SACJ 32(1) July 2020
Research Article

An OpenCL-based parallel acceleration of a Sobel edge detection algorithm Using Intel FPGA technology

Abedalmuhdi Almomany[a], Ahmad M. Al-Omari[b], Amin Jarrah[a], Mohammed Tawalbeh[c], Amin Alqudah[a]

[a] Department of Computer Engineering, Yarmouk University, Irbid, Jordan
[b] Department of Biomedical systems and Bioinformatics Engineering, Yarmouk University, Irbid, Jordan
[c] Information Technology & Communications Center, Jordan University of Science and Technology, Irbid, Jordan

ABSTRACT
This paper examines the feasibility of using commercial out-of-the-box reconfigurable field programmable gate array (FPGA) technology and the open computing language (OpenCL) framework to create an efficient Sobel edge-detection implementation, which is considered a fundamental aspect of image and video processing. This implementation enhances speedup and energy consumption attributes when compared to general single-core processors. We created the proposed approach at a high level of abstraction and executed it on a high commodity Intel FPGA platform (an Intel De5-net device was used). This approach was designed in a manner that allows the high-level compiler/synthesis tool to manipulate a task-parallelism model. The most promising FPGA and conventional implementations were compared to their single-core CPU software equivalents. For these comparisons, local-memory, pipelining, loop unrolling, vectorization, internal channel mechanisms, and memory coalescing were manipulated to provide a much more effective hardware design. The run-time and power consumption attributes were estimated for each implementation, resulting in up to 37-fold improvement of the execution/transfer time and up to a 53-fold improvement in energy consumption when compared to a specific single-core CPU-based implementation.

Keywords: FPGA, reconfigurable computing, parallel processing, edge detection, OpenCL, image processing, integrated circuits

Categories: • **Hardware** ∼ **Reconfigurable logic applications**

Email:
Abedalmuhdi Almomany emomani@yu.edu.jo (CORRESPONDING),
Ahmad M. Al-Omari aomari@yu.edu.jo,
Amin Jarrah amin.jarrah@yu.edu.jo,
Mohammed Tawalbeh mt@just.edu.jo,
Amin Alqudah amin.alqudah@yu.edu.jo

Article history:
Received: 25 Sep 2019
Accepted: 11 Mar 2020
Available online: 20 Jul 2020

Almomany, A., Al-Omari, A.M., Jarrah, A., Tawalbeh, M., and Alqudah, A. (2020). An OpenCL-based parallel acceleration of a Sobel edge detection algorithm Using Intel FPGA technology. *South African Computer Journal 32*(1), 3–26. https://doi.org/10.18489/sacj.v32i1.749

SACJ is a publication of the South African Institute of Computer Scientists and Information Technologists. ISSN 1015-7999 (print) ISSN 2313-7835 (online).

1 INTRODUCTION

Reconfigurable computing devices, such as FPGAs have become widely utilized in many applications, including image processing, security, finance, networking, machine learning, pattern recognition, and scientific computing [1–3]. The use of custom dedicated hardware, such as the application specific integrated circuit (ASIC), leads to better performance when compared to general-purpose processors. Nonetheless, the ASIC's architecture and functionality cannot be changed. Reconfigurable computing devices, such as FPGAs, are used to achieve a level of performance comparable to that achieved using dedicated hardware devices (Tessier et al., 2015). FPGA technology reconstructs fine-grain control logic and the data path characteristics of the underlying hardware before and during the run-time. These processes yield better matching between the temporal needs and underlying algorithmic structure of the application under consideration. FPGA technology also promotes software flexibility. FPGAs can be reconfigured many times with a massive number of possible configurations, thus simplifying the processes of optimizing or modifying the existing design.

FPGAs comprise an immense number of small building blocks that are attached by on-chip finely-grained hierarchical switching and routing fabric. These building blocks (in Intel devices) usually incorporate adaptive logic modules (ALMs), SRAM memory, extensive computation blocks or digital signal processing (DSP) blocks, and streaming I/O ports. FPGAs can also contain other kinds of blocks, such as phase-lock loops (PLLs), which can be utilized to adjust the internal clock frequency. The fabricated design on the FPGA platform does not have fetch and decode instruction steps overhead associated with regular memory instruction-set for a general processor; this is because the data path and control circuitry are optimized according to the proposed design.

Moreover, the ALMs, made of at least one lookup table (LUT) each of which is composed of one or more flip-flop (FF), are spread throughout the FPGA fabric, making the FPGAs very amenable to temporally parallel (systolic or pipelined) computation that can be employed to monopolize the loop-level concurrency that exists in diverse applications. In such cases, the body of the loop is split into executable pieces where each piece is targeted for execution on a different stage of computational logic created within the FPGA. Data passed among pipelined stages are stored in discrete and accessible ALM flip-flop resources. Commonly, when it is fully pipelined, the time needed to pass an item of data from one stage to another in a mere temporal pipeline is one clock cycle. All stages are concurrently performing their computations but with various data. In such cases, the number of clock cycles to treat any single item, usually called pipeline latency, would match the number of stages in the system to treat the body of the loop. However, if the number of elements in the loop is vast, then the most influential metric is the initiation interval (II), which is the average number of clock cycles that the system should wait before the next item is allowed to enter the pipeline.

A custom-created pipeline within an FPGA reveals the low-level structure of an application effectively. In addition to reducing latency, FPGAs are also widely used to reduce the overall energy consumption in many applications according to existing studies [5–8]. FPGAs generally

use less energy when compared to other platforms such as CPUs and graphical processor units (GPUs), because they carry fine-grain distribution of computation across the integrated circuit (IC) in ways that improve data locality and computational efficiency while depreciating data and instruction storage.

The Sobel edge detection algorithm (see Section 2) is implemented on an FPGA platform using a De5-net acceleration board from Terasic. This board has adequate resources that can be utilized effectively to synthesize the user's code in various complex applications; it consists of 234,770 ALMs, 2,560 RAM blocks, and 256 DSP blocks. A high-speed interface connection or a peripheral component interconnect express (PCIe) connects the target board and the host CPU, providing the possibility of transferring data very quickly between the computation units. The initial conventional approach to program these FPGAs is to use hardware description languages (HDLs) such as very high speed integrated circuit (VHSIC) hardware description language (VHDL) and Verilog. However, using these languages introduces challenges to designers to be knowledgeable about the underlying hardware, such as being aware of the developing control states, hardware circuits, and handling timing issues. All these difficulties make adopting these languages less preferable, particularly when the design becomes more complicated. The OpenCL (Xu, 2011) platform was introduced as a simple C-language extension to overcome all of these issues and simplify the process of programming FPGAs through abstracting most of the hardware details. OpenCL may also lead to reducing the kernel design time significantly (Hill et al., 2015a). Generally, OpenCL is a software development tool that supports heterogeneous computing in which different kinds of computations units exist. The power of OpenCL allows for distributing tasks among multiple platforms such as CPUs, FPGAs, or GPUs. The OpenCL platform is based on having one host (CPU) and one or more devices that could be one or more computation platforms, as shown in Figure 1.

