In Proceedings of the IEEE Workshop on Embedded Computer Vision, San Diego, California, June 2005

Computer Vision on FPGAs: Design Methodology and its Application to Gesture Recognition

Mainak Sen*, Ivan Corretjer*, Fiorella Haim*, Sankalita Saha*, Jason Schlessman†, Shuvra S. Bhattacharyya*, and Wayne Wolf†

*Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD, 20742, USA.
†Department of Electrical Engineering,
Princeton University, Princeton, NJ, 08544, USA.

Abstract

In this paper we develop a design methodology for generating efficient, target specific Hardware Description Language (HDL) code from an algorithm through the use of coarse-grain reconfigurable dataflow graphs as a representation to guide the designer. We demonstrate this methodology through an algorithm for gesture recognition that has been developed previously in software [9]. Using the recently introduced modeling technique of homogeneous parameterized dataflow (HPDF) [3], which effectively captures the structure of an important class of computer vision applications, we systematically transform the gesture recognition application into a streamlined HDL implementation, which is based on Verilog and VHDL. To demonstrate the utility and efficiency of our approach we synthesize the HDL implementation on the Xilinx Virtex II FPGA. This paper describes our design methodology based on the HPDF representation, which offers useful properties in terms of verifying correctness and exposing performance-enhancing transformations; discusses various challenges that we addressed in efficiently linking the HPDF-based application representation to target-specific HDL code; and provides experimental results pertaining to the mapping of the gesture recognition application onto the Virtex II using our methodology.

1 Background and motivation

Computer vision methods based on real-time video analysis form a challenging and increasingly important domain for embedded system design. Due to their data-intensive nature, hardware implementations for real-time video are often more desirable than corresponding software implementations despite the relatively longer and more complicated development processes associated with hardware implementation. In this paper we propose a methodology for systematically deriving efficient hardware implementations for an important class of real-time video- and image processing applications. The methodology is developed here in the context of design representations and associated application mapping techniques to guide the hardware designer throughout the implementation process. Additional formalization and software infrastructure may also render it suitable for use in automated design tools. Our approach assumes that the targeted application can be represented through a particular form of dataflow graph, called a homogeneous parameterized dataflow (HPDF) graph. In a previous related work, HPDF graphs have been shown to effectively represent the behavior of a useful class of image and video processing applications [3].

2 Related work

A number of studies have been undertaken in recent years on the design and implementation of multimedia applications on FPGAs using formal or otherwise systematic approaches. For example, Streams-C [4] provides compiler technology that maps high-level, *parallel C* language descriptions into circuit-level netlists targeted to FPGAs. To use Streams-C effectively, the programmer needs to have some application-specific hardware mapping expertise as well as expertise in parallel programming under the CSP (Communicating Sequential Process) model of computation [12]. Streams-C consists of a small number of libraries and intrinsic functions added to a subset of C that the user must use to derive synthesizable HDL code.

Handel-C [5] represents another important effort towards developing a hardware oriented C language. Handel-C is based on a subset of the ANSI C standard along with extensions that support a synchronous parallel mode of operation. This language also conforms to the CSP model.

Match [1] or AccelFPGA as it is called now, generates VHDL or Verilog from an algorithm coded in MATLAB, a programming language that is widely used for prototyping image and video processing algorithms. AccelFPGA has various compiler directives that the designer can use to explore the design space for optimized hardware implementation. Loop unrolling, pipelining, and user-defined memory mapping are examples of implementation aspects that can be coordinated through AccelFPGA directives.

Compaan [6] is a another design tool for translating MATLAB programs into HDL for FPGA implementation. Compaan performs its translation through an intermediate representation that is based on the Kahn process network model of computation [7].

