Optimizing Data Intensive Window-based Image Processing on Reconfigurable Hardware Boards

Haiqian Yu
Northeastern University
Boston, MA 02115
Email: hyu@ece.neu.edu

Miriam Leeser
Northeastern University
Boston, MA 02115
Email: mel@ece.neu.edu

***Abstract*— Most image processing applications are not only computationally intensive, but also data intensive. Reconfigurable hardware boards provide a convenient and flexible solution to speed up these algorithms. To get a high performance design without going through the time-consuming hardware design process for each different algorithm, we present a simple design flow for window-based image processing applications. By finding the three upper bounds according to area constraints, memory bandwidth constraints and on-chip memory constraints, the block structure of the design which can fully utilized the available resources on the board is determined. A new buffering method is also discussed in this paper to build an efficient memory hierarchy for this type of application.**

I. Introduction

A Field Programmable Gate Array (FPGA) based computing board, working as a co-processor together with the host, can be used to speed up computational intensive and data intensive algorithms. Before actually implementing the algorithm in hardware, designers usually want to know what is the maximum speedup given the computing board and the algorithm. Unfortunately, the quantitative estimate of the speedup is dependent on the algorithm and the way it is implemented. The design space for hardware implementation for each algorithm is so huge that finding an optimal design and the corresponding speedup would require great effort. Defining a general rule to estimate the speedup for all the algorithms given a FPGA board is too complex.

This paper concentrates on one class of applications called Sliding Window Operation (SWO) based applications, which are widely used in image processing and require much computation and data manipulation. A common data access pattern is generalized and a near optimal memory hierarchy can be built using our block buffering method. By defining three upper bounds according to the area constraints, memory bandwidth constraints and on-chip memory size constraints, we can provide a fast yet accurate estimate of the maximal speedup. Moreover, once the tightest upper bound is selected, a corresponding hardware block structure as well as a performance estimate are determined. The design process presented in this paper allows the design cycle to be shortened and an estimate of the speedup to be obtained before circuit is actually implemented.

The remainder of the paper is organized as follows. Section II gives a brief introduction to SWO applications and FPGA boards. Related work is also discussed in this section. Section III presents the details of the design method and how to find the three upper bounds according to different resource constraints. Section IV describes a simple experiment to illustrate the design method. Section V concludes the paper and closes with thoughts about future work.

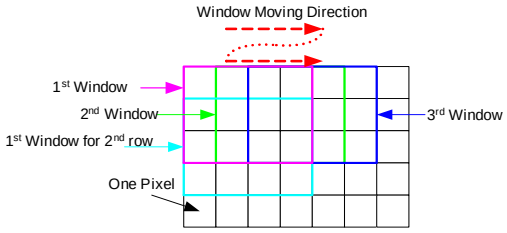

Fig. 1. Example of Sliding Windowing Operation

II. Background

Figure 1 shows an example of a 3×4 SWO. In this example, the window is moving in raster-scan order.

For most SWO applications, processing high resolution images is both data intensive and computationally intensive. Fortunately, SWOs are inherently highly parallelizable and hardware implementations are favored in delay sensitive applications. An FPGA-based system can provide a fast and flexible solution for these applications by combining both fine-grained parallelism and coarse-grained parallelism to achieve a high performance implementation.

The three basic elements of an FPGA are configurable logic blocks (CLBs), I/O blocks and programmable routing [1]. Arbitrary logic functions can be implemented by appropriately configuring these CLBs and connecting them through programmable routing. In this paper, we consider CLBs, I/O blocks, programmable routing and on-chip memory as our available FPGA hardware resources.

Memory hierarchy management had not been considered by FPGA designers until on-chip memory of FPGAs became available. FPGA chips usually have connections with external memory banks, but if the data needs to be read from or written to external memory banks on every access, the performance of the design is greatly degraded because accessing the external memory is usually slow. In this case, using on-chip memory to

form an efficient memory hierarchy becomes very important. The smaller but faster on-chip memory, if organized properly, can be used as a buffer to store temporary or repeatedly used data so that off-chip memory accesses are reduced. In our research, we find that for most SWO based applications, there exists a common data access pattern. Therefore, we present the block buffering method which can build an efficient memory architecture for this type of application.