The OpenCL programming model incorporates two programs. The first one is the host program that runs on the host machine, usually written in C/C++, and includes responsibilities such as loading the OpenCL programs, memory management, data movement, and error handling. The second program is the device code, an OpenCL-based program that can be run on the available devices. The Intel software development kit (SDK) for OpenCL provides the ability to implement parallel algorithms on the target device with minimal effort. The device code compilation process is usually lengthy; it can last up to several days according to the complexity of the user's code and the number of resources used in the synthesizing process. Consequently, the device code should be compiled first to generate the final executable design used within the host code, known as the offline programming model. However, the Intel SDK tools provide an environment where the host code can be emulated on a similar FPGA platform to verify results before beginning a time-consuming compilation process. The Intel Compiler creates pipeline architecture according to the device code and aims to execute complex instructions in one clock cycle. Figure 2 describes the compilation process flow.

https://doi.org/10.18489/sacj.v32i1.749

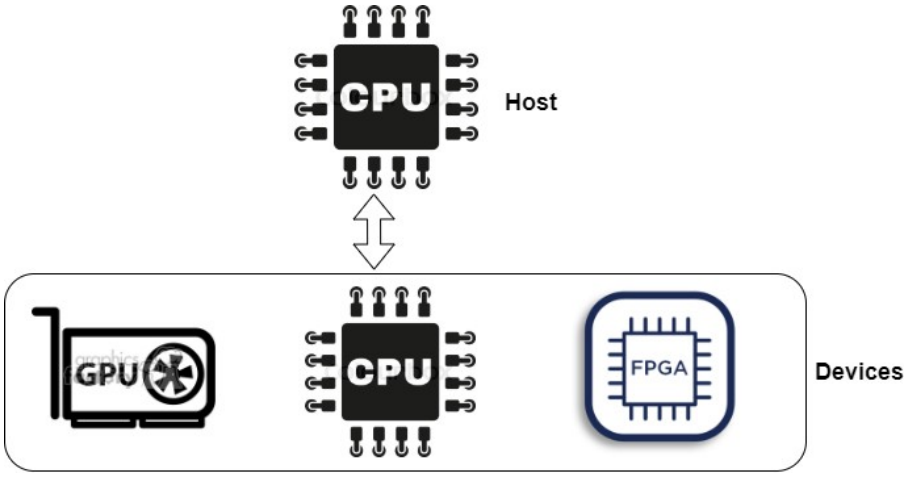

Figure 1: OpenCL programming model–heterogenous computing environment. The CPU (host) can be connected to one or more of the available computation platforms (devices).

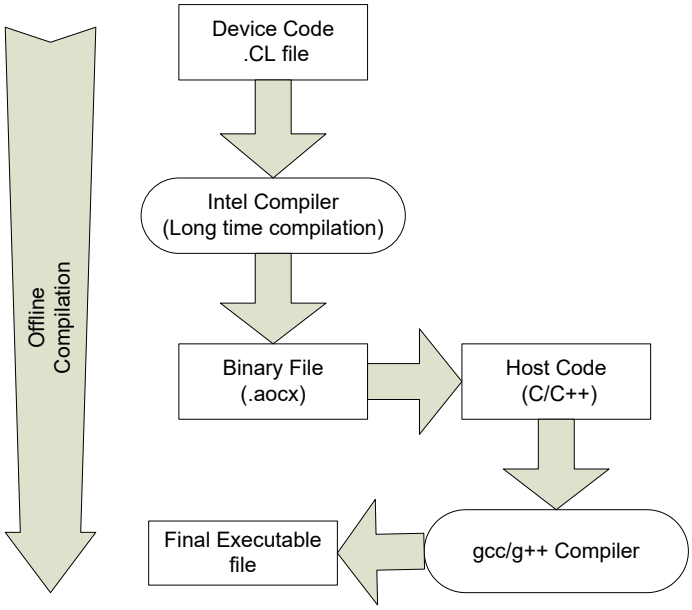

Figure 2: Compilation process flow

https://doi.org/10.18489/sacj.v32i1.749

Several studies have discussed the optimization process of the Sobel edge detection algorithm on FPGAs, GPUs, and multi-core systems (Abbasi & Abbasi, 2007; Chouchene et al., 2014; Dore, 2014; Halder et al., 2012; Hill et al., 2015b; Nausheen et al., 2018; Vanishree & Reddy, 2013; Yasri et al., 2008; You et al., 2017). One study used a GPU (NVIDIA GeForce 310) to get a significant speedup where a data-parallelism model utilizes a large number of available cores. This study also used an FPGA (Xilinx Virtex-5 device) platform to speed up the process while implementing the synthesis code in the hardware description language (VHDL). The results demonstrate the effectiveness of using the FPGA platform to accelerate this application compared to the CPU platform (Chouchene et al., 2014). Nausheen (Nausheen et al., 2018) suggested a new modified hardware implementation of the Sobel edge detection that reduces the number of resources and space complexity. The algorithm was tested on the Xilinx Sparta 6 FPGA device to achieve a double clock frequency rate compared to the old design with approximately two nanoseconds to process each pixel (Halder et al., 2012).

The distinguishing feature of this described work is using the OpenCL abstract language to optimize the edge operator on the Intel FPGA De5-net device, thus reducing the design complexity, compilation process time, and code portability across different devices/platforms (Hill et al., 2015a) while achieving significant execution time improvement. Compared to a similar study utilizing the OpenCL to implement the Sobel operator on a DE1-SoC Intel device (You et al., 2017), here in this paper there is ten times the performance improvement for the large image size (1920 x 1080). Moreover, there is more than 26-times performance improvement compared to using NVIDIA GTX 470 GPU (Dore, 2014). This study's approach is to create multiple tasks (threads) and implement them in hardware with very efficient pipeline structures that communicate using high-speed internal buffers. Such an approach further illustrates the effectiveness of using FPGAs to reduce energy consumption.

The parallelism in this paper was performed by dividing the main task into four sub-tasks (threads) using the task-parallel model. Thus, to reduce the significant time required to access the data from global memory, only one task will access these data and send it to other threads using high-speed internal buffers (channels) in a very short time. This allows all threads to work together without waiting until one thread finishes its job and passes the whole data to the next thread. Vectorization data types, such as int8, float8, and int16, are used to increase the amount of work per clock cycle, so that eight or sixteen operations can be done in one clock cycle. The optimization report generated by the Intel Compiler is also exploited to ensure that all loops are pipelined successfully with almost one initiation interval (II). Results show that the FPGA device can process five pixels every clock cycle compared to 200 clock cycles and 81 clock cycles needed to process each pixel in the conventional and optimized CPU implementations.

