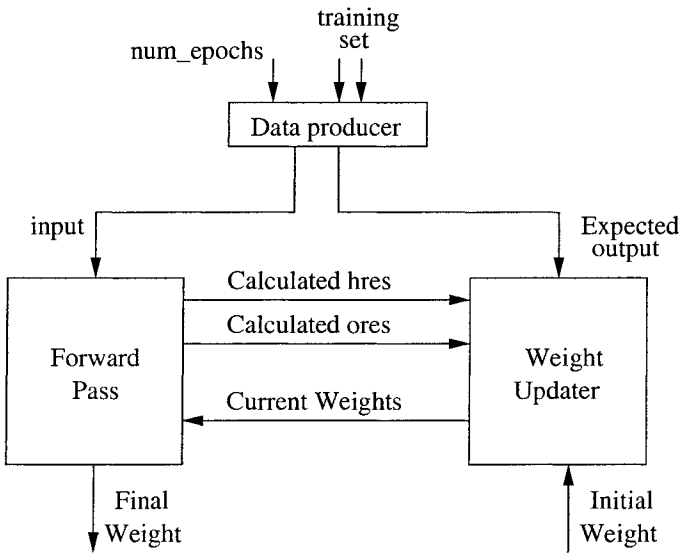



Figure 5.10: Learning neural network using concurrent processes

The learning neural network is inherently sequential; this is because the updated weights are used when the network operates on the next training set. The concurrent

process model has only one process active at a given time; the forward-pass process is idle when the weight-updating process is active and vice versa. The SA-C* implementation takes 100 cycles for every forward pass along with update of the weights.

# Chapter 6

# Related Research

The specification of a circuit mapped on an FPGA can be provided by using a hardware description language (like VHDL or Verilog) or by providing a schematic design. This hardware description can then be synthesized and placed-and-routed for the specific FPGA. In an attempt to reduce the complexity of design and the time-to-market, there have been moves to raise the abstraction level of the design. This can be achieved by designing in high-level languages (like SA-C*, StreamC, HandelC) that hide the hardware details from the programmer. Multiple research groups are also looking into the issues involved in stream-oriented computations and their mappings on hardware.

## 6.1 StreamC

Stream-C[14][15] is a high level language, designed at Los Alamos, that targets FPGA hardware. It is an extended version of C that supports stream-based computations on FPGAs. Stream-C provides capabilities to express a static network of concurrent process, communicating only via stream connections. There are restrictions placed on the components of a stream; a stream can only carry components of small size (one byte to one word). Stream-C follows the Communicating Sequential Processes (CSP) [24] parallel programming model. Communication among concurrent processes (via streams) is non-buffered and synchronized. Hence, to avoid deadlocks in applications like Pease FFT (that require buffering), the programmer must handle the buffering explicitly inside a process.

## 6.2 Occam

The Oxford programming research group is looking at compiling Occam onto FPGAs [32] [36]. Occam (like Stream-C) is based on the Communicating Sequential Processes parallel programming model. Hence the communication occurs over non-buffered, synchronized channels. The stream (channels) can only carry small, fixed-size components. This prevents the language from being able to handle say a stream of image frames from a camera (unless each image is split into a stream of pixels). Occam expressions can be mapped as combinational hardware logic on the FPGA. Side-effect statements like assignments, stream reads and write are made up of hardware logic implementing the statement, along with a *request* signal indicating when to execute the logic block and an *acknowledge* signal indicating when the statement block has completed execution. This target hardware model simplifies mapping of non-deterministic *alt* constructs; the *request* signal for each alternative block is calculated using the boolean condition for each block.

## 6.3 HandelC

The Oxford University Computing Laboratory created the language Handel-C [35][22]. Handel-C is a high level language, based on C, that is designed specifically for hardware. It provides extensions to C (like explicit parallel expressions and variable bit precision) to take advantage of features of hardware. The language extensions that provide expressibility for parallel processes and communications via streams follow the Occam syntax. The target model for Handel-C is also based on the CSP model. Currently Celoxica manages the language and the compiler.

## 6.4 StreaMIT

StreaMIT [44] is a high-level language and compiler designed for stream programming. The goal of the language is to improve the productivity by raising the level of abstraction in stream programming while the goal of the compiler is to perform stream-specific optimizations. It targets the RAW [46] architecture. The current version of the StreamIt

compiler requires the production and consumption rates on the streams be statically known (like synchronous data flow). This enables stream analysis like deadlock detection and buffer overflow, but prevents programming of compression algorithms like run-length encoding. Communication is modeled as a blocked, buffered, send-receive. Also, the process model allows for occasional modification to the stream connections between processes.

## 6.5 Imagine

Imagine [26] is a high-level system used to describe set of kernels (filters/processes) connected by streams. The Imagine language is made up of two parts: KernelC used to describe the kernel processes and StreamC used to describe the interconnections among the kernels. The kernels in Imagine operate independently on each input data set; they are not allowed to carry over results from the previous computation to the next computation. The language targets the Imagine Stream Architecture which is a novel architecture that executes stream-based programs. It provides high performance with 48 floating-point arithmetic units (that make up eight computation units) and an efficient register organization. The KernelC code is compiled by the KernelC scheduler into VLIW instructions. Since each kernel computation is independent, performance gain can be obtained by making the eight VLIW arithmetic clusters perform the SIMD operations on different elements of the input stream simultaneously. The stream buffering is handled using a stream register file and a stream controller. The scheduler handles the buffer allocations and compiles the StreamC code to generate the sequence of stream operations to be given to the stream controller.

## 6.6 Score

Stream Computations Organized for Reconfigurable Execution (SCORE) [42] addresses the issue of application portability among reconfigurable hardware because of the lack of a unifying target model. SCORE developed an abstract, stream-oriented, computa-

tional model to address the issue of mapping a computation efficiently on a wide-range of reconfigurable hardwares. It provides an abstract view of the reconfigurable hardware, which exposes its strengths, while abstracting the actual composition of physical resources available on the hardware. The key concept in this model is that a computation is broken up into compute pages. Compute pages are linked together in a data-flow manner with unbounded FIFO queues (called streams). A run-time OS manager allocates and schedules the compute pages at run-time for both for computations and memory.

## 6.7 Ptolemy

Ptolemy [38] is a tool for modeling, design and simulation of heterogeneous, concurrent systems. In Ptolemy, the concurrent processes and the communication among them do not follow a single model of computation. The key innovation of Ptolemy is that a system can be made of multiple domains (each implementing its own model of computation); some of the models allowed are synchronous data flow, Kahn processes, discrete events, CSP. Ptolemy uses object-oriented software principles to allow heterogeneous models to coexist seamlessly; the OO model also simplifies the addition of new models to the system (thus providing extensibility). Each system is implemented in Java using threads; each concurrent process is mapped to an individual thread. Concurrent accesses to shared resources (processes accessing a thread) is synchronized using monitors.

## 6.8 SA-C* in Context of Related Work

The SA-C* compiler follows different models of concurrent computations in different parts of compiler. The processes in the SA-C* language follow the Kahn model of computation with one restriction; the edges among processes correspond to fixed sized buffers. The Kahn process model allow algorithms (like run-length encoding) with variable rates of production and/or consumption to be expressed in SA-C*. The fixed buffer size restriction is placed because no run-time mallocs are allowed in the target hardware model. The data flow graph in the SA-C* compiler also follows the Kahn model of computation

with each DFG node corresponding to a process and each DFG edge corresponding to an unlimited sized buffer. The abstract hardware architecture in the SA-C* compiler follows the communicating sequential processes (CSP) model; each clocked/synchronized node corresponding to a CSP process with all the edges and combination nodes corresponding to the instantaneous communication. The abstract hardware model handles fixed amount of buffered communication (like synchronous data flow) by inserting explicit buffer nodes on the edges; this moves the buffering inside the AHAHA node while keeping the communication along the edges as instantaneous.

# Chapter 7

# Conclusions, Future Work

## 7.1 Conclusions

The goal of this dissertation was to study the improvement of the expressibility of SA-C* language and the compiler technology required to create performance improvements with respect to execution time and circuit size. The main questions answered in the dissertation are as follows:

1. How can we map problem-size independent, space-efficient circuits onto the FPGA? Mapping of nested loops, aggregate loops and intermediate arrays on the coprocessor provides controllable circuit sizes.

2. How can the SA-C* compiler generate time efficient circuits? Non-strictness (with streams) and concurrent processes in SA-C* provide algorithm speed-up.

3. How can the expressibility of the SA-C* language be improved? Streams and concurrent processes in SA-C* also improve the expressibility.

4. What are trade-offs between space and time efficiency in SA-C* codes? These trade-offs can be obtained using compiler optimizations like loop fusion and partial unrolling.

5. Do we need I and M-structures in the SA-C* language? Most target applications can be expressed using array induction variables and/or concurrent processes. Hence I and M-structures are not added to this versions of the SA-C* language.

### 7.1.1 Space Efficient Circuits

In the old, restrictive compiler, only innermost loops were mapped onto the coprocessor. Hence, in case of some algorithms (like tridiagonal solver), some of the loops needed to be unrolled to allow mapping of the complete algorithm on the coprocessor; this forced the circuit size to be dependent on the input data size. Also, this made mapping of such algorithms for large input data size extremely difficult (and in some cases impossible). Expanding the SA-C* compiler to efficiently map nested loops and intermediate arrays allowed the circuit size for such algorithms to be independent of the input data size, thus removing the restriction on large input data sets.

