FPGA IMPLEMENTATION OF A DENSE OPTICAL FLOW ALGORITHM USING ALTERA OPENCL SDK

A Thesis

by

Umut Ulutaş

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
August 2017

FPGA IMPLEMENTATION OF A DENSE OPTICAL FLOW ALGORITHM USING ALTERA OPENCL SDK

Approved by:

Assoc. Prof. H. Fatih Uğurdağ, Advisor
Department of Electrical and Electronics Engineering
Özyeğin University

Asst. Prof. Ahmet Tekin
Department of Electrical and Electronics Engineering
Özyeğin University

Prof. Sezer Gören Uğurdağ
Department of Computer Engineering
Yeditepe University

Date Approved: 15 August 2017

To My Mother and My Sister

ABSTRACT

FPGA acceleration of compute-intensive algorithms is usually not regarded feasible because of the long Verilog or VHDL RTL design efforts they require. Data-parallel algorithms have an alternative platform for acceleration, namely, GPU. Two languages are widely used for GPU programming, CUDA and OpenCL. OpenCL is the choice of many coders due to its portability to most multi-core CPUs and most GPUs. OpenCL SDK for FPGAs and High-Level Synthesis (HLS) in general make FPGA acceleration truly feasible. In data-parallel applications, OpenCL based synthesis is preferred over traditional HLS as it can be seamlessly targeted to both GPUs and FPGAs. This thesis shares our experiences in targeting a demanding optical flow algorithm to a high-end FPGA as well as a high-end GPU using OpenCL. Throughput and power consumption results on both platforms are presented.

ÖZETÇE

Yoğun hesaplama gerektiren algoritmaların FPGA üzerinde geliştirilmesi, gerektirdikleri uzun Verilog veya VHDL RTL tasarım sürelerinden ötürü genellikle çok makul değildir. Paralel hesaplama yapılınabilir algoritmaların hız için alternatif bir platformu vardır: GPU. GPU programlama için CUDA ve OpenCL dilleri yaygın şekilde kullanılmaktadır. OpenCL çoğu çok çekirdekli işlemci ve çoğu GPU'ya taınabilir olması nedeniyle bir çok programcının seçeneğidir. FPGA'ler için OpenCL SDK ve Yüksek Seviyeli Sentez (YSS) genel olarak FPGA hızlandırmasını gerçekten mümkün kılmaktadır. Paralel programlanabilir uygulamalarda, geleneksel YSS'ye göre OpenCL tabanlı sentez tercih edilir çünkü sonu ç hem GPU'ları hem de FPGA'leri sorunsuz bir şekilde hedef alabilir. Bu tezde, yoğun hesaplama gerektiren bir optik akış algoritmasını OpenCL kullanarak yüksek düzey bir GPU'da ve yüksek düzey bir FPGA'de çalıtırarak ulaılan sonuçlar paylaşılmıştır. Her iki platformda da hız ve güç tüketimi sonuçları sunulmuştur.

ACKNOWLEDGMENTS

This thesis work has been at the crossroads of multiple projects, namely, European Union Artemis JU Project called ALMARVI (GA 621439) and TÜBİTAK projects ARDEB-114E343 and KAMAG-114G029. My financial support came from TÜBİTAK project numbered KAMAG-114G029, which is being carried out under the leadership of Asst. Prof. Furkan Kıraç with help from my thesis advisor, Assoc. Prof. H. Fatih Uğurdağ.

I would first like to thank my two professors who made this thesis possible, Asst. Prof. Furkan Kıraç (for my research assistantships appointment) and my advisor Assoc. Prof. H. Fatih Uğurdağ for their patience, motivation, and extensive knowledge. Without their support it would not be possible to conduct this work.

I would also like to thank the rest of my thesis committee: Asst. Prof. Ahmet Tekin and Prof. Sezer Gören Uğurdağ.

I would like to thank all the members of nEMESysLab at Ozyeğin University for helping me even with my slightest problems that I have faced during this thesis and also being such good friends.

Lastly, I would like to thank my mother and sister for the endless encouragement, support, and attention.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This work benchmarks a dense optical flow algorithm on both FPGA and GPU using OpenCL language and related tools. This thesis may be useful to people interested in accelerating optical flow in particular or accelerating compute-intensive data-parallel algorithms using OpenCL on FPGA and/or GPU. Although various research has done on similar topics, this is the only work so far that covers acceleration of optical flow on FPGA versus GPU using OpenCL.

1.1 Problem Statement

Optical flow is a motion analysis algorithm that takes two 2D images and finds the motion vectors of each pixel for the first image according to their movement by comparing it to the second image. Optical flow has been widely studied in the literature. Two methods that are widely used for optical flow are Lucas-Kanade method [5] and Horn-Schunk method [6], which also happen to be the works that popularized the subject in the literature.

Dense optical flow is a type of optical flow that is designed to process all pixels in an image to compute the motion vectors, unlike sparse optical flow, which looks for interesting regions to estimate the motion. Since dense optical flow works on every pixel and gives a motion vector for each of them, it requires a higher amount of memory and computational power. It is suitable to compute dense optical flow in platforms that allow parallel computation in higher degrees such as GPUs [7], [8], [9] and FPGAs [10], [11].

Open Computing Language (OpenCL) is a language that is mainly used for multi-core programs. OpenCL can be used on many different platforms such as FPGA, CPU, GPU, and DSPs. This opens up many possibilities. It is possible to run the same program on different platforms, greatly increasing its availability and portability. Also, it makes it easier to carry out performance tests and benchmarking across platforms for a given project. On FPGAs, OpenCL code can synthesized and converted to HDL. That means the user can work on a re configurable platform using C-like languages without directly writing RTL code. Since it is more desirable, FPGA companies developed toolchains/SDK based on OpenCL and research was conducted in the literature on optimizing OpenCL for FPGAs [10], [11].

The thesis proceeds by first presenting the particular optical flow algorithm that is implemented. Later, some implementation details are presented. Then the last section contains the results of FPGA/GPU comparisons, and conclusion.

1.2 Contributions of the Thesis

This thesis contributes to the literature by

- Executing a certain Optical Flow algorithm in OpenCL on FPGA, CPU, and GPU.
- Giving detailed analyses of the test results and evaluate said platforms for future use.
- Comparing the use of OpenCL on applications for FPGA instead of using RTL.

CHAPTER II

RELATED WORK

Optical Flow and OpenCL are large subjects that are worked on the literature very often. The brief history of optical flow subject are given below and later respective FPGA and OpenCL applications explained.

Numerous works have been published in the literature that deal with acceleration of optical flow on FPGA versus GPU (e.g., [1], [2]) and that deal with OpenCL based acceleration on FPGA versus GPU (e.g., [3], [4]).

2.1 History of Optical Flow

As said, optical flow has been widely studied in the literature. Two methods that are named as widely used and popularizing factors for optical flow are Lucas-Kanade method [5] and Horn-Schunk method [6]. Those two algorithms are sparse optical flow algorithms. That means instead of scanning the whole frame and comparing every pixel with the next frame, they look into critical regions and try to predict the movement. By doing that saving from both computation time and power is aimed.

Horn-Schunk algorithm assumes that between each frame a certain amount of time has passes and each object have displaced to another location by a small amount. It predicts the movements of objects based on the change of the intensity levels of a gray-scale image. For example, lets say the image is gradually becoming darker from left side to right side originally. On the next frame if an object is becoming darker, algorithm predicts that the object may moved in to right direction based on the change of the intensity. It may not give best results if the light/brightness of the scene is inconsistent. It is an iterative dense optical flow method, meaning it iterates through every pixel and making calculation on each of them. It can even use more than two frames if possible.

Lucas-Kanade is another popular optical flow algorithm, and it is sparse optical flow, meaning it does not iterate through each pixel every time but tries to find interesting regions/objects and tries to predict their movement. At the end, it gives faster results than Horn-schunk although it may not be feasible between two frames if the object moved very fast or got completely out of the frame. It is best when the overall flow is slow. it may not give best results if the light/brightness of the scene is inconsistent or the displacement of the objects are large.

Another big work optical flow is done by Middlebury university. They created a very large optical flow benchmark page where very high amount of optical flow algorithms are presented and compared according to their performance to ground truth. Ground truth is the true optical flow values between two images where manually calculated using advanced photography techniques. Fig.1 shows two frames with slight amount of movement between them and the ground truth image in the middle where it shows the true optical flow.

The Middlebury work also gives little information about each selected optical flow algorithm and it gives its frame execution time. Although frame execution times are not entirely comparable since the codes are tried on different platforms/computers and most of them written on different languages.

2.2 GPU

A lot of research has been made about testing, developing and optimizing the OpenCL platform on the literature, and big majority of them are about OpenCL applications that run on GPU's.