Two design metrics are important for FPGA based design: time-to-market and performance. Lots of work has been done to shorten the design cycle or to improve implementation performance or both.

High Level Synthesis (HLS) tools provide a bridge between the algorithm written in a high level language (Matlab, C, C++, etc) and a lower level Hardware Description Language (HDL). Handel-C [2], SA-C (Single Assignment C) [3], Streams-C [4], The MATCH (MATlab Compiler for distributed Heterogeneous computing systems) compiler project [5] and DEFACTO (Design Environment For Adaptive Computing TechnOlogy) [6] all belong to this category. All of these design automation tools aim to raise the level of abstraction in hardware design to simplify the design process and therefore shorten the design cycle. The tools have to be general to deal with different algorithms and different hardware targets. This generalization leads to inefficiency of the design. Furthermore, most design automation tools do not efficiently use the on-chip memory of the FPGA and the performance is limited by the external memory accesses for data intensive applications. Diniz et. al. [7] and Weihardt et. al. [8] exploits the data reuse to minimize the number of memory accesses. Again, due to their generalization, the performance speedup over software are not satisfying.

To improve the performance of the design, further application specific optimization is needed. SWO based applications, widely used for image processing, have a common data access pattern and are suitable for further optimization. Guo et. al. [9] presented a compiler algorithm to reuse data in widow-based codes. However, hardware constraints are not considered and maximal performance is not guaranteed. Liang et. al. [10], [11] took memory access and data buffering into consideration, which is essential for data intensive applications. But the design space is very large especially when the size of the window or the number of memory banks is large. Moreover, their buffering method is not efficient when the available buffer size is not large enough for full image row buffering.

III. Design Tradeoffs

Our goal is to have an implementation which can fully utilize the available resources for optimal performance. By estimating the upper bounds of the parallelism according to different constraints, we can decide which constraint is the most critical one and then produce a design accordingly. For FPGA based COTS coprocessor boards, we assume that the FPGA chips and their connections to the external memory banks have already been built on the board.

There are three constraints we consider. First, the number of slices in the FPGA limits how many copies of processing elements we can put on the chip. Second, memory bandwidth defines the maximum data transfer speed between the FPGAs and the external memory banks. Third, the size of the on-chip memory that is used as buffers reduces redundant data transfer. Since our goal is to maximize the usage of all the available resources for maximum parallelism, we ignore timing constraints for each pipeline stage; they can be optimized after we determine the design structure. The following sections go into the details of how these three constraints influence our final implementation. Although most of our analysis is based on a coprocessor board with a single FPGA chip, the same analysis can hold for boards with multi-FPGA chips because most COTS systems have a symmetrical layout of the interconnection between FPGA chips and memory banks. We can divided the problem into several equal sized sub-problems by exploring coarse-grained parallelism. After finding solutions for the sub-problems, we can combine these solutions to achieve a final solution for the whole problem. Even if the interconnection is different for different FPGA chips, we can still use the same analysis process iteratively until we get an optimal division of the problem and allocate the sub-problems to different chips. The discussion that follows uses the parameters defined in Table I.

TABLE I
Parameter Definition

Parameter	Symbol
Image size	$M \times N$
Window size	$m \times n$
Pixel value of input image	$PI_{i,j}$
Pixel value of output image	$PO_{i,j}$
Bits per input pixel	W_{pi}
Bits per output pixel	W_{po}
No. of total memory bank	k
No. of input memory bank	k_{in}
No. of output memory bank	k_{out}
Total available buffer size	B_{total}
Memory bit width	W_m
Block buffer size (in pixels)	$p \times q$
Duplication factor under area constraints	D_a
Lower bound of duplication factor under memory bandwidth constraints	D_{ml}
Upper bound of duplication factor under memory bandwidth constraints	D_{mu}
Duplication factor under buffer size constraints	D_b

A. Area Availability

When we implement arbitrary logic in FPGAs, every function unit (FU), eg. adders, multipliers, registers etc., consumes one or more slices of the FPGA. The number of slices in an FPGA is proportional to the area in an ASIC and we use area here for simplicity.

