Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective

Jan Frigo
Los Alamos National Laboratory
Los Alamos, NM, 87545
jfrigo@lanl.gov

Maya Gokhale
Los Alamos National Laboratory
Los Alamos, NM, 87545
maya@lanl.gov

Dominique Lavenier
IRISA - CNRS
Campus de Beaulieu
35042 Rennes cedex - FRANCE
lavenier@irisa.fr

ABSTRACT

The Streams-C compiler ([5]) synthesizes hardware circuits for reconfigurable FPGA-based computers from parallel C programs. The Streams-C language consists of a small number of libraries and intrinsic functions added to a synthesizable subset of C, and supports a communicating process programming model. The processes may be either software or hardware processes, and the compiler manages communication among the processes transparently to the programmer. For the hardware processes, the compiler generates Register-Transfer-Level (RTL) VHDL, targeting multiple FPGAs with dedicated memories. For the software processes, a multi-threaded software program is generated.

The Streams-C language and compiler offer a very high level of expressivity for reconfigurable computing application development, particularly for stream-processing applications. We find this is reflected in productivity, for a factor of up to 10 times improvement in time to produce a program. However, use of the tool in the "real world" is predicated on performance: only if such a compiler can deliver performance comparable to hand-coded performance will it be used in practice.

This paper presents an application study of the Streams-C compiler. Four applications have been written in Streams-C and compiled to the AMC Wildforce board containing Xilinx 4036's. Those same applications have been hand-coded in a combination of RTL and structural VHDL. We compare performance of the generated code with the hand-optimized code. Our study shows that the compiler-generated designs are 1.37–4 times the area and 1/2–1 times the clock frequency of the hand designs. We find that the compiler, based on the SUIF infrastructure, can be greatly improved through various standard compiler optimizations that are not currently being exploited. Thus we are currently rewriting a public domain version of Streams-C to better optimize and target the Virtex chip.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA 2001, February 11-13, 2001, Monterey, CA.
Copyright 2001 ACM 1-58113-341-3/01/0002 ..$5.00

Keywords

FPGA, configurable computing, silicon compiler, FPGA design tools, high-level synthesis, hardware-software co-design

1. INTRODUCTION

Over the past ten years, Reconfigurable Computing has demonstrated factors of 10 to 100 speedup over conventional high performance workstations at relatively modest cost. Field Programmable Gate Array (FPGA)-based accelerator boards with customized hardware programmed into the FPGAs have been used in signal and image processing applications for real-time embedded computation.

A major drawback to the widespread use of Reconfigurable Computing has been the cost of developing applications for these parallel systems. Current state of practice is to use low level Hardware Description Language (HDL) to describe the circuits realizing an algorithm. Not only is this task extremely time consuming, but it requires hardware design expertise. Thus successfully fielding applications for Reconfigurable Computers typically requires a team of domain experts, software programmers, and hardware designers.

There has been considerable research interest in reducing design time for Reconfigurable Computer applications. Approaches have ranged from high-level optimization schemes, to low-level, technology-specific, optimized designs. Some examples of high-level optimization methods include: a C to HDL method for high-speed pipeline circuits, focusing on the exhaustive parts of an application such as loop and recursive programs([10]); and a pipeline vectorizaton([15])technique to optimize and pipeline candidate inner space loops for hardware acceleration. Other techniques map a MATLAB application([11]) to a distributed computing environment, use graphical programming ([13], [14]), employ automatic parallelization and synthesis ([7]) techniques. Low-level efforts target technology-specific, optimized designs ([2]) or automatic data storage and control ([12]) for computation.

In the Streams-C approach, we target an intermediate level of expression. Our compiler processes a subset of C suitable for automatic synthesis to FPGAs. Our programming model is targeted at stream-oriented reconfigurable computing applications. In this model, parallel processes communicate via data streams. We optimize compiler synthesis for high-rate flow of data streams; small, fixed size data packets; and low-precision fixed point computation, all characteristics of FPGA-based reconfigurable parallel processing applications. 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. Our compiler currently targets the Annapolis Microsystems Wildforce board, containing 5 Xilinx 4036 FPGAs and four banks of 32bit x 65K SRAM.

