Optimizing the FPGA Memory Design for a Sobel Edge Detector

Craig Moore, Harald Devos and Dirk Stroobandt
Hardware and Embedded Systems Group
Electronics and Information Systems Department
Ghent University, Belgium
phone +32 9 264 33 66 | fax +32 9 264 35 94
{craig.moore, harald.devos, dirk.stroobandt}@elis.ugent.be

***Index Terms*—Field programmable gate arrays, Memories, Memory architecture, Buffers**

***Abstract*—This research explored different memory systems on FPGA chips in order to show the various trade-offs involved with choosing one memory system over another. We explored the different memory components that are found on FPGA chips using the example of a Sobel edge detector. We demonstrated how the different FPGA chip's memories affected I/O performance and area. By exploiting the trade-offs between these a designer should be able to find an optimal on-chip memory system for a given application. Given further study, we believe we can develop application-specific memory templates that can be used with a hardware compiler to generate optimal on-chip memory systems.**

I. Introduction

Application specific hardware designs enjoy exponential gains in computation speed, compared to equivalent software designs, since they can take advantage of the highly parallel nature of their architecture. Hardware designers are free to utilize the hardware that runs their applications in any way they choose. However, this makes the size of their problem space much larger since they have more considerations related to hardware selection. The same is not true for software designers since many of the hardware considerations are handled by the compiler. We believe our research is important in this regard, since hardware compilers (or high level synthesis tools) are not yet able to efficiently choose the correct memory system for an application without significant human intervention. Furthermore, since the memory system can be a major bottleneck in system performance, it is important that hardware compilers be able to select the optimum on-chip memory system for a given application.

II. Metrics

There are four important metrics for hardware designs when it comes to memory: bandwidth, latency, size, and area. Using a traditional pipeline analogy, bandwidth would be the size (flow rate) of the pipe or the number of pipes used. Latency can be described as the length of the pipe(s). The longer the pipe, the longer it takes for the data to reach its destination and the slower the overall system. If items can be stored near their destination, then they will not need to be retrieved further upstream which reduces the overall execution time. This is why size is an important metric. If there is insufficient space to store all the values needed, then the system must replace stale values for new ones when needed. A well designed memory system should only replace those values that are no longer needed in order to minimize (or eliminate) multiple calls to external memory for the same value. Often, the larger the size of the memory required means a larger area occupied in hardware so area is also an important consideration.

There are typically three different types of memory available: registers, block memory (RAM), and external memory. As seen in Table I, each memory type has its own benefits and drawbacks. Registers are fast, but are evenly distributed on the FPGA so consume a larger amount of area when connected together to form a larger memory system. Block RAM on the FPGA, while slower than registers, is much more compact allowing it to occupy less area on the FPGA by comparison. External RAM, is much larger than the other two and occupies no space on the FPGA. However, it is much slower since data has to travel further to reach its destination.

TABLE I: Proposed taxonomy of trade-offs.

Type	Latency	Bandwidth	Size	Area
registers	++	++	±	– –
block memory	+	+	±	+
external memory	–	– –	++	++

± Depends on FPGA architecture selected

III. Exploration of Memory Systems

We explored the trade-offs between memory components using a Sobel edge detector because it offers a number of opportunities for data reuse, prefetching, and parallel access. The Sobel edge detector example is also similar to a number of other applications that use windowing operations, such as a FIR filter and wavelet transforms [1]. Sobel edge detectors are used in image processing to identify and isolate areas on an image with strong intensity changes from one pixel to the next.

We developed four designs which each have a different type of memory system to demonstrate the trade-offs in performance and area. Each design consisted of four components: an input buffer, the data path, an output buffer, and a simulated

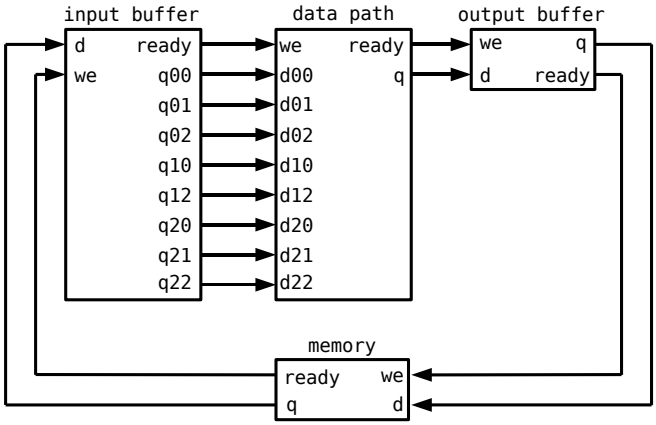

Fig. 1: System configuration used for each design

external memory as shown in Fig. 1. The designs each use a different type of input buffer, but have the same data path with a slightly different output buffer in each design. The first design uses primarily external memory and a very small input buffer of eight registers for the sliding window. The remaining designs all have input and output buffers that are able to reuse values without having to reread them from external memory. The second design's input buffer uses only registers and no block RAM. The third design's input buffer uses primarily block RAM, and the fourth design uses a combination of block RAM with registers.