3 Problem definition

Rather than adapting a sequential programming language for hardware design, as the approaches described in Section 2 do, the approach that we pursue in this paper is based on direct representation by the designer of application concurrency using dataflow principles. Dataflow provides an application modeling paradigm that is well-suited to parallel processing (and to other forms of implementation streamlining) for signal, image, and video processing (DSP) systems [13]. Furthermore, we use the dataflow representation as a conceptual tool to be used by the designer rather than as the core of an automated translation engine for generating HDL code. This combination of a domain-specific model of computation and its use as a conceptual design tool rather than an automated one allows great flexibility in streamlining higher level steps in the design process for a particular application.

As an important front-end step in exploiting this flexibility, we employed HPDF (homogeneous parameterized dataflow) [3] semantics to represent the behavior of the targeted gesture recognition system. HPDF is a restricted form of dynamic dataflow that is at an exploratory stage of development and is not supported directly by any existing synthesis tools. However, an HPDF-based modeling approach captures the high-level behavior of our gesture recognition application in a manner that is highly effective for design verification and efficient implementation. As our work in this paper demonstrates, the HPDF-based representation is useful to the designer in structuring the design process and bridging the layers of algorithm and architecture, while synthesis tools play the complementary role of bridging the architecture and the target platform.

In the following two sections, we review, respectively, the HPDF model of computation and the targeted gesture recognition application in more detail.

4 HPDF

Homogeneous parameterized dataflow (HPDF) [3] is a metamodeling technique that was introduced recently to model DSP applications that involve dynamic dataflow between functional modules (*actors*). For some applications it was shown to be a simpler and more powerful alternative to a related metamodeling technique, called parameterized dataflow [2], that was developed earlier to represent reconfigurable dataflow graphs in a general manner.

Many applications involve actors whose data production and consumption rates are input-dependent, and cannot be determined at synthesis time. HPDF models this phenomenon by encapsulating variable amounts of data into individual vectors that are passed between modules as single units of statically-known, homogeneous "size" (a single vector value on each production or consumption operation). In many image and video processing applications, this simple, widely-applicable convention of aggregating certain blocks of variable-sized data values into vectors exposes high-level static structure that can be exploited at design time to help validate an implementation and improves its efficiency. For a more detailed discussion of HPDF principles, we refer the reader to [3].

5 Description of the gesture recognition algorithm

As a consequence of continually-improving CMOS technology, it is now possible to develop "smart camera" systems that not only capture images, but also process image frames in sophisticated ways to extract "meaning" from video streams.

One important application of smart cameras is gesture recognition from video streams of human subjects. In the gesture recognition algorithm discussed in [9], for each image captured, real-time image processing is performed to identify and track human gestures. As the flow of images is increased, a higher level of reasoning about human gestures becomes possible. This type of processing occurs inside the smart camera system using advanced very large scale integration (VLSI) circuits for both low-level and high-level processing of the information contained in the images. Figure 1 gives an overview of the smart camera gesture recognition algorithm.

The functional blocks of particular interest in this paper are the low-level processing Region, Contour, Ellipse, and Match portions (within the dotted rectangle in Figure 1). Each of these blocks operate at the pixel level to identify and classify human body parts in the image and are thus

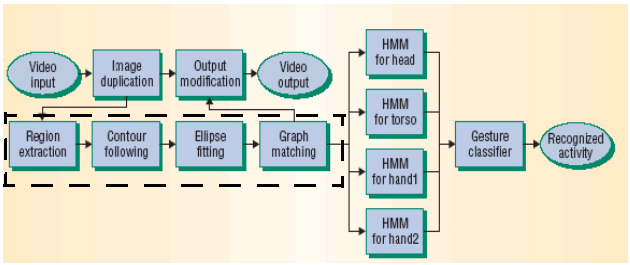

Figure 1: Block level representation of the smart camera algorithm [9].

good candidates for implementation on a high performance field-programmable gate array (FPGA).

The computational core of the block diagram in Figure 1 can be converted from being an intuitive flow diagram to a precise behavioral representation through integration of HPDF modeling concepts. This exposes significant patterns of parallelism and of predictability, homogeneous vector-dataflow as described in Section 4, which together help us to map the application efficiently into hardware as described below.

