FPGA implementation of a license plate recognition SoC using automatically generated streaming accelerators

Nikolaos Bellas, Sek M. Chai, Malcolm Dwyer, Dan Linzmeier
Embedded Systems Research
Motorola, Inc.
{bellas@labs.mot.com}

Abstract

Modern FPGA platforms provide the hardware and software infrastructure for building a bus-based System on Chip (SoC) that meet the applications requirements. The designer can customize the hardware by selecting from a large number of pre-defined peripherals and fixed IP functions and by providing new hardware, typically expressed using RTL. Hardware accelerators that provide application-specific extensions to the computational capabilities of a system is an efficient mechanism to enhance the performance and reduce the power dissipation. What is missing is an integrated approach to identify the computationally critical parts of the application and to create accelerators starting from a high level representation with a minimal design effort.

In this paper, we present an automation methodology and a tool that generates accelerators. We apply the methodology on an FPGA-based license plate recognition (LPR) system used in law enforcement. The accelerators process streaming data and support a programming model which can naturally express a large number of embedded applications resulting in efficient hardware implementations. We show that we can achieve an overall LPR application speed up from 1.2x to 2.6x, thus enabling real-time functionality under realistic road scenes.

1. Introduction

Embedded systems require designers to work under tight time-to-market, power dissipation, area, performance and cost constraints. The continuously increasing NRE and mask set costs for smaller transistor geometries make an ASIC standard-cell design applicable only to high volume products with a well-defined functionality that is not expected to vary considerably during the product life span. Reconfigurable computing using FPGAs has emerged in the last few years as a potential replacement technology in many cases. At the same time, there has been intensive research and commercial activity in tools that abstract out the hardware design process to the algorithmic level in order to further reduce the time to market. An architectural automation tool should be able to combine interactive architectural exploration, automatic hardware-software partition and an efficient mapping of one or multiple kernels to the reconfigurable fabric.

Typically, scalar processors like the PPC405 in the Virtex family of FPGAs, or the Nios synthesizable processor in Altera FPGAs are reasonably efficient in handling non-frequently executed or conditional code with a low degree of instruction and data level parallelism, even more efficient than mapping the same code into gates [14]. However, scalar processors are very inefficient for high throughput, parallelizable code due to limited support for parallelism (instruction, data, and task) and because of limited memory bandwidth from the memory hierarchy to the narrow pipes of the main core.

We have developed an automation process which maps streaming data flow graphs (sDFG) to accelerators of the main scalar core. An sDFG expresses computation kernels that process streams of data with a relatively limited lifetime and deterministic memory access pattern. The access patterns can be described independently from the computations of the sDFG. The streaming model decouples the description of the memory access sequences from the computation, thus making the customization of each of these two components easier and more re-usable. An example figure of an sDFG is given later when we discuss the LPR application.

To show the applicability of the streaming model in embedded systems, we describe how the process is used in the design of an Automatic License Plate Recognition (LPR) system. This is a stand-alone "smart camera" running an RTOS implemented using a SoC design methodology on a Virtex-II Pro FPGA [19]. Although the application at hand is LPR, the "smart camera" can be used in a variety of applications like automotive, security, home monitoring and control, etc. The aim is to offload the scalar PPC processor from the computational kernels that can be mapped into gates. Our methodology generates hardware accelerators from a large space of designs that follow a predefined template.

The application speed up, the required bandwidth and the size of the generated accelerators can be adjusted depending on application requirements, area constraints and user preferences. The contributions of the paper are the following:

- we propose the usage of the streaming paradigm in generating coprocessors in a reconfigurable fabric and

we outline a framework-based methodology that evaluates the set of potential solutions, and

- we detail how this approach is used in an autonomous LPR system.

The rest of the paper is organized as follows: Section 2 gives brief background information on the streaming programming paradigm and explains how it exploits technology trends that favor computation over communication. Section 3 explains our tool methodology, and Section 4 discusses the license plate recognition application and platform. Section 5 presents the experimental evaluation of the method, Section 6 gives a summary of previous work on the relative areas, and Section 7 presents the conclusion.

2. Streaming Programming Model

Our method produces coprocessors that process and produce data streams [1]. The streaming programming model exploits the "arithmetic intensity" [6] of VLSI technology by clustering execution units together and exposing data movement and staging to the programmer. Under the streaming model, the data fetching and storing units are decoupled from the computation unit, so that each one can be optimized separately and more efficiently.

