Mapping of Generalized Template Matching onto Reconfigurable Computers

Xuejun Liang and Jack Jean

***Abstract*—Image processing algorithms for template matching, 2D digital filtering, morphologic operations, and motion estimation share some common properties. They can all benefit from using reconfigurable computers that use co-processor boards based on FPGA (field programmable gate array) chips. This paper characterizes those applications as generalized template matching (GTM) operations and describes the mapping of the GTM operations onto reconfigurable computers. A three-step approach is described. The first two steps enumerate and prune the design space of basic GTM building blocks, which consist of FPGA buffers and GTM computation cores. The last step is to achieve a solution through an optimal combination of these building blocks where the cost function is the FPGA computation time and the constraints are FPGA co-processor board resources. Various FPGA buffers are presented so as to introduce design options of basic GTM building blocks. Algorithms used for the mapping are described. Experimental results are summarized to reveal the relationship between the GTM mapping results and FPGA board resource parameters.**

***Index Terms*—FPGA, Reconfigurable Computing, Image Analysis, High-Level Synthesis, Template Matching**

I. Introduction

Reconfigurable computers can offer significant performance advantages over conventional processors as they can be tailored to the particular computational needs of a given application. The technology has been demonstrated for the acceleration of various applications such as automatic target recognition (ATR) [1]-[3], image processing [4], machine vision [5], and morphology operation [6]. However, the programming of reconfigurable computers is extremely cumbersome, demanding that software developers also assume the role of hardware designers. Thus, one key to unlocking the full potential of these systems is developing truly automatic mapping tools. Motivated by such a need, this paper focuses on the mapping of generalized template matching (GTM) onto reconfigurable computers to help designers explore the design space and get a near optimal GTM design.

Manuscript received December 17, 2001. This research was supported by a DAGSI/AFRL grant and an Ohio State research challenge grant.

Xuejun Liang was with the Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435 USA. He is now with the Department of Computer Science, Jackson State University, Jackson, MS 39217 USA (xuejun.liang@ccaix.jsums.edu).

Jack Jean is with the Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435 USA (phone: 937-775-5106; fax: 937-775-5133; email: jjean@cs.wright.edu).

The reconfigurable computer addressed in the paper is a host computer with a co-processor board based on field programmable gate arrays (FPGAs). The target FPGA board may contain multiple FPGA chips, each with an array of homogeneous memory banks. Fig. 1 shows such a board structure where the dotted line and box are optional. The host may access an on-board memory either directly or through the FPGA chip. The host may also access the FPGA through a FIFO (or a Xbar).

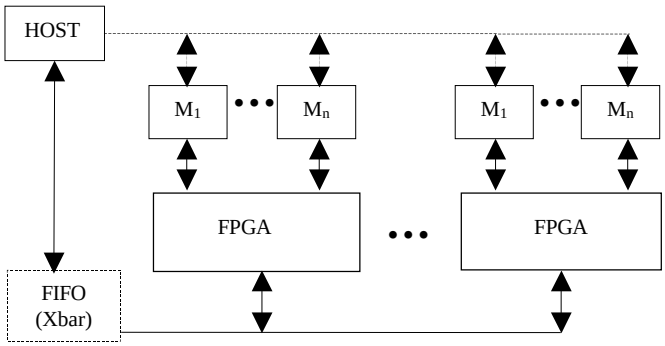


Fig. 1. Target Board Architecture

In Fig. 1, there are n memory banks for each FPGA chip. Although a memory bank may be double-ported so as to allow concurrent access of the host and the FPGA chip, each memory bank is considered as single-ported as far as the mapping process is concerned. So the words “memory bank” and “memory port” are used interchangeably in the paper. All memory banks have the same sizes in terms of storage capacity and port width. All FPGA chips on the board have the same structure and there is no direct connection between them. Copies of the same image frame may be stored in different memory banks to facilitate the evaluation of multiple templates. One image frame may be distributed among memory banks, sometimes with overlapping, to enable parallel evaluation of a single template. The host machine is responsible for the distribution of image frames to memory banks.

The generalized template matching (GTM) operations proposed in the paper include image processing algorithms for 2D digital filtering, morphologic operations, motion estimation, template matching and so on. They all involve moving a "window" (or template) pixel by pixel in a scanned line order. The GTM operations are similar to the "Sliding Window-Based Operations" (SWO) as in [7]. However, the GTM is more general in that all the pixels (or samples) in a SWO window are involved in the window computation while

in GTM the template in a window may be quite "sparse" and only a low percentage of pixels in a window is involved.

The overall approach of building the GTM design contains three steps. The first two steps enumerate, evaluate, and list enough number of basic GTM building blocks, called region functions. Each region function contains an FPGA buffer and a pipelined functional unit, called a unit function, which evaluates the window computation at one or more consecutive pixel locations. Different region functions have different throughputs, occupy different FPGA areas, and require different numbers of memory ports. The third step is to bind one or more region functions to each FPGA chip so that the total execution time is minimal under the FPGA board resource constraints such as the number of FPGA chips, the size of FPGA chips, the number of memory ports, and the width of memory ports. Region functions on all FPGA chips work independently and in parallel on different image regions and/or, if any, different templates under the control of a host program.

Related Research Works There have been many research projects on design environments for reconfigurable systems. They include COBRA-ABS high-level synthesis system [8], PECompiler [9], SPLASH Environment [10], RAW Machines [11], Napa-C compiler [12,13], Rapid (Rapid-C) [14, 15], GARP Compiler [16], DEFACTO [17], and Single Assignment C [18]. These system tools are usually aimed to support more general applications than the GTM operations and targeted to their particular architectures. Therefore it is difficult for them to explore the parallelism inherent in the GTM operations. In contrast, the various levels of parallelism of GTM operations can be explored systematically by the proposed mapping methodology. In addition, the GTM mapping is not geared to particular hardware components such as the dynamic control in the Rapid structure.

Brilliant FPGA designs have previously been proposed for some particular GTM applications, including automated target recognition (ATR) and 2-D convolution. These designs provide a clue for us to attack the GTM mapping problem systematically, although the design optimization with respect to the change of FPGA resource parameters is usually not considered in these individual designs.

Very different mapping strategies are used in [2] and [3] even though they are for the acceleration of the same ATR algorithm which requires correlating a huge number of predefined binary templates to the image area of interest. The researchers at UCLA use very compact adder trees that take advantage of template sparseness and FPGA lookup-table memory capability [2]. Their approach maps template information directly into the hardware and relies on fast reconfiguration to switch template information. They also take advantage of template overlap by computing the results of multiple correlations simultaneously. In contrast with this method, the researchers at BYU use statically configured hardware and memory-stored templates [3]. The technique computes the correlations column by column, and sums up the partial sums for all columns of template. In this method all column correlations are computed in parallel but only one column of data needs to be available for processing. This type of buffering is called partial buffering in [19].

The 2-D convolution is an essential image-processing function. The authors of [20] discuss several architectural solutions to a convolver design. The architecture for a complete 3×3 convolver includes shift registers for pixel values contained in delay lines and for the 3×3 convolution window. Because of these shift registers, the convolution can be carried out one pixel location each clock period. This type of buffering is called full buffering in [19]. Note that an alternative implementation of delay lines is to use the Configurable Logic Block (CLB) RAMs or Block RAMs inside Xilinx FPGAs.

One technique used in parallel compilers is closely related to the research. It is the software pipelining (or modulo scheduling) [21]-[23] that allows overlapping execution of consecutive loop iterations, with one fixed schedule for the loop body. In the paper, this technique is employed for the mapping of the GTM operations that can be characterized as nested loop computations.

Since for many applications the throughput of the reconfigurable coprocessor is limited by external memory accesses, it is very important to speed up the memory access by buffering frequently used data on-chip and scheduling as many external memory accesses in parallel as possible. The problem of buffering image data has been well studied [3], [20], [24], and [25]. In [24], [25] efforts were devoted to identifying data buffers for a nested loop from a compiler's perspective. Because their problem domain of nested loops is more general and therefore more difficult to handle than GTM, no effort was attempted in [24], [25] to optimize the buffer design under constraints of available resources. One way to schedule as many external memory accesses as possible in parallel is to distribute arrays over several memory banks. The paper in [24] formulates the array allocation problem as an Integer Linear Programming problem. In the formulation, one array is restricted to be allocated on one memory bank. This may eliminate the chance in which the parallelism could be achieved by using multiple memory banks for one array such as an image. The Napa-C compiler [13] also demands the same restriction. In our approach, one image array can be overlapped and distributed over several memory banks.

In [6] a two-level compilation scheme generates high-speed binary morphology pipelines that can handle a sequence of morphology templates described in a script file. The binary templates are all of size 3×3 and individual operations can be implemented with the same hardware circuit, called a supercell, that contains a full buffer and a 256-bit look-up table (8-bit input and 1-bit output). The first-level compiler, used only once given an FPGA board, generates a set of supercells with fixed connections that will fit in the FPGA chips. The second-level compiler can customize the look-up tables depending on the script file contents. The approach has

many advantages. One of them is very high compilation speed because application users only need to use the second-level compiler that takes seconds each time. The approach may not be feasible for more complicated templates. For example, it is not practical to use a single look-up table for a 3×3 convolution window on 8-bit pixels.