### 7.1.2 Time Efficient Circuits

The analysis of algorithms (like Fast Fourier Transforms), implemented using nested loops and strict arrays, emphasized the limitations of strict data structures. Hence non-strict data structures (streams) and concurrent processes were introduced in the SA-C* language. Concurrent execution of processes in SA-C* provided task parallelism and speed-up in algorithms like Fast Fourier Transform.

### 7.1.3 Expressibility of SA-C*

Streams not only provided improvement in performance but also increased the types of algorithms that could be efficiently expressed in SA-C*. SA-C* streams can be used to represent variable sized arrays produced by while-loop array reduction, thus allowing mapping of algorithms like run-length encoding on the coprocessor.

The SA-C* hardware model restrictions also prevented loops creating arrays-of-arrays from being mapped on the coprocessor. For example, in algorithms like neural networks, creating a 1-D row in an inner loop and tiling it in an outer loop to produce a 2-D array cannot be mapped onto the coprocessor. This restriction can be overcome by using SA-C* streams and creating a stream of arrays. The SA-C* language also puts restrictions on updating induction variables; they can only be updated one level deep in nesting. Algorithms like learning neural networks require updating weights two levels deep (in

nested loops); hence one cannot write them efficiently using induction variables. This restriction can be overcome by using a cyclic network of processes.

### 7.1.4 Computation Model for SA-C* Processes

One of the questions that required answering was deciding the model of computation for the SA-C* processes. The Kahn model was selected over Synchronous Data flow (SDF) because it expanded the types of applications that could be expressed in SA-C*. Data dependent process networks like the $2^i.3^j.5^k$ algorithm cannot be expressed in an SDF model. Another problem that was presented was that of deadlock detection and scheduling of concurrent processes. Since the model of computation for the SA-C* processes is selected to be the Kahn model (with variable rates of production and consumption), it is not possible to always detect deadlocks and statically determine buffer sizes (like in the restrictive SDF model). The programmer is left with the problem of identifying the required buffer size and deadlock detection.

### 7.1.5 Space versus Time Trade-offs

#### 7.1.5.1 Aggregate Loops

Another question asked was related to the execution speed of aggregate loops with strict arrays. In the presence of strict arrays, the consumer loop must wait until the producer loop has completed generating the whole array. This results in performance slow-down. One approach to removing the slow-down is to use loop-fusion, thus avoiding the intermediate array storage. Another solution is to implement each loop as a concurrent process, thus allowing the consumer loop to execute concurrently. Loop fusion can cause the computation loop body to get large (by replication) if the rates of consumption and production differ. Concurrent processes avoid this replication, but concurrent processes require rewriting of the algorithm in the process model. Since most of the compiler optimizations are focused on loops and arrays, it is better to use loop-fusion if the resultant circuit can fit on the FPGA. In case the circuit gets too large because of replication, a better solution is to rewrite the algorithm using concurrent processes with streams.

#### 7.1.5.2 Partial Unroll in Inner Dimension

During the analysis of nested-loop SA-C* programs, it was noticed that the bottleneck of the innermost dimension of a loop was not based on the memory IO access but was based on the serialization of input pixels or packing of output results. This is because of packing of data in the innermost dimension. During the analysis, it was noticed that an opportunity existed to decrease the time spent by serialization by implementing horizontal unrolling (partial unrolling in the innermost dimension).

#### 7.1.5.3 Pipelining and Induction Variables

The SA-C* compiler pipelines loop bodies to improve the clock frequency of the circuit. Pipelining a SA-C* loop with circularity (generated by induction variables) does improve the clock frequency but slows down the output rate. This is because only one section of the pipeline in the circular circuit is active at a given time. If this pipelined loop is nested inside an outer loop, the final production rate can be improved if the multiple pipeline stages are active simultaneously, with each stage working on a different outer-loop iteration. This can be achieved by rewriting the code using concurrent processes: one concurrent process alternates data from multiple iterations of the outer loop while the other concurrent process implements the pipelined circular loop body. Figure 7.1 shows the concurrent process model for keeping multiple pipelined stages active.

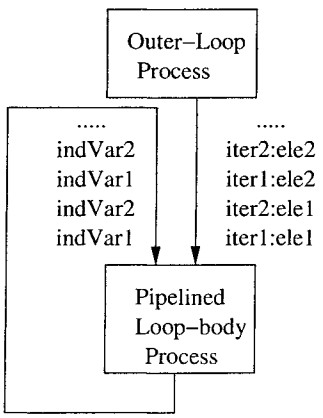



Figure 7.1: Speed-up in pipelined loops with circularity

### 7.1.6 I-Structures and M-Structures

This research attempted to answer the questions related to I-structures and M-structures: Are these data types required in SA-C*? If required, what are the issues involved in mapping them onto the coprocessor? It was realized that all the target applications could be efficiently written using induction variables (nextified arrays) with very little overhead. The overhead included need for three memory locations to represent the nextified array: one for the initial value and two for toggling intermediate results. It was also realized that I or M-structures would have an overhead of presence and defer bits for each location. Also, the calculation of the handshaking for each access of the I or M-structure would have at least the latency of a read for the presence bit as well as the defer bit. Since all target applications could be mapped using nextified arrays and the overhead for I and M-structure handshaking was large, they were not implemented in this version of the SA-C* compiler.

## 7.2 Future Work

The SA-C* compiler currently maps SA-C* programs onto an abstract hardware model. The compiler needs to be extended to complete the final phase of mapping of the abstract hardware model onto the FPGA. (This final phase is part of Charles Ross' dissertation). The abstract hardware model also needs to be made to mimic the actual hardware model as closely as possible by comparing both the execution models.

The SA-C* compiler can map concurrent processes to the abstract hardware model; the compiler can now be extended to automatically generate concurrent processes. For example, the compiler can be extended to automatically convert variable sized array reductions in while-loops to a process with a fixed-size buffer holding partial results at a given time. Also, an optimization can be added to the compiler to automatically convert aggregate loops into producer and consumer processes, thus removing the overhead of strict arrays and allowing concurrent execution of the loops.

Work also needs to be done in the applications domain to test the versatility of the SA-C* streams, concurrent processes and nested loops. Vision applications like face-recognition algorithms and networking applications like TCP/IP need to be written in SA-C*. The issues involved in integrating the SA-C* compiler with industry tools can also be looked at in order to make the compiler more accessible to end-users.

# REFERENCES


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

[2] Alpha Data, San Jose, CA. *Alpha Data Reference Manual.* `www.alpha-data.com`.

[3] Annapolis Micro Systems, Inc., Annapolis, MD. *StarFire Reference Manual.* `www.annapmicro.com`.

[4] Annapolis Micro Systems, Inc., Annapolis, MD. *WildStar Reference Manual.* `www.annapmicro.com`.

[5] Arvind, K. Gostelow, and W. Plouffe. The (preliminary) Id report. Technical report, Univ. of California, Irvine, Dept. of Information and Computer Science, 1978.

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

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

[8] W. Böhm, B. Draper, W. Najjar, J. Hammes, C. Ross, and M. Chawathe. Optmized compilation of embedded applications on FPGAs. In *Fifth Annual High Performance Computing Workshop*, Lincoln Labs., MIT, Nov. 2001.

[9] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping single assignment programming language to reconfigurable systems. *Journal of Supercomputing*, 21:117–130, 2002.

[10] M. Chawathe. Data flow graph to VHDL translation, April 2000.

[11] B. Draper, W. Najjar, W. Böhm, J. Hammes, R. Rinker, C. Ross, and J. Bins. Compiling and optimizing image processing applications for fpgas. In *IEEE International Workshop on Computer Architecture for Machine Perception*, Sept. 2000.

[12] Eric W. Weisstein et al. Jacobi method. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/JacobiMethod.html.

[13] FPGA. Fpga - from wikipedia. Information about FPGAs is available at from Wikipedia, http://en.wikipedia.org/wiki/FPGA.

[14] M. Gokhale. The Stream-C language. http://www.streams-c.lanl.gov.

[15] M. Gokhale. Stream-oriented fpga computing in Stream-C. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

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

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

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

[19] J. Hammes, W. Böhm, C. Ross, M. Chawathe, B. Draper, R. Rinker, and W. Najjar. Loop fusion and temporal common subexpression elimination in window-based loops. In *IPDPS 8th Reconfigurable Architecture Workshop*, April 2001.

[20] J. Hammes, R. Rinker, W. Böhm, W. Najjar, B. Draper, and R. Beveridge. Cameron: High level language compilation for reconfigurable systems. In *Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[21] 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.

[22] Handel-C. Information about Handel-C is available at the project's web site, www.celoxica.com/tech/handel-c/default.asp.

[23] Haskell. Information about Haskell is available at the project's web site, www.haskell.org.

[24] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[25] D. Hyde. *Introduction to Programming language Occam*, 1995. Document available from www.eg.bucknell.edu/~cs366/occam.pdf.

[26] Imagine. Information about Imagine is available at the project's web site, http://cva.stanford.edu/imagine.

[27] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[28] G. Kahn. The semantics of a simple language for parallel processing. In *IFIP Congress*, pages 471–475, 1974.

[29] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, Amsterdam, Holland, 1977.

[30] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of IEEE*, 75(9):1235–1245, 1987.

[31] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform.* SIAM, Philadelphia, PA, 1992.