The programmer describes the shape and location of data in memory using stream descriptors, and the computations using the sDFG. This decoupling allows the stream interface units to take advantage of available bandwidth to prefetch data before it is needed. The memory accesses are not computed using sDFG operations which allows for aggressive prefetching before data are requested by the data path. The architecture becomes dependent on average bandwidth of the memory subsystem with less sensitivity to the peak latency to access a data element. Data is transferred though the stream interface units which are programmed using stream descriptors (Figure 1). A stream descriptor is represented by the tuple *(Type, Start_Address, Stride, Span Skip, Size)*[1] where:

- *Type* indicates how many bytes are in each element (Type is 0 for bytes, 1 for 16-bit half-words, etc.)
- *Start_Address* represents the memory address of the first stream element.
- *Stride* is the spacing, in number of elements, between two consecutive stream elements.
- *Span* is the number of elements that are gathered before applying the skip offset.
- *Skip* is the offset that is applied between groups of span elements, after the stride has been applied.

1 We will limit our discussion in a single dimension of stream descriptors, because the majority of applications are covered by this model. Naturally, multidimensional or even non-rectangular spaces can be used under well defined semantics.

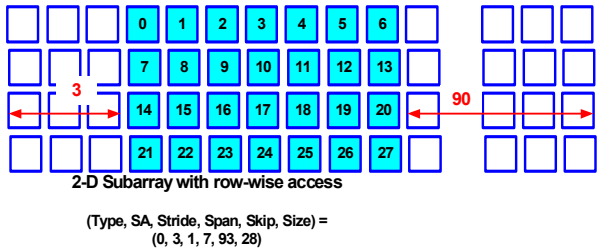

Figure 1: Stream descriptors for a row-wise rectangular access pattern

- *Size* is the number of elements in the stream.

Separately from the stream descriptors, the kernel computations are expressed using a streaming Data-flow Graph (sDFG) language. An sDFG consists of nodes, representing basic arithmetic, and logical operations and directed edges representing the dependency of one operation on the output of a previous operation [3]. Each node of the graph describes the stream operation type, the incoming inputs from the parent nodes, the size and signedness of the output result.

The input operands are specified as relative references to previous nodes rather than named registers. This feature helps eliminate the unnecessary contention for named registers as well as the overhead associated with register re-naming. The edges of the graph represent streaming data either between nodes or between an external stream source/sink and the sDFG.

3. Template-based hardware generation

3.1. Methodology

We have developed a framework to automatically generate synthesizable streaming accelerators. Our approach is to select designs from a well-engineered framework, instead of generating the given hardware from a generic representation of a high level language. We generate highly optimized designs at various points at the cost-performance space based on the given application, the user requirements, and the capabilities of the rest of the system. The main points of the tool flow are the following:

- a common template based on a simple data flow architecture that processes streaming data,
- an iteration engine that instantiates designs based on system parameters that meet system and user constraints to initiate the next iteration of space search,
- a scheduler that performs sDFG scheduling and hardware allocation based on the parameters set by the iterator,
- an RTL constructor engine that produces optimized Verilog code for the data path and the stream interface modules, and