The research paper "GPU Implementation of an Anisotropic Huber-L Dense Optical Flow Algorithm Using OpenCL"[7] is as its name suggests about implementing

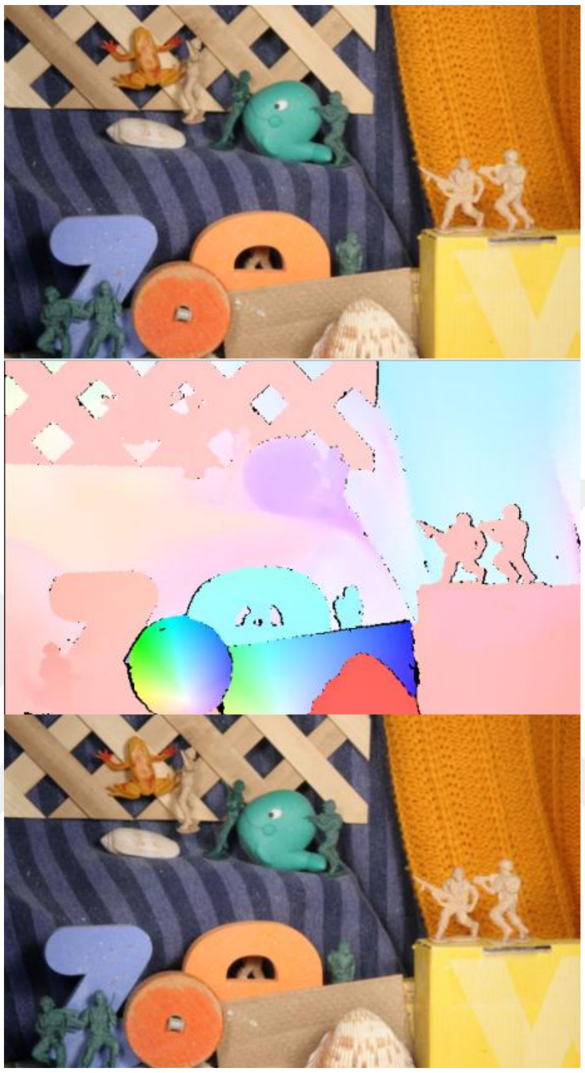

Figure 1: Two frames from Middlebury optical flow research and its ground truth

and testing the Anisotropic Huber-L Dense Optical Flow Algorithm on OpenCL platform. It purely focuses on comparing the CPU code that runs on Matlab versus the GPU code that is implemented by OpenCL. This paper is one of the main origin points of this thesis work. It gives a little bit insight about the implementation details and concludes the huge performans improvements by using GPU to compute such algorithms that are very suitable to parallel computations.

"A Self Organization-Based Optical Flow Estimator with GPU Implementation" [8] work is again about optical flow and GPUs. The work presents a new optical flow estimator method which itself called self organization-based. One difference is that in this work author used CUDA to execute its own optical flow method on the GPU instead of using OpenCL.

2.3 FPGA

There are again lots of work about FPGAs, HLS, OpenCL in general, and GPU vs FPGA comparisons. However, three researches listed below are very similar in terms of both content and conclusion. They are discussed in detail.

"Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis"[3] work is mainly about comparing the FPGA and GPU. It executes several different algorithms on both of them. It compares the standard GPU version with the High Level Synthesis based FPGA version. It makes tests about its speed and power consumption and discusses the results. It concludes that FPGAs are always better in terms of energy while most algorithms perform better on GPUs in terms of speed.

In "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs"[11], similarly to the previous work, the author makes a GPU vs FPGA comparison. However this time, author uses OpenCL to directly map the GPU design into the FPGA. It is again concluded that while the code being easily portable into the FPGA and FPGA's power consumption are nice bonuses, GPU's performance is usually better since the ported code is specifically GPU optimized.

"Program Acceleration in a Heterogeneous Computing Environment Using OpenCL, FPGA, and CPU"[4] is a very similar work listed above. This time author uses specifically parallel computing in Heterogeneous Computing Environment and makes performance testes. It being very different from the other two examples, its very hard to make a definitive conclusion based on test results. Sometimes FPGA performs better than GPU and CPU, sometimes GPU takes the lead and under specific conditions

CPU may over perform the FPGA. At the end author specifies how using OpenCL instead of RTL languages makes FPGA development so much easier since the language itself is easier and used by more people.

CHAPTER III

ENVIRONMENT

In this chapter, detailed analysis of the environment that thesis is built upon is given. First the hardware components, then the used software are explained.

3.1 Hardware

Since the work done on three different hardware; CPU, GPU and FPGA, in this section general short information about each hardware is given and the specification of the chosen hardware are shared.

At the beginning, it was planned to make this work run on Cyclone V FPGA board. However, since the project requires large amount of work and include 10 kernel functions, Altera Offline Compiler simply couldn't be able to fit the design into Cyclone V board. After various failing attempts, it is decided to work on a more powerful available FPGA board.

Selected FPGA board is the Nallatech Arria 10 385A FPGA Accelerator Card. Nallatech 385A Arria10 FPGA has 1506 system logic elements, ensuing a large space for calculations. Its peak floating point performance is 1366 GFLOPS [16].

For GPU, Nvidia GeForce GTX 980 Ti with 6GB GDDR5 Memory, 336.5 GB/sec memory bandwidth and 1000 MHz clock speed is chosen for the implementation of the algorithm.

The CPU of the main host computer is Intel Xeon 1650 v3 3.5 GHz.

3.2 Sofware

This section gives necessary information about all the software that is used during developing this work.

3.2.1 Nallatech BSP / MMD

The host application is linked with the board-specific Memory-Mapped Design (MMD). This MMD is the communication layer between the upper Altera OpenCL SDK software and accelerator board. The MMD provided by Nallatech OpenCL BSP is used as custom platform in this application [15].

3.2.2 Altera Offline Compiler

Altera Offline Compiler uses the BSP to be able to map the kernel designs into a specific FPGA board. It tries the best way to map the kernels into the FPGA, it may try several different approaches during its run. One kernel compilation process takes up to between 12 and 18 hours. After the compilation is done a board specific AOCX file is produced as the product to be used as kernels in the OpenCL design.

3.2.3 AOC Simulation and Debugger

the emulator could be used instead of FPGA to quickly test the project. Once "-march=emulator" is added to the line, it makes it to use the emulator instead of compiling for FPGA. Emulation time is less than 30 seconds as opposed to hours of real compiling, making it suitable for testing quick changes on the code.

AOC also provides an emulation option to debug and test the designs beforehand. It takes the information of the targeted FPGA and creates an emulation as if the design is working on the targeted FPGA. In VHDL, debugging is done via verifying specific bits in simulations, however, AOC provides GNU debugger support in a software environment. While emulating the hardware design on the host computer, the designer is able to debug the prototype by using regular debugging methods with GNU debugger, such as creating software breakpoints and tracing variables.

Another perk of this emulation is that, on traditional VHDL systems, the design and test of host code begins after kernels are targeted on to FPGA. This creates two different design and testing processes to be handled. While, in AOC emulation, both kernels and the C/C++ host code that operates/uses kernels are designed and tested together, making it possible to analyze and set the timing and power during the design phase of kernels.

CHAPTER IV

OPTICAL FLOW

In this chapter, a general definition for optical flow is given and some optical flow types are discussed. Then, the selected algorithm to be implemented for this research is explained in detail.

4.1 Definition

Optical Flow Estimation is about estimating the motion between two frames and produce motion vectors as results. This motion vectors hold the information about each pixel's movement according to the previous pixel.

There are two main types of optical flow estimation algorithms, sparse and dense. In dense optical flow, every pixel is iterated through in calculations. Although it is more likely to give high precision results, it is also computationally expensive since every pixel has to be iterated in each frame. In sparse flow, only the regions of interest are processed through the algorithm. Those region of interests represents the areas that are most likely to move and detecting those interesting areas varries between algorithms.

4.2 Anisotropic Huber-L Algorithm Design

The optical flow estimation algorithm of "Anisotropic Huber-L^1" has been selected as it ensures good quality and is easily suitable to parallel computing. According to Middlebury optical flow benchmark results [12], it has an average rating of 62.6. It is also one of the algorithms that can provide very fast results, making it very suitable for video processing.

The starting point of calculations is to find a solution to the optical flow problem based on L^1 data term and isotropic TV regularization, which is disparity preserving and spatially continuous formulation. For two input images I_0 and I_1 defined on a regular rectangular domain, $\Omega \in$ the function can be formalized as Eq. 1:

$$min\left\{\int_\Omega \sum_{d=1}^2 |\nabla u_d| + \lambda |I_1(\vec{x}+\vec{u}(\vec{x})) - I_0(\vec{x})| d\vec{x}\right\} \tag{1}$$

where $\vec{x} = (x_1, x_2)$ is the vector that contains coordinates of corresponding pixels in the image and $\vec{u}(\vec{x}) = (u_1(\vec{x}), u_2(\vec{x}))$ is the flow vectors that include displacement values. Free parameter λ is used to balance the relative weight of the data and regularization term.