[32] W. Luk, D. Ferguson, and I. Page. Structured hardware compilation of parallel programs. In *W. Moore, W. Luk, Eds., FPGAs, Abingdon EE&CS Books*, pages 271–283, England, 1991.

[33] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[34] W. Najjar, B. Draper, W. Böhm, and J. R. Beveridge. The Cameron Project: High-level programming of image processing applications on reconfigurable computing machines. In *PACT '98 - Workshop on Reconfigurable Computing*, pages 83–88, Oct. 1998.

[35] O.H.C.Group. OXFORD hardware compiler group, the Handel Language. Technical report, Oxford University, 1997.

[36] I. Page and W. Luk. Compiling occam into fpgas. In *W. Moore, W. Luk, Eds., More FPGAs, Abingdon EE&CS Books*, England, 1994.

[37] P. Patil. Design and software implementation of the SA-C abstract hardware architecture, May 2001.

[38] Ptolemy. Information about Ptolemy is available at the project's web site, `http://ptolemy.eecs.berkeley.edu`.

[39] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 1993.

[40] C. Ross. An abstract hardware architecture for hierarchical dataflow compilation, April 2003. PhD Proposal (in progress).

[41] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[42] SCORE. Information about SCORE is available at the project's web site, `http://brass.cs.berkeley.edu/SCORE`.

[43] R. Stephens. A survey of stream processing. In *Acta Informatica*, pages 491–541, 1997.

[44] StreamIt. Information about StreamIt is available at the project's web site, `www.cag.lcs.mit.edu/streamit/`.

[45] Synplicity, Inc. *Synplify User's Guide, Release 5.2.2*, 1999. `www.synplicity.com`.

[46] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *Computer*, Sept. 1997.

[47] Xilinx, Inc., San Jose, CA. *The Programmable Logic Databook*, 1998. `www.xilinx.com`.

# Appendix A

# $2^i.3^j.5^k$ in SA-C*

```
// MAXVAL is 2^32 because type is uint32
#define MAXVAL 0xffffffff

process multiply (in stream uint32 inS, out stream uint32 outS,
                  param uint32 n) {
    while (inS) {
        uint32 ipval = get(inS);
        if (ipval <= MAXVAL/n) {
            put( ipval * n, outS );
        };
    };
};

process merge_min (in stream uint32 inS2, stream uint32 inS3,
                   stream uint32 inS5, out stream uint32 outS) {
    // Merge-min process starts the cyclic network
    put(1, outS);

    while (inS2 || inS3 || inS5)
    {
        uint32 val2 = (inS2)? peek(inS2): MAXVAL;
        uint32 val3 = (inS3)? peek(inS3): MAXVAL;
        uint32 val5 = (inS5)? peek(inS5): MAXVAL;
        uint32 vals[3] = {val2, val3, val5};
        uint32 minval = array_min(vals);

        if (inS2 && (val2 == minval)) { get(inS2); };
        if (inS3 && (val3 == minval)) { get(inS3); };
        if (inS5 && (val5 == minval)) { get(inS5); };

        put(minval, outS);
    };
};
```

```
process replicate4(in  stream uint32 inS,
                   out stream uint32 outS2, stream uint32 outS3,
                       stream uint32 outS5, stream uint32 outS) {
    while (inS) {
        uint32 copyval = get(inS);
        put(copyval, outS2);
        put(copyval, outS3);
        put(copyval, outS5);
        put(copyval, outS);
    };
};

process network main (out stream uint32 powers) {
    stream[3] uint32 multIn;
    stream[3] uint32 multOut;
    stream uint32 minS;

    uint8 factors[3] = {2, 3, 5};

    for uint8 i in [3] {
        instantiate multiply (multIn[i], multOut[i], factor[i]);
    };

    instantiate merge_min (multOut[0], multOut[1], multOut[2], minS);
    instantiate replicate4 (minS, multIn[0], multIn[1],
                            multIn[2], powers);
};
```

# Appendix B

# Run Length Encoding in SA-C*

```
export main;

process rle(in stream uint8 inpixS,
            out stream uint8 outpixS, stream uint8 cntS) {
  if (inpixS) {
    uint8 pixel, uint8 cnt = get (inpixS), 1;

    uint8 finalpixel, uint8 finalcnt = while (inpixS) {
      uint8 nextpixel = get (inpixS);

      put (pixel, pixel != nextpixel, outpixS);
      put (cnt, pixel != nextpixel, cntS);

      next pixel = nextpixel;
      next cnt = (pixel==nextpixel)?cnt+1:1;
    } return(final(pixel), final(cnt));

    put (finalpixel, outpixS);
    put (finalcnt, cntS);
  };
};

process network main(in stream uint8 inpixS,
            out stream uint8 outpixS, stream uint8 cntS) {
  hardware() {
    instantiate rle(inpixS, outpixS, cntS);
  };
};
```

# Appendix C

# Tridiagonal Solver in SA-C*

```
// Iterative TriDi solver A.X = B (using Jacobi method)
// Given A and X, compute B, then solve A.Y = B
// A occurs as a 2D array of three diagonals L D U in its columns
// with a zero in front of L and after U to equalize lengths
// The iterative part performs computation in fix16.8
#define SIZE      4
#define CALC_TYPE fix16.8
#define MAX_ITER  32
#define MAX_DELTA (CALC_TYPE)0.0001

CALC_TYPE[:] main (float A[:,:], float X[:]) {
  uint8 cX = extents(X);
  uint8 rA, uint8 cA = extents(A);
  uint8 sz = SIZE;
  assert (cX==SIZE, "X must have ", sz, " elements");
  assert (rA==SIZE, "A must have ", sz, " rows");
  assert (cA==3,    "A must have 3 columns");

  // Calculate B = A.X
  float Xperim[:] = array_conperim(X, 1, 0);
  float B[:] =
    for rA (~,:) in A dot WX[3] in Xperim {
      float ip = for a in rA dot x in WX return (sum (a * x));
    } return ( vector(ip));

  // Split A into L, U, V and invert D
  float L[:], float D[:], float U[:] = A[:,0], A[:,1], A[:,2];
  float Dinv[:] = for d in D return(array(1.0/d));

  // DiB = Dinv*B  DiL =  Dinv*L  DiU = Dinv*U
  float DiB[:], float DiL[:], float DiU[:] =
    for  d in Dinv dot b in B dot l in L dot u in U
    return( array(d*b), array(d*l), array(d*u) );
```

```
// Convert to fixed point
CALC_TYPE DiBf[:] = DiB;
CALC_TYPE DiLf[:] = DiL;
CALC_TYPE DiUf[:] = DiU;

// Initial estimate for solution
CALC_TYPE Y[:] = DiB;
uint8 curr_iter = 0;
bool continue_solver = true;

// Iterative Solver on hardware/AHAHA
hardware() {
  CALC_TYPE res[:] =
    // PRAGMA (input(DiBf bram, DiLf bram, DiUf bram))
    while (continue_solver) {
      CALC_TYPE Yperim[:] =
      // PRAGMA(no_unroll, output(bram_toggle))
      for uint8 i in [SIZE+2] {
        // Needed to remove conditional array indexing
        uint8 idx = (i==0 || i==(SIZE+1))?0:(i-1);
        CALC_TYPE yval = Y[idx];
      } return(array(i==0 || i==(SIZE+1)?(CALC_TYPE)(0.0):yval));

      // Compute  next Y = DiB - (DiL+DiU) * Y
      next Y, bool large_delta_exists =
      // PRAGMA(no_unroll, output(bram_toggle, -))
      for b in DiBf dot l in DiLf dot u in DiUf dot
        window WY[3] in Yperim {
          CALC_TYPE next_y_ele = b - (l*WY[0] + u*WY[2]);
          CALC_TYPE delta = next_y_ele - WY[1];
      } return(array(next_y_ele),
               or((delta > MAX_DELTA) || (delta < -MAX_DELTA)));

      next curr_iter = curr_iter + 1;
      next continue_solver = (curr_iter < MAX_ITER) &&
                             large_delta_exists;
    } return(final(Y));
  };
}return(res);
```

