Initial Results of GOM (GTM Optimal Mapping)

Jack Jean*, Xuejun Liang**, Xinzhong Guo*, Hua Zhang*, and Fei Wang*
* Computer Science and Engineering Department, Wright State University, Dayton, Ohio 45435, USA
** Computer Science Department, Jackson State University, Jackson, MS39217, USA
(jack.jean@wright.edu or xuejun.liang@ccaix.jsums.edu)

Abstract: The generalized template matching (GTM) operations can be accelerated using reconfigurable systems with field programmable gate array (FPGA) hardware resources. Manually design an optimal FPGA hardware for a GTM operation is an iterative process that normally takes weeks or even months. GOM (GTM Optimal Mapping) is a software tool developed to reduce the design time to days or even hours. The tool design flow and its major components are described in this paper. Several design examples are used to evaluate the tool and to identify the weakness of the tool.

Keywords: Generalized Template Matching, Configurable Computing, Field Programmable Gate Array (FPGA), Design Automation

1 Introduction

The generalized template matching (GTM) operations include image-processing algorithms for 2D digit filtering, morphologic operations, motion estimation, and template matching [3, 4]. Mapping GTM operations on reconfigurable computers automatically and optimally is desirable.

The computation of a GTM operation is to move a template pixel by pixel in scanning line order, similar to the "Sliding Window-Based Operations" (SWO) as in [7]. The GTM is more general in that all the pixels in a SWO window are involved in the window computation while in GTM a template may be quite sparse and only a low percentage of pixels in a window is involved in the computation.

The GTM computation can be formulated as a nested loop. The computation iterates through image regions, templates, and pixel locations. The loop body computation, called the *Basic Function* (BF), involves applying one template window at one image pixel location. For example, for a 2-D convolution that uses a 3x3 window, the BF consists of nine multiplication operations and the summation of nine numbers. The *image region* is a set of consecutive image rows. The image pixels in the template window that are used for the BF evaluation are called *active points*.

Figure 1 illustrates a typical GTM design on a reconfigurable computer based on an FPGA co-processor board. This paper assumes that only one FPGA chip on the board is to be used for GTM and there is only one memory port connected to the FPGA chip. (The limitations are imposed by the code generator part of the current tool even though the optimization part of the tool can handle more general cases.)
A typical GTM design includes three components: a computation unit, an optional buffer unit, and a memory interface controller.

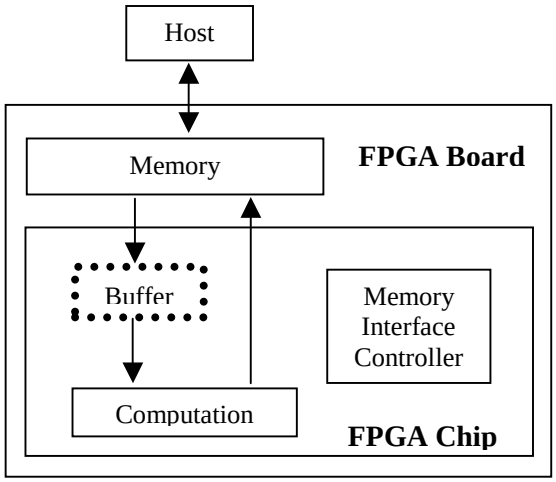


Figure 1: A typical GTM design

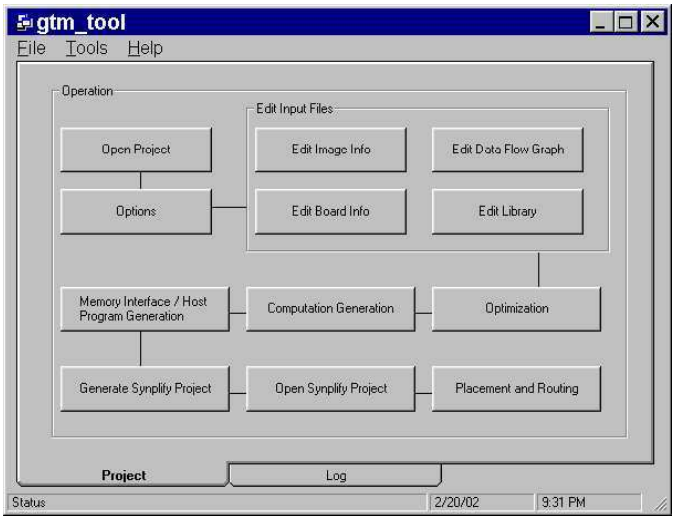
The computation unit normally consists of pipelined hardware that handles the parallel evaluation of one or more copies of BFs at several consecutive pixel locations. Using buffers to store image pixels inside the FPGA chip serves two purposes: (1) it can reduce redundant memory accesses as the template window moves, and (2) it can provide more data in parallel to the computation unit than a limited memory port can. However, they are achieved at the expense of more FPGA area. The memory interface

controller provides data to the right place at the right time.

