Heterogeneous Hardware Accelerator Architecture for Streaming Image Processing

Cuong Pham-Quoc, Zaid Al-Ars, Koen Bertels
Computer Engineering Lab, Delft University of Technology
Email: {P.PhamQuocCuong,Z.Al-Ars,K.L.M.Bertels}@tudelft.nl

***Abstract*—This paper proposes a heterogeneous hardware accelerator architecture to support streaming image processing. Each image in a data-set is pre-processed on a host processor and sent to hardware kernels. The host processor and the hardware kernels process a stream of images in parallel. The Convey hybrid computing system is used to develop our proposed architecture. We use the Canny edge detection algorithm as our case study. The data-set used for our experiment contains 7200 images. Experimental results show that the system with the proposed architecture achieved a speed-up of the kernels by $2.13\times$ and of the whole application by $2.40\times$ with respect to a software implementation running on the host processor. Moreover, our proposed system achieves 55% energy reduction compared to a hardware accelerator system without streaming support.**

I. Introduction

Stream processing is a new computing paradigm in which a series of operations is applied to each element in a set of data. Stream processing not only can address the memory accessing bottlenecks by decouples computation and memory accesses [1] but also can be used for applications in which data is continuously generated during the computation. Streaming image processing is one domain of stream processing. It is widely used in many application domains such as digital film processing [2], medical image diagnosis [3], real-time detection of changes in environmental phenomena [4], autovision driver assistance [5]. The computation in these systems is intensive especially when the resolution of data images is increased. A pure software implementation usually does not satisfy the real time performance requirement. Therefore, hardware acceleration for such systems is a necessity.

Meanwhile, with the rapid development of technology, more and more transistors can be integrated into one system. The more transistors a system has, the more power the system offers. Nowadays, it is possible to integrate more than 7 billion transistors [6] into one system. However, we need to solve a couple of challenges such as power consumption, thermal emission, etc., when more transistors are integrated. Moreover, it is not easy and straightforward to develop all such above systems by using only hardware technologies such as FPGAs, ASICs, or integrated circuits. Hence, heterogeneous hardware accelerator represents one approach to overcome the challenges. A hardware accelerator system usually contains a host processor to execute some software functions of the application and some hardware fabric such as FPGA to accelerate some computationally intensive functions of the application. Another approach is homogeneous multicore systems. However, compared to homogeneous multicore systems, heterogeneous multicore systems offer more computation power and efficient energy consumption [7].

In this work, we propose an architecture for a hardware accelerator system to support streaming image processing. In this architecture, streams are sequences of images which are processed by some software functions on the host processor and hardware kernels implemented on the hardware fabric. Our proposed system contains a host processor (a general purpose processor) and hardware kernels (accelerating computationally-intensive functions of the application) with controllers that support stream processing (in our experimental implementation, we use FPGA as hardware accelerator fabric). The host processor is responsible for receiving or sending the input or output image from or to other devices such as camera recorder or secondary hard disk, respectively. It also executes some functions of the application which cannot be executed on hardware. The hardware accelerator fabric consists of different kernels that implement the computationally intensive functions of the application. These kernels execute in parallel with the input streams. A shared memory is used for data communication of the host and the hardware kernels.

We implement our proposed architecture on the Convey hybrid computing system [8] that contains an Intel Xeon 5408 CPU as the host processor and four Xilinx FPGAs as the hardware accelerator fabric. We use the Canny edge detection algorithm as our case study. A data-set contains 7200 images is used to test the performance of the proposed system. The experimental results show that our system achieve a speed-up of the overall application by $2.40\times$ compared to a pure software implementation on the host processor and by $2.20\times$ compared to a hardware accelerator system without streaming support.

The main contributions of the paper are as follows: (1) propose a hardware accelerator architecture for streaming image processing; (2) present a speed-up estimation model for a streaming image processing application using a hardware accelerator system; (3) analyze the results of the synthesized implementation of the Canny edge detection algorithm, a well-known edge detection algorithm, in the proposed system.

