TOOLS AND TECHNIQUES FOR IMPLEMENTATION OF REAL-TIME VIDEO PROCESSING ALGORITHMS

A Dissertation

by

Vecdi Emre Levent

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

Doctor of Philosophy

in the
Department of Computer Science

Özyeğin University
September 2018

TOOLS AND TECHNIQUES FOR IMPLEMENTATION OF REAL-TIME VIDEO PROCESSING ALGORITHMS

Approved by:

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

Asst. Prof. Onur Demir
Department of Computer Engineering
Yeditepe University

Prof. Murat Uysal
Department of Electrical and
Electronics Engineering
Özyeğin University

Prof. Nizamettin Aydın
Department of Computer Engineering
Yıldız Technical University

Date Approved: 10 September 2018

Asst. Prof. Furkan Kıraç
Department of Computer Science
Özyeğin University

Liberty and independence are my character

— Mustafa Kemal Atatürk

ABSTRACT

Hardware implementation of video processing algorithms, which are usually real-time by nature, need architectural exploration so that we achieve the required performance with minimal cost. In addition, the video algorithm to be implemented may need to be used with different frames-per-second and resolution in different applications. Hence, we usually need to design a parameterized IP block instead of a fixed design. Also, during the hardware design process, the requirements fed from the algorithms team may change as well as the algorithm itself. As a result of these, hardware implementation iterations need to be as fast as the algorithms development iterations. This is only possible with the use of tools and techniques specifically geared towards hardware design generation for video processing. The tools and techniques discussed in this dissertation include host software, FPGA interface IP, HLS, RTL generation tools, an architectural estimation tool, flow based verification approach, and logic synthesis automation as well as architectural concepts (e.g., nested pipelining). The architectural estimation tool estimates many design metrics. These metrics are area, throughput, latency, DRAM usage, interface bandwidth, temperature, and compilation time. While we explain the above tools and techniques within a specific use case, namely, optical flow, we also present results from another use case, image fusion. Using our methodology and tools, we were able to design and bring up to 11 versions of optical flow and 3 versions of image fusion on 3 different FPGAs from 2 different vendors. The first version of these designs (hence the generators) took several months; however, the subsequent design versions each took a few days with a few people. In the case where only architectural trade-off is needed, we were able to generate and synthesize around one thousand designs in a single day on a 48-core server.

ÖZETÇE

Video işleme algoritmalarının donanım gerçeklemelerinin (doğası gereği çoğunlukla gerçek zamanlı) mimari ödünleşime ihtiyacı vardır. Böylece asgari maliyetle gerekli performansı elde edebiliriz. Ek olarak, gerçeklenecek video algoritmalarının farklı uygulamalarda farklı fps ve çözünürlük ile kullanılması gerekli olabilir. Bu nedenle, genellikle sabit bir tasarım yerine, parametrize bir IP bloğuna ihtiyaç duyarız. Ayrıca, donanım geliştirme sürecinde, algoritma takımı tarafından verilen gereksinimler algoritmanın kendisi gibi değişebilir. Bunların sonucu olarak, donanım geliştirme iterasyonları, algoritma geliştirme iterasyonları kadar hızlı olmalıdır. Bu, ancak özellikle video işleme için donanım tasarım üretimine yönelik araç ve tekniklerin kullanımı ile mümkündür. Bu tezde bahsedilen araç ve teknikler arasında sunucu yazılımı, FPGA arayüz IP'leri, HLS, RTL üretim araçları, bir mimari tahmin aracı, akış tabanlı doğrulama yaklaşımı ve lojik sentez otomasyonunun yanı sıra mimari kavramlar (örneğin. iç içe boruhattı) bulunmaktadır. Mimari tahmin aracı bir çok metriği tahmin etmektedir. Bu metrikler alan, çıktı verme, gecikme, DRAM kullanımı, arayüz bantgenişliği, sıcaklık ve derleme zamanıdır. Yukarıdaki araç ve teknikleri, spesifik bir kullanım durumu olan optik akış üzerinde açıklarken, ayrıca diğer bir kullanım durumu olan görüntü füzyonunun da sonuçlarını sunmaktayız. Metodolojimizi ve araçlarımızı kullanarak, optik akışın 11 versiyonu ve görüntü füzyonun 3 versiyonunu 2 farklı şirketin 3 farklı FPGA'i üzerinde tasarladık ve ayağa kaldırdık. Bu tasarımların ilk versiyonları (RTL üreticisi nedeniyle) birkaç ay aldı, ancak müteakip tasarım versiyonlarının her biri birkaç kişi ile birkaç gün aldı. Mimari ödünleşime ihtiyaç duyulduğu durumlarda, bin civarında tasarımı bir gün içerisinde 48 çekirdekli sunucuda üretip sentezleyebilmekteyiz.

ACKNOWLEDGEMENTS

First and foremost, I want to thank my advisor Assoc. Prof. H. Fatih Uğurdağ, who supported me in every phase of my PhD study. It has been an honor to be his first PhD graduate. I would also like to thank all my dear friends in nEMESys and OKATEM labs at Özyeğin University. I spent 3 years in nEMESys lab and 1 year in OKATEM. I would like to express my gratitude to thesis examination committee members Prof. Murat Uysal, Asst. Prof. Furkan Kıraç, Prof. Nizamettin Aydın, and Asst. Prof. Onur Demir.

This work has been done as part of a project jointly supported by TÜBİTAK ARDEB-EEEAG and European Unions Artemis Joint Undertaking. On TÜBİTAK side, the project was called M-RIVA: Methodology development for Real-time Implementation of Video Algorithms on FPGAs (project no. 114E343). On Artemis side, the project is called ALMARVI: Algorithms, Design Methods, and Many Core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing (project no. 621439).

During the course of this work, I was a full-time research assistant supported by the above mentioned TÜBİTAK project (PI: Assoc. Prof. H. Fatih Ugurdag) between April 2014 and April 2017, and starting with Aug 2017 I have been supported by TÜBİTAK ARDEB-EEEAG 1003 project (PI: Prof. Murat Uysal, co-PI: Assoc. Prof. H. Fatih Ugurdag) named Innovative Optical Wireless Communication Technologies for 5G and Beyond (project no. 215E311). I must acknowledge the tuition waiver I received from Özyeğin University throughout my studies as well as the great research and working environment the university provides.

Last but not least, I would like to thank my family for their never-ending support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

FPGAs are considered for implementation of real-time video processing algorithms at an increasing rate, especially due to their lower power consumption compared to GPUs and multi-core CPUs. However, development times of FPGA designs are normally much longer.

Hardware teams interfacing with algorithm teams, sooner or later, realize that what they have is a moving target. Before they finish the first version of the hardware, the algorithm developers modify the algorithm. If not, they have the expectation that the hardware team (especially if they are working on FPGAs) can also easily create several versions with different performance, form-factor, and cost parameters. After all, both teams realize their implementations through coding, which is usually Matlab/C versus Verilog/VHDL.

There is no way a hardware team can keep up with an algorithms team without a code generation approach. And that is what we did in our case, and as a result we have reached the ability of coming up with a working design of 30 thousand lines of Verilog in 10 days with 2 people. Obviously, even this is possibly slower than what algorithm developers expect. However, such quick turn-around time is much faster then what most hardware teams can sustain.

1.1 Problem Statement

The process of chip development is a quite demanding process, especially when the specifications are not clear, and the algorithm can be updated at any time. It is shown that the development time and effort can be greatly reduced by using some tools and methods specific to video processing algorithms.

It is wise to first discover the hidden parameters of the design problem, i.e., parameters of architectural trade-off. In this case, there is a chicken and egg dilemma. It is not possible to know for sure whether the selected parameters are suitable or not before making some progress with the design. For this, it is necessary to be able to predict the final state of the design with the desired parameters before the design finishes. We have to solve this problem with the architectural estimation tool. The tool should identify and generate the most suitable design according to the constraints given.

In order to make these estimations, many designs have to be produced and synthesized in advance. The estimator created is based on previous synthesized designs. Accuracy test of the estimator is performed with the synthesis results which are not used in creation of estimator. Too many synthesis results are needed for creating estimation tool. However, it is very difficult to perform these syntheses manually. Depending on the selected parameters, a tool is required to perform thousands of syntheses at the same time and parse the generated reports. Batch synthesis tool solves this problem. The batch synthesis tool feeds the inputs that the estimation tool needs.

Design units such as FPUs that work in multiple clock cycles are also needed. These units can be produced according to different latency and cycle-time. However, it is not possible to know which one is best for design without generating these units. Since each unit will take a long time to be produced and synthesized, it should not be done manually. A tool has been developed that analyzes the FPUs according to the selected FPGA model, latency, and cycle time interval, then gives the result with the lowest area-time product. Area-time multiplication is a parameter that must be considered if the highest output speed is desired.

Another very important approach is the code generator. While a design is being implemented, trying to design parametrizations in RTL can make things very complicated. When you need a parametric design, we prefer script languages like Perl/Python. Non-parametric fixed RTL designs are produced from these languages. However, the reason why these designs are parametrized is that the code generator produces according to the parameter it takes. This approach saves lives a lot of time. Manual design of thousands of line designs is both time-consuming and will increase the likelihood of producing buggy designs. In this approach, the length of the code to be produced is not an issue for the designer.

It is very useful to use High Level Synthesis (HLS) tools where the algorithm can change over time. Because every algorithm changes, the scheduling is broken and everything has to be designed from scratch. Since the HLS tools get input from high-level languages such as C, the changes in the algorithm can be done directly on the C code.

Advanced architectural design techniques are needed in video processing algorithms that require high throughput. The throughput of output can be increased with the pipeline mechanisms that are nested inside. How this can be used to design interlaced pipeline mechanisms is described in chapter 4. 3 pipeline based on frame, line and pixel are run in parallel.

Verification of FPGA designs for video processing algorithms is difficult in large designs. Each unit in the design is made a lot of calculations and pixel transfers to other modules. It is difficult to verify both the calculations and the transfer of pixels between modules. When it is attempted to be verified together, it is very difficult to determine whether the problem is due to flow or calculation units. Therefore, once the pixel transfers between the modules have been verified, the calculations within the modules themselves must be checked. In this context, a tool was developed to verify movements. Details are found in the Flow Verification Section 4.

Developing from scratch for all needs can be time consuming. It is very useful to use a variety of previously verified video processing sub-blocks for this. In this thesis, some modules which are frequently used in video processing algorithms are presented as IP. With this approach, development time is greatly reduced.

Finally, the video processing design needs to communicate with the outside of chip. Various interfaces need to be designed for communication. However, the design of these interfaces can sometimes be more complex than the design of the video processing algorithm. Especially when interfaces such as PCIe or Ethernet are selected, development time is very long. Various interfaces suitable for video streams have been developed for this. It is easily usable for another video processing algorithms.

The above mentioned problems are found in many video processing algorithms. We have tested the tools and methodologies we have developed to solve these problems in order to realize the FPGA design of the optical flow algorithm. Approximately 1 year field development processes have gone down to 1 week. The number of people needed in the team has also decreased considerably.

1.2 Scope and Contributions

Since the video processing algorithms can be huge and complex, various tools and techniques need to be used when designing them. Our tools and methodologies are within the scope of:

- Scalability
- Extensibility
- Reusability
- Reduction of development time
- Reduction of verification time
- High speed interfaces
- Proven IPs

- Architectural exploration

The above mentioned items contribute to the FPGA design flow in the designated areas of Fig. 1.

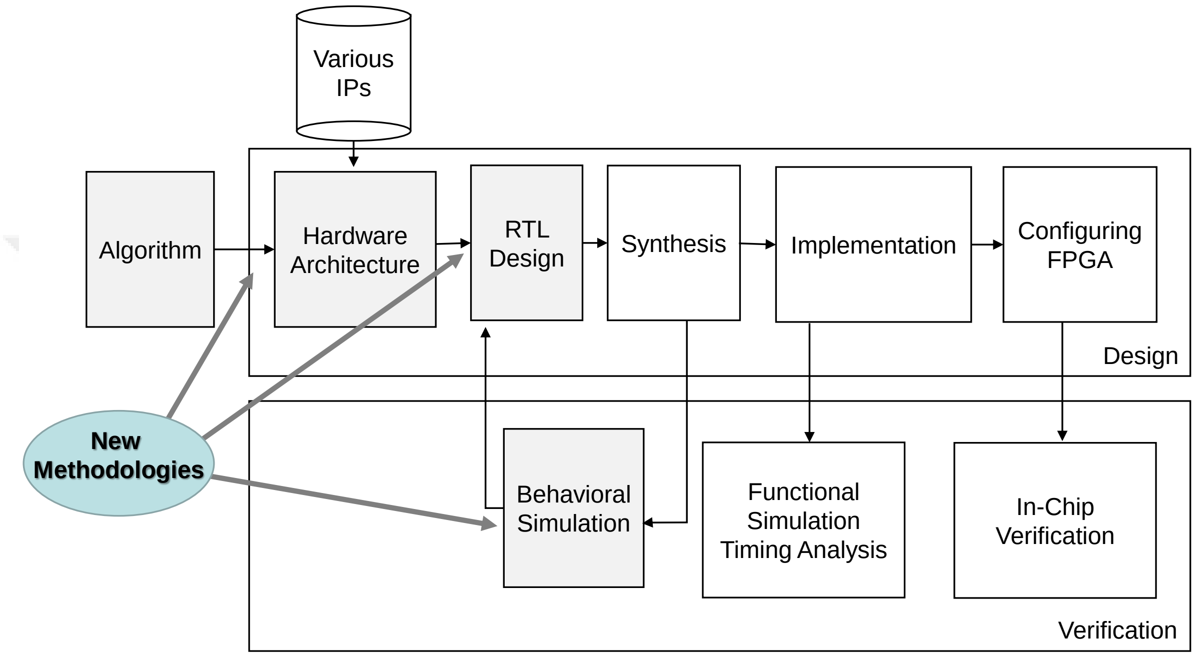

Figure 1: Contributions to FPGA Design Flow.

1.3 Related Work

There is a productivity gap between algorithm development and hardware development. Or more accurately, there is a productivity gap between development of simulation models in software and their efficient real-time implementation on custom hardware platforms. HLS has been looked at as a panacea to this problem. However, HLS is good at synthesizing "basic blocks" and loops around them. On the other hand, a video processing algorithm is not simply a loop around a single basic block.

There are usually multiple loops in a video processing algorithm, some of which are nested. There are also interface design issues, i.e., PCIe or USB for streaming video in and out, DRAM for previous frames' storage. Besides HLS, design generation techniques need to be used to quickly produce HDL for designer-conceived pipelines. Speeding up the implementation process requires not only speeding up the design process, but also the architectural design, verification, and logic synthesis.

This work is the only work that encompasses all these dimensions although some of these dimensions have been approached specifically for video processing, i.e., streaming applications. Actually, everything done within the score of this work can apply to applications outside video processing and even applications that are not streaming. The reason our claim is in the context of video processing is because the use cases we worked on were video processing applications.

Since there is no comparable work in the literature, in the rest of this section, we will cover the literature within the context of the dimensions listed above.

HLS is possibly the most important step in our design flow, and we designed our own HLS tool, a tool which especially fits video processing applications although it is yet general-purpose. However, our work builds on what has been done in the literature in the past. That is why we would like to discuss the HLS history and literature.

In the early days of VLSI design (1970s), designers went all the way from algorithms to layout manually. Tediousness of hand-drawing of layout resulted in layout generating software. Later, the VLSI design community started raising the level of abstraction. Eventually by 1979, “silicon compilers” became a goal. What was meant by a silicon compiler was a tool that could take in a behavioral description in a programming language and output layout with no intervention. Soon (by 1981) it was clear that this was a utopia and that rather a “layered approac” was needed, where point tools would have to be used in a sequence (if needed in multiple design iterations) with intervention at cleanly defined design flow interfaces. There were, by the way, already efforts in the direction of a layered approach [1]. The first layer in this new layered approach would take in a high-level behavioral (i.e., functional) description of the design. It would then directly output gates or output RTL, which would then go into a logic synthesis tool. The design flow would continue with physical design automation tools. By 1981, researchers called this most front-end step in their layered approach architectural synthesis [2] or data path synthesis [3]. Starting around 1984, these two terms were mostly replaced by the term HLS. In 1985, people started calling it behavioral synthesis. Since then both terms, HLS and behavioral synthesis, survived, with HLS being used more commonly in the academia.

Roughly, 1985 to 1995 saw a huge interest in HLS in the academia. The euphoria in the academia resulted in commercial HLS tool releases in the industry as well as some in-house tools [4] by 1995. Rise and fall of HLS in the industry was quick. In about 10 years at the most (by mid 2000s), it was clear that HLS was not going anywhere. It was very surprising, because its sister, logic synthesis, had become a commodity tool long ago.

Starting around the same time (mid 2000s), HLS were seen as a means to make FPGAs easily adoptable into applications with few or no hardware experts in the team. And since 2011-12, HLS has been successfully used by the FPGA community and its users. HLS is again in a euphoric state; however, this time industry success has spawned academic interest, whereas before it was the (presumed) academic success spawned industry interest.

Why has HLS failed first, in the years 1980-2005, but then seems to be succeeding since 2011? As in many markets, a complete answer requires the grasp of the full history of how things progressed, which can be found in survey articles. We recommend [5] for a survey relatively recent work (mostly since 2000). For a more complete but older survey, please see [6].

We have a relatively simple answer to the above question. In the first phase of HLS (1980-2005), we all expected too much from HLS. We wanted it to be general-purpose, work for the design of commodity ICs, and produce results that even beat expert designers. Imagine a software that plays chess, checkers, and backgammon and beats world champions in all three. And imagine a problem that is even harder than that, because in the case of HLS, it was not even clear what it was designed for in the first phase of its history. Yes, the same is possible for software compilers, but then, they are not designed to produce real-time implementations, and they do not beat a human assembly coder in terms of code footprint.

How come HLS appears to be successful now? That is because the FPGA community expects only "acceleration" from it, and the execution model is more well-defined than before [6]. Or as in the case of synthesis from OpenCL [7], it is "domain specific".

Having said that, we claim for HLS to be successful for commodity IC designs and/or for truly high throughput systems (i.e., for beyond acceleration), it cannot be a one size fits all type of tool. We look at it in the spirit of "semi-automation" in terms of the bigger picture [8]. Even then, it may have to be domain specific for truly high-throughput systems. There is a plethora of work on domain specific languages in the software domain [e.g., [9]]; and thankfully, we are also seeing an influx of work on domain specific HLS such as [10][11]. Furthermore, the problem of HLS problem becomes more tractable within a particular design framework.

A framework is a flow or methodology, which usually comes up with a top-level design as well as libraries. Framework based approach is common in hardware verification but not as much in hardware design itself so far. There are several frameworks in the literature and some commercial offerings for improving hardware design productivity. Since this is a vast problem, such works are usually domain or design specific (e.g., [12, 13, 14, 15, 16]). The ones that are general-purpose usually aim "acceleration" (e.g., [17, 18, 19, 20]).

In acceleration, the goal is to obtain a run-time speed-up (most often single digit) over the pure software implementation and not necessarily a real-time implementation. Acceleration oriented frameworks target ease of programmability, and as a result, they are software-centric. The design is viewed and done as a software running on one or more cores with some software functions that have hardware bindings. Usually, all hardware bindings have to communicate over a processor subsystem, and that is what usually slows down the system, but yet that is what makes the system general-purpose. If the algorithm is coded in an SIMD paradigm with parallelism explicitly stated (as is the case in OpenCL), much higher speed-ups are possible [21, 22, 23, 24, 25]. However, not every application is amicable to an SIMD programming paradigm.

The purpose of this work was to devise tools and techniques that can benefit a variety of designs, especially video processing designs. The dissertation discusses a specific use case, namely, an optical flow algorithm, and it explains the tools and techniques developed in the context of optical flow, although they are quite general-purpose in their scope.

Our approach can be evaluated in the context of "semi-automation" [8]. It is tailored to designs where the required speed-up (double or triple digits) can only be achieved through careful and creative hardware-architecting and implementation. The idea is to answer the question "what tools and techniques can we use to both achieve the extreme throughputs necessary while operating at design productivity levels close to software". Such high goals can at most be met with methods that are partly domain specific (if not design specific). Having said that, we only claim that tools and techniques presented here can be used in video processing hardware designs and can be evaluated for other domains. Even in video processing domain, the possible contexts and requirements are many. [26, 27] address some of those contexts.

[26] proposes an image video processing platform based on Xilinx's MicroBlaze soft processor. The focus in [26] is on the hardwired pixel processing pipeline, which is synthesized by Synfora's PICO HLS tool, where the MicroBlaze simply initializes and controls the pipeline. Work [27] builds on to the concept of "weakly-programmable

architectures" and enables early verification of the applications correctness as well as its performance figures. It also offers automated assembly of the final hardware design. We have an overlap in terms of advocating use of HLS in the pixel processing pipeline. When it comes to [27], we deal with early verification of correctness and throughput as well as automated assembly of the pixel processing pipeline, However, we do not use an architecture based on weakly-programmable architectures. Higher speed-ups possible at the expense of lower design productivity.

Besides the above-mentioned incorporation of HLS, early verification, throughput estimation, automated pipeline assembly, in this article, we present tools and techniques in the context of host software, FPGA interface IP (PCIe, USB 3.0, DRAM), synthesis automation as well as architectural concepts (e.g., 4 levels of nested pipelining), and an architectural estimation tool. With all of these at our disposal, we were able to get 11 versions of an optical flow algorithm running on 3 different FPGAs (from 2 different vendors), while we generated and synthesized several thousand designs for architectural trade-off. Although we keep mentioning FPGA as the target platform, most aspects of our approach equally apply to front-end development stage (i.e., RTL development and core generation/integration) of ASIC/SoC implementations.

There are studies in the literature to estimate the resource usage on the FPGA [28, 29, 30, 31, 32, 33]. In Schumacher's work [28], it was emphasized that it would take a long time to fully synthesize in every design iteration. The main goal of this work is to be able to give the designer a very fast feedback while making decisions about the design. Fig. 2 shows the flow diagram of the Resource Estimation tool of Xilinx.