# Appendix D

# K-Means in SA-C*

```
export main;


#define K 16                    // Number of clusters
#define CLUSTER_TYPE uint8      // hold max 256 clusters
#define THRESHOLD 2
#define SZ 32                   // Known image size

#define ABSDIFF(v0, v1) (((v0)>(v1))?((v0)-(v1)):((v1)-(v0)))

uint8[:,:] main(uint8 Img[SZ,SZ]) {
  // Initialize State: Select diagonal pixels
  uint32 vMean[:], uint32 xMean[:], uint32 yMean[:] =
    // PRAGMA (no_unroll)
    for uint32 k in [K] {
      uint32 j = k*SZ/K;
    } return (array(Img[j,j]), array(j), array(j));
  CLUSTER_TYPE cluster[:,:] =
    // PRAGMA (no_unroll)
    for _,_ in [SZ,SZ] {
    } return (array(K));
  uint32 num_changes = THRESHOLD + 1;
  uint32 iter = 0;

  hardware(board: "alphadata_0") {
    // Classify pixels iteratively
    CLUSTER_TYPE final_cluster[:,:], uint32 final_vMean[:] =
      // PRAGMA (input(Img memory 1 - , cluster memory_toggle 2 3 -,
                 vMean bram_toggle, xMean bram_toggle,
                 yMean bram_toggle))
      while (num_changes > THRESHOLD) {
        // Use multiple scalarvalues s as accum does not map to AHA
        next num_changes, next cluster,
```

```
uint32 vSum0,  uint32 vSum1,  uint32 vSum2,  uint32 vSum3,
uint32 vSum4,  uint32 vSum5,  uint32 vSum6,  uint32 vSum7,
uint32 vSum8,  uint32 vSum9,  uint32 vSum10, uint32 vSum11,
uint32 vSum12, uint32 vSum13, uint32 vSum14, uint32 vSum15,
uint32 xSum0,  uint32 xSum1,  uint32 xSum2,  uint32 xSum3,
uint32 xSum4,  uint32 xSum5,  uint32 xSum6,  uint32 xSum7,
uint32 xSum8,  uint32 xSum9,  uint32 xSum10, uint32 xSum11,
uint32 xSum12, uint32 xSum13, uint32 xSum14, uint32 xSum15,
uint32 ySum0,  uint32 ySum1,  uint32 ySum2,  uint32 ySum3,
uint32 ySum4,  uint32 ySum5,  uint32 ySum6,  uint32 ySum7,
uint32 ySum8,  uint32 ySum9,  uint32 ySum10, uint32 ySum11,
uint32 ySum12, uint32 ySum13, uint32 ySum14, uint32 ySum15,
uint32 nPix0,  uint32 nPix1,  uint32 nPix2,  uint32 nPix3,
uint32 nPix4,  uint32 nPix5,  uint32 nPix6,  uint32 nPix7,
uint32 nPix8,  uint32 nPix9,  uint32 nPix10, uint32 nPix11,
uint32 nPix12, uint32 nPix13, uint32 nPix14, uint32 nPix15 =
// PRAGMA (no_unroll)
for ele in Img dot c in cluster at (uint32 i, uint32 j) {
  uint32 min_dist, CLUSTER_TYPE min_k = 0xffffffff, 0;
  CLUSTER_TYPE this_k =
  // PRAGMA (no_unroll)
  for v in vMean dot x in xMean dot y in yMean
    at (CLUSTER_TYPE k) {
      uint32 dist = 4*ABSDIFF(ele,v) + ABSDIFF(i,x) +
                    ABSDIFF(j,y);
      next min_k = (dist < min_dist)?k:min_k;
      next min_dist = (dist < min_dist)?dist:min_dist;
  } return(final(min_k));
  bool change_occured = (c != this_k);
} return(sum((uint32)1, change_occured), array(this_k),
  sum((uint32)ele, this_k==0), sum((uint32)ele, this_k==1),
  sum((uint32)ele, this_k==2), sum((uint32)ele, this_k==3),
  sum((uint32)ele, this_k==4), sum((uint32)ele, this_k==5),
  sum((uint32)ele, this_k==6), sum((uint32)ele, this_k==7),
  sum((uint32)ele, this_k==8), sum((uint32)ele, this_k==9),
  sum((uint32)ele, this_k==10), sum((uint32)ele, this_k==11),
  sum((uint32)ele, this_k==12), sum((uint32)ele, this_k==13),
  sum((uint32)ele, this_k==14), sum((uint32)ele, this_k==15),
  sum(i, this_k==0), sum(i, this_k==1), sum(i, this_k==2),
  sum(i, this_k==3), sum(i, this_k==4), sum(i, this_k==5),
  sum(i, this_k==6), sum(i, this_k==7), sum(i, this_k==8),
  sum(i, this_k==9), sum(i, this_k==10), sum(i, this_k==11),
  sum(i, this_k==12), sum(i, this_k==13), sum(i, this_k==14),
  sum(i, this_k==15), sum(j, this_k==0), sum(j, this_k==1),
  sum(j, this_k==2), sum(j, this_k==3), sum(j, this_k==4),
  sum(j, this_k==5), sum(j, this_k==6), sum(j, this_k==7),
```

```
    sum(j, this_k==8), sum(j, this_k==9), sum(j, this_k==10),
    sum(j, this_k==11), sum(j, this_k==12), sum(j, this_k==13),
    sum(j, this_k==14), sum(j, this_k==15),
    sum((uint32)1, this_k==0), sum((uint32)1, this_k==1),
    sum((uint32)1, this_k==2), sum((uint32)1, this_k==3),
    sum((uint32)1, this_k==4), sum((uint32)1, this_k==5),
    sum((uint32)1, this_k==6), sum((uint32)1, this_k==7),
    sum((uint32)1, this_k==8), sum((uint32)1, this_k==9),
    sum((uint32)1, this_k==10), sum((uint32)1, this_k==11),
    sum((uint32)1, this_k==12), sum((uint32)1, this_k==13),
    sum((uint32)1, this_k==14), sum((uint32)1, this_k==15));

uint32 vSum[:], uint32 xSum[:], uint32 ySum[:], uint32 nPix[:] =
// PRAGMA (no_unroll, output(bram_toggle, bram_toggle,
                              bram_toggle, bram_toggle))
for uint8 i in [K] {
  uint32 s, uint32 x, uint32 y, uint32 n = switch(i) {
    case 0:  return (vSum0, xSum0, ySum0, nPix0)
    case 1:  return (vSum1, xSum1, ySum1, nPix1)
    case 2:  return (vSum2, xSum2, ySum2, nPix2)
    case 3:  return (vSum3, xSum3, ySum3, nPix3)
    case 4:  return (vSum4, xSum4, ySum4, nPix4)
    case 5:  return (vSum5, xSum5, ySum5, nPix5)
    case 6:  return (vSum6, xSum6, ySum6, nPix6)
    case 7:  return (vSum7, xSum7, ySum7, nPix7)
    case 8:  return (vSum8, xSum8, ySum8, nPix8)
    case 9:  return (vSum9, xSum9, ySum9, nPix9)
    case 10: return (vSum10, xSum10, ySum10, nPix10)
    case 11: return (vSum11, xSum11, ySum11, nPix11)
    case 12: return (vSum12, xSum12, ySum12, nPix12)
    case 13: return (vSum13, xSum13, ySum13, nPix13)
    case 14: return (vSum14, xSum14, ySum14, nPix14)
    default: return (vSum15, xSum15, ySum15, nPix15)
  };
} return(array(s), array(x), array(y), array(n));

next iter = iter+1;

next vMean, next xMean, next yMean =
// PRAGMA (no_unroll)
for v in vSum dot x in xSum dot y in ySum dot n in nPix {
  uint32 zn = (n==0)?1:n;
  uint32 vm = divide_using_reciprocal(v, zn);
  uint32 xm = divide_using_reciprocal(x, zn);
  uint32 ym = divide_using_reciprocal(y, zn);
} return (array(vm), array(xm), array(ym));
```

```
      } return (final(cluster), final(vMean));
    };

    uint8 R[:,:] =
      // PRAGMA(no_unroll)
      for c in final_cluster {
        uint32 v =  final_vMean[c];
        uint8 vthr = (v > 255)?255:v;
      } return(array(vthr));

} return (R);

/* This is more efficient when you are dividing multiple numbers
   by the same denominator. Assumes b < 2^30. Also, there is an
   error by 1 when a is completely divisible by b and b is not a
   power of two.*/
uint32 divide_using_reciprocal (uint32 a, uint32 b) {
  bits32 bnumer = ((bits32)1) << 30;
  uint32 unumer = bnumer;
  uint64 reciproc = divide(unumer, b);
  uint64 shift_res = a * reciproc;
  bits32 res = (bits64)shift_res >> 30;
} return ((uint32)res);

uint32 divide (uint32 a, uint32 b) {
  bits64 aa = a;
  bits64 bb = (bits64)b << 32;
  bits32 ac = 0x0;
  bits32 res =
    for uint8 i in [32] {
      next bb = bb >> 1;
      uint64 va = aa;
      uint64 vb = bb;
      uint64 vaa, next ac =
        if (aa >= bb)
          return (va-vb, (ac<<1)|0x1)
        else
          return (va, ac<<1);
      next aa = vaa;
    } return (final (ac));
} return ((uint32)res);
```