2 SOBEL EDGE DETECTION OPERATOR

With edge detection, it is possible to reduce the image size significantly while keeping the most useful information (Asghari & Jalali, 2015), making it a popular technique used to study

images and extract essential features. Based on a gradient approach, a Sobel operator is constructed to calculate the gradient of the target image intensity by having high spatial frequency values (changes in pixels value or the digital number between adjacent pixels) that distinguish the edge-regions from non-edge-regions (Liao et al., 2010). Horizontal and vertical masks (3x3 in dimension) are used to calculate the first derivative along the x- and y-axes. Equation 1 describes the two masks. The next step is to find the approximate gradient amplitude gx,y at each pixel in the 2D image array using the mathematical expression in Equation 2 (Deng et al., 2011). Finally, the gradient orientation is calculated for each pixel value according to the expression in Equation 3. The gradient amplitude is compared to a predetermined threshold value to determine if there is an edge. Various applications and studies widely utilize edge detection, including driver safety (Liao et al., 2010), license plate detection (Israni & Jain, 2016), facial identification analysis, face detection (Singh et al., 2016), email protection (Wan & Uehara, 2012), transmission lines monitoring (Zhai et al., 2017), and medical and health fields (Yusoff et al., 2018).

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{1}$$

$$gx, y = |G_x| + |G_y| \tag{2}$$

$$\Theta = tan^{-1} \frac{|G_y|}{|G_x|} \tag{3}$$

The hardware implementation of the Sobel operator is a relatively complicated task compared to software implementations; several studies suggested architectures to implement the edge operator (Abbasi & Abbasi, 2007; Vanishree & Reddy, 2013; Yasri et al., 2008). Memory blocks store the monochromatic image, and these blocks are accessed by a hardware circuit performed on the FPGA. The convolution kernels can be implemented using a set of buffers and shift registers, as can the implementation of convolution equations using a set of flip-flops, adders and comparators units. The finite state machine (FSM) controller can also be created to control the overall process of the edge detection in which in each state, a set of one or more operations, can be performed before moving to the next state. Usually, the overall design can be represented using hardware description languages such as VHDL or Verilog.

3 RELATED WORK

A Sobel edge detection algorithm implementation requires an extensive amount of computations (millions of operations) and, as mentioned, is used widely in diverse applications. All these reasons encourage researchers to develop improved implementations to utilize the most recent machines that have heterogeneous computing environments such as multi-core, multi-processor, GPUs, and FPGAs. Acceleration devices have promising features if considering the

FPGA platform, which could be utilized in real-time applications. For example, an Intel Cyclone IV significantly speedup the process of Hevea leaves disease identification compared to using software solutions implemented in MATLAB and ran on a general processing computation element (CPU platform) (Yusoff et al., 2018). FPGAs can be used to implement real-time edge detection algorithms as they carry high levels of parallelization structures (G. N. Chaple et al., 2015; G. Chaple & Daruwala, 2014). A similar study that utilized an Intel EP4CE30 FPGA device to optimize an 8-directions Sobel algorithm demonstrated the feasibility of using this platform in these kinds of problems to improve the overall performance (Xiangxi et al., 2018). Several studies (Tian et al., 2015; Yasri et al., 2009) confirmed the high-performance implementation with a high degree of accuracy through employing FPGA devices, while other studies focused on utilizing GPUs to achieve a high-performance factor by significantly reducing the execution time (Al-Omari et al., 2015; Fredj et al., 2017). Most of these designs employed the compute unified device architecture (CUDA) as a primary tool to work within a GPU environment. A more than 900 times speed improvement is achieved when compared to a general single-core processing element. However, GPUs consume more power compared to other computation platforms (Firmansyah et al., 2018; Ghosh & Chapman, 2011). The work described in this study optimizes the Sobel implementation to reduce the execution time and power consumption significantly.

4 SOBEL IMPLEMENTATION ON FPGA TARGET PLATFORM

4.1 Task-Parallel model (single work item)

The device code consists of one or more functions (known as kernels) that should be run on the objective accelerated device. Generally, these kernels can be manipulated and executed as a single thread model (also known as a single work item) or as a multiple to a vast number of threads model (also known as an NDRange model). GPUs usually use the second model as it holds a large number of processing elements (PEs); the data is shared among these PEs, each of which executes the same instruction while accessing a different data item (single instruction, multiple data (SIMD)) [27–29]. FPGAs can be utilized similarly; however, in most cases, the single work item is the preferred model. FPGAs have a different architecture that can be adapted to create an effective pipeline structure where data can be shared among multiple pipelined loop iterations using a high-speed access private memory (Waidyasooriya et al., 2018). This is a favored model because its data dependencies slow the use of the multiple threads model, particularly when costly mechanisms, such as a barrier, are used to preserve dependencies between active threads. Multiple single work items, each of which works on a separate task, formulate what is known as a task-parallel model. Commonly, these work items are executed simultaneously, and different work items access different data.

Below is a summary of the differences between the task-parallel model and the data-parallel model (NDRange model) (Waidyasooriya et al., 2018):

- In the task-parallel model, only one thread runs throughout the task-execution versus a

large number of threads (thousands of threads) in the data-parallel model.

- Data is shared between loop-iterations (using private memory) in the task-parallel model, whereas in the data-parallel model, the data sharing is between threads (using local (shared) memory).
- In the task-parallel model, loop iterations are pipelined, whereas thread executions are pipelined in the data-parallel model.

4.2 Sobel optimisation

The input image should have a monochromatic format, where the image sizes vary from the smallest image size (144 x 256 pixels) to the largest one (3480 x 5760 pixels). The process is broken into four steps to perform edge detection. In the first two steps, the convolution masks Gx and Gy are applied to every pixel value along the x- and y-axes. In the last two steps, the gradient magnitude (an approximate magnitude as in (2)), and direction are estimated, and the magnitude is compared to a predetermined threshold value to decide whether or not there is an edge pixel.