After simulating each design, we verified that it produces the same output images as the C code which we based our designs on [2] and synthesized it using Altera's Quartus II Version 8.0 targeting a Stratix EP1S25F1020C5 board. The results are shown in Table II.

A. External Memory

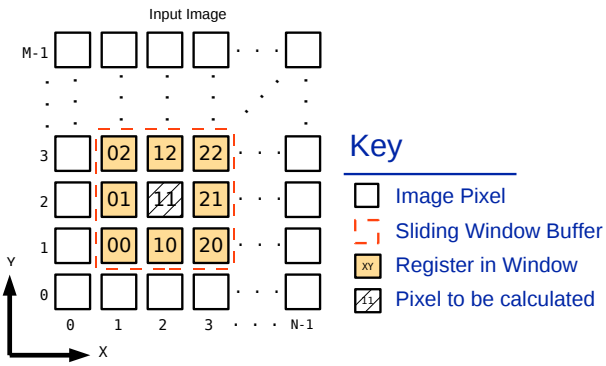

Fig. 2: External memory design: input image with input buffer registers centered at pixel 2,2

The first design listed in Table II consists of a small input buffer of eight registers where each register stores one pixel in the sliding window, as show in Fig. 2. Before the data path can begin calculating the first window, the input buffer must read eight pixels. Pixels are read in from external memory each time they are needed so there is no way to reuse pixels after they are used. As can be seen in the table of results, this design has the fastest clock speed, utilizes the least amount of resources on the FPGA, but has the longest execution time. This long execution time is due to the fact that no pixels are reused.

B. Registers with Data Reuse

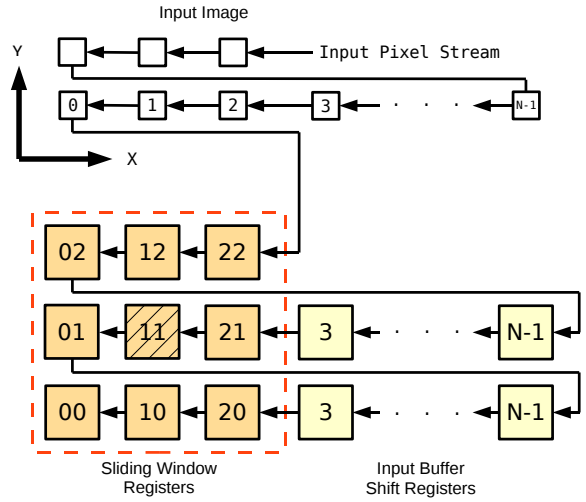

Fig. 3: Registers with Data Reuse: Input buffer using registers organized as a shift register

We can reduce the number of execution cycles by adding more registers to the input buffer so that pixels can be stored and reused until they are no longer needed. We did this by adding shift registers, as shown in Fig. 3 where values are read in sequentially from the input image as a stream of input pixels. The input buffer stores two rows of pixels from the input image and top three pixels of the sliding window. Values are then shifted through the sliding windows as shown by the arrows. When pixels are no longer needed they are simply shifted out of the input buffer. Before the data path can start processing the input buffer must first read in the first two rows and three pixels from the third row. Overall this was the fastest design compared with the others because it required the fewest number of execution cycles with almost the same clock speed as external memory design. However, because the input buffer is composed entirely of registers, which are distributed uniformly over the FPGA, it utilized the largest area on the FPGA.

C. Block RAM with Data Reuse

We can reduce the area utilized by using block RAM instead of registers in the input buffer as show in Fig. 4. However, this design has a longer pipeline latency because calculation can only start once the first three rows of the input image have been buffered. While having a slower clock speed and a larger number of execution cycles, this design utilized a much smaller area on the FPGA.

TABLE II: Table of Synthesis Results for each design.