The GTM design process for human designers is tedious because (1) there are many design options for the computation unit and the buffer, (2) the memory interface controller changes with design options and its design is error-prone, and (3) manual coding and evaluation of one design normally take at least several hours, most probably days, if not weeks. Those design options need to be evaluated so as to maximize the throughput subject to the FPGA board resource constraints (such as the FPGA size and the memory port width.) The software described in this paper evaluates designs through an optimization process and reduces the coding time with a code generator. It therefore can drastically improve a designer's productivity.

Section 2 describes the tool design flow and its major components. Section 3 presents several test cases that were used to debug and evaluate the tool performance. Section 4 concludes the paper.

2 GTM Mapping Tool



This section describes the tool design flow and its major components. The tool includes three major components, an optimal mapping tool, a code generator, and a library manager. These three components are integrated together through a graphical user interface (GUI), as roughly shown above, that schematically guides a designer through the design process.

The design flow of the tool is shown in Figure 2.

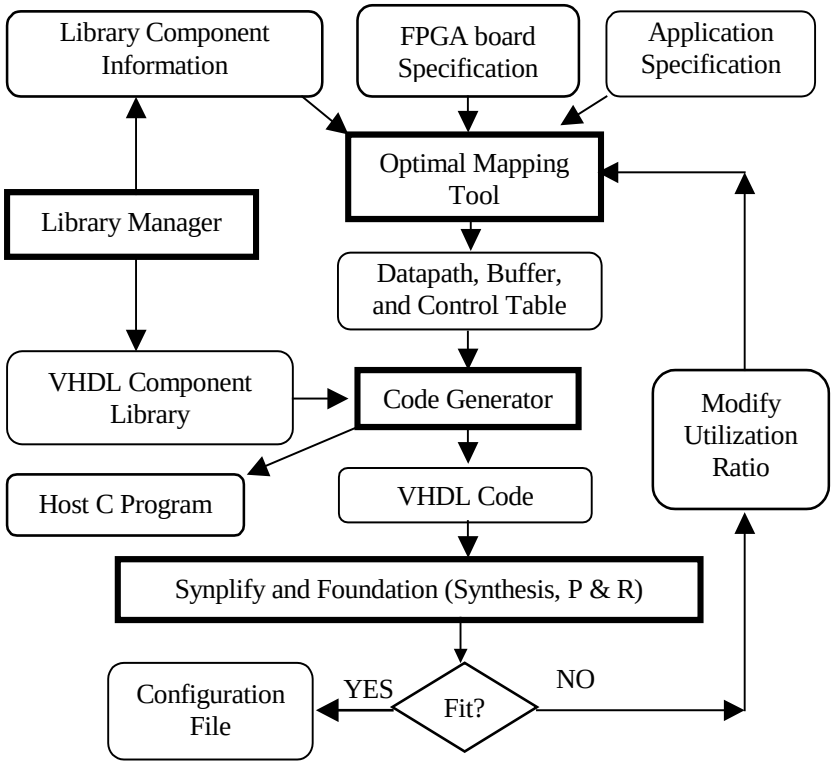


Figure 2: GTM design flow

- To use the tool, a designer specifies the GTM application using the GUI and a text editor.
- The optimal mapping tool can be invoked through the GUI so as to produce an optimal design subject to a hardware utilization ratio constraint. Here the utilization ratio is the percentage of FPGA CLBs (Configurable Logic Blocks) that is to be used by the design.
- The code generator can then be invoked, again through the GUI, so as to produce the VHDL files for the FPGA design and the C files for the host program.
- Synplify, a commercial VHDL synthesis tool, and Xilinx Foundation, a commercial FPGA placement and routing tool, are then used to produce the FPGA configuration file to be used at run time.
- Because the area estimates used in the optimization process may not be accurate enough, the placed and routed design may not fit into the targeted FPGA chip. In this case, the utilization ratio needs to be reduced through the GUI, and the design process should be repeated.
- The library manager can be used to maintain the VHDL operator components and the corresponding area and timing information required by the optimal mapping tool.