L^1 data term is the distance between motion warped I_1 image by using flow vectors $I_1(\vec{x}+\vec{u}(\vec{x}))$ and I_0 image. Also, regularization term is the sum of absolute gradients of 2D flow/displacement vectors. To further simplify the minimization calculation, new auxiliary variables are introduced. Vector $\vec{v}$ and connecting term to guarantee that $\vec{v}$ is in near proximity of $\vec{v}$. Then, anisotropic Huber regularization is applied to the formula and some other mathematical operations to simplify it further, which is explained in detail in the anisotropic Huber-L^1 optical flow paper. This results in:

$$\underset{\vec{u},\vec{v}}{min} \underset{|\vec{p}_d|\leq 1}{sup} \left\{\int_\Omega \sum_{d=1}^2 [(D^{\frac{1}{2}}\nabla u_d)\cdot \vec{p}_d - \epsilon\frac{|\vec{p}_d|^2}{2} + \frac{1}{2\theta}(u_d - v_d)^2] + \lambda|\rho(\vec{v}(\vec{x}))| d\vec{x}\right\} \tag{2}$$

Anisotropic Huber in this equation (Eq. 2) is the regularization term that points to the final shape, because it is image-driven and discontinuity preserving, also uses Huber cost. Huber cost for small differences is quadratic, while for large differences it is linear. A new equation is produced by solving this optimization problem iteratively using alternating minimization procedure. For fixed $\vec{v}$ solution yields to:

$$u_d^{n+1} = v_d^n + \theta div(D^{\frac{1}{2}} p_d^{-n+1}) \tag{3}$$

$$p_d^{-n+1} = \frac{p_d^{-n} + \tau(D^{\frac{1}{2}} \nabla u_d^{-n+1} - \epsilon p_d^{-n})}{max\{1, |p_d^{-n} + \tau(D^{\frac{1}{2}} \nabla u_d^{n+1} - \epsilon p_d^{-n})|\}} \tag{4}$$

Then, the solution for $\vec{u}$ results:

$$\underset{\vec{v}}{min} \left\{ \int_\Omega \frac{1}{2\theta} \sum_{d=1}^{2} (u_d - v_d)^2 + \lambda |\rho(\vec{v}(\vec{x}))| d\vec{x} \right\} \tag{5}$$

A final thresholding part is applied with three cases results in a direct solution:

$$v^{-n+1} = u^{-n+1} + \begin{cases} \lambda\theta\nabla I_1, if \ \rho(u^{-n+1}) < -\lambda\theta|\nabla I_1|^2 \\ -\lambda\theta\nabla I_1, if \ \rho(u^{-n+1}) > -\lambda\theta|\nabla I_1|^2 \\ -\rho(u^{-n+1}) \frac{\nabla I_1}{|\nabla I_1|^2}, else \end{cases} \tag{6}$$

Equations 3, 4, and 6 are the formulae to be used to calculate the flow, by replacing the corresponding operations with discrete values that are calculated on 2D image grids.

CHAPTER V

OPENCL IMPLEMENTATION

Altera OpenCL SDK is used to implement this application, which is an OpenCL synthesis tool for FPGAs. Altera OpenCL SDK's programming model includes three parts: host application, OpenCL kernels, and a custom platform that provides the board design as shown in Fig. 2. The language is C/C++ for the host application and a specific C-like language for the kernels. They are compiled by Altera Offline Compiler (AOC). Altera OpenCL puts a hold to compilation until the last phase of the design, decreasing the productivity bottlenecks of VHDL tools. The design can easily be ported to other FPGAs that supports Altera OpenCL. The compiler also optimizes the design flow itself. Ability to work through a C-like language for the hardware makes it easier to design specialized hardware architectures without relying on managing complicated RTL codes [13].

5.1 Main Structure

OpenCL kernels contain the core algorithm. Then, AOC converts those kernel codes to hardware design. Kernels are used in the host application. The OpenCL host application uses standard OpenCL runtime APIs to manage device configuration, data buffers, kernel launches, and synchronization. Host application also contains functions such as file I/O, or portions of the source code that do not run on an accelerator device [14].

5.2 Kernel Details

A total of 10 OpenCL kernels are used in the OpenCL application, 3 of which can be categorized as main kernels and 7 as auxiliary kernels. In the host code, to find the

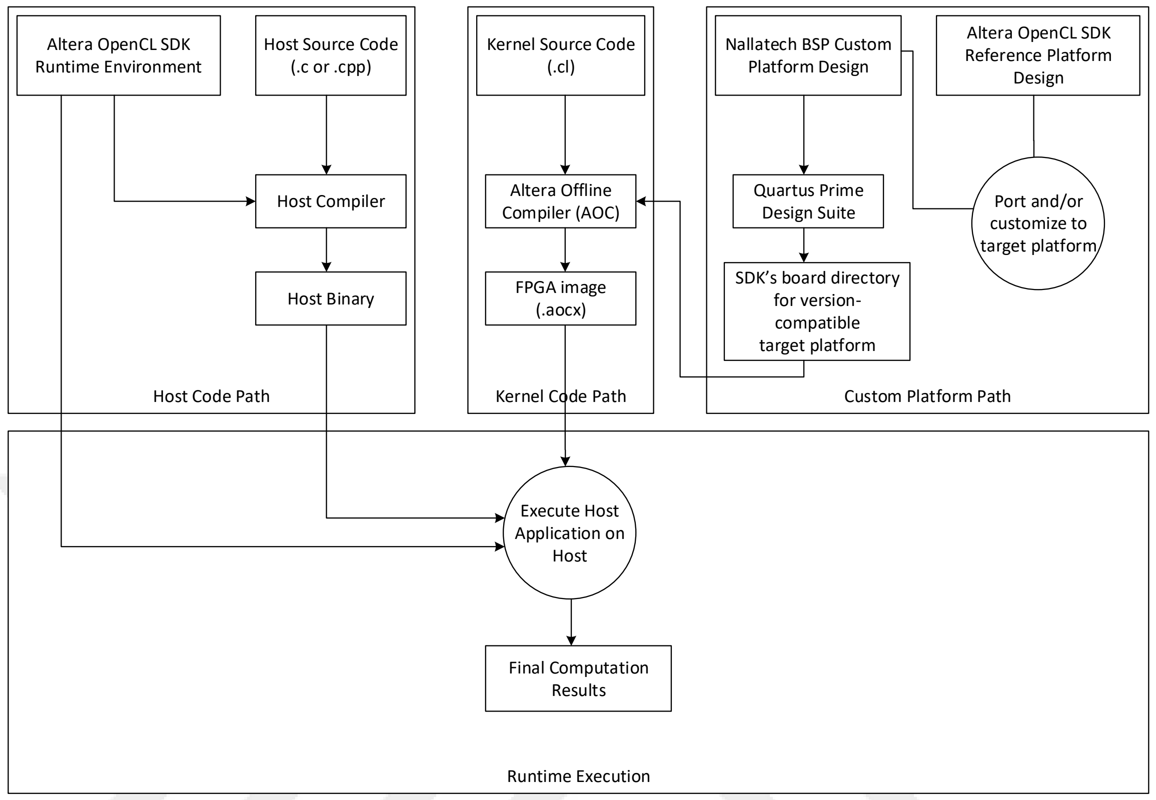
Altera OpenCL SDK Runtime Environment
Host Source Code (.c or .cpp)
Host Compiler
Host Binary
Host Code Path
Kernel Source Code (.cl)
Altera Offline Compiler (AOC)
FPGA image (.aocx)
Kernel Code Path
Nallatech BSP Custom Platform Design
Altera OpenCL SDK Reference Platform Design
Quartus Prime Design Suite
Port and/or customize to target platform
SDK's board directory for version-compatible target platform
Custom Platform Path
Execute Host Application on Host
Final Computation Results
Runtime Execution

Figure 2: Schematic diagram of the Altera OpenCL SDK programming model

optical flow of the video, the video is first read using OpenCV. Two frames are read in succession and the optical flow is calculated using OpenCL on these frames. Once the optical flow is calculated, the values are stored in a file. In addition, these values are displayed on screen as a new video where the optical flow can be visually seen on screen as it is calculated. Those are the kernels that are used in the process are those, first 3 of which (Warping, Compute-P and Loop Flow) are the main kernels:

- Warping
- Compute-p
- Loop Flow
- Gaussian (3 variations)

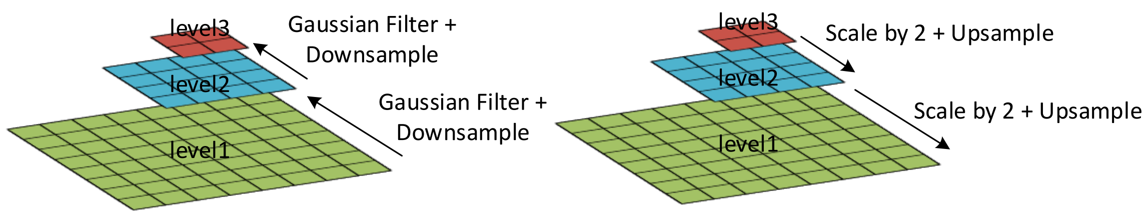

Figure 3: Image and flow vector pyramids

- Upscale
- Copy (2 variations)
- Zerosize

The calculations are done on 3 Gaussian pyramid levels for each frame as shown in Fig. 3. Warping is called 10 times for each level, compute-P and loop flow are called 50 times for each warp. Auxiliary kernels are also used many number of times each frame when necessary. To explain the application, it is better to go over kernels one by one, explaining each of them while also explaining the overall process.

5.2.1 Warping

The warping part is the part in which 2D image is updated based on the motion vectors calculated so far. Here, the kernel takes two images and their motion vectors (u and v vectors) as input. The image is updated using the motion vectors u and v. That is, each pixel in the frame is matched with the corresponding motion values from the vectors u and v, and new values are calculated using those values. The warping kernel also calculates difference between the second frame and the warping applied first frame and creates a vector of it.