# Appendix E

# Pease FFT-64 in SA-C*

```
export main;

#define N       64  // N can be max 512 (uint8 represents 0..511)
#define HALFN   32
#define LOGN    6
#define TOP_BSZ 32
#define BOT_BSZ 64
#define FFTTYPE fix26.12
#define ROUTYPE fix14.12

FFTTYPE, FFTTYPE, FFTTYPE, FFTTYPE butterflyOp(FFTTYPE ar, FFTTYPE ai,
                  FFTTYPE br, FFTTYPE bi, ROUTYPE wr, ROUTYPE wi) {
  // bw = b * w
  FFTTYPE bwr = (br * wr) - (bi * wi);
  FFTTYPE bwi = (br * wi) + (bi * wr);

  // top = a + bw
  FFTTYPE topr = ar + bwr;
  FFTTYPE topi = ai + bwi;

  // bot = a - bw
  FFTTYPE botr = ar - bwr;
  FFTTYPE boti = ai - bwi;
} return (topr, topi, botr, boti);

// Assume data in blocks of N
process bitReverse(in stream FFTTYPE inReal, out stream FFTTYPE outReal,
                   stream FFTTYPE outImag) {
  while (inReal) {
    FFTTYPE inBlk[N] =
      // PRAGMA(no_unroll)
      for _ in [N] {
      } return (array(get(inReal)));
```

```
    // PRAGMA(no_unroll)
    for uint8 idx in [N] {
      uint8 bridx = bitReverseIndex(idx, LOGN);
      put(inBlk[bridx], outReal);
      put((FFTTYPE)0.0, outImag);
    };
  };
};

// First stage of FFT network
process FFT2_1in(in  stream FFTTYPE inReal, stream FFTTYPE inImag,
                 out stream FFTTYPE outRealT, stream FFTTYPE outImagT,
                 stream FFTTYPE outRealB, stream FFTTYPE outImagB,
                 param ROUTYPE twiddleReal0, ROUTYPE twiddleImag0) {
  while (inReal) {
    FFTTYPE topInReal, FFTTYPE topInImag = get(inReal), get(inImag);
    FFTTYPE botInReal, FFTTYPE botInImag = get(inReal), get(inImag);
    FFTTYPE topOutReal, FFTTYPE topOutImag,
    FFTTYPE botOutReal, FFTTYPE botOutImag =
      butterflyOp(topInReal, topInImag, botInReal,
                  botInImag, twiddleReal0, twiddleImag0);
    put(topOutReal, outRealT); put(topOutImag, outImagT);
    put(botOutReal, outRealB); put(botOutImag, outImagB);
  };
};

// Other stages of FFT network
process FFT2_pairin(in  stream FFTTYPE inRealT, stream FFTTYPE inImagT,
                    stream FFTTYPE inRealB, stream FFTTYPE inImagB,
                    out stream FFTTYPE outRealT, stream FFTTYPE outImagT,
                    stream FFTTYPE outRealB, stream FFTTYPE outImagB,
                    param ROUTYPE twiddleReal[:], ROUTYPE twiddleImag[:],
                    uint8 repeatFac) {
  uint8 cnt = 0; bool selT = true;
  while (inRealT || inRealB) {
    FFTTYPE topInReal, FFTTYPE topInImag,
    FFTTYPE botInReal, FFTTYPE botInImag =
      if (selT) {
      }return (get(inRealT), get(inImagT), get(inRealT), get(inImagT))
      else {
      }return (get(inRealB), get(inImagB), get(inRealB), get(inImagB));
    uint8 twIdx = (cnt/repeatFac)*repeatFac; // bit selection noop
    FFTTYPE topOutReal, FFTTYPE topOutImag,
      FFTTYPE botOutReal, FFTTYPE botOutImag =
      butterflyOp(topInReal, topInImag, botInReal,
                  botInImag, twiddleReal[twIdx], twiddleImag[twIdx]);
```

```
    put(topOutReal, outRealT); put(topOutImag, outImagT);
    put(botOutReal, outRealB); put(botOutImag, outImagB);
    uint8 cntP1 = cnt+1;
    next cnt = (cntP1 == N/2)?0:cntP1;
    next selT = (cnt < N/4);
  };
};

process merge(in stream FFTTYPE inRealT, stream FFTTYPE inImagT,
      stream FFTTYPE inRealB, stream FFTTYPE inImagB,
      out stream FFTTYPE outReal, stream FFTTYPE outImag) {
  uint8 cnt = 0; bool selT = true;
  while (inRealT || inRealB) {
    FFTTYPE rr, FFTTYPE ii =
      if (selT) {} return (get(inRealT), get(inImagT))
      else {} return (get(inRealB), get(inImagB));
    put (rr, outReal); put (ii, outImag);
    uint8 cntP1 = cnt+1;
    next selT = (cntP1 == N/2)?(!selT):selT;
    next cnt = (cntP1 == N/2)?0:cntP1;
  };
};

process network main(in stream FFTTYPE inReal,
                out stream FFTTYPE outReal, stream FFTTYPE outImag) {
  stream FFTTYPE real0; stream FFTTYPE imag0;
  stream FFTTYPE realT0; stream FFTTYPE imagT0;
  stream FFTTYPE realB0; stream FFTTYPE imagB0;
  stream FFTTYPE realT1; stream FFTTYPE imagT1;
  stream FFTTYPE realB1; stream FFTTYPE imagB1;
  stream FFTTYPE realT2; stream FFTTYPE imagT2;
  stream FFTTYPE realB2; stream FFTTYPE imagB2;
  stream FFTTYPE realT3; stream FFTTYPE imagT3;
  stream FFTTYPE realB3; stream FFTTYPE imagB3;
  stream FFTTYPE realT4; stream FFTTYPE imagT4;
  stream FFTTYPE realB4; stream FFTTYPE imagB4;
  stream FFTTYPE realT5; stream FFTTYPE imagT5;
  stream FFTTYPE realB5; stream FFTTYPE imagB5;

  ROUTYPE twiddleReal[:], ROUTYPE twiddleImag[:] = generate_twiddles(N);

  instantiate bitReverse(inReal, real0@regfile N, imag0@regfile N);
  hardware() {
    instantiate FFT2_1in(real0, imag0, realT0@regfile TOP_BSZ,
        imagT0@regfile TOP_BSZ, realB0@regfile BOT_BSZ,
        imagB0@regfile BOT_BSZ, twiddleReal[0], twiddleImag[0]);
```

```
    instantiate FFT2_pairin(realT0@regfile TOP_BSZ,
        imagT0@regfile TOP_BSZ, realB0@regfile BOT_BSZ,
        imagB0@regfile BOT_BSZ, realT1, imagT1, realB1, imagB1,
        twiddleReal, twiddleImag, 16);
    instantiate FFT2_pairin(realT1@regfile TOP_BSZ,
        imagT1@regfile TOP_BSZ, realB1@regfile BOT_BSZ,
        imagB1@regfile BOT_BSZ, realT2, imagT2, realB2, imagB2,
        twiddleReal, twiddleImag, 8);
    instantiate FFT2_pairin(realT2@regfile TOP_BSZ,
        imagT2@regfile TOP_BSZ, realB2@regfile BOT_BSZ,
        imagB2@regfile BOT_BSZ, realT3, imagT3, realB3, imagB3,
        twiddleReal, twiddleImag, 4);
    instantiate FFT2_pairin(realT3@regfile TOP_BSZ,
        imagT3@regfile TOP_BSZ, realB3@regfile BOT_BSZ,
        imagB3@regfile BOT_BSZ, realT4, imagT4, realB4, imagB4,
        twiddleReal, twiddleImag, 2);
    instantiate FFT2_pairin(realT4@regfile TOP_BSZ,
        imagT4@regfile TOP_BSZ, realB4@regfile BOT_BSZ,
        imagB4@regfile BOT_BSZ, realT5, imagT5, realB5, imagB5,
        twiddleReal, twiddleImag, 1);
    instantiate merge(realT5@regfile TOP_BSZ,
        imagT5@regfile TOP_BSZ, realB5@regfile BOT_BSZ,
        imagB5@regfile BOT_BSZ, outReal, outImag);
  };
};

uint8 bitReverseIndex(uint8 idx, uint8 num_bits) {
  bits8 b_idx = (bits8)idx;
  bits8 bridx = 0x0;
  bits8 final_bridx = for _ in [num_bits] {
      bits8 t = b_idx & 0x1;
      next bridx = (bridx << 1) | (b_idx & 0x1);
      next b_idx = b_idx >> 1;
    } return (final(bridx));
} return ((uint8)final_bridx);

// Ignore the parameter
ROUTYPE[:], ROUTYPE[:] generate_twiddles(uint8 n) {
  ROUTYPE TReal[:], ROUTYPE TImag[:] =
    // PRAGMA(no_unroll)
    for uint8 i in [HALFN] {
      float angle = -(44.0/7.0)*((float)i/(float)N);
    } return (array((ROUTYPE)cos(angle)), array((ROUTYPE)sin(angle)));
} return (TReal, TImag);
```