The paper is organized as follows. Section II describes the GTM mapping problem. Section III presents the mapping methodology. Section IV gives experimental results. Section V concludes the paper.

II. GTM Mapping Problem

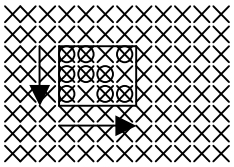
In the following, the GTM terminology and some basic assumptions are given in Section *A*. The GTM mapping problem and various design options are described in Section *B*.

A. GTM Terminology

The following example in C language syntax is used throughout the paper.

Example 1:

```
for (i=0; i<450; i++)
for (j=0; j<360; j++){
    y[i,j]=x[i,j]+x[i,j+1]+x[i,j+3]+
           x[i+1,j]+x[i+1,j+1]+x[i+1,j+2]+
           x[i+2,j]+x[i+2,j+2]+x[i+2,j+3]) }
```



The active points in a template include all the points necessary for the template computation. A template is usually specified with, for each active point, the active point location, which denotes the offset of the active point in the template, and the active point value**,** which denotes the "weight" associated with the active point. In Example 1 the number of active points is nine, and the active point locations are (0,0), (0,1), (0,3), (1,0), (1,1), (1,2), (2,0), (2,2) and (2,3), and the active point values are all ones.

The loop body that is iterated pixel by pixel through an image is called a window function. A window function is evaluated by applying one template at one pixel location at one time. In Example 1 the window function is the summation of nine image pixels at active points. A GTM operation is the application of a window function to an image frame. The image frame may be partitioned into several image regions IR_i (i=1,2,..,n), each being a set of consecutive image rows. For the evaluation of a window function in each IR_i, there is a set of templates $\{T_{i,1}, T_{i,2}, \ldots, T_{i,m(i)}\}$. Different image regions may associate with different sets of templates. A GTM operation thus can be formulated as the following nested loop computation.

```
For all i ←1 to n                    //all image regions
    For all j←1 to m(i)              //all templates
        For all pixel P in IR_i      //all pixels
            Window-Function (P, T_i,j)
```

From this formulation, the GTM operations may possess two levels of parallelism. First, for the evaluation of a window function different templates can be applied in parallel. This is called template-level parallelism. Second, the evaluation of a window function can be carried out in parallel at several pixel locations. This is called pixel-level parallelism. Some further assumptions about the GTM operations and FPGA designs are given below in order to narrow down the scope of the paper. Assumptions (1) and (2) are related to GTM operations. Assumptions (3) to (5) are related to the GTM FPGA designs.

(1) Only one image is processed in a GTM operation. The size of the image frame and the sizes of all image regions are available before mapping.

(2) No loop-carry dependency exists in a window function.

(3) The input image data are stored on the off-chip memory of FPGA boards. The output data are also stored on the off-chip memory. An upper bound of the number of output data is known beforehand.

(4) The default FPGA design style is generic. The generic design style treats the template data (weights and locations) as variables. Therefore, a generic design can be used in the window computation of multiple templates. If the other design style, a hard-coded one, is to be used instead, the template data need to be constants and available before the mapping. It is assumed that there is only one image region in any hard-coded design.

(5) An FPGA library of operator and storage components are available. The library also contains the area and the timing information.

Some notations used in the paper are listed in Table 1 for convenience. Some of them will be explained later. In Example 1, N_{AP}=9, N_{MW}=1, R_{WIN}=3, C_{WIN}=4, R_{IMG}=360, and C_{IMG}=450.

TABLE 1: GTM NOTATIONS

N_{BITS}	No. of Bits per Image Pixel
C_{IMG}	No. of Image Columns
C_{WIN}	No. of Columns of Template Window
N_{MW}	No. of Memory Writes in a Window Function
N_{AP}	No. of Active Points
R_{IMG}	No. of Image Rows
R_{WIN}	No. of Rows of Template Window
N_{MP}	No. of Memory Ports to an FPGA Chip
N_P	No. of Memory Ports Used ($N_P \le N_{MP}$)
$N_R(P)$	No. of Reads from Memory Port P
N_R	No. of Reads in Window Evaluation
N_W	No. of Writes in Window Evaluation
$N_W(P)$	No. of Writes to Memory Port P
N_{LB}	No. of Line Buffers
$N_{LB}(P)$	No. of Line Buffers for Memory Port P
PF	Packing Factor

B. GTM Design Options

Design options in the GTM mapping process can be grouped into those for the window movement (data buffering and packing in Section *1*), the window function evaluation (unit function in Section *2*), and the integration (region function in Section *3*). The GTM mapping complexity is discussed in Section *4*.

1) Data Buffering and Packing Strategies for GTM Operations

The data buffering and packing can be used to support the pixel-level parallelism of GTM operations and to reduce redundant memory accesses. The full and partial buffering and the packing structures were described in [19]. The basic idea of buffering and packing is reviewed and generalized here.

When no image datum is buffered inside the FPGA chip, the window computation at each pixel location needs to read N_{AP} pixel values from the memory. Since N_{AP} is usually much larger than one, it is desirable to buffer image data inside the FPGA chip. The window function evaluation at a new pixel location with the full buffering needs only one image pixel from the external memory, whereas with the partial buffering it needs R_{WIN} image pixels. When the number of image pixels provided from the external memory is less than R_{WIN} and greater than one, a hybrid buffering scheme is needed. There is a trade-off in minimizing the memory access and the buffer size.

For a 3×4 template window, the full buffering requires two image line buffers, while the partial buffering does not require any line buffer. Fig. 2 shows a hybrid buffer with one line buffer for a 3×4 template window where $c=C_{IMG}$. Pixels 0, 1, 2, and 3 are in the first row of the template window, pixels c+0, c+1, c+2, and c+3 in the second row, and pixels 2c+0, 2c+1, 2c+2, and 2c+3 in the third. Pixels from 4 to c-1 are in a line buffer. When the window moves to the next pixel location, two pixels c+4 and 2c+4 need to be read from the external memory. A control circuit is needed to enable pixel c+0, not pixel 2c+0, to enter the line buffer.

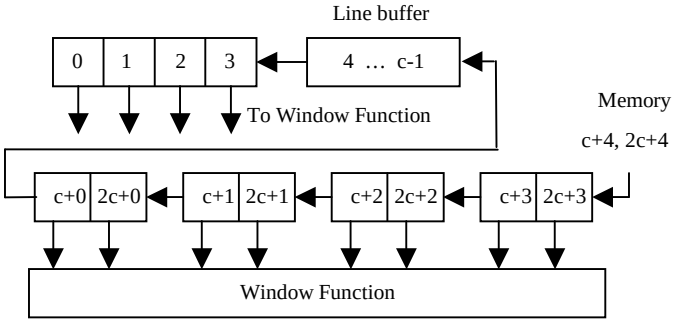


Fig. 2. Hybrid Buffering for a 3×4 Template

When several image pixels are stored in one memory location, each memory read can provide multiple image pixels. Therefore, it is possible to use multiple copies of the window function hardware to compute at several consecutive pixel locations in parallel. In order to support this parallelism, a special buffer called internal buffer is needed to distribute image pixels to the corresponding hardware [19]. The number of copies of window function hardware used is called packing factor (PF).

The buffering can be used together with the packing. Fig. 3 shows a hybrid buffering with packing where packing factor is 2. Pixels 0, 1, 2, 3, c+0, c+1, c+2, c+3, 2c+0, 2c+1, 2c+2, and 2c+3 are in the even window. Pixels 1, 2, 3, 4, c+1, c+2, c+3, c+4, 2c+1, 2c+2, 2c+3, and 2c+4 are in the odd window. All the pixels with even addresses are in the top registers or the top line buffer, whereas all the pixels with odd addresses are in the bottom registers or the bottom line buffer. When the window computation proceeds to the next two pixel locations, two memory reads are needed. The first brings in two pixels c+6 and c+7; the second brings in two pixels 2c+6 and 2c+7. A control circuit is needed to enable pixels c+0 and c+1, not pixels 2c+0 and 2c+1, to enter the line buffers.