Each major component of the tool is described in more detail as follows.

2.1 Design Optimization

The objective of the GTM optimal mapping tool is to obtain a pipelined circuit design that maximizes the throughput subject to the FPGA CLB count constraint. Precise FPGA routing resource requirements are not considered. Building an optimal GTM design consists of two steps.

Step 1: Enumerate all non-dominated memory access patterns (MAPs)

The concept of MAP, as described in [5], is a key to the optimization process because both the computation unit and the buffer structure are determined from a MAP. (Roughly speaking, dominated MAPs are those that can be disregarded in terms of optimization consideration.) Efficient algorithms proposed in [5] have been implemented for this step. The throughput of each non-dominated MAP is proportional to the ratio of its packing factor (PF) over its data initiation interval (II) [5]. (The computation unit can process PF copies of BFs in parallel; but it takes II clock cycles for the pipelined computation unit to "consume" the PF copies of BF computation before it becomes ready for the next set of PF copies.) The enumerated non-dominated MAPs can therefore be rank ordered based on their throughputs.

Step 2: Derive GTM designs based on MAPs

This step itself solves an optimization problem:

> Given a non-dominated MAP, derive a minimal-CLB-count design that satisfies the MAP constraint.

The algorithm implemented in the tool involves graph scheduling and hardware sharing [3]. (Each BF is described as a dataflow graph and PF copies of DFGs are scheduled together.) The algorithm is applied to MAPs according to their throughputs, starting from the fastest one. The process stops when the minimal-CLB-count design is small enough to fit into the FPGA chip. That design is considered optimal and a synthesis process is applied to produce the corresponding data path description, control schedule, and buffer type.

2.2 Code Generation

The code generator receives design information from the optimal mapping tool and produces VHDL codes for the buffer, the computation part, and the memory interface controller. In addition, it produces a Synplify project file for the purpose of FPGA synthesis and a C code for the host program.

VHDL Code Generator

Given the data path from the optimal mapping tool, the generation of the VHDL files for the computation unit is straightforward, as long as the corresponding operators have been implemented and stored in the library. Generating the memory interface controller codes from the control schedule is more complicated mainly because different buffers have different initial data filling timing requirements.

The generation of the VHDL files for the buffers is handled differently. No operator library is used. Instead a C++ program was implemented for the code generator that produces five different buffers [3].

- A *full buffer* stores all the necessary pixels, including several image rows, so that only one new pixel needs to be read from memory when a template window moves.
- A *partial buffer* stores only the pixels in the current template window. As a result, when a template window moves, there is a need to read ***r*** new pixels, where ***r*** is the number of rows in a template.
- A *hybrid buffer* is a partial buffer with the ability to store some image rows.
- A *null buffer* means no pixel is buffered.
- An *internal buffer* is used only when a template is sparse and huge, e.g., one that has 70 active points in a 50x50 window.

Figure 3 shows a hierarchy diagram of the buffer generator. It illustrates the five buffers in terms of class inheritance. Dual port RAM, shift register, and crossbar are three base classes. Packing in the figure refers to squeezing more than one image pixels in a memory location and therefore a single memory access brings in multiple pixels into the buffer.

Host program The programming on a reconfigurable computer includes the FPGA hardware design and the writing of a host program that controls the hardware. Four host application programming interface (API) functions have been implemented so as to reduce the efforts involved in writing a host program. The API functions are GTM_FPGA_Open(), GTM_FPGA_LoadDesign(), GTM_FPGA_Execution(), and GTM_FPGA_Close().
The API implementation is board-specific and is currently for the WildForce™ FPGA board. The API functions provide controlled access to lower level functions and isolate the users from future changes in a low-level run time library. Based on the API functions, the code generator can produce a program skeleton that controls the FPGA board loading, execution and the host-FPGA communication. Some GTM specifications are passed to API functions as their function parameters. The skeleton strategy allows users to call the API functions directly in their own C/C++ programs.