The optimization process begins by choosing the appropriate parallel model for the accelerated device, namely, a task-parallel model. Many experiments have been performed to select among the best possible combinations of optimization procedures, including dividing the device code into four separate kernels, each of which is executed using a single thread, as shown in Figure 3. These kernels are Convx, Convy, Magn (x, y), and Dir(x, y). The idea of the task parallel model is to let multiple threads work on different tasks simultaneously. The first two kernels are completely independent, and they can run simultaneously. However, the calculations in the last two kernels depend on the results from the first two kernels. In the conventional implementation (conventional implementation means that it is similar to the original C program without modifying the code to use any possible optimization techniques), the output results from the first two kernels should be written back to the global memory before calling the last two kernels. The first adopted optimization technique uses high-speed channels (internal local buffers implemented using RAM blocks and registers) to avoid this lengthy operation as well as avoid copying results back to the global memory and rereading them as inputs to the last two kernels. In the first two kernels, after calculating the Gx(x,y) and Gy(x,y) for a pixel located at (x, y), the result will be forwarded to both Magn(x, y) and Dir(x, y) kernels using dedicated channels so that all four kernels can operate together simultaneously as shown in Figure 4. This process should be performed for all pixels, where multiple pixels are read in every iteration to magnify memory bandwidth utilization.

https://doi.org/10.18489/sacj.v32i1.749

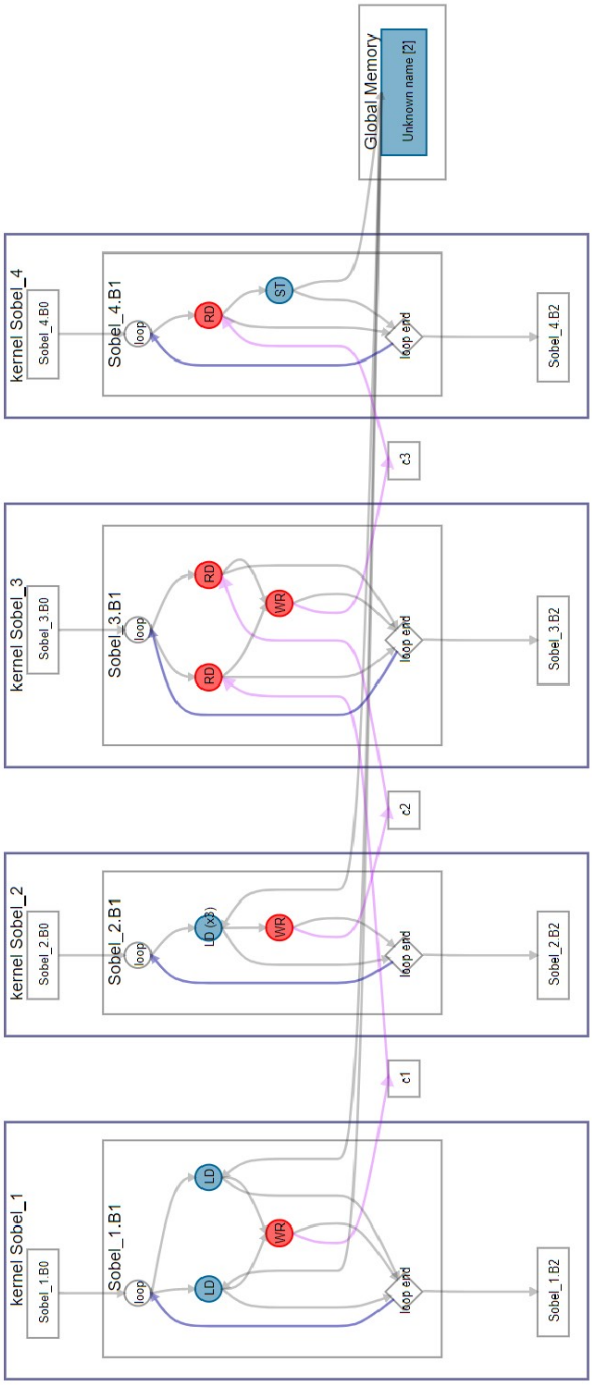

Figure 3: System viewer generated by the Intel compiler. All tasks are running simultaneously, where local memory and channels are employed to reduce the long global memory access time.

https://doi.org/10.18489/sacj.v32i1.749

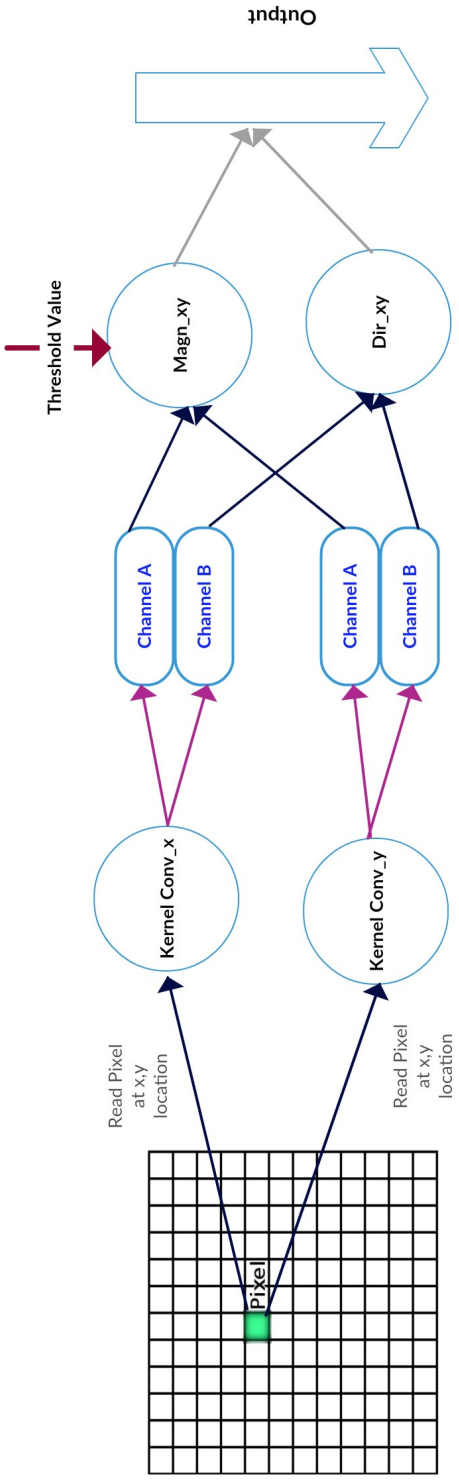

Figure 4: Task-parallel model where four tasks, Convx(x, y), Convy(x, y), Magn(x, y), and Dir(x, y), run simultaneously. Data exchange between kernels occurs using dedicated channels to avoid extra read/write from/to global memory requests.

https://doi.org/10.18489/sacj.v32i1.749

For a single work item, techniques such as loop-unrolling (manual loop-unrolling is used here to increase the amount of work performed in each clock cycle; the loop unrolling pragma cannot be used as it contradicts the use of a channel mechanism), vectorization, coalesce memory access and utilizing fast local memory are conventional techniques used to improve the performance and increase the level of parallelism. All these methods enhance memory bandwidth and coalesce memory access, as well as increase the amount of work done in every clock cycle in the created pipeline structure. The files generated by the Intel Compiler can be examined to create an efficient pipeline structure by solving the dependencies of executed loops and data, decreasing the initiation interval to the optimal value or the smallest possible number, and coalescing the memory access. It is essential to mention that there are difficulties faced with the increase in performance beyond a certain level, including maximum memory bandwidth and the required time to move data from/to accelerated device.