In the warping process, it is not possible to directly move to pixel to its motion warped location and read the values from there. Because the values in motion vectors are usually floating point numbers rather than integers. Instead, each pixel in the first frame is moved according to the vector u and v, and the value is taken from the values of pixels in a 2x2 window at the target point. The values are calculated using bilinear interpolation from the 2x2 pixels in the window. Warping kernel calculates this interpolation manually.

As input, kernel takes two frames, and u and v motion vectors. Picture type is set to unsigned char*, and the type of the vector is set to float*. The kernel also takes width, height, and stride parameters to do corner detection, which their type is set to int. The kernel outputs x, y, t, and gradSqr vectors, whose types are float* similar to other vectors. Those input/output valus and their relationship with global memory is shown in Fig.4

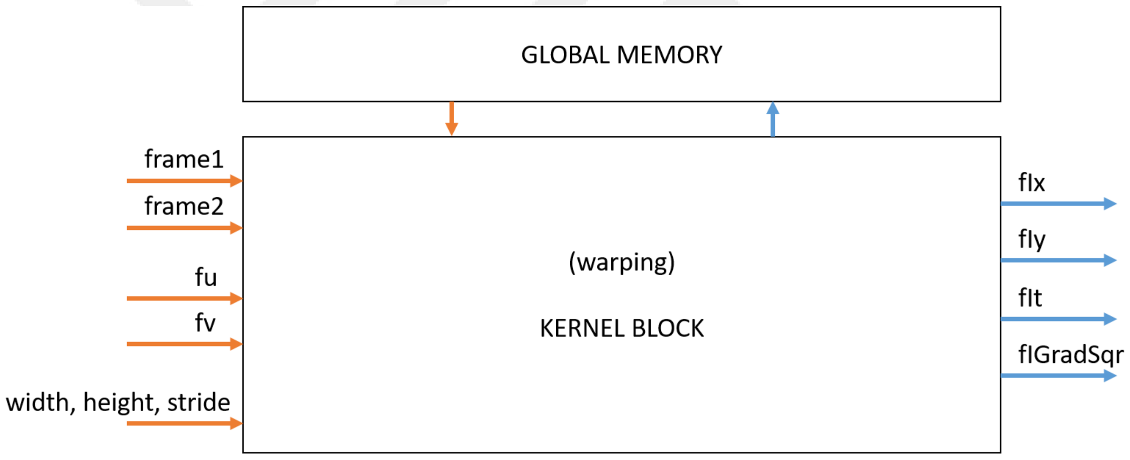

Figure 4: Warping kernel design schema

When using OpenCL kernels, global and local workgroup sizes are specified. Local workgroups are small units within the global workgroup, working in parallel within themselves. The size of the global workgroup has been chosen to be the same as the height and width in the warping kernel, and the size of the local workgroup is 64x4. Hence, OpenCL processes 64x4 pixels at the same time in a frame. The buffer sizes of all other parameters (vectors and squares) that the kernel receives as input, except for the width, height, and step parameters, are set to be the same as the global workgroup size.

In the process; the pixel is mapped according to the values in the u and v motion vectors and their values are assigned to a newly created fx and fy floating points. These values are increased by one to produce fx1 and fy1 values. Those values are converted to integer and four points are created using those values (fx, fy), (fx, fy1), (fx1, fy), (fx1, fy1). The values at these four points enter the linear interpolation and the value of the current pixels warped value at global index is calculated.

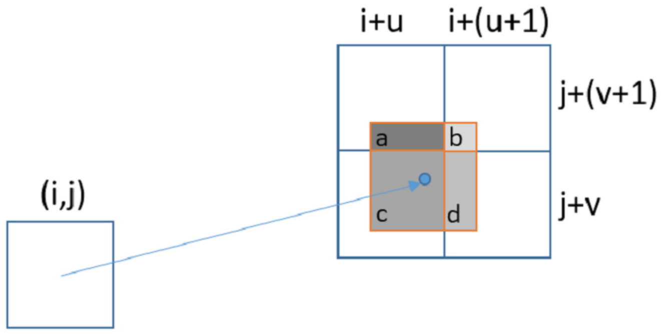

Figure 5: Warping process

These operations are illustrated in Fig. 5 To better illustrate the point, it is best to go over the point (i, j) in the figure and explain the process. The aim is to find the warping values for that point. This point is shifted by the values in the u and v motion values and the new point is found. As said, these values are usually not integer numbers, so that surrounding pixels in the target coordinate are also included in the work, and these 4 pixels ((i + u, j + V), (i + u, j + (v + 1)) and (i + (u + 1), j + (v + 1)) at the destination are used in the warping calculation..

These 4 pixel values are processed and the result (i, j) becomes the warped value of our pixel. For this operation a linear interpolation is applied. To understand it better, it is best to think of a square in the size of a pixel drawn around our target point. The value of a pixel we draw around this point is calculated by proportioning the areas (a, b, c, and d fields) within the 4 target pixels. For example, since the area of the rectangle c in the diagram is the largest, the share of the point (i + u, j + v) in the final value of the point (i, j) is greater than the share of the point (i + (u + 1), j + (v + 1))s also the smallest because b has the smallest rectangle area.

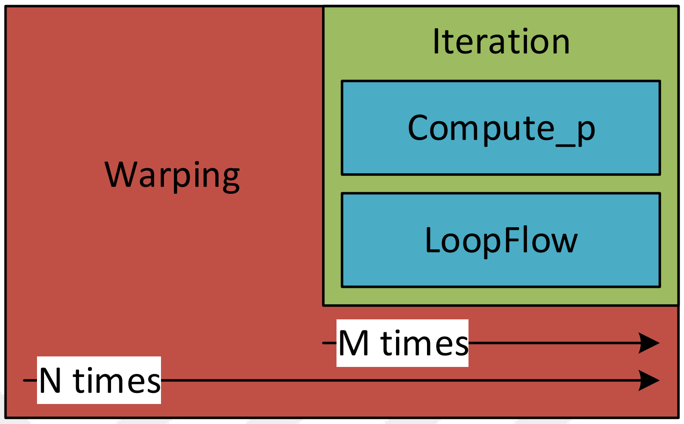

Figure 6: Algorithm design flow

This warping process is executed 10 times each pyramid level is shown in Fig. 6

5.2.2 Compute-P

P vectors are calculated using the motion vectors u and v, and the vector w. Later, p vectors are also used in the Loop Flow kernel to calculate these u, v, and w vectors. Compute-p kernel calculates and updates the p vectors. Then the Loop Flow kernel comes in and uses the previously computed p vectors and updates the vectors u, v, and w. These two kernels do calculations one after another, passing their output data/vectors to each other in 50 iterations is shown in Fig. 6. It is aimed to produce the most accurate result in this way.

One p vector holds the x and y change values separately for the vectors u, v, and w. Therefore, 6 p vectors are defined. 3 of those vectors are for horizontal displacements while the other 3 are vertical.

The six p vectors shown in Fig 7 is designed to hold these three vectors vertically and horizontally. For example, suppose that the target pixel is (i, j), motion vector u's vertical displacement values are calculated as p2.

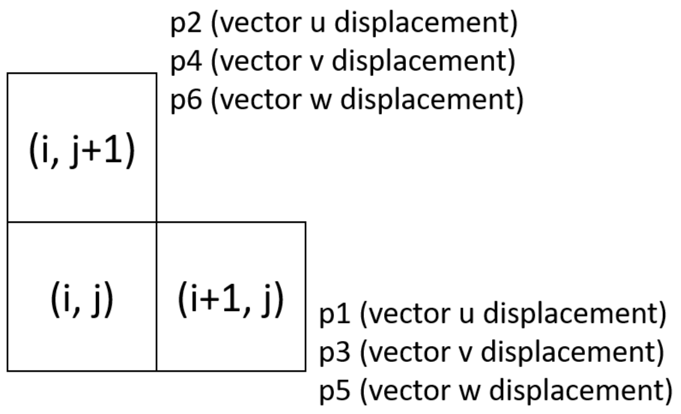
p2 (vector u displacement)
p4 (vector v displacement)
p6 (vector w displacement)
(i, j+1)
(i, j)
(i+1, j)
p1 (vector u displacement)
p3 (vector v displacement)
p5 (vector w displacement)

Figure 7: 6 p vectors that are calculated for every pixel

The calculation process of p vectors can be explained by showing the calculation of one of the six. For point (i,j), the vertical p vector that uses the derivative of u motion vector is calculated as follows:

$$p[i][j] = p[i][j] + (u[i][j+1]u[i][j]) * f1$$

Similarly, vertical and horizontal displacements of u, v and w vectors are calculated and all vectors are found. After calculating all p vectors, the kernel normalizes the values between 1 and 0.

While computing the p vectors, threads need to read the same pixel values when processing adjacent pixels, the kernel can take advantage of the local memory buffer. For this reason, this kernel is prepared using local memory cache. For vector type, float4 data type was used in this kernel. It is a data type in OpenCL that provides 4 floating point numbers.