In the next section, we briefly review the Streams-C language. Next we describe the performance study methodology, describe each application, and compare performance between compiler-generated versus hand-optimized code. We end with a summary of results and a discussion of future work.

2. OVERVIEW OF STREAMS-C

The concept of stream-based computation is a fundamental formalism for high performance embedded systems, which is characterized by (multiple) streams of data produced at a high rate, with complex operations performed on the incoming data. The Streams-C [5, 3] language supports this computational model with a minimal number of language extensions and library functions callable from a C program. The compiler targets a combination of software and hardware.

For computation occurring in hardware, the compiler generates RTL VHDL for a target FPGA board containing multiple FPGAs, external memories, and interconnect. The language extensions, such as declarations for a process or stream, allocate resources on the board for these objects. These extensions allow the programmer to allocate registers on an FPGA and define register bit lengths; assign variables to memories; define concurrent processes; define stream connections between processes; and read/write streams to communicate data between processes. The processes operate asynchronously, and synchronize through stream operations, which may occur anywhere within the body of the process. A distributed memory model is followed, with local state belonging to each process and inter-process communication via streams. The extensions include mapping directives to give the applications developer control over the mapping of processes to hardware components and of streams to communication media on the target application board.

A hardware streams library has been built for the Annapolis Microsystems Wildforce accelerator board. The compiler, based on the Napa C compiler and Malleable Architecture Generator (MARGE), synthesizes hardware circuits from a C-language program. Although the target is a synchronous set of circuits on multiple communicating FPGAs, the C programmer does not have to be concerned with synchronizing state machines, or other hardware timing events. The compiler-generated state machines control sequencing and loops. The hardware streams library encapsulates the data flow synchronization between stream reader and writer. The combination of compiler-generated computation nodes with the hardware streams library allows applications developers to target FPGA boards from a high level concurrent language.

A software library using POSIX threads provides concurrent processes and stream support in software. Thus the software libraries support a dual function: when all processes are mapped to software, our system provides a functional simulation environment for the parallel program. When processes are mapped to a combination of software and hardware, the software libraries are used for communication among software processes and between software and hardware processes. Hardware libraries are used for communication among hardware processes and for the hardware side of communication to software processes. Figure 1 shows the software development flow for applications using the Streams-C compiler.

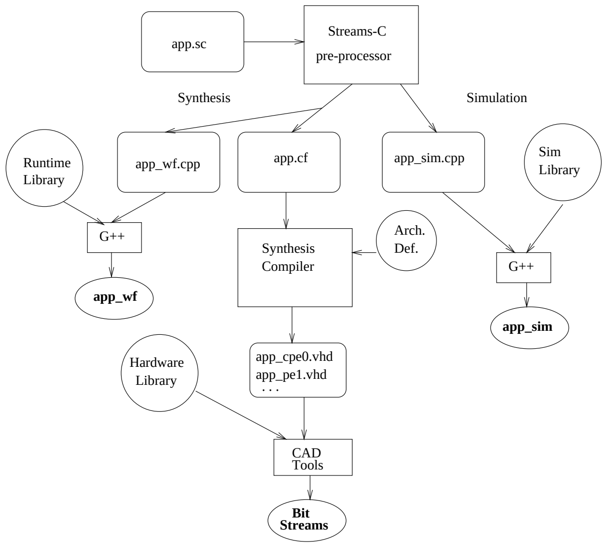

Figure 1: Organization of the Streams-C compiler: The application written in streams-C (app.sc) goes through the streams-C preprocessor which produces three descriptions: app_sim.cpp for simulation purpose, app.cf as an entry point for the synthesis compiler, and app_wf.cpp for running the host process. The synthesis compiler translates the app.cf file into a VHDL description for each processing element of the Wildforce board. Thus the VHDL files are applied to the Xilinx CAD tools which generate the bit-stream for the FPGA components.