The serial code is written in a conventional C++ language and targets a CPU-based platform; the code was then compiled using a g++ compiler, with and without an O3 compiler optimization argument. O3 is a level of compiler optimization that lets the compiler improve the code performance through applying a set of optimization techniques such as instruction reordering and scheduling, loop unrolling, prefetching, pipelining and other general optimizations; however, this usually increases the compilation time (Medeiros et al., 2018). The target machine has an Intel Xeon E5 @ 2.9 GHz processor, with 48 GB of RAM installed. Similar code was compiled using the Intel FPGA compiler to run on the Terasic De5-net acceleration card. However, this code has been altered to create an efficient pipeline structure, and the main task is divided into subtasks to exploit the concept of the task-parallel model. This study uses various grayscale-image sizes, each of which is a 2D array of pixels. The sizes are 144 x 256, 240 x 426, 360 x 480, 480 x 640, 720 x 1280, 1080 x 1920, 1440 x 2560, 2160 x 3860, and 3480 x 5760.

This study considers four cases:

1. serial implementation—the conventional code compiled with just the default compiler argument settings, without passing any optimization arguments

2. optimized serial implementation—the same as the first case, but compiled with optimization arguments turned on using O3

3. conventional FPGA implementation—similar to serial implementation but run on a FPGA platform

4. optimized FPGA implementation—the design is altered to utilize the architecture benefits, and a combination of optimization mechanisms are used to improve the overall performance.

https://doi.org/10.18489/sacj.v32i1.749

5 RESULTS DISCUSSION

The Terasic De5-net FPGA accelerator device implements the Sobel operator using the OpenCL framework; the Intel Compiler is then used to compile and synthesize the proposed code to the equivalent hardware circuit. Table 1 shows the FPGA resource utilization; these resources are adaptive look-up tables (ALUTs), flip-flops, memory RAM blocks, and DSP blocks. The table shows the amount and percentage of the utilized resources compared to the total available resources. Techniques used to optimize the design lead to an increase in the amount of resource usage; the exception is the number of RAM blocks used, where reducing the interaction of global memory leads to reduced RAM usage. Normally, more resources can be used to increase performance; however, this may lead to increased design complexity and critical path delay (latency) (Zheng et al., 2014). It is also not possible to expand the resource usage beyond certain percentages as this introduces the challenge of providing routing channels between connected elements/blocks (Asghar & Parvez, 2015; Vanderbauwhede & Benkrid, 2013).

Table 1: Terasic De5-net FPGA resource utilisation

	Used qty		Percentage usage	
Resource	Optimised	Conventional	Optimised	Conventional
ALUTs	148496	78633	32	17
FFs	143776	101043	15	11
RAMs	740	2265	29	88
DSPs	144	15	56	6

It is worth mentioning that there is an overhead of using OpenCL to program the FPGAs over using hardware description languages such as VHDL in terms of resource usage. Using VHDL or Verilog can save many resources; the compilation report for an empty kernel code indicates that there are 40,650 ALUTs (9%), 52,976 FFs (6%), and 283 RAM blocks (11%) reserved and not available for use. The proposed design is operated on the target FPGA device at a 265 MHz clock frequency. The sample-input gray-scale image, shown in Figure 5 (a), is processed by the Sobel operator to produce the output image that contains all edges detected, as shown in Figure 5 (b).

https://doi.org/10.18489/sacj.v32i1.749

Figure 5: Sobel image detection result: (a) grey-scale image; (b) output image (contains edges)

https://doi.org/10.18489/sacj.v32i1.749

6 PERFORMANCE EVALUATION

The purpose of this study is to implement the Sobel edge detection algorithm to run on the FPGA platform (Terasic De5-net device) in order to improve speedup and energy consumption attributes compared to using a general single-core processor. Input data to this operator is a grayscale image with a 2D array of pixels. Experimentally, we examined several images with different sizes and recorded the execution time and the power consumption in every test. For verification purpose, we repeated each experiment with a specific input image size two hundred times and calculated the execution time by averaging the results. Power or energy consumption is also profiled in each iteration of the experiment; the total power consumption is the sum of the static, leakage, and dynamic power dissipation (Wiltgen et al., 2013). Static power consumption is measured while the device is idle, and this mainly depends on the voltage supply. Finally, dynamic power consumption for a given application is measured by averaging the power reading values while the application is running. Traditionally, the dynamic power consumption is a result of charging/discharging capacitors and therefore depends mainly on the frequency of operation, capacitance value, and the supply voltage (Silva et al., 2018).

The Kill A Watt EZ P4460 power meter device was used to measure the dynamic power consumption, as many studies, such as Bartram et al. (2010), employ this device. For the FPGA platform, the Power Analyzer tool available within the Intel Quartus software can be used to profile full power consumption details; many studies adopted this tool when it was necessary to profile the energy or power dissipation (Cromar et al., 2009; Hossain et al., 2011; Shah et al., 2012). Although the static power consumption in a CPU-based platform is higher than in the FPGA-based platform, the focus in this study is to analyze the dynamic power consumption, which is considered here for comparison purposes.

6.1 Execution Time analysis

According to the integrated FPGA programming model that uses the OpenCL framework, the data should be transferred from the host to the device through a PCIe connection, and results should be sent back to the host after completing the algorithm's execution. The overall time is the summation of both the execution time and the data transfer time. As a result, the performance bottleneck is not only the device-global memory access time but also the required time to transfer data between host and device. We measured data transfer time experimentally by running a large number of tests (one hundred times). Table 2 shows the average values in milliseconds. Table 3 lists the execution time profiled for each case. The second column displays the conventional execution time, the third column displays the optimized implementation, and the fourth column displays the average number of nanoseconds required to process each pixel; each pixel needs approximately between 0.7 to 0.8 nanoseconds for the large image size. This means five pixels can be processed every clock cycle, as shown in the fifth column. We measured the execution time experimentally and considered the possibility of a small margin

of error in this case. Table 4 lists the overall FPGA time results, which include both phases.

Table 2: Terasic De5-net FPGA: Average data transfer time (in ms) for different image sizes (in pixels)

Image size	Overall transfer time (ms)
144 x 256	0.07
240 x 426	0.19
360 x 480	0.30
480 x 640	0.47
720 x 1280	1.16
1080 x 1920	2.23
1440 x 2560	3.95
2160 x 3860	8.50
3480 x 5760	22.4

Table 3: Terasic De5-net FPGA, Sobel operator: Average execution time for different image sizes (in pixels)

Image size	Execution time			
	Conventional design (ms)	Optimised design (ms)	Optimised design per pixel (ns)	Optimised design pixels per clock
144 x 256	0.70	0.16	4.34	0.9
240 x 426	1.75	0.21	2.05	1.8
360 x 480	2.90	0.24	1.39	2.7
480 x 640	5.10	0.33	1.07	3.5
720 x 1280	15.35	0.80	0.87	4.3
1080 x 1920	33.90	1.50	0.72	5.2
1440 x 2560	59.95	2.90	0.79	4.8
2160 x 3860	138.0	6.00	0.72	5.2
3480 x 5760	363.5	16.0	0.80	4.7

https://doi.org/10.18489/sacj.v32i1.749

Table 4: Terasic De5-net FPGA, Sobel operator: Overall processing time (in ms) for different image sizes (in pixels)