The front-end processing is performed by Region extraction, which accepts three images as inputs. One image has just background regions and is used in processing the two other images, which have foreground regions. In one of the foreground images, Region marks areas that are of human skin-tones, and in the other it marks areas that are of non-skin tone. Each of the images is independent of the other, revealing image-level parallelism. Additionally, within each image the individual pixels are all independent from one other, leading to pixel-level parallelism. Furthermore, the operations performed are of similar complexity suggesting that a synchronous pipeline implementation with little idle time between stages is possible.

After separating foreground regions into two images each containing only skin and non-skin tone regions respectively, the next processing stage that occurs is Contour following. Here, each image is scanned linearly pixel by pixel until one of the regions marked in the Region stage is encountered. For all regions in both images (i.e., regardless of skin or non-skin tone), the contour algorithm traces out the periphery of each region, and stores the (x, y) locations of the boundary pixels. In this way the boundary pixels making up each region are grouped together in a list and passed to the next stage.

The Ellipse fitting functional block processes each of the contours of interest and characterizes their shapes through an ellipse-fitting algorithm. The process of ellipse fitting is imperfect and allows for tolerance in the deformations caused during image capture (such as objects obscuring portions of the image). At this stage each contour is processed independent of the others revealing contour-level parallelism.

Finally, the Graph matching functional block waits until each contour is characterized by an ellipse before beginning its processing. The ellipses are then classified into head, torso, or hand regions based on several factors. The first stage attempts to identify the head ellipse, which allows the algorithm to gain a sense of where the other body parts should be located relative to the head. After classifying the head ellipse, the algorithm proceeds to find the torso ellipse. This is done by comparing the relative sizes and locations of ellipses adjacent to the head ellipse, and using the fact that the torso is usually larger by some proportion than other regions and that it is within the vicinity of the head. The conditions and values used to make these determinations are part of a piece-wise quadratic Bayesian classifier that only requires the six characteristic parameters from each ellipse in the image [9].

6 Modeling of algorithm

We initially prototyped the gesture recognition algorithm through HPDF representation in Ptolemy II [8], a widely-used software tool for experimenting with new models of computation and integrating different models of computation. This prototype was developed to validate our HPDF-based representation of the application, simulate its functional correctness, and provide a reference to guide the mapping of the application into hardware. In the top-level, the HPDF application representation contains four hierarchical actors (actors that represent nested subsystems) — Region, Contour, Ellipse and Match — as shown in Figure 2. The symbols on the edges represent the numbers of data values produced and consumed on each execution of the actor. Here n and p are parameterized data transfer rates that are not known statically. Furthermore, the rates can vary during execution subject to certain technical restrictions that are imposed by the HPDF model [3].

By examining the HPDF graph in conjunction with the intra-actor specifications (the actors were specified using Java in our Ptolemy II prototype), we derived a more detailed representation as a major step in our hardware mapping process. This representation is illustrated across Figure

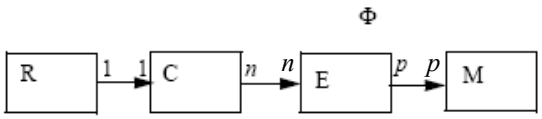

Figure 2: HPDF model of the application with parameterized token production and consumption rates where R is region, C contour, E ellipse and M Match.

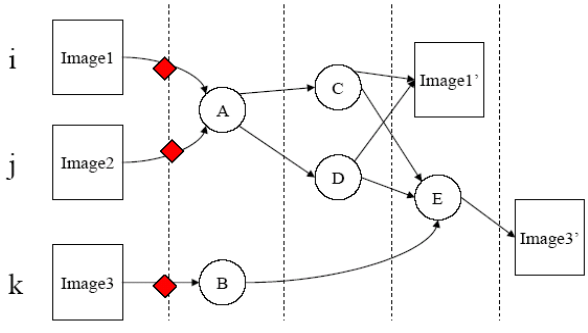

Figure 3: Region is shown to be broken into a four stage pipeline process.