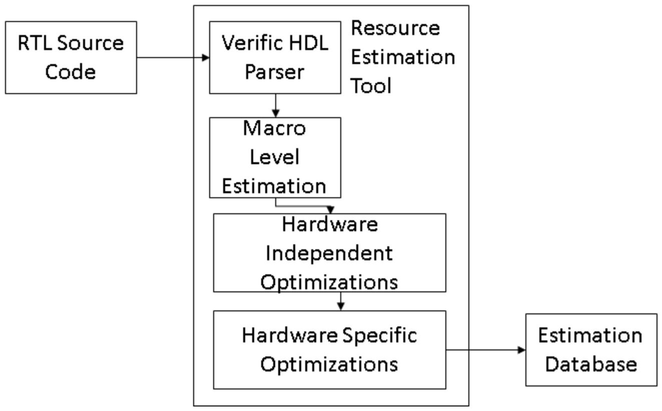

Figure 2: Xilinx Resource Estimation Tool.

The first stage of the developed tool parses the given RTL code using the design automation tool named Verific. Verific's library provides bitwise/binary logic, unary logic, arithmetic elements, comparators, multiplexers, shifters, and storage elements from the parsed code. The elements found are multiplied by coefficients according to element bit widths. Hardware-independent and hardware-specific optimizations must also be included in the calculations. The synthesis tool can do these things, and the results can change quite a bit. Hardware-independent optimizations are optimizations that can be done for all FPGAs. Hardware-independent optimizations are modeled by resource sharing, comparator with constants, dangling/disabled logic, and ROM implementations. Hardware-specific optimizations are optimized according to the resources in the target FPGA. This optimization is achieved by modeling elements such as block RAM, multiplier/arithmetic DSPs, shift registers, register controls, multiplexer trees, and binary logic tree's. Tests were conducted targeting Spartan-3, Virtex-4, and Virtex-5 devices. The mean errors were 21.9%, 26.3%, and 14.2%, respectively. The execution time of the tool was seen to be 60 times faster than Xilinx's synthesis tool. It completes all operations within an average of 19 seconds.

1.4 Outline of the Dissertation

The organization of the dissertation is as follows. Section 2 explains optical flow as well as the particular algorithm and its requirements that caused us to develop the tools and techniques outlined in this dissertation. Section 3 details our hardware implementation (including nested pipelining) from an architectural point of view without going into the details of the tools. Section 4 goes into the details of the tools and techniques that give us our impressive design productivity and explains how they can be used in other video processing designs. Section 5 provides information on the architectural estimation tool. Section 6 shares some design metrics from optical flow as well as another design, i.e., “image fusion”. And Section 7 wraps up the dissertation and talks about possible future work.

CHAPTER II

OPTICAL FLOW ALGORITHM

Estimating the motion, direction, and magnitude, of an object is a necessary part of some video processing algorithms. Commonly used motion estimation algorithms are listed below.

- Block-matching algorithm
- Phase correlation and frequency domain methods
- Pixel recursive algorithms
- Optical flow

These algorithms try to estimate motion with different calculation methods. Each algorithm has different motion estimation performance. In this chapter, the performance of the motion estimation algorithms, the optical flow algorithm chosen to be implemented and the effect of the parameters of the iter and warp number change in this algorithm on performance is shown.

2.1 Motion Estimation

There are many studies on motion estimation. The most commonly used methods in the literature are block-matching and optical flow algorithms for motion estimation. In the study performed by Boer and Kalksma, the quality and computation times of the optical flow and block-matching algorithms used were compared when measuring the position changing for UAVs [34]. When the image resolution grows, results show that optical flow computation is significantly faster than block matching algorithms. The calculation quality varies according to the input dataset. The block-matching algorithm gave the worst performance on dotted floor surface data. This is because the brightness values of most pixels are close to each other in the dotted floor surface. In 4x4 and 8x8 pixel-binned datasets, the optical flow is more successful, but the success of block matching in the 16x16 binned dataset seems to increase. Golemati et al. performed motion analysis on carotid artery wall ultrasound images [35]. Optical flow and block-matching algorithms are compared in terms of total displacement. In non-noise data, block-matching is more successful than optical flow. However, the success of optical flow is closer to block-matching. In other datasets, it is obvious that the optical flow algorithm is more successful. Philip et al. made comparisons for low resolution data [36]. The block-matching algorithm is better than the computational cost, but it shows that optical flow improves the motion quality better. Liu et al. performed motion analysis on ultrasound images taken to evaluate cardiac functions [37]. They applied shear strain and rotation on the data they used. While the optical flow algorithm is not much affected by these operations, the performance of the block-matching algorithm is reduced. The block-matching algorithm can be used for works where the calculation quality is not very important and there is a low computational cost requirement. Our work requires as high a resolution, a high quality as possible. Optic flow is better than block-matching according to these requirements. Therefore, we used the optical flow algorithm in our work.

2.2 Optical Flow

Optical Flow algorithms are a class of algorithms used to extract motion vectors from consecutive frames on a pixel by pixel basis. There are also block based motion estimation methods, which offer lower computational complexity at the expense of lower motion vector resolution [38, 39, 40, 41]. Motion vectors of video frames are used in video compression, frame rate conversion, machine vision, and video analytics. Sometimes motion vectors are part of the features we are looking for, and sometimes they are used to track objects or reduce bitstreams (i.e., compression) or reduce processing time (by limiting us to regions of interest).

Our team evaluated a number of optical flow algorithms benchmarked at [42, 43]. We picked "Anisotropic Huber-L1 Optical Flow" (AHL1) algorithm [44], as it was one of the fastest algorithms and was amicable to GPU parallelization. Then, the GPU implementation discussed in [45] was created, which is based on the Matlab code associated with [44].

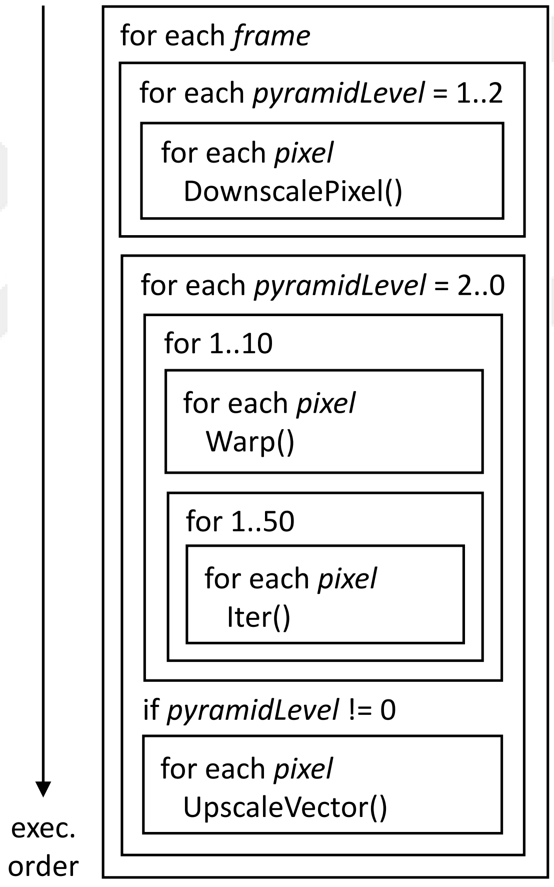

Figure 3: Nested Loop Structures of AHL1 Optical Flow.

Fig. 3 shows the structure of the optical flow algorithm that we had to implement. It is a highly iterative algorithm. Every rectangle is a loop in Fig. 3. The frame and pixel loops are typically found in almost every video processing algorithm. This algorithm, in addition, creates a number of downscaled versions (2 in our implementation) of every input frame (i.e., DownscalePixel() in Fig. 3). With the frame itself (i.e., frame_0), we have 3 versions of it (frame_0, frame_1, frame_2). These frames are thought of as different levels in a pyramid, with frame_0 occupying the base, and with frame_1 and frame_2 occupying higher and smaller cross sections in that order. Since cross sections of a pyramid parallel to its base are geometrically scaled down versions of its base, a pyramid analogy is used. If the downscale ratio is r and frame_0 has P pixels, then frame_1 has P/r^2 pixels, and frame_2 has P/r^4 pixels.

AHL1 applies its iterative math on frame_2 (the smallest downscaled frame) first. Then, it takes the vectors found from frame_2 and upscales them to match the frame_1 (i.e., UpscaleVector() in Fig. 3). It starts the iterative motion vector computation process of frame_1 (i.e., for 1..10 loop and its nested loops in Fig. 3) with the vectors computed (i.e., upscaled) from frame_2 as the initial values. Then, it applies the final vectors from frame_1 to frame_0 as initial vectors. Note that AHL1 algorithm can only find a few pixels of motion (i.e., displacement) vector for each pixel. However, applying it to downscaled versions of the input frame (and hence the previous frame) not only speeds up the optimization but also allows us to compute large vectors at the resolution of the original frame.

In Fig. 3, at every pyramid level, the body of the iterative algorithm (i.e., Iter()) is applied to a frame 50 times. After 50 iterations, the vectors found are taken as initial values for the next 50 iterations, just like the case between pyramid levels. As for applying the initial values, actually the frame is "warped" (i.e., Warp()) by the "sum" of motion vectors found so far. That is, the initial motion vector is zero but at each batch of iterations (50 Iter() calls) the input frame is not the same as the original one, instead it is a warped frame. Note that Iter() is a quite heavy code fragment with 86 floating-point operations and some integer operations. More specifically, it has 30 floating-point multiplications, 23 subtractions, 16 additions, 9 divisions, 5 comparisons, 2 square-roots, and 1 absolute value operation.

Fig. 3 shows the maximal version of AHL1 implemented by us, with 3 pyramid levels, 50 Iter() per warp, and 10 warps (i.e., a total of 500 iterations per level). All these parameters are adjustable in our generators. 500 iterations is for high-accuracy use-cases. These parameter values were also used by [44] for a use-case of restoring severely degraded historic video material via an optical flow-based interpolation. Another use-case for a large total of iterations per unit time is when the same hardware is shared for multiple video streams.

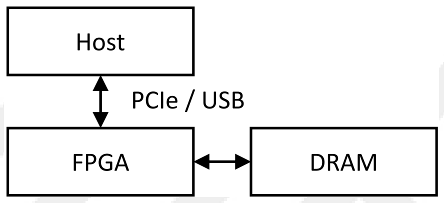

Figure 4: Top Level of The System.

We will not go into the mathematical details of the algorithm as they are not critical in carrying out the hardware implementation. What is most important is the structure of the algorithm, the data and control dependencies among its substructures, and its computational complexity versus its throughput requirements (pixels processed per second). Just like how a compiler does not need to understand the reasoning behind a code it compiles but instead it needs to understand its flow and dependencies, that is all we need to implement optical flow and most algorithms.

2.3 Benchmark Results

We evaluated the performance of the TV-L1 optical flow algorithm we used. For this we fed two consecutive video frames to the algorithm in MATLAB. The ground truths of these video frames are also available. The results produced by Matlab are compared with the results that should be. Vector size and angular errors are listed. In Table 1, errors are given according to the vector dimensions. The Fig. 5 created according to this table.

Table 1: Optical Flow Vector Size Errors.

W./I.	1	2	5	10	25	50	100	250	500
1	109.05	114.28	125.87	129.64	132.05	118.95	120.98	118.45	118.46
2	#N/A	114.27	#N/A	144.87	#N/A	159.35	149.84	149.50	147.73
5	#N/A	#N/A	134.11	157.46	172.19	160.93	110.96	72.49	70.86
10	#N/A	#N/A	#N/A	158.78	#N/A	142.13	83.91	62.14	51.92

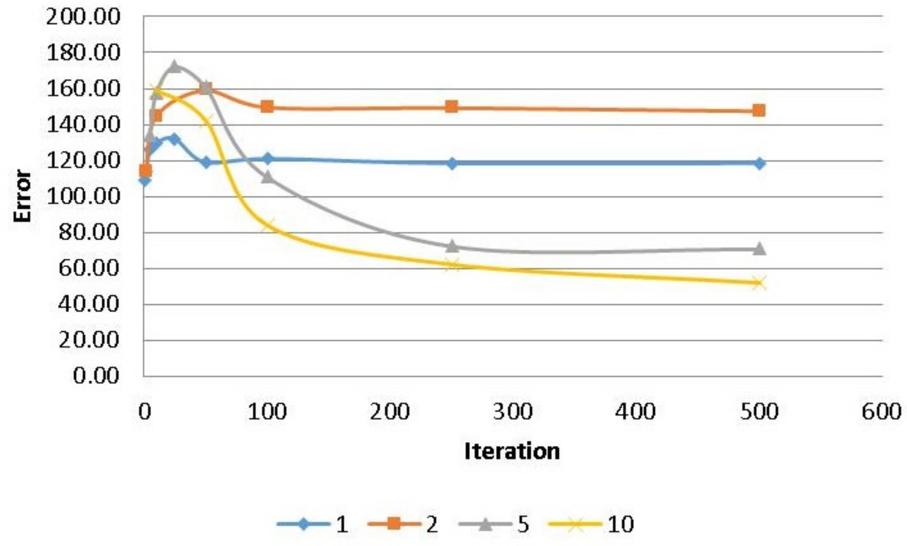

Figure 5: Vector Length Benchmark.

In Table 2, errors are given according to the angular errors. The Fig. 6 created according to this table.

Table 2: Optical Flow Angular Errors.

W./I.	1	2	5	10	25	50	100	250	500
1	62.32	53.43	37.89	27.35	18.00	13.24	11.78	12.01	12.10
2	#N/A	53.32	#N/A	26.53	#N/A	11.57	9.89	8.71	8.64
5	#N/A	#N/A	37.18	26.16	15.94	11.01	8.93	7.04	6.90
10	#N/A	#N/A	#N/A	25.80	#N/A	10.68	9.09	7.01	6.81

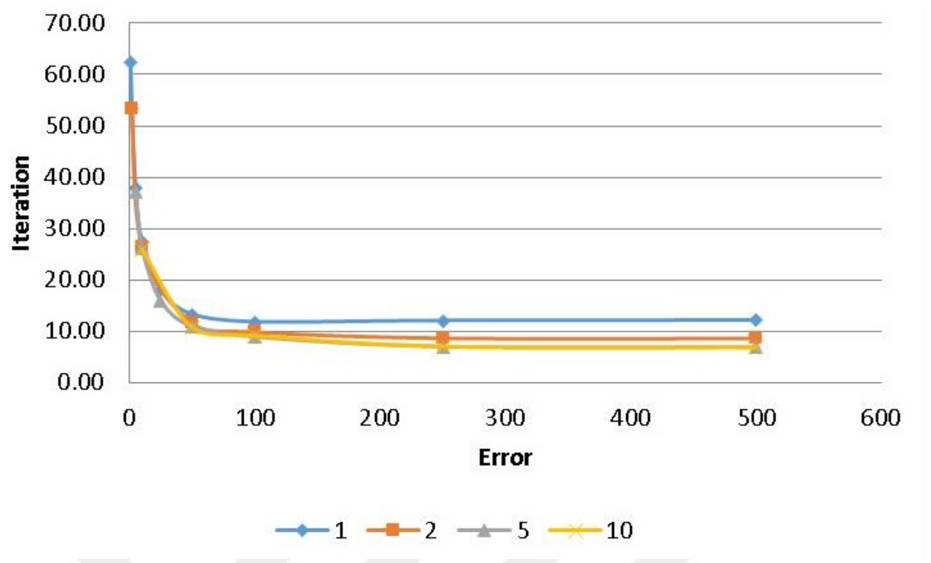

Figure 6: Angle Error Benchmark.

CHAPTER III

OPTICAL FLOW HARDWARE

The top-level of our overall system is given in Fig. 4. Unlike the acceleration oriented frameworks mentioned in Section 1, our framework does not revolve around a host processor. Although in a previous project [46] we implemented Ethernet and HDMI interfaces, in this project we used a host processor as our video source and sink. That is to say that our Host simply supplies the input video stream and collects the output vector stream and does not have any supervisory role over the FPGA. The focus in this work is on the FPGA.

Subsection 3.1 explains the top-level schedule and hence the top-level of our nested pipeline (i.e., frame-level pipeline). Subsections 3.2 through 3.5 give the details of our FPGA architecture. They also explain the lower levels of our nested pipelines. Finally, Subsection 3.6 goes over the interplay of the host software with the FPGA.

3.1 Frame Pipelining

Fig. 7 shows the top-level schedule, hence, the frame-level schedule. Pipelining implies “overlapping”, and our hardware design overlaps frames. That is, the processing of a new frame starts before the processing of the current frame finishes, which is indicated by, for example, “Vin f2” starting before “Vout f1” finishing in Fig. 7(b). On the other hand, Fig. 7(a) shows the schedule of a straight-forward sequential software implementation of the same set of blocks. The hardware offers a speed-up over the sequential implementation due to its frame pipelining. Unlike the figure, the hardware versions of blocks may be shorter, besides their overlaps, hence introducing a further speed-up. Also the Vin and Vout blocks used for the hardware schedule are a little longer than the software schedule. The reason for this is the time to copy video frames to the hardware DRAM.

The loop starting with the line "for each *pyramidLevel* = 2..0" in Fig. 3 (with Warp(), Iter(), and UpscaleVector() inside) is shown as "Iters" in Fig. 7. Iters of frame 1, for example, is shown as Iters f1. On the other hand, the loop starting with the line "for each *pyramidLevel* = 1..2" in Fig. 3 (with DownscalePixel() inside) is shown as "DS" in Fig. 7. Vin and Vout indicate "Video in" and "Video out", respectively. Every frame has to be input (i.e., streamed) to the FPGA by the Host from a camera or storage over PCIe/USB (see Fig. 4). Then, over the same interface motion vectors should be streamed out back to the Host on a per-pixel basis.

We shaded the blocks of odd-numbered frames (i.e., f1, f3, and so on) in order to make the figure easily readable. We shifted Iters f2 and Iters f4 to the right a little bit, because those Iters overlap with their consecutive Iters, and when they horizontally line up with the Iters they overlap with, the readability of the figure suffers. The overlap between Iters is not only because Iters is implemented with a pipeline of Iter blocks but also because an Iter block is internally pipelined. That is, an Iter block can accept a new pixel before it outputs the previous pixel. The amount of overlap between two consecutive Iters (e.g., Iters f1 and Iters f2) is equal to $I \cdot L - C$, where I is the number of Iter blocks, while L and C are respectively the Latency and Cycle-time of an Iter block.

Vin is a process that is carried out by the Host, FPGA, and a PCIe or USB interface bus between them (see Fig. 4). Host runs a software thread to stream a video frame over PCIe/USB. The FPGA absorbs it in its corresponding asynchronous FIFOs and immediately writes it into bigger circular buffers in the DRAM i.e., the DCBs in Fig. 9. While Fig. 7 shows the frame pipelining along the time axis, Fig. 9 shows it at hardware level. DS and Iters involve the FPGA and DRAM. Vout is just like Vin, and it is in the reverse direction, i.e., from the DRAM to the FPGA to the Host. Although frame-level pipelining results in a repeatable schedule, the schedule is not hard-coded. It is instead a result of the interaction among the blocks through asynchronous FIFOs. The schedule is determined by the slowest block in the pipeline, i.e., Iters Pipe.

The top-level schedule may not be perfectly periodic at the microscopic level, especially in the first few frames, but that is acceptable, because the DCBs act like FIFOs and make the blocks work with each other in synchrony.

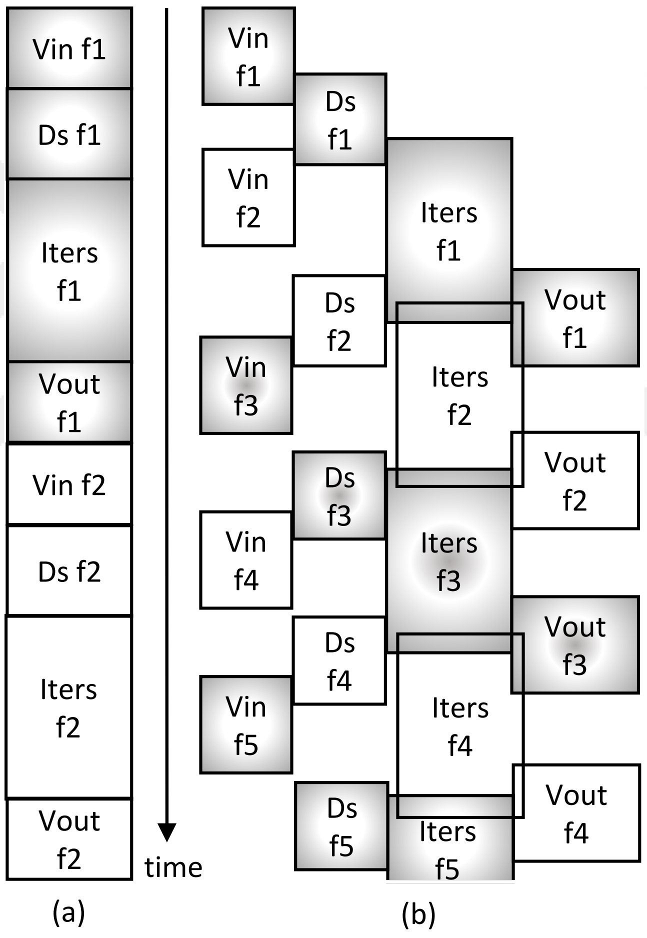

Figure 7: (a) Software versus (b) Hardware Schedule of Frames.

3.2 FPGA Architecture

Fig. 8 shows the top level of the FPGA in our optical flow hardware design, strictly from a structural (i.e., circuit) point of view.