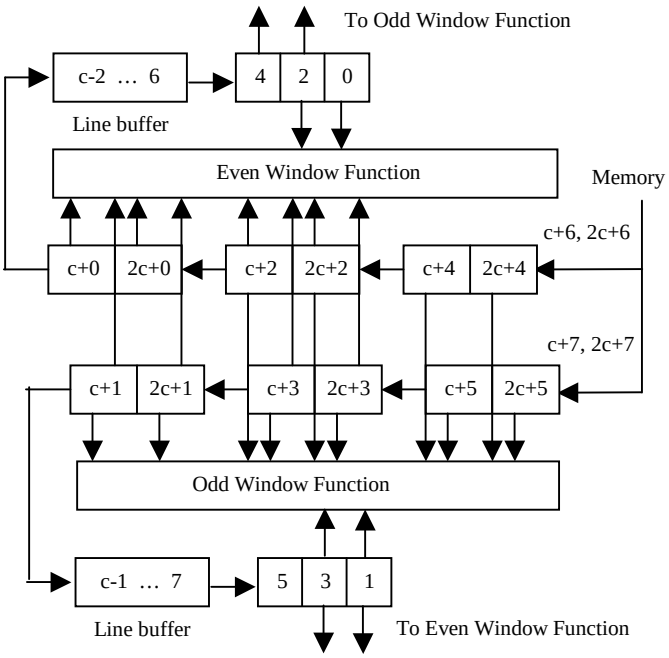


Fig. 3. Hybrid Buffering with Packing (packing factor=2) for a 3×4 Window

In general, an FPGA buffer can be structured to use multiple memory ports, and different memory ports may support different numbers of reads during the window function evaluation. A general FPGA buffer with N_P ($1 \le N_P \le N_{MP}$) memory ports consists of N_P FPGA buffers, each corresponding to one memory port. These buffers can be full, partial, hybrid, or internal buffers. The template window is partitioned into N_P regions, each with several rows of the template window and corresponding to one memory port. An example of such a general FPGA buffer structure is shown in Fig. 4. In this example, $N_R=3$ and $R_{WIN}=5$. The buffer structure uses two memory ports ($N_P=2$), and therefore the template window is partitioned into two parts. The top part is for two rows of the template window. The corresponding memory port supplies one image pixel during each window computation ($N_R(1)=1$), and thus is connected to a full buffer. The bottom part is for three rows. The port supplies two pixels ($N_R(2)=2$), and thus is connected to a hybrid buffer. Both memory ports need to buffer one image line ($N_{LB}(1)=N_{LB}(2)=1$) aside from data inside the template window. Also, if the number of active points is less than the number of points in the template window, a selector that outputs only active point pixels is necessary when the generic design style is used. Note that when the memory packing is involved, there could

be more than one selector [26]. In addition the shift registers in Fig. 4 must have parallel outputs that are connected either directly to the window computation part or through the selector.

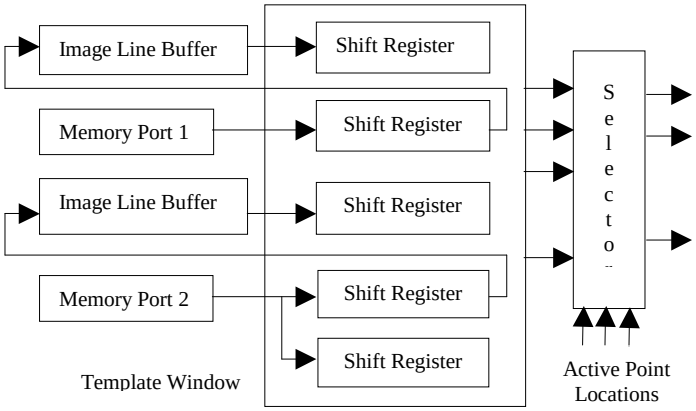


Fig. 4: A General FPGA Buffer Example

In the GTM mapping, such FPGA buffers are provided when $1 \le N_R \le R_{WIN}$. The internal buffers [19] to distribute pixel data to different copies of window computation are provided when $N_R = N_{AP}$ and PF > 1. No buffering strategy is considered for $R_{WIN} < N_R < N_{AP}$ because of the complicated memory controller design and relatively few performance benefits. Note it is assumed that $R_{WIN} < N_{AP}$, i.e., the number of active points is greater than the number of rows of the template window.

2) Unit Function

As discussed previously, when the packing strategy is used, it is possible to evaluate in parallel a window function at consecutive pixel locations along the scanned line. A unit function is an FPGA pipelined hardware functional unit that computes a window function at PF consecutive pixel locations by using PF copies of a window function design. The unit function may share the hardware across these copies. As a result, there are many unit function design options that trade off hardware space and speed. There is no predefined structure of a unit function; a unit function design depends on the scheduling of window function operations and the assignment of operations to hardware components. The loop pipelining technique of modulo scheduling is used in the unit function design in this paper. Recall that the window function is a loop body. In modulo scheduling, iterations of a loop body are initiated at a constant time apart. This constant time (in clock cycle) is called the initiation interval (II) [21]. Therefore, the throughput of a unit function is proportional to PF/II when the number of iterations is large. In the GTM mapping, all possible PFs and IIs are considered.

3) Region Function

A region function consists of a unit function and an FPGA buffer as shown in Fig. 5. The II of the unit function should be equal to the data introduction interval, which is the clock cycles needed for external memory access (by the buffer and the unit function) in each window computation. Given a data introduction interval, it can be proved that there exists a modulo schedule of the unit function with II equal to it [27]. (Assumption (2) in Section 2.1 was used in the proof.) The region function iterates the execution of its unit function through a set of templates (when the design style is generic) and pixels in a part of image regions. So a region function is obtained by combining one unit function and one FPGA buffer, and then being assigned a set of templates and a part of image regions. Also a region function has to be assigned to a specific FPGA chip and the memory ports connected to the chip.

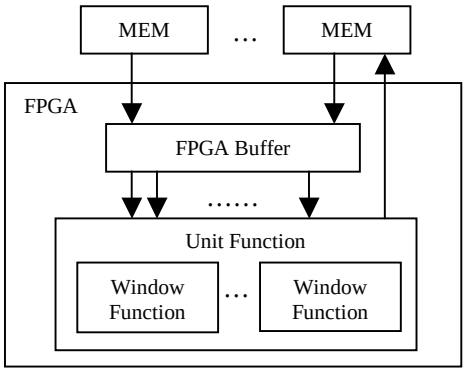


Fig. 5: Region Function

One or more region functions should be selected for each FPGA chip such that the total execution time is minimal under the FPGA board resource constraints such as the number of FPGA chips, the size of FPGA chips, the number of memory ports, and the width of memory ports. Region functions on all FPGA chips work independently and in parallel on different parts of image regions and/or, if any, different templates under the control of a host program. Note that different region functions on the same FPGA chip do not share the memory banks because each region function may need to access the memory in every clock cycle during the computation.

4) GTM Mapping Complexity

A naïve approach is to enumerate FPGA buffers and unit function designs and to compare all the combinations so as to get the final region functions. The complexity of this approach is very high as shown below.

The number of different FPGA buffers (including non buffering and internal buffer) for a GTM operation is lower bounded by $\eta \times N_{PF}$ where η is the number of ways to distribute memory reads for a window function among memory ports and N_{PF} is the number of all possible packing factors. Furthermore $\eta = \eta_1 + \eta_2$ where η_1 is the number of ways of read distribution when $1 \le N_R \le R_{WIN}$ and η_2 is that when $N_R = N_{AP}$. More specifically, η_1 is the number of possible pairs of vectors $(N_R(1), \ldots, N_R(N_p)$ and $(N_{LB}(1), \ldots, N_{LB}(N_p))$ where $1 \le N_p \le N_{MP}$ and $1 \le N_R \le R_{WIN}$, that satisfy the following conditions:

$$N_R = \sum_{p=1}^{p=N_P} N_R(p)$$
$$N_R(1) \geq N_R(2) \geq ... \geq N_R(N_P) \geq 1$$
$$R_{WIN} - N_R = \sum_{p=1}^{p=N_P} N_{LB}(p)$$
$$N_{LB}(p) \geq 0,\ 1 \leq p \leq N_P$$

And η_2 is the number of vectors ($N_R(1)$, …, $N_R(N_p)$) where $1 \leq N_p \leq N_{MP}$ and $N_R = N_{AP}$, that satisfy the following conditions:

$$N_R = \sum_{p=1}^{p=N_P} N_R(p)$$
$$N_R(1) \geq N_R(2) \geq ... \geq N_R(N_P) \geq 1$$

There are no general formula to express η_1 and η_2. But the growth of η_2 alone is $\theta(N_{AP}{}^{(N_{MP}-1)})$.

In the unit function design, all possible PFs and IIs are considered. For each pair of (PF, II), the corresponding unit function design needs to go through a pipeline synthesis process, which includes scheduling of PF copies of a dataflow graph, resource sharing and binding, and datapath and controller generation. For scheduling a general dataflow graph (not a tree), the minimum-latency resource-constrained scheduling problem and the minimum-resource latency-constrained scheduling problem are known to be intractable [31]. The scheduling in the GTM mapping has to be performed as many times as the number of all possible pairs of (PF, II).

As to the building of region functions, given *m* FPGA buffers and *n* unit function designs, there are *mn* region functions and the number of sets of region functions to be considered is 2^{mn}. Note that the image region partitioning, the processing region binding, and the template bindings have not been considered yet.

