Efficient OpenCL Accelerators for Canny Edge Detection Algorithm on a CPU-FPGA Platform

Samah Rahamneh and Lina Sawalha
Computer and Electrical Engineering
Western Michigan University
Kalamazoo, MI
Email: {samah.z.rahamneh, lina.sawalha}@wmich.edu

Abstract—**The processing demands of current and emerging applications, such as image/video processing, are increasing due to the deluge of data, generated by mobile and edge devices. This raises challenges for a vast range of computing systems, starting from smart-phones and reaching cloud and data centers. Heterogeneous computing demonstrates its ability as an efficient computing model due to its capability to adapt to various workload requirements. Field programmable gate arrays (FPGAs) provide power and performance benefits and have been used in many application domains from embedded systems to the cloud. In this paper, we used a closely coupled CPU-FPGA heterogeneous system to accelerate a sliding window based image processing algorithm, *Canny* edge detector. We accelerated Canny using two different implementations: code partitioned and data partitioned. In the data partitioned implementation, we proposed a weighted round robin based algorithm that partitions input images and distributes the load between the CPU and the FPGA based on latency. The paper also compares the performance of the proposed accelerators with separate CPU and FPGA implementations. Using our hybrid CPU-FPGA based algorithm, we achieved a speedup up to 4.8× over a CPU-only and up to 2.1× over a FPGA-only implementations. Moreover, the estimated total energy consumption of our algorithm is more efficient than a CPU-only implementation. Our results show a significant reduction in energy delay product (EDP) compared to the CPU-only implementation, and comparable EDP results to the FPGA-only implementation.**

Index Terms—**Hardware acceleration, heterogeneous computing, edge detection algorithms.**

I. Introduction

Heterogeneous Computing (HC) is gaining attraction in both academia and industry due to its ability to serve a vast range of application domains and adapt efficiently with their requirements. In addition, HC provides significant energy and performance benefits. Today, data centers handle a wide range of workloads such as machine learning, image/video processing, and high-performance financial algorithms [1]. The diverse characteristics of workloads have stimulated the deployment of heterogeneous architectures to accommodate applications' demands. A heterogeneous architecture is mashed up of different processing powers, for instance, Central Processing Unit (CPU), General Purpose Graphical Processing Unit (GPGPU), Field Programmable Gate Array (FPGA), etc. FPGAs have advantages over other accelerators in cloud-based environments because of their power and performance benefits [2]–[4]. As such, a large fraction of data center nodes is expected to include FPGA logic by 2021 [4].

Aiming at decreasing time-to-value and abstracting modern FPGAs complexity, High Level Synthesis (HLS) environments and tool chains have been developed, for example, Intel's FPGA Software Development Kit (SDK) for Open Computing Language (OpenCL) [5], and Xilinx's SDAcceel tool [6]. The demand for deploying FPGAs in the cloud is expected to rise as emerging application domains are compatible with FPGA logic, similar to GPGPUs journey in the cloud [7].

Sliding window-based image-processing algorithms are among the widely used algorithms in image processing applications such as computer vision and image segmentation [8]. An edge detector, a sliding window-based algorithm, identifies edges in digital images through a compute and data-intensive convolution process [9]. The majority of the literature focused on offloading compute-intensive components to FPGAs, leaving CPU cycles unused. In addition, most of the emerging high-performance algorithms such as big data and machine learning algorithms do not fit completely on one FPGA board. In this work, we use a hybrid CPU-FPGA system and overlap execution on both the CPU and the FPGA to increase system performance and reduce energy consumption.

We designed and implemented two algorithms for Canny edge detector, as a case study, on a heterogeneous CPU-FPGA architecture using OpenCL. In the first implementation, we partitioned the computation between the CPU and the FPGA to increase the overall system throughput, reduce latency and improve resource utilization. In the second implementation, we utilized a delay-based Weighted Round Robin (WRR) algorithm to partition and distribute images between the CPU and the FPGA. Our data-partitioning implementation outperforms the CPU-only implementation by up to 4.8×, and the FPGA-only implementation by up to 2.1×. In addition, it results in 55% reduction in energy consumption, on average, compared to the CPU-only implementation. We utilized Intel's Hardware Research Acceleration Program (HARP), also known as Heterogeneous Architecture Research Platform, cluster to implement the hybrid accelerator.