The rest of the paper is organized as follows. Section II introduces related work on hardware accelerator systems and streaming image processing as well as shows a background on Canny edge detection algorithm. Section III presents our proposed architecture in detail. Section IV illustrates how

we implement the Canny edge detection algorithm on the proposed system. Our experimental results are shown and analyzed in Section V. Finally, Section VI concludes the paper.

II. Background and Related Work

A. Heterogeneous hardware accelerator systems

Heterogeneous multicore design is one of two approaches to utilize such a high volume of transistors integrated in one system (another approach is homogeneous multicore design). Many heterogeneous platforms are now commercially available such as the Sandy Bridge processor [9] and the Intel Atom E6x5C processor [10]. Hardware accelerator systems represent one of the main approaches to implement a heterogeneous multicore design. In such systems, there is often one traditional general purpose processor that functions as a host processor and one or more hardware accelerators that function as co-processors to speed-up the processing of special kernels of applications running on the host processor.

Heterogeneous hardware accelerator systems can be implemented in one chip where the host processor can be a hardwire embedded processor such as PowerPC in Xilinx FPGA or soft processor such as the MicroBlaze or the Nios processor. The Molen architecture [11], the Warp processor [12] and the LegUp architecture [13] are some examples for this kind of hardware accelerator. Alternatively, different chips are used for the host processor and the hardware accelerator fabric such as P2012 [14] or the Convey hybrid computing system [8]. Our proposed architecture can be developed on both kinds of implementations.

B. Streaming image processing with hardware acceleration

There are two approaches for streaming image processing. The first approach uses sequences of image-pixels as the streams while the second one uses sequences of images as the streams. The first approach introduces some overhead for segmenting the input image and combining the result after the processing. However, this approach does not require a large memory to store the image. The work in [15] proposed an FPGA implementation of low latency 2-D wavelet transforms for streaming image processing in which a stream is a sequence of image-rows. The work in [16] implemented an algorithm skeleton to support streaming image processing on a heterogeneous platform containing an SIMD and an ILP processor.

The second approach of streaming image processing does not introduce the segmentation and combination overhead, but it requires a large memory to contain the images. The work in [17] proposed an Image-Set Processing streaming framework that uses the power of a heterogeneous CPU/GPU platform. In that work, the CPU is responsible for reading and writing an image while all image processing steps are done by the GPU. In contrast to that work, we use FPGA as accelerator and the host processor is responsible not only for reading and writing image but also for processing.

C. Canny edge detection algorithm

Canny edge detection [18] is a well-know and powerful edge detection algorithm. In this work, we use the Canny edge detection algorithm as our case study. The program flow can be clearly partitioned into four main steps: (i) using the gaussian filter to remove noises (*gaussian* function); (ii) determining the edge strength (*derrivative_x_y* and *magnitude_x_y* functions); (iii) applying non-maximal suppression (*non_max_supp* function); and (iv) applying hysteresis (*hysteresis* function). The most computationally intensive function is *gaussian*. The size of a filter matrix used by *gaussian* affects the execution time of this function and the quality of the result. Figure 1 shows the results of the Canny algorithm with different size of the filter matrix.

(a) (b) (c)

Fig. 1. (a) Original; (b) 6×6 filter matrix; (c) 3×3 filter matrix

III. Architecture

This section presents the execution model of the host processor and hardware kernels, our proposed architecture, and our proposed multiple clock domains.

A. Hardware-software streaming model

Figure 2 illustrates the hardware-software streaming model for a streaming image processing application using three hardware kernels named *kernel_1, kernel_2*, and *kernel_3*. In the rest of the paper, we use the following terminologies for our explanation:

- A *step* is a processing phase of the algorithm, e.g., the image processing algorithm in Figure 2 has five steps: an initializing step done by the host, three following steps done by kernel_1, kernel_2, and kernel_3 and a finalizing step done by the host;
- A *stage* is an execution phase in which one or more steps are executed in parallel on different input data, e.g., at stage 0 in Figure 2, kernel_1 processes image 1 while the rest of kernels are idle; at stage 1, kernel_1 processes image 2 and kernel_2 processes image 1 while kernel_3 is still idle.