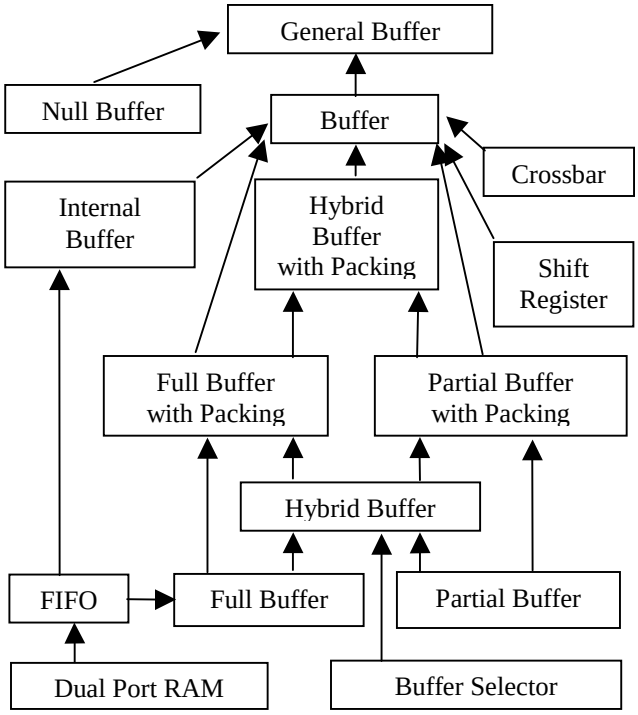


Figure 3: Buffer hierarchy

2.3 Library Management

The management of reusable VHDL components is important for the GTM tool. The optimal mapping tool needs a specification file of VHDL component library as input to list the interface, area and timing attributes of the components in the library. The file influences the optimization results. Because the optimization is always based on certain FPGA board and chip, which decide the area and timing attributes of a VHDL component, the information of FPGA boards and chips is also a reusable resource. A requirement built in for future tool extension is the searching of the different implementations of a certain operator in order for the optimal tool to choose the optimal one from multiple designs. The code generator also needs the VHDL component library to generate the final VHDL codes.

The current solution of the VHDL component management system for GTM is the VCL (VHDL Component Library) platform motivated by multi-tier applications for information management system. It is a three-tier architecture based on Microsoft COM (Component Object Model) technology. The details are omitted due to space.

3 Test Cases

The GOM tool has been tested with three test applications: (1) An artificial application which sums up all the active point values according to a 4 x 3 template. (2) A 2-D convolution with 3 x 3 and 5 x 5 template windows, and (3) a 2-D morphologic operation with 3x3 and 5x5 template windows. The first application was mainly used to verify the correctness of the VHDL/C codes generated, while the other two applications were also used to check out the FPGA design quality. The test platform is a WildForce™ FPGA board. Only one FPGA chip, a Xilinx XC4085, and its single memory port have been used. The memory port width (to the FPGA) is 32 bits.

To verify the test results, computation results from a software solution (without FPGA) and a hardware solution (with FPGA) were compared. For the cases tested the VHDL codes were successfully synthesized, placed, and routed. The host programs generated by the tool also proved to be very valuable for the verification.

3.1 Convolution with a 3x3 Template

The 3x3 convolution was applied to an image of 320 rows and 256 columns (320x256). Each pixel was 8 bits. To avoid the boundary problem, the template was applied only to the internal region of size 318x254, called *processing region*.

To specify the application, a DFG as shown in Figure 4 is needed for the BF. The DFG contains nine multiplication nodes and an adder tree. The dark circles indicate memory data, including nine inputs and one output. The nine inputs in white circles are for the weights of the 3x3 template. The DFG is specified as a text file to the tool. Each DFG node has an attribute about which library operator it needs. In the figure, some nodes are labeled with numbers, from one to six. Those numbers are the scheduled times for those nodes, in one of the situation tested (PF=1 and II=2). As a result, because nodes of the same operator type but scheduled at different times may share hardware, the tool produced a data path design as shown in Figure 5. Basically, two "sub-trees" in the DFG share the same hardware that contains four multipliers and three adders, while the bottom two DFG addition-nodes (labeled with 5 and 6) share the same adder. The input data are sent to the hardware in a pipelined fashion and, every two clock cycles, the hardware