The paper is organized as follows: Section II provides background information about Intel's HARP CPU-FPGA heterogeneous platform, and Canny. Section III presents the related work. Next, we describe our two CPU-FPGA implementations of Canny in section IV. Experimental results and their discussion are presented in section V. Finally, Section VI concludes the paper and provides insights for future work.

II. Background

This section provides a background information on the underlying CPU-FPGA platform and Canny. It discusses the HARP system's architectural characteristics. In addition, it discusses the implemented algorithm.

A. Hardware Accelerator Research Program (HARP)

In 2017, Intel announced the second generation of HARP program (HARP v2), which consists of Intel Broadwell Xeon CPU (14 cores) that is integrated with Intel Arria 10 GX 1150 FPGA into a Multi-Chip Package (MCP). In this heterogeneous platform, the CPU and the FPGA share an address space in the global memory. The FPGA contains a cache to mitigate the latency of accessing the shared memory. The cache is connected only to the QPI interconnect, and it is 64 KB in size with a cache line of 64 bytes. In a read cycle of the FPGA, in case of a read miss, the FPGA will read from the CPU's last level cache (35 MB in size) [10], [11].

B. Sliding Window Based Edge Detectors: Canny Filter

Edge detection is used in many image processing applications such as image segmentation and computer vision. It is considered an efficient preprocessing stage to reduce the amount of data to be processed by filtering noise and outliers in digital images. Canny filter is a powerful edge detection algorithm [12], [13], yet a compute-intensive one. The algorithm receives a Red-Green-Blue (RGB) image as an input and produces an enhanced gray-scale edge detected image through a multi-stage image processing framework. The algorithm consists of the following steps (as shown in Figure 1): Image Blurring, Derivatives Magnitude and Orientation, Non-maximal Suppression, and Hysteresis. These are calculated over sliding windows of the image.

III. Related Work

There has been several efforts to accelerate Canny using FPGAs [12], [14]–[16]. Most of the existing implementations offloads the entire computation of Canny to an accelerator, a GPGPU or a FPGA, leaving the CPU idle waiting for the final result from the co-processor. Some works partitioned the algorithm among the CPU and the FPGA [17]. Quinne et al. [17] proposed a HW/SW co-design framework to accelerate image processing pipelines. It partitions the image pipeline components between hardware and software using exact and heuristic algorithms, where the number of components is limited to 20. This method boosts performance by offloading certain components to the FPGA. Gentsos et al. [18] designed and implemented a parallel architecture to process Canny algorithm in real time. They processed four pixels at a time instead of one pixel to maximize throughput. Their implementation demonstrated the ability to improve the total throughput with a minor increase in resource utilization using Xilinx FPGAs. He and Yuan [14] optimized the thresholding stage of the

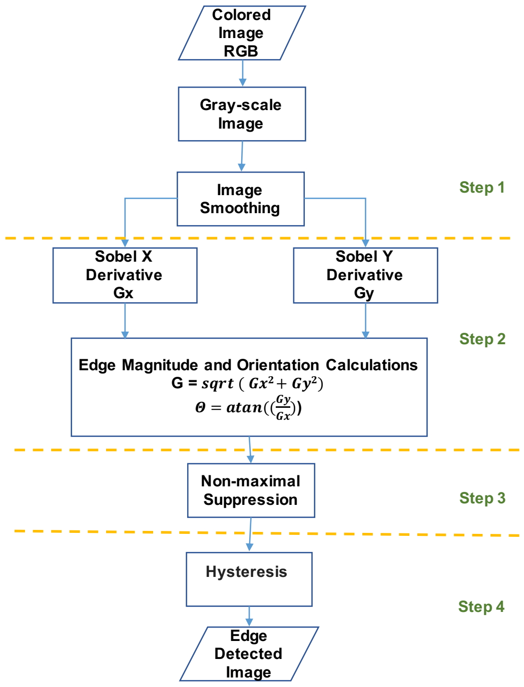

Fig. 1: Canny Algorithm Flow Chart.