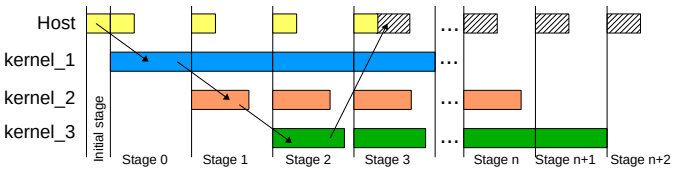

Fig. 2. The streaming model

In this work, a shared double buffer mechanism is used for data communication between the host processor and the hardware kernels as well as among the hardware kernels. Assume that the data dependency between the host processor and hardware kernels follows the arrows in Figure 2 where the kernel at the top of an arrow consumes data produced by the kernel at the root of the arrow. We need two buffers (named $k1\_i1$ and $k1\_i2$ meaning that the input buffer 1 and 2 for kernel_1) for data communication between the host processor and kernel_1. During the initial stage, the host processor transfers the pre-processed image 1 to $k1\_i1$. During stage 0, while kernel_1 processes data located on $k1\_i1$, the host processor pre-processes and transfers image 2 to buffer $k1\_i2$. During stage 1, while kernel_1 processes data located on $k1\_i2$, the host processor pre-processes and transfers image 3 to buffer $k1\_i1$. The same procedure is applied for the next stages. Similarly, we also need two buffers for each data communication between two hardware kernels such as $k2\_i1$ and $k2\_i2$ for data communication between kernel_1 and kernel_2. Two buffers named $k3\_o1$ and $k3\_o2$ are used for data communication between kernel_3 and the host processor. At stage 2, kernel_3 processes image 1 and writes its output to $k3\_o1$. During the next stage, kernel_3 writes the result of image 2 to $k3\_o2$ while the host processor does the post-processing for image 1 located in $k3\_o1$. When the first hardware kernel is started by a function call in the host processor (stage 0), the host processor sends the addresses of all the buffers to the corresponding kernels.

Due to the fact that we accelerate the most computationally intensive functions in hardware, the execution time of functions on the host is usually not longer than of the kernels. Assume that the execution time of kernel i for each image is t_i^k ($\forall i \in \{1,2,3\}$ - in the case presented in Figure 2) and of functions on the host for each image is t^s (we also assume that $t^s < min(t_i^k\big|_{i=1..3})$), the total execution time for the data-set in the case when stream processing is not applied (T_{nostr}) is as follows:

$$T_{nostr} = n \times (t^s + \sum_{i=1}^{3} t_i^k) \tag{1}$$

where n is the number of image in the data-set.

In the case when stream processing is applied (T_{str}) as described above, the total execution time for the data-set is as follows:

$$T_{str} = n \times max(t_i^k\big|_{i=1..3}) + max(t_i^k\big|_{i=2,3}) + t_3^k + t^s \tag{2}$$

Hence, the performance speed-up of stream processing compared to non-stream processing, S_p, is as follows:

$$S_p = \frac{T_{nostr}}{T_{str}} = \frac{n \times (t^s + \sum_{i=1}^{3} t_i^k)}{n \times max(t_i^k\big|_{i=1..3}) + max(t_i^k\big|_{i=2,3}) + t_3^k + t^s} \tag{3}$$

The speed-up (S_p) can be transformed as follows:

$$S_p < \frac{n \times (t^s + \sum_{i=1}^{3} t_i^k)}{n \times max(t_i^k\big|_{i=1..3})} < \frac{t^s + 3 \times max(t_i^k\big|_{i=\overline{1,3}})}{max(t_i^k\big|_{i=1..3})} \tag{4}$$

$$< \frac{t^s}{max(t_i^k\big|_{i=1..3})} + 3 \tag{5}$$

In general, the speed-up of a stream processing algorithm where m steps are accelerated by hardware compared to non-stream processing is lower than ($\frac{t^s}{max(t_i^k\big|_{i=1..m})} + m$). Therefore, to improve the speed-up, we should reduce the execution time of the longest kernel and increase the number of accelerated steps.