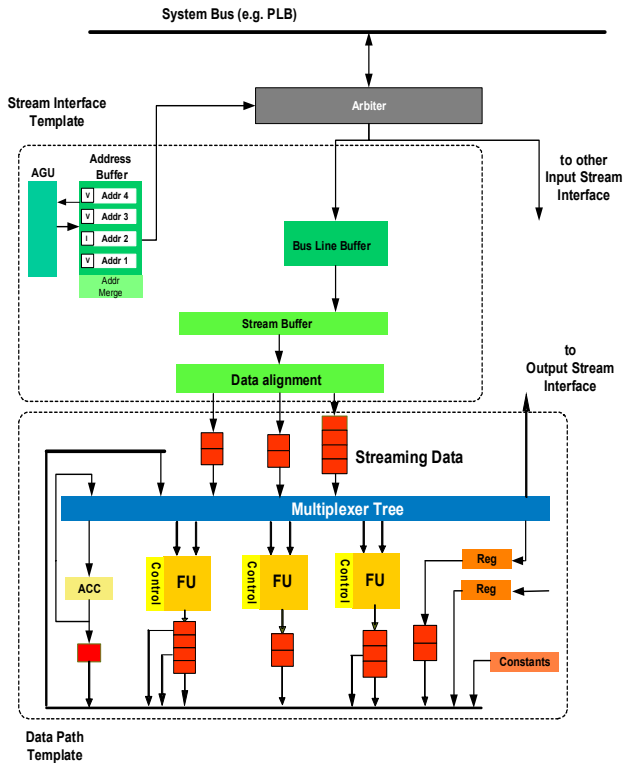

Figure 2: The accelerator template consists of the data path and the stream interface unit templates. Different optimizations criteria are used in each case.

- an evaluation phase that synthesizes the designs in an FPGA and produces quality metrics such as area, and clock speed

Each of the data path and the stream interface have their own acceleration generation process. The rest of the section details each one of these engines and their interfaces. For brevity, we will only outline the main points of the accelerator templates, without detailing the hardware generation algorithms.

3.2. Architectural template

The architectural template consists of two parts: the stream interface unit and the data path (Figure 2). The stream interface unit consists of one or more input and output stream modules, and can be generated to match the characteristics of the stream descriptors, and the characteristics of the bus-based system and the streaming data path. The stream interface unit is used to transfer data from a system memory or peripheral, through a system bus and present them in-order to the data path, and also to transfer processed data back to the memory.

The stream queue and the alignment unit store the incoming stream data and present them to the data path in-order. The number of storage elements, their size, and their interconnect depend on the stream descriptors and the requested bandwidth of the data path so that the number of elements is at least equal to the peak stream bandwidth requested by the data path. The bus line buffer is used to temporarily hold the data accessed from the system bus, and filter them to the stream queue when there is enough space. The address generation unit (AGU) is hardwired to generate the memory access pattern of the stream descriptors.

The data path can be generated to execute a given sDFG to match user and system constraints in the specification space. The data path template is an interconnect of reconfigurable functional units that produce and consume streaming data, and communicate via reconfigurable links. The links are chained at the output of a slice of a functional unit, and have a single input and potentially multiple outputs. They implement variable delay lines without the need of an explicitly addressable register file. The template also allow for the usage of a set of named registers that can be used by the sDFG to pass values from one sDFG iteration to the next and implement cross-iteration dependencies, and also to pass parameters to the program. Furthermore, the programming model allows for the use of accumulators for reduction operations [3].

The control logic of the data path is distributed and spatially close to the corresponding functional unit, multiplexer or line queue. We avoid using a centralized control unit, such as a VLIW control word, to reduce interconnect delays.

The type of the functional units (ALUs, multipliers, shifters, etc.), the specific operation performed within a type (e.g. only addition and subtraction for an ALU), the width of the functional unit, the size and number of storage elements of a FIFO, the interconnects between functional units (via FIFOs), the bandwidth from and towards the stream interface units, are some of the reconfigurable parameters of the data path. The data path requests data sourcing from the input stream interface unit and data sinking from the output stream interface unit. A simple, demand-driven protocol between the two modules is used to implement the communication. Stall signals from the stream interface unit to the data path allow for a less than perfect memory system. A stall signal from any stream interface will cause the stall of the accelerator engine.

4. License Plate Recognition

4.1. Application

License plate recognition (LPR) is a form of intelligent transportation technology that not only recognizes vehicles, but distinguishes each as unique. An LPR system is used for electronic toll or speeding ticket collection, secure-access control, law enforcement vehicle identification, etc. Each application has different performance and accuracy requirements. For example, in secure-access control, any character misprediction is unacceptable because that could allow unauthorized entry (false positive) or deny admission to an authorized person (false negative). On the other hand, in an

inventory control application, a false recognition may be acceptable in some circumstances if the error can be corrected. The factors that influence the design of an LPR system include:

- vehicle speed
- volume of traffic flow
- camera to license distance
- ambient illumination
- plate type and variety
- weather, etc.