3 and Figure 4. Figure 3 gives a schematic model of Region. As in general dataflow diagrams, the round nodes (A, B, C, D, and E) represent computations, and the edges represent unidirectional data communication. The square nodes represent image buffers or memory, and the diamond nodes on edges represent initial data values (or equivalently, delays in the associated data streams). The representation of Figure 3 reveals that even though buffers *Image1* and *Image3* are being read and written into, the reading and writing occur in a mutually non-interfering way. Furthermore, separating the two buffers makes the four stage pipeline implementation a natural choice.

In Contour (Figure 4), the dotted edges represent *conditional data transfer*. In each such conditional edge, zero or one data item can be produced by the source actor depending on its input data. More specifically, in Figure 4 there will either be one data value produced on the edge between A and B or on the self looped edge, and the other edge will have zero data items produced. The representation of Figure 4 and its data transfer properties motivated us to map the associated functionality into a four stage, self-timed process.

The ellipse module utilizes floating-point operations to fit ellipses to the various contours. The original C code algorithm uses a moment-based initialization procedure along with trigonometric and square root calculations. The initialization procedure computes the averages of the selected contours pixel locations and uses these averages to compute the various moments. The total computation cost is:

$$5nC_{+} + 6nC_{-} + 3nC_{*} + 5C_{/},$$

where n is the number of pixels in the contour, and each term C_{OP} represents the cost of performing operation OP. In an effort to save hardware and reduce complexity, the following transformation was applied to simplify the resulting hardware to calculate the averages and moments:

$$mxx = \left(\sum_{i=1}^{n} \frac{(x_i - \bar{x})^2}{n} \right) \Rightarrow \left(\sum_{i=1}^{n} \frac{(x_i)^2}{n} - \overline{(x)^2} \right)$$

and similarly for mxy and myy. The computation complexity after the transformation is:

$$5nC_{+} + 3nC_{*} + 9C_{/} + 3C_{-} + 3C_{*}.$$

Comparing this with the expression for the previous version of the algorithm, we observe a savings of $3nC_{-}$, which increases linearly with the number of contour pixels, at the expense of a fixed overhead $4C_{/} + 3C_{*}$. This amounts to a large overall savings for practical image sizes.

Further optimizations that were performed on the ellipse-fitting implementation included splitting the calculations up into separate stages. This allowed for certain common values to be computed in earlier stages and reused in later stages to remove unnecessary computations.

The characterization of ellipses in Match is accomplished in a serial manner, in particular, information about previously identified ellipses is used in the characterization of future ellipses. Our functional prototype of the matching process clearly showed this dependency of later stages on previous stages. The hardware implementation that we derived is similar to that of Contour, and employs a a six-stage self-timed process to efficiently handle the less predictable communication behavior.

7 Experimental setup

The target FPGA board chosen for this application is the multimedia and microblaze development board from Xilinx. The board can act as a platform to develop a wide variety of applications such as image processing and ASIC prototyping. It features the XC2V2000 device of the Virtex II family of FPGAs.

Some of the more important features of the board include the following.

- Five external, independent 512Kx36 bit ZBT RAMs
- A video encoder-decoder.
- An audio codec.
- Support for PAL/NTSC TV input/output.
- On-board ethernet support.
- An RS-232 port.
- Two PS-2 serial ports.
- A JTAG port.
- A System ACE-controller and Compact Flash storage device to program the FPGA.