B. System architecture

Figure 3 depicts our proposed architecture with three hardware kernels representing for three kernel types: *input kernel* (kernel_1), *intermediate kernel* (kernel_2), and *output kernel* (kernel_3). The input kernel receives data from the host and processes the first accelerated step while the output kernel processes the final accelerated step and transfers the results of the kernels to the host. The intermediate kernel is responsible for the intermediate accelerated step. An image processing algorithm may require one or more intermediate accelerated steps corresponded to one or more intermediate kernels.

The shared buffers in the shared memory such as $k1\_i1$, $k2\_i1$ are explained in the previous section. Due to the shared memory, data is not need to be transferred from buffer to buffer. At each kernel, the *control* unit selects the right buffer for the current stage based on the number of images the kernel processed. Based on the number of images of the data-set, the control unit also determines if the kernel should continue executing. The host sends the addresses of the shared buffers and the number of images of the data-set to the kernels. This information is transferred through the *dispatch interface* unit.

The *core* unit in each kernel is responsible for the main task of the kernel. The core units process data stored in their local *buffers*, usually the on-chip memory. The *load/store* unit in each kernel is responsible for loading data from the corresponding shared buffer (the shared memory) to the local buffer when the kernel is started. The *load/store* unit writes the result from the local buffer to the corresponding shared buffer when the core finishes (*end_op* signal is asserted). The core unit is started (*start_op* signal is asserted) when input data is ready.

The kernel is launched whenever its *start* signal is asserted. The starting of the kernels is synchronous with each other. A kernel is going to be started if other kernels which are responsible for the previous steps, are executed before. In other words, the kernel can be started if and only if all the *done* signals (the signal used for synchronization) of all the kernels are active and input data for the kernel is ready. The *done* signal is active if no workload needs to be done by the corresponding kernel. For example, in the first stage, kernel_1 (in Figure 3) is invoked by the *start* signal from the dispatch