An LPR algorithm consists of three processing stages: license plate detection, character segmentation and optical character recognition [8]. License plate detection is the most challenging and crucial stage in the whole system because a potential error will steer the system away from any solution. The aim in this stage is to detect the coordinates of the license plates under the typical assumption that their shape is a rectangular bounding box. Once this has been achieved, character segmentation detects the location of the alphanumeric characters in the box, and optical character recognition (OCR) verifies the characters against a preloaded symbol table.

The LPR system used in this paper is an extension of the algorithms used in [8]. The algorithm is based on the structural characteristics of a license place rather than the color variations [10] and is more stable under different lighting conditions. It is based on the observation that the license plates are patterns composed of several characters with a high contrast to their surrounding environment.

The LPR application, similar to a lot of computer vision and pattern recognition applications, consists of a series of low and intermediate (or high) level processing steps. Low level processing is applied on pixels as a series of imaging filters that eliminate unneeded visual information and enhance important cues to facilitate further semantic analysis. Intermediate and high level analysis extracts and processes higher level objects, like license plates, and attempts to analyze the scene further in order to detect presence of particular types of objects, to understand the meaning or content of an object, to study object interactions, and so on. LPR follows a similar processing pattern as shown in Figure 3.

The processing steps of the LPR algorithm used in our system are detailed in Figure 5 along with an example of a processed image of a vehicle. Special grayscale image sensors are typically used in automotive applications to provide enhanced infrared (IR) sensitivity for weak lighting conditions, global shutter for simultaneous total pixel exposure, and high dynamic range.

The Open operation is defined as an erosion operation on the whole image, followed by a dilation operation on the whole image using the same mask, whereas the Close operation is the inverse of the Open [7]. The erosion and dilation on the pixel with coordinates *(r,c)* in image *I* are defined as follows:

$$e(r,c) = \mathrm{MIN}_{i \in D1,\, j \in D2}[I(r+i, c+j)]$$

$$d(r,c) = \mathrm{MAX}_{i \in D1,\, j \in D2}[I(r+i, c+j)]$$

where D_1 and D_2 define the window of the mask applied on the pixel. The Open and Close are defined as follows:

$$\mathrm{Open}(r,c) = d(e(r,c))$$

$$\mathrm{Close}(r,c) = e(d(r,c))$$

The basic effect of the Erosion operator on a binary image is to erode foreground pixels close to large areas of background pixels. Often, these pixels are noise that should be eliminated before any subsequent processing. Another effect of the operator is to filter out clusters of foreground pixels that have a different shape than the mask. The Dilation operation enhances foreground areas that are close to background areas.

The Open and Close filters are a less destructive version of Erosion and Dilation, respectively. For example, the effects of the Open filter using a vertical 3x9 mask is shown in Figure 4, in which the aim is to eliminate all non-horizontal foreground pixels.

A horizontal opening and a horizontal closing with an Nx1 mask (N=7 in our case) and a subsequent image differencing detects the vertical edges of the image, including the vertical edges of the license plate and, at the same time, it de-emphasizes the horizontal lines (Figure 5).

The vertical edges in a license plate are adjacent to each other, so that a subsequent closing operation with a vertical 1xN mask connects them by propping up the horizontal lines between adjacent vertical lines. All the adjacent vertical lines form a connected region which includes the license plate and noise from areas of high visual contrast like the car periphery, trees, houses, road texture, etc. As we will discuss later, the existence of such noise in the picture creates large variations in execution time of the application. A threshold function transforms the image into black and white.

Connected components labeling scans the image and classifies the pixels into components based on pixel connectivity and intensity values, i.e. all white pixels that are adjacent belong to the same strongly connected component. The components correspond to regions that are candidates for the location of the license plate.

This step concludes the image-based low-level processing of LPR by producing the coordinates of the bounding boxes (BBs) of the regions. The next step of the plate location stage reduces the number of potential license plate regions by eliminating regions that have a small gross confidence value.

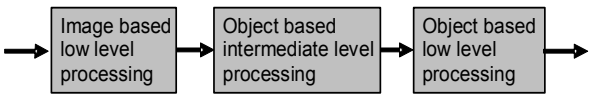

Figure 3: LPR processing features

Figure 4: The effect of an Open filter using a vertical 3x9 mask. Only vertical lines with a height of at least 9 and width of at least 3 are preserved.