Each FPGA has a fixed area (fixed number of slices) when it is manufactured; the area size is dependent on the model of the FPGA. Designs implemented in a specific FPGA chip are limited by the available area, which means the total area

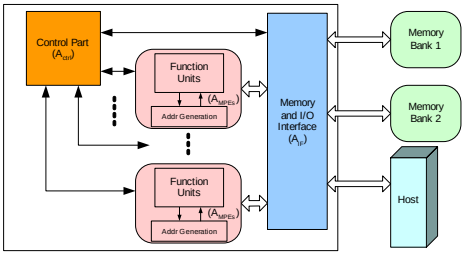

Fig. 2. Block Diagram of an Commercial FPGA Computing Engine

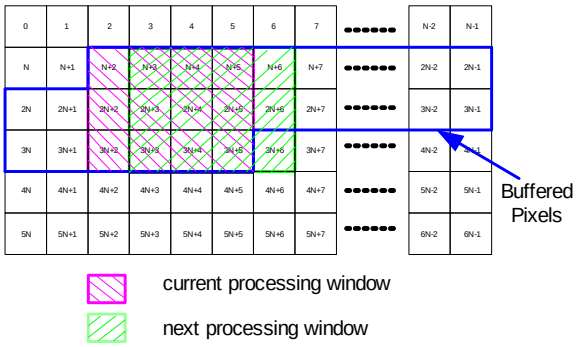

Fig. 3. Using Line Buffering Scheme

of all the functional units and routing cannot exceed the total area of that FPGA chip. It is extremely difficult to estimate the routing area, so we reserve 20% of the total chip area for routing purposes, which means about 80% of the total area (A_{total}) can be used for FUs.

SWOs involve a set of repeated operations at different image locations. We define a Micro Processing Element (MPE) as a set of pipelined function units which can process one SWO. Once we build the MPE, we can sequentially feed the pixel data from different windows into this pipeline and get the outputs one by one. Since each window is independent, if we have D_a duplicate copies of the same MPE, we can process D_a windows at different locations simultaneously and get a speedup of D_a times. Given the total available area, we can estimate the maximum D_a based on the area consumed by the MPEs and the overhead which coordinates these MPEs.

Maximizing D_a requires an accurate area estimate of the MPE, the control signals and the interfacing signals. We define A_{MPEs} as the sum of all function unit area for processing one window, which includes the address generation for one window. A_{IF} and A_{ctrl} are the area overhead of interfacing signals and control signals, respectively. A_{IF} is easy to determine because it will not change much when the design changes. In most cases, the interface modules are pre-defined and their area can be estimated before we start the design process. A_{ctrl} is the sum of the area consumed by the control part. The control part coordinates between MPEs so that they can share the memory interface without any conflict. It is very difficult to estimate the control signal area before it is actually implemented. Here we simplify the estimate by adding 15∼20% of the area of an MPE, based on experience, as the area consumed by the control part for one window processing. This overhead may be too pessimistic in some cases, but with current FPGA technology, it is affordable. Figure 2 shows the block diagram of the area consumption for different blocks. Equation 1 shows the constraint for maximizing D_a.

$$D_a \times (A_{MPEs} + 20\% \times A_{MPEs}) + A_{IF} \leq 80\% \times A_{total} \quad (1)$$

Once scheduling and binding of one MPE is fixed, the area is determined. Getting an optimal scheduling and binding scheme given area constraints is an NP-hard problem [12]. We assume that the MPE has already been designed. We further simplify our method by assuming there is sufficient area for one window processing. This assumption is reasonable for small size window SWOs and current FPGA technology. D_a can be easily modified according to Equation 1 when A_{MPE} changes because a different implementation is chosen. D_a gives us the number of parallel pipelines that fit in the available hardware.

B. Memory Bandwidth Limitation

In section III-A, we only consider the area constraints of the design. The feasibility of this implementation depends on whether or not we can have all the data ready for the function units. If memory bandwidth becomes a bottleneck, then even if we have D_a MPEs on chip, some of them will be idle until the data arrives. In this case, it is not necessary to have D_a copies of MPEs. We derive another upper bound from the memory bandwidth so that we can make sure all the copies of MPEs are fully working at all times.