# Appendix F

# Non-Learning Neural Network in SA-C*

```
#define NINPUTS  2
#define NHIDDENS 5
#define IOTYPE fix10.8      // type for IO of each unit
#define WTYPE  fix16.8      // type for the weigths
#define WITYPE fix16.8      // type of wt*input
#define WSUMTYPE fix16.8    // type of dot product

export main;

IOTYPE[NHIDDENS] performHiddenUnitComp(IOTYPE currDataSet[NINPUTS],
                 WTYPE layerIpWts[NHIDDENS, NINPUTS],
                 WTYPE layerConstWts[NHIDDENS]) {
  IOTYPE hiddenRes[NHIDDENS] =
    // PRAGMA(no_unroll, output(memory 4 -))
    for currWtVec(~,:) in layerIpWts dot constWt in layerConstWts {
      WSUMTYPE weigthedSum =
      //PRAGMA(no_unroll)
      for w in currWtVec dot c in currDataSet {
      } return (sum((WSUMTYPE)((WITYPE)w*c))) + constWt;
      IOTYPE currHiddenOut = logistic(weigthedSum);
    } return (array(currHiddenOut));
} return(hiddenRes);

IOTYPE performOutputUnitComp(IOTYPE currDataSet[NHIDDENS],
                      WTYPE layerIpWts[NHIDDENS], WTYPE constWt) {
  WSUMTYPE weigthedSum =
    //PRAGMA(no_unroll)
    for w in layerIpWts dot c in currDataSet {
    } return (sum((WSUMTYPE)((WITYPE)w*c))) + constWt;
  IOTYPE currOut = logistic(weigthedSum);
} return(currOut);
```

```
// Produce a sequence of output sets for a sequence of input sets
IOTYPE[:] main(IOTYPE dataSets[:,NINPUTS],
       WTYPE  hiddenLayerIpWts[NHIDDENS, NINPUTS],
       WTYPE  hiddenLayerConstWts[NHIDDENS],
       WTYPE  outputLayerHiddenWts[NHIDDENS],
       WTYPE  outputLayerConstWts) {
  hardware(){
  IOTYPE Res[:] =
    // PRAGMA(no_unroll, input(dataSets memory 1 -, hiddenLayerIpWts
       memory 2 -, hiddenLayerConstWts bram, outputLayerHiddenWts
       memory 2 -), output(memory 3 -))
    for currDataSet(~,:) in dataSets {
      IOTYPE hiddenRes[NHIDDENS] = performHiddenUnitComp(currDataSet,
                                hiddenLayerIpWts, hiddenLayerConstWts);
      IOTYPE outRes = performOutputUnitComp(hiddenRes,
                              outputLayerHiddenWts, outputLayerConstWts);
    } return(array(outRes));
  };
} return (Res);

// Piece wise linear calculation
IOTYPE logistic(WSUMTYPE x) {
  WSUMTYPE xPosRange[6] = {0, 0.25, 0.5, 1, 2, 4};
  float yPosRange[6] = {0.0, 0.124353, 0.244919, 0.462117,
                        0.761594, 0.964028};

  // Last idx with xposrange <= absx
  WSUMTYPE absX = (x<0)?(-x):x;
  uint8 idx[1] =  for xPos in xPosRange at (uint8 i) {
  } return (vals_at_last_max(xPos, {i}, xPos<=absX));

  fix16.8 slope, IOTYPE y1, WSUMTYPE x1 =
    switch(idx[0]) {
      case 0: return ((fix16.8)((yPosRange[1]-yPosRange[0])/
                      (xPosRange[1]-xPosRange[0])),
                      (IOTYPE)yPosRange[0], (WSUMTYPE)xPosRange[0])
      case 1: return ((fix16.8)((yPosRange[2]-yPosRange[1])/
                      (xPosRange[2]-xPosRange[1])),
                      (IOTYPE)yPosRange[1], (WSUMTYPE)xPosRange[1])
      case 2: return ((fix16.8)((yPosRange[3]-yPosRange[2])/
                      (xPosRange[3]-xPosRange[2])),
                      (IOTYPE)yPosRange[2], (WSUMTYPE)xPosRange[2])
      case 3: return ((fix16.8)((yPosRange[4]-yPosRange[3])/
                      (xPosRange[4]-xPosRange[3])),
                      (IOTYPE)yPosRange[3], (WSUMTYPE)xPosRange[3])
```

```
      case 4: return ((fix16.8)((yPosRange[5]-yPosRange[4])/
                      (xPosRange[5]-xPosRange[4])),
                      (IOTYPE)yPosRange[4], (WSUMTYPE)xPosRange[4])
      default: return ((fix16.8)0, (IOTYPE)1, (WSUMTYPE)0)
    };

  IOTYPE absY = y1 + slope*(absX-x1);
  IOTYPE y = (x<0)?(-absY):absY;
} return (y);
```