3. APPLICATION STUDY

In this section, we will discuss four applications that have been mapped to reconfigurable hardware. These are

- contrast enhancement (previously reported in [6])
- polyphase filter bank[1]
- Pixel Purity Index (PPI) [8]
- K-means clustering [9]

The contrast enhancement algorithm is used for grayscale adjustments to pixels in an images. The polyphase filter bank[1], is a key component of the digital receiver architecture being developed by Los Alamos (see rcc.lanl.gov). The PPI and K-means clustering perform classification of features in multi- and hyper-spectral imagery.

Each of these applications was implemented with hand-coded VHDL and with the Streams-C compiler generated VHDL. The same target hardware, the Annapolis Microsystems Wildforce board, was used for each. This board consists of five Xilinx 4000 Series FPGAs ($X0 \ldots X4$), each with a dedicated SRAM. $X0, X1$ and $X4$ have bi-directional FIFO connections to the host processor. Each FPGA $X1 \ldots X4$ has direct connection to its immediate linear neighbor. In addition, all the FPGAs can communicate over a crossbar.

In this section, the algorithms will be described with respect to algorithm parallelization and mapping to the hardware. We compare hardware area utilization, speed and development time estimates for both versions.

3.1 Contrast Enhancement

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.

This algorithm was previously reported in [4], and thus we briefly summarize the results here. The algorithm has several phases. First, a histogram of the input image is generated (Phase 1, Histogram Generation) and the total number N of grayscale values in the image is computed. 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 by setting the new grayscale value (n) for each input pixel to be n/N, yielding a scaled value in the 8-bit range (Phase 3, Image Remapping). The new pixel value is then output.

In mapping to the Wildforce, Phases 1 and 2 are performed on a single chip, and Phase 3 on a different chip. Phase 1 reads the input pixel, updates the histogram table, and writes the pixel to memory. When all pixels of the image have been read in, Phase 2 assigns the "stretch" values, and reads the pixels back out of memory, passing the pixel and stretch amount to an adjacent chip. Phase 3 does the divide with a table lookup into its SRAM bank. The partitioning between two chips is driven by the fact that Wildforce only has one memory bank per FPGA chip, and Phase 3 needs a dedicated memory for the table lookup. Thus for maximum parallelism, two chips are used.

Written in Streams-C, this program consists of two host processes and 2 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 X2, which performs phases 1 and 2. The X2 process then sends an input pixel plus the scaling amount in a 16-bit packet to X1, which does the table lookup and then sends groups of four output pixels to a host process. The host process assembles the output frame.

The most computationally intensive processing is done on X2, the process performing histogram 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.

The hand-coded design outputs a result every clock cycle. In the compiler-generated design, a result is output every other clock cycle. This is because the hand-coded version uses the CLB RAMS to store the histogram table, and so can do two memory operations in one cycle (store pixel and store histogram value). While the compiler supports multiple memories and can produce a pipeline schedule with single-tick result generation, we have not yet made CLB RAMs accessible to the compiler, and thus had to sequentialize the memory writes to a single memory.

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.

3.2 Poly-phase Filter Bank

In the field of signal detection, multi-rate filter banks have been employed to help detect RF signals in noisy environments. By decomposing a signal into various frequency subbands, filter banks enhance many algorithms because they make it easier to identify pertinent material on a band by band basis. The polyphase implementation[1] is a multirate filter structure combined with a Fast Fourier Transform (FFT) designed to extract subbands from an input signal[1]. The polyphase filter portion of the structure is based on a prototype baseband lowpass Finite Impulse Response (FIR) filter with symmetric coefficients, i.e., the first $n/2$ and the last $n/2$ coefficents are the same, albeit in reverse order. The remaining filters of the filter bank are frequency shift versions of the prototype. The symmetry of this prototype filter combined with the structured frequency shifts allows for an optimal implementation of the filter bank. First, a prototype low-pass FIR filter, *h0[n]*, with the desired filter parameters is designed. The polyphase filters, *pk[n]*, are expressed in terms of the prototype filter,

$pk[n] = h0[k + lM]$ k = 0..M-1, $l = 0..L-1$

n is the length of the FIR prototype, M is the number of polyphase filters, L is the length of the individual polyphase filters, (L = n/M = 4). The FFT is used following the polyphase filtering structure to provide the frequency shifts for the various channels.