As shown in Figure 1, each time the $m \times n$ window moves from left to right, m new data are needed from memory and one output is generated. If there is no buffering, we discard the m pixels which have moved out of the window. However, of these discarded pixels, $m-1$ pixels will be reused when the window moves to the next row. In this case, we load the pixels redundantly and increase the memory bandwidth requirements. If we have enough buffer space, we can keep the $m-1$ pixels in the buffer and discard only 1 pixel which will never be used for the following windows. By this means we can minimize the loading redundancy. Figure 3 shows a buffering method which can achieve this goal [13]. These two cases (loading m pixels per window and loading 1 pixel per window) can be defined as the lower bound and upper bound of the parallelism factor, D_{ml} and D_{mu}, respectively, according to the memory bandwidth limitation.

- Lower Bound D_{ml}:

$$D_{ml} \times (m \times W_{pi} + 1 \times W_{po}) \leq \sum_{i=1}^{k} W_{mi} \quad (2)$$

D_{ml} is subject to the constraints shown in Equation 3 because input and output memory ports are allocated separately.

$$\begin{cases} D_{ml} \times (m \times W_{pi}) \leq \sum_{i=1}^{k_{in}} W_{mi} \\ D_{ml} \times (1 \times W_{po}) \leq \sum_{i=1}^{k_{out}} W_{mi} \\ k_{in} + kout \leq k \end{cases} \quad (3)$$

- Upper Bound D_{mu}:

$$D_{mu} \times (1 \times W_{pi} + 1 \times W_{po}) \leq \sum_{i=1}^{k} W_{mi} \quad (4)$$

D_{mu} is subject to the constraints shown in Equation 5 because input and output memory ports are allocated separately.

$$\begin{cases} D_{mu} \times (1 \times W_{pi}) \leq \sum_{i=1}^{k_{in}} W_{mi} \\ D_{mu} \times (1 \times W_{po}) \leq \sum_{i=1}^{k_{out}} W_{mi} \\ k_{in} + kout \leq k \end{cases} \quad (5)$$

C. On-chip Memory Availability

Section III-B gives the upper bound and lower bound of the duplication factor under memory bandwidth constraints. To achieve the upper bound, Figure 3 shows the line buffer method requiring the buffer size to be at least $((m-1) * N + m) * W_{pi}$ bits. This may be larger than the buffer space available. Buffer size may be the constraint for determining the duplication factor. Our goal is to optimize the buffer size while still keeping the number of external memory data accesses as small as possible.

1) Block Buffering Method: We proposed a new method we call the block buffering method. It can greatly reduce buffer size while still keeping the data loading redundancy low. Figure 4 gives an illustration of the block buffering method. Before processing is begun, a $p \times q$ block of pixel data is buffered, where $p \geq m$ and $q \geq n$. In this example the window size is $m = 3$, $n = 4$ and the block size is $p = 4$, $q = 6$. With a $p \times q$ block buffer, we can process a total of $(p - m + 1) \times (q - n + 1)$ windows without loading any new data. While we are processing the current block, we can at the same time load the data for the next $p \times q$ block buffer. As shown in Figure 4, the total number of pixels needed to be loaded from off-chip memory is $p \times (q - n + 1)$ when the block moves horizontally. So the average number of off-chip memory access for this type of move is $\frac{p \times (q-n+1)}{(p-m+1) \times (q-n+1)} = \frac{p}{(p-m+1)}$ pixels per window operation. This is a significant saving when compared to off-chip memory access without buffering, which requires m pixels to be loaded per window operation.

When the block moves to the next row block, the number of pixels needed from off-chip memory to initiate the new windowing processing is $q \times (p - m + 1)$. Thus the memory access requirement is $\frac{q}{q-n+1}$ pixels per window operation.

Once we decide the values of p and q, we can determine the corresponding duplication factor D_b subject to buffer availability. Equation 6 shows how we can determine the value of D_b if we are only concerned about the memory requirement when blocks move from left to right.