# Appendix G

# Learning Neural Network in SA-C*

```
#define NINPUTS  2
#define NHIDDENS  5

#define IOTYPE fix16.12           // type for IO of each unit
#define WTYPE  fix16.12           // type for the weigths
#define WITYPE fix16.12           // type of wt*input
#define WSUMTYPE fix16.12         // type of dot product
#define CONST_RATE ((WITYPE)0.1)  // learning rate with default value

export main;

process forward_pass(in stream IOTYPE ipSetStream[NINPUTS],
            stream WTYPE hiddenLayerIpWtsStream[NHIDDENS, NINPUTS],
            stream WTYPE hiddenLayerConstWtsStream[NHIDDENS],
            stream WTYPE outputLayerHiddenWtsStream[NHIDDENS],
            stream WTYPE outputLayerConstWtsStream,
            out stream WTYPE hiddenFwdResStream[NHIDDENS],
            stream IOTYPE calcOpStream,
            stream WTYPE finalHiddenLayerIpWtsStream[NHIDDENS, NINPUTS],
            stream WTYPE finalHiddenLayerConstWtsStream[NHIDDENS],
            stream WTYPE finalOutputLayerHiddenWtsStream[NHIDDENS],
            stream WTYPE finalOutputLayerConstWtsStream) {
  while (ipSetStream) {
    IOTYPE currIpDataSet[NINPUTS] = get(ipSetStream);
    WTYPE hiddenLayerIpWts[NHIDDENS, NINPUTS] =
          get(hiddenLayerIpWtsStream);
    WTYPE hiddenLayerConstWts[NHIDDENS]=get(hiddenLayerConstWtsStream);
    WTYPE outputLayerHiddenWts[NHIDDENS] =
          get(outputLayerHiddenWtsStream);
    WTYPE outputLayerConstWts = get(outputLayerConstWtsStream);
```

```
    WTYPE hiddenFwdRes1[NHIDDENS], WTYPE hiddenFwdRes2[NHIDDENS] =
      performHiddenUnitComp(currIpDataSet, hiddenLayerIpWts,
                            hiddenLayerConstWts);
    put(hiddenFwdRes1, hiddenFwdResStream);
    IOTYPE outRes = performOutputUnitComp(hiddenFwdRes2,
          outputLayerHiddenWts, outputLayerConstWts);
    put(outRes, calcOpStream);
  };

  WTYPE hiddenLayerIpWts[NHIDDENS,NINPUTS]=get(hiddenLayerIpWtsStream);
  WTYPE hiddenLayerConstWts[NHIDDENS]=get(hiddenLayerConstWtsStream);
  WTYPE outputLayerHiddenWts[NHIDDENS] =
        get(outputLayerHiddenWtsStream);
  WTYPE outputLayerConstWts = get(outputLayerConstWtsStream);
  WTYPE copyHiddenLayerIpWts[NHIDDENS, NINPUTS] =
    // PRAGMA(no_unroll)
    for ele in hiddenLayerIpWts {} return(array(ele));
  WTYPE copyHiddenLayerConstWts[NHIDDENS] =
    // PRAGMA(no_unroll)
    for ele in hiddenLayerConstWts {} return(array(ele));
  WTYPE copyOutputLayerHiddenWts[NHIDDENS] =
    // PRAGMA(no_unroll)
    for ele in outputLayerHiddenWts {} return(array(ele));
  put(copyHiddenLayerIpWts, finalHiddenLayerIpWtsStream);
  put(copyHiddenLayerConstWts, finalHiddenLayerConstWtsStream);
  put(copyOutputLayerHiddenWts, finalOutputLayerHiddenWtsStream);
  put(outputLayerConstWts, finalOutputLayerConstWtsStream);
};

process back_prop(in stream WTYPE hiddenFwdResStream[NHIDDENS],
        stream IOTYPE ipSetStream[NINPUTS],
        stream IOTYPE realOpStream, stream IOTYPE calcOpStream,
        out stream WTYPE hiddenLayerIpWtsStream[NHIDDENS, NINPUTS],
        stream WTYPE hiddenLayerConstWtsStream[NHIDDENS],
        stream WTYPE outputLayerHiddenWtsStream[NHIDDENS],
        stream WTYPE outputLayerConstWtsStream,
        param WTYPE hiddenLayerIpWts[NHIDDENS, NINPUTS],
        WTYPE hiddenLayerConstWts[NHIDDENS],
        WTYPE outputLayerHiddenWts[NHIDDENS],
        WTYPE outputLayerConstWts) {
  WTYPE copyHiddenLayerIpWts[NHIDDENS, NINPUTS] =
    // PRAGMA(no_unroll)
    for ele in hiddenLayerIpWts {} return(array(ele));
  WTYPE copyHiddenLayerConstWts[NHIDDENS] =
    // PRAGMA(no_unroll)
    for ele in hiddenLayerConstWts {} return(array(ele));
```

```
    WTYPE copyOutputLayerHiddenWts[NHIDDENS] =
      // PRAGMA(no_unroll)
      for ele in outputLayerHiddenWts {} return(array(ele));
    put(copyHiddenLayerIpWts, hiddenLayerIpWtsStream);
    put(copyHiddenLayerConstWts, hiddenLayerConstWtsStream);
    put(copyOutputLayerHiddenWts, outputLayerHiddenWtsStream);
    put(outputLayerConstWts, outputLayerConstWtsStream);

    // PRAGMA(input(hiddenLayerIpWts bram_toggle, hiddenLayerConstWts
              bram_toggle, outputLayerHiddenWts bram_toggle))
    while (realOpStream) {
      IOTYPE currIpDataSet[NINPUTS] = get(ipSetStream);
      IOTYPE realOutRes = get(realOpStream);
      IOTYPE calcOutRes = get(calcOpStream);
      WTYPE hiddenFwdRes[NHIDDENS] = get(hiddenFwdResStream);

      IOTYPE err = realOutRes - calcOutRes;
      WITYPE deltaerr = err * Dlogistic(calcOutRes) * CONST_RATE;

      WITYPE deltaHiddenBkwdRes[NHIDDENS], next outputLayerHiddenWts,
        WTYPE copyOutputLayerHiddenWts[NHIDDENS] =
        // PRAGMA (no_unroll, output(bram_toggle, -, bram_toggle))
        for hres in hiddenFwdRes dot ow in outputLayerHiddenWts {
          WITYPE dhr = deltaerr*ow*Dlogistic(hres);
          WTYPE tres = ow+deltaerr*hres;
        } return (array(dhr), array(tres), array(tres));

      next outputLayerConstWts = outputLayerConstWts + deltaerr;
      next hiddenLayerConstWts, WTYPE copyHiddenLayerConstWts[NHIDDENS]=
        // PRAGMA (no_unroll, output(-, bram_toggle))
        for dhbr in deltaHiddenBkwdRes dot hlcw in hiddenLayerConstWts {
        } return (array(hlcw+dhbr), array(hlcw+dhbr));
      next hiddenLayerIpWts,
        WTYPE copyHiddenLayerIpWts[NHIDDENS,NINPUTS]=
        // PRAGMA (no_unroll, output(-, bram_toggle))
        for hliw in hiddenLayerIpWts at (uint8 i, uint8 j) {
          WTYPE tres = hliw+deltaHiddenBkwdRes[i]*currIpDataSet[j];
        } return (array(tres), array(tres));

      put(copyHiddenLayerIpWts, hiddenLayerIpWtsStream);
      put(copyHiddenLayerConstWts, hiddenLayerConstWtsStream);
      put(copyOutputLayerHiddenWts, outputLayerHiddenWtsStream);
      put(outputLayerConstWts, outputLayerConstWtsStream);
    };
  };
```

```
process data_producer(in stream uint16 iterCntStream,
        out stream IOTYPE ipSetStream1[NINPUTS],
        stream IOTYPE ipSetStream2[NINPUTS],stream IOTYPE opSetStream,
        param IOTYPE ipDataSets[:,NINPUTS], IOTYPE outDataSets[:]) {
  uint16 iterCnt = get(iterCntStream);
  for _ in [iterCnt] {
    for currIpDataSet(~,:) in ipDataSets dot
        currOutDataSet in outDataSets {
      // Create a copy to output on the stream
      IOTYPE ipCopy1[NINPUTS], IOTYPE ipCopy2[NINPUTS]  =
        // PRAGMA(no_unroll)
        for ele in currIpDataSet {} return(array(ele), array(ele));
      put(ipCopy1, ipSetStream1);
      put(ipCopy2, ipSetStream2);
      put(currOutDataSet, opSetStream);
    };
  };
};

process network main(in stream uint16 iterCntStream,
       out stream WTYPE finalHiddenLayerIpWtsStream[NHIDDENS,NINPUTS],
       stream WTYPE finalHiddenLayerConstWtsStream[NHIDDENS],
       stream WTYPE finalOutputLayerHiddenWtsStream[NHIDDENS],
       stream WTYPE finalOutputLayerConstWtsStream,
       param IOTYPE ipDataSets[:,NINPUTS], IOTYPE outDataSets[:],
       WTYPE hiddenLayerIpWts[NHIDDENS, NINPUTS],
       WTYPE hiddenLayerConstWts[NHIDDENS],
       WTYPE outputLayerHiddenWts[NHIDDENS],
       WTYPE outputLayerConstWts) {
  stream IOTYPE ipSetStream1[NINPUTS];
  stream IOTYPE ipSetStream2[NINPUTS];
  stream IOTYPE opSetStream;
  stream WTYPE hiddenLayerIpWtsStream[NHIDDENS, NINPUTS];
  stream WTYPE hiddenLayerConstWtsStream[NHIDDENS];
  stream WTYPE outputLayerHiddenWtsStream[NHIDDENS];
  stream WTYPE outputLayerConstWtsStream;
  stream WTYPE hiddenFwdResStream[NHIDDENS];
  stream IOTYPE calcOpStream;
  stream WITYPE deltaHiddenBkwdResStream[NHIDDENS];

  hardware () {
    instantiate data_producer(iterCntStream,
                    ipSetStream1@regfile 2 bram_toggle,
                    ipSetStream2@regfile 2 bram_toggle,
                    opSetStream, ipDataSets, outDataSets);
```

```
    instantiate forward_pass(ipSetStream1,
                hiddenLayerIpWtsStream@regfile 2 bram_toggle,
                hiddenLayerConstWtsStream@regfile 2 bram_toggle,
                outputLayerHiddenWtsStream@regfile 2 bram_toggle,
                outputLayerConstWtsStream,
                hiddenFwdResStream@regfile 2 bram_toggle,calcOpStream,
                finalHiddenLayerIpWtsStream@regfile 2 bram_toggle,
                finalHiddenLayerConstWtsStream@regfile 2 bram_toggle,
                finalOutputLayerHiddenWtsStream@regfile 2 bram_toggle,
                finalOutputLayerConstWtsStream);
    instantiate back_prop(hiddenFwdResStream,ipSetStream2,opSetStream,
        calcOpStream,hiddenLayerIpWtsStream,hiddenLayerConstWtsStream,
        outputLayerHiddenWtsStream, outputLayerConstWtsStream,
        hiddenLayerIpWts, hiddenLayerConstWts, outputLayerHiddenWts,
        outputLayerConstWts);
  };
};

IOTYPE[NHIDDENS], IOTYPE[NHIDDENS] performHiddenUnitComp
     (IOTYPE currDataSet[NINPUTS], WTYPE layerIpWts[NHIDDENS,NINPUTS],
      WTYPE layerConstWts[NHIDDENS]) {
  IOTYPE hiddenRes1[NHIDDENS], IOTYPE hiddenRes2[NHIDDENS] =
    // PRAGMA(no_unroll, output(bram_toggle, bram_toggle))
    for currWtVec(~,:) in layerIpWts dot constWt in layerConstWts {
      WSUMTYPE weigthedSum =
      //PRAGMA(no_unroll)
      for w in currWtVec dot c in currDataSet {
      } return (sum((WSUMTYPE)((WITYPE)w*c))) + constWt;
      IOTYPE currHiddenOut = logistic(weigthedSum);
      WSUMTYPE tmpSum =
      //PRAGMA(no_unroll)
      for w in currWtVec dot c in currDataSet {
      } return (sum((WSUMTYPE)((WITYPE)w*c)));
    } return (array(currHiddenOut), array(currHiddenOut));
} return(hiddenRes1, hiddenRes2);

IOTYPE performOutputUnitComp(IOTYPE currDataSet[NHIDDENS],
     WTYPE layerIpWts[NHIDDENS],
     WTYPE constWt) {
  WSUMTYPE weigthedSum =
    //PRAGMA(no_unroll)
    for w in layerIpWts dot c in currDataSet {
    } return (sum((WSUMTYPE)((WITYPE)w*c))) + constWt;
  IOTYPE currOut = logistic(weigthedSum);
} return(currOut);
```

```
// Piece wise linear calculation
IOTYPE logistic(WSUMTYPE x) {
  WSUMTYPE xPosRange[6] = {0, 0.25, 0.5, 1, 2, 4};
  float yPosRange[6] = {0.0, 0.124353, 0.244919, 0.462117,
                        0.761594, 0.964028};

  // Last idx with xposrange <= absx
  WSUMTYPE absX = (x<0)?(-x):x;
  uint8 idx[1] =  for xPos in xPosRange at (uint8 i) {
  } return (vals_at_last_max(xPos, {i}, xPos<=absX));

  fix16.8 slope, IOTYPE y1, WSUMTYPE x1 =
    switch(idx[0]) {
      case 0: return ((fix16.8)((yPosRange[1]-yPosRange[0])/
                      (xPosRange[1]-xPosRange[0])),
                      (IOTYPE)yPosRange[0], (WSUMTYPE)xPosRange[0])
      case 1: return ((fix16.8)((yPosRange[2]-yPosRange[1])/
                      (xPosRange[2]-xPosRange[1])),
                      (IOTYPE)yPosRange[1], (WSUMTYPE)xPosRange[1])
      case 2: return ((fix16.8)((yPosRange[3]-yPosRange[2])/
                      (xPosRange[3]-xPosRange[2])),
                      (IOTYPE)yPosRange[2], (WSUMTYPE)xPosRange[2])
      case 3: return ((fix16.8)((yPosRange[4]-yPosRange[3])/
                      (xPosRange[4]-xPosRange[3])),
                      (IOTYPE)yPosRange[3], (WSUMTYPE)xPosRange[3])
      case 4: return ((fix16.8)((yPosRange[5]-yPosRange[4])/
                      (xPosRange[5]-xPosRange[4])),
                      (IOTYPE)yPosRange[4], (WSUMTYPE)xPosRange[4])
      default: return ((fix16.8)0, (IOTYPE)1, (WSUMTYPE)0)
    };

  IOTYPE absY = y1 + slope*(absX-x1);
  IOTYPE y = (x<0)?(-absY):absY;
} return (y);

WITYPE Dlogistic(WITYPE y)
{
  WITYPE t = (WITYPE)(y*y);
} return ((WITYPE)1 - t);
```

# Appendix H

# New AHAHA Model in SA-C*

The new target hardware model for the SA-C* compiler is called the aggregate, hierarchical, abstract hardware architecture (AHAHA) model. This new model has been designed by Charles Ross as a part of his doctoral dissertation. The new model provided capabilities that allowed mapping of while-loops and nested loops. The requirements of the new AHAHA model (with consequences in translation) are shown below:

1. In order to simplify the new AHAHA synchronization model, only fully-clocked and purely combinational nodes are allowed. The semi-clocked nodes from the old AHA model are converted into fully-clocked nodes by passing the *end_of_sequence* signal through them and requiring them to handle synchronization issues along with the clock and state issues. For example, counters in the new AHAHA model are fully-clocked.

2. In the new AHAHA model, nodes communicating with the memory take the bus-widths into consideration. Hence change-widths are sometimes necessary to convert the run-time constants into widths corresponding to the appropriate bus. This simplifies the final VHDL code-generation and provides better space and time estimation.