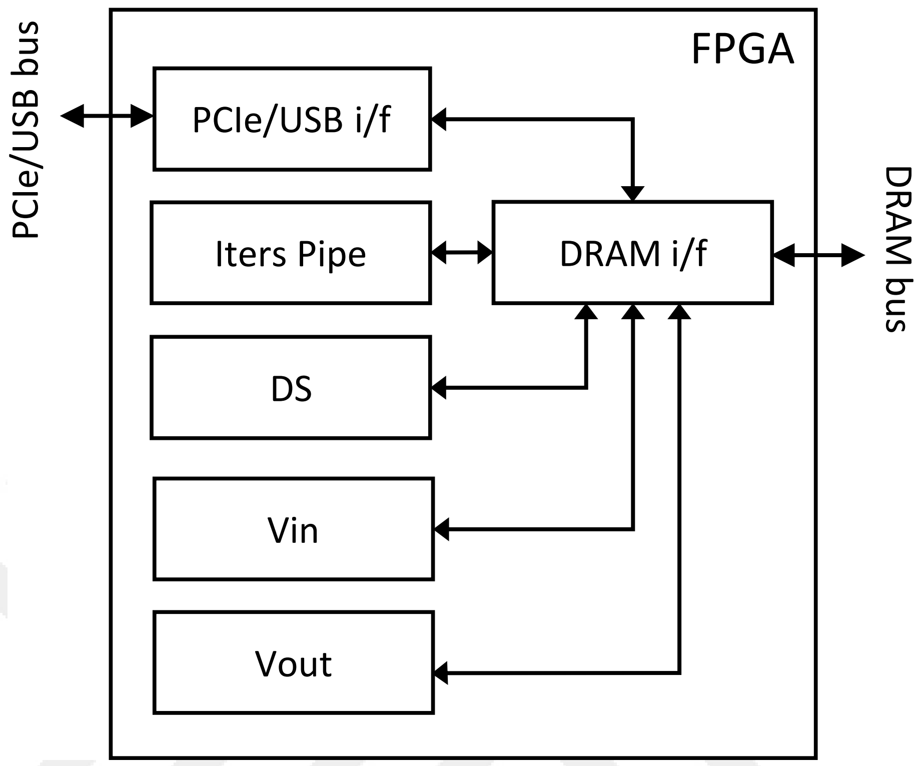

Figure 8: FPGA's Top Level.

The blocks in Fig.s 8 and 9 have one-to-one correspondence except DRAM i/f block. The DRAM and associated interface (i/f) logic in Fig. 8 turns into the multiple DRAM Circular Buffers (DCBs) of Fig. 9.

The DRAM emulates concurrent DCBs through Time Division Multiplexing (TDM), that is, the DRAM i/f serves DCBs with a particular priority scheme, but at such a speed that, it looks as if they are operating concurrently. The DCBs with the associated read and write pointers behave like FIFOs. While a regular FIFO has one read and one write pointer, the top DCB in Fig. 9 (stores the original frames) has one write pointer (kept by Vin) and two read pointers (kept by DS and Iters Pipe). The middle DCB keeps the frames that are downscaled once and has one write pointer (kept by DS) and two read pointers (kept by DS and Iters Pipe). The bottom DCB keeps the frames that are downscaled twice and has one write pointer (kept by DS) and one read pointer (kept by Iters Pipe). One other difference between a FIFO and our DCB is that the DCB allows the read pointer advance only after a certain amount of data arrives to it.

All DCBs keep 10 frames. This buffer size has been experimentally determined with the goal of being able to smooth out DRAM traffic. On the other hand, we buffer two full frames of the original video stream (i.e., 0 downscaled) and its 1-time downscaled version as well as one one full frame of the 2-times downscaled video in the circular buffers, before Iters Pipe starts functioning. These numbers come from data dependencies. Also note that Iters Pipe processes the downscaled frames (0 downscaled, 1-time downscaled, 2-times downscaled) in the reverse order (2-times downscaled, 1-time downscaled, 0 downscaled) as also indicated by the order of indices in the *pyramidLevel* for-loops in Fig. 3 (1..2 versus 2..0).

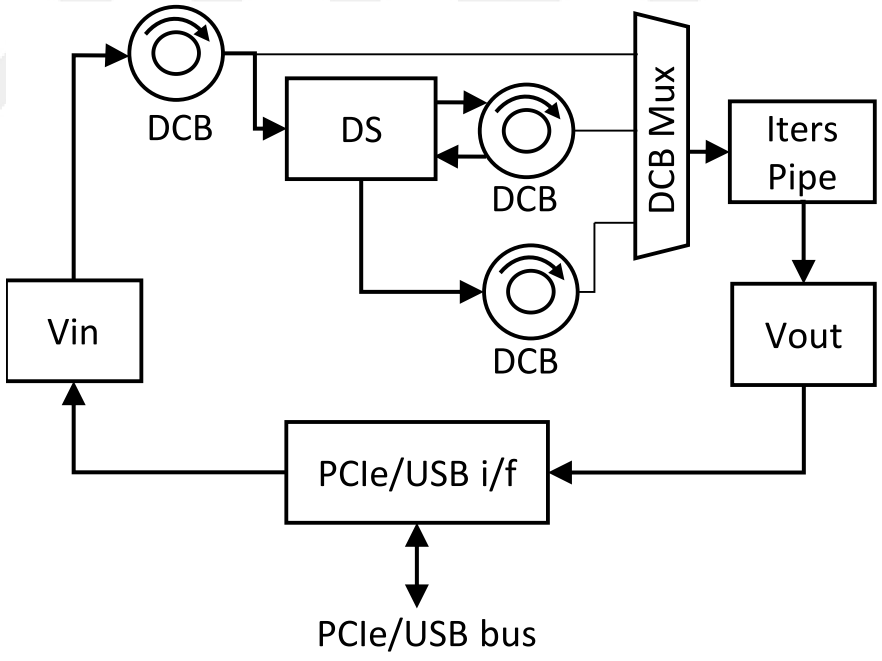

Figure 9: FPGA's Top with DRAM Circular Buffers (DCBs).

Iters Pipe in Fig.s 8 and 9 is, by far, the most sophisticated block in our design. It implements Iters of Fig. 7. Since it has 3 nested pipelines (Iter level, pixel level, and arithmetic unit level), we call it "Iters Pipe". Fig. 10 shows the internals of Iters

Pipe.

The Iters Pipe drawn in Fig. 10 has 2 Iter() per 1 Warp() call as opposed to the 50 in Fig. 3. The 10 Warp() calls in Fig. 3 can be in this particular design any multiple of 2, as the design has the capability to circle back on itself. For example, if we want to warp 10 times as in Fig. 3, pixels go through the pipeline 5 times. DCBmux (shown in Fig.s 9 and 10) streams in the frame to be processed from DRAM by feeding pixels into Iters Pipe one by one. Unless the frame resolution is very low, pixels start emerging from the output of the pipeline before DCBmux finishes streaming in the pixels of the input frame to be processed. Hence, the output of the pipeline has to be buffered in DRAM in a DCB. As soon as streaming of the input pixels is finished, the buffered output pixels are streamed in through the 2:1 Mux when it selects its input coming from the DCB. Note that DCBmux and Vout are drawn with dashed lines in Fig. 10 as they are not part of Iters Pipe.

US in Fig. 10 represents the Upscaler. As explained earlier, we go from the higher levels of the pyramid towards the bottom. As we do this, we need to map the vectors we computed from low-resolution versions of the frame to higher resolutions. In order to do that, we need an upscaler (US). However, the US needs to function only at the end of a pyramid level. If we have to circle through Iter Pipe multiple times for one pyramid level, then US is bypassed through the Mux in front of it except for the last run through the pipeline.

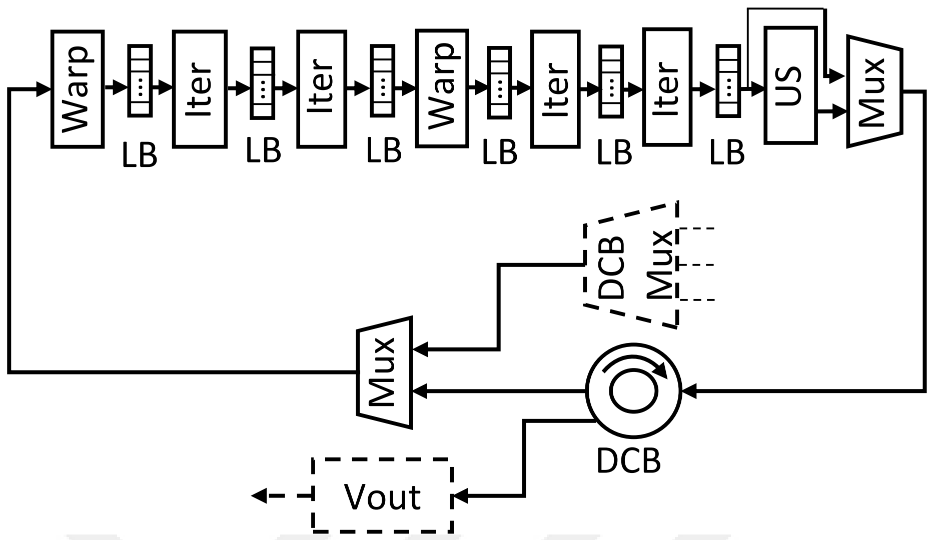

Figure 10: Iters Pipe Block with Line Buffers (LBs).

3.3 Iter, Pixel, and Arithmetic Pipelining

Fig. 11 shows how Iter calls of pixels are pipelined, hence overlapped in time, given a representative frame of 8 pixels. The left hand side of Fig. 11 shows sequential execution and corresponds exactly to the order of computations in Fig. 3. There, we execute the first Iter over all pixels of a frame, one pixel at a time, then it executes the second Iter over all pixels of the same frame, then the third Iter, and so on. The right hand side of Fig. 11, on the other hand, shows how we do the same processing in Iters Pipe. A new pixel's Iter is started before the current pixel is completed. Every upright rectangle in Fig. 11 corresponds to an Iter call on one pixel from its launch time to its completion time. In other words, the height of the rectangle is "pixel latency". The time difference between the launch time of pixels is, however, "pixel cycle-time" and is one of the most critical parameters of the design in terms of the throughput, namely, frames per second (fps). Iter Pipe is "stallable" though, that is, if pixels arrive slower than cycle-time and also arrive with non-periodic timing, it still works. On the other hand, pixel streaming over PCIe/USB and DRAM buses are done in burst mode for efficiency reasons with bubbles in between.

The first Iter block on the left in Fig. 10 does the computations marked with Iter 1 in Fig. 11, while the Iter block to its right in 10 does the computations marked with Iter 2. We had stated that these blocks in Fig. 10 (Iters Pipe) form a pipeline, and indeed, they form a pipeline as can be seen in Fig. 11, because they overlap in time. Therefore, we have shown here that we both have an Iter as well as a pixel pipeline. In addition to Iter-level and pixel-level pipelining, the Floating Point arithmetic Units (FPUs) are pipelined. Hence, with the outer frame-level pipeline, we have 4 nested pipelines.

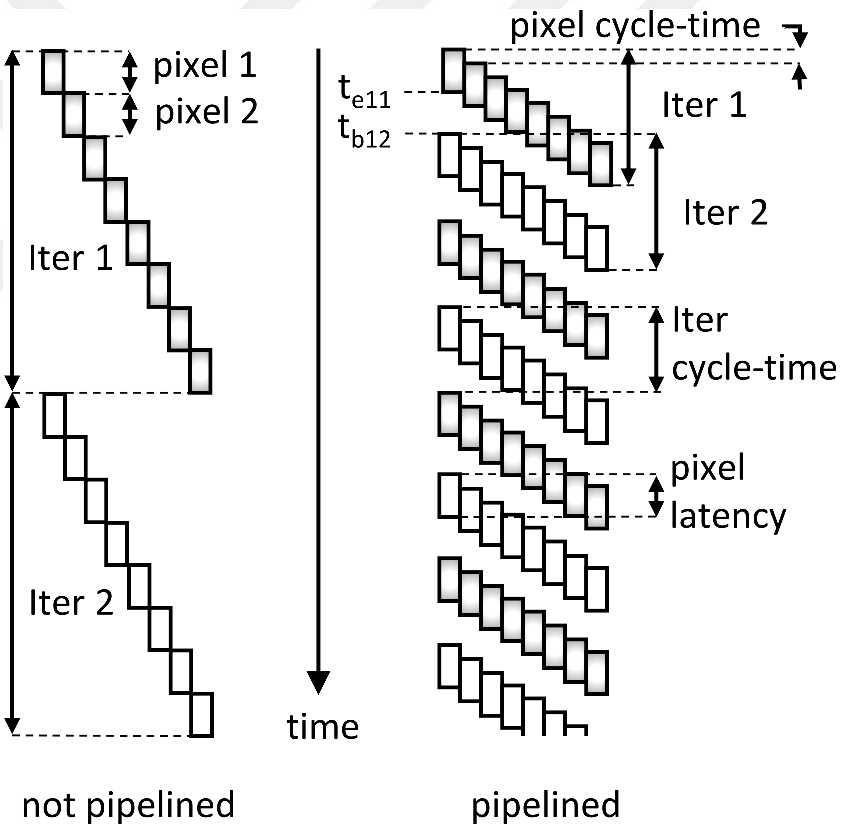

Figure 11: Iter Calls and Pixels Pipelined.

In Fig. 11, the time point t_{e11} indicates the $\underline{e}$nd of pixel $\underline{1}$'s Iter $\underline{1}$ (hence the subscript e11), while the time point t_{b12} indicates the $\underline{b}$eginning of pixel $\underline{1}$'s Iter $\underline{2}$, (hence the subscript b12). The difference between t_{e11} and t_{b12} is equal to the wait-time of a pixel in an LB in Iters Pipe, which is one line of pixels, hence the number of pixels in a line times pixel cycle-time.

Note that while Iter pipeline is a simple "hardware pipeline", the pixel pipeline is a "functional pipeline" (a.k.a. loop pipelining) [47]. A hardware pipeline is a chain of hardware blocks coupled through storage elements (registers or FIFOs). A functional pipeline is, on the other hand, is about overlapping consecutive iterations (pixel processing in our case) and coming up with a new schedule and implementing that schedule. In a functional pipeline, the pipelining is taken care of when the scheduling is done. It actually requires unrolling the loop and rerolling (a.k.a. loop folding) [48]. Fig. 12 shows how functional pipelining is carried out. In this particular case, the latency is 3 times the cycle-time. Compare Fig. 12 to Fig. 11; instead of the 8 pixels at the top right of that figure, here we have 5 pixels. Note that, in our case, prelude and postlude can be handled by disabling the outputs from the pipeline instead of implementing different schedules.

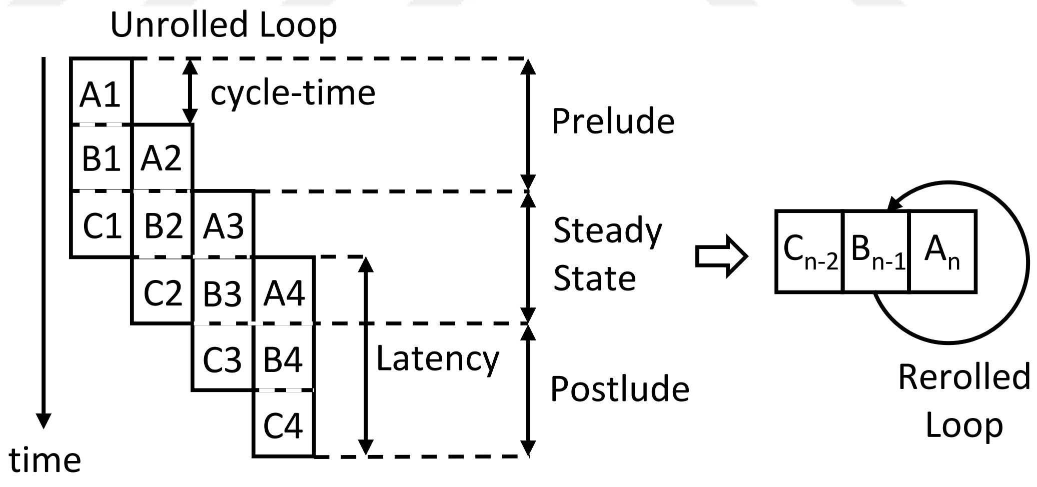

Figure 12: Functional Pipelining is Illustrated.

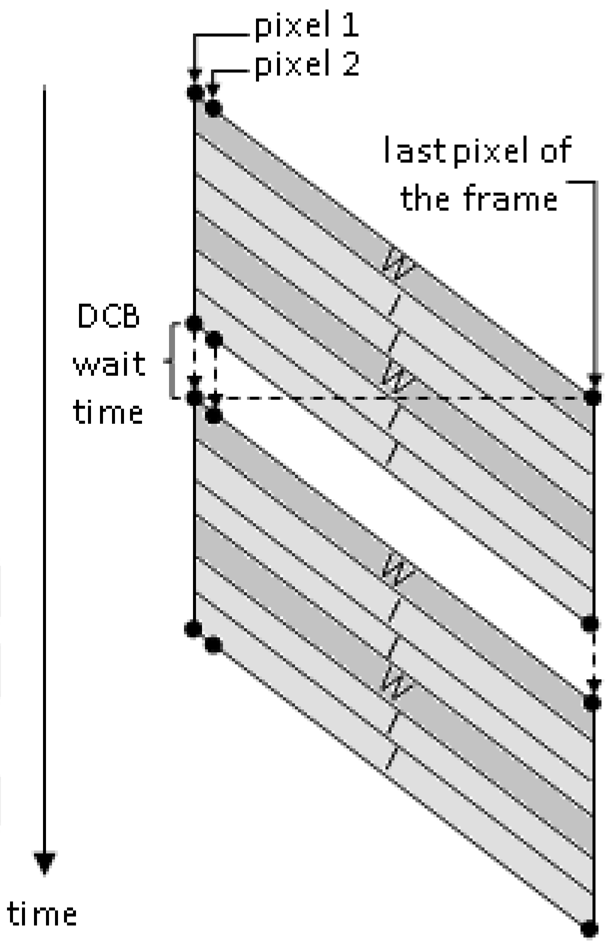

Figure 13: Circulating Around the Iter Pipe.

The pipeline in Fig. 10 offers 2 Is (i.e., Iter() calls) per W and 2 Ws (i.e., Warp() calls). Suppose we would like to do 4 Ws and 2 Is per W. Then, we would go around the pipe twice. In Fig. 13, the first parallelogram shows the first round of pixels going through the pipe, while the second parallelogram shows the second round. The top (inclined) edge of the parallelograms indicate pixels entering the pipe in a staggered fashion in time. The bottom of the parallelograms indicate the pixels coming out of the pipe. Note that the time that pixels wait in the LBs in between W-I, I-I, or I-W are included in W and I times in Fig. 13.

Before we feed the output of the pipe from the right back into the pipe on the left, we must wait until all pixels of the frame are fed into the pipe. That is what

Fig. 13 shows. The point denoted by "last pixel of the frame" is when the last frame enters the pipe and right exactly that point in time is when we can start feeding the pixels output from the first round. In between, we have to buffer the output pixels of the pipe in a DCB.

3.4 Internals of Iter Block

An Iter block in Fig. 10 can be regarded as two blocks as shown in Fig. 14. The line buffer LB there is the same as the LB in Fig. 10. IterTop, LBinternal, and IterBottom put together form the Iter in Fig. 10. The letters r in the LBs in Fig. 14 show the pixels read. An LB is like a FIFO, except that multiple pixels are read instead of a FIFO's one read, and usually it starts working after it fills up and it stays full. The letter w shows the write pointer and points to the first empty slot. The letter c indicates the "current pixel", the pixel for which the output values are computed. LB and LBinternal are both one-line (plus one pixel) long buffers. However, there is a slight difference between them. LB adds a bubble of pixel cycle-time times the line width (in terms of number of pixels) to the latency of a pixel (see the gap between Iter 1 and Iter 2 in Fig. 11, while LBinternal does not add any bubbles to the latency. That is because the current pixel is the oldest pixel in the buffer in LB's case, whereas it is the newest pixel in the buffer in LBinternal's case.

LB and LBinternal can be implemented by a 1-read and 1-write memory block, even when cycle-time is 1 cycle. This is possible by keeping the shaded pixels in Fig. 14 in shift registers (sort of like a cache) and by just reading the tail of the LBs and writing into its head, which is a quite standard trick (see 15). One other thing to note about LB (but not LBinternal) is that it has a regular FIFO (with 4 slots) at its input, and that makes the design stallable.

Iter() in Fig. 3 involves 81 floating-point operations, which comprise of 16 additions, 23 subtractions, 28 multiplications, 8 divisions, 4 comparisons, and 2 square roots. It also involves 6 integer operations. All these operations have to be completed in one pixel cycle-time. We can divide the number of floating point operations by the cycle-time and calculate the number of FPUs necessary.

We can have fully packed schedules, that is, they can be used every clock cycle as there are no “backwards inter-iteration dependencies” within the same Iter stage. This can be achieved by properly elongating the schedule of Iter (hence by increasing the latency) [49]. Note that an inter-iteration dependency is a data dependency of an operation to an operation in a previous iteration. On the other hand, a backwards inter-iteration dependency occurs when the inter-iteration dependency forms a loop in the Data Flow Graph (DFG [47]).

The designs we validated on FPGA boards (3 different) have a pixel cycle-time (a.k.a. initiation interval [47]) of 10 cycles. Therefore, we need 4 adder/subtractor FPU ($= ceiling((16+23)/10)$), 3 multipliers, 1 divider, 1 comparator, and 1 square root FPU. Adder/subtractor (selectable) FPU has a latency of 11 cycles, multiplication 5, division 14, comparison 1, and square root 28 cycles. The latency is 226 cycles. Note that the critical (i.e., longest) path of the DFG of Iter is 204 cycles. The extra 22 cycles comes from limited resources (i.e., FPUs).

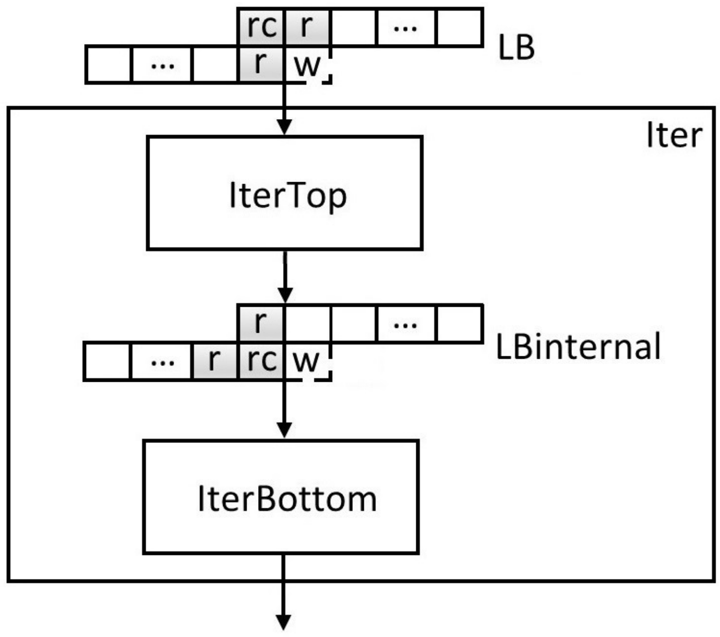

Figure 14: Internals of Iter.

