Stream-Oriented FPGA Computing in the Streams-C High Level Language

Maya Gokhale
Los Alamos National Laboratory
Space Data Systems Group
MS D440
Los Alamos NM 87545
maya@lanl.gov

Jan Stone
Consultant
Princeton, NJ
jmstone@nerc.com

Jeff Arnold
Adaptive Silicon, Inc.
987 University Ave
Los Gatos CA 95032
jmarnold@znet.com

Mirek Kalinowski
Sarnoff Corporation
CN5300
Princeton, NJ 08543-5300
mirek@sarnoff.com

Abstract

Stream oriented processing is an important methodology used in FPGA-based parallel processing. Characteristics of stream-oriented computing include high-data-rate flow of one or more data sources; fixed size, small stream payload (one byte to one word); compute-intensive operations, usually low precision fixed point, on the data stream; access to small local memories holding coefficients and other constants; and occasional synchronization between computational phases. In this paper we describe language constructs, compiler technology, and hardware/software libraries embodying the Streams-C system which has been developed to support stream-oriented computation on FPGA-based parallel computers. The language is implemented as a small set of library functions callable from a C language program. The Streams-C compiler synthesizes hardware circuits for multiple FPGAs as well as a multi-threaded software program for the control processor. Our system includes a functional simulation environment based on POSIX threads, allowing the programmer to simulate the collection of parallel processes and their communication at the functional level. Finally we present an application written both in Streams-C and hand-coded in VHDL. Compared to the hand-crafted design, the Streams-C-generated circuit takes 3x the area and runs at 1/2 the clock rate. In terms of time to market, the hand-done design took a month to develop by an experienced hardware developer. The Streams-C design took a couple of days, for a productivity increase of 10x.

1. Introduction

The concept of using Field Programmable Gate Arrays (FPGAs) as customizable compute engines began in the late 1980's, and since then, many realizations of that concept have been developed that have delivered the promised performance acceleration. However, that 10x-100x of performance that can be obtained for suitable applications on Reconfigurable Computers (RCCs) has been at the cost of 10x-100x increase in difficulty of application development. FPGA-based systems offer the programmability of software, allowing a vast number of applications and application variants to be mapped onto them. Despite many promising research efforts, the mainstream of application development must use Computer-Aided Design (CAD) tools that are oriented towards hardware rather than software development, characterized by high cost of the tool set, long compile times if a reasonable level of abstraction and portability are desired, and most important, the necessity of developing and completely understanding the cycle-by-cycle behavior of millions of gates spanning multiple FPGA chips and fixed function units.

In recent years, a concerted effort has been launched to remedy the design tool problem. Many of these tools are embedded in high level design simulation environments (Ptolemy, Khoros, MatLab, Handel C)([6], [7]) or target “dusty deck” sequential procedural code ([3]), while others target low level, technology-specific, optimized designs ([1], [5]).

In this work, we take an intermediate approach between those very high level and very low level design tools. Our
With current compiler technology, parallelization of the application and mapping to the FPGA board architecture are best performed by the application developer. This is in keeping with methods of programming conventional parallel machines, in which the application developer usually manually parallelizes the program and inserts message-passing and synchronization logic. However, it is our thesis that the application developer should not have to be a hardware designer in order to develop reasonably efficient programs, that clock-cycle-level of specification should not be required. With this middle approach, software engineers knowledgeable in parallel programming can create applications on FPGA-based processors.

The Streams-C project described in this paper embodies the above design goals. The Streams-C programming model is targeted at stream-oriented FPGA applications. Characteristics of stream-oriented computing include high data-rate flow of one or more data sources; fixed size, small stream payload (one byte to one word); compute-intensive operations, usually low precision fixed point, on the data stream; access to small local memories holding coefficients and other constants; and occasional synchronization between computational phases.