We apply geometric criteria to compute the gross confidence for a region as the weighted sum of confidence components based on the properties of license plates:

- The Area of the license plate must be between a minimum and maximum value.
- The Width/Height aspect ratio must be within a certain range. For example the aspect ratio for European plates is about 5:1, whereas for US plates is 2:1.
- The number of foreground pixels in the region, known as region density, must be larger than a threshold because the region is mostly filled.
- The region must be close to the center of the frame

In Figure 5 the license plate region has the highest gross confidence and is marked with a red box.

The last stages apply filtering transformation to each different region to extract and verify the characters.

The procedure attempts to minimize spurious noise between characters and between the plate edges and the characters by computing the horizontal projection of each column in the region and setting as background all the pixels in a column whose projection is less than a threshold. Then, a labeling of connected components follows to create character regions and eliminate background noise. After character extraction, each subregion is compared to a preloaded data base of alphanumeric symbols for the final matching. The matching is a pixel-wise comparison between the two symbols, as well as comparison of the horizontal projection of each column and the vertical projection of each row of the symbols. If the differences are within a threshold, the system declares a match and moves to the next character. A number of filtering steps are also included in the last few stages of the algorithm but are not explained here because of limited space. Their role is to eliminate noise in the license plate to make character identification and separation easier, and to resize the characters to the same size as the symbol table in the data base.

4.2. Implementation using a stream accelerator

Morphological filters have high degree of instruction and data level parallelism, low control overhead and

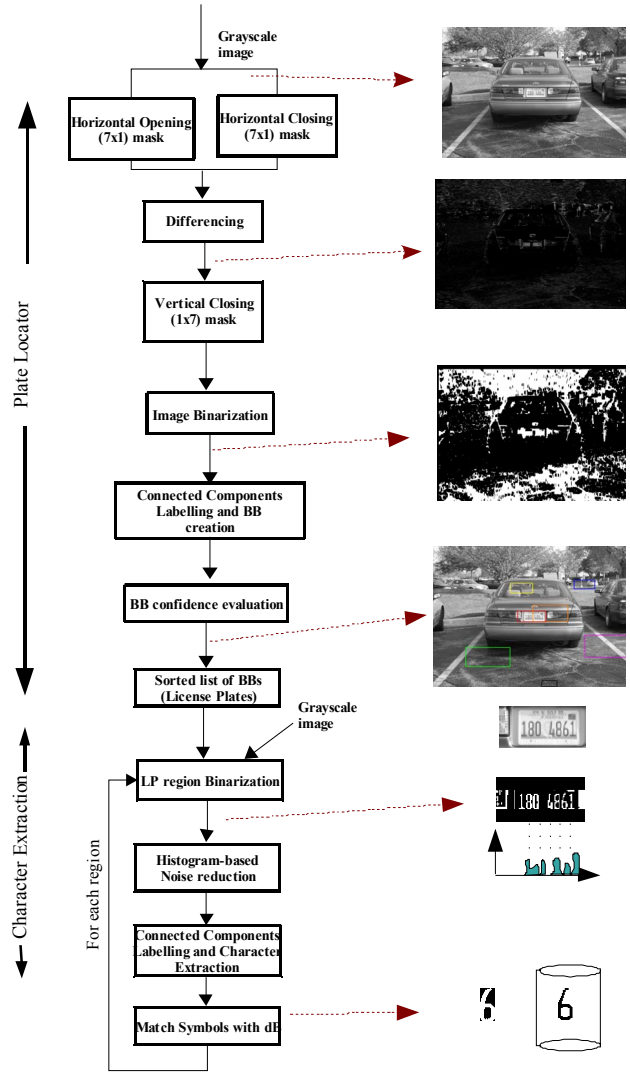

Figure 5: The LPR algorithm detects the bounding box (BB) coordinates of potential license plates. Then, it applies character segmentation and OCR in each BB.

operate on the whole frame in each invocation. We manually allocate the execution of parallelizable code on accelerators, and of sequential code with limited ILP to the embedded processor.

Table 1 shows the computational complexity of the main LPR processing steps for three input images in an ARM9 processor. The variation on execution time is due to the large number of connected components in the image because of ambient visual noise. For example, images similar to the one shown in Figure 5 cause the application to spend a large amount of time trying to merge all the foreground pixels into increasingly larger connected components through multiple passes in the image.