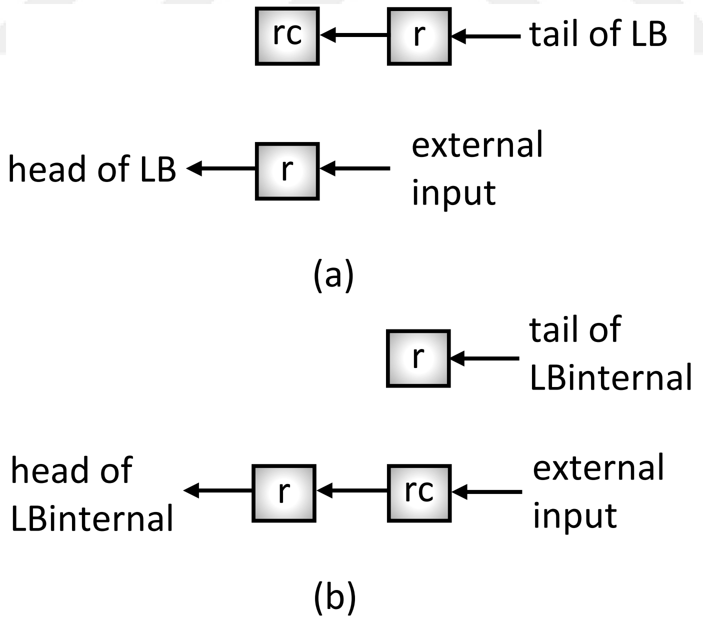

Figure 15: Shift Registers of (a) LB and (b) LBinternal.

We have implemented the functional pipeline in Iter block with HLS. We mainly used our own HLS tool (MAFURES) [49], which we will discuss in Subsection 4.3. However, we are able to implement it with Xilinx Vivado HLS [50] as well. The FPUs are produced by the respective IP core generators of the FPGA vendors. MAFURES is designed with Altera FPUs in mind. For Xilinx, we have wrappers that make their FPUs look like Altera FPUs. Note that we use HLS only to synthesize the resource-shared datapath of Iter, which takes in all inputs from parallel ports at the beginning of the first cycle of the cycle-time. We have a logic block that we call “wrapper”, which contains the FIFO, LBs, shift registers, and more so that the data input and output assumptions of the HLS is satisfied.

3.5 Internals of Scaler and Warp Blocks

We have a downscaler (DS) and an upscaler (US). The DS is shown in Fig.s 7, 8, and, 9. The US is shown in Fig. 10.

In [51], we describe an efficient DS design that works with any downscale ratio. Although this work is focused on DS, the same approach can be used for US. In both designs, we compute a weighted average of 4 neighboring pixels of the original frame in order to compute the downscaled or upscaled pixel.

Note that in the designs we validated on our FPGA boards, we used downscale/upscale ratio of 2. That resulted in a relatively simple design compared to [51].

The Warp design is similar to DS/US as it also does a weighted average of 4 neighboring pixels, except that the 4 pixels are not accessed in raster-scan order but through the previously computed motion vectors. In a way, they are random-access.

Although Fig. 11 assumes only Iter blocks in the pipeline, the picture with Warp blocks in the pipeline as in Fig. 10 would not be too different. Warp would also be shown in Fig. 11 as a group of staggered pixel boxes but the pixel boxes would have a different height compared to the ones in an Iter. Just like the overlap of Iter blocks, Warp block also overlaps with the following Iter block. Also, Warp requires more lines in the LB at its input. It has 8 lines, a number which is based on the largest vector that can be computed between two warps.

Although Warp() is, in principle, similar to DS and US, it in fact involves a much larger number of operations compared to them. That is because it warps several different quantities. It also requires several operations to compute the coordinates from which to warp. Its boundary condition checking is more complicated than DS and US. Warp() involves 89 floating-point operations, which comprise of 41 additions, 30 subtractions, 18 multiplications. In addition, it has 20 float2int and 35 int2float operations as well as 33 integer operations. Just like Iter, all these operations have to be completed in one pixel cycle-time. The latency is 229 cycles, while the longest path of the DFG is 192 cycles.

3.6 Host Implementation

We have a software running on the Host that is comprised of 5 threads:

1. Video capture (from hard disk, webcam RTSP stream, etc.)
2. Stream video to the FPGA (Vin in Fig. 7)
3. Receive the output stream from the FPGA (Vout in Fig. 7)
4. Post process Vout stream
5. Display the output stream

These threads form a long pipeline together with the blocks in hardware (FPGA) as shown in Fig. 16, where they are marked from 1 to 5. If the software part of the system had not been pipelined (through multi-threading), the hardware pipeline would have been of no use, and the system's cycle-time (frame time) would be equal to the system's latency.

The heartbeat of the pipeline is one frame time (i.e., 1/fps). That is also the latency of all hardware and software blocks (i.e., threads) except for Iters Pipe. Since it is a pipeline in itself, its latency does not have to be one frame time. It can even be one plus a fractional number. The pipeline stages are isolated from each other through some sort of FIFOs. They are "Queue" classes in the case of software part of the pipeline. In the case of the FPGA part of the pipeline, they are DCBs in DRAM (explained earlier) or SRAM FIFO (made using FPGA Block RAM).

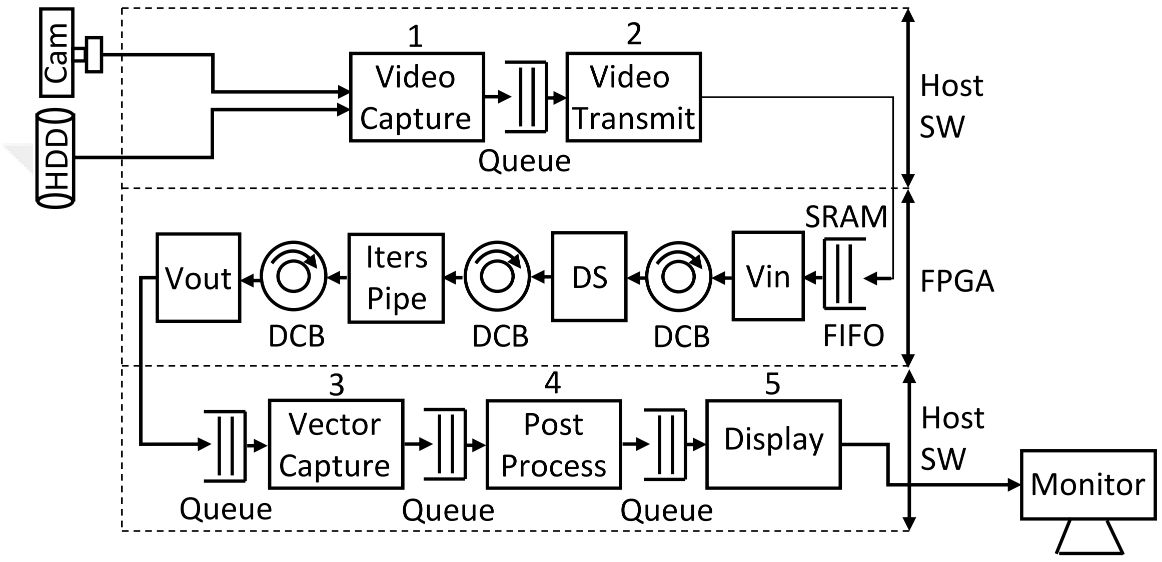

Figure 16: The Complete HW/SW Pipeline.

The output of the FPGA application is shown in Fig. 17. This image is the visual optical flow result that is produced from the current and the previous frame. Stationary parts stay white while moving parts of the images are color-coded. Different colors show different motion vector directions, and the intensity of the color shows the magnitude of the motion vector.

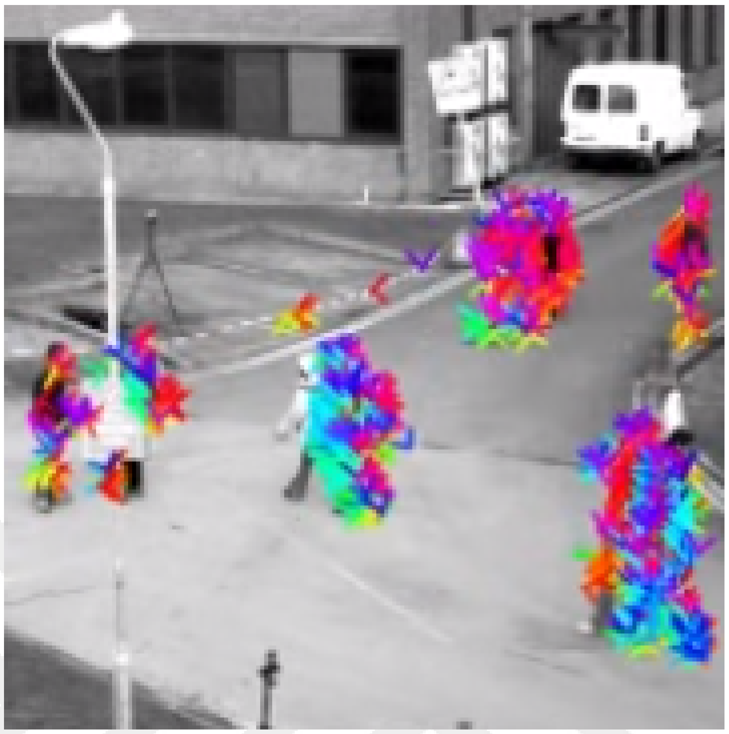

Figure 17: Output of the Optical Flow Algorithm.

CHAPTER IV

TOOLS AND TECHNIQUES

While doing our family of designs, we used a rich set of tools and techniques that can benefit many FPGA-based (even ASIC-based) designs, mainly in video processing.

Subsection 4.1 gives an overview of our methodology and the tools/techniques used within and how they can benefit other video processing or similar designs. Some of the tools and techniques we used have already been explained with significant detail in previous sections. For those that need further explanation, we have subsections in this section.

4.1 Overview of the Tools and Techniques

Our overall methodology, shown in Fig. 18, is quite typical. "Design" refers to writing or generating the design in Verilog. "Verification" refers to simulations. When verification catches bugs, we go back to design and fix the bug. Once verification passes, we synthesize. If "Synthesis" does not meet timing or does not produce a design that fits the FPGA, we go back to the design step. If synthesis passes timing and resource constraints, we go to "Validation", which refers to testing the design on the actual hardware platform (which includes the FPGA).

What is unique in our methodology is how we carry out the "Design" step in the above methodology. Fig. 19 shows what happens in our case inside the Design box in Fig. 18. Note that we have a few unique approaches in "Verification" and "Synthesis" as well.

Our (video) algorithms engineer coded the algorithm in Matlab just like so many of their counterparts (who may also code in C or some other high-level language). The dotted line in Fig. 19 shows the interface of the algorithms group and hardware group. The first thing we do is to convert the code to Java (C could also be used), as it makes the loops explicit and makes the operations scalar. HLS works on scalar operations. Then, what we have to do is to "Capture Requirements". However, the requirements cannot be captured simply from the Java code. The exact I/O ports of the FPGA, i.e., where the input video comes from, the output video goes to, and the throughput of the system (pixels per second) are not found in Matlab or Java code as these are purely functional languages. To capture all requirements, the hardware team needs to hold discussions with the algorithms team. Given the Java code, the "Compute Structure" (similar to Fig. 3) and performance requirements (i.e., operations per second) can be extracted.

Based on all that, clock frequencies and whether a DRAM will used can be decided. Usually, there are many different options for a given pixel processing performance with different area, power, and DRAM bandwidth. For example, in our case, we can lower pixel cycle-time in Iters Pipe and hence can lower the number of Iter blocks in Iters Pipe. That gives us better utilization of FPUs but increases DRAM bandwidth as fewer Iter blocks requires that we circulate around Iters Pipe more. That requires us to access the DCBs in DRAM more. We can do all these trade-offs in our architectural estimator, i.e., a tool that has been developed as part of our design approach for optical flow. The "Estimator" can be thought of as part of our "Capture Requirements" step.

"Extract Basic Blocks" divides the overall Java code into "Basic Blocks" such as the body of an innermost loop. Think of these as formulas, and HLS builds a datapath using resource-sharing for each formula. HLS-generated formula compute engines are plugged into our design architecture. Our "Generate Design Architecture" process uses some automation and some manual design, uses HLS-generated cores, and IPs (especially for the interfaces) and integrates everything. This process is shown in Fig. 21.

We will now list the tools and techniques that we used in our optical flow design, and which, we think, might be useful for other designs. Each will then be explained in its own subsection.

- Nested Pipelining
- Architectural Estimation
- HLS and Code Generation
- PCIe/USB/DRAM Interface Framework
- Flow Based Verification
- Synthesis Batch Runs

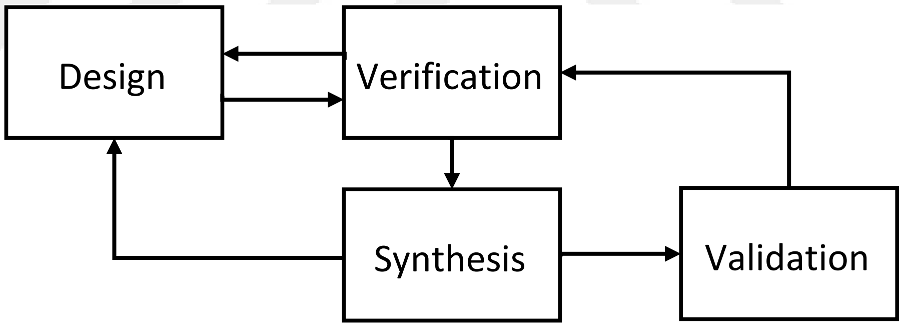

Figure 18: Our Implementation Methodology.

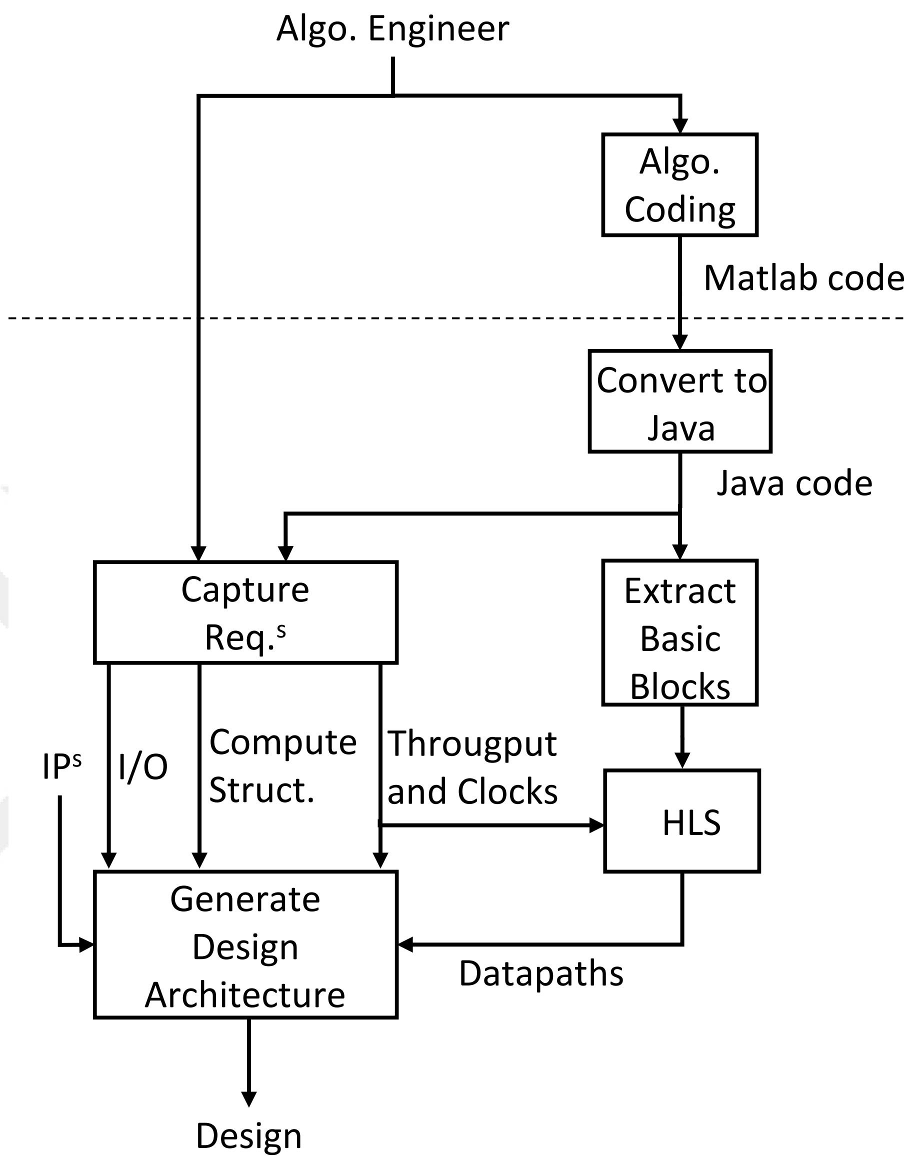

Figure 19: Our Design Process.

4.2 Nested Pipelining

In Subsections 3.1 and 3.3, we have explained the nested pipeline in our optical flow design in quite detail. In this subsection, we will explain how our nested pipelining can be applied to other video processing designs. Our nested pipeline processes frames of video. For every frame it applies Iter and Warp functions repeatedly to the frame multiple times. These functions do what they do on pixels of a frame. On top of it, the floating point and integer function units (FPUs, etc.) of these functions are also pipelined.

Almost every video processing hardware can use all of these types of pipelines except the pipelining between the repeated instances of Iter and Warp units. Other video processing algorithms may not be applying the same function over and over again. Actually, that only results in an Iters Pipe with different pipe stages in Fig. 10. The only case that does not exactly fit our paradigm is when we have more Iter calls than the Iter blocks in the pipeline, and we have to circulate over the pipeline. That is to say that all our nested pipelines can be applied to other video processing problems; the only problem is with "circulation", which is actually not pipelining.

However, even that can be applied to other designs that have many functions applied to a frame back to back. Let us suppose instead of our repeated Iter and Warp calls, we have foo1(), foo2(), foo3() through foo12(). Then, Fig. 10 becomes the pipeline in Fig. 20. The pipe stages there are multifunctional. For example, the leftmost pipe stage executes either foo1() or foo7(). The HLS problem here is similar to (but not exactly the same as) scheduling two disjoint DFGs (respectively for foo1 and foo7) to twice the number of clock cycles as pixel cycle-time.

One other thing that is unique about our pipelines is that the cycle-time of the pipeline is not equal to the latency of the longest pipe stage. For example, in Fig. 7, the timing of the top-level pipeline (Fig. 9) is shown. The pip stage there with the longest latency is Iters Pipe but the cycle-time of the pipeline (i.e., frame time) is shorter than that latency, which is indicated by Iters overlapping in 7. That is possible because Iters Pipe contains a pipeline internally.

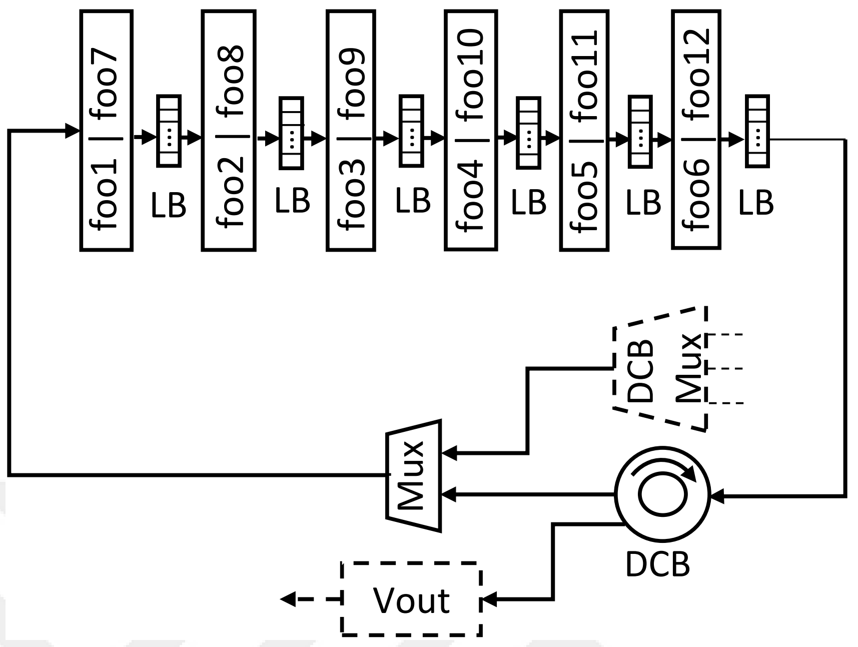

Figure 20: Reusing (Circulating) the Whole Pipeline.

4.3 HLS and Code Generation

This subsection describes our code generators, namely, Pipe Generator, Wrapper Generator, and HLS (see Fig. 21). This process is mostly automated, except where it says "manual".

Iters Pipe (Fig. 10) and some of its building blocks are automatically generated. The top-level of the pipeline, generated by an approximately 600-line Perl script, can be as many as 16k lines of Verilog for a 50-Iter pipeline.

Iter and Warp modules are both synthesized through our own in-house HLS tool (MAFURES). LB as well as LBinternal and the part of Iter block that interfaces to the LBs are generated from what we call "wrapper generator". The DRAM client block that implements the DCB in Fig. 10 is a manually written module but highly parameterized. It is quite sophisticated; it keeps track of frame resolution, where we are in the pyramid, how many times we are circling around the pipe, buffer sizes, etc.

The wrapper generator reads in an input configuration file with a few lines that describe the LBs in terms of r and c pointers in Fig. 15 and buffer size. As a result, it generates the read/write and reset logic for the LBs. It also generates the necessary buffering and logic for “forward inter-iteration dependencies”.

Since the architecture of our Iters Pipe is applicable to other design problems, our pipe generator, wrapper generator, and highly parameterized DRAM client are also applicable to other design problems with not much modification.