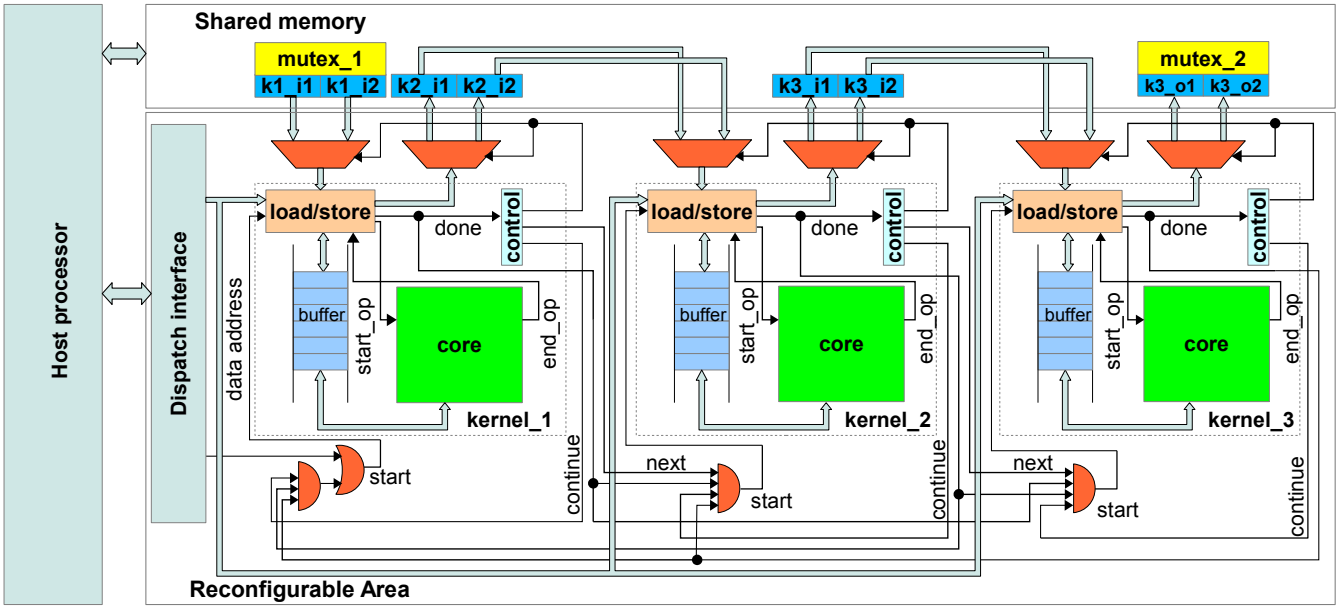

Fig. 3. The system architecture supporting pipeline for streaming applications

interface (function call from the host) while the other kernels are idle because their input data is not yet ready (their *done* signals are active). When kernel_1 finishes, its *done* signal and *next* signal are asserted to notify kernel_2 that input data is ready. During the next stage, kernel_1 and kernel_2 process the next image and the output of kernel_1, respectively, while kernel_3 is not executed due to the de-asserted *next* signal of kernel_2.

C. Multiple clock domains

Because we separate the shared memory access operation and the computational operation, we can use different clock domains for the system. The load/store unit, control unit and dispatch interface use a default system clock frequency, which is usually at a moderate level. Meanwhile, the core can execute with a higher clock frequency to improve the performance. Assume that the system clock frequency is f_{sys} and the clock frequency for the core is f_{core}. The speed-up estimation in Equation 3 is modified as follow:

$$S'_p = \frac{n \times (t^s + \sum_{i=1}^{3} t_i^k)}{[n \times max(t_i^k|_{i=\overline{1,3}}) + max(t_i^k|_{i=\overline{2,3}}) + t_3^k] \times \frac{f_{sys}}{f_{core}} + t^s} \quad (6)$$

Due to the assumption $f_{core} > f_{sys}$, we have $S'_p > S_p$.

IV. Case Study: Canny Edge Detection

This section presents the implementation of the Canny edge detection algorithm (presented in Section II-C) using our proposed architecture. We use the ANSI C implementation version provided by the University of South Florida [19] in our experiment.

The *gprof* profiling tool [20] is used to identify the most computationally intensive functions. We accelerate four functions of the Canny application by hardware kernels. Those are *gaussian*, *derrivative_x_y*, *magnitude_x_y* and *non_max_supp*. The DWARV compiler [21] is used to generate a VHDL description for those functions (the core units in our architecture). We simulate those kernels to estimate their execution time by ModelSim. The simulation result shows that the most computationally intensive function, *gaussian*, takes about $2.2\times$ longer than the total execution time of the three other kernels, *derrivative_x_y*, *magnitude_x_y*, and *non_max_supp*. Therefore, we decide to implement two hardware kernels for the *gaussian* function, i.e., two images are processed by the *gaussian* kernels at each stage. Moreover, the kernels of the *magnitude_x_y* and *non_max_supp* functions can be started right after the finishing of *derrivative_x_y* kernel because their input data is ready right after the *derrivative_x_y* kernel finishes. Therefore, we modify the proposed architecture such that the three kernels *derrivative_x_y*, *magnitude_x_y* and *non_max_supp* can be started asynchronously with the *gaussian* kernel as depicted in Figure 4. During a stage (except Stage 0, Stage n and Stage $n+1$), while two *gaussian* kernels process two images (k and $k+1$ where $2 < k < 2n-1$; $2n$ is the number of images of the data-set), the rest three kernels are executed two times to process the results of the gaussian kernels in the previous stage (the image $k-2$ and $k-1$).