Therefore, the complexity of this naïve approach is very high, and a much more efficient approach needs to be developed. Note that, although the mapping problem is formulated as a constrained optimization problem, there is no attempt in getting optimal solutions in this paper due to the problem complexity. Instead this paper proposes efficient algorithms on getting "good" solutions, hopefully "near-optimal" because of the solution optimality to some sub-problems involved.

III. Mapping Methodology

A. Overall Approach

The methodology of building an near optimal GTM design is to first enumerate, evaluate, and list enough number of region functions, i.e., pairs of unit function and FPGA buffer, and then to select a subset of candidate region functions, bind them to FPGA chips and memory ports, and partition the total workload among the region functions. It has three steps.

Step 1: Enumerate all non-dominated memory access patterns (MAPs). The concept of MAP, as defined later, is a key to the mapping process because both unit functions and buffer structures can be determined from a MAP.

Step 2: A set of region functions, one for each non-dominated MAP, can therefore be obtained. These region functions can be ranked based on their throughputs, i.e., PF/II.

Step 3: The region function selection and binding and workload partition are performed so as to minimize the total execution time.

The first step is an enumeration process that requires an efficient algorithm. The second step is a synthesis and generation process. For each non-dominated MAP, a corresponding unit function design can be obtained via a synthesis process and a corresponding FPGA buffer can be generated. Then areas of the unit function and the buffer can be estimated. The last step is a combinatorial optimization process. In the following, each step is described in detail after the input representations are introduced.

B. Input Representations

The inputs of the GTM mapping process include the FPGA board specification, the VHDL FPGA component library of the operators, and the GTM operation specification.

The target FPGA board can be specified with the number of FPGA chips (N_{FPGA}), the number of memory ports connected to an FPGA chip (N_{MP}), the width of memory port (W_{PORT}), and the CLB (or SLICE) count of FPGA (S_{FPGA}).

The VHDL library contains the information about the operator and storage component VHDL designs, which should include the area and the timing. The implementation of an operator in the library can be either pipelined or non-pipelined. A non-pipelined operator is considered as a pipelined design with only one stage. The timing of an operator can be specified by two parameters, clock cycle and clock period. The area of an operator can be measured by FPGA CLB counts (or SLICE counts). Because the library component information may vary for different FPGA families, the library is required to specify the FPGA family. The FPGA buffers are not included in the library, because they can be generated by an FPGA buffer generator.

The GTM operation specification includes two parts. One is the image regions and templates that include the number of image regions, the number of templates associated with each image region, and the size of each image region. The other is the dataflow graph (DFG) of the window function. The DFG nodes include input nodes, output nodes and operation nodes. The input nodes include one type of particular nodes called pixel nodes, which represent the input image pixels at the active points. Each operation node needs to specify the resource type that is used to implement the operation. For example, the window function of Example 1 in Section II can be represented with a DFG as shown in Fig. 6(a). In this example, the window function simply sums up nine image pixel values at active points. The nine circle nodes at the top represent the nine image data at the active points, which may

come from an FPGA buffer or directly from memory ports. The cylinder node at the bottom represents a memory write operation. Fig. 4(b) shows another DFG, in which the hexagon nodes denote the inputs of template weights at the active points. A special text format has been developed to describe such a DFG.

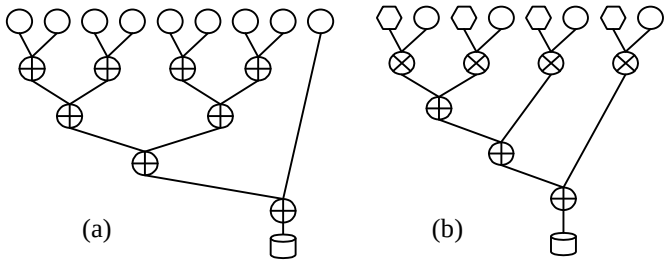


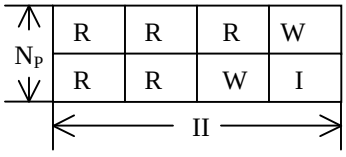
Fig. 6: Two Dataflow Graphs

C. Step 1: Memory Access Pattern Enumeration

For a given pair of PF (packing factor) and II (initiation interval), a memory access pattern (MAP) includes the following information.

1. The number of memory ports used (N_P), which should be less than or equal to N_{MP}.
2. The number of memory reads ($N_R(p)$) from each port, p, ($1 \le p \le N_P$) in the II clock cycles. The vector ($N_R(1)$, $N_R(2)$,…, $N_R(N_P)$) is called the memory read pattern.
3. The number of memory writes ($N_W(p)$) to each port, p, ($1 \le p \le N_P$) in II clock cycles. The vector ($N_W(1)$, $N_W(2)$,…, $N_W(N_P)$) is called the memory write pattern.

Intuitively, a MAP for a given pair of PF and II can be represented as a rectangle with $N_P \times$II cells. Each cell is labeled with R for reading, W for writing, or I for idling. A row of cells stands for memory accesses of one memory port, and a column of cells stands for the memory accesses in one particular clock cycle. For example, when PF=2, II=4, and N_{MP}=4, there exist many MAPs. One of them is shown below, which uses two memory ports (N_P=2). For this MAP, the memory read pattern is (3,2) and the memory write pattern is (1,1).



The GTM operation involves intensive memory accesses when image is stored off-chip. At each pixel location, the window function consumes N_{AP} image pixel values. The GTM FPGA designs that use different MAPs tend to have significant differences in performance and hardware resource requirements. Both the FPGA buffer and the unit function design can be derived based on a MAP (see the next section). Therefore, the MAP enumeration is a systematic way to enumerate region functions. However, not all MAPs can lead to a useful region function.

A MAP can be evaluated with the following four quality measures. The first is the number of image line buffers (N_L) required by a MAP. When the total number of memory reads N_R by the MAP is less than the number of rows of the template window (R_{WIN}), $N_L = R_{WIN} - N_R$. Larger N_L requires more FPGA area for the buffer. The second is the number of memory port (N_P). Larger N_P means more memory banks and more FPGA input/output pins. The third is the initiation interval (II). Larger II corresponds to a lower throughput of the unit function. The last measure is the memory size requirement (S_M). Lager S_M requires more memory space.

Definition: For two MAPs A and B with the same packing factor, A dominates B if A's $N_L \le$ B's N_L, A's $N_P \le$ B's N_P, A's $S_M \le$ B's S_M, and A's II ≤ B's II.

Therefore, if MAP A dominates MAP B, then the region function corresponding to A is most likely better than the region function corresponding to B. Hence, a dominated MAP can be safely discarded. A MAP not dominated by any other MAPs is called a *non-dominated MAP*.

Note that non-dominated MAPs are only a very small fraction of all MAPs. Algorithms to prune the MAP design space (remove dominated MAPs without enumeration) and to obtain all non-dominated MAPs were developed by the authors of this paper. Their effectiveness was detailed in [28] where it was shown that, when the pruning was applied to a particular case (four memory ports and each memory location storing four pixels), 72 MAPs which were less than 2% of the MAP space survive for a small problem (a 3×3 template with 9 active points) and 346 MAPs which were less than 0.2% remain for a bigger problem (a 15×15 template with 30 active points).

D. Step 2: Region Function Generation

A region function consists of two parts, a unit function and an FPGA buffer. Section *1* describes the unit function mapping process and the unit function area estimate. Section *2* presents the FPGA buffer generation and its area estimation

.

1) Unit Function Mapping and Area Estimate

The inputs for the unit function mapping include FPGA board information, FPGA library components, a DFG of window function, and a MAP including PF and II. A scheduled DFG and a resource table that summarizes the usage of hardware resources are produced after scheduling and binding. They can be converted into a datapath and a control table based on which the unit function area is computed.

The problem of the unit function mapping is closely related to the well-studied high-level synthesis (HLS) [29-32]. Automated approaches to the fundamental HLS problem consist of two related constrained optimization problems: temporal scheduling and spatial binding. HLS techniques have been widely applied to the compiler (or synthesis tool) designs on reconfigurable systems. The COBRA-ABS high-level synthesis system [8] performs globally optimizing high-level synthesis using simulated annealing, integrating all partitioning, scheduling, binding, and allocation operations in one optimization step.

The objective of the unit function mapping in this paper is to minimize the unit function area under constraints of an initiation interval (II), a packing factor (PF), and latency. The approach in the unit function mapping, unlike that in [8], is to perform the scheduling and the resource binding separately. The unit function mapping consists of five sequential tasks:

- Task 1: Schedule one DFG to determine the latency that is used in Task 2.
- Task 2: Schedule PF copies of the DFGs together using a list-scheduling algorithm.
- Task 3: Perform the operator sharing and binding.
- Task 4: Generate the datapath and the controller.
- Task 5: Estimate the unit function area.

Each task is described as follows.