7.1 ZBT memory

One of the key features of this board is its set of five fully-independent banks of 512k x 32 ZBT RAM [11] with a maximum clock rate of 130 MHz. These memory devices support a 36-bit data bus, but pinout limitations on the FPGA prevent the use of the four parity bits. The banks operate completely independently of one another, as the control signals, address and data busses and clock are unique to each bank with no sharing of signals between the banks. The byte write capability is fully supported as it is the burst-

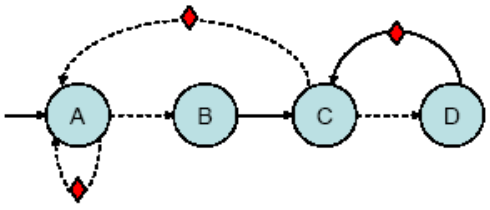

Figure 4: Contour shown to have *conditional edges* and serial execution. Implemented as a four stage self-timed process.

mode, in which the sequence starts with an externally supplied address.

Due to the size of the images, we needed to store them using these external RAMs. A memory controller module was written in Verilog, simulated, synthesized, and downloaded onto the board. We then successfully integrated this module with the Region module.

7.2 RS-232

In order to communicate between the host PC and the board, we chose to use the RS-232 protocol. We adapted a RS232 controller core with a wishbone interface and configurable baud rate [14] [15] to write images from the PC to the memory. The board acts as a DCE device; we implemented the physical communication using a straight-through three wire cable (pins 2, 3 and 5) and used the Windows Hyperterminal utility to test it. This interface was successfully integrated to the Region and Memory Controller modules and tested in the board.

Figure 5 illustrates the overall experimental setup, including the interactions between the PC and the multimedia board, and between the board and the HDL modules.

8 Design Tradeoffs and Optimizations

There were various design decisions made during implementation of the algorithm, some of which were specific to the algorithm at hand. In this section, we explore in more detail the tradeoffs that were present in the important design space associated with memory layout. We also present a step-by-step optimization that we performed on one of the design modules for reducing its resource requirements on the FPGA.

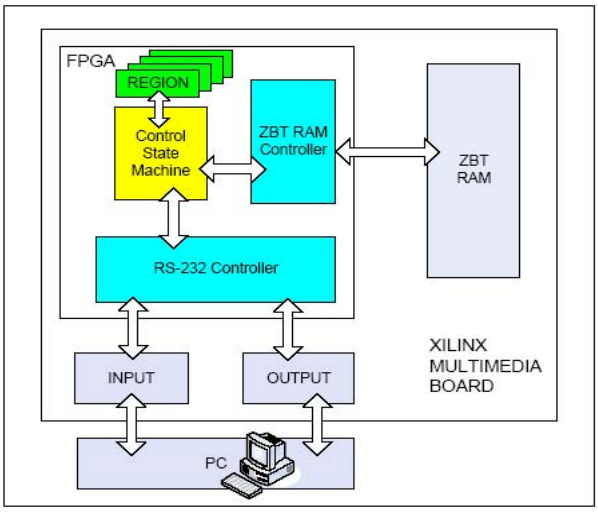

Figure 5: The overall setup showing interaction between various modules of our design and components of the multimedia board.

8.1 Memory layout tradeoffs

The board memory resources are consumed by the storing of the images. Each of the 5 ZBT RAM banks can store 512 K words that are 32 bits long, for a total storage capacity of 10 Mbytes. Given that each pixel requires one byte of storage and that there are 384 x 240 pixels per image, 90 Kbytes of memory are required to store each image. The first module, Region, has 3 images as inputs, and 2 images as outputs. These two images are scanned serially in the second module, Contour. The total amount of memory needed for image storing is then 450 Kbytes, less than 5% of the external memory available on board. However, reorganization of the images in the memory can dramatically change the number of memory access cycles performed and the number of banks used. These trade-offs also affect the total power consumption.

Several strategies are possible for storing the images in the memory. The simplest one (Case 1) would be to store each of the five images in a different memory bank, using 90 K addresses and the first byte of each word. In this way, the 5 images can be accessed in the same clock cycle (Figure 6a). However, we can minimize the number of memory banks used by exploiting the identical order in which the reading and writing of the images occurs (Case 2). Thus, we can store the images in only two blocks, using each of the bytes of a memory word for a different image, and still access all the images in the same clock cycle (Figure 6b).