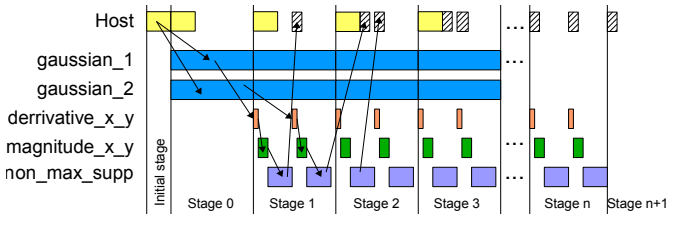

Fig. 4. The execution model and data dependency between kernels for the Canny algorithm

We use the Convey hybrid computing system [8] as a

hardware accelerator platform to develop our architecture. The Convey system consists of one Intel Xeon 5408 CPU, working at 2.14GHz, as host processor and four Xilinx Virtex 5 LX330 FPGA as the hardware accelerator fabric (so-called co-processor). Figure 5 depicts the Convey system architecture. The communication between the host and the co-processors is done by the HCMI bus. The shared memory consists of 128GB for the host processor and 64GB for the co-processors.

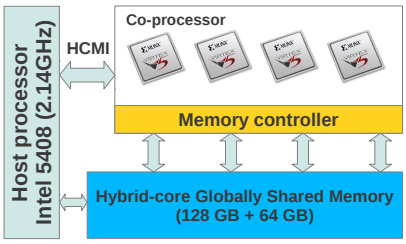

Fig. 5. The Convey hybrid computing system

In this case study, we configure each FPGA device with two *gaussian* kernels, one *derrivative_x_y* kernel, one *magnitude_x_y* kernel and one *non_max_supp* kernel. Therefore, we can process 8 images at one stage following the execution model in Figure 4. FPGA devices run independent from each other. The kernels in Device 0 process image i and $(i+1)$ in data set where i is multiple of 8. Consequently, the kernels in Device 1 process image $(i+2)$ and $(i+3)$; the kernels in Device 2 process image $i+4$ and $i+5$; and the kernels in Device 3 process image $i+6$ and $i+7$.

V. Experimental Result

This section presents our experimental results with the Canny application using a data-set containing 7200 images extracted from a 5 minutes video. The image size is 152×114 pixels. We set the standard deviation gaussian, low and high thresholds to 2.0, 0.5 and 0.5, respectively (i.e., we use an 11×11 filter matrix for the *guassian* function). Beside the system presented in the previous section, we develop two other accelerator systems for a comparison. Firstly, we run the software version on the host processor (Intel Xeon 5408 Quad-core working at 2.14GHz). Secondly, we develop a hardware accelerator system for the application without streaming but with multiple clock domains, i.e., following the non-streaming model presented in Section III-A and using f_{core} for the cores while f_{sys} for the rest units in the kernels. We, then, build another hardware accelerator system with streaming model but all units use the system clock frequency (f_{sys}). Finally, we implement the system with streaming model and multiple clock domains (the proposed architecture). All the systems have two hardware kernels for the *gaussian* function and one hardware kernel for each another function. In the multiple clock domains systems, the cores of the kernels are executed 225MHz (f_{core}) while other units run at 150MHz (f_{sys}).

Table I shows the execution time of the application with different systems and the speed-up compared to the pure software execution. The streaming system with multiple clock domains achieves a speed-up of the overall application by $2.40\times$ compared to the software execution and by $2.20\times$ compared to the non-streaming system (system 2). Moreover, the proposed system achieves a speed-up of the overall application by $1.47\times$ compared to the streaming system without multiple clock domains (system 3).