Task 2: List-Scheduling with PF DFGs: The scheduling problem in the unit function mapping is to obtain a modulo schedule of a functional pipeline that allows operator resources with pipelined implementations. The schedule minimizes the area requirement subject to latency and II constraints. In order to reduce the complexity of the scheduling problem, a list-scheduling algorithm (heuristic) is used. It is based on a similar algorithm for the functional pipeline scheduling in [31] where each operator resource is assumed to be non-pipelined. The computation of resource lower bounds is modified accordingly.

The initial upper bound of resource instances for each resource type in the list-scheduling algorithm [31] is one. A new heuristic on a suitable initial upper bound is used in the unit function mapping. The initial upper bound are defined by the new heuristic as follows. Based on the initial scheduling information from Task 1, for each required resource type, compute the first control step T_S that an operation of this type is scheduled, and then count the number N_{OP} of operations of this type that are scheduled at the time interval from T_S+1 to T_S+II. The initial upper bound of this type of resource instances is defined as $\lceil N_{OP} / II \rceil$. Setting the initial upper bounds to be ones may not lead to a scheduling result as good as the new heuristic. A comparison of the two methods can be found in [26].

Task 3: Operator Sharing and Binding: Once the DFG (actually PF copies of the original DFG) is scheduled and the start time of each node v_i is denoted by t_i, i=1, 2, …, n, two operations v_i and v_j with the same resource type executing at

$$t_i = k_i + p_i \cdot II, \text{ where } 1 \le k_i \le II \text{ for some } p_i, \text{ and}$$
$$t_j = k_j + p_j \cdot II, \text{ where } 1 \le k_j \le II \text{ for some } p_j.$$

can share the resource instance if and only if $k_i \neq k_j$. This condition is easy to check. In the current implementation, the hardware sharing is always selected whenever the above condition holds. That means, the area cost of multiplexers is assumed to be less than that of the hardware being shared (which may not be true for simple operators). It is also assumed that one operator can have only one resource type that implements it. Task 3 produces a resource table according to the scheduled DFG.

Task 4: Datapath and Controller Generation: The datapath generation involves the insertion of registers and multiplexers and the interconnections of components. The controller generation is to produce a control table that controls registers and multiplexers to steer data flows in the datapath. The inputs to the datapath and controller generation are the scheduled DFG and the corresponding resource table. The tasks involved include:

1. Building ports: When no FPGA buffer is used, some pixel nodes need to be merged to share memory ports according to the MAP.
2. Adding delay registers: The data coming from the FPGA buffer or the memory ports are assumed to be valid for only one clock cycle. If an image pixel value is available at the time step *n* and is consumed at the time step *m*, then *m*-*n* registers are needed to delay the datum. Note that when the FPGA buffer strategy is used, *n* is 1; otherwise *n* is the data arrival time. Also, because the output of an operator is valid for II clock cycles, if the output is not consumed after II clock cycles, delaying registers are needed.
3. Adding node registers: They hold the computation results of nodes.
4. Adding multiplexers when multiple nodes share the same resource instance. Note that when the pixel nodes are grouped, the inputs to some operators are also grouped accordingly. Thus the number of inputs to the corresponding multiplexer can be reduced.
5. Generating the datapath in a net list format and the control table according to the previous results.

Task 5: Unit Function Area Computation

The unit function area can be computed based on the datapath and the control table as follows. Note that the FPGA routing area is not considered.

1. For the datapath area, the task is straightforward because every component in the datapath is from the FPGA component library and has its area specified. So simply summing up these component areas is enough. Note that when the library component is implemented in VHDL with generic parameters, formula of calculating the areas according to the parameters are assumed to be available. Because the CLBs in practice may be shared by different library components, the datapath area is overestimated in our approach.
2. For the controller area estimate, the task is more complicated. The controller is to control the datapath to compute through a pipelined loop. It goes through three phases, which are prologue, steady state and epilogue. Assume that (1) the lengths (clock cycles) of prologue, steady state and epilogue are n_1, n_2, and n_3 respectively, (2) the number of iterations through the steady state phase is n, and (3) one-hot encoding is used in the controller state encoding. Then at least $n_1+n_2+n_3+\lceil \log_2(n) \rceil$ flip-flops or $((n_1+n_2+n_3+\lceil \log_2(n) \rceil)/2$ CLBs) are needed for the state encoding and transitions, where $n_1+n_2+n_3$ flip-flops are

used for the state encoding and $\log_2(n)$ flip-flops are used for a counter. Control signals of the controller include register enable signals and multiplexer selection signals. Their area estimates can be found in [26].

2) FPGA Buffer Generation and Buffer Area Estimate

Given a MAP, N_{AP}, and sizes of image regions and template window, the FPGA buffer can be generated as follows. In order to reduce the initial buffer filling time, the number of line buffers for memory port p, $N_{LB}(p)$ ($1 \le p \le N_P$), can be computed by the minmax decomposition of R_{WIN}-N_R with respect to N_P. That is, $N_{LB}(1)+\ldots+N_{LB}(N_P)= R_{WIN}-N_R$ and the maximum values of $N_{LB}(1),\ldots,N_{LB}(N_P)$ is minimized. For example, {3, 2, 2} is the minmax decomposition of 7 with respect to 3. Then, when $N_R = N_{AP}$ and PF>1, each of the N_P memory ports corresponds to an internal buffer; when $1 \le N_R \le R_{WIN}$, the buffer type for each memory port can be determined as follows.

Buffer Type	Condition
Partial	$N_{LB}(p)=0$
Full	$N_R(p)=1$, $N_{LB}(p)\neq 0$
Hybrid	$N_R(p)>1$, $N_{LB}(p)\neq 0$

After the buffer structure is determined, the template window can be partitioned accordingly. Each partition (several consecutive rows of the original template window) corresponds to one of the four types of basic FPGA buffers—Internal Buffer, Full Buffer, Partial Buffer, and Hybrid Buffer. An entire FPGA buffer can then be generated by combining those buffers and selectors, if needed. An FPGA buffer for each memory port can be built in a component hierarchy. For example, a full buffer consists of FIFOs (for line buffers) and shift registers (with parallel outputs for pixels at the template window), and a FIFO is composed of a dual port RAM and an address controller, and so on. The sizes of buffer components can also be computed according to the buffer parameters.

A bottom-up approach is used in estimating the FPGA buffer area. The area of each base component of the buffer is computed first, and then the area of a higher layer component is obtained by adding up areas of all its components. The following gives some examples of buffer component area computation.

Area of Dual Port RAM: Assume that the depth and the width of the dual port RAM are *n* and *w*, respectively. When the LUT in a CLB is used for the RAM implementation, the number of CLBs is $w \times \lceil n/16 \rceil$. When the dedicated BlockRAM is used, assume that W_B is the smallest value of BlockRAM data width that is not less than *w*, then the number of BlockRAMs is $\lceil n/(N_{BRAM}/W_B) \rceil$, where N_{BRAM} is the number of bits per BlockRAM.

Area of Shift Register (with parallel outputs): Assume that the depth and the width of the shift register are *n* and *w* respectively. The number of CLBs for this shift register is $\lceil n \times w/2 \rceil$.

Area of FIFO: The FIFO consists of two parts—Dual Port RAM and Address Control. Assume that the depth and the width of a FIFO are *n* and *w*, respectively. The control part consists of two counters with enable and reset. The two counters use $2 \times \lceil \log_2(n) \rceil$ flip-flops, and thus $\lceil \log_2(n) \rceil$ CLBs. Then the total area is the sum of both components.

Area of Full Buffer with Packing: The full buffer with packing consists of FIFOs with equal size and shift registers with equal size. In order to compute the area, the numbers of FIFOs and shift registers, the lengths of the FIFO and the shift register are needed. The full buffer with packing has the following parameters: the number of columns on image region (C_{IMG}), the number of rows on template window (R_{WIN}), the number of columns on template window (C_{WIM}), the packing factor (PF), and the number of bits on image pixel (N_{BITS}).

The number of shift registers is $R_{WIN} \times PF$. The number of FIFOs is $(R_{WIN}-1) \times PF$. Let L_{REG} be the length of the shift register and L_{FIFO} be the length of the FIFO. Then

$$L_{REG} = \left\lceil \frac{C_{WIN} + PF - 1}{PF} \right\rceil \quad and \quad L_{FIFO} = C_{IMG} / PF - 1.$$

For example, when PF=1, C_{IMG}=80, and C_{WIN}=4, then L_{REG}=4 and L_{FIFO}=79. When PF=2, C_{IMG}=80, and C_{WIN}=4, then L_{REG}=3 and L_{FIFO}=39. It is assumed that C_{IMG} is divisible by PF. Then the full buffer area is the sum of areas of FIFOs and areas of shift registers. Note that when PF=1, the computed area is for the full buffer without packing. The area estimates for other types of buffers can be obtained similarly (see [26] for details).