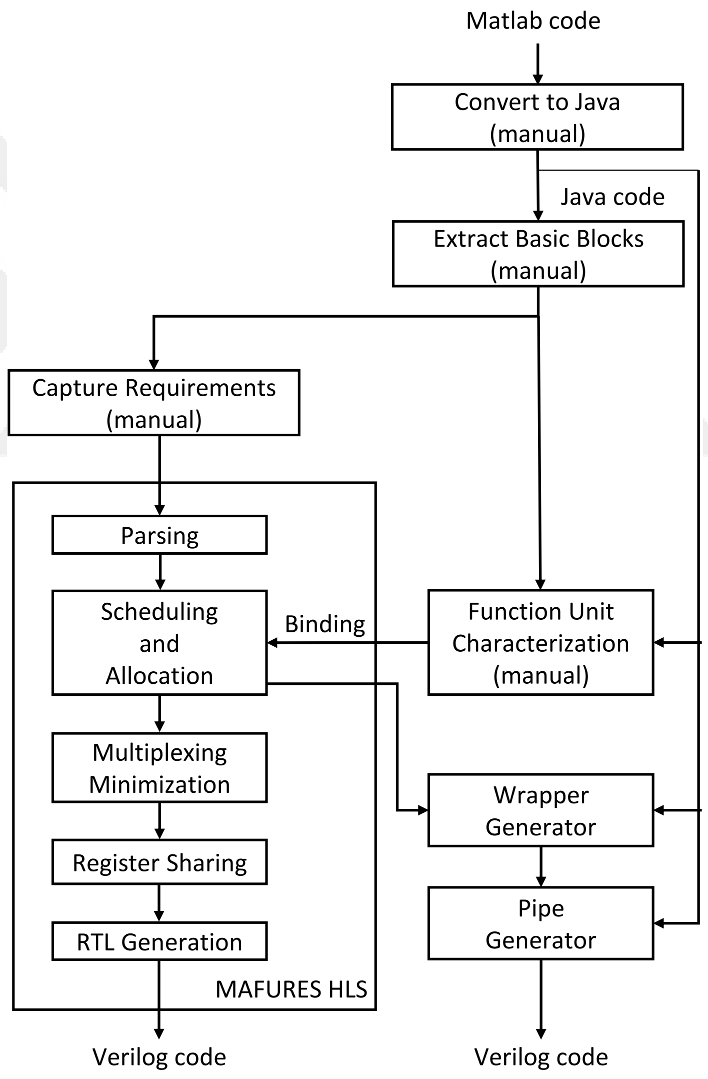

Figure 21: How Our Code Generation Works.

Having said all of this, the most crucial, biggest, and most general-purpose code generation we do comes out of the HLS. We have developed our own HLS tool.

However, we are also able to use Vivado HLS. It is very difficult to do FPGA design with a traditional approach when the video processing algorithm to be implemented is not completely frozen. Even a small update in the algorithm could cause all the hardware work done so far to become garbage. In such cases, it is best to use a HLS tool that can generate code from the high-level code that describes the algorithm (e.g., Matlab, C, Java, or similar). Because it is much easier to make changes in a high-level language. Only new code generation is required after the change if a HLS tool is used.

We call our HLS tool MAFURES, short for "Multiplexing Aware FUnction and Register Scheduler" [49]. MAFURES generates concise Register Transfer Level (RTL) Verilog code, i.e., hence synthesizable by a simple-minded logic synthesis tool. Since MAFURES' output is concise (i.e., much fewer lines compared to its counterparts), it is readable, therefore, more easily debuggable. One other consequence is that simulation runs faster. MAFURES is multiplexing (muxing) aware, that is, (i) it tries to minimize muxing and (ii) schedules muxes into their separate clock cycles.

Mux minimization produces fewer muxes and muxes with fewer inputs. Therefore, it both minimizes area and maximizes clock frequency. MAFURES also offers the option of not sharing registers. That eliminates muxes at register inputs, which lowers combinational logic used, may improve clock frequency, but increases the number of registers needed.

Mux scheduling, on the other hand, maximizes clock frequency but may increase area. Mux scheduling may have an adverse effect on fps maximization. That is because fps is proportional to both clock frequency and the number of Iter blocks we fit in our FPGA. Although mux scheduling improves clock frequency, since it uses more area, the number of Iter blocks may go down. Muxes can be scheduled to their separate clock cycle in most video processing designs. That is because video processing algorithms usually do not have backwards inter-iteration dependencies and hence the schedule can have arbitrarily large latencies (which are longer than the cycle-time).

Although MAFURES has a separate mode for mux scheduling, the same result can be reached by inserting registers at the inputs of the FPUs (in addition to the registers we already have at FPU outputs) and increasing FPU latencies by one cycle.

MAFURES, like any other HLS tool, carries out parsing (Java basic block to DFG), scheduling, allocation, binding, mux minimization, and register sharing as shown in Fig. 21. In terms of how exactly it does these, it may be slightly unique. In terms of parsing, it can handle *if* statements. It does scheduling and allocation as a combined step. In terms of binding operations to FPUs, it uses only one type of FPU for every operation. It reads the FPU from a library file (coming from Function Unit Characterization in Fig. 21) and does not trade-off among multiple FPU implementations. In mux minimization, it tries to make the number of mux inputs before all FPUs equal as much as possible. To do that, it tries to move an operation from the FPU it was initially allocated to another FPU without changing its schedule. It can include the muxes that come from *if* statements. In register sharing, it offers no register sharing as well as multiple register sharing algorithms.

MAFURES is a quite general-purpose HLS tool. It can even be used for applications outside the area of video processing. It currently supports synthesis of basic blocks with scalar inputs. For vector inputs (i.e., arrays), wrappers are needed, and we have wrapper generators for our specific applications. The original version of MAFURES, with which we produced most of our synthesis results, handles simple single operation statements. However, it also can handle *if* statements. Currently, we have a beta version of a parser that can handle a variety of complex expressions. The only application-specific aspect of MAFURES is that it assumes there are no backwards inter-iteration dependencies, which does not impose any constraint on the latency. Our latencies are usually very high (a few hundred cycles) with respect to our cycle-times (usually between 1 and 20). That is because we use FPUs (Floating Point Units), which have long latencies. When we schedule operations on to FPUs, we have a greedy approach (ASAP heuristic). We start from the earliest clock cycle and allocate the operation to be scheduled (op) to the first available (i.e., not busy) FPU (with capability to execute op) available in that cycle. If all FPUs that can execute op are busy, we try the next cycle. If the FPUs are all busy in the last cycle also, we "wrap around" to the first cycle and decrement the iteration count of the operation. That, in a way, introduces a new forward inter-iteration dependency, requiring special buffering in the wrapper.

4.4 Interface Framework, Development, and Verification

It is very important that the FPGA can communicate with the outside of chip. The interface framework developed for this purpose and the methodologies used during development are described in the subtitles.

4.4.1 PCIe/USB/DRAM Interface Framework

Table 3 shows our approach to developing Host/FPGA interfaces (PCIe and USB). Our layered, and hence modular, approach offers us portability, which implies faster development. That means we have a high degree of reuse–shown with R's in the table. Table 4 shows our DRAM interface design approach using the same symbolism.

Let us now explain how to read Table 3 and Table 4. We implemented our design on 3 hardware platforms:

1. Terasic DE-150 board with Altera **Cyclone IV** FPGA, **PCIe** gen 1.0, and **SDR** SDRAM

2. Linera FMU3G-S6 board with Xilinx **Spartan-6** FPGA, **USB** 3.0, and **DDR2** SDRAM

Table 3: Layers of the Host Interfaces and Their Reuse.

PCIe Cyclone IV		**PCIe Arria 10**		**USB Spartan-6**		
N	1: Host App	**R**	2: Host App	**R**	2: Host App	HL
N	1: App-RIFFA API	**R**	2: App-RIFFA API	**R**	2: App-RIFFA API	
P	1: RIFFA API	**R**	2: RIFFA API	N	RIFFA-USB API	
P	1: PCIe 1.0 Driver	P	2: PCIe 3.0 Driver	P	2: USB 3.0 Driver	LL
G	1: PCIe 1.0 PL Soft IP	G	2: PCIe 3.0 PL Hard IP	P	USB 3.0 PL IC	CP
P	1: RIFFA IP	**R**	2: RIFFA IP	P	USB IP	
				N	USB-RIFFA Bridge	
N**R**	1,3: Bus Controller	M	2: Bus Controller	**R**	4: Bus Controller	
N**R**	1,3: Vin and Vout	M	2: Vin and Vout	**R**	4: Vin and Vout	
N	1: FPGA App	**R**	2: FPGA App	**R**	2: FPGA App	FP

HL: Higher Layer software **LL:** Lower Layer software
FP: Farther from Pins **CP:** Closer to Pins
R: Reused **P**: Pluggedin **G:** Generated **M:** Modified **N:** New
PL: Physical Layer

3. Nallatech 385A board with Altera **Arria 10** FPGA, **PCIe** gen 3.0, and **DDR3** SDRAM

The columns in the two tables correspond to the above hardware platforms. Although we worked on them in the above order, we placed hardware platforms 1 and 3 next to each other in the tables, because they both use PCIe, and hence, it is more important to see how they relate to each other than to see how they relate to the USB-based platform.

The rows of the two tables, on the other hand, correspond to the layers in our layered approach. In Table 3, the higher layers (in gray) are software layers running

Table 4: Layers of the DRAM Interfaces and Their Reuse.

SDR Cyclone IV		**DDR3 Arria 10**		**DDR2 Spartan-6**		
G	1: SDR Soft IP	G	2: DDR3 Hard IP	G	2: DDR2 Soft IP	CP
				N	Avalon-Native Bridge	
N**R**	1,3: DRAM Controller	M	2: DRAM Controller	**R**	4: DRAM Controller	
N	1: RW Queues	**R**	2: RW Queues	**R**	2: RW Queues	
N	1: FPGA App	**R**	2: FPGA App	**R**	2: FPGA App	FP

FP: Farther from Pins **CP:** Closer to Pins
R: Reused **G:** Generated **M:** Modified **N:** New

on the Host, going from the Host Application (marked with HL, i.e., Higher Layer) to the software driver level (marked with LL, i.e., Lower Layer). Then, we go over the PCIe or USB wires (corresponds to the horizontal line between the gray area and the white area underneath) to the FPGA pins and then on to blocks in the FPGA farther and farther away form PCIe/USB pins (hence the labels CP and FP).

Let us first examine the leftmost column in Table 3. The topmost cell in the column is Host App(lication), which in our case corresponds to the threads 2 and 3 in Fig. 16. These apps use functions in an API (Application Programming Interface) written by us (i.e., App-RIFFA API). This API is shown in the second row from the top. The layer below is the API that comes with RIFFA, which is a high-level hardware/software framework that supports PCIe 3.0 and below and works across many hardware platforms. RIFFA [52] stands for "Reusable Integration Framework for FPGA Accelerators" and presents the same API to its users, irrespective of the version and parameters of the particular PCIe on the host platform. The particular version of RIFFA we finally used is 2.2.

To the left of the cell that has RIFFA API in it in the leftmost column of Table 3, there is the letter P, which is short for "Plugged in". That indicates that RIFFA API was not developed by us and was plugged in as a ready-made (software) module.

That is in contrast to Host App and App-RIFFA API. These have the letter N to their left, which indicates that they are "New", i.e., they were new development works by us. These were then "Reused" (hence the letter R) in our implementations on the other hardware platforms (PCIe Arria 10 and USB Spartan-6) without any changes, resulting in complete portability.

The lowest layer (LL) of the software part of the PCIe interface on PCIe Cyclone IV platform is the PCIe 1.0 Driver, which was also plugged in (P). The lowest layer in software interfaces to the lowest layer in hardware (FPGA), which happens to be "Closest to Pins" (CP). That is PCIe 1.0 PL Soft IP. PL stands for "Physical

Layer", while IP stands for "Intellectual Property" (i.e., hardware library block). This particular block is Soft IP, meaning that there is no fixed block (i.e., hard macro) for this job on the FPGA. It is a synthesizable block "generated" (hence the letter G) differently based on the specific PCIe parameters supplied by the user through the appropriate IP core generator wizard (GUI). Next layer up in the hierarchy of layers is RIFFA IP, a parameterized hardware block, which works with the RIFFA API on the software side. RIFFA IP is also plugged in, i.e., not generated.

The case of the Bus Controller is rather different and unique. We first developed a high-level bus controller for our Cyclone IV platform. The bus controller gathers the read/write requests for the PCIe/USB bus and carries them out in such an order that the bus is efficiently used and yet the requirements of the clients are satisfied. This requires appropriately sized bursts and an appropriate arbitration between different client queues (i.e., FIFOs). When we wanted to use this bus controller for the Arria 10 board, we realized that it had to be modified (hence the letter M) to cope with the much higher bandwidth requirements of the Arria 10 based system. However, we did it in such a way that it is backwards compatible with the Cyclone IV. To recap, Bus Controller was a N(ew) design for Cyclone IV, then was M(odified) for Arria 10, then was R(eused) by Cyclone IV. Table 3 does even state this order. Number 1 to the left of Bus Controller cell in Cyclone IV column indicates that it was the first one designed. Number 2 to next to Bus Controller for Arria 10 indicates that it was designed next. Number 3 next to Cyclone IV's Bus Controller indicates that it was redesigned after number 2. Number 4 next to the USB Spartan-6's Bus Controller indicates that it was taken care of next by, in fact, through reuse.

Actually, we first did the USB Bus Controller in an ad hoc way, and then, revisited it and cleaned it up. In doing that, we decided to reuse the bus controller we did for Arria 10, which interfaced to RIFFA IP. Therefore, we wrote a "Bridge" from

RIFFA interface to the interface that the USB IP that came with Linera's USB-based Spartan-6 board, which yet uses an integrated circuit (IC) for USB physical layer.

We carried out development effort for Vin and Vout modules (of Fig.s 8 and, 9) in parallel with the Bus Controller. That is why the N, M, R tags and the numbers next to them are identical to the ones for the Bus Controller.

As a result of our approach, as you can see, not only we achieved perfect reuse (hence portability) for the Host Apps but also for the FPGA Apps, that is, for our optical flow FPGA design. Without this, our FPGA design job would be extremely difficult, because it is already very complex, and without this framework, we would have to modify it for each platform.

Once Table 3 is understood, Table 4 can easily be deciphered. In the case of our DRAM interfaces, there is no software involved as it is only interfaced by the FPGA. The DRAM Controller here involved a very similar development path, and we achieved perfect reuse not only for the FPGA App but also our DRAM Read/ Write Queues (i.e., FIFOs). Avalon in Table 4 is an Altera SoC Bus, and since we mostly worked on Altera platforms, our DRAM Controller interfaces to the Avalon Bus. This is similar to our Bus Controller for PCIe and USB interfacing to RIFFA. Therefore, just like we have a RIFFA-USB Bridge in Table 3, for our USB-based DRAM interface, we have an Avalon-Native Bridge, where Native is a peer-to-peer Xilinx interface.

Note that the numbers we have in all rows (1:, 2:, etc.) indicate the order of our development activities. If we do not have a number on the left of a cell, that means it could have been developed before or after or concurrently with the similar blocks in other columns.

Our expectation from this part of our work was fast turn-around times for porting design to new platforms. That requires high reuse. In our two tables, not only cells labeled with R show reuse but also the ones labeled with P and G show reuse. P and G show reuse of others' IPs. Assuming our N(ewly) developed layers will mostly continue to be reused, one possible metric of our level of reuse is the ratio of "all cells in the two tables except the ones labeled with N and M" over "all cells in the two tables except the ones labeled with N". That gives us a reuse ratio of 91%, which is impressively high.

Our hope and expectation is that through our work here within the context of this section will result in quick development of streaming interfaces for video processing and other types of applications. Currently, we used our framework for an image fusion application as well as an optical wireless communications application.

4.4.2 Interface Development and Verification Methodologies

In general, the design development and validation process on FPGAs are very difficult. However, the design of the interfaces to communicate with the outside of the chip or with other modules is much more difficult. The main reason for this difficulty is the strict rules that must be observed when designing the interface. Communication is not possible with the smallest error or shortage.

This subsection will describe the methodology that should be followed when an interface development problem is encountered that has not been experienced before.

The operations to be performed for the interface can be divided into two parts, in-chip and out-of-chip.

If an IP is not used in the design, the designer is free to decide how to communicate between the modules it has developed. The designer can use a standard to communicate, but this is optional. However, there are some points to pay attention to if IPs designed by other designers are to be used in their own design. The most common types of interfaces used by designers need to be known. These are listed below.

1. AXI, Xilinx
2. Avalon, Altera
3. Wishbone, Opencores

AXI (Advanced eXtensible Interface) interface is used by Xilinx. Developed by ARM company. There are many variants of the AXI interface. 3 varieties are frequently used by IP developers. There are some of examples of the work done with the AXI interface [53, 54, 55, 56] mentioned.

Frequently used species are listed below.

1. AXI4
2. AXI4-Lite
3. AXI4-Stream

AXI4 offers a memory mapped design option. It allows transfer of up to 256 data at a time. AXI4-Lite offers a simpler interface. Used in low throughput needs. It is usually preferred when the value of a register needs to be read. The AXI4-Stream can transfer unlimited data bursts. This interface is not memory mapped.

The AXI interface has 5 channels. These are dataWrite, dataRead, addressWrite, addressRead, and writeResponce. Fig. 22 shows the read, Fig. 23 shows the write channels.

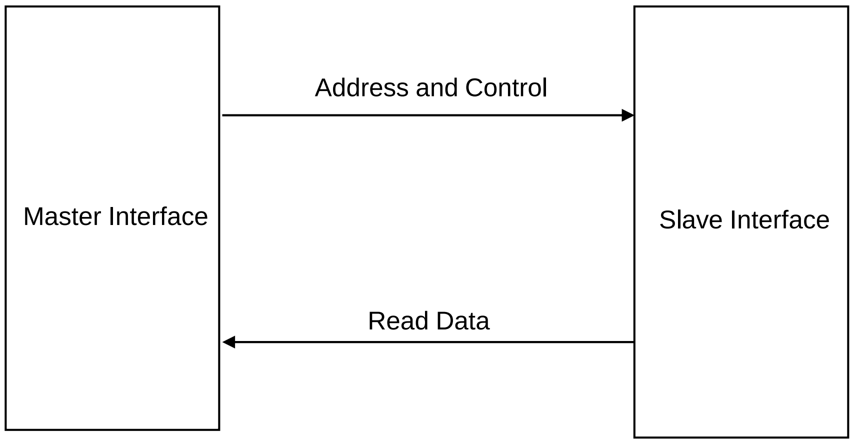

Figure 22: AXI4 Read Channels.

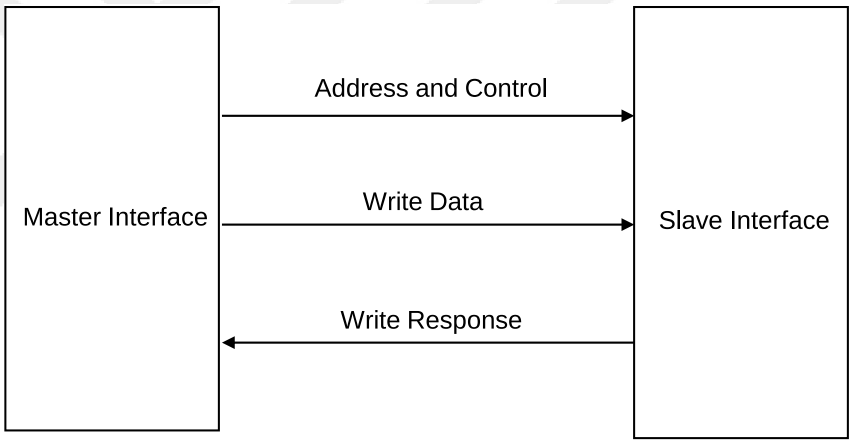

Figure 23: AXI4 Write Channels.

All the details of the AXI signals can be viewed at [57] reference.

When IPs that Altera provides are used, an interface called Avalon needs to be deployed. Examples in the literature are [58, 59, 60, 61]. Avalon bus also has many types. There are three species that are frequently used. These are listed below.

1. Avalon-ST, Streaming

2. Avalon-MM, Memory mapped

3. Avalon-Conduit, Individual signals

Detailed information about the signals in the Avalon interface is available in reference [62].

Finally, many IPs in OpenCores are usually designed to conform to the wishbone interface standard. Work on the Wishbone interface can be found in references [63, 64, 65, 66]. There are four different search options. These are listed below.

1. PointToPoint

2. Data Flow

3. Shared Bus

4. Crossbar Switch

Wishbone signals can be viewed from the [67] reference.

Various methods can be followed to verify the developed interfaces. These are listed below.

1. In-Chip Verification

2. IP Verification

3. C Models

4. Commercially Tools

In-chip verification is very useful when the designer does not have full knowledge of the developed interface. It is necessary to observe the data fed and received at the interface and dynamically to give new data. In other cases, it is necessary to modify the design and synthesize again. This greatly increases development time. As a solution to this, there are a variety of tools offered by Xilinx and Altera to monitor the signals in the design. These are Chip Scope and Signal Tap software. With these softwares, a register can be observed on the FPGA. The most important issue is to use the validation CPU in order not to synthesize again and again. The interface must be controlled with software that will be sent to the validation CPU so that many conditions can be tested by loading new software into the CPU at runtime.

In order to verify the IPs, the interfaces that they use must be driven correctly. For this, Xilinx and Altera provide TBs for the IPs they produce. With these TBs, the behavior of IP can be learned.

Another verification method is behavior verification. If the designer wants to match the C code and FPGA design, there will be very small differences if floating point operations are used. The model C of the IP used for this is needed. These models are supplied by the IP manufacturer. When the library provided is used, exact matching can be provided.

Commercially available tools that contain various interfaces can be used. An example is the Cadence TLM.

4.5 Flow Based Verification

In a video processing application with 4 nested and stallable pipelines, which traverse several buffers in several different types of memories, the only thing can go wrong is not number crunching. Often, synchronization problems occur between pipeline stages. One cannot tell why the values of a pixel are wrong based on simply the values themselves. If there is a mismatch between the values in simulation and what they have to be (coming from test vectors dumped from a behavioral golden reference), maybe it is because the pixel we are checking is not the pixel we are expecting.