Design	Frequency	Cycles[1]	Time[1,2]	Area[1,3]	Registers[1]	RAM[1,4]	Accesses[1,5]	Bandwidth[1,6]
1	142.82 MHz	810,275	5.67 ms	187	91	0 bits	808,992	142.93 B/s
2	139.80 MHz	102,726	0.73 ms	5,474	5,322	0 bits	102,400	140.27 B/s
3	123.20 MHz	104,000	0.84 ms	459	252	7,584 bits	102,400	121.30 B/s
4	134.73 MHz	103,683	0.77 ms	492	260	5,056 bits	102,400	133.06 B/s

1. Values when used with a 320×320 pixel, 8-bit encoded bitmap input image
2. Execution time $= cycles/frequency$
3. The total number of logic elements utilized by the design
4. Bits of on-chip RAM memory blocks utilized
5. Number of accesses to external memory made by the design
6. Required external memory bandwidth = $\big(memory\ accesses \cdot (bytes/access)\big)/time$

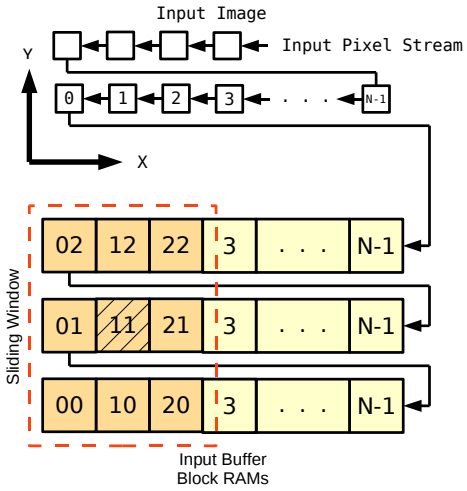

Fig. 4: Block RAM with Data Reuse: Input buffer using block RAM organized as a shift register

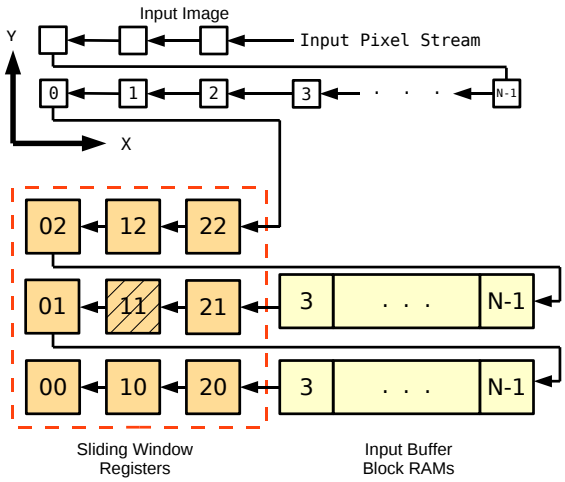

Fig. 5: Block RAM and Registers with Data Reuse: Input buffer using block RAM and registers organized as a shift register

D. Block RAM and Registers with Data Reuse

The final design is a combination of the register design with the block RAM design as seen in Fig. 5. This design has almost the same performance as the register design and utilizes almost the same area as the block RAM design. As show in Table II, this design makes the best trade-off between area and performance compared with the other three designs.

IV. Related Work

Many current hardware compiler projects can convert high level programming languages like C to a hardware description language (HDL) for FPGAs [3]–[11]. Part of this work involves developing the memory systems to be used for the applications it compiles. One component of this is identifying the design patterns of the application. A survey of such work can be found in [12]. Another component is developing the memory systems which are best suited for a particular application [13], [14], so that once the design pattern of the application is identified the compiler can pair it with the memory system best suited for that application. Our work has shown how important exploration of an application's memory system can be on performance.

Dong et al. proposed a universal memory structure to be used within high level synthesis of sliding window applications [14]. During our exploration, we utilized a similar memory structure to their system, as described in section III-D, with registers for the window pixels and block memory for buffering values from external memory. However, their design uses three rows of pixels in block memory while ours only requires two rows. Additionally, their system uses a *dataflow controller* in between the window registers and buffer memory. We do not need this because the values for the window are shifted into place without a controller's intervention. They have better clock speeds than our design, but without further info regarding their memory block utilization numbers we can not make a direct comparison.

Guo et al. also developed a buffer which uses input data reuse in window operations [1] and further extended it for simplified loop control [7]. We have found that their *smart* buffer can't hold enough values without having to reread values from external memory. However, the benefit of their buffer is that, like ours, a *dataflow controller* is not required.