Figure 2 shows the parallelization for two polyphase symmetric filters and below is the Streams-C source code:

```
//coefficients for filter
#define C1 1
#define C2 117
#define C3 1741
#define C4 128

  SC_FLAG(tag);
  SC_REG(data, 32);
  SC_REG(data_o, 32);
  int s; //sample input data
  int in1, in2, in3, in4;
  int e1, e2, e3, e4;
  int o1, o2, o3, o4;
  int evenp; //flag for even or odd data
  int y1, y2; //filter output data

  while(SC_STREAM_EOS(input_stream) != SC_EOS) {
#pragma ALP pipeline
    //Get input data samples from the stream
    s = SC_REG_GET_BITS_INT(data, 0, 8);
    in1 = C1 * s;
    in2 = C2 * s;
    in3 = C3 * s;
```

[1]The filter structure was developed in collaboration with Prof. John Villesenor's team at UCLA.

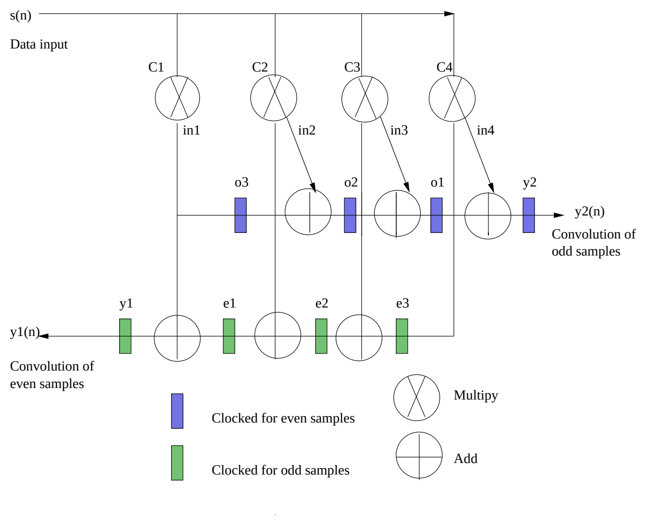

Figure 2: Poly Phase filter bank implementation

```
    in4 = C4 * s;

    if (evenp) {
      y1 = e1 + in1; //convolution of even data
      e1 = e2 + in2;
      e2 = e3 + in3;
      e3 = in4;
      SC_REG_SET_BITS_INT(data_o, 0, 16, y1);
    }
    else {
      y2 = o1 + in4; //convolution of odd data
      o1 = o2 + in3;
      o2 = o3 + in2;
      o3 = in4;
      SC_REG_SET_BITS_INT(data_o, 0, 16, y2);
    }
  evenp = !evenp;

    SC_STREAM_WRITE(output_stream, data_o, tag);
    SC_STREAM_READ(input_stream, data, tag);
   }
```

For comparison a filter bank of four is implemented on one chip, computing only the convolution of even data samples.[2] The input comes onto the FPGA via a stream from the host and is unsigned, fixed point, 8 bit data. The coefficents are unsigned, fixed point 12 bit values. The hand-coded design mapped a bank of four polyphase filters to 27% of the area at 40 MHz and the Streams-C version resulted in 37% area utilization for the same speed. We notice that the Streams-C compiler optimized away a multiplier when the coefficient, C1, for the multiplier was one, i.e. Streams-C implemented three multipies in the generated VHDL code. The hand-coded version relies on the synthesis tool, (Synplify in this case) to optimize the multiply operations. Both designs deliver a result every clock cycle. The manual version of the filter took about two weeks to implement on the hardware while Streams-C design-to-implementation took a few days, a development time savings of approximately 5 times.