Canny algorithm by automatically setting the threshold value by the algorithm itself. A pipelined implementation of the new algorithm was realized on an FPGA board. The new algorithm improved the quality of the resultant edges compared to the original algorithm. None of the previous works, to the best of our knowledge, overlapped the CPU and the accelerator execution. However, in one of our implementations, we exploit the different processing units in the system (CPU and FPGA) to accelerate the algorithm by overlapping the execution between the CPU and the FPGA. In this way, both the CPU and the FPGA take a portion of the processing/workload in order to minimize the overall execution time.

IV. Hybrid CPU-FPGA Acceleration for Canny Algorithm

We designed two different hybrid CPU-FPGA implementations that simultaneously use both a CPU and a FPGA to accelerate Canny. In the first implementation, we partitioned the algorithm to process certain parts on the CPU and other parts on the FPGA. In the second implementation, both the CPU and the FPGA execute the entire algorithm on different portions of the image (image tiles) to maximize performance.

A. Canny Code Partitioning Between the CPU and the FPGA

Code partitioning is an efficient way to increase the performance of algorithms. We partitioned the code of Canny to overlap the execution between the CPU and the FPGA, where each processes only part of the algorithm in a cooperative way to produce the output. Figure 2 shows Canny partitioned on both the CPU and the FPGA, running on a HARP node. The CPU is responsible for processing the first step of the algorithm, gray scale conversion and blurring, for the entire input image. Once the CPU processes a portion of the image

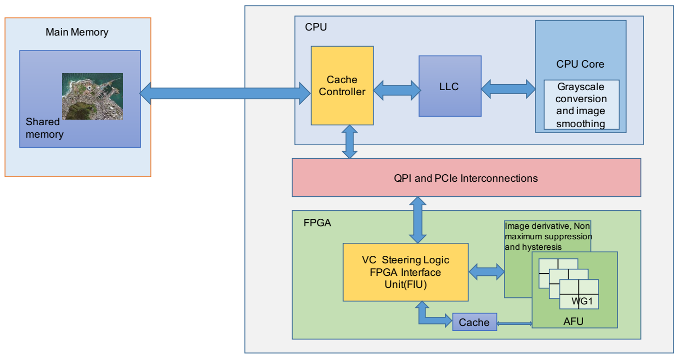
Main Memory
Shared memory
CPU
Cache Controller
LLC
CPU Core
Grayscale conversion and image smoothing
QPI and PCIe Interconnections
FPGA
VC Steering Logic FPGA Interface Unit(FIU)
Image derivative, Non maximum suppression and hysteresis
WG1
Cache
AFU

Fig. 2: CPU-FPGA Code-Partitioned Processing for Canny Edge Detection Algorithm.

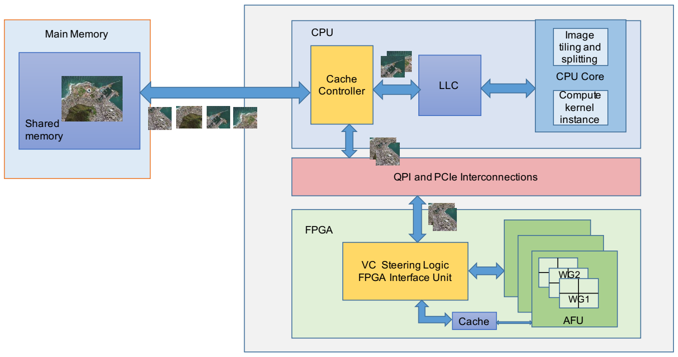

Fig. 3: CPU-FPGA Tile-Based Processing for Canny Edge Detection Algorithm.

consisting of several windows, it sends them to the FPGA to continue processing the rest of the algorithm. The CPU sends several blurred gray-scale windows of the image to the FPGA. Then the FPGA starts processing those windows and at the same time the CPU continues processing Step1 for the rest of the image. The kernel on the FPGA processes the image in a pipelined hardware for rest of the steps of Canny, implemented as one kernel.