3. Run-time constants are no longer sticky. The DFG to AHAHA translator needs to replicate the run-time inputs depending upon how many times they are used. This simplifies the translation of nested loops into AHAHA graphs.

4. Just like in the old AHA model, each synchronous node in the new AHAHA model is made up of two halves (the consumer and the producer half). The difference is that each half can now be made up of one or more clusters (each one with their own firing rules). This requirement is a side-effect of the non-sticky inputs. For example, the consumer half can be made up of a cluster that handles the initial state and another cluster handling the repetitive execution. Similarly, the producer half can be made up of a repetitive state and a cluster handling the final result.

5. The multiple cluster model requires that a section in the new AHAHA graph be made up of producer cluster(s), combination node(s) and consumer cluster(s). Note that in the old AHA model, the synchronization occurred between halves and not clusters.

6. To generalize the memory model, the *end_of_sequence* signal does not bypass the write-word in the new AHAHA model. The bypass of the *end_of_sequence* signal (in the old AHA model) worked as long as the latency of the write-word was exactly one clock-cycle. The new model allows for longer latencies for write operations.

7. Also, the output node in the new AHAHA model gets a single true done token (and not a series of false *end_of_sequence* tokens followed by a true *end_of_sequence* token). This is achieved by inserting a *done* AHAHA node. This node simply consumes the false *end_of_sequence* tokens and only passes the true signal through. This change simplifies the logic for the interrupt back to the host processor.

Some of the AHAHA nodes have *pragmas* attached to them. They provide compile-time known information about the AHAHA node. This information can simplify the resultant VHDL code as it can be used as a *generic* in the VHDL code. Some pragmas are user provided while others are compiler generated. For example, pragmas on nodes communicating with memory are user provided and provide information about which memory to communicate with. This allows the compiler to group nodes for better memory arbitration. Pragmas on nodes like shift-register are created by the compiler

to indicate the size and the step of the sliding window. The *generic* in VHDL can then produce the exact amount of registers to hold all the values. A description of the fully-clocked AHAHA nodes is given below.

1. **Token Generator**: It acts as a driver of a loop with a known number of iterations. The node has two inputs and one output. The inputs correspond to the iteration count of the loop and an *end_of_sequence* from the outer loop or outer dimension driver. For each input pair received, the token generator produces a sequence of *end_of_sequence* flags (an iteration count number of false flags followed by a single true flag).

2. **Counter**: A counter node has three inputs (a start value, an increment value and an input *end_of_sequence* flag) and two outputs (the current count and output *end_of_sequence* flag). In the initial state, the counter reads in the start and increment value. Then, for each *end_of_sequence* input received, it pairs the next value in the sequence ($start$, $start + increment$, $start + 2 * increment$, ...) with the *end_of_sequence* read in as the next output sequence. When the *end_of_sequence* output signal is true, the counter resets itself to the initial state (to read in the next pair of start and increment values).

3. **Address Calculator**: It has three inputs (a start address, an increment and an input *end_of_sequence* flag) and two outputs (the current address and output *end_of_sequence* flag). For each input set read in (with *end_of_sequence* flag as false), the address calculator calculates and produces a sequence of addresses ($start$, $start + increment$, $start + 2 * increment$, ...). The number of addresses to be produced is indicated by a compiler generated *depth* pragma on the node. For a true *end_of_sequence* input flag, the address calculator node produces a single address (equal to the start address) with output *end_of_sequence* flag as true.

4. **Read Word**: This node has two inputs (the address and an *end_of_sequence* flag) and two outputs (the data read from the specified address and an output

*end_of_sequence* flag). The output data corresponds to the data read in from the specified address when *end_of_sequence* flag is false but is a garbage/ignored value when *end_of_sequence* flag is true. Also, there are two types of read words: one reading from memory and another reading from block RAM. Having two different types of read words simplifies memory arbitration and better handles the differences in latency issues.

5. **Write Word**: This node has three inputs (the address location for the write, the data to be written and an *end_of_sequence* flag) and one output (an output *end_of_sequence* flag). When the input *end_of_sequence* flag is false, the input data is written at the specified at the address. When the input *end_of_sequence* flag is true, the write is simply ignored. Once this action is complete, the node simply passes the input *end_of_sequence* flag on its output port. Also, there are two types of write words: one writing to memory and another writing to block RAM.

6. **Shift Register**: A shift register performs the sliding window operation. The width of the window is specified by the *depth* pragma while the amount of shifting/sliding is specified by the *step* pragma. It has two inputs (a pixel and an *end_of_sequence* flag) and two outputs (the packed window and an *end_of_sequence* flag). For each input with a false *end_of_sequence* flag, the node shifts the new pixel into its current window. When a new window with the appropriate shifting is complete, the node outputs the complete window with a false *end_of_sequence* flag. If it receives a true *end_of_sequence* flag before completion of the next sliding window, it outputs a garbage/ignored window along with a true *end_of_sequence* flag.

7. **Fifo Pack**: This node packs a sequence of input pixels into a parallel group of pixels. It is generally used to group the output pixels of a tile into a word. The number of pixels to be grouped is provided by the *depth* pragma. The node has two inputs (an input pixel and an *end_of_sequence* flag) and two outputs (the grouped pixels and an *end_of_sequence* flag). The node outputs the packed pixels along with

a false *end_of_sequence* flag. In case it receives a true *end_of_sequence* flag before all the *depth* number of pixels are received, the node packs the unfilled locations with garbage values and outputs this packed result with a false *end_of_sequence* flag. This is followed by a complete garbage result with a true *end_of_sequence* flag. In certain situations, it is possible to predict that the number of pixels arriving before a true *end_of_sequence* flag will always be a multiple of *depth* (for example, packing of multi-dimension windows will always have correct packing in the outer dimensions). In these situations, a more efficient node called *fifo pack full* can be used.

8. **Fifo Unpack**: This node unpacks parallel, grouped pixels into a sequence of pixels. It has two inputs (the grouped pixels and an *end_of_sequence* flag) and two outputs (the split output pixel sequence and an *end_of_sequence* flag sequence). The number of pixels in each group is given by the *depth* pragma. For each pair of grouped pixel received with a false *end_of_sequence* flag, the fifo unpack produces *depth* number of paired sequences (of pixels and false *end_of_sequence* flag). For a true *end_of_sequence* flag received, it outputs a single pair of a garbage pixel with a true output *end_of_sequence* flag.

9. **Buffer**: This node is used to balance latency issues across different parts of an AHAHA graph. The buffer node can buffer a sequence of values. The amount of values that it can buffer is given by the *depth* pragma. It has exactly one input (for the data that needs buffering) and one output (corresponding to the buffered output).

10. **Replicator**: This node is used for repeating a data value multiple times. It has two inputs (data to be repeated and an *end_of_sequence* flag) and a single output (corresponding to the repeated value). In the initial state, the replicator flag reads in the input data (that needs replication). Then, for each *end_of_sequence* flag received (true or false), the node repeats the data already read in on its output

port. After the node has read in a true *end_of_sequence* flag, it resets itself into the initial state (to read in a new input value for replication).

11. **Mask**: This node is used to handle dummy iterations (that can be generated because of window scrunching optimization [19]). It has three inputs (data, mask and *end_of_sequence* flag) and two outputs (data and *end_of_sequence* flag). This node passes the input data and the *end_of_sequence* flag through onto its outputs ports only when the mask is false or when the *end_of_sequence* flag is true.

12. **Done**: This node has a single input and a single output both corresponding to the *end_of_sequence* flag. This node simply consumes all the false input *end_of_sequence* flags and only allows the true *end_of_sequence* flag to pass through.

13. **Circulate**: This node handles induction variables in the hardware model. It has four inputs (the initial value, the next value, dummy flag and *end_of_sequence* flag) and two outputs (the final value and the current iteration value). The two outputs are needed as the final value is used exactly once while the current iteration value is used every loop iteration. The different consumption rates implies they need to lie in different clusters (and hence on different output ports). The node has an internal register that stores the current value of the induction variable. In the initial state, the node obtains in the initial value from its input port and sets the register to that value. In the repetitive state, the node reads in the next value along with the dummy and *end_of_sequence* flags. The internal register gets updated only when the dummy and *end_of_sequence* flag are both false. When the *end_of_sequence* flag is true, the register value is delivered on the final value output port. For all other executions, the registered value is delivered on the current iteration output port.

14. **Hold**: This node is used to delay the transfer of values (especially location information for an array). This is achieved by giving the location information to the producer loop as well as a hold node. The output of the hold node is connected to all the consumer loops of the array. The hold node also receives a done signal from

the producer loop so that the location information is transferred to the consumer loops only after the producer loop has completed execution.

15. **Array Info Select**: This node is used to toggle between multiple array locations for nextified arrays. It receives three sets of location information (the location for the initial array and two locations for toggling during repeated iterations of outer loop). It also receives the done signals of the producer and consumer loops, the dummy and *end_of_sequence* flags from its driver loop. It produces three sets of output locations (the location of the final result, the location for loops consuming the array created in the previous iteration and the location for next array to the produced and consumed in the current iteration). The two toggled, output locations can be associated with the SA-C* code easily. The first location is for consumer loops occurring above the *next* assignment in the loop-body of the SA-C* code while the second output location is for the producer (or right hand side) of the *next* assignment and the consumers of the array following the assignment in the SA-C* loop-body.