[2]The loop as written is pipelinable, but our compiler does not yet pipeline loops containing "if" statements. This extension to the compiler is in progress.

3.3 Pixel Purity Index

The Pixel Purity Index (PPI) is an algorithm employed in remote sensing for analyzing hyperspectral images. Particularly for low-resolution imagery, a single pixel usually covers several different materials, and its observed spectrum is (to a good approximation) a linear combination of a few *pure* spectral shapes. The PPI algorithm tries to identify these pure spectra by assigning a pixel purity index to each pixel in the image; the spectra for those pixels with a high index value are candidates for basis elements in the image decomposition.

The algorithm proceeds by generating a large number of random D-dimensional vectors, called skewers, through the hyperspectral image. For each skewer, every data point is projected onto the skewer, and the position along the skewer is noted. The data points which correspond to extrema in the direction of a skewer are identified, and placed on a list. As more skewers are generated, this list grows. The number of times a given pixel is placed on this list is also tallied. The pixels with the highest tallies are considered the most pure, and the pixel's count provides its pixel purity index.

Most of the execution time of the PPI algorithm is spent in computing dot-products between the pixels and the skewers. These dot-product are highly independent and could be done simultaneously. This leads to many ways to parallelize the algorithm, but our approach targets the limited resources available on real FPGA boards. A sequential version of the Pixel Purity Index algorithm [8] is:

```
  PIXELS[N][D]; // an image of N hyperpixels
  SKEWER[K][D]; // a set of K random skewers
  PPI[N];       // the PPI result

// reset pixel purity index
  for (n=0; n < N; n++) PPI[n]=0;
  for (k=0; k < K; k++)  // K skewers
   {
     dpmax=MIN_INT; dpmin=MAX_INT;
     for (n=0; n < N; n++) // N pixels
      {
// compute a Dot-Product
        dp = 0;
        for (d=0; d < D; d++)
           dp = dp + SKEWERS[k][d]*PIXELS[n][d];
// detect extrema
        if (dp > dpmax) { imax=n; dpmax=dp; }
        if (dp < dpmin) { imin=n; dpmin=dp; }
      }

     // update PPI
     PPI[imax]++;
     PPI[imin]++;
   }
```

For each skewer, N dot-products are computed to determine the two pixels which produce the largest and the smallest dot-product. The pixel index (`PPI` vector) is modified accordingly. A pixel `n` is a candidate to be a pure pixel if `PPI[n]` has a high value.

From the above description it can easily be seen that all the dot-products can be computed independently: there are no dependencies between any of them. The parallelization takes advantage of this by computing $KS \times NS$ dot-products simultaneously, where KS and NS represent respectively the number of skewers and pixels which can be processed in parallel. This mapping to the hardware is shown in Figure 3.

The skewer data is represented as signed 3 bit data and is input via a stream from the host. The pixel data is unsigned,