We used OpenCL to implement Canny, where each work item in OpenCL is processed through one pipeline. This process repeats until the CPU finishes the first step of the algorithm for the entire image, and the FPGA finishes the other steps for the entire image and stores the final image into the shared DRAM. The CPU's and the FPGA's calculations overlap; however, the FPGA is at first idle until the CPU finishes processing Step 1 for some windows. Although, we partitioned the code between the two processing units to achieve higher performance and increase simultaneous processing, the CPU and the FPGA stay idle for some portions of the time. This opens another opportunity to optimize the hybrid algorithm by increasing the overlap of the execution between both processing units, as discussed below.

B. Delay-based Weighted Round Robin Distribution of the Workload Among the CPU and the FPGA

In the second hybrid CPU-FPGA implementation of Canny, we aimed at maximizing the CPU-FPGA processing overlap by allowing both to execute the same kernel on different parts of the image simultaneously. We first split the image into tiles. The tiles then were distributed to the CPU and the FPGA using a Weighted Round Robin (WRR) algorithm. Tiling is efficient in hyper scale data centers, where billions of images need to be processed in a real-time fashion; it allows pipelined image processing algorithms to be more efficient and meet timing constraints.

After tiling the source image, tiles are distributed to available system processing units (CPU and FPGA) in such a way that minimizes execution time and maximizes system throughput. Weights are assigned to both the CPU and the FPGA using a CPU-to-FPGA ratio of the execution time of a single tile. The tiles are then distributed to the CPU and the FPGA proportionally to their execution time of a single tile. This delay-aware split of tiles between the CPU and the FPGA boosts the parallelism of the system through simultaneous handling of different data sets by the different devices. We used the equation below to assign weights to the CPU and the FPGA:

$$W_{CPU} = tile\ exec.\ time_{CPU} / tile\ exec.\ time_{FPGA} \quad (1)$$

Since the FPGA is found to be approximately two times faster than the CPU for Canny, the CPU is assigned double the weight of that of the FPGA. This method can be used for many other heterogeneous architectures and algorithms to balance the load of the different architectures and maximize performance. Furthermore, fine-grained tiles can be used to achieve higher processing parallelism and thus performance. Fig. 3 shows a block diagram of the CPU-FPGA system with an illustration of Canny's data-partitioned implementation.

V. Experiments, Results and Discussion

In the experiments, we used one HARP v2 node to accelerate Canny and different image sizes as inputs to Canny algorithm (from 0.5 megapixel to 8 megapixels). Table I shows system specifications for a HARP node. Our implementation of Canny uses a sliding window of 3*3 in size. We implemented two different hybrid CPU-FPGA algorithms, a code partitioned implementation and a data partitioned implementation using OpenCL. The OpenCL kernel is an ND-Range one, and the size of each WorkGroup is 128 work items. In the data partitioned implementation, we partitioned the input images into tiles to be processed by the CPU and the FPGA simultaneously. Each tile represents a portion of the image; the tiles were padded with neighboring pixels from the source image for correct functionality.

TABLE I: System Specifications

CPU configurations	
Host CPU Model	Intel Broadwell Xeon
CPU Frequency	2.1 GHz
LLC	35 MB
FPGA Fabric	Arria 10 GX 1150
DRAM	95 GB
FPGA configurations	
Adaptive logic modules	427,200
Logic elements	1,150,000
Registers	1,708,800
Memory	65.5 kib
DSP	1,518

A. Code Partitioning

As mentioned in Section IV-A, we partitioned Canny algorithm between the CPU and the FPGA. This approach reduces the total execution time of the algorithm compared to a CPU-only implementation. The energy consumption of this implementation is also more efficient than a CPU-only implementation. This is because the compute-intensive portion of the code is offloaded on a power-efficient processing unit, the FPGA. Code partitioning accelerated the algorithm about two fold as shown in Figure 5 compared to a CPU-only implementation. On the other hand, code partitioning accelerates the algorithm up to 1.5× for images smaller than four megapixels. However, for larger images, four megapixels and bigger, the FPGA-only implementation slightly outperforms the code-partitioned implementation. The reason behind this is the communication overhead when passing the partially processed image from the CPU to the FPGA to be completely processed and then the final output image is written back to the shared memory.