The kernel takes 6 p vectors, u and v motion vectors and w auxiliary vector as input. All of these are specified as float4, as indicated. It also takes the auxiliary parameters height, width, and stride values as input. These auxiliary parameters are of type int. The stride parameter is used to calculate the coordinate values in the vector using the global indexes. The height and width parameters are also used in

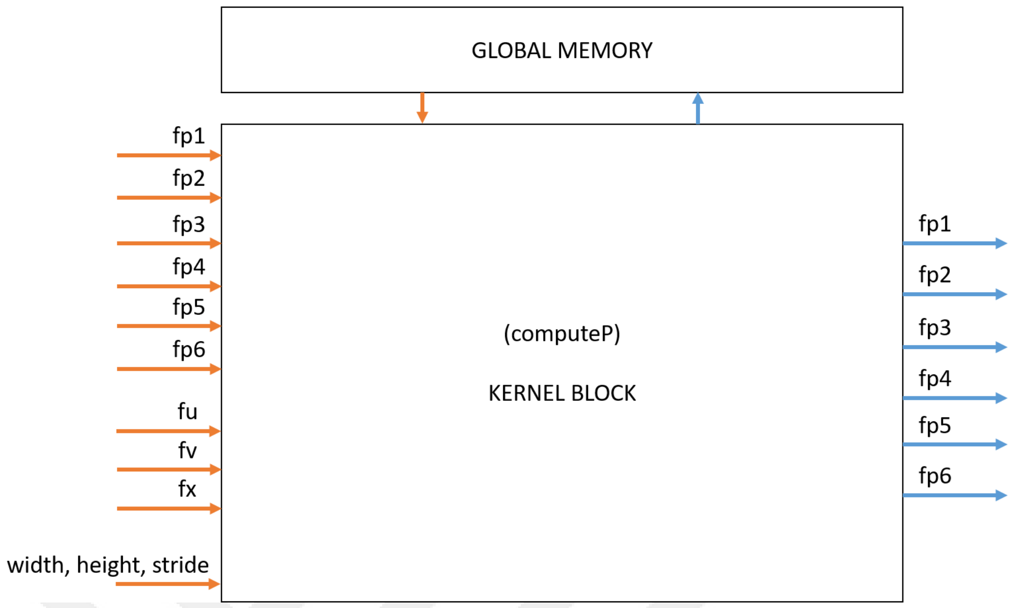

Figure 8: Compute-P kernel design schema

corner detection, that is, to determine whether the algorithm's pixel window is on the corner of the frame, where different treatments are applied to solve this particular situation. Kernel outputs 6 updated p vectors with new calculations. Compute-P kernel design schema is shown in Fig 8.

For this kernel, the size of the global workgroup is chosen as the height of the image, and its width is one-fourth of the width of the image. This is because float4 data type is used, so there are 4 calculations at each point. The size of the local workgroup is also set to 32x4. Similarly, the buffer sizes of all other parameters (vectors and squares) that the kernel receives as input, except for the width, height, and step parameters, are set to be the same as the global workgroup size.

5.2.3 Loop Flow

Loop Flow kernel runs in the same iteration with Compute-p kernel, and they feed each other. As previously said, while Compute-p kernel calculates p vectors using u, v, and w vectors, Loop Flow kernel calculates u, v, and w vectors using p vectors. Also, in Loop Flow kernel x, y, and t vectors, which are the output vectors of Warping kernel, are used in the calculations of u, v, and w vectors. After calculating u, v, and w vectors, the kernel applies thresholding.

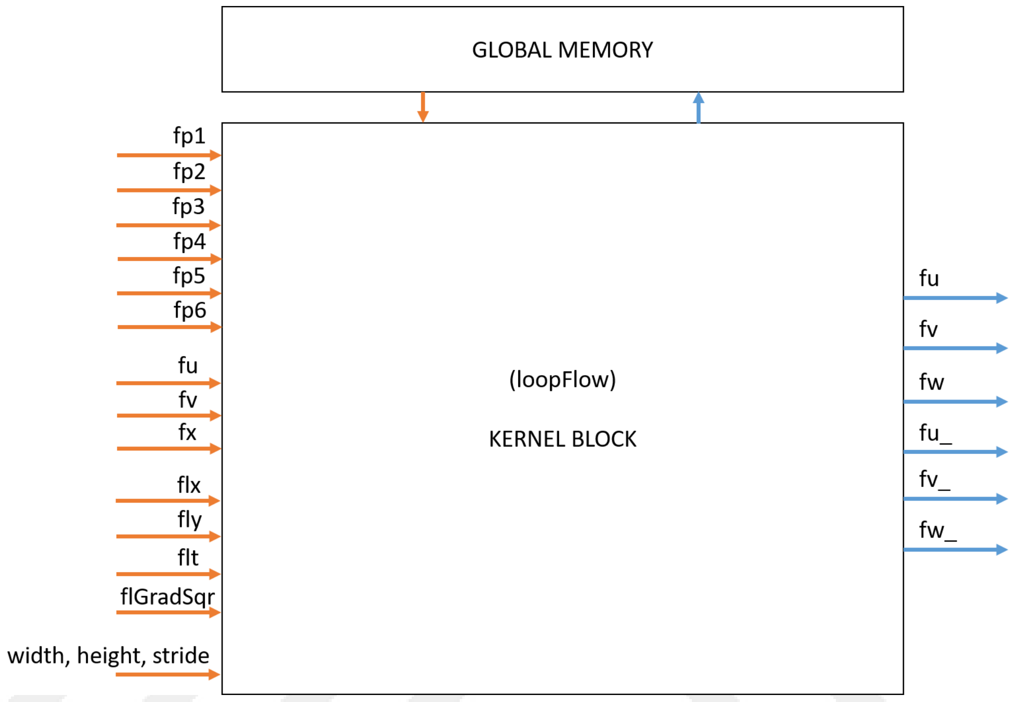

Figure 9: Loop Flow kernel design schema

Inputs and outputs of the kernel are similar to Compute-P kernel with some reversed roles and extra inputs. The kernel receives 6 p vectors, u and v motion vectors and w auxiliary vector as input. All of these are specified as float4. Auxiliary parameters width and stride are also used as input. As stated, it takes vectors x, y, and t computed in warping kernel as input and uses them to update u, v, and w vectors. Their type is also specified as float4. The kernel also outputs updated u, v, and w vectors. Those are shown in Fig. 9.

5.2.4 Gauss

This kernel takes two 2D image vectors and one filter as input. It applies the filter to the first image source and writes the filtered results to the second image. It then gives the second image as output as shown in Fig. 7.

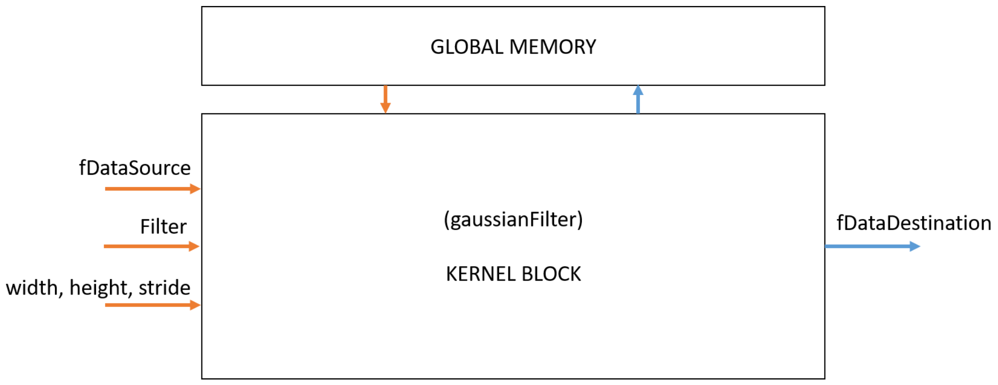

Figure 10: Gauss kernel design schema

5.2.5 Upscale / Downscale

It takes a data source in float4 type and a scale value in integer type as input. First, it upsamples the data source by 2, which fills the empty gaps with duplicating pixels, then scales the data source by the scale value. It then outputs the result. Upscale kernel schema is shown in Fig. 11.

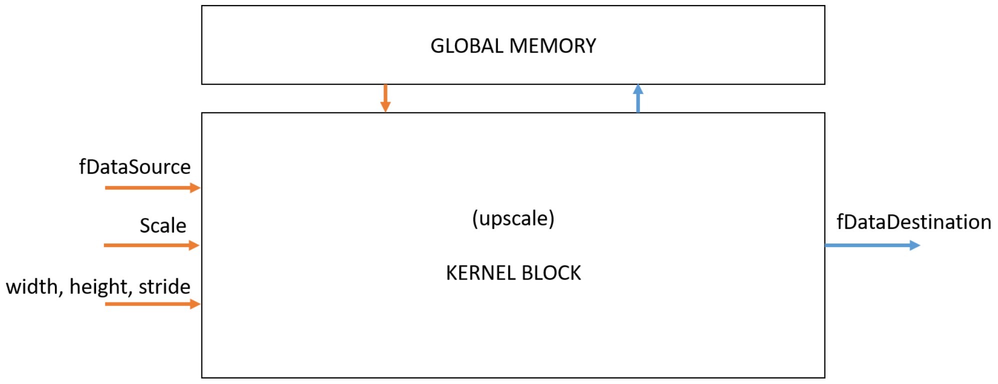

Figure 11: Upscale kernel design schema

5.2.6 Auxiliary Kernels

5.2.6.1 Copy

It takes a data source as an input, copies this data source, and outputs it. The types are specified as float4. There are two variations of this kernel, one of them produces one copy and other produces two copies of the data source.