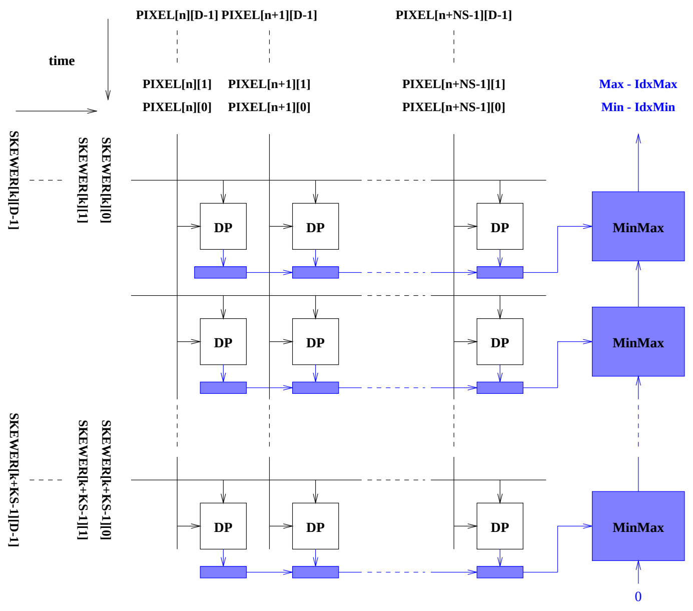

Figure 3: PPI algorithm architeture mapping to the Wildforce board

fixed point, 8 bit data which is stored in off-chip memory. We compare the most computationally intensive part of the algorithm, the dot product (DP), for both versions of this algorithm. The hand-coded version of the dot product maps 2 dot products at 25 MHz with an 22.5% area utilization of the chip after place and route. The Streams-C version has 100% chip area utilization for two dot products at a speed of 15 MHz. Since the main pipelineable loop contains “if” statements, we are not able to pipeline the loop. However, manual application of our extended pipeline algorithm to the loop yields a schedule that delivers an output every clock cycle. The hand-coded algorithm also has this throughput. The hand-coded version was manually placed and the dot product units were manually packed into CLBs. The hand-coded version took six weeks of development time while the Streams-C approach took four to five days, a productivity speed up of 6 times.

3.4 K-means Custering Algorithm

The basic principle of the image clustering process is to take an original image and to represent the same image using only a small number of pixel values[9]. The K-means clustering algorithm performs this task by attempting to minimize a cost function (the absolute value of a difference) over a set of NB_CLASS cluster centers. First, the algorithm assigns pixels randomly to NB_CLASS classes, computes the centers of the classes. There is a outer loop for a number of iterations, N, which can be either fixed in advance or undetermined, and an inner loop which scans all the pixels. For each pixel we check if it still belongs to its class. If not, the pixel is moved to another class and the two centers, corresponding to both the new and the old classes, are updated. The number of pixels in a class is stored as well as the sum accumulation necessary for recomputing the class centers. The class centers are periodically updated every block of B pixels.

The computation can roughly be split into three parts:

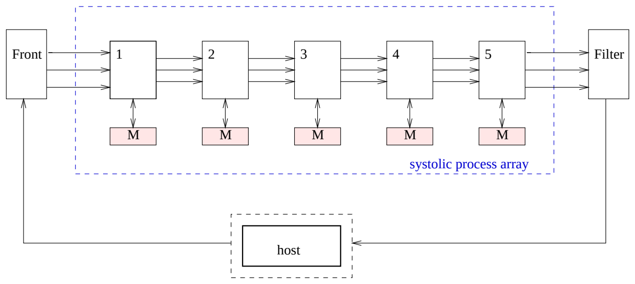

Figure 4: K-means hardware implementation

the distance calculation between a pixel and a class center, the accumulator update and the center update. The most time consuming part of the algorithm is the distance computation between the pixels and the class centers, even if the class center is freqently updated. The accumulator and the class center updates represent only a small percentage of the total computation time, especially for a partition into a large number of classes. For example, for a class partition of 32, the distance computation represents more than 99.6 % of the computation time.

Our architecture focuses only on parallelizing the most time consuming part, that is the distance computation between the pixels and the class centers. The idea is to flow a pixel stream through a linear array of processors. The number of processors is equal to the number of classes. A processor k computes a distance between the class k and the current flowing pixel. The result is taken at the rightmost end of the array by the filter process and corresponds to the index class for which a minimum distance has been found. The algorithm mapping to the Wildforce is shown in Figure 4.