The Streams-C language is actually a small set of annotations and library functions callable from a conventional C program. The annotations are used to declare a process, stream, or signal, and to assign resources on the FPGA board to those objects. The library functions are used to communicate stream data between the processes. The Streams-C compiler synthesizes hardware circuits for one or more FPGAs as well as a multi-threaded control program on the host processor. The compiler includes previously reported features ([2]) extended to pipelined stream computation, so that the generated hardware/software is capable of pipelining a computation across multiple FPGAs and the conventional processor. Our system includes a functional simulation environment based on POSIX threads, allowing the programmer to simulate the collection of parallel processes and their communication at the functional level.

In this paper we describe the Streams-C language, give an overview of the compiler, focusing for the most part on the Streams-C hardware libraries, and then show an application that has been written in Streams-C as well as directly implemented by hand in VHDL. Our results show that the Streams-C-generated hardware for an image processing application, a histogram projection contrast enhancement algorithm, takes approximately 3x the number of logic blocks on a Xilinx 4036 FPGA as a hand-coded design and runs at 1/2 the clock rate. The design time with Streams-C is a factor of 10 less.

2. Streams-C Programming Model

Streams-C follows the Communicating Sequential Processes (CSP) [4] parallel programming model. The implementation is a combination of annotations and library functions callable from C. This is for pragmatic reasons as Streams-C builds on several years of compiler development within the SUIF C-processing framework.

In our model, there are three distinguished objects, processes, streams and signals. A process is an independently executing object with a process body that is given by a C subroutine. A process can run on the host processor or on an FPGA chip. For an FPGA process, the process body accesses only local data and is written in a subset of C supported by the Streams-C compiler. In addition, intrinsic functions to perform stream or signal operations may be referenced. All declared processes are initiated when the program begins and run until their subroutine bodies complete.

2.1. Process Declarations

Examples of process declarations are shown in Figure 1. In the Figure, five processes have been declared, with their logical stream names. Process read_image_run has two output streams, called word_o and ImageDef_o respectively. Process controller_run has a single input stream input_i and a single output stream frame_o. The other processes are similarly defined.

2.2. Process Body

Following a process declaration, the process body is given, as shown in Figure 2. The process may be either a host process or an FPGA process. Host processes can do file I/O and use the full C language. FPGA processes must adhere to our supported C language subset. The compiler can synthesize logic for operations using unsigned integers of arbitrary bit length and arrays of integers. The programs must use structured programming constructs. In-lining of function calls is currently being implemented.

The declaration SC_FLAG is used when a stream element is accompanied by a one-bit flag to be used by the application. In this program, a fragment of a histogramming contrast enhancement algorithm, the flag is used to signal end-of-frame. The SC_REG declaration is used to identify
PROCESS-FUN.read-image-run
/// OUTPUT word_o
/// OUTPUT ImageDef_o

PROCESS-FUN.controller_run
/// INPUT input_i
/// OUTPUT frame_o

PROCESS-FUN.contrast_run
/// INPUT frame_i
/// OUTPUT remap_o

PROCESS-FUN.remap_run
/// INPUT remap_i
/// OUTPUT output_o

PROCESS-FUN.write_image_run
/// INPUT inimg_i
/// INPUT ImageDef_i

Figure 1. Process Declarations in Streams-C

stream ports. Stream operations include open, close, read, write, and end-of-stream check. The intrinsic functions SC_REG_GET_BITS_INT and SC_REG_PUT_BITS_INT extract bits from and insert bits into stream ports. Variable declarations and executable statements are standard C code.

2.3. Process and Stream Attributes

Attributes of the processes and streams must be defined in annotations as shown in Figure 3.

There are five stream declarations. A declaration gives the stream name and type. For synthesis, a physical path is specified. There follow the bit width of the stream element, the number of buffers, the number of readers, and the first bit used.

The example also shows process specifications for readimage, controller, and contrast. These give the physical input and output streams corresponding to the logical streams used by the process. In addition, for synthesis, the process is mapped to a physical chip and the name of the process's subroutine body is given.