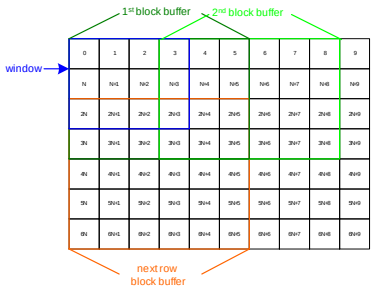

Fig. 4. Block Buffering Method Example

$$D_b \times (\frac{p}{p-m+1} \times W_{pi} + 1 \times W_{po}) \leq \sum_{i=1}^{k} W_{mi} \quad (6)$$

Again, D_b is subject to the constraints shown in Equation 7, because the input and the output memory ports are allocated separately.

$$\begin{cases} D_b \times (\frac{p}{p-m+1} \times W_{pi}) \leq \sum_{i=1}^{k_{in}} W_{mi} \\ D_b \times (1 \times W_{po}) \leq \sum_{i=1}^{k_{out}} W_{mi} \\ k_{in} + k_{out} \leq k \end{cases} \quad (7)$$

2) Selecting p And q: When we increase the value of p, we reduce the total number of external memory accesses from m to $\frac{p}{(p-m+1)}$ when the block buffer moves from left to right. At the same time increasing the value of q can reduce the memory access from n to $\frac{q}{q-n+1}$ when the block buffer moves to the next line. We wish to balance between p and q, where p, q are constrained by

$$p \times q \times W_{pi} \leq B_{total} \quad (8)$$

Here B_{total} stands for the total on-chip memory to be used for the block buffer. Given M, N, m, n, we can determine the total number of pixels need to be loaded from external memory, L_{total}, as a function of p, q (Equation (9)). Here, p, q are subject to the constraint in Equation 8.

$$\begin{aligned} L_{total} = & \underbrace{[(\frac{N-q}{q-n+1}) \times (\frac{M-p}{p-m+1} + 1) \times (p(q-n+1))]}_{\text{when block moves left to right}} \\ & + \underbrace{[(\frac{M-p}{p-m+1}) \times (q(p-m+1))]}_{\text{when block moves top to bottom}} \end{aligned} \quad (9)$$

Figure 5(a) plots L_{total} using $M = 1024$, $N = 1024$, $m = 3$, $n = 3$, $B_{total} = 500 * 8bits$. Clearly, the total number of external memory accesses mainly depends on the value of p. Moreover, from this figure, we can see that there exists a p_{opt} for a specific q such that when $p > p_{opt}$, the memory access requirement won't reduce much as we increase p. Our aim is to find this p_{opt} so that we can maximally reduce the external memory access requirement given the limited amount of buffer space.

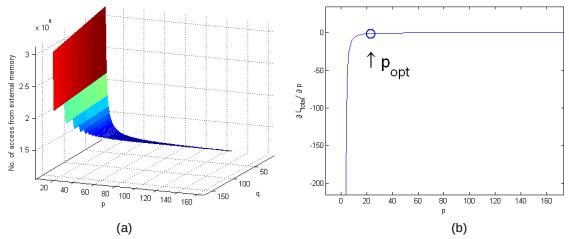

Fig. 5. (a)External Memory Access Requirement for Different p and q Values; (b)Deciding p_{opt} according to $\partial L_{total}/\partial p$

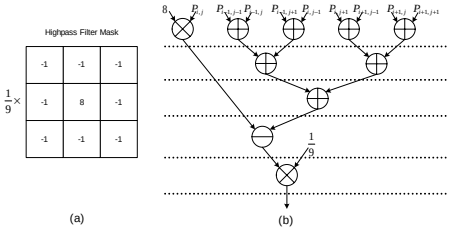

Fig. 7. (a)Highpass Filter Example; (b)Sequencing Graph for the High Pass Filter

Further analysis shows that p_{opt} is relatively independent of q. By setting the value of q to its minimum value n, we can get the total access number as a function of p. Taking the derivative of the function we can get p_{opt} according to the value of $\frac{\partial L_{total}}{\partial p}$. Figure 5(b) shows the derivative when we select $B_{total} = 500 \times 8bits$, $q = n$ and the other values are the same as used for Figure 5. Once p_{opt} is found, we can easily get the value of q, $q = \frac{B_{total}}{p_{opt}}$.