Each processor has a small memory storing the class center (a vector of NB_BAND values), and performs the following computation:

```
index = my_processor_number;
while (! end_of_stream) {
  dist = 0;
  for (d=0; d<NB_BAND; d++) {
    stream_read (pixel_value);
    dist = dist + ABS(pixel_value - class_center[d]);
    stream_write (pixel_value);
  }
  stream_read (left_dist, left_index);
  if (dist < left_dist) {
    left_dist = dist; left_index = index;
  }
  stream_write (left_dist, left_index);
}
```

The above code does not compute class centers: it only determines the class number of a pixel. This information is available at the rightmost end of the array each time a pixel (its last vector element) comes out of the array. The host processor is in charge of flushing the pixel stream to the array, getting the results indicating the class number of each pixel, and if a pixel has moved, recalculating the class center accordingly.

The input data (pixel and class centers) comes to the FPGA board from the host as previously described. The Streams-C version implements one processor per this algorithm. It utilizes 14 percent of the area on the chip at a speed of 20 MHz. For a pipelined Streams-C implementation of this algorithm, an output every clock cycle is expected. In comparision, the hand-coded version uses 9.4 percent at the same speed. The Streams-C application took one day of development time, and the hand-coded version took one to two weeks.

4. SUMMARY

The results of place and route with the Xilinx 4036 architecture are shown in Figure 5. The results for the pixel purity index, and the contrast enhancement implementations show that Streams-C generally has a two to three times increase in area utilization on the chip at about one half the clock speed compared to the hand-coded versions. The time savings for implementation is approximately 5 to 10 times in favor of Streams-C. The efficiency of Streams-C compared to hand-coding depends greatly on how the algorithm is parallelized and what operations are mapped to the hardware. For example, the K-means clustering application shows that functions such as addition or subtraction can be automated rather efficiently by Streams-C, without much increase in area utilization or clock speed compared to the manual implementation. The Streams-C productivity increase for the Kmeans implementation is a speed up of 12 times. Key to the success of faster algorithm run-time is assessing the algorithm, parallelizing the computationally expensive processes, and mapping them to hardware. This was accomplished successfully with the Kmeans clustering method by using the host to do part of the processing, and the hardware to do the computationally expensive parts of the algorithm.

Operations such as multiplication are not optimized for synthesis or place and route in this implementation of Streams-C. For this case, manual implementation decreases area utilization and improves speed, for example, in the poly-phase filter application, Streams-C uses 37% of the total area of the chip. Three multipliers are synthesized and one multiplier is optimized away by the compiler, otherwise the area utilization would be almost double that of the hand-coded version. In addition, Streams-C does not yet handle local arrays without accessing off-chip memory which could make a large improvement to area utilization and speed for algorithms like the pixel purity index and contrast enhancement. The continuing Streams-C compiler research will manage local arrays internal to the chip.

The synthesis results for the Virtex 1000 architecture are shown in Figure 6. The Virtex has 27,648 logic cells, 96x64 array of configuration logic blocks (CLBs), the functional elements for constructing user logic. In comparison, the Xilinx 4036E has 3078 logic cells and 1296 CLBs. For technology such as the Virtex series architecture, if your objective is for fast turn around time for alternative implementations of reconfigurable components, Streams-C benefits the user via the development time savings. It is usually not feasible to hand-code alternative implementations without such a tool.

The future version of the Streams-C compiler will handle local, on-chip memory, pipeline loops with control flow, arrays of processes on-chip, and variable length data type declarations, all of which should help optimize the area utilization and speed of an application.

Reconfigurable Computing is a well known speed up over conventional software system implementations, for example, the results of the Pixel Purity Index (PPI) [8] show the FPGA implementation speed up of 80 times over the software implementaton. The objective of this study is to analyze the performance of the Streams-C compiler with respect to optimized hand-coded designs for practical applications in the image and signal processing domain. Four different applications were choosen, in order to show that the parallelization of the algorithm, and efficient hardware mapping impacts the run-time speed of the application. Streams-C benefits the user via faster development time, but area utilization is the penalty.

This application study shows that C-to-hardware technology in conjunction with a parallel programming model and efficient hardware libraries is within reach for eventual production use. Our future work is directed toward a new Streams-C implementation that will be available as modules within the SUIF compiler framework.[3] In addition to restructuring the compiler phases to better exploit standard optimizations, we plan to add support for arrays of processes and streams and for block and CLB RAMS on a variety of boards targeting the Virtex chip.