5.2.6.2 Zerosize

This kernel takes a data source in the form of float4 as input, then makes all the bits of the source equal to zero, then outputs it.

Since they share a similar inputs and outputs, their kernel design schema can be shown in one figure, which is shown in 12.

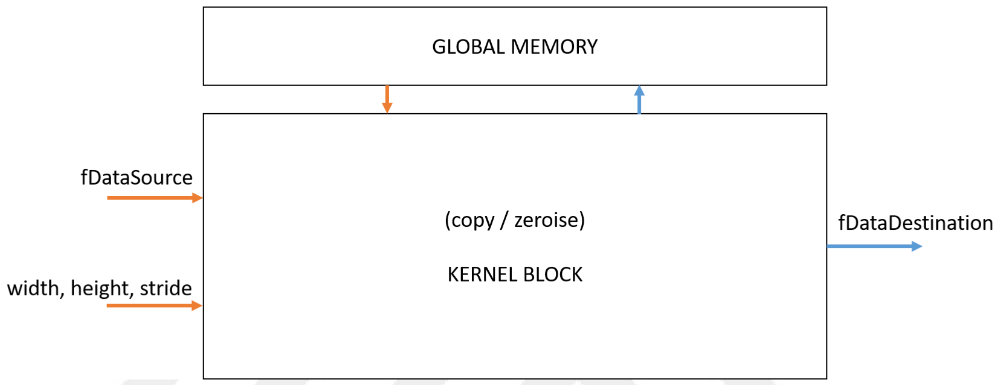

Figure 12: Copy and Zeroise kernel design schema

5.3 Techniques

5.3.1 OpenCL Structure

In this section a general information about how an OpenCL application is built and what other techniques are used to make the application to work on FPGAs. In a generic OpenCL program, those following functions are used, and they are generally called in this order:

- clGetPlatformIds
- clCreateContext
- clCreateCommandQueue
- clCreateProgramWithSource
- clCreateKernel

- clCreateBuffer
- clEnqueueWriteBuffer
- clsetKernelArgs
- clEnqueueNDRangeKernel
- clFinish
- clEnqueueReadBuffer
- Release Objects

It is best to briefly explain those functions and give information about where and why each of them is used in this work.

5.3.1.1 *clGetPlatformIds*

They are used to determine which platform and device the program will run on. It is generally used to choose whether the project will run on CPU or GPU, and run on which GPU/CPU. In this work, it is also used to specify the project will select the FPGA as the target platform. Although it requires other modifications, this is the part where it is selected the optical flow work will run on either CPU, GPU or FPGA.

5.3.1.2 *clCreateContext*

After the device is chosen, a context is created to manage command queue, kernels and programs. In this work one context is created to manage them all. Context is the selected device group.

5.3.1.3 *clCreateCommandQueue*

It is used to create a command-queue in the selected context. OpenCL functions that are sent to a command-queue are queued and called at the time the calls are made.

Again, for this project, one command queue is created to handle all OpenCL function calls.

5.3.1.4 clCreateProgramWithSource

It is used to read the kernel file to create the program object. In the GPU and the CPU version this function gets called as many times as the number of kernels since all kernels are available in separate files. However, in the FPGA version only one program object is created because all kernel functions being in the same single AOCX file. In GPU and CPU versions it reads CL files while in the FPGA function, as said, it reads AOCX files that are created by the Altera Offline Compiler.

5.3.1.5 clCreateKernel

It takes a program object and creates a kernel to be used. In both versions it gets called as many times as number of kernels. One difference is that the program object it takes as input changes as kernel changes in the GPU/CPU version while in the FPGA version all kernels are read and created from the same program object. Its input arguments are the kernel function name, which is the name of the function in the kernel file and a valid program object.

5.3.1.6 clCreateBuffer

It is used to reserve space in the selected device. Context gets size and flag as inputs. Flag is there to specify the purpose of this space (read only, write only, read and write). In the project, the two buffers are created for the two frames that will be worked on. Their types are defined as read only since only read operation will be done on those frames to calculate optical flow. Apart from those two frames, for every input and output vector of a kernel, a buffer is created. That is in total more than 40 buffers to hold those vectors.

5.3.1.7 *clEnqueueWriteBuffer*

This function is used to write data to buffers from the host memory. It is used in this work in only two instances, where the two subsequent frames are read and written into buffers. Other generated buffers are all handled (written into or read from) by kernels. The generated buffers allow us to write data from the host memory. Mainly as input it takes a valid buffer object and the pointer array of the data.

5.3.1.8 *clsetKernelArgs*

It is a simple function that is used to set the inputs and outputs of kernel functions. Gets called many times for each argument of each kernel.

5.3.1.9 *clEnqueueNDRangeKernel*

The main function to execute a kernel function. For example, in this work, for each compute-P kernels, this function gets called as number of iterations in each warp (gets called 50 times each warp to execute compute-P when iterations are set to 50.)

5.3.1.10 *clFinish*

It is used to wait all operations to finish since the program cannot pass this line unless all command-queue operations are finished. In this work this is used between many kernel executions since many of the kernels are very dependent the each other in terms of input are output, mainly being loop flow kernel and compute-P kernel.

5.3.1.11 *clEnqueueReadBuffer*

As similarly to write buffer function, this lets you read values from a specific buffer. At the final stage of the algorithm, calculated optical flow U and V vectors are read from their respective buffers, later to be used to create the visible optical flow frame on the screen.

5.3.1.12 Release Objects

Six functions were used to release the objects after the operations are finished: clReleaseEvent, clReleaseMemObject, clReleaseKernel, clReleaseCommandQueue, clReleaseContext, clReleaseProgram.

5.3.2 FPGA and OpenCL Interaction

As said, OpenCL applications consist two parts: Kernel and Main Code. When working on a GPU, kernel files have to extension of .cl and they are designed to work on GPUs. To make kernels work on FPGA, another set of instructions are needed. Those operations can be explained in 3 steps:

- Gather all kernels in a single kernel file
- Use Altera Offline Compiler to compile them
- Use the newly created aocx file in the program instead of kernels

Now lets look at those steps in detail. Collecting all kernel file into a single kernel file is necessary since the Altera Offline Compiler maps this kernel file into FPGA as a whole. Simply all kernel functions are cut from their respective files and put into a long single file. As said, then Altera Offline Compiler is used to map those kernels to FPGA and produce a design that can be used with OpenCL. The following command is used to execute the process:

```
aoc device/opticalFlow.cl -o bin/opticalFLow.aocx
--fp-relaxed --board s5_ref
```

Once the Offline Compiler start the process it may take a very long time. When operation finishes, an AOCX file and many auxiliary files are produced. These auxiliary files are the compilation reports and summaries. They consist of information like what would be the speed that FPGA works with a certain design or what is going to be the space that the design will cover once it is in the FPGA. Since in this project high amount of kernels used, at the first phases Altera Offline Compiler was unable to fit the all kernels in a single AOCX file, because FPGAs available area was not enough to hold such design. After that some optimization's, like combining some kernels into a single function, are made on the kernel files to be able to fit the design into the FPGA.

After producing the AOCX file, next step is to use this AOCX file instead of kernel CL files on the actual OpenCL application. Here are the steps that has to be applied to the main and the kernel code to be able to make it run on FPGAs:

- Memory allocations should be in the form of alignedMalloc instead of regular malloc. On OpenCL FPGA architectures, alignment requirement are necessary for memory allocations. Alignment requirements specify what address offsets can be assigned to what types. Allocating the host-side buffers allows direct memory access transfers to occur to and from the FPGA, which improves buffer transfer efficiency.

- In the OpenCL application, as explained in the above, kernel and the functions within the kernel are introduced separately in the code. Since this project has 11 kernels originally, 11 different kernel objects are created and for each of them, one kernel function object is created, resulting again total of 11 objects. However, since we have 1 AOCX file instead of 11 CL files, the host code has to create only one kernel object with 11 functions. Mainly kernel declaration function that is called 11 times for each kernel removed and replaced with a function that read the AOCX file once and creates a single object. Later that object is used to call the kernels (as functions) within the AOCX file.

- Syntax of many object declarations on the main code also should be updated according to be suited to FPGAs. OpenCL specific object declarations like cl_memory, cl_kernel, cl_command_queue should be done in scoped_array format. After all objects are declared within scoped arrays, using them should be inside for loops that iterates through potential devices, since all objects are now in arrays.

- FPGA should be identified instead of the GPU as a target platform. If the FPGA and Altera Tools are correctly installed on the machine, simply changing the target platform to Altera is enough to make the main code to recognize the FPGA.
- While reading values from clEnqueueReadBuffer, blocking flag should be chosen instead of non-blocking flag as choosing non-blocking results in a performance decrease in the final outcome.
- Insert the restrict keyword in pointer arguments in kernels whenever possible. Including the restrict keyword in pointer arguments prevents Altera Offline Compiler from creating unnecessary memory dependencies.
- Performance can be further improved by unrolling the loops on OpenCL Kernels which decreases the number of iterations that the kernel executes. This is done by adding an unroll parameter before the loop that is wanted to be unrolled.
- All created OpenCL objects should be reset before any of them are used if they are not already.