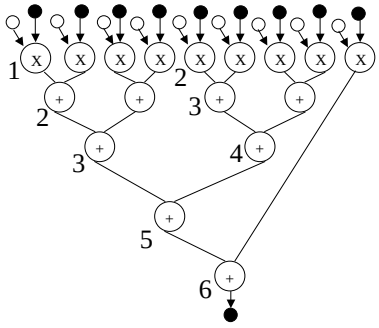


Figure 4: A DFG for 3x3 convolution

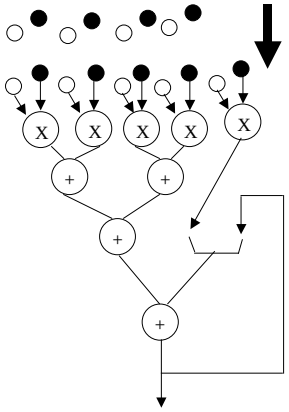


Figure 5: A data path design

can take a new problem instance (i.e., a new BF). That is the meaning of II=2.

To facilitate the generation of particular types of optimal designs, the tool allows the specification of design option restrictions in terms of PF and buffer type. It was observed that PF=1, 2, or 4 for this test case could be produced with correct results. (Note that when PF=4, a graph containing four copies of the DFG are scheduled for hardware sharing consideration.) In the process, different buffer types, such as full buffer and hybrid buffer, have been tested successfully.

To evaluate the design quality, a particular case (PF=1, II=2, full buffer) was used. The design quality was evaluated by examining the produced data path, the CLB count, and the FPGA execution time.

In terms of the data path, it was found that, unlike the data path in Figure 5, the data path produced by the tool used several more "redundant" multiplexers simply because no output register sharing was considered when DFG nodes shared the same hardware.

As to the CLB count, for the memory interface controller, it varied from 150 to 210 (or 5% to 7% of the 4085 chip area) for various processing region sizes, numbers of active points, and template sizes. For the whole GTM design, the CLB estimates produced by the tool was compared to both the Synplify estimates after synthesis and the Foundation results obtained after placement and routing. The GOM tool estimates were considered acceptable, although much more data need to be collected.

For the FPGA execution time, note that the host machine was a Pentium III 600-MHz machine with 512 MB of memory while the particular FPGA design ran at 40 MHz, even though the Foundation's estimate of the maximal frequency was only 25.458 MHz. All the time obtained was averaged over 100 execution loop iterations.

The Pentium program was compiled under the Visual C++ release mode. The execution time without FPGA was an average of 12.7 ms, which includes a division computation for result normalization. When the inner-most two-level nested loops for the 3x3 template computation were unrolled (as suggested in [2]), the average became 10.0 ms.

The real FPGA computation time computed based on the image size, the clock rate, and II (=2) was 4.1 ms. However when measured from the host the FPGA execution time, including various overheads, was an average of 23.4 ms. The overheads included the image input/output buffer formatting and normalization (9.92 ms), sending image pixels to FPGA board memory (4.60 ms), and reading results from the board memory (4.50 ms). The reason for the high overheads was because while the host supported the 32-bit data type and the PCI bus supported 32-bit data transfer, the FPGA design used the 8-bit data type. The overheads can potentially be reduced by packing image pixels and results, and by performing normalization on chip. In that case, the data transfer time can theoretically be reduced to one quarter of 9.10 ms, while the other overhead of 9.92 ms can be totally removed. That would greatly speed up the computation. Note that there is no such overhead in embedded applications.

For the convolution, a different version was tested that used saturated outputs without normalization. In that case, the software approach with loop-unrolling took 5.6 ms while the hardware approach (again measured from the host) took 21.93 ms. The data formatting time was 7.31ms. Again the real FPGA computation time was 4.1ms at 40 MHz.

3.2 Convolution with a 5x5 Template

Going from a 3x3 template to a 5x5 template mainly involves the modification of the DFG. Because the DFG description is in a text file, the input of a DFG is quite tedious. (Although a graphic editor is included in the tool, drawing a DFG is still not convenient.) Once the DFG was modified and the processing region specified, the GOM tool easily produced a design with PF=1. Unfortunately the design could only run at 10 MHz (with a Foundation estimate of 3.708 MHz). The reason was because the design used a large buffer which slowed it down.