In System 3 and System 4, the speed-up of the kernels are lower than the speed-up of the whole application ($1.45\times$ compared to $1.63\times$ and $2.13\times$ compared to $2.40\times$, respectively) due to the fact that we execute software hardware streaming model. In order words, the software task for $n-1$ image is done in parallel with the hardware tasks (the kernels). The execution time of the whole application is the sum of the execution time of hardware kernels, software initializing step for image 1 and finalizing step for image n. Therefore, the execution time for the whole application is slightly longer than the kernels. Figure 6 shows the speed-up of both kernels and the whole application of the three last systems with respect to the first system - the software implementation only.

TABLE I
Application execution time and speed-up of different systems

System	Kernel time	Application time	Kernel speed-up	Application speed-up
System 1	46.88s	52.85s	$1.0\times$	$1.0\times$
System 2	40.46s	48.43s	$1.16\times$	$1.09\times$
System 3	32.44s	32.46s	$1.45\times$	$1.63\times$
System 4	21.97s	21.99s	$2.13\times$	$2.40\times$

System 1: Software only
System 2: Non-streaming model with multiple clock domains
System 3: Streaming with the same clock frequency for all units
System 4: Streaming with multiple clock domains (the proposed architecture)

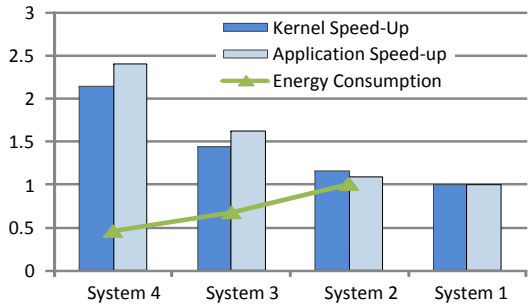

Fig. 6. The speed-up and energy consumption comparison between the systems

Table II shows the resource usage of the kernels in the streaming system with multiple clock domains (the proposed architecture) in term of the number of LUTs, the number of registers, and the number of DSP for each kernel (including the core, the load/store unit and the control unit). It also presents the total resource usage for the whole system (including other module used by the Convey system such as the memory controller and the dispatch interface). Each kernel has 132KB buffer implemented by block RAM (11.45% FPGA BRAM). The number of block RAMs used for the whole system is 904KB (78%).

We use Xilinx XPower Analyzer to estimate the power consumption for each FPGA device. The total power consumption of the hardware fabric in the three systems is almost

TABLE II
THE RESOURCE USAGE FOR EACH KERNEL AND THE WHOLE SYSTEM

Kernel	#LUTs	#Registers	#DSP
gaussian	10,231 (5%)	13,438 (6.3%)	7 (3.6%)
derrivative_x_y	2,907 (1%)	2,246 (1%)	0
magnitude_x_y	4,058 (1%)	3650 (1%)	5 (2%)
non_max_supp	8,571 (4.1%)	10,020 (4.9%)	5 (2%)
total	96,231 (46%)	113,781 (54%)	24 (12%)

Xilinx Virtex 5 LX330 contains 207,360 LUTs, 207,360 Register, 192 DSP and 1152 KB block RAM

identical. Table III shows the power consumption distribution of one FPGA device as well as the amount of resource usage (LUTs and FFs) for each system. The amount of resource usage for all three systems is almost identical. However, the power consumption of each resource type (clock, logic and BRAM) of System 3 is smallest because System 3 uses clock frequency at 150MHz for all component while the two other systems use a 225MHz clock frequency for some components. The power consumption of System 4 is larger than of System 2 because System 4 uses more logic elements (LUTs and FFs) than System 2. These logic elements are used for the streaming controllers.