B. Data Partitioning

In the code-partitioned approach, although the CPU and the FPGA run parts of the algorithm at the same time, they can be idle for certain times. The CPU will be idle after it finishes processing Step 1 of the algorithm for the entire image. Additionally, the FPGA will be waiting for the CPU to finish Step1 for several windows to start processing the other steps for several windows simultaneously, due to the parallel nature of the FPGA. As such, an implementation that totally overlaps CPU and FPGA processing is desired to further increase performance. In the second approach, we partitioned the input images into tiles as discussed previously in section IV-B. Tiles are mapped to the CPU or the FPGA dynamically using the WRR algorithm. Each tile is assigned to either the CPU or the FPGA, then processed and the output of the edge-detected image is written back to the shared memory.

Figure 4 depicts the effect of different tile sizes on the total execution time. As shown in the figure, a tile size of 250 kilo-pixels produces the minimum execution time for all the tested images. Smaller tile sizes increase the total execution time for all image sizes. This is due to the overhead of the padding pixels that wrap the image tiles after tiling. For instance, tiling a four megapixel image, using a tile size of 50 kilo-pixels results in a five fold padding pixels compared to a tile size of 250 kilo-pixels. These additional pixels mean extra reading, processing, and writing time. On the other hand, increasing the tile size has also a negative impact on the total execution time. This is because of the performance difference between the CPU and the FPGA, so coarse-grained tiles would result in a reduced overlap of both the CPU and the FPGA processing times, and thus reduced performance. As such, we chose a tile size of 250 kilo-pixels for our experiments. The overhead of the padding pixels is calculated as follows:

$$\begin{aligned} padding = 2*(No.\_of\_tiles)*(tile\_width+2) \\ +2*(No.\_of\_tiles)*(tile\_height+2) \end{aligned} \quad (2)$$

Where $padding$ is the number of the additional padding pixels.

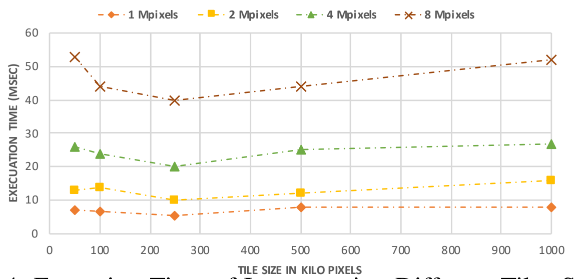

Fig. 4: Execution Time of Images using Different Tiles Sizes.

Figure 5 shows the performance achieved using our tile-based hybrid implementation over the CPU and the FPGA implementations for different image sizes. For example, using a two-megapixel image, the speedup gained by the hybrid data-partitioned implementation is 4.8× over a CPU-only and 2.1× over a FPGA-only implementations. For a one-megapixel image, the hybrid implementation results in 2.2× speedup over the CPU and no noticeable speedup over the FPGA-only implementation. This is because as we tend to process small images, the CPU becomes a bottleneck and its execution time can become dominant over the FPGA's execution time. The FPGA consumes its data while the CPU is still processing its part. The CPU bottleneck can be solved by assigning the FPGA a higher weight leaving the CPU with only a small portion of the frame.