2.4. Functional Simulation

A Streams-C program may be simulated at the functional level (see figure 4). Our functional simulator uses the Linux pthreads package to support concurrent processes and stream communication. At this level, the programmer can use conventional software debuggers and "print" state-

Figure 2. Process Body in Streams-C
// stream definitions

STREAM word_stream TYPE VALID_BIT
    PATHwf_h_b0pe0 32 1 1 0
STREAM ImageDef_stream TYPE BUFFER
    PATHwf_h_h 32 1 1 0
STREAM frame_stream TYPE VALID_BIT
    PATH wf_b0pe0_b0pe2 32 1 1 0
STREAM remap_stream TYPE BUFFER
    PATH wf_b0pe2_b0pe1 16 1 1 0
STREAM output_stream TYPE VALID_BIT
    PATHwf_b0pe1_h 32 1 1 0

// process definitions

// read image
// this process is on the host
PROCESS read_image PROCESSOR wf_host
    RUN_FUNCTION read_image_run
    OUTPUT word_stream
    OUTPUT ImageDef_stream
    PROCESS_END

// controller
// this process is on PE 0
PROCESS controller PROCESSOR wf_b0pe0
    RUN_FUNCTION controller_run
    INPUT word_stream 0
    OUTPUT frame_stream
    PROCESS_END

// contrast
// this process is on PE 2
PROCESS contrast PROCESSOR wf_b0pe2
    RUN_FUNCTION contrast_run
    INPUT frame_stream 0
    OUTPUT remap_stream
    PROCESS_END

Figure 3. Mapping Directives in Streams-C

ments to understand the parallel program’s concurrency behavior. The programmer can detect many potential deadlock and livelock conditions, and get a good approximation for buffer sizes required for correct program execution.

Our simulation tools use the /// annotations and SC macro calls to generate a C++ program that links the process function body with the simulation library. The generated C++ source program is then linked with our “ptstreams” library to produce a Linux executable that can run on the Linux workstation.

3 SYNTHESES COMPILER

As shown in Figure 4, a pre-processor converts the annotations and SC macro calls into pragmas that are parsed by our Streams-C compiler. Thus a single source file is the basis for both simulation and synthesis. The Streams-C compiler builds on the infrastructure built for the NAPA C compiler ([2]). This infrastructure, based on the SUFI compiler infrastructure and the Malleable Architecture Generator (MARGE) datapath generator includes extraction and scheduling of datapath blocks from the Abstract Syntax Tree (AST), pipelining of for loops, and generation of a control program to sequence the generated hardware blocks. Aspects of the compiler new to Streams-C include semantic validation of process and stream pragmas, pipelining and control of while loops, state machine generation for on-board sequencing, and an efficient stream communication hardware library, components of which are instantiated by the compiler to effect the desired stream communications.

3.1. Pragma Processing

The purpose of pragma processing is first to ensure that the process and stream declarations are consistent, and then to use the information to map each process onto the correct chip and to select the appropriate instance of a stream communication module for each stream used in the program. The compiler has no built-in information about the target architecture. This information is kept externally in header and architecture definition files, allowing the compiler to easily be re-targeted to a different board. The header files establish names for each FPGA chip and for each path between chips. The names are the link into an architecture definition file that describes stream library modules. For example, for the Wildforce architecture (our first target architecture), if a process is on P1 and writes a stream to a process on the host, the stream uses the hardware “FIFOout” connection on the Wildforce board, and a stream module is selected for instantiation on P1 that observes the FIFO protocol and connects to those pins on P1. Similarly, in the host process, the stream read calls the Wildforce driver to read FIFO1.
An architecture definition file also describes characteristics of memories accessible to the FPGAs. Read and write latencies for each memory are used by the pipeline scheduler, so that loops that access memory (as well as read/write streams) are pipelined correctly.

### 3.2. Instruction Sequencing

Within a process body, the compiler analyzes the AST and partitions the tree into datapath, encompassing basic blocks and pipeline blocks, and control flow. A basic block is simply a straight line section of code without branches. A pipeline block is an inner loop body of a loop with independent iterations (do-all loop) that can be pipelined by our pipelining algorithm. Control flow is used for loops that cannot be pipelined. We note that if-statements do not usually result in control flow, as the compiler converts if-statements into guards controlling execution of the if-body.