In the actual implementation, we usually take $q_{opt} = \frac{q}{2}$ so that we can overlap data processing and data loading time. Clearly, if we put $p_{opt} \times q$ pixels in the buffer, then when we want to move the block to the next position, we have to make sure the new data will not replace the old data before they are completely processed. Extra control mechanisms are needed to take care of the data conflict. But if we on use only half of the buffer by loading the block size $p_{opt} \times q_{opt}$, then while we are processing the current window, we can load the next block of data in parallel. Figure 6 shows that instead of using one whole block buffer, we can use two smaller buffers alternately and greatly reduce the waiting time. This adjustment will result in a small overhead of memory bandwidth requirement, but compared to the performance it can achieve, this overhead is negligible.

D. Summary

Once we get the three upper bounds according to different constraints, the tightest duplication factor is selected. Therefore, we can determine how many copies of MPEs will be implemented in the FPGA chip and which buffering method will be used. This information, combined with the delay and throughput of the MPE, can be used to estimate how many clock cycles are needed for loading and processing the data. To summarize, we can quickly determine the hardware block structure and the maximal performance by using our design method.

IV. Experiment

To demonstrate our method, we present an example. We assume:

- Input image size is 1024×1024, each pixel has 8 bits.
- The window applied to the image is a 3×3 high pass filter, as shown in Figure 7(a). Each output pixel also has 8 bits.
- The board we have only contains one FPGA chip with 12,288 slices, 10,000 bits of on-chip memory.
- 4 memory banks are connected to this FPGA chip, with data widths of 32, 32, 64 and 64 respectively.
- We assume the processing clock is the same as the memory read/write clock so that we do not have to consider cross clock boundary issues. Extra buffers and converting the data transfer rate into bits/processing cycle are needed if these two clocks are different.

Figure 7(b) shows a schedule using the ASAP scheduling algorithm [12] for our high pass filter example. The dotted horizontal lines correspond to clock cycle boundaries. We assume the multipliers and the adders have the same unit delay; this assumption can be modified depending on the library binding.

Table II shows the area usage for different function units, memory interfaces and control parts. We use 20% of one MPEs area as the estimate for controller area.

TABLE II
AREA USAGE OF HIGH PASS FILTER BLOCKS

Blocks		Slices Consumed
Memory Interfaces		1768
MPEs	Multiplier	69
	Adders	42
	Registers	52
	Address Generators	24
	Total	187
Control (20% of MPEs total)		38

According to Equation 1 and the data we get from Table II, we can derive the value of D_a as follows:

$$1768 + D_a \times (187 + 38) \leq 80\% \times 12288$$

Solving this equation, we get the maximum value $D_a = 35$. If we only consider area constraints, we can have at most 35 copies of one MPE and thus have 35 windows being processed simultaneously.

To process these parallel 35 windows continuously, we need to feed the input data and store the output data through the memory interface. According to our assumptions, we only have $(64+64+32+32)$bits/cycle memory bandwidth, which cannot meet the data transfer requirement. Obviously, area is not the critical constraint of the design and we have to consider the upper bound and lower bound of the duplication factor subject to the memory bandwidth constraint.

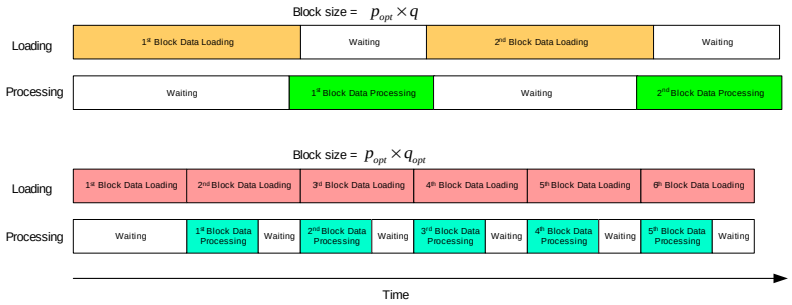

Fig. 6. Overlapping the Loading and Processing Time by Selecting $q_{opt} = q/2$

Based on the fact that the number of memory ports k is not large, we can enumerate all the input/output memory port allocation possibilities to obtain the maximum D_{ml} and D_{mu}. In this example, the optimal allocation would be to assign the two 64 bit width memory ports as input memory while the two 32 bit width memory ports are output memory for D_{ml}. By substituting these the numerical values into Equations 2 and 3 we get $D_{ml} = 5$, which is much smaller than $D_a = 35$. Using a similar strategy to calculate D_{mu}, we find that allocating one 64 bit and one 32 bit wide memory as input memories and the others as output memories, we can get the maximum value of $D_{mu} = 12$ under the constraints of Equation 5.