Image size	Conventional design	Optimised design
144 x 256	0.77	0.23
240 x 426	1.94	0.40
360 x 480	3.20	0.54
480 x 640	5.57	0.80
720 x 1280	16.51	1.96
1080 x 1920	36.13	3.73
1440 x 2560	63.9	6.85
2160 x 3860	146.5	14.50
3480 x 5760	385.9	38.40

https://doi.org/10.18489/sacj.v32i1.749

We also profiled the overall execution time for a single-core CPU-based platform; Table 5 lists both the conventional and the optimized serial implementation results. The average number of nanoseconds required to process each pixel is approximately 28 nanoseconds, as shown in the fourth column. Figure 6 shows the gained speedup factor normalized to the conventional single-core execution time. As per Tables 4, 5, and 6, observations indicate that it is possible to gain a speedup factor of up to 88 times without holding data transfer time, and up to 37 times when considering the transfer time. Also, the performance speedup gained by optimized FPGA implementation is 36-fold and 25-fold when compared to optimized single-core and conventional FPGA implementations, respectively. Moreover, if we consider the data transfer time between host and device, then 15-fold and 11-fold enhancements are achieved, respectively.

Table 5: Single-core CPU-based platform, Sobel operator: Overall processing time (in ms) for different image sizes (in pixels)

Image size	Conventional design	Optimised design	Optimised FPGA per pixel (ns)
144 x 256	2.51	1.07	29.0
240 x 426	6.8	2.95	28.9
360 x 480	11.71	5.0	28.9
480 x 640	21	8.83	28.7
720 x 1280	62	25.10	27.3
1080 x 1920	133	55.0	26.5
1440 x 2560	240	98.0	26.6
2160 x 3860	527	214.0	25.7
3480 x 5760	1390	558.0	27.8

https://doi.org/10.18489/sacj.v32i1.749

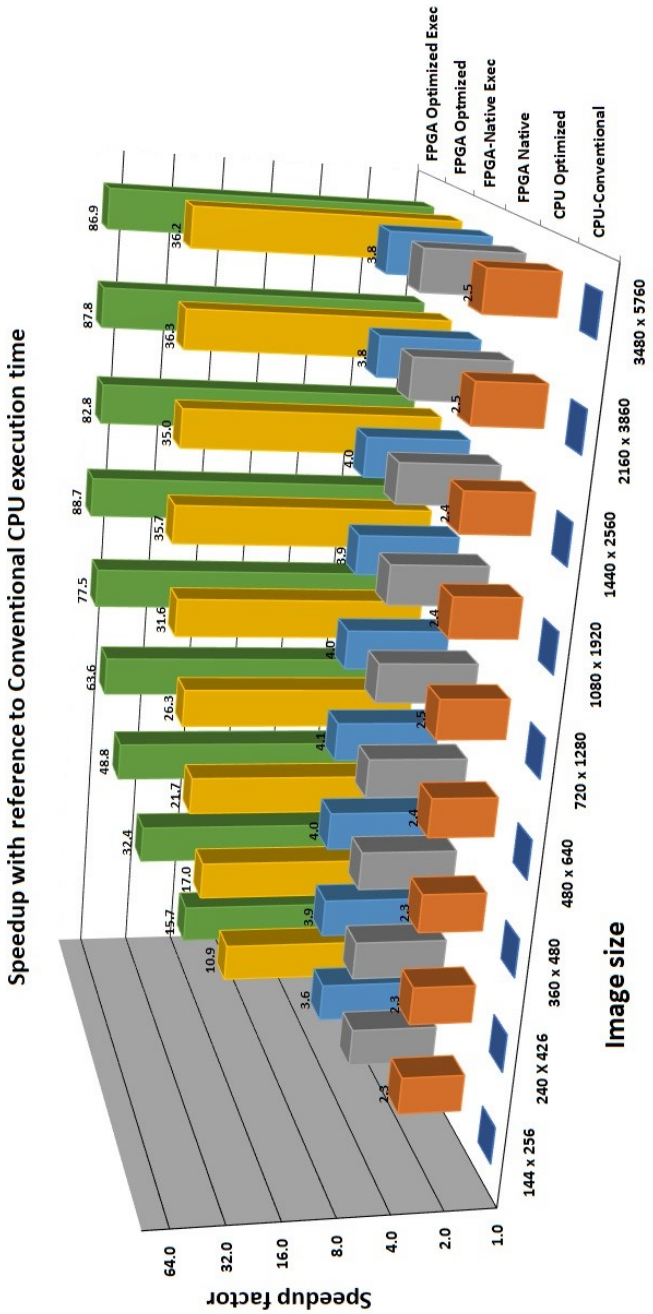

Figure 6: Execution time speedup factor regarding the conventional single-core Sobel operator implementation. FPGA-Native (Conventional) Exec and FPGA Optimized Exec do not consider the image data transfer time between host and device.

https://doi.org/10.18489/sacj.v32i1.749

6.2 Power and energy analysis

Here, we consider the dynamic power energy or power consumption, which is a result of raising the power level while the target application (Sobel edge operator) is running. The Power Analyzer tool is used to profile the power consumption on the FPGA device, while the Kill A Watt EZ P4460 device is used to profile the power consumption on the CPU-based platform. According to the experiments, the average dynamic power consumption when the Terasic De5-net FPGA is used is 7.687 W and approximately 31 W when the Intel Xeon E5 processor is used. We estimate the average consumed energy per image size through multiplying the average execution time by the average dynamic power consumption while the computation platform is running the proposed edge detection operator. The plan was to compare the energy consumption factor and show how the use of Intel FPGA acceleration devices significantly reduces energy consumption when compared to a general single-core processor (CPU-based platform).

Table 6 shows the average energy consumed (in Joules) per image sizes in three implementation cases:

1. optimized FPGA on Terasic De5-net device;
2. conventional CPU with default compiler optimization
3. optimized CPU through using an O3-compiler argument

Table 6: Energy consumption in Joules per image size

Image size	Conventional single-core	Optimised single-core	Optimised FPGA
144 x 256	0.078	0.033	0.003
240 x 426	0.363	0.155	0.011
360 x 480	0.363	0.155	0.011
480 x 640	0.651	0.274	0.017
720 x 1280	1.922	0.778	0.042
1080 x 1920	4.123	1.705	0.081
1440 x 2560	7.440	3.03	0.145
2160 x 3860	16.337	6.634	0.310
3480 x 5760	43.090	17.298	0.817

A point to consider is how to calculate the energy while the FPGA runs the operator. We highlight two cases:

1. data transfer between CPU and FPGA—hence, both platforms are considered to calculate the consumed energy
2. running the algorithm with FPGA—in this case, the Terasic De5-net is only considered in energy profiling.