We also estimated the energy consumption of Canny algorithm on the different architectures. The CPU's energy consumption was estimated based on Intel's power documentation for Xeon processors, and the FPGA's energy consumption was estimated using Power-Play, which is a power estimator for Intel Arria 10 devices. The hybrid data-partitioned CPU-FPGA implementation reduces energy consumption up to approximately 73% and on average 55% for the different image sizes compared to the CPU only implementation. This reduces the total energy consumption of such heterogeneous CPU-FPGA servers and also reduces cooling requirements. We also calculated the Energy Delay Product (EDP) for the different implementations, as in high-performance and cloud computing the execution time is of significance although lower energy consumption is desired. Figure 6 shows the EDP for the CPU-only, FPGA-only and the hybrid CPU-FPGA implementations. The figure shows that the EDP of the hybrid implementation of the algorithm is close from that of the FPGA-only implementation and much lower than the CPU-only implementation. It also shows that for image sizes below 4 mega pixels, the EDP values for the FPGA-only and the hybrid implementations are similar.

In order to reduce the hybrid implementation's execution time, further optimizations for the hardware accelerator were implemented. By default, each OpenCL kernel is implemented in hardware as a single compute unit. Adding multiple compute units can increase performance, where each compute unit has its separate memory interfaces. All compute units are capable of processing multiple workgroups simultaneously. We varied the number of compute units from one to ten. A reduction in execution time is observed until it reaches a

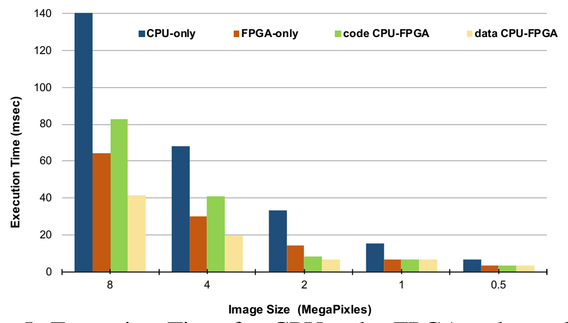

Fig. 5: Execution Time for CPU-only, FPGA-only, and CPU-FPGA Hybrid Implementations.

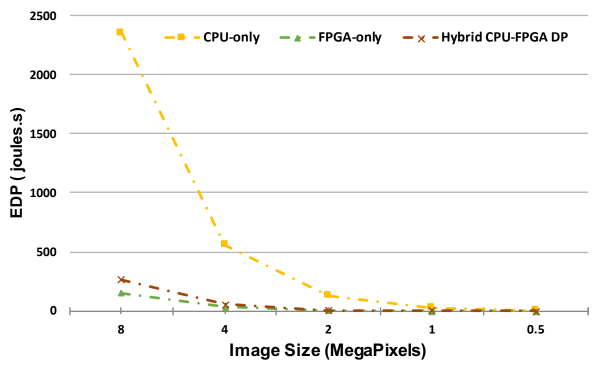

Fig. 6: Energy Delay Product

certain limit (3 compute units). This is due to a contention among the compute units while accessing the FPGA's cache. Moreover, increasing the number of compute units increases kernel resource utilization of the FPGA, in addition to increasing power consumption. Hence, the number of compute units is subject to power and area constraints of the design.

Another optimization technique that we studied is bundling work items within the same workgroup into Single Instruction Multiple Data (SIMD) lanes. A workgroup with multiple SIMD lanes results in a slightly lower execution time compared to a work-group without SIMD. This is because using multiple SIMD lanes increases the amount of parallel computations within the same work group. Moreover, SIMD behavior allows the hardware compiler to coalesce memory accesses. Although both multiple compute units and multiple SIMD lanes increase a kernel's performance, using multiple SIMD lanes does not consume additional FPGA resources compared to increasing the number of compute units.

VI. Conclusion and Future Work

In this paper, we designed and implemented two different approaches for accelerating a sliding window-based image processing algorithm, Canny, using OpenCL. In the first approach, the code was partitioned among the CPU and the FPGA, and resulted in up to 2.3× speedup compared to a CPU-only implementation. In the second approach, both the CPU and the FPGA executed the entire algorithm for different portions of the image. Our implementation outperforms both CPU-only and FPGA-only implementations by up to 4.8× and 2.1× respectively. It also results in 55.3% reduction in energy consumption, on average, compared to the CPU-only implementation. In addition, its energy-delay product is comparable to the FPGA-only implementation. Our results showed that collaborative execution between the CPU and the FPGA in heterogeneous computing environments significantly impact the execution time.