From the AST representation of the control flow, the compiler generates a state machine to sequence the basic and pipeline blocks of the data path. Conditional expressions are evaluated in the datapath and cause state changes in the sequencer. Figure 5 shows representative C code and how the compiler partitions it into basic and pipelined blocks. Numerals at the right margin indicate the instruction numbers of the corresponding FPGA code. Figure 6 shows a fragment of the generated state machine.

As shown in Figure 6, after Instruction 2 is dispatched to the datapath, the state machine cycles until the datapath signals completion. Then instruction 3 is dispatched.

Figure 7 shows a portion of the corresponding datapath.
Figure 8. Hierarchy Generated by the Streams-C Compiler

3.3. Generated Chip and Software Architecture

The compiler generates a complete VHDL architecture for each chip. The structure of the generated entities is illustrated in Figure 8. Each chip contains zero or more processes and zero or more stream modules. A process contains the instruction sequencer and the datapath module. Within a datapath module is an instruction decoder, zero or more pipeline controllers, and the datapath circuits. The stream module is specific to the size of the stream payload, the type of resource used by the stream (FIFO, Crossbar, Inter-PE bus), and the synchronization protocol (valid tag, buffered).

The compiler also generates a multi-threaded control program. Before initiating host processes, the control program loads the FPGA configuration bit files and initializes local memories. Then processes mapped to the conventional processor are initiated. These processes also communicate via streams. When a stream connects Host processes, the control program loads the FPGA configuration bit files and initializes local memories. Then processes mapped to the conventional processor are initiated. These processes also communicate via streams. When a stream connects Host processes, the control program loads the FPGA configuration bit files and initializes local memories. Then processes mapped to the conventional processor are initiated. These processes also communicate via streams.

4. COMPILER LIBRARIES

The Streams-C hardware libraries are written in synthesizable RTL-level VHDL with heavy use of generics. Depending on the context of use, the compiler instantiates specific modules from these libraries. The software libraries use the POSIX threads package pthreads to simulate software processes and streams communication in software.

4.1. Pipeline Control

A pipeline controller is instantiated whenever there is a pipeline instruction in the datapath. There are two types of pipeline controller: definite iteration, where the number of iterations is fixed at loop entry, and indefinite iteration, where it is not. There is an enable register with one bit for each level of the pipeline. The pipeline control module sets the appropriate bits in this shift register to control data path operations at each level.

A pipeline controller must first be initialized with

- the number of pipeline stages
- pipeline initialization interval
- number of flush cycles
- number of iterations (for definite iteration controller).

Then the pipeline instruction is issued by the sequencing state machine. The initiation interval determines the number of pipeline stages that are active at the same time. The pipeline enable register is a shift register, with a '1' being shifted in every "initiation interval" clock cycles. Since some of the operations occurring within a pipeline level might stall, the pipeline controller is also stallable. On a stall signal, the enable register is set to '0's. When the stall ends, the original contents of the enable register are restored.

Figure 9 shows the pipeline control structure. The Iteration Cnt register is used for definite iterations, and the Terminate signal is used for indefinite iterations. The PipeMaster register is used to restore the PipeEnable register after a stall.

4.2. Hardware Stream Library

The hardware stream library consists of approximately 30 modules. Each stream module uses a standard interface to communicate with the process on one side and the I/O port on the other side. The modules make use of the various Wildforce channel mappings such as nearest neighbor, Crossbar, FIFOs, or intra-FPGA communication. Two protocols, Valid Tag and Buffered, have been implemented. The Valid Tag protocol delivers a data item accompanied by
a one-bit tag every clock cycle. The tag indicates whether the data item is valid. The Buffered protocol fills a buffer on stream write and then empties the buffer on stream read. If the buffer nears full, the stream read module sends a signal to the stream write module to stall, which the stream write module propagates to the pipeline controller and instruction sequencer of the producing process.

5 CONTRAST ENHANCEMENT APPLICATION

In this section we present an application from the image processing domain that has been mapped onto the Wildforce board directly in VHDL as well as through Streams-C.