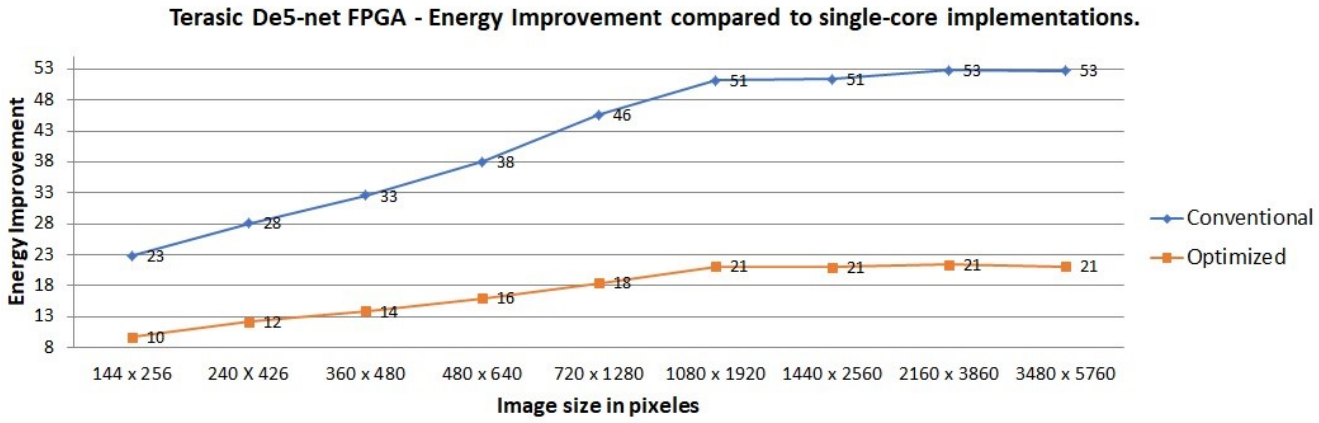

Figure 7: Total energy improvements when using the Terasic De5-net FPGA device. The energy improvement factor is normalized to the conventional and optimized implementations on the single-core processing element.

Figure 7 summarizes the energy improvement factor gained by using the Intel FPGA technology over using a regular single-core computation platform. As per the values in Figure 7, there is up to 21 times improvement compared to the optimized single-core design and up to 53 times compared to the conventional design. As per the values in Table 6, the improvement factor increases as the image size increases.

7 CONCLUSIONS

This study uses the Intel FPGA technology to demonstrate the feasibility of using this platform to improve the runtime and the energy consumption factors when running a common edge detection algorithm. We modified the proposed edge detection implementation according to the FPGA platform in which the task-parallel model is used to break the main task into multiple, simultaneously running subtasks, while the use of channels reduces the memory access time significantly. Other optimizations techniques are also used to reduce the initiation interval and to create an efficient pipeline structure to maximize the overall performance. The OpenCL framework is used to create an optimized design with a high level of abstraction such that it reduces the complexity of the hardware solutions. Our results demonstrate significant improvements in terms of running time and energy consumption, achieved by using the Intel FPGA technology.

https://doi.org/10.18489/sacj.v32i1.749

ACKNOWLEDGMENTS

This study was supported by the Deanship of Scientific Research at Yarmouk University under Grant number: 26/2018. It was also supported by the Intel FPGA university program Ticket numbers: LR4043 and BR 11211.

References

Abbasi, T. A. & Abbasi, M. U. (2007). A novel FPGA-based architecture for Sobel edge detection operator. *International Journal of Electronics*, *94*(9), 889–896. https://doi.org/10.1080/00207210701685253

Al-Omari, A., Griffith, J., Judge, M., Taha, T., Arnold, J. & Schuttler, H.-B. (2015). Discovering regulatory network topologies using ensemble methods on GPGPUs with special reference to the biological clock of Neurospora crassa. *IEEE Access*, *3*, 27–42. https://doi.org/10.1109/access.2015.2399854

Asghar, A. & Parvez, H. (2015). An improved diffusion based placement algorithm for reducing interconnect demand in congested regions of FPGAs. *International Journal of Reconfigurable Computing*, *2015*, 1–10. https://doi.org/10.1155/2015/756014

Asghari, M. H. & Jalali, B. (2015). Edge detection in digital images using dispersive phase stretch transform. *International Journal of Biomedical Imaging*, *2015*, 1–6. https://doi.org/10.1155/2015/687819

Bartram, L., Rodgers, J. & Muise, K. (2010). Chasing the negawatt: Visualization for sustainable living. *IEEE Computer Graphics and Applications*, *30*(3), 8–14. https://doi.org/10.1109/mcg.2010.50

Chaple, G. N., Daruwala, R. D. & Gofane, M. S. (2015). Comparisions of Robert, Prewitt, Sobel operator based edge detection methods for real time uses on FPGA, In *2015 International Conference on Technologies for Sustainable Development (ICTSD)*, IEEE. https://doi.org/10.1109/ictsd.2015.7095920

Chaple, G. & Daruwala, R. D. (2014). Design of Sobel operator based image edge detection algorithm on FPGA, In *2014 International Conference on Communication and Signal Processing*, IEEE. https://doi.org/10.1109/iccsp.2014.6949951

Chouchene, M., Sayadi, F. E., Said, Y., Atri, M. & Tourki, R. (2014). Efficient implementation of Sobel edge detection algorithm on CPU, GPU and FPGA. *International Journal of Advanced Media and Communication*, *5*(2/3), 105. https://doi.org/10.1504/ijamc.2014.060506

Cromar, S., Lee, J. & Chen, D. (2009). FPGA-targeted high-level binding algorithm for power and area reduction with glitch-estimation, In *Proceedings of the 46th Annual Design Automation Conference on ZZZ - DAC '09*, ACM Press. https://doi.org/10.1145/1629911.1630125

Deng, C., Ma, W. & Yin, Y. (2011). An edge detection approach of image fusion based on improved Sobel operator, In *2011 4th International Congress on Image and Signal Processing*, IEEE. https://doi.org/10.1109/cisp.2011.6100499

Dore, A. (2014). Performance analysis of Sobel edge filter on heterogenous system using OpenCL. *International Journal of Research in Engineering and Technology*, *03*(15), 53–57. https://doi.org/10.15623/ijret.2014.0315011

Firmansyah, I., Wijayanto, Y. N. & Yamaguchi, Y. (2018). 2D stencil computation on Cyclone V SoC FPGA using OpenCL, In *2018 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET)*, IEEE. https://doi.org/10.1109/icramet.2018.8683924

Fredj, H. B., Ltaif, M., Ammar, A. & Souani, C. (2017). Parallel implementation of Sobel filter using CUDA, In *2017 International Conference on Control, Automation and Diagnosis (ICCAD)*, IEEE. https://doi.org/10.1109/cadiag.2017.8075658