Acknowledgement

This work was funded by the National Science Foundation Grant No. 1821691, and Intel Inc. equipment access grant to HARP v2.

References

[1] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed systems*, vol. 22, no. 6, pp. 931–945, Jun. 2011.

[2] Y. Pu, J. Peng, L. Huang, and J. Chen, "An efficient KNN algorithm implemented on fpga based heterogeneous computing system using opencl," in *Field-Programmable Custom Computing Machines (FCCM), IEEE 23rd Annual International Symposium on*, May 2015, pp. 167–170.

[3] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2015, pp. 111–118.

[4] P. Gupta, "Accelerating datacenter workloads," in *26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2016.

[5] "OpenCL - The open standard for parallel programming of heterogeneous systems," Jul. 2013. [Online]. Available: https://www.khronos.org/opencl/

[6] "SDAccel Development Environment." [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[7] L. Hu, X. Che, and Z. Xie, "Gpgpu cloud: A paradigm for general purpose computing," *Tsinghua Science and Technology*, vol. 18, no. 1, pp. 22–23, Feb. 2013.

[8] L. Shao, R. Yan, X. Li, and Y. Liu, "From heuristic optimization to dictionary learning: A review and comprehensive comparison of image denoising algorithms," *IEEE Transactions on Cybernetics*, vol. 44, no. 7, pp. 1001–1013, Jul. 2014.

[9] W. Cao, Y. Zhou, C. P. Chen, and L. Xia, "Medical image encryption using edge maps," *Signal Processing*, vol. 132, pp. 96–109, Mar. 2017.

[10] "Intel Acceleration Stack for Intel Xeon CPU with FPGAs 1.0 Errata." [Online]. Available: https://www.intel.com/content/www/us/en/programmab- le/ documentation/ffv1519794536166.html

[11] T. Faict, "Exploring OpenCL on a CPU-FPGA Heterogeneous Architecture Research Platform (HARP)," p. 104.

[12] Q. Xu, S. Varadarajan, C. Chakrabarti, and L. J. Karam, "A distributed canny edge detector: algorithm and fpga implementation," *IEEE Transactions on Image Processing*, vol. 23, no. 7, pp. 2944–2960, Jul. 2014.

[13] G. N. Chaple, R. Daruwala, and M. S. Gofane, "Comparisions of robert, prewitt, sobel operator based edge detection methods for real time uses on fpga," in *Technologies for Sustainable Development (ICTSD), International Conference on*, 2015, pp. 1–4.

[14] W. He and K. Yuan, "An improved canny edge detector and its realization on fpga," in *Intelligent Control and Automation, WCICA 7th World Congress on*, Jun. 2008, pp. 6561–6564.

[15] X. Li, J. Jiang, and Q. Fan, "An improved real-time hardware architecture for canny edge detection based on fpga," in *Third International Conference on Intelligent Control and Information Processing*, 2012, pp. 445–449.

[16] J. Lee, H. Tang, and J. Park, "Energy efficient canny edge detector for advanced mobile vision applications," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 28, no. 4, pp. 1037–1046, 2016.

[17] H. Quinn, L. A. S. King, M. Leeser, and W. Meleis, "Runtime assignment of reconfigurable hardware components for image processing pipelines," in *Field-Programmable Custom Computing Machines, FCCM 11th Annual IEEE Symposium on*, Apr. 2003, pp. 173–182.

[18] C. Gentsos, C.-L. Sotiropoulou, S. Nikolaidis, and N. Vassiliadis, "Real-time canny edge detection parallel implementation for fpgas," in *Electronics, Circuits, and Systems (ICECS), 17th IEEE International Conference on*, Dec. 2010, pp. 499–502.