E. Step 3: Region Function Selection and Binding

In the region function selection and binding process, a set of region functions is selected for each FPGA chip and each region function is assigned particular memory ports, templates, and processing regions (consecutive rows of an image region). The process therefore includes the FPGA chip binding, the memory port binding, the image region partitioning and the processing region binding, and the template binding. For an FPGA chip, i ($1 \le i \le N_{FPGA}$), the region functions $RF_{i,j}$, $1 \le j \le q(i)$, together form the chip design, which has to satisfy an FPGA area constraint and a memory port constraint. As a result, for the FPGA chip binding and the memory port binding, the combinatorial optimization problem can be formulated as follows.

$$\begin{cases} \text{To minimize} \\ \quad \max\{Time(RF_{i,j}) \mid 1 \le i \le N_{FPGA}, \text{and } 1 \le j \le q(i)\} \\ \text{Subject to} \\ \quad \begin{cases} \sum_{1 \le j \le p \backslash q(i)} Area(RF_{i,j}) \le AreaConstrant, \ 1 \le i \le N_{FPGA} \\ \sum_{1 \le j \le q(i)} Port(RF_{i,j}) \le N_{MP}, \ 1 \le i \le N_{FPGA} \end{cases} \end{cases}$$

In the above formulation, the objective function is the GTM computation time, N_{FPGA} is the number of FPGA chips on the target board, and N_{MP} is the number of memory ports connected to each FPGA chip. Time($RF_{i,j}$) stands for the $RF_{i,j}$ execution time, Area($RF_{i,j}$) for the $RF_{i,j}$ FPGA area, and

Port($RF_{i,j}$) for the $RF_{i,j}$ memory ports. The execution time of the GTM design is the maximum execution time of all $RF_{i,j}$ execution times because all $RF_{i,j}$ work independently and in parallel. Note that *Time*($RF_{i,j}$) cannot be obtained before the processing region binding and the template binding.

Assume there are r image regions, IR_s ($1 \le s \le r$), for a GTM operation and there are r_s templates, $T_{s,b}$ ($1 \le b \le r_s$), in each image region IR_s. Assume that the number of rows of each IR_s is d_s. A partition of IR_s is a set of disjoint subsets, $PR_{s,t}$ ($1 \le t \le p_s \le d_s$), of IR_s, each with $d_{s,t}$ consecutive rows of the image, that satisfy

$$\sum_{t=1}^{p_s} d_{s,t} = d_s.$$

To solve the above region function selection and binding problem, an efficient algorithm was developed with the assumption that region functions have the same clock rate. Due to the space limitation, only the algorithm outline is given here (refer to [26] for details).

1. The problem can be simplified to the case of a single FPGA chip by using the same set of region functions for different FPGAs.
2. The problem can be further simplified by decoupling the memory port binding from the image region partitioning and processing region binding. When the memory port binding problem is solved, the image region partitioning and processing region binding is straightforward.
3. The memory port binding problem can be decoupled into two problems, a generalized knapsack problem and a generalized integer partition problem. Either problem can be solved efficiently.

IV. Experimental Results and Analysis

In this section, several experiments are performed to learn the mapping results under different constraints, and to compare the mapping results of different restricted GTM cases.

A. Mapping Results

In this experiment, Example 1 in Section II is used to illustrate the mapping results. It is assumed that N_{FPGA}=1, N_{MP}=1, W_{PORT} =16, B_{DATA}=8, N_{MW}=1, R_{WIN}=3, C_{WIN}=4, C_{IMG}=320, R_{IMG}=200, and N_{AP}=9. In this simple example, the FPGA board has only one FPGA chip that is connected to only one memory port. So the FPGA chip can contain only one region function and thus the whole GTM design consists of only one region function. There are totally eight non-dominated MAPs and thus eight candidate region functions as shown in Table 4. If the FPGA chip has 500 CLBs, then the fourth region function is the fastest solution that needs 160,000 clock cycles to compute.

For the WildForce FPGA board [33], which can be connected with a host computer such as a PC via the PCI bus of the host computer, there are five FPGA chips, each with one memory port connected, each memory word is 32 bits wide, and each FPGA has 3,136 CLBs. In this case, N_{FPGA}=5, N_{MP}=1, W_{PORT} =32, S_{FPGA}=3136, and the GTM mapping tool produces a region function (PF=4 and II=5) with an area of 1236 that works on the five FPGA chips in parallel. The computation time is 16,000 clock cycles. It can be observed that each FPGA chip still has extra area but the region function is the fastest design (the tool can produce) already. For this small example, if there were more memory ports, then a bigger region function with more copies of the window function and/or multiple region functions could be accommodated and further speedup could be obtained. Assume that each FPGA chip has 4 memory ports instead, i.e. N_{MP}=4. Then the GTM mapping tool produces 43 candidate region function designs and binds two region functions to each FPGA chip, one with an area of 1,236 (PF=4, II=5 and N_P=1), and the other with an area of 982 (PF=4, II=2 and N_P=3). The computation time is 4,800 clock cycles. The speedup is about 3.3 compared with the single memory port case. If the FPGA CLB count were larger than 4,994 (=1,236×4), then the GTM mapping tool would have selected four copies of the fastest region function design in each FPGA and obtained a speedup of four.

TABLE 2: ALL CANDIDATE REGION FUNCTION DESIGNS

PF	II	Non-Dominated MAP	Buffer Type	Area (CLB Counts)			Cycle
				Buff	UF	RF	
2	3	RWW	Full/Packing	568	244	812	96000
2	4	RRWW	Hybrid/Packing	394	232	626	128000
1	2	RW	Full	458	100	558	128000
2	5	RRRWW	Partial/Packing	216	240	456	160000
1	3	RRW	Hybrid	291	112	403	192000
1	4	RRRW	Partial	120	116	236	256000
2	11	RRRRRR RRRWW	Internal	32	220	252	352000
1	10	RRRRRR RRRW	No Buffer	0	110	110	640000

B. Mapping Results Under Different Constraints

Several experiments are performed to show how the mapping results change when the FPGA board parameters change. They include CLB counts, the number of the memory ports, and the width of the memory data port.

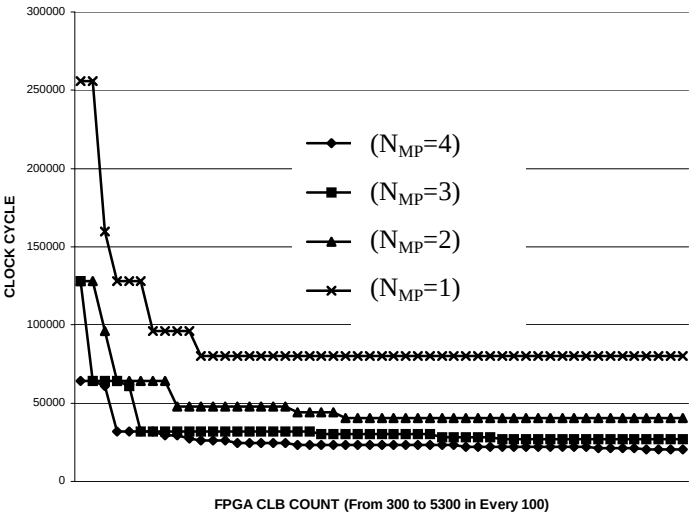


Fig. 7: Mapping Results of Different Memory Port Numbers (WPORT =32)

Again Example 1 in Section II is used. It is assumed that N_{FPGA}=1, W_{PORT} =32, B_{DATA}=8, N_{MW}=1, R_{WIN}=3, C_{WIN}=4, C_{IMG}=320, R_{IMG}=200, and N_{AP}=9. Fig. 7 shows the number of clock cycles of the GTM mapping results. The FPGA CLB count is in the range from 300 to 5,300. As the CLB count increases, initially the computation time of the mapping results for the one memory port case is reduced rapidly. But when the FPGA CLB count is over 1,300, the computation time stays constant. That means increasing the FPGA size further does not help any more. Such an FPGA CLB count is called the critical CLB number. For the two-memory port case, the critical CLB number is 2,100. For the three-memory port case, the critical CLB number is 3,600. For the four-memory port case, the critical CLB number is 5,000. It can be seen from Fig. 7 that the speedup of using two memory ports compared with using one port is roughly 2 when the FPGA is large enough. Also at some points, adding more memory ports leads to only marginal speedup. Similar results can be observed when data port is equal to 16 or 8 bits wide.

Fig. 8 compares the mapping results of different memory data port widths. When the FPGA CLB counts are large, the GTM mapping tool always produces a faster GTM design given a wider memory data port. But the speedup is less than the ratio of the increase in memory data port widths. This is because although a wider memory data port allows the packing of more window function copies in a unit function, the packing strategy increases the number of memory writes as well and thus increases the minimal II. On the other hand, when the FPGA CLB counts are small, increasing the width of memory data ports may not lead to speed increase. Similar results can be obtained when the number of ports is equal to 3, 2, or 1.