The folder structure of the project is shown in Fig.13. Although the original kernel files and Altera Offline Compiler produces AOCX files don't need to be in the same file structure, it is a good practice to put them in respective folders since a change in the kernels requires the recompilation of the AOCX file. So at the very end, device folder holds kernel files, bin folder holds board specific AOCX file and the host folder holds the main host code.

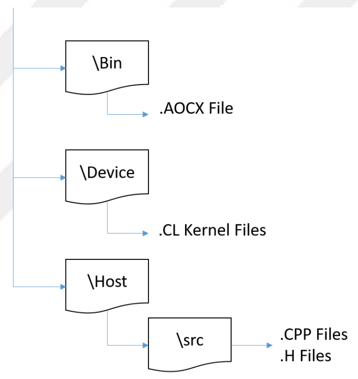

Figure 13: Project Directory Structure

CHAPTER VI

RESULTS

The technical details of the FPGA implementation are shown in Table 1 and 2. In Table 1, the results show when the design is compiled normally, while in Table 2, the results show when the design is compiled using another method for optimization. In here, the optimization method used is "fp-relaxed" (float point relaxed), a method which further optimizes the floating point declarations in kernels, resulting in slightly fewer total RAM and DSP block usage in the FPGA.

The specifications of the environment used for the tests are:

- **CPU:** Intel Xeon 1650 v3 3.5 GHz
- **GPU:** Nvidia GeForce GTX 980 Ti with 6GB GDDR5 Memory, 336.5 GB/sec memory bandwidth and 1000 MHz clock speed
- **FPGA:** Nallatech 385A Arria10 FPGA Accelerator Card with 8GB DDR3 on-card memory, up to 1.5 TFlops and 2133 MHz clock speed

All results were obtained on the same host computer to be able to make a fair comparison. During measurements, execution is the only system function that spends serious amount of system resources. Measurements were done in three categories

Table 1: Resource Utilization of Arria10 FPGA without any optimization directive

Logic Utilization (in ALMs)	213,940 / 427,200 (50%)
Total Registers	446,217 / 1,708,800 (26%)
Total RAM Blocks	1,947 / 2,713 (72%)
Total DSP Blocks	763 / 1,518 (50%)
Total PLLs	60 / 112 (54%)

Table 2: Resource Utilization of Arria10 FPGA with fp-relaxed optimization directive

Logic Utilization (in ALMs)	212,076 / 427,200 (50%)
Total Registers	446,217 / 1,708,800 (26%)
Total RAM Blocks	1,939 / 2,713 (71%)
Total DSP Blocks	714 / 1,518 (47%)
Total PLLs	60 / 112 (54%)

depending on the device, where the main calculations are made: CPU, GPU, and FPGA. GPU and FPGA ran the same OpenCL kernels with some modification in the host code to make it suitable for the desired device. The OpenCL application is implemented using double precision floating point numbers. CPU measurements were done in MATLAB, and it is also double precision because of MATLAB being double precision inherently.

In all the tests, respectively 50 and 2 iterations per warp were carried out; those are the iterations consisting of computing p vectors and calculating loop flow as explained above. Here are the values of the constants that are used in all three tests: $\lambda = 40$, $\beta = 0.01$, $\tau = 1/\sqrt{8}$ and $\epsilon = 0$. All measurements are done on the same videos three variations in different resolutions.

The main hardware setup that is used for this project is shown in Figure 14 and Figure 16. A regular main board with the PCI-express slots is used. Host CPU and the FPGA or GPU communicates through the PCI-express.

Power consumption measurements are done via connecting a device that measure the total power of the PC. This device is shown in Figure **??**. First, the idle power consumption of the PC is measured. Then optical flow calculations are started and the power consumption is measured again. Net difference between the idle state and working state is noted as the power consumption of the used device.

Frame execution times of CPU, GPU, and FPGA are shown in Table 3. These results show that the GPU offers a much better performance compared to the FPGA.

Figure 14: FPGA connected to host main board PCI-express slot

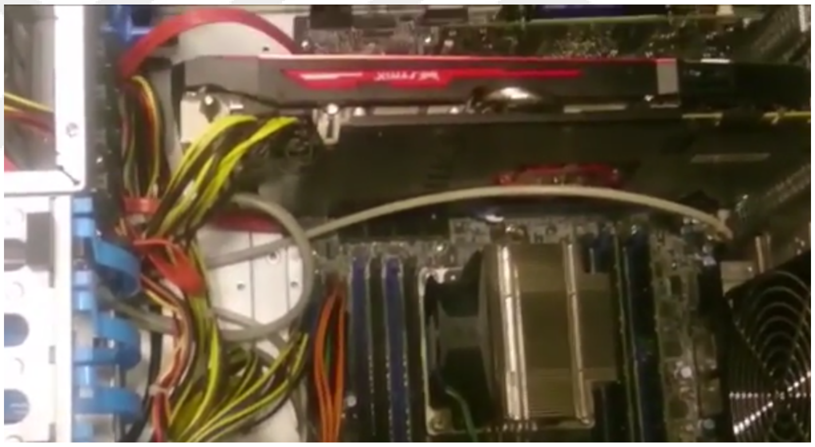

Figure 15: GPU connected to host main board PCI-express slot

Note that the FPGA used (15 billion transistors in 20 nm) was even more high-end than the GPU (8 billion transistors in 28 nm). The GPU frame execution times that are obtained in this work are quite competitive; for instance, you may compare them to the frame execution times reported in [7] by some of the authors of this paper.

As said, the results for CPU times that are shown in the Table 3 is calculated in Matlab. Since Matlab code is written in an entirely different language and manner, those results may be less acceptable to compare with the results of OpenCL GPU and FPGA implementations. Also Matlab itself performing slower in general than

Figure 16: Power meter is connected between the main computer and plug

C++ and OpenCL for this particular project, it further makes it harder to compare those values side by side.

To be able to make a more precise comparison, it is better to use the same OpenCL code that runs on GPU and FPGA but set it to run on CPU. Table 4 shows the CPU execution times when OpenCL code chooses CPU as its target platform. Although as expected the results are better in OpenCL CPU implementation than the Matlab implementation, since the optical flow application requires high amount of parallel computing, the results are still slower than GPU and FPGA implementations of the same code, since those two devices as, expected, are better suited to parallel computing.

It is possible to able to obtain higher performance on the FPGA than GPU in [17] when the design was directly done in Verilog instead of OpenCL. Therefore, this work shows that with an OpenCL based design flow, speedup potentials of FPGAs may not be fully utilized although impressive gains are possible in design time.

Table 6 and Table **??** show the power consumption measurements between FPGA and CPU. While FPGA offers better performance over CPU as shown before, it also requires a lot less power to progress.

Table 3: CPU(Matlab), FPGA, GPU frame execution times in sec. for 50 iterations per warp

Resolution	CPU	FPGA	GPU
576x384	107	2.734	0.128
640x480	156	3.594	0.168
768x576	471	4.854	0.254

Table 4: CPU, FPGA, GPU OpenCL frame execution times in sec. for 50 iterations per warp

Resolution	CPU	FPGA	GPU
576x384	8.6	2.734	0.128
640x480	9.6	3.594	0.168
768x576	11.2	4.854	0.254

Table 5: FPGA and CPU frame execution times in and power consumption values for 50 iterations per warp

Resolution	FPGA Frame Time	CPU Frame Time	FPGA Power	CPU Power
576x384	2734 ms	8600 ms	15 W	51 W
640x480	3594 ms	9600	19 W	51 W
768x576	4854 ms	11200 ms	22 W	52 W

Table 6: FPGA and CPU frame execution times and power consumption values for 2 iterations per warp

Resolution	FPGA Frame Time	CPU Frame Time	FPGA Power	CPU Power
576x384	201 ms	783 ms	10 W	46 W
640x480	269 ms	860 ms	11 W	46 W
768x576	362 ms	1030 ms	11 W	46 W

Table 7: FPGA and GPU power consumption with equal frame execution times for 50 iterations per warp

Resolution	Frame Time	FPGA Power	GPU Power
576x384	2734 ms	15 W	58 W
640x480	3294 ms	19 W	58 W
768x576	4854 ms	22 W	60 W

Table 8: FPGA and GPU power consumption with equal frame execution times for 2 iterations per warp

Resolution	Frame Time	FPGA Power	GPU Power
576x384	201 ms	10 W	56 W
640x480	269 ms	11 W	57 W
768x576	362 ms	11 W	59 W

The results in Table 7 were obtained after the GPU was slowed down to the same frame execution times as the FPGA 8. These results show that even with an OpenCL based flow FPGAs offer significantly lower power consumption compared to GPU implementations. Note that these power consumption figures were obtained after subtracting the power consumption of the idle system from the actual power consumption. Lowering GPU power consumption by slowing down its clock instead of adding sleep cycles is also tried. This did not seem possible on Nvidia GPUs, whereas it may be possible on, for example, AMD GPUs.

The results in Table 8 fare even better for the FPGA. That is, when the compute requirements of the algorithm is lowered (2 versus 50 iterations), then the FPGA starts producing more acceptable frame rates and yet become even more power efficient compared to the GPU.