A point not addressed in this paper, but important to be

aware of, is the problem of interfacing with external memory. Diniz et al. [15], [16] state that most designs do not interface directly with external memory, rather they use vendor-specific IP cores that have interface signals which can be utilized by the on-chip memory system. In their implementation of the DEFACTO compilation and synthesis system, they define two interfaces and a set of parameterizable abstractions to interface with existing synthesis tools. Using this type of implementation ensures that they can still use application-specific scheduling. They do this by using distinct ordering of memory accesses which interfaces with the vendor-specific memory IP core that handles the access to external memory. Since this is a fixed cost for any design, we did not take it into consideration, but in future work we will investigate it further to see if there may be optimizations to be exploited.

V. Conclusions and Future Work

We found that a combination of registers and block memory worked best for a Sobel edge detector because it makes the best trade-off between area and performance. It used a type of *smart* buffer that shifts values into the sliding window, without having to reread values from memory. We believe this design could be used with a hardware compiler and may be an improvement over the memory systems developed in [1], [14].

Future work will include, incorporating our memory design into the JCCI compiler developed by Devos et al. [17]. We will develope a parameterizable version of our memory design that can be easily adapted to the size of the system. Then we will look at how it performs with other applications using our compiler. We also want to investigate other performance metrics such as power and data locality like the work done by Cathoor et al [18]. Finally, we will be investigating adding an external memory controller that can be adapted to suit vendor-specific IP memory interfaces like the one used by the DEFACTO compiler [15].

VI. Acknowledgments

This research is supported by the I.W.T. grant 060068. Ghent University is a member of the HiPEAC Network of Excellence. The authors would like to thank Karel Bruneel and Wim Meeus for their valuable suggestions.

References

[1] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, July 2004, pp. 249–256. [Online]. Available: http://www.cs.ucr.edu/~zguo/

[2] B. Green, "Edge detection tutorial," http://www.pages.drexel.edu/~weg22/edge.html, 2002, last Accessed 2009.02.24.

[3] Mentor Graphics, "Catapult C synthesis," http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthe%sis/, Accessed 01.2009.

[4] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, *SPARK, A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004.

[5] Agility (Formerly Celoxica), "Handel-C," February 2009. [Online]. Available: http://www.agilityds.com/products/c_based_products/dk_design_suite/hand%el-c.aspx

[6] Impulse Accelerated Technologies, "Impulse C," http://www.impulsec.com/, Accessed 01.2009.

[7] Z. Guo, W. Najjar, and B. Buyukkurt, "Efficient hardware code generation for FPGAs," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, pp. 1–26, May 2008. [Online]. Available: http://www.cs.ucr.edu/\~zguo/papers/TACO_manuscript.pdf

[8] F. Diet, E. H. D'Hollander, K. Beyls, and H. Devos, "Embedding smart buffers for window operations in a stream-oriented C-to-VHDL compiler," in *4th IEEE International Symposium on Electronic Design, Test & Applications (DELTA'08)*, Hong Kong, January 2008, pp. 142–147.

[9] D. Ghica, "Function interface models for hardware compilation," School of Computer Science, University of Birmingham, UK, drg@cs.bham.ac.uk, Technical Report CSR-08-04, November 2008.

[10] B. So, M. W. Hall, and H. E. Ziegler, "Custom data layout for memory parallelism," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2004, p. 291.

[11] M. Bednara and J. Teich, "Automatic synthesis of FPGA processor arrays from loop algorithms." *Journal of Supercomputing*, vol. 26, no. 2, pp. 149–165, September 2003.

[12] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton, "Design patterns for reconfigurable computing," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM*, April 2004, pp. 13–23.

[13] H. Devos, "Loop transformations for the optimized generation of reconfigurable hardware," Ph.D. dissertation, Ghent University, February 2008.

[14] Y. Dong, Y. Dou, and J. Zhou, "Optimized generation of memory structure in compiling window operations onto reconfigurable hardware," in *International Workshop on Applied Reconfigurable Computing (ARC)*, ser. Lecture Notes in Computer Science, vol. 4419, 2007, pp. 110–121.

[15] J. Park and P. C. Diniz, "Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines," in *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*. New York, NY, USA: ACM, 2001, pp. 221–226. [Online]. Available: http://www.isi.edu/~pedro/PUBLICATIONS/isss2001.html

[16] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler, "Automatic mapping of C to FPGAs with the DEFACTO compilation and synthesis system," *Microprocessors and Microsystems*, vol. 29, no. 2-3, pp. 51–62, April 2005.

[17] H. Devos, W. Meeus, and D. Stroobandt, "CLooGVHDL and JCCI," in *DATE University Booth*, Nice, France, April 2009, pp. 1–2 (CD–ROM).

[18] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Boston, USA: Kluwer Academic Publishers, 1998.