Table 1: Cycle distribution for three input images

			% execution time		
	Number of cycles (10^6)	*Number of initial connected components*	*Morphological Filters*	*Connected components labeling and confidence evaluation*	*Character extraction and match*
Image A	540	1029	21.2%	60.0%	18.7%
Image B	211	401	54.1%	23.1%	19.3%
Image C	147	219	77.3%	8.4%	10.2%

In all of our experiments, connected components outside a 200 pixel radius from the center of the VGA (640x480) image are assigned a lower confidence value and tend to be eliminated from further merging with other regions.

We used our tool methodology to accelerate the Opening and Closing morphological filters as well as the image Differencing and Binarization. By merging all the filters in a single sDFG (shown in Figure 6) we eliminate wasted bandwidth to fetch processed frames back and forth from the main memory.

In Figure 6 the *vtunnel* nodes use named registers (z_i)

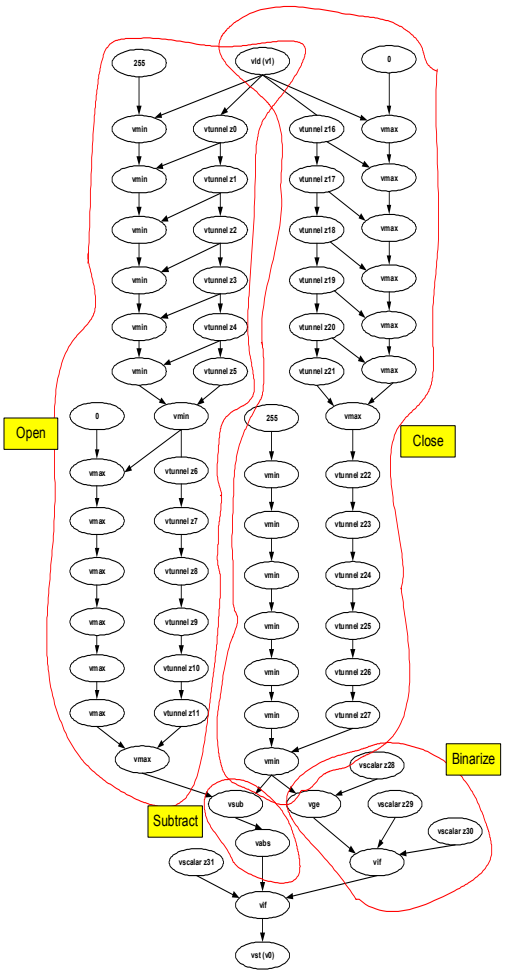

Figure 6: The sDFG of the morphological filters

to provide a single cycle latency to reuse incoming pixel values in the next sDFG iteration. The sDFG is invoked twice during a frame, once for the horizontal filters (Open, Close) and once for the vertical filters (Close), using a horizontal and vertical image scan, respectively. The *vif* (vector-if) node is used to select between the two cases at the end.

A separate accelerator is used in the MatchSymbols function to compute the horizontal and vertical projections of each license plate symbol, as well as the pixel-wise matching between a license plate symbol and each one of the symbols in the symbol table data base.

The connected components labeling and subsequent confidence evaluation, and connected components merging is not parallelized because it is mostly sequential code with limited parallelism.

5. Experimental Evaluation

The LPR system was implemented as a stand-alone platform based on a Virtex-II Pro FPGA [19] using the Xilinx Platform studio 7.1 and the Xilinx ISE 7.1. The Virtex FPGA is equipped with a PPC405 running the sequential code and the OS, and with a selected synthesized set of peripherals to communicate with the image sensors, the Ethernet, the USB 2.0, and external DDR memory. Two image sensors are used, one grayscale as the image source of the LPR application, and one color sensor as a backup for video replay. Separately, we have developed, in C and C++, the hardware accelerator generator which reads streaming DFGs, schedules them under different template configurations, and generates Verilog RTL for each case. The accelerators are manually connected to the PLB bus of the PPC processor to get a complete system, and the ISE toolset is used for the back-end hardware implementation. In the final design, the PPC core was clocked at 350 Mhz, and the rest of the circuit at one third of that.