Output of the FPGA implementation is shown in Fig. 17. The left image is the input frame, while the right image is a visualization of the motion vectors produced from an input frame and its previous frame. Stationary parts are shown in white, while moving pixels of the frame are color-coded indicating different motion directions and speeds.

CHAPTER VII

CONCLUSIONS & FUTURE WORK

In this work, anisotropic Huber-L optical flow estimation algorithm was implemented on Nallatech 385A Aria10 FPGA board using Altera OpenCL. Details of the algorithm was explained, and implementation process on the FPGA was discussed. Performance and power consumption results were presented for GPU and FPGA. CPU performance results are also given.

On the basis of performance results, when FPGA design is compared with the CPU implementation, FPGA design provides 30-100x better performance in execution times. On the other hand, when FPGA Altera OpenCL design is compared to GPU OpenCL design, a decline in the performance is observed. Although the result depends on the input videos resolution, GPU implementation can process a single frame up to 20x faster on the average than the FPGA implementation. The warping parameters and iteration numbers can be further reduced to achieve faster frame execution times, but this also reduces the quality of the optical flow estimation, resulting in a trade-off between execution time and quality. Note that better performance on FPGA can be obtained with domain-specific synthesis tools even when OpenCL is used [**?**].

Although FPGA performs worse than the case where the algorithm is coded via RTL, OpenCL still can be preferable because of a few reasons. In OpenCL, it is relatively easier to adapt the existing GPU projects to work in FPGAs, while in RTL probably whole project needs to be redesigned. Also developing time in RTL designs are higher compared to near standard C++ designs. This also makes it harder to find the available man power for the project, since coding in RTL is harder and requires more work than OpenCL C++.

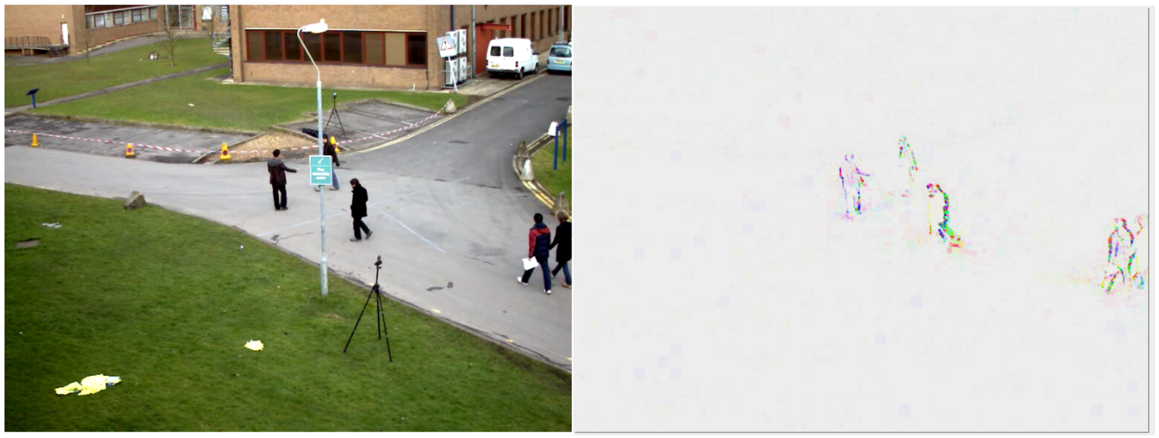

Figure 17: FPGA output

It is found that the power consumption is a lot less in FPGA implementation, resulting around 4x better power consumption compared to the GPU implementation depending on the input resolution. In the end, it may be more suitable to use the FPGA implementation based on the needs of a designer, since although FPGA implementation results in higher frame execution times, it has much better power consumption. It can suit projects where low power consumption is required or designs that the extra performance that is provided by the GPU is not needed.

Developments planned to be made on the project can be viewed under following titles:

- The OpenCL code that is written for FPGA would be further optimized using ”pipe” structures. Those ”pipe” structures connects kernels in such a way that they can send and recieve data between themselves without coming back to the host code.

- The OpenCL code would be tried on several different GPU, CPU and FPGA’s to make the work more passable.

- Apart from pipes, further optimization techniques would be researched, making kernels utilize the bigger part of the FPGA, making it run faster at the end.

REFERENCES

[1] K. Pauwels, M. Tomasi, J.D., E. Ros, and M.M. Van Hulle, “A Comparison of FPGA and GPU for Real-Time Phase-Based Optical Flow, Stereo, and Local Image Features,” IEEE Trans. on Computers, vol. 61, no. 7, 2012, pp. 999-1012.

[2] J. Bodily, B. Nelson, Z. Wei, D.-J. Lee, and J. Chase, “A Comparison Study on Implementing Optical Flow and Digital Communications on FPGAs and GPUs,” ACM Trans. on Reconfigurable Technology and Systems, vol. 3, no. 2, 2010, Article 6.

[3] F. bin Muslim, L. MA, M. Roozmeh, and L. Lavagno, “Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis,” IEEE Access, vol. 5, 2017, pp. 2747-2762.

[4] H.N. Hoffman, “Program Acceleration in a Heterogeneous Computing Environment Using OpenCL, FPGA, and CPU” University of Rhode Island DigitalCommons@URI, Open Access Master’s Thesis, 2017.

[5] B.D. Lucas and T. Kanade. “An iterative image registration technique with an application to stereo vision,” In Proceedings of International Joint Conf. on Artificial intelligence (IJCAI), vol. 2., San Francisco, CA, USA, 1981. pp. 674-679.

[6] B.K.P. Horn and B.G. Schunck. “Determining Optical Flow,” Technical Report. Massachusetts Institute of Technology, Cambridge, MA, USA, 1980.

[7] D. Büyükaydın and T. Akgün, “GPU implementation of an anisotropic Huber-L1 dense optical flow algorithm using OpenCL,” In Proceedings of International Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), Samos, Greece, 2015, pp. 326-331.

[8] M. Shiralkar, "A Self Organization-Based Optical Flow Estimator with GPU Implementation," All Dissertations. 2010. pp. 630.

[9] A. Wedel, T. Pock, C. Zach, H. Bischof, and D. Cremers. "An Improved Algorithm for TV-L1 Optical Flow," In: Cremers D., Rosenhahn B., Yuille A.L., Schmidt F.R. (eds) Statistical and Geometrical Approaches to Visual Motion Analysis. Lecture Notes in Computer Science, vol 5604. Springer, Berlin, 2009.

[10] I. Janik, Q. Tang, and M. Khalid, "An overview of Altera SDK for OpenCL: A user perspective" In Proceedings of IEEE Canadian Conf. on Electrical and Computer Engineering (CCECE), Halifax, NS, Canada, 2015, pp. 559-564.

[11] H. R. Zohouri, N. Maruyamay, A. Smith, M. Matsuda and S. Matsuoka, "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs," In Proceedings of International Conf. for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 2016, pp. 409-420.

[12] Middlebury Optical Flow webpage, http://vision.middlebury.edu/flow/. Last accessed on June 2017.

[13] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA" In Proceedings of IEEE International Conf. on Application-specific Systems, Architectures, and Processors (ASAP), Toronto, ON, Canada, 2015, pp. 189-193.

[14] Altera SDK for OpenCL Programming Guide, https://www.altera.com/ja_JP/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf. Last accessed on June 2017.

[15] NALLATECH OpenCL A10 BSP Reference Guide, http://www.nallatech.com/store/pcie-accelerator-cards/nallatech-385a-arria10-1150-fpga. Last accessed on June 2017.

[16] Altera product selection guide, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/pt/arria-10-product-table.pdf. Last accessed on June 2017.

[17] A.E. Güzel, V.E. Levent, M. Tosun, M.A. Özkan, T. Akgün, D. Büyükaydın, C. Erbas, H.F. Ugurdag, "Using High-Level Synthesis for Rapid Design of Video Processing Pipes," In Proceedings of East-West Design Test Symposium (EWDTS), Yerevan, Armenia, 2016.

[18] M.A. Özkan, O. Reiche, F. Hannig, and J. Teich, "FPGA-Based Accelerator Design from a Domain-Specific Language," In Proceedings of International Conf. on Field-Programmable Logic and Applications (FPL), Lausanne, Switzerland, 2016.

[19] Artemis JU Project, ALMARVI: Algorithms, Design Methods, and Many-Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing, GA 621439. http://www.almarvi.eu

VITA

Name: Umut Ulutaş

Date of Birth: 13/06/1991

Birth Place: Adıyaman

Languages: Turkish, English

Education:

- MS: Ozyegin University, Computer Sciene 2017
- BS: Ozyegin University, Computer Sciene 2014
- High School: Turk Egitim Vakfi Inanc Turkes Ozel Lisesi - 2010

Work Experience:

- Internship at IT Department of Duzce University
  Summer Internship 2014
- Internship at Credit Europe Bank
  Summer Internship 2012

Publication:

U. Ulutaş, M. Tosun, V.E. Levent, T. Akgün, D. Büyükaydın, H.F. Uğurdağ, ”FPGA Implementation of a Dense Optical Flow Algorithm Using Altera OpenCL SDK”, ICT Innovations Conference, Skopje, Macedonia, 2017.

Honors and Awards

- Honor awarded by Ozyeğin University between 2010 and 2013