This is similar to a network switch chip. It moves packets around. It also modifies the headers and/or payloads of packets. However, first thing we verify in such chips is the order and timing of packets. Maybe the packet we are looking at is not the packet we are expecting. In the case of networking, that is easy, because we can tell packets apart usually from their headers and/or payloads.

In a chip like ours, the flow of pixels within the FPGA is quite complex and it is similar to the flow of packets in a networking chip. However, the situation is even more complex. Because we cannot tell which pixel is which just by looking at the values of pixels. To address that, we may also store and carry around an ID number for every pixel. However, that increases the area of the design as well as power and may lower its clock frequency. The solution is doing this during simulation and disabling it in the production version the design.

A solution that is even better is doing first a simulation with just id numbers. That is, do not do the number crunching, and tag pixels with additional tags to indicate that the number crunching is done. We do this for Iters Pipe by not plugging in the "formula compute engine" from HLS for Iter. We still run HLS and get the schedule from it and appropriately generate the "wrapper". The wrapper writes ID numbers and flags instead of the numbers produced by the formula compute engine.

For this purpose, a software has been developed which tests the accuracy of flow by removing calculation units from the design. Verification times and the effort required to confirm are significantly reduced. It divides verification into two parts. The debug process is becoming easier.

The module names and input/output port names of the designs to be tested for flow must be determined. Advanced Verilog/VHDL parser software or libraries can be used for this task. However, these softwares are not used because only the mentioned parsing process is very simple. With simple regular expression methods module names and ports are obtained. Accordingly, the hierarchy is constructed as in Fig. 24.

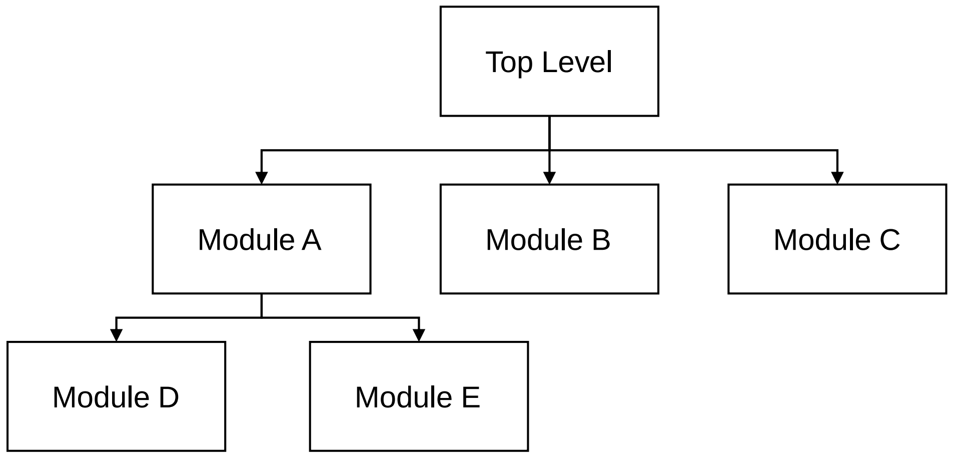

Figure 24: Module Hierarchy.

At this stage, the script creates simulation models. Simulation models are created by removing the calculations of existing modules and placing only the flow related logic. Fig. 25 shows the creation of simulation models. The newly created modules take input and output timings as parameters. According to these parameters, these modules takes pixels as a input.

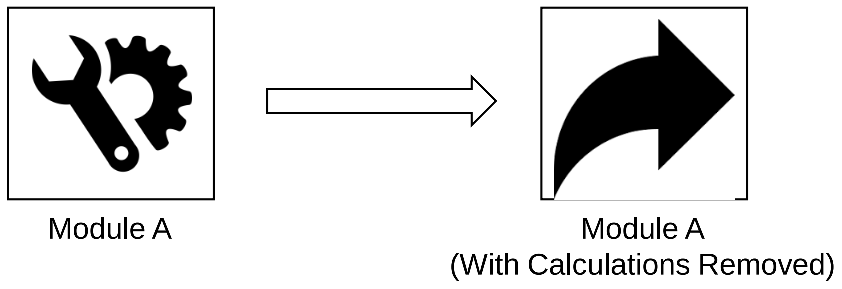

Figure 25: Simulation Model Creation.

Simulation models prints to a file with timing of inputs. At the end of this process we get a text file containing the movements of the pixels. The general flow is shown in Fig. 26.

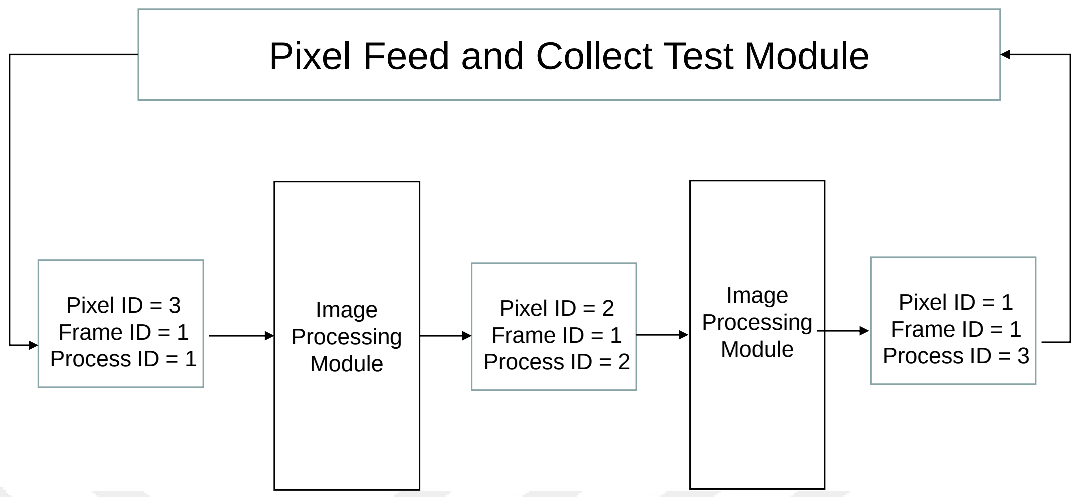

Figure 26: Simulation Flow.

Table 5 shows a sample output file.

Table 5: A Sample of Simulation Model Output File.

Pixel 1, Module A, Time 300 ns
Pixel 2, Module A, Time 310 ns
Pixel 1, Module B, Time 310 ns
...

A second script takes the generated text file as a input. The rules that the design should comply and the flow that should be defined in this script. This script checks whether the flow of the design is correct.

The second script for optical flow design checks the movement of the following modules.

- Downscaler
- Upscaler
- Iters

- DRAM

The design includes input feeder and output collector modules to feed input to the top module and to receive the generated outputs. Input feeder inputs according to the specified cycle-time. Output collector is taken according to the given cycle-time in the outputs.

4.6 Synthesis Batch Runs

We have our Architectural Estimation tool to do architectural trade-offs. However, we cannot exactly know the max clock frequency we can achieve and the number of Iter blocks we can fit on the FPGA without doing synthesis. Estimation lets us narrow down our search space for a good solution. That is, if we do estimation, we can run fewer syntheses.

We wrote Perl/C# scripts (totaling around 600 lines) that can run many syntheses with different parameters in batch mode. It actually generates our optical flow design, runs synthesis, and analyzes the log files. The script takes in parameters that it passes to our design generators (including HLS), the implementation tools (synthesis through power analysis), and analyzers. Below are the steps of the script:

1. Design Generation
2. Synthesis
3. Mapping
4. Timing Analysis
5. Power Analysis
6. Dispatcher
7. Report collector

8. Report Parser

9. Table Maker

Design Generation step takes in the following parameters:

- Cycle-time
- Number of Iter blocks
- Mux-aware HLS on or off
- HLS Register Sharing on or off

Synthesis, mapping, timing analysis takes in Target Clock Frequency as a parameter. Dispatcher dispatches a new thread of steps 1 through 5 of the above list as long as we are using less 80% of the server's memory. Note that the implementation tools are also multi-threaded within themselves, and we allowed them to spawn 2 threads.

Report Collector gathers the reports (i.e., implementation tools' log files) into a single directory. Report Parser parses the log files looking for critical information, while the Table Maker creates the associated Excel tables.

Using this tool, we carried out around 3,000 synthesis runs in a net running time of approximately one week. We ran them on a 48-core machine. On the average, we got a 20x speed-up compared to single-thread synthesis run. Each synthesis run from step 1 (design generation) to step 5 (power analysis) can take up to a few hours.

CHAPTER V

ARCHITECTURAL ESTIMATION

Most video processing designs have design parameters that have to be decided. The decision requires that we implement several versions of the design with different parameter values and look at metrics such as performance, area, memory size and bandwidth, etc. Doing all these designs may require a prohibitive amount of time. Hence, we have to pick a few parameter combinations. This is a chicken or egg problem. How can we pick parameter values and exclude others without knowing the merit factor of the ones we are excluding?

This dilemma can be partly solved by using "design generation tools" (custom and/or general-purpose) so that many designs with different design parameters are created quickly. This approach solves only part of the problem. We still have to verify and synthesize those many designs. Synthesis time growth can be addressed by again automation (or semi-automation) but verification takes too much time even with automation. Irrespective of whether we use design generation (RTL generators, HLS, etc.) or not, we need an architectural estimator to narrow down our choices in terms of design parameters.

Design parameters are fps, frame resolution, FPGA to use, clock frequencies to use, FPUs and other IPs to use and their parameters, power consumption, whether to use DRAM or not, DRAM size, DRAM bandwidth, how to stream the data in and out. On top of this, in our optical flow problem, the number of Warp() calls, Iter() calls per Warp(), pyramid levels, and the number Iter blocks are also parameters.

In summary, an architectural estimation tool is indispensable in video processing design as well as others that have an abundance of design and algorithmic parameters.

Although the tool will have to be design specific to a certain extent, some ideas can be borrowed from the discussion here.

Architecture exploration tool estimates 7 design metrics. Some of these metrics are estimated by modeling mathematically. Some metrics are where the relationship between input and output is quite chaotic. The approach in such metrics is to model the relationship between input and output by taking a large number of samples. The subtopics describe how the metrics calculated by the estimator tool are implemented.

5.1 Area Usage

One of the most important constraints on FPGAs is area utilization. The limiting factors in area usage are register, ALM, BRAM, and DRAM. Changes in the algorithm may result in not having enough resources. Synthetic processing is needed to understand whether the resources are available. However, the design must be completed to perform the synthesis process. Therefore, once the design is complete, it may not fit in the FPGA. In this case, the design may become completely or partially unusable. Development time will increase considerably.

A tool for predicting the use of area for an optical flow algorithm, before starting the design development process, according to certain parameters is presented. There are 6 main blocks running on the FPGA. These are given below.

- Interface Controller
- DRAM Controller
- Downscaler
- Warping
- Iteration
- Top Level Controller

The interface control logic varies according to the selected interface (Ethernet, PCIe etc.). However, the changes in the algorithm do not affect the area usage of the interface control logic. The area to be occupied by the interface logic is previously synthesized and obtained by lookup table approach. Also the DRAM control logic is not affected by the changes of the algorithm like the interface logic. The area used by the DRAM controller logic varies according to the DRAM type to be used. For this, DRAM controllers were synthesized and a loop-up table was created. The interface logic and DRAM controller occupied areas are defined as INA and DRA.

Downscaling at different ratio is done according to a parameter which is input to the module. It supports all necessary resolutions, it has a fixed structure. Once synthesized, it can be added to the result of area usage of the entire system. Downscaler unit area usage defined as DSA.

Warping and iteration module contain BRAMs for buffering requirement and calculation units. Calculation units operation type can be fixed point or floating point. These units also have different area usage according to their selected delays. Which unit to use depends on the sensitivity and performance requirements of the algorithm. The selected calculation units can be connected to each other to perform calculations at all cycles. However, this approach causes too many unit usage. So even a single iteration or warping module may not fit to the FPGA. For this reason, the iteration and warping units should be considered to be produced by resource sharing method. The disadvantage of this approach is that the inputs and outputs of a computing unit connected to multiple sources. Therefore, the multiplexing cost occurs in the inputs and outputs of the calculation units. Synthesis results were obtained on FPGA between 1-50 cycle time for analysis of the multiplexing cost. Fig. 27 and 28 shows the relationship between cycle time and multiplexing.

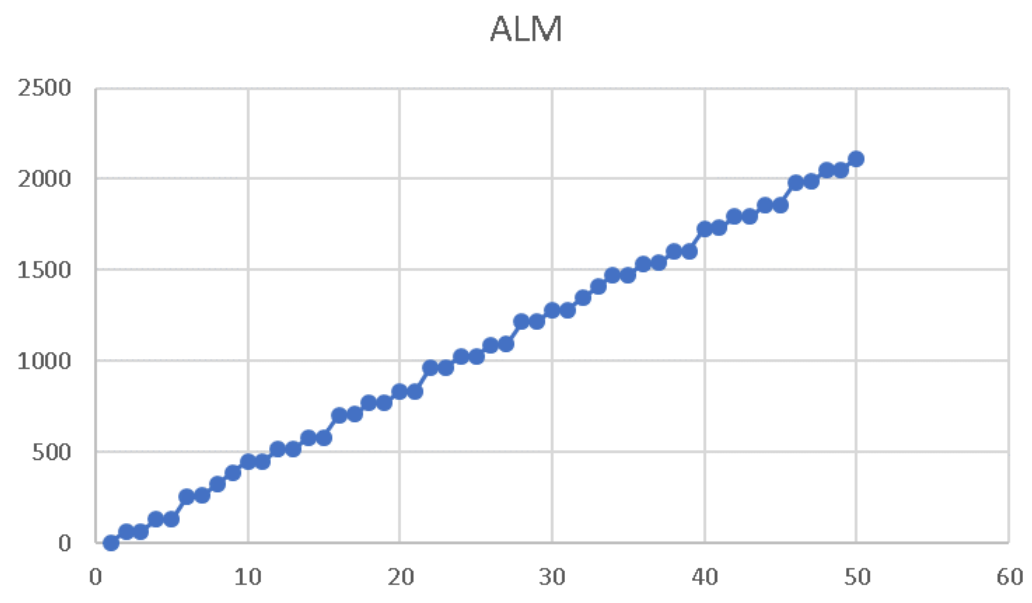

Figure 27: Cycle-Time 1-50 - ALM Relationship.

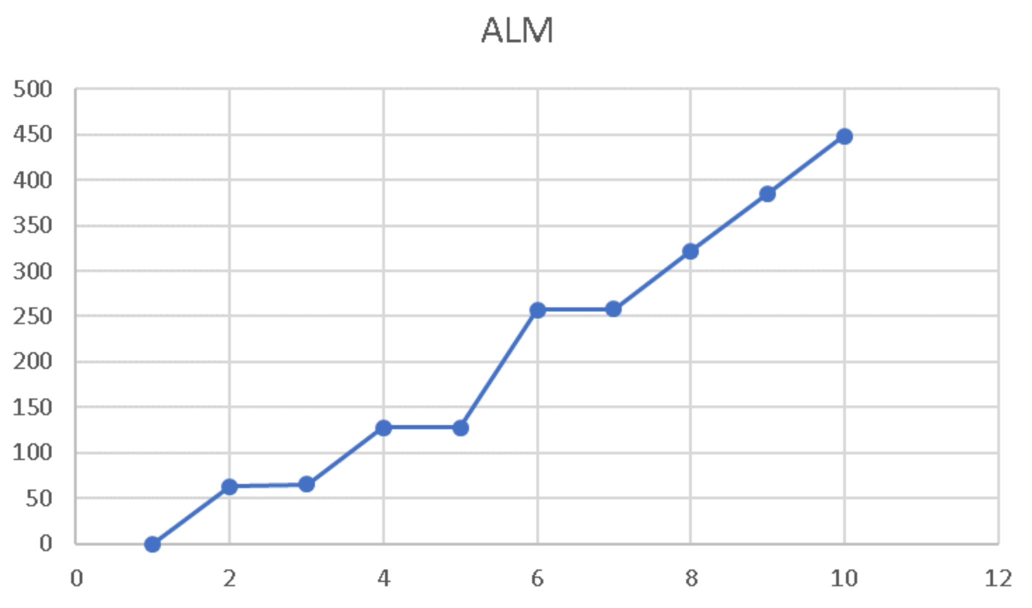

Figure 28: Cycle-Time 1-10 - ALM Relationship.

A look-up table was created with the synthesis results for calculation (1) of total number of ALM (TALM). This look-up table is defined as a multiplexing cost function.

$$TALM = MUXCOST(CT) * TCU + \sum_{i=1}^{TCU} FPU_{\text{i}}(ALM) \tag{1}$$

The total number of calculation units (TCU) multiplied by the multiplexing cost was found to be the total multiplexing cost. ALM usage of the used calculation units and total multiplexing cost are summed.

Register and DSP usage are calculated in the same way as ALM calculation (2, 3).

$$TREG = MUXCOST(CT) * TCU + \sum_{i=1}^{TCU} FPU_{\text{i}}(REG) \tag{2}$$

$$TDSP = MUXCOST(CT) * TCU + \sum_{i=1}^{TCU} FPU_{\text{i}}(DSP) \tag{3}$$

The number of BRAMs depends on the buffering requirement in the algorithm. The number of BRAMs required for buffering (4) is obtained by multiplying the number of rows needed (RC), the number of data in single row (SRC), the bit width of each data (BG) and the number of how many modules are used (MC).

$$TBRAM = RC * SRC * BG * MC \tag{4}$$

5.2 Throughput

There are too many parameters to decide even with an estimation tool. Usually, designers deal with one of the following two problems:

1. Given an fps and frame resolution, come up with the lowest cost or lowest power consumption solution. (It would be even better if some hardware choices are fixed.)

2. Given a particular hardware platform (a set of hardware choices), produce the highest fps or frame resolution (hence highest number of pixels processed per

second).

We focused more on Problem 2: "What is the highest fps we can obtain from a particular FPGA board?" Given this problem, our estimator does architectural exploration adjusting two parameters, namely, pixel cycle-time and number of Iter blocks in Iters Pipe. It takes cycle-time as a free parameter, and based on that, calculates the number of FPGA resources an Iter block requires (logic cells, flip-flops, BRAMs, DSP slices). From that, it calculates how many Iter blocks it can fit on the given FPGA. The deciding factor in terms of the number of Iter blocks can be any of the FPGA resources. Then, it calculates how many times we have to circulate over Iters Pipe.

For the fps calculation (5), clock frequency (f), Cycle-time (C), Loop Count (LC), frame Resolution (R), Pyramid factor (P) parameters are used.

We estimate f by synthesizing one Iter block. L is the number of times we have to circulate around the loop. P is, on the other hand, the ratio of the total number of pixels in all the levels of the pyramid to the pixels in one frame. P is greater than 1.

$$fps = \frac{f}{C * LC * R * P} \tag{5}$$

where

$$LC = \left\lceil \frac{i}{I} \right\rceil \tag{6}$$

and

$$I = min_{j \in T} \left\lceil \frac{A_j}{U_j} \right\rceil \tag{7}$$

$$P = \sum_{k=0}^{l-1} d^k \tag{8}$$

LC is computed (6) by dividing the number of Iter() calls (i) by the number of Iter blocks (I). Since it has to be an integer, we have to take the ceiling of the division. I is the number of Iter blocks we can fit, which can be decided by dividing the total available amount (A_j) of every hardware resource type (j) in the set of Types (T) by the amount needed by one Iter block (U_j) (7). The hardware resource types are logic slices, DSP slices, registers, RAM blocks, and DRAM bandwidth. P can be computed by a summation of ratios (8), where l is the number of levels and d is the downscale ratio.

There is a trade-off between cycle-time and the number of Iter blocks as well as the DRAM bandwidth. The lower the cycle-time, the bigger an Iter block, hence the fewer the Iter blocks we can fit on the FPGA and the larger the DRAM bandwidth. Due to the DRAM bandwidth available, we may have to insert bubbles in the pipeline. You may think of it like a wait time between consecutive reuses of the pipeline. Although lower cycle-time increases the fps, fewer Iter blocks lower the fps. In addition, the lower the cycle-time, the more we circulate around the pipe and hence eventually hitting the maximum available DRAM bandwidth and hence limit the fps we can achieve. Because of all these factors, the most ideal cycle-time may be somewhere in the middle.

If we try cycle-time values one by one, and we try to narrow down the number of Iter blocks, we do not need to estimate the clock frequency of Iter. That is because to maximize fps, we better fit as many Iter blocks in the FPGA as possible, and that depends on the area of Iter.

However for Problem 1 above, we better estimate the clock frequency of Iter. That is due to the fact that whether we meet our fps target with a certain number of Iter blocks depends on the clock frequency. We estimate the clock frequency by just synthesizing one Iter instead of the complete design, which takes a few minutes as opposed to 30 minutes. The clock frequency that comes from this synthesis can be $\pm 25\%$ of the actual frequency based on data we collected. Therefore, whatever number of Iter blocks we calculate from the estimated clock frequency, it may be sufficient to try the number of Iter blocks that are within $\pm 25\%$ of that.

5.3 Latency

The optical flow algorithm produces the output vectors by processing the input pixels into a long pipeline. The length of this path depends on parameters such as DRAM usage, number of iterations, and delays of selected FPUs. It is important to know the total latency of the system. If the system has a low latency requirement, it needs to know what the latency will be before the design development starts.