For the results of Tables 1 and 2, we used a system-level simulator, and not the actual board. The memory system used in this simulation has the following characteristics: 18 cycles are needed to open a new row in the DRAM, and the first word is produced 21 cycles after the memory request. After that, new 64-bit data are

Table 2 Speed up of the LPR application and kernels

	Speed Up	
	Application	***Kernel***
Image A	1.2	5
Image B	1.76	5
Image C	2.62	5

burst through the PLB bus every three core cycles, up to a maximum of 256 bits, which is the burst size.

Table 3 shows synthesis and mapping results for the sDFG of Figure 6 under six different configurations. The c_i parameter refers to user constraints to the generator tool in terms of the maximum number of computational resources that the tool can utilize to schedule the sDFG. For this experiment, c_1 corresponds to a minimal configuration in which only one resource for each operational type is provided (e.g. only one adder, one port per input stream, etc.), c_3 corresponds to a very wide configuration with a large number of functional units and c_2 corresponds to an intermediate configuration, which is the same as the RSVP™-II implementation [4]. The u_i parameters show the degree of unrolling of the sDFG to achieve higher throughput. In wider configurations, sDFG unrolling can be an effective means to use resources that would otherwise remain unused, and increase the effective bandwidth. However, a higher degree of unrolling can strain the bus and memory resources and may result in low or negligible speed up. The results of Table 2 are obtained using a (c_2,u_1) configuration.

The results enforce our premise that a template-based approach can produce fast and area efficient designs. Wider designs (c_3 configuration) require more resources mainly because a large number of queuing elements at the output of each functional units is needed to temporarily store all the live variables at each cycle. On the other hand, wider configurations tend to be faster due to the lack of large multiplexers at the inputs of functional units. The last row refers to the throughput requested by the data path to the stream interface. This is an upper bound of the total bandwidth between the memory system and the accelerator.

The main limiting factor of the streaming accelerators is that a high bandwidth path to memory should exist to be able to keep up with the available computational power. The next step in our system integration is to develop a memory hierarchy that understands the semantics of the streaming model and is optimized for stream data transfer. A streaming memory controller can be used to optimize the scheduling of the DRAM memory accesses using a variety of techniques such as packing of streaming data, out of order accesses, merging of multiple requests, priority-based accesses, data buffering and staging and so on.

6. Related Work

There has been an intense interest in the research community in the last decade to automate the architectural process for ASIC of FPGA tool flows starting from a high level representation like C, Java, Matlab, DFGs etc. [5]. The PICO project from HP labs incorporated a lot of concepts from earlier work on VLIW machines, and described a methodology to generate a VLIW engine along with an accelerator optimized for a particular application [9]. Similar projects include the OCAPI tool from IMEC [13], the DEFACTO compiler from USC [15], the ASC streaming compiler effort from the Imperial College [11], and the CASH compiler from CMU that maps the complete the C application onto asynchronous circuits [15]. The Impulse-C [12] and Handel-C [17] languages are efforts to utilize C with extensions as a high level RTL language for FPGA design. At an even higher level of abstraction, AccelChip [2] is commercializing a compiler to automatically generate gates from Matlab code. The tool targets mainly DSP kernels on FPGA platforms.

Most of the above mentioned approaches use C as a more "user-friendly" hardware description language, and they add constructs to enhance concurrency, variable bitwidth, and so on in order to make C more amenable to hardware design. We believe that a template-based

Table 3: FPGA synthesis results for the LPR sDFG of Figure 6

	(c_1,u_1)	(c_1,u_2)	(c_2,u_1)	(c_2,u_2)	(c_3,u_1)	(c_3,u_2)
Data path slices	1181	1252	1124	1672	1957	3393
Stream interface unit slices	1531	1531	1531	1557	1531	1777
Total accelerator slices	2712	2783	2655	3229	3488	5170
Max data path clock frequency (Mhz)	165	163	156	126	203	204
Max stream interface clock frequency	140	140	140	140	140	148
Requested throughput (Bytes/cycle)	0.07	0.14	0.29	0.57	2	4

architectural automation that evaluates a large number of potential designs and focus on the most "profitable" parts of the code is able to offer both design efficiency in terms of speed and cost, as well as programmability for developers that are not well-versed in hardware design.

A number of companies have developed LPR platforms for a variety of applications[16][18]. Their approach is to either use high performance microprocessors or platforms based on embedded processors and FPGAs for the heavy duty tasks.