Histogram projection contrast enhancement is a well known image processing transformation to perform grayscale adjustment to pixels in an image. Using statistics of the image itself to control adjustments, the algorithm stretches contrast within the image to use the entire dynamic range of the display. It is commonly used in IR video enhancement.

The algorithm has several phases. First, a histogram of the input image is generated (Phase 1, Histogram Generation). For each grayscale pixel value in the 8-bit range, we count how many input pixels have that value. We find the total number N of grayscale values in the image. New grayscale values are then assigned, with the darkest grayscale value getting value 0, and the brightest grayscale value getting value N-1. Intermediate brightness pixels are given values in the range 0 through N-1. The new assignments are stored in a "contrast stretch table" (Phase 2, Contrast Stretch Table Generation). Next the image is remapped to the new grayscale values. The new grayscale value (n) for each input pixel is looked up in the contrast stretch table, and the value of the input pixel is multiplied by n/N, yielding a scaled value in the 8-bit range (Phase 3, Image Remapping). The new pixel value is then output.

There are many possible mappings of this simple algorithm to the Wildforce board. One limiting factor is that the entire image must be read in before histogram generation is complete. The Contrast Stretch Table cannot be generated until the histogram has been done. Thus the image must get stored in local memory. Another limiting factor is that there is a divide operation, which is commonly done by table lookup. If the table lookup technique is desired, another memory is needed to hold the lookup table. We use one memory to temporarily store the image and another to hold the lookup table. On Wildforce, the implication of this need for two memories is that phases 2 and 3 are on different chips, as Wildforce has a single memory per chip. Since phase 3 will output pixels of the transformed image, that operation must be put on a chip with output to the conventional processor. Finally, since we expect to use only two chips for processing an image, we can double the processing rate by alternating two sets of two chips: "even" image frames go to one set, and "odd" frames to the other set. All these considerations lead us to the mapping shown in Figure 10. Input comes from the host through the InputFIFO to P0. P0 sends a frame of data four pixels at a time over the crossbar to P2 for even frames and P3 for odd frames. Phases 1 and 2 are mapped to P2 and P3. Phase 3 occurs on P1 and P4, which then use their output FIFOs to send alternating frames back to the host process.

Written in Streams-C, this program consists of two host processes and 5 FPGA processes. The first host process reads images from disk and sends four pixels at a time to a "controller" process on P0. The controller simply forwards stream pixels onto the crossbar, which broadcasts the pixel stream to processes on P2 and P3, which perform phases 1 and 2. These processes send an input pixel plus the scaling amount in a 16-bit packet to P1 and P4, which do the table lookup and then send groups of four output pixels to a host process. The host process assembles the output frame and then writes it to disk.

The most computationally intensive processing is done on P2 and P3, which do the histogram generation and contrast table generation. For that design, the hand-crafted version used 18 percent of the chip, and runs at 40 MHz. The compiler generated version uses 57 percent of the chip, and runs at 20 MHz. Thus compiler overhead adds a factor of 3 to the area, and a factor of 2 to the clock frequency. In terms of design time, the hand done version took a month to get working, while the Streams-C version took a couple of days. This translates to a factor of 10 in productivity.
CONCLUSIONS

The Streams-C compiler provides a practical vehicle for developing high level language applications for FPGA boards. While the inventive part of application development, the mapping of computation onto the FPGA board, is done explicitly by the programmer, clock-cycle level design of hardware circuits is performed by the compiler. The compiler pipelines computation and manages stream synchronization. Compiler overhead is within a factor of 3x hand-crafted designs, with clock frequency within a factor of 2x. Productivity increases by a factor of 10 by using the Streams-C compiler and simulator.

ACKNOWLEDGEMENTS

This work was supported by DARPA through Contract DABT63-97-C-00693 to Sarnoff Corporation and by Los Alamos National Laboratory. Members of the Streams-C team include Jeff Arnold, Maya Gokhale, Mirek Kaliowski, and Janice Stone. The original contrast enhancement application was written by Jim Kaba of Sarnoff Corporation. The original definite iteration pipeline controller was written by Scott Karlin of Princeton University.

References