If there is no need for DRAM in the algorithm, pixels will only pass from the calculation units. The data coming from the interface is buffered in BRAM, the whole video frame and the 2 lines 2 pixels of the second video frame. The downscaler unit does not need to wait for this buffering. However, even if the incoming pixel is immediately downscaled, the algorithm must wait for the second frame to be downsized. When the algorithm includes pyramid (P) design, the algorithm can not start without getting the smallest pyramid of the second frame. The downscale unit performs calculations based on the output time of first input (OFI), cycle-time (DSC), resolution (R) and, downscale ratio (DR). Video frames are coming in consecutively from the interfaces. For this reason, the second frame comes after all pixels of the first frame's arrives. In order for the algorithm start to execution, the lowest pyramid level of current, and previous images must be calculated. Calculation of all levels of the first image and calculation of all the pyramids except the final level of the second image is required. After 2 lines and 2 pixels of the pyramid at the end of the second frame are ready, the algorithm can start to execution. The number of elements in a row in the pyramid is defined as PRC. The delay (DSL) caused by the downscale operation is calculated with DSC, pyramid factor, and OFI (9).

$$DSL = (DSC * (\sum_{i=0}^{P-1} DR^{\mathrm{i}} + \sum_{i=0}^{P-2} DR^{\mathrm{i}}) + 2 * PRC_{(\mathrm{P\text{-}1})} + 2) + OFI \tag{9}$$

In the next step, the pyramid frames are taken by the warping and iteration modules. The these modules are considerably larger than the downscaling module. Warping and iteration modules may have different design alternatives. Different designs may arise depending on the scheduling of FPU units, cycle-time, and how much line buffering needs. The warping module must buffer the line (WB) as many lines as it will support the warping. Each iteration module can start to work after 1 row pixel (IR) + 2 pixel buffered. After the warp operation is performed in the algorithm, a certain number of iteration operations are performed. This process is repeated until the desired number of iterations (IC) is reached. The loop count (LC) specifies how many times iter-warp modules will executed. The latency (see 10) of iteration and warping (IWL) is modeled with the DR parameter for all pyramidal levels.

$$IWL = CT * WB * LC * IC * (IR + 2) * \sum_{i=0}^{P-1} DS^{\mathrm{i}} \tag{10}$$

When the DRAM is used in the algorithm, it is used most intensively to store the vectors generated by the iteration modules. Delays are caused by waiting (DRW) while accessing new rows in DRAM. When the required number of iteration module does not fit into the FPGA, it is necessary to loop the data on the iteration modules. This number of loops (LC) indicates how dense the access to DRAM. The delay of DRAM (DRL) is when the iteration modules, which are not consecutively accessed to RAM, generate vectors (IV) and receive it again from DRAM. It must be multiplied by 2 for both reading and writing, and modeled for all pyramid levels (11).

$$DRL = LC * IV * 2 * DRW * \sum_{i=0}^{P-1} DS^{\mathrm{i}} \tag{11}$$

Latency due to the interface vary according to the selected interface. This parameter is called interface delay (ID). The total latency of the system (STL) is calculated by summation of SDL, IWL, DRL, and ID (12).

$$STL = DSL + IWL + DRL + ID \tag{12}$$

The STL gives the total delay of the system in cycles. Latency in terms of seconds can be found by dividing the STL to the frequency (FR) of system will execute (STLT) (13).

$$STLT = STL/FR \tag{13}$$

5.4 DRAM Bandwidth and Area

The use of DRAM for optical flow algorithm is very important. The algorithm needs to store the generated vectors as intermediate values. Depending on the resolution and the number of iterations, the vectors are often too large to be stored in the Block RAMs on the FPGA. Even if the DRAM area meets the needs of the algorithm, the bandwidth needs to be considered. Bandwidth must be adequate when accessing DRAM to achieve the desired output rate. In the algorithm there are many blocks that need access to DRAM. These are listed below.

- Video streaming through PCIE
- DRAM Controller
- Downscaler

5.4.1 DRAM Area Usage

In order to be able to start the optical flow algorithm, it is necessary that 1 video stream is stored in each case and the first 2 lines and 3 pixels of the next video stream have arrived. This causes an inevitable buffering requirement.

DRAM area requirements are depends on number of iterations (IN), iterations that fits on FPGA (IFF), pyramids (P), BRAMs (BR), required iteration count (IRC), and downscale ratio (DR).

Firstly, situations that do not need DRAM are examined. If the inevitable input frame buffering requirement in the algorithm can not be met by the block RAMs on the FPGA, DRAM must be used. If it can be buffered, the other parameters need to be checked.

The CPU transfers the raw image through the interface to the FPGA. The optical flow algorithm starts from the smallest of the pyramid levels. The downscaler module takes the original image and downscales it. However, as the algorithm continues to execution, it emerges towards the upper layers of the pyramid. So all the frames need to be stored. We call that storage needing to the required buffer area (RBA) (14). The resolution (R) is previous full frame, 2 resolution lines, and 2 numbers (RS) indicate the start of the current.

$$RBA = (R * 2 * RS + 2) * \sum_{i=0}^{P-1} DS^{\mathrm{i}} \tag{14}$$

If the images to be stored are smaller than the FPGA's existing block RAM (BR), the first requirement of the algorithm can be achieved without DRAM. Frame fit checking (FFC) to the FPGA is made for DRAM usage decision (15).

$$FFC = RBA < BR \tag{15}$$

It is necessary to examine the units with other DRAM access in the algorithm. The output U and V vectors of the last iteration module can be fed directly into the output interface, when required iteration (RI) number smaller than the number of iterations that can fit (FI) in the FPGA. Then there is no need for DRAM usage. This iteration loop control (ILC) is comparison of RI and FI (16).

$$ILC = RI < FI \tag{16}$$

The number of required iterations may be greater than the number of fitted iteration modules in the FPGA. In this case, the data on the iteration modules need to be loop on themselfs. The sum of number of fitted iteration modules in FPGA multiplied by iteration module latency and number of BRAM bytes exists on FPGA (BF). If this sum is higher than the resolution, there is no DRAM requirement (17).

$$DR = FI * IL + BF > R \tag{17}$$

In all other cases, DRAM must be used. In these cases, it is necessary to pay attention to the bandwidth of the DRAM as well as the area. In cases where DRAM use is mandatory, it is of course possible to retain some of the data in DRAM at the BRAM. If enough DRAM bandwidth is available, all data can be held in DRAM to reduce the use of BRAM. The same DRAM space can be used for each level of the pyramid. Because after the vector transfer between the pyramids, the vectors in the previous pyramid are not used again. So the area to store the vectors in the largest pyramid level will be sufficient for the whole algorithm. The DRAM area (DRA) needed by the iteration calculated with parameters, resolution and iteration module output vector number (V) multiplied by bit width of each vector (18).

$$DRA = R * \sum_{i=1}^{V} Byte(Vector_{\text{i}}) \tag{18}$$

The total area (DTA) required in DRAM is calculated with summation of RBA and DRA (19).

$$DTA = RBA + DRA \tag{19}$$

5.4.2 DRAM Bandwidth Usage

To calculate the DRAM bandwidth (DBI), it is necessary to find out how many times the iteration modules must loop on their own. The number of iterations required is divided by the number of fitted iterations in the FPGA. The number of loops count (LC) indicates how many times the vectors from the iteration modules need to be write to and read from DRAM. This writing and reading is done for every pyramid level and for each frame. In Throughput section, the maximum FPS calculated is multiplied by LC and result is the amount of access bytes required in one second. The result is multiplied by 2 because both reading and writing are done (20).

$$DBI = DRA * LC * FPS * 2 * \sum_{i=0}^{P-1} DS^{\mathrm{i}} \tag{20}$$

Other factors affecting the DRAM traffic are the raw images sent by the CPU and the output frames of the downscaler module. We call it DRAM bandwidth (DBF) usage by total frames. The resolution is multiplied by FPS to find how many frames should be written at the moment. This value is multiplied by 2 because of both reading and writing are done (21).

$$DBF = R * FPS * 2 * \sum_{i=0}^{P-1} DS^{\mathrm{i}} \tag{21}$$

The algorithm's total DRAM bandwidth requirement (DBT) is calculated by adding DBI and BBF (22).

$$DBT = DBI + BBF \tag{22}$$

If the DRAM model to be used is already determined, several methods can followed to determine if the developed tool estimates that system can not work with a single DRAM. If limit is area usage, then tool finds the minimum number of DRAM chips required. If there is a limit on the bandwidth, the FPS can be reduced or the minimum number of DRAM chips is specified.

When the number of DRAM chips is increased, area, and bandwidth are increasing linearly. Accordingly, the estimator specifies the minimum number of DRAM chips required (MCR). The smallest number N that provides the formula is the number of chips required (23, 24).

$$DTA < \sum_{i=1}^{N} MCR_{\text{area}} \tag{23}$$

$$DBT < \sum_{i=1}^{N} MCR_{\text{bandwidth}} \tag{24}$$

After picking up the smallest chip counts required and for the bandwidth and area, whichever is the bigger is selected (25).

$$MCR = MCR_{\text{area}} > MCR_{\text{bandwidth}} ? MCR_{\text{area}} : MCR_{\text{bandwidth}} \tag{25}$$

5.5 Interface Bandwidth

The optical flow algorithm inputs video frames and produces two U-V output vectors. In order to feed the input frames and retrieve the output vectors, an interface is needed to talk with the optical flow module. This interface is necessary to ensure that the optical flow can be carried out by the host CPU. The bandwidth requirement is determined by which interface should be used. The interface bandwidth depends on the parameters required for throughput calculation. The bandwidth is related to the maximum FPS calculated in the throughput section. The input bandwidth (IB) is expressed (26) as the number of frames per second (FPS) multiplied by the resolution (R).

$$IB = R * FPS \tag{26}$$

The output vectors U and V are output. The output bandwidth (OB) requirement is found by using these vectors, FPS, and R values (27).

$$OB = Byte(UandV) * FPS * R \tag{27}$$

Once the input and output bandwidths have been calculated, the choice of interface can be made. In full duplex systems, the interface can be selected according to which input or output is greater bandwidth requirement. The total bandwidth must be calculated for the selection of interfaces using the same path for reception and transmission. The total interface bandwidth (TBI) is used to measure the requirements of the interface (28).

$$TBI = IB + OB \tag{28}$$

5.6 Temperature Estimation

It is very important to predict at what temperature the FPGA design will work. In order to avoid damage to the FPGA, it is necessary to stay under a certain temperature. Cooling precautions may be taken if necessary. If the target temperature can not be reached after the design is completed, very likely the targeted clock frequency will not be unusable.

The temperature depends on the parameters of the algorithm and other factors. Algorithm changes affect the total captured power. The temperature varies (29) depending on the amount of power (PW) drawn. The temperature (TJ) of the FPGA is related to the thermal impedance (TA), Die to Package (DP) and Package to ambient (PA).

$$TJ = TA + (PW * (DP + PA)) \tag{29}$$

The Acura AC 9270 infrared temperature meter was used to obtain the FPGA temperature values. Fig. 29 shows the acquisition of the measurement.

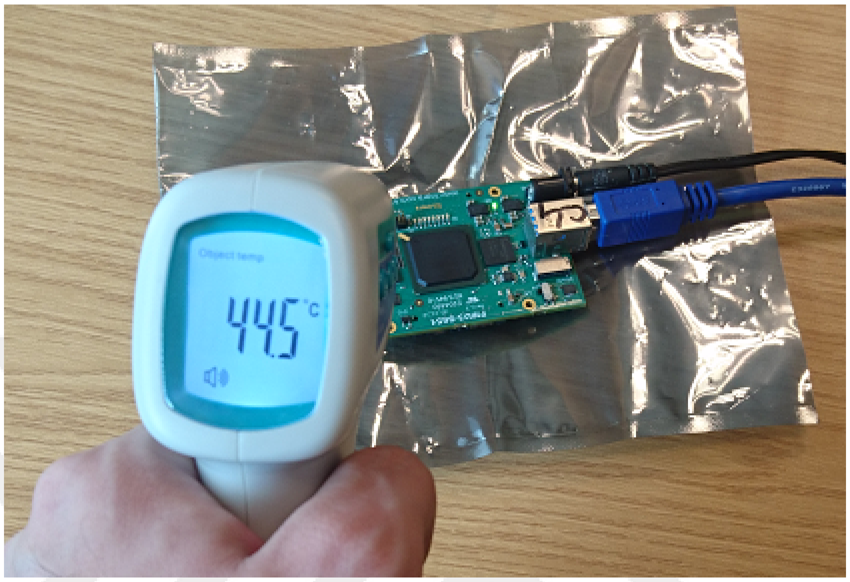

Figure 29: Temperature Measurement of FPGA.

5.7 Compilation Time

The compilation time (CMT) is very important when the design on the simulation is completed and is being tested on the FPGA. When the error is encountered, it has to be synthesized again between each iteration and a certain time has passed. Knowing this time in advance is necessary for giving information about how long the work can last.

In the optical flow algorithm, the iteration unit occupies the most area and imposes frequency limits. As the number of iteration units increases, compilation becomes more difficult, and takes more time. Also the synthesis time varies according to the FPGA systhesis tool. The extraction model is given for compilation time. Eureqa[68] named data mining software was used to extract the formula (30).

$$CMT = 54.0 + 1.6 * iter^2 - 5.6 * CT \tag{30}$$

CHAPTER VI

RESULTS

In Subsection 6.1, we will first present our optical flow designs validated on FPGA. Then, in Subsection 6.2, we will discuss estimates our Estimator produced, and how we do design exploration using them. Next, in Subsection 6.3, we will share some synthesis results where MAFURES and Vivado HLS are compared. Last but not least, in Subsection 6.4, we will present some performance and power consumption results on CPU (Matlab and OpenCL), GPU, FPGA (OpenCL and Verilog).

6.1 Designs Fully Tested on FPGA

We have applied our tools and techniques to synthesize over 3000 optical flow designs. We fully verified and then tested 11 of them on the FPGA in the order they are listed in Table 6. In these 11 designs, we aimed to utilize all three boards we had that were suitable for the optical flow design. Also, we wanted to bring up several versions of the design as long as they fit on the particular board. Note that we also applied the tools and techniques we created in the process of doing the optical flow design to an "image fusion" [69, 70] design and fully verified and tested 3 of them.

These designs on the average contain around 15 thousand lines of Verilog code, 500 lines of C++ Host code, and in addition, vendor-specific FPGA IP files.

The design in the top row in Table 6 took 3 people 1 year. On the other hand, the design in the bottom row took 2 people 10 days. This reduction in development time is mostly due to all the tools and techniques we developed along the process. However, the experience gathered in the first year by the team was also a factor (though secondary) in this time reduction.

These two people were the team that made the first design in one year. However, other developers can extract designs rapidly by using developed tools.

This was possible although this last design was much bigger and more complicated.

In Table 6, the leftmost column shows the hardware platform used. We implemented our designs on 3 different boards. The second column shows the number of Iter blocks in Iters Pipe. All designs have 1 Warp block in the pipeline. The rows with a single number in this column indicate we do not circulate around the pipeline. Therefore, the number Iter() calls is equal to the Iter blocks. 50x10 indicates we have 1 Warp and 50 Iter blocks in the pipeline and we circulate around the pipeline 10 times.

50x10x3 means we have a pyramid with 3 levels. That is, we effectively circulate around the pipeline 30 times. (All other designs have 1 pyramid level.) This design contains around 30 thousand lines of Verilog besides some IP files. Only 2,000 lines of this code is in common with the other designs. 6,000 lines of it comes from MAFURES. 20 thousand lines of it comes from our Pipe Generator. And 700 lines of it is design-specific code we manually wrote. This last design used a MAFURES generated Warp module. Previous designs (in the rows above) used a manual design we did. MAFURES boosted the clock frequency of Warp from 60 MHz to over 150 MHz. The clock frequency in this case was decided by the wrapper logic of the Iter block. This attests to the quality of the designs done by MAFURES.

Our Image Fusion designs are shown in Table 7. The design in the top row contains 7 thousand lines of Verilog plus C++ each (plus some vendor-specific IP files). 3,000 lines come from MAFURES. 1,500 lines came from Optical Flow. 500 lines come from code generators specific to Image Fusion. 2,000 lines are manually written code specific to the design. The other two designs have quite similar statistics. In Table 7, C stands for "cycle-time". C for all our optical flow designs is 10 clock cycles.

On the verification side, we used around a total of 2,000 lines of Verilog, Java, and Matlab for Optical Flow, while we used around 1,000 lines of a combination of

Verilog and C for Image Fusion.

The above shows that we achieved a high degree of design automation, which is not only applicable within Optical Flow but also applies to another design, namely, Image Fusion. We are confident that many other video and image processing designs can benefit from our work. Recently, we applied our PCIe framework to a wireless communications design in a very short period of time.

6.2 Architectural Estimation Results

Table 8 presents some estimation and synthesis results for Problem 2 mentioned in Subsection 5 when our Cyclone IV board is targeted. The 3 rightmost columns are the outputs of our Estimator (see Subsection 5). The other columns except for the leftmost are results obtained after synthesis. The leftmost column is the cycle-time (C), and we have a separate set of results for every C value. Our MAFURES HLS tool has two switches, namely, Mux-aware (Ma) synthesis on/off and Register sharing (Rs) on/off. Therefore, we have 2x2=4 combinations in Table 8. Columns I (short for #Iter^s) list the number of Iter blocks we can fit on the FPGA for each combination. Columns fps list the max fps can be achievable with the I at hand (5 in Subsection 5). DL stands for "DRAM Limited" and means if I (7) is determined by the DRAM bandwidth.

The max fps estimated by the Estimator is shown in the fps column on the right for every C. Note that the Estimator does not have a way of estimating different numbers for different combinations of Ma and Rs switches.

The max fps possible is proportional to the max pixels processed per second, which is inversely proportional to C. However, since every Iter block processes a pixel in parallel, it has to be multiplied by the number Iter blocks that can be crammed into the FPGA (I).

Here is the process through which we identify the design with the highest fps. We estimate using the Estimator the max fps for different Cs starting from 1 and going up one by one. We see that it makes a peak at C=3. We then continue incrementing C and by C=7, we see that it is consistently dropping. Hence, there is no need to consider larger Cs. For every C, the Estimator also reports the number of Iter blocks that fits on the FPGA (I). We then synthesize for that C and 5 different I values, that is, the I reported by the Estimator±2. After synthesis, we see that the I that fits is actually within ±1 of the estimated I. In Table 8, we report the max I values we obtained for each case. The results of the synthesis and the estimated results are close to each other.

In summary, the Estimator limits our architectural exploration search space. Within that search space, we synthesize every thing and go with the best combination. And that is a C of 3, which allows for an I of 5, resulting in an fps of 0.71.

Table 6: Optical Flow Designs Tested on the FPGA.

Platform	#Iter^s	fps	Resolution	f (MHz)
PCIe 1.0 Altera Cyclone IV	1	20.0	576x768	30
	2	20.0	576x768	30
	3	20.0	576x768	30
USB 3.0 Xilinx S6	1	5.1	576x768	15
	2	5.1	576x768	15
PCIe 3.0 Altera Arria 10	1	19.5	640x480	60
	2	19.5	640x480	60
	25	19.5	640x480	60
	50	19.5	640x480	60
	50x10	1.9	640x480	60
	50x10x3	3.7	640x480	150

Table 7: Image Fusion Designs Tested on the FPGA.

Platform	C	fps	Resolution	f (MHz)
PCIe 1.0 Altera Cyclone IV	1	15	640x480	73
	3	40	640x480	95
PCIe 3.0 Altera Arria 10	1	86	1920x1080	188

Table 8: Optimizing fps Using Architectural Estimation.

	Synthesized								Estimated		
			Ma		Rs		Rs + Ma				
C	I	fps	I	fps	I	fps	I	fps	I	fps	DL
7	6	0.47	6	0.46	6	0.45	6	0.46	5	0.37	No
6	6	0.53	6	0.53	6	0.51	6	0.49	5	0.43	No
5	6	0.63	6	0.63	6	0.59	6	0.62	5	0.49	No
4	5	0.66	5	0.70	5	0.67	5	0.68	4	0.62	No
3	5	0.71	5	0.71	5	0.71	5	0.71	4	0.71	Yes
2	3	0.43	3	0.43	3	0.43	3	0.43	4	0.57	Yes
1	2	0.28	2	0.28	2	0.28	2	0.28	2	0.28	Yes

6.3 MAFURES and Vivado HLS Compared

Our MAFURES HLS is possibly the most critical component in our design process. However, it can be replaced by other HLS tools. We tried multiple alternatives but the only other HL tool that was user-friendly enough was Vivado HLS from Xilinx [50], and its main drawback is that it supports only Xilinx FPGAs. (However, it is conceptually possible to use it with other FPGAs through wrappers.)

MAFURES is FPGA vendor agnostic. MAFURES spits out shorter and more readable code, which improves simulation and debugging time. In addition as we show in Table 9, we were able obtain favorable results for MAFURES when we targeted a Xilinx Kintex-7 FPGA for the core of Iter block. Nevertheless, the drawback of MAFURES is that it accepts lower-level code with respect to Vivado HLS. Also, it does not do function unit selection, you have to specify which function units it has to use.

When we ran "timing-driven" synthesis, MAFURES obtained 10% better timing at the expense of around 5% more area with 50% of the Verilog lines in Vivado HLS. Timing-driven synthesis in MAFURES means Mux-aware synthesis with no Register sharing (i.e., Ma option in Table 8). For Vivado HLS, timing-driven implies giving an almost impossible timing goal, so that the tool does the best it can.

When we ran "area-driven" synthesis, MAFURES obtained around 10 to 15% better area at the same timing with one third of the Verilog lines in Vivado HLS. Area-driven synthesis in MAFURES means no Mux-aware synthesis with Register sharing (i.e., Rs option in Table 8). For Vivado HLS, area-driven implies giving a very relaxed timing goal, so that the tool focuses on area minimization. In our case, we first did the synthesis with MAFURES and gave Vivado HLS the timing we achieved with MAFURES (3.7 ns).

Table 9: MAFURES and Vivado HLS Comparison.

Driven by	HLS Tool	Clk Per.(ns)	Verilog Lines	Logic Slices	DSP Slices
Timing	MAFURES	2.8	1453	2899	6
	Vivado HLS	3.1	2966	2719	6
Area	MAFURES	3.7	1112	1893	17
	Vivado HLS	3.7	3362	2203	19

6.4 Performance and Power Results

In this subsection, we compare FPGA results to GPU and CPU results. We also present results for Altera Offline Compiler (AOC) [71], which synthesizes OpenCL code on to Altera FPGAs. Except for Table 13, the FPGA is Altera Arria 10, GPU is Nvidia GeForce GTX 980 Ti, and the CPU is Intel Xeon 1650 v3, which are all high-end in their own domain. In Table 13, the FPGA is Altera Cyclone IV and GPU is AMD Radeon HD 7870M; they are mid-range in their categories.

In terms of development time CPU based development is fastest, then comes GPU, then OpenCL on FPGA, and HDL based development on FPGA takes the longest time. And this work aims to reduce the handicap of HDL based design of FPGA implementations. However, even with tools and techniques here, Verilog/VHDL design takes longer. It is the goal of this subsection to show that traditional FPGA design with HDL offers superior metrics in terms of power consumption to all other options including GPU, and yet it can more or less match GPU performance.