7. Conclusion

We presented a methodology and a prototype tool to automate the construction of hardware accelerators that process streaming data. In comparison to other architectural synthesis tools, we utilize a framework-based approach which is based on a well-engineered set of implementations and allows fast convergence to an area and speed efficient accelerator. This methodology focuses on kernels that can be parallelized while leaving the sequential code to be executed by the scalar processor. The streaming programming model allows the decoupling of data access and execution, and enables separate optimizations of these two modules to match the requirement of the application and the system.

We showed how this approach is used in a license plate recognition SoC. This methodology facilitates the field upgrade of such a system with new algorithms and new accelerators without costly re-designs of the system. Besides the license plate recognition system, our approach can be used in a multitude of applications that can naturally be expressed as a series of streaming filters such as communications, DSP, multimedia, image processing, etc.

References

[1] Amarasinghe S., Thies B. Architectures, Languages and Compilers for the Streaming Domain. *Tutorial at the 12th Annual International Conference on Parallel Architectures and Compilation Techniques,* New Orleans, LA

[2] Banerjee P. et. al. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. *Proceedings of the IEEE Symposium on Field Custom Computing Machines (FCCM)*, April 17-19, 2000, pp. 39-48, Napa Valley, CA

[3] Chirisescu S., et. al. The Reconfigurable Streaming Vector Processor, RSVP™. *Proceedings of the 36th International Conference on Microarchitecture,* December 2003, pp. 141-150, San Diego, CA

[4] Chirisescu S., et. al. RSVP II: A Next Generation Automotive Vector Processor. *IEEE International Vehicle Symposium*, June 2005

[5] Compton K., Hauck S.. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, vol. 34, No. 2, June 2002, pp. 171-210

[6] Dally W. J., Hanrahan P., Erez M., Knight T. J., Labonté F., Ahn J.H., Jayasena N., Kapasi U. J., Das A., Gummaraju J., Buck I. Merrimac: Supercomputing with Streams. *Proceedings of the 2003 Supercomputing Conference*, November 2003, pp-35-42, Phoenix, AZ

[7] Lee J., Haralick R., Shapiro L. Morphological Edge Detection. *IEEE Journal of Robotics and Automation*, vol. 3, issue 2, April 1987

[8] Jun-Wei Hsieh, Shih-Hao Yu, Yung-Sheng Chen. Morphology-based License Plate Detection from Complex Scenes. *16th International Conference on Pattern Recognition (ICPR)*, vol. 3, pp 176-179, August 2002

[9] Kathail V., Aditya S., Schreiber R., Rau B.R., Cronquist D., Sivaraman M. PICO: Automatically Designing Custom Computers. *IEEE Computer Magazine*, vol. 35, no. 9, September 2002, pp. 39-47

[10] K.K. Kim et. al. Learning based approach for License Plate Recognition. *IEEE Signal Processing Society Workshop on Neural Networks for Signal Processing.* Vol. 2, pp.614-623, 2000

[11] Mencer O., Pierce D. J., Howes L.W., Luk W. Design Space Exploration with a Stream Compiler. *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*, December 2003, Tokyo, Japan

[12] Pellerin D., Thibault S. Practical FPGA Programming in C. Prentice Hall, 2005

[13] Schaumont P., Vernalde S., Rijnders L., Engels M., Bolsen I. A programming environment for the design of complex high speed ASICs. *Proceedings of the 35th Design Automation Conference (DAC)*, June 1998, pp. 315-320, San Francisco, CA

[14] Vidiu M., Venkataramani , Chelcea T., Goldstein S.C. Spatial Computation. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 9-13, 2004, pp. 14- 26, Boston, MA

[15] H. Ziegler H., Hall M. Evaluating Heuristics in Automatically Mapping Multi-Loop Applications to FPGA *Proceedings of the 13th International Symposium on FPGAs*, February 2005, pp. 184-195, Monterey, CA

[16] Atom Imaging Technologies, www.atomimaging.com

[17] Celoxica Corporation, Handel-C language reference manual, www.celoxica.com

[18] Citysync, http://www.citysync.co.uk/

[19] Virtex-2 FPGA handbook, www.xilinx.com

A patent is pending that claims aspects of items and methods described in this paper