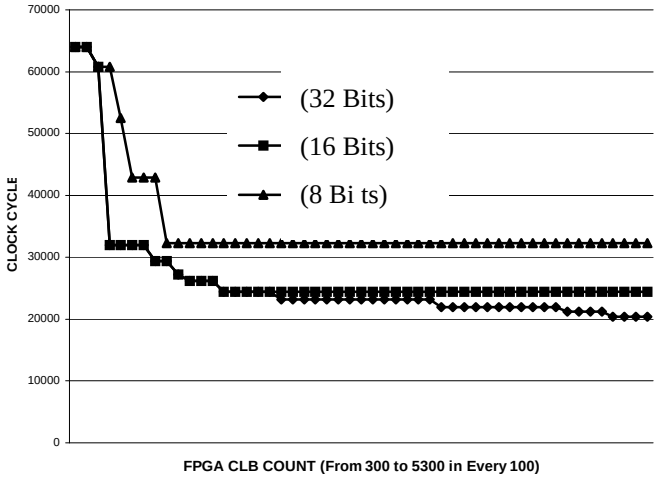


Fig. 8: Mapping Results of Different Memory Data Port Widths (N_{MP}=4)

C. Mapping Results Under Different Restricted Cases

Human designers often impose some restrictions in their FPGA designs to simplify the design tasks, hoping there is not much sacrificing in the design optimality. The experiments here try to provide some explanations to such human designers' decisions. Table 3 lists three types of design restrictions. In the experiments, Example 1 in Section II is used. It is assumed that N_{FPGA}=1, W_{PORT} =32, B_{DATA}=8, N_{MW}=1, R_{WIN}=3, C_{WIN}=4, C_{IMG}=320, R_{IMG}=200, and N_{AP}=9.

TABLE 3: RESTRICTIONS ON GTM MAPPING PROBLEM

	Design	Restriction
1	Region Function	With only one region function
2		With the same region functions
3	Unit Function	With only one memory port
4		With only one memory port or With multiple memory ports, but each exclusively for reading or writing
5	Buffer	Without hybrid buffer
6		Without packing
7		With only one memory port

Fig. 9 shows the number of clock cycles of the GTM mapping results under various restrictions on region function and unit function when the FPGA CLB count is from 800 to 5300.

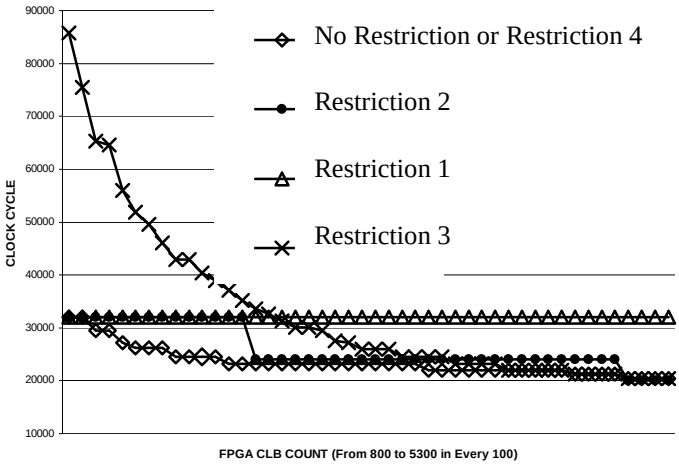


Fig. 9: Mapping Results Under Restrictions on Region Function or Unit Function

Note that the results corresponding to no restriction and Restriction 4 are identical. This means that, for this example, the mapping results is not degraded by limiting the unit function design to one memory port or to multiple memory ports where each is exclusively used in reads or writes. The mapping results with Restriction 2 have the same execution times (clock cycles) as those without restrictions when FPGA CLB counts are very small (less than 900) or very large (larger than 4,600), but are a little slower otherwise. The mapping results with Restriction 1 have the same execution times (clock cycles) as those without restrictions when FPGA CLB counts are very small (less than 900), but are slower otherwise. The mapping results with Restriction 3 have almost the same execution times (clock cycles) as those without restrictions when FPGA CLB counts are large (greater than 3,700), but are very slow when FPGA CLB counts are small (less than 1,700).

It can be concluded that when FPGA CLB counts are relatively large (larger than 3,700 for this example), Restrictions 2, 3, and 4 do not degrade the quality of the GTM

design. In this case, human designers may choose the constraints corresponding to Restrictions 2 and 3. Limiting unit function design to one memory port reduces many unit function design options. Limiting FPGA chip design to the same region functions reduces the complexity of region function selection and binding to a linear complexity.

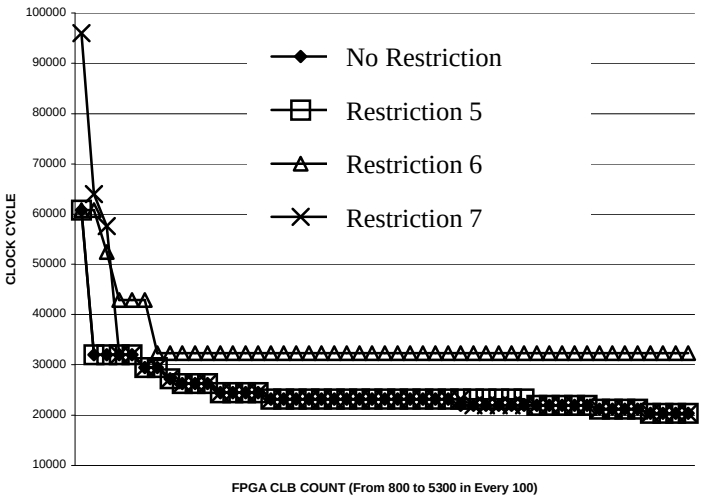


Fig. 10: Mapping Results Under Restrictions on FPGA Buffer

For the same example, Fig. 10 shows the number of clock cycles of the GTM mapping results under the restrictions on the FPGA buffer when the FPGA CLB count is from 500 to 5,300. It can be seen that the packing strategy (Restriction 6) affects the optimality of the mapping results very much. Therefore, the buffering with packing strategy is valuable for the speedup of GTM operations. When FPGA area is not too small (less than 800 CLBs), the buffer with one memory port (Restriction 7) corresponds to the same mapping result as those without restrictions. Without hybrid buffer (Restriction 5), the mapping result is not as good as that without the restriction when FPGA CLB counts are in a range from 3,400 to 4,000. Note that the hybrid buffer could affect the optimal mapping result more if the template window had more rows.

Note that the availability of the mapping tool enables the evaluation of board parameters with respect to a GTM application requirement. It provides useful information when a user is trying to devise an "optimal" structure of FPGA board, such as the number of FPGA chips, the number of external memory ports, the memory word width, and the size of FPGA, for a given performance requirement of a GTM operation. How to derive an optimal structure systematically is an interesting issue.

V. Conclusions and Future Works

The GTM operations cover a useful set of image processing algorithms, and their speedup by using reconfigurable computers has been shown in many research papers. However, the human design process for GTM operations is usually tedious even without exploring the whole design space and the compiler-like software tools for reconfigurable computing are far from efficient. Therefore, it is quite desirable to map this type of applications onto reconfigurable computers automatically with some degree of design space exploration.

In this research, the GTM operations are characterized and formulated as a special nested loop computation. GTM parallelism is explored by pipelining and by applying multiple copies of hardware units, such as multiple window functions in a unit function and multiple region functions in an FPGA chip. Various FPGA buffers are presented which provide design options for the tradeoff among the FPGA computation time, the FPGA area, and the memory size requirement. The design options also exist in circuit synthesis, FPGA chip and memory port binding, and image region partitioning and binding. The overall solution strategy is to enumerate the design space of basic pipelined FPGA design units and then select an optimal combination from these basic FPGA design units.

The enumeration process of the FPGA buffers and the computation cores is performed through the memory access pattern enumeration. Effective pruning algorithms are created to obtain all the non-dominated memory access patterns. Then the FPGA buffer is generated according to the memory access pattern, and the datapath and controller of the pipelined computation core circuit is obtained through the high-level synthesis under the constraints of the memory access pattern. Finally an optimal combination from basic design units is selected by a combinatorial optimization process of FPGA chip and memory port binding, and image region partitioning and binding.

The GTM mapping procedures and algorithms developed in this work are the basis of an automatic GTM software design tool that can produce a near optimal design, boost designers' productivity, and improve design portability. The current design tool can produce VHDL codes for the mapping results targeting the WildForce FPGA board.

The GTM mapping on reconfigurable computers involves a wide range of research topics. Many assumptions were made so as to limit the scope of the research. Relaxing some of these assumptions will certainly enlarge the application range of the mapping methodology. Several limitations and possible future works are listed in the following.

1. The FPGA design area estimates should be improved. Currently routing areas are not considered at all. Further study is needed. In our current mapping tool, an FPGA area utilization ratio is used to overcome the inaccurate area estimate. After FPGA placement and routing, if the design does not fit, the utilization ratio is reduced so as to force the tool to produce a smaller design. The process is iterated until a small enough design is found.