Ghosh, S. & Chapman, B. (2011). Programming strategies for GPUs and their power consumption, In *2011 International Conference on Parallel Architectures and Compilation Techniques*, IEEE. https://doi.org/10.1109/pact.2011.51

Halder, S., Bhattacharjee, D., Nasipuri, M. & Basu, D. K. (2012). A fast FPGA based architecture for Sobel edge detection, In *Progress in VLSI Design and Test*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-31494-0_34

Hill, K., Craciun, S., George, A. & Lam, H. (2015a). Comparative analysis of OpenCL vs. HDL with image-processing kernels on stratix-v FPGA, In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE. https://doi.org/10.1109/asap.2015.7245733

Hill, K., Craciun, S., George, A. & Lam, H. (2015b). Comparative analysis of OpenCL vs. HDL with image-processing kernels on stratix-v FPGA, In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE. https://doi.org/10.1109/asap.2015.7245733

Hossain, F. S., Ali, M. L. & Syed, M. A. A. A. (2011). A very low power and high throughput AES processor, In *14th International Conference on Computer and Information Technology (ICCIT 2011)*, IEEE. https://doi.org/10.1109/iccitechn.2011.6164810

Israni, S. & Jain, S. (2016). Edge detection of license plate using sobel operator, In *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, IEEE. https://doi.org/10.1109/iceeot.2016.7755367

Liao, Q., Hong, J. & Jiang, M. (2010). A comparison of edge detection algorithm using for driver fatigue detection system, In *2010 2nd International Conference on Industrial Mechatronics and Automation*, IEEE. https://doi.org/10.1109/icindma.2010.5538090

Medeiros, G. E., Bortolon, F. T., Reis, R. & Ost, L. (2018). Evaluation of compiler optimization flags effects on soft error resiliency, In *2018 31st Symposium on Integrated Circuits and Systems Design (SBCCI)*, IEEE. https://doi.org/10.1109/sbcci.2018.8533246

Nausheen, N., Seal, A., Khanna, P. & Halder, S. (2018). A FPGA based implementation of Sobel edge detection. *Microprocessors and Microsystems*, *56*, 84–91. https://doi.org/10.1016/j.micpro.2017.10.011

Shah, S. A. B., Nooshabadi, S. & Har, D. S. (2012). Efficient implementation of channel coding and interleaver for Digital Video Broadcasting (DVB-T2) on FPGA, In *2012 IEEE 16th International Symposium on Consumer Electronics*, IEEE. https://doi.org/10.1109/isce.2012.6241749

Silva, V. R. G., Furtunato, A., Georgiou, K., Eder, K. & Xavier-de-Souza, S. (2018). Energy-optimal configurations for single-node HPC applications [Last checked 04 Jul 2020]. https://arxiv.org/abs/1805.00998

Singh, A., Singh, M. & Singh, B. (2016). Face detection and eyes extraction using Sobel edge detection and morphological operations, In *2016 Conference on Advances in Signal Processing (CASP)*, IEEE. https://doi.org/10.1109/casp.2016.7746183

Tessier, R., Pocek, K. & DeHon, A. (2015). Reconfigurable computing architectures. *Proceedings of the IEEE*, *103*(3), 332–354. https://doi.org/10.1109/jproc.2014.2386883

Tian, J., Wu, J. & Wang, G. (2015). Realization of real-time Sobel adaptive threshold edge detection system based on FPGA, In *2015 IEEE International Conference on Information and Automation*, IEEE. https://doi.org/10.1109/icinfa.2015.7279750

Vanderbauwhede, W. & Benkrid, K. (Eds.). (2013). *High-performance computing using FPGAs*. Springer New York. https://doi.org/10.1007/978-1-4614-1791-0

Vanishree & Reddy, K. R. (2013). Implementation of pipelined Sobel edge detection algorithm on FPGA for high speed applications, In *2013 International Conference on Emerging Trends in Communication, Control, Signal Processing and Computing Applications (C2SPCA)*, IEEE. https://doi.org/10.1109/c2spca.2013.6749364

Waidyasooriya, H. M., Hariyama, M. & Uchiyama, K. (2018). *Design of FPGA-based computing systems with OpenCL*. Springer International Publishing. https://doi.org/10.1007/978-3-319-68161-0

Wan, P. & Uehara, M. (2012). Spam detection using Sobel operators and OCR, In *2012 26th International Conference on Advanced Information Networking and Applications Workshops*, IEEE. https://doi.org/10.1109/waina.2012.24

Wiltgen, A., Escobar, K. A., Reis, A. I. & Ribas, R. P. (2013). Power consumption analysis in static CMOS gates, In *2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI)*, IEEE. https://doi.org/10.1109/sbcci.2013.6644863

Xiangxi, Z., Yonghui, Z., Shuaiyan, Z. & Jian, Z. (2018). FPGA implementation of edge detection for Sobel operator in eight directions, In *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, IEEE. https://doi.org/10.1109/apccas.2018.8605703

Xu, J. (2011). OpenCL - The open standard for parallel programming of heterogeneous systems [Last checked 04 Jul 2020]. https://api.semanticscholar.org/CorpusID:16804330

Yasri, I., Hamid, N. & Yap, V. (2008). Performance analysis of FPGA-based Sobel edge detection operator, In *2008 International Conference on Electronic Design*, IEEE. https://doi.org/10.1109/iced.2008.4786751

Yasri, I., bin Hamid, N. H. & Yap, V. V. (2009). An FPGA implementation of gradient based edge detection algorithm design, In *2009 International Conference on Computer Technology and Development*, IEEE. https://doi.org/10.1109/icctd.2009.39

You, B., Sheng, W., Ma, H., Gu, Y. & Qin, Y. (2017). Implementation of sobel edge detection on FPGA based on OpenCL, In *2017 IEEE 7th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*, IEEE. https://doi.org/10.1109/cyber.2017.8446103

Yusoff, N. M., Halim, I. S. A., Abdullah, N. E. & Rahim, A. A. A. (2018). Real-time hevea leaves diseases identification using Sobel edge algorithm on FPGA: A preliminary study, In *2018 9th IEEE Control and System Graduate Research Colloquium (ICSGRC)*, IEEE. https://doi.org/10.1109/icsgrc.2018.8657603

Zhai, Y., Wang, G., Yu, H. & Wei, G. (2017). Research on the application of the edge detection method for the UAVs icing monitoring of transmission lines, In *2017 IEEE International Conference on Mechatronics and Automation (ICMA)*, IEEE. https://doi.org/10.1109/icma.2017.8015972

Zheng, H., Gurumani, S. T., Rupnow, K. & Chen, D. (2014). Fast and effective placement and routing directed high-level synthesis for FPGAs, In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays - FPGA '14*, ACM Press. https://doi.org/10.1145/2554688.2554775