Table 10 shows the "frame time" achieved by 5 different implementations of AHL1 algorithm. Frame time is equal 1/fps. In Table 10, we have 2 CPU implementations. The one on the left is the original Matlab implementation. The FPGA implementation on the left is the design we discusses throughout this dissertation. AOC is also, in a way, an FPGA implementation. AOC tool implements a GPU architecture on the FPGA with custom processors. Looking at the table, the ranking of the implementations from slowest to fastest are as follows: Matlab on CPU, OpenCL on CPU, OpenCL on FPGA (AOC), OpenCL on GPU, and Verilog on FPGA. Note that OpenCL on CPU allows for easy exploitation of the multiple cores on the CPU.

Table 10: CPU, GPU, AOC, and FPGA Frame Times.

Resol.	Matlab	OpenCL			
	CPU	CPU	GPU	AOC	FPGA
576x384	107.764 s	8.645 s	0.128 s	2.734 s	0.151 s
640x480	156.983 s	9.612 s	0.168 s	3.594 s	0.210 s
768x576	471.542 s	11.277 s	0.254 s	4.854 s	0.302 s

Table 11 compares the power consumption of OpenCL on CPU, GPU, and AOC solutions. All solutions execute 10 warps at 50 Iter() calls per warp without pyramid (50x10 in Table 6). We measured power consumption with "Edimax WiFi Smart Plug with Energy Management (SP-2101W)". For all configurations, we subtracted the power consumption at "Idle". In this table, we slowed down the GPU so that it matches the frame time (hence the fps) of the AOC solution. Comparing two systems running at different fps does not make sense. A system consumes more power when run at higher fps. Nonetheless, we ran the CPU at its own lower frame time. We could slow down the AOC and GPU implementations to match the CPU but we thought that would be artificially slow. Even the existing table gives us a clear picture, which is AOC is quite superior in power consumption, while GPU and CPU are comparable. If AOC could be run at much faster pixel throughputs, GPU could maybe beat it in power consumption.

Table 11: CPU, GPU, AOC Power Results.

Resol.	fTime	AOC	GPU	fTime	CPU
576x384	2734 ms	15 W	58 W	8600 ms	51 W
640x480	3294 ms	19 W	58 W	9600 ms	51 W
768x576	4854 ms	22 W	60 W	11200 ms	52 W

In Table 12, the FPGA (Arria 10) is 3x more power-efficient than the GPU (GTX 980 Ti). Looking at the way they scale (which is linear), it seems like the relationship will stay as 3x. These runs were such that there is no circulation. That is, we have 50 Iter() calls in the bottom row for example, not 50x10. However, they all run at 12.5 fps. That is, the implementations are slowed down in the case of lower number of Iter() calls. FPGA, in this case, is not superior in one metric, and that is the price. Although the two chips here (GPU and FPGA) are similar in complexity, since such high-end FPGAs do not sell in volume, they are more expensive compared to their GPU counterparts. However, that divide diminishes in the case of mid-range GPUs and FPGAs. Table 13 shows such a case with Altera Cyclone IV FPGA and AMD Radeon HD 7870M GPU. As can be seen, the FPGA is much superior in power consumption when all run at the same fps.

Table 12: High-end GPU-FPGA Power Results at 12.5 fps.

#Iter^s	GPU	FPGA
1	70 W	25 W
2	71 W	25 W
25	80 W	27 W
50	90 W	30 W

We were able to show through this project that we are able to achieve lower power consumption with an FPGA solution (whether HDL based or OpenCL based) as compared to GPU for a given performance (pixels or frames per second).

Table 13: Mid-range GPU-FPGA Power Results at 6 fps.

#Iter^s	GPUmid	FPGAmid
1	36 W	1.4 W
2	37 W	1.5 W
3	38 W	1.5 W

When all the results of the tables are examined together, it is seen that the FPGA performs much better than other platforms (as expected). Since there is no other work in the literature to realize the same design, the different parameters of the same design and the designs on different FPGAs are compared with each other.

Table 10-13 contain performance comparisons. In all comparisons, although the CPU draws power close to the GPU, its performance is very low, and the FPGA is seen to have higher fps, though consuming much less power.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

FPGAs can offer throughputs in video processing almost as high as GPUs but at lower power consumption. Nevertheless, the development man-hours they need are significantly more than what GPUs need. Therefore, FPGA world needs a design productivity boost. Part of it comes from HLS. However, we need a variety of tools, some of which may be design specific. Using OpenCL synthesis tools for FPGAs may address the development time issue but it does not offer performances anywhere close to GPUs and HDL-based FPGA designs.

This dissertation presented an FPGA design framework that addresses these issues. In addition, we created a quite general-purpose HLS tool, called MAFURES, which is FPGA vendor agnostic and has some advantages over Vivado HLS, a tool that is currently making quite a positive impact in the industry as well as academia. We also explained in some detail our high-level and general-purpose framework to supporting interfaces such as PCIe and USB as well as DRAM. Our work builds on top of RIFFA in the PCIe domain, which is also quite useful and popular.

The dissertation also discussed other tools (e.g., Architectural Estimation and Flow Verification) and shown techniques (e.g., 4 levels of Nested Pipelining), which can be ported to other designs with some modifications. With the Architectural Estimation tool, we can uncover 8 design metrics before the design is completed.

We applied all these tools and techniques to multiple subblocks in our Optical Flow design as well as a second design, Image Fusion. The dissertation showed that in a fairly short amount of time, we were able to implement 11 versions of the optical flow algorithm running on 3 different FPGAs (from 2 different vendors), while we generated and synthesized several thousand designs for architectural trade-off.

The tools and methods developed in this study show that the ability to apply 2 different image processing is applicable to other image processing algorithms. It is not just the implementation of two algorithms, it is used in the implementation of many different versions of these algorithms.

The deep pipeline structures used can be applied to many designs. We used pipelining at frame, iteration, pixel, and arithmetic unit level. We can say that this methodology can be used in other image processing algorithms without the need to make changes. However, this structure can not be used in state machine type processing designs. The designer can design simple algorithms with the state machine method, but the pipelining method is life-saving when operations are complicated.

Some of the tools and designs developed can be used for general purposes, while others are dependent on the optical flow algorithm. Now we will talk about how these tools can be used in other video processing applications.

First, we will describe the HLS tool used to schedule arithmetic operations. Each video processing algorithm has calculation blocks that must be scheduled. If a fixed design is not targeted and the algorithm is likely to change over time, the scheduling process must be done with a tool. HLS tool named MATURES, it needs this. No modification is required for use in any other video processing algorithm. It is enough to feed the algorithm to the MAFURES.

Flow verification methodology can be applied to many designs. As the design grows, it facilitates verification. The flow verification tool developed with this methodology is somewhat dependent on the optical flow algorithm. This dependency is due to the specific design of the data transfer timing between the modules. With this approach, a person who wants to verify the design must first decide on the timing that should be in his design. Verification should be done with these timing values. The biggest problem here is that it is very difficult for any tool to predict the behavior of the design. So, without trying to predicting, verification is done according to the specifications given by the designer.

Interfaces that allow the design to talk to the outside of the chip are suitable for applications involving streaming. We believe that it will be suitable for many other application types as well as application video processing applications including streaming. A variety of communication layers are designed on the backplane for the operation of this interface provided to video processing designers. These two interfaces have been used in the demonstrations and the developed layered structure does not need to be modified for other designs.

The architecture estimation tool, which we consider to be very important, has some dependencies on the optical flow algorithm. In our estimation approach, there are the number of iteration and loop count parameters that are specific to optical flow. Our approach is to make an estimation over the biggest unit in the design of the video processing algorithm. The largest processing unit in optical flow is the iteration unit. So instead of examining the whole design, this unit was examined. In other image processing algorithms, first, the most complex units in the algorithm should be determined. This unit should be synthesized and analyzed. If the algorithm needs to use this unit more than once, the estimation is very useful. Relevant approaches can be used by replacing the largest unit with the iteration described in the estimation section.

The CPU, GPU, and FPGA comparisons given in Results section. These comparisons provide an opportunity to analyze the value of different platforms, such as development time, power consumption. In other applications, these platforms may resemble each other. It is therefore important to examine the results for different applications.

The code generation technique plays an important role when we want to design very quickly. Code generation is done in Perl, a scripting language. And it would be very useful to use this method in parameterized applications.

Finally, various IPs were developed which we think can be used in other applications. The internal structures of these IPs are given in the optical flow hardware section. These IPs can be used without being modified, if they can meet the needs of other applications.

As always, there is a lot of room for improvement in this dissertation. For example, we need to work on more use cases. Based on the experience with these new use cases, we can especially improve flow based verification and architectural estimation approaches. It should be possible to apply a data mining approach to architectural estimation. There is also need for improving our HLS tool; more comparisons are needed between our tool and mainstream HLS tools. Last but not least, nested pipelining could prove very useful if more use cases were studied.

Bibliography

[1] E. A. Snow, D. P. Siewiorek, and D. E. Thomas, "A technology-relative computer-aided design system: Abstract representations, transformations, and design tradeoffs," in *Proceedings of the Design Automation Conference (DAC)*, pp. 220–226, 1978.

[2] H. W. Lawson Jr, "New directions for micro and system architectures in the 1980s," in *Proceedings of the National Computer Conference (NCC)*, pp. 57–62, 1981.

[3] S. Director, A. Parker, D. Siewiorek, and D. Thomas, "A design methodology and computer aids for digital VLSI systems," *IEEE Transactions on Circuits and Systems*, vol. 28, no. 7, pp. 634–645, 1981.

[4] H. F. Ugurdag and T. E. Fuhrman, "Autocircuit: A clock edge general behavioral synthesis system with a direct path to physical datapaths," in *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 514–523, 1996.

[5] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[7] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 531–534, 2012.

[8] H. F. Ugurdag, "Experiences on the road from EDA developer to designer to educator," in *Proceedings of the East-West Design & Test Symposium (EWDTS)*, pp. 308–301, 2013.

[9] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Korner, and W. Eckert, "HIPAcc: A domain-specific language and compiler for image processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, 2016.

[10] M. A. Ozkan, O. Reiche, F. Hannig, and J. Teich, "FPGA based accelerator design from a domain-specific language," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 1–9, 2016.

[11] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne, "Making domain-specific hardware synthesis tools cost-efficient," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pp. 120–127, 2013.

[12] R. Wang, T. J. Hamilton, J. Tapson, and A. van Schai, "An FPGA design framework for large-scale spiking neural networks," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 457–460, 2014.

[13] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martnez, and C. Guestrin, "Graphgen: An FPGA framework for vertex-centric graph computation," in *Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 25–28, 2014.

[14] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated FPGAs: FPGA framework for convolutional neural networks," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pp. 265–268, 2016.

[15] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 25–34, 2017.

[16] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An FPGA framework for edge-centric graph processing," in *Proceedings of the International Conference on Computing Frontiers (CF)*, pp. 69–77, 2018.

[17] "SDSoC development environment." https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html. Accessed: 2017-11-12.

[18] "SoCEDS getting started." http://www.alterawiki.com/wiki/SoCEDSGettingStarted. Accessed: 2017-11-12.

[19] C. Liu, H. C. Ng, and H. K. H. So, "QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay," in *Proceedings of the International Conference on Field Programmable Technology (FPT)*, pp. 56–63, 2015.

[20] J. Fowers, J. Liu, and G. Stitt, "A framework for dynamic parallelization of FPGA-accelerated applications," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pp. 1–10, 2014.

[21] L. Kalms and D. Gohringer, "Exploration of OpenCL for FPGAs using SDAccel and comparison to GPUs and multicore CPUs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2017.

[22] "Intel FPGA SDK for OpenCL." https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html. Accessed: 2017-11-12.

[23] J. Yinger, E. Nurvitadhi, D. Capalija, A. Ling, D. Marr, S. Krishnan, D. Moss, and S. Subhaschandra, "Customizable FPGA OpenCL matrix multiply design template for deep neural networks," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pp. 259–262, 2017.

[24] A. C. Ling, U. Aydonat, S. O'Connell, D. Capalija, and G. R. Chiu, "Creating high performance applications with Intel's FPGA OpenCL SDK," in *Proceedings of the International Workshop on OpenCL (IWOCL)*, pp. 1–11, 2017.

[25] R. Domingo, R. Salvador, H. Fabelo, D. Madroal, S. Ortega, R. Lazcano, E. Jurez, G. Callic, and C. Sanz, "High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier," in *Proceedings of the International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, 2017.

[26] C. Desmouliers, E. Oruklu, S. Aslan, J. Saniie, and F. M. Vallina, "Image and video processing platform for field programmable gate arrays using a high-level synthesis," *IET Computers & Digital Techniques*, vol. 6, no. 6, pp. 414–425, 2012.

[27] H. Sahlbach, D. Thiele, and R. Ernst, "A system-level FPGA design methodology for video applications with weakly-programmable hardware components," *Journal of Real-Time Image Processing*, vol. 13, no. 2, pp. 291–309, 2017.

[28] P. Schumacher and P. Jha, "Fast and accurate resource estimation of RTL-based designs targeting FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 59–64, 2008.

[29] P. Bjureus, M. Millberg, and A. Jantsch, "FPGA resource and timing estimation from matlab execution traces," in *Proceedings of the International Symposium on Hardware/Software Codesign (CODES)*, pp. 31–36, 2002.

[30] V. Degalahal and T. Tuan, "Methodology for high level estimation of FPGA power consumption," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 657–660, 2005.

[31] P. A. Milder, M. Ahmad, J. C. Hoe, and M. Puschel, "Fast and accurate resource estimation of automatically generated custom DFT IP cores," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pp. 211–220, 2006.

[32] C. Shi, J. Hwang, S. McMillan, A. Root, and V. Singh, "A system level resource estimation tool for FPGAs," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, pp. 424–433, 2004.

[33] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An integrated partitioning and synthesis system for dynamically reconfigurable Multi-FPGA architectures," *Elsevier Parallel and Distributed Processing*, pp. 31–36, 2005.

[34] J. de Boer and M. Kalksma, "Choosing between optical flow algorithms for UAV position change measurement," in *Proceedings of the Student Colloquium Groningen Conference (SC@RUG)*, pp. 69–74, 2015.

[35] S. Golemati, J. S. Stoitsis, A. Gastounioti, A. C. Dimopoulos, V. Koropouli, and K. S. Nikita, "Comparison of block matching and differential methods for motion analysis of the carotid artery wall from ultrasound images," *IEEE Transactions on Information Technology in Biomedicine*, vol. 16, no. 5, pp. 852–858, 2012.

[36] J. T. Philip, B. Samuvel, K. Pradeesh, and N. K. Nimmi, "A comparative study of block matching and optical flow motion estimation algorithms," in *Proceedings of the International Conference on Magnetics (ICMM)*, pp. 1–6, 2014.

[37] Z. Lui and J. Luo, "Performance comparison of optical flow and block matching methods in shearing and rotating models," in *Proceedings of the International society for optics and photonics (SPIE)*, pp. 1–7, 2017.

[38] O. Palaz, H. F. Ugurdag, O. Ozkurt, B. Kertmen, and F. Donmez, "RImCom: Raster-order image compressor for embedded video applications," *Journal of Signal Processing Systems*, vol. 88, no. 3, pp. 149–165, 2017.

[39] Y. Huang, C. Y. Chen, C. H. Tsai, C. F. Shen, and L. G. Chen, "Survey on block matching motion estimation algorithms and architectures with new results," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 42, no. 3, pp. 297–320, 2006.

[40] B. D. Choi, J. W. Han, C. S. Kim, and S. J. Ko, "Motion-compensated frame interpolation using bilateral motion estimation and adaptive overlapped block motion compensation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 4, pp. 407–416, 2007.

[41] C. H. Cheung and L. M. Po, "Novel cross-diamond-hexagonal search algorithms for fast block motion estimation," *IEEE Transactions on Multimedia*, vol. 7, no. 1, pp. 16–22, 2005.

[42] "Middlebury optical flow benchmark database." http://vision.middlebury.edu/flow/eval/results/results-e1.php. Accessed: 2017-11-12.

[43] "The kitti vision benchmark suite." http://www.cvlibs.net/datasets/kitti/. Accessed: 2017-04-1.

[44] M. Werlberger, W. Trobin, T. Pock, A. Wedel, D. Cremers, and H. Bischof, "Anisotropic Huber-L1 optical flow," in *Proceedings of the British Machine Vision Conference (BMVC)*, pp. 1–11, 2009.

[45] D. Büyükaydın and T. Akgün, "GPU implementation of an anisotropic Huber-L1 dense optical flow algorithm using OpenCL," in *Proceedings of International*

Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp. 326–331, 2015.

[46] V. E. Levent, "FPGA based hardware platform for video processing," Master's thesis, Yıldız Technical University, 2015. YÖK Thesis No: 406560.

[47] C. Hwang, Y. Hsu, and Y. Lin, "Scheduling for functional pipelining and loop winding," in *Proceedings of the Design Automation Conference (DAC)*, pp. 764–769, 1991.

[48] K. K. Parhi, "VLSI digital signal processing systems: Design and implementation," *Wiley*, 1999.

[49] A. E. Guzel, V. Levent, M. Tosun, M. A. Ozkan, T. Akgun, C. Erbas, and H. F. Ugurdag, "Using high-level synthesis for rapid design of video processing pipes," in *Proceedings of the East-West Design & Test Symposium (EWDTS)*, pp. 1–4, 2016.

[50] "Vivado design suite user guide: High-level synthesis (ug902 v2014.3)." http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/ug902-vivado-high-level-synthesis.pdf. Accessed: 2017-11-12.

[51] M. Buyukmihci, V. Levent, A. Guzel, O. Ates, M. Tosun, T. Akgun, C. Erbas, S. Goren, and H. Ugurdag, "Output domain downscaler," in *Proceedings of the International Symposium on Computer and Information Sciences (ISCIS)*, pp. 262–269, 2016.

[52] M. Jacobsen, D. Richmond, N. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 4, pp. 1–23, 2015.

[53] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, 2014.

[54] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using Xilinx Zynq," in *Proceedings of the FPGA World Conference (FPGAWORLD)*, pp. 1–8, 2013.

[55] K. Vissers, S. Neuendorffer, and J. Noguera, "Building real-time HDTV applications in FPGAs using processors, AXI interfaces and high level synthesis tools," in *Proceedings of the Design, Automation & Test in Europe Conference (DATE)*, pp. 1–3, 2011.

[56] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, "An efficient and flexible host-FPGA PCIe communication library," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–6, 2014.

[57] "AMBA AXI4 interface protocol - xilinx." https://www.xilinx.com/products/intellectual-property/axi.html. Accessed: 2018-06-1.

[58] E. Salminen, A. Kulmala, and T. Hamalainen, "HIBI-based multiprocessor SoC on FPGA," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 3351–3354, 2005.

[59] S. Grinberg and S. Weiss, "Investigation of Transactional Memory Using FPGAs," in *Proceedings of the IEEE Convention of Electrical & Electronics Engineers (IEEEI)*, pp. 119–122, 2006.

[60] J. A. Kalomirosa and J. Lygourasb, "Design and evaluation of a hardware/software FPGA-based system for fast image processing," *Elsevier Microprocessors and Microsystems*, vol. 32, no. 2, pp. 95–106, 2008.

[61] X. Wang and S. G. Ziavras, "Parallel direct solution of linear equations on FPGA-based machines," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–8, 2003.

[62] "Avalon interface specifications." https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf. Accessed: 2018-06-1.

[63] B. Krill, A. Ahmad, A. Amira, and H. Rabah, "An efficient FPGA-based dynamic partial reconfiguration design flow and environment for image and signal processing IP cores," *Elsevier Signal Processing: Image Communication*, vol. 25, no. 5, pp. 377–387, 2010.

[64] K. Lewandowski, R. Graczyk, K. T. Pozniak, and R. S. Romaniuk, "FPGA based PCI mezzanine card with digital interfaces," in *Proceedings of the Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments (SPIE)*, pp. 1–9, 2007.

[65] U. Kretzschmar, A. Astarloa, J. Lzaro, M. Garay, and J. D. Ser, "Robustness of different TMR granularities in shared Wishbone architectures on SRAM FPGA," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (RECONFIG)*, pp. 1–6, 2012.

[66] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, "A design methodology for communication infrastructures on partially reconfigurable FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 331–338, 2007.

[67] "SoC interconnection: Wishbone." https://opencores.org/howto/wishbone. Accessed: 2018-06-1.

[68] "Eureqa data mining tool." https://www.nutonian.com/products/eureqa/. Accessed: 2018-06-1.

[69] E. Reinhard, M. Ashikhmin, B. Gooch, and P. Shirley, “Color transfer between images,” *IEEE Computer Graphics and Applications*, vol. 21, no. 5, pp. 34–41, 2001.

[70] A. Toet, “Natural colour mapping for multiband nightvision imagery,” *Elsevier Information Fusion*, vol. 4, no. 3, pp. 155–166, 2003.

[71] I. Janik, Q. Tang, and M. Khalid, “An overview of Altera SDK for OpenCL: A user perspective,” in *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 559–564, 2015.

VITA

Vecdi Emre Levent received the BS degree in Computer Engineering from Arel University, İstanbul, Turkey, in 2013, and the MS degree in Computer Engineering from Yıldız Technical University, İstanbul, Turkey, in 2015. He joined nEMESysLab Research Group as a research assistant working toward his doctorate under the supervision of Assoc. Prof. H. Fatih Uğurdağ at Özyeğin University, İstanbul, Turkey, in 2014. He has served as a reviewer for Elseiver Digital Signal Processing Journal and VLSI-SoC Conference. During his PhD studies, he has contributed to several research projects. These are ALMARVI (Algorithms, Design Methods, and Many-core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing, Artemis 2013 GA 621439, an EU project), M-RIVA (Methodology Development for Real-time Implementation of Video Algorithms on FPGAs, TÜBİTAK 1001 - 114E343) and VLC (Innovative Optical Wireless Communication Technologies for 5G and Beyond, TÜBİTAK 1003 - 215E311). He has so far published three journals and ten conference papers. His research interests are FPGA design and automation, video processing frameworks, interfaces and video processing hardware design.