2. The clock frequency of a design should be considered in the optimization process. Two technical issues are involved. (1) How to accurately estimate the clock frequency without going through placement and routing? (2) How to modify the optimization process to incorporate the frequency estimates?

Because the problem domain is limited to GTM, there is a chance to come out with rules of thumbs about frequency estimates.

3. Hardware sharing overhead needs to be considered in terms of (multiplexer) area and design clock frequency. In the current implementation, hardware sharing is always performed whenever possible, even when the shared hardware has less area cost than that of the added multiplexers. A potential solution is to check the area overhead and add a clock cycle time constraint. Since routing delays can be expected to be a problem for the time constraint, further study is needed.

4. The pipeline synthesis of the unit function currently is performed in a very limited way. It should be enhanced in the following aspects. First, the assumption that each operator in a dataflow graph has only one implementation should be removed. This will add another dimension to the design space and make the synthesis much more complicated. Second, the register sharing is not considered in the current synthesis algorithm. Third, the conventional compiler optimization techniques for a dataflow graph, such as constant folding, operator reduction, etc, are not considered. When the templates are fixed, the constant folding task is expected to be performed by the users through providing a simplified DFG.

There are many other limitations in the current implementation of the GTM mapping tool. For example, the window function is required to be template permutation-invariant. The template permutation is to change the ordering of pairs of the template weight and the image value at the same template locations. A window function is said to be template permutation-invariant if any template permutation does not change the value of the window function. In Example 1 in Section II a template permutation is to change the ordering of the nine pixels. Since it does not change the sum, the summation (the window function) is template permutation-invariant. This property comes into play when examining non-buffering cases. In these cases, the pixel nodes are grouped and are ordered in each group. There is a need to compute the locations of image pixels at all active points when the memory controller read the image data. With the template permutation-invariant property the memory controller can read image pixels at all active points in any order as long as using the same order as the template weights. This limitation simplifies the implementation of the memory controller, and can be fixed without difficulty.

REFERENCES

[1] J.S.N. Jean, X. Liang, B. Drozd, and K. Tomko, "Accelerating An IR Automatic Target Recognition Application with FPGAs," in IEEE Symposium on Field- Programmable Custom Computing Machine, pp. 290-291, April 1999.

[2] K. Chia, H. Kim, S. Lansing, W. Mangione-Smith, and J. Villasenor, "High-Performance Automatic Target Recognition through Data-Specific VLSI," in IEEE Transactions on VLSI Systems, Vol. 6, No. 3, pp. 364-371, 1998.

[3] M. Rencher, and B. L. Hutchings, "Automated Target Recognition on Splash 2," in IEEE Symposium on FPGA Custom Computing Machines, pp. 192-200, April 1997.

[4] S. Singh and R. Slous, "Accelerating Adobe Photoshop with the Reconfigurable Logic," in IEEE Symposium on FPGA Custom Computing Machines, pp. 236-244, 1998.

[5] W.E. King, T.H. Drayer, R.W. Conners, and P. Araman, "Using MORRPH in an Industrial Machine Vision System," in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1996.

[6] Scott Hemmert and Brad Hutchings, "An Application Compiler for High-Speed Binary Image Morphology", in IEEE Symposium on FPGA Custom Computing Machines, 2001.

[7] C. Thibeault and G. Begin, "A Scan-Based Configurable, Programmable, and Scalable Architecture for Sliding Window-Based Operations," in IEEE Transactions on Computers, VOL. 48, NO. 6, pp.615-627, 1999.

[8] [8] A. A. Duncan, "An Overview of the COBRA-ABS High Level Synthesis system for Multi-FPGA Systems," in IEEE Symposium on FPGA Custom Computing Machines, pp. 106-115, April 1998.

[9] Q. Wang, etc, "Automated Field-Programmable Compute Accelerator Design Using Partial Evaluation," in IEEE Symposium on FPGA Custom Computing Machines, pp. 145-154, April 1997.

[10] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in Proc. 4th Ann. ACM Symp. Parallel Algorithms and Architectures, pp. 316-322, 1992.

[11] E. Waingold, M. Taylor, et al, "Baring it all to Software: Raw Machines an attached processing unit," in IEEE Computer, pp. 86-93, September 1997.

[12] M.B. Gokhale, J.M. Stone, "NAPA C: Compiling for a Hybrid RISK/FPGA Architecture," in IEEE Symposium on FPGA Custom Computing Machines, pp. 126-135, April 1998.

[13] M.B. Gokhale and J.M. Stone, "Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks," in IEEE Symposium on Field Programmable Custom Computing Machine, pp. 63-69, 1999.

[14] D. C. Cronquist, etc, "Specifying and Compiling Applications for RaPiD," in IEEE Symposium on FPGA Custom Computing Machines, pp. 116-125, April 1998.

[15] C. Ebelling, "Mapping Application to the RaPid configurable Architecture," in IEEE Symposium on FPGA Custom Computing Machines, pp. 106-115, April 1997.

[16] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. "The Garp Architecture and C Compiler," IEEE Computer, April 2000.

[17] M. Hall, P. Diniz, K. Bondalapati, H. Ziegler, P. Duncan, R. Jain and J. Granacki, "DEFACTO:A Design Environment for Adaptive Computing Technology," in Proceedings of the 6th Reconfigurable Architectures Workshop (RAW'99), Springer-Verlag, 1999.

[18] B. Draper, W. Bohm, J. Hammes, W. Najjar, R. Beveridge, C. Ross, M. Chawathe, M. Desai, J. Bins, "Compiling SA-C Programs to FPGAs: Performance Results," International Conference on Vision Systems, Vancouver, p. 220-235, July 7-8, 2001.

[19] X. Liang, J.S.N. Jean and K. Tomko, "Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems," The Journal of Supercomputing, Special Issue on Engineering of Reconfigurable Hardware/Software Objects, 19(1):77-91, 2001.

[20] B. Bosi, G. Bois and Y. Savaria, "Reconfigurable Pipeline 2-D Convolvers for Fast Digital Signal Processing," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pp. 299-308, Vol. 7, No. 3, 1999.

[21] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," Fourteenth Annual Workshop on Microprogramming, pp. 183-198, October 1981.

[22] A. E. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," in Computer, 14(9):18-27, September 1981.

[23] M. Lam, "Software Pipelining: An effective scheduling technique for VLIW machines," in Proceeding of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, pp. 318-328, June 1988.

[24] M. Weinhardt and W. Luk, "Memory Access Optimization and RAM Inference for Pipeline Vectorization," in Proceedings of FPL'99, pp.61-70, 1999.

[25] P. Diniz and J. Park, "Automatic Synthesis of Date Storage and Control Structures for FPGA-based Computing Engines," in IEEE Symposium on Field Programmable Custom Computing Machines, 2000.

[26] X. Liang, “Mapping of Generalized Template Matching on Reconfigurable Computers,” PhD Dissertation, Wright State University, December, 2001.
[27] X. Liang and J. Jean, “Memory Access Scheduling and Loop Pipelining”, Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms”, pp. 183-189, Las Vegas, Nevada, USA, June 2002
[28] X. Liang and J.S.N. Jean, "Memory Access Pattern Enumeration in GTM Mapping on Reconfigurable Computers," in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, pp. 8-14, Las Vegas, Nevada, USA, June 2001.
[29] Wayne Wolf, Andres Takach and Tien-Chien Lee "Architectural Optimization Methods for Control-Dominated Machines", in Paul Composano and Wayne Wolf, editors, High-Level VLSI Synthesis, pp.231-254, Kluwer Academic Publishers, 1991
[30] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and System, Vol. 8, No. 6, pp. 661-679, July 1989
[31] Giovanni de Micheli, "Synthesis and Optimization of Digital Circuits", Mcgraw-Hill, Inc., 1994
[32] D. Gajski, N. Dutt, A. Wu and S. Lin, "High-Level Synthesis--Introduction to Chip and System Design", Kluwer Academic Publishers, 1992
[33] “WildForce™ Reference Manual”, Annapolis Systems, Inc.

Xuejun Liang (M’02) received his Ph.D. degree in computer science and engineering from Wright State University, Ohio, USA, in 2001, and his M. S. degree in mathematics from Beijing Normal University, Beijing, China, in 1985.

He was a graduate research assistant at Wright State University from 1997 to 2001. He served as an instructor, an assistant professor, and an associate processor, respectively, in the Department of Mathematics of Beijing Normal University from 1985 to 1997. Currently he is an assistant professor in the Department of Computer Science of Jackson State University, Jackson, MS 39157, USA. His current research interests include adaptive computing system, FPGA application, and computer architecture.

Jack Shiann-Ning Jean received the B.S. and M.S. degrees from the National Taiwan University, Taiwan, in 1981 and 1983, respectively, and the Ph.D. degree from the University of Southern California, Los Angeles, C.A., in 1988, all in electrical engineering. Currently he is a Professor in the Computer Science and Engineering Department of Wright State University, Dayton, Ohio. His research interests include parallel processing, reconfigurable computing, and machine learning.