The number of simultaneously processed windows also depends on the size of the buffer. According the our analysis, we know that D_b satisfies the following condition:

$$D_{ml} \leq D_b \leq D_{mu}$$

and the value of D_b depends on the buffer size. In this experiment, we have a total of 10,000 bits of on-chip memory which can be used as the buffer. The line buffering method requires $(n-1) \times N + m$ pixels to be buffered, larger than the available on-chip memory size. In this case, we need to use the block buffering method instead.

Our computation shows given $B_{total} = 10,000$ (in pixels), the value of p_{opt} is 23. We select 24 as an optimal p value because it is much easier for hardware implementation. The corresponding q_{opt} value is 26. By allocating one 64 bit and one 32 bit width memory ports as input memory, the other 64 and 32 bit ports as output memory, we get $D_b = 11$, which doubles the duplication factor D_{ml} and is close to D_{mu}. The resulting optimal design has 11 copies of one SWO processing running in parallel. The block buffering method with block size of $p = 24, q = 26$ is selected for the best performance.

V. Conclusion and Future Work

In this paper, we determine the analytical representations of the three upper bounds according to three different constraints of the FPGA computing board. By selecting the tightest upper bound, a hardware block structure as well as an estimation in clock cycles is determined before we actually implement the SWO algorithm in hardware. Our block buffering method proves to be efficient and flexible for different on-chip memory sizes. We are currently investigating an automated tool using Integer Linear Programming (ILP) to automatically get the upper bounds given the board constraints and the parameters of the SWO applications. In the future, more SWO applications with a wide range of window sizes will be investigated. The results will be compared to previous hand-crafted designs to verify the accuracy of the estimates using our method.

References

[1] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicro FPGAs*. USA: Kluwer Academic Publisher, February 1999.

[2] "Handel-C, Software-Compiled System Design," *http://www.celoxica.com/methodology/handelc.asp*, Last accessed Dec 15, 2004.

[3] B. A. Draper, J. R. Beveridge, A. P. W. Bohm, C. Ross, and M. Chawathe, "Accelerated Image Processing On FPGAs," *IEEE Transactions on Image Processing*, vol. 12, no. 12, pp. 1543–1551, December 2003.

[4] M. Gokhale, J. Stone, and J. Arnold, "Stream-Oriented FPGA Computing in the Streams-C High Level Language," *FCCM'00*, pp. 49–56, April 2000.

[5] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A Matlab Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems," *FCCM'00*, pp. 39–48, April 2000.

[6] B. So, M. W. Hall, and P. C. Diniz, "A Compiler Approach to Design Space Exploration in FPGA-Based Systems," *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 165–176, June 2002.

[7] P. Diniz and J. Park, "Automatic synthesis of data storage and control structures for fpga-based computing engines," in *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2000, p. 91.

[8] M. Weinhardt and W. Luk, "Memory access optimization and ram inference for pipeline vectorization," in *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 1999, pp. 61–70.

[9] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A Quantitative Analysis of the Speedup Factors of FPGAs over Processors," *Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pp. 162–170, February 2004.

[10] X. Liang, J. Jean, and karen 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*, pp. 77–91, 2001.

[11] X. Liang and J. S.-N. Jean, "Mapping of Generalized Template Matching onto Reconfigurable Computers," *IEEE Transactions on VLSI Systems*, vol. 11, no. 3, pp. 485–498, 2003.

[12] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, S. W., Ed. McGraw-Hill, Inc., 1994.

[13] X. Liang and J. Jean, "Memory Access Scheduling and Loop Pipelining," *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, vol. 11, no. 3, pp. 485–498, June 2003.