We compute the energy consumption of the hardware fabric (without energy consumed by the host processor) by the product of power consumption and the execution time. Due to the short execution time, the system using streaming model with multiple clock domains (System 4) uses less energy than the non-streaming model system (System 2) and the streaming model system without multiple clock domains (System 3). Figure 6 shows the energy consumption normalized to energy consumption of the system with non-streaming and multiple clock domains. As shown in the figure, system 4 achieves 55% energy reduction compared to System 2.

TABLE III
POWER CONSUMPTION (W) AND RESOURCE USAGE OF THE SYSTEMS

	Resource	System 2	System 3	System 4
Power	Clock	0.501	0.383	0.502
	Logic	0.695	0.798	0.827
	BRAM	1.721	1.456	1.721
	Total	**18.334**	**18.038**	**18.481**
Amount	FF	112,384	113,781	
	LUT	102,836	104,534	

The systems are explained in Table I

VI. CONCLUSION

This paper presented a heterogeneous hardware accelerators architecture with multiple clock domains for a streaming image processing application. The work also introduced a model to estimate the speed-up of a streaming image processing system compared to a non-streaming one. The Canny edge detection algorithm was used as a case study. Experimental results show that the streaming system achieved a speed-up of the overall system by $2.40\times$ and of the kernels only by $2.13\times$ compared to the software implementation executed on Intel Xeon 5408 CPU. The experiment also compared the proposed system to the streaming system without multiple clock domains. The streaming system with multiple clock domains consumes less energy than the stream system without multiple clock domains although the system without multiple clock domains has a lower power consumption.

ACKNOWLEDGMENT

This work has been funded by the projects Smecy 100230, iFEST 100203, REFLECT 248976, and Vietnam Ministry of Education and Training.

REFERENCES

[1] M. Benjamin and D. Kaeli, "Stream image processing on a dual-core embedded system," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2007, pp. 149–158.
[2] V. Bove and J. Watlington, "Cheops: a reconfigurable data-flow system for video processing," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 5, no. 2, pp. 140–149, 1995.
[3] B. Davis, P. Fletcher, E. Bullitt, and S. Joshi, "Population shape regression from random design data," in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, 2007, pp. 1–7.
[4] C. A. Rueda-Velasquez, "Geospatial image stream processing: Models, techniques, and applications in remote sensing change detection," *PhD Thesis, University of California Davis*, 2007.
[5] C. Claus and W. Stechele, "AutoVisionreconfigurable hardware acceleration for video-based driver assistance," in *Dynamically Reconfigurable Systems*. Springer Netherlands, 2010, pp. 375–394.
[6] NVIDIA, "NVIDIA Kepler GK110 Architecture Whitepaper," 2012.
[7] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, 2005.
[8] Convey Computer, "Convey reference manual," 2012.
[9] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A fully integrated multi-CPU, GPU and memory controller 32nm processor," in *ISSCC*, 2011.
[10] "Intel® atom™ processor E6x5C series-based platform for embedded computing," 2010.
[11] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The MOLEN polymorphic processor," *Computer*, pp. 1363–1375, 2004.
[12] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," *ACM Trans. Embed. Comput. Syst.*, pp. 1–22, 2009.
[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *FPGA*, 2011, pp. 33–36.
[14] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *DATE*, 2012, pp. 983–987.
[15] O. Benderli, Y. Tekmen, and N. Ismailoglu, "A real-time, low latency, FPGA implementation of the 2-D discrete wavelet transformation for streaming image applications," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, 2003, pp. 384–389.
[16] W. Caarls, P. Jonker, and H. Corporaal, "Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, p. 9.
[17] L. Ha, J. Kruger, J. Comba, C. Silva, and S. Joshi, "ISP: An optimal out-of-core image-set processing streaming architecture for parallel heterogeneous systems," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 6, pp. 838–851, 2012.
[18] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. PAMI-8, no. 6, pp. 679 –698, nov. 1986.
[19] U. S. Florida, "Canny edge detector," 1999.
[20] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982.
[21] R. Nane, V. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, "DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 619–622.