Because Virtex and Virtex II chips have internal BlockRAMs that are more suitable for implementing larger buffers, the code generator was modified to produce designs that utilize BlockRAMs when available. For the resulting design, the Foundation estimates were 43.324 MHz and 62.861 MHz for XCV600-6 and XC2V1000-6, respectively. It was noted that the design automatically used the dedicated multipliers when the Virtex II chip was the target.

3.3 Morphological Operation

Starting from the convolution case, a morphological operation case was set up by modifying the DFG in terms of replacing each multiplication with an addition and replacing each addition with a maximum operator. When each pixel had 4 (or 8) bits, the output was checked against 15 (255) to produce saturated results. The FPGA results for the 4-bit case are summarized in the following table where all times are in ms.

	PF	II	Clock (MHz)	Total Time	FPGA Time	CLB Count
5x5	8	9	3	37.2	31.6	1773
5x5	4	5	20	12.1	5.3	1152
5x5	2	3	47	11.9	2.6	792
3x3	8	9	16	11.6	6.1	937
3x3	4	5	45	9.3	2.4	543
3x3	2	3	50	11.8	2.5	423

The pure software approach with loop unrolling took 9.2 ms and 19.6 ms for 3x3 and 5x5 templates, respectively. Two observations about the results:

1. The II selected by the tool was equal to PF+1. This is because result data were not packed and therefore it took PF cycles to output results to memory per window location. With output data packing, the minimal II would be two. (If two memory ports are available, then II can be one.)
2. Higher PF lead to lower clock rate. This was partly due to the higher II (more hardware sharing) and partly due to the more complicated buffer structures.

3.4 Current Limitations

Although the tool enumerates and evaluates many design options before it picks a final design, the optimality of the design is far from guaranteed because of the following reasons.

1. The execution speeds of designs are evaluated largely based on PF/II. A larger PF/II is assumed to lead to a faster design. This is over-simplified because different designs have very different clock rates. There is a need to characterize the design clock rates for the purpose of design evaluation without FPGA synthesis, placement, and routing.
2. The data to/from the FPGA board should be packed so as to fully utilize the data width of the PCI bus. The lower bound of II should depend of the data packing.

At this point, the FPGA design produced by the tool is not very attractive compared to the pure software approach. In addition the implementation of data packing, further efforts are needed to include the considerations of : (1) output register sharing, (2) low level timing constraints and floor planning, (3) better FPGA chips, such as Virtex or Virtex II chips, and (4) more memory ports per chip, which would allow higher PF and II.

4 Conclusions

This paper presents GOM, a software tool intended for optimally mapping generalized template matching (GTM) operations on reconfigurable computers. With the tool, the design time can be greatly reduced. Some weakness of the tool has been identified for further improvement.

The GOM approach is very different from many compiler development efforts for (C or C-like) high-level languages on reconfigurable computers [1, 6]. By restricting the target applications to GTM, the approach can take into account various data buffering structures and perform an in-depth optimization that is formulated as a constrained optimization problem.

Acknowledgements

This research was supported by a DAGSI/AFRL grant and an Ohio State research challenge grant.

References

[1] W. Bohm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar, “Mapping a Single Assignement Programming Language to Reconfigurable systems,” Supercomputing, 21:117-130, 2002

[2] E. Jamro and K. Wiatr, “Implementation of Convolution Operation on General Purpose Processors,” in the Proc. Of the Euromicro Conference on Multimedia and Telecommunication 2001

[3] Xuejun Liang, Jack Jean and Karen Tomko, "Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems", Supercomputing, Special issue on Engineering of Reconfigurable Hardware/Software Objects, Vol. 19, No. 1, pp. 77-91, May 2001

[4] Xuejun Liang, “Mapping of Generalized Template Matching on Reconfigurable Computers”, Ph.D. Dissertation, Wright State University, Nov. 2001

[5] Xuejun Liang and Jack Jean, “Memory Access Pattern Enumeration in GTM Mapping on Reconfigurable Computers,” in the Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms, pp. 8-14, June 2001

[6] T. Maruyama and T. Hoshino, “A C to HDL Compiler for Pipeline Processing on FPGAs,” IEEE Symposium on Field-Programmable Custom Computing Machines, 2000.

[7] C. Thibeault and G. Begin, “A Scan-Based Configurable, Programmable, and Scalable Architecture for Sliding Window-Based Operations”, in IEEE Transactions on Computers, pp. 615-627, 1999