On the other hand, a more efficient configuration in order to minimize the number of memory access cycles (Case 3) would be to store each image in a different bank, but using the four bytes of each memory word consecutively (Figure 6c). Other configurations are possible, for example, (Case 4) we can have two images per bank, storing 2 pixels of each image in the same word (Figure 6d). Table 2 summarizes the number of banks and memory access cycles needed for each of these configurations.

Case 3 appears to be the most convenient memory organization form. Here, the time associated to the images reading and writing is 69120 memory access cycles, and the total number of memory access cycles is also the lowest, 161280. This reduced number of memory access cycles suggests that power consumption will also be relatively low in this configuration. Figure 6 illustrates all the cases we discussed.

8.2 Floating point optimizations

Floating-point operations are used throughout the implementation of the Ellipse and Match functional blocks. The

Ellipse functional block processes the (x, y) location of every pixel that is along the border of a contour. From these locations averages, moments, and rotation parameters are derived that characterize a fitted ellipse to the particular contour. Due to the non-uniform shape of the contours, the ellipse fitting is imperfect and introduces some approximation error. By representing the parameters using floating point values, the approximations made have more precision than if integer values were used. To further motivate the need for floating point numbers, the Match functional block uses these approximations to classify each ellipse as a head, torso, or hand. To do so, the relative locations, sizes, and other parameters are processed to within some hard-coded tolerances for classification. As an example the algorithm considers two ellipses, one being around X times larger than the other, within a distance Y of each other to be classified as a head/torso pair. It is because of the approximations and tolerances used by the algorithm that floating-point representations are desirable as they allow the algorithm to operate with imperfect information and still produce reasonable results.

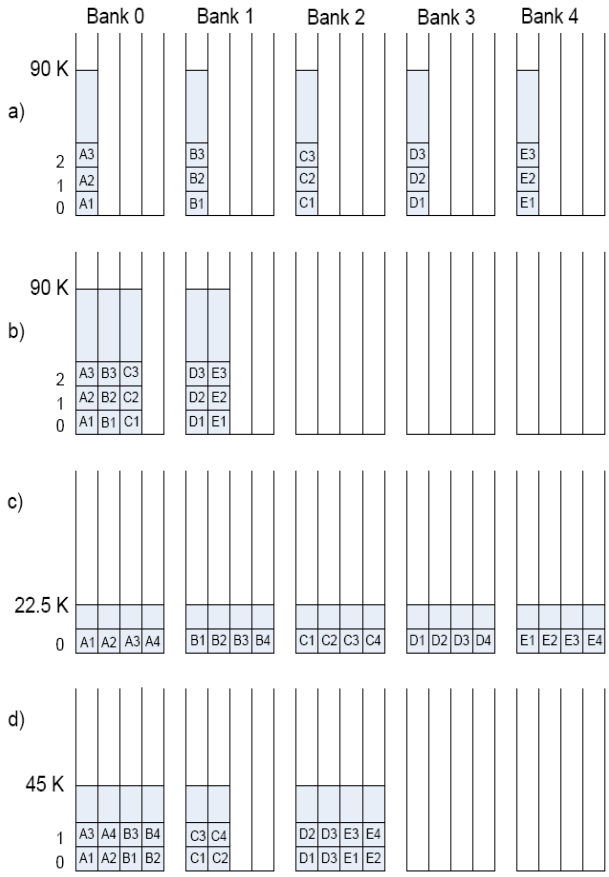

Figure 6: Image storage distribution a) Case1: Each image in a separate bank using only first byte of first 90K words b) Case2: Three images in bank 0 and two in bank 1 c) Case3: Each image in separate bank but all four bytes used in each word, using 22.5K words d) Case4: Images stored in three banks, each using 2 bytes of the first 45K words

For our implementation, we used the IEEE 1076.3 Working Group floating-point packages, which are free and easily available from [10]. These packages have been under development for some time, have been tested by the IEEE Working Group, and are on a fast track to becoming IEEE standards. Efficient synthesis of floating point packages involved the evaluation of floating-point precision required by the smart camera algorithm. The C code version of the algorithm utilizes variables of type `double`, which represent 64-bit floating-point numbers. Utilizing the floating-point library mentioned before, we were able to vary the size of the floating-point numbers to see how the loss in precision affected the algorithm outputs as well as the area of the resulting synthesized design.