Xilinx 4036 Architecture

	Streams-C VHDL			Handcoded VHDL		
	Area %	Speed MHz	Time wks	Area %	Speed MHz	Time wks
CE	55	20	1/2	18	40	4
PPF	37	40	1/2	27	40	2
PPI DPs 4x2	100	15	1	22.5	25	6
Kmeans	14	20	1/4	9	20	1-2

Figure 5: Place and route results for the Xilinx 4036 on the Wildforce board comparing the Streams-C versus handcoded VHDL.

Virtex V1000 Architecture

	Streams-C VHDL		Handcoded VHDL	
	Area %	Speed MHz	Area %	Speed MHz
CE	3	40	1	40
PPF	1	40	1	40
PPI DPs 4x2	6	40	2	45
Kmeans	< 1	40	< 1	40

Figure 6: Synthesis results for the Virtex X1000 comparing the Streams-C versus the handcoded VHDL .

5. ACKNOWLEDGEMENTS

The application study was supported by Los Alamos National Laboratory's Deployable Adaptive Computing Systems (DAPS) program. The Streams-C compiler development was primarily supported by the DARPA Adaptive Computing Systems Program, as was the polyphase filter algorithm. The hand-coding of PPI and K-Means was supported by a DoD grant to LANL.

[3](see suif.stanford.edu)

6. REFERENCES

[1] Joseph Arrowood. Comparison of filter banks for signal detection. In *LAUR Number 99-4551*, Los Alamos, NM, March 2000.

[2] Xilinx Corp. http://www.xilinx.com/xilinxonline/jbits.htm. 1999.

[3] M. B. Gokhale, J. Frigo, and J. Stone. Parallel c programming of reconfigurable computers: the Streams-C approach. In *HPEC 2000*, September 2000.

[4] M. B. Gokhale and J. M. Stone. Co-synthesis to a hybrid RISC/FPGA architecture. *Journal of VLSI Signal Processing Systems*, 24, March 2000.

[5] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE international Symposium on FPGAs for Custom Computing Machines*, 2000.

[6] Maya Gokhale, Janice Stone, and Edson Gomersall. Co-synthesis to a hybrid risc/fpga architecture. *Journal of VLSI Signal Processing Systems*, September 2000.

[7] Mary Hall et al. Defacto: A design environment for adaptive computing technology. *Proceedings of the 6th Reconfigurable Architectures Workshop (RAW'99)*, 1999.

[8] Dominique Lavenier, James Theiler, John Szymanski, Maya Gokhale, and Janette Frigo. Fpga implementation of the pixel purity index algorithm. In *SPIE, FPGAs and Reconfigurable Processors for Computing and Applications, vol 4212*, Boston, MA, November 2000.

[9] Miriam Leeser. Applying reconfigurable hardware to segmentation for multispectral imagery. In *HPEC 2000*, Boston, MA, September 2000.

[10] T. Maruyama and T. Hoshino. A c to hdl compiler for pipeline processing on fpgas. In *FCCM 00*, Napa, CA, April 2000.

[11] et. al. P. Banerjee. A matlab compiler for distributed, heterogeneous, reconfigurable computing systems. In *FCCM 00*, Napa, CA, April 2000.

[12] J. Park P. Diniz. Automatic synthesis of data storage and control structures for fpga-based computing engines. In *FCCM 00*, Napa, CA, April 2000.

[13] Eric Pauer, Paul Fiore, John Smith, and Cory Myers. Algorithm analysis and mapping environment for adaptive computing systems. *FPGA2000*, 2000.

[14] S. Periyayacheri et al. Library functions in reconfigurable hardware for matrix and signal processing operations in matlab. *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conference (PDCS'99)*, November 1999.

[15] Markus Weinhardt and Wayne Luk. Pipeline vectorization for reconfigurable systems. In *FCCM 99*, April 1999.