We reduced the number of bits used in the floating-point number representation and performed a series of simulations to determine the loss in accuracy relative to the original 64-bit algorithm. Figure 7 shows the resulting RMS for various sizes of floating-point numbers. For the smart camera algorithm, we found that the range from 20 to 18 bit floating-point number representations gave sufficient accuracy and any lower precision (such as 16-bit) caused a dramatic increase in the errors. The values that are most affected by the loss in precision are rotX, aX, and to some extent aY. These values depend on the computation of the arctangent function. As the precision is lowered, small variations cause large changes in the output of arctangent. The dxAvg and dyAvg parameters are not as affected by the loss in precision as the only computations they require are addition and division. Table 1 presents the area in number of look-up tables required for each of the floating-point number representations. As expected, when we reduce the number of bits, the area of the resulting design decreases, but at the cost of lost precision.

Table 1: Synthesis results

Number of bits	Area (in LUTs)
32-bit	110092
21-bit	54944
20-bit	46951
18-bit	41088
16-bit	23923

9 Some simulation outputs

In this section, we present some representative simulation results from the HDL modules. Figure 8 and Figure 9 show the outputs of the first two blocks after they were implemented in HDL.We used Modelsim XE II 5.8c for HDL simulation, Synplify Pro 7.7.1 for synthesis of modules that used floating point and Xilinx ISE 6.2 for rest of the synthesis and for downloading the bitstream into the FPGA. Figure 10 illustrates the Modelsim simulation and shows signals on various input and output lines of one ZBT memory bank when it is being written with four different data items into four bytes of address zero.

10 Conclusions

In this paper, we have developed and applied a novel design methodology for effective platform-specific FPGA implementation of computer vision applications. In this work, we applied the homogeneous parameterized dataflow graph (HPDF) representation format to model a gesture recognition algorithm that exhibits dynamically-varying data production and consumption rates between certain pairs of key functional components. The top level HPDF model and subsequent intermediate representations that we derived from this model naturally suggested efficient hardware architectures (e.g., synchronous pipelined, self-timed, or a combination) for implementation of the main subsystems. The HDL code for the four modules of the algorithm was developed following these suggested architectures. The modules were verified for correctness, and synthesized to target a multimedia board from Xilinx. Memory management and floating point handling also played a major role in our design process. We explored various trade-offs in these dimensions and integrated our findings seamlessly with the architectural decisions described above.

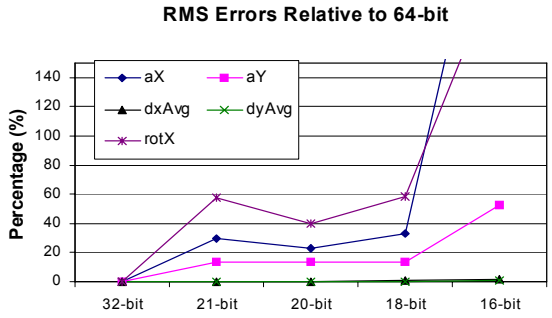

Figure 7: Comparison of percentage RMS error for different length floating point representations normalized to 64-bit floating point representation.

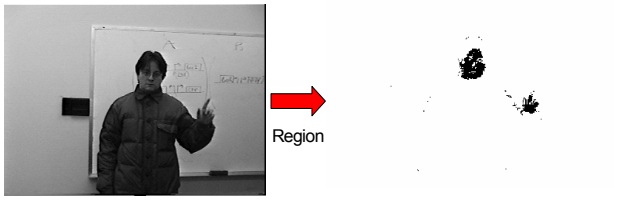

Figure 8: HDL representation of Region transforms the image on the left to the output on the right.

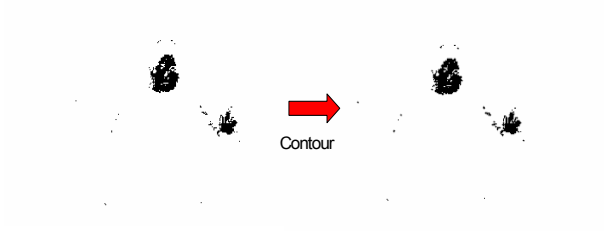

Figure 9: Actual transformation to the image done by HDL representation of Contour.

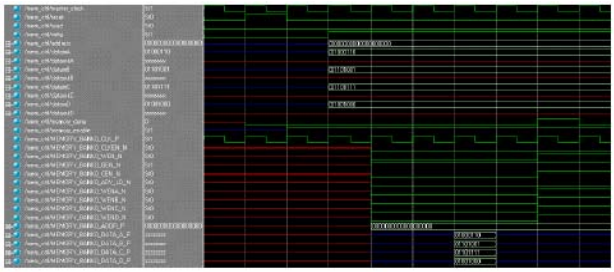

Figure 10: Modelsim simulation screenshot of 4 different data items being written into 4 bytes of addr 0.

Acknowledgement

This research was supported by grant number 0325119 from the U. S. National Science Foundation.

References

[1] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe, *Automatic Conversion of Floating-point MATLAB Programs into Fixed-point FPGA based Hardware Design*, Proceedings of the 41st Annual Conference on Design Automation, pp. 484-487, 2004.

[2] B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 84-89, Paris, France, June 2000.

[3] M.Sen, S. S. Bhattacharyya, T. Lv, W. Wolf. Modeling Image Processing Systems with Homogeneous Parameterized Dataflow Graphs. ICASSP 2005.

[4] M. Gokhale et. al. Stream-oriented FPGA computing in the Streams-C High Level Language. In IEEE international symposium on Field-Programmable Custom Computing Machines.

[5] S. Chappell, C. Sullivan. Handel-C for co-processing & co-design of Field Programmable System on Chip. White paper, Sept 2002.

[6] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. Presented at the 8th International Workshop on Hardware/Software Codesign (CODES'00), May 2000, San Diego. CA.

[7] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*, 1974.

[8] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong. Taming heterogeneity - the ptolemy approach. Proceedings of the IEEE, January 2003.

[9] W. Wolf, B. Ozer, T. Lv. Smart cameras as embedded systems. IEEE Computer Magazine Vol 35, Iss 9, Sept 2002, Pages 48-53.

[10] IEEE Working Group, http://www.eda.org/vhdl-200x/vhdl-200x-ft/packages/files.html.

[11] Data-sheet for ZBT memory,http://www.samsung.com/Products/Semiconductor/SRAM/SyncSRAM/NtRAM_FT_n_PP/16Mbit/K7N163631B/ds_k7n16xx31b_rev04.pdf

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[13] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker, Inc., 2000.

[14] http://jonathan-lawrence.co.uk/vhdl-march.php?p=2

[15] http://www.opencores.org/projects.cgi/web/miniuart2/overview.

Table 2: Comparison of different memory layout strategies

Configuration	Banks Used	Read cycles-Region	Write cycles-Region	Read cycles-Contour	Total non-overlapping cycles	Total number of cycles
Case 1	5	92160X3	92160X2	184320X1	276480	645120
Case2	2	92160X1	92160X1	184320X1	276480	368640
Case3	5	23040X3	23040X2	46080X1	69120	161280
Case4	3	46080X2	46080